Inverted Index

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

October 8, 2023

Contents

Abstract

Nowadays, we rely on search engines in our daily lives to stay updated on news and pursue our hobbies. It is remarkable how Google consistently provides results, even when our queries are not perfectly articulated. One key process facilitating this is the inverted index, which maintains a list of words and their respective occurrences across the World Wide Web. Enhancements to this process could include features like sorting, targeted field searches, and user experience analysis to refine output for specific inputs.

The objective of our project is to construct a local document search engine based in Python. To achieve this goal, we will utilize a data lake for storing book-related information (original content, cleaned content, and metadata), along with a data mart where we will store results from the inverted index, counting word occurrences in each book. Subsequently, we will design a query to validate the correctness of the inverted index application.

Our approach involves benchmarking various strategies for organizing our datamart and datalake, using a Python-based benchmarking framework to draw comparisons, which will lead to conclude that a database will be more efficient than a JSON file.

1 Introduction

In the ever-evolving landscape of information retrieval, the inverted index stands as a cornerstone technology. It plays a pivotal role in swiftly locating and accessing relevant information within vast collections of documents, making it an indispensable tool for applications ranging from internet search engines to database management systems. This paper embarks on an in-depth exploration of the inverted index, delving into its principles, applications, and optimization strategies.

The inception of the inverted index dates back to the early days of information retrieval systems. Originally conceived as a means to expedite text searches, it has since grown into a fundamental component of modern data retrieval solutions. The proliferation of digital content, accelerated by the internet unprecedented expansion, has rendered traditional search methodologies inadequate. Consequently, the inverted index has emerged as a linchpin technology, enabling efficient and scalable information retrieval across diverse domains.

The relentless growth of digital data continues unabated, presenting a pressing challenge in the era of information overload. As users increasingly expect instantaneous access to relevant content, the need to further optimize the inverted index's capabilities becomes ever more imperative. This study is motivated by the burgeoning demand for enhanced precision and speed in information retrieval systems. By delving into the inner workings of the inverted index and exploring several techniques for its improvement, this paper aims to contribute to the ongoing efforts to meet the demands of today's data-rich environment and empower researchers and practitioners to extract insights and knowledge more efficiently.

2 Project Structure

2.1 Datalake

The Datalake directory is where all the data is stored and separated. It is organized into four distinct directories:

1. **books**: This directory houses unaltered books sourced from gutenberg.org, with stop words intact. Metadata for these books is stored separately in the following directory.
2. **metadata not in json**: Here, you'll find metadata for the books stored in the "books" directory. Notably, this data separation was carried out manually, given the project's primary focus on developing the inverted index, without the use of automated functions.
3. **metadata**: The "metadata" directory contains metadata from the "metadata not in json" directory but in JSON format. This data is instrumental for the query engine, aiding in response generation and the retrieval of crucial book information, including titles.
4. **content**: Lastly, the "content" directory stores processed data from the "books" directory, which has undergone cleaning and processing through the CleanBooks functions.

2.2 CleanBooks

- The CleanBooks module assumes responsibility for processing documents residing in the "books" directory. The processing workflow is structured as follows:

1. Retrieval of documents from the "books" directory.
2. Identification of documents from the "content" directory, followed by a search for duplicates to process only new documents.
3. Parsing of document content, including the elimination of stop words and punctuation marks, as specified in lists within the CleanBooks file.
4. Saving of processed document data in the "content" directory, renaming each file using the unique ID assigned by gutenberg to each book.

The results stored in the "content" directory are the documents utilized by the inverted index, replacing the original documents located in the "books" directory.

2.3 Controller

The Controller module encapsulates essential functions that facilitate the seamless operation of the Main module. These functions include query activation and efficient management of data addition to both datamart implementations. The Controller module assembles various functions from other code files, orchestrates their execution in the correct order, and supplies them with the necessary parameters.

2.4 DatabaseController

This is the class responsible for interacting with an SQLite database named 'datamart-db' and facilitates the addition of columns and data to that table. When an instance of DatabaseWriter is created, a connection to the database is established, and a cursor (self.conn and self.c) is created to interact with it. The `connect_to_db` function is defined but not used in the class. The `create_table` function is used to create the "datamart" table if it doesn't already exist. The `delete_table` function allows the deletion of a specified table using an SQL statement, although it's not used in the class. The `add_columns` method adds columns to the "datamart" table based on the columns provided in the `index_content` argument. It uses `self.sql_writer` to generate the necessary SQL statements. The `insert_data` method inserts data into the "datamart" table from the data provided in `index_content`, also utilizing `self.sql_writer` to generate the SQL statements. The `extract_columns` method retrieves the names of existing columns in the "datamart" table and returns them as a list.

2.5 InvertedIndex

In this section, the series of functions and code work together to build an inverted index from text documents and manage files in a directory. It consists of three functions, and their operation is as follows:

1. `normalize_text(text)`: This function takes text as input and performs the following operations: it converts the text to lowercase, removes punctuation, and eliminates extra white spaces. This helps normalize the text to make it easier to process and search.

2. `inverted_index_from_files_with_count(books_directory)`: This function creates an inverted index with counts from documents in a directory. It iterates through all the files in the directory, reads each file, normalizes its content, and then splits the normalized text into words. It then updates the inverted index to keep a record of which words appear in which documents and how many times. It uses a nested dictionary to store this information.

3. `get_file_paths(folder)`: This function takes a directory as input and performs the following operations: it obtains a list of all items in the directory, filters only the file names, extracts numbers from the file names, creates a dictionary of file paths associating numbers with file paths, and returns the dictionary.

2.6 JsonWriter

The 'JsonWriter.py' file serves as a central repository for various utilities related to JSON file handling. It consolidates functions that can be imported and utilized by other files within the project. The primary focus of this module is to facilitate the creation and manipulation of JSON-based datamarts.

Write Json function

Initially, we perform a check to ascertain the existence of the 'datamart.json' file. If it doesn't exist, we invoke another function named 'create_json()', which we will elaborate on later in this explanation. Conversely, if the file does exist, we proceed by opening it in read mode, loading its content into a variable named 'existing data'.

Subsequently, we employ the provided 'dictionary' argument, which is expected to contain the most recent inverted index, to update the current dictionary. This process involves overwriting existing entries for words that appear in more books and adding new entries for words that were not previously present. Following this update, we open the 'datamart.json' file in write mode and incorporate the updated dictionary using the 'json.dump' function, saving the modified data back to the file.

Create Json function

The purpose of this function is straightforward: it creates the 'datamart.json' file if it is the first time we run our inverted index project. Since there is no previous dictionary information to update, the function directly writes the current dictionary of the inverted index into the file. We have another function called 'json_to_dict()', whichconverts a JSON file into a Python dictionary, simplifying its use within the query engine.

2.7 Main

- The Main module serves as the project's entry point, orchestrating critical tasks:

1. Using a Controller object, it initiates the first implementation, where the datamart operates as a SQLite database.

2. Employing the same Controller object, it launches the second implementation, utilizing a JSON file as the datamart. This file stores the results of the inverted index.

3. The Controller object, again, comes into play to activate a query via an API. This query retrieves content generated by the inverted index. For testing and visualization, we recommend using the Talend API Tester extension in Google.

2.8 Query Engine

A vital component of this project is searching for specific words within documents using the inverted index. To fulfill this requirement, we have opted for a Python REST API library called Flask, which facilitates querying with various parameters. For detailed installation requirements and instructions, please refer to the Flask official website[1][2]. In summary, to run the web service successfully, you will need to:

- Open a terminal in the directory where the file query_engine.py is located.
- Execute the following command: flask –app query_engine run.
- Use an api tester like Talented Api Tester [3]
- Use one of the following endpoints with the address: http://127.0.0.1:5000. Endpoints: /db or /dict.
- Select get method and add values to the field search in the query.

Query structure

To begin, we need to ensure the availability of arguments for the 'search' field. Upon receiving the 'search' value, we will normalize the text by converting it to lowercase and removing any commas. Subsequently, we will generate a list of unique words, using as well the function process_query_params(). This list will serve as a parameter for the get_words function, which will be elaborated upon later. The function will return the query result in JSON format using Flask's 'jsonify' method.

Get the metadata information

In section Get words data from the database, we introduced the function 'get_info()', which is relatively straightforward. Its primary purpose is to traverse all the JSON files containing metadata for the books located in the directory Datalake/Metadata and process them. The function loads the JSON files and converts them into dictionaries, which are then added to a list. Ultimately, this function returns a list of the metadata dictionaries.

Get words data from the datamart

The 'get_words_db()' function returns a dictionary with words as keys and lists of book dictionaries as values.. To establish a database connection, we utilize the 'connect_db()' function located in the 'datamart.py' file.

Additionally, we need to retrieve metadata information from each book. This metadata is stored in a JSON file, conveniently converted to a dictionary.The list of metadata proves valuable for obtaining the book titles based on their IDs, providing users with more descriptive information alongside the book IDs.

Let's consider a scenario where we have limited knowledge about the columns in our datamart, apart from the primary key containing the words. To identify and select the other columns, we'll utilize the 'PRAGMA table_info' command. This command provides insights into the fields of our table, with the column names located in the second field of each result.Once we filter the columns of the books ids, we need to pass this list of column names as an argument to the 'retrieve database_info' function.

Retrieve database information

Now, let's dive deeper into the 'retrieve_database_info()' function. After filtering the columns to contain book IDs, we will enter a loop where we iterate through all the words in our list. Inside this loop, another loop will traverse the columns we've collected. For each column, we will execute an SQL query that selects the value of the current column where we find a row for the current word. If the result of the query is empty or the first element of the contained tuple is 0 (indicating that the word is not used in the current book), we will skip creating a dictionary entry for that book.

[1] Flask Installation: https://flask.palletsprojects.com/en/3.0.x/installation/

[2] Flask Quickstart: https://flask.palletsprojects.com/en/3.0.x/quickstart/

[3] Talented Api Tester: https://chrome.google.com/webstore/detail/talend-api-tester-free-ed/aejoelaoggembcahagimdiliamlcdmfm?hl=en

On the other hand, when our result meets the requirements, we enter another loop for the metadata dictionaries. We check if the ID of the current dictionary matches the column name (column[1:]). When this condition is true, we create a new dictionary where the keys 'id', 'title', and 'count' are set equal to column[1:], the title field in the current metadata dictionary, and the first element of the tuple in our result, respectively.

Finally, we append the dictionary for a specific word and book to the list of dictionaries for the aforementioned word. After we have checked the word for all the books, we add the key of the current word to the resultant dictionary with the corresponding list of dictionaries. This process is repeated for all the words.

Variant of the query for a second datamart implementation

In this scenario, we will employ a function called 'get_words_dict()' to generate a filtered dictionary containing the words from the query that match entries in our JSON datamart. If our filtered dictionary turns out to be empty, it implies that our queried word does not appear in any of the saved books. Conversely, if it's not empty, we will proceed to invoke the 'retrieve_dict_info()' function.

Retrieve json information In this case, we are working with a dictionary of words, where each entry contains another dictionary as its value. This inner dictionary records the book IDs and the frequency of occurrence of the word (the key) within those books. Our task involves traversing all the items within this word dictionary.

Within this traversal, we further iterate through the list of metadata dictionaries. For each dictionary, we check whether its 'id' corresponds to the 'id' within the current word value dictionary (accessible via 'word[1].keys()'). If this condition holds true, we follow a process similar to the one described in Section Retrieve database information to build a new dictionary.

These resulting dictionaries are collected in a list. Once we have checked all the books to determine whether they appear in the current values dictionary, we proceed to add a new field in the output dictionary. This new field has the key corresponding to the word ('word[0]'), and its value is the list of dictionaries generated during our processing. The final output dictionary contains all the words from the query (if they appear in any book), as well as a list of their occurrences.

2.9 SqliteWriter

The 'SqliteWriter.py' file acts as a central repository for a collection of utilities related to working with databases through SQLite. It serves as a hub for functions that can be imported and used by other files within the project. The primary objective of this module is to streamline the process of creating and managing SQLite-based datamarts.

Prepare columns for create table function

The 'prepare_columns_for_create_table()' function simplifies the task of generating a list of column definitions in SQL format, which can be used later to create a database table. This function takes the provided column names and sets their definitions to integers with a default value of 0. The assumption here is that these columns represent books, and by default, they do not contain any of the specified words.

Add columns to table function

The 'add_columns_to_table' function takes a list of column names as its argument. For each column name provided, it generates a Data Definition Language (DDL) statement using the "ALTER TABLE" command to add these columns to the datamart table. The function then returns a list containing these DDL statements.

Get column names function

The 'get_columns_names()' function takes the current columns in the datamart and a dictionary of the inverted index where the book IDs are the keys for each word value in the dictionary. The function returns a list of column names that are not already present in the current datamart columns. This is useful for determining the exact columns to insert without overwriting the existing ones.

Insert columns of function

The 'insert_columns_of' function takes a list of columns from the current datamart and the index dictionary as parameters. With these two parameters, we invoke the 'get_columns_names' function, which we explained in Get column names function. This function returns a list of column names that haven't been added to our datamart table yet.

Next, we utilize the 'prepare_columns_for_create_table()' function, as discussed in Prepare columns for create table function, to create column definitions for storing integers with a default value of 0. The result of this function will be used within the 'add_columns_to_table' function explained in Add columns to table function.

Insert data into table

In the 'insert_data_into_table' function, we receive a list of column names from our datamart and a dictionary of the inverted index (where words are keys and their values are subdictionaries with book IDs and word occurrences). We iterate through the items of this dictionary and also through the list of columns. During each iteration, we create a row dictionary containing the information for the row we intend to insert.

The row dictionary is then supplemented with the number of occurrences of the word for each book, represented by adding keys for the book IDs and their respective counts. We extract this information directly from the values in the inverted index dictionary.

Once we have dictionaries for all the rows to insert as a list, we iterate through each of them and create the SQL insert or replace statements for the datamart. We indicate the columns and use the values from our row dictionary as a tuple. The function returns a list of these insert or replace statements

Each test case defines a reference function, and the 'benchmark' function is used to execute these functions and measure their execution time.

These benchmark tests are essential for evaluating the computational efficiency of our research project and ensuring its optimal performance.

2.10 Test Json

In the 'test_json' module, we conduct performance tests for both the 'clean_books' and 'build_inverted_index' functions. These functions are integral to our research project. The 'clean_books' function cleans a specified directory of books, while the 'build_inverted_index' function constructs an inverted index from books.

The benchmark tests are defined using 'pytest.mark.benchmark,' and each test case defines a reference function. The 'benchmark' function is then used to execute these reference functions and measure their execution time. The 'test_JSON_datamart_performance' benchmark assesses the performance of the JSON datamart implementation.

2.11 Test Database

In the 'test_database' module, we perform benchmark tests for the database implementation. Similar to the 'test_json' module, these tests measure the performance of 'clean_books' and 'build_index' functions, which are crucial for data cleaning and inverted index construction. Additionally, the 'test_database_management' benchmark evaluates the performance of database management tasks.

3 Development of the Inverted Index

The process of creating an inverted index involves several essential steps:

Data Extraction: It begins with the extraction of words and relevant data from a collection of documents. This includes text cleaning to remove unwanted characters and converting text to lowercase.

Tokenization: The text is divided into smaller units called tokens or terms. These terms are used as entries in the inverted index.

Association with Documents: It records which terms appear in which documents. This involves assigning each term to a list of documents where it is found.

Term Count and Frequencies: It keeps track of how many times a term appears in a specific document and how many times it appears in the entire collection. These counts are used to calculate the relevance of a document for a query.

In the ”datamart-builder” project, two variants of the inverted index were implemented:

3.1 Inverted Index with TXT Storage

In this variant, the inverted index data is stored in plain text files (TXT). This implementation has advantages and limitations:

Advantages:

- It is simple and easy to understand.
- It may be suitable for small datasets.

Disadvantages:

- The data structure is flat and not scalable for large document collections.
- File I/O operations can be time and resource-consuming.
- It is not efficient for complex or fast searches.

3.2 Inverted Index with JSON Storage

In this second variant, JSON (JavaScript Object Notation) files are used to store the inverted index. This provides benefits in:

Advantages:

- Data Organization: JSON structures data in a more organized and hierarchical manner than plain text.
- Readability: JSON files are more readable and easier to understand for developers.
- Integration: JSON integrates easily with other systems and programming languages.

Disadvantages:

- Space Consumption: JSON tends to occupy more space compared to storage in plain text format. This can increase storage requirements, especially for large document collections.
- I/O Performance: Reading and writing JSON files can be slower compared to plain text files, impacting performance, especially for frequent read and write operations.

4 Comparison of Indexes

Note: The number at the end means 1: using a database, 2: using a JSON file

Table 1: Results of benchmark ’build_index1’

Name (time in ms)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_inverted_index	233.4579	299.0534	249.5453	28.1009	235.7200	24.9925	1;1	4.0073	5	1

Table 3: Results of benchmark ’build_index2’

Name (time in ms)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_inverted_index_performance	230.7296	265.6095	240.8007	14.1054	236.4878	11.3184	1;1	4.1528	5	1

Table 5: Results of benchmark ’clean_books1’

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS (Kops/s)	Rounds	Iterations
test_clean_books	504.3000	1,373.9000	580.8482	87.2870	543.3000	81.0500	129;67	1.7216	1193	1

Table 7: Results of benchmark ’clean_books2’

Name (time in us)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS (Kops/s)	Rounds	Iterations
test_clean_books_performance	479.8000	1,342.8000	608.7662	128.9346	558.3000	87.6750	134;123	1.6427	977	1

Table 9: Results of benchmark ’db_management’

Name (time in ms)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_database_management	414.0268	445.8717	425.6308	13.0118	422.6648	18.9841	1;0	2.3495	5	1

Table 11: Results of benchmark ’json_datamart’

Name (time in ms)	Min	Max	Mean	StdDev	Median	IQR	Outliers	OPS	Rounds	Iterations
test_JSON_datamart_performance	414.8832	445.8155	427.2087	12.8571	423.0309	20.4356	1;0	2.3408	5	1

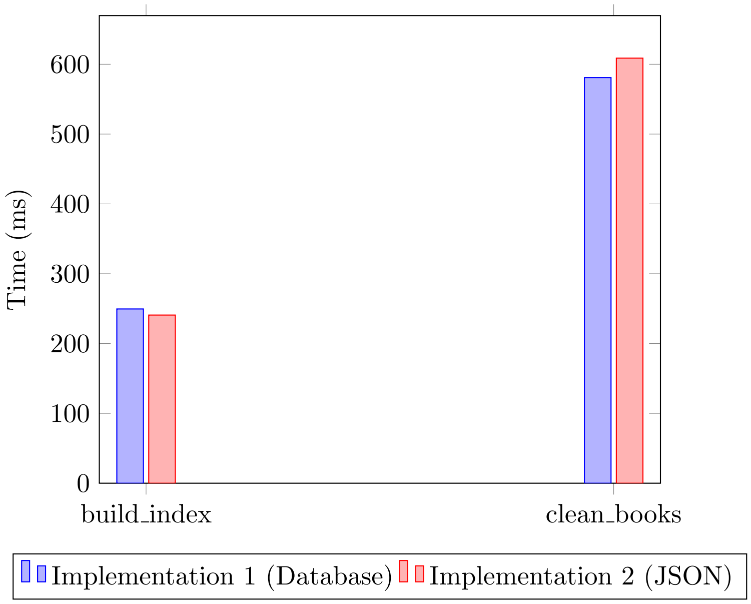

Figure 1: Comparison of Benchmark Times for 'build_index' and 'clean_books'

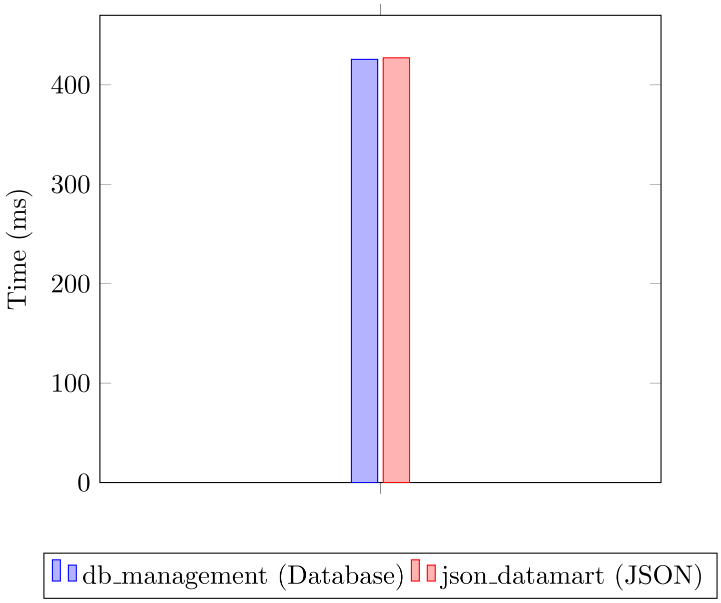

Figure 2: Comparison of Benchmark Times for 'db_management' and 'json_datamart'

5 Conclusion

Based on the benchmark results presented in Tables 1, 3, 5, 7, 9, and 11, as well as the corresponding visualizations in Figures 1 and 2, we can draw the following conclusions regarding the performance of the two implementations: using a database (Database) and using a JSON file (JSON).

5.1 Build Index Performance

When comparing the `build_index` operation between the two implementations, it is evident that the 'Database' implementation outperforms the 'JSON' implementation. The 'Database' implementation has a significantly lower mean execution time (**249.55 ms**) compared to the 'JSON' implementation (**240.80 ms**). This indicates that building an index using a database is faster for this specific task.

5.2 Clean Books Performance

Similarly, in the clean_books operation, the 'Database' implementation exhibits superior performance compared to the 'JSON' implementation. The mean execution time for clean_books with the 'Database' implementation is **580.85 ms**, while it is **608.77 ms** for the 'JSON' implementation. Therefore, clean_books is more efficient when utilizing a database.

5.3 Database Management vs. JSON Datamart

Focusing on the comparison between db_management and json_datamart, it is evident that db_management with the 'Database' implementation (**425.63 ms**) is marginally faster than json_datamart with the 'JSON' implementation (**427.21 ms**). However, the difference is relatively small, suggesting that both approaches perform similarly in this specific context.

In conclusion, based on these benchmark results, it can be stated that, for the given tasks of building an index and cleaning books, the 'Database' implementation consistently outperforms the 'JSON' implementation, indicating that using a database is the more efficient choice. However, when considering db_management and json_datamart, both implementations exhibit comparable performance, with a slight advantage for the 'Database' implementation in terms of execution time. The choice between the two implementations should consider other factors such as ease of implementation, scalability, and specific project requirements.

6 Future Work

In the pursuit of further enhancing our system's capabilities, several promising avenues for future development emerge:

6.1 1. Automation of Gutenberg File Collection and Metadata Separation

A logical progression in our project's evolution would be the automation of the collection process for Gutenberg files. Currently, the process involves manual intervention, which is not scalable for large-scale applications. Implementing an automated mechanism for gathering Gutenberg files and separating content from metadata would significantly streamline data acquisition and preparation. This automation could potentially involve web scraping techniques and natural language processing (NLP) methods to categorize and organize the collected data efficiently.

6.2 2. Implementing Relevance-Based Search Algorithms

To enhance user experience and the relevance of search results, the integration of advanced search algorithms holds great promise. One avenue to explore is the development of algorithms capable of returning results ordered by "importance" based on the user's search query. These algorithms can leverage various factors such as keyword frequency, document popularity, and user behavior to rank search results effectively. Incorporating machine learning and ranking algorithms can provide users with more relevant and context-aware search outcomes.

6.3 3. Multi-Keyword Search Capabilities

Expanding the project's search capabilities to support queries with multiple keywords is another avenue for improvement. Enabling users to input multiple words or phrases and obtaining satisfactory results accordingly can significantly enhance the search system's versatility. Implementing techniques like query expansion and semantic search can aid in this endeavor. Additionally, fine-tuning the system's ranking algorithms to handle multi-keyword queries efficiently would be beneficial.

Incorporating these future enhancements would not only elevate the project's efficiency and user-friendliness but also position it as a more robust and versatile tool for information retrieval and analysis.