Massive Data Processing 1st Assignment

Settings of the virtual machine and eclipse environment

I install the cloudera 5.8.0 rather than 5.5.0 because of certain unresolved bugs in 5.5.0
https://downloads.cloudera.com/demo_vm/virtualbox/cloudera-quickstart-vm-5.8.0-0-virtualbox.zip

Libraries from this website are also imported:
https://drive.google.com/open?id=0B8S5KiDRR6Y-cXhtUnFnZWRoRnM

I also create the input and output folders under the project.

It has to be declared that a considerable part of the code uses the Hadoop tutorial code from the course of Stanford university *CS246: Mining Massive Datasets Hadoop tutorial* for reference:

```
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));
      }
   }
}
```

Some modifications on the codes are also inspired by https://github.com/rohitsden/wordcount:

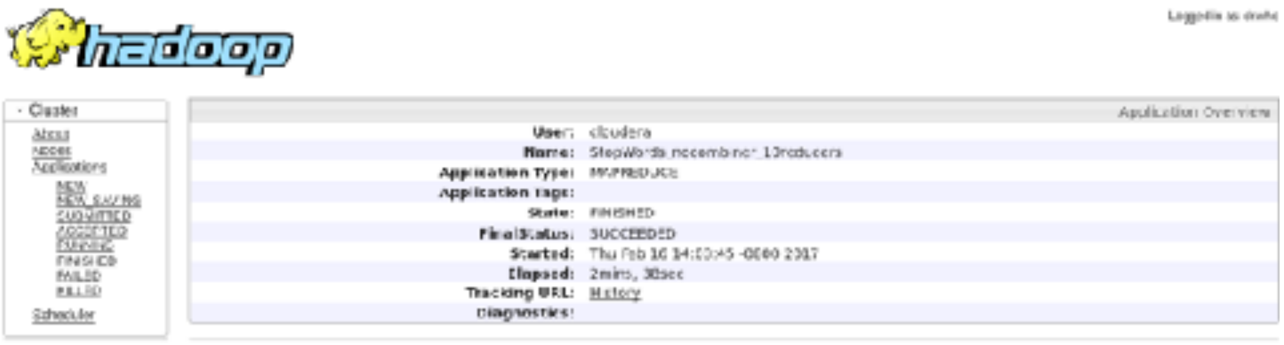
```
public class WordCountMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable>{

      public void map(LongWritable key, Text value, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {

            StringTokenizer st = new
StringTokenizer(value.toString().toLowerCase());

            while(st.hasMoreTokens()) {

                  output.collect(new Text(st.nextToken()), new IntWritable(1));
            }
      }
}

public class WordCountReducer extends MapReduceBase implements Reducer<Text,
IntWritable, Text, IntWritable>{
      public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text,IntWritable> output, Reporter reporter) throws IOException
{

            int count = 0;

            while(values.hasNext()) {

                  count += values.next().get();

            }
            output.collect(key, new IntWritable(count));
      }
}
```

Question a)

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.

i. Use 10 reducers and do not use a combiner. Report the execution time.
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?
iii. Run the same program again, this time compressing the intermediate results of map (using any code you wish). Report the execution time. Is there any difference in the execution, time compared to the previous execution? Why?
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) The number of reducers is controlled by `job.setNumReduceTasks()` and there are 120 lines of output



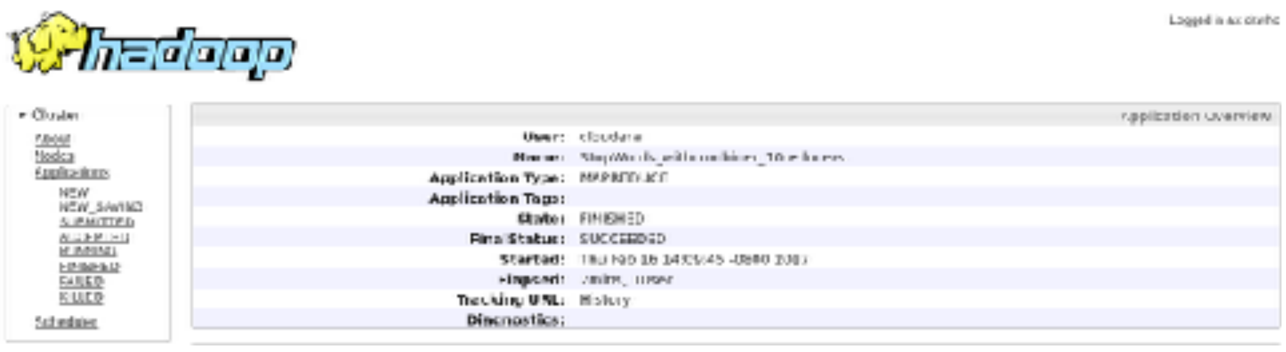
User: cloudera
Name: StopWords nocombiner 10reducers
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 14:00:45 -0800 2017
Elapsed: 2mins, 30sec
Tracking URL: History
Diagnostics:

```
about,7350

be,27239

before,4219

by,19509

her,24277
```

```
mr.,4742

much,4712

old,4092

up,8608

where,4197

......
```

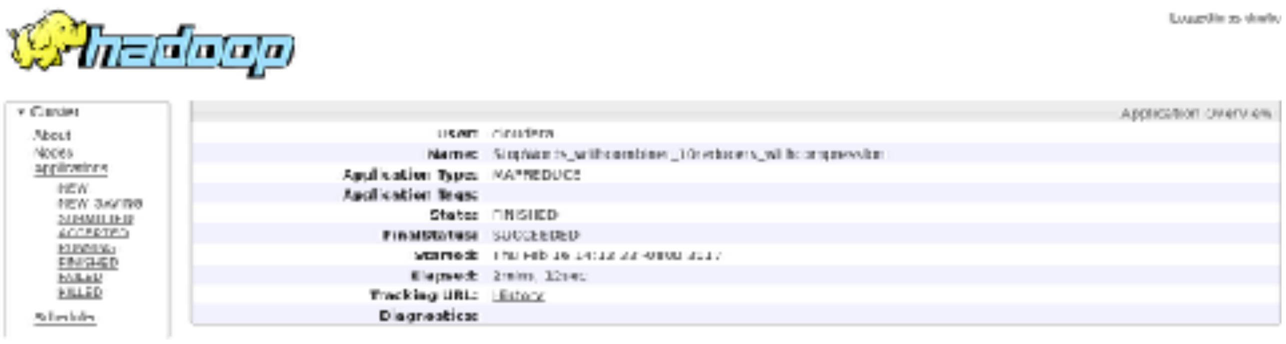
ii) We still use the above code and add only `job.setCombinerClass(Reduce.class)` in the hadoop driver configuration:

It is quicker than the above one.

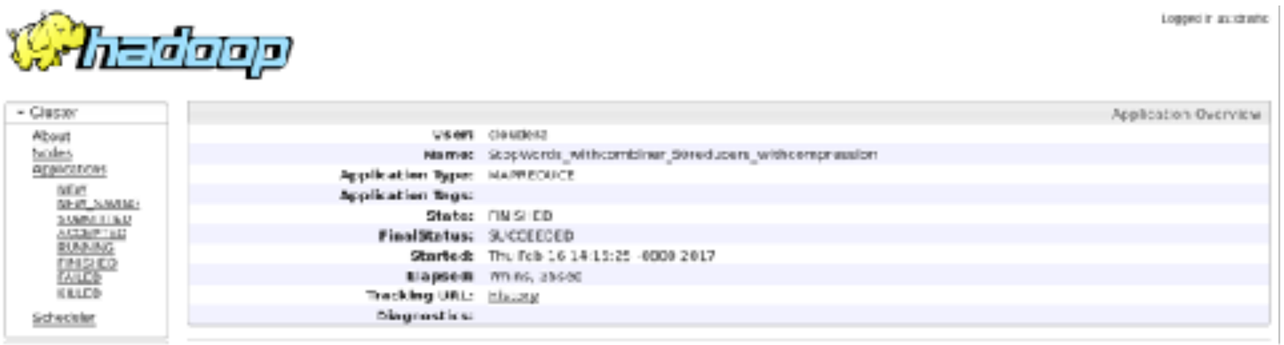


```
User: cloudera
Name: StopWords_withcombiner_10reducers
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 14:05:45 -0800 2017
Elapsed: [illegible]
Tracking URL: History
Diagnostics:
```

iii) We use `setCompressOutput()` of the `org.apache.hadoop.mapreduce.lib.output.FileOutputFormat` class. It is a little slower compared to the previous one.



```
User: cloudera
Name: StopWords_withcombiner_10reducers_withcompression
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 14:12:22 -0800 2017
Elapsed: 2mins, 32sec
Tracking URL: History
Diagnostics:
```



```
User: cloudera
Name: StopWords_withcombiner_5reducers_withcompression
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: Thu Feb 16 14:15:25 -0800 2017
Elapsed: [illegible]
Tracking URL: History
Diagnostics:
```

iv) We add the code job.setNumReduceTasks(50) in the hadoop configuration. There is a large increase in terms of the time required to run the program.

Question b)

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

As described in the question and as recommended online, for every word in the documents, the program needs to output a (key, value) pair of the form (word, collection of filenames). To realise such requirement, we can set the output format of both keys and values as Text in the Hadoop driver in advance:

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

Our map function will get, for each word, the filename in which it appears. Also, it will have to remove the stopwords.

The format can be changed quickly from the one followed in stopwords.csv output file of the previous MapReduce program to this more convenient stopwords.txt:

The code is below:

```
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);

    }
}
```

The following code help reduce duplication:

```
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 (first 10 lines) can be seen below:

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 code is attached here:

```
…import…

public class InvertedIndex_unique extends Configured implements Tool {

      public static enum CUSTOM_COUNTER {
            UNIQUE_WORDS,
      };
      public static void main(String[] args) throws Exception {
            System.out.println(Arrays.toString(args));
            int res = ToolRunner.run(new Configuration(),
                        new InvertedIndex_unique(), args);
            System.exit(res);
      }

      @Override
      public int run(String[] args) throws Exception {

            System.out.println(Arrays.toString(args));

            Job job = new Job(getConf(), "InvertedIndex_simple");

            job.setJarByClass(InvertedIndex_unique.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(Text.class);
            job.setMapperClass(Map.class);
            job.setReducerClass(Reduce.class);
            job.setInputFormatClass(TextInputFormat.class);
            job.setOutputFormatClass(TextOutputFormat.class);
            job.getConfiguration().set(

                        "mapreduce.output.textoutputformat.separator", " -> ");

            job.setNumReduceTasks(1);

            FileInputFormat.addInputPath(job, new Path(args[0]));
            FileOutputFormat.setOutputPath(job, new Path(args[1]));
            FileSystem fs = FileSystem.newInstance(getConf());
            if (fs.exists(new Path(args[1]))) {
                  fs.delete(new Path(args[1]), true);
            }

            job.waitForCompletion(true);
            return 0;
      }
      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);
              }
        }
        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()))
                    }
              }
        }
}
```

In the driver, we also add the custom counter:

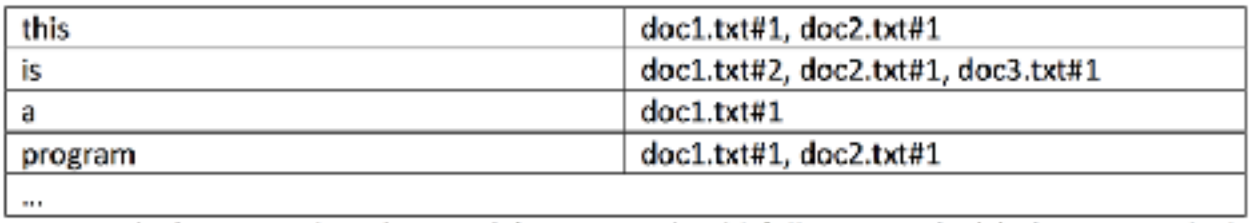
```
public static enum CUSTOM_COUNTER {
      UNIQUE_WORDS,
    };
```

Some of the output can be seen below:

```
 -> pg31100.txt
! -> pg3200.txt
!" -> pg3200.txt
"'fnobjectionbilltakuzhlcoixrssoreferred!'" -> pg3200.txt
"'i -> pg31100.txt
"'journal'" -> pg3200.txt
"'kahkahponeeka'?" -> pg3200.txt
"'nothing.' -> pg3200.txt
"'of -> pg3200.txt
"'pears -> pg3200.txt
"'tis -> pg31100.txt
"'yes.' -> pg3200.txt
") -> pg3200.txt
```

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.

We add this part to the code:

```
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()));

    }
}
```

Some of the output can be seen below:

```
 -> 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
```