Web: Inceptez.com Mail: info@inceptez.com Call: 7871299810, 7871299817

SPARK CORE

Copy data into /home/hduser/sparkdata directory.

hadoop fs -put /home/hduser/sparkdata/empdata.txt

Creating an RDD

From file:

// RDD creation using spark context

```
import spark.implicits._

val hadooplines= sc.textFile("hdfs:/user/hduser/empdata.txt")

val lines = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

val alllines = sc.textFile("file:/home/hduser/sparkdata/*data.txt")
```

// RDD creation using spark session

```
import org.apache.spark.sql.SparkSession
val spark = SparkSession.builder().getOrCreate()

val hadooplines = spark.sparkContext.textFile("hdfs:/user/hduser/empdata.txt")

val lines = spark.sparkContext.textFile("file:/home/hduser/sparkdata/empdata.txt")

val alllines = spark.sparkContext.textFile("file:/home/hduser/sparkdata/*data.txt")

alllines.foreach(println)
```

```
hadooplines.foreach(println)
```

From another RDD:

```
val filterlines = lines.filter{ l => l.length > 37}

filterlines.foreach(println)
```

From Memory:

```
filterlines.cache

filterlines.count
```

Transformations

A transformation method of an RDD creates a new RDD by performing a computation on the source RDD.

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

```
val lines = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

val lengths = lines map { l => l.length}

lengths.foreach(println)

val lengths = {lines.map(x=>x.split(",")).map(l => l.length)}

lengths.foreach(println)
```

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.

```
val chennaiLines = lines.map(x=>x.split(",")).filter(l => l(1).toUpperCase=="CHENNAI" )
chennaiLines.collect
```

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.

```
val fmrdd = lines.flatMap( l => l.split(",")).map(x=>x.toString.toUpperCase)

fmrdd.foreach(println)
```

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 lengths = lines.mapPartitions ( x => x.filter( l => l.length>20))
lengths.foreach(println)
```

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.

```
val linesFile1 = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

val linesFile2 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")

val linesFromBothFiles = linesFile1.union(linesFile2)

linesFromBothFiles.foreach(println)
```

Intersection and Subtract are given in additional use cases

Distinct

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

```
val uniqueNumbers = linesFromBothFiles.distinct

uniqueNumbers.foreach(println)
```

Zip:

Joins two RDDs by combining the i-th of either partition with each other.

```
val linesFile1 = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")
val ziplines = linesFile1.zip(linesFile1)
ziplines.foreach(println)
```

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 Customer(name: String, city: String, age: Int)

val customers = lines.map(x=>x.split(",")).map(x=>Customer(x(0), x(1), x(2).toInt))
customers.collect
val groupByZip = customers.groupBy { a => a.city}
groupByZip.collect
```

or

```
val groupByZip = customers.groupBy { _.city}

groupByZip.foreach(println)
```


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 100).toList,5)
```

To see the number of partitions

```
numbers.partitions.size
```

```
numbers.glom().collect
```

Coalesce reduced the partition number to 2 from 5

```
val numbersWithTwoPartition = numbers.coalesce(2)
```

```
numbersWithTwoPartition.glom().collect
```

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 numbersWithFourPartition = numbers.repartition(6)
numbersWithFourPartition.partitions.size
numbersWithFourPartition.glom().collect
```

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.

~~val custone=customers.coalesce(1)~~

```
val sortedByAge = customers sortBy( p => p.age, true)
```

```
sortedByAge.foreach(println)
```

Join:

The join method takes an RDD of key-value pairs as input and performs an inner join on the source and input 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.foreach(println)

val joinRdd = pairRdd1.rightOuterJoin(pairRdd2)

joinRdd.foreach(println)

val joinRdd = pairRdd1.fullOuterJoin(pairRdd2)

joinRdd.foreach(println)

val joinRdd = pairRdd1.leftOuterJoin(pairRdd2)

joinRdd.foreach(println)
```

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)))

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

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

sumByKeyRdd.collect

minByKeyRdd.collect
```

Checkpoint:

Checkpointing is a process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system. Will create a checkpoint when the RDD is computed next. Checkpointed RDDs are stored as a binary file within the checkpoint directory which can be specified using the Spark context. (Warning: Spark applies lazy evaluation. Checkpointing will not occur until an action is invoked.)

Note: the directory "/tmp/ckptdir" should exist in all slaves. As an alternative you could use an HDFS directory URL as well.

```
sc.setCheckpointDir("/tmp/ckptdir")

val ckptrdd = sc.parallelize(1 to 4)

ckptrdd.checkpoint

ckptrdd.count
```

HDFS:

```
sc.setCheckpointDir("hdfs://localhost:54310/tmp/ckptdir")
```

Actions :

Actions are RDD methods that return a value to a driver program. This section discusses the commonly used RDD actions.

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.

```
val rdd = sc.parallelize((1 to 10000).toList)

val filteredRdd = rdd filter { x => (x % 1000) == 0}

val filterResult = filteredRdd.collect
```

Count :

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

```
val rdd = sc.parallelize((1 to 10000).toList)
```

```
val total = rdd.count
```

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.

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

val counts = rdd.countByValue
```

Reduce :

The higher-order reduce method aggregates the elements of the source RDD using an associative and commutative binary operator provided to it. It is similar to the fold method; however, it does not require a neutral zero value.

```
val numbersRdd = sc.parallelize(List(2, 5, 3, 1))

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

val product = numbersRdd.reduce((x, y) => x * y)
```

Actions on RDD of key-value Pairs :

RDDs of key-value pairs support a few additional actions, which are briefly described next.

CountByKey :

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

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

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(("a", 1), ("b", 2), ("c", 3), ("a", 11), ("b", 22), ("a",1)))
```

```
val values = pairRdd.lookup("a")
```

SaveAsTextFile :

The saveAsTextFile method saves the elements of the source RDD in the specified directory on any Hadoop-supported file system. Each RDD element is converted to its string representation and stored as a line of text.

```
val logs = sc.textFile("file:/usr/local/hadoop/logs/hadoop-hduser-datanode-Inceptez.log")

val errorsAndWarnings = logs filter { l => l.contains("ERROR") || l.contains("WARN")}

val fs = org.apache.hadoop.fs.FileSystem.get(new java.net.URI("hdfs://localhost:54310"),

sc.hadoopConfiguration)

fs.delete(new org.apache.hadoop.fs.Path("/user/hduser/errorsAndWarnings"),true)

errorsAndWarnings.saveAsTextFile("hdfs:///user/hduser/ errorsAndWarnings")
```

`RDD Caching Methods :

The RDD class provides two methods to cache an RDD: cache and persist.

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()

val errorLogs = errorsAndWarnings filter { l => l.contains("ERROR")}

val warningLogs = errorsAndWarnings filter { l => l.contains("WARN")}

 val errorCount = errorLogs.count

val warningCount = warningLogs.count
```

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 behavior is identical to that of the cache method.

```
errorsAndWarnings.persist()
```

```
errorsAndWarnings.unpersist()
```

Cache Memory Management

Spark automatically manages cache memory using LRU (least recently used) algorithm. It removes old RDD partitions from cache memory when needed. In addition, the RDD API includes a method called unpersist(). An application can call this method to manually remove RDD partitions from memory.

StorageLevel

StorageLevel describes how an RDD is persisted (and addresses the following concerns):
Does RDD use disk?
Does RDD use memory to store data?
How much of RDD is in memory?
Does RDD use off-heap memory?
Should an RDD be serialized or not (while storing the data)?
How many replicas (default: 1) to use (can only be less than 40)?
There are the following StorageLevel (number _2 in the name denotes 2 replicas):

NONE (default)
DISK_ONLY
DISK_ONLY_2
MEMORY_ONLY (default for cache operation for RDDs)
MEMORY_ONLY_2
MEMORY_ONLY_SER
MEMORY_ONLY_SER_2
MEMORY_AND_DISK
MEMORY_AND_DISK_2
MEMORY_AND_DISK_SER
MEMORY_AND_DISK_SER_2
OFF_HEAP

```
val lines = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

lines.persist(org.apache.spark.storage.StorageLevel.DISK_ONLY)
lines.unpersist()
lines.persist(org.apache.spark.storage.StorageLevel.DISK_ONLY_2)
lines.unpersist()
lines.persist(org.apache.spark.storage.StorageLevel.MEMORY_ONLY_2)
lines.unpersist()
lines.persist(org.apache.spark.storage.StorageLevel.MEMORY_ONLY_SER)
```

```
lines.unpersist()
lines.persist(org.apache.spark.storage.StorageLevel.MEMORY_AND_DISK_SER_2)
```

getStorageLevel

Retrieves the currently set storage level of the RDD. This can only be used to assign a new storage level if the RDD does not have a storage level set yet. The example below shows the error you will get, when you try to reassign the storage level.

```
lines.getStorageLevel.description
```

String = Disk Serialized 1x Replicated

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.

Broadcast variables are created by wrapping with SparkContext.broadcast function as shown in the following Scala code

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

val broadcastVar = sc.broadcast(2)

val added = input.map(x => broadcastVar.value + x)
added.foreach(println)

val multiplied = input.map(x => broadcastVar.value * x)

multiplied.foreach(println)
```

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.

```
val accum = sc.accumulator(0,"IZ Accumulator")
sc.parallelize(Array(1, 2, 3)).foreach(x => accum += x)
accum.value
```

More Transformations and Actions for more self-practice:

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.

```
val linesFile1 = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

val linesFile2 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")

val linesPresentInBothFiles = linesFile1.intersection(linesFile2)

linesPresentInBothFiles.foreach(println)
```

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.

```
val linesInFile1Only = linesFile1.subtract(linesFile2)

linesInFile1Only.foreach(println)
```

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.

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

val alphabets = sc.parallelize(List("a", "b", "c", "d"))

val cartesianProduct = numbers.cartesian(alphabets)

cartesianProduct.foreach(println)
```

First :

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

```
val rdd = sc.parallelize(List(10, 5, 3, 1))
val firstElement = rdd.first  firstElement:
```

Int = 10

Max :

The max method returns the largest element in an RDD.

```
val rdd = sc.parallelize(List(2, 5, 3, 1))
val maxElement =
rdd.max
```

maxElement: Int = 5

Min :

The min method returns the smallest element in an

RDD. val rdd = sc.parallelize(List(2, 5, 3, 1))

val minElement = rdd.min minElement:

Int = 1

Take :

The take method takes an integer N as input and returns an array containing the first N element in the source RDD.

```
val rdd = sc.parallelize(List(2, 5, 3, 1, 50,100))
val first3 = rdd.take(3) first3: Array[Int] =
```

Array(2, 5, 3)

TakeOrdered :

The takeOrdered method takes an integer N as input and returns an array containing the N smallest elements in the source RDD.

```
val rdd = sc.parallelize(List(2, 5, 3, 1, 50,100))  val
smallest3 = rdd.takeOrdered(3)

smallest3: Array[Int] = Array(1, 2, 3)
```

Top :

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

```
val rdd = sc.parallelize(List(2, 5, 3, 1, 50, 100))

val largest3 = rdd.top(3)

largest3: Array[Int] = Array(100, 50, 5)
```

Fold :

The higher-order fold method aggregates the elements in the source RDD using the specified neutral zero value and an associative binary operator. It first aggregates the elements in each RDD partition and then aggregates the results from each partition.

```
val numbersRdd = sc.parallelize(List(2, 5, 3, 1))

val sum = numbersRdd.fold(0) ((partialSum, x) => partialSum + x)

sum: Int = 11
```

val product = numbersRdd.fold(1) ((partialProduct, x) => partialProduct * x)

product: Int = 30

Actions on RDD of Numeric Types :

RDDs containing data elements of type Integer, Long, Float, or Double support a few additional actions that are useful for statistical analysis.

Mean :

The mean method returns the average of the elements in the source

RDD. val numbersRdd = sc.parallelize(List(2, 5, 3, 1))

val mean =

numbersRdd.mean mean:

Double = 2.75

Stdev :

The stdev method returns the standard deviation of the elements in the source RDD.

val numbersRdd = sc.parallelize(List(2, 5, 3,

1)) val stdev = numbersRdd.stdev

stdev: Double = 1.479019945774904

Sum :

The sum method returns the sum of the elements in the source

RDD. val numbersRdd = sc.parallelize(List(2, 5, 3, 1))

val sum = numbersRdd.sum

sum:

Double = 11.0

Variance :

The variance method returns the variance of the elements in the source RDD.

```
val numbersRdd = sc.parallelize(List(2, 5, 3, 1))

val variance = numbersRdd.variance

variance: Double = 2.1875
```