**Siham Ouazzani**

# Massive Data Processing – Assignment 2

## I – Pre-Processing the input

One of the main issues when answering set-similarity joins using the MapReduce paradigm is to decide how data should be partitioned and replicated. Thus, pre-processing the input for the following questions is a necessary step. First, we will be using assignment 2's "stop-words" as input, the final output should be a document with lines empty from stop-words and with unique words. We will consider each line as a document for the similarity joins.
Each document:

- has no stop words
- has unique words
- Is not an empty line

The Mapper will find all words that are A-Z, a-z, 0-9, non-stop words and no empty lines.

```
public static class JoinMapper extends Mapper<Text, Text, Text, Text> {
         private Text word = new Text();

     //Upload The stop words.txt file from Assignment 1
         File StopWords = new File("/home/cloudera/Desktop/stopwords.txt");
     //Save the txt file into a Hashset
         Set<String> StopWordsSet = new HashSet<String>();//HashSets take unique words, there is no repetition

         protected void ReadStopWord(Context context) throws IOException, InterruptedException { //Reading stop words
              BufferedReader readSW = new BufferedReader(new FileReader(StopWords));
              String stopword = null; //opening the file to read stop words + initialisation to null
              while ((stopword = readSW.readLine()) != null){
                   StopWordsSet.add(stopword);
              }//Putting all stopwords within a set
              readSW.close();
         }
          @Override
          public void map(Text key, Text value, Context context) throws IOException, InterruptedException {

             List <String> Count = new ArrayList <>();
             if (value.toString().length() != 0){ //find empty lines and deletes them
                for (String str : value.toString().replaceAll("[^A-Za-z0-9]"," ").split("\\s+")){ //only leaves values with A-Z, a-z, 0-9 and replace words
                    if (!StopWordsSet.contains(str.toLowerCase())){ //If the word is not a stop word
                        if (!Count.contains(str.toLowerCase())){ //If it is not reported in the document
                            Count.add(str.replaceAll("[^A-Za-z0-9]","").toString().toLowerCase());
                            word.set(str.toString());
                            if(word.toString().length() != 0){
                                context.write(key, word);// Mapper will find all words that are A-Z, a-z, 0-9, non stop words and no empty lines
                            }
                        }
```

The Reducer takes word and frequency and prepares the output-preprocessed documents as asked for the requirements.

```
public static class JoinReducer extends Reducer<Text, Text, LongWritable, Text> {
    private HashMap<String, Integer> Docfreq = new HashMap<String, Integer>(); //Hash map will be each word and its frequency stored
    private long iter = 0; //keep track of number of documents printed (only works if using one reducer)

    //las palabras se ordenan antes de los id de documento
    @Override
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        int frequency = 0; //Frequency of tokens
        LinkedList<String> WordDoc = new LinkedList<String>(); //Each document's word is stored
        for(Text valeur : values) {
            if(key.charAt(0) != '{') { //https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/charAt
                frequency += Integer.parseInt(valeur.toString());
            }
            else { //word counts ids
                String[] MotValeur = valeur.toString().split("\\s");
                for(int i = 0; i < MotValeur.length; i++) {
                    int k = 0;
                    String mot = MotValeur[i];//
                    if(!WordDoc.contains(mot)) { //
                        int addfrequency = Docfreq.get(mot);
                        while(k < WordDoc.size() && addfrequency > Docfreq.get(WordDoc.get(k))) { //get position in set
                            k++;
                        }
                        WordDoc.add(k, mot); //add the word to the list
                    }
```

```
        if(key.charAt(0) != '{') { //put both word and frequency
            Docfreq.put(key.toString(), frequency);
        }
        else {
            String index = key.toString(); //key to string
            int index1 = index.lastIndexOf("#") + 1;
            long id = Long.parseLong(index.substring(index1));
            String output = "";
            for(String s : WordDoc) {
                output = output + s + " ";
            }
            if(iter == 0) { //
                iter = id + 1;
            }
            context.write(new LongWritable(iter), new Text(output)); //write in output the counter for word frequency and the word
            iter++; //increment and loop back
        }
    }
 }//Reducer takes word and frequency and prepare the output preprocessed documents as asked for the requirements.
'El Fin
```

Record Lines number is ‘114461’

```
    try {
            Long RecordNum = job.getCounters().findCounter("org.apache.hadoop.mapred.Task.Counter", "Record").getValue();
            BufferedWriter writer = new BufferedWriter(new FileWriter("/home/cloudera/Desktop/Record.txt"));
            writer.write(String.valueOf(RecordNum));
            writer.close();
        } catch(Exception e) {
            throw new Exception(e);
        }
```

Output found in Output: OutputPreProcessing/PreProcessingOutput.txt

```
1       EBook Complete Works Gutenberg Shakespeare William Project by
2       Shakespeare William
3       anyone anywhere eBook cost use This at no
4       restrictions whatsoever copy almost away give may You or no
5       reuse included License Gutenberg Project terms under
6       wwwgutenbergorg online eBook or at
7       Details COPYRIGHTED Below eBook Gutenberg Project This
8       guidelines copyright file Please follow
9       Title Complete Works Shakespeare William
10      Author Shakespeare William
11      September Posting 2011 100 Date EBook 1
12      1994 Release Date January
13      Language English
14      EBOOK WORKSWILLIAM START GUTENBERG COMPLETE SHAKESPEARE PROJECT THIS OF
15      Produced Future Inc World Library their from by
16      100th Etext presented file Gutenberg Project This by
17      cooperation Inc World Library presented their from
18      CDROMS Future Library Gutenberg Shakespeare Project
19      Etexts releases Public Domain placed often NOT are
20      Shakespeare
21      implications Etext copyright certain read has This should
22      VERSION WORKS COMPLETE THIS WILLIAM THE ELECTRONIC OF
23      COPYRIGHT 19901993 WORLD INC LIBRARY SHAKESPEARE IS BY AND
24      ILLINOIS COLLEGE BENEDICTINE PROVIDED ETEXT GUTENBERG PROJECT BY OF
25      PERMISSION READABLE MACHINE WITH BE MAY COPIES ELECTRONIC AND
26      YOUR LONG SUCH AS SO OTHERS 1 DISTRIBUTED COPIES ARE FOR OR
27      PERSONAL ONLY USED USE NOT 2 DISTRIBUTED ARE AND OR
28      COMMERCIALLY DISTRIBUTION COMMERCIAL PROHIBITED INCLUDES ANY BY
29      MEMBERSHIP DOWNLOAD SERVICE CHARGES THAT TIME FOR OR
30      cooperate World Library Gutenberg Project proud
```

## II – Set Similarity Joins

### a- Naïve Approach

The naïve approach is to perform all pairwise comparisons between documents and then output only the pairs that are similar. The code has a mapper reducer and similarity function within the later. Pre-proccessing file displays all lines with unique words and no stop words. It will be used as an input. Additional information is that threshold for similarity comparison is 0.8.

PS: Due to the excessive runtimes, only the first 100 lines of the output pre-processed were inputted as a sample to test the code. Then the number will increase to have a decently sized output for similar sets.

- the mapper generates a value and key for each document id. The key is the number of both documents. We extract their line's text, which will later be processed by the reducer.

```
public static class JoinMapper extends Mapper<LongWritable, Text, Text, Text>{
        private Text keyout = new Text();
        private Text line = new Text();
        public Long ID = 100L;
        public void map(LongWritable key, Text value, Context context
                ) throws IOException, InterruptedException {
            Long documentID = Long.parseLong(value.toString().split("\t")[0]);
            line.set(value.toString().split("\t")[1]);
            for (Long id = 1L; id < documentID; id = id + 1L) {
                keyout.set(Long.toString(id)+"/"+Long.toString(documentID));
                context.write(keyout, line);
            }
            for (Long id = documentID + 1L; id < ID + 1L; id = id + 1L) {
                keyout.set(Long.toString(documentID)+"/"+ Long.toString(id));
                context.write(keyout, line);
```

- The reducer reads and iterates through all the lines of the input. The function similarity is implemented to compute the pair wise similarity.

```
public static class JoinReducer extends Reducer<Text,Text,Text,FloatWritable> {

        public static Float Threshold = 0.8f;
        private FloatWritable output = new FloatWritable();
        public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {

                String line1 = values.iterator().next().toString();
                String line2 = values.iterator().next().toString();
                Float similarity = Similarity(line1, line2); //Similarity is a function defined to compute the jaccard simil
                    if (similarity > Threshold) {
                        output.set(similarity);
                        context.write(key, output);
                    }
        }
```

Lines are stores in hashsets, intersection and union are implemented similarity to compute the jaccard similarity that is then compared to the 0.8 threshold.

```
Set<String> union = new HashSet<String>(hashline1);
Set<String> intersection = new HashSet<String>(hashline2);
```

Disclaimer: the reason I used 1000 lines is to avoid having my VM crash. Installing the virtual machine was time consuming and I will not be able to redo it properly in such time constraint.
I initially started with 100 lines to see if the code was working and finally set 1000 documents as a limit, it shows that the code is working properly by displaying a good amount of line similarity outputs.

Output for 100 lines:

```
22|38  1.0
23|39  0.8888889
```

Output for 1000 lines:

```
22|122  1.0
22|38   1.0
23|123  1.0
23|39   0.8888889
24|124  1.0
25|125  1.0
26|126  1.0
27|127  1.0
28|128  1.0
29|129  1.0
38|122  1.0
39|123  0.8888889
```

Number of comparison for 100 lines:

```
Assignment2.SetSimilarityA$JoinReducer$CountersEnum
        COMPARISONS=4950
```

Number of comparison for 1000 lines:

```
Assignment2.SetSimilarityA$JoinReducer$CountersEnum
        COMPARISONS=499500
```

The run times for Naïve Approach Similarity for a 100 documents input is 32 s
The run times for Naïve Approach Similarity for a 1000 documents input is 48 s

## b- Prefix Filtering Approach

Prefix filtering, which is based on the pigeonhole principle and works as follows: the tokens of strings are ordered based on a global token ordering. For each string, we define its prefix of length n as the first n tokens of the ordered set of tokens. The required length of the prefix depends on the size of the token set, the similarity function, and the similarity threshold. (http://event.cwi.nl/SIGMOD-RWE/2010/17-8cbba9/paper.pdf)

The prefix filtering principle states that similar strings need to share at least one common token in their prefixes. Using this principle, records of one relation are organized based on the tokens in their prefixes. Then, using the prefix tokens of the records in the second relation, we can probe the first relation and generate candidate pairs. The prefix filtering principle gives a necessary condition for similar records, so the generated candidate pairs need to be verified. A good performance can be achieved when the global token ordering corresponds to their increasing token-frequency order, since fewer candidate pairs will be generated.

Mapper creates inverted index for the first word of each document
To define this index: |d| - [ t • |d| ] + 1

```
public static class JoinMapper extends Mapper<LongWritable, Text, Text, Text> {
      private Text index = new Text();

public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
      String[] doc = value.toString().split("\t")[1].split(" ");
      //|d| - [t •|d|] + 1
      int prefixindex = ((Double) (doc.length - Math.ceil(Threshold * doc.length) + 1)).intValue();//index of words
          for (int i = 0; i < prefixindex; i += 1) {
              index.set(doc[i]);
              context.write(index, value);//out is the key of the words and the documents
          }
   }
}
```

The reducer takes both the key and the value and computes similarities of document pairs. It outputs only the similar pairs as well as the value of their jaccard similarity.

```
public static class JoinReducer extends Reducer<Text,Text,Text,FloatWritable> {
      private Text index = new Text();
      private FloatWritable output = new FloatWritable();
public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
      List<Long> id = new ArrayList<Long>();
      List<String> lines = new ArrayList<String>();
      //the reducer will iterate through the entire document of the pre-processed input file
          for (Text out : values) {
              id.add(Long.parseLong(out.toString().split("\t")[0]));
              lines.add(out.toString().split("\t")[1]);
          }
          //It performs pair wise similarities
          for (int i = 0; i < id.size(); i += 1) {
              Long id1 = id.get(i);
              String line1 = lines.get(i);
                  for (int j = i + 1; j < id.size(); j += 1) {
                      Long id2 = id.get(j);
                      String line2 = lines.get(j);
                      Float similar = Similarity(line1, line2);
                          if (similar> Threshold) { //comparing similairty to threshold
                              String idset = (id1 < id2) ? Long.toString(id1) + "|" + Long.toString(id2) : Long.toString(id2)
                              index.set(idset);
                              output.set(similar);
                              context.write(index, output);
                          }
```

Output for 100 documents:

```
23|39   0.8888889
22|38   1.0
```

Output for 1000 documents:

```
23|39   0.8888889
23|123  1.0
39|123  0.8888889
24|124  1.0
28|128  1.0
29|129  1.0
26|126  1.0
27|127  1.0
25|125  1.0
22|122  1.0
38|122  1.0
22|38   1.0
```

Number of comparison for 100 documents:

```
Assignment2.SetSimilarityB$JoinReducer$CountersEnum
        COMPARISONS=33
```

Number of comparison for 1000 documents:

```
Assignment2.SetSimilarityB$JoinReducer$CountersEnum
        COMPARISONS=576
```

The run times for Prefix Filtering Approach Similarity for a 100 documents input is 18s.
The run times for Prefix Filtering Approach Similarity for a 1000 documents input is 21s.

***C- Explain and justify the difference between a) and b) in the number of performed comparisons, as well as their difference in execution time***

The second approach is faster than the first one (48s vs 21s) because it performs less comparison. The naïve approach performs all pair wise comparison between documents and filters the similarities. On the other hand, The prefix filtering approach states that similar strings need to share at least one common token in their prefixes, thus filters the rare words in documents and performs less comparison in a lesser time.