Spring 2016

Concept Based Search Engine: Concept Creation

Aishwarya Rastogi
San Jose State University

Concept Based Search Engine: Concept Creation

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfilment

of the Requirements for the Degree

Master of Science

By

Aishwarya Rastogi

February 2016

The Designated Project Committee Approves the Project Titled

CONCEPT BASED SEARCH ENGINE: CONCEPT CREATION

By

Aishwarya Rastogi

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

February 2016

Dr. Tsau Young Lin	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Mr. Eric Louie	IBM Research Center

ABSTRACT

Concept Based Search Engine: Concept Creation

By Aishwarya Rastogi

Data on the internet is increasing exponentially every single second. There are billions and billions of documents on the World Wide Web (The Internet). Each document on the internet contains multiple concepts (an abstract or general idea inferred from specific instances).

In this paper, we show how we created and implemented an algorithm for extracting concepts from a set of documents. These concepts can be used by a search engine for generating search results to cater the needs of the user. The search result will then be more targeted than the usual keyword search.

The main problem was to extract concepts from a set of documents. Each page could have thousands of combinations that could be potential concepts. An average document could have millions of concepts. Combine that to the vast amount of data on the web, we are talking about an enormous amount of dataset and samples. As a result, the main areas of concern are the main memory constraints and the time complexity of the algorithm.

This paper introduces an algorithm which is scalable, independent of the main memory and has a linear time complexity.

ACKNOWLEDGEMENTS

I would like to thank my project advisor, Dr. Tsau Young Lin, for his constant guidance and trust on me. It wouldn’t have been possible without his contribution throughout the project.

I would also like to thank my committee members Dr. Robert Chun and Eric Louie for their invaluable advices, crucial comments during the project development and taking out time for guiding me throughout the project.

Most importantly, I would like to thank my family and friends for always being there and providing me with the love and support throughout my master’s program. It would not have been possible without their constant encouragement and blessings.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

The objective of this paper is to come up with an algorithm to generate concepts from a given set of documents. Let us start with describing what a **set of documents** is. The set of documents for this project is a set of ten thousand IEEE papers which were converted to text files. The input for this algorithm can be any set of text files.

A **concept** is defined as "an abstract or general idea inferred or derived from specific instances". Here we consider concepts as a word or combination of words that appears in multiple documents and represent an idea or an abstract e.g. Data Structure, Neural Networks, Wall Street. In this project the maximum number of words in concept are six as most of the concepts will be covered with a concept length six and also it will decrease the number of combinations created exponentially. A point to note is that the concepts are a subset of the possible combinations.

If we consider this algorithm as a black box, the input and output for this project will look something like this,

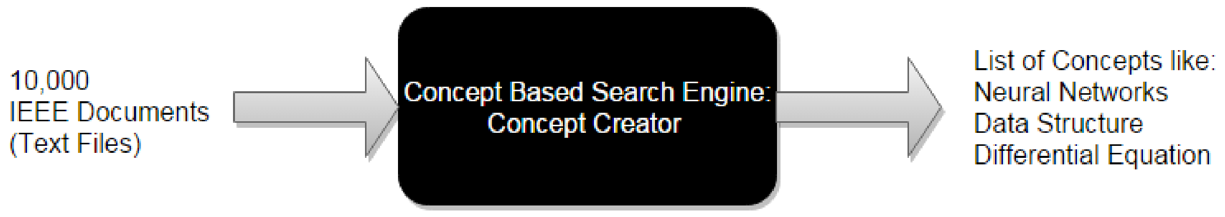


Figure 1.1: Concept Creation Overview

CHAPTER 2

Hefty Combination Challenge

As we have mentioned in the introduction section that concepts are a subset of combinations that means we need an efficient algorithm to generate and store combinations from a group of words and handle it tactfully and efficiently.

The most difficult problem to tackle while creating combinations for a set of words is the exponential growth of the possible outcomes with each additional word. If we consider all the possible combinations for a single page it would be huge. There can be a large number of combinations for every page. Let's take an example.

If we look at different **combinations** possible from a group of words then it would be exponential. The table below shows all possible combinations that can be created by adding a single word in every step. Let us assume A, B, C and D are four words, then the table below list the possible combinations and number of possible combinations.

Words Considered for creating combinations	Combinations	Number of combinations
A	A	1
A B	A, B, AB	3
A B C	A, B, C, AB, AC, BC, ABC	7
A B C D	A, B, C, D, AB, AC, BC, AD, BD, CD, ABC, ABD, ACD, BCD, ABCD	15

Table 2.1: Number of Words Vs Number of Combinations

As we can see from the table above, the numbers of combinations increase exponentially with the number of available words. This becomes worst when we are trying to consider the combinations for the whole page which may easily contain more than five hundred words.

To give you an idea of how huge the number of combination can grow we have a graph which shows the exponential growth of the combinations. The number of combinations reaches up to a billion in just 30 words.

As the number of words grows up to 30 words, the number of combinations will look something like this. The graph below shows us the statistic of the growth of the number of concepts vs the number of words.

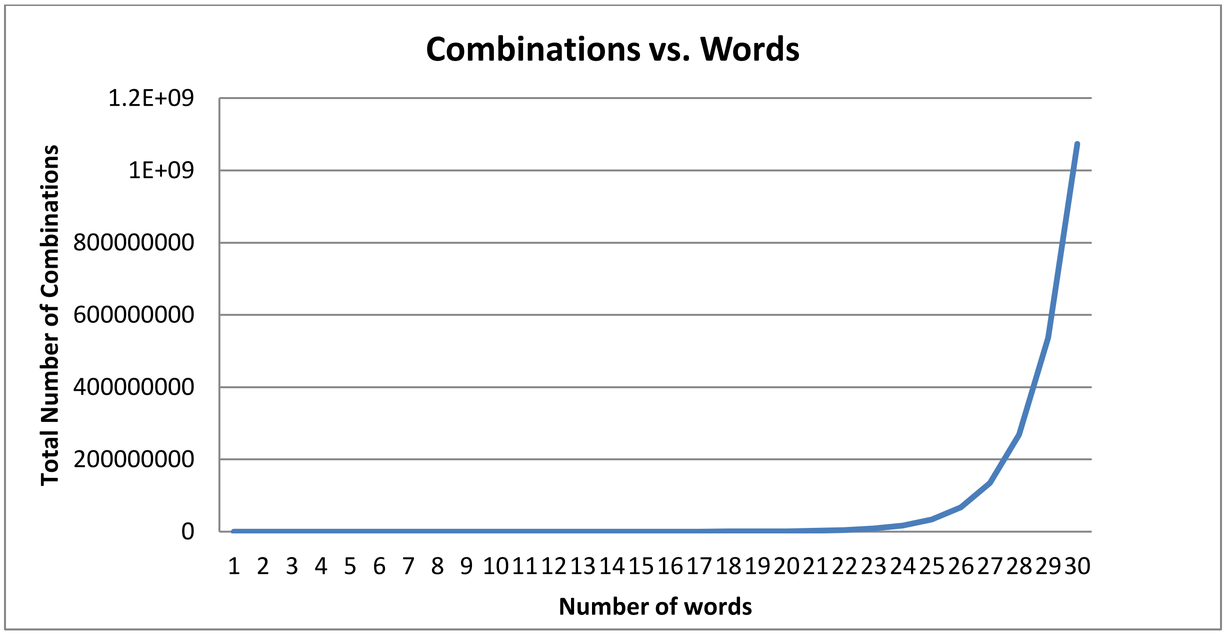


Figure 2.1: No. of Words vs. No. of Combinations

Now as we have defined the major challenge let's discuss in detail how we designed and implemented the algorithm. The aim of the project was to design an algorithm and come up with an implementation that is independent of the main memory constrains caused by the hefty combination challenge and hence making it scalable for a larger number of documents.

CHAPTER 3

Concept Based Search Engine

In this chapter we will give an overview of the entire algorithm and in the later sections we will define each module of the code in detail. We have tried to include software engineering and UML diagrams wherever possible to give you a better understanding of the algorithm and the reasoning that went behind it.

The whole software is divided into four main parts:

- Pre-Processing raw data files
- Creating all possible combinations
- Merging all the combinations
- Filtering combinations to create Concepts

Below is the flowchart of the code execution of these four modules:

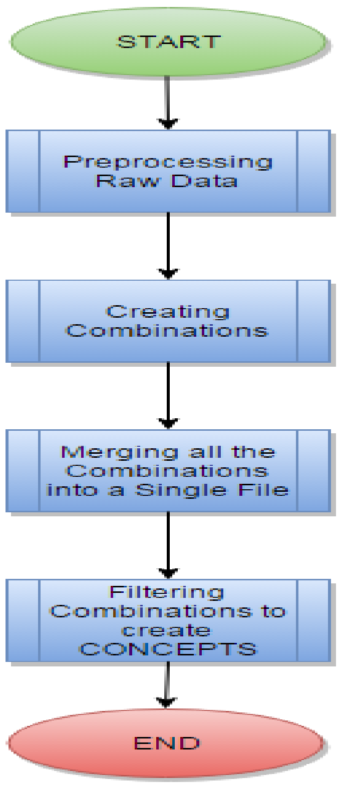

START
Preprocessing Raw Data
Creating Combinations
Merging all the Combinations into a Single File
Filtering Combinations to create CONCEPTS
END

Figure 3.1: Bird's eye view of the Algorithm

Preprocessing Raw Data:

This module just takes the raw data that was provided by the web crawler in text format and makes it ready for the concept creator. The main function of this preprocessor is to remove stop words, remove all the special character and convert them all to a lower case. **Stop words** are the most common words in a language; in our case we took a list of around six hundred stop words from the internet.

Algorithm:

- Reading the stop word file line by line and adding the stop words in a hash map.
- For each file in the Raw Data set:
    - Append it to the map file (File ID: File Name).
    - For every line in the file:
        - Remove all the special characters.
        - Split the line into words.
            - For each word in the line:
                - Check if the word is a stop word. If not, change it to lowercase and write it to the new file.

Creating Combinations:

This is the most important module of the algorithm. The combinations are created here in this module. The main function of this module is to create combinations for every page reading one paragraph at a time.

Algorithm:

- For each page which was created after the per processing stage:
    - Create a hash-map
    - For each paragraph (30 words) in the page:
        - Make combinations
        - Merge the created Combinations and their frequencies in the page hash-map
    - Sort the Hash-map key set
        - Write all the keys in the Keyset with the DocID in a file

Merge Combinations:

In this module we merge all the combinations created in all the files into a single file. The algorithm is very much similar to a merge sort as the input files for this module are already in a sorted order.

Algorithm:

- For every 2 files of the combinations created in the above module:
    - Merge them together.

Filter Concepts:

This last module separates the concepts from the combinations. The main function of this module is to calculate the number of documents in which a combination was present and filter out the combinations that do meet the frequency threshold.

Algorithm:

- For each line in the merge combination file:
    - Calculate the document frequency of each document.
    - If the frequency is greater than the filter frequency declared, it as a concept, else ignore and read the next line.

Below is the diagram that gives an overview of files created throughout the project:

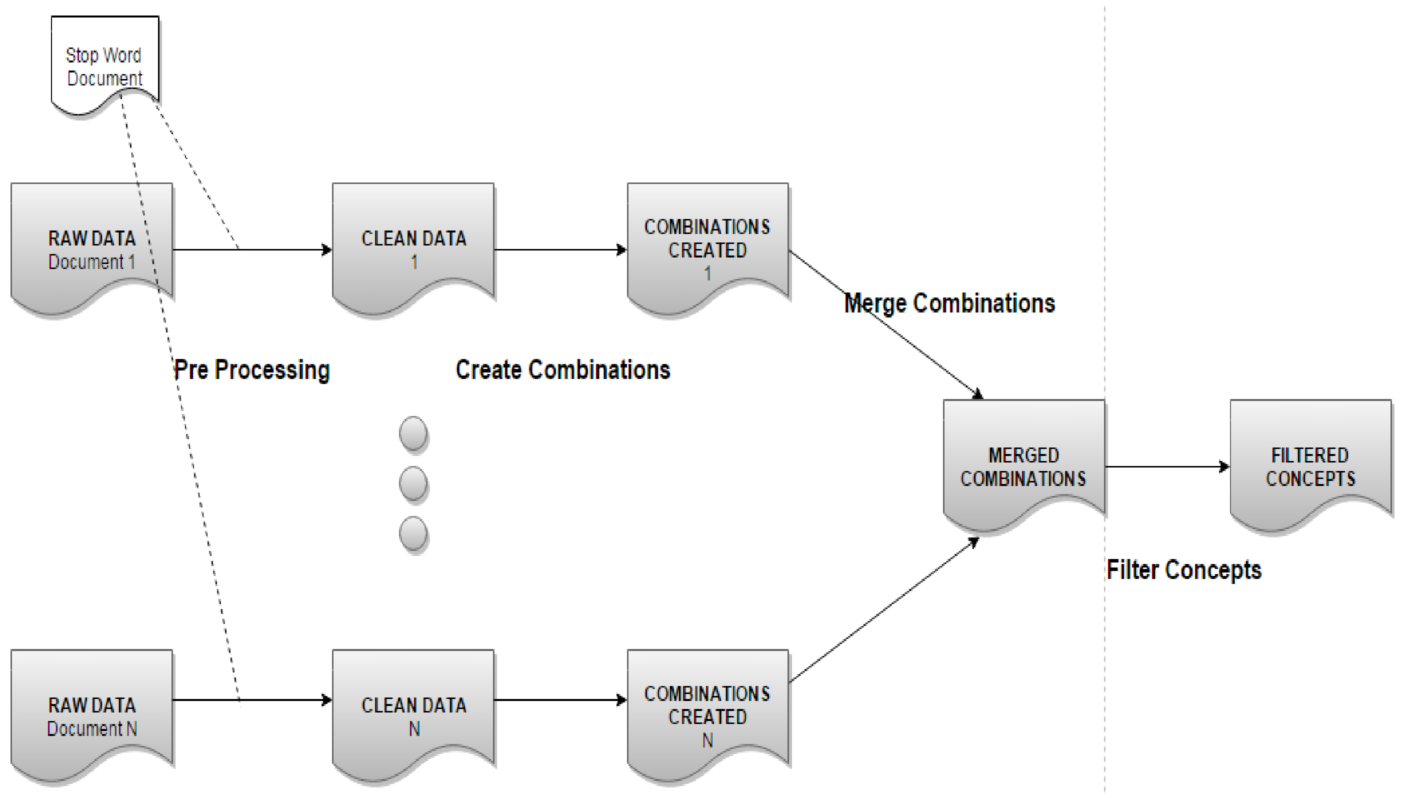
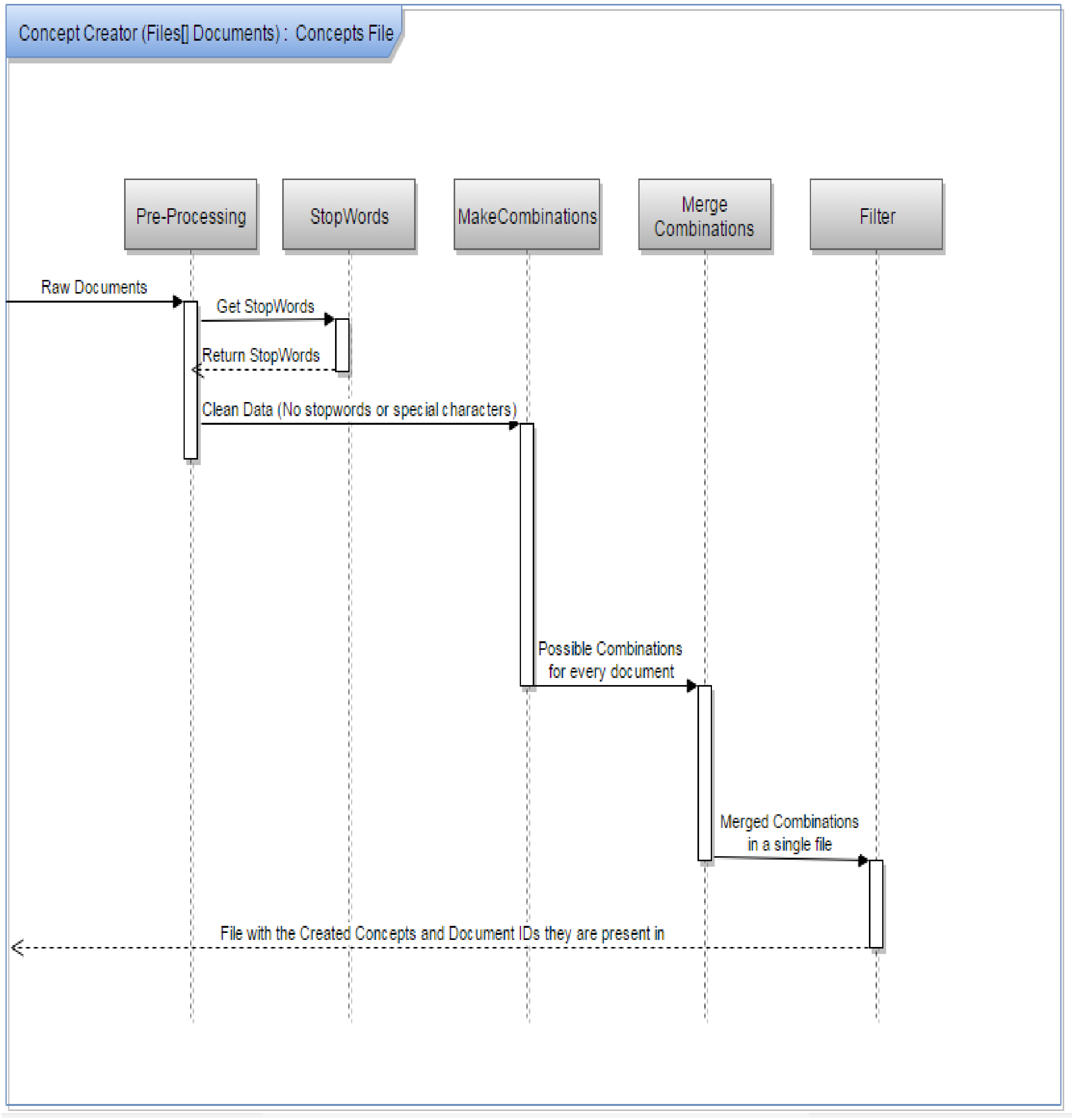


Figure 3.2: Algorithm

Input for this project was a set of Raw Data files. After we are done with the Pre-Processing we get a set of clean data files which can be easily processed by the create combination module. Once all the clean data files are processed by the create combination module into created combinations files, these files are merged into a single merged combinations file. In the last phase the concepts are filtered and placed in the filtered concepts file.

Now let's take a look at a few more UML diagrams and try to understand the overall algorithm in a little more detail.

Below is the Sequence diagram of the entire algorithm. It shows how and when the objects interact with each other and when is the control given to one of the objects:



Concept Creator (Files[] Documents) : Concepts File
Pre-Processing
StopWords
MakeCombinations
Merge Combinations
Filter
Raw Documents
Get StopWords
Return StopWords
Clean Data (No stopwords or special characters)
Possible Combinations for every document
Merged Combinations in a single file
File with the Created Concepts and Document IDs they are present in

Figure 3.3: Sequence Diagram

As we can see in the previous diagram the raw data files are send to the Preprocessing modules and it interacts with the stop words object to remove all the stop words present in the raw data. Once the data is clean it is send to the Make Combination module which then generates the list of possible concepts in the given set of clean data files and send it out for the next step, the Merge Combination module. The Merge Concepts module then merges all the generated concepts to a single file and passes it to the Filter module where the concepts are separated from the list of combinations provided to it.

Now let’s take a look at the different classes used in this algorithm with a class diagram that shows the classes, their attributes and operations of these classes. As we know that the whole project is divided onto four modules Pre-Processing, Creating Combinations, Merging Combinations and Filter Combination. Let’s look at these classes one by one:

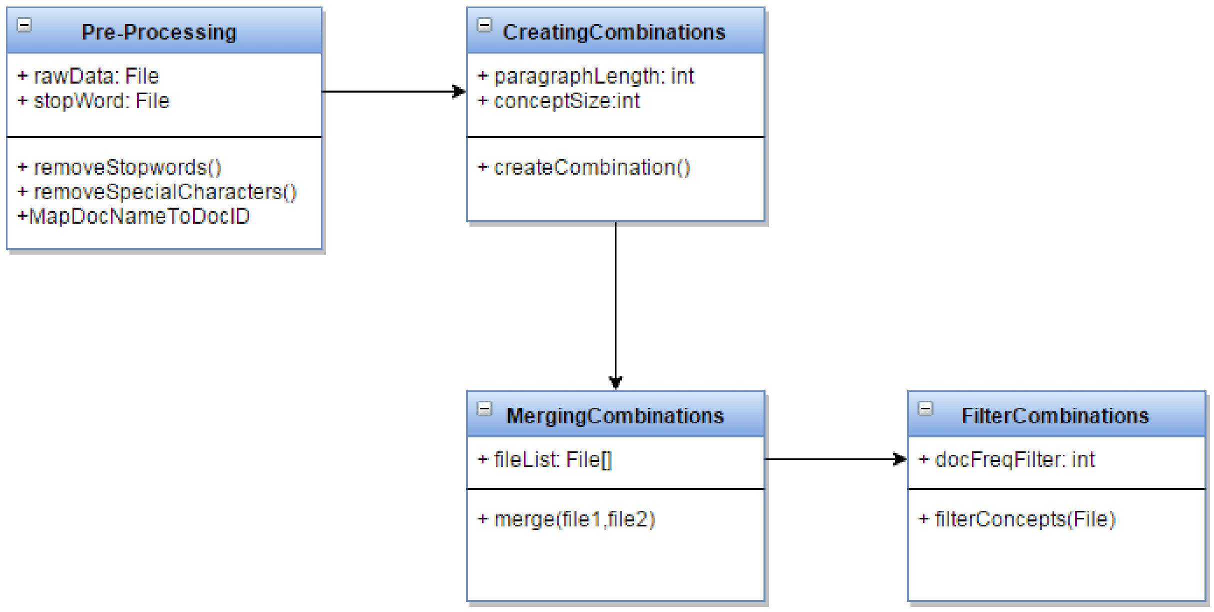


Figure 3.4: Class Diagram

Pre-Processing Class: The two most important attributes of this class are the raw data files and the list of stop words. The most important operations performed by this class are removing stop words, removing the special characters and mapping the file name with the file ID in a map file.

Creating Combinations Class: The two main attributes of this class are the paragraph length and concept size. *Paragraph length* is the number of words we consider as a paragraph. Two words are considered to be in one combination only if they are present in the same paragraph. *Concept size* is the maximum number of words a combination can have .The operation performed by this class is creating the combinations. We will talk about the create combination method later in detail.

Merging Combinations Class: The attribute is the file list which is the set of files that was send by the create combination module and the operation is the merge operation.

Filter Combinations Class: It contains the document frequency filter as the attribute. The *document frequency filter* is the frequency of the combinations in all the documents. The operation Filter Concept uses the document frequency filter to separate the concepts from the combinations.

The Entity-Relationship diagram shows the relationship between the different objects. It is a good way to look at the different objects and how these objects are related.

Below is the Entity-Relationship diagram:

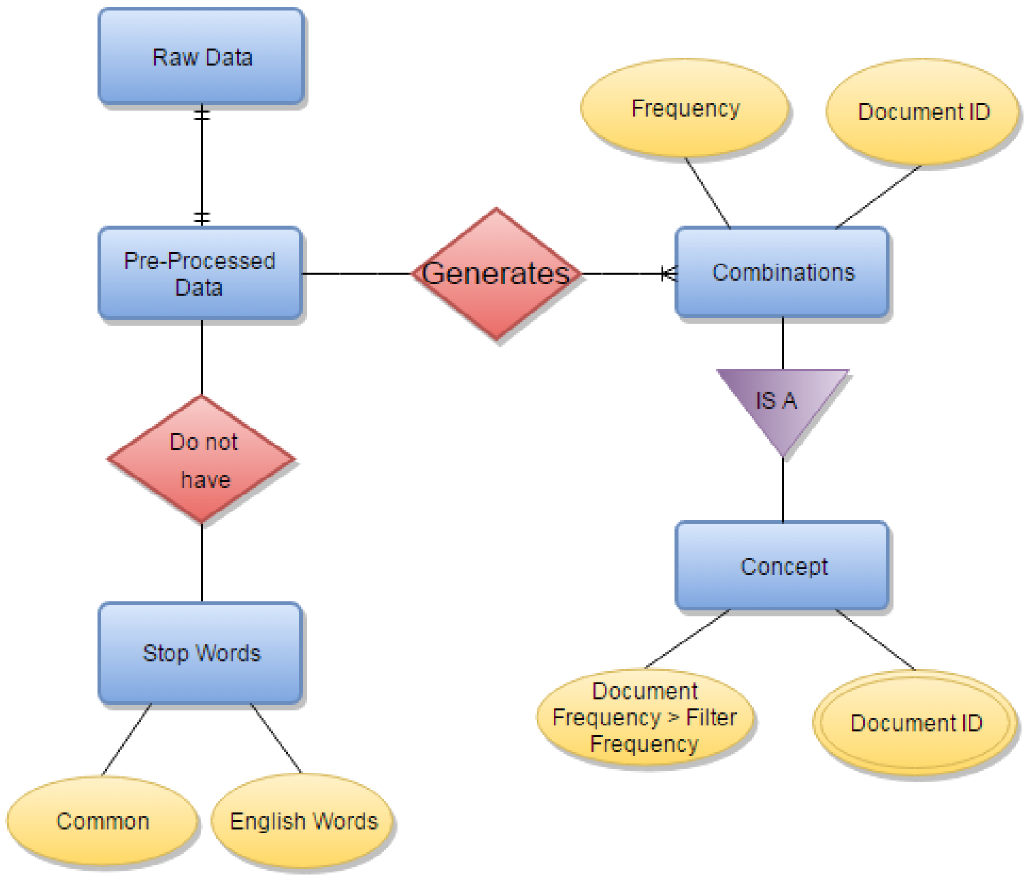


Figure 3.5: Entity - Relationship Diagram

From the ER Diagram we can see that for every Raw Data file there is one Pre-processed file. The Preprocessed data *do not contain* any stop words. The stop words have attributes like "common" and "English words".

From every preprocessed data files there are multiple combinations generated and every Combination is connected with the frequency and document ID in which it was present.

The concept is a combination which has a document frequency greater than the filter frequency.

A Use Case diagram can help us figure out how the search engine can use the concept creator. The Use Case Diagram looks something like this:

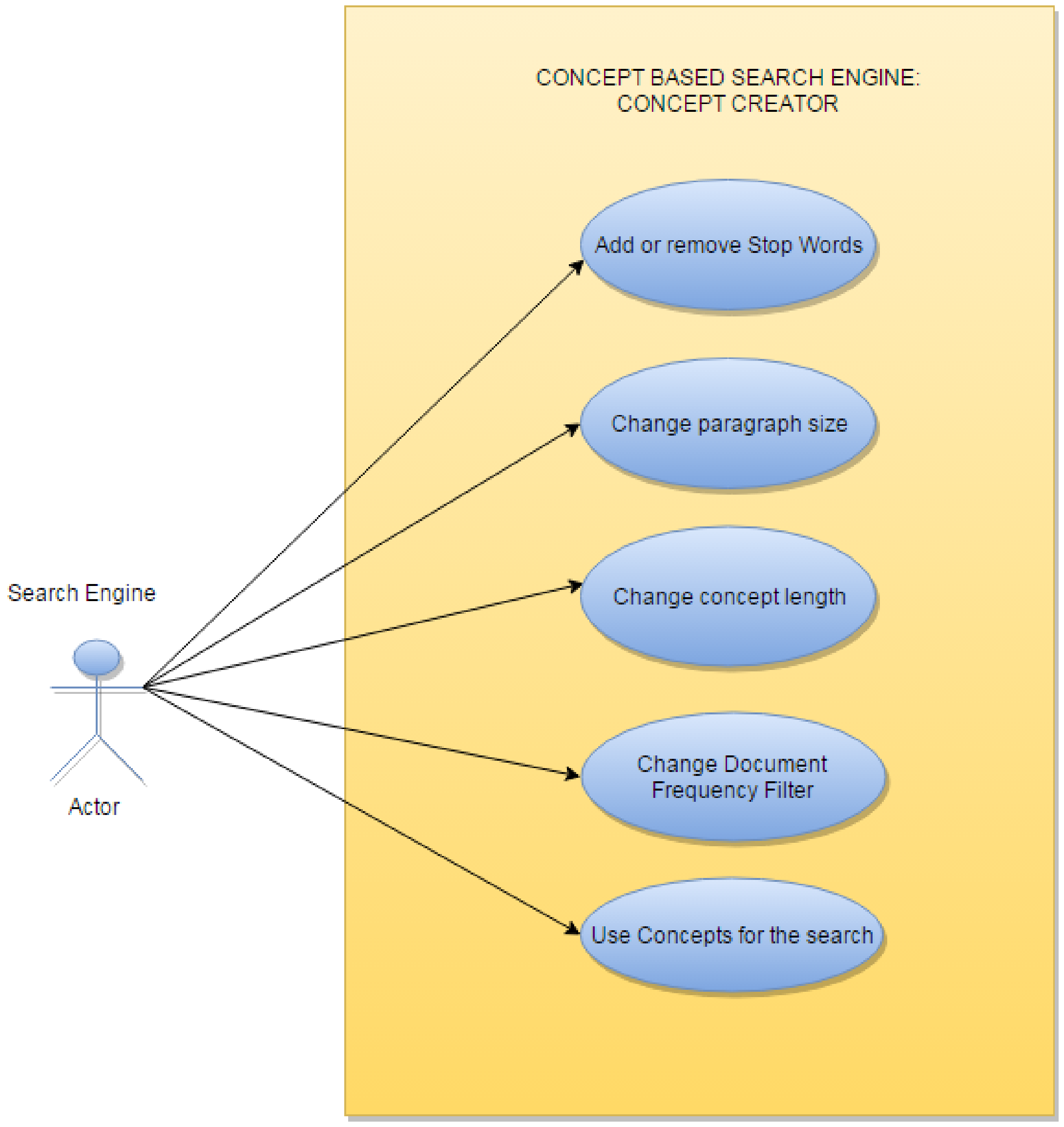


Figure 3.6: Use Case Diagram

The concept creator will be used by a search engine so the actor in this use case diagram is the search engine. A search engine can add or remove the stop words in the stop words file. It can change the paragraph size or the concept length size in the program. This will affect the speed of the code. The search engine can also change the Document frequency filter so as to improve the quality of the concepts generated. And also it uses the concepts generated by the concept creator while doing a search.

CHAPTER 4

Main Modules

In this chapter we will take a closer look on each of the modules. A quick recap of the Modules and their main operations:

The software is divided into four main modules:

- *Pre-Processing:* Raw data is pre-processed
- *Create Combinations:* Creates all possible combinations
- *Merge Combination*: Merges all the combinations into a single file
- *Filter Concepts*: Filters combinations and saves concepts

Now let’s take a look on how each of these modules work.

4.1 Data Loading and preprocessing

Definition:

Stop Words (Dictionary meaning):

> "Any of a number of very commonly used words, as a, and, in, and to, that are normally excluded by computer search engines or when compiling a concordance."

Input:

Inputs for this module are:

- *Raw Data Files*: These were the files which were directly provided to the Search Engine from the web crawler without any modifications.
- *Stop Word File*: For our project we have used a list of around 650 words found online which are considered as stop words.

Process:

This module is the first module of the Concept Creator and its main operation is to clean the data. The main function of this module is to remove all the stop words and the special characters from the text that was provided as an input.

Before reading every file which was provided as the raw data to the module, it reads and stores all the stop words in a hash map, making it a constant time lookup. It then reads every file word by word and performs these operations for every word it reads:

- Cleans the data by removing all non-word characters (that is all characters except alphabets and digits).
- Checks if the word is a stop word.
- If it's not a stop word it converts the word to lowercase and writes it to the output file.

After the complete file is read, the file is mapped with a document ID which is used later on in the project instead of using the entire file name.

Output:

- Clean data files (in lowercase and without special characters or stop words).
- Map files (Document ID: Document Name)

Below is the flowchart of the preprocessing module:

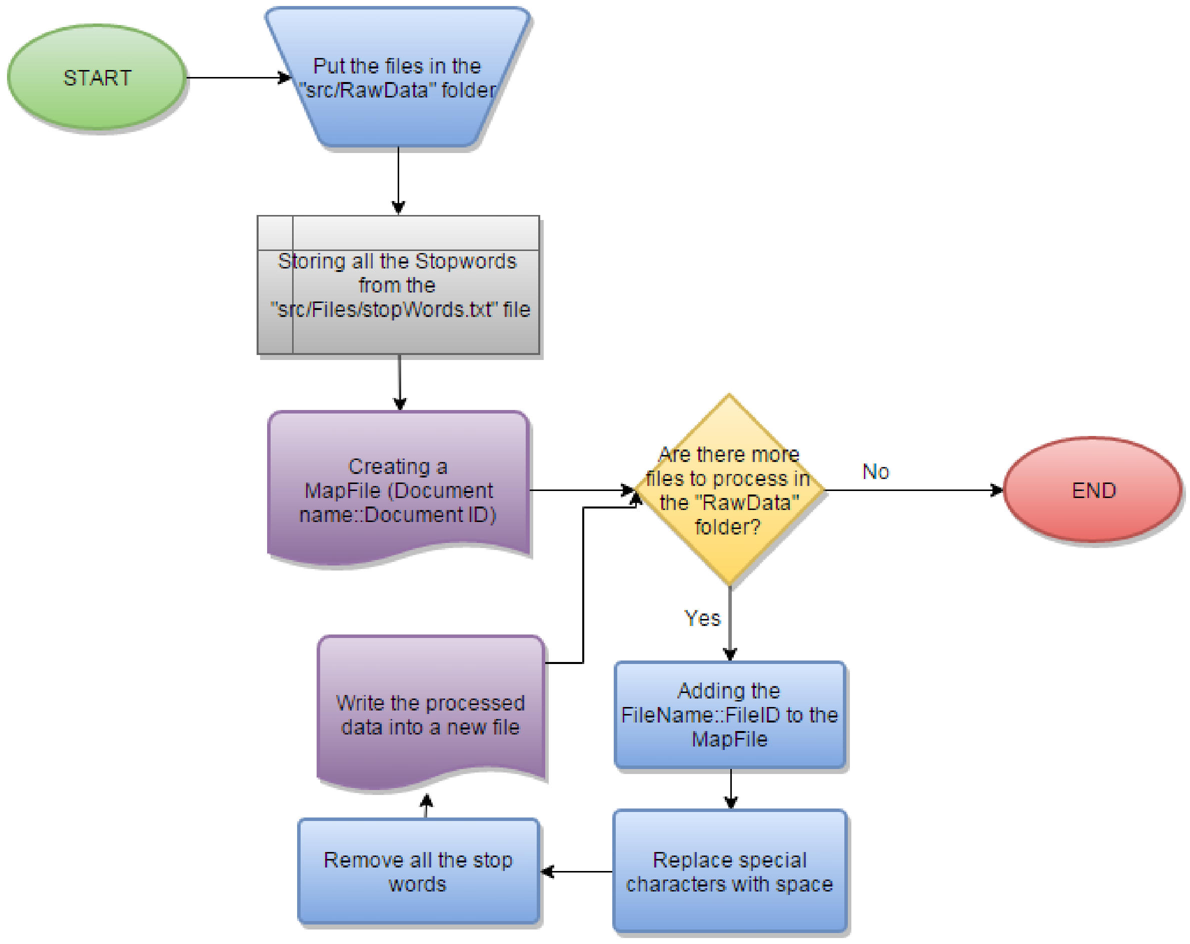

START
Put the files in the "src/RawData" folder
Storing all the Stopwords from the "src/Files/stopWords.txt" file
Creating a MapFile (Document name::Document ID)
Are there more files to process in the "RawData" folder?
No
END
Yes
Adding the FileName::FileID to the MapFile
Replace special characters with space
Remove all the stop words
Write the processed data into a new file

Figure 4.1.1: Preprocessing Raw Data

4.2 Make Combinations

Definition:

Combination (Dictionary definition):

“Joining or merging of different parts or qualities in which the component elements are individually distinct.”

Input:

Input for this module is:

- Data Files: These are the clean data files which were generated in the previous module.

Process:

This module is one of the most important modules of the project. The main function of this module is to divide a page into paragraph and calculate all the viable combinations in that paragraph.

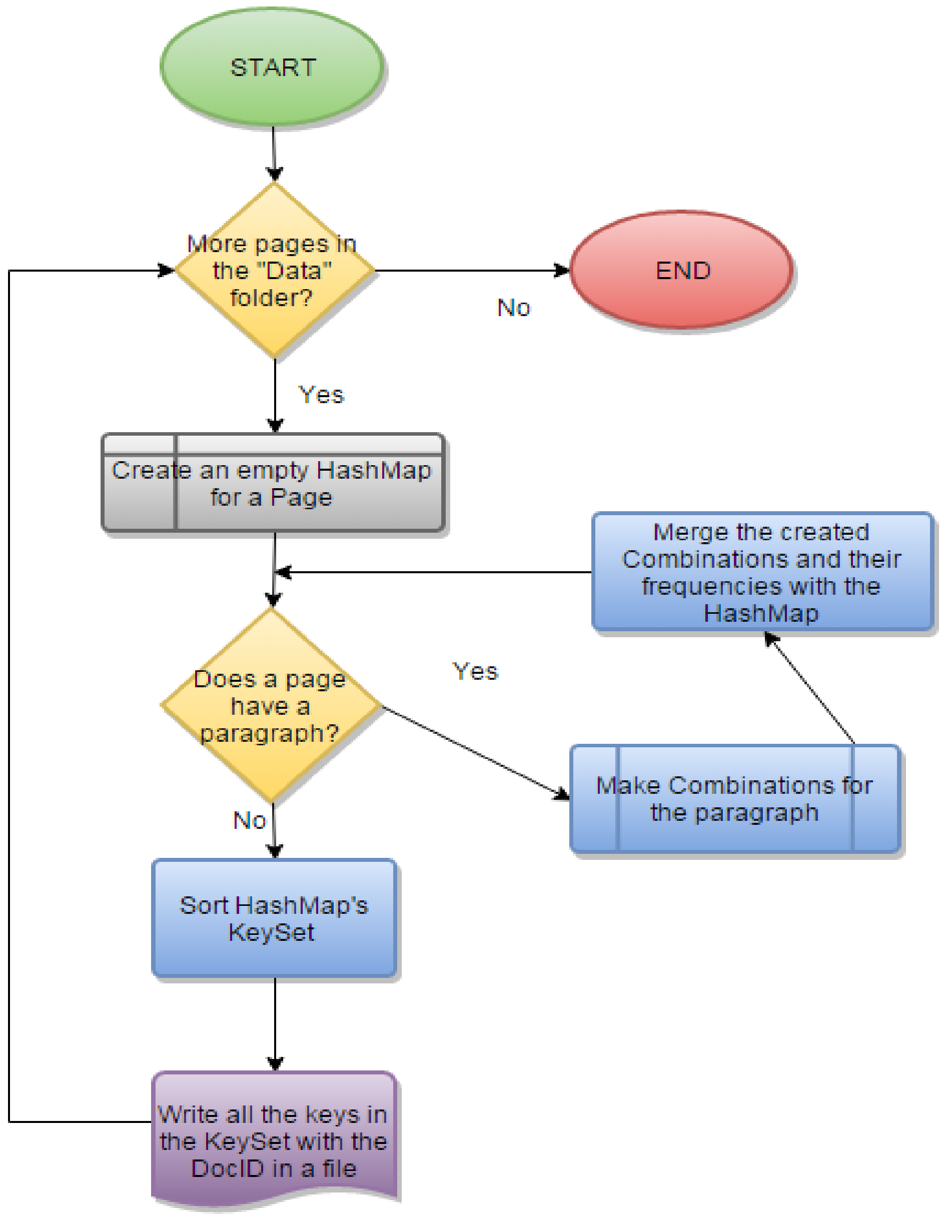
The main function of this module is the “makeCombination” method. We will discuss that method in detail later in this section. The module generates all possible combinations using this “makeCombination” method.

The module keeps on storing all the generated combinations in a hash map and then sorts it before writing it into the output file along with the Document ID.

Output:

- Combinations with Document ID for all the data files (Combination: Document ID)

Below is the flowchart for the concept generator module:



START
More pages in the "Data" folder?
No
END
Yes
Create an empty HashMap for a Page
Merge the created Combinations and their frequencies with the HashMap
Does a page have a paragraph?
Yes
Make Combinations for the paragraph
No
Sort HashMap's KeySet
Write all the keys in the KeySet with the DocID in a file

Figure 4.2.1: Creating Combinations

As we are now done with explaining the overview of the create combination module. Let look into the most important method of this algorithm. We will see how exactly the code generates and stores such a large amount of combinations for each paragraph and pages.

makeCombinations Method:

The method makeCombinations (String s) takes in a paragraph as a String and returns a Hash Map containing all the viable Combinations and their frequency (number of times they were present in the paragraph).

The input string is split into tokens and the tokens are stored in a String array. For each token in the string array we read the key set of the "combination" Hash Map, append the token to all the keys in the key set until the length of the combination is less than 6 (combination size) and then put all the appended keys in the Hash Map of the "combination". The Hash Map is then returned to the calling function.

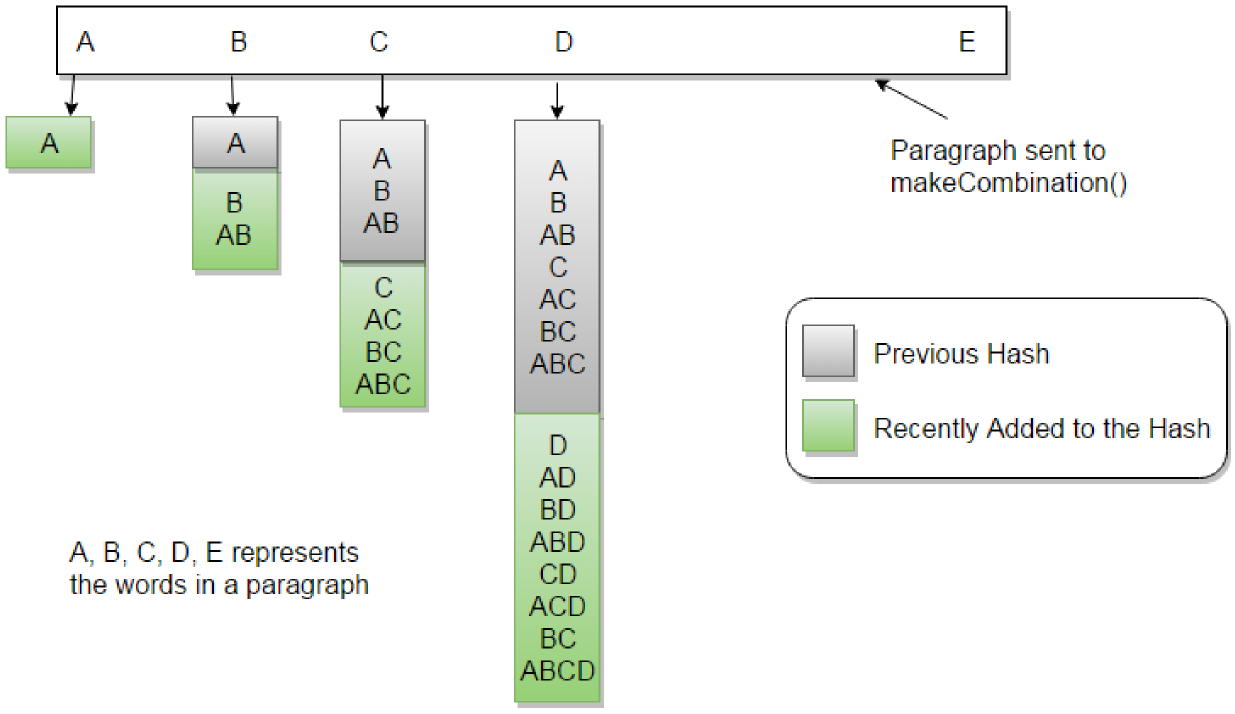

A B C D E
A
A
B AB
A B AB
C AC BC ABC
A B AB C AC BC ABC
D AD BD ABD CD ACD BC ABCD
Paragraph sent to makeCombination()
Previous Hash
Recently Added to the Hash
A, B, C, D, E represents the words in a paragraph

Figure 4.2.2: Making Combinations

The main algorithm behind this method can be explained by looking at the diagram above. The word is stored in a hash map and every time a new word is added we keep the keyset from the previous hash map and then append the new word along with the keyset in the previous hash map after appending the new word at the end of that key set.

As mentioned earlier the main function of this module is to divide the file into 30 words (paragraph size) paragraphs and then send each paragraph to the "makeCombination" method. The "makeCombination" method then returns a Hash Map that contains all the viable combinations and the frequencies in which these combinations occurred in the paragraph.

Frequency Analysis Of The Combinations Created:

When we studied the frequencies of Combinations for random files we found that more than 90% of the combinations created have a frequency 1. That means that these combinations were only created /occurred once in the whole file. Below is the graph showing the frequency distribution of the combinations created in this module:

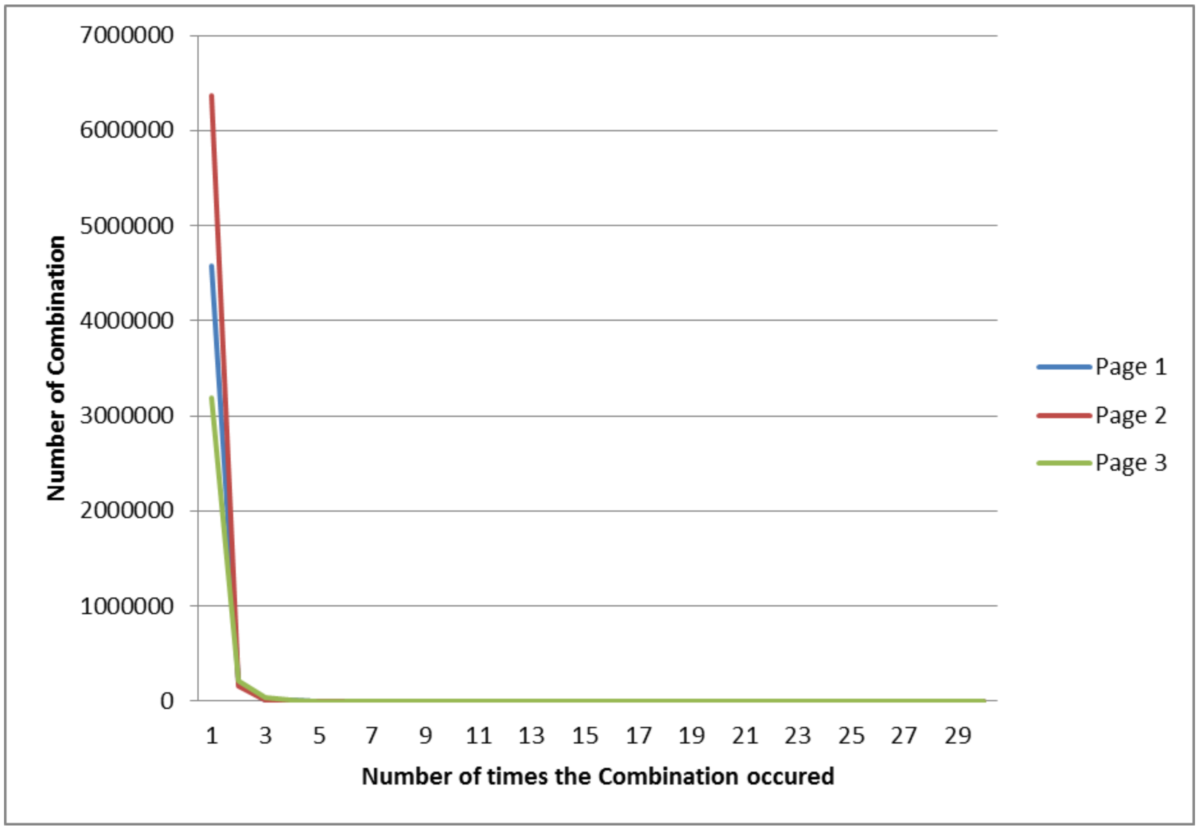


Figure 4.2.3: Frequency map

As you can see most of the combinations occurred just once in a document. This means those combinations are least likely to become a concept. Even if some of these combinations were concepts, we will still ignore it because it will not be a relevant concept to the file as a combination that would have occurred more than once in the same document.

So once the Hash Map is received, the combinations which have a frequency equal to 1 are ignored and the combinations with frequency greater than 1 are stored along with the Document ID in sorted order.

4.3 Merge Combinations

<u>Input:</u>

- Combinations with Document ID for all the data files (Combination: Document ID).

<u>Process:</u>

This module is the module that merges all the Combinations that were created in the previous module into a single document. The main function is to merge all the combinations into one single file with a list of all the Document IDs they were present in. The list of all the Document IDs which were generated in this module provides the Document frequency for all the combinations.

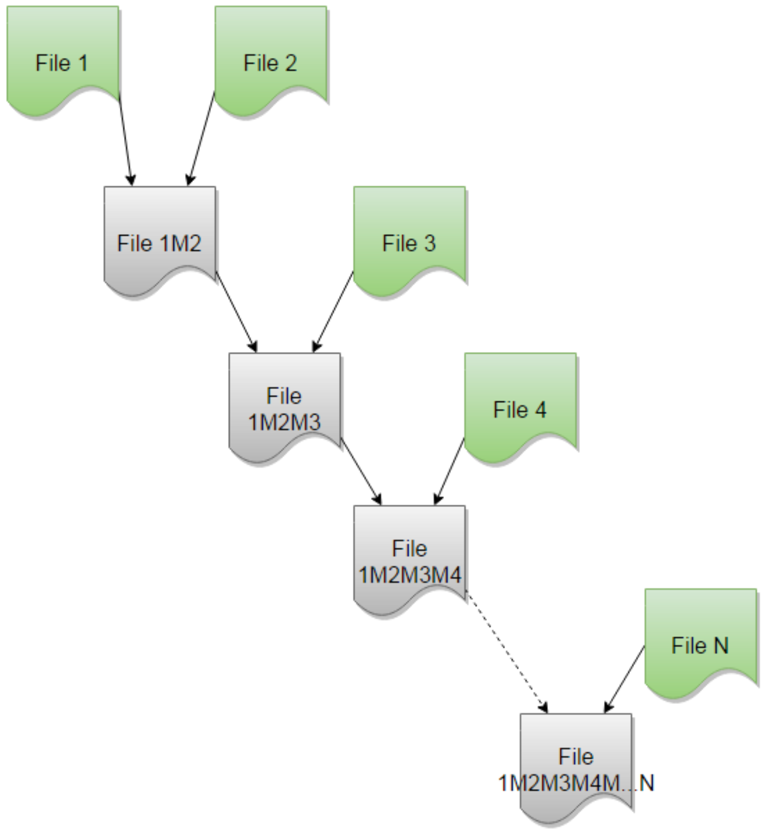
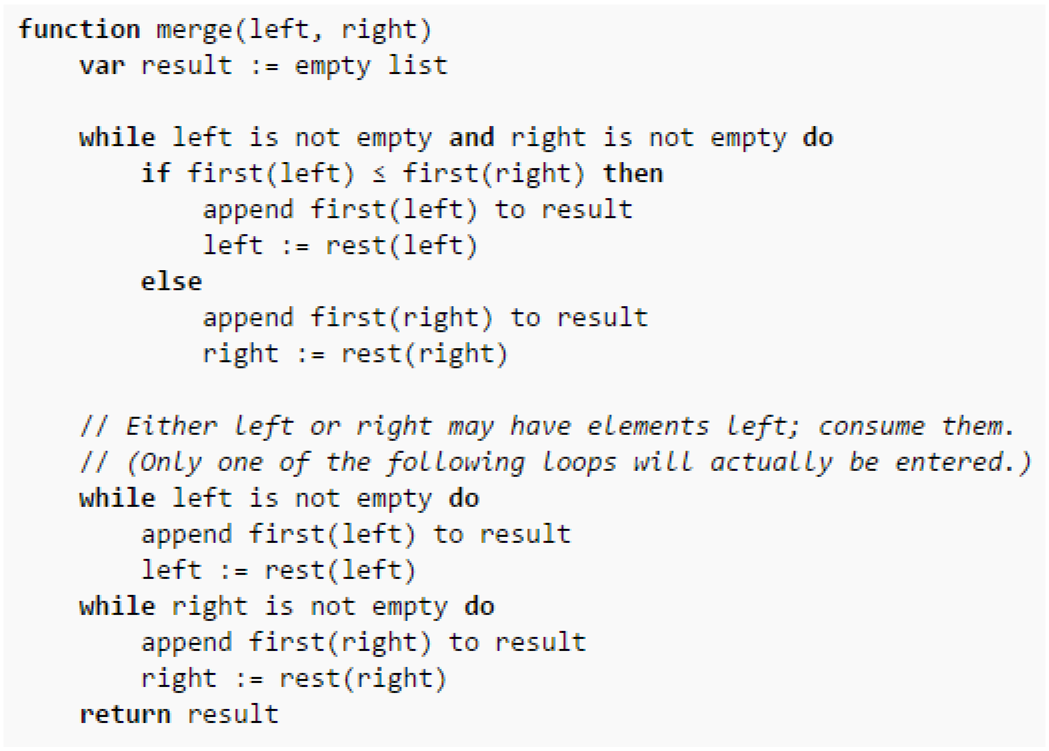


Figure 4.3.1: Merge Combinations

The files from the previous modules are provided to the “Merge” method two at a time. This Merge function is very similar to the merge method in the merge sort. This is the pseudo code form Wikipedia:

Pseudo code: (Merge function in the Merge Sort):



```
function merge(left, right)
    var result := empty list

    while left is not empty and right is not empty do
        if first(left) ≤ first(right) then
            append first(left) to result
            left := rest(left)
        else
            append first(right) to result
            right := rest(right)

    // Either left or right may have elements left; consume them.
    // (Only one of the following loops will actually be entered.)
    while left is not empty do
        append first(left) to result
        left := rest(left)
    while right is not empty do
        append first(right) to result
        right := rest(right)
    return result
```

It is possible to use a similar kind of algorithm for merging the combination into a single file as the input files are in a sorted order. The combinations are sorted in Lexicographical order.

<u>Output:</u>

- Merged file (Combination 1: Document ID 1, Document ID 2….)

Below is the flow diagram for merging the combination files together:

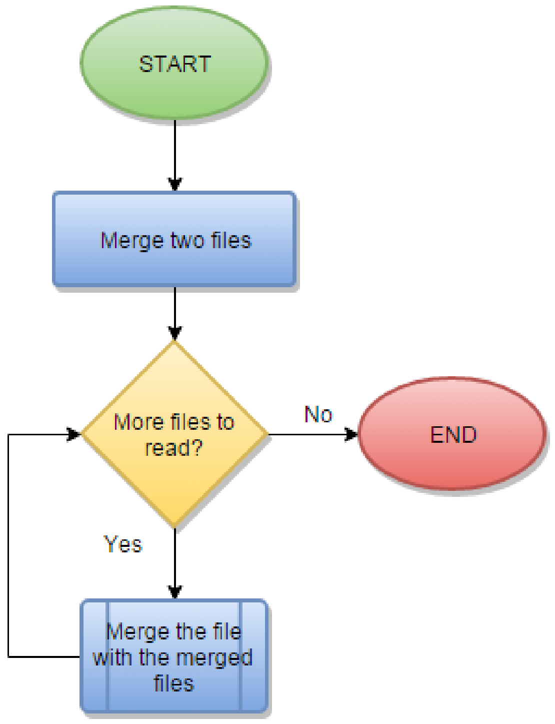


Figure 4.3.2: Merging Combinations

4.4 Filter Concepts

Input:

- Merged file (Combinations : Document ID 1, Document ID 2….)

Process:

This is the smallest module but it plays a very important role in filtering out the Concepts from the Combinations. Being the last module of the project this module provides the final filter which is needed to get the concepts.

The merged file is read line by line to check the number of Documents IDs present for every combination. This is the combination document frequency for every combination.

A filter is then applied to the combination document frequency. If the document frequency for a combination is greater than the filter frequency then the combination is considered a Concept.

Output:

- Concept File (Concept: Document ID 1, Document ID 2….)

Below is a part of the Entity Relationship Diagram that shows how a Concept is related to a combination:

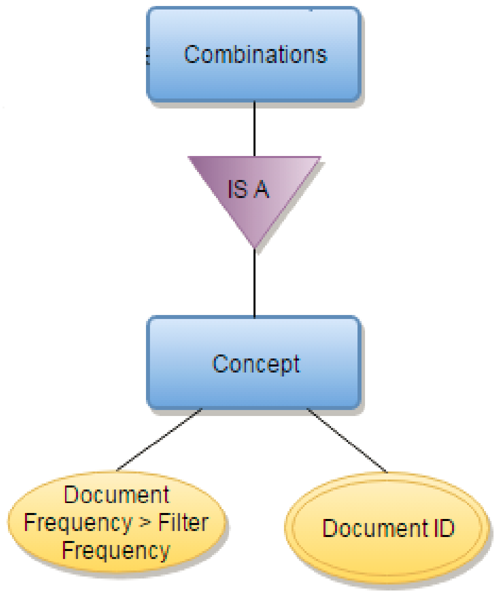


Figure 4.4.1: Filtering Concepts

You can clearly see from the above diagram that a concept is a combination and has a document frequency which is greater than the filter frequency and it is present in multiple documents.

The overall flow diagram for filtering out the concepts from the combination is given below:

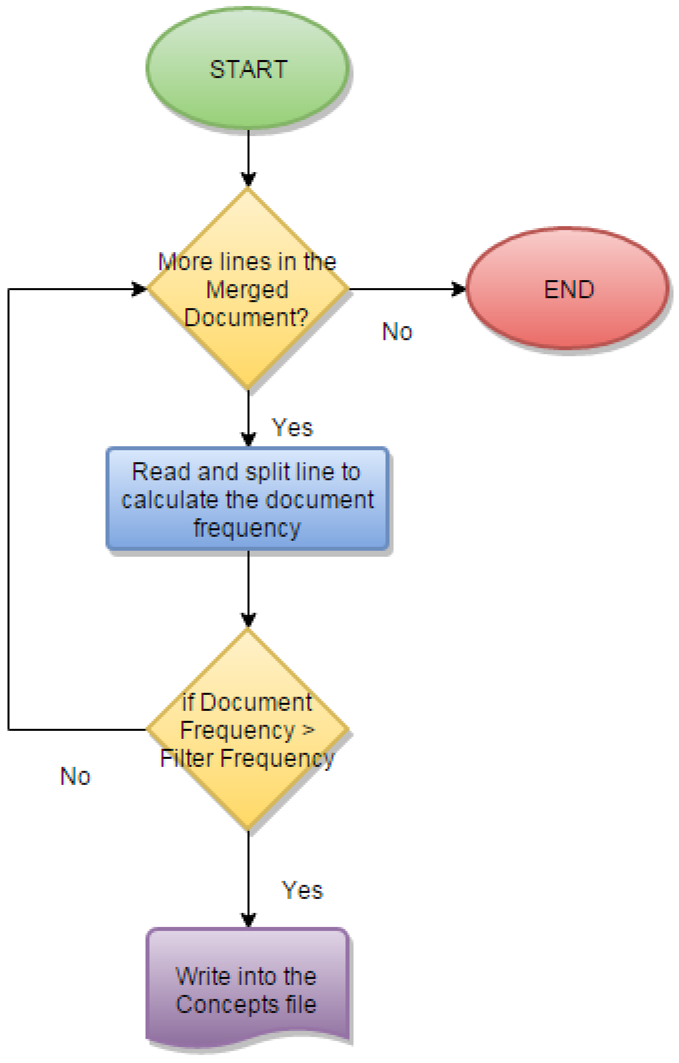

START
More lines in the Merged Document?
END
No
Yes
Read and split line to calculate the document frequency
if Document Frequency > Filter Frequency
No
Yes
Write into the Concepts file

Figure 4.4.2: Filtering Combinations

Chapter 5

Software Description

Below are the software specifications that were used while running the code for this project:

5.1 Software Specification

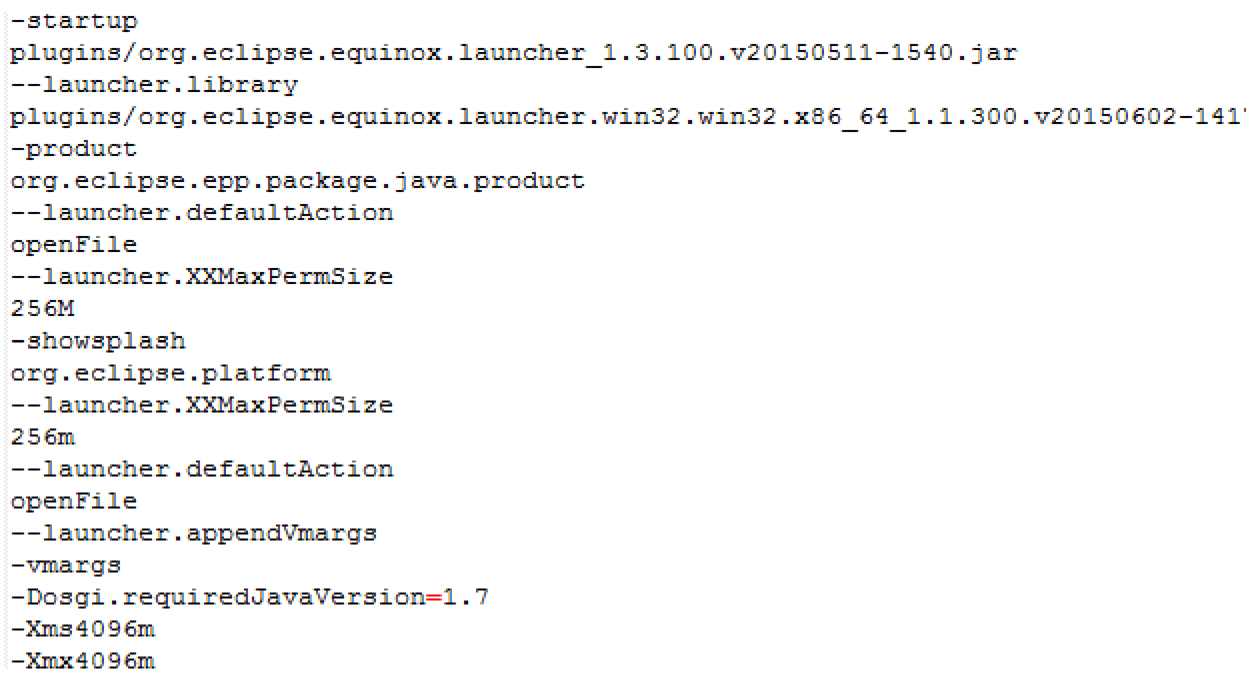
- Operating System: Windows 7
- Programming Language: Java
- IDE: Eclipse
- Data Base: File System

5.2 Hardware Specification

- CPU: IntelI CoreI i5-5300U CPU @ 2.3GHz 2.3GHz
- RAM: 12 GB

5.3 Eclipse heap size settings:

Eclipse.ini file looks something like this:



```
-startup
plugins/org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.300.v20150602-141
-product
org.eclipse.epp.package.java.product
--launcher.defaultAction
openFile
--launcher.XXMaxPermSize
256M
-showsplash
org.eclipse.platform
--launcher.XXMaxPermSize
256m
--launcher.defaultAction
openFile
--launcher.appendVmargs
-vmargs
-Dosgi.requiredJavaVersion=1.7
-Xms4096m
-Xmx4096m
```

Note: You may use a different type of environment but that will affect the run time.

Chapter 6

Experiments and Results

We mentioned that the algorithm was linear when we add more files. Here is an experiment to prove that the time taken for the algorithm is linear as we add more files to it.

We have run the program and noted the time taken by different modules. In the end it should be a linear growth in time when more pages are added to the algorithm.

Experiment:

Time taken to run 10 files:

Preprocessing.java- Run took: 217ms

MakeCombinations.java- Run took: 139107ms

MergeCombinations.java- Run took: 9458ms

FilterConcepts.java- Run took: 1266ms

Total time to generate concepts from 10 files-

Run took: 150048ms = 2.5008mins

Time taken to run 20 files:

Preprocessing.java- Run took: 364ms

MakeCombinations.java- Run took: 312354ms

MergeCombinations.java- Run took: 30842ms

FilterConcepts.java- Run took: 2643ms

Total time to generate concepts from 20 files-

Run took: 346203ms = 5.77mins

Time taken to run 30 files:

Preprocessing.java- Run took: 423ms

MakeCombinations.java- Run took: 507126ms

MergeCombinations.java- Run took: 88619ms

FilterConcepts.java- Run took: 3744ms

Total time to generate concepts from 30 files-

Run took: 599912ms = 9.99mins

Time taken to run 40 files:

Preprocessing.java- Run took: 422ms

MakeCombinations.java- Run took: 663863ms

MergeCombinations.java- Run took: 129626ms

FilterConcepts.java- Run took: 4534ms

Total time to generate concepts from 40 files-

Run took: 798445ms= 13.30mins

Time taken to run 50 files:

Preprocessing.java- Run took: 481ms

MakeCombinations.java- Run took: 704694ms

MergeCombinations.java- Run took: 206821ms

FilterConcepts.java- Run took: 5835ms

Total time to generate concepts from 40 files-

Run took: 917831ms= 15.29mins

Results:

Below is the graph showing time as the function of number of files used for creating concepts. From the graph below we can see that Pre-Processing module and the Filter Concepts modules take very less time as compared to the Make Combinations and Merge Combinations modules. This means that we can ignore the Pre-Processing and Filter Concepts. Even if we look at the data above they do have a linear time complexity.

We can see that the main modules Make Combinations and Merge Combinations which will add the most time to the total time taken by the algorithm are both following a linear trend.

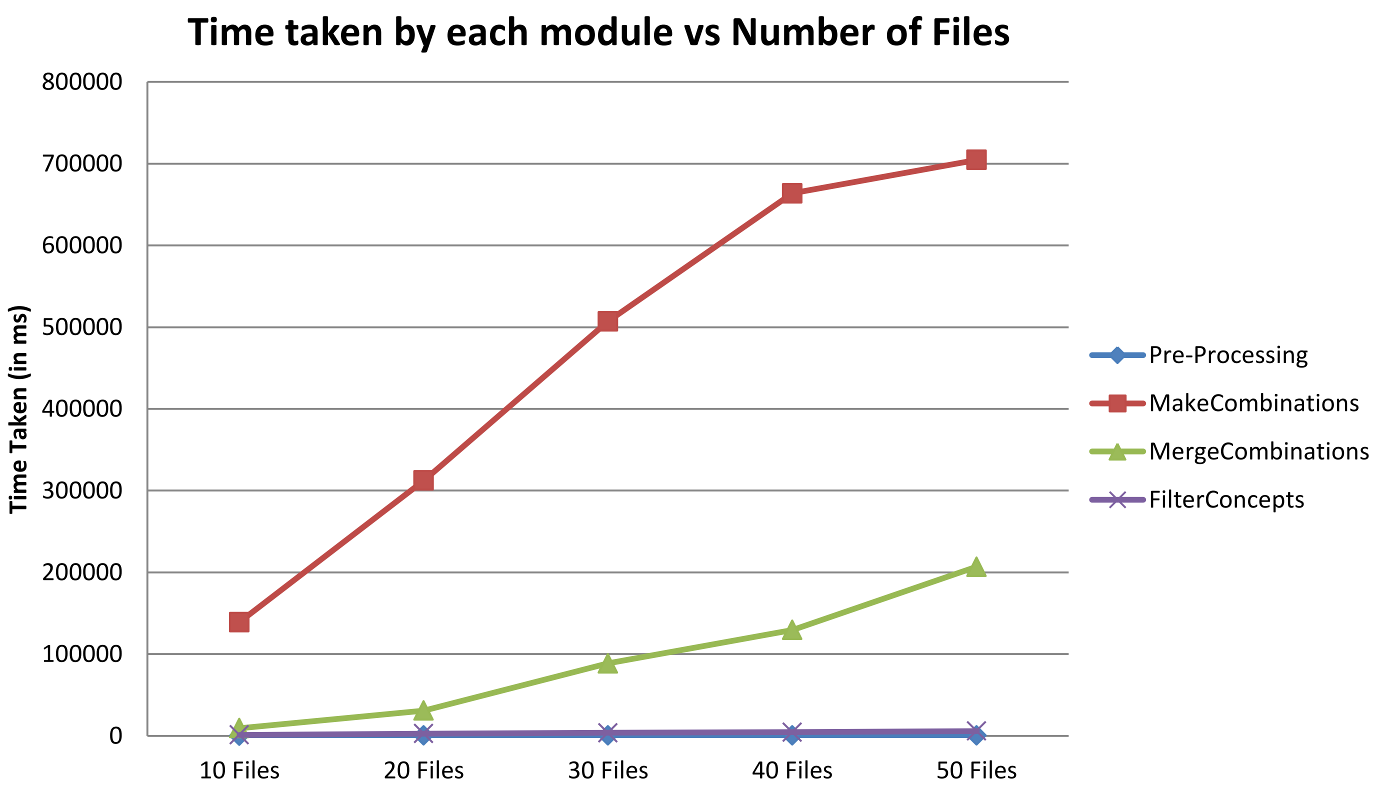


Figure 6.2: Total time taken vs Number of files Executed

Now let’s take a look at the total time taken by the algorithm if we increase the number of files. The graph below shows a linear growth in time. This proves that our algorithm has a linear time complexity with respect to the number of files added in the system.

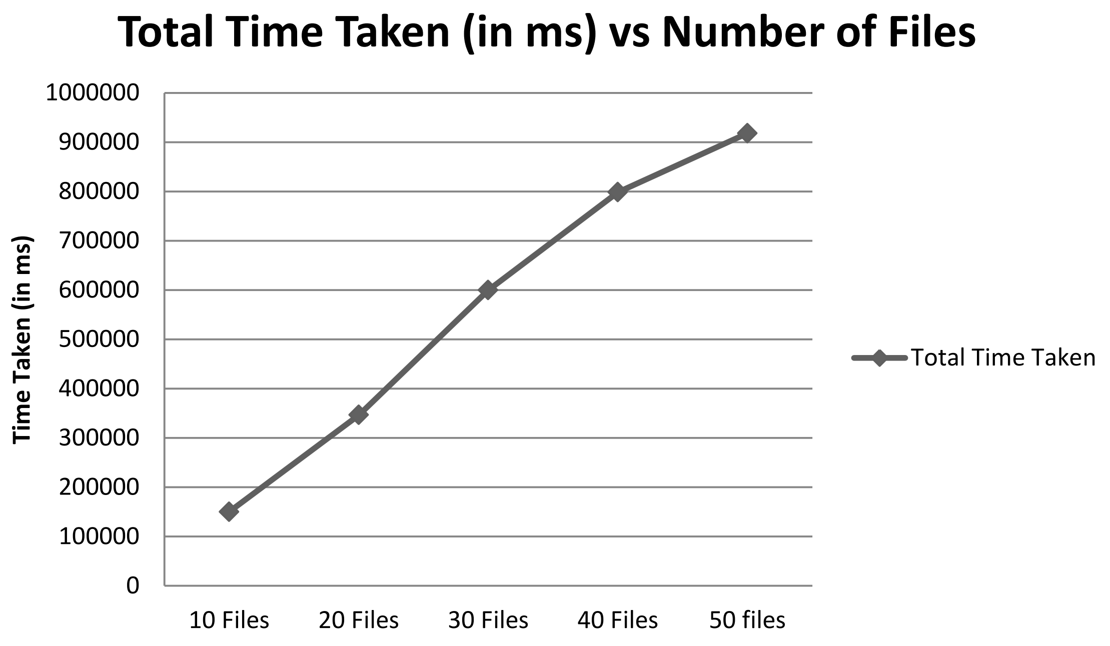


Figure 6.2: Total time taken vs Number of files Executed

Chapter 7

Conclusion

This algorithm was designed and implemented to generate concepts from a given set of documents. The algorithm is independent of the main memory constrains as it computes one document at a time. Moreover, the algorithm was built in such a way that it is scalable if the number of documents is increased. These are the two things that were achieved by using this algorithm.

Algorithm is independent of main memory constrains:

- Main memory processing is limited to a single page which makes it independent of the main memory.

Algorithm is scalable:

- The time complexity is linear which enables the program to run smoothly even if more documents are added to the set of documents as seen from the experiment done in Chapter 7.

Chapter 8

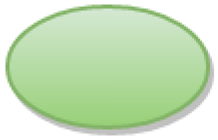
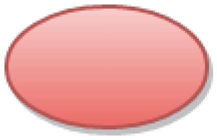
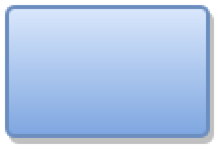
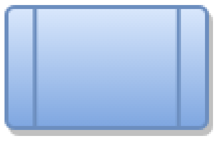
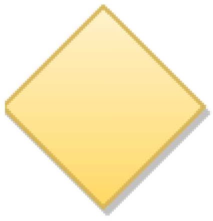
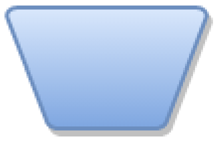
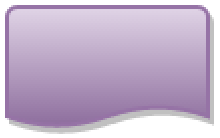
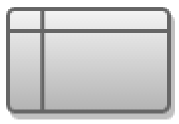
Future Work

- The fine tuning of the filter frequency needs to be done so that the concepts are effectively separated from the rest of the combinations.

- Some kind of a page ranking system can be developed so that the most important pages are shown first followed by the lower priority ones.

- This algorithm can be helpful in classification algorithms as well where the documents can be classified into topics or concepts they contain.

Appendix: Reference table for the flow charts

	Show the start point in a process. When used as a Start symbol, terminators depict a *trigger action* that sets the process flow into motion.
	Show the end point in a process.
	Show a Process or action step. This is the most common symbol in both process flowcharts and process maps.
	A Predefined Process symbol is a marker for another process step or series of process flow steps that are formally defined elsewhere. This shape commonly depicts sub-processes (or subroutines in programming flowcharts). If the sub-process is considered "known" but not actually defined in a process procedure, work instruction, or some other process flowchart or documentation, then it is best not to use this symbol since it implies a formally defined process
	When used as a Start symbol, terminators depict a *trigger action* that sets the process flow into motion.
	Manual Operations flowchart shapes show which process steps are not automated. In data processing flowcharts, this data flow shape indicates a looping operation along with a loop limit symbol
	The Document flowchart symbol is for a process step that produces a document.
	Used in programming flowcharts to mean information stored in memory, as opposed to on a file.

List of References

[1] Tsau Young (T. Y.) Lin, Albert Sutojo and Jean-David Hsu; Concept Analysis And Web Clustering using Combinatorial Topology (2006)

[2] Tsau Young (T. Y.) Lin and Jean-David Hsu; Knowledge Based Search Engine Granular Computing on the Web

[3] Apriori algorithm; http://www.cs.sunysb.edu/~cse634

[4] Introduction to Information Retrieval - By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze ; Website: http://informationretrieval.org/ ; Cambridge University Press

[5] Google Search, http://www.google.com.

[6] Roy, Pradeep, "Concept Based Semantic Search Engine" (2014).Master's Projects. Paper 351.

[7] R. Agrawal and R. Srinkat. Fast algorithms for mining association rules. Proceedings of the 20th VLDB Conference, 1994.

[8] T. Y. Lin. Granular computing: Examples, intuitions and modeling. In: the Proceedings of 2005 IEEE International Conference on Granular Computing, 2005.

[9] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983.

[10] Introduction to Information Retrieval - By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze ; Website: http://informationretrieval.org/ ; Cambridge University Press