Hands-on Exercises with Big Data

Lab Sheet 1: Getting Started with MapReduce and Hadoop

The aim of this exercise is to learn how to begin creating MapReduce programs using the Hadoop Java framework. In particular, you will learn how to set up the Eclipse integrated development environment (IDE) for development of programs using Hadoop, build an initial word-counting program for use on tweets and run it both on your local machine and remotely on the Hadoop Cluster hosted on Amazon Elastic MapReduce and S3.

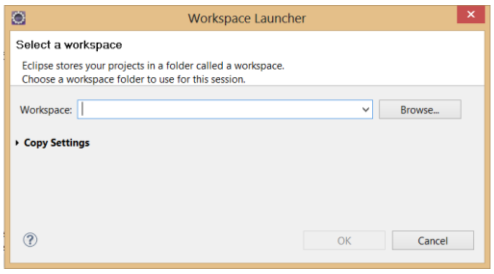
Part 1: Loading an existing Eclipse project

We will be using the Eclipse IDE for the development of MapReduce tasks. We have provided a compressed directory containing all of the code needed to perform this lab named:

SICSA-Lab.tar.gz

Extract the directory to a folder on the machine you are working on. Keep a note of the path to this folder, we will refer to this folder as <LabFolder> in these lab sheets. The machine you are working on may not come with Eclipse pre-loaded. If not, a copy can be downloaded from http://www.eclipse.org.

As Eclipse starts, it will ask for a workspace directory, as shown below:



Workspace Launcher

Select a workspace

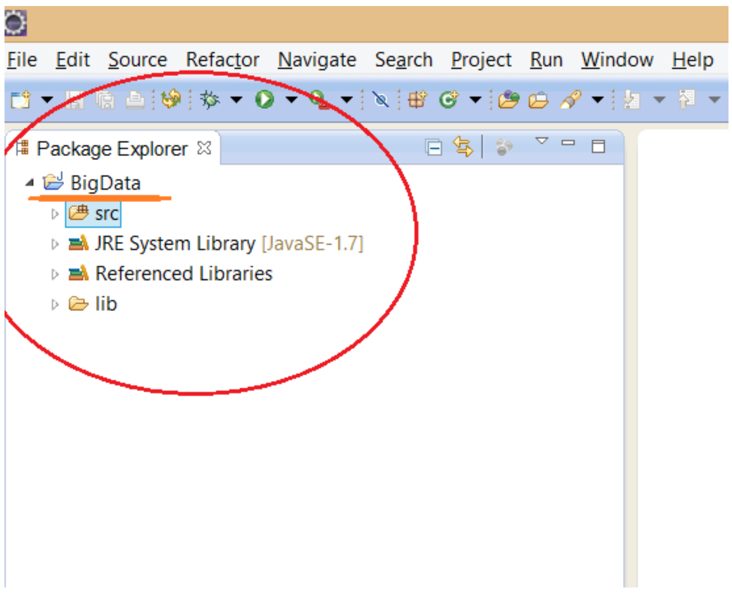
Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace:

The workspace is a folder containing all of the source code that you write. We have provided a ready workspace in the compressed file. Enter the path to the folder named 'workspace' located within the <LabFolder> directory that you extracted from the compressed file.

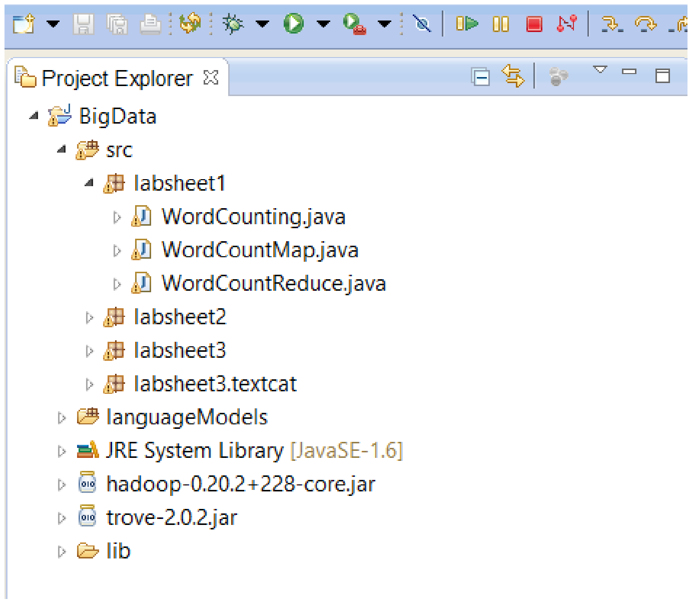
This will load the Java project containing the code that you will be working on within this lab. Within Eclipse, you should see the following project:

By Richard McCreadie (richard.mccreadie@glasgow.ac.uk)

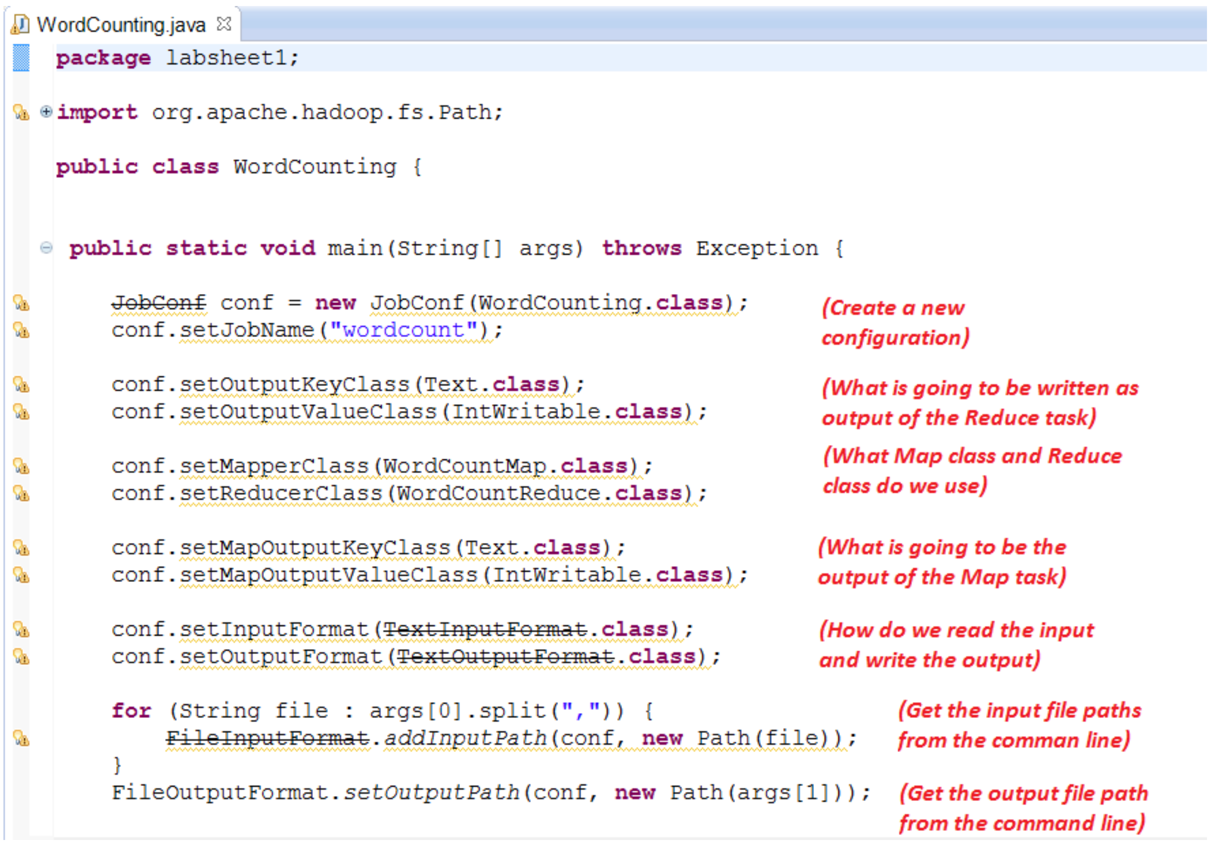

File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
BigData
src
JRE System Library [JavaSE-1.7]
Referenced Libraries
lib

Part 2: Examining a simple MapReduce Class: Word Counting

Open the 'src' (source) directory within the 'Big Data' project. You will see three packages belonging to the project, one per lab sheet for this hands on session. Open the labsheet1 package. Inside you will find three Java classes called WordCounting, WordCountingMap and WordCountingReduce as shown below.



These classes contain all of the source code needed to perform word counting in large datasets. Open the WordCounting class by double clicking upon it. As can be seen, the word counting class is comprised of a main() method. This configures the MapReduce job by specifying the classes that going to be used for input, processing and output and starts the Hadoop job.


WordCounting.java

```java
package labsheet1;

import org.apache.hadoop.fs.Path;

public class WordCounting {

 public static void main(String[] args) throws Exception {

     ~~JobConf~~ conf = new JobConf(WordCounting.class);          (Create a new
     conf.setJobName("wordcount");                                configuration)

     conf.setOutputKeyClass(Text.class);                          (What is going to be written as
     conf.setOutputValueClass(IntWritable.class);                 output of the Reduce task)

     conf.setMapperClass(WordCountMap.class);                     (What Map class and Reduce
     conf.setReducerClass(WordCountReduce.class);                 class do we use)

     conf.setMapOutputKeyClass(Text.class);                       (What is going to be the
     conf.setMapOutputValueClass(IntWritable.class);              output of the Map task)

     conf.setInputFormat(~~TextInputFormat~~.class);              (How do we read the input
     conf.setOutputFormat(~~TextOutputFormat~~.class);            and write the output)

     for (String file : args[0].split(",")) {                     (Get the input file paths
         ~~FileInputFormat~~.addInputPath(conf, new Path(file));  from the comman line)
     }
     FileOutputFormat.setOutputPath(conf, new Path(args[1]));     (Get the output file path
                                                                  from the command line)
```

For those unfamiliar with MapReduce, a short overview of the map and reduce methods is provided below:

During operation, multiple instances of the Map and Reduce classes will be made, where each can be run on different machines. MapReduce jobs can be defined in terms of the inputs and outputs of the Map and Reduce method. The Map method takes as input a series of <key1,value1> pairs and outputs a series of <key2,value2> pairs. The <key2,value2> pairs are grouped by key2 and sent the one or more processes running the Reduce class. The Reduce class takes as input all emitted values (value2) for a single key (key2), i.e. it takes as input a series of <key2, value2[]> tuples and then emits multiple <key3,value3> pairs.

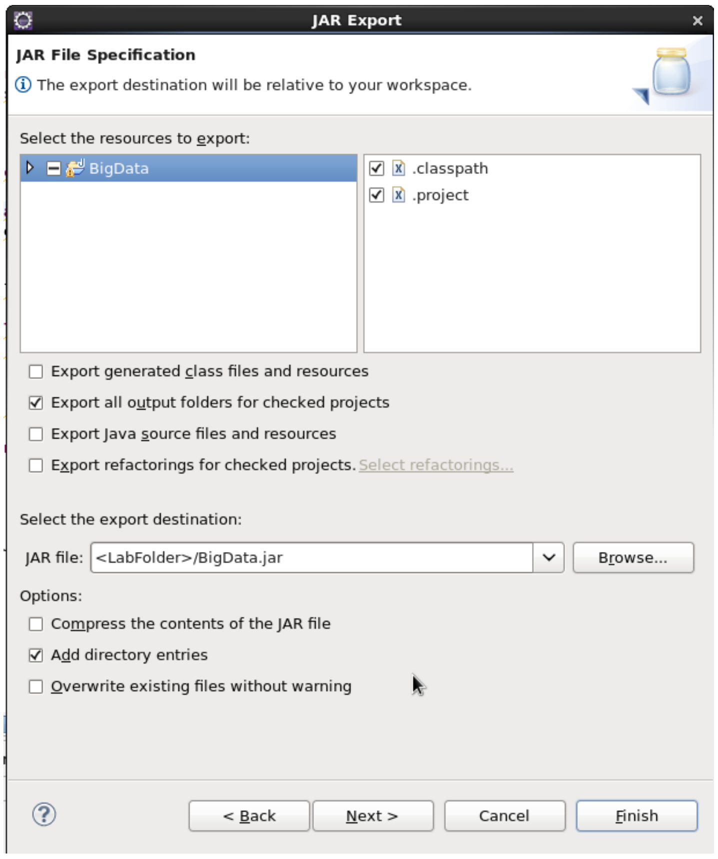
In Hadoop, there is one dedicated class for Map and Reduce, WordCountingMap and WordCountingReduce in this example. The Map class implements the Mapper interface and takes as input <LongWritable, Text> pairs, where LongWritable is a number that represents the unique identifier of the document and Text is a object containing the terms within the document. The map function uses the built-in StringTokeniser class to split the Text object into single tokens and then emits one <Text,IntWritable> pair for each, representing a term and the fact that the document contained that term (the IntWritable containing the number '1').

The Reduce class implements the Reducer interface and takes as input all of the IntWritables for a single term, i.e. the number of times that the term appears, sums them to find the total word count for that term and then emits the term and word count.

Part 3: Compiling the Word Counting Example and running it Locally

Having examined how the word counting example translates into a map and reduce tasks, we will now compile the Word Counting example and run it on your local machine.

First, we need to create a .jar file containing the Word Counting example. To do so in Eclipse, select 'export' from the 'file' menu and select 'JAR file' under the Java folder and press next. Select the BigData project from the resources list and then fill in the path where you want to save the jar file. For the remainder of this lab we (and the scripts you will use later) will assume you extracted it to <LabFolder>.



JAR Export

JAR File Specification

The export destination will be relative to your workspace.

Select the resources to export:

- BigData
- ☑ .classpath
- ☑ .project

- ☐ Export generated class files and resources
- ☑ Export all output folders for checked projects
- ☐ Export Java source files and resources
- ☐ Export refactorings for checked projects. Select refactorings...

Select the export destination:

JAR file: <LabFolder>/BigData.jar

Options:

- ☐ Compress the contents of the JAR file
- ☑ Add directory entries
- ☐ Overwrite existing files without warning

Press next twice and under the main class, enter the main class of the word counting example, namely labsheet1.WordCounting. Press finish to create the compiled BigData.jar file in the specified directory.

Next, we need to tell Hadoop to run the Word Counting code in local mode. To this end, we require both an input directory containing the files to be word counted and the output folder where the counts of each word will be placed.

We have provided a small input folder for you containing 300,000 tweets. You can find this in the directory you uncompressed, named 300k-Tweets. Create an output folder to store counted text in.

To run Word counting using Hadoop from the command line, the command is of the form:

```
<hadoop executable> jar <your jar file> <command line arguments>
```

Assuming you are currently within the <LabFolder> directory, run the following command:

```
./hadoop-1.1.2/bin/hadoop jar $( pwd )/BigData.jar   $(pwd)/300k-Tweets/SICSA-SummerSchool-Data.1.gz,$(pwd)/300k-Tweets/SICSA-SummerSchool-Data.2.gz,$(pwd)/300k-Tweets/SICSA-SummerSchool-Data.3.gz  <PATH-YOUR-OUTPUT-File>
```

Replacing the output directory path as appropriate. If Hadoop states that JAVA_HOME is not set, then you need to run:

```
export JAVA_HOME=$(/usr/libexec/java_home)
```

Running the WordCounting example locally will give the following output:

```
13/07/04 19:49:49 INFO util.NativeCodeLoader: Loaded the native-hadoop library
13/07/04 19:49:49 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.
13/07/04 19:49:49 WARN mapred.JobClient: No job jar file set.  User classes may not be found. See JobConf(Class) or JobConf#setJar(String).
13/07/04 19:49:49 INFO input.FileInputFormat: Total input paths to process : 1
13/07/04 19:49:49 WARN snappy.LoadSnappy: Snappy native library not loaded
13/07/04 19:49:49 INFO mapred.JobClient: Running job: job_local_0001
13/07/04 19:49:50 INFO util.ProcessTree: setsid exited with exit code 0
13/07/04 19:49:50 INFO mapred.Task:  Using ResourceCalculatorPlugin : org.apache.hadoop.util.LinuxResourceCalculatorPlugin@14fdb00d
13/07/04 19:49:50 INFO mapred.MapTask: io.sort.mb = 100
13/07/04 19:49:50 INFO mapred.MapTask: data buffer = 79691776/99614720
13/07/04 19:49:50 INFO mapred.MapTask: record buffer = 262144/327680
13/07/04 19:49:50 INFO zlib.ZlibFactory: Successfully loaded & initialized native-zlib library
13/07/04 19:49:50 INFO compress.CodecPool: Got brand-new decompressor
13/07/04 19:49:50 INFO mapred.JobClient:  map 0% reduce 0%
13/07/04 19:49:50 INFO mapred.MapTask: Spilling map output: record full = true
13/07/04 19:49:50 INFO mapred.MapTask: bufstart = 0; bufend = 3111183; bufvoid = 99614720
13/07/04 19:49:50 INFO mapred.MapTask: kvstart = 0; kvend = 262144; length = 327680
13/07/04 19:49:50 INFO mapred.MapTask: Finished spill 0
13/07/04 19:49:51 INFO mapred.MapTask: Spilling map output: record full = true
13/07/04 19:49:51 INFO mapred.MapTask: bufstart = 3111183; bufend = 6247558; bufvoid = 99614720
13/07/04 19:49:51 INFO mapred.MapTask: kvstart = 262144; kvend = 196607; length = 327680
13/07/04 19:49:51 INFO mapred.MapTask: Finished spill 1
13/07/04 19:49:51 INFO mapred.MapTask: Spilling map output: record full = true
13/07/04 19:49:51 INFO mapred.MapTask: bufstart = 6247558; bufend = 9393853; bufvoid = 99614720
13/07/04 19:49:51 INFO mapred.MapTask: kvstart = 196607; kvend = 131070; length = 327680
13/07/04 19:49:51 INFO mapred.MapTask: Finished spill 2
13/07/04 19:49:51 INFO mapred.MapTask: Starting flush of map output
13/07/04 19:49:51 INFO mapred.MapTask: Finished spill 3
13/07/04 19:49:51 INFO mapred.Merger: Merging 4 sorted segments
13/07/04 19:49:52 INFO mapred.Merger: Down to the last merge-pass, with 4 segments left of total size: 13473046 bytes
13/07/04 19:49:52 INFO mapred.Task: Task:attempt_local_0001_m_000000_0 is done. And is in the process of commiting
13/07/04 19:49:52 INFO mapred.LocalJobRunner:
13/07/04 19:49:52 INFO mapred.Task: Task 'attempt_local_0001_m_000000_0' done.
13/07/04 19:49:52 INFO mapred.Task:  Using ResourceCalculatorPlugin : org.apache.hadoop.util.LinuxResourceCalculatorPlugin@49d8c528
13/07/04 19:49:52 INFO mapred.LocalJobRunner:
13/07/04 19:49:52 INFO mapred.Merger: Merging 1 sorted segments
13/07/04 19:49:52 INFO mapred.Merger: Down to the last merge-pass, with 1 segments left of total size: 13473040 bytes
13/07/04 19:49:52 INFO mapred.LocalJobRunner:
13/07/04 19:49:52 INFO mapred.JobClient:  map 100% reduce 0%
13/07/04 19:49:53 INFO mapred.Task: Task:attempt_local_0001_r_000000_0 is done. And is in the process of commiting
13/07/04 19:49:53 INFO mapred.LocalJobRunner:
13/07/04 19:49:53 INFO mapred.Task: Task attempt_local_0001_r_000000_0 is allowed to commit now
13/07/04 19:49:53 INFO output.FileOutputCommitter: Saved output of task 'attempt_local_0001_r_000000_0' to /users/richardm/hadoop-1.1.2/output2
13/07/04 19:49:53 INFO mapred.LocalJobRunner: reduce > reduce
13/07/04 19:49:53 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
13/07/04 19:49:53 INFO mapred.JobClient:  map 100% reduce 100%
13/07/04 19:49:53 INFO mapred.JobClient: Job complete: job_local_0001
13/07/04 19:49:53 INFO mapred.JobClient: Counters: 20
13/07/04 19:49:53 INFO mapred.JobClient:   File Output Format Counters
13/07/04 19:49:53 INFO mapred.JobClient:     Bytes Written=4948052
13/07/04 19:49:53 INFO mapred.JobClient:   FileSystemCounters
13/07/04 19:49:53 INFO mapred.JobClient:     FILE_BYTES_READ=49213814
13/07/04 19:49:53 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=58937614
13/07/04 19:49:53 INFO mapred.JobClient:   File Input Format Counters
13/07/04 19:49:53 INFO mapred.JobClient:     Bytes Read=4397133
13/07/04 19:49:53 INFO mapred.JobClient:   Map-Reduce Framework
13/07/04 19:49:53 INFO mapred.JobClient:     Map output materialized bytes=13473044
13/07/04 19:49:53 INFO mapred.JobClient:     Map input records=100000
13/07/04 19:49:53 INFO mapred.JobClient:     Reduce shuffle bytes=0
13/07/04 19:49:53 INFO mapred.JobClient:     Spilled Records=2888928
13/07/04 19:49:53 INFO mapred.JobClient:     Map output bytes=11543220
13/07/04 19:49:53 INFO mapred.JobClient:     Total committed heap usage (bytes)=321519616
13/07/04 19:49:53 INFO mapred.JobClient:     CPU time spent (ms)=0
13/07/04 19:49:53 INFO mapred.JobClient:     SPLIT_RAW_BYTES=133
13/07/04 19:49:53 INFO mapred.JobClient:     Combine input records=0
13/07/04 19:49:53 INFO mapred.JobClient:     Reduce input records=962976
13/07/04 19:49:53 INFO mapred.JobClient:     Reduce input groups=282361
13/07/04 19:49:53 INFO mapred.JobClient:     Combine output records=0
13/07/04 19:49:53 INFO mapred.JobClient:     Physical memory (bytes) snapshot=0
13/07/04 19:49:53 INFO mapred.JobClient:     Reduce output records=282361
13/07/04 19:49:53 INFO mapred.JobClient:     Virtual memory (bytes) snapshot=0
13/07/04 19:49:53 INFO mapred.JobClient:     Map output records=962976
```

Notice how when running in local mode, only a single Map task (attempt_local_0001_m_000000_0) and a single Reduce task are run (attempt_local_0001_r_000000_0). To have multiple Map and Reduce tasks, we will need to have run the job remotely on a real cluster. However, local mode is useful for testing your jobs!

You can view the output of your job in the output folder specified. The output is logged by each reducer. In this case the word counted terms can be found in the part-r-00000 file (i.e. that produced by reducer 0).

Part 4: Running Word Counting on a Remote Cluster

While your local machine is useful for testing your Hadoop jobs on small amounts of data, it is insufficient to process large datasets. Instead, we need a remote cluster of multiple machines. For this lab, we will be using Amazons Elastic MapReduce service. This is a pay-for service that will provide pre-configured machines for running Hadoop jobs (the jobs you run here will be free).

Notably, within Amazon's service cloud, the data and computation are separated, with Elastic MapReduce (EMR) providing computation and S3 providing data storage. You will be communicating with EMR using the official command line tool provided by Amazon known as elastic-mapreduce (http://aws.amazon.com/developertools/2264). You will be moving data to and from S3 using a second tool s3cmd (http://s3tools.org/s3cmd).

There are eight steps to running a MapReduce job on Amazon:

1. Configure your Amazon Web Services credentials
2. Create a new 'bucket' in S3 to store your input/output
3. Upload your input data to S3
4. Upload your Hadoop Jar file to S3
5. Allocate a new cluster of machines, referred to as a '*Job Flow*'
6. Launching your MapReduce job, known as adding a jar step to the job flow.
7. Downloading your results when the job is finished
8. Terminate the cluster and delete files

In this lab, you will perform stems 1,2,4,5,6,7 and 8. We will omit step 2 because the datasets you will be using have already been loaded onto S3 for you to save time.

For each of these steps, we have provided a script to do the work within the <LabFolder> (this is to avoid you making typing errors for some of the longer commands).

Configure your Amazon Web Services credentials

For this lab you will be using a pre-defined Amazon Web Services account. We have configured this to avoid you incurring costs while testing. To this end, both elastic-mapreduce and s3cmd need to be configured with AWS credentials. We have already configured elastic-mapreduce for you (the configuration is stored in the credentials.json file in <LabFolder>/elastic-mapreduce/elastic-mapreduce-ruby). However, you will need to configure s3cmd yourself. To do so, open a terminal window in <LabFolder> and then enter the following command:

```
./s3cmd-1.0.1/s3cmd --configure
```

It will ask you for an access key, enter `AKIAJDMGALL7MMD6TXCA` and press enter.

It will then ask you for the secret key, enter `Kdcurlnj+Md24PT7V5iov512G9YohxBW37U7Zo5d` and press enter. Then keep pressing enter until the program exits (the other options are not needed). IF you create your own AWS account at a later date you will need to find your own access and secret key from your Amazon security credentials page on the AWS website as well as modify the credentials.json file entries.

Create a new 'bucket' in S3 to store your input/output

First, we will create what amounts to a folder on S3 to hold the output of the jobs that you will be running. To do so, run:

```
<LabFolder>/makeS4Bucket.sh <name>
```

where <name> is a unique name for your jobs (it may only contain lower-case letters), keep a note of <name> as you will be using it often. What this script does is call

```
s3cmd md s3://sicsa-summerschool.<name>/
```

which uses the 'md' method to make the bucket *sicsa-summerschool.<name>* on S3.

Upload your Hadoop Jar File

Next we need to upload the Jar file containing the Word Counting example, i.e. the BigData.jar file you created earlier. Assuming that you extracted it to *<LabFolder>/BigData.jar*, run the following script:

```
<LabFolder>/addJarToBucket.sh <name> BigData.jar
```

This script similarly calls s3cmd, except using the 'put' method that copies a local file to S3:

```
s3cmd put BigData.jar s3://sicsa-summerschool.<name>/BigData.jar
```

Allocate a new cluster of machines

Next, we need to create a Hadoop processing cluster on which we can run MapReduce jobs. To do so, we need to tell elastic-mapreduce to start a new job flow comprised a certain number of instances. The first instance is turned into a master node that performs administration functions while the remaining instances are allocated to a cluster for your hadoop jobs.

To create your first Hadoop cluster, run:

```
<LabFolder>/createJobFlow.sh <name>
```

This calls the following command:

```
elastic-mapreduce -c <where the user credentials are storied> --create --alive --
loguri  s3n://sicsa-summerschool.<name>/log/  --slave-instance-type  m1.large  --
master-instance-type m1.small --num-instances 2
```

The -c flag tells elastic-mapreduce where our user credentials are stored (we pre-configured this for you). --create indicates that we want a new job flow and --alive indicates that it should start at once. --loguri defines a location to store the output of the job. --*slave-instance-type* and --*master-instance-type* specify the types of machines we want for the processing and master nodes. --*num-instances* specifies the total number of machines, in this case the 2 will be allocated, one master and one processing node.

Upon running this command you will receive a response of the form

```
Created job flow <Job-Flow ID>
```

You will need to record the *<Job-Flow ID>* as you will use it to access this cluster.

Although the cluster has been allocated, it will still be starting up. Run the script:

```
<LabFolder>/listWorkFlows.sh
```

to see the status of your work flow. You will likely see multiple responses of the form:

```
 <Job-Flow ID> STARTING
```

Look for the job with the <Job-Flow ID> matching yours (the others will have been created by other people in the lab). When the job flow is ready for work, *STARTING* will change to *WAITING.* This may take a few minutes.

Launching your MapReduce Job

To launch your MapReduce job, run the following command:

```
<LabFolder>/launchMapReduceJob.sh <name> <Job-Flow ID> labsheet1.WordCounting
```

This script adds a new job flow step to the overall job flow comprised of your jar file and using the java main class labsheet1.WordCounting, i.e. the main class within that jar. You can have longer MapReduce tasks containing multiple steps, although we will not cover that in this lab. The command in full is shown below:

```
elastic-mapreduce -c <where the user credentials are storied> --jobflow <Job-Flow
ID>       --jar       s3://sicsa-summerschool.<name>/BigData.jar       --main-class
labsheet1.WordCounting       --arg       s3://sicsa-twitter-data/SICSA-SummerSchool-
Data.1.gz,s3://sicsa-twitter-data/SICSA-SummerSchool-Data.2.gz,  s3://sicsa-twitter-
data/SICSA-SummerSchool-Data.3.gz --arg s3://sicsa-summerschool.<name>/output/
```

Breaking this command down, --jobflow specifies that you are going to modify the job flow just created. --jar gives the location of your .jar file. Each of the --arg entries specify the arguments to be passed to the main class, i.e. the input and outputs in this case.

You can check the state of your job using the *<LabFolder>/listWorkFlows.sh <Job-Flow ID>* script as before. While running, under the entry for your <Job-Flow ID>, it will show up as:

```
      RUNNING       Example Jar Step
```

The work flow will return to WAITING status and the Example Jar Step status will change to either COMPLETED or FAILED when it is done.

Note that you can launch more MapReduce tasks onto this same job flow. These will show up as additional 'Example Jar Step' lines under the entry for your <Job-Flow ID>. Indeed, this is often worth doing rather than ending the job flow and creating a new one, since Amazon charges for a full hour's worth of usage on a job flow's machines even if you only use it for 5 minutes. A practical point of information is that the output path is fixed in the script (to `s3://sicsa-summerschool.<name>/output/)`, you will need to empty this folder or specify a different output folder before launching any more jobs. This is because Hadoop will not overwrite existing files in this directory, i.e. the reducer output (more information on how to do this can be found in the Terminate the Cluster and Delete Remote Files section).

Download your Results

If the job was successful, your results will be currently stored on S3 in the *s3://sicsa-summerschool.<name>/output/* folder. The script:

```
<LabFolder>/getYourBucketContents.sh <name>
```

Will copy the contents of your bucket to *<LabFolder>.* Creating both a log and output folder. The log folder contains all of the logging files from MapReduce. The output folder contains the files written by the reducer(s). As with local mode, there is only a single reducer, hence the output will be held in a single file named part-000000.

Terminate the Cluster and Delete Remote files

Once your job(s) are done, it is important to remember to end the cluster and delete any remote files that are not needed anymore, since Amazon charges by the hour for usage. We have prepared a terminate script for you:

```
<LabFolder>/terminateJobFlow.sh <name> <Job-Flow ID>
```

This script calls the elastic-mapreduce command to terminate a specified job flow (and its allocated machines):

```
elastic-mapreduce -c <where the user credentials are storied> --jobflow <Job-Flow
ID> --terminate
```

It then calls the s3cmd twice, first to empty all of the contents of your bucket and then to delete the bucket itself:

```
s3cmd del --recursive --force s3://sicsa-summerschool.<name>/

s3cmd rb s3://sicsa-summerschool.<name>
```

Note that if you just want to delete the contents of your bucket, for instance because you want to run another job with the same job flow then you can use:

```
<LabFolder>/deleteBucketContents.sh <name>
```

(don't forget to upload your new Jar file though before running!)

You have just run your first distributed MapReduce job using Hadoop - ***Congratulations!***

By Richard McCreadie (richard.mccreadie@glasgow.ac.uk)

Lab Sheet 2: Top Hashtag Identification

In this lab sheet you will learn how to modify a MapReduce job and how to use the configure and close methods to make use of stateful tasks. In particular, you will take the code of the word counting example from the previous task and modify it to find the top 10 hashtags from the input corpus.

Part 1: Modify the Map class to emit only hashtags

To begin, open up the Eclipse IDE and the BigData project. Within the 'src' folder open the labsheet2 package, you will see as before three classes, namely: TopHashTags; TopHashTagsMap; and TopHashtagsReduce, representing the main class, map task and reduce task, respectively.

Open the TopHashTagsMap class. This class contains only the map method that currently tokenises (splits each document into terms) and emits those terms. Modify this method such that it only contains hashtags (hashtags are terms that start with the '#' character).

Part 2: Modify the Reduce class to store hashtags and emit only the top 10

Next, open the TopHashTagsReduce task. Note that this class contains three methods, namely configure, reduce and close.

- The *configure* is called once as the MapReduce job is initialising before any documents are processed.
- The *close* method is called after all documents have been processed.
- The *reduce* method currently takes in all of the frequency counts for each term (which will now only be hashtags) and emits the those terms and counts.

Modify this class to store the hashtags as they arrive and emit only the 10 most common hashtags, i.e. those with the highest frequency. Note that you will need to perform the emits during the close method once you have seen all of the hashtags.

Part 3: Compile and launch the Job

Finally, you need to compile and launch the new job as in lab sheet 1. Select 'export' in the 'file' menu and select 'JAR file' under the Java folder and press next. Select the BigData project from the resources list and then fill in the path where you want to save the jar file. Again we are assuming you are extracting it to <LabFolder>.

Press next twice, then fill in the main class for launching, in this case *labsheet2.TopHashTags.* Then press 'finish' to compile your code.

You can run the job locally using the following command

```
./hadoop-1.1.2/bin/hadoop jar $( pwd )/BigData.jar   $(pwd)/300k-Tweets/SICSA-
SummerSchool-Data.1.gz,$(pwd)/300k-Tweets/SICSA-SummerSchool-Data.2.gz,$(pwd)/300k-
Tweets/SICSA-SummerSchool-Data.3.gz <PATH-YOUR-OUTPUT-File>
```

By Richard McCreadie (richard.mccreadie@glasgow.ac.uk)

or by following the 8 steps from part 4 of labsheet 1 to run it on Amazon Elastic MapReduce.

The target results top 10 hashtags are:

```
#backintheday	807
#MentionKe	686
#np	471
#nowplaying	420
#BackInTheDay	249
#sutadora	233
#fb	175
#jfb	165
#codysimpson	164
#agoodwoman	139
```

By Richard McCreadie (richard.mccreadie@glasgow.ac.uk)

Lab Sheet 3: Indexing English Tweets

In this lab exercise you will learn how to use external .jar files with a project, use additional resources and change the output format of a MapReduce job. In particular, this exercise is focused on using a MapReduce job to build an index of English tweets from a generic tweet sample. You will create a map task that will classify incoming tweets as English or not and create a new output format for Hadoop that will write the classified documents in a format that a search engine (the Terrier IR Platform) can understand.

Part 1: Performing tweet language classification

To begin, open up the Eclipse IDE and the BigData project. Within the 'src' folder open the labsheet2 package, you will see as before four classes, namely: TweetClassification, TweetClassificationMap, TweetClassificationReduce and TRECOutputFormat. The first step is to modify the TweetClassificationMap class to perform classification of incoming tweets.

To do so, we require an additional classification package. We suggest you download and use the Stanford text classifier, available at: http://nlp.stanford.edu/software/

Import a classification tool and modify TweetClassificationMap such that it only emits English tweets.

Part 2: Saving Tweets in TREC format

The second part of this exercise involves changing the output of the Hadoop job. So far, we have used the TextOutputFormat class that is built into Hadoop to write our output. This class simply writes a key value pair on a single line in the output file for each call of the collect(key,value) made by the reducer.

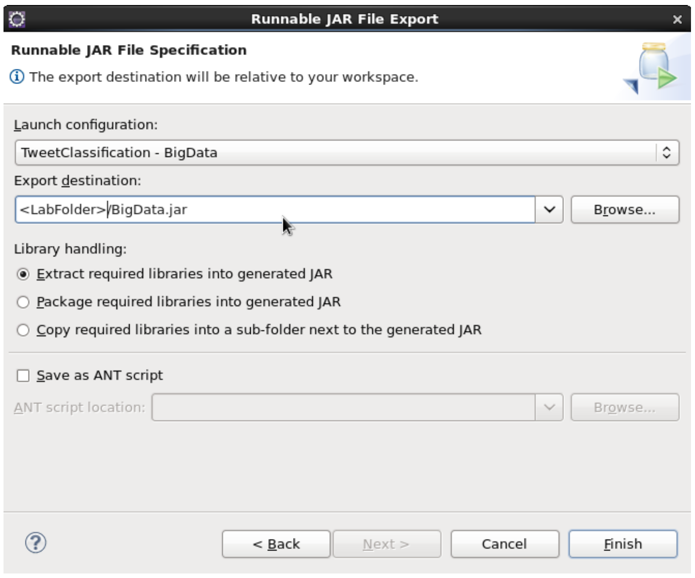
However, to index the classified tweets, you will make a new output format that writes in a standard TREC format (used by the Text REtrieval Conference - http://trec.nist.gov) that the target search engine Terrier can understand. This format is as follows:

```
<DOC>
<DOCNO>9876</DOCNO>
<TEXT>tweet text</TEXT>
</DOC>
```

Open the TRECOutputFormat class. This class implements a single method getRecordWriter() that returns an object that implements the RecordWriter interface, i.e. that has a write(key,value) method. The getRecordWriter() creates the output stream that will write tweets to a file and creates a new TRECRecordWriter object which implements RecordWriter. TRECRecordWriter is an inner class of TRECOutputFormat. The current version is missing the writeTREC(Object docno, Object text) method. Finish writing this method such that it writes tweets into the format shown above.

Part 3: Compile and run the Tweet Classification Code

You need to compile and launch the new job as in lab sheet 1, except that we will be creating a Runnable Jar rather than a basic Jar file. The difference is that a runnable jar must have a main class specified and will include all additional jars that are included on the classpath. Select 'export' in the 'file' menu and select 'Runnable Jar' under the Java folder and press next. Under launch configuration, select the main class for labsheet3, i.e. TweetClassification (If this class does not appear in the drop down list then press cancel, right click on the class in project explorer and select 'Run As>Java Application', then retry creating the runnable jar). Then under export destination fill in the path where you want to save the jar file. Again we are assuming you are extracting it to <LabFolder>.


Runnable JAR File Export

Runnable JAR File Specification

The export destination will be relative to your workspace.

Launch configuration: TweetClassification - BigData

Export destination: <LabFolder>/BigData.jar

Library handling:
- [✓ Extract required libraries into generated JAR]
- Package required libraries into generated JAR
- Copy required libraries into a sub-folder next to the generated JAR

☐ Save as ANT script

ANT script location:

You can then run the compiled jar file as normal either in local using the following command

```
./hadoop-1.1.2/bin/hadoop jar $( pwd )/BigData.jar $(pwd)/300k-Tweets/SICSA-SummerSchool-Data.1.gz,$(pwd)/300k-Tweets/SICSA-SummerSchool-Data.2.gz,$(pwd)/300k-Tweets/SICSA-SummerSchool-Data.3.gz <PATH-YOUR-OUTPUT-File>
```

or by following the 8 steps from part 4 of labsheet 1 to run it on Amazon Elastic MapReduce. Note that classification is a lot more costly than, word counting hence it takes quite a bit longer.

The output file produced by the reducer should contain the tweets matching the TREC format.

```
<DOC>
<DOCNO>6023</DOCNO>
<TEXT>Out to eat with my peeps.</TEXT>
</DOC>
<DOC>
<DOCNO>7726</DOCNO>
<TEXT>#twitteroff</TEXT>
</DOC>
<DOC>
<DOCNO>9192</DOCNO>
<TEXT>Oh...um.... #subtweet</TEXT>
</DOC>
```

Part 4: Indexing the Tweets

Finally, to illustrate a use-case of your MapReduce classification job. We will now index the English tweets you extracted. We have included a copy of the Terrier search engine in <LabFolder>/terrier-3.5/

First, specify where the Terrier home directory is:

```
export TERRIER_HOME=$( pwd )/terrier-3.5
```

Then configure Terrier to access the your output file using the following command:

```
./terrier-3.5/bin/trec_setup.sh ./output/part-00000.
```

This assumes that your output file is ./output/part-00000.

Then copy the Terrier configuration file:

```
cp ./terrier-3.5/terrier.properties ./terrier-3.5/etc/
```

Then, you can index the documents using the following command:

```
./terrier-3.5/bin/trec_terrier.sh -i
```

You can then run searches on the tweets indexed with interactive terrier using:

```
./terrier-3.5/bin/http_terrier.sh 8080 ./terrier-3.5/src/webapps/sicsa
```

You can then search for tweets using the Web interface at localhost:8080 in your web browser..

Extra Tasks

Congratulations, you have finished all of the Lab exercises and how know how to create and launch MapReduce tasks. Here are some other things you can try:

- We have provided a large 6 million tweet sample at S3://sicsa-twitter-data/SICSA-SummerSchool-Data.6m.gz - how would you modify the addMapReduceTask.sh script to use this file instead?
- Amazon has a good series of Video tutorials introducing some of the more advanced features of Elastic MapReduce - why not create an account and give them a go? - http://aws.amazon.com/elasticmapreduce/training/

By Richard McCreadie (richard.mccreadie@glasgow.ac.uk)