Apache Spark

Apache Spark is a unified analytics engine for large-scale data processing. It provides high level APIs like Spark SQL, Spark Streaming, MLib, and GraphX to allow interaction with core functionalities of Apache Spark. Spark also facilitates several core data abstractions on top of the distributed collection of data which are RDDs, DataFrames, and DataSets.

Spark RDD (Resilient Distributed Dataset)

Spark RDD stands for **Resilient Distributed Dataset** which is the core data abstraction API and is available since very first release of Spark (**Spark 1.0**). It is a lower-level API for manipulating distributed collection of data. The RDD APIs exposes some extremely useful methods which can be used to get very tight control over underlying physical data structure. It is an immutable (read only) collection of partitioned data distributed on different machines. RDD enables in-memory computation on large clusters to speed up big data processing in a fault tolerant manner.

To enable fault tolerance, RDD uses **DAG (Directed Acyclic Graph)** which consists of a set of vertices and edges. The vertices and edges in DAG represent the RDD and the operation to be applied on that RDD respectively. The transformations defined on RDD are lazy and executes only when an action is called. Let's have a quick look at features and limitations of RDD:

RDD Features:

1. **Immutable collection:** RDD is an immutable partitioned collection distributed on different nodes. A partition is a basic unit of parallelism in Spark. The immutability helps to achieve fault tolerance and consistency.
2. **Distributed data:** RDD is a collection of distributed data which helps in big data processing by distributing the workload to different nodes in the cluster.
3. **Lazy evaluation:** The defined transformations do not gets evaluated until an action is called. It helps Spark in optimizing the overall transformations in one go.
4. **Fault tolerant:** RDD can be recomputed in case of any failure using DAG(Directed acyclic graph) of transformations defined for that RDD.
5. **Multi-language support:** RDD APIs supports **Python, R, Scala, and Java** programming languages.

Limitation of RDD:

1. **No optimization engine:** RDD does not have an in-built optimization engine. Programmers need to write their own code in order to minimize the memory usage and to improve execution performance.

Spark DataFrame

Spark 1.3 introduced two new data abstraction APIs – **DataFrame and DataSet**. The DataFrame APIs organizes the data into named columns like a table in relational database. It enables programmers to define schema on a distributed collection of data. Each row in a DataFrame is of object type row. Like an SQL table, each column must have same number of rows in a DataFrame. In short, DataFrame is lazily evaluated plan which specifies the operations needs to be performed on the distributed collection of the data. DataFrame is also an immutable collection. Below are the features and limitations of DataFrame:

DataFrame Features:

1. **In-built Optimization:** DataFrame uses **Catalyst** engine which has an in-built execution optimization that improves the data processing performance significantly. When an action is called on a DataFrame, the Catalyst engine analyzes the code and resolves the references. Then, it creates a logical plan. After that, the created logical plan gets translated into an optimized physical plan. Finally, this physical plan gets executed on the cluster.
2. **Hive compatible:** The DataFrame is fully compatible with Hive query language. We can access all hive data, queries, UDFs, etc using Spark SQL from hive MetaStore and can execute queries against these hive databases.
3. **Structured, semi-structured, and highly structured data support:** DataFrame APIs supports manipulation of all kind of data from structured data files to semi-structured data files and highly structured parquet files.
4. **Multi-language support:** DataFrame APIs are available in **Python, R, Scala, and Java**.
5. **Schema support:** We can define a schema manually or we can read a schema from a data source which defines the column names and their data types.

DataFrame Limitations:

1. **Type safety:** Each row in a DataFrame is of object type row and hence is not strictly typed. That is why DataFrame does not support compile time safety.

Spark DataSet

As an extension to the DataFrame APIs, **Spark 1.3** also introduced DataSet APIs which provides strictly typed and object-oriented programming interface in Spark. It is immutable, type-safe collection of distributed data. Like DataFrame, DataSet APIs also uses Catalyst engine in order to enable execution optimization. DataSet is an extension to the DataFrame APIs. Features and limitations of the DataSet are as below:

DataSet Features:

1. **Combination of RDD and DataFrame:** DataSet enables functional programming like RDD APIs and relational queries and execution optimization like DataFrame APIs. Thus, it provides the benefit of best of both worlds – RDDs and DataFrames.
2. **Type-safe:** Unlike DataFrames, DataSet APIs provides compile time type safety. It conforms the specification at compile time using defined case classes (for Scala) or Java beans (for Java).

DataSet Limitations:

1. **Limited language support:** DataSet is only available to JVM based langauges like Java and Scala. Python and R do not support DataSet because these are dynamically typed languages.
2. **High garbage collection:** JVM types can cause high garbage collection and object instantiation cost.

RDD vs DataFrame vs DataSet

Below table represents a quick comparison between RDD, DataFrame and DataSet:

Feature	RDD	DataFrame	DataSet
Immutable	Yes	Yes	Yes
Fault tolerant	Yes	Yes	Yes
Type-safe	Yes	No	Yes
Schema	No	Yes	Yes
Execution optimization	No	Yes	Yes
Level	Low	High	High

RDD, DataFrame, and DataSet – Comparison

References:

https://sqlrelease.com/rdd-dataframe-and-dataset-introduction-to-spark-data-abstraction#:~:text=in%20Apache%20Spark.-,Spark%20Data%20Abstraction,computations%20in%20a%20distributed%20way.

https://data-flair.training/forums/topic/how-many-abstractions-are-provided-by-apache-spark/

https://spark.apache.org/docs/3.4.0/rdd-programming-guide.html#resilient-distributed-datasets-rdds

https://spark.apache.org/docs/latest/quick-start.html

https://spark.apache.org/docs/latest/rdd-programming-guide.html

https://data-flair.training/blogs/spark-rdd-operations-transformations-actions/

Parallelized Collections and RDD examples in Spark:

```
data = [1, 2, 3, 4, 5]
distData = sc.parallelize(data)
```

Interactive Analysis with the Spark Shell

Basics

Spark's shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively. It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries) or Python. Start it by running the following in the Spark directory:

- **Scala**
- **Python**

```
./bin/pyspark
```

Or if PySpark is installed with pip in your current environment:

```
pyspark
```

Spark's primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Python's dynamic nature, we don't need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it `DataFrame` to be consistent with the data frame concept in Pandas and R. Let's make a new DataFrame from the text of the README file in the Spark source directory:

```
>>> textFile = spark.read.text("README.md")
```

You can get values from DataFrame directly, by calling some actions, or transform the DataFrame to get a new one. For more details, please read the *API doc*.

```
>>> textFile.count()  # Number of rows in this DataFrame
126

>>> textFile.first()  # First row in this DataFrame
Row(value=u'# Apache Spark')
```

Now let's transform this DataFrame to a new one. We call `filter` to return a new DataFrame with a subset of the lines in the file.

```
>>> linesWithSpark = textFile.filter(textFile.value.contains("Spark"))
```

We can chain together transformations and actions:

```
>>> textFile.filter(textFile.value.contains("Spark")).count()  # How many
lines contain "Spark"?
15
```

More on Dataset Operations

Dataset actions and transformations can be used for more complex computations. Let's say we want to find the line with the most words:

- **Scala**
- **Python**

```
>>> from pyspark.sql.functions import *
>>> textFile.select(size(split(textFile.value,
"\s+")).name("numWords")).agg(max(col("numWords"))).collect()
[Row(max(numWords)=15)]
```

This first maps a line to an integer value and aliases it as “numWords”, creating a new DataFrame. `agg` is called on that DataFrame to find the largest word count. The arguments to `select` and `agg` are both *Column*, we can use `df.colName` to get a column from a DataFrame. We can also import pyspark.sql.functions, which provides a lot of convenient functions to build a new Column from an old one.

One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily:

```
>>> wordCounts = textFile.select(explode(split(textFile.value,
"\s+")).alias("word")).groupBy("word").count()
```

Here, we use the `explode` function in `select`, to transform a Dataset of lines to a Dataset of words, and then combine `groupBy` and `count` to compute the per-word counts in the file as a DataFrame of 2 columns: “word” and “count”. To collect the word counts in our shell, we can call `collect`:

```
>>> wordCounts.collect()
[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...]
```

Spark RDD Operations- Transformations & Actions with Example

1. Spark RDD Operations

Two types of **Apache Spark** RDD operations are- Transformations and Actions. A **Transformation** is a function that produces new **RDD** from the existing RDDs but when we want to work with the actual dataset, at that point **Action** is performed. When the action is triggered after the result, new RDD is not formed like transformation.

Spark RDD Operations-Transformation & Action with Example

2. Apache Spark RDD Operations

Before we start with Spark RDD Operations, let us deep dive into RDD in Spark.

Apache Spark RDD supports two types of Operations-

- Transformations
- Actions

Now let us understand first what is Spark RDD Transformation and Action-

3. RDD Transformation

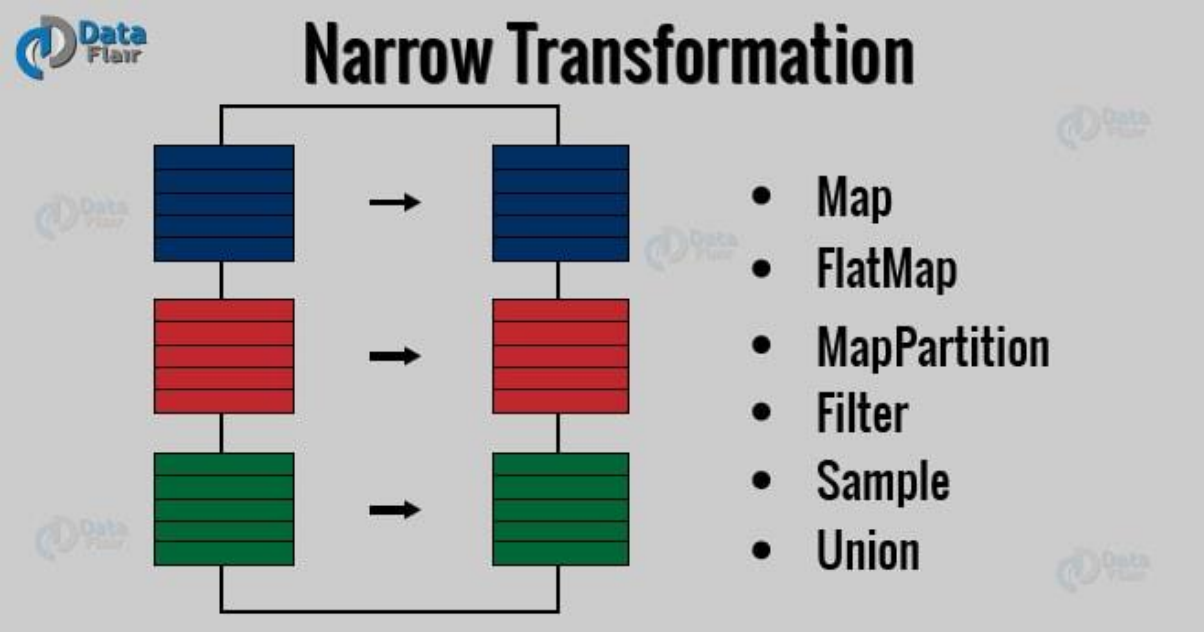
Spark Transformation is a function that produces new RDD from the existing RDDs. It takes RDD as input and produces one or more RDD as output. Each time it creates new RDD when we apply any transformation. Thus, the so input RDDs, cannot be changed since RDD are immutable in nature.

Applying transformation built an **RDD lineage**, with the entire parent RDDs of the final RDD(s). RDD lineage, also known as **RDD operator graph** or **RDD dependency graph.** It is a logical execution plan i.e., it is Directed Acyclic Graph (DAG) of the entire parent RDDs of RDD.

Transformations are lazy in nature i.e., they get execute when we call an action. They are not executed immediately. Two most basic type of

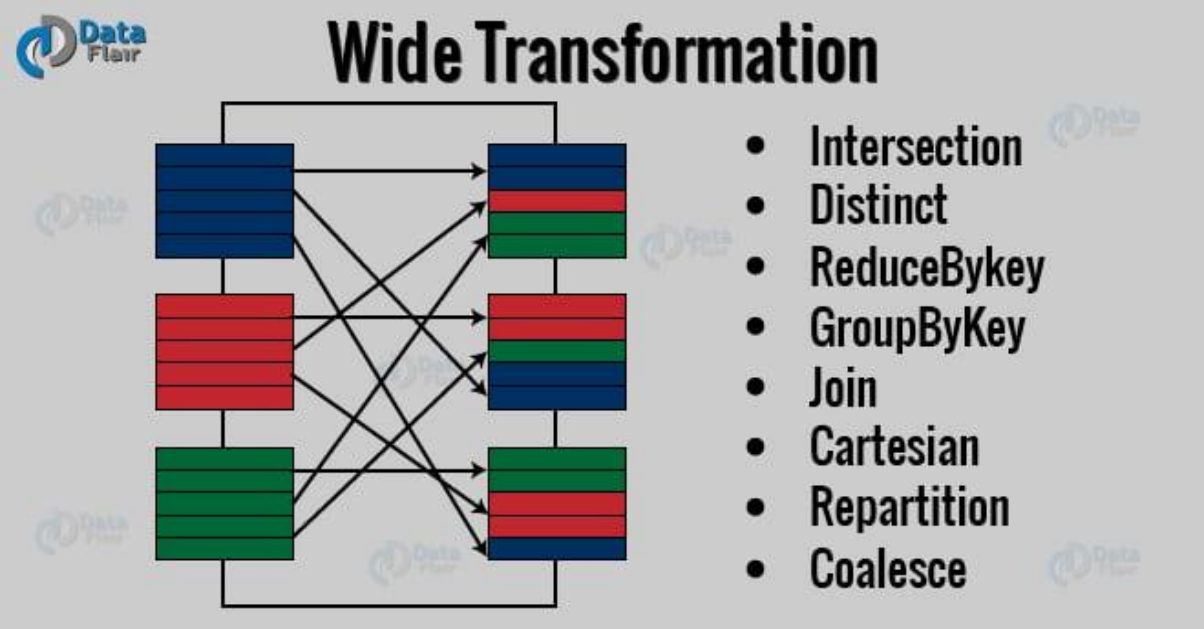
transformations is a map(), filter().
After the transformation, the resultant RDD is always different from its parent RDD. It can be smaller (e.g. filter, count, distinct, sample), bigger (e.g. flatMap(), union(), Cartesian()) or the same size (e.g. map).
There are two types of transformations:

- **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().*



Apache Spark Narrow Transformation Operation

- **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()*.



Spark Wide Transformation Operations

There are various functions in RDD transformation. Let us see RDD transformation with examples.

3.1. 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).

Also Read: **How to create RDD**
Map() example:
[php]import org.apache.spark.SparkContext
import org.apache.spark.SparkConf
import org.apache.spark.sql.SparkSession
object mapTest{
def main(args: Array[String]) = {

val spark =
SparkSession.builder.appName(“mapExample”).master(“local”).getOrCrea
te()
val data = spark.read.textFile(“spark_test.txt”).rdd
val mapFile = data.map(line => (line,line.length))
mapFile.foreach(println)
}
}[/php]

spark_test.txt”

```
hello...user! this file is created to check the operations of
spark.


?, and how can we apply functions on that RDD partitions?. All
this will be done through spark programming which is done with
the help of scala language support…
```

- ***Note –*** In above code, map() function map each line of the file with its length.

3.2. 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.
flatMap() example:
[php]val data = spark.read.textFile(“spark_test.txt”).rdd
val flatmapFile = data.flatMap(lines => lines.split(” “))
flatmapFile.foreach(println)[/php]

- ***Note –*** In above code, flatMap() function splits each line when space occurs.

3.3. 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.

Filter() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")
println(mapFile.count())[/php]

- ***Note*** – In above code, flatMap function map line into words and then count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

Read: Apache Spark RDD vs DataFrame vs DataSet

3.4. mapPartitions(func)

The **MapPartition** converts each *partition* of the source RDD into many elements of the result (possibly none). In mapPartition(), the map() function is applied on each partitions simultaneously. MapPartition is like a map, but the difference is it runs separately on each partition(block) of the RDD.

3.5. mapPartitionWithIndex()

It is like mapPartition; Besides mapPartition it provides *func* with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.
Learn: Spark Shell Commands to Interact with Spark-Scala

3.6. union(dataset)

With the **union()** function, we get the elements of both the RDD 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).

Union() example:

```
[php]val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",
2014)))
val rdd2 =
spark.sparkContext.parallelize(Seq((5,"dec",2014),(17,"sep",2015)))
val rdd3 =
spark.sparkContext.parallelize(Seq((6,"dec",2011),(16,"may",2015)))
val rddUnion = rdd1.union(rdd2).union(rdd3)
rddUnion.foreach(Println)[/php]
```

- ***Note –*** In above code union() operation will return a new dataset that contains the union of the elements in the source dataset (rdd1) and the argument (rdd2 & rdd3).

3.7. intersection(other-dataset)

With the **intersection()** function, we get only the common element of both the RDD in new RDD. The key rule of this function is that the two RDDs should be of the same type.

Consider an example, the elements of **RDD1** are (Spark, Spark, Hadoop, Flink) and that of **RDD2** are (Big data, Spark, Flink) so the resultant ***rdd1.intersection(rdd2)*** will have elements (spark).

Intersection() example:

```
[php]val rdd1 =
spark.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014,
(16,"feb",2014)))
val rdd2 =
spark.sparkContext.parallelize(Seq((5,"dec",2014),(1,"jan",2016)))
val comman = rdd1.intersection(rdd2)
comman.foreach(Println)[/php]
```

- ***Note –*** The intersection() operation return a new RDD. It contains the intersection of elements in the rdd1 & rdd2.

Learn to Install Spark on Ubuntu

3.8. 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).
Distinct() example:

```
[php]val rdd1 =
park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2014)))
val result = rdd1.distinct()
println(result.collect().mkString(", "))[/php]
```

- ***Note –*** In the above example, the distinct function will remove the duplicate record i.e. (3,'"nov",2014).

3.9. groupByKey()

When we use **groupByKey()** on a dataset of (K, V) pairs, the data is shuffled according to the key value K in another RDD. In this transformation, lots of unnecessary data get to transfer over the network.
Spark provides the provision to save data to disk when there is more data shuffled onto a single executor machine than can fit in memory. Follow this link to **learn about RDD Caching and Persistence mechanism** in detail.
groupByKey() example:

```
[php]val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)[/php]
```

- ***Note –*** The groupByKey() will group the integers on the basis of same key(alphabet). After that *collect()* action will return all the elements of the dataset as an Array.

3.10. reduceByKey(func, [numTasks])

When we use **reduceByKey** on a dataset (K, V), the pairs on the same machine with the same key are combined, before the data is shuffled.
reduceByKey() example:
[php]val words =
Array("one","two","two","four","five","six","six","eight","nine","ten")
val data = spark.sparkContext.parallelize(words).map(w =>
(w,1)).reduceByKey(_+_)
data.foreach(println)[/php]

- ***Note –*** The above code will parallelize the Array of String. It will then map each word with count 1, then reduceByKey will merge the count of values having the similar key.

Read: Various Features of RDD

3.11. sortByKey()

When we apply the **sortByKey() function** on a dataset of (K, V) pairs, the data is sorted according to the key K in another RDD.
sortByKey() example:
[php] val data =
spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science", 82), ("computer",65), ("maths",85)))
val sorted = data.sortByKey()
sorted.foreach(println)[/php]

- ***Note –*** In above code, sortByKey() transformation sort the data RDD into Ascending order of the Key(String).

Read: Limitations of RDD

3.12. join()

The **Join** is database term. It combines the fields from two table using common values. join() operation in Spark is defined on pair-wise RDD. Pair-wise RDDs are RDD in which each element is in the form of tuples. Where the first element is key and the second element is the value.
The boon of using keyed data is that we can combine the data together. The join() operation combines two data sets on the basis of the key.

Join() example:

[php]val data = spark.sparkContext.parallelize(Array((‘A’,1),(‘b’,2),(‘c’,3)))
val data2
=spark.sparkContext.parallelize(Array((‘A’,4),(‘A’,6),(‘b’,7),(‘c’,3),(‘c’,8)))
val result = data.join(data2)
println(result.collect().mkString(“,”))[/php]

- ***Note*** – The join() transformation will join two different RDDs on the basis of Key.

Read: RDD lineage in Spark: ToDebugString Method

3.13. coalesce()

To avoid full shuffling of data we use coalesce() function. In **coalesce()** we use existing partition so that less data is shuffled. Using this we can cut the number of the partition. Suppose, we have four nodes and we want only two nodes. Then the data of extra nodes will be kept onto nodes which we kept.

Coalesce() example:

[php]val rdd1 =
spark.sparkContext.parallelize(Array(“jan”,”feb”,”mar”,”april”,”may”,”jun”)
,3)
val result = rdd1.coalesce(2)
result.foreach(println)[/php]

- ***Note –*** The coalesce will decrease the number of partitions of the source RDD to numPartitions define in coalesce argument.

4. RDD Action

Transformations create RDDs from each other, but when we want to work with the actual dataset, at that point action is performed. When the action is triggered after the result, new RDD is not formed like transformation. Thus, Actions are Spark RDD operations that give non-RDD values. The values of action are stored to drivers or to the external storage system. It brings laziness of RDD into motion.

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. Some of the actions of Spark are:

4.1. 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.

Count() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.flatMap(lines => lines.split(" ")).filter(value =>
value=="spark")
println(mapFile.count())[/php]

- ***Note –*** In above code *flatMap()* function maps line into words and count the word "Spark" using *count()* Action after filtering lines containing "Spark" from mapFile.

Learn: Spark Streaming

4.2. 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.
Collect() example:
[php]val data = spark.sparkContext.parallelize(Array(('A',1),('b',2),('c',3)))
val data2
=spark.sparkContext.parallelize(Array(('A',4),('A',6),('b',7),('c',3),('c',8)))
val result = data.join(data2)
println(result.collect().mkString(","))[/php]

- ***Note –*** *join()* transformation in above code will join two RDDs on the basis of same key(alphabet). After that *collect()* action will return all the elements to the dataset as an Array.

4.3. 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}

Take() example:
[php]val data =
spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)

val group = data.groupByKey().collect()

val twoRec = result.take(2)

twoRec.foreach(println)[/php]

- ***Note*** – The *take(2)* Action will return an array with the first *n* elements of the data set defined in the taking argument.

Learn: Apache Spark DStream (Discretized Streams)

4.4. 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.
Top() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val mapFile = data.map(line => (line,line.length))
val res = mapFile.top(3)
res.foreach(println)[/php]

- ***Note*** – *map()* operation will map each line with its length. And top(3) will return 3 records from mapFile with default ordering.

4.5. 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)}

countByValue() example:
[php]val data = spark.read.textFile("spark_test.txt").rdd
val result= data.map(line => (line,line.length)).countByValue()
result.foreach(println)[/php]

- ***Note –*** The *countByValue()* action will return a hashmap of (K, Int) pairs with the count of each key.

Learn: Apache Spark Streaming Transformation Operations

4.6. reduce()

The **reduce()** function takes the two elements as input from the RDD and then produces the output of the same type as that of the input elements. The simple forms of such function are an addition. We can add the elements of RDD, count the number of words. It accepts commutative and associative operations as an argument.
Reduce() example:
[php]val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))
val sum = rdd1.reduce(_+_)
println(sum)[/php]

- ***Note*** – The *reduce()* action in above code will add the elements of the source RDD.

4.7. fold()

The signature of the **fold()** is like *reduce().* Besides, it takes "zero value" as input, which is used for the initial call on each partition. But, the **condition with zero value** is that it should be the **identity element of that operation**. The key difference between *fold()* and *reduce()* is that, *reduce()* throws an exception for empty collection, but *fold()* is defined for empty collection. For example, zero is an identity for addition; one is identity element for multiplication. The return type of *fold()* is same as that of the element of RDD we are operating on.
For example, rdd.fold(0)((x, y) => x + y).
Fold() example:
[php]val rdd1 = spark.sparkContext.parallelize(List(("maths", 80),("science", 90)))
val additionalMarks = ("extra", 4)

val sum = rdd1.fold(additionalMarks){ (acc, marks) => val add = acc._2 + marks._2
("total", add)
}
println(sum)[/php]

- ***Note –*** In above code *additionalMarks* is an initial value. This value will be added to the int value of each record in the source RDD.

Learn: Spark Streaming Checkpoint in Apache Spark

4.8. aggregate()

It gives us the flexibility to get data type different from the input type. The **aggregate()** takes two functions to get the final result. Through one function we combine the element from our RDD with the accumulator, and the second, to combine the accumulator. Hence, in aggregate, we supply the initial zero value of the type which we want to return.

4.9. foreach()

When we have a situation where we want to apply operation on each element of RDD, but it should not return value to the *driver*. In this case, **foreach()** function is useful. For example, inserting a record into the database.

Foreach() example:

[php]val data = spark.sparkContext.parallelize(Array(('k',5),('s',3),('s',4),('p',7),('p',5),('t',8),('k',6)),3)
val group = data.groupByKey().collect()
group.foreach(println)[/php]

- ***Note –*** *The foreach()* action run a function *(println)* on each element of the dataset group.