cloudera®

Developer Training for Spark and Hadoop I

cloudera®

Capturing Data with Apache Flume

Chapter 9

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ **Capturing Data with Apache Flume**	**Ingesting Streaming Data**
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ Conclusion	Course Conclusion

Capturing Data with Apache Flume

In this chapter you will learn

- **What are the main architectural components of Flume**
- **How these components are configured**
- **How to launch a Flume agent**
- **How to configure a standard Java application to log data using Flume**

Chapter Topics

Capturing Data with Apache Flume | **Introduction to Flume**

- **What is Apache Flume?**
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

What Is Apache Flume?

- **Apache Flume is a high-performance system for data collection**
  - Name derives from original use case of near-real time log data ingestion
  - Now widely used for collection of any streaming event data
  - Supports aggregating data from many sources into HDFS
- **Originally developed by Cloudera**
  - Donated to Apache Software Foundation in 2011
  - Became a top-level Apache project in 2012
  - Flume OG gave way to Flume NG (Next Generation)
- **Benefits of Flume**
  - Horizontally-scalable
  - Extensible
  - Reliable

Flume's Design Goals: Reliability

- **Channels provide Flume's reliability**
- **Memory Channel**
  - Data will be lost if power is lost
- **Disk-based Channel**
  - Disk-based queue guarantees durability of data in face of a power loss
- **Data transfer between Agents and Channels is transactional**
  - A failed data transfer to a downstream agent rolls back and retries
- **Can configure multiple Agents with the same task**
  - For example, 2 Agents doing the job of 1 'collector' – if one agent fails then upstream agents would fail over

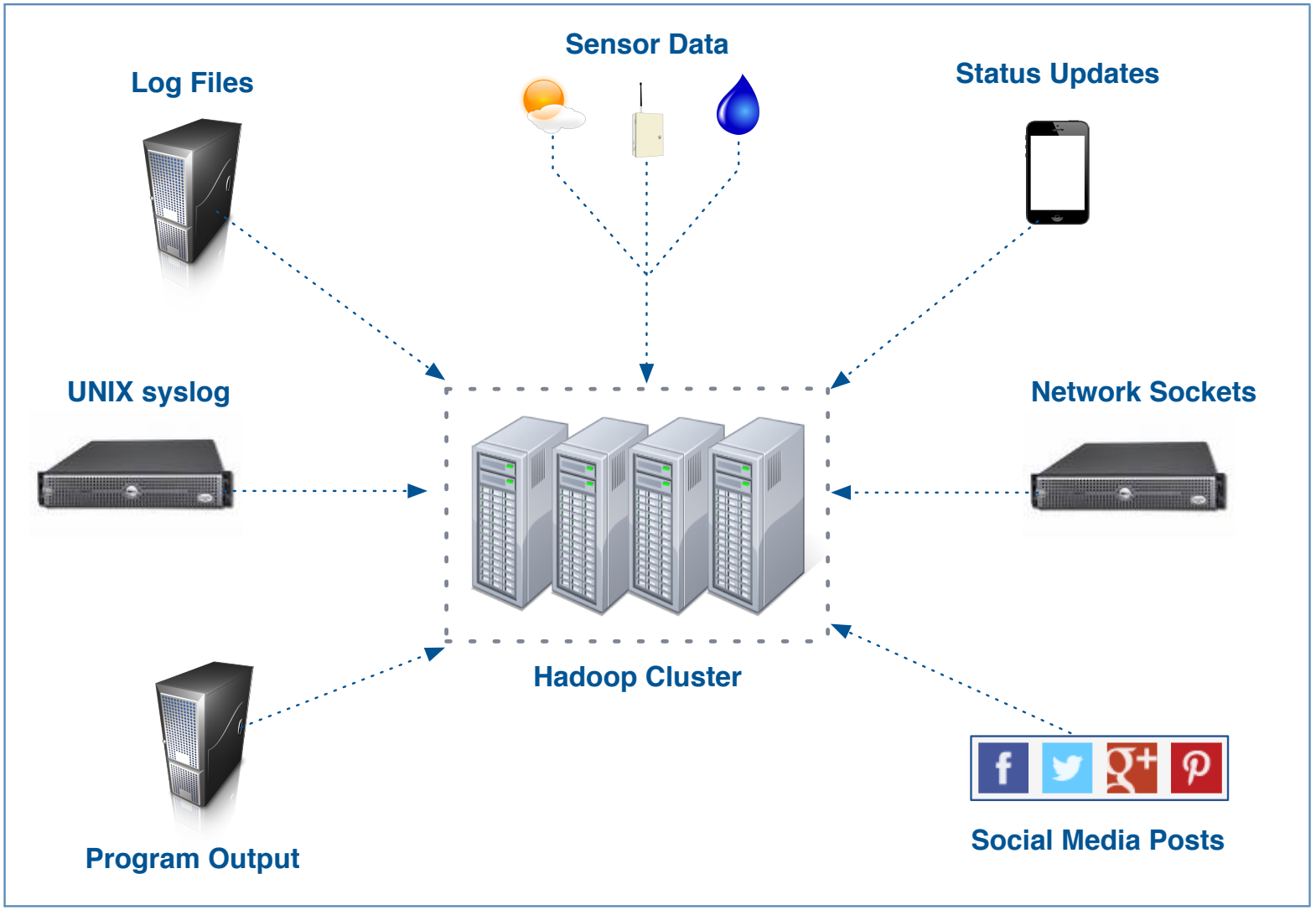
Flume's Design Goals: Scalability

- **Scalability**
  - The ability to increase system performance linearly – or better – by adding more resources to the system
  - Flume scales horizontally
    - As load increases, more machines can be added to the configuration

Flume's Design Goals: Extensibility

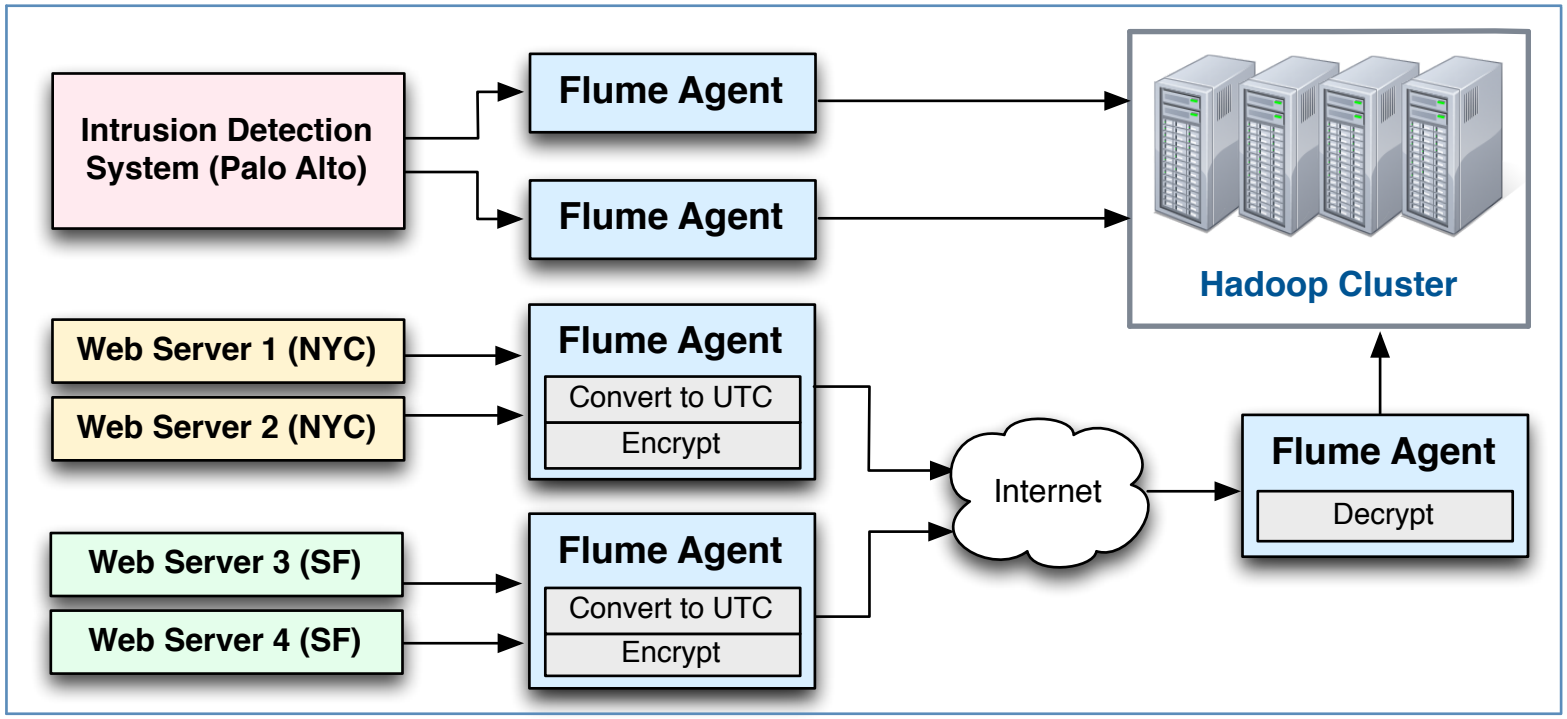
- **Extensibility**
  - The ability to add new functionality to a system
- **Flume can be extended by adding Sources and Sinks to existing storage layers or data platforms**
  - General Sources include data from files, syslog, and standard output from any Linux process
  - General Sinks include files on the local filesystem or HDFS
  - Developers can write their own Sources or Sinks

Common Flume Data Sources



Large-Scale Deployment Example

- **Flume collects data using configurable "agents"**
  - Agents can receive data from many sources, including other agents
  - Large-scale deployments use multiple tiers for scalability and reliability
  - Flume supports inspection and modification of in-flight data



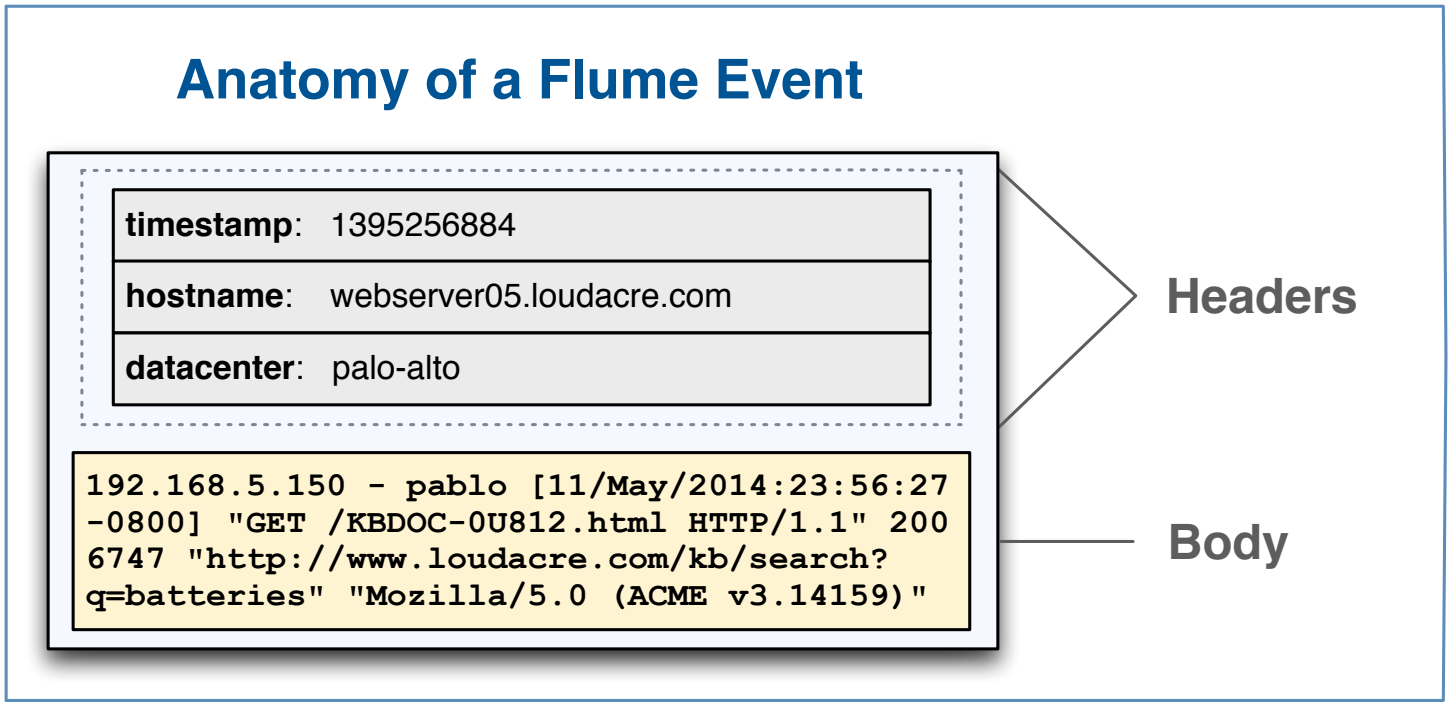
Chapter Topics

Capturing Data with Apache Flume | **Introduction to Flume**

- What is Apache Flume?
- **Basic Flume Architecture**
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Flume Events

- **An *event* is the fundamental unit of data in Flume**
  - Consists of a body (payload) and a collection of headers (metadata)
- **Headers consist of name-value pairs**
  - Headers are mainly used for directing output

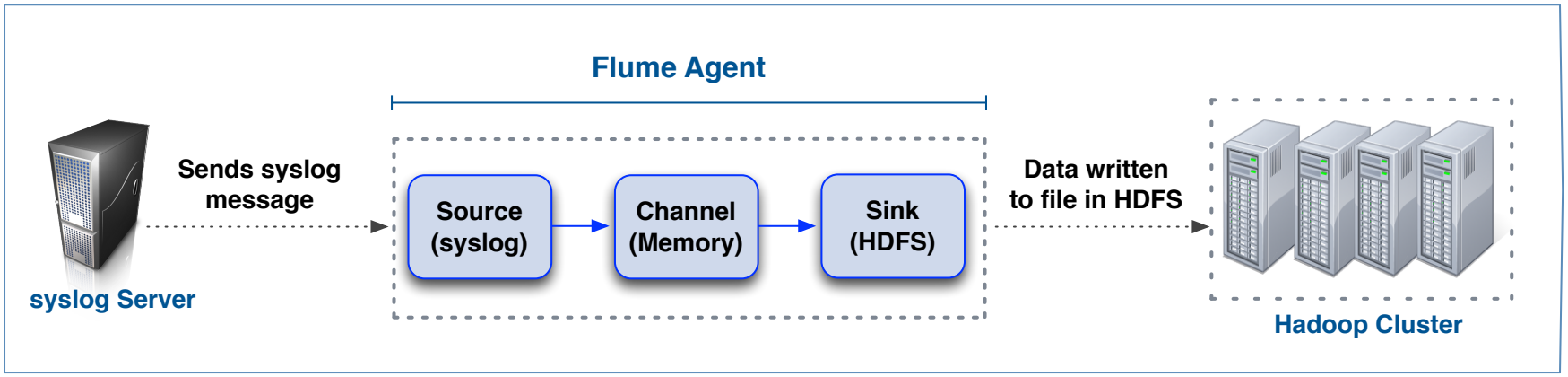


Components in Flume's Architecture

- **Source**
  - Receives events from the external actor that generates them
- **Sink**
  - Sends an event to its destination
- **Channel**
  - Buffers events from the source until they are drained by the sink
- **Agent**
  - Java process that configures and hosts the source, channel, and sink

Flume Data Flow

- **This diagram illustrates how syslog data might be captured to HDFS**
  1. Message is logged on a server running a syslog daemon
  2. Flume agent configured with syslog source receives event
  3. Source pushes event to the channel, where it is buffered in memory
  4. Sink pulls data from the channel and writes it to HDFS



Chapter Topics

Capturing Data with Apache Flume | **Introduction to Flume**

- What is Apache Flume?
- Basic Flume Architecture
- **Flume Sources**
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Notable Built-in Flume Sources

- **Syslog**
  - Captures messages from UNIX syslog daemon over the network
- **Netcat**
  - Captures any data written to a socket on an arbitrary TCP port
- **Exec**
  - Executes a UNIX program and reads events from standard output *
- **Spooldir**
  - Extracts events from files appearing in a specified (local) directory
- **HTTP Source**
  - Receives events from HTTP requests

* Asynchronous sources do not guarantee that events will be delivered

Chapter Topics

Interesting Built-in Flume Sinks

- **Null**
  - Discards all events (Flume equivalent of `/dev/null`)
- **Logger**
  - Logs event to INFO level using SLF4J
- **IRC**
  - Sends event to a specified Internet Relay Chat channel
- **HDFS**
  - Writes event to a file in the specified directory in HDFS
- **HBaseSink**
  - Stores event in HBase

SLF4J: Simple Logging Façade for Java

Chapter Topics

Built-In Flume Channels

- **Memory**
  - Stores events in the machine's RAM
  - Extremely fast, but not reliable (memory is volatile)
- **File**
  - Stores events on the machine's local disk
  - Slower than RAM, but more reliable (data is written to disk)
- **JDBC**
  - Stores events in a database table using JDBC
  - Slower than file channel

Chapter Topics

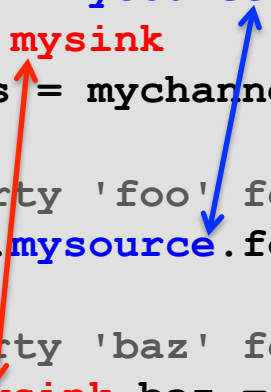
Capturing Data with Apache Flume | **Introduction to Flume**

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- **Flume Configuration**
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Flume Agent Configuration File

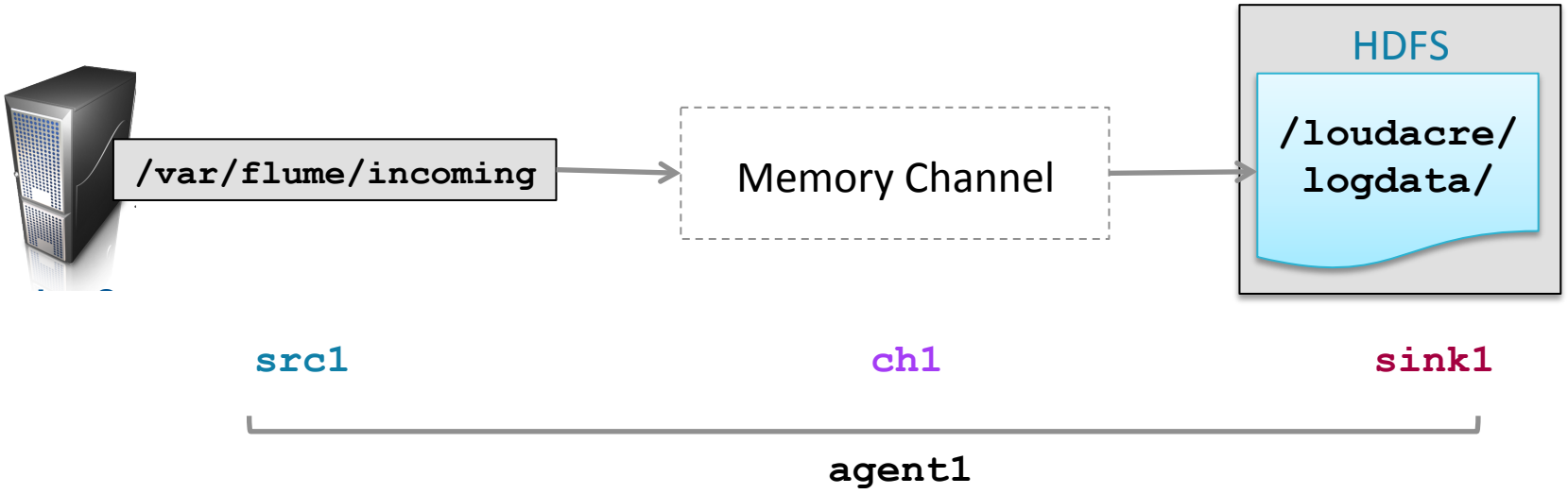
- **Flume agent is configured through a Java properties file**
  - Multiple agents can be configured in a single file
- **The configuration file uses hierarchical references**
  - Each component is assigned a user-defined ID
  - That ID is used in the names of additional properties

```
# Define sources, sinks, and channel for agent named 'agent1'
agent1.sources = mysource
agent1.sinks = mysink
agent1.channels = mychannel

# Sets a property 'foo' for the source associated with agent1
agent1.sources.mysource.foo = bar

# Sets a property 'baz' for the sink associated with agent1
agent1.sinks.mysink.baz = bat
```



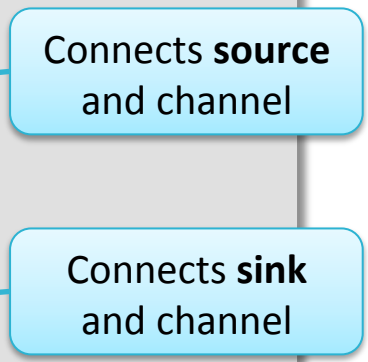
Example: Configuring Flume Components (1)

- **Example: Configure a Flume Agent to collect data from remote spool directories and save to HDFS**



Example: Configuring Flume Components (2)

```
agent1.sources = src1
agent1.sinks = sink1
agent1.channels = ch1

agent1.channels.ch1.type = memory

agent1.sources.src1.type = spooldir
agent1.sources.src1.spoolDir = /var/flume/incoming
agent1.sources.src1.channels = ch1

agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata
agent1.sinks.sink1.channel = ch1
```



- **Properties vary by component type (source, channel, and sink)**
  - Properties also vary by subtype (e.g., netcat source vs. syslog source)
  - See the Flume user guide for full details on configuration

Aside: HDFS Sink Configuration

- **Path may contain patterns based on event headers, such as timestamp**
- **The HDFS sink writes uncompressed SequenceFiles by default**
  - Specifying a codec will enable compression

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata/%y-%m-%d
agent1.sinks.sink1.hdfs.codeC = snappy
agent1.sinks.sink1.channel = ch1
```

- **Setting `fileType` parameter to `DataStream` writes *raw* data**
  - Can also specify a file extension, if desired

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata/%y-%m-%d
agent1.sinks.sink1.hdfs.fileType = DataStream
agent1.sinks.sink1.hdfs.fileSuffix = .txt
agent1.sinks.sink1.channel = ch1
```

Starting a Flume Agent

- **Typical command line invocation**
  - The **--name** argument must match the agent's name in the configuration file
  - Setting root logger as shown will display log messages in the terminal

```
$ flume-ng agent \
   --conf /etc/flume-ng/conf \
   --conf-file /path/to/flume.conf \
   --name agent1 \
   -Dflume.root.logger=INFO,console
```

* ng = Next Generation (prior version now referred to as og)

Chapter Topics

Capturing Data with Apache Flume | **Introduction to Flume**

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- **Conclusion**
- Hands-On Exercise: Collect Web Server Logs with Flume

Essential Points

- **Apache Flume is a high-performance system for data collection**
  - Scalable, extensible, and reliable
- **A Flume agent manages the source, channels, and sink**
  - Source receives event data from its origin
  - Sink sends the event to its destination
  - Channel buffers events between the source and sink
- **The Flume agent is configured using a properties file**
  - Each component is given a user-defined ID
  - This ID is used to define properties of that component

Bibliography

The following offer more information on topics discussed in this chapter

- **Flume User Guide**
  - `http://flume.apache.org/FlumeUserGuide.html`

Chapter Topics

Capturing Data with Apache Flume | **Introduction to Flume**

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- **Hands-On Exercise: Collect Web Server Logs with Flume**

Hands-On Exercise: Collect Web Server Logs with Flume

- **In this exercise you will**
  - Configure Flume to ingest web server log data to HDFS
- **Please refer to your Hands-On Exercise Manual for instructions**

cloudera®

Spark Basics

Chapter 10

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ **Spark Basics** ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Spark Basics

In this chapter you will learn

- **How to start the Spark Shell**
- **About the SparkContext**
- **Key Concepts of Resilient Distributed Datasets (RDDs)**
  - What are they?
  - How do you create them?
  - What operations can you perform with them?
- **How Spark uses the principles of functional programming**

Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- **What is Apache Spark?**
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming in Spark
- Conclusion
- Hands-On Exercises

What is Apache Spark?

- **Apache Spark is a fast and general engine for large-scale data processing**

- **Written in Scala**
  - Functional programming language that runs in a JVM
- **Spark Shell**
  - Interactive – for learning or data exploration
  - Python or Scala
- **Spark Applications**
  - For large scale data processing
  - Python, Scala, or Java

Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- What is Apache Spark?
- **Using the Spark Shell**
- RDDs (Resilient Distributed Datasets)
- Functional Programming in Spark
- Conclusion
- Hands-On Exercises

Spark Shell

- **The Spark Shell provides interactive data exploration (REPL)**
- **Writing Spark applications without the shell will be covered later**

Python Shell: **`pyspark`**

```
$ pyspark

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Python version 2.7.8 (default, Aug 27
2015 05:23:36)
SparkContext available as sc, HiveContext
available as sqlCtx.
>>>
```

Scala Shell: **`spark-shell`**

```
$ spark-shell

Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM)
64-Bit Server VM, Java 1.7.0_67)
Created spark context..
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

REPL: Read/Evaluate/Print Loop

Spark Context

- **Every Spark application requires a Spark Context**
  - The main entry point to the Spark API
- **Spark Shell provides a preconfigured Spark Context called `sc`**

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

Scala

```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- What is Apache Spark?
- Using the Spark Shell
- **RDDs (Resilient Distributed Datasets)**
- Functional Programming With Spark
- Conclusion
- Hands-On Exercise: Getting Started with RDDs

RDD (Resilient Distributed Dataset)

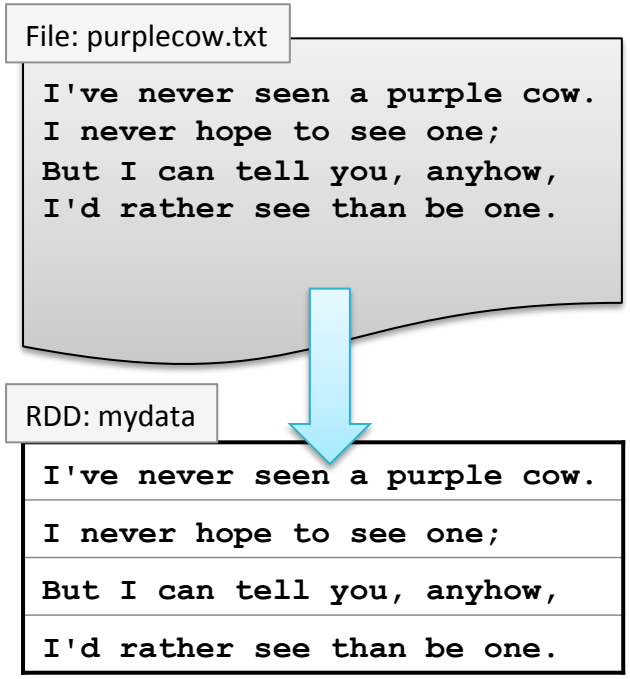
- **RDD (Resilient Distributed Dataset)**
  - Resilient – if data in memory is lost, it can be recreated
  - Distributed – processed across the cluster
  - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming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

Creating an RDD

- **Three ways to create an RDD**
  - From a file or set of files
  - From data in memory
  - From another RDD

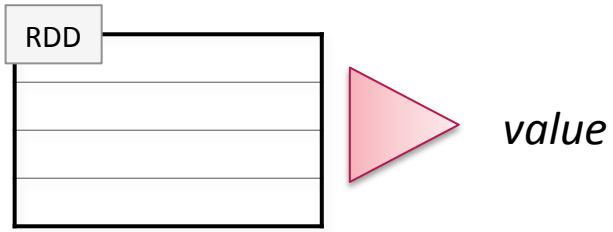
Example: A File-based RDD

```
> val mydata = sc.textFile("purplecow.txt")
…
15/01/29 06:20:37 INFO storage.MemoryStore:
   Block broadcast_0 stored as values to
   memory (estimated size 151.4 KB, free 296.8
   MB)

> mydata.count()

…
15/01/29 06:27:37 INFO spark.SparkContext: Job
   finished: take at <stdin>:1, took
   0.160482078 s
4
```

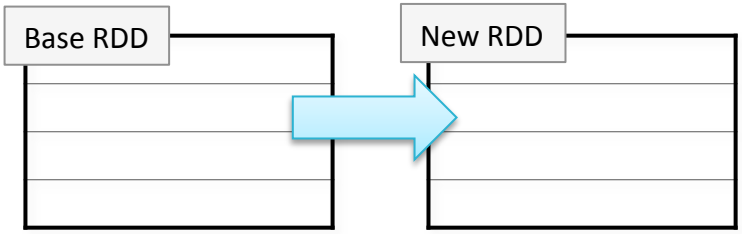


RDD Operations

- **Two types of RDD operations**
  - Actions – return values



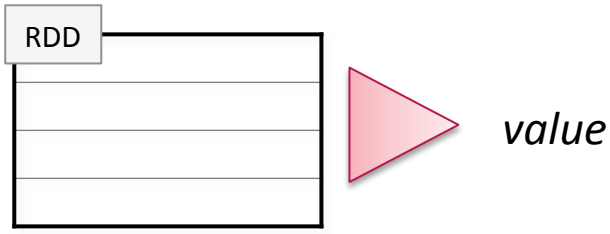
  - Transformations – define a new RDD based on the current one(s)



- **Pop quiz:**
  - Which type of operation is `count()`?

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)

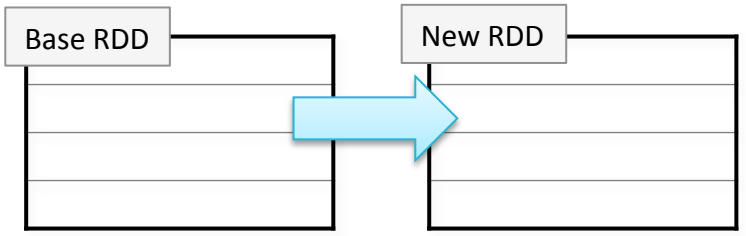



```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

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

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

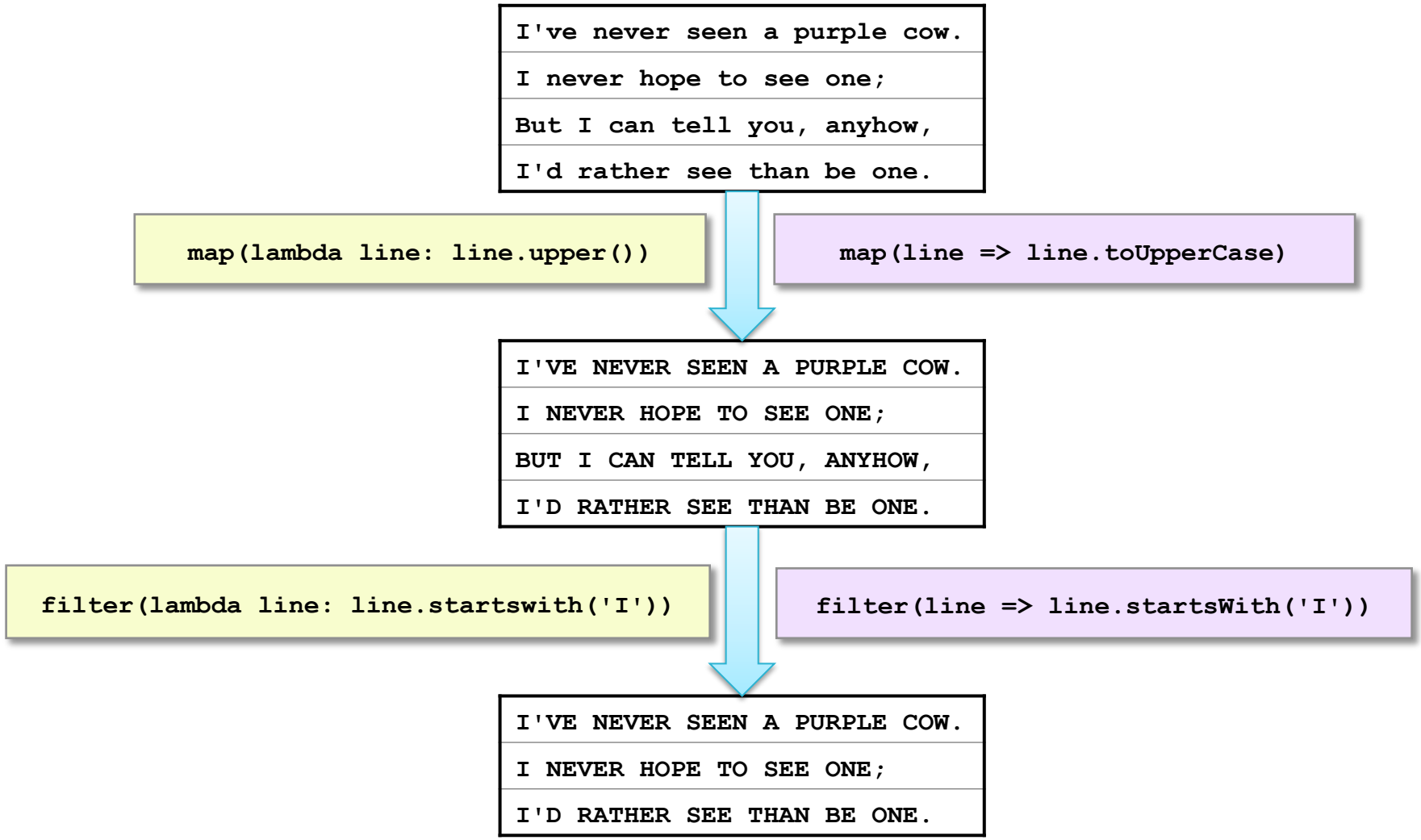
RDD Operations: Transformations

- **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` Transformations



Lazy Execution (1)

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

File: purplecow.txt

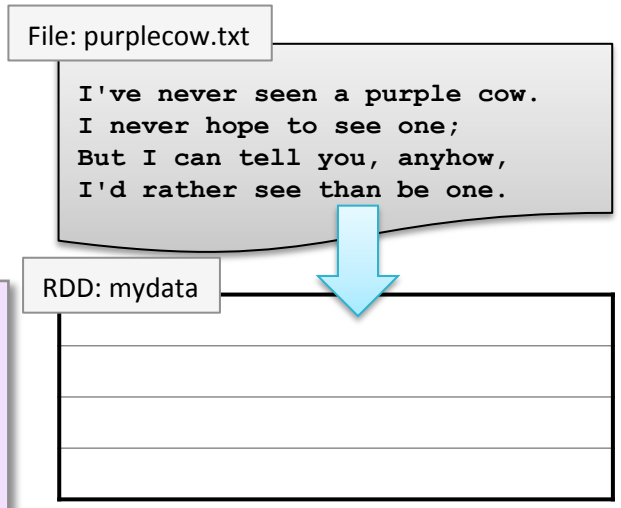
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution (2)

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

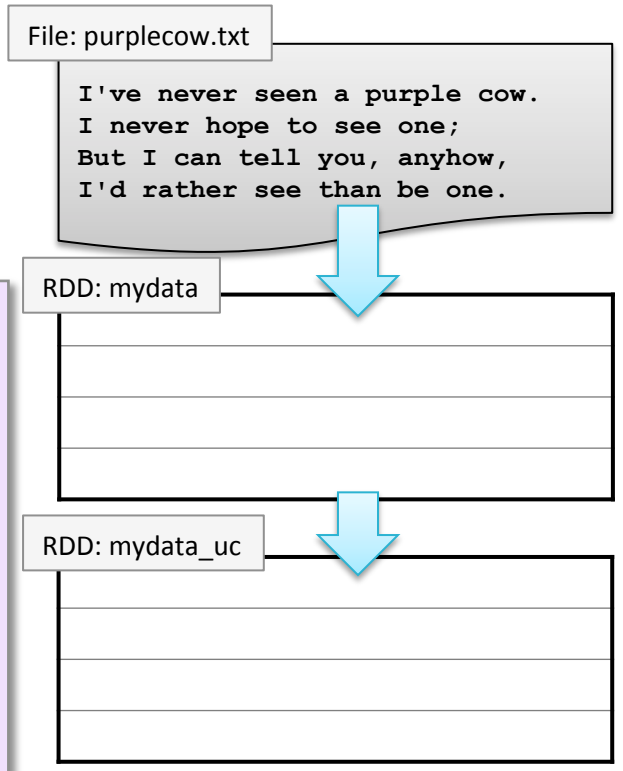
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution (3)

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

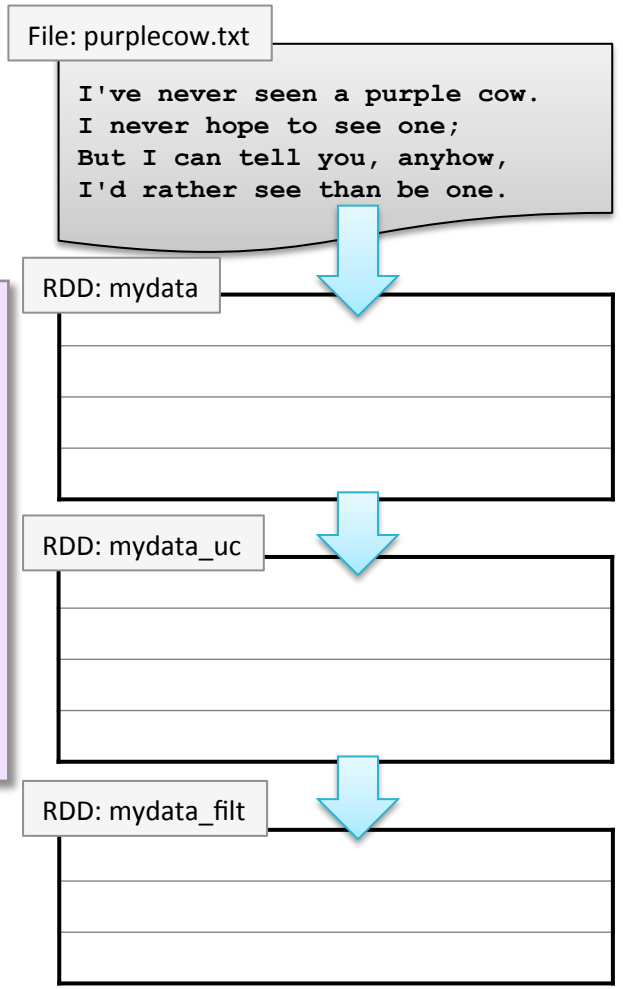
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution (4)

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

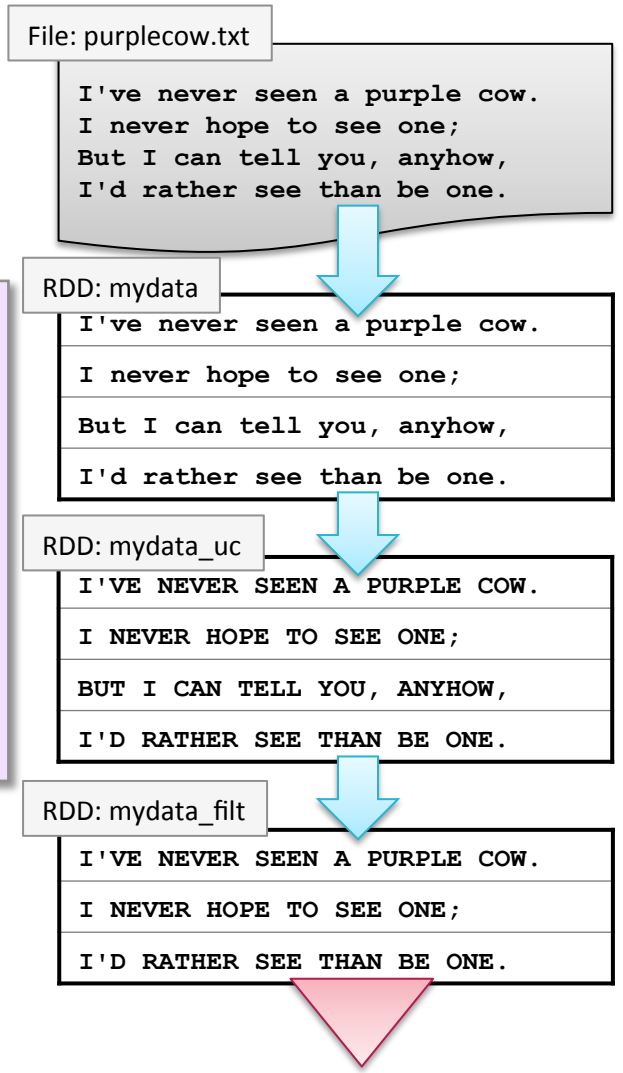
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution (5)

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

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Chaining Transformations (Scala)

- **Transformations may be chained together**

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line => line.startsWith("I"))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).
   filter(line => line.startsWith("I")).count()
3
```

Chaining Transformations (Python)

- **Same example in Python**

```
> 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
```

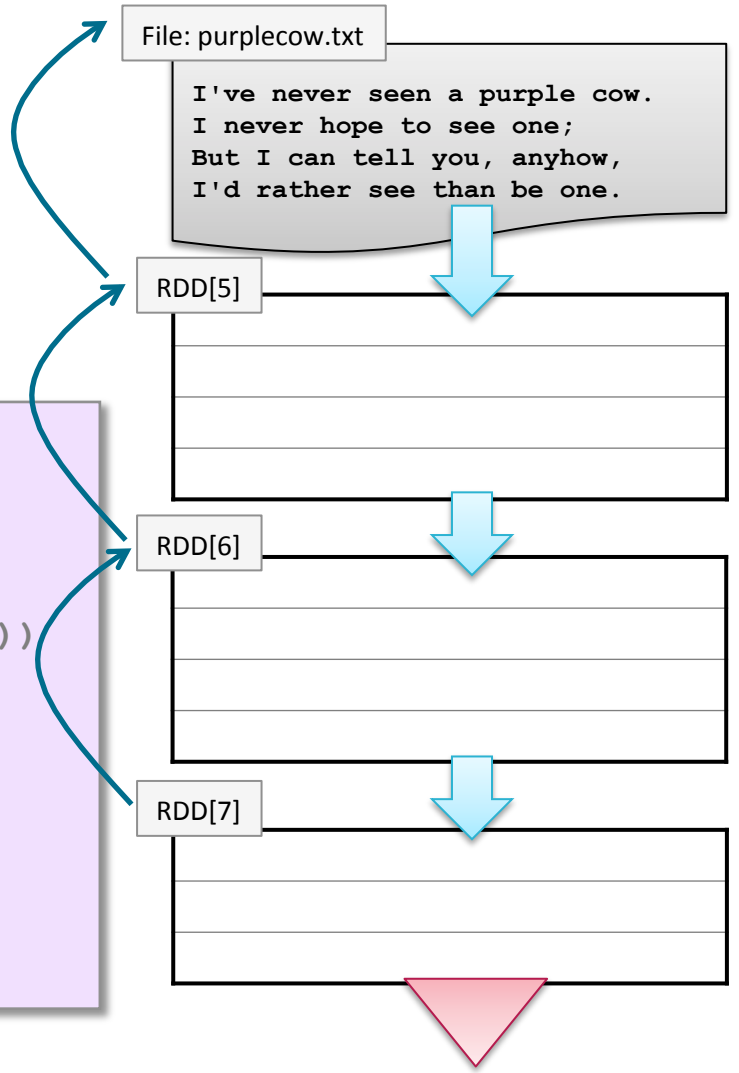
is exactly equivalent to

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

RDD Lineage and `toDebugString` (Scala)

- **Spark maintains each RDD's *lineage* – the previous RDDs on which it depends**
- **Use `toDebugString` to view the lineage of an RDD**

```
> val mydata_filt =
    sc.textFile("purplecow.txt").
    map(line => line.toUpperCase()).
    filter(line => line.startsWith("I"))
> mydata_filt.toDebugString

(2) FilteredRDD[7] at filter …
 |  MappedRDD[6] at map …
 |  purplecow.txt MappedRDD[5] …
 |  purplecow.txt HadoopRDD[4] …
```



RDD Lineage and `toDebugString` (Python)

- **`toDebugString` output is not displayed as nicely in Python**

```
> mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at …\n |  purplecow.txt MappedRDD[7] at textFile
at …[]\n |  purplecow.txt HadoopRDD[6] at textFile at …[]
```

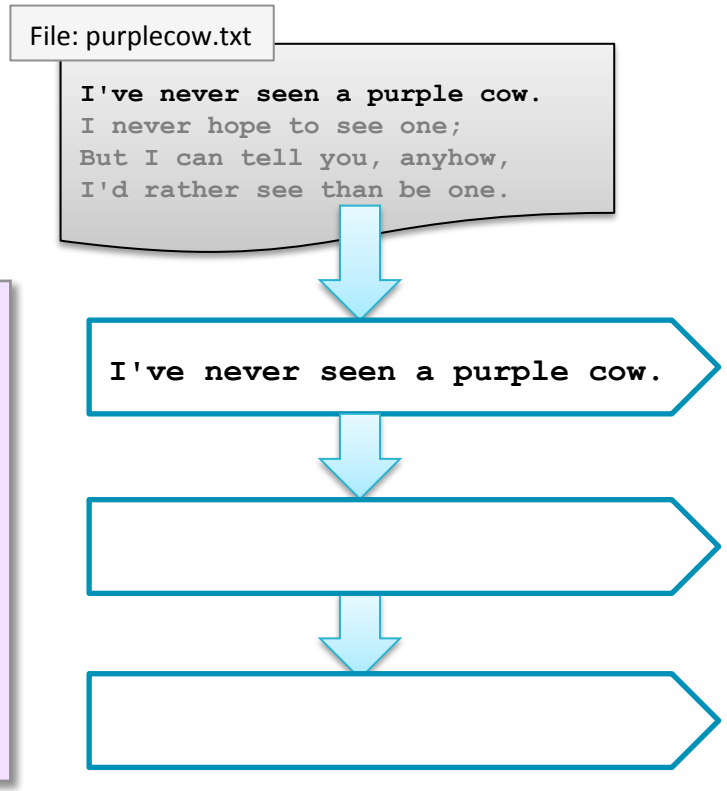
- **Use `print` for prettier output**

```
> print mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at …
 |  purplecow.txt MappedRDD[7] at textFile at …
 |  purplecow.txt HadoopRDD[6] at textFile at …
```

Pipelining (1)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

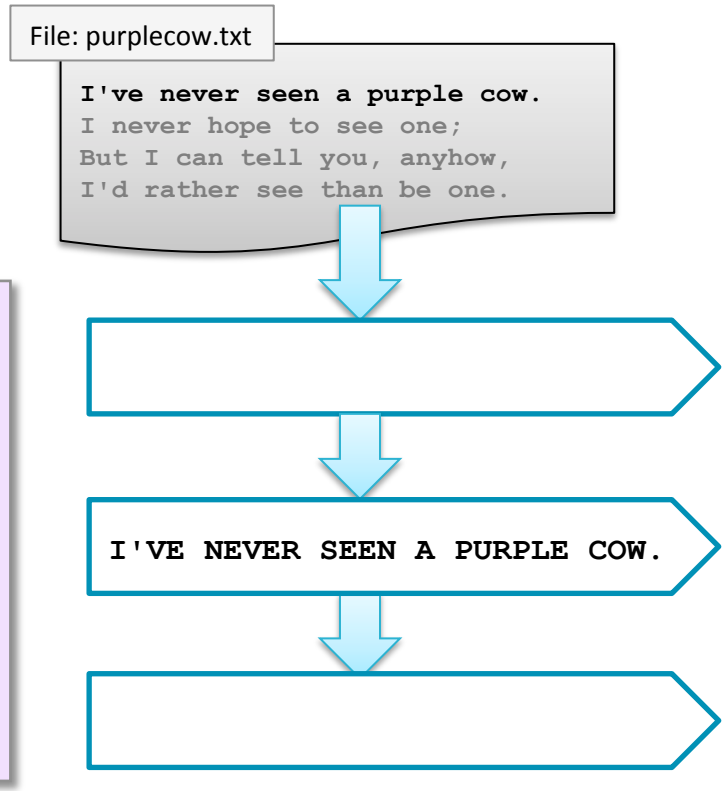
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (2)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

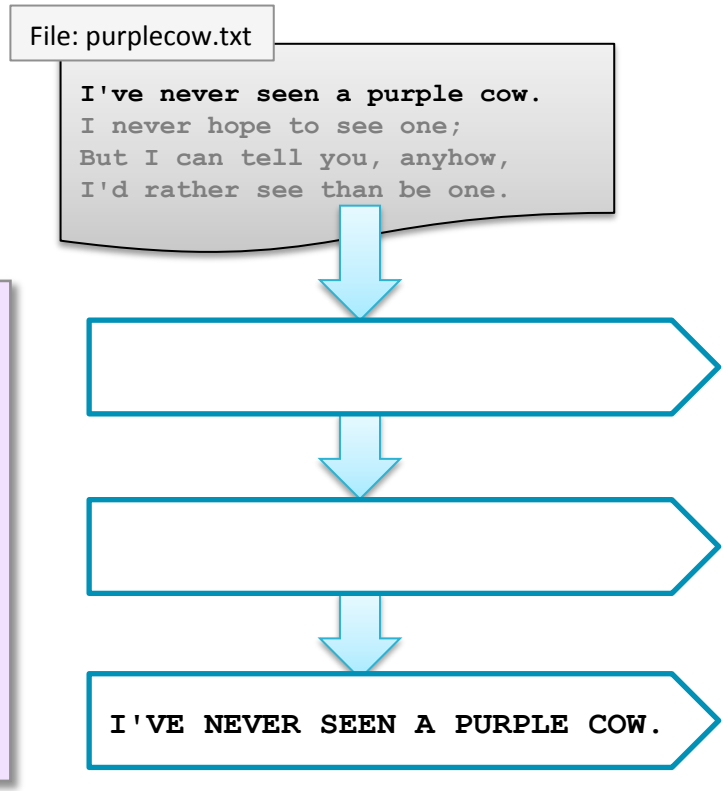
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (3)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

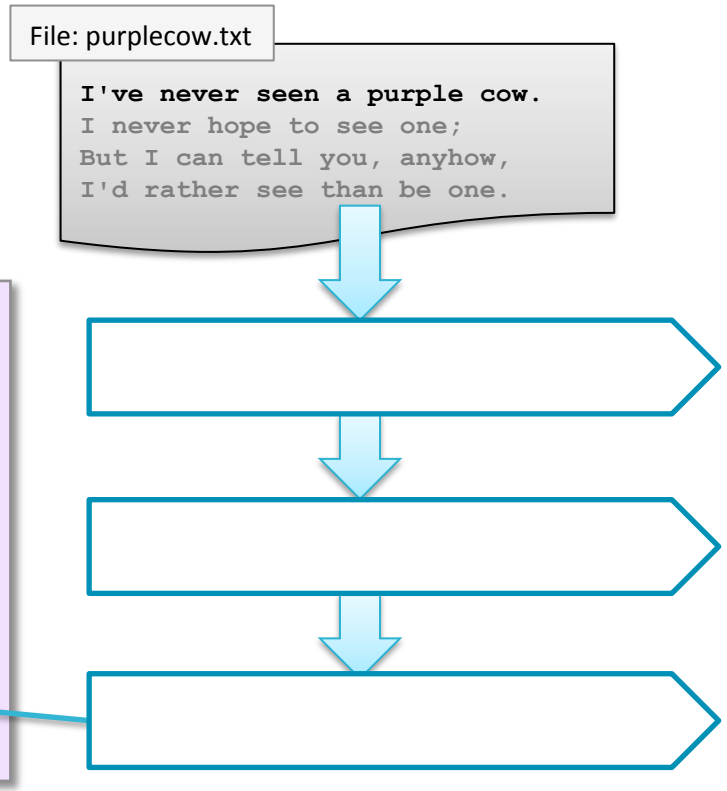
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (4)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

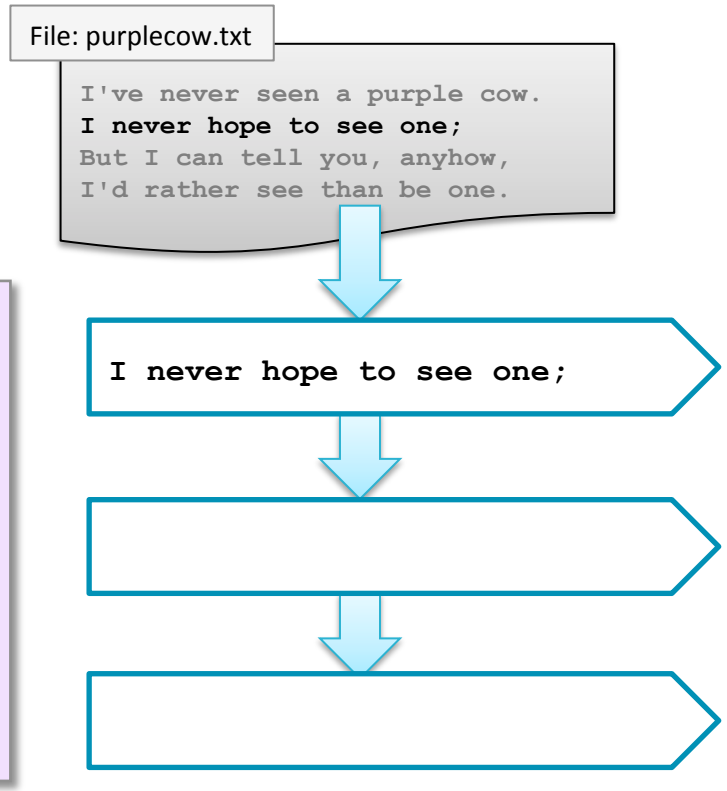
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (5)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

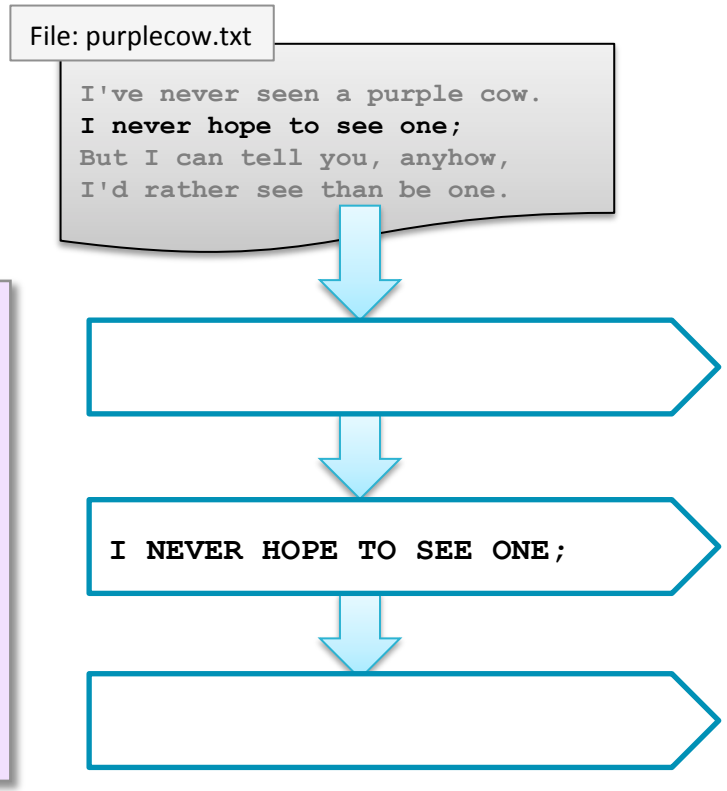
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (6)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

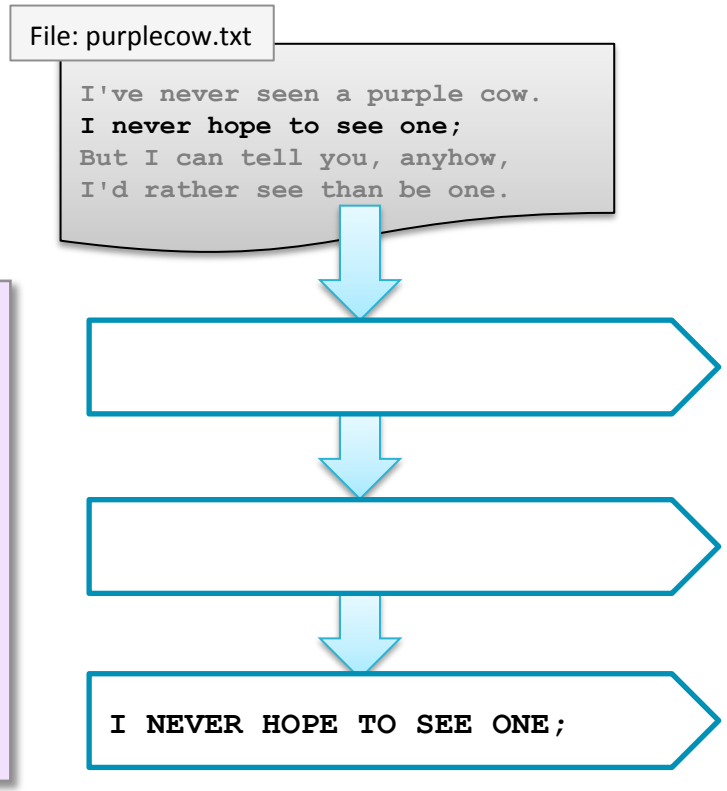
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (7)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

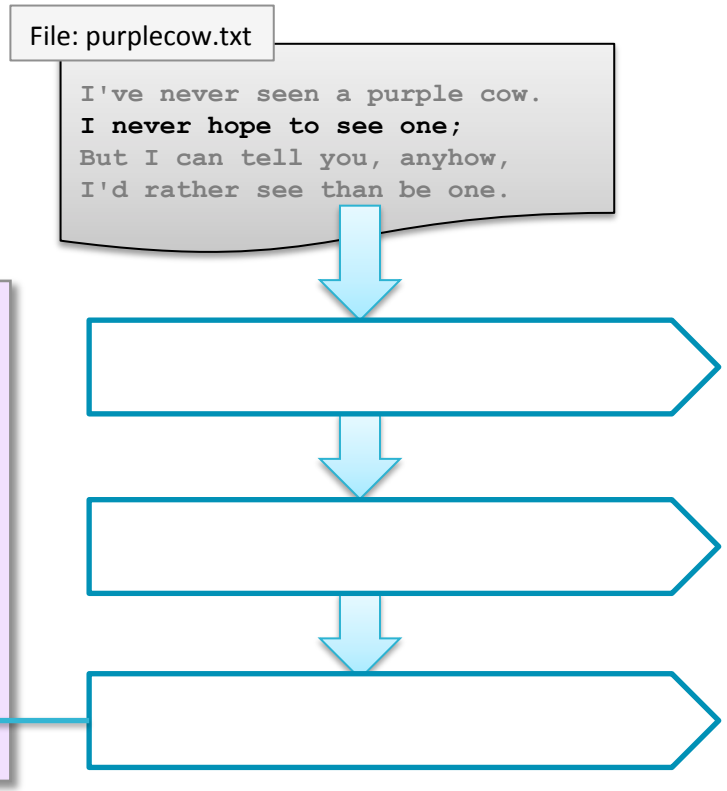
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (8)

- **When possible, Spark will perform sequences of transformations by row so no data is stored**

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```



Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- **Functional Programming in Spark**
- Conclusion
- Hands-On Exercises

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
    - No state or side effects
- **Key concepts**
  - Passing functions as input to other functions
  - Anonymous functions

Passing Functions as Parameters

- **Many RDD operations take functions as parameters**
- **Pseudocode for the RDD map operation**
  - Applies function **`fn`** to each record in the RDD

```
RDD {
    map(fn(x)) {
        foreach record in rdd
        emit fn(record)
    }
}
```

Example: Passing Named Functions

- **Python**

```
> def toUpper(s):
     return s.upper()
> mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

- **Scala**

```
> def toUpper(s: String): String =
    { s.toUpperCase }
> val mydata = sc.textFile("purplecow.txt")
> mydata.map(toUpper).take(2)
```

Anonymous Functions

- **Functions defined in-line without an identifier**
  - Best for short, one-off functions
- **Supported in many programming languages**
  - Python: `lambda x: ...`
  - Scala: `x => ...`
  - Java 8: `x -> ...`

Example: Passing Anonymous Functions

- **Python:**

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

- **Scala:**

```
> mydata.map(line => line.toUpperCase()).take(2)
```

OR

```
> mydata.map(_.toUpperCase()).take(2)
```

Scala allows anonymous parameters using underscore (_)

Example: Java

Java 7

```
...
  JavaRDD<String> lines = sc.textFile("file");
  JavaRDD<String> lines_uc = lines.map(
    new MapFunction<String, String>() {
      public String call(String line) {
        return line.toUpperCase();
      }
    });
...
```

Java 8

```
...
 JavaRDD<String> lines = sc.textFile("file");
 JavaRDD<String> lines_uc = lines.map(
    line -> line.toUpperCase());
...
```

Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming With Spark
- **Conclusion**
- Hands-On Exercises

Essential Points

- **Spark can be used interactively via the Spark Shell**
  - Python or Scala
  - Writing non-interactive Spark applications will be covered later
- **RDDs (Resilient Distributed Datasets) are a key concept in Spark**
- **RDD Operations**
  - Transformations create a new RDD based on an existing one
  - Actions return a value from an RDD
- **Lazy Execution**
  - Transformations are not executed until required by an action
- **Spark uses functional programming**
  - Passing functions as parameters
  - Anonymous functions in supported languages (Python and Scala)

Chapter Topics

Spark Basics | **Distributed Data Processing with Spark**

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming With Spark
- Conclusion
- **Hands-On Exercises**

Introduction to Spark Exercises: Pick Your Language

- **Your choice: Python or Scala**
  - For the Spark-based exercises in this course, you may choose to work with either Python or Scala
- **Solution and example files**
  - `.pyspark` – Python shell commands
  - `.scalaspark` – Scala shell commands
  - `.py` – complete Python Spark applications
  - `.scala` – complete Scala Spark applications

Hands-On Exercises

- **Now, please do the following three Hands-On Exercises**
  1. *View the Spark Documentation*
     - Familiarize yourself with the Spark documentation; you will refer to this documentation frequently during the course
  2. *Explore RDDs Using the Spark Shell*
     - Follow the instructions for either the Python or Scala shell
  3. *Use RDDs to Transform a Dataset*
     - Explore Loudacre web log files
- **Please refer to your Hands-On Exercise Manual**

cloudera®

Working With RDDs in Spark

Chapter 11

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ **Working with RDDs in Spark** ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ Conclusion	Course Conclusion

Working With RDDs

In this chapter you will learn

- **How RDDs are created from files or data in memory**
- **How to handle file formats with multi-line records**
- **How to use some additional operations on RDDs**

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- **Creating RDDs**
- Other General RDD Operations
- Conclusion
- Hands-On Exercise: Process Data Files with Spark

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
  - Double RDDs
    - RDDs consisting of numeric data

Creating RDDs From Collections

- **You can create RDDs from collections instead of files**
  - `sc.parallelize(`*`collection`*`)`

```
> myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2)
['Alice', 'Carlos']
```

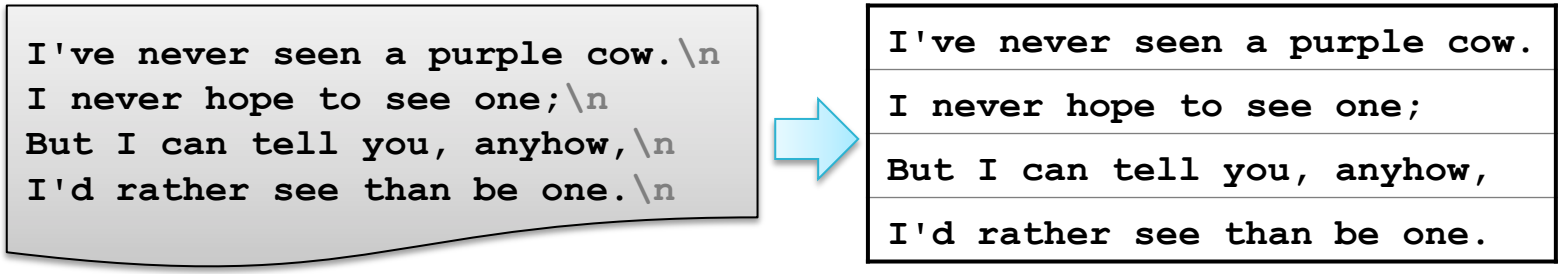
- **Useful when**
  - Testing
  - Generating data programmatically
  - Integrating

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**
- **What about other formats?**

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":"123"} )
(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"]
```

```
> 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))
```

Output:

```
Fred
Barney
```

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- **Other General RDD Operations**
- Conclusion
- Hands-On Exercise: Process Data Files with Spark

Some Other General RDD Operations

- **Single-RDD Transformations**
  - **`flatMap`** – maps one element in the base RDD to multiple elements
  - **`distinct`** – filter out duplicates
  - **`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

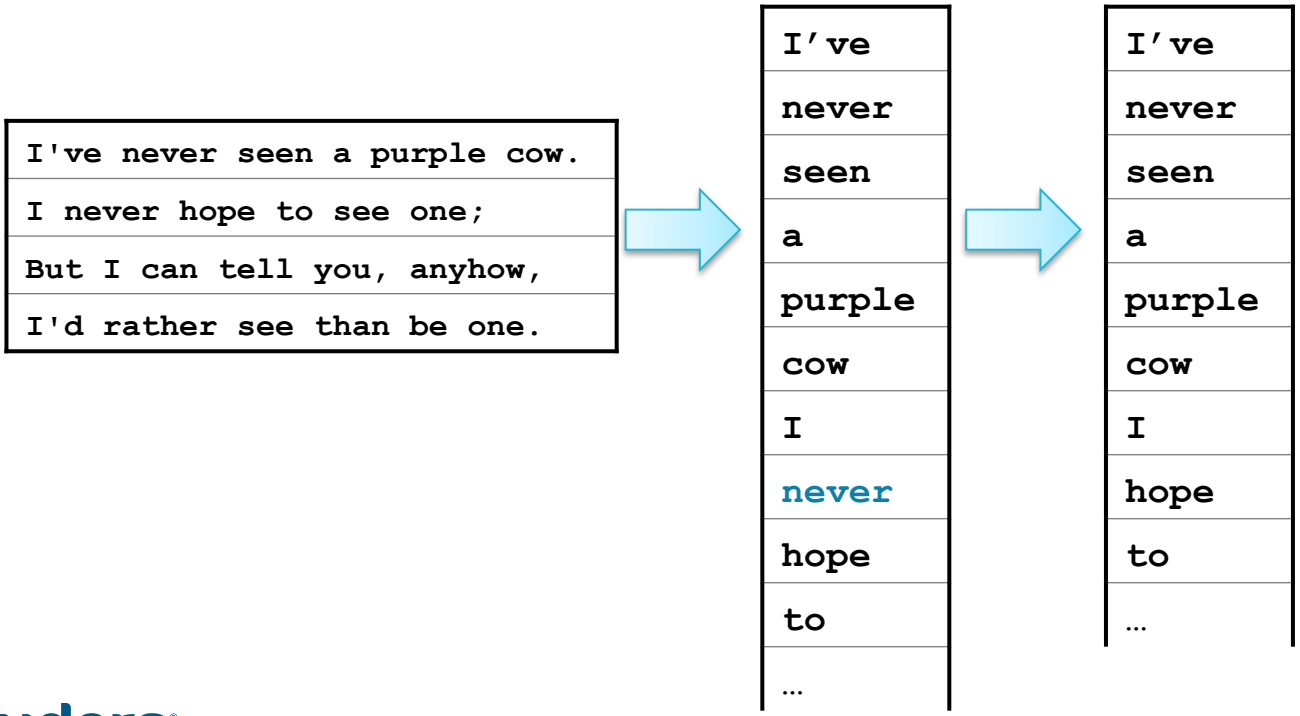
Example: `flatMap` and `distinct`

Python

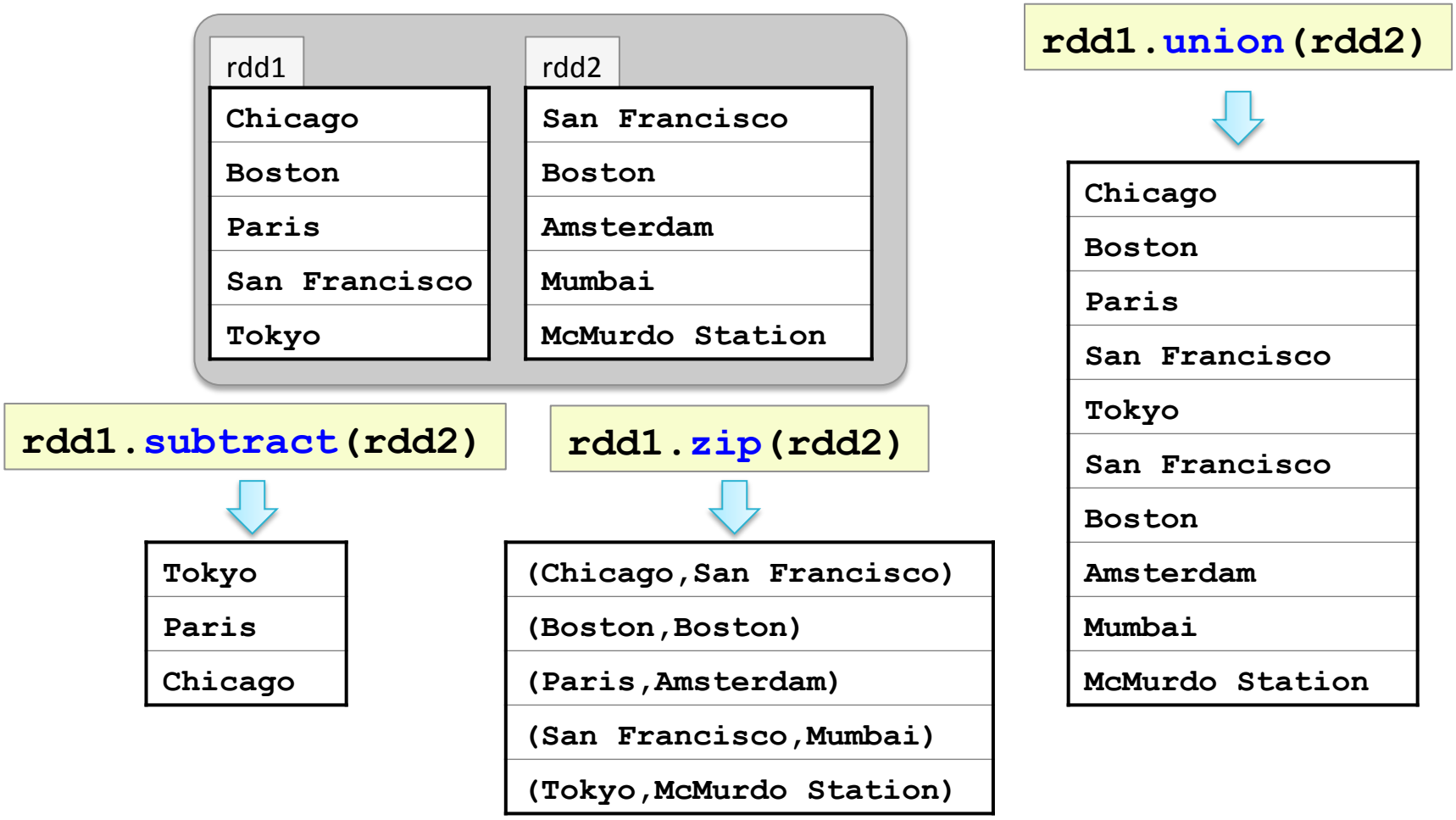
```
> sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .distinct()
```

Scala

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



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`**

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- Other General RDD Operations
- **Conclusion**
- Hands-On Exercise: Process Data Files with Spark

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**

Chapter Topics

Working With RDDs in Spark | **Distributed Data Processing with Spark**

- Creating RDDs
- Other General RDD Operations
- Conclusion
- **Hands-On Exercise: Process Data Files with Spark**

Hands-On Exercise: Process Data Files with Spark

- **In this exercise you will**
  - Process a set of XML files using **`wholeTextFiles`**
  - Reformat a dataset to standardize format (bonus)
- **Please refer to the Hands-On Exercise Manual**

cloudera®

Aggregating Data with Pair RDDs

Chapter 12

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ Data File Partitioning	Importing and Modeling Structured Data
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ **Aggregating Data with Pair RDDs** ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	**Distributed Data Processing with Spark**
▪ 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**

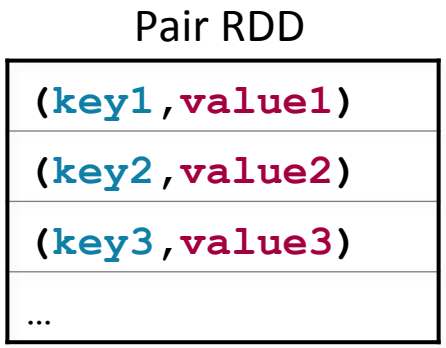
Chapter Topics

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

- **Key-Value Pair RDDs**
- Map-Reduce
- Other Pair RDD Operations
- Conclusion
- Hands-On Exercise: 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**

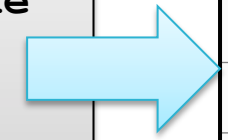
Python

```
> 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))
```

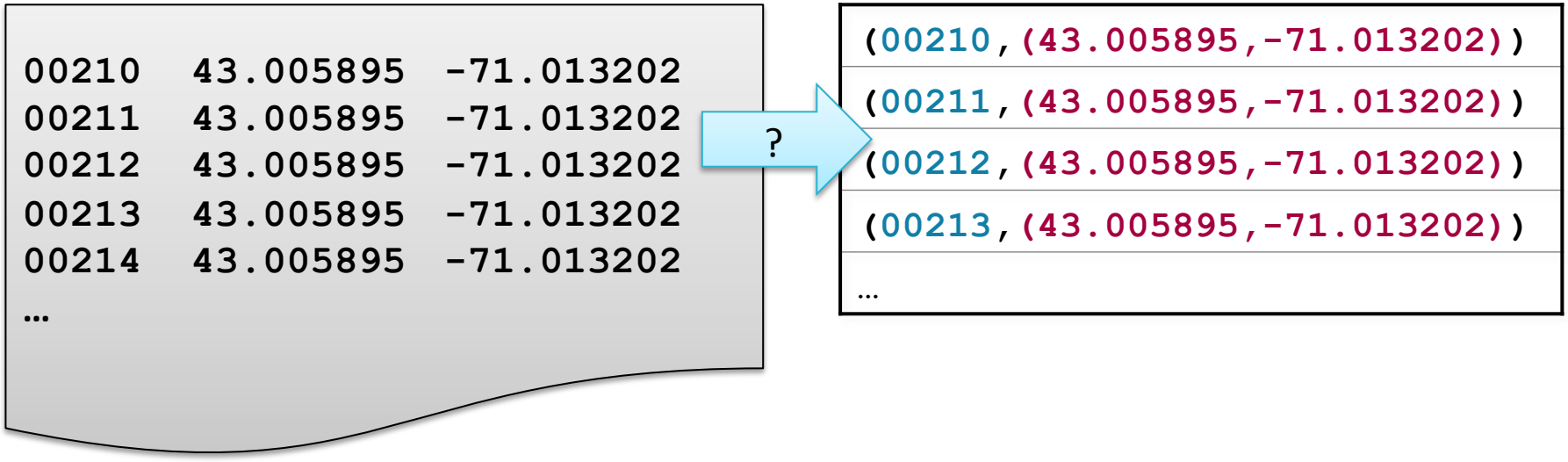
User ID

```
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)

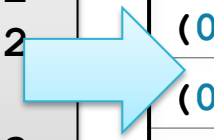


Answer 1: Pairs With Complex Values

```
> 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))))
```

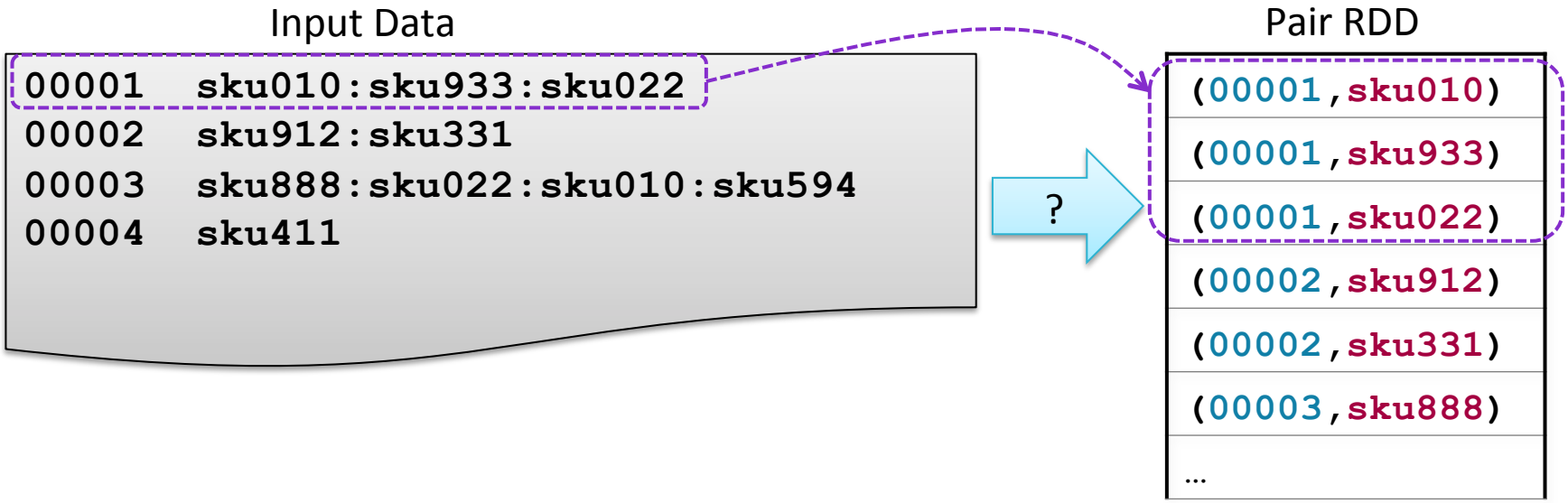
```
00210   43.005895   -71.013202
01014   42.170731   -72.604842
01062   42.324232   -72.67915
01263   42.3929     -73.228483
…
```



```
(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))
```

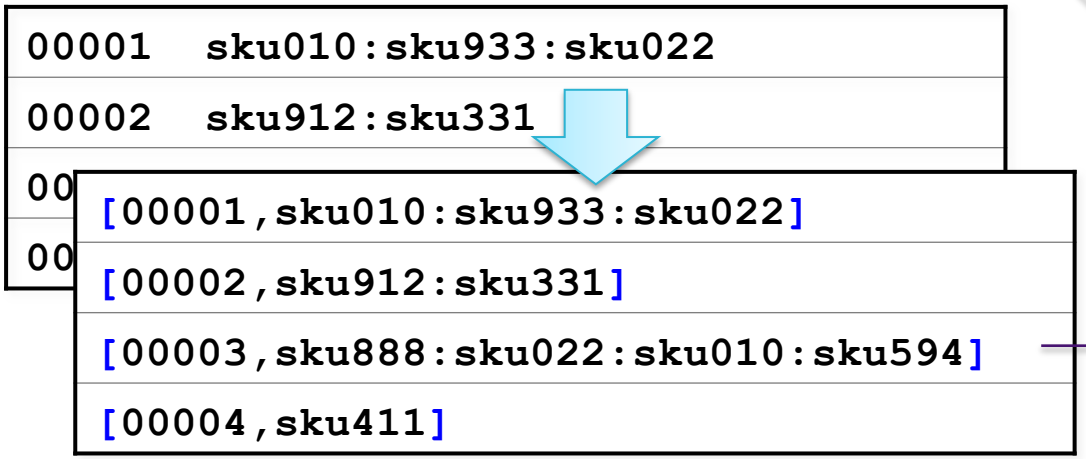
Answer 2: Mapping Single Rows to Multiple Pairs (1)

```
> 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)

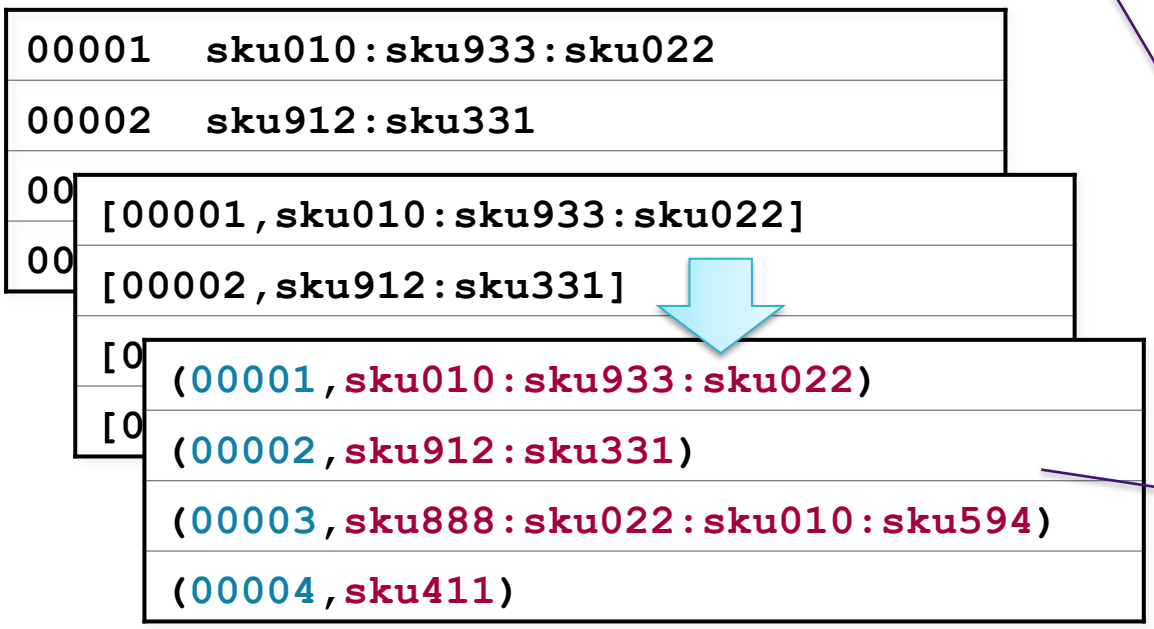
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t'))
```



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

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

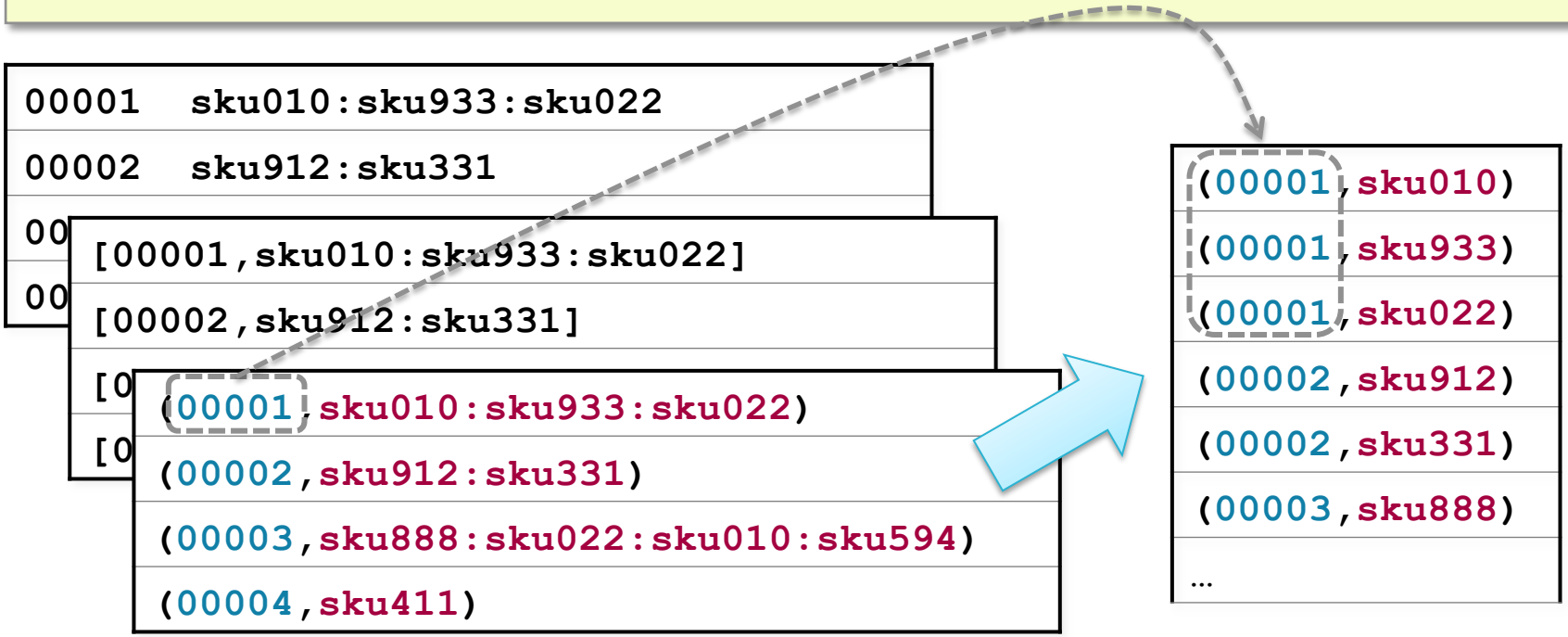
```
> sc.textFile(file) \
  .map(lambda line: line.split('\t')) \
  .map(lambda fields: (fields[0],fields[1]))
```



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
- Hands-On Exercise: 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

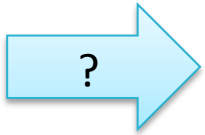
Map-Reduce in Spark

- **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

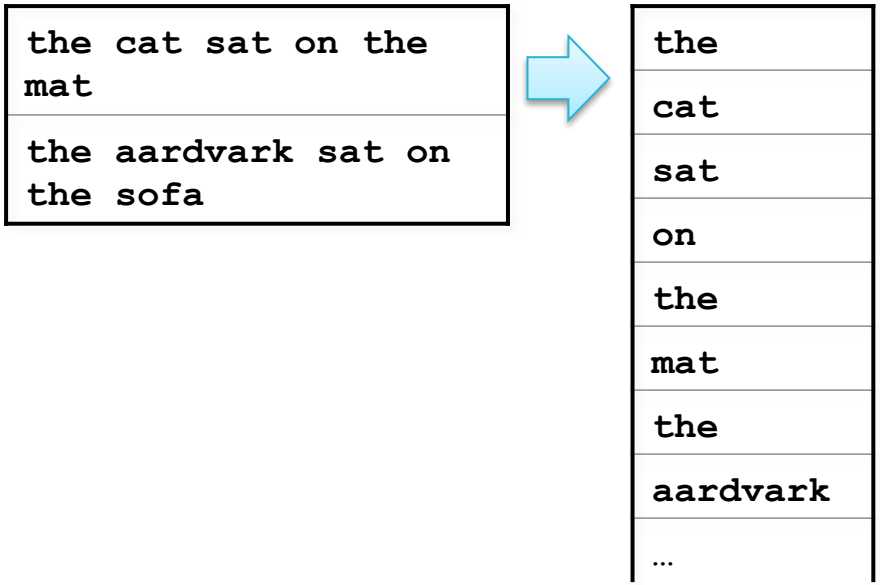
Example: Word Count (1)

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

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

Example: Word Count (2)

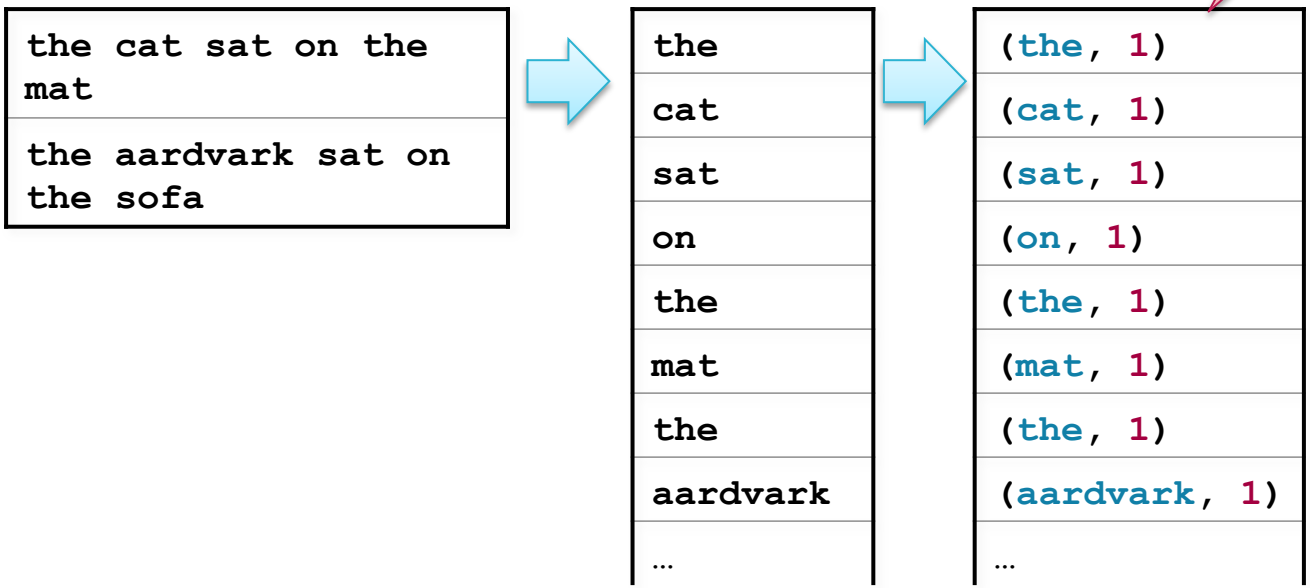
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



Example: Word Count (3)

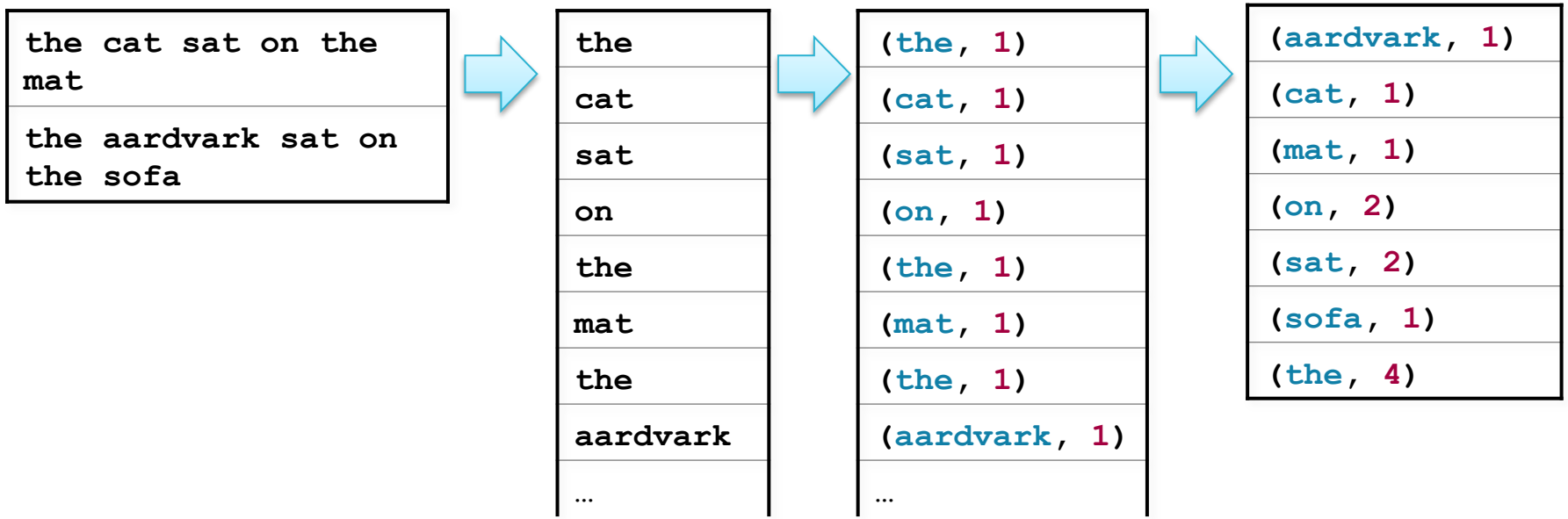
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1))
```

Key-Value Pairs



Example: Word Count (4)

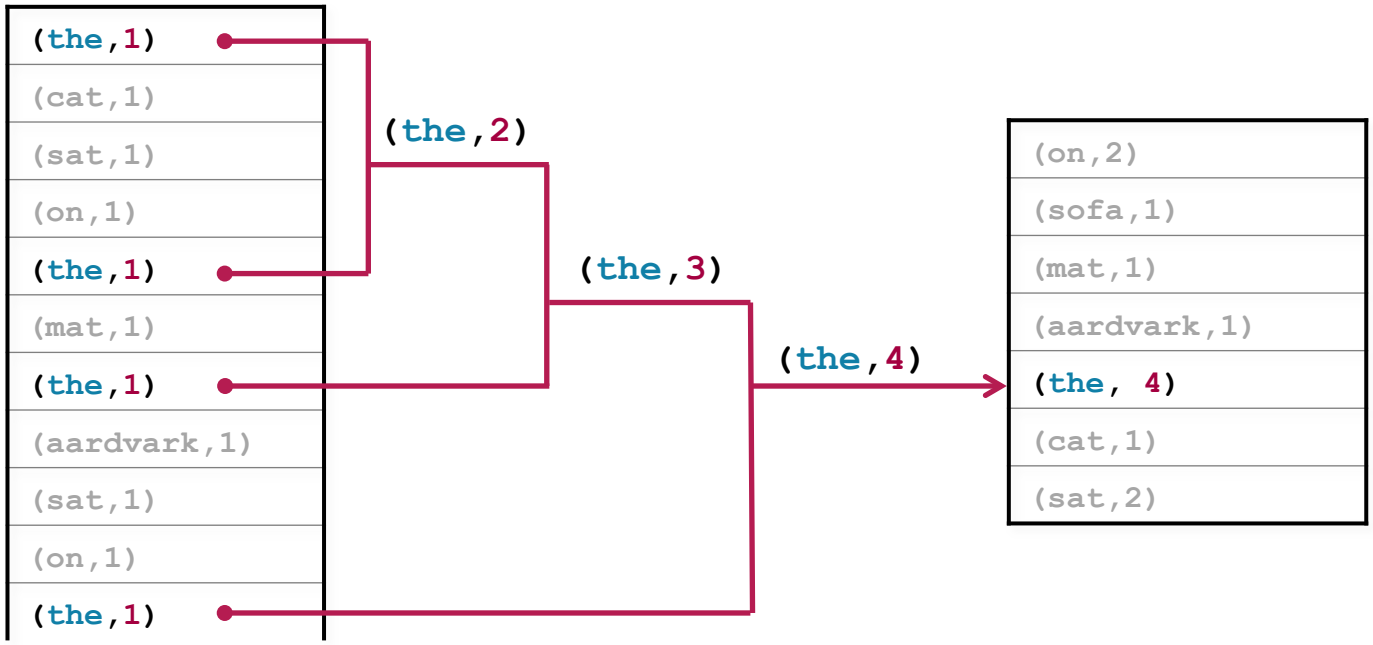
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey (1)

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

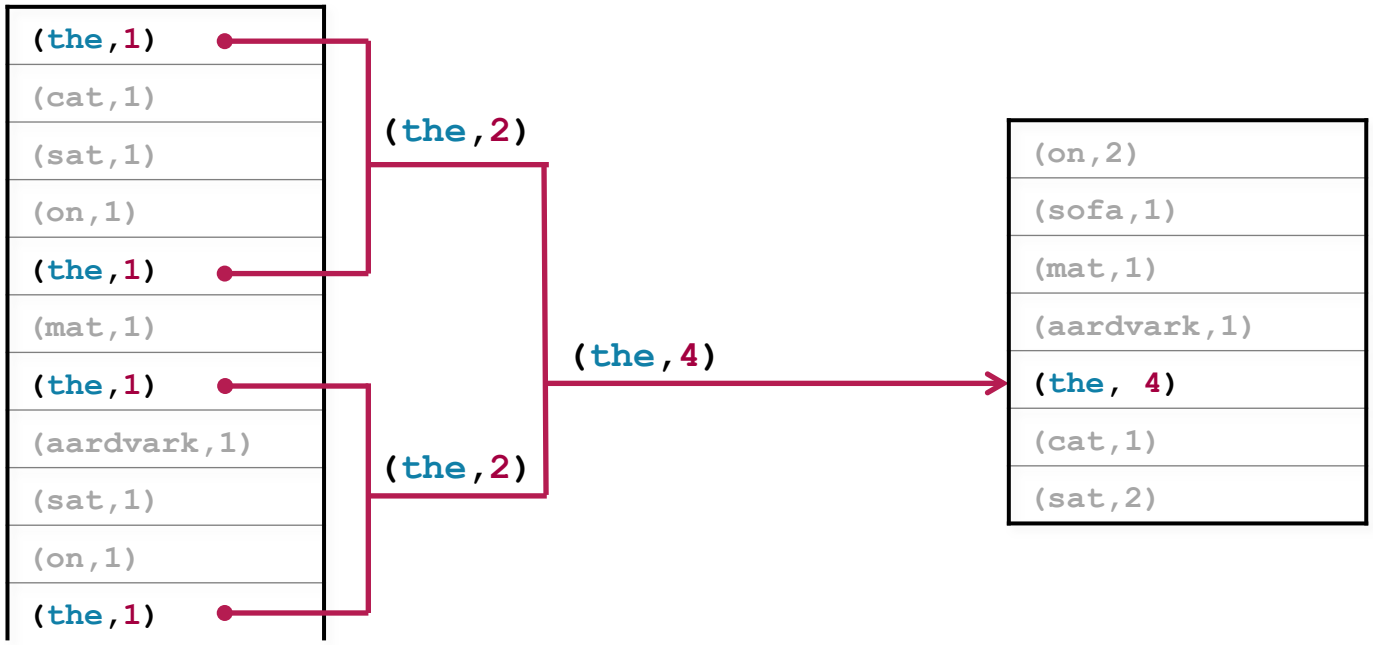
```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



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?

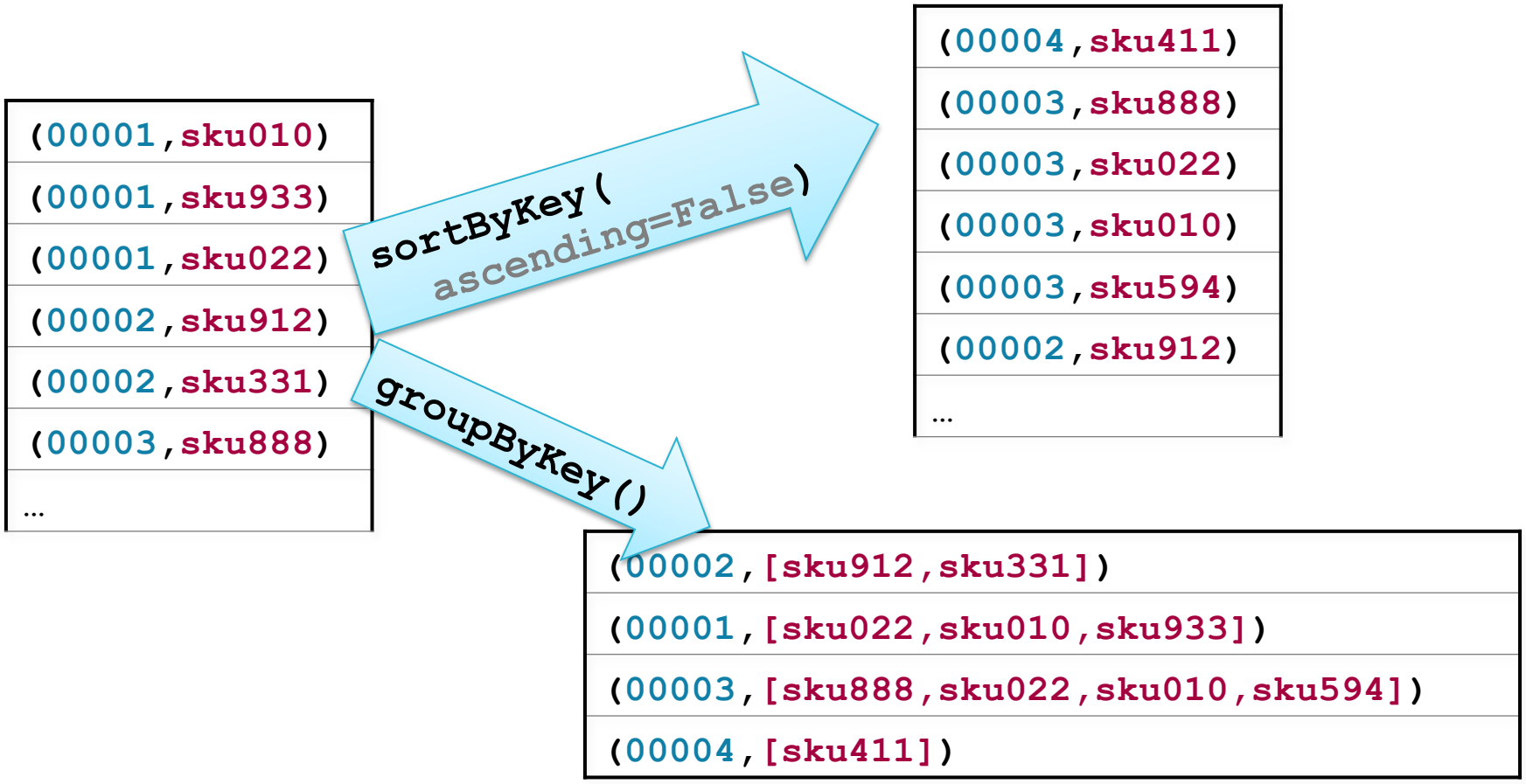
- **Word count is challenging over massive amounts of data**
  - Using a single compute node would be too time-consuming
  - Number of unique words could exceed available memory
- **Statistics are often simple aggregate functions**
  - Distributive in nature
  - e.g., max, min, sum, count
- **Map-reduce 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

Pair RDD Operations

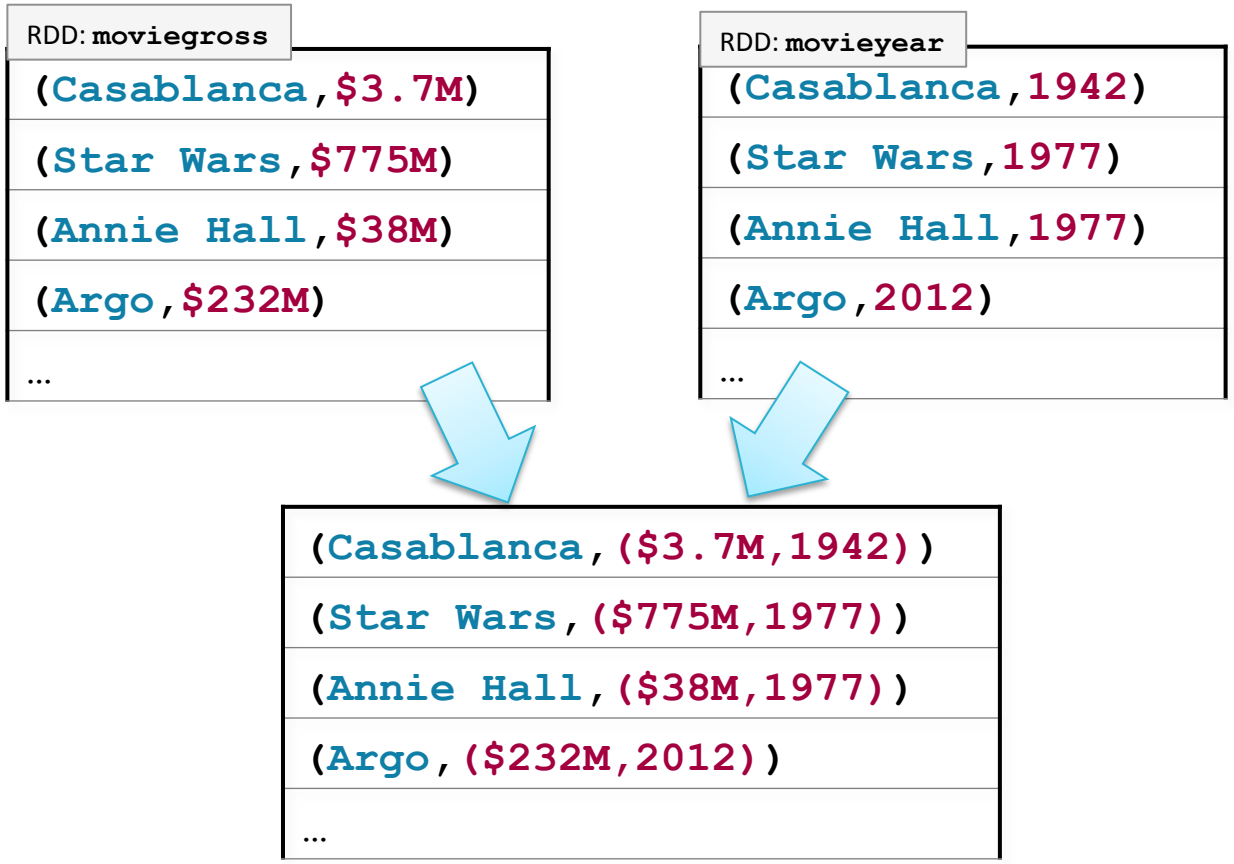
- **In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs**
- **Examples**
  - **`countByKey`** – return a map with the count of occurrences of each key
  - **`groupByKey`** – group all the values for each key in an RDD
  - **`sortByKey`** – sort in ascending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

Example: Pair RDD Operations



Example: Joining by Key

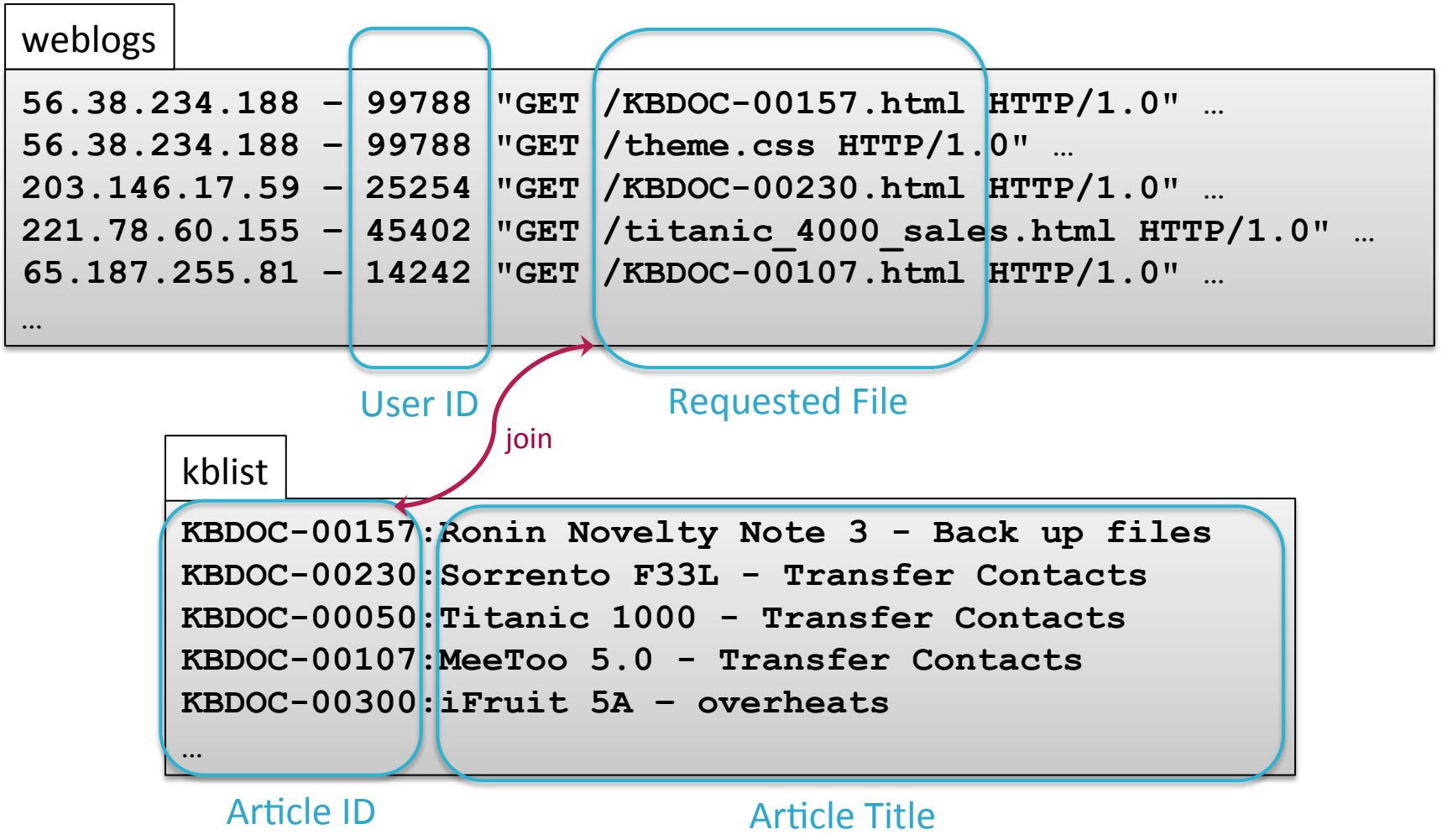
```
> movies = moviegross.join(movieyear)
```



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)



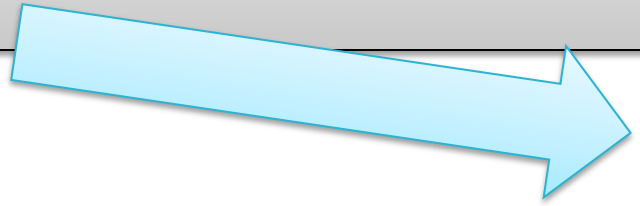
Example: Join Web Log With Knowledge Base Articles (2)

- **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]))
```

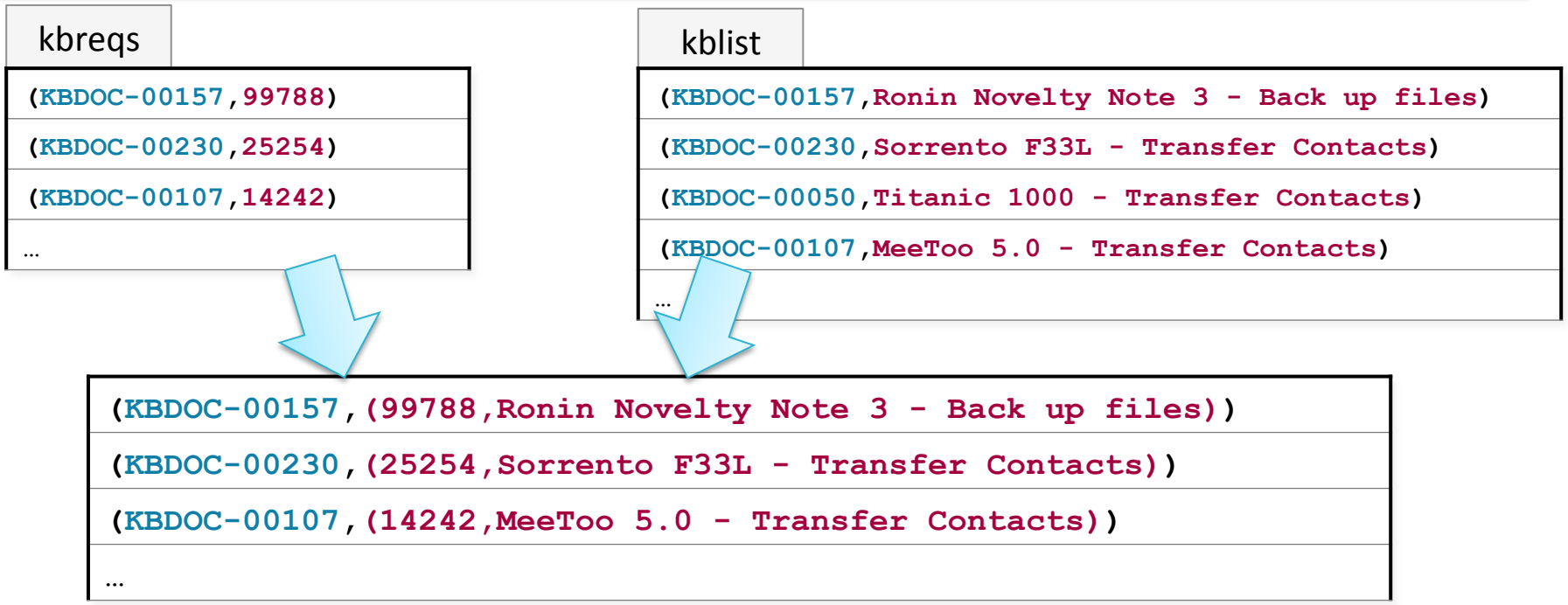
```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A – overheats
…
```

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

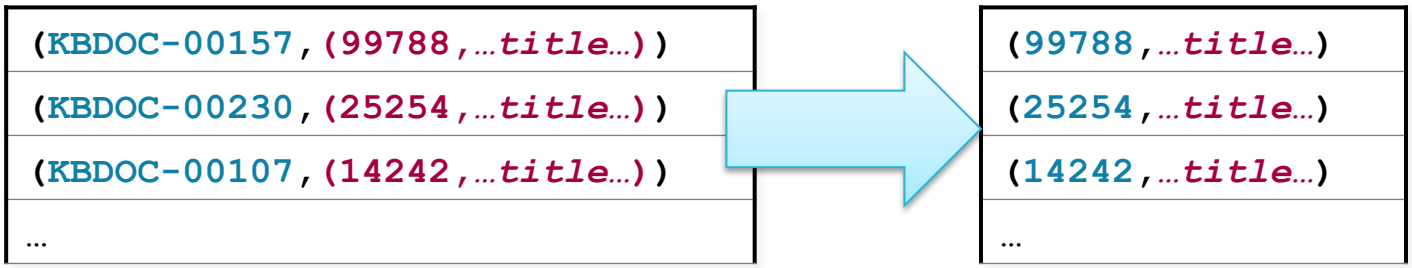
```
> map(lambda (docid,(userid,title)): (userid,title))
```

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

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

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

Hands-On Exercise: Use Pair RDDs to Join Two Datasets

- **In this exercise you will**
  - Continue exploring web server log files using key-value Pair RDDs
  - Join log data with user account data
- **Please refer to the Hands-On Exercise Manual**