Working with Pair RDDs

MSBA 6330 Prof Liu

Outline

– What is a Pair RDD?
– How to create Pair RDDs
– Special operations available on Pair RDDs
– How map-reduce algorithms are implemented in Spark

What is a Pair RDD?

Working with Pair RDDs

Pair RDDs

- Pair RDDs are a special form of RDD
  - Each element must be a **key-value pair** (a two-element tuple)
  - Keys and values can be any type
- Why?
  - Use with map-reduce algorithms
  - Many additional functions are available for common data processing needs
    - e.g., sorting, joining, grouping, counting, etc.

Pair RDD

`(key1,value1)`
`(key2,value2)`
`(key3,value3)`
…

How to create Pair RDDs

Working with Pair RDDs

Creating Pair RDDs

- The first step is to decide:
  - What should the RDD should be keyed on?
  - What is the value?
- Commonly used functions to create Pair RDDs
  - `map`
  - `flatMap`
  - `flatMapValues` (Keep the keys, just map values)
  - `keyBy` (keep the values, just add key)

Create a Pair RDD using map

Example: Create a Pair RDD from a tab–separated file

The new row is a tuple

Python

```
> users = sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0],fields[1]))
```

Scala

```
> val users = sc.textFile(file)
    .map(line => line.split('\t'))
    .map(fields => (fields(0),fields(1)))
```

```
user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
…
```

```
(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
…
```


Create a Pair RDD using keyBy

Example: Keying Web Logs using keyBy

Python

```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

Scala

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

```
                    (User ID)
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

keyBy: Constructs two-component tuples (key-value pairs) by applying a function on each data item. The result of the function becomes the key and the original data item becomes the value of the newly created tuples.

Create Pairs With Complex Values

- **How would you do this?**
  - Input: a list of postal codes with latitude and longitude
  - Output: postal code (key) and lat/long pair (value)

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],(fields[1],fields[2])))
```

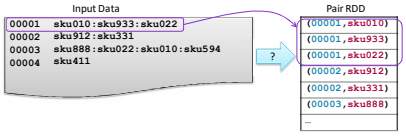
```
> sc.textFile(file).
   map(line => line.split('\t')).
   map(fields => (fields(0),(fields(1),fields(2))))
```

```
00210   43.005895   -71.013202
01014   42.170731   -72.604842
01062   42.324232   -72.67915
01263   42.3929     -73.228483
…
```

```
(00210,(43.005895,-71.013202))
(01014,(42.170731,-72.604842))
(01062,(42.324232,-72.67915))
(01263,(42.3929,-73.228483))
…
```

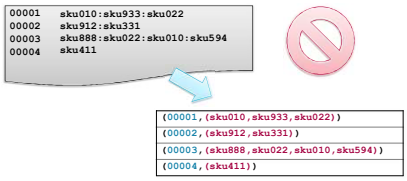
Mapping Single Rows to Multiple Pairs (1)

- **How would you do this?**
  - Input: order numbers with a list of SKUs in the order
  - Output: order (key) and sku (value)



Mapping Single Rows to Multiple Pairs (2)

Hint: map alone won't work



Answer: Mapping Single Rows to Multiple Pairs (1)

```
> sc.textFile(file)
```

00001	sku010:sku933:sku022
00002	sku912:sku331
00003	sku888:sku022:sku010:sku594
00004	sku411

Answer: Mapping Single Rows to Multiple Pairs (2)

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t'))
```

```
00001  sku010:sku933:sku022
00002  sku912:sku331
```

```
[00001,sku010:sku933:sku022]
[00002,sku912:sku331]
[00003,sku888:sku022:sku010:sku594]
[00004,sku411]
```

Note that `split` returns 2-element arrays, not pairs/tuples

Answer: Mapping Single Rows to Multiple Pairs (3)

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
```

```
00001  sku010:sku933:sku022
00002  sku912:sku331
```

```
[00001,sku010:sku933:sku022]
[00002,sku912:sku331]
```

```
(00001,sku010:sku933:sku022
(00002,sku912:sku331)
(00003,sku888:sku022:sku010:sku594)
(00004,sku411)
```

Map array elements to tuples to produce a Pair RDD

Answer: Mapping Single Rows to Multiple Pairs (4)

```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
  .flatMapValues(lambda skus: skus.split(':'))
```

```
00001  sku010:sku933:sku022
00002  sku912:sku331
```

```
[00001,sku010:sku933:sku022]
[00002,sku912:sku331]
```

```
(00001,sku010:sku933:sku022
(00002,sku912:sku331)
(00003,sku888:sku022:sku010:sku594)
(00004,sku411)
```

```
(00001,sku010)
(00001,sku933)
(00001,sku022)
(00002,sku912)
(00002,sku331)
(00003,sku888)
...
```

Special operations available on Pair RDDs

Working with Pair RDDs

Pair RDD Operations

- Spark has several operations specific to Pair RDDs
  - `reduceByKey` [transformation] - Merge the values for each key using an associative reduce function.
  - `countByKey` [action] – Count the number of elements for each key, and return **the result to the master** as a Map.
  - `groupByKey` [transformation] – Group the values for each key in the RDD into a single sequence
  - `sortByKey` [transformation] – sort the pair RDD by key
  - `join` [transformation] – return an RDD containing all pairs with matching keys from two RDDs

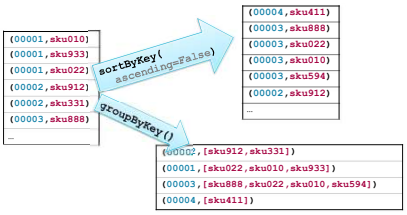
Key-value Transformation

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6), (3,3)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6), (3, 3)] -> [(1,2), (3,13)]

>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] -> [(1,['a','b']), (2,['c'])]
```

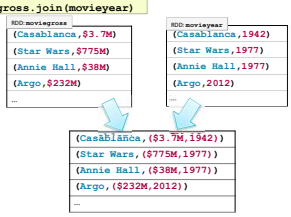
Be careful using `groupByKey`() as it can cause a lot of data movement across the network and create large iterables at workers. Some suggests that you should avoid this operation.

Example: sortByKey and groupByKey



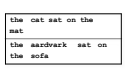
Example: join

```
> movies = moviegross.join(movieyear)
```



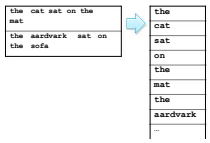
Case Study : Word Count (1)

```
> counts = sc.textFile(file)
```



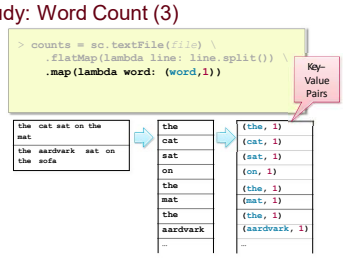
Case Study : Word Count (2)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



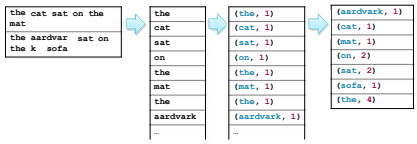
Case Study: Word Count (3)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```



Case Study : Word Count (4)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

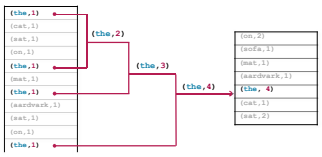


ReduceByKey (1)

The function passed to `reduceByKey` combines values from two keys

- Function must be binary

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```

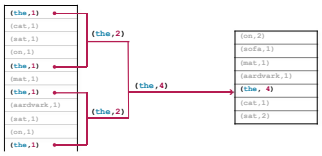


ReduceByKey (2)

The function might be called in any order, therefore must be

- Commutative – x+y = y+x
- Associative – (x+y)+z = x+(y+z)

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Recap (the Scala Version)

```
> val counts = sc.textFile(file).
  flatMap(line => line.split("\\W")).
  map(word => (word,1)).
  reduceByKey((v1,v2) => v1+v2)
```

`\W` is regex for white space

OR

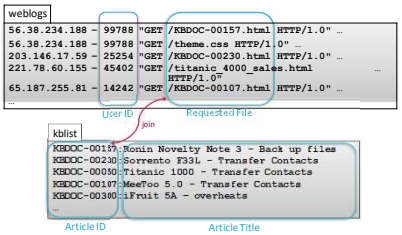
```
> val counts = sc.textFile(file).
  flatMap(_.split("\\W")).
  map((_,1)).
  reduceByKey(_+_)
```

Underscores are placeholders for arguments (to replace explicit arguments names)

Case Study: Using Join to combine data sources

- **A common programming pattern**
  1. Map separate datasets into key-value Pair RDDs
  2. Join by key
  3. Map joined data into the desired format
  4. Save, display, or continue processing…

Example: Join Web Log With Knowledge Base Articles (1)



We want to obtain a list of articles visited by each user

Example: Join Web Log With Knowledge Base Articles (2)

- Steps
  1. Map separate datasets into key--value Pair RDDs
     - Map web log requests to (`docid`,`userid`)
     - Map KB Doc index to (`docid`,`title`)
  2. Join by key: `docid`
  3. Map joined data into the desired format: (`userid`,`title`)
  4. Further processing: group titles by User ID

Step 1a: Map Web Log Requests to **(docid,userid)**

```
> import re
> def getRequestDoc(s):
    return re.search(r'KBDOC-[0-9]*',s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBDOC-' in line) \
    .map(lambda line: (getRequestDoc(line),line.split(' ')[2])) \
    .distinct()
```

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
221.78.60.155 - 45402 "GET /titanic_4000_sales.html HTTP/1.0" …
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.0" …
…
```

kbreqs

```
(KBDOC-00157,99788)
(KBDOC-00203,25254)
(KBDOC-00107,14242)
…
```


Step 1b: Map KB Index to **(docid,title)**

```
> kblist = sc.textFile(kblistfile) \
    .map(lambda line: line.split(':')) \
    .map(lambda fields: (fields[0],fields[1]))
```

```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A - overheats
…
```

kblist

```
(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…
```


Step 2: Join By Key **docid**

```
> titlereqs = kbreqs.join(kblist)
```

kbreqs

```
(KBDOC-00157,99788)
(KBDOC-00230,25254)
(KBDOC-00107,14242)
…
```

kblist

```
(KBDOC-00157,Ronin Novelty Note 3 - Back up files)
(KBDOC-00230,Sorrento F33L - Transfer Contacts)
(KBDOC-00050,Titanic 1000 - Transfer Contacts)
(KBDOC-00107,MeeToo 5.0 - Transfer Contacts)
…
```

```
(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…
```

Step 3: Map Result to Desired Format `(userid,title)`

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title))
```

```
(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
…
```

```
(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…
```


Step 4: Continue Processing – Group Titles by User ID

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid,(userid,title)): (userid,title)) \
    .groupByKey()
```

```
(99788,Ronin Novelty Note 3 - Back up files)
(25254,Sorrento F33L - Transfer Contacts)
(14242,MeeToo 5.0 - Transfer Contacts)
…
```

```
(99788,[Ronin Novelty Note 3 - Back up files, Ronin S3
        - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts, MeeToo
        5.1 - Back up files, iFruit 1 - Back up
        files, MeeToo 3.1 - Transfer Contacts])
…
```

Note: values are grouped into Iterables

Example Output

```
> for (userid,titles) in titlereqs.take(10):
    print 'user id: ',userid
    for title in titles: print '\t',title

user id: 99788
  Ronin Novelty Note 3 - Back up files
  Ronin S3 - overheating
user id: 25254
  Sorrento F33L - Transfer Contacts
user id: 14242
  MeeToo 1.- Transfer Contacts
  MeeToo 2.- Back up files
  iFruit 1 - Back up files
  MeeToo 3.1 - Transfer Contacts
```

```
(99788,[Ronin Novelty Note 3 - Back up files,
        Ronin S3 - overheating])
(25254,[Sorrento F33L - Transfer Contacts])
(14242,[MeeToo 5.0 - Transfer Contacts,
        MeeToo 5.1 - Back up files,
        iFruit 1 - Back up files,
        MeeToo 3.1 - Transfer Contacts])
…
```

Other Pair Operations

- **Some other pair operations**
  - `keys` – return an RDD of just the keys, without the values
  - `values` – return an RDD of just the values, without keys
  - `lookup(key)` – return the value(s) for a key
  - `leftOuterJoin`, `rightOuterJoin` , `fullOuterJoin` – join, including keys defined in the left, right or either RDD respectively
  - `mapValues`, `flatMapValues` – execute a function on just the values, keeping the key the same
- **See the `PairRDDFunctions` class Scaladoc for a full list**
  - https://spark.apache.org/docs/latest/api/java/org/apache/spark/rdd/PairRDDFunctions.html

Spark and MapReduce

- Hadoop MapReduce is Somewhat limited
  - Each job has one Map phase, one Reduce phase
  - Job output is saved to files
- Spark implements map-reduce with much greater flexibility
  - Map and reduce functions can be interspersed
  - Results can be stored in memory
    - Operations can easily be chained
  - Higher level APIs

Map-Reduce in Spark

- **Map-reduce in Spark works on Pair RDDs**
- **Map phase**
  - Operates on one record at a time
  - "Maps" each record to one or more new records
  - e.g. `map`, `flatMap`, `filter`, `keyBy`
- **Reduce phase**
  - Works on map output
  - Consolidates multiple records
  - e.g. `reduceByKey`, `mean`

Essential Points

- Pair RDDs are a special form of RDD consisting of Key-Value pairs (tuples)
- Spark provides several operations for working with Pair RDDs
  - Introduction to Pair RDD operations
  - Pair RDD Class documentation
- Spark implements map-reduce with Pair RDDs
  - Hadoop MapReduce and other implementations are limited to a single map and single reduce phase per job
  - Spark allows flexible chaining of map and reduce operations
  - Spark provides operations to easily perform common map-reduce algorithms like joining, sorting, and grouping