SPARK NOTES

By. Mr. Gopal Krishna

Sr.Hadoop Architect

CCA 175 – Spark and Hadoop Developer Certified Consultant

****

Spark is an in-memory cluster computing framework for processing and analyzing large amounts of data.

It provides a simple programming interface, which enables an application developer to easily use the CPU, memory, and storage resources across a cluster of servers for processing large datasets

Key Features of Spark:

1. **Easy to Use**
2. **Fast**
3. **General Purpose**
4. **Scalable**

Fault Tolerant

Easy to Use:

- Spark provides a simpler programming model than that provided by Map Reduce. Developing a distributed data processing application with Spark is a lot easier than developing the same application with Map Reduce.
- Hadoop Map Reduce provides only two operations for processing the data like **"Map" & "Reduce"** , where as Spark comes with **80 plus data processing operations** to work with big data applications.
- Hadoop MapReduce requires every problem to be broken down into a sequence of map and reduce jobs and so its very difficult to implement some complex algorithms only with Map Reduce where as

For Spark its much easier to implement the to do the complex data processing with its **in-memory processing**

- Spark development code is very concise compared to Hadoop Map Reduce. What Map Reduce requires 50 lines of code for a specific application development, Spark only requires less than 10 lines of code.
- 5 to 10 times of more productivity with Spark compared to Map Reduce.

Fast:

- If data fits in the memory, Spark is 100 times faster than Map Reduce..Even if the data is fits in Memory also, Spark is 10 times faster than Map reduce.
- Spark is faster than Hadoop Map Reduce for two reasons.
  1. It allows in-memory cluster computing
  2. It implements an advanced execution engine.
- The sequential read throughput when reading data from memory compared to reading data from a hard disk is 100 times greater i.e. When the data volume is less, it may not be that much significant where as when we are reading/writing Terabytes of data ..it makes lot of difference in Network I/O.
- **Spark does not automatically cache input data in memory. A common misconception is that Spark cannot be used if input data does not fit in memory. It is not true. Spark can process terabytes of data on a cluster that may have only 100 GB total cluster memory.**
- It is up to an application to decide what data should be cached and at what point in a data processing pipeline that data should be cached.

General Purpose:

- Spark provides a unified integrated platform for different types of data processing jobs. It can be used for:

  1. Batch processing
  2. Interactive Processing
  3. Stream Processing
  4. Machine Learning
  5. Graph Computing

- With Map Reduce we can only perform the batch processing, and for driving Stream Processing, Graph processing developer has to use different frameworks.
- Using different frameworks for different types of data processing jobs creates many challenges.

    1. A developer has to learn multiple frameworks, each of which has a different interface. This reduces developer productivity
    2. Each framework may operate in a silo. Therefore, data may have to be copied to multiple places.

- Spark comes pre-packaged with an integrated set of libraries for batch processing, interactive analysis, stream processing, machine learning, and graph computing.
- With Spark, you can use a single framework to build a data processing pipeline that involves different types of data processing tasks. There is no need to learn multiple frameworks or deploy separate clusters for different types of data processing jobs

Scalable:

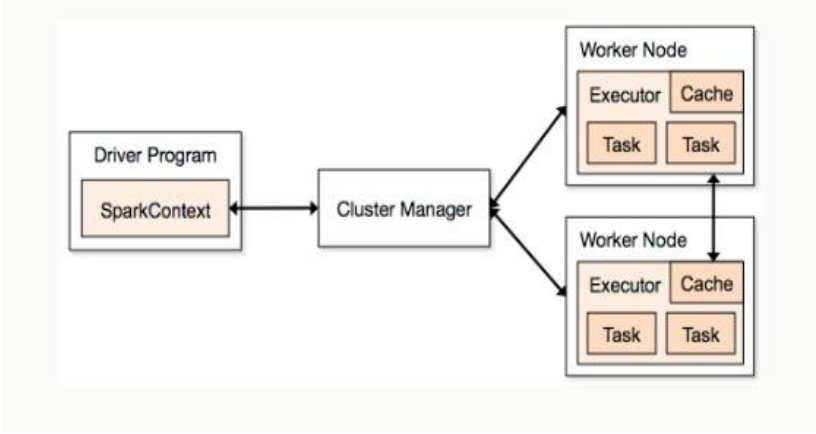
- Spark is scalable. The data processing capacity of a Spark cluster can be increased by just adding more nodes to a cluster.
- You can start with a small cluster, and as your dataset grows, you can add more computing capacity. Thus, Spark allows you to scale economically.
- In addition, Spark makes this feature automatically available to an application. No code change is required when you add a node to a Spark cluster

Fault Tolerant:

- Spark is fault tolerant. In a cluster of a few hundred nodes, the probability of a node failing on any given day is high. The hard disk may crash or some other hardware problem may make a node unusable.
- Spark automatically handles the failure of a node in a cluster. Failure of a node may degrade performance, but will not crash an application.
- Since Spark automatically handles node failures, an application developer does not have to handle such failures in his application. It simplifies application code

Spark Architecture

Spark Architecture



-

- **Spark Context** – Is the coordinator object which will run different spark applications.
- SparkContext is a class defined in the Spark library. It is the main entry point into the Spark library
- Spark Context will run in a program called **'Driver Program'** – the main program in spark.
- A Spark application must create an instance of the SparkContext class. Currently, an application can have only one active instance of

SparkContext. An instance of the SparkContext class can be created as below:

```
val sc = new SparkContext()
```

NOTE: In this case, SparkContext gets configuration settings such as the address of the Spark master, application name, and other settings from system properties

If we want to mention the same programmatically:

```
val config = new SparkConf()
             .setMaster("spark://host:port”)
             .setAppName("big app")


val sc = new SparkContext(config)
```

Cluster Manager

- Cluster Manager will allocate resources across applications of cluster
- to run on a cluster, the **Spark Context** can connect to several types of ***cluster managers*** (either Spark’s own standalone cluster manager or YARN).

Executors

Once resources connected, Spark acquires ***executors*** on nodes in the cluster, which are processes that run computations and store data for your application

- Next, cluster manager sends your **application code** (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends ***tasks*** for the executors to run.

DIFFERENT CLUSTER MANAGERS IN - SPARK ARCHITECTURE

- As part of Spark Runtime architecture , there are 3 cluster managers

    - **Standalone Cluster Manager (default)**
    - **Hadoop YARN**
    - **Apache Mesos**

Apache Mesos

- Apache Mesos is a general-purpose cluster manager that can run both analytics workloads and long-running services (e.g., web applications or key/value stores) on a cluster.
- To use Spark on Mesos, **pass a *mesos://* URI to spark-submit:**

  spark-submit --master mesos://masternode:5050 yourapp

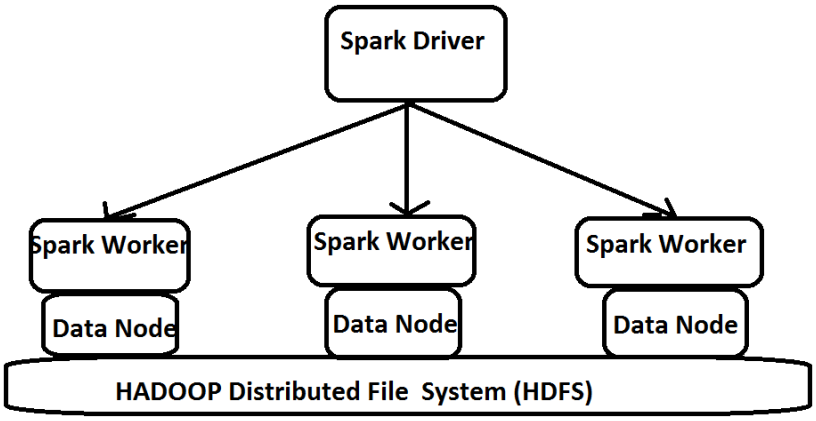
Hadoop YARN

- YARN is a cluster manager introduced in Hadoop 2.0 that allows diverse data processing frameworks to run on a shared resource pool, and is typically installed on the same nodes as the Hadoop filesystem (HDFS).

- Running Spark on YARN in these environments is useful because it lets Spark access HDFS data quickly, on the same nodes where the data is stored. To use Spark on Hadoop YARN use the below command

spark-submit --master yarn yourapp

Spark Architecture



Resilient Distributed Datasets (RDD)

RDD represents a collection of partitioned data elements that can be operated on in parallel. It is the primary data abstraction mechanism in Spark.

It is defined as an abstract class in the Spark library. Conceptually, RDD is similar to a Scala collection, except that it represents a distributed dataset and it supports lazy operations

key characteristics of an RDD:

Immutable:

- ✓ An RDD is an immutable data structure.
- ✓ Once created, it cannot be modified in-place. Basically, an operation that modifies an RDD returns a new RDD.

Partitioned

- ✓ Data represented by an RDD is split into partitions.
- ✓ These partitions are generally distributed across a cluster of nodes.
  - ✓ when Spark is running on a single machine, all the partitions are on that machine

RDD OPERATIONS

Spark applications process data using the methods defined in the RDD class or classes derived from it. These methods are also referred to as **operations**.

RDD operations can be categorized into two types:

- ⇨ **Transformations**
- ⇨ **Actions**

Transformations

- ✓ A transformation method of an RDD creates a new RDD by performing a computation on the source RDD
- ✓ RDD transformations are conceptually similar to Scala collection methods.
- ✓ The key difference is that the Scala collection methods operate on data that can fit in the memory of a single machine, whereas RDD methods can operate on data distributed across a cluster of nodes
- ✓ RDD transformations are lazy, whereas Scala collection methods are strict.

Map

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

```
gopal@ubuntu:~$ hadoop fs -cat /SparkInputDir/Input.log
hadoop is one of the bigdata tool
bigdata is not only for hadoop
hadoop is ment for bigdata storage and processing
hadoop is good for analytics also
thanks hadoop
gopal@ubuntu:~$
```

Example Code:

```
scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")

scala> val fileLength = file.map(l => l.length)

scala> fileLength.collect
```

OUTPUT:

```
res0: Array[Int] = Array(33, 30, 49, 33, 13)
```

filter

- ✓ 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.
- ✓ A Boolean function takes an input and returns true or false.
- ✓ The filter method returns a new RDD formed by selecting only those elements for which the input Boolean function returned true.
- ✓ Thus, the new RDD contains a subset of the elements in the original RDD.

Example Code:

```
scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")

scala> val fileLength = file.filter(l => l.length >= 33)

scala> fileLength.collect
```

OUTPUT:

```
Array[String] = Array(hadoop is one of the bigdata tool, hadoop is ment for bigdata storage and processing, hadoop is good for analytics also)

scala> val fileLength = file.filter(l => l.length == 30)
```

OUTPUT:

Array[String] = Array(bigdata is not only for hadoop)

scala> val fileLength = file.filter(l => l.length != 33)

OUTPUT:
Array[String] = Array(bigdata is not only for hadoop, hadoop is ment for bigdata storage and processing, thanks hadoop)

scala> val fileLength = file.filter(l => l.contains("analytics"))

OUTPUT:
Array[String] = Array(hadoop is good for analytics also)

flatMap

- ✓ The **flatMap** method is a higher-order method that takes an input function, which returns a sequence for each input element passed to it.
- ✓ The **flatMap** method returns a new RDD formed by flattening this collection of sequence

Example Code:

scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")

scala> val fileWords = file.flatMap(l => l.split(" "))

scala> fileWords.collect

OUTPUT:

res7: Array[String] = Array(hadoop, is, one, of, the, bigdata, tool, bigdata, is, not, only, for, hadoop, hadoop, is, ment, for, bigdata, storage, and, processing, hadoop, is, good, for, analytics, also, thanks, hadoop)

scala> println(fileWords.collect().mkString("\t"))

OUTPUT:
16/09/09 02:36:17 INFO DAGScheduler: Job 5 finished: collect at <console>:26, took 0.273978 s
hadoop is one of the bigdata tool bigdata is not only for hadoop hadoop is ment for bigdata storage and processing hadoop is good for analytics also thanks hadoop

scala> println(fileWords.collect().mkString("\n"))

hadoop
is
one
of
the
bigdata
tool
bigdata
is
not
only
for

hadoop
hadoop
is
ment
for
bigdata
storage
and
processing
hadoop
is
good

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 input function to the mapPartitions method takes an **iterator as input** and returns another **iterator as output**
- ✓ The mapPartitions method returns new RDD formed by applying a user-specified function to each partition of the source RDD

Example Code:

```
scala> val file = sc.textFile("hdfs://localhost:54310/SparkInputDir/Input.log")

scala> val fileMapPar = file.mapPartitions(i => i.map( l => l.length))
fileMapPar: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[19] at mapPartitions at
<console>:23

scala> fileMapPar.collect
```

OUTPUT:

```
res15: Array[Int] = Array(33, 30, 49, 33, 13)
```

Union:

The union method takes an RDD as input and returns a new RDD that contains the union of the elements in the source RDD and the RDD passed to it as an input

INPUT DATA:

```
gopal@ubuntu:~$ pwd
/home/gopal
gopal@ubuntu:~$ cat File1.txt
THIS IS LINE1
THIS IS LINE2
gopal@ubuntu:~$ cat File2.txt
THIS IS LINE3
THIS IS LINE4
gopal@ubuntu:~$
```

scala> val file1 = sc.textFile("/home/gopal/File1.txt")

scala> val file2 = sc.textFile("/home/gopal/File2.txt")

scala> val unionFile = file1.union(file2)

unionFile: org.apache.spark.rdd.RDD[String] = UnionRDD[24] at union at <console>:25

scala> unionFile.collect

OUTPUT:
Array[String] = Array(THIS IS LINE1, THIS IS LINE2, THIS IS LINE3, THIS IS LINE4)

Example Code 2:

scala> val tech = sc.parallelize(List("Spark","Scala","Java"))

scala> val technew = sc.parallelize(List("Perl","Scala","Phython"))

scala> val unionTech = tech.union(technew)

scala> unionTech.collect

OUTPUT:

Array[String] = Array(Spark, Scala, Java, Perl, Scala, Phython)

Intersection:

The intersection method takes an RDD as input and returns a new RDD that contains the **intersection of the elements in the source RDD and the RDD** passed to it as an input.

INPUT DATA:

gopal@ubuntu:~$ cat File1.txt
THIS IS LINE1
THIS IS LINE2
gopal@ubuntu:~$ cat File2.txt
THIS IS LINE2
THIS IS LINE4
gopal@ubuntu:~$

Example Code:

scala> val file1 = sc.textFile("/home/gopal/File1.txt")

scala> val file2 = sc.textFile("/home/gopal/File2.txt")

scala> val intersectFile = file1.intersection(file2)

scala> intersectFile.collect

OUTPUT:

Array[String] = Array(THIS IS LINE2)

Example Code 2:

```
scala> val tech = sc.parallelize(List("Spark","Scala","Java"))

scala> val technew = sc.parallelize(List("Perl","Scala","Phython"))

scala> val interTech = tech.intersection(technew)

scala> interTech.collect
```

OUTPUT:

```
Array[String] = Array(Scala)
```

Subtract:

The subtract method takes an RDD as input and returns a new **RDD that contains elements in the source RDD but not in the input RDD**.

```
scala> val file1 = sc.textFile("/home/gopal/File1.txt")

scala> val file2 = sc.textFile("/home/gopal/File2.txt")

scala> val subtractFile = file1.subtract(file2)

scala> subtractFile.collect
```

OUTPUT:

```
Array[String] = Array(THIS IS LINE1)
```

Example Code 2:

```
scala> val tech = sc.parallelize(List("Spark","Scala","Java"))

scala> val technew = sc.parallelize(List("Perl","Scala","Phython"))

scala> val interTech = tech.subtract(technew)

scala> interTech.collect
```

OUTPUT:

```
Array[String] = Array(Java, Spark)
```

Parallelize: Parallelized collections are created by calling **SparkContext's parallelize** method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be **operated on in parallel**. Below is an example, to create a parallelized collection holding the numbers 1 to 5:

Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase,Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat

scala> val inputdata = Array(1, 2, 3, 4, 5)
inputdata: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(inputdata)
distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[49] at parallelize at <console>:23

scala> distData.collect

OUTPUT:
Array[Int] = Array(1, 2, 3, 4, 5)

Distinct:

The distinct method of an RDD returns a new RDD containing the distinct elements in the source RDD

Example Code:

scala> val tech = sc.parallelize(List("Spark","Spark","Spark","Scala","Java","Scala"))

scala> val distTech = tech.distinct

scala> distTech.collect

OUTPUT:

Array[String] = Array(Java, Scala, Spark)

Cartesian:

- ✓ The cartesian method of an RDD takes an RDD as input and returns an RDD containing the Cartesian product of all the elements in both RDDs.
- ✓ It returns an RDD of ordered pairs, in which the first element comes from the source RDD and the second element is from the input RDD.
- ✓ The number of elements in the returned RDD is equal to the product of the source and input RDD lengths
- ✓ This method is similar to a cross join operation in SQL.

Example Code:

scala> val ratings = sc.parallelize(List(1,2,3))
scala> val tech = sc.parallelize(List("Spark","Scala","Java"))
scala> val cartRatTech = ratings.cartesian(tech)
scala> cartRatTech.collect

OUTPUT:
res0: Array[(Int, String)] = Array((1,Spark), (1,Scala), (1,Java), (2,Spark), (3,Spark), (2,Scala), (2,Java), (3,Scala), (3,Java))

```
scala> val cartRatTech = ratings.cartesian(ratings)
scala> cartRatTech.collect

OUTPUT:
res1: Array[(Int, Int)] = Array((1,1), (1,2), (1,3), (2,1), (3,1), (2,2), (2,3), (3,2), (3,3))

scala> val cartRatTech = tech.cartesian(tech)
scala> cartRatTech.collect

OUTPUT:
res2: Array[(String, String)] = Array((Spark,Spark), (Spark,Scala), (Spark,Java),
(Scala,Spark), (Java,Spark), (Scala,Scala), (Scala,Java), (Java,Scala), (Java,Java))
```

ZIP:

- ✓ The **zip** method takes an RDD as input and returns an RDD of pairs, where the first element in a pair is from the source RDD and second element is from the input RDD.
- ✓ Unlike the **cartesian method**, the RDD returned by zip has the same number of elements as the source RDD.
- ✓ **Both the source RDD and the input RDD must have the same length**. In addition, both RDDs are assumed to have **same number of partitions** and same number of elements in each partition.

Example Code:

```
scala> val ratings = sc.parallelize(List(1,2,3))
scala> val tech = sc.parallelize(List("Spark","Scala","Java"))
scala> val zipRatTech = ratings.zip(tech)
scala> zipRatTech.collect

OUTPUT:
Array[(Int, String)] = Array((1,Spark), (2,Scala), (3,Java))

scala> val zipRatTech = ratings.zip(ratings)
scala> zipRatTech.collect
OUTPUT:
Array[(Int, Int)] = Array((1,1), (2,2), (3,3))

scala> val zipRatTech = tech.zip(tech)
scala> zipRatTech.collect

OUTPUT:
Array[(String, String)] = Array((Spark,Spark), (Scala,Scala), (Java,Java))
```

ZipWithIndex:

The **zipWithIndex** method zips the elements of the source RDD with their indices and returns an RDD of pairs:

Example Code:

```
scala> val names = sc.parallelize(List("AAA","BBB","CCC","DDD"))
scala> val zipWithIndexNames = names.zipWithIndex
scala> zipWithIndexNames.collect
```

OUTPUT:
Array[(String, Long)] = Array((AAA,0), (BBB,1), (CCC,2), (DDD,3))

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.
- ✓ It applies this function to all the elements in the source RDD and returns an RDD of pairs.
- ✓ In each returned pair, the first item is a key and the second item is a collection of the elements mapped to that key by the input function to the groupBy method.

Example Code:

```
scala> val names = sc.textFile("/home/gopal/names.csv");
scala> val namesRows = names.map( l => l.split(","))
scala> val yearName = namesRows.map( c => ( c(1),c(2)) )
scala> yearName.groupByKey.collect
```

INPUTDATA:
gopal@ubuntu:~$ cat names.csv
2013,**Raj,India**,Male,25
2014,**Ram,China**,Male,34
2013,**Raj,India**,Male,25
2013,**Ramya,USA**,Female,34
2014,Rajesh,UK,Male,25
2014,Rajesh,AUS,Male,25
2014,Rajesh,UK,Female,34
gopal@ubuntu:~$

OUTPUT:
Array[(String, Iterable[String])] = Array((Ram,CompactBuffer(China)), (Ramya,CompactBuffer(USA)), (Raj,CompactBuffer(India, India)), (Rajesh,CompactBuffer(UK, AUS, UK)))

sortBy:

- ✓ The higher-order **sortBy** method returns an RDD with sorted elements from the source RDD.
- ✓ It takes two input parameters. The first input is a function that generates a key for each element in the source RDD.
- ✓ The second argument allows you to specify ascending or descending order for sort.

```
scala> val numbers = sc.parallelize(List(21, 29, 41, 15, 50,22,42,31))
scala> val sorted = numbers.sortBy(x => x, true)
```

```
scala> sorted.collect
```

OUTPUT:
Array[Int] = Array(15, 21, 22, 29, 31, 41, 42, 50)

```
scala> val names = sc.parallelize(List("AAA","XXX","UUU","WWW","BBB","SSS","C"))
scala> val sorted = names.sortBy(x => x, true)
scala> sorted.collect
```

OUTPUT:
Array[String] = Array(AAA, BBB, C, SSS, UUU, WWW, XXX)

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.

Example Code:

```
scala> val numbers = sc.parallelize(1 to 100)
scala> val numbersWithTwoPartition = numbers.coalesce(2)
scala> numbersWithTwoPartition.collect
```

OUTPUT:
Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 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, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)

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.
- ✓ The **coalesce** and **repartition** methods look similar, but the first one is used for reducing the number of partitions in an RDD, while the second one is used to increase the number of partitions in an RDD
- ✓

```
scala> val numbers = sc.parallelize(1 to 100).toList)
scala> val numbersWithOnePartition = numbers.repartition(2)
scala> numbersWithOnePartition.collect
```

OUTPUT:
Array[Int] = Array(1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93, 95, 97, 99, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36,

38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100)

scala>

Sample

The sample method returns a sampled subset of the source RDD. It takes three input parameters. The first parameter specifies the replacement strategy. The second parameter specifies the ratio of the sample size to source RDD size. The third parameter, which is optional, specifies a random seed for sampling.

```
scala> val numbers = sc.parallelize(1 to 9)

scala> val sampleNumbers = numbers.sample(true,.2)

scala>sampleNumbers.collect
```

OUTPUT:
res24: Array[Int] = Array(3, 9)

Transformations on RDD of key-value Pairs

Keys

The **keys** method returns an RDD of only the keys in the source RDD

```
scala> val stuNameMarks = sc.parallelize(List(("Raj", 70), ("Ramesh", 89), ("Rakesh", 79)))
scala> val stuNames = stuNameMarks.keys
scala> stuNames.collect
```

OUTPUT:

Array[String] = Array(Raj, Ramesh, Rakesh)

Values

The **values** method returns an RDD of only the values in the source RDD

```
scala> val stuNameMarks = sc.parallelize(List(("Raj", 70), ("Ramesh", 89), ("Rakesh", 79)))
scala> val stuMarks = stuNameMarks.values
scala> stuMarks.collect
```

OUTPUT:

Array[Int] = Array(70, 89, 79)

MapValues

- ✓ The mapValues method is a higher-order method that takes a function as input and applies it to each value in the source RDD.
- ✓ It returns an RDD of key-value pairs.
- ✓ It is similar to the map method, except that it appliesthe input function only to each value in the source RDD, so the keys are not changed.
- ✓ The returned RDD has
- ✓ the same keys as the source RDD

Example Code:

scala> val stuNameMarks = sc.parallelize(List(("Raj", 70), ("Ramesh", 89), ("Rakesh", 79)))
scala> val stuMarksFiltered = stuNameMarks.mapValues { x => 2*x}
scala> stuMarksFiltered.collect

OUTPUT:
Array[(String, Int)] = Array((Raj,140), (Ramesh,178), (Rakesh,158))

Join:

- ✓ The join method takes an RDD of key-value pairs as input and performs an inner join on the source and input RDDs.
- ✓ It returns an RDD of pairs, where the **first element** in a pair is a **key found in both source and input RDD** and the **second element** is a **tuple containing values mapped to that key in the source and input RDD**.

Example Code:

scala> val data1 = sc.parallelize(List(("Spark",10),("Scala",20),("Java",30)))
scala> val data2 = sc.parallelize(List(
("Phython","PaltformDep"),("Scala","JVM"),("Spark","Cache")))
scala> val joinData = data1.join(data2)
scala> joinData.collect

OUTPUT:

Array[(String, (Int, String))] = Array((Scala,(20,JVM)), (Spark,(10,Cache)))

scala> val joinData = data2.join(data1)
scala> joinData.collect

OUTPUT:

Array[(String, (String, Int))] = Array((Scala,(JVM,20)), (Spark,(Cache,10)))

scala> val joinData = data1.join(data1)
scala> joinData.collect

OUTPUT:

Array[(String, (Int, Int))] = Array((Java,(30,30)), (Scala,(20,20)), (Spark,(10,10)))

LeftOuterJoin:

- ✓ The leftOuterJoin method takes an RDD of key-value pairs as input and performs a left outer join on the source and input RDD.
- ✓ It returns an RDD of key-value pairs, where the first element in a pair is a key from Source RDD and the second element is a tuple containing value from source RDD and optional value from the input RDD.
- ✓ An optional value from the input RDD is represented with Option type.

Example Code:

```
scala> val data1  = sc.parallelize(List( ("Spark",10),("Scala",20),("Java",30) ))
scala> val data2  = sc.parallelize(List(
("Phython","PaltformDep"),("Scala","JVM"),("Spark","Cache") ))
scala> val joinData = data1.leftOuterJoin(data2)
scala> joinData.collect
```

OUTPUT:

Array[(String, (Int, Option[String]))] = Array((Java,(30,None)), (Scala,(20,Some(JVM))), (Spark,(10,Some(Cache))))

RightOuterJoin:

- ✓ The rightOuterJoin method takes an RDD of key-value pairs as input and performs a right outer join on the source and input RDD.
- ✓ It returns an RDD of key-value pairs, where the first element in a pair is a key from input RDD and the second element is a tuple containing optional value from source RDD and value from input RDD.
- ✓ An optional value from the source RDD is represented with the Option type.

Example Code:

```
scala> val data1  = sc.parallelize(List( ("Spark",10),("Scala",20),("Java",30) ))
scala> val data2  = sc.parallelize(List(
("Phython","PaltformDep"),("Scala","JVM"),("Spark","Cache") ))
scala> val joinData = data1.rightOuterJoin(data2)
scala> joinData.collect
```

OUTPUT:

Array[(String, (Option[Int], String))] = Array((Phython,(None,PaltformDep)), (Scala,(Some(20),JVM)), (Spark,(Some(10),Cache)))

FullOuterJoin:

- ✓ The fullOuterJoin method takes an RDD of key-value pairs as input and performs a full outer join on the source and input RDD.
- ✓ It returns an RDD of key-value pairs.

Example Code:

```
scala> val data1  = sc.parallelize(List( ("Spark",10),("Scala",20),("Java",30) ))
scala> val data2  = sc.parallelize(List(
("Phython","PaltformDep"),("Scala","JVM"),("Spark","Cache") ))
scala> val joinData = data1.fullOuterJoin(data2)
scala> joinData.collect
```

OUTPUT:

Array[(String, (Option[Int], Option[String]))] = Array((Java,(Some(30),None)), (Phython,(None,Some(PaltformDep))), (Scala,(Some(20),Some(JVM))), (Spark,(Some(10),Some(Cache))))

GroupByKey:

- ✓ The **groupByKey** method returns an RDD of pairs, where the **first element** in a pair is a **key from the source RDD** and the **second element** is a **collection of all the values that have the same key**.
- ✓ It is similar to the **groupBy** method. The difference is that groupBy is a higher-order method that takes as
- ✓ input a function that returns a key for each element in the source RDD.
- ✓ The groupByKey method operates on an RDD of key-value pairs, so a key generator function is not required as input

Example Code:

```
scala> val data2  = sc.parallelize(List(
(10,1),(20,2),(10,3),(10,4),(20,8),(40,9),(20,5),(10,6) ))
scala> val groupByKeyData = data2.groupByKey()
scala> groupByKeyData.collect
```

OUTPUT:

Array[(Int, Iterable[Int])] = Array((40,CompactBuffer(9)), (20,CompactBuffer(2, 8, 5)), (10,CompactBuffer(1, 3, 4, 6)))

Example Code 2:

```
scala> val iniData  = sc.parallelize(List(
("Spark",1),("Scala",2),("Spark",3),("Spark",4),("Spark",8),("Java",9),("Scala",5),("Spark",
6) ))
scala> val groupByKeyData = iniData.groupByKey()
scala> groupByKeyData.collect
```

OUTPUT:

Array[(String, Iterable[Int])] = Array((Java,CompactBuffer(9)), (Scala,CompactBuffer(2, 5)), (Spark,CompactBuffer(1, 3, 4, 8, 6)))

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.
- ✓ **A binary operator** takes **two values as input** and **returns a single value as output**.
- ✓ An associative operator returns the same result regardless of the grouping of the operands.
- ✓ The **reduceByKey** method can be used for aggregating values by key.
- ✓ **For example**, it can be used for
  - ➢ Calculating sum,
  - ➢ Calculating product,
  - ➢ Calculating minimum or maximum of all the values mapped to the same key.

Example Code :

```
scala> val iniData  = sc.parallelize(List(
("Spark",1),("Scala",2),("Spark",3),("Spark",4),("Spark",8),("Java",9),("Scala",5),("Spark",
6) ))
scala> val sumOfIniData = iniData.reduceByKey( (x,y) => (x+y) )
scala> sumOfIniData.collect
```

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,7), (Spark,22))

```
scala> val minByKeyData = iniData.reduceByKey((x,y) => if (x < y) x else y)
scala> minByKeyData.collect
```

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,2), (Spark,1))

```
scala> val maxByKeyData = iniData.reduceByKey((x,y) => if (x > y) x else y)
scala> minByKeyData.collect
```

OUTPUT:

Array[(String, Int)] = Array((Java,9), (Scala,5), (Spark,8))

Actions

Actions are RDD methods that **return a value** to a **driver program**.

Collect:

- ✓ The collect method returns the elements in the source RDD as an array.
- ✓ This method should be used with caution since it moves data from all the worker nodes to the driver program.
- ✓ It can crash the driver program if called on a very large RDD.

Example Code :

```
scala> val rdd = sc.parallelize((1 to 10000).toList)
scala> val filteredRdd = rdd.filter { x => (x % 1000) == 0 }
val filterResult = filteredRdd.collect
```

OUTPUT:

Array[Int] = Array(1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000)

Count:

The count method returns a count of the elements in the source RDD.

Example Code:

```
scala> val rdd = sc.parallelize((1 to 10000).toList)
scala> val totalCount = rdd.count
```

OUTPUT:

totalCount: Long = 10000

Example 2:

```
scala> val data = sc.parallelize(List ("AAA","BBB","CCCC","DD","EEEEE","FFF"))
scala> println(data.count)
```

OUTPUT:

6

Example 3:

INPUTDATA:

```
gopal@ubuntu:~/SPARK$ pwd
/home/gopal/SPARK
gopal@ubuntu:~/SPARK$ cat Input.log
spark is in-memory cluster computing system
spark is ment for spark processing
spark is having diff modules
spark is based on rdd processing only
thanks saprk
hadoop and spark are going to lead bigdata market
```

spark is in-memory cluster computing framework
spark is ment for spark processing
spark is having diff modules
spark is based on rdd processing only
thanks saprk
hadoop and spark are going to lead bigdata market
gopal@ubuntu:~/SPARK$

CODE:

```
scala> val file = sc.textFile("/home/gopal/SPARK/Input.log") // Base RDD
scala> val filData = file.filter(x => x.contains("spark")) // Transformed RDD
scala> println(filData.count)// ActionRDD or ComputedRDD
```

OUTPUT:

10

countByValue:

- ✓ The countByValue method returns a **count of each unique element** in the **source RDD**.
- ✓ It returns an instance of the **Map class containing each unique element** and **its count** as a **key-value pair**.

Example Code:

```
scala> val data = sc.parallelize(List
("AAA","BBB","CCCC","DD","AAA","BBB","AAA","BBB","AAA"))
scala> val countByValueResult = data.countByValue
```

OUTPUT:

countByValueResult: scala.collection.Map[String,Long] = Map(DD -> 1, CCCC -> 1, BBB -> 3, AAA -> 4)

first:

The first method returns the first element in the source RDD.

```
scala> val data = sc.parallelize(List
("AAA","BBB","CCCC","DD","AAA","BBB","AAA","BBB","AAA"))
scala> val firstElement = data.first
```

OUTPUT:

```
firstElement: String = AAA
```

Example 2:

INPUTDATA:

```
gopal@ubuntu:~/SPARK$ pwd
/home/gopal/SPARK
gopal@ubuntu:~/SPARK$ cat Input.log
spark is in-memory cluster computing system
spark is ment for spark processing
spark is having diff modules
```

```
spark is based on rdd processing only
thanks saprk
hadoop and spark are going to lead bigdata market
spark is in-memory cluster computing framework
spark is ment for spark processing
spark is having diff modules
spark is based on rdd processing only
thanks saprk
hadoop and spark are going to lead bigdata market
gopal@ubuntu:~/SPARK$
```

CODE:

```
scala> val file = sc.textFile("/home/gopal/SPARK/Input.log") // Base RDD
scala> val filData = file.filter(x => x.contains("spark")) // Transformed RDD
scala> println(filData.first) // ActionRDD
```

OUTPUT:

spark is in-memory cluster computing system

max:

The max method returns the largest element in an RDD.

```
scala> val rdd = sc.parallelize(List(45,98,73,61,55,85,91,10))
scala> rdd.max
```

OUTPUT:

Int = 98

```
scala> val rdd = sc.parallelize(List("Spark","Spark&Scala","A","Java"))
scala> rdd.max
```

OUTPUT:

String = Spark&Scala

min:

The min method returns the smallest element in an RDD

```
scala> val rdd = sc.parallelize(List(45,98,73,61,55,85,91,10))
scala> rdd.min
```

OUTPUT:

Int = 10

```
scala> val rdd = sc.parallelize(List("Spark","Spark&Scala","A","Java"))
scala> rdd.min
```

OUTPUT:

String = A

Top:

The top method takes an **integer N as input** and **returns an array containing the N largest elements in the source RDD**.

```
scala> val rdd = sc.parallelize(List(45,98,73,61,55,85,91,10))
scala> rdd.top(3)
```

OUTPUT:
Array[Int] = Array(98, 91, 85)

```
scala> val rdd = sc.parallelize(List("Spark","Spark&Scala","A","Java"))
scala> rdd.top(2)
```

OUTPUT:
Array[String] = Array(Spark&Scala, Spark)

reduce:

The higher-order reduce method aggregates the elements of the source RDD using an associative and commutative binary operator provided to it.

```
scala> val data = sc.parallelize(List(10,20,10,30,50,70))
scala> val sum = data.reduce( (x,y) => (x + y) )
```

OUTPUT:
sum: Int = 190

```
scala> val multipliedProduct = data.reduce( (x,y) => (x * y) )
```

OUTPUT:
multipliedProduct: Int = 210000000

Actions on RDD of key-value Pairs

CountByKey:

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

```
scala> val iniData  = sc.parallelize(List(
("Spark",1),("Scala",2),("Spark",3),("Spark",4),("Spark",8),("Java",9),("Scala",5),("Spark",
6) ))
scala> val countOfKeys = iniData.countByKey()
```

OUTPUT:
countOfKeys: scala.collection.Map[String,Long] = Map(Java -> 1, Scala -> 2, Spark -> 5)

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**.

```
scala> val iniData  = sc.parallelize(List(
("Spark",1),("Scala",2),("Spark",3),("Spark",4),("Spark",8),("Java",9),("Scala",5),("Spark",
6) ))
scala> val countOfKeys = iniData.lookup("Spark")
```

OUTPUT:
countOfKeys: Seq[Int] = WrappedArray(1, 3, 4, 8, 6)