Working With RDDs

Working With RDDs

- In this chapter, you will learn
  - How RDDs are created
  - Additional RDD operations
  - Special operations available on RDDs of key/value pairs
  - How MapReduce algorithms are implemented in Spark

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

RDDs

- RDDs can hold any type of element
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dictionaries, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
    - java.io.Serializable by default
    - But customizable
  - Mixed types
- Some types of RDDs have additional functionality
  - Pair RDDs
    - RDDs, each of whose elements is a Key/Value pair
    - Allowing Hadoop developers to leverage their investment in Hadoop
    - https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD
  - Double RDDs
    - RDDs, each of whose elements is a Double
    - DoubleRDD API provides numerous convenience functions for statistical analysis
    - https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.DoubleRDDFunctions
    - http://www.sparkexpert.com/2015/04/25/basic-descriptive-statistics-in-apache-spark/

Creating RDDs From Collections

- You can create RDDs from collections instead of files
  - sc.parallelize(collection)
    - Create an array of 10000 random (floating point) numbers with

```
import random
randomnumlist = [random.uniform(0,10) for _ in xrange(10000)]
randomrdd = sc.parallelize(randomnumlist)
print "Mean: %f" % randomrdd.mean()
```

    - Compute the mean of the values in the RDD
- Useful for
  - Testing
  - Generating data programmatically/algorithmically
  - Integration
    - e.g. Creating an RDD from a dataset generated by another application (like ElasticSearch)

Other General RDD Operations (1 Of 3) (Python)

Transformations	
flatMap	Maps one element in the base RDD to multiple elements in the new RDD (1:*)
distinct	Copies a single instance of each value from one RDD into another
union	Combines all elements of two RDDs into a single new RDD

Sampling	
sample	Creates a new RDD consisting of a sample of elements from a designated RDD
takeSample	Returns an array (not an RDD) of n sampled elements

Double	
Statistical functions	mean, sum, variance, stdev, ...

Other	
first	Returns the first element of an RDD (as a separate RDD)
foreach	Applies a function to each element of an RDD (useful for producing side effects)
top	Returns the largest n elements of an RDD using natural ordering (as a separate RDD)

- https://spark.apache.org/docs/latest/api/python/index.html
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD
- See Transforms.pdf
- See Actions.pdf

Other General RDD Operations (2 Of 3) (Python)

Transformations	Example	Output
flatMap	`sc.parallelize([2, 3, 4]).flatMap(lambda x: range(1, x)).collect()`	`[1, 1, 2, 1, 2, 3]`
distinct	`sc.parallelize([2, 3, 4]).flatMap(lambda x: range(1, x)).distinct().collect()`	`[1, 2, 3]`
Sampling	**Example**	**Output**

sample	`import random`
	`sc.parallelize([random.uniform(0,10) for _ in xrange(10)]).sample(True, .5).collect()`
	`# True: Can elements be sampled multiple times`
		`[9.1674331344832876, 9.1674331344832876, 4.8263093908216037, 5.8318674056047568]`

Double	**Example**	**Output**
Statistical Functions	`import random` `sc.parallelize([random.uniform(0,10) for _ in xrange(10)]).sample(True, .5).mean()`	`5.6889833610103295`
Other	**Example**	**Output**
first	`import random` `sc.parallelize([random.uniform(0,10) for _ in xrange(10)]).sample(True, .5).first()`	`8.6704205204876779`

foreach	`import random`
	`from __future__ import print_function`
	`sc.parallelize([random.uniform(0,10) for _ in xrange(4)]).foreach(print)`
		`1.05694781678`
		`3.47039246703`
		`5.50591983006`
		`3.03530215129`

top(n)	`import random` `sc.parallelize([random.uniform(0,10) for _ in xrange(10)]).sample(True, .5).top(3)`	`[8.6498080625522764, 7.2998596726820857, 6.8250380436858649]`

Other General RDD Operations (3 Of 3)

- Of the operations listed on the previous slide, foreach is a special case
  - It neither returns a new RDD nor a value, making it unlike either a Transformation or an Action
  - It is typically used to produce side effects
    - Printing output
    - Accumulating state in a variable

Example: flatMap and distinct

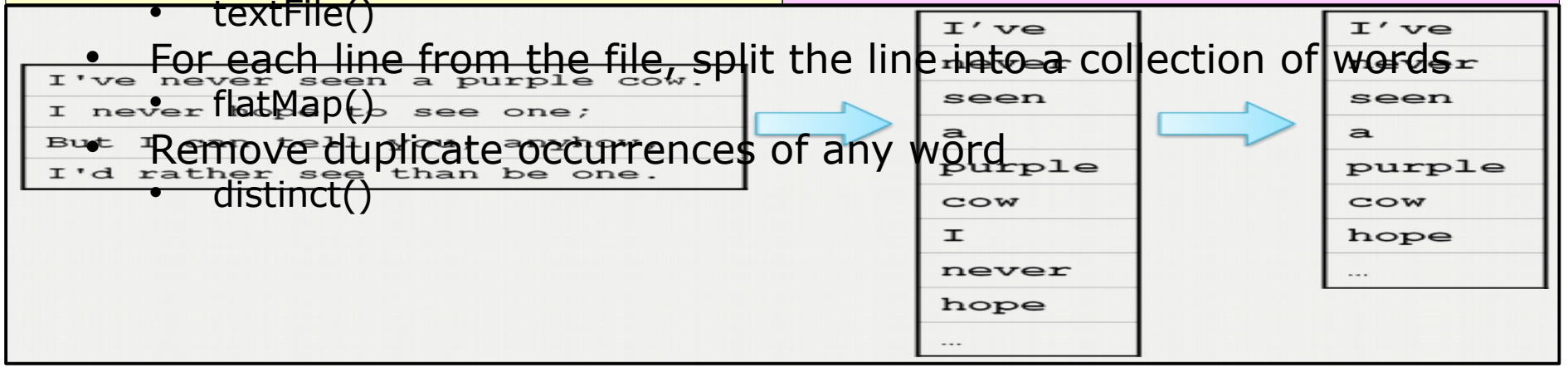
- List the words in a file, removing duplicates
  - Load the file
    - textFile()
  - For each line from the file, split the line into a collection of words
    - flatMap()
  - Remove duplicate occurrences of any word
    - distinct()

Python Shell: pyspark

```
sc.textFile("purplecow.txt") \
  .flatMap(lambda line: line.split()) \
  .distinct()
```

Spark Shell: spark-shell

```
sc.textFile("purplecow.txt")
  .flatMap(line => line.split("\\W"))
  .distinct()
```



Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: 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 of any type

Pair RDDs

```
(key1, value1)      # In our courseware, we use blue to identify a Pair RDD key
(key2, value2)      # and red to identify a Pair RDD value
(key3, value3)
```

  - e.g., sorting, joining, grouping, counting, summing, etc.
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions

Creating Pair RDDs

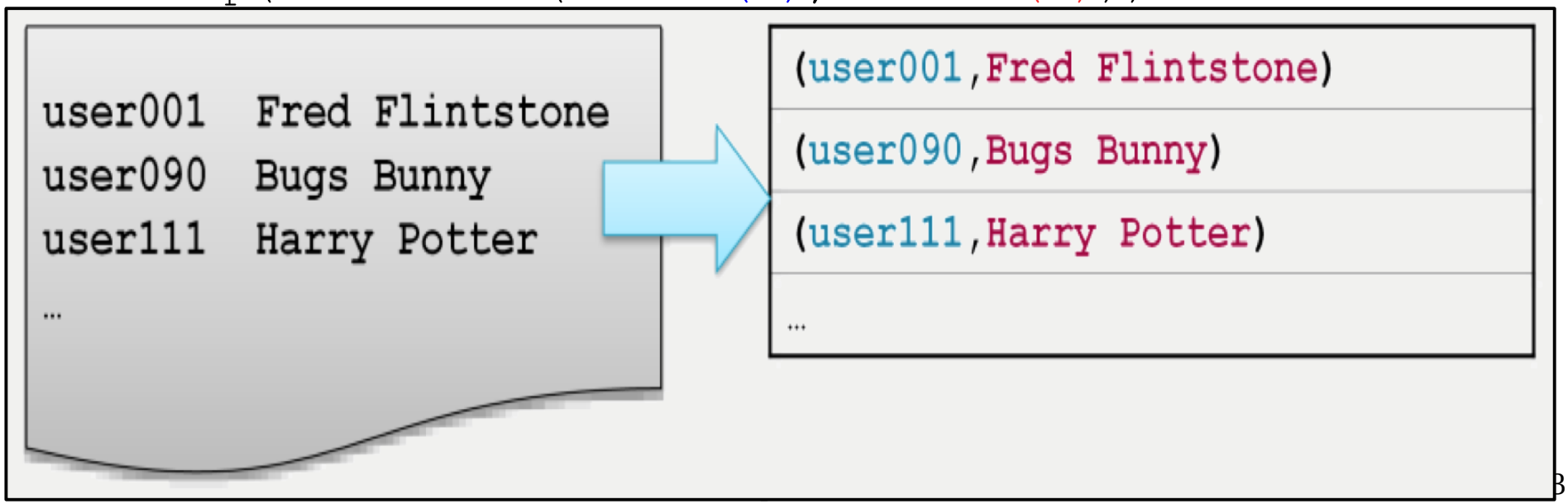
- The first step in most workflows is to get the data into key/value form, which requires you to make strategic decisions about
  - What should the RDD be keyed on?
  - What should be the value associated with each key?
- Commonly used functions that create Pair RDDs
  - map
  - flatMap
  - flatMapValues
  - keyBy

Example: A Simple Pair RDD

- Create a Pair RDD from a tab-delimited file

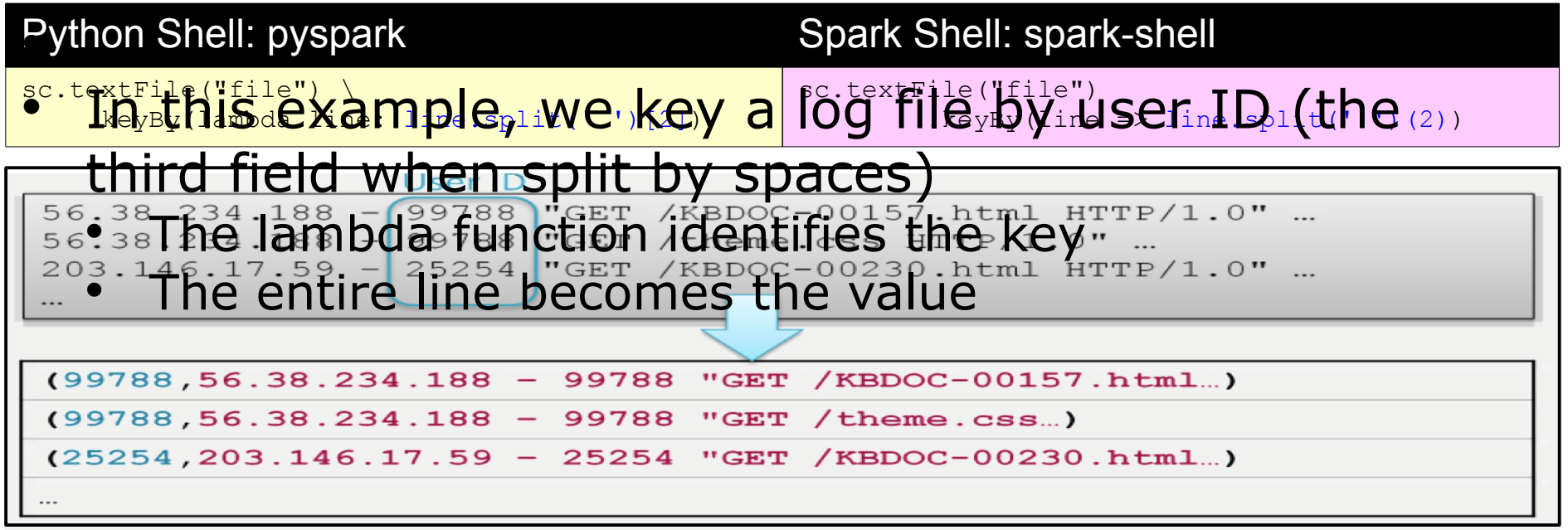
```
users = sc.textFile("file") \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0], fields[1]))
```

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



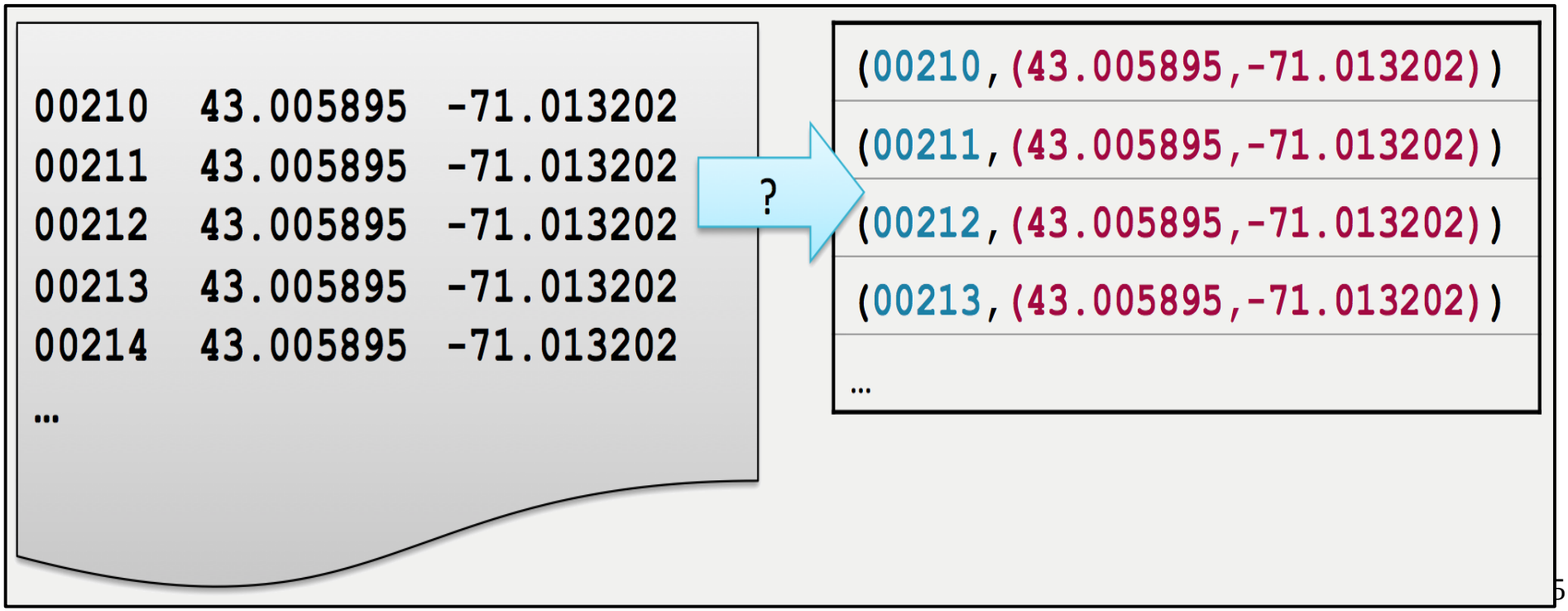
Example: Keying Web Logs By User ID

- In this example, we key a log file by user ID (the third field when split by spaces)
- The lambda function identifies the key
- The entire line becomes the value



Question 1: Pairs With Complex Values

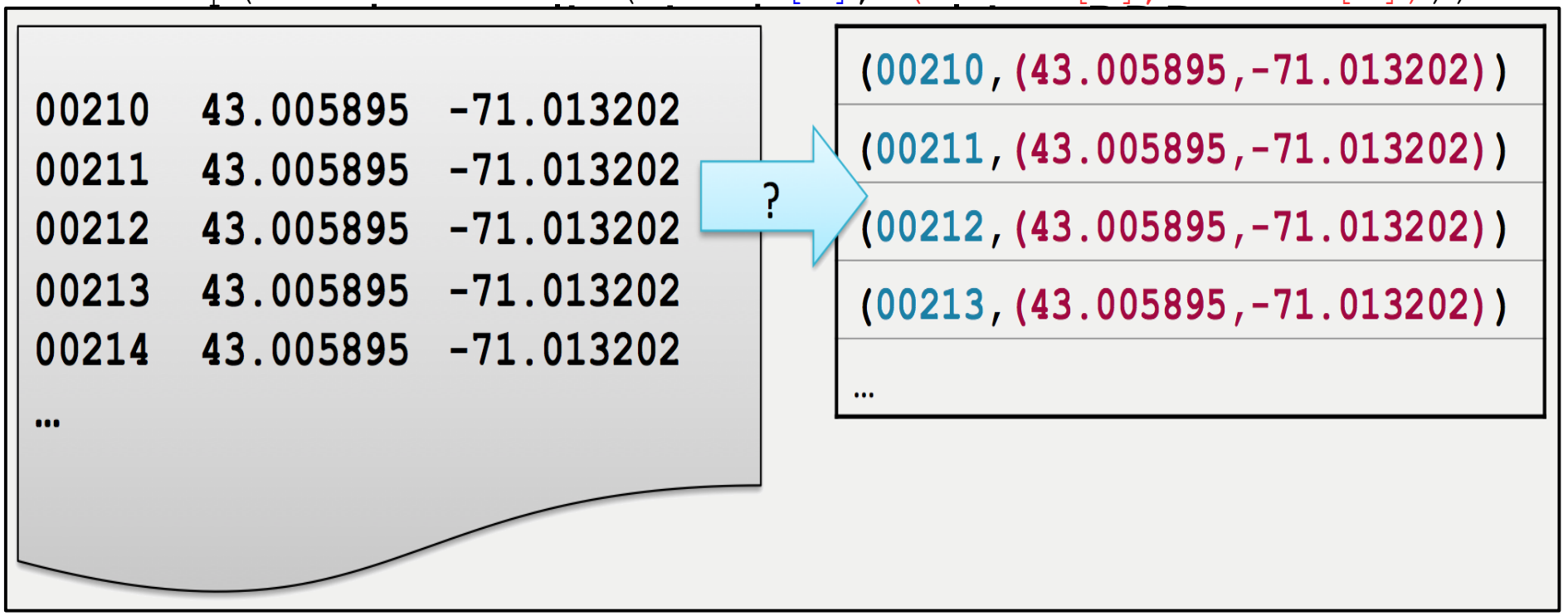
- How would you do this?
  - Input:
    - A list of postal codes with latitude and longitude
  - Output:
    - Postal code (key) and latitude/longitude pair (value)



Answer 1: Pairs With Complex Values

```
sc.textFile("file") \
    .map(lambda line: line.split()) \
    .map(lambda fields: (fields[0], (fields[1], fields[2])))
```

- Use the map function when each line in a file

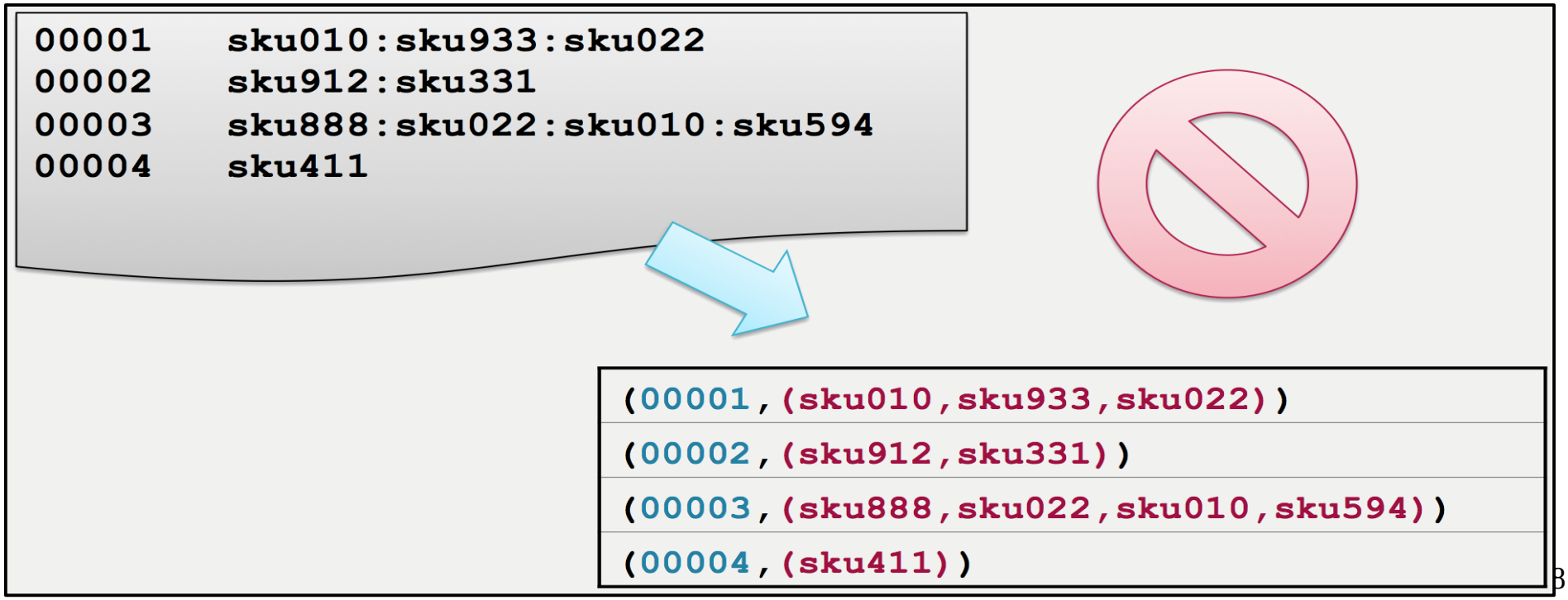


Question 2: Mapping Single Rows To Multiple Pairs (1 Of 2)

- How would you do this?
  - Input:
    - Order numbers with a list of values in the order
  - Output:
    - Order key paired individually with each value
- Would the map function be appropriate here?

Question 2: Mapping Single Rows To Multiple Pairs (2 Of 2)

- The map function alone would not be suitable here
  - The map function is suitable when each line in a file corresponds to a line in the resulting RDD (1:1)
  - Here, each line in a file corresponds to multiple lines in the resulting RDD (1:*)



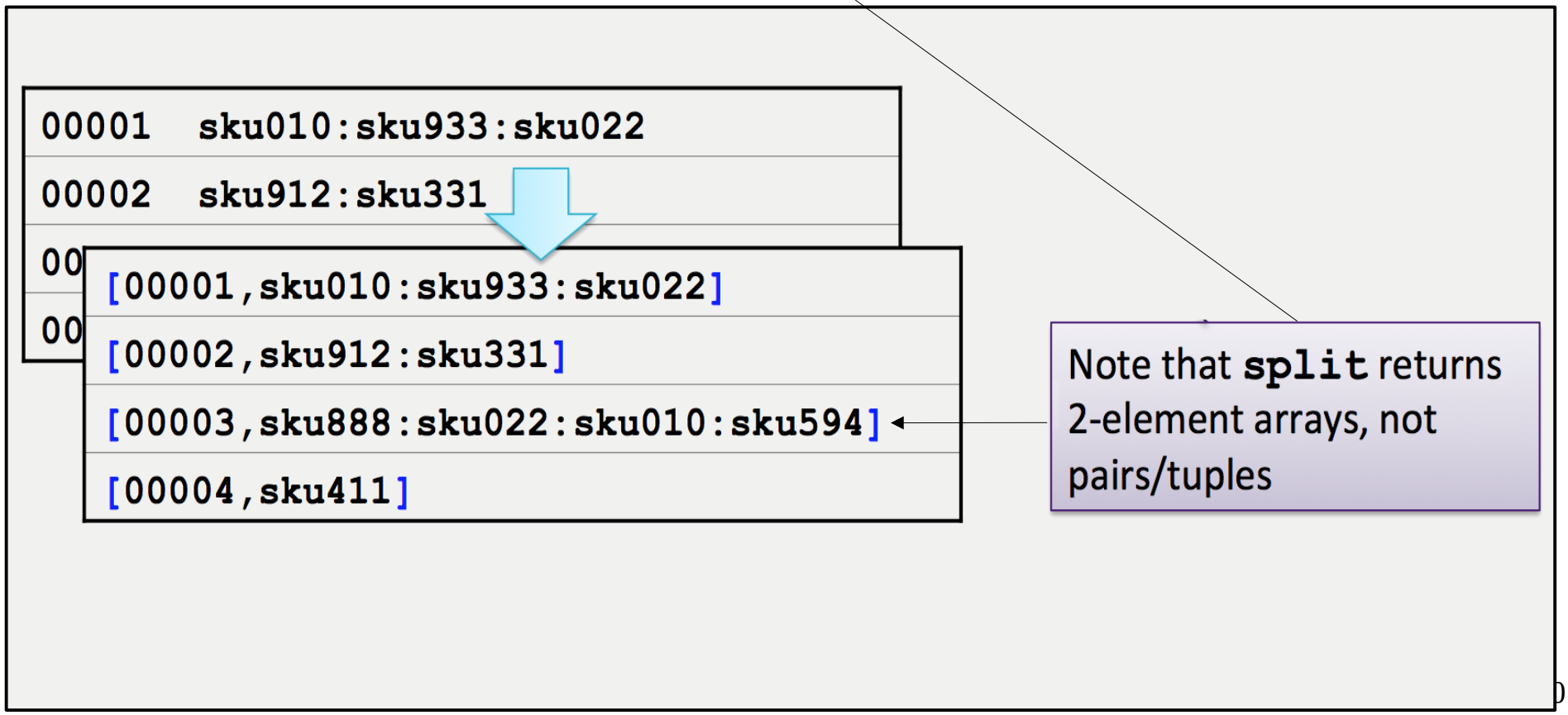
Answer 2: Mapping Single Rows To Multiple Pairs (1 Of 7)

```
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 Of 7)

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

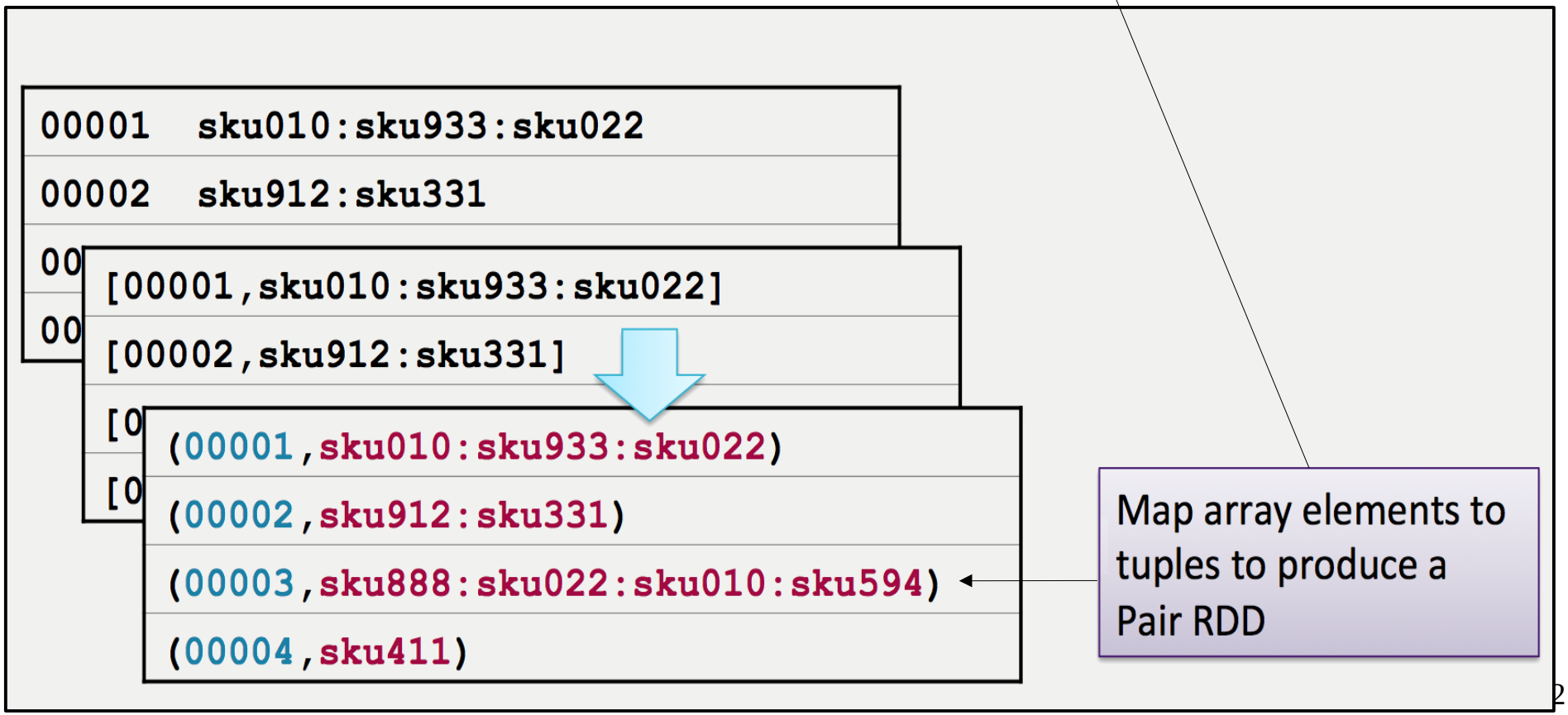


Answer 2: Mapping Single Rows To Multiple Pairs (3 Of 7)

- Start by separating each key from its list of values
  - Keys are delimited from values by the '\t' character
- Note that the resulting RDD here is not a Pair RDD, because the rows are arrays of two elements (strings), not pairs (tuples)
  - This is because we map the “split” function to elements of the line, which returns an array
    - Thus the use of square braces [] instead of parens () to encapsulate the result
      - [] - Array
      - () - Pair

Answer 2: Mapping Single Rows To Multiple Pairs (4 Of 7)

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

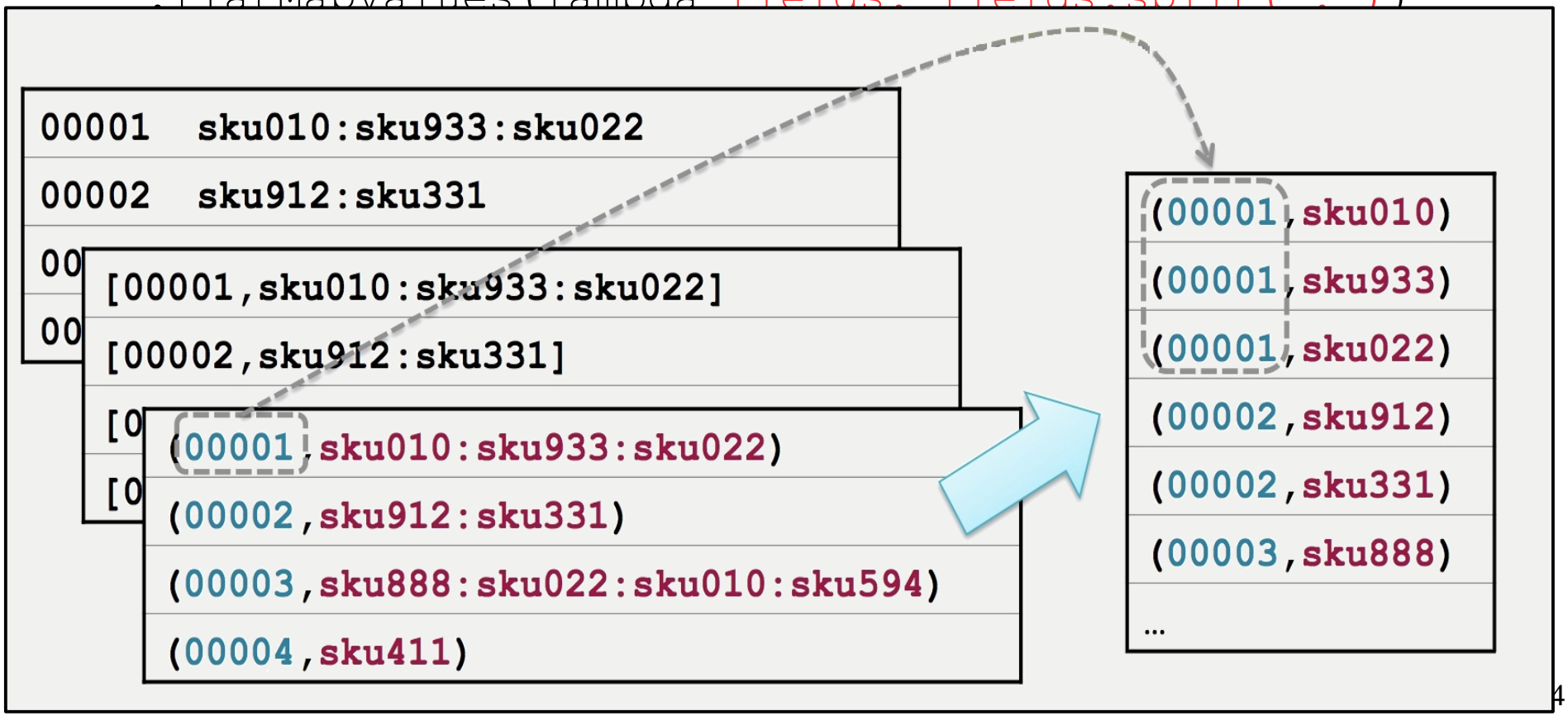


Answer 2: Mapping Single Rows To Multiple Pairs (5 Of 7)

- Next, convert each two element array into a pair using the map function
  - Incidentally, Python Spark actually treats an RDD of two element arrays as a Pair RDD, but Scala doesn't, so it is best practice to convert them to pairs explicitly so as to avoid confusion

Answer 2: Mapping Single Rows To Multiple Pairs (6 Of 7)

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



Answer 2: Mapping Single Rows To Multiple Pairs (7 Of 7)

- Finally, apply flatMapValues
  - This is necessary because in this example each line in the file corresponds to multiple records in the RDD
  - The flatMap and flatMapValues functions are designed for precisely this scenario
    - The difference is that flatMap processes a list of values while flatMapValues processes the values in a list of pairs
- Note the function passed as an argument to the flatMapValues function
  - Its input is the pair value (which is why the value is in red; the key is not passed to this function)
  - This function (parameter) outputs an array of values
- Spark will automatically pair the key individually which each element in the resulting array

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

Food For Thought (1 Of 5)

- Problem:
  - Suppose that you want to count the number of occurrences of each word appearing in a book (perhaps to build an index for the book)
  - How might you go about it?
- Strategy:
  - Start with a set of index cards
  - For each word in the book write the word on a separate index card
  - Next, group the index cards by word, so that each group has the same word
  - Finally, for each group ...
    - Count the number of cards in the group (representing the frequency with which a word appeared in the book)
    - Write the word and its frequency on a separate index card

Food For Thought (2 Of 5)

- Problem:
  - What would you change in the aforementioned strategy if 10 people volunteered to help you count words?
- Strategy:
  - Assign each person a different chapter
  - Have each person apply the strategy described on the previous slide individually to their chapter
- Benefits
  - Parallelized counting
- Costs
  - No one person has the complete answer
    - S/he can tell you the frequency with which each word appears in his/her particular chapter, but not in the entire book
  - We need to aggregate the work of those 10 people into a single coherent result

Food For Thought (3 Of 5)

- Problem:
  - How might you aggregate the chapter by chapter breakdowns?
- Strategy:
  - Collect the new batch of index cards from each person
    - Each card consisting of a word and a number representing the frequency with which that word appeared in a particular chapter
  - Next, group these cards by word so that all of the cards for a given word are grouped together, regardless of the chapter they come from
  - Then, for each group (word) compute the sum of the individual numbers from each card (representing the frequency with which each word appeared in a particular chapter) to determine the frequency with which that word appeared in the book
  - Finally, write the word and the sum of its occurrences on a new index card
-

Food For Thought (4 Of 5)

- Problem:
  - What would you change in the aforementioned strategy if 10 other people volunteered to help you compute the word totals?
- Strategy:
  - Assign each person 1+ card groups, each group representing the occurrence of a single word across multiple chapters
  - Then, have each person apply the strategy described on the previous slide individually to their card groups
- Benefits
  - Parallelized summation
- Costs
  - A bit more complexity

Food For Thought (5 Of 5)

- These are the ideas at the core of the programming paradigm known as MapReduce

MapReduce

- MapReduce 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
      - Though Hadoop does support Mapper or Reducer chains, and "Map-Only" jobs
    - Job output is saved to HDFS
      - Which then becomes the input to the next job
    - This implementation limits performance, particularly when applied to complex workflows involving many steps
- Spark implements MapReduce with much greater flexibility than Hadoop
  - Map functions can be pipelined
    - Without the need for an intervening (expensive) Shuffle and Sort
  - Results are stored in memory (or on a local disk, rather than HDFS)
    - Unlike Hadoop, in which operations are typically performed in separate applications which are then coordinated by a separate workflow manager, with output written to HDFS between each application

MapReduce In Spark

- MapReduce in Spark works on Pair RDDs
- Map phase
  - Operates on one record at a time
  - “Maps” each record to one or more new records
  - map, flatMap and keyBy are examples of map operations
- Reduce phase
  - Works on Map output
  - Consolidates (organizes/aggregates) multiple records
  - reduceByKey (which, for those of you familiar with Hadoop, works like a combiner and reducer), sortByKey and mean are examples of reduce operations
- In other words, we implement MapReduce in Spark, not by using the Hadoop APIs, but by using Spark's equivalents of these API operations

MapReduce Example: Word Count

Example: Word Count (1 Of 8)

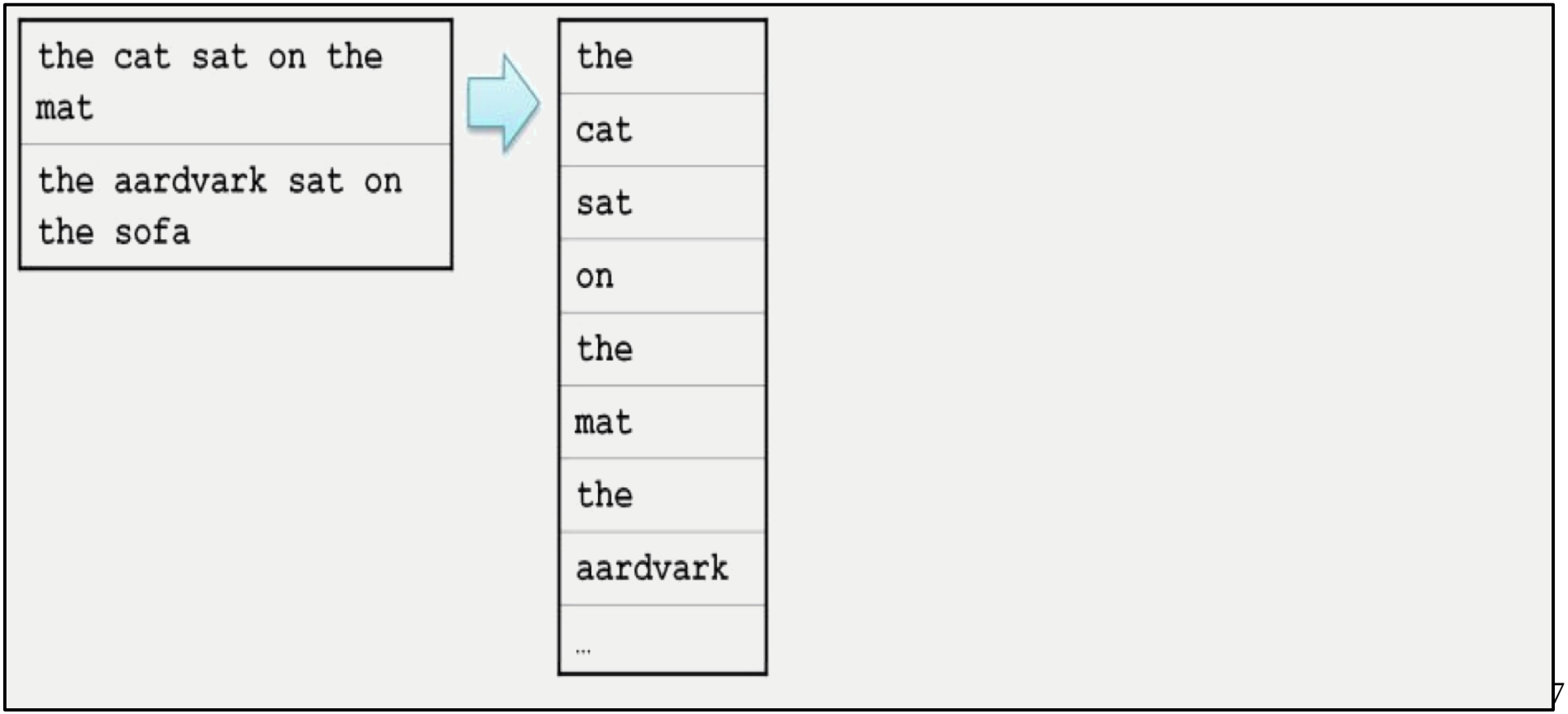
```
counts = sc.textFile("file")
```

Example: Word Count (2 Of 8)

- As in previous examples, start by reading one line of text at a time from an input file

Example: Word Count (3 Of 8)

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

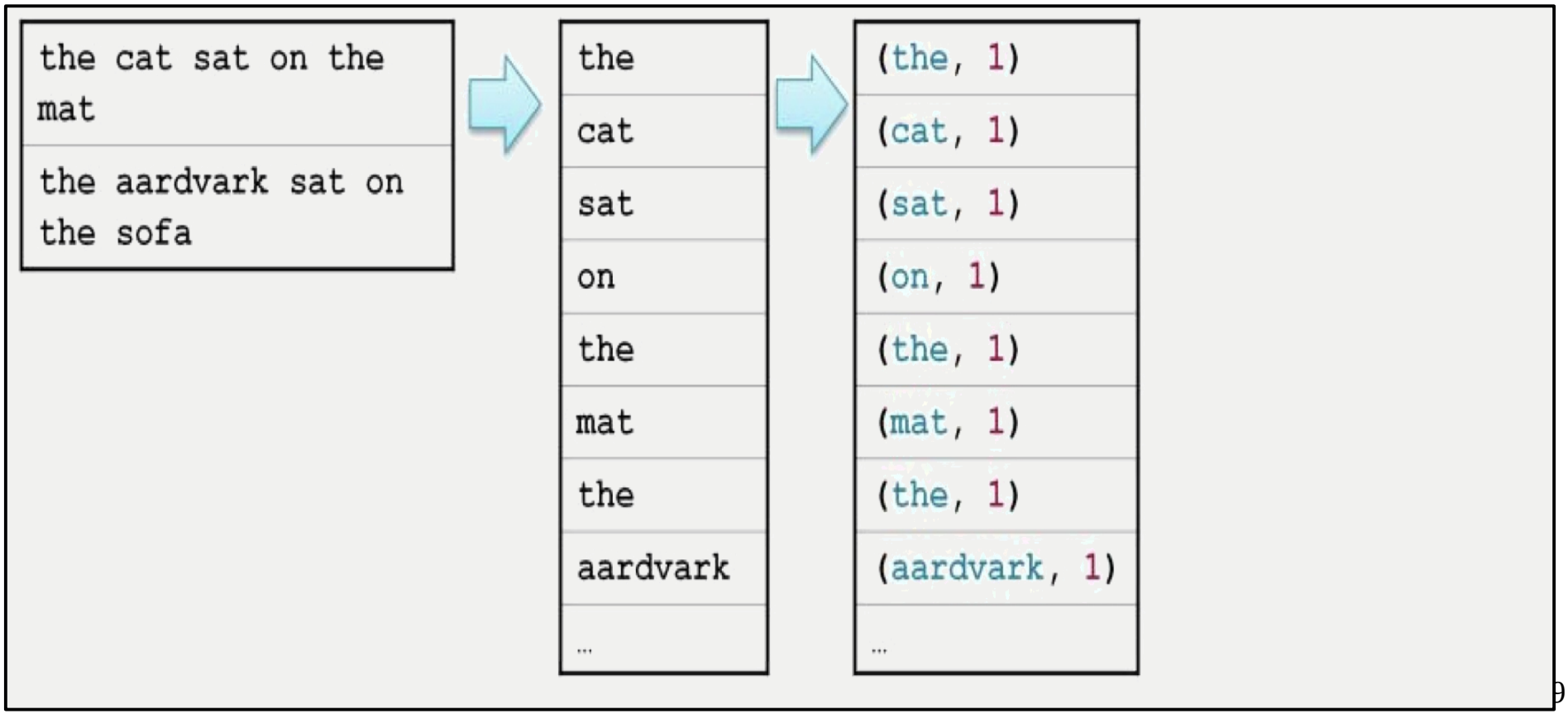


Example: Word Count (4 Of 8)

- The flatMap transformation takes a single record (in this case a line) and produces one or more records (in this case individual words from the line)
  - Note that the map function would not produce the effect we want
    - Its output for a single line of input would be a single line of output consisting of a list of words, not the list of words itself, which is what we want

Example: Word Count (5 Of 8)

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

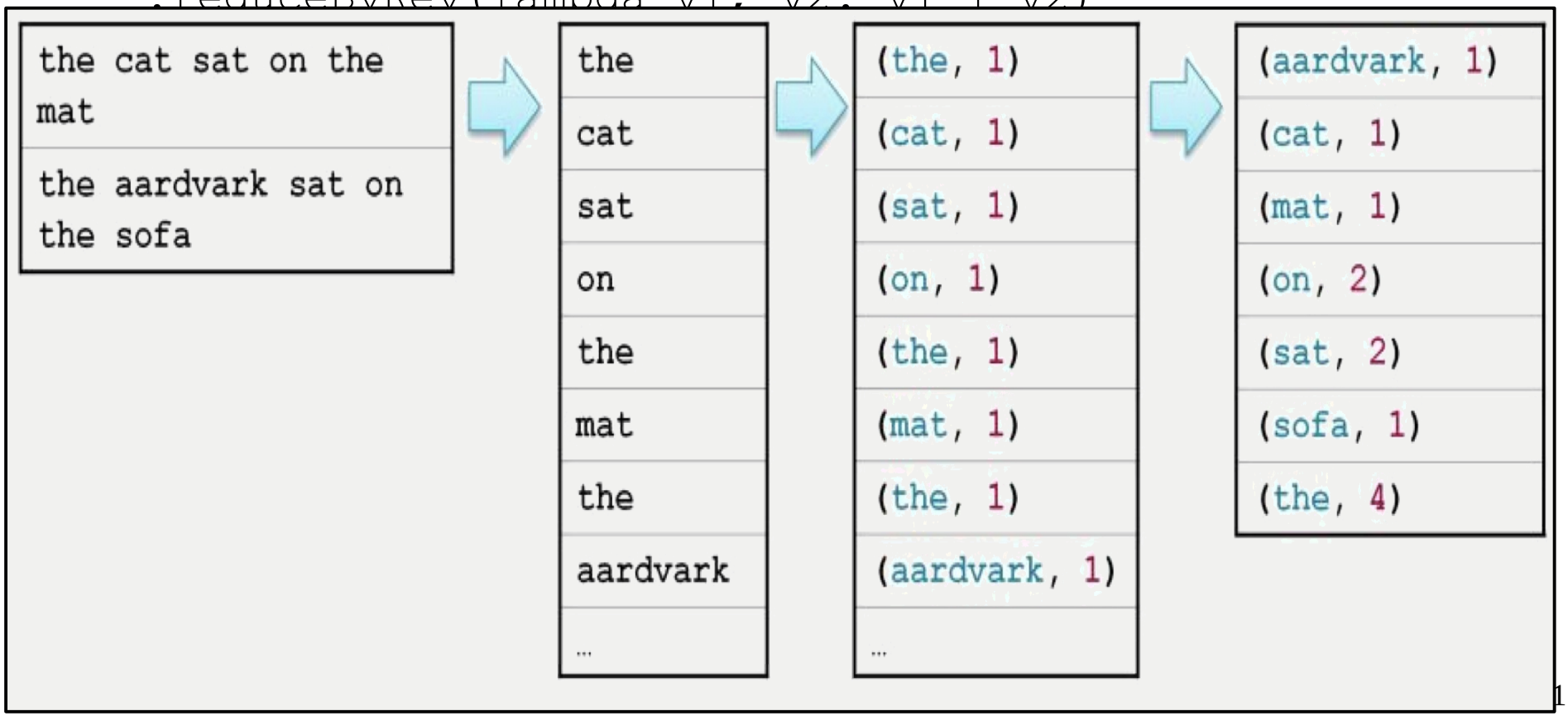


Example: Word Count (6 Of 8)

- For each word, create a pair whose key is the word and whose value is the number 1
  - The value represents a single “occurrence” of the key from this line

Example: Word Count (7 Of 8)

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

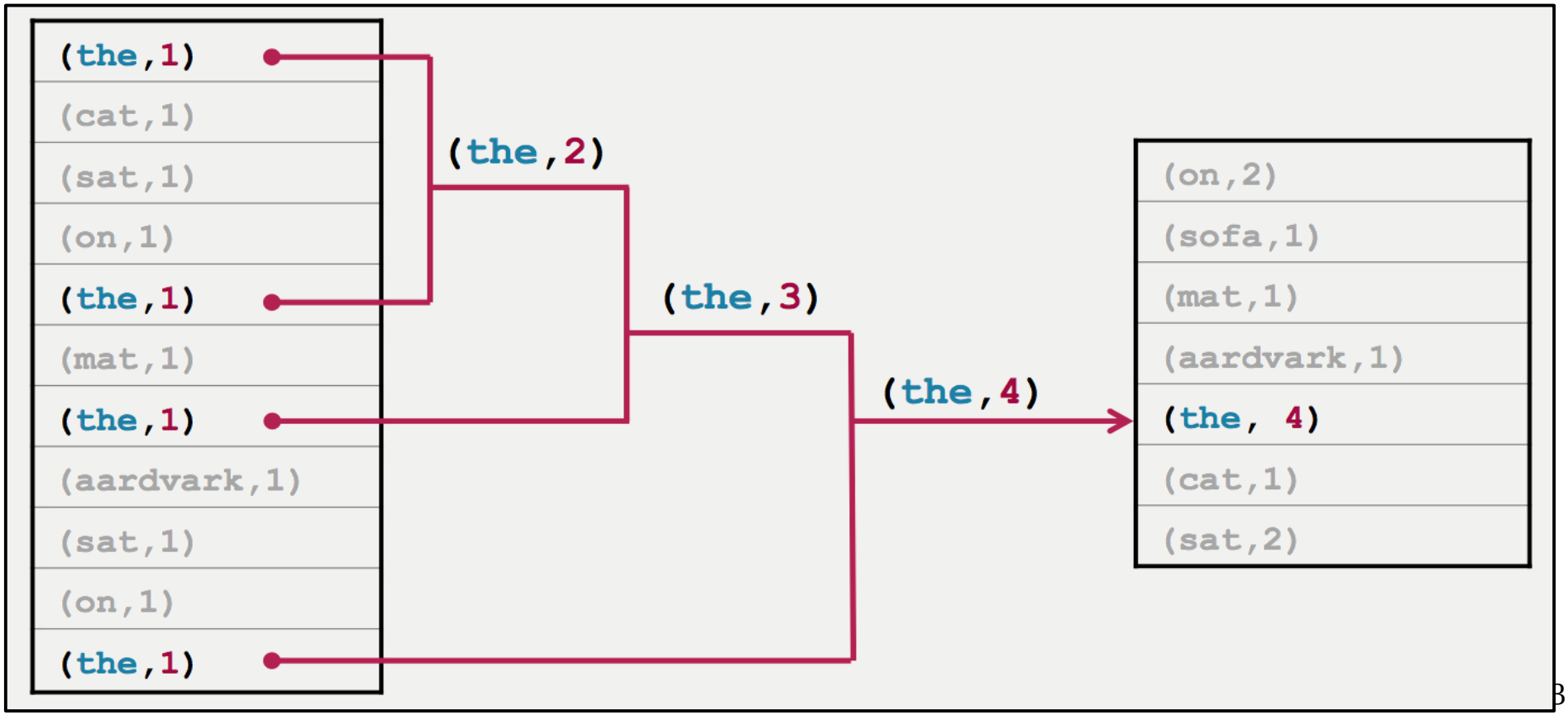


Example: Word Count (8 Of 8)

- For those of you familiar with functional programming in Python, Spark's reduceByKey function is quite similar to Python's built-in reduce function
  - It takes all of the values with a common key and groups them together for some subsequent operation passed as a parameter to the function
- For those of you familiar with Hadoop MapReduce, Spark's reduceByKey function is somewhat different from Hadoop's built-in reduce function
  - Acting as both a “Combiner” and “Reducer”

ReduceByKey (1 Of 4)

- The function passed to ReduceByKey must be:
  - Binary (take 2 arguments)
  - It combines values from 2 instances of the same key



ReduceByKey (2 Of 4)

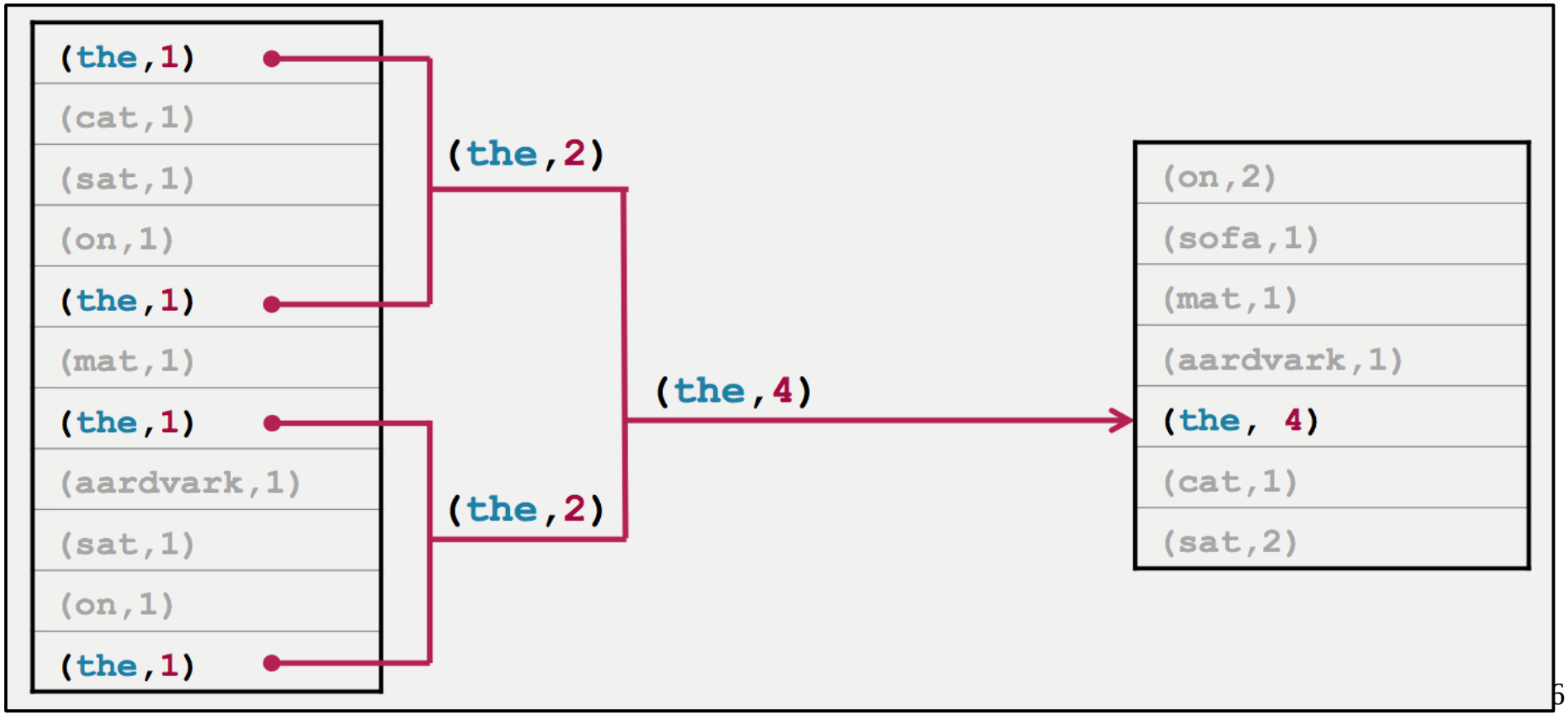
- ReduceByKey takes two values sharing a common key and combines them into a single value, producing a new pair consisting of the key and combined value
  - Note that this explanation is conceptual
    - It captures how to express the reduce operation
    - The actual implementation could be different
  - Note also that the the dataset is likely to be distributed across many nodes of the cluster, meaning the algorithm must run
    - On each node of the cluster to compute the occurrences of each word on that node
    - Across multiple nodes of the cluster to compute the occurrences of each word across the cluster
  - Because of this distribution of data, there is no guarantee of the order in which the result will be produced, hence the requirement that the function be both associative and commutative

ReduceByKey (3 Of 4)

- For those familiar with Hadoop MapReduce, reduceByKey works differently than a Hadoop Reducer
  - In Hadoop, a reducer is passed a key and a list of values, and can aggregate those values any way it wants
  - In Spark, a reducer is repeatedly passed two values for the same key, (but not the key itself) and aggregates just those two values
- A common example of a Reducer in Hadoop is the average reducer, which takes a list of values and calculates their average
  - You could not do this in Spark using reduceByKey, because averaging requires division, and division is neither a commutative nor associative operation
    - ((10 / 2) / 5) = 1, but (10 / (2/5)) = 25
  - To do this in Spark you could use groupByKey, and then mapValues to calculate the average of the inputs for each key

ReduceByKey (4 Of 4)

- The function passed to ReduceByKey must also be:
  - Commutative: x + y = y + x
  - Associative: (x + y) + z = x + (y + z)



Word Count Recap (Scala)

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

- Or
- These examples are identical

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

Why Do We Care About Counting Words?

- As simple as the problem of word counting appears, it typifies the characteristics of big data problems
  - Its hard to do in a typical RDBMS
    - And slow as well
  - Working with massive amounts of data
    - More than can be held in memory at once
    - More than can be processed in a timely manner on a single machine
- So, to surmount these problems we divide and conquer
  - Breaking data down into manageable chunks that can be processed
    - In isolation of one another
    - In parallel
  - Before aggregating the collective results of these parallel tasks into a final outcome
- Statistics functions are often aggregate functions
- Many common tasks are similar to Word Count
  - e.g. log analysis, were we count ip addresses, file types, dates and times, etc.

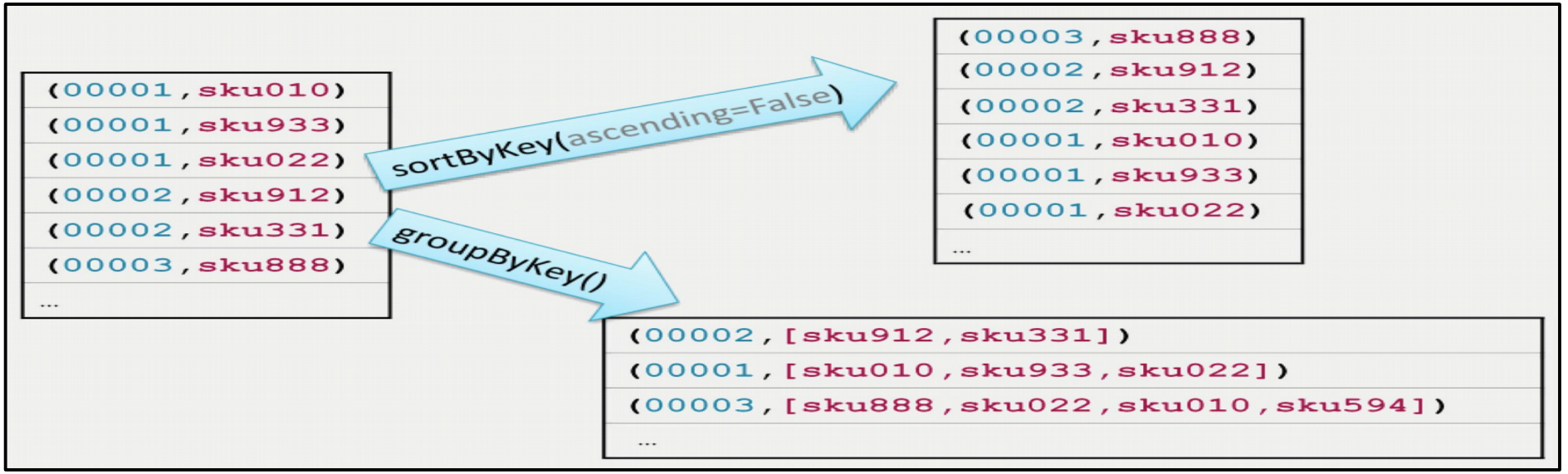
Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

Pair RDD Operations

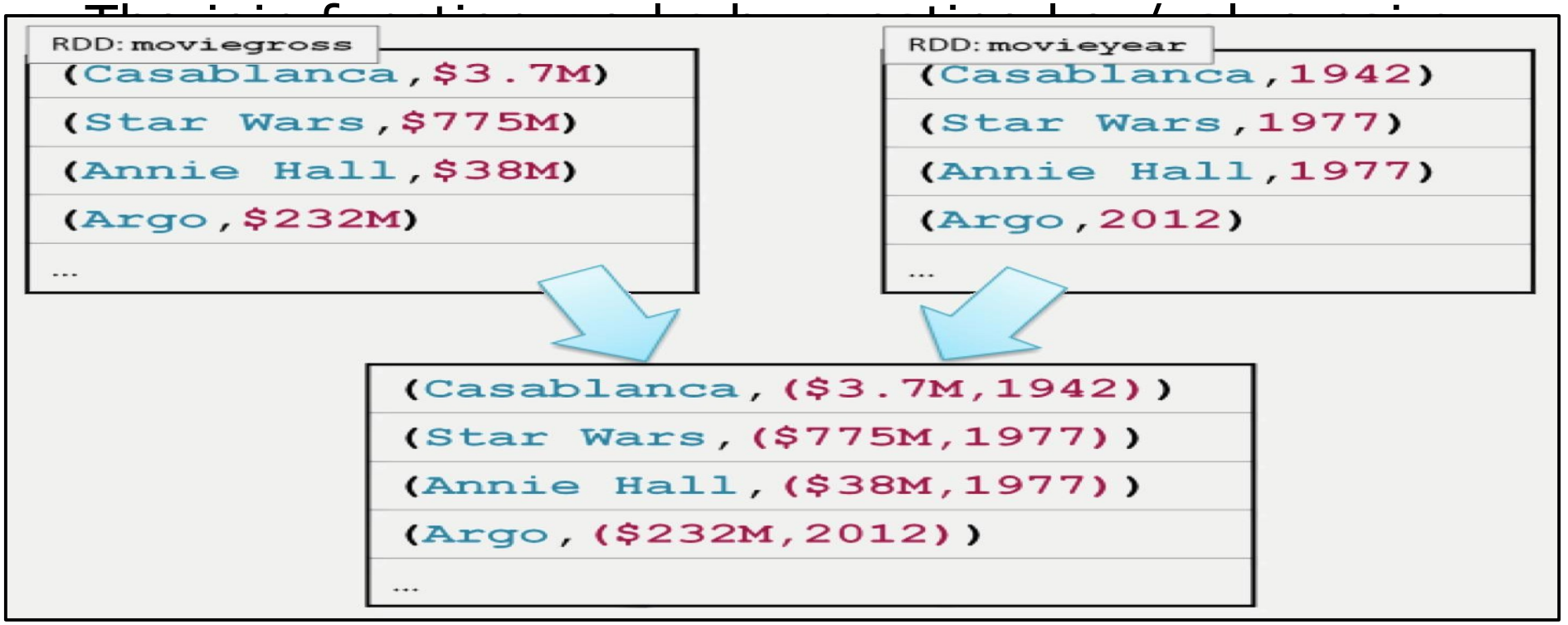
- In addition to map and reduce functions, Spark has several operations specific to Pair RDDs
  - countByKey
    - Counts the occurrences of each key in an RDD
  - groupByKey
    - Groups the values in an RDD by their key
  - sortByKey
    - Sorts the pairs in an RDD by their key in ascending or descending order
  - join
    - Returns an RDD containing all pairs with matching keys from two RDDs
    - Inner join by default, but outer joins are also supported
  - Most of these functions are just convenience functions correlating to map, flatMap or reduceByKey functions
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.PairRDDFunctions

Example: Pair RDD Operations



Example: Joining By Key

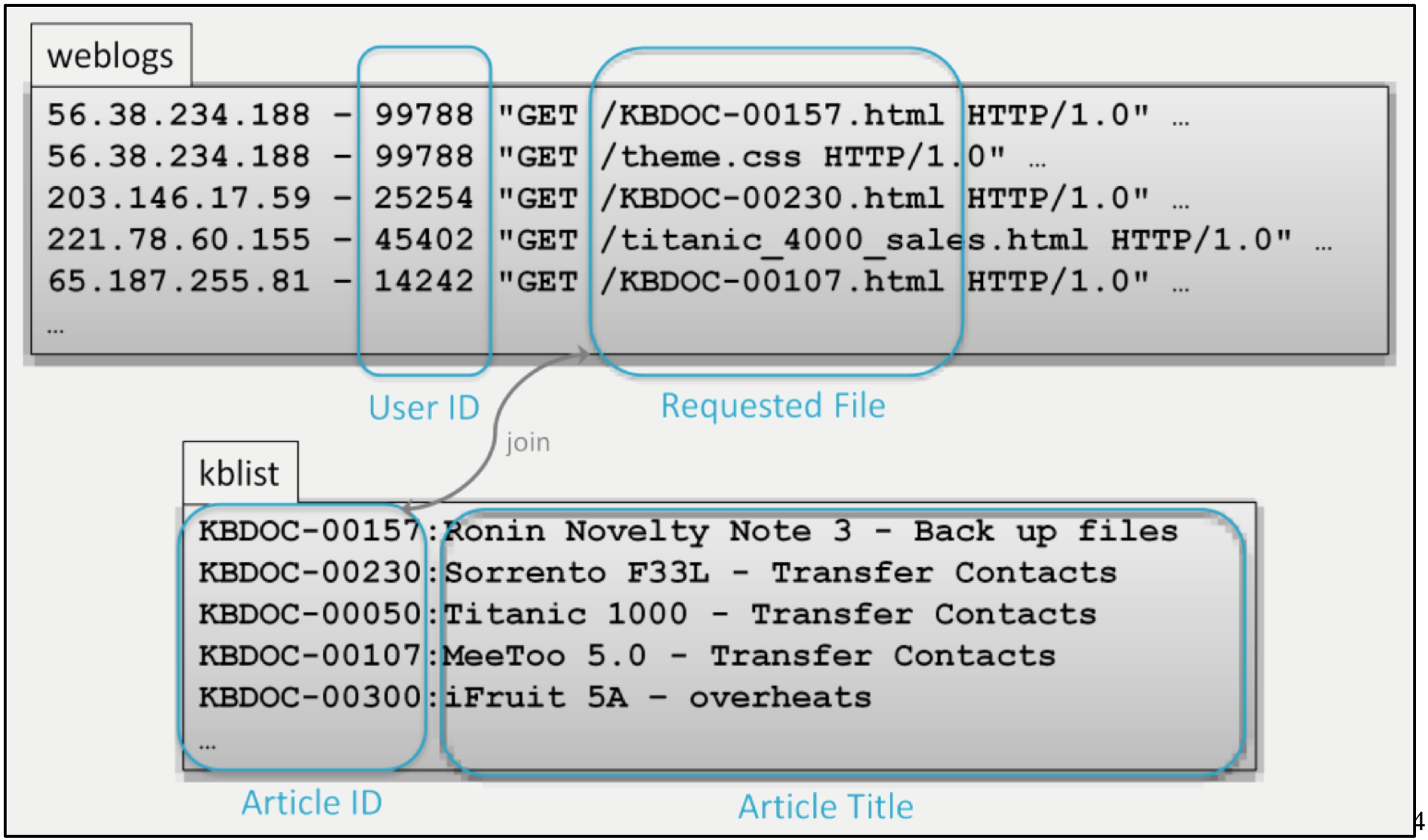
```
movies = moviegross.join(movieyear)
```



Using Join

- A common programming pattern
  - Map separate datasets into key/value Pair RDDs
  - Join by their common key
  - Map joined data into the desired format
  - Save, display, or continue processing
- This is actually an example of an even more typical pattern found in many Spark programs:
  - Parse/format data into key/value pairs
  - Aggregate the data (e.g. reduce, join, sort)
  - Parse/format the output to prepare it for the next job/stage or save/display the final values

Example: Join Web Log With Knowledge Base Articles (1 Of 3)



Example: Join Web Log With Knowledge Base Articles (2 Of 3)

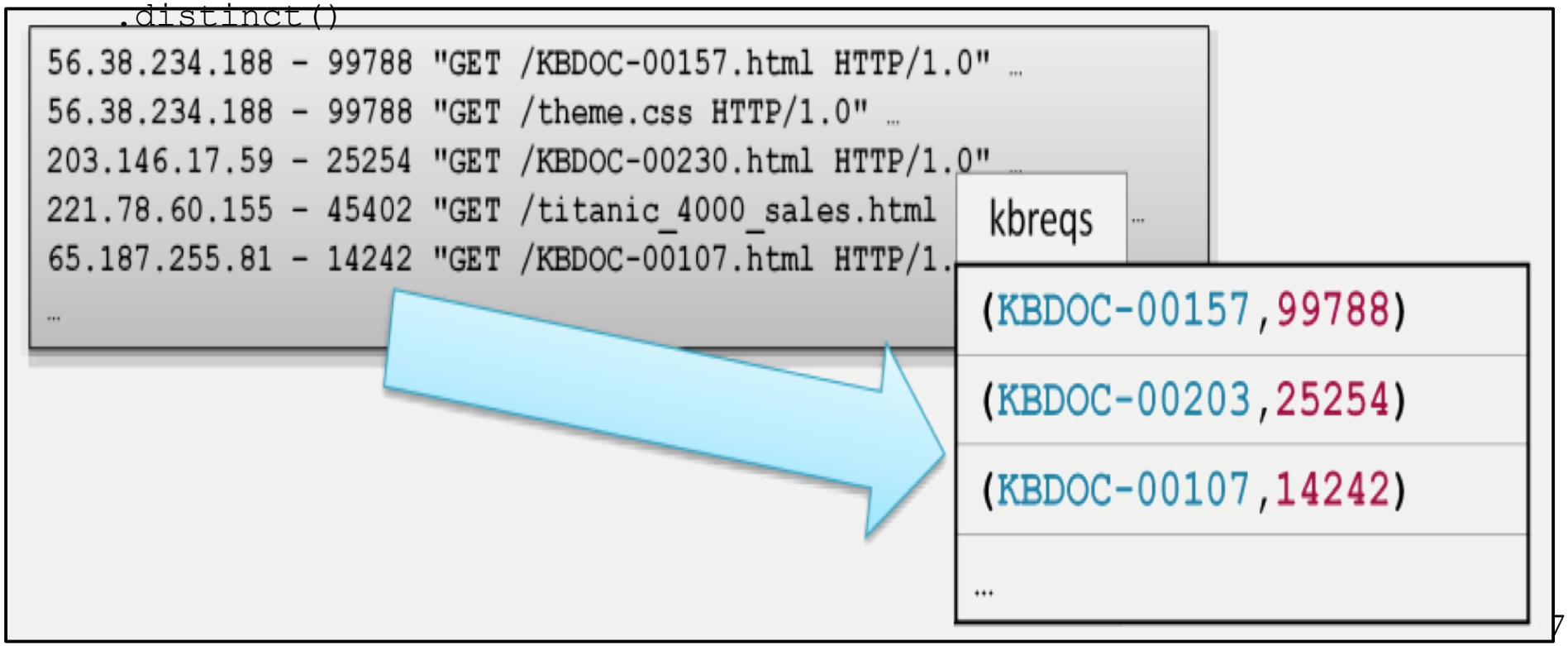
- In the previous slide you saw the contents of 2 files
  - A web server log, in which each line's:
    - Third field was the user ID of the user making the request
    - Fourth field was the actual request, from which we must extract the file name, which maps to the name of a Knowledge Base article, whose name follow the format
      - KBDOC—#####.html
  - An article index, in which each line's:
    - First field was the file name which maps to a knowledge base article ID
    - Second field was the article title
- From these two data sets we want to produce a list of article titles requested by each user

Example: Join Web Log With Knowledge Base Articles (3 Of 3)

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

Step 1A: Map Web Log Requests To (docid, userid) (1 Of 2)

```
> import re
> def getRequestDoc(s):        # Return the entire matching pattern
      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()
```



Step 1A: Map Web Log Requests To (docid, userid) (2 Of 2)

- Read the file
- Filter out lines that aren't requests for KBDocs
- Create a new pair RDD with the KB article ID as the key (using the regular expression function we just defined), and the user id as the value (the 3rd field in the line.)
- Filter out any duplicates, because for this particular analysis we don't care if one user viewed the same article multiple times

Step 1B: Map Web Log Requests To (docid, userid)

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

- Get the KB article list into pair form by splitting the line around the ":" delimiter, into its key and value

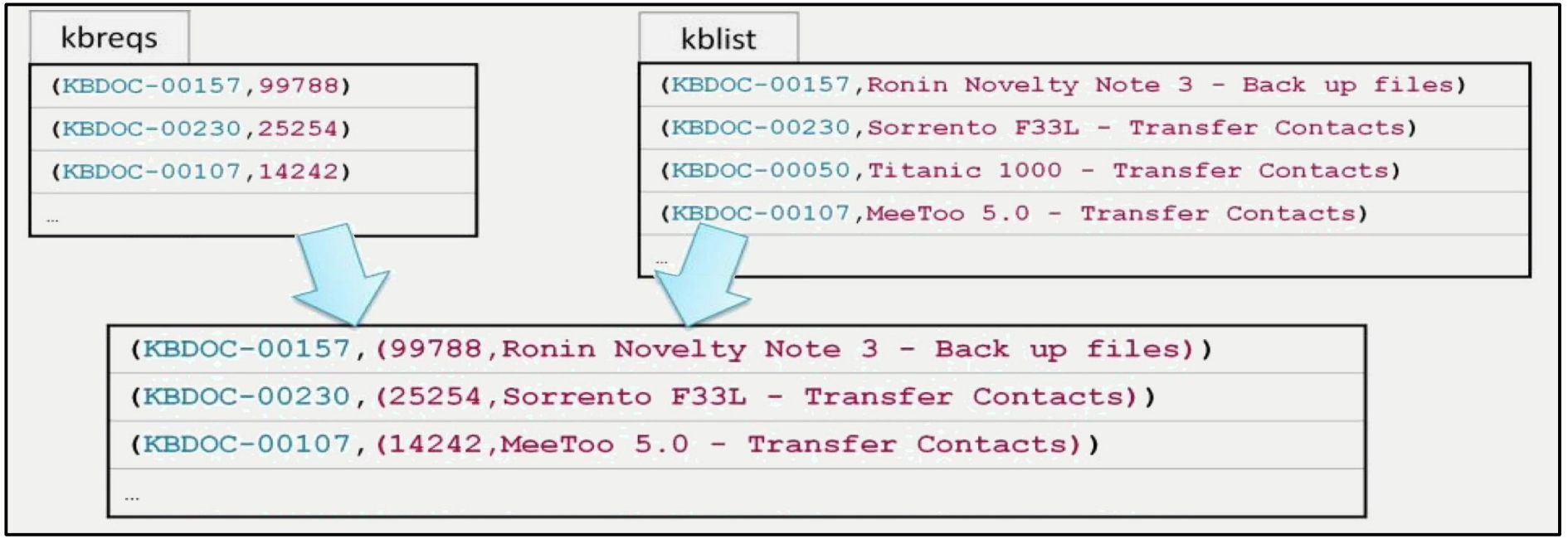
```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A - overheats
…
```

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 The Result To The Desired Format (userid, title)

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

- Remove the KB article ID (key) and re-key by the

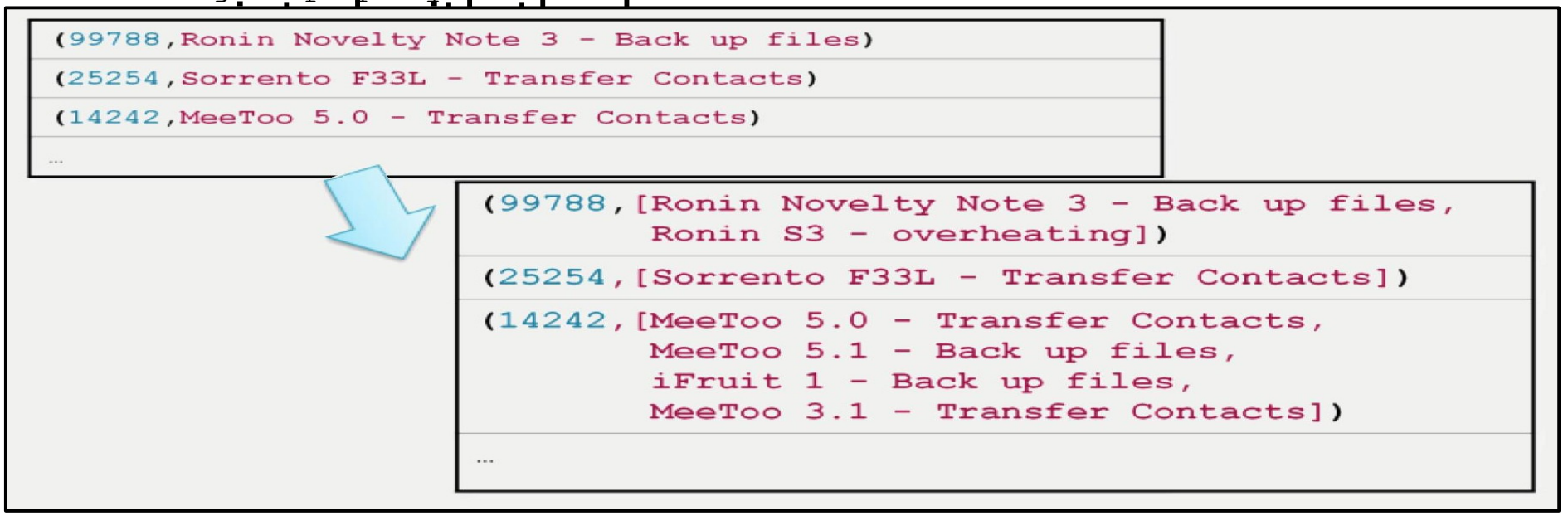
```
(KBDOC-00157,(99788,Ronin Novelty Note 3 - Back up files))
(KBDOC-00230,(25254,Sorrento F33L - Transfer Contacts))
(KBDOC-00107,(14242,MeeToo 5.0 - Transfer Contacts))
...
```

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

Step 4: Group Titles By User ID

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

- Note that the value is an iterable of all of the titles



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
...
```

- Note that what we have done here is quite similar to what data analysts do to prepare data for analysis
  - Take some raw data, then manipulate it to make it look right for subsequent processing

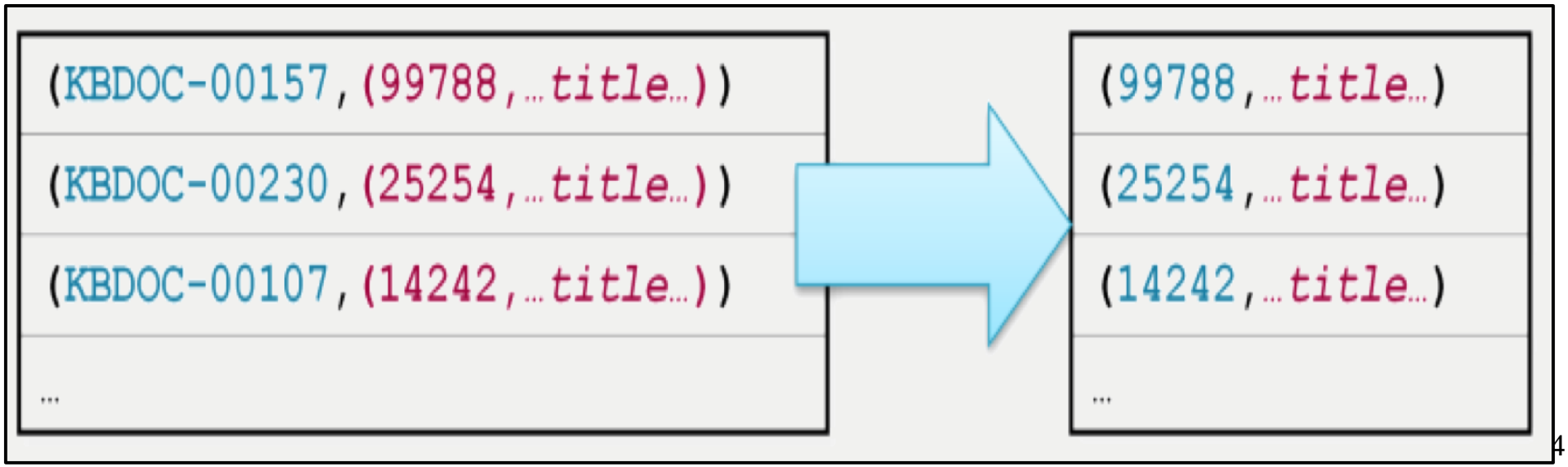
Aside: Anonymous Function Parameters (1 Of 2)

- Python and Scala pattern matching can help improve code readability

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

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

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



Aside: Anonymous Function Parameters (2 Of 2)

- Both Python and Scala support “pattern matching” for when you want to access some, but not all parameters to a function
  - Technically this is known as a ”Partial Function”, which means a function that is defined only for some parameters values, not all
  - By using a case statement for pattern matching, we are saying that this partial function is defined for input types that can be matched to the pattern:
    - A tuple (String, Tuple(String, String))
    - http://blog.bruchez.name/2011/10/scala-partial-functions-without-phd.html

Other Pair Operations

- keys
  - Returns an RDD of just the keys in an RDD, without the values
- values
  - Returns an RDD of just the values in an pair, without the keys
- lookup(key)
  - Returns the values for a given key as a list
  - Only supported in Scala
- leftOuterJoin, rightOuterJoin
  - Joins 2 pairs, including keys defined only in the left or right RDDs respectively
- mapValues, flatMapValues
  - Executes a function on just the values of each pair in an RDD, keeping the key the same
- See the PairRDDFunctions class in Scaladoc for a full list
  - The Scala documentation separates PairRDD functions from other RDD functions

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

Summary

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

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

Review

- What is the distinguishing characteristic of a Pair RDD?

Review Answer

- What is the distinguishing characteristic of a Pair RDD?
  - Each member is a Key/Value pair (tuple)

Review

- How does the output of the map function differ from that of the flatMap function?

Review Answer

- How does the output of the map function differ from that of the flatMap function?
  - There is a 1:1 relation between the input to the map function and its output
  - There is a 1:* relation between the input to the flatMap function and its output
    - In other words, flatMap decomposes or flattens its input into multiple outputs

Review

- What is MapReduce?
- How does Spark improve upon MapReduce?

Review Answer

- What is MapReduce?
  - MapReduce is a programming model applicable to distributed processing of large data sets
- How does Spark improve upon MapReduce?
  - Spark implements MapReduce with much greater flexibility than Hadoop
    - Map and Reduce functions can be interspersed
    - Results stored in memory
    - Operations can easily be chained (internally)

Review

- Given tab-delimited files named “movielens/movie/*” in our home directory of HDFS that looks like this ...

```
1    Casablanca   1942
2    Star Wars    1977
3    Annie Hall   1977
```

  ... newest movie to oldest movie, for movies released after 1930

```
from __future__ import print_function
sc.textFile("movielens/movie/*") \
    .___(lambda line: line.split('\t')) \
    .______(lambda fields: int(fields[2]) > 1930) \
    .___(lambda fields: (int(fields[2]), fields[1])) \
    .________(False) \
    .foreach(lambda (key, value): \
        print("(", key, ":", value, ")"))
```

Review Answer

- Given tab-delimited files named “movielens/movie/*” in our home directory of HDFS that looks like this ...

```
1    Casablanca   1942
2    Star Wars    1977
3    Annie Hall   1977
```

... newest movie to oldest movie, for movies released after 1930

```
from __future__ import print_function
sc.textFile("movielens/movie/*") \
    .map(lambda line: line.split('\t')) \
    .filter(lambda fields: int(fields[2]) > 1930) \
    .map(lambda fields: (int(fields[2]), fields[1])) \
    .sortByKey(False) \
    .foreach(lambda (key, value): \
        print("(", key, ":", value, ")"))
```

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

References

- None

Chapter Topics

- A Closer Look At RDDs
- Key/Value Pair RDDs
- MapReduce
- Other Pair RDD Operations
- Summary
- Review
- References
- Hands-On Exercise: Working With Pair RDDs

Hands-On Exercise

- Working with Pair RDDs
  - Continue exploring the web server log files using key/value Pair RDDs
  - Join log data with user account data
- Please refer to the Hands-On Exercise Manual