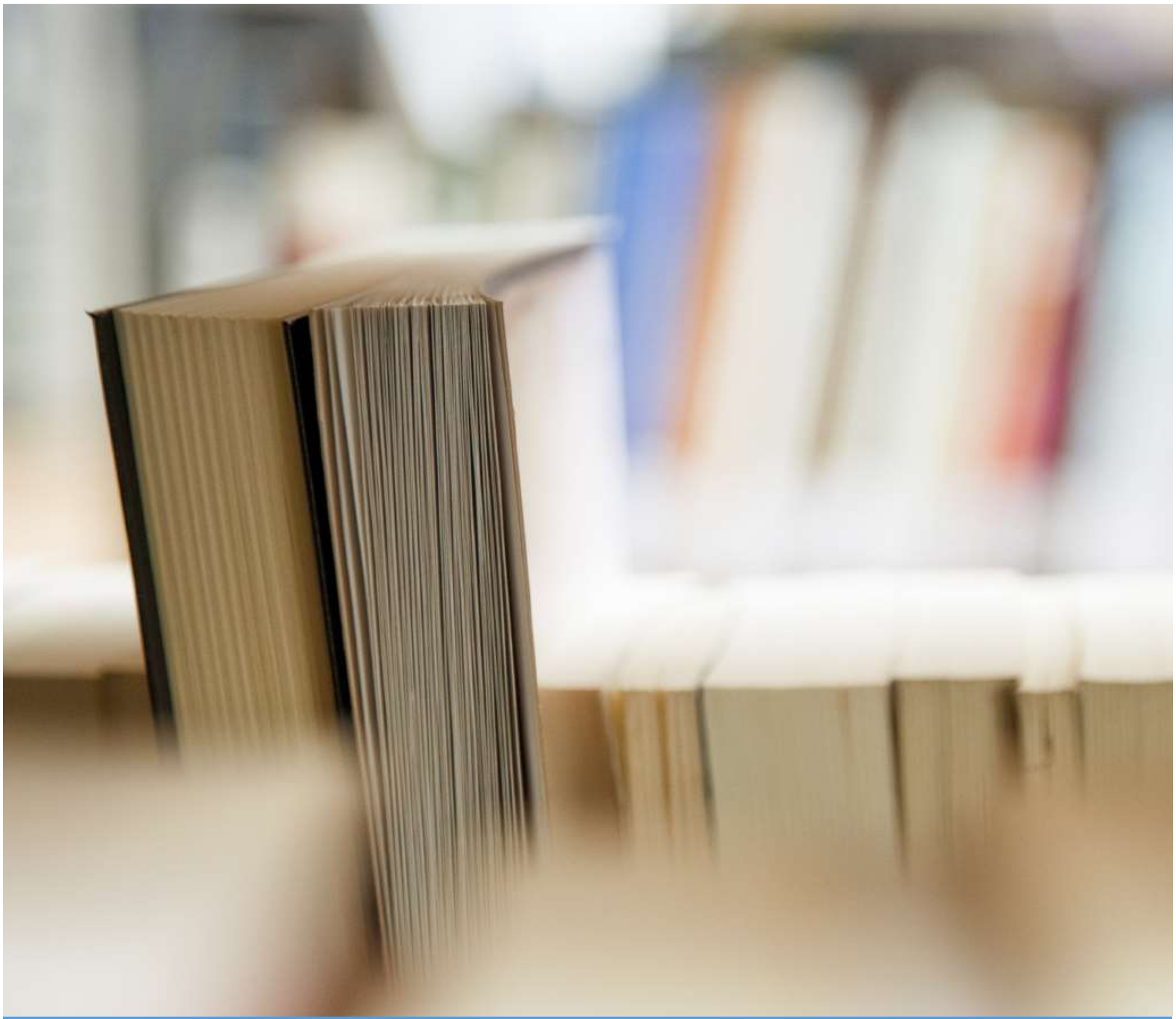


BIG DATA HADOOP & SPARK TRAINING

Assignment on Introduction to Spark

Rashmi Krishna 4/30/18

Task 1

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

Command to create List of integer numbers:

val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))

command explanation:

parallelize: The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.

sc: The first thing a Spark program must do is to create a **SparkContext** object, which tells Spark how to access a cluster.

- find the sum of all numbers

Command to create sum of elements in the list:

val sum = x.reduce(_+_)

command explanation:

reduce(_+_): Is an action, that aggregates all the elements of the RDD using some function and returns the final result to the driver program

- find the total elements in the list

Command to find the number of elements in the list:

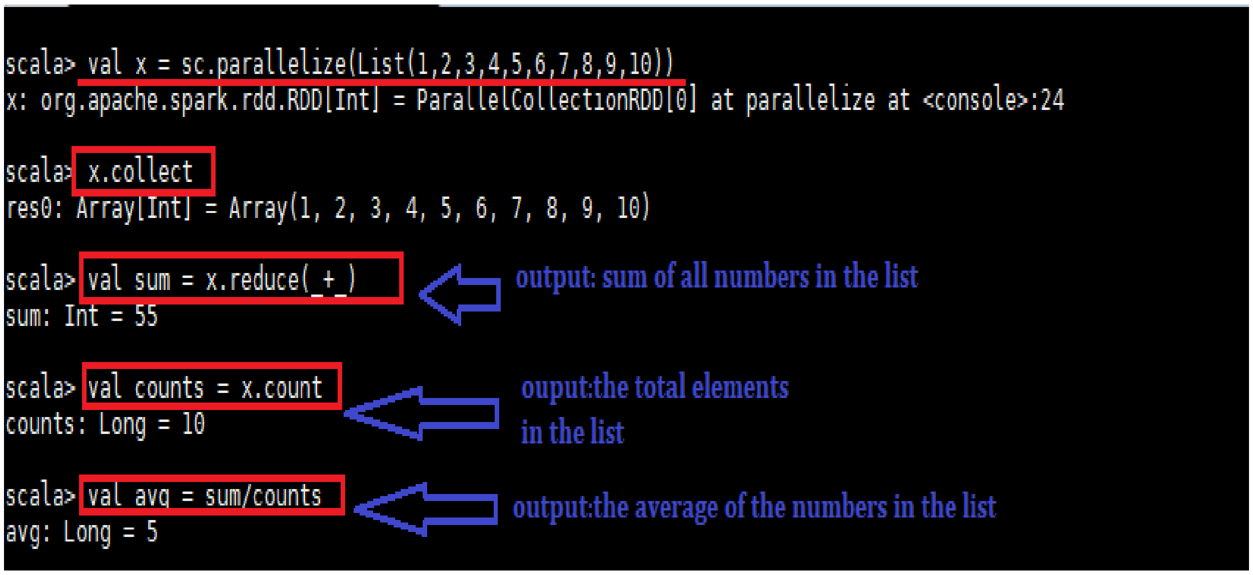
val counts = x.count

- calculate the average of the numbers in the list

To calculate average, we have to compute sum of all elements divided by counts:

val avg = sum/counts

Outputs of all the above commands are as shown below:



```
scala> val x = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))
x: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> x.collect
res0: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val sum = x.reduce(_+_)
sum: Int = 55

scala> val counts = x.count
counts: Long = 10

scala> val avg = sum/counts
avg: Long = 5
```

output: sum of all numbers in the list
ouput:the total elements in the list
output:the average of the numbers in the list

- find the sum of all the even numbers in the list

We compute the sum of even numbers i.e. we calculate the modulus and check if their remainder is zero, if so then the number is divisible by 2.

val filter = x.filter(value => value % 2 == 0).reduce(_+_)

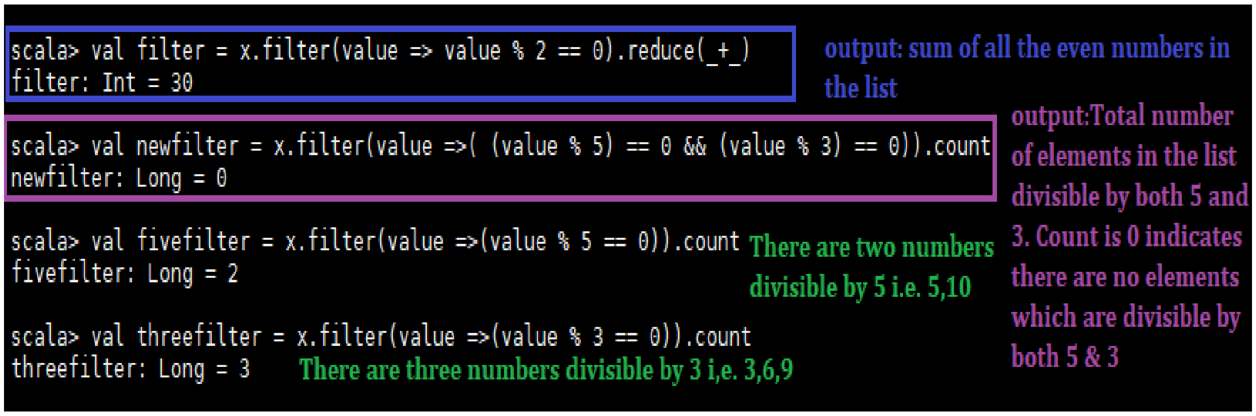
command explanation:

filter: Return a new dataset formed by selecting those elements of the source on which *func* returns true.

- find the total number of elements in the list divisible by both 5 and 3

We compute this by finding modulus of 5 & 3 and check if their remainder is zero, if so then that number is divisible by both 5 & 3

val newfilter = x.filter(value =>((value % 5) == 0 && (value % 3) == 0)).count



```
scala> val filter = x.filter(value => value % 2 == 0).reduce(_+_)
filter: Int = 30

scala> val newfilter = x.filter(value =>( (value % 5) == 0 && (value % 3) == 0)).count
newfilter: Long = 0

scala> val fivefilter = x.filter(value =>(value % 5 == 0)).count
fivefilter: Long = 2

scala> val threefilter = x.filter(value =>(value % 3 == 0)).count
threefilter: Long = 3
```

output: sum of all the even numbers in the list
output:Total number of elements in the list divisible by both 5 and 3. Count is 0 indicates there are no elements which are divisible by both 5 & 3
There are two numbers divisible by 5 i.e. 5,10
There are three numbers divisible by 3 i,e. 3,6,9

Task 2:

1.Pen down the limitations of MapReduce.

MapReduce cannot handle the following:

- Interactive Processing
- Real-time (stream) Processing
- Iterative (delta) Processing
- In-memory Processing
- Graph Processing

1. Issue with Small Files

- **Hadoop** is not suited for small data.
- **HDFS** lacks the ability to efficiently to randomly read small files because of its high capacity design.

2. Slow Processing Speed

- MapReduce process large data sets.
- There are tasks that need to be performed: **Map** and **Reduce** and, MapReduce requires a lot of time to perform these tasks thereby increasing latency.
- Data is distributed and processed over the cluster in MapReduce which increases the time and reduces processing speed.

3. Support for Batch Processing only

- Hadoop does not process streamed data, and hence overall performance is slower.
- MapReduce framework of Hadoop does not leverage the memory of the **Hadoop cluster** to the maximum extent.

4. No Real-time Data Processing

- Hadoop is designed for batch processing, i.e. it takes huge amount of data as input, process it and produce the result.
- Although batch processing is very efficient for processing a high volume of data, but depending on the size of the data being processed and computational power of the system, an output can be delayed significantly.
- Hadoop is not suitable for Real-time data processing.

5. No Delta Iteration

Hadoop is not so efficient for iterative processing, as Hadoop does not support cyclic data flow i.e. a chain of stages in which each output of the previous stage is the input to the next stage.

6. Latency

- In Hadoop, MapReduce framework is comparatively slower, since it is designed to support different format, structure and huge volume of data.
- In **MapReduce**, Map takes a set of data and converts it into another set of data, where individual element are broken down into **key value pair**
- Reduce takes the output from the map as input and process further
- MapReduce requires a lot of time to perform these tasks thereby increases the latency.

7. Not Easy to Use

- In Hadoop, MapReduce developers need to hand code for each and every operation which makes it very difficult to work.
- MapReduce has no interactive mode, but adding one such as **hive** and **pig** makes working with MapReduce a little easier for adopters.

8. No Caching

- Hadoop is not efficient for caching.
- In Hadoop, MapReduce cannot cache the intermediate data in memory for a further requirement which reduces the performance of Hadoop.

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

RDD (Resilient Distributed Dataset) is the fundamental data structure of Spark which are an immutable collection of objects distributed across many compute nodes that can be manipulated in parallel. Each and every dataset in Spark RDD is logically partitioned across many servers so that they can be computed on different nodes of the cluster.

Features of RDD are as follows:

2.1. In-memory computation

The data inside RDD are stored in memory for as long as you want to store. Keeping the data in-memory improves the performance.

2.2. Lazy Evaluation

The data inside RDDs are not evaluated on the go. The changes or the computation is performed only after an action is triggered.

2.3. Fault Tolerance

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one.

2.4. Immutability

RDDS are immutable, if we perform any transformation, it creates new RDD. We achieve consistency through immutability.

2.5. Persistence

We can store the frequently used RDD in in-memory and we can also retrieve them directly from memory without going to disk, this speedup the execution. We can perform Multiple operations on the same data, this happens by storing the data explicitly in memory by calling persist() or cache() function.

2.6. Partitioning

RDD partition the records logically and distributes the data across various nodes in the cluster. The logical divisions are only for processing and internally it has no division. Thus, it provides parallelism.

2.7. Parallel

RDD, process the data in parallel over the cluster.

2.8. Location-Stickiness

RDDs are capable of defining placement preference to compute partitions. Placement preference refers to information about the location of RDD. The DAG Scheduler places the partitions in such a way that task is close to data as much as possible. Thus speed up the computation.

2.9. Coarse-grained Operation

We apply coarse-grained transformations to RDD, i.e. the operation applies to the whole dataset not on an individual element in the data set of RDD.

2.10. Typed

We can have RDD of various types like: RDD [int], RDD [long], RDD [string].

2.11. No limitation

we can have any number of RDD. there is no limit to its number, but depends on the size of disk and memory.

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

Spark RDD Operations

Two types of Apache Spark RDD operations are-

1. *Transformations.*
2. *Actions.*

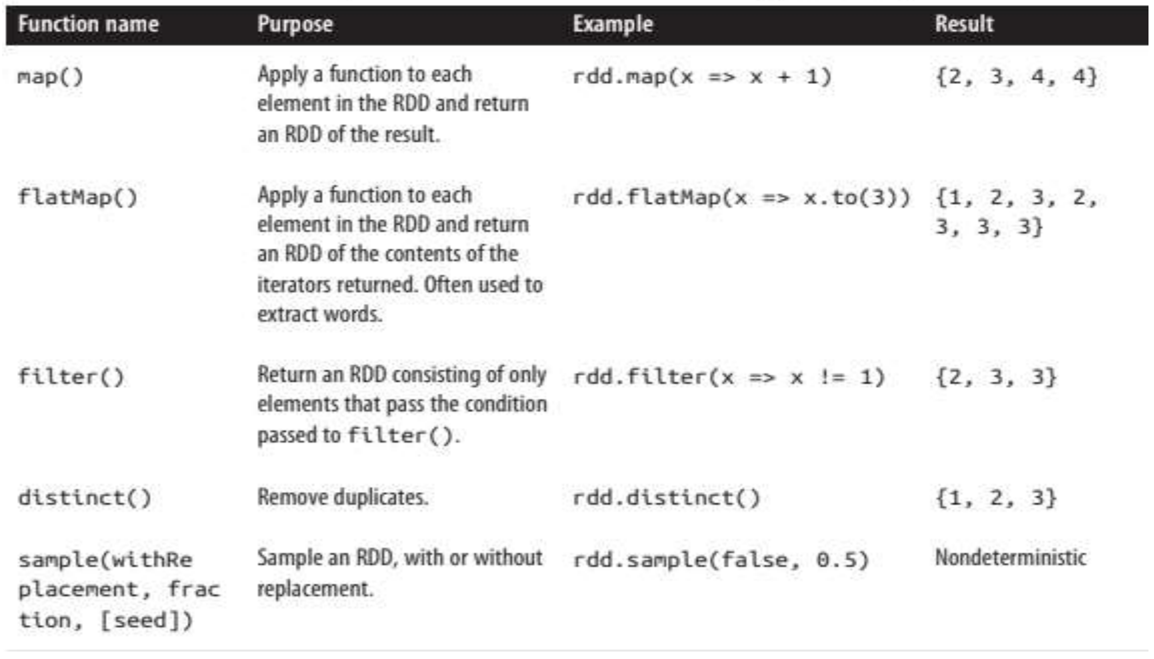
Transformations: They are operations on RDDs that return a new RDD and also produce new RDD from the existing RDDs. They are computed lazily, i.e. only when you use them in an action. Many transformations are element-wise; that is, they work on one element at a time; but this is not true for all transformations.

Actions: They are the operations that return a final value to the driver program or write data to an external storage system. Actions force the evaluation of the transformations required for the RDD they were called on, since they need to actually produce output.

RDD Operations are as listed below:

3.1 RDD Transformation

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}



Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

3.1. 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.

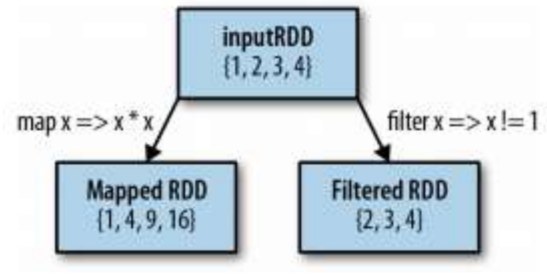
In the map, we have the flexibility that the input and the return type of RDD may differ from each other. For example, we can have input RDD type as String, after applying the map() function the return RDD can be Boolean.

For Eg: we need to square all the values in the list

```
val input = sc.parallelize(List(1, 2, 3, 4))

val result = input.map(x => x * x)

println(result.collect().mkString(","))
```



3.1.2 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.

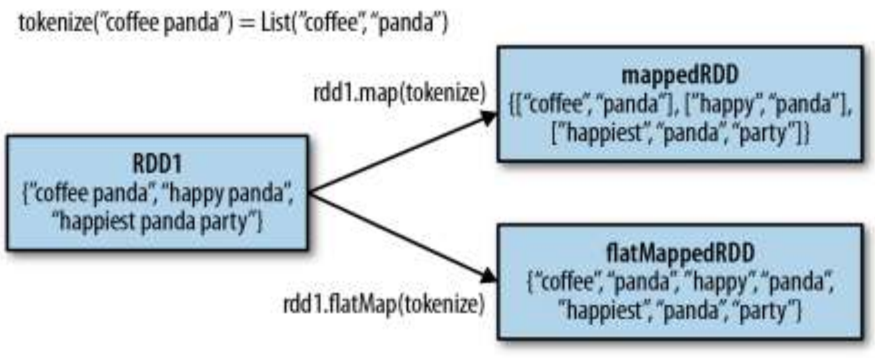
Map and flatMap are similar in the way that they take a line from input RDD and apply a function on that line. The key difference between map() and flatMap() is map() returns only one element, while flatMap() can return a list of elements.

flatMap() example:

```
val data = spark.read.textFile("spark_test.txt").rdd

val flatmapFile = data.flatMap(lines => lines.split(" "))

flatmapFile.foreach(println)
```


tokenize("coffee panda") = List("coffee", "panda")
rdd1.map(tokenize)
mappedRDD
{["coffee", "panda"], ["happy", "panda"], ["happiest", "panda", "party"]}
RDD1
{"coffee panda", "happy panda", "happiest panda party"}
flatMappedRDD
{"coffee", "panda", "happy", "panda", "happiest", "panda", "party"}
rdd1.flatMap(tokenize)

Pictorial representation of difference between Map() and FlatMap()

3.1.3. filter(func)

Spark RDD filter() function returns a new RDD, containing only the elements that meet a predicate. It does not shuffle data from one partition to many partitions.

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 map line into words and then count the word "Spark" using count() Action after filtering lines containing "Spark" from mapFile.

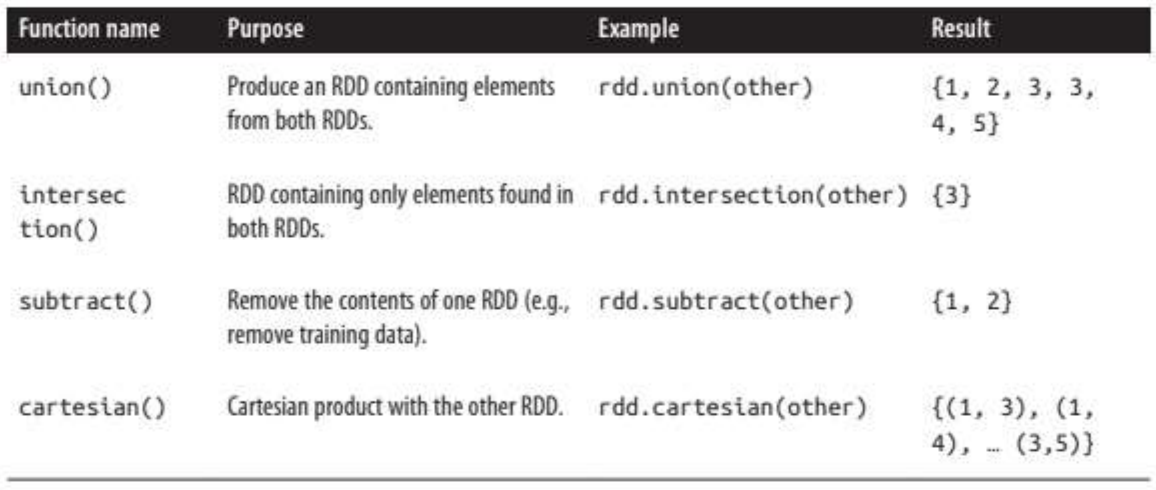
3.1.4. mapPartitions(func)

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.

3.1.5. mapPartitionWithIndex()

It is like mapPartition; Besides mapPartition it provides function with an integer value representing the index of the partition, and the map() is applied on partition index wise one after the other.

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}



Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersec tion()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

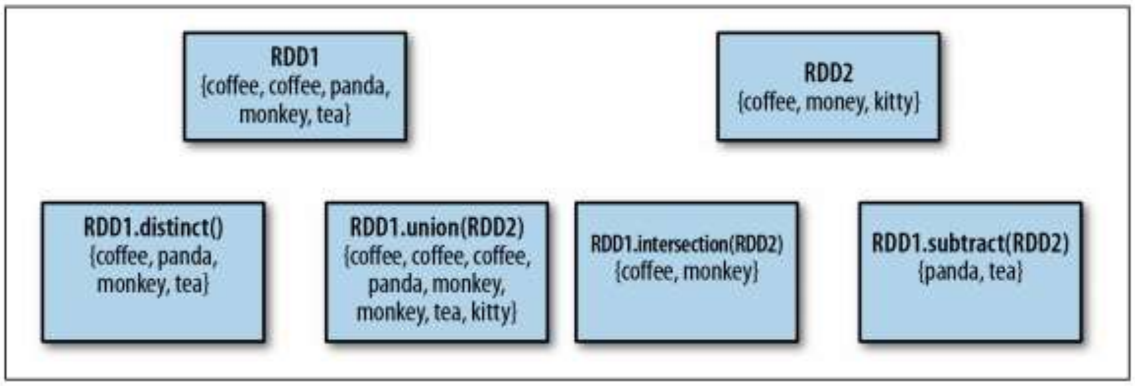


Figure 3-4. Some simple set operations

3.1.6. 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.

Union() example:

```
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)
```

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).

3.1.7. 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.

Intersection() example:

```
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)
```

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

3.1.8. distinct()

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

Distinct() example:

```
val rdd1 =
park.sparkContext.parallelize(Seq((1,"jan",2016),(3,"nov",2014),(16,"feb",2014),(3,"nov",2
014)))

val result = rdd1.distinct()

println(result.collect().mkString(", "))
```

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

3.1.9. 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.

groupByKey() example:

```
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.

3.1.10. 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:

```
val x = sc.parallelize(Array(("a", 1), ("b", 1), ("a", 1),("a", 1), ("b", 1),("b", 1),("b", 1), ("b",
1)))
val y = x.reduceByKey((key, value) => (key + value))
y.collect()
```

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

3.1.11. 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:

```
val data = spark.sparkContext.parallelize(Seq(("maths",52), ("english",75), ("science",82),
("computer",65), ("maths",85)))

val sorted = data.sortByKey()

sorted.foreach(println)
```

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

3.1.12. 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.

Join() example:

```
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(","))
```

3.2. RDD Action

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	4
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0)) ((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return `num` elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9

Some of the actions of Spark are:

3.2.1. count()

Action count() returns the number of elements in RDD.

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 maps line into words and count the word “Spark” using count() Action after filtering lines containing “Spark” from mapFile.

3.2.2 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:

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.

3.2.3. 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.

Take() example:

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 data set defined in the taking argument.

3.2.4. 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:

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)

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

3.2.5. countByValue()

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

countByValue() example:

val data = spark.read.textFile("spark_test.txt").rdd

val result= data.map(line => (line,line.length)).countByValue()

result.foreach(println)

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

3.2.6. 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:

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.

3.2.7. 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:

```
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)
```

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.

3.2.8. 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.

3.2.9. 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:

```
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 foreach() action run a function (println) on each element of the dataset group