UNIVERSITY of WASHINGTON

BIGDATA 210: Introduction to Data Engineering

Autumn 2018

Module 3: Data Processing Using Spark Part I

Jerry Kuch

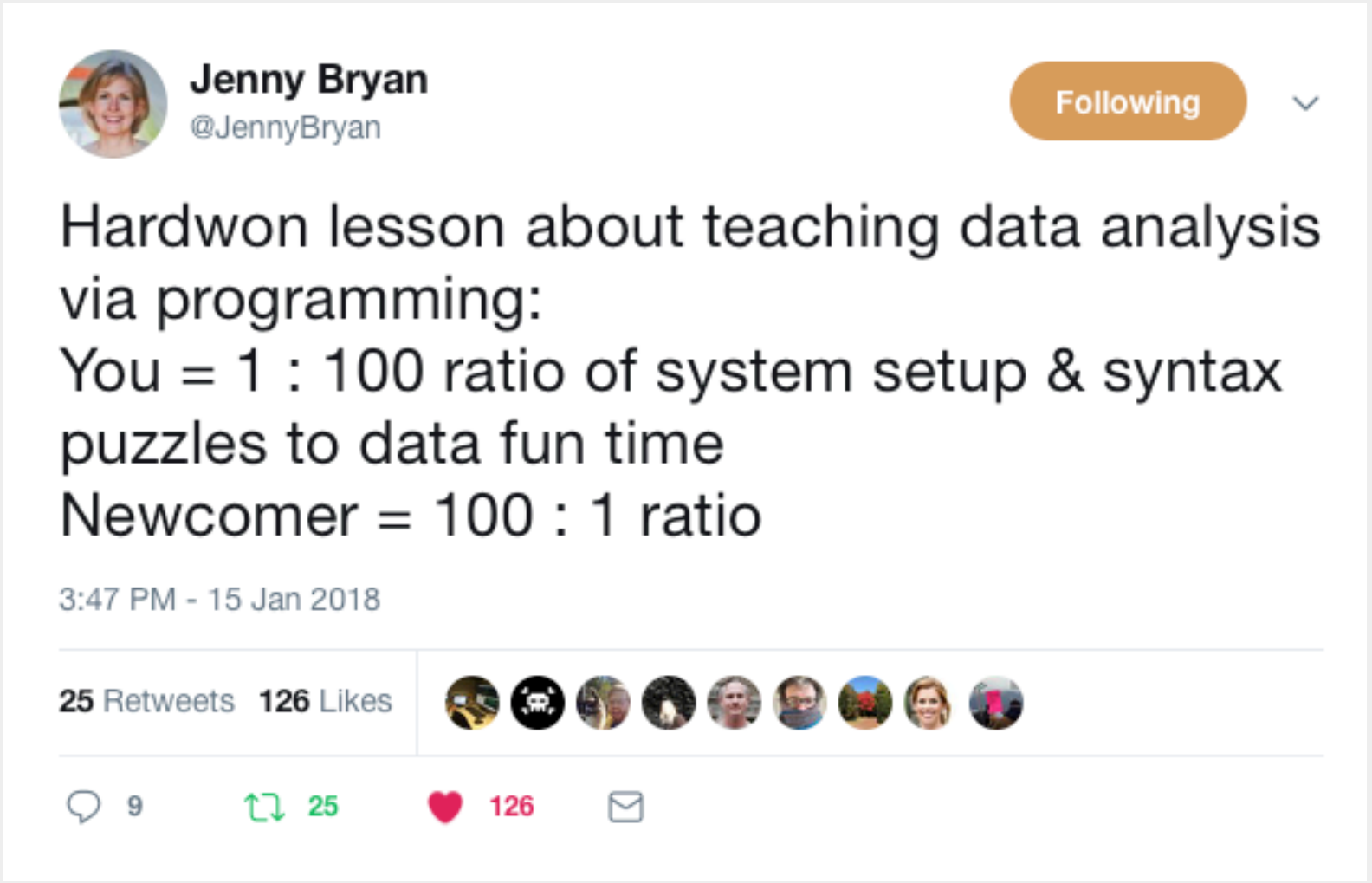
jkuch@uw.edu

Week 3 Agenda

- VM/Sandbox Discussion
- Week 2: Review
- New Topic: Programming in Spark Part I
  - What is Spark?
  - Spark architecture
  - Programming in Spark
    - Functional Programming Concepts
    - RDDs

Week 2: Recap

Cautionary / Inspiring Note



Jenny Bryan
@JennyBryan

Following

Hardwon lesson about teaching data analysis via programming:
You = 1 : 100 ratio of system setup & syntax puzzles to data fun time
Newcomer = 100 : 1 ratio

3:47 PM - 15 Jan 2018

25 Retweets **126** Likes

9 25 126

More on Week 2

- Reference: Sandbox “Learning the Ropes” tutorial
  - https://hortonworks.com/tutorial/learning-the-ropes-of-the-hortonworks-sandbox
  - It’s in the Resources and you might find it useful
- We’ll take a tour of things now:
  - The Azure VM
  - The Hortonworks Sandbox Docker container
  - Ambari and the command lines
  - Hive
- That will leave you set up for the Week 2 Assignment
- We’ll review the idea of MapReduce with a by-hand example

VM, Sandbox, etc. Walkthrough

- Microsoft Azure
  - The Azure Portal and finding your VM
  - Set up login credentials and log in
- The Hortonworks Sandbox (in a container)
- The *Ambari* Hadoop Web UI
- The HDFS command line
- Using Hive via GUI and command line

MapReduce: By Hand

- Before we move on…
- Let's review MapReduce…
- …with an example done by hand.
- The ponderous classic… *Word Count.*

What is Spark?

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

What is Spark?
A Response to Hadoop's Discontents

- Recall our classic Hadoop grievances:
  - Programming Model
    - Strictly map -> reduce with key/value pairs
    - Code requires much boilerplate and pipefitting
  - Slow
    - HDFS dependent—many reads and writes to/from slow storage media.
    - Doesn't support multiple transformation stages directly—all communicate through HDFS.

What is Spark?

- Spark addresses these issues (and more):
  - Programming Model
    - ~~Strictly map -> reduce with key/value pairs~~
    - Supports many Scala/Java/Python/R/SQL constructs
    - Higher level APIs for most functionality (almost like working with a local collections library in your language)
  - Slow
    - ~~HDFS Dependent~~
    - ~~Doesn't support multiple stages of transformation~~
    - In-memory execution model
      - Algorithms that can reuse/share data run exponentially faster
    - Supports other storage media
    - Can be 10X to 100X faster than Hadoop MapReduce!

What is Spark?
Contrast with Hadoop MapReduce

```
// mapper
private IntWritable one = new IntWritable(1);
private IntWritable output = new IntWritable();
protected void map(LongWritable key, Text value, Context context) {
    String[] fields = value.split("\t");
    output.set(Integer.parseInt(fields[1]));
    context.write(one, output);
}

// reducer
IntWritable one = new IntWritable(1);
DoubleWritable average = new DoubleWritable();
protected void reduce(IntWritable key, Iterable<IntWritable> values, Context context) {
    int sum = 0;
    int count = 0;
    for(IntWritable value : values) {
        sum += value.get();
        count++;
    }
    average.set(sum / (double) count);
    context.write(key, average);
}
```

```
val input = sc.textFile("/path/to/input")
val sums = numbers.flatMap(line => line.split("\t")).reduce(_+_)
val count = numbers.count
val avg = sum / (count * 1.0)
```

What is Spark?
Contrast with Hadoop MapReduce

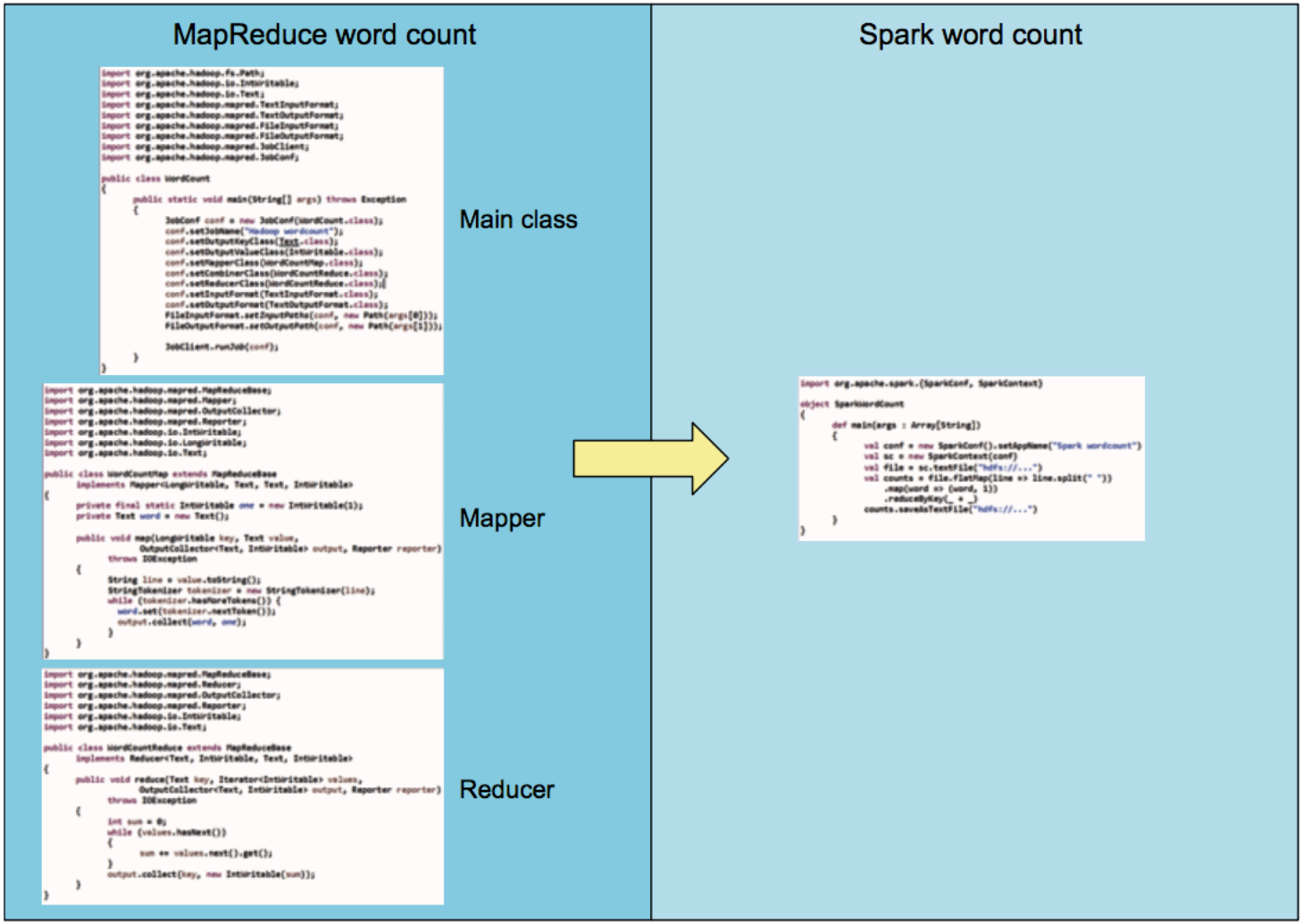
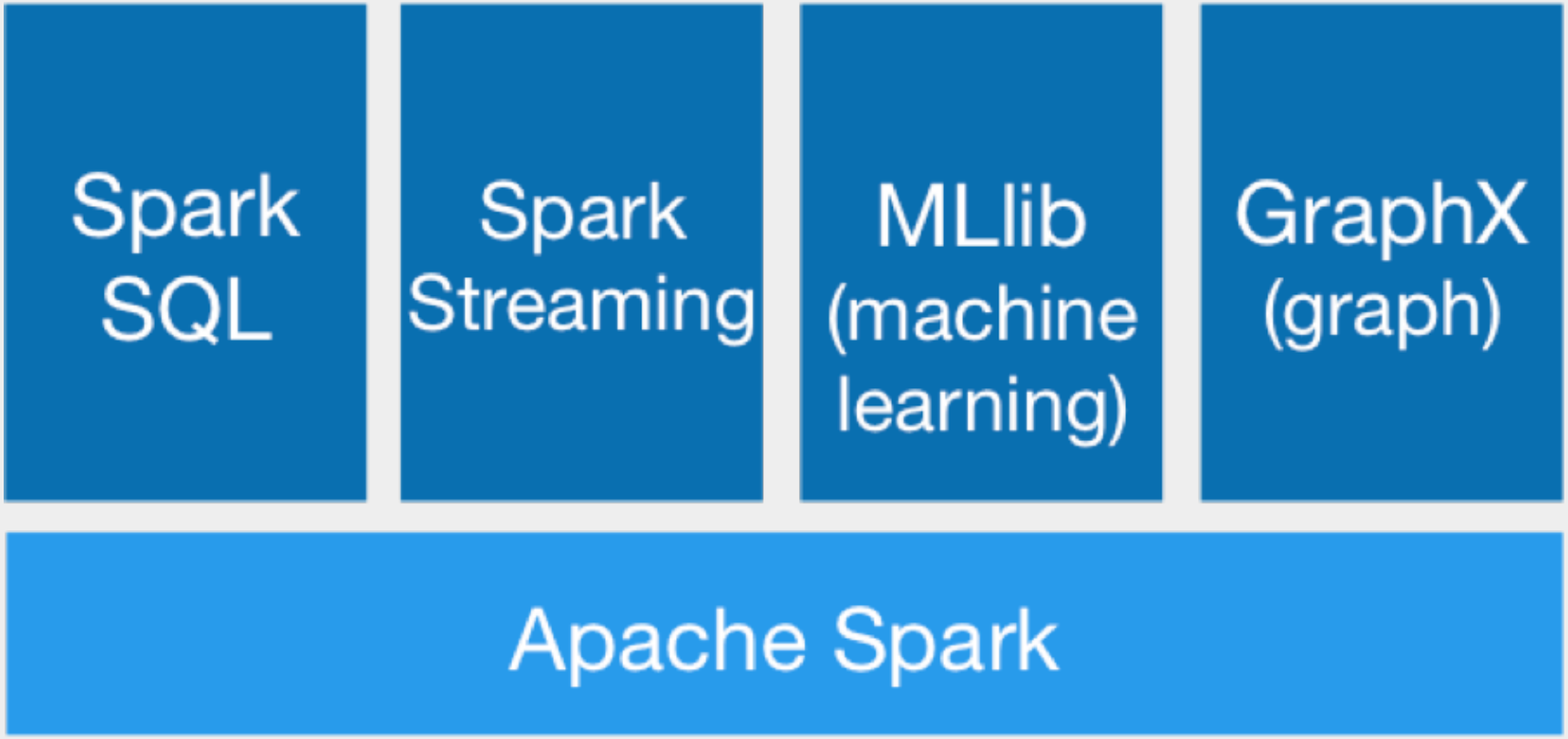


Figure 1.1 **A word-count program demonstrates Spark's conciseness and simplicity. The program is shown implemented in Hadoop's MapReduce framework on the left and as a Spark Scala program on the right.**

What is Spark?
Contrast with Hadoop MapReduce

- Hadoop MapReduce:
  - Much boilerplate code in Mappers, Reducers, Combiners…
  - Clunky Main program to set up configuration, runtime context, and chain together Mappers, Reducers, etc.
  - All of it is bulky Java code that you:
    - Write, compile, ship out to cluster, and wait for!
- Spark on the other hand:
  - Demands less boilerplate
  - Has a higher level, collections-like API (and others)
  - Has an interactive REPL!

What is Spark?
The Main Components



Spark's Components: The Core and Nearby

- **Spark Core**
  - Basic functionality to run jobs
  - *RDD (Resilient Distributed Dataset)* abstraction and API
  - Ability to access various data sources (HDFS, S3, etc.)
- **Spark SQL**
  - SQL atop Spark, akin to Hive atop MapReduce
  - *DataFrame* and *DataSet* abstractions:
    - Easier handling of structured data
    - Enable database-style query optimization
  - Many supported data formats and sources (JSON, Parquet, RDBMS, Hive, etc.)
  - Spark SQL can take advantage of classical database style performance optimizations, somewhat automatically

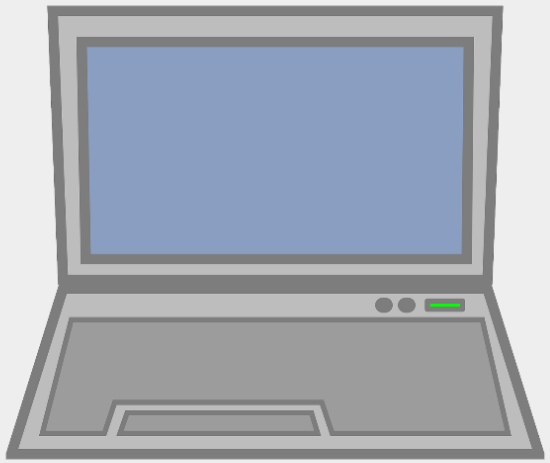
Spark's Components
Other Stuff: Streaming and MLlib

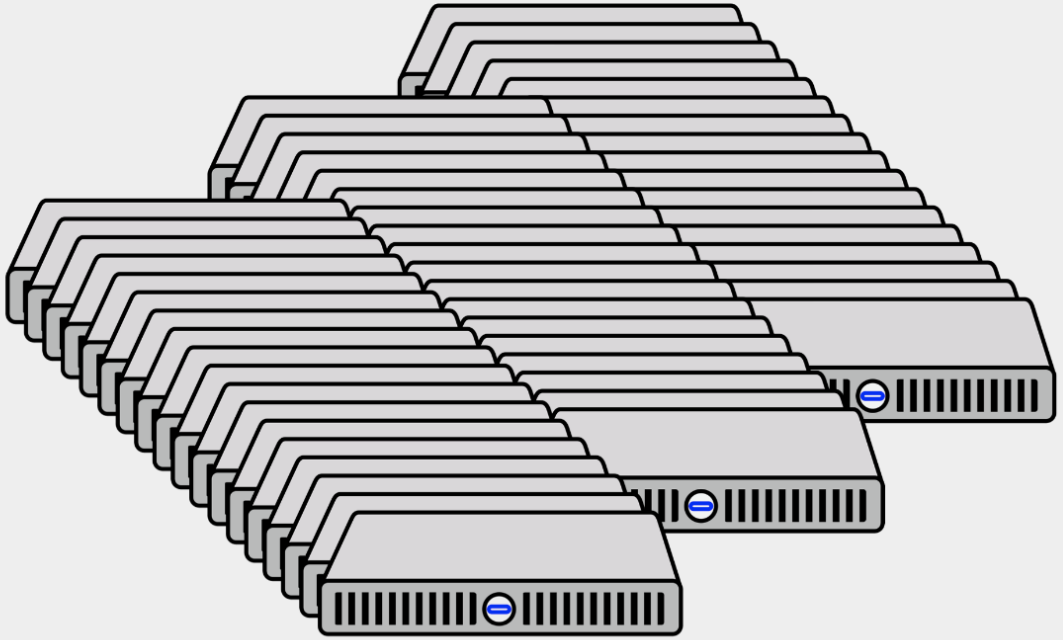
- **Spark Streaming**
  - Ingests "real-time" streaming data
  - Supported sources: HDFS, Kafka, Flume, ZeroMQ, user customized…
  - Makes RDDs out of batched windows
- **Spark MLlib**
  - Machine learning algorithms
  - Regression, naive Bayes classifiers, SVMs, decision trees, random forests, k-means clustering
  - Mahout is migrating from MapReduce to Spark

Spark's Components
Other Stuff: GraphX

- **Spark GraphX**
  - Graph as in G = (V,E), vertices and edges
  - *EdgeRDD* and *VertexRDD*
  - Provides key algorithms: PageRank, connected components, shortest paths, SVD
  - *Pregel* style message-passing API (also used in Apache Giraph which runs on Hadoop)

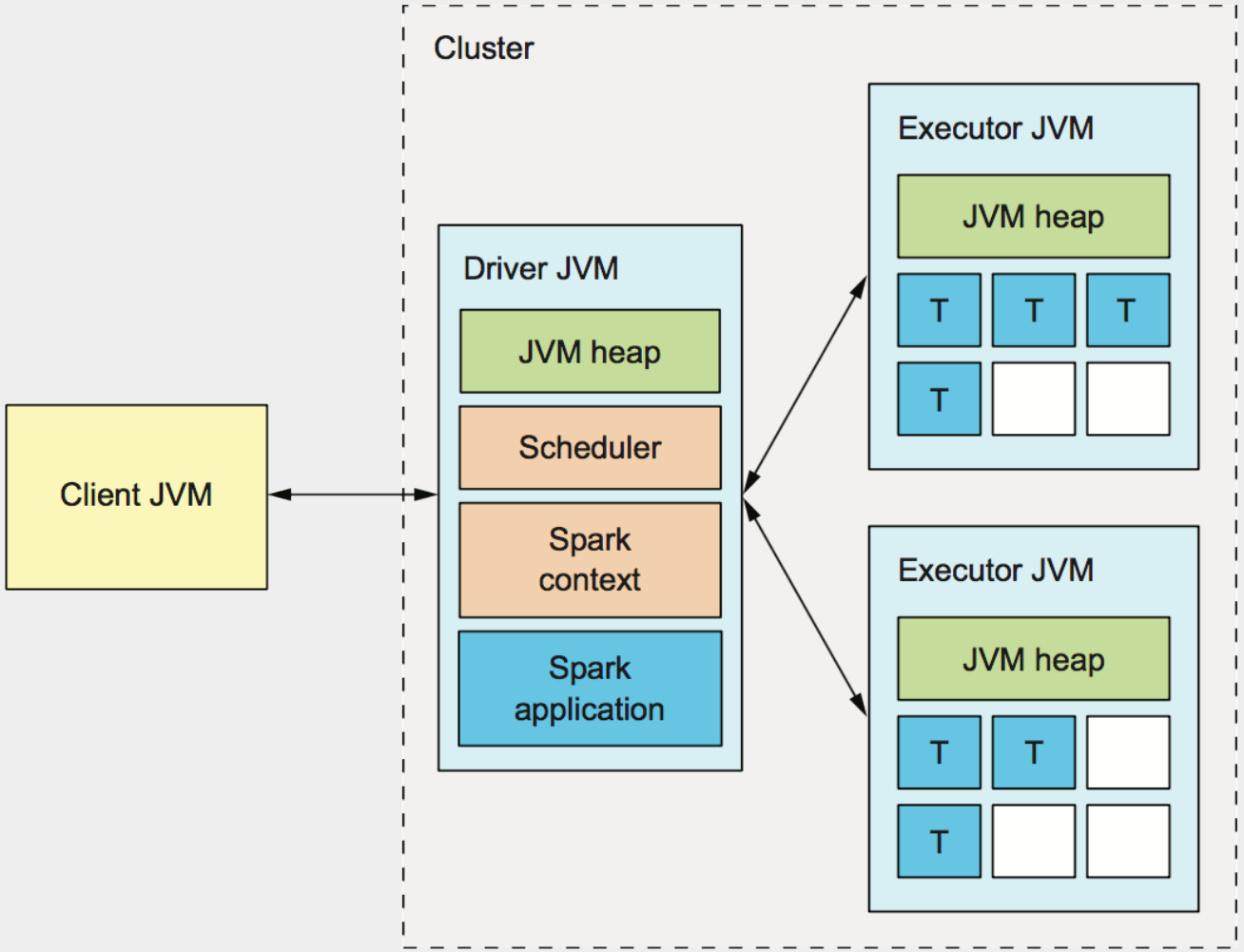
Spark Application Architecture: Again Clusters of Commodity Hardware





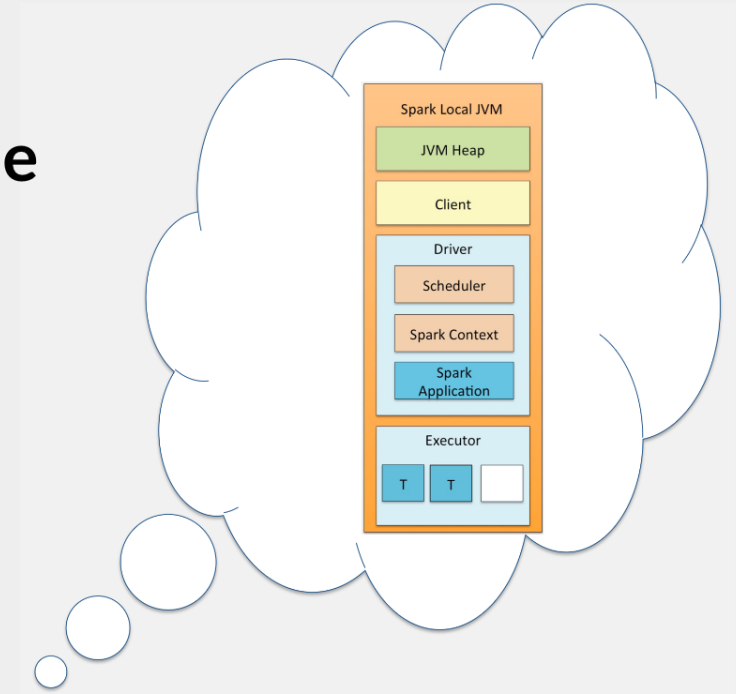
Spark Application Architecture: The Software

Spark Runtime Architecture



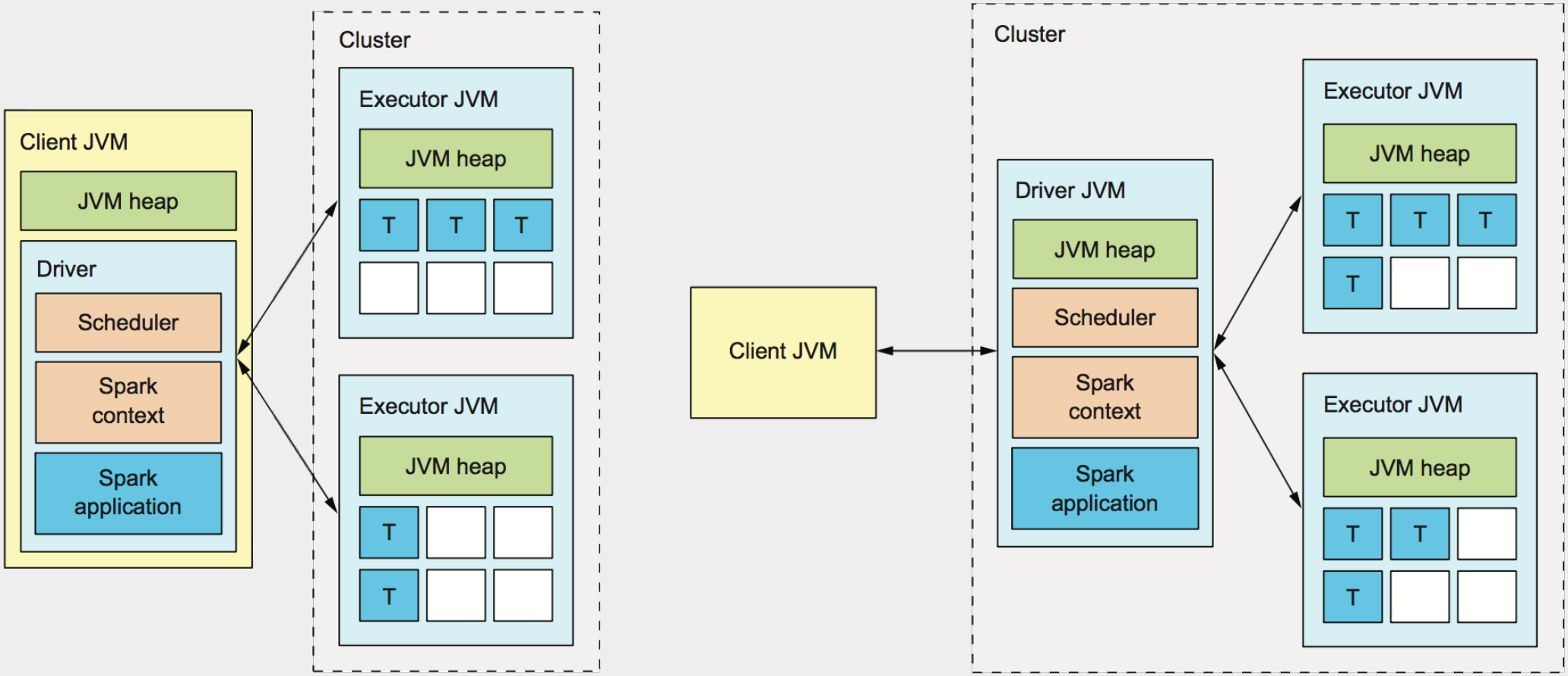
Spark Application Architecture: Local Mode

Local Mode



Spark Application Architecture: Cluster Mode

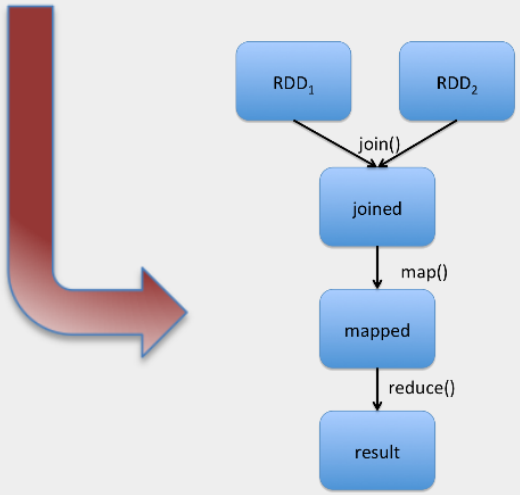
Cluster Mode

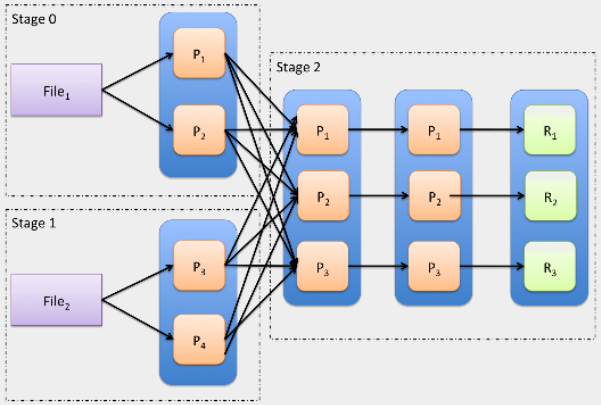


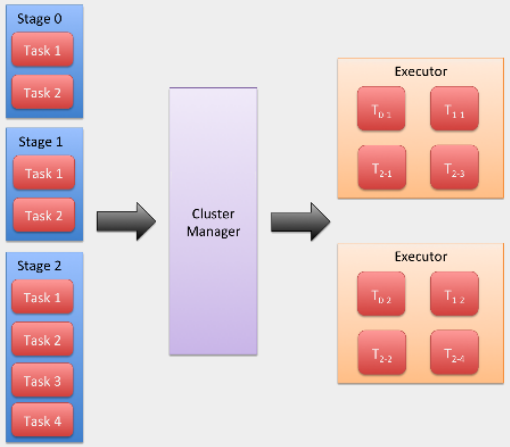
Spark Application Architecture

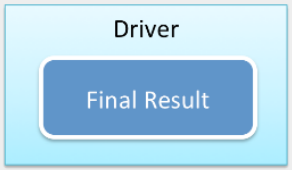
Spark Application Lifecycle

```
val rdd1 = sc.textFile(…)
val rdd2 = sc.textFile(…)
val joined = rdd1.join(rdd2)
val mapped = joined.map(…)
val results = mapped.reduce(…)
```









Functional Programming

A Brief Aside

Programming Spark: What is Functional Programming?

- What is Functional Programming anyway?
- What are some example FP languages?

Programming Spark: What is Functional Programming?

- What is Functional Programming?
  - Computation is *evaluation*
  - *Pure* functions don't have *side effects*
  - *Higher order functions* can take other functions as arguments
  - Functions can be treated as data/objects
  - Immutability is favored

Programming Spark: Functional Programming, Advantages

- We saw several advantages of immutability in MapReduce
- Map and Reduce themselves *are* higher order functions!
- Eliminating side effects eases reasoning

Programming Spark: FP: Declarative vs. Imperative

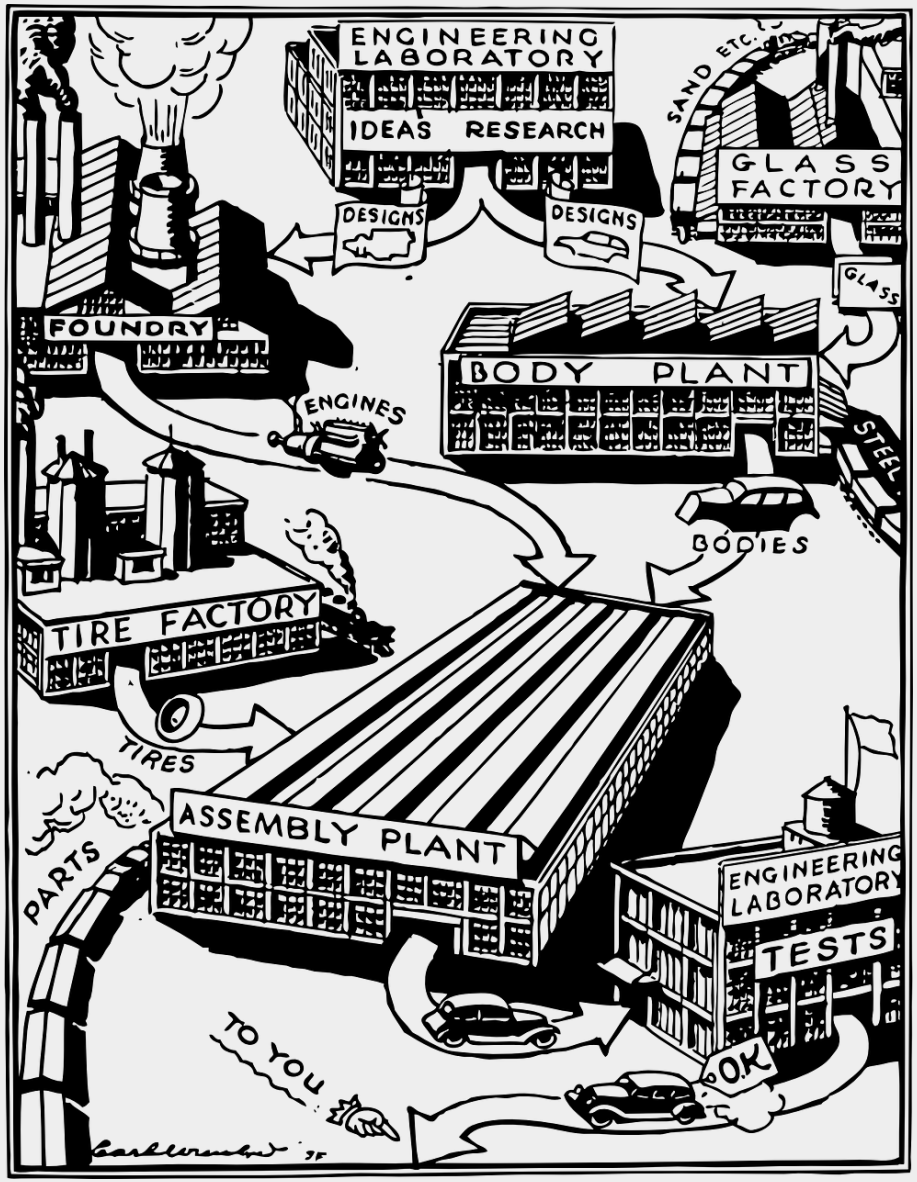
- Declarative vs Imperative Paradigms…
  - Functional programs are often more declarative (what do I want?) rather than imperative (how do I do what I want)…

```
int[] array = new int[] {1, 2, 3, 4, 5};
for (int i = 0; i < array.length; i++) {
    array[i] = array[i] * 2;
}
```

```
val array = Array(1, 2, 3, 4, 5)
val doubled = array.map(x => x * 2)
```

Imperative Programming: A bunch of guys fiddling with stuff

Functional Programming: Artifacts that feed into processes that make other artifacts…



Functional Programming and Spark

- Spark is written in Scala (a FPL) and its design is essentially functional
- AN executing Spark Application can be:
  - A program you wrote and compiled, or…
  - REPL Shell, used interactively
    - Read, Eval, Print, Loop

Spark RDDs

Resilient, Distributed Datasets

Programming Spark: RDDs Basic Properties

- Resilient Distributed Dataset (RDD)
  - Data is partitioned or *sharded* across cluster nodes
  - Built-in fault tolerance via *replication*
  - Can represent different kinds of data
    - Text, Key/Value pairs, custom classes/objects
  - RDDs are *IMMUTABLE… Recall benefits:*
    - For *concurrency* and *scheduling*
    - For *failure tolerance* and *recovery*

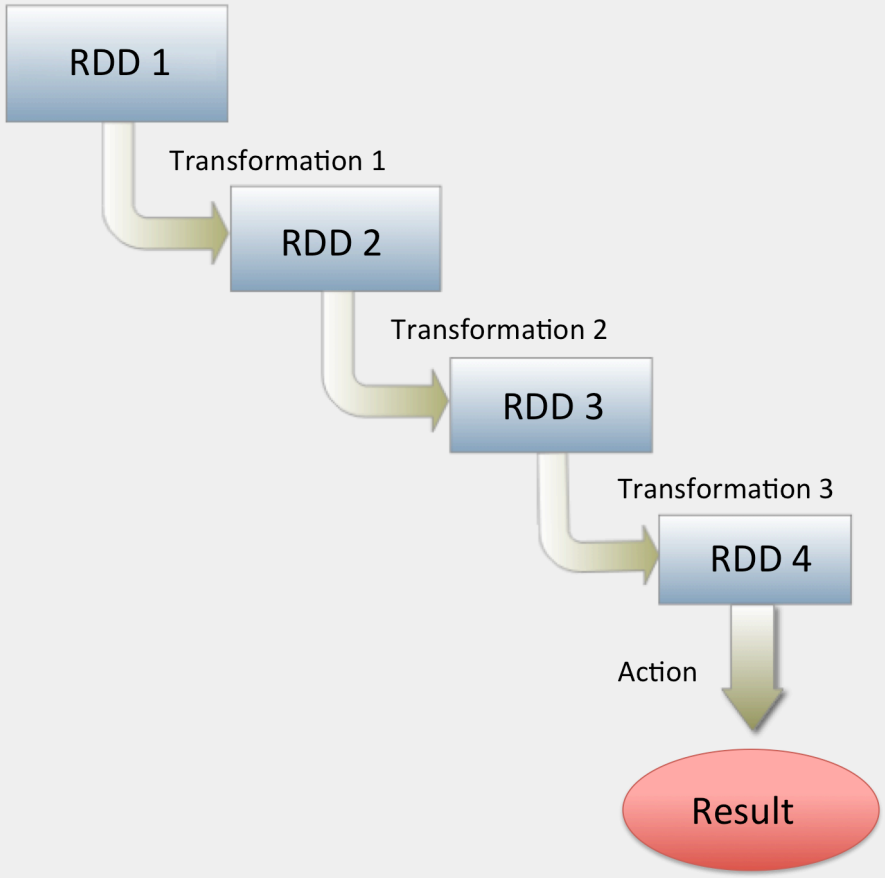
Programming Spark: RDDs Laziness

- Resilient Distributed Dataset (RDD)
  - LAZY EXECUTION

Programming Spark: RDDs Transformations and Actions

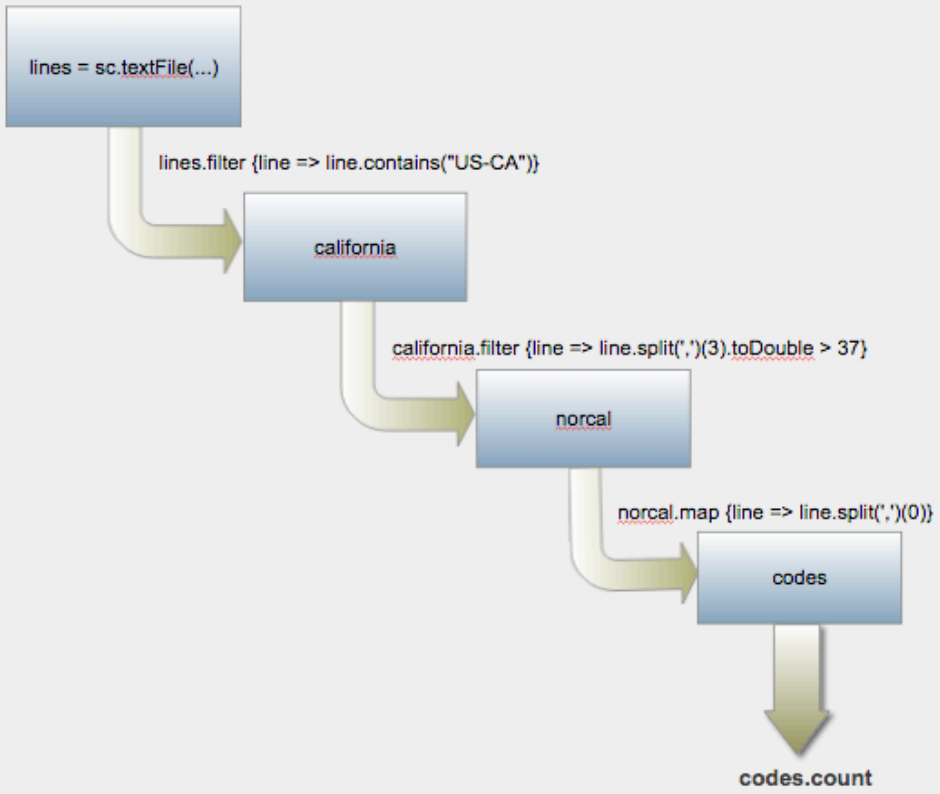
- Resilient Distributed Dataset (RDD)
  - Transformation
    - Create a new data set from an existing one
  - Action
    - Return a result to the driver
    - Does not need to be a “final” result

Programming Spark



Programming Spark

```
val lines = sc.textFile("file:///data/airport_codes.csv")
val california = lines.filter {line => line.contains("US-CA")}
val norcal = california.filter {line => line.split(',')(3).toDouble > 37}
val codes = norcal.map {line => line.split(',')(0)}
codes.count
```



Programming Spark

- Ways to create or load RDDs:
  - From an existing Scala collection

```
import scala.util._
val randoms = Seq.fill(1000)(Random.nextInt)
print(randoms.take(10))
```

  - From a Text File (on local FS or HDFS)

```
val lines = sc.textFile("file:///data/home_data.csv")
```

```
val lines = sc.textFile("hdfs://sandbox.hortonworks.com:8020/tmp/home_data.csv")
```

  - From a Whole Directory of Files

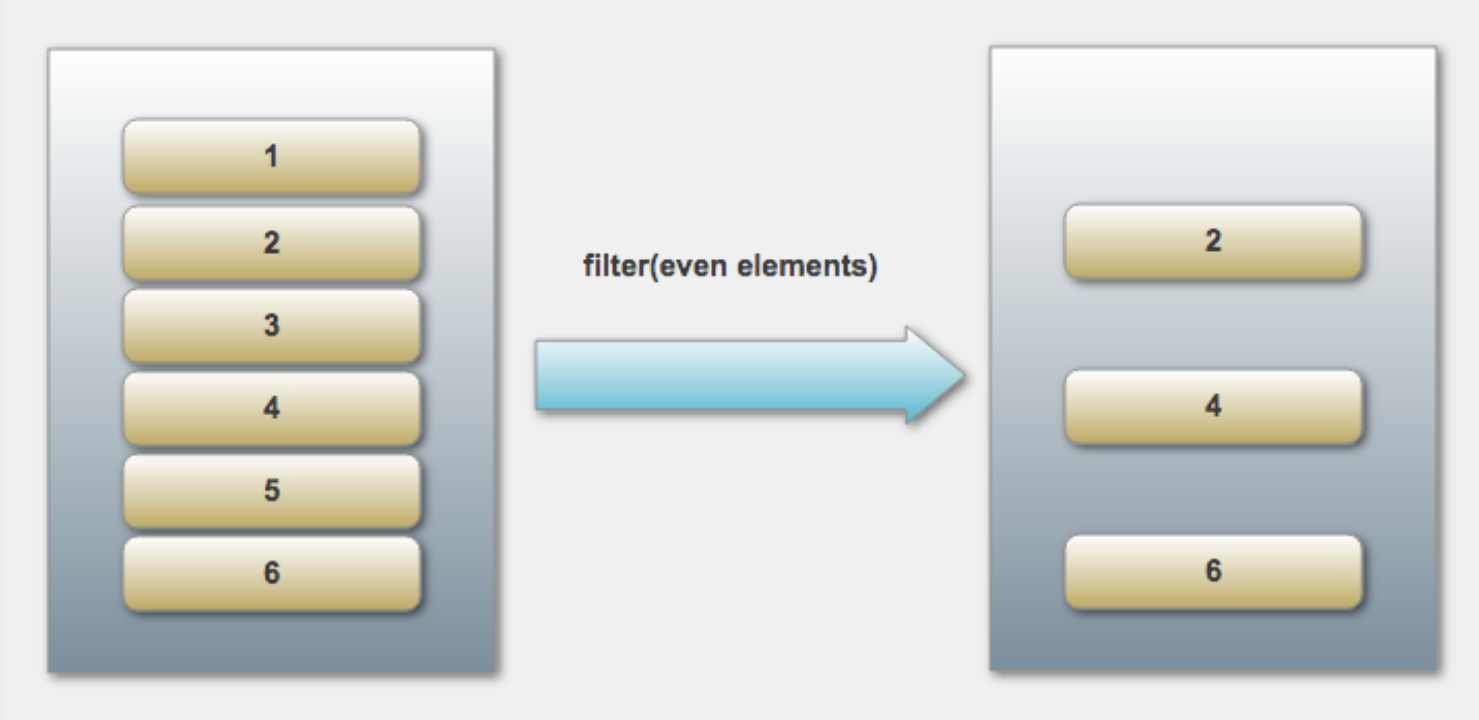
```
val files = sc.wholeTextFiles("file:///data/*.csv")
```

RDD Transformations

Transformations make a new RDD from an existing RDD in a specified way…

RDD Transformations

Filter



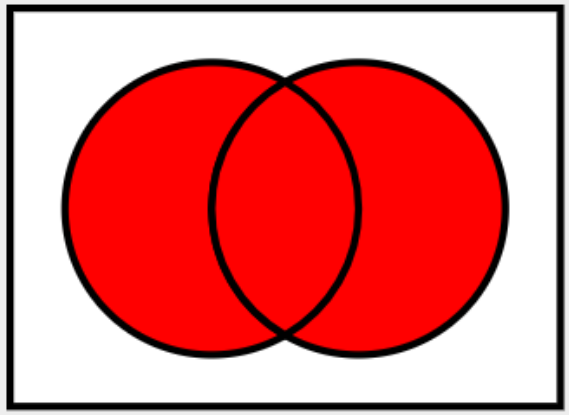
def filter(f: (T) ⇒ Boolean): RDD[T]

Return a new RDD containing only the elements that satisfy a predicate.

```
val exampleRDD = sc.parallelize(1 to 20)
val filteredRDD = exampleRDD.filter(value => value % 2 == 0)
```

RDD Transformations

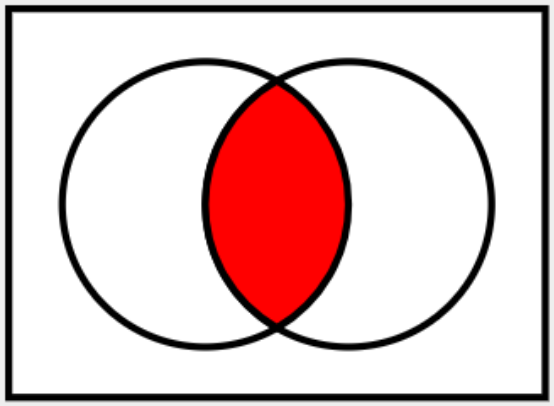
Union



def union(other: RDD[T]): RDD[T]

Return the union of this RDD and another one.

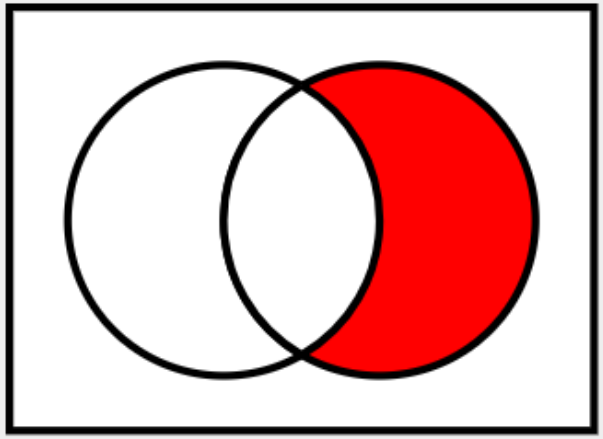
Intersection



def intersection(other: RDD[T]): RDD[T]

Return the intersection of this RDD and another one.

Subtraction



def subtract(other: RDD[T]): RDD[T]

Return an RDD with the elements from this that are not in other.

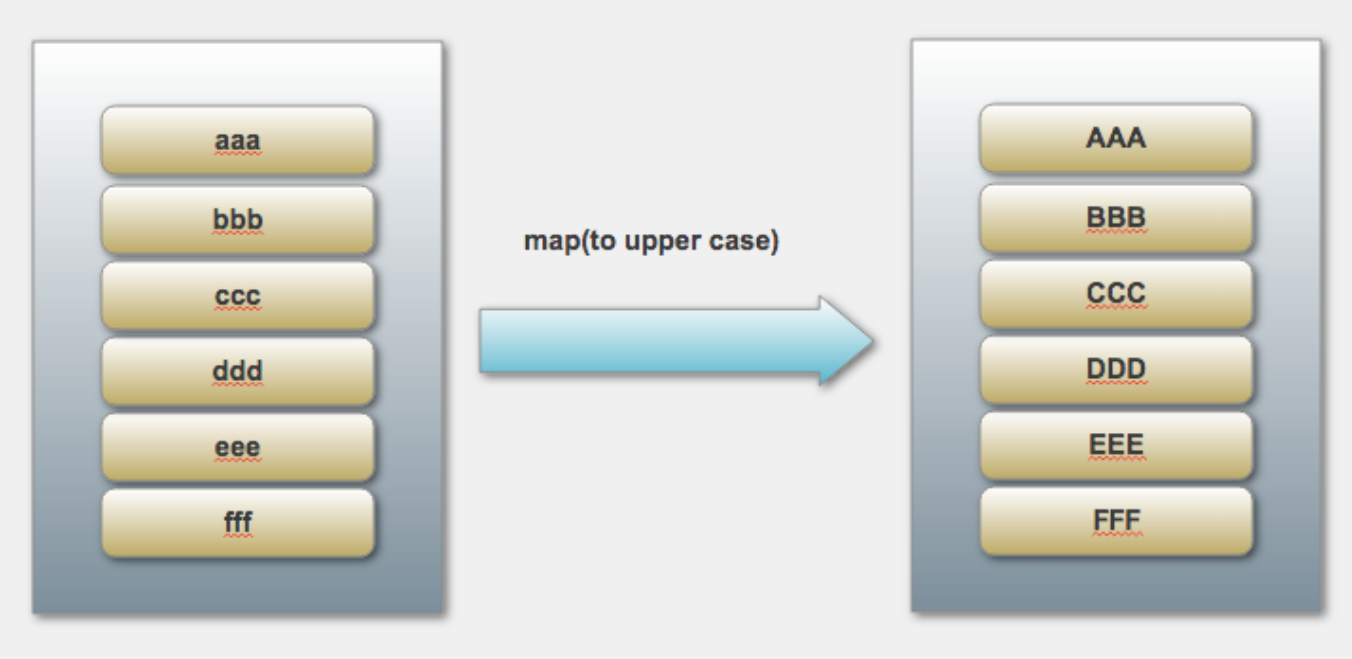
RDD Transformations

Set Operations

```
val a = sc.parallelize(1 to 10)
val b = sc.parallelize(11 to 20)


val aUnionB = a.union(b)
// Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

val c = sc.parallelize(1 to 10 by 2
val aIntersectC = a.intersection(c)
// Array(1, 9, 5, 3, 7)

val aUnionBSub.subtract(a)
// Array(16, 20, 12, 17, 13, 18, 14, 19, 11, 15)
```

RDD Transformations

Map

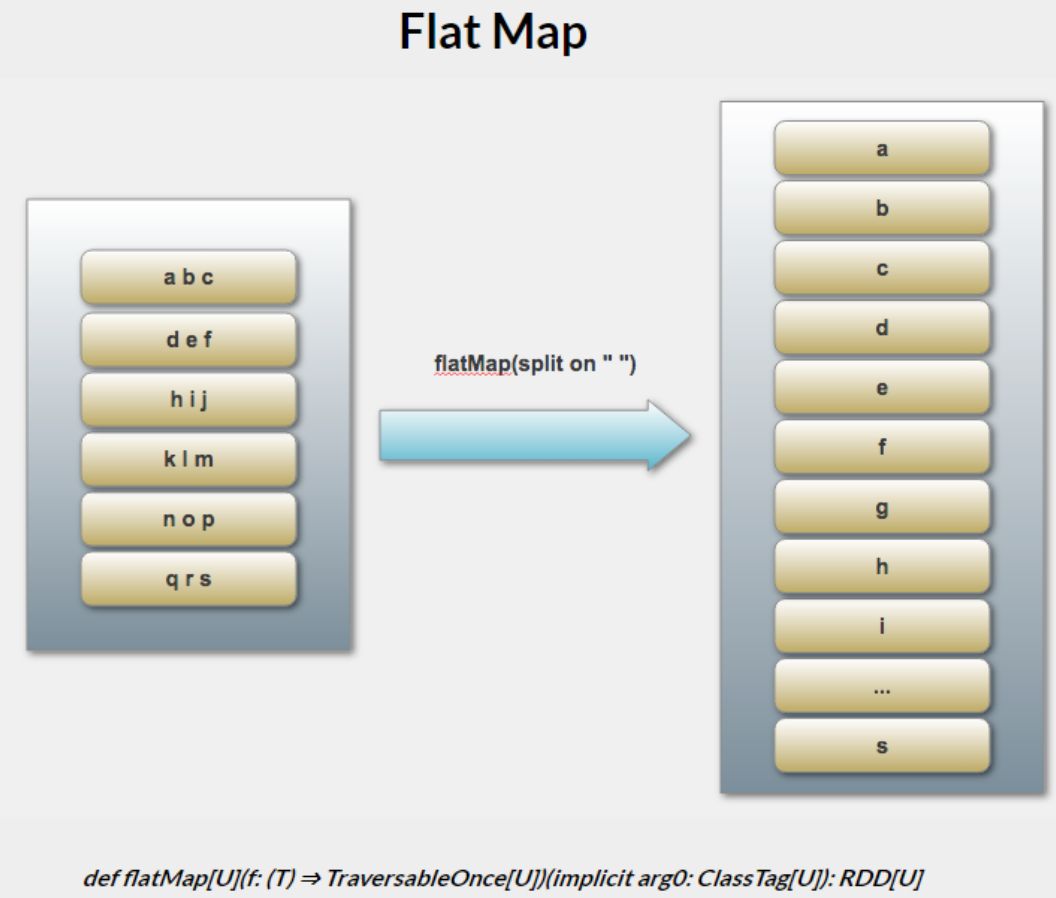


def map[U](f: (T) ⇒ U)(implicit arg0: ClassTag[U]): RDD[U]

Return a new RDD by applying a function to all elements of this RDD.

```
val text = sc.textFile("file:///data/war_and_peace.txt")
val upperText = text.map(line => line.toUpperCase())
```

RDD Transformations

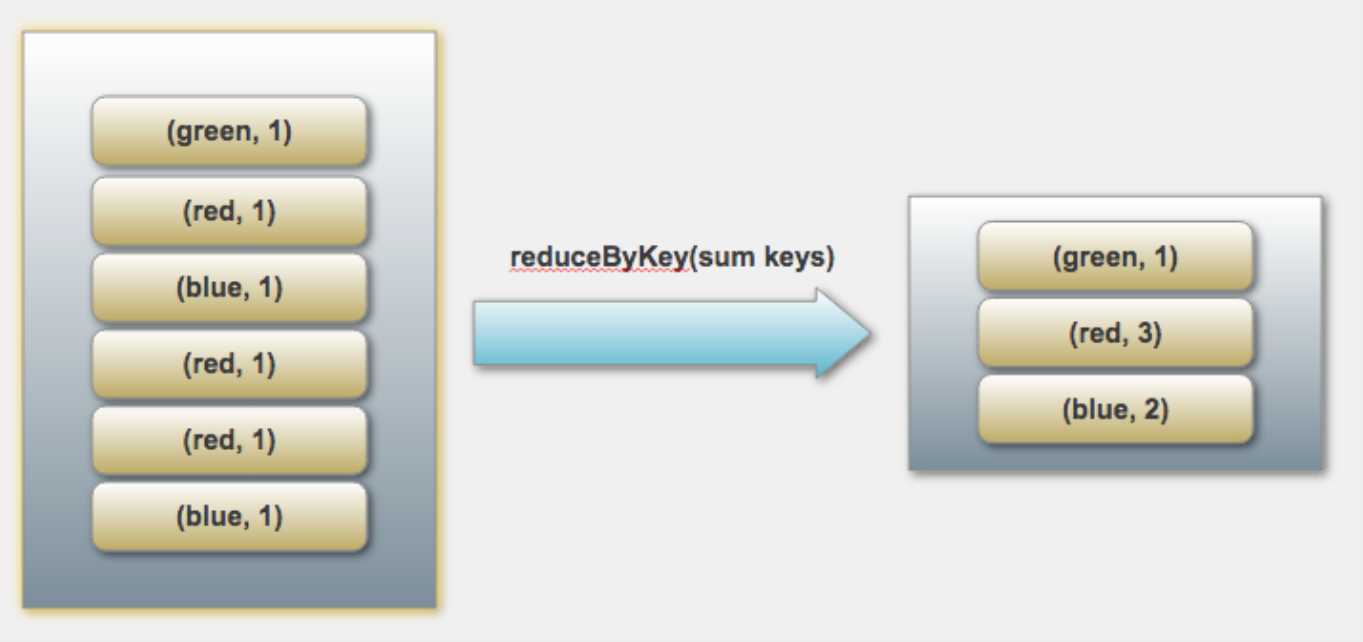


def flatMap[U](f: (T) ⇒ TraversableOnce[U])(implicit arg0: ClassTag[U]): RDD[U]

```
val text = sc.textFile("file:///data/war_and_peace.txt")
val words = text.flatMap(line => line.split(" "))
```

RDD Transformations

Reduce By Key



def reduceByKey(func: (V, V) ⇒ V): RDD[(K, V)]

Merge the values for each key using an associative and commutative reduce function.

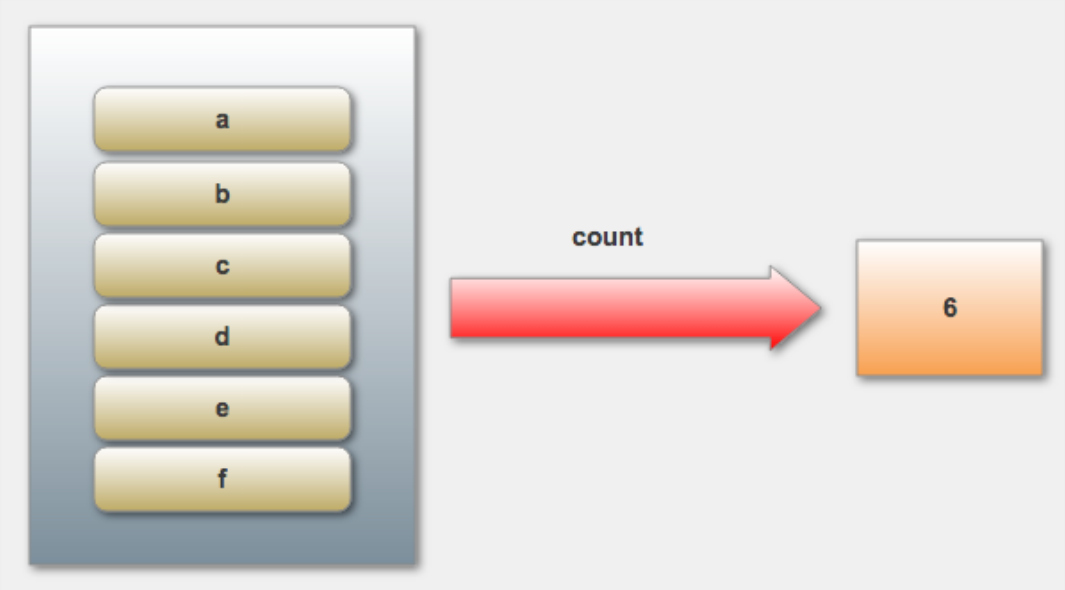
```
val text = sc.textFile("file:///data/war_and_peace.txt")
val words = text.flatMap(line => line.split(" "))
val wordsPairs = words.map(word => (word,1))
val counts = wordsPairs.reduceByKey((value1,value2) => value1 + value2)
```

RDD Actions

RDD *Actions* cause the transformations chained together to make a sequence of RDDs to be acted upon to produce some kind of result.

RDD Actions

Count



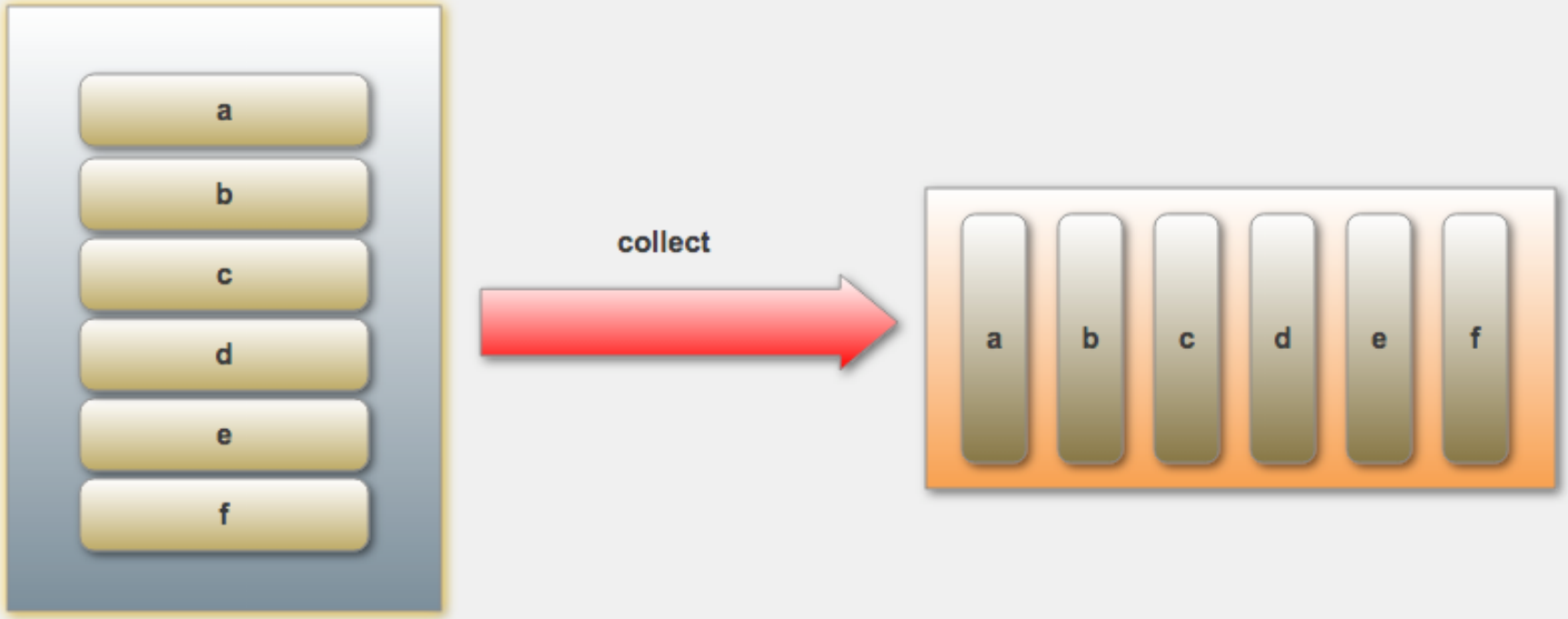
def count(): Long

Return the number of elements in the RDD.

```
val a = sc.parallelize(1 to 10)
a.count

// 10
```

RDD Actions

Collect



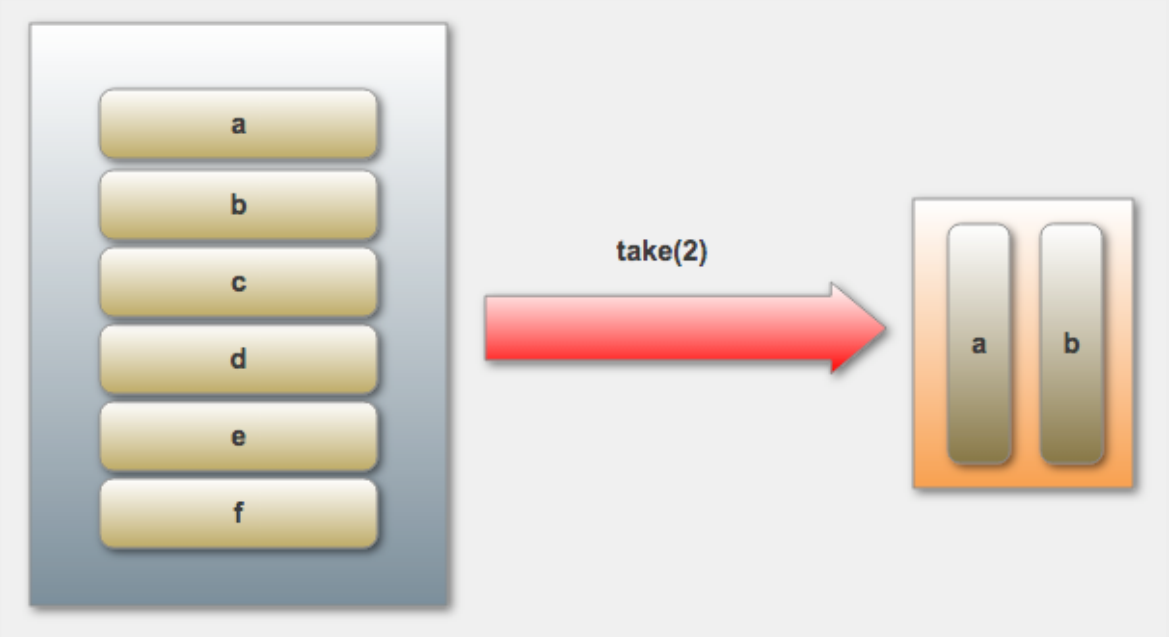
def collect(): Array[T]

Return an array that contains all of the elements in this RDD.

```
val a = sc.parallelize(1 to 10)
a.collect
// Array(1,2,3,4,5,6,7,8,9,10)
```

RDD Actions

Take



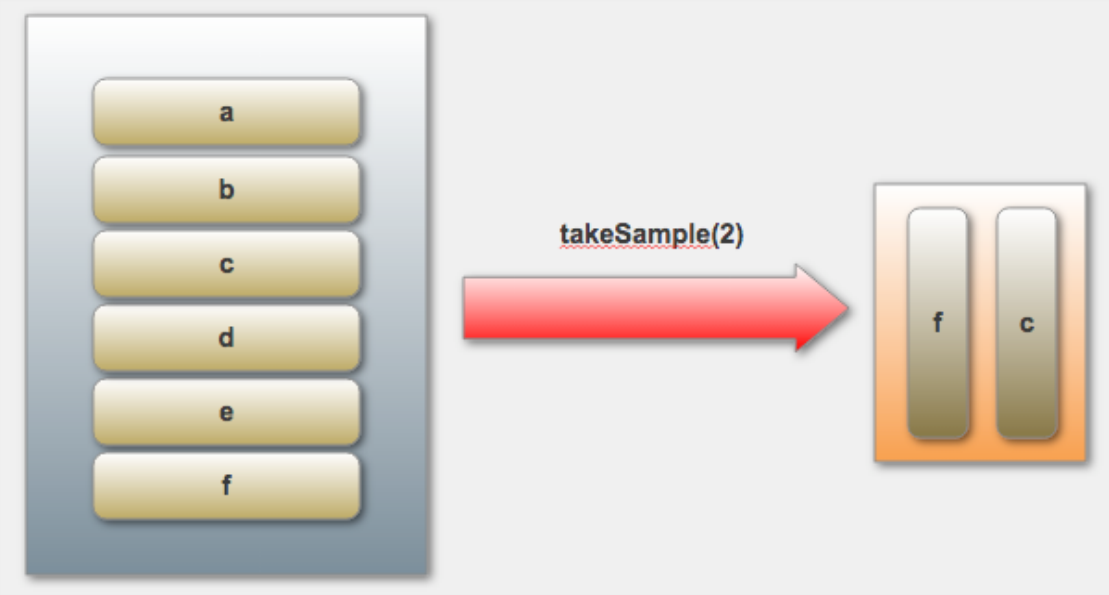
def take(num: Int): Array[T]

Take the first num elements of the RDD.

```
val a = sc.parallelize(1 to 10)
a.take(2)
// Array(1,2)
```

RDD Actions

Take Sample



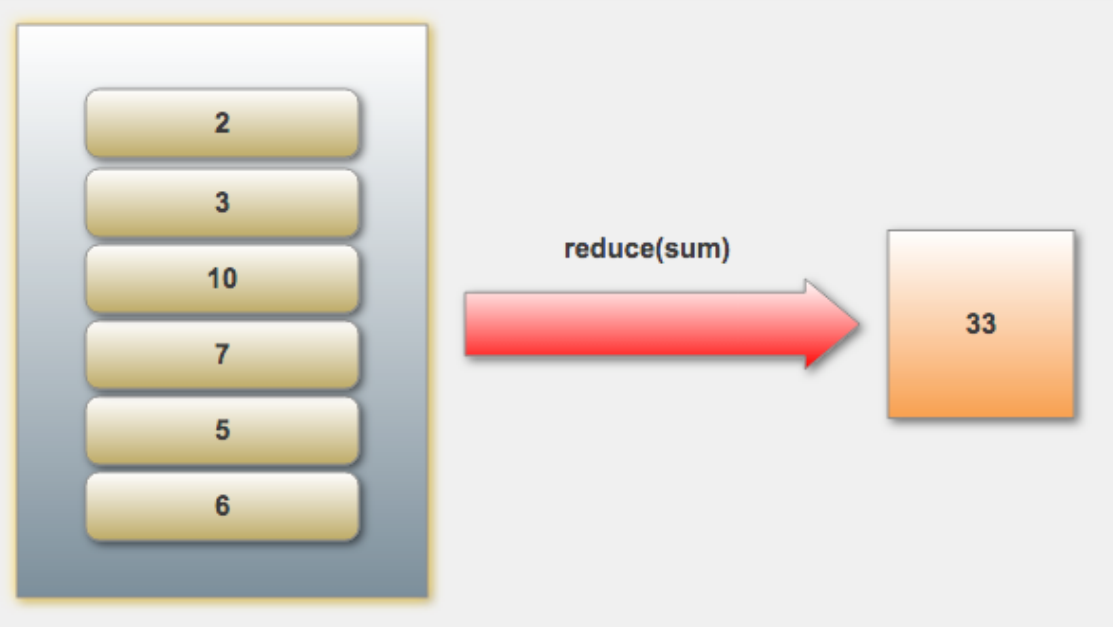
def takeSample(withReplacement: Boolean, num: Int, seed: Long = Utils.random.nextLong): Array[T]

Return a fixed-size sampled subset of this RDD in an array

```
val a = sc.parallelize(1 to 10)
a.takeSample(false, 2)
// Array(6, 8)
```

RDD Actions

Reduce



def reduce(f: (T, T) ⇒ T): T

Reduces the elements of this RDD using the specified commutative and associative binary operator.

```
val a = sc.parallelize(1 to 100)
a.reduce((v1,v2) => v1 + v2)
// 5050
```

Saving RDDs

Sometimes you want to save an RDD

RDD Save Actions

Save an RDD to disk as a text file, one element per line. Note that is this an output directory, not a file name. Each individual partition will be written to a separate file

```
counts.saveAsTextFile("file:///data/word_counts")
```

Save an RDD to disk using Java serialization. Also specifies an output directory, with individual files for each partition

```
counts.saveAsObjectFile("file:///data/word_counts_binary")
```

Spark Resources

API documentation

http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD

http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

http://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html

Week 3: Introduction to Spark Part 1 Summary

- The troubles with classic Hadoop
- Apache Spark: components and architecture
- Functional Programming, Scala and Spark
- Spark RDDs
  - Immutability, Laziness, Distribution
  - Transformations and Actions

Week 3: Homework

- *Just Enough Scala for Spark* worksheet to get you warmed up
- Get started using Spark itself
- Work some data analysis problems similar in scope/size to Week 2's Hive exercises