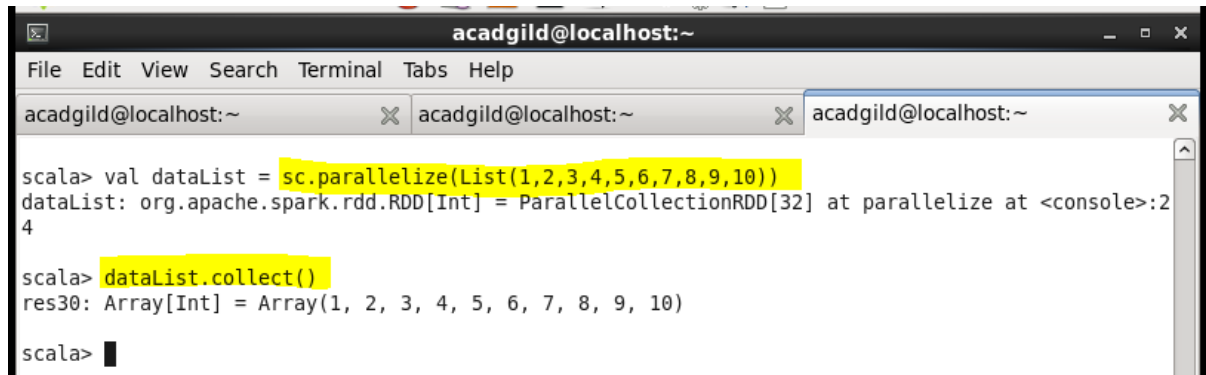


# Assignment-18

## Task 1:

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

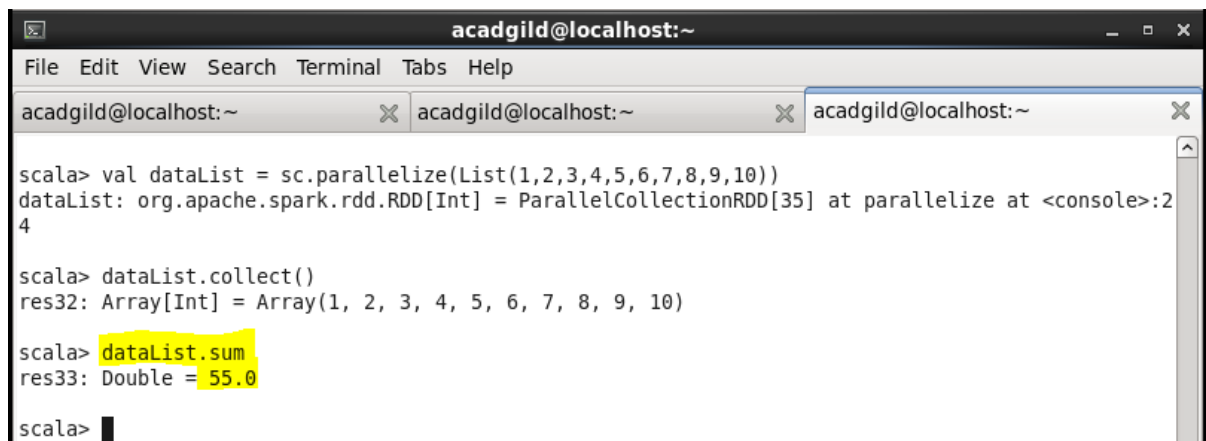
Given list created in spark with RDD(Int) as shown in the below screenshot.



```
acadgild@localhost:~  
File Edit View Search Terminal Tabs Help  
acadgild@localhost:~ acadgild@localhost:~ acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[32] at parallelize at <console>:2  
4  
scala> dataList.collect()  
res30: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
scala> █
```

- Find the sum of all numbers

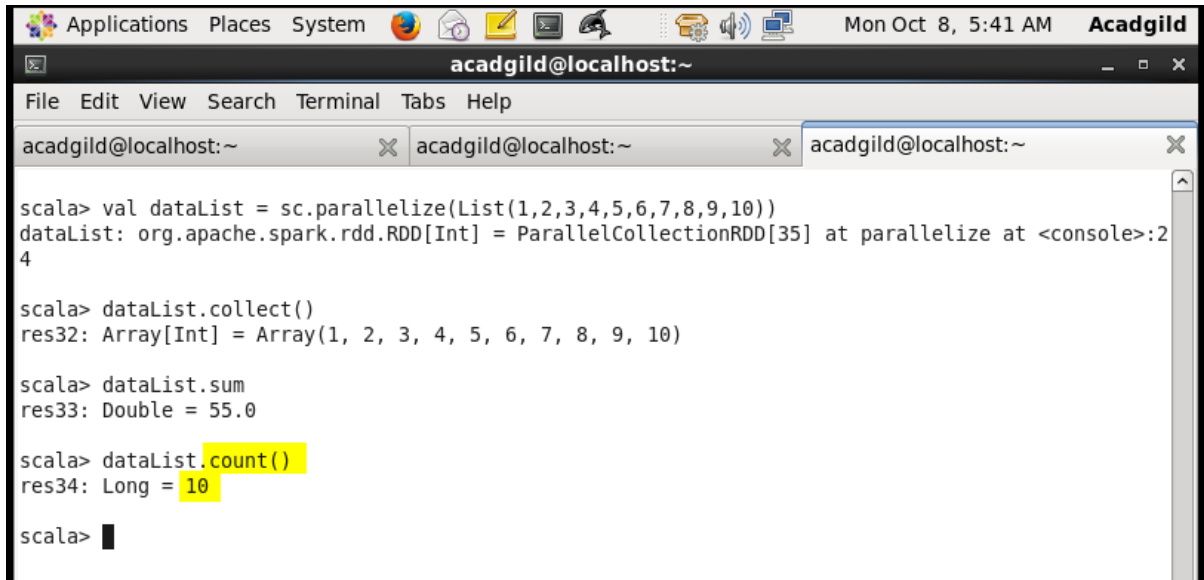
The sum of all the numbers present in the list is found using the command “**dataList.count**” as shown in the screenshot.



```
acadgild@localhost:~  
File Edit View Search Terminal Tabs Help  
acadgild@localhost:~ acadgild@localhost:~ acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:2  
4  
scala> dataList.collect()  
res32: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
scala> dataList.sum  
res33: Double = 55.0  
scala> █
```

➤ Find the total elements in the list

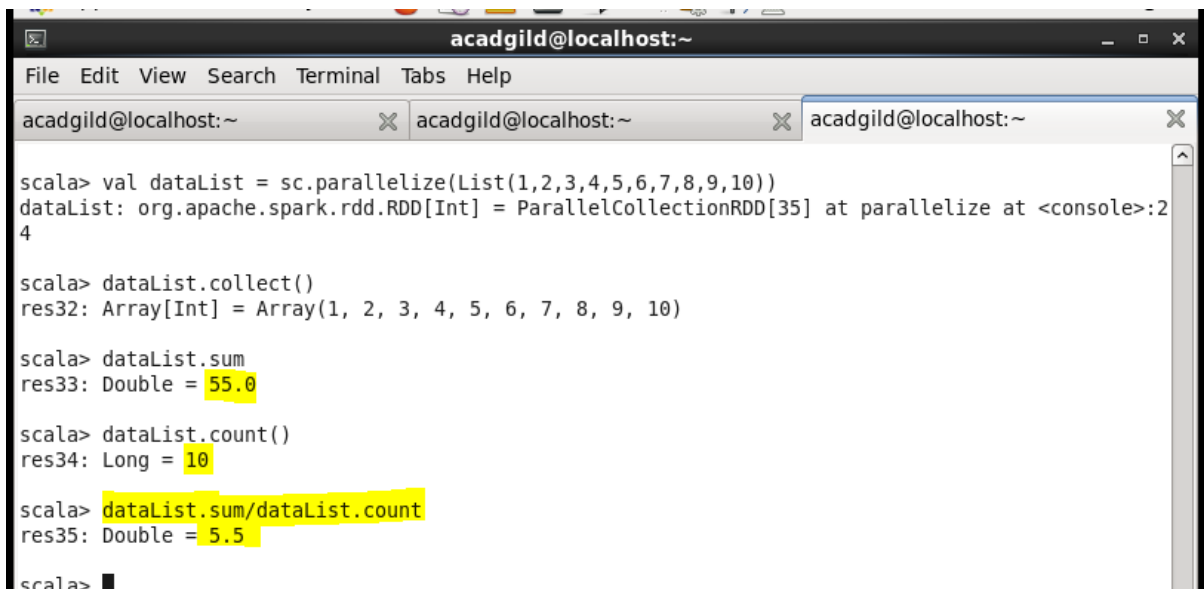
The total elements in the list is found using the command (**dataList.count()**) as shown in the screenshot.



```
acadgild@localhost:~  
File Edit View Search Terminal Tabs Help  
acadgild@localhost:~ acadgild@localhost:~ acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:24  
scala> dataList.collect()  
res32: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
scala> dataList.sum  
res33: Double = 55.0  
scala> dataList.count()  
res34: Long = 10  
scala> █
```

➤ Calculate the average of the numbers in the list

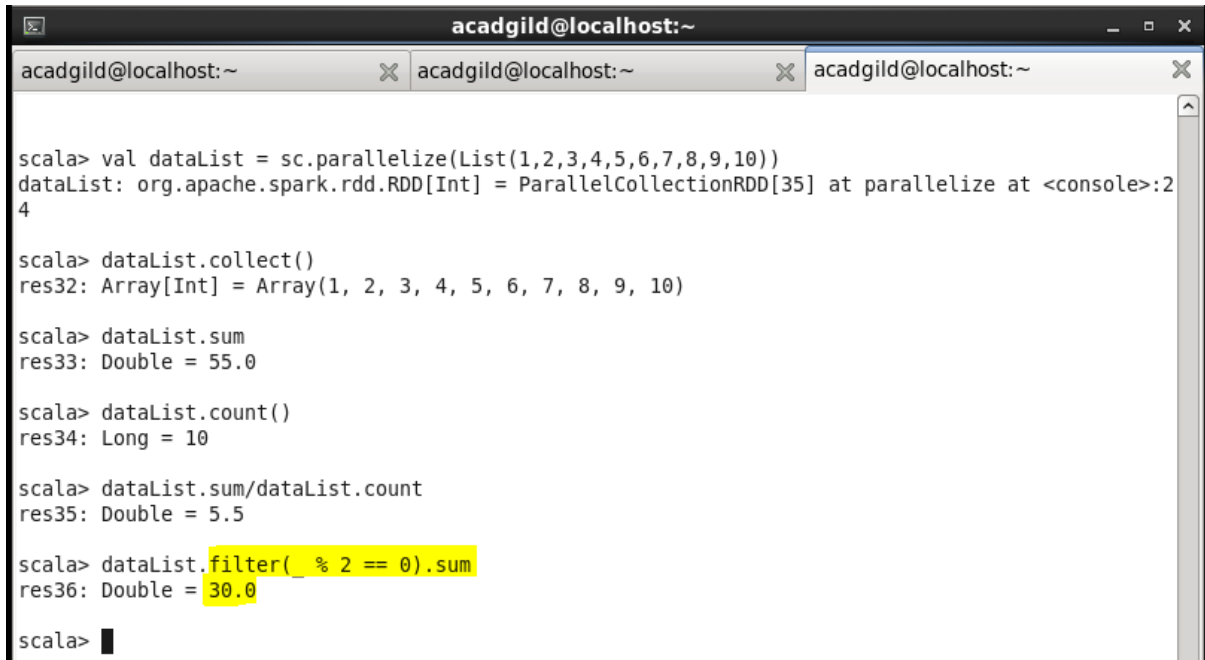
Average of all the numbers in the list is found using the command “**dataList.sum/dataList.count**” as shown in the below screenshot.



```
acadgild@localhost:~  
File Edit View Search Terminal Tabs Help  
acadgild@localhost:~ acadgild@localhost:~ acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:24  
scala> dataList.collect()  
res32: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
scala> dataList.sum  
res33: Double = 55.0  
scala> dataList.count()  
res34: Long = 10  
scala> dataList.sum/dataList.count  
res35: Double = 5.5  
scala> █
```

➤ Find the sum of all the even numbers in the list

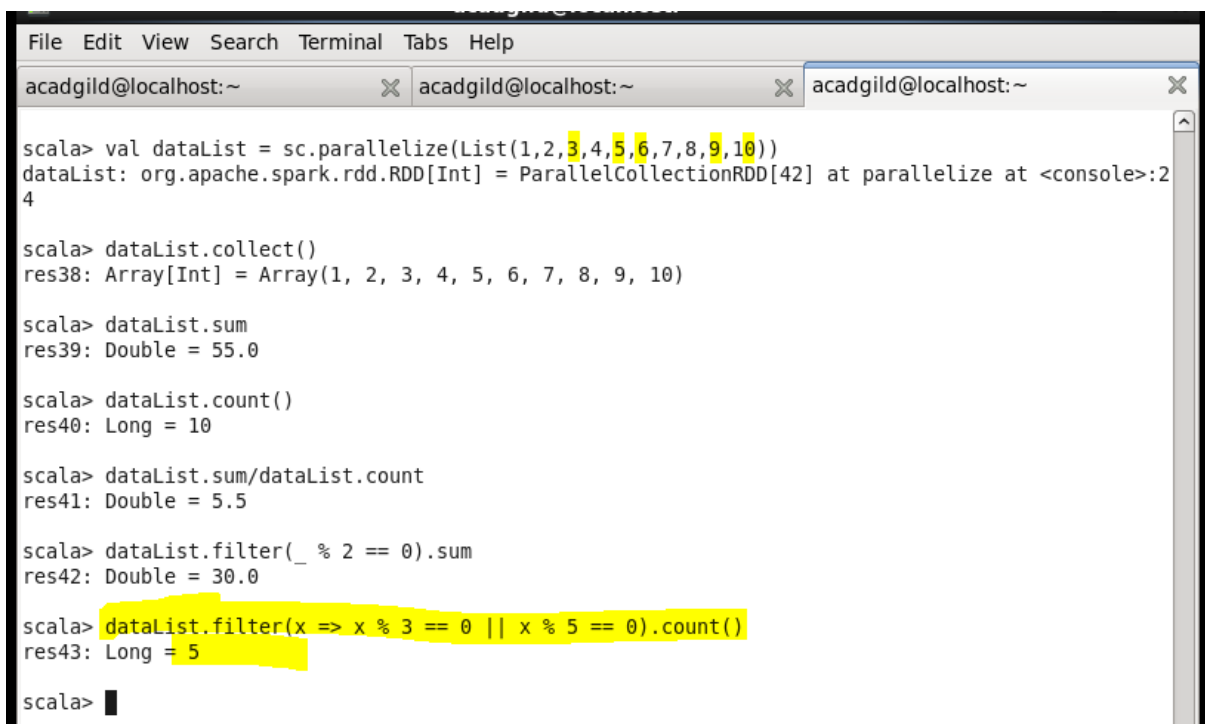
The sum of all the even numbers in the list is found using the command “**dataList.filter(\_ % 2 == 0).sum**” as shown in the below screenshot.



```
acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[35] at parallelize at <console>:24  
  
scala> dataList.collect()  
res32: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
scala> dataList.sum  
res33: Double = 55.0  
  
scala> dataList.count()  
res34: Long = 10  
  
scala> dataList.sum/dataList.count  
res35: Double = 5.5  
  
scala> dataList.filter(_ % 2 == 0).sum  
res36: Double = 30.0  
  
scala> █
```

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

The total number of elements divisible by both 5 and 3 in the list is found using the command “**dataList.filter(x => x % 3 == 0 || x % 5 == 0).count()**” as shown in the below screenshot.



```
File Edit View Search Terminal Tabs Help  
acadgild@localhost:~  
scala> val dataList = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))  
dataList: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[42] at parallelize at <console>:24  
  
scala> dataList.collect()  
res38: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
scala> dataList.sum  
res39: Double = 55.0  
  
scala> dataList.count()  
res40: Long = 10  
  
scala> dataList.sum/dataList.count  
res41: Double = 5.5  
  
scala> dataList.filter(_ % 2 == 0).sum  
res42: Double = 30.0  
  
scala> dataList.filter(x => x % 3 == 0 || x % 5 == 0).count()  
res43: Long = 5  
  
scala> █
```

## Task 2:

### 1) Pen down the limitations of MapReduce.

Various limitations of Apache Hadoop are given below along with their solution-

#### ○ **Issues with Small Files:**

The main problem with Hadoop is that it is not suitable for small data. **HDFS** lacks the ability to support the random reading of small due to its high capacity design.

Small files are smaller than the HDFS Block size (default 128MB). If you are storing these huge numbers of small files, HDFS cannot handle these lots of small files. As HDFS was designed to work with a small number of large files for storing large data sets rather than a large number of small files. If there are lot many small files, then the NameNode will be overloaded since it stores the namespace of HDFS.

#### ○ **Slow Processing Speed**

MapReduce processes a huge amount of data. In Hadoop, MapReduce works by breaking the processing into phases: **Map** and **Reduce**. So, MapReduce requires a lot of time to perform these tasks, thus increasing latency. Hence, reduces processing speed.

#### ○ **Support for Batch Processing only**

Hadoop only supports batch processing, it is not suitable for streaming data. Hence, overall performance is slower. MapReduce framework doesn't leverage the memory of the Hadoop cluster to the maximum.

#### ○ **No Real-time Processing**

Apache Hadoop is a batch processing framework. It means it takes a huge amount of data in input, processes it and produces the result. Batch processing is very efficient for processing a high volume of data, but depends on the size of data being processed and computational power of the system; an output can be delayed significantly. Apache Hadoop is not suitable for Real-time processing.

- **Latency**

MapReduce in Hadoop is slower because it supports different format, structured and huge amount of data. In MapReduce, Map takes a set of data and converts it into another set of data, where an individual element is broken down into a key-value pair. Reduce takes the output from the map as and Reduce takes the output from the map as input and process further. MapReduce requires a lot of time to perform these tasks thereby increasing latency.

- **No Ease of Use**

MapReduce developer in Hadoop needs to hand code for each and every operation which makes it very difficult to work. In Hadoop, MapReduce has no interactive mode, but adding hive and pig makes working with MapReduce little easier.

- **Security Issue**

Apache Hadoop is challenging in maintaining the complex applications. Hadoop is missing encryption at the storage and network levels, which is a major point of concern. Apache Hadoop supports Kerberos authentication, which is hard to manage.

- **No Caching**

Apache Hadoop is not efficient for caching. MapReduce cannot cache the intermediate data in memory for the further requirement and this diminishes the performance of Hadoop.

- **Lengthy Code**

Apache Hadoop has 1, 20,000 line of code. The number of lines produces the number of bugs. Hence it will take more time to execute the programs.

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

**RDD:** Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs – parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

### Features of RDD:

- **In-Memory**

It is possible to store data in spark RDD. Storing of data in spark RDD is size as well as quantity independent. We can store as much data we want in any size. In-memory computation means operate information in the main random access memory. It requires operating across jobs, not in complicated databases. Since operating jobs in databases slow the drive.

- **Lazy Evaluations**

By its name, it says that on calling some operation, execution process doesn't start instantly. To trigger the execution, an action is a must. Since that action takes place, data inside RDD cannot get transform or available. Through DAG, Spark maintains the record of every operation performed. DAG refers to **Directed Acyclic Graph**.

- **Immutable and Read-only**

Since, RDDs are immutable, which means *unchangeable* over time. That property helps to maintain consistency when we perform further computations. As we cannot make any change in RDD once created, it can only get transformed into new RDDs. This is possible through its transformations processes.

- **Cacheable or Persistence**

We can store all the data in persistent storage, memory, and disk. Memory (most preferred) and disk (less Preferred because of its slow access speed). We can also extract it directly from memory. Hence this property of RDDs makes them useful for fast computations. Therefore, we can perform multiple operations on the same data. Also, leads reusability which also helps to compute faster.

- **Partitioned**

Each dataset is logically partitioned and distributed across nodes over the cluster. They are just partitioned to enhance the processing, not divided internally. This arrangement of partitions provides parallelism.

- **Parallel**

As we discussed earlier, RDDs are logically partitioned over the cluster. While we perform any operations, it executes parallelly on entire data.

- **Fault Tolerance**

While working on any node, if we lost any RDD itself recovers itself. When we apply different transformations on RDDs, it creates a logical execution plan. The logical execution plan is generally known as lineage graph. As a consequence, we may lose RDD as if any fault arises in the machine. So by applying the same computation on that node of the lineage graph, we can recover our same dataset again. As a matter of fact, this process enhances its property of Fault Tolerance.

- **Typed**

We have several types of RDDs which are: RDD [long], RDD [int], RDD [String].

- **Coarse-grained Operations**

RDDs support coarse-grained operations. That means we can perform an operation on entire cluster once at a time.

- **No-Limitations**

There is no specific number that limits the usage of RDD. We can use as much RDDs we require. It totally depends on the size of its memory or disk.

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

## Apache Spark RDD Operations

- Transformations
- Actions

## Transformation Operations

Transformations are kind of operations which will transform your RDD data from one form to another. And when you apply this operation on any RDD, you will get a new RDD with transformed data (RDDs in Spark are immutable, Remember????). Operations like map, filter, flatMap are transformations.

Now there is a point to be noted here and that is when you apply the transformation on any RDD it will not perform the operation immediately. It will create a DAG(Directed Acyclic Graph) using the applied operation, source RDD and function used for transformation. And it will keep on building this graph using the references till you apply any action operation on the last lined up RDD. That is why the transformation in Spark are lazy.

**Narrow transformation** – In Narrow transformation, all the elements that are required to compute the records in single partition live in the single partition of parent RDD. A limited subset of partition is used to calculate the result. Narrow transformations are the result of map(), filter().

**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. The partition may live in many partitions of parent RDD. Wide transformations are the result of groupByKey() and reduceByKey().

### ➤ Map(func)

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).



### ➤ FlatMap()

With the help of **flatMap()** function, to each input element, we have many elements in an output RDD. The most simple use of flatMap() is to split each input string into words.

Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

### ➤ Filter(func)

Spark RDD **filter()** function returns a new RDD, containing only the 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.

### ➤ Distinct()

It returns a new dataset that contains the **distinct** elements of the source dataset. It is helpful to remove duplicate data.

For example, if RDD has elements (Spark, Spark, Hadoop, Flink), then **rdd.distinct()** will give elements (Spark, Hadoop, Flink).

## Action Operations

This kind of operation will also give you another RDD but this operation will trigger all the lined up transformation on the base RDD (or in the DAG) and then execute the action operation on the last RDD. Operations like collect, count, first, saveAsTextFile are actions.

### ➤ 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.

### ➤ 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.

Action Collect() had a constraint that all the data should fit in the machine, and copies to the driver.

### ➤ Take(n)

The action **take(n)** returns n number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection. We cannot presume the order of the elements.

For example, consider RDD {1, 2, 2, 3, 4, 5, 5, 6} in this RDD “take (4)” will give result { 2, 2, 3, 4}

### ➤ Top()

If ordering is present in our RDD, then we can extract top elements from our RDD using **top()**. Action *top()* use default ordering of data.

### ➤ CountByValue()

The countByValue() returns, many times each element occur in RDD.

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