Aims

This exercise aims to get you to apply the design patterns you have learned in Chapter 3 on MapReduce programming.

Background

Create a project “Lab3” in Eclipse, and create a package “comp9313.lab3” in this project. Put all your codes written in this week’s lab in this package, and keep a copy for yourself after you have finished all the problems.

Download the input file “pg100.txt” from the following link:

https://webcms3.cse.unsw.edu.au/COMP9313/18s1/resources/15854

For all problems, using the following code to tokenize a line of document:

```
StringTokenizer itr = new StringTokenizer(value.toString(),
        " *$&#/\t\n\f\"'\\,.:;?![](){}<>~-_");
```

Documents will be split according to all characters specified (i.e., `" *$&#/\t\n\f\"'\\,.:;?![](){}<>~-_"`), and higher quality terms will be generated.

Convert all terms to lower case as well (by using toLowerCase() function).

Apply this to WordCount.java we used in Lab2:

a) Start HDFS and YARN
```
$ $HADOOP_HOME/sbin/start-all.sh
```
b) Put the input file to HDFS by:
```
$ $HADOOP_HOME/bin/hdfs dfs -mkdir input
$ $HADOOP_HOME/bin/hdfs dfs -put ~/pg100.txt input
```
c) Create a new class WordCount.java in the package comp9313.lab3. The file can be downloaded at WebCMS3/Lab2.

d) Use the new method to tokenize a line of document

e) Run the code in Eclipse: Right click the class->Run As->Run Configuration->Right click Java Application and select New. Then specify the arguments: “hdfs://localhost:9000/user/comp9313/input hdfs://localhost:9000/user/comp9313/output”

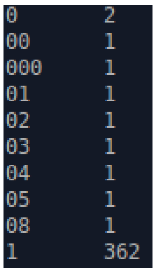
f) Remember to delete to output folder if it exists

g) If you forget the details, please refer to the instructions for Lab2.

Type the following command in the terminal:

```
$ hdfs dfs -cat output/* | head
```

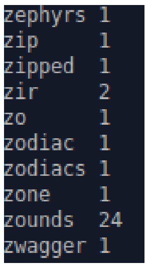
You should see results:


```
0       2
00      1
000     1
01      1
02      1
03      1
04      1
05      1
08      1
1       362
```

Use the following command:

```
$ hdfs dfs -cat output/* | tail
```

You should see:


```
zephyrs 1
zip     1
zipped  1
zir     2
zo      1
zodiac  1
zodiacs 1
zone    1
zounds  24
zwagger 1
```

If you are not sure about the correctness of your codes, compare with the file WordCount.java at:

https://webcms3.cse.unsw.edu.au/COMP9313/18s1/resources/14985

Problem 1. Improve WordCount

Based on the WordCount.java we used in Lab2, you are required to write an improved version using "in-mapper combining". The input file is still the one we used in Lab 2.

Create a new class "WordCount2.java" in the package "comp9313.lab3" and solve this problem. Your results should be the same as generated by WordCount.java.

Hints:

1. Refer to the pseudo-code shown in slide 10 of Chapter 3.
2. You need to use a HashMap in the mapper class to buffer the partial results for different calls of the map() function. You can define an object of type HashMap<String, Integer> for this problem.
   If you are not familiar with HashMap, the following pages may be helpful:
   https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html

http://www.java67.com/2013/02/10-examples-of-hashmap-in-java-programming-tutorial.html
http://www.java2novice.com/java-collections-and-util/hashmap/

For example, in order to iterate the key-value pairs in a HashMap, you can do like below:

```
Set<Entry<String, Integer> > sets = map.entrySet();
for(Entry<String, Integer> entry: sets){
      //entry.getKey() to get the key
      //entry.getValue() to get the value
}
```

3. The results are not emit in map() now. Rather, you will need to override the cleanup() function in the mapper class to generate the key-value pairs. The cleanup() function is defined like:

```
public void cleanup(Context context) throws IOException,
InterruptedException {
        //generate the output of the mapper
        … …
}
```

This function will be called once at the end of the map task. You do not need to call this function, and Hadoop will do it (just like the function map()). More usage of cleanup() please refer to:
https://hadoop.apache.org/docs/r2.7.2/api/org/apache/hadoop/mapreduce/Mapper.html

You mapper class will be like:

```
public static class TokenizerMapper extends Mapper<Object,
Text, Text, IntWritable> {
        … …
       private static HashMap<String, Integer> map = new
HashMap<String, Integer>();

       public void map(Object key, Text value, Context context)
throws IOException, InterruptedException {
            //Your map function
       }

       public void cleanup(Context context) throws IOException,
InterruptedException {
            //iterate the HashMap object, and send the results
to the reducer
       }
}
```

4. Do you need to change the reducer?
5. How to specify the configuration in the main function?

6. **Note:** All Java primitives are wrapped by the Writable class in Hadoop. Your key-in, key-out must be an instance of Writable! You cannot do below:

   ```
   context.write(a String object, an Integer object)
   ```

   Instead, you need to wrap the String object by Text, and the Integer object by IntWritable, and then:

   ```
   context.write(a Text object, an IntWritable object)
   ```

Problem 2. Compute a Nonsymmetric Term Co-occurrence Matrix Using the "Pair" Approach

The problem is to compute the number of co-occurrence for each pair of terms (w, u) in the document. In this problem, the co-occurrence of (w, u) is defined as: u appears after w in a line of document. This means that, the co-occurrence counts of (w, u) and (u, w) are **different!** The task is to use the "pair" approach to solve this problem.

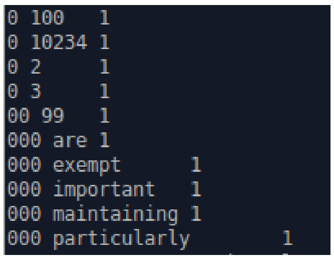
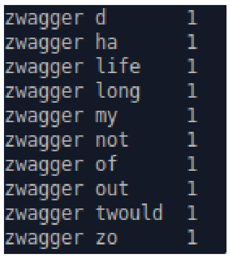
Input is in format of (line number, line). Output is in format of ((w, u), co-occurrence count).

Create a new class CoTermNSPair.java in package "comp9313.lab3".

Hints:

1. Refer to the pseudo-code in slide 22 of Chapter 3. Note that the condition u is in "NEIGHBORS(w)" means that u appears after w in the same line in this problem.
   For example, give a line "a, b, c, d", for term a, you will generate ((a, b), 1), ((a, c), 1), and ((a, d), 1).
2. What is the data type of the map output key? How to store the pair of terms? (A simple method is to concatenate the two terms as a string)
3. How to write the reducer?
4. How about the combiner?
5. How to configure the job in main function?

The head and the tail of the result are like:


```
0 100   1
0 10234 1
0 2     1
0 3     1
00 99   1
000 are 1
000 exempt      1
000 important   1
000 maintaining 1
000 particularly        1
```

```
zwagger d       1
zwagger ha      1
zwagger life    1
zwagger long    1
zwagger my      1
zwagger not     1
zwagger of      1
zwagger out     1
zwagger twould  1
zwagger zo      1
```

If your code is correct, you should output 1,161,210 pairs.

Problem 3. Compute a Nonsymmetric Term Co-occurrence Matrix Using the "Stripe" Approach

The problem is the same as Problem 2. The task is to use the "stripe" approach to solve it.

Input is in format of (line number, line). Output is in format of ((w, u), co-occurrence count).

Create a new class CoTermNSStripe.java in package "comp9313.lab3".

Hints:

1. Refer to the pseudo-code in slide 25 of Chapter 3.
2. You need to use MapWritable as the map output value. MapWritable is a wrapper for java Map in Hadoop.
   When processing a line in the map() function, in the first for loop, you need to create a MapWritable object, and for all terms appearing after the current term w, you need to use the MapWritable object to store the information.

   More usage about MapWritable please refer to:
   https://hadoop.apache.org/docs/r2.7.2/api/org/apache/hadoop/io/MapWritable.html
   Some MapWritable examples can be found at:
   http://www.programcreek.com/java-api-examples/index.php?api=org.apache.hadoop.io.MapWritable

3. In the reducer, you will receive a list of MapWritable objects for the same term w. You need to aggregate them and generate a final "stripe", and then output the key-value pairs in format of ((w, u), co-occurrence count). You can use the following code to iterate the key-value pairs in a MapWritable object:

```
for (MapWritable val : values) {
        Set<Entry<Writable, Writable> > sets = val.entrySet();
        for(Entry<Writable, Writable> entry: sets){
                //entry.getKey() to get the key
                //entry.getValue() to get the value
                ……
        }
}
```

4. The mapper output is <Text, MapWritable>, while the reducer output is <Text, IntWritable>, and the output value types are different. Therefore, you need to specify this in the main function, by adding the following line:

   ```
   job.setMapOutputValueClass(MapWritable.class)
   ```

5. How to write the combiner? Can you use the reducer as the combiner in this problem? Remember, the input of the combiner is the output of the mapper, and the output of the combiner is the input of the reducer. If you need to write a combiner class, define it as:

   ```
   public static class YOURCOMBINERCLASS extends Reducer<Text,
   MapWritable, Text, MapWritable>
   ```

   Then, in this class, you need to override the function reduce(), which is similar as in the reducer class.

If your codes are correct, the results obtained should be the same as obtained in Problem 2.

Solutions of the Three Problems

I hope that you are able to finish all problems by yourself, since the hints are already given.

All the source codes will be published in the course homepage on Friday next week.