MASSIVE DATA PROCESSING (Ecole Centrale)

ASSIGNMENT 1 - HADOOP

Raunaq Paul

1. OBJECTIVE

The objective of the assignment is to get used to working with Haddoop Framework and execute wordcount and map reduce jobs on given text files. The following are the questions answered in this assignment:

We 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 (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents.The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database. The inverted file may be the database file itself, rather than its index. It is the most popular data structure used in document retrieval systems,used on a large scale for example in search engines.

It 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):

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

Before the inverted index, we will also be executing wordcount jobs in order to find StopWords or frequently appearing words which do not affect the content of the documents much e.g. articles.

2. Initialization

First we create a new workspace called MDP_Assignment1 from the Cloudera Terminal. Then, we create an input and output directory in our workspace:

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

We then download the text files and put the, in the input directory of our MDP_Assignment1 workspace. We can now put the input text files into Hadoop:

```
[cloudera@quickstart ~]$ cd workspace/MDP_Assignment1
[cloudera@quickstart MDP_Assignment1]$ hadoop fs -mkdir input
[cloudera@quickstart MDP_Assignment1]$ hadoop fs -put input/pg100.txt input
[cloudera@quickstart MDP_Assignment1]$ hadoop fs -put input/pg3200.txt input
[cloudera@quickstart MDP_Assignment1]$ hadoop fs -put input/pg31100.txt input
[cloudera@quickstart MDP_Assignment1]$ hadoop fs -ls input
Found 3 items
-rw-r--r--   1 cloudera cloudera    5589917 2017-02-16 05:09 input/pg100.txt
-rw-r--r--   1 cloudera cloudera    4454075 2017-02-16 05:11 input/pg31100.txt
-rw-r--r--   1 cloudera cloudera   16013958 2017-02-16 05:10 input/pg3200.txt
```

We are now ready to start creating our jobs for the assignment.

2.1. Question: StopWords

Run a MapReduce 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.

We will now implement our map reduce program. We will try to find the stopwords in our corpus by finding the wordcounts for the entire corpus first and then separating the words with frequency > 4000 a "stopwords" by adding the condition.

For this, we reused the code from Standford http://snap.stanford.edu/class/cs246-data-2014/WordCount.java given in the CS246: Mining Massive Datasets Hadoop tutorial.

```
package edu.stanford.cs246.wordcount;

import java.io.IOException;
import java.util.Arrays;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {
   public static void main(String[] args) throws Exception {
      System.out.println(Arrays.toString(args));
      int res = ToolRunner.run(new Configuration(), new WordCount(), args);

      System.exit(res);
   }

   @Override
   public int run(String[] args) throws Exception {
      System.out.println(Arrays.toString(args));
      Job job = new Job(getConf(), "WordCount");
      job.setJarByClass(WordCount.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      job.setMapperClass(Map.class);
      job.setReducerClass(Reduce.class);

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

```
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.waitForCompletion(true);

        return 0;
    }

    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);
                context.write(word, ONE);
            }
        }
    }

    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();
            }
            context.write(key, new IntWritable(sum));
        }
    }
}
```

The changes made to the above code are as follows:

- Added the following code to the Main Class (Driver):

  ```
  job.getConfiguration().set("mapreduce.output.textoutputformat.separator", ",");
  ```

  The above code creates our output in the form of a csv format as needed per the task.

- toLowerCase() method and replaceAll("[^a-zA-Z]", "") in the words iteration in the Mapper function in order to remove redundancy/duplicates and only have alphabetical outputs:

```
public static class StubMapper 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().replaceAll("[^a-zA-Z ]'
                                {
                                   word.set(token.toLowerCase());
                                   context.write(word, ONE);
                                }
             }
        }
```

- if statement in the Reducer class that selects only the stopwords (words with Frequency>4000), before writing the (key,value) outputs:

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

    }
```

After this , we execute the code and export the Jar file and run it using the following hadoop job code:

```
hadoop jar MDP_Assignment1.jar stopwords.stopwords input output
```

We can now see the results located in the HDFS output directory by using the following command in the terminal:

```
hadoop fs -ls output
Found 2 items
-rw-r--r--   1 cloudera cloudera          0 2017-02-16 05:14 output/_SUCCESS
```

```
-rw-r--r--   1 cloudera cloudera       1333 2017-02-16 05:14 output/part-r-00000
```

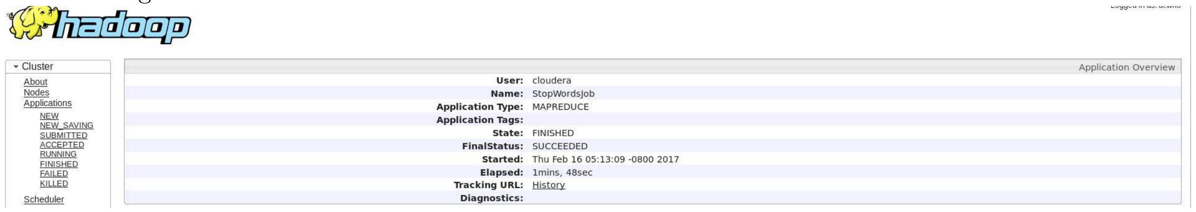
We now Merge the output file to the output folder in our workspace/MDP_Assignment into stopwords.csv

```
hadoop fs -getmerge output output/stopwords.csv
```

The stopwords.csv is in the following format:

```
a,100360
about,7928
after,4651
again,4900
all,21349
am,6747
```

The following is the log execution time which can be seen in the Hadoop Yarn Resource Manager:



User: cloudera
Name: StopWordsJob
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 05:13:09 -0800 2017
Elapsed: 1mins, 48sec
Tracking URL: History
Diagnostics:

Now we answer the following questions as per the tasks:

2.1.1. Use 10 reducers and do not use a combiner. Report the execution time.

The number of reducers is defined in the Hadoop driver configuration (main class) by the method: job.setNumReduceTasks(). In this case, we use: job.getConfiguration().set(“mapreduce.o
“,”); job.setNumReduceTasks(10);

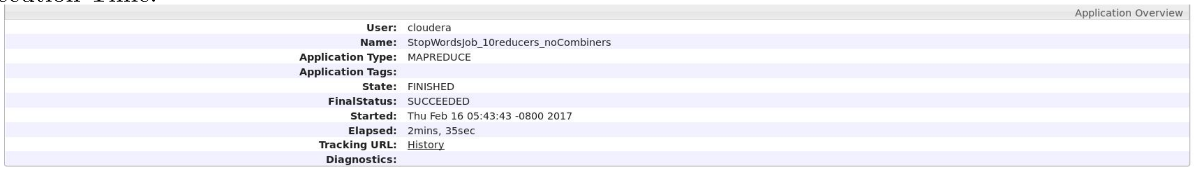
The following is the Hadoop Job code in the terminal:

```
hadoop jar MDP_Assignment1.jar stopwords.stopwords2 input output
hadoop fs -ls output
Found 11 items
-rw-r--r--   1 cloudera cloudera          0 2017-02-16 05:46 output/_SUCCESS
-rw-r--r--   1 cloudera cloudera        100 2017-02-16 05:45 output/part-r-00000
-rw-r--r--   1 cloudera cloudera        219 2017-02-16 05:45 output/part-r-00001
-rw-r--r--   1 cloudera cloudera        160 2017-02-16 05:45 output/part-r-00002
-rw-r--r--   1 cloudera cloudera        121 2017-02-16 05:45 output/part-r-00003
-rw-r--r--   1 cloudera cloudera         68 2017-02-16 05:45 output/part-r-00004
-rw-r--r--   1 cloudera cloudera        138 2017-02-16 05:45 output/part-r-00005
-rw-r--r--   1 cloudera cloudera        139 2017-02-16 05:46 output/part-r-00006
-rw-r--r--   1 cloudera cloudera        139 2017-02-16 05:46 output/part-r-00007
-rw-r--r--   1 cloudera cloudera        182 2017-02-16 05:46 output/part-r-00008
```

```
-rw-r--r--   1 cloudera cloudera         67 2017-02-16 05:46 output/part-r-00009
hadoop fs -getmerge output output/stopwords2.csv
```

The csv file stopwords2 consists of all words except the ones in stopwords.csv

```
about,7928
be,28477
before,5630
by,20273
her,28370
```

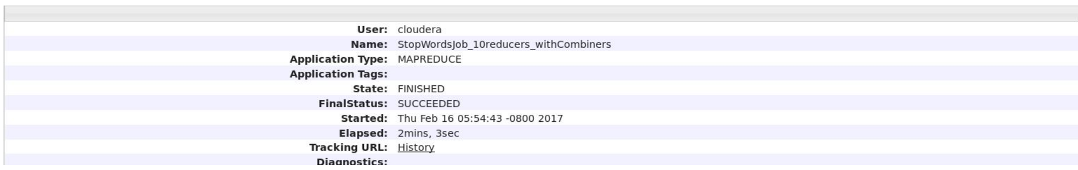
Execution Time:

Application Overview

User: cloudera
Name: StopWordsJob_10reducers_noCombiners
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 05:43:43 -0800 2017
Elapsed: 2mins, 35sec
Tracking URL: History
Diagnostics:

2.1.2. 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?

We can run the same program as in 2.1.1 but this time adding the following code in the configuration class:

```
job.setCombinerClass(StubReducer.class);
```


User: cloudera
Name: StopWordsJob_10reducers_withCombiners
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 05:54:43 -0800 2017
Elapsed: 2mins, 3sec
Tracking URL: History
Diagnostics:

Execution Time:
There is a difference in execution time as the program with combiner takes 32 seconds less.

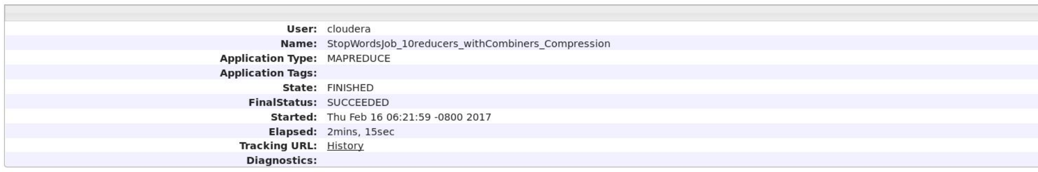
The Combiner is a semi-reducer.The Combiner class is used in between the Map class and the Reduce class to reduce the volume of data transfer between Mapper and Reducer.It does a partial reduction at the map nodes before transfering the data to the reduction phase. Since Reduction cannot perform parallelization but the mapper can, the combiner decreases the data to be reduced by the Reducer class and thus leads to a decrease in execution time.

2.1.3. 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?

We run the same program in 2.1.2 but add the following code in the configuration class:

```
FileOutputFormat.setCompressOutput(job, true);
        FileOutputFormat.setOutputCompressorClass(job,
                org.apache.hadoop.io.compress.BZip2Codec.class);
```

The BZip2 codec has been chosen for this instance. bzip2 is a free and open-source file compression program that uses the Burrows–Wheeler algorithm. It only compresses single files and is not a file archiver. It compresses data in blocks of size between 100 and 900 kB and uses the Burrows–Wheeler transform to convert frequently-recurring character sequences into strings of identical letters. It then applies move-to-front transform and Huffman coding. There are other codecs which can be used like: Snappy, LZO , GZip. BZip2, LZO, and Snappy formats are splittable, but GZip is not.

Execution Time: 

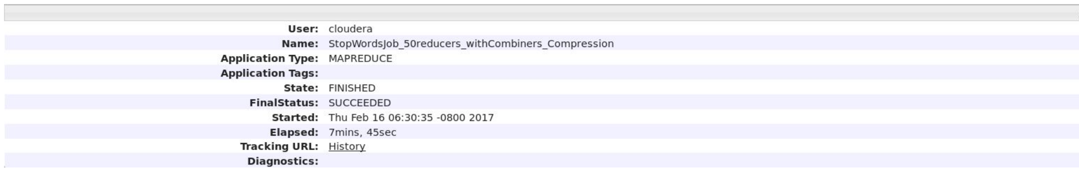
```
User:               cloudera
Name:               StopWordsJob_10reducers_withCombiners_Compression
Application Type:   MAPREDUCE
Application Tags:
State:              FINISHED
FinalStatus:        SUCCEEDED
Started:            Thu Feb 16 06:21:59 -0800 2017
Elapsed:            2mins, 15sec
Tracking URL:       History
Diagnostics:
```

There is a slight increase of 13 seconds in the execution time which is due to the time taken for compression.

2.1.4. 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?

We make the following change to the code in 2.1.3

```
job.setNumReduceTasks(50);
```

Execution Time: 

```
User:               cloudera
Name:               StopWordsJob_50reducers_withCombiners_Compression
Application Type:   MAPREDUCE
Application Tags:
State:              FINISHED
FinalStatus:        SUCCEEDED
Started:            Thu Feb 16 06:30:35 -0800 2017
Elapsed:            7mins, 45sec
Tracking URL:       History
Diagnostics:
```

There is a huge difference compared to the execution time. As there are 50 reducers and since Reducers dont implement parallelization, it takes lot more time to run 50 of them and also having compression done too.

2.2. Question: Inverted Index

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

Now, we have to make changes to the Map class for this. Since it is an inverted index, for each word in the corpus the program needs to generate the output : (word,list of file names). So we set both key and value to Text class.

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

Now our program needs to not only create the inverted index but also create it without using the words in the stopwords list. Thus we need to read the words from the stopwords.csv file. The stopwords.csv has been converted into stopwords.txt to make it simpler to execute:

```
a
about
after
again
all
```

We also changed the output config format from csv :

```
job.getConfiguration().set(
                "mapreduce.output.textoutputformat.separator", " -> ");
```

We use the FileSplit function to get the filenames and the String Tokenizer to split. The BufferedReader creates Reader object to read from the "stopwords.txt" file. The following is the mapper class code:

```
public static class StubMapper 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/MDP_Assignment1/stopwords.txt'
            String line;
            while ((line = Reader.readLine()) != null) {
                stopwords.add(line.toString().toLowerCase());
            }
            Reader.close();
            String filenameStr = ((FileSplit) context.getInputSplit()).getPath().getName();
            filename = new Text(filenameStr);
            StringTokenizer tokenizer = new StringTokenizer(value.toString().replaceAll("[\\
            //System.out.print(tokenizer);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken().toString().toLowerCase());
                context.write(word,filename);
            }
        }
```

```
    }
```

The above mapper class will give a (key, value) pair output of the form (word, filename):

```
word1, doc1.txt
word1, doc1.txt
word1, doc1.txt
word1, doc2.txt
word2, doc1.txt
word2, doc2.txt
word2, doc2.txt
word2, doc3.txt
```

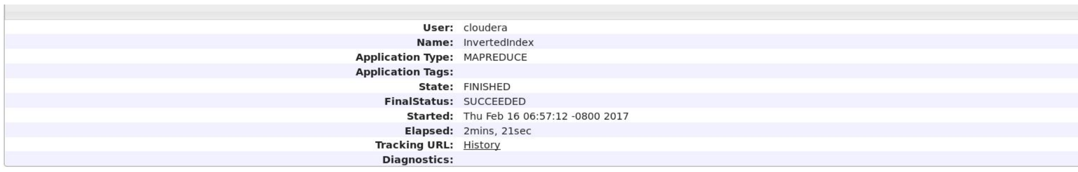
The following reducer class then stores all the values of the filenames for each word in a HashSet. The HashSet is used to avoid duplicates in case of the words.

```java
    public static class StubReducer extends Reducer<Text, Text, Text, Text> {

        @Override
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            HashSet<String> rehash = new HashSet<String>();
            for (Text value : values) {
                rehash.add(value.toString());
            }
            StringBuilder builder = new StringBuilder();
            String prefix = ", ";
            for (String value : rehash) {
                builder.append(prefix);
                builder.append(value);
            }
            context.write(key, new Text(builder.toString()));
        }
    }
```

The Reducer class gives a output of the form (word, collection of filenames) in the following format:

```
a -> pg31100.txt, pg3200.txt, pg100.txt
aachen -> pg3200.txt
aar -> pg3200.txt
aaron -> pg3200.txt, pg100.txt
```

Execution Time: 

User:	cloudera
Name:	InvertedIndex
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Thu Feb 16 06:57:12 -0800 2017
Elapsed:	2mins, 21sec
Tracking URL:	History
Diagnostics:	

2.2.2. 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.

We first add the following counter to the previous program in 2.2.1 :

```
public static enum COUNTER {
        UniqueWords,
    };
```

The mapper function will remain the same as we are not doing further selective mapping. We will have to make changes to our reudcer function so that only unique words are selected. UniqueWords are the words which are present in only one file and not in multiple ones. Thus the size of filelist for such words will be 1.

The following is our Reducer Class: There is a if statement which only count the words whose filename size is 1. The increment function increments the counter by 1 for every unique word.

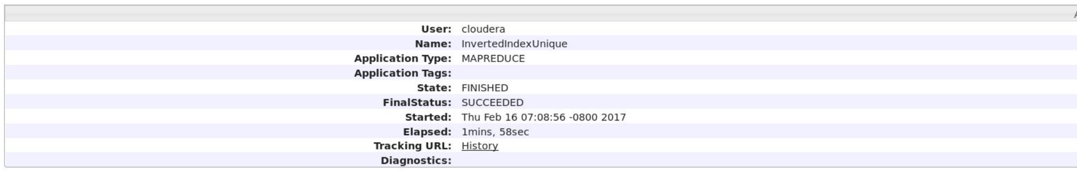
```
public static class StubReducer extends Reducer<Text, Text, Text, Text> {

        @Override
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            HashSet<String> rehash = new HashSet<String>();
            for (Text value : values) {
                rehash.add(value.toString());
            }
            if (rehash.size() == 1) {
                context.getCounter(COUNTER.UniqueWords).increment(1);
            StringBuilder builder = new StringBuilder();
            String prefix = ", ";
            for (String value : rehash) {
                builder.append(prefix);
                builder.append(value);
            }
            context.write(key, new Text(builder.toString()));
        }
        }
    }
```

After executing the Hadoop Job: We can see the Counter details in the job summary:-

```
hadoop jar MDP_Assignment1.jar invertedindex.InvertedIndexUnique input output
//in the job summary
  invertedindex.InvertedIndexUnique$COUNTER
```

```
        UniqueWords=35726
```

Execution Time: 

```
            User:  cloudera
            Name:  InvertedIndexUnique
Application Type:  MAPREDUCE
Application Tags:
           State:  FINISHED
     FinalStatus:  SUCCEEDED
         Started:  Thu Feb 16 07:08:56 -0800 2017
         Elapsed:  1mins, 58sec
    Tracking URL:  History
     Diagnostics:
```

2.2.3. 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:

word	Filename and Frequency
this	doc1.txt#1, doc2.txt#1
is	doc1.txt#2, doc2.txt#1, doc3.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.

In this case we can extend the Mapper function from the previous program with the only difference being:

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

        @Override
        public void map(Object key, Text value, Context context)
```

Now, we have to use a combiner which will partially reduce the function before it is passed onto the actual reducer. The combiner extends the reducer class. The Reducer class is as follows:

```
public static class StubReducer extends Reducer<Text, Text, Text, Text> {
    private  Text combo=new Text(); // takes the combination of filename and frequency
        @Override
        public void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {
            String result=new String();
            Map<String,Integer> nmap=new HashMap<String,Integer>(); //map the dataset
            String Filename=new String();
            Integer count=new Integer(0);
            for (Text value : values) {
                String[] s=value.toString().split("#");
                Filename=s[0];
                count=new Integer(s[1]);
                if(nmap.containsKey(Filename))
                {
                    nmap.put(Filename, nmap.get(Filename)+count);
                }
                else
                {
                    nmap.put(Filename, count);
                }
            }
            for (String v :nmap.keySet())
            {
             result+=","+v+"#"+nmap.get(Filename).toString();
            }
     combo.set(result);
         context.write(key, combo);

        }
    }
```

The Combiner Class is as follows:

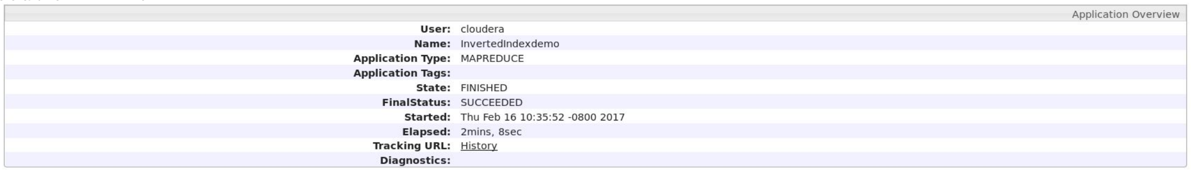
```
public static class StubCombiner extends Reducer<Text,Text,Text,Text>
    {   private Text combo= new Text();
        public void reduce(Text key, Iterable<Text> values, Context context) //Iterable to c
                throws IOException, InterruptedException {
            Integer frequency=new Integer(0);
            String fname= new String();

            for(Text val:values)
            {
                frequency=frequency+1;   //for each word in a document increment the frequen
                if(frequency==1)
                {
                    fname=val.toString();
                }
            }
            combo.set(fname+"#"+frequency.toString());  //joining the document and frequency
            context.write(key,combo);
        }
    }
```

The logic behind using the combiner is to have partial reduction at the Mapper phase thus reducing the workload for the reducer class. The output csv file is as follows:

```
a -> ,pg31100.txt#73628,pg3200.txt#73628,pg100.txt#73628
aachen -> ,pg3200.txt#1
aar -> ,pg3200.txt#3
aaron -> ,pg3200.txt#97,pg100.txt#97
```

Execution Time:



	Application Overview
User:	cloudera
Name:	InvertedIndexdemo
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	Thu Feb 16 10:35:52 -0800 2017
Elapsed:	2mins, 8sec
Tracking URL:	History
Diagnostics:	

3. CONCLUSION

The overview of the Stopwords portion of the assignment:

Method	**Runtime (in secs)**
Stopwords	108
10 Reducers and no combiners	155
10 Reducers and a combiner	123
10 Reducers,combiner,compression	135
50 Reducers,combiner,compression	465

We can conclude that due to lack of parallelization in case of a Reducer, the higher the number of reducers the larger is the execution time. Compression also affects the execution time depending on the type(codec). Combiners are used for optimizing and thus reduces the execution time.

The Runtime Overview of the InvertedIndex jobs:

Method	**Runtime(in secs)**
Inverted Index Simple	141
Inverted Index Unique Words	118
Inverted Index Word Frequency using Custom Combiner	128

The Inverted Index jobs were simple version of what can be used in search engine implementation and optimization. Even though the Inverted Index with Frequency has more complexity than the simple Inverted Index job, the use of a combiner reduces the runtime.