SPIMI Project 1

Concordia University

COMP 479

Fall 2019

Prepared by
Eve Raya, 40016485

Packages, Classes, methods and Folders

BlockHandling (pckg): This package has a java class with the same name that includes two functions:

- *addToBlock(String token, String docID)*
  - This function is used to add the token with its' respective docId's found to a block
- *getBlock(String blockName*)
  - This function is designed to retrieve a block with its' respective LinkedHashmap that includes the terms with their posting lists.

FileHandling (pckg): This package has a java class with the same name that includes two functions:

- *sendToBlock(int blockNumber):*
  - This function takes the block number and creates a new block file with the corresponding number and outputs the non-sorted dictionary to it.
- *sendToFinalFile(TreeMap sortedList, int final_file_number):*
  - This function uses a TreeMap to sort the keys alphabetically and sends every 25,000 terms to a final block text file.
- *DeleteFiles():*
  - This function is used at the beginning of the Main.java to delete any previous blocks in the DISK folder.

Common (pckg): This package has an abstract java class with the same name that includes the following, the abstract class is used to be able to instantiate the same variable in more than one class differently:

- Getters and setter for the *SortedDictionnary* using a *TreeMap<String, LikedHashSet<String>>*
- Getters and setters for the *PostingList* using *LinkedHashSet<String>*
- Getters and setters for the *Dictionnary* using *HashMap<String, LinkedHashSet<String>>*

CompressionTable (pckg): This package has a java class with the same name that includes the following:

- *noNumber*:
  - count of numbers removed
- *case_fold*:
  - count of case_fold done
- *stop_words*:
  - count of stop words removed
- *percent_numbers_from_unfiltered*:
  - % of numbers from unfiltered
- *percent_casefolding_from_unfiltered*:
  - % of case folding from unfiltered
- *percent_stopwords_from_unfiltered*:
  - % of stopwords from unfiltered
- *percent_previous_numbers*:
  - % of numbers from previous
- *percent_previous_case_folding*:
  - % of case folding from previous

- *percent_previous_stopwords*:
    - % of stopwords from previous
- *printTable():*
    - prints the compression table

Preprocess (pckg): This package has a java class with the same name that includes the following:

- *ArrayList<String> Stopwords:*
    - has a list of stopwords we will compare to later
- *Remove_stopwords(String token):*
    - method to remove stopwords and count them
- *Remove_numbers(String token):*
    - method to remove digits and count them
- *Case_folding(String token):*
    - method to convert to lower case and count the conversions
- *Punctuation(String token):*
    - method to remove unnecessary punctuation

Query(pckg): This package has a java class with the same name that includes the following:

- *searchTerm(String query):*
    - searches the FinalBlocks for 1 query term and outputs the posting list
- *searchDictionary(String query):*
    - searches the FinalBlocks for the query term
- *CustomAndQuery(String query):*
    - searches the dictionary for the terms in the query and performs the intersection
- *CustomORQuery(String query):*
    - searches the dictionary for the terms in the query and performs the union
- *intersection(List<String> list1, List<String> list2):*
    - method for intersection used in AND query
- *union(List<String> result, LinkedHashSet<String> linkedHashSet):*
    - method for intersection used in OR query
- printQueries():
    - prints the custom queries

Spimi_merge (pckg): This package has a java class with the same name that includes the following:

- *Merge(int nbrOfBlocks):*
    - merge all 42 blocks into final blocks of 25,000 terms.

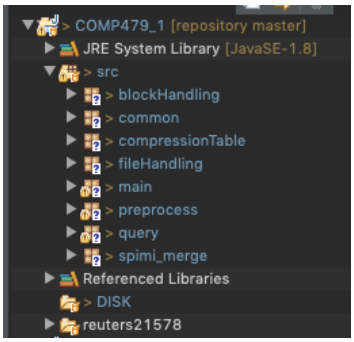
Main (pckg): This package has a java class with the same name that includes the following:

- The main.java runs the program by calling all previous functions. It starts by running the the reuters files and parsing them, tokenizing them and sending them to blocks. Then these blocks are merged, sorted and sent to final files. A query search term is done at the end and a compression table is printed.

DISK (Folder): This folder contains all of the different blocks created when storing the terms in the dictionary. There are currently 42 blocks created, in which each block has terms from 500 different

documents. This folder also includes the FinalBlock file, which has all of the terms merged from all 42 blocks, which are sorted by their keys in alphabetical order.

Reuters21578(Folder): This folder contains the reuters files from which the SPIMI inverted index is created



How the project works

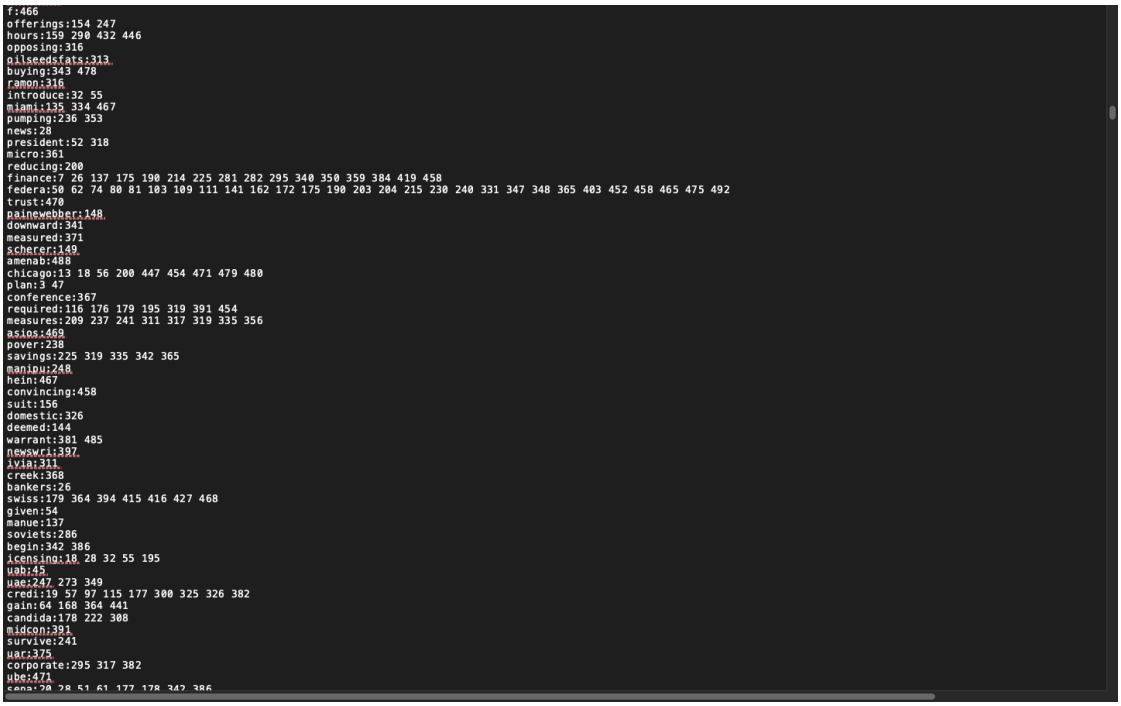
To run the program, only the Main.java needs to be used. The system will output each time it runs through a file to start indexing the terms into a dictionary and then use the compression techniques while preprocessing the files. Once this is done, each document ID is saved in a term-docID posting list to be used later for query search. There is a tracker for the document ID, when it reaches 500, the posting lists are then stored into a block and the tracker goes back to 0. After all the blocks are generated, we merge them into a final block sorted alphabetically. Each 25,000 terms are written to a block.

For the dictionary, a LinkedHashSet was used for the values of the posting list because it removes duplicates, compared to an ArrayList.

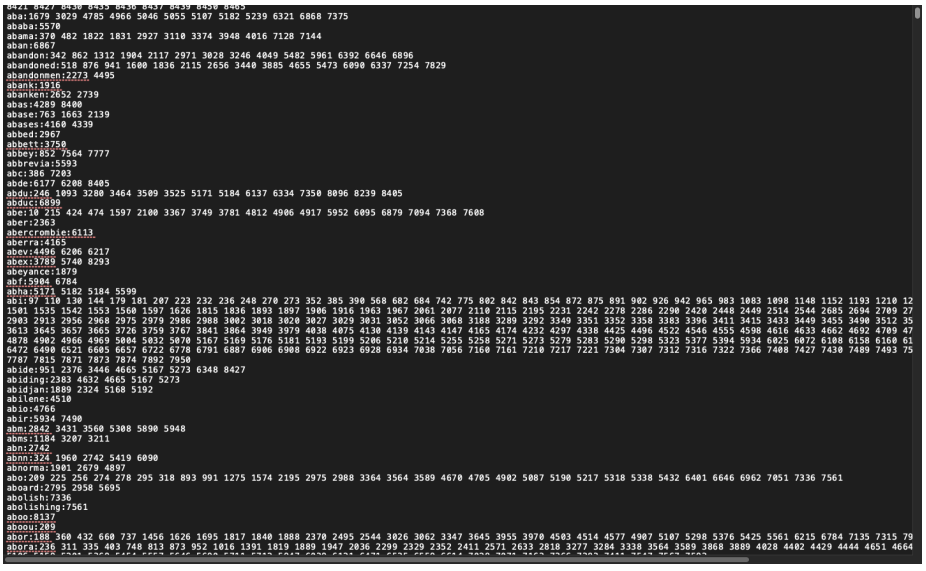
When merging into the final block, we need the keys to be sorted alphabetically, so a TreeMap is used.

Block Example

Example of Block1.txt

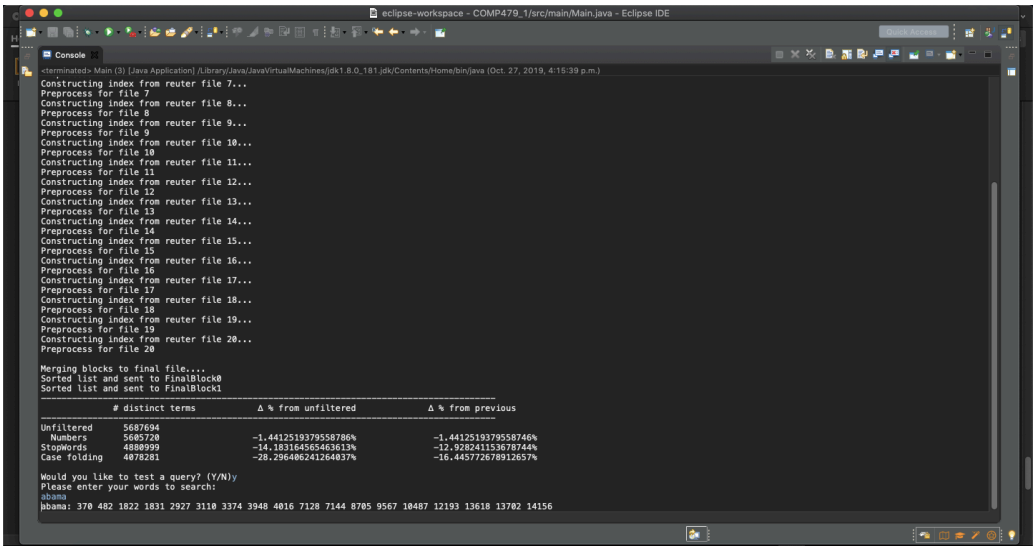


Final Block Example



Test Queries

- *1 query term*



As shown above, in the FinalBlock0.txt, all doc ID's are found in this query search. The only bug in my code is that when the 25,000 terms are separated into more than 1 Final block, they are not grouped by terms. Meaning that the same term can be found in both my final blocks even though they are still sorted alphabetically. But we can still manage to see that in order to search for a query, all final blocks are searched, and the correct result is outputted.

- *AND queries*
  **While comparing with my colleagues, I found out that for the second query, mine was empty. I thought at first that my AND query method was faulty, and this is why I have added other queries not offered by the teacher. Since my new queries came back with a result, I came to the conclusion that maybe I had done a parsing and compression different than my colleagues, which might give different results. **
  - *Jimmy AND Carter: []*
  - *Green AND Party: []*
  - *Innovation AND in AND telecommunication: []*
  - *Abusive AND Added: [1477]*
  - *Advances AND age: [2601, 1694, 8137, 12741, 3185, 296]*

```
==========Starting with AND queries==========
jimmy AND carter: []
green AND party: []
abusive AND added: [1477]
advances AND age: [2601, 1694, 8137, 12741, 3185, 296]
innovation AND in AND telecommunication: []
```

- *OR queries (I added extra queries to make sure the OR query works since the one provided was empty)*
  I did not order them
  - *Environmentalist OR ecologist: []*
  - *Jimmy AND Carter: [13540, 12136]*
  - *Green AND Party: [8630, 5124, 6698, 8193, 2404, 13629, 12878, 1753, 7415, 10230, 16115, 15209, 2627, 3714, 3397, 2162, 13190, 1249, 4719, 345, 10682, 9328, 5107, 12300, 14427, 707]*

```
==========Starting with OR queries==========
environmentalist OR ecologist: []
jimmy OR carter: [13540, 12136]
green OR party: [8630, 5124, 6698, 8193, 2404, 13629, 12878, 1753, 7415, 10230, 16115, 15209, 2627, 3714, 3397, 2162, 13190, 1249, 4719, 345, 10682, 9328, 5107, 12300, 14427, 707]
```

Conclusion

While completing this project, I firstly tried to do it using python. I figured that with all the libraries available for NLP it would be easier. But learning a new language in a short period of time was not the most efficient way to do this. I then turned to Java, even though I knew the level of difficulty would be much higher because of the lack of libraries and frameworks. That being said, I did learn a lot. I found that the pre-processing part of the assignment was one of the toughest. It took me a while. To figure out how to split the html document and read the document ID and then the body of it. After that, the tokenization was not a big problem, neither was the compression techniques.

I started the project with one main class, as a monolithic approach. Towards the end, I started splitting classes into packages that made more sense for them to be together. This makes the code look cleaner than having one class. This way it would be easier to implement more features in Project 2 without having to redo all the code.

Moreover, I did find this assignment to be quite hard. It took a lot of thinking and analyzing different approaches possible. But at the end, I learned the true meaning of efficiency and optimization.