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

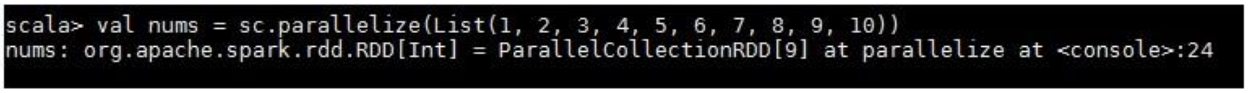
- find the sum of all the even numbers in the list

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

Task1 - Find the sum of all numbers

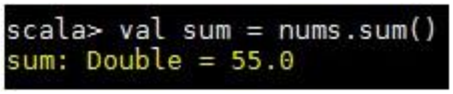
RDD,

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



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

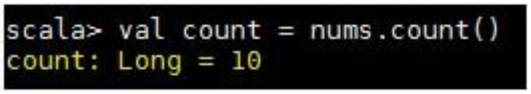
val sum=nums.sum()



```
scala> val sum = nums.sum()
sum: Double = 55.0
```

Task2 - find the total elements in the list

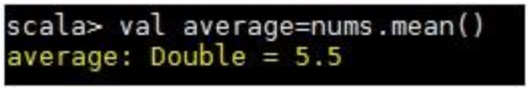
val count = nums.count()



```
scala> val count = nums.count()
count: Long = 10
```

Task3 - calculate the average of the numbers in the list

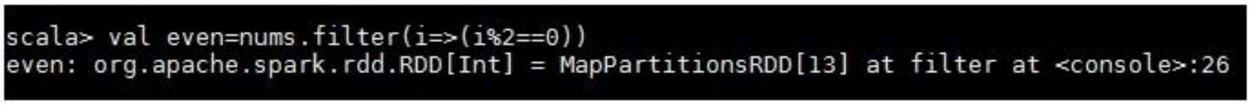
val average=nums.mean()



```
scala> val average=nums.mean()
average: Double = 5.5
```

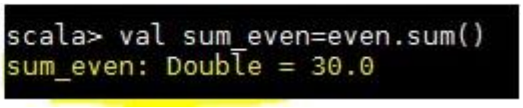
Task4 - find the sum of all the even numbers in the list

val even=nums.filter(i=>(i%2==0))



```
scala> val even=nums.filter(i=>(i%2==0))
even: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[13] at filter at <console>:26
```

val sum_even=even.sum()

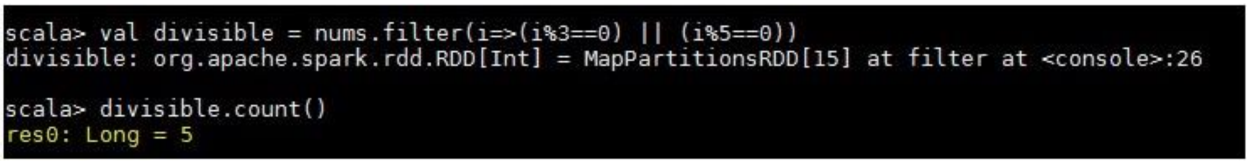


```
scala> val sum_even=even.sum()
sum_even: Double = 30.0
```

Task5 - find the total number of elements in the list divisible by both 5 and 3.

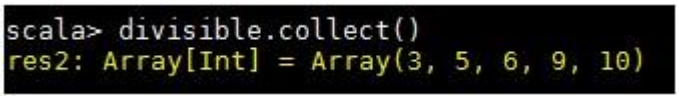
val divisible = nums.filter(i=>(i%3==0) || (i%5==0))

divisible.count()



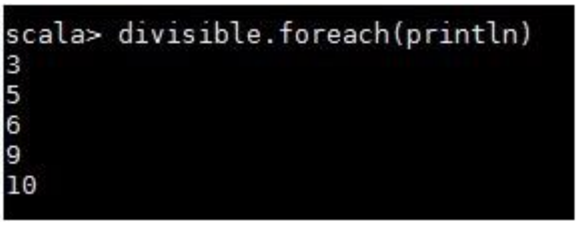
```
scala> val divisible = nums.filter(i=>(i%3==0) || (i%5==0))
divisible: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[15] at filter at <console>:26

scala> divisible.count()
res0: Long = 5
```

divisible.collect()



```
scala> divisible.collect()
res2: Array[Int] = Array(3, 5, 6, 9, 10)
```

divisible.foreach(println)



```
scala> divisible.foreach(println)
3
5
6
9
10
```

Task 2

1) Pen down the limitations of MapReduce.

MapReduce is a programming model and an associated implementation for processing and generating big
data sets with a parallel, distributed algorithm on a cluster.

It’s based on disk computing

- Suitable for single pass computations - not iterative computations.
- Needs a sequence of MR jobs to run iterative tasks,
- Needs integration with several other frameworks/tools to solve bigdata use cases,
  - Apache Storm for stream data processing
  - Apache Mahout for machine learning
- Hadoop Map Reduce 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.
- Slow Processing Speed,
- No Real-time Data Processing
- Lengthy Line of Code and

- MapReduce only ensures that data job is complete, but it’s unable to guarantee when the job willbe complete.

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

- RDD stands for **Resilient Distributed Datasets** are Apache Spark's data abstraction, RDD is a logical
- reference of a dataset which is partitioned across many server machines in the cluster. RDDs are
- **Immutable** and are self-recovered in case of failure. Dataset could be the data loaded externally by the
- user. RDDs can only be created by reading data from a stable storage such as HDFS or by transformations
- on existing RDDs.

Why RDD?

- When it comes to iterative distributed computing, i.e. processing data over multiple jobs in computations
- such as Logistic Regression, K-means clustering, and Page rank algorithms, it is fairly common to reuse or
- share the data among multiple jobs or you may want to do multiple ad-hoc queries over a shared data
- set.

Few 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 by an order of magnitudes.

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.

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

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

4) Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input
5) and produces one or more RDD as output. Each time it creates new RDD when we apply any
6) transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.
7) Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD
8) lineage, also known as **RDD operator graph** or **RDD dependency graph**. It is a logical execution plan i.e.,
9) it is **Directed Acyclic Graph (DAG)** of the entire parent RDDs of RDD.
10) Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed
11) immediately. Two most basic type of transformations is a **map(), filter().**
12) After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g.
13) filter, count, distinct, sample), bigger (e.g. **flatMap(), union(), Cartesian()**) or the same size (e.g. map).
14) There are two types of transformations:
15) ***Narrow transformation*** – In Narrow transformation, all the elements that are required to compute the
16) records in single partition live in the single partition of parent RDD. A limited subset of partition is used to
17) calculate the result. Narrow transformations are the result of **map(), filter().**
18) ***Wide transformation*** – In wide transformation, all the elements that are required to compute the records
19) in the single partition may live in many partitions of parent RDD. The partition may live in many partitions
20) of parent RDD. Wide transformations are the result of **groupbyKey() and reducebyKey().**

21)RDD Action

22) Transformations **create RDDs** from each other, but when we want to work with the actual dataset, at that
23) point action is performed. When the action is triggered after the result, new RDD is not formed like
24) transformation. Thus, Actions are Spark RDD operations that give **non-RDD** values. The values of action
25) are stored to drivers or to the external storage system. It brings laziness of RDD into motion.
26) An action is one of the ways of sending data from Executer to the driver. Executors are agents that are
27) responsible for executing a task. While the driver is a JVM process that coordinates workers and execution
28) of the task. Some of the actions of Spark are:
29) ***count(), collect(), take(n), top(),ountByValue(),reduce(),fold(),aggregate() and foreach().***