

Spark

Resilient Distributed Datasets (RDD)

klucz-wartość

Krzysztof Jankiewicz

Plan

- Typy RDD
- PairRDDFunctions – RDD par i jego znaczenie
- Metody tworzące RDD par
- Metody przetwarzające pojedyncze RDD par
- Łączenie RDD par

Typy RDD

- Spark wykorzystuje wiele typów RDD, które pojawiają się w wyniku przetwarzania RDD za pomocą określonych metod, lub wynikają ze struktury rekordów RDD
- Przykładowe typy RDD:
 - `ParallelCollectionRDD` – wynik działania metody `SparkContext.parallelize`

```
val rddOfInts = sc.parallelize(1 to 200 map scala.util.Random.nextInt)
rddOfInts: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[143] at parallelize at <console>:30
```

- `MapPartitionsRDD` – wynik operacji: `map`, `flatMap`, `mapPartitions`, `filter` itp.

```
val rddOfInts2 = rddOfInts.map(x => (x, 2*3.14*x) )
rddOfInts2: org.apache.spark.rdd.RDD[(Int, Double)] = MapPartitionsRDD[146] at map at <console>:32
```

- `ShuffledRDD` - wynik operacji przenoszenia (shuffle) zbiorów danych pomiędzy węzłami, np. podczas repartycjonowania lub scalania partycji


```
val rddOfInts3 = rddOfInts2.groupByKey()
rddOfInts3: org.apache.spark.rdd.RDD[(Int, Iterable[Double])] = ShuffledRDD[149] at groupByKey at <console>:34
```

- Na szczególną uwagę zasługuje typ `PairRDD` (`PairRDDFunctions`), który powszechnie występuje podczas bardziej złożonego przetwarzania
 - pojawia się on w sytuacji kiedy rekord RDD jest parą wartości, interpretowaną jako para (klucz, wartość)
 - istnieje wiele metod RDD odnoszących się tylko i wyłącznie do RDD par.

PairRDDFunctions

- Operacje na parach klucz-wartość to **podstawa** przetwarzania Big Data
MapReduce: Simplified Data Processing on Large Clusters; Jeffrey Dean and Sanjay Ghemawat; Google; 2004
- PairRDDFunctions określany jest zazwyczaj jako **RDD par**
- RDD par "pojawia" się **automatycznie** w przypadku RDD[(K,V)]

```
scala> val gamesByGenreRdd = gameInfosRdd.map(gi => (gi.genre, gi))
scala> :type gamesByGenreRdd
org.apache.spark.rdd.RDD[(String, GameInfo)]
```

- RDD par pozwala na przetwarzanie wartości powiązanych z określoną wartością klucza niezależnie i **równolegle**
- Istnieje wiele **metod**, które związane są tylko z RDD par  *akcja*
 - groupByKey([numTasks])
 - reduceByKey(func)
 - join(otherDataset, [numTasks])
 - cogroup(otherDataset, [numTasks])
 - aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])
 - sortByKey([ascending], [numTasks])
 - countByKey()
 - mapValues(func)
 - lookup(key)
 - keys()
 - values()
- Wiele **zagadnień** związanych z **wydajnością** przetwarzania jest powiązane z tym typem danych: partycjonowanie, przenoszenie (shuffling), zależności pomiędzy obliczeniami, tolerancja na awarie, odtwarzanie obliczeń w przypadku awarii

Metody tworzące RDD par

- `def groupBy[K](f: (T) => K): RDD[(K, Iterable[T])]`
- `def keyBy[K](f: (T) => K): RDD[(K, T)]`
- `def map[(K,V)](f: (T) => (K,V)): RDD[(K,V)]`

```
gameInfosRdd.groupBy(gi => gi.genre).  
  mapValues(gis => gis.aggregate(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my))
```

```
gameInfosRdd.map(gi => (gi.genre,gi)).  
  aggregateByKey(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my)
```

```
gameInfosRdd.keyBy(gi => gi.genre).  
  aggregateByKey(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my)
```

```
scala> :type gameInfosRdd  
org.apache.spark.rdd.RDD[GameInfo]
```

Jakiego typu są:

- `gameInfosRdd.groupBy(gi => gi.genre)`
- `gameInfosRdd.map(gi => (gi.genre,gi))`
- `gameInfosRdd.keyBy(gi => gi.genre)`

```
case class GameInfo (  
  score_phrase: String,  
  title: String,  
  url: String,  
  platform: String,  
  score: Double,  
  genre: String,  
  editors_choice: String,  
  release_year: Integer,  
  release_month: Integer,  
  release_day: Integer  
);
```

Metody przetwarzające pojedynczy RDD par

- `groupByKey([numTasks])` – wywoływany na parach (K,V) zwraca dla każdej unikalnej wartości klucza K (K, Iterable<V>)
- `reduceByKey(func)` – wywoływany na parach (K,V) zwraca dla każdej wartości klucza K wynik postaci (K,V), gdzie wynikowy V obliczany jest na podstawie *func* o formacie (V,V) -> V
- `sortByKey([ascending], [numTasks])` – uruchamiane na parach (K,V) daje w wyniku RDD będący parami (K,V) posortowanymi względem klucza K, który musi być typu Ordered.
- `aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])` – uruchamiane na parach (K, V), daje w wyniku RDD będący parami (K, U), w których wartości dla każdego klucza wyznaczone są w oparciu o wartość początkową U, funkcję agregującą wartości V z wartościami pośrednimi U ((U,V)=>U) oraz funkcję łączącą wartości pośrednie ((U,U)=>U).

```
gameInfosRdd.keyBy(gi => gi.genre).  
  aggregateByKey(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my)
```

```
gameInfosRdd.map(gi => (gi.genre,gi)).groupByKey.  
  mapValues(gis => gis.aggregate(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my))
```

```
gameInfosRdd.map(gi => (gi.genre,gi.score)).reduceByKey((mx,my) => mx + my)
```

Metody przetwarzające pojedynczy RDD par

- `def mapValues[U](f: (V) => U): RDD[(K, U)]` – mapowanie dotyczy tylko wartości
- `def lookup(key: K): Seq[V]` – wydobywa wartości powiązane z kluczem
- `def keys: RDD[K]` – tworzy RDD składające się z samych kluczy
- `def values: RDD[V]` – tworzy RDD składające się z samych wartości

Akcje:

- `def countByKey(): Map[K, Long]` – tworzy lokalną mapę zawierającą dla każdego klucza liczbę wystąpień
- `def collectAsMap(): Map[K, V]` – tworzy lokalną mapę

```
gameInfosRdd.map(gi => (gi.genre,gi)).groupByKey.  
  mapValues(gis => gis.aggregate(0.0)((m,gi) => m + gi.score, (mx,my) => mx + my))
```

```
gameInfosRdd.keyBy(gi => gi.genre).countByKey()
```

```
gameInfosRdd.keyBy(gi => gi.platform). . . .
```

Dokończ powyższy fragment kodu aby wyznaczyć liczbę platform objętych recenzjami gier

Do tej pory liczyliśmy sumy ocen...
Jak wyglądałoby obliczenie średniej
Podpowiedź: zastosuj metody: `mapValues` (może nie raz?),
`reduceByKey`.

`countByKey` jest akcją – punktem końcowym przetwarzania

Jak wyglądałaby transformacja wyliczająca dokładnie to samo, ale pozostawiająca dane w postaci RDD?

Połączenia

- Nie zawsze wymagane obliczenia da się przeprowadzić wykonując sekwencję transformacji
- Bywają przypadki, w których:
 - przetwarzanie trzeba rozwidlić
 - wynik przetwarzania oparty jest o wiele zbiorów RDD
- W każdym z takich przypadków pomocne są połączenia
- Połączenia
 - działają analogicznie jak w przypadku relacji
 - oparte są na kluczu, który musi wystąpić (i być kompatybilny) w obu łączonych zbiorach RDD
- Metody:
 - `def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`
 - `def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]`
 - `def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]`
 - `def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]`
 - `def cogroup[W](other: RDD[(K, W)]): 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, W3]. . .`

Czy wynikiem operacji `join`, `cogroup` są także RDD par?

Co jest wartością w każdym z przypadków?

W przypadku wyniku których metod wartość klucza może się powtarzać?

Połączenia – przykłady

```
val ps4rdd = gameInfosRdd.filter(gi => gi.platform == "PlayStation 4").
    keyBy(gi => gi.release_year) // RDD[(Integer, GameInfo)]
val wiiUrdd = gameInfosRdd.filter(gi => gi.platform == "Wii U").
    keyBy(gi => gi.release_year) // RDD[(Integer, GameInfo)]
val xb0nerdd = gameInfosRdd.filter(gi => gi.platform == "Xbox One").
    keyBy(gi => gi.release_year) // RDD[(Integer, GameInfo)]
```

```
val alljoin = ps4rdd.join(wiiUrdd).join(xb0nerdd) // Jaki jest typ wyniku?
```

```
val allouterjoin = ps4rdd.fullOuterJoin(wiiUrdd).fullOuterJoin(xb0nerdd) // A tu?
```

```
val allcogroup2 = ps4rdd.cogroup(wiiUrdd).cogroup(xb0nerdd) // A tu?
```

```
val allcogroup3 = ps4rdd.cogroup(wiiUrdd,xb0nerdd)
```

```
val allval =
    ps4rdd.mapValues(gi => 1).reduceByKey((x, y) => x + y).
    join(wiiUrdd.mapValues(gi => 1).reduceByKey((x, y) => x + y)).
    join(xb0nerdd.mapValues(gi => 1).reduceByKey((x, y) => x + y)).
    mapValues . . .
```

```
allval.sortByKey(true).collect().
    foreach(println)
(2013,(34,44,23))
(2014,(84,23,59))
(2015,(97,13,85))
(2016,(61,6,41))
```

Co wpisać zamiast ...

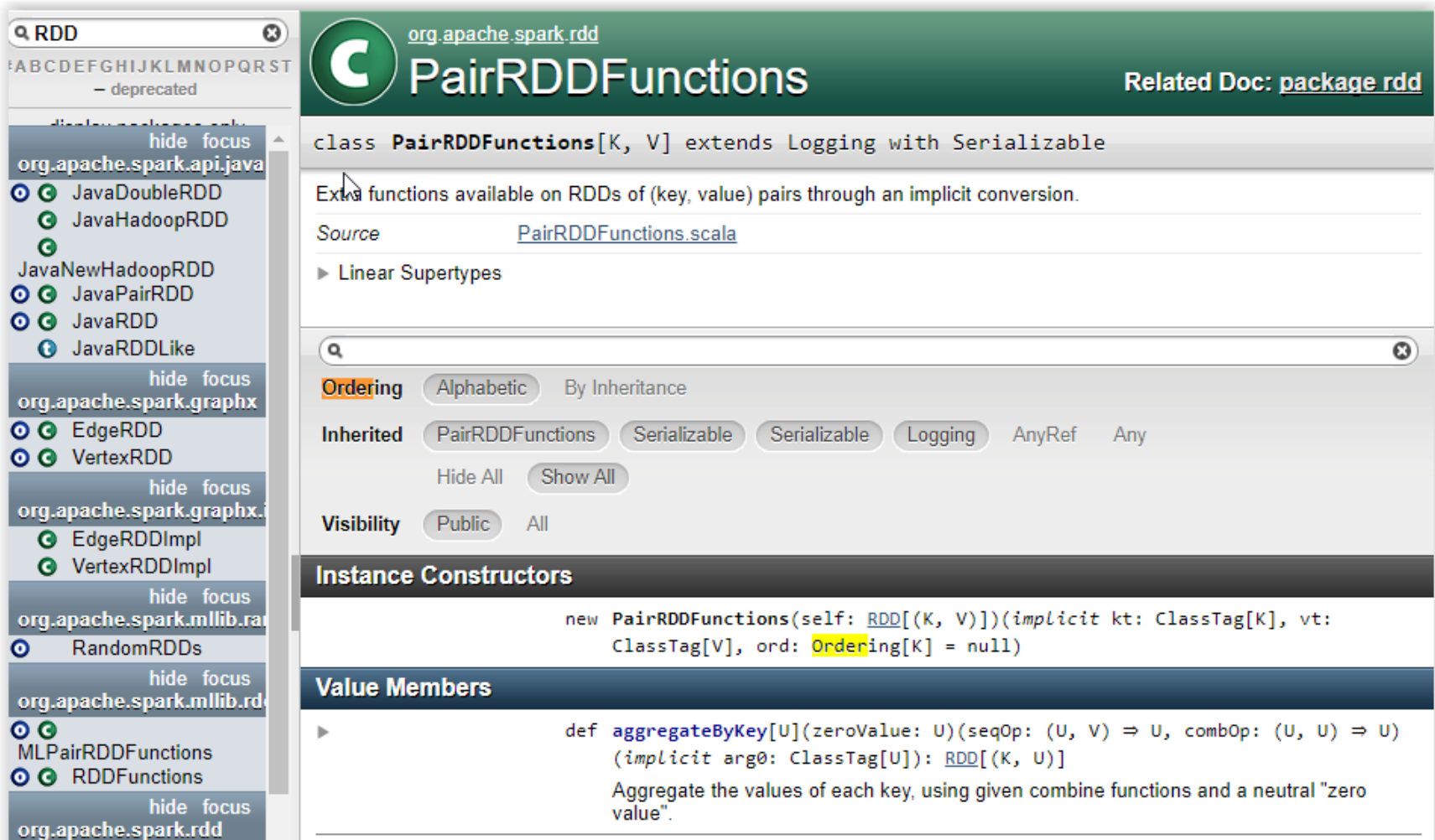
aby wyświetlanie informacji zadziałało poprawnie?

Jak powinny wyglądać transformacje alljoin aby utworzyły analogiczną zawartość jak allval?

A w przypadku allouterjoin albo allcogroup2?

Spark DocAPI

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



The screenshot displays the Spark DocAPI for the `PairRDDFunctions` class. On the left, a sidebar shows a search bar with "RDD" and a list of packages including `org.apache.spark.api.java`, `org.apache.spark.graphx`, `org.apache.spark.mllib.rdd`, and `org.apache.spark.rdd`. The main content area features a green header with the Spark logo and the text "org.apache.spark.rdd PairRDDFunctions". A "Related Doc: package rdd" link is present. The class definition is shown as `class PairRDDFunctions[K, V] extends Logging with Serializable`. Below this, a description states: "Extra functions available on RDDs of (key, value) pairs through an implicit conversion." The source file is `PairRDDFunctions.scala`. A section for "Linear Supertypes" is visible. A filter section includes tabs for "Ordering" (selected), "Alphabetic", and "By Inheritance", and a list of "Inherited" traits: `PairRDDFunctions`, `Serializable`, `Logging`, `AnyRef`, and `Any`. The "Visibility" section shows "Public" and "All". The "Instance Constructors" section displays the constructor: `new PairRDDFunctions(self: RDD[(K, V)])(implicit kt: ClassTag[K], vt: ClassTag[V], ord: Ordering[K] = null)`. The "Value Members" section shows the `aggregateByKey` method: `def aggregateByKey[U](zeroValue: U)(seqOp: (U, V) => U, combOp: (U, U) => U)(implicit arg0: ClassTag[U]): RDD[(K, U)]`, followed by a description: "Aggregate the values of each key, using given combine functions and a neutral 'zero value'."

Search: RDD

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".

Podsumowanie

- Typy RDD
- PairRDDFunctions – RDD par i jego znaczenie
- Metody tworzące RDD par
- Metody przetwarzające pojedyncze RDD par
- Łączenie RDD par
- Spark DocAPI