cloudera®

Aggregating Data with Pair RDDs

Chapter 12

Course Chapters

1	Introduction	Course Introduction
2	Introduction to Hadoop and the Hadoop Ecosystem	Introduction to Hadoop
3	Hadoop Architecture and HDFS	
4	Importing Relational Data with Apache Sqoop	Importing and Modeling Structured Data
5	Introduction to Impala and Hive	
6	Modeling and Managing Data with Impala and Hive	
7	Data Formats	
8	Data File Partitioning	
9	Capturing Data with Apache Flume	Ingesting Streaming Data
10	Spark Basics	**Distributed Data Processing with Spark**
11	Working with RDDs in Spark	
12	**Aggregating Data with Pair RDDs**	
13	Writing and Deploying Spark Applications	
14	Parallel Processing in Spark	
15	Spark RDD Persistence	
16	Common Patterns in Spark Data Processing	
17	Spark SQL and DataFrames	
18	Conclusion	Course Conclusion

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**

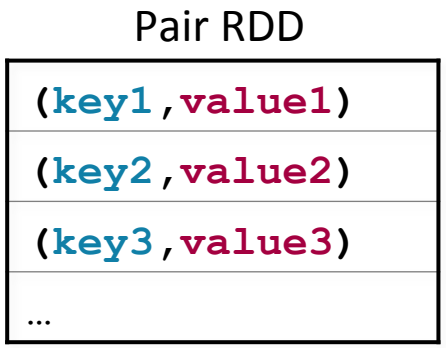
Chapter Topics

Aggregating Data with Pair RDDs | **Distributed Data Processing with Spark**

- **Key-Value Pair RDDs**
- Map-Reduce
- Other Pair RDD Operations
- Conclusion
- Homework: Use Pair RDDs to Join Two Datasets

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.



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**

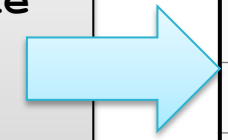
Python

```
> 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))
```

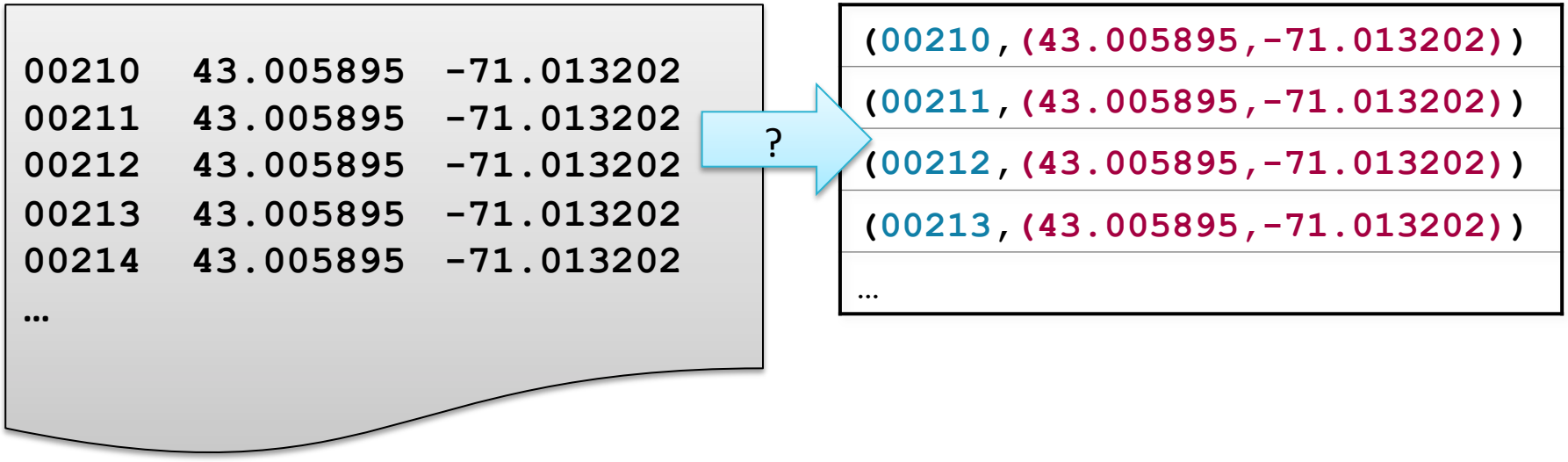
User ID

```
56.38.234.188 - 99788 "GET /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…)
…
```

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)

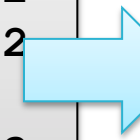


Answer 1: Pairs With Complex Values

```
> 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))))
```

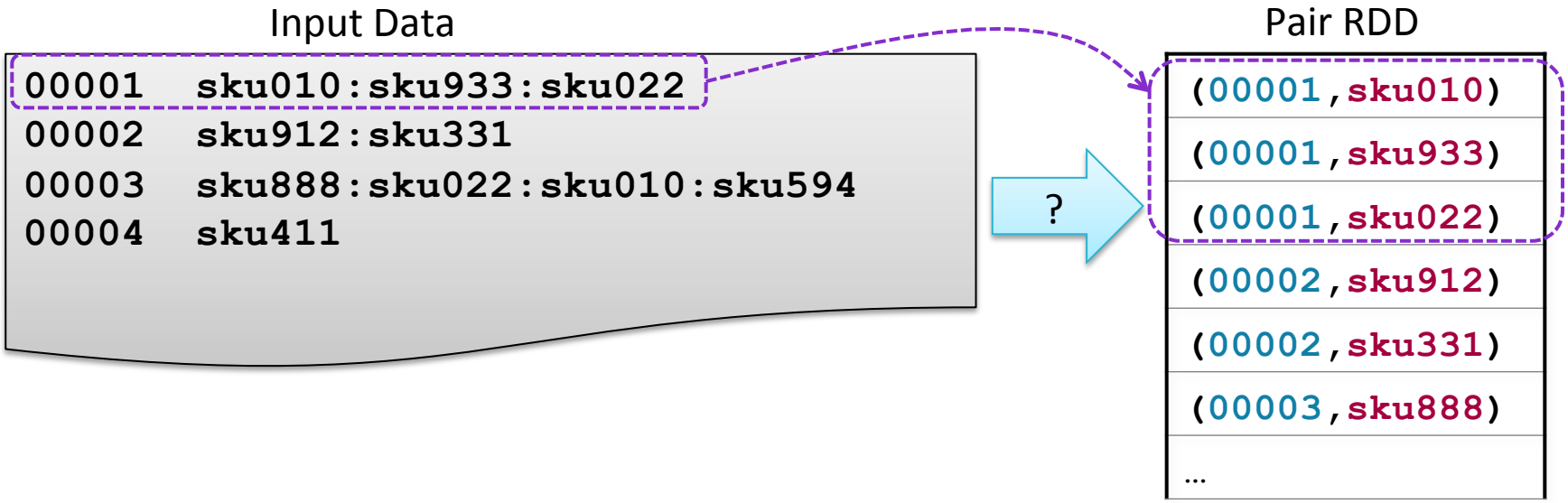
```
00210   43.005895   -71.013202
01014   42.170731   -72.604842
01062   42.324232   -72.67915
01263   42.3929     -73.228483
…
```



```
(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))
```

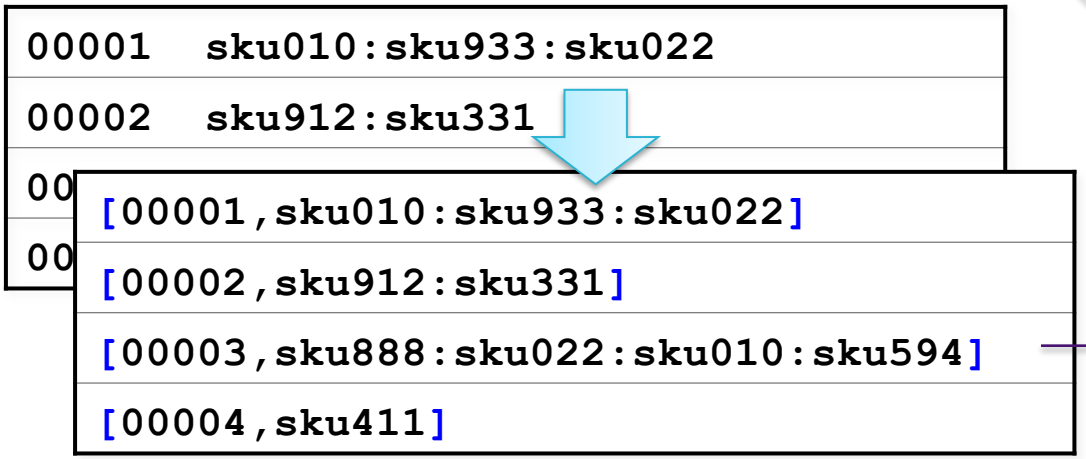
Answer 2: Mapping Single Rows to Multiple Pairs (1)

```
> 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)

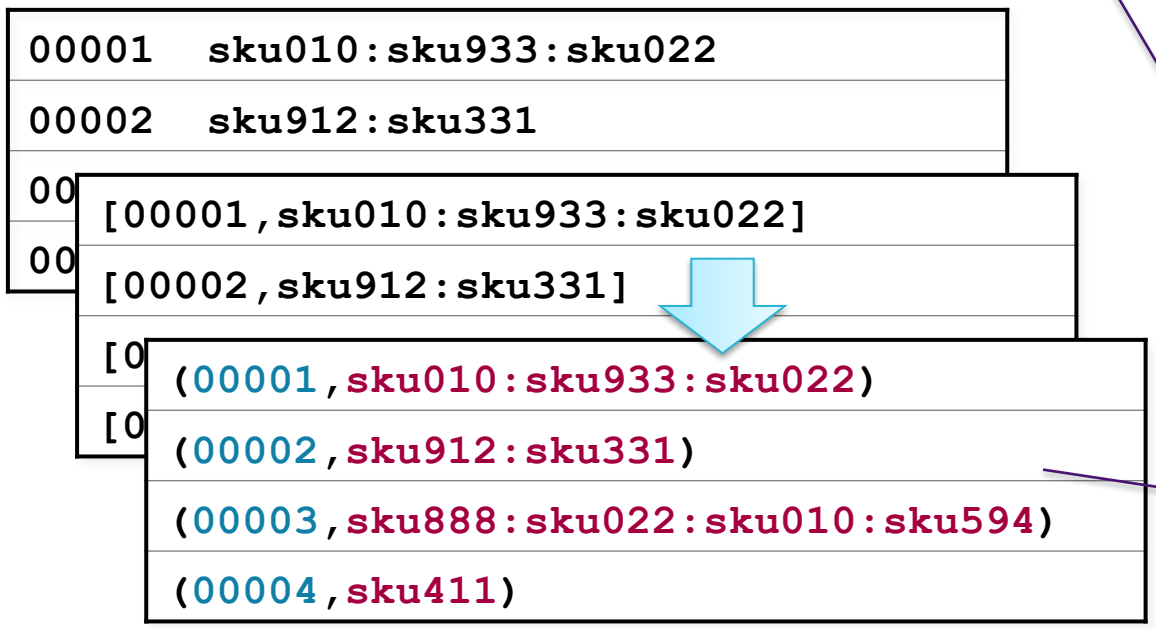
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t'))
```



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

Answer 2: Mapping Single Rows to Multiple Pairs (3)

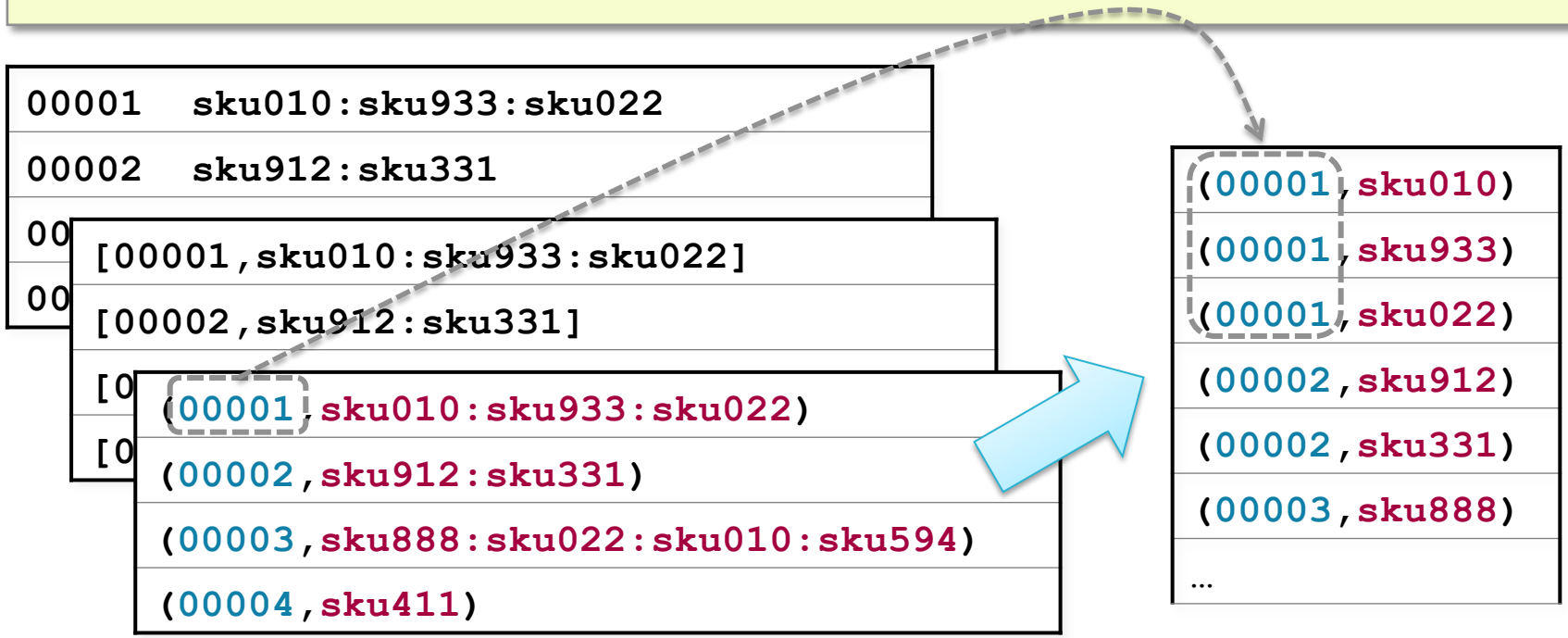
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
```



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(':'))
```



Chapter Topics

Aggregating Data with Pair RDDs | **Distributed Data Processing with Spark**

- Key-Value Pair RDDs
- **Map-Reduce**
- Other Pair RDD Operations
- Conclusion
- Homework: Use Pair RDDs to Join Two Datasets

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

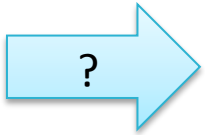
Map-Reduce in Spark

- **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

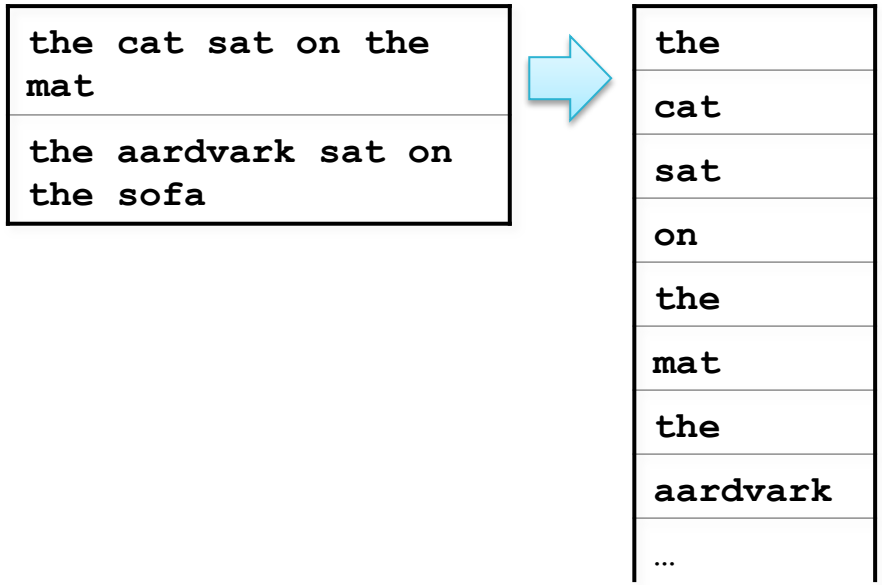
Example: Word Count (1)

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

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

Example: Word Count (2)

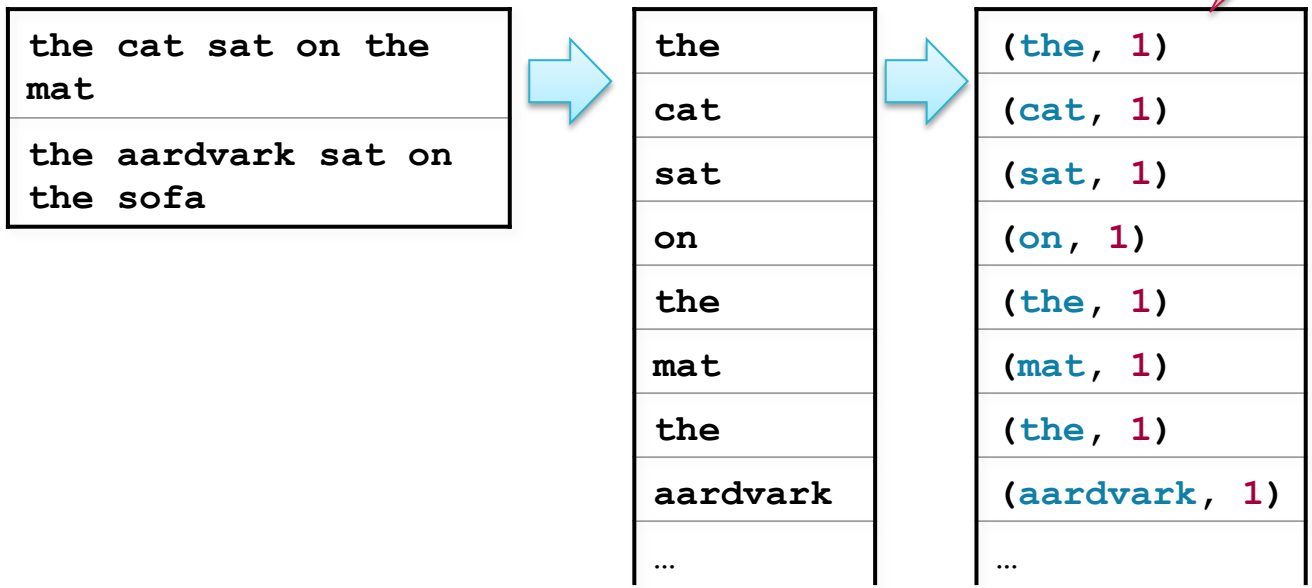
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



Example: Word Count (3)

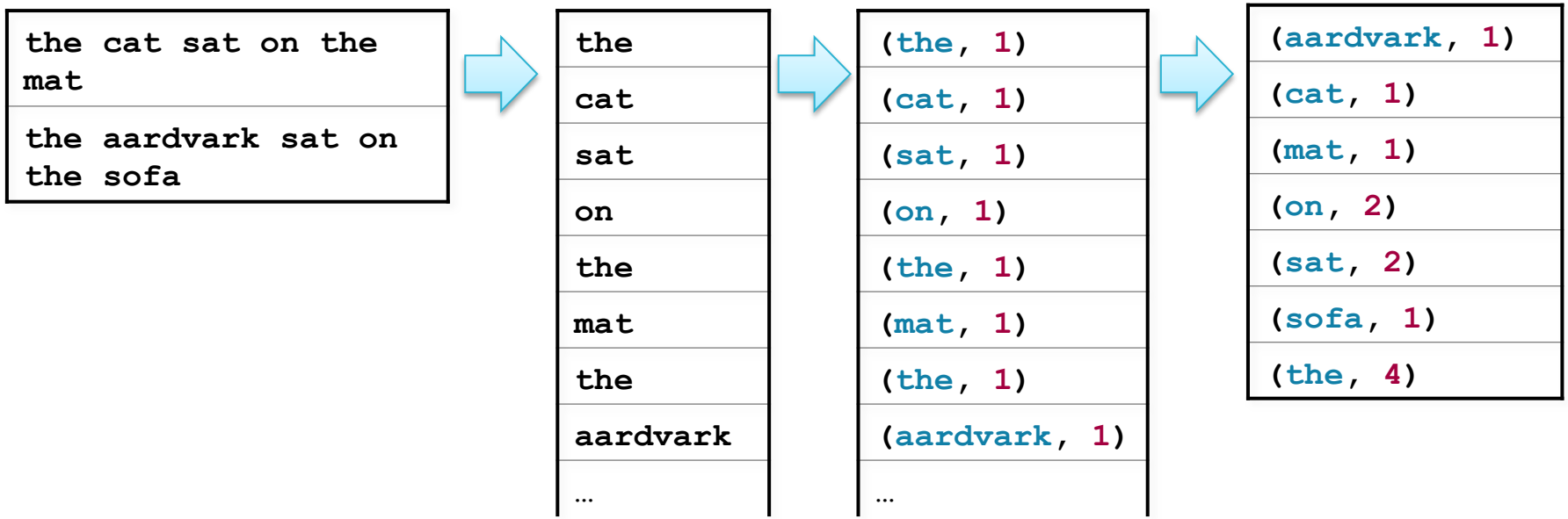
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Example: Word Count (4)

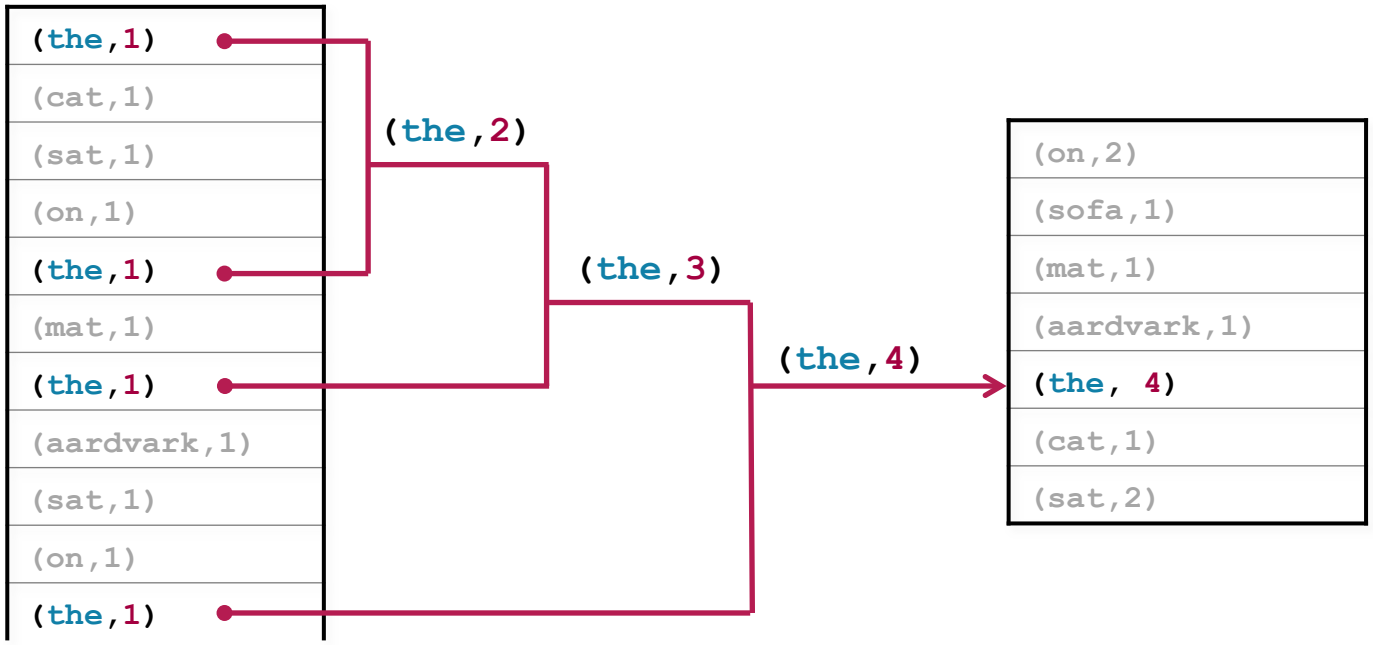
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey (1)

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

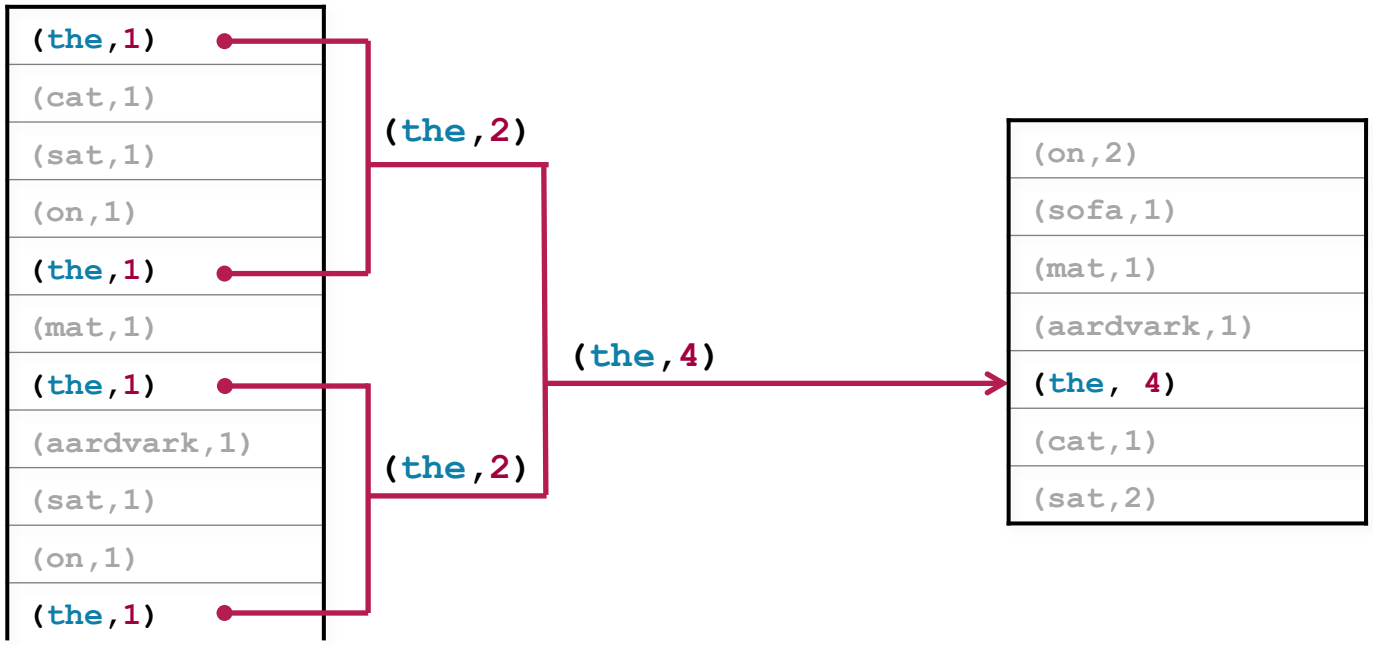
```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



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

Chapter Topics

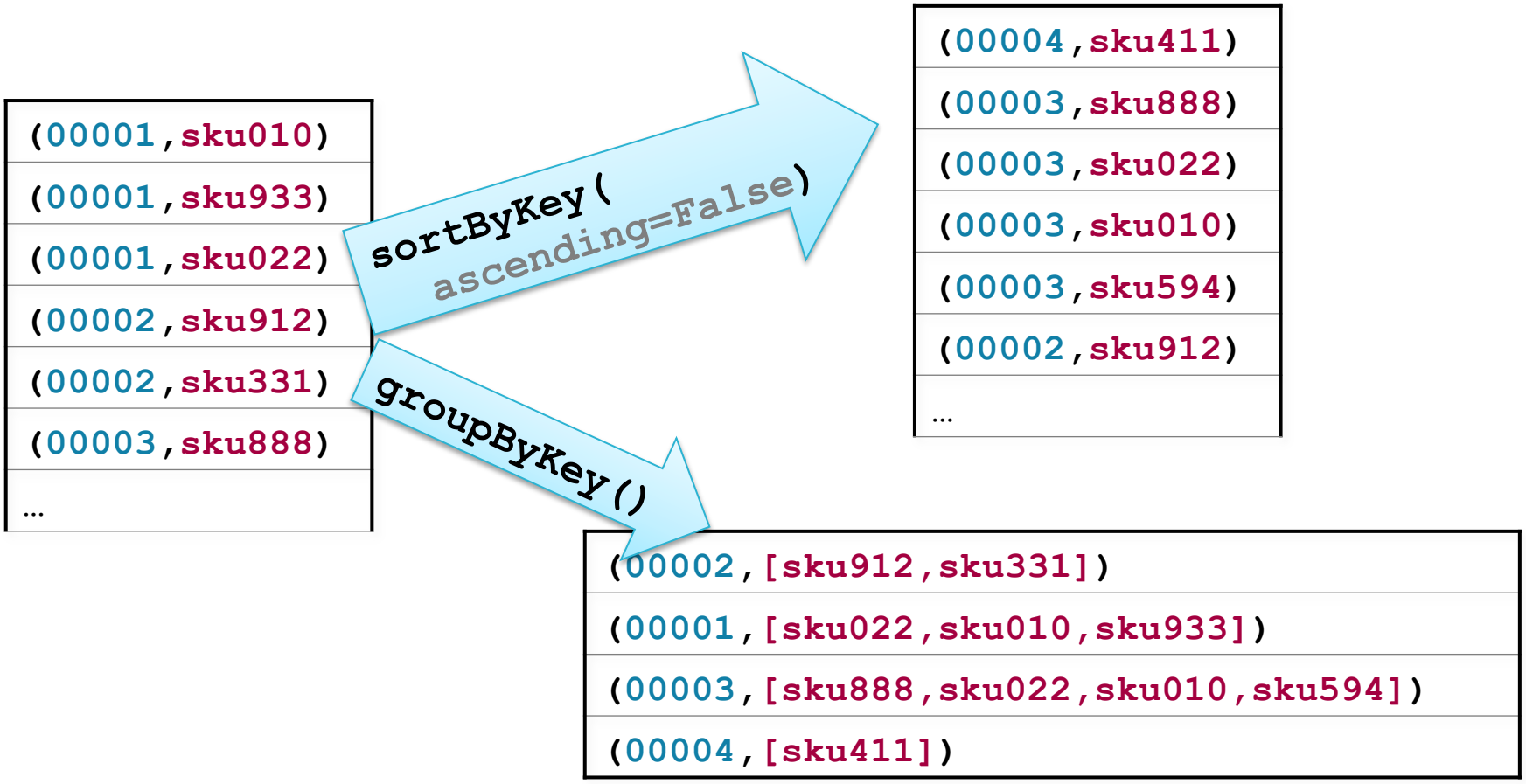
Aggregating Data with Pair RDDs | **Distributed Data Processing with Spark**

- Key-Value Pair RDDs
- Map-Reduce
- **Other Pair RDD Operations**
- Conclusion
- Homework: Use Pair RDDs to Join Two Datasets

Pair RDD Operations

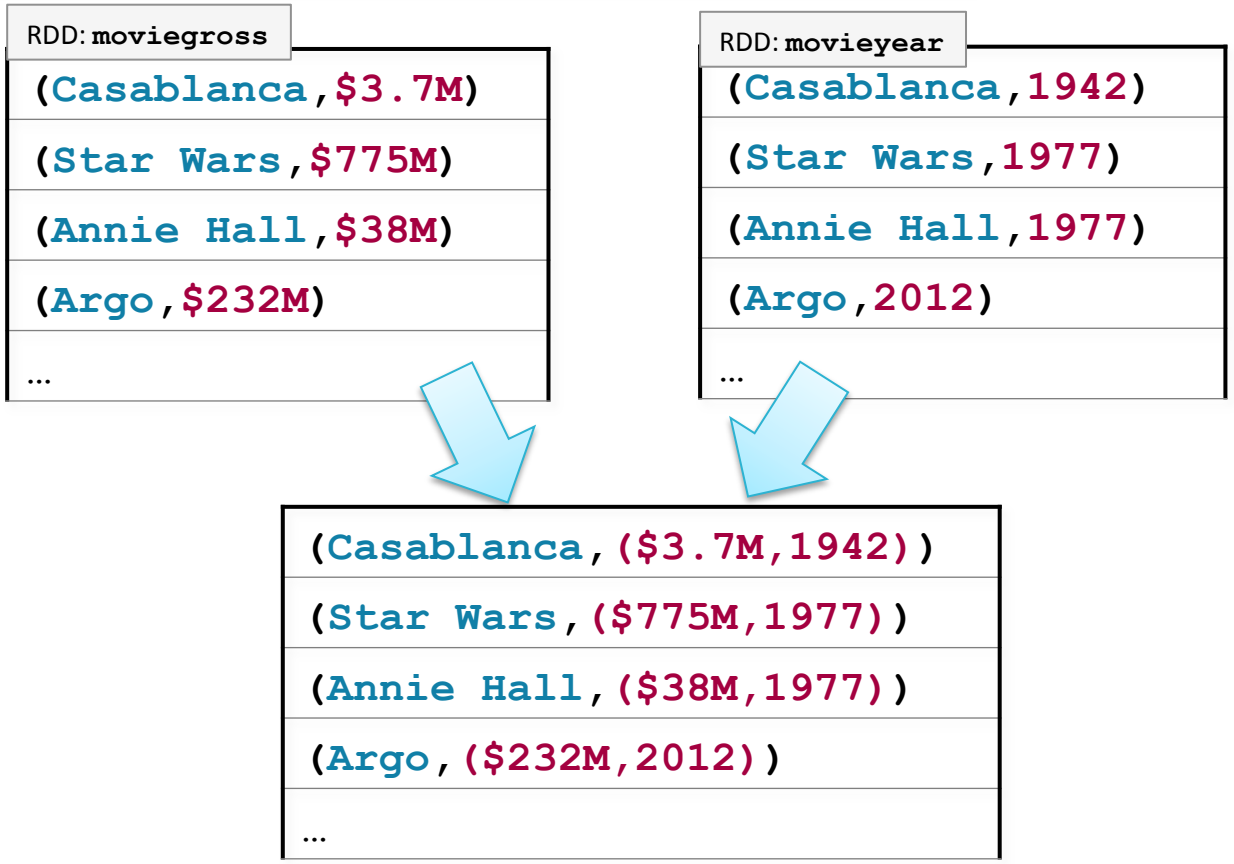
- **In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs**
- **Examples**
  - **`countByKey`** – return a map with the count of occurrences of each key
  - **`groupByKey`** – group all the values for each key in an RDD
  - **`sortByKey`** – sort in ascending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

Example: Pair RDD Operations



Example: Joining by Key

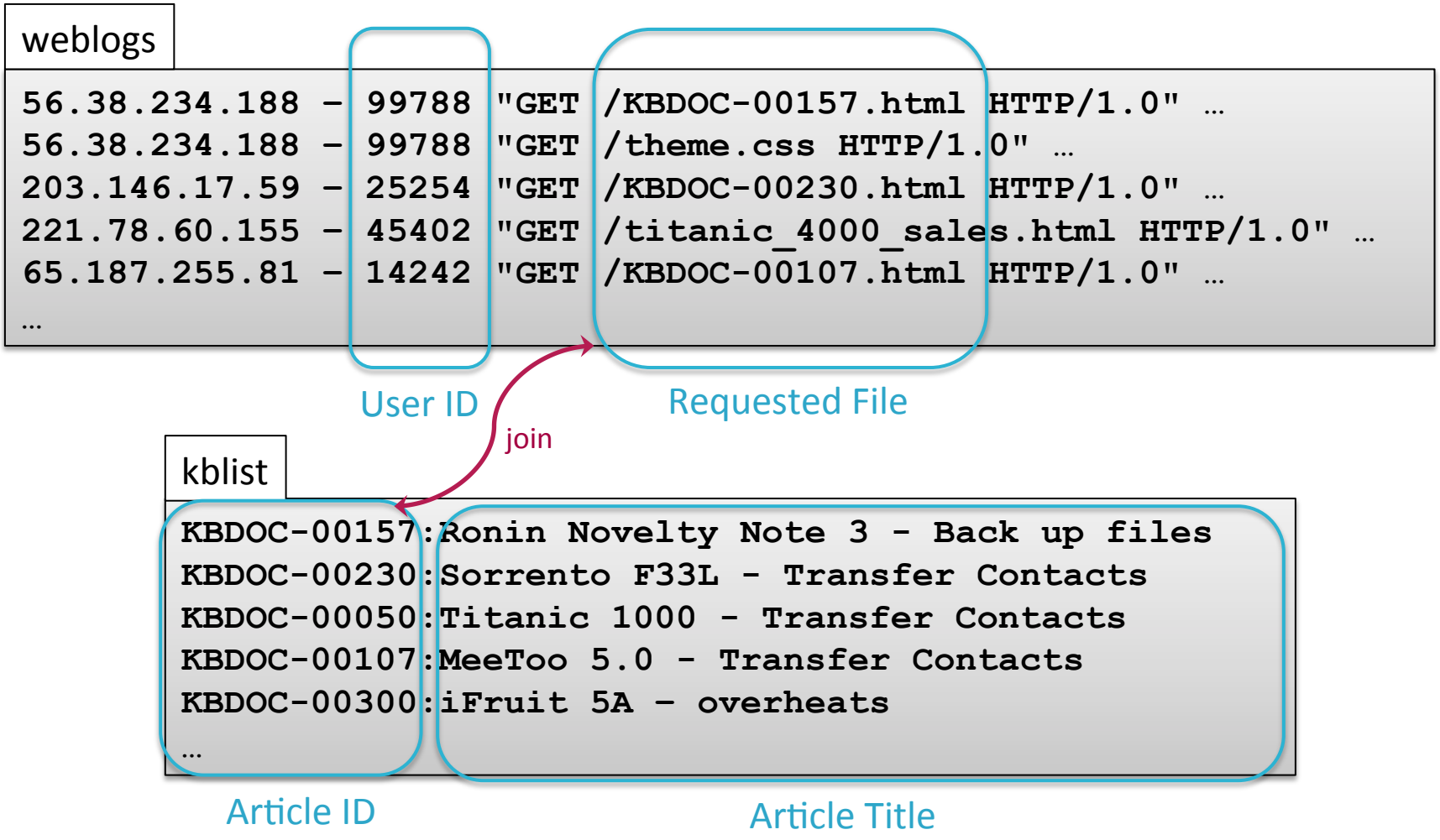
```
> movies = moviegross.join(movieyear)
```



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)

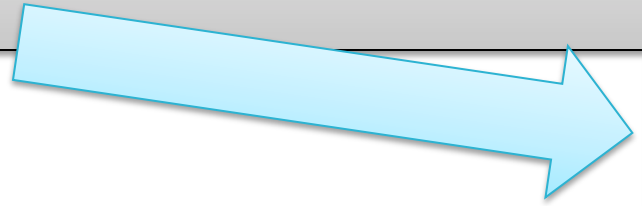
- **Steps**
  1. Map separate datasets into key-value Pair RDDs
     a. Map web log requests to `(docid,userid)`
     b. Map KB Doc index to `(docid,title)`
  2. Join by key: `docid`
  3. Map joined data into the desired format: `(userid,title)`
  4. Further processing: group titles by User ID

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

```
> import re
> def getRequestDoc(s):
    return re.search(r'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
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.
…
```



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]))
```

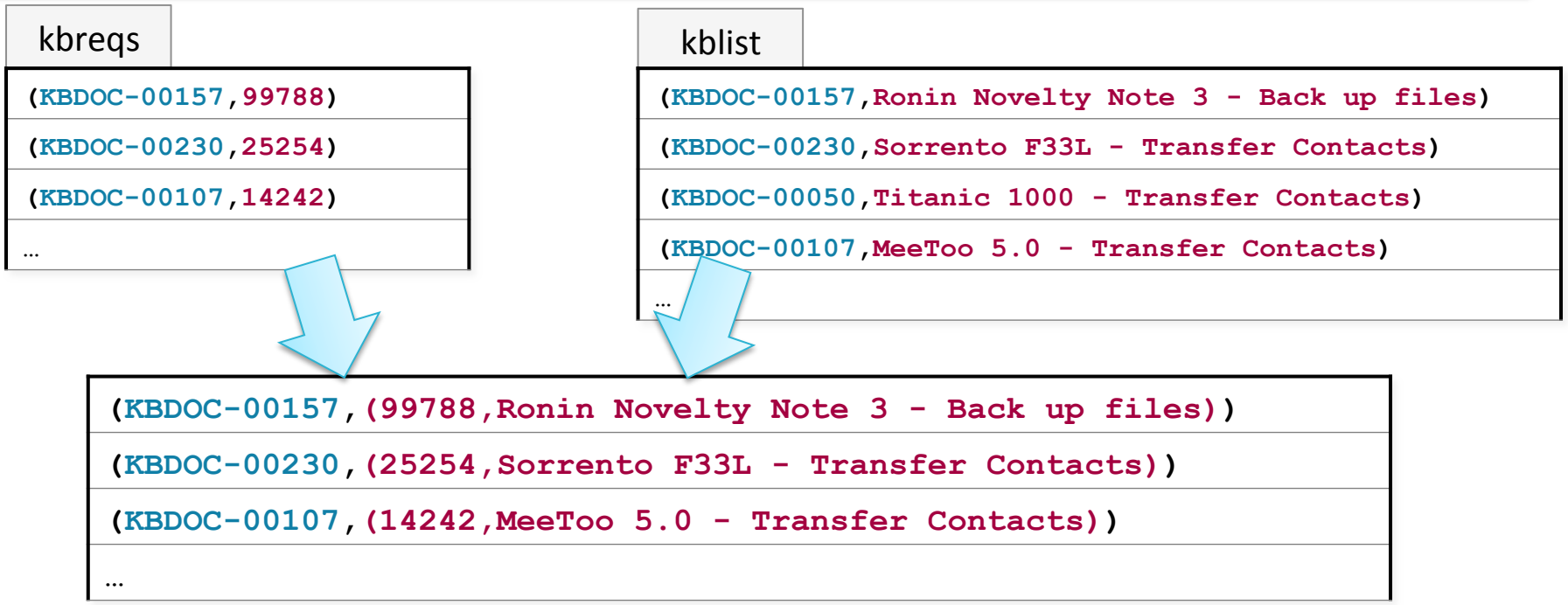
```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A – overheats
…
```

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)
```



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 Cont
user id: 14242
  MeeToo 5.0 - Transfer Contact
  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])

...
```

Aside: Anonymous Function Parameters

- **Python and Scala pattern matching can help improve code readability**

Python

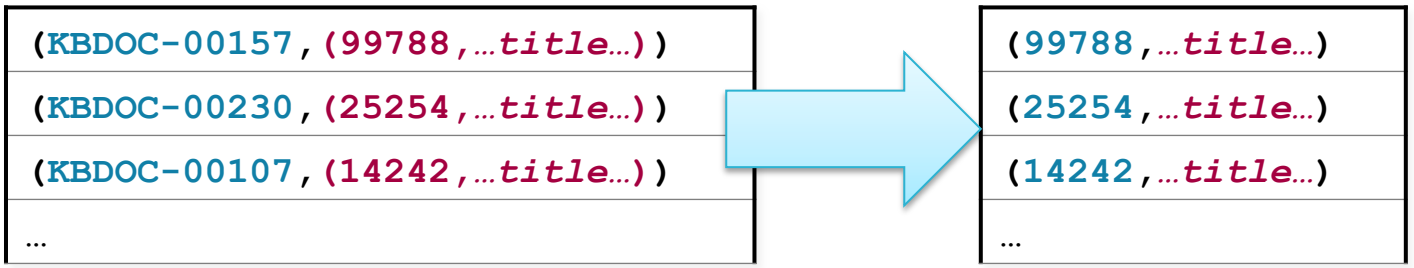
```
> map(lambda (docid,(userid,title)): (userid,title))
```

Scala

```
> map(pair => (pair._2._1,pair._2._2))
```

OR

```
> map{case (docid,(userid,title)) => (userid,title)}
```



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**

Chapter Topics

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- **Conclusion**
- Homework: Use Pair RDDs to Join Two Datasets

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

Chapter Topics

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- Conclusion
- **Homework: Use Pair RDDs to Join Two Datasets**

Homework: Use Pair RDDs to Join Two Datasets

- **In this homework assignment you will**
  - Continue exploring web server log files using key-value Pair RDDs
  - Join log data with user account data
- **Please refer to the Homework description**