RDDs

Primero almacenamos los datos en una variable, luego contamos las lineas con un count, luego almacenamos todo el contenido seguido en un array de strings con el metodo collect y finalmente ponemos todas las lineas usando un foreach:

```
scala> relato.count()
res0: Long = 23

scala> relato.collect()
res1: Array[String] = Array(Two roads diverged in a yellow wood,, And sorry I could not travel both, And be one traveler, long I st
 as just as fair,, And having perhaps the better claim,, Because it was grassy and wanted wear;, Though as for that the passing the
dden black., Oh, I kept the first for another day!, Yet knowing how way leads on to way,, I doubted if I should ever come back., ""
--, I took the one less traveled by,, And that has made all the difference.)

scala> relato.foreach(println)
Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as for that the passing there
Had worn them really about the same,
```

Creamos una variable en la que almacenamos un solo log pero, luego creamos otra que será un RDD en la que almacenaremos ese log:

```
scala> val log="file:/home/BIT/data/weblogs/2013-09-15.log"
log: String = file:/home/BIT/data/weblogs/2013-09-15.log

scala> val logs = sc.textFile(log)
logs: org.apache.spark.rdd.RDD[String] = file:/home/BIT/data/weblogs/2013-09-15.log MapPartitionsRDD[3] at textFile at <console>:29
```

Pair RDDs

Almacenamos en una variable los datos que vamos a usar que en este caso son logs:

```
scala> var logs=sc.textFile("file:/home/BIT/data/weblogs/*")
logs: org.apache.spark.rdd.RDD[String] = file:/home/BIT/data/weblogs/* MapPartitionsRDD[1] at textFile at <console>:27
```

Después vamos a mapear esos logs y luego reducirlos de manera que por una parte tienes una palabra de los logs y por otra parte tienes otra palabra para luego juntarlas ambas:

```
scala> var userreqs = logs.map(line => line.split(' ')).map(words => (words(2),1)).reduceByKey((v1,v2) => v1+v2)
userreqs: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:29
```

Ahora queremos obtener los id de usuario con el mayor numero de accesos así que cambiamos la clave por el valor con un swap para luego poder poder hacer un sort by key y luego a la hora de mostrar los datos volvemos a poner la id y el valor en su sitio por lo que en el segundo comando (en el que se muestra) pues tienes otro swap pava ver id y numero de accesos:

```
scala> val swapped=userreqs.map(field => field.swap)
swapped: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[5] at map at <console>:31

scala> swapped.sortByKey(false).map(field => field.swap).take(10).foreach(println)
(193,1603)
(77,1547)
(119,1540)
(34,1526)
(182,1524)
(64,1508)
(189,1508)
(20,1502)
(173,1500)
(17,1500)
```

A continuación, creamos un rdd para almacenar la id de un usuario junto con las ip con las que se ha conectado. Pues creas una variable en la que mapeas las líneas de los logs y luego coges las líneas con la ip y con la id para agrupar por la clave, luego coges 10 con la función take:

```
scala> var userips = logs.map(line => line.split(' ')).map(words => (words(2), words(0))).groupByKey()
userips: org.apache.spark.rdd.RDD[(String, Iterable[String])] = ShuffledRDD[12] at groupByKey at <console>:29

scala> userips.take(10)
res1: Array[(String, Iterable[String])] = Array((79844,CompactBuffer(136.132.254.160, 136.132.254.160, 53.251.68.51, 53.251.68.51)),
, 17.159.12.204, 17.159.12.204, 96.24.214.109, 96.24.214.109, 123.79.96.8, 123.79.96.8, 20.117.86.221, 20.117.86.221, 142.96.254.175
5.209.169.137, 15.209.169.137, 15.209.169.137, 15.209.169.137, 51.239.242.13, 51.239.242.13, 51.239.242.13, 51.239.242.13, 18.76.240
.150.177.226, 215.150.177.226, 215.150.177.226, 39.175.103.131, 39.175.103.131, 102.17...
scala>
```

Con un nuevo dataset mapeamos los registros para organizarlos y que tengan una clave y un valor, despues hacemos un join para unir este nuevo rdd y el anterior, después creas un nuevo rdd a partir de la fusion de los anteriores que contendrá el id, nombre, apellido por lo que usamos un bucle:

```
scala> var accounts = sc.textFile("file:/home/BIT/data/accounts.csv").map(line => line.split(',')).map(account => (account(0), account))
accounts: org.apache.spark.rdd.RDD[(String, Array[String])] = MapPartitionsRDD[16] at map at <console>:27

scala> var accounthits = accounts.join(userreqs)
accounthits: org.apache.spark.rdd.RDD[(String, (Array[String], Int))] = MapPartitionsRDD[19] at join at <console>:33

scala> for(pair <- accounthits.take(10)){println(pair._1,pair._2._2,pair._2._1(3),pair._2._1(4))}
(34344,8,Michael,Herron)
(28996,8,Charles,Adamson)
(104230,6,Kathy,Vanwormer)
(31208,8,John,Stoddard)
(100135,6,Robert,Estevez)
(31572,6,Clifford,Andrews)
(19497,70,Michael,Oconnell)
(10054,64,Tom,McKenzie)
(26875,18,Brittany,Evans)
(69386,14,Terry,Atkinson)
```

Creamos un RDD en el que el código postal es la clave, después creamos un RDD de pares con el código postal y el nombre al que corresponden, finalmente ordenamos por código postal y ponemos los nombres atribuidos a dicho número postal:

```
scala> var accountsByPCode = sc.textFile("file:/home/BIT/data/accounts.csv").map(_.split(',')).keyBy(_(8))
accountsByPCode: org.apache.spark.rdd.RDD[(String, Array[String])] = MapPartitionsRDD[23] at keyBy at <console>:27

scala> var namesByPCode = accountsByPCode.mapValues (values => values(4) + ',' + values(3)).groupByKey()
namesByPCode: org.apache.spark.rdd.RDD[(String, Iterable[String])] = ShuffledRDD[25] at groupByKey at <console>:29
```

```
scala> namesByPCode.sortByKey().take(10).foreach{
     | case(x,y) => println ("---" + x)
     | y.foreach(println)};
21/05/24 04:33:07 WARN memory.TaskMemoryManager: leak 17.2 MB memory from org.apache.spark.util.collection.ExternalSorter@40f9c505
21/05/24 04:33:07 ERROR executor.Executor: Managed memory leak detected; size = 17997936 bytes, TID = 736
---85000
Willson,Leon
Clark,Ronald
Rush,Juanita
Woodhouse,Roger
Baptist,Colin
King,Percy
Carmack,David
Milan,Ana
McCurdy,Kendra
Pitts,Robert
Hopkins,Leslie
Butler,Paul
Barth,Phyllis
---85001
```

SparkSQL (JSON)

Se crea un nuevo contexto y se importan los implicits que permiten convertir RDDs en DataFrames:

```
scala> var ssc = new org.apache.spark.sql.SQLContext (sc)
ssc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@4c9871fc

scala> import sqlContext.implicits._
import sqlContext.implicits._
```

Cargamos los datos, los mostramos y filtramos:

```
scala> var zips = ssc.load("file:/home/BIT/data/zips.json", "json")
warning: there were 1 deprecation warning(s); re-run with -deprecation for details
zips: org.apache.spark.sql.DataFrame = [_id: string, city: string, loc: array<double>, pop: bigint, state: string]

scala> zips.show()
+-----+---------------+--------------------+-----+-----+
|  _id|           city|                 loc|  pop|state|
+-----+---------------+--------------------+-----+-----+
|01001|         AGAWAM|[-72.622739, 42.0...|15338|   MA|
|01002|        CUSHMAN|[-72.51565, 42.37...|36963|   MA|
|01005|          BARRE|[-72.108354, 42.4...| 4546|   MA|
|01007|    BELCHERTOWN|[-72.410953, 42.2...|10579|   MA|
|01008|      BLANDFORD|[-72.936114, 42.1...| 1240|   MA|
|01010|      BRIMFIELD|[-72.188455, 42.1...| 3706|   MA|
|01011|        CHESTER|[-72.988761, 42.2...| 1688|   MA|
|01012|   CHESTERFIELD|[-72.833309, 42.3...|  177|   MA|
|01013|       CHICOPEE|[-72.607962, 42.1...|23396|   MA|
|01020|       CHICOPEE|[-72.576142, 42.1...|31495|   MA|
|01022|   WESTOVER AFB|[-72.558657, 42.1...| 1764|   MA|
|01026|     CUMMINGTON|[-72.905767, 42.4...| 1484|   MA|
|01027|      MOUNT TOM|[-72.679921, 42.2...|16864|   MA|
|01028|EAST LONGMEADOW|[-72.505565, 42.0...|13367|   MA|
|01030|  FEEDING HILLS|[-72.675077, 42.0...|11985|   MA|
|01031|   GILBERTVILLE|[-72.198585, 42.3...| 2385|   MA|
|01032|         GOSHEN|[-72.844092, 42.4...|  122|   MA|
|01033|         GRANBY|[-72.520001, 42.2...| 5526|   MA|
|01034|        TOLLAND|[-72.908793, 42.0...| 1652|   MA|
|01035|         HADLEY|[-72.571499, 42.3...| 4231|   MA|
+-----+---------------+--------------------+-----+-----+
only showing top 20 rows


scala> zips.filter(zips("pop") > 10000).collect()
res1: Array[org.apache.spark.sql.Row] = Array([01001,AGAWAM,WrappedArray(-72.622739, 42.070206),15338,MA], [01002,CUSH
013,CHICOPEE,WrappedArray(-72.607962, 42.162046),23396,MA], [01020,CHICOPEE,WrappedArray(-72.576142, 42.176443),31495,
.067203),13367,MA], [01030,FEEDING HILLS,WrappedArray(-72.675077, 42.07182),11985,MA], [01040,HOLYOKE,WrappedArray(-72
y(-72.654245, 42.324662),27939,MA], [01075,SOUTH HADLEY,WrappedArray(-72.581137, 42.23...
scala>
```

Es equivalente hacer la consulta con sql o mediante filter

```
scala> zips.filter(zips("pop") > 10000).collect()
res1: Array[org.apache.spark.sql.Row] = Array([01001,AGAWAM,WrappedArray(-72.622739, 42.070206),15338,
013,CHICOPEE,WrappedArray(-72.607962, 42.162046),23396,MA], [01020,CHICOPEE,WrappedArray(-72.576142, 4
.067203),13367,MA], [01030,FEEDING HILLS,WrappedArray(-72.675077, 42.07182),11985,MA], [01040,HOLYOKE,
y(-72.654245, 42.324662),27939,MA], [01075,SOUTH HADLEY,WrappedArray(-72.581137, 42.23...
scala> zips.registerTempTable ("zips")

scala> ssc.sql("select * from zips where pop > 10000").collect()
res3: Array[org.apache.spark.sql.Row] = Array([01001,AGAWAM,WrappedArray(-72.622739, 42.070206),15338,
013,CHICOPEE,WrappedArray(-72.607962, 42.162046),23396,MA], [01020,CHICOPEE,WrappedArray(-72.576142, 4
.067203),13367,MA], [01030,FEEDING HILLS,WrappedArray(-72.675077, 42.07182),11985,MA], [01040,HOLYOKE,
y(-72.654245, 42.324662),27939,MA], [01075,SOUTH HADLEY,WrappedArray(-72.581137, 42.23...
scala>
```

Consultas normales en tablas:

```
scala> ssc.sql("select sum(pop) as POPULATION from zips where state='WI'").show()
+----------+
|POPULATION|
+----------+
|   4891769|
+----------+


scala> ssc.sql("select * from zips where pop > 10000").collect()
res7: Array[org.apache.spark.sql.Row] = Array([01001,AGAWAM,WrappedArray(-72.622739, 42.07
013,CHICOPEE,WrappedArray(-72.607962, 42.162046),23396,MA], [01020,CHICOPEE,WrappedArray(-
.067203),13367,MA], [01030,FEEDING HILLS,WrappedArray(-72.675077, 42.07182),11985,MA], [01
y(-72.654245, 42.324662),27939,MA], [01075,SOUTH HADLEY,WrappedArray(-72.581137, 42.23...
scala> ssc.sql("select * from zips where pop > 10000")
res8: org.apache.spark.sql.DataFrame = [_id: string, city: string, loc: array<double>, pop

scala> ssc.sql("select state, sum(pop) as POPULATION from zips group by state order by sum
+-----+----------+
|state|POPULATION|
+-----+----------+
|   CA|  29760021|
|   NY|  17990455|
|   TX|  16986510|
|   FL|  12937926|
|   PA|  11881643|
|   IL|  11430602|
|   OH|  10847115|
|   MI|   9295297|
|   NJ|   7730188|
|   NC|   6628637|
|   GA|   6478216|
|   VA|   6187358|
|   MA|   6016425|
|   IN|   5544159|
|   MO|   5114343|
|   WI|   4891769|
|   TN|   4876457|
|   WA|   4866692|
|   MD|   4781468|
|   MN|   4375099|
+-----+----------+
only showing top 20 rows
```

SparkSQL (hive)

Se ha de copiar el hive-site.xml de la carpeta de hive/conf a la carpedta de spark/conf para poder acceder a las tablas de hive desde spark y reiniciar el shell de spark:

```
[cloudera@quickstart ~]$ ls /usr/lib/hive/conf/
beeline-log4j.properties.template  hive-exec-log4j.properties  hive-site.xml
hive-env.sh.template               hive-log4j.properties       ivysettings.xml
[cloudera@quickstart ~]$ sudo cp /usr/lib/hive/conf/hive-site.xml /usr/lib/spark
/conf/
[cloudera@quickstart ~]$ ls /usr/lib/spark/conf/
docker.properties.template   slaves.template
fairscheduler.xml.template   spark-defaults.conf
hive-site.xml                spark-defaults.conf.template
log4j.properties.template    spark-env.sh
metrics.properties.template  spark-env.sh.template
slaves
[cloudera@quickstart ~]$
```

Creamos una variable de contexto y una base de datos a partir de ella:

```
scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext (sc)
sqlContext: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.hive.HiveContext@f010df1

scala> sqlContext.sql("CREATE DATABASE IF NOT EXISTS hivespark")
21/05/25 00:15:42 WARN metastore.ObjectStore: Version information not found in metastore. hive.metastore.schema.v
21/05/25 00:15:43 WARN metastore.ObjectStore: Failed to get database default, returning NoSuchObjectException
res0: org.apache.spark.sql.DataFrame = [result: string]
```

Creamos una tabla y le cargamos unos registros desde un archivo de texto:

```
scala> sqlContext.sql("CREATE TABLE hivespark.empleados(id INT, name STRING, age INT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LINES TERMINATED BY '\n'")
res2: org.apache.spark.sql.DataFrame = [result: string]

scala> sqlContext.sql("LOAD DATA LOCAL INPATH '/home/cloudera/empleado.txt' INTO TABLE hivespark.empleados")
res3: org.apache.spark.sql.DataFrame = [result: string]

scala>
```

Para realizar una consulta la podemos hacer tanto desde spark como desde hive, desde spark tendremos que almacenar la consulta en una variable y luego mostrarla y en hive simplemente podemos poner la consulta:

```
scala> var query1 =sqlContext.sql("select * from hivespark.empleados")
query1: org.apache.spark.sql.DataFrame = [id: int, name: string, age: int]

scala> query1.show()
+----+-------+---+
|  id|   name|age|
+----+-------+---+
|1201|nombre1| 25|
|1202|nombre2| 28|
|1203|nombre3| 39|
|1204|nombre4| 23|
|1205|nombre5| 23|
+----+-------+---+


scala>
```

```
File  Edit  View  Search  Terminal  Help
hive> select * from hivespark.empleados
    > ;
OK
1201    nombre1 25
1202    nombre2 28
1203    nombre3 39
1204    nombre4 23
1205    nombre5 23
Time taken: 0.1 seconds, Fetched: 5 row(s)
hive>
```

SparkSQL (DataFrames)

Para trabajar con dataframes creamos una variable de contexto y realizamos los imports necesarios:

```
scala> var ssc = new org.apache.spark.sql.SQLContext (sc)
ssc: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@6ce20427

scala> import sqlContext.implicits._
import sqlContext.implicits._

scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row

scala> import org.apache.spark.sql.types.{StructType, StructField, StringType}
import org.apache.spark.sql.types.{StructType, StructField, StringType}
```

Creamos una variable con la ruta del dataset y leemos su contenido:

```
scala> var ruta_datos="file:/home/cloudera/Desktop/DataSetPartidos.txt"
ruta_datos: String = file:/home/cloudera/Desktop/DataSetPartidos.txt

scala> var datos = sc.textFile
                                                                def textFile(path: String, minPartitions: Int): rdd.RDD[String]

scala> var datos = sc.textFile(ruta_datos)
datos: org.apache.spark.rdd.RDD[String] = file:/home/cloudera/Desktop/DataSetPartidos.txt MapPartitionsRDD[1] at textFile at <console>:34
```