4 Delivery: Spark

Objective

We are introducing the use of Spark console and the basic operations with the Scala API to process datasets stored in a HDFS repository.

This tutorial is based on the public Apache Spark documentation. More details on operations and Spark usage is available on `spark.apache.org` website.

4.1 Working with the Spark interpreter

To work with Scala we need to open a console and open the Spark interactive shell:

```
spark-shell
```

We should get the following welcome message:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/
Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
   1.7.0_67)
```

Alternatively, we can use PySpark if we are more familiar with Python's syntax:

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Python version 2.6.6 (r266:84292, Jul 23 2015 15:22:56)
```

4.2 Counting words with Spark

We will test two implementations of a Word Count using (a) Scala, and (b) PySpark. The implementations can be seen, respectively, in snippets 1 and 2 below.

Snippet 1: Word count implementation using Scala

```
val f = sc.textFile("/hduser/input/pg2000.txt")
val wc = f.flatMap(l => l.split("␣")).
           map(word => (word, 1)).
           reduceByKey(_ + _)
wc.saveAsTextFile("/hduser/spark/wc_scala_out")
```

Now let's check the output:

```
hdfs dfs -cat /hduser/spark/wc_scala_out/part-00000 | head -n5
```

```
(agradable,,1)
(derecho.,1)
(vecindad,1)
(invierno,6)
(deshonrar,3)
```

Snippet 2: Word count implementation using PySpark

```
from operator import add
f = sc.textFile("/hduser/input/pg2000.txt")
wc = f.flatMap(lambda x: x.split(' ')).
        map(lambda x: (x, 1)).
        reduceByKey(add)
wc.saveAsTextFile("/hduser/spark/wc_pyspark_out")
```

Now let's check the output:

```
hdfs dfs -cat /hduser/spark/wc_pyspark_out/part-00000 | tail -n5
```

```
(u'-Voy', 1)
(u'cupo,', 1)
(u'quit\xe1rselos', 2)
(u'arrojarlos', 1)
(u'embotan', 1)
```

Comparing the outputs of snippets 1 and 2, it's clear that both implementation works, but the format of the outputs is completely different.

Scala seems to be a tad more intuitive and seems to ease the eventual process of post-processing the results.

4.3 Working with Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) are the primary abstraction in Spark, a fault-tolerant collection of elements that can be operated on in parallel. There are currently two types:

(i) Parallelized collections: take an existing Scala collection and run functions on it in parallel:

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

(ii) Hadoop datasets: run functions on each record of a file in Hadoop distributed file system or any other storage system supported by Hadoop:

```
val distFile =
    sc.textFile("/hduser/input-weather/weather-data1901.txt")
```

```
distFile.map(l => l.split("␣")).collect()
```

```
res0: Array[Array[String]] = Array(
Array(00290290709999919010101060004+64333+023450...),
Array(00290290709999919010101130004+64333+023450...),
Array(00290290709999919010101200004+64333+023450...),
Array(00290290709999919010102060004+64333+023450...),
Array(00290290709999919010102130004+64333+023450...),
Array(00290290709999919010102200004+64333+023450...
```

```
distFile.flatMap(l => l.split("␣")).collect()
```

```
res1: Array[String] = Array(
00290290709999919010101060004+64333+023450...,
00290290709999919010101130004+64333+023450...,
00290290709999919010101200004+64333+023450...,
00290290709999919010102060004+64333+023450...,
00290290709999919010102130004+64333+023450...,
00290290709999919010102200004+64333+023450...
```

The difference between the behaviour between `map()` and `flatMap()` is that `map()` passes each element of the source through our mapper function (we get an `Array` of `Array`s), whilst `flatMap()` passes them all at once (we get a single `Array`).

Type and understand the following source code in your Spark console. You must upload `reg.tsv` and `clk.tsv` at your own HDFS data input folder.

```
hdfs dfs -mkdir /hduser/input-spark
hdfs dfs -put ~/data/clk.tsv /hduser/input-spark/clk.tsv
hdfs dfs -put ~/data/reg.tsv /hduser/input-spark/reg.tsv
```

```
val format = new java.text.SimpleDateFormat("yyyy-MM-dd")
```

```
format: java.text.SimpleDateFormat =
   java.text.SimpleDateFormat@f67a0200
```

Now we define our `Register` and `Click` classes:

```
case class Register (d: java.util.Date, uuid: String,
   cust_id:String, lat: Float, lng: Float)
case class Click (d: java.util.Date, uuid: String, landing_page:Int)
```

Now we read the `reg.tsv` from our HDFS and split the lines using the tabulator as a separator; then we give format to each line as an array with the different fields we are interested into using a map job and the `Register` class:

```
val reg = sc.textFile("/hduser/input-spark/reg.tsv").
             map(_.split("\t")).
             map( r => (r(1), Register(format.parse(r(0)), r(1),
                r(2), r(3).toFloat, r(4).toFloat)) )
```

Then we read the `clk.tsv` from our HDFS and split the lines using the tabulator as a separator; then we give format to each line as an array with the different fields we are interested into using a map job and the `Click` class:

```
4 val clk = sc.textFile("/hduser/input-spark/clk.tsv").
5              map(_.split("\t")).
6              map( c => (c(1), Click(format.parse(c(0)), c(1),
                  c(2).trim.toInt)))
```

We can use the **`join()`** transformation to join `reg` with `clk` and then return all the elements of the dataset as an array using `collect()`:

```
5 reg.join(clk).collect()
```

```
res0: Array[(String, (Register, Click))] = Array(
(81da510acc4111e387f3600308919594,(Register(Tue Mar 04 00:00:00 PST
   2014,81da510acc4111e387f3600308919594,2,33.85701,-117.85574),
   Click(Thu Mar 06 00:00:00 PST
   2014,81da510acc4111e387f3600308919594,61))),
(15dfb8e6cc4111e3a5bb600308919594,(Register(Sun Mar 02 00:00:00 PST
   2014,15dfb8e6cc4111e3a5bb600308919594,1,33.659943,-117.95812),
   Click(Tue Mar 04 00:00:00 PST
   2014,15dfb8e6cc4111e3a5bb600308919594,11))))
```

This final order prints the RDD Lineage Graph as a DAG (Directed Acyclic Graph) to get the dataset **`reg.join(clk)`**. The indentations indicate a shuffle boundary.

```
6 reg.join(clk).toDebugString
```

```
res1: String =
(1) MapPartitionsRDD[30] at join at <console>:38 []
 |  MapPartitionsRDD[29] at join at <console>:38 []
 |  CoGroupedRDD[28] at join at <console>:38 []
 +-(1) MapPartitionsRDD[20] at map at <console>:33 []
 |  |  MapPartitionsRDD[19] at map at <console>:32 []
 |  |  /hduser/input-spark/reg.tsv MapPartitionsRDD[18] at textFile
    at <console>:31 []
 |  |  /hduser/input-spark/reg.tsv HadoopRDD[17] at textFile at
    <console>:31 []
 +-(1) MapPartitionsRDD[24] at map at <console>:33 []
    |  MapPartitionsRDD[23] at map at <console>:32 []
    |  /hduser/input-spark/clk.tsv MapPartitionsRDD[22] at textFile
       at <console>:31 []
    |  /hduser/input-spark/clk.tsv HadoopRDD[21] at textFile at
       <console>:31 []
```

4.4 Processing weather data with Spark

The implementation of the mapreduce job to find the Minimum Temperature of each year is quite straightforward; it is very similar to what we did in Hadoop, but with a much simpler syntax. This implementation can be seen in snippet 3 below.

It is of crucial importance to apply the `filter(line => line.length >= 93)` transformation, otherwise Java will complain about some lines that are shorter than expected (a problem one would not have using PySpark, for instance, as it would automatically discard those lines).

Snippet 3: Minimum Temperature implementation using Scala

```
val inputRDD = sc.textFile("/hduser/input-weather/*")

val minTemp = inputRDD.filter(line => line.length >= 93).
    map{ case line =>
            val quality: String = line.substring(92, 93)
            val sign: Int = if(line.charAt(87) == '+') 1 else -1
            val temp: Int = line.substring(88, 92).toInt
            val year: Int = line.substring(15, 19).toInt
            (quality, year, sign, temp) }.
    filter { case (q, _, _, t ) => q.matches("[01459]") && t !=
       9999}.
    map { case (_, y, s, t) => ( y, s*t) }.
    reduceByKey({ case (prev, curr) => Math.min(prev, curr) })

minTemp.saveAsTextFile("/hduser/spark/output-min-temp")
```

Now we only need to check the output of our Scala mapreduce job. The results are exactly the same that those obtained using Java classes or Python in Hadoop.

```
hdfs dfs -cat /hduser/spark/output-min-temp/part*
```

```
(1920,-344)
(1901,-333)
(1902,-328)
(1903,-306)
(1904,-294)
(1905,-328)
(1906,-250)
(1907,-350)
(1908,-378)
(1909,-378)
(1910,-372)
(1911,-378)
(1912,-411)
(1913,-372)
(1914,-378)
(1915,-411)
(1916,-289)
(1917,-478)
(1918,-450)
(1919,-428)
```