# 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

**b**

# 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

1. Taks, which is introduction of the task that have to be solved in the given section;

2. Basic Approach, which describes the solution;

3. Technical description, explaining the software architecture used in the solution;

4. Testing considerations, considering the corresctness of the solution;

5. 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. Several extensions also have been implemented, namely

1. Changes to the client GUI

2. Implementation of the WebCrawler

## 0.2 GitHub

The source code that accompanties this report as a singel zip dile called ... zip has been handed alongside with this report. The code is also available on ITU's GitHub: https://github.itu.dk/wilr/ip18groupR

## 0.3 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

# Faster Queries using an Inverted Index

## 1.1 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 using an Inverted Index, 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. However, before this it should be ensured that the database will be read correctly and any websites not containing either title or words are omitted.

## 1.2 Basic Approach

Based on the task the `parseFile(String fileName)` method has to be able to only take in websites that fulfill the following format:

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

In order for the website to be given to the index, the website must have a title and the amount of words has to be more than 0 for it to be considered.

The `Index` was generalized into an interface to make it easy to test the various indices and switch between them. The following methods define the `Index`:

- `build` The build method processes a list of websites into the index data structure.
- `lookup` Given a query string, returns a list of all websites that contain the 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 `InvertedIndices` were then implemented using inheritance, since both the `InvertedIndexHashMap` and `InvertedIndexTreeMap` can be given exactly the same methods with the only things that differ are the data structure.

## 1.3 Technical Description

In accordance with the task description, a generalised `Index` interface was created. Each of the classes below implements this interface, visualized in 1.1.

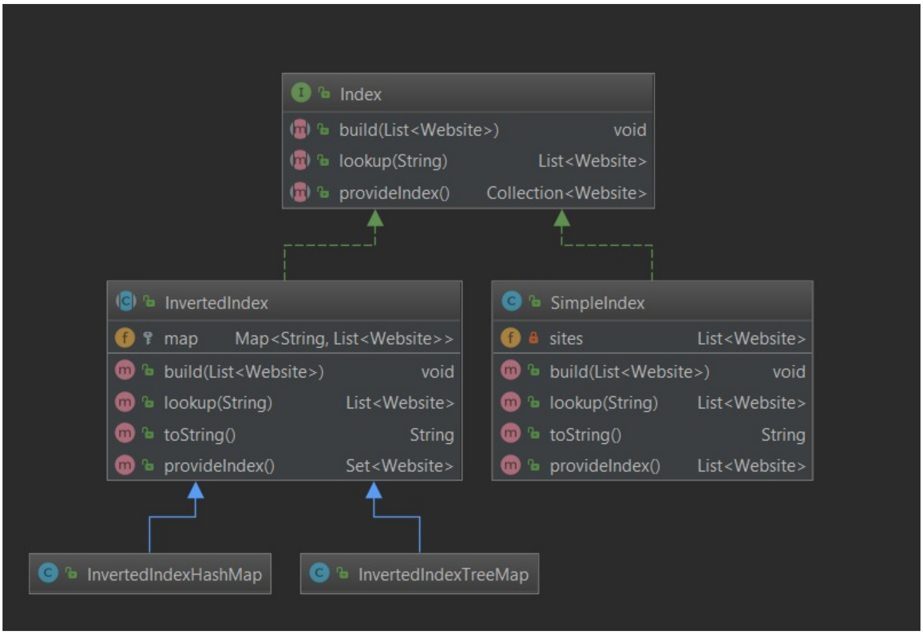
Index
build(List<Website>) void
lookup(String) List<Website>
provideIndex() Collection<Website>
InvertedIndex
map Map<String, List<Website>>
build(List<Website>) void
lookup(String) List<Website>
toString() String
provideIndex() Set<Website>
SimpleIndex
sites List<Website>
build(List<Website>) void
lookup(String) List<Website>
toString() String
provideIndex() List<Website>
InvertedIndexHashMap
InvertedIndexTreeMap

**Figure 1.1:** UML Diagram for the Software Architecture of Index data structures.

### 1.3.1 FileHelper

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 meat 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]+?");
```

and the methods of the `Matcher` class. 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.

### 1.3.2 SimpleIndex

The provided default way of storing indexes was called SimpleIndex. This solution is implemented using an ArrayList<website>, which contains all of the sites and each site than have their own ArrayList<String> containing all the words on the sites.

### 1.3.3 InvertedIndexTreeMap

The second approach to store indexes was called InvertedIndexTreeMap. Here the relationship between site and it's words is inverted, meaning that each word knows to which sites it belongs to. 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**.[Ora18b] TreeMap use 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. [Bae18]

### 1.3.4 InvertedIndexHashMap

The third approach to store indexes was called InvertedIndexHashMap. Also in the HashMap the relationship between the site and it's words is inverted. To accomplish this a HashMap was used, where the words were used as Keys and a List of websites as a Value. The underlying data structure of the HashMap is a Hash table based implementation. This implementation gives *constant-time* performance for the basic operations such as **get** and **put**. [Ora18a] 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 node of TreeNodes, similar to those in TreeMap. [Bae18] 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. HashMap solutions should be chosen in cases when the approximate amount of items have to be maintained in the collection is known and the order in which items have been stored is not important. [Bae18]

## 1.4 Benchmarking

In order to choose one of the implementations, namely ArrayList, TreeMap or HashMap, 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 provides information about an average Score, measured in nanoseconds per operation, the results of which can be found in table 1.1
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 1.1: Benchmark Score in ns/op average for all indices**

| Data sets | SimpleIndext | Inv.IndexHashMap | Inv.IndexTreemap |
|---|---|---|---|
| | **avgt Score ns/op** | **avgt Score ns/op** | **avgt Score ns/op** |
| enwiki-tiny | 18944.884 | 1052.067 | 1591.311 |
| enwiki-small | 8819338.592 | 1883.776 | 3622.582 |
| enwiki-medium | 233498546.571 | 27451.020 | 30176.993 |

The benchmark results shows that the SimpleIndex is significantly slower than both of the Inverted Map implementations: 233498546.571 ns/op versus 27451.020 ns/op for the InvertedIndexHashMap and 30176.993 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 seams to be better fit than a TreeMap for Search Engine solution, 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 datasets in benchmark test. This is the reasoning for choosing HashMap implementation over the TreeMap implementation for this Search Engine project.

## 1.5 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.

#### 1.5.0.1 FileHelper tests

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 proccess 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 verfied along two axes: the size of the `List<Website>` returned, and 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.

#### 1.5.0.2 Index tests

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.

Lastly, a test of checking whether the websites contained in the Index is the same as the websites read by the `FileHelper` class was done. 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

# Refines Queries

## 2.1 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. Additionally, we were asked to enable a feature that aggregates results from different (possibly) multi-word queries when the "`OR`" keyword is present.

## 2.2 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 input from the `cleanQuery` method in a meaningful and orderly fashion, passing it then, element by element, 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 of irrelevant input.
- `intersectedSearch`: Auxiliary private method that returns websites that match simultaneously all the words in the input `String` parameter.

## 2.3 Technical description

### 2.3.1 Field

The `QueryHandler` class takes an `Index` object as the initialising parameter. This `Index` can be any of the indexes described in the previous chapter, as it consists of an `interface`. This was achieved with the following piece of code:

```
public class QueryHandler {
    private Index idx = null;
    public QueryHandler(Index idx) {
    this.idx = idx;
    }
```

### 2.3.2 getMatchingWebsites core method

As soon as this class is instantiated, it's 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. The signature is as follows:

```java
public List<Website> getMatchingWebsites(String line) {
```

As the description in the Basic Approach reads, 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. The code used to achieve this is the following:

```java
List<String> query = cleanQuery(line);
```

### 2.3.3 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 with the following code:

```java
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, 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 results of each search will then be aggregated to achieve the final result. The code that allows for the described step is as follows:

```java
List<String> searches = new ArrayList<>(Arrays.asList(input.split("
    OR")));
```

What follows is a sequence of steps that enforces consistency in our `List<String> searches`. This can elegantly be achieved by using lambdas. You can find below the remaining steps, followed by a small description.

```
searches.replaceAll(String::trim);
// trim all the searches, just in case they start or end with empty spaces

searches.removeAll(Arrays.asList(""));
// delete all empty entries

searches.replaceAll(String::toLowerCase);
// make everything lower case, because of the way the websites are crawled
```

Lastly, the result is returned:

```
return searches;
}
```

### 2.3.4 Intermediate steps in the `getMatchingWebsites` method

After having our input refined and organised on `List<String> query`, it is time to perform the actual search. As previously explained, this is done by passing each element of such list to the auxiliary method `intersectedSearch` and store the result of each iteration on 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. The piece of code that implements this description is as follows:

```
Set<Website> results = new HashSet<>();

for (String inputs : query) {
results.addAll(intersectedSearch(inputs));
}
```

### 2.3.5 `intersectedSearch` auxiliary method

Given a certain `String` parameter, this auxiliary private method returns a `Set<Website>` where each of the `Websites` matches simultaneously all the words in such parameter. Our approach to this consisted of the following steps:

- Initialising a local variable, `List<String> queriedWords`.
- Splitting the `String input` (from the method parameter) by space characters.
- Converting the resulting `String[]` to an `ArrayList<String>`.
- Storing this result on `queriedWords`.

This is achieved by the following piece of code:

```
private Set<Website> intersectedSearch(String input) {
List<String> queriedWords = new ArrayList<>(Arrays.asList(input.
    split(" ")));
```

After having a `List` of words to search for, the approach taken to find `Websites` that match simultaneously all the words in such `List` was the following:

- Initialising a local variable, `Set<Website> matches`, which will later on be returned by the method.
- Calling the `lookup` method on the field `Index idx`, providing the first element of `List<String> queriedWords` as argument.
- Storing this result on `matches`.

Should there be more than one word to intersect the results with, this first set of results will be used to compare with the results of the remaining words in the `List<String> queriedWords`. 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. The code that fulfils this task looks like the following:

```
if (queriedWords.size() > 1) {
for (String queriedWord : queriedWords) {
matches.retainAll(idx.lookup(queriedWord));
}
}
```

Lastly, the result is returned:

```
return matches;
}
```

### 2.3.6 Final steps in the `getMatchingWebsites` method

When all the results from each of the different intersected searches are gathered, they need to be put on a `List <Website>` so it can be returned by the method. This is achieved by creating a `new ArrayList<>` where the argument is the `Set<Website> results`. This is achieved with the following piece of code:

```
return new ArrayList<>(results);
}
```

## 2.4 Testing considerations

Upon testing for correctness, we split the test cases in two main groups: one that tests basic functionality, and a second one that tests for bad user behaviour.

- 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.

## 2.5 Reflection

Chapter 3

# Chapter 4: Ranking Algorithms

## 3.1 Task

Typically, results returned from search engines are ranked in some way so as to return the more relevant search results first. 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 the site. Then, for intersected searches (that is, searches where all words are required to be present on the webpage for it to be a match) the score is a summation of the scores for each individual word in the search, and for unioned searches (that is, searches where as long as either part of the union is present on the website it is a match), the score is taken as the maximum of the score for each part of the union search. For this task, the following ranking algorithms were introduced:

- Term frequency
- Term frequency - inverse document frequency
- Okapi BM25

  which are different implementations of the same aforementioned general idea: assigning a score to a search term based on some metric of relevance.

## 3.2 Basic Approach

Research was conducted into the implementations of each of the ranking algorithms.

- Term frequency The term frequency (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 the website. However there are various permutations on this basic formula. The TF formula settled on in the end was

  $$termfrequency = \frac{numberoftimesthewordappearsonthewebsite}{numberofwordsonthewebsite}$$

  to normalise the score a bit, as a word appearing 10 times on a website 50 words long has a different significance to a word appearing 10 times on a website 500 words long.

- Term frequency - inverse document frequency Before the term frequency - inverse document frequency (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. The inverse document frequency formula is designed to take this into account and is calculated as follows

  $$inversedocumentfrequency = log_{10}(\frac{numberofwebsitesinthesearchengineindex}{numberofwebsitescontainingtheword})$$

  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. That is, common words that are likely to appears 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. The TFIDF score is calculated as follows:

  $$TFIDF = TF * IDF$$

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 and therefore give no impact in intersected searches.

- Okapi BM25 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, with optimisation variables too. The version of the formula used in this project is:

$$okapiBM25score = \sum_{i=1}^{n} IDF(w_i) \cdot \frac{TF(w_i) \cdot (k_1 + 1)}{TF(w_i) + k_1 \cdot (1 - b + b \cdot \frac{numberofwordsonthewebsite}{averagenumberofwordsonawebpage})}$$

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

It was also decided that the various permutations of Score classes created would be solely responsible for the calculations. Any required logic was to be handled by the QueryHelper class.

## 3.3 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

and each of the below classes implement this interface.

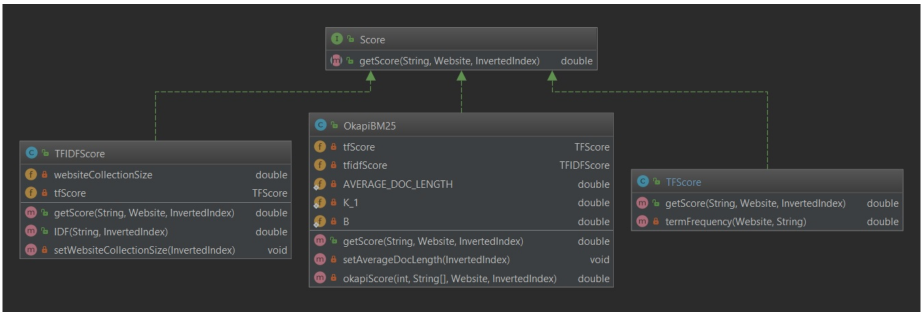
Score
getScore(String, Website, InvertedIndex) double
OkapiBM25
tfScore TFScore
tfidfScore TFIDFScore
AVERAGE_DOC_LENGTH double
K_1 double
B double
getScore(String, Website, InvertedIndex) double
setAverageDocLength(InvertedIndex) void
okapiScore(int, String[], Website, InvertedIndex) double
TFIDFScore
websiteCollectionSize double
tfScore TFScore
getScore(String, Website, InvertedIndex) double
IDF(String, InvertedIndex) double
setWebsiteCollectionSize(InvertedIndex) void
TFScore
getScore(String, Website, InvertedIndex) double
termFrequency(Website, String) double

**Figure 3.1:** UML Diagram for the Software Architecture of Score data structures.

### 3.3.1 IFScore class

The `TFScore` class was the simplest of the three classes to implement, as shown in Figure 3.1. 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.

### 3.3.2 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. The number of websites in the index was built from the index `Map<String, Website>` as below,

```
    private void setWebsiteCollectionSize(InvertedIndex index) {
        Set<Website> siteCollection = new HashSet<>();
        Set<String> words = index.getIndexMap().keySet();
            for(String word : words) {
                List<Website> sites = index.getIndexMap().get(word);
                siteCollection.addAll(sites);
            }
        this.websiteCollectionSize = siteCollection.size();
    }
```

making 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

### 3.3.3 OkapiBM25 class

Again, 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 as before. The optimisation constants and the average document length 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))));
        }
    }
```

## 3.4 Testing considerations

The mathematical correctness of the score calculations were verified using unit tests, which can be found in the ScoreTest.java file. The 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. For the tests on the `OkapiBM25Score` class, single word and multi word query tests were constructed.
The application of the algorithms were tested by RackScoreTest.java file. Test was used to see how do ranking algorithms Term frequency (TF) and Term frequency - inverse document frequency behave on the set up, which tries to imitate real life case. The two ranking algorithms were testing websites which were set up in a fallowing manner: Website 1.com contains 12 words, it repeats "Queen of Denmark" three times and has two "randomwords". Website 3.com contains 14 words, also contain two "randomwords", "Queen of Denmark" is mentioned once and the rest of the words are "of". There are also two websites in the `setUp` that contains only words "of". There role of these two websites is to create a background noise, as it would be expected that in the real websites would contain the filler words like "of" frequently.
It was expected that when TF algorithm would rank higher the websites that have more words matching the search query "Queen of Denmark" compared to "randomwords", therefore Website 1.com, consisting of 12 words, where all words except two "randomwords" match the query ranks lower than the Website 3.com which consist of 14 words, where all words except two "randomwords" match the query. These tests passed.
It was expected that testing TFIDF ranking, would rank Website 1.com higher than Website 3.com, because it considers not only if the words on the website match the search query, but also "jusge" the relevance of the words in the search query, and in the given test there are two websites that contains words "of" and not "Queen" or "Denmark". These tests also passed.

## 3.5 Reflection

After learning about and implementing two different ranking algorithms and testing their implication it was possible to compare the different results they provided. While the Term Frequency algorithm sums together the term frequency of each word in the search query (namely the number of times word appears on the website divided by the number of words on the website), the Term Frequency - Inverse Document Frequency 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 search query would consist of "Queen of Denmark", the TFIDF ranking algorithm would provide more relevant results than TF ranking algorithm, because one could assume that the word "of" would appear significantly more often across different websites, than words "Queen" or "Denmark" and without using log function to lower the significance of the words that are less relevant to the search query, the search results would be less relevant. 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 potential relationship between the search query and the background of the rest of the websites in a abstract and simplified manner.

Based on the theoretical assumptions on the relevance of the two algorithms, when applied to search tasks and the two algorithm behavior in the simplified RackScoreTest, where the website containing "Queen of Denmark" (Website 1.com) three times was ranked higher than the website containing "Queen of Denmark" once and word "of" ten times (Website 3.com) when ranked by TFIDF ranking algorithm, and vice versa when ranked by TF ranking algorithm, it was concluded that between these two algorithms the TFIDF is the most relevant to this search engine project.

Chapter 4

# Chapter 5 Extensions

## 4.1 Section 5.1

Text

## 4.2 Section 5.2

Text

### 4.2.1 Subsection 5.2.1

Text

Chapter 5

# Chapter 6 Conclusion

## 5.1 Section 6

Text

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`

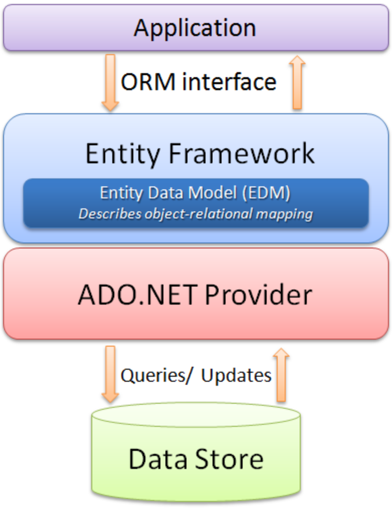

**Figure A.1:** Microsoft Entity Framework

# 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 |
|---|---|---|---|
| A | "wrongfilename.txt" | Exception | FileNotFoundException |
| 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<br>Title1 | *PAGE:http://page1.com<br>Title1<br>word1<br>word2<br>*PAGE:http://page2.com<br>Title2<br>word1<br>word3 | word1<br>word2<br>*PAGE:http://page1.com<br>Title1<br>word1<br>word2<br>*PAGE:http://wrong1.com<br>Title1<br>*PAGE:http://wrong2.com<br>*PAGE:http://wrong3.com<br>Titleword1 Titleword2<br>*PAGE:http://page2.com<br>Title2<br>word1<br>word3 |

# Bibliography

[Bae18] Baeldung. Java treemap vs hashmap. https://www.baeldung.com/java-treemap-vs-hashmap, 2018.

[Ora18a] Oracle. Class hashmap. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html, 2018.

[Ora18b] Oracle. Class treemap. https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html, 2018.