Extreme Computing Hadoop Lab Session

Based on material from Stratis Viglas, Michail Basios & Sasa Petrovic

Part 1: Getting Started

This lab is mostly performed in terminal. Open your terminal application to begin.

Non-DICE

Firstly, if you are doing this tutorial using a machine that isn't DICE, you'll need to ssh into a DICE machine from either a UNIX terminal or a Windows ssh application like PuTTY, where `sXXXXXXX` is your matriculation number. If you're already using a DICE computer, you can skip this step. For more information, see computing support.

```
ssh sXXXXXXX@student.ssh.inf.ed.ac.uk
```

You can now continue with the DICE steps.

DICE

We're going to connect to one of the nodes of the cluster. This command will randomly pick one of the 7 servers and connect to it:

```
ssh scutter0$((RANDOM%7+1))
```

You can now test Hadoop by running the following command:

```
hdfs dfs -ls /
```

If you get a listing of directories on HDFS, you've successfully configured everything. If not, make sure you do **ALL** of the described steps **EXACTLY** as they appear in this document. Note that you should not continue if you have not managed to do this section. If the `hdfs` command isn't available, contact a demonstrator

HDFS

In order to let you copy-paste commands, we'll use `$USER` which the shell will turn into your user name (i.e. sXXXXXXX).

Here are a number of small pointers you should work through to familiarise yourself with navigating around HDFS

1. Make sure that your home directory exists:

```
hdfs dfs -ls /user/$USER
```

To create a directory called `/user/$USER/data` in Hadoop:

```
hdfs dfs -mkdir /user/$USER/data
```

Create the following directories in a similar way (these directories will NOT have been created for you, so you need to create them yourself):

- `/user/$USER/data/input`
- `/user/$USER/data/output`
- `/user/$USER/source`

Confirm that you've done the right thing by typing

```
hdfs dfs -ls /user/$USER
```

For example, if your matriculation number is `s0123456`, you should see something like:

```
Found 2 items
drwxr-xr-x   - s0123456 s0123456          0 2011-10-19 09:55 /user/s0123456/data
drwxr-xr-x   - s0123456 s0123456          0 2011-10-19 09:54 /user/s0123456/source
```

2. Copy the file `example1.txt` to `/user/$USER/data/output` by typing:

```
hdfs dfs -cp /data/labs/example1.txt /user/$USER/data/output
```

It might warn you about `DFSInputStream`. Just ignore that.

3. Obviously, `example1.txt` doesn't belong there. Move it from `/user/$USER/data/output` to `/user/$USER/data/input` where it belongs and delete the `/user/$USER/data/output` directory:

```
hdfs dfs -mv /user/$USER/data/output/example1.txt /user/$USER/data/input/
hdfs dfs -rm -r /user/$USER/data/output/
```

4. Examine the contents of `example1.txt` using `cat` and then `tail`:

```
hdfs dfs -cat /user/$USER/data/input/example1.txt
hdfs dfs -tail /user/$USER/data/input/example1.txt
```

5. Create an empty file named `example2` in `/user/$USER/data/input`. Use `test` to check if it exists and that it is indeed zero length.

```
hdfs dfs -touchz /user/$USER/data/input/example2
hdfs dfs -test -z /user/$USER/data/input/example2
```

6. Remove the file `example2`:

```
hdfs dfs -rm /user/$USER/data/input/example2
```

List of HDFS Commands

What follows is a list of useful HDFS shell commands.

- `cat` -- copy files to `stdout`, similar to UNIX `cat` command:

```
hdfs dfs -cat /user/$USER/data/input/example1.txt
```

- `copyFromLocal` -- copy single source, or multiple sources from local file system to the destination filesystem. Source must be a local file reference:

```
hdfs dfs -copyFromLocal <localfile> /user/$USER/file1
```

- `copyToLocal` -- copy files to the local file system. Destination must be a local file reference.

```
hdfs dfs -copyToLocal /user/$USER/file1 <localfile>
```

  Options:

  - `-ignoreCrc` -- files that fail the CRC check will be copied.
  - `-crc` -- files and CRCs will be copied.

- `cp` -- copy files from source to destination. This command allows multiple sources as well in which case the destination must be a directory. Similar to UNIX `cp` command.

```
hdfs dfs -cp /user/$USER/file1 /user/$USER/file2
```

- `getmerge` -- take a source directory and a destination file as input and concatenate files in src into the destination local file. Optionally `-nl` can be set to enable adding a newline character at the end of each file.

```
hdfs dfs -getmerge /data/labs/example-data ~/result_file
```

- `ls` -- for a file returns stat on the file with the format:
  `filename num_replicas size modification_date modification_time permissions userid groupid`

For a directory it returns list of its direct children as in UNIX, with the format:
`dirname <dir> modification_time modification_time permissions userid groupid`

```
hdfs dfs -ls /user/$USER
```

You can also pass `-R` for recursive listing.

- `mkdir` -- create a directory.

```
hdfs dfs -mkdir /user/$USER/deleteme
```

You can pass `-p` to make directories along a path

```
hdfs dfs -mkdir -p /user/$USER/deleteme/and/this
```

- `mv` -- move files from source to destination similar to UNIX `mv` command. This command allows multiple sources as well in which case the destination needs to be a directory. Moving files across filesystems is not permitted.

```
hdfs dfs -mv /user/$USER/file1 /user/$USER/file2
```

- `rm` -- delete files, similar to UNIX `rm` command. Only deletes empty directories and files.

```
hdfs dfs -rm /user/$USER/file1
```

Also supports `-r` to recursively delete files like `rm -r` on UNIX.

- `tail` -- Displays last kilobyte of the file to stdout. Similar to UNIX `tail` command.

```
hdfs dfs -tail /user/$USER/file1
```

Options:

  - `-f` output appended data as the file grows (follow)

- `test` -- perform various test.

```
hdfs dfs -test -e /user/$USER/file1
```

Options:

  - `-e` check to see if the file exists. Return 0 if true.
  - `-z` check to see if the file is zero length. Return 0 if true.
  - `-d` check return 1 if the path is directory else return 0.

`-test` returns the value of its test (0 or 1) to the environment variable `$?`, to view its value enter the following into your terminal:

```
echo $?
```

- `touchz` -- create a file of zero length. Similar to UNIX `touch` command.

```
hdfs dfs -touchz /user/$USER/file1
```

Running Jobs

The Hadoop examples are in `/opt/hadoop/hadoop-2.7.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.3.jar` which is a lot to type. So you might want to set an environment variable

```
export EXAMPLES=/opt/hadoop/hadoop-2.7.3/share/hadoop/mapreduce/hadoop-mapreduce-example
```

Then we can use `$EXAMPLES` to refer to that path.

Computing pi

This example estimates the mathematical constant π to some error. The error depends on the number of samples we have (more samples $\rightarrow$ more accurate estimate). Run the example as follows:

```
hadoop jar $EXAMPLES pi <num_maps> <num_samples>
```

Where `<num_maps>` is the number of mapper jobs, and `<num_samples>` is the number of samples, for example using 10 mappers and 5 samples:

```
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7
```

Try the following combinations for `<num_maps>` and `<num_samples>` and see how the running time and precision change:

Number of Maps	Number of Samples	Time (s)	$\hat{\pi}$
2	10		
5	10		
10	10		
2	100		
10	100		

Do the results match your expectations? How many samples are needed to approximate the third digit after the decimal dot correctly?

Word Counting

Hadoop has a number of demo applications and here we will look at the canonical task of word counting.

> **TASK:** Try running through the following example

We will count the number of times each word appears in a document. For this purpose, we will use the `/data/labs/example3.txt` file, so first copy that file to your input directory. Second, make sure you delete your output directory before running the job or the job will fail. We run the wordcount example by typing:

```
hadoop jar $EXAMPLES wordcount /user/$USER/data/input /user/$USER/data/output
```

Where `/user/$USER/data/input` and `/user/$USER/data/output` are the input and output directories, respectively. After running the example, examine (using `ls`) the contents of the output directory. From the output directory, copy the file `part-r-00000` to a local directory (somewhere in your home directory) and examine the contents. Was the job successful?

Running Streaming Jobs

Hadoop streaming is a utility that allows you to create and run map/reduce jobs with **any executable or script** as the mapper and/or the reducer. The way it works is very simple: input is converted into lines which are fed to the `stdin` of the mapper process. The mapper processes this data and writes to `stdout`. You can learn more about `stdin` and `stdout` here.

Lines from the `stdout` of the mapper process are converted into key/value pairs by splitting them on the **first tab character** (of course, this is only the default behavior and can be changed). The key/value pairs are fed to the `stdin` of the reducer process which collects and processes them.

Finally, the reducer writes to `stdout` which is the final output of the program. Everything will become much clearer through examples later.

It is important to note that with Hadoop streaming mappers and reducers can be any programs that read from `stdin` and write to `stdout`, so the choice of the programming language is left to the

programmer. Here, we will use Python.

Writing a Word-Counting Program in Python 2.7.5

In this subsection, we will see how to create a program in Python that can count the number of words of a specific file. Initially, we will test the code locally on small files before using it in a streaming MapReduce job.

As we will see later, this is important as it helps in not running jobs in Hadoop that can give wrong results.

Word-Counting Python Mapper

1. Using Streaming, a Mapper reads from `stdin` and writes to `stdout`
2. Keys and Values are delimited (by default) using tabs
3. Records are split using newlines

Create a file somewhere in your home directory called `mapper.py` -- there are a number of Python IDEs available, including PyCharm on DICE machines. Alternatively, simply use `gedit`:

```
gedit mapper.py
```

In the directory you want your mapper to be in. Then copy the following code into `mapper.py` and save. It's worth typing this out by hand rather than copy / pasting, to understand what the code is doing. If you are unfamiliar with the `.format` syntax of string interpolation Python, please refer here.

Python

```python
#!/usr/bin/python

import sys

for line in sys.stdin:                  # input from standard input
    line = line.strip()                 # remove whitespaces
    tokens = line.split()               # split the line into tokens

    for token in tokens:                # write the results to standard output
        print("{0}\t{1}".format(token, 1))
```

Make sure you save the file `mapper.py`.

Word-Counting Python Reducer

Create a file called `reducer.py` in the same directory as `mapper.py`, and copy the following code into it:

```python
#!/usr/bin/python

import sys

prev_word = ""
value_total = 0
word = ""

for line in sys.stdin:          # For ever line in the input from stdin
    line = line.strip()         # Remove trailing characters
    word, value = line.split("\t", 1)
    value = int(value)
    # Remember that Hadoop sorts map output by key reducer takes these keys sorted
    if prev_word == word:
        value_total += value
    else:
        if prev_word:  # write result to stdout
            print("{0}\t{1}".format(prev_word, value_total))

        value_total = value
        prev_word = word

if prev_word == word:  # Don't forget the last key/value pair
    print("{0}\t{1}".format(prev_word, value_total))
```

Once again, save this file before continuing.

Testing the Code

We perform local testing conforming to typical UNIX-style piping, our testing will take the form:

```
cat <data> | map | sort | reduce
```

Which emulates the same pipeline that Hadoop will perform when streaming, albeit in a non-distributed manner. You have to make sure that files `mapper.py` and `reducer.py` have execution permissions:

```
chmod u+x mapper.py
chmod u+x reducer.py
```

Try the following command and explain the results (hint: type `man sort` in your terminal window to find out more about the `sort` command):

```
echo "this is a test and this should count the number of words" | ./mapper.py | sort -k1
```

Sanity Check

The output from the above code should result in the following output:

```
a       1
and     1
count   1
is      1
number  1
of      1
should  1
test    1
the     1
this    2
words   1
```

> **TASK:** Count the number of words a text file of your chosing contains.

Running a Streaming MapReduce Job

After running locally the code successfully, the next step is to run it in Hadoop. Suppose you have your mapper, `mapper.py`, and your reducer, `reducer.py`, and the input is in `<input>`.

We always have to specify `/opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar` as the jar to run, and the particular mapper and reducer we use are specified through `-mapper` and `-reducer` options.

In the case that the mapper and/or reducer are not already present on the remote machine (which will often be the case), we also have to package the actual files in the job submission. Assuming that neither `mapper.py` nor `reducer.py` were present on the machines in the cluster, the previous job would be run as follows:

```Bash
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
 -input <input> \
 -output <output> \
 -mapper mapper.py \
 -file mapper.py \
 -reducer reducer.py \
 -file reducer.py
```

Here, the `-file` option specifies that the file is to be copied to the cluster. This can be very useful for also packaging any auxiliary files your program might use (dictionaries, configuration files, etc). Each job can have multiple `-file` options.

> **TASK:** We will run a simple example to demonstrate how streaming jobs are run, follow these steps

Copy the file: `/data/labs/source/random-mapper.py` from HDFS to a local directory (a directory on the local machine, **not** on HDFS.) This mapper simply generates a random number for each word in the input file, hence the input file in your `input` directory can be anything. Run the job by typing:

```bash
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
 -input /user/$USER/data/input \
 -output /user/$USER/data/output \
 -mapper random-mapper.py \
 -file random-mapper.py
```

IMPORTANT NOTE: for Hadoop to know how to properly run your Python scripts, you **must** include the following line as the **first** line in *all* your mappers and reducers:

```
#!/usr/bin/python
```

> **TASK:** What happens when instead of using `mapper.py` you use `/bin/cat` as a mapper?
>
> **TASK:** What happens when you use `/bin/cat` as *both* a mapper and reducer?

Setting Job Configuration

Various job options can be specified on the command line, we will cover the most used ones in this section. The general syntax for specifying additional configuration variables is `-jobconf <name>=<value>`

To avoid having your job named something like `streamjob5025479419610622742.jar`, you can specify an alternative name through the `mapred.job.name` variable:

```
-jobconf mapred.job.name="My job"
```

> **TASK:** Run the `random-mapper.py` example again, this time naming your job `"Random job <matriculation_number>"`, where `<matriculation_number>` is your matriculation number.
>
> After you run the job (and preferably before it finishes), open the browser and go to http://jobtracker.inf.ed.ac.uk:8088/cluster/nodes. In the list of running jobs look for the job with the name you gave it and click on it. You can see various statistics about your job -- try to find the number of reducers used. How many reducers did you use? If your job finished before you had a chance to open the browser, it will be in the list of finished jobs, not the list of running jobs, but you can still see all the same information by clicking on it.

Secondary Sorting

As was mentioned earlier, the key/value pairs are obtained by splitting the mapper output on the first tab character in the line. This can be changed using `stream.map.output.field.separator` and `stream.num.map.output.key.fields` variables. For example, if I want the key to be everything up to the second `-` character in the line, I would add the following:

```
-jobconf stream.map.output.field.separator=- \
-jobconf stream.num.map.output.key.fields=2
```

Hadoop also comes with a partitioner class `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner` which is useful for cases where you want to perform a *secondary* sort on the keys. Imagine you have the following list of IPs:

```
192.168.2.1
190.191.34.38
161.53.72.111
192.168.1.1
161.53.72.23
```

You want to partition the data so that addresses with the first 16 bits are processed by the same reducer. However, you also want each reducer to see the data sorted according to the first 24 bits of the address. Using the mentioned partitioner class you can tell Hadoop how to group the data to be processed by the reducers. You do this using the following options:

```
-jobconf mapreduce.map.output.key.field.separator=.
-jobconf num.key.fields.for.partition=2
```

The first option tells Hadoop what character to use as a separator (just like in the previous example), and the second one tells how many fields from the key to use for partitioning. Knowing this, here is how we would solve the IP address example (assuming that the addresses are in `/user/hadoop/input`):

Bash

```
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
 -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib.partition.
 -D mapreduce.map.output.key.field.separator=. \
 -D stream.map.output.field.separator=. \
 -D stream.num.map.output.key.fields=3 \
 -D num.key.fields.for.partition=2 \
-input <input> \
-output <output> \
-mapper cat \
-reducer cat \
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

The line with `-D num.key.fields.for.partition=2` tells Hadoop to partition IPs based on the first 16 bits (first two numbers), and `-D stream.num.map.output.key.fields=3` tells it to sort the IPs according to everything before the third separator (the dot in this case) -- this corresponds to the first 24 bits of the address.

> **TASK:** Copy the file `secondary` to your input directory. Lines in this file have the following format:

```
last_name.first_name.address.phone_number
```

Using what you have learned in this section, your task is to:

1. Partition the data so that all people with the same last name go to the same reducer.
2. Partition the data so that all people with the same last name go to the same reducer, and also make sure that the lines are sorted according to first name.

Partitioning the Data, and Secondary Sorting

We want to partition the data so that all the people with the same first and last name go to the same reducer, and that they are sorted according to address. An implementation is given:

```Bash
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
 -D mapreduce.job.output.key.comparator.class=org.apache.hadoop.mapreduce.lib.partition.
 -D mapreduce.map.output.key.field.separator=. \
 -D stream.map.output.field.separator=. \
 -D stream.num.map.output.key.fields=3 \
 -D num.key.fields.for.partition=2 \
 -D mapreduce.partition.keypartitioner.options=-k1,2 \
 -D mapreduce.partition.keycomparator.options=-k3 \
-input /data/labs/secondary.txt \
-output /user/$USER/output \
-mapper cat \
-reducer cat \
-partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner
```

NOTE: The order of the arguments is significant. Arguments using `-D` are *generics* and must come before *command* arguments such as `-input`

`hdfs dfs -cat /user/$USER/output/*` returns the expected output as follows:

```
Simmons.Gene.Elm street 1.555-1000
Stanley.Cup.Elm street 1.555-1002
Singer.Eric.Elm street 2.555-1001
Stanley.Paul.Elm street 3.555-1002
Thayer.Tommy.Elm street 4.555-1003
Simmons.Gene.Elm street 5.555-666
```

The key points are the configurations of the KeyFieldBasedComparator class and the KeyFieldBasedPartitioner class. The documentation provides further examples of how you may use these features.

Note that *multiple* options mean that they should be enclosed in quotation marks, unlike the previous example of just `-k3`. If we wanted to sort by the first column in descending order, and both the second column, and then the third column, in ascending order:

```
 -D mapreduce.partition.keypartitioner.options=-k1,2 \
 -D mapreduce.partition.keycomparator.options="-k1,1r -k2,2 -k3,3"
```

This yields the result:

```
Thayer.Tommy.Elm street 4.555-1003
Stanley.Cup.Elm street 1.555-1002
Stanley.Paul.Elm street 3.555-1002
Singer.Eric.Elm street 2.555-1001
Simmons.Gene.Elm street 1.555-1000
Simmons.Gene.Elm street 5.555-666
```

If your keys are numeric, you need to us the `-n` modified. For instance, imagine the first key is a name, and the second is an age (integer), to sort both in descending order (note the `-n` and `-r` become `nr` after the index notation:

```
 -D mapreduce.partition.keycomparator.options="-k1,1r -k2,2nr"
```

Part 2: Running a MapReduce Program in Java

> If you are unfamiliar with Java, or would like to know more, it is recommended that you review relevant video lectures and materials from the IJP course.

Setting Up the Environment

You can either set these environment variables per-session, or append them to your `.bash_profile` file in your home directory:

```Bash
export JAVA_HOME=/usr/lib/jvm/java-1.8.0-sun-1.8.0.91/
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

The Code (Adapted from Hadoop Documentation)

Java

```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> {
    // Making objects is expensive. Instantiate outside the loop and re-use
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context) throws IOException, Interru
      StringTokenizer itr = new StringTokenizer(value.toString());

      // Whilst iterating over the token iterator
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());  // Store the next token in our Text object
        context.write(word, one);  // Give a <word, 1> pair
      }
    }
  }

  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 I
      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");

    // Make this class the main in the JAR file
    job.setJarByClass(WordCount.class);

    // Set out Mapper class, conforming to the API
```

```
    job.setMapperClass(TokenizerMapper.class);

    // Set out Combiner & Reducer classes, conforming to the (same) API
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);

    // Set the ouput Key type
    job.setOutputKeyClass(Text.class);

    // Set the output Value type
    job.setOutputValueClass(IntWritable.class);

    // Set number of reducers
    job.setNumReduceTasks(10);

    // Get the input and output paths from the job arguments
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Compiling the Java Code

We're going to compile this Java code into a JAR file.

```
hadoop com.sun.tools.javac.Main WordCount.java
jar cf mywordcount.jar WordCount*.class
```

As a note, the number of mappers can *be suggested* via a command line argument `-D mapred.map.tasks=5`, but ultimately the `InputFormat` will decide upon the number needed.

Running the Job

Now, deploy the JAR file with the input data noted below, and the output directory we just created in HDFS.

```
hadoop jar mywordcount.jar WordCount /data/labs/example3.txt /user/$USER/lab2
```

Sanity Check - Output

Print the top 10 lines of the output to the terminal and compare it to the output below. If all was done correctly, they should be the same. If not, check over your code and try again. If that still doesn't work, ask for help!

To print the top 10 lines:

```
hdfs dfs -cat /user/$USER/lab2/* | head -n 10
```

Make sure that your output matches the following:

```Bash
But     1
ask     1
both    1
desert  2
up      6
you     21
aching  1
been    2
down    2
game    1
```

> **TASK:** Run again the same task as previously but use 5 reducers instead of 10. Observe the output folder and the number of files produced.
>
> **TASK:** Run the example of the wordcount problem from the first lab by using only two reducers.

Managing Hadoop Jobs

Command Line

Listing Jobs

Running the following command will list all (completed and running) jobs on the cluster:

```
mapred job -list all
```

This will result in a table like this being displayed:

```
JobId                 State     StartTime     UserName Priority SchedulingInfo
job_201009151640_0001 2         1285081662441 s0894589 NORMAL   NA
job_201009151640_0002 2         1285081976156 s0894589 NORMAL   NA
job_201009151640_0003 2         1285082017461 s0894589 NORMAL   NA
job_201009151640_0004 2         1285082159071 s0894589 NORMAL   NA
job_201009151640_0005 2         1285082189917 s0894589 NORMAL   NA
job_201009151640_0006 2         1285082275965 s0894589 NORMAL   NA
job_201009151640_0009 2         1285083343068 s0894589 NORMAL   NA
job_201009151640_0010 3         1285106676902 s0894589 NORMAL   NA
job_201009151640_0012 3         1285106959588 s0894589 NORMAL   NA
job_201009151640_0013 3         1285107094387 s0894589 NORMAL   NA
job_201009151640_0014 2         1285107283359 s0894589 NORMAL   NA
job_201009151640_0015 2         1285109169514 s0894589 NORMAL   NA
job_201009151640_0016 2         1285109271188 s0894589 NORMAL   NA
job_201009151640_0018 1         1285148710573 s0894589 NORMAL   NA
```

Job Status:

To get the status of a particular job, we can use

```
mapred job -status $jobid
```

Where the `$jobid` is ID of a job found in the first column of the list table above. The status will show the percent of completion of mappers and reducers, along with a tracking URL and the location of a file with all the information about the job. We will soon see that the web interface provides much more detail.

Kill a Job

To kill a job, run the following command, where `$jobid` is the ID of the job you want to kill:

```
mapred job -kill $jobid
```

> **TASK:** Run through the following exercise

To try this, copy the file `/data/labs/source/sleeper.py` somewhere on your local filesystem (**not HDFS**) and run it as a streaming job (specifying `sleeper.py` as the mapper, with no reducers, remember to use the `-file` option). You can set the input to be anything you like because the mapper doesn't actually do anything (except wait for you to kill it), and also remember to set the output to be a directory on HDFS that does not exist. It might be helpful if you name your job to something familiar – this will reduce the time needed to find its ID later.

Open another terminal, log into the Hadoop cluster, and list all the jobs. Find the ID of the job you just started (use the name you gave it for faster searching). Find out the status of your job, and finally kill it. After you run the kill command, look at the terminal where you originally started the job and watch the job die.

Web Interface

All of the previous actions can also be performed using the web interface. Open a browser and navigate to (or simply click if viewing in a browser) http://scutter02.inf.ed.ac.uk:8088. Note: you will need to be inside Informatics or use the VPN to see this page.

This shows the web interface of the jobtracker. We can see a list of running, completed, and failed jobs. Clicking on a job ID is similar to requesting its status from the command line, but it shows much more details, including the number of bytes read/written to the filesystem, number of failed/killed task attempts, and nice graphs of job completion.

Using Side Information

It is often useful to package other, external files together with the job. For example, if your application uses a dictionary or a file that stores some configuration settings, one would want these files to be available to the program just as they would in a non-mapreduce setting. This can be achieved using the

`-file` option that we already used to package the source files. The following program takes a dictionary and counts only those words that appear in the dictionary, ignoring everything else. First copy the `/data/labs/source/mapper-dict.py` and `/data/labs/source/reducer.py` to a **local** directory:

```
hdfs dfs -get /data/labs/source/mapper-dict.py ~/
hdfs dfs -get /data/labs/source/reducer.py ~/
```

Now copy the dictionary to a **local** directory(`-get` and `-copyToLocal` provide the same functionality):

```
hdfs dfs -copyToLocal /data/labs/dict.eng ~/
```

> **TASK:** Run the program by typing (assuming you still have `example3.txt` in your input directory and that your output directory doesn't exist):

```Bash
hadoop jar /opt/hadoop/hadoop-2.7.3/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \
 -input  /user/$USER/data/example3.txt \
 -output /user/$USER/data/output \
 -mapper mapper-dict.py \
 -file mapper-dict.py \
 -reducer reducer.py \
 -file reducer.py \
 -file dict.eng
```

The program will use `dict.eng` as the dictionary and count only those words that appear in that list. Look at the source of `mapper-dict.py` to see how to open the dictionary file.

Further Resources

You now have a broad knowledge of the way in which hadoop is used. If you'd like to know more, please use the following resources:

1. Official Hadoop Documention
2. Hadoop Cheat Sheet
3. Hadoop for Dummies Cheat Sheet
4. YouTube Playlist -- Hadoop Tutorials

These are presented as optional reading, and good places to consult if you're stuck.