Practical 4

Today you will use Apache Spark in this practical.

The native language for Spark is Scala, but we'll use Python. There's a Python API for Spark called PySpark, and you can run it directly in your terminal window in a similar way to the previous tools we have used:

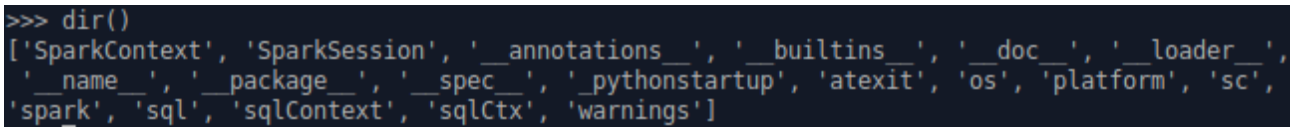
```
$ pyspark
```

The Spark prompt will look a bit fancier than what we're used to though…

From here you can see that there are two modules already loaded and available, there's a pyspark.SparkContext as '**sc**' and a pyspark.sql.SparkSession as '**spark**'.

You can use the command **dir()** to see what methods are available to you.

```
>>> dir()
```



Also, the majority of other Python commands will work within PySpark too. We can see that SparkContext and its alias sc are ready to go, you can again use the dir command to see what methods are available in that module.

```
>>> dir(SparkContext)
```

And as you would expect this command yields the same result.

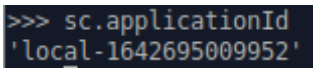
```
>>> dir(sc)
```

The official documentation can be found here where you'll find an in-depth description about what each command does. You will also see a list of commands for the other classes, besides SparkContext.

https://spark.apache.org/docs/3.2.0/api/python/

Lets try some commands now. Please note that commands are case-sensitive.

```
>>> sc.applicationId
```



It returns a unique identifier for the Spark application. We can see that our Spark application is running in '**local**' mode i.e. the Spark Standalone Manager. If it were to be using YARN as a resource manager, the string preceding the long list of numbers would say 'application' instead. We can define which cluster type to use when we make a program. For now, we will stick with the simple-to-use Spark Standalone Manager.

You will need a dataset of movie information from GroupLens, a research lab in the Department of Computer Science and Engineering at the University of Minnesota. You can download the dataset from:

https://grouplens.org/datasets/movielens/1m/

Download the **ml-1m.zip** file and unzip it. The data files are ending with a .dat file extension. You can find the data dictionary in the **README** file. Move a copy of the **users.dat** file saved from the 1M Movie dataset and place it in a local directory **/home/prac/prac4/input** as well as on HDFS in a **/user/prac/prac4/input** directory.

Now you can load the data in Spark using **sc.textFile()** method.

```
>>> data = sc.textFile("/user/prac/prac4/input/users.dat")
```

By default, textFile is assuming that your file resides on HDFS. If your file is in a local directory, you need to specify that by adding file:// to the start of the file's path.

```
>>> data = sc.textFile("file:///home/prac/prac4/input/users.dat")
```

Next use the type command to get a look at what type of variable data is.

```
>>> type(data)
```

From the output you can see that the variable **data** is already an RDD. RDD is short for Resilient Distributed Dataset, which is Spark's fundamental data structure, based on partitioning and parallelising. By default, the **textFile** command loads file as RDD's into Spark.

There are a number of methods available to us for inspecting data. First, we could use the **collect()** command to look at the whole **data** variable at once. Use this at your own peril, as it will literally print the entirety of the data, and should only be used for files that are small enough to fit in the memory of your computer (not cluster). Furthermore, it will take a lot longer to run than most commands, as it relies on pulling all the pieces of data from all of the nodes on your cluster. The **collect()** command is much better suited to showing the results of some calculation, where the output will be of a size much smaller than the original data.

```
>>> data.collect()
```

Instead of the **collect()** command we can get some smaller summaries of our text file data. This command will show the first line of the text file.

```
>>> data.first()
```

This command will show the first 5 lines of the text file.

```
>>> data.take(5)
```

This command will return a fixed-size sampled subset of the RDD. The first input is *withReplacement*, which determines whether sampled elements are replaced back into the dataset. The second input is the size of the subset to be sample and the final input is the random seed (this input is optional).

```
>>> data.takeSample(False,5,3)
```

This command will count the number of elements in an RDD. In this case it is the number of users. Keep in mind that our text file, loaded as an RDD, does not yet discriminate between the variables that we can easily interpret by looking at the file. Instead, each row is saved as a single element.

```
>>> data.count()
```

The **union()** command can be used to return the union of two RDD data.

```
>>> data2 = data.union(data)
```

Now use the **count()** command to check that **data2** is twice the size of **data**.

```
>>> data2.count()
```

Most of the time we'll want to use **textFile()** to parallelize (make RDD's out of) our external data. For two reasons, 1) normally that's just how we get data, given to us in some txt or dat file, but 2) if we try to generate data internally, creating our own RDD, it requires that the whole thing fits in memory. The second point doesn't really fit in with our ideas for big data. There is a method for doing what the second point describes though, it's called **parallelize()**, and it's mostly used for prototyping and testing. Let's parallelize some testing data now.

```
>>> data3 = [1,2,3,4]
>>> rdd = sc.parallelize(data3)
```

And take a look at its contents:

```
>>> rdd.collect()
```

We can also check how many partitions our new RDD has been divided into.

```
>>> rdd.getNumPartitions()
```

Try using **getNumPartitions()** on the RDD you made earlier using the **users.dat** file. Did the output align with your expectations?

There are a few other commands we can use to help generate and manipulate testing data. The standard python **range()** command can be used to generate lists.

```
>>> data4 = range(100)
```

Print the contents of this list simply by typing **data4** in the terminal. Then make another list using range.

```
>>> data5 = range(5,105)
```

Use the **len()** command as you would in Python to make sure that both lists are the same length.

Now lets turn them both into RDDs.

```
>>> rdd1 = sc.parallelize(data4)
>>> rdd2 = sc.parallelize(data5)
```

We can use the zip command to zip one RDD with another one. It returns a key value pair where the first key and value are the elements of each RDD, the second key and value are the second elements

of each RDD etc. As you might have inferred, this requires that each RDD has the same number of partitions as well as the same number of elements in each partition. Normally these conditions will be achieved through some transformation like mapping one RDD onto another (which we'll learn more about next practical), fortunately, we happened to make such a pair of RDD's with the new **data4** and **data5**.

```
>>> rdd3 = rdd1.zip(rdd2)
>>> rdd3.collect()
```

Read through the common RDD Tranformations and Actions in a PDF made by Databricks. It has visual expanation of how the Spark Operations work.

https://training.databricks.com/visualapi.pdf

Practices

Once you get an idea of how Spark RDD works, try to finish a few practices.

Create another text file in your **/home/prac/prac4/input** directory with whatever name and whatever contents you'd like. Make sure you have a few words per line and at least a few lines. If you're struggling to come up with something, you can use the text file we used in Practical 2 when doing the Hadoop Streaming exercise.

Now start a PySpark session in the terminal and try to do the following. Sample solutions are provided at the end of the practical.

Q1

Create a list called data that has the numbers 1 to 5. Convert that list into an RDD called RDD1. Use the map function to create a new RDD called RDD2 that adds one to each element of RDD1. Collect RDD2 to check your results.

Q2

Create a new RDD called RDD3 that only contains the even numbers from RDD2. Collect RDD3 to check your results.

Q3

Read the text file you created earlier and store it in an RDD called lines. It's a good idea to check whether the text document we're working with has any empty lines and to remove them. Use the count action to check how many rows are in the lines RDD. Now try using the appropriate transformation to keep only lines that aren't empty.

Q4 (Harder, uses a transformation we haven't seen yet)

It turns out, that rather than make intermediate RDD's for every step, we can pipe a number of transformations together in the same command. Recall that when we first discovered Pig one of the main draws was how much easier it was to write a word count program than it was in MapReduce? We wrote a script with a handful of commands and it achieved the same as the large Mapper and Reduce programs we used to write. In Spark, we can do it in one line:

```
>>> counts = lines.flatMap(lambda line: line.split(" "))
.map(lambda word: (word, 1)).reduceByKey(lambda v1, v2: v1 + v2)
```

But what if we wanted to count the number of characters in the document instead? Can you do it in one line?

Solutions

Q1

```
data = [1, 2, 3, 4, 5]
RDD1 = sc.parallelize(data)
RDD2 = RDD1.map( lambda x: x+1 )
RDD2.collect()
```

Q2

```
RDD3 = RDD2.filter( lambda x: x%2 == 0)
RDD3.collect()
```

Q3

```
lines = sc.textfile("file:///home/prac/prac4/input/test_text.txt")
lines.count()
lines_nonempty = lines.filter( lambda x: len(x) > 0)
lines_nonempty.count()
```

Q4

```
char_count = lines.map( lambda x: len (x)).reduce(lambda x, y: x+y)
```

Useful Resources

https://spark.apache.org/docs/3.2.0/api/python/

https://training.databricks.com/visualapi.pdf