



# Aggregating Data with Pair RDDs

Prof. Hyuk-Yoon Kwon

# Aggregating Data with Pair RDDs

---

In this chapter you will learn

- How to create Pair RDDs of key-value pairs from generic RDDs
- Special operations available on Pair RDDs
- How map-reduce algorithms are implemented in Spark

# 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)
...

# Creating Pair RDDs

---

- **The first step in most workflows is to get the data into key/value form**
  - What should the RDD should be keyed on?
  - What is the value?
- **Commonly used functions to create Pair RDDs**
  - `map`
  - `flatMap / flatMapValues`
  - `keyBy`

# Example: A Simple Pair RDD

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

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)
...

# Example: Keying Web Logs by User ID

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 /KBD0C-00157.html HTTP/1.0" ...
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" ...
203.146.17.59 - 25254 "GET /KBD0C-00230.html HTTP/1.0" ...
...
```



```
(99788,56.38.234.188 - 99788 "GET /KBD0C-00157.html...")
(99788,56.38.234.188 - 99788 "GET /theme.css...")
(25254,203.146.17.59 - 25254 "GET /KBD0C-00230.html...")
...
```

# Question 1: 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)

00210	43.005895	-71.013202
00211	43.005895	-71.013202
00212	43.005895	-71.013202
00213	43.005895	-71.013202
00214	43.005895	-71.013202
...		



(00210, (43.005895, -71.013202))
(00211, (43.005895, -71.013202))
(00212, (43.005895, -71.013202))
(00213, (43.005895, -71.013202))
...

# Answer 1: Pairs With Complex Values

```
> sc.textFile(file) \
  .map(lambda line: line.split()) \
  .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
...
```



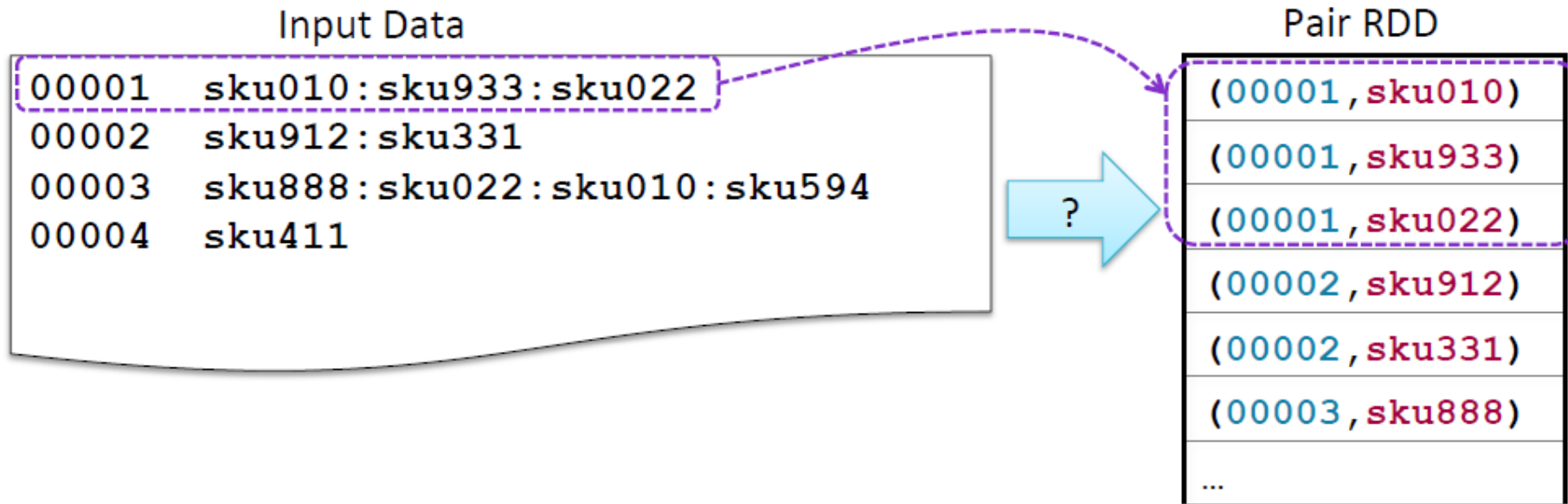
```
(00210, (43.005895, -71.013202))
(01014, (42.170731, -72.604842))
(01062, (42.324232, -72.67915))
(01263, (42.3929, -73.228483))
...
```



## Question 2: 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)



## Question 2: Mapping Single Rows to Multiple Pairs (2)

- Hint: map alone won't work

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



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

## Answer 2: 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 2: Mapping Single Rows to Multiple Pairs(2)

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

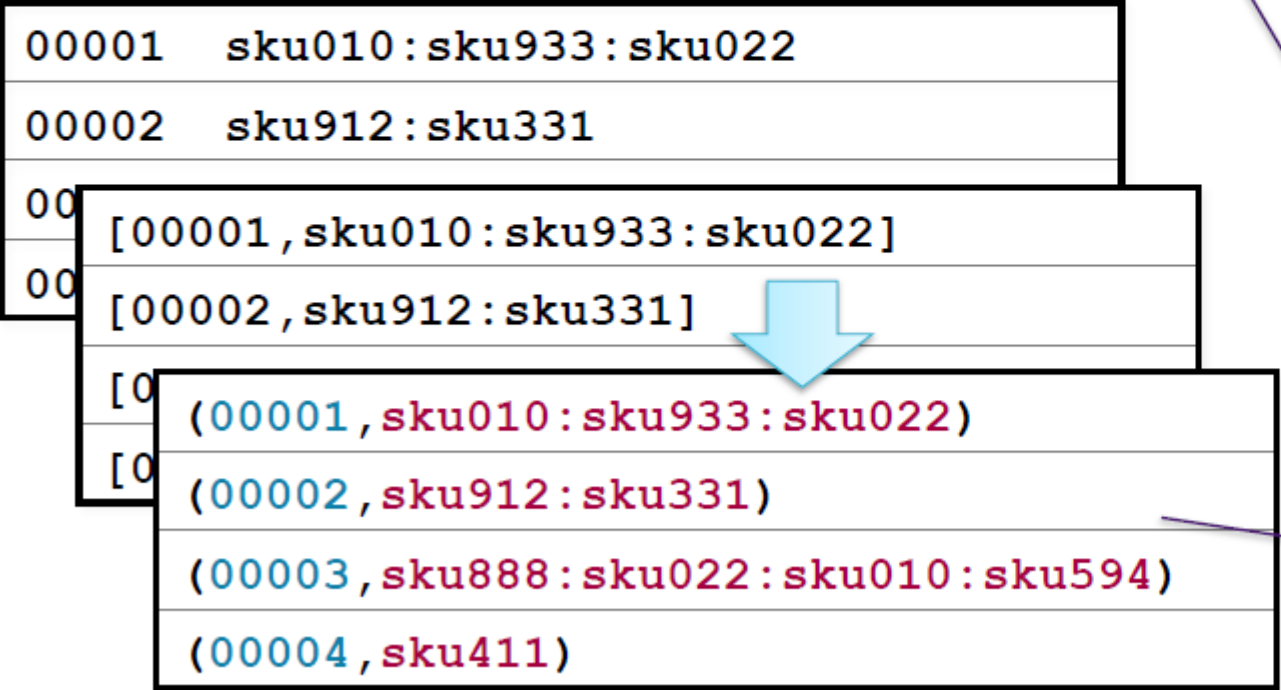
00001	sku010:sku933:sku022
00002	sku912:sku331

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 2: Mapping Single Rows to Multiple Pairs(3)

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



Map array elements to tuples to produce a Pair RDD

# Answer 2: 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]

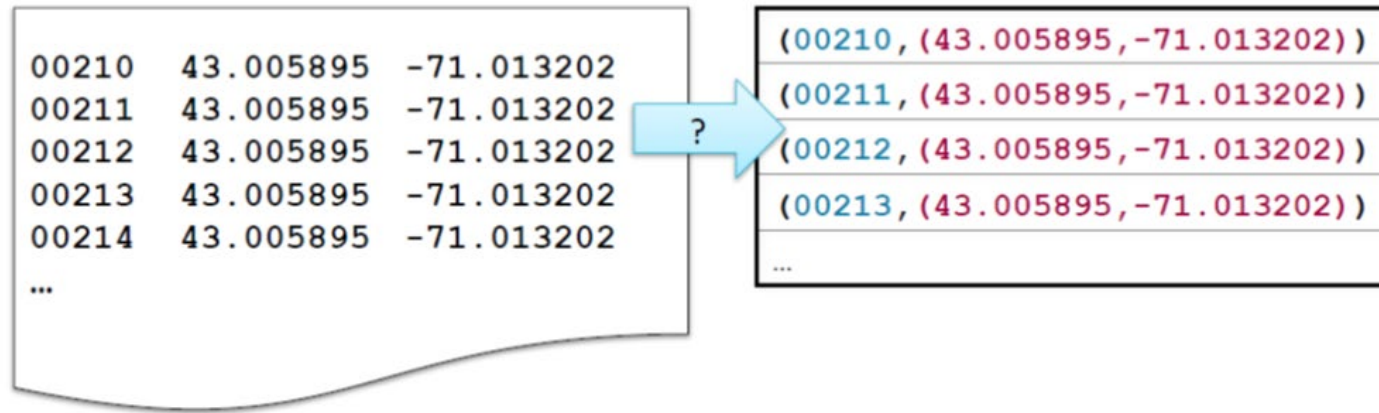
00001	(sku010:sku933:sku022)
00002	(sku912:sku331)
00003	(sku888:sku022:sku010:sku594)
00004	(sku411)

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

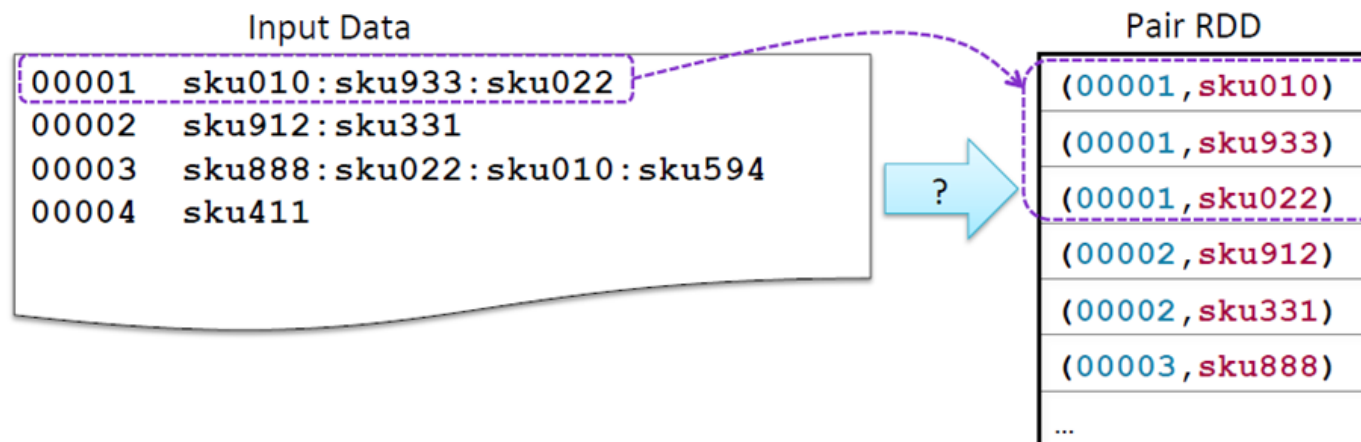
# Practice

## ■ Practice Questions 1 and 2 using a sample data presented in the previous slides

- Question 1



- Question 2



# Map-Reduce

---

- **Map-reduce is a common programming model**
  - Easily applicable to distributed processing of large data sets
- **Hadoop MapReduce is the major implementation**
  - 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



# 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`, `sortByKey`, `mean`

# Map-Reduce Example: Word Count

Input Data

```
the cat sat on the mat  
the aardvark sat on the sofa
```



Result

aardvark	1
cat	1
mat	1
on	2
sat	2
sofa	1
the	4

# Example: Word Count(1)

---

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

the cat sat on the mat
the aardvark sat on the sofa

## Example: Word Count(2)

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

the cat sat on the mat
---------------------------

the aardvark sat on the sofa
---------------------------------



the
-----

cat
-----

sat
-----

on
----

the
-----

mat
-----

the
-----

aardvark
----------

...
-----

# Example: Word Count(3)

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

Key-  
Value  
Pairs

the cat sat on the mat
the aardvark sat on the sofa



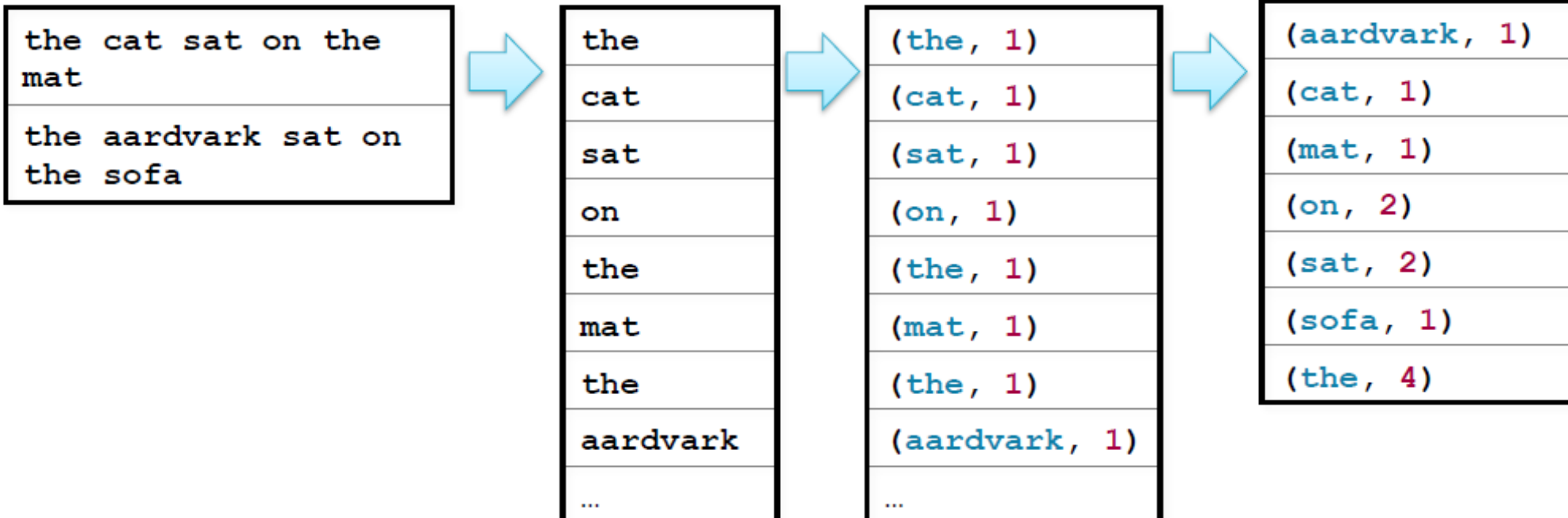
the
cat
sat
on
the
mat
the
aardvark
...



(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
...

# Example: Word Count(4)

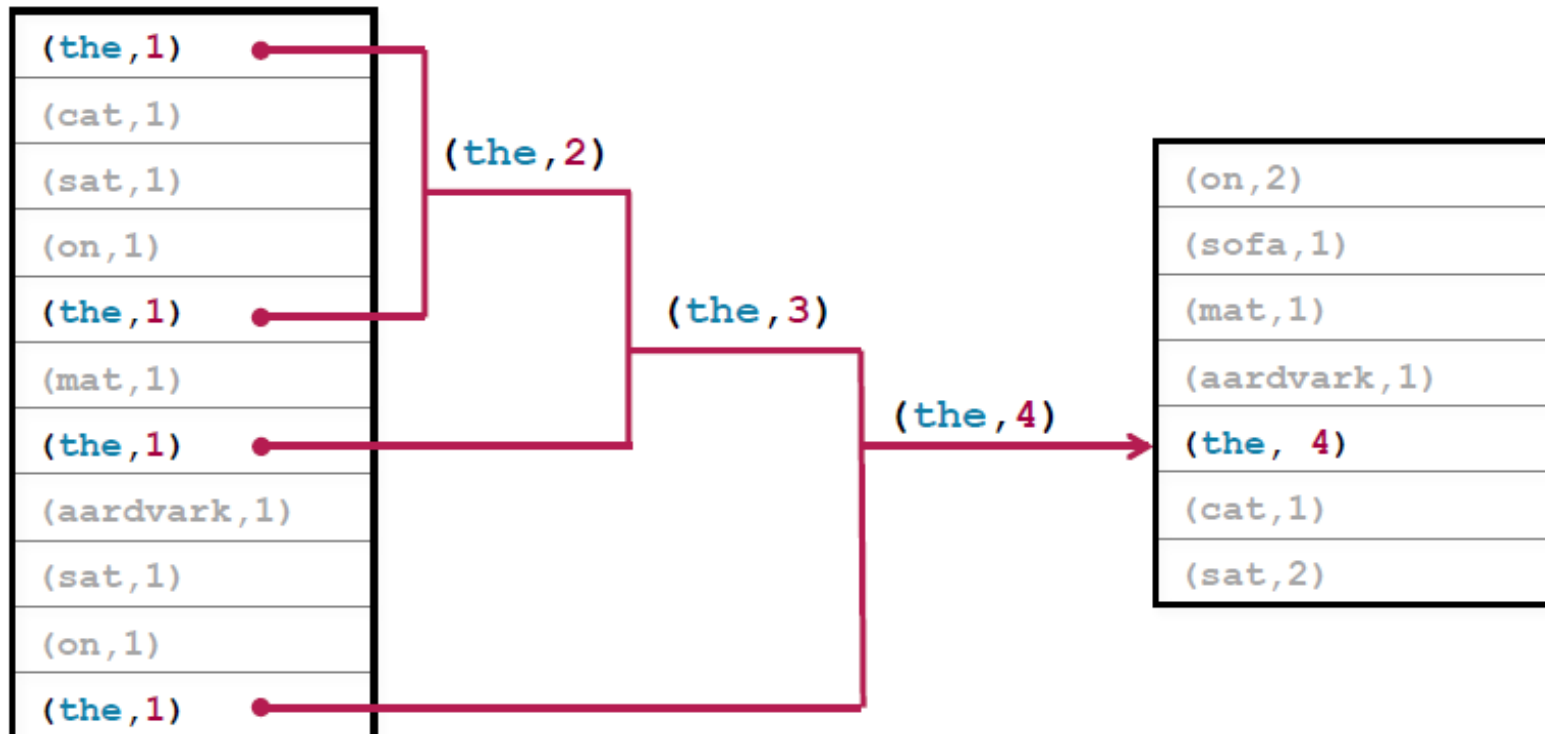
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



# ReduceByKey(1)

- The function passed to `reduceByKey` combines values from two keys
  - Function must be binary

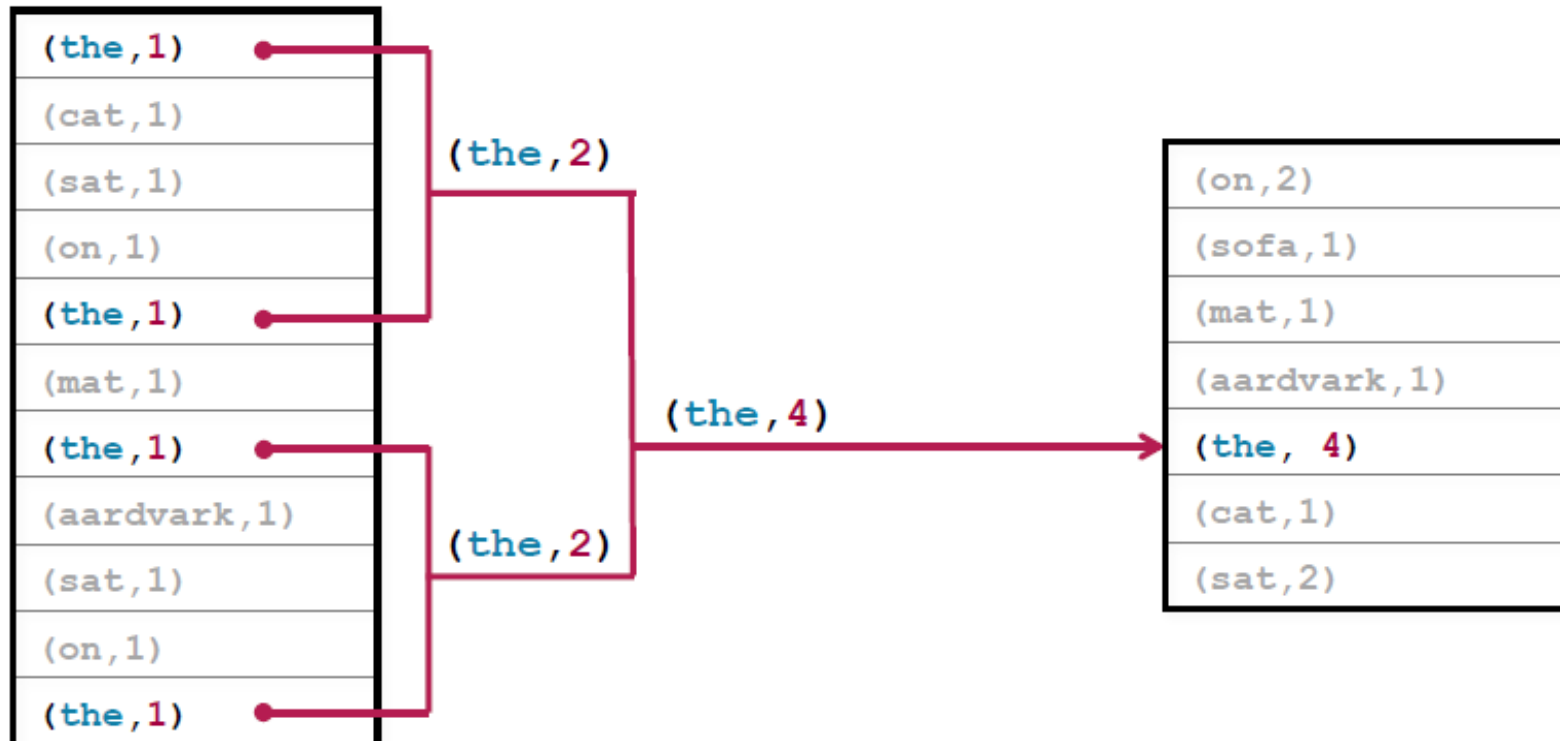
```
> counts = sc.textFile(file) \  
  .flatMap(lambda line: line.split()) \  
  .map(lambda word: (word,1)) \  
  .reduceByKey(lambda v1,v2: v1+v2)
```



# 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)
```

OR

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

# Why Do We Care About Counting Words?

---

- **Word count is challenging over massive amounts of data**
  - Using a single compute node would be too time-consuming
  - Number of unique words could exceed available memory
- **Statistics are often simple aggregate functions**
  - Distributive in nature
  - e.g., max, min, sum, count
- **Map-reduce breaks complex tasks down into smaller elements which can be executed in parallel**
- **Many common tasks are very similar to word count**
  - e.g., log file analysis

# Practice: Reduce

## ■ Data sets (HDFS)

- /loudacre/weblogs
- First, upload weblog data in your local file system into the HDFS

## ■ Using map-reduce, count the number of requests from each user.

- Use map to create a Pair RDD with the user ID as the key, and the integer 1 as the value. (The user ID is the third field in each line). Your data will look something like this:

(userid,1)
(userid,1)
(userid,1)
...

- Use reduce to sum the values for each user ID. Your RDD data will be similar to:

(userid,5)
(userid,7)
(userid,2)
...

# Pair RDD Operations

---

- In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs
- Examples
  - `countByKey` – return a map with the count of occurrences of each key
  - `groupByKey` – group all the values for each key in an RDD
  - `sortByKey` – sort in ascending or descending order
  - `join` – return an RDD containing all pairs with matching keys from two RDDs

# Example: Pair RDD Operations

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

sortByKey(  
ascending=False)

groupByKey()

(00004, sku411)
(00003, sku888)
(00003, sku022)
(00003, sku010)
(00003, sku594)
(00002, sku912)
...

(00002, [sku912, sku331])
(00001, [sku022, sku010, sku933])
(00003, [sku888, sku022, sku010, sku594])
(00004, [sku411])

# Practice: Count and Group

---

- This practice continues to use the previous data sets
- Use `countByKey` to determine how many users visited the site for each frequency. That is, how many users visited once, twice, three times and so on.
  - Use `map` to reverse the key and value, like this:

(5, <i>userid</i> )
(7, <i>userid</i> )
(2, <i>userid</i> )
...

- Use the `countByKey` action to return a Map of frequency:user-count pairs

■ Create an RDD where the user id is the key, and the value is the list of all the IP addresses that user has connected from.

- Hint: Map to (userid, ipaddress) and then use groupByKey

(userid, 20.1.34.55)
(userid, 245.33.1.1)
(userid, 65.50.196.141)
...




(userid, [20.1.34.55, 74.125.239.98])
(userid, [75.175.32.10, 245.33.1.1, 66.79.233.99])
(userid, [65.50.196.141])
...

# Example: Joining by Key

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

RDD:moviegross
(Casablanca,\$3.7M)
(Star Wars,\$775M)
(Annie Hall,\$38M)
(Argo,\$232M)
...

RDD:movieyear
(Casablanca,1942)
(Star Wars,1977)
(Annie Hall,1977)
(Argo,2012)
...



(Casablanca, (\$3.7M,1942))
(Star Wars, (\$775M,1977))
(Annie Hall, (\$38M,1977))
(Argo, (\$232M,2012))
...



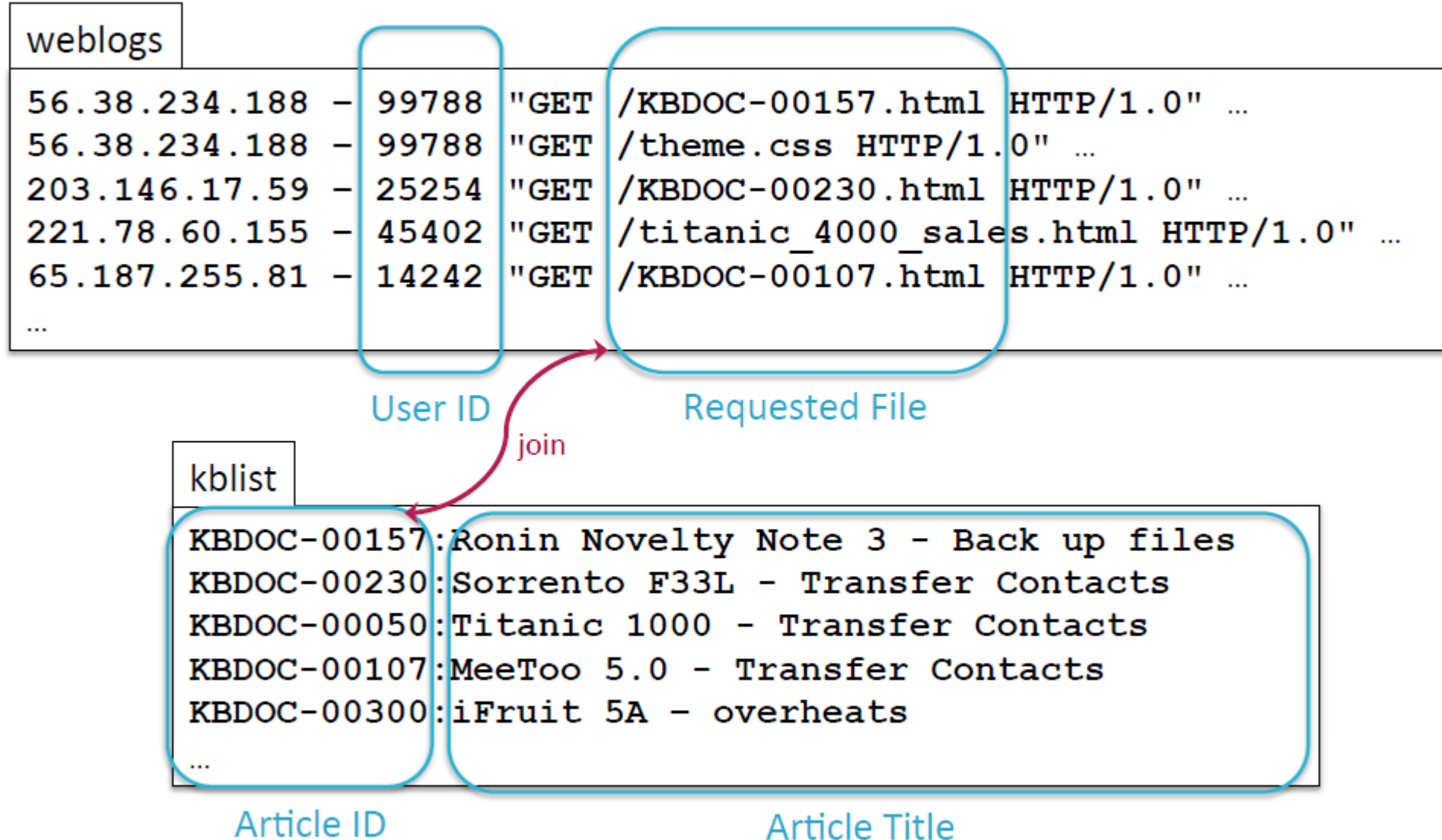
# Using Join

---

- **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)



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

---

## ■ Steps

1. Map separate datasets into key-value Pair RDDs
  - a. Map web log requests to `(docid, userid)`
  - b. 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'KBD0C-[0-9]*', s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBD0C-' in line) \
    .map(lambda line: (getRequestDoc(line), line.split(' ')[2])) \
    .distinct()
```

```
56.38.234.188 - 99788 "GET /KBD0C-00157.html HTTP/1.0" ...
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" ...
203.146.17.59 - 25254 "GET /KBD0C-00230.html HTTP/1.0" ...
221.78.60.155 - 45402 "GET /titanic_4000_sales.html ...
65.187.255.81 - 14242 "GET /KBD0C-00107.html HTTP/1.0" ...
...
```

kbreqs

(KBD0C-00157, 99788)

(KBD0C-00203, 25254)

(KBD0C-00107, 14242)

...

## Step1b: Map KB Index to (docid, title)

```
> kblast = sc.textFile(kblastfile) \
    .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
...
```



kblast

(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

(KBD0C-00157,99788)

(KBD0C-00230,25254)

(KBD0C-00107,14242)

...

kblist

(KBD0C-00157,Ronin Novelty Note 3 - Back up files)

(KBD0C-00230,Sorrento F33L - Transfer Contacts)

(KBD0C-00050,Titanic 1000 - Transfer Contacts)

(KBD0C-00107,MeeToo 5.0 - Transfer Contacts)

...

(KBD0C-00157,(99788,Ronin Novelty Note 3 - Back up files))

(KBD0C-00230,(25254,Sorrento F33L - Transfer Contacts))

(KBD0C-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))
```

(KBD0C-00157, (99788, Ronin Novelty Note 3 - Back up files))
(KBD0C-00230, (25254, Sorrento F33L - Transfer Contacts))
(KBD0C-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)
...



Note: values  
are grouped  
into Iterables

(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])
...



# 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 5.0 - Transfer Contacts

MeeToo 5.1 - 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])

...

# Practice: Join Web Log Data with Account Data

## ■ Data files (HDFS)

- /loudacre/weblogs
- /loudacre/accounts

## ■ Join the accounts data with the weblog data to produce a dataset keyed by user ID which contains the user account information and the number of website hits for that user.

- Create an RDD based on the accounts data consisting of key/value-array pairs: (userid, [values ...])

(userid1,[userid1,2008-11-24 10:04:08,\N,Cheryl,West,4905 Olive Street,San Francisco,CA,...])
(userid2,[userid2,2008-11-23 14:05:07,\N,Elizabeth,Kerns,4703 Eva Pearl Street,Richmond,CA,...])
(userid3,[userid3,2008-11-02 17:12:12,2013-07-18 16:42:36,Melissa,Roman,3539 James Martin Circle,Oakland,CA,...])
...

- Join the Pair RDD with the set of user-id/hit-count pairs calculated in the first step

<code>(userid1, ([userid1, 2008-11-24 10:04:08, \N, Cheryl, West, 4905 Olive Street, San Francisco, CA, ...], 4))</code>
<code>(userid2, ([userid2, 2008-11-23 14:05:07, \N, Elizabeth, Kerns, 4703 Eva Pearl Street, Richmond, CA, ...], 8))</code>
<code>(userid3, ([userid3, 2008-11-02 17:12:12, 2013-07-18 16:42:36, Melissa, Roman, 3539 James Martin Circle, Oakland, CA, ...], 1))</code>
...

- Display the user ID, hit count, and first name (3<sup>rd</sup> value), and last name (4<sup>th</sup> value) for the first 5 elements, e.g.,:

<code>userid1 4 Cheryl West</code>
<code>userid2 8 Elizabeth Kerns</code>
<code>userid3 1 Melissa Roman</code>
...

# 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**

# 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**
- **Map-reduce is a generic programming model for distributed processing**
  - 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