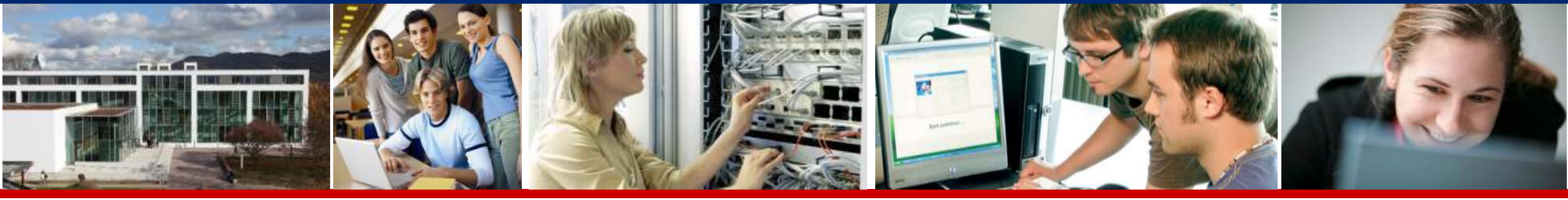


Apache Spark
Basic Operations on RDDs

Prof. Dr. Stephan Trahasch, based on Slides of Cloudera

Outline

- RDDs Basics
- Aggregating Data with Pair RDDs

RDDs

- RDDs can hold any type of element
  - Primitive types: integers, characters, booleans, etc.
  - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- Some types of RDDs have additional functionality
  - Pair RDDs: RDDs consisting of Key-Value pairs
    A few special operations are only available on RDDs of key-value pairs. The most common ones are distributed “shuffle” operations, such as grouping or aggregating the elements by a key.
  - Double RDDs: RDDs consisting of numeric data
    Extension contains many useful methods for aggregating numeric values. They become available if the data items of an RDD are implicitly convertible to the Scala data-type double

Creating RDDs From Collections

You can create RDDs from collections instead of files

`sc.parallelize(`*`collection`*`)`

```
> val myData = Array("Alice","Carlos","Frank","Barbara")

> val myRdd = sc.parallelize(myData)

> myRdd.take(2)
['Alice', 'Carlos']
```

Useful when

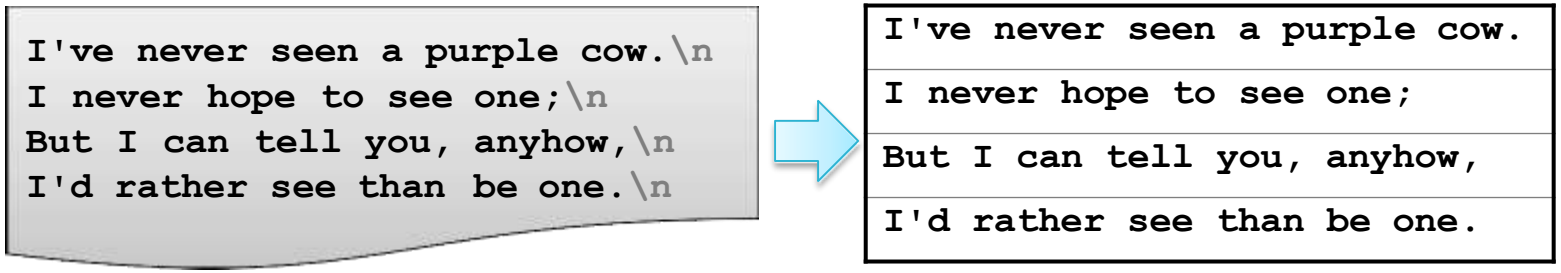
- Testing
- Generating data programmatically
- Integrating

Creating RDDs from Files (1)

- For file-based RDDs, use **`SparkContext.textFile`**
  - Accepts a single file, a wildcard list of files, or a comma-separated list of files
  - Examples
    - **`sc.textFile("myfile.txt")`**
    - **`sc.textFile("mydata/*.log")`**
    - **`sc.textFile("myfile1.txt,myfile2.txt")`**
  - Each line in the file(s) is a separate record in the RDD
- Files are referenced by absolute or relative URI
  - Absolute URI:
    - **`file:/home/training/myfile.txt`**
    - **`hdfs://localhost/loudacre/myfile.txt`**
  - Relative URI (uses default file system): **`myfile.txt`**

Creating RDDs from Files (2)

`textFile` maps each line in a file to a separate RDD element



`textFile` only works with line-delimited text files.

Input and Output Formats (1)

Spark uses Hadoop InputFormat and OutputFormat Java classes

Some examples from core Hadoop

- TextInputFormat / TextOutputFormat – newline delimited text files
- SequenceInputFormat / SequenceOutputFormat
- FixedLengthInputFormat

Many implementations available in additional libraries
e.g. AvroInputFormat / AvroOutputFormat in the Avro library

Input and Output Formats (2)

Specify any input format using `sc.hadoopFile` or `newAPIhadoopFile` for New API classes

Specify any output format using `rdd.saveAsHadoopFile` or `saveAsNewAPIhadoopFile` for New API classes

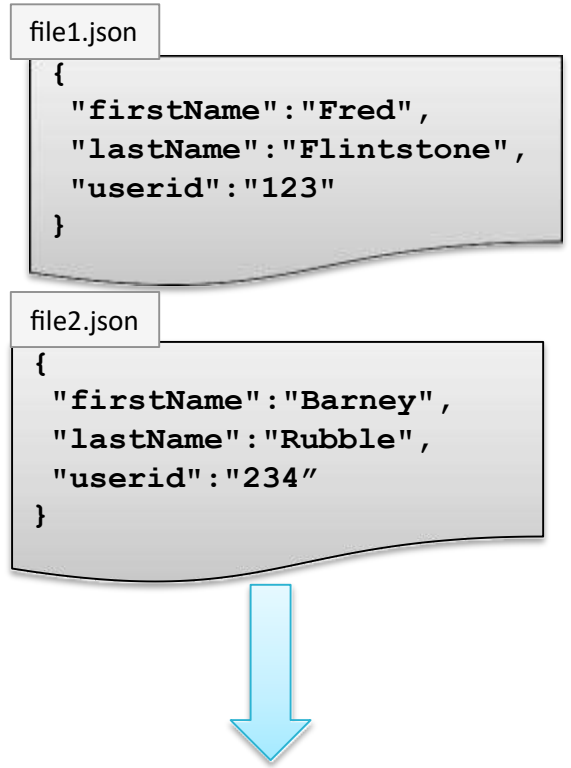
`textFile` and `saveAsTextFile` are convenience functions

`textFile` just calls `hadoopFile` specifying `TextInputFormat`

`saveAsTextFile` calls `saveAsHadoopFile` specifying `TextOutputFormat`

Whole File-Based RDDs (1)

- **`sc.textFile` maps each line in a file to a separate RDD element**
  - What about files with a multi-line input format, e.g. XML or JSON?
- **`sc.wholeTextFiles(`*`directory`*`)`**
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)



(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123"})
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":"234"})
(file3.xml,…)
(file4.xml,…)

Whole File-Based RDDs (2)

```
> import json
> myrdd1 = sc.wholeTextFiles(mydir)
> myrdd2 = myrdd1
  .map(lambda (fname,s): json.loads(s))
> for record in myrdd2.take(2):
>     print record["firstName"]
```

Output:

```
Fred
Barney
```

```
> import scala.util.parsing.json.JSON
> val myrdd1 = sc.wholeTextFiles(mydir)
> val myrdd2 = myrdd1.map(pair => JSON.parseFull(pair._2).get.
                asInstanceOf[Map[String,String]])
> for (record <- myrdd2.take(2))
    println(record.getOrElse("firstName",null))
```

Some Other General RDD Operations

Single-RDD Transformations

- **`flatMap`** – maps one element in the base RDD to multiple elements
- **`distinct`** – filter out duplicates and returns a new dataset that contains the distinct elements of the source dataset.
- **`sortBy`** – use provided function to sort

multi-RDD Transformations

- **`intersection`** – create a new RDD with all elements in both original RDDs
- **`union`** – add all elements of two RDDs into a single new RDD
- **`zip`** - pair each element of the first RDD with the corresponding element of the second
- **`coalesce(numPartitions)`** - Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.

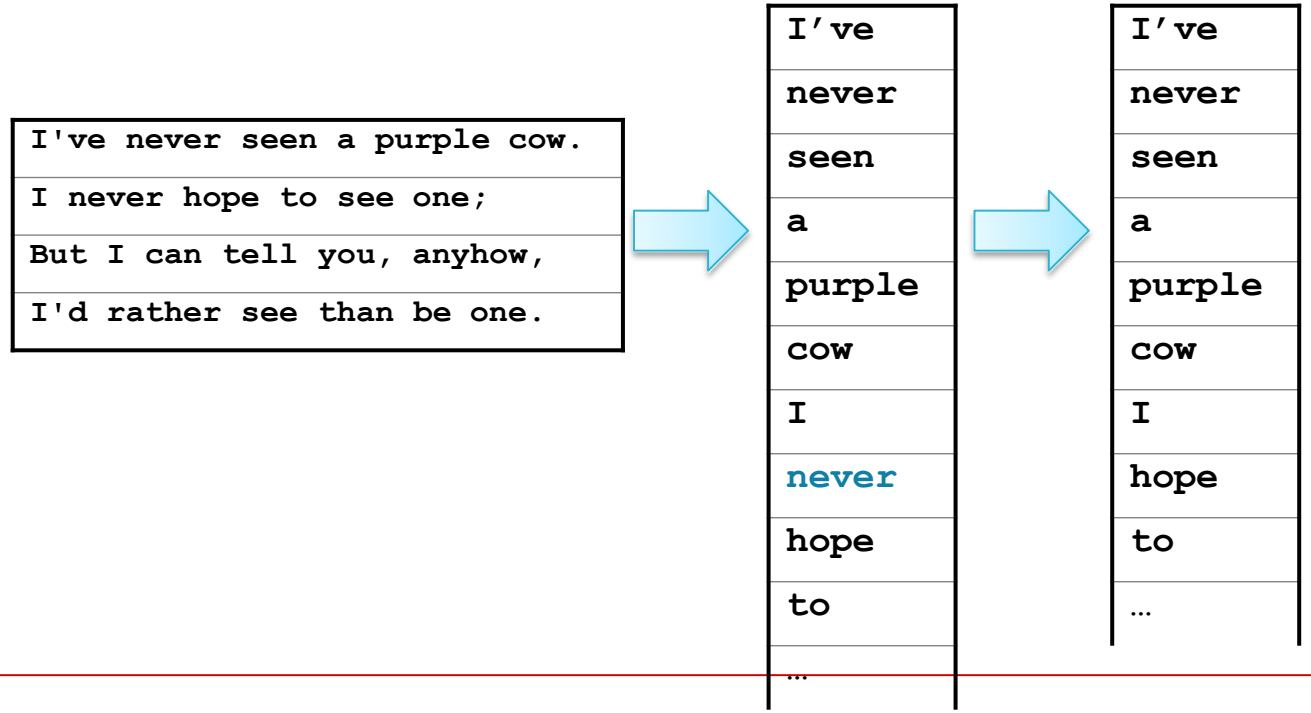
Example: `flatMap` and `distinct`

Python

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

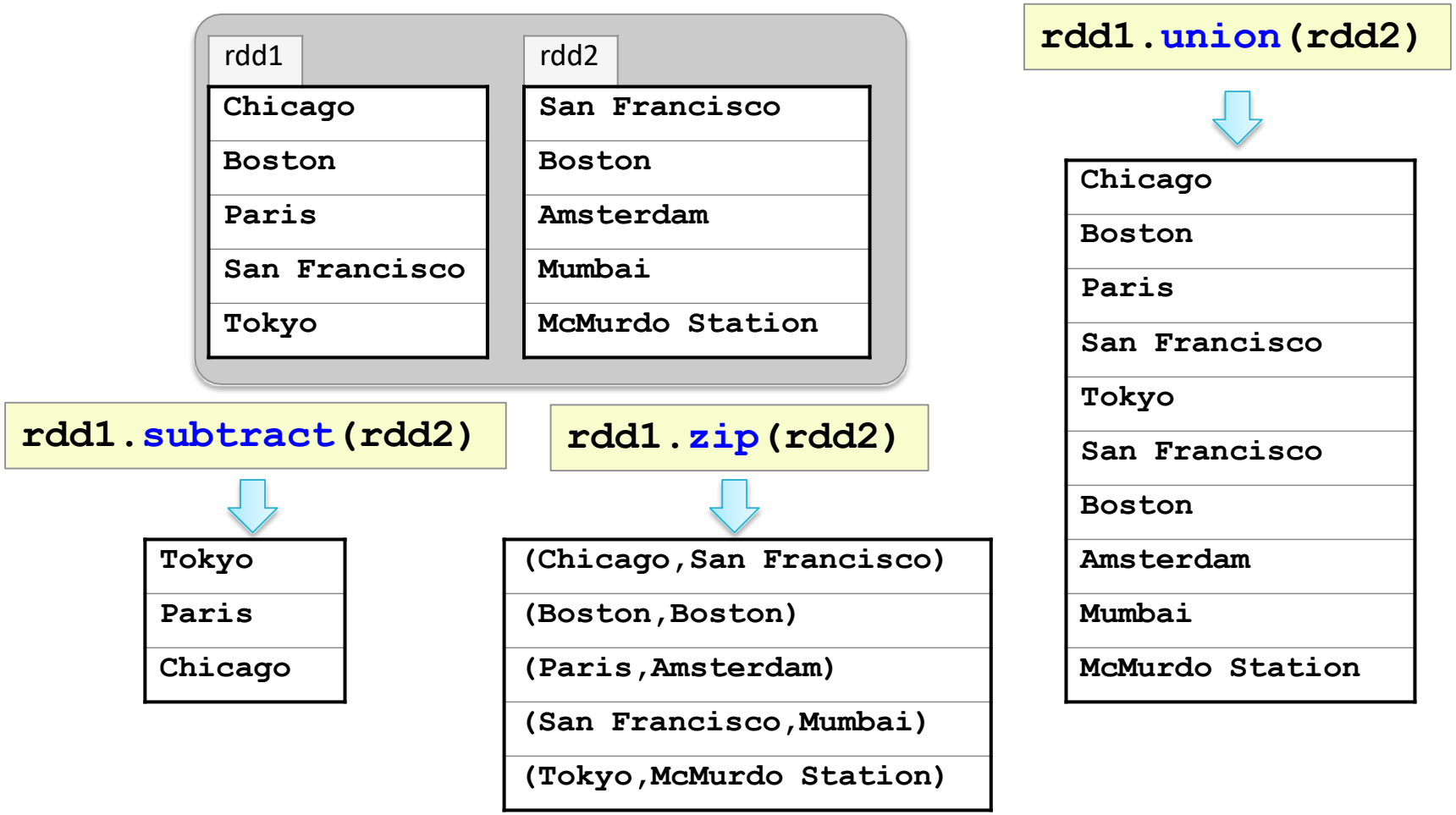
Scala

```
> sc.textFile(file).
   flatMap(line => line.split(' ')).
   distinct()
```



Function passed to flatMap must return a collection (e.g an array, iterator, etc.). Each the returned collection is mapped to a single element in the result RDD.

Examples: Multi-RDD Transformations



Some Other General RDD Operations

- Other RDD operations
  - `first` – return the first element of the RDD
  - `foreach` – apply a function to each element in an RDD
  - `top(`*n*`)` – return the largest *n* elements using natural ordering
- Sampling operations
  - `sample` – create a new RDD with a sampling of elements
  - `takeSample` – return an array of sampled elements
- Double RDD operations
  - Statistical functions, e.g., `mean`, `sum`, `variance`, `stdev`

RDDs – Essential Points

- RDDs can be created from files, parallelized data in memory, or other RDDs
- `sc.textFile` reads newline delimited text, one line per RDD record
- `sc.wholeTextFile` reads entire files into single RDD records
- Generic RDDs can consist of any type of data
- Generic RDDs provide a wide range of transformation operations

Overview

- RDDs Basics
- Aggregating Data 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

Pair RDD

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

- 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

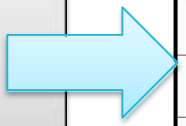
Python

```
> 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   Fred Flintstone
User090   Bugs Bunny
user111   Harry 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" …
```

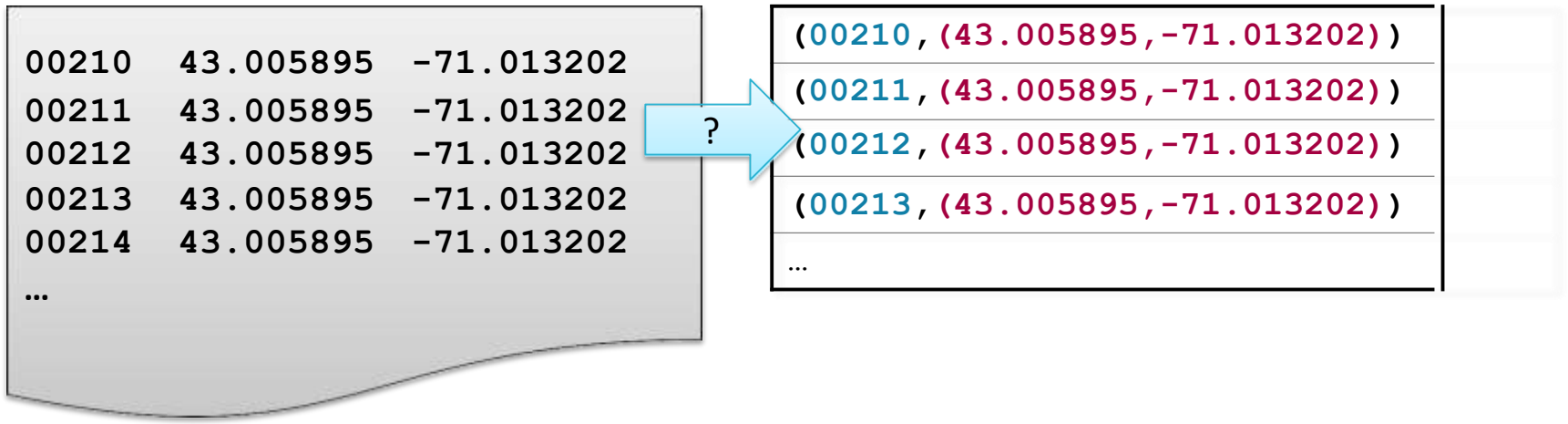
…

```
(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…)
…
```

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 tuple

Question 1: Pairs With Complex Values

- Input: a list of postal codes with latitude and longitude
- Output: postal code (key) and lat/long pair (value)

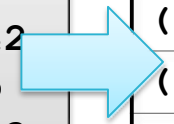


Answer 1: Pairs With Complex Values

```
> 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))))
```

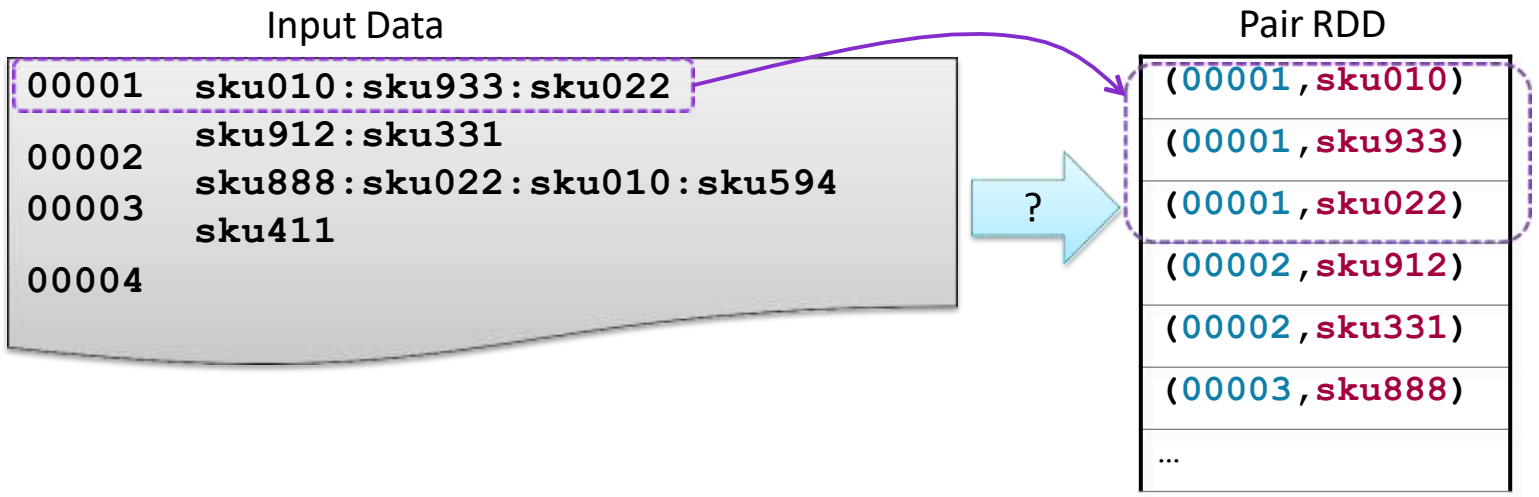
```
00210   43.005895  -71.013202
01014   42.170731  -72.604842
01062   42.324232  -72.67915
01263   42.3929    -73.228483
…
```



```
(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)

- 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))
```

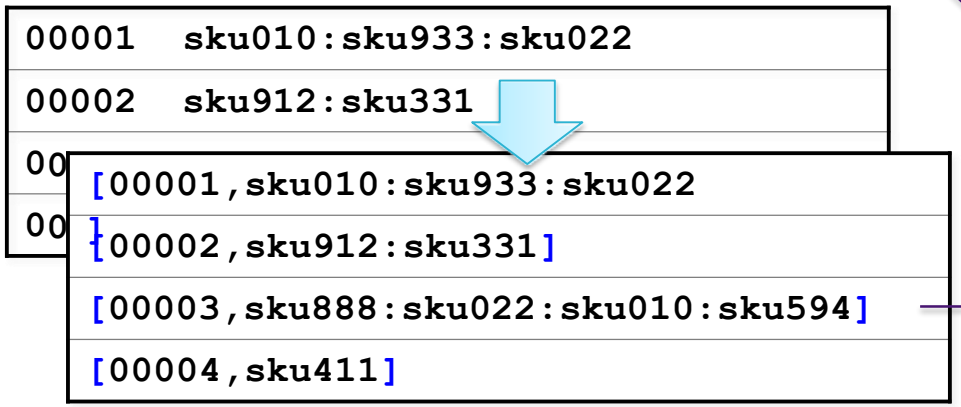
Answer 2: Mapping Single Rows to Multiple Pairs (1)

```
> 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)

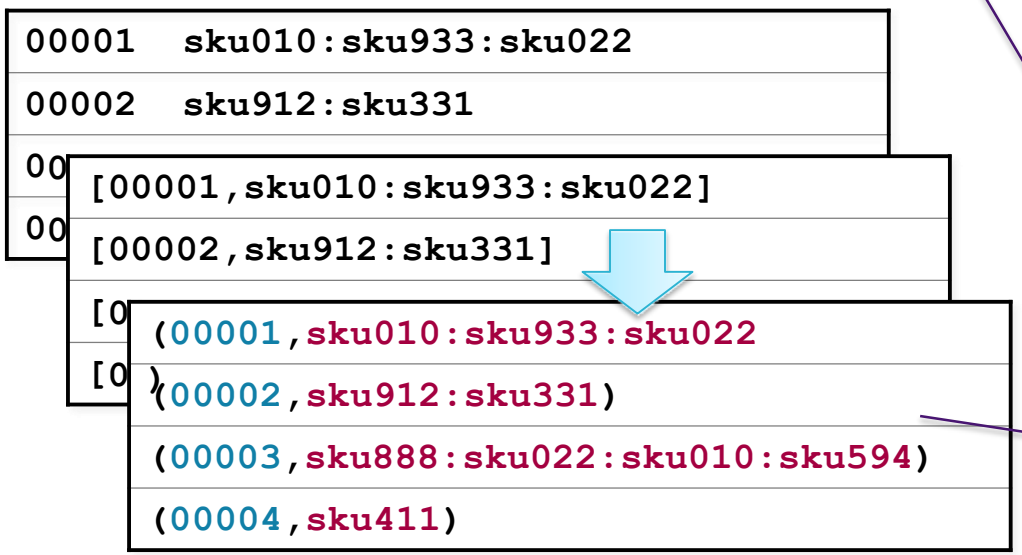
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t'))
```



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

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

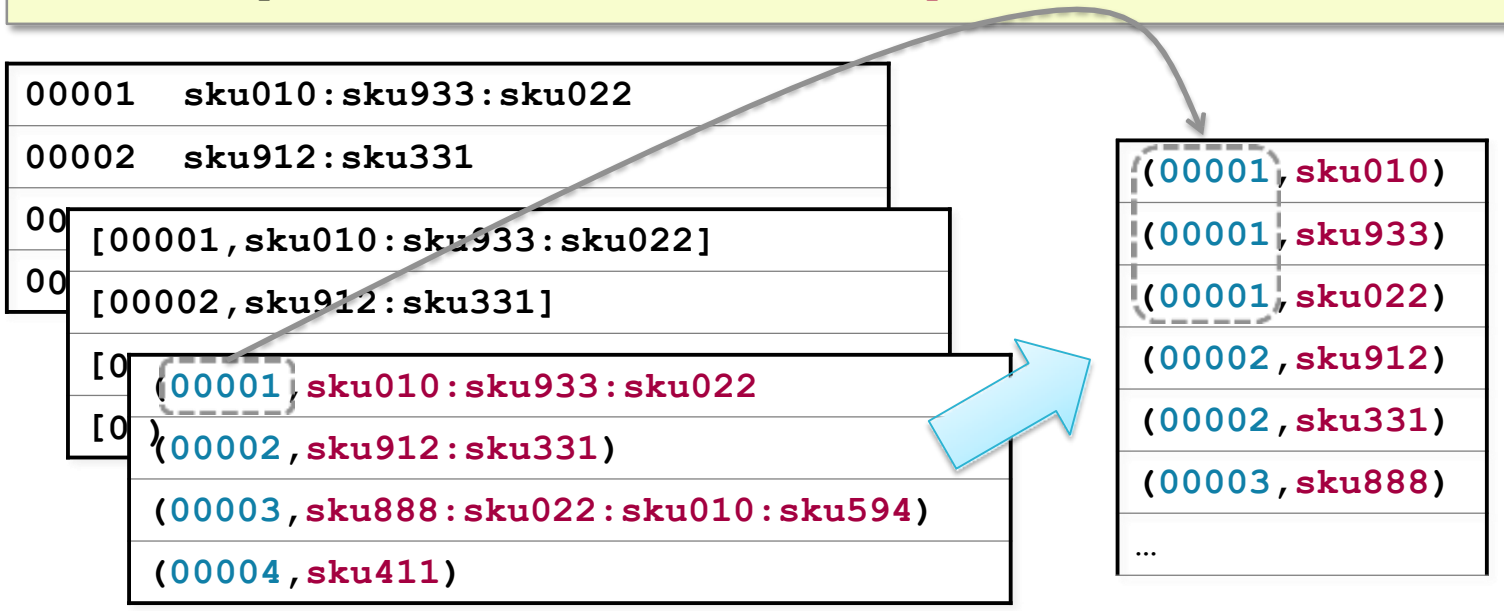
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
```



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(':'))
```



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
  - 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

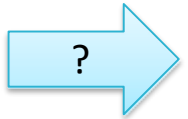
MapReduce in Spark

- 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
    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

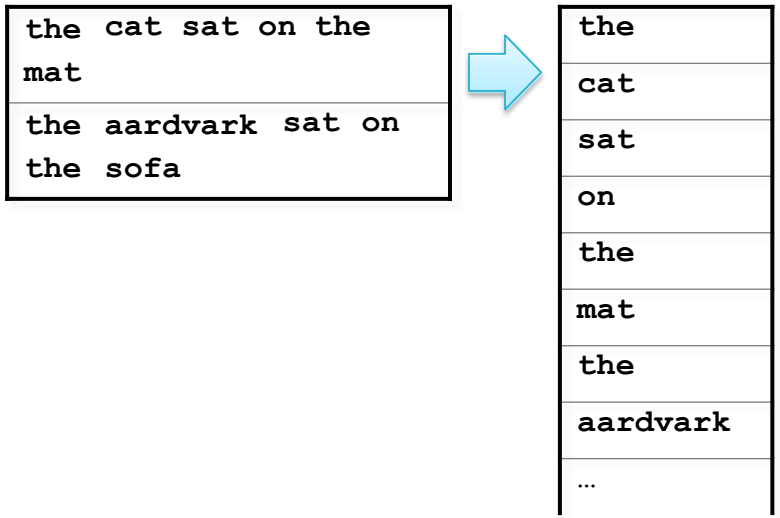
Example: Word Count (1)

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

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

Example: Word Count (2)

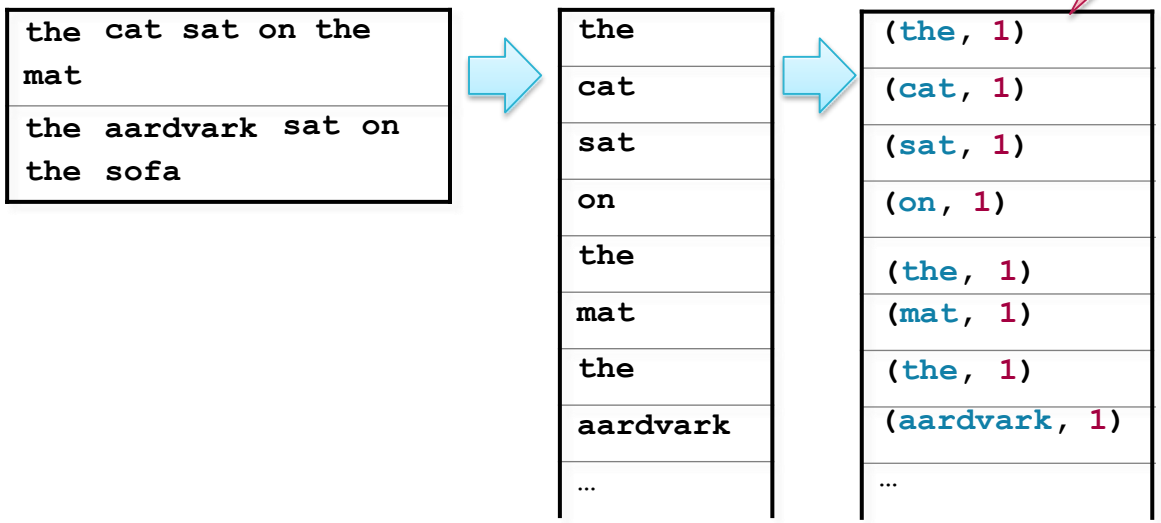
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



Example: Word Count (3)

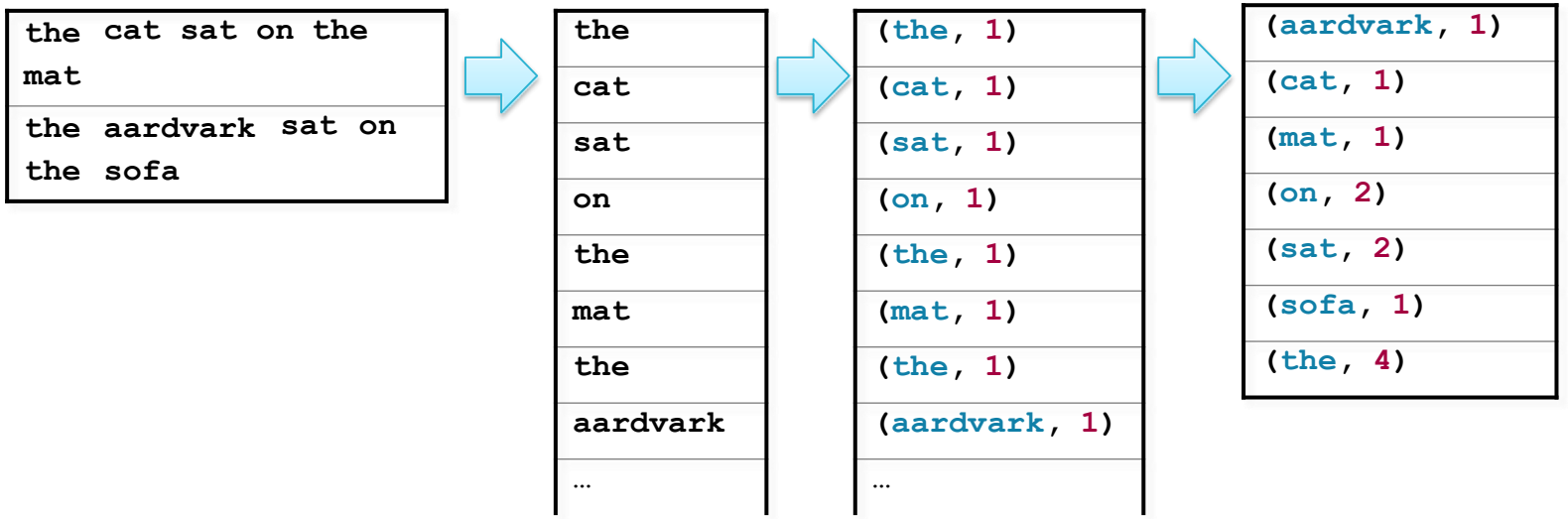
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Example: Word Count (4)

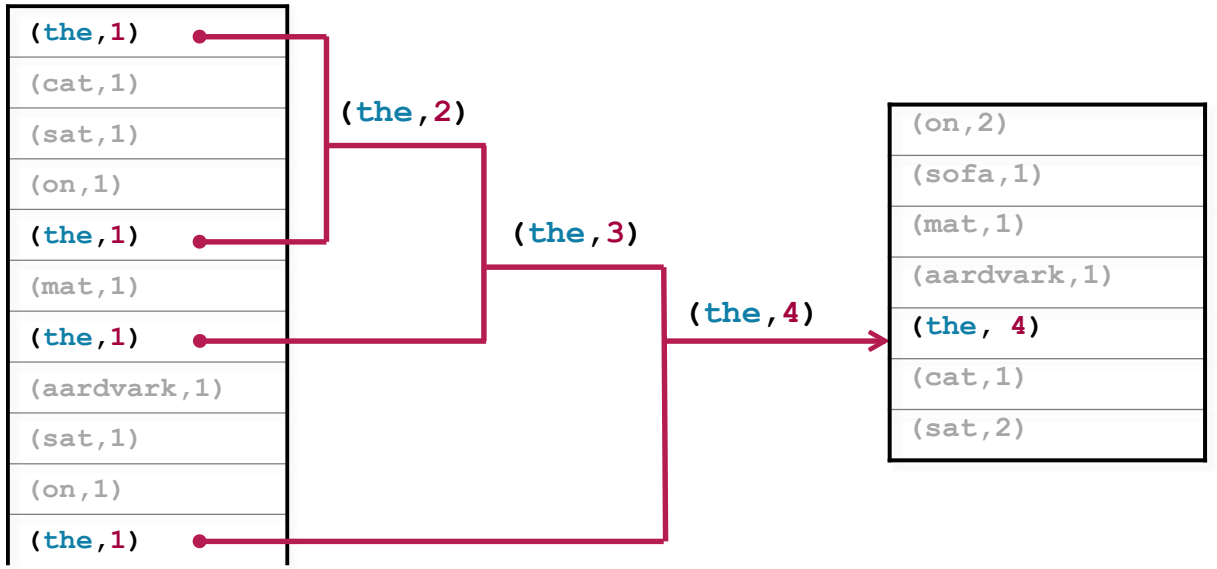
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



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)
```



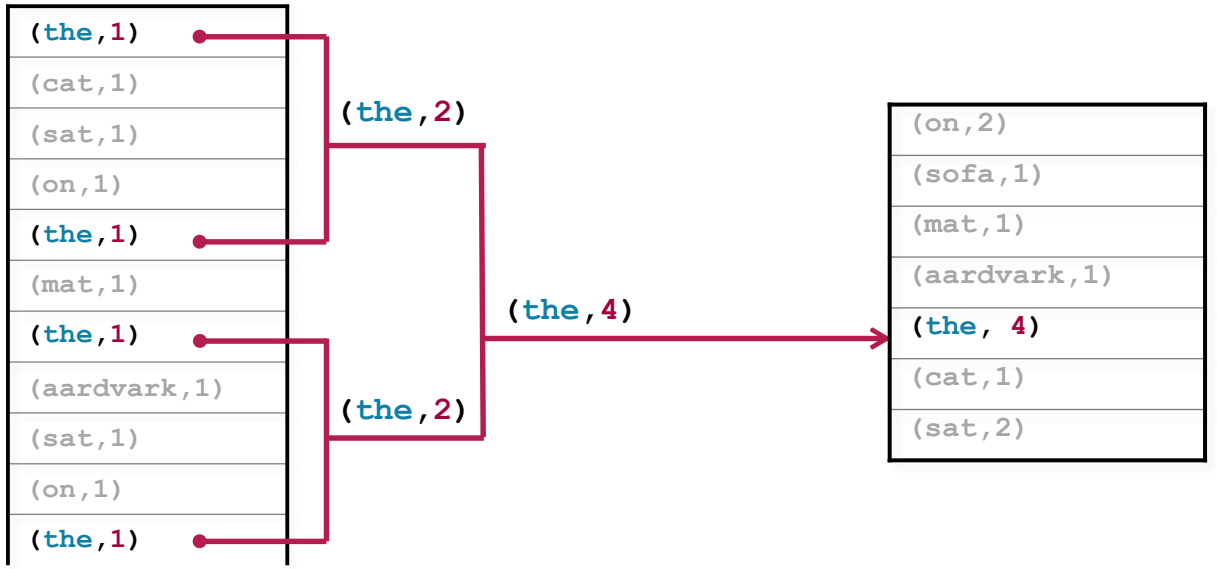
ReduceByKey (2)

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

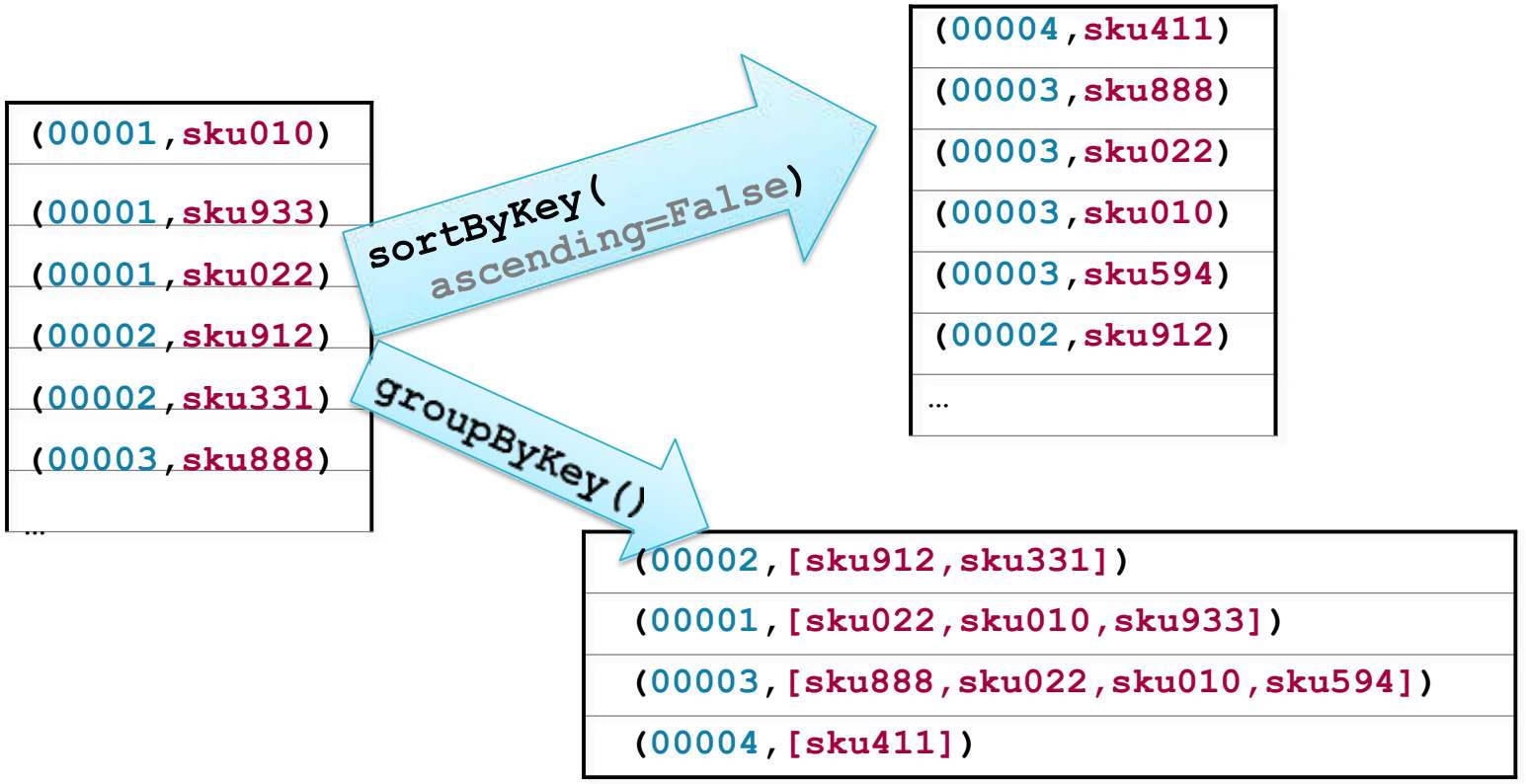
Other Pair RDD Operations

In addition to map and reduce functions, Spark has several operations specific to Pair RDDs

Examples

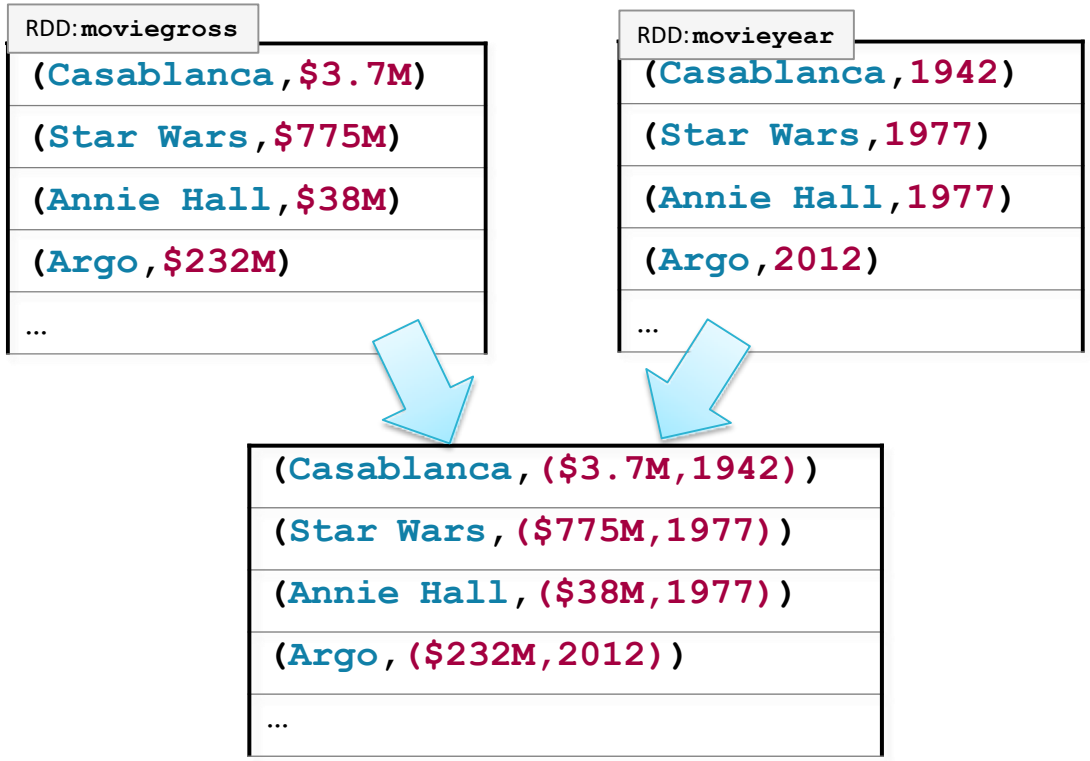
- countByKey – return a map with the count of occurrences of each key
- groupByKey – group all the values for each key in an RDD
- sortByKey – sort in ascending or descending order
- join – return an RDD containing all pairs with matching keys from two RDDs

Example: Pair RDD Operations



Example: Joining by Key

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

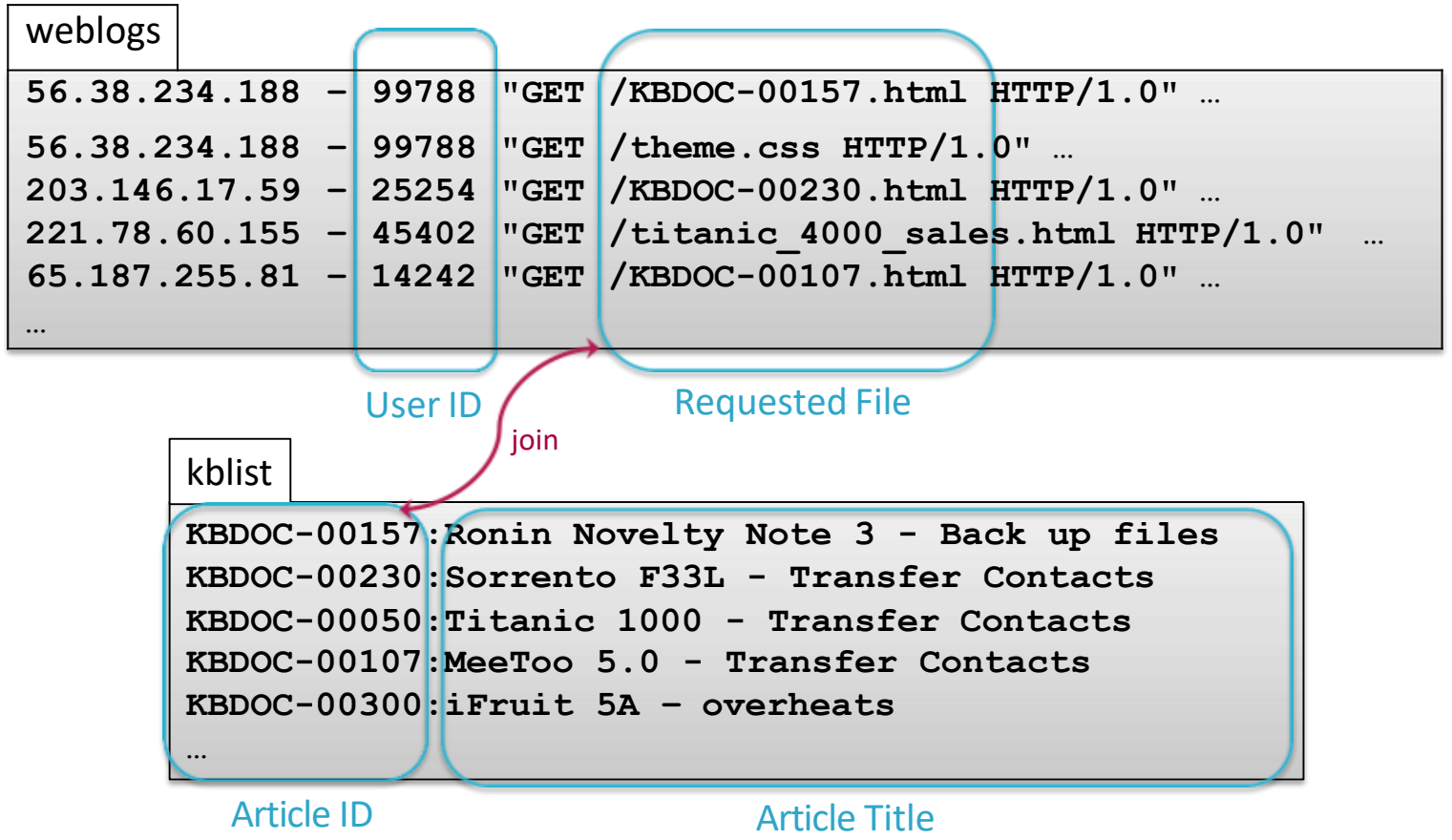


Using Join

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)

- 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 UserID

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! P/1.0" …
65.187.255.81 - 14242 "GET /KBDOC-00107.html HT
….
```

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 upfiles
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A – overheats
```

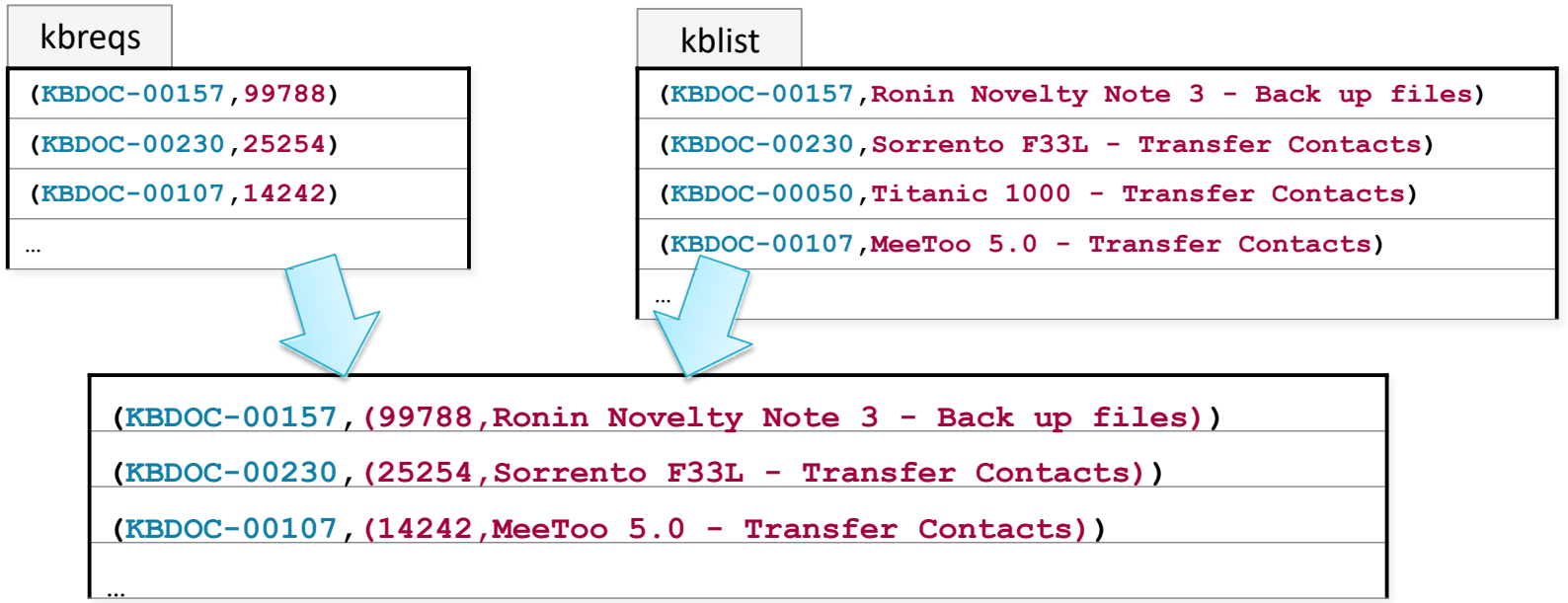
…

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 - Backup 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])
...
```

Aside: Anonymous Function Parameters

Python and Scala pattern matching can help improve code readability

Python

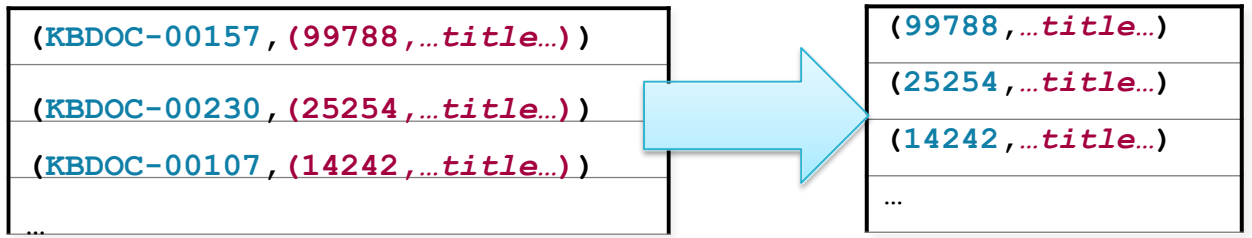
```
> map(lambda (docid,(userid,title)): (userid,title))
```

Scala

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

OR

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



Other Pair Operations

- 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