# Spark Core

Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters

## RDD (Resilient Distributed Dataset):

* Resilient – if data in memory is lost, it can be recreated
* Distributed – stored in memory across the cluster
* Dataset – initial data can come from a file or be created programmatically

RDD are the fundamental unit of data in spark

Most Spark programming consists of performing operations on RDDs

RDDs are designed to be immutable, which means you cannot specifically modify a particular row in the dataset represented by that RDD. RDD’s can be manipulated by calling specific function to create another RDD. The immutability of RDDs essentially requires an RDD to carry its lineage information that Spark leverages to efficiently provide the fault tolerance capability

Lineage Graph is a dependencies graph in between existing RDD and new RDD

In Spark, all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result

## Distributed in-memory computation

Machine learning algorithms are iterative in nature, meaning they need to go through many iterations to arrive at an optimal state. This is where distributed in-memory computation can help in reducing the completion time from days to hours. Another use case that can hugely benefit from distributed in-memory computation is interactive data mining, where multiple ad hoc queries are performed on the same subset of data

## Partitions:

1. A set of partitions, which are the chunks that make up the entire dataset
2. A set of dependencies on parent RDDs
3. A function for computing all the rows in the data set
4. Metadata about the partitioning scheme (optional)
5. Where the data lives on the cluster (optional); if the data lives on HDFS, then it would be where the block locations are located

The first three pieces of information make up the lineage information, which Spark uses for two purposes. The first one is determining the order of execution of RDDs, and the second one is for failure recovery purposes

## RDD Operations:

They include the ability to perform data transformation, filtering, grouping, joining, aggregation, sorting, and counting. These operations is that they operate at the coarse-grained level, meaning the same operation is applied to many rows, not to any specific row

|  |  |  |
| --- | --- | --- |
| **Type** | **Evaluation** | **Returned Value** |
| Transformation | Lazy | Another RDD |
| Action | Eager | Some result or write result to disk |

## Transformations:

Since RDD are immutable, RDD can be transformed from to another, this is not executed until there is an action performed. Actions are used to display or store the result, causing to run all the dependent RDD lineage in a Lazy execution method i.e RDD’s are not run until there is an action, but that the same time lineage graph is created ready to execute.

Transformation operations are lazily evaluated, meaning Spark will delay the evaluations of the invoked operations until an action is taken

Action operation will trigger the evaluation of all the transformations that preceded it, and it will either return some result to the driver or write data to a storage system, such as HDFS or the local file system.

In short, RDDs are immutable, RDD transformations are lazily evaluated, and RDD actions are eagerly evaluated and trigger the computation of your data processing logic

## Creating RDDs:

There are three ways to create an RDD:

### Creating an RDD from an Object Collection using Parallelize

parallelize an object collection, meaning converting it to a distributed dataset that can be operated in parallel

val stringList = Array("Spark is awesome","Spark is cool")

stringList: Array[String] = Array(Spark is awesome, Spark is cool)

val stringRDD = spark.sparkContext.parallelize(stringList)

stringRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[4] at parallelize at command-1324902525822925:1

stringRDD.getNumPartitions

res30: Int = 8

The stringRDD variable represents an RDD that you can apply transformation or action operations to

### Creating an RDD from a File Data Source

val fileRDD = spark.sparkContext.textFile("file:/databricks/driver/cars.json")

fileRDD: org.apache.spark.rdd.RDD[String] = file:/databricks/driver/cars.json MapPartitionsRDD[7] at textFile at command-1324902525822926:1

val fileRDD2 = sc.textFile("file:/databricks/driver/cars.json")

fileRDD2: org.apache.spark.rdd.RDD[String] = file:/databricks/driver/cars.json MapPartitionsRDD[9] at textFile at command-1324902525822927:1

fileRDD2.getNumPartitions

res29: Int = 2

hdfs:// prefix, it points to a path or a file that resides on HDFS

s3n:// prefix, then it points to a path or a file that resides on AWS S3

If a URI points to a directory, then the textFile method will read all the files in that directory

One important to note for Spark beginners is that the textFile method is lazily evaluated, which means if you made the mistake of specifying a wrong file or path or misspelling a directory name, then this problem would not surface until one of the actions is taken

### Create an RDD is by invoking one of the transformation operations

val fileRDD = sc.textFile("file:/databricks/driver/cars.json")

val fileRDD2 = fileRDD.filter(line => line.contains("USA"))

fileRDD: org.apache.spark.rdd.RDD[String] = file:/databricks/driver/cars.json MapPartitionsRDD[13] at textFile at command-1324902525822929:1

fileRDD2: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[14] at filter at command-1324902525822929:2

RDD2 is created from RDD1 but, both are not executed until there is an action. Even if the input file name is wrong, we will know only after running the action command

## Transformations:

These transformations operate on the dataset being associated with an RDD instance and return a new RDD.

### map(func)

It is used to transform some aspect of the data per row to something else

val stringList = Array("Spark is awesome","Spark is cool")

val stirngRDD = sc.parallelize(stringList)

val capsRDD = stringRDD.map(line => line.toUpperCase)

capsRDD.collect().foreach(println)

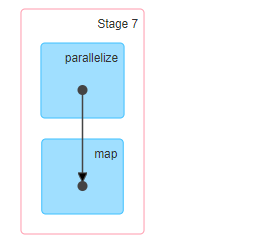
stringList: Array[String] = Array(Spark is awesome, Spark is cool)

stirngRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[23] at parallelize at command-1324902525822932:2

capsRDD: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[24] at map at command-1324902525822932:3

SPARK IS AWESOME

SPARK IS COOL



Using map(func) with user defined functions

def square(number:Int):Int = {

number\*number

}

square: (number: Int)Int

val numbers = Array(1,2,3,4,5,6,7,8,9,10)

val numberSquares = numbers.map(item => square(item))

numbers: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

numberSquares: Array[Int] = Array(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

Using a map Transformation to Convert Text Data into Scala Contact Objects

case class Contact(id:Int, name:String, email:String)

val contactData = Array("1#JohnDoe#jdoe@domain.com","2#MaryJane#mjane@domain.com")

val contactDataRDD = sc.parallelize(contactData)

val contactRDD = contactDataRDD.map(l=> {

val contactArray = l.split("#")

Contact(contactArray(0).toInt, contactArray(1), contactArray(2))

})

defined class Contact

contactData: Array[String] = Array(1#John Doe#jdoe@domain.com, [2#MaryJane#mjane@domain.com](mailto:2#MaryJane))

contactDataRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[27] at parallelize at command-1324902525822934:3

contactRDD: org.apache.spark.rdd.RDD[Contact] = MapPartitionsRDD[28] at map at command-1324902525822934:4

contactRDD.collect().foreach(println)

Contact(1,John Doe,jdoe@domain.com)

Contact(2,MaryJane,mjane@domain.com)

Access individual item

val take = contactRDD.take(1)

take: Array[Contact] = Array(Contact(1,John Doe,jdoe@domain.com))

val takename1 = contactRDD.take(1)(0).name

takename1: String = John Doe

Note about the map transformation is that the **input type** and **the return type** of func **don’t have to be of the same type**

val stringList = Array("Spark is awesome","Spark is cool")

val stringRDD = spark.sparkContext.parallelize(stringList)

val stringLenRDD = stringRDD.map(l => l.length)

stringLenRDD.collect.foreach(println)

stringList: Array[String] = Array(Spark is awesome, Spark is cool)

stringRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[47] at parallelize at command-1324902525822944:2

stringLenRDD: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[48] at map at command-1324902525822944:3

output:

16

13

### flatMap(func)

flatMap flatters the array and creates a single Array of Strings

val strings = Array("Khadhar Basha", "Khalil Basha")

strings: Array[String] = Array(Khadhar Basha, Khalil Basha)

val stringRDD = sc.parallelize(strings)

stringRDD.map(line => line.split(" ")).collect

Array[Array[String]] = Array(Array(Khadhar, Basha), Array(Khalil, Basha))

stringRDD.flatMap(line => line.split(" ")).collect

Array[String] = Array(Khadhar, Basha, Khalil, Basha)

From the above example when applied on String Array

Map 🡺 create Array of (Array of words separated by space)

flatMap 🡺 Flatter the array and give array of string

### filter

filter a dataset down to the rows that meet the conditions defined inside the given func

A simple example is to find out how many lines in the stringRDD contain the word awesome. Another example is to filter a 1TB log file down to only the lines that contain the word Exception

The filter method is a higher-order method that takes a Boolean function as input and applies it to each element in the source RDD to create a new RDD

val strings = Array("Khadhar Basha", "Khalil Basha")

val stringRDD = sc.parallelize(strings)

val shortRDD = stringRDD.filter(x=> x.contains("Khadhar"))

shortRDD.collect

res4: Array[String] = Array(Khadhar Basha)

### union(otherRDD)

Unlike previous transformations that take a function as an argument, a union transformation takes another RDD as an argument, and it will return an RDD that combines the rows from both RDDs. This is useful for situations when there is a need to append some rows to an existing RDD. This transformation does not remove duplicate rows of the resulting RDD.

val rdd1 = spark.sparkContext.parallelize(Array(1,2,3,4,5))

val rdd2 = spark.sparkContext.parallelize(Array(1,6,7,8))

val rdd3 = rdd1.union(rdd2)

rdd3.collect()

Array[Int] = Array(1, 2, 3, 4, 5, 1, 6, 7, 8)

### intersection(otherRDD)

If there were two RDDs and there is a need to find out which rows exist in both of them, then this is the right transformation to use. The way this transformation figures out which rows exist in both RDDs is by comparing their hash codes. This transformation guarantees the returned RDD will not contain any duplicate rows

val rdd1 = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8))

val rdd2 = spark.sparkContext.parallelize(Array(1,6,7,8,9,10,12))

val rdd3 = rdd1.intersection(rdd2)

rdd3.collect()

Array[Int] = Array(8, 1, 6, 7)

### substract(otherRDD)

Removes the rdd2 from rdd1

val rdd1 = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8))

val rdd2 = spark.sparkContext.parallelize(Array(1,6,7,8,9,10,12))

val rdd3 = rdd1.subtract(rdd2)

rdd3.collect()

Array[Int] = Array(2, 3, 4, 5)

### distinct()

The distinct transformation represents another flavor of transformation where it doesn’t take any function or another RDD as an input parameter. Instead, it is a directive to the source RDD to remove any duplicate rows

To remove duplicate rows in an RDD, it simply computes the hash code of each row and compares them to determine whether two rows are identical.

val rdd1 = spark.sparkContext.parallelize(Array(1,2,3,4,5,6,7,8))

val rdd2 = spark.sparkContext.parallelize(Array(1,6,7,8,9,10,12))

val rdd3 = rdd1.union(rdd2)

rdd3.collect()

res9: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 1, 6, 7, 8, 9, 10, 12)

rdd3.distinct.collect()

res10: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12)

### MapPartitions:

The higher-order mapPartitions method allows you to process data at a partition level. Instead of passing one element at a time to its input function, mapPartitions passes a partition in the form of an iterator. The mapPartitions method returns new RDD formed by applying a user-specified function to each partition of the source RDD.

val strings = Array("Khadhar Basha", "Khalil Basha")

val stringRDD = sc.parallelize(strings)

val shortRDD = stringRDD.filter(x=> x.contains("Khadhar"))

val mapDD = stringRDD.mapPartitions(x=>x.filter(x =>x.contains("Khadhar")))

### GroupBy :

The higher order groupBy method groups the elements of an RDD according to a user specified criteria. It takes as input a function that generates a key for each element in the source RDD.

case class Contact(id:Int, name:String, email:String, age:Int,city:String)

val contactData = Array("1#John Doe#jdoe@domain.com#28#chennai","2#MaryJane#mjane@domain.com#30#nellore","3#Jackson#mjane@domain.com#40#chennai")

val contactDataRDD = sc.parallelize(contactData)

val contactRDD = contactDataRDD.map(l=> {

val contactArray = l.split("#")

Contact(contactArray(0).toInt, contactArray(1), contactArray(2), contactArray(3).toInt, contactArray(4))

})

val groupByZip = contactRDD.groupBy { a => a.city}

groupByZip.collect

groupByZip: org.apache.spark.rdd.RDD[(String, Iterable[Contact])] = ShuffledRDD[58] at groupBy at command-4319519382734740:1 res19: Array[(String, Iterable[Contact])] = Array((chennai,CompactBuffer(Contact(1,John Doe,jdoe@domain.com,28,chennai), Contact(3,Jackson,mjane@domain.com,40,chennai))), (nellore,CompactBuffer(Contact(2,MaryJane,mjane@domain.com,30,nellore))))

### ReduceByKey :

The higher-order reduceByKey method takes an associative binary operator as input and reduces values with the same key to a single value using the specified binary operator.

val pairRdd = sc.parallelize(List(("a", 1), ("b",2), ("c",3), ("a", 11), ("b",22), ("a",111)))

Array[(String, Int)] = Array((a,1), (b,2), (c,3), (a,11), (b,22), (a,111))

val sumByKeyRdd = pairRdd.reduceByKey((x,y) => x+y)

val minByKeyRdd = pairRdd.reduceByKey((x,y) => if (x < y) x else y)

sumByKeyRdd.collect

Array[(String, Int)] = Array((a,123), (b,24), (c,3))

minByKeyRdd.collect

Array[(String, Int)] = Array((a,1), (b,2), (c,3))

### Partition Handling:

#### Coalesce:

The coalesce method reduces the number of partitions in an RDD. It takes an integer input and returns a new RDD with the specified number of partitions.

Coalesce uses existing partitions to minimize the amount of data that's shuffled.

Coalesce results in partitions with different amounts of data (sometimes partitions that have much different sizes) and repartition results in roughly equal sized partitions.

val numbers = sc.parallelize((1 to 50).toList,5)

numbers.partitions.size

res21: Int = 5

numbers.glom().collect

res25: Array[Array[Int]] = Array(Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10), Array(11, 12, 13, 14, 15, 16, 17, 18, 19, 20), Array(21, 22, 23, 24, 25, 26, 27, 28, 29, 30), Array(31, 32, 33, 34, 35, 36, 37, 38, 39, 40), Array(41, 42, 43, 44, 45, 46, 47, 48, 49, 50))

val numbersWithTwoPartition = numbers.coalesce(2)

res26: Array[Array[Int]] = Array(Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20), Array(21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50))

#### Repartition :

The repartition method takes an integer as input and returns an RDD with specified number of partitions. It is useful for increasing parallelism. It redistributes data, so it is an expensive operation.

val numbersWithTwoPartition = numbers.repartition(3)

numbersWithTwoPartition.glom().collect

Array[Array[Int]] = Array(Array(3, 6, 9, 12, 15, 18, 23, 26, 29, 32, 35, 38, 42, 45, 48), Array(1, 4, 7, 10, 13, 16, 19, 21, 24, 27, 30, 33, 36, 39, 43, 46, 49), Array(2, 5, 8, 11, 14, 17, 20, 22, 25, 28, 31, 34, 37, 40, 41, 44, 47, 50))

NOTE: EQUAL partitions

## Actions:

### collect()

It collects all the rows from each of the partitions in an RDD and brings them over to the driver program. If your RDD contains 100 million rows, then it is not a good idea to invoke the collect action because the driver program most likely doesn’t have sufficient memory to hold all those rows. As a result, the driver will most likely run into an out-ofmemory error and your Spark application or shell will die. This action is typically used once the RDD is filtered down to a smaller size that can fit the memory size of the driver program

val numberRDD = spark.sparkContext.parallelize(List(1,2,3,4,5,6,7,8,9,10), 2)

numberRDD.collect()

Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

### Count():

It returns the number of rows in an RDD by getting the count from all partitions and finally sums them up

val numberRDD = spark.sparkContext.parallelize(List(1,2,3,4,5,6,7,8,9,10), 2)

numberRDD.count()

Long = 10

### first()

This action returns the first row in the RDD

scala> warn.first()

res4: String = 2020-07-18 21:45:27,770

### take(n)

This action returns the first n rows in the RDD by collecting rows from the first partition and then moves to the next partition until the number of rows matches n or the last partition is reached. If n is larger than the number of rows in the dataset, then it will return all the rows. take(1) is equivalent to the first() action

scala> warn.take(5).foreach(println)

2020-07-18 21:45:27,770 WARN org.apache.hadoop.util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

2020-07-18 22:23:15,345 WARN org.apache.hadoop.hdfs.server.datanode.DataNode: IOException in offerService

2020-07-18 22:24:05,677 WARN org.apache.hadoop.util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

2020-07-18 22:29:46,732 WARN org.apache.hadoop.hdfs.server.datanode.DataNode: IOException in offerService

2020-07-18 22:46:51,102 WARN org.apache.hadoop.util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

### reduce(func)

It reduces all the rows in the dataset to a single value using the provided function. A common use case is to perform a sum of all the integers in the dataset

There are two rules that the provided functions must follow

* It must be a binary operator, meaning it must take two arguments of the same type, and it produces an output of the same type
* It must follow the commutative and associative properties in order for the result to be computed correctly in a parallel manner

val numbersRdd = sc.parallelize(List(20, 50, 30, 10))

val sum = numbersRdd.reduce ((x, y) => x + y)

**sum: Int = 110**

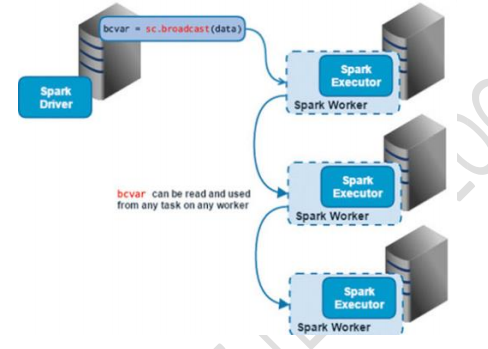
val multi = numbersRdd.reduce ((x, y) => x \* y)

**multi: Int = 300000**

## Shared Variables:

### Broadcast Variables:

Broadcast variables allow Spark developers to keep a secured read-only variable cached on different nodes, other than merely shipping a copy of it with the needed tasks. For an instance, they can be used to give a node a copy of a large input dataset without having to waste time with network transfer I/O. Spark has the ability to distribute broadcast variables using various broadcast algorithms which will in turn largely reduce the cost of communication.



Below program has 8 partition data, the broadcasted variable is shipped from driver program to node program to make it locallly for each task and the operation is executed

val sal = sc.parallelize(List(1000,2000,10000,20000,30000,34000),8)

val bonus = sc.broadcast(20)

val added = sal.map(x=>x+bonus.value)

ACCUMULATORS

Accumulators are variables which may be added to through associated operations. There are many uses for accumulators including implementing counters or sums. Spark supports the accumulation of numeric types. If there is a particular name for an accumulator in code, it is usually displayed in the Spark UI, which will be useful in understanding the running stage progress.



### Accumulators:

## RDD Caching Methods:

The RDD class provides two methods to cache an RDD: cache and persistCache :

### Cache():

The cache method stores an RDD in the memory of the executors across a cluster. It essentially materializes an RDD in memory.

errorsAndWarnings.cache()

### Persist/UnPersist :

The persist method is a generic version of the cache method. It allows an RDD to be stored in memory, disk, or both.

It optionally takes a storage level as an input parameter. If persist is called without any parameter, its behaviour is identical to that of the cache method.

errorsAndWarnings.persist()

errorsAndWarnings.unpersist()

## Working with key/value Pari RDD

### Creating Key/Value Pair RDD:

The simplest way to create a pair RDD is to arrange the data of each row into two parts: key and value. Using Tuple

scala> val team = List("Khadhar Basha", "Farheen", "Haneef", "Sheema", "Aleem", "Khalil Basha")

team: List[String] = List(Khadhar Basha, Farheen, Haneef, Sheema, Aleem, Khalil Basha)

scala> val teamRDD = sc.parallelize(team)

teamRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[2] at parallelize at <console>:26

scala> val teamLength = teamRDD.map(x=> (x.length, x))

teamLength: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[3] at map at <console>:25

scala> teamLength.collect

res2: Array[(Int, String)] = Array((13,Khadhar Basha), (7,Farheen), (6,Haneef), (6,Sheema), (5,Aleem), (12,Khalil Basha))

## Key/Value Pair RDD Transformations:

### groupByKey([numTasks])

It will group all the rows with the same key into a single row.

Essentially the number of rows in the returned RDD will be the same as the number of unique keys in the parent RDD.  Each row in the returned RDD contains a unique key and a list of values of that same key

Groups all the values of the same key together.

For a dataset of (K,V) pairs, the returned RDD has the type (K, Iterable)

val groceries = sc.parallelize(List(("veg","carrot"),("non-veg","mutton"),("fruit","papaya"),("veg","beans"),("veg","chicken"),("veg","drumstick"),("fruit","apple"),("veg","carrot")))

val categories = groceries.groupByKey()

categories.collect().foreach(println)

(veg,CompactBuffer(carrot, beans, chicken, drumstick, carrot))

(non-veg,CompactBuffer(mutton))

(fruit,CompactBuffer(papaya, apple))

### reduceByKey(func, [numTasks])

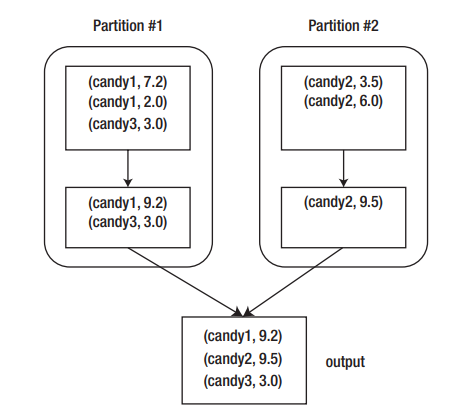
This transformation is used to reduce all the values of the same key to a single value

The process is carried out in two steps

* Group the values of the same key together
* Apply the given reduce function to the list of values of each key.

The first level is at each individual partition, and the second level is across all the partitions.

For a dataset of (K,V) pairs, the returned RDD has the type of (K, V).



val pairRdd = sc.parallelize(List(("a", 1), ("b",2), ("c",3), ("a", 11), ("b",22), ("a",111)))

val sumByKeyRdd = pairRdd.reduceByKey((x,y) => x+y)

sumByKeyRdd.collect()

Array[(String, Int)] = Array((a,123), (b,24), (c,3)

### sortByKey([ascending], [numTasks])

Sorts the rows according to the keys. By default, the keys are sorted in ascending order

val summaryByTotal = sumByKeyRdd.map(t=> (t.\_2, t.\_1)).sortByKey()

summaryByTotal.collect()

Array[(String, Int)] = Array((a,123), (b,24), (c,3)) 🡪 Unsorted

Array[(Int, String)] = Array((3,c), (24,b), (123,a)) 🡪 Sorted

### join(otherRDD, [numTasks])

Joins the rows in both RDDs by matching their keys.

Each row of the returned RDD contains a tuple where the first element is the key and the second element is another tuple containing the values from both RDDs

val pairRdd1 = sc.parallelize(List(("a", 1), ("b",2), ("c",3)))

val pairRdd2 = sc.parallelize(List(("b", "second"), ("c","third"), ("d","fourth")))

val joinRdd = pairRdd1.join(pairRdd2)

joinRdd.collect().foreach(println)

(b,(2,second)) (c,(3,third))

val joinRdd = pairRdd1.rightOuterJoin(pairRdd2)

joinRdd.collect().foreach(println)

(b,(Some(2),second)) (c,(Some(3),third)) (d,(None,fourth))

val joinRdd = pairRdd1.leftOuterJoin(pairRdd2)

joinRdd.collect().foreach(println)

(a,(1,None)) (b,(2,Some(second))) (c,(3,Some(third)))

val joinRdd = pairRdd1.fullOuterJoin(pairRdd2)

joinRdd.collect().foreach(println)

(a,(Some(1),None)) (b,(Some(2),Some(second))) (c,(Some(3),Some(third))) (d,(None,Some(fourth)))

## Key/Value Pair RDD Actions

### Lookup

The lookup method takes a key as input and returns a sequence of all the values mapped to that key in the source RDD

val pairRdd = sc.parallelize(List(("IT",1 ), ("ACC", 2), ("MKT", 3), ("IT", 11), ("ACC", 22), ("IT", 10)))

val values = pairRdd.lookup("IT")

values

values: Seq[Int] = WrappedArray(1, 11, 10)

### countByKey()

The countByKey method counts the occurrences of each unique key in the source RDD. It returns a Map of key-count pairs

Reports only the number of values with the same key for each key to the driver

val pairRdd = sc.parallelize(List(("IT", 1), ("ACC", 2), ("MKT", 3), ("IT", 11), ("ACC", 22), ("IT", 1)))

pairRdd.countByKey()

scala.collection.Map[String,Long] = Map(ACC -> 2, IT -> 3, MKT -> 1)