Big Data Processing & Analytics: assignment 1:

Abdelhamid EZZERG

February 16,2017

1 Introduction:

This paper report the results obtained for our first mapreduce job: the implementation of an inverted index for the document corpus composed of the files available at: http://www.gutenberg.org/cache/epub/100/pg100.txt, http://www.gutenberg.org/cache/epub/31100/pg31100.txt and http://www.gutenberg.org/cache/epub/3200/pg3200.txt respectively.
The inverted index job will, starting from a document corpus, provide for each word the filename of the documents where it appears along with additional information such as the frequency of words.
The following is an example of an inverted index as described in the assignment file:
Suppose we have a document corpus composed of the three following files:

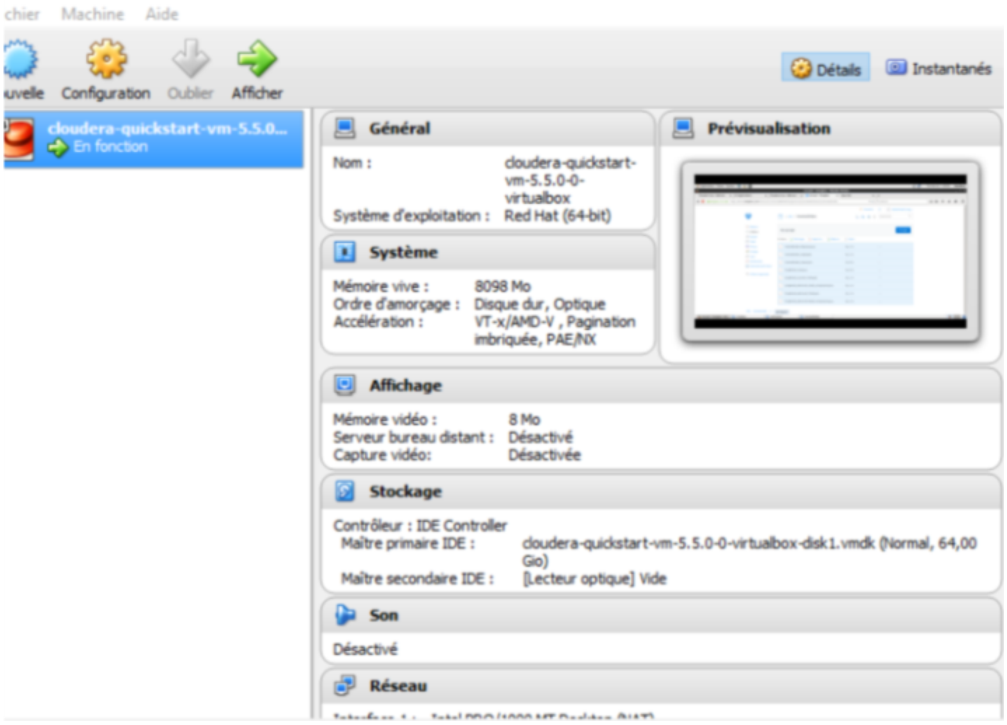
- doc1.txt: "This is a very useful program. It is also quite easy."
- doc2.txt: "This is my first MapReduce program."
- doc3.txt: "Its result is an inverted index."

An inverted index of the data above should provide the following results:

this	doc1.txt, doc2.txt
is	doc1.txt, doc2.txt, doc3.txt
a	doc1.txt
program	doc1.txt, doc2.txt
...	...

2 Virtual machine setup:

The following picture shows some of the specs of the virtual machine:



Général

Nom : cloudera-quickstart-vm-5.5.0-0-virtualbox
Système d'exploitation : Red Hat (64-bit)

Système

Mémoire vive : 8098 Mo
Ordre d'amorçage : Disque dur, Optique
Accélération : VT-x/AMD-V , Pagination imbriquée, PAE/NX

Affichage

Mémoire vidéo : 8 Mo
Serveur bureau distant : Désactivé
Capture vidéo: Désactivée

Stockage

Contrôleur : IDE Controller
Maître primaire IDE : cloudera-quickstart-vm-5.5.0-0-virtualbox-disk1.vmdk (Normal, 64,00 Gio)
Maître secondaire IDE : [Lecteur optique] Vide

Son

Désactivé

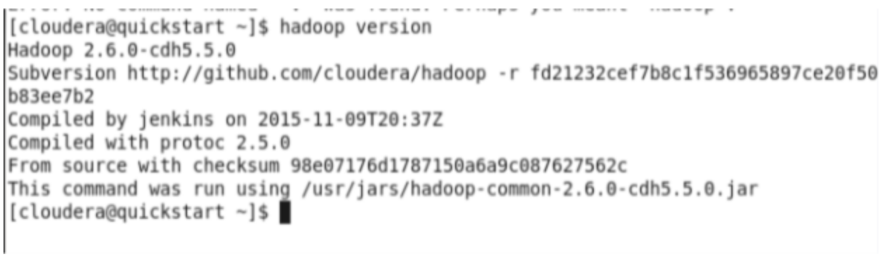
Réseau

Figure 1: Virtual Machine specs

The following table shows abrief description of the main specs of the virtual machine:

Specs	Description
OS	Red Hat 64-bit
RAM	8Go
Hard Drive	64Go
Acceleration	AMD-V

I kept the default specs for the virtual machine except for the RAM. In fact the Operating System is a 64-bit one and so it can support RAM that is larger than 4Go. Sine I had enough RAM, I decided to run the app using 8Go of RAM so that the process will go smoothly.
As for the used hadoop version, it is hadoop 2.6.



```
[cloudera@quickstart ~]$ hadoop version
Hadoop 2.6.0-cdh5.5.0
Subversion http://github.com/cloudera/hadoop -r fd21232cef7b8c1f536965897ce20f50
b83ee7b2
Compiled by jenkins on 2015-11-09T20:37Z
Compiled with protoc 2.5.0
From source with checksum 98e07176d1787150a6a9c087627562c
This command was run using /usr/jars/hadoop-common-2.6.0-cdh5.5.0.jar
[cloudera@quickstart ~]$
```

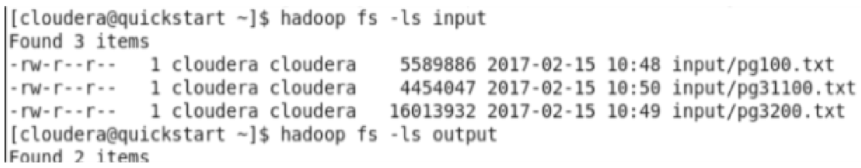
3 Quick setup of the environment:

The first thing to do is to setup quickly the environment to process the code. The first step being the creation of a new package that I called InvertedIndex. The package was created through the command prompt using the following lines:

```
cd workspace/InvertedIndex
mkdir input
mkdir output
```

The following step was to download the text files of the corpus using the curl command.
After this, it is necessary to load the relevant files into hadoop hdfs using the folowing commands:

```
hadoop fs -mkdir input
hadoop fs -put workspace/InvertedIndex/input/pg100.txt input
hadoop fs -put workspace/InvertedIndex/input/pg3200.txt input
hadoop fs -put workspace/InvertedIndex/output/pg31100.txt input
```



```
[cloudera@quickstart ~]$ hadoop fs -ls input
Found 3 items
-rw-r--r--   1 cloudera cloudera    5589886 2017-02-15 10:48 input/pg100.txt
-rw-r--r--   1 cloudera cloudera    4454047 2017-02-15 10:50 input/pg31100.txt
-rw-r--r--   1 cloudera cloudera   16013932 2017-02-15 10:49 input/pg3200.txt
[cloudera@quickstart ~]$ hadoop fs -ls output
Found 2 items
```

Once these simple steps are done, we can start writing our map reduce code which is detailed in the next sections.

4 Stop words:

4.1 Question a)i):

In this section we are interested in determining the stop words that appear in our documents corpus. A stop word is defined as a word that appear with high frequency within the document corpus. In this assignment the threshold value that discriminates a stop word from a normal word is fixed at 4000 times. In other words, our mapreduce job should return words whose frequencies are superior to 4000.
We noticed that the worked we are asked to complete in this part of the assignment is very similar to the one we had to complete in the self-assignment: the WordCount mapreduce job. In fact, what we need to implement here in order to determine the stop words is merely using the same mapper and reducers as the ones that were used for the word count job with simply adding a condition on the frequency within the reducer. For the word count source code, we have used the one that we used i the previous assignment and that was available at: http://snap.stanford.edu/class/cs246-data-2014/WordCount.java.

Our Mapper class is as follows:

```
public static class Map extends
                Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable ONE = new IntWritable(1);
        private Text word = new Text();

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
          for (String token : value.toString().split("\\s+")) {
                word.set(token.toLowerCase());
                context.write(word, ONE);
                        }
                }
        }
```

The reducer is given as follows:

```
public static class Reduce extends
        Reducer<Text, IntWritable, Text, IntWritable> {
        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
         Context context) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
                sum += val.get();
        }
        if (sum > 4000) {
                context.write(key, new IntWritable(sum));
                        }
                }
        }
```

The main changes that occured to the codes in order to write the mapper and reducer are:

- The use of toLowerCase() method in order to avoid processing words twice just because the difference in the case.
- In the reducer we added the condition if(sum>4000), this is the main difference compared to the word count exercise.
- The output file needed to be an .csv file. Thus we added the following line in the code: job.getConfiguration().set("mapreduce.output.textoutputformat.separator", ",");

In order to look at the full lisitng of the code you can look at the file StopWords_main.java (This is a hyperlink).

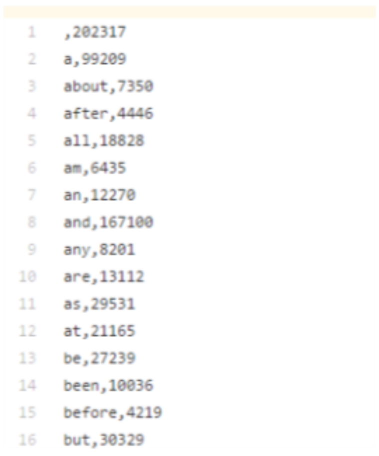
Once the code is written, we export it into a jar file so that it can be ran afterwards. Exporting the file into jar is easily done using the Eclipse IDE. The file being exported, we can execute the mapreduce job using the following command:

```
hadoop jar InvertedIndex.jar InvertedIndex.StopWords_main input output
```

We then use the following command to load the output file from the hdfs into the workspace folder:

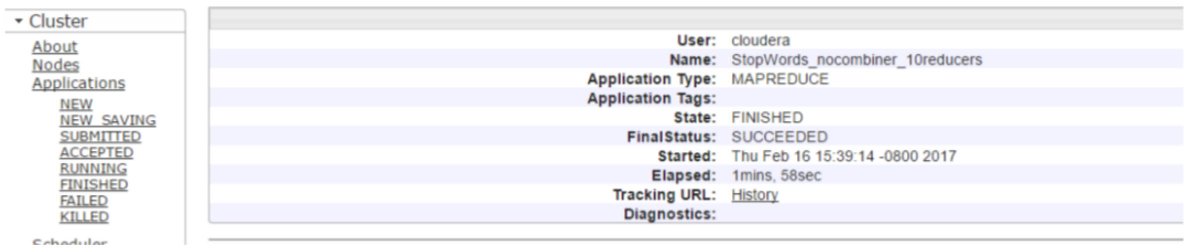
```
hadoop fs -getmerge output workspace/InvertedIndex/output/StopWords_main.csv
```

The following picture shows the beginning of the file StopWords_main.csv:



```
,202317
a,99209
about,7350
after,4446
all,18828
am,6435
an,12270
and,167100
any,8201
are,13112
as,29531
at,21165
be,27239
been,10036
before,4219
but,30329
```

It is also possible to check the execution time from the hadoop YARN Resources Manager:



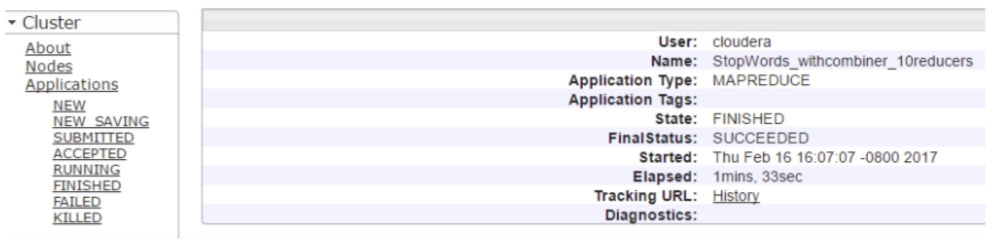
User: cloudera
Name: StopWords_nocombiner_10reducers
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 15:39:14 -0800 2017
Elapsed: 1mins, 58sec
Tracking URL: History
Diagnostics:

Figure 2: hadoop job log for: stop words with 10 reducers and no combiner

4.2 stop words using 10 reducers and no combiner:

The number of reducers is defined by using the method job.setNumReduceTasks(). We simply add write job.setNumReduceTasks(10).
Figure 2 shows the execution time which is **1min 58sec**. The output file is called: StopWords_nocomb_10red.csv. You can check the complete code at:StopWords_nocomb_10red.java.

4.3 Stop Words using 10 reducers and a combiner:

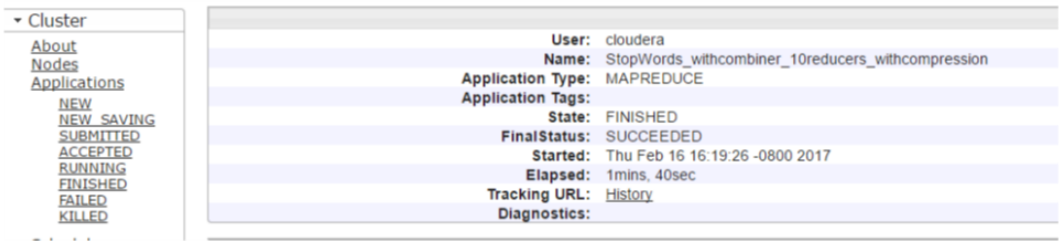


User: cloudera
Name: StopWords_withcombiner_10reducers
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 16:07:07 -0800 2017
Elapsed: 1mins, 33sec
Tracking URL: History
Diagnostics:

Figure 3: hadoop job log for: stop words with 10 reducers and with combiner

Actually, this is the same as the previous subsection except that we need to add the following line of code in order to add the combiner: job.setCombinerClass(Reduce.class);.
We report an improved execution time which was equal to **1min 33 sec**.
The output file is called: StopWords_withcomb_10red.csv and the full listing of code can be checked at: StopWords_withcomb_10red.java.

4.4 Stop Words using 10 reducers, a combiner and results of map compression:



User: cloudera
Name: StopWords_withcombiner_10reducers_withcompression
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 16:19:26 -0800 2017
Elapsed: 1mins, 40sec
Tracking URL: History
Diagnostics:

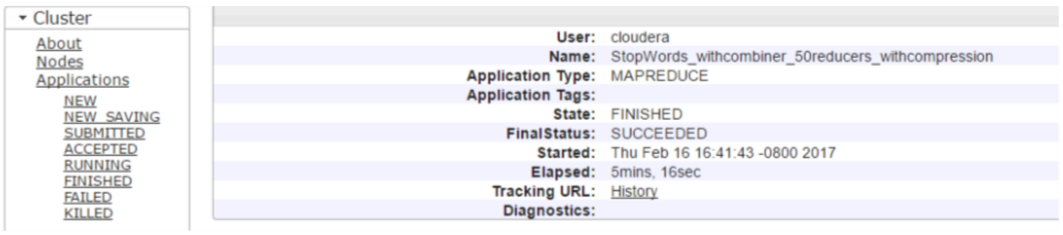
Figure 4: hadoop job log for: stop words with 10 reducers and with combiner and compression

Again, this is the same code with a slight modification. In fact, we just needed to add the following lines of code to the hadoop driver configuration:

```
FileOutputFormat.setCompressOutput(job, true);
FileOutputFormat.setOutputCompressorClass(job,
        org.apache.hadoop.io.compress.SnappyCodec.class);
```

The execution time is: **1min 40sec**. The output file is called StopWords_withcomb_10red_compression.csv.
The full listing can be consulted at StopWords_withcomb_10red_compression.java

4.5 Stop Words using 50 reducers, a combiner and results of map compression:



User: cloudera
Name: StopWords_withcombiner_50reducers_withcompression
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 16:41:43 -0800 2017
Elapsed: 5mins, 16sec
Tracking URL: History
Diagnostics:

Figure 5: hadoop job log for: stop words with 50 reducers and with combiner and compression

Same as before, just with 50 reducers.
The reported execution time is **5min 16sec**.
The output file is called StopWords_withcomb_50red_compression.csv.
The fill code lisitng can be consulted at: StopWords_withcomb_50red_compression.java
The following Table summarizes the execution times for the different variations we have tested:

Stop Words variant	Execution time
10 reducers / no combiner	1min 58sec
10 reducers / with combiner	1min 33sec
10 reducers / with combiner and compression	1min 40sec
50 reducers / with combiner and compression	5min 16sec

5 Inverted index:

First of all, we needed to change the file containing the stop words to be in a txt format instead of the csv one. We have processed these data without the use of a precise command.

5.1 The main idea:

This time adapting the WordCount source code is not as trivial as it was in the first part. thus, we will modify quite a few lines in order to implement the inverted index algorithm.
The main idea in what we wrote is as follows: the algorithm will need to provide a (key, value) pair of the form (word, collection of filename). Thus we can set both the keys and values output as Text in the hadoop driver:

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
```

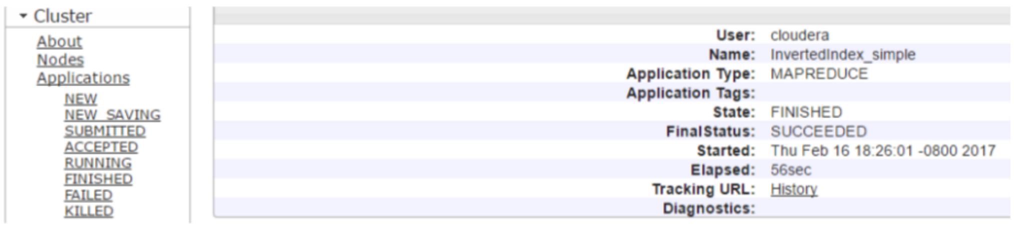
The mapper class has two roles: the first one is to create the pair (word, filename) or, in other words, determine for each word the filename of the document containing that word. The second role or task is to delete the stop words obtained from the previous section. (Hence the operation of transforming the stopwords.csv to stopwords.txt).
Here is the listing of the Mapper class:

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

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

                HashSet<String> stopwords = new HashSet<String>();
                BufferedReader Reader = new BufferedReader(
                                new FileReader(
                                        new File(
        "/home/cloudera/workspace/InvertedIndex/output/StopWords_main.txt")));
                String pattern;
                while ((pattern = Reader.readLine()) != null) {
                        stopwords.add(pattern.toLowerCase());
                        }

                String filenameStr = ((FileSplit) context.getInputSplit())
                                        .getPath().getName();
                filename = new Text(filenameStr);

                for (String token : value.toString().split("\\s+")) {
                        if (!stopwords.contains(token.toLowerCase())) {
                                word.set(token.toLowerCase());
                                }
                        }

                context.write(word, filename);

                }
        }
```

Each mapper will provide an output containing a list of (word, filename) pairs which contains duplicates. In fact, if word1 appears in the file file1 several times, say 3 times for example, the output file of the mapper would be off the form word1, file1; word1 , file1; word1 , file1.
The Reducer class task will be to organise these output in a list that does not accept duplicates: The final list will be a combination of the outputs of the mapper that would be expressed in the following form: word1, file1, file2; word2, file2, file 3 ....

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

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

                HashSet<String> set = new HashSet<String>();

                for (Text value : values) {
                        set.add(value.toString());
                }

                StringBuilder builder = new StringBuilder();

                String prefix = "";
                for (String value : set) {
                        builder.append(prefix);
                        prefix = ", ";
                        builder.append(value);
                }

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

The output file is called InvertedIndex_simple.csv. And the complete code can be consulted at InvertedIndex_simple.java The following figure shows the execution time for the algorithm:



Cluster
About
Nodes
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED

```
            User:  cloudera
            Name:  InvertedIndex_simple
Application Type:  MAPREDUCE
Application Tags:
           State:  FINISHED
     FinalStatus:  SUCCEEDED
         Started:  Thu Feb 16 18:26:01 -0800 2017
         Elapsed:  56sec
    Tracking URL:  History
     Diagnostics:
```

Figure 6: Execution time for inverted index job

5.2 Unique words:

In our code, we can define a customized counter.
We will keep the same mapper as before. However, the reducer needs to be modified slightly. In fact, the reducer keeps more or less the same structure except that it incorporates two new tasks: the first one is to select only unique words through an if statement. In other words, the reducer will only select whose filename list is of length 1. The second task is to update the customized counter whenever a unique word has been encountered.
Here is the listing of the new reducer:

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

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

                HashSet<String> set = new HashSet<String>();

                for (Text value : values) {
                        set.add(value.toString());
                }

                if (set.size() == 1) {

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

                        StringBuilder builder = new StringBuilder();

                        String prefix = "";
                        for (String value : set) {
                                builder.append(prefix);
                                prefix = ",␣";
                                builder.append(value);
                        }

                        context.write(key, new Text(builder.toString()));

                        }
                }
        }
```

The output is called InvertedIndex_unique.csv and the full code listing is available at: InvertedIndex_unique.java. The execution time is: 52sec.
The output of the hadoop job printed at the command prompt enables us to see the number of unique words:

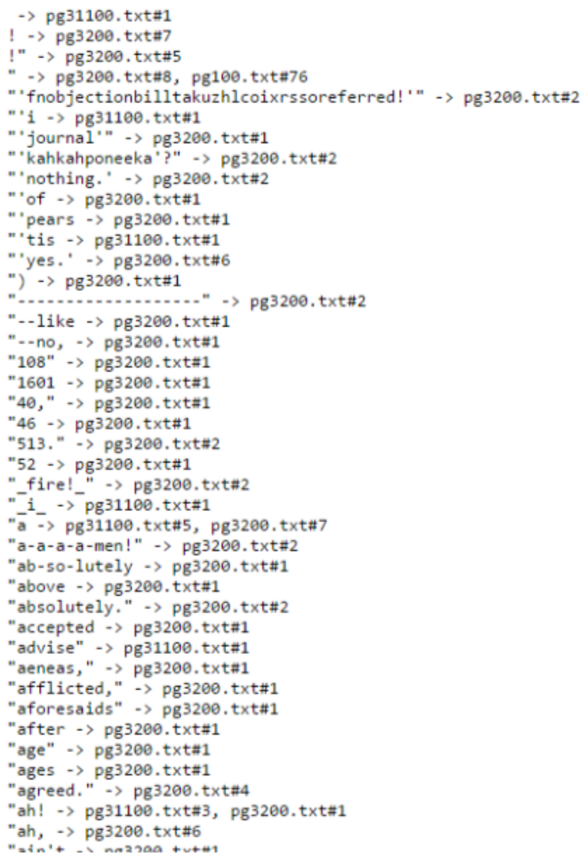
```
InvertedIndex.InvertedIndex_unique$CUSTOM_COUNTER
    UNIQUE_WORDS=68476
```

5.3 Extended inverted index:

The code of this section is pretty similar to the one used in question b). We will keep the same mapper as before and a very similar reducer. The only difference that appears in the case of the reducer is that, this time, it should allow repeated filenames. In fact, since we want access to the frequency, we need to be able to store repeated filenames.
Here is the listing of the reducer class:

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

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

                ArrayList<String> list = new ArrayList<String>();

                for (Text value : values) {
                        list.add(value.toString());
                }

                HashSet<String> set = new HashSet<String>(list);
                StringBuilder builder = new StringBuilder();

                String prefix = "";
                for (String value : set) {
                        builder.append(prefix);
                        prefix = ", ";
                        builder.append(value + "#" + Collections.frequency(list,
         value));
                }

                context.write(key, new Text(builder.toString()));

                }
        }
```

We see that in the reducer, and in order to count the number of repeated filenames, we used the java method Collections.frequency(array, object). The following figure shows a pert of the output file:



```
 -> pg31100.txt#1
! -> pg3200.txt#7
!" -> pg3200.txt#5
" -> pg3200.txt#8, pg100.txt#76
"'fnobjectionbilltakuzhlcoixrssoreferred!'" -> pg3200.txt#2
"'i -> pg31100.txt#1
"'journal'" -> pg3200.txt#1
"'kahkahponeeka'?" -> pg3200.txt#2
"'nothing.' -> pg3200.txt#2
"'of -> pg3200.txt#1
"'pears -> pg3200.txt#1
"'tis -> pg31100.txt#1
"'yes.' -> pg3200.txt#6
") -> pg3200.txt#1
"-------------------" -> pg3200.txt#2
"--like -> pg3200.txt#1
"--no, -> pg3200.txt#1
"108" -> pg3200.txt#1
"1601 -> pg3200.txt#1
"40," -> pg3200.txt#1
"46 -> pg3200.txt#1
"513." -> pg3200.txt#2
"52 -> pg3200.txt#1
"_fire!_" -> pg3200.txt#2
"_i_ -> pg31100.txt#1
"a -> pg31100.txt#5, pg3200.txt#7
"a-a-a-a-men!" -> pg3200.txt#2
"ab-so-lutely -> pg3200.txt#1
"above -> pg3200.txt#1
"absolutely." -> pg3200.txt#2
"accepted -> pg3200.txt#1
"advise" -> pg31100.txt#1
"aeneas," -> pg3200.txt#1
"afflicted," -> pg3200.txt#1
"aforesaids" -> pg3200.txt#1
"after -> pg3200.txt#1
"age" -> pg3200.txt#1
"ages -> pg3200.txt#1
"agreed." -> pg3200.txt#4
"ah! -> pg31100.txt#3, pg3200.txt#1
"ah, -> pg3200.txt#6
"ain't -> pg3200.txt#1
```

Figure 7: output sample

The output file is called InvertedIndex_frequency.csv and the full code lisitng is available at: InvertedIndex_frequency.java

6 Remarks and notes:

- In the first part, it seems that adding a combiner to hadoop job improves the performance. This was quite expected since this combiner class is generally used to reduce the volume of data transfer between Map and Reduce. The combiner summarizes the output data of the Mapper and thus it makes the process much easier for the Reducer.

- It seems that using compression did not result in a tremendous change in the execution time (1min33sec vs 1min 40sec). At first I thought that the algorithm using compression would be faster. However, I see that I was jumping to conlusions too early. In fact, since I do not have any information on the complexity of the compression and decompression schemes it is hard to decide which effect must be expected from the use of compression.
  However, it must be noted that the compression is quite useful when it comes to big data since it means a huge saving in terms of storage space and also in terms of the required time to transfer data between the different clusters/ servers ...

- While some people might think that the more parallelized the processing is, the faster it gets. The experience of this assignment proves the opposite. Adding many reducers may decrease the performance because we are not sure whether or not all the equipments of the system can handle too much parallelization: some parts like the CPU or the OS may be a bottleneck to the operation. And using many reducers may, as an example, cause the disk head to turn too much.
  In our example we achieved better performance with less reducers (50 was too much).

- For the last section, the use of a combiner to accelerate the jobs. I haven't tried it on the second part. However, judging from the performances of the first part, it seems that the combiner would accelerate the jobs.