Massive Data Processing

First Assignment:

A variation of an inverted index

Flavie Vampouille MSc in DS&BA

When I created the VirtualBow machine I put the parameters to:

- 1CPU
- 4096 Mo RAM (8 Gio RAM in my computer)
- 80 Mo video memory
- 64 Gio Maître primaire IDE dynamically allocated

To debug the eclipse installation and to write my codes I use a lot Stack Overflow and OpenClassrooms. And other web sites with java code exemples.

You are asked to implement an inverted index in MapReduce for the document corpus of pg100.txt (from http://www.gutenberg.org/cache/epub/100/pg100.txt), pg31100.txt (from http://www.gutenberg.org/cache/epub/31100/pg31100.txt) and pg3200.txt (from http://www.gutenberg.org/cache/epub/3200/pg3200.txt). This corpus includes the complete works of William Shakespear, Mark Twain and Jane Austen, respectively.

An inverted index provides for each distinct word in a document corpus, the filenames that contain this word, along with some other information (e.g., count/position within each document). For example, assume you are given the following corpus, consisting of doc1.txt, doc2.txt and doc3.txt:

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 would contain the following data (in random order):

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

I first created in Eclipse a new project name "InvertedIndex".

In the /home/cloudera/workspace/InvertedIndex folder I created a new folder name "input", in which I registered document corpus of:

- pg100.txt (from http://www.gutenberg.org/cache/epub/100/pg100.txt),
- pg31100.txt (from http://www.gutenberg.org/cache/epub/31100/pg31100.txt),
- pg3200.txt (from http://www.gutenberg.org/cache/epub/3200/pg3200.txt).

I then put those files in Hadoop with the following command lines:

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

I then check that the inputs are correctly registered.

```
[cloudera@quickstart InvertedIndex]$ hadoop fs -ls input
Found 3 items
-rw-r--r--   1 cloudera cloudera    5589889 2017-02-14 10:06 input/pg100.txt
-rw-r--r--   1 cloudera cloudera    4454050 2017-02-14 10:07 input/pg31100.txt
-rw-r--r--   1 cloudera cloudera   16013935 2017-02-14 10:10 input/pg3200.txt
```

Each time I run a new java application, I create the associated .jar file (export + jar file) and type the following command lines:

```
cd /home/cloudera/workspace/InvertedIndex/JARexport
hadoop jar question_name.jar class_name input output_name
```

To find the execution time I then look on localhost:8088.

Finally, to export the results in a unique csv file I type the following command line:

```
hadoop fs -getmerge output_name
/home/cloudera/workspace/InvertedIndex/output/Question_name.csv
```

Question a.

Run a Map Reduce program to identify stop words (words with frequency > 4000) for the given document corpus. Store them in a single csv file on HDFS (stopwords.csv). You can edit the several parts of the reducers' output after the job finishes (with hdfs commands or with a text editor), in order to merge them as a single csv file.

I create a new package in the InvertedIndex project named "question_a".

For each sub question, I create a class in this package named respectively for questions a_i, a_ii, a_iii and a_iv

- i_StopWords_10Reducers_NoCombiner
- ii_StopWords_10Reducers_Combiner
- iii_StopWords_10Reducers_Combiner_Compression
- iv_StopWords_50Reducers_Combiner_Compression

And I exported the results in a unique csv file named Result_a.

My whole code is based on the Assignment 0: WordCount project from Stanford (http://snap.stanford.edu/class/cs246-data-2014/WordCount.java) in which I modify some elements. Each time I modify something I put a comment on the code.

(i) Use 10 reducers and do not use a combiner. Report the execution time.

The main changes I do on the WordCount code for i_StopWords_10Reducers_NoCombiner class are:

To obtain output txt files and set the number of Reducer to 10:

```
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
// setNumReduceTasks returns the number of reduce tasks for this job
job.setNumReduceTasks(10);

job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);

// getConfiguration return the configuration for the job to create the txt file output
job.getConfiguration().set("mapreduce.output.textoutputformat.separator", " ");
```

To count as the same the words whether they are in lower or upper case and if they are preceded or followed by non-letter character (cat cat. cat; “cat cat” cat! …)

```
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+")) {
            // in word.set add toLowerCase to put everything in lower case before counting
            // and remove all non-letter characters
            word.set(token.toLowerCase().replaceAll("[^a-zA-Z ]", " "));
            context.write(word, ONE);
        }
    }
}
```

To count only stop words, i.e. words which appear more than 4000 times:

```java
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();
        }
        // to write only stop words in the output files, i.e. words with frequency > 4000
        if (sum > 4000) {
            context.write(key, new IntWritable(sum));
        }
    }
}
```

The output I obtained, once merged, is of the form:

about	7350
be	27239
before	4219
by	19509
her	24277
much	4712
old	4092
up	8608
where	4197
you	35121

The execution time is: 9 minutes, 9 seconds

Application Overview	
User:	cloudera
Name:	StopWords_10Reducers_NoCombiner
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Tue Feb 14 10:27:28 -0800 2017
Elapsed:	9mins, 9sec
Tracking URL:	History
Diagnostics:	

(ii) Run the same program again, this time using a Combiner. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

I modify my code of question a_i to use a combiner:

```
// Configure a job's combiner implementation job.setCombinerClass(MyJob.MyReducer.class);
job.setCombinerClass(Reduce.class);
```

The execution time is: 6 minutes, 58 seconds

Application Overview	
User:	cloudera
Name:	ii_StopWords_10Reducers_Combiner
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Tue Feb 14 10:41:07 -0800 2017
Elapsed:	6mins, 58sec
Tracking URL:	History
Diagnostics:	

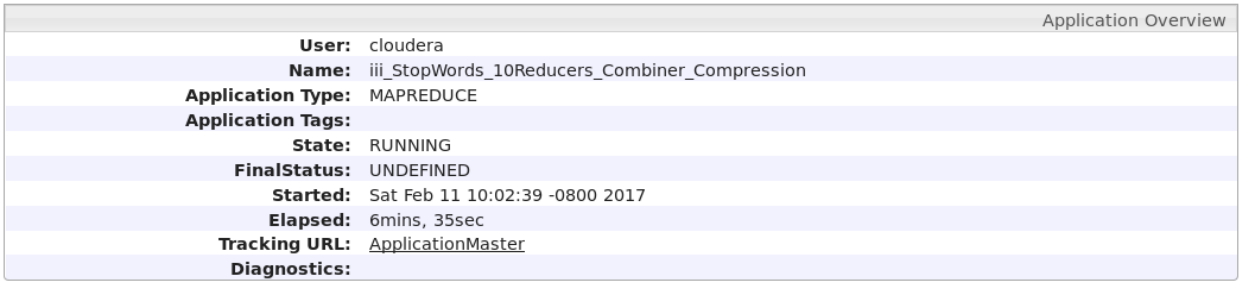
The Combiner phase takes each key-value pair from the Map phase, processes it, and produces the output as **key-value collection** pairs. The execution time is slightly faster with a combiner.

(iii) Run the same program again, this time compressing the intermediate results of map (using any codec you wish). Report the execution time. Is there any difference in the execution, time compared to the previous execution? Why?

I modify my code to use compression in the following way:

```
// compression
job.getConfiguration().setBoolean("mapred.compress.map.output", true);
job.getConfiguration().setClass("mapred.map.output.compression.codec",BZip2Codec.class, CompressionCodec.class);
```

The execution time is: 6 minutes, 35 seconds



Application Overview	
User:	cloudera
Name:	iii_StopWords_10Reducers_Combiner_Compression
Application Type:	MAPREDUCE
Application Tags:	
State:	RUNNING
FinalStatus:	UNDEFINED
Started:	Sat Feb 11 10:02:39 -0800 2017
Elapsed:	6mins, 35sec
Tracking URL:	ApplicationMaster
Diagnostics:	

Compression input files

This is not the case here but if the input file is compressed, then the bytes read in from HDFS is reduced, which means less time to read data. This time conservation is beneficial to the performance of job execution. If the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use. For example, a file ending in .gz can be identified as gzip-compressed file and thus read with GzipCodec.

Compressing output files

Once again I don't do it there but often we need to store the output as history files. If the amount of output per day is extensive, and we often need to store history results for future use, then these accumulated results will take extensive amount of HDFS space. However, these history files may not be used very frequently, resulting in a waste of HDFS space. Therefore, it is necessary to compress the output before storing on HDFS.

Compressing map output

This is what interest me here: even if the MapReduce application reads and writes uncompressed data, it benefits from compressing the intermediate output of the map phase. Since the map output is written to disk and transferred across the network to the reducer nodes, by using a compressor you can get performance gains simply because the volume of data to transfer is reduced.

(iv) Run the same program again, this time using 50 reducers. Report the execution time. Is there any difference in the execution time, compared to the previous execution? Why?

I use the code with compression and just the number of reducers to 50:

```
// setNumReduceTasks returns the number of reduce tasks for this job
job.setNumReduceTasks(50);
```

The execution time is: 25 minutes, 39 seconds

Application Overview	
User:	cloudera
Name:	iv_StopWords_50Reducers_Combiner_Compression
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Tue Feb 14 12:36:11 -0800 2017
Elapsed:	25mins, 39sec
Tracking URL:	History
Diagnostics:	

Putting 50 reducers really increases the execution time.

Question b.

Implement a simple inverted index for the given document corpus, as shown in the previous Table, skipping the words of stopwords.csv.

1. SimpleInvertedIndex class

In the previous questions, we want to obtain a list of words appearing in the texts with the number of times they appear. Here we want the words and the name of the file in which they appear. So instead of a (string, integer) output we will have a (string, string) output. It is hence necessary to modify the output value class.

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

2. Map class

The goal of this map function is to ignore stop words and to get, for each other word, the name of the file in which it appears.

I first create a txt file with the results of question a containing one stop words per line in lower case that I name StopWords.txt. Then I create a HashSet Class (“blocked”) containing all this words (except the empty one) using hashSet (to create the class) and BufferedReader (to read the StopWords.txt file).

Then I need to be able to locate the word which is mapped: in which file is it located? For that I use context.getInputSplit()).getPath().getName().

And finally, I applied the usual function (as in question a) but ignoring the stop words and mapping (word, filename) instead of (word, ONE).

```
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 {

        // to create a HashSet Class of the words we want to block in the map function (the stop words)
        HashSet<String> blocked = new HashSet<String>();
        // use BufferedReader to read every line of the stop words file created with question a
        BufferedReader BR = new BufferedReader(new FileReader(
                new File("/home/cloudera/workspace/InvertedIndex/StopWords/StopWords.txt")));
        // add every stop word to the HashSet Class without the empty one
        String motLu;
        while((motLu = BR.readLine()) != null){
            blocked.add(motLu);
        }

        // to identify the file from which the word comes
        String location = ((FileSplit) context.getInputSplit()).getPath().getName();
        filename = new Text(location);

        // in word.set add toLowerCase to put everything in lower case before counting
        // and remove all non-letter characters
        for (String token : value.toString().split("\\s+")) {
            if (!blocked.contains(token.toLowerCase().replaceAll("[^a-zA-Z ]", ""))) {
                word.set(token.toLowerCase().replaceAll("[^a-zA-Z ]", ""));
            }
        }
        context.write(word, filename);
    }
```

3. Reduce class

In word count and question a we had for each word a count of one each time it appears and we just have to do the sum for each particular word (output key class). We now have for all words, the associated file name each time it appears, and we want to know for each particular word (output key class) in which document it appears without being interested in the number of times.

As our Map class produce an output of the form

word1, file1
word1, file1
word1, file2
word1, file3
word2, file2
word2, file3
word2, file3
word2, file3
…

we now want to know in which file word_i appears at least one time. For that I again use HashSet as it takes in count only one time the identical elements.

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

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

        // to create a HashSet Class with value the file names
        HashSet<String> HSCfile = new HashSet<String>();
        for (Text val : values) {
            HSCfile.add(val.toString());
        }

        // to create the new output
        StringBuilder SB = new StringBuilder();
        String separator = " ";
        for (String val : HSCfile) {
            SB.append(separator);
            SB.append(val);
        }

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

    }
}
```

The results are in the csv file Result_b.csv. They are of the form:

abhor	pg31100.txt	pg100.txt	
abhorrd	pg100.txt		
abhorred	pg31100.txt	pg3200.txt	pg100.txt
abhorrence	pg31100.txt		
abhorrent	pg31100.txt		
abhors	pg100.txt		
abhorson	pg100.txt		
abide	pg31100.txt	pg3200.txt	pg100.txt
abides	pg3200.txt	pg100.txt	
abideth	pg3200.txt		
abiding	pg3200.txt		
abiling	pg3200.txt		
abilities	pg31100.txt	pg3200.txt	pg100.txt
ability	pg3200.txt	pg100.txt	
abilitys	pg100.txt		
abject	pg3200.txt		

The execution time is: 4 minutes, 7 seconds

Application Overview	
User:	cloudera
Name:	SimpleInvertedIndex
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Feb 15 06:03:08 -0800 2017
Elapsed:	4mins, 7sec
Tracking URL:	History
Diagnostics:	

Question c.

How many unique words exist in the document corpus (excluding stop words)? Which counter(s) reveal(s) this information? Define your own counter for the number of words appearing in a single document only. What is the value of this counter? Store the final value of this counter on a new file on HDFS.

The Map-Reduce Framework gives the number of words that exist in the document corpus (excluding stop words) because it gives the Reduce output records. Here: 40691.

Map-Reduce Framework
- Map input records=507535
- Map output records=507535
- Map output bytes=9142116
- Map output materialized bytes=463973
- Input split bytes=375
- Combine input records=0
- Combine output records=0
- Reduce input groups=40691
- Reduce shuffle bytes=463973
- Reduce input records=507535
- Reduce output records=40691
- Spilled Records=1015070
- Shuffled Maps =3
- Failed Shuffles=0
- Merged Map outputs=3
- GC time elapsed (ms)=2368
- CPU time spent (ms)=0
- Physical memory (bytes) snapshot=0

Virtual memory (bytes) snapshot=0
Total committed heap usage (bytes)=773603328

To count the number of words appearing in a single document only I use my own counter as below:

```
public static enum WordsCounter {
      UniqueWord,
    };


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

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

        // to create a HashSet Class with value the file names
        HashSet<String> hs_FileName = new HashSet<String>();
        for (Text val : values) {
            hs_FileName.add(val.toString());
        }

        // to count unique word and store them in a file
        if (hs_FileName.size() == 1) {
            context.getCounter(WordsCounter.UniqueWord).increment(1);
            // to create the new output
            StringBuilder SB = new StringBuilder();
            String separator = " ";
            for (String val : hs_FileName) {
                SB.append(separator);
                SB.append(val);
            }
            context.write(key, new Text(SB.toString()));
        }
    }
}
```

To export the result of the counter I use:

```
job.waitForCompletion(true);

System.out.print("WordsCounter.UniqueWord = " +job.getCounters().findCounter(WordsCounter.UniqueWord).getValue());
```

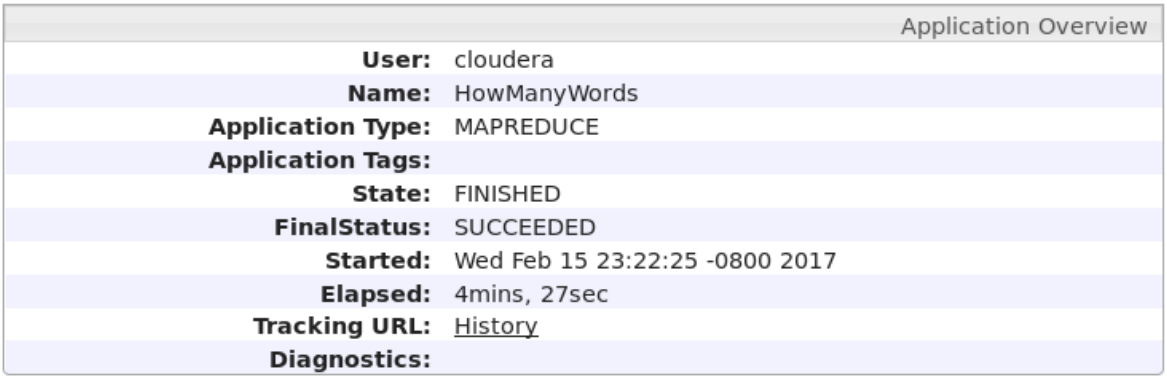
I obtain: WordsCounter.UniqueWord = 29725

The output is in the csv file Result_c.csv and the counter value in the file NumberUniqueWord_c.csv which I then store on HDFS on mkdir “UniqueWord”.

The output is of the form:

aaaamen	pg3200.txt
aachen	pg3200.txt
aaron	pg100.txt
aart	pg3200.txt
abaft	pg3200.txt
abandoned	pg3200.txt
abandons	pg3200.txt
abasement	pg3200.txt
abatement	pg3200.txt
abatfowling	pg100.txt
abating	pg31100.txt
abbess	pg100.txt

The execution time is: 4 minutes, 27 seconds



Application Overview	
User:	cloudera
Name:	HowManyWords
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Feb 15 23:22:25 -0800 2017
Elapsed:	4mins, 27sec
Tracking URL:	History
Diagnostics:	

Question d.

Extend the inverted index of (b), in order to keep the frequency of each word for each document. The new output should be of the form:

this	doc1.txt#1, doc2.txt#1
is	doc1.txt#2, doc2.txt#1, doc3.txt#1
a	doc1.txt#1
program	doc1.txt#1, doc2.txt#1
...	

which means that the word frequency should follow a single '#' character, which should follow the filename, for each file that contains this word. You are required to use a Combiner.

I create a HashMap in the following way to answered this question:

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

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


        HashMap<String, Integer> hs_FileName = new HashMap<String, Integer>();
        for (Text val : values) {
            String[] splitted = val.toString().split(" ");
            for (int i = 0; i < splitted.length; i++) {
                if (!hs_FileName.containsKey(splitted[i])) {
                    hs_FileName.put(splitted[i], 1);
                }
                else {
                    hs_FileName.put(splitted[i], (Integer) hs_FileName.get(splitted[i]) + 1);
                }
            }
        }


        // to create the new output
        StringBuilder SB = new StringBuilder();
        String separator = " ";
        for (Object val : hs_FileName.keySet()) {
            SB.append(separator);
            SB.append(val + "#" + (Integer) hs_FileName.get(val));
        }

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

The results are in the csv file Result_d.csv.

They are of the form:

abhominable	pg100.txt#1		
abhor	pg31100.txt#1	pg100.txt#9	
abhorrd	pg100.txt#3		
abhorred	pg31100.txt#1	pg3200.txt#2	pg100.txt#2
abhorrence	pg31100.txt#6		
abhorrent	pg31100.txt#1		
abhors	pg100.txt#2		
abhorson	pg100.txt#6		
abide	pg31100.txt#2	pg3200.txt#18	pg100.txt#15
abides	pg3200.txt#1	pg100.txt#2	
abideth	pg3200.txt#1		
abiding	pg3200.txt#2		
abiling	pg3200.txt#2		
abilities	pg31100.txt#11	pg3200.txt#6	pg100.txt#1
ability	pg3200.txt#15	pg100.txt#3	

The execution time is: 3 minutes, 6 seconds

Application Overview	
User:	cloudera
Name:	FrequencyInvertedIndex
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Wed Feb 15 06:25:47 -0800 2017
Elapsed:	3mins, 6sec
Tracking URL:	History
Diagnostics:	

I see after that it was ask to do it with a combiner so I rewrite it:

```
job.setMapperClass(Map.class);
job.setReducerClass(Reduce.class);
job.setCombinerClass(Combiner.class);
job.setNumReduceTasks(1);
```

The map doesn’t change.

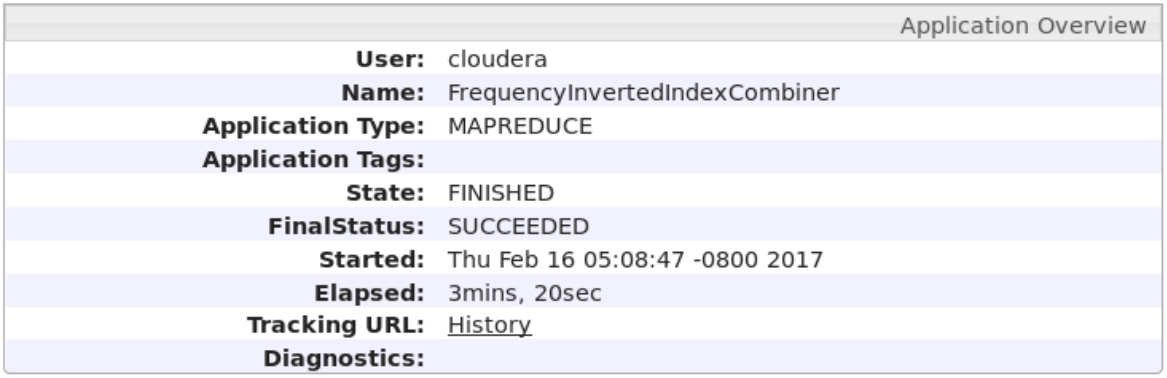
```
public static class Combiner extends Reducer<Text, Text, Text, Text> {
      @Override
      public void reduce(Text key, Iterable<Text> values, Context context)
              throws IOException, InterruptedException {

          HashMap<String, Integer> hs_FileName = new HashMap<String, Integer>();
            for (Text val : values) {
                String[] splitted = val.toString().split(" ");
                for (int i = 0; i < splitted.length; i++) {
                    if (!hs_FileName.containsKey(splitted[i])) {
                        hs_FileName.put(splitted[i], 1);
                    }
                    else {
                        hs_FileName.put(splitted[i], (Integer) hs_FileName.get(splitted[i]) + 1);
                    }
                }
            }
            context.write(key, new Text(hs_FileName.toString().replace("=", "#").replace("}", "").replace("{", "")));
      }
}


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

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

        // to create a HashSet Class with value the file names
        HashSet<String> hs_FileName = new HashSet<String>();
        for (Text val : values) {
            hs_FileName.add(val.toString());
        }

        // to create the new output
        StringBuilder SB = new StringBuilder();
        String separator = " ";
        for (String val : hs_FileName) {
            SB.append(separator);
            SB.append(val);
        }

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

    }
}
```

I obtained exactly the same output which I store in Result_dCombiner.csv:

abhominable	pg100.txt#1		
abhor	pg31100.txt#1	pg100.txt#9	
abhorrd	pg100.txt#3		
abhorred	pg100.txt#2	pg31100.txt#1	pg3200.txt#2
abhorrence	pg31100.txt#6		
abhorrent	pg31100.txt#1		
abhors	pg100.txt#2		
abhorson	pg100.txt#6		

abide	pg31100.txt#2	pg100.txt#15	pg3200.txt#18
abides	pg100.txt#2	pg3200.txt#1	
abideth	pg3200.txt#1		
abiding	pg3200.txt#2		
abiling	pg3200.txt#2		
abilities	pg31100.txt#11	pg100.txt#1	pg3200.txt#6
ability	pg3200.txt#15	pg100.txt#3	

The execution time is: 3 minutes, 20 seconds



This is slightly longer than without the combiner but my computer doesn't run well on VirtualBox (a bit old) so the times are always a bit long.