TASK 1

Creating num RDD:

```
scala> val num = sc.parallelize(List(1,2,3,4))
num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:27
```

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

```
scala> num.sum
res6: Double = 10.0
```

2. **Total elements in a list**

```
scala> num.count()
res11: Long = 4
```

3. **Average of numbers in list**

```
scala> num.mean()
res15: Double = 2.5
```

4. **Find the sum of all the even numbers in the list**

```
scala> val sum_even = num.filter(x => (x%2==0)).sum
sum_even: Double = 6.0
```

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

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

TASK 2

1. Limitations of Map reduce

- **M/R is based on disk based computing, which involves lot of I/O operation which makes hadoop slow.**

When we submit a job and map reduce task is invoked, the mapper output which are intermediate Key-Value pairs stored in disk. And while processing again this mapper output is loaded into memory. This process creates lot of I/O operation and which is costlier.

- **Map reduce is most suitable only for Batch processing and not for real time processing.**

  Hadoop is meant only for Batch processing. Batch processing means that we will be collecting data over a period of time and perform processing and analytics based on this data (Processing on Historical/Archival data).

- **Iterative computation is not possible in Hadoop**

Suppose my requirement is First filter data with city == ‘ Delhi’ ---------- > filter ------ > male / female ---------- > filter Aadhar no.

Here this iterative computation so not possible in a single Hive query

Example-2 - > Pig.

A = load;

B = filter ‘Delhi’

C = filter M/F

D= filter aadhar.

It is possible in Pig but again Pig uses M/R framework for every script. Which is very slow. i.e Hadoop needs a sequence of MR jobs to run iterative tasks.

- **Hadoop needs integration with several other frameworks (Hive, Pig, Sqoop……..) to solve Bigdata problems. These are not available in a single platform.**

2. What are RDD's? What are the features of RDD?

RDD's are the primary way of creating datasets in Spark.

Features:

- **RDDs are immutable**

- **In – memory computation**

Spark RDDs have a provision of in-memory computation. It stores intermediate results in distributed memory (RAM) instead of stable storage (disk)

- **We can perform two types of operation on RDD:**

Transformation: Which creates a new dataset from previous RDD

Action: Which return a value to the driver after performing the computation on datasets

- **Immutable**
  - Immutability of RDDs makes data secure to share across different processes.
  - Since it is immutable it can be created anytime using lineage graph. Hence the name RESILIENT.

- **Resilient – reliable**

RDD keeps track of transformation performed on datasets. This is called Lineage graph (DAG). So when ever the node fails the RDD partition can be rebuild using this Lineage graph. This is how Fault tolerance is achieved in Spark.

- **Lazy evolutions**

All transformations in Apache Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base data set.

Spark computes transformations when an action requires a result for the driver program

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

Consider the data set – Array(1,2,3,4)

1. Map:

The map transformation takes in a function and applies it to each element in the RDD with the result of the function being the new set values of each element in resultant RDD.

```
scala> val nums = sc.parallelize(Array(1, 2, 3, 4))
nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[5] at parallelize at <console>:27

scala>

scala> val squared = nums.map(x => x * x).collect()
squared: Array[Int] = Array(1, 4, 9, 16)
```

2. Take(n):

Return an array with the first *n* elements of the dataset.

```
scala> var baseRDD = sc.textFile("/home/acadgild/Desktop/Spark_Learn/Dataset/worldcup_data.tsv");
baseRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[12] at textFile at <console>:27

scala> baseRDD.count()
res9: Long = 736
```

3. Coalesce

Coalesce(*numPartitions*) : Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

In the above filterRDD no of Partitions = 8.

But if we want to decrease partitions then use Coalesce

```
scala> filterRDD.coalesce(5).getNumPartitions
res1: Int = 5
```

Creating num RDD:

```
scala> val num = sc.parallelize(List(1,2,3,4))
num: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[8] at parallelize at <console>:27
```

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

```
scala> num.sum
res6: Double = 10.0
```

2. **Total elements in a list**

```
scala> num.count()
res11: Long = 4
```