Search Engine in Java

Jorge Rey, Carlos Mendoza, Carlos Esplá, Adam Prochazka, Maciej Jamka

November 5, 2023

Abstract

Search engines are very powerful tools that allow users to enter queries and retrieve data in the form of web pages, documents, images, etc. These search engines can implemented using the java programming language. To do this there are several methods and classes we need to take into account, such as a crawler, a datamart, an API, an indexer... All of which will be shown in this paper, as well as the overall performance comparison to the previous implementation in Python.

1 Introduction

This research paper will dive into the implementation of a search engine in Java using a crawler. The search engine will work on a library of over 60,000 books.

1.1 Background

Search engines are powerful software systems designed to help users discover information on the internet and other online sources. They work by indexing vast amounts of content, and therefore handle huge amounts of data, requiring efficient processing and indexing capabilities. When users enter queries, search engines match these queries with indexed content and retrieve the results, which are then ranked depending on how well they match the search terms.

A crawler is an automated software program that navigates the internet, visiting web pages and collecting data from them. Search engines use crawlers to create and update their indexes, ensuring that users can find the most current and relevant content when they perform online searches.

1.2 Motivation

For this project, we are going to implement a crawler that periodically downloads books from a library of books(Project Gutenberg) and uploads them to the document repository. Then we will develop a search engine that for any given query, in this case words, retrieves all the books that contain said words.

This research paper aims to demonstrate the implementation of this search engine in Java, utilizing a crawler to index the collection of books. By efficiently processing and

indexing the provided dataset, our objective is to enable users to retrieve relevant books containing specific words or queries. Through the use of Java's robust performance, scalability, and concurrency support, we aim to create an effective and responsive search engine for information retrieval. And it also aims to compare the performance of this implementation of the search engine in Java to the previous one we already did in Python.

2 Methodology

Now we will dive into the implementation of the project. For starters, we will have the following modules, each with their own classes and methods:

- Crawler
- Datalake
- Datamart
- Indexer
- SearchQuery

For brevity purposes, most code snippets can be found on the github repository.

2.1 Crawler

In the module crawaler is were we will access the library to download, periodically, random books to send to the Datalake.

The module crawler has the following structure:

- ContentManager
- Controller
- Downloader
- Main

2.1.1 ContentManager

ContentManager contains 3 methods, and its purpose is to retrieve the book titles and their content.First it retrieves the title of the book using the method **getBookTitle()** and then the method **cleanFilename()** removes unwanted characters or patterns. The final method **getBookContent()** retrieves the content of the books by using a matcher and regular expressions to determine where the content starts.

2.1.2 Controller

Controller is resposible for determining the time period between the downloads of books. It manages this schedules using **´ScheduledExecutorService´** interface.

Then the Controller class is declared with two instance variables: **DataLake dataLake** and **Downloader downloader**. And the constructor then intializes the Datalake and downloader.

Moreover, we find the **start()** method, in which the **´executor´** (a "ScheduledExecutorService") calls for the **downloader** to run with a certain schedule, in this case, once every 1 minute using **scheduleAtFixedRate()**.

See Github repository for code.

2.1.3 Downloader

Downloader is responsible of accessing the Project Gutenberg and downloading the books given the **Controller** schdule into the DataLake.

We use **Jsoup** library for making HTTP connections and parsing HTML content. Our class **Downloader** is declared with instance variable **Datalake datalake**.

The **run()** method generates a random book url to download, checks if it has already been downloaded and if not, calls the **downloadBook()** method.

The **downloadBook()** method takes a book URL and its identifier, and retrieves its content by calling for **ContentManager.getBookTitle(), ContentManager.cleanFilename()** and **ContentManager.getBookContent()**. Then all the book contents are saved into the DataLake with the **dataLake.saveToFile()** method.

See Github repository for code.

2.1.4 Main

The **Main** class sets up the necessary components for the application by creating instances of the **DataLake and Controller** classes. It then starts the application by calling the **start()** method of the **Controller** instance, which kicks off the scheduled downloads and begins execution.

See Github repository for code.

2.2 DataLake

The module DataLake has the following structure:

- Book
- BookPersistance
- DataLake
- Reader

2.2.1 Book

Book is a very simple class which provides the structure for representing the books. It provides the instance variables **count**, **index**, **name** and **words**(a list). Then we initialize a **´Book´** variable with a unique index obtained by increasing the value of the static variable **count** by one for every new book. And finally we define the getter methods for the name, words and index.

See Github repository for code.

2.2.2 BookPersistance

BookPersistance class provides methods for loading and saving book data to a JSON file using the Jackson library. Our instance variable is **path**, which represent the path were the Json file will be stored. Then we initialize **path** with the specified directory path.

Furthermore, we have method **load()**, that reads data from the Json file and returns a **Map¡Integer, String¿** after checking if the file exists, if it doesnt, it initializes an empty Hashmap and returns it.

Finally, we have a **save** method that takes the **Map¡Integer, String¿** and converts it to a Json string, to be written in **books.json** in the specified directory.

See Github repository for code.

2.2.3 DataLake

DataLake class serves as a central storage and management component for book data. It provides the variables **path** , **booknames** and an instance of **BookPersistance**.

It provides several methods, like **addBook**, **creatFolderIfNotExists**, **isBookInDataLake**, **saveToFile**, **getTitle** and **getDataLakePath**. These are all very simple and self explanatory. *See Github repository for code.*

2.2.4 Reader

Reader class is responsible for reading books, preprocessing their content, and extracting book titles. Here we have the method **readBook**, which basically reads the content of a book, preprocesses it and extracts its title by using **extractBookName**. We also use stopwords, which are words that do not seem useful for any query, so when the method encounters one it does not add it to the arraylist of words. *See Github repository for code.*

2.3 DataMart

This module is arguably the most important of the project, given that the performance of the search engine depends highly on the implementation of the DataMart. We opted for using files to manage inverted index of words. We have the following interfaces and classes:

- DataMart

- FileDataMart
- BookIndexWordPersistence
- ListPersistenceFacade
- FileListHandler
- FileRepository

2.3.1 DataMart

DataMart interface, which outlines the methods required for managing the inverted index of words in a collection of books.

We have the **getIndexOf** method retrieves the inverted index for a specific word, and the **addBookIndexToWord** allows us to add book indexes to words.

See Github repository for code.

2.3.2 FileDataMart

We have the instance variables **UniqueIntegerListInserter uniqueListInserter** and **BookIndexWordPersistence bookIndexWordPersistence**.

On the other hand we have the methods **getIndexOf** and **addBookIndexToWord**. *See Github repository for code.*

2.3.3 BookIndexWordPersistence

BookIndexWordPersistence class is a specialized implementation of the general **ListPersistenceFacade**, and it inherits the basic functionality for persisting and retrieving lists of data structures (in this case, book indexes) to and from files on the specified path with the given file extension. *See Github repository for code.*

2.3.4 ListPersistenceFacade

ListPersistenceFacade class acts as an interface for the client code, providing methods for reading and saving lists of data (book indexes) associated with specific words.

It provides the methods **getBookIndexesOf** and **saveBookIndexesOf** using instances of **FileRepository fileRepository** and **FileListHandler listPersistence** *See Github repository for code.*

2.3.5 FileListHandler

FileListHandler class provides methods for reading and saving lists of integers to and from files. As said, we have the **read** method and the **save** method, which basically read and save lists of integers respectively using serialization and deserialization. *See Github repository for code.*

2.3.6 FileRepository

FileRepository class manages the creation and retrieval of files in a specified directory. It provides methods to create files and to find files, these are **create** and **get** respectively. *See Github repository for code.*

2.4 Indexer

The **Indexer** class is responsible for indexing books and adding their word indices to the DataMart. It initializes one instance from Datamart and one from Datalake.

It also provides method **indexOne** that reads a book from a specified path and using the **´Reader´** and calls the **´addBookIndexToWord´** to add the word index to the DataMart. And method **indexAll**, that indexes all books in the directory by iterating through the files and calling **indexOne**. *See Github repository for code.*

2.5 SearchQuery

Here we find the following classes:

- CLI
- QueryMethods

2.5.1 CLI

The **CLI** class serves as the interface through which users can interact with the search engine. We first have to import a scanner, then what we basically have is a loop in which the program will ask for queries until the user enters ":q". If the query is not ":q" then the method **queryWord** from **QueryMethods** is called. *See Github repository for code.*

2.5.2 QueryMethods

QueryMethods class provides a method to query the search engine based on the imputted word. In this class, we initialize a **DataMart** object, as well as a **DataLake** object. Our method **queryWord** takes a word as an input, retrieves the inverted indexes of the input word from the dataMart object. For each index in the retrieved list, it retrieves the corresponding book title from the dataLake object using the **getTitle** method and prints it. *See Github repository for code.*

3 Results

To make a comparison between both implementations we did some benchmarking to see the difference in performance of the two when indexing books. These were the results: .

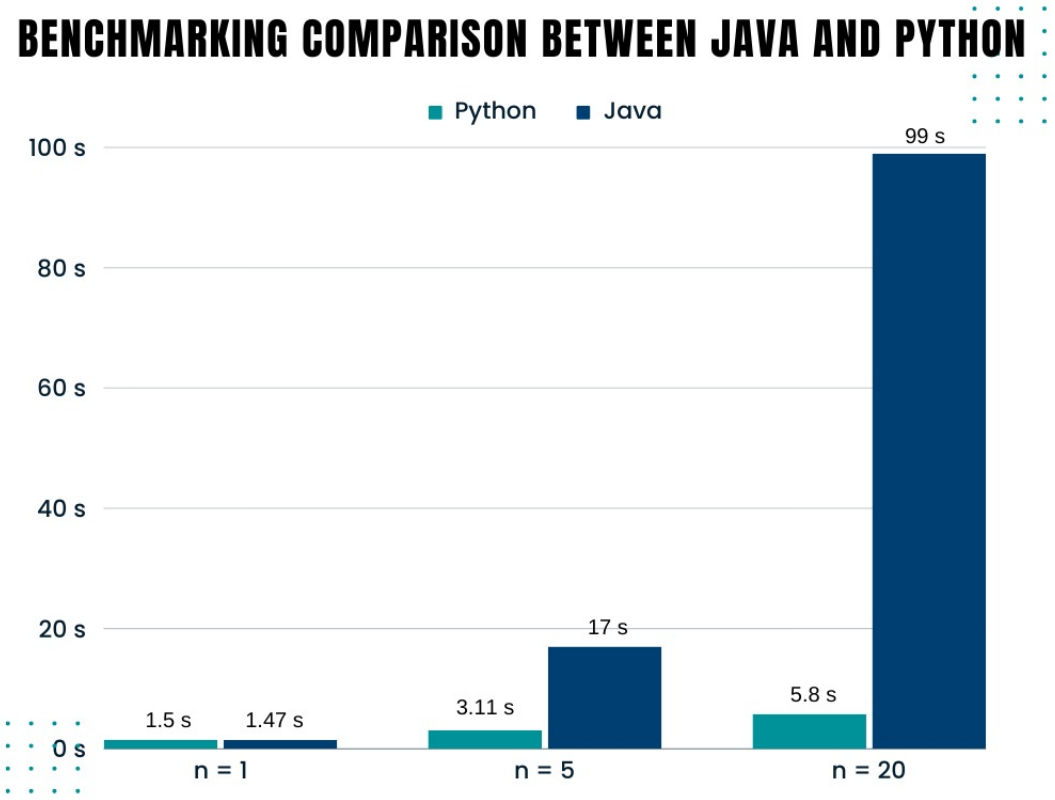

Fig. 1

As you can see, our implementation in Java is faster for one book only, however, with more books, python becomes much faster than java for our current implementations. This could possibly be due to our Datamart implementation, as with more books, more files are recquired to be opened and iterated through.

4 References

https://github.com/adam-prochazka/search-engine