

ASSIGNMENT SCALA 5

Problem Statement

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

EXPLANATION: Here we are creating a list with some elements and then used for loops for doing operations on the elements inside the list

Operations like:

A) Sum of all the Elements inside the List

EXPLANATION: Here used for-loop for calculating the sum of the elements present in the List.

```
CODE : for (i <- Numbers)      //for-loop for getting sum of the elements
inside the list
{
    sum = sum + i
}
```

B) Total number of Elements inside the List

EXPLANATION: Using Length function we can calculate total number of elements present in the List.

```
CODE: println("Total Elements in the List = "+Numbers.length)
```

C) Sum of all the Even numbers inside the List

EXPLANATION: Used for-loop with if condition to filter out non even numbers and does the sum operation on all the even numbers present in the List.

```
CODE: for (n <- Numbers if n%2==0) //for-loop for getting some of even
numbers in the list
{
    sumeven = sumeven + n
}
```

D) Average of the Elements inside the List

EXPLANATION: We can get the average from the List using logic (Sum of all the elements / Total number of elements present in the List)

CODE: var avg : Double = sum / Numbers.length //Data type used here is double to get correct value for the Average

E) Total Number of Elements which are Divisible by 5 & 3

EXPLANATION: Used for-loop with if condition to filter out non Divisible(5 & 3) numbers and gets the Total number of elements present in the List that are Divisible by 5 & 3

CODE: val divisible = for (e <- Numbers if e % 5 == 0 || e % 3 == 0) yield e

SOLUTION REPORT:

```
package scala_assignment
```

```
object Acadgild {
```

```
  def main (arg:Array[String]) {  
    //List With Some Elements
```

```
    val Numbers = List[Int](1,2,3,4,5,6,7,8,9,10)
```

```
    var sum : Double = 0
```

```
    var sumeven = 0
```

```
    for (i <- Numbers)          //for-loop for getting sum of the elements  
inside the list
```

```
    {  
      sum = sum + i  
    }
```

```
    for (n <- Numbers if n%2==0) //for-loop for getting some of even numbers  
in the list
```

```
    {  
      sumeven = sumeven + n  
    }
```

```
//for-loop used to get total elements divisible by 5 & 3 from the list
```

```
val divisible = for (e <- Numbers if e % 5 == 0 || e % 3 == 0) yield  
e
```

```
println("Sum of the List = "+sum)
```

```
println("Total Elements in the List = "+Numbers.length)
```

```
//To get average of the elements inside the list
```

```
var avg : Double = sum / Numbers.length //Data type used here is double  
to get correct value for the Average
```

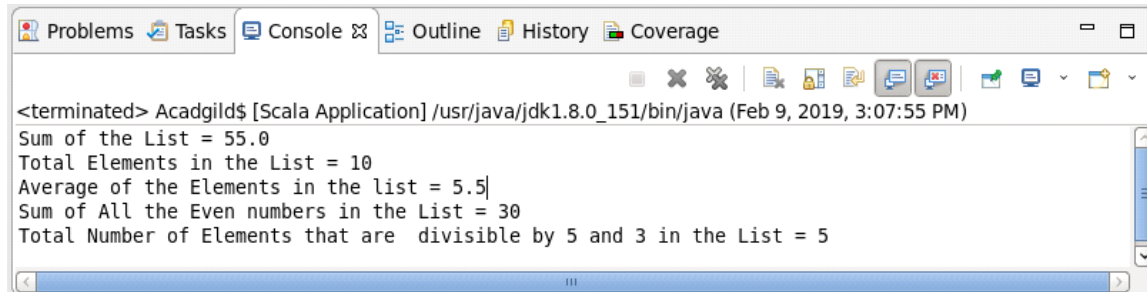
```
println("Average of the Elements in the list = "+avg)
```

```
println("Sum of All the Even numbers in the List = "+sumeven)
```

```

        println("Total Number of Elements that are divisible by 5 and 3 in the
List = "+divisible.length)
    }
}

```



```

<terminated> Acadgild$ [Scala Application] /usr/java/jdk1.8.0_151/bin/java (Feb 9, 2019, 3:07:55 PM)
Sum of the List = 55.0
Total Elements in the List = 10
Average of the Elements in the list = 5.5
Sum of All the Even numbers in the List = 30
Total Number of Elements that are divisible by 5 and 3 in the List = 5

```

Task 2

1) Pen down the limitations of MapReduce.

1. Issue with Small Files

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

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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.

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

RDD (Resilient Distributed Dataset) is the fundamental data structure of Apache Spark which are an immutable collection of objects which computes on the different node of the cluster. Each and every dataset in Spark RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

Features of using RDD are -

1) In-memory computation : The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance by an order of magnitudes. refer this comprehensive guide to Learn Spark in-memory computation in detail.

2) Lazy Evaluation : The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered. Thus, it limits how much work it has to do. Follow this guide to learn Spark lazy evaluation in great detail.

3) Fault Tolerance : Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data. Learn Fault tolerance in Spark in detail.

4) Immutability : RDDs are immutable in nature meaning once we create an RDD we can not manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

5) Persistence : We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling `persist()` or `cache()` function. Follow this guide for the detailed study of RDD persistence in Spark.

6) Partitioning : RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

7) Parallel : Rdd, process the data parallelly over the cluster.

8) Typed : We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

9) No limitation : we can have any number of RDD. there is no limit to its number. the limit depends on the size of disk and memory.

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

Transformations:

Any function that returns an RDD is a transformation. Transformation is functions which create a new data set from an existing one by passing each data set element through a function and returns a new RDD representing the results.

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.

Some transformations like map, flatmap, filter which are commonly used.

Spark rdd Map:

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

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[32] at textFile at <console>:21

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[33] at map at <console>:23

scala> val flatmap_test = map_test.flatMap(line => line)
flatmap_test: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[34] at flatMap at <console>:25

scala> flatmap_test.collect
res15: Array[String] = Array(Michael Phelps, 23, United States, 2008, 08-24-08, Swimming, 8, 0, 0, 8, Michael Phelps, 19, United States, 2004, 08-29-04, Swimming, 6, 0, 2, 8, Michael Phelps, 27, United States, 2012, 08-12-12, Swimming, 4, 2, 0, 6, Natalie Coughlin, 25, United States, 2008, 08-24-08, Swimming, 1, 2, 3, 6, Aleksey Nemov, 24, Russia, 2000, 10-01-00, Gymnastics, 2, 1, 3, 6, Alicia Coutts, 24, Australia, 2012, 08-12-12, Swimming, 1, 3, 1, 5, Missy Franklin, 17, United States, 2012, 08-12-12, Swimming, 4, 0, 1, 5, Ryan Lochte, 27, United States, 2012, 08-12-12, Swimming, 2, 2, 1, 5, Allison Schmitt, 22, United 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>
```

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 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.

Spark rdd Flat map:

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

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[32] at textFile at <console>:21

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[33] at map at <console>:23

scala> val flatmap_test = map_test.flatMap(line => line)
flatmap_test: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[34] at flatMap at <console>:25

scala> flatmap_test.collect
res15: Array[String] = Array(Michael Phelps, 23, United States, 2008, 08-24-08, Swimming, 8, 0, 0, 8, Michael Phelps, 19, United States, 2004, 08-29-04, Swimming, 6, 0, 2, 8, Michael Phelps, 27, United States, 2012, 08-12-12, Swimming, 4, 2, 0, 6, Natalie Coughlin, 25, United States, 2008, 08-24-08, Swimming, 1, 2, 3, 6, Aleksey Nemov, 24, Russia, 2000, 10-01-00, Gymnastics, 2, 1, 3, 6, Alicia Coutts, 24, Australia, 2012, 08-12-12, Swimming, 1, 3, 1, 5, Missy Franklin, 17, United States, 2012, 08-12-12, Swimming, 4, 0, 1, 5, Ryan Lochte, 27, United States, 2012, 08-12-12, Swimming, 2, 2, 1, 5, Allison Schmitt, 22, United 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, Swi...
```

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. Below is the sample demonstration of the above scenario. We can see that all the records of India are present in the output.

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[47] at textFile at <console>:21

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[48] at map at <console>:23

scala> val fil = map_test.filter(line => line(2).contains("India"))
fil: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[49] at filter at <console>:25

scala> fil.collect
res20: Array[Array[String]] = Array(Array(Yogeshwar Dutt, 29, India, 2012, 08-12-12, Wrestling, 0, 0, 1, 1), Array(Sushil Kumar, 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, India, 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, ...)
```

ReduceByKey:

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

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[106] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[107] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(2), line(9).toInt))
map_test1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[108] at map at <console>:26

scala> map_test1.reduceByKey(_+_).collect
res38: Array[(String, Int)] = Array((Australia,609), (Great Britain,322), (Brazil,221), (Canada,370), (Uzbekistan,19), (Barbados,1), (Japan,282), (Cyprus,1), (Finland,118), (Singapore,7), (Montenegro,14), (Uruguay,1), (Moldova,5), (Colombia,13), (Sweden,181), (Vietnam,2), (Serbia,31), (Iran,24), (Slovakia,35), (Mozambique,1), (Cameroon,20), (Denmark,89), (Turkey,28), (Panama,1), (Saudi Arabia,6), (Hungary,145), (Portugal,9), (Paraguay,17), (Jamaica,80), (Georgia,23), (Dominican Republic,5), (Kyrgyzstan,3), (Netherlands,318), (Iceland,15), (Morocco,11), (Belarus,97), (Mongolia,10), (Kazakhstan,42), (Kenya,39), (Syria,1), (Indonesia,22), (Eritrea,1), (Uganda,1), (Norway,192), (Puerto Rico,2), (Poland,80), (Tajikistan,3), (Grenada,1), (Trinidad and Tobago,19), (Afghanistan,2), (Israel,4), ...)
```

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. Refer the below screen shot for the same.

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[64] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[65] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(0),line(2)))
map_test1: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[66] at map at <console>:26

scala> map_test1.collect
res22: Array[(String, String)] = Array((Michael Phelps,United States), (Michael Phelps,United States), (Michael Phelps,United States), (Natalie Coughlin,United States), (Aleksey Nemov,Russia), (Alicia Coutts,Australia), (Missy Franklin,United States), (Ryan Lochte,United States), (Allison Schmitt,United States), (Natalie Coughlin,United States), (Ian Thorpe,Australia), (Dara Torres,United States), (Cindy Klassen,Canada), (Nastia Liukin,United States), (Marit Bjergen,Norway), (Sun Yang,China), (Kirsty Coventry,Zimbabwe), (Libby Lenton-Trickett,Australia), (Ryan Lochte,United States), (Inge de Bruijn,Netherlands), (Petria Thomas,Australia), (Ian Thorpe,Australia), (Inge de Bruijn,Netherlands), (Gary Hall Jr.,United States), (Michael Klin,Australia), (Susie O'Neill,Australia), (Jenny Thomp...
scala>
```

In the above screenshot, we have displayed the Athlete name and his country as two elements from the map method and performed collect action on the newly created rdd as map_test1. The result is displayed on the screen.

Count:

count is used to return the number of elements in the RDD. Below is the sample demonstration of the above scenario.

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[64] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[65] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(0),line(2)))
map_test1: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[66] at map at <console>:26

scala> map_test1.collect
res22: Array[(String, String)] = Array((Michael Phelps,United States), (Michael Phelps,United States), (Michael Phelps,United States), (Natalie Coughlin,United States), (Aleksey Nemov,Russia), (Alicia Coutts,Australia), (Missy Franklin,United States), (Ryan Lochte,United States), (Allison Schmitt,United States), (Natalie Coughlin,United States), (Ian Thorpe,Australia), (Dara Torres,United States), (Cindy Klassen,Canada), (Nastia Liukin,United States), (Marit Bjergen,Norway), (Sun Yang,China), (Kirsty Coventry,Zimbabwe), (Libby Lenton-Trickett,Australia), (Ryan Lochte,United States), (Inge de Bruijn,Netherlands), (Petria Thomas,Australia), (Ian Thorpe,Australia), (Inge de Bruijn,Netherlands), (Gary Hall Jr.,United States), (Michael Klin,Australia), (Susie O'Neill,Australia), (Jenny Thomp...
scala> map_test1.count
res23: Long = 8618

scala>
```

In the above screenshot, you can see that there are 8618 records in the RDD map_test1.

CountByValue:

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


```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[71] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[72] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(2),line(5)))
map_test1: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[73] at map at <console>:26

scala> map_test1.countByValue
res25: scala.collection.Map[(String, String),Long] = Map((Ukraine,Archery) -> 7, (Egypt,Boxing) -> 3, (Uzbekistan,Gymnastics) -> 1, (Colombia,Athletics) -> 1, (Bulgaria,Weightlifting) -> 5, (China,Archery) -> 13, (Netherlands,Badminton) -> 1, (Czech Republic,Cycling) -> 1, (Brazil,Taekwondo) -> 1, (Russia,Boxing) -> 22, (Austria,Athletics) -> 1, (North Korea,Wrestling) -> 2, (Austria,Luge) -> 6, (Lithuania,Cycling) -> 1, (Spain,Shooting) -> 1, (Serbia,Swimming) -> 1, (Macedonia,Wrestling) -> 1, (Dominican Republic,Boxing) -> 1, (Spain,Wrestling) -> 1, (Ukraine,Gymnastics) -> 9, (Netherlands,Judo) -> 12, (Great Britain,Cycling) -> 45, (Norway,Curling) -> 9, (Spain,Triathlon) -> 1, (Finland,Wrestling) -> 2, (Italy,Alpine Skiing) -> 4, (Gabon,Taekwondo) -> 1, (Netherlands,Rowing) -> 53, (C...
scala>
```

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:

Below is the sample demonstration of the above scenario where we have taken the total number of medals column in the dataset and loaded into the RDD map_test1. On this RDD we are performing reduce operation. Finally, we have got that there is a total of 9529 medals declared as the winners in Olympic.

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[93] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[94] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(9).toInt))
map_test1: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[95] at map at <console>:26

scala> map_test1.reduce((a,b)=>a+b)
res34: Int = 9529

scala>
```

Take:

take will display the number of records we explicitly specify. Here in the above screenshot, you can see that when we performed collect operation, it displayed all the elements of the RDD. But when we perform take operation we can limit the number of elements getting displayed by explicitly passing some integer value as an argument.

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/olympix_data.csv")
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[106] at textFile at <console>:22

scala> val map_test = textFile.map(line => line.split("\t"))
map_test: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[107] at map at <console>:24

scala> val map_test1 = map_test.map(line => (line(2),line(9).toInt))
map_test1: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[108] at map at <console>:26

scala> map_test1.reduceByKey(_+_).collect
res38: Array[(String, Int)] = Array((Australia,609), (Great Britain,322), (Brazil,221), (Canada,370), (Uzbekistan,19), (Barbados,1), (Japan,282), (Cyprus,1), (Finland,118), (Singapore,7), (Montenegro,14), (Uruguay,1), (Moldova,5), (Colombia,13), (Sweden,181), (Vietnam,2), (Serbia,31), (Iran,24), (Slovakia,35), (Mozambique,1), (Cameroon,20), (Denmark,89), (Turkey,28), (Panama,1), (Saudi Arabia,6), (Hungary,145), (Portugal,9), (Paraguay,17), (Jamaica,80), (Georgia,23), (Dominican Republic,5), (Kyrgyzstan,3), (Netherlands,318), (Iceland,15), (Morocco,11), (Belarus,97), (Mongolia,10), (Kazakhstan,42), (Kenya,39), (Syria,1), (Indonesia,22), (Eritrea,1), (Uganda,1), (Norway,192), (Puerto Rico,2), (Poland,80), (Tajikistan,3), (Grenada,1), (Trinidad and Tobago,19), (Afghanistan,2), (Israel,4), ...)
scala>

scala> map_test1.reduceByKey(_+_).take(2)
res39: Array[(String, Int)] = Array((Australia,609), (Great Britain,322))

scala>
```