Data Analytics with HPC: Hadoop Walkthrough

In this walkthrough you will learn to execute simple Hadoop Map/Reduce job on a Hadoop cluster. We will use Hadoop to count the occurrences of words in four novels by Arthur Conan Doyle (obtained by Project Gutenberg).

The tutorial files

The `data` directory will contain the data files needed for this project. In here you will find the text of the four novels (`pg126.txt` , `pg244.txt` , `pg3070.txt` and `pg537.txt`) and the text of a slightly altered version of the opening line from Star Wars (`StarWars.txt`). We will use the Star Wars text for very simple testing and will run the full job on the book texts.

It's important that the full path to the directory for this tutorial does NOT include any spaces. Hadoop does not like files in directories with spaces and will produce errors.

Hadoop streaming

For this tutorial we will use Hadoop streaming. This allows you write Hadoop Map, Reduce and Combine functions as Unix scripts that read data from the standard input and write results to the standard output. Since you have learnt Python on this module you will write your scripts using Python but it is also possible to write these scripts in other languages including C, Ruby and R.

Hadoop streaming is an alternative way to program Hadoop than the traditional approach of writing and compiling Java code.

Map script

Our map script must read lines of data and process the lines one at a time. To output key value pairs the script will simply write them as text to the standard output using a tab character to separate the key and the value.

To count words we simply parse the input to extract each word and write it out as the key followed by the value `1` . For example, for input:

```
A far time ago in
a galaxy far, far away...
```

We wish to output (where tab characters are used to separate the key and value):

```
a        1
far      1
time     1
ago      1
in       1
a        1
galaxy   1
far      1
far      1
away     1
```

We will start with the Python code `map.py` in the `src` directory. Look at the code in this file and try to understand what it does.

Run the script on the Star Wars text to see if it produces the correct result:

```
cat data/StarWars.txt | src/map.py
```

You should notice that it has failed to produce the correct output in three places. Fix the code so that it produces the correct output.

Hint: in Python punctuation can be removed from a string by the following line

```
myString = myString.translate(None,string.punctuation):
```

Hint: Python has a method called `lower()` that works on the string class to convert the string to lower case, e.g. `myString = myString.lower()`

Reduce script

When using Hadoop streaming the reduce script reads in key/value pairs (one per line and separated by a tab character) and outputs new key value pairs (again one per line and separated by a tab character). Hadoop guarantees that all the keys/value pairs with the same key will be sent to the same reducer. Additionally, all the lines with the same key will be grouped together. Because lines with the same key are grouped together it is guaranteed then if the script reads a new key then all the data lines associated with the previous key will have been read.

The input to the reduce script is therefore similar to that produced by running:

```
cat data/StarWars.txt | src/map.py | sort
```

Here you see the role of sorting in the Map/Reduce execution. On a large scale Hadoop system each reducer will only see a portion of the data, the reducer that handles a particular key will see all the pairs with that key.

A reduce function that counts the number of times each key occurs is coded in `src/reduce.py` . Look at this file and understand how it works.

You can test the whole map/reduce pipeline by running:

```
cat data/StarWars.txt | src/map.py | sort | src/reduce.py
```

Has the output successfully counted the occurrences of each word?

Testing on a sample of the book data

Before running on the full book data sets we should test the scripts on a sample of the data. Hadoop supports a variety of sampling methods that are useful in practice but we will continue to use Unix at this stage of the development. The Unix command `head` returns the first *n* lines of a file so we can use it as a very simple sampler (note that for many data sets sampling just the first *n* records can be a very bad idea in practice). Run:

```
head -500 data/pg126.txt | src/map.py | sort | src/reduce.py
```

Are you happy with the result? If not, where is the problem?

If you see an error like:

```
Traceback (most recent call last):
  File "src/reduce.py", line 20, in <module>
    word, count = line.split('\t', 1)
ValueError: need more than 1 value to unpack
```

Then use:

```
head -500 data/pg126.txt | src/map.py | sort
```

to look at the first few lines of the output from the mapper. Change the mapper code to fix the problem.

Now we are getting close to running the job on Hadoop…

Details of the Hadoop cluster

The Hadoop cluster used for this exercise consists of two nodes and the replication is set to two. This means that each data block will be replicated in two nodes of the cluster. The data block size is 64MB which is well beyond the size of any of the files used in this exercise. Each book will therefore be stored in a single data block. This is why the exercise uses four books to ensure we read from four different data blocks and ideally four different nodes.

Hadoop distributed file system (HDFS)

Before we can process any data in Hadoop we must first upload the data to HDFS. To interact with the Hadoop file system you will need to use the `hadoop fs` command (run `hadoop fs -help` for more details).

To copy the book data files from the local filesystem to HDFS run:

```
hadoop fs -copyFromLocal data/pg*.txt .
```

After copying you can now list the files in your Hadoop file system:

```
hadoop fs -ls
```

You will see something like:

```
-rw-r--r--   2 ahume supergroup     187650 2015-11-25 16:25 pg126.txt
-rw-r--r--   2 ahume supergroup     267468 2015-11-25 16:25 pg244.txt
-rw-r--r--   2 ahume supergroup     345766 2015-11-25 16:25 pg3070.txt
-rw-r--r--   2 ahume supergroup     440321 2015-11-25 16:25 pg537.txt
```

This is very similar to what you would see on a Unix filesystem but notice the number before your username. This is the replication index for the file it tells you how many copies of the file are in HDFS.

Running the job on Hadoop

Finally, we can now run the job on Hadoop. Run the following command on a single line:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar
  -files src/map.py,src/reduce.py
  -input pg*.txt -output wordCountResult
  -mapper map.py -reducer reduce.py
```

This will create a directory called `wordCountResult` on HDFS. Note that if this directory already exists the job will fail so if you run the program multiple times choose a different output directory or delete the directory before running the job (`hadoop fs -rm -r wordCountResult`).

If you see an error like that shown below then this will be because you have a space the path to your directory. Do not use directory name with spaces when using Hadoop.

```
Exception in thread "main" java.lang.IllegalArgumentException: java.net.URISyntaxE
xception: Illegal character in path
at index 38: file:/home/sxxxxxxxxx/Documents/Semester One/DM/Hadoop/hadoopPractica
l/src/mymap.py
```

The job will run and print out some many statistics as it does so. Things to notice in the output include the number of map and reduce tasks and the number of input and output records for the mapper and reducer:

```
INFO mapreduce.JobSubmitter: number of splits:4
INFO mapred.MapTask: numReduceTasks: 1
```

and

```
Map input records=24633
Map output records=226736

Reduce input records=226736
Reduce output records=12939
```

Now look at the files produced in the result directory using the command:

```
hadoop fs -ls wordCountResult/
```

You will see something like:

```
-rw-r--r--   2 ahume supergroup          0 2015-11-26 14:31 wordCountResult/_SUCCE
SS
-rw-r--r--   2 ahume supergroup     134422 2015-11-26 14:31 wordCountResult/part-0
0000
```

Here the `_SUCCESS` file tells us the job completed successfully and the result is in the `part-00000` file. There is only one output file because our job has a single reducer (which is Hadoop's default).

You can look at the contents of the file with a command like:

```
hadoop fs -cat wordCountResult/part-00000
```

Or you can copy the file back to the local filesystem with:

```
hadoop fs -copyToLocal wordCountResult/part-00000 wordCountResult.data
```

Specifying more than one reducer

Often you will wish to have more than one reducer so the reducer work can be distributed over Hadoop's nodes. The number of reducers to use is easily specified when using Hadoop streaming by the `-numReduceTasks` flag.

Run the following command (on a single line) to execute the job with two reducers:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar
  -files src/map.py,src/reduce.py
  -input pg*.txt -output wordCountTwoReducers
  -mapper map.py -reducer reduce.py
  -numReduceTasks 2
```

The output job details should now specify that two reducers were used:

```
INFO mapreduce.JobSubmitter: number of splits:4
INFO mapred.MapTask: numReduceTasks: 2
```

If you look at the output of this job you will now see two data files:

```
hadoop fs -ls wordCountTwoReducers
```

gives:

```
-rw-r--r--   2 ahume supergroup          0 2015-11-26 14:44 wordCountTwoReducers/_
SUCCESS
-rw-r--r--   2 ahume supergroup      67384 2015-11-26 14:44 wordCountTwoReducers/p
art-00000
-rw-r--r--   2 ahume supergroup      67038 2015-11-26 14:44 wordCountTwoReducers/p
art-00001
```

The result data is now split into two parts. If you look at the data you will see that the parts are sorted alphabetically but both files contain different words and span the whole range of the alphabet. To produce the same output as we got from a single reducer these two output files would have to be merged together. For many large files this could be a time consuming exercise.

To create part files that can simply be concatenated together to produce the same sorted output as was produced with one reducer you will need to use a special partitioner called `TotalOrderPartitioner`. This partitioner requires that the user specify details of how the data should be partitioned over the reducers and is bit complicated to set up. It is outside the scope of this practical, but it is good to know that it can be done.

Adding a combiner

Now we will decrease the amount of data that is transferred between nodes by adding a combiner. A combiner can be thought of as a local reducer that processes the data before it is given to the partitioner. A combiner may be run zero, one or more times and this is controlled by the Hadoop framework. The combiner must therefore be considered as an optional optimisation stage.

To use a combiner for the word count example the combiner must simply count up the occurrences of each key exactly as the current reducer does. The combiner and reducer can therefore simply use the same code in this case (note that this is not always the case).

Let's assume that each line of the Star Wars text is processed by different nodes. The first node processes the first line (given by `head -1`) and uses the reduce code as a combiner. It will therefore execute something similar to:

```
head -1 data/StarWars.txt | src/map.py | sort | src/reduce.py
```

and the second node processed the second line (given by `tail -1`) and produce output similar to:

```
tail -1 data/StarWars.txt | src/map.py | sort | src/reduce.py
```

The reducer must now be able to take the combined input from these two nodes and still produce the correct answer. For testing purposes the concatenated output is in the file `data/StarWarsCombinerOutput.txt` . If we put this into your reducer what do you get?

```
cat data/StarWarsCombinerOutput.txt | sort | src/reduce.py
```

Is the result correct? How many times has the word ‘far’ been counted?

Fix the reducer so that the result for ‘far’ is 3, while at the same time ensuring that the result of ‘a’ also remains 2.

When this works we can now run the job on Hadoop using the combiner. Run:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar
   -files src/map.py,src/reduce.py
   -input pg*.txt -output wordCountWithCombiner
   -mapper map.py -reducer reduce.py -combiner reduce.py
   -numReduceTasks 2
```

You should see in the report that the combiner has significantly reduced the number of input records to the reducer to about about 10% of the previous figure. This will result in considerably less data being transferred between nodes:

```
Map input records=24633
Map output records=226736

Combine input records=226736
Combine output records=24804

Reduce input records=24804
Reduce output records=12939
```

Optional extras

If you are more experienced with Hadoop or simply get to this stage very quickly then you may wish to implement some of the Map/Reduce examples discussed in the lecture. The citation data used in some of the examples can be downloaded from: http://www.nber.org/patents/ (http://www.nber.org/patents/) (download the ASCII version of Cite75_99.txt).

An alternative, possibly simpler, extra would be to use Haddoop to process the output of the first word count exercise (wordCountResult/part-00000) such the words are sorted by number of times they occur. Additionally, filter the output so that only those words that occur more than 100 times are output.