Hands-on tutorial
Algorithms in MapReduce

1. Here are some handy commands to remember:

Initialize your Hadoop environment variables
```
. /usr/local/bin/hadoop_env
```

Create a repository in your HDFS space
```
hadoop fs -mkdir /user/username/
```

Load a file into your HDFS space
```
hadoop fs -put file /user/username
```

List the content of your HDFS space
```
hadoop fs -ls /user/username
```

2. 1st program: WordCount

Our first program consists of one class (*WordCount*) with two inner classes (*TokenizerMapper*, *IntSumReducer*). Please have a look at their code, and only after that copy/paste it into a separate file that will be called *WordCount.java*. Then move on to the instructions that follow.

WordCount.java

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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.output.FileOutputFormat;

public class WordCount {

   public static class TokenizerMapper extends Mapper <Object, Text, Text, IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map (Object key, Text value, Context context) throws IOException, InterruptedException {
         StringTokenizer itr = new StringTokenizer(value.toString());
         while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
         }
      }
   }

   public static class IntSumReducer extends Reducer <Text, IntWritable, Text, IntWritable> {
      private IntWritable result = new IntWritable();

      public void reduce(Text key, Iterable <IntWritable> values, Context context) throws IOException, InterruptedException {
         int sum = 0;
         for (IntWritable val: values) {
            sum += val.get();
         }
```

```
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Instructions:

01. Initialize your Hadoop environment variables:
```
. /usr/local/bin/hadoop_env
```

02. Compile the Java classes with this command:
```
javac -cp $HADOOP_CLASSPATH -d . WordCount.java
```

Note: The **javac** tool reads class and interface definitions, written Java, and compiles them into bytecode class files. After running this command, you should have three brand-new files: *WordCount.class*, *WordCount$IntSumReducer.class*, and *WordCount$TokenizerMapper.class*.

03. Create a JAR for these three classes:
```
jar cf wordcount.jar WordCount*.class
```

04. View the contents of the JAR file to verify it contains our three classes (*WordCount.class*, *WordCount$IntSumReducer.class*, and *WordCount$TokenizerMapper.class*):
```
jar tf wordcount.jar
```

05. Create a text file (you can use a filler text generator, such as http://generator.lorem-ipsum.info/) and then import it into your HDFS space:
```
hadoop fs -put loremlpsum /user/username
```

06. Double-check that the text file was correctly stored into your HDFS space:
```
hadoop fs -ls /user/username
```

07. Run the application:
```
hadoop jar wordcount.jar WordCount /user/username/loremlpsum
/user/username/t01
```

08. Let's look at the results:
```
hadoop fs -ls t01
```

09. Output in detail:
```
hadoop fs -cat t01/part-r-00000
```

10. Extract the result files from your HDFS space:
```
hadoop fs -get /user/username/t01
```

3. 2nd program: Back to DBLP

For this exercise, we're going to use the DBLP publication database. We will be using a TSV (tab-separated value) version of its four tables (*authors*, *papers*, *venue*, *paperauths*). TSV files are simple text formats for storing data in a tabular structure. Each record in each table is separated from the next by a tab stop character (\t). You can access the DBLP tables on this location:

```
/infres/ir430/bd/dsm2/datasets/dblp_text/
```

In particular, let's look at the *papers* table. This is its schema:

```
papers (id: INTEGER, name: VARCHAR NOT NULL, venue: INTEGER REFERENCES
VENUE(id), pages: VARCHAR (50), url: VARCHAR);
```

Goal: Find out the total number of pages published in each venue referenced in this table.

We start with a map function, where for every row of the *papers* table (for every paper) we will need to extract the venue ID associated with it as well as its number of pages. Remember that *pages* is of type VARCHAR, so we will have to parse it just like we did for assignment 4. Once again, let's keep it simple and just parse *ranges of pages* (that is to say, values of pages with the format *Page1 – Page2*) and ignore all other formats.

01. Let's first take a look at the input file we're going to be working with.

```
head -4 /infres/ir430/bd/dsm2/datasets/dblp_text/papers.tsv
```

The output of this command (first 4 rows of the file) should look something like this:

```
0    Parallel Integer Sorting and Simulation Amongst CRCW Models 0     607-619
http://dx.doi.org/10.1007/BF03036466
1    Pattern Matching in Trees and Nets     1    227-248
http://dx.doi.org/10.1007/BF01257084
2    NP-complete Problems Simplified on Tree Schemas   1     171-178
http://dx.doi.org/10.1007/BF00289414
3    On the Power of Chain Rules in Context Free Grammars    3     425-433
http://dx.doi.org/10.1007/BF00264161
```

As you can see, our input is a TSV file. We will need to work with only two columns: column 2 (venue ID) and column 3 (pages). 2 and 3 are because of zero-based numbering (remember first column is column 0). Take a look at our Mapper class (*ProjectMapper*).

PageCount.java

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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.output.FileOutputFormat;

public class PageCount {

   public static class ProjectionMapper extends Mapper<Object, Text, IntWritable, IntWritable> {
      private IntWritable venue = new IntWritable();
      private IntWritable pages = new IntWritable();
```

```
        protected void map(Object key, Text value, Context context) throws IOException, InterruptedException {
            // Our file contains space-limited values: paper_id, paper_name, venue_id, pages, url
            // We project out: venue_id, pages
            String[] tokens = value.toString().split("\t");
            if (tokens.length == 5 && tokens[2].matches("^-?\\d+$")) {
                venue.set(Integer.parseInt(tokens[2])); // venue_id
                if (tokens[3].contains("-")) {
                    String[] range = tokens[3].split("-");
                    if (range.length == 2 && range[0].trim().matches("^-?\\d+$") && range[1].trim().matches("^-?\\d+$")) {
                        int length = Integer.parseInt(range[1].trim()) - Integer.parseInt(range[0].trim());
                        pages.set(length);
                        context.write(venue, pages);
                    }
                }
            }
        }
    }

    public static class IntSumReducer extends Reducer <IntWritable, IntWritable, IntWritable, IntWritable> {
        private IntWritable result = new IntWritable();

        public void reduce(IntWritable key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val: values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }

    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "page count");
        job.setJarByClass(PageCount.class);
        job.setMapperClass(ProjectionMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Our mapper function will map input key/value pairs to a set of intermediate key/value pairs whose internal representation (for the first 4 rows of our input file) would look something like this:

```
(0, 12)
(1, 21)
(1, 7)
(3, 8)
```

Hadoop then transforms the map output so that the values are brought together for a given key, in a process called the *shuffle*. In our internal representation, the input to the reduce step (for the first 4 rows of our input file) would look something like this:

```
(0, [12])
(1, [21, 7])
(3, [8])
```

We continue with our reduce function. All that our implementation has to do is sum the counts. Take a look at our Reducer class (*IntSumReducer*).

The output of the reducer will look like this (for the first 4 rows of our input file).

```
(0, 12)
(1, 28)
(3, 8)
```

Finally, look at our *main* method. Before we run our job, we need some driver code to wire up the mapper and reducer, which we do using this method. The Job instance in the code specifies various things about the job: a name for display purposes, the mapper and reducer classes that we discussed earlier, and the job output types, which have to match the mapper and reducer output types. The last line of the *main* method first launches the job and then waits for it to complete. While it is running, it prints the progress on the console.

Let's test it now. Copy/paste our code into a separate file that will be called *PageCount.java*. Then move on to the instructions that follow.

02. Initialize your Hadoop environment variables:

```
. /usr/local/bin/hadoop_env
```

03. Compile the Java classes with this command:

```
javac -cp $HADOOP_CLASSPATH -d . PageCount.java
```

04. Create a JAR for these three classes:

```
jar cf pagecount.jar PageCount*.class
```

05. Import *papers.tsv* into your HDFS space:

```
hadoop fs -put /infres/ir430/bd/dsm2/datasets/dblp_text/papers.tsv
/user/username/papers
```

06. Double-check that the text file was correctly stored into your HDFS space:

```
hadoop fs -ls /user/username
```

07. Run the application:

```
hadoop jar pagecount.jar PageCount /user/username/papers /user/username/t02
```

08. Let's look at the results:

```
hadoop fs -ls t02
```

09. Output in detail:

```
hadoop fs -cat t02/part-r-00000
```

10. Extract the result files from your HDFS space:

```
hadoop fs -get /user/username/t02
```