ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

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: groupBy~~Key~~

Recall groupBy from Scala collections.

```scala
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: groupBy~~Key~~

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.

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

Example:

```scala
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.)

```scala
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.

```scala
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,320000)
// (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(
```

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

Result should look like:

(org, (totalBudget, 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.

(budget, 1)

```
// 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))]
```

budgets

total # events

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, numberOfEvents) => budget / numberOfEvents
}
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 is 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 is 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 is may not be possible to collect all keys at one node.

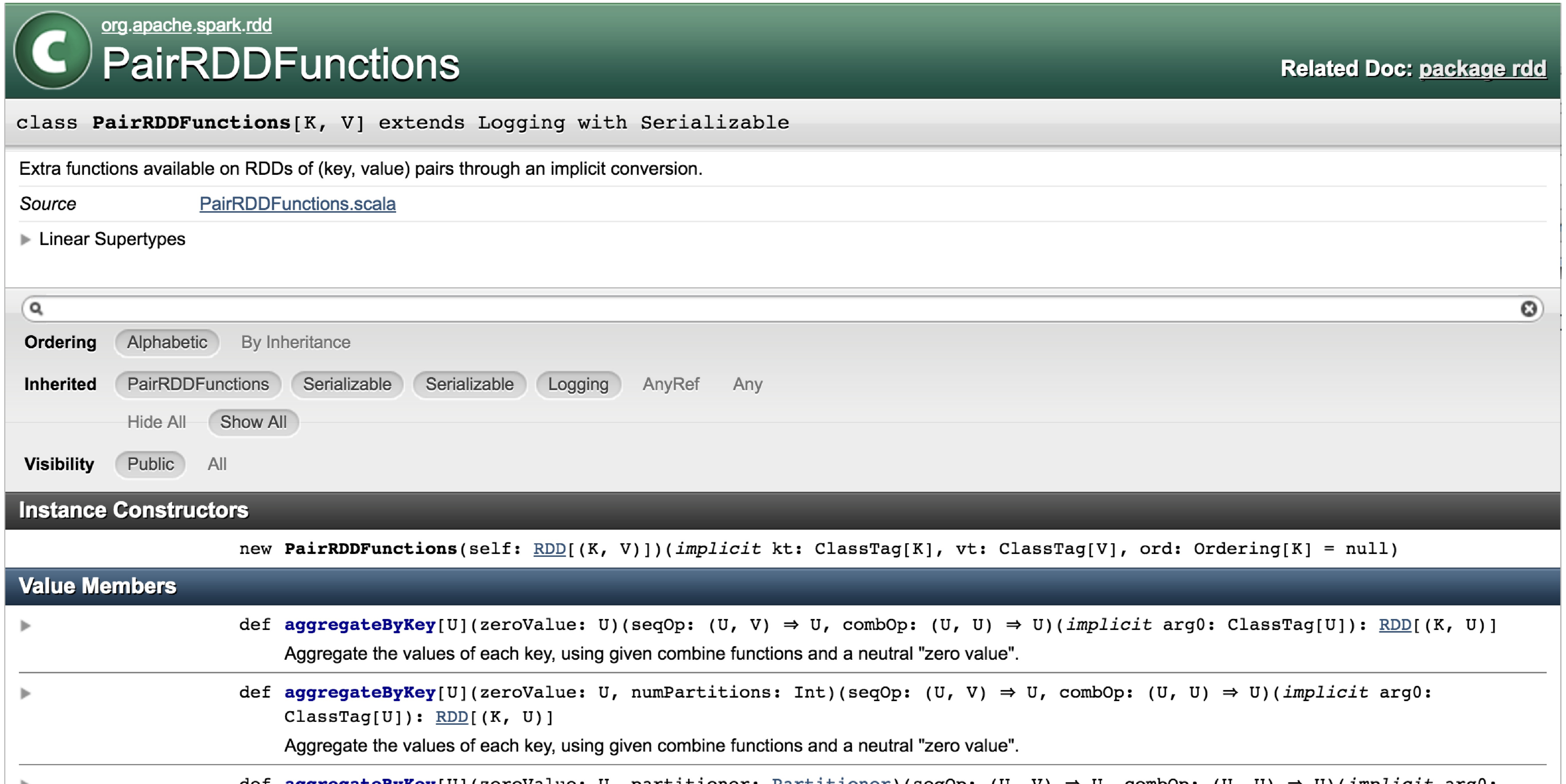
Example: we can count the number of unique visitors to a website using the `keys` transformation.

```scala
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



org.apache.spark.rdd

PairRDDFunctions

Related Doc: package rdd

```
class PairRDDFunctions[K, V] extends Logging with Serializable
```

Extra functions available on RDDs of (key, value) pairs through an implicit conversion.

Source PairRDDFunctions.scala

▸ Linear Supertypes

Ordering Alphabetic By Inheritance
Inherited PairRDDFunctions Serializable Serializable Logging AnyRef Any
Hide All Show All
Visibility Public All

Instance Constructors

```
new PairRDDFunctions(self: RDD[(K, V)])(implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null)
```

Value Members

```
def 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".

```
def 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".

```
def aggregateByKey[U](zeroValue: U, partitioner: Partitioner)(seqOp: (U, V) ⇒ U, combOp: (U, U) ⇒ U)(implicit arg0:
```