Session 18 - Introduction to Spark

Assignment 1

Task 1:

Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- find the sum of all numbers

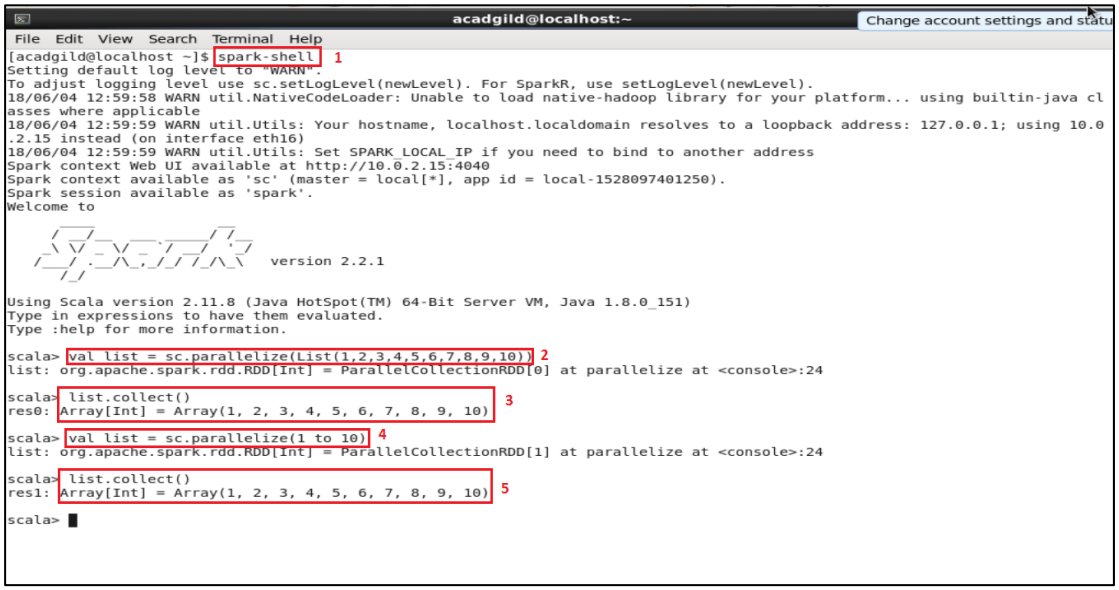
- find the total elements in the list

- calculate the average of the numbers in the list

- find the sum of all the even numbers in the list

- find the total number of elements in the list divisible by both 5 and 3

Creating an RDD from the list :



With reference to the screenshot above,

1 : Enter the spark cmd for scala using the command '*spark-shell*'

2 & 4: Create an RDD out of the list of numbers using the following command. Actually the list can be specified in two ways.

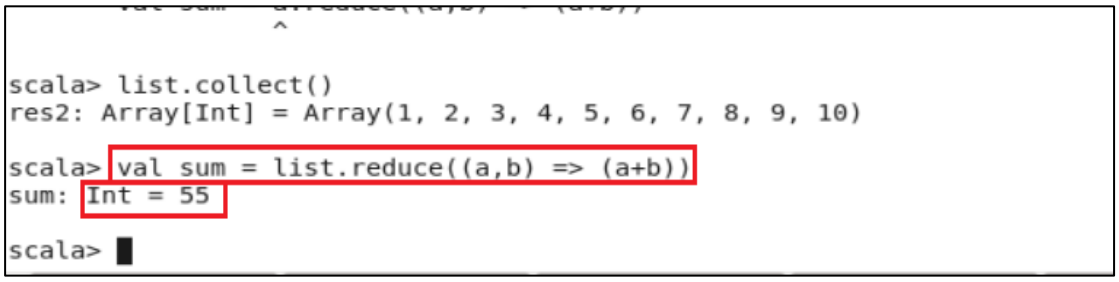
val list = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))

val list = sc.parallelize(1 to 10)

3 &5 : Using either of the ways, an RDD is created. The contents of the list are shown by using the following command.

list.collect()

I) find the sum of all numbers



Command :

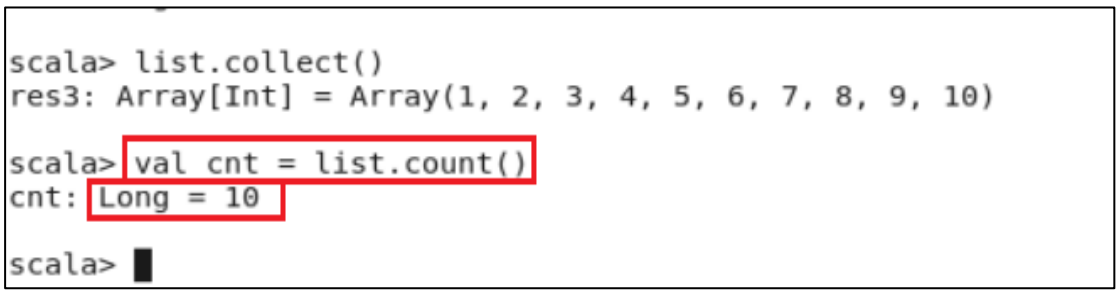
val sum = list.reduce((a,b) => (a+b))

Output :

55

-> sum of all numbers from 1 to10 is 55

II) Find the total elements in the list



Command :

val cnt = list.count()

Output :

10

-> count of numbers is 10

III) Calculate the average of the numbers in the list

```
scala> list.collect()
res4: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val avg = list.reduce((a,b) => (a+b)).toFloat/list.count()
avg: Float = 5.5

scala>
```

Command :

val avg = list.reduce((a,b) => (a+b)).toFloat/list.count()

Output :

5.5

-> sum of all the numbers in the list divided by the count of the list gives the average I.e., 5.5

IV) Find the sum of all the even numbers in the list

```
scala> list.collect()
res5: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val even = list.filter(x => x%2 == 0).reduce((a,b) => (a+b))
even: Int = 30

scala>
```

Command :

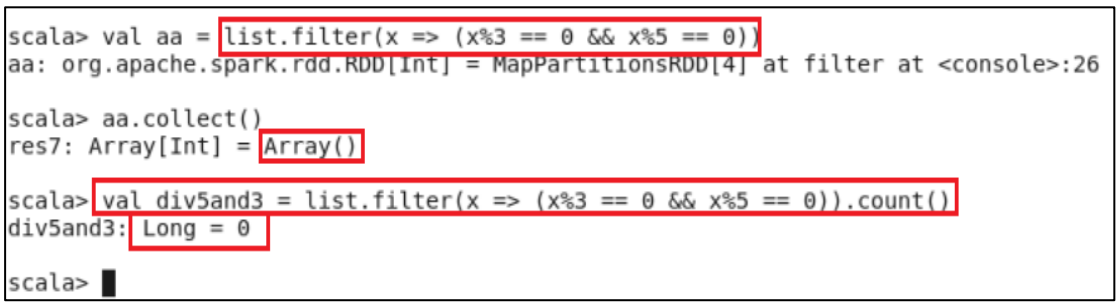
val even = list.filter(x => x%2 ==0).reduce((a,b) => (a+b))

Output :

30

-> filtering out the numbers divisible by 2 and then calculating the sum of those filtered numbers.

V) Find the total number of elements in the list divisible by both 5 and 3



Command :

val even = list.filter(x => (x%3 == 0 && x%5 ==0)).count()

Output :

0

-> filtering out the numbers divisible by both 3 and 5 and then counting those filtered numbers. Since between 1 and 10, there is no number that is divisible by both 3 and 5 hence the Array is empty and the count is 0.

Task 2:

I) Pen down the limitations of MapReduce.

Below are some of the limitations of MapReduce.

Issue with Small Files

Hadoop is not suited for small data. (HDFS) Hadoop distributed file system lacks the ability to efficiently support the random reading of small files because of its high capacity design.

Slow Processing Speed

In Hadoop, with a parallel and distributed algorithm, MapReduce process large data sets. There are tasks that need to be performed: Map and Reduce and, MapReduce requires a lot of time to perform these tasks thereby increasing latency. Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

Support for Batch Processing only

Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower. MapReduce framework of Hadoop does not leverage the memory of the Hadoop cluster to the maximum.

No Real-time Data Processing

Apache Hadoop is designed for batch processing, that means it take a huge amount of data in input, process it and produce the result. Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly. Hadoop is not suitable for Real-time data processing.

No Delta Iteration

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow(i.e. a chain of stages in which each output of the previous stage is the input to the next stage).

Latency

In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data. In MapReduce, Map takes a set of data and converts it into another set of data, where individual element are broken down into key value pair and Reduce takes the output from the map as input and process further and MapReduce requires a lot of time to perform these tasks thereby increasing latency.

Not Easy to Use

In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work. MapReduce has no interactive mode, but adding one such as hive and pig makes working with MapReduce a little easier for adopters.

No Caching

Hadoop is not efficient for caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.

II) What is RDD? Explain few features of RDD?

RDD stands for ‘‘Resilient Distributed Dataset’’. It is the fundamental data structure of Apache Spark. RDD in Apache Spark is an immutable collection of objects which computes on the different node of the cluster.

Decomposing the name RDD:

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph(DAG) and so able to recompute missing or damaged partitions due to node failures.
- **Distributed**, since Data resides on multiple nodes.
- **Dataset** represents records of the data you work with. The user can load the data set externally which can be either JSON file, CSV file, text file or database via JDBC with no specific data structure.

Features :

In-memory Computation

Spark RDDs have a provision of in-memory computation. It stores intermediate results in distributed memory(RAM) instead of stable storage(disk).

Lazy Evaluations

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program. Follow this guide for the deep study of Spark Lazy Evaluation.

Fault Tolerance

Spark RDDs are fault tolerant as they track data lineage information to rebuild lost data automatically on failure. They rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets (by transformations like a map, join or groupBy) to recreate itself. Follow this guide for the deep study of RDD Fault Tolerance.

Immutability

Data is safe to share across processes. It can also be created or retrieved anytime which makes caching, sharing & replication easy. Thus, it is a way to reach consistency in computations.

Partitioning

Partitioning is the fundamental unit of parallelism in Spark RDD. Each partition is one logical division of data which is mutable. One can create a partition through some transformations on existing partitions.

Persistence

Users can state which RDDs they will reuse and choose a storage strategy for them (e.g., in-memory storage or on Disk).

Coarse-grained Operations

It applies to all elements in datasets through maps or filter or group by operation.

Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD.
The DAGScheduler places the partitions in such a way that task is close to data as much as possible. Thus, speed up computation. Follow this guide to learn What is DAG?

III) List down few Spark RDD operations and explain each of them.

RDDs perform two types of operations: **Transformations**, which creates a new dataset from the previous RDD and **Actions**, which return a value to the driver program after performing the computation on the dataset.

Transformations:

Transformations create a new data set from an existing one by passing each dataset element through a function and returns a new RDD representing the results. In short, creating an RDD from an existing RDD is 'transformation'.

All transformations in Spark are lazy. They do not compute their results right away. Instead, they just remember the transformations applied to some base data

set (e.g. a file). The transformations are only computed when an action requires a result that needs to be returned to the driver program.

Map

Map will take each row as input and return an RDD for the row.

```
scala>

scala> val txtfile = sc.textFile("hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv")
txtfile: org.apache.spark.rdd.RDD[String] = hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv MapPartitionsRDD[12]
at textFile at <console>:24

scala> val map_test = txtfile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[13] at map at <console>:26

scala> map_test.collect
res11: Array[Array[String]] = Array(Array(Michael Phelps, 23, United States, 2008, 08-24-08, Swimming, 8, 0, 0, 8), Array(Mic
hael Phelps, 19, United States, 2004, 08-29-04, Swimming, 6, 0, 2, 8), Array(Michael Phelps, 27, United States, 2012, 08-12-1
2, Swimming, 4, 2, 0, 6), Array(Natalie Coughlin, 25, United States, 2008, 08-24-08, Swimming, 1, 2, 3, 6), Array(Aleksey Nem
ov, 24, Russia, 2000, 10-01-00, Gymnastics, 2, 1, 3, 6), Array(Alicia Coutts, 24, Australia, 2012, 08-12-12, Swimming, 1, 3,
1, 5), Array(Missy Franklin, 17, United States, 2012, 08-12-12, Swimming, 4, 0, 1, 5), Array(Ryan Lochte, 27, United States,
2012, 08-12-12, Swimming, 2, 2, 1, 5), Array(Allison Schmitt, 22, United States, 2012, 08-12-12, Swimming, 3, 1, 1, 5), Array
(Natalie Coughlin, 21, United States, 2004, 08-...
scala>

scala> val map_test1 = map_test.map(line => (line(1),line(2)))
map_test1: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[14] at map at <console>:28

scala> map_test1.collect
res12: Array[(String, String)] = Array((23,United States), (19,United States), (27,United States), (25,United States), (24,Ru
ssia), (24,Australia), (17,United States), (27,United States), (22,United States), (21,United States), (17,Australia), (33,Un
ited States), (26,Canada), (18,United States), (29,Norway), (20,China), (24,Zimbabwe), (23,Australia), (24,United States), (3
0,Netherlands), (28,Australia), (21,Australia), (27,Netherlands), (25,United States), (23,Australia), (27,Australia), (27,Uni
ted States), (22,Netherlands), (20,South Korea), (17,Russia), (16,United States), (26,Russia), (30,Netherlands), (24,Norway),
 (28,Norway), (20,Croatia), (23,United States), (20,France), (18,Australia), (27,United States), (22,Japan), (28,United State
s), (21,Netherlands), (22,France), (25,Australi...
scala>

scala>
```

You can see that in the above screen shot we have created a new RDD using sc.textFile method and have used the map method to transform the created RDD.

In the first map method i.e., map_test we are splitting the each record by '\t' tab delimiter as the data is tab separated. And you can see the result in the below step.

We have transformed the RDD again by using the map method i.e., map_test1. Here we are creating two columns as a pair. We have used column2 and column3. You can see the result in the below step.

Flatmap

FlatMap will take an iterable data as input and returns the RDD as the contents of the iterator.

```
scala> val txtfile = sc.textFile("hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv")
txtfile: org.apache.spark.rdd.RDD[String] = hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv MapPartitionsRDD[16]
at textFile at <console>:24

scala> val map_test = txtfile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[17] at map at <console>:26

scala> val flatmap_test = map_test.flatMap(line => line)
flatmap_test: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[18] at flatMap at <console>:28

scala> flatmap_test.collect
res13: Array[String] = Array(Michael Phelps, 23, United States, 2008, 08-24-08, Swimming, 8, 0, 0, 8, Michael Phelps, 19, Uni
ted States, 2004, 08-29-04, Swimming, 6, 0, 2, 8, Michael Phelps, 27, United States, 2012, 08-12-12, Swimming, 4, 2, 0, 6, Na
talie Coughlin, 25, United States, 2008, 08-24-08, Swimming, 1, 2, 3, 6, Aleksey Nemov, 24, Russia, 2000, 10-01-00, Gymnastic
s, 2, 1, 3, 6, Alicia Coutts, 24, Australia, 2012, 08-12-12, Swimming, 1, 3, 1, 5, Missy Franklin, 17, United States, 2012, 0
8-12-12, Swimming, 4, 0, 1, 5, Ryan Lochte, 27, United States, 2012, 08-12-12, Swimming, 2, 2, 1, 5, Allison Schmitt, 22, Uni
ted States, 2012, 08-12-12, Swimming, 3, 1, 1, 5, Natalie Coughlin, 21, United States, 2004, 08-29-04, Swimming, 2, 2, 1, 5,
Ian Thorpe, 17, Australia, 2000, 10-01-00, Swim...
scala>

scala>
```

Previously the contents of map_test are iterable. Now after performing flatMap on the data, it is not iterable.

Filter

Filter returns an RDD which meets the filter condition.

```
scala> val txtfile = sc.textFile("hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv")
txtfile: org.apache.spark.rdd.RDD[String] = hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv MapPartitionsRDD[20]
at textFile at <console>:24

scala> val map_test = txtfile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[21] at map at <console>:26

scala> val fil = map_test.filter(line => line(2).contains("India"))
fil: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[22] at filter at <console>:28

scala> fil.collect
res14: Array[Array[String]] = Array(Array(Yogeshwar Dutt, 29, India, 2012, 08-12-12, Wrestling, 0, 0, 1, 1), Array(Sushil Kum
ar, 29, India, 2012, 08-12-12, Wrestling, 0, 1, 0, 1), Array(Sushil Kumar, 25, India, 2008, 08-24-08, Wrestling, 0, 0, 1, 1),
 Array(Karnam Malleswari, 25, India, 2000, 10-01-00, Weightlifting, 0, 0, 1, 1), Array(Vijay Kumar, 26, India, 2012, 08-12-12
, Shooting, 0, 1, 0, 1), Array(Gagan Narang, 29, India, 2012, 08-12-12, Shooting, 0, 0, 1, 1), Array(Abhinav Bindra, 25, Indi
a, 2008, 08-24-08, Shooting, 1, 0, 0, 1), Array(Rajyavardhan Rathore, 34, India, 2004, 08-29-04, Shooting, 0, 1, 0, 1), Array
(M. C. Mary Kom, 29, India, 2012, 08-12-12, Boxing, 0, 0, 1, 1), Array(Vijender Singh, 22, India, 2008, 08-24-08, Boxing, 0,
0, 1, 1), Array(Saina Nehwal, 22, India, 2012, ...
scala>

scala>
```

You can see that all the records of India are present in the output.

ReduceByKey

reduceByKey takes a pair of key and value pairs and combines all the values for each unique key.

```
scala> val txtfile = sc.textFile("hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv")
txtfile: org.apache.spark.rdd.RDD[String] = hdfs://localhost:8020/user/acadgild/hadoop/olympic_data.csv MapPartitionsRDD[24]
at textFile at <console>:24

scala> val map_test = txtfile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[25] at map at <console>:26

scala> val map_test1 = map_test.map(line => (line(2),line(9).toInt))
map_test1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[26] at map at <console>:28

scala> map_test1.reduceByKey(_+_).collect
res15: Array[(String, Int)] = Array((Australia,609), (Brazil,221), (Mexico,38), (Uzbekistan,19), (France,318), (South Korea,3
08), (Finland,118), (Germany,629), (Macedonia,1), (Montenegro,14), (Uruguay,1), (Cuba,188), (Bahrain,1), (North Korea,21), (S
weden,181), (Vietnam,2), (Serbia,31), (Iran,24), (Slovakia,35), (Venezuela,4), (Denmark,89), (Chinese Taipei,20), (Saudi Arab
ia,6), (Paraguay,17), (Serbia and Montenegro,38), (Sudan,1), (Botswana,1), (Greece,59), (Italy,331), (Slovenia,25), (Kuwait,2
), (Iceland,15), (Netherlands,318), (Spain,205), (Hong Kong,3), (Mongolia,10), (Malaysia,3), (Kazakhstan,42), (Ukraine,143),
(Romania,123), (Egypt,8), (Latvia,17), (Eritrea,1), (Indonesia,22), (Armenia,10), (Norway,192), (Thailand,18), (Poland,80), (
Tajikistan,3), (Afghanistan,2), (Israel,4), (Gr...
scala>

scala>
```

Here in this scenario, we have taken a pair of Country and total medals columns as key and value and we are performing reduceByKey operation on the RDD.

We have got the final result as country and the total number of medals won by each country in all the Olympic games.

Actions :

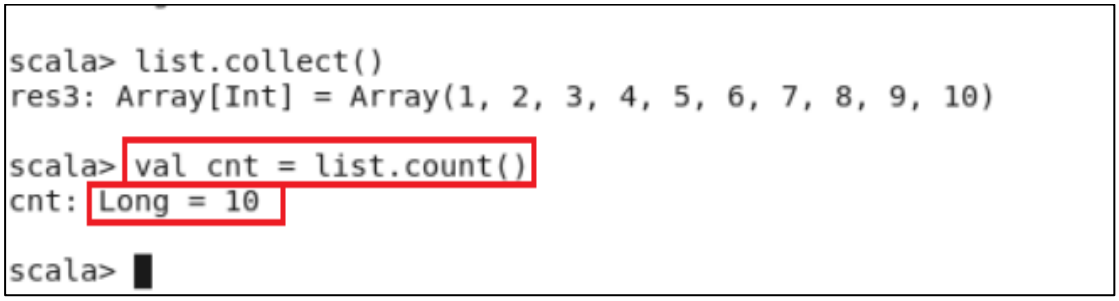
Actions return final results of RDD computations. Actions trigger execution using lineage graph to load the data into original RDD and carry out all intermediate transformations and return the final results to the Driver program or writes it out to the file system.

Collect

Collect is used to return all the elements in the RDD. In all the examples given above, we have used collect to show the contents of the RDD.

Count

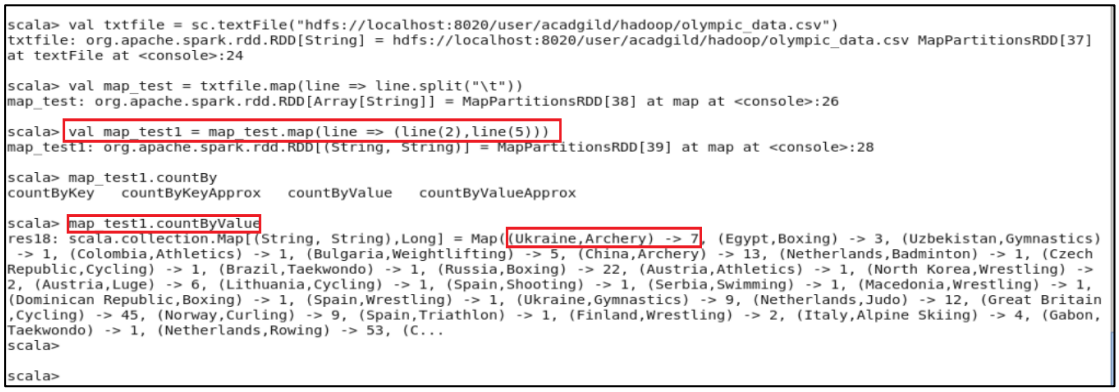
Count is used to return the number of elements in the RDD.



The RDD list contains 10 values hence count returns 10.

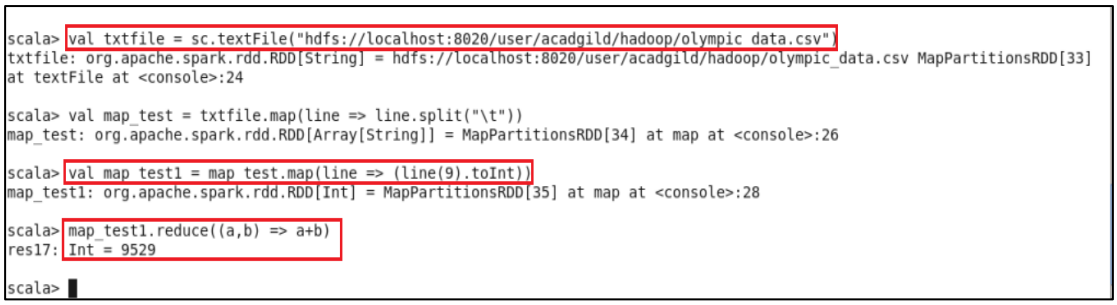
CountByValue

CountByValue is used to count the number of occurrences of the elements in the RDD.



In the above scenario, we have taken a pair of Country and Sport. By performing countByValue action we have got the count of each country in a particular sport.

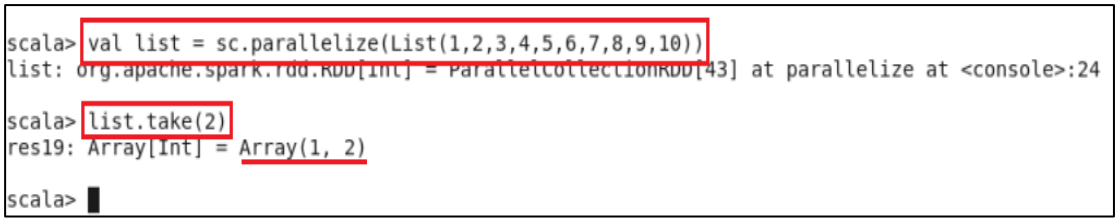
Reduce



On the RDD map_test1 we are performing reduce operation. Finally, we have got that there is a total of 9529 medals declared as the winners in Olympic.

Take

take will display the number of records we explicitly specify.



We have specified take(2), so first 2 values from the RDD list are displayed.