

Transformations and Actions on Pair RDDs

Big Data Analysis with Scala and Spark

Heather Miller

Some interesting Pair RDDs operations

Important operations defined on Pair RDDs:
(But not available on regular RDDs)

Transformations

- ▶ groupByKey
- ▶ reduceByKey
- ▶ mapValues
- ▶ keys
- ▶ join
- ▶ leftOuterJoin/rightOuterJoin

Action

- ▶ countByKey

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections.

Pair RDD Transformation: ~~groupByKey~~

Recall `groupBy` from Scala collections.

```
def groupBy[K](f: A => K): Map[K, Traversable[A]]
```

Partitions this traversable collection into a map of traversable collections according to some discriminator function.

In English: Breaks up a collection into two or more collections according to a function that you pass to it. Result of the function is the key, the collection of results that return that key when the function is applied to it. Returns a Map mapping computed keys to collections of corresponding values.

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections.

```
def groupBy[K](f: A => K): Map[K, Traversable[A]]
```

Example:

Let's group the below list of ages into "child", "adult", and "senior" categories.

```
val ages = List(2, 52, 44, 23, 17, 14, 12, 82, 51, 64)
val grouped = ages.groupBy { age =>
    if (age >= 18 && age < 65) "adult"
    else if (age < 18) "child"
    else "senior"
}
// grouped: scala.collection.immutable.Map[String,List[Int]] =
// Map(senior -> List(82), adult -> List(52, 44, 23, 51, 64),
// child -> List(2, 17, 14, 12))
```

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections. groupByKey can be thought of as a groupBy on Pair RDDs that is specialized on grouping all values that have the same key. As a result, it takes no argument.

```
def groupByKey(): RDD[(K, Iterable[V])]
```

Pair RDD Transformation: groupByKey

Recall groupBy from Scala collections. groupByKey can be thought of as a groupBy on Pair RDDs that is specialized on grouping all values that have the same key. As a result, it takes no argument.

```
def groupByKey(): RDD[(K, Iterable[V])]
```

Example:

```
case class Event(organizer: String, name: String, budget: Int)
```

```
val eventsRdd = sc.parallelize(...)  
    .map(event => (event.organizer, event.budget))
```

```
val groupedRdd = eventsRdd.groupByKey()
```

Here the key is organizer. What does this call do?

Pair RDD Transformation: groupByKey

Example:

```
case class Event(organizer: String, name: String, budget: Int)
val eventsRdd = sc.parallelize(...)
    .map(event => (event.organizer, event.budget))

val groupedRdd = eventsRdd.groupByKey()

// TRICK QUESTION! As-is, it "does" nothing. It returns an unevaluated RDD

groupedRdd.collect().foreach(println)
// (Prime Sound,CompactBuffer(42000))
// (Sportorg,CompactBuffer(23000, 12000, 1400))
// ...
```

Pair RDD Transformation: reduceByKey

Conceptually, `reduceByKey` can be thought of as a combination of `groupByKey` and `reduce-ing` on all the values per key. It's more efficient though, than using each separately. (We'll see why later.)

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

Pair RDD Transformation: reduceByKey

Conceptually, reduceByKey can be thought of as a combination of groupByKey and reduce-ing on all the values per key. It's more efficient though, than using each separately. (We'll see why later.)

```
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
```

Example: Let's use eventsRdd from the previous example to calculate the total budget per organizer of all of their organized events.

```
case class Event(organizer: String, name: String, budget: Int)
val eventsRdd = sc.parallelize(...)
    .map(event => (event.organizer, event.budget))

val budgetsRdd = ...
```

Pair RDD Transformation: reduceByKey

Example: Let's use eventsRdd from the previous example to calculate the total budget per organizer of all of their organized events.

```
case class Event(organizer: String, name: String, budget: Int)  
val eventsRdd = sc.parallelize(...)  
    .map(event => (event.organizer, event.budget))
```

```
val budgetsRdd = eventsRdd.reduceByKey(_+_)
```

```
reducedRdd.collect().foreach(println)  
// (Prime Sound,42000)  
// (Sportorg,36400)  
// (Innotech,32000)  
// (Association Balélec,50000)
```

Pair RDD Transformation: `mapValues` and Action: `countByKey`

`mapValues` (`def mapValues[U](f: V => U): RDD[(K, U)]`) can be thought of as a short-hand for:

```
rdd.map { case (x, y): (x, func(y))}
```

That is, it simply applies a function to only the values in a Pair RDD.

`countByKey` (`def countByKey(): Map[K, Long]`) simply counts the number of elements per key in a Pair RDD, returning a normal Scala Map (remember, it's an action!) mapping from keys to counts.

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)  
val intermediate = ??? // Can we use countByKey?
```

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey( $\lambda$ )
```

$(\underset{K}{\text{org}}, \underset{V}{\text{budget}}) \rightarrow \underline{(\text{org}, (\text{budget}, 1))}$

Result should look like:

$(\text{org}, (\text{total Budget}, \text{total \# events organized}))$

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]
```

The diagram illustrates the transformation of an RDD of events into an intermediate RDD. It starts with an RDD of events, represented by a blue arrow pointing to the first line of code. This RDD is transformed into an intermediate RDD where each event is mapped to a pair consisting of its budget and a count of 1. A blue arrow points from the first line of code to this step. The resulting RDD is then reduced by key, where the values are summed. A blue arrow points from the second line of code to this step. The final result is an RDD where each key is a tuple of two integers: the total budget and the total number of events. Handwritten annotations in blue ink label these components: '(budget, 1)' at the top right, 'budgets' under the first component of the sum, and 'total # events' under the second component.

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]

val avgBudgets = ???
```

Pair RDD Transformation: mapValues and Action: countByKey

Example: we can use each of these operations to compute the average budget per event organizer, if possible.

```
// Calculate a pair (as a key's value) containing (budget, #events)
val intermediate =
  eventsRdd.mapValues(b => (b, 1))
    .reduceByKey((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2))
// intermediate: RDD[(String, (Int, Int))]

val avgBudgets = intermediate.mapValues {
  case (budget, numberEvents) => budget / numberEvents
}
avgBudgets.collect().foreach(println)
// (Prime Sound,42000)
// (Sportorg,12133)
// (Innotech,106666)
// (Association Balélec,50000)
```

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Example: we can count the number of unique visitors to a website using the keys transformation.

```
case class Visitor(ip: String, timestamp: String, duration: String)
val visits: RDD[Visitor] = sc.textfile(...)
  .map(v => (v.ip, v.duration))

val numUniqueVisits = ???
```

Pair RDD Transformation: keys

keys (def keys: RDD[K]) Return an RDD with the keys of each tuple.

Note: this method is a transformation and thus returns an RDD because the number of keys in a Pair RDD may be unbounded. It's possible for every value to have a unique key, and thus it may not be possible to collect all keys at one node.

Example: we can count the number of unique visitors to a website using the keys transformation.

```
case class Visitor(ip: String, timestamp: String, duration: String)
val visits: RDD[Visitor] = sc.textfile(...)
    .map( v => (v.ip, v.duration))
val numUniqueVisits = visits.keys.distinct().count()
// numUniqueVisits: Long = 3391
```

PairRDDFunctions

For a list of all available specialized Pair RDD operations, see the Spark API page for `PairRDDFunctions` (ScalaDoc):

<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions>

The screenshot shows the ScalaDoc interface for the `PairRDDFunctions` class. At the top, there's a navigation bar with a logo, the package name `org.apache.spark.rdd`, the class name `PairRDDFunctions`, and a link to the `Related Doc: package rdd`. Below the header, the class definition is shown: `class PairRDDFunctions[K, V] extends Logging with Serializable`. A note below it states: "Extra functions available on RDDs of (key, value) pairs through an implicit conversion." Navigation links include "Source" (leading to `PairRDDFunctions.scala`) and "Linear Supertypes". The main content area has a search bar and filters for "Ordering" (set to "Alphabetic"), "Inherited" (listing `PairRDDFunctions`, `Serializable`, `Serializable`, `Logging`, `AnyRef`, `Any`), and "Visibility" (set to "Public"). A section titled "Instance Constructors" contains the constructor definition: `new PairRDDFunctions(self: RDD[(K, V)])(implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null)`. The "Value Members" section lists several methods, each with a brief description:

- `aggregateByKey[U](zeroValue: U)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".
- `aggregateByKey[U](zeroValue: U, numPartitions: Int)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".
- `aggregateByKey[U](zeroValue: U, partitions: Partitioner)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`
Aggregate the values of each key, using given combine functions and a neutral "zero value".