Assignment 18.1
Introduction to Spark

<u>Task 1</u>

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

- find the sum of all numbers

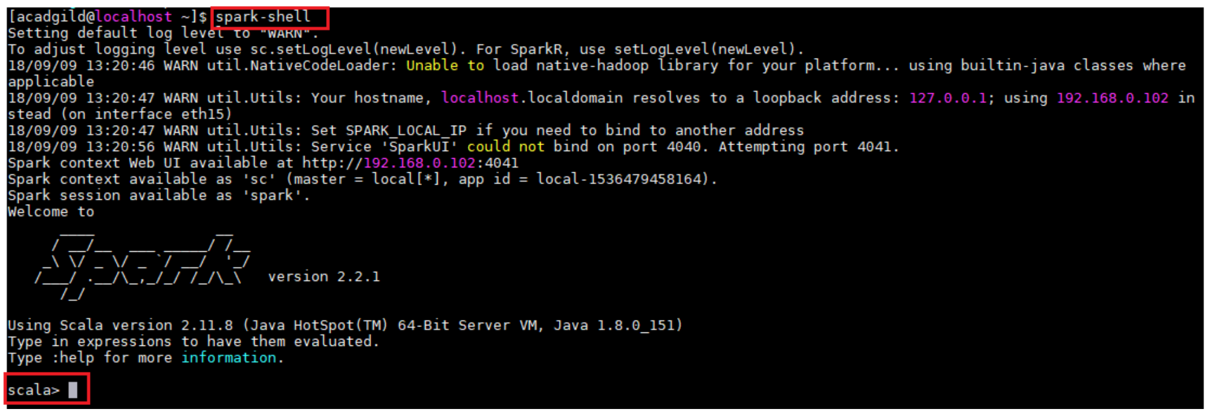
- find the total elements in the list

- 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

We have started spark shell by using command : **spark-shell**



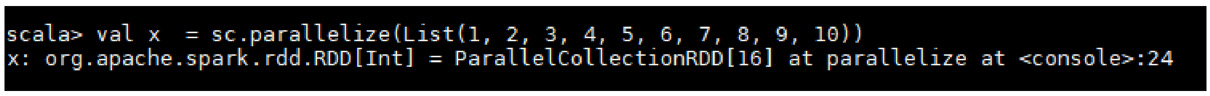
```
[acadgild@localhost ~]$ spark-shell
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/09/09 13:20:46 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where
applicable
18/09/09 13:20:47 WARN util.Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 192.168.0.102 in
stead (on interface eth15)
18/09/09 13:20:47 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
18/09/09 13:20:56 WARN util.Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
Spark context Web UI available at http://192.168.0.102:4041
Spark context available as 'sc' (master = local[*], app id = local-1536479458164).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   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>
```

First we have assigned List of Integers to val x by using parallelize as shown below:

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

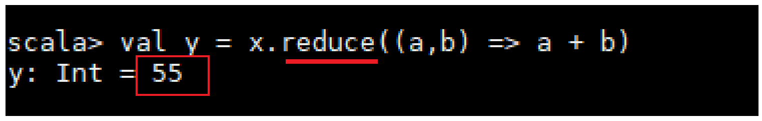


```
scala> val x  = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[16] at parallelize at <console>:24
```

1. **find the sum of all numbers**

Then to find the sum of all numbers, we have used **reduce** action:

val y = x.reduce((a,b) => a + b)



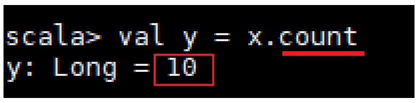
```
scala> val y = x.reduce((a,b) => a + b)
y: Int = 55
```

Here you could see that result is 55 which is summation of all Integers from 1 to 10.

Assignment 18.1
Introduction to Spark

2. **find the total elements in the list**
   To find the total elements in the list, we have used **count** action:

   val y = x.count

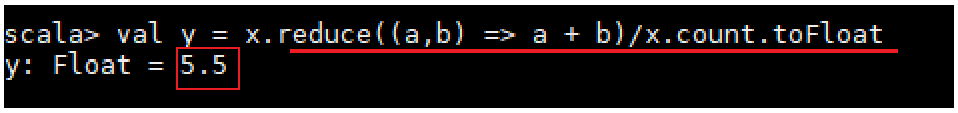


```
scala> val y = x.count
y: Long = 10
```

   Here, you could see that result is 10 as total elements in List are **10**.

3. **calculate the average of the numbers in the list :**
   Here we have taken sum of all elements first by using reduce action and then we have taken count of these elements by using count action.
   Then we have taken average by dividing sum by count value and converted it into Float.

   val y = x.reduce((a,b) => a + b)/x.count.toFloat



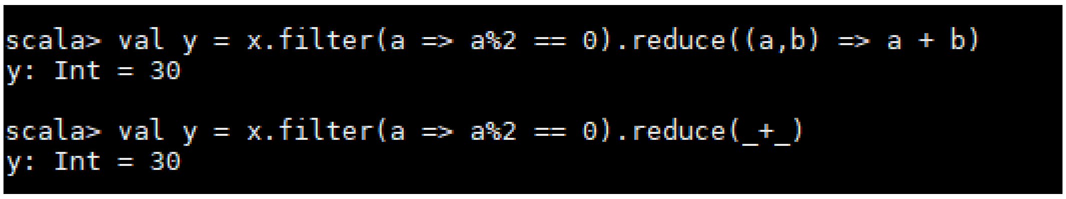
```
scala> val y = x.reduce((a,b) => a + b)/x.count.toFloat
y: Float = 5.5
```

   Here, you could see that result is **5.5** as division of sum with count is **55/10 = 5.5**

4. **find the sum of all the even numbers in the list :**
   Here we have filtered all even numbers first by using **filter** transformation and then we have taken sum of these elements by using **reduce** action.

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

   val y = x.filter(a => a%2 == 0).reduce(_+_)



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

scala> val y = x.filter(a => a%2 == 0).reduce(_+_)
y: Int = 30
```

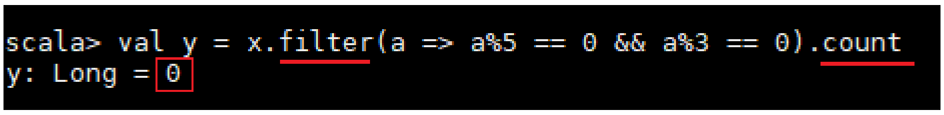
   We could see that result is **30** as sum of even numbers is 2 + 4 + 6 + 8 + 10 = **30.**

Assignment 18.1
Introduction to Spark

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

Here we have filtered all numbers which are divisible by both 5 and 3 by using filter transformation and then we have taken count of these elements by using count action.

val y = x.filter(a => a%5 == 0 && a%3 == 0).count



```
scala> val y = x.filter(a => a%5 == 0 && a%3 == 0).count
y: Long = 0
```

As we could confirm that between 1 to 10 integers, there is no number which could be divisible by both 5 and 3, so number of elements is **0.**

Task 2

1) Pen down the limitations of MapReduce.

1. MapReduce requires a lot of time to perform map and reduce tasks thereby increases the execution time (and latency) and reduces processing speed.
2. MapReduce only supports batch processing, but it does not process streamed data, and hence overall performance is slower.
3. MapReduce does not supports Real time data processing.
4. MapReduce is not so suitable for iterative processing, as Hadoop does not support cyclic data flow.
5. MapReduce does not supports caching. In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which diminishes the performance of Hadoop.
6. MapReduce has lengthy line of code. More number of lines produces, more number of bugs and it takes more time to execute the program.
7. MapReduce is less secured as it lacks encryption.

Assignment 18.1
Introduction to Spark

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

- **RDD** stands for **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.

- 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).
3. Creating RDD from already existing RDDs.

- There are two operations in RDD :

1. **Transformation**
2. **Action**.

Features of RDD :

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 to a great extent.

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.

3. Fault Tolerance
Upon the failure of worker node, we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data.

4. Immutability
RDDS are immutable in nature meaning once we create an RDD we cannot manipulate it. If we perform any transformation over it, it creates a 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.

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.

Assignment 18.1
Introduction to Spark

7. Parallel
RDD process the data in parallel over the cluster.

8. 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.

9. Coarse-grained Operation
We apply coarse-grained transformations to RDD. Coarse-grained meaning the operation applies to the whole dataset not on an individual element in the dataset of RDD.

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

11. No limitation
We can have any number of RDD. There is no limit to its number as such. The limit depends on the size of disk and memory.

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

There are two operations in RDD :

1. **Transformation** :
   It is a function that produces new RDD from the existing RDD.
   Two most basic type of transformations are map() and filter().

   After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).

2. **Action** :
   When we want to work with the actual dataset, at that point Action is performed.

   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.

Assignment 18.1
Introduction to Spark

Let’s see some of these operations:

1. **Map :**

The map function iterates over every line in RDD and split into new RDD.
Using map() transformation we take in any function, and that function is applied to every element of RDD.

In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For example, in RDD {1, 2, 3, 4, 5} if we apply “rdd.map(x=>x+2)” we will get the result as (3, 4, 5, 6, 7).

2. **Filter :**

Filter() function returns a new RDD, containing only elements that meet a predicate. It is a narrow operation because it does not shuffle data from one partition to many partitions.

For example, Suppose RDD contains first five natural numbers (1, 2, 3, 4, and 5) and the predicate is check for an even number. The resulting RDD after the filter will contain only the even numbers i.e., 2 and 4.

3. **Union :**

With the union() function, we get the elements of both RDDs in new RDD. The key rule of this function is that the two RDDs should be of the same type.

For example, the elements of RDD1 are (Spark, Spark, Hadoop, Flink) and that of RDD2 are (Big data, Spark, Flink) so the resultant rdd1.union(rdd2) will have elements (Spark, Spark, Spark, Hadoop, Flink, Flink, Big data).

4. **Count :**

Action count() returns the number of elements in RDD.

For example, RDD has values {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “rdd.count()” will give the result 8.

5. **Collect :**

The action collect() is the common and simplest operation that returns our entire RDDs content to driver program. The application of collect() is unit testing where the entire RDD is expected to fit in memory. As a result, it makes easy to compare the result of RDD with the expected result.