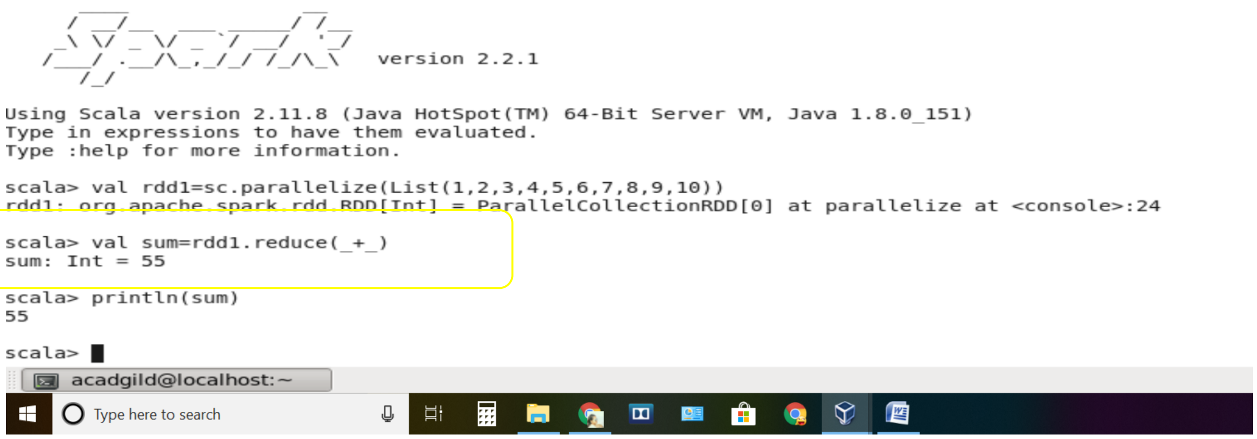
Session 18: INTRODUCTION TO SPARK

Assignment 1

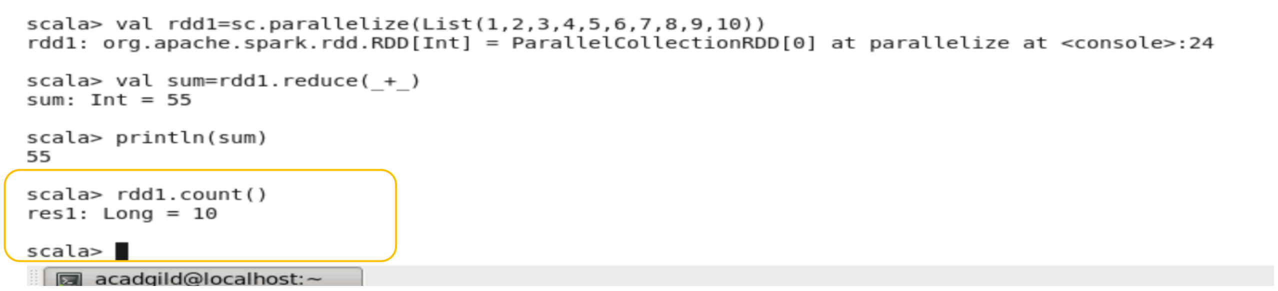
Task 1
Given a list of numbers - List[Int] (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

- find the sum of all numbers



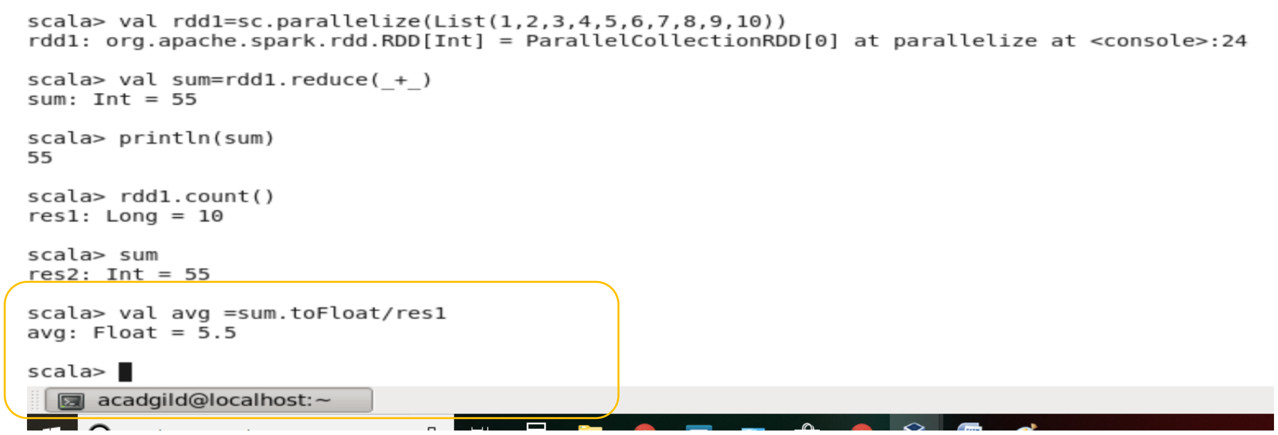
```
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_151)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala> █
```

- find the total elements in the list



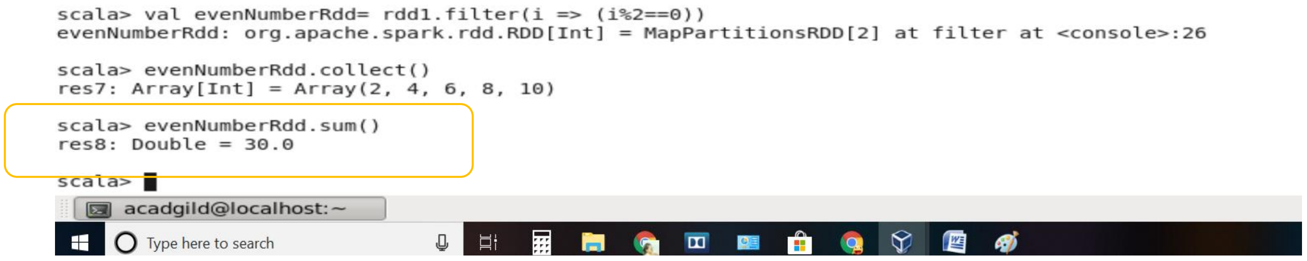
```
scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala> rdd1.count()
res1: Long = 10

scala> █
```

- calculate the average of the numbers in the list
  Sum is variable and res1 is count of elements



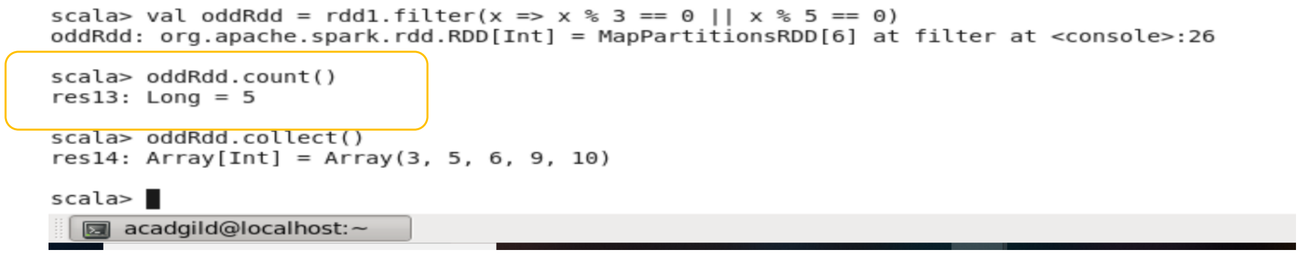
```
scala> val rdd1=sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val sum=rdd1.reduce(_+_)
sum: Int = 55

scala> println(sum)
55

scala> rdd1.count()
res1: Long = 10

scala> sum
res2: Int = 55

scala> val avg =sum.toFloat/res1
avg: Float = 5.5

scala> █
```

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



```
scala> val evenNumberRdd= rdd1.filter(i => (i%2==0))
evenNumberRdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at filter at <console>:26

scala> evenNumberRdd.collect()
res7: Array[Int] = Array(2, 4, 6, 8, 10)

scala> evenNumberRdd.sum()
res8: Double = 30.0

scala> 
```

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



```
scala> val oddRdd = rdd1.filter(x => x % 3 == 0 || x % 5 == 0)
oddRdd: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[6] at filter at <console>:26

scala> oddRdd.count()
res13: Long = 5

scala> oddRdd.collect()
res14: Array[Int] = Array(3, 5, 6, 9, 10)

scala> 
```

Task 2

1) *Pen down the limitations of MapReduce.*

Map Reduce was developed just to handle Batch processing

MapReduce cannot handle:

- Interactive Processing
- Real-time (stream) Processing
- Iterative (delta) Processing
- In-memory Processing
- Graph Processing

I. Issue with Small Files
- Hadoop is not suited for small data. HDFS lacks the ability to efficiently support the random reading of small files because of its high capacity design.

II. Slow Processing Speed

- Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

III. Support for Batch Processing only

- Hadoop supports batch processing only, it does not process streamed data, and hence overall performance is slower.

IV. No Real-time Data Processing

- Hadoop is not suitable for Real-time data processing.

V. 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).

VI. Latency

- 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.

VII. 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 stands Resilient Distributed Dataset. RDDs are the fundamental abstraction of Apache Spark. It is an immutable distributed collection of the dataset. Each dataset in RDD is divided into logical partitions. On the different node of the cluster, we can compute these partitions. RDDs are a read-only partitioned collection of record. We can create RDD in three ways:

1. Parallelizing already existing collection in driver program.
2. Referencing a dataset in an external storage system (e.g. HDFS, Hbase, shared file system).
3. Creating RDD from already existing RDDs

Few features of RDD

a. *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.

b. *Lazy Evaluation*- The changes or the computation is performed only after an action is triggered

c. *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.

d. *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**.**

e. *Partitioning*-RDD partitions 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.

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

Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

RDD Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation.

<u>Transformations are lazy</u> in nature i.e., they get execute when we call an action.

There are two types of transformations:

Narrow transformation **–** ll the elements that are required to compute the records in single partition live in the single partition of parent RDD. *Narrow transformations* are the result of *map(), filter(),flatmap(),union().*

Wide transformation **–** In wide transformation, all the elements that are required to compute the records in the single partition may live in many partitions of parent RDD. *Wide transformations* are the result of *groupbyKey()* and *reducebyKey()*,join(),intersection()

RDD Action

An action is one of the ways of sending data from *Executer* to the *driver.* Executors are agents that are responsible for executing a task. While the driver is a JVM process that coordinates workers and execution of the task. Some of the actions of Spark are**:**

- Count() : returns the number of elements in RDD.
- Collect() is the common and simplest operation that returns our entire RDDs content to driver program.
- take(n) returns n number of elements from RDD.
- reduce() function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements**.**