Programming assignment 3

Aljaž Koželj

Fakulteta za računalništvo in informatiko
Univerza v Ljubljani
ak6689@student.uni-lj.si

1 Introduction

In the final programming assignment we were asked to implement text querying. We were given a set of web pages from which we had to build a inverted index. By using the inverted index, we can implement speedy querying.

2 Technologies

2.1 Java

Java was selected as the programming language for the project. It was chosen mainly for it's familiarity, strong library (and framework) support and performance. It also has a strong concurrency support, which was needed for this project.

2.2 Spring Boot

Spring Boot is the main reason that Java was chosen. It's a pre-configured Spring framework which allows us to easily build stand-alone, production-grade applications. This allows us quick setup of features like logging, object-oriential mapping and dependecy injection. These features make development faster and easier. Business logic and data manipulation beans are created behind the scenes and the developer doesn't have to bother with that. The Spring Data module, for example, provides basic CRUD operations, over an entity, by simply injecting an interface.

2.3 PostgreSQL

The project demanded the use of the PostgreSQL database for storing the data. The schema for it was provided, but we were allowed to add additional stuff. The database was ran inside a docker container. The command is provided in the README.md file.

2.4 Maven

Maven is a build tool for Java projects. It provides a straightforward way of adding new libraries and building the project with them. By building the Spring boot project it provides us with a runnable .jar.

2.5 Other libraries used

Table 1. List of other libraries

Google Guava	A Java utility library. We use it for data structures and for loading of web page content.
JSoup	A Java library that allows work with HTML. We use it to extract text from web page.
Apache OpenNLP	A library that allows for natural language processing in Java. We use it for it's tokenization feature.
StreamEx	A library that enchances Java 8 streams. It provides additional methods and is more readable than vanilla streams.

3 Program Structure

The program is accessible via a REST API. A basic API provides the user all necessary endpoints that could be used for triggering of inverted index building and search queries.

3.1 IndexerRestController.java - rest controller

In this rest controller we map rest api requests to the appropriate service methods. If the methods return string content, proper formatting is also applied.

The rest endpoints are:

- **GET /loadFiles** - This request triggers the indexing of the web pages.
- **POST /search** - Returns the results of the query. Requires the query JSON payload (see below).
- **POST /search-slowly** - Same as the /search endpoint, while using a slower method. Requires the query JSON payload (see below).

Query JSON payload:

```
{
    "query": "Sistem SPOT"
}
```

3.2 FileImporter.java - service

This service is responsible for working with web page files. It mainly does loading of local files and their indexing.

3.3 SearchEngine.java - service

This service is responsible for generating query results. Given the search terms and the database index, it constructs the best matching results. It provides us with a quick method, that utilizes indexing, and a naive slow method for querying.

4 Implementation details

4.1 Indexing

First, we will cover the details of how the inverted index is constructed. As mentioned above, this is triggered by sending a GET request to the /loadFiles endpoint.

First we load web page file paths into memory. We do this by providing a data directory, that is then recursively searched for all HTML web pages (files ending with .html).

We also load Slovenian stop words. These are included in the sl.txt file in the resources folder. We retrieved this file from www.stopwords.org .

Next, we iterate over all file paths. We load their contents with Google Guava and extract the text with Jsoup. This text is then tokenized by using Apache OpenNLP's SimpleTokenizer, which seems to work similarly to the Python solutions. We iterate over all tokens, convert them to lowercase and remember their index in the tokenized array. While doing this, we ignore all words that are stop words. WThis results in a map of words mapping their indexes. We use this to create new index words entries in the database and new posting entries.

4.2 Search query (fast)

The program receives a query request via the REST api and the body payload. The query entry is split by whitespace and each querying term is equivalent to the others.

First we want to retrieve the top 5 web pages with most matches. Consider one match as one search term appearing in the web page text once. We achieve this with a few SQL queries, that can be seen in PostingRepository.java .

This provides us with multiple posting entries. We convert those into indexes (that are split by ',') and merge them my their document names. This leaves us with a map of document names mapping to word indexes.

We know the names of our result documents and the number of matches, but we still need a snippet. Therefore we load the content of the target web pages, so we can build the snippet.

Snippet building: First, we iterate over each list of indexes and group them, if they are close enough. According to the instructions, we want to capture the neighbourhood of 3 words, together with the search term. We do this by converting every index into a number span as following: index 7 –> (index 4, index 10). While doing this, we merge neighbours if their bound intersect: (4,10) + (8,14) -> (4, 14). We do this, so we get a cleaner snippet output without any duplicated words.

We end up with a list of index spans, that we will use with the tokenized web page content in order to extract the correct text. The provided SimpleTokenizer has a neat feature, where it does the tokenization, but it returns the substring indexes of the text and not the strings themselves.

So what we do is the following: Take an index span, that we built above. Take the start and end indexes. Using the start index, retrieve the corresponding substring indexes span. Do the same for the end index. Extract a substring using the substring start index and the substring end indexes.

We do this for all index spans. Then we merge them into a single string with '...' between them. If necessary, we put '...' at the start and/or the end of the snippet.

We return the results and those are then formatted into a table. For the purpose of this assignment, all snippets are limited to a width of 95 characters.

4.3 Search query (slow)

We also implemented a slow query, for the sake of testing and demonstration. This method is very similar to the one above, the only difference is how the indexes are found.

Same as above, we receive a list of search terms. Then we load stop words and all file paths. For each file path we load the text content and tokenize the text. Then we save the resulting indexes of the search terms occurring in the tokenized list. While doing this, we make sure that no stop word sneaks in.

After that, we keep the documents with the most occurring indexes and load their text content again. Then, we repeat the same snippet building process as described above.

5 Results

5.1 Index building

Here are some stats from building the index database. Do note, that the tokenizer allowed for tokenization of punctuation marks. Those are thus included in the database. They are of course separate, but they represent a large number of entries. Search queries with them have a skewered bias towards documents with more punctuation marks. This problem would probably have been solved while filtering for stop words.

Table 2. Index building stats

Indexing time	1671,7 seconds (28.85 minutes)
Number of index word entries	43587
Number of posting entries	416406

5.2 Querying results

For the purpose of the assignment we were already provided with a set of queries we had to test on. We also had to provide a few of our own. These are our selected queries:

– rekreacija
– rekreacija social services
– rekreacija social services vrt

We ran fast and slow queries for the 6 search terms. The results were save into the query-results folder at the root of the repository. Here are the time benchmarks:

Table 3. Index building stats

Query	Fast method time	Slow method time
“predelovalne dejavnosti”	283ms	2711ms
“trgovina”	231ms	2512ms
“social services”	100ms	2362ms
“rekreacija”	228ms	2555ms
“rekreacija social services”	244ms	2539ms
“rekreacija social services vrt”	248ms	2550ms

6 Conclusion

As we can see from the above time benchmarks, the time gains are pretty considerable. It also demonstrates why an inverted index is crucial in the search engine business, since no one would use Google if the results took seconds to load.

The implementation works pretty well, but there is more work to be done. The stop words especially need more love. We mentioned the punctuation mark issue above.