Project: Search Engine

Group: ip18groupR

IT UNIVERSITY OF COPENHAGEN

Group name: Busca.*
Introductory Programming
Master of Science in Software Development
IT University of Copenhagen
December 14, 2018

Contents

0.1 Introduction

The goal of this project is to develop search engine as part of the Introductory Programming course at the IT University of Copenhagen. Sections 2- 4 are reporting the solutions on the mandatory tasks posted in the project description, namely Faster Queries using an Inverted Index, Refined Queries and Ranking Algorithms. Each of the mandatory tasks contain description of the

- Taks, which is introduction of the task that have to be solved in the given section;
- Basic Approach, which describes the solution;
- Technical description, explaining the software architecture used in the solution;
- Testing considerations, considering the corresctness of the solution;
- Benchmark/ Reflection, includes benchmarking results or other conclusions based on the observations or theoretical considerations;

Besides the mandatory tasks, the challenge of the implementating OkapiBM25 algorithm for the task Ranking Algorithms have been solved as well. Some extensions also have been implemented, namely:

- Changes to the client GUI
- Implementation of the WebCrawler

0.2 GitHub

0.3 Project Delivery

The project documentation have been submitted via learnit.dk and the source code that accompanties this report as a singel zip dile called ip18groupR.zip has been handed alongside with this report. The code is also available on ITU's GitHub: https://github.itu.dk/wilr/ip18groupR
To start the program one should:

1. Download the zip file;
2. Select directory for saving the file;
3. Open the Command Promt;
4. Find the directory, where programm files have been saved;
5. Build the Gradlew by calling *./gradlew* for Linux users (L) or *gradlew* for Mac and Windows users;
6. To start the server as a Spring boot application use the command *gradlew runWeb*;
7. After the server is started, open a browser and type in *http://localhost:8080/*;
8. Start searching by typing in search queries in the search textbox shown on the HTML page and press Enter on the keyboard or *Busca* button;
9. To have an overlook on other tasks performed by gradlew, call *gradlew tasks*;

To change the index file:

- Easiest way to change the index file is to open the *configuration.properties* file and change the database property to a different file;
- Other way to change the index file is to give the file path as arguments *(args w/ –args)* when calling the gradle task *runWeb*

To start the extension WebCrawler one should:

- Call the gradlew task by writing the command *gradlew startWebCrawler*;
- Every time the WebCrawler have visited a web page it will append it to the *real_data_file.txt in the data*;
- One has to make sure there are no previous WebCrawler running into the background, as it will continue to add results to the *real_data_file.txt in the data*;
- In order to search throught the WebCralwer results, the *real_data_file.txt in the data* have to be used as index file for the search engine;
- In order to search throught the historical WebCralwer results where data set have been build for under six hours, the *real_data_file_20181213.txt in the data* have to be used as index file for the search engine;

0.4 Statement of Contribution

All authors contributed equally to all parts of the mandatory tasks. Ashley Rose Parsons-Trew took up the challange of implementing OkapiBM25 algorithm for the task Ranking Algorithms, Hugo Brito made the graphic design and the client GUI extension, Ieva worked with the WebCrawler extension and Jonas Hartmann Andersen enabled the team to successfully work whith this GitHub.

CHAPTER 1

FileHelper

1.1 Task

This chapter comprises the changes and improvements done on the `FileHelper` class, which can be found in the folder src/main/java/searchengine. The intended functionality of such class is to parse a file containing data about websites, returning its result as a `List<Website>` to be used later on for by other parts of the program.

1.2 Basic Description

The `parseFile` method is designed to take in parse a file and extract all the websites that are contained in the file. It features the following:

- `@param filename` — The filename of the file that we want to load. It needs to include the directory path as well.
- `@return` — The list of websites that contains all the websites that have both titles and words that were found in the file.

Each file lists a number of websites including their URL, their title, and the words that appear on a website. Moreover, it should only take in consideration data on websites that fulfilled the following format:

```
*PAGE:http://www.websiteURL.com/
Website's title
word 1
word 2
...
word n
```

This meant that, for a website to be passed on to the index, it must have a title, an URL, and the amount of words has to be more than zero.

1.3 Technical Description

As part of the set up of this task, the `FileHelper` class — specifically the `parseFile(String filename)` method — was updated such that from the database file, only websites that have a url, title, and at least one word of webpage content are read-in and stored in the server.
This was accomplished by an `if` statements to check the assignments of the URL and title fields prior to adding a `new Website` object to the `ArrayList<Website>`.
However, the major of the changes made to this method were to how the method recognised the content of each line scanned in in order to know how to treat it. Previously, this was accomplished by making use of the knowledge of the very specific file format, `String` methods, and `boolean` field variables. This was all replaced by two regular expressions:

```
    Pattern website = Pattern.compile("(https?:\\/\\/[A-Za-z0-9.\\/
        _]+)");
    Pattern webTitle = Pattern.compile("[A-Z][a-z]+[A-Za-z0-9\\s]+?
        ");
```

This was followed by methods of the `Matcher` class.
Even though it does not look to be that big of a change, doing so means that the two field variables are no longer needed, hence less has to be juggled when reading and making further changes to the code.

1.4 Testing

White-box tests were developed around the branching statements in the updated method, and a coverage table was produced, please refer to Appendix B.1. From this coverage table the B.2 was produced. The data set data/test-file1.txt is an empty file, and the rest contained the data shown in B.3. This process using the Coverage and Expectancy table shown in Appendix B, is an example of how we construct our tests. JUnit tests were then produced from table B.2, as found in `FileHelperTest.java`.
Correctness was verified along two axis:

- the size of the `List<Website>` returned,
- the specific contents of the `List`.

As you can see from the Actual Output column of B.2, the updated code failed test B3, highlighting a weakness in the code, and subsequently had to be debugged. Including another `if` statement after the `while` loop resolved the issue, and following that all tests were passed.
Lastly, a test to check if the websites contained in the `Index` were the same as the websites read by the `FileHelper` class was wrote and performed. This was performed on the tiny, small and medium files and was meant to see whether the behaviour of the index would stay the same when the database size changed.

Chapter 2

Faster Queries using an Inverted Index

2.1 Task

When using an search engine, the most important aspect is to be able to perform a search and get the results almost instantaneously. One way of doing this is by using an `InvertedIndex`, which sorts the websites according to the words contained in each website. Hence when searching for a specific term, instead of going over every website, it will go over all the words instead and then provide the websites related to searched term. While building the Inverted Index can be system heavy, it is a one time operation that will allow the search engine to operate significantly faster.

2.2 Basic Approach

All the files regarding the classes mentioned on this chapter can be found on the folder src/main/java/searchengine.
The `Index` was generalised into an interface to make it easy to test the different

indices and switch between them. The following methods define the aforementioned `Index` interface:

- `build` — Processes a list of websites into the data structure.
- `lookup` — Given a query string, returns a list of all websites that contain such query.
- `provideIndex` — Provides all websites in a given `Index` as a collection. This specific method was added for the ranking algorithm and the testing of the index.

The inverted indices were then implemented using inheritance, since both the `InvertedIndexHashMap` and `InvertedIndexTreeMap` can be given exactly the same methods, being only difference their individual data structure.

2.3 Technical Description

As previously stated, a generalised `Index` interface was created. Each of the classes below implements this interface, visualized in 2.1.

2.3.1 `SimpleIndex`

The provided default way of indexing data was called `SimpleIndex`. This solution is implemented looping through every word of every website, storing the matching website that matched the given query on an `ArrayList<Website>`.

2.3.2 `InvertedIndex`

The second (and improved) approach to index the data was to use an `InvertedIndex`. As the name implies, here the relationship between a website and its words is inverted, meaning that each word knows to which sites it belongs to. In Java terms, `Maps` are used where every word is a `Key` with and associated `Value`, which consists of an `ArrayList<Website>`.
Since it did not make sense to create instances of `InvertedIndex`, this class was made `abstract` and, since it implemented methods of the `Index` interface, it implements it.

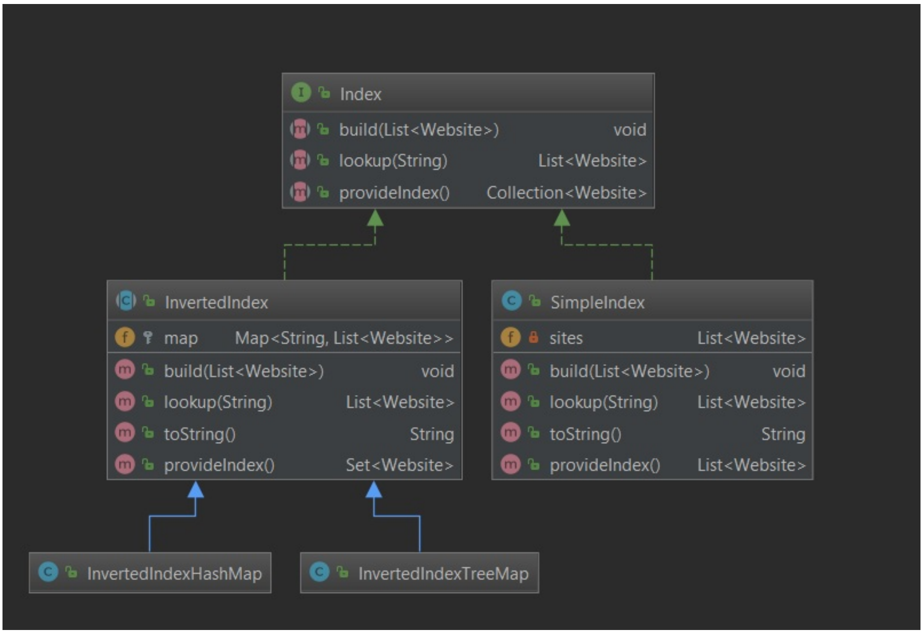

Figure 2.1: UML Diagram for the Software Architecture of Index data structures.

2.3.2.1 `InvertedIndexTreeMap`

This class `extends InvertedIndex`.
The underlying data structure of the `TreeMap` is a Red-Black tree based `NavigableMap` implementation, sorted either by the natural ordering of its keys or by a `Comparator`. `TreeMap` provides *guaranteed log(n) time* performance for the operations **containsKey**, **get**, **put**, **remove**.Oracle (2018b) `TreeMap` uses only the amount of memory needed to hold it's items, therefore this solution is suited when it is not known how many items have to be sorted in memory and there are memory limitations. Solutions is also suited when the order in which items have been stored is important and O(log n) search time is acceptable.Baeldung (2018)

2.3.2.2 `InvertedIndexHashMap`

This class `extends InvertedIndex`.
The underlying data structure of the `HashMap` is a hash table based implementation. This implementation provides *constant-time* performance for the

basic operations such as **get** and **put**.Oracle (2018a) However this is true under assumption that there are not too many collisions. This is because this `Map` implementation acts as a basket hash table and when buckets get too large, they get transformed into tree nodes, similar to those of `TreeMap`. Baeldung (2018) Some of the downsides of building the HashMap are that it requires more memory than it is necessary to hold its data and when a HashMap becomes full, it gets resized and rehashed, which is costly.Baeldung (2018)

2.4 Benchmarking

In order to choose one of the implementations, for the search engine, the benchmark test was performed to gain empirical data of the performance of each of the implementations. For the benchmark test, JMH (a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks) was used. OpenJDK:jmh The benchmark test was carried out using 20 words (random nouns, verbs, adjectives and conjunctions), which were looked-up using the three different `Index` implementations and in three different size databases: `enwiki-tiny`, `enwiki-small`, `enwiki-medium`. JMH then provides information about an average Score, measured in nanoseconds per operation, whose results of which can be found in table 2.1
During the benchmark it was assured that the test environment is as similar as possible among the different trials, meaning that all tests were performed on the same machine and no other applications running on the background.

Table 2.1: Benchmark Scores. Each score is an average in ns/op

Data set	Simple Index	Inverted Index	
		HashMap	TreeMap
enwiki-tiny	18 944,884	1 052,067	1 591,311
enwiki-small	8 819 338,592	1 883,776	3 622,582
enwiki-medium	233 498 546,571	27 451,020	30 176,993

The benchmark results shows that the `SimpleIndex` is significantly slower than both of the `InvertedIndex` implementations: 233 498 546,571 ns/op versus 27 451,020 ns/op for the `InvertedIndexHashMap` and 30 176,993 ns/op for the `InvertedIndexTreeMap` using the `enwiki-medium` dataset. In order to describe the results, let the number of websites be m and words be n. The difference in performance can be explained as follows:
When the `SimpleIndex` is looking up the search word, it looks though all the sites, which takes *O(m)* time, and for each site it looks through all the words which takes *O(n)* time, therefore total search time is $O(m \cdot n)$. The two other

methods provide faster performance time. `InvertedIndexTreeMap` provides a *guaranteed* performance of *O(log(n))*. `InvertedIndexTreeMap` provides best-case performance of constant time *O(1)* and the worst-case performance of *O(log(n))* time (since Java 8). Worst-case performance occurs when the hash function is not implemented correctly and values are distributed poorly in buckets, leading to high hash collision.
There are several considerations when choosing the implementation for storing the data for the Search Engine.
`HashMap` seems to be better fit than a `TreeMap` for our search Engine implementation, because in this case the order of data is not important whereas the performance looking up the websites corresponding the search word is. The `HashMap` can be expected to perform in constant time which is better than `TreeMap`'s *log(n)* time, and only in the `HashMap`'s worst-case performance is it *log(n)* time. The given data sets are fixed, therefore the costly resizing and rehashing is not going to occur implementing Hashmap. HashMap performed the best on all of the given different size data sets in benchmark test. This is the reasoning for choosing HashMap implementation over the TreeMap implementation for this Search Engine project.

2.5 Testing Considerations

After the above changes were implemented, development tests were written in order determine the viability of the code and whether the changes satisfied the requirements of the task. To that end, JUnit tests were devised for each class that was updated.
The correctness of the `build` and `lookUp` method were verified using unit tests (JUnit 5), which can be found in the `IndexTest` file.
When setting up the test, a small `List<Website>` was created which made it easier to predict the expected results of the methods. Each test checks all of the indices using the white-box coverage considerations. The `SimpleIndex` were more used as a reference to the others, and the tests as it should be able to pass all test, due its simple nature.
The `build` method was verified by creating a `String` of what was expected the index should contain and then calling the `toString` on the index.
The `lookUp` method was tested by providing it with words and then checking the size of the list returned against the expected size of that list.

Chapter 3

Refined Queries

3.1 Task

This task enables complex query handling. This is a basic feature that is expected from a search engine: to be able to understand queries that consist of more than just one word (intersected search: all words m). Additionally, the task required the search engine to be able to handle aggregated results from different (possibly) multi-word queries when the 'OR' keyword is present (unioned search).

3.2 Basic Approach

All the logic necessary to handle the queries was implemented in the class `QueryHandler`. When considering what needs to be accomplished as well as what the user can input in the search field, the present task is accomplished by following these steps:

1. Sanitise the query: this comprises of checking if the query is meaningful and fulfils basic criteria in morphological terms, and to enforce it in the

cases it does not.

2. Separate the query into sub-queries whenever the 'OR' keyword is present.

3. For each of the sub-queries, find websites that contain all the words in that sub-query.

4. Aggregate all the results of the multiple sub-queries on a list.

In order to achieve encapsulation and responsibility-driven design, the following methods were implemented in the above-mentioned class:

- `getMatchingWebsites`: Core method of the class. It is responsible for:
  - Receiving the input and passing it to the `cleanQuery` helper method.
  - Receiving the return `List<String>` from the `cleanQuery` method and passing it, element by element, as a parameter to the `intersectedSearch` method.
  - Receiving every result of the `intersectedSearch` method and store it, so that when every element of the list is processed, it returns the matching websites.
- `cleanQuery`: Auxiliary private method to make sure that the input is free from unaccounted or irrelevant input.
- `intersectedSearch`: Auxiliary private method that returns websites that match simultaneously all the words in the input `String` parameter.

3.3 Technical description

3.3.1 Field

The `QueryHandler` class takes an `Index` object and a `Score` object as the initialising parameters and assigns them to private fields. The `Index` can be any of the indices described in the previous chapter as all extend the same `Interface`. The same can be said for the `Score` interface, and the details of this can be found in the next chapter.

3.3.2 getMatchingWebsites core method

As soon as this class is instantiated, its intended use expects the getMatchingWebsites method to be called. This method takes in a String as parameter, which consists of the search terms and returns a List<Website>, which consists of the matching results. As the description in the Basic Approach states, this method uses two auxiliary methods to process the data as necessary. The first data processing happens when the parameter is passed to cleanQuery method, which then returns a list of Strings that can be used to proceed with the search.

3.3.3 cleanQuery auxiliary method

This method enforces consistency in the input to be later on used to search for results. The first steps of the process consist of:

- Replacing all the punctuation characters by spaces
- Replacing every one or more space characters by a single space character

The above mentioned steps are achieved by making use of the String method replaceAll.

```
private List<String> cleanQuery (String input){
    input = input.replaceAll("\\p{Punct}", " ").replaceAll("\\s+",
        " ");
```

After this, the 'OR' keyword is used as criteria for splitting the input String using the split method, which is then used to create a List<String> searches. The idea is that every element of the list will consist of an intersected search, and the search results of each element will then be aggregated to achieve the final result. The split method gives String [] as a result, but is then parsed as an ArrayList<String> as this is more maleable, and through the java.util methods allows for the consistency in the List<String> searches to be enforced (such as trimming all the searches in case they start or end with empty spaces, deleting all empty entries in the List<String>, and making everything lower case due to the way the website content is stored in the index). This was elegantly achieved by using lambdas.

```
searches.replaceAll(String::trim);

searches.removeAll(Arrays.asList(""));

searches.replaceAll(String::toLowerCase);
```

3.3.4 Intermediate steps in the getMatchingWebsites method

The refined search query, now stored in a List<String>, is iterated through and each element passed as a parameter to the auxiliary method intersectedSearch, the results of which are stored in a Set<Website> results. The root of the reason for the choice of such data structure is the fact that it does not allow duplicates (as opposed to a List, which expedites the process. Since we intend to iterate through a set of data, it seemed appropriate to implement a for loop.

3.3.5 intersectedSearch auxiliary method

Given a certain String parameter, this auxiliary private method returns a Set<Website> where each Website matches simultaneously all the words in such parameter. The idea is that:

- The String parameter is split by space characters using the split method.
- The resulting String[] is converted to an ArrayList<String>.
- The lookup method, using field Index idx, is called on the first element in the ArrayList<String>.
- The result of this is stored in a HashSet<Website>.

Should there be more than one String in the ArrayList<String> i.e. more than one word in the parameter String to intersect the first set of results with, these results will be used to compare with the results of the remaining String in the ArrayList<String>. In order to accomplish such task, the results of every other given word were successively compared with the results from the first element of the list. The refining criteria was to keep only the websites that were present on both lists. To this end, the Set method retainAll was utilised.

3.3.6 Final steps in the `getMatchingWebsites` method

All the results from each of the different intersected searches performed by the `getMatchingWebsites` method are added to a `HashSet<Website>` (again, to avoid any duplication of websites), which is then passed to the `rankWebsites` method and returned as a `List<Website>`. The details of the `rankWebsites` method will be discussed in more depth in the next chapter.

3.4 Testing considerations

Upon testing for correctness, we split the test cases in two main groups: one that tests basic functionality using valid data (positive testing), and a second one that tests functionality with bad user behaviour (negative testing).

- Testing for the basic functionality:
  - One word
  - One or more words separated by spaces;
  - Two words with the "`OR`" keyword in between;
  - Groups of words separated by the "`OR`" and by spaces;
- Testing for bad user behaviour, where the query:
  - Is empty;
  - Starts with white space followed by the "`OR`" keyword;
  - Repeats the words separated by the "`OR`" keyword;
  - Consists of solely the "`OR`" keyword;
  - Starts with the "`OR`" keyword followed groups of words separated by spaces;
  - Starts with white space followed by the "`OR`" keyword followed groups of words separated by the "`OR`" and by spaces and ends with "`OR`" ;
  - Consists of only white space;
  - Starts with punctuation and white space and is followed by a group of words separated by spaces;
  - Consists of several "`OR`" keywords separated by spaces, followed by a group of words separated by spaces and ends with several "`OR`" keywords separated by spaces;

- Consists of several words separated by the `"OR"` keyword where there is more that one `"OR"` occurrence between words;
- Contains only upper case characters;
- Contains no spaces but a word surrounded by several `"OR"` keywords;
- Contains an upper case word next to an `"OR"` keyword with no spaces in between, followed by another `"OR"` keyword and a word.
- Lastly, we considered the worst case scenario where a query contained many of the above-mentioned cases and also for when there was punctuation between words.

3.5 Reflection

This task set out to enable the `QueryHandler` class to handle more complex queries (i.e. multi-word queries referred to as intersected queries, and queries making use of the `"OR"` keyword which are referred to as unioned searches). Due to the fact that this update involved merging the results of individual searches on single words to return an amalgamated `List<Website>`, `Set` was used liberally to avoid duplication of `Website` results. Regular expressions were used to parse the search query quickly in the `cleanQuery` method, and the `String []` were converted to `ArrayList<String>` for flexibility as `ArrayList` offers methods that were integral in sanitising the original search query.

CHAPTER 4

Ranking Algorithms

4.1 Task

Typically, results returned from search engines are ranked so the more relevant search results first are featured on the top of the list. The general idea behind this is that, for a given website, a score is calculated for each word to indicate the importance of that word on such website.
As a query might consist of a union of a certain number of intersected searches, the score is calculated as follows:

- Intersected Search: the score is a summation of the scores for each individual word in the search;
- Union Search: the score is taken as the maximum of the score for each part of the union search.

This score should then be used to order the results of a given query.

4.2 Basic Approach

For this task, the following ranking algorithms were introduced:

- Term Frequency (TF)
- Term Frequency — Inverse Document Frequency (TFIDF)
- Okapi BM25 (which is discussed in detail on the Extensions section of this paper)

Please note that these consist of different implementations of the same aforementioned general idea: assigning a score to a website based on some metric of relevance.
Research was conducted into the implementations of each of the ranking algorithms.

4.2.1 Term Frequency

The TF is simply this: given a word and a document (which in this case, is a webpage), how many times does a word appears on that document? (Luhn, 1957)
Even though there are various implementations on this basic formula, the TF formula settled on in the end was:
Let

- TF be the term frequency
- f be number of occurrences of a given word on the website
- w be the total number of words on that website

We have:

$$TF = \frac{f}{w} \tag{4.1}$$

Such formula enabled us to add a layer of normalisation on the score, as a word occurring 10 times on a 50-word-long website has a different significance to a word appearing 10 times on a 500-word-long website.

4.2.2 Term Frequency — Inverse Document Frequency

Before the TFIDF algorithm can be discussed, the idea behind "Inverse Document Frequency" must be explained.
The idea behind the inverse document frequency is that, while the number of times a word appears on a webpage is a good indication of how important that word is to that webpage, there are many common words such as 'the', 'and', 'this', etc., that will inevitably appear multiple times on a website and will therefore skew the score of any kind of score based on term frequency (Jones, 1972). The inverse document frequency formula is designed to take this into account and is calculated as follows:
Let

- IDF be the inverse document frequency
- s be number of websites in the search engine index
- sw number of websites containing the word

We have:

$$IDF = log_2\left(\frac{s}{sw}\right) \tag{4.2}$$

Taking the log of this ratio means that, the more times a word appears on a website in the database, the closer the IDF gets to 0, and a word that appears on every website in the database is awarded an IDF value of 0. In other words, common words that are likely to appear on multiple (if not all) websites will have no impact on the ranking score.
With that in mind, the meaning behind the $TFIDF$ ranking algorithm becomes clear. By (4.1) and (4.2), the $TFIDF$ score is calculated as follows:

$$TFIDF = TF \times IDF$$

The TF score judges the relevance of the word to the website, and the IDF is a weighting to adjust for common words. Very common words are awarded a $TFIDF$ score of 0 (or close to 0) and therefore give no impact in intersected searches.
It was also decided that the various implementations of `Score` classes created would be solely responsible for the calculations.

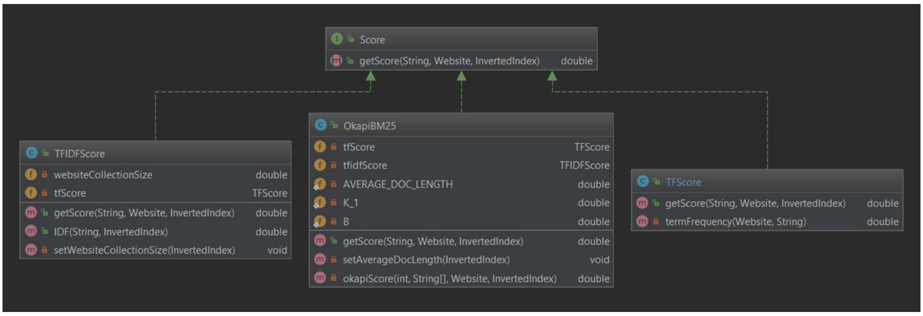

Figure 4.1: UML Diagram for the Software Architecture of Score data structures.

4.3 Technical Description

As per the task description, a generalised `Score` interface was created with only one method — `getScore` — which performs the score calculation for the given word with the given website, taking the following parameters:

- `@param word`: a word from the search query
- `@param site`: the website being scored against the search string
- `@param index`: the index of websites

Each of the below classes implement this interface. Besides this the `rankWebsites` method was added to the `QueryHandler`. That essentially takes one of the given ranking algorithms and scores the websites relevant to the query and then order them from highest-to-lowerst. This method will also be used by the OkapiBM25.

4.3.1 `TFScore` class

The `TFScore` class was the simplest of the three classes to implement, as shown in Figure 4.1. Due to the version of the term frequency formula used in this project, there is a helper method to supplement the required `getScore` method, leaving the `getScore` method to only handle the division. Due to the changes made to the `FileHelper` class — namely, that no website that lacks a title or words can be created — it's not possible for there to be a divide by zero error, so no steps were made in this method to account for it.

4.3.2 `TFIDFScore` class

As the $TFIDF$ makes use of the TF calculation, the `TFIDFScore` class was given a field of type `TFScore` with which to call methods on as needed, rather than creating a new object each time it was required. Again, helper methods were added to deal with the different aspects of the calculation i.e. the IDF and the number of websites in the search engine index. Also, in order to enable any index to work with this impementation, a helper method was added there, `provideIndex()`, which returns a `Collection<Website>`, much needed to compute this. The number of websites in the index was built from the index `Map<String, Website>`. Such implementation makes use of the fact that a `HashSet` allows no duplicates in order to calculate the number of distinct websites in the index. Again, the `getScore` method only handles the multiplication.

For the tests on the `OkapiBM25Score` class, single word and multi word query tests were constructed.

4.4 Testing considerations

The mathematical correctness of the score calculations were verified using unit tests, which can be found in the ScoreTest.java file. The positive testing approach were applied in this context. The set up comprised of building a small index of websites, which allowed for the score values of various words on various websites to be calculated manually and compared to the results of the `getScore` method. Each test covered one class, and the individual tests were determined using the standard white-box coverage considerations. For the tests on the `TFIDFScore` class, a comparison was also included to confirm that a word occurring once on more than one site will have a lower score than a word occurring once on just one site.

The application of the algorithms were tested in the RackScoreTest.java file. Black box and positive testing where used in these tests. The test checked whether ranking algorithm TF and TFIDF would rank the websites correctly. The two ranking algorithms were testing websites which were set up specifically for the tests to allow easier prediction of how the ranking should be. Two websites in the `setUp` was made to create noise, to see how the algorithms would react to them.

4.5 Reflection

After implementing the two different ranking algorithms and testing their implication it was possible to compare the different results they provided. While the TF algorithm sums together the term frequency of each word in the search query, the TFIDF algorithm also considers the relative frequency of each of the words in the search query appearing across the collection of websites on which the search is performed. This means that if the search query would consist of "Queen of Denmark", the TFIDF ranking algorithm would provide more relevant results than the TF algorithm, because one could assume that the word "of" would appear significantly more often across different websites, than words "Queen" or "Denmark". Therefore while the unit tests were used to secure that the software functions as intended, test `setUp` in RackScoreTest.java was also set to demonstrate the potential relationship between the search query and the background of the rest of the websites in an abstract and simplified manner.
Based on the theoretical assumptions on the relevance of the two algorithms, when applied to search tasks and their behavior in the RackScoreTest, it can be concluded that the TFIDF is the most relevant to as it provides a more appropriate result to the user's search query.

Chapter 5

Extensions

5.1 Okapi BM25

5.1.1 Okapi BM25

(Basic Approach) The Okapi BM25 algorithm is a more sophisticated type of TFIDF ranking algorithm. It's a summation over all words that make up the search term, making use of the TF calculation as well as the IDF calculation along with variables that can be used for optimisation (Robertson et al., 2009). The version of the formula used in this project is:

$$OBM25 = \sum_{i=1}^{n} IDF(w_i) \cdot \frac{TF(w_i) \cdot (k_1 + 1)}{TF(w_i) + k_1 \cdot (1 - b + b \cdot \frac{W}{W})} \tag{5.1}$$

with the optimisation variables set as $k_1 = 1.2$ and $b = 0.75$ since no advanced optimisation was considered and

- $OMB25$ is the Okapi BM 25 score

- A search term w consists of individual words $w_1, w_2, ..., w_n$
- $IDF(w_i)$ is the inverse document frequency score applied to the word w_i
- $TF(w_i)$ is the term frequency score applied to the word w_i
- W is the number of words on the webpage
- $\bar{W}$ is the average number of words on a webpage

5.1.2 OkapiBM25Score class

(Technical Description) As the Okapi BM25 algorithm makes use of both the TF calculation and the TFIDF calculation, these objects were stored as fields in the `OkapiBM25Score` class in a similar manner to what was done for the `TFIDFScore` class. The optimisation constants and the average document length were set as static fields. Two helper methods were added: `setAverageDocLength` and `okapiScore`. `setAverageDocLength` calculates the mean number of words per website based on the websites in the index, and `okapiScore` is a recursive method to perform the summation of all the individual scores of all the words in the search query to return to the `getScore` method.

```
private double okapiScore(int count, String[] words, Website
    site, InvertedIndex index) {
int docLength = site.getWords().size();
if(count != 0) {
    double IDF = this.tfidfScore.IDF(words[count], index);
    double termFrequency = this.tfScore.getScore(words[count],
        site, index);
    double score = IDF*((termFrequency*(K_1 + 1))/(
        termFrequency + K_1*(1 - B + B*(docLength/
        AVERAGE_DOC_LENGTH))));
    return score + okapiScore(count-1, words, site, index);
} else {
    double IDF = this.tfidfScore.IDF(words[0], index);
    double termFrequency = this.tfScore.getScore(words[0], site
        , index);
    return IDF*((termFrequency*(K_1 + 1))/(termFrequency + K_1
        *(1 - B + B*(docLength/AVERAGE_DOC_LENGTH))));
    }
}
```

5.1.3 OkapiBM25Score class

(Testing Considerations) The mathematical correctness of the OBM25 score calculations were verified using unit tests, which can be found in the ScoreTest.java file alongside the unit tests for the other `Score` classes. The set up was the same as previously mentioned in the Ranking Algorithm's chapter. Positive tests for single word and multi word query tests were constructed in the following manner:

- the word doesn't occur on the specified website
- the word doesn't occur in the website index at all
- the word occurs once on the specified website
- the word occurs once on the specified website and at least one other website
- the word occurs more than once on the specified website
- multi-word query: words don't occur on the specified website
- multi-word query: words occur once on the specified website
- multi-word query: words occur more than once on the specified website
- multi-word query: words occur once on the specified website and at least one other website
- comparison of the above score values

Negative testing consideration were harder to formulate. The most obvious to test would be dividing by 0, however as seen in equation 5.1, it's not actually possible for the denominator to be 0 due to the optimisation variables: either $TF(w_i)$ or $\frac{W}{\bar{W}}$ has to be negative, which is not possible. To that end, no negative tests were considered for the `OkapiBM25` class.

5.2 Improve the Client GUI

We were given the possibility to improve the front-end of the search challenge as an added task. The client side of our product consists of a set of files that dictate, among other things, the aspect of the page, implements the pieces and bits of code that will ultimately allows the user communicate with the server

and perform the searches; and arranges the results of the queries in a more user-friendly fashion.
The files containing the code that concerns the front-end of the search engine can be found in the folder static. Here follows each of the files' description:

- index.html — This is the first file the browser reads upon accessing the root of a website hosted in any given domain. Hence it holds what other files to read also (such as the styling sheet), provides written unformatted text which will be displayed on the browser, which may also includes links to other webpages.
- style.css — It is possible to style a given webpage from a given html file, but it is best practice to do in on a separate file (such as the present one). Should one build a website with several pages (which for each a separate html file is necessary), styling can become cumbersome and even result in styling inconsistencies. So this file provides a styling guide that can be used for several different pages providing consistency among all of them, and for this is only necessary to add the line of code that points to such file in the html.
- code.js — It holds the javascript code that allows for changes in the html (or even style), which will result in changes on what the user sees. Our javascript code was responsible for receiving the search term(s), sending them to the server, receiving the results of the given search, and translating them into html.
- Image files in static/img/ — Some images needed to provide the website with the desired aspect.

The basic implementation of the client GUI allowed the very basic functionality of performing a search. So the accomplished tasks in this regard will be described in the following subsections.

5.2.1 Adding content to be displayed by the html

A wireframe of a preliminary graphic design of the website can be found in the appendix. Several aspects of the GUI were changed to improve the user's experience. We included a footer with links to ITU's website, the course page, as well as our LinkedIn profiles. Overall names of the classes and id's to be used in the styling sheet were also changed to achieve the intended design. Code was also added to include background images. Additionally, it seemed intuitive to allow the enter key to trigger a search, so such feature was implemented by including a small script in the html file.

5.2.2 Styling

All the aspect of the website was described in the style.css file. In here, virtually everything was changed, namely:

- Centering the content of the webpace;
- The aspect of every given class, id, link, as well as behaviour of certain elements when, for example, the user hovers the mouse over that specific element;
- Providing responsiveness no the website (adjusting the aspect of the content depending on the size of the viewport;
- Behaviour of the background images, where they would display as cross-faded slide show;
- Behaviour of the searchbar, where it would change its size by clicking on it.

5.2.3 Adding functionality through javascript

Changes in the javascript code, which can be found in the static/code.js, where made in order to allow for:

- Provide a different answer depending on the given different queries. The cases we accounted for were the following:
  - No query was provided;
  - The query did not provide any result;
  - The query in a number of results different than 0.
- Besides the title and the URL, the results are also accompanied by a certain number word from the given website.
- Clicking on a result will open it on a new tab (instead of the current tab).

Appendix A

Test Figure reference

This is a test of the appendix and how to reference to something in it. Below is shown an image which is used for test[1]testimage.

[1]this is just for testing...`www.test.dk`

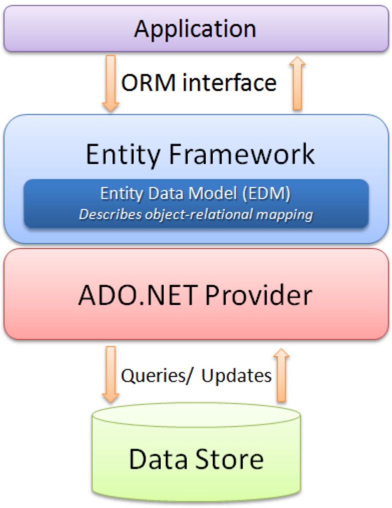

Figure A.1: Microsoft Entity Framework

Appendix B

Tables

Table B.1: Coverage table of the parseFile(String filename) method

Choice	**Input property**	**Input data set**
1 catch	incorrect file name	A
1 try	file name	B
2 while: zero times	empty file	B1
2 while: once	file has one line	B2
2 while: more than once	file has two lines	B3
2 while: more than once	file has at least three lines	B4
3 true	the line contains a web url	B3, B4
3 false	the line does not contain a web url	B1, B2
4 true	either the listOfWords field or the title field is null	B3, B4
4 false	both the listOfWords and the title fields are not null	B4
5 true	the url field is not null	B4
5 false	the url field is null	B3, B4
6 true	the line contains a website title	B3, B4
6 false	the line doesn't contain a website title	B2
7 true	listOfWords is null	B2, B4
7 false	listOfWords is not null	B4

Table B.2: Expectancy table of the JUnit tests

Input data set	Input data	Expected output	Actual output
B1	"data/test-file1.txt"	returns an ArrayList<website>, size() == 0	returns an ArrayList<website>, size() == 0
B2	"data/test-file2.txt"	returns an ArrayList<website>, size() == 0	returns an ArrayList<website>, size() == 0
B3	"data/test-file3.txt"	returns an ArrayList<website>, size() == 0	returns an ArrayList<website>, size() == 1
B4	"data/test-file-errors.txt"	returns an ArrayList<website>, size() == 2	returns an ArrayList<website>, size() == 2
B4	"data/test-file4.txt"	returns an ArrayList<website>, size() == 2	returns an ArrayList<website>, size() == 2

Table B.3: Data Set

data/test -file2.txt	**data/test-file3.txt**	**data/test-file4.txt**	**data/test-file-errors.txt**
word3	http://example.com Title1	*PAGE:http://page1.com Title1 word1 word2 *PAGE:http://page2.com Title2 word1 word3	word1 word2 *PAGE:http://page1.com Title1 word1 word2 *PAGE:http://wrong1.com Title1 *PAGE:http://wrong2.com *PAGE:http://wrong3.com Titleword1 Titleword2 *PAGE:http://page2.com Title2 word1 word3

Bibliography

Baeldung (2018). Java treemap vs hashmap. `https://www.baeldung.com/java-treemap-vs-hashmap`.

Jones, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21.

Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of research and development*, 1(4):309–317.

Oracle (2018a). Class hashmap. `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`.

Oracle (2018b). Class treemap. `https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html`.

Robertson, S., Zaragoza, H., et al. (2009). The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.