Comparative Analysis of Inverted Indexing Strategies for Efficient Search in Gutenberg Project Books

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

October 2023

Abstract

This paper presents a comparative analysis of multiple strategies and implementations for building an inverted indexer for books sourced from the Gutenberg Project. The primary objective was to create efficient indexing systems capable of extracting and cataloging English words from a diverse collection of literary works, enabling users to search for specific words and discover relevant books.

The introduction emphasizes the importance of indexing in information retrieval and discusses the choice of the Gutenberg Project as the data source. A literature review provides context for the project's methodology, focusing on inverted indexing in search engines.

The methodology section outlines the indexer's architecture, including data acquisition and preprocessing from the Gutenberg Project. It includes descriptions of various implementation strategies, such as Python's dictionary-based approach, SQLAlchemy-based implementations and SQLite databases.

Experiments were conducted to evaluate each implementation's performance, comparing indexing speed, search speed, and accuracy. The discussion section analyzes the advantages and limitations of each approach and addresses project challenges. The conclusion highlights the contributions of this research and suggests potential future enhancements for managing large textual datasets effectively.

This project makes a valuable contribution to information retrieval, offering practical solutions for searching and discovering content within the Gutenberg Project's extensive literary collection.

1 Introduction

In the ever-expanding digital landscape, the ability to efficiently and effectively retrieve information from vast textual repositories has become increasingly paramount. As information seekers, researchers, and enthusiasts alike navigate the wealth of knowledge available online, search engines and indexing systems stand as the gatekeepers to this digital treasury. Among the many challenges in the realm of information retrieval, constructing robust and responsive indexing mechanisms is fundamental.

The Gutenberg Project, a longstanding initiative that offers a treasure trove of public domain literary works, presents a unique opportunity for exploring and refining indexing techniques. This paper delves into the development and comparative analysis of multiple strategies for constructing an inverted indexer specifically tailored to the Gutenberg Project's collection of English-language books.

1.1 Motivation

The motivation for this endeavor is rooted in the sheer volume and diversity of literary content hosted by the Gutenberg Project. With over 60,000 titles encompassing a wide array of genres, styles, and historical periods, this digital library presents a rich dataset for indexing experiments. By creating an indexer capable of extracting and cataloging English words from this corpus, we aim to facilitate comprehensive search and discovery within these literary treasures.

1.2 Objectives

The primary objectives of this project are as follows:

- To design and implement multiple strategies for building an inverted indexer.
- To evaluate the performance of each implementation strategy in terms of indexing speed, search speed, and accuracy.
- To provide insights into the advantages and limitations of different approaches in managing large textual datasets.
- To contribute practical solutions for searching and discovering content within the Gutenberg Project's extensive collection.

1.3 Structure of the Paper

This paper is organized as follows:

- **Section 2: Literature Review** explores the importance of inverted indexing in search engines and reviews relevant literature in the field of information retrieval.

- **Section 3: Methodology** outlines the architecture and technical aspects of the indexer, with distinct subsections for each implementation strategy.
- **Section 4: Experiments and Comparisons** discusses the experimental setup, performance metrics, and results obtained from evaluating the different strategies.
- **Section 5: Conclusion** summarizes the main contributions and suggests avenues for future research.
- **Section 6: Appendices** contains supplementary information, including code listings and additional experiment details.
- **References** provide citations for the sources and references used throughout the paper.

In summary, this paper embarks on a journey to explore the intricacies of indexing within the context of the Gutenberg Project's vast literary archive. Through a comparative analysis of various implementation strategies, we aim to shed light on the nuances of information retrieval in the digital age, offering insights and practical solutions for managing and searching extensive textual datasets.

2 Literature Review

The importance of indexing in information retrieval cannot be overstated. In the digital age, search engines and indexing mechanisms play a pivotal role in enabling users to access and retrieve relevant information from vast textual repositories. In this section, we explore the foundational concepts of inverted indexing in search engines and review key literature in the field of information retrieval.

2.1 Inverted Indexing

Inverted indexing is a fundamental technique in information retrieval systems. It involves the creation of an index that maps terms (usually words) to the documents or locations in which they occur. This inverted structure enables efficient and rapid searching of documents containing specific terms. The concept of an inverted index was popularized by Gerard Salton in the 1960s as part of the SMART Information Retrieval System [3]. The inverted index is constructed through a two-step process: indexing and querying. During indexing, documents are parsed, and terms are extracted and stored in the index along with information about their occurrences, such as document IDs and positions. Querying involves searching the index for terms and retrieving the associated documents.

2.2 Information Retrieval Systems

Information retrieval systems have evolved significantly over the years, with various models and techniques proposed to improve retrieval effectiveness. The Vector Space Model (VSM) is a widely adopted approach in information retrieval, representing documents and queries as vectors in a multi-dimensional space. Cosine similarity is often used to measure the relevance between queries and documents in this model [2, 4].

2.3 Open Source Search Engines

Open-source search engines like Apache Lucene and Elasticsearch have gained prominence in the realm of information retrieval. These systems provide powerful indexing and searching capabilities, making them popular choices for building custom search engines and information retrieval applications [1].

2.4 Gutenberg Project and Text Mining

The Gutenberg Project has been a valuable resource for researchers and enthusiasts interested in text mining and natural language processing. Researchers have utilized Gutenberg's extensive collection of public domain books for various text analysis tasks, including sentiment analysis, topic modeling, and language modeling. However, to our knowledge, there is limited literature specifically addressing the development of inverted indexers tailored to the Gutenberg Project's collection and the comparative analysis of different implementation strategies.

2.5 Summary

In summary, the literature review highlights the significance of inverted indexing in information retrieval, emphasizing its role in facilitating efficient and accurate document retrieval. We have also touched upon various information retrieval models, open-source search engines, and the potential of the Gutenberg Project's dataset for text mining and indexing tasks. While existing literature provides a foundation for our work, our project aims to contribute by exploring and comparing multiple strategies for building an inverted indexer specifically for the Gutenberg Project's extensive collection of English-language books.

3 Methodology

The project's methodology revolves around the development of an indexer for English words extracted from Gutenberg Project books. It encompasses multiple implementations, including a Python dictionary-based approach, a SQLAlchemy/PostgreSQL database-driven solution and a SQLite database. The methodology involves data retrieval, tokenization, and the creation of an inverted index

that associates words with their respective books and word counts. Additionally, a user-friendly search interface allows users to query the indexer and explore books where specific words appear. This comprehensive methodology guides the project's development, offering insights into different strategies for efficiently cataloging and accessing literary content from the Gutenberg Project.

First, a file containing books in .txt format has been traversed in order to read these books line by line and separate them into metadata and content. Subsequently, two files will be created where the corresponding metadata for each book with its respective title will be saved. The content will be saved in the other remaining file, also with the title of each book.

3.1 Implementation using Python's Dictionary

The implementation of the indexer in this subsection is based on Python's built-in dictionary data structure. This approach leverages Python's simplicity and efficiency to construct an inverted index for the Gutenberg Project's collection of English-language books. The methodology for this implementation can be summarized as follows:

3.1.1 Document Processing

The first step in building the inverted index is document processing. The application downloads and processes individual books from the Gutenberg Project's online repository. The **key document processing steps** include:

- **Locating the Start of the eBook:** The document processing begins by identifying the start of the eBook content within the downloaded text. This is done by searching for specific markers or patterns in the document. In this implementation, the start of the eBook is identified using the pattern "`*** .*? ***`". This pattern will match the first appearance of text surrounded by "`***`". In the books it will be text "`*** START OF THE PROJECT GUTENBERG EBOOK book title ***`", which marks the start of eBook.
- **Removing Metadata:** Once the start of the eBook is located, any metadata or introductory content preceding it is removed. This ensures that the indexer focuses on the actual content of the book.
- **Extracting Title:** The title of the book is extracted from the document. This information is used to create a `Book` object, which includes the book's title and a URL to the book on the Gutenberg Project's website.

3.1.2 Word Processing and Indexing

With the eBook content prepared, the next step involves word processing and indexing. This process is responsible for extracting individual words from the document and creating an inverted index. The **main components of this step** are as follows:

- **Tokenization:** The NLTK library is employed for word tokenization. NLTK's tokenization capabilities are utilized to break down the eBook's text into individual words.
- **Word Validation:** Before indexing, each word is subjected to validation. This validation step ensures that stop words and prepositions are excluded from the indexing process. Words that do not pass this validation are skipped.
- **Lowercasing:** To ensure case-insensitive indexing, all words are converted to lowercase.
- **Inverted Indexing and Counting:** A Python dictionary is used as the basis for constructing the inverted index. The words serve as keys in the dictionary, and the associated values are lists of `DictionaryEntry`. For each word-book association, a count is maintained to track how many times the word appears in the document. The `DictionaryEntry` consists of the Book object which represents the document and the count of word's appearance in the specific book.

3.1.3 Search and Retrieval

Once the inverted index is constructed, the application provides a search and retrieval mechanism through an API, enabling users to search for words within the indexed books. The **search functionality** includes:

- **User Input:** Users can interact with the application by making requests to a designated API endpoint, providing the word they want to search for.
- **Search Operation:** The entered word serves as a query parameter in the API request. The application uses this word as a key to query the Python dictionary-based inverted index. The response consists of a list of books in which the word is found, along with the count of appearances in each book.
- **Display Results:** The API responds with a list of books where the word appears, along with the respective counts, allowing users or client applications to explore and access relevant literary works programmatically.

3.1.4 Dictionary Persistence using JSON

To ensure the longevity of the constructed inverted index and enable fast retrieval of data in subsequent runs, the application includes mechanisms for persisting the dictionary. This is achieved by storing the dictionary in a JSON file format, which allows for easy serialization and deserialization. The key components of dictionary persistence are as follows:

- `save_dictionary` **Method:** The `save_dictionary` method is responsible for serializing the inverted index, which is implemented as a Python dictionary, into a JSON-formatted file. This method ensures that the current state of the index is preserved for future use. It opens a designated JSON file, encodes the dictionary data, and writes it to the file.

- `load_dictionary` **Method:** The `load_dictionary` method performs the inverse operation of `save_dictionary`. It reads the JSON-formatted file containing the serialized inverted index, decodes the data, and reconstructs the dictionary in memory. This method is used when the application starts to load the previously saved dictionary, enabling the resumption of indexing and search operations from the last saved state.

The use of JSON for dictionary persistence ensures that the application can efficiently manage and retrieve the inverted index, even across different runs of the program. It enhances the usability of the indexer by reducing the need to re-index documents during each execution and improves overall system performance.

3.1.5 Summary

This implementation approach showcases the utilization of Python's dictionary data structure to create an efficient inverted index for text documents while also providing information about the frequency of word appearances within each document. It offers a simple yet effective solution for searching and discovering content within the Gutenberg Project's extensive collection of English-language books.

3.2 Implementation using SQLAlchemy and PostgreSQL

In this subsection, we present the methodology employed for implementing the indexer using SQLAlchemy, a powerful Object-Relational Mapping (ORM) library, and PostgreSQL, a robust relational database system. All the codes used in this section can be found in the subsection 7.1. This approach offers enhanced data management capabilities, persistence, and efficient querying. The key steps in this implementation are as follows:

3.2.1 Database Running in a Docker Container

To ensure a streamlined and consistent development environment, the PostgreSQL database is configured to run within a Docker container. This approach offers several advantages, including isolation, ease of setup, and portability across different systems.

The PostgreSQL database container is created using the provided Dockerfile (`Dockerfile`), which specifies the necessary configurations, including the PostgreSQL version and environment variables.

By encapsulating the database within a Docker container, developers can work with a consistent database environment regardless of their local setup. This eliminates potential compatibility issues and simplifies the process of setting up the database for the indexer.

The Docker container is named and mapped to port 5432, allowing easy access to the PostgreSQL database from the host system. This containerization approach enhances project portability and accelerates development by providing a reliable and isolated database environment.

3.2.2 Database Configuration and Model Definition

The first crucial step is setting up the PostgreSQL database and defining the necessary data models. This includes the creation of tables to store information about books, words, and their associations. The critical components of this step include:

- **Database Configuration:** The database connection details, such as the host, database name, user, and password, are specified in a configuration file. In this implementation, a 'database.ini' file is used for configuration.
- **Model Definition:** Data models for books, words, and their associations (book-words) are defined using SQLAlchemy. The models include attributes for book titles, URLs, word text, and word counts.

3.3 Document Processing and Indexing

In this phase of the implementation, we outline the key steps involved in processing and indexing books sourced from the Gutenberg Project using SQLAlchemy and PostgreSQL. This process involves the retrieval, validation, and indexing of books and their associated words. The primary components include:

- **Document Retrieval:** The application downloads individual books from the Gutenberg Project's repository. It employs a `FileManager` utility that separates the book's metadata and content and save these two parts into separated files.
- **Database Insertions:** Extracted book information is then inserted into the PostgreSQL database. SQLAlchemy's session and repository classes are utilized for this purpose. This includes adding book records, words, and their associations in a structured manner.
- **Word Validation:** Prior to indexing, words undergo validation to ensure their correctness. Similar to the Python dictionary-based approach, this step involves excluding common stop words and prepositions from the indexing process to enhance data quality.
- **Inverted Indexing and Counting:** Inverted indexing is achieved by associating words with the books in which they appear and counting their

occurrences. SQLAlchemy's capabilities are employed to manage relationships and update word counts within the database.

3.3.1 Search and Retrieval

The search and retrieval functionality remains consistent with the Python dictionary implementation, allowing users to search for words and discover the books in which those words appear. The main components include:

- **User Input:** Users can enter a word they wish to search for within the indexed books.
- **Database Querying:** The application queries the PostgreSQL database to retrieve books associated with the searched word, along with their respective word counts.
- **Display Results:** The search results are displayed to users, enabling them to explore and access relevant books from the Gutenberg Project's collection.

This implementation using SQLAlchemy and PostgreSQL enhances the indexer's capabilities by leveraging a relational database for data storage and retrieval. It provides a robust and scalable solution for managing large volumes of text data, indexing words, and facilitating efficient searches.

3.4 Database Inverted Index

In this methodology, we have harnessed a repertoire of classic Python techniques. Our implementation hinges on a meticulously designed database, housing two pivotal tables. One table is dedicated to storing the vast library of books within our data lake, while the other serves as an integral inverted index. To meticulously craft this inverted index and ensure a seamless execution, we strategically segmented the project into the following distinct classes:

3.4.1 Books Processing

To initiate the process of indexing words extracted from the books within our data lake, our first challenge was handling the raw book files sourced from the Gutenberg Project. These files presented a unique complexity as they combined both metadata and book content within the same file. Furthermore, the filenames did not correspond to the actual book titles. To streamline our operations and enhance manageability, we devised the following classes:

- **FileManager:** This class is a collection of functions specifically crafted to interact with and manipulate files. Its primary purpose is to facilitate the operations performed by the `BookManager` class, which is responsible for handling the content of the files.

- **BookManager:** The core function of this class revolves around the separation of content within the files obtained from the Gutenberg Project. These files typically comprise two distinct sections: metadata and the book's actual content. To accomplish this task, we establish two separate directories, one for housing the metadata and another for storing the book content.

3.4.2 Indexing and Processing

To manage word indexing with book IDs in a database, the 'Indexer' class is used. This class has access to the DataLake and the 'DatabaseManager,' which controls the database using SQLite.

The process starts by accessing the path containing only the content of the books, excluding metadata, and iterating through the files present in it. Each file is read individually, the title (corresponding to the file name) is extracted, and then its content is read.

For each read book, an index is assigned. Next, the content of the book is split into words, passing through a word filter with the help of the 'nltk' library to omit words such as prepositions, focusing only on meaningful words. Additionally, the selected words are converted to lowercase and added to a dictionary, where they act as keys, and their value is a list of book IDs to which they belong.

Once all the files have been processed in this way, the 'DatabaseManager' class receives the words and their respective IDs from the dictionary through a function. These data are then inserted into the inverted index table of the database.

3.4.3 Database management

In order to ensure the proper development of a database system while adhering to the principles of the Single Responsibility Rule, it is imperative to implement subclasses for each table intended for inclusion within the database structure. In essence, this approach involves the creation of a superclass, denoted as `DatabaseManager`, which is primarily responsible for invoking its respective parent classes, namely `BookTableManager` and `InvertedIndexTableManager`. As previously stipulated, multiple tables may be created as necessitated by the application, and in the current context, we consider two specific tables.

- `BookTableManager`: is a class that serves the vital function of overseeing the proper functioning of the "book" table, incorporating three distinct methods within its structure. These methods are designed to facilitate fundamental operations, encompassing table creation, selection of values for API responses, and the insertion of data into the table.

- `InvertedIndexTableManager`: is a class which corresponds to the "inverted_index" table, is also equipped with three essential methods: creation, selection, and data insertion. It is noteworthy that both the database

and the tables have been established utilizing the Python library `sqlite3`. This library is employed to facilitate the requisite database operations, thereby ensuring the integrity and efficiency of the database management system.

3.4.4 Query

To implement the query, we have developed an API using the 'Flask' library. This API allows the user to specify the word they want to search for as a parameter. First, this word is passed as input to a function within the DatabaseManager class, which returns the identifiers (IDs) of the books containing that word. Then, these IDs are used as input for another function within the same class, which returns the details of the corresponding books. Finally, the results are presented in JSON format through the API. If the searched word does not exist in the database, an empty list is returned as a response.

4 Experiments and Comparisons

In this section, we will delve into the results obtained from the experimental phase of each implementation and evaluate their respective performance.

To begin, let's discuss the Python dictionary implementation. The results achieved in this implementation were impressive enough to warrant its acceptance as a final project deliverable. Despite its simplicity, it notably stands out as the fastest among the three implementations.

Moving on to the implementation that utilizes SQLAlchemy and PostgreSQL, it produced even more favorable results compared to the Python dictionary approach. This implementation is notably complex due to its reliance on a class-based database structure, but it excels in terms of efficiency. However, it's important to note that the primary drawback of this implementation lies in its longer processing time, as it took 528 seconds to index the book "Pinocchio."

Finally, we come to the database implementation, which performs better than one might anticipate from a typical SQLite3 setup. While its execution time is slower than the dictionary implementation, this was expected due to the initial process of segregating files into metadata and book content stored in separate directories. In terms of complexity, it is less intricate than the SQLAlchemy and PostgreSQL implementation, but it does require a higher level of precision than the Python dictionary implementation.

5 Conclusion

In conclusion, this segment of the project underscores the significance of this modest undertaking within the broader context of future project development. As previously demonstrated in the Experiments and Comparisons section, it becomes evident that the utilization of a dictionary implementation significantly

outperforms the process of indexing an inverted index. This performance superiority holds true regardless of the specific dependency employed; the dictionary implementation consistently exhibits markedly improved efficiency.

However, it is important to note that the deployment of a database in this implementation offers a distinct advantage: the ability to persistently store and reuse the database for various purposes. This feature enhances the versatility and long-term utility of the implemented solution. Consequently, while the dictionary-based approach excels in terms of performance, the database-based implementation brings an added dimension of data persistence and reusability to the table, making it a valuable asset for future endeavors.

6 Future Work

This project can help us with future work because it provides information on which real-time and easily accessible information storage method is faster when accessing it. In this case, the dictionary is faster than creating a database. However, we wouldn't have the information stored in a dictionary because the dictionary is initialized every time the file is executed. To improve this, the information could be saved in JSON format for future work.

Another thing to consider for future work is that instead of having predownloaded books, our project should automatically search the Gutenberg Project website for books and download them, and then separate them into content and metadata.

Finally, for a possible future implementation, it would be that when searching for a word, it returns the number of appearances in each of the corresponding books.

7 Appendices

7.1 SQLAlchemy implementation

7.1.1 Main

```python
from sqlalchemy.orm import Session

from app.db import database
from app.db.model import Base
from app.indexer.document_processor import DocumentProcessor
from app.repository.book_repository import BookRepository
from app.repository.bookwords_repository import BookWordsRepository
from app.repository.word_repository import WordRepository

from app.file_manager.file_manager import FileManager

session = Session(database.engine)

book_repository = BookRepository(session)
word_repository = WordRepository(session)
bookwords_repository = BookWordsRepository(session)

if __name__ == "__main__":
    Base.metadata.drop_all(database.engine)
    Base.metadata.create_all(database.engine)

    file_manager = FileManager("data/")
    file_manager.download_book(500)

    document_processor = DocumentProcessor(book_repository,
    word_repository, bookwords_repository)
    document_processor.index_documents(file_manager.content_dir)

    while True:
        word = input("Search for word: ")
        print(book_repository.find_books_from_word(word.lower()))
```

Listing 1: main.py file for the SQLAlchemy implementation

7.1.2 Repositories

```python
from sqlalchemy import select

from app.db.model import Word


class WordRepository:

    def __init__(self, session):
        self.session = session

    def find_word(self, word: str):
        stmt = select(Word).where(Word.word == word)
        return self.session.scalars(stmt).first()

    def add_word(self, word_str):
```

```
        word = Word(word=word_str)
        self.session.add(word)
        self.session.commit()
        return word

    def add_word_with_book(self, word_str, book):
        word = self.add_word(word_str)
        word.books.append(book)
        self.session.commit()
        return word

    def add_book_to_word(self, word, book):
        word_db = self.session.get(Word, word.id)
        word_db.books.append(book)
        self.session.commit()
        return word_db
```

Listing 2: word_repository.py file for the SQLAlchemy implementation

```
from sqlalchemy import select

from app.db.model import Book, Word, BookWord


class BookRepository:
    def __init__(self, session):
        self.session = session

    def add_book(self, title):
        book = Book(book_title=title)
        self.session.add(book)
        self.session.commit()
        return book

    def find_book_by_id(self, id):
        stmt = select(Book).where(Book.id == id)
        return self.session.scalars(stmt).first()

    def find_books_from_word(self, word: str):
        stmt = select(Word).where(Word.word == word)
        word = self.session.scalars(stmt).first()
        if word is None:
            return None

        result: (Book, int) = []

        for book in word.books:
            stmt = (select(BookWord.count)
                    .where(BookWord.book_id == book.id, BookWord.word_id == word.id))
            result.append((book, self.session.scalars(stmt).first()))

        return result

    def find_book_by_title(self, title):
        stmt = select(Book).where(Book.book_title == title)
```

```
        return self.session.scalars(stmt).first()
```

Listing 3: book_repository.py file for the SQLAlchemy implementation

```
from sqlalchemy import select

from app.db.model import Book, Word, BookWord


class BookRepository:
    def __init__(self, session):
        self.session = session

    def add_book(self, title):
        book = Book(book_title=title)
        self.session.add(book)
        self.session.commit()
        return book

    def find_book_by_id(self, id):
        stmt = select(Book).where(Book.id == id)
        return self.session.scalars(stmt).first()

    def find_books_from_word(self, word: str):
        stmt = select(Word).where(Word.word == word)
        word = self.session.scalars(stmt).first()
        if word is None:
            return None

        result: (Book, int) = []

        for book in word.books:
            stmt = (select(BookWord.count)
                    .where(BookWord.book_id == book.id, BookWord.word_id == word.id))
            result.append((book, self.session.scalars(stmt).first()))

        return result

    def find_book_by_title(self, title):
        stmt = select(Book).where(Book.book_title == title)
        return self.session.scalars(stmt).first()
```

Listing 4: boookwords_repository.py file for the SQLAlchemy implementation

7.1.3 Model

```
from typing import List

from sqlalchemy import String, ForeignKey, Column, Integer
from sqlalchemy.orm import DeclarativeBase, Mapped, relationship
from sqlalchemy.orm import mapped_column


class Base(DeclarativeBase):
    pass

```

```
11
12 class BookWord(Base):
13     __tablename__ = "book_words"
14     book_id = Column(Integer, ForeignKey('books.id'), primary_key=
       True)
15     word_id = Column(Integer, ForeignKey('words.id'), primary_key=
       True)
16     count = Column(Integer, default=1)
17
18
19 class Book(Base):
20     __tablename__ = "books"
21     id: Mapped[int] = mapped_column(primary_key=True)
22     book_title: Mapped[str] = mapped_column(String(100), unique=
       True, nullable=False)
23
24     def __str__(self) -> str:
25         return str(self.book_title)
26
27     def __repr__(self) -> str:
28         return str(self.book_title)
29
30
31 class Word(Base):
32     __tablename__ = "words"
33     id: Mapped[int] = mapped_column(primary_key=True)
34     word: Mapped[str] = mapped_column(String(100), unique=True,
       nullable=False)
35     books: Mapped[List["Book"]] = relationship(secondary="
       book_words")
```

Listing 5: model.py file for the SQLAlchemy implementation

7.1.4 Processors

```
1 import ssl
2
3 import nltk
4 from nltk.corpus import stopwords
5
6 from app.db.model import Book
7
8 try:
9     _create_unverified_https_context = ssl.
      _create_unverified_context
10 except AttributeError:
11     pass
12 else:
13     ssl._create_default_https_context =
       _create_unverified_https_context
14
15 nltk.download('stopwords')
16 stop_words = set(stopwords.words('english'))
17
18 prepositions = ["about", "above", "across", "after", "against", "
       along", "amid", "among", "around", "as", "at",
19                 "before", "behind", "below", "beneath", "beside", "
       between", "beyond", "but", "by", "concerning",
```

```
                "considering", "despite", "down", "during", "except", "for", "from", "in", "inside", "into", "like",
                "near", "of", "off", "on", "onto", "out", "outside", "over", "past", "regarding", "round", "since",
                "through", "to", "toward", "under", "underneath", "until", "unto", "up", "upon", "with", "within",
                "without"]


class WordProcessor:

    def __init__(self, book_repository, word_repository, bookwords_repository):
        self.book_repository = book_repository
        self.word_repository = word_repository
        self.bookwords_repository = bookwords_repository

    @staticmethod
    def is_word_correct(word):
        return not (word in stop_words or word in prepositions or not word.isalpha())

    @staticmethod
    def change_word(word):
        return word.lower()

    def insert_word_to_db(self, word: str, book: Book):

        word = self.change_word(word)
        if not self.is_word_correct(word):
            return

        # Check if word is already in database
        word_entity = self.word_repository.find_word(word)

        # Word is not in the database
        if word_entity is None:
            self.word_repository.add_word_with_book(word, book)
            return

        # Word is in the database
        # Check if word is already assigned to the book
        if book not in word_entity.books:
            self.word_repository.add_book_to_word(word_entity, book)
        else:
            self.bookwords_repository.increase_word_count(word_entity, book)
```

Listing 6: word_processor.py file for the SQLAlchemy implementation

```
import os

import nltk
import requests

import app.indexer.word_processor as word_processor
from app.repository.book_repository import BookRepository
```

```
from app.repository.bookwords_repository import BookWordsRepository
from app.repository.word_repository import WordRepository


class DocumentProcessor:

    def __init__(self, book_repository, word_repository,
    bookwords_repository):
        self.book_repository: BookRepository = book_repository
        self.word_repository: WordRepository = word_repository
        self.bookwords_repository: BookWordsRepository =
    bookwords_repository
        self.word_processor: word_processor.WordProcessor =
    word_processor.WordProcessor(book_repository, word_repository,
    bookwords_repository)

    def process_book(self, book, document):
        words = nltk.word_tokenize(str(document))
        for word in words:
            self.word_processor.insert_word_to_db(word, book)

    def index_documents(self, content_dir: dir):
        for file in os.listdir(content_dir):

            filename = os.path.join(content_dir, file)
            f = open(filename, "r")
            content = f.read()

            title, file_extension = os.path.splitext(file)
            if self.book_repository.find_book_by_title(title) is
    not None:
                print("Book with title=", title, " is already
    indexed.", sep='')
                return
            book = self.book_repository.add_book(title)

            self.process_book(book, content)
```

Listing 7: document_processor.py file for the SQLAlchemy implementation

7.1.5 Database setup

```
[postgresql]
host=localhost
dbname=database
user=postgres
port=5432
password=postgres
```

Listing 8: Database configuration file for the SQLAlchemy implementation

```
import configparser

from sqlalchemy import URL
from sqlalchemy import create_engine

config = configparser.ConfigParser()
config.read("app/db/database.ini")
```

```
8
9  url_object = URL.create(
10     "postgresql",
11     username=config["postgresql"]["user"],
12     password=config["postgresql"]["password"],
13     host=config["postgresql"]["host"],
14     port=int(config["postgresql"]["port"]),
15     database=config["postgresql"]["dbname"])
16
17 engine = create_engine(url_object)
```

Listing 9: database.py file for the SQLAlchemy implementation

7.1.6 Managing files

```
1  import os
2  import re
3
4  import requests
5
6
7  class FileManager:
8
9      def __init__(self, document_dir):
10         self.document_dir = document_dir
11
12         self.metadata_dir = os.path.join(document_dir, "metadata")
13         self.content_dir = os.path.join(document_dir, "content")
14
15         self.make_directory(self.metadata_dir)
16         self.make_directory(self.content_dir)
17
18     def download_book(self, book_id: int):
19         data = requests.get(self.make_url_from_id(book_id)).content
20
21         file_name = FileManager.sanitize_file_name(self.get_title(
       data))
22
23         self.separate_metadata(data, file_name)
24         self.separate_content(data, file_name)
25
26     @staticmethod
27     def sanitize_file_name(file_name):
28         invalid_chars = '\\/:*?"<>|'
29         for char in invalid_chars:
30             file_name = file_name.replace(char, '-')
31         file_name = file_name.strip().rstrip('.')
32         return file_name
33
34     def separate_metadata(self, document, name):
35         metadata = FileManager.get_metadata(document)
36         FileManager.write_file(os.path.join(self.metadata_dir, name
        + ".txt"), metadata)
37
38     def separate_content(self, document, name):
39         content = FileManager.get_content(document)
40         FileManager.write_file(os.path.join(self.content_dir, name
       + ".txt"), content)
```

```
41
42      @staticmethod
43      def make_url_from_id(doc_id):
44          return "https://www.gutenberg.org/cache/epub/" + str(doc_id
        ) + "/pg" + str(doc_id) + ".txt"
45
46      @staticmethod
47      def get_metadata(document: bytes):
48          match = "***"
49          index = str(document).find(match) - len(match)
50          return str(document)[:index]
51
52      @staticmethod
53      def get_content(document: bytes):
54          start_of_content = FileManager.locate_start_of_content(
        document)
55          return str(document)[start_of_content:]
56
57      @staticmethod
58      def get_title(document):
59          title = "Title: "
60          start_index = str(document).find(title) + len(title)
61          end_index = str(document).find('\\', start_index)
62          return str(document)[start_index:end_index]
63
64      @staticmethod
65      def write_file(file_path, data):
66          with open(file_path, 'w', encoding='utf-8') as file:
67              file.write(data)
68
69      @staticmethod
70      def locate_start_of_content(document: bytes):
71          return re.search("\*\*\* .*? \*\*\*", str(document)).end()
72
73      @staticmethod
74      def make_directory(directory):
75          os.makedirs(directory, exist_ok=True)
```

Listing 10: file_manager.py file for the SQLAlchemy implementation

7.2 SQLite3 implementation

7.2.1 Main

```
1 from controller import Controller
2
3 if __name__ == "__main__":
4     controller = Controller()
5     controller.controller()
```

Listing 11: main.py file for the SQLite3 implementation

7.2.2 Controller

```
1 from indexer import Indexer
2 from database_manager import DatabaseManager
3 from book_manager import BookManager
```

```
from api import app

class Controller:
    def __init__(self):
        pass

    def controller(self):
        dbm = DatabaseManager('database')
        idx = Indexer('datalake', 'content')
        mcm = BookManager('datalake', 'metadata', 'content')

        dbm.create_book_table()
        dbm.create_inverted_index_table()

        mcm.makedirs()
        mcm.book_manager()
        idx.inverted_index_of()

        app.run(debug=True)
```

Listing 12: controller.py file for the SQLite3 implementation

7.2.3 Api

```
from database_manager import DatabaseManager
from flask import Flask, jsonify, request


app = Flask(__name__)
dbm = DatabaseManager('database')


@app.route('/api/search/<word>', methods=['GET'])
def get_books(word):
    try:
        books_ids = dbm.select_books_ids(word)
        books = dbm.select_book_title(books_ids)
        return jsonify({"books": books})
    except Exception as e:
        return jsonify({"books": []})
```

Listing 13: api.py file for the SQLite3 implementation

7.2.4 Indexer

```
from database_manager import DatabaseManager
from file_manager import FileManager
import re
import nltk
from nltk.corpus import stopwords
nltk.download('stopwords')

stop_words = set(stopwords.words('english'))
import os

class Indexer:

    dbm = DatabaseManager('database')
```

```
    fm = FileManager()

    def __init__(self, datalake_dir, content_dir):
        self.datalake_dir = datalake_dir
        self.content_dir = content_dir
        self.dictionary = dict()

    def inverted_index_of(self):
        files = [self.fm.join(self.datalake_dir, self.content_dir,
    file)
                 for file in self.fm.listdir(self.fm.join(self.
    datalake_dir, self.content_dir))]

        for idx, file in enumerate(files):
            f, b = self.__read_file(file)
            self.dbm.insert_into_book_table(idx+1, b)
            self.__word_divider(f, idx+1)
        self.__insert_into_inverted_index_table(self.dictionary)

    def __read_file(self, filename):
        file = open(filename, 'r', encoding='utf-8')
        book = os.path.basename(filename).replace('.txt', '')
        file.close()
        return file, book

    def __word_divider(self, file, idx):
        file = open(file.name, 'r', encoding='utf-8')
        for line in file:
            words = re.findall(r'\b[a-zA-Z\']+\b', line)
            words = [word.lower() for word in words]
            self.__add_to_dictionary(idx, words)
        file.close()

    def __add_to_dictionary(self, idx, words):
        for word in words:
            if word not in stop_words:
                if word not in self.dictionary:
                    self.dictionary[word] = [idx]
                elif word in self.dictionary and idx not in self.
    dictionary[word]:
                    self.dictionary[word].append(idx)

    def __insert_into_inverted_index_table(self, dictionary):
        for word, book_id in dictionary.items():
            self.dbm.insert_into_inverted_index_table(word, book_id
    )
```

Listing 14: indexer.py file for the SQLite3 implementation (note that Overleaf autoformat code is incorrect)

7.2.5 Book Manager

```
from file_manager import FileManager


class BookManager:

```

```
    fm = FileManager()

    def __init__(self, datalake_dir, metadata_dir, content_dir):
        self.datalake_dir = datalake_dir
        self.metadata_dir = metadata_dir
        self.content_dir = content_dir

    def book_manager(self):
        for file_name in self.fm.listdir(self.datalake_dir):
            if file_name.endswith('.txt'):
                file_path = self.fm.join(self.datalake_dir,
    file_name)
                with open(file_path, 'r', encoding='utf-8') as
    book_file:
                    lines = book_file.readlines()
                metadata, content, title = self.
    __metadata_content_divider(lines=lines)
                title = self.__sanitize_file_name(title) + '.txt'
                new_file_path = self.fm.join(self.datalake_dir,
    title)
                self.fm.rename(file_path, new_file_path)
                self.__add_metadata(metadata, title)
                self.__add_content(content, title)

    def makedirs(self):
        self.fm.makedirs(self.fm.join(self.datalake_dir, self.
    metadata_dir))
        self.fm.makedirs(self.fm.join(self.datalake_dir, self.
    content_dir))

    def __metadata_content_divider(self, lines):
        for idx, line in enumerate(lines):
            if line.startswith('***'):
                metadata = lines[:idx]
                content = lines[idx+1:]
            if line.startswith('Title: '):
                title = line[len('Title: '):].strip()
        return metadata, content, title

    def __sanitize_file_name(self, file_name):
        invalid_chars = '\\/:*?"<>|'
        for char in invalid_chars:
            file_name = file_name.replace(char, '-')
        file_name = file_name.strip().rstrip('.')
        return file_name

    def __add_metadata(self, metadata, file_name):
        metadata_file_path = self.fm.join(self.datalake_dir ,self.
    metadata_dir, file_name)
        self.__writter(metadata_file_path, metadata)

    def __add_content(self, content, file_name):
        content_file_path = self.fm.join(self.datalake_dir, self.
    content_dir, file_name)
        self.__writter(content_file_path, content)

    def __writter(self, file_path, lines):
```

```
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write('\n'.join(lines))
```

Listing 15: book_manager.py file for the SQLite3 implementation

7.2.6 Database Manager

```
from book_table_manager import BookTableManager
from inverted_index_table_manager import InvertedIndexTableManager


class DatabaseManager(BookTableManager, InvertedIndexTableManager):
    def __init__(self, db_name):
        self.db_name = db_name
        BookTableManager.__init__(self, db_name)
        InvertedIndexTableManager.__init__(self, db_name)
```

Listing 16: database_manager.py file for the SQLite3 implementation

7.2.7 Book Table Manager

```
import sqlite3


class BookTableManager:
    def __init__(self, db_name):
        self.db_name = db_name

    def create_book_table(self):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()

        create_table_query = '''
            CREATE TABLE IF NOT EXISTS book (
                book_id INTEGER,
                title TEXT
            )
        '''
        cursor.execute(create_table_query)

        conn.commit()
        conn.close()

    def select_book_title(self, books_ids):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()
        titles = []

        for book_id in books_ids:
            consult = "SELECT title FROM book WHERE book_id = ?"
            cursor.execute(consult, (book_id,))
            result = cursor.fetchone()
            if result:
                titles.append(result[0])

        conn.close()
        return titles

```

```
    def insert_into_book_table(self, idx, title):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()

        cursor.execute('SELECT * FROM book WHERE book_id = ?', (idx,))
        existing_entry = cursor.fetchone()

        if existing_entry is None:
            cursor.execute(f'INSERT INTO book (book_id, title) VALUES (?, ?)', (idx, title))

        conn.commit()
        conn.close()
```

Listing 17: book_table_manager.py file for the SQLite3 implementation

7.2.8 Inverted Index Table Manager

```
import sqlite3


class InvertedIndexTableManager:
    def __init__(self, db_name):
        self.db_name = db_name

    def create_inverted_index_table(self):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()

        create_table_query = '''
            CREATE TABLE IF NOT EXISTS inverted_index (
                word TEXT,
                book_id TEXT
            )
        '''
        cursor.execute(create_table_query)

        conn.commit()
        conn.close()

    def select_books_ids(self, word):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()

        consult = f"SELECT book_id FROM inverted_index WHERE word = ?"
        cursor.execute(consult, (word,))
        books_id = [register[0] for register in cursor.fetchall()]
        conn.close()
        return eval(books_id[0])

    def insert_into_inverted_index_table(self, word, book_id):
        conn = sqlite3.connect(self.db_name + '.db')
        cursor = conn.cursor()

        cursor.execute(f'SELECT * FROM inverted_index WHERE word = ? AND book_id = ?', (word, str(book_id)))
```

```
        existing_entry = cursor.fetchone()

        if existing_entry is None:
            cursor.execute(f'INSERT INTO inverted_index (word, book_id) VALUES (?, ?)', (word, str(book_id)))

        conn.commit()
        conn.close()
```

Listing 18: inverted_index_table_manager.py file for the SQLite3 implementation

7.2.9 File Manager

```
import os


class FileManager:
    def __init__(self):
        pass

    def makedirs(self, directory):
        try:
            os.makedirs(directory, exist_ok=True)
        except Exception as e:
            print(f"Error creating directory {directory}: {str(e)}")

    def join(self, *args):
        return os.path.join(*args)

    def listdir(self, directory):
        return os.listdir(directory)

    def rename(self, file_path, new_file_path):
        os.rename(file_path, new_file_path)
```

Listing 19: file_manager.py file for the SQLite3 implementation

7.3 Dictionary implementation

7.3.1 Main

```
from controller import Controller

if __name__ == "__main__":
    content_path = 'datalake/content'
    json_file_path = 'index.json'

    controller = Controller(content_path, json_file_path)
    controller.run()
```

Listing 20: main.py file for the Dictionary implementation

7.3.2 Controller

```
from indexer import Indexer
from book_manager import BookManager
```

```
from api import API
from time import time

class Controller:
    def __init__(self, content_path, json_file_path):
        self.content_path = content_path
        self.json_file_path = json_file_path

    def run(self):

        start = time()
        idx = Indexer()
        bm = BookManager('datalake', 'metadata', 'content')

        bm.makedirs()
        bm.book_manager()
        idx.build_index(self.content_path)

        idx.save_index_to_json(self.json_file_path)

        end = time()

        print(f"Time taken: {end - start} seconds")

        app = API(self.json_file_path)
        app.run()
```

Listing 21: controller.py file for the Dictionary implementation

7.3.3 Indexer

```
import string
import nltk
from nltk.corpus import stopwords
from collections import defaultdict
from file_manager import FileManager
import re
import json

nltk.download('stopwords')
class Indexer:
    def __init__(self):
        self.contents = {}
        self.dictionary = defaultdict(lambda: defaultdict(int))
        self.fm = FileManager()

    def __read_contents(self, folder_path):
        for file_name in self.fm.listdir(folder_path):
            if file_name.endswith(".txt"):
                file_name_without_extension = re.sub(r'\.txt$', '',
 file_name)
                file_path = self.fm.join(folder_path, file_name)
                with open(file_path, "r", encoding="utf-8") as file
:
                    content = file.read()
                self.contents[file_name_without_extension] =
content

```

```
    def __tokenize_and_filter_words(self, text):
        text_without_punctuations = text.replace(".", "").replace('
    "', '').replace("'", "")
        words = nltk.word_tokenize(text_without_punctuations,
    language='english')
        stop_words = set(stopwords.words('english'))
        filtered_words = [word.lower() for word in words if word.
    lower() not in stop_words and word not in string.punctuation]
        cleaned_text = ' '.join(filtered_words)
        return cleaned_text

    def __process_file(self, content):
        cleaned_text = self.__tokenize_and_filter_words(content)
        words = cleaned_text.split()
        return words

    def build_index(self, folder_path):
        self.__read_contents(folder_path)
        for file_name, content in self.contents.items():
            words = self.__process_file(content)
            for word in words:
                self.dictionary[word][file_name] += 1


    def save_dictionary(self, json_file_path):
        with open(json_file_path, 'w', encoding='utf-8') as
    json_file:
            json.dump(self.dictionary, json_file)

    def load_dictionary(self, json_file_path):
        with open(json_file_path, 'r', encoding='utf-8') as
    json_file:
            self.dictionary = json.load(json_file)


    def search_word_in_dictionary(self, word):
        w = word.lower()
        frequencies = self.dictionary.get(w, None)
        if frequencies:
            sorted_frequencies = sorted(frequencies.items(), key=
    lambda x: x[1], reverse=True)
            file_names = [file_name for file_name in
    sorted_frequencies]
            return file_names
        return None
```

Listing 22: indexer.py file for the Dictionary implementation

7.3.4 Api

```
from flask import Flask, jsonify
from indexer import Indexer

class API:
    def __init__(self, index_file_path):
        self.app = Flask(__name__)
        self.idx = Indexer()
        self.idx.load_index_from_json(index_file_path)
```

```
9
10          @self.app.route('/api/search/<word>', methods=['GET'])
11          def get_word(word):
12              books = self.idx.search_word_in_dictionary(word)
13              return jsonify({"books": books})
14
15      def run(self):
16          self.app.run(debug=True)
17
18  if __name__ == "__main__":
19      index_file_path = 'index.json'
20      app = API(index_file_path)
21      app.run()
```

Listing 23: api.py file for the Dictionary implementation

7.3.5 Book Manager

```
1  from file_manager import FileManager
2
3
4  class BookManager:
5
6      fm = FileManager()
7
8      def __init__(self, datalake_dir, metadata_dir, content_dir):
9          self.datalake_dir = datalake_dir
10         self.metadata_dir = metadata_dir
11         self.content_dir = content_dir
12
13     def book_manager(self):
14         for file_name in self.fm.listdir(self.datalake_dir):
15             if file_name.endswith('.txt'):
16                 file_path = self.fm.join(self.datalake_dir,
       file_name)
17                 with open(file_path, 'r', encoding='utf-8') as
       book_file:
18                     lines = book_file.readlines()
19                 metadata, content, title = self.
       __metadata_content_divider(lines=lines)
20                 title = self.__sanitize_file_name(title) + '.txt'
21                 new_file_path = self.fm.join(self.datalake_dir,
       title)
22                 self.fm.rename(file_path, new_file_path)
23                 self.__add_metadata(metadata, title)
24                 self.__add_content(content, title)
25
26     def makedirs(self):
27         self.fm.makedirs(self.fm.join(self.datalake_dir, self.
       metadata_dir))
28         self.fm.makedirs(self.fm.join(self.datalake_dir, self.
       content_dir))
29
30     def __metadata_content_divider(self, lines):
31         for idx, line in enumerate(lines):
32             if line.startswith('***'):
33                 metadata = lines[:idx]
34                 content = lines[idx+1:]
```

```
            if line.startswith('Title: '):
                title = line[len('Title: '):].strip()
        return metadata, content, title

    def __sanitize_file_name(self, file_name):
        invalid_chars = '\\/:*?"<>|'
        for char in invalid_chars:
            file_name = file_name.replace(char, '-')
        file_name = file_name.strip().rstrip('.')
        return file_name

    def __add_metadata(self, metadata, file_name):
        metadata_file_path = self.fm.join(self.datalake_dir ,self.
    metadata_dir, file_name)
        self.__writter(metadata_file_path, metadata)

    def __add_content(self, content, file_name):
        content_file_path = self.fm.join(self.datalake_dir, self.
    content_dir, file_name)
        self.__writter(content_file_path, content)

    def __writter(self, file_path, lines):
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write('\n'.join(lines))
```

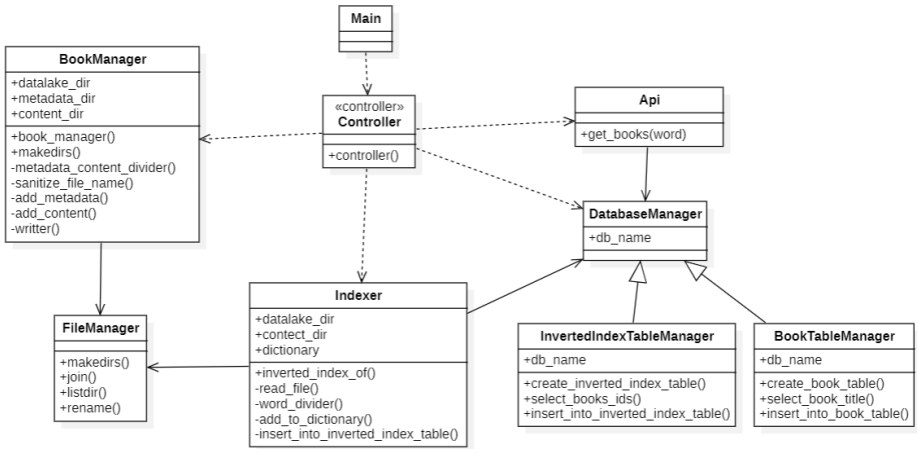
Listing 24: book_manager.py file for the Dictionary implementation

7.3.6 File Manager

```
import os


class FileManager:
    def __init__(self):
        pass

    def makedirs(self, directory):
        try:
            os.makedirs(directory, exist_ok=True)
        except Exception as e:
            print(f"Error creating directory {directory}: {str(e)}"
    )

    def join(self, *args):
        return os.path.join(*args)

    def listdir(self, directory):
        return os.listdir(directory)

    def rename(self, file_path, new_file_path):
        os.rename(file_path, new_file_path)
```

Listing 25: file_manager.py file for the Dictionary implementation

7.4 Class Diagram of SQLite3 Implementation



References

[1] Xinyuan Chang. The analysis of open source search engines. *Highlights in Science, Engineering and Technology*, 32:32–42, 02 2023.

[2] S. Ibrihich, A. Oussous, O. Ibrihich, and M. Esghir. A review on recent research in information retrieval. *Procedia Computer Science*, 201:777–782, 2022. The 13th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 5th International Conference on Emerging Data and Industry 4.0 (EDI40).

[3] Ajit Kumar Mahapatra and Sitanath Biswas. Inverted indexes: Types and techniques. *International Journal of Computer Science Issues*, 8, 07 2011.

[4] Vaibhav Singh and Vinay Singh. Vector space model: An information retrieval system. 07 2022.