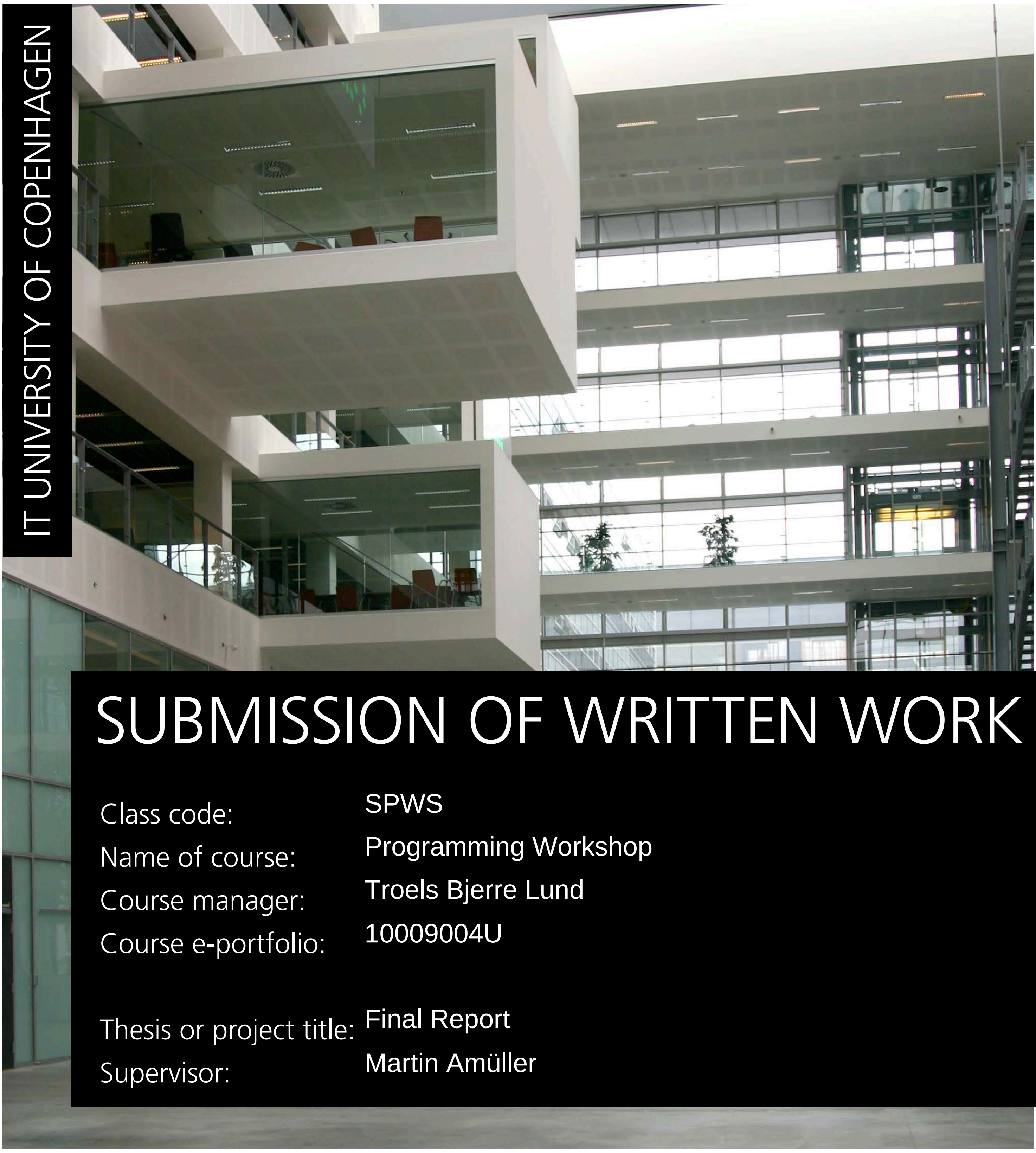

SUBMISSION OF WRITTEN WORK

Class code:	SPWS
Name of course:	Programming Workshop
Course manager:	Troels Bjerre Lund
Course e-portfolio:	10009004U
Thesis or project title:	Final Report
Supervisor:	Martin Amüller

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Ivan Filipov Mladenov	13-07-1991	ivml@itu.dk
2.	Vlad Limbean	14/11-1988	vlli@itu.dk
3.	Rudolf Podkriavacký	24/04-1994	rpod@itu.dk
4.	Joachim Sogn	16/04-1992	jsog@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Programming Workshop 2016

-Final Report-

ITU Copenhagen

Group-C

Ivan Mladenov, **Joachim Sogn**, **Vlad Limbean**, **Rudolf Podkrivacky**

ivml@itu.dk, jsog@itu.dk, vlli@itu.dk, rpod@itu.dk

December 11, 2016

1. Introduction

This document describes the search engine project developed during the Programming Workshop 2016, at the IT University of Copenhagen. In Sections 2–6, we report on solutions to the mandatory assignments posed in the project description. Furthermore, we developed the following extensions which are described in detail in the following sections:

- Okapi BM25 ranking algorithm.
- Improvements to the GUI.
- Implementing a web crawler.
- Performance optimizations to ranking algorithms.

The description for each section is split into the following parts:

- **Task:** a short review of the assignment.
- **Basic Approach:** a short description of the assignment solution.
- **Technical description:** description of the the software architecture for the respective assignment.
- **Testing considerations:** description of the team approach to testing the assignment implementation.
- **Benchmarking/Reflection:** (if applicable) report on benchmarking results for a respective assignment with respective metrics.

1.1. Comment on final handin

The source code accompanying this report is contained in as a single zip file called GroupC-Report.zip. The archive contains two directories: **finished** and **webapp**. The former directory contains the source code which resolves assignments 2 and 3, each in different zip files. The latter directory contains the source code that solves all other parts and includes the changes from Assignment 2 and 3 which are applicable in the final web application. The source code is also available in its entirety on ITU's Github under the url https://github.itu.dk/ivml/Group-C.

Important note: Due to an external library used in the Crawler extension, the library needs to be added in the project settings in IntelliJ. The library is available in **webapp/server/libs/gson-2.8.0.jar**.

2. Assignment 2: Modifying the Basic Search Engine

2.1. Task

The assignment adds the following functionality to the search engine:

- Modify the program such that the program quits if the user provides the search string **"quit"**. Tell the user about this behavior in your program.
- Modify the program such that if no websites match the query, the program outputs **"No website contains the query word."**
- Modify the **FileHelper** such that it only creates a website if both the title is set and the word list contains **at least** one word.

2.2. Basic Approach

2.2.1. Quitting

This task needed small changes to the **SearchEngine.java** class. Quitting the program was achieved by checking whether the input provided equals **"quit"** and breaking out of the while loop in the respective case. It was done in the following way:

```java
if (query.equals("quit"))
{
    System.out.println("Bye :)");
    break;
}
```

2.2.2. No results found

The task was completed by introducing an integer which holds the number of websites found for a given search. Before the search starts, the counter is initialized with a starting value of 0. It is incremented by 1 with every result found. After the search is complete, a check is made on the number of results. The code is as follows:

```java
//start a counter
int counter = 0;

//search happens here

//when a result is found
counter++;

//search finishes

//Check if results were found
if (counter == 0)
{
    System.out.println("No results found!");
}
```

The reason we used a counter instead of a boolean is that we wish to show the number of results in the later stages of development. Specifically, to show the total number of results on the HTML page.

2.2.3. Adding websites only when they have url, title and keywords set

Issue resolved by creating a static function called **canAddWebsite** within FileHelper which checks if the Website's **URL**, **title** are not null and **list of words** is not empty. The list of words is not checked for being null because it is always initialized as an empty ArrayList prior to calling **canAddWebsite**. In case this logical check is false, then the Website object is **not** created.

```java
private static boolean canAddWebsite(String url, String title, List<String> keywords)
{
      return (url != null && title != null && keywords.size() > 0);
}
```

We use **canAddWebsite** because FileHelper creates Website objects in two different places: (1) once when a new website is about to start; (2) and once after reading whole file in order to add the last website.

2.3. Technical description

There were no changes to the basic source code provided in the lecture apart from the ones mentioned above.

2.4. Testing considerations

The basic changes to SearchEngine.java were manually tested by running the program.
The correct behavior of the FileHelper has been tested by writing a JUnit test which checks if a provided text file is parsed correctly. The text file is located in **test-resources/test.txt**. Furthermore, a faulty database was also used to make sure only the valid Website objects were created. The faulty file is located in **test-resources/test-errors.txt**.
We were interested that the FileHelper would read the correct url, title and list of words from the respective file. Consequently, we wanted to test for websites with multiple words in the title or websites with no set title or url and a few or no words. With this set up FileHelper was tested against creating an incorrect Website object. This test is located in **tests/FileHelperErrorTest.java**. Alternatively, the test with the legal database is located in **tests/FileHelperTest.java**.

3. Assignment 3: Inverted Indices, Benchmarking, Documentation, Unit Testing

3.1. Task

This assignment added an index data structure to the search engine. Following the guidance during the lectures, we created two different index implementations -

SimpleIndex and InvertedIndex. Specifically, a list of websites is used for the SimpleIndex, respectively a Map structure for the InvertedIndex.
The rationale behind using a map data structure concerns search time performance. While searching in a list structure would become increasingly time consuming as the list grows, the performance of a map structure would not be similarly encumbered. The main reason is that InvertedIndex stores all keywords, and links them to all websites they appear in. This removes the need of going through all websites in the Index when searching.
In order to decide on what index type to use, we created and executed a benchmark test.

3.2. Basic Approach

We followed the suggestions in the lecture and described an interface Index containing the following methods:

- **build** - Creates the index which holds all websites.
- **lookup** - Searches for websites containing a query provided by user.

The inverted index was implemented by using Java's Map data structure. Java allows several possible implementations:

- Constructor takes a Map parameter.
- Constructor takes a boolean parameter and creates the map.
- Different constructors for each Map type.
- Classes which extends InvertedIndex.
- Methods for creating the map, which need to be called immediately after the constructor.

After internal discussions, the team decided the easiest way to implement different Map data structures was to provide a boolean parameter in the constructor of the index. The main reasoning is there will be two different implementations of InvertedIndex: one uses a **HashMap**; the other uses a **TreeMap**.

3.3. Technical description

We build the index data structures as described in Figure 1 below. The different Map data structures can be switched by changing the value of the boolean parameter when using the constructor.

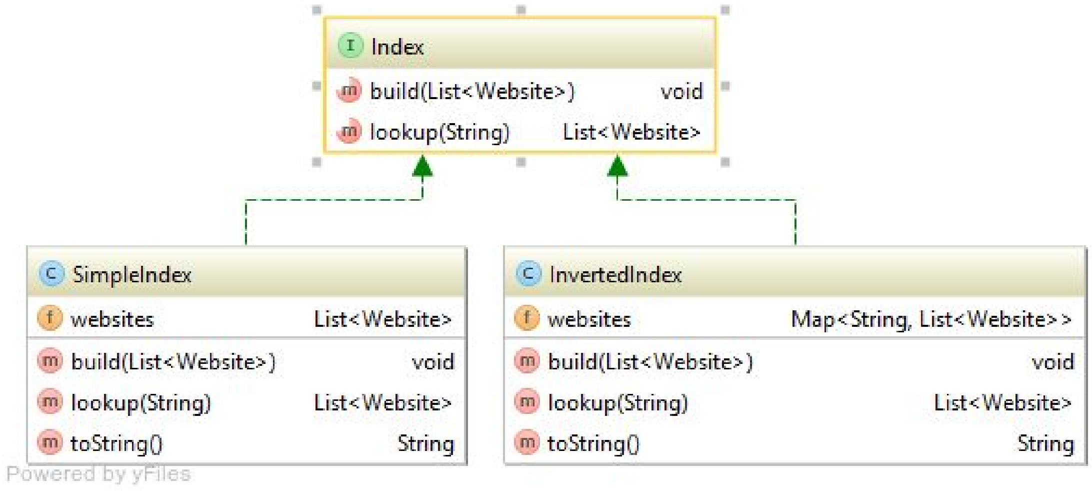
Index
build(List<Website>) void
lookup(String) List<Website>
SimpleIndex
websites List<Website>
build(List<Website>) void
lookup(String) List<Website>
toString() String
InvertedIndex
websites Map<String, List<Website>>
build(List<Website>) void
lookup(String) List<Website>
toString() String
Powered by yFiles

Figure 1: UML diagram of the Index implementation

The **simple index** stores the list of websites received from FileHelper and doesn't change it in any way.

Building the **inverted index** was accomplished by the following code:

```java
//Go through every website in the list
for (Website website : websites)
{
    //Go through every keyword in the website
    for (String word : website.getKeywords())
    {
        //Declare a list variable
        List<Website> emptyList;

        //If the Map contains a key equal to the current word
        if (this.websites.containsKey(word))
        {
            //Get the value of the keyword from the Map, and assign it to emptyList
            emptyList = this.websites.get(word);
        }
        else
        {
            //If the map doesn't contain the keyword, create an empty list
            emptyList = new ArrayList<>();
        }

        //Add the website to the list of websites, only if it's not already added
        if (!emptyList.contains(website))
        {
            emptyList.add(website);
        }
        //Creates a new key in the Map or updates the old one, after adding the current 
website
        this.websites.put(word, emptyList);
    }
}
```

The lookup function for the **simple index** goes through all websites, and checks if the query is inside the website's list of words. If the query is present, then it is added to a list of results. The implementation is as follows:

```
//Initialize a list that will hold all results
List<Website> results = new ArrayList<>();

//Go through all websites
for (Website w : websites)
{
    //If the specific website contains the query, add the website to the list of results
    if (w.getKeywords().contains(query))
    {
        results.add(w);
    }
}

return results;
```

The lookup for the **inverted index** checks if the query exists as a key in the Map which was created in the build function. If the key is present in the map, then it returns the value associated with the key. If it doesn't, it returns an empty list. The code is as follows:

```
//Check if there is a key in the Map equal to the word we are searching for
if (this.websites.containsKey(query))
{
    //It does, so return the value of the key
    return this.websites.get(query);
}
else
{
    //It doesn't, so return an empty list
    return new ArrayList<>();
}
```

3.4. Testing considerations

We verified the correctness of the build and lookup methods with unit tests. These can be found in the file **tests/IndexTest.java**. Here, the **build method** of the Index interface was tested by building the index and checking if the number of websites it stores matches the expected amount. The **lookup method** of the interface was tested by creating a list of test websites. The test was conducted by searching for different queries and looking at the size the list returned list and titles of respective pages.

All three variations of the Index can be tested by **exchanging the constructor** used in the **setUp method**. The example shows how this is achieved:

```java
private Index indexToTest;

@Before
public void setUp() throws Exception
{
    List<Website> testList = FileHelper.parseFile("test-resources/test.txt");

    //Create an inverted index with a TreeMap implementation. Easily changable by replacing
the right part of the line with something else

    this.indexToTest = new InvertedIndex(false);

    this.indexToTest.build(testList);
}

//Continue with tests
```

3.5. Benchmarking/Reflection

A benchmark was conducted in order to compare the search time of all Indices. The benchmark was designed to use the same code for each Index.

The benchmark logs the start time of the benchmark. Then, it performs 1000 searches for each of the following queries - *"america"*, *"denmark"*, *"of"*, *"university"*. Finally, the end time is logged, and the average time is calculated using the following formula, which applies to all index types:

$$Average\ time\ (ns) = \frac{(end\ time - start\ time)}{(1000 \times number\ of\ queries)}$$

The function is executed 3 times - once for each Index.

The results from the benchmark is shown in Figure 2. It can clearly be seen that the **Inverted index** with a **HashMap** is fastest when searching. Simple index always goes through every website in the database when searching, which means the search time grows exponentially as the database grows (it has an average search time of **Θ(n)**). The HashMap's implementation takes a constant time (**Θ(1)**) when searching, while TreeMap has a search time of **Θ(lon(n)**.
As a consequence, in the final version of the search engine will use an InvertedIndex with the HashMap implementation.

Index type	**File used**	**Average search time (nanoseconds)**
Simple Index	enwiki-small.txt	89804
Hash Index	enwiki-small.txt	122
Tree Index	enwiki-small.txt	277
Simple Index	enwiki-medium.txt	4410256
Hash Index	enwiki-medium.txt	113
Tree Index	enwiki-medium.txt	245

Figure 2 - Results from the lookup benchmark

4. Assignment 4 - Implementing a web service

4.1. Task

This assignment involved the implementation of a web service to replace the command-line application developed so far. This allows for an actual implementation of the search engine, outside of a development environment. An example for that would be a website which can communicate with said web service based on user interaction.

4.2. Basic Approach

Steps taken:

- Download the web service prototype from https://github.itu.dk/maau/SPWS16.
- Move downloaded code in the project directory.
- Import prototype in IntelliJ.
- Replace the existing search implementation.

4.3. Technical description

The process involves replacing all classes in the prototype with our implementation of the search engine.

However, this doesn't apply to all classes. **SearchEngine.java** contains code specific to Spring.io. Consequently, instead of replacing the whole file, the following changes were applied:

- The static variable **List<Website>** was replaced with the **Index** interface from Assignment 3.
- The Index object is created and populated with websites (via **FileHelper** and the **build** method) in the **main** method.
- The search was implemented using the lookup method. The results from the search are iterated through and the URL of each result is added to a list. This list is then returned as a response. The code sample below describes how we used the **lookup** method to search:

```java
//Search for the query in the list of websites.
List<Website> resultList = index.lookup(query);

//Create the string list that will be returned by the method
List<String> listToReturn = new ArrayList<>();
for (Website w : resultList)
{
    listToReturn.add(w.getUrl());
}

```

```
//Return the final list of results
return listToReturn;
```

4.4. Testing considerations

All pre-existing tests were ran again to check for errors.
The web service ran correctly and the HTML website was used to manually test for errors in the implementation.

5. Assignment 5 - Refined queries

5.1. Task

This assignment allows the search engine to work with more complex queries. Specifically, the implementation allows users to search for more queries containing multiple words or it allows the use of one or more “OR“ operators. Only websites that feature all words in the query are considered, unless the “OR” operator is used between words. The operator acts as a divisor of queries, and websites are then considered as results if they match any subquery from the division.
The following terms are used from here throughout the rest of the document:

- **Query** - the full input the user entered, eg. “queen denmark OR red roses”.
- **Subquery** - a collection of keywords that are only separated by spaces - “queen denmark”, “red roses”.
- **Query word** - a keyword that is a part of a subquery - “queen”, “denmark”, “red”, “roses”.

5.2. Basic Approach

The implementation was based on suggestions offered during the lecture. We implemented a new class called **QuerySplit**. Its purpose is to handle the search for these complex queries. For example, “queen denmark OR red roses”, would return list of all websites containing at least one of the following subquery:

- “Queen denmark”
- “Red roses”

Since the Index class works correctly when searching for only one word, the subquery is further divided, and a lookup is executed for each word in it. Finally, only websites that feature all words in the subquery are considered as correct results. In the previous example:

- **Queen denmark** returns websites that have both “queen” and “denmark” as keywords. If a website’s keywords features “denmark” but doesn’t have “queen”, it is omitted, and the other way around.
- **Red roses** returns websites that feature both “red” and “roses” as keywords. Same rules apply here - the website needs to have both keywords in order to be considered a result.

5.3. Technical description

Our implementation of the **QuerySplit** class has three methods:

5.3.1. getMatchingWebsites

It splits the query in separate search strings by the **“ OR “** operator, resulting in an array of subqueries. Then, it executes **evaluateFullQuery** which subsequently provides the result.

5.3.2. evaluateFullQuery

It receives the separated subqueries from the previous function. It iterates through all subqueries and further splits them by whitespace. The result from this second splitting is an array of query words (one array of words for each subquery), which is then sent to **evaluateSubQuery**. A “final” list of results is created. The list gets and stores all results that are returned from **evaluateSubQuery**.

5.3.3. evaluateSubQuery

It receives separate query words as an array, performs the search for each word, and keeps a list of results that features websites that were found in all searches (meaning the website feature all query words in the array). This is achieved by using the following code:

```java
//Initialize the list that will hold all search results
List<Website> resultsToReturn = null;

for (String subquery : splitBySpace)
{
    //Search for the subquery, making sure to ignore capitalization
    List<Website> partialResults = index.lookup(subquery.toLowerCase());

    //If the resultsToReturn list is null, that means we are in the first iteration of the
loop
    if (resultsToReturn == null)
    {
        //Since this is the first iteration of the loop, add all results to the list
        resultsToReturn = partialResults;
    }
    else
    {
        //This is NOT the first iteration of the loop, only keep the websites that are in
both lists
        resultsToReturn.retainAll(partialResults);
    }
}

```

To summarize, the **retainAll** function is the way we store websites which contain all keywords from a given subquery.

In order to support searching for complex queries, the main search function in the **SearchEngine** class needed to be modified. Instead of searching directly in the Index, the function was changed to execute **QuerySplit.getMatchingWebsites**, which consequently looks up in the index. No other changes were necessary in the class, since the **getMatchingWebsites** function returns a list of websites.

5.4. Testing considerations

The implementation was checked by running the code, performing the search, and manually checking if the results were correct.

Furthermore, unit tests were created to verify the correctness of the code. The tests are located in the tests folder, more specifically in **QuerySplitTest.java**. The tests check the validity of the search when looking up the following queries:

- A single word that should return results, to verify that this type of search hasn't changed.
- A single word that shouldn't return any results.
- A longer query that contains two words, separated only by a whitespace. The result is checked to see if it only has websites that feature both words.
- A longer query that contains four words, separated by "OR" in two smaller subqueries. The result is checked to see if it contains all websites from the two subqueries.
- A query that features two words, separated by a whitespace. The two words are featured in a number of websites, but no websites contain both words. Therefore, the test checks if the number of results is 0.
- A query that features two words, separated by two whitespaces. The number of whitespaces should not matter, and the whole query should be considered as if it has only 1 whitespace.
- A query that features two words, separated by "OR" and a couple of whitespaces between. The number of whitespaces should be irrelevant when searching; the test should find results as if there were only one whitespace before and after the "OR" operator.
- A query that features only whitespaces, with no words. The QuerySplit class should return zero results and not crash.
- A query that features only whitespaces and one "OR" operator. Same as the previous test, it should return zero results, without crashing.
- A query that features multiple "OR" operators. The expected number of results is checked, and the titles of all results are checked to make sure the correct websites are returned.

5.5. Benchmarking/Reflection

A benchmark was performed to check the speed of the complex query search compared to the previous implementation, which searches for single words only. While the process is similar to the lookup benchmark in Section 3, the queries were modified to support multiple words. The query array is as follows - *“america”*, *”america denmark”*, *“of university OR denmark copenhagen”*, *“america OR denmark”*. The results are shown in Figure 3. As a summary, search time is prolonged for a number of reasons:

- The whole query is split up twice - once by the “OR” operator, and again by whitespace
- QuerySplit looks up multiple words, instead of just one.
- retainAll combines results for multiple keywords which are separated by whitespace.
- When a search for a subquery finishes (query which features only whitespaces between words), results from the search are added only if they are not already added from a previous subquery search.

Index type	**File used**	**Average search time (nanoseconds)**
Simple Index	enwiki-small.txt	1535600
Hash Index	enwiki-small.txt	6622
Tree Index	enwiki-small.txt	6889
Simple Index	enwiki-medium.txt	28748128
Hash Index	enwiki-medium.txt	57654
Tree Index	enwiki-medium.txt	78624

Figure 3 - Results from the QuerySplit benchmark

6. Assignment 6 - Ranking algorithms

6.1. Task

The final mandatory assignment provides the search engine with means to rank the websites. The overall goal is to show the most appropriate websites first, keeping in mind what the user searched for. The returned list of websites is ultimately ordered by frequency of words / phrases per respective website.

6.2. Basic Approach

Scoring a website is considered as calculating a numerical representation of how much a website is relevant to the query a user searches for. The higher the score of a website, the more relevant it is to the query. Websites are then ordered by their scores, with the highest scoring website appearing first, while the lowest scoring website - last.

To support a ranked ordering of results, we created an interface called Score. This allows for the use of different ranking algorithms which will implement the Score interface. The main function each implementation of the interface needs to have is **getScore**, which calculates a website's score towards a query word.

Our first implementation of the **Score** interface is **ScoreTFIDF.java**. It calculates the "term frequency-inverse document frequency". TF-IDF considers how many times a query word is seen on a website, and how often in general is the word seen in the whole database. The basic formula is:

$TFIDF(w,\ S,\ D)\ =\ TF(w,\ S)\ \times IDF(w,\ D)$, where:

- **TFIDF** is the final score
- **w** is the query word
- **S** is the specific website that we are calculating a score for
- **D** is the number of websites that contain the query word

- **TF** is the Term Frequency of the word in the specific website. The term frequency represents the amount of times a word is contained in a website.
- **IDF** is the Inverse Document Frequency of the word. It is calculated based on the total number of websites, and the amount of websites that contain the word. This puts a "weight" on uncommon words in favour of common words, for example prepositions and pronouns.

When calculating the scores, every word in the query is considered. Query words are used to calculate the score of a website. The score of a subquery is considered the sum of scores for each query word. For each website, only the subquery with the largest score is taken into account when ordering.

6.3. Technical description

Note: Optimizations to the approach were made after implementing the scoring algorithm and the challenge - **Okapi BM25**. These are documented in **Section 10**, along with a performance benchmark report.

The class **ScoreTFIDF** implements two methods for each part of the equation - one that calculates the term frequency, and another which calculates the inverse document frequency. The final score of a website is then calculated by multiplying the results from both methods.

6.3.1. CalculateTermFrequency

The function goes through all words in a specific website. This is done in order to calculate the number of times the query word is seen in the website. The implementation is as follows:

```java
//Initialize a counter
int counter = 0;

//Go through all words in the website's list of keywords
for(String word : website.getKeywords())
{
    //Increment the counter when the keyword matches the word in the list
    if(word.equals(keyword))
    {
        counter++;
    }
}

//Return the final term frequency
return counter;
```

6.3.2. calculateInverseDocumentFrequency

The Inverse document frequency calculates how uncommon the query word is. IDF is used as a weighting factor - the more uncommon the word, the more important it is when ranking. It is calculated by taking the total number of websites in the database, divided by the number of websites which feature the query word. Then the base 2 logarithm of that value is calculated. The ultimate result represents how rare a query is across the entire list of websites. Its formula is as follows:

$$IDF(w,\ D)\ =\ log_2(\frac{|d|}{n})$$, where:

- **d** is the total number of websites
- **n** is the number of websites that contain the query

The function gets the total number of websites in the Index and calculates the number of websites that contain the query word by looking up in the index. Since the formula divides the two numbers, a check is required to make sure the denominator is not 0. Finally, the logarithm (base 2) of the fraction is calculated and returned. The code is as follows:

```
//Get the total number of websites
int numberOfWebsites = index.getSize();
//Get the total number of websites
int numberOfMatches = index.lookup(keyword).size();

//If the denominator is 0, the inverse document frequency cannot be calculated
if (numberOfMatches == 0)
{
    return 0;
}

//Calculate the inverse document frequency
double result = Math.log10((double)numberOfWebsites / (double)numberOfMatches) /
Math.log10(2);
```

```

//Return the result
return result;
```

The code example uses a method called **getSize**, available in the index. This was implemented to easily get the size of the database, without having long computations during runtime. The problem is seen specifically in **InvertedIndex**, due to the difficulties of iterating through a map and constantly checking if a website was already accounted for. The specific implementations implementations are as follows:

- **SimpleIndex** - returns the size of the list of websites.
- **InvertedIndex** - a new instance variable was added, which stores the size of the database. The size is determined in the build method, and is equal to the size of the list of websites that is passed as a parameter. The getSize method returns the value of this instance variable.

6.3.3. GetScore

The method computes the total score for a given word and a specific website. It does so by executing the methods described above and then returning the product of their respective results. This method is the one defined in the **Score** interface.

6.3.4. Implementing the scoring algorithm

The **QuerySplit** class was changed in order to use the ranking algorithm. In the second function, **evaluateFullQuery**, the list of websites is replaced with a Map object. It holds a website as a key, and a score as a value. When a website is featured as a result in more than one subquery, the score is calculated for both subqueries, but only the higher one is kept. This is reflected in the aforementioned map - the value is the highest calculated score.

When a subquery is evaluated and results are returned, each website from the results has its score calculated for each of the words in the subquery. The scores for each individual word are summed up, representing the final score of the website towards the subquery. This calculation is done in a newly created function, **calculateSubqueryRanking**:

```
calculateSubqueryRanking(String[] splitByWhitespace, Website website, Index indexToUse,
Score rankingHandler)
{
    //Initialize the final score holder
    double scoreForQuery = 0;

    //Go through all words in the subquery
    for (String substring : splitByWhitespace)
    {
        //Calculate the score for each word and add it to the final score
        scoreForQuery += rankingHandler.getScore(substring.toLowerCase(), website);
    }

    //Return the final score of the whole query
    return scoreForQuery;
}
```

The calculated score is compared to the one that is saved in the map and only replaces if it is larger than the stored value. If the map doesn't contain the website, then the website is added to the map along with its respective score that was just calculated. This sequence is executed in another function, called **updateWebsiteInMap**:

```
updateWebsiteInMap(Website websiteToUpdate, double newlyCalculatedScore, Map<Website,
Double> currentMap)
{
    //Check is the website is already in the ranking map
    if (currentMap.containsKey(websiteToUpdate))
    {
        //The map contains the website, so check its' currently saved score
        double scoreInMap = currentMap.get(websiteToUpdate);

        //Check if the calculated score is bigger than the score saved in the map
        if (newlyCalculatedScore > scoreInMap)
        {
            //Update the score because we found a max
            currentMap.put(websiteToUpdate, newlyCalculatedScore);
        }
    }
    else
    {
        //This is the first time we add this website to the rankin
        currentMap.put(websiteToUpdate, newlyCalculatedScore);
    }

    return currentMap;
}
```

The following changes in **evaluateFullQuery** were necessary to implement the aforementioned functions:

```
//Initialize the map that holds all results. Key is the website, value is its rank in the 
query
Map<Website, Double> rankingMap = new HashMap<>();

//Iterate through the subqueries
for (String subquery : splitByOr)
{
    //Split by whitespace (as before)
    String[] splitBySpace = fullquery.split(" ");

    //Execute search for the query words array (as before)
    List<Website> partialResults = evaluateSubQuery(splitBySpace, index);

    //Iterate through the results
    for (Website w : partialResults)
    {
        //Calculate the ranking score of the website
        double scoreForSubquery = calculateSubqueryRanking(splitBySpace, w, index, 
rankingHandler);

        //Update the ranking map based on the score that was just calculated
        rankingMap = updateWebsiteInMap(w, scoreForSubquery, rankingMap);
    }
}
```

After the loop finishes, the ranking is final. The websites need to be ordered in descending order. This is done via a large lambda expression, which sorts the map based on scores. Then it takes the keys (or websites) of the map and creates a list from them, without further changing the order.

6.4. Testing considerations

The implementation was checked using unit tests. They are located in the **tests/RankingTest.java**. The ranking of results is tested for the following cases:

- A single word query that returns 1 website. The website's score is calculated and compared to the expected score.
- A single word query that returns 2 websites. Each website has its score calculated and compared to the expected score.
- A query containing two words, separated by whitespace. The ranking algorithm should sum their scores. Each website's score is calculated and compared (word by

word), and their ordering is also checked. The second website in the results should have a higher score when scores are summed, therefore it should be placed first in the final list.

- A query containing two words separated by "OR". We expect a result of total 3 websites. The first website contains word1; the second website contains word2; the third contains both word1 and word2. In this specific case, the scoring algorithm should only consider the highest score per word instead of summing the scores of both words. The correct outcome of the test occurs when the website ranked is second in the final list.

6.5. Benchmarking/Reflection

For the purpose of consolidation, the benchmark results to both TFIDF and BM25 have been moved to Section 10 of this document.

7. Extension assignment - Okapi BM25

7.1. Task

The extension covers the implementation of the **Okapi BM25** algorithm. BM25 incorporates and builds on the previous scoring functions presented earlier.
BM25 balances the bias created by the **Term Frequency** calculation by taking into account the length of a given document relative to the TF of a given query. For example, if a word is featured twice in document 1 (with length 10), and twice in document 2 (with length 256), then the TF of the word in no way accounts for the document length. Which, considering real world search engine necessities would greatly skew the results.

7.2. Basic Approach

We implemented the scoring algorithm in the **ScoreBM25.java** class. Similar to the **TF-IDF** score calculation in **Section 6**, BM25 implements the Score interface. The getScore

method takes an index and calculates the **term frequency*** and **inverse document frequency**. Finally, it returns the product of the two.
The basic formula is:

$$BM25(w, S, D) = TF*(w, S, D) * IDF(w, D)$$, where:

- **BM25** is the final result of the Okapi BM25 score
- **TF** $*$ is term frequency*
- **IDF** is the Inverse Document Frequency of the word, same as the one in **Section 6**.
- **W** is the query word
- **S** is the website containing the query word w
- **D** is the number of all websites

Term frequency* is an upgrade over term frequency, based on the fact that it takes into account the length of a website. This balances the ranking for longer documents to not be higher only due to their length. The basic formula for term frequency* is:

$$TF*(w, S, D) = TF(w, S) * \frac{(k+1)}{k*(1 - b + b * dl/avdl) + tf(w, S)}$$, where:

- **TF** is the Term Frequency of the word in the specific website, same as **Section 6**.
- **k** is a constant between 1.2 and 2.0, in our case - 1.75
- **b** - constant at 0.75
- **dl** - number of words in the website S
- **avdl** - average number of words per website across all websites

7.3. Technical description

Note: Optimizations to the approach were made after implementing the Okapi BM-25 scoring algorithm. These are documented in **Section 10**, along with a performance benchmark report.

The implementation is similar to the one in Section 6. A method is implemented for each part of the equation - calculating the term frequency, calculating the inverse document frequency, and calculating term frequency*.

7.3.1 calculateTermFrequency

The implementation is the same as in the TF-IDF algorithm in **section 6**. The method counts the number of times a query word is featured in the website.

7.3.2 calculateInverseDocumentFrequency

Similar to **calculateTermFrequency**, the implementation doesn't differ from the one seen in the TF-IDF algorithm in **section 6**. The method divides the total number of websites with the number of websites that feature a query word, and the logarithm (base 2) of the fraction is calculated.

7.3.3 termFrequencyStar

The method starts by executing the **calculateTermFrequency** method, as its value is used twice in the final calculation. The total number of words in the website is retrieved, by using the list of words' size method. The average number of words across all websites is retrieved from the index. Finally, the whole equation is calculated. The k and b values are predefined constants.

```
//Define equation constants
double k = 1.75;
double b = 0.75;

//Calculate the term frequency of the website towards the query word
double TF = termFrequency(query, website);

//Calculate the number of words divided by the average number of words
double words = website.getKeywords().size() / index.getAverageWordsCount();

//Calculate the denominator of the whole equation
double div = k* (1-b + (b * words)) + TF;

//Calculate the final result
double result = TF * (k+1)/ div;

return result;
```

The code example uses a method called **getAverageWordsCount**, available in the index. It was implemented to easily access the average number of words across the whole database. The specific implementations are as follows:

- **SimpleIndex** - iterates through the list of websites in order to sum their words count, and then divides the sum by the number of websites.
- **InvertedIndex** - a new instance variable was added, which stores the average words count. Its value is determined in the build method, where the sum of words is calculated while iterating through the list of websites. After the iteration, the sum is divided by the number of websites and the result is stored in the new instance variable. The **getAverageWordsCount** method returns the value of this instance variable.

7.4. Testing considerations

The same tests from **Assignment 6** were ran using the **Okapi BM25** algorithm to ensure the correctness of the implementation.

7.5. Benchmarking/Reflection

For the purpose of consolidation, the benchmark results to both BM25 and TF-IDF have been moved to **Section 10** of this document.

8. Extension assignment - Improve the client GUI

8.1. Task

The main purpose of this extension is to change the client code, such that the visuals of the website are improved. As explained in ***Assignment 4***, we were provided with basic web files that represent visuals (**style.css**), web-page (**index.html**) and client side script (**code.js**) which communicates with our web service and shows the results from searching in the web-page.

Extra activities, like adding pagination to results and auto-complete to queries, **were not developed** due to time constraints.

8.2. Basic Approach

In this extension we performed changes to the files mentioned in the section above. However, there were also changes in how we return results in main search function of the web service - **SearchEngine.search**. The main reason for the change was to allow titles and summaries to be shown along with the hyperlink. This was achieved by returning a JSON holding all properties of a website, instead of only the hyperlinks. Furthermore, the **Website** class was changed to hold the first 250 characters of an extract. This is done by adding a new instance variable, the value of which is determined when reading text files in **FileHelper.java**.

8.3. Technical description

Index.html was changed to show a logo in the middle of the page, and a link to the github repository, in the footer of the page. Furthermore, classes were added to most HTML elements, which provides us with an easy way to style them in the CSS file, style.css. The added elements are as follows:

```html
<head>
    <!-- added to <head> element, links to the main font we are using -->
    <link href="https://fonts.googleapis.com/css?family=Roboto" rel="stylesheet">
</head>

<!-- replaces the <h1> tag that previously showed the title of the website -->
<img src="logo.png" />

<!-- Footer with link to github -->
<footer class="center">
    <div class="link-text">
        <a href="https://github.itu.dk/ivml/Group-C">Link to github</a>
    </div>
</footer>
```

Style.css represents the CSS styling of the elements in the website, making the website more appealing. The main changes here are:

- Almost every element is aligned to the center of the page

- Text elements, excluding hyperlinks, are colored red
- The results of the search are placed in a wrapper, which has a light-red background color and is positioned to the middle of a page.
- A specific font is used and line spacing is increased to improve readability.

These changes look as follows:

```
<!-- Ensures the line spacing is bigger and sets the default font -->
html {
    line-height: 1.5;
    font-family: "Roboto", sans-serif;
}

<!-- Ensures the image has some space before it, and that it resizes based on the width of
the website -->
img {
    margin: 5rem 0 0 0;
    width: 100%;
}

<!-- Ensures all elements with this class have a red coloring and are not underlined -->
.red-text {
    text-decoration: none;
    color: #F44336;
}

<!-- Ensures all text in elements with this class is centered in the middle of the screen
-->
.center {
    text-align: center;
}

<!-- The wrapper which holds all results will now have a light-pink background color and
light-grey color of text -->
#urllist {
       color: #969696;
       background-color: rgb(255,235,238);
}
```

Code.js represents the javascript that sends the user request to the server and processes the result received. The following changes were made to it:

- The time the request takes to respond is calculated inside the javascript code and it is shown to the user when presenting the results from searching.
- The response from the server will contain an array of websites, each having a title, URL and description. The array is iterated through and a wrapping div element is created for each website. Then, the title is appended as a child of this div, and the

same is done for the URL and description. Finally, all div elements are inserted inside the previously available div element, #urllist, which holds results.

The added code looks as follows:

```
//After user presses search button, before sending request
var start = new Date();

//Request finishes successfully

//Calculate the time the search took (including sending the request and receiving a
response)
var totalTime = new Date() - start;

//Show number of results and time the search took in div element with id "responsesize"
$("#responsesize").html("<p>" + data.length + " results found in " + totalTime + " ms</p>");

//Add a div that will hold all results to a buffer. This buffer will later be added in the
HTML
buffer = "<div class=\"results-container\">";

//Go through each element in the received data
$.each(data, function(index, value) {
    //Start with a div that will hold only one website
    buffer += "<div class=\"result-entry\">";

    //Add the title of the page as a anchor element with an importance of heading 3
    buffer += "<a class=\"red-text\" href=\"" + value.url + "\"><h3>" + value.title +
"</h3></a>";

    //Add the url of the page, with a class of link-text (so the CSS can make it blue)
    buffer += "<div class=\"link-text\"><a href=\"" + value.url + "\">" + value.url +
"</a></div>";

    //Add the extract of the website in a span element
    buffer += "<span>" + value.extract + "</span>";

    //Close the div that holds only one website
    buffer += "</div>";
});

//Close the wrapping div that holds all results
buffer += "</div>";

//Push the buffer in the div with ID "urllist" in the HTML page. This visualizes the
results.
$("#urllist").html(buffer);
```

In order for code.js to read titles, URL's and extracts, changes were required to three java classes - **SearchEngine.java**, **Website.java** and **FileHelper.java**.

The **Website** class needed to keep track of the extract so it can be presented easily after searching. Due to the changes made in the class in **Assignment 10** (removing the list of keywords in exchange for a Map of keywords and their respective term frequency in the website), generating extracts on runtime wasn't a viable option. Even without these changes, generating said extracts when searching takes time. For these reasons, we added a new String as an instance variable. Its task is to store the first few sentences of the extract. The extract is generated by reading the words in order and appending them one after another.

The generation of all websites' extracts was added in **FileHelper**. A StringBuilder object is initialized when the process of reading text files starts. If the current line in the file is a keyword it is appended to the StringBuilder object. This is done only when the extract is less than 250 characters, which ensures extracts don't become too long. When the text file reaches the end of a Website, it is created as before, but with an added parameter in the constructor - the string representation of the extract. After that, the StringBuilder is cleared so the extract of the next website doesn't have words from the previous. The code looks as follows:

```java
//Static variable in the class
private final static int extractLength = 250;

//At the start of the process
StringBuilder extract = new StringBuilder();

//Start reading the text file
//Read a line from the text file
//The line is a keyword

//Add the current line to the extract, only if the extract's length is less than
extractLength characters
if (extract.length() < extractLength)
{
    extract.append(currentLine);
    extract.append(" ");
}


//When adding a website to the List of websites, add the extract as a new parameter
finalList.add(new Website(url, title, extract.toString(), frequencies, wordsCount));
```

Finally, **SearchEngine.java** was modified to support sending back the extract and title as well. This is achieved by formatting the whole results List as a JSON, instead of only the URL's of said results.

8.4 Testing considerations

We have tried entering simple queries and checked in both the IntelliJ and the browser console whether the query had been registered by our server. Furthermore, checks were made to ensure edge cases like zero results, or an empty query, don't crash the website and the server. The website was manually tested with different browsers to see if everything works as expected. Several screen sizes were also manually tested via the Chrome developer tools. This helped us resolve any problems with elements that could have been offset of our view, as a result of using different screen size or device.

9. Extension assignment - Implement a web crawler

9.1. Task

The webcrawler implementation was supposed to support our search engine in finding and indexing external websites found on the internet. Our webcrawler starts off with a wikipedia article extracted via the Wikipedia API and continues on its path for a set amount of crawls. The final result is a database of websites that will replace the ones available in the prototype.
After the crawler was developed, it was ran and a large database of websites was created. The file is present in the attached zip file, and can be safely used when running the search engine. The file is located in **webapp/server/data/crawled-medium.txt**.

9.2. Basic Approach

In order to keep things simple and not jump in deep water, we reached the decision to only crawl Wikipedia articles, instead of the whole web. This allows us to focus on one parser only, which should work for every article.

As previously mentioned, we start off with one wikipedia article that we decided was a good starting point. The article's contents were parsed through, and all articles mentioned in the first one are also crawled through. The process continues until we have a database of 20,000 websites. In order to get an article's contents, we use the **Wikipedia API**. By constructing a specific link and making a request using this link, the API sends us back a JSON format string containing the following data:

- The **title** of the article.
- The **extract** of the article - we only read the first paragraph of the article.
- The **URL** of the article
- A **list of other articles** that are mentioned in the article.

The crawler parses the data to an easy to use format, writes the article's contents to a file, and repeats the same process for all articles linked in the current one.

9.3. Technical description

For the operation of the crawler, we created two new classes:

1) **Crawler** - contains the main logic of the crawler
2) **WikiJson** - used in Crawler to parse the JSON response and convert articles to this class.

Both are packaged as **searchengine.Crawler**, and appear in the folder src/main/java/searchengine/Crawler/.

The crawler has a number of methods, created to separate the different stages of the process.

9.3.1 requestMaker

The method's task is to generate a request that will be sent to the Wikipedia API, execute it, and wait for a response. The request differs based on the articles we want to read. The difference is seen in the URL of the request. The base URL is as follows:

https://en.wikipedia.org/w/api.php?action=query&format=json&prop=extracts%7Cinfo%7Clinks&titles=*{titles}*&utf8=1&exlimit=max&exintro=1&explaintext=1&inprop=url&pllimit=max

In order to perform a successful request, our code needs to replace the ***{titles}*** part of the URL with a string representation of the articles we want to crawl through. The API returns up to 20 extracts per request. For this reason, we also limit the number of titles in the URL to 20.

The titles are then added together, separated by a "|" symbol (as the Wikipedia API documentation states), and the final string is encoded to replace special characters like whitespace and the aforementioned vertical bar with escaping characters.

The final URL of the request is generated by replacing the {titles} part with the newly constructed string of titles.

Finally, a connection with the API is opened, the request is sent, and the response is read, and the resulting JSON is sent to the **wikiReader** method.

```java
//Wait for 2 second to ensure we don't DDOS
Thread.sleep(2000);

//The string of titles is generated
StringBuilder allTitlesString = new StringBuilder();
for (int i = 0; i < titles.size(); i++)
{
    allTitlesString.append(titles.get(i));
    if (i < titles.size() - 1)
    {
        //A vertical bar is added if the current title is not the last one
        allTitlesString.append("|");
    }
}

//Encode the titles string (because URL's shouldn't have whitespace and other symbols
String titlesURL = URLEncoder.encode(allTitlesString.toString(), "UTF-8");

//Create the final URL
String currentURL = preURL.concat(titlesURL).concat(postURL).concat(continueStatement);
URL currentLink = new URL(currentURL);

//Open a connection to the Wikipedia API
URLConnection beyond = currentLink.openConnection();
//Send the request and retrieve the response
InputStream websiteReader = beyond.getInputStream();

//Create a scanner that will read the response
Scanner sc = new Scanner(new BufferedInputStream(websiteReader), "UTF-8");

//Used to read the entiner input. For some reason it doesn't read without the pattern
```

```java
String result = sc.useDelimiter(Pattern.compile("\\A")).next();

//Send the result to the JSON reader method
wikiReader(result);
```

9.3.2 wikiReader

This method is responsible with reading the whole JSON, retrieved from a Wikipedia request, and parsing it to the Java class we created - **WikiJson**. For every article in the request, a new instance of the WikiJson class will be created. The request returns a number of properties that we don't need for the purpose of the Crawler (eg. last revision date, length of the article). For this reason, WikiJson only has variables that we use:

- **Article title**
- **Extract**
- **Article URL**
- A list of outward **wikipedia articles**.

To parse from a JSON response to a readable list of **WikiJson** objects, we used the deserialization library Gson, maintained by Google. All articles received in the response are contained in a parenting "pages" object, which itself is a child of a "query" object. Our code reads through both and creates the final list of articles.

To further complicate things, the Wikipedia API returns only up to 500 links. In the case of the requested articles having more links, a "continue" object is returned by the API. If that happens, the same request needs to be made in the aforementioned requestMaker method, with the value of the "continue" object added in the URL.

For this reason, when parsing a JSON, the resulting WikiJson objects are saved in a Map instance variables. In the case of a "continue" object existing, the method turns back to requestMaker. The map's contents are updated every time a "continue" request is made. If the "continue" object doesn't exist, the method knows that all links and extracts are finally received, and the WikiJson objects for these specific articles will no longer need an update. In this case, the method's job is finished for the specific collection of articles. The code is as follows:

```java
//Create the objects that parse through a JSON string
Gson jsonLoader = new Gson();
JsonParser parser = new JsonParser();

//Parse the results from the request and generate an object from it.
JsonObject firstParse = parser.parse(result).getAsJsonObject();

//If the JSON doesn't feature a "query" object, that means there's no more information to
read through
if (!firstParse.has("query"))
{
    return;
}

//See if there is a "continue" statement in the response
String continueStatement = "";
if (firstParse.has("continue"))
{
    JsonObject continueObject = firstParse.getAsJsonObject("continue");
    if (continueObject.has("plcontinue"))
    {
        //A continue statement exists, so we need to read it and save its value
        continueStatement =
                    firstParse.getAsJsonObject("continue")
                    .getAsJsonPrimitive("plcontinue").getAsString();
    }
}

//Read the "query" object and its child object - "pages", to get an object with all articles
JsonObject jsonObject = firstParse.getAsJsonObject("query").getAsJsonObject("pages");

//Iterate through the articles inside the newly created JsonObject

//For each articles in jsonObject, parse it to a WikiJson class using the Gson library
WikiJson wikiPage = jsonLoader.fromJson(currentLine.getValue(), WikiJson.class);

//Check if the map of articles (an instance variable) already has the articles
if (partiallyCrawledSites.containsKey(wikiPage.title))
{
    //It does, which means the article has new data to be added

    //Get the previous value
    WikiJson savedSite = partiallyCrawledSites.get(wikiPage.title);

    //Add all links in the currently read article object to the article object in the map
    savedSite.links.addAll(wikiPage.links);

    //Update the article in the map
    partiallyCrawledSites.put(savedSite.title, savedSite);
}
else
{
    //The map doesn't feature the article yet, so save it
    partiallyCrawledSites.put(wikiPage.title, wikiPage);
}

```

```java
//After the iterator finishes, check if a continue statement was read
if (!continueStatement.isEmpty())
{
    //There is a continue object in the JSON, which means we need to execute the same
request, but with the following string appended to the end of it
    String finalContinueStatement = "&plcontinue=" + URLEncoder.encode(continueStatement,
"UTF-8");

    //The continue postfix was generated, execute the request again (this time with it)
    requestMaker(sitesWhenContinuing, finalContinueStatement);
}
```

9.3.3 wikiWriter

Once the **requestMaker** method doesn't have a "continue" object in the response, the articles in the map will no longer be updated. This means that we can safely consider their state as final. For this reason, we can actually write them to a .txt file without worrying about new words being added. The method **wikiWriter** goes through all articles in the map, and writes its contents to a text file, only if the article has an extract (some articles in Wikipedia appeared to miss having a extract). The format is identical to that of the enwiki-tiny, medium or large files.

```java
for (WikiJson wikiPage : crawledSites.values())
{
    //Check if the article has an extract. If it doesn't skip it.
    if (wikiPage.extract == null || wikiPage.extract.isEmpty())
    {
        continue;
    }

    //Write the URL and the title of the article
    saveFile.println("*PAGE:" + wikiPage.fullurl);
    saveFile.println(wikiPage.title);

    //Get all keywords in the extract by splitting by a whitespace
    String[] extractWords = wikiPage.extract.split(" ");

    //Iterate through the words

    //Write the current word in the iterator to the text file
    saveFile.println(currentWord);
}
```

Here, it should be noted that all punctuation and special characters are kept and written to the text file. This is done so the final extracts in the website are presented in a readable form. These special characters are removed when starting the search engine and reading the text file in **FileHelper**. When the current line is a keyword, it is appended to the extract, and special characters are removed prior to adding the line to the term frequencies map:

```
//Start reading the file

//Iterate through all lines

//Current line is a keyword

//Append line to extract

//The regular expression removes all non-alphanumeric characters (only a-z and 0-9
characters are left)
//Current line has any potential capitalization removed
String lowerCaseWord = currentLine.replaceAll("[^a-zA-Z0-9']", "").toLowerCase();

//Add lowerCaseWord to frequencies map
```

9.3.4 addNewLinksToCrawl

Once again, when **requestMaker** doesn't feature a "continue" object in the response, that means the articles in the map will no longer be updated and no more links will be added to them. The **addNewLinksToCrawl** method is then executed, which iterates through the articles in the map, and adds all their external links to the queue of titles. This queue is later used in subsequent requests. Only titles that have not been read are added to the queue, to remove duplicates in the final text file. It should be noted that links are only added if they are not referring to any special types of articles, as seen in the code example below:

```
for (WikiJson site : crawledSites.values())
{
    for (WikiJson.JsonWikiLinks link : site.links)
    {
        //The following links are considered "special" articles in Wikipedia, without any
content the search engine is interested in. Skip them
        if (link.title.startsWith("Category:") ||
            link.title.startsWith("Portal:") ||
            link.title.startsWith("Help:") ||
            link.title.startsWith("Template talk:") ||
            link.title.startsWith("File:") ||
            link.title.startsWith("Wikipedia:") ||
            link.title.contains("(disambiguation)") ||
            link.title.startsWith("List of"))
        {
            continue;
        }
        //Only add the link if it’s never been read and it’s not in the queue
        if (!writtenSites.contains(link.title))
            titles.offer(link.title);
    }
}
```

9.3.5 queueCrawler

The method handles execution of the whole crawler. It starts by reading titles from the queue. Then, it sends them to the **requestMaker** method, and awaits the map to be populated with articles. When the articles reach their final state, the **wikiWriter** and **addNewLinksToCrawl** methods are executed, in order to populate the text file with the articles in the map, and to populate the queue with new links to crawl through. Once the maximum number of websites are crawled, the method closes the .txt file and the crawler finishes working.

```
//Execute continuously until the articles in the text file reach the maxWebsites value
(20,000 right now), and while there are still article titles in the queue
while (titles.size() > 0 && crawlCount < maxWebsites)
{
    //Initialize a list of titles, which holds the titles that will be requested next
    List<String> titlesToCrawlTogether = new ArrayList<>();

    //Populate the list with up to 20 article titles
    while (titles.size() > 0 && titlesToCrawlTogether.size() < maxWebsitesInOneRequest)
    {
        //Get the first article in the queue and add it to the list
        String currentTitle = titles.poll();
        titlesToCrawlTogether.add(currentTitle);
    }

    //Clear the map of articles, since we are starting a new request sequence
    crawledSites.clear();
    requestMaker(titlesToCrawlTogether, "");

    //After all requests for the 20 titles finish, write them to a file
    wikiWriter();
    addTitlesToQueue();
}

//Close the stream once the file is written
saveFile.close();
```

9.4. Testing considerations

The crawler was tested initially with a limited number of websites. As the pages were crawled, read and saved, we would verify if the information in the generated text file was consistent with the website information.
When the final database was created, the search engine was ran using it, and tested manually to ensure no duplicate, no meaningless and no empty articles exist in the database.

10. Extension assignment - Performance Optimizations to Ranking Algorithms

10.1. Task

After completing the implementation of both **TF-IDF** and **Okapi BM25** we discovered a lot of extra calculations were performed every time a search is performed. The main task in this section was to improve the performance of all scoring algorithms during runtime by introducing various optimizations. A lot of the computations done at runtime are redundant and can be calculated when starting the web service.

10.2. Basic Approach

The following changes were made to support faster scoring calculations:

- A **term frequency map** is added to all websites, with the term frequency for each word calculated as FileHelper is reading the given text file.
- The **inverse document frequency** used to look up the number of websites featuring a query word. This exponentially increased the amount of times the lookup method of an index was executed (once for every word in a query multiplied by the number of results).
- The **logarithm (base 10) of 2** is calculated once when creating the Score object. The Score object is created immediately in the main method of **SearchEngine.java**.

10.3. Technical description

10.3.1 Adding a term frequency map

Since the term frequency of words doesn't change over time, the obvious optimization was to calculate it immediately as the web service starts up. To support this improvement, the **Website** class was changed to include a map of term frequencies - a query word as the key, and the number of times it's featured in the website being the value. Another instance variable is also created to store the total number of words without having to calculate them every time. Getter methods were added for both the map and the word counter. The implementation of the improvements in the class is as follows:

```java
private Map<String, Integer> termFrequencyMap;
private int wordsCount;

public int getTermFrequency(String word)
{
    //Returns the number of times a word is seen in a website, if it is seen at all
    if (this.termFrequencyMap.containsKey(word))
    {
        return this.termFrequencyMap.get(word);
    }

    //Returns 0 if the word is not seen in the website
    return 0;
}

public int getWordsCount()
{
    //Returns the total number of words
    return this.wordsCount;
}
```

The values of both instance variables are set in the constructor, where they are received as parameters.

The map of term frequencies replaces the previously available list of keywords for two main reasons - it uses less memory (due to keeping any word only once, instead of once for every time it's seen in the website), and because every functionality that uses the list can use the map instead.

FileHelper was modified to incorporate these changes in the Website class. Instead of creating a list of keywords, it creates a term frequencies map, and populates it while reading the text file. This is done via the following code:

```java
//At the start of the process - initialize the word counter and the frequency map
int wordsCount = 0;
Map<String, Integer> frequencies = new HashMap<String, Integer>();

//Start reading the text file
//Read a line from the text file
//The line is a keyword

//Remove capitalization in the word
String lowerCaseWord = currentLine.toLowerCase();

//The amount of times the word is seen so far is calculated - we start at one, since this
line is the first we know of
int counter = 1;
if (frequencies.containsKey(lowerCaseWord ))
{
    //The counter is updated because the map already has this word inside it
    counter = frequencies.get(lowerCaseWord) + 1;
}

//The frequencies map is updated by inserting the keyword and it's current term frequency
count. If it's not in the map yet, the counter will be 1, otherwise it will update the old
value
frequencies.put(lowerCaseWord, counter);

//The total words count is incremented in order to keep track of total words count
wordsCount++;



//When adding a website to the List of websites, add the frequencies map and words count as
new parameters, replacing the old List<Website> parameter
finalList.add(new Website(url, title, frequencies, wordsCount));

//After the website is added, the map and counter are restarted
frequencies = new HashMap<>();
wordsCount = 0;
```

Furthermore, most of the classes needed to be updated to reflect the change in the **Website** class. The most notable changes are:

- The **canAddWebsite** function in FileHelper was updated to use the term frequencies map instead of the list of websites.
- The **build** method in **InvertedIndex** was updated to use every website's **getWordsCount** when calculating the average number of words (see **Section 7**). Furthermore, the keys in the term frequencies map are now used to iterate

through all words when building the index, instead of the previously available list of words.

- The **lookup** method in **SimpleIndex** was updated to check if the term frequency of a query word is bigger than 0 (meaning the word is seen in the website), instead of using the contains method provided by the list of keywords.
- The **calculateTermFrequency** methods in both TF-IDF and Okapi BM25 were removed, in favor of using the website's **getTermFrequency** method, which doesn't involve further computation.
- All test files that used the list of keywords in any way were updated to use the frequencies map instead.

10.3.2 Removing unnecessary lookups in inverse document frequency

When calculating the ranking scores, the inverse document frequency looks up in the index to determine the number of websites in which a word is featured. Instead of doing this operation for every word and for every website, we decided to provide the scoring algorithms with the number of results. This is done via a new parameter in the **getScore** method in the **Score** interface. Furthermore, this required changes to three functions in the QuerySplit class - **evaluateSubQuery**, **evaluateFullQuery** and **calculateSubqueryRanking**.

calculateSubqueryRanking was modified to receive the number of results for each query word as a parameter, and pass the specific word's number of results to the scoring algorithm. The implementation is as follows:

```java
//Function now takes an array of numbers as a parameter, which corresponds to the amount of results for each query word in the String array parameter
private static double calculateSubqueryRanking(String[] splitByWhitespace, Website website, int[] numberOfResults, Score rankingHandler)
{
    //Initialize the score holder (same as before)
    double scoreForQuery = 0;

    //Replace old for each loop with a for loop (in order to link query words to their results count
    for (int i = 0; i < splitByWhitespace.length; i++)
    {
        //Calculate the score of the website by providing the scoring algorithm with the
query word, the website, and also the number of websites the query word is featured in
        scoreForQuery += rankingHandler.getScore(toLowerCase, website, numberOfResults[i]);

    }
}
```

Since every query word is seen in a specific number of websites, but **evaluateSubQuery** combines results for the whole subquery, it was modified to also return the number of websites each query word is featured in. This is achieved by returning a Pair instead of a List of websites. Returning a pair allows us to return both the list of websites and the number of results for each query word, without creating a special class just for this return statement. The key in the pair is the previously implemented list of websites, while the value is an array of numbers corresponding to the number of results each query word has. The change looks as follows:

```
//Modify the return type of the method
private static Pair<List<Website>, int[]> evaluateSubQuery(String[] splitBySpace, Index
index)
{
    //Create an array of numbers, the size of which is the same as the number of words in
the subquery
    int[] resultsPerWord = new int[splitBySpace.length];

    //Replace the foreach with a for loop (as we need the position of the query word)
    for (int i = 0; i < splitBySpace.length; i++)
    {
        String subquery = splitBySpace[i];

        //Execute the search as before
        List<Website> partialResults = index.lookup(subquery.toLowerCase());

        //Save the number of results in the array of numbers
        resultsPerWord[i] = partialResults.size();
    }

    //Return a Pair, instead of the previous List implementation
    return new Pair<>(resultsToReturn, resultsPerWord);
}
```

evaluateFullQuery was modified to support the changes in **evaluateSubQuery** and **calculateSubqueryRanking**, meaning that it receives the aforementioned Pair instead of a list of websites. Then, it sends both the key and the value of the Pair object to the method that calculates the scores - **calculateSubqueryRanking**. The example below shows the changes:

```
//...previous implementation

Pair<List<Website>, int[]> partialResults = evaluateSubQuery(splitBySpace, index);
List<Website> partialResultsList = partialResults.getKey();
int[] resultsCount = partialResults.getValue();

//Iterate through the results
for (Website w : partialResultsList)
{
    //Calculate the total score of the subquery by using both the key and the value in the
Pair object
    double scoreForQuery = calculateSubqueryRanking(splitBySpace, w, resultsCount,
rankingHandler);

    //Continue as before
}
```

10.3.3 Calculating the logarithm (base 10) of 2

Since Java doesn't provide a convenient way of calculating a logarithm (base 2), we use one of the theorems in math - rebasing logarithms. The only logarithm function in Java's Math class is in base 10, so we divide the logarithm (base 10) of the inverse document frequency by the logarithm (base 10) of 2. The equation looks like this:

$$log_2\left(\frac{|d|}{n}\right) = \frac{log_{10}\left(\frac{|d|}{n}\right)}{log_{10}2}$$

To summarize, we replace the left part of the equation with the right part. Since the denominator is always the same, instead of calculating it every single time, we calculate it only once and keep the result as an instance variable. We decided to do that in the constructor, so the logarithm is calculated immediately upon creating the Score object. The following code shows the implementation of the optimization and is seen in both the **ScoreTFIDF** and the **ScoreBM25** classes:

```java
private final double logOfTwo;

//Constructor, the same logic is seen in ScoreTFIDF.java
public ScoreBM25(Index index)
{
    this.logOfTwo = Math.log10(2);
    //Continue with constructor
}

public void calculateInverseDocumentFrequency(String keyword, int numberOfResults)
{
    //Previous implementation up until final calculation

    //Replace the previous denominator with the instance variable
    this.inverseDocumentFrequency =
            Math.log10((double)numberOfWebsites / (double)numberOfResults) / this.logOfTwo;
}
```

10.4. Testing considerations

All available tests in the tests folder were ran to ensure no bugs were created in the process of optimizing.

Results from searching and ranking scores were manually checked by running the web service and searching for specific queries for which we knew the results.

10.5. Benchmarking/Reflection

A benchmark was carried out to test the speed of the ranking algorithms. It is located in **tests/BenchmarkScore.java**. The benchmark, similar to previous ones, searches through the index for 4 specific keywords - *america*, *denmark*, *of*, *university*. Then it calculates the score of each website in the resulting list. The test is done 1000 times for each keyword in order to avoid anomalies and check the average amount of time it takes to score a website. It should be noted that this benchmark only uses the HashMap implementation of InvertedIndex, since it's the one SearchEngine will use.

The same benchmark was executed before the implementation of this section, to compare results with the previous implementation of the two ranking algorithms (see **Sections 6 and 7**). Figure 4 shows the results of our optimizations:

Ranking algorithm	**File used**	**Average rank time (nanoseconds)**
TF-IDF (before optimizations)	enwiki-medium.txt	2220260
Okapi BM-25 (before optimizations)	enwiki-medium.txt	2132765
TF-IDF (after optimizations)	enwiki-medium.txt	530792
Okapi BM-25 (after optimizations)	enwiki-medium.txt	551905

Figure 4 - Results from the ranking benchmark before and after optimizations

Furthermore, the previously developed benchmark for the **QuerySplit** class was ran again to ensure that the time saved from optimizations is carried over when calculating scores of complex queries. To simplify the table, only results for the Hash Index are shown because it is the one to be used in the final version of the Search Engine. The results are shown in figure 5:

Ranking algorithm	**File used**	**Average search time (nanoseconds)**
TF-IDF (before optimizations)	enwiki-medium.txt	3068458
Okapi BM-25 (before optimizations)	enwiki-medium.txt	3035163
TF-IDF (after optimizations)	enwiki-medium.txt	2041916
Okapi BM-25 (after optimizations)	enwiki-medium.txt	2082516

Figure 5 - Results from the QuerySplit benchmark before and after optimizations

To summarize, we improved the execution time of the ranking algorithms by a factor of ~4. The speed of the whole searching logic was improved by ~150%.