Bogazici University

CmpE493
Homework 1

Spring 2018

Irmak Kavasoglu
2013400090

Mar 7, 2018

1. Implementation Details

For this project, I have used Java programming language with Eclipse IDE. In this chapter, I will try to explain the steps I have taken to finish the project and explain details of the implementation. You can find the project progress steps in my github.

The main function of the program contains only two lines of code; two function calls for the two parts of the project. First one is the *prepareDictionaryAndIndexes()* which prepares the necessary objects for the engine to work, and the second one is *startQueryEngine()* which takes queries from the user and answers them.

1.1. Preparing Dictionary and Indexes

This part of the program is triggered by the *prepareDictionaryAndIndexes()* function in the *Main* class. It first tries to read the *dictionary* and *indexes* files from the Dataset directory. If it fails to read any of them, it starts recreating both files. To do this, it requires to have *Dataset/reut2-0xx.sgm* files ready. Since it first needs to create the *dictionary* and *indexes* files before it can read it, I will first explain the creation process.

1.1.1 Creating Dictionary and Indexes

If an exception is thrown while reading the dictionary and indexes, recreation process starts. This process is made up of five steps:

- Reading stories from files
- Reading stop words
- Tokenizing stories
- Creating dictionary
- Creating indexes

1.1.1.1 Reading Stories from Files

The *readStoriesFromDocuments()* function is a member of the *Main* class. It creates a *documents* array, which will have an element for each *.sgm* document. It has a for loop to generate the file names and for each file, it uses the *getStoriesFromDocument()* function of the *StoryExtractor* class. The returned results are arrays of news stories. After getting the stories ready, it calls the *giveIDsToStories* to give each story an id.

1.1.1.1.1 Getting Stories from a Document

The *getStoriesFromDocument()* function is a member of the *StoryExtractor* class. It takes in a file name and returns an array of news stories. Each story is kept in a *NewsStory* object. These objects has the following fields:

- Title
- Body
- Title tokens
- Body tokens
- Id

This function creates the story array in four steps. All four of these funcitons are members of the *StoryExtractor* class as well.

1.1.1.1.1.1 Tokenizing by Tags and Strings

The *tokenizeByTagsAndStrings()* function takes in a file name. It reads the file line by line. For each line, using regex, it finds the tags in form "<*(.*?)*>". It divides every line into pieces containing one of the following:

- The part of the line before the first tag
- Tags
- Strings between tags
- The part of the line after the last tag

Then adds these pieces in a String array. This is the result of this function.

1.1.1.1.1.2 Organizing Tags and Strings

The *organizeTagsAndStrings()* function takes in the result of the previous function. Iterating each element, it creates a new array. This new array either contains a tag, or a string which is the merged version of the sequential strings of the other array. The result of this function has the form {tag, string, tag, string, tag…} and so on.

1.1.1.1.1.3 Extracting Texts

The *extractTexts()* function takes in the result of the previous function. The input is the tags for a whole file, and this function divides it into stories. To do this, it uses the <*TEXT*> and <*/TEXT*> tags. Iterating over the elements, it takes the elements between these two tokens and makes a new array out of them. In the end, it returns an array of these arrays. The last *NEWID* before the <*TEXT*> tag is selected as the id for that story.

1.1.1.1.1.4 Extracting Title and Body

The *extractTitleAndBody()* function takes in the result of the previous function. The *NewsStory* format is created in this function. For all given arrays, this function finds the tokens between <*TITLE*> and <*/TITLE*>*, and* <*BODY*> and <*/BODY*> tags. It concatenates these tokens with a space between them and assings them to NewsStory's title and body fields. While doing that, it fixes two special cases:

- "<" which stands for "<" sign
- "&#3;" which stands for "end of the file character"

1.1.1.2 Reading Stop Words

This function is a straightforward one. It reads the *Dataset/stopwords.txt* file line by line and sets the *stopWords* array variable in *StoryTokenizer* class.

1.1.1.3 Tokenizing Stories

The *tokenizeStories()* function is a function of the *Main* class. It takes in the result of *readStoriesFromDocuments()* function. Then for every document, tokenizes the stories by using *readStoriesFromDocuments()* function from the *StoryTokenizer* class. This function iterates all the stories and for every title and body, runs the *tokenizeString()* and *stem()* function in this order. It assigns the results to the *titleTokens* and *bodyTokens* fields of the story.

1.1.1.3.1 Tokenizing Strings

The *tokenizeString()* method takes in a String. It processes each String by following these steps:

- Casefolding: Converts all the letters to lowercase.
- Removing punctuation: Replaces the characters ".", ",", """, "/", "-", "<", ">", "\n" with a space.
- Removing stop words: It iterates over the stop-words array and replaces all of the words by empty string. It makes sure that the replaced String is within a word boundry.
- Tokenizing: It splits the text by the space character and iterates over the words. It ignores the tokens which has lenght less than 2 characters. It also ignores tokens which can be converted to integers.

1.1.1.3.2 Stemming

The *stem()* method takes in an array of Strings. Using the *PorterStemmer* it stems each word and returns the stemmed tokens.

1.1.1.4 Creating Dictionary

After all the previous operations are done, we will have the stories ready and tokenized and stemmed. From this state, it is relatively easy to create the dictionary. The *createDictionary()* method in *Main* class iterates over the documents with tokenized stories and for each document, calls the *updateDictionaryWithNewDocument()* function from *DictionaryBuilder* class.

The update function simply takes all the tokens and adds them to the dictionary with IDs. IDs are determined by the current number of words in the dictionary.

After the dictonary is created, the *writeDictionaryToDocument()* function is called within *createDictionary* function after the *updateDictionaryWithNewDocument()* operation. This function takes in the location of the file from *Constants* class. The *Constants* class has all of the file locations. During the write operation, the HashMap object of the dictionary is written to the file using *ObjectOutputStream*'s *writeObject()* function.

1.1.1.5 Creating Indexes

The *createIndexes()* function is similar to *createDictonary()* function. It iterates over the files and calls an update indexes function for each of them. In the end, it calls a function to write the results to a file.

1.1.1.5.1 Updating Indexes with a Document

The *updateIndexesWithNewDocument()* function is the main function that does indexing. The working principle is as follows.

- It iterates through the stories.
- For every story, it iterates over the title and body tokens.
- For every token, it gets the word id from the dictionary.
- After getting the word id, it checks the indexes to see if the word has already been indexes before. If not, adds an empty entry for that word.
- Next, it gets the current entry for that word in a variable. This entry will contain a HashMap, containing page IDs as keys and indexes as values.
- It checks if the current page (story id) is already in the variable. If not, adds an emty entry.
- Next, it gets the current occurences of that word within the current file.
- Since it is iterating over the tokens currently and this word is the current word, it adds the current word index as an occurence to the indexes list of that word within this page.
- It updates the occurences of the word within current page.
- It updates the current page within the indexes variable.
- Same things happen for the body part as well. Body tokens are ID'ed starting from the last id of the title tokens plus one.

1.1.1.5.2 Writing Indexes to a Document

Now, this was a challenge. I will explain why. Since the dictonary is a smaller object and the created file is around 700-800 KBs, it was very okay to use the *writeObject()* method. But when I did the same thing to the indexes object, I realized it was super slow.

Here are some optimization steps I have taken with approximate storage/speed information

- No Optimization:
  If I use *writeObject()* to write down the whole *indexes* object;
  - It takes 45 mb of disk space
  - It takes 1 minute and 15 seconds
- Optimization 1:
  I was keeping the indexes of the occurances of the word within a page in an ArrayList<Integer>. I thought maybe if I use an int[] it might be faster, which it did;
  - It takes 32 mb of disk space
  - It takes 29 seconds
- Optimization 2:
  Then I realized that the outer HashMap was for word IDs and word IDs are starting from zero and incrementing until it reaches the vocabulary size. This means that I can just use line number as this id and do not need to write the outer HashMap. When I did it;
  - It takes 26 mb of disk space
  - It takes 24 seconds
- Optimization 3:
  After figuring out that each Java object occupies a lot of space and requires time to be prepared, I decided to get rid of all objects and write down everything in primitive types, that is, integer. So, instead of writing arrays as objects, I wrote down the size of it and then the elements of it in int form. The results were amazing;
  - It takes 17 mb of disk space
  - It takes 1.5 seconds

So I sticked with the last one.

1.1.2 Reading Dictionary and Indexes

These methods are pretty straightforward. The *readDictionaryFromDocument()* and *readIndexesFromDocument()* functions try to find the files in given addresses, which are again taken from *Constants* class, and convert them into Java objects. The methods are relatively in *DictionaryBuilder* and *IndexBuilder* classes.

If the files cannot be read or cannot be found, these methods throw an exception and cause the creation process to start.

1.2. Starting Query Engine

The query engine is started by calling the function *startQueryEngine()* in class *QueryManager*. This function expects queries from user in an infinite loop. The expected input is one integer indicating which type of query it is going to be and a string which is the query. The query types are:

- 0: Exits the engine without waiting for a query.
- 1: Expects a conjunctive query.
- 2: Expects a phrase query.
- 3: Expects a proximity query.

The query engine will start the related function with the rest of the query (excluding the query type).

1.2.1 Conjunctive Queries

The *processConjunctiveQuery()* function takes in a query in form "w1 AND w2 AND w3" etc. To get the word list, it replaces all ANDs with an empty string, it makes sure that these ANDs are within word boundaries. Then it uses three functions:

- *tokenizeString()* from *StoryTokenizer* class
- *stem()* from *StoryTokenizer* class
- *filterOutNonDictionaryWords()* from *QueryManager* class

We already know what the first two functions do from previous explanations. The third function is to get rid of any words which are not in the dictionary, such as stop words.

The rest is pretty easy;

- It gets the *wordID* of each token in the query from the dictionary.
- Gets the *keySet()* of each word using *wordID* from *indexes*, these are the *pageIDs*
- It takes the intersection of these *pageID* sets and orders them.

1.2.2 Phrase Queries

The *processPhraseQuery()* function takes in a query in form “w1 w2 w3” etc. It follows the same three steps of functions as the previous one. The next steps are as follows:

- It gets the first word’s occurences as the base. This is gotten from *indexes.*
- It iterates over each word. For each word, the *wordID* is captured from dictionary.
- The occurences of the word is gotten. So in each loop, we have the occurences we have up to now and the occurences of the current word.
- The current pages are gotten from the occurences keyset (whenever I say occurences, I mean the set we built up to now.
- It iterates over these current pages.
- For each page, which we now was valid until now, the current word’s occurences are checked. If the current word does not occur in that page, then we discard it.
- If that page is in the list for the current word as well, we check the occurence indexes. If the current word occurs in anywhere which is one word later than the existing occurences, we accept the page. Otherwise, we decline the page and discard it.
- In the end, we return the keyset of our newly built occurences object.

1.2.3 Proximity Queries

The *processProximityQuery()* function takes in a query in form “w1 /k1 w2 /k2 w3” etc. It is exactly the same as the phase queries except for one thing. When the function checks for the page validity, it does not only check the occurence for *order+1*, it checks every occurence from *order+1* till *order+k+1*.

2. Statistical Information

This section answers the questions from the project description.

a. How many tokens does the corpus contain before stopword removal and stemming?

It has **2831183** tokens.

b. How many tokens does the corpus contain after stopword removal and stemming?

It has **1898345** tokens.

c. How many terms (unique tokens) are there before stopword removal, stemming, and case-folding?

It has **237570** terms.

d. How many terms (unique tokens) are there after stopword removal, stemming, and case- folding?

It has **37939** terms.

e. List the top 20 most frequent terms before stopword removal, stemming, and case-folding?

They are:

the, to, and, said, mln, Reuter, that, will, with, company, were, market, they, Bank, after, trade, first, could, price, expected.

f. List the top 20 most frequent terms after stopword removal, stemming, and case-folding?

They are:

to, said, pct, billion, not, net, price, trade, two, unit, profit, debt, offici, secur, total, februari, current, five, januari, firm.

3. Data Structures

The project has two main structures. The dictionary and the indexes objects.

3.1 Dictionary

The dictionary has the form:

HashMap<String, Integer>

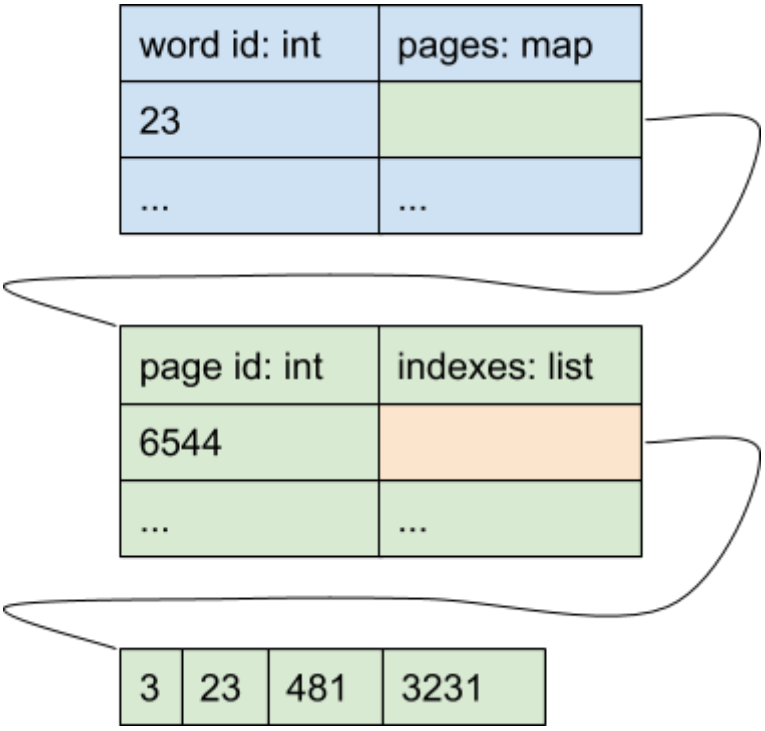
Word: String	Word Id: Int
“secur”	0
“januari”	1
…	…

The stemmed words are the keys and the values are unique word IDs.

3.2 Indexes

The indexes (or postings) has the form:

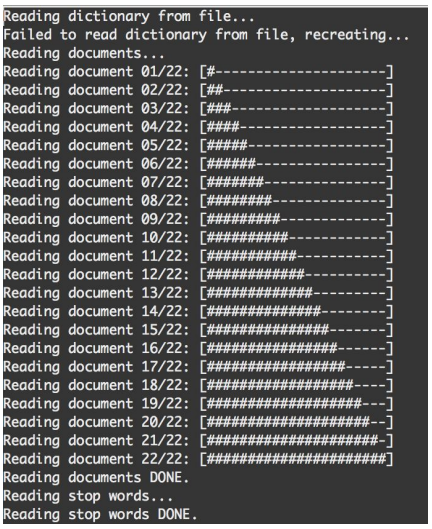
HashMap<Integer, HashMap<Integer, ArrayList<Integer>>>

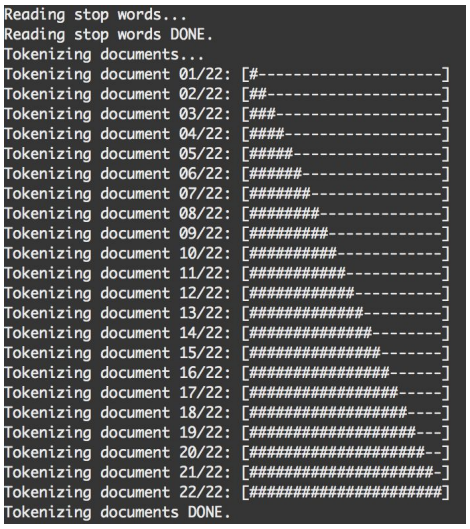


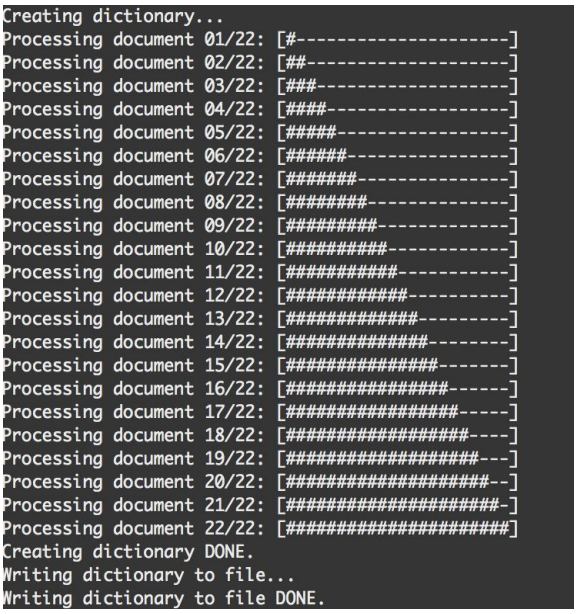
Outer HashMap has the keys as the word ids. Each word id has the pages it has occured. Each page id has a list of indexes where that word occured in that page.

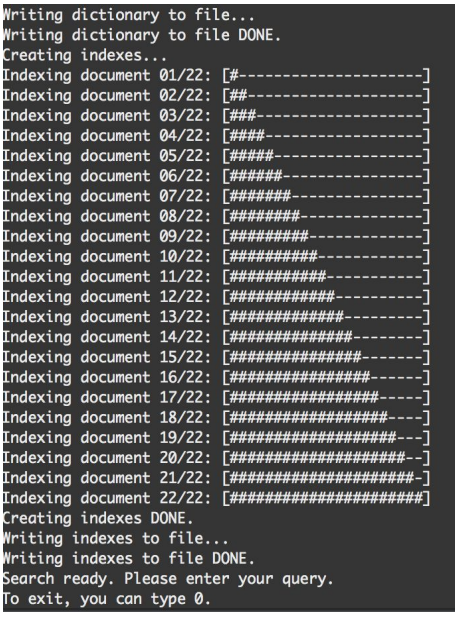
4. Screenshots

The screenshots of the program running for the first time:

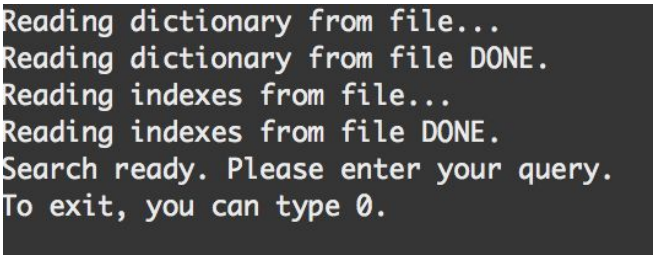








The screenshots of the program after the first run:



4.1 Conjunctive Query

```
1 rains
[319, 634, 1246, 1499, 1535, 1582, 1785, 2389, 3110, 3364, 3560, 3561, 3576, 3981, 4008, 4470, 5289, 5826, 6114, 6153, 6645, 6829, 6890,
1 showers
[1, 6114, 6118, 6142, 6153, 8705, 10388, 11536, 13618, 15043, 16975, 17241, 19262]
1 rains AND showers
[6114, 6153, 8705, 10388, 11536, 13618, 15043, 16975, 17241, 19262]
```

4.2 Phrase Query

```
2 heavy
[110, 278, 339, 340, 346, 633, 875, 926, 942, 960, 1019, 1093, 1246, 1312, 1410, 1616, 1738, 1836, 1854, 1892, 1902, 1906, 1911,
2 showers
[1, 6114, 6118, 6142, 6153, 8705, 10388, 11536, 13618, 15043, 16975, 17241, 19262]
2 heavy showers
[6153]
```

4.3 Proximity Query

```
1 rains AND showers
[6114, 6153, 8705, 10388, 11536, 13618, 15043, 16975, 17241, 19262]
2 rains showers
[]
3 rains /5 showers
[16975]
3 rains /10 showers
[8705, 16975, 19262]
3 showers /10 rains
[6153, 16975]
```