Project: Search Engine

Group: ip18groupR

IT UNIVERSITY OF COPENHAGEN

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

Contents

Chapter 1

Introduction

1.1 Introduction

The goal of htis project is to implement a large piece of software and develop web-based search engine. Several software development tools and tehniques have been used: version control(Git), testing (JUnit), debugging, documentation (JAvadoc), benchmarking, build tools (Gradle), and code review. The fallowing chapters describe the project in detail. Project is broken down into three main parts, Task 1: Fester Queries using an Inverted Index; Task 2: Refined Queries; Task 3: Ranking Algorithms.
Result
This projecct result in...

Chapter 2

Chapter 2: Faster Queries using an Inverted Index

2.1 Introduction

In this section we are evaluating three different approaches for storing the search indexes, namely array lists, hash map and tree map.
Inverted hash map and tree map...
Ran tests to compare the results, which

2.2 Set Up

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 doesn't look to be that big of a change, doing so means that the two field variables are no longer needed, which means less has to be juggled when reading and making further changes to the code.

2.3 Indexes and Data Structures

2.3.1 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 eash site than have their own ArrayList<String> containing all the words on the sites.

2.3.2 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 it's 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 threre 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]

2.3.3 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 assumpton that there are not too many colisions. This is becaseu this Map implementation acts as a busket 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 it's 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]

2.4 Benchmark

In order to choose one of the implementations, namely ArrayList, TreeMap or HashMap, for the Search Engine, the benchamrk 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 bechmark test was carried out using 20 words (random nouns, verbs, adjectives and conjunctions), which were looked-up using the three different indexes implementations and in three difrernt size databases: enwiki-tiny, enwiki-small, enwiki-medium. JMH provides information about an avrage Score, measured in nanoseconds per operation, see results in table 2.1
During the benchmark it was assured that the test environment is as similar as possible among the different tirals, meaning that all tests were performed on the same machine and no other applications running on the background.

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, respectively. In order to describe the reulsts let the number of websites be m and words be n. The difference in performance can be explainde as fallows:

Table 2.1: Benchmark Score in ns/op for SimpleIndext, InvertedIndexHashMap and InvertedIndexTreeMap using enwiki-tiny, enwiki-small and enwiki-medium data sets on average

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

When the SimpeIndex 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 $O(m \cdot n)$. Two other methods provides 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 since the Java8 of *O(log(n))* time. Worst-case performance occurs, when hash function is not implemented correctly, values are are distributed poorly in buckets and there is high hash collision.

2.4.1 Conclusion

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, becasue in this case the order of data is not important versus the performance looking up the websites corresponding tho 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 HashMap's worst-perofmance is be *log(n)* time. The given data sets are fixed, therefore the costly resizing and rehashing is not going to occure 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 ovet the TreeMap implementation for this Search Engine project.

2.5 Testing

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.

2.5.1 JUnit tests

2.5.1.1 FileHelper Class

White-box tests were developed around the branching statements in the updated method, and a coverage table was produced.

Table 2.2: 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

From the coverage table an expectancy table was produced.

Table 2.3: 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

where data/test-file1.txt is an empty file, and the rest contained the following data:

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

As you can see from the Actual Output column of **??**, 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.

2.5.1.2 InvertedIndexHashMap Class

2.5.1.3 InvertedIndexTreeMap Class

Chapter 3

Chapter 3: Refines Queries

3.1 Section 3.1

Text

3.2 Section 3.2

Text

3.2.1 Subsection 3.2.1

Text

Chapter 4

Chapter 4: Ranking Algorithms

4.1 Introduction

Typically, results returned from search engines are ranked in some way so as to return the more relevant search results first. For this task, the following ranking algorithms were introduced:

- Term frequency
- Inverse document - term frequency
- Okapi BM25

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

4.2 Analysis

A Score interface was created with only one method: getScore(String, Website, Index), and each of the following classes implemented this interface.

4.2.1 IFScore class

4.2.2 IFIDFScore class

4.2.3 OkapiBM25 class

4.2.4 Testing

Text

Chapter 5

Chapter 5 Extensions

5.1 Section 5.1

Text

5.2 Section 5.2

Text

5.2.1 Subsection 5.2.1

Text

Chapter 6

Chapter 6 Conclusion

6.1 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`

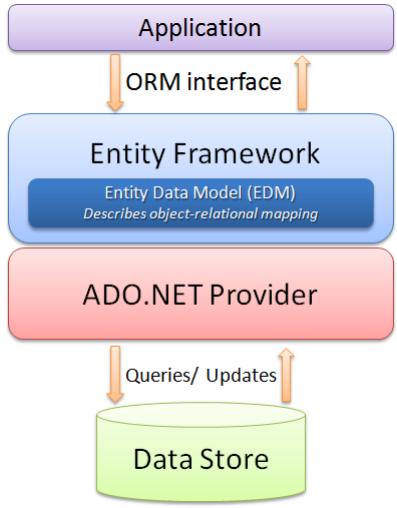

Figure A.1: Microsoft Entity Framework

Appendix B

Test tabel reference

This appendix is a test of creating and referencing a table in latex. In table ?? a example from Peter can be seen. can be seen.

Table B.1: Oversigt over testdeltagerne

Deltager	Navn	Stilling	Rolle
1	Ole Nørrekær Mortensen	Projektleder	Kundeadministrator
2	Allan Booker	Driftsleder	Inspektør
3	Ronni Bing Simonsen	Ingeniør	Kunde

Table B.2: Test af tabel

Columm1	Columm2
Celle 1	Celle 2
Celle 3	Celle 4

Tabellen har nummer B.2.

En lidt mere avanceret tabel:

I tabel B.3 kan du se hvordan teksten er justeret: l=left, c=centreret og r=right.

Table B.3: Test af tabel2

Celle 1..... Celle 4	Celle 2..... Celle 5	Celle 3..... Celle 6

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.