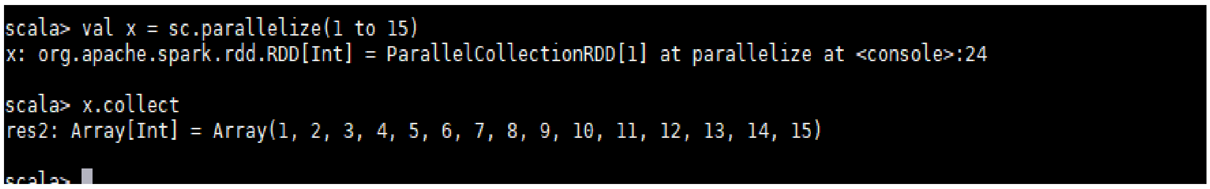
Assignment Submission- Session 18

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

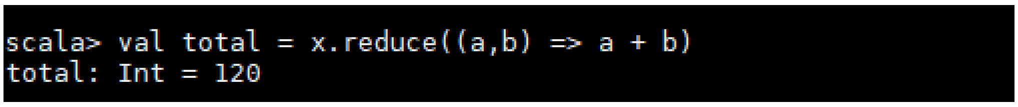
Solution:

Created list using below command:



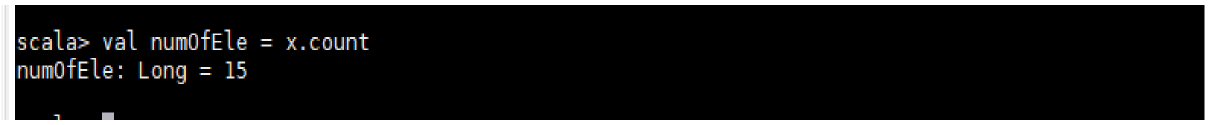
```
scala> val x = sc.parallelize(1 to 15)
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24

scala> x.collect
res2: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)

scala>
```

- find the sum of all numbers



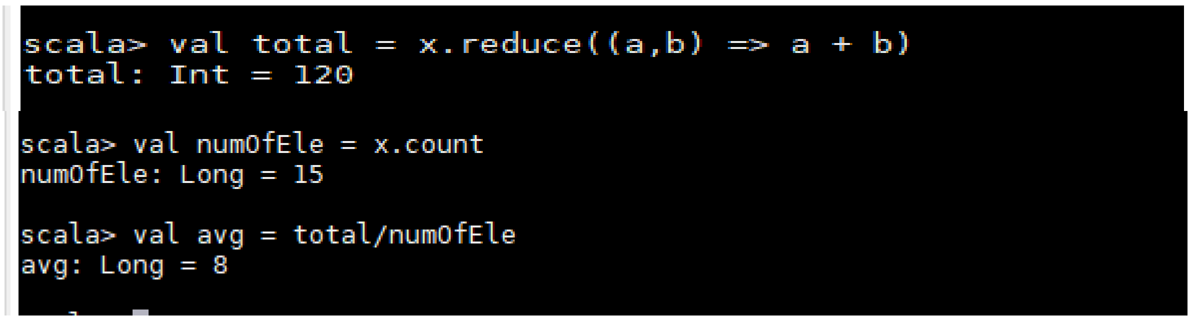
```
scala> val total = x.reduce((a,b) => a + b)
total: Int = 120
```

- find the total elements in the list



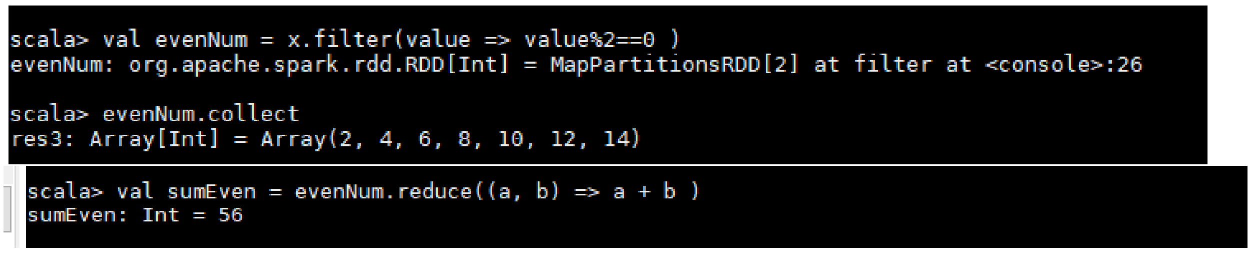
```
scala> val numOfEle = x.count
numOfEle: Long = 15
```

- calculate the average of the numbers in the list



```
scala> val total = x.reduce((a,b) => a + b)
total: Int = 120

scala> val numOfEle = x.count
numOfEle: Long = 15

scala> val avg = total/numOfEle
avg: Long = 8
```

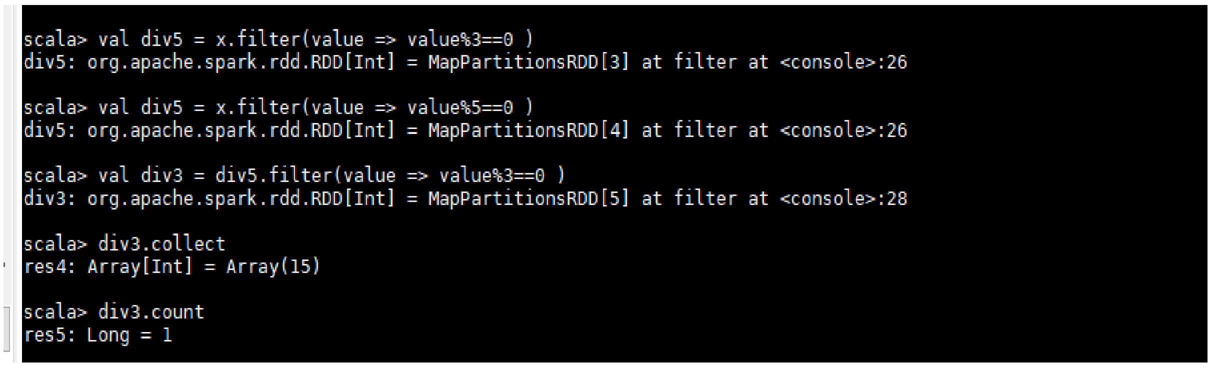
- find the sum of all the even numbers in the list



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

scala> evenNum.collect
res3: Array[Int] = Array(2, 4, 6, 8, 10, 12, 14)
```

```
scala> val sumEven = evenNum.reduce((a, b) => a + b )
sumEven: Int = 56
```

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



```
scala> val div5 = x.filter(value => value%3==0 )
div5: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at filter at <console>:26

scala> val div5 = x.filter(value => value%5==0 )
div5: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at filter at <console>:26

scala> val div3 = div5.filter(value => value%3==0 )
div3: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[5] at filter at <console>:28

scala> div3.collect
res4: Array[Int] = Array(15)

scala> div3.count
res5: Long = 1
```

Task 2

1) Pen down the limitations of MapReduce.

Answer:

Limitations are as follows:

- When we need to perform multiple operations on the data, we would need to write multiple jobs for each process and equivalent number of parses through the data for each process, where as we can program our operations in spark under 1 entity.
- Map Reduce writes the mapper output on the disk, which is picked by the shuffle and sort phase. So, it adds up processing time and effects performance, while spark does in-memory processing and performs way faster than MR.
- MR is not suitable for the requirements that require iterative processing like as done in Machine Learning model building which spark supports.

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

Answer: RDD stands for “Resilient Distributed Datasets”. An RDD in Spark is an automatic immutable distributed collection of objects. Each RDD is split into multiple partitions. RDD are sparks way of understanding data. They are Spark API Scala objects which maintains references of all the partition Scala objects. The partition Scala objects are the reference to the actual data residing on HDFS, Local or created on the fly.

Features of RDD:

- 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.

- 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.

- 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.

- Immutable: RDDs are immutable in nature meaning once we create an RDD we cannot manipulate it. And if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

- Persistence: We can store the frequently used RDD in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. This happens by storing the data explicitly in memory by calling persist() or cache() function.

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

Answer:

There are following two types of operations that can be done on RDDs

Transformations:

- It either loads the data or constructs a new RDD from the previous one
- It does not give you any return value, instead it gives RDD object that is created after executing the command.

Actions:

- Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS)

Basic RDD operations :

1. Loading data on the fly using parallelize:

We can load data on the fly using spark context object and parallelize:

```
val name = sc.parallelize(List("spark example", "sample example 2"))
```

2. Loading data from local/hdfs using textFile:

```
val textFileImport = sc.textFile("/Users/Akshat /test.txt");
```

3. Getting result using collect. (Action RDD)

After loading data using a transformation RDD, we can get result using action RDD.

```
val name = sc.parallelize(List("spark example", "sample example 2"))
```

- name.collect

- name.count

4. Store the frequently used RDD in-memory

```
val name = sc.parallelize(List("spark example", "sample example 2"))
name.persist(org.apache.spark.storage.StorageLevel.MEMORY_AND_DISK)
```

5. We can join 2 RDDs as well.

```
val salesprofit = sc.parallelize(Array(("Cadbury's", 3.5),("Nestle", 2.8),("Mars", 2.5),
("Thorton's", 2.2)))

val salesyear = sc.parallelize(Array(("Cadbury's", 2015),("Nestle", 2014),("Mars", 2014),
("Thorton's", 2013)))

val join = salesprofit.join(salesyear)
```