SPARK PAIR RDD JOINING ZIPPING, SORTING AND GROUPING

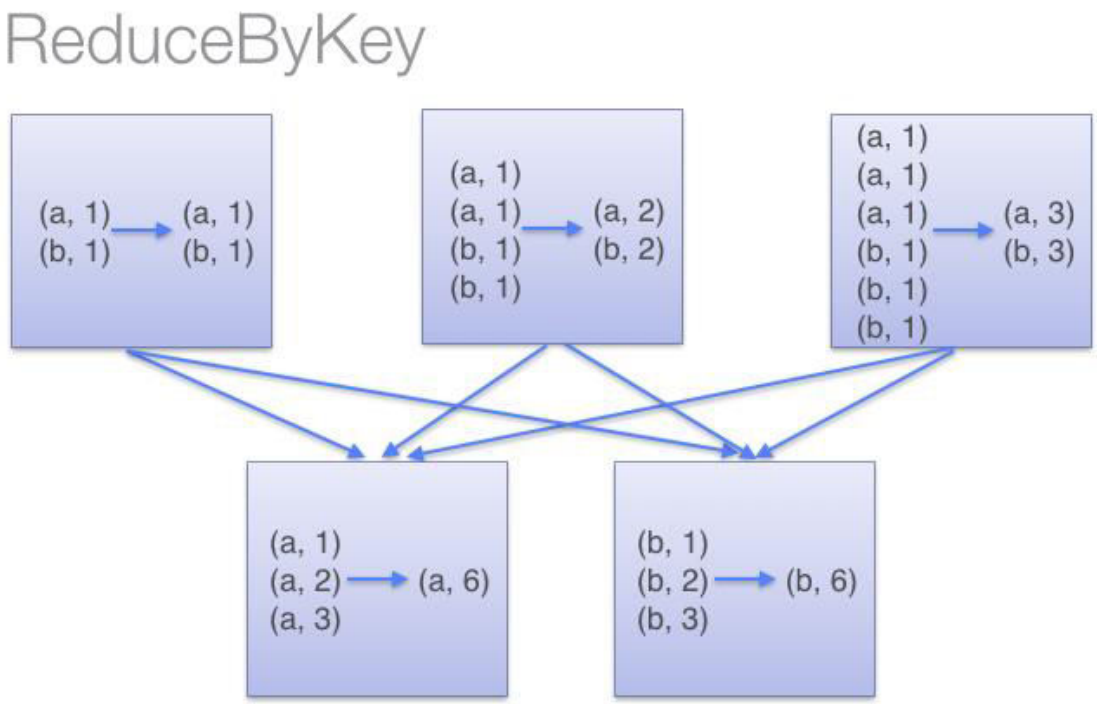
By www.HadoopExam.com

Note: These instructions should be used with the HadoopExam Apache Spark: Professional Trainings. Where it is executed and you can do hands on with trainer.

reduceByKey Versus groupByKey

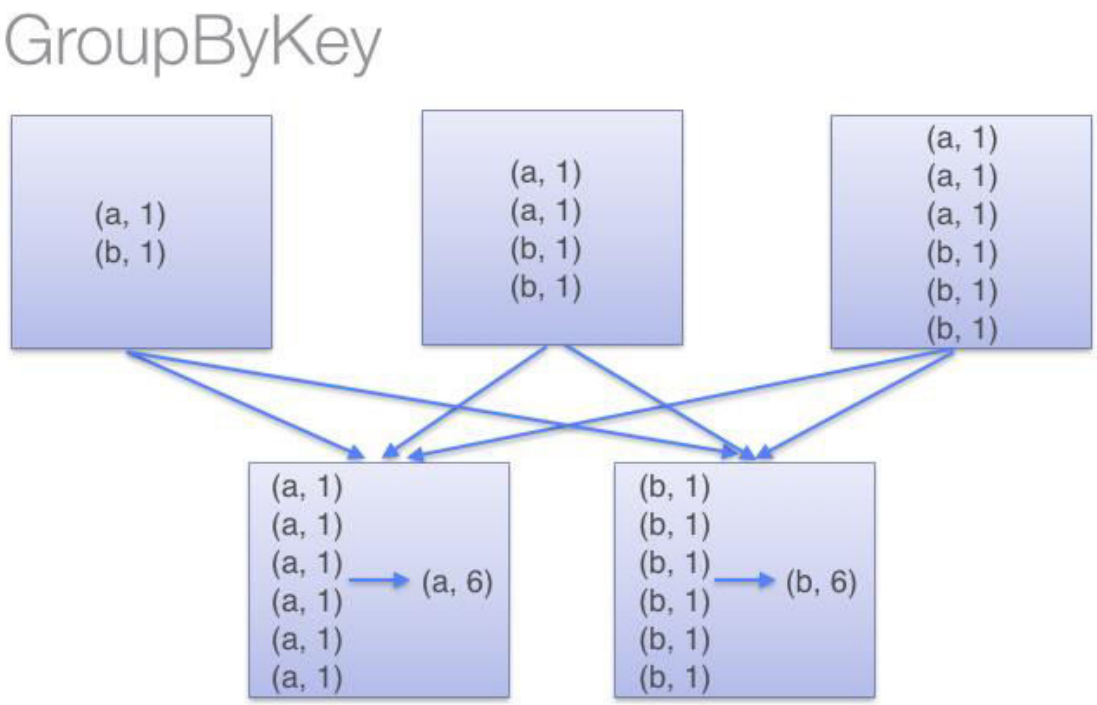
```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD.reduceByKey(_ + _).collect
val wordCountsWithGroup = wordPairsRDD.groupByKey().collect
val wordCountsWithGroup = wordPairsRDD.groupByKey().map(t => (t._1, t._2.sum))
```



To determine which machine to shuffle a pair to, Spark calls a partitioning function on the key of the pair. Spark spills data to disk when there is more data shuffled onto a single executor machine than can fit in memory. However, it flushes out the data to disk one key at a time - so if a single key has more key-value pairs than can fit in memory, an out of memory exception

occurs.



Here are more functions to prefer over `groupByKey`:

- `combineByKey` can be used when you are combining elements but your return type differs from your input value type.
- `foldByKey` merges the values for each key using an associative function and a neutral "zero value".

Cogroup:- It will combine / group All RDDs for the same key.

- Its a generalization of group.

rdd1 → [(a,1), (a,2), (b,3)]

rdd2 → [(a,4), (a,5), (b,6)]

rdd3 → [(a,7), (a,8), (b,9)]

rdd1.cogroup(rdd2, rdd3).collect

⇒ [[a, (1,2,4,5,7,8)], [b, (3,6,9)]]

If one of the RDD does not have elements for a given that is present in the other RDD, the corresponding Iterator is simply Empty.

⇒ Cogroup is used as a building block for the joins and many more operations.

Join:- Joining Datasets (RDDs), similar to we do in RDBMS / Database.

- Left outer join
- Right outer join
- Cross joins
- Inner Joins (only keys that are present in both the RDDs are output.)

Go to Hands on

Sorting Data:- We can sort on RDD with key/value pairs, provided is ordering defined on the key.

⇒ Once we have sorted our data, any subsequent call on the sorted data to collect() or save() will result in ordered data.

⇒ Custom comparator will help us implementing custom sorted order.

cogroup : A very powerful set of functions that allow grouping up to 3 key-value RDDs together using their keys.

```
def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

def cogroup[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W]))]

def cogroup[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (Iterable[V], Iterable[W]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], numPartitions: Int): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]

def cogroup[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)], partitioner: Partitioner): RDD[(K, (Iterable[V], Iterable[W1], Iterable[W2]))]

def groupWith[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

def groupWith[W1, W2](other1: RDD[(K, W1)], other2: RDD[(K, W2)]): RDD[(K, (Iterable[V], IterableW1], Iterable[W2]))]
```

Example:

```
val a = sc.parallelize(List(1, 2, 1, 3), 1)
val b = a.map((_, "b"))
val c = a.map((_, "c"))
b.cogroup(c).collect

//Array[(Int, (Iterable[String], Iterable[String]))] = Array(
//(2,(ArrayBuffer(b),ArrayBuffer(c))),
//(3,(ArrayBuffer(b),ArrayBuffer(c))),
//(1,(ArrayBuffer(b, b),ArrayBuffer(c, c)))
//)


val d = a.map((_, "d"))
b.cogroup(c, d).collect

val x = sc.parallelize(List((1, "apple"), (2, "banana"), (3, "orange"), (4, "kiwi")), 2)
val y = sc.parallelize(List((5, "computer"), (1, "laptop"), (1, "desktop"), (4, "iPad")), 2)
x.cogroup(y).collec
```

Join: Performs an inner join using two key-value RDDs. Please note that the keys must be generally comparable to make this work.

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]
def join[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V, W))]
```

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.join(d).collect

//res0: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)),
//(6,(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)),
//(3,(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)),
//(3,(rat,gnu)), (3,(rat,bee)))
```

leftOuterJoin : Performs an left outer join using two key-value RDDs. Please note that the keys must be generally comparable to make this work correctly.

```
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
def leftOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V,
Option[W]))]
def leftOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K, (V,
Option[W]))]
```

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.leftOuterJoin(d).collect

//res1: Array[(Int, (String, Option[String]))] = Array((6,(salmon,Some(salmon))),
```

```
//(6,(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (6,(salmon,Some(salmon))),
//(6,(salmon,Some(rabbit))), (6,(salmon,Some(turkey))), (3,(dog,Some(dog))),
//(3,(dog,Some(cat))), (3,(dog,Some(gnu))), (3,(dog,Some(bee))), (3,(rat,Some(dog))),
//(3,(rat,Some(cat))), (3,(rat,Some(gnu))), (3,(rat,Some(bee))), (8,(elephant,None)))
```

rightOuterJoin : Performs an right outer join using two key-value RDDs. Please note that the keys must be generally comparable to make this work correctly.

```
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]
def rightOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V],
W))]
def rightOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K,
(Option[V], W))]
```

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c =
sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","bear","bee"),
3)
val d = c.keyBy(_.length)
b.rightOuterJoin(d).collect

res2: Array[(Int, (Option[String], String))] = Array((6,(Some(salmon),salmon)),
(6,(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (6,(Some(salmon),salmon)),
(6,(Some(salmon),rabbit)), (6,(Some(salmon),turkey)), (3,(Some(dog),dog)),
(3,(Some(dog),cat)), (3,(Some(dog),gnu)), (3,(Some(dog),bee)), (3,(Some(rat),dog)),
(3,(Some(rat),cat)), (3,(Some(rat),gnu)), (3,(Some(rat),bee)), (4,(None,wolf)),
(4,(None,bear)))
```

fullOuterJoin: Performs the full outer join between two paired RDDs.

```
def fullOuterJoin[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (Option[V],
Option[W]))]
def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]
def fullOuterJoin[W](other: RDD[(K, W)], partitioner: Partitioner): RDD[(K,
(Option[V], Option[W]))]
```

```
val pairRDD1 = sc.parallelize(List( ("cat",2), ("cat", 5), ("book", 4),("cat", 12)))
```

```
val pairRDD2 = sc.parallelize(List( ("cat",2), ("cup", 5), ("mouse", 4),("cat", 12)))
pairRDD1.fullOuterJoin(pairRDD2).collect

//res5: Array[(String, (Option[Int], Option[Int]))] = Array((book,(Some(4),None)),
//(mouse,(None,Some(4))), (cup,(None,Some(5))), (cat,(Some(2),Some(2))),
//(cat,(Some(2),Some(12))), (cat,(Some(5),Some(2))), (cat,(Some(5),Some(12))),
//(cat,(Some(12),Some(2))), (cat,(Some(12),Some(12)))
```

Zip : Joins two RDDs by combining the i-th of either partition with each other. The resulting RDD will consist of two-component tuples which are interpreted as key-value pairs by the methods provided by the PairRDDFunctions extension.

```
def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]
```

```
val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
a.zip(b).collect
res1: Array[(Int, Int)] = Array((1,101), (2,102), (3,103), (4,104), (5,105), (6,106),
(7,107), (8,108), (9,109), (10,110), (11,111), (12,112), (13,113), (14,114), (15,115),
(16,116), (17,117), (18,118), (19,119), (20,120), (21,121), (22,122), (23,123),
(24,124), (25,125), (26,126), (27,127), (28,128), (29,129), (30,130), (31,131),
(32,132), (33,133), (34,134), (35,135), (36,136), (37,137), (38,138), (39,139),
(40,140), (41,141), (42,142), (43,143), (44,144), (45,145), (46,146), (47,147),
(48,148), (49,149), (50,150), (51,151), (52,152), (53,153), (54,154), (55,155),
(56,156), (57,157), (58,158), (59,159), (60,160), (61,161), (62,162), (63,163),
(64,164), (65,165), (66,166), (67,167), (68,168), (69,169), (70,170), (71,171),
(72,172), (73,173), (74,174), (75,175), (76,176), (77,177), (78,...
```

```
val a = sc.parallelize(1 to 100, 3)
val b = sc.parallelize(101 to 200, 3)
val c = sc.parallelize(201 to 300, 3)
a.zip(b).zip(c).map((x) => (x._1._1, x._1._2, x._2 )).collect
res12: Array[(Int, Int, Int)] = Array((1,101,201), (2,102,202), (3,103,203),
(4,104,204), (5,105,205), (6,106,206), (7,107,207), (8,108,208), (9,109,209),
(10,110,210), (11,111,211), (12,112,212), (13,113,213), (14,114,214), (15,115,215),
(16,116,216), (17,117,217), (18,118,218), (19,119,219), (20,120,220), (21,121,221),
(22,122,222), (23,123,223), (24,124,224), (25,125,225), (26,126,226), (27,127,227),
```

```
(28,128,228), (29,129,229), (30,130,230), (31,131,231), (32,132,232), (33,133,233),
(34,134,234), (35,135,235), (36,136,236), (37,137,237), (38,138,238), (39,139,239),
(40,140,240), (41,141,241), (42,142,242), (43,143,243), (44,144,244), (45,145,245),
(46,146,246), (47,147,247), (48,148,248), (49,149,249), (50,150,250), (51,151,251),
(52,152,252), (53,153,253), (54,154,254), (55,155,255)...
```

sortByKey: This function sorts the input RDD's data and stores it in a new RDD. The output RDD is a shuffled RDD because it stores data that is output by a reducer which has been shuffled. The implementation of this function is actually very clever. First, it uses a range partitioner to partition the data in ranges within the shuffled RDD. Then it sorts these ranges individually with mapPartitions using standard sort mechanisms.

```
def sortByKey(ascending: Boolean = true, numPartitions: Int = self.partitions.size):
RDD[P]
```

```
val a = sc.parallelize(List("dog", "cat", "owl", "gnu", "ant"), 2)
val b = sc.parallelize(1 to a.count.toInt, 2)
val c = a.zip(b)
c.sortByKey(true).collect
res74: Array[(String, Int)] = Array((ant,5), (cat,2), (dog,1), (gnu,4), (owl,3))
c.sortByKey(false).collect
//res75: Array[(String, Int)] = Array((owl,3), (gnu,4), (dog,1), (cat,2), (ant,5))

val a = sc.parallelize(1 to 100, 5)
val b = a.cartesian(a)
val c = sc.parallelize(b.takeSample(true, 5, 13), 2)
val d = c.sortByKey(false)
//res56: Array[(Int, Int)] = Array((96,9), (84,76), (59,59), (53,65), (52,4))
```