BDPA assignment 2: Set-similarity joins

Abdelhamid EZZERG

March 17, 2017

1 Introduction:

The objective of this assignment is determining pairs of documents that are similar. To accomplish this task, we will be using the document corpus that we used in the previous assignment. Namely, we will be using the pg100.txt document that we previously used which can be downloaded from http://www.gutenberg.org/cache/epub/100/pg100.txt. In this assignment, we will consider lines as separate documents. Before going straight to determining documents' similarity, we will need to preprocess the data first. At a second time, we will proceed to the main task of the assignment.

2 Data pre-processing:

As mentioned above, we need to pre-process the data before going any further. The processing should involve, as requested by the assignment questions, the following steps:

- Removing stop words (we used the results of the previous assignment to realize this task) and removing special characters.
- Removing empty lines.
- Storing on HDFS the number of total lines.
- Ordering the tokens of each line according to their global frequency. The chosen order is an ascending one.
- Storing the remaining lines on HDFS in text format.

The rest of this section is a walk through of the main steps used to realize the pre-processing task:

2.1 (optional) reducing the size of the input file:

Actually, running the code for the entire document corpus can be quite time-consuming. For development, test and debugging reasons, we chose to work using a small sample of the document corpus in order to run the code. In this manner, we will be able to assess more quickly if the algorithm is diverging from what it is supposed to do. Furthermore, we will see in the final section that we can assess the performance of the two explored approaches even for small samples.
The sample file was called pg100-head.txt and can be checked using this link: https://github.com/EZZERG/BDPA_Assign2_AEZZERG/blob/master/input/pg100-head.txt. This sample file contains the first 200 lines of the original document corpus.

2.2 Removing the stop words:

Stop words are words that are commonly used in every text and thus they have high appearance's frequency. In natural language processing and other tasks, it is common practice to remove these words since they don't enable us to discriminate between sentences.
As a part of the pre-processing process, we will proceed to removing the stop words from the

document corpus as a first step. To achieve this goal, we will use the same stop words that we spotted during the previous assignment. The list of stop words used for this assignment can be checked through this link: https://github.com/EZZERG/BDPA_Assign2_AEZZERG/blob/master/input/StopWords_main.txt In order to remove the stop words, two approaches are available: first, we can try to load directly the stop words file into the java code as was done in the previous assignment or we can load the file onto the HDFS so that we can have access to it from the command prompt using some added parameters. In this assignment, we have chosen the second approach. Using this solution, we only need to add the -skip argument to the hadoop jar command. The new command will have the following form: "ancient commend" + "-skip path to stopwords on the HDFS".
Handling the -skip argument requires some extra changes to the code. These changes can be summarized as follows:

- We needed to add some parametrization to the driver using the following code:

```
for (int i = 0; i < args.length; i += 1) {
    if ("-skip".equals(args[i])) {
        job.getConfiguration().setBoolean(
                "Preprocessing.skip.patterns", true);
        i += 1;
        job.addCacheFile(new Path(args[i]).toUri());
        LOG.info("Added␣file␣to␣the␣distributed␣cache:␣" + args[i]);
    }
}
```

- Moreover, we needed to add two other methods to the mapper: the setup method to load the stop words file and the parseSkipFile method in order to parse it:

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

    protected void setup(Mapper.Context context) throws IOException,
            InterruptedException {
        if (context.getInputSplit() instanceof FileSplit) {
            ((FileSplit) context.getInputSplit()).getPath().toString();
        } else {
            context.getInputSplit().toString();
        }
        Configuration config = context.getConfiguration();
        if (config.getBoolean("Preprocessing.skip.patterns", false)) {
            URI[] localPaths = context.getCacheFiles();
            parseSkipFile(localPaths[0]);
        }
    }

    private void parseSkipFile(URI patternsURI) {
        LOG.info("Added␣file␣to␣the␣distributed␣cache:␣" + patternsURI);
        try {
            fis = new BufferedReader(new FileReader(new File(
                    patternsURI.getPath()).getName()));
            String pattern;
            while ((pattern = fis.readLine()) != null) {
                patternsToSkip.add(pattern);
            }
        } catch (IOException ioe) {
            System.err
                    .println("Caught␣exception␣while␣parsing
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣the␣cached␣file␣'"
```

```
                                + patternsURI
                                + "': "
                                + StringUtils.stringifyException(ioe));
        }
    }
```

- It is now possible ti use an if loop in order to skip stop words since we have a stop words list.

2.3 Removing special characters:

In order to remove special characters, we can add a regex pattern to the mapper. In fact, we used in the code a patter Pattern p = Pattern.compile("[A-Za-z0-9]") that the words have to match in order to be retained by the algorithm. This can be easily done by introducing the fllowing conditions: p.matcher(word.toLowerCase()).find() and wrd.toLowerCase().isEmpty() in a if loop so that we are sure that empty words and special characters are correctly removed. This can be rapidely tested using the header of the pg100.txt. (We used a sample file to run the code quickly).

2.4 Removing empty line:

This can be done in the same fashion as before (and it is even more simpler): in the if loop, we add another condition: alue.toString().length() == 0. Since empty lines can be represented as list whose length is 0 of words or, in other words, an empty list of words.

2.5 Keeping a unique copy of each word per line:

The pre-processing job should include words ordered by their global frequency without redundance. This last criterion is what this step is used for. In fact in this step, we try to store the words that appear in each line but without repeating them. This can be easily done using any type of list that does not accept duplicates namely a HashSet. This representation of data was used within the reducer. Here follows the relative lines of code:

```
ArrayList<String> wordsL = new ArrayList<String>();

for (Text word : values) {
    wordsL.add(word.toString());
}

HashSet<String> wordsHS = new HashSet<String>(wordsL);
```

2.6 Storing the number of output records on the HDFS:

In this step, we try to add the feature of storing/ counting the number of output records. This information can be preserved by using a counter in our code that will be incremented as the algorithm progresses. Here is how we defined the counter in the code:

```
public static enum CUSTOM_COUNTER {
    NB_LINES,
};
```

As requested by the assignment, the value of this counter should be stored on the HDFS. To do so, we added the following method to the driver:

```
job.waitForCompletion(true);
long counter = job.getCounters().findCounter(CUSTOM_COUNTER.NB_LINES)
        .getValue();
Path outFile = new Path("NB_LINES.txt");
BufferedWriter br = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
br.write(String.valueOf(counter));
br.close();
return 0;
```

Moreover, we incremented the counter at the end of the reducer using the following line:

```
context.getCounter(CUSTOM_COUNTER.NB_LINES).increment(1);.
```

And finally, it is worth mentioning that we obtained 149 as the counter's value for the text sample.

2.7 Adding the global frequency to each word:

It is possible to try and compute the global frequency at the same time we are pre-processing the data but it seems that this approach is slow and inefficient. Instead of trying to execute the two tasks in parallel, it seems chaining two MapReduce procedures is a good idea. In fact this is what we will do in this step.
At a first time, we used a word count algorithm in order to estimate the global frequency. The WordCount algorithm is the same one used in previous assignments and the code can be found here.The WordCount's output (can be checked in here) can be used as an input for the second (main MapReducer). In this manner, we avoided computing the global frequency of a word each time we encounter it thus minimizing the number of operations.
The reducer will first load the wordcount.txt file then it will create a HashMap:

```
 HashMap<String, String> wordcount = new HashMap<String, String>();
reader = new BufferedReader(
        new FileReader(
                new File(
                        "/home/cloudera/workspace/
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣StringSimilarityJoins/output/wordcount.txt")));
String pattern;
while ((pattern = reader.readLine()) != null) {
    String[] word = pattern.split(",");
    wordcount.put(word[0], word[1]);
}
```

Using the HashMap, it is easy to get the frequency of a word by simply calling the method:wordcount.get(word). Now, it possible to add global frequency estimated using the previous wordcount MapReduce algorithm for each encountered word:

```
HashSet<String> wordsHS = new HashSet<String>(wordsL);

StringBuilder wordswithcountSB = new StringBuilder();

String firstprefix = "";
for (String word : wordsHS) {
    wordswithcountSB.append(firstprefix);
    firstprefix = ",␣";
    wordswithcountSB.append(word + "#" + wordcount.get(word));
}
```

2.8 Ordering the encountered words in ascending frequency order:

Using the steps above, we should be getting for each line in the output a list of unique words each followed by their global frequency. However, the assignment requests having an ascending order of the words within each line. To achieve this goal, we define a sorting method within the reducer as follows:

```
java.util.List<String> wordswithcountL = Arrays
        .asList(wordswithcountSB.toString().split("\\s*,\\s*"));

Collections.sort(wordswithcountL, new Comparator<String>() {
    public int compare(String o1, String o2) {
        return extractInt(o1) - extractInt(o2);
    }
```

```
    int extractInt(String s) {
        String num = s.replaceAll("[^#\\d+]", "");
        num = num.replaceAll("\\d+#", "");
        num = num.replaceAll("#", "");
        return num.isEmpty() ? 0 : Integer.parseInt(num);
    }
});
```

2.9 Storing the results on the HDFS:

The reducer outputs the (key,value) pairs in TextOutputFormat in HDFS:

```
StringBuilder wordswithcountsortedSB = new StringBuilder();

String secondprefix = "";
for (String word : wordswithcountL) {
    wordswithcountsortedSB.append(secondprefix);
    secondprefix = ",␣";
    wordswithcountsortedSB.append(word);
}

context.getCounter(CUSTOM_COUNTER.NB_LINES).increment(1);

context.write(key, new Text(wordswithcountsortedSB.toString()));
```

It is possible to check the code here and the output in here.

2.10 Storing the pre-processing output without frequency:

We are almost done with the requested pre-processing task. The only thing that remains is to remove the global frequency which appears beside every encountered word.
This task is realized by making a slight change to our reducer. In fact, we only need to print the words in the reducer without printing the frequency which worresponds to the following line of code:

```
context.write(key, new Text(wordswithcountsortedSB.toString()
.replaceAll("#\\d+", "")));
```

At this stage, the pre-processing task is done.
The code can be checked in this link and the output can be seen here

3 Set-similarity joins:

The goal of this section (and of this assignment) is to design an efficient way to determine similar documents (lines in our case since we consider separate lines as separate documents int this assignment). Documents are declared similar according to a s similarity function if that similarity function exceeds a fixed threshold.

```
Note:
The following section uses the result of the pre-processing task as an input
```

3.1 Question a:

Avoiding redundant comparison of documents:
What we mean by redundant comparison is the repeated comparison between two docs. In fact, if we run blindly our code we will end up comparing documents di and dj at least twice (comparing the couples (di,dj) and (dj,di)). Since we aim to design an efficient algorithm, overcome this problem seems to be of a high importance.
Removing redundant comparison can be done by defining a new class equivalent to a tuple where

the order of apparition doen't matter and where (di,dj) and (dj,di) are identical.
We will also need to define a new comparison method for our newly defined class that corresponds to theses specifications. The following listing shows the used code to achieve that goal:

```
class TextPair implements WritableComparable<TextPair> {

    private Text first;
    private Text second;

    public TextPair(Text first, Text second) {
        set(first, second);
    }

    public TextPair() {
        set(new Text(), new Text());
    }

    public TextPair(String first, String second) {
        set(new Text(first), new Text(second));
    }

    public Text getFirst() {
        return first;
    }

    public Text getSecond() {
        return second;
    }

    public void set(Text first, Text second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }

    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }

    @Override
    public String toString() {
        return first + "␣" + second;
    }

    @Override
    public int compareTo(TextPair other) {
        int cmpFirstFirst = first.compareTo(other.first);
        int cmpSecondSecond = second.compareTo(other.second);
        int cmpFirstSecond = first.compareTo(other.second);
        int cmpSecondFirst = second.compareTo(other.first);

        if (cmpFirstFirst == 0 && cmpSecondSecond == 0 || cmpFirstSecond == 0
```

```
                && cmpSecondFirst == 0) {
            return 0;
        }

        Text thisSmaller;
        Text otherSmaller;

        Text thisBigger;
        Text otherBigger;

        if (this.first.compareTo(this.second) < 0) {
            thisSmaller = this.first;
            thisBigger = this.second;
        } else {
            thisSmaller = this.second;
            thisBigger = this.first;
        }

        if (other.first.compareTo(other.second) < 0) {
            otherSmaller = other.first;
            otherBigger = other.second;
        } else {
            otherSmaller = other.second;
            otherBigger = other.first;
        }

        int cmpThisSmallerOtherSmaller = thisSmaller.compareTo(otherSmaller);
        int cmpThisBiggerOtherBigger = thisBigger.compareTo(otherBigger);

        if (cmpThisSmallerOtherSmaller == 0) {
            return cmpThisBiggerOtherBigger;
        } else {
            return cmpThisSmallerOtherSmaller;
        }
    }

    @Override
    public int hashCode() {
        return first.hashCode() * 163 + second.hashCode();
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof TextPair) {
            TextPair tp = (TextPair) o;
            return first.equals(tp.first) && second.equals(tp.second);
        }
        return false;
    }

}
```

Map: Emitting the document ids as a key and the content as value: The following listings show the code used in this step:

```
public static class Map extends Mapper<Text, Text, TextPair, Text> {

    private BufferedReader reader;
```

```
    private static TextPair textPair = new TextPair();

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

        HashMap<String, String> linesHP = new HashMap<String, String>();
        reader = new BufferedReader(
                new FileReader(
                        new File(
                                "/home/cloudera/workspace/StringSimilarityJoins
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣/output/preprocessing_nofreq.txt")));
        String pattern;
        while ((pattern = reader.readLine()) != null) {
            String[] line = pattern.split(",");
            linesHP.put(line[0], line[1]);
        }
```

```
for (String line : linesHP.keySet()) {
    if (key.toString().equals(line)) {
        continue;
    }

    textPair.set(key, new Text(line));
    context.write(textPair, new Text(value.toString()));
}
}
}
```

In this code, we used our defined class TextPair in order to emit the document id with another document id as a key and the document's content as a value.

Perform the Jaccard computations:
This computations are done within the reducer class.
First, we need to define a function that performs the Jaccard computation:

```
public static class Reduce extends Reducer<TextPair, Text, Text, Text> {

    private BufferedReader reader;

    public double jaccardsim(TreeSet<String> s1, TreeSet<String> s2) {

        if (s1.size() < s2.size()) {
            TreeSet<String> s1bis = s1;
            s1bis.retainAll(s2);
            int inter = s1bis.size();
            s1.addAll(s2);
            int union = s1.size();
            return (double) inter / union;
        } else {
            TreeSet<String> s1bis = s2;
            s1bis.retainAll(s1);
            int inter = s1bis.size();
            s2.addAll(s1);
            int union = s2.size();
            return (double) inter / union;
        }

    }
```

We then load the result of the pre-processing task as follows:

```
@Override
public void reduce(TextPair key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

    HashMap<String, String> linesHP = new HashMap<String, String>();
    reader = new BufferedReader(
            new FileReader(
                    new File(
                            "/home/cloudera/workspace/StringSimilarityJoins/output
    String pattern;
    while ((pattern = reader.readLine()) != null) {
        String[] line = pattern.split(",");
        linesHP.put(line[0], line[1]);
    }
```

Then, we create a first set which will take as values the words list of the second document doc2 in the key pair of documents (di, dj):

```
TreeSet<String> wordsof2ndlineinpairTS = new TreeSet<String>();
String wordsof2ndlineinpairS = linesHP.get(key.getSecond()
        .toString());
for (String word : wordsof2ndlineinpairS.split("␣")) {
    wordsof2ndlineinpairTS.add(word);
}
```

Then, we create an second set which will take as values the words list of the first document doc1 in the key pair of documents (di, dj):

```
TreeSet<String> wordsTS = new TreeSet<String>();

for (String word : values.iterator().next().toString().split("␣")) {
    wordsTS.add(word);
}
```

We are now able to perform the Jaccard computations between the two sets:

```
context.getCounter(CUSTOM_COUNTER.NB_COMPARISIONS).increment(1);
double sim = jaccardsim(wordsTS, wordsof2ndlineinpairTS);
```

Reporting the number of comparisons:
We will use the same trick as in the pre-processing task to keep track of the number of comparisons: we define a counter as follows:

```
public static enum CUSTOM_COUNTER {
    NB_COMPARISIONS_A,
};
```

Moreover, we need to define another method within the driver in order to store the result on HDFS:

```
job.waitForCompletion(true);
long counter = job.getCounters()
        .findCounter(CUSTOM_COUNTER.NB_COMPARISIONS_A).getValue();
Path outFile = new Path("NB_COMPARISIONS_A.txt");
BufferedWriter br = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
br.write(String.valueOf(counter));
br.close();
return 0;
```

Finally, the entre code can be checked using this link. The output is available here..
In order to retrieve the execution time, we can look at the localhost hadoop page. The following figure gives the execution time for this code. Please note that the code was run only for the header

of the document corpus which explains why I got such short times. The execution times are much bigger when it comes to using the entire document corpus.

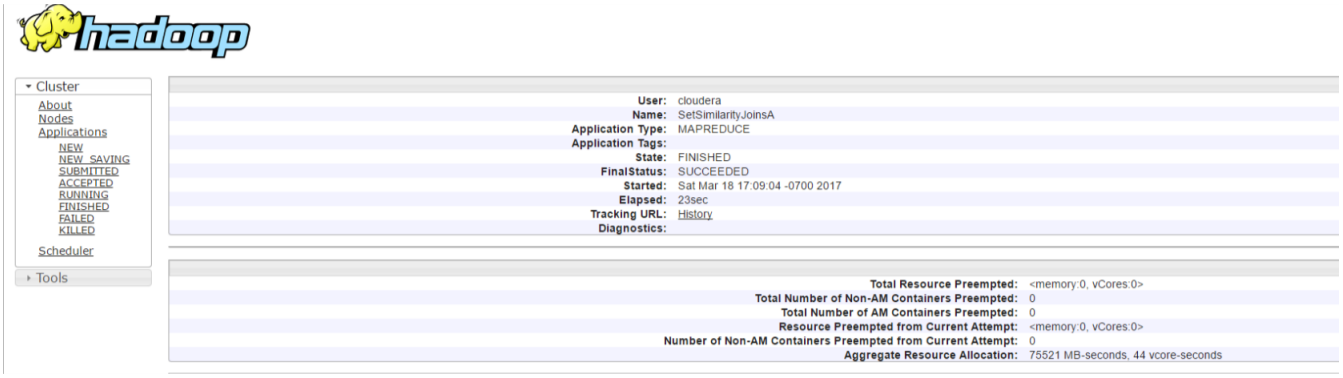

Figure 1: Execution time for question a)

3.2 Question b):

As suggested by the assignment, the value of $|d| - \lceil t|d| \rceil$ seems to be an important factor in the code. The following listing gives how we defined that number in order to limit the number of words within each document to be under that level.

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

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

        String[] wordsL = value.toString().split("␣");
        long numberofwordstokeep = Math.round(wordsL.length
                - (wordsL.length * 0.8) + 1);
        String[] wordstokeepL = Arrays.copyOfRange(wordsL, 0,
                (int) numberofwordstokeep);

        for (String wordtokeep : wordstokeepL) {
            word.set(wordtokeep);
            context.write(word, key);
        }
    }
}
```

Proceeding to similarity computation: In the reducer, we will reuse the same method used to estimate the Jaccard similarity:

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    private BufferedReader reader;

    public double jaccardsim(TreeSet<String> s1, TreeSet<String> s2) {

        if (s1.size() < s2.size()) {
            TreeSet<String> s1bis = s1;
            s1bis.retainAll(s2);
            int inter = s1bis.size();
            s1.addAll(s2);
            int union = s1.size();
```

```
            return (double) inter / union;
        } else {
            TreeSet<String> s1bis = s2;
            s1bis.retainAll(s1);
            int inter = s1bis.size();
            s2.addAll(s1);
            int union = s2.size();
            return (double) inter / union;
        }

    }
```

In the same fashion as in question a), we load the result of the pre-processing task in the reducer function using a HashMap element:

```
HashMap<String, String> linesHP = new HashMap<String, String>();
reader = new BufferedReader(
        new FileReader(
                new File(
                        "/home/cloudera/workspace/StringSimilarityJoins/
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣output/preprocessing_nofreq.txt")));
String pattern;
while ((pattern = reader.readLine()) != null) {
    String[] line = pattern.split(",");
    linesHP.put(line[0], line[1]);
}
```

We get all possible combinations of documents pairs in the document list if a word is present in several documents:

```
if (wordsL.size() > 1) {
    ArrayList<String> pairs = new ArrayList<String>();
    for (int i = 0; i < wordsL.size(); ++i) {
        for (int j = i + 1; j < wordsL.size(); ++j) {
            String pair = new String(wordsL.get(i) + "␣"
                    + wordsL.get(j));
            pairs.add(pair);
        }
    }
```

Finally, we create 2 sets containing the words in each document of the key pair, using HashMap object previously defined. It is possible to compute the Jaccard Similarity of 2 documents in a pair:

```
for (String pair : pairs) {
    TreeSet<String> wordsof1stlineinpairTS = new TreeSet<String>();
    String wordsof1stlineinpairS = linesHP
            .get(pair.split("␣")[0].toString());
    for (String word : wordsof1stlineinpairS.split("␣")) {
        wordsof1stlineinpairTS.add(word);
    }

    TreeSet<String> wordsof2ndlineinpairTS = new TreeSet<String>();
    String wordsof2ndlineinpairS = linesHP
            .get(pair.split("␣")[1].toString());
    for (String word : wordsof2ndlineinpairS.split("␣")) {
        wordsof2ndlineinpairTS.add(word);
    }

    context.getCounter(CUSTOM_COUNTER.NB_COMPARISIONS_B)
            .increment(1);
    double sim = jaccardsim(wordsof1stlineinpairTS,
```

```
            wordsof2ndlineinpairTS);
```

The final step will be to add an if loop in order to output only pairs whose similarity function is greater than the fixed threshold:

```
if (sim >= 0.8) {
    context.write(new Text("(" + pair.split("␣")[0] + ",␣"
            + pair.split("␣")[1] + ")"),
            new Text(String.valueOf(sim)));
}
}
}
}
}
}
```

Retrieving the number of comparisons: We use the same method as in question a to keep the number of comparisons and to store it on the HDFS. The following listings give the relevant piece of code to this step:

```
public static enum CUSTOM_COUNTER {
    NB_COMPARISIONS_B,
};
```

```
job.waitForCompletion(true);
long counter = job.getCounters()
        .findCounter(CUSTOM_COUNTER.NB_COMPARISIONS_B).getValue();
Path outFile = new Path("NB_COMPARISIONS_B.txt");
BufferedWriter br = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
br.write(String.valueOf(counter));
br.close();
return 0;
```

Finally, the code can be checked via this link and the output via this link.
The following figure gives the execution time for this algorithm:

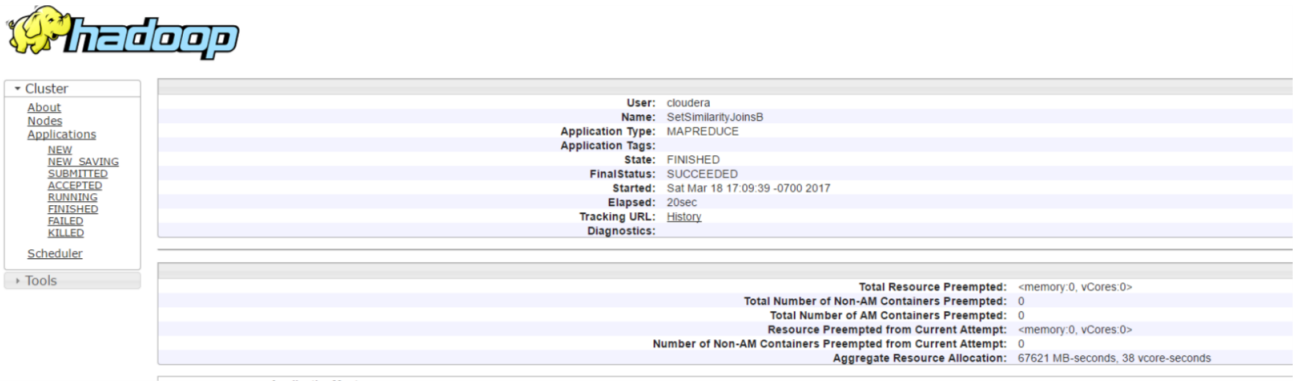

Figure 2: Execution time for the algorithm of question b)

3.3 Question c):

```
Note: these results were obtained only for the header of document corpus.
The actual execution time are much bigger if the code is run for the entire docume
(it can take a while).
```

The following table summarises the execution time and the number of comparisons:

	Question a)	Question b)
Number of comparisons	11026	70
Execution time	23sec	20sec

Table 1: Execution times and numbers of comparisons

The two algorithm aren't that far away when it comes to the execution time This may be caused by the too small size of the sample of the data corpus that I took to run the code. however, when it comes to the number of performed comparisons, it is astonishingly big. It is quite natural that the algorithm of question b) takes a lesser number of comparisons than the one of question a), the reason being simply the fact that the algorithm b) was designed to reduce the number of comparisons by indexing only words that have small frequencies. It is in fact a judicious choice to target words with small frequencies since these words can enable us the most to discriminate between sentences. This operation enable us to avoid making an overwhelming number of comparisons between documents that are not similar.