Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Assignment 1

Information Retrieval

October 22, 2018

Professor Sérgio Matos

Diogo Ferreira ***76504***

Luís Leira ***76514***

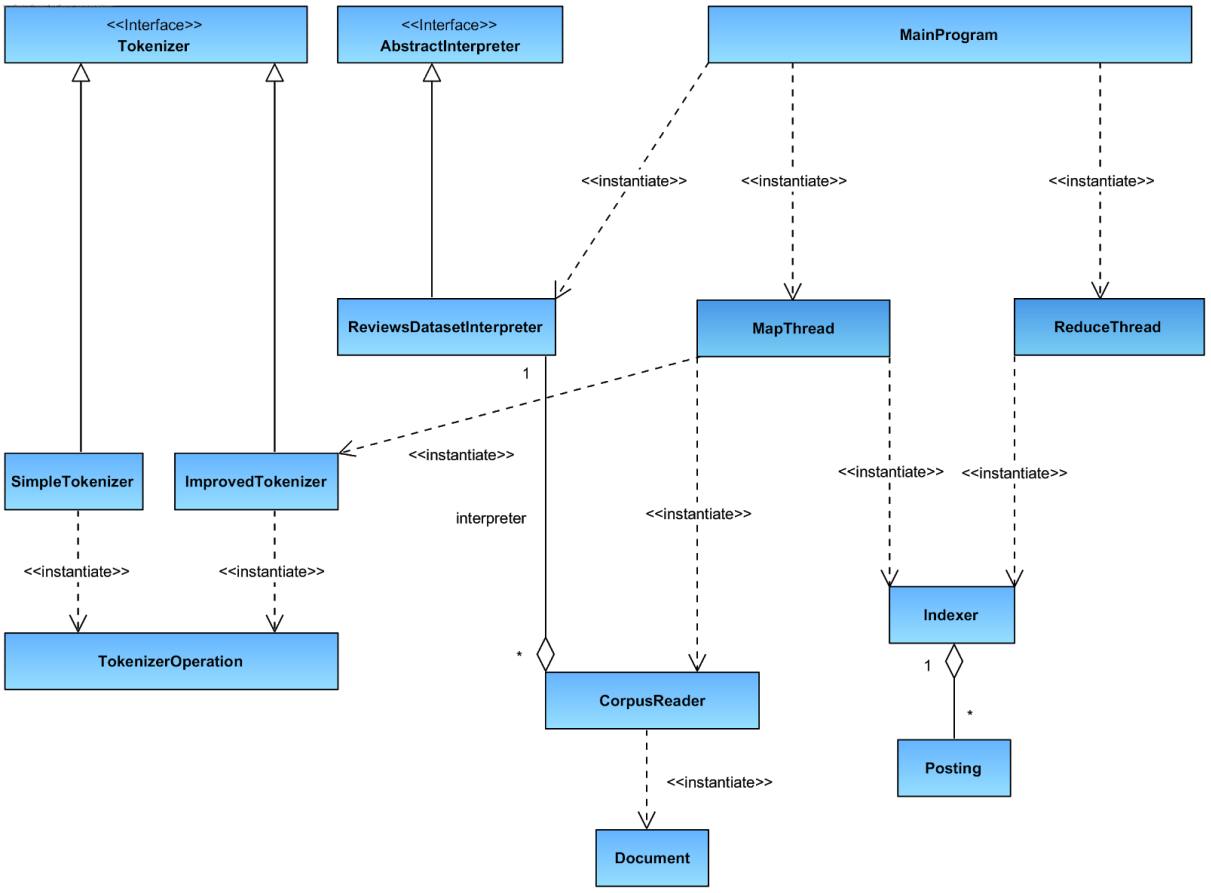
Context

This report explains the work done in the first assignment of the Information Retrieval course. The assignment aim is to create a document indexer, in which the main components consist in a corpus reader, a document processor, a tokenizer and an indexer. These components will be thoroughly explained in this report, with the help of the architecture diagram and the data flow between them. It also contains details about execution instructions, external libraries used, efficiency measurements and final conclusions.

The corpus for this project is the Amazon Customer Reviews Dataset available at https://s3.amazonaws.com/amazon-reviews-pdf/readme.html.

Architecture

On this section, the software architecture is presented in the form of a class diagram. Each class function and main methods are described below.



MainProgram

This class contains the main flow of execution of the project. It has the responsibility of creating the workers for the map and the reduce phase (explained in the next section, Data Flow). According to the maximum number of parallel threads and maximum number of documents per indexer, the number of threads and documents per indexer will be adjusted. As such, before running on a new configuration, it may be needed to adjust the settings adequately to prevent memory errors or lower the execution time.

After launching workers to execute the map phase and the reduce phase, the exported index will be read from the files and imported into the application. The index will only contain the terms and the number of documents in which they appear, to be able to fit in memory.

Finally, three operations will be done with the indexer:

- The vocabulary size will be calculated;

- The first ten terms in only one document will be calculated with the method firstNTermsInMDocuments(Map<String, Integer> index, int n, int m);
- The ten terms with higher frequency in the indexer will be calculated with the method nTermsHigherDocumentFrequency(Map<String, Integer> index, int n).

MapThread

Each MapThread is responsible for reading part of the input file, specified by the start and end line. It starts by creating a corpus reader from a file and iterating through the allowed documents inside its range. For each document, a tokenizer will be applied and the resulting tokens will be added to the indexer. After all terms have been indexed, the indexer will export the index to multiple files, each file containing the index with the terms that start with a different alphabetical letter. The index with all other terms that do not start with an alphabetical letter will also be saved into a different file.

ReduceThread

The ReduceThreads are launched after all MapThreads are done. Each thread is responsible for creating a unified index for the terms that start with a specific alphabetical letter (one thread is also responsible for creating the index with terms that do not start with an alphabetical letter). To do so, it is necessary to read all indexes with terms that start with an alphabetical letter, and merge them all into one index. Because the indexes are sorted, it can be used a strategy to merge indexes without loading the entire indexes into memory. It can be specified a number of index entries to read from each file, and then sort them and append them into a temporary file. With this strategy, it is possible to construct large indexes without running out of memory. The long explanation of this algorithm is described in the Indexer subsection.

CorpusReader

The CorpusReader class deals with the opening, reading and pre-processing of the documents. It receives in the construction of a new object the path to the file where the corpus is, an interpreter to interpret each line as a document, a start line and an end line. The start line and the end line indicate a region from where to fetch documents from the file. The method *getNextDocument()* retrieves the next document object from the corpus in each call, and if there is no next document or the corpus reader has reached the end line, it returns null.

Document

The Document class represents a retrieved document from the corpus. It contains an ID (different for each document), the product title, the review headline and the review body.

AbstractInterpreter

The AbstractInterpreter interface contains the method *interpret(string doc)* and it is used for other classes that want to implement a method for retrieving documents from strings.

ReviewsDatasetInterpreter

The ReviewDatasetIntepreter implements the AbstractInterpreter and it is the implementation of an interpreter for the reviews dataset used in this work.

Tokenizer

The Tokenizer interface specifies the method *tokenize(String words)* that returns a list of tokens given the input. The input is a list of words to be tokenized.

SimpleTokenizer

The SimpleTokenizer class implements the Tokenizer interface to create a tokenizer. The *tokenize(String words)* method does four operations: splits the words on whitespaces, converts the tokens to lower case, removes all non-alphabetic characters and removes terms with less than three characters.

ImprovedTokenizer

The ImprovedTokenizer class is another implementation of the Tokenizer interface. The operations available are the following ones:

- splits the words on whitespaces;
- converts the tokens to lower case;
- removes invisible characters;
- removes all tokens that are e-mails;
- removes all tokens that start by a number and have a punctuation next to the number (this dataset has many tokens with that format);
- removes meaningless characters;
- removes tokens that are hyperlinks;
- removes full stops at the beginning or end of token;
- removes all “^” characters without numbers on both sides of it (can indicate an exponential number);
- removes hyphenation at the beginning and end of tokens;
- removes the stop words according to a list of stop words;

- stems the tokens with the Porter Stemmer in English language.

  The operations are carefully described in the TokenizerOperations class subsection.

TokenizerOperations

The TokenizerOperations class has the methods to apply transformations to words and turn them into tokens. When the class is loaded, the stop words are loaded from file, using the method loadStopWords(). When the class is instantiated, the constructor receives a string of words that transforms into a list of tokens, splitting the words on whitespace. The process results on a stream of tokens, that is stored inside the object. The stream of tokens can be retrieved using the method getTokens(). The operations can be piped into each other, because the return type is the TokenizerOperation class. The methods that can be applied to tokenizer the words are:

- removeNonAlphabeticCharacters() – Removes all the non-alphabetic characters from the tokens;
- removeTermsWithLessThanNCharacters(int n) – Removes all tokens with less than n characters;
- toLowerCase() – Converts all token characters to lower case;
- stem() – Using the Porter Stemmer, applies the stem operation to all tokens;
- removeStopWords() – According to the loaded list of stop words, removes all the tokens that are present in that list;
- removesNumbersFollowedByPunctuation() – Removes all the tokens that start by a number and have punctuation next to that number (common problem in one analyzed corpus);
- removeMeaninglessCharacters() – Removes list of pre-defined characters from tokens. The list of characters is composed by special characters, like ‘$’, ‘!’, ‘[’ or ‘?’;
- removeSoloHyphenation() – Removes the hyphens from tokens where they are in the beginning or end of the token. Remove also all consecutive hyphens;
- removeHyperlinks() – Removes all tokens that are hyperlinks;
- removeFullStops() – Removes full stops at the beginning or end of token;
- removeExpoentSymbolWithoutNumbers() – Removes all ‘^’ characters that do not have a number by each side. When there is a number on both sides, the character is kept, because it may symbolize an exponential operation;
- removeEmails() – Removes all tokens that are e-mails;
- removeInvisibleCharacters() - Removes invisible characters that can be fetched when reading from a file.

Indexer

The Indexer class has the goal of creating an inverted índex composed by the terms and its postings. The indexer will consist on a TreeMap, with the terms being the keys and a TreeSet of Postings as the values. The TreeMap and TreeSet both have complexity O(log2(n)) for insertion and search, and keep terms and postings ordered.

The addToIndexer(List<String> tokens, int docID) method receives as argument a list of tokens and the id of the document where the tokens were retrieved, and it will add the terms to the indexer, using the method addToIndexer(String term, int docID, int freq) to each term.

The method addToIndexer(String term, int docID, int freq) starts by checking if the term is already on the indexer. If it is not found, it creates a new entry in the index, and associates with it a new TreeSet, with one posting. It the term is found, retrieves the list of postings of the term and searches for a posting with the id of the document. If the posting is found, the frequency of the term in the document is incremented. If the posting is not found, a new posting is inserted into the list with a new document id and the frequency specified as argument.

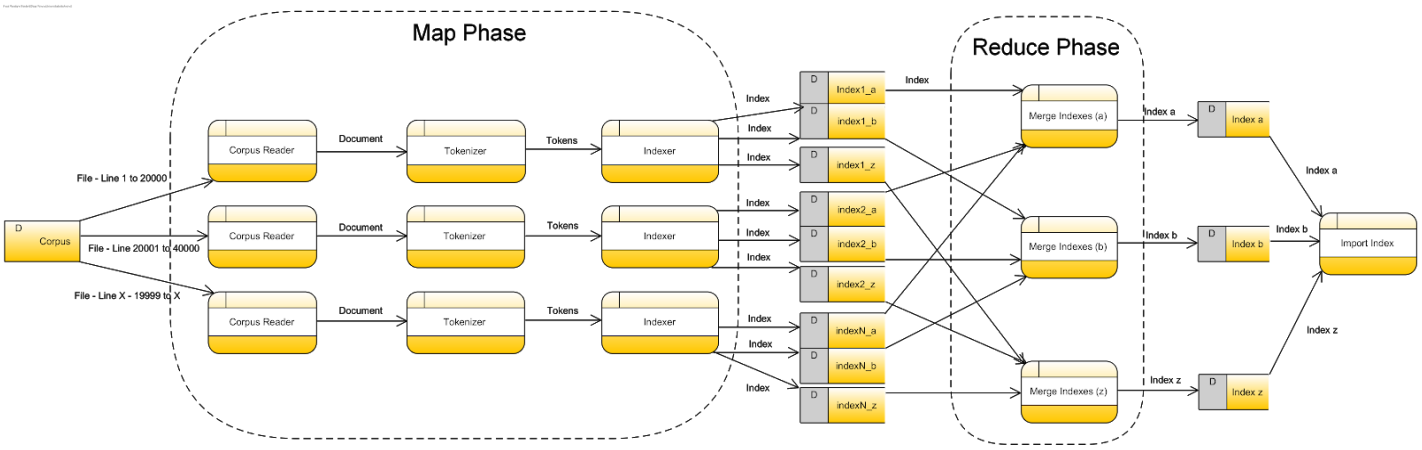
The exportToFileByLetter(String prefix) method exports the created index to files. The index will be separated into multiple sub-indexes. 27 files can be generated, each one with the content of each sub-index. Each sub-index is identified by the first letter of its terms (26 indexes for a to z, and one index for other characters).

The mergeMultipleIndexes(String path, String postfix, String outputFilePath, int maxDocumentsInIndex) method merges the indexes created in the exportToFileByLetter(String prefix) by the first character of the terms in one index. To do so, receives as argument a postfix that all index files follow, to list them. Then, the reading of the indexes in those files is made in sequence. For all files, it will be read a limited number of lines (maxDocumentsInIndex), to not cause an out of memory error. Because the indexes are already sorted, the read index entries can be merged and sorted easily. It is kept track of the last term read in each file. When a number of lines is read from all files, it is searched for the last read term in all files with a lower value. It can be proved that from the beginning of the index until that term all entries are sorted, and can be appended into a file. The entries written into the file are removed from the index in memory, and the files are read again. This process continues until all files are fully read. At the end, a complete index is outputted to a file, resulting on the merging of multiple indexes.

Posting

The Posting class contains a document ID and the frequency of that document in a specific term. When a term is found more than once in a document, it is possible to increment the frequency of the term in the document, incrementing the frequency in the respective posting, using the method incrementFrequency(), or incrementFrequency(int n) if we want to increment it n times. This class also implements the Comparable interface, to compare two postings by the document ID. This feature is useful to automatically sort the Posting instances in a sorted list, in the indexer.

Data Flow



The data flow diagram of the project can be seen above. The map phase consists in a number of threads processing the documents in parallel. For simplicity, the number of threads depicted in the diagram is three. In each thread, the corpus is loaded by the Corpus Reader, and the Corpus Reader is responsible for reading part of the file. The corpus reader will produce documents, that are sent to the tokenizers. After the tokens for a document are produced, they will be sent to the indexer, to be indexed according to the document. Each index will be written on a file. Each thread will produce 27 indexes (one for each alphabetical letter, one for others characters).

The reduce phase will also be done with multi-threading. 27 threads will be launched: one for each alphabetical letter and one for other characters. Each thread will perform a merge of the indexes that start with a different alphabetical letter. To prevent from out of memory errors, each thread will read a number from lines from each file, merge the indexes, sort them and append them into a file. The process is repeated until all files have been fully read. In the end, each thread will produce a file with the index with terms that start with each alphabetical letter and one index for terms that start with other characters.

Finally, to answer the questions proposed in the assignment, the indexes will be read from memory and merge into one index. That index will not contain all document ID's from each term, only the number of documents of a term. In that way, the index can be merged into memory without the need for storage in intermediate steps.

Another variation of the data flow was also tested. In that variation, the threads that implement the map phase would not read directly from the file, instead they would call a method that would read sequentially the documents in the corpus. However, the execution time of the software was not as good as the data flow described above. A deeper explanation and discussion on the differences and the results is done in the conclusion section.

External libraries

The different tokenizer implementations have two methods that use an external library whose documentation can be seen in http://snowball.tartarus.org/. The stem method is an integration of the Porter stemmer according to the library, whose information is available at http://snowball.tartarus.org/download.html. The removeStopWords method is a stopword filter that uses a default list according to the same library, whose information is available at http://snowball.tartarus.org/algorithms/english/stop.txt.

Execution instructions

There are two main ways to compile and execute the code: by the command line or with an IDE. Both are explained below.

The parameters available to change (*MAX_PARALLEL_THREADS* and *MAX_DOCUMENTS_PER_INDEXER*, explained in the efficiency measurements) are in the MainProgram. The file path of the corpus is also specified in the main program. To change the tokenizer, change the tokenizer variable in the MapPhase class.

Command line

To use this method, Maven must be installed.

Go to the project root directory.

The external library must be integrated by doing the download of the libstemmer (available at http://snowball.tartarus.org/download.html) and following the README instructions.

Then, build the project by executing the following command:

```
mvn package
```

After building the project, you may test the compiled and packaged JAR with the *MainProgram* from the solution by executing the following command:

```
java -cp target/IRProject-1.0-SNAPSHOT.JAR assignment1.MainProgram
```

IDE

This project can also be imported in any IDE that supports Maven like NetBeans or Eclipse.

After importing the project, do “Build” and then go to Run > Configurations > Customize > Run and select the *MainProgram* from the solution. No arguments are needed to run the application.

Finally, click on “Run project”.

Efficiency measurements

On this section it will be described the tests done to measure the execution time of the software and the total index size on disk. These efficiency measurements were taken in a 3 year old laptop with an *Intel i7-4720HQ (4 cores, 8 threads) 2.60GHz*, *16GB RAM, 1TB HDD disk (5400rpm)* running *Kubuntu 18.04*.

The execution times measured are from the beginning until the end of the program. Each value on the following tables is the mean of 10 successive tests.

From the program, there are two variables to take into account:

- *MAX_PARALLEL_THREADS* - the maximum number of parallel threads that can be running simultaneously. This variable was tested with different values to find the best execution time.
- *MAX_DOCUMENTS_PER_INDEXER* - Maximum number of documents per index (to not exceed memory). This value was tested with different values, and it is expressed as 100000/MAX_PARALLEL_THREADS (depends on the previous variable). 100000 is a fixed value and it was chosen because it provided the best measurements among other tested values (10000, 100000 and 1000000).

There are two Tokenizer implementations (*SimpleTokenizer* and *ImprovedTokenizer)* and they were tested with two datasets:

- *amazon_reviews_us_Watches_v1_00.tsv* (~393Mb).
- *amazon_reviews_us_Wireless_v1_00.tsv* (~3900Mb).

SimpleTokenizer

amazon_reviews_us_Watches_v1_00.tsv

Number of threads	**No JVM memory restrictions**	**JVM memory restricted to 512mb**
1	1m55.529s	1m57.377s
4	47.522s	45.648s
5	50.946s	49.296s
8	1m02.36s	58.016s

Total index size on disk: ~170Mb

amazon_reviews_us_Wireless_v1_00.tsv

Num. of threads	JVM memory restricted to 512mb
1	24m40.156s
4	28m09:720s
5	33m38.724s

Total index size on disk: ~2300Mb

ImprovedTokenizer

amazon_reviews_us_Watches_v1_00.tsv

Number of threads	No JVM memory restrictions	JVM memory restricted to 512mb
1	7m37.207s	7m20.962s
4	2m33.955s	2m30.773s
5	2m48.218s	2m37.041s
8	2m53.766s	2m50.259s

Total index size on disk: ~170Mb

amazon_reviews_us_Wireless_v1_00.tsv

Number of threads	JVM memory restricted to 512mb
1	1h17m45.477s
4	46m20.962s

Total index size on disk: ~1700Mb

Notes about the results

The results on the *Watches* dataset show that the more efficient number of concurrent threads on this dataset is 4. As the CPU has 4 cores, it represents the ideal parallelism relationship as it is a single thread per core. With less threads, the program will not be maximized as it could be (not fully using the CPU resources) and therefore promoting less parallelism. On the other hand, if we have more than 4 threads, it can have one or more threads queued, slowing up the execution time.

The heap size restricted to 512mb has slightly better results because the garbage collector takes a lot of the time spent during the execution, so if the memory size is lower, it will also have less memory to clean and consequently spend less time doing that operation.

For this reason and due to the limited time available to make these measurements in our personal laptop, we only tested the *Wireless* dataset with the best obtained parameters, therefore maintaining the heap size restriction (512Mb). On this dataset, the results were interesting. With the SimpleTokenizer, the results are better with less threads. On the other side, with the ImprovedTokenizer, the results are better with four threads, just like the previous dataset. This can be caused by several reasons. The main difference between both is the indexer size (the SimpleTokenizer version has a much larger index: 600 Mb more than the ImprovedTokenizer version). One possible reason may be the processor cache, because it is divided by the memory management and one core may want to read a cache line that is being modified by another core, introduces waiting times, slowing down the process. Another reason may be that the context switching can introduce more overhead due to the larger indexes created, and possibly making the program run slower. Finally, because a bigger index means more index files created, the overhead of creating those files can cause a tremendous slow down on the overall execution time, given the IO blocking operations.

From the obtained results, it is clear that the execution time with the ImprovedTokenizer takes more than the double of the execution time with the SimpleTokenizer. This happens due to the complex tokenizing operations applied in the ImprovedTokenizer. It is also possible to check that the resulting index has lower or approximately equal size in the ImprovedTokenizer version than in the SimpleTokenizer. This happens due to the extensive tokenizer operations applied, that filter more tokens that using the SimpleTokenizer version. With these results we can conclude that the tokenizer operations in the ImprovedIndexer version take most of the execution time, because the execution time has more than doubled for all cases, and the indexer should take approximately less or equal time, because there are less or equal number of tokens.

Results

The following results are the answers to the questions proposed on the assessment:

- What is your vocabulary size using each tokenizer?
- For each tokenizer, list the then first terms (in alphabetical order) that appear in only one document (document frequency = 1).
- For each tokenizer, list the ten terms with higher document frequency.

SimpleTokenizer

sample_us.tsv

Vocabulary size: 577 words

Ten first terms that appear only in one document:

- able
- about
- abovemore
- absoloutely
- access
- adding
- adorable
- advent
- again
- age

Ten terms with higher document frequency:

- the
- and
- for
- this
- with
- but
- was
- there
- are
- have

amazon_reviews_us_Watches_v1_00.tsv

Vocabulary size: 236790 words

Ten first terms that appear only in one document:

- aaaaaaaaaa
- aaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaaa

- aaaaaaaaaaaaalll
- aaaaaaaaaaand
- aaaaaaaaabr
- aaaaaaaanything
- aaaaaahhhbr
- aaaaahh

Ten terms with higher document frequency:
- the
- watch
- and
- this
- for
- but
- very
- with
- was
- that

amazon_reviews_us_Wireless_v1_00.tsv

Vocabulary size: 995337 words

Ten first terms that appear only in one document:
- aaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
- aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
- aaa

Ten terms with higher document frequency:
- the
- and
- this
- for

- phone
- but
- not
- with
- that
- was

ImprovedTokenizer

sample_us.tsv

Vocabulary size: 472 words

Ten first terms that appear only in one document:
- 34knock-off34
- 34painted.34
- 34wow
- 6yr
- 7-yr
- abl
- above...mor
- absolout
- access
- ad

Ten terms with higher document frequency:
- love
- like
- great
- use
- littl
- old
- one
- play
- get
- good

amazon_reviews_us_Watches_v1_00.tsv

Vocabulary size: 248601 words

Ten first terms that appear only in one document:
- .....lo
- ....and
- ...br
- ...i
- ...much

- ...still
- ...veri
- ...watchv4lts66ew2ta
- .25second
- .5br

Ten terms with higher document frequency:

- watch
- look
- love
- great
- time
- like
- good
- one
- nice
- band

amazon_reviews_us_Wireless_v1_00.tsv

Vocabulary size: 1159021 words

Ten first terms that appear only in one document:

- .-.the
- .-\\"br
- .-t
- .-tand
- .-that
- .....umm.....hmm.....noth
- .....other
- .....`-...`..``.-.....\.....br
- .....and
- .....that

Ten terms with higher document frequency:

- phone
- case
- work
- great
- use
- one
- good
- like
- fit
- product

Conclusions

The software created in this assignment can successfully create an indexer from a large corpus reader (theoretically of infinite size) to allow better search for terms in documents. Because the processing step relies on creating files of small dimensions, it is possible to process a file that creates a big index that would not fit in RAM memory with the use of the disk memory. The main limitation becomes the disk memory and not the main memory when processing large files.

With the use of a map-reduce architecture, the software achieves higher throughput and lower execution time than single threaded applications. Besides that, it easily scalable to execute on a distributed computing system, improving even more the robustness and the execution time of the indexer creation.

With the use of a TreeMap and TreeSet for the construction of the index and the postings, the tree is always ordered, which prevents the operation of sorting (average complexity $O(n \log_2(n))$) when writing to file. The search operation is also fast when compared to other data structures (average complexity $O(\log_2(n))$), however the insertion takes in average more time than other data structures (average complexity $O(\log_2(n))$).

Several token operations were applied to improve the relevance of the tokens in the dataset. Some tokenizer operations are to remove invisible characters, to remove numbers followed by punctuation or to remove hyperlinks. Although the tokenizer operations largely increase the execution time, by inspection of the terms in the indexer, we believe that the produced tokens are more relevant than the tokens produced by the simple tokenizer.

As a downside, the maximum number of documents read by each file must be set manually. While in most cases the default value is suited to the environment, in systems with very constrained resources it is possible that the system runs out of memory, being needed to adjust (lower) the number of documents by indexer. On the other hand, if a resourceful system is being used, a higher number of documents per indexer can be set, to lower the I/O time needed to write files in memory.

Another downside of the software is the distribution of the terms that start with different alphabetical characters. The different indexes are divided by the first letter of each term, which causes the indexes to be highly unbalanced in terms of size. While there are studies that indicate the frequency of each letter in each alphabet, the implementation would depend on the language of the documents and the type of documents. In this work, those divisions were not applied, because it is considered out of scope for the assignment.

As each map thread reads a fixed number of lines from the corpus reader, the reading process could be made in a batch way. Instead of reading the documents line by line, the lines could be read sequentially and saved in a cache memory. A variation of this method was tested. Each thread used the same CorpusReader class that reads the whole file sequentially, to take advantage of the locality principle. The CorpusReader class was synchronized to avoid concurrency problems. However, the results were worse than with the reading being made by each thread. A probable reason for that to happen is that the corpus reader method of reading the next document was an overhead to the thread execution, since the corpus reader could only read one document at a time. The overhead of reading the

same file in different locations was probably also mitigated because the system where we tested it used an SSD as disk storage, which helped to reduce the burden of reading simultaneously in different locations of the same file.

Finally, the total execution time could be lowered if instead of writing files directly on disk, another approach were taken to reduce the IO blocking time. Again, it was not implemented because it is considered out of scope for this assignment.

On a closing note, we believe that the software produced in this assignment is scalable and robust enough to index terms of large documents with an adequate distributed system, taking advantage of the available parallel computing power.