Extreme computing lab exercises
Session one

Miles Osborne (original: Sasa Petrovic)

October 23, 2012

1 Getting started

First you need to access the machine where you will be doing all the work. Do this by typing:

```
ssh namenode
```

If this machine is busy, try another one (for example `bw1425n13`, `bw1425n14` or any other number up to `bw1425n24`).

Hadoop is installed on Dice under `/opt/hadoop/hadoop-0.20.2` (this will also be referred to as the *Hadoop home directory*). The `hadoop` command can be found in the `/opt/hadoop/hadoop-0.20.2/bin` directory. To make sure you can run Hadoop command without having to be in the same directory, use your favorite editor (emacs) to edit the `~/.benv` file and add the following line:

```
PATH=$PATH:/opt/hadoop/hadoop-0.20.2/bin/
```

Don't forget to save the changes! After that, run the following command:

```
source ~/.benv
```

to make sure new changes take place right away. After you do this, run

```
hadoop 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.

2 HDFS

Hadoop consists of two major parts: a distributed filesystem (HDFS for Hadoop DFS) and the mechanism for running jobs. Files on HDFS are distributed across the network and replicated on different nodes for reliability and speed. When a job requires a particular file, it is fetched from the machine/rack nearest to the machines that are actually executing the job.

You will now proceed to do some simple commands in HDFS. REMEMBER: all the commands mentioned here refer to HDFS shell commands, NOT to UNIX commands which may have the same name. You should keep all your source and data within a directory named `/user/<your_matriculation_number>` which should already be created for you.

1. Make sure that your home directory exists: ◁ **Tasks**

   ```
   hadoop dfs -ls /user/sXXXXXXX
   ```

   where sXXXXXXX should be your matriculation number. To create a directory in Hadoop:

   ```
   hadoop dfs -mkdir dirName
   ```

   Create directories `/user/sXXXXXXX/data/input`, `/user/sXXXXXXX/source`, and `/user/sXXXXXXX/data/output` in a similar way (these directories will NOT have been created for you, so you need to create them yourself). Confirm that you've done the right thing by typing

```
hadoop dfs -ls /user/sXXXXXXX
```

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 `/user/sasa/data/example1` to `/user/sXXXXXXX/data/output` by typing:

```
hadoop dfs -cp /user/sasa/data/example1  /user/sXXXXXXX/data/output
```

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

```
hadoop dfs -mv /user/sXXXXXXX/data/output/example1 /user/sXXXXXXX/data/input
```

```
hadoop dfs -rmr /user/sXXXXXXX/data/output/
```

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

```
hadoop dfs -cat /user/sXXXXXXX/data/input/example1
```

```
hadoop dfs -tail /user/sXXXXXXX/data/input/example1
```

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

```
hadoop dfs -touchz /user/sXXXXXXX/data/input/example2
```

```
hadoop dfs -test -z /user/sXXXXXXX/data/input/example2
```

6. Remove the file `example2`:

```
hadoop dfs -rm /user/sXXXXXXX/data/input/example2
```

2.1 List of HDFS commands

What follows is a list of useful HDFS shell commands.

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

```
hadoop dfs -cat /user/hadoop/file4
```

- `copyFromLocal` – copy single src, or multiple srcs from local file system to the destination filesystem. Source has to be a local file reference.
  Example:

```
hadoop dfs -copyFromLocal localfile /user/hadoop/file1
```

- `copyToLocal` – copy files to the local file system. Files that fail the CRC check may be copied with the `-ignorecrc` option. Files and CRCs may be copied using the `-crc` option. Destination must be a local file reference.
  Example:

```
hadoop dfs -copyToLocal /user/hadoop/file localfile
```

- `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.
  Example:

  ```
  hadoop dfs -cp /user/hadoop/file1 /user/hadoop/file2
  ```

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

  ```
  hadoop dfs -getmerge /user/hadoop/mydir/ ~/result_file
  ```

- `ls` – for a file returns stat on the file with the format:
  `filename <number of 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`
  Example:

  ```
  hadoop dfs -ls /user/hadoop/file1
  ```

- `lsr` – recursive version of `ls`. Similar to UNIX `ls -R` command.
  Example:

  ```
  hadoop dfs -lsr /user/hadoop/
  ```

- `mkdir` – create a directory. Behaves similar to UNIX `mkdir -p` command creating parent directories along the path (for bragging rights, what is the difference?)
  Example:

  ```
  hadoop dfs -mkdir /user/hadoop/dir1 /user/hadoop/dir2
  ```

- `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.
  Example:

  ```
  hadoop dfs -mv /user/hadoop/file1 /user/hadoop/file2
  ```

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

  ```
  hadoop dfs -rm /user/hadoop/file1
  ```

- `rmr` – recursive version of `rm`. Same as `rm -r` on UNIX.
  Example:

  ```
  hadoop dfs -rmr /user/hadoop/dir1/
  ```

- `tail` – Displays last kilobyte of the file to stdout. Similar to UNIX `tail` command. Options:
  -f output appended data as the file grows (follow)

  ```
  hadoop dfs -tail /user/hadoop/file1
  ```

- `test` – perform various test. 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.
  Example:

  ```
  hadoop dfs -test -e /user/hadoop/file1
  ```

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

  ```
  hadoop dfs -touchz /user/hadoop/file1
  ```

3 Running jobs

3.1 Computing π

NOTE: in this example we use the `hadoop-0.20.2-examples.jar` file. This file can be found in `/opt/hadoop/hadoop-0.20.2/`, so make sure to use the full path to that file if you are not running the example from the `/opt/hadoop/hadoop-0.20.2/` directory.

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

```
hadoop jar hadoop-0.20.2-examples.jar pi <nMaps> <nSamples>
```

Task ▷ where <nMaps> is the number of mapper jobs, and <nSamples> is the number of samples. Try the following combinations for <nMaps> and <nSamples> 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?

3.2 Demo: Word counting

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

Task ▷ We will count the number of times each word appears in a document. For this purpose, we will use the `/user/sasa/data/example3` 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 hadoop-0.20.2-examples.jar wordcount <in> <out>
```

where `in` and `out` 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?

3.3 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. Lines from the stdout of the mapper process are converted into key/value pairs by splitting them on the *first tab character*.[1] 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.

How to actually run the job

Suppose you have your mapper, `mapper.py`, and your reducer, `reducer.py`, and the input is in `/user/hadoop/input/`. How do you run your streaming job? It's similar to running the DFS examples from the previous section, with minor differences:

[1] Of course, this is only the default behavior and can be changed.

```
hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar \
           -input /user/hadoop/input \
           -output /user/hadoop/output \
           -mapper mapper.py \
           -reducer reducer.py
```

Note the differences: we always have to specify `contrib/streaming/hadoop-0.20.2-streaming.jar` (this file can be found in `/opt/hadoop/hadoop-0.20.2/` directory so you either have to be in this directory when running the job, or specify the full path to the file) as the jar to run, and the particular mapper and reducer we use are specified through `-mapper` and `-reducer` options.

In 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

```
hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar \
           -input /user/hadoop/input \
           -output /user/hadoop/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.

We will run a simple example to demonstrate how streaming jobs are run. Copy the file: ◁ **Task**

`/user/sasa/source/random-mapper.py`

from HDFS to a local directory (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:

```
hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar \
           -input /user/sXXXXXXX/data/input \
           -output /user/sXXXXXXX/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/reducers:

`#!/usr/bin/python`

What happens when instead of using `mapper.py` you use `/bin/cat` as a mapper? ◁ **Task**

What happens when you use `/bin/cat` as a mapper AND as a reducer? ◁ **Task**

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 `mapred.job.name` variable. For example,

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

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://hcrc1425n01.inf.ed.ac.uk:50030/`. 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. ◁ **Task**

3.4 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 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):

```
hadoop jar contrib/streaming/hadoop-0.20.2-streaming.jar \
            -input /user/hadoop/input \
            -output /user/hadoop/output \
            -mapper cat \
            -reducer cat \
            -partitioner org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner \
            -jobconf stream.map.output.field.separator=.  \
            -jobconf stream.num.map.output.key.fields=3 \
            -jobconf map.output.key.field.separator=.  \
            -jobconf num.key.fields.for.partition=2
```

The line with `-jobconf num.key.fields.for.partition=2` tells Hadoop to partition IPs based on the first 16 bits (first two numbers), and `-jobconf 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 `/user/sasa/data/secondary` to your input directory. Lines in this file have the following format:

```
LastName.FirstName.Address.PhoneNo
```

Using what you 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.
3. 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.