Massive Data Processing
Assignment 2

Objectives of the Assignment:

You are asked to efficiently identify all pairs of documents (d1, d2) that are similar (sim(d1, d2) >= t), given a similarity function sim and a similarity threshold t. Specifically, assume that:

- each output line of the pre-processing job is a unique document (line number is the document id),
- documents are represented as sets of words,
- sim(d1, d2) = Jaccard(d1, d2) = |d1∩ d2| / |d1 U d2|,
- t = 0.8

0 - Input Pre-Processing:

In this assignment, you will use the document corpus of pg100.txt, as in your previous assignments, assuming that each line represents a distinct document (treat the line number as a document id). Implement a pre-processing job in which you will:

- Remove all StopWords (you can use the StopWords file of your previous assignment), special characters (keep only [a-z],[A-Z] and [0-9]) and keep each unique word only once per line. Don't keep empty lines
- Store on HDFS the number of output records (i.e., total lines)}
- Order the tokens of each line in ascending order of global frequency.

We first need to use our stopwords file from the previous assignment. However, since we only use one text file, I considered that in this case stopwords are words with a frequency of 2500 or more:

```
public static class Reduce extends // Reducer code
        Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException {
        int sum = 0; // count for each word
        for (IntWritable val : values) {
            sum += val.get();
        }
        if (sum > 2500) { // condition for a stop word
            context.write(key, new IntWritable(sum)); // we write the stop word with its count
        }
    }
}
```

After that we need to preprocess the lines of pg100.txt not to keep stopwords:

```
public static class Map extends Mapper<LongWritable, Text, LongWritable, Text> {
     private Text word = new Text();
    Set<String> StopWords = new HashSet<String>();

    protected void setup(Context context) throws IOException, InterruptedException {

        BufferedReader reader = new BufferedReader(new FileReader(new File("stopwords2.txt")));
        String stopword;
        while ((stopword = reader.readLine()) != null){
           StopWords.add(stopword);
        }
        reader.close();
    }
```

Here we read all lines from the stopwords file into a HashSet. Next, we want to delete word duplicates and stopwords from the lines we read. Note that we only keep [a-z],[A-Z] and [0-9] by using the replaceAll method:

```
@Override
public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

  List <String> words= new ArrayList <>();
      for (String token: value.toString().replaceAll("[^A-Za-z0-9]"," ").split("\\s+")){// only keep alphanumeric characters and split by space (or mulstiple spaces)
          if (!StopWords.contains(token) && (!words.contains(token)) && (token.length() != 0)){ //here we avoid empty lines and we don't consider the stopwords
                  words.add(token.replaceAll("[^A-Za-z0-9]",""));
                  word.set(token.toString());
                  if(word.toString().length() != 0){ //again we make sure not to write empty words
                      context.write(key, word);
          }
      }
  }
```

Note: Here the mapper output key is the word number and the value is the word.

Next, we want to order the words by frequency, we will use the wordcount from the previous assignment in order to have access to the frequency of each word.

Note: since the total number of reducer output is printed in the console, it is easy to report this number.

1 – Set-Similarity Join:

a) Naïve Approach to set-similarity:

The issue with the naïve approach lies in the number of comparisons performed. This not only requires polynomial time to run but also the space complexity is very high. In order to run the naïve approach on the whole preprocessed file, several gigabytes of hard drive are needed.

Due to the complexity associated with the naïve approach we choose a sample of the preprocessed file, we only take the first 100 line. Before that I tried running the code on the whole input file, this resulted in rendering my virtual machine completely unusable as all the diskspace got filled during the execution time (of several hours). Unsurprisingly, the code requires about 30 seconds to run since 100 lines is quite short.

In order to perform the naïve approach the mapper class will emit as key the document id, the value will be the text corresponding to the id. The mapper will provide with all combinations of possible pairs.

```
for (Long id = 1L; id < Id; id++) {
    Key.set(Long.toString(id) + "|" + Long.toString(Id));
    context.write(Key, Value);
}

for (Long id = Id + 1L; id < 100 + 1L; id++) {//change here to select fewer lines
    Key.set(Long.toString(Id) + "|" + Long.toString(id));
    context.write(Key, Value);
}
```

The reducer will compute the similarity between the ids received. The reducer receives a key with two id documents: id1|id2 and the corresponding text values.

After retrieving the corresponding two texts in two different sets: set1 and set2, we compute the similarity between the two ids as follows

```
Set<String> Set1 = new HashSet<String>(Arrays.asList(doc1.split("\\s+")));
Set<String> Set2 = new HashSet<String>(Arrays.asList(doc2.split("\\s+")));
Set<String> union = new HashSet<String>(Set1);
Set<String> intersection = new HashSet<String>(Set2);
union.addAll(Set2);
intersection.retainAll(Set1);

Float similarity = new Float(intersection.size()) / (new Float(union.size()));

if (similarity > threshold) {
      Value.set(similarity);
    context.write(key, Value);
}
```

Here's the screenshot from Yarn showing the execution time for only 100 lines:

The number of comparisons is 100(99)/2 = 4950 and can also be show in the console output from the counter nb_comparisons that is incremented in the reducer class.

b)

c) Using the inverted index approach to set similarity joins results in huge performance gains compared to the naïve approach. As the number of comparisons drastically decreases, so does the time to run the code. Running the prefix code on the whole input took 35 seconds, while the naïve approach would have taken several hours.

Output of the job in Yarn: