Practical 2

In this practical we will learn how to

1. Access the Hadoop Distributed File System (HDFS),
2. Create a Python MapReduce program locally,
3. Run the MapReduce program on Hadoop Streaming,
4. Copy the output from the MapReduce program to Linux.

Part 1 - Access HDFS

In the previous practical we set up a Linux virtual machine (VM) running on your computer. That virtual machine has its own file system. For future reference when you see **Linux** mentioned in a practical tutorial, its referring to that file system (directories and files).

On the VM we are running Hadoop. Hadoop clusters can have several dataNodes in order to distribute processing tasks. What we are using is a single node virtual cluster Hadoop installation. In our practicals we refer to this file system as **HDFS**.

It is important for us to note the distinction, and that the two different systems have their own directories and files. To get data to HDFS you need to be able to move it from your linux file system to HDFS. This normally means getting a file onto the linux VM and then onto HDFS. To access HDFS from within your linux VM, use the Hadoop FS Shell commands. A full documentation of these commands can be found here.

Last prac we went over some basic Linux commands. In this prac we're still heading to the same terminal window, we'll just be learning Hadoop Shell commands instead.

1. Create a subdirectory on your Linux home. We'll be making different mapReduce programs in the future, and each one should have its own folder. We'll call the folder for our first program **prac2** to denote this Practical 2.

```
$ mkdir /home/prac/prac2
```

2. Remember that HDFS is a different file system, so we need to make a folder for our mapReduce program there too. The option -p means create the parent directories if not created.

```
$ hadoop fs -mkdir -p /user/prac/prac2
```

3. Create a mirrored set of subdirectories within both Linux and HDFS. The first subdirectory we'll make is called **input**. To create the **input** directory under the **prac2** directory in Linux, type the following:

```
$ mkdir /home/prac/prac2/input
```

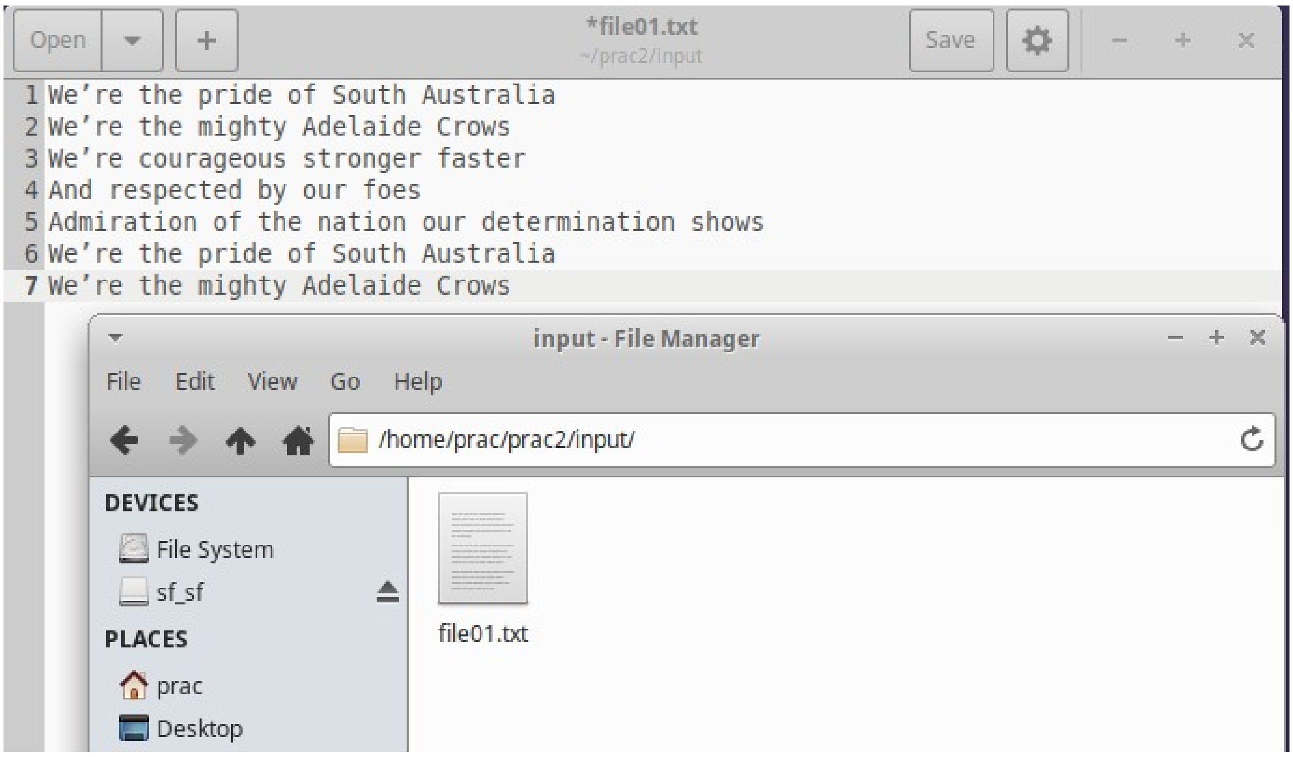
Similarly, to make the directory in HDFS:

```
$ hadoop fs -mkdir /user/prac/prac2/input
```

Now, on your own, create one more subdirectory under **prac2** called **src**. Note, to repeat a previous line in the terminal you can simply press the up key, then edit some changes in.

4. In this step we place a file into Linux and then HDFS. In the **input** directory, use gedit text editor used in last practical to create a new text file called **file01.txt** with the following content.

```
We’re the pride of South Australia
We’re the mighty Adelaide Crows
We’re courageous stronger faster
And respected by our foes
Admiration of the nation our determination shows
We’re the pride of South Australia
We’re the mighty Adelaide Crows
```



```
We’re the pride of South Australia
We’re the mighty Adelaide Crows
We’re courageous stronger faster
And respected by our foes
Admiration of the nation our determination shows
We’re the pride of South Australia
We’re the mighty Adelaide Crows
```

/home/prac/prac2/input/

To move the file from our linux **input** directory to our hdfs **input** directory we use the following command:

```
$ hadoop fs -put /home/prac/prac2/input/* /user/prac/prac2/input
```

5. We can us the **ls** command to list files in our HDFS system. To see the contents of the HDFS subdirectory **prac2** type the following:

```
$ hadoop fs -ls /user/prac/prac2
```

Check the contents of the **input** subdirectory too, to make sure the text file made it there.

Part 2 - Create a Python MapReduce Program

The Hadoop framework is written in Java, why don't we use that? Because high-level Java skills are required to write and run even simple MapReduce programs. On the contrary, Python is an incredibly popular language among data scientists, and it is much easier to write code when creating Mapper and Reducer functions.

In this practical, we will write a simple word counting program in Python using Mapper and Reducer. To do so, our MapReduce program should do two things:

1. Read a text file, counting how often each word occurs in the file;
2. Write a text file, displaying each unique word and its count.

Our output will be a set of **key/value** pairs, separated by a tab. This is a good way to approach writing a MapReduce program, starting with a visualisation of what our final output will look like. Once we have an idea of what the output should look like we can work backwards through the program to reach it. For example, if we know that our text file contains the word **spoon**, and that **spoon** occurs four times in the file, we also know one of the rows in our output should look like this:

```
spoon	4
```

In order for a MapReduce program to work, we need three files, namely

1. A mapper file (mapper.py),
2. A reducer file (reducer.py),
3. An input file (file01.txt) we have created in Part 1.

Writing the Mapper program

The Mapper program will read lines from **stdin** (standard input) and write a key/value pair to **stdout** (standard output). For example, a key/value pair contains a key (a word), and a value (in our case, the value is **1**).

The input is the lines from our text file, and those lines need to be split up into words.

The output is lines of text following processing. As we said earlier, the output is tab-separated key/value pairs, so each line will have three components; a word, a tab and the number 1 (denoting this word has just occurred once).

Use gedit to create a mapper.py file in the src directory.

```
$ gedit /home/prac/prac2/src/mapper.py
```

```
#!/usr/bin/env python3

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
        # remove leading and trailing whitespace
        line = line.strip()
        # split the line into words
        words = line.split()

        # increase counters
        for word in words:
                # write the results to STDOUT (standard output);
                # what we output here will be the input for the
                # Reduce step, i.e. the input for reducer.py
                # tab-delimited; the trivial word count is 1
                print('%s\t%s' % (word, 1))
```

(If you copy / paste the above code to the gedit from the PDF document, you need to edit the content accordingly. Python uses indentation as blocks of the functions.)

We should test this code before we try it with MapReduce, so make the program executable:

```
$ chmod +x /home/prac/prac2/src/mapper.py
```

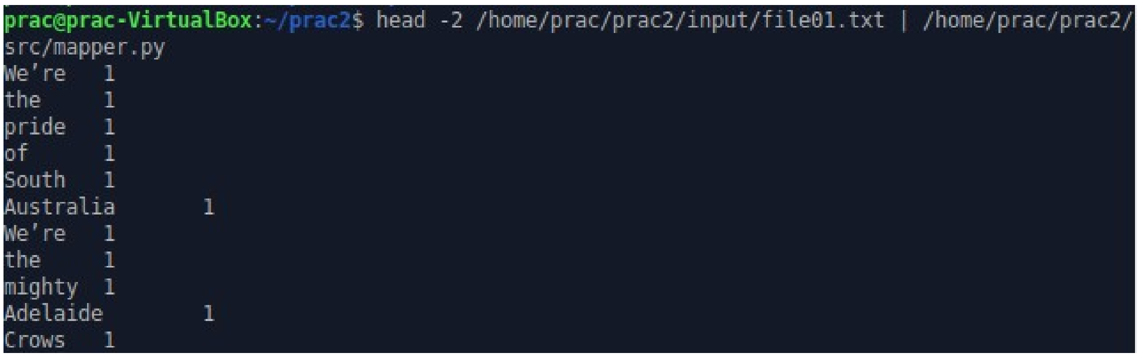
You can now test your python program, pass the text from **file01.txt** to **mapper.py**:

```
$ cat /home/prac/prac2/input/file01.txt |
/home/prac/prac2/src/mapper.py
```

The output of running the program above probably already filled up the window of your terminal. Now imagine if we were working on a 100 MB text file! Our goal is to eventually be working with big data and we should keep that in mind when learning HDFS and MapReduce. For instance, if we want to test our program for the first time, we wouldn't run it on such a massive file. Instead, we can sample a smaller version of the same file. You might remember from the first practical that we can use the **head** command to pass a number of lines from the top of our file. We can then **pipe** this to the mapper file.

```
$ head -2 /home/prac/prac2/input/file01.txt |
/home/prac/prac2/src/mapper.py
```

This gives the following output:



```
prac@prac-VirtualBox:~/prac2$ head -2 /home/prac/prac2/input/file01.txt | /home/prac/prac2/
src/mapper.py
We're   1
the     1
pride   1
of      1
South   1
Australia       1
We're   1
the     1
mighty  1
Adelaide        1
Crows   1
```

If you've obtained this output, your mapper program works!

The vertical line, **|** , that we used above is a symbol for 'piping'. This symbol tells Linux to take the results from the first command (here it was **cat** or **head**) and 'pipe' them in the second command, rather than output them on the screen. You can build a very long chain of commands piping results iteratively. Let's add the sort command to the end of our current chain.

```
$ cat /home/prac/prac2/input/file01.txt |
/home/prac/prac2/src/mapper.py | sort
```

Let's break down what the above command does. First we use **cat** to read the **file01.txt** file and pipe it into the **mapper.py** file. Next results from **mapper.py** are piped to the command **sort**. Finally, results from **sort** will be outputted on the screen.

Writing the Reducer program

The reducer program accepts a stream of lines containing key/value pairs, and sums the values associated with each key. Create a **reducer.py** file in the **src** subdirectory, and enter the code below.

```python
#!/usr/bin/env python3

import sys

# initial values
temp_count = 0
temp_word = None

# input comes from STDIN
for line in sys.stdin:
      # remove leading and trailing whitespace
      line = line.strip()
      # parse the input we got from mapper.py
      word, count = line.split('\t', 1)
```

```
    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        continue

    # check if we get the same word as before
    # if yes, then increase counter;
    # if no, then output old results and start counting again
    if word == temp_word:
        temp_count = temp_count + count
    else:
        if temp_word != None:
            print('%s\t%s'% (temp_word, temp_count))
        temp_word = word
        temp_count = count

# output very last word in the list
print('%s\t%s'% (temp_word, temp_count))
```

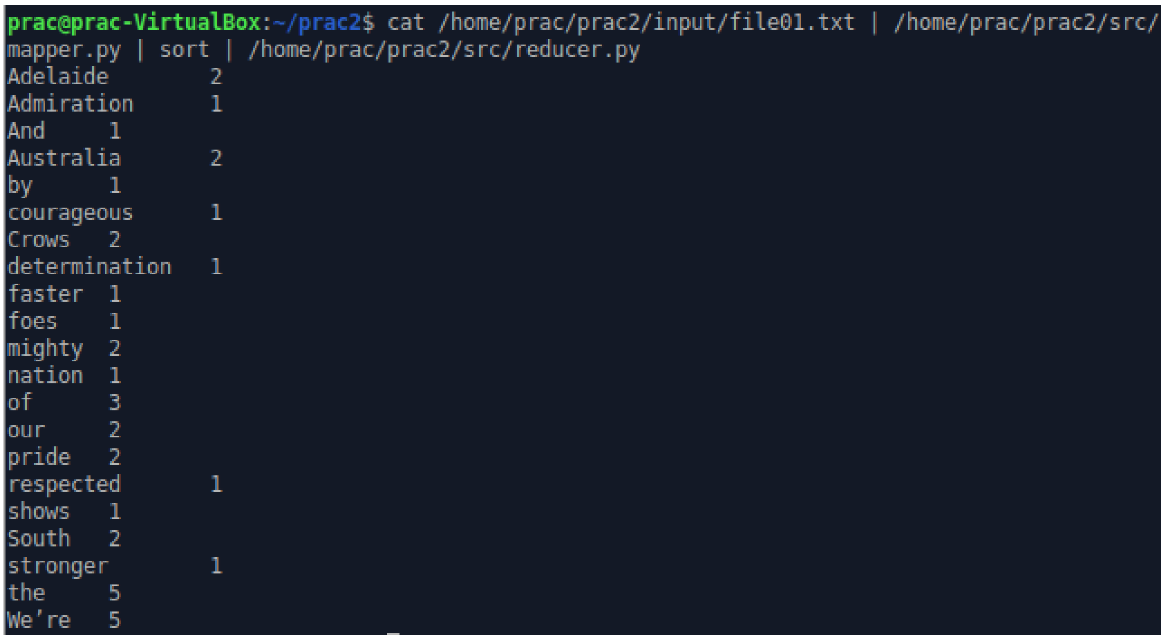
Change the access permissions for reducer.py the same way as you did for mapper.py.

```
$ chmod +x /home/prac/prac2/src/reducer.py
```

Now test the reducer.py program:

```
$ cat /home/prac/prac2/input/file01.txt |
/home/prac/prac2/src/mapper.py | sort |
/home/prac/prac2/src/reducer.py
```

Hopefully the output looks like this:



```
prac@prac-VirtualBox:~/prac2$ cat /home/prac/prac2/input/file01.txt | /home/prac/prac2/src/
mapper.py | sort | /home/prac/prac2/src/reducer.py
Adelaide        2
Admiration      1
And     1
Australia       2
by      1
courageous      1
Crows   2
determination   1
faster  1
foes    1
mighty  2
nation  1
of      3
our     2
pride   2
respected       1
shows   1
South   2
stronger        1
the     5
We're   5
```

Part 3 - Run the MapReduce Program on Hadoop

In order to run our MapReduce program on Hadoop, the text file needs to be uploaded onto HDFS, which Hadoop can access across multiple nodes. We have already uploaded the file01.txt file using the **-put** command in Part 1.

There's another command called **-copyFromLocal** that can achieve the same result. It works for our machine only, so you can not copy from Hadoop to Hadoop, whereas the **-put** command allows you to do whatever you want.

```
$ hadoop fs -copyFromLocal /home/prac/prac2/input/file01.txt
/user/prac/prac2/input
```

To run a MapReduce job with the Hadoop Streaming Utility, you have to pass four parameters to tell Hadoop Streaming: what you are using as the mapper program, what you are using as the reducer program, where the input data is and where the output data should go. Be patient as this may take some time to process.

```
$ mapred streaming \
-input /user/prac/prac2/input/* \
-output /user/prac/prac2/output \
-mapper /home/prac/prac2/src/mapper.py \
-reducer /home/prac/prac2/src/reducer.py
```

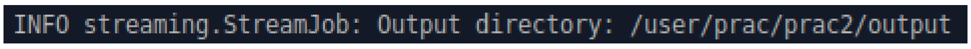
As an aside, in the above code the \ symbol tells Linux that we're using a single input line but it's just actually spanning several lines. You could remove them and run the whole command in one long line.

```
$ mapred streaming -input /user/prac/prac2/input/* -output
/user/prac/prac2/output -mapper /home/prac/prac2/src/mapper.py -
reducer /home/prac/prac2/src/reducer.py
```

(Please note that if you copy/paste the above line to the terminal, it won't work as PDF will auto insert line breaks. Please edit accordingly.)

One final note. Our output folder must **not** already exist. If it does the process will fail.

If everything runs fine and the MapReduce job has been completed, the last message displayed should look like this:

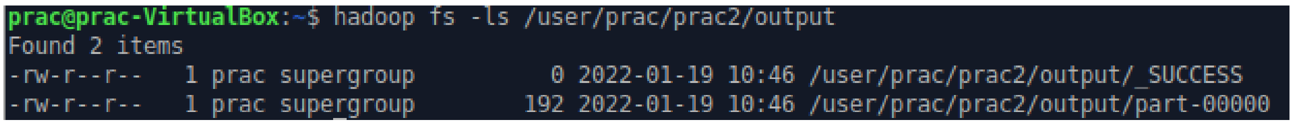


```
INFO streaming.StreamJob: Output directory: /user/prac/prac2/output
```

We can now check the output directory.

```
$ hadoop fs -ls /user/prac/prac2/output
```

And the result should look like this:

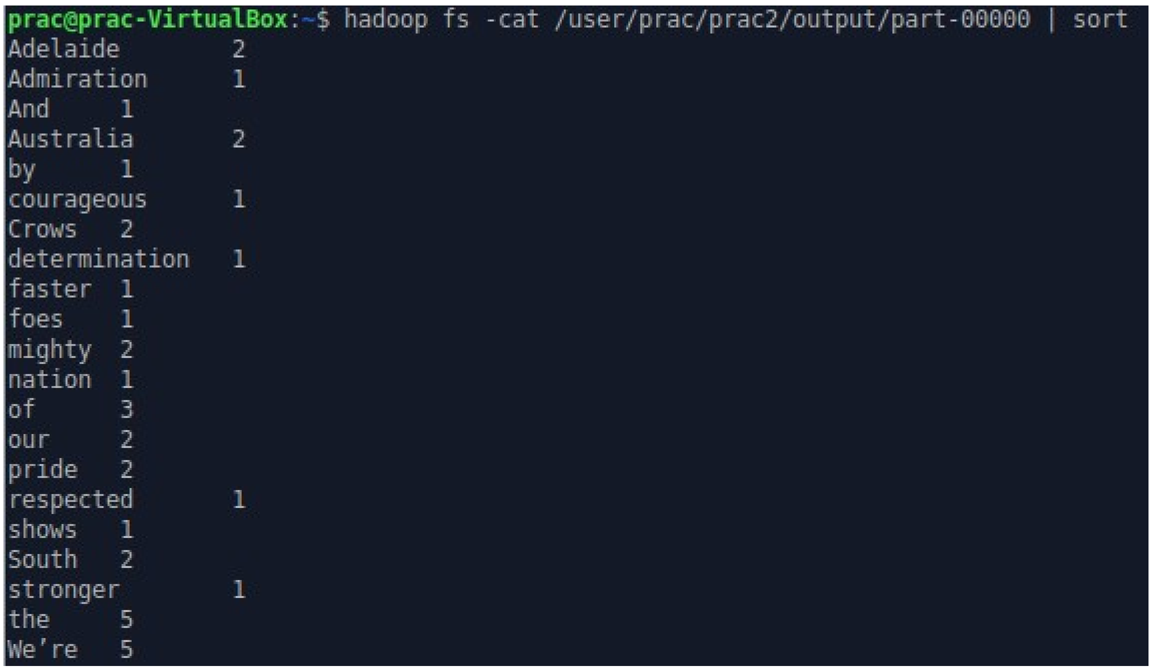


```
prac@prac-VirtualBox:~$ hadoop fs -ls /user/prac/prac2/output
Found 2 items
-rw-r--r--   1 prac supergroup          0 2022-01-19 10:46 /user/prac/prac2/output/_SUCCESS
-rw-r--r--   1 prac supergroup        192 2022-01-19 10:46 /user/prac/prac2/output/part-00000
```

To view the contents of the results file:

$ hadoop fs -cat /user/prac/prac2/output/part-00000 | sort

And again, it should look like this:



```
prac@prac-VirtualBox:~$ hadoop fs -cat /user/prac/prac2/output/part-00000 | sort
Adelaide        2
Admiration      1
And     1
Australia       2
by      1
courageous      1
Crows   2
determination   1
faster  1
foes    1
mighty  2
nation  1
of      3
our     2
pride   2
respected       1
shows   1
South   2
stronger        1
the     5
We're   5
```

Part 4 - Copy the output from HDFS

Finally, we can copy the results from HDFS back to our Linux file system. Begin by creating the **output** subdirectory inside our **prac2** directory in Linux.

```
$ mkdir /home/prac/prac2/output
```

Then, get the output file/s from HDFS:

```
$ hadoop fs -get /user/prac/prac2/output/part*
/home/prac/prac2/output/
```

Now that we've made a copy on Linux, we can erase the files from HDFS:

```
$ hadoop fs -rm -r /user/prac/prac2/output
```

Why did we have to add the extra -r ? It's in the documentation.

Apart from general cleanliness, we remove the folder in HDFS because, once again, if the folder exists when we try to run a job it will not work. If the specific output folder we're referencing already exists MapReduce will not run a new job. This is a protective measure, created to stop us from accidentally overwriting the results of a previous job. So every time you run a new MapReduce job make sure to either:

- delete the previous output folder after copying the results, or
- specify a new output folder like **output2**, for example.

Useful Resources

https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html

http://techalpine.com/how-hadoop-streaming-works/