SPARK CORE

As pre-requisite let's have a sample file data in local linux machine or in HDFS or both.
For example we are going to play with the following sample files.

sample files:

empdata.txt

[hduser@tech sparkdata]$ **cat empdata.txt**

```
ArunKumar~chennai~33~2016-09-20~100000
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
```

empdata1.txt

```
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000
```

employee.txt

```
10,bharath
20,kumar
30,arvind
40,bala
```

sales.csv

[hduser@tech sparkdata]$ **cat sales.csv**

```
productId,productName,stdCost,stdPrice
PURA100,Pure Soft Detergent - 100ml, $1.50 , $3.00
PURA200,Pure Soft Detergent - 200ml, $2.00 , $3.99
PURA250,Pure Soft Detergent - 250ml, $2.30 , $4.50
PURA500,Pure Soft Detergent - 500ml, $3.50 , $6.50
DETA100,Detafast Stain Remover - 100ml, $3.00 , $6.00
DETA200,Detafast Stain Remover - 200ml, $3.50 , $6.50
DETA800,Detafast Stain Remover - 800ml, $6.00 , $9.00
SUPA101,Super Soft - Product Sample, $0.30 , $-
SUPA102,Super Soft - 250ml, $2.50 , $4.50
SUPA103,Super Soft - 500ml, $3.50 , $6.99
SUPA104,Super Soft - 1 Litre, $5.00 , $9.99
SUPA105,Super Soft Bulk - 2 Litres, $8.00 , $14.50
```

quiz.json

[hduser@tech sparkdata]$ **cat quiz.json**

```
{
   "quiz": {
      "sport": {
         "q1": {
```

```
            "question": "Which one is correct team name in NBA?",
            "options": [
              "New York Bulls",
              "Los Angeles Kings",
              "Golden State Warriros",
              "Huston Rocket"
            ],
            "answer": "Huston Rocket"
          }
        },
        "maths": {
          "q1": {
            "question": "5 + 7 = ?",
            "options": [
              "10",
              "11",
              "12",
              "13"
            ],
            "answer": "12"
          },
          "q2": {
            "question": "12 - 8 = ?",
            "options": [
              "1",
              "2",
              "3",
              "4"
            ],
            "answer": "4"
          }
        }
    }
}
```


Creating an RDD

From file:

```
// RDD creation using spark context
Import org.apache.spark_
val sc = new sparkContext();
```

```
scala> val empfile = sc.textFile("hdfs:/user/hduser/sparkworks/empdata.txt")
empfile: org.apache.spark.rdd.RDD[String] = hdfs:/user/hduser/sparkworks/empdata.txt MapPartitionsRDD[13]
at textFile at <console>:24
```

```
scala> empfile.foreach(println)
```

Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
ArunKumar~chennai~33~2016-09-20~100000
Varun~chennai~34~2016-09-21~10000

scala> **empfile.glom.collect**
res3: Array[Array[String]] = Array(Array(ArunKumar,chennai,33,2016-09-20,100000, Varun,chennai,34,2016-09-21,10000), Array(Lara,chennai,55,2016-09-21,10000, vasudevan,banglore,43,2016-09-23,90000))

// RDD creation using spark session

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

scala> **val empfile = spark.read.textFile("hdfs:/user/hduser/sparkworks/empdata.txt")**
empfile: org.apache.spark.rdd.RDD[String] = hdfs:/user/hduser/sparkworks/empdata.txt MapPartitionsRDD[13] at textFile at <console>:24

Below command filters the rows which are less than 37 in length.
scala> **val empfilter = empfile.filter{z=>z.length > 37}**
empfilter: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[58] at filter at <console>:26

scala> **empfilter.foreach(println)**
ArunKumar,chennai,33,2016-09-20,100000
vasudevan,banglore,43,2016-09-23,90000

From Memory:
scala> **empfilter.cache**
res13: empfilter.type = MapPartitionsRDD[58] at filter at <console>:26

scala> **empfilter.count**
res14: Long = 2

Transformations

Map:

scala> **val empmap = empfile.map{z=>z.split(",")}**
empmap: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[74] at map at <console>:26

scala> **empmap.collect**
res31: Array[Array[String]] = Array(Array(ArunKumar, chennai, 33, 2016-09-20, 100000), Array(Varun, chennai, 34, 2016-09-21, 10000), Array(Lara, chennai, 55, 2016-09-21, 10000), Array(vasudevan, banglore, 43, 2016-09-23, 90000))

Filter:

Below command filters the rows which are less than 37 in length.

scala> **val empfilter = empfile.filter{z=>z.length > 37}**
empfilter: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[58] at filter at <console>:26

scala> **empfilter.foreach(println)**
ArunKumar,chennai,33,2016-09-20,100000
vasudevan,banglore,43,2016-09-23,90000

FlatMap:

scala> **val empflat = empfile.flatMap{z=>z.split(",")}**
empflat: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[75] at flatMap at <console>:26

scala> **empflat.collect**
res32: Array[String] = Array(ArunKumar, chennai, 33, 2016-09-20, 100000, Varun, chennai, 34, 2016-09-21, 10000, Lara, chennai, 55, 2016-09-21, 10000, vasudevan, banglore, 43, 2016-09-23, 90000)

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.

scala> **val maprddpart = sc.parallelize(List("hello,kannan","how,are,you","age 31","engineer","male"),3).map{z=>z.split(",")}**
maprddpart: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[212] at map at <console>:28

scala> **maprddpart.glom.collect**
res80: Array[Array[Array[String]]] = Array(Array(Array(hello, kannan)), Array(Array(how, are, you), Array(age 31)), Array(Array(engineer), Array(male)))

- Above command uses map function with 3 cores to use for computation.
- So we can see three nested arrays created as output of map function.
- Let's do the same with mapPartitions.
- Since split is not supported by mapFunctions, let's try to filter with same 3 cores.

scala> **val maprddpart = sc.parallelize(List("hello,kannan","how,are,you","age 31","engineer","male"),3).mapPartitions{z=>z.filter(x=>x.length < 40)}**
maprddpart: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[215] at mapPartitions at <console>:28

scala> **maprddpart.glom.collect**
res81: Array[Array[String]] = Array(Array(hello,kannan), Array(how,are,you, age 31), Array(engineer, male))

- ❖ Here elements of RRD are not taking into consideration while creating another RDD.
- ❖ Instead partitions of RDD is considered and based on it transformed RDD is created.
- ❖ Here in our example typically no of core = no of partitions.

Union: (horizontal join)

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.

```
scala> val empfile = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")
empfile: org.apache.spark.rdd.RDD[String] = file:/home/hduser/sparkdata/empdata.txt MapPartitionsRDD[196]
at textFile at <console>:28
```

```
scala> val empfile1 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")
empfile1: org.apache.spark.rdd.RDD[String] = file:/home/hduser/sparkdata/empdata1.txt
MapPartitionsRDD[198] at textFile at <console>:28
```

- ❖ The above two files has data as shown at the top of this document.

```
scala> val empfileunion1 = empfile.union(empfile1)
empfileunion1: org.apache.spark.rdd.RDD[String] = UnionRDD[199] at union at <console>:32
```

```
scala> empfileunion1.foreach(println)
ArunKumar~chennai~33~2016-09-20~100000
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000
```

- ❖ here we can notice that the data from two files joined horizontally.
- ❖ Note that duplicate rows are not ignored. We don't have Union/Union All concept here as in RDBMS.
- ❖ Here while doing UNION, duplicates wont be ignored.

```
scala> val employee = sc.textFile("file:/home/hduser/sparkdata/employee.txt")
employee: org.apache.spark.rdd.RDD[String] = file:/home/hduser/sparkdata/employee.txt
MapPartitionsRDD[201] at textFile at <console>:28
```

```
scala> val empfileunion2 = empfile.union(employee)
empfileunion2: org.apache.spark.rdd.RDD[String] = UnionRDD[204] at union at <console>:32
```

- ❖ In the above UNION function, we have combined two files which have different structure.
- ❖ In SPARK Union doesn't require two files to be with same structure as required in RDBMS.

```
scala> empfileunion2.foreach(println)
ArunKumar~chennai~33~2016-09-20~100000
```

Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
10,bharath
20,kumar
30,arvind
40,bala

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.

scala> **val empfiledist = empfileunion1.distinct**
empfiledist: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[207] at distinct at <console>:34

scala> **empfiledist.distinct.foreach(println)**
kannan~coimbatore~31~1987-11-22~100000
ArunKumar~chn~33~2016-09-20~100
Varun~chennai~34~2016-09-21~10000
ArunKumar~chennai~33~2016-09-20~100000
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~43~2016-09-23~90000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~39~2016-09-23~90000

- here we can notice that distinct function is applied on the RDD empfileunion1 which was derived in the above discussion, and the duplicate rows have been removed.

Zip:

- The zip function is used to combine two RDDs into the RDD of the Key / Value form.
- The default number of RDD partitions and the number of elements are the same, otherwise an exception is thrown
- Joins two RDDs by combining the i-th of either partition with each other.

scala> **val zipsamefile = empfile.zip(empfile)**
zipsamefile: org.apache.spark.rdd.RDD[(String, String)] = ZippedPartitionsRDD2[217] at zip at <console>:30

scala> **zipsamefile.foreach(println)**
(Lara~chennai~55~2016-09-21~10000,Lara~chennai~55~2016-09-21~10000)
(vasudevan~banglore~43~2016-09-23~90000,vasudevan~banglore~43~2016-09-23~90000)
(ArunKumar~chennai~33~2016-09-20~100000,ArunKumar~chennai~33~2016-09-20~100000)
(Varun~chennai~34~2016-09-21~10000,Varun~chennai~34~2016-09-21~10000)

- In the above commands, we have done zip on same file.
- Since we have used same file, the partition and no or elements in partition will be same.
- So we were able to zip-join the files

scala> **val zipdiffile = empfile.zip(empfile1)**
zipdiffile: org.apache.spark.rdd.RDD[(String, String)] = ZippedPartitionsRDD2[218] at zip at <console>:32

scala> **zipdiffile.foreach(println)**
(ArunKumar~chennai~33~2016-09-20~100000,ArunKumar~chn~33~2016-09-20~100)
(Varun~chennai~34~2016-09-21~10000,srinivasan~chn~33~2016-09-21~1000)
(Lara~chennai~55~2016-09-21~10000,Lara~chennai~55~2016-09-21~10000)
(vasudevan~banglore~43~2016-09-23~90000,kannan~coimbatore~31~1987-11-22~100000)
19/06/02 08:02:26 ERROR executor.Executor: Exception in task 0.0 in stage 84.0 (TID 1754)
org.apache.spark.SparkException: Can only zip RDDs with same number of elements in each partition
at org.apache.spark.rdd.RDD$$anonfunzip1$$anonfun$apply$26$$anon$2.hasNext(RDD.scala:836)
at scala.collection.Iterator$class.foreach(Iterator.scala:893)
.....
.....

- In the above example, we have used two different files.
- Empfile has 4 rows and empfile1 has 7 rows.
- So partitions will be different for each file.
- While we try to do zip on these two files the below exception error is thrown.

scala> val zipdiffile1 = empfile.zip(employee)
zipdiffile1: org.apache.spark.rdd.RDD[(String, String)] = ZippedPartitionsRDD2[219] at zip at <console>:32

scala> zipdiffile1.foreach(println)
(ArunKumar~chennai~33~2016-09-20~100000,10,bharath)
(Varun~chennai~34~2016-09-21~10000,20,kumar)
(Lara~chennai~55~2016-09-21~10000,30,arvind)
(vasudevan~banglore~43~2016-09-23~90000,40,bala)

- In the above example, both files have 4 rows and so both files would be having 2 partitions typically in this case.
- So the zip function will work in this case.

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.

scala> **case class emp(name:String, city:String, age:Int, date:String, amount:Int){}**
defined class emp

scala> **val empcase = empfiledist.map{z=>z.split("~")}.map{x=>emp(x(0),x(1),x(2).toInt,x(3),x(4).toInt)}**
empcase: org.apache.spark.rdd.RDD[emp] = MapPartitionsRDD[227] at map at <console>:38

scala**> val empamt = empcase.groupBy(a=>a.city)**
empamt: org.apache.spark.rdd.RDD[(String, Iterable[emp])] = ShuffledRDD[229] at groupBy at <console>:40

- if we notice the output of println below, we can see the group by function by “city”.
- For each city, the corresponding rows will be gathered together

scala> **empamt.foreach(println)**
(chn,CompactBuffer(emp(ArunKumar,chn,33,2016-09-20,100), emp(srinivasan,chn,33,2016-09-21,1000)))
(coimbatore,CompactBuffer(emp(kannan,coimbatore,31,1987-11-22,100000)))
(chennai,CompactBuffer(emp(Varun,chennai,34,2016-09-21,10000), emp(ArunKumar,chennai,33,2016-09-20,100000), emp(Lara,chennai,55,2016-09-21,10000)))
(banglore,CompactBuffer(emp(vasudevan,banglore,43,2016-09-23,90000), emp(vasudevan,banglore,39,2016-09-23,90000)))
(mysore,CompactBuffer(emp(vimal,mysore,43,2016-09-23,9000)))

scala> **empamt.collect**
res96: Array[(String, Iterable[emp])] = Array((chennai,CompactBuffer(emp(Varun,chennai,34,2016-09-21,10000), emp(ArunKumar,chennai,33,2016-09-20,100000), emp(Lara,chennai,55,2016-09-21,10000))), (chn,CompactBuffer(emp(ArunKumar,chn,33,2016-09-20,100), emp(srinivasan,chn,33,2016-09-21,1000))), (coimbatore,CompactBuffer(emp(kannan,coimbatore,31,1987-11-22,100000))), (banglore,CompactBuffer(emp(vasudevan,banglore,43,2016-09-23,90000), emp(vasudevan,banglore,39,2016-09-23,90000))), (mysore,CompactBuffer(emp(vimal,mysore,43,2016-09-23,9000))))

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.

scala> **val maprddpart = sc.parallelize(List("hello,kannan","how,are,you","age 31","engineer","male"),10)**
maprddpart: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[230] at parallelize at <console>:28

scala> **maprddpart.partitions.size**
res98: Int = 10

- in the above example, we are creating a RDD with 10 partitions explicitly
- when we check the partition size, it returns 10
- let's save this RDD as a text file..
- NOTE: we will discuss about saving as text file in detailed later.

scala> **maprddpart.saveAsTextFile("hdfs:/user/hduser/sparkworks/coaleacerdd.txt")**

[hduser@tech sparkdata]$ **hadoop fs -ls /user/hduser/sparkworks/coaleacerdd.txt/**
Found 11 items
-rw-r--r-- 1 hduser hadoop 0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/_SUCCESS
-rw-r--r-- 1 hduser hadoop 0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00000
-rw-r--r-- 1 hduser hadoop 13 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00001
-rw-r--r-- 1 hduser hadoop 0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00002
-rw-r--r-- 1 hduser hadoop 12 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00003
-rw-r--r-- 1 hduser hadoop 0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00004

```
-rw-r--r--   1 hduser hadoop          7 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00005
-rw-r--r--   1 hduser hadoop          0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00006
-rw-r--r--   1 hduser hadoop          9 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00007
-rw-r--r--   1 hduser hadoop          0 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00008
-rw-r--r--   1 hduser hadoop          5 2019-06-02 10:20 /user/hduser/sparkworks/coaleacerdd.txt/part-00009
```

- After saving the above RDD with 10 partitions, if we check the saved location we can see 10 different files created .. one file per partition.
- Same of one file is shown below.

```
[hduser@tech sparkdata]$ hadoop fs -cat /user/hduser/sparkworks/coaleacerdd.txt/part-00001
hello,kannan
```

- now lets take the same RDD and lets reduce its partition and see what happens.

```
scala> val coalescerdd = maprddpart.coalesce(1)
coalescerdd: org.apache.spark.rdd.RDD[String] = CoalescedRDD[232] at coalesce at <console>:30

scala> coalescerdd.partitions.size
res99: Int = 1

scala> coalescerdd.saveAsTextFile("hdfs:/user/hduser/sparkworks/coaleacerdd1")

[hduser@tech sparkdata]$ hadoop fs -ls /user/hduser/sparkworks/coaleacerdd1/
Found 2 items
-rw-r--r--   1 hduser hadoop          0 2019-06-02 10:27 /user/hduser/sparkworks/coaleacerdd1/_SUCCESS
-rw-r--r--   1 hduser hadoop         46 2019-06-02 10:27 /user/hduser/sparkworks/coaleacerdd1/part-00000

[hduser@tech sparkdata]$ hadoop fs -cat /user/hduser/sparkworks/coaleacerdd1/part-00000
hello,kannan
how,are,you
age 31
engineer
male
```

- in the above example, we have reduced the partition of the same RDD to 1. And while saving that RDD as a text file, we can see only one file created.

- Typically this coalesce is used in two scenarios.
  - When the final output is to be saved, instead of saving as multiple files, we can reduce the number of partitions to one and can save into a single file.
  - When a large data is being processed, we use multiple partitions to achieve performance.

When the data is narrowed down to small data, then that many partitions (parallel threads) are not required to process that small data.
So no of partitions can be reduces to small number and can process that small data.

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.

```
scala> val repartcoalescerdd = coalescerdd.repartition(6)
repartcoalescerdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[237] at repartition at <console>:32

scala> repartcoalescerdd.partitions.size
res102: Int = 6

scala> repartcoalescerdd.glom.collect
res103: Array[Array[String]] = Array(Array(), Array(hello,kannan), Array(how,are,you), Array(age 31),
Array(engineer), Array(male))
```

as an example,

- ❖ Consider we handle a small data with lesser partition.
- ❖ We get a requirement that we need to merge some larger data with this small data and process it
- ❖ Then we can increase the partition number after merge and process it for better efficiency.

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 empcase = empfile1.map{z=>z.split(",")}.map{x=>emp(x(0),x(1),x(2).toInt,x(3),x(4).toInt)}
empcase: org.apache.spark.rdd.RDD[emp] = MapPartitionsRDD[10] at map at <console>:28

scala> val empsort = empcase.sortBy(_.name)          → this is equal to (z=>z.name)
empsort: org.apache.spark.rdd.RDD[emp] = MapPartitionsRDD[15] at sortBy at <console>:30

scala> empsort.foreach(println)
emp(ArunKumar,chennai,33,2016-09-20,100000)
emp(kannan,chennai,33,2016-09-20,10)
emp(Lara,chennai,55,2016-09-21,10000)
emp(vandhana,chennai,33,2016-09-20,10)
emp(Varun,chennai,34,2016-09-21,10000)
emp(vasudevan,banglore,43,2016-09-23,90000)
```

Join:

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

- ❖ This join function works only for key-value pair RDDs.
- ❖ So to join two RDDs we need to make them as key-value pairs.
- ❖ In the below example, collection of key-value paired tuples are gathered into a list.

```
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)

scala> joinRdd.foreach(println)
(c,(3,third))
(b,(2,second))

val joinRdd = pairRdd1.rightOuterJoin(pairRdd2)

scala> joinRdd.foreach(println)
(c,(Some(3),third))
(d,(None,fourth))
(b,(Some(2),second))

val joinRdd = pairRdd1.leftOuterJoin(pairRdd2)

scala> joinRdd.foreach(println)
(a,(1,None))
(c,(3,Some(third)))
(b,(2,Some(second)))

val joinRdd = pairRdd1.fullOuterJoin(pairRdd2)
```

- ❖ Above we have seen some simple example. Now lets see how to get data from file and make it into key-value pair and then join them.
- ❖ Lets create couple of RDDs (empfile1 and empfile2) using different files.

```
scala> val empfile1 = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")

scala> val empfile2 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")

scala> empfile1.foreach(println)
ArunKumar~chennai~33~2016-09-20~100000
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
```

subha~canada~45~2019-02-01~41265
sudhan~canada~10~2019-02-01~25652
shavan~canada~7~2019-02-03~2565

scala> **empfile2.foreach(println)**
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000

- now lets pick the name and amout from the file and make is as key-value pair.

scala> **val emptbl1 = empfile1.map{_.split("~")}.map{z=>(z(0),z(4).toInt)}**

scala> **emptbl1.foreach(println)**
(subha,41265)
(sudhan,25652)
(shavan,2565)
(ArunKumar,100000)
(Varun,10000)
(Lara,10000)
(vasudevan,90000)

scala> **val emptbl2 = empfile2.map{_.split("~")}.map{z=>(z(0),z(4).toInt)}**

scala> **emptbl2.foreach(println)**
(ArunKumar,100)
(srinivasan,1000)
(vimal,9000)
(vasudevan,90000)
(Lara,10000)
(kannan,100000)

- We have obtained key-value pair RRDs form both the files.
- Lets use these two RDDs to see how join works.

LEFT OUTER JOIN
scala> **val empleft = emptbl1.leftOuterJoin(emptbl2)**

scala> empleft.foreach(println)
(subha,(41265,None))
(sudhan,(25652,None))
(shavan,(2565,None))
(vasudevan,(90000,Some(90000)))
(ArunKumar,(100000,Some(100)))
(Lara,(10000,Some(10000)))

(Varun,(10000,None))

RIGHT OUTER JOIN

scala> val empright = emptbl1.leftOuterJoin(emptbl2)

scala> empright.foreach(println)
(ArunKumar,(100000,Some(100)))
(Lara,(10000,Some(10000)))
(Varun,(10000,None))
(subha,(41265,None))
(sudhan,(25652,None))
(shavan,(2565,None))
(vasudevan,(90000,Some(90000)))

FULL OUTER JOIN

scala> val empfull = emptbl1.fullOuterJoin(emptbl2)

scala> empfull.foreach(println)
(subha,(Some(41265),None))
(sudhan,(Some(25652),None))
(shavan,(Some(2565),None))
(kannan,(None,Some(100000)))
(vimal,(None,Some(9000)))
(vasudevan,(Some(90000),Some(90000)))
(ArunKumar,(Some(100000),Some(100)))
(Lara,(Some(10000),Some(10000)))
(Varun,(Some(10000),None))
(srinivasan,(None,Some(1000)))

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.

- We have seen how group by works. Group by also works with key-value paired RDDs.
- Now lets see how to make aggregate action on key-value paired data.

Let's do the following in the below example and learn how Reduce by Key works.

- Create two RDDs from two different files. (here is have taken two diff files which is of same structure)
- Union the two files
- Apply map function and Split the data by ("~")
- Apply map function and take "city" and "amount" columns alone as key-value paired tuples.
- Apply **reduceByKey** "city" and get sum of "amount".
- This is similar to select city, sum(amount) from RDD group by city;

scala> **val empfile1 = sc.textFile("file:/home/hduser/sparkdata/empdata.txt")**

scala> **empfile1.foreach(println)**

ArunKumar~chennai~33~2016-09-20~100000
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
subha~canada~45~2019-02-01~41265
sudhan~canada~10~2019-02-01~25652
shavan~canada~7~2019-02-03~2565

scala> **val empfile2 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")**

scala> **empfile2.foreach(println)**
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000

scala> **val empfulfile = empfile1.union(empfile2)**
empfulfile: org.apache.spark.rdd.RDD[String] = UnionRDD[47] at union at <console>:28

scala> **empfulfile.foreach(println)**
ArunKumar~chennai~33~2016-09-20~100000
subha~canada~45~2019-02-01~41265
sudhan~canada~10~2019-02-01~25652
shavan~canada~7~2019-02-03~2565
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000

scala> **val empSumOfAmt = empfulfile.map{_.split("~")}.map{z=>(z(1),z(4).toInt)}.reduceByKey((x,y) => x+y)**
empSumOfAmt: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[52] at reduceByKey at <console>:30

- now see the result of reduceByKey..

scala> empSumOfAmt.foreach(println)
(chn,1100)
(coimbatore,100000)
(chennai,130000)
(canada,69482)
(banglore,180000)
(mysore,9000)

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 that check point will break the lineage. (after the consecutive action is performed)

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

- Lets create check point for the above recently created RDD empSumOfAmt

sc.setCheckpointDir("/tmp/ckptdir")

(or)

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

empSumOfAmt.checkpoint

scala> **empSumOfAmt.collect**
res25: Array[(String, Int)] = Array((chennai,130000), (chn,1100), (coimbatore,100000), (canada,69482), (banglore,180000), (mysore,9000))

[hduser@tech ~]$ **hadoop fs -ls /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f**
19/06/02 17:42:33 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 4 items
drwxr-xr-x - hduser supergroup 0 2019-06-02 17:36 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-15
drwxr-xr-x - hduser supergroup 0 2019-06-02 17:37 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-17
drwxr-xr-x - hduser supergroup 0 2019-06-02 17:39 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-24
drwxr-xr-x - hduser supergroup 0 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28

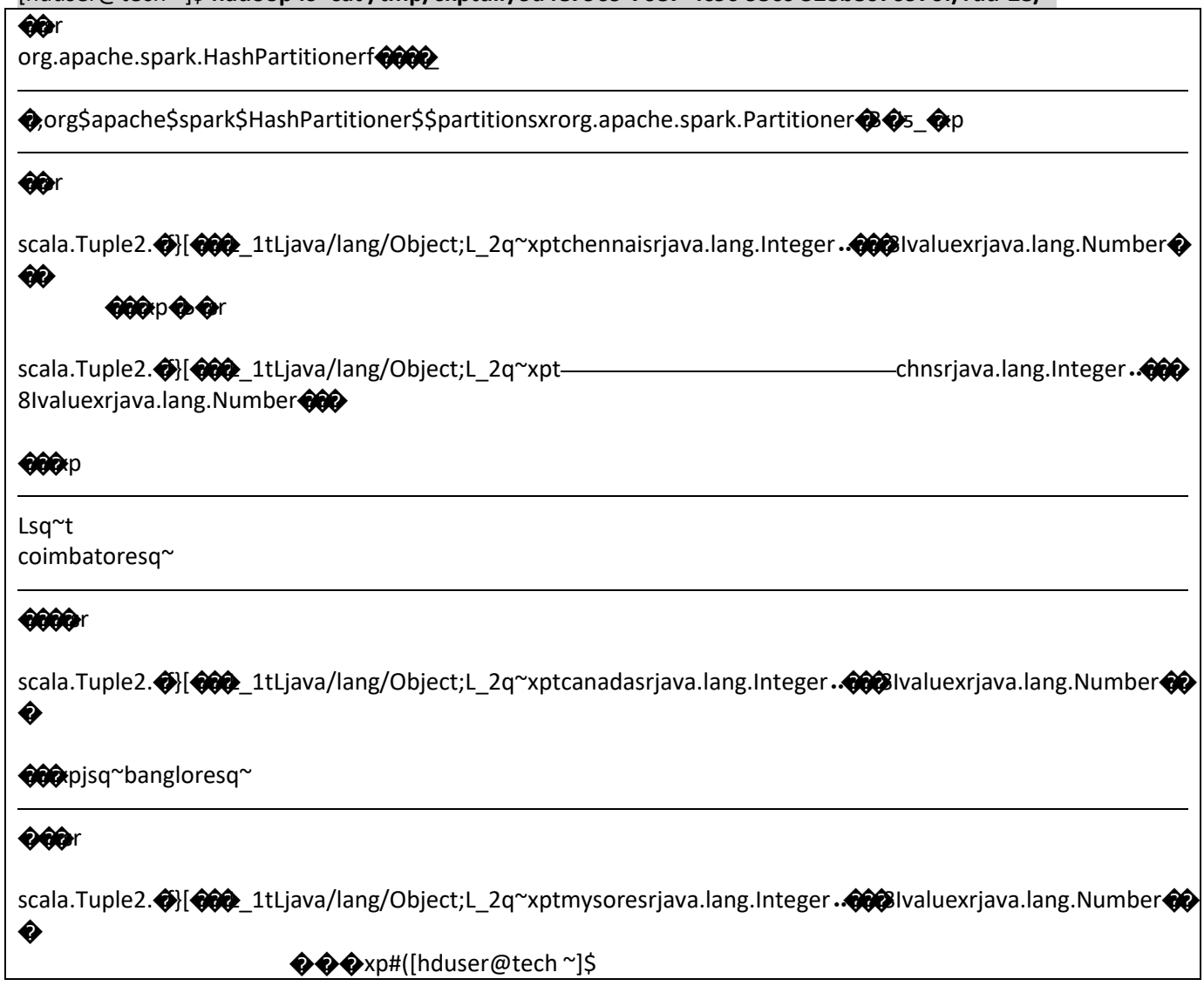
- the above highlighted folder is the recent one which was created after we applied check point on empSumOfAmt

[hduser@tech ~]$ **hadoop fs -ls /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28**
Found 5 items
-rw-r--r-- 1 hduser supergroup 147 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/_partitioner
-rw-r--r-- 1 hduser supergroup 156 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/part-00000
-rw-r--r-- 1 hduser supergroup 181 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/part-00001
-rw-r--r-- 1 hduser supergroup 182 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/part-00002

-rw-r--r-- 1 hduser supergroup 155 2019-06-02 17:42 /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/part-00003

- Note: if we browse the check point file, it's not readable. It's in some machine language format.
- The retention period for this check point file is configurable

[hduser@tech ~]$ **hadoop fs -cat /tmp/ckptdir/9a4e7969-7087-4c9c-95cc-328b8076970f/rdd-28/***



```
r
org.apache.spark.HashPartitionerf
---
,org$apache$spark$HashPartitioner$$partitionsxrorg.apache.spark.Partitioner s_p
---
r

scala.Tuple2.}[_1tLjava/lang/Object;L_2q~xptchennaisrjava.lang.Integer..8Ivaluexrjava.lang.Number

pr

scala.Tuple2.}[_1tLjava/lang/Object;L_2q~xpt————————————chnsrjava.lang.Integer..8Ivaluexrjava.lang.Number

p
---
Lsq~t
coimbatoresq~
---
r

scala.Tuple2.}[_1tLjava/lang/Object;L_2q~xptcanadasrjava.lang.Integer..8Ivaluexrjava.lang.Number

pjsq~bangloresq~
---
r

scala.Tuple2.}[_1tLjava/lang/Object;L_2q~xptmysoresrjava.lang.Integer..8Ivaluexrjava.lang.Number

xp#([hduser@tech ~]$
```

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.

❖ Lets collect the RDD which we created during the above discussion.

scala> **empSumOfAmt.collect**
res25: Array[(String, Int)] = Array((chennai,130000), (chn,1100), (coimbatore,100000), (canada,69482), (banglore,180000), (mysore,9000))

Count :

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

❖ Lets apply count function for the above created RDD.

scala> **empSumOfAmt.count**
res27: Long = 6

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.

Here what we do is,

❖ Take the union applied RDD (combined data of empfile1 and empfile2)
❖ Apply map function to split
❖ Apply map function to take amount alone as tuple
❖ Make it as list by applying collect
❖ At last apply reduce to find the sum of amount.

ReduceByKey	Reduce
• Its **Transformation** which creates another RDD • Its applied on key-value pair • Its similar to select key, sum(value) from RDD group by key	• Its **Action** which produced visible output • Its applied only on a list of value. • This is similar to Select value, sum(value) From RDD Group by value

scala> **val empMulOfAmt = empfulfile.map{_.split("~")}.map{z=>z(4).toInt}.collect.toList**
empMulOfAmt: List[Int] = List(100000, 10000, 10000, 90000, 41265, 25652, 2565, 100, 1000, 9000, 90000, 10000, 100000)

scala> **val empSumOfAmt = empfulfile.map{_.split("~")}.map{z=>z(4).toInt}.collect.toList.reduce((x,y) => x+y)**

❖ the out put is just a variable.. Not a RDD.
❖ Means empSumOfAmt is just a immutable variable of type Int. Not RDD.

scala> **empSumOfAmt**
res35: Int = 489582

scala> **val empMulOfAmt = empfulfile.map{_.split("~")}.map{z=>z(4).toInt}.collect.toList.reduce((x,y) => x-y)**

empMulOfAmt: Int = -289582

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.

- ❖ Lets take these two file RDDs and apply union to make it as bit large RDD.
- ❖ Then apply map functions to take city alone
- ❖ Convert it to list by applying collect function
- ❖ Then countByValue to get he count of the occurrence of each city.
- ❖ The output will be in the form of key-value pair.. (city,count)

ReduceByKey	CountByValue
• Its **Transformation** which creates another RDD • Its applied on key-value pair • Its similar to select key, sum(value) from RDD group by key	• Its **Action** which produced visible output • Its applied only on a list of value. • This is similar to Select value, count(value) From RDD Group by value

scala> **val empfulfile = empfile1.union(empfile2)**
empfulfile: org.apache.spark.rdd.RDD[String] = UnionRDD[47] at union at <console>:28

scala> **empfulfile.foreach(println)**
ArunKumar~chennai~33~2016-09-20~100000
subha~canada~45~2019-02-01~41265
sudhan~canada~10~2019-02-01~25652
shavan~canada~7~2019-02-03~2565
Varun~chennai~34~2016-09-21~10000
Lara~chennai~55~2016-09-21~10000
vasudevan~banglore~43~2016-09-23~90000
ArunKumar~chn~33~2016-09-20~100
srinivasan~chn~33~2016-09-21~1000
vimal~mysore~43~2016-09-23~9000
vasudevan~banglore~39~2016-09-23~90000
Lara~chennai~55~2016-09-21~10000
kannan~coimbatore~31~1987-11-22~100000

val empcity = empfulfile.map{_.split("~")}.map{z=>z(1)}

scala> **empcity.foreach(println)**
chennai
canada
chennai
canada
chennai
canada
banglore

chennai
coimbatore
chn
chn
mysore
banglore

scala> **val emplist = sc.parallelize(empcity.collect.toList)**
emplist: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[61] at parallelize at <console>:32

scala> **emplist.countByValue**
res34: scala.collection.Map[String,Long] = Map(chennai -> 4, mysore -> 1, chn -> 2, banglore -> 2, canada -> 3, coimbatore -> 1)

scala> **res34.foreach(println)**
(chennai,4)
(mysore,1)
(chn,2)
(banglore,2)
(canada,3)
(coimbatore,1)

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.

❖ The function ends with **…ByKey** will be applied on key-value pair.

ReduceByKey	CountByValue	CountByKey
• Its a **Transformation** which creates another RDD • Its applied on key-value pair • Its similar to select key, sum(value) from RDD group by key	• It's an **Action** on list which produced visible output • Its applied only on a list of value. • This is similar to Select value, count(value) From RDD Group by value	• Its an Action on key-value paired RDD. • applied on (k-v) pair RDD. • This is similar to select key, count(value) from RDD group by key

scala> **val empcntbykey = empfulfile.map{_.split("~")}.map{z=>(z(1),z(4).toInt)}**
empcntbykey: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[47] at map at <console>:30

scala> **val cntbykey = empcntbykey.countByKey**
cntbykey: scala.collection.Map[String,Long] = Map(chennai -> 4, mysore -> 1, chn -> 2, banglore -> 2, canada ->

3, coimbatore -> 1)

scala> **cntbykey.foreach(println)**
(chennai,4)
(mysore,1)
(chn,2)
(banglore,2)
(canada,3)
(coimbatore,1)

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.

❖ Lookup is not any aggregate function. Its just like a select statement with where condition.

This is similar to
Select amount
from RDD
Where city = “Canada”

scala> empcntbykey.foreach(println)
(chennai,100000)
(chennai,10000)
(chennai,10000)
(banglore,90000)
(canada,41265)
(canada,25652)
(canada,2565)
(chn,100)
(chn,1000)
(mysore,9000)
(banglore,90000)
(chennai,10000)
(coimbatore,100000)

scala> val lookuprdd = empcntbykey.lookup("canada")
lookuprdd: Seq[Int] = WrappedArray(41265, 25652, 2565)

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.

❖ Lets load the data note log file into a RDD.

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

- and lets filter the warning and error messages alone into another RDD.

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

- now before saving this RDD into a file we need to check whether the targer directory exists already.
- To do that check operation, we need to make use of hadoopConfiguration class from sparkContext.
- Then using hadoopConfiguration class, we can check whether the target part exists.
- It returns a Boolean value as output.

```
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)
```

- if the above command returns false, then it means path doesn't exists.
- So we can proceed saving the file.
- If the above command returns true, it means path exists. So then if we try to save, it will throw ugh error saying file exists.

```
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.

- We have seen the concept of lineage and lazy evaluation.

```
val logs = sc.textFile("file:/usr/local/hadoop/logs/hadoop-hduser-datanode-tech.log")
val errorsAndWarnings = logs filter { l => l.contains("ERROR") || l.contains("WARN")}
```

- when the above commands are executed, the data wont be loaded into logs or filter wont be done, until any action is performed on erorsAndWarnings.

- When the below action is executed, then logs will get data and erorsAndWarnings will get the filtered data, and then cnt will get the count of errorsAndWanings.
- Once the below action is performed, the data in the RDDs logs and errorsAndWarnings will be flushed by the garbage collector.
- So if we again perform the action errorsAndWarnings.count, then again this action will ask data from the previous RDD errosAndWarnings and that RDD will ask data from its previous RDD logs and then logs will lodad the data from the life.
- Finally cnt will get the count again.

```
Val cnt = errorsAndWanings.count
```

- Each and every time when an action is applied, the data flow happens right from the starting of the DAG. Which means data flows right from the initial source till the action.
- Instead if we plan to reuse a particular RDD again and again, we can cache it.

- So that the garbage collector wont flush that particular RDD even after an action is performed.
- Cache is performed as shown below.

```
errorsAndWarnings.cache()
```

- now after performing action, the RDD logs will be flushed out.
- But the RDD errorsAndWarnings will be kept untouched in the memory. Until another command called unpersist() is done on it. We will see about unpersist later.

- Now if we delete the source file "file:/usr/local/hadoop/logs/hadoop-hduser-datanode-tech.log" and if we do the same action, we will be getting the count.
- The count will be pulled from the RDD errorsAndWarnings. Not form the source file or the RDD logs.
-
- If we dint cache the RDD errorsAndWarnings, then while doing the action .count, it will throw an error saying the command logs failed due to source file not exists.

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()
```

- persist() and cache() are the same.
- To clear the cached memory, we need to issue the command 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)
```

in other way of explanation,

- Broadcast variables are useful when large datasets needs to be cached in executors(where task runs)
- Without broadcast variables these variables would be shipped to each task(data note) for every transformation and action, and this can cause network overhead.
- By using broadcast variable, it is shipped to all tasks only once and they are cached there.
- So that for each transformation/action the task don't want to reach out to driver program to get its value reducing the network i/o overhead. It can use its value from cache.
- Broadcast variables are immutable.
- Once the task is completed, we should apply **destroy** function to release cached memory.

After a variable is created with some data as show below,
scala> **val empfile2 = sc.textFile("file:/home/hduser/sparkdata/empdata1.txt")**

we can broadcast this variable as show below.
Scala> **val empfilebroadcast = sc.broadcast(empfile2)**

In the data node where the task runs, the broad casted variable can be accessed as shown below.
Scala> **empfilebroadcast.value**

Once the task is completed, we should release the cached memory by issuing the below command.
Scala> **empfilebroadcast.destroy**

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
```

in other words

- Accumulators are variables that are used for aggregating information across the executors.
- For example, this information can pertain to data or API diagnosis like how many records are corrupted or how many times a particular library API was called.

Lets see this with a simple example.

- Consider we have a job running in distributed fashion.
- When even a step in the job is completed, the job should write its status in its log file.
- Based on the status we need to check the health of the job.
- If the log file is empty after the step is completed, then it means one step is not successful in the job.
- So we need to collect the stats of how many steps failed in a job.
- For that we need to know how many the log file is generated as empty in a job.

For this requirement , if we don't use an accumulator and if we go for a normal variable as show below,

```
Var blanklines:Int = 0

Sc.textFile("hdfs:///user/hduser/job1/log.txt", 4)
        .foreach {
                  Line => if(lines.length() = =0) blanklines += 1
                 }

Println(s"Number of steps failed in job1 =  $blanklines" )
```

- The problem with the above code is that when the driver prints the variable blankLines its value will be zero.
- This is because when Spark ships this code to every executor the variables become local to that executor and its updated value is not relayed back to the driver.
- To avoid this problem we need to make blankLines an accumulator such that all the updates to this variable in every executor is relayed back to the driver.

So as a resolution we can re-write the above code as show below.

```
Var blanklines = sc.broadcast(0,"Blank Lines")

Sc.textFile("hdfs:///user/hduser/job1/log.txt", 4)
        .foreach {
                  Line => if(lines.length() = =0) blanklines += 1
                 }

Println(s"Number of steps failed in job1 =  ${blanklines.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
```