Inverted Index
BIG DATA

Grado en Ciencia e Ingenier´ıa de Datos a

Universidad de Las Palmas de Gran Canaria

Inverted index: Crawler Indexer API

Wojciech Dróżdź	wojciech.dro101@alu.ulpgc.es
Zuzanna Furtak	zuzanna.furtak101@alu.ulpgc.es
Olga Kalisiak	olga.kalisiak101@alu.ulpgc.es
Jakub Król	jakub.krol101@alu.ulpgc.es
Max von Massenbach	max@lkdrehberg.de

Profesors: Hernández Cabrera, José Juan Évora Gómez

Important link

Contents

Abstract

In an era of data proliferation, efficient retrieval of relevant information is essential for advancements in science and technology. This project focuses on constructing the core components of a search engine optimized for large-scale textual data, specifically targeting digital books from Project Gutenberg. Our approach centers on the development of a web crawler to periodically download and preprocess documents, and an indexer to organize the content using an inverted index. The inverted index structure facilitates rapid and efficient search capabilities by mapping terms to document locations. Additionally, a REST API is designed to enable seamless query execution and data retrieval, with performance benchmarking employed to compare two indexing implementations—database-based and JSON-based—in terms of speed and efficiency. Benchmarking results indicate that the JSON-based index is faster, whereas the database approach supports complex querying and scalability. This paper presents an analysis of these implementations, recommending the JSON-based approach for projects where rapid indexing is prioritized over complex querying requirements.

Part I

Introduction

In today's world, we're surrounded by an overwhelming amount of data, and being able to effectively process and search through it is key for progress in areas like science and technology. One of the essential tools that helps us navigate this data is a search engine. For this project, we're creating the core components of a search engine: a crawler, which automatically downloads documents, and an indexer, which organizes the data to make searching faster and more efficient. We'll be focusing on a digital book repository from Gutenberg.org as the foundation for this process.

Part II

Problem statement

In the modern era of vast information retrieval systems, search engines play a pivotal role in enabling users to efficiently locate relevant content within massive datasets. The core objective of this project is to design and implement a fully functional search engine capable of indexing and retrieving documents with high precision and speed. The key challenge is to build an efficient system that can process large volumes of textual data while providing fast responses to user queries.

The search engine will be based on the inverted index structure, which is a fundamental data structure used by leading search engines and databases, such as Google, MongoDB, Apache Solr, and ElasticSearch. The inverted index allows for fast lookups by mapping each term or keyword in the corpus to the documents that contain that term. This structure is essential for quick and efficient searching, especially when dealing with large-scale datasets.

The project will encompass the following major components:

- **Crawler**: A module responsible for periodically downloading documents from external sources such as Project Gutenberg. This crawler will navigate through the target website, retrieve books, and store them in a structured document repository for further processing. Efficient crawling is critical to ensure that the latest documents are indexed in a timely manner.

- **Indexer**: The indexer processes the documents retrieved by the crawler, extracting meaningful content and constructing an inverted index. The indexer will also store relevant metadata (such as authors, publication year, and language) in a separate metadata database. Furthermore, the system will avoid indexing stop words (common words such as "the" and "and") to enhance search efficiency.

- **Query Engine**: The query engine is the core component that allows users to perform searches on the indexed documents. It will expose a REST API through which users can submit search terms. The query engine will interact with both the inverted index and the

metadata store to identify and return the most relevant documents based on the user's input.

In addition to building a fully functional search engine, the system will be evaluated using performance benchmarking techniques. Specifically, *pytest-benchmark* will be employed to compare the performance of two separate implementations of the search engine. Given the complexity of multi-threaded environments and the potential interference in CPU and memory measurements, the focus of the benchmarking will be on measuring execution time. This approach ensures that we obtain reliable and meaningful results regarding the system's speed and efficiency without the confounding effects of external factors on CPU and memory usage.

Part III

Methodology

The methodology for developing the search engine will be structured around several key phases, each focusing on a specific aspect of the system's functionality. These phases include the development of the web crawler, the construction of the inverted index, the implementation of the query engine with a RESTful API, and comprehensive performance benchmarking to evaluate the system.

1 Crawler Development

The crawler was developed based on the *requests* Python library. Thanks to the simple url structure of the *Gutenberg* website it's easy to download the books by id.
The format of the urls used to download the books:

```
f"https://www.gutenberg.org/cache/epub/{book_id}/pg{book_id}.txt"
```

Furthermore all of the downloaded data follows a simple structure, where the beginning of the document contains the metadata, and everything after a header **** START OF THE PROJECT GUTENBERG EBOOK* will be the book contents. During testing it was noticed, that the crawler will spend a lot of time establishing connection with the server and processing the data, on top of downloading. Using the 'multiprocessing' libaray and the 'Pool' class we managed to split the

work into multiple concurrent workers, which sped up the process significantly and in the end allowed us to download over a 1000 books per minute.

2 Indexer Creation

The second phase involves developing the indexer, which will process the documents collected by the crawler and build an inverted index. The indexer will:

- **Document Parsing**: Each document will be tokenized, breaking down the text into individual words (tokens). The indexer will extract key information such as word occurrences and their positions within the document.
- **Inverted Index Construction**: The core data structure of the search engine, the inverted index, will map each term to the list of documents where it appears. Two types of indexing will be explored:
  - **Record-level indexing**: This approach will store the references to documents that contain each term, allowing quick lookup of documents.
  - **Word-level indexing**: In addition to storing document references, this approach will store the positions of the terms within each document, enabling more precise querying (such as phrase searches).
- **Stop Word Removal**: Common stop words (e.g., "the", "is", "and") will be filtered out during the indexing process to reduce the size of the index and improve search performance.
- **Metadata Storage**: In addition to the inverted index, relevant metadata (e.g., author, language, publication date) will be stored in a separate database. This metadata will be used to support more advanced search queries.

3 REST API Design

To enable interaction with the search engine, a REST API will be implemented. This API will allow users to submit queries and retrieve results:

- **Search Queries**: Users will be able to submit search terms through the API. The API will then query both the inverted index and the metadata store to find documents that match the search criteria.
  - Term-based search (e.g., returning documents that contain a given word).
  - Metadata-based search (e.g., returning documents by a specific author or within a certain date range).
- **Response Format**: The API will return results in a structured format such as JSON, making it easy for clients to parse and display search results.

4 Performance Benchmarking

To ensure the search engine performs optimally, two separate versions of the system will be implemented and compared. The benchmarking process will include:

- **Implementation Variants**: Two different approaches for the implementation of the search engine will be explored. This could include variations in indexing strategies or optimization techniques.
- **Benchmarking Framework**: *pytest-benchmark* will be used to systematically measure the execution time of each version. Given the complexity of multi-threaded environments and the overlapping factors that affect CPU and memory measurements, we will focus exclusively on timing benchmarks. Measuring execution time provides a clearer, more reliable metric for comparing performance across different implementations.
- **Comparative Analysis**: The results of the benchmarking will be analyzed to identify the strengths and weaknesses of each implementation. Based on these findings, the most efficient version will be selected for deployment.

Part IV

Experiments

5 Crawler

The Crawler is responsible for downloading and processing books from Project Gutenberg and saving them in a local SQLite database. The process can be broken down into several key steps:

5.1 Downloading the book from the server

The `download()` function retrieves the book's content from Project Gutenberg by constructing a URL based on the book's unique identifier (ID), for example, `https://www.gutenberg.org/cache/epub/{book_id}/pg{book_id}.txt`. Then it uses the `download_gutenberg_text()` function to make an HTTP request to fetch the book's plain text file.

5.2 Processing the book's text

Once the book is downloaded, the `extractBook()` function processes the text to extract only the book's content, removing any headers, footers, or licensing information. In parallel, the `extract_metadata()` function extracts metadata such as the author, release date, and language from the book's header section. It uses pattern matching to locate the relevant lines in the text and save these fields.

5.3 Saving the book and metadata

After processing, the `download_and_save_to_db()` function saves both the book content and its metadata into an SQLite database. If the book was successfully downloaded and processed, the content and metadata are inserted into the `books` table. If the download or metadata extraction fails, an error message is printed.

5.4 Batch downloading

The `download_batch()` function allows the crawler to download multiple books in parallel using Python's `multiprocessing.Pool`. This function takes a range of book IDs and downloads them concurrently, saving the results into the database for future indexing.

6 Indexer

The Indexer is responsible for building an inverted index from the books saved by the Crawler. This index allows efficient querying of books by words and getting their metadata as well as book where it appeared and position in the book. We tried two different implementations to benchmark their efficiency: one storing the index in a relational database - SQLite, and the other one storing data in a JSON file.

6.1 Database-based Indexer

The database-based approach stores the inverted index in an SQLite database. Here's a step-by-step breakdown of how it works:

- **Connecting to the database:** The `connect_db()` function establishes a connection to the SQLite database. We use it to connect to *books* database with data saved by Crawler and to *inverted index* where we create the indexed later shared to the API.
- **Creating the database structure:** The `create_database_structure()` function creates three tables:
  - `Words`: Stores unique words found in all books.
  - `WordOccurrences`: Records the occurence of each word in each book, linking to the `Words` and tables and using bookID provided by *Books* database.
  - `Positions`: Tracks the exact positions of word occurrences within the books.
  - We decided to skip the 'Books' table in this database as it would store correlated book ids (one from 'Books' database and local one)

- **Saving the inverted index:** After indexing the books, the inverted index is stored in these tables using the `save_to_database()` function. It inserts each word, the books it appears in, its frequency, and its positions within the text into the corresponding tables.
- **Indexing books:** The `index_books()` function calls the utility function `utils.index_documents()` to create the inverted index, which is then saved in the database.

Example code snippet for database-based indexing:

```
def run_indexer():
    db = connect_db("../databases/books.db")
    if db is None:
        return None
    index_books(db)
    db.close()
    print("Successfully indexed")
```

6.2 JSON-based Indexer

In the JSON-based approach, the inverted index is stored in a JSON file. Here's how it works:

- **Connecting to the database:** As in the database approach, the `connect_db()` function connects to the database to retrieve the book data for indexing.
- **Saving the inverted index to JSON:** Instead of saving the index in a database, the `save_to_json()` function writes the inverted index to a JSON file, storing the word, book ID, frequency, and positions of each word.
- **Indexing books:** The process of indexing the books remains the same. The function `utils.index_documents()` is used to generate the inverted index, which is then saved to a JSON file for later use.

Example code snippet for JSON-based indexing:

```
def run_indexer():
    db = connect_db("../databases/books.db")
    if db is None:
        return None
    index_books(db)
    db.close()
    print("Successfully indexed")
```

7 Query Engine

The query engine acts as the intermediary between the Crawler and the Indexer, providing a system that can efficiently search and retrieve information from the indexed books.

7.1 Crawler-Indexer relationship

The Crawler downloads and processes the books, storing them in a database, which serves as the raw data for the Indexer. Once the books are stored, the Indexer creates an inverted index, either in a database or a JSON file, allowing the Query Engine to perform fast searches. The inverted index maps each word to the books where it appears, along with its frequency and positions, enabling efficient lookups.

7.2 Querying the data

The query engine can retrieve the books and metadata based on keywords using the inverted index. Whether the index is stored in a database or a JSON file, the system allows for querying by word occurrences, making it easy to search for specific terms across the entire book corpus.

8 Comparison

8.1 Indexing

To compare the performance of the two different indexing methods (database vs JSON), benchmarks were run to measure the time taken to index the books. The test code uses the `pytest` benchmarking tool to measure the indexing time for both approaches.

```
import pytest
import sys
import os


sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '../indexer')))
from indexer import run_indexer


def test_indexing_time(benchmark):
    def run_index():
        run_indexer()


    benchmark(run_index)
```

The benchmark results for indexing a set of books were as follows:

- **JSON**: 5.7386 seconds
- **Database**: 11.7330 seconds

8.1.1 Analysis

- **JSON**: The JSON-based indexer performed faster because writing data to a JSON file involves fewer overheads compared to inserting data into a relational database. The JSON approach is simpler and better suited for smaller datasets or applications where fast indexing is needed, but it may not scale well for larger datasets.

- **Database**: The database-based indexer took longer because of the additional steps involved in managing the relational structure, handling foreign key constraints, and ensuring data consistency. However, a database approach offers more flexibility for complex queries, data relationships, and scalability in larger datasets.

8.2 Query Engine

In order to compare the speed of the query engine 100 books were indexed. Then 100 common English words were chosen to test the speed of the engines.

Words chosen for the comparison:

```
    common_words = [
    "that", "have", "for", "not", "with", "you", "this", "but","his", "most",
    "from", "they", "say", "her", "she", "will", "one", "all", "would", "day",
    "there", "their", "what", "out", "about", "who", "get", "which", "when", "make",
    "can", "like", "time", "just", "him", "know", "take", "person", "into", "year",
    "your", "good", "some", "could", "them", "see", "other", "than", "then", "now",
    "look", "only", "come", "its", "over", "think", "also", "back", "after", "use",
    "how", "our", "work", "first", "well", "way", "even", "new", "want", "because",
    "any", "these", "give",
]
```

Following results were archived:

```
-------------------------------------------------------------------------------------------------
Name (time in s)               Min               Max              Mean           Median
-------------------------------------------------------------------------------------------------
test_sqlite_search_time     2.4751 (1.0)      2.4926 (1.0)      2.4825 (1.0)     2.4814 (1.0)
test_json_search_time       3.2497 (1.31)     3.4318 (1.38)     3.3846 (1.36)    3.4246 (1.38)
-------------------------------------------------------------------------------------------------
```

8.2.1 Analysis

Slightly suprisingly, it turned out that searching in SQLite was faster than in JSON. This is most likely due to Python struggling with managing large dictionary - this dicrionary is over 500MB in size and has to be loaded in it's entireity in order to fetch data from it. The problem will get even worse if we continue adding more books. The SQLite hovewer, thanks to its organised structure and implemented SQL indexes allows for much faster on large datasets at the cost of larger database size increased by the indexes.

Part V

Conclusion

9 Conclusions

The comparison between the two indexing methods—JSON and database—highlighted notable differences in performance. The benchmarks showed that the JSON-based approach took 5.7386 seconds, while the database-based approach took 11.7330 seconds to index the same set of books. This significant difference in time can be attributed to the additional overhead involved in managing the relational database structure, such as handling foreign key constraints and ensuring data consistency, which is not required in the simpler JSON format.

The outlook on both methods changes, once the Query Engine performance is taken into consideration. Managing thousands of books would be very difficult, because the entire JSON dictionary would have to be loaded into RAM in order to fetch the data. SQLite does not have such issues as it uses it's query engine to fetch data more efficiently.

One of the future solutions would be using a real non-relation database for storage (for example MongoDB) instead of relying on a simple JSON file. Based on archived results, we have concluded that for future scalability and maintainability SQLite based approach is a better solution, even at the cost of larger storage taken up by the database and noticeably slower indexing time, as we can safely assumed that data will be read from the index much more often than it is written to it.

Part VI

Future work

In the upcoming assignment, our objective will be to refactor the existing Python code for the web crawler, indexer, query engine and API into Kotlin. We will strongly consider modifying the existing non-relational solution to a more robust-industry standard ways of storing such data. Switching to a JVM language will also allow us to get much faster results from our algorithms. We are sure that the code readability and robustness will also benefit from a strongly typed language.