

Class 3: Spark Overview

New York University

Summer 2017



1. **Review**
2. Distributed Processing Background
3. Spark Overview
4. Spark Basics
5. Using Spark RDDs

1. Review
2. Distributed Processing Background
3. Spark Overview
4. Spark Basics
5. Using Spark RDDs

How can our programs scale?

- Write solutions using tools that scale
- Parallelize code
- Choose an expandable platform

How can our programs scale? (continued)

- Code should scale with the size of the data, without needing to be re-written, re-tested, or re-deployed
- The platform should scale with the size of the data, without needing to switch platforms or move data

What kinds of tools exist?

- There is a large selection of tools that are mostly open source and together provide a framework that scales
- Apache Hadoop and Apache Spark frameworks facilitate scalable storage and compute power
 - Popular
 - Deployed in production
 - Low cost

Why Hadoop?

- Extensible
- Scales linearly
- Automatically parallelizes code
- Autonomic

Why Hadoop? (continued)

- Designed-in fault tolerance
 - Failures anticipated
- This design makes Hadoop an excellent platform for long-running programs

How do we succeed at developing analytics for Big Data?

1. Pre-processing
2. Profiling
3. Iteration

1. Pre-processing

- We develop code to clean and format data
 - Big Data is too big to be visually scanned for anomalous data values
- We need a tool that can correct glitchy data
 - We either write such a tool, or use existing data exploration tools
 - Tableau is a Big Data visualization tool helpful for exploring Big Data

1. Pre-processing
2. Profiling
3. Iteration

2. Profiling

We develop tools to perform profiling because

- Big Data is too big to be visually scanned to identify the range of values in any given column
- We need to understand the data we have, especially since we are often using open data, or data collected/curated by others

How can we learn about the data?

- Get the range of values in a given column
- Identify the types for each column
- This is an especially important step if you are using data that you did not collect yourself

1. Pre-processing
- 2. Profiling**
3. Iteration

3. Iteration

- Modeling and analysis may require multiple attempts
- Some algorithms require iteration in order for results to converge
- Satisfy service level agreements (SLAs)

1. Pre-processing
2. Profiling
3. Iteration

1. Review
2. Distributed Processing Background
3. **Spark Overview**
4. Spark Basics
5. Using Spark RDDs

What is Apache Spark?

- Apache Spark originated at UC Berkeley AMPLab and was contributed to the Apache Software Foundation
- Apache Spark first became available in *2009*

Apache Spark is -

- An open source *distributed processing framework*
 - Runs your program on servers in the Spark cluster
 - Elegant model for writing programs

Comparing Spark and MapReduce

- Spark and MapReduce frameworks
 - Parallelize code
 - Run code in a cluster
- Programs written in either framework can process huge amounts of data efficiently
- Both frameworks scale linearly through the addition of more servers as datasets grow
- The underlying storage system is commonly HDFS
- Both frameworks provide fault tolerance

Comparing Spark and MapReduce (continued)

- Spark does not generate MapReduce jobs under the covers
- Spark can be used to implement a Map-Reduce style solution

Comparing Spark and MapReduce (continued)

- In general, Spark is easier to use than Hadoop MapReduce
 - Spark code is succinct
 - Requires little boilerplate code in programs compared to MapReduce
 - Supports SQL and Streaming

Why Spark?

Spark goes beyond MapReduce in three ways:

1. Flexibility
2. Expressiveness
3. Speed

1. Flexibility

- MapReduce executes a rigid set of stages:

map ➡ *shuffle sort* ➡ *reduce*

- Spark can pass the results of a map step directly to the user-specified next step in the workflow
- Spark supports iteration

- 1. Flexibility
- 2. Expressiveness
- 3. Speed

2. Expressiveness

- Spark provides a rich set of transformations
- Spark allows you to create complex processing workflows in a few lines of code

1. Flexibility
- 2. Expressiveness**
3. Speed

3. In-Memory Processing

- Spark can materialize data in memory at any point in the workflow
- MapReduce, on the other hand, writes intermediate results to disk, making it slow

1. Flexibility
2. Expressiveness
- 3. Speed**

- Spark improves productivity by providing a single language for
 - Cleaning and pre-processing data
 - Building models and experimenting
 - Building production applications
- Spark runs on top of the Java Virtual Machine (JVM), making it possible to leverage debugging tools built for the Java stack

***How can Scala
leverage a cluster?***

Scala Program

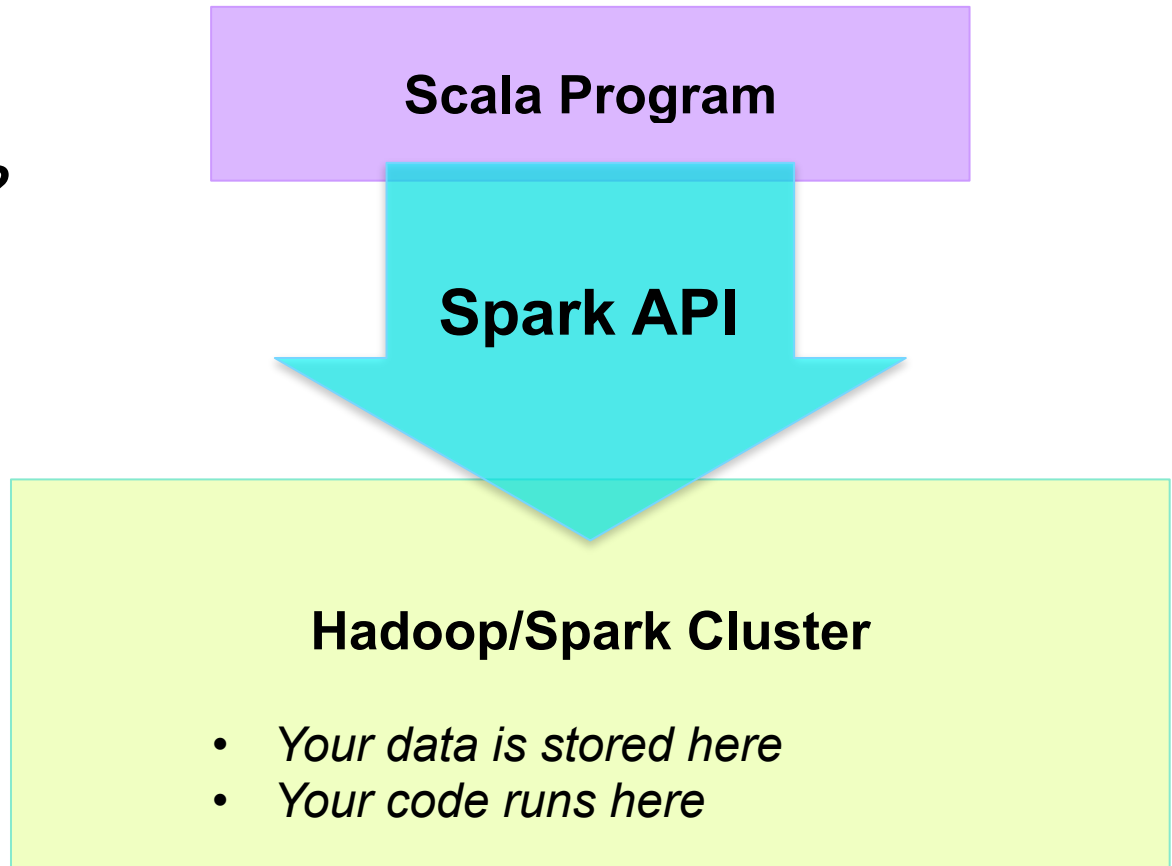
?

Hadoop/Spark Cluster

- *Your data is stored here*
- *Your code runs here*

How can Scala leverage a cluster?

- *Scala programs call the Spark API*
- *Spark API communicates with the cluster*



Spark provides a framework that supports writing code for both exploratory and operational analytics in Scala or Python

- Scala or Python code relies on the Spark API to run code in the cluster
- The Spark framework parallelizes programs and runs them in a distributed way to process data stored in the cluster

- Use Spark in the REPL for exploratory analytics
 - Test interactively, without going to an IDE
- Compile Spark code for operational analytics
 - Write production-grade analytics in Spark

1. Review
2. Distributed Processing Background
3. Spark Overview
4. **Spark Basics**
5. Using Spark RDDs

- **The Spark Shell provides interactive data exploration (REPL)**
 - **REPL: Read/Evaluate/Print Loop**
- **Every Spark application requires a Spark Context**
 - The main entry point to the Spark API
- **Spark Shell provides a preconfigured Spark Context called `sc`**

- **RDD (Resilient Distributed Dataset)**
 - Resilient
 - If data *in memory* is lost, it can be recreated
 - Distributed
 - Processed across the cluster
 - Dataset
 - Data is pulled from a file or created programmatically
- **RDDs are the fundamental unit of data in Spark**
 - **Much Spark programming consists of performing operations on RDDs**

- **Create an RDD from**
 - One or more files
 - Data in memory
 - Another RDD

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

4

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.

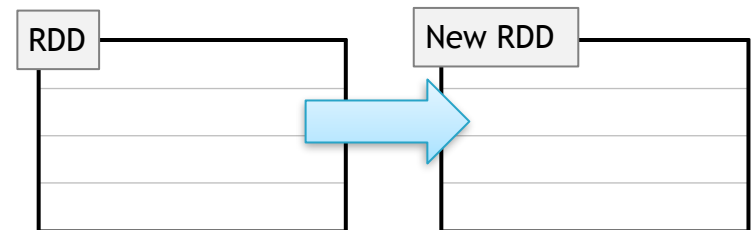
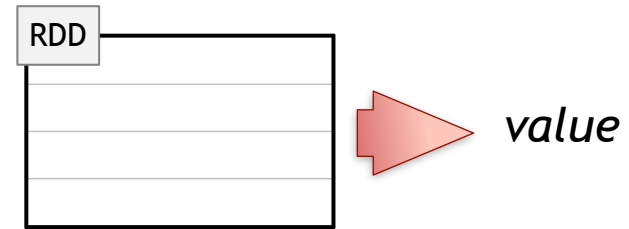


RDD: mydata

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.

■ Two types of RDD operations

- Action
 - Return a value
- Transformation
 - Defines a new RDD based on the current one



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

■ Common Spark actions

- `count()`
- `take(n)`
- `collect()`
- `saveAsTextFile(file`

- Spark action: `count()`

- Returns the number of elements in the RDD

```
> mydata.count()
```

```
4
```

```
> for (line <- mydata.take(2))  
  println(line)
```

```
I've never seen a purple cow.
```

```
I never hope to see one;
```

■ Common Spark actions

- **count()**
 - Returns the number of elements in the RDD
- **take(*n*)**
 - Returns an array of the first *n* elements
- **collect()**
 - Returns an array of all elements
- **saveAsTextFile(*file*)**
 - Saves data to text file(s)

- A transformation creates a new RDD from an existing one
- 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
- Data in RDDs is not processed until an *action* is performed

- The `map()` and `filter()` functions will yield the following results after an action is executed

```
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.
```



```
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.
```



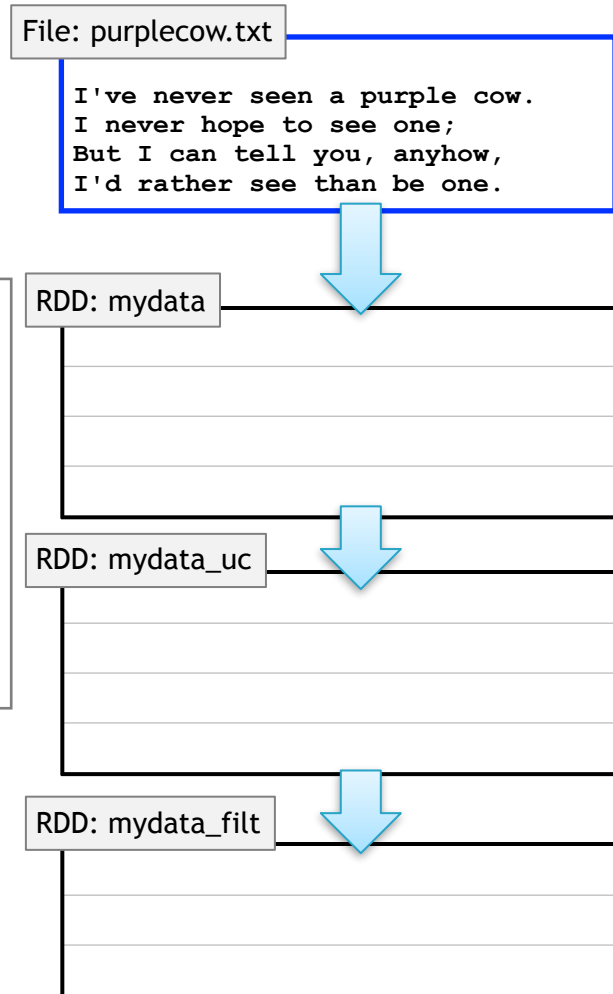
```
I'VE NEVER SEEN A PURPLE COW.  
I NEVER HOPE TO SEE ONE;  
I'D RATHER SEE THAN BE ONE.
```

```
map(line => line.toUpperCase)
```

```
filter(line => line.startsWith('I'))
```

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

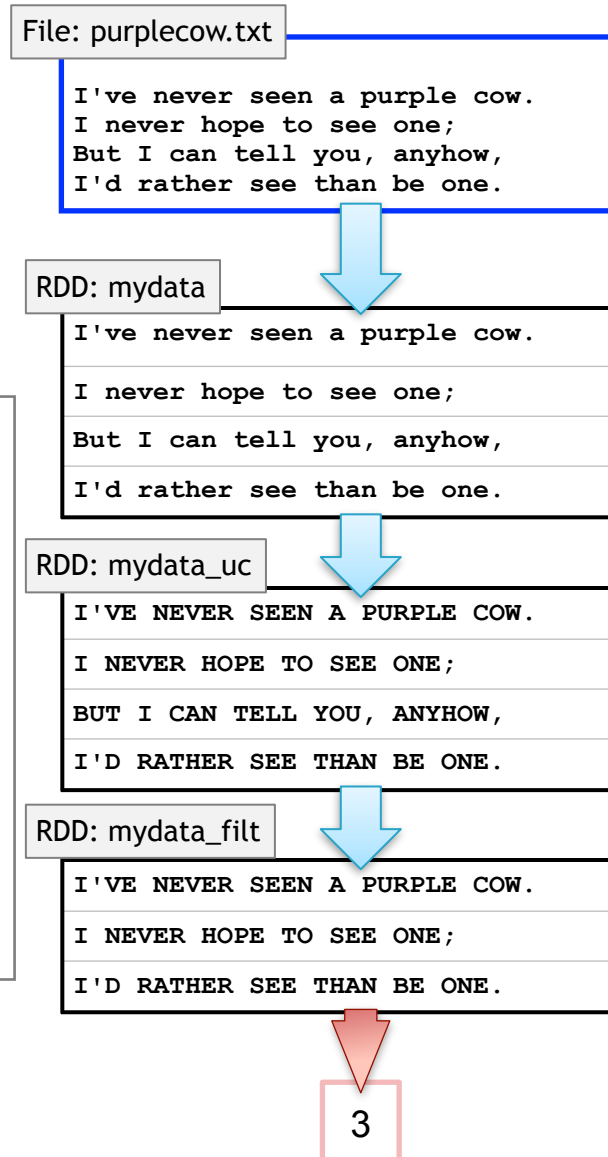
```
> val mydata = sc.textFile("purplecow.txt")  
  
> val mydata_uc = mydata.map(line =>  
  line.toUpperCase())  
  
> val mydata_filt = mydata_uc.filter(line  
  => line.startsWith("I"))
```



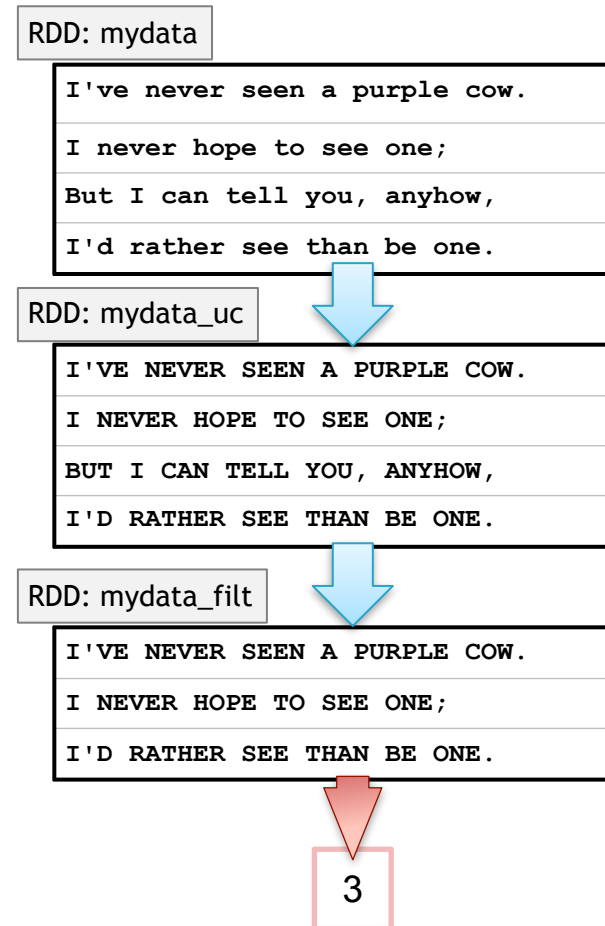
- Data in RDDs is not processed until an *action* is performed
- This is called lazy execution
 - Allows the Spark framework to optimize processing

```
> 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



- Each RDD generated is not available in future steps
 - Data is not automatically cached
- Invoking an action, even a simple `count()` action, will cause the three transformations to be *re-run*



- Output from each transformation can be assigned to a new variable

```
> 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
```

... or, transformations can be chained as follows ...

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


- In the following example, we do not store to RDDs explicitly
 - An RDD is implicitly created for each of these transformations
- We have no way of accessing implicitly created RDDs
- Spark maintains information about all RDDs - whether explicitly, or implicitly, created

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

```
3
```

*** *Important Point:* We can't chain a transformation after an action, because actions output *values*, not RDDs

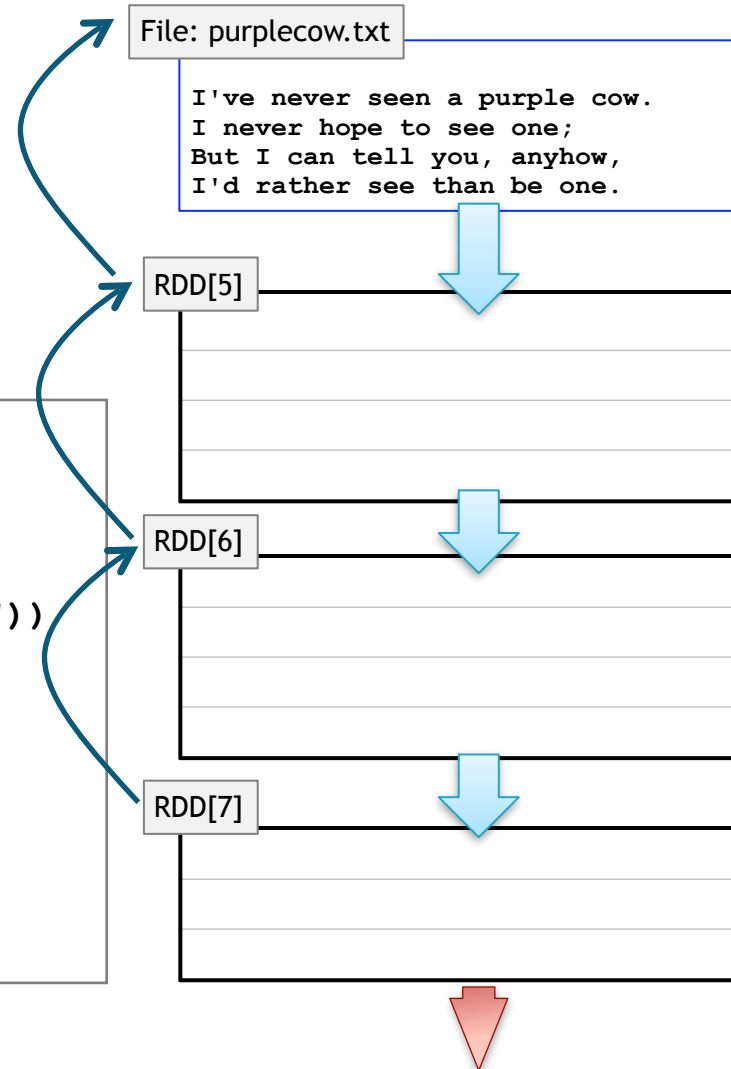
```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).  
  filter(line => line.startsWith("I")).take(2).toLowerCase()
```



- 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] ...
```



- 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)
```

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.



I've never seen a purple cow.



```
> 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)
```

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.



I'VE NEVER SEEN A PURPLE COW.



```
> 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)
```

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.

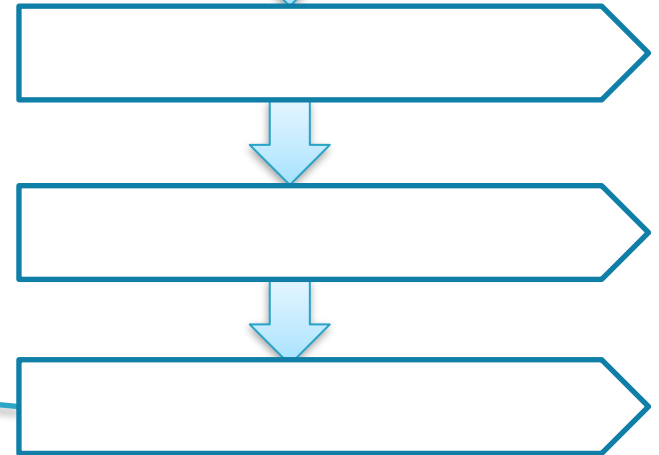


I'VE NEVER SEEN A PURPLE COW.

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.

```
> 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.
```



```
> 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.
```

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.



I never hope to see one;



```
> 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.
```

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.



I NEVER HOPE TO SEE ONE;



```
> 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.
```

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.



I NEVER HOPE TO SEE ONE;

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

```
> 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;
```

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.



- RDD data is not stored in memory between operations
 - The entire data set is *not* read in and buffered from step to step
 - RDD data is processed one RDD element at a time
- The whole dataset is stored (in memory or on disk) only when
 - Persisted explicitly (using **`persist`** or **`cache`**)
 - Required for shuffle (e.g. for a reduce operation)

- **Spark depends heavily on the concepts of *functional programming***
 - Functions are the fundamental unit of programming
 - Functions have input and output only
 - Minimal or no side effects
- **Key concepts, covered earlier in the Scala part of this course**
 - Passing functions as input to other functions
 - Anonymous functions

- Pseudocode for the RDD `map()` operation
 - Applies function `fn` to each RDD element

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


- Use *def* to define a function once
 - This function can be called multiple times

```
> def toUpper(s: String): String = {s.toUpperCase}

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

> mydata.map(toUpper).take(2)
```

- **Anonymous functions**
 - Defined in-line without an identifier
 - Best for short functions that are only called once
- **Supported in many programming languages**
 - Python: `lambda x: ...`
 - Scala: `x => ...`
 - Java 8: `x -> ...`

- Scala allows anonymous parameters using underscore

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

... becomes ...

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

■ Using Java with Spark

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());  
...
```

1. Review
2. Distributed Processing Background
3. Spark Overview
4. Spark Basics
5. **Using 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

- You can create RDDs from collections instead of files
 - `sc.parallelize(collection)`
- Useful when
 - Testing
 - Generating data programmatically
 - Integrating

- **For file-based RDDs, use `SparkContext.textFile`**
 - Accepts a single file, a wildcard list of files, or a comma-separated list of files

```
sc.textFile("myfile.txt")
```

```
sc.textFile("mydata/*.log")
```

```
sc.textFile("myfile1.txt,myfile2.txt")
```

- **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**

- `textFile` maps each line in the file to its own RDD element

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



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.

- `textFile` only works with newline-delimited text files
- What about other formats?

- **Spark uses Hadoop InputFormat and OutputFormat Java classes**
 - **TextInputFormat / TextOutputFormat**
 - Newline-delimited text files
 - **SequenceInputFormat / SequenceOutputFormat**
 - **FixedLengthInputFormat**
- **Other implementations available in additional libraries, such as the Avro library**
 - **AvroInputFormat / AvroOutputFormat**

- `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": "Wilma",  
  "lastName": "Flintstone",  
  "userid": "123"  
}
```

file2.json

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



```
(file1.json, {"firstName": "Wilma", "lastName": "Flintstone", "userid": "123"} )  
(file2.json, {"firstName": "Betty", "lastName": "Rubble", "userid": "234"} )  
(file3.xml, ... )
```

■ Process JSON

```
> 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:

```
Wilma
Betty
```

■ Single-RDD Transformations

- **flatMap**

- Maps one element in the base RDD to multiple elements

- **distinct**

- Filters out duplicates

- **sortBy**

- Use provided function to sort

■ Multi-RDD Transformations

– **intersection**

- Creates a new RDD with all elements in both original RDDs

– **union**

- Adds all elements of two RDDs into a single new RDD

– **zip**

- Pairs each element of the first RDD with the corresponding element of the second

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

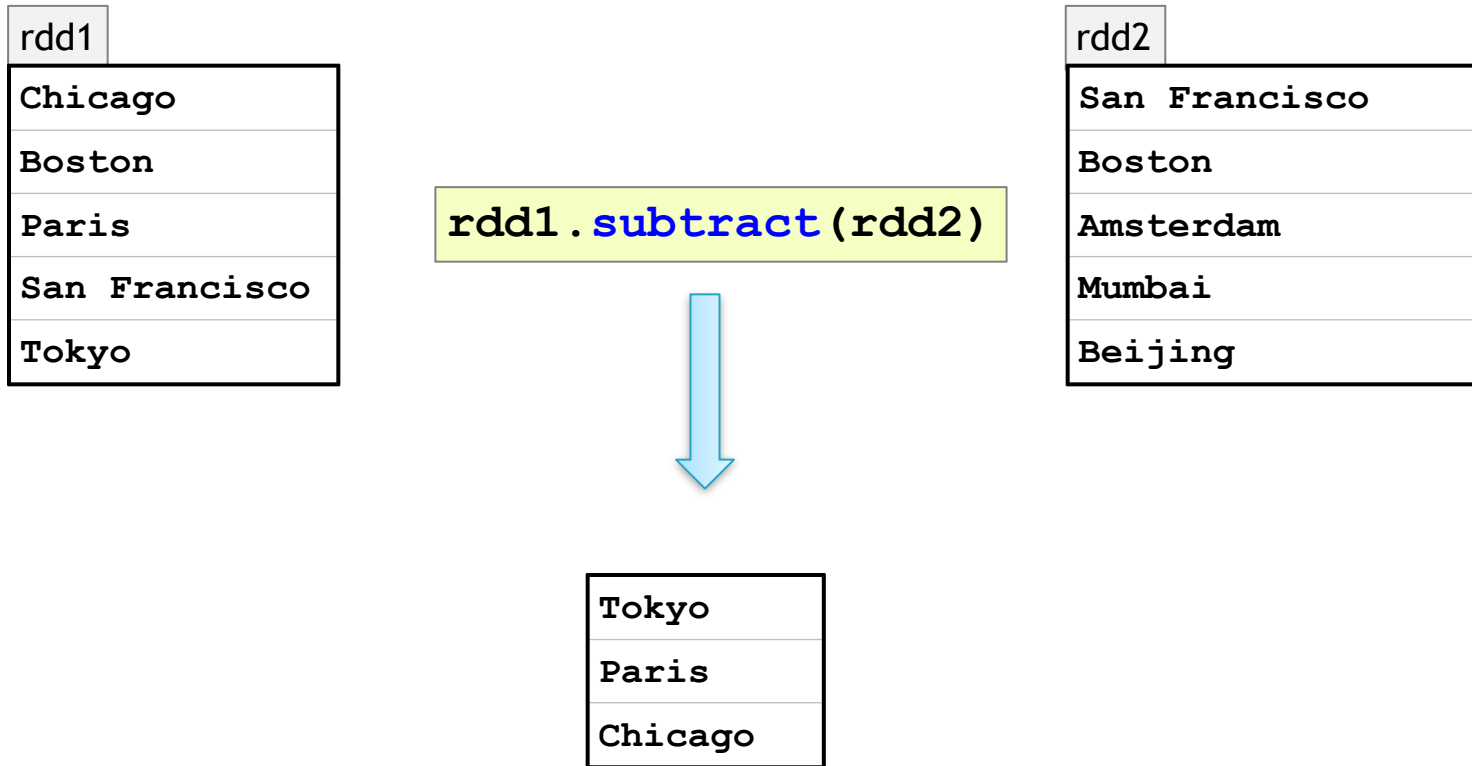
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

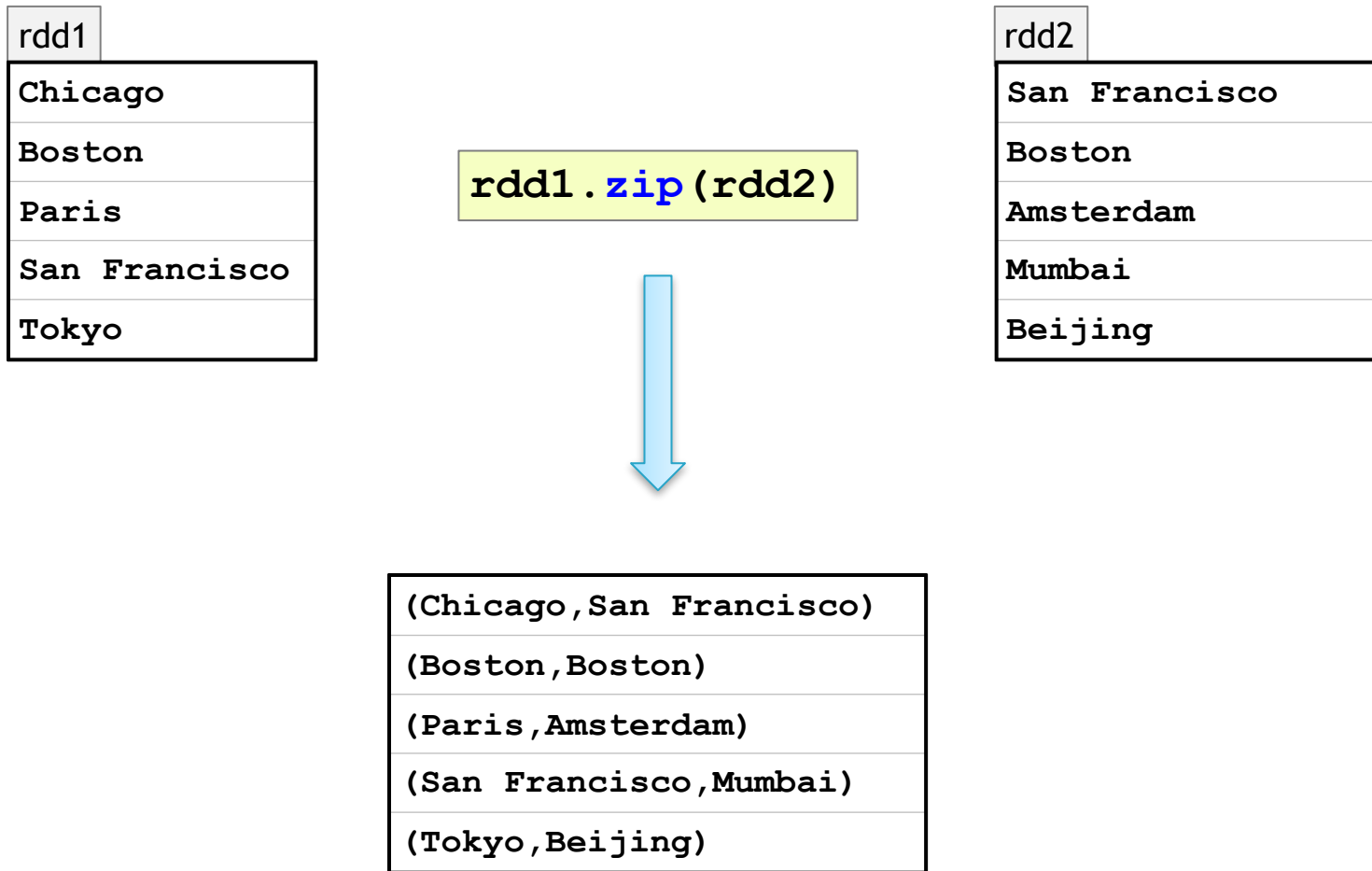


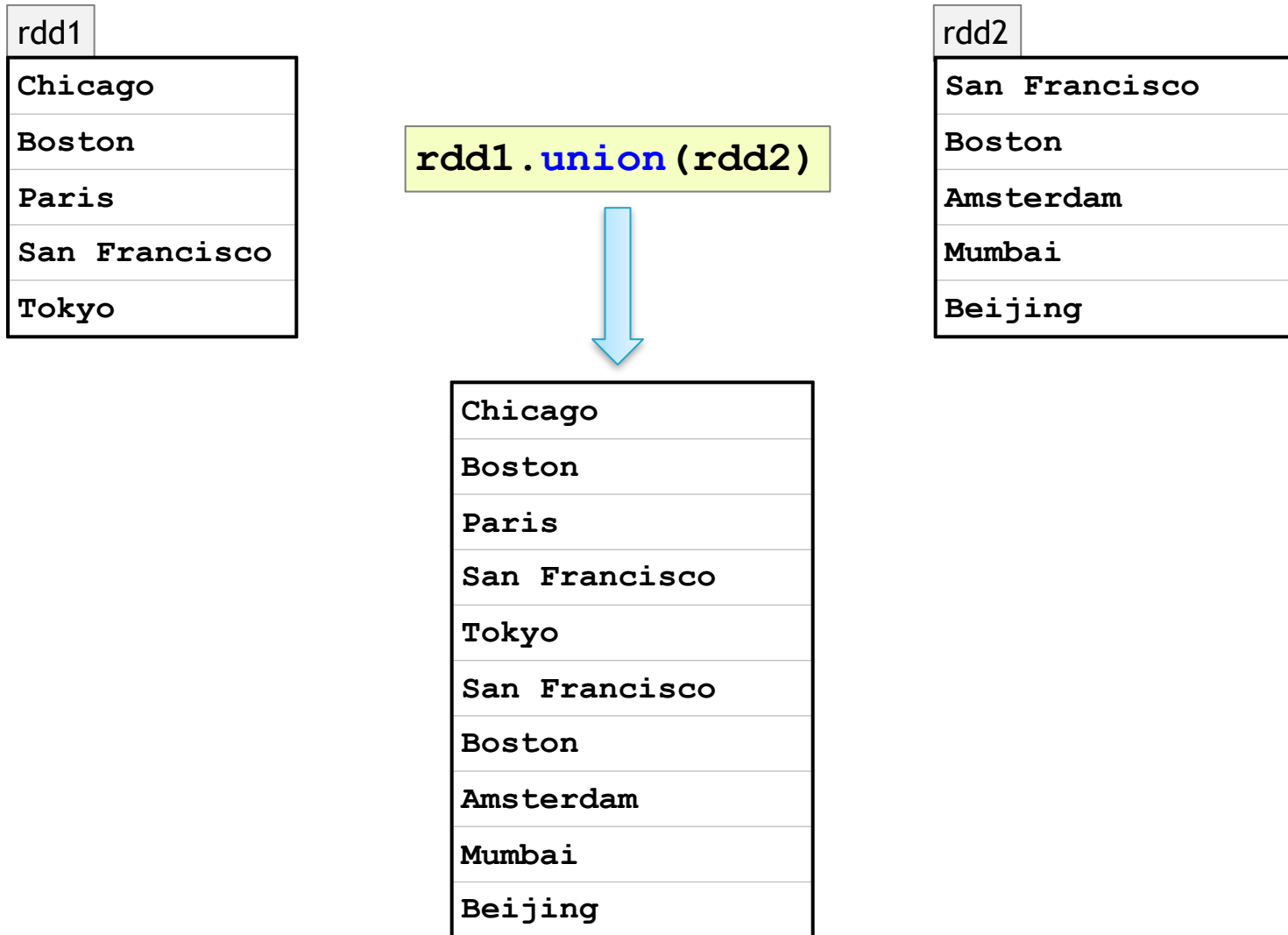
I've
never
seen
a
purple
cow
I
never
hope
to
...



I've
never
seen
a
purple
cow
I
hope
to
...







■ Other RDD operations

- **first**

- Returns the first element of the RDD

- **foreach**

- Applies a function to each element in an RDD

- **top(*n*)**

- Returns the largest *n* elements using natural ordering

- **Sampling operations**

- **sample**

- Creates a new RDD with a sampling of elements

- **takeSample**

- Returns an array of sampled elements

- **Double RDD operations**

- Statistical functions: **mean, sum, variance, stdev**

Homework

See homework packet.