University of Salford MANCHESTER

SCHOOL OF SCIENCE, ENGINEERING & ENVIRONMENT

Program: **MSc of Data Science**

Module: **Big Data Tools and Techniques**

Week 3 - Part 2

Resilient Distributed Datasets (**RDDs**)

2025

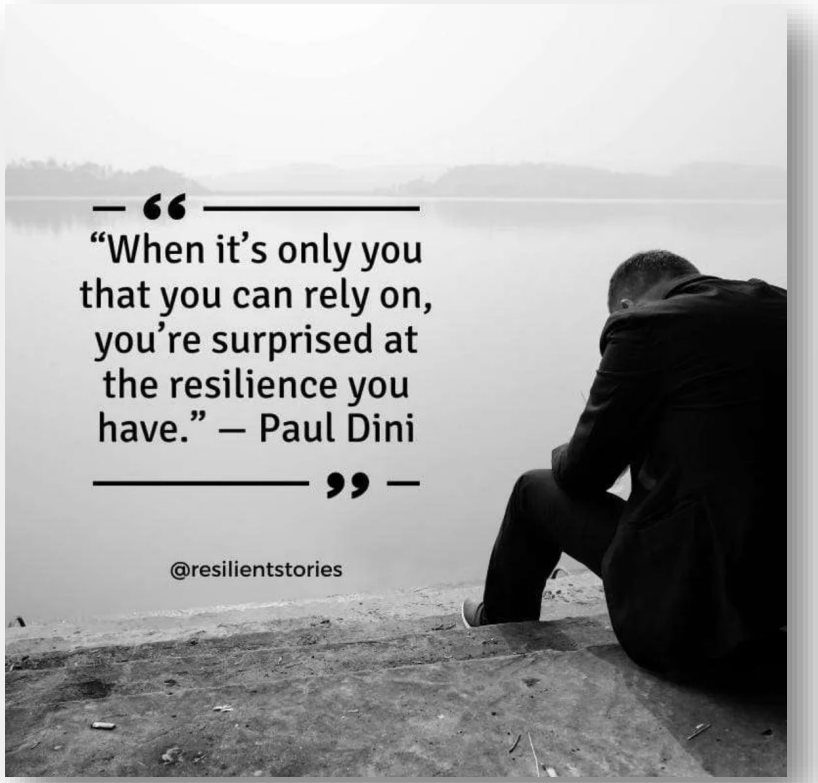
Learning Outcomes

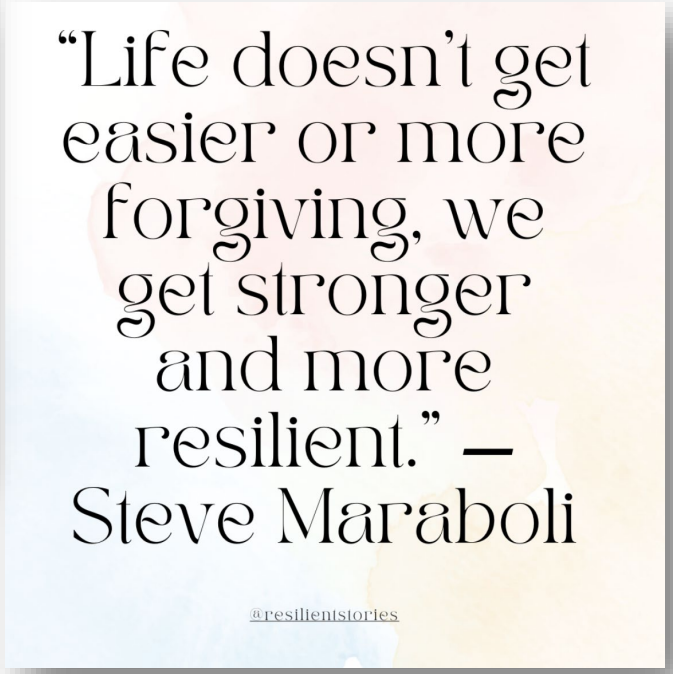
1. How to Create an RDD
2. RDD Operations
3. RDD Types
4. RDD & Lazy Execution
5. Functional programming in Spark
6. Pair RDDs

Refresher: What is RDD?

- RDD (Resilient Distributed Dataset)
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – processed across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- RDDs are the fundamental unit of data in Spark
- Most Spark programming consists of performing operations on RDDs

Resilient!!


“When it’s only you that you can rely on, you’re surprised at the resilience you have.” — Paul Dini
@resilientstories

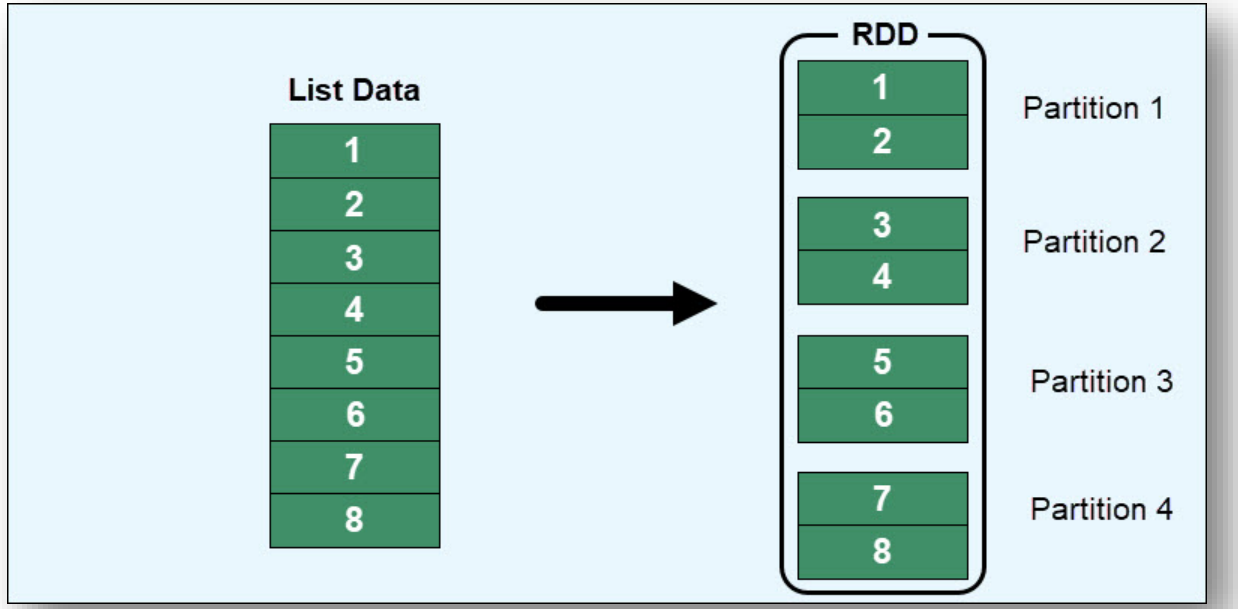

“Life doesn’t get easier or more forgiving, we get stronger and more resilient.” — Steve Maraboli
@resilientstories


The New York Times
RESILIENCE
What Makes Some People More Resilient Than Others
The very earliest days of our lives, and our closest relationships, can offer clues about how we cope with adversity.
Share full article
Monika Aichele

Refresher: RDDs on a cluster

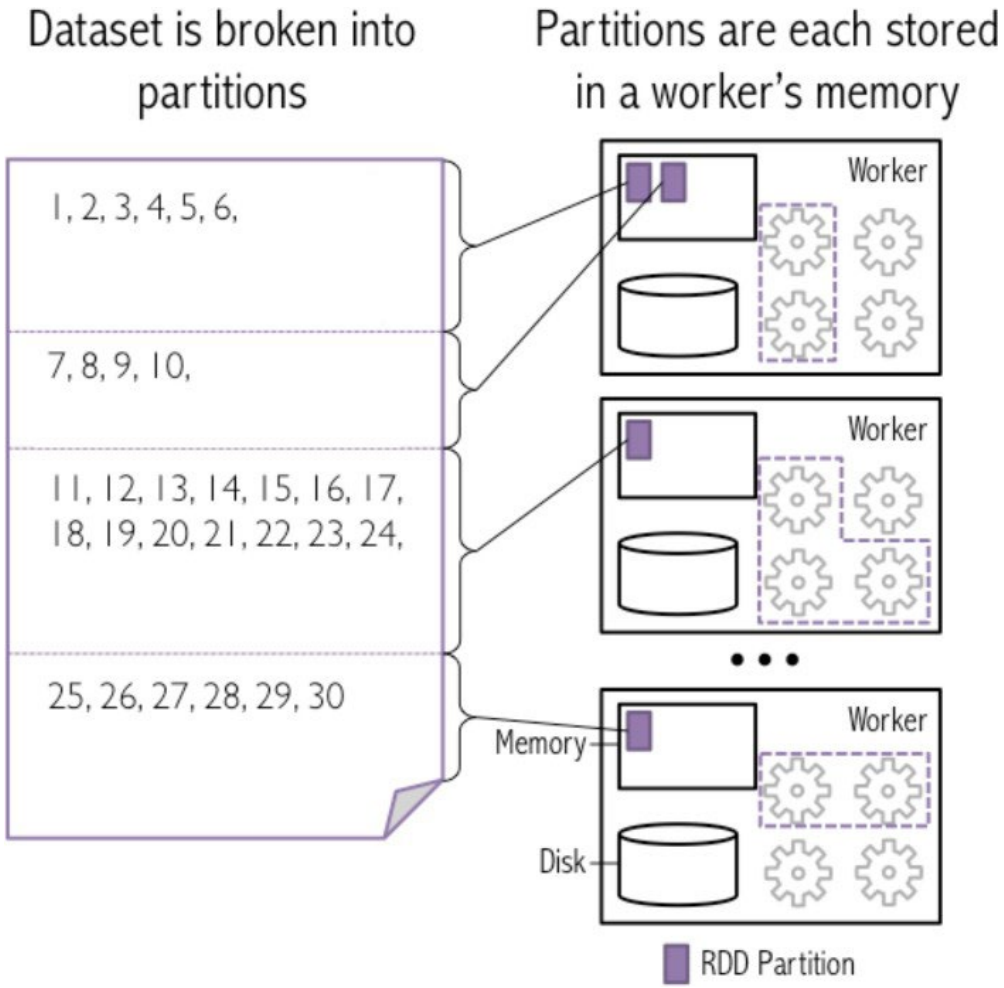
Resilient Distributed Datasets (RDD)

- Data is partitioned in worker nodes



Refresher: RDDs on a cluster

- Resilient Distributed Datasets
  - Data is partitioned across worker nodes (e.g. Data is from 1 to 30)
- Partitioning is done automatically by Spark
  - Optionally, you can control how many partitions are created



How to Create an RDD

Three ways to create an RDD

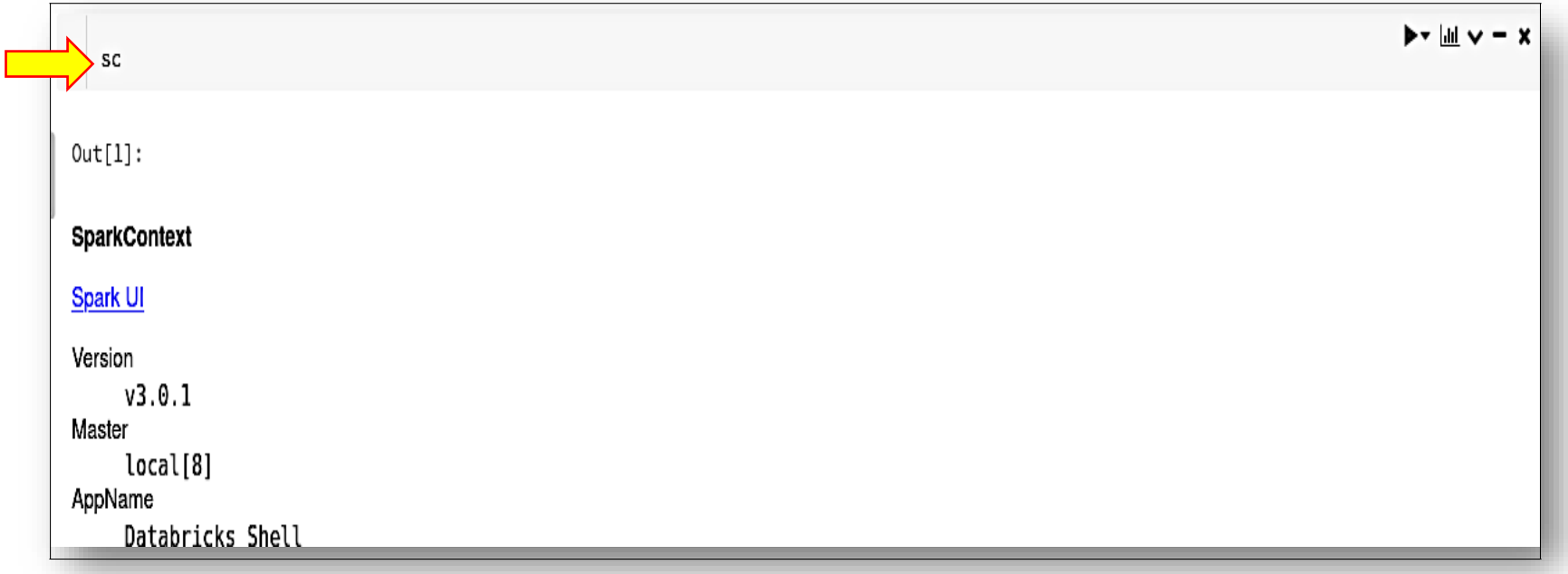
- From a file or set of files
- From data in memory

(In general, when we say "data in memory," it means that the data is stored and accessed directly from a computer's **RAM (Random Access Memory)** rather than from a storage device like a hard drive or SSD)

- From another RDD

Spark Context (SC)

- Every Spark application requires a Spark Context
- The main entry point to the Spark API



Example: A file based RDD

```
mydata = sc.textFile("purplecow.txt")
mydata.count()
```

File: purplecow.txt

I've never seen a purple cow.
I hope I never see one;
But I can tell you anyhow,
I'd rather see than be one.

RDD: mydata

I've never seen a purple cow.
I hope I never see one;
But I can tell you anyhow,
I'd rather see than be one.

SC: Spark Context

```
mydata = sc.textFile("purplecow.txt")
mydata.count()
```

textFile is a method of SparkContext class that reads a text file and return it as an RDD of Strings.

File: purplecow.txt

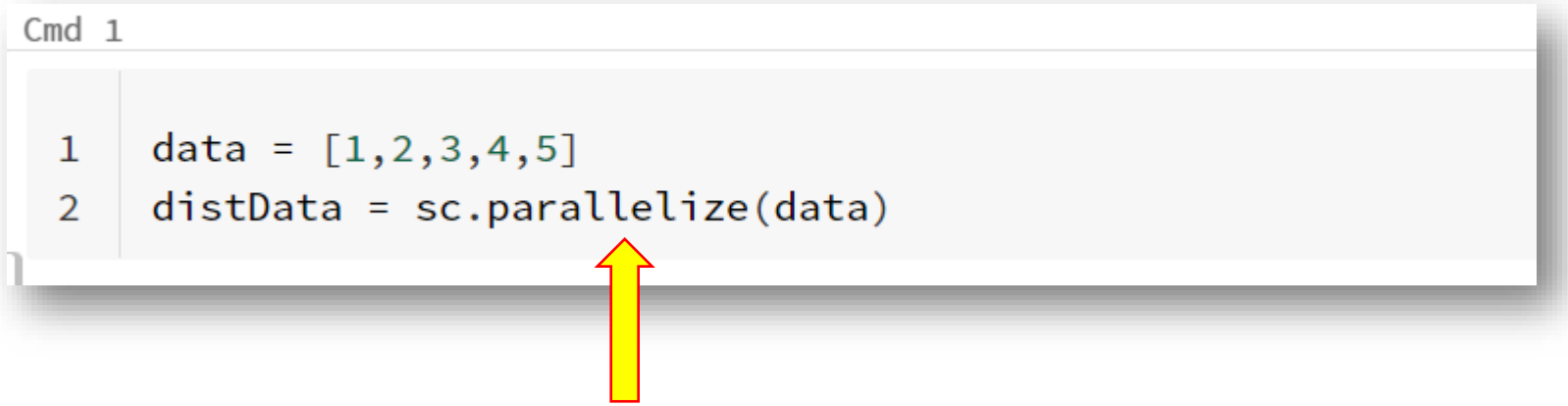
I've never seen a purple cow.
I hope I never see one;
But I can tell you anyhow,
I'd rather see than be one.

⇓

RDD: mydata

I've never seen a purple cow.
I hope I never see one;
But I can tell you anyhow,
I'd rather see than be one.

Example: RDD from data in memory

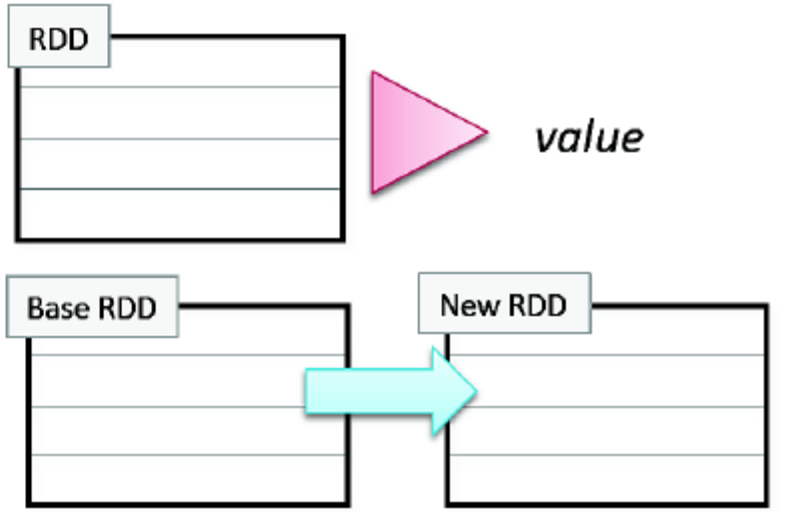


Parallelize is a method of SparkContext (SC) to create an RDD from an existing collection (e.g an array) which is already in the driver node.

RDD Operations

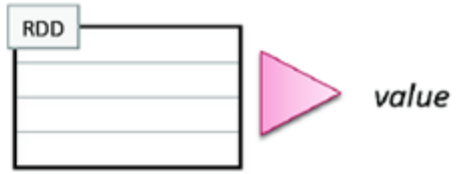
RDD operations

- Two types of RDD operations
  - **Actions** – return values
  - **Transformations** – define a new RDD based on the current one(s)



RDD operations: Actions

- Some common actions
  - `count()` – return the number of elements
  - `take(`*n*`)` – return an array of the first *n* elements
  - `collect()` – return an array of all elements
  - `saveAsTextFile(`*file*`)` – save to text file(s)



RDD: mydata

I've never seen a purple cow.
I hope I never see one;
But I can tell you anyhow,
I'd rather see than be one.

```
1  mydata = sc.textFile("dbfs:/FileStore/tables/purplecow.txt")
2  print(mydata.count())
3  for line in mydata.take(2):
4    print(line)
```

▸ (2) Spark Jobs

```
4
I've never seen a purple cow.
I hope I never see one.
```

Another example

```
1  data = [1,2,3,4,5]
2  distData = sc.parallelize(data)
```

```
Command took 0.05 seconds
```

Cmd 2

```
1  distData.count()
```

```
▸ (1) Spark Jobs
Out[11]: 5
Command took 0.51 seconds
```

Cmd 3

```
1  distData.collect()
```

```
▸ (1) Spark Jobs
Out[10]: [1, 2, 3, 4, 5]
Command took 0.24 seconds
```

Cmd 4

We can create RDDs from collections instead of files

Cmd 6

```
1  myData = ["BDTT","ADB","ASDV","MLDM"]
2
3  myRdd = sc.parallelize(myData)
4
5  myRdd.take(2)
```

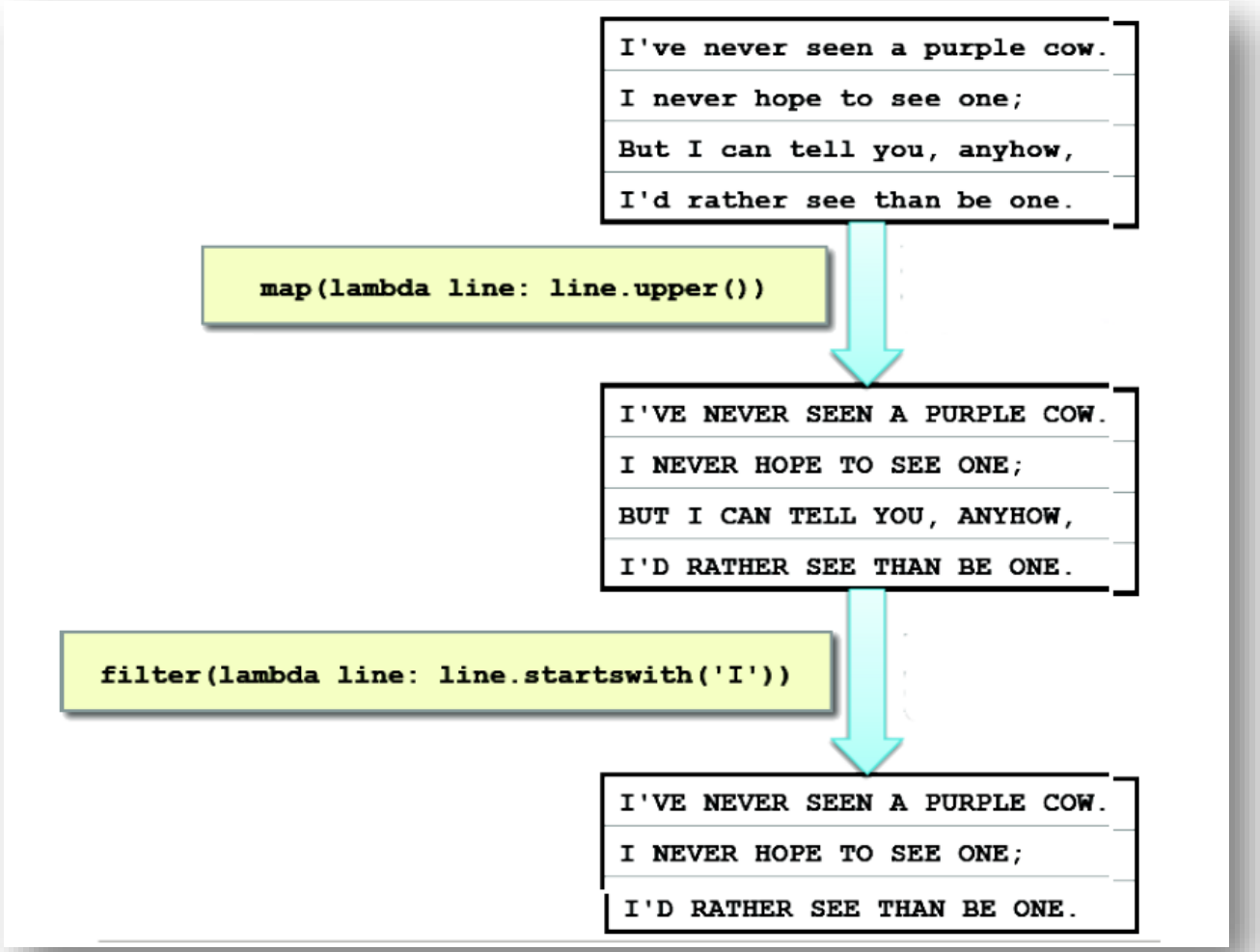
▸ (2) Spark Jobs

```
Out[19]: ['BDTT', 'ADB']
```

RDD operations: Transformation

- Transformations create a new RDD from an existing one
- RDDs are immutable
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- Some common transformations
  - **map**(*function*) – creates a new RDD by performing a function on each record in the base RDD
  - **filter**(*function*) – creates a new RDD by including or excluding each record in the base RDD according to a boolean function

Example: map and filter actions



RDD Types

RDD Types

- 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)
- Pair RDDs
  - RDDs consisting of Key-Value pairs

For a file based RDD use *sc.TextFile*

```
sc.textFile("myfile.txt")
sc.textFile("mydata/*.log")
sc.textFile("myfile1.txt,myfile2.txt")
```

Files are referenced by absolute or relative URL

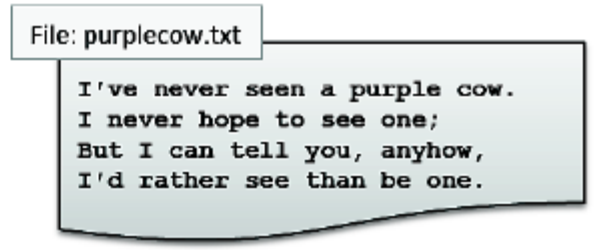
- Absolute URI:
  - `file:/home/training/myfile.txt`
  - `hdfs://localhost/loudacre/myfile.txt`
  - `s3://bucket/folder/filename.csv` (AWS S3)
  - `wasb://bucket/folder/filename.csv` (Azure WASBs)
  - `gs://bucket/folder/filename.csv` (Google Cloud Storage)
  - `dbfs://folder/filename.csv` (Databricks DBFS)
- Relative URI (uses default file system): `myfile.txt`

RDD & Lazy Execution

RDD has a Lazy execution

1

Data in RDDs is not processed until an *action* is performed

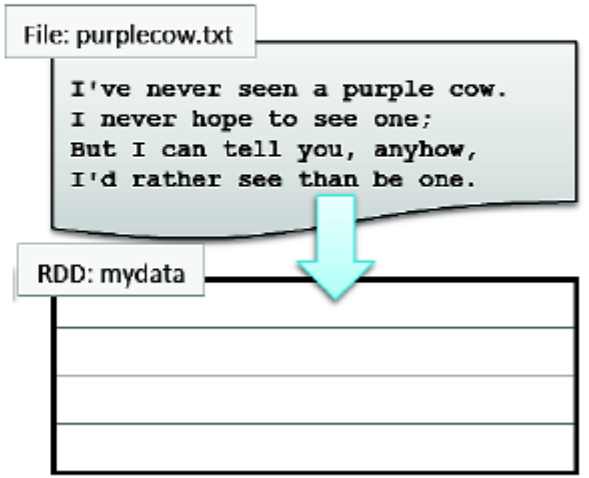


Lazy execution

2

Data in RDDs is not processed until an *action* is performed

```
mydata = sc.textFile("purplecow.txt")
```

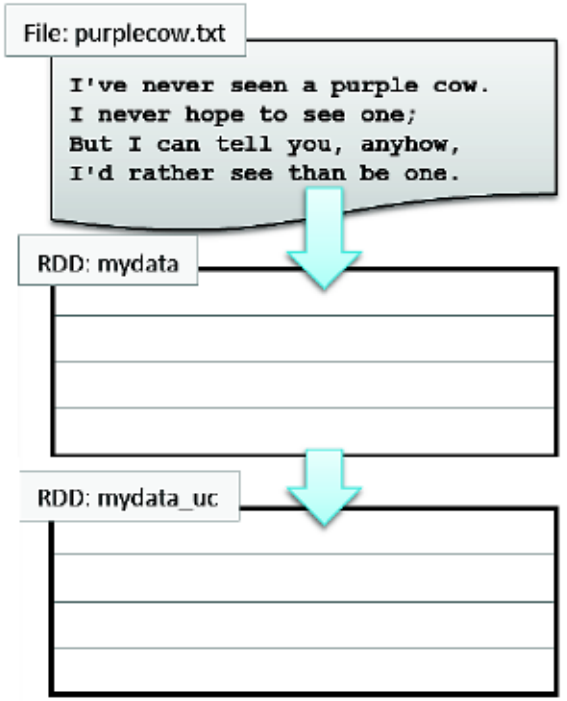


Lazy execution

3

Data in RDDs is not processed until an *action* is performed

```
mydata = sc.textFile("purplecow.txt")
mydata_uc = mydata.map(lambda line:
  line.upper())
```

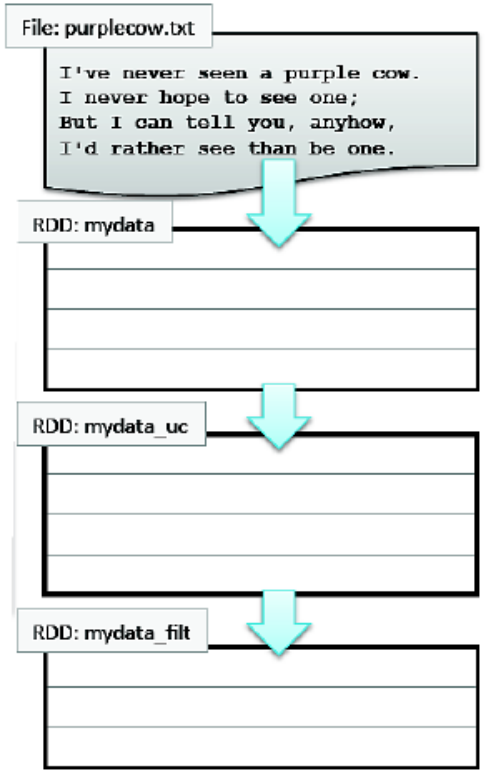


Lazy execution

4

Data in RDDs is not processed until an *action* is performed

```
mydata = sc.textFile("purplecow.txt")
mydata_uc = mydata.map(lambda line:line.upper())
mydata_filt = mydata_uc.filter(lambda
  line: line.startswith("I"))
```

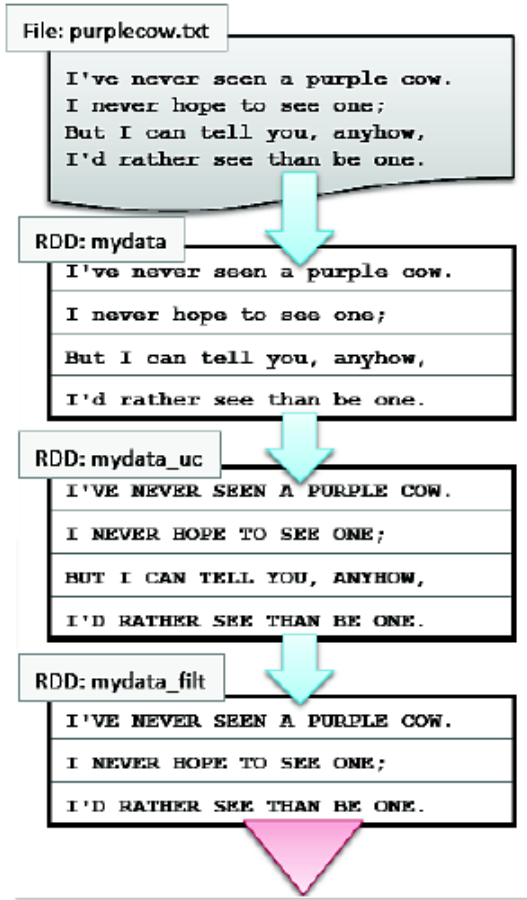


Lazy execution

Data in RDDs is not processed until an *action* is performed

```
mydata = sc.textFile("purplecow.txt")
mydata_uc = mydata.map(lambda line:line.upper())
mydata_filt = mydata_uc.filter(lambda
  line:  line.startswith("I"))
mydata_filt.count()
3
```

count() is a action



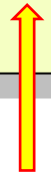
Chaining transformation

Transformations may be chained together

```
mydata = sc.textFile("purplecow.txt")
mydata_uc = mydata.map(lambda s: s.upper())
mydata_filt = mydata_uc.filter(lambda s: s.startswith('I'))
mydata_filt.count()
3
```

is exactly equivalent to

```
sc.textFile("purplecow.txt").map(lambda line: line.upper()) \
  .filter(lambda line: line.startswith('I')).count()
3
```



Use . to chain functions (dot is a pipeline)

Functional programming in Spark

Functional programming in Spark

- Spark depends heavily on the concepts of functional programming
  - Functions are the fundamental unit of programming
  - Functions have input and output only
- Key concepts
  - Passing functions as input to other functions
  - Anonymous functions

Passing named functions

purplecow.txt - Notepad

File Edit Format View Help

```
I never saw a purple cow.
I never hope to see one.
But I can tell you, anyhow,
I'd rather see than be one!
```

purplecow.txt file

Cmd 4

```
1  def toUpper(s):
2   return s.upper()
3
4  mydata = sc.textFile("FileStore/tables/purplecow.txt")
5
6  mydata.map(toUpper).take(2)
```

▸ (1) Spark Jobs

```
Out[14]: ['I NEVER SAW A PURPLE COW.', 'I NEVER HOPE TO SEE ONE.']
```

Passing anonymous functions

Cmd 5

```
1  mydata.map(lambda line : line.upper()).take(2)
```

▸ (1) Spark Jobs

```
Out[15]: ['I NEVER SAW A PURPLE COW.', 'I NEVER HOPE TO SEE ONE.']
```

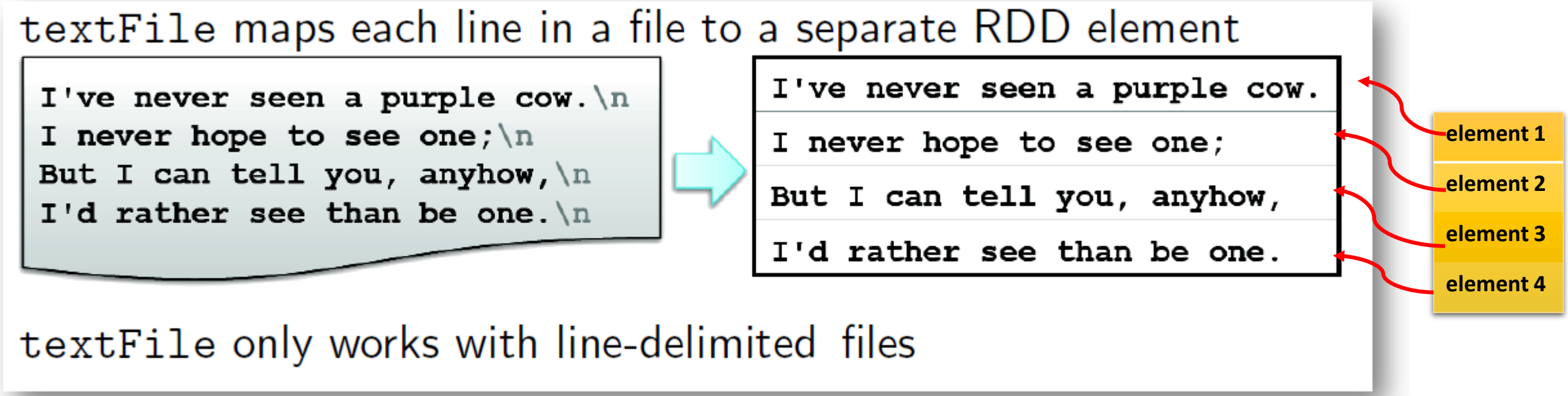
Lambda functions are best for short, one-of functions

sc.TextFile

vs

sc.wholeTextFiles

Can Sc.TextFile works with all files?



What about multiple files in a directory?

sc.wholeTextFiles(directory)

maps entire contents of each file in a directory
to a single RDD element.

Works only for small files (elements must fit in memory)

file1.json

```
{
 "firstName":"Fred",
 "lastName":"Flintstone",
 "userid":"123"
}
```

file2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234"
}
```

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

QC1.json - Notepad

File Edit Format View Help

```
{"quiz":{"sport":{"q1":{"question": "Which one is
"maths":{"q1":{"question": "18 + 21 = ?","options"
```

QC2.json - Notepad

File Edit Format View Help

```
{"quiz":{"sport":{"q1":{"question":"Which one is
"maths":{"q1":{"question": "3 + 6 = ?","options":
```

QC3.json - Notepad

File Edit Format View Help

```
{"quiz":{"sport":{"q1":{"question":"Which one is
"maths":{"q1":{"question": "5 + 7 = ?","options":
```

We have 3 json files and we want to make one RDD from all 3 files

```
1  import json
2
3  myrdd1= sc.wholeTextFiles("/FileStore/tables/QC*.json")
4
5  myrdd1.take(3)
```

QC* means QC1
and QC2 and QC3

▸ (2) Spark Jobs

```
Out[45]: [('dbfs:/FileStore/tables/QC1.json',
  '{"quiz":{"sport":{"q1":{"question": "Which one is correct team name in NBA?","options":["Mi
s"}},\r\n"maths":{"q1":{"question": "18 + 21 = ?","options":["32","39","41","28"],"answer": "3
2"}}}}'),
 ('dbfs:/FileStore/tables/QC2.json',
  '{"quiz":{"sport":{"q1":{"question":"Which one is correct team name in NBA?","options":["Bos
s"}},\r\n"maths":{"q1":{"question": "3 + 6 = ?","options":["9","10","11","17"],"answer": "9"},'
 ('dbfs:/FileStore/tables/QC3.json',
  '{"quiz":{"sport":{"q1":{"question":"Which one is correct team name in NBA?","options":["New
uston Rocket"}},\n"maths":{"q1":{"question": "5 + 7 = ?","options":["10","11","12","13"],"answ
r":"4"}}}}')]
```

QC1.json is
1st element of myrdd1

QC2.json is
2nd element of myrdd1

QC3.json is 3rd
element of myrdd1

List of RDD Operations

(Transformations and Actions)

Single RDD Transformations

- `.map(f)` – returns a new RDD formed by passing each element through a function `f`
- `.filter(f)` – returns a new RDD based on selecting elements for which the function `f` returns true
- `.flatMap(f)` – similar to map, but the new RDD flattens all elts
- `.sample(withReplacement, fraction, seed)` – samples a fraction of the data, with or without replacement
- `.distinct()` – returns a new RDD containing the distinct elements of the source RDD
- `.zipWithIndex()` – appends (or ZIPs) the RDD with the elt indices
- `.reduceByKey(f)` – reduces the elts of the RDD using `f` by key
- `.sortByKey(asc)` – orders (key, value) RDD by key and returns an RDD in ascending or descending order
- `.sortBy` – use provided function to sort

Map vs flatMap

Input RDD:

python Copy Edit

```python
rdd = sc.parallelize(["a,b,c", "d,e", "f"])
```

Using `map`

python Copy Edit

```python
mapped_rdd = rdd.map(lambda x: x.split(","))
print(mapped_rdd.collect())  # Output: [['a', 'b', 'c'], ['d', 'e'], ['f']]
```

Explanation: Each input string is split into a list, resulting in a nested list structure.

Using `flatMap`

python Copy Edit

```python
flat_mapped_rdd = rdd.flatMap(lambda x: x.split(","))
print(flat_mapped_rdd.collect())  # Output: ['a', 'b', 'c', 'd', 'e', 'f']
```

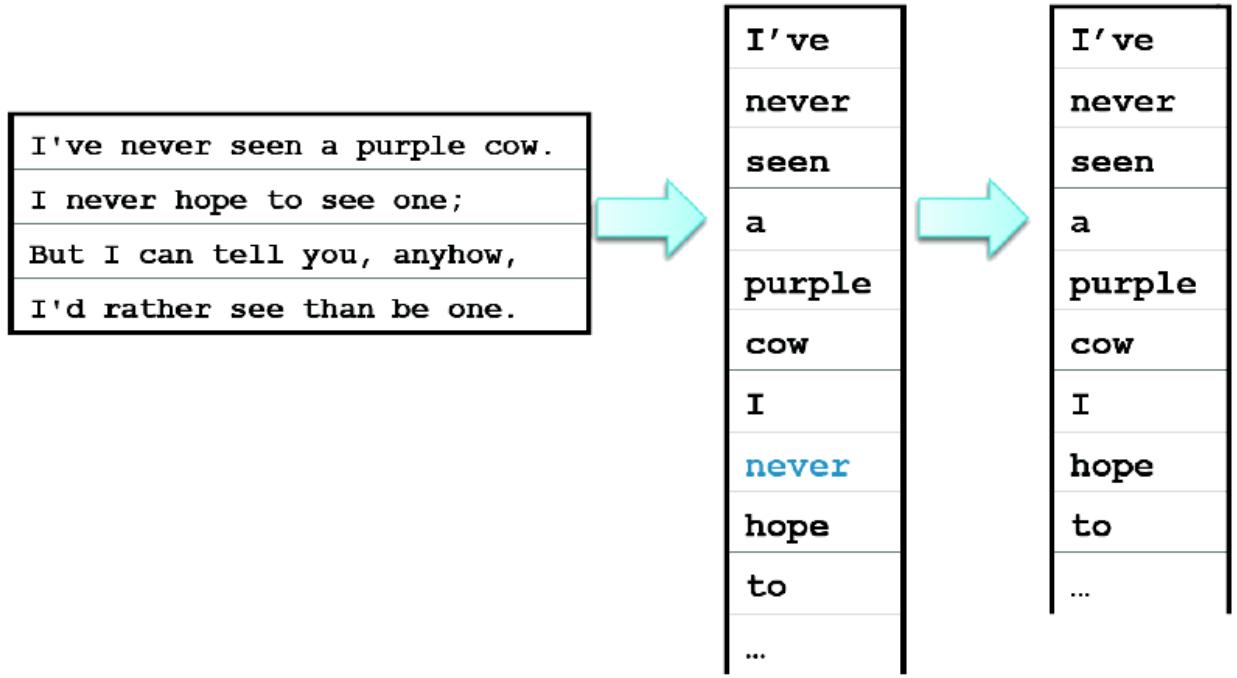
Explanation: The lists are "flattened" into a single RDD of elements.

Key Differences Between `map` and `flatMap`

Feature	`map`	`flatMap`
Output elements	One output element for each input element	Zero, one, or more output elements per input
Resulting RDD	Same number of elements as input	Flattens multiple outputs into a single RDD
Use case	Use when you want a **1:1 mapping**	Use when you want a **1:N mapping** or flattening

Example: flatMap and distinct

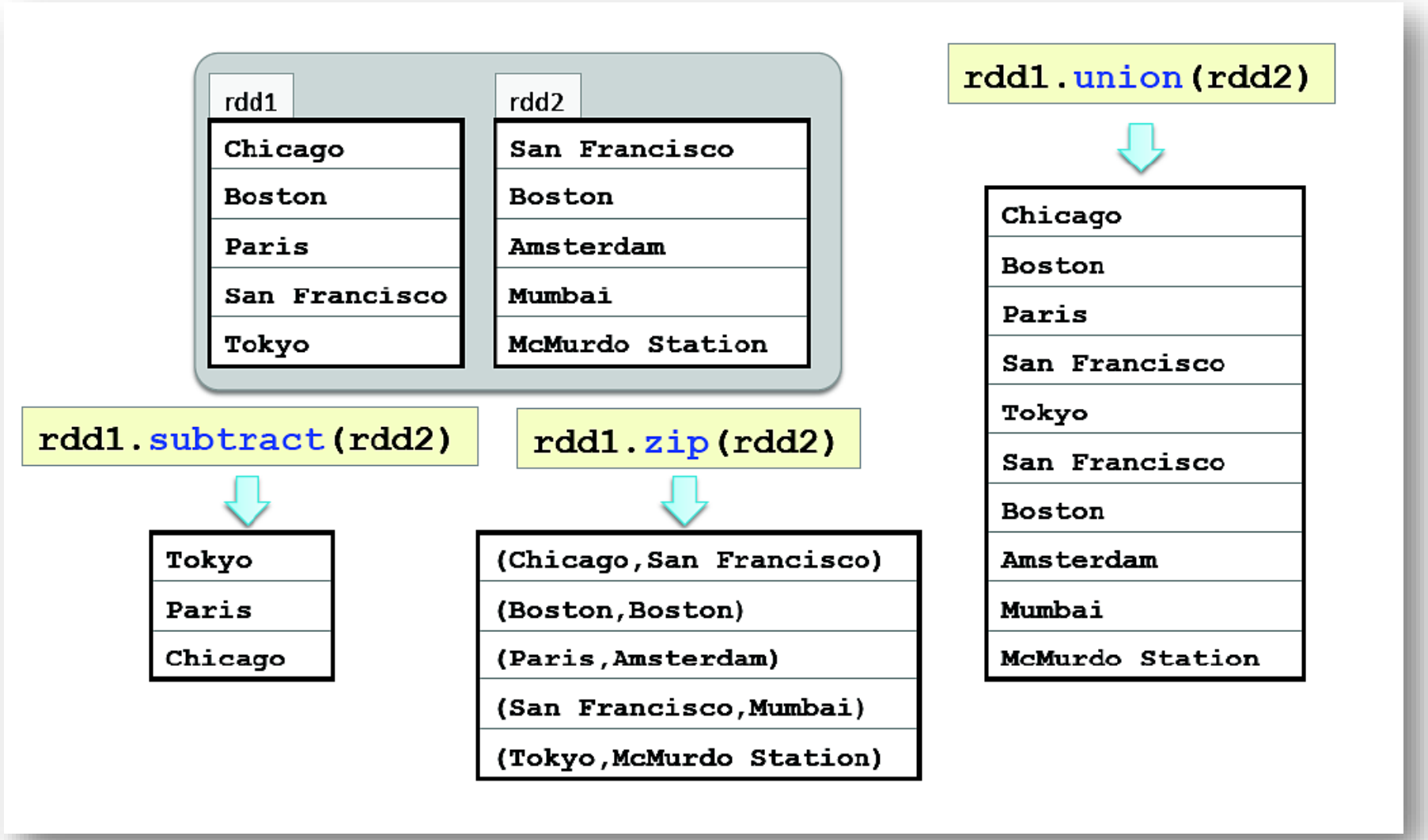
```
sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .distinct()
```



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
- `join` – joins two RDDs. Outer joins are supported through left outer join, right outer join, and full outer join.

Example: subtract, zip and union



RDD Actions

- `.take()` – returns an array with the first `n` elements of the RDD.
- `.collect()` – returns all of the elements from the workers to the driver.
- `.reduce(f)` – aggregates the elements of an RDD by `f`. The `f` function should be commutative and associative* so that it can be computed correctly in parallel.
- `.count()` - returns the number of elements in the RDD.

***commutative** and **associative**, meaning that the order of operations does not affect the final result.

RDD Actions

- `.take()` – returns an array with the first `n` elements of the RDD.
- `.collect()` – returns all of the elements from the workers to the driver.
- `.reduce(f)` – aggregates the elements of an RDD by `f`. The `f` function should be commutative and associative* so that it can be computed correctly in parallel.
- `.count()` - returns the number of elements in the RDD.

***commutative** and **associative**, meaning that the order of operations does not affect the final result.

Some other RDD operations: Reduce(f)

Sum of All Elements

python

```python
rdd = sc.parallelize([1, 2, 3, 4, 5])
total_sum = rdd.reduce(lambda x, y: x + y)
print(total_sum)  # Output: 15
```

Finding the Maximum Value

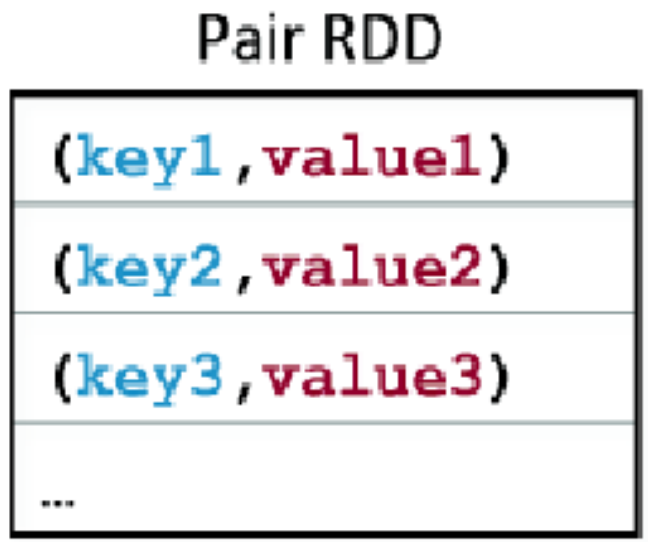
python

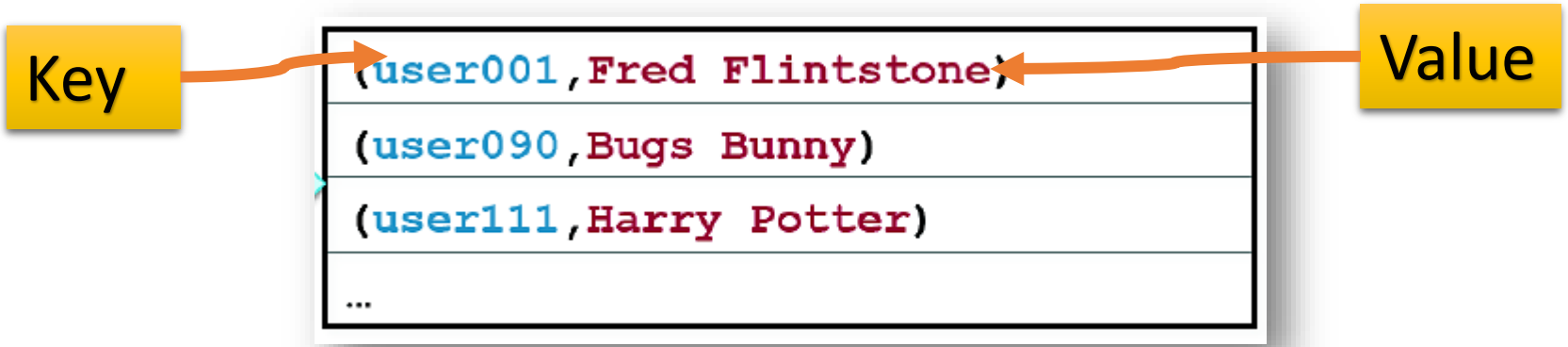
```python
rdd = sc.parallelize([10, 20, 5, 8, 30])
max_value = rdd.reduce(lambda x, y: x if x > y else y)
print(max_value)  # Output: 30
```

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





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?

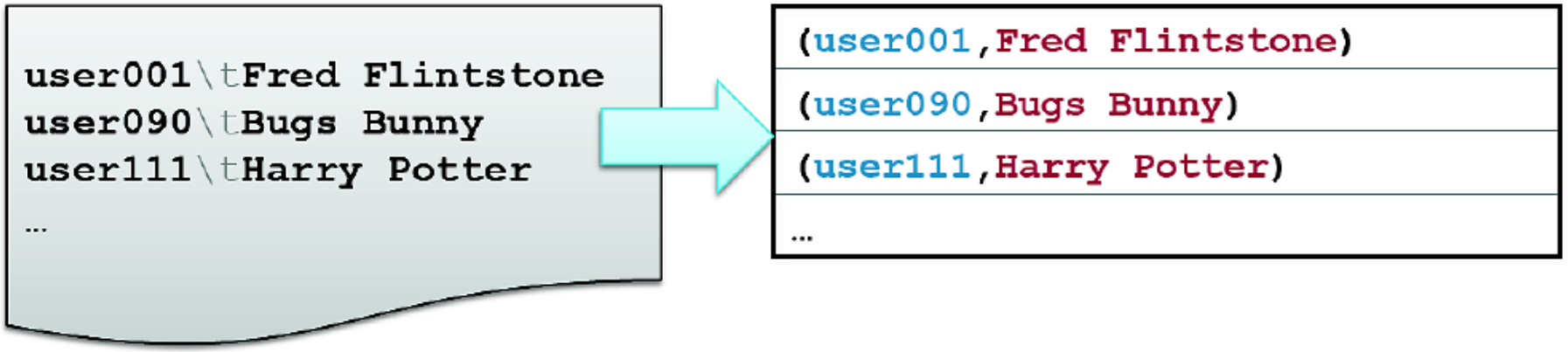
Commonly used functions to create Pair RDDs

- `map()`
- `flatMap()` / `flatMapValues()`
- `keyBy()`

Example: creating a simple pair RDD

Example: create a pair RDD from a tab-separated file

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



Example 1: Keying web logs by user ID

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

User ID

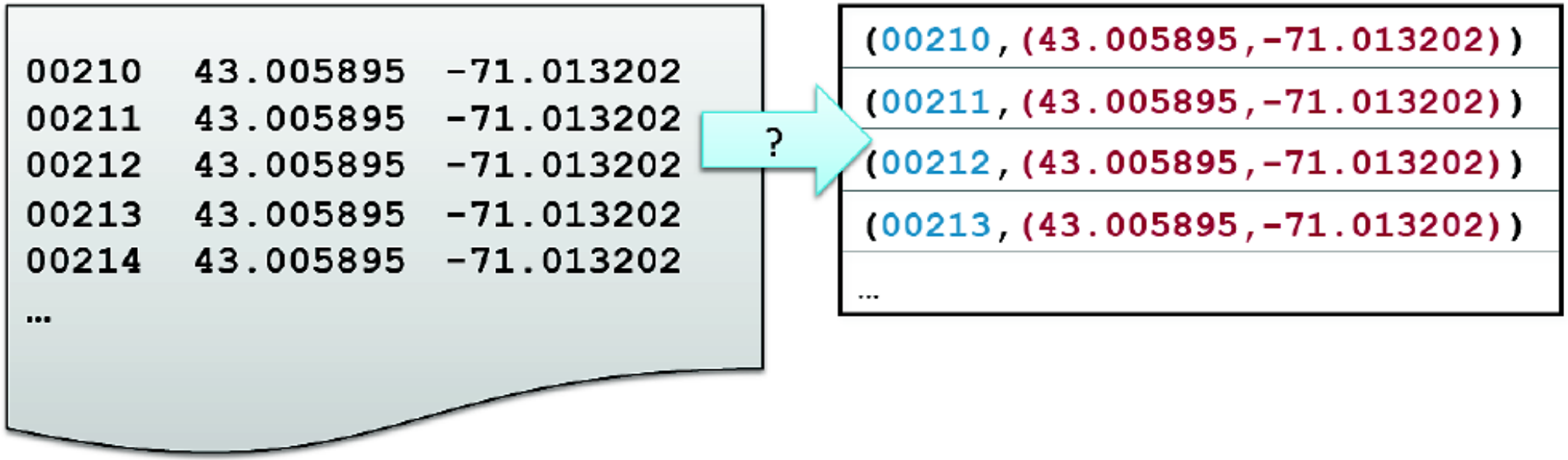
```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

```
(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: Pairing 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: Pairing complex values

First, split the contents of each line
Second, put them together in a new format

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

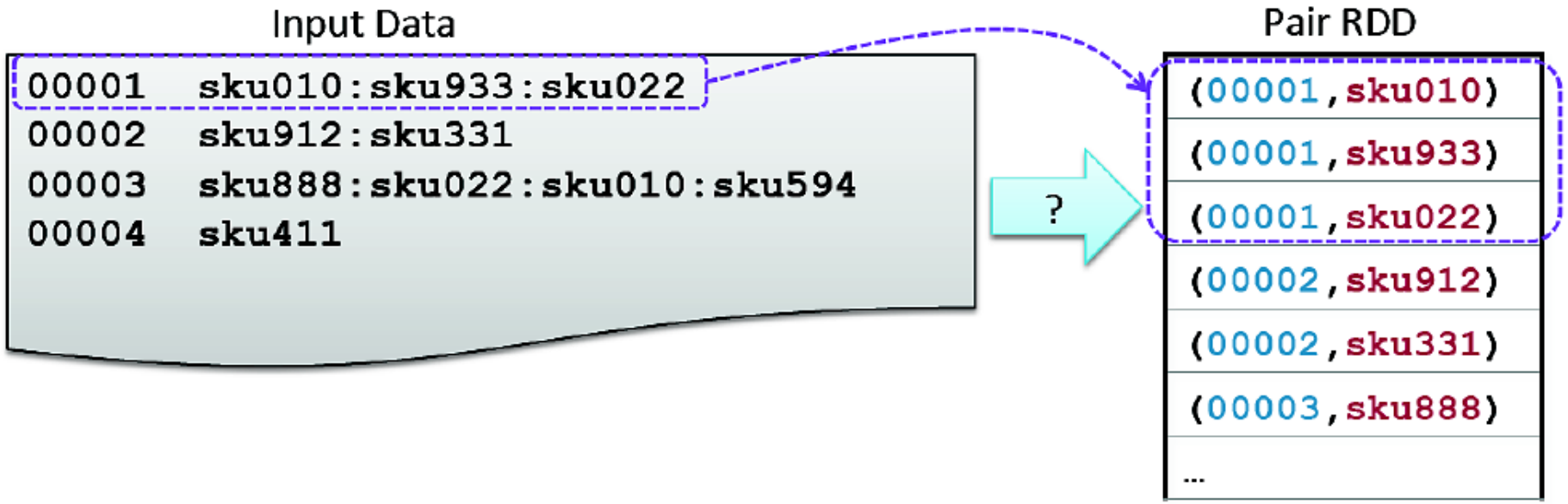
```
00210   43.005895   -71.013202
01014   42.170731   -72.604842
01062   42.324232   -72.67915
01263   42.3929     -73.228483
…
```

```
(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

How would you do this?

- Input: order numbers with a list of SKUs in the order
- Output: order (key) and sku (value)



Answer 2: Mapping single rows to multiple pairs

1

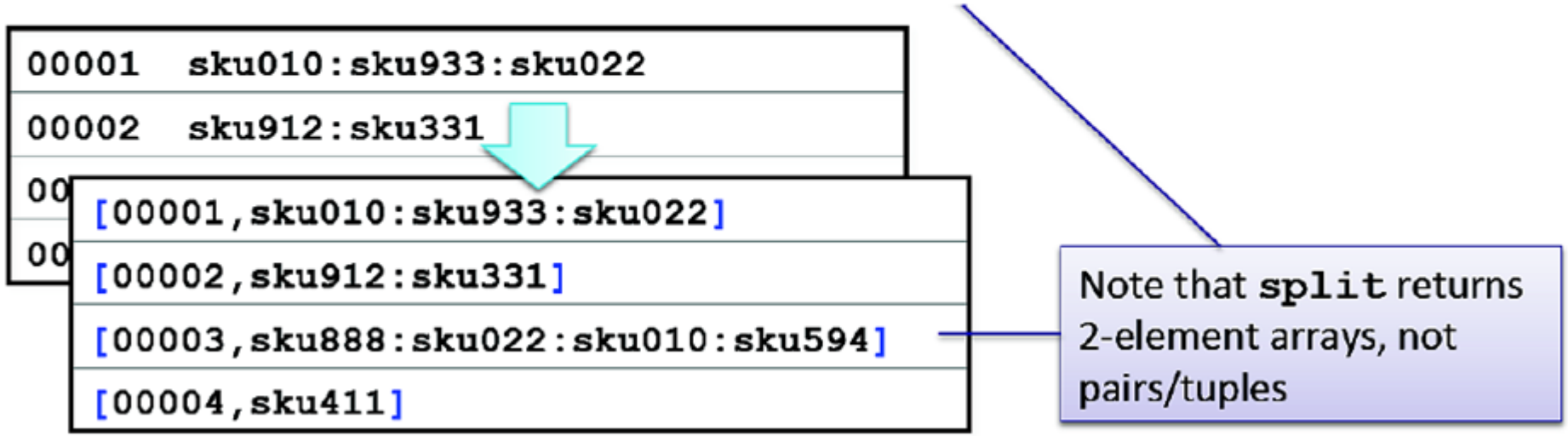
```
sc.textFile(file)
```

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

Answer 2: Mapping single rows to multiple pairs

2

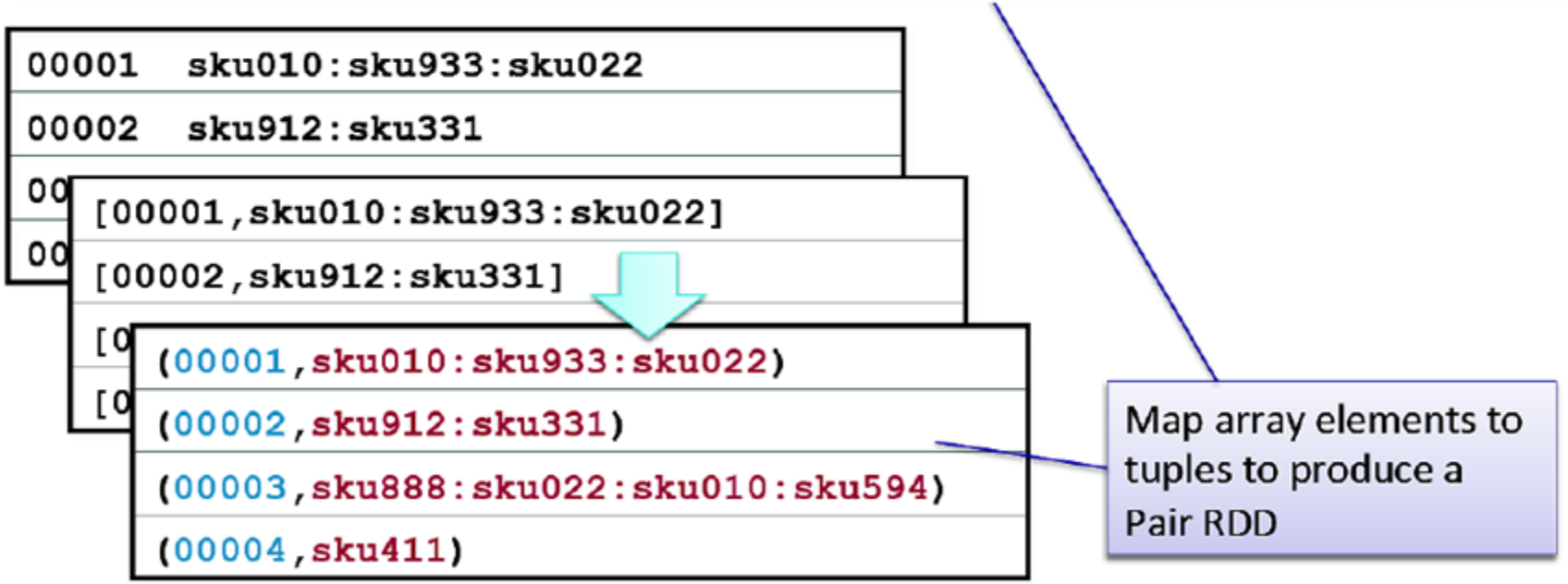
```
sc.textFile(file)\
   .map(lambda line: line.split('   '))
```



Answer 2: Mapping single rows to multiple pairs

3

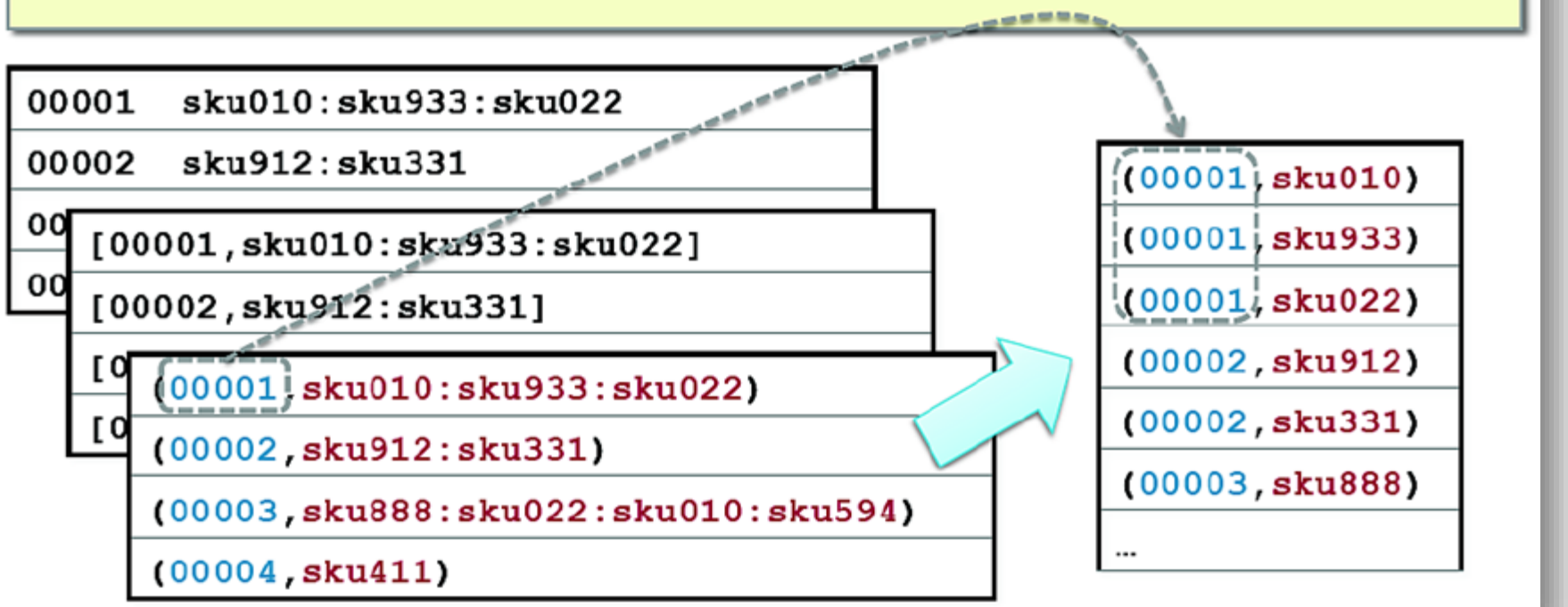
```
sc.textFile(file)\
   .map(lambda line:  line.split(''))\
   .map(lambda fields: (fields[0],fields[1]))
```



Answer 2: Mapping single rows to multiple pairs

4

```
sc.textFile(file) \
.map(lambda line: line.split('  ')) \
.map(lambda fields: (fields[0],fields[1]))
.flatMapValues(lambda skus: skus.split(':'))
```



Map-reduce programming in Spark

- Map-reduce is a common programming model
- Easily applicable to distributed processing of large data sets

Map-reduce in Hadoop

Hadoop MapReduce is the major implementation

Somewhat limited

- Each job has one Map phase, one Reduce phase
- Job output is saved to files

Map-reduce in Spark

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

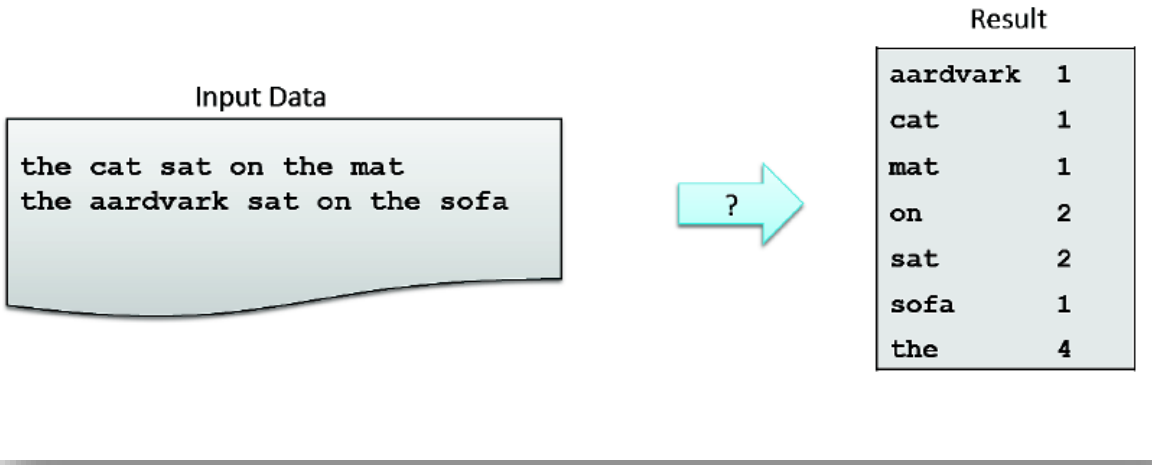
Map phase

- 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`

Map-reduce example: Word count



Creating the initial RDD

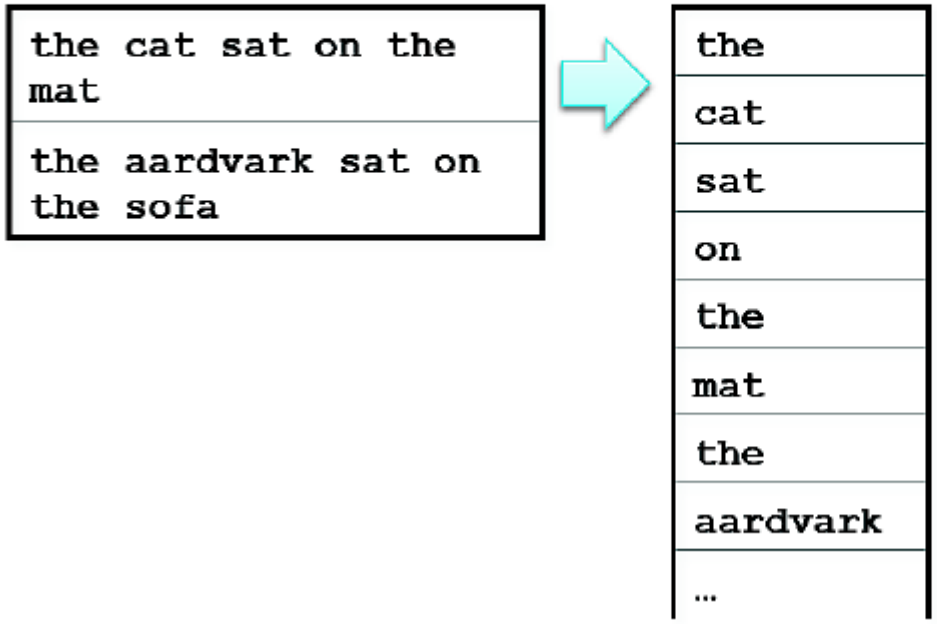
1

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

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

First Mapping

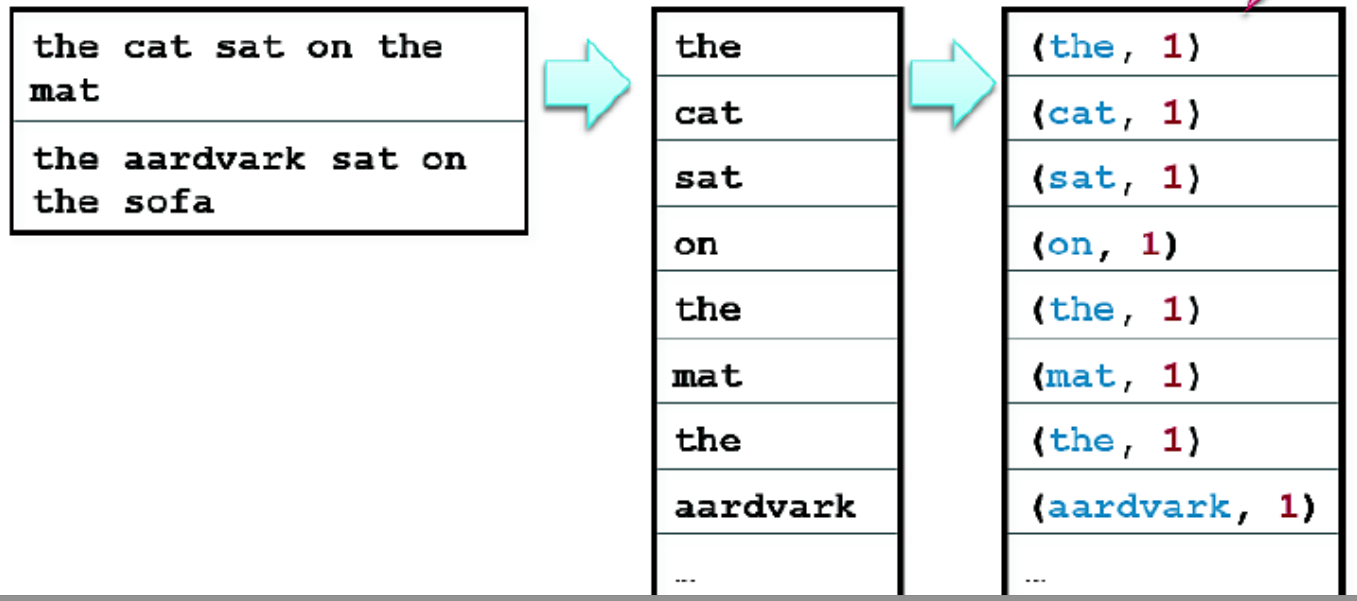

```
counts = sc.textFile(file)\
    .flatMap(lambda line: line.split())
```



Second Mapping

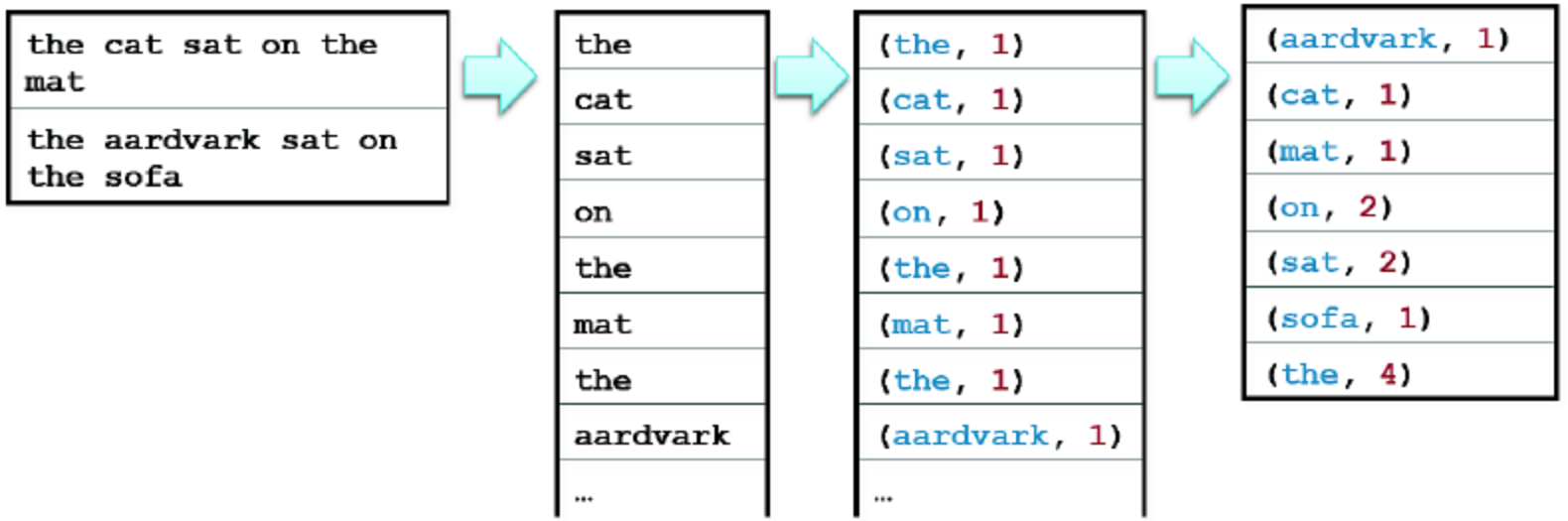

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Reducing

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



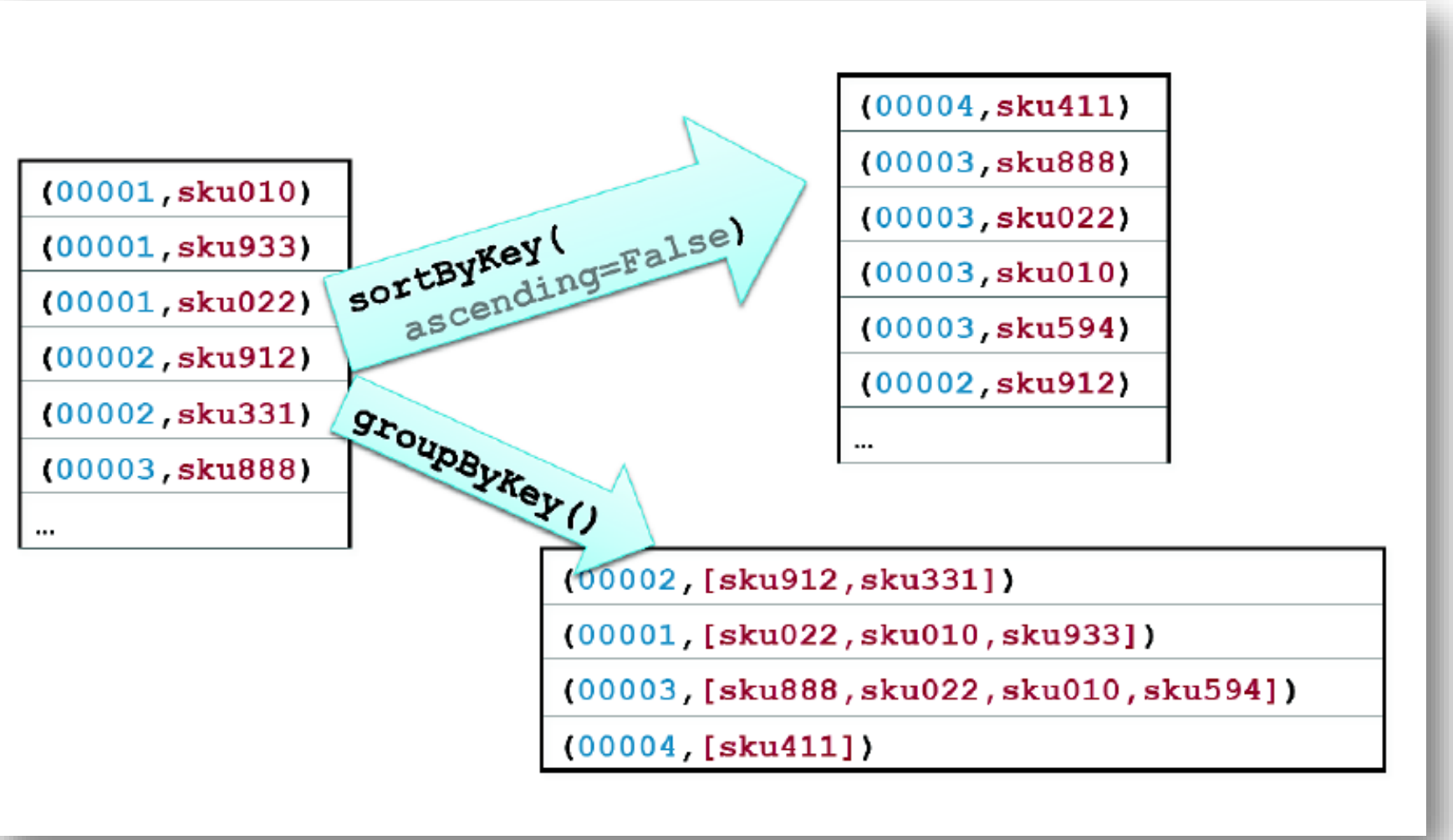
Pair RDD Operations

Pair RDD operations

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

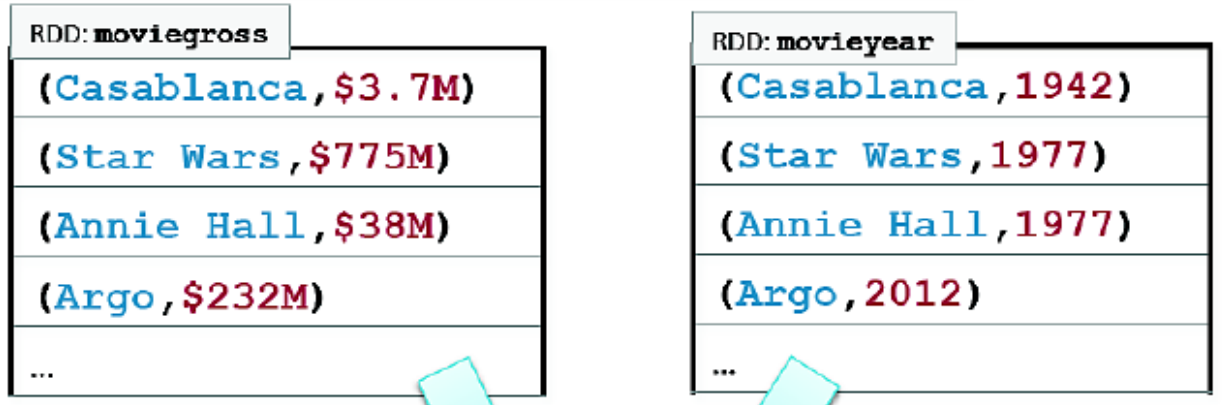
- Examples
  - `reduceByKey` – merge the values for each key using a reduce function
  - `countByKey` – return a map with the count of occurrences of each key
  - `groupByKey` – group all the values for each key in an RDD
  - `sortByKey` – sort in ascending or descending order
  - `join` – return an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

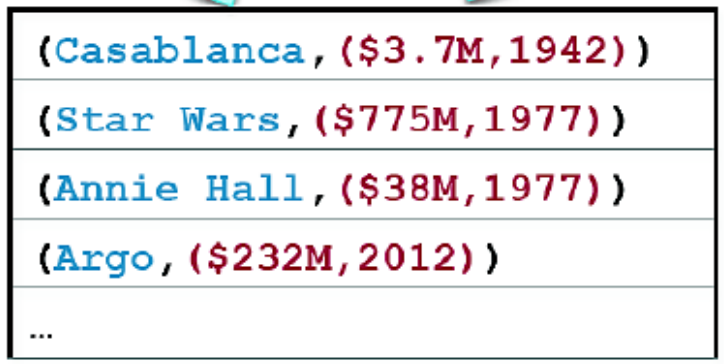
Example of pair RDD operations



Example: Joining by key

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





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

The following offer more information on topics discussed in this lecture:

Apache Spark Documentation
https://spark.apache.org/docs/latest/

RDD Programming Guide
https://spark.apache.org/docs/latest/rdd-programming-guide.html

Chapter 4: In-Memory Computing with Spark Book (Data Analytics with Hadoop)