Information Retrieval - Assignment 1

DEPARTAMENTO DE ELETRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

MESTRADO INTEGRADO EM ENG. DE COMPUTADORES E TELEMÁTICA

ANO 2018/2019

Beatriz Coronha 92210

Pedro Santos 76532

1. Introduction

This report will explain the development of a document indexer that consists of a corpus reader, a document processor, tokenizers and indexer. This report will explain some of the reasoning in the development and some of the architecture used in the solution. Also, it will be given some instructions to explain how to run the code and in the end the results of a series of tests to measure efficiency of this project will be presented.

2. Class Diagram

The class diagram for the project is presented below (Figure 1).

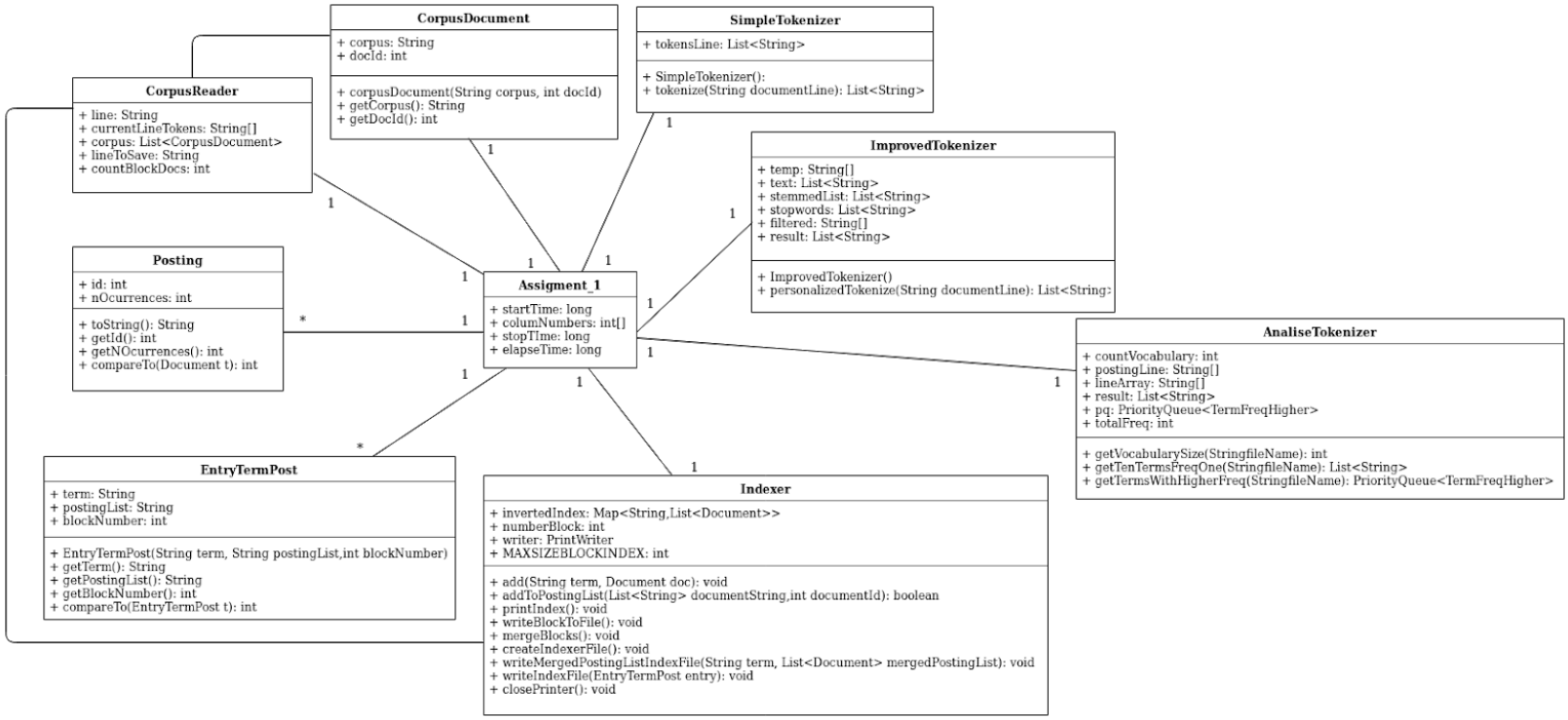

Figure 1: Class Diagram

3. Description of Class/Methods

This section will explain the different classes, the methods in them and how they're connected. The classes were developed to be as efficient as possible.

a. Assigment_1.java

This is the main class, this is where the CorpusReader and the indexer are instantiated. The main class has some variables to help initialising the objects. One of the variables is *filename*. This variable is used to set the document that it is suppose to read and them create an index. The variable *columnNumbers* is an array where it is placed the number of the columns that it is supposed to read from the file. Also, there are some variables to take some measurements about the processing time. In the main class is where the functions from the class of each tokenizer will be called before adding the tokens to the indexer. The other classes will be called and, with the "currentTimeMillis()" function the indexing time is calculated. Basically, the main class will set the flow of the program.

b. CorpusDocument.java

This class has the purpose of save each document of the file. The attributes of this class are the content of it and the id of the doc and has two functions to get this fields. When each document is being read it is instantiated an object of this class where the content of the columns that matter are saved in the attribute *corpus* and it is setted the doc id. Then, it is added in a arraylist that saves elements of this class.

c. CorpusReader.java

As the name says, this class starts by reading the file (the corpus) that will be used. This class has only one function *corpusReader* that will iterate each line of the document and it will save the content like was described when it was explained the *CorpusDocument* class. *It was decided that the CorpusReader class was necessary to be instantiate to save the "state" of it when it is reading a document, because after each 10000 docs they will be sent to the indexer until there are no more documents. It was decided to do this to not use a lot of memory and ensure that we do not run out of memory.*

d. Posting.java

Posting Class will represent each posting of the posting list of each term. This class is used to save in a arraylist the posting of the terms of the index and also it is used when the merge of the index block is being made and there is terms that are equal in different files and the posting list must be merged. When the term is unique in all blocks, there is no problem and the posting list can be processed like a string, however if we want to have the posting list order by ids and the term appears in different blocks, it is necessary to process the posting list, so this class is used to save each post of the posting list and the attribute doc id is a integer, so it is easier to compare. To print for the file direct the posting list, this class overrides the method *toString()* and to be able to order the elements and use a priority queue for that this class also overrides the method *compareTo()*.

e. EntryTermPost.java

This class has the purpose of saving the term and the posting list (“docId” and frequency of the term). The objects of this class are initialized when it is being read each indexer block, basically each line of the block will be used to initialize this objects. To merge the blocks it is used a queue and to know from which file was read that term, this class has an attribute to save that information. This will be explained better in indexer class.

f. Indexer.java

This class is the class that creates the indexer and also merges the several indexer blocks created. The first method called from this class is *addToPostingList().* This method receives the tokens and the corresponding doc id. This method will count the number of times that each term appears in the doc. Then it will remove the repeated terms and then it will add to the indexer each term and the posting. Before adding each term to the indexer, it will verify the size of the block, each block has size of 30000 terms, so if the size of the index is equal to 30000 it will write the block. Also, after that, it is verified if the memory used by the pc is bigger than 97% in case that some pc’s don’t handle blocks so big or run out of memory for some unusual problem. This value was tested to see if it would work and it was a success, however it is not the normal workflow. The function *mergeBlocks()* is the method that will merge all the created blocks and the final index. In this function the first thing that is done is initialize the file readers and buffers for each block. Then, it has a priority queue that has objects of the type EntryTermPost. The block indexers are sorted alphabetically, so it is added from each block one term and its posting list. After this, the function enters in a cycle until the queue is empty. To handle the repeated terms in several blocks, the elements are removed from the queue and added to a list until the head of this queue is different from the first element removed in the beginning of the correspondent cycle. Then, if the list with the objects of EntryTermPost Class removed from the queue is bigger than one it means that it is necessary to merge the posting lists. So, the posting lists are merged by adding all the posts of each term in a queue that will retrieve in sequential order the posting lists by the doc id. So, the final indexer will have the posting lists in ascending order by doc id. After this, the term and the merged posting list is written in the final indexer. If the list with the objects of EntryTermPost Class removed from the queue has only size one it means that that term is unique to that block, so it is possible to write that directly in the final indexer. The last step is to add again to the queue the same number of terms that were removed in the

same cycle. So, it is necessary to iterate the list that has the removed objects from the queue to know from what file they were removed and read from that file another line and add that line to the queue. After this process we have an indexer without repeat terms, ordered by alphabetical order and with merged posting lists.

g. SimpleTokenizer.java

This is the Simple Tokenizer class, it splits on whitespace, lowercases tokens, removes all non-alphabetic characters and removes terms with less than three characters.

In this class there is only one function and this function is static, so it is not necessary to instantiate the class to use the function tokenize. To process the string all the white spaces bigger than one are transformed in only one and then it is made a split using as delimiter the white space. The last step consists in processing each word by removing all the nonalphabetic characters and also filtering the words that are smaller than three characters.

h. ImprovedTokenizer.java

This class has some addictions and differences comparing to the simple tokenizer. The first thing that is done is to replace the characters like parenthesis, comma by white space. This is done to avoid lost some information, for example it is quite normal to neglect by mistake the space after commas or dots, so by doing this we preserve the information. Then, all the spaces are transformed in only one white space. The next step is to remove the stop words and then perform the stemming. To be able to use it,it has to be declared as english, then it's possible to make the operations. Here and also in the SimpleTokenizer it was used always functional programming by using streams and calling methods like filter and map. In the end all the special characters are removed from each word.

i. TermFreqHigher.java

This class was created to solve the exercise 2.3 c). This class has two attributes one is the term and the other one is the total frequency of the term in the indexer. This class implements Comparable to be able to use the object in a priority queue where the head of the queue is the element with a lower frequency.

j. AnaliseTokenizer.java

This class deals with three problems posed for this assignment:

1. What is the vocabulary size using each tokenizer?

2. For each tokenizer, list the ten first terms (in alphabetical order) that appear in only one document (document frequency = 1).
3. For each tokenizer, list the ten terms with higher document frequency.

For the problem number 1 was created a function called “getVocabularySize” that will get the file that has the indexer. To count the number of terms, basically it is counted the number of lines that the document has.

For the second problem was created the function “getTenTermsFreqOne” where the file is read line by line, if the length of the line is equal to 2 that means that the term associated to that line only appears in one document, because the first item in the line is the term that is being referenced and then the list of document it appears, and if the frequency is equal to one that means that the term only appears once in the document and in only one document, so it’s possible to add it to the result list. When that list size is equal to ten, this process ends, because it is only necessary the first ten terms.

The last problem is the most complex one, for that it was created a function called “getTermsWithHigherFreq”, it was also created a priority queue that keeps objects with the term and the sum of times it appears in all documents. Since only ten elements are needed for this, first the queue is filled with the first 10 terms/freqs. Then, it will enter in another cycle to iterate the rest of the indexer and comparing the sum of the frequency of the element that it is in the head of the queue with the sum of the frequency with the element that it is reading and if the element that is reading has a bigger frequency it will remove the head of the queue and add the new element to the queue. In the end, the queue will have the 10 elements with higher frequency.

4. Data Flow

The data flow for this project is presented below (Figure 2). In it is possible to see the most important steps of the code, first there is the “Initialization”, where the “CorpusReader” and the “Indexer” are initialized. Then, the “Processing” happens, the CorpusReader reads 10000 documents and then the tokenize will process that documents and the tokens will be consumed by the indexer. Then, another 10000 documents will be processed and this process happens until there are no more documents. After this part, it is necessary to merge all the different blocks and create the final indexer.

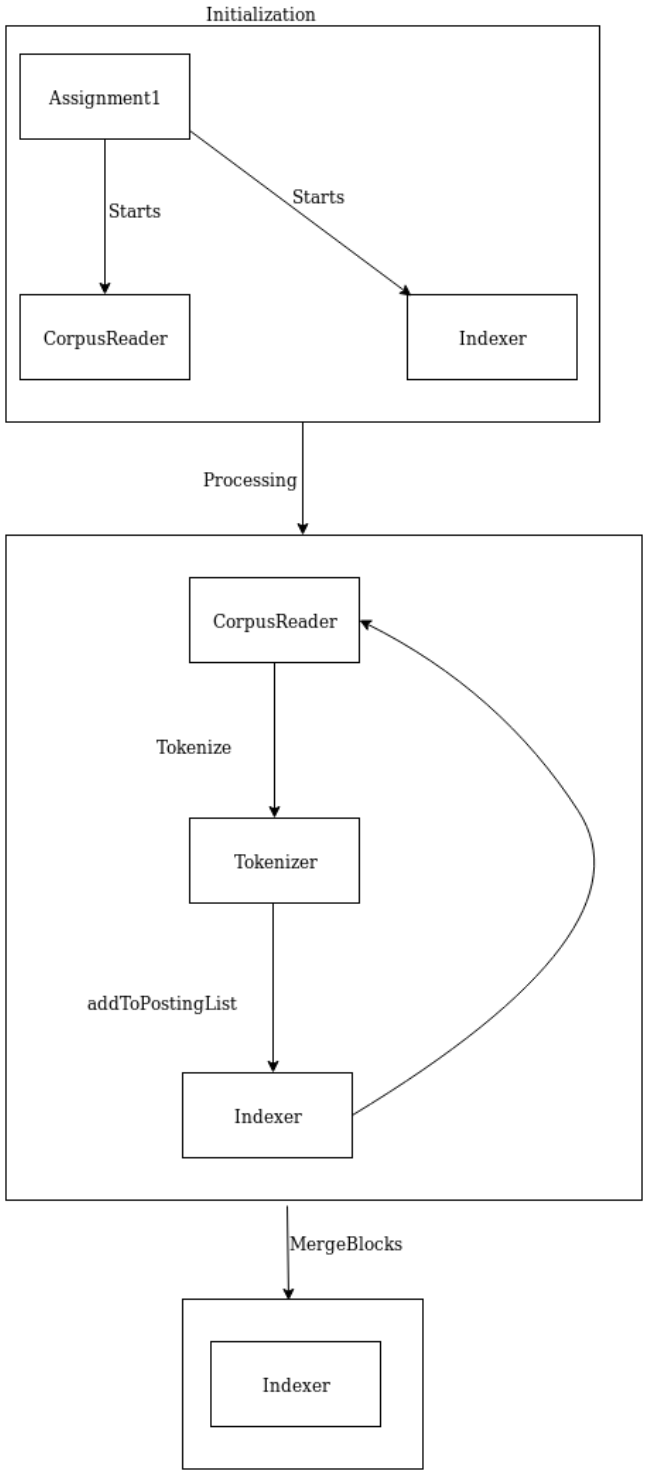

Figure 2: Data Flow

5. Results/Tests

a. Exercise 2

2)

a)

	Vocabulary Size
Simple Tokenizer 400 mb	268196
Improved Tokenizer 400 mb	219008
Simple Tokenizer 4 gb	1107029
Improved Tokenizer 4 gb	662952

b)

Simple Tokenizer 400 mb-

aaaaaaaaaaa, aaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaalll, aaaaaaaaaand, aaaaaaaaabr, aaaaaaaanything, aaaaaaamazing, aaaaaahhhbr

Improved Tokenizer 400 mb-

0000023, 000015, 000090, 0000h, 0000hour, 0000xxx, 00013227513, 00036, 000f, 000hz

Simple Tokenizer 4 gb-

aaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaa

Improved Tokenizer 4 gb-

0000000000000, 00000000000000000001, 00000000000000001, 0000000000001, 000000000001, 00000000001, 0000000000139, 00000000001mm, 000000099, 00000010

c)

Simple Tokenizer 400 mb-

Term: “watch” Frequence: 2420383,
Term: “the” Frequence: 2599421,
Term: “and” Frequence: 1465716,
Term: “this” Frequence: 866529,
Term: ”for” Frequence: 799108,
Term: “with” Frequence: 532678,
Term: “but” Frequence: 451986,
Term: “mens” Frequence: 466673,
Term: ”that” Frequence: 436292,
Term: “not” Frequence: 405992,

Improved Tokenizer 400 mb-

Term: “watch“ Frequence: 2666917
Term: “look“ Frequence: 427488,
Term: “br” Frequence: 434041,
Term: “band” Frequence: 350130,
Term: “men” Frequence: 484560,
Term: “time” Frequence: 334731,
Term: “great” Frequence: 372679,
Term: “black” Frequence: 269421,
Term: “love” Frequence: 310090,
Term: “good” Frequence: 265350,

Simple Tokenizer 4 gb-

Term: this Frequence: 6846082
Term: for Frequence: 12168539,
Term: the Frequence: 24961078,
Term: and Frequence: 14470381,
Term: iphone Frequence: 5165625,
Term: with Frequence: 6477355,

Term: case Frequence: 9593544,
Term: not Frequence: 4394610,
Term: phone Frequence: 5768089,
Term: that Frequence: 4118421,

Improved Tokenizer 4 gb-

Term: “work“ Frequence: 3240852
Term: “great” Frequence: 3702912,
Term: “case” Frequence: 10342657,
Term: “iphon” Frequence: 5207846,
Term: “phone” Frequence: 6356955,
Term: “screen” Frequence: 3118240,
Term: “galaxi” Frequence: 3072341,
Term: “br” Frequence: 4399503,
Term: “samsung” Frequence: 2810551,
Term: “use” Frequence: 2797704,

b. Tests

	Total indexing time	**Total index size on disk**
400 mb File Simple Tokenizer	107.8 segundos	358,9 MB
4 gb File Simple Tokenizer	1201.5 segundos	4000 MB
400 mb File Improved Tokenizer	140,1 segundos	267,9 MB
4 gb File Improved Tokenizer	1525.52 segundos	3,1 GB

The tests were running in a laptop with a processor intel i7-4710hq and 8 gb of ram. Despite being an i7, it is necessary to have in consideration that the laptop has around

4 years old and also the tests were running with netbeans, chrome and some other applications opened.

6. How to Run

In this section, it will be presented instructions to run the code developed. There will not be any instruction on how to download the stemmer or on how to build the jar file that will be mentioned in the next subsection since the project already possess it, just as the project already has the stop word list, so this won't be mentioned either.

However, there are things the user needs to do to be able to run the code, one of these things is to change two directories. One of these is the directory of the file that will be read. In the class "Assigment_1.java" change the "filename" path to the one that the file wanted is.

a. Requirements

The most important requirement, and the only one the user needs to worry about, is the integration of the porter stemmer. As mentioned before, this subsection will only focus on the steps that matters for the user to run the code, not on what the developers needed to install and add to it.

The first thing to do to be able to run this code is to install the porter stemmer, to do that is necessary to open the directory of the code in the terminal and run the following: mvn install:install-file -Dfile=lib/libstemmer.jar -DgroupId=org.tartarus -DartifactId=snowball -Dversion=1.0 -Dpackaging=jar. By doing that, the file "libstemmer.jar" is being installed with maven. The next step is to do a clean install, so that the target will be cleaned before the installation. To do that, run this line in the terminal: mvn clean install in the "assignment1" directory.

7. Conclusion

The goal of creating and simple document indexer was accomplished, however there are some things that could be improved with a little more time to be an indexer even more scalable and faster. One of the things would be using threads and the other thing would be not reading a entire line when reading files, because with more documents it is possible to that line be bigger than the available memory.