RDD In Spark

- An RDD (Resilient Distributed Dataset) is a core data structure in Apache Spark, forming its backbone since its inception. It represents an immutable, fault-tolerant collection of elements that can be processed in parallel across a cluster of machines.
- RDDs serve as the fundamental building blocks in Spark, upon which newer data structures like datasets and data frames are constructed.
- RDDs are designed for distributed computing, dividing the dataset into logical partitions. This logical partitioning enables efficient and scalable processing by distributing different data segments across different nodes within the cluster.
- RDDs encompass a wide range of operations, including transformations (such as map, filter, and reduce) and actions (like count and collect). These operations allow users to perform complex data manipulations and computations on RDDs. RDDs provide fault tolerance by keeping track of the lineage information necessary to reconstruct lost partitions.

In summary, RDDs serve as the foundational data structure in Spark, enabling distributed processing and fault tolerance. They are integral to achieving efficient and scalable data processing in Apache Spark.

Features of Spark RDD:

1. **Immutability**: You cannot change the state of RDD. If you want to change the state of RDD, you need to create a copy of the existing RDD and perform your required operations.
2. **In- Memory Computation**: Spark supports in-memory computation which stores data in RAM instead of disk. Hence, the computation power of Spark is highly increased.
3. **Lazy-evaluation**: Transformations in RDDs are implemented using lazy operations. In lazy evaluation, the results are not computed immediately. It will generate the results, only when the action is triggered. Thus, the performance of the program is increased.
4. **Fault Toleran**t: once you perform any operations in an existing RDD, a new copy of that RDD is created, and the operations are performed on the newly created RDD. Thus, any lost data can be recovered easily and recreated. This feature makes Spark RDD fault-tolerant.
5. **Partitioning**: Data items in RDDs are usually huge. This data is partitioned and send across different nodes for distributed computing.
6. **Persistence**: Intermediate results generated by RDD are stored to make the computation easy. It makes the process optimized.

Creation of RDD:

In Apache Spark, RDDs can be created in three ways.

- Parallelize method by which already existing collection can be used in the driver program.
- By referencing a dataset that is present in an external storage system such as HDFS, HBase.
- New RDDs can be created from an existing RDD.

Operations of RDD:

1. **Transformation:** Transformations are the processes that you perform on an RDD to get a result which is also an RDD. The example would be applying functions such as **filter(), union(), map(), flatMap(), distinct(), reduceByKey(), mapPartitions(), sortBy()** that would create an another resultant RDD. Lazy evaluation is applied in the creation of RDD.
2. **Actions :** Actions return results to the driver program or write it in a storage and kick off a computation. Some examples are **count(), first(), collect(), take(), countByKey(), collectAsMap(),** and **reduce().**

Transformations will always return RDD whereas actions return some other data type.

When To Use RDDs:

- RDD is preferred to use when you want to apply low-level transformations and actions. It gives you a greater handle and control over your data.
- RDDs can be used when the data is highly unstructured such as media or text streams.
- RDDs are used when you want to add functional programming constructs rather than domain-specific expressions.
- RDDs are used in the situation where the schema is not applied.

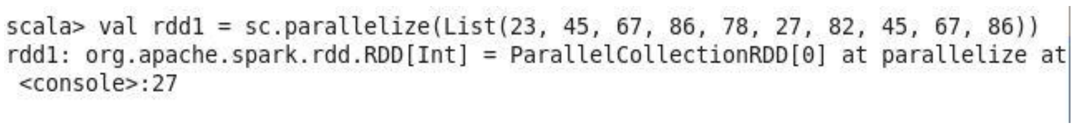
Practical Demo of RDD Operations:

Create RDD:

create an RDD using parallelize() method which is the simplest method.

val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))

sc denotes SparkContext and each element is copied to form RDD.

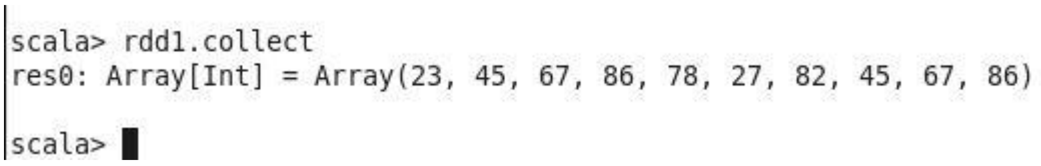


```
scala> val rdd1 = sc.parallelize(List(23, 45, 67, 86, 78, 27, 82, 45, 67, 86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at
 <console>:27
```

Read Result:

We can read the result generated by RDD by using the collect operation.

rdd1.collect

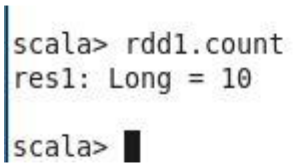


```
scala> rdd1.collect
res0: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)

scala> █
```

Count:

The count action is used to get the total number of elements present in the particular RDD.

rdd1.count

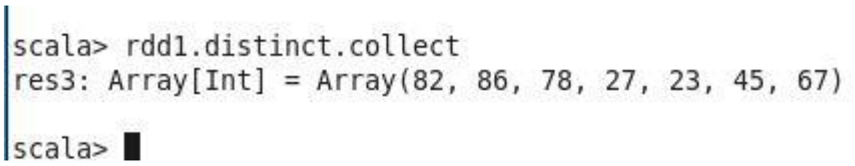


```
scala> rdd1.count
res1: Long = 10

scala> █
```

Distinct:

Distinct is a type of transformation that is used to get the unique elements in the RDD.

rdd1.distinct.collect

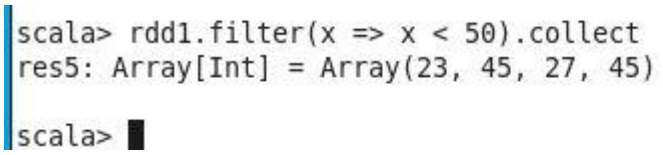


```
scala> rdd1.distinct.collect
res3: Array[Int] = Array(82, 86, 78, 27, 23, 45, 67)

scala> █
```

Filter:

Filter transformation creates a new dataset by selecting the elements according to the given condition.

rdd1.filter(x => x < 50).collect



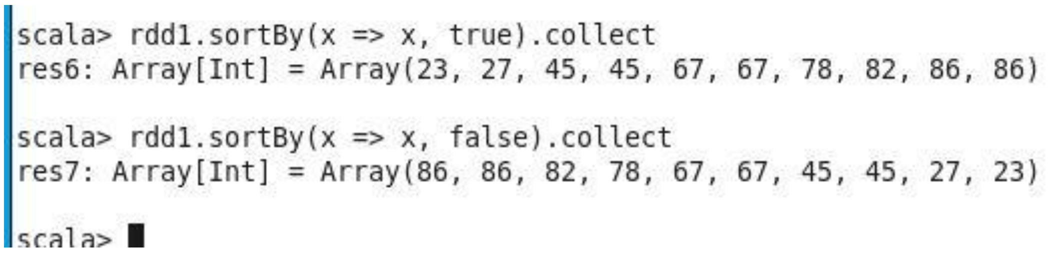
```
scala> rdd1.filter(x => x < 50).collect
res5: Array[Int] = Array(23, 45, 27, 45)

scala> █
```

sortBy:

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

rdd1.sortBy(x => x, true).collect

rdd1.sortBy(x => x, false).collect

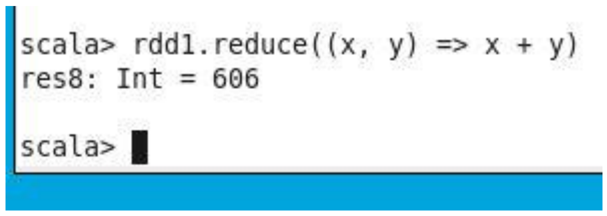


```
scala> rdd1.sortBy(x => x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)

scala> rdd1.sortBy(x => x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)

scala> █
```

Reduce:

Reduce action is used to summarize the RDD based on the given formula.

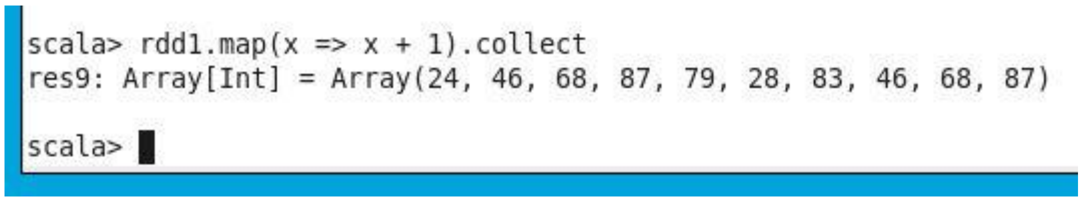
rdd1.reduce((x, y) => x + y)


```
scala> rdd1.reduce((x, y) => x + y)
res8: Int = 606

scala> 
```

Map:

Map transformation processes each element in the RDD according to the given condition and creates a new RDD.

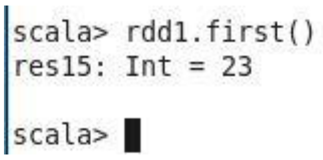
rdd1.map(x => x + 1).collect


```
scala> rdd1.map(x => x + 1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)

scala> 
```

First:

First is a type of action that always returns the first element of the RDD.

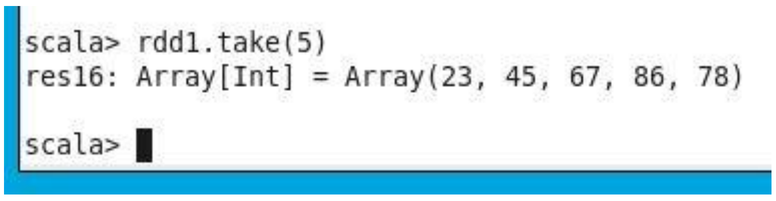
rdd1.first()


```
scala> rdd1.first()
res15: Int = 23

scala> 
```

Take:

Take action returns the first n elements in the RDD.

rdd1.take(5)


```
scala> rdd1.take(5)
res16: Array[Int] = Array(23, 45, 67, 86, 78)

scala> 
```