There are two operations in RDD namely [transformation and Action](http://data-flair.training/blogs/rdd-transformations-actions-apis-apache-spark/).

**a)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. 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**](http://data-flair.training/blogs/directed-acyclic-graph-dag-in-apache-spark/)) of the entire parent RDDs of RDD.

[Transformations are lazy](http://data-flair.training/blogs/lazy-evaluation-in-apache-spark-guide/) 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().*
* **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*.

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

**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).

**Map() example:**

|  |  |
| --- | --- |
|  | 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").getOrCreate()  val data = spark.read.textFile("spark\_test.txt").rdd  val mapFile = data.map(line => (line,line.length))  mapFile.foreach(println)  }  } |

***Note –***In above code map() function if used, which map each line of the file with its length

### 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()](http://data-flair.training/blogs/map-vs-flatmap-operation-in-apache-spark/) is map() returns only one element, while flatMap() can return a list of elements.

**flatMap() example:**

|  |  |
| --- | --- |
|  | val data = spark.read.textFile("spark\_test.txt").rdd  val flatmapFile = data.flatMap(lines => lines.split(" "))  flatmapFile.foreach(println) |

**Note –**In above code flatMap() function if used which split each line when space occurs.

### 3.Filter(func)

Spark RDD **filter** function returns a new RDD, containing only the elements that meet a predicate. It is a *narrow operation* as 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:**

|  |  |
| --- | --- |
|  | val data = spark.read.textFile("spark\_test.txt").rdd  val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")  println(mapFile.count()) |

***Note***– In above code flatMap function used to map line into words and count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

**4.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](http://data-flair.training/blogs/apache-spark-rdd-persistence-caching/) in detail.

**groupByKey() example:**

|  |  |
| --- | --- |
| 1  2 | 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) |

***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.

**5.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:**

|  |  |
| --- | --- |
|  | 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) |

***Note –*** The code written above 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.

**b)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 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.

Spark drivers and external storage system store the value of action. 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:

**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:**

|  |  |
| --- | --- |
|  | val data = spark.read.textFile("spark\_test.txt").rdd  val mapFile = data.flatMap(lines => lines.split(" ")).filter(value => value=="spark")  println(mapFile.count()) |

***Note* –** In above code*flatMap* function used to map line into words and count the word “Spark” using *count()* Action after filtering lines containing “Spark” from mapFile.

**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 expected result.

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

**Collect() example:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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(",")) |

***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.

**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:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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) |

***Note*** – The *take(2)* Action will return an array with the first *n* elements of the dataset defined in thetaking argument.

**4.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:**

|  |  |
| --- | --- |
|  | val rdd1 = spark.sparkContext.parallelize(List(20,32,45,62,8,5))  val sum = rdd1.reduce(\_+\_)  println(sum) |

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