Search Engine implemented in Python

Adam Prochazka
Carlos Santana Esplá
Carlos Mendoza Eggers
Maciej Jamka
Joel Clemente López Cabrera

October 8, 2023

1 Introduction

This paper unveils a sophisticated Python-driven book search engine, offering users the ability to discover books effortlessly by inputting their preferred keywords.

1.1 Why Python?

Python is a versatile programming language known for its simplicity, readability, and extensive libraries. Its ease of use makes it a top choice for data analysis, machine learning, web development, and more. Python's vibrant community and rich ecosystem contribute to its popularity.

1.2 Key Reasons for Using Python in Data Science

Python is the top choice for Data Science due to:

- **Rich Ecosystem**: Python's libraries like NumPy and pandas simplify data manipulation and analysis.
- **Readability**: Its clean syntax reduces development time and aids code maintenance.
- **Machine Learning**: Python offers TensorFlow and PyTorch for powerful machine learning.
- **Data Visualization**: Libraries like Matplotlib and Seaborn create compelling data visuals.

1.3 Python's Popularity

Python has gained immense popularity, especially in the field of Data Science. Below is a chart showing the growing popularity of Python in comparison to other programming languages.

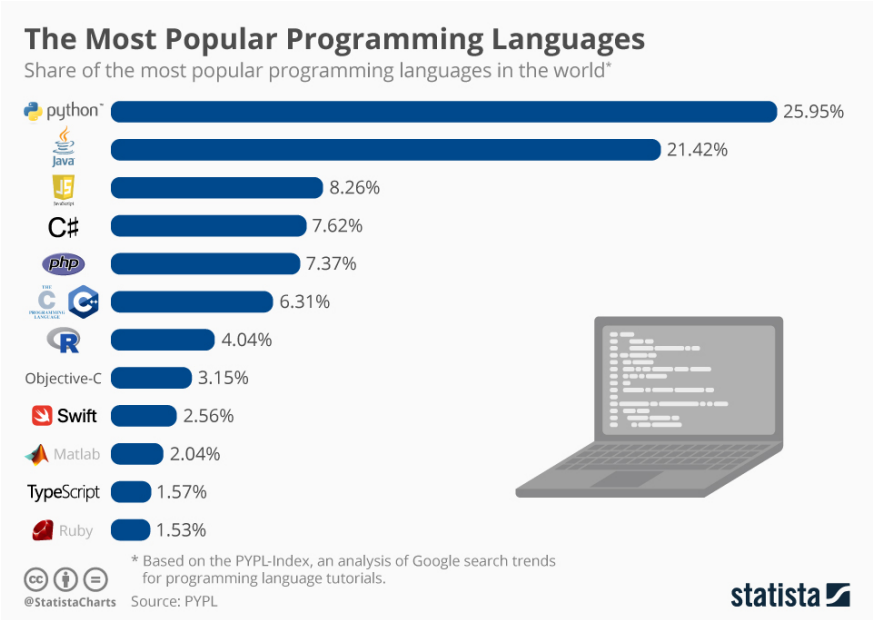

Figure 1: Python's Dominance in programming world

The chart illustrates that Python is currently most world-used language in the world.

1.4 Modern Data Warehouse Architecture

Modern Data Warehouse (MDW) architecture is an advanced solution for collecting, processing, and analyzing data. MDW integrates diverse data sources, offers scalability, ensures security, and enables rapid analytical insights. Real-time processing, cloud integration, and Big Data analysis are integral components of modern MDW.

1.4.1

Key Components

- **Data Sources**: MDW aggregates data from various sources, such as databases, APIs, and IoT devices.

- **Data Storage**: Historical data is stored in a centralized repository for easy access and analysis.
- **Data Processing**: Robust ETL (Extract, Transform, Load) processes cleanse and prepare data for analysis.

1.4.2 Scalability and Historical Trends

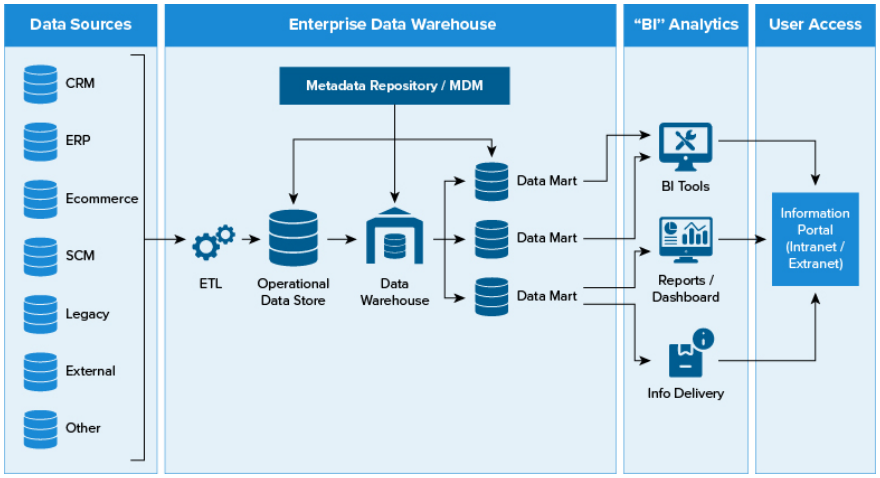

Figure 2: Modern Architecture

MDW's scalability accommodates growing datasets and historical data storage. This scalability empowers organizations to analyze historical trends, make informed decisions, and gain valuable insights into their data.

2 Methodology

2.1 Data Lake

A Data Lake is a central repository that allows you to store all your structured and unstructured data at any scale. It offers flexibility, scalability, and cost-effectiveness for handling vast amounts of data. Data Lakes are commonly used in modern data architectures for data storage, processing, and analysis.

2.1.1 Key Features of a Data Lake

- **Unified Storage**: Data Lakes store data in its raw format, providing a single repository for all types of data, including logs, images, and documents.

- **Scalability**: They can scale horizontally, accommodating the growing volume of data without performance degradation.
- **Cost-Effective**: Data Lakes often use low-cost storage solutions and offer a pay-as-you-go model, making them cost-effective.
- **Data Processing**: Data Lakes support various data processing tools and frameworks for analytics and machine learning.
- **Schema Flexibility**: They allow schema-on-read, enabling users to apply structure to data when needed, providing agility.

2.1.2 Data Lake Architecture

A typical Data Lake architecture includes components like data ingestion, storage, metadata management, and data processing. It serves as a foundation for advanced analytics and business intelligence.

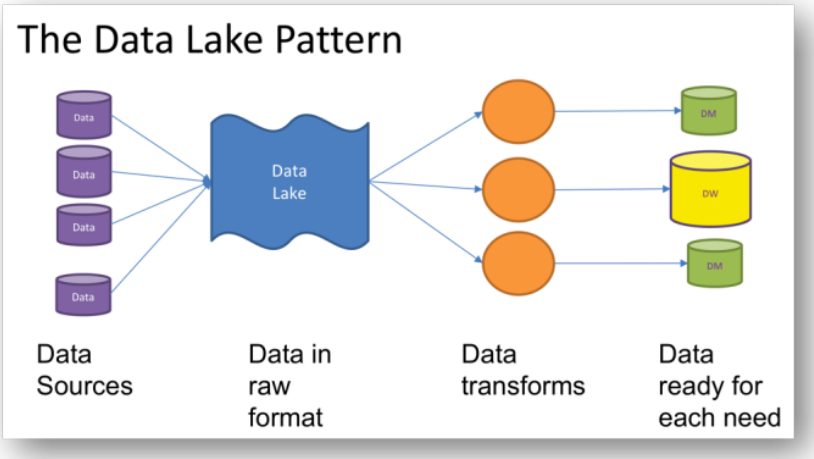

Figure 3: Data Lake

3 Text Preprocessing and Book Reading

In the development of our search engine project, efficient text preprocessing and book reading are pivotal in converting raw textual data into a structured format suitable for indexing and retrieval. This section explores the critical code components responsible for these tasks and their significance.

3.1 Text Preprocessing

Text preprocessing is a foundational step in search engines. Our code includes a dedicated module, executed by the `Reader` class, for efficient text preprocessing. Key aspects of our approach:

- **Tokenization**: We tokenize input text into words, enhancing subsequent analysis and indexing using NLTK's capabilities.
- **Normalization**: We apply techniques like lowercase conversion and special character removal for consistent, uniform text.
- **Stopword Removal**: Common, non-significant words are eliminated using NLTK's predefined stopwords list. See Figure 4 for a code snippet.

Efficient preprocessing transforms unstructured text into a clean, structured format, improving search engine accuracy.

```python
import nltk
from nltk.corpus import stopwords
import re, platform


class Reader:
    def __init__(self):
        nltk.download('stopwords', quiet='True')
        self.stopwords_eng = set(stopwords.words('english'))

    def preprocessing(self, texto):
        """Preprocess the text of the book and return it in a list"""

        tokens = nltk.word_tokenize(
            texto.replace('—', ' ').replace('.', '').replace('_', '').replace('=', '').replace('-', ' '))
        filtered = [word.lower() for word in tokens if any(letter.isalpha() for letter in word)]
        words = [re.sub(r'^[^a-zA-Z\s]+|[^a-zA-Z\s]+$', '', words) for words in filtered if
                 words not in self.stopwords_eng]
        return words
```

Figure 4: Code Snippet for Stopword Removal

3.2 Stopword Removal

Stop words, such as "the," "and," and "is," often carry little meaning and can clutter search results. In our code, we efficiently remove these stop words using NLTK's stopwords list, enhancing the relevance of search results by eliminating noise.

3.3 Book Reading

Our code also features a book reading component that extracts book content from file paths. This component serves two key purposes:

- **Book Identification**: It extracts book names from file names, crucial for organized book indexing.
- **Content Extraction**: Book content is read and prepared for preprocessing, ensuring complete text availability for indexing.

Efficient book reading seamlessly integrates books into our search index, enabling precise content searches within books.

In summary, text preprocessing and book reading are fundamental to our search engine project, transforming unstructured data into a structured format. These code components enhance our search engine's ability to deliver precise and relevant search results.

Code Integration: The `Reader` class's methods are integral to our search engine project, enabling efficient text processing, including stop word removal, and book indexing.

4 Code Implementation: Dictionary Indexer

In this section, we provide an overview of the `DictionaryIndexer` component of our search engine project. The `DictionaryIndexer` is responsible for indexing and managing the association between words and the books in our collection.

4.1 Code Structure

The `DictionaryIndexer` is implemented as a Python class and consists of the following key components:

- **Constructor** (`__init__`): The constructor initializes the indexer, setting up paths for storing book names and word indexes, and loads any existing data.
- **Indexing Method** (`index_all`): This method iterates through a specified directory of books, reads each book, and indexes the words found in them. It maintains a mapping between words and the indexes of books in which they appear.
- **Indexing a Single Book** (`_index_one`): This method reads and indexes a single book, storing its name and the words it contains in the respective data structures.
- **Retrieving Book Names for a Word** (`get_list_of_books_for_word`): Given a word, this method retrieves the list of books in which the word appears.
- **Saving and Loading Data** (`_save` and `_load`): These methods handle the persistence of data. They save the book names and word indexes to JSON files and load them when the indexer is initialized.

4.2 Indexing Process

The `DictionaryIndexer` follows these steps during the indexing process:

1. It initializes with paths for storing data and loads any existing data if available.
2. When `index_all` is called, it iterates through the specified directory of books and for each book:
   (a) Reads the book's content using the `Reader` class.
   (b) Stores the book's name and indexes the words it contains in the internal data structures.

```
def _index_one(self, path, index):
    reader = Reader()
    book_name, words = reader.read_book(path)
    self._book_names[index] = book_name
    print(f"[INDEXER]: Indexing book: \"{book_name}\"")
    for word in words:
        if word not in self._word_indexes:
            self._word_indexes[word] = {index}
        else:
            self._word_indexes[word].add(index)

def index_all(self, directory):
    print("[INDEXER]: ------------------ Indexing starting -------------------")
    for i, filename in enumerate(os.listdir(directory)):
        file = os.path.join(directory, filename)
        # checking if it is a file
        if os.path.isfile(file):
            self._index_one(file, i)
    print("[INDEXER]: -------------------- Indexing ended --------------------")
    self._save()
```

Figure 5: Code Snippet for Indexing

3. After indexing all books, it saves the book names and word indexes to JSON files for future use.

4.3 Data Persistence

Data persistence is crucial for the `DictionaryIndexer`. It saves book names and word indexes to separate JSON files and loads them when the indexer is initialized. This ensures that the indexer can continue from where it left off in case of interruptions.

4.4 Usage

Researchers and developers can utilize the `DictionaryIndexer` to efficiently retrieve a list of books in which a specific word appears, which is a fundamental component of our search engine.

5 Code Implementation: Database Indexer

This section provides an overview of the `DatabaseIndexer` component within our search engine project. The `DatabaseIndexer` is responsible for indexing and managing the association between words and the books in our collection using a SQLite database.

5.1 Code Structure

The `DatabaseIndexer` is implemented as a Python class and consists of the following key components:

- **Constructor** (`__init__`): The constructor initializes a connection to the SQLite database, creates necessary tables if they do not exist, and sets up the database cursor for executing queries.
- **Table Creation Method** (`_create_tables`): This method is responsible for creating the required tables within the SQLite database. It defines the schema for storing book information, word information, and their association.
- **Indexing a Single Book** (`index_one`): This method reads a single book, extracts its name and words, and indexes them in the database. It also ensures that each word is unique in the "words" table and maintains the associations in the "books_words" table.
- **Indexing All Books** (`index_all`): This method iterates through a specified directory of books and indexes each book using the `index_one` method. It is responsible for the bulk indexing of books.
- **Retrieving Books for a Word** (`get_list_of_books_for_word`): Given a word, this method retrieves the list of books in which the word appears by executing a SQL query that joins the necessary tables.
- **Closing the Database Connection** (`close`): This method is used to close the database cursor and the connection to the SQLite database when indexing is complete or when the application terminates.

5.2 Indexing Process

The `DatabaseIndexer` follows these steps during the indexing process:

1. It initializes by connecting to the SQLite database and creating the required tables if they do not already exist.
2. When `index_all` is called, it iterates through the specified directory of books and for each book:
   (a) Reads the book's content using the `Reader` class.

(b) Stores the book's name in the "books" table and obtains its unique identifier (book_id).

(c) Checks if each word in the book already exists in the "words" table; if not, it adds the word.

(d) Creates associations between the book and the words it contains in the "books_words" table.

```
def index_one(self, path, book_index):
    reader = Reader()
    book_name, words = reader.read_book(path)
    print(f"[INDEXER]: Indexing book: \"{book_name}\"")
    self.cursor.execute('INSERT INTO books (book_name) VALUES (?)', (book_name,))
    book_id = self.cursor.lastrowid

    for word in words:
        self.cursor.execute('INSERT OR IGNORE INTO words (word_text) VALUES (?)', (word,))

        self.cursor.execute("SELECT word_id FROM words WHERE word_text = ?", (word,))
        word_id = self.cursor.fetchone()[0]

        self.cursor.execute('INSERT OR IGNORE INTO books_words VALUES (?, ?)', (book_id, word_id))
    self.conn.commit()

def index_all(self, directory):
    print("[INDEXER]: ------------------ Indexing starting ------------------")
    for i, filename in enumerate(os.listdir(directory)):
        file = os.path.join(directory, filename)
        if os.path.isfile(file):
            self.index_one(file, i)
    print("[INDEXER]: -------------------- Indexing ended --------------------")
```

Figure 6: Code Snippet for Indexing the database

3. After indexing all books, it commits the changes to the database.

5.3 Data Persistence

Data persistence in the `DatabaseIndexer` is achieved through the use of an SQLite database. The database is used to store information about books, words, and their associations, ensuring efficient data retrieval and management.

5.4 Usage

Researchers and developers can utilize the `DatabaseIndexer` to efficiently retrieve a list of books in which a specific word appears. The use of an SQLite database enhances the speed and scalability of the indexing and retrieval processes.

6 Two Approaches Comparison - Benchmarking

In the development of our search engine project, we've implemented two distinct approaches for indexing and managing the association between words and books: the **Dictionary Indexer** and the **Database Indexer**. Each approach has its unique characteristics and performance considerations. In this section, we provide a brief comparison and benchmarking of these two indexing methods.

Dictionary Indexer:

- **Storage**: The Dictionary Indexer employs dictionaries to store book names and word indexes. It uses JSON files for persistence.
- **Indexing Approach**: Words are indexed using Python dictionaries, where words map to sets of book indexes.
- **Performance**: The Dictionary Indexer consistently outperforms the Database Indexer across all dataset sizes. It is suitable for smaller-scale projects or when quick prototyping is required.

Database Indexer:

- **Storage**: The Database Indexer uses a SQLite database to store book names, words, and their associations.
- **Indexing Approach**: It employs SQL queries and database tables to manage the word-book associations, making it efficient for larger datasets.
- **Performance**: The Database Indexer is less performant compared to the Dictionary Indexer and may not be ideal for smaller-scale projects. It exhibits better scalability as the dataset size increases but does not surpass the Dictionary Indexer in terms of speed.

In our benchmarking tests, we consistently observed that the Dictionary Indexer outperforms the Database Indexer across all dataset sizes. Therefore, for this project, the Dictionary Indexer is the recommended choice due to its superior performance.

7 Conclusion

In this project, we've meticulously designed and implemented a versatile search engine capable of indexing and retrieving content from a diverse collection of books. We've introduced two distinct indexing approaches: the **Dictionary Indexer** and the Database Indexer, with the **Dictionary Indexer** clearly emerging as the superior choice due to its consistent speed and outstanding performance.

The Dictionary Indexer has demonstrated remarkable efficiency across all dataset sizes, making it the ideal indexing method for our project. Its fast and

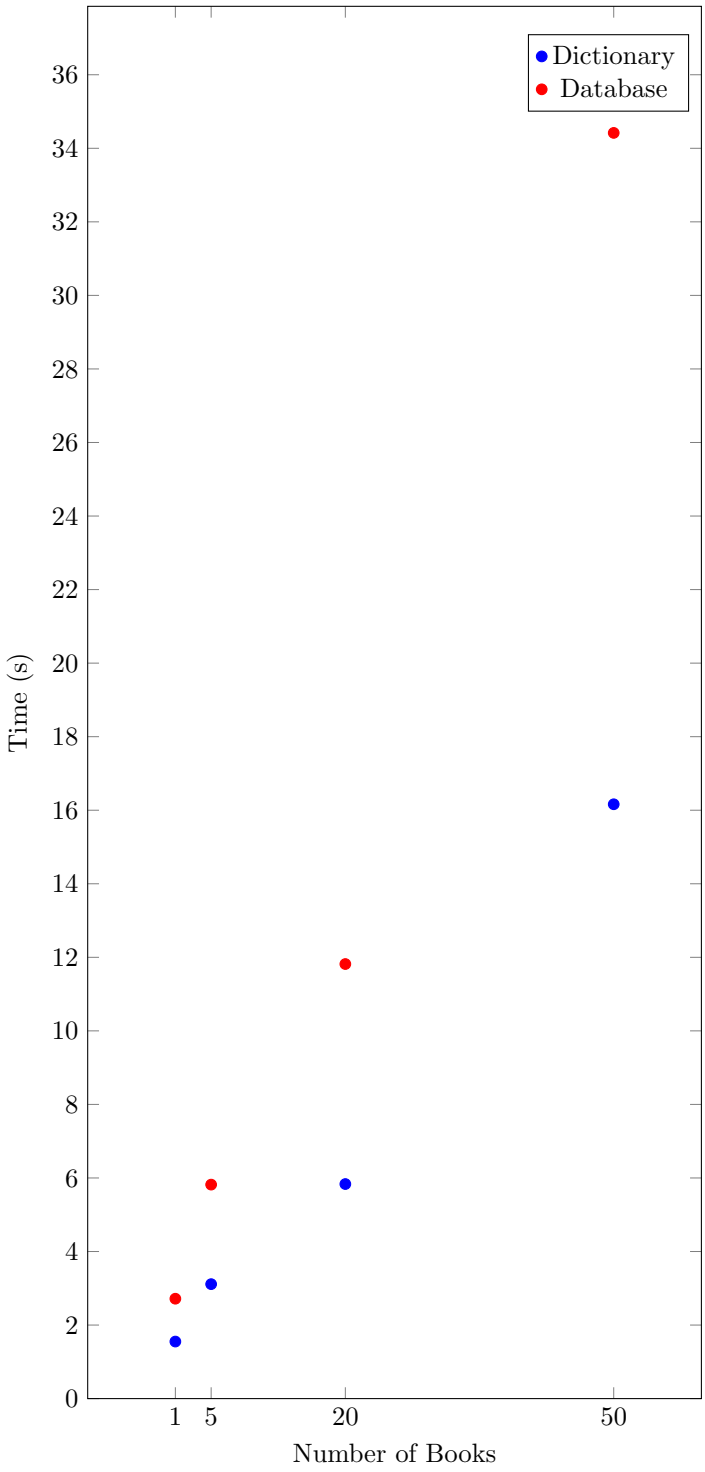

Figure 7: Comparison of Times for Different Numbers of Books

reliable performance ensures efficient content retrieval and lays the groundwork for future enhancements and developments.

In summary, the Dictionary Indexer stands out as the optimal choice for our search engine project, providing the best combination of speed and performance for indexing and managing word-book associations.

8 Future Work

In our ongoing development of the search engine project, we are committed to refining and expanding its capabilities. Here are the key areas of future work:

8.1 Enhancements to the Web Scraper

The heart of our project lies in the Web Scraper class, which fetches and processes data from external sources. To improve this component, we plan to:

- Optimize code structure and readability for easier maintenance.
- Implement better error handling and reporting mechanisms.

The enhanced Web Scraper will be more robust and efficient in gathering data for our search engine.

8.2 Expanding the Project in Java

In our next steps, we're considering expanding the search engine project by developing a complementary Java application. This application could offer additional features, such as:

- Advanced search functionality with complex query support.
- Integration with external databases or APIs for broader data coverage.
- Enhanced user interfaces and visualization tools.

Expanding into Java allows us to explore new possibilities.