Inverted Index

Jorge Lorenzo Lorenzo, Owen Urdaneta Morales, Luis Perera Pérez,
Juan López de Hierro, Carlos Jose Moreno Vega, Toffalori Christian

October 20, 2024

Abstract

The methods used to store information, such as the structure or the order in which data is stored, are crucial when it comes to data access and maintaining coherence between different datasets. While these two aspects do not always align, the ideal approach is to strike a balance between them. In this project, multiple books are downloaded from the Project Gutenberg website using a Crawler, with the aim of processing the words and indexing them in two distinct structures to compare their performance: one based on a dictionary structure, and the other utilizing a local network NoSQL database, MongoDB.

1 Introduction

Storing books can be a highly resource-intensive task, especially when the primary goal is to process the words for a user. While real-time reading is possible, if the aim is to create a service-oriented project, the wait time for the user can be substantial. This issue becomes more pronounced as the number of books and their sizes increase. Therefore, it is optimal to preprocess the information to minimize response times for potential clients.

This project, developed in pyhton, applies the same concept of preprocessing words to ensure fast access. This is achieved through two indexing methods: one based on a matrix structure, and the other leveraging a NoSQL database, MongoDB. The process begins by downloading the books from the web, and then, through the execution of these two indexers, we store the words along with relevant information to locate them, such as which books they appear in and their position within those books. The ultimate goal of the program is to provide a search engine for the user.

2 Method

2.1 Crawler

The Crawler is the part of the code responsible for downloading the data to be used in the project, as well as preprocessing it. In this case, that data consists of books from the Gutenberg Project.

Full Code

```
import requests
import os
import csv
from bs4 import BeautifulSoup
import re

# Create directories to store books and metadata if they do not exist
os.makedirs('datalake/books', exist_ok=True)
os.makedirs('datalake', exist_ok=True)

BOOKS_TO_DOWNLOAD = 5
METADATA_CSV_FILE = 'datalake/metadata.csv'

def execute():
    """Main function to download eBooks and their metadata."""
```

```
    id_limit_book_downloaded = check_limit_id_books()
    id_book = id_limit_book_downloaded + 1
    count_downloaded_books = 0

    # Create the CSV file and write the header if it doesn't exist
    if not os.path.exists(METADATA_CSV_FILE):
        with open(METADATA_CSV_FILE, 'w', newline='', encoding='utf-8') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(["ID", "Title", "Author", "Release Date", "Most Recently Updated", "Language"])

    while count_downloaded_books < BOOKS_TO_DOWNLOAD:
        if not check_book_exist(id_book):
            title = download_ebook(id_book)
            if title:  # Proceed only if the download was successful
                metadata = obtain_metadata(id_book, title)
                if metadata:  # Only process if metadata was retrieved successfully
                    append_metadata_to_csv(metadata)
                processing_book(id_book, title)
                os.remove(f"datalake/books/{id_book} {title}.txt")
                count_downloaded_books += 1
        id_book += 1

    # Save the ID and overwrite only the first line of the lastBookId.txt file
    with open("resources/lastBookId.txt", "r+") as file:
        lines = file.readlines()  # Read all lines from the file
        if lines:
            lines[0] = str(id_book - 1) + "\n"  # Update only the first line
        else:
            lines.append(str(id_book - 1) + "\n")  # If empty, add the first line

        # Go back to the start of the file and overwrite lines
        file.seek(0)
        file.writelines(lines)
        file.truncate()  # Remove any leftover content if lines were shortened

def download_ebook(id_book):
    """Download the eBook by its ID and return its title."""
    try:
        book_url = f"https://www.gutenberg.org/ebooks/{id_book}"
        response = requests.get(book_url)  # Make a request to the book's URL (ID)
        if response.status_code == 200:
            soup = BeautifulSoup(response.text, 'html.parser')
            title_tag = soup.find('h1')  # Find the h1 tag for the title
            title = title_tag.get_text() if title_tag else f"Unknown Title id {id_book}"
            text_link = soup.find('a', href=True, string="Plain Text UTF-8")  # Find the link to the plain text file
            if text_link:
                link_txt = 'https://www.gutenberg.org' + text_link['href']
                response_txt = requests.get(link_txt)
                if response_txt.status_code == 200:
                    title = re.sub(r'[^\w\s]', '', title)  # Clean title
                    with open(f"datalake/books/{id_book} {title}.txt", 'w', encoding='utf-8') as file:
                        file.write(response_txt.text)  # Write the book content to the text file
                        print(f"Book with ID {id_book} has been downloaded.")
                    return title
                else:
                    print(f"Error accessing the text file of the book with ID {id_book}: {response_txt.status_code}")
            else:
                print(f"The book with ID {id_book} does not have a text file available.")
        else:
            print(f"Error accessing the book with ID {id_book}: {response.status_code}")
    except Exception as e:
        print(f"Error accessing the book with ID {id_book}: {e}")

```

```python
def obtain_metadata(id_book, title):
    """Obtain metadata from the downloaded eBook."""
    try:
        with open(f"datalake/books/{id_book} {title}.txt", 'r', encoding='utf-8') as file:
            text = file.read()
            metadata = {
                "ID": id_book,
                "Title": re.search(r"Title: (.+)", text).group(1) if re.search(r"Title: (.+)", text) else "Unknown Title",
                "Author": re.search(r"Author: (.+)", text).group(1) if re.search(r"Author: (.+)", text) else "Unknown Author",
                "Release Date": re.search(r"Release Date: (.+)", text).group(1) if re.search(r"Release Date: (.+)", text) else "Unknown",
                "Most Recently Updated": re.search(r"Most recently updated: (.+)", text).group(1) if re.search(r"Most recently updated: (.+)", text) else "Unknown",
                "Language": re.search(r"Language: (.+)", text).group(1) if re.search(r"Language: (.+)", text) else "Unknown",
            }
            return metadata  # Return the metadata for further processing
    except Exception as e:
        print(f"Error accessing the book with ID {id_book}: {e}")
        return None  # Return None if an error occurred

def append_metadata_to_csv(metadata):
    """Append the metadata of the book to the CSV file."""
    with open(METADATA_CSV_FILE, 'a', newline='', encoding='utf-8') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow([metadata["ID"], metadata["Title"], metadata["Author"], metadata["Release Date"],
                         metadata["Most Recently Updated"], metadata["Language"]])

def processing_book(id_book, title):
    """Process the downloaded eBook to extract the main content."""
    try:
        with open(f"datalake/books/{id_book} {title}.txt", 'r', encoding='utf-8') as file:
            text = file.read()
            content_start = re.search(r"\*\*\* START OF THE PROJECT GUTENBERG EBOOK .+ \*\*\*", text)
            content_end = re.search(r"\*\*\* END OF THE PROJECT GUTENBERG EBOOK .+ \*\*\*", text)

            if content_start and content_end:
                raw_content = text[content_start.end():content_end.start()].strip()
                lines = raw_content.splitlines()

                # Process the lines into paragraphs
                paragraphs = []
                current_paragraph = []
                empty_line_count = 0

                for line in lines:
                    stripped_line = line.rstrip()
                    if stripped_line:
                        if empty_line_count > 2:
                            paragraphs.append("")  # Add a line break for paragraph separation
                        current_paragraph.append(stripped_line)
                        empty_line_count = 0
                    else:
                        if current_paragraph:
                            paragraphs.append(" ".join(current_paragraph))
                            current_paragraph = []
                        empty_line_count += 1

                if current_paragraph:
                    paragraphs.append(" ".join(current_paragraph))

                final_content = "\n".join(paragraphs)
```

```
                # Save processed content to a new file
                with open(f"datalake/books/{id_book}.txt", 'w', encoding='utf-8') as file:
                    file.write(final_content)

    except Exception as e:
        print(f"Error accessing the book with ID {id_book}: {e}")

def check_book_exist(id_book):
    """Check if the book with the given ID already exists in the download directory."""
    for filename in os.listdir('datalake/books/'):
        if filename.startswith(f"{id_book}"):
            print(f"The book with ID {id_book} has already been downloaded.")
            return True
    return False

def check_limit_id_books():
    """Check the last downloaded book ID from the lastBookId.txt file."""
    if os.path.exists("resources/lastBookId.txt"):
        with open("resources/lastBookId.txt", "r") as file:
            first_line = file.readline().strip()  # Read the first line and strip whitespace
            return int(first_line) if first_line.isdigit() else 0  # Check if it's a number
    else:
        with open("resources/lastBookId.txt", "w") as file:
            file.write("0\n")
        return 0

if __name__ == "__main__":
    execute()
```

Listing 1: Crawler for downloading books from Gutenberg

2.1.1 Code Explanation

Directory Setup The crawler first ensures that the necessary directories for storing books and metadata exist. This is achieved by using the `os.makedirs()` function, which creates the directories `datalake/books` for storing downloaded books, and `datalake` for storing the corresponding metadata if they do not already exist.

It is important to note that for the crawler to function properly, a directory named `resources` must exist, containing a file called `lastBookId.txt`. This file is essential as it keeps track of the last downloaded book's ID, allowing the crawler to resume downloading from the next available book. Without this file, the system would not be able to determine which book to start downloading next.

Main Execution Flow The core functionality is handled by the `execute()` function, which controls the downloading and processing of books. It first checks the ID of the last downloaded book by calling the `check_limit_id_books()` function, which reads from the `lastBookId.txt` file. Starting from the next book ID, the crawler attempts to download and process up to 5 books, as defined by the `BOOKS_TO_DOWNLOAD` constant.

If a book has already been downloaded, this is verified by the `check_book_exist()` function, which looks for the book in the `datalake/books` directory. If the book has not yet been downloaded, it proceeds to the `download_ebook()` function, which attempts to download the book and save it locally.

Once the book is successfully downloaded, the `obtain_metadata()` function retrieves relevant metadata, such as the title, author, release date, and language, by parsing the book's content. This metadata is then saved in a CSV file using the `append_metadata_to_csv()` function.

Afterward, the book content is processed by the `processing_book()` function to extract only the main text, removing any headers, footers, and irrelevant content. Finally, the processed book file is deleted to free up space, and the loop continues until 5 books have been successfully downloaded and processed.

Downloading Books The `download_ebook()` function is responsible for downloading a book from the Gutenberg Project by its ID. It constructs the URL for the book and retrieves the HTML content of the page. Using BeautifulSoup, the crawler identifies the title and the link to the plain text version of the book. If the plain text version is available, the book's content is downloaded and saved locally. The title is also cleaned to remove any special characters to ensure a valid file name.

If the book does not have a plain text version, or if any error occurs during the download process, appropriate error messages are logged.

Extracting Metadata Once a book is downloaded, the `obtain_metadata()` function extracts key metadata from the text, such as the title, author, release date, and language. This is done using regular expressions, which search for specific metadata tags within the text. If the metadata is found, it is stored in a dictionary and returned to be saved in the metadata CSV file.

If any error occurs while accessing or extracting the metadata, the function logs an error message and returns `None` to avoid interrupting the workflow.

Processing Book Content After the book is downloaded and the metadata is extracted, the `processing_book()` function processes the content to isolate the main text of the book. This involves stripping out the header and footer content added by the Gutenberg Project, which is enclosed between specific tags ("`START OF THE PROJECT GUTENBERG EBOOK`" and "`END OF THE PROJECT GUTENBERG EBOOK`").

The main content is then processed into paragraphs, and excessive line breaks are removed to ensure a cleaner format. The processed content is saved into a new file, which only contains the body of the book.

Metadata Storage All the collected metadata, such as the book's ID, title, author, release date, and language, is stored in a CSV file named `metadata.csv`. This allows easy reference and provides a structured way to access the information later. The function `append_metadata_to_csv()` ensures that each downloaded book's metadata is appended to the CSV file.

Book ID Management To prevent downloading the same books multiple times, the crawler keeps track of the last downloaded book's ID in the file `lastBookId.txt`. The `check_limit_id_books()` function reads this file to retrieve the ID of the last downloaded book, while `execute()` updates the file each time new books are downloaded. This process ensures that downloads can resume seamlessly in subsequent executions.

Error Handling and Logging The crawler includes comprehensive error handling throughout its functions to ensure robustness. If any error occurs while downloading a book, extracting metadata, or processing the content, the error is logged, and the crawler continues with the next book. This ensures that one failed download does not interrupt the entire process.

2.2 Inverted Indexer

In this work, an inverse indexer has been implemented to process a collection of books and store each word in individual JSON files. These files are organized into directories labeled from A to Z, where each folder contains words that begin with that letter. The indexer processes books that have not been previously indexed, ensuring no duplicate processing, and updates the corresponding file with the number of occurrences of the word, along with the line and position within the line.

The indexing process ensures that each word from a book is stored in a dedicated JSON file. For example, if the word "book" appears in multiple books, all occurrences are recorded in a single `book.json` file, located in the directory corresponding to the letter B. Inside this file, the exact position of each appearance of the word is stored, indicating the line number and the word's position in that line.

To prevent repeated processing of books, the system uses a tracking file, `resources/lastBookId.txt`, where the ID of the last indexed book is stored. This way, every time the program is executed, the indexer starts with the next book, ensuring that only new books are processed. At the end of the process, the tracking file is updated with the ID of the most recently indexed book.

A key aspect of this indexer is the lemmatization of words. Using the `simplemma` library, words are normalized so that morphological variations like plurals or verb conjugations are stored under the same root. This means that words like "books" and "book" are grouped into the same JSON file, facilitating more efficient searches. Additionally, the use of `unidecode` normalizes words by removing special characters and accents, ensuring greater consistency in the stored data.

The indexer also tracks the exact line number and the position within each line where a word appears. This functionality provides precise context about the location of each word in the indexed books, which is essential for conducting detailed reverse searches.

Full Code

```
import os
import json
import re
from unidecode import unidecode
from simplemma import lemmatize

# Create necessary directories
OUTPUT_DIR = 'datamart/reverse_indexes/'
GLOBAL_INDEX_FILE = 'datamart/global_index.txt'

def create_directory(path):
    os.makedirs(path, exist_ok=True)

def read_lines(file_path):
    try:
        with open(file_path, 'r') as file:
            return file.readlines()
    except FileNotFoundError:
        print(f"The file '{file_path}' does not exist.")
        return []

def write_lines(file_path, lines):
    with open(file_path, 'w') as file:
        file.writelines(lines)

def read_json(file_path):
    if os.path.exists(file_path):
        with open(file_path, 'r', encoding='utf-8') as json_file:
            return json.load(json_file)
    return None

def write_json(file_path, data):
    with open(file_path, 'w', encoding='utf-8') as json_file:
        json.dump(data, json_file, ensure_ascii=False, indent=4)

def read_global_index():
    global_index_dict = {}
    if os.path.exists(GLOBAL_INDEX_FILE):
        lines = read_lines(GLOBAL_INDEX_FILE)
        for line in lines:
            line = line.strip()
            if line:
                parts = line.split(" : ")
                if len(parts) == 2:
                    word, books = parts
                    global_index_dict[word] = books.split(", ")
    return global_index_dict

def write_global_index(global_index_dict):
    lines = [f"{word} : {', '.join(books)}\n" for word, books in global_index_dict.
        items()]
    write_lines(GLOBAL_INDEX_FILE, lines)

def clean_word(word):
    word = re.sub(r'^_.*|_.*_|_.*$|[\d]+|[^\w\s]', '', word)
    return unidecode(word).strip('_')

```

```python
def lemm_add(dictionary, word, lang='en'):
    lemm = lemmatize(word, lang)
    if lemm != word:
        if lemm not in dictionary:
            dictionary[lemm] = {'allocations': {}, 'total': 0}
        if "allocations" in dictionary[word]:
            pop = dictionary.pop(word)
            for book_key, new_info in pop['allocations'].items():
                if book_key in dictionary[lemm]['allocations']:
                    existing_positions = set(tuple(pos) for pos in dictionary[lemm]['allocations'][book_key]['position'])
                    existing_positions.update(tuple(pos) for pos in new_info['position'])
                    dictionary[lemm]['allocations'][book_key]['position'] = list(existing_positions)
                    dictionary[lemm]['allocations'][book_key]['times'] = len(existing_positions)
                else:
                    dictionary[lemm]['allocations'][book_key] = new_info
            dictionary[lemm]['total'] += pop['total']

def index_book(book_id):
    dictionary = {}
    book_file_path = f"datalake/books/{book_id}.txt"

    with open(book_file_path, encoding='utf-8') as fp:
        for line_number, line in enumerate(fp, start=1):
            words = [clean_word(word) for word in line.lower().split()]
            for position, word in enumerate(words):
                if word:
                    if word not in dictionary:
                        dictionary[word] = {'allocations': {}, 'total': 0}
                    book_key = f"BookID_{book_id}"

                    if book_key not in dictionary[word]['allocations']:
                        dictionary[word]['allocations'][book_key] = {'times': 0, 'position': []}

                    pos_tuple = (line_number, position + 1)
                    if pos_tuple not in dictionary[word]['allocations'][book_key]['position']:
                        dictionary[word]['allocations'][book_key]['position'].append(pos_tuple)
                        dictionary[word]['allocations'][book_key]['times'] += 1

                    dictionary[word]['total'] += 1

    for word in list(dictionary.keys()):
        lemm_add(dictionary, word)

    save_index(dictionary, book_id)

def save_index(dictionary, book_id):
    reserved_words = ['con', 'prn', 'aux', 'nul', 'com1', 'com2', 'com3',
                      'com4', 'com5', 'com6', 'com7', 'com8', 'com9',
                      'lpt1', 'lpt2', 'lpt3', 'lpt4', 'lpt5', 'lpt6',
                      'lpt7', 'lpt8', 'lpt9']

    for word, data in dictionary.items():
        first_letter = word[0].lower()
        if word.lower() in reserved_words:
            print(f"Word '{word}' skipped because it is a reserved name.")
            continue

        create_directory(f"datamart/reverse_indexes/{first_letter}")
        json_file_path = f"datamart/reverse_indexes/{first_letter}/{word}.json"

        existing_data = read_json(json_file_path) or {
            'word': word,
            'allocations': data['allocations'],
            'total': 0
```

```
        }

        if existing_data:
            existing_allocations = existing_data['allocations']
            for book_key, new_info in data['allocations'].items():
                if book_key in existing_allocations:
                    existing_positions = set(tuple(pos) for pos in
                        existing_allocations[book_key]['position'])
                    existing_positions.update(tuple(pos) for pos in new_info['position
                        '])
                    existing_allocations[book_key]['position'] = list(
                        existing_positions)
                    existing_allocations[book_key]['times'] = len(existing_positions)
                else:
                    existing_allocations[book_key] = new_info

            existing_data['total'] = sum(info['times'] for info in
                existing_allocations.values())

        write_json(json_file_path, existing_data)

    update_global_index(dictionary, book_id)

def update_global_index(dictionary, book_id):
    global_index_dict = read_global_index()

    for word in dictionary.keys():
        if word in global_index_dict:
            if f"BookID_{book_id}" not in global_index_dict[word]:
                global_index_dict[word].append(f"BookID_{book_id}")
        else:
            global_index_dict[word] = [f"BookID_{book_id}"]

    write_global_index(global_index_dict)

def read_numbers_from_file(file_path):
    try:
        lines = read_lines(file_path)
        first_number = int(lines[0].strip())
        second_number = int(lines[1].strip()) if len(lines) > 1 else 1

        write_lines(file_path, [f"{first_number}\n", f"{first_number}\n"])
        return first_number, second_number
    except ValueError:
        print("The file content is not a valid integer.")

def execute():
    n, m = read_numbers_from_file("resources/lastBookId.txt")
    for i in range(m, n + 1):
        index_book(i)
        print(f"Book with ID {i} has been indexed.")

if __name__ == "__main__":
    execute()
```

Listing 2: Inverted indexer Code

2.2.1 Code Explanation

Directory and File Handling The code begins by defining helper functions for file operations like reading and writing text files (`read_lines()` and `write_lines()`), and working with JSON files (`read_json()` and `write_json()`). These are critical for interacting with book content and saving the indexed word information.

Word Normalization The `clean_word()` function cleans individual words by removing special characters, numbers, and diacritics. This ensures that words are stored in a uniform format. The `lemm_add()` function is responsible for adding lemmatized words to the dictionary, grouping together different word forms (e.g., "running" and "run") under a single entry.

Book Indexing The core functionality of indexing each book is handled by `index_book()`. It processes the book file line by line, identifies the words in each line, and tracks their position within the book. For each word, it creates an "allocation" object that stores the positions where the word appears in the book.

Saving the Index and Reserved Words After processing, the `save_index()` function saves the word allocations and positions in a JSON file. Words that are reserved in the system (like "con", "aux", etc.) are skipped to avoid system conflicts. The file path for each word is based on its first letter, which helps organize the data for fast access.

Global Index Update The `update_global_index()` function updates a global index file that keeps track of which books contain each word. This ensures that the system can quickly identify the presence of words across all books in the system.

Tracking the Last Book The `read_numbers_from_file()` function reads the last processed book's ID from `lastBookId.txt`, and the `execute()` function ensures the indexer resumes from the correct point, allowing new books to be indexed without reprocessing old ones.

2.3 MongoDB Indexer

In order to use MongoDB, it is necessary to carry out some preliminary installations. First, in the Python environment where the project is being developed, execute the following installation command in the terminal:

```
pip install pymongo
```

This is necessary to perform MongoDB operations through Python code. Afterward, you need to install the MongoDB services by downloading them from their official website:

https://www.mongodb.com/try/download/community

Since this project runs a local database, it is required to install the Community Server version of MongoDB. It is recommended to download and install the '.msi' file, as it provides a guided setup. Perform a full installation and accept the installation of MongoDB Compass, which is the GUI used to visually inspect the database content.

At this point, it is important to highlight that these steps have been executed on a Windows operating system, as it involves interacting with directories and system variables. After the installation is complete, navigate to your system drive and create a new folder called 'data' with another folder inside named 'db'. This directory is where MongoDB will store the data from the connections.

Once these steps are completed, go to the Program Files directory on your system and find the MongoDB directory. Then, proceed to enter the 'Server', '8.0' (or whichever version you choose to install; in this project, version 8.0 was used), and 'bin' directories. Copy the directory path. Afterward, access the system's environment variables and locate the 'Path' under the user variables. Select it and add a new entry with the copied directory path.

After doing this, you should be able to launch MongoDB by running the command 'mongod' from the system terminal, which will start a local network database that will continue to listen as long as it remains running.

Finally, using MongoDB Compass, you can connect to the database by adding the connection string 'mongodb://localhost:27017' to visually manage the databases in the connection.

Full Code

```
import os
from pymongo import MongoClient
import re
from unidecode import unidecode
from simplemma import lemmatize

# Configure the MongoDB connection
client = MongoClient('mongodb://localhost:27017/')
db = client['SEARCH_ENGINE']  # Database name
```

```
words_collection = db['WORDS']  # Collection name for words
processed_books_collection = db['PROCESSED_BOOKS']  # Collection name for processed
    books

# Predefined pre-path
PRE_PATH = "datalake/books/"

# Clean and lemmatize words
def clean_word(word):
    word = re.sub(r'^_.*|_.*_|_.*$|[\d]+|[^\w\s]', '', word)
    word = unidecode(word).strip('_').strip()
    return word if word else None

def lemm_add(dictionary, word, lang='en'):
    if word:
        lemm = lemmatize(word, lang)
        if lemm != word:
            if lemm not in dictionary:
                dictionary[lemm] = {'allocations': {}, 'total': 0}
            if "allocations" in dictionary[word]:
                pop = dictionary.pop(word)
                for book_key, new_info in pop['allocations'].items():
                    if book_key in dictionary[lemm]['allocations']:
                        existing_positions = set(tuple(pos) for pos in dictionary[lemm
                            ]['allocations'][book_key]['position'])
                        existing_positions.update(tuple(pos) for pos in new_info['
                            position'])
                        dictionary[lemm]['allocations'][book_key]['position'] = list(
                            existing_positions)
                        dictionary[lemm]['allocations'][book_key]['times'] = len(
                            existing_positions)
                    else:
                        dictionary[lemm]['allocations'][book_key] = new_info
                dictionary[lemm]['total'] += pop['total']

def process_book(file_name):
    file_path = os.path.join(PRE_PATH, file_name)
    book_name = file_name

    if processed_books_collection.find_one({'book_name': book_name}):
        print(f"The book '{book_name}' has already been processed.")
        return

    with open(file_path, 'r', encoding='utf-8') as file:
        text = file.read()

    word_occurrences = {}
    paragraphs = text.split("\n\n")

    for i, paragraph in enumerate(paragraphs):
        words = [clean_word(word) for word in paragraph.lower().split()]
        words = [word for word in words if word]

        for word in set(words):
            word_count = words.count(word)

            if word not in word_occurrences:
                word_occurrences[word] = {'count': 0, 'books': {}}
            word_occurrences[word]['count'] += word_count

            if book_name not in word_occurrences[word]['books']:
                word_occurrences[word]['books'][book_name] = {}

            word_occurrences[word]['books'][book_name][f"paragraph {i}"] = word_count

    for word in list(word_occurrences.keys()):
        lemm_add(word_occurrences, word)

        existing_data = words_collection.find_one({'word': word})

        if existing_data:
```

```
            new_count = existing_data['count'] + word_occurrences[word]['count']

            for book, paragraphs in word_occurrences[word]['books'].items():
                if book in existing_data['books']:
                    for paragraph, count in paragraphs.items():
                        if paragraph not in existing_data['books'][book]:
                            existing_data['books'][book][paragraph] = count
                        else:
                            existing_data['books'][book][paragraph] += count
                else:
                    existing_data['books'][book] = paragraphs

            words_collection.update_one(
                {'word': word},
                {'$set': {'count': new_count, 'books': existing_data['books']}}
            )
            print(f"Updated {word}: {new_count} occurrences across books.")
        else:
            words_collection.insert_one({
                'word': word,
                'count': word_occurrences[word]['count'],
                'books': word_occurrences[word]['books']
            })
            print(f"Inserted {word}: {word_occurrences[word]['count']} occurrences.")

    processed_books_collection.insert_one({'book_name': book_name})
    print(f"The book '{book_name}' has been processed and recorded.")

def execute():
    choice = input("Do you want to process a (1) single file or (2) all files in the
        folder? (1/2): ")
    if choice == '1':
        file_name = input("Enter the name of the file to process (without the path): "
            )
        process_book(file_name)
    elif choice == '2':
        folder_path = PRE_PATH
        for filename in os.listdir(folder_path):
            if filename.endswith('.txt'):
                process_book(filename)
                print(f"Processed file: '{filename}'")
    else:
        print("Invalid choice. Please try again.")

    print("Data saved in the MongoDB database.")

if __name__ == "__main__":
    execute()
```

Listing 3: MongoDB Indexer Code

2.3.1 Code Explanation

The MongoDB indexer script serves to process the text files of books stored locally and to store the relevant word data in a MongoDB database. The process includes cleaning, lemmatizing, and organizing words to optimize their searchability in a future retrieval system. Below is an explanation of the major components of the script:

MongoDB Connection Setup: The script begins by establishing a connection to a local MongoDB instance using the `MongoClient` from the `pymongo` library. It connects to the `SEARCH_ENGINE` database and creates two collections: `WORDS` and `PROCESSED_BOOKS`. Importantly, MongoDB does not require the database or collections to be created in advance. They are automatically generated when the first write operation occurs.

Cleaning and Lemmatizing Words: The `clean_word()` function is used to clean the text by removing unwanted characters, digits, and punctuation. The `lemm_add()` function then lemmatizes

the words using the `simplemma` library. This process normalizes the words, reducing them to their base forms, and ensures that similar words are grouped together.

Processing a Book: The `process_book()` function handles the main logic for processing the text of each book. It reads the text file, splits it into paragraphs, and extracts words from each paragraph. Each word's occurrence is counted and stored, along with its paragraph and the total count across the book. The processed word occurrences are then either inserted into the MongoDB collection or updated if they already exist.

Lemmatization and Insertion in MongoDB: After the words are cleaned and lemmatized, they are checked against the existing entries in the MongoDB `WORDS` collection. If the word already exists, its count and paragraph occurrences are updated. If not, it is inserted as a new entry. MongoDB automatically manages the insertion and updating of documents in the collections.

Storage Format: The word occurrences are stored in MongoDB using a nested document structure. Each word is a document in the `WORDS` collection, and the associated metadata for each word is stored in a dictionary format. The structure of each word document is as follows:

```
{
    "word": "example",  # The word itself
    "count": 15,  # Total occurrences across all books
    "books": {
        "book_name_1": {
            "paragraph 1": 3,  # Occurrences in paragraph 1
            "paragraph 4": 2   # Occurrences in paragraph 4
        },
        "book_name_2": {
            "paragraph 2": 1
        }
    }
}
```

This structure allows for efficient querying of specific words, as well as tracking in which books and paragraphs each word appears. MongoDB's flexible schema makes it ideal for this kind of dynamic storage, where the number of books or paragraphs can vary from word to word. The `PROCESSED_BOOKS` collection keeps track of books that have already been processed to avoid redundant operations.

Execution Flow: The `execute()` function allows the user to either process a single file or all files in the folder. Once processed, the data is stored in MongoDB, allowing for future queries and retrievals to be executed efficiently.

2.4 Inverted Indexer API

This work implements a terminal-based API that allows users to search for words within a collection of books that have been previously indexed. The user enters a word into the terminal, and the system looks up that word in the corresponding JSON files, organized by the first letter of the word. If the word is found in the index, the system returns detailed information about its occurrences, including the book title, the exact line where the word appears, and its position within the line. Furthermore, the system considers lemmatization, so variations of the word, such as plurals or conjugations, are treated as the same base word. It also handles case-insensitive searches, ensuring that both uppercase and lowercase versions of the word are correctly matched. Below is a detailed explanation of the process and the key parts of the code that make this functionality possible.

Full Code

```python
import csv
import json
from unidecode import unidecode
from simplemma import lemmatize

# Function to obtain the book title from the metadata.csv file
def get_book_title(book_id):
    with open('datalake/metadata.csv', mode='r', encoding='utf-8') as file:
        reader = csv.DictReader(file)
        for row in reader:
            if row['ID'] == str(book_id):
                return row['Title']
    return f"BookID_{book_id}"  # If the title is not found, return the BookID.

# Function to get word occurrences from the corresponding JSON file
def get_word_occurrences_in_books(word):
    try:
        with open(f"datamart/reverse_indexes/{word[0].lower()}/{word}.json", 'r',
            encoding='utf-8') as json_file:
            existing_data = json.load(json_file)
            return existing_data['allocations']  # Returns a dictionary with books and
                positions
    except FileNotFoundError:
        raise ValueError(f"The word '{word}' has not been indexed yet.")

# Function to read paragraphs and highlight the word where it appears
def read_paragraphs(book_id, occurrences, search_word):
    with open(f"datalake/books/{book_id}.txt", encoding='utf-8') as fp:
        result = []
        current_occurrence = 0
        occurrences.sort(key=lambda x: x[0])  # Ensure positions are in order
        for i, line in enumerate(fp, start=1):
            # If the current line is where the word appears
            while current_occurrence < len(occurrences) and i == occurrences[
                current_occurrence][0]:
                # Highlight the word in purple using ANSI escape codes
                highlighted_line = line.replace(search_word, f"\033[35m{search_word
                    }\033[0m")
                result.append(f"Line {i}: {highlighted_line.strip()} (Position {
                    occurrences[current_occurrence][1]})")
                current_occurrence += 1
                if current_occurrence >= len(occurrences):
                    break
    return result

# Main function to search for a word across all books
def search_word_across_books(word, lang='en'):
    word = unidecode(word.lower()).strip()  # Clean the word
    word = lemmatize(word, lang)  # Lemmatize the word

    try:
        occurrences_by_book = get_word_occurrences_in_books(word)  # Retrieve books
            and positions
        if occurrences_by_book:
            print(f"The word '{word}' appears in the following books:")
            for book_id, info in occurrences_by_book.items():
                title = get_book_title(book_id.split('_')[-1])  # Get the book title
                    from metadata.csv
                valid_positions = [pos for pos in info['position'] if pos]
                if valid_positions:
                    print(f"\nBook Title: {title}")
                    paragraphs = read_paragraphs(book_id.split('_')[-1],
                        valid_positions, word)
                    for paragraph in paragraphs:
                        print(paragraph)
        else:
            print(f"The word '{word}' was not found in any books.")
    except ValueError as e:
        print(e)

```

```
if __name__ == "__main__":
    word = input("Enter the word to search: ")
    search_word_across_books(word)
```

Listing 4: Inverted Indexer Word Search API

Code Explanation

Retrieving Book Titles The `get_book_title()` function is responsible for obtaining the title of a book based on its ID. This function reads the `metadata.csv` file where book metadata is stored, such as the book ID and title. If the title is found, it is returned. If not, the book's ID is returned as a fallback. This ensures that even if the title isn't available, the system has a way to identify the book.

Looking Up Word Occurrences The `get_word_occurrences_in_books()` function retrieves the occurrences of a word from the corresponding JSON file. The file is organized in folders based on the first letter of the word. If the word is not found (i.e., if the file does not exist), the function raises an exception, informing the user that the word has not yet been indexed.

Displaying Word Occurrences The function `read_paragraphs()` is responsible for displaying the occurrences of the word within each book. It reads through the text file of the book and highlights the occurrences of the word using ANSI escape codes. These escape codes change the color of the word to purple when displayed in the terminal. The function also shows the line and position of each occurrence of the word, which helps the user locate the word precisely within the book.

Main Search Function The main function, `search_word_across_books()`, ties everything together. It first normalizes the word by converting it to lowercase and applying lemmatization to handle variations (such as singular/plural or verb forms). Then, it retrieves the books where the word occurs using `get_word_occurrences_in_books()`. For each book, it fetches the title and displays the lines where the word appears using the `read_paragraphs()` function. If the word is not found in any book, a message is displayed indicating that no occurrences were found.

2.5 MongoDB Indexer API

The API performs a search on a MongoDB collection where the words are stored. Each word is associated with a list of books where it appears, and in each book, the paragraphs and the number of times the word occurs in those paragraphs are saved. When the user enters a word in the terminal, the API queries MongoDB, searches for the corresponding books, and returns the exact positions within the paragraphs where the word was found.

In addition to returning the results, the API uses the `lemmatize` function to normalize the words, so that variations such as "book" and "books" are considered the same word. It also handles case-insensitivity to ensure that all possible occurrences are captured.

Once the information is obtained, the API prints the book titles, which are extracted from a `metadata.csv` file, and then shows the lines and positions where the word appears, highlighting it in magenta to enhance visualization.

Full Code

```
import csv
import re
from pymongo import MongoClient
from unidecode import unidecode
from simplemma import lemmatize
from termcolor import colored

# MongoDB connection configuration
client = MongoClient('mongodb://localhost:27017/')
db = client['SEARCH_ENGINE']
words_collection = db['WORDS']

```

```
# Load metadata from the CSV file to get the book title
def get_book_title_from_csv(book_id):
    with open('datalake/metadata.csv', mode='r', encoding='utf-8') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            if row['ID'] == book_id:
                return row['Title']
    return f"BookID_{book_id}"  # Default title if not found

# Read the lines of a book and search for those containing the search word (case-insensitive)
def read_lines_with_word(book_file, paragraphs, search_word):
    try:
        with open(f"datalake/books/{book_file}", encoding='utf-8') as file:
            lines = file.readlines()

        result = []
        paragraph_count = 0
        current_paragraph_lines = []

        search_word_lower = search_word.lower()  # Convert the word to lowercase for case-insensitive search

        for i, line in enumerate(lines):
            # If the line is empty, the paragraph has ended
            if line.strip() == "":
                if current_paragraph_lines:
                    # Only if the current paragraph is in the selected ones
                    if paragraph_count in paragraphs:
                        for current_line in current_paragraph_lines:
                            # Case-insensitive search
                            if search_word_lower in current_line.lower():
                                # Highlight the word in magenta while maintaining the original case
                                highlighted_line = re.sub(f"(?i)({re.escape(search_word)})", colored(r"\1", 'magenta'), current_line)
                                result.append(f"Line {i + 1}: {highlighted_line.strip()}")
                    current_paragraph_lines = []  # Reset the paragraph
                    paragraph_count += 1  # Increase the paragraph counter
            else:
                # Add the current line to the paragraph
                current_paragraph_lines.append(line)

        return result

    except FileNotFoundError:
        print(f"The file '{book_file}' was not found.")
        return []

# Extract the paragraph number from MongoDB keys
def extract_paragraph_numbers(paragraph_keys):
    paragraph_numbers = []
    for key in paragraph_keys:
        match = re.search(r'\d+$', key)  # Look for numbers at the end of the string
        if match:
            paragraph_numbers.append(int(match.group()))
        else:
            print(f"Skipping invalid paragraph key: {key}")
    return paragraph_numbers

# Search for a word in MongoDB and print the lines that contain it
def search_word(word, lang='en'):
    word = unidecode(word.lower()).strip()  # Normalize and clean the word
    word = lemmatize(word, lang)  # Lemmatize the word

    # Query MongoDB to search for the word
    result = words_collection.find_one({'word': word})

    if result:
```

```
        # Print the total occurrences
        print(f"\nThe word '{word}' appears in the following books:")

        # Iterate over the books where the word appears
        for book_file, book_data in result['books'].items():
            book_id = book_file.split('.')[0]  # Extract the book ID
            book_title = get_book_title_from_csv(book_id)  # Get the title from the
                CSV
            print(f"\nBook Title: {book_title}")

            # Extract the paragraph numbers
            paragraphs = extract_paragraph_numbers(book_data.keys())
            lines_with_word = read_lines_with_word(book_file, paragraphs, word)
            for line in lines_with_word:
                print(line)
    else:
        print(f"\nThe word '{word}' was not found in the database.\n")

# Main function
def main():
    word_to_search = input("Enter the word you want to search: ")
    search_word(word_to_search)

# Execute the program
if __name__ == "__main__":
    main()
```

Listing 5: MongoDB Word Search API

Code Explanation

MongoDB Configuration and Book Title Retrieval The first part of the code establishes a connection with a local MongoDB instance through the `MongoClient`. The database used for the search system is `SEARCH_ENGINE`, and the collection where the words are stored is `WORDS`. To retrieve the title of a book, the function `get_book_title_from_csv()` is used, which looks up the title in a `metadata.csv` file that stores book metadata (like title and ID). If the title is not found, the function returns a default string based on the book ID.

Reading Lines Containing the Word The function `read_lines_with_word()` processes each book to find the lines that contain the search word. It operates on a case-insensitive basis by converting both the line and the word to lowercase. Additionally, it highlights the found word using the `termcolor` package, displaying it in magenta. The paragraphs are processed by splitting the book's content into sections and scanning through each paragraph.

Extracting Paragraph Information The function `extract_paragraph_numbers()` takes the keys from MongoDB (which are strings that identify the paragraphs) and extracts the corresponding paragraph numbers. If a key does not contain a valid number, it is skipped, and an appropriate message is printed.

Searching for a Word in MongoDB The `search_word()` function handles the main logic of searching for a word in the database. It first cleans the word, converts it to lowercase, and applies lemmatization to normalize it. Afterward, it queries the MongoDB collection for the word. If found, it prints the books where the word appears and calls `read_lines_with_word()` to display the relevant lines. If the word is not found, an appropriate message is displayed.

Main Program Execution Finally, the `main()` function prompts the user to input a word they wish to search for and executes the search using the previously defined functions. The word is searched for in the MongoDB database, and the relevant results are printed in the terminal.

3 Experiments

3.1 Benchmarking the Indexing Function

To evaluate the performance of the `execute` function, the `pytest-benchmark` tool was used. This module in `pytest` allows measuring the time it takes for a function to complete, along with additional data such as standard deviation, mean, and operations per second (OPS).

The following code was used to perform the benchmark:

```python
import pytest
import shutil
import os
from indexer1 import execute, update_last_book_id

@pytest.fixture(scope="function")
def setup_teardown():
    # Setup: Remove the "datamart" directory and reset the ID file
    shutil.rmtree("datamart", ignore_errors=True)  # Remove datamart if it exists
    update_last_book_id("resources/lastBookId.txt", 0, 1)  # Reset the ID file
    yield
    # Additional cleanup can be done here if needed

def test_execute_benchmark(benchmark, setup_teardown):
    # Benchmark the execute function
    benchmark(execute)
```

Listing 6: Benchmark Code for the execute Function (First Test)

This test uses a fixture to set up the environment before each benchmark execution. It deletes the `datamart` directory and resets the `lastBookId.txt` file, ensuring that every test runs under the same initial conditions.

3.1.1 First Benchmark Results

The results of the benchmark provide the following execution times for the `execute` function:

- **Minimum**: 0.4507 ms
- **Maximum**: 30.4305 ms
- **Mean**: 0.6052 ms
- **Standard Deviation**: 0.8659 ms
- **Median**: 0.5446 ms
- **Interquartile Range (IQR)**: 0.1190 ms
- **Operations Per Second (OPS)**: 1.6523 Kops/s

The benchmark results show a wide range of execution times, with a mean of 0.6052 milliseconds and a standard deviation of 0.8659 milliseconds, indicating variability in execution speed across different runs. The function achieved approximately 1.65 operations per second (Kops/s).

3.1.2 Second Benchmark Results

In the second benchmark, the `execute` function was tested with a different input choice (''2''), which processes all files. The following results were obtained:

```python
import pytest
from indexer2 import execute

@pytest.mark.benchmark
def test_execute_benchmark(benchmark):
    # Benchmark the `execute()` function with '2' as the input to process all files
    benchmark(execute, choice='2')
```

Listing 7: Benchmark Code for the execute Function (Second Test)

The results of the second benchmark are as follows:

- **Minimum**: 4.3335 ms
- **Maximum**: 5.0354 ms
- **Mean**: 4.6000 ms
- **Standard Deviation**: 0.3195 ms
- **Median**: 4.4035 ms
- **Interquartile Range (IQR)**: 0.5250 ms
- **Operations Per Second (OPS)**: 217.3904 ops/s

The benchmark results show a narrower range of execution times compared to the first test, with a mean of 4.6000 milliseconds and a standard deviation of 0.3195 milliseconds. The function achieved approximately 217.39 operations per second (ops/s).

4 Conclusion

As we can see on the benchmarks, handling the storage and indexing with ad-hoc functions yields better results than using an external database, especially for smaller books, where the cost of connecting to the database is disproportionate. For larger books however, the solutions we used seem to be less effective than the solutions implemented in the database, so future versions of the program have room for optimization when working with large texts.

4.0.1 Future improvements

In terms of possible improvements, we have considered adding multilanguage support, requesting specific works from projeect Güthenberg, and generating a summary of the work through a LLM.

4.0.2 Source Code on GitHub

The source code and the benchmark tests are available in the following GitHub repository:
https : //github.com/LuisPereraPerez/BigData_proyect