SP03_l2_23-Spark-RDD-par

November 4, 2023

1 RDD par klucz-wartość

1.1 Wprowadzenie

Podczas transformacji RDD, struktura jego składowych może przybierać różną postać. Szczególnym wariantem jest para - krotka składająca się dwóch pól. W takim przypadku RDD zmienia swój typ w `PairRDD` (`PairRDDFunctions`), który powszechnie występuje podczas bardziej złożonego przetwarzania.

Istnieje wiele metod RDD odnoszących się tylko i wyłącznie do RDD par.

Operacje na parach klucz-wartość to podstawa przetwarzania Big Data patrz: *MapReduce: Simplified Data Processing on Large Clusters; Jeffrey Dean and Sanjay Ghemawat; Google; 2004*

RDD par pozwala na przetwarzanie wartości powiązanych z określoną wartością klucza niezależnie i równolegle.

Istnieje wiele metod, które związane są tylko z RDD par:

- `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.

[2]:
```
# w przypadku korzystania z kernela Python
from pyspark import SparkContext, SparkConf
```

[2]:
```
# w przypadku korzystania z kernela Python
# w przypadku korzystania z klastra Hadoop
conf = SparkConf().setAppName("Spark - RDD par").setMaster("yarn")
```

```
sc = SparkContext(conf=conf)
```

[4]:
```
# w przypadku korzystania z kernela Python
# w przypadku korzystania ze Sparka lokalnie
conf = SparkConf().setAppName("Spark - RDD par").setMaster("local")
sc = SparkContext(conf=conf)
```

[5]:
```
import os
def remove_file(file):
    if os.path.exists(file):
        os.remove(file)

remove_file("ign.csv")
```

[7]:
```
import requests
r = requests.get("https://jankiewicz.pl/bigdata/bigdata-sp/ign.csv",␣
 ↪allow_redirects=True)
open('ign.csv', 'wb').write(r.content)
```

[7]: 2019628

[ ]:
```
%%sh
# w przypadku korzystania z klastra Hadoop
hadoop fs -copyFromLocal ign.csv .
```

[8]:
```
%%sh
head ign.csv
```

```
,score_phrase,title,url,platform,score,genre,editors_choice,release_year,release
_month,release_day
0,Amazing,LittleBigPlanet PS Vita,/games/littlebigplanet-
vita/vita-98907,PlayStation Vita,9.0,Platformer,Y,2012,9,12
1,Amazing,LittleBigPlanet PS Vita -- Marvel Super Hero
Edition,/games/littlebigplanet-ps-vita-marvel-super-hero-
edition/vita-20027059,PlayStation Vita,9.0,Platformer,Y,2012,9,12
2,Great,Splice: Tree of
Life,/games/splice/ipad-141070,iPad,8.5,Puzzle,N,2012,9,12
3,Great,NHL 13,/games/nhl-13/xbox-360-128182,Xbox 360,8.5,Sports,N,2012,9,11
4,Great,NHL 13,/games/nhl-13/ps3-128181,PlayStation 3,8.5,Sports,N,2012,9,11
5,Good,Total War Battles: Shogun,/games/total-war-battles-
shogun/mac-142565,Macintosh,7.0,Strategy,N,2012,9,11
6,Awful,Double Dragon: Neon,/games/double-dragon-neon/xbox-360-131320,Xbox
360,3.0,Fighting,N,2012,9,11
7,Amazing,Guild Wars 2,/games/guild-wars-2/pc-896298,PC,9.0,RPG,Y,2012,9,11
8,Awful,Double Dragon: Neon,/games/double-dragon-neon/ps3-131321,PlayStation
3,3.0,Fighting,N,2012,9,11
```

[9]:
```
rawIgn = sc.textFile("ign.csv")
rawIgn.count()
```

[9]:
```
18626
```

[10]:
```
rawIgn.first()
```

[10]:
```
',score_phrase,title,url,platform,score,genre,editors_choice,release_year,releas
e_month,release_day'
```

[11]:
```
import re
tabIgn = rawIgn.map(lambda line: re.split(",(?=(?:
 ↪[^\"]*\"[^\"]*\")*[^\"]*$)",line))
tabIgn.count()
```

[11]:
```
18626
```

[12]:
```
tabIgn.first()
```

[12]:
```
['',
 'score_phrase',
 'title',
 'url',
 'platform',
 'score',
 'genre',
 'editors_choice',
 'release_year',
 'release_month',
 'release_day']
```

[13]:
```
PLATFORM = 4
SCORE = 5
GENRE = 6
RELEASE_YEAR = 8
```

[14]:
```
gameInfosRdd = tabIgn.filter(lambda tab: len(tab)== 11 and len(tab[0])>0)
```

[15]:
```
gameInfosRdd.first()
```

[15]:
```
['0',
 'Amazing',
 'LittleBigPlanet PS Vita',
 '/games/littlebigplanet-vita/vita-98907',
 'PlayStation Vita',
 '9.0',
 'Platformer',
```

```
 'Y',
 '2012',
 '9',
 '12']
```

1.2 Metody tworzące RDD par

- `groupBy(f: Callable[[T], K](...)) → pyspark.rdd.RDD[Tuple[K, Iterable[T]]]`
- `keyBy(f: Callable[[T], K]) → pyspark.rdd.RDD[Tuple[K, T]]`
- `map(f: Callable[[T], Tuple[K, T]], preservesPartitioning: bool = False) → pyspark.rdd.RDD[Tuple[K, T]]`

[16]:
```
# jakiego typu są g1, g2 i g3?
g1 = gameInfosRdd.groupBy(lambda gi: gi[GENRE])
```

[17]:
```
g2 = gameInfosRdd.map(lambda gi: (gi[GENRE],gi))
```

[18]:
```
g3 = gameInfosRdd.keyBy(lambda gi: gi[GENRE]) # genre
```

2 Metody przetwarzające pojedynczy RDD par

2.1 Transformacje

- `groupByKey([numTasks])` – wywoływany na parach `(K,V)` zwraca dla każdej unikalnej wartości klucza `K` wynik postaci `(K,Iterable)`
- `reduceByKey(func: Callable[[V, V], V], numPartitions: Optional[int] = None, (...)) → pyspark.rdd.RDD[Tuple[K, V]]` – 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: Optional[bool] = True, numPartitions: Optional[int] = None, (...)) → pyspark.rdd.RDD[Tuple[K, V]]` – uruchamiane na parach `(K,V)` daje w wyniku `RDD` będący parami `(K,V)` posortowanymi względem klucza `K`,
- `aggregateByKey(zeroValue: U, seqFunc: Callable[[U, V], U], combFunc: Callable[[U, U], U], (...)) → pyspark.rdd.RDD[Tuple[K, U]]` – uruchamiane na parach `(K,V)`, daje w wyniku `RDD` będący parami `(K,U)`, w których wartości dla każdego klucza wyznaczane 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`.

2.1.1 Przykłady

[19]:
```
# Co wyliczają poniższe wyrażenia?
# Jakiego typu są wyniki?
# Czy wszystkie dają ten sam wynik?
g4 = gameInfosRdd.keyBy(lambda gi: gi[GENRE]).aggregateByKey(0.0, lambda m, gi:␣
 ↪m + float(gi[SCORE]), lambda mx,my: mx + my)
```

[20]:
```
g5 = gameInfosRdd.map(lambda gi: (gi[GENRE],gi)).groupByKey().\
                        mapValues(lambda gis: sum(list(map(lambda x:␣
 ↪float(x[SCORE]), gis))))
```

[21]:
```
g6 = gameInfosRdd.map(lambda gi: (gi[GENRE],float(gi[SCORE]))).
 ↪reduceByKey(lambda mx,my: mx + my)
```

Zastanów się, które z powyższych rozwiązań jest najbardziej wydajne

[22]:
```
%%time
g4.first()
```

```
CPU times: user 13.9 ms, sys: 3.56 ms, total: 17.4 ms
Wall time: 1.53 s
```

[22]:
```
('Platformer', 5914.5000000000055)
```

[23]:
```
%%time
g5.first()
```

```
CPU times: user 4.59 ms, sys: 23.1 ms, total: 27.7 ms
Wall time: 1.24 s
```

[23]:
```
('Platformer', 5914.5000000000055)
```

[24]:
```
%%time
g6.first()
```

```
CPU times: user 12 ms, sys: 356 µs, total: 12.3 ms
Wall time: 509 ms
```

[24]:
```
('Platformer', 5914.5000000000055)
```

2.2 Transformacje cd

- `mapValues(f: Callable[[V], U]) → pyspark.rdd.RDD[Tuple[K, U]]` – mapowanie dotyczy tylko wartości
- `lookup(key: K) → List[V]` – wydobywa wartości powiązane z kluczem
- `keys() → pyspark.rdd.RDD[K]` – tworzy RDD składające się z samych kluczy
- `values() → pyspark.rdd.RDD[V]` – tworzy RDD składające się z samych wartości

2.3 Akcje

- `countByKey() → Dict[K, int]` – tworzy obiekt `Dict` zawierający dla każdego klucza liczbę wystąpień
- `countByValue() → Dict[V, int]` – tworzy obiekt `Dict` zawierający dla każdej wartości liczbę wystąpień
- `collectAsMap() → Dict[K, V]` – tworzy obiekt `Dict` odzworowujący zawartość RDD

[25]:
```
g7 = gameInfosRdd.map(lambda gi: (gi[GENRE],gi)).groupByKey().\
                        mapValues(lambda gis: sum(list(map(lambda x:␣
 ↪float(x[SCORE]), gis))))
```

[26]:
```
g8 = gameInfosRdd.keyBy(lambda gi: gi[GENRE]).countByKey()
{k: g8[k] for k in list(g8)[:10]}
```

[26]:
```
{'Platformer': 823,
 'Puzzle': 776,
 'Sports': 1916,
 'Strategy': 1071,
 'Fighting': 547,
 'RPG': 980,
 '': 36,
 '"Action, Adventure"': 765,
 'Adventure': 1175,
 'Action': 3797}
```

2.4 Ćwiczenia

`countByKey` jest akcją – punktem końcowym przetwarzania

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

[27]:
```
g9 = gameInfosRdd
```

Dokończ poniższy fragment kodu tak, aby wyznaczyć liczbę platform objętych recenzjami gier.

[ ]:
```
gameInfosRdd.keyBy(lambda gi: gi[PLATFORM])
```

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

[ ]:
```
result = gameInfosRdd.map(lambda gi: (gi[GENRE],gi))
```

[ ]:
```
sorted(result, key=lambda v: v[1], reverse=True)[:5]
```

2.4.1 Rozwiązania

[31]:
```
g9 = gameInfosRdd.map(lambda gi: (gi[GENRE],1)).reduceByKey(lambda x, y: x + y)
g9.take(10)
```

[31]:
```
[('Platformer', 823),
 ('Puzzle', 776),
 ('Sports', 1916),
 ('Strategy', 1071),
 ('Fighting', 547),
```

```
 ('RPG', 980),
 ('', 36),
 ('"Action, Adventure"', 765),
 ('Adventure', 1175),
 ('Action', 3797)]
```

[32]:
```
gameInfosRdd.keyBy(lambda gi: gi[PLATFORM]).groupByKey().count()
```

[32]:
```
59
```

[33]:
```
result = gameInfosRdd.map(lambda gi: (gi[GENRE],gi))\
                     .mapValues(lambda gi: (float(gi[SCORE]),1))\
                     .reduceByKey(lambda p1, p2: (p1[0]+p2[0], p1[1]+p2[1]))\
                     .mapValues(lambda p: p[0]/p[1]).collect()
```

[34]:
```
sorted(result, key=lambda v: v[1], reverse=True)[:5]
```

[34]:
```
[('"Compilation, Compilation"', 9.5),
 ('Hardware', 9.15),
 ('"Puzzle, RPG"', 9.1),
 ('"Other, Action"', 9.0),
 ('"Adventure, Episodic"', 8.9)]
```

3 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

3.1 Metody

- `join(other: pyspark.rdd.RDD[Tuple[K, U]], numPartitions: Optional[int] = None) → pyspark.rdd.RDD[Tuple[K, Tuple[V, U]]]`
- `leftOuterJoin(other: pyspark.rdd.RDD[Tuple[K, U]], numPartitions: Optional[int] = None) → pyspark.rdd.RDD[Tuple[K, Tuple[V, Optional[U]]]]`
- `rightOuterJoin(other: pyspark.rdd.RDD[Tuple[K, U]], numPartitions: Optional[int] = None) → pyspark.rdd.RDD[Tuple[K, Tuple[Optional[V], U]]]`
- `cogroup(other: pyspark.rdd.RDD[Tuple[K, U]], numPartitions: Optional[int] = None) → pyspark.rdd.RDD[Tuple[K, Tuple[pyspark.resultiterable.ResultIterable[V], pyspark.resultiterable.ResultIterable[U]]]]`

3.1.1 Pytania

- 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?

3.2 Przykłady

[35]:
```
ps4rdd = gameInfosRdd.filter(lambda gi: gi[PLATFORM] == "PlayStation 4").
 ↪keyBy(lambda gi: gi[RELEASE_YEAR])
wiiUrdd = gameInfosRdd.filter(lambda gi: gi[PLATFORM] == "Wii U").keyBy(lambda␣
 ↪gi: gi[RELEASE_YEAR])
xbOnerdd = gameInfosRdd.filter(lambda gi: gi[PLATFORM] == "Xbox One").
 ↪keyBy(lambda gi: gi[RELEASE_YEAR])
```

[36]:
```
#Jaki tu będzie typ wyniku?
alljoin = ps4rdd.join(wiiUrdd).join(xbOnerdd)
```

[37]:
```
alljoin.first()
```

[37]:
```
('2014',
 (['17452',
   'Amazing',
   'Tomb Raider: Definitive Edition',
   '/games/tomb-raider-definitive-edition/ps4-20009692',
   'PlayStation 4',
   '9.1',
   'Action',
   'Y',
   '2014',
   '1',
   '25'],
  ['17489',
   'Good',
   'Dr. Luigi',
   '/games/dr-luigi/wii-u-20010245',
   'Wii U',
   '7.5',
   'Puzzle',
   'N',
   '2014',
   '1',
   '10']),
 ['17451',
  'Amazing',
  'Tomb Raider: Definitive Edition',
  '/games/tomb-raider-definitive-edition/xbox-one-20009691',
  'Xbox One',
```

```
  '9.1',
  'Action',
  'Y',
  '2014',
  '1',
  '25']))
```

[38]:
```
# A tu?
allouterjoin = ps4rdd.fullOuterJoin(wiiUrdd).fullOuterJoin(xbOnerdd)
```

[39]:
```
allouterjoin.first()
```

[39]:
```
('2012',
 (['192',
   'Amazing',
   'Sound Shapes',
   '/games/sound-shapes-queasy-games/ps4-20007461',
   'PlayStation 4',
   '9.0',
   'Platformer',
   'Y',
   '2012',
   '8',
   '8'],
  ['259',
   'Okay',
   'Sing Party',
   '/games/sing/wii-u-135741',
   'Wii U',
   '6.3',
   'Music',
   'N',
   '2012',
   '12',
   '18']),
 None))
```

[40]:
```
# A tu?
allcogroup2 = ps4rdd.cogroup(wiiUrdd).cogroup(xbOnerdd)
```

[41]:
```
allcogroup2.first()
```

[41]:
```
('2012',
 (<pyspark.resultiterable.ResultIterable at 0x7f9fc774b4f0>,
  <pyspark.resultiterable.ResultIterable at 0x7f9fc774bbe0>))
```

3.3 Ćwiczenie

[ ]:
```
# Co wpisać zamiast  . . . aby wyświetlanie informacji zadziałało poprawnie ␣
 ↪jak pokazano poniżej?

allval = ps4rdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y).\
    join(wiiUrdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y)).\
    join(xbOnerdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y)).\
    mapValues . . .
```

[ ]:
```
allval.sortByKey(True).collect()
# wynik ma byc nastepujacy:
# [('2013', (34, 44, 23)),
# ('2014', (84, 23, 59)),
# ('2015', (97, 13, 85)),
# ('2016', (61, 6, 41))]
```

3.3.1 Rozwiązanie

[44]:
```
allval = ps4rdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y).\
    join(wiiUrdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y)).\
    join(xbOnerdd.mapValues(lambda gi: 1).reduceByKey(lambda x, y: x + y)).\
    mapValues(lambda p: (p[0][0],p[0][1],p[1]))
```

[45]:
```
allval.sortByKey(True).collect()
```

[45]:
```
[('2013', (34, 44, 23)),
 ('2014', (84, 23, 59)),
 ('2015', (97, 13, 85)),
 ('2016', (61, 6, 41))]
```

[ ]: