

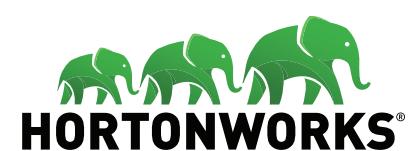
Presentation Slides
For INSTRUCTOR Use Only

Spark For Developers

Neha Priya

Version 20180209 Rev 1.01

Copyright © Hortonworks All rights reserved.



About Me: Neha Priya



Course Overview

- In-Depth, technical Introduction to Spark
- Course coverage includes the following
 - **Motivation:** Spark's place in the world of Big Data
 - **Architecture:** Spark structure and distributed compute engine
 - **Installation/Setup/Usage**
 - **Spark Shell:** For ad-hoc interactive operations
 - **RDDs:** Resilient Distributed Datasets
 - **Spark API:** Programming interface and standalone programs
 - **Spark SQL/DataFrames:** High level API for structured data
 - **Spark Streaming:** For streaming (real-time) data
 - **Performance:** Guidelines and tuning

Course Agenda

- Session 1 (Optional): **Scala Ramp Up**
- Session 2: **Introduction to Spark**
- Session 3: **RDDs and Spark Architecture**
- Session 4: **Spark SQL, DataFrames, and Datasets**
- Session 5: **Shuffling Transformations and Performance**
- Session 6: **Performance Tuning**
- Session 7: **Creating Standalone Applications**
- Session 8: **Streaming Overview**

Prerequisites & Expectations

- Reasonable familiarity and comfort with programming
 - There are many labs requiring programming
- Scala is the original Spark language
 - Course examples use Scala—labs may support other languages (Scala, Python, or Java)
 - We provide a quick Scala Primer
- Basic understanding of development environment
 - e.g. Linux or Windows
 - Command line navigation
 - Editing files
- **No previous Spark knowledge** is assumed
 - This is an introduction
 - Class pace based on the pace of majority of the students

Teaching Philosophy

- Clear coverage of concepts & fundamentals
- API: Enough to clearly understand how it works
 - But not so much that it obscures everything else — it's not a reference manual
- Highly interactive (questions, discussions, etc. are welcome)
- Hands-on (learn by doing)
- Current to recent releases of Spark
 - Which is evolving rapidly

About You And Me

- About you
 - Your Name
 - Your background (developer, admin, manager ...etc)
 - Technologies you are familiar with
 - Familiarity with Spark (scale of 1 – 4 ; 1 – new, 4 – expert)
 - **Something non-technical about you!
(favorite ice cream flavor / hobby...etc)**



Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

`val bi : BigInt = 1`

- Code fragments are the same, e.g. `bi + 2`
- We **bold/color** text for emphasis
- New terms are often introduced in bold italics
 - Filenames and paths are also in italics, e.g., *Catalog.scala*
- Notes are indicated with a superscript number ⁽¹⁾ or a star *
- Longer code examples appear in a separate code box — e.g.

```
object TestApp { // Basic Spark App (Scala)
    def main(args: Array[String]) {
        val sc = new SparkContext(
            new SparkConf().setMaster("local").setAppName("TestApp")
        )
        val totalWords = sc.textFile("file")
            .flatMap(l => l.split(" ")).count()
        println ("# lines : " + totalWords)
    }
}
```

Labs and Mini-Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
 - The detailed lab instructions **are separate** from the main lecture manual
- Setup zip files are provided with skeleton code for the labs
 - Students add code focused on the topic they're working with
 - There is a solution zip with completed lab code
- Mini labs are inlined in the book
 - They are usually short and demonstrate one concept

Lab 0.1: Setup Lab Environment

Different environments run the labs differently (many use a virtual machine). Your instructor will give directions for setting up your lab environment.

Session 1: Scala Primer

- Introduction
- Collections
- Functions/Methods
- Class/Object/Trait

Session Objectives

- Learn what Scala is, and why it's useful
- Understand Scala basics and syntax
- Be able to write and understand Scala code
- Be comfortable with functional programming in Scala
 - One of the core features used with Spark
- This lesson assumes you know basic programming and OO principles
 - It does not cover any of these basics — just Scala basics



**Introduction
Collections
Functions/Methods
Class/Object/Trait**

What is Scala?

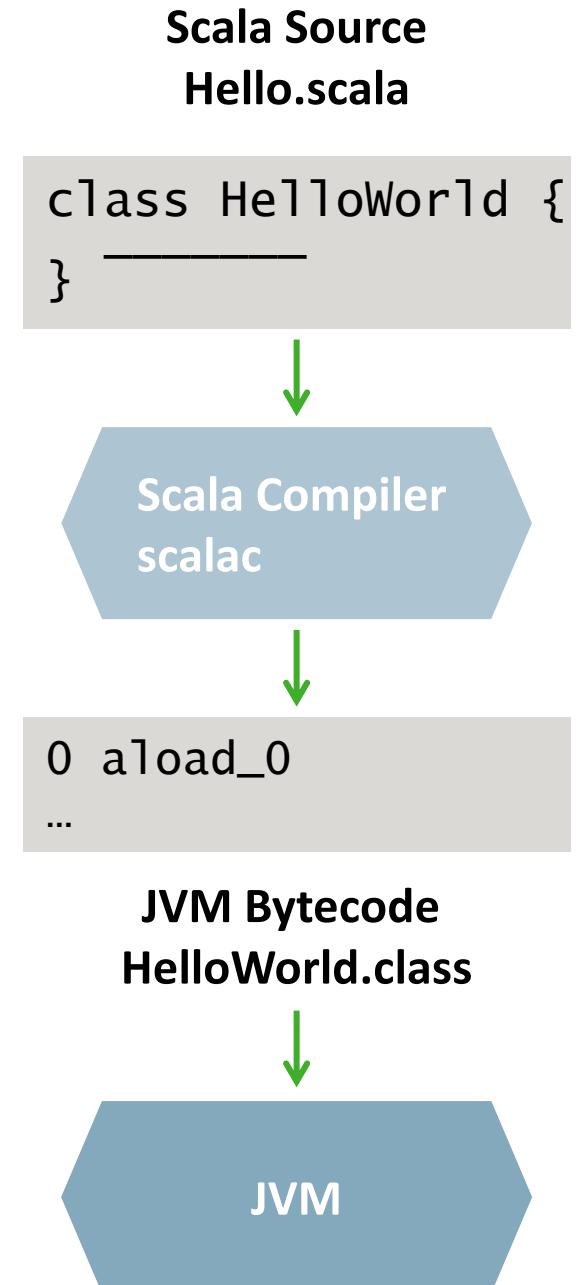
- An **object-oriented** programming language:
 - Pure OO language — everything is an object
 - Including numbers and functions
- A **functional** language
 - Functions are full class objects
 - They can be passed as arguments
- Interoperates with **Java**
 - Runs on Java Virtual Machine (JVM)
 - Scala programs can use Java libraries
- Very concise / dense language (Java is quite verbose)
 - Java : `System.out.println ("hello world");`
 - Scala : `println ("hello world")`

Why Scala and Spark?

- Spark designers had two core goals
- Work within **the Hadoop ecosystem**
 - JVM-based, so Spark had to run on the JVM
- Wanted **concise** programming interface
 - With strong support for **functional programming**
- **Scala** was the only language that provided this
 - Including ability to capture functions and ship them across the network ⁽¹⁾
 - Java 8 adds better support for functional programming — the Spark API supports this ⁽²⁾

Working with Scala

- Scala code/programs written/run similarly to Java programs
 - Compiled to bytecode, and run in a **JVM**
 - Java Virtual Machine
- Scala also has an **interpreter**
 - For learning and interactive coding
 - Also called the REPL⁽¹⁾
 - Good for ad-hoc querying in Spark
 - Used a lot in the course
- We'll show isolated code samples to start
 - And run things in the REPL⁽²⁾
 - We'll write standalone programs later



Scala Variables

- **var**: Defines a variable
- **val**: Defines a constant (immutable/read-only)
- Type is optional — Scala infers it if not supplied
 - Compile time checking—generally works well⁽¹⁾

```
var x = 5          // Variable (type inferred)
x = 7;           // Change the value (With optional semicolon)

val y = 10        // Constant
y=11             // error: reassignment to val

var d: Double = 5 // Explicit type declaration of Double

// Error-type mismatch. 1.1 is a double, not assignable to Int
val n : Int = 1.1

// Chained assignment doesn't work in Scala
var z = 1
z = x = 2 // This will give an error
```

The Scala Interpreter

- Start at command line via `scala` or `bin/spark-shell` for Spark ⁽¹⁾
 - Prints welcome text then a prompt
 - Type `:help` to get help, `:history [n]` to list recent history
 - Use arrow keys to scroll through command history
 - Emacs search supported (e.g. Ctrl-R to reverse search history)
 - **Tab completion** is well supported (try it!)

```
> :help
All commands can be abbreviated, e.g. :he instead of :help.
Those marked with a * have more detailed help, e.g. :help imports

:cp <path>           add a jar or directory to the classpath
:help [command]        print this summary or command-specific help
... Remaining detail omitted

// We will use > as the shell prompt to save space (not scala>)
> :history 2 // Get last two commands
339 :help
340 :history 2 // Get last two commands
```

Using the Interpreter

- Anything typed at the `scala>` prompt is read and evaluated
 - Response is the result of the evaluation given in form:
`name : Type = Value`
 - It then waits for more input ⁽¹⁾

```
> val y = 10
y: Int = 10

> y = 5
<console>:12: error: reassignment to val

> var z: Double = 5
z: Double = 5.0

> 1+2
res0: Int = 3

> var n:Int = 1.1
<console>:10: error: type mismatch;
```

Numeric / Boolean Types

- Scala Numeric Types: Byte, Char, Short, Int, Long, Float, Double, Boolean
 - Numeric values are objects — can call methods on them
- Helper classes add functionality (StringOps, RichInt ...)
 - This functionality can be used as if added to the class itself ⁽¹⁾
 - StringOps adds over 100 operations on strings
 - e.g. `"Hello".intersects("loop")` // Yields `lo`
 - Can also be written `"Hello" intersects "loop"`
 - `scala.Predef` provides easily accessed common definitions
 - e.g. `println` is alias to `Scala.console.println`
- Some special types
 - `Any`/`AnyVal`: Root of all types / Root of numerical types
 - `Unit`: Equivalent to Java void — no usable value

Operator Overloading

- Arithmetic operators look pretty normal, e.g. adding two Ints: a, b
a + b
 - But this actually calls the method **a.+**(b)**** ⁽¹⁾
 - This is a special case of calling **a.meth(b)** using a **meth b** ⁽²⁾
- Overloading's available for other types, and used in the Scala libs
 - e.g. for BigInt shown below ⁽³⁾
- Easy to add nifty new methods/operators, e.g.
 - 1 to 10 // Really 1.to(10)-numbers ranging from 1 to 10

```
> val bi : BigInt = 1
bi: BigInt = 1

> bi + 2 // Same as bi.+(2)
res14: scala.math.BigInt = 3

> 1 to 10 // We'll work with collections and ranges soon
res15: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6,
7, 8, 9, 10)
```

Looping

- Scala has fairly standard `for` and `while` loops⁽¹⁾
 - We illustrate the syntax below (using the values in `1 to 2`)
 - We use the built in `println()` function for output

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

// We'll see other ways to iterate over a collection later
> for (i <- 0 to b.length-1) println(b(i))
1
2

> for (cur <- b) println(cur) // Another way to iterate
1
2

> var i = b.length-1 // Result of assignment not shown here
> while (i >= 0) {
    println(b(i))
    i -= 1 }
2
1
```

Conditionals

- Fairly standard **if/else** statements
 - Supports same syntax as Java/C, as shown below
- An if/else has a value
 - The value of the if or else (depending on the condition result)
 - Can use the value in an assignment as shown at bottom ⁽¹⁾

```
> val i = 1
> val j = 2    // Results not shown

> if (i>j) {
|   println ("i greater")
| } else {
|   println("j greater")
| }
j greater

> val k = if (i>j) i else j    // Use value of if / else
k: Int = 2
```

match Expression

- match lets you select from a number of alternatives of form
 - case matchValue => expression-to-eval
 - The match can return a value (second example below)

```
> val monthNum = 2

> monthNum match {
|   case 1 => println("January")
|   case 2 => println("February")
|   case 3 => println("March")
|   case _ => println("Bad month") // _ = Default case
|
February

> val monthName = monthNum match {
|   case 1 => "January"
|   case 2 => "February"
|   case 3 => "March"
|   case _ => "Bad month"
|
monthName: String = February
```

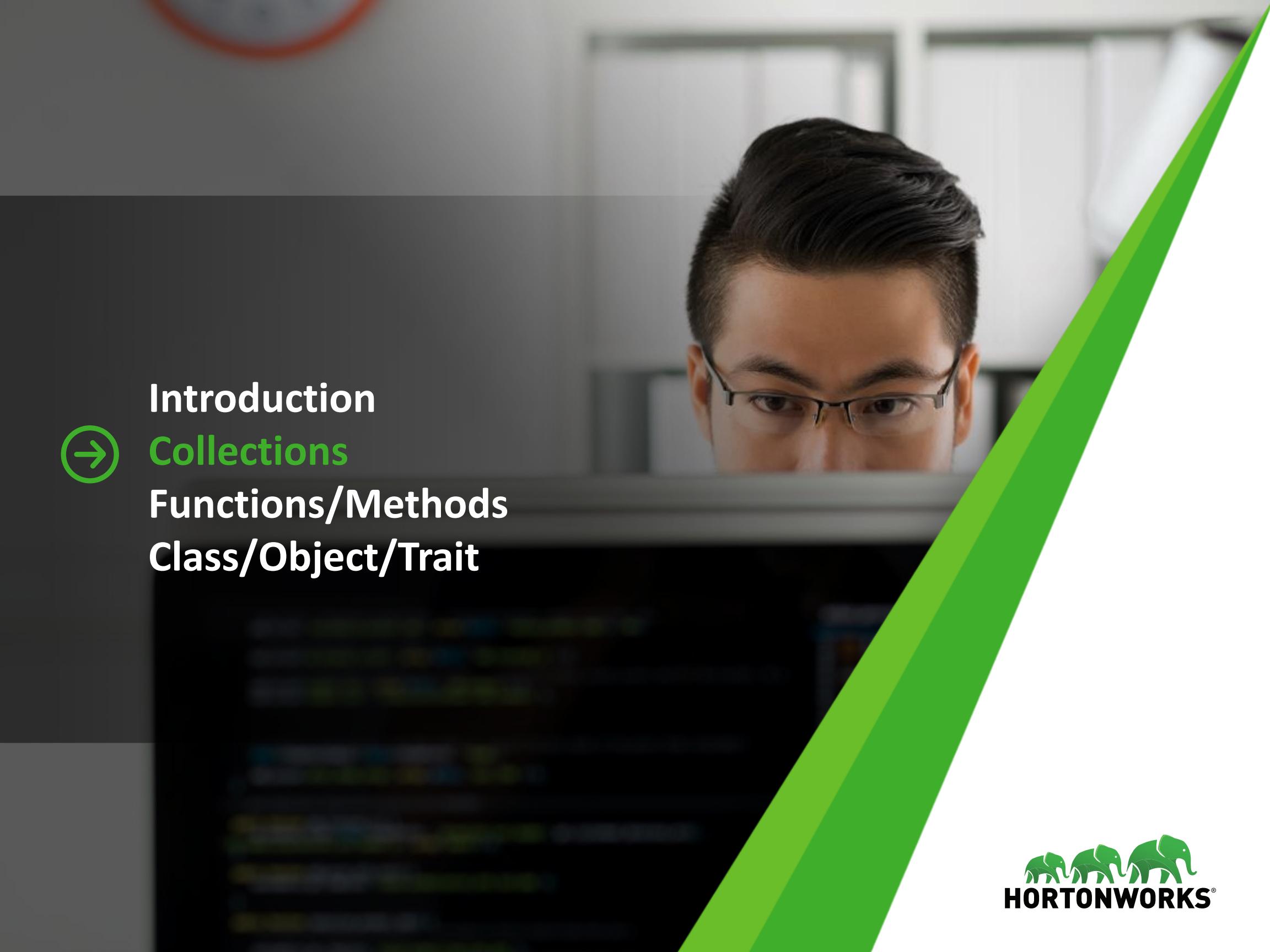
Lab 1.1: Start Scala Interpreter

There are several mini-labs in this session that require the Scala interpreter. Start the interpreter as per the separate instructions for your environment, then come back to the mini-lab

Play with Scala

Mini-Lab — 5 minutes

- Start up the Scala interpreter as in the previous lab
 - Type `:help` to get help
 - Review the `:paste` command (useful for pasting in code)
- See Notes for some samples of what we'll try here
- Declare some variables (including immutable ones)
 - Let Scala infer the type on some
 - Specify the type on some
- Use some looping constructs and conditionals
- Look (briefly) at the API docs (<http://www.scala-lang.org/api>)
 - To filter, type the name of a type in the search box in the upper left
 - Try looking at `Predef`, `StringOps`, `RichInt`

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

Introduction
→ **Collections**
Functions/Methods
Class/Object/Trait

Collection Classes

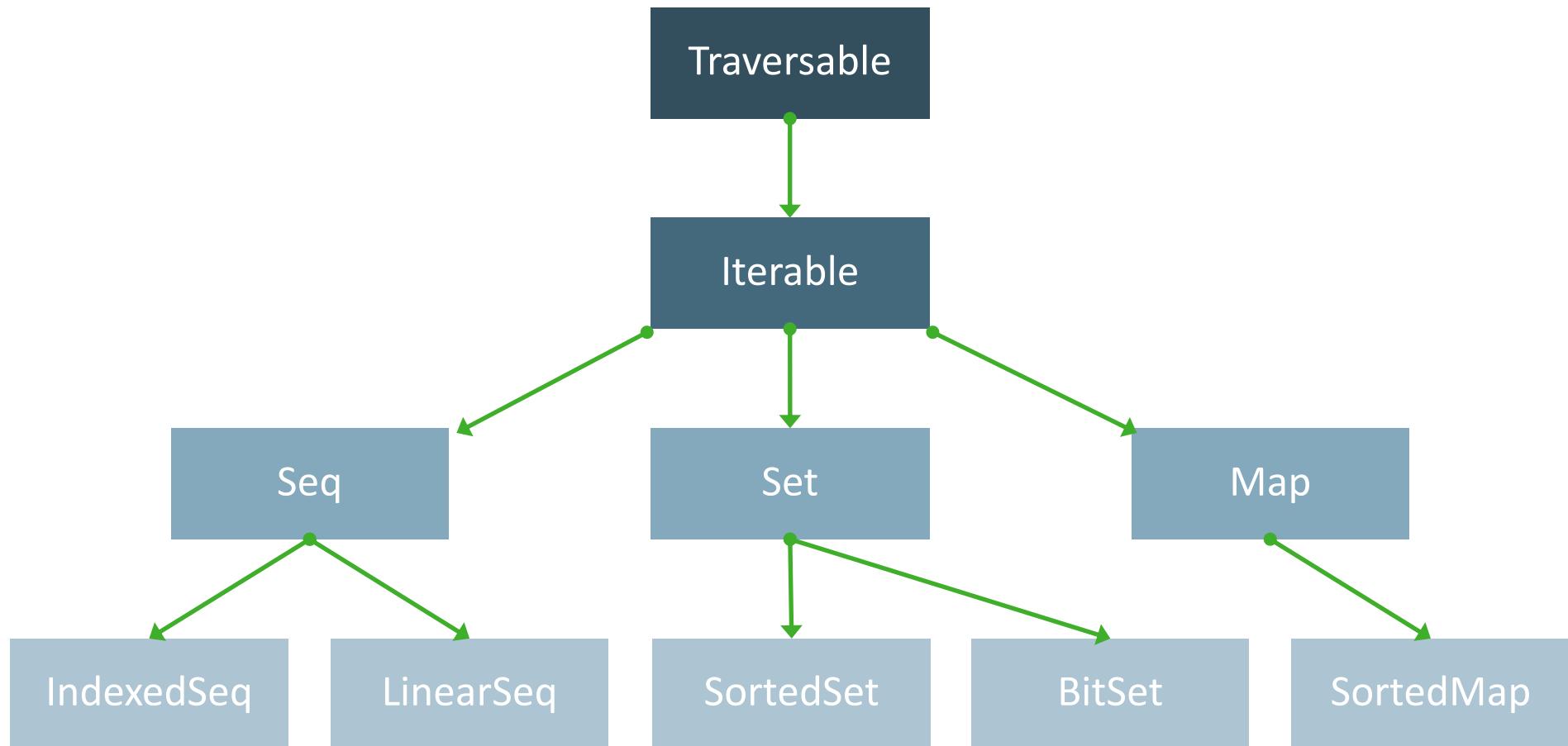
- Rich set of collection classes (Containers for things)
 - **Seq**: Sequences — ordered collection (**List**, **Vector**)
 - **Set**: Unordered collection with no duplicates
 - **Map**: Collection of key/value pairs
 - Won't explore details here — fairly standard
- Collections are parameterized by a type using [type]
 - And can hold instances of that type
 - The type is often inferred by contents, and not explicitly set ⁽¹⁾

```
// Create List and explicitly specify type
> val colors = List[String]("red", "green", "blue")
colors: List[String] = List(red, green, blue)

// Create List, let type be inferred
> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
```

Core Collection Types

- In package `scala.collection`
 - Traversable defines many common boilerplate methods



Creating Collection Instances

- Create an instance using the collection name
 - Followed by a list of elements in parentheses
 - Note that there are mutable (`scala.collection.mutable`) and immutable (`scala.collection.immutable`) versions
 - Scala defaults to immutable versions

```
> val pets = Set("dog", "dog", "cat") // Duplicates filtered out
pets: scala.collection.immutable.Set[String] = Set(dog, cat)

// Create map of pets with their count
> val petCount = Map("dog"->2, "cat"->1, "mice"->0)
petCount: scala.collection.immutable.Map[String,Int] = Map(dog -> 2,
cat -> 1, mice -> 0)

// Create mutable set
> val petsNow = scala.collection.mutable.Set("dog", "dog", "cat")
petsNow: scala.collection.mutable.Set[String] = Set(cat, dog)
```

Accessing Collections

- Capabilities vary based on the type of collection
 - All collections support **iteration** (shown later)
 - **Sequences** (`scala.collection.Seq`) support indexed access
 - **Maps** provide access by key
 - There are many different access options for each type

```
> val pets = Set("mouse", "dog", "cat")
pets: scala.collection.immutable.Set[String] = Set(mouse, dog, cat)
> pets.tail
res7: scala.collection.immutable.Set[String] = Set(dog, cat)

> val a = 1 to 5
a: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
> a(0)
res10: Int = 1

> val petCount = Map("dog" -> 2, "cat" -> 1, "mice" -> 0)
petCount: scala.collection.immutable.Map[String,Int] = Map(dog -> 2,
cat -> 1, mice -> 0)
> petCount("dog")
res11: Int = 2
```

Collection Methods

- Many methods — we list several that will be of use to us
 - **Map operations** (`map`, `flatMap`): Apply a function to each element to produce a new collection
 - `flatMap` is a combination of `map` and `flatten`
 - **Sub-collection retrieval** (`take`, `filter`, `slice`): Return sub-collection identified by index range or predicate
 - **Folds/reductions** (`fold`, `reduce`): Apply a binary operation to successive elements
 - Many, many more — see the documentation
- Many are common when using Spark
 - We'll give some examples shortly
 - Once we know a bit more about functions

Pairs and Other Tuples

- Tuples combine a fixed number of items together
 - e.g. a pair combines two items
 - Items can be of different types
 - Access them via notation `._n` (n is the 1-based index) ($1 \leq n \leq 22$)

```
> val pair = ("apple", 4)
pair: (String, Int) = (apple,4)

> pair._1
res149: String = apple

> println(pair._2)
4

> val wordCounts = Array(("apple",3), ("pear",2), ("grape",1))
wordCounts: Array[(String, Int)] = Array((apple,3), (pear,2), (grape,1))

> wordCounts(0)._1
res152: String = apple

> val edge = (1L,2L, 7)      // A triple
edge: (Long, Long, Int) = (1,2,7)
```

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

**Introduction
Collections**



Functions/Methods

Class/Object/Trait

Function Definition

- At bottom, we define the function max

- General form is

```
def function-name(arg1: Type ...) : ReturnType =  
    { function-body }
```

- The arguments always require a type spec
 - You can leave out the return type and braces (second example) ⁽¹⁾

```
> def max(x: Int, y: Int): Int = {  
    if (x > y) x  
    else y  
} // Returns the last expression evaluated(2)  
max: (x: Int, y: Int)Int  
  
> def max2(x: Int, y: Int) = if (x > y) x else y // Shorter version  
max: (x: Int, y: Int)Int  
  
> max (1,2) // Just call the function by name  
res79: Int = 2
```

Anonymous Functions / Function Literals

- You can define unnamed functions
 - Most often to pass to other functions for processing
 - Below, we use an anonymous function to process a collection
 - It prints out the square of each element in the collection
- General syntax is:
 - (argumentList) => { functionBody }
 - Can leave out the () if there is only one argument
 - Can leave out the {} braces if the body is one statement
 - Can leave out type of argument if it can be inferred ⁽¹⁾

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

> b.foreach( x:Int => { println(x*x) } ) // Anonymous function
1
4

> b.foreach( x => println(x*x) )           // Shorter form
```

Function Literal Syntax

- There are many choices in function literal syntax ⁽¹⁾
 - We show several choices for declaring literal functions below
 - The functions test if the input argument is even (they return a Boolean)

```
// Declare argument type, not return type
> val f = (i: Int) => { i%2==0 }
> val f = (_:Int)%2==0
f: Int => Boolean = <function1> // All variants have this signature

// Declare the argument and return type
> val f: (Int) => Boolean = i => { i % 2 == 0 }
> val f : Int => Boolean = i => i%2==0
> val f : Int => Boolean = _%2==0
```

Using filter and map

- We now know how to pass functions to the collection methods
 - So lets try them out
- **filter**: Select all elements satisfying the supplied predicate
 - Filter out those that don't
- **map**: Create new collection by applying supplied function to each element

```
'> val b = 1 to 4
b: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4)

// Filter out all odd values
'> b.filter(a => a%2==0)
> b.filter(f) // Same thing using function literal from earlier slide
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

// Create a new collection containing the squares of the values
'> b.map(a => a*a)
res3: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16)
```

Multiple Function Arguments and reduce

- Below, we define an anonymous function with two args
 - We also assign the function to a variable ⁽¹⁾
 - Below that, we use it to reduce an Int range ⁽²⁾
 - We also show a version inlining the anonymous function

```
> val sum = (a:Int, b:Int) => a+b
sum: (Int, Int) => Int = <function2>

> (1 to 3).reduce(sum)
res21: Int = 6

// Anonymous function – types of arguments inferred
> (1 to 3).reduce( (a,b) => a+b )
res22: Int = 6
```

_ (Underscore) Placeholders

- `_` (underscore) can be used for anonymous function parameters
 - As long as the parameters are used only once
- Below, we show examples without and with placeholders
 - Note how we use `_` twice in the anonymous function with two params
 - The first use means the first param, the second use the second param

```
> (1 to 4).filter( a => a%2 == 0)
res30: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 4).filter(_%2 == 0) // Placeholder equivalent – one param
res31: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 3).reduce( (a:Int, b:Int) => a+b ) // Anonymous function
res22: Int = 6

> (1 to 3).reduce( _+_ ) // Placeholder equivalent – two params
res113: Int = 6
```

Play with Collections and Functions

Mini-Lab — 10 minutes

- Do your work in the Scala interpreter as before
- Create some collection instances
 - Create a List instance, create a numeric Range instance using "to"
 - Store them in variables
 - Access some of the values (use the docs / tab completion as needed)
- Use some of the functions we've seen
 - Use **filter** to create a subset of one of your collections
 - Use **reduce** on your numeric Range to add up all the values
 - Do this without underscore placeholders and with underscore placeholders
 - **[Optional]** Use **map** on your numeric Range to map to a new collection containing the square of each value

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

**Introduction
Collections
Functions/Methods
→ Class/Object/Trait**

Defining a Class

- Class is a blueprint for objects (of course)
 - Defining the type in terms of methods and data
- Below, we define a simple class — we'll give more detail soon

```
> class Square (val size : Int) {  
|     def area : Int = size * size  
|     def perimeter : Int = size*4  
|     def display : Unit = {  
|         println("Size = " + size)  
|         println("Area = " + area)  
|     }  
| }  
defined class Square
```

```
> val sqr = new Square(6)  
sqr: Square = $iwC$$iwC$Square@478eb50
```

```
> sqr.area  
res145: Int = 36
```

```
> sqr.size  
res146: Int = 6
```

Class Def Details

- The below defines a constructor and a data member — size

- Accessible throughout the class
 - Also accessible as instance.size since we used val keyword

```
class Square (val size : Int) {
```

- Define accessor methods for area (a calculated value here)

```
def area : Int = size * size
```

- Called without any parentheses when defined this way
`sqr.area`

- We define a method that does some work (trivial work here)

```
def display : Unit = {  
    println("Size = " + size)  
    println("Area = " + area)  
}
```

Singleton Objects

- You can define singleton objects with the keyword **object**
 - One and only instance — accessed via the object name
 - e.g. Counter.currentCount
 - Basically a container for non-instance related code

```
> object Counter {  
|   var count = 0  
|  
|   def currentCount(): Long = {  
|     count += 1  
|     count  
|   }  
| }  
defined module Counter  
  
> Counter.currentCount  
res153: Long = 1  
  
> Counter.currentCount  
res154: Long = 2
```

Traits

- Traits are types defining fields and methods that you can mix into your classes
 - It defines a role played by objects
 - It may have some implementation, but doesn't have to
- Below we define a trait with three accessor methods
 - But no shared implementation

```
> trait Geometric {  
|   def height : Int  
|   def area   : Int  
|   def perimeter : Int  
| }  
defined trait Geometric
```

Using a Trait

- Use a trait in the class def
 - Via the **extends** or **with** keywords ⁽¹⁾

```
> class Square (val size : Int) extends Geometric {  
|     def area : Int = size * size  
|     def height : Int = size  
|     def perimeter : Int = size*4  
|     def display : Unit = {  
|         println("Size = " + size)  
|         println("Area = " + area)  
|     }  
| }  
defined class Square  
  
> val sqr = new Square(6)  
sqr: Square = $iwC$$iwC$Square@4f90cf86  
  
> val geom : Geometric = sqr  
geom: Geometric = $iwC$$iwC$Square@4f90cf86  
  
> geom.area  
res155: Int = 36
```

Case Class

- Plain, immutable types that are initialized via constructor only
 - Initialization simpler than regular class
 - Can use pattern matching on the class itself
 - Implicitly defines an equality operator (value based)
 - We illustrate definition and some usage below

```
> case class Person (name: String, gender: String, age: Long)
defined class Person

> val p1 = Person("John", "M", 45)
> val p2 = Person("Mary", "F", 50)
> val p3 = Person("John", "M", 45)

> if (p1==p2) "Equal" else "Non-Equal"
res0: String = Non-Equal

> if (p1==p3) "Equal" else "Non-Equal"
res1: String = Equal
```

Packages and Imports

- Packages organize code and control namespace
 - Package names are hierarchical and dot-separated
 - e.g. `scala.math`
 - Packages generally hold related types
- Declare a package with a package statement, e.g.
`package com.mycompany.time`
- Import type names from other packages with import
`import com.mycompany.time._ // Import all types`
`// Import Timepiece only`
`import com.mycompany.time.Timepiece`

Packages and Imports Illustrated

- Below we define and use some types in different packages

```
// File Timepiece.scala
package com.mycompany.time;
import scala.math._;
trait TimePiece { /* ... */ }
```

```
// File AlarmClock.scala
package com.mycompany.time;
import scala.math._;
class AlarmClock extends TimePiece { /* ... */ }
```

```
// File Store.scala
package com.mycompany.products;
import com.mycompany.time.TimePiece

class Store {
  def timepieces : Array[TimePiece] = new Array[TimePiece](100)
}
```

A Standalone Scala Program

- Is an object with a main method, as shown below
 - If the main method has a parameter defined as shown in the example, it will receive the program arguments
- You compile this with the Scala compiler scalac
 - > **scalac SimpleApp.scala**
- You run it with the scala command
 - > **scala SimpleApp**

```
object SimpleApp {  
    def main(args: Array[String]) { // args holds command line args  
        for (cur <- args) println(cur) // Print out the args  
    }  
}
```

Session Summary

- Scala is a **functional** as well as OO language
- The **Scala Interpreter** is a quick and easy way to use Scala
 - The **REPL** (Read-Evaluate-Print Loop)
 - Helps in learning and testing code
 - Load external file with **:load <file>**
- All Scala data is **typed**
 - As specified in classes with methods and data
 - Declare singletons using the **object** keyword
 - **No** "primitive types" like Java and C
 - Scala can often **infer** a type (type inference)
- Scala is **garbage collected**
 - Automatically reclaims unused objects (never done manually)

Session Summary

- Values (`val`) are immutable
 - More stable than variables (`var`)
- A tuple is an ordered container of values of multiple types
 - Accessed by `_n` syntax (e.g. `myTuple._1`)
 - Can't iterate through tuple members
- Many useful **collection types** in Scala
 - Set, List, Map
 - Many useful functions, e.g. `filter`, `map`, `reduce`
- The **Unit** type is used for functions returning no data
 - Same as void in Java and C-like languages

Scala Resources

- Book: [Programming in Scala](#)
 - By Martin Odersky (Scala language Designer), et. al
 - 1st edition online at: <http://www.artima.com/pins1ed/>
- Scala website: www.scala-lang.org
 - The place for all things Scala, including API docs and tutorials
 - [Scala CheatSheet](http://docs.scala-lang.org/cheatsheets/): <http://docs.scala-lang.org/cheatsheets/>

A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt and blue jeans. He is standing in what appears to be a workshop or industrial setting, with pipes and equipment visible in the background. He is gesturing with his hands while speaking, holding a small dark object in his left hand. A green diagonal bar runs from the bottom right corner across the slide.

Session 2: Introduction to Spark

- Introduction
- Spark in the Big Data Ecosystem
- First Look at Spark

Session Objectives

- Understand the needs that Spark addresses
- Be familiar with Spark's capabilities and advantages
 - And how they fit into the Big Data landscape
- Gain an understanding of a basic Spark installation



Introduction

Spark in the Big Data Ecosystem

First Look at Spark

Current Big Data Processing Challenges

- Processing needs outpacing 1st generation tools
- **MapReduce** (MR) / Hadoop has major limitations
 - MR **performance bottlenecks**
 - **Batch processing** doesn't fit needs
 - Programming can be **difficult and verbose**
 - MR add-ons attempt to address these
 - But many are inherent in the core architecture
- **Spark**: 2nd generation tool addressing these needs
 - One of the premier next generation technologies
 - Let's look at it

What is Spark?

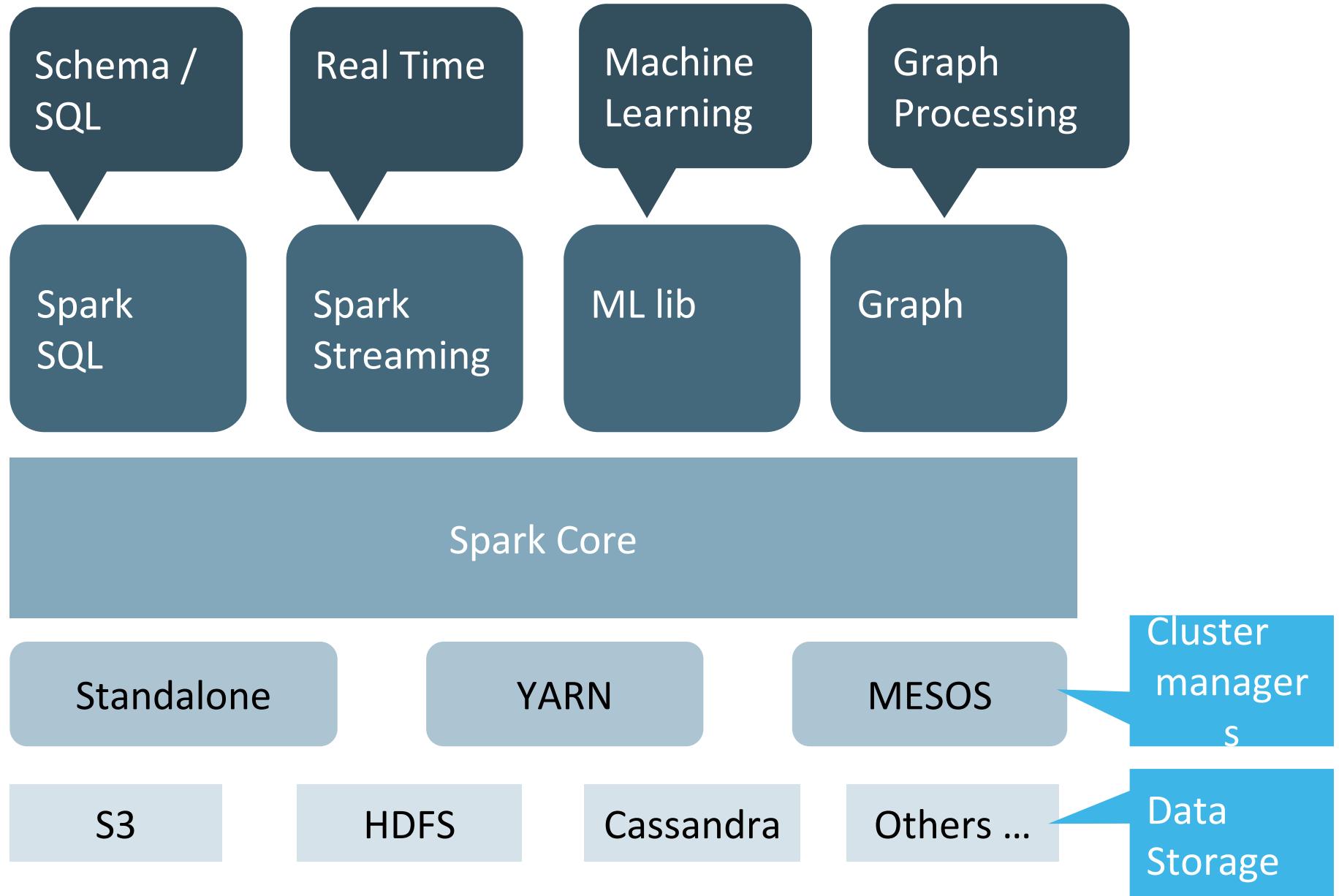
- Open source **cluster computing engine**
 - **Very fast:** In-memory ops 100x faster than MR
 - On-disk ops 10x faster than MR
 - **General purpose:** MR, SQL, streaming, machine learning, analytics
 - **Compatible:** Clusters run on Hadoop/YARN, Mesos, standalone
 - Works with many data stores: HDFS, S3, Cassandra, HBase, Hive, ...
 - **Easier to code:** Word count in 2 lines
- **Spark's roots:**
 - Came out of Berkeley AMP Lab
 - Now top-level Apache project
 - Version 1.6 / Jan. 2016
 - Version 2.0 / July 2016, Version 2.1 / Dec. 2016

“First Big Data platform to integrate batch, streaming and interactive computations in **a unified framework**” – stratio.com

Spark Eco-System

- **Apache** top level project
 - Very active, fast growing community ⁽¹⁾
- **Databricks**: supporting / developing Spark
 - Founded by Spark's creators
 - Employs the most active committers
- **Hadoop** vendors (Hortonworks / Cloudera)
 - Include Spark in their distributions
- **Spark packages** repository: Community index of Spark add-ons
 - <http://spark-packages.org/>

Spark Illustrated



Spark Structure

- **Data Storage:** Pluggable data storage systems
 - Integrates with HDFS, S3, Cassandra DB and more
- **Cluster Manager:** Manages distributed node clusters
 - Provides the distributed execution environment
 - Works with Mesos, Yarn, and its own standalone manager
- **Spark Core:** Distributed Compute Engine
- **Spark Components:** Modules layered on top of core
 - Specialized functionality — e.g. Spark SQL for SQL querying
- **Structure changes** in Spark 2.0+
 - We'll see when we look at DataFrames/Datasets

Spark Core

- **Building blocks** for distributed compute engine
 - Task schedulers and memory management
 - Fault recovery (recovers missing pieces on node failure)
 - Storage system interfaces
- Defines **Spark API** and **data model**
- **Core Data Model: RDD** (Resilient Distributed Dataset)
 - Distributed collection of items
 - Can be worked on in parallel
 - Easily created from many data sources (Any HDFS InputSource)
- **Spark API**: Scala, Python, Java, and R
 - Compact API for working with RDD and interacting with Spark
 - Much easier to use than MapReduce API

Working with Spark

- Spark can run on:
 - Local machine for easy development
 - On clusters on premises or in cloud for production
- **Spark shell** supports interactive computing
 - Work with Spark from command line
 - Easy prototyping and ad-hoc exploration of data
 - Can connect to any cluster
- **Spark API:** Compact API for working with Spark
 - Multiple languages: Scala, Python, Java, and R (and more)
 - All provide same capabilities
 - Much easier to use than MapReduce API

Spark Components

- **Spark SQL/DataFrames/Datasets**: Structured data
 - Supports SQL and HQL (Hive Query Language)
 - Data sources include Hive tables, JSON, CSV, Parquet
- **Spark Streaming**: Live streams of data in real-time
 - Low latency, high throughput (1000s events / sec)
 - Log files, stock ticks, sensor data / IOT (Internet of Things) ...
- **ML/MLlib**: Machine Learning **at scale**
 - Classification/regression, collaborative filtering ...
 - Model evaluation and data import
- **GraphX/GraphFrames**: Graph manipulation, graph-parallel computation
 - Social network friendships, link data, ...
 - Graph manipulation and operations and common algorithms

Spark : 'Unified' Stack

- Spark components support **multiple programming models**
 - Map / Reduce style batch processing
 - Streaming / real-time processing
 - Querying via SQL
 - Machine learning
- All modules are **tightly integrated**
 - Facilitates rich applications
- Spark can be the only stack you need !
 - No need to run multiple clusters (Hadoop cluster, Storm cluster, etc.)

Spark Optimizations for Fast Computing

- **Pipeline** multiple operations
 - Plus lazy evaluation
 - Postpones any processing until really needed to generate results
 - Produces significant optimizations
 - e.g. You need (only) the first 10 results of a computation
 - Spark can halt processing of entire pipeline once those are produced
- **Cache intermediate results** in memory
 - Once data is initially read, it stays in memory
 - Dramatic performance increase for pipelined / iterative operations
 - No Disk I/O if all data (for a given node) fits in memory

Spark Use Cases

Data Scientist	Data Engineer
<ul style="list-style-type: none">• Ad hoc exploration• Analyze data, build models• Come up with ‘insights’ (recommendations, etc.)	<ul style="list-style-type: none">• Build data infrastructure (pipelines, processing, etc.)• Productize ideas from Data Scientists (incorporate recommendation within online store)
Shells are great for ad-hoc analysis (Scala & Python)	Shells are good way to debug / test programs
Can re-use rich libraries in Scala / Java / Python / R	Can use already familiar languages
Interactive Development (not waiting minutes / hours for runs)	ditto
Unified stack (program within one stack)	ditto

Spark Use Case Examples

- **Facebook**: 60 TB+ production use case—feature evaluation ⁽¹⁾
 - Replaced existing Hive implementation
 - Reduced overall job time from 60 hours to 10 hours
- **Yahoo**: News personalization ⁽²⁾
 - 120 line Scala program with MLlib replaced 15,000 lines of C++
 - Took 30 min to run on 100 million record sample data set
- **Alibaba**: Discover e-commerce communities (forming around a product)
- **Netflix**: Analyze streaming events (450 Billion events / day)
 - Personalization / Recommendations

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

Introduction

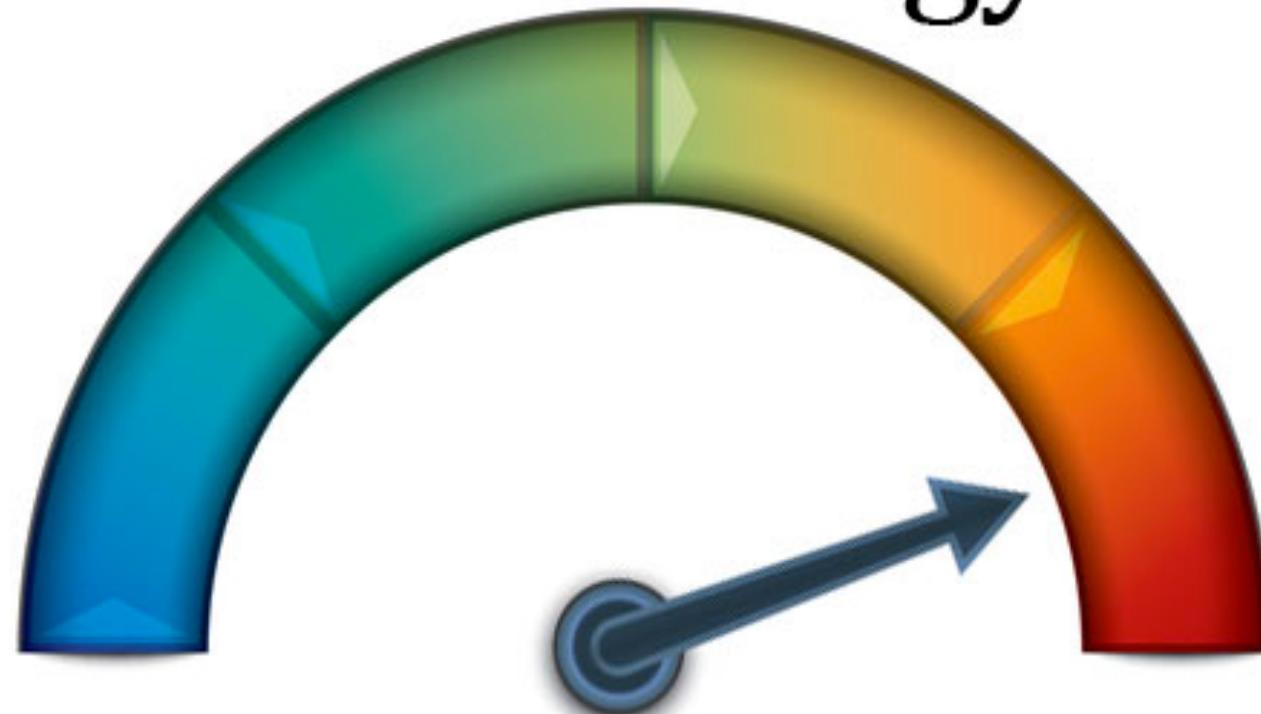


Spark in the Big Data Ecosystem

First Look at Spark

Where Does Spark Stand?

Technology



Hype-o-meter

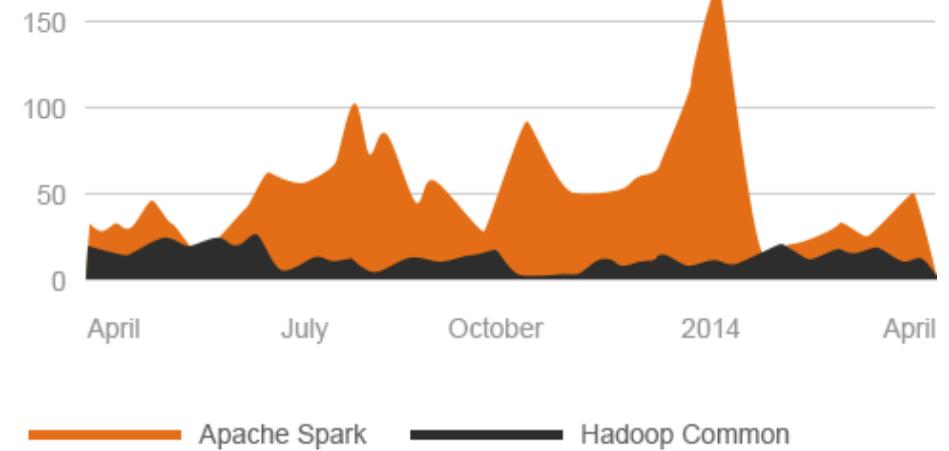
Used More and More

Apache Spark Job Trends



MOST ACTIVE BIG DATA PROJECT IN 12 MONTHS

Commits to master, excluding merge commits.

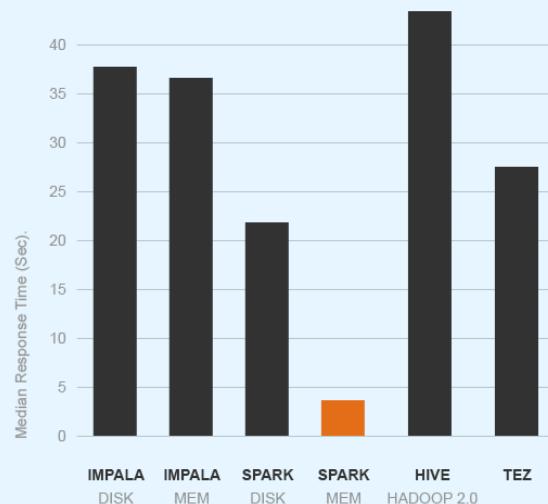


Faster Performance—Smaller Code Size

BENCHMARK

SCAN QUERY

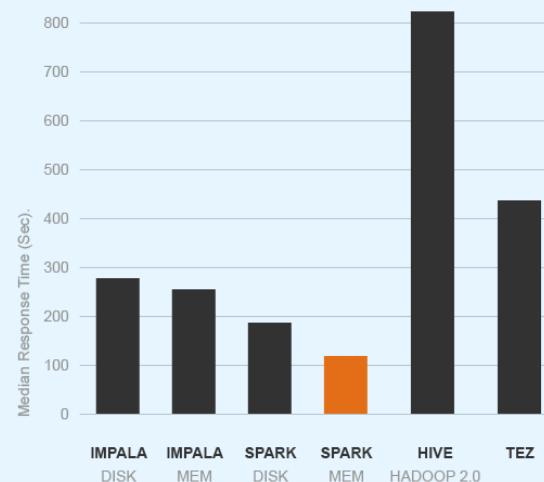
Query 1C - 89,974.976 results



Source: Amplab Uc Berkeley

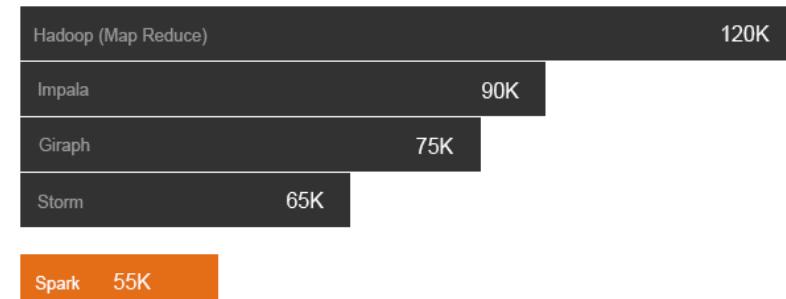
AGGREGATION QUERY

Query 2C - 253,890.330 groups



7X LESS CODE SIZE

350K code lines comparing to 55k.



Comparison With Hadoop/MapReduce

Hadoop and MapReduce	Spark
Distributed Storage + Distributed Compute	Distributed Compute Only
MapReduce framework only	Generalized computation
Usually data on disk (HDFS)	On disk / in memory (Tachyon)
Not ideal for iterative work	Great at Iterative workloads (machine learning, etc.)
Batch process	<ul style="list-style-type: none">- Up to 2x – 10x faster for data on disk- Up to 100x faster for data in memory
Java supported fully. Other language support possible	Compact code Java, Python, Scala, R supported
No equivalent of shell	Shell for ad-hoc exploration

Spark vs. MapReduce

- Spark is **easier** than MapReduce
 - Simpler code, less code
- **Friendlier** for data scientists / analysts
 - Interactive shell
 - Fast development cycles
 - Ad hoc exploration
 - Specialized components (Graphs, Machine Learning, ...)
- API supports multiple languages
 - Java, Scala, Python, R
- Great for small (Gigs) to medium (100s of Gigs) data

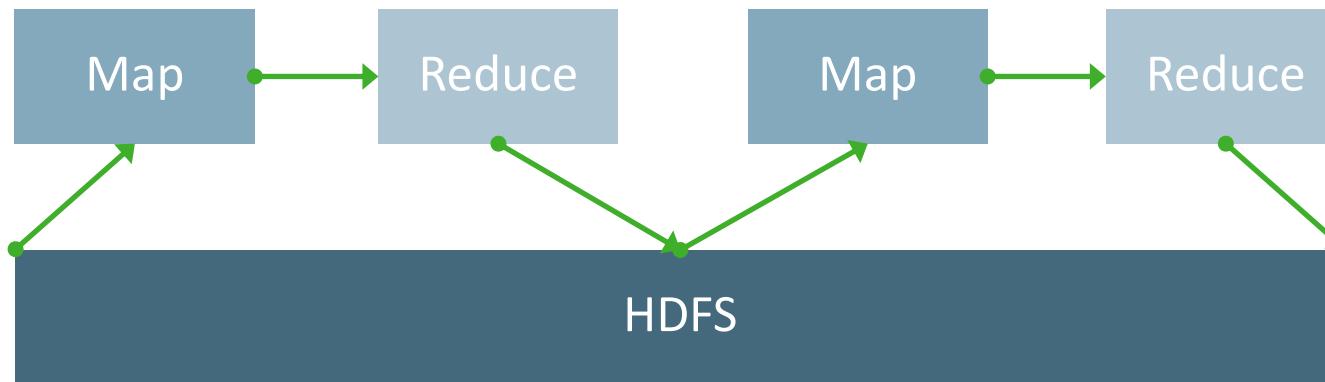
Spark: World Record Large-Scale Sort

- Sorts 100TB in 23 min on 206 nodes
 - Wins Daytona GraySort 2014 contest ⁽¹⁾
 - Vs. Hadoop record of 72 min. on 2100 nodes

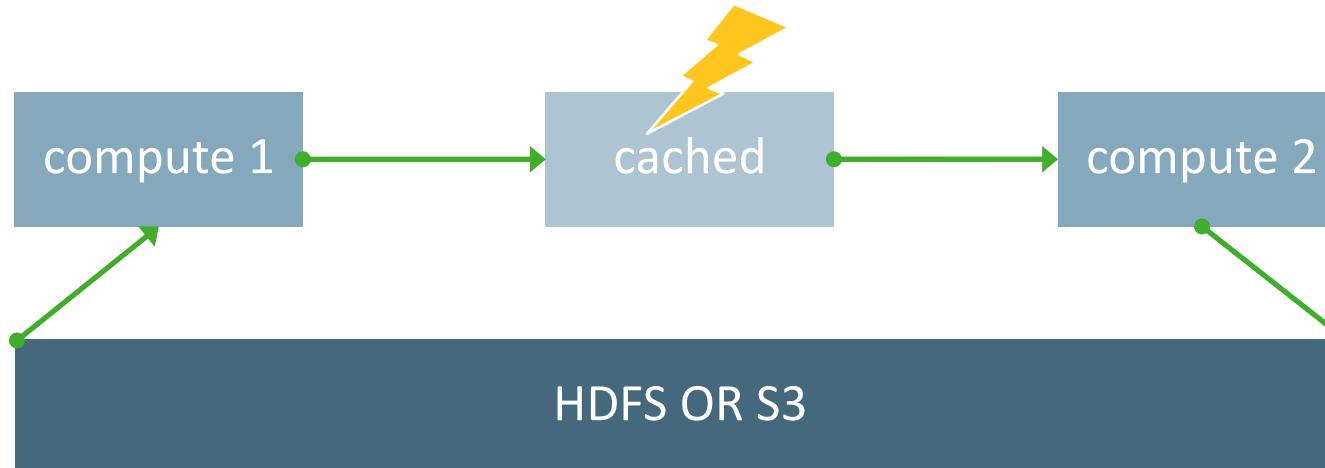
	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Spark: A Better Fit for Iterative Workloads

Hadoop Map Reduce

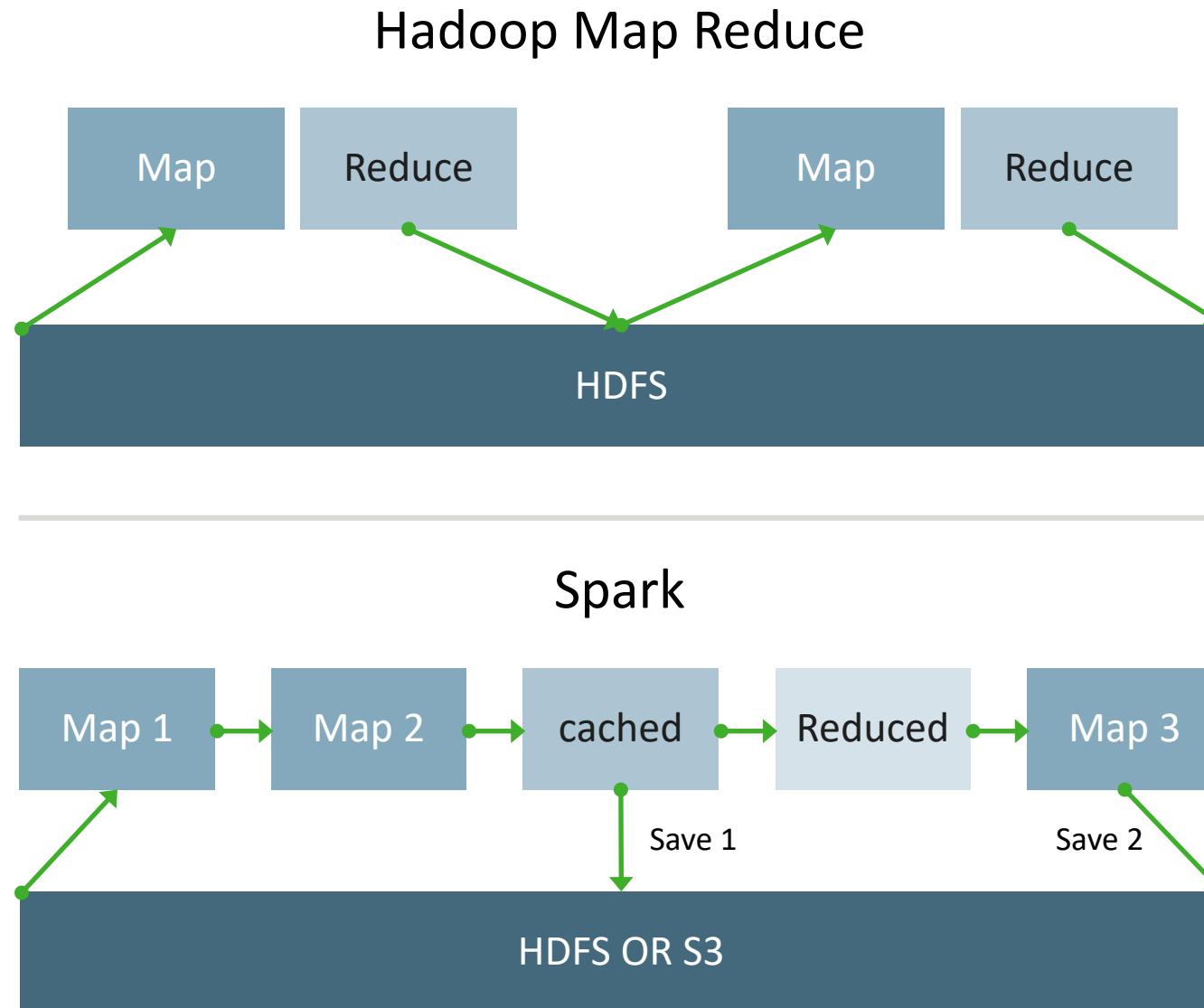


Hadoop Map Reduce



Spark: A More Generic Programming Model

- This reduces dependencies on the storage system (disk)



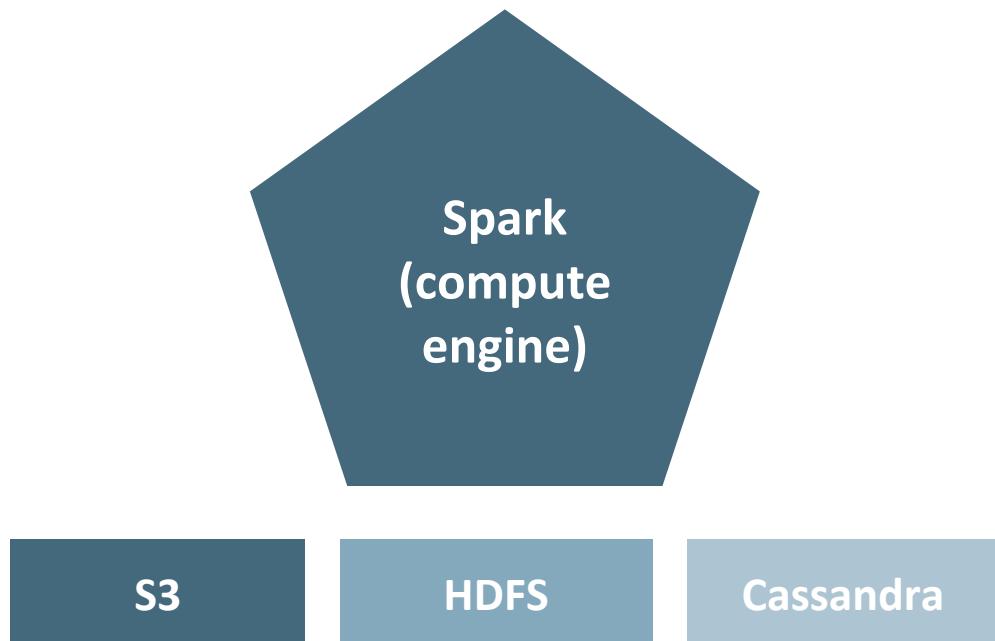
Hadoop vs. Spark

- Spark's unified stack supports almost all needs

Use Case	Hadoop	Spark
Storage	HDFS	<ul style="list-style-type: none">• HDFS / S3 / Cassandra for now• Tachyon (memory-centric distributed storage)
Cluster Manager	YARN	YARN or Mesos
Batch processing	MapReduce (Java, Pig, Hive)	Spark MR
SQL querying	Hive	Spark SQL (can also query Hive/HQL)
Stream Processing / Real Time processing	Storm	Spark Streaming
Machine Learning	Mahout	Spark MLlib
Real time lookups	NoSQL (HBase, Cassandra, etc)	No Spark component. But Spark can query data in NoSQL stores.

Is Spark Replacing Hadoop?

- Spark currently runs on Hadoop / YARN — Complimentary
 - Can be seen as generic MapReduce
- Spark is great if data fits in cluster memory (few hundred gigs)
 - And still very good if data spills over to external storage
- Spark is 'storage agnostic'



Spark @ Large Scale

- Cluster
 - 8000 nodes
 - 400 TB+ data
 - @ Tencent (Social network in China)
- Single job
 - 1 PB
 - Image processing @ Alibaba
- Streaming
 - 1 TB / hour
 - Analyze medical images @ Janelia farm
- 2014 Sort record mentioned earlier



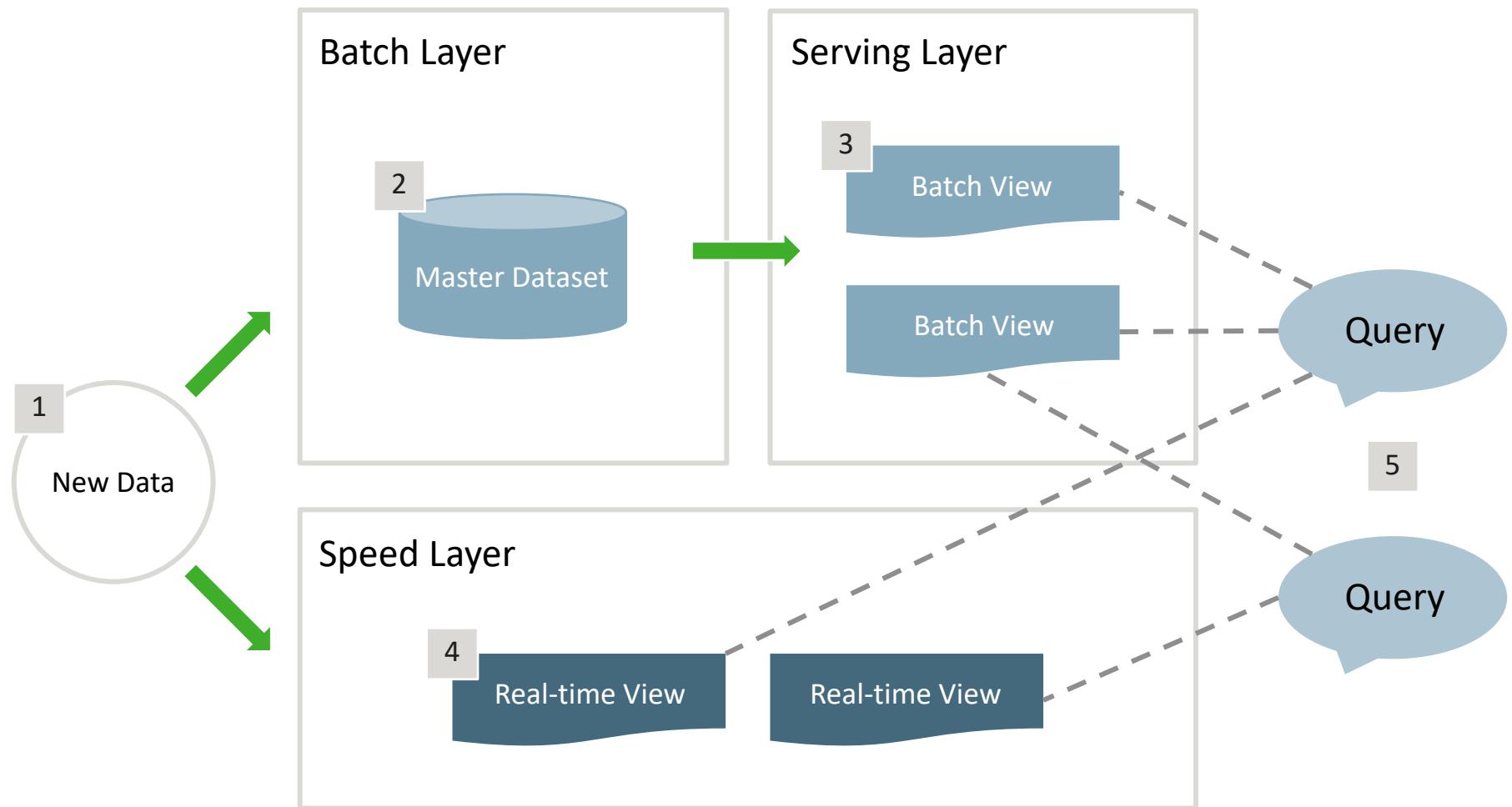
Spark New and Noteworthy

- Data Science
 - DataFrames / Datasets (inspired by R / Python)
 - Machine Learning pipelines (inspired by SciKit Learn)
 - R language support (!)
- Platform
 - Unified access across multiple datasources (HDFS/noSQL, etc.)
 - Spark packages : community libraries
 - <http://spark-packages.org>
- Core / Engine
 - **Tungsten**: Off-heap memory
 - **Succinct**: Direct queries on compressed data
 - Visualization / debugging tools

Spark for Lambda Architectures (LA)

- LA: Design Pattern for data infrastructure
 - Addresses needs of real-world, scalable applications ⁽¹⁾
 - Came out of Twitter (Nathan Marz, et al)
 - Key points below (Spark is a great fit for these)
- **Fault tolerant** to hardware failures and human error
- Layered approach incorporates batch and real-time needs
 - **Batch Layer**: Manages master data set
 - Immutable, append-only, raw data
 - Also pre-computes batch views
 - **Serving Layer**: Indexes batch views for low-latency queries
 - **Speed Layer**: Accommodates requests needing low latency
 - Recent data only using fast and incremental algorithms

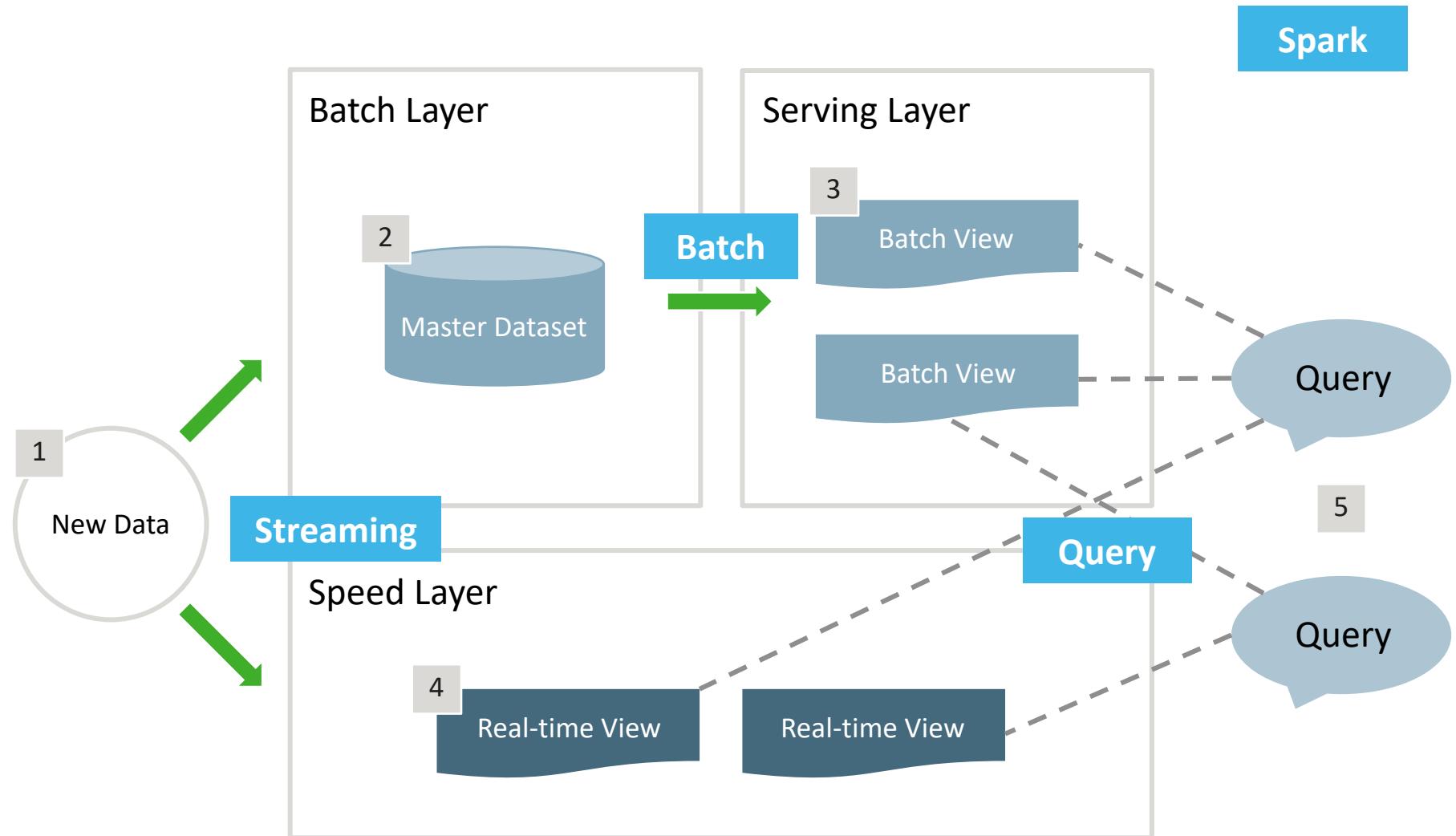
Lambda Architecture Illustrated



Spark and LA

- LA requires both batch and real-time components
 - Challenging to architect — often require multiple systems
 - Not good in terms of implementation/maintenance ⁽¹⁾
 - **Until Spark!**
- Spark supports LA in a single system, e.g. mining log files ⁽²⁾
 - Use core Spark functionality to process data (batch)
 - e.g. Mining and storing log events with ERROR conditions
 - Create / store aggregates that contain useful views (batch)
 - e.g. Number of errors in a day
 - Use **Spark streaming** for real-time views of same data (speed)
- Spark is arguably the **best platform for LA**
 - Easily supporting batch and stream in same app **at scale**

Spark + Lambda Architecture



A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen displays several lines of code in a dark-themed editor. The background is slightly blurred, showing what appears to be a server rack or a stack of books.

Introduction

Spark in the Big Data Ecosystem



First Look at Spark

Apache Spark Project / Download

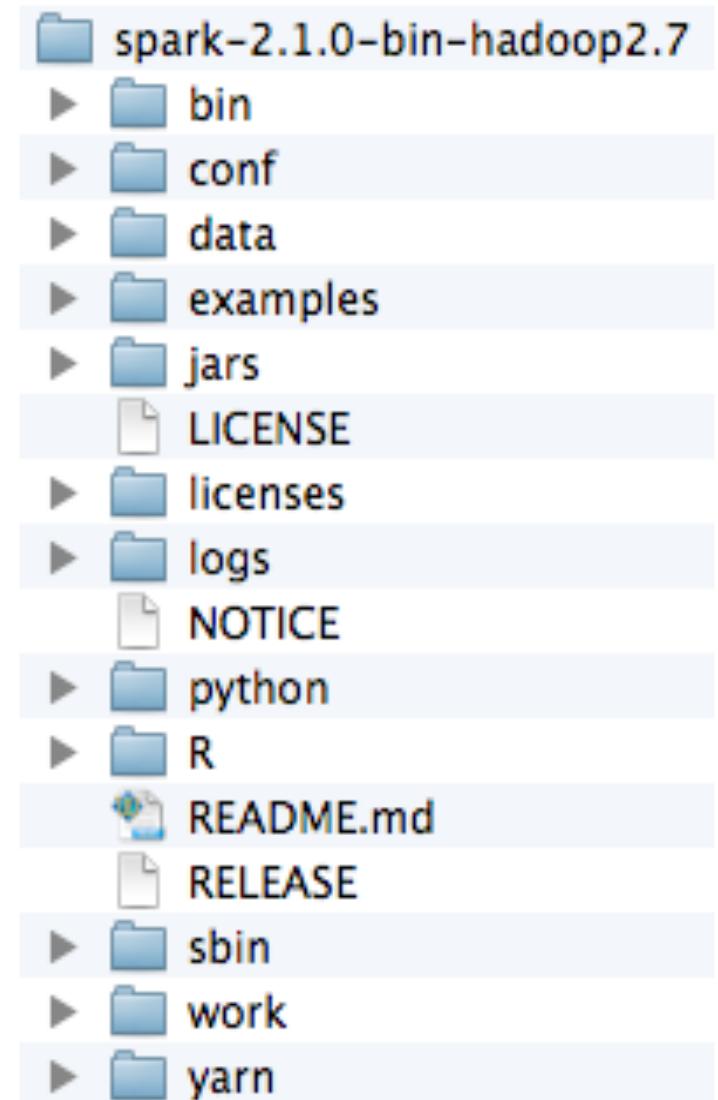
- Top level Apache project (open-source)
 - <http://spark.apache.org/>
- Written in Scala
 - Runs on the Java Virtual Machine (JVM)
- Binary tarballs available on the project website
 - Contain Spark/Scala libraries, interactive shells, run scripts
 - Optionally with Hadoop distribution (several are supported)
- Can be used in multiple clustering modes
 - Standalone: We'll start with this first
 - Using external cluster manager
 - Hadoop/YARN and Mesos

Spark Releases—Every Quarter !

- March 2015 : Spark 1.3
 - DataFrames (easy data access), MLlib new algorithms
 - Streaming : Kafka direct access (2x faster than previous implementation)
- June 2015 : Spark 1.4
 - SparkR : R & Spark !, DAG Visualizing tool, ML pipelines
 - DataFrame enhancements, Streaming improvements for Kafka & Kinesis
- Sept 2015: Spark 1.5
 - Core / Performance improvements, Streaming flow control (backpressure)
 - ML : improvements, SparkR : more R integration
- Jan 2016 : Spark 1.6
 - New **Dataset API**, Spark ML **pipeline persistence**
 - Automatic memory configuration, Spark Streaming optimized state storage
- July 2016 : Spark 2.0 / Dec. 2016 : Spark 2.1
 - **API Improvements**: Unified **DataFrames/Dataset**, **SparkSession**, and more
 - **Structured Streaming** for easier, more efficient streaming computation

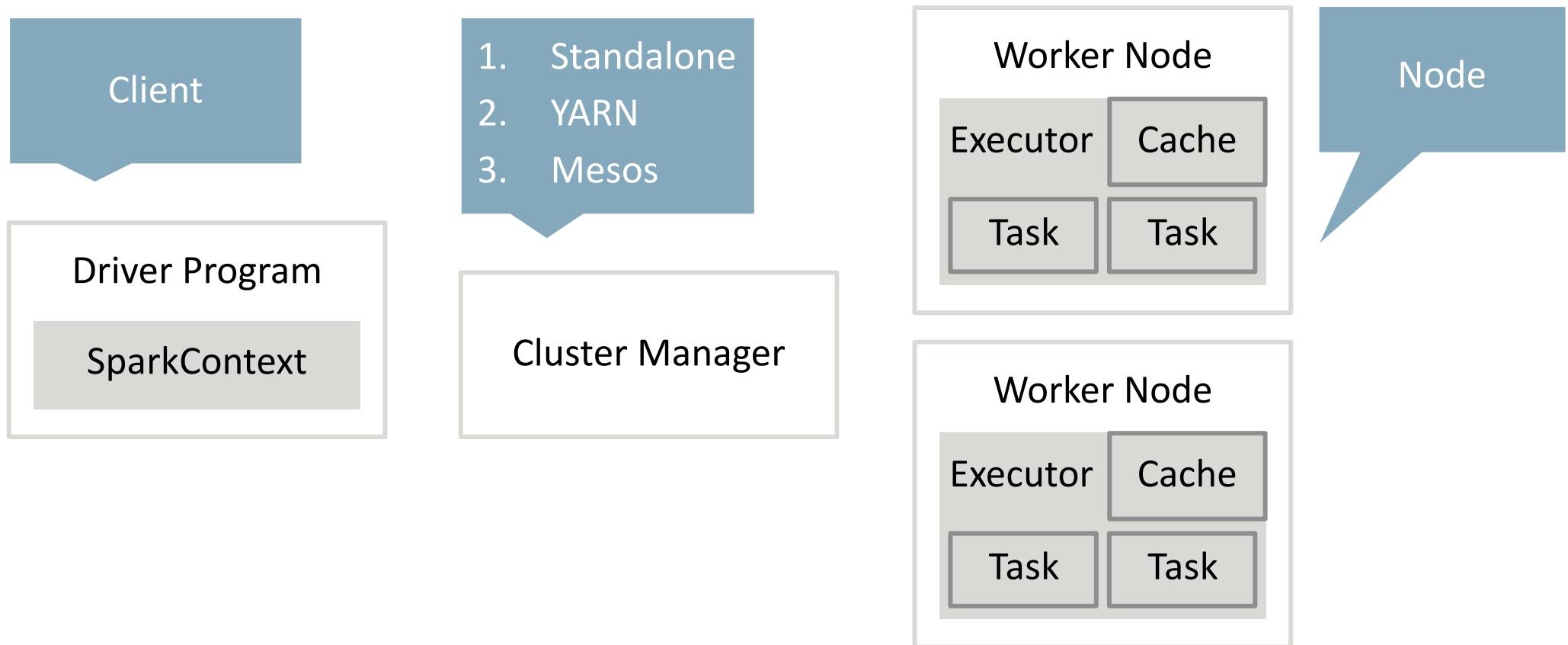
System Requirements and Installation

- Java 7+ (Spark 1.5+), Java 8+ (Spark 2.0)
- Unzip Spark tarball—get folder structure:
 - **bin**: executables
(Spark/Python shells, utilities, etc.)
 - **conf**: Configuration templates
(e.g. spark-env.sh.template)
 - **data**: Sample data files
 - **ec2**: Scripts/files to run on EC2
 - **examples**: Example programs
(Scala, Python, Java)
 - **lib**: Jar files
 - **logs**: Log files
 - **python**: Python source
 - **R**: SparkR (R frontend)
 - **sbin**: Shell scripts



Runtime Architecture

- **Cluster of worker nodes** performs work
- **Cluster manager** manages the worker nodes
- **Driver program** kicks off tasks and sends code to them



Running Spark

- We'll start with two simple ways to run Spark
- Use the **standalone cluster manager** to start a cluster
 - And start all nodes on one machine
 - `<spark>/sbin/start-all.sh` starts up a small cluster (a master and one worker) — `stop-all.sh` stops them
 - May require a bit of setup for ssh access — even on one machine
- Use an interactive shell
 - Ships with Scala and Python shells
 - **Scala**: `<spark>/bin/spark-shell`
 - **Python**: `<spark>/bin/pyspark`
 - By default, these use an **embedded** Spark instance
 - Can configure to connect to a cluster

Standalone Cluster Manager

- Simple standalone cluster support
 - No need for other manager (e.g. YARN)
 - Easy setup/startup for development
- Easy startup via provided launch scripts
 - `<spark>/sbin/start-all.sh` starts a manager (master) and workers
 - Default: One worker, one slave on local machine
 - Or based on configuration files (next slide)
 - Provides master Web UI at `hostname:8080`
- Other launch scripts include
 - `start-master.sh`: Start master on this machine
 - `start-slaves.sh`: Start workers on machines listed in conf/slaves
 - `stop-*.*sh`: Variations to stop master, slaves, or all

Spark Configuration

- *conf/log4j.properties*: Logging configuration
 - log4j.properties.template is starter file ⁽¹⁾
- *conf/slaves file*: List of worker hosts (default localhost)
 - slaves.template is starter file
- *conf/spark-env.sh*: Overall configuration
 - IP addresses, ports, memory, cores, etc.
 - *spark-env.sh.template* starter file has complete description
 - For example, standalone configuration includes:
 - **SPARK_MASTER_HOST / PORT / WEBUI_PORT** : Master connectivity
 - **SPARK_WORKER_CORES / INSTANCES** : Worker resources
 - And many, many more

Starting Spark and viewing UI XX-Image

```
$ ./sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to
/home/student/spark/logs/spark-student-org.apache.spark.deploy.master.Master-
1-localhost.localdomain.out
localhost: starting org.apache.spark.deploy.worker.Worker, logging to
/home/student/spark/logs/spark-student-org.apache.spark.deploy.worker.Worker-
1-localhost.localdomain.out
```

The screenshot shows the Apache Spark 2.1.0 master UI running at `localhost:8080`. The title bar reads "Spark Master at spark://mycomputer2574.home:7077". The UI displays the following information:

- URL:** `spark://mycomputer2574.home:7077`
- REST URL:** `spark://mycomputer2574.home:6066 (cluster mode)`
- Alive Workers:** 1
- Cores in use:** 8 Total, 0 Used
- Memory in use:** 15.0 GB Total, 0.0 B Used
- Applications:** 0 Running, 0 Completed
- Drivers:** 0 Running, 0 Completed
- Status:** ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20170306080531-192.168.1.2-60319	192.168.1.2:60319	ALIVE	8 (0 Used)	15.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

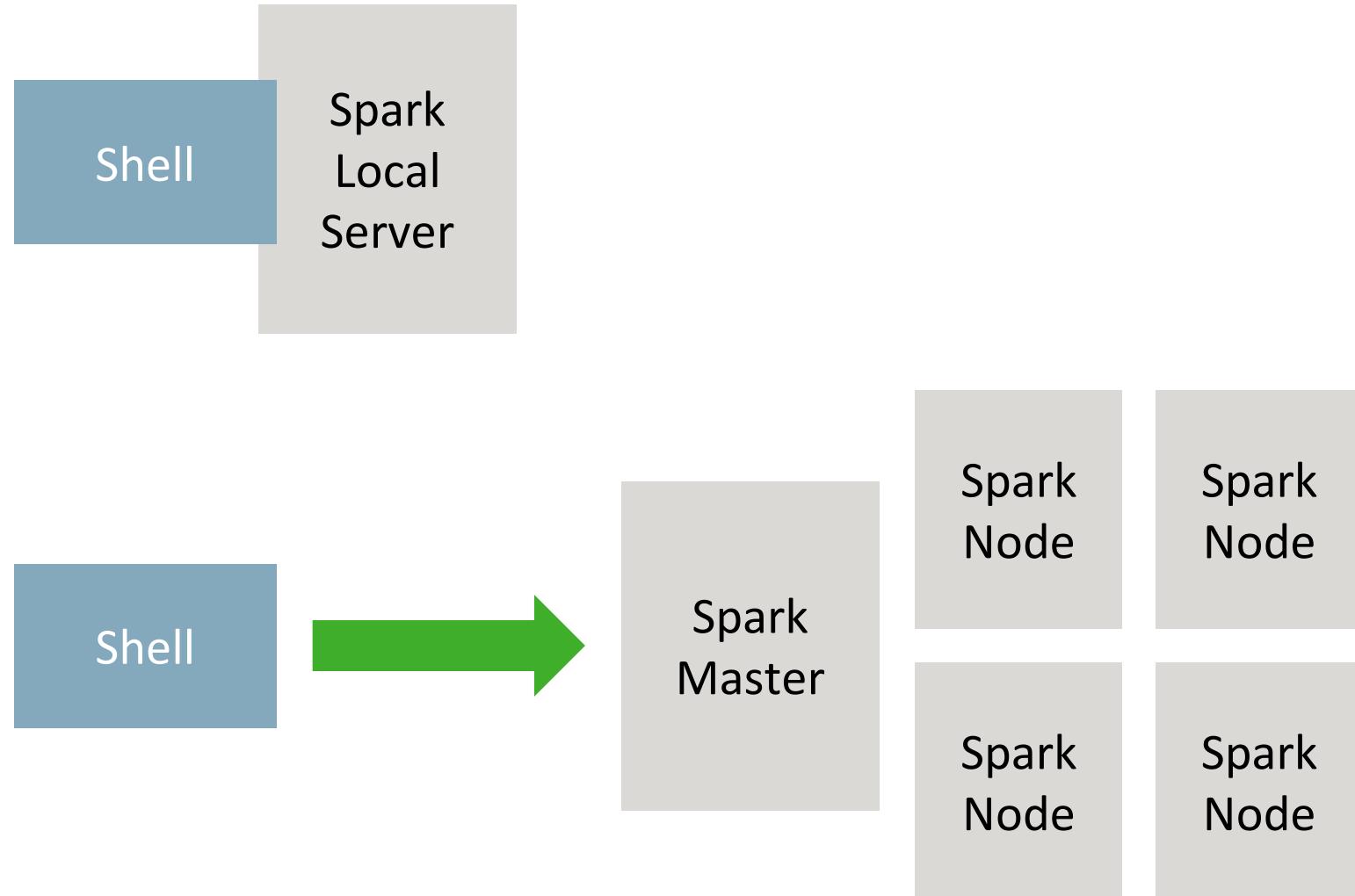
Lab 2.1: First Look at Spark

Set up and run a Spark cluster

Spark Shells

- **Interactive** shells support ad-hoc operations
 - Ad-hoc queries can access large clusters for fast response
 - They have full access to all capabilities
 - Used extensively in the course
 - Standalone apps covered later
- **Scala** shell: bin/**spark-shell** (We'll mostly use this.)
- Python shell: bin/**pyspark**
- Shells run in one of:
 - **Local** (pseudo) mode: Uses embedded, in-process spark server
 - NOT the same as using the standalone manger ⁽¹⁾
 - **Cluster** mode: Connect to cluster via URL of master

Shell Execution Modes



Spark Shell Startup Examples

- **spark-shell**: Startup using embedded server, and 1 thread
 - The default
- **spark-shell --master local[4]**: Startup using embedded server, and 4 worker threads
 - Generally limit to number of cores you have
- **spark-shell --master spark://my-computer.home:7077**
 - Connect to spark cluster with master at URL above
 - Assumes standalone cluster running on my-computer.home
- **spark-shell --help**: Show help

Other Spark Shell Options

- **Classpath** related options (all take comma separated lists)
 - **--jars**: List of local jars to include on classpath
 - **--packages**: List of maven coordinates of jars to include on classpath
 - **--exclude-packages**: List of groupId:artifactId to exclude while resolving --packages dependencies
- **Configuration** options
 - **--conf PROP=VALUE**: Sets a single config property
 - **--properties-file FILE**: Sets properties from FILE
- **JVM Options:**
 - **--driver-memory MEM**: Set memory for driver (default 1024M)
 - Several others
- See the help !

Master URL Options

Master URL	Details
local	Run Spark locally on one thread. No parallelism.
local[k]	Run Spark locally with K worker threads – which should be less or equal to # of cores on your machine.
local[*]	When you don't know how many cores are on your machine, you can use a wild card.
spark://HOST:PORT	Connect to Spark standalone cluster master. 7077 is default
mesos://HOST:PORT	Connect to Mesos cluster. 5050 is default
yarn-client	Connect to a YARN cluster in client mode.
yarn-cluster	Connect to YARN cluster in cluster mode.

Starting Spark Shell (Local Mode)

```
$ bin/spark-shell
```

```
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
```

```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, ...
```

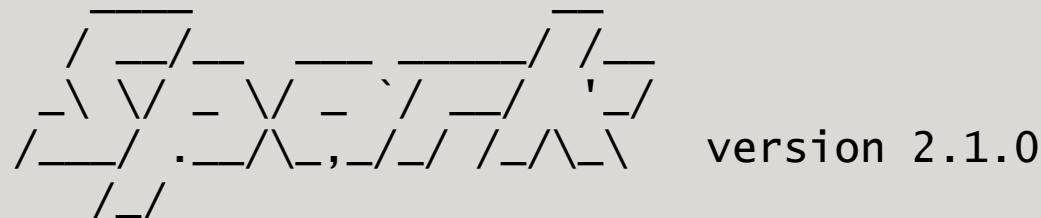
```
// WARN logging messages omitted ...
```

```
Spark context Web UI available at http://192.168.1.2:4040
```

```
Spark context available as 'sc' (master = local[*], app id = local-1488813130684).
```

```
Spark session available as 'spark'.
```

```
Welcome to
```



```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_45)
```

```
Type in expressions to have them evaluated.
```

```
Type :help for more information.
```

```
scala> // We will use > instead of scala> to denote the prompt in examples
```

Web UI for Spark Shell

- Web-based access at **<shell-host>:4040**
 - Can set port ⁽¹⁾
 - UI is built into the `SparkContext` ⁽²⁾
 - Every `SparkContext` instance launches a Web UI instance
 - Provides information about driver program and spark jobs
 - We illustrate the event timeline at right

The screenshot shows the Apache Spark 2.1.0 Web UI interface. At the top, the URL is `localhost:4040/jobs/`. Below the header, the Apache Spark logo is displayed. The navigation bar includes tabs for Jobs, Stages, Storage, Environment, Executors, and SQL. The main content area is titled "Spark Jobs" with a help icon. It displays user information: User: yaakov, Total Uptime: 5.1 min, and Scheduling Mode: FIFO. A section titled "Event Timeline" is expanded, showing a checkbox for "Enable zooming". Below this, there are two tables: one for "Executors" and one for "Jobs". The "Executors" table has columns for status (Added or Removed) and time. A tooltip "Executor driver added" is shown over a blue cell in the first column. The "Jobs" table has a single column for status (Succeeded). The bottom right corner of the slide features the Hortonworks logo, which consists of three green elephants and the word "HORTONWORKS".

Web UI Tabs

- **Jobs**: Status of all jobs in a Spark application (SparkContext)
 - Either scheduled or already run
 - Can click through to a job detail
- **Stages**: Current state of all stages of all jobs
 - Plus tasks and statistics for a stage
- **Storage**: Including RDD size and memory use
- **Environment**: Spark and system properties, classpath, runtime info
- **Executor**: Processing/storage for each executor
 - Can click through to thread dump for an executor
- **SQL**: Displays SQL query executions
 - Including details of completed queries

SparkContext — First Look

- Access to Spark is via a SparkContext instance
 - Represents connection to Spark cluster
 - Pre-created in shell as variable `sc`
 - Below, we illustrate some simple uses of the context
 - Note the tab completion in the second example ⁽¹⁾

```
> sc
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@2c01a0dd

> sc.[tab]
accumable      accumableCollection  accumulator      addFile          addJar
addSparkListener appName           applicationId
binaryFiles
binaryRecords    broadcast
clearCallSite
clearFiles       clearJars
defaultMinSplits
... Remaining detail omitted

> sc.isLocal
res1: Boolean = true

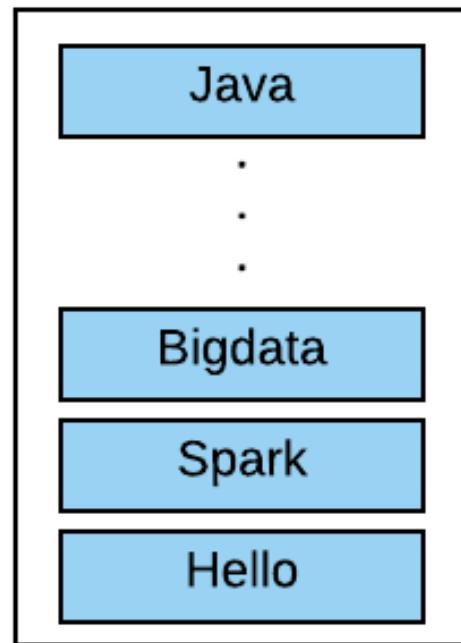
> sc.master
res2: String = local[*]
```

RDD—First Look

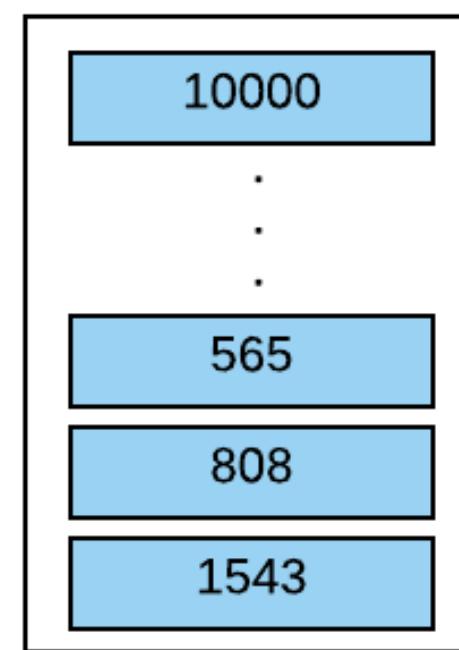
- **Resilient Distributed Dataset:** Distributed Spark collection
 - **Core data abstraction** of the computation engine
 - Ops on RDDs are automatically parallelized across cluster
 - Higher level APIs (e.g. Dataset) don't require using RDDs
 - But they're important for understanding/using Spark
 - And there is existing code using them
- We'll illustrate simple examples now
 - Creating an RDD via the SparkContext
 - Basic work with the API
 - API based on Scala collections ⁽¹⁾
- More detail later
 - We'll also look at higher level APIs

Resilient Distributed Datasets (RDDs)

-

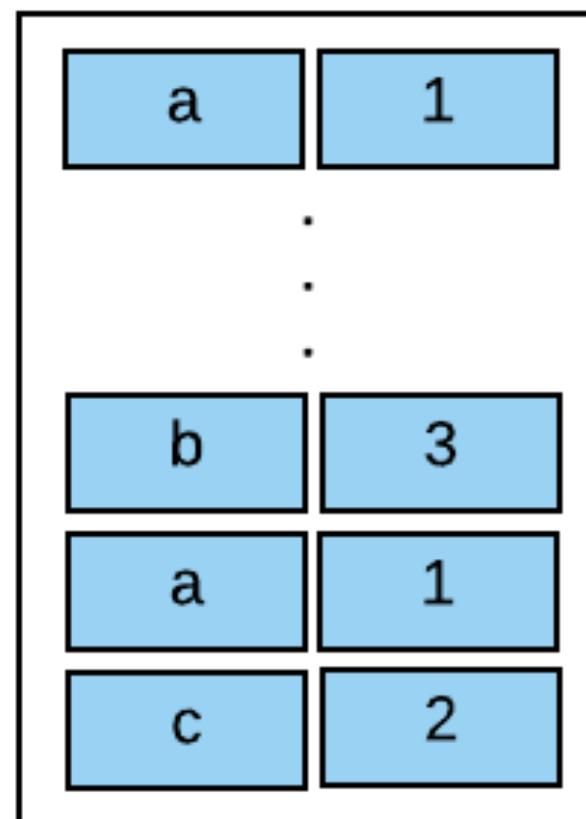


RDD of Strings



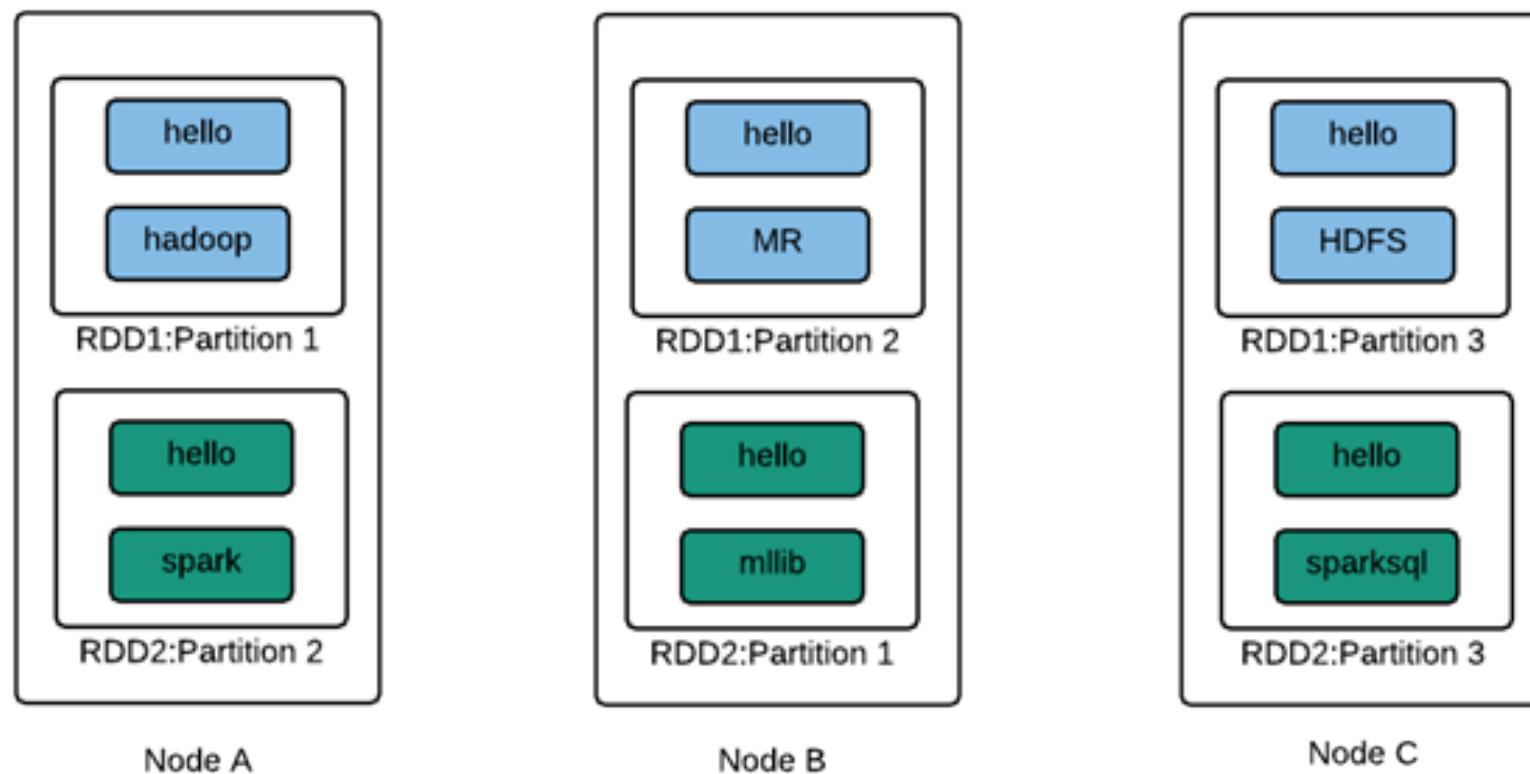
RDD of Integers

Pair RDD



Pair RDD

RDD



RDD

- RDDs can contain any types of objects, including user-defined classes.
- An RDD is simply a encapsulation around a very large dataset. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result.
- Under the hood, Spark will automatically distribute the data contained in RDDs across your cluster and parallelize the operations you perform on them.

How to create a RDD

1. Load RDDs from external storage.

```
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
```

How to create a RDD

2. Take an existing collection in your program and pass it to SparkContext's parallelize method.

```
List<Integer> inputIntegers = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> integerRdd = sc.parallelize(inputIntegers);
```

What can we do with RDDs?

Transformations

Actions

Transformations

- Apply some functions to the data in RDD to create a new RDD.
- One of the most common transformations is **filter** which will return a new RDD with a subset of the data in the original RDD.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
JavaRDD<String> linesWithFriday = lines.filter(line -> line.contains("Friday"));
```

Actions

- Compute a result based on an RDD.
- One of the most popular Actions is **first**, which returns the first element in an RDD.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
String firstLine = lines.first();
```

Lazy evaluation

```
//No thing happens when Spark sees textFile() statement.  
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");  
  
//No thing happens when Spark sees filter() transformation  
JavaRDD<String> linesWithFriday = lines.filter(line -> line.startsWith("Friday"));  
  
//Spark only starts loading the "in/uppercase.text" file when first() action is called on linesWithFriday RDD.  
String firstLineWithFriday = linesWithFriday.first();  
  
// Spark scans the file only until the first line starting with Friday is detected;  
// it doesn't even need to go through the entire file.
```

How to create a RDD

- Take an existing collection in your program and pass it to SparkContext's **parallelize** method.

```
List<Integer> inputIntegers = Arrays.asList(1, 2, 3, 4, 5);  
JavaRDD<Integer> integerRdd = sc.parallelize(inputIntegers);
```

- All the elements in the collection will then be copied to form a distributed dataset that can be operated on in parallel.
- Very Handy to create an RDD with little effort.
- NOT practical working with large datasets.

How to create a RDD

- Load RDDs from external storage by calling **textFile** method on Sparkcontext.

```
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
```

- The external storage is usually a distributed file system such as **Amazon S3** or **HDFS**.
- There are other data sources which can be integrated with Spark and used to create RDDs including JDBC, Cassandra, and Elasticsearch, etc.

Transformation

- Transformations are operations on RDDs which will return a new RDD.
- The two most common transformations are **filter** and **map**.

filter() transformation

- Takes in a function and returns an RDD formed by selecting those elements which pass the filter function.
- Can be used to remove some invalid rows to clean up the input RDD or just get a subset of the input RDD based on the filter function.

```
JavaRDD<String> cleanedLines = lines.filter(line -> !line.isEmpty());
```

map() transformation

- Takes in a function and passes each element in the input RDD through the function, with the result of the function being the new value of each element in the resulting RDD.
- It can be used to make HTTP requests to each URL in our input RDD, or it can be used to calculate the square root of each number.

```
JavaRDD<String> URLs = sc.textFile( path: "in/urls.text");
URLs.map(url -> makeHttpRequest(url));
```

- The return type of the map function is not necessary the same as its input type.

```
JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");
JavaRDD<Integer> lengths = lines.map(line -> line.length());
```

flatMap VS map

```
/**  
 * Base interface for a map function used in Dataset's map function.  
 */  
public interface MapFunction<T, U> extends Serializable {  
    U call(T value) throws Exception;  
}  
  
/**  
 * A function that returns zero or more output records from each input record.  
 */  
public interface FlatMapFunction<T, R> extends Serializable {  
    Iterator<R> call(T t) throws Exception;  
}
```

flatMap VS map

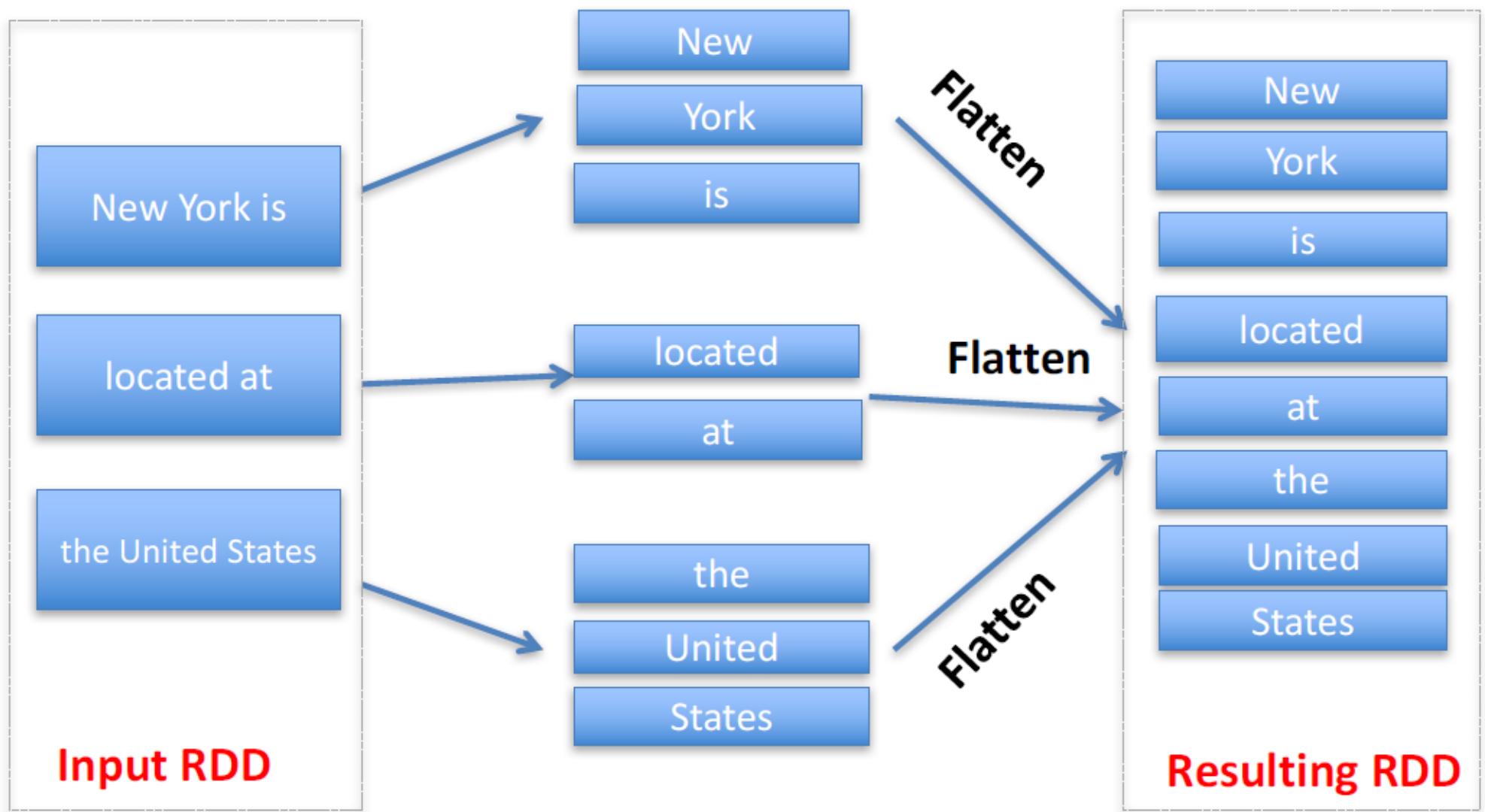
map:

1 to 1 relationship

flatMap:

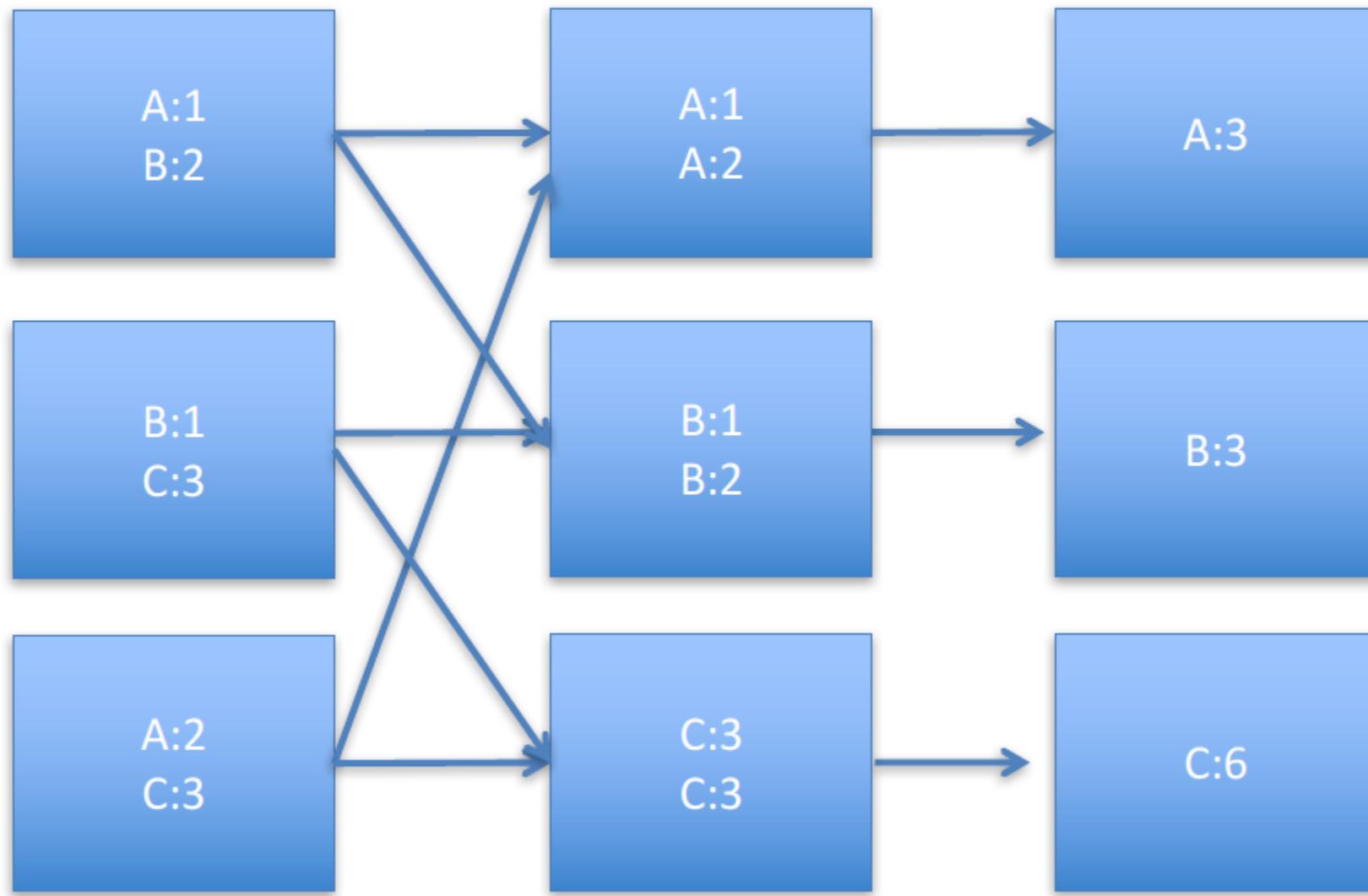
1 to many relationship

flatMap example: split lines by space

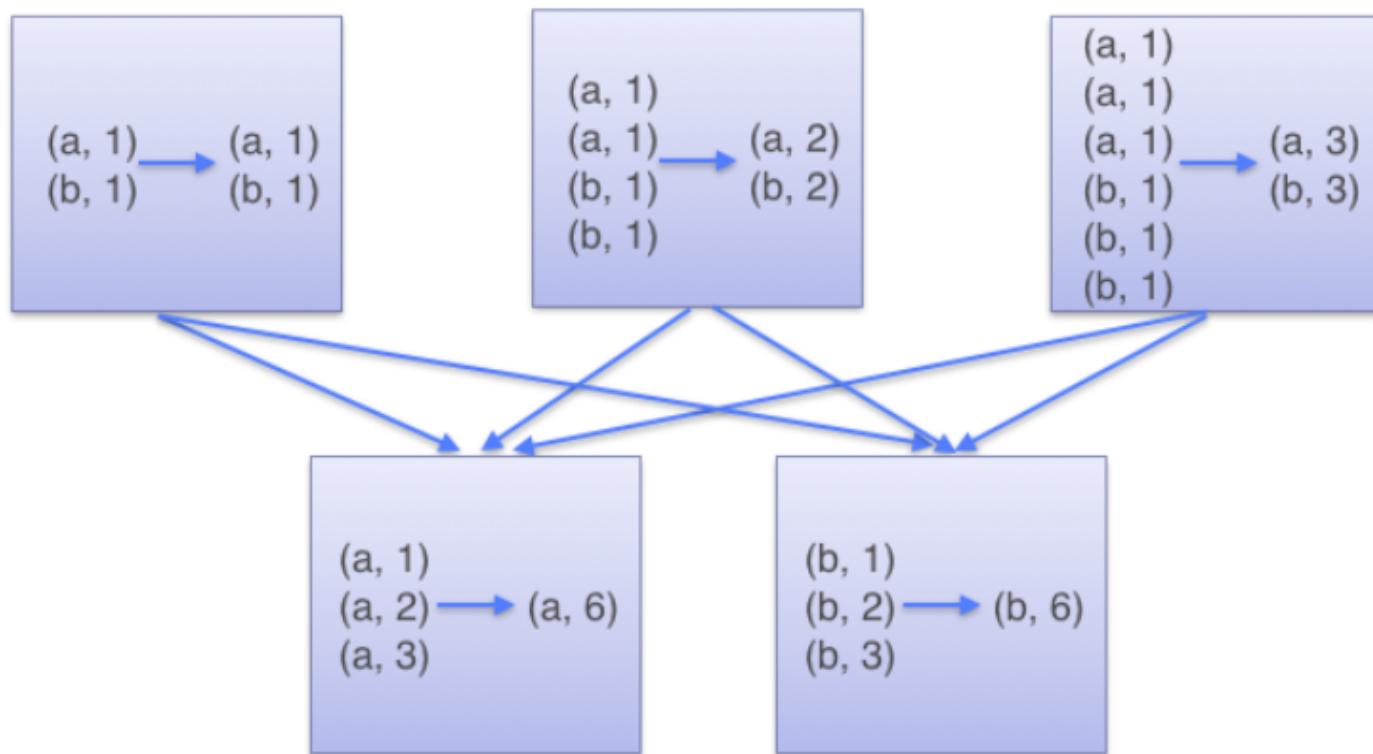


partitionBy

groupByKey



ReduceByKey



```
public class Uppercase {

    public static void main(String[] args) throws Exception {

        SparkConf conf = new SparkConf().setAppName("uppercase").setMaster("local[*]");
        JavaSparkContext sc = new JavaSparkContext(conf);
        JavaRDD<String> lines = sc.textFile( path: "in/uppercase.text");

        // anonymous inner class
        lines.filter(new Function<String, Boolean>() {
            @Override
            public Boolean call(String line) throws Exception {
                return line.startsWith("Friday");
            }
        });

        // named class
        lines.filter(new StartsWithFriday());

        // lambda expression
        lines.filter(line -> line.startsWith("Friday"));
    }

    static class StartsWithFriday implements Function<String, Boolean> {
        public Boolean call( String line){
            return line.startsWith("Friday");
        }
    }
}
```

Set operations which are performed on **one** RDD:

- **sample**
- **distinct**

sample

```
public org.apache.spark.api.java.JavaRDD<T> sample(boolean withReplacement, double fraction)
```

- The sample operation will create a random sample from an RDD.
- Useful for testing purpose.

distinct

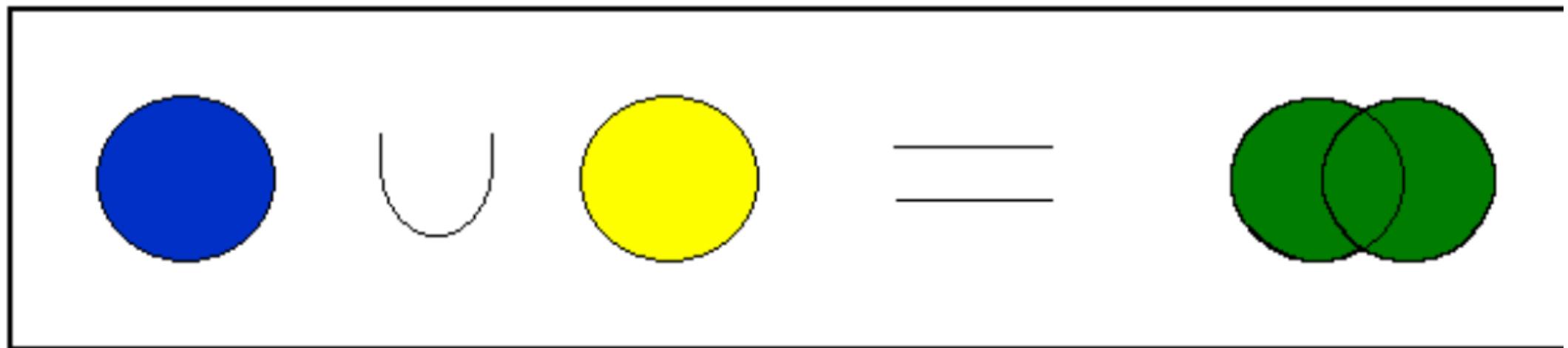
public org.apache.spark.api.java.JavaRDD<T> distinct()

- The distinct transformation returns the distinct rows from the input RDD.
- The distinct transformation is expensive because it requires shuffling all the data across partitions to ensure that we receive only one copy of each element.

Set operations which are performed on **two** RDDs and produce **one** resulting RDD:

- **union**
- **intersection**
- **subtract**
- **cartesian product**

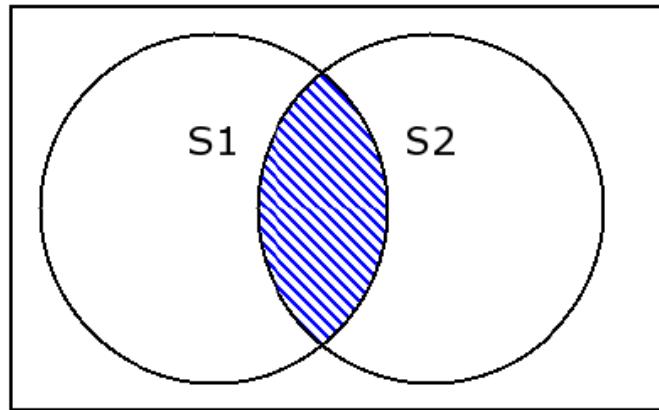
union operation



```
public org.apache.spark.api.java.JavaRDD<T> union(org.apache.spark.api.java.JavaRDD<T> othe
```

- Union operation gives us back an RDD consisting of the data from both input RDDs
- If there are any duplicates in the input RDDs, the resulting RDD of Spark's union operation will contain duplicates as well.

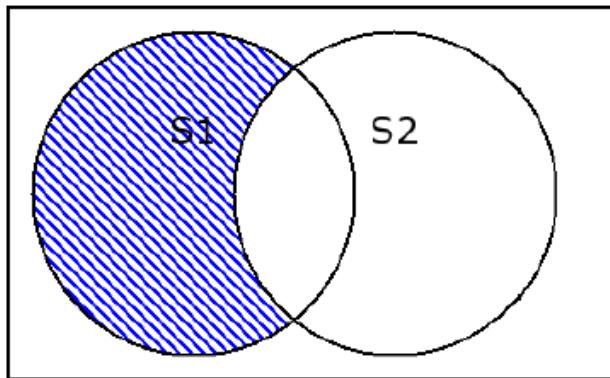
intersection operation



```
public org.apache.spark.api.java.JavaRDD<T> intersection(org.apache.spark.api.java.JavaRDD<T> other)
```

- Intersection operation returns the common elements which appear in both input RDDs.
- Intersection operation removes all duplicates including the duplicates from single RDD before returning the results.
- Intersection operation is quite expensive since it requires shuffling all the data across partitions to identify common elements.

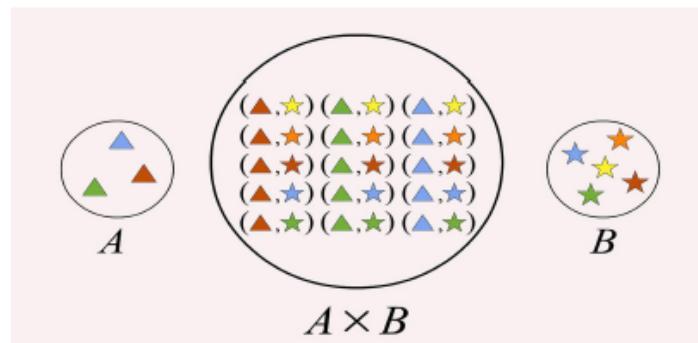
subtract operation



```
public org.apache.spark.api.java.JavaRDD<T> subtract(org.apache.spark.api.java.JavaRDD<T> other)
```

- Subtract operation takes in another RDD as an argument and returns us an RDD that only contains element present in the first RDD and not the second RDD.
- Subtract operation requires a shuffling of all the data which could be quite expensive for large datasets.

cartesian operation



```
public <U> JavaPairRDD<T, U> cartesian(JavaRDDLike<U, ?> other)
```

- Cartesian transformation returns all possible pairs of a and b where a is in the source RDD and b is in the other RDD.
- Cartesian transformation can be very handy if we want to compare the similarity between all possible pairs.

Common Actions in Spark

- **collect**
- count
- countByValue
- take
- saveAsTextFile
- reduce

count and countbyValue

- If you just want to count how many rows in an RDD, **count** operation is a quick way to do that. It would return the count of the elements.
- **countbyValue** will look at unique values in the each row of your RDD and return a map of each unique value to its count.
- **countbyValue** is useful when your RDD contains duplicate rows, and you want to count how many of each unique row value you have.

take

```
List<String> words = wordRdd.take(n);
```

- **take** action takes **n** elements from an RDD.
- **take** operation can be very useful if you would like to take a peek at the RDD for unit tests and quick debugging.
- **take** will return n elements from the RDD and it will try to reduce the number of partitions it accesses, so it is possible that the take operation could end up giving us back a **biased** collection, and it doesn't necessarily return the elements in the order we might expect.

saveAsTextFile

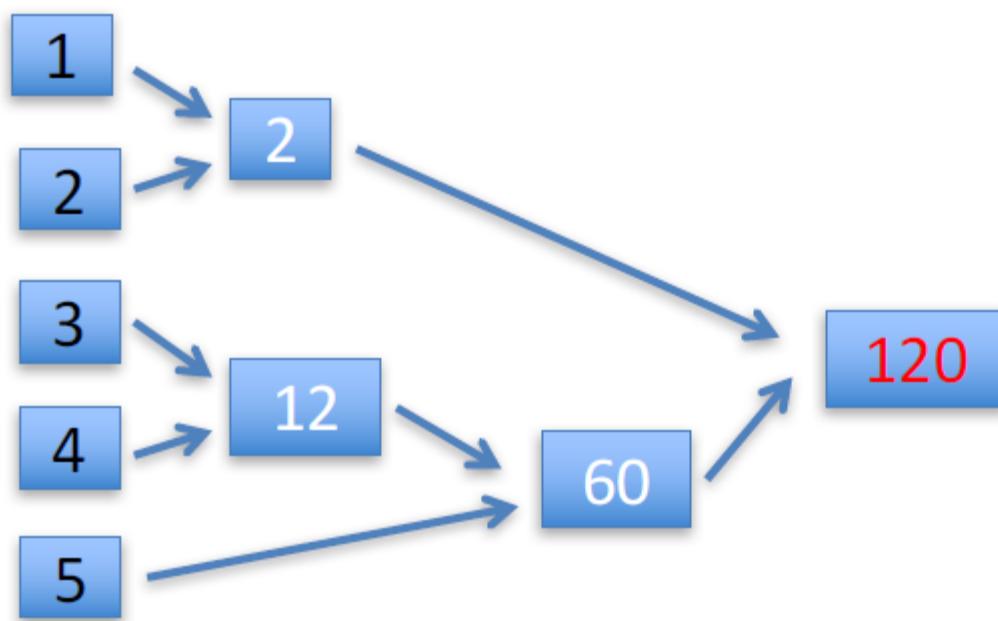
```
airportsNameAndCityRdd.saveAsTextFile( path: "out/airports.text");
```

saveAsTextFile can be used to write data out to a distributed storage system such as HDFS or Amazon S3, or even local file system.

reduce

```
/**  
 * A two-argument function that takes arguments of type T1 and T2 and returns an R.  
 */  
public interface Function2<T1, T2, R> extends Serializable {  
    R call(T1 v1, T2 v2) throws Exception;  
}  
  
Integer product = integerRdd.reduce((x, y) -> x * y);
```

- **reduce** takes a function that operates on **two** elements of the type in the input RDD and returns **a** new element of the same type. It reduces the elements of this RDD using the specified binary function.
- This function produces the same result when **repetitively** applied on the same set of RDD data, and reduces to a single value.
- With **reduce** operation, we can perform different types of aggregations.



Transformations return RDDs, whereas **actions** return some other data type.

Transformations:

```
JavaRDD<String> linesWithFriday = lines.filter(line -> line.contains("Friday"));  
JavaRDD<Integer> lengths = lines.map(line -> line.length());
```

Actions:

```
List<String> words = wordRdd.collect();  
String firstLine = lines.first();
```

How to create **Pair RDDs**

1. Return Pair RDDs from a list of key value data structure called tuple.
2. Turn a regular RDD into a Pair RDD.

Get Pair RDDs from **tuples**

Java doesn't have a built-in tuple type, so Spark's Java API allow users to create tuples using the *scala.Tuple2* class.

```
Tuple2<Integer, String> tuple = new Tuple2<>(_1: 12, _2: "value");
Integer key = tuple._1();
String value = tuple._2();

public <K, V> JavaPairRDD<K, V> parallelizePairs(List<Tuple2<K, V>> list)
```

Transformations on Pair RDD

- Pair RDDs are allowed to use all the transformations available to regular RDDs, and thus support the same functions as regular RDDs.
- Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements.

filter transformation

- This **filter** transformation that can be applied to a regular RDD can also be applied to a Pair RDD.
- The **filter** transformation takes in a function and returns a Pair RDD formed by selecting those elements which pass the **filter** function.

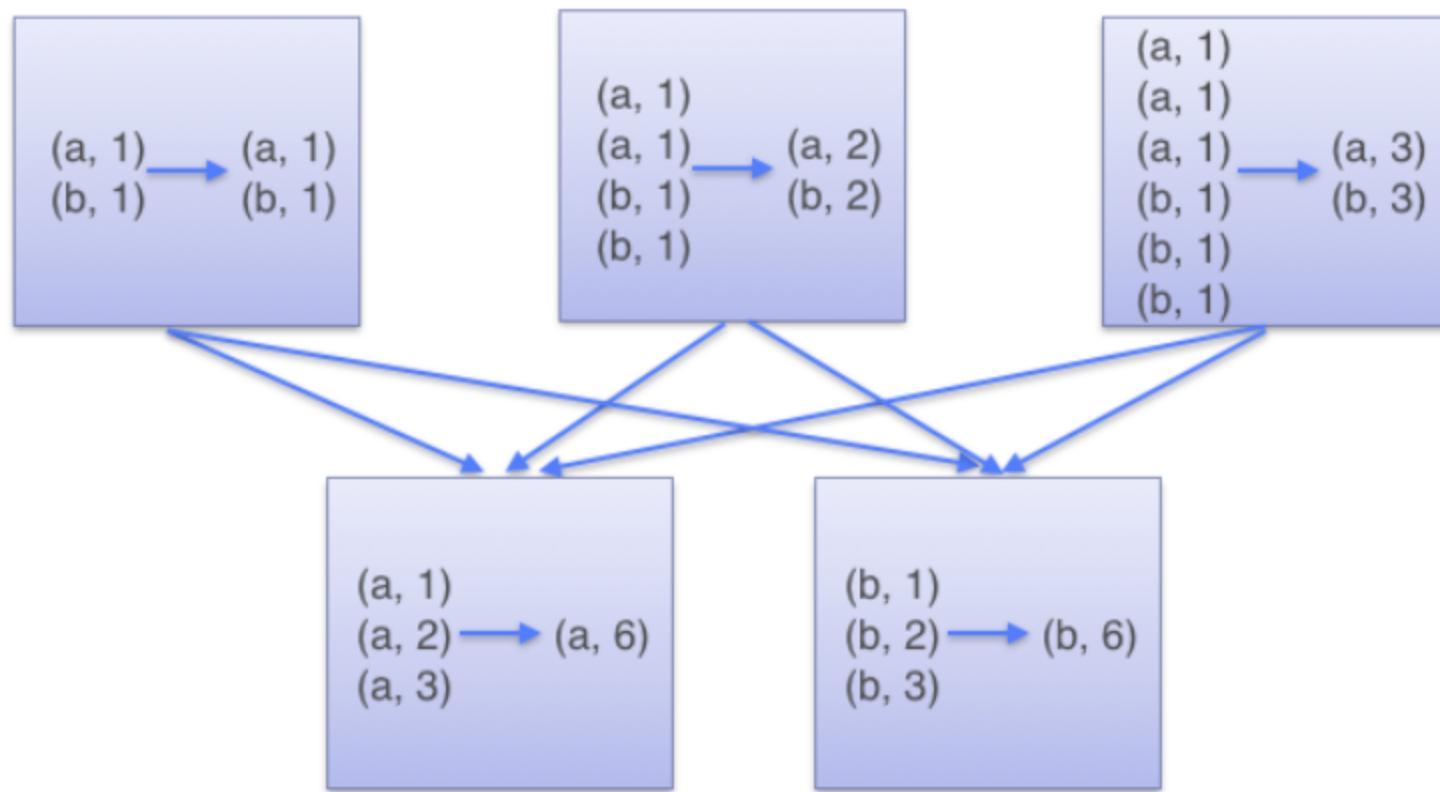
map and mapValues transformations

- The `map` transformation also works for Pair RDDs. It can be used to convert an RDD to another one.
- But most of the time, when working with pair RDDs, we don't want to modify the keys, we just want to access the value part of our Pair RDD.
- Since this is a typical pattern, Spark provides the `mapValues` function. The `mapValues` function will be applied to each key value pair and will convert the values based on `mapValues` function, but it will not change the keys.

Aggregation

- When our dataset is described in the format of key-value pairs, it is quite common that we would like to aggregate statistics across all elements with the same key.
- We have looked at the `reduce` actions on regular RDDs, and there is a similar operation for pair RDD, it is called `reduceByKey`.
- `reduceByKey` runs several parallels `reduce` operations, one for each key in the dataset, where each operation combines values that have the same key.
- Considering input datasets could have a huge number of keys, `reduceByKey` operation is not implemented as an action that returns a value to the driver program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

ReduceByKey



Recent property listings in San Luis Obispo, California

1st column: unique ID for the house

2nd column: location of the property

3rd column: the price of the property in US dollars

4th column: the number of bedrooms of the property

5th column: the number of bathrooms of the property

6th column: the size of the house in square feet

7th column: the price of the house per square foot

8th column: the state of sale

Sample Solution for the **Average House** problem

Task: compute the average price for houses with different number of bedrooms

This could be converted to an aggregation type problem of Pair RDD.

Average price for different type of houses problem:

- key: the number of bedrooms
- value: the average price of the property

Word count problem:

- key: word
- value: the count of each word

groupByKey

- A common use case for Pair RDD is grouping our data by key. For example, viewing all of an account's transactions together.
- If our data is already keyed in the way we want, **groupByKey** will group our data using the key in our Pair RDD.
- Let's say, the input pair RDD has **keys** of type K and **values** of type V, if we call group by key on the RDD, we get back a Pair RDD of **type K**, and **Iterable V**.

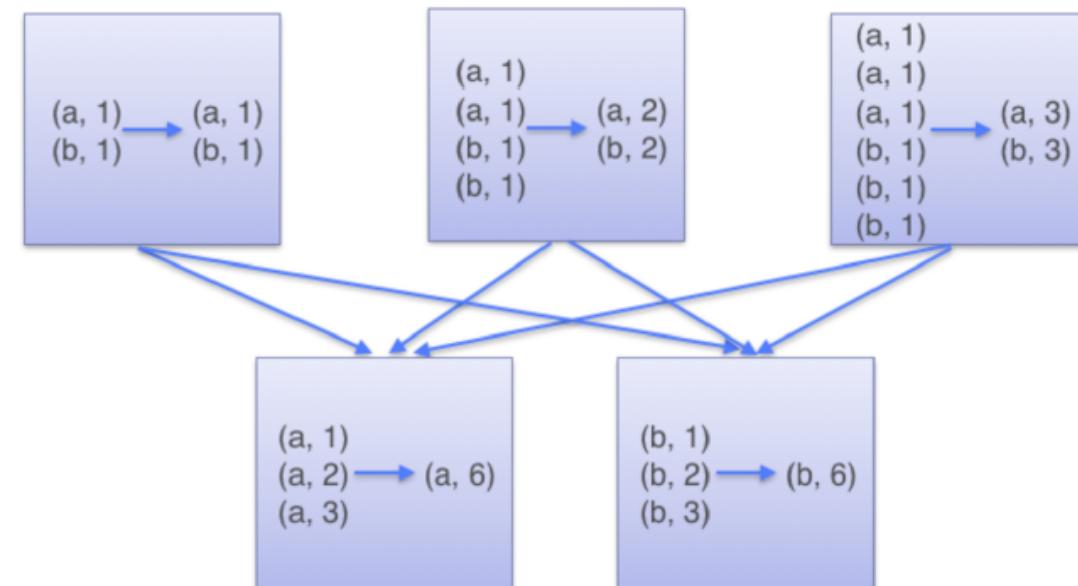
```
public JavaPairRDD<K, Iterable<V>> groupByKey()
```

`groupByKey + [reduce, map, mapValues]`

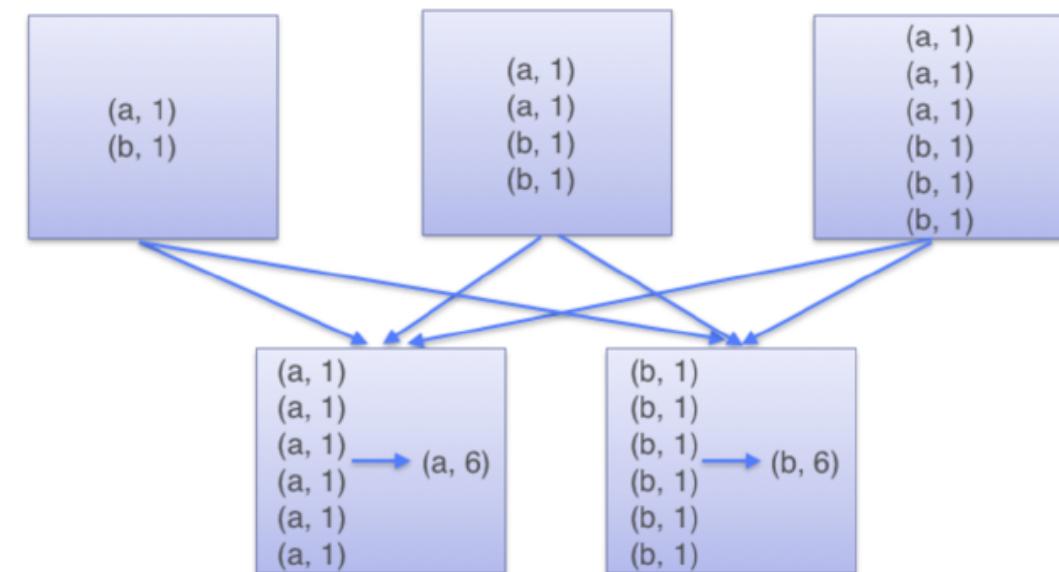
can be replaced by one of the per-key aggregation functions such as `reduceByKey`

ReduceByKey(preferred) VS GroupByKey

ReduceByKey



GroupByKey



Sort Pair RDD

- We can sort a Pair RDD as long as there is an ordering defined on the key.
- Once we have sorted our Pair RDD, any subsequent call on the sorted Pair RDD to collect or save will return us ordered data.

sortByKey in reverse order

```
public org.apache.spark.api.java.JavaPairRDD<K,V> sortByKey(boolean ascending)  
wordsPairRdd.sortByKey(true);
```

custom comparator

```
public org.apache.spark.api.java.JavaPairRDD<K,V> sortByKey(java.util.Comparator<K> comp)  
countPairRdd.sortByKey((a, b) -> Math.abs(a) - Math.abs(b));
```

- What if we want to **sort** the resulting RDD by the **number of bedrooms** so that we can see exactly how the price changes while the **number of bedrooms** increases?

- What if we want to **sort** the resulting RDD by the **number of bedrooms** so that we can see exactly how the price changes while the **number of bedrooms** increases?

- Sorted Number of Bedrooms problem:
 - The **number of bedrooms** is the **key**
 - Can call **sortByKey** on the Pair RDD
- Sorted Word Count problem:
 - The **count** for each word is the **value** not the **key** of the Pair RDD
 - We can't use **sortByKey** directly on the count

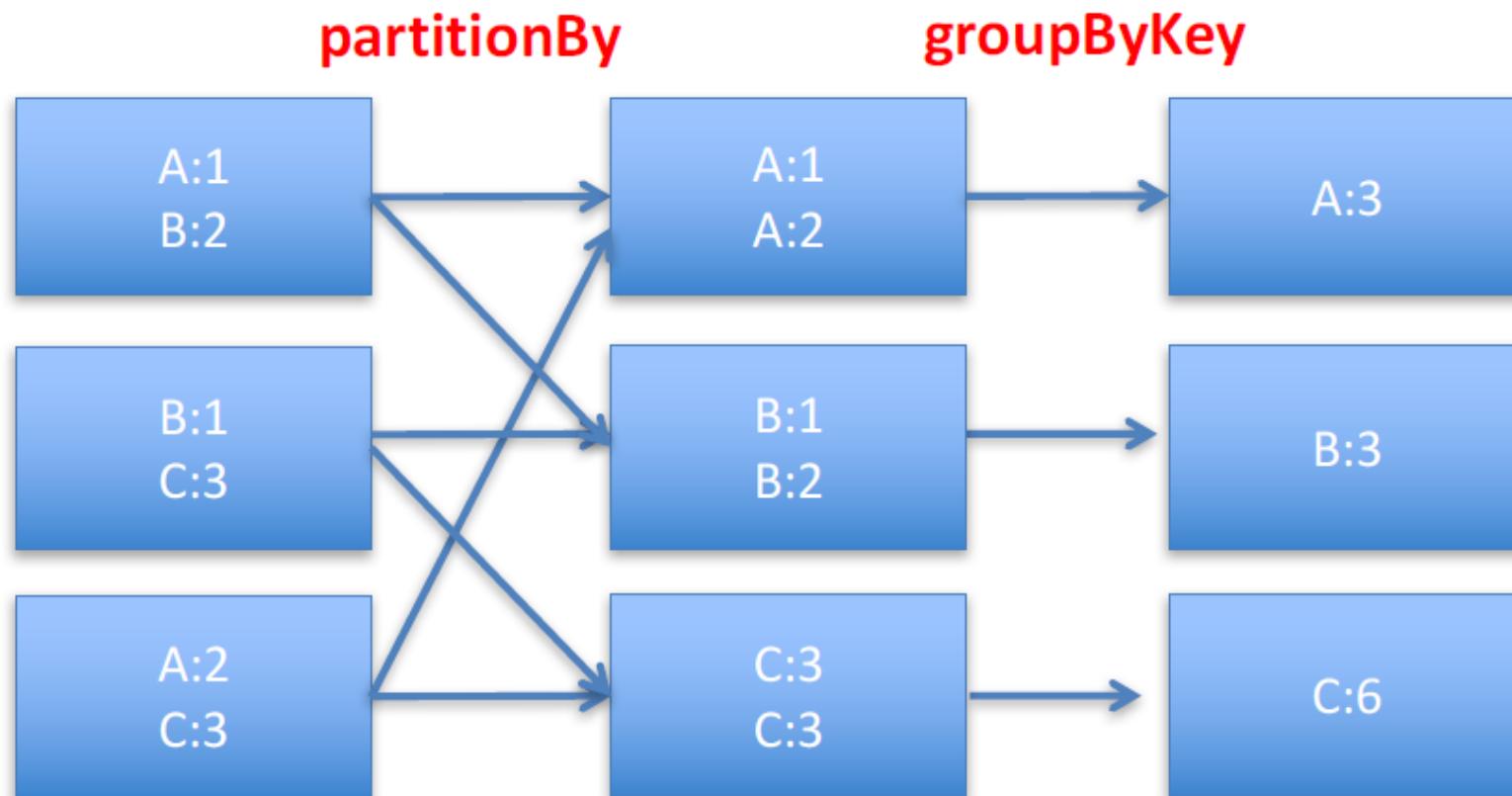
Solution:

1. Flip the key value of the word count RDD to create a new Pair RDD with the **key** being the **count** and the **value** being the **word**.
2. Do **sortByKey** on the intermediate RDD to sort the count.
3. Flip back the Pair RDD again, with the **key** back to the **word** and the **value** back to the **count**.

- In general, we should avoid using `groupByKey` as much as possible.

reduce the amount of shuffle for `groupByKey`

```
val partitionedWordPairRdd = wordPairRdd.partitionBy(new HashPartitioner(4))
partitionedWordPairRdd.persist(StorageLevel.DISK_ONLY)
partitionedWordPairRdd.groupByKey().collect()
```



RDD Usage Examples

- Our simple examples below:
 - **Read a file** (guess how it's represented in Spark)
 - Get some of the file's data
 - Filter the data (what do you get back?)
 - We'll explain the filtering syntax later ⁽¹⁾

```
> val myfile = sc.textFile("README.md") # Create an RDD from contents of file
`myfile: org.apache.spark.rdd.RDD[String] = README.md MapPartitionsRDD[1] at textFile at
<console>:27

> val first = myfile.first()
first: String = # Apache Spark

> val all = myfile.collect()
all: Array[String] = Array(# Apache Spark, "", Spark is a fast and general cluster comp
# ... Remaining detail omitted

> val scalaLines = myfile.filter(line => line.contains("Scala"))
`scalaLines: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at filter at
<console>:29

> scalaLines.collect()
res1: Array[String] = Array(high-level APIs in Scala, Java, and Python, and an optimized
engine that, ## Interactive Scala Shell, The easiest way to start using Spark is through
the Scala shell:)
```

A photograph of a man and a woman sitting at a desk, looking at a laptop screen together. They are both smiling. The man has a beard and is wearing a light-colored shirt. The woman has her hair pulled back and is wearing a white top. A green diagonal stripe runs from the bottom right corner across the slide.

Lab 2.2: First Look at the Spark Shell

In this lab, we will work with the Spark Shell

Session Review

- What is Spark, and what are its capabilities?
- How do you work with Spark?
- How does Spark compare to Hadoop / MapReduce?

Session Summary

- Spark is an open source cluster computing engine
 - **Core**: Distributed processing of large data sets on a cluster
 - Additional Packages:
 - **Spark SQL**: Structured data
 - **Spark Streaming**: Live data streams
 - **ML Lib**: Machine learning
 - **GraphX**: Graph Processing
 - Fast, flexible, and relatively easy to code
 - Including Scala, Java, Python, R, and other APIs
 - Top level Apache project, supported by Databricks

Session Summary

- Clusters run standalone, or integrated with YARN or Mesos
 - Simple install by extracting tarball
 - Included in some Hadoop distributions (Cloudera/Hortonworks)
- Spark **worker processes** run on distributed nodes
 - Running tasks as assigned by the driver
- The **Spark Shell** supports ad-hoc, interactive operations
 - Using the Scala or Python API
 - A **SparkContext** provides access to Spark
- **Resilient Distributed Datasets** (RDDs) hold cluster data
 - A distributed collection of items that cluster works on in parallel

Session Summary

- Spark overcomes many MapReduce (Hadoop) limitations
 - Addresses current Big Data needs
 - Faster and easier to program
 - Unified stack for wide applicability
- Usage growing rapidly
 - Increasingly wide adoption
 - Extremely active community evolving the code base
 - Ecosystem is still not as mature as Hadoop
 - e.g. management and monitoring solutions are not as robust

A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt and blue jeans. He is standing in what appears to be a workshop or industrial setting, with pipes and equipment visible in the background. He is gesturing with his hands while speaking, holding a small dark object in his left hand. In the foreground, the back of another person's head and shoulders are visible, suggesting an audience. A large green diagonal stripe runs from the top right corner across the slide.

Session 3: RDDs and Spark Architecture

- RDD Concepts
- Working with RDDs

Session Objectives

- Understand what a Resilient Distributed Dataset (RDD) is
- Be familiar with Spark's architecture and how it works
- Use Spark's API to work with RDDs
- NOTE: DataFrames/Datasets have replaced RDD as the preferred API for Spark programming
 - However, they are still a core part of the Spark engine, and useful to know in understanding Spark
 - DataFrame/Dataset code is translated to RDD code
 - There is also still plenty of RDD code in use
 - You may run into it



RDD Concepts

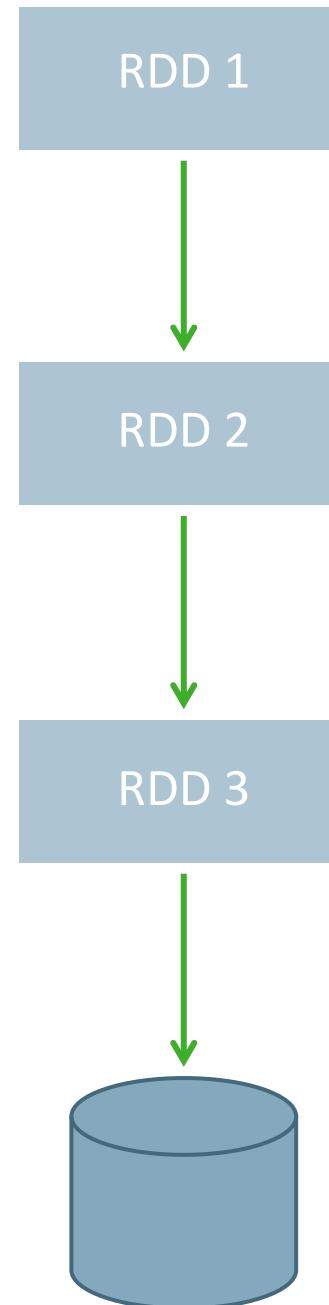
Working with RDDs

What is an RDD?

- **Resilient Distributed Dataset: Core Spark data abstraction**
 - **Distributed collection** of elements
 - **Partitioned** — usually **across different nodes**
 - **Read-only** (immutable)
 - Operations execute **in parallel** on the partitions
 - **Fault tolerant**: Can recover from loss of a partition
 - The "Resiliency"
 - **Efficient** — Re-computed, not stored ⁽¹⁾ (more on this soon)
- Two operation types
 - **Transformation**: Lazy operation creating new RDD
 - e.g. `map()`, `filter()`
 - **Action**: Return a result or save it
 - e.g. `take()`, `save()`

RDD Lifecycle

- RDD is **created** by either:
 - Loading an external dataset
 - Distributing a local collection
- RDD is **transformed**:
 - e.g. filter out elements
 - Result: **A new RDD** ⁽¹⁾
 - Often have a sequence of transformations
- Data is eventually **extracted**
 - By an **action** on an RDD
 - e.g. save the data
- At right, we read/transform a log file, then save the result



Create: Read a log file
(e.g. from HDFS)
sc.textFile("server.log")

Filter: Keep lines
starting with "ERROR"

Filter: Keep lines
containing "mysql"

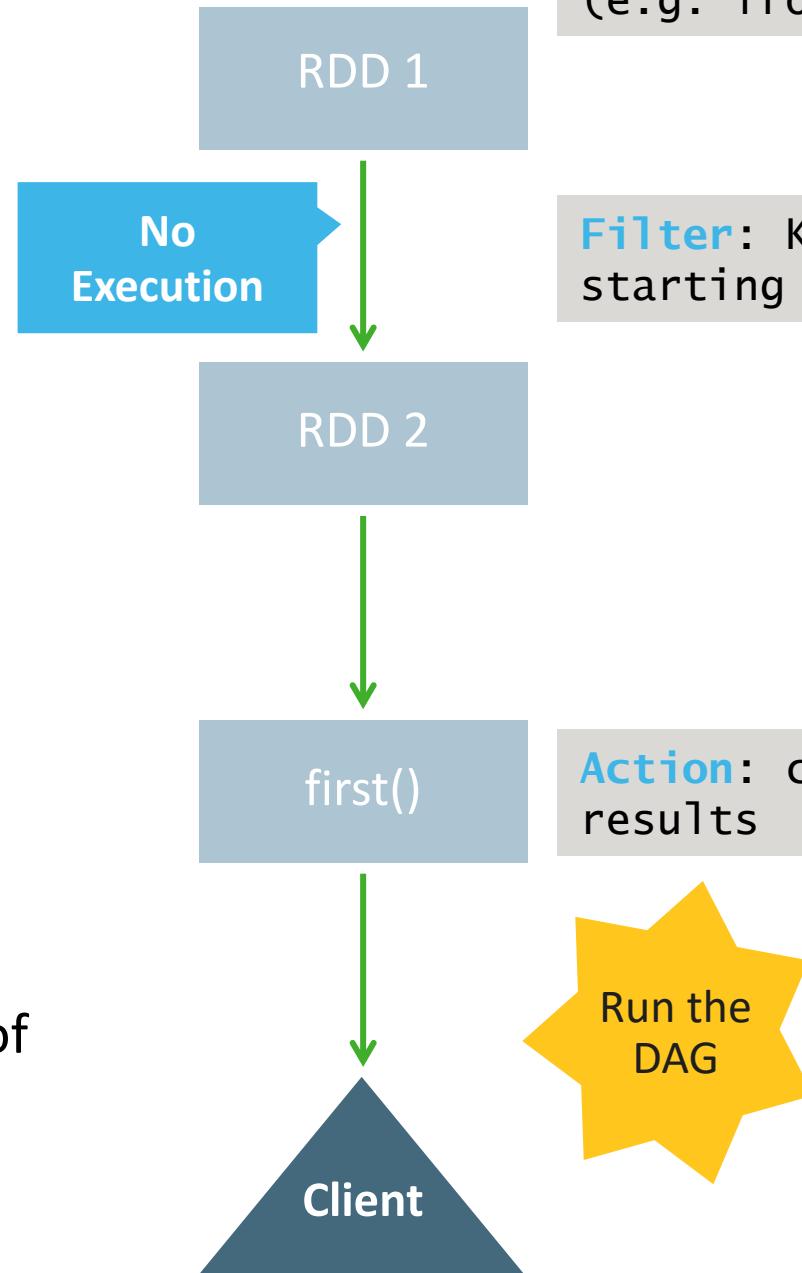
Action: Write result to
storage

All Transformations are Lazy

- Spark **doesn't immediately** compute results
 - Transformations stored as a graph (DAG) from a base RDD
 - Consider an RDD to be a set of operations
 - The ops required to produce data from a base
 - It's not really a container for specific data
- The DAG is executed when an action occurs
 - When it needs to provide data
- Allows Spark to:
 - **Optimize** required calculations (we'll view this soon)
 - **Efficiently** recover RDDs on node failure (more on this later)

Lazy Evaluation

- This example reads a log file
 - It filters out all but error lines
- At this point, **no work done**
- Client requests the first line
 - Triggers evaluation of the DAG
 - **Here**, the work is done
 - Result is sent to client
- Many possible optimizations
 - Stop filtering after the first ERROR line encountered ⁽¹⁾
 - Doesn't even need to read all of the log file ⁽²⁾



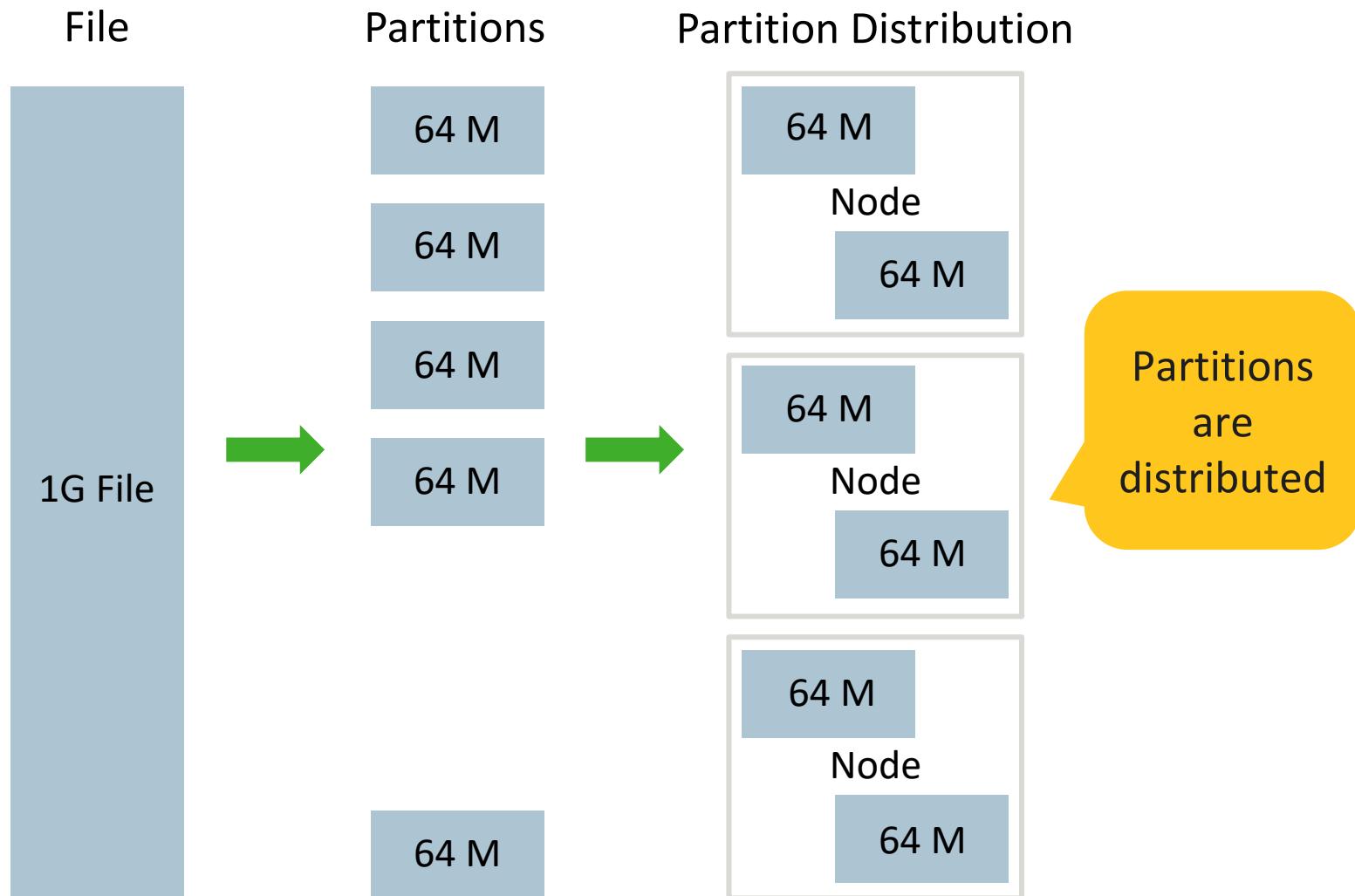
Create: Read a log file (e.g. from HDFS)

Filter: Keep lines starting with "ERROR"

Action: collect the results

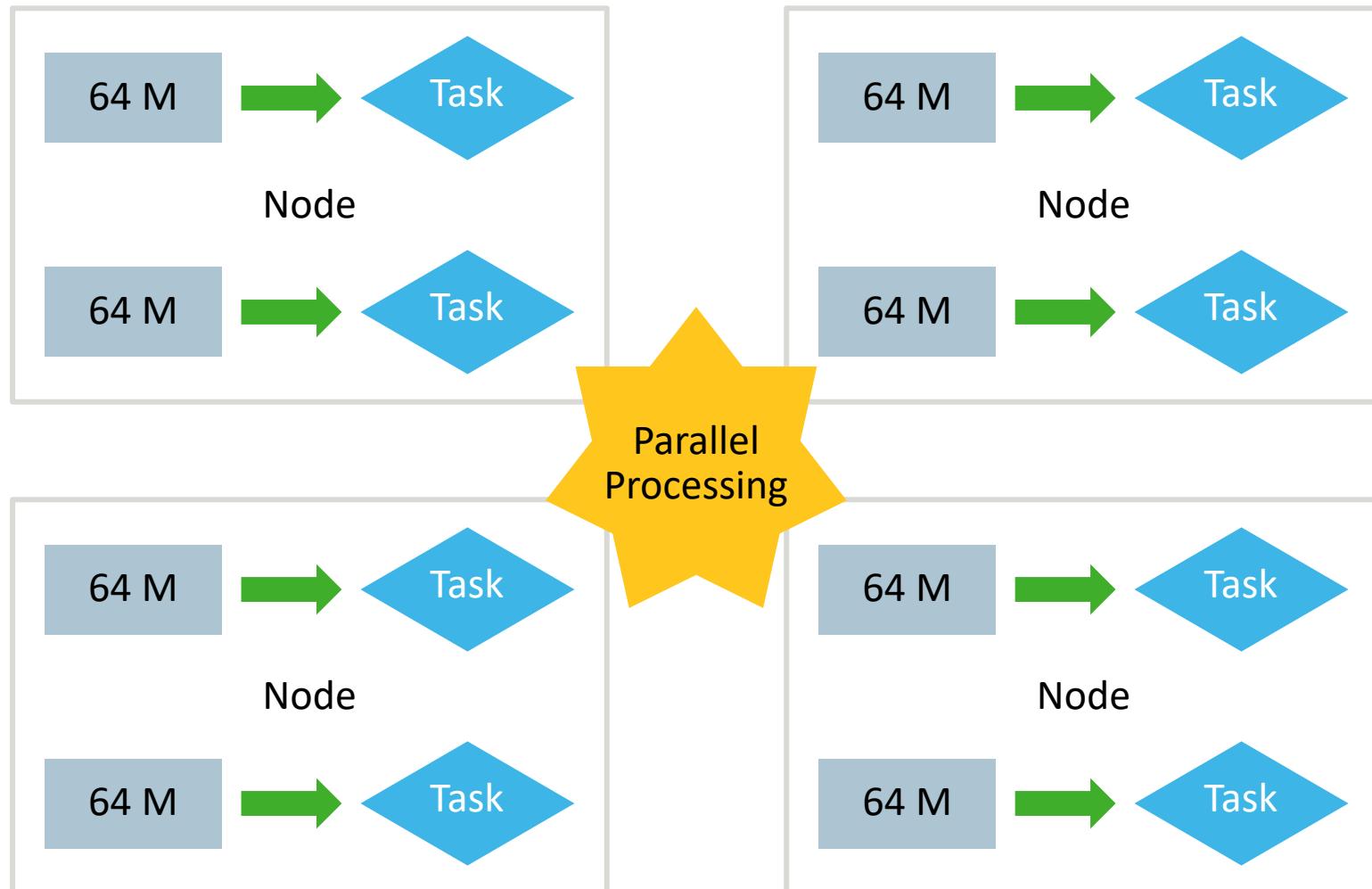
RDD Partitioning

- Data in an RDD is partitioned around the cluster
 - e.g. With HDFS, Spark creates RDD partitions from HDFS blocks



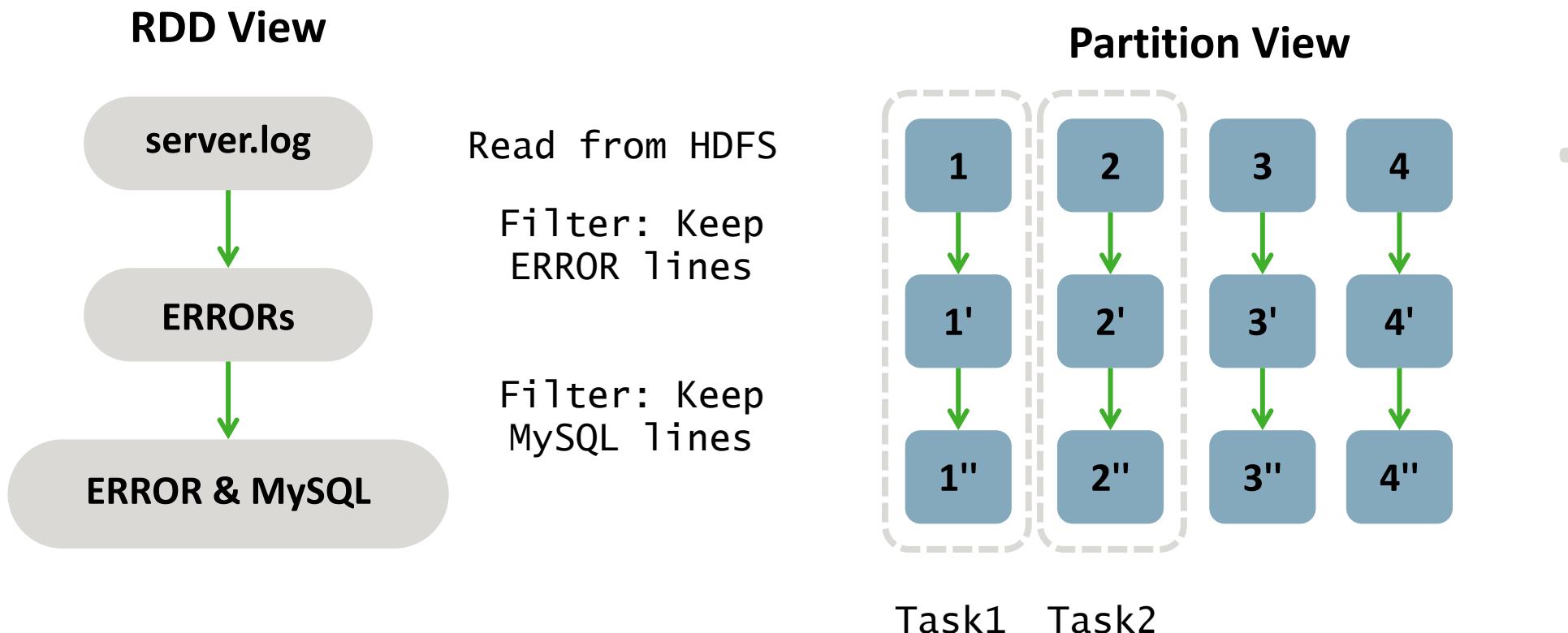
RDD Partitions and Parallel Processing

- Nodes execute tasks in parallel on the partitions
 - Spark will co-locate tasks with their data (HDFS block if HDFS)



Transformations Generate New Partitions

- A transformation on a partition creates a partition of the new RDD
- Succeeding transformations on it may be pipelined on a task ⁽¹⁾
- Often, it can all be done with in-memory data (fast)
- Some transformations require data shuffling (covered later)



Example: RDD Partitions

- Below, we illustrate partitioning of our server log RDD
 - Including sample log messages
 - We label the messages for tracking them in the next examples
 - e.g. msg1, msg2, etc.

Partition 1	Partition 2	Partition 3
INFO, (msg1 ...MySQL...)	INFO (msg6 ...)	ERROR, (msg11 ...MySQL ...)
ERROR, (msg2 ...)	ERROR, (msg7 ...)	WARN, (msg12, ...)
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)	INFO, (msg13 ...)
INFO, (msg4 ...)	WARN, (msg9 ...)	WARN, (msg14 ...)
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)	INFO, (msg15 ...)

Example: RDD Transformation

- We transform (filter) the RDD — each partition works on its own data

Partition 1	Partition 2	Partition 3	RDD
INFO, (msg1 ...MySQL...)	INFO (msg6 ...)	ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)	WARN, (msg12, ...)	
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)	INFO, (msg13 ...)	
INFO, (msg4 ...)	WARN, (msg9 ...)	WARN, (msg14 ...)	
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)	INFO, (msg15 ...)	



Filter: Keep
ERROR lines

Partition 1'	Partition 2'	Partition 3'	RDD '
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)		
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)		
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)		

Example: RDD Transformation

- We transform (filter) the RDD again

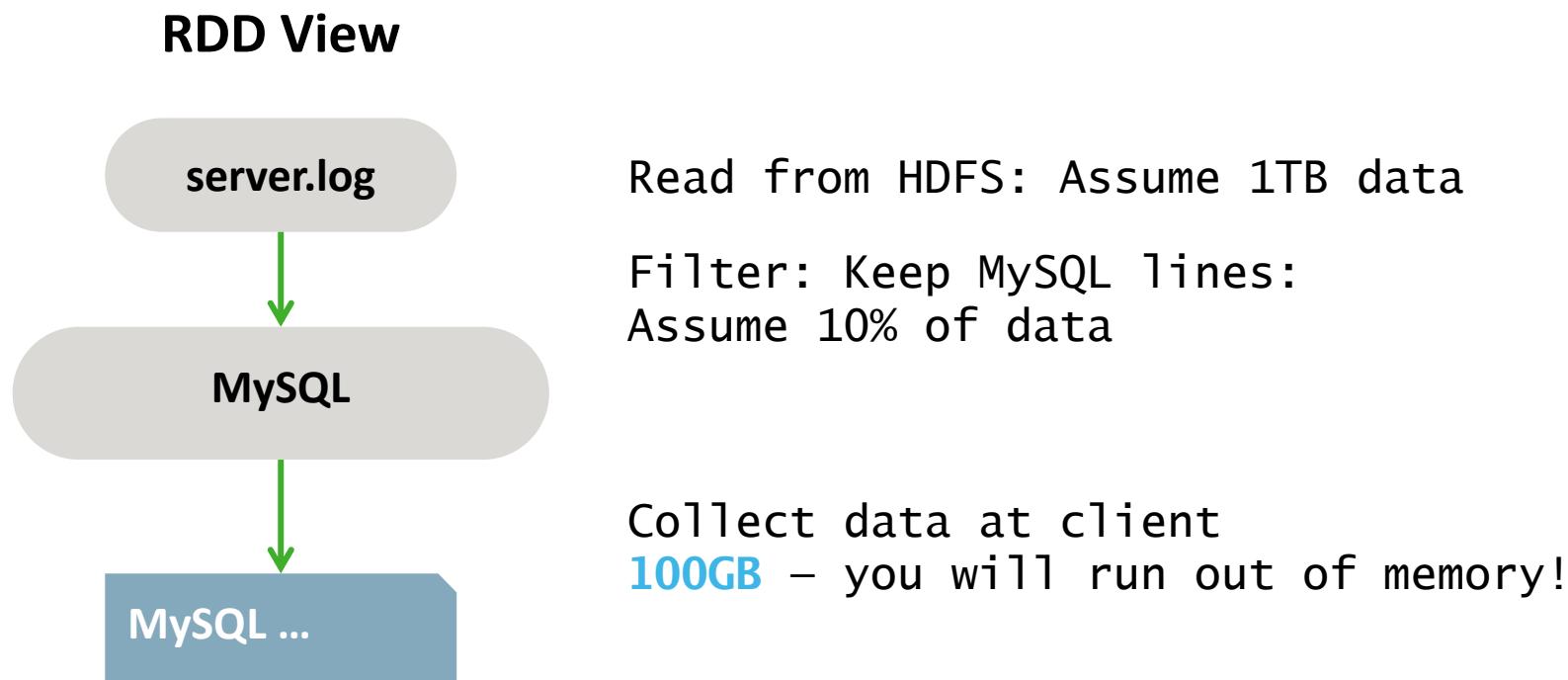
Partition 1'	Partition 2'	Partition 3'	RDD '
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)		
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)		
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)		

↓ Filter: Keep MySQL lines

Partition 1''	Partition 2''	Partition 3''	RDD ''
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg3 ...MySQL ...)			
ERROR, (msg5 ...MySQL...)			

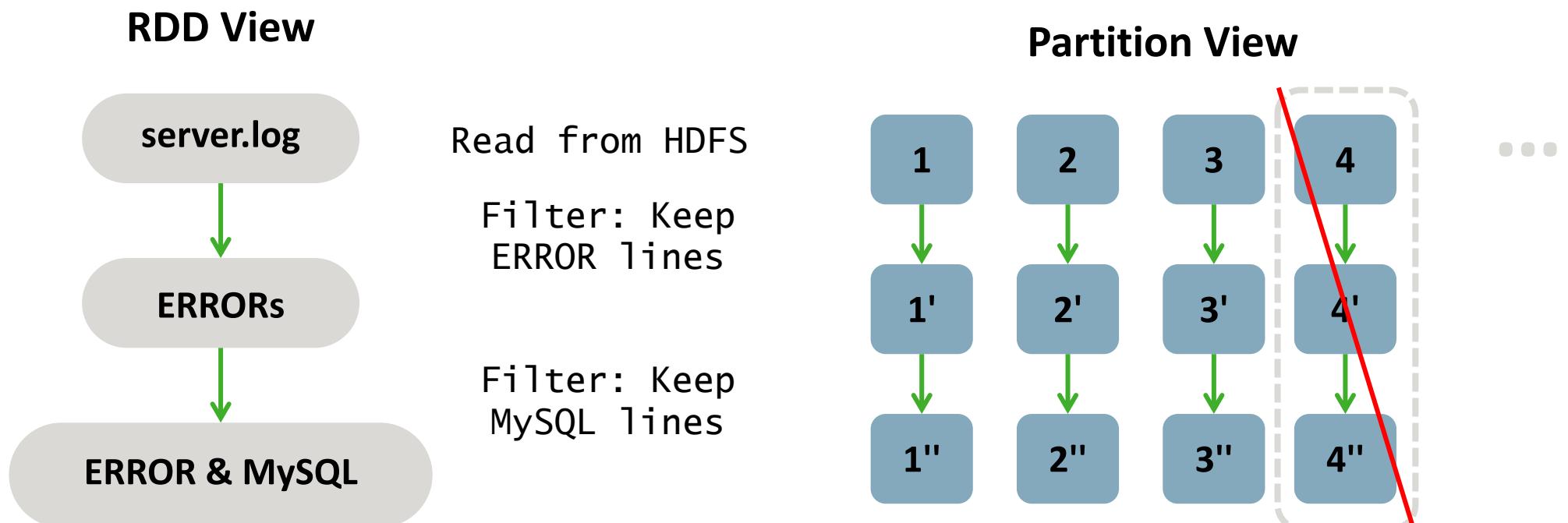
Careful with Data to Client

- Don't transfer large data sets to your client!
 - You can easily run out of memory
 - OK to save results to a distributed file system like HDFS



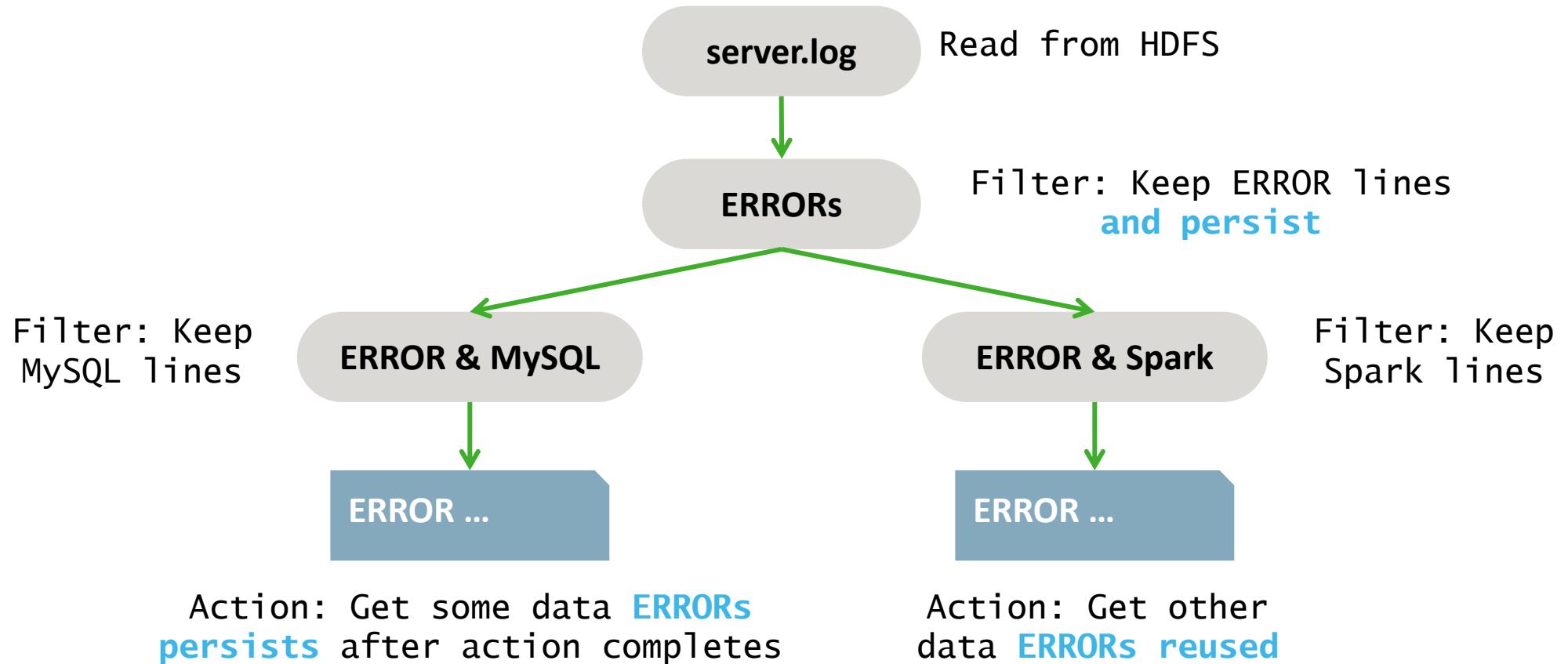
Fault Tolerance

- Spark tracks transformations that create an RDD
 - **Lineage**: The series of transformations producing an RDD
- A lost partition can be rebuilt from its lineage
 - e.g. If partition 4 is lost, Spark can read the HDFS block again, apply the transformations, and recover the partition
 - Efficient, and adds little overhead to normal operation ⁽¹⁾



RDDs Are Transient By Default

- Once an action completes, its RDDs disappear (by design ⁽¹⁾)
 - If you need one again, it's recomputed
- You can tell Spark to persist an RDD to keep it in memory
 - Useful for reusing an RDD that's expensive to create ⁽²⁾



A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some code or data. The background is blurred.

RDD Concepts

→ Working with RDDs

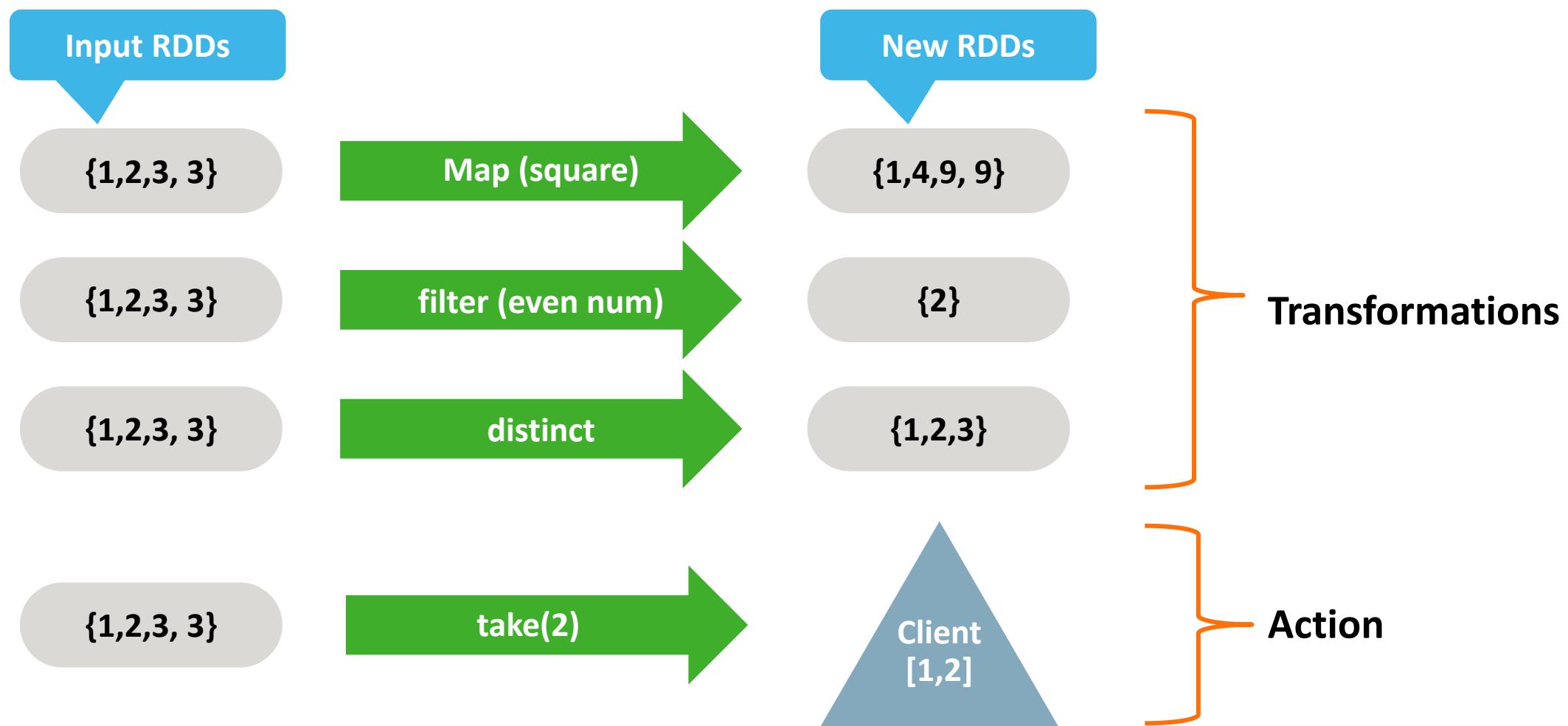
Creating an RDD

- Two ways to create
 - **Load data file(s)**: From local or distributed file system
 - **Parallelize a collection**: For small data or testing
 - It will all have to fit in memory on your driver node

```
// Turn a Scala collection into an RDD  
> val numbers = sc.parallelize(List(1,2,3,3))  
  
// Create from local file  
> val oneFile = sc.textFile("README.md")  
  
// Create from multiple files  
> val multiFile = sc.textFile("data/mllib/*.txt")  
  
// Create from a file in HDFS  
> val hdfsFile =  
    sc.textFile("hdfs://namehost:9000/logs/log20170301.txt")
```

Transformation/Action Examples

- Below, we illustrate three transformations and an action
 - Let's look at how to code some of these
 - We'll be looking at other APIs soon (Dataset API)



map() Details

- **map(f: (T)=>U)**: Applies function f to all elements
 - f() takes one argument, returns a mapped value ⁽¹⁾
 - **Return**: New RDD of mapped elements (may be a different type)



- Below is a simple map example
 - Creates a new RDD containing squares of the input RDD
 - Argument to map() is an anonymous function
 - See notes for a brief discussion ⁽²⁾

```
> val numbers = sc.parallelize (List(1,2,3,3))
// Map numbers RDD by squaring each element (1)
> val squares=numbers.map(x=> x*x)

// Collect all data in the RDD
> squares.collect()
res0: Array[Int] = Array(1, 4, 9, 9)
```

filter() Details

- **filter(f(T=>Bool))**: filters elements on predicate f()

- f() takes one argument, returns a boolean ⁽¹⁾
- **Return:** New RDD with elements where f(element)==true
 - It has the same element type as the parent



- Below is a simple filter example
 - Creates a new RDD containing odd numbers from the input RDD

```
> val numbers = sc.parallelize (List(1,2,3,3))

// Filter numbers RDD by squaring each element (1)
> val odds=numbers.filter(x=> x%2 == 1)
// Alternatively numbers.filter(_%2 == 1) (2)

> odds.collect()
res1: Array[Int] = Array(1, 3, 3)
```

Actions Overview

- Below, we illustrate some common actions and results

```
> val numbers = sc.parallelize (List(1,2,3,3))
> val odds=numbers.filter(x=> x%2 == 1)

// Get count of RDD
> numbers.count()
res2: Long = 4
> odds.count()
res3: Long = 3

// Get first two elements
> odds.take(2)
res4: Array[Int] = Array(1, 3)
```

Lab 3.1: RDD Basics

In this lab, we will create and work with RDDs

reduce() Details

- **reduce($f: (T, T) \Rightarrow T$):** reduces elements using f
 - $f()$ operates on two elements of the RDD, returns one element
 - For example, adding two numbers together
 - **Return:** Single element of same type
 - f is applied repeatedly until only one element left (the result)



- Below, we show two examples of reduce

```
> val numbers = sc.parallelize (List(1,2,3,4))

// Reduce by adding all numbers together
> numbers.reduce ((a,b)=> a+b)
// Alternatively numbers.reduce(_+_)
res16: Int = 10

// Reduce by multiplying (factorial)
> numbers.reduce ((a,b)=> a*b) // Or reduce(_*_)
res19: Int = 24
```

flatMap() Details

- **flatMap(f: (T) ⇒ TraversableOnce[U]):** applies function f to all elements, then flattens the results
 - f returns an object that can be iterated over (e.g. a collection)
 - The elements in each iterator are then combined to create the RDD ("flattened")



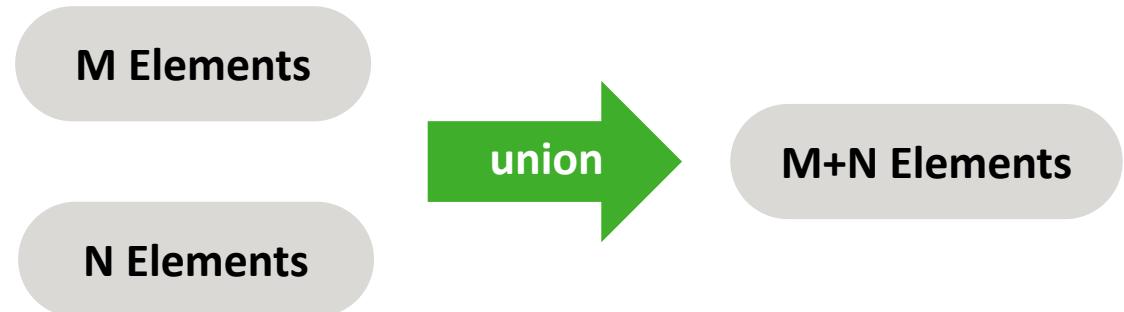
```
> val numbers = sc.parallelize (List(1,2,3))

// Map each number by creating list of 3 numbers
> val mapped = numbers.flatMap(a=> List(a-1, a, a+1))
mapped: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[4] at flatMap at
<console>:23

> mapped.collect // 9 element - 3 from each original
res14: Array[Int] = Array(0, 1, 2, 1, 2, 3, 2, 3, 4)
```

union() Details

- **union(other: RDD[T]): RDD[T]**: Returns union of this RDD with another RDD
 - Operates on two RDDs
 - Duplicates are included (remove with distinct())



```
> val odds = sc.parallelize (List(1,3,5,7))
> val evens = sc.parallelize (List(2,4,6,8))

// Create union
> val all = odds.union(evens)
all: org.apache.spark.rdd.RDD[Int] = UnionRDD[2] at union at <console>:25

> all.collect.sorted
res20: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8)
```

RDD Transformations Summary (1 of 2)

RDD r = {1,2,3,3}

Transformation	Description	Example	Result
map(func)	apply func to each element in RDD	r.map(x => x*2)	{2,4,6,6}
filter(func)	Filters through each element when func is true (aka grep)	r.filter(x=> x % 2 == 1)	{1,3,3}
distinct	Removes dupes	r.distinct()	{1,2,3}
flatMap	Like map, but can output more than one result per element		
mapPartitions	Like map, but runs on the whole partition not on each element		

RDD Transformations Summary (2 of 2)

RDD r1 = {1,2,3,3}

RDD r2 = {2,4}

Transformation	Description	Example	Result
union(RDD)	Merges two RDDs (duplicates are kept)	r1.union(r2)	{1,2,3,3,2,4}
intersection (RDD)	Returns common elements in two RDDs	r1.intersection(r2)	{2}
subtract(RDD)	Takes away elements from one	r1.subtract(r2)	{1,3,3}
sample	Take a small sample from RDD		

RDD Actions Summary

- Note that actions return values, not an RDD
 - e.g. count() returns an long, and take() returns an Array
 - And trigger execution of the transformation DAG

RDD r = {1,2,3,3}

Action	Description	Example	Result
count()	Counts all records in an rdd	r.count()	4
first()	Extract the first record	r.first ()	1
take(n)	Take first N lines	r.take(3)	[1,2,3]
collect()	Gathers all records for RDD. All data has to fit in memory of ONE machine (don't use for big data sets)	r.collect()	[1,2,3,3]
saveAsTextFile()	Save to storage		
	... Many more — see docs ...		

More Complex Transformation

- The RDD API has considerably more capability
 - In particular, operations on key-value pairs
 - These are generally used for aggregation (grouping, counting, etc.)
- We will first introduce the DataFrame/Dataset API
 - Has a simpler interface
- We'll then introduce more complex operations
 - Focusing on the DataFrame/Dataset API
 - Supplying some examples of programming with the RDD API

MINI-LAB: Review RDD Documentation

- We'll take a few minutes to briefly review the RDD API docs
 - They're hosted on the Spark Apache website

Mini-Lab

- Browse to <http://spark.apache.org/docs/latest/>
 - On the top menu bar, go to [API Docs | Scala](#)
 - In the left hand pane, find the [org.apache.spark.rdd](#) package
 - Hint: Click "display packages only" to collapse the package listing
 - Find org.apache.spark.rdd and click the [Show](#) link to expand it (only)
 - Within that package, click on the RDD entry
 - Alternatively, can search for RDD in the search filter on the upper left
 - This brings you to the RDD API documentation
- Review the RDD API briefly
 - In particular, look at `map()`, `filter()`, `count()`, and `take()`

Lab 3.2: Operations on Multiple RDDs

In this lab, we will work with some of the operations that use multiple RDDs

Review Questions

- What is an RDD?
- What happens when you execute a transformation on an RDD?
- Where does data in an RDD reside?
- What are some typical transformations used with RDDs?

Session Summary

- **RDD: Resilient Distributed Dataset**
 - Distributed collection of elements partitioned across nodes in a cluster
 - Core abstraction in the Spark engine
- RDDs support **transformations** and **actions**
 - **Transformations** create a new RDD from an existing RDD (e.g. mapping data)
 - They are lazy — they are stored in a DAG when encountered
 - **Actions** (e.g. collect) extract information — triggering execution of the DAG

Session Summary

- Data in an RDD is **partitioned** across a Spark cluster
 - Each node holds part of the data, which is then worked on in parallel
- Common operations on RDD data include:
 - **filter**: filter out elements on a predicate
 - **map**: Apply a function to each element, yielding a new element
 - **collect**: Get the data at the client

A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt, standing in what appears to be a lecture hall or conference room. He is gesturing with his hands while speaking. In the foreground, the back of another person's head and shoulders are visible, suggesting an audience member. The background shows rows of seating and overhead projector equipment.

Session 4: Spark SQL, DataFrames, and Datasets

- Overview
- SparkSession and Data Load/Store
- Introducing DataFrame/Dataset
- The Query DSL
- Datasets
- flatMap, explode, split

Session Objectives

- Learn Spark SQL's API and capabilities
 - Use them for creating efficient transformations
 - Understand the advantages
- Load and store data in common supported formats
 - Use SparkSession and DataFrameReader/Writer
 - Work with JSON, Parquet and other data
- Use the DataFrame/Dataset API (untyped and typed API)
 - Use the Query DSL to write transformations
- Explore the many capabilities in the API
 - Cover important principles and methods
 - But **NOT cover** every method of every type!
 - For a particular method, you have the javadoc



Overview

SparkSession and Data Load/Store

Introducing DataFrame/Dataset

The Query DSL

Datasets (The Typed API)

flatMap, explode, split

Limitations of RDDs

- Low level API
 - You specify details (the how) not intent (the what)
 - API has little intelligence for dealing with common data formats
 - e.g. JSON
- Opaque to Spark engine (uses arbitrary lambdas)
 - Queries can't easily be optimized by Spark
 - Not hard to write inefficient transformations
 - Often not obvious
 - And Spark can't make them better
- No support for SQL-like querying
 - Limits applicability
 - SQL is well known

Introducing Spark SQL

- Module for **structured data** processing
 - Built on top of Spark Core, and way more than just SQL
 - Has many features, including those below ⁽¹⁾
- Supports **schema/structure** for Spark data
 - Structured as table/rows/columns
 - **Loads data** from many sources
- Provides **high level API** for data processing
 - **Dataset/DataFrame** API and **SQL**
- **Optimizes** execution for high performance
 - **Catalyst** query optimizer optimizes queries
 - **Tungsten** optimizes memory/CPU usage
 - Performance stable across APIs (Scala, Python, Java, R, ...)

Supported Data Formats

- Can load data from many data sources
 - Built-in support for many common formats
 - Many other formats supported with external libraries
 - Can deduce schema structure for some of these



DataFrames and Datasets

- Distributed collections of data (as are RDDs)
 - **DataFrame**: Adds **schema** information
 - **Dataset**: Has schema, and adds **compile-time type safety**
- **Query DSL** (Domain Specific Language)
 - Comprehensive querying API — filter, map, groupBy, etc.
 - Fluent interface very easy to use
- **SQL** queries fully supported
 - Familiar to many developers
 - More expressive in many situations
 - e.g. computing multiple aggregates in one SQL statement

Simple Processing Example

- Consider data files related to people
 - Name, gender, and age
 - Consider getting the **average age** for each **gender**?
- We'll illustrate this with RDDs and DataFrames
 - RDDs will use the text input file below (*people.txt*)
 - DataFrames will use the JSON input file at bottom (*people.json*)

```
John M 35
Jane F 40
Mike M 20
Sue F 52
```

```
{"name": "John", "gender": "M", "age": 35 }
{"name": "Jane", "gender": "F", "age": 40 }
{"name": "Mike", "gender": "M", "age": 20 }
{"name": "Sue", "gender": "F", "age": 52 }
```

RDD Processing Example

- Below, we calculate average age via RDDs
 - It's not the only way, and may not be the best
- But — it illustrates the issues, such as:
 - **What the heck** is that? Can you tell at a glance?
 - See the notes if you want more detail
 - **Is it correct?** (It is)
 - But how can you tell?????
 - Spark **doesn't understand the details** either

```
> val folksRDD=sc.textFile("people.txt") // Get the data
> folksRDD.map(_.split(" "))           // Split it into fields
    .map(x => (x(1), Array(x(2).toDouble, 1))) // Pair up the data
    .reduceByKey( (x,y) => Array(x(0)+y(0), x(1)+y(1)) ) // Count
    .map(x => Array(x._1, x._2(0)/x._2(1))) // Figure average
    .collect                                // Get results

res1: Array[Array[Any]] = Array(Array(F, 46.0), Array(M, 27.5))
```

DataFrame Processing Example

- Below, we calculate average age with a DataFrame
- Does it feel different?
 - Can you guess what it does?
 - With a little thought, yes — even if you've never seen it before
 - Is it correct? (It is)
 - And looking at it, you can see it makes sense
 - Spark also **understands the details** and can optimize

```
> val folksDF=spark.read.json("people.json") // Get the data
> folksDF.groupBy($"gender") // Group by gender
    .agg(avg($"age")) // Get average age of groups
    .show // Display data

+-----+
|gender|avg(age)|
+-----+
|      F|     46.0|
|      M|     27.5|
+-----+
```

SQL/DataFrame Processing Example

- Below, we calculate the average via DataFrames and SQL
- Even simpler than the previous example
 - Most folks can understand SQL
 - Spark **understands** the details and can optimize

```
> val folksDF=spark.read.json("people.json") // Get the data
> folksDF.createOrReplaceTempView("people") // Setup for using SQL

// Get the average by gender
> spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
+-----+-----+
|gender|avg(age)|
+-----+-----+
|      F|     46.0|
|      M|     27.5|
+-----+-----+
```

Dataset Processing Example

- Below, we illustrate calculation of the average with a Dataset
 - Similar to DataFrames — but type safe
 - Uses a lambda to group — more on this later

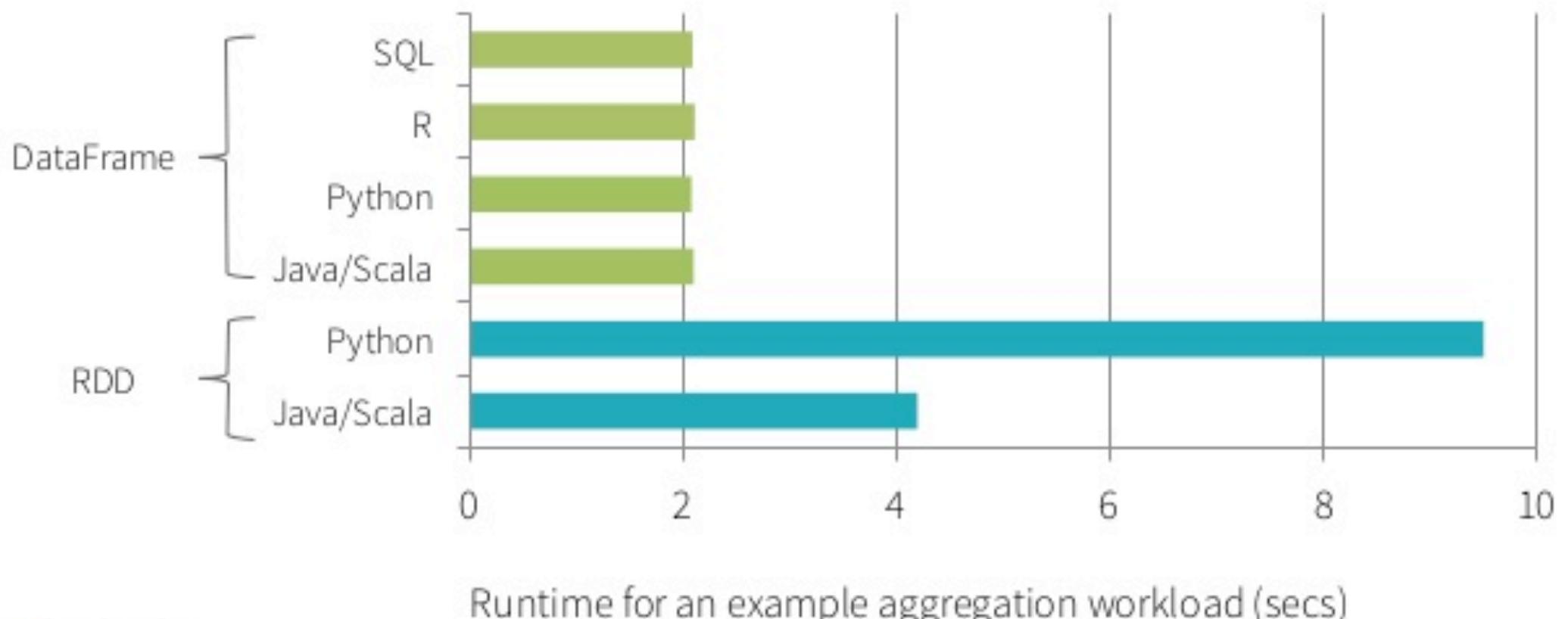
```
> val folksDF=spark.read.json("people.json") // Get the data  
  
// Declare case class and create Dataset from it  
> case class Person (name: String, gender: String, age: Long)  
> val folksDS=folksDF.as[Person]  
> folksDS: org.apache.spark.sql.Dataset[Person] = [age: bigint... ]  
  
// Get the average by gender  
> folksDS.groupByKey(T => T.gender).agg(avg($"age").as[Double]).show  
+----+-----+  
|value|avg(age)|  
+----+-----+  
|    F|    46.0|  
|    M|    27.5|  
+----+-----+
```

Catalyst Optimizer

- **Query optimizer** for structured data
 - Works with Dataset, DataFrame, and SQL
 - Automatically transforms queries to execute efficiently
- Easily extended to support
 - New data sources including semi-structured data like JSON
 - "Smart" data stores to which you can push filters (e.g. HBase)
 - User-defined functions and types
- Catalyst understands the data's schema and the queries
 - Providing many optimization opportunities
 - Operations evaluated lazily, so complete sequence can be analyzed for optimizations
 - **Lambdas can't** be analyzed (they are opaque)

Greatly Improved Performance

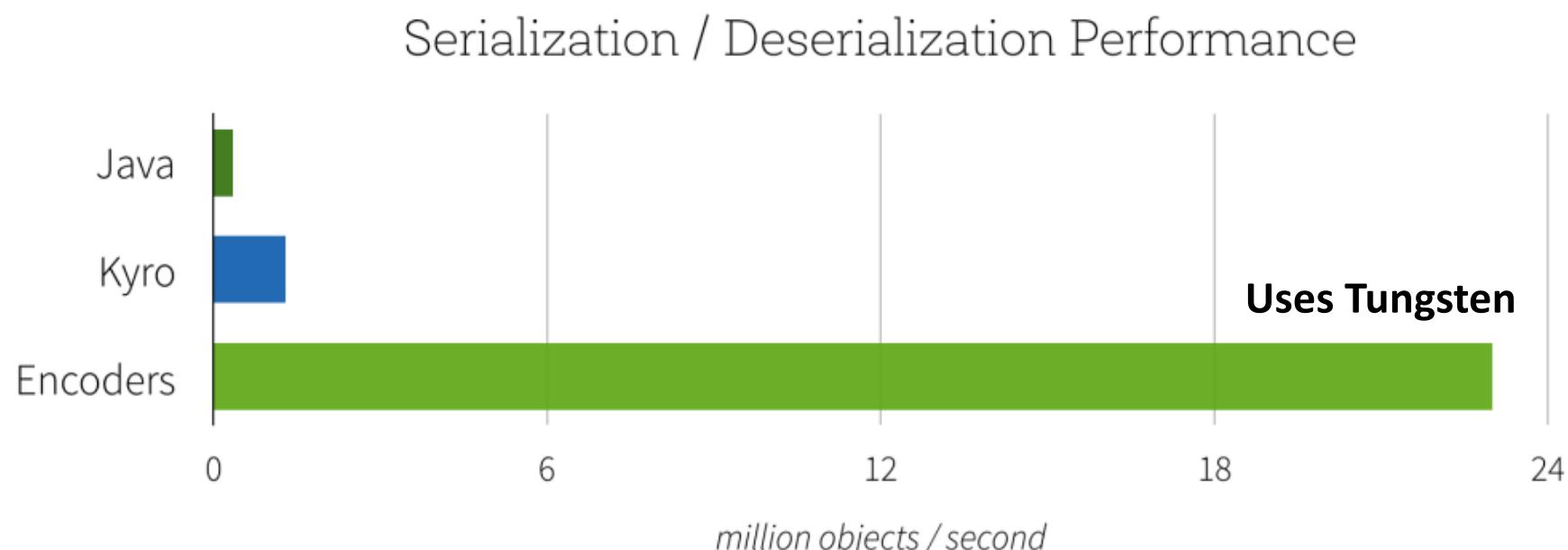
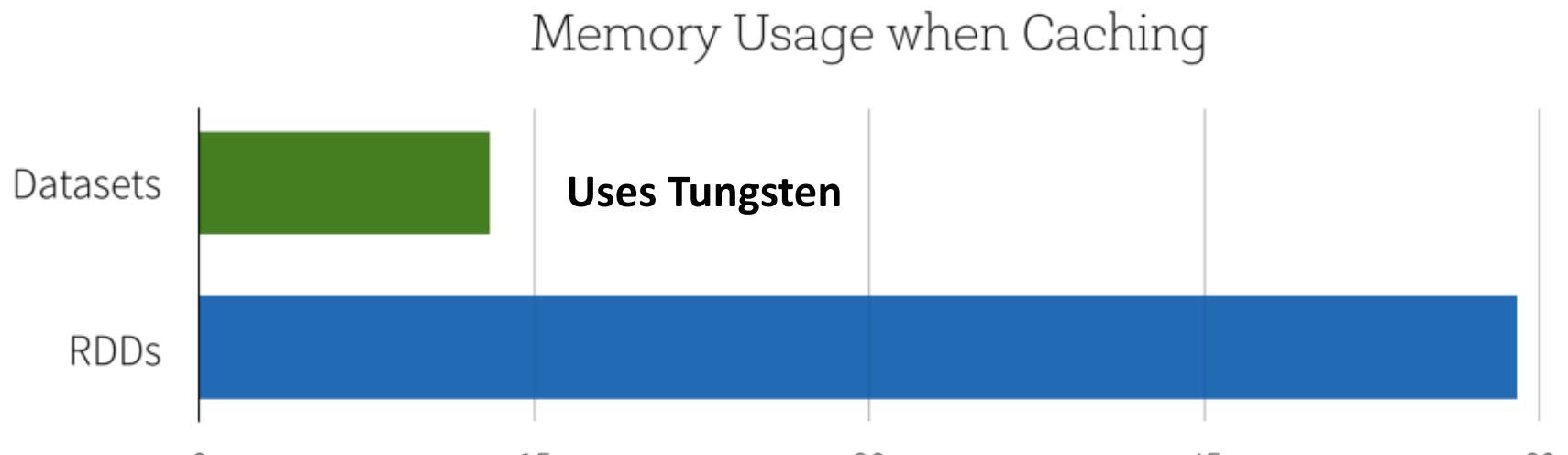
- We illustrate this below
 - DataFrame performance much better than RDD
 - Performance similar across languages (e.g. Scala and Python)
 - They all now generate similar execution plans



Tungsten Optimizer

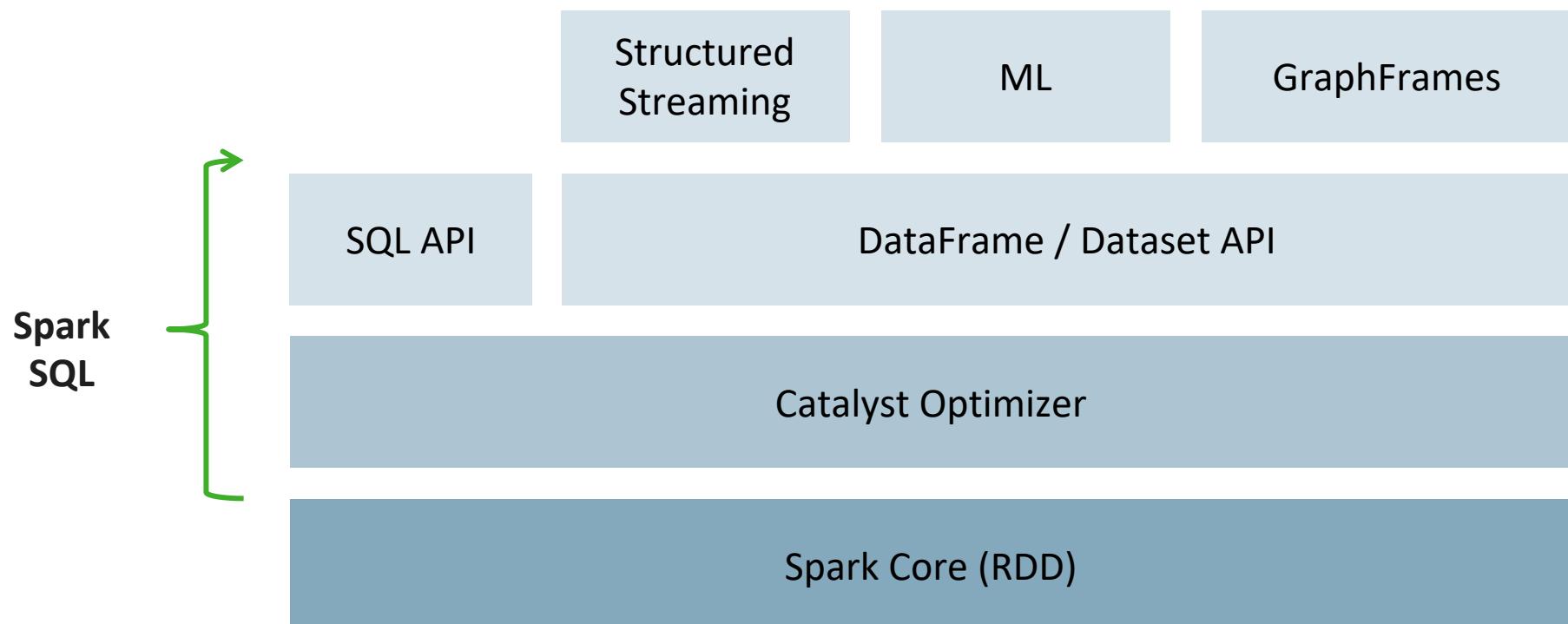
- Substantially improves **memory/CPU performance** of Spark
 - CPU/memory are increasingly becoming bottlenecks
 - Spark I/O, network performance, and data storage are good ⁽¹⁾
 - They are utilized better, but memory/CPU utilization has not kept up
 - Pushes performance closer to hardware limits
- Stores in-memory data in **off-heap binary** format
 - Instead of in Java objects (on-heap)
 - Reduces memory size, eliminates garbage collection (GC)
 - Can operate directly on binary data (no deserialization)
 - Understands and optimizes for different cache (L1, L2, ...)
- **Generates code** for expression evaluation
 - No need to deserialize for many operations
- See next slide for comparison to RDD

Reduced Memory / Increased Performance



Spark Structure with DataFrames/Datasets

- Spark SQL is a core building block
 - Supports Catalyst optimizations for other modules
 - Other modules are evolving
 - GraphX (RDD-based) => GraphFrames (DataFrame-based)
 - These modules in varying stages of completeness



Summary

- Spark SQL: Higher level API for Spark computations
 - Works with structured (and semi-structured) data
- Components include:
 - DataFrame/Dataset API
 - SQL API
 - Catalyst query optimizer
 - Tungsten execution optimizer
- Becoming the **primary interface** for Spark development
 - New development is DataFrame/Dataset based
 - Including higher level modules like Graph processing (GraphFrames)
 - APIs are still evolving — still need RDDs sometimes

Overview



SparkSession and Data Load/Store

Introducing DataFrame/Dataset

The Query DSL

Datasets (The Typed API)

flatMap, explode, split

Core API Types

- All types discussed here in package: `org.apache.spark.sql`
- **DataFrame**: Distributed collection with a **schema**
 - Synonym for `Dataset[Row]` — more on this soon
- **Dataset**: **Strongly typed** distributed collection with schema
- **Column**: A column in a DataFrame
 - Used to create expressions in the query DSL
- **Row**: Represents data in tabular format (name/value columns)
- **SparkSession**: Entry point for Spark SQL API
 - Replaces older SQLContext and HiveContext types
 - Pre-created in Spark shell (Creation covered later)
- **DataFrameReader/Writer**: To load/store data

SparkSession (API Entry Point)

- Capabilities of the class include:
 - Reading data files (via **DataFrameReader**)
 - Creating DataFrame and Dataset instances
 - Executing SQL queries
- Instances accessed via a builder (a factory)
 - Builder is part of **object** SparkSession
 - Companion object to **class** SparkSession⁽¹⁾
 - We illustrate this below, more detail later⁽²⁾
 - REPL pre-creates a session in **spark** var
 - `getOrCreate()` returns this existing one

```
> SparkSession.builder.getOrCreate
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5

> spark
org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@748321c5
```

DataFrameReader

- Interface to load data from external storage
 - Acquire via `SparkSession.read()`
- Built-in support for reading popular formats
 - Including `inferring the schema` automatically
 - json, parquet, csv, jdbc to relational database, hive, and more
- Below, `spark.read()` returns a DataFrameReader
 - `json("people.json")` loads data from the *people.json* file
 - Data must be in JSON Lines format
 - One JSON object per line, plus other limitations ⁽¹⁾

```
val folksDF=spark.read.json("people.json") // Get the data
```

DataFrameReader API

- Methods for loading particular formats
 - `csv(path: String)`: Load from CSV file
 - `jdbc(...)`: Load from relational database (arguments omitted)
 - `json(path: String)`: Load from JSON file
 - `parquet(path: String)`: Load from parquet storage
 - `text(path: String)`: Load from text file
 - Many variants of each
- Can also specify details explicitly, as shown below
 - `format()`: Specify data format
 - Provide fully qualified classname or built-in format like "json"
 - `schema()`: Specify data schema
 - Details not shown (via StructType/StructField instances)

```
val folksDF=spark.read.format("json").schema(...).load("people.json")
```

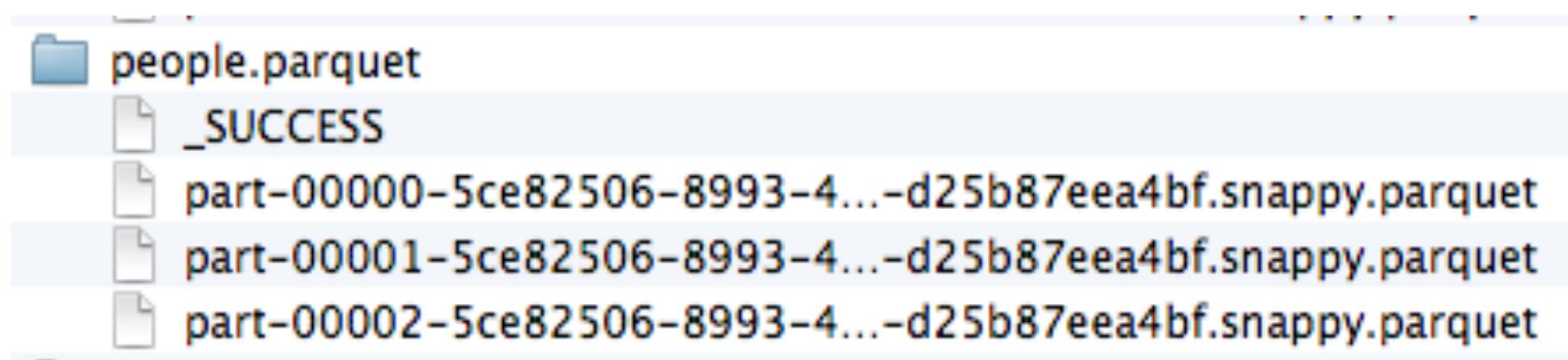
DataFrameWriter

- Interface to store data in external storage
 - Acquire via `SparkSession.write()`
 - Similar in capabilities to DataFrameReader
 - Built-in support to write csv, jdbc, json, parquet, and text
- Below, we illustrate storing data in parquet format
 - It has been read in JSON format
 - We've (easily) transformed the data (JSON => parquet)

```
val folksDF=spark.read.json("people.json") // Get the data (JSON)  
folksDF.write.parquet("people.parquet") // Write the data (parquet)
```

Multiple Files from Writes

- **Storage format** for DataFrame writes is
 - **Top level folder** with your specified name
 - **Multiple files** within the folder holding the data
 - One file for each partition, which you'll generally write to a distributed file system (e.g. HDFS)
 - Generally what you want — but you can also write as one file ⁽¹⁾
- We illustrate the output of writing *people.parquet* below
 - Assuming three partitions



Fluent Interfaces

- **Fluent interfaces** are designed to be readable and to flow
 - Making it easy to read/write
 - Generally let you chain calls, and often use a builder
- The Spark SQL API uses fluent interfaces
 - As in our previous read example repeated below
 - We'll see this style throughout our coverage
 - A (hypothetical) non-fluent version is illustrated at bottom
 - More verbose, and harder to read

```
val folksDF=spark.read.format("json").schema(...).load("people.json")
```

```
// Hypothetical non-fluent example
val reader = spark.read
reader.setFormat("json")
reader.setSchema(...)
val folksDF=reader.load("people.json")
```

MINI-LAB — Review Documentation

Tasks to Perform

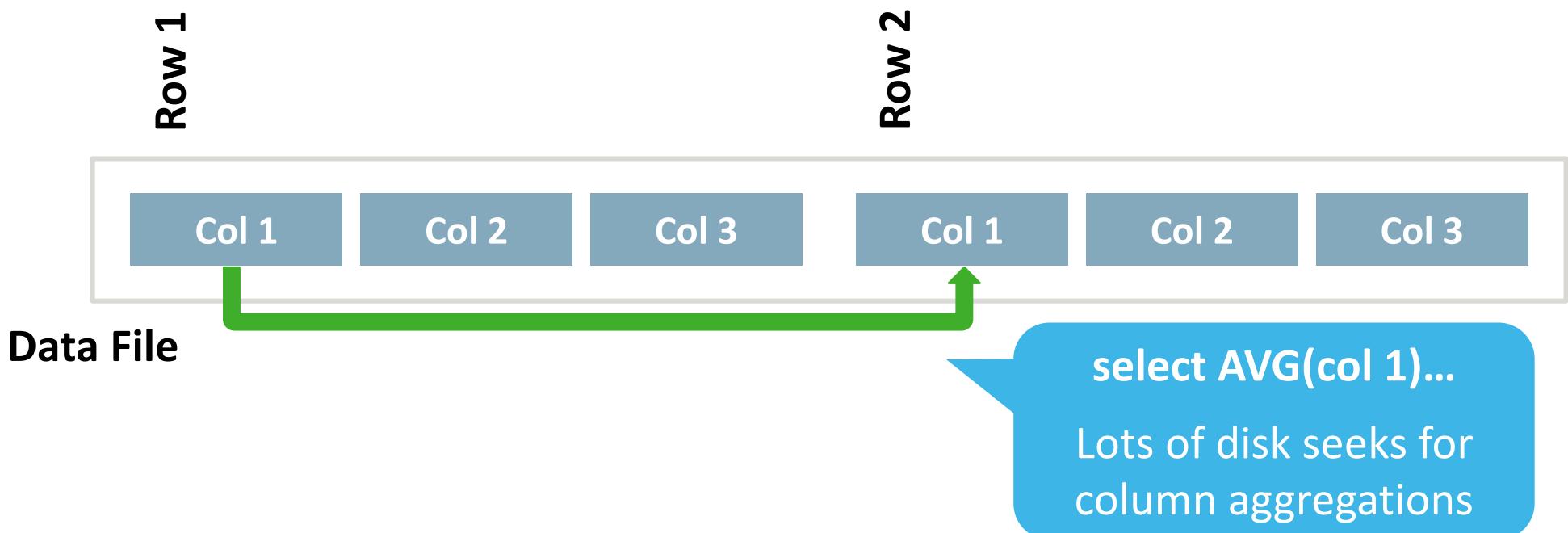
- Go to the Spark docs at <http://spark.apache.org/docs/latest/>
 - In the left hand pane find package `org.apache.spark.sql`
 - Collapse the packages and find the one we want as previously
- Review the following in the package
 - **Class `SparkSession`** (click the `C` next to it in the package listing)
 - Review at least the `read()` and `createDataFrame()` methods
 - **Object `SparkSession`** (click the `O` next to it in the package listing)
 - Look at the various methods
 - Follow a link to the `Builder` class and review it
 - **DataFrameReader** and **DataFrameWriter**
 - Review at least the methods we covered in this section

Data Formats

- Spark supports many data formats
- We'll give a brief overview of the most common formats
 - As well as the Spark support
 - We'll illustrate read support — write support is similar
- Support includes:
 - Row and column based formats
 - Text based formats (e.g. JSON and CSV)
 - Binary formats

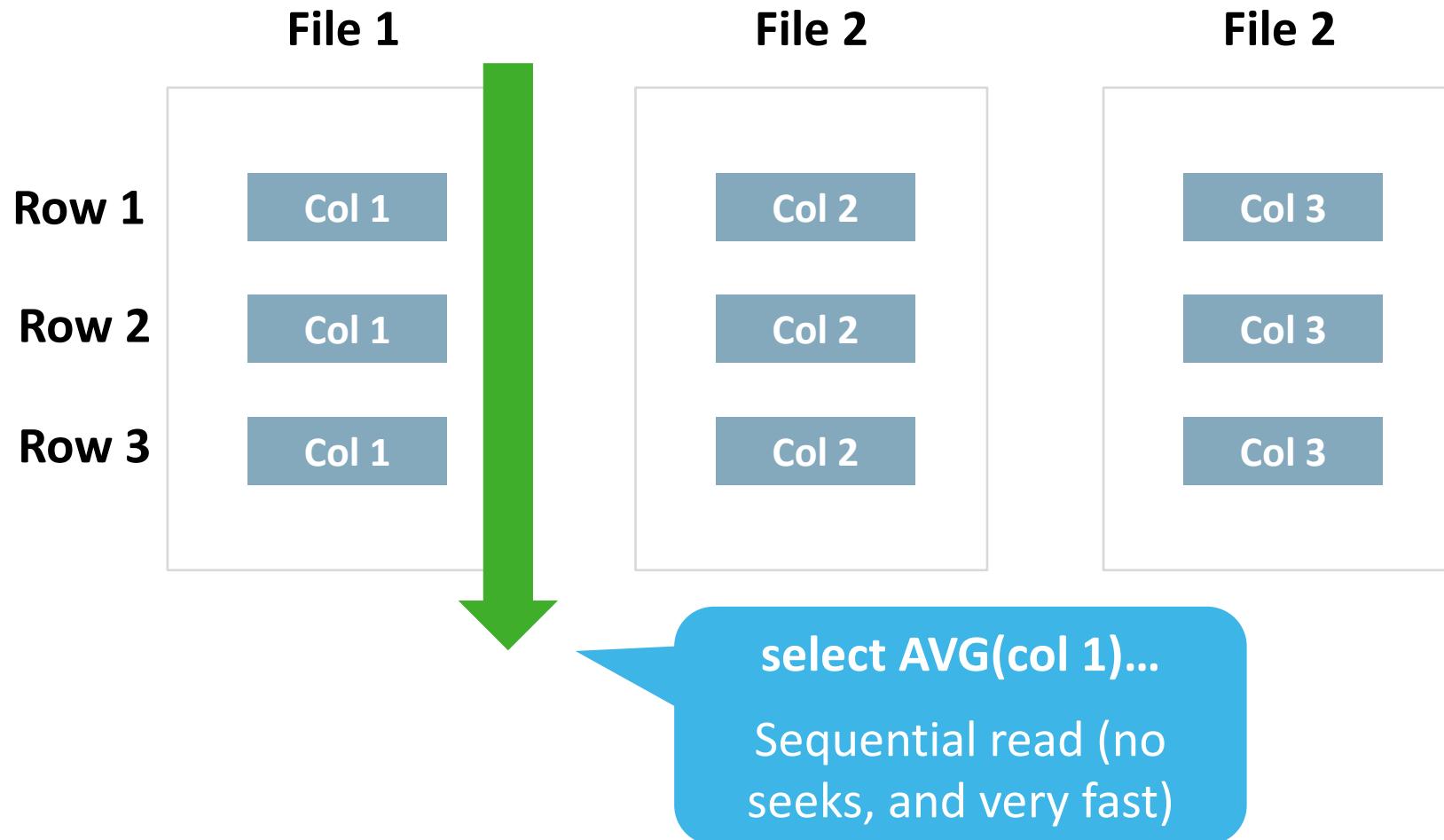
Overview: Row Based Stores

- Stores rows physically together
 - Generally heavily indexed (e.g. relational DB)
 - Good for select * type queries
 - Not so good for aggregation (e.g. get average)



Overview: Column Based Stores

- Stores **columns** physically together (not rows)
 - Optimized for fast column based aggregations
`select MAX(temp) from sensors;`
 - Not as good for `select*` type queries



Common Text-based Data Formats

- All are row-based
- **JSON**: (JavaScript Object Notation)
 - Lightweight data interchange format
 - Read via `DataFrameReader.json()`
 - Can automatically `infer` the schema (more on schema later)
- **CSV**: (Comma Separated Values)
 - Simple file format for tabular data
 - Read via `DataFrameReader.csv()`
 - Can automatically `infer` the schema
- Free form text
 - Read via `DataFrameReader.text()`
 - Generally parse text and apply a schema manually

Parquet

- Popular columnar data format (Apache project)
 - Binary storage, supporting efficient compression/encoding
 - Schema stored with data (file is **self-describing**)
 - Very efficient for column-based queries
 - Well supported in Hadoop ecosystem and elsewhere
- Has become format of choice in many systems
 - Well beyond Hadoop
- Read via **DataFrameReader.parquet()**

Other Formats

- **Avro:** Binary row-based format (Apache Avro project)
 - Schema stored as part of data (easy decoding)
 - Supports schema evolution/versioning
 - Supported by external spark-avro library
- **Hadoop-based** formats (binary / sequence)
 - Row-based, key-value pairs
 - Supported by several SparkContext methods (`binaryFile`, `hadoopFile`, `newAPIHadoopFile`)
 - RDD-based
- **Optimized Row Columnar** (ORC): Hybrid row/column format
 - Stores rows, within rows data stored in columnar format
 - Commonly used in Hive data stores
 - Spark can read using its Hive support

Lab 4.1: Data Formats

In this lab, we will read/write data in several formats

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

Overview

SparkSession and Data Load/Store



Introducing DataFrame/Dataset

The Query DSL

Datasets (The Typed API)

flatMap, explode, split

Overview

- **DataFrame**: Distributed collection of data with a **schema**
 - **Structured data**: Data organized into named columns
- The schema can be created in multiple ways, including:
 - **Inferred** from data (depending on the format)
 - **Declared** explicitly
 - We'll look at examples
- Diverse API for queries
 - **DSL**: Functional API plus expression language
 - **SQL** queries
 - **Lambdas** supported, but often not the best choice
- **Dataset** adds compile-time type safety (Covered later)

DataFrame/Dataset: Some History

- Introduced as **SchemaRDD** (1.0), renamed to **DataFrame** (1.3)
- **Dataset** type introduced in Spark 1.6
 - Separate type from DataFrame for backwards compatibility
- Dataset and DataFrame **unified** in 2.0
 - DataFrame is now a typedef for **Dataset[Row]** (Scala)
 - The API is defined in Dataset, and divided into sections
 - **Untyped** operations derive historically from DataFrame
 - **Typed** operations derive historically from Dataset
- A bit confusing — especially if you worked with early releases
 - And documentation can be unclear ⁽¹⁾
- We'll use **DataFrame** to refer to the untyped API

Creating DataFrames/Datasets

- We've seen creation via loading data from file
- Seq of case classes easily converted to DataFrame/Dataset
 - Via `toDF()` and `toDS()`, as illustrated below
- Can also be created via transformations (covered soon)

```
// Declare case class
> case class Person (name: String, gender: String, age: Long)

> val folks = Seq (Person("John", "M", 35), Person("Jane", "F", 40),
Person("Mike", "M", 20), Person("Sue", "F", 52))

// Create DataFrame
> val folksDF = folks.toDF
folksDF: org.apache.spark.sql.DataFrame = [name: string ...]

// Create Dataset
> val peopleDS = folks.toDS
peopleDS: org.apache.spark.sql.Dataset[Person] = [name: string ...]
```

Scala Magic: toDF and toDS

- **Seq** is a class in the Scala standard library
 - It, of course, doesn't define Spark's toDF and toDS methods
 - These are added via Scala **implicit conversions**
- **Implicit conversions** convert objects to another class
 - Which can seemingly add new methods to a type ⁽¹⁾
- Spark defines implicit conversions on Seq for toDF / toDS
 - In **spark.implicits** which is imported by the Spark Shell ⁽¹⁾
 - The shell automatically does "**import spark.implicits._**"
 - Must explicitly import in other environments
 - Spark uses many implicits
 - Will point them out for clarity when introduced
 - We won't go into detail on them

"Showing" the DataFrame

- `show()` displays the Dataset data in a tabular form
 - We illustrate below
 - Various versions (see the docs)
 - e.g. `show (numRows: Int)` displays numRows rows
 - Useful when doing ad-hoc queries

```
// Display the data
> folksDF.show
+---+---+---+
|name|gender|age|
+---+---+---+
| John|      M| 35|
| Jane|      F| 40|
| Mike|      M| 20|
| Sue |      F| 52|
+---+---+---+
```

The Schema

- Every DataFrame/Dataset has a **schema**
 - Describing its fields (with name, type, and nullability)
 - Data must conform to the schema
 - In this example, schema easily inferred from the case class
- **printSchema** prints to console in easy tree format
- **schema** displays the internal schema representation (more soon)

```
> folksDF.printSchema // Pretty schema display
root
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- name: string (nullable = true)

> folksDF.schema // Schema internal structure - more on this soon
res10: org.apache.spark.sql.types.StructType =
StructType(StructField(name,StringType,true),
StructField(gender,StringType,true), StructField(age,LongType,false))
```

How the Schema is Determined

- Many ways, including those below
 - We'll look at some in more detail
- **Inferred from MetaData:** Spark uses data's built-in schema
 - Parquet files, DB schema ⁽¹⁾
 - Scala case classes, JavaBean classes
- **Inference from Data:** Spark figures it out from data
 - Supported for JSON and CSV
- **Specified programmatically:** You do it via code
 - e.g. for text data
 - Multiple ways to specify schema

Inferring the Schema from JSON

- Seems simple: `spark.read.json("path-to-file")`
 - And magically you get your schema
- Some things to know about this:
 - It's **NOT lazy**: Spark **immediately** scans data to infer the schema
 - It's **fragile**: Any issues in your data will affect schema inference
 - We give an example below
 - See notes for the JSON file and other examples

```
> var folksBadDF = spark.read.json("data/people-bad.json")

> folksBadDF.show
+-----+-----+-----+
| _corrupt_record | age | gender | name |
+-----+-----+-----+
| {"name": "John", ... | null | null | null |
|                   null | 40 | F | Jane | // Remaining data omitted
```

Declaring the Schema Programmatically

- Below, we create a schema using StructType
 - Which is used in the internal schema structure
 - No table scan used to infer schema
 - Still fragile with data issues — e.g. assume a row has a bad age:
 - `{"name": "John", "gender": "M", "age": "35"}`

```
> import org.apache.spark.sql.types._ // Import schema types
> val mySchema = (new StructType).add("name", StringType).add("gender",
StringType).add("age", IntegerType)

// Resulting DF has the schema above
> var folksDF = spark.read.schema(mySchema).json("data/people.json")

> folksDF.show
17/06/01 13:57:38 WARN JacksonParser: Found at least one malformed records //
remaining warning omitted ...
+---+---+---+
| name | gender | age |
+---+---+---+
| null | null | null | // Whoops – row with bad age is all null.
| Jane | F | 40 | // Remaining data omitted.
```

Creating the Schema via Querying

- First, specify the schema as all strings
 - Then use DataFrame capabilities to create schema ⁽¹⁾

```
> val mySchema = (new StructType).add("name", StringType).add("gender",  
StringType).add("age", StringType) // Note that age is now a string  
  
// Resulting DF has the schema above (all strings)  
> var folksDF = spark.read.schema(mySchema).json("data/people.json")  
  
> val folksWithNewSchemaDF = folksDF.select(  
  'age.cast("integer"), 'name, 'gender ) // More detail on this later  
  
> folksWithNewSchemaDF schema  
... StructType(StructField(age, IntegerType, true),  
StructField(name, StringType, true), StructField(gender, StringType, true))  
  
> folksWithNewSchemaDF.show  
+---+---+---+  
|age|name|gender|  
+---+---+---+  
| 35|John|      M| // This worked with same data as previous slide !  
| 40|Jane|      F| // Remaining data omitted
```

Dataset API: Operation Categories

- **Actions:** Retrieve data, display it, process locally
 - e.g. `collect()`, `first()`, etc. — many similar to RDD actions
- **Basic Functions:** General management
 - e.g. `cache()`, `unpersist()`, `schema()`, `explain()`, `write()`
- **Typed Transformations:** Return `Dataset[T]`
 - Some are familiar from RDD, e.g. filter function taking a lambda
 - `filter(func: (T) ⇒ Boolean): Dataset[T]`
 - Others are less familiar, e.g. filter function taking a `Column`
 - `filter(condition: Column): Dataset[T]`
- Untyped Transformations: Return `DataFrame` (mostly)
 - `select()`, `groupBy()`, `join()`, `agg()` (aggregate)

MINI-LAB — Review Documentation

Tasks to Perform

- Go to the Scala docs at <http://spark.apache.org/docs/latest/>
 - Find package `org.apache.spark.sql` as before
 - Note how DataFrame is defined as Dataset[Row]
- Briefly review `class Dataset`
 - Note how the listing is divided into actions, basic functions, typed transformations, and untyped transformations
 - Review each section briefly
 - We're going to work with representative samples of many of them
 - A lot here — don't try and understand it all, but get a high level view
- Review `SparkSession.implicits`
 - In the SparkSession API page, find its `implicits` object, and click it
 - It's where some of the "Scala magic" is defined

Lab 4.2: Spark SQL Basics

In this lab, we load data, and create basic DataFrames

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus, showing what appears to be a server rack.

Overview

SparkSession and Data Load/Store

Introducing DataFrame/Dataset



The Query DSL

**Datasets (The Typed API) flatMap,
explode, split**

DSL Overview

- Popular and powerful way to express transformations
- The DSL consists of:
 - The `Column` class: Represents a column (generally computed)
 - DataFrame methods (e.g. `agg`, `groupBy`, `select`)
 - And some typed (Dataset) methods written in terms of `Column`
 - Auxiliary functions, e.g. `spark.implicits._`, `spark.functions`
 - Below, the DSL expression is: `$"age">25`
`folksDF.filter($"age">25)`
- Can also express transformations via `SQL Expressions`
 - Supports standard SQL, and we'll cover it later
 - As well as all-SQL queries
 - Below, the SQL expression is: `"age>25"`
`folksDF.filter("age>25")`

Introducing class Column

- **Column**: Core DSL class representing a DataFrame column
 - Generally computed from data in the dataframe ⁽¹⁾
 - package: **org.apache.spark.sql**
- Many DataFrame operations take Column parameters
 - e.g. filter() definition is:
 - `filter(condition: Column): Dataset[T]`
 - `DataFrame.col()` provides access to a specific column
 - **col(colName: String):Column**
 - Column defines many expression operators
 - e.g. < (LT), > (GT), **==** (equality), **!=** (inequality), etc. ⁽²⁾
 - Expressions are then passed to DataFrame transformations
 - Let's look at an example

Example: Using the DSL

- Access a dataframe instance's column via the `col()` function
 - `col(colName: String):Column`
 - Use `Column` operators to create expressions
 - Below, we create a `Column` instance specifying "all rows with the age column holding a value > 25"

```
folksDF.filter(folksDF.col("age") > 25)
```

- Scala magic provides shorthand equivalents for the above ⁽¹⁾

```
folksDF.filter(folksDF("age") > 25)
```

```
folksDF.filter($"age" > 25)
```

```
folksDF.filter('age > 25)
```

Examining a Transformation

```
> folksDF.filter('age>25).groupBy('gender).avg("age").show
+-----+
|gender|avg(age)|
+-----+-----+
|      F|    46.0|
|      M|    35.0|
+-----+
```

- **filter('age>25)** gets rows with age>25 (we know this one)
 - 'age is a Scala symbol — converted to a Column by Scala magic ⁽¹⁾
 - filter() returns a DataFrame
- **groupBy('gender)**: Groups by gender column values (more detail later)
 - groupBy() is used for aggregation, and has this signature
 - groupBy(cols: Column*) : RelationalGroupedDataset
 - RelationalGroupedDataset defines aggregation methods
- **avg("age")** gets the average age for the group
 - Defined in RelationalGroupedDataset with signature
 - **avg(colNames: String*) : DataFrame**

Working with the Query DSL

- Let's examine several examples to:
 - Gain an idea of how the DSL works and its capabilities
 - Understand where the pieces are
 - Scattered in different places — can be confusing
- We'll view both typed and untyped transformations
 - The common thread is they use the DSL (e.g. Column instances)

Example: Dataset.orderBy()

- Typed transformation — two versions
 - `orderBy(sortExprs: Column*): Dataset[T]`
 - `orderBy(sortCol: String, sortCols: String*): Dataset[T]`
 - Both are aliases for `sort()`
 - Relatively straightforward

```
// Sort on age column ascending – pass in name of column (a string)

> folksDF.orderBy("age").show // equivalent to folksDF.sort("age")
+---+---+---+
| age|gender|name|
+---+---+---+
| 20|M|Mike|
| 35|M|John|    // Remaining detail omitted

// Sort on age column descending pass in column instance
// 'age' is a Scala symbol – converted to a Column via Scala magic
// desc is a method on class Column

> folksDF.orderBy('age.desc).show // Results in descending order
```

Example: RelationalGroupedDataset.agg()

- `groupBy` returns a `RelationalGroupedDataset`
 - After grouping, use `RelationalGroupedDataset` methods
 - We demonstrate its `agg()` method here
- `agg(exprs: Map[String, String]): DataFrame`
 - Specify map from column name to aggregate methods
 - Available aggregate methods: `avg`, `max`, `min`, `sum`, `count`

```
// Use RelationalGroupedDataset.agg() method taking pairs made of
// column name/aggregation function name
> folksDF.filter('age > 25).groupBy('gender).agg(
    | "age" -> "avg",
    | "age" -> "max",
    | "gender" -> "count").show
+-----+-----+-----+
| gender|avg(age)|max(age)|count(gender)|
+-----+-----+-----+
|      F|     46.0|      52|          2|
|      M|     35.0|      35|          1|
+-----+-----+-----+
```

Example: Dataset.agg()

- `agg(expr: Column, exprs: Column*): DataFrame`
 - Aggregate on entire dataframe without groups
 - Below, the `min`, `max`, and `avg` functions are from the object `org.apache.spark.sql.functions`
 - The functions are available for all DataFrame operations

```
> folksDF.agg(max('age')).show
+-----+
| max(age) |
+-----+
|      52 |
+-----+

> folksDF.agg(min('age'), max('age'), avg('age')).show
+-----+-----+-----+
| min(age) | max(age) | avg(age) |
+-----+-----+-----+
|      20 |       52 |    36.75 |
+-----+-----+-----+
```

Example: Column Operators

```
// && – Logical "and" (age between 25, 50)
> folksDF.filter('age>25 && 'age<50).show

> folksDF.filter('age === 35) // Age is 35 – Equality is 3 equal signs
> folksDF.filter('age.equalTo(35)) // Equality using equalTo

> folksDF.filter('age != 35) // Age not 35 – Inequality op is !=

> folksDF.agg(avg('age).alias("avgAge")).show // Column alias
+-----+
| avgAge|
+-----+
| 36.75|
+-----+

> folksDF.filter('name.startsWith("J")) // Filter: names starting with J
> folksDF.filter('name.like("%e")) // Filter: names ending with e
```

Example: Select

- Below, we use Column selectors to create a projection
 - Result is a dataframe containing two columns (gender, age)

```
> val genderAgeDF = folksDF.select('gender, 'age)
genderAgeDF: org.apache.spark.sql.DataFrame =
                           [gender: string, age: bigint]
```

```
> genderAgeDF.show
+----+---+
|gender|age|
+----+---+
|     M| 35|
|     F| 40|
|     M| 20|
|     F| 52|
+----+---+
```

Example: Literals

- Literals support operations on columns using simple values
- `Lit()` wraps a literal value in a `Column` instance
 - It can then be used easily in the DSL

```
> folksDF.select('age*lit(2), 'name, 'gender).show
+-----+----+----+
| (age * 2) | name | gender |
+-----+----+----+
|      70 | John |      M |
|      80 | Jane |      F |
|      40 | Mike |      M |
|     104 | Sue  |      F |
+-----+----+----+
```

More Complex Data

- Suppose your people data has nested address info also — e.g.

```
{"name": "John", "gender": "M", "age": 35,  
 "address": {"street": "123 Aspen St.", "city": "Buffalo",  
 "state": "NY", "zip": "14205"} }
```

- Access nested elements using . (dot) notation, as shown below

```
> val folksAddressDF = spark.read.json("data/people-with-address.json")  
> folksAddressDF.select($"address.city").show  
+-----+  
|      city|  
+-----+  
| Buffalo| // Remaining data omitted.  
  
> folksAddressDF.select('address.city).show // This will NOT work (1)  
  
// Access the complete column. Use show(false) to prevent truncation  
> folksAddressDF.select($"address").show(false)  
+-----+  
| address |  
+-----+  
|[Buffalo,NY,123 Aspen St.,14205] | // Remaining data omitted.
```

UDF: User Defined Function

- UDFs define new Column-based functions (for both DSL and SQL)
- To define and use a UDF:
 - Define a regular function
 - Use `udf()` to wrap it ⁽¹⁾ (`org.apache.spark.sql.functions`)
 - Use it with the DSL — as shown below

```
// Define regular function to test age (some detail omitted ...)
> val young : (Int) => Boolean = (age) => (age<45)
> import org.apache.spark.sql.functions.udf
> val youngUDF = udf(young) // Wrap as a UDF
youngUDF: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function1>,BooleanType,Some(List(IntegerType)))
// Use it
> folksDF.filter(youngUDF('age')).show
+---+----+----+
| age|gender|name|
+---+----+----+
|  35|M|John|
|  40|F|Jane|
|  20|M|Mike|
+---+----+----+
```

More Complex UDF

- Below, we illustrate a UDF that takes two arguments
 - Same principals with more complex syntax

```
// Define regular function to test age, gender
> val youngMale : (Int, String) => Boolean = (age, gender) =>
   (age<45 && gender=="M")
youngMale: (Int, String) => Boolean = <function2>

// Wrap it as a UDF
> import org.apache.spark.sql.functions.udf
> val youngMaleUDF = udf(youngMale)
youngMaleUDF: org.apache.spark.sql.expressions.UserDefinedFunction =
UserDefinedFunction(<function2>,BooleanType,Some(List(IntegerType,
StringType)))

// Use it
> folksDF.filter(youngMaleUDF('age, 'gender)).show
+---+-----+---+
| age|gender|name|
+---+-----+---+
|  35|M|John|
|  20|M|Mike|
+---+-----+---+
```

Introducing Row

- **Row**: Represents one row of output in a DataFrame
 - Remember, DataFrame == DataSet[**Row**]
 - Often don't deal directly with Row ⁽¹⁾
 - package: **org.apache.spark.sql**
- Accessing Row values
 - **apply(i: Int)**: Any — Generic access by ordinal
 - **getXXX(i: Int): XXX** — Native primitive access
 - getInt(i: Int): Int, getString(i: Int): String, etc.
 - Illustrated below ⁽²⁾

```
> folksDF.map(r => (r.getLong(0), r.getString(1), r.getString(2))).show
+---+---+---+
| _1| _2| _3|
+---+---+---+
| 35| M|John| // Remaining data omitted.

> folksDF.rdd.map(r => (r(0), r(1), r(2))).collect
res14: Array[(Any, Any, Any)] = Array((35,M,John), ,,,)
```

Extracting Row Data into a Class

- It's usually easier to move the data into a class
 - Often a case class
 - We show two examples of this below

```
scala> case class Person (age: Long, gender: String, name: String)
defined class Person

scala> folksDF.map( r => Person(r.getLong(0), r.getString(1),
r.getString(2))).show
+---+-----+----+
|age|gender|name|
+---+-----+----+
| 35|M|John|
| 40|F|Jane| // Remaining data omitted.

// Alternative using pattern match – same result as above.
> folksDF.map{
  case Row (age:Long, gender:String, name:String) => Person(age, gender,
name)
}.show
```

Why Use the DSL

- Fluent interface easy to use, easy to understand
 - And generally performs better (more on Catalyst optimizer later)
- Consider the example at bottom
 - A simple RDD version might look like the following

```
folksRDD.filter(p=>p.age>25)
          .map(p=>(p.gender, p.age)).collect
```

 - This produces pairs (tuples) of (gender, age)
 - Which is easier to understand?

```
folksDF.filter('age>25).select('gender, 'age).show
+---+---+
|gender|age|
+---+---+
|      M| 35|
|      F| 40|
|      F| 52|
+---+---+
```

Using SQL

- We've seen the example at bottom earlier
- **createOrReplaceTempView()**: Registers a table name
 - i.e. So it is known to the SQL parser
 - Session-scoped, and dropped when session terminates
- **SparkSession.sql()**: Executes a SQL query using Spark
 - Supports SQL 2003 (can configure SQL dialect)
 - Signature: `sql(sqlText: String): DataFrame`

```
folksDF.createOrReplaceTempView("people")      // Setup for using SQL

// Get the average by gender
spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
+-----+-----+
|gender|avg(age)|
+-----+-----+
|      F|     46.0|
|      M|     27.5|
+-----+-----+
```

SQL vs DataFrame DSL Queries

- They execute the same
 - Once parsed, both are processed the same
 - Catalyst analyzes and optimizes the query either way
 - Performance is the same
- Choose the more natural approach
 - If you like SQL, that may be easier
 - If you like code, the DSL may be easier
 - Some queries are easier to express one way or the other
 - So use the one that works best
- We'll look at (typed) Datasets soon
 - These are anything other than Dataset[Row]/DataFrame

MINI-LAB — Review Documentation

Tasks to Perform

- Go to the Scala docs at <http://spark.apache.org/docs/latest/>
 - Find package `org.apache.spark.sql`s before
- Review `class Dataset`, in particular do the following:
 - Review the methods we just used, and where class `Column` is used
 - Note how there are multiple versions of many methods (for flexibility)
- Briefly review `class Column`
 - Note the selection of operators
- Briefly review `class RelationalGroupedDataset`
 - Note the selection of operators
- Briefly review `org.apache.spark.sql.functions`
 - See notes for help in finding this

Lab 4.3: DataFrame Transformations

In this lab, we will work with basic DataFrame transformations

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus, showing what appears to be a server rack.

Overview

SparkSession and Data Load/Store

Introducing DataFrame/Dataset

The Query DSL



Datasets (The Typed API)

flatMap, explode, split

Creating Datasets[T]

- Often created from a DataFrame using `as[T]` (1)
`as[U](implicit arg0: Encoder[U]): Dataset[U]`
 - Since SparkSession generally loads data into DataFrames

```
// Assume folksDF created as previously

case class Person (name: String, gender: String, age: Long)

val folksDS=folksDF.as[Person]
folksDS: org.apache.spark.sql.Dataset[Person] = [age: bigint, gender: string
... 1 more field]

folksDS.show
+---+-----+----+ // Detail omitted ...
| age|gender|name|
+---+-----+----+
| 35|M|John|
| 40|F|Jane|
```

Dataset API: Typed Transformations

- Has numerous transformations
 - Several take lambdas — similar to RDD transformations
 - We illustrate some below (much detail omitted)
- `filter(func: (T) ⇒ Boolean): Dataset[T]`:
 - Filter based on func
- `flatMap[U](func: (T) ⇒ TraversableOnce[U]): Dataset[U]`
 - Similar to RDD version
- `groupByKey[K](func: (T) ⇒ K): KeyValueGroupedDataset[K, T]`
 - Group by key (of type K) returned by func
- `select[U1](c1: TypedColumn[T, U1]): Dataset[U1]`
 - Return new Dataset composed of selected column
 - Many variants for selecting more columns
- There are others — we'll look at the docs and examples

Transformations with Datasets

- Some transformations return values of the same type
 - e.g. `filter()` which has several variations, including:
`filter(func: (T) ⇒ Boolean): Dataset[T]`
`filter(conditionExpr: String): Dataset[T]`
`filter(condition: Column): Dataset[T]`
 - We illustrate all three variations below

```
// Assume folksDS created as previously
// The filters below produce the same results
// They express the condition differently

folksDS.filter(p => p.age > 25)      // lambda condition
folksDS.filter("age > 25").show        // SQL condition
folksDS.filter($"age" > 25)            // DSL condition
folksDS.filter('age > 25')             // DSL condition (w/Scala symbol)
```

Example: Type Changing Transformation

- Some operations return different types from the starting one
 - At bottom, we use `groupByKey()` to group the values by gender
`groupByKey[K](func: (T) => K)(implicit arg0: Encoder[K]): KeyValueGroupedDataset[K, T]`
 - Data is grouped by the given function (which converts a T to a K)

```
// Assume folksDS created as previously

> val folksDSGrouped = folksDS.groupByKey(p => p.gender)
folksDSGrouped: org.apache.spark.sql.KeyValueGroupedDataset[String,Person] =
...

> folksDS.groupByKey(T => T.gender).count.show // Use the grouping
+---+-----+
|value|count(1)|
+---+-----+
|    F|      2|
|    M|      2|
+---+-----+
```

Example: Average Age

- At bottom, we calculate average age by gender again ⁽¹⁾

- Using `KeyValueGroupedDataset.agg`, which looks like this:

`agg[U1](col1: TypedColumn[V, U1]): Dataset[(K, U1)]`

- Note the `TypedColumn` argument

- The strong typing carries through

- We create the `TypedColumn` via `Column.as[Double]`

`as[U](...): TypedColumn[Any, U]`

```
folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double])
res5: org.apache.spark.sql.Dataset[(String, Double)] = [value: string,
avg(age): double]
```

```
folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double]).show
+-----+
| value|avg(age) |
+-----+
|    F|    46.0|
|    M|    27.5|
+-----+
```

Example: Select

- The first example below selects (type safe) columns
 - Notice the `as[]` on each column selector
 - It converts them to a `TypedColumn`
 - This causes the typed version of select to be called
 - The result is a `Dataset` of `(String, Long)` pairs
- The second example below, uses normal `Column` selectors
 - This results in the untyped `select` being called
 - Result is a `dataframe`

```
folksDS.select('gender.as[String], 'age.as[Long])
res6: org.apache.spark.sql.Dataset[(String, Long)] = [gender: string, age: bigint]

folksDS.select('gender, 'age)
res130: org.apache.spark.sql.DataFrame = [gender: string, age: bigint]
```

From Datasets to DataFrames

- Easy to go from a Dataset to a DataFrame
 - Just call `toDF()`, or call any of the untyped methods
 - The untyped methods return DataFrames
 - We illustrate this below (`select` showed this also)

```
// groupBy('gender) is an untyped (DataFrame) operation
// By calling it, we move from the Dataset to the DataFrame world

val results = folksDS.filter('age > 25).groupBy('gender).avg("age")
results: org.apache.spark.sql.DataFrame = [gender: string, avg(age): double]

results.show
+-----+-----+
|gender|avg(age)|
+-----+-----+
|      F|    46.0|
|      M|    35.0|
+-----+-----+
```

Confusing?

- Yes, the API is complex and can be confusing
 - Multiple ways to specify transformations
 - DSL, SQL, lambdas
 - **DataFrames** and **Datasets**
 - With a fairly fluid boundary between them
 - What should we use?
- To help us judge, we'll compare the following areas
 - **Errors**: When are errors caught
 - **Ease of use**: How easy is the API to use
 - **Maturity and Stability**: How full and stable is the API
 - **Performance**: Tradeoffs that affect performance

Errors

- The APIs catch errors at **different times**
 - The table below illustrates when different error types are caught
 - **Syntax Error**: e.g. misspelling: "**SELECT**" or **df.filler()**
 - **Analysis Error**: e.g. unknown property, e.g.
 - SQL: "**GROUP BY gnder**"
 - DataFrame: **df.groupBy("gnder")**
 - Dataset: **ds.groupByKey(p=>p.gndr)**
- Datasets catch more errors at **compile time**
 - A benefit

Error Type	SQL	DataFrame	Dataset
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Ease of Use

- Below, we illustrate a transformation three ways
 - The Dataset version is arguably the most complex
 - Both the functions and the API docs describing them
 - Can be even more true in more complex transformations
- Dataset type safety **adds complexity** when coding

```
// DataFrame  
folksDF.filter('age>25).groupBy('gender).avg("age")  
  
// Dataset  
folksDS.groupByKey(T => T.gender).agg(avg('age).as[Double])  
  
// SQL  
spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
```

Maturity and Stability

- DataFrames and SQL have been around for a while
 - 1.0/1.3
- Datasets introduced in 1.6, refined in 2.0
 - Much of the typed (Dataset) API are marked **experimental**
- It seems likely that Datasets will continue to evolve
 - If you use it, you'll need to evolve your code base with it
 - You're also more likely to run into areas that don't support your needs

Performance

- **DataFrames** and SQL basically perform the same
 - They are optimized and transformed by the runtime to essentially the same set of actions
- **Datasets** rely on lambdas (for a good deal of the typed API)
 - Use of lambdas can **affect performance**
 - The Catalyst optimizer can't understand them
 - Can't optimize for their affects
 - Can have a large negative performance impact
 - We will cover this more later

Summary

- You have a lot of choices
- The query DSL, Dataset typed API, and SQL queries are arguably all better than RDDs
 - Unless you need to do something only supported by RDDs
- The query DSL and SQL are mostly equivalent
 - Use whichever you're most comfortable with
- The Dataset typed API adds compile-time type safety
 - Catches the most errors at compile time
 - But has other shortcomings
 - Decide if the benefit is worth it to you ⁽¹⁾

Lab 4.4: The Dataset Typed API

In this lab, we will work with the Dataset typed API

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or data. The background is slightly out of focus, showing what appears to be a server rack or a stack of books.

Overview

SparkSession and Data Load/Store

Introducing DataFrame/Dataset

The Query DSL

Datasets (The Typed API)



flatMap, explode, split

Splitting Data Up — flatMap()

- **Splitting** input rows into smaller pieces is a common need
 - e.g. Word splits lines into words to start processing
- Many ways to split data, with similar end results
 - We'll review some common techniques
- **flatMap()** is a common choice
flatMap[U](func: (T) ⇒ TraversableOnce[U]): Dataset[U]
 - func generates a collection from each element
 - The collections are combined into the result
 - Part of the Dataset **typed** API

DataFrame flatMap() Example

- Below, we split lines using flatMap
 - flatMap takes a **lambda** function
 - Each element is a Row object
 - We access it via Row.getString()
 - The notes have sample output for each step

```
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF("line")

// Split line via regular expression specifying whitespace
// Each line produces collection of words, which are combined
> val flatMappedWords =
  linesDF.flatMap(_.getString(0).toLowerCase().split("\\s+"))

// flatMap above uses shorthand for this identical lambda
// flatMap(line => line.getString(0).toLowerCase().split("\\s+"))
```

DataFrame explode/split example

- Below, we split the lines using the DSL
 - `split()` splits a column based on a regular expression
`split(str: Column, pattern: String): Column`
 - `explode()` creates a Row for each element in an Array/Map
`explode(e: Column): Column`
 - We lowercase separately, due to technical reasons
 - It's not legal to nest the predicates — e.g. `lower(explode(split))`

```
// Split line via split() and same regular expression
// split produces Rows containing array<string>
// explode produces Rows containing strings
// See notes for details

> val splitWordsDF = linesDF.select(explode(split('line, "\\s+")).as("word"))
  .select(lower('word).as("word"))
```

Dataset/RDD flatMap() Example

- Dataset similar to DataFrame but simpler
 - The Dataset contains simple strings, not Rows
- RDD similar to Dataset (but processing will be lower level)
- explode/split not available for RDD or Dataset typed API

```
// Dataset: Use our dataframe to make it
> val linesDS = linesDF.as[String]
linesDS: org.apache.spark.sql.Dataset[String] = [line: string]

// Split line via flatMap – the elements are strings now
> val splitWordsDS = linesDS.flatMap(_.toLowerCase().split("\\s"))
```

```
// RDD: Create via parallelize
> val linesRDD = sc.parallelize(Seq("Twinkle twinkle ...", "...", "..."))
res12: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[602] at flatMap at
<console>:27

// Split line via flatMap – the elements are strings now
> val splitWordsRDD = linesRDD.flatMap(_.toLowerCase().split("\\s"))
```

Summary

- Simply splitting lines presents many choices
 - DataFrames vs. Datasets vs. RDDs
 - flatMap() vs. explode/split
 - The API is large, with a lot of choices
- Generally, choose Dataset or DataFrame over an RDD
 - Cleaner, and often better performance
- We'll look at the Catalyst and Tungsten enhancements soon
 - Will provide more guidelines for choosing your approach

A photograph of a man and a woman sitting at a desk, looking at a laptop screen together. They are both smiling. The man has a beard and is wearing a light-colored shirt. The woman has her hair pulled back and is wearing a white top. A green diagonal stripe runs from the bottom right corner across the slide.

Lab 4.5: Splitting up Data

Review Questions

- What is Spark SQL, and why do we use it?
- How do you read data using the SparkSession? What kinds of data can be read?
- What is a DataFrame? A Dataset?
- What is the DataFrame query DSL?

Session Summary

- Spark SQL is a module for structured data processing
 - It adds schema to the data we work with
 - It provides a simpler API and more efficient processing via Catalyst and Tungsten
- Use **SparkSession**, **DataFrameReader**, and **DataFrameWriter** to load/store data
 - Many common data types are supported, including
 - JSON, CSV, database, parquet, text
- A DataFrame adds **schema information** to data
 - In some cases (e.g. JSON) Spark can infer the schema
- A Dataset adds **compile-time type safety**

Session Summary

- You can query DataFrames with:
 - The DSL: A Domain Specific Language for querying
 - SQL: SQL 2003 queries
 - Typed transformations

A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt, standing in what appears to be a lecture hall or conference room. He is gesturing with his hands while speaking. In the foreground, the back of another person's head and shoulders are visible, looking towards the speaker. The background shows rows of audience members and overhead projector equipment.

Session 5: Shuffling Transformations and Performance

- Grouping and Reducing
- Shuffling
- Catalyst Optimizer
- Tungsten Optimizer
- Summary

Session Objectives

- Explore transformations that involve shuffling
 - Grouping, reducing, joins, etc.
- Understand performance implications of shuffling
- Understand how Catalyst and Tungsten improve performance
 - Be aware of transformations inhibiting their improvements
- We'll use a **word count** example to drive this session
 - Looking at the various tasks involved, the choices you have, and the implications of each



Grouping and Reducing

Shuffling

Catalyst Optimizer

Tungsten Optimizer

Grouping Overview

- Grouping arranges data into groups
 - Based on values for specified column(s)
 - Generally used for **aggregation** (e.g. count, max, sum, etc.)
 - Our earlier examples grouped into two groups by gender
 - One group for 'M', one for 'F'
- Grouping involves **shuffling** data
 - Redistributing data across partitions
 - Important performance implications — more detail later
- Dataset/DataFrame grouping methods are simple to use
 - **[KeyValue/Relational]GroupedDataset** handles the aggregation
- RDD `groupByKey` is more complex
 - Low level, you deal with the (unwieldy) aggregation yourself

Grouping Methods

- **DataFrame** (Untyped) Versions:
groupBy(cols: Column*): **RelationalGroupedDataset**
 - Group using the specific Columns
 - **groupBy(col1: String, cols: String*)**: **RelationalGroupedDataset**
 - Group by the columns with the given names
- **Dataset** (Typed) Versions:
groupByKey[K](func: (T) ⇒ K): **KeyValueGroupedDataset[K, T]**
 - Group by key (of type K) returned by func
- RDD Version (in `org.apache.spark.rdd.PairRDDFunctions`)
groupByKey(): `RDD[(K, Iterable[V])]`
 - Work on (key, value) pairs
 - Return another RDD

DataFrame Word Counting

- Simple — just group and count
 - The aggregation methods are easy to use
 - In RelationalGroupedDataset
 - Note, that this example **uses the DSL** throughout
 - If splitWordsDF is created with explode/split

```
// Use splitWordsDF seen earlier (contains one word per row)
// Group by words, and then count them
> val countDF = splitWordsDF.groupBy('word).count

> countDF.show
+---+---+
| word|count|
+---+---+
| you| 1|
| how| 1| // Remaining data omitted
```

Dataset Word Counting

- Also simple — `groupByKey` and `count`
 - Also has easy to use aggregation methods
 - In `KeyValueGroupedDataset`
 - Note that this example uses lambdas
 - Here, and in creating `splitWordsDS`

```
// Use splitWordsDS seen earlier (contains one word per row)
// Group by words, and then count them
> val countDS = splitWordsDS.groupByKey(s => s).count

countDS.show
+-----+-----+
| value|count(1)|
+-----+-----+
|   you|      1|
|   how|      1|  // Remaining data omitted
```

RDD Grouping

- **Pair RDD** required for grouping
 - RDD containing key/value pairs of form **(key, value)**
 - e.g. (M, 35) or (twinkle, 1)
 - Many other methods require a pair RDD
 - **PairRDDFunctions** defines extra functions for pair RDDs
- Our Pair RDD for counting occurrences of age is:
 - **key**: The age, **value**: The count (1 in this case)
 - We code the aggregation ourselves
- RDD grouping harder to understand than DataFrame/Dataset
 - The grouping returns a key (the group) paired with an Iterable (the elements in that group) — low level and hard to work with
groupByKey(): `RDD[(K, Iterable[V])]`

RDD Word Count with Grouping

- We create pair RDDs (required to group)
 - Our pairs are of form (**twinkle**, 1)
 - See the notes for more detail on the example below
- Grouping is inefficient for this example
 - Other methods better (e.g. reduceByKey)
 - But this serves as a grouping example

```
// Use splitWordsRDD seen earlier

splitWordsRDD.map(x => (x, 1)) // Create (word, 1) pairs
  .groupByKey()                // Group into word groups
  .map( x => (x._1, x._2.size)) // Size of second value is count
  .collect

res15: Array[(String, Int)] = Array((how,1), (i,1), (star,2), (wonder,1),
  (are,1), (twinkle,4), (what,1), (little,2), (you,1))
```

RDD Word Count with reduceByKey

- Can be done more efficiently using `reduceByKey()`

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

- Merge values for each key using an associative and commutative reduce function
- **Merges locally** on each mapper before sending results to a reducer
- More on efficiency soon

```
// Use splitWordsRDD seen earlier

splitWordsRDD.map(x => (x, 1))      // Create (word, 1) pairs
  .reduceByKey(_+_)
  .collect

res18: Array[(String, Int)] = Array((how,1), (i,1), (star,2), (wonder,1),
(are,1), (twinkle,4), (what,1), (little,2), (you,1))
```

Lab 5.1: Exploring Grouping

In this lab, we will explore grouping and aggregations



Grouping and Reducing

→ Shuffling

Catalyst Optimizer

Tungsten Optimizer

Shuffling Examined

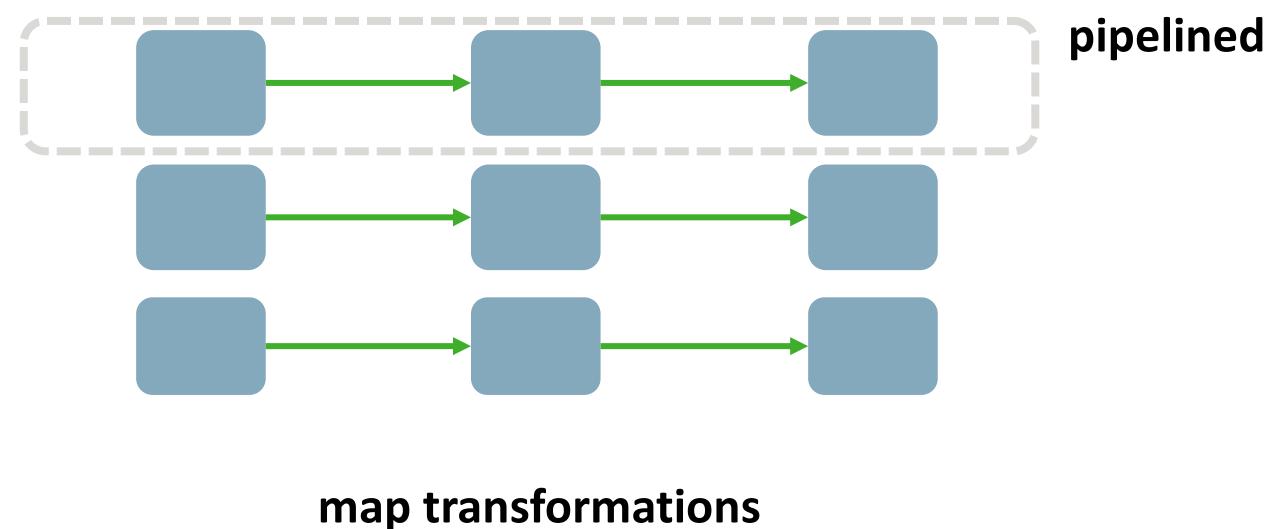
- Consider a word count program (high level)
 - Create records with the words as keys
 - For each word occurrence, emit a record with the word and the value 1
 - **Sum up the values** for each key (word) — giving the count
- **Question:** How do we sum the values?
 - Remember — words are distributed across partitions
- **Answer:** Move all values for a given word to one machine
 - Called **shuffling** the data

Narrow and Wide Dependencies

- Different transformations perform very differently
 - Often because of dependencies between RDDs
 - Two main types of dependencies
- **Narrow Dependency**: Operate on a **single** partition
 - One partition of the child RDD depends on parent RDD partition
 - 1 to 1 (parent to child)
- **Wide Dependency**: Operates on data from multiple partitions
 - Multiple child partitions may depend on a parent partition
 - 1 to many (parent to child)
 - Requires **shuffling**
- Narrow/wide transformations perform very differently
 - Lets look at them

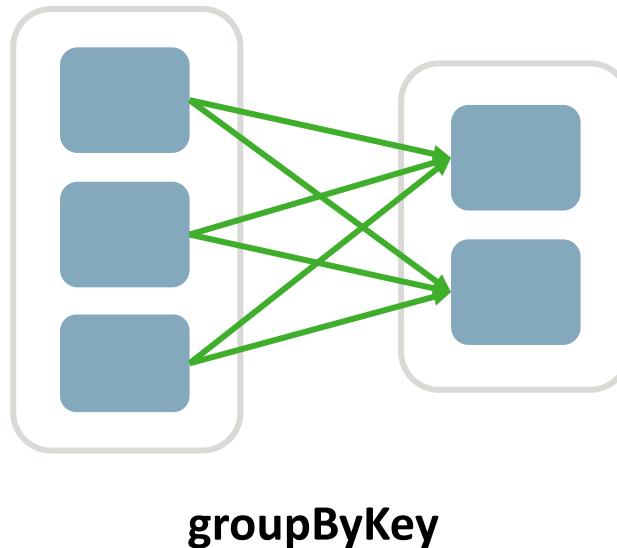
Narrow Dependency Illustrated

- `map()` has a **narrow dependency**
 - Below, each colored box is a partition
 - Each parent partition is used **by only one child** partition
- Spark **pipelines execution** on one cluster node
 - Each input partition transformed to a result partition
 - Very efficient — no data is transferred
 - Efficient fault tolerance — partitions are cheaply recalculated ⁽¹⁾



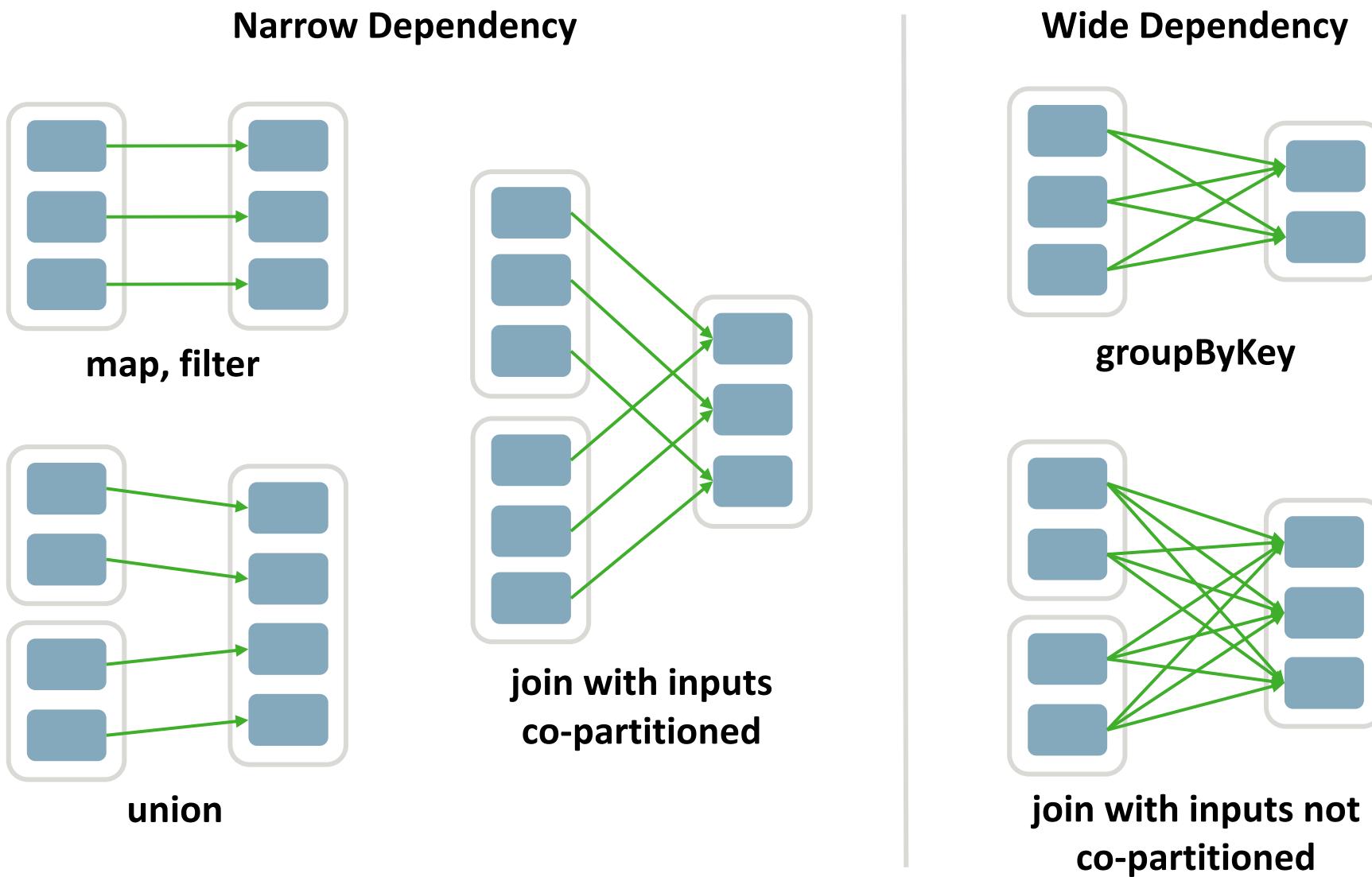
Wide Dependency Illustrated

- `groupByKey()` has a **wide dependency**
 - Below, a parent partition is used by **multiple child partitions**
- Requires data from **all parent partitions** be available
 - Data is **shuffled across the nodes** to accomplish this
 - Relatively expensive, especially for large data
 - Fault tolerance expensive — recovery involves many nodes ⁽¹⁾



Dependencies Illustrated

- Below, we illustrate some common transformations
 - Categorized by dependency type



Shuffling Illustrated

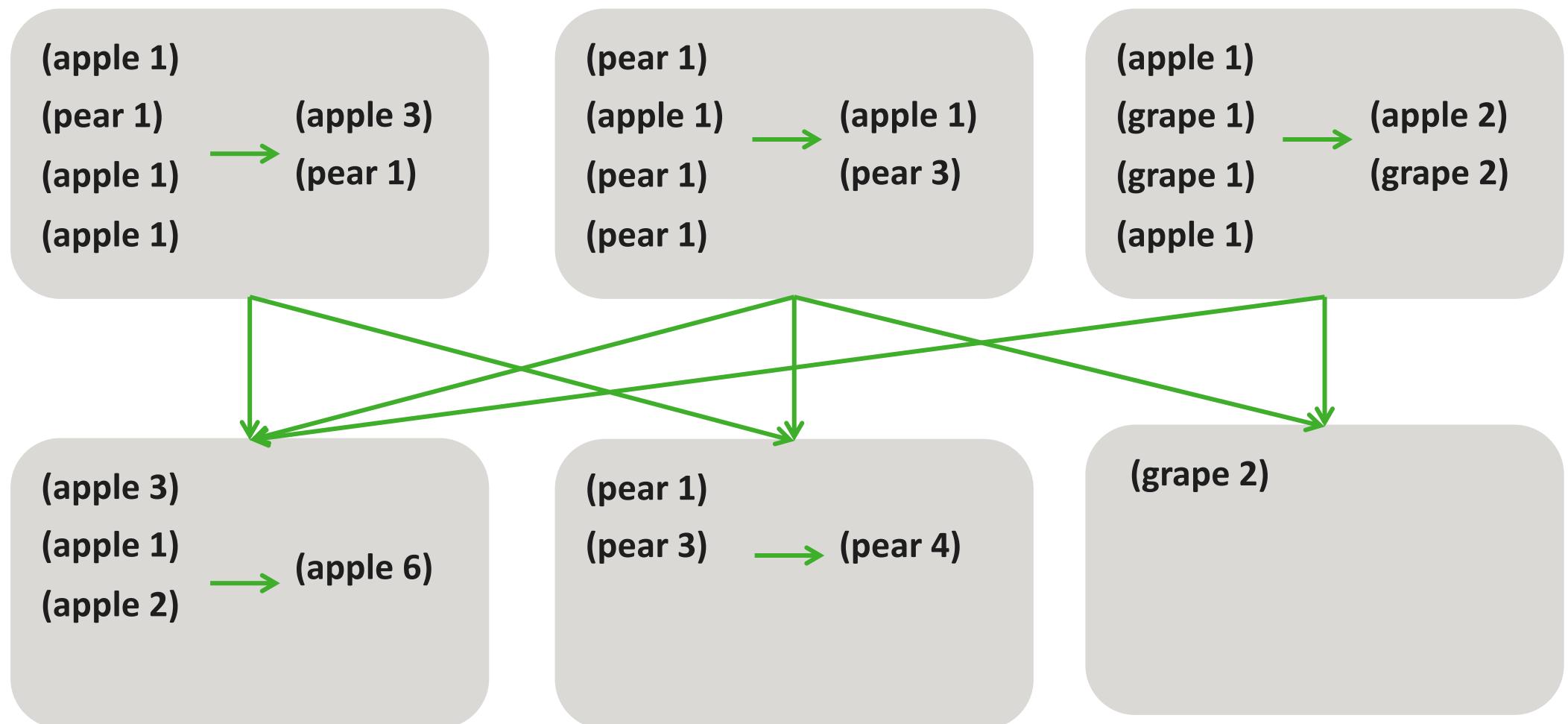
- Shuffling data is **expensive** in multiple ways
 - Initial computation, fault tolerance, etc.
 - It's not our best friend, so minimize it
- Let's review two word count programs regarding shuffling
 - They are illustrated below
 - RDD based

```
val line = "apple pear apple grape pear apple ..."  
val wordPairsRDD = sc.parallelize(line.split("\\s+")).  
                           map(word => (word,1))  
val countsRDD = wordPairsRDD.reduceByKey(_ + _)
```

```
val countsRDD = wordPairsRDD.  
                           .groupByKey()  
                           .map(t => (t._1, t._2.sum))
```

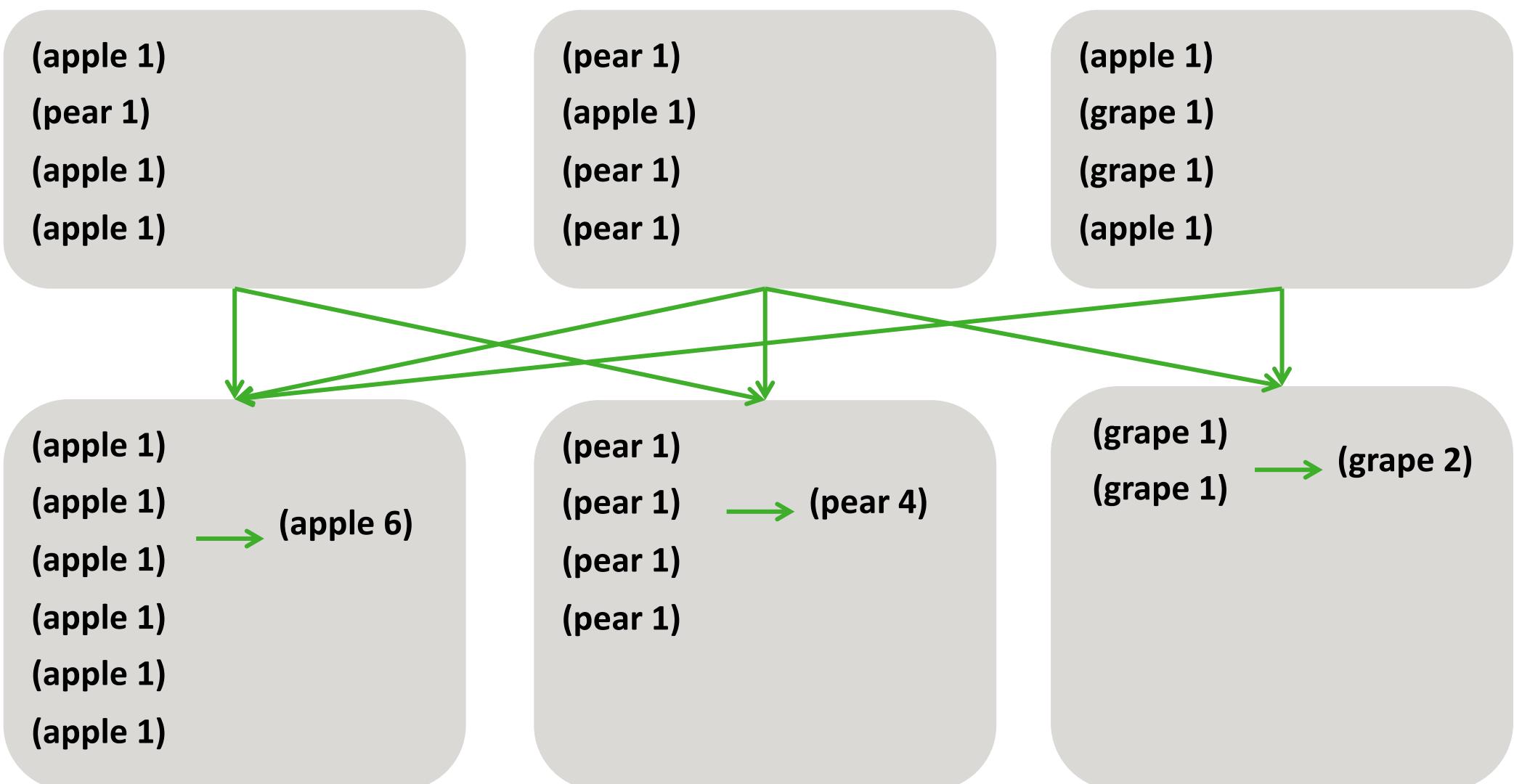
reduceByKey() Examined

- Note how pairs on the same partition are combined first
 - Significantly reduces data size before shuffling
 - Data is then shuffled — less shuffling needed



groupByKey() Examined

- All data is shuffled before reducing
 - More (and unnecessary) data transfer



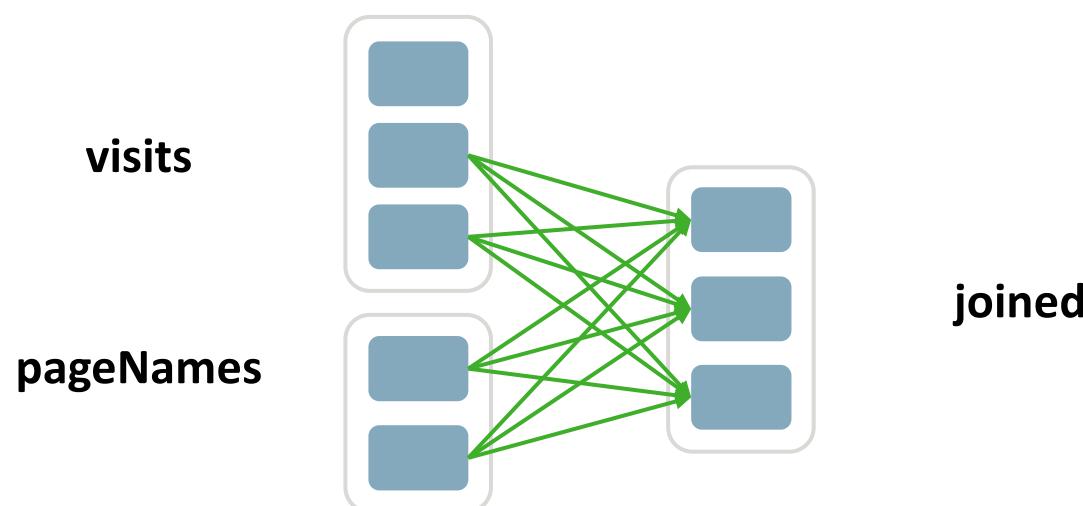
Joins and Shuffling

- A join combines rows from two or more Datasets
 - Based on values of a related column
 - The fields in this column are compared to do the join
- For a distributed join, the data is **shuffled**,
 - So that rows with the same values for the join column(s) are on the same machine
 - The join can then be carried out on each partition, then combined
- The join above can require a **lot of shuffling**
 - Especially if you have a large dataset
 - Shuffling == reduced performance

Shuffling on Join Example

- Consider the example below (1)
 - Spark does a shuffle to join the RDDs (at bottom)

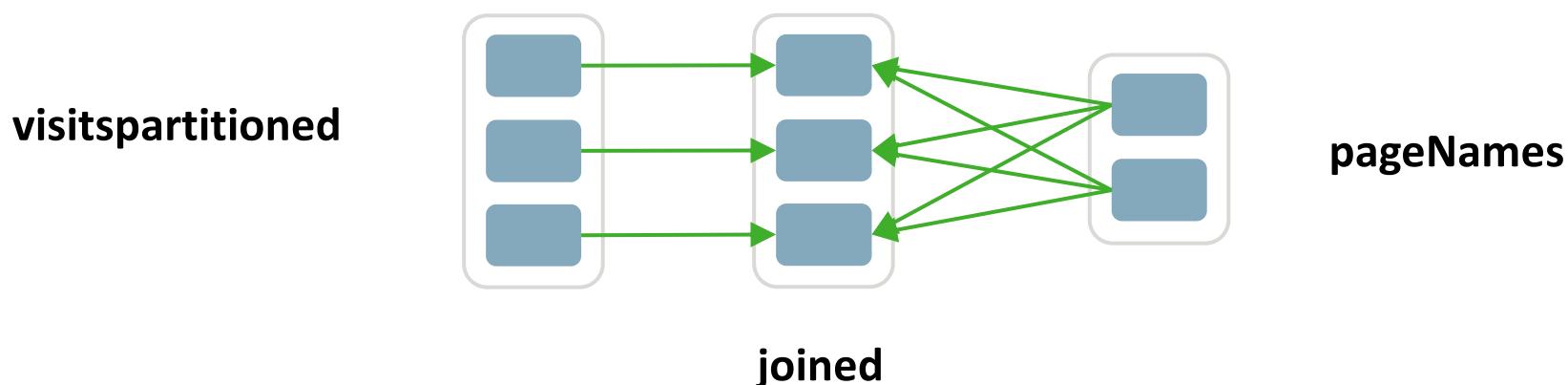
```
// RDD of (URL, visit) e.g. { ("index.html", "1.2.3.4"),  
// ("about.html", "3.4.5.6"), ("index.html", "1.3.3.1") }  
> val visits = sc.textFile("visits.txt").map(...)  
  
// RDD of (URL, name) e.g.  
// { ("index.html", "Home"), ("about.html", "About") }  
> val pageNames = sc.textFile("pages.txt").map(...)  
  
// Join them producing { (about.html, (3.4.5.6, About) ... }  
> val joined = visits.join(pageNames)
```



Pre-Partitioning to Reduce Shuffling

- We want to minimize the data shuffling
 - If you hash-partition visits before joining, it won't need shuffling
 - And generally, visits data is much larger than pages data
 - So this can greatly reduce shuffling
 - We show this below in the code and diagram

```
val visitsPartitioned = sc.textFile("visits.txt").map(...)  
    .partitionBy(new HashPartitioner(2))  
val pageNames = sc.textFile("pages.txt").map(...)  
val joined = visits.join(pageNames)
```



Summary: Do I Need to Think About This?

- We've illustrated some concerns regarding shuffling
 - These are important to understand
 - But very low level — the **HOW** of Spark's computations
 - You're thinking about Spark internals
 - **NOT** what we want to be thinking of for every transformation
 - Tedious, complex, and easy to get wrong
 - We want to focus on the **WHAT**
 - Not the HOW
- The **Catalyst Optimizer** lets you focus on the **WHAT**
 - It will optimize the HOW for you
 - When working appropriately in DataFrames, Datasets, and SQL
 - **NOT** used when you code to RDDs
 - We'll look at it next

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or data. The background is slightly out of focus.

**Grouping and Reducing
Shuffling**

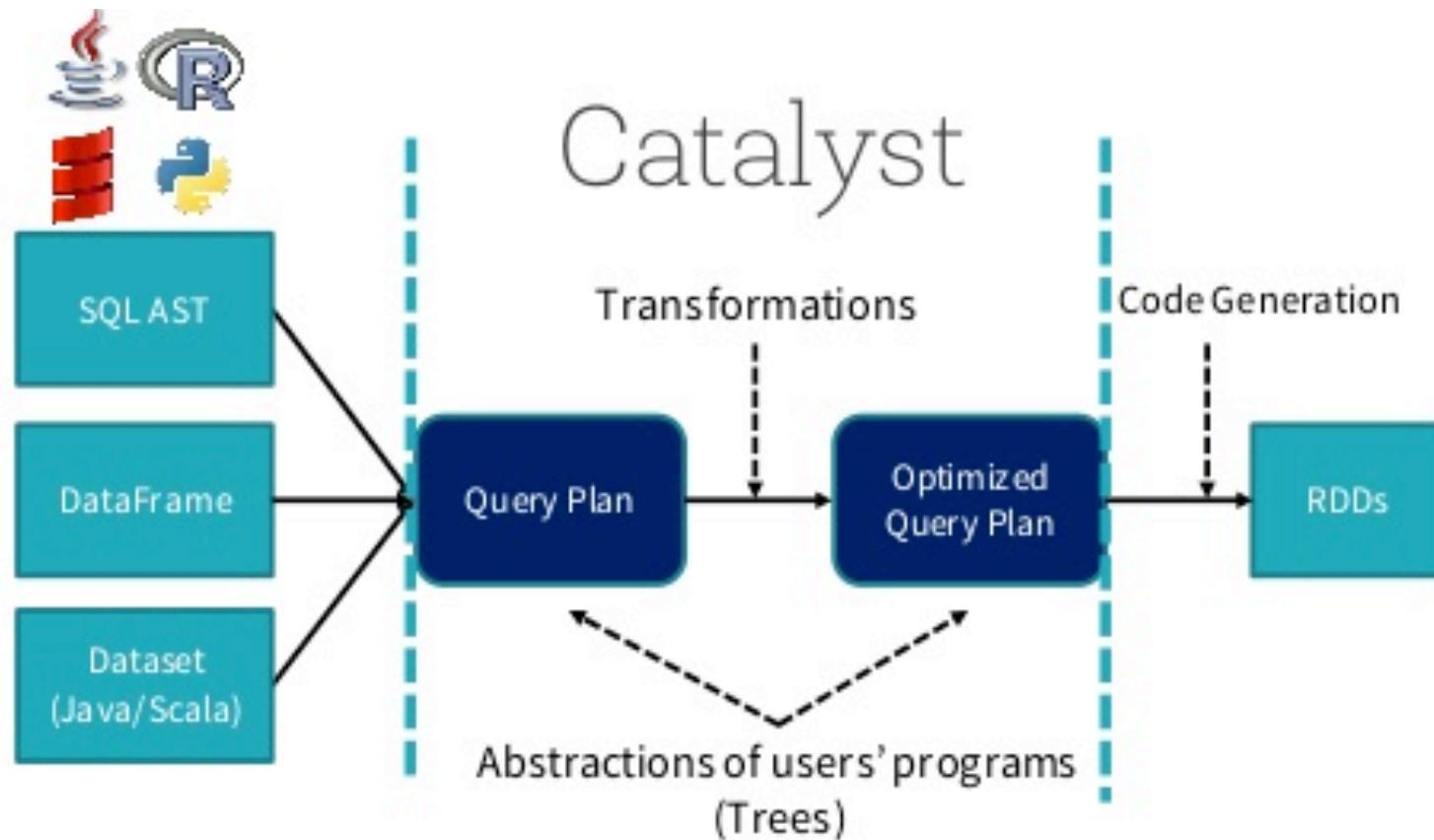


Catalyst Optimizer

Tungsten Optimizer

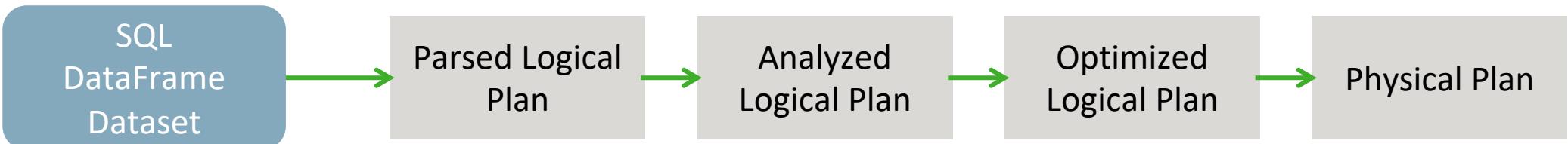
Catalyst Overview

- Catalyst **automatically optimizes** your Spark code
 - When using SQL, DataFrames, or Datasets
 - You focus on the WHAT
 - Catalyst automatically figures out the best HOW



How Catalyst Works

- Optimizes your total transformation
 - Creates an abstract representation (a tree)
 - The [Parsed Logical Plan](#)
 - Analyzes and optimizes the abstract representation
 - Using many optimization rules
 - Creating the [Analyzed Logical Plan](#) and [Optimized Logical Plan](#)
 - Converts the abstract representation to actual transformations
 - Creates multiple physical plan, applies cost model, chooses best one
 - Creating the [Physical Plan](#)



What Optimizations does Catalyst Do?

- Likely, more than you can think of, including:
 - **Predicate Pushdown**: Push filtering as early as possible
 - Eliminate data rows not satisfying preconditions early
 - **Projection Pushdown**: Project as early as possible
 - Eliminating data columns not needed early
 - **Reduce shuffling**: e.g. by reducing before grouping
 - Or joining in the most efficient manner
 - **Constant Folding**: Fold constant calculations into literals
 - **Other Optimizations**: e.g. convert Decimal ops to long ops
 - And more
- Let's look at examples to see it in action
 - Enough for a clear understanding
 - Don't need all the low level details to benefit from it

Example: Word Count with the DSL

- Uses the DSL for all transformations
 - Will need to shuffle data (uses groupBy)
 - Filters out results at two different times
 - Once **before** aggregation (counting) is done, and once **after**

```
// Prepare the data
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF("line")

// Split into words
> val splitWordsDF = linesDF.select(explode(split('line, "\\s+")).as("word")
).select(lower('word).as("word"))

// Filter then count
> val filterThenCountDF = splitWordsDF // Filter before count
  .filter('word != "twinkle").groupBy('word).count (1)

// Count then filter
> val countThenFilterDF = splitWordsDF.groupBy('word)
  .count.filter('word != "twinkle") // Filter after count
```

"Explaining" Catalyst Optimizations

- Call `explain()` on a DataFrame/Dataset to view Catalyst's plans
 - `explain(): Unit` — Display the Physical Plan
 - `explain(extended: Boolean): Unit` — Display the Logical and Physical plans
 - We illustrate below
- We'll look at filtering `before` counting, and `after` counting
 - Both will result in the `same plan` due to Catalyst

```
filterThenCountDF.explain
== Physical Plan ==
*HashAggregate(keys=[word#1168], functions=[count(1)])
+- Exchange hashpartitioning(word#1168, 200)
  +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
// Remaining detail omitted
```

Overview of Explain Output

- In the output at bottom
 - 1-3 prepares the data
 - 4 explodes/splits the data
 - 5 filters the data (**note where this is** — more on this soon)
 - 7 does a partial_count (this is **important** — more on this soon)
 - 8 does the shuffling (called an exchange)
 - 9 does the final counting (an aggregation on word which counts ⁽¹⁾)

```
countThenFilterDF.explain // Filter after counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7 +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6 +- *Project [lower(word#1165) AS word#1168]
5 +- *Filter NOT (lower(word#1165) = twinkle)
4 +- Generate explode(split(line#852, \s+)), false, false,
   [word#1165]
3 +- *Project [value#850 AS line#852]
2 +- *SerializeFromObject [staticinvoke(...) AS value#850]
1 +- Scan ExternalRDDScan[obj#849]
```

Optimization: Predicate Pushdown

- The plan below is for filtering before counting
 - The previous slide showed filtering after counting
- Both transformations **have exactly the same plan**
 - Even though our transformations are written differently
 - Catalyst **pushes the filter down** as far as it can

```
filterThenCountDF.explain // Filter before counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7 +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6   +- *Project [lower(word#1165) AS word#1168]
5     +- *Filter NOT (lower(word#1165) = twinkie)
4       +- Generate explode(split(line#852, \s+)), false, false,
          [word#1165]
3         +- *Project [value#850 AS line#852]
2           +- *SerializeFromObject [staticinvoke(...) AS value#850]
1             +- Scan ExternalRDDScan[obj#849]
```

Optimization: Reduce Shuffling

- The plan **minimizes shuffling**
 - It does the count in two stages — at 7 (locally) and 9 (after shuffle)
 - This is possible because counting is commutative
 - Catalyst has chosen the equivalent of our RDD reduceByKey⁽¹⁾
 - And not the more expensive groupByKey

```
filterThenCountDF.explain // Filter before counting
== Physical Plan ==
9 *HashAggregate(keys=[word#1168], functions=[count(1)])
8 +- Exchange hashpartitioning(word#1168, 200)
7 +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
6   +- *Project [lower(word#1165) AS word#1168]
5     +- *Filter NOT (lower(word#1165) = twinkle)
4       +- Generate explode(split(line#852, \s+)), false, false,
          [word#1165]
3         +- *Project [value#850 AS line#852]
2           +- *SerializeFromObject [staticinvoke(...) AS value#850]
1             +- Scan ExternalRDDScan[obj#849]
```

More Plan Detail

- Below, we show how the filter is pushed down
 - From the Parsed to the Optimized Logical Plan
 - It then generates physical plans, and picks the best one

```
countThenFilterDF.explain(true)
== Parsed Logical Plan ==
'Filter NOT ('word = twinkle)
+- Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
  +- Project [lower(word#1165) AS word#1168]
    // ...

== Analyzed Logical Plan ==
word: string, count: bigint
Filter NOT (word#1168 = twinkle)
// ...

== Optimized Logical Plan ==
Aggregate [word#1168], [word#1168, count(1) AS count#1292L]
+- Project [lower(word#1165) AS word#1168]
  +- Filter NOT (lower(word#1165) = twinkle)
    // ...
```

Lambdas Impede Catalyst Optimization

- Below, we filter with a lambda, after counting
 - Where does filtering happens in the physical plan?
 - After the aggregation
 - Catalyst can't optimize with a lambda (or UDF) — it's opaque ⁽¹⁾
 - Result — less efficient transformation

```
// Count then filter with lambda. splitWordsDF uses split/explode
> val countThenFilterDF = splitWordsDF.groupBy('word)
   .count.filter(w => w.getString(0) != "twinkle").explain
== Physical Plan ==
*Filter <function1>.apply
8 *HashAggregate(keys=[word#1168], functions=[count(1)])
7 +- Exchange hashpartitioning(word#1168, 200)
6 +- *HashAggregate(keys=[word#1168], functions=[partial_count(1)])
5 +- *Project [lower(word#1165) AS word#1168]
4   +- Generate explode(split(line#852, \s+)), false, false,
     [word#1165]
3     +- *Project [value#850 AS line#852]
2       +- *SerializeFromObject [staticinvoke(...) AS value#850]
1         +- Scan ExternalRDDScan[obj#849]
```

But Not All Lambdas

- Below, we groupByKey with a lambda, then count, then filter
 - Our transformation code filters **after** grouping/aggregating
 - In the physical plan, filtering happens **before** the aggregation — even though the grouping uses a lambda
 - Catalyst understands the aggregation (count) and still optimizes
 - However, if the filter uses a lambda, it still won't push it down ⁽¹⁾

```
val linesDS = linesDF.as[String] // Generate a Dataset
linesDS.flatMap(_.toLowerCase().split("\\s+")). // Split into words
    .groupByKey(s => s).count // Group and count
    .filter('value != "twinkle").explain // Filter after

== Physical Plan ==
*HashAggregate(keys=[value#1420], functions=[count(1)])
+- Exchange hashpartitioning(value#1420, 200)
   +- *HashAggregate(keys=[value#1420], functions=[partial_count(1)])
      +- *Project [value#1420]
         +- *Filter (isNotNull(value#1420) && NOT (value#1420 = twinkle))
```

Summary

- When using the DSL, you don't have to think (much) about efficiency or the **HOW**
 - Catalyst will think about efficiency for you
 - In general, it will generate a highly optimized execution plan
- **Lambdas** impede Catalyst
 - But not always
 - As we saw with our groupByKey example
 - You need to know something about how Spark/Catalyst works
 - `explain` is your friend if in doubt
- Let's look at Tungsten next

Lab 5.2: Seeing Catalyst at Work

In this lab, we'll examine several transformations and observe how Catalyst optimizes them

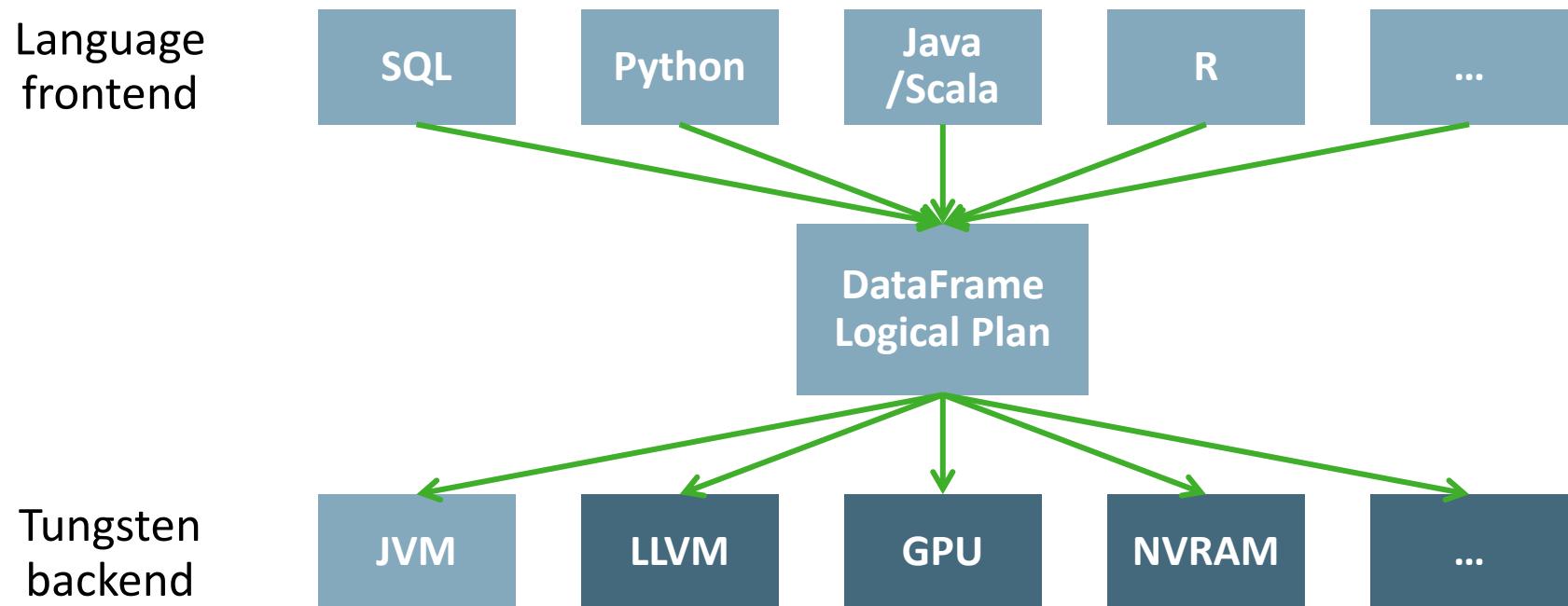
A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or data. The background is slightly out of focus, showing what appears to be a server rack.

**Grouping and Reducing
Shuffling
Catalyst Optimizer
Tungsten Optimizer**



Tungsten Overview

- Improve Spark execution by optimizing CPU / memory usage
 - Understands and optimizes for hardware architectures
 - Tunes optimizations for Spark's characteristics



Tungsten's Binary Format

- Binary representation of Java objects (Tungsten row format)
 - Different from Java serialization and Kryo
- Advantages include:
 - Much smaller size than native Java serialization
 - Supports off-heap allocation
 - Structure supports Spark operations **without deserialization**
 - e.g. You can sort data while it remains in the binary format
 - Avoids GC overhead
- Result:
 - Much faster, less memory, less CPU
 - Can process much larger datasets

Cache-Aware Computation

- Modern hardware has many types of memory
 - Main memory
 - Multiple levels of cache (L1, L2, ...)
 - Registers
 - These all have different characteristics
- Tungsten is aware of the different types of memory and different hardware architectures
 - It generates code that is optimized to utilize modern hardware
- Result: Much faster performance
 - For instance, it will use a cache-aware sorting algorithm
 - Giving 3x improvement over the non-cache aware version

Whole-Stage Code Generation

- Tungsten optimization to improve execution performance
 - Collapses a query expression into a single optimized function
- For example suppose we were filtering on the following
`filter('age>25 && 'age<50)`
 - Code generation dynamically generates bytecode for this
 - All code contained in one function
 - Instead of classic interpretation
 - With boxing of primitives, polymorphic function calls ...
- Result:
 - Eliminates virtual function calls
 - Leverages CPU registers for intermediate data
 - Can meet/exceed performance of hand-tuned function for a task

How Do You Use Tungsten

- Generally, don't think about it much
 - Just enjoy the benefits
- Ahhh — Except sometimes for **lambdas!**
 - These require Java objects, and can't be executed with data in the Tungsten format
 - This adds a layer of complexity when they interact with Tungsten
- Let's look at it in action
 - We can see it at work in the Physical Plan
- We will look at a DataFrame and Dataset Physical plan
 - Illustrating what Tungsten is doing
 - Exploring some issues with lambdas

DataFrame Physical Plan

- **SerializeFromObject** converts data to Tungsten binary format
 - First thing that's done after reading data in
 - All operations after that are done on this highly efficient format
 - Operations with asterisk (*) use Whole-Stage Code Gen

```
// Prepare the data
> val linesDF = sc.parallelize(Seq("Twinkle twinkle little star", "How I
wonder what you are", "Twinkle twinkle little star")).toDF()
// View physical plan with DataFrames
> linesDF.select(explode(split('value, "\s+")).as("word"))
   .select(lower('word).as("word")).groupBy('word).count.explain
== Physical Plan ==
*HashAggregate(keys=[word#563], functions=[count(1)])
+- Exchange hashpartitioning(word#563, 200)
  +- *HashAggregate(keys=[word#563], functions=[partial_count(1)])
    +- *Project [lower(word#560) AS word#563]
      +- Generate explode(split(value#555, \s+)), false, false, [word#560]
        +- *SerializeFromObject [staticinvoke(class
org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0,
java.lang.String, true], true) AS value#555]
          +- Scan ExternalRDDScan[obj#554]
```

Tungsten Improves Efficiency

- The previous example demonstrates several things
- Operations are performed on **data in Tungsten Binary Format**
 - Data is serialized into this format (`SerializeFromObject`) at the very start of the pipeline
 - All succeeding operations work on data in this format
 - Highly memory efficient.
- Most operations use **Whole-Stage Code Generation**
 - Indicated by the asterisk (*) before the operation name
 - Highly CPU efficient
- Result: Significantly faster execution and reduced memory usage
 - Huge win for big data workflows

Dataset Physical Plan

- **AppendColumnsWithObject** converts to Tungsten format
 - After MapPartitions, which must work on Java objects
 - Note that AppendColumnsWithObject does **NOT** use code gen
 - Less efficient than the previous plan

```
// Convert linesDF to Dataset, set up word count, and look at plan
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
  s).count.explain
== Physical Plan ==
*HashAggregate(keys=[value#496], functions=[count(1)])
+- Exchange hashpartitioning(value#496, 200)
  +- *HashAggregate(keys=[value#496], functions=[partial_count(1)])
    +- *Project [value#496]
    +- AppendColumnsWithObject <function1>, [...]
      +- MapPartitions <function1>, obj#492: java.lang.String
        +- Scan ExternalRDDScan[obj#483]
```

Extra Serialize/Deserialize

- We've made a trivial change when loading the data
 - Naming our input row "line" instead of the default "value"
 - Note how this adds extra work in our plan
 - There is an extra serialize (to Tungsten format) then deserialize (to Java format) step — the flatMap lambda needs a Java object
 - This is unexpected, and adds extra overhead ⁽¹⁾

```
// Trivial change - name column "line" instead of default "value"
val linesDF = sc.parallelize(Seq("Twinkle twinkle" ... )).toDF("Line")
linesDF.as[String].flatMap(_.toLowerCase().split("\\s")).groupByKey(s =>
s).count.explain
== Physical Plan == // rest of plan as before
+- *Project [value#520]
  +- AppendColumnsWithObject <function1>, ... AS value#520]
    +- MapPartitions <function1>, obj#516: java.lang.String
      +- DeserializeToObject line#509.toString, obj#515: java.lang.String
        +- *Project [value#507 AS line#509]
          +- *SerializeFromObject [... AS value#507]
            +- Scan ExternalRDDScan[obj#506]
```

Lab 5.3: Seeing Tungsten at Work

In this lab, we'll examine several transformations and observe how Tungsten optimizes them

A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt, standing in what appears to be a large industrial or warehouse setting. He is gesturing with his hands while speaking. In the foreground, the back of another person's head and shoulders are visible, suggesting an audience. The background shows high ceilings with exposed pipes and beams.

Session 6: Performance Tuning

- Caching
- Joins, Shuffles, Broadcasts, Accumulators
- General Guidelines

Session Objectives

- Understand some of the mechanics of Spark "under the hood"
 - And how they relate to performance
- Examine expensive operations such as joins and unions
 - Why they are expensive, and how to optimize them
- Learn some general guidelines for optimization



Caching

Joins, Shuffles, Broadcasts, Accumulators

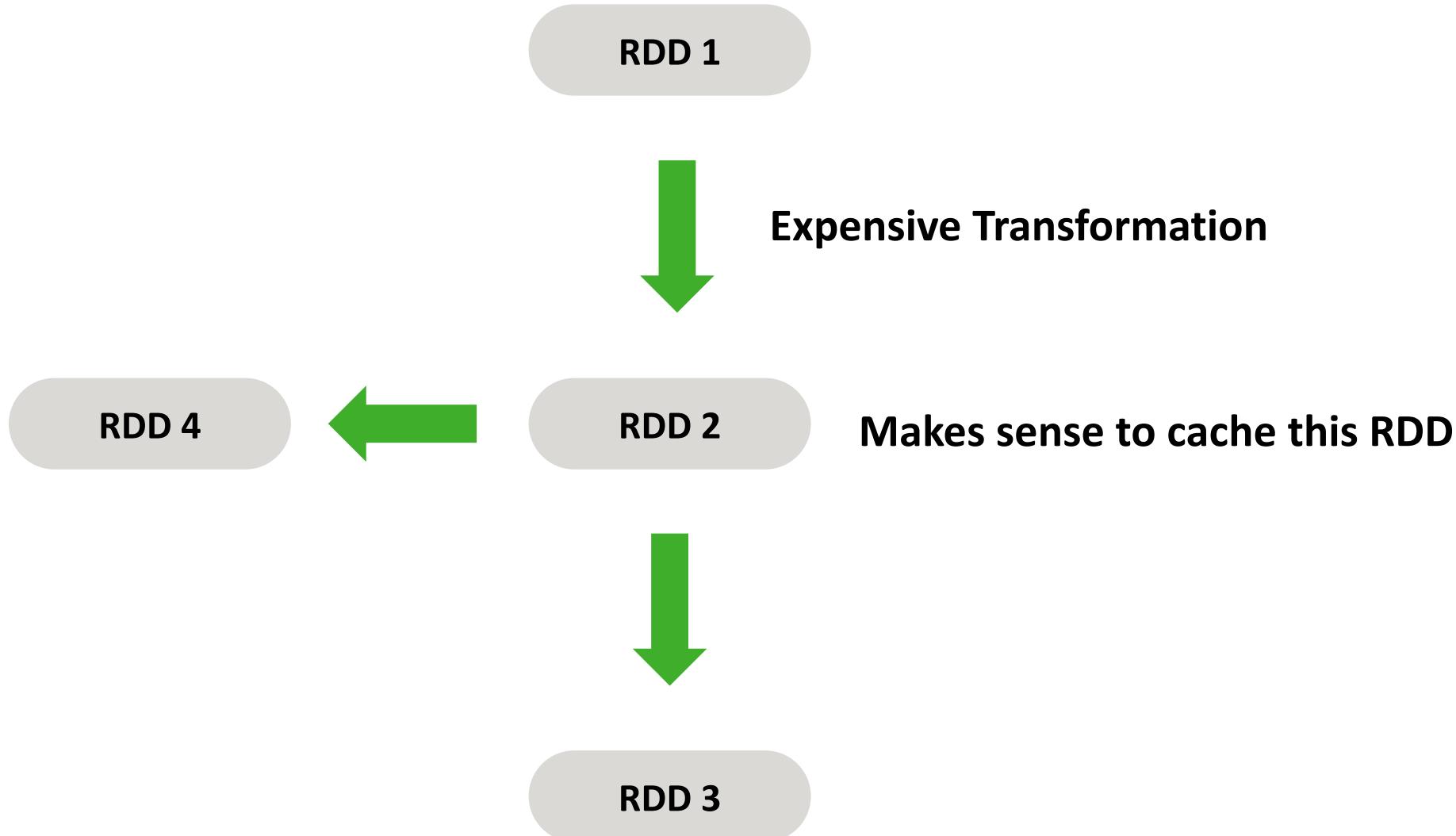
General Guidelines

Motivation for Caching

- Standard Spark job sequence:
 - Build a graph of transformations
 - Upon an action, run the DAG, get the result
 - **Don't save** intermediate RDDs or Datasets
- Intentional — you could quickly run out of memory otherwise
 - But sometimes you do want to cache an RDD
- Spark can persist an RDD/Dataset across operations
 - You must explicitly ask for this
 - RDDs and Datasets have the same API and behavior ⁽¹⁾
- Caching use cases include
 - Saving a result of expensive compute
 - Iterative workloads (machine learning)

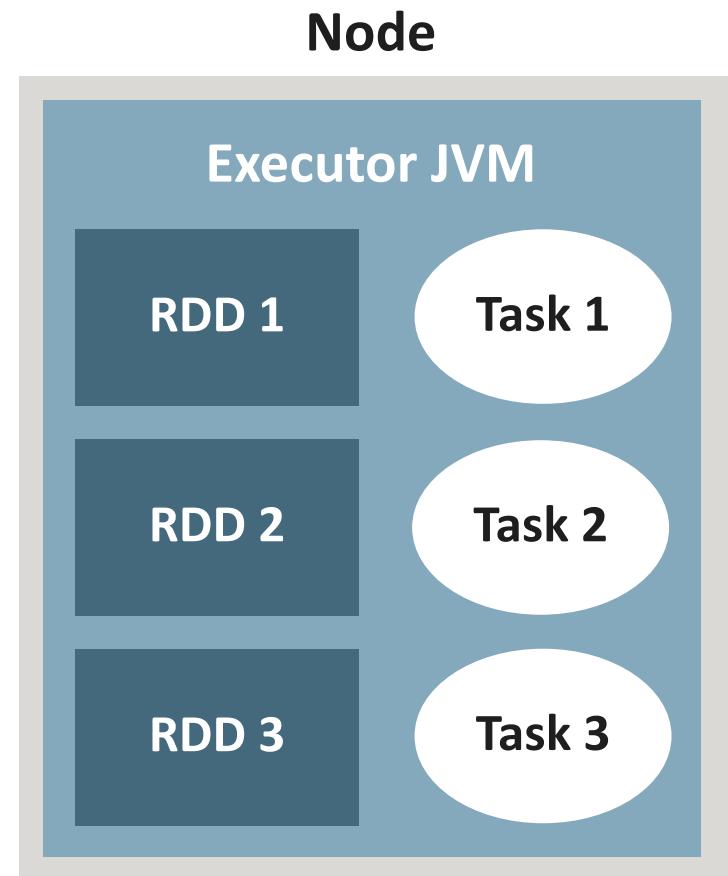
Caching Illustrated

- Below, RDD 2 is used to compute two other RDDs
 - It may make sense to cache it



Caching Mechanics

- Can cache into
 - Memory
 - Disk
 - Combination of above
- Memory caching is done by **executors on worker nodes**
- Beware of JVM memory limits
 - Min JVM memory : 4-8G
 - Max JVM memory : 40+G
 - Generally, the more memory, the more GC time
 - Limits depends on your GC
 - Java 8 GC ('G1') has higher limits



Persistence Levels

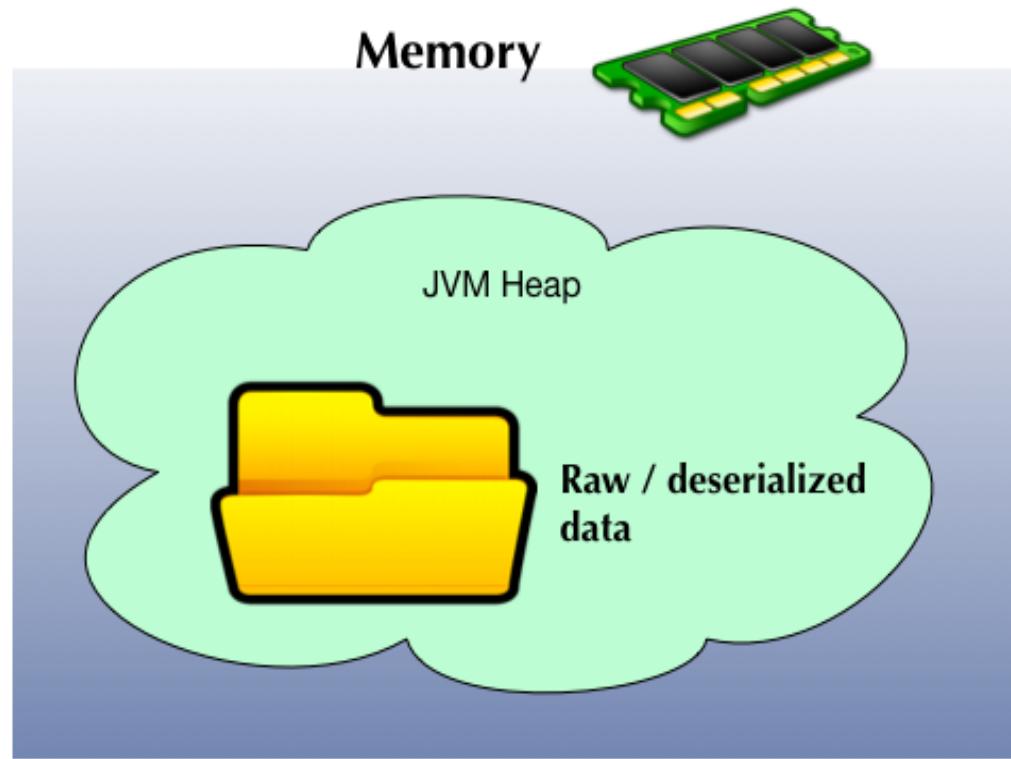
- Spark provides several levels of persistence
 - As described below and the following slides
 - Nodes store partitions for reuse, so future actions are faster

Storage Level	Behavior
MEMORY_ONLY (default level)	Store as deserialized Java objects in JVM. If RDD doesn't fit in memory, some partitions not cached, and recomputed
MEMORY_AND_DISK	Store as deserialized Java objects in JVM. If RDD doesn't fit in memory, store those partitions on disk, and read as needed
MEMORY_ONLY_SER (Java and Scala)	Store as serialized Java objects. Generally more space-efficient than deserialized, but more CPU-intensive to read
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Store data in serialized format off-heap. Reduces garbage collection overhead as well as other benefits

MEMORY_ONLY (Raw Java Objects)

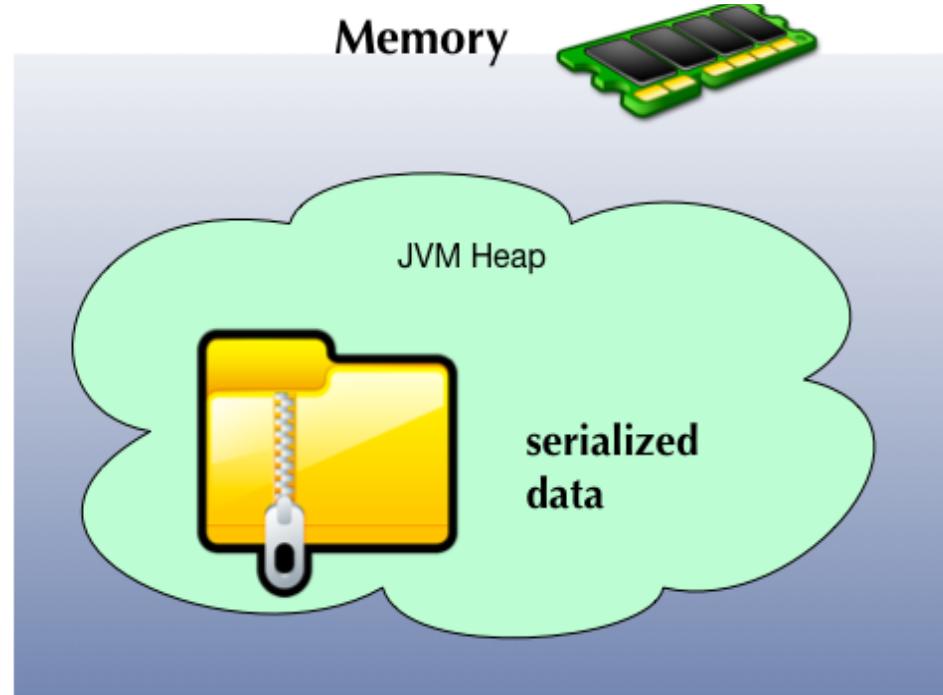
- `cache()` or `persist(MEMORY_ONLY)`

- Most CPU efficient
- Data stored as ‘raw’ / serialized
- Takes up memory (3x – 5x)
- 1G raw data uses 3G – 5G memory



MEMORY_ONLY_SER (Serialized)

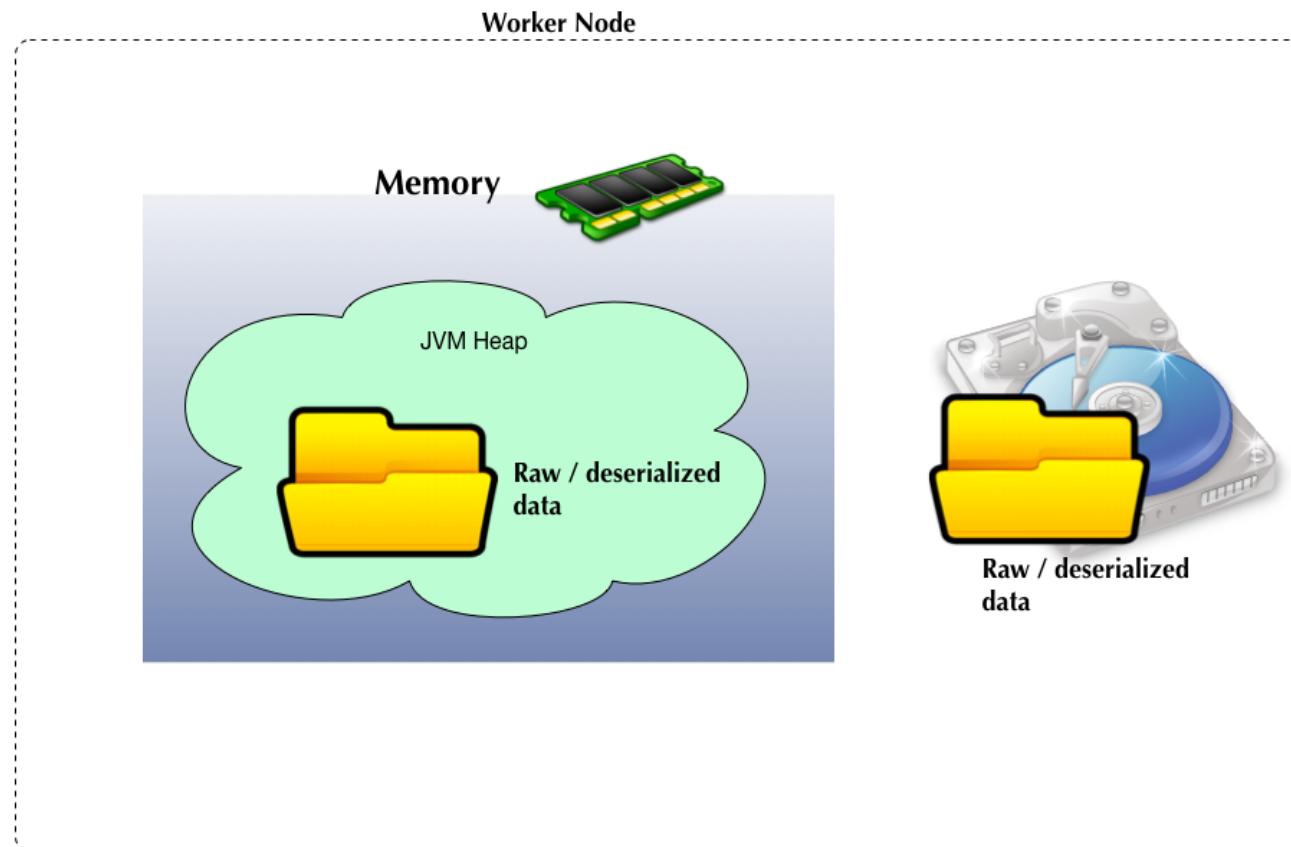
- `persist(MEMORY_ONLY_SER)`
 - Most memory efficient option
 - Little memory overhead
 - CPU intensive (to serialize / deserialize)
 - Default Java serializer is OK — use ‘kryo’ serializer for high performance



MEMORY_AND_DISK

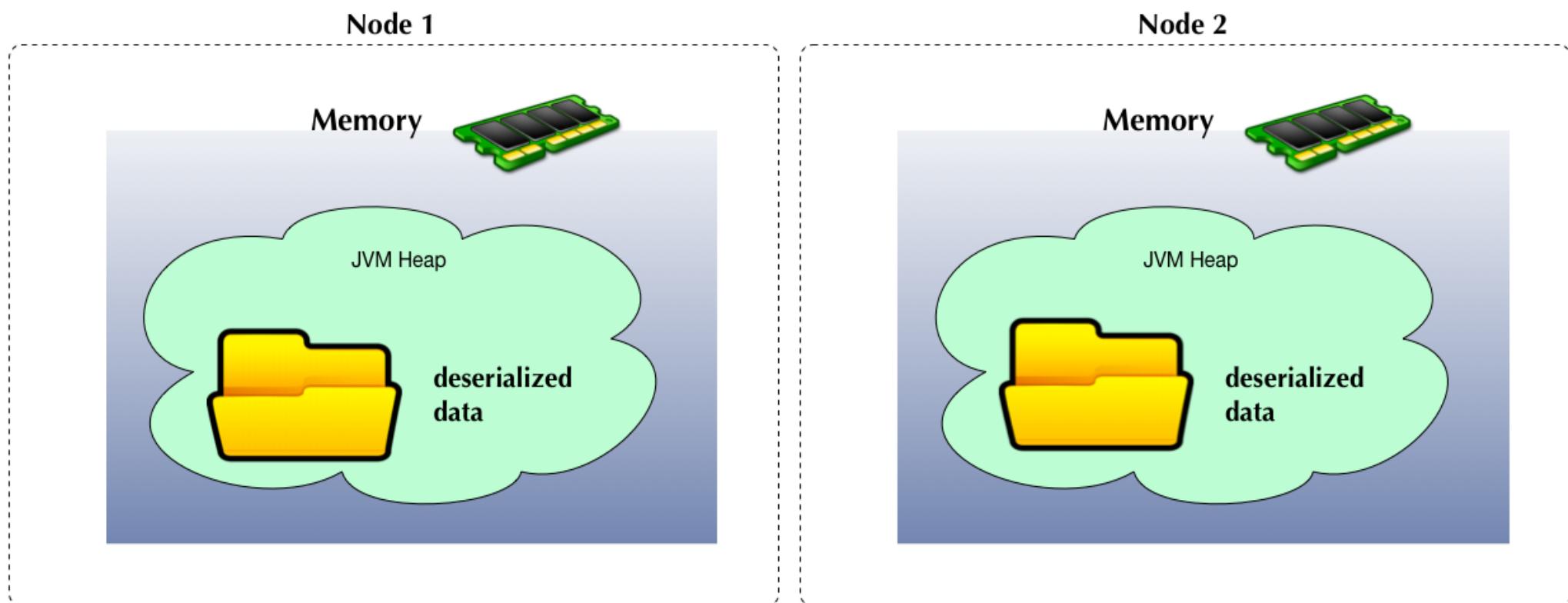
- `persist(MEMORY_AND_DISK)`

- Both in memory & disk
- Can survive memory eviction



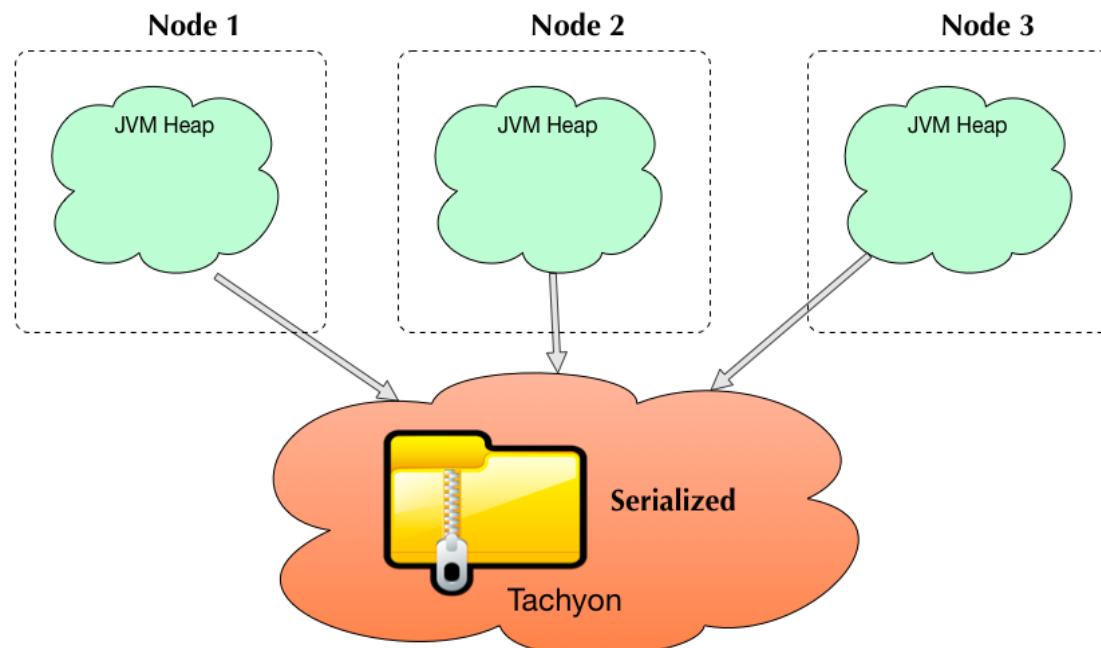
MEMORY_ONLY_2: Cache on Multiple Nodes

- `persist(MEMORY_ONLY_2)`
 - Survives a node failure

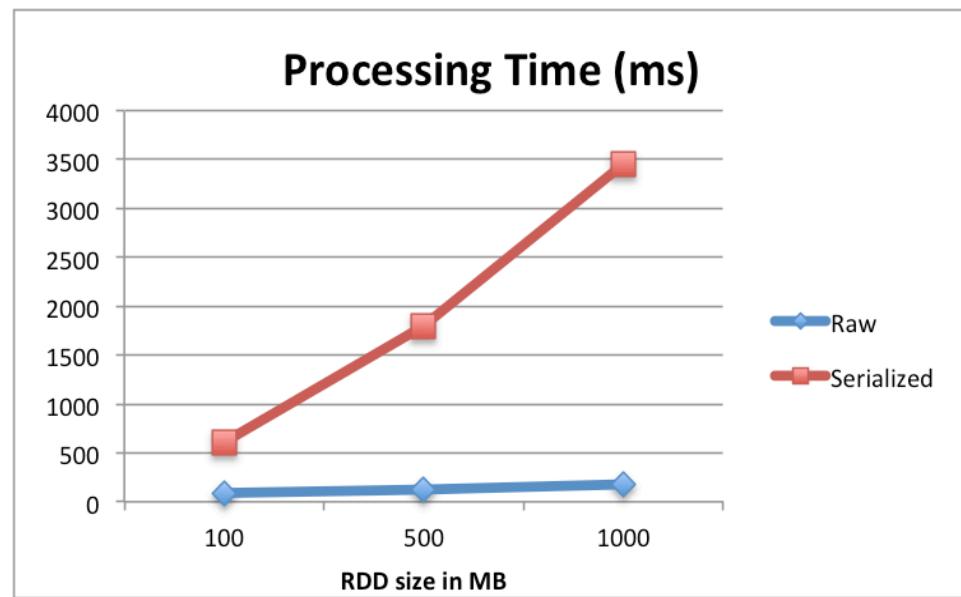
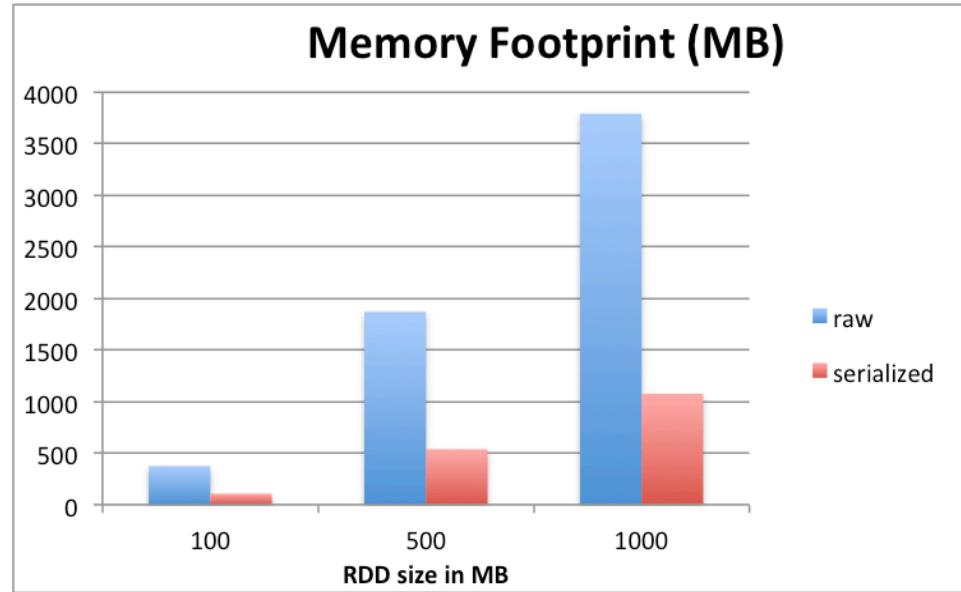


OFF_HEAP (Experimental)

- In memory caching (Uses Tachyon/Alluxio)
- Cached data distributed across nodes (scale out)
- Doesn't use JVM for storage
 - No need to worry about Garbage Collection (GC)
 - Can accommodate very large data sets



Memory Caching Implications



- Raw caching consumes more memory (2-5x)
 - But is faster to process
 - MEMORY_ONLY
- Serialized caching uses less memory
 - Processing time is more
 - MEMORY_ONLY_SER

Guidelines to Using Caching

- Only cache data that is being reused the most
 - If you cache too much data, you'll run out of memory
 - The least-recently-used data will be pushed out
- If data fits in memory use MEMORY_ONLY (the default)
 - Most CPU efficient
- If not, try using MEMORY_ONLY_SER
 - And select a fast serialization library⁽¹⁾
- Don't use disk unless computation is expensive or filters lots of data
 - Otherwise, recomputing may be as fast as reading from disk

Guidelines to Using Caching

- Use replicated storage for fast fault recovery.
 - You have fault recovery anyway, but replicated has less down time ⁽¹⁾
- Consider OFF_HEAP (Tachyon) storage
 - When you have lots of memory
 - When multiple applications need to share the data
 - Several advantages
 - Multiple executors can share memory pool in Tachyon
 - Reduces garbage collection
 - Cached data not lost when an executor crashes
 - Applications external to Spark can access the data

Persistence API

- The following methods are used for persistence
 - **persist(newLevel: StorageLevel)**: Set storage level to newLevel (only valid if storage level never set)
 - **persist()**: Same as persist(StorageLevel.MEMORY_ONLY)
 - **cache()**: Same as persist(StorageLevel.MEMORY_ONLY)
- Below, we give a usage example
 - Note that the call to cache doesn't have any immediate affect
 - It's added to the DAG, and when it's eventually created, Spark knows to cache the data

```
val visits = sc.textFile("visits.txt").map(...)  
val pageNames = sc.textFile("pages.txt").map(...)  
val joined = visits.join(pageNames) // We will be reusing this  
joined.cache() // So cache it
```

Lab 6.1: Caching

In this lab, we'll persist some of our data and examine the affects on performance.



Caching
→ **Joins, Shuffles, Broadcasts, Accumulators**
General Guidelines

Joins Revisited

- Let's consider the following two Datasets

```
> val largeDF = // ... See notes for initialization - has 1M rows
> largeDF.repartition(100)
> largeDF.limit(3).show
+---+---+
| num|bit|
+---+---+
| 1| 1|
| 2| 0|
| 3| 1| // Remaining rows to 1M omitted ...

> val smallDF = // ... See notes for initialization. Has 2 rows
smallDF.show
> smallDF.show
+---+-----+
| bit|bitName|
+---+-----+
| 0| zero|
| 1| one|
+---+-----+
```

Consider a Standard Join

- We join our two DataFrames below — this is problematic ⁽¹⁾
 - The sort-merge-join shuffles all the rows of largeDF
 - By the bit key (the join column) — Lots of shuffling
 - Not enough keys for parallelism !
 - End up with 2 (only) non-empty partitions
 - Adding more worker nodes does NOTHING to help this job

```
> largeDF.join(smallDF, largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*SortMergeJoin [bit#1345], [bit#1338], Inner
:- *Sort [bit#1345 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(bit#1345, 200)
:    +- *Filter isnotnull(bit#1345)
:      +- Scan ExistingRDD[num#1344,bit#1345]
+- *Sort [bit#1338 ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(bit#1338, 200)
    +- *Filter isnotnull(bit#1338)
      +- *SerializeFromObject [...]
      +- Scan ExternalRDDScan[obj#1337]
```

Reduce Shuffling: Share Data from App

- One technique is to remove the shuffle completely
 - By sending data from the client (the driver program)
 - We illustrate this below

```
> val largeDF = // ... As previously - has 1M rows
> case class Bit(bit:Int, bitName: String)
> val smallArray = List (Bit(0,"zero"), Bit(1,"one"))
> case class Result(num:Int, bit:Int, bitName:String)
> val mappedDF = largeDF.map(r => {
  if (r.getInt(1) == smallArray(0).bit) {
    Result(r.getInt(0), r.getInt(1), smallArray(0).bitName)
  } else {
    Result(r.getInt(1), r.getInt(1), smallArray(1).bitName)
  } })
> mappedDF.limit(2).show
+---+---+-----+
| num|bit|bitName|
+---+---+-----+
|  1|  1|    one|
|  2|  0|   zero|
```

Shared Data Physical Plan

- The plan below shows there is no shuffling
 - But each time `smallArray` is used, it's **sent across the network**
 - For bigger data (e.g. 20MB) used repeatedly, could be a lot of overhead
- There is also a lot of serialization activity
 - A Java object is created for our lambda (from Tungsten format)
 - Then, it's serialized back into Tungsten format

```
> mappedDF.explain
== Physical Plan ==
*SerializeFromObject [...]
+- *MapElements <function1>, obj#21: $line24.$read$$iw$$iw$Result
  +- *DeserializeToObject createExternalRow(...)
    +- Scan ExistingRDD[num#2,bit#3]
```

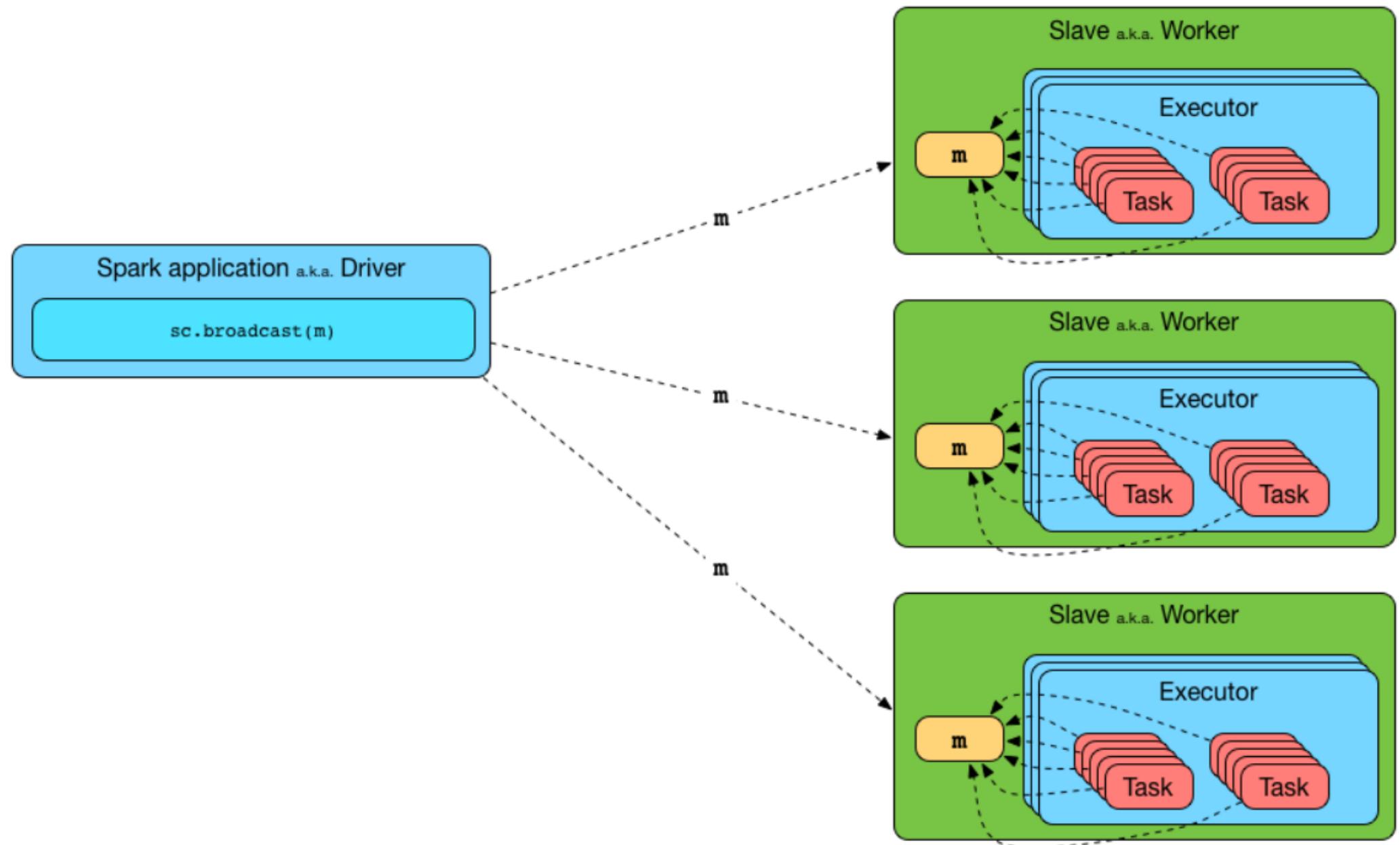
Spark Joins



Broadcast Variables for Sharing Data

- **Broadcast Variables** cache read-only data on each node
 - And can be reused at no additional cost
 - Create it via `SparkContext.broadcast()`
 - Access as `variable.value`
 - Broadcast variables are **immutable**
 - Changes are NOT propagated anywhere ⁽¹⁾
 - We illustrate a broadcast variable sharing the `smallArray`

```
// largeDF smallArray, and case class Result declared as previously
// Create Broadcast Variable
> val smallArrayBC = sc.broadcast(smallArray)
// Use it in the map
> val mappedDF = largeDF.map(r => {
  if (r.getInt(1) == smallArrayBC.value(0).bit) {
    Result(r.getInt(0), r.getInt(1), smallArrayBC.value(0).bitName)
  } else {
    Result(r.getInt(1), r.getInt(1), smallArrayBC.value(1).bitName)
  } })
```



Lifecycle of Broadcast Variable

```
scala> val b = sc.broadcast(1)
b: org.apache.spark.broadcast.Broadcast[Int] = Broadcast(0)
```

```
DEBUG BlockManager: Put block broadcast_0 locally took 430 ms
DEBUG BlockManager: Putting block broadcast_0 without replication took 431 ms
DEBUG BlockManager: Told master about block broadcast_0_piece0
DEBUG BlockManager: Put block broadcast_0_piece0 locally took 4 ms
DEBUG BlockManager: Putting block broadcast_0_piece0 without replication took 4 ms
```

Lifecycle of Broadcast Variable

After creating an instance of a broadcast variable, you can then reference the value using [value](#) method.

```
scala> b.value  
res0: Int = 1
```

When you are done with a broadcast variable, you should [destroy](#) it to release memory.

```
scala> b.destroy  
  
DEBUG BlockManager: Removing broadcast 0  
DEBUG BlockManager: Removing block broadcast_0_piece0  
DEBUG BlockManager: Told master about block broadcast_0_piece0  
DEBUG BlockManager: Removing block broadcast_0
```

Catalyst and Broadcasting

- Catalyst can handle the broadcast for you
 - You can give it a hint to broadcast (shown below)
 - `broadcast()` is a function that does this
- Catalyst uses a **BroadcastHashJoin**
 - Which broadcasts the `smallArrayDF`
 - Also — **no serializing**
 - This is the most efficient plan, by far

```
// largeDF and smallDF declared as previously
import org.apache.spark.sql.functions.broadcast
> largeDF.join(broadcast(smallDF),
                 largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#60], Inner, BuildRight
:- *Filter isnotnull(bit#3)
:  +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...))
   +- LocalTableScan [bit#60, bitName#61]
```

Catalyst Automatic Broadcasting

- Catalyst can automatically choose a broadcast (see at bottom)
 - If your dataframe size is under the value of config param **spark.sql.autoBroadcastJoinThreshold** (10MB default)
 - And catalyst can tell the size of your data ⁽¹⁾
 - We previously disabled this by setting the threshold to -1
 - To illustrate the non-broadcast plan in earlier examples

```
// largeDF and smallDF declared as previously
// Set broadcast threshold explicitly to 10MB
> spark.conf.set("spark.sql.autoBroadcastJoinThreshold", 1024*1024*10)

> largeDF.join(smallDF, largeDF("bit") === smallDF("bit")).explain
== Physical Plan ==
*BroadcastHashJoin [bit#3], [bit#8], Inner, BuildRight
:- *Filter isnotnull(bit#3)
:  +- Scan ExistingRDD[num#2,bit#3]
+- BroadcastExchange HashedRelationBroadcastMode(...)
   +- LocalTableScan [bit#8]
```

Accumulators for Shared Calculation

- **Accumulators** are variables that can be added to in parallel
 - Added to via special associative operators ($+=$, add)
 - Workers can add to the variable, only the driver can read it
 - Can be used for counters or sums
 - We illustrate a simple example below
 - Use cases : counting website traffic / transactions per minute .etc

```
> val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0

> sc.parallelize(Array(1, 2, 3, 4)).
           foreach(x => accum += x)

> accum.value
res5: Int = 10
```

Broadcast Summary

- Useful to reduce shuffling
 - Remember — broadcast data has to fit in memory
 - And if it's not being used more than once, it may be fine to pass it in the transformation instead of broadcasting
- Catalyst can help you!
 - It has optimizations that will automatically broadcast when appropriate
 - Or you can do it manually
- Even if you broadcast, you might have other issues
 - e.g. the serialization we saw with the shared data approach

Lab 6.2: Broadcast

In this lab, we'll view how broadcasts can improve our efficiency



Caching
Joins, Shuffles, Broadcasts, Accumulators



General Guidelines

Use the Spark UI

- It will point you to problem areas
 - We've seen it in many labs and slides (port 4040)
- Things to look at
 - Task execution time
 - Amount of shuffle data
 - Partitions and overall data volume
 - GC time
 - DAG lineage
- Can run the history server for stats of completed jobs
 - Pulls info from application event logs
 - Displays it in typical Spark UI form

Use Efficient Transformations

- Transformation choices will greatly affect your performance
 - We've seen that many times
 - A few guidelines will help
- Use **DataFrames/Datasets** for Catalyst/Tungsten benefits
- **Minimize shuffles** and use effective joins
 - As seen earlier — Catalyst will help you with this
 - Consider broadcast variables and accumulators
- Beware of **lambda functions** — they raise issues with Catalyst/Tungsten
- Be aware of your data characteristics
 - e.g. is all your data on one partition — **problem!**

Filter Early

- Filter out unused data as early as possible
 - Don't process it, then throw it out
 - More network traffic, CPU usage, storage requirements, etc.
 - Catalyst will help you with this, but not always
 - Depending on your transformations
- Filter runs parallel on all partitions
 - Narrow dependency — very efficient
- Filtered partitions can be **unbalanced**
 - With one much larger than another
 - Beware of this — consider coalesce or repartition
 - Be judicious— these can be expensive
 - Check the resource usage (e.g. Web UI)

Use Good Data Storage

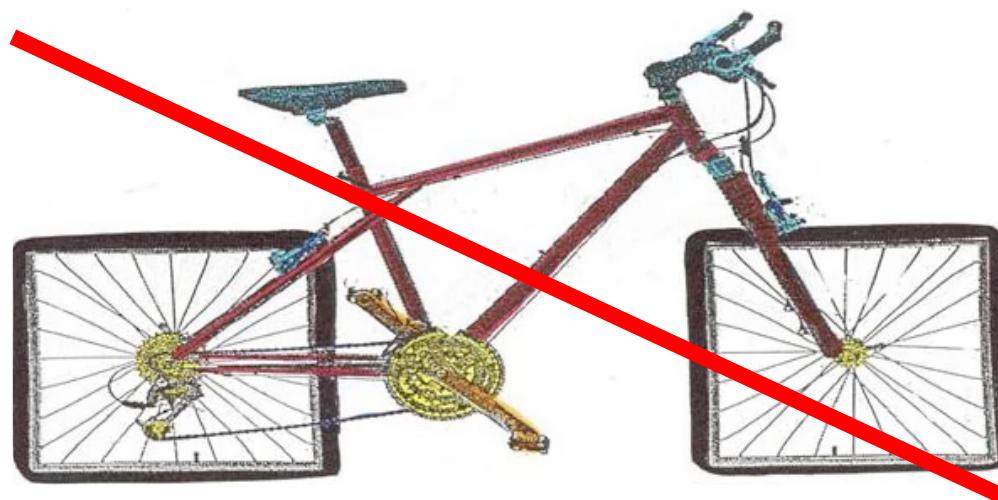
- Spark needs a reliable data store for data
 - Several choices
- **HDFS** is often a good choice
 - Cheap(er), Reliable, Scalable
 - Proven infrastructure
- **NoSQL** data stores
 - Good for real time work loads
 - Cassandra, Couchbase, Aerospike, etc.
 - Many are already integrated with Spark
- Exploit data locality
 - Process data on same node — supporting high throughput
 - Works well with HDFS, as Spark understands its storage

Monitor, Monitor, Monitor

- Put resources into planning your monitoring
 - Will help you diagnose performance issues
- Monitor at the system level
 - CPU, Memory
 - JVM, Garbage collection
- Monitor at application level
 - Transformation times and resources
 - Should be checked while developing — often by developer
- Collect and graph metrics
 - Codahale, Graphite, Graphana

Don't Reinvent the Wheel

- Spark has a large community
 - Someone should be a part of it to gain from its experience
- There are a lot of resources
 - There are a number of excellent books by Spark contributors
 - This is true even for Spark 2
 - Use them, follow their expert advice
- Don't start from scratch with a square wheel !
 - Hey all — this didn't work, you should go another route



A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt and blue jeans. He is standing in what appears to be a workshop or industrial setting, with pipes and equipment visible in the background. He is gesturing with his hands while speaking. A green diagonal bar runs from the bottom right corner across the slide.

Session 7: Creating Standalone Applications

- Core API
- Building and Running Applications
- Application Lifecycle
- Cluster Managers
- Logging & Debugging

Session Objectives

- Cover the Spark API for writing self-contained programs
 - As opposed to using the Spark Shell
- Write and compile standalone Spark applications
- Submit applications to Spark



Core API

Building and Running Applications

Application Lifecycle

Cluster Managers

Logging & Debugging

Spark Applications

- We've been using the Spark Shell
 - Stand-alone
 - Or connecting to cluster
- Shell is great for
 - Ad-hoc / interactive
 - Developing apps / Debugging
- For production code, want an actual application
 - Main difference — you create a `SparkSession`
 - Instead of using pre-created session in shell
 - Fairly simple using some boilerplate code
 - Can be in Scala / Python / Java
 - We'll cover the Scala API, but all are similar

Basic Code for Client (Driver)

- Create a program (object with `main` method)
- Access a `SparkSession.Builder`
 - Fluent interface for configuring/building a session
 - Access Builder API docs through `object` `SparkSession`
- Configure the session (via a fluent interface)
- Create the session, then use as needed the same as previously

```
// Need to import whatever types we need now (1)
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object MyApp {                                // Basic Spark App (Scala)
    def main(args: Array[String]) {
        val spark = SparkSession.builder // Access the builder
            .appName("MyApp")          // Set application name
            .master("local[4]")         // Run locally with 4 cores
            .getOrCreate()              // Create the session
        // Use session as needed ...
    }
}
```

Common Builder Methods

- All return the Builder object itself
 - Except for `getOrCreate()` which returns the session

Method	Description	Example
<code>appName(name: String)</code>	Set name for app — shown in Web UI	<code>appName("MyApp")</code>
<code>master(master: String)</code>	Set URL of master to connect to	<code>master("local[4]")</code>
<code>config(key: String, value: String)</code>	Set a config option with string value	<code>config("cassandra.host", "host1")</code>
<code>config(key: String, value: xxx)</code> Multiple versions for common types (e.g. Long)	Set a config option with other type of value (e.g. Long)	<code>config("spark.driver.cores", 1)</code>
<code>enableHiveSupport()</code>	Enables Hive Support	<code>enableHiveSupport()</code>
<code>getOrCreate()</code>	Gets existing SparkSession or creates new one (does not return Builder)	<code>getOrCreate()</code>

Master URL Variants

Master Key	Description	Example
Local		
local	localhost with a single CPU core	“local”
local[N]	Run on localhost with N CPU cores	“local[4]”
local[*]	Run on localhost with all CPU cores	“local[*]”
Distributed		
spark://host:port	Spark master (running stand alone)	spark://masterhost1:7077
mesos:// host:port	Spark master (running on Mesos)	mesos://host1:5050
Yarn	Running on YARN	“yarn”

SparkSession vs. SparkContext

- A SparkSession wraps a SparkContext instance
 - Generally, you'll program to a SparkSession
 - Spark uses the wrapped SparkContext internally to do computations
 - Note that the session (and underlying context) is a **singleton** —
 - `getOrCreate()` will return an already existing session
- Can access the SparkContext as needed (e.g. for broadcast variables) via;
`spark.sparkContext`
- Can create a SparkContext directly
 - Instead of a SparkSession
 - See notes for example

Some Common Config Properties

- These are useful for applications
 - Many, many more
 - See <https://spark.apache.org/docs/latest/configuration.html>

Property	Default	Meaning
<code>spark.master</code>	(none)	Master URL (same as <code>master()</code>)
<code>spark.app.name</code>	(none)	Application name (same as <code>appName()</code>)
<code>spark.driver.cores</code>	1	Number of cores for driver process (cluster mode only)
<code>spark.driver.maxResultSize</code>	1G	Total size of serialized results for Spark action
<code>spark.driver.memory</code>	1G	Amount of memory for driver process (not used in client mode)
<code>spark.local.dir</code>	/tmp	Directory for Spark scratch space
<code>spark.submit.deployMode</code>	(none)	"client" (launch locally) or "cluster" (launch on node in cluster)

Configuring Runtime Properties

- Can access/alter existing Spark runtime properties
 - Through the **SparkSession.conf** member
 - Of type `org.apache.spark.sql.RuntimeConfig`
 - We illustrate below

```
// Code fragment
// Set a single option
> spark.conf.set("spark.executor.memory", "2g")

// Get all the settings - most detail omitted
> val configMap:Map[String, String] = spark.conf.getAll
configMap: Map[String, String] = Map(spark.driver.host -> 192.168.1.128,
spark.driver.port -> 49760, ..., spark.executor.memory -> 2g, ...)
```

Other Configuration Options

- Can create **SparkConf** instance, and set its properties
 - Then pass to builder via **config(conf: SparkConf)**
- Can pass properties to spark-submit using **--conf**

```
./bin/spark-submit ... \
--conf spark.master=spark://1.2.3.4:7077
```
- spark-submit reads config options in the file **<spark>/conf/spark-defaults.conf** for
 - In standard key=value properties file format
 - Precedence order (highest to lowest):
 - (1) Properties set directly on Builder
 - (2) props passed to **spark-submit**
 - (3) *spark-defaults.conf*

MINI-LAB: Review Documentation

- We'll take a few minutes to review the API docs on these new types

Mini-Lab

- Browse to <http://spark.apache.org/docs/latest/>
 - On the top menu bar, go to [API Docs | Scala](#)
 - In the left pane, type in [SparkSession](#) in the filter box
 - Click on the O to go to the Object docs
 - In the builder() method docs, click on the [Builder](#) return val
 - This brings you to the [SparkSession.Builder](#) docs — review them
 - Go to the class SparkSession docs, and review the [conf](#) member
 - Click on its [RuntimeConfig](#) type, and review this class
 - Browse to <https://spark.apache.org/docs/latest/configuration.html>
 - Spend a few minutes reviewing this





Core API



Building and Running Applications

Application Lifecycle

Cluster Managers

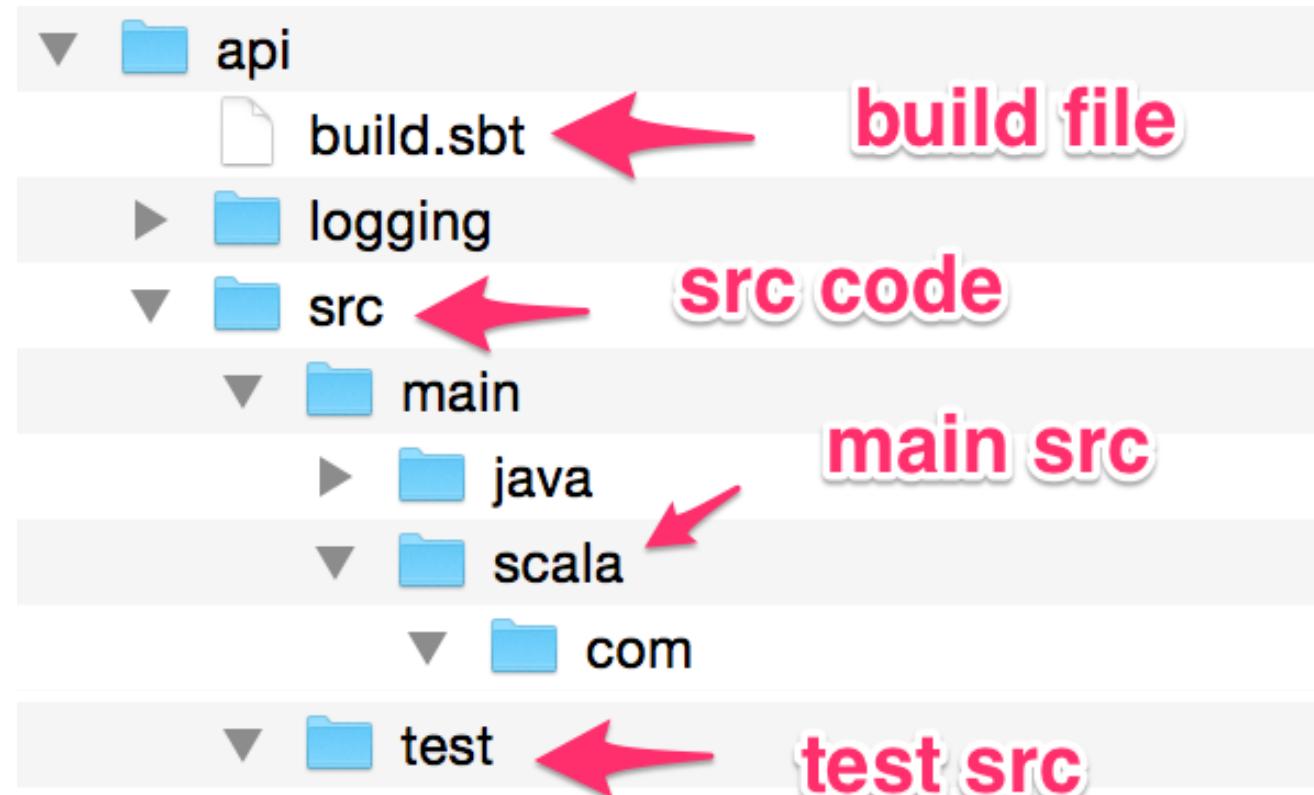
Logging & Debugging

Tools for Building/Coding

- Several choices — we'll only cover sbt
 - **sbt**: Simple Building Tool (For Scala)
 - <http://www.scala-sbt.org/>
 - **maven**: Just need the Spark dependencies — for example
- ```
<dependency>
 <groupId>org.apache.spark</groupId>
 <artifactId>spark-core_2.11</artifactId>
 <version>2.1.1</version>
</dependency>/
```
- **Scala IDE**: Eclipse based Scala IDE
  - **IntelliJ**: Excellent Scala support, fast / incremental compile
  - **Sublime**: Sophisticated text editor — full Scala support
    - <http://www.sublimetext.com/>

# sbt Application Layout (Scala)

- Uses maven layout by default



# build.sbt

- The build file for sbt
  - This one sets the app name, app version, and Scala version
  - It then configures dependencies
  - We go into enough detail on sbt for basic use, but not deep detail

```
name := "MyApp"

version := "1.0"

scalaVersion := "2.11.7"

// ++= means concatenate sequence of dependencies
// % means append Scala version to the next part (1)
libraryDependencies ++= Seq(
 "org.apache.spark" %% "spark-core" % "2.1.0" % "provided"
)

// need this to access files on S3 or HDFS
// += means just append the dependency
libraryDependencies += "org.apache.hadoop" % "hadoop-client" % "2.7.0"
exclude("com.google.guava", "guava")
```

# Compiling Code

- *build.sbt* generally in project root dir
  - Same purpose as *pom.xml* for Maven
- Automatically downloads dependencies
- sbt commands
  - sbt **compile**
  - sbt **package** — builds a jar
  - sbt **assembly** — builds a ‘fat’ jar with all the dependencies
  - sbt **clean** — cleans up all generated artifacts
- To re-build completely  
**sbt clean package**
  - First run takes a few minutes to download all deps...
  - Go for a stroll ☺

# spark-submit: Submitting An Application

- <spark>/bin/spark-submit can launch apps on the cluster
  - Can be used with all supported cluster managers
  - See next page for options explanations
- At bottom, we submit to a standalone manager, set executor memory, and pass an argument (a file name)

```
./bin/spark-submit \
--class <main-class>
--master <master-url> \
--deploy-mode <deploy-mode> \
--conf <key>=<value> \
... # other options
<application-jar> \
[application-arguments]
```

```
$ spark-submit --master spark://localhost:7077 \
--executor-memory 4G --class com.mycompany.MyApp \
target/scala-2.11/myapp.jar 1G.data
```

# spark-submit Options

Option	Description	Example
--master <master url>	Master url	--master Spark://host1:7077
--name <app name>	Application Name	--name MyApp
--class <main class>	Main class	--class com.mycompany.MyApp
--driver-memory <val>	Memory for app driver (default 512M)	--driver-memory 1g
--executor-memory <val>	Memory for executors (more important!)	--executor-memory 4g
--deploy-mode <deploy-mode>	Deploy driver to worker (cluster) or run locally (client)	--deploy-mode cluster
--conf <key>=<value>	Spark Config Property	
--help	Print out all options	

# Lab 7.1: Spark Job Submission

In this lab, we'll write a standalone job, and run it on the cluster

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

Core API

Building and Running Applications

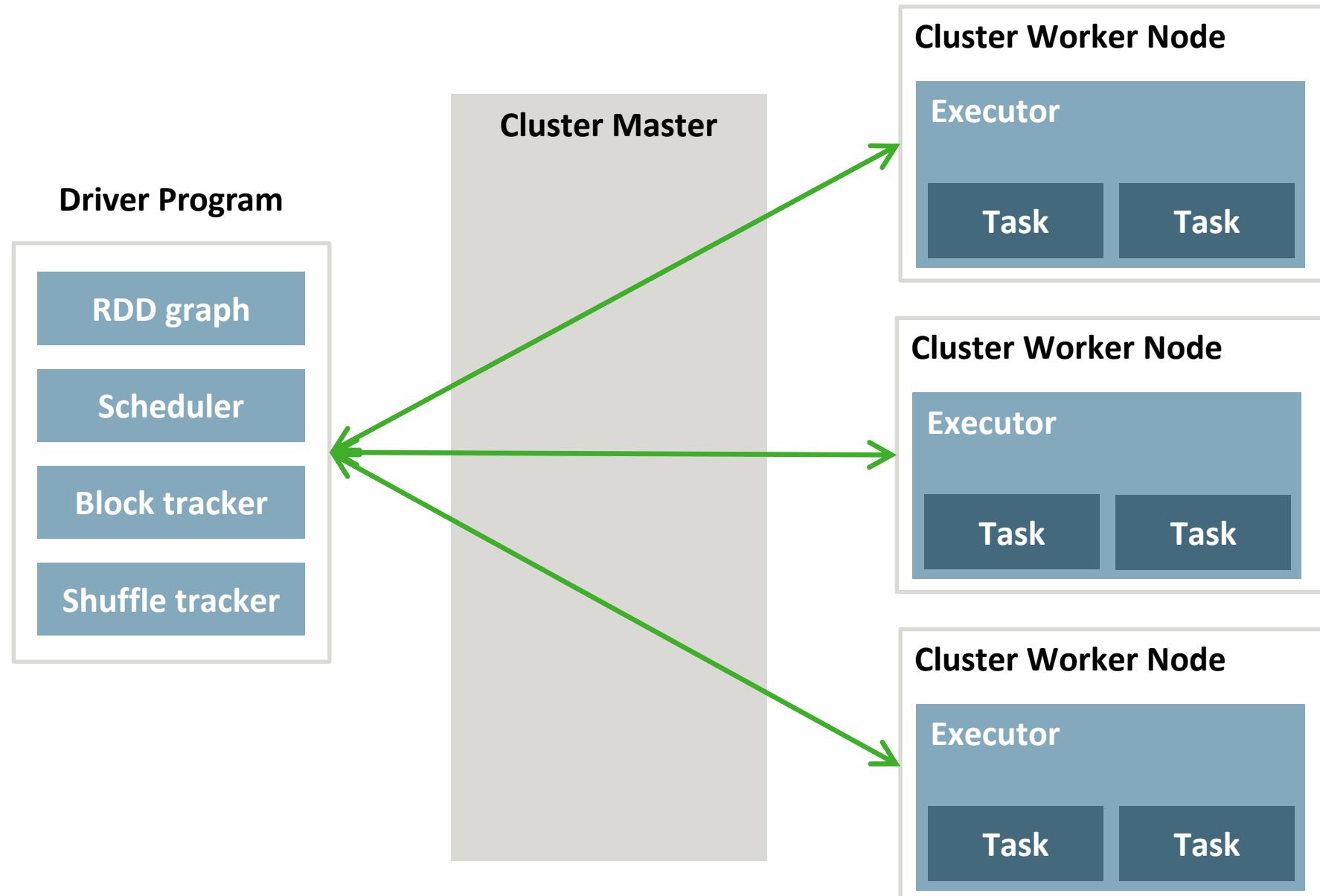


**Application Lifecycle**

Cluster Managers

Logging & Debugging

# Spark Application Architecture



# Application Driver (the Client)

- The 'main' method of an application
  - It's where the **SparkSession/SparkContext** is created
  - Establishes the connection to the cluster
  - Creates a DAG (Direct Acyclic Graph) of operations
- Connects to cluster manager to allocate resources
  - Acquires **executors** on worker nodes
  - Sends app code to executors
  - Sends **tasks** for executors to run
- Driver should be close to the worker nodes
  - Preferably on the same LAN

# Executors and Tasks

- **Executors**

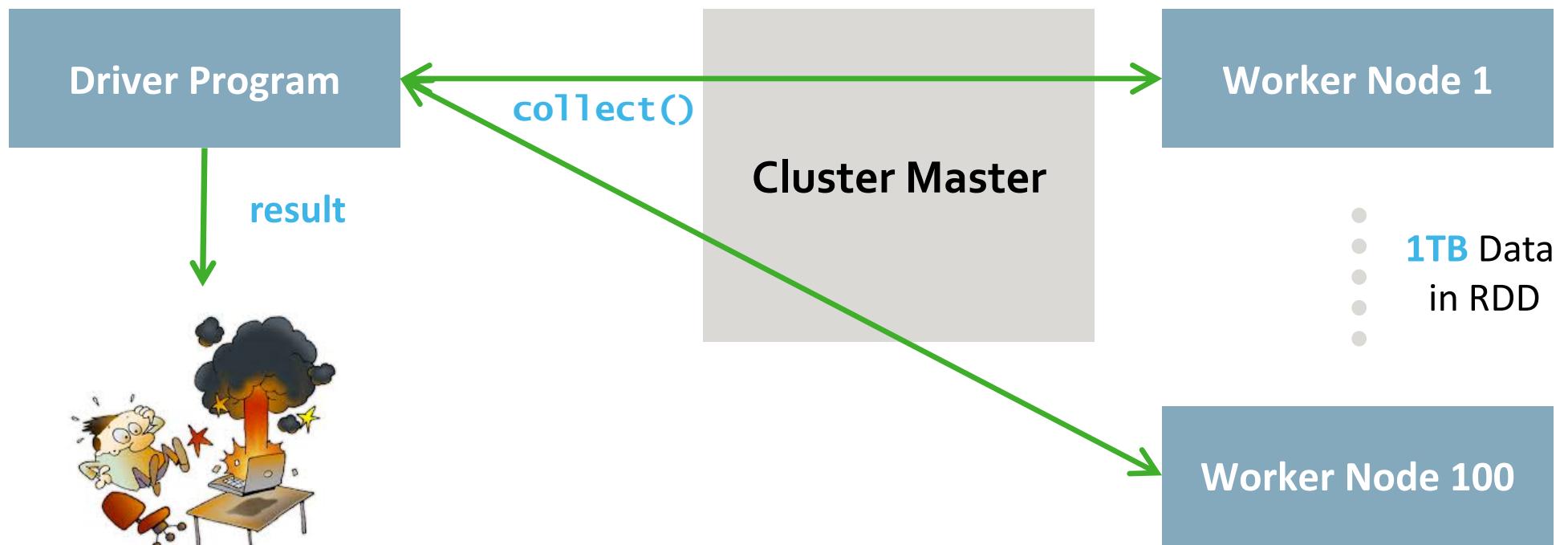
- Processes that run computations and store data
- Each app gets its own executors
- Launched at application startup, run for duration of the app
- JVM containers (tasks from different apps in different JVM)
- Execute tasks (in threads)
- Provide memory for cache storage

- **Tasks**

- 'Smallest' execution units
- Process data in partitions
- Takes into account 'data locality'
- Runs as 'threads' within executor JVM

# Driver Memory vs. Executor Memory

- Driver memory is generally small
- Executor memory is where data is cached — can be big
- **RDD.collect()** or similar operations will send data to driver
  - Large collections will cause out of memory error in driver
  - Find a different way !



A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus, showing what appears to be a server rack.

**Core API**

**Building and Running Applications**

**Application Lifecycle**



**Cluster Managers**

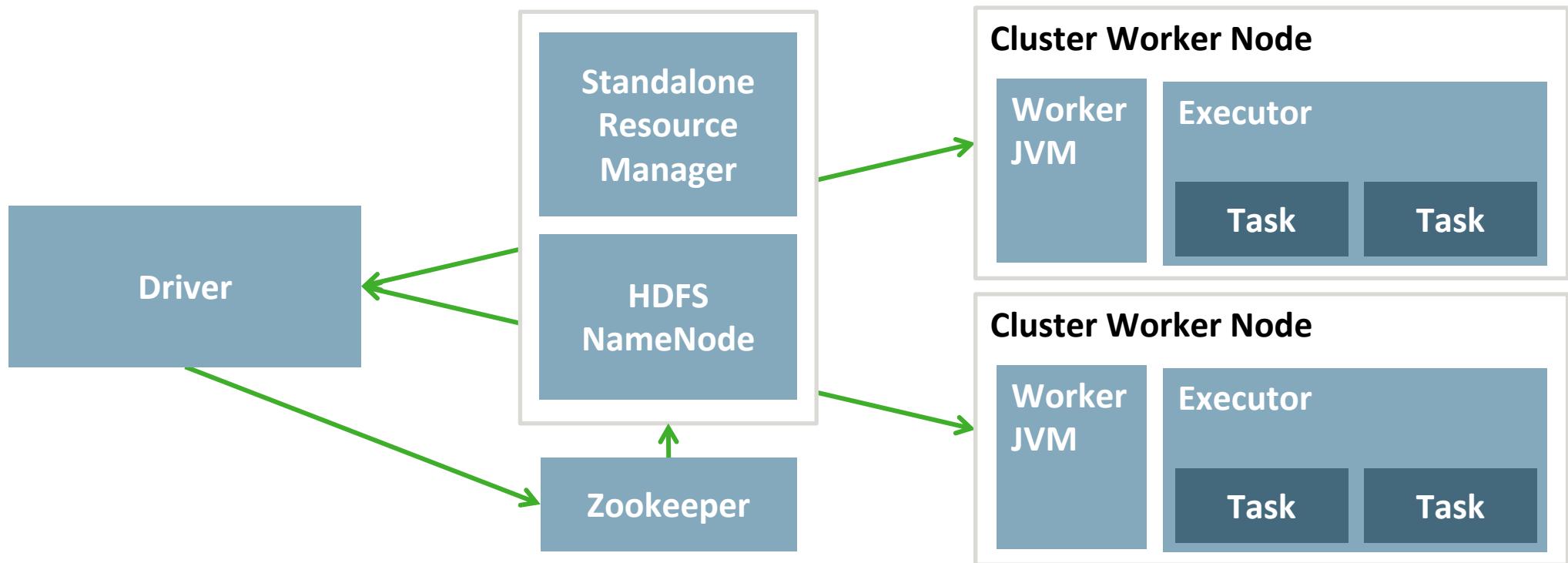
**Logging & Debugging**

# Cluster Manager Overview

- **Standalone**: Spark-only cluster manager
  - We've seen this before
  - To submit applications to a standalone cluster, use a master URL of form **spark://<master-node>:7077**
  - Can access managers Web UI at **http://<master-node>:8080**
- Apache **YARN**: Second generation Hadoop/MR cluster manager
  - Yet Another Resource Negotiator
  - Provides resource management and scheduling
  - It is decoupled from the data processing
- Apache **Mesos**: Cluster manager originating at UC Berkeley
  - By some of the same folks that created Spark
  - Provides dynamic resource allocation for multiple frameworks <sup>(1)</sup>

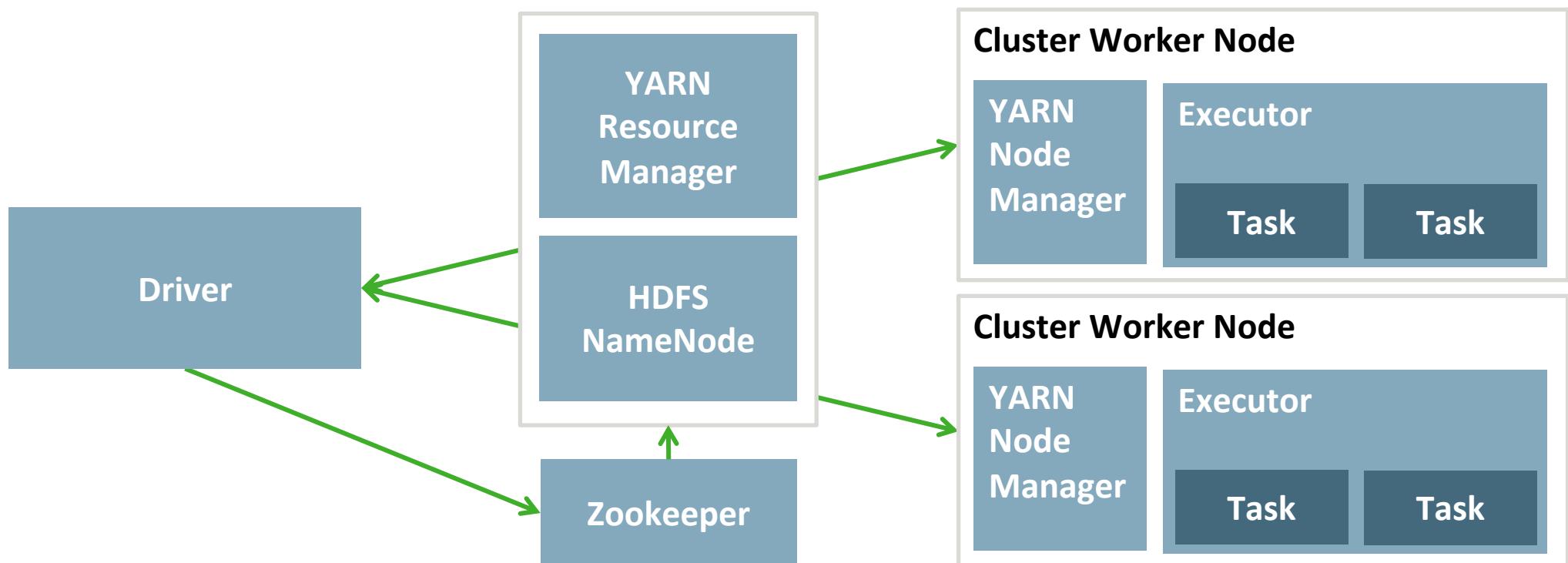
# Spark on Standalone Architecture

- Standalone manager is part of Spark
- Worker JVMs start Executors (also JVMs)
- Suitable for many production systems
- Master fault tolerance via Zookeeper



# Spark on YARN Architecture

- **YARN Node Manager** start Executors
- Master fault tolerance via Zookeeper



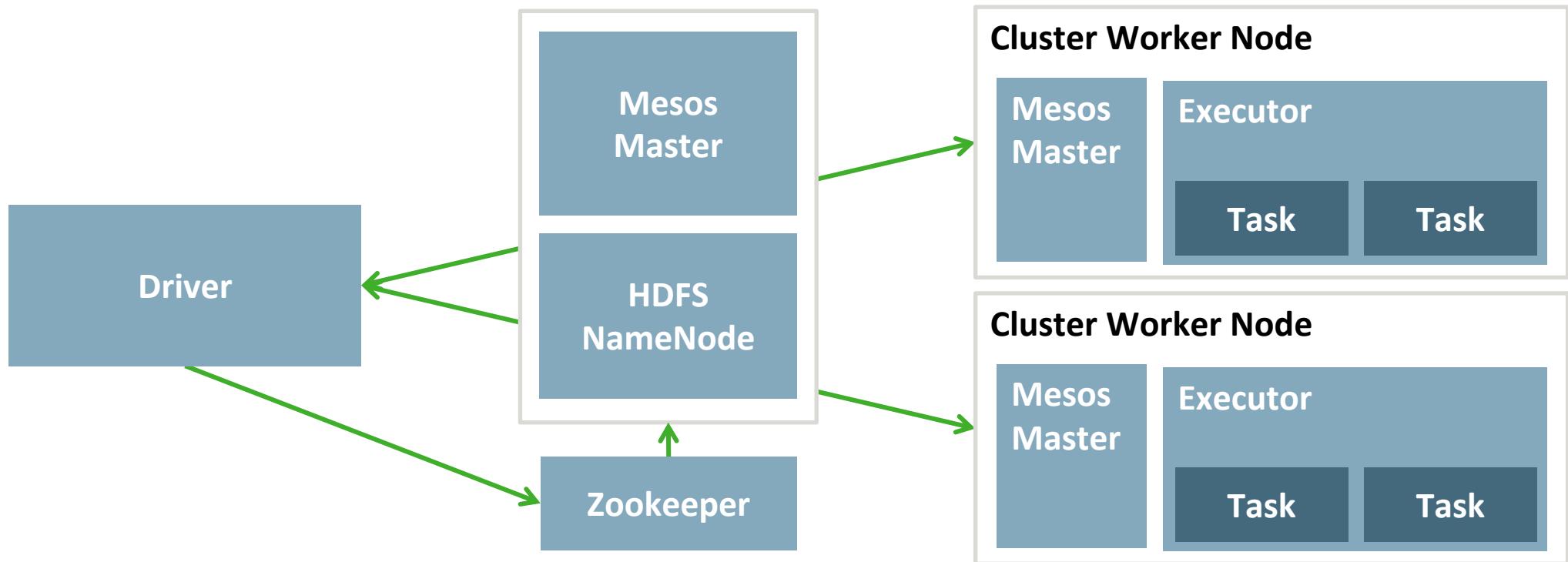
# YARN Usage

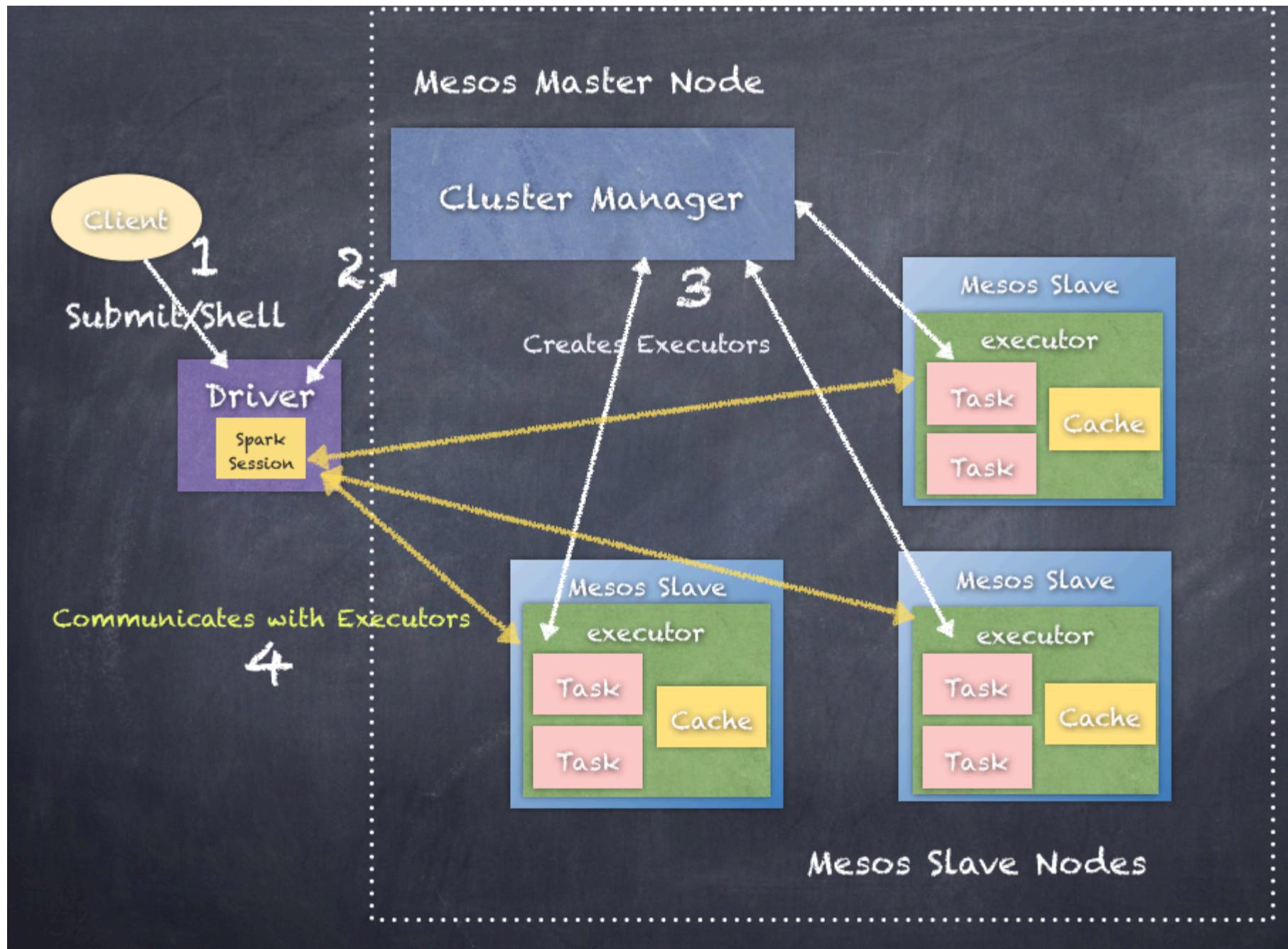
- Easy configuration:
  - Need to have a working Hadoop/YARN system
  - Set environment variables
    - `HADOOP_CONF_DIR` or `YARN_CONF_DIR` to the client-side config dir of the Hadoop cluster
    - The config files are used to connect to the YARN resource manager
  - Quite a few YARN-specific properties you can set (see the docs)<sup>(1)</sup>
  - e.g. `spark.driver.cores`: driver cores used in cluster mode
- Deploy modes:
  - `yarn-cluster`: Spark driver runs inside an app master process managed by YARN on the cluster
  - `yarn-client`: Driver runs in a client process — app master just used for resource allocation

```
$ spark-submit --master yarn-cluster \
--executor-memory 4G --class com.mycompany.MyApp \
target/scala-2.11/myapp.jar 1G.data
```

# Spark on Mesos Architecture

- Mesos slave starts Executors
- Master fault tolerance via Zookeeper





# Mesos Usage

- Uses standard Mesos install
- Requires Spark binary be available on worker nodes
  - e.g. on HDFS
- Configuration
  - Set environment variables in spark-env.sh
    - `export MESOS_NATIVE_LIBRARY=<path to libmesos.so>`. (or <path to libmesos.dylib> on Mac OS X)
    - `export SPARK_EXECUTOR_URI=<URL of spark binary file>`
  - Set `spark.executor.uri` to <URL of spark binary file>
- Deploy modes: Supports client mode only

```
$./bin/spark-submit --master mesos://<mesos-host>:5050 \
--conf spark.executor.uri=<URI-of-Spark-binary> \
--class com.es.spark.ProcessFiles \
target/scala-2.10/testapp.jar 1G.data
```

# Mesos Run Mode

- Spark can run in two modes over Mesos
- **fine-grained** (default): Each Spark task is a separate Mesos task
  - Multiple Spark instances and other frameworks share machines at a fine granularity
  - Executors scale number of CPUs up/down as they execute tasks
- **coarse-grained**: One long-running Spark task on each worker and schedule mini-tasks within it
  - Lower startup overhead, but reserves resources for entire life of app
  - Set **spark.mesos.coarse=true** to choose this mode

# Which Manager to Use

- Run Spark workers on HDFS nodes with any of the managers
  - For fast data access
- Development / New Deployments: Standalone
  - Easiest to setup and use
  - Pretty good if only running Spark
- If already running Hadoop 2: YARN
  - It will be pre-installed
- If fine-grained resource allocation is important: Mesos
  - Will scale down resource usage when job is less active

A close-up photograph of a young man with dark hair and glasses, looking intently at a computer screen. The screen is visible in the foreground, showing some blurred text or code. The background is slightly out of focus.

**Core API**

**Building and Running Applications**

**Application Lifecycle**

**Cluster Managers**

**Logging & Debugging**

# Web UI (SparkContext)

- We've seen it before — it is very useful for standalone apps
  - Can monitor what is happening — we illustrate stages below

The screenshot shows the Spark Web UI at `localhost:4040/stages/`. The top navigation bar includes links for Jobs, Stages (which is selected), Storage, Environment, and Executors. The main content area displays the following information:

**Spark Stages (for all jobs)**

Total Duration: 16 min  
Scheduling Mode: FIFO  
Active Stages: 0  
Completed Stages: 0  
Failed Stages: 0

**Active Stages (0)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
----------	-------------	-----------	----------	------------------------	-------

**Completed Stages (0)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input
----------	-------------	-----------	----------	------------------------	-------

**Failed Stages (0)**

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
----------	-------------	-----------	----------	------------------------	-------	--------

# Job Display

- Below, we illustrate a single call to collect()
  - Note how there is one completed job
- Clicking on the Job Description brings you to a detail page
  - See next slide

Active Jobs (0)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
Completed Jobs (1)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:19	2015/04/17 13:49:44	0.6 s	2/2	16/16
Failed Jobs (0)					
Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

# Viewing Stages

- Below, you can see the stages of the job
  - There are two — one which requires a shuffle
  - Clicking on the stage description will bring up a detail page for it

The screenshot shows the Spark 1.2.1 application UI. The top navigation bar includes the Spark logo, version 1.2.1, and tabs for Jobs, Stages, Storage, Environment, and Executors. To the right of the tabs, it says "Spark shell application UI". The main content area is titled "Details for Job 0". It displays the status as "SUCCEEDED" and "Completed Stages: 2". Below this, a table titled "Completed Stages (2)" lists the two stages:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	<a href="#">collect at &lt;console&gt;:19</a> +details	2015/04/17 13:49:44	88 ms	8/8				
0	<a href="#">map at &lt;console&gt;:14</a> +details	2015/04/17 13:49:44	0.4 s	8/8				1012.0 B

# Stage Detail

## Details for Stage 1

Total task time across all tasks: 0.5 s

▶ Show additional metrics

### Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	57 ms	58 ms	59 ms	62 ms	62 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

### Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Output	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	my-computer.home:53515	0.6 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

### Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Errors
0	8	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	59 ms		
1	9	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	58 ms		
3	11	0	SUCCESS	PROCESS_LOCAL	0 / my-computer.home	2015/04/17 13:49:44	62 ms		

# Master UI: <master-host>:8080

← → ⌂ my-computer.home:8080 ⭐ 📁 🔍

## Spark 1.2.1 Master at spark://my-computer.home:7077

URL: spark://my-computer.home:7077  
Workers: 1  
Cores: 8 Total, 8 Used  
Memory: 15.0 GB Total, 512.0 MB Used  
Applications: 1 Running, 0 Completed  
Drivers: 0 Running, 0 Completed  
Status: ALIVE

### Workers

ID	Address	State	Cores	Memory
worker-20150417133917-my-computer.home-53427	my-computer.home:53427	ALIVE	8 (8 Used)	15.0 GB (512.0 MB Used)

### Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150417134807-0000	Spark shell	8	512.0 MB	2015/04/17 13:48:07	yaakov	RUNNING	15 min

### Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----	------	-------	-----------------	----------------	------	-------	----------

# Master UI: Application Detail

- This is the detail page for the spark shell
  - Note how you can access stdout and stderr directly

The screenshot shows a web browser window displaying the Spark Application Detail UI. The URL in the address bar is `my-computer.home:8080/app/?appId=app-20150417134807-0000`. The page title is "Application: Spark shell". The UI displays the following details:

- ID:** app-20150417134807-0000
- Name:** Spark shell
- User:** yaakov
- Cores:** Unlimited (8 granted)
- Executor Memory:** 512.0 MB
- Submit Date:** Fri Apr 17 13:48:07 EDT 2015
- State:** RUNNING

A blue link labeled "Application Detail UI" is present. Below this, there is a section titled "Executor Summary" containing a table:

ExecutorID	Worker	Cores	Memory	State	Logs
0	<a href="#">worker-20150417133917-my-computer.home-53427</a>	8	512	LOADING	<a href="#">stdout</a> <a href="#">stderr</a>

# Logging

- Master logs appear in <spark>/logs
  - Log files are named after the user and computer name
  - e.g. **spark-student-org.apache.spark.deploy.master.Master-1-my-computer.home.out**
- Application logs appear in <spark>/work
  - In a subdirectory created when the app starts
  - For each app, there is a file for stdout and stderr logging
  - The logging output is also visible in the Master UI, as seen earlier
    - But not all output — e.g. INFO output is not visible there

# Customizing Logging

- Customize logging by creating the file *conf/log4j.properties*
  - The existing *conf/log4j.properties.template* can be copied
  - Below, we illustrate how to change the root logger level to WARN
  - This reduces some of the copious logging that Spark produces
  - This file will be detected and used automatically
  - You can also use the `--files` option to spark-submit to send this file along with your app jar
  - Make sure to distribute the log4j.properties file to all nodes

```
log4j.properties.template - INFO level to console
log4j.rootCategory=INFO, console
Remaining detail omitted ...
```

```
log4j.properties - WARN level to console
log4j.rootCategory=WARN, console
Remaining detail omitted ...
```

## Lab 7.2: Additional Capabilities

In this lab, we'll use some additional capabilities for working with standalone applications

# Review Questions

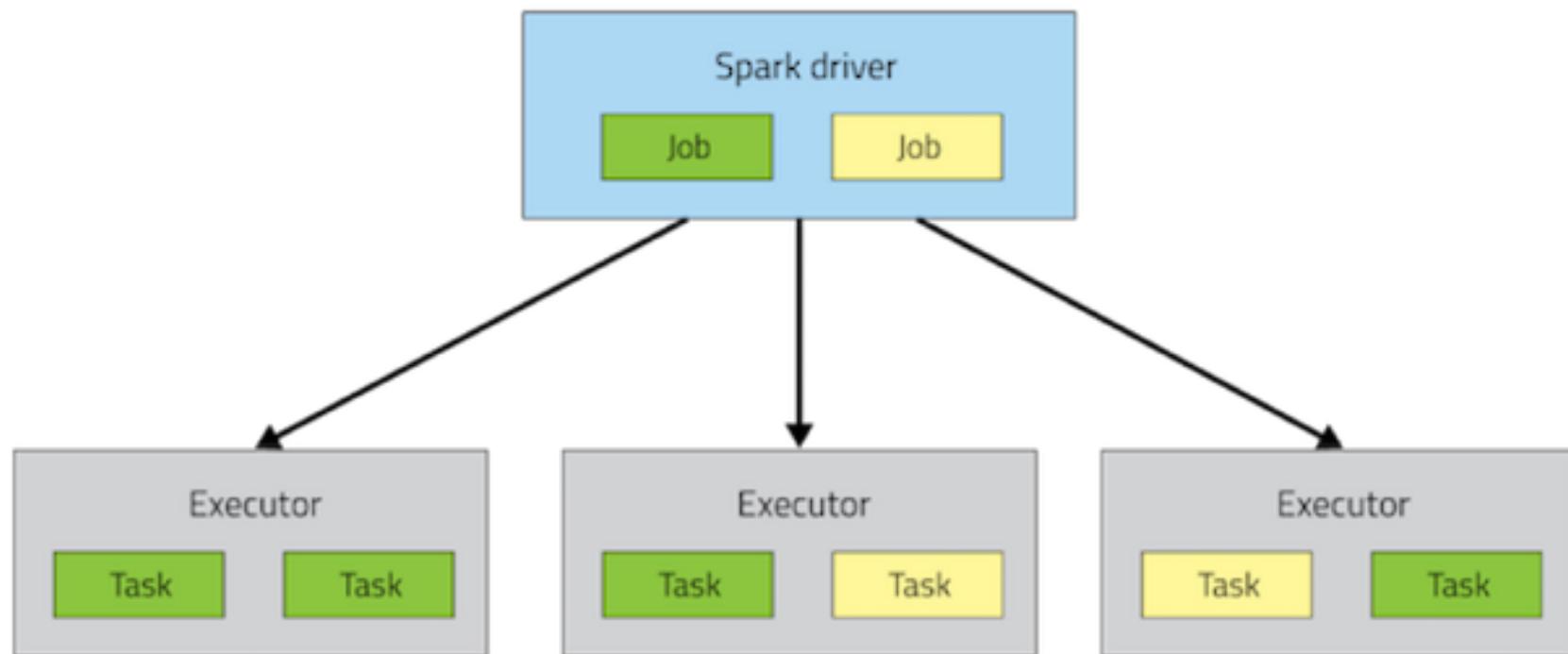
- How do you write a Spark application?
- How do you run a Spark application?
- What is a cluster manager, and which ones are supported by Spark

# Lesson Summary

- There are two core types used in writing a Spark app
  - **SparkSession**: Handle on cluster operations
  - **SparkSession.Builder**: Factory class for configuring SparkSession
- Spark applications are generally run using the spark-submit command
- Cluster managers manage the distributed compute cluster.
  - There are three supported cluster managers.
  - **Standalone**: Comes with Spark, and is a spark-only solution.
  - **YARN**: Hadoop 2 cluster manager
  - **Mesos**: Cluster manager originating at Berkeley
    - Can provide dynamic resource allocation for efficient use of resources

# How to Tune Spark Jobs

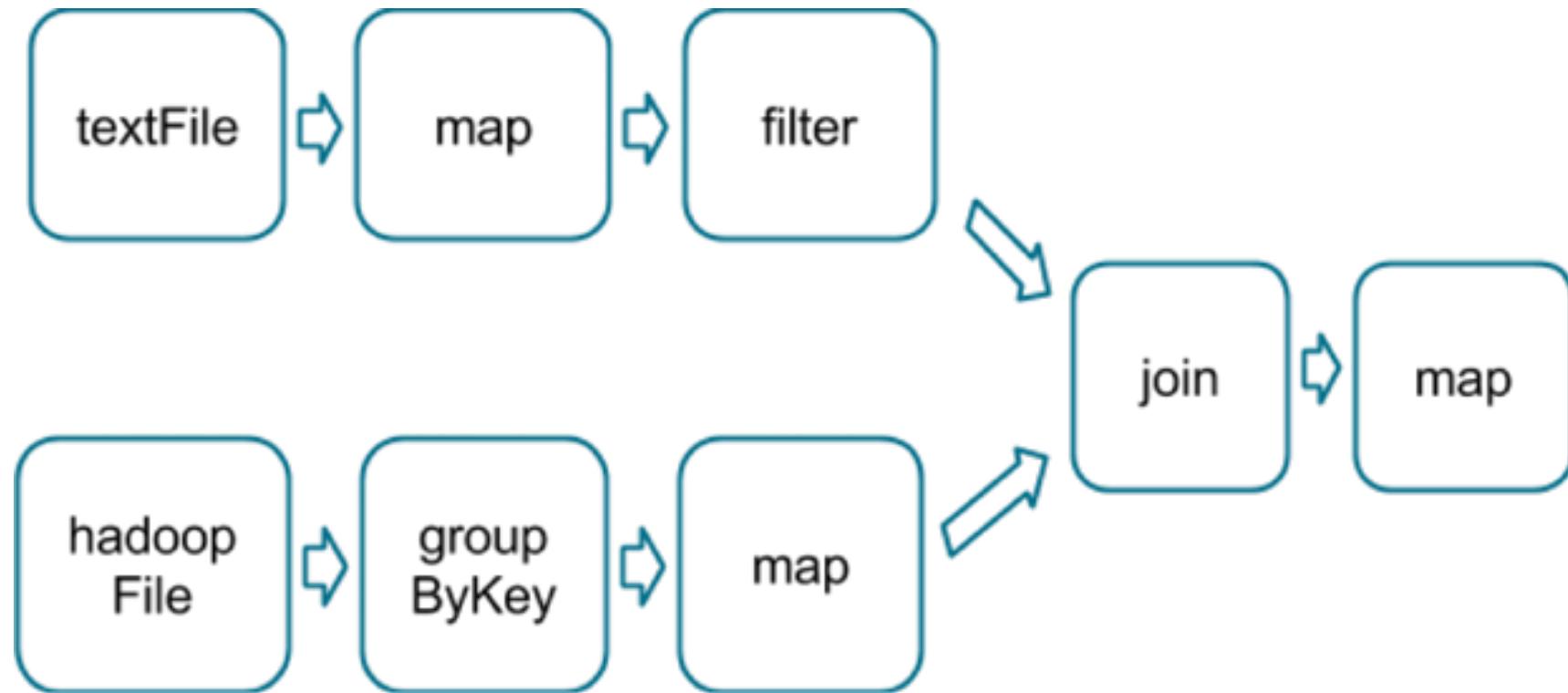
## How Spark Executes Your Program



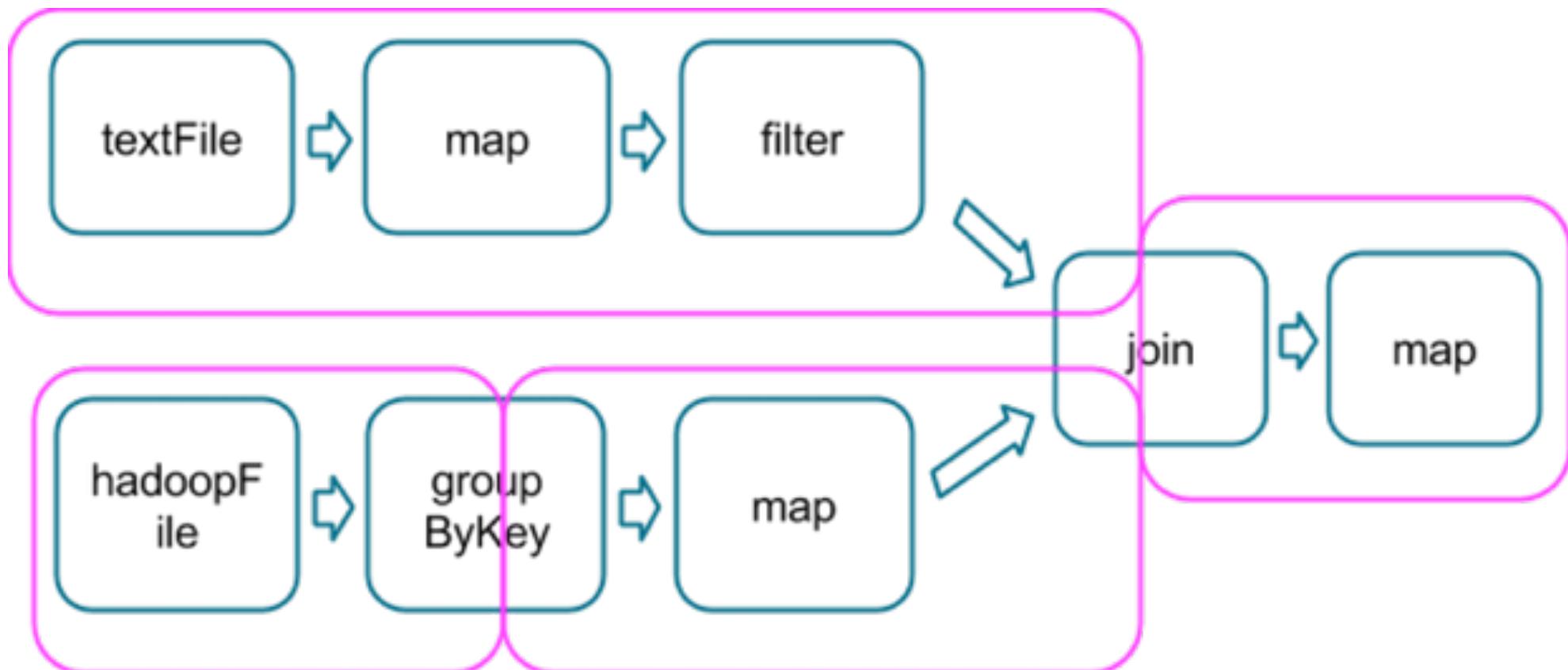
# Stages- Narrow and Wide Transformations

- val tokenized = sc.textFile(args(0)).flatMap(\_.split(' '))
- val wordCounts = tokenized.map((\_, 1)).reduceByKey(\_ + \_)
- val filtered = wordCounts.filter(\_.value >= 1000)
- val charCounts = filtered.flatMap(\_.value.toCharArray).map((\_, 1)).reduceByKey(\_ + \_)
- charCounts.collect()

# Transformation Graph



# Resulting stage graph



# Picking the Right Operators

## Avoid groupByKey

When performing an associative reductive operation. For example,  
`rdd.groupByKey().mapValues(_.sum)`

will produce the same results as

`rdd.reduceByKey(_ + _).`

# Picking the Right Operators

**Avoid reduceByKey When the input and output value types are different**

For example, consider writing a transformation that finds all the unique strings corresponding to each key. One way would be to use map to transform each element into a Set and then combine the Sets with reduceByKey

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2))
```

```
.reduceByKey(_ ++ _)
```

# Picking the Right Operators

**Avoid the flatMap-join-groupBy pattern.**

When two datasets are already grouped by key and you want to join them and keep them grouped, you can just use cogroup. That avoids all the overhead associated with unpacking and repacking the groups.

# When Shuffles Don't Happen

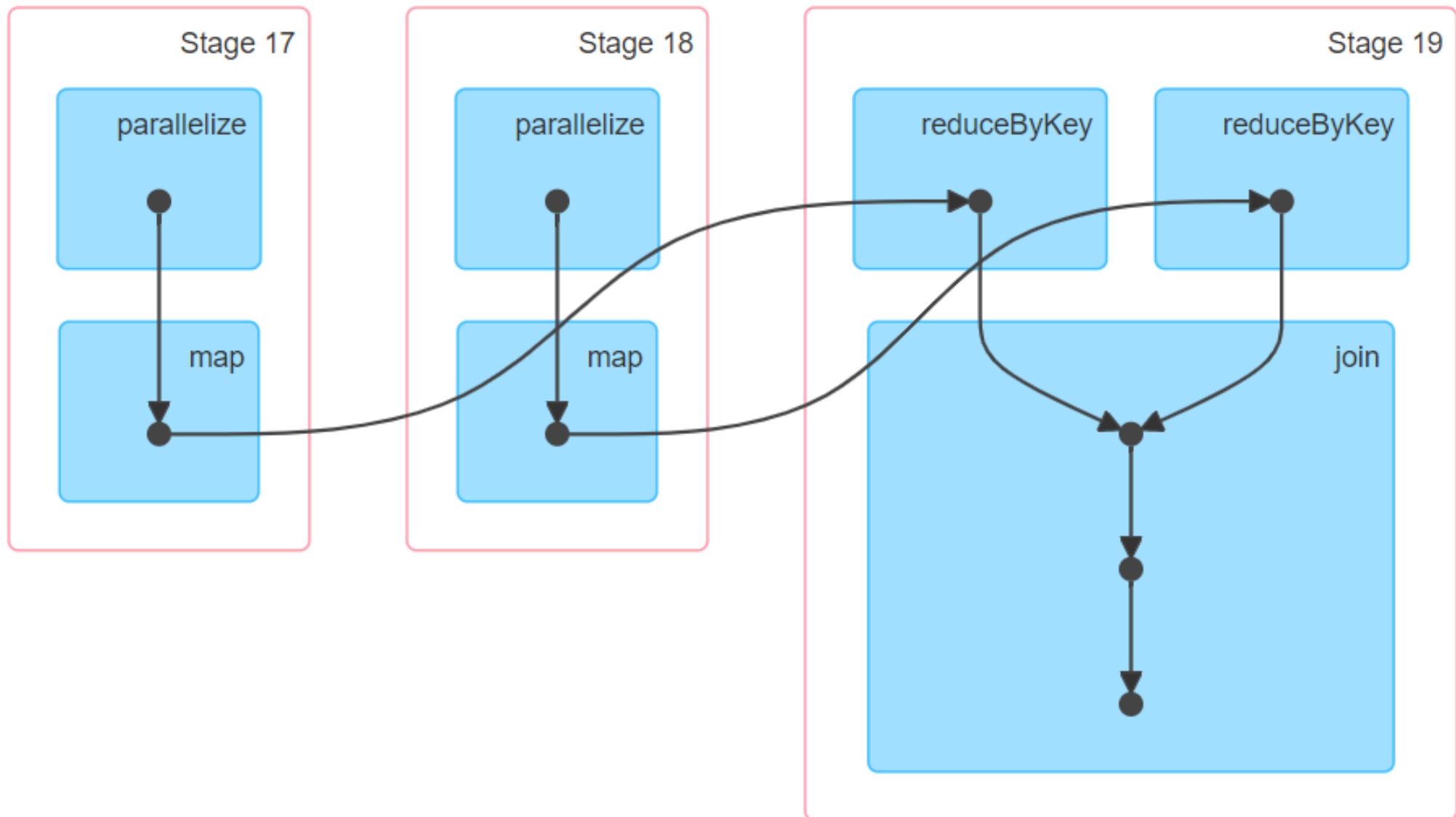
It's also useful to be aware of the cases in which the above transformations will *not* result in shuffles.

Spark knows to avoid a shuffle when a previous transformation has already partitioned the data according to the same partitioner.

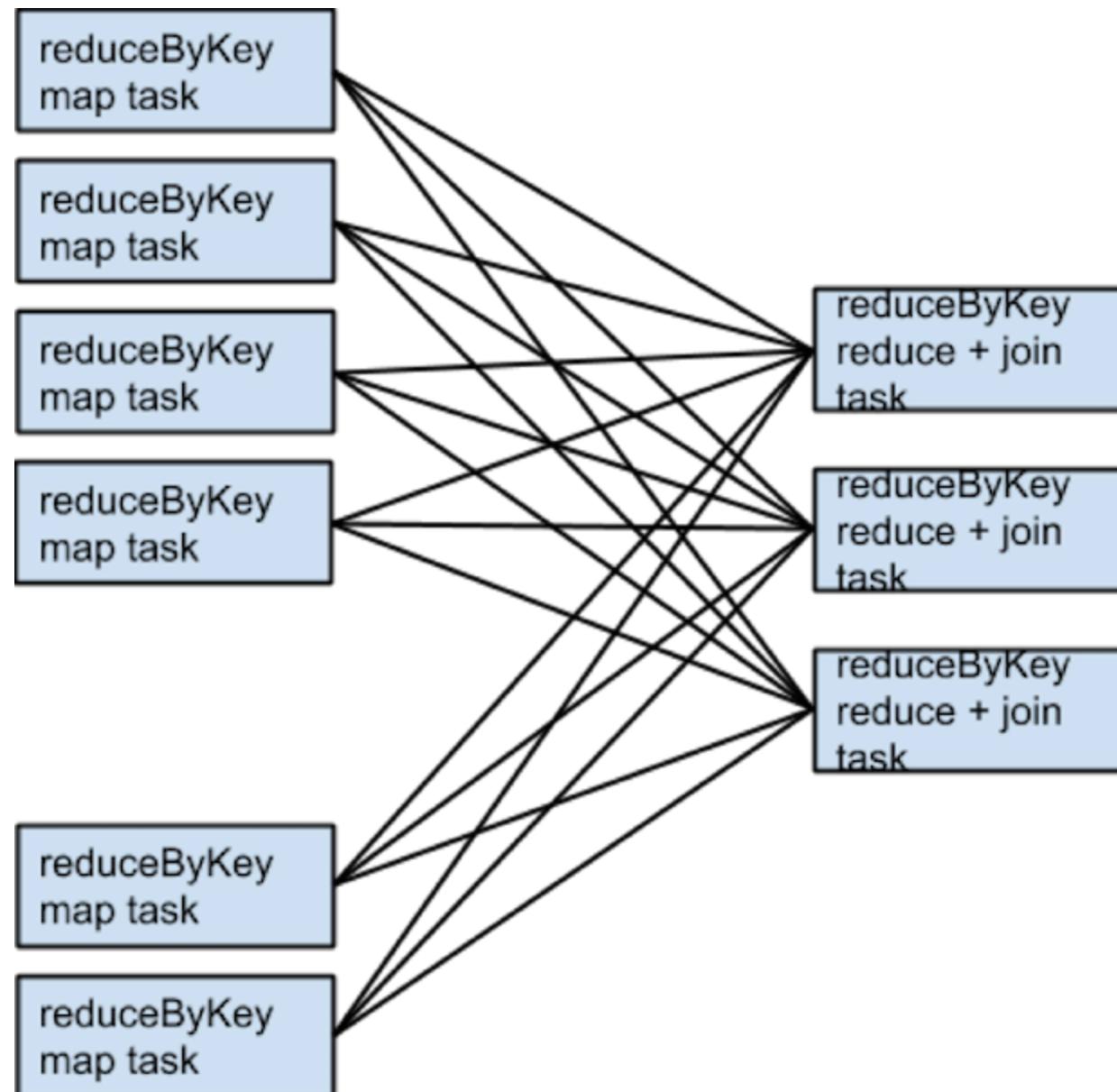
# How many stages?

- `rdd1 = someRdd.reduceByKey(...)`
- `rdd2 = someOtherRdd.reduceByKey(...)`
- `rdd3 = rdd1.join(rdd2)`

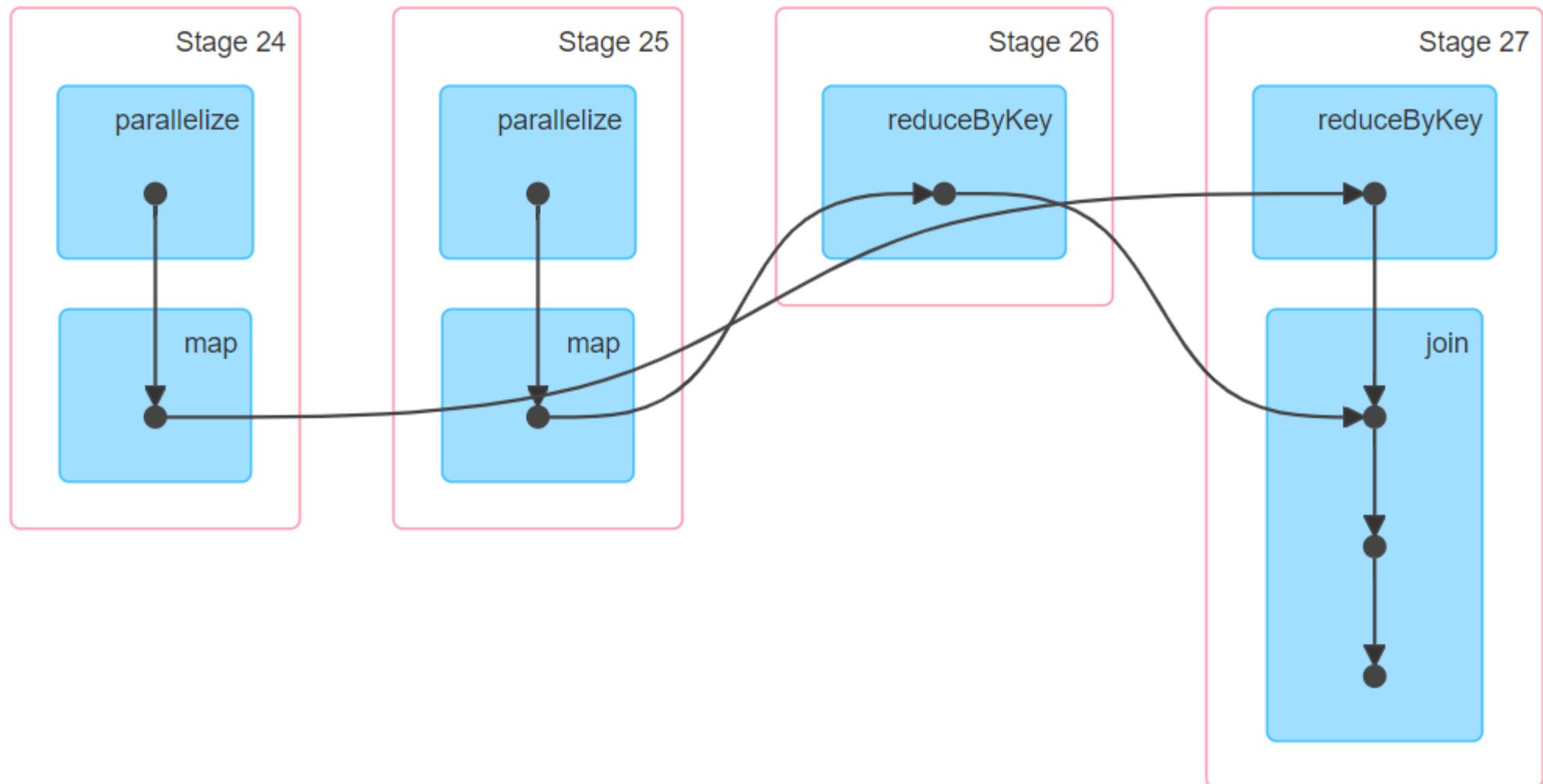
# DAG Visualization



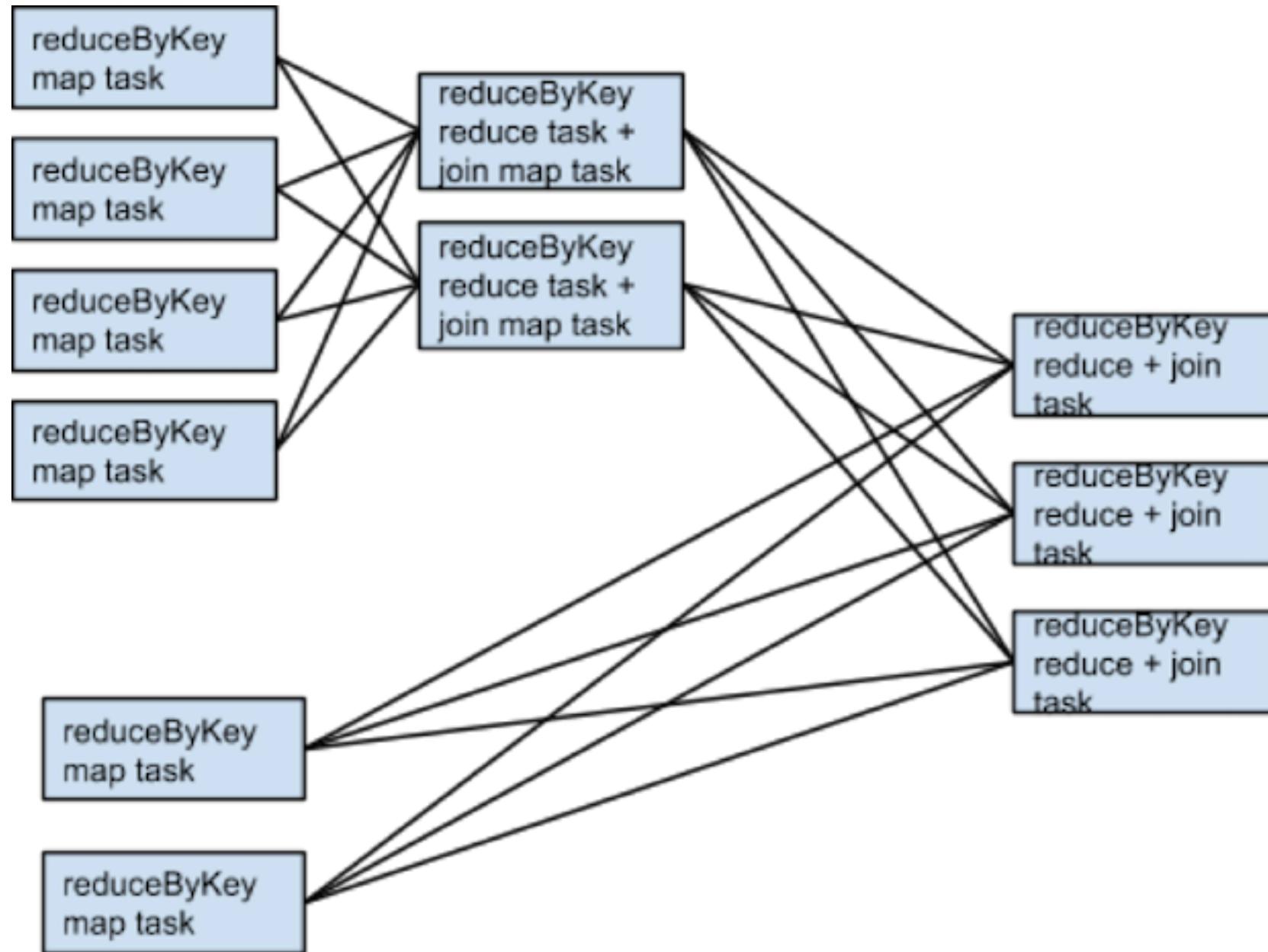
# Transformation with same number of partition



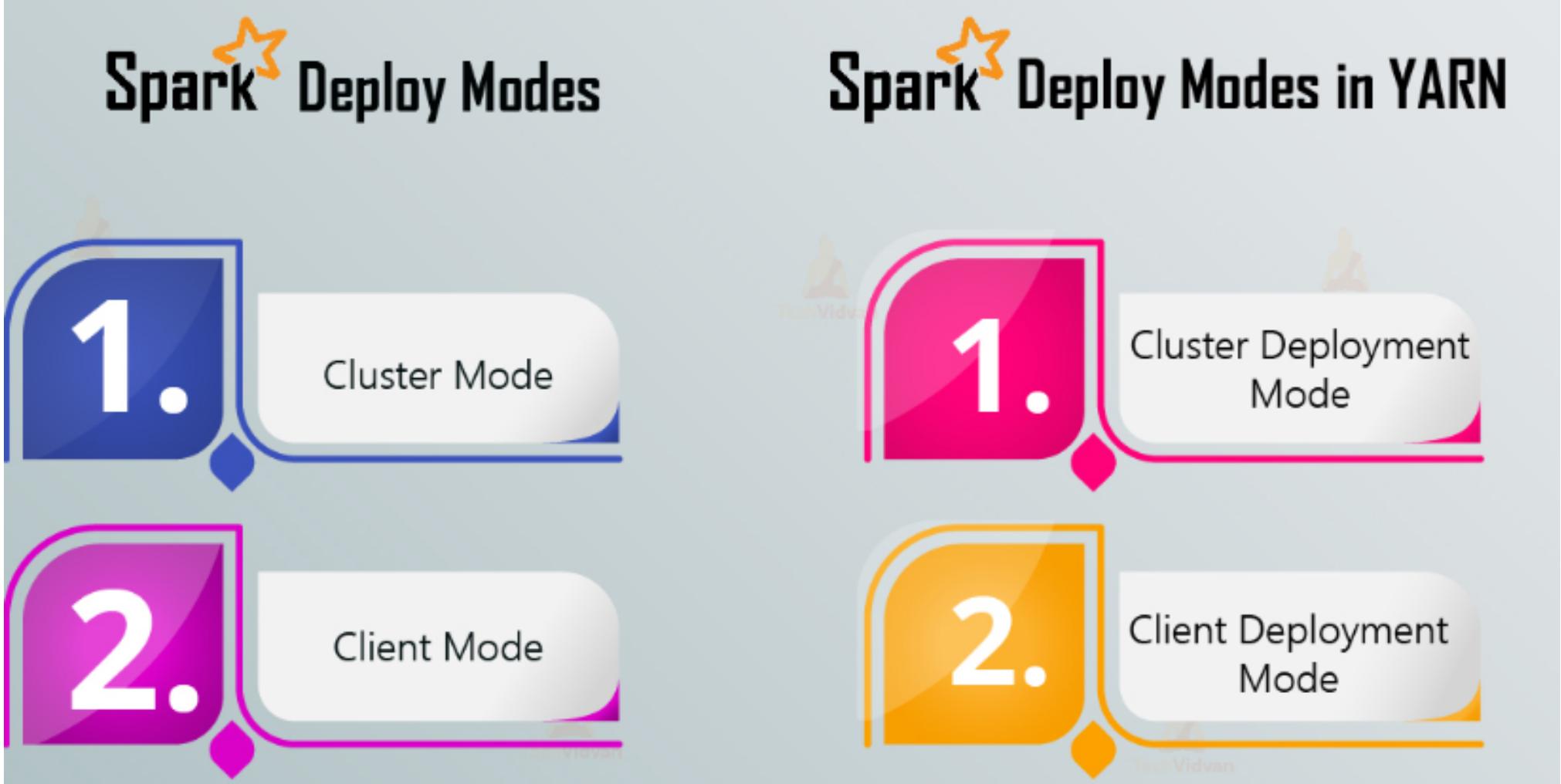
# DAG Visualization



# Same transformations, same inputs, different number of partitions:



# Deployment Modes



A photograph of a man with a beard and short brown hair, wearing a maroon long-sleeved shirt and blue jeans. He is standing in what appears to be a large industrial or warehouse space, with pipes and structural elements visible in the background. He is gesturing with his hands while speaking. A green diagonal bar runs from the bottom right corner across the slide.

# Session 8: Spark Streaming Overview

- Introduction to Streaming
- Spark Streaming (1.0+)
- [Optional] Spark Streaming in Depth (1.0+)
- Spark Structured Streaming (2.0+)
- Consuming Kafka Data

# Session Objectives

- Understand streaming architectures and the needs they fill
- Be familiar with both Spark Streaming (1.0+) and Spark Structured Streaming (Spark 2.0+)
- Use Spark's streaming capabilities in conjunction with other Spark capabilities



## Introduction to Streaming

Spark Streaming (1.0+)

[Optional] Spark Streaming in Depth (1.0+)

Spark Structured Streaming (2.0+)

Consuming Kafka Data

# Big Data Evolution

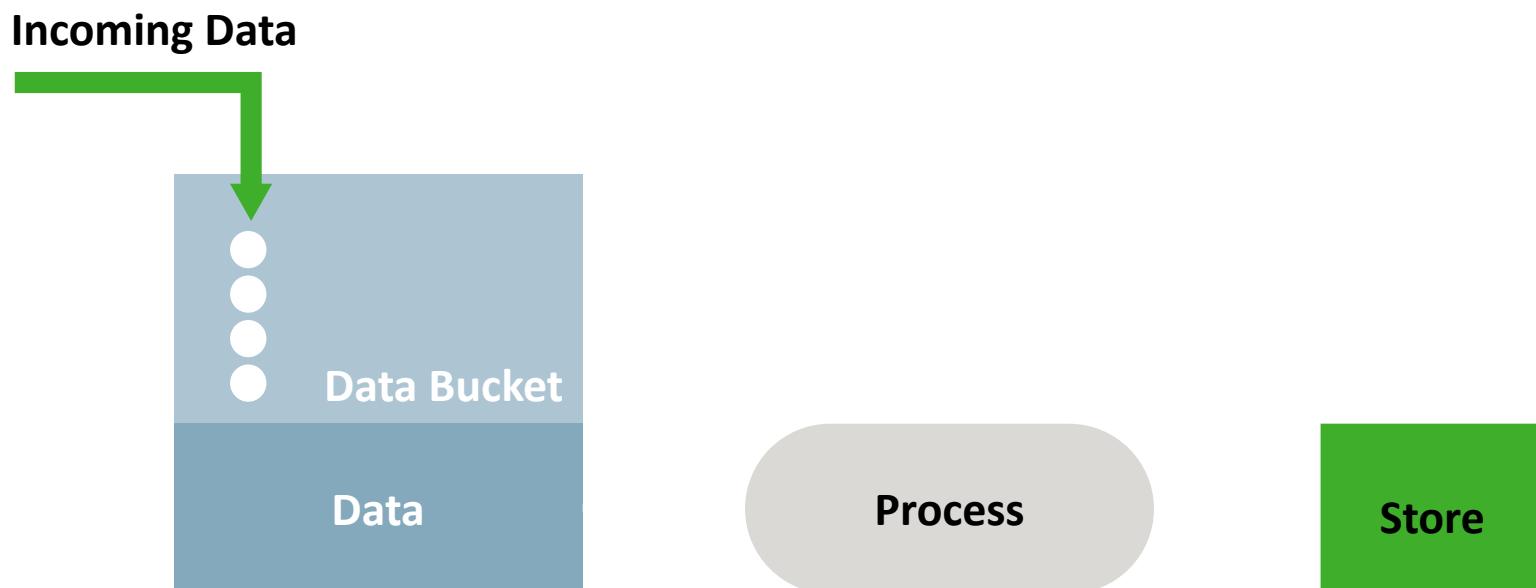
- V1: Back in the day
  - **Decision times:** Batch oriented (hours/days)
  - **Use Cases:** Reporting, modeling ETL
- Now: All the above **plus** need to process **streams of data**
  - **Decision times:** (Near) real time (millisecond, seconds)
  - **Use cases:** Alerts (medical/security, Fraud detection ...)
    - Connected Devices / Internet of Things (IoT)
    - And so on
  - Need for faster processing / analytics

# Streaming Data Overview

- **Streaming data** is generated continuously
  - Often by many data sources (e.g. IoT)
  - Generally with smaller payload sizes
- Processing needs on streaming data include:
  - **Sequential** and **incremental** processing
  - **Record-by-record** and **sliding window** processing
  - **Transformations** on data, including aggregations
    - e.g. filtering, averaging, counting
- Streaming data/processing is a core requirement
  - Many technologies support this (e.g. Storm, Flink, etc.)
  - We'll focus on Spark's streaming capabilities

# High-level Streaming Architecture

- **Data Bucket**: Captures/Buffers incoming data
  - Choices: **Kafka**, MQ, Amazon Kinesis
- **Process**: Low latency processing
  - Choices: **Spark**, Storm, Flink, ...
- **Store**: Store data — often in NoSQL store
  - NoSQL Storage: HBase, Cassandra ..

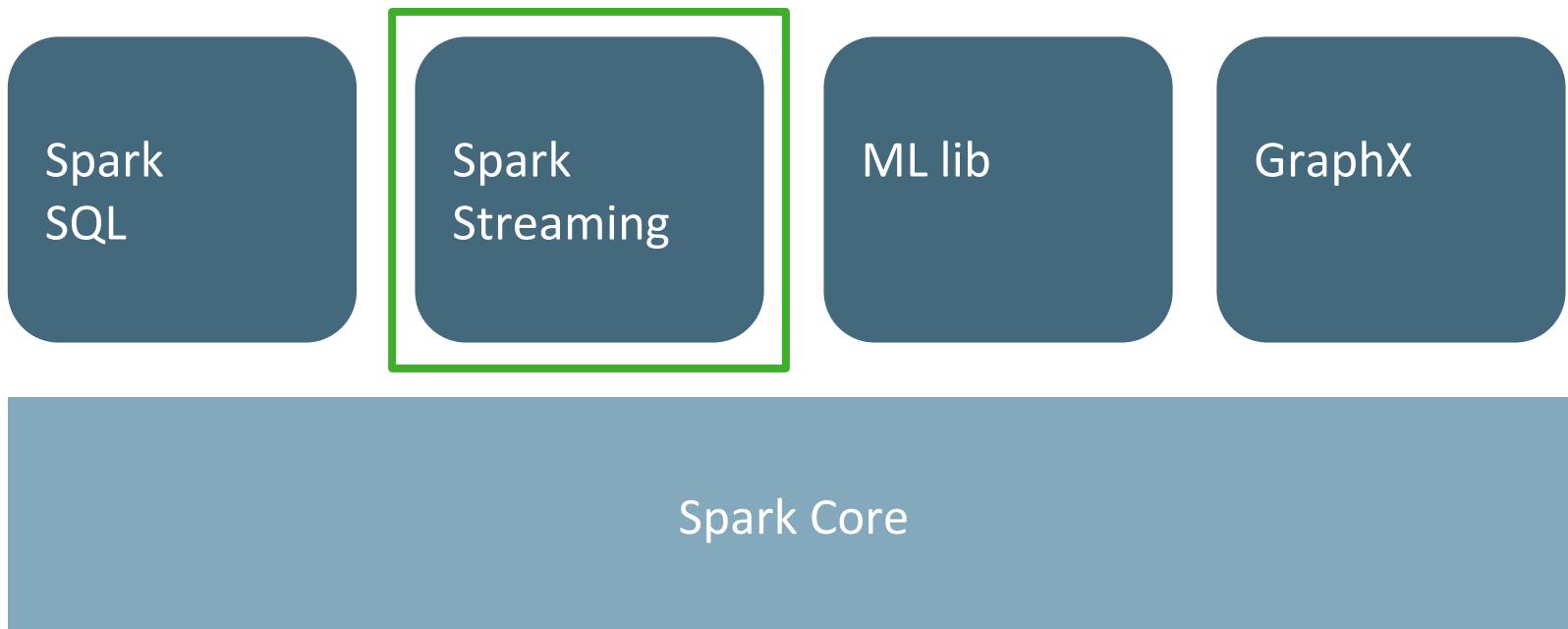


# Spark Streaming — Two Choices

- **Spark Streaming:** Released early (0.7+)
  - RDD based, complex API
  - New effort going to Structured Streaming
  - This will continue to be supported
- **Spark Structured Streaming: Alpha** in Spark 2.1
  - DataFrame based, improved architecture
  - Simpler API (Additions to DataFrame)
  - Direction of future work, but not fully mature now
- Both benefit from underlying Spark capabilities
  - Can integrate with transformations, graphing, ML
  - High throughput, scaling, fault tolerance
  - We'll provide overviews of both

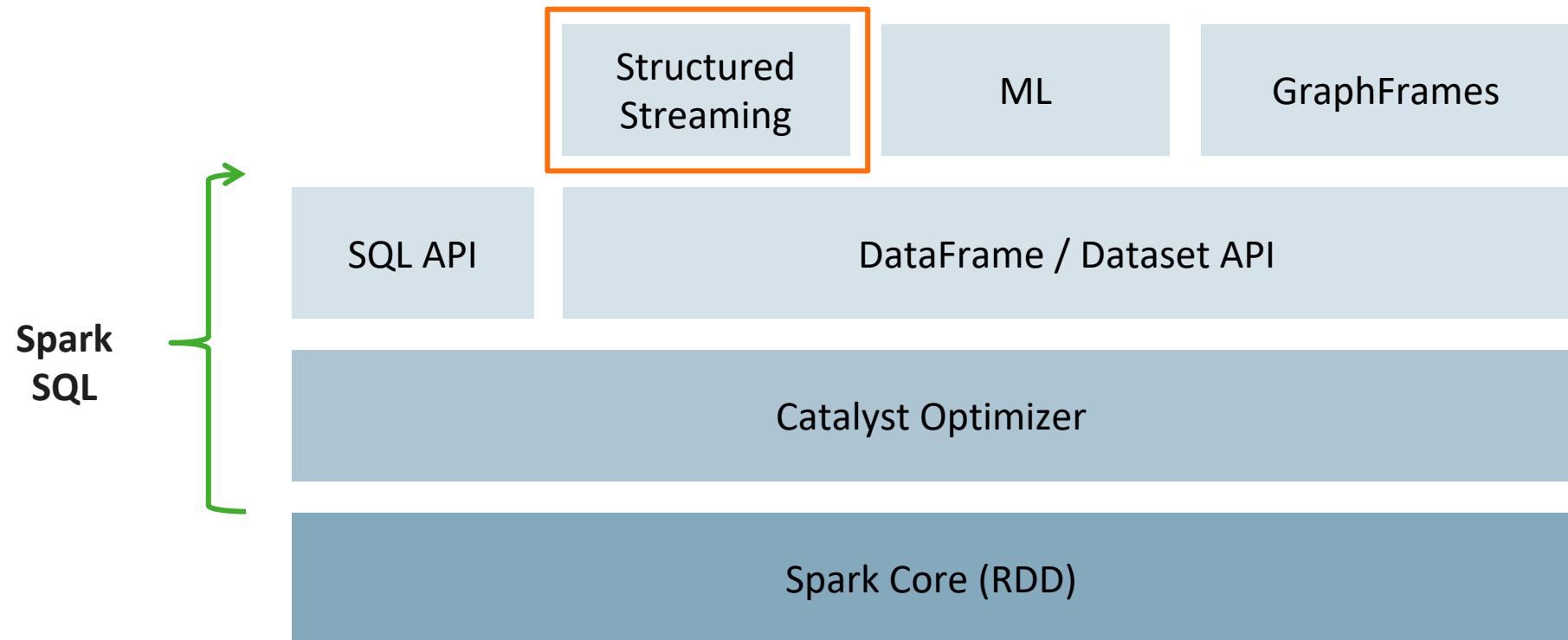
# Spark Streaming's Place in Spark

- Built on top of **Spark Core**
  - RDD-based



# Structured Streaming's Place in Spark

- Builds on top of Spark SQL
  - DataFrame based
  - Benefits from all improvements of Spark SQL



# Streaming Systems Feature Comparison

Feature	Storm	Spark Streaming	Spark Structured Streaming	Flink
Processing Model	Event based by default	Micro Batch	Micro Batch	Event based + Micro Batch based
Windowing operations	Supported by Trident	Yes	Yes	Yes
Latency	Milliseconds	Seconds		Milliseconds
Internal Processing	at least once	exactly once	exactly once	exactly once
Interactive Queries	NO	YES	YES	NO
Join with static data	NO	YES	YES	NO

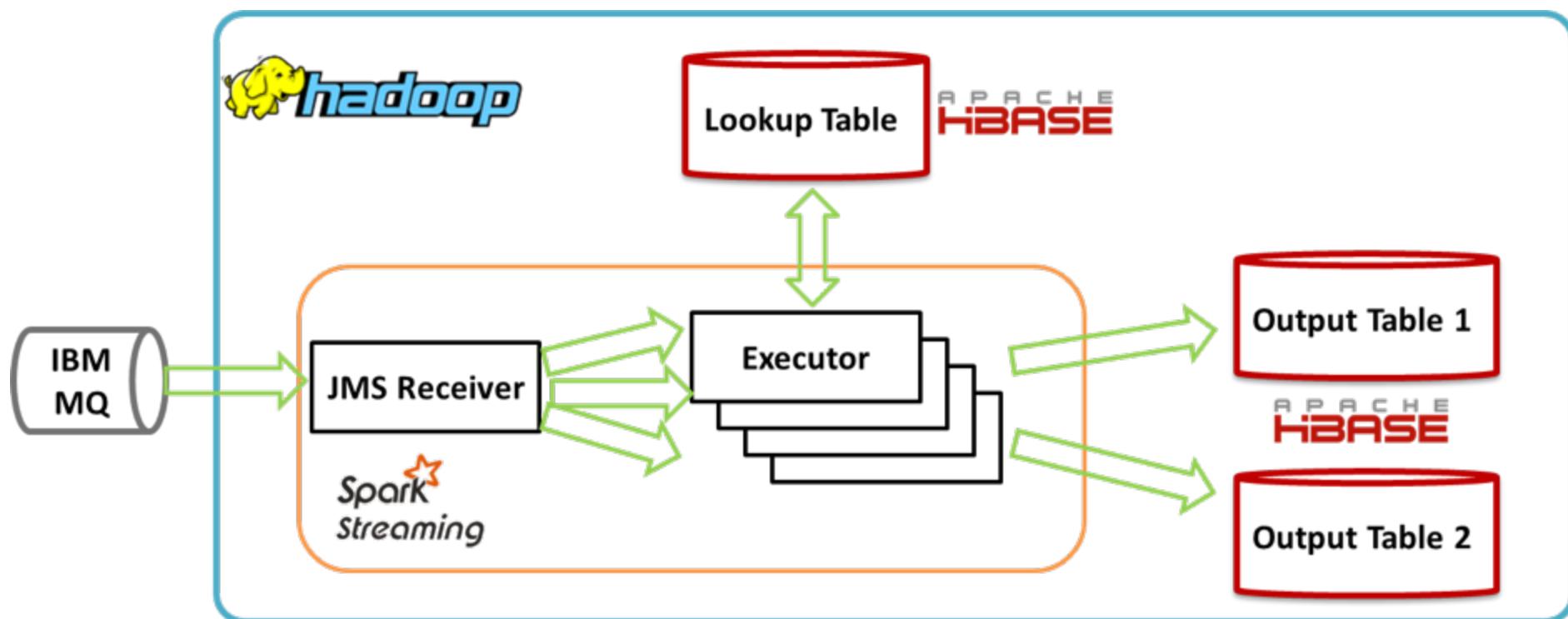


**Introduction to Streaming**  
→ **Spark Streaming (1.0+)**  
**[Optional] Spark Streaming in Depth (1.0+)**  
**Spark Structured Streaming (2.0+)**  
**Consuming Kafka Data**

# Key Concepts

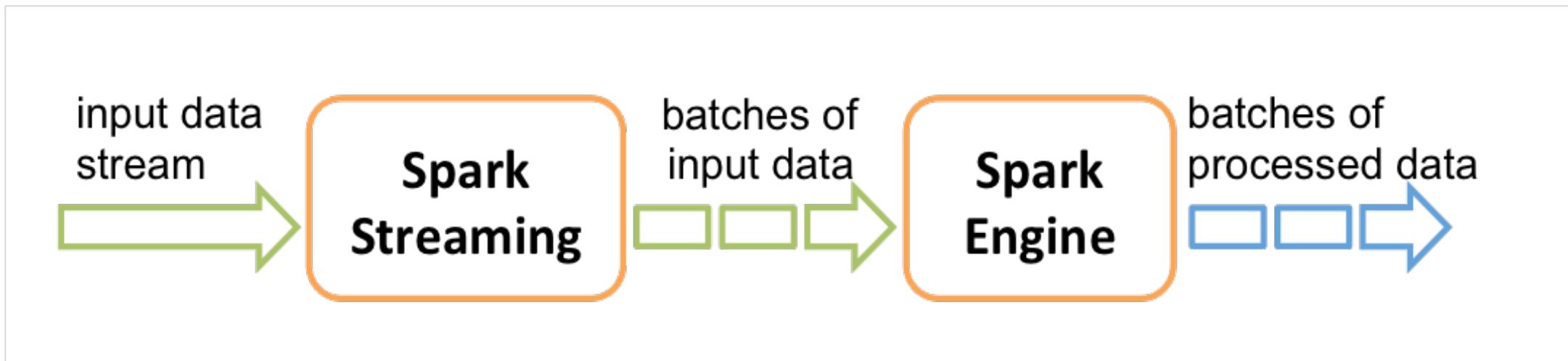
- Transforms raw streaming data to processed data
  - Low latency and fault tolerance
- **DStream**: Sequence of RDDs representing input data
- **Transformations**: Modify a DStream RDD into another RDD
  - Provides Stateless and Stateful transformations
- **Output Operations**: Send to external entity
  - Save to storage
  - Process the batch in some other way

# Spark Streaming



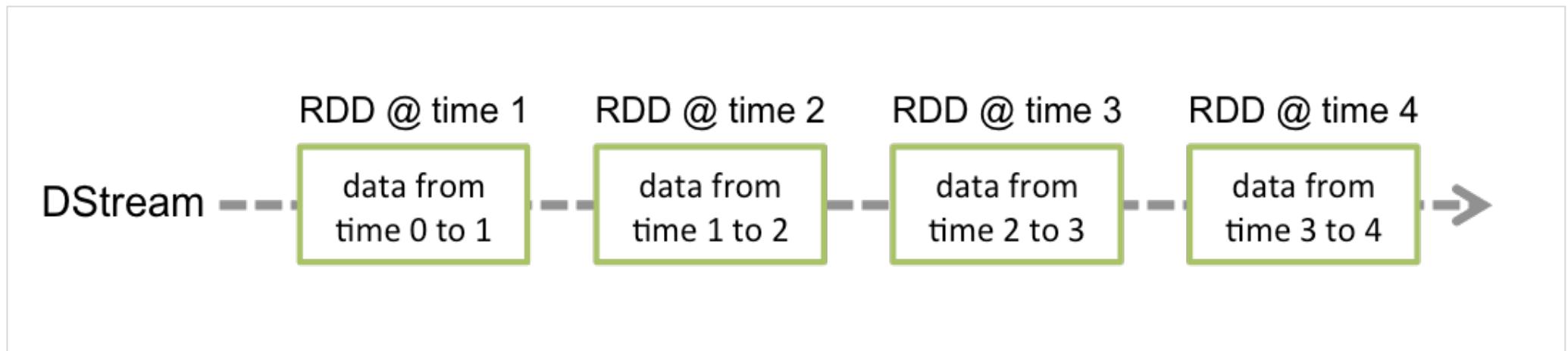
# How Does it Work?

- Structures streaming computation as a series of **small, stateless, deterministic batch jobs** (DStreams)
  - Divides live stream into batches of some interval (seconds)
  - Each batch becomes an RDD, processed via RDD operations
  - Processed results are returned in batches

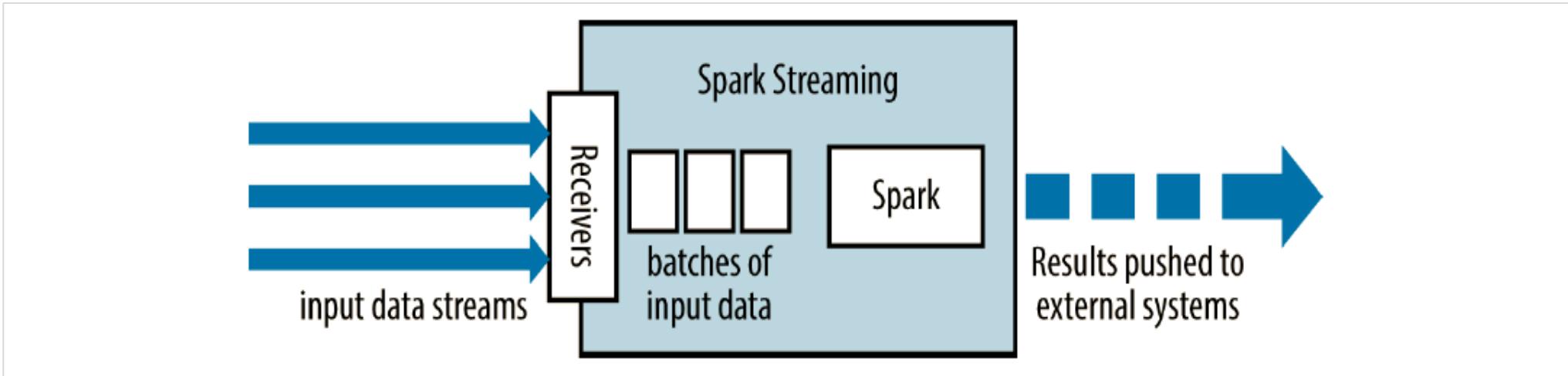


# Discretized Streams (DStreams)

- **DStream**: Sequence of RDDs representing data
  - Data arrives over time (streams in)
    - From many popular input sources — Flume, Kafka, HDFS ...
  - Data is broken down into micro-batches (RDDs), then processed
- DStream Operation Types:
  - **Transformation**: Modify data (resulting in another DStream)
    - Supports most standard RDD operations
    - Provides stateful operations — e.g. windowed operations
  - **Output**: Send to an external device

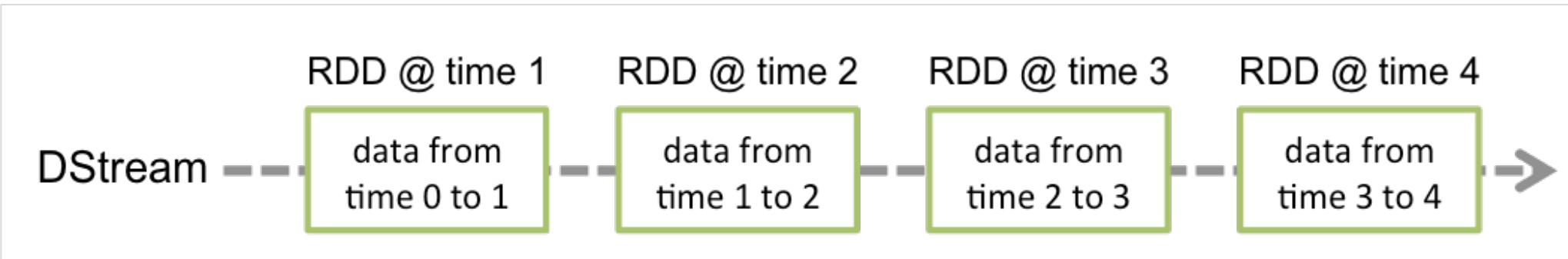


# Architecture – Receivers



- Each input DStream is associated with a **receiver**
  - Receivers are tasks running in the application's executors.
  - They collect the data and convert it into RDDs
  - Receivers are 'long running tasks'
- Streaming data can be received over the network
  - Kafka, Flume, sockets, etc.
- Can also create a stream by periodically loading data from external storage (e.g. HDFS)

# Architecture — Batching



- DStream data from input sources is bound into batches
  - Based on (configurable) **batch interval**
  - Data within a given time interval is added to the batch
    - Interval usually from 0.5 sec. to many seconds (based on your needs)
  - Batch is closed at end of interval
  - Each batch becomes a discrete RDD
- Processing is distributed across machines, like normal RDDs
  - Streaming can continue as batch processing occurs (done in parallel)
  - Can integrate with non-DStream RDDs (e.g. via a join)

# DStream Processing

- Each input batch results in a new RDD
  - As the batch interval passes, new RDDs are continuously generated
  - Containing the streamed data captured in that interval
- DStreams can be transformed
  - Accomplished by transforming all the RDDs in the DStream
- So a DStream periodically generates RDDs by either:
  - **Batching live data** into an initial RDD
  - **Transforming an RDD** generated by a parent DStream

# DStream Transformation Types

- **Stateless**: Transformation on a batch doesn't depend on the previous batch
  - Common RDD transformations such as `map()`, `filter()`, `countByValue()`, `reduceByKey()`, etc.
  - e.g. filter a particular word/string from a stream of text
- **Stateful**: Uses data/results from previous batches to compute the current batch
  - Based on **sliding windows** and tracking state across time
  - Operations include `window()` and `countByValueAndWindow()`
  - Example use cases: Calculate % of price increase per trade for a given stock on NASDAQ

# Sample Program — Overview

- We will illustrate a simple program that
  - Sets up a streaming receiver that reads from a socket
    - Uses a batch duration of 5 sec
    - Filters out all input lines, except those containing string "Scala"
  - Writes the processed input to the console
- Our input data is created via the **nc** (netcat) program
  - Typing in the nc console window sends typed input over the network
  - Configure host and port on nc for clients to connect

# Sample Code (1 of 3 — Initialization)

- Import required types
- Set **batchDuration**: Time interval to process new data
- Create **StreamingContext**: Main entry point for streaming
  - Provides methods to create DStreams from various input sources

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds

// Code excerpt showing streaming code only

val conf = new SparkConf()
 .setMaster("local[2]").setAppName("StreamingExample")
val batchDuration = Seconds(5)
val ssc = new StreamingContext(conf, batchDuration)
// Now you're ready to process ...
```

# Sample Code (2 of 3 — Set up Input)

- Set up input source
  - Here, we receive streaming text over the network (a socket)
- Create a DStream for the input source
  - `socketTextStream()` creates a network DStream reading text
- Transform using RDDs
  - In this case, filter for lines containing "Scala"
  - Then print them

```
val port = 9999 // Would use appropriate port for your app
val lines : ReceiverInputDStream[String] =
 ssc.socketTextStream("localhost", port)
val scalaLines = lines.filter(_.contains("Scala"))
scalaLines.print()
```

# Sample Code (3 of 3 — Process)

- Three StreamingContext methods to control streaming:
  - `start()`: Start the computation (should only be done once)
    - As shown below
  - `stop()`: Stop the computation manually
  - `awaitTermination()`: Wait for the computation to stop
    - As shown below — blocks waiting for termination
- Run this like any other Spark program
  - e.g. using spark-submit, or the shell

```
ssc.start() // Start it up !
ssc.awaitTermination() // Run until processing terminates
```

# Complete Program

```
package com.mycompany.streaming
import org.apache.spark._
import org.apache.spark.streaming._

object StreamingExample {
 def main(args: Array[String]) {
 // Create streaming context
 val ssc = new StreamingContext("local[2]",
 "StreamingExample", Seconds(5))
 // Create DStream from socket stream source
 val lines = ssc.socketTextStream("localhost", 9999)
 // Filter (creates new DStream)
 val scalaLines = lines.filter(_.contains("Scala"))
 scalaLines.print() // Output

 ssc.start()
 ssc.awaitTermination() // Standalone programs
 }
}
```

# Results

- At left we emulated a network source with netcat<sup>(1)</sup>
  - We typed a bit, waited, typed a bit more
- At right is the terminal window where we ran the program
  - You can see the output came in several batches (our batch interval was 5 sec = 5,000 ms)

```
$ nc -lk 9999
Scala 1
English 2
Java 3
Burp 4
Scala 5
Buzz
6
French 7
Scala 8
```

```
[info] Running
com.mycompany.streaming.StreamingExample

Time: 1429590630000 ms

Time: 1429590640000 ms

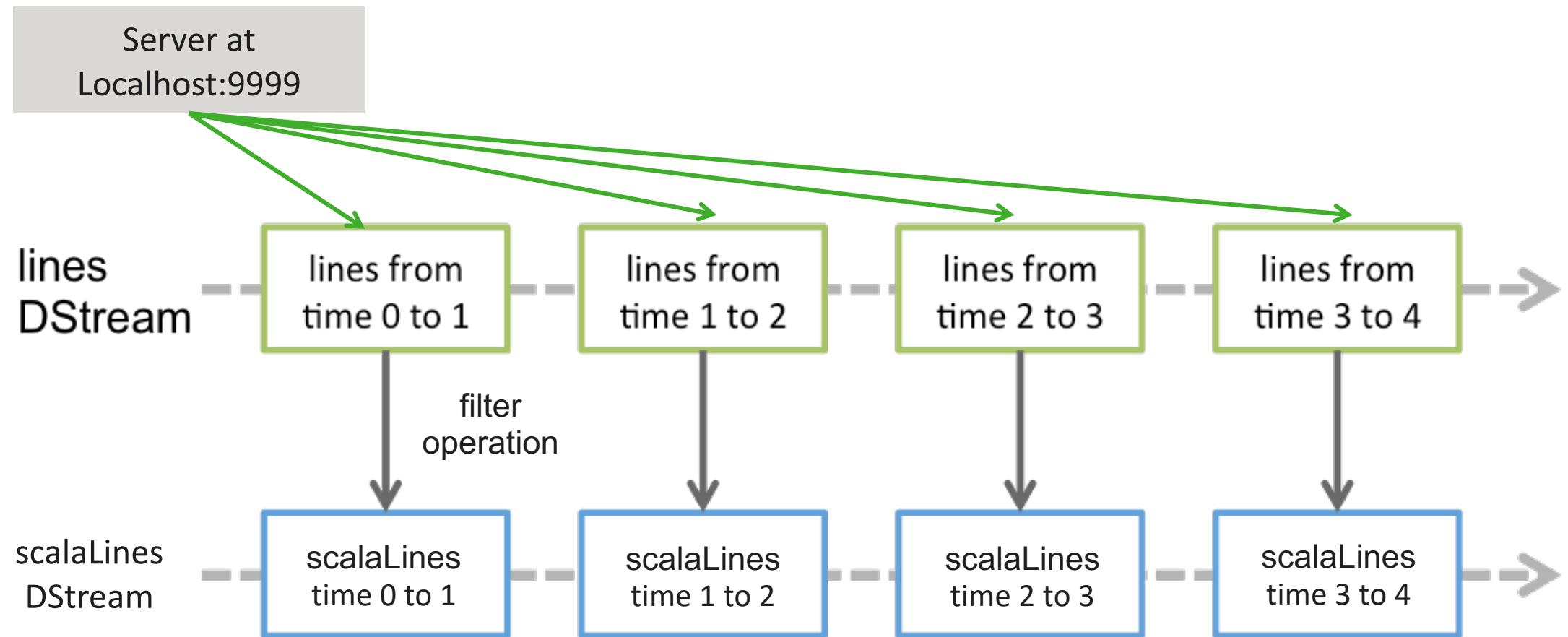
Scala 1
Scala 5

Time: 1429590650000 ms

Scala 8
```

# DStreams Illustrated

- Below, we illustrate how the input from the network is batched into DStreams, and then transformed



# Driver UI

- You can see that multiple jobs have run

The screenshot shows the Spark Driver UI at `localhost:4040/jobs/`. The UI has a header with tabs for Jobs, Stages, Storage, Environment, Executors, Streaming, and the current view, FindScalaLines application UI. The Jobs tab is selected.

## Spark Jobs (?)

Total Duration: 23 s  
Scheduling Mode: FIFO  
Active Jobs: 1  
Completed Jobs: 3

### Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	start at FindScalaLines.scala:27	2015/04/21 01:08:50	21 s	0/1	0/1

### Completed Jobs (3)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	print at FindScalaLines.scala:16	2015/04/21 01:09:10	10 ms	1/1	1/1
2	print at FindScalaLines.scala:16	2015/04/21 01:09:00	19 ms	1/1	2/2
1	print at FindScalaLines.scala:16	2015/04/21 01:09:00	40 ms	1/1	1/1

# MINI-LAB: Review Documentation

## Tasks to Perform

- Go to the Spark docs at <http://spark.apache.org/docs/latest/>
  - In the search box at the upper left, search for the types below
  - Review their documentation
  - **StreamingContext** (`org.apache.spark.streaming`)
    - Review the `socketTextStream()`, `start()`, `stop()`, and `awaitTermination()` functions
    - Optionally review other functions
  - **DStream** and **ReceiverInputStream**  
(`org.apache.spark.streaming.dstream`)
    - Look at the various transformation methods, e.g. `filter()`
    - These mirror RDD methods

# Lab 8.1: Spark Streaming

In this lab, we'll write and run a basic Spark Streaming program



# Introduction to Streaming

## Spark Streaming (1.0+)

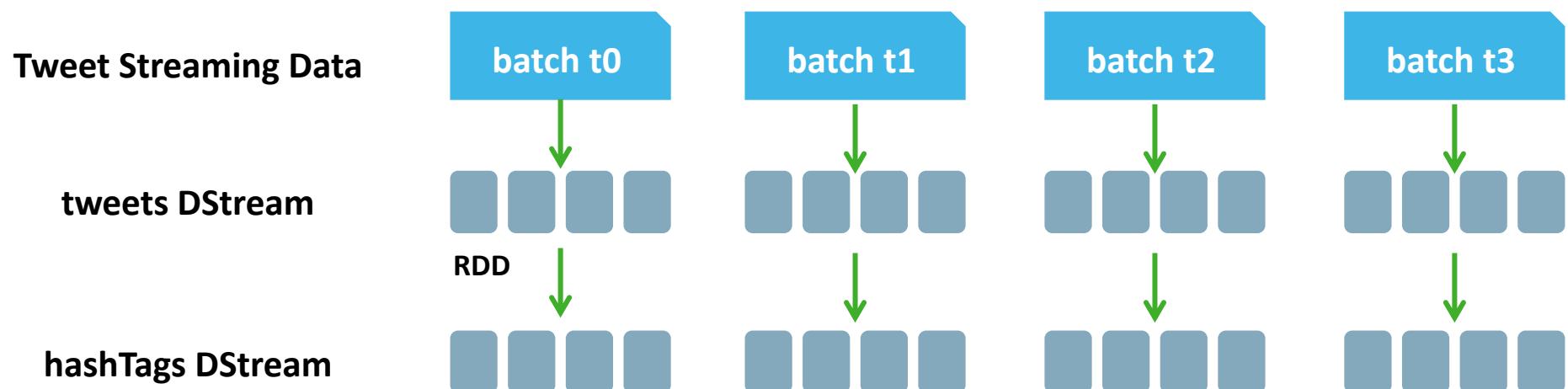
→ [Optional] Spark Streaming in Depth (1.0+)

## Spark Structured Streaming (2.0+)

### Consuming Kafka Data

# Stateless Transformation Illustrated

- At bottom we illustrate transformations on Twitter streams
  - Assume a source streaming Twitter tweets into Spark <sup>(1)</sup>
  - For each batch, an RDD is generated in the DStream <sup>(2)</sup>
    - Illustrated by the **tweets** DStream below
  - We then map/filter the DStream to get the tags from each tweet
    - Illustrated by the **hashTags** DStream below
  - Direct lineage from a parent RDD to a transformed RDD <sup>(3)</sup>



# DStream Stateless Transformations

- Supports many normal RDD transformations including:
  - Remember — These are applied to each RDD in the stream <sup>(1)</sup>

Transformation	Description	Example	f's Signature
map(f)	Apply f to each element in DStream	ds.map(x => x*2)	f: T -> U
flatMap(f)	Like map, but can output more than one result per element	ds.flatMap( x => x.split (" ") )	f: T -> Iterable[U]
filter(f)	Filter through each element when f is true	ds.filter( x=> x % 2 == 1)	f: T -> Boolean
repartition(n)	Change number of partitions	ds.repartition(10)	NA (numerical)
reduceByKey(f)	Combine values with same key using f (for pair data)	ds.reduceByKey( (x,y) => x+y )	f: (T, T) -> T
groupByKey()	Group values with same key (for pair data)	ds.groupByKey()	NA
mapPartitions(func)	Like map, but runs on the whole partition not on each element		

# Example of Stateless Transformations

- At bottom, we map a DStream that represents an access log
  - The map() produces a pair DStream of the form  
(IP address, 1)
  - The reduceByKey() generates an RDD with entries of the form below  
(basically visits by IP)  
(IP address, total access count)
- These transformations are very similar to regular RDD ones
  - The difference? As data streams in, new RDDs are created every batch interval to hold the data

```
// Assume accessLogsDStream is created elsewhere
val ipDStream = accessLogsDStream.map(entry =>
 (entry.getIpAddress(), 1))

val ipCountDStream = ipDStream.reduceByKey((x,y) => x+y)
```

# Data from Multiple DStreams

- Stateless transformations can also combine data from multiple DStreams within each time step
  - DStreams have the same join-related transformations as RDDs
- Example operations are similar to their RDD counterparts, e.g.
  - `cogroup()`, `join()`, and `leftOuterJoin()`
- Merge two streams using DStream's `union()` operator
  - Similar to regular RDDs
  - Use `StreamingContext.union()` for multiple streams

# Advanced Stateless Transformations

- At bottom, we join two DStreams that are produced by map and reduce transformations
  - The map transforms an IP data stream to a pair DStream with (IP address, content size)
  - reduceByKey computes the total bytes for a given IP
  - Lastly, we join the request count per IP (shown previously) with the total bytes for the given IP

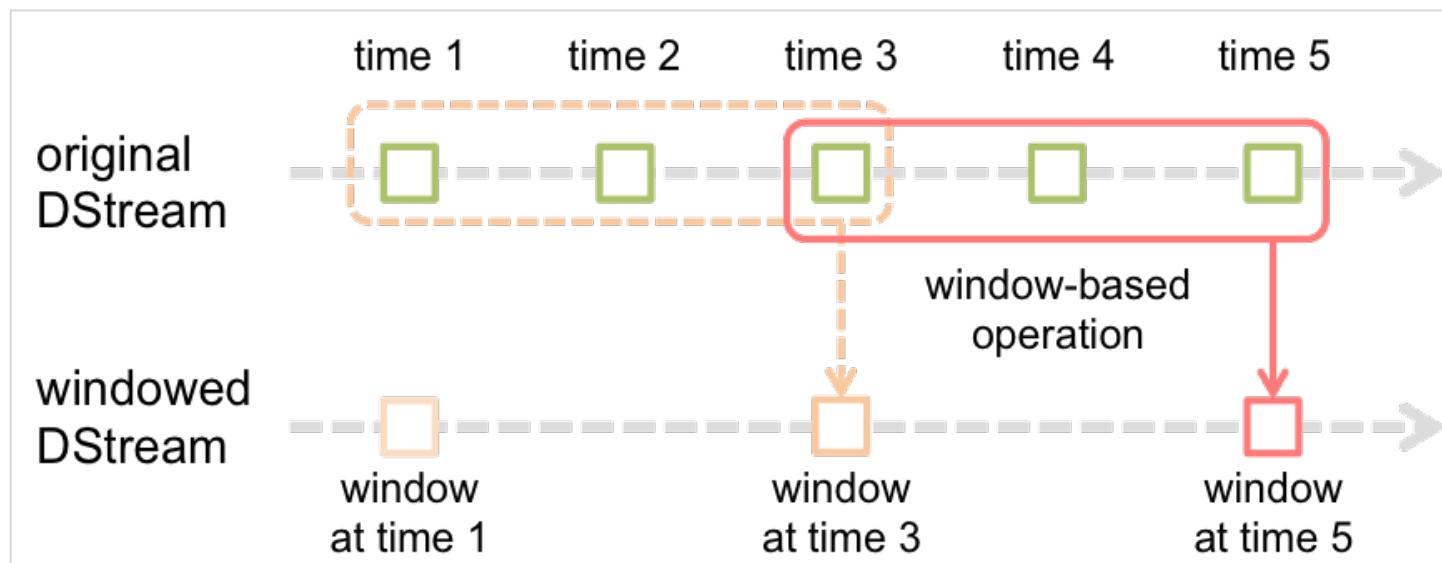
```
// Assume accessLogsDStream is created elsewhere
val ipBytesDStream = accessLogsDStream.map(entry =>
 (entry.getIpAddress(), entry.getContentSize()))

val ipBytesSumDStream =
 ipBytesDStream.reduceByKey((x, y) => x + y)

val ipBytesRequestCountDStream =
 ipCountsDStream.join(ipBytesSumDStream)
```

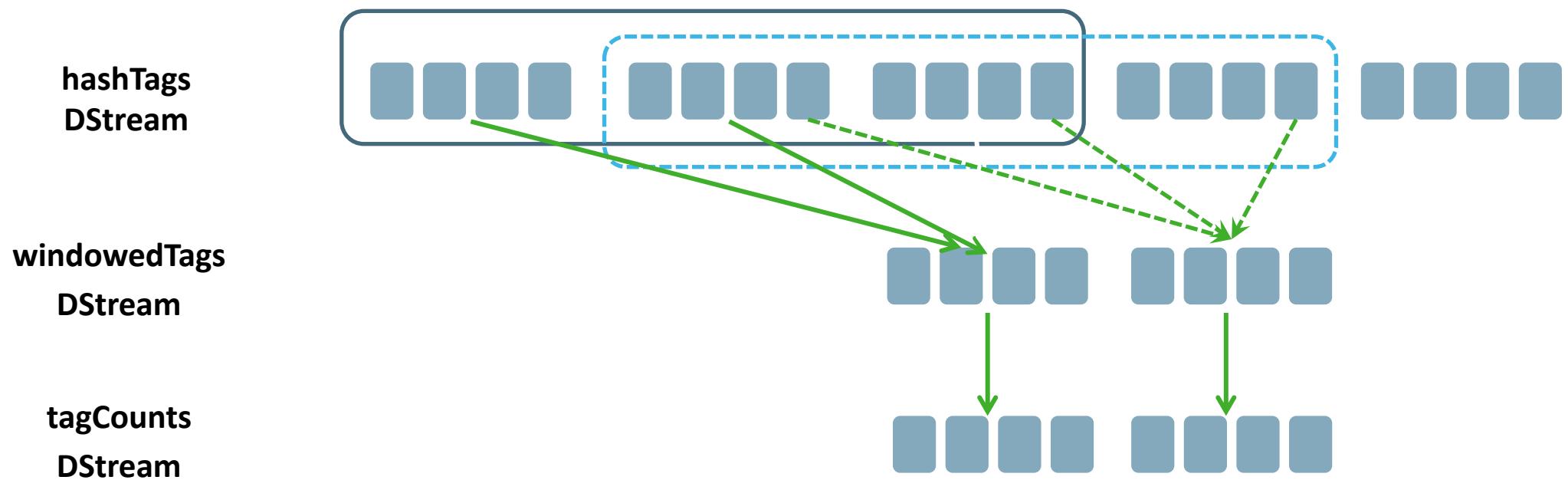
# Stateful (Windowed) Transformations

- These track data across time
  - Previous data used to generate or run current transformations
- Each windowed operation, requires 2 parameters
  - Both are multiples of batch interval
  - **Window Duration**: How many previous batches of data to use
  - **Slide Duration**: How often the new DStream computes results <sup>(1)</sup>
  - Below, window duration = 3, and slide duration = 2



# Windowed Transformation Illustrated

- Below we transform the hashTags DStream by windowing it <sup>(1)</sup>
  - With window duration = 3, and slide duration = 1 (x batch interval)
  - This results in the **windowedTags** DStream shown below
  - We might process this further — e.g. by counting the occurrence of each tag in the window, as shown in the **tagCounts** DStream <sup>(2)</sup>



# Stateful Transformations API

- Windowed computations apply transformations over a sliding window of data
  - As defined by the window length and sliding interval
- The simplest usage is to just take a window of data

```
def window(windowDuration: Duration,
 slideDuration: Duration)
```

  - Both args must be multiples of the batch interval
- We will demonstrate this via a windowed word count
  - It produces word counts of data in a sliding window
  - Let's look at the code next — we'll look at non-windowed and windowed versions

# Non-windowed Word Count Example

- The below should look familiar — it's just word count
  - But there is still a difference — it takes data from a stream, and continues running until stopped
  - We've seen the results — data is batched up

```
// Code fragment
val ssc =
new StreamingContext("local[2]", "WordCount", Seconds(5))

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
// Without windowing
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

# Windowed Word Count Example

- Below, `pairsWindow` is a windowed version of the pairs RDD
  - With a Window Duration =15, Slide Duration=5
- `wordCountsWindow` now contains data from a sliding window
  - Let's see how that works

```
// Code fragment
val ssc =
 new StreamingContext("local[2]", "WordCount", Seconds(5))
 val lines = ssc.socketTextStream("localhost", 9999)
 val words = lines.flatMap(_.split(" "))
 val pairs = words.map(word => (word, 1))
 // With windowing
 // Simple windwoing.
 val pairsWindow = pairs.window(Seconds(15), Seconds(5))
 val wordCountsWindow = pairsWindow.reduceByKey(_ + _)
 wordCountsWindow.print()
```

# Windowed Word Count Results

- At bottom, we see the input
- At right, the output <sup>(1)</sup>
  - Note how the words persist for the duration of a window interval
  - e.g., the "a" appears in the first 3 sets of results (since our window duration is 3)

```
$ nc -l k 9999
a a a b b c
d d d e e f
g g g h h i
```

(b,2)  
(a,3)  
(c,1)

-----  
(d,3)  
(b,2)  
(f,1)  
(e,2)  
(a,3)  
(c,1)

-----  
(d,3)  
(b,2)  
(f,1)  
(e,2)  
(a,3)  
(c,1)

-----  
(d,3)  
(h,2)  
(f,1)  
(e,2)  
(i,1)  
(g,3)

(h,2)  
(i,1)  
(g,3)

-----  
(h,2)  
(i,1)  
(g,3)



# reduceByKeyAndWindow()

- Lets you specify reduction over a window

`reduceByKeyAndWindow(reduceFunc: (V, V) ⇒ V, windowDuration: Duration, slideDuration: Duration): DStream[(K, V)]`

- Combines windowing and reducing — as shown below
- You get the same result as with the previous version
- However — both these versions have to sum up a full windows worth of data for each window duration interval

```
val ssc =
 new StreamingContext("local[2]", "WordCount", Seconds(5))
 val lines = ssc.socketTextStream("localhost", 9999)
 val words = lines.flatMap(_.split(" "))
 val pairs = words.map(word => (word, 1))
 // With more complex windowing
 val wordCountsWindow =
 pairs.reduceByKeyAndWindow((a:Int,b:Int) => (a + b),
 Seconds(15), Seconds(5))
 wordCountsWindow.print()
```

# Other Windowing Operations

- **countByWindow()**: Returns number of elements in each Window
- **countByValueAndWindow()**: Returns counts for each value
  - All of these methods return a DStream, with each RDD containing the appropriate value for its parent batch
- Below, we show some examples of using these

```
val ipDStream =
 accessLogsDStream.map{entry => entry.getIpAddress()}
val ipAddressRequestCount =
 ipDStream.countByValueAndWindow(Seconds(30), Seconds(10))
val requestCount =
 accessLogsDStream.countByWindow(Seconds(30), Seconds(10))
```

# Output Operations

- **print(num: Int)**: Print 1<sup>st</sup> num elements from each batch
- **print()**: Print 1<sup>st</sup> ten elements from each batch
  - DStream is materialized to accomplish this
- **save()**: Saves elements in a Dstream in a separate directory.
  - ipAddressRequestCount.saveAsTextFiles("outputDir", "txt")
- **saveAsHadoopFiles()**: Save each RDD as Hadoop file
  - Can pass prefix and suffix for filenames as arguments
  - Also saveAsTextFiles(), saveAsObjectFiles()
- **foreachRDD(forEachFunc)**: Apply function to each RDD <sup>(1)</sup>

```
ipAddressRequestCount.foreachRDD { rdd =>
 rdd.foreachPartition { partition =>
 val connection = openConnection // Connection to storage system
 partition.foreach { record => connection.send (record)}
 connection.close()
 }
}
```

# Input Sources

- Core Sources:
  - Stream of Files
  - Akka Actor Stream
- Popular Input Sources:
  - Sockets
  - Apache Kafka
  - HDFS
  - Apache Flume (Push based receiver / Pull based receiver)
  - Twitter
  - To include these additional receivers add the Maven artifact **spark-streaming-[projectname]\_2.10**,
  - e.g. **spark-streaming-kafka\_2.10**

# DStream Internal Interface (1 of 2)

- Defines how to generate batch in each interval
  - List of dependent (parent) DStreams
    - `def dependencies: List[DStream[_]]`
  - Slide duration: Interval at which it computes RDDs
    - `def slideDuration: Duration`
  - Method to compute RDD at a given time
    - `def compute(validTime: Time): Option[RDD[T]]`
- Example: Mapped DStream
  - Dependencies: **One parent DStream**
  - Slide duration: **Same as parent DStream**
  - Compute method: **Apply map function on parent DStream's RDD at the given time**

# DStream Internal Interface (2 of 2)

- Example: Windowed DStream
  - Dependencies: **One parent DStream**
  - Slide duration: **Window slide duration**
  - Compute method: **Apply union over all RDDs of parent DStream in the current window**
- Example: Receiver Input DStream (Networked — uses receiver)
  - Dependencies: **None**
  - Slide duration: **Batch Duration**
  - Compute method: **Create RDD with all data received in the last batch interval**

# Fault Tolerance

- For fault tolerance, input data can be replicated
  - Replicated across two nodes — tolerates single worker failure
    - Default for streams receiving data over the network (Kafka, Flume, ...)
  - Only the raw input data is replicated in memory
    - Not transformed RDDs
- Spark's memory manager called Block memory keeps the replicated data as long as it's required
  - And RDDs remember their lineage
- Data lost due to worker failure is recomputed using the raw input data and the lineage
  - So transformed data is also fault-tolerant
- **Checkpointing** is very critical for stateful transformations

# Checkpointing

- Saves RDD state periodically to a reliable filesystem
  - HDFS or S3
- **Why?** Stateful DStream RDDs can have very large lineage <sup>(1)</sup>
  - Where result RDDs depend on RDDs of previous batches, the dependency chain keeps growing longer with time
  - So recovery time could be large for data that has built up over a long time period
  - Task sizes / launch time also grow larger
- **Solution:** Checkpoint periodically
  - To recover from failure, only need to go back to the last checkpoint
  - Typically checkpoint interval = 5-10 times DStream sliding interval
    - Tradeoff between overhead of checkpoint, vs. time saved

# 24/7 Operations

- Spark Streaming can be run in 24/7 mode even if a worker or drivers fail.
- In order for Streaming to run 24/7, one needs reliable storage system such as S3 or HDFS for Checkpointing  
`ssc.checkpoint("hdfs://...")`
- Lack of Checkpointing throws a warning/error even in local setup.



**Introduction to Streaming  
Spark Streaming (1.0+)  
[Optional] Spark Streaming in  
Depth (1.0+)**



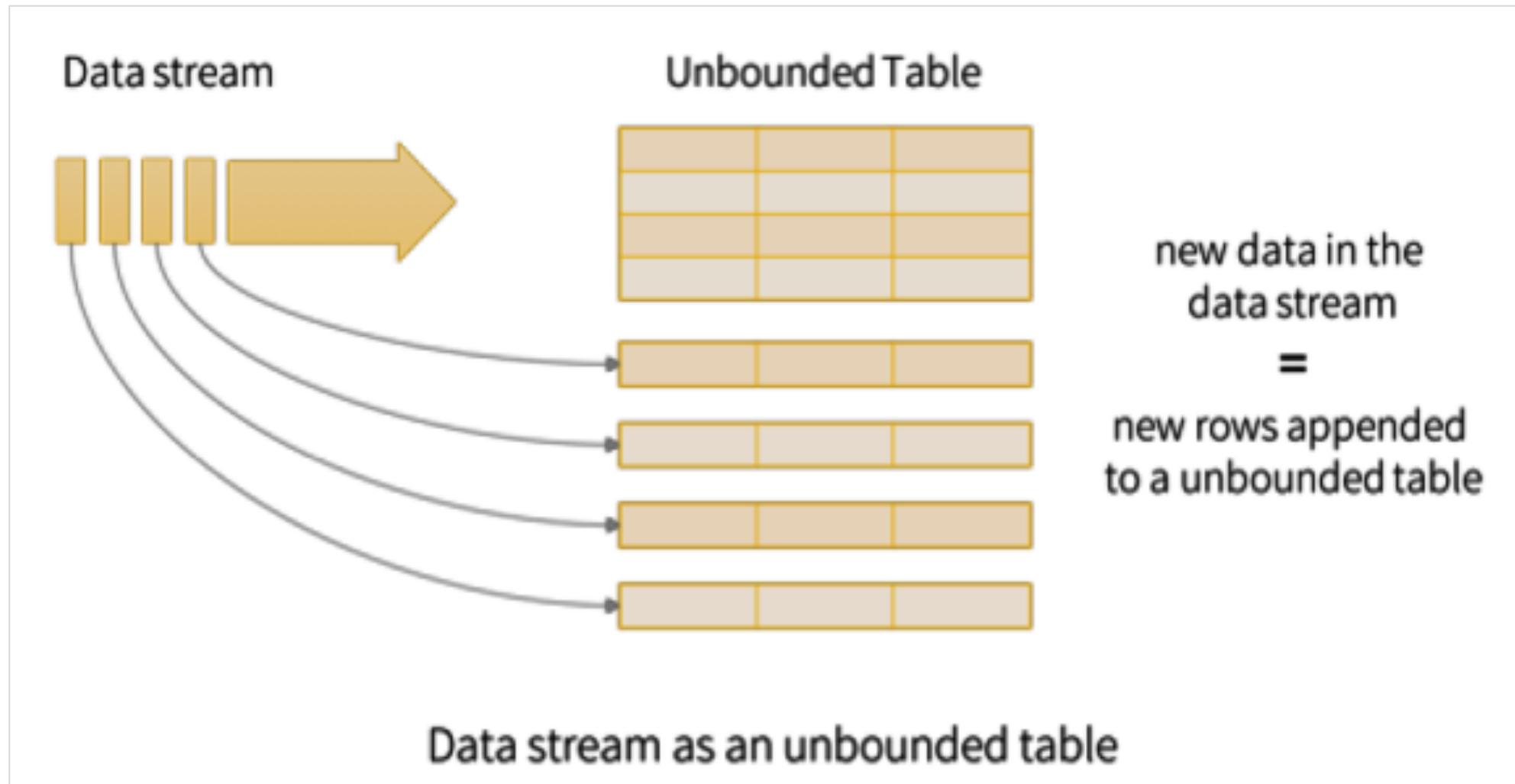
**Spark Structured Streaming (2.0+)**  
**Consuming Kafka Data**

# Key Concepts

- Designed to support **continuous applications**:
  - End-to-end application that reacts to data in real-time
- Built over DataFrames — higher level than Spark Streaming
  - Streaming API is same as batch API !
- Important new features to support continuous applications:
  - **Streaming job consistent with batch jobs**:
    - Written using DataFrame API
    - Output guaranteed to be same as running batch job on prefix of data
  - **Transactional integration with storage systems**:
    - Process data exactly once
    - Updates output sinks transactionally
  - **Integrates with rest of Spark**
    - Spark SQL, ML, etc.
    - Goal: Every library in Spark runs incrementally on Structured Streaming

# How Does it Work?

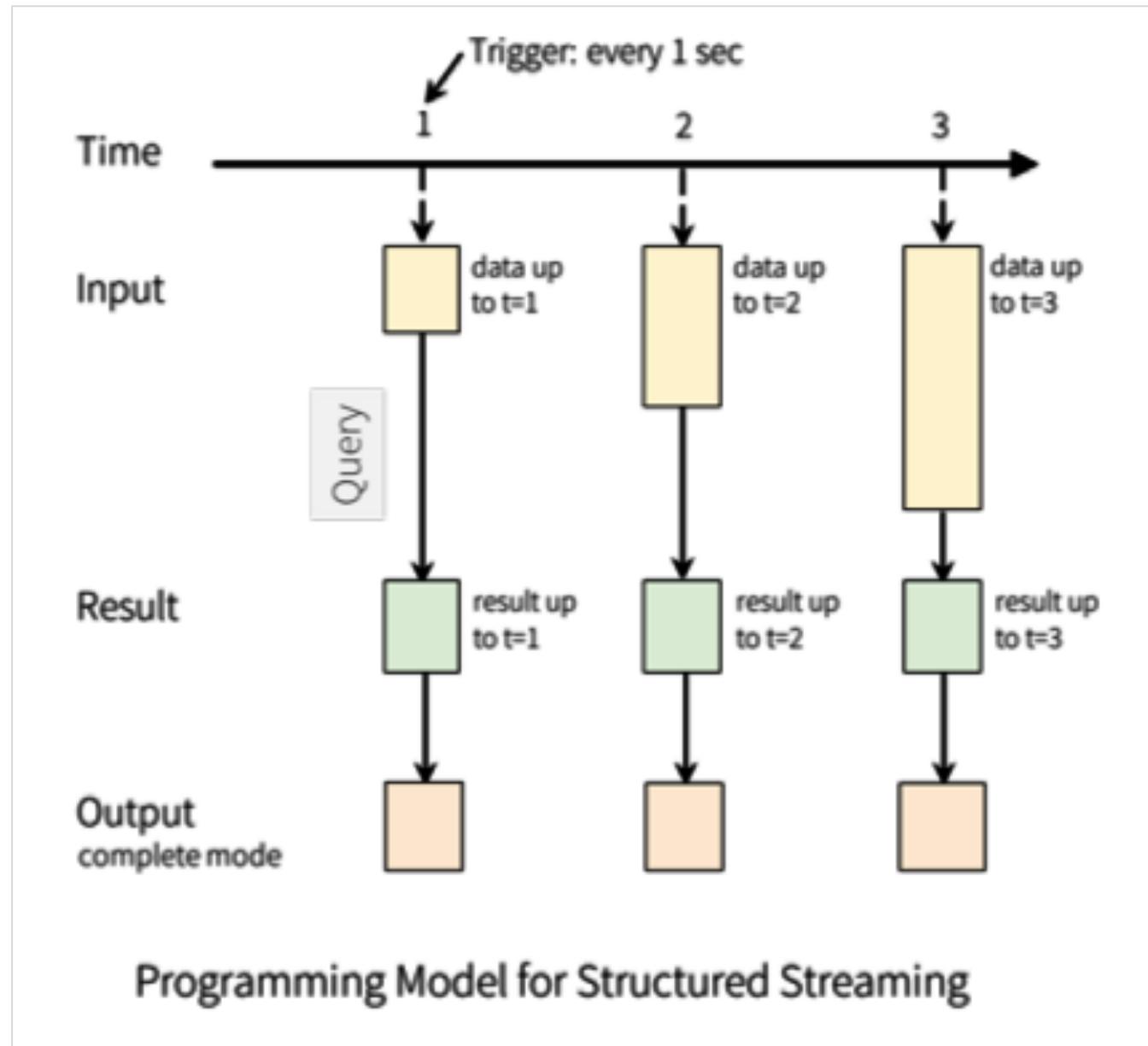
- Consider the input data stream as an input table
  - New data that arrives is like a new row appended to the table



# Result Table

- A query on the input creates a result table
  - For each trigger interval (e.g. 1 sec), new rows get appended to the input table
  - Eventually, the result table is updated
  - New data is then processed by your query transformations
- Supports three **output modes**:
  - **complete**: Entire updated table is output
  - **append**: New rows appended since last trigger are written
  - **update**: Rows updated since last trigger are written

# Result Table Illustrated



# Steps for Structured Streaming

- Set up input DataFrame
  - Use `SparkSession.readStream()` to create `DataStreamReader`
  - Specify the input source via `format()`
    - Currently file, Kafka or socket sources are supported
  - Set input source options (depends on source type)
- **Execute query** to start streaming
  - Use `DataSet.writeStream()` to create `DataStreamWriter`
  - Specify **trigger interval** (how often data is sampled)
    - Default — as soon as possible after data is available
  - Specify **output sink details** (data format, location, etc.)
  - Specify **output mode**
  - **Start** the query processing

# Sample Program — Overview

- This simple program does the same processing as our previous Streaming (1.x) example
  - Sets up an input stream, with a source that reads from a socket
    - Using a trigger interval of 5 sec.
  - Filters out all input lines, except those containing string "Scala"
    - Using standard DataFrame operations
  - Writes the filtered input to the console
    - Using an output stream
- The input data is created via the **nc** (netcat) program as previously

# Sample Code (1 of 2 — Initialization)

- Gets a **DataStreamReader** from session via **readStream()**
  - Uses **format()** to specify input data source format
    - Socket in this example
  - Uses **option()** to specify any options for data source
    - Host and port in this example
  - Calls **load()** to load input (evaluated lazily)
    - No work done until you consume the streaming data with a sink
  - Filters the data — standard **DataFrame.filter()**

```
// Code excerpt showing streaming code only

val lines = spark.readStream
 .format("socket")
 .option("host", "localhost")
 .option("port", 9999)
 .load()
val scalaLines = lines.filter('value.contains("Scala"))
```

# Sample Code (2 of 2 – Consume Data)

- Creates **DataStreamWriter** via **writeStream()**
  - Sets trigger interval to be 5 seconds
  - Sets output mode to be **append** (appends new output)
  - Sets format to be **console** (outputs to console)
  - Starts processing via **start()**
    - Returns a **StreamingQuery** instance

```
import org.apache.spark.sql.streaming.ProcessingTime

// Code fragment
val query = scalaLines.writeStream
 .trigger(ProcessingTime("5 seconds"))
 .outputMode("append")
 .format("console")
 .start()

query.awaitTermination() // Standalone programs
```

# Supported Sources and Sinks

- **DataStreamReader** currently supports these input sources
  - **Socket streams**: (Testing only) read input from socket
    - Via `format("socket")`
  - **File streams**: CSV, JSON, text, Parquet
    - Via `csv()`, `json()`, `parquet()`, `textFile()` functions
  - **Kafka**: Poll data from Kafka <sup>(1)</sup>
    - Via `format("kafka")`
- **DataStreamWriter** currently supports these output sinks
  - **Console sink**: (for debugging) — outputs to console
    - Via `format("console")`
  - **File sink**: CSV, JSON, text, Parquet
    - Via `format("XXX")` where XXX is "parquet", "json", etc.
  - **Memory sink**: (for debugging) — store output on in-memory table
    - Via `format("memory")`
  - **foreach sink**: Run arbitrary computation on output records
    - Via `foreach(...)`

# Summary

- Spark Structured Streaming is better than Spark Streaming
  - Based on Spark SQL / DataFrames
  - Leverages all benefits — Catalyst, Tungsten, higher level API
  - Future enhancements will be done here
    - Spark Streaming is legacy now
- Spark Structured Streaming is an **Alpha** release
  - Not feature complete
  - May change before final release
  - Use with caution

# MINI-LAB: Review Documentation

## Tasks to Perform

- Go to the Spark docs at <http://spark.apache.org/docs/latest/>
  - In the search box at the upper left, search for the types below
  - Review their documentation
  - **DataStreamReader** (`org.apache.spark.sql.streaming`)
    - `format()`, `csv()`, `json()`, `parquet()`, `text()`, `textFile()` functions
    - Optionally review other functions
  - **DataStreamWriter** (`org.apache.spark.sql.streaming`)
    - `format()`, `outputMode()`, `foreach()`, `trigger()`
    - Look at `start()`, and the `StreamingQuery` type
    - In `StreamingQuery`, look at the status methods, and methods to control execution (`stop()`, etc.)

## Lab 8.2: Spark Structured Streaming

In this lab, we'll write and run a basic Spark Structured Streaming program



**Introduction to Streaming**  
**Spark Streaming (1.0+)**  
**[Optional] Spark Streaming in**  
**Depth (1.0+)**  
**Spark Structured Streaming (2.0+)**

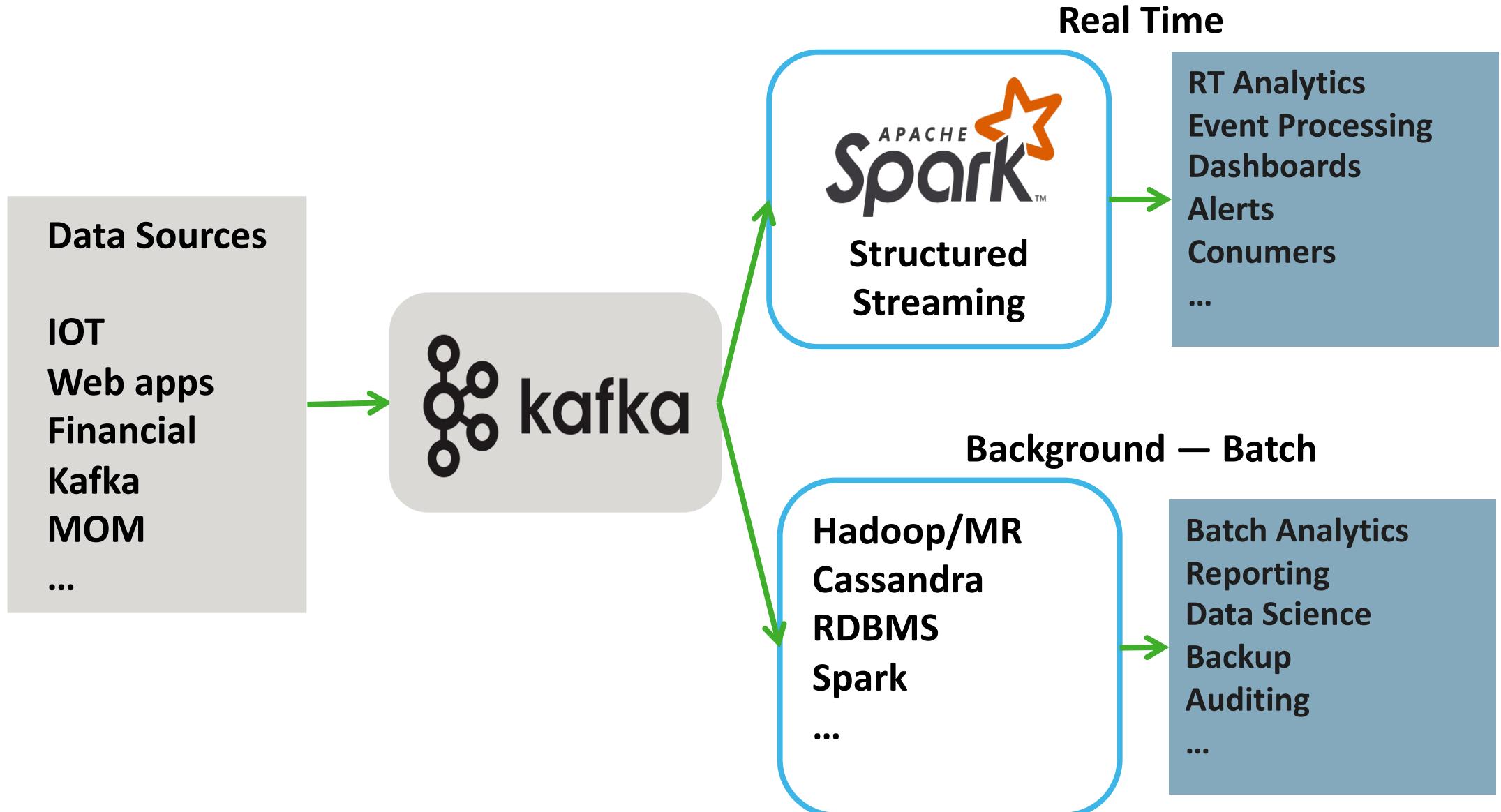


**Consuming Kafka Data**

# Kafka Overview

- **Apache Kafka:** Distributed Streaming Platform
  - Publish and Subscribe to streams of records
  - Fast, scalable, durable, and fault-tolerant
  - Often replaces JMS and other messaging systems
    - Higher throughput, reliability and replication
- Very popular for Big Data applications
  - Great performance and scalability
  - Simple operationally
  - Robust Replication
  - Integrates well with other systems
    - Spark Streaming, Flume/Flafka, Storm, HBase, ...
  - Many large adopters
    - LinkedIn, Twitter, Uber, PayPal, ...

# Kafka/Spark Streaming Architecture



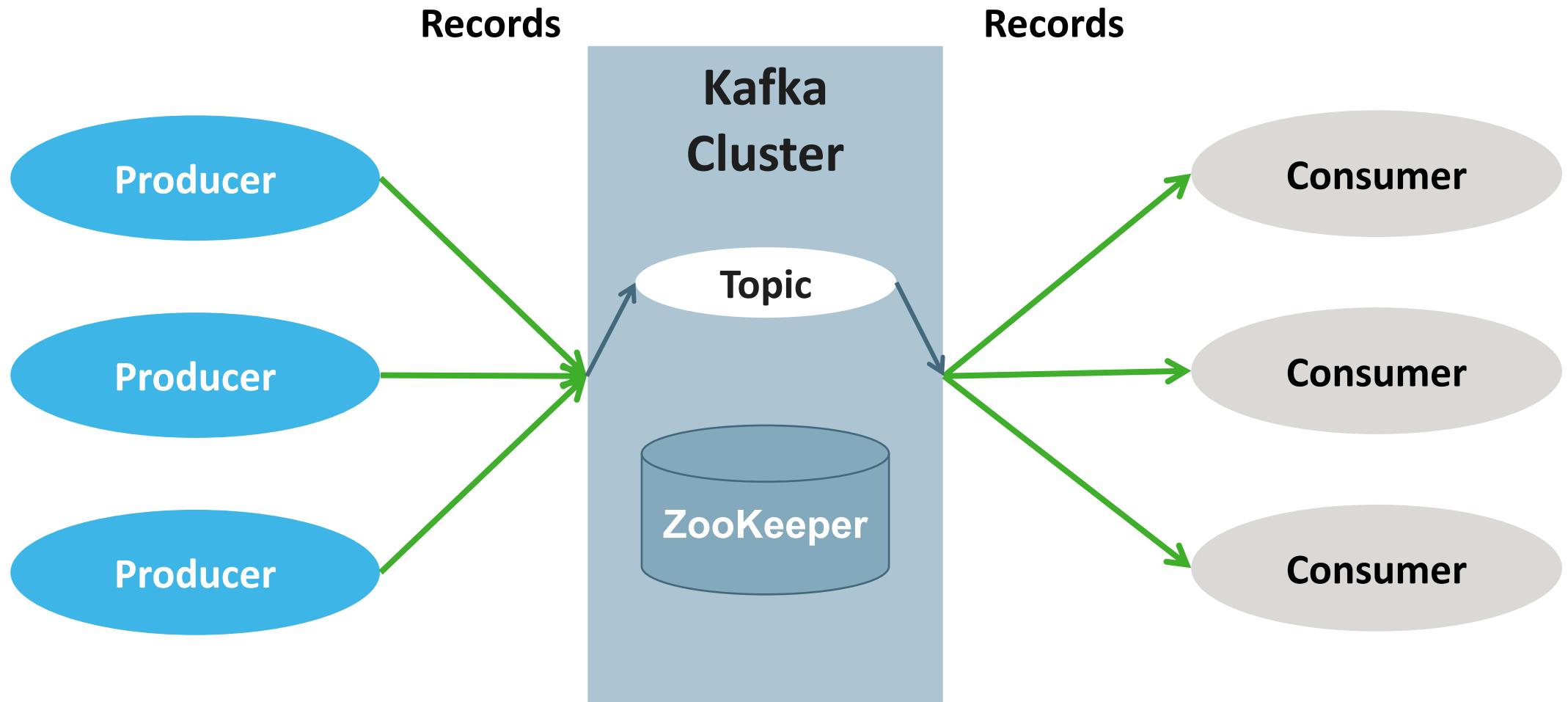
# Kafka Advantages

- Kafka sits between data sources and consumers
  - Decouples data streams from consumers
  - Data sent to consumers in parallel and fault-tolerant way
  - Flexible message production/consumption
- Kafka messaging supports
  - Feeding high-latency daily/hourly data analysis to Spark
  - Feeding real-time data to microservices
  - Event handling
  - Feeding data to real-time analytics
  - Updating dashboards and summaries
  - Etc.

# Kafka Fundamentals

- **Record**: Basic data organizational unit
  - Contain key (optional), value, and timestamp; Immutable
- **Topic**: Stream of records (“/orders”, “/user-signups”)
  - Think of it as a feed name
  - Stored on disk in Log format (made of partitions/segments)
- **Producer API**: Produces streams of records
- **Consumer API**: Consumes streams of records
- **Broker**: Kafka server in a Kafka Cluster.
  - Cluster consists of multiple Kafka Brokers distributed across multiple machines
- **ZooKeeper**: Coordinates broker/cluster topology
  - Kafka depends on ZooKeeper

# Kafka: Topics, Producers, and Consumers



# Basic Kafka Usage

- **Producers append** records to a topic
  - Cluster keeps all records for a configurable retention time
- **Consumers read** data from a topic
  - At their own pace
  - Multiple consumers can subscribe to a topic — they receive incoming messages as they arrive
- Three choices for consumers to ingest data
  - **earliest**: Start reading at beginning of stream
    - Get all records (that haven't expired)
  - **latest**: Start now, processing only new data arriving after query starts
  - **assign**: Specify precise location (beyond scope of course)

# Spark Structured Streaming and Kafka

- Uses structured streaming API we've already seen
  - You work with DataFrames, as usual
- Steps for consuming Kafka data
  - Get a DataStreamReader via `readStream()`
  - Specify "**kafka**" format, the Kafka bootstrap servers, the topic name, and the mode for ingesting data (e.g. `earliest`)
  - Doesn't ingest data yet — still need to start the streaming

```
// Code excerpt showing streaming code only

val kafkaDF = spark
 .readStream
 .format("kafka")
 .option("kafka.bootstrap.servers", "localhost:9092")
 .option("subscribe", "someTopic")
 .option("startingOffsets", "earliest")
 .load()
```

# Kafka Options Overview

- **kafka.bootstrap.servers**: Advertised name of kafka server
  - May need to set advertised.listeners in Kafka server.properties  
advertised.listeners=PLAINTEXT://localhost:9092
- **subscribe**: Comma separated list of topics to subscribe to
- **subscribePattern**: Regex for matching topics to subscribe to
- **startingOffsets**: Start point for query (e.g. earliest)
- **endingOffsets**: Ending point for batch query
  - **latest**: The latest data
  - A specific offset
  - Creates query on defined range of offsets for batch querying

# An Example — Voter Data

- Let's look at ingesting the data at bottom
  - It's simple voter data consisting of a voters gender (M or F), age, and party (R or D)
    - This implies no endorsement on our part of any political or gender worldview — it's just a very simple example
  - It's in JSON format
- Assume that the data below is coming into Kafka
  - On the topic "voters"
  - Exit poll data from election night !

```
{"gender": "F", "age": 26, "party": "D"}
{"gender": "M", "age": 48, "party": "D"}
{"gender": "M", "age": 48, "party": "R"}
{"gender": "F", "age": 74, "party": "D"}
{"gender": "M", "age": 96, "party": "R"}
{"gender": "F", "age": 95, "party": "D"}
...
```

# Ingesting the Data

- Below, we create a DataFrame to stream in Kafka data
  - It's schema doesn't look like our topic data (see next slide)
  - Why? Our data is embedded in the data from Kafka
    - In the **value** column
    - Other columns are Kafka-related

```
// Our Voter stream in Spark
> val kafkaVoterDF = spark
 .readStream
 .format("kafka")
 .option("kafka.bootstrap.servers", "localhost:9092")
 .option("subscribe", "voters")
 .option("startingOffsets", "earliest")
 .load()

kafkaVoterDF: org.apache.spark.sql.DataFrame = [key: binary, value: binary
... 5 more fields]
```

# The Kafka Streaming Data Schema

- The schema for data streaming from Kafka is shown below
  - Standard for all data originating in Kafka
  - The **value** column holds the actual payload

```
// Schema from our Kafka Voter stream data

scala> kafkaVoterDF.printSchema
root
|-- key: binary (nullable = true)
|-- value: binary (nullable = true)
|-- topic: string (nullable = true)
|-- partition: integer (nullable = true)
|-- offset: long (nullable = true)
|-- timestamp: timestamp (nullable = true)
|-- timestampType: integer (nullable = true)
```

# Starting the Query / Viewing the Data

- Below, we start a streaming query, as seen previously
  - It outputs to the console
  - No data from the topic yet — but we can see the fields

```
// Query Raw Data - straight from kafkaVoterDF

val rawVoterQuery = kafkaVoterDF.writeStream
 .trigger(ProcessingTime("5 seconds"))
 .outputMode("append")
 .format("console")
 .start()
```

```

Batch: 0

```

```
+---+---+---+---+---+---+
|key|value|topic|partition|offset|timestamp|timestampType|
+---+---+---+---+---+---+
+---+---+---+---+---+---+
```

# Viewing Some Data

- Some data has come into the topic — and streamed into Spark
  - It's automatically processed by our query
  - But — the value is in binary format — not very useful
  - We need to transform it — fairly easy to do (e.g. cast to a String)
  - For JSON, we'll use a Spark SQL helper function

```

Batch: 1

+---+-----+-----+-----+
| key| value| topic|partition|offset|
timestamp|timestampType|
+---+-----+-----+-----+
| null|[7B 22 67 65 6E 6...|voters| 0| 0|2017-08-09 17:50:...|
0| |
| null|[7B 22 67 65 6E 6...|voters| 0| 1|2017-08-09 17:50:...|
0|
// ... Remaining data omitted
```

# Processing the Data as a String

- Below, we select the value column, and cast to string
  - Sample streaming results are shown below

```
> val voterStringDF =
kafkaVoterDF.select('value.cast("string")).as("voterString")

> val stringVoterQuery = voterStringDF.writeStream
.trigger(ProcessingTime("5 seconds"))
.outputMode("append") // append is required when no aggregation (1)
.format("console")
.option("truncate", "false")
.start()

Batch: 0

+-----+
| value |
+-----+
| {"gender": "F", "age": 26, "party": "D"} |
| {"gender": "M", "age": 48, "party": "D"} |
```

# Processing the Data as JSON

- To ingest as JSON, we use the `from_json` function
  - `org.apache.spark.sql.functions`
  - Must specify the schema — no auto-scanning for streaming
- We define the schema below, and apply it to our data
  - We then aggregate by gender and party
  - Voter affiliation is important info on election night !

```
> val voterSchema = (new StructType).add("gender", StringType)
 .add("age", LongType).add("party", StringType)

// from_json creates a level of nesting (voterJSON is top level)
> val voterJSONDF =
 kafkaVoterDF.select(from_json('value.cast("string")', voterSchema)
 .as("voterJSON"))

// To extract our data, we need to navigate through voterJSON
> val voterStatsDF =
 voterJSONDF.groupBy("voterJSON.gender", "voterJSON.party").count
```

# Start the Query

- Starting is same as before — but now we're transforming the data
  - Using our familiar DataFrame capabilities
  - As data streams in, it's processed (once a minute) and displayed

```
> val voterStatsQuery = voterStatsDF.writeStream
 .trigger(ProcessingTime("1 minute"))
 .outputMode("complete") // complete required here (1)
 .format("console").start()

// ... Batch 0-7 Output omitted

Batch: 8

+---+---+---+
|gender|party|count|
+---+---+---+
| F| R| 1029|
| M| D| 1023|
| M| R| 1080|
| F| D| 970|
+---+---+---+
> voterStatsQuery.stop // Stop the query processing
```

# What We've Done so Far

- In a few slides, we've ingested data from Kafka
  - It's JSON, which we parsed fairly easily into a DataFrame
  - We applied some useful transformations
    - Standard DataFrame stuff
- **Viola!** We've got useful information from the data stream
  - A count of voters by gender and party
    - Giving talking heads something to talk about
    - Updated in real time every minute as our data streams in
- It's easy to do further analysis
  - On the next pages, we group the data into age buckets and party
  - Also useful for the talking heads
  - It's a bit complex, so don't sweat the details

# Creating Buckets

- We first create an age bucket
  - An extra column that spans 10 years <sup>(1)</sup>
  - Then we group the data by the buckets

```
> val binnedVoterDF = spark
 .readStream
 .format("kafka")
 .option("kafka.bootstrap.servers", "localhost:9092")
 .option("subscribe", "voters")
 .option("startingOffsets", "earliest")
 .load()
 .select(from_json('value.cast("string"), voterSchema).as("voterJSON"))
 .withColumn("bucket", ((voterJSON.age)-18) / 10).cast("int"))

// Create data grouped by bucket - done in SQL which is a bit easier
> binnedVoterDF.createOrReplaceTempView("binnedVoters")
> val rangedVoterDF = spark.sql("SELECT bucket, bucket*10+18 AS startAge,
 (bucket+1)*10+18-1 AS endAge, voterJSON.party, count(*) FROM binnedVoters
 GROUP BY bucket, voterJSON.party ORDER BY bucket ASC").toDF
```

# Stream Data into the Buckets

- Now the talking heads can talk about age range turnout !
  - The 28-37 Ds are sleeping at home ...

```
> import org.apache.spark.sql.streaming.ProcessingTime
> val voterRangeQuery = rangedVoterDF.writeStream
 .trigger(ProcessingTime("1 minute")).outputMode("complete")
 .format("console").start()

Batch: 2 // ... Batch 0-1 Output omitted

+---+---+---+---+
|bucket|startAge|endAge|party|count(1)|
+---+---+---+---+
| 0| 18| 27|R| 106|
| 0| 18| 27|D| 111|
| 1| 28| 37|R| 108|
| 1| 28| 37|D| 88|
| 2| 38| 47|D| 116|
| 2| 38| 47|R| 107|
// Remaining data omitted
+---+---+---+---+
> voterRangeQuery.stop // Stop the query processing
```

# Summary

- Not much code needed to ingest data from Kafka
  - And then use the full power of Spark, Spark SQL, DataFrames ...
  - There are more options and choices — but you have the basics
- Structured Streaming
  - Based on DataFrames — making it very powerful
  - Brings new capabilities to processing streaming data
  - Kafka integration is relatively easy to use
    - The programming API is built on familiar Spark SQL capability
  - It's a win

A photograph of a man and a woman smiling and looking at a computer screen. The man has a beard and is wearing a light-colored shirt. The woman has blonde hair tied back and is wearing a white top. A green diagonal bar runs from the bottom right corner across the slide.

## Lab 8.3: Spark Structured Streaming with Kafka

In this lab, we'll consume data from Kafka using Spark Structured Streaming

# Review Questions

- What is a streaming application? Why do we need it?
- What is Spark Streaming (1.0+) and what are its principal types?
- What is Spark Structured Streaming (2.0+), and what are its principal types?

# Lesson Summary

- Streaming data comes in continuously, often from many sources, with small payloads
  - Processing it requires new capabilities — incremental processing, windowed processing, etc.
- **Spark Streaming** was introduced early in Spark
  - It is based on **RDDs**, **StreamingContext**, and **DStreams**
  - **DStream**: Sequence of RDDs representing streaming input data
    - DStreams support many of the familiar RDD transformations
  - Spark Streaming supports stateless and stateful (windowed) processing

# Lesson Summary

- **Spark Structured Streaming** was introduced in Spark 2
  - It supports continuous applications that react to data in real-time
  - Based on **DataFrames**, **DataStreamReader**, **DataStreamWriter**, and **StreamingQuery**
  - Has many new features such as consistency with batch jobs, and TX integration with storage systems
  - **Alpha** release in Spark 2.0/2.1
  - New streaming work is focused on structured streaming

# Recap

# Recap of What We've Done

- We've covered the key elements of Spark
  - Architecture, API, Guidelines
- We've explored many of the Spark 2 capabilities
  - Spark SQL's easier API using Datasets, and DataFrames
  - Catalyst and Tungsten optimizers
- Spark is evolving rapidly
  - Its API has expanded greatly from earlier releases
  - It should be relatively stable for now

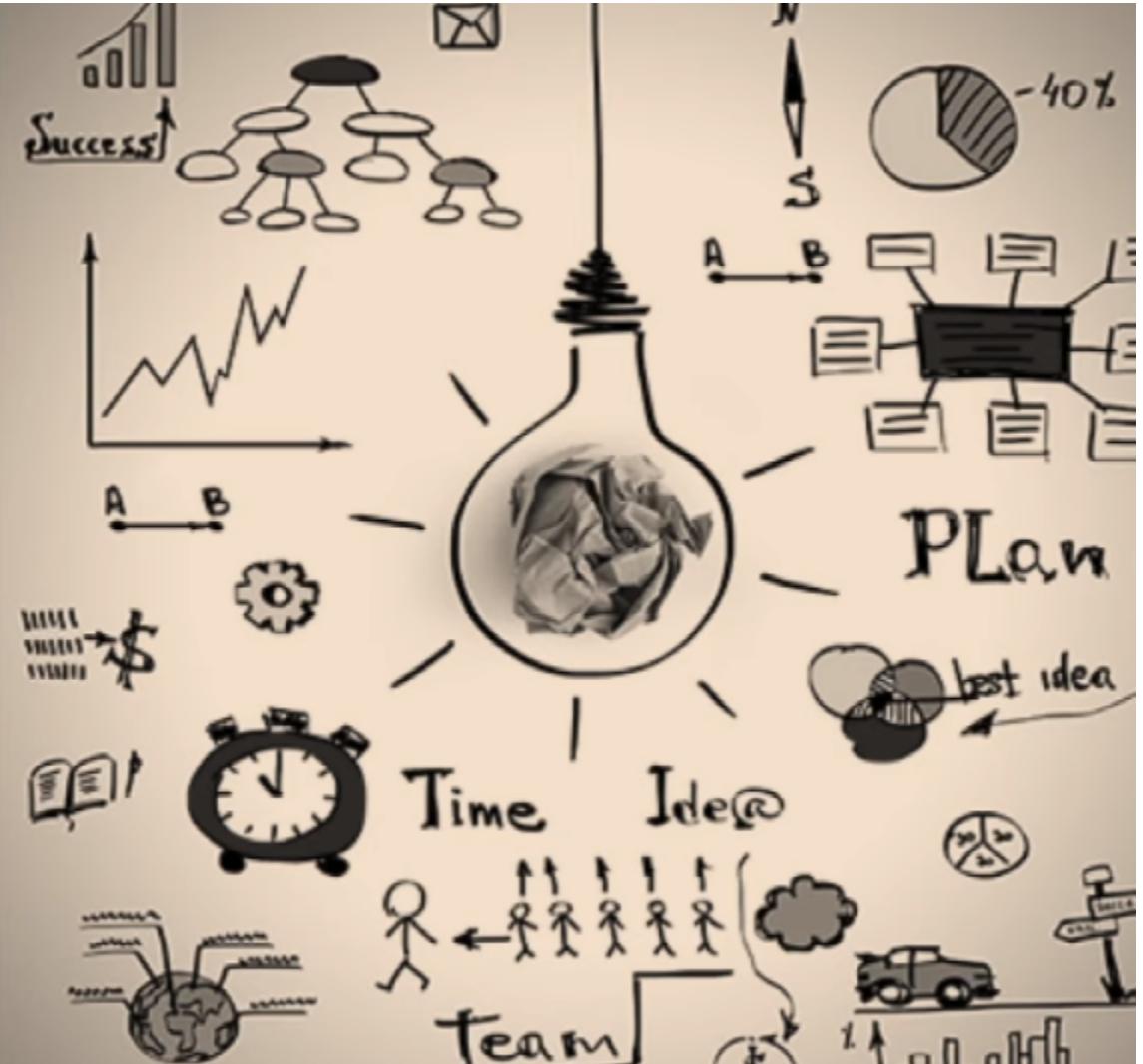
# Resources

- Books
  - “[Learning Spark](#)” by Patrick Wendel, Matai Zaharia, ...
  - “[Advanced Analytics With Spark](#)” by Sandy Ryza, ...
  - "Spark The Definitive Guide" by Bill Chambers and Matei Zaharia
    - Excellent and thorough upcoming book currently available in pre-release
- The API docs of course!
  - And Quick Start and programming guide
  - All available at: <http://spark.apache.org/docs/latest/>



# Maximize Performance

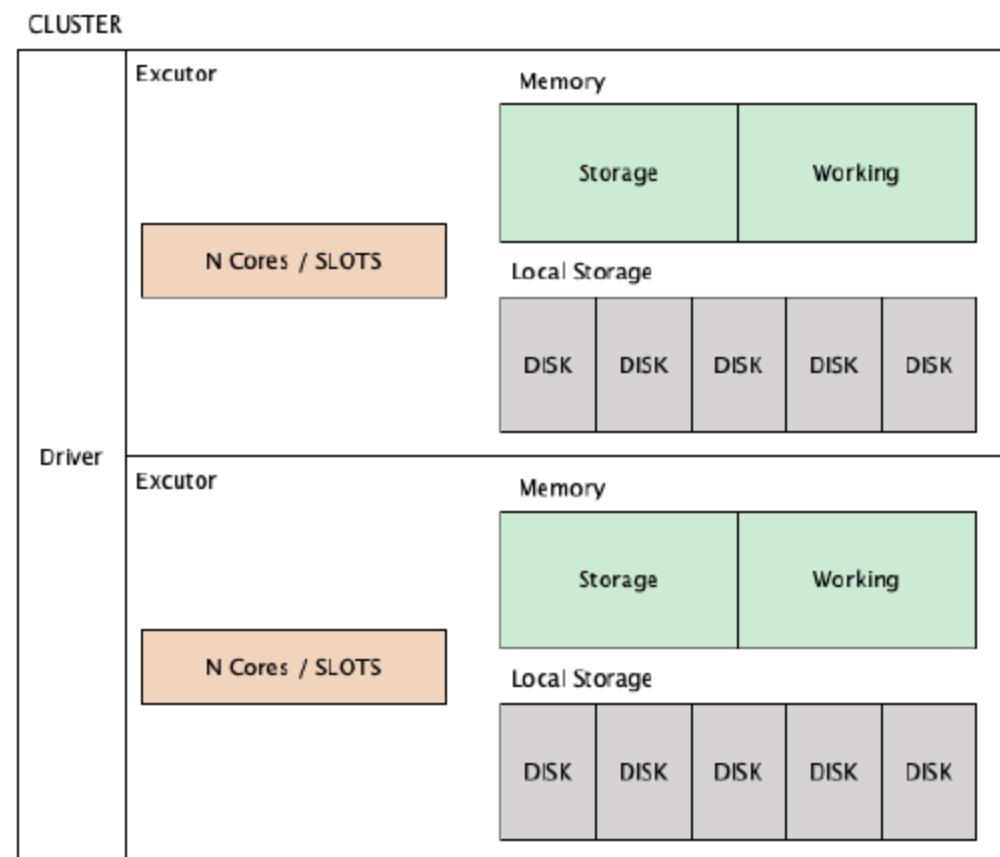
Read Plan.  
Interpret Plan.  
Tune Plan.  
Track Execution.



# Talking Points

- Spark Hierarchy
- The Spark UI
- Rightsizing & Optimizing
- Advanced Optimizations

# Spark Hierarchy



# Spark Hierarchy

- Actions are eager
  - Made of transformations (lazy)
    - narrow
    - wide (requires shuffle)
  - Spawn jobs
    - Spawn Stages
      - Spawn Tasks
        - » Do work & utilize hardware

Action
Jobs
Stages
Tasks = Slots = Cores 1 Task 1 Partition 1 Slot 1 Core

# Understand Your Hardware

- Core Count & Speed
- Memory Per Core (Working & Storage)
- Local Disk Type, Count, Size, & Speed
- Network Speed & Topology
- Data Lake Properties (rate limits)
- Cost / Core / Hour
  - Financial For Cloud
  - Opportunity for Shared & On Prem

# Get A Baseline

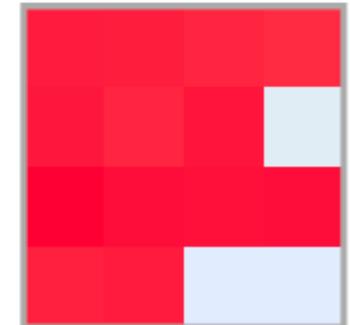
- Is your action efficient?
  - Long Stages, Spills, Laggard Tasks, etc?
- CPU Utilization
  - GANGLIA / YARN / Etc
  - Tails

## Goal

CPUs Total: **216**  
Hosts up: **14**  
Hosts down: **0**

Current Load Avg (15, 5, 1m):  
**34%, 79%, 125%**  
Avg Utilization (last hour):  
**0%**

## Server Load Distribution



# Minimize Data Scans (Lazy Load)

- Data Skipping
  - HIVE Partitions
  - Bucketing
    - Only Experts – Nearly Impossible to Maintain

Cmd 8

```
1 val master_no_lazy = ss_sub
2 .join(dt_sub.withColumnRenamed("d_date_sk", "ss_sold_date_sk"), Seq("ss_sold_date_sk"))
3 .join(item.withColumnRenamed("i_item_sk", "ss_item_sk"), Seq("ss_item_sk"))
4 .join(inv.withColumnRenamed("inv_item_sk", "ss_item_sk"), Seq("ss_item_sk"))
```

# No Lazy Loading

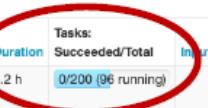
Cmd 8

```
1 val master_no_lazy = ss_sub
2 .join(dt_sub.withColumnRenamed("d_date_sk", "ss_sold_date_sk"), Seq("ss_sold_date_sk"))
3 .join(item.withColumnRenamed("i_item_sk", "ss_item_sk"), Seq("ss_item_sk"))
4 .join(inv.withColumnRenamed("inv_item_sk", "ss_item_sk"), Seq("ss_item_sk"))
```

## Simple

Active Stages (1)

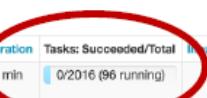
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	7127041251392214025	master_no_lazy.write.mode('overwrite').format("... save at command-885910:1 +details [kill]	2019/03/31 17:11:16	6.2 h	0/200 (96 running)		58.7 GB		



## Extra Shuffle Partitions

Active Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
6	3026732521940153177	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885910:2 +details [kill]	2019/03/31 23:30:54	10 min	0/2016 (96 running)		5.6 GB		



# With Lazy Loading

Cmd 12

```
1 val master_lazy = ss_sub
2 .join(dt_sub
3 .filter(year('d_date).between(2000,2001)),
4 'd_date_sk === 'ss_sold_date_sk')
5 .join(item.withColumnRenamed("i_item_sk", "ss_item_sk"), Seq("ss_item_sk"))
6 .join(inv,
7 'inv_item_sk === 'ss_item_sk &&
8 'inv_date_sk === 'ss_sold_date_sk')
```

## Details for Stage 21 (Attempt 0)

Total Time Across All Tasks: 12.9 h  
 Locality Level Summary: Process local: 200  
 Output: 21.8 GB / 3112776340  
 Shuffle Read: 53.9 GB / 1883116073  
 Shuffle Spill (Memory): 552.0 GB  
 Shuffle Spill (Disk): 67.4 GB

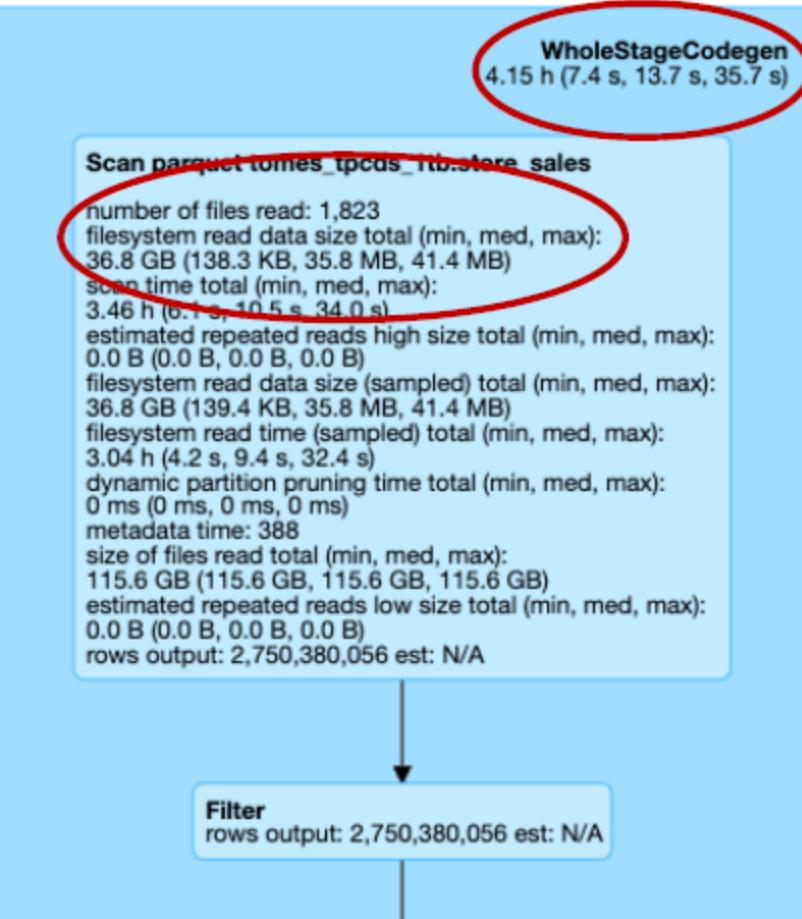
- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

## Summary Metrics for 200 Completed Tasks

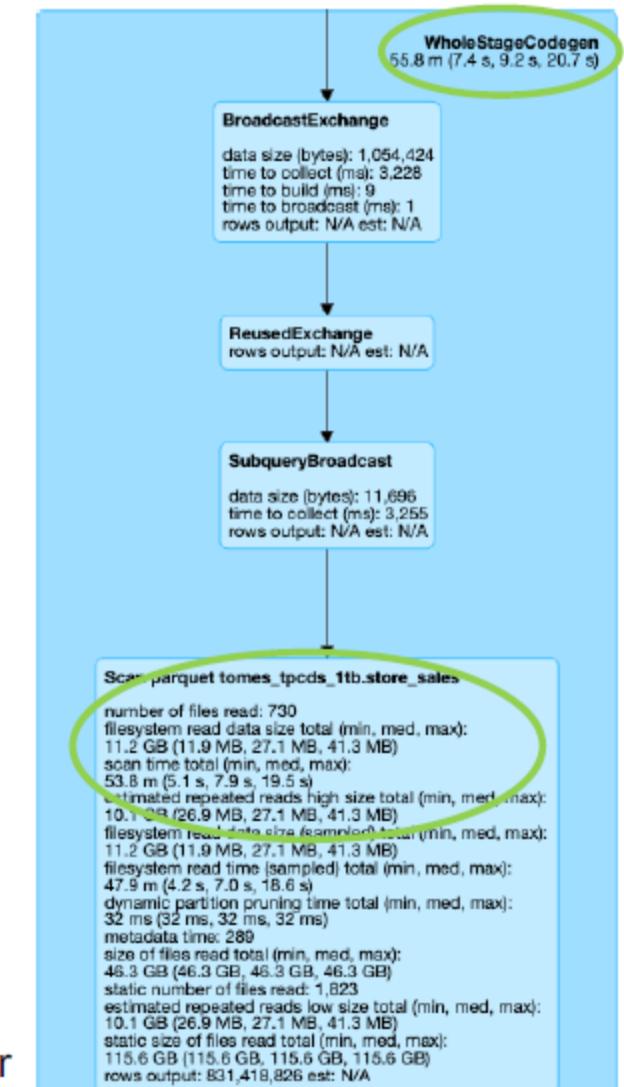
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.7 min	3.8 min	3.9 min	4.2 min	4.6 min
GC Time	2 s	7 s	11 s	23 s	30 s
Output Size / Records	111.0 MB / 15333760	111.6 MB / 15524100	111.7 MB / 15566840	111.9 MB / 15608700	114.0 MB / 15771320
Shuffle Read Size / Records	275.0 MB / 9379740	275.8 MB / 9405412	276.1 MB / 9415012	276.4 MB / 9425519	277.7 MB / 9458918
Shuffle spill (memory)	0.0 B	2.9 GB	2.9 GB	2.9 GB	2.9 GB
Shuffle spill (disk)	0.0 B	358.9 MB	359.7 MB	360.1 MB	360.6 MB

▼ Com

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
21	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-88595:1 +details	2019/03/31 14:17:4	8.5 min	200/200		21.8 GB	53.9 GB	
20	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-88595:1 +details	2019/03/31 14:16:15	27 s	124/124		3.8 GB		8.6 GB
19	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-88595:1 +details	2019/03/31 14:16:15	1.3 min	452/452		14.7 GB		45.4 GB



## Without Partition Filter



## Shrink Partition Range Using a Filter on HIVE Partitioned Column

## With Partition Filter

# Spark Partitions – Types

- Input
  - Controls - Size
    - spark.default.parallelism (don't use)
    - spark.sql.files.maxPartitionBytes (mutable)
      - assuming source has sufficient partitions
- Shuffle
  - Control = Count
    - spark.sql.shuffle.partitions
- Output
  - Control = Size
    - Coalesce(n) to shrink
    - Repartition(n) to increase and/or balance (shuffle)
    - df.write.option("maxRecordsPerFile", N)

## Partitions – Shuffle – Default

Default = 200 Shuffle Partitions

## **Partitions – Right Sizing – Shuffle – Master Equation**

- Largest Shuffle Stage
  - Target Size <= 200 MB/partition
- Partition Count = Stage Input Data / Target Size
  - Solve for Partition Count

## Partitions – Right Sizing – Shuffle – Master Equation

- Largest Shuffle Stage
  - Target Size  $\leq$  200 MB/partition
- Partition Count = Stage Input Data / Target Size
  - Solve for Partition Count

### EXAMPLE

Shuffle Stage Input = 210GB

$$x = 210000\text{MB} / 200\text{MB} = 1050$$

`spark.conf.set("spark.sql.shuffle.partitions", 1050)`

BUT -> If cluster has 2000 cores

`spark.conf.set("spark.sql.shuffle.partitions", 2000)`

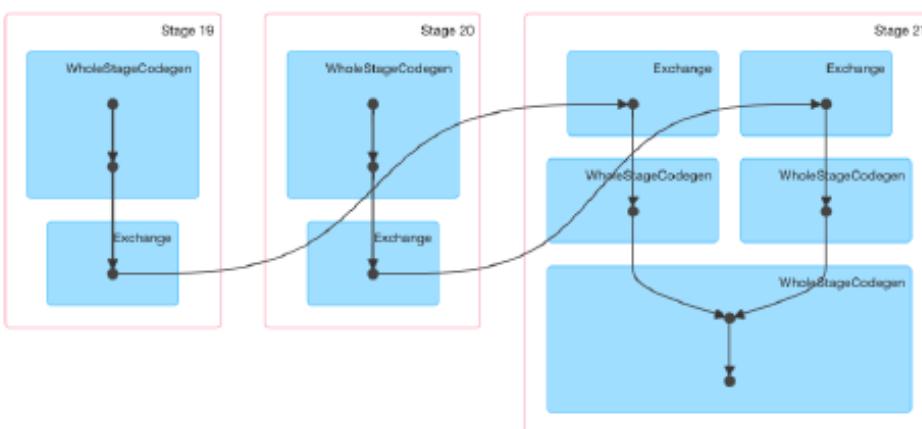
## Cluster Spec

96 cores @ 7.625g/core  
 3.8125g Working Mem  
 3.8125g Storage Mem

### Details for Job 11

Status: SUCCEEDED  
 Associated SQL Query: 95  
 Job Group: 238012049243624904\_7941856623374133785\_21481c000c0214ff2aeebf2d8342a3ea5  
 Completed Stages: 3

► Event Timeline  
 ▼ DAG Visualization



### ▼ Completed Stages (3)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded / Total	Input	Output	Shuffle Read	Shuffle Write
21	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1	+details	2019/03/31 14:17:34	8.5 min	200/200	21.8 GB	53.9 GB	
20	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1	+details	2019/03/31 14:16:15	27 s	131/131	3.8 GB		8.6 GB
19	238012049243624904	master_lazy_item_bc.write.format("parquet").sav... save at command-885895:1	+details	2019/03/31 14:16:15	1.3 min	452/452	14.7 GB		45.4 GB

## Stage 21 -> Shuffle Fed By Stage 19 & 20

THUS

Stage 21 Shuffle Input = 45.4g + 8.6g == 54g

Default Shuffle Partition == 200 == 54000mb/200parts ==~ 270mb/shuffle part

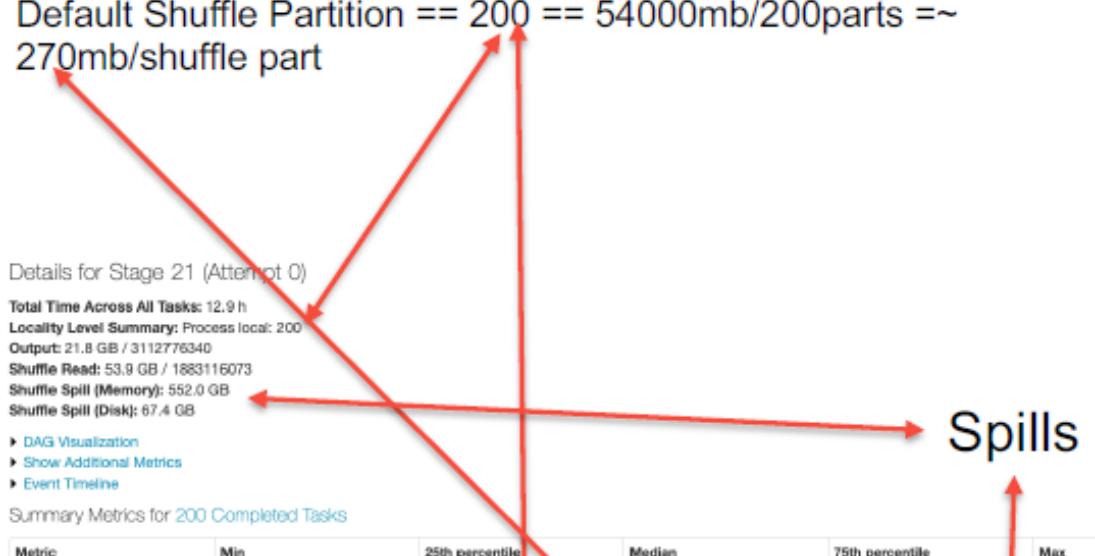
### Details for Stage 21 (Attempt 0)

Total Time Across All Tasks: 12.9 h  
 Locality Level Summary: Process local: 200  
 Output: 21.8 GB / 3112776340  
 Shuffle Read: 53.9 GB / 1883116073  
 Shuffle Spill (Memory): 552.0 GB  
 Shuffle Spill (Disk): 67.4 GB

► DAG Visualization  
 ▶ Show Additional Metrics  
 ▶ Event Timeline

### Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.7 min	3.8 min	3.9 min	4.2 min	4.6 min
GC Time	2 s	7 s	11 s	23 s	30 s
Output Size / Records	111.0 MB / 15333760	111.6 MB / 15524100	111.7 MB / 15566840	111.9 MB / 15608700	114.0 MB / 15771320
Shuffle Read Size / Records	275.0 MB / 9379740	275.8 MB / 9405412	276.1 MB / 9415012	276.4 MB / 9425519	277.7 MB / 9458918
Shuffle spill (memory)	0.0 B	2.9 GB	2.9 GB	2.9 GB	2.9 GB
Shuffle spill (disk)	0.0 B	358.9 MB	359.7 MB	360.1 MB	360.6 MB



## Cluster Spec

96 cores @ 7.625g/core

3.8125g Working Mem

3.8125g Storage Mem

```
1 spark.conf.set("spark.sql.shuffle.partitions", 480)
```

### Completed Stages (3)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
37	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details)	2019/03/31 14:53:45	7.8 min	480/480		19.9 GB	54.0 GB	
36	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details)	2019/03/31 14:52:25	25 s	131/131	3.8 GB			8.7 GB
35	238012049243624904	spark.conf.set("spark.sql.shuffle.partitions", ... save at command-885895:2 +details)	2019/03/31 14:52:25	1.3 min	452/452	14.7 GB			45.3 GB

480 shuffle partitions – WHY?

Target shuffle part size == 100m

p = 54g / 100m == 540

540p / 96 cores == 5.625

96 \* 5 == 480

If p == 540 another 60p have to be loaded  
and processed after first cycle is complete

#### Details for Stage 37 (Attempt 0)

Total Time Across All Tasks: 11.6 h

Locality Level Summary: Process local: 480

Output: 19.9 GB / 3112776340

Shuffle Read: 54.0 GB / 1883116073

- ▶ DAG Visualization
- ▶ Show Additional Metrics
- ▶ Event Timeline

## NO SPILL

#### Summary Metrics for 480 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1.2 min	1.5 min	1.5 min	1.6 min	1.8 min
GC Time	0.8 s	2 s	2 s	3 s	12 s
Output Size / Records	41.5 MB / 5362420	42.2 MB / 6456400	42.5 MB / 6484840	42.7 MB / 6515940	43.4 MB / 6601140
Shuffle Read Size / Records	114.3 MB / 3895329	115.0 MB / 3917334	115.2 MB / 3923186	115.4 MB / 3929412	115.9 MB / 3948584

# Input Partitions – Right Sizing

- Use Spark Defaults (128MB) unless...
  - Increase Parallelism
  - Heavily Nested/Repetitive Data
  - Generating Data – i.e. Explode
  - Source Structure is not optimal (upstream)
  - UDFs

```
spark.conf.set("spark.sql.files.maxPartitionBytes", 16777216)
```

Cmd 48

```
1 spark.conf.set("spark.sql.files.maxPartitionBytes", 134217728) 128mb
2 val master_source_10p = spark.read.parquet("/tmp/tomes/scratch/SAIS19/10p_source")
3 master_source_10p.rdd.partitions.size
```

▶ (1) Spark Jobs

```
▶ [] master_source_10p: org.apache.spark.sql.DataFrame = [ss_item_sk: integer, ss_customer_sk: integer ... 33 m
master_source_10p: org.apache.spark.sql.DataFrame = [ss_item_sk: int, ss_customer_sk: in
res34: Int = 90
```

```
1 spark.conf.set("spark.sql.files.maxPartitionBytes", 1024 * 1024 * 16) 16mb
2 val master_source_10p = spark.read.parquet("/tmp/tomes/scratch/SAIS19/10p_source")
3 master_source_10p.rdd.partitions.size
```

▶ (1) Spark Jobs

```
▶ [] master_source_10p: org.apache.spark.sql.DataFrame = [ss_item_sk: integer, ss_customer_sk: integer ... 33 r
master_source_10p: org.apache.spark.sql.DataFrame = [ss_item_sk: int, ss_customer_sk: i
res36: Int = 710
```

Stage	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
Id	Metric	Min	25th percentile	Median	75th percentile	Max	
3	7427136891458074341	spark.conf.set("spark.sql.files.maxPartitionBy... <a href="#">save at command-899638:4</a>	2019/04/23 13:49:35	4.6 min	90/90	11.9 GB	24.1 GB
	Metric	Min	25th percentile	Median	75th percentile	Max	
	Duration	2.9 min	3.8 min	4.4 min	4.5 min	4.5 min	
	GC Time	1 s	1 s	1 s	2 s	2 s	
	Input Size / Records	105.9 MB / 26344585	138.9 MB / 35511851	139.0 MB / 35544755	139.1 MB / 35576807	139.4 MB / 35647533	
	Output Size / Records	213.8 MB / 26344585	280.5 MB / 35511851	281.1 MB / 35544755	281.5 MB / 35576807	282.6 MB / 35647533	
	Metric	Min	25th percentile	Median	75th percentile	Max	
	Duration	6 s	1.1 min	1.5 min	1.6 min	2.5 min	
	GC Time	48 ms	3 s	4 s	5 s	6 s	
	Input Size / Records	11.0 MB / 0	26.9 MB / 4414822	27.0 MB / 4441791	27.1 MB / 4471397	27.4 MB / 4537083	
	Output Size / Records	0.0 B / 0	25.1 MB / 4414822	25.4 MB / 4441791	25.6 MB / 4471397	26.2 MB / 4537083	

Stage	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
Id	Metric	Min	25th percentile	Median	75th percentile	Max	
1	7427136891458074341	spark.conf.set("spark.sql.files.maxPartitionBy... <a href="#">save at command-899639:4</a>	2019/04/23 13:46:34	2.9 min	710/710	18.6 GB	17.4 GB
	Metric	Min	25th percentile	Median	75th percentile	Max	
	Duration	6 s	1.1 min	1.5 min	1.6 min	2.5 min	
	GC Time	48 ms	3 s	4 s	5 s	6 s	
	Input Size / Records	11.0 MB / 0	26.9 MB / 4414822	27.0 MB / 4441791	27.1 MB / 4471397	27.4 MB / 4537083	
	Output Size / Records	0.0 B / 0	25.1 MB / 4414822	25.4 MB / 4441791	25.6 MB / 4471397	26.2 MB / 4537083	

# Output Partitions – Right Sizing

- Write Once -> Read Many
  - More Time to Write but Faster to Read
- Perfect writes limit parallelism
  - Compactions (minor & major)

Write Data Size = 14.7GB

Desired File Size = 1500MB

Max write stage parallelism = 10

96 – 10 == 86 cores idle during write

# Only 10 Cores Used

## Completed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
113	9020956004722677074	val item_bc = broadcast(item.withColumnRenamed(... save at command-885938:10 +details	2019/03/31 19:51:10	14 min	10/10	18.9 GB	14.3 GB		

Cmd 21

```
1 getSizeInfo(s"$s3Prefix/store_sales_2y_10p")
```

▶ (4) Spark Jobs  
Table store\_sales\_2y\_10p has 10 files and average 1.5304093662 gb with stddev of 250 mb  
Command took 0.61 seconds -- by daniel.tomes@databricks.com at 3/31/2019, 2:21:49 PM on Tomes\_Base

Average File Size == 1.5g

# All 96 Cores Used

## Completed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
97	9020956004722677074	val item_bc = broadcast(item.withColumnRenamed(... save at command-885971:10 +details	2019/03/31 19:43:41	1.8 min	96/96	15.1 GB	14.3 GB		

Cmd 22

```
1 getSizeInfo(s"$s3Prefix/store_sales_2y_96p")
```

▶ (4) Spark Jobs  
Table store\_sales\_2y\_10p has 96 files and average 0.15996809254166666 gb with stddev of 35 mb  
Command took 2.00 seconds -- by daniel.tomes@databricks.com at 3/31/2019, 3:46:45 PM on Tomes\_Base

Average File Size == 0.16g

# Output Partitions – Composition

- `df.write.option("maxRecordsPerFile", n)`
- `df.coalesce(n).write...`
- `df.repartition(n).write...`
- `df.repartition(n, [colA, ...]).write...`
- `spark.sql.shuffle.partitions(n)`
- `df.localCheckpoint(...).repartition(n).write...`
- `df.localCheckpoint(...).coalesce(n).write...`

# Partitions – Why So Serious?

- Avoid The Spill
- Maximize Parallelism
  - Utilize All Cores
  - Provision only the cores you need

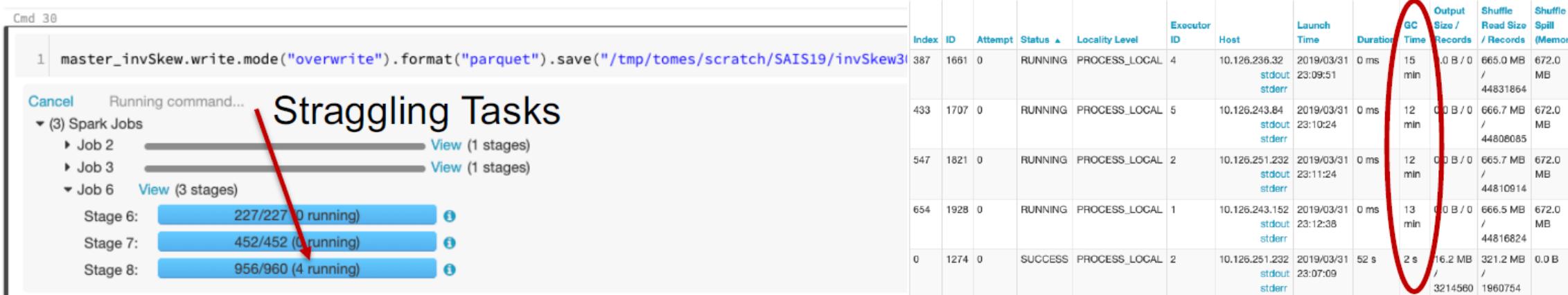
# Advanced Optimizations

- Finding Imbalances
- Persisting
- Join Optimizations
- Handling Skew
- Expensive Operations
- UDFs
- Multi-Dimensional Parallelism

# Balance

- Maximizing Resources Requires Balance
  - Task Duration
  - Partition Size
- SKEW
  - When some partitions are significantly larger than most

Input Partitions  
Shuffle Partitions  
Output Files  
Spills  
GC Times



75<sup>th</sup> percentile ~ 2m recs

max ~ 45m recs

stragglers take > 22X longer IF no spillage

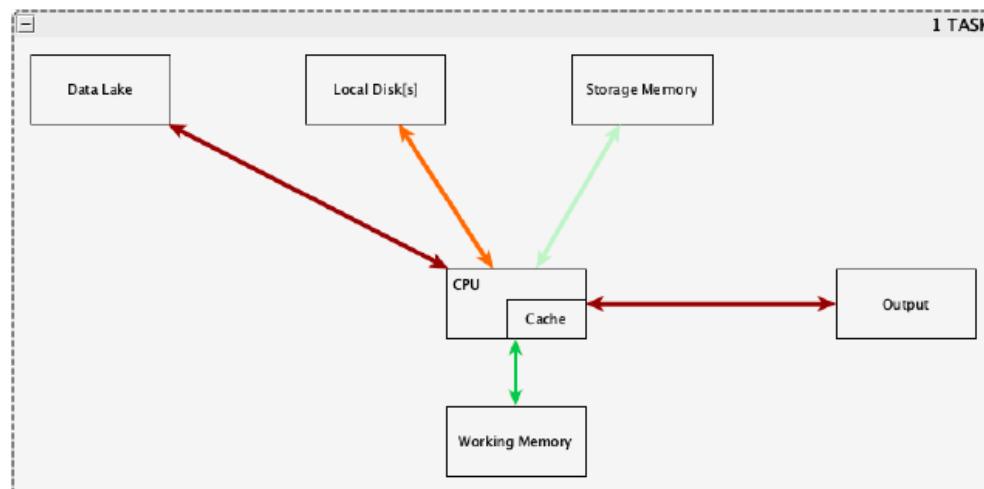
With spillage, 100Xs longer

#### Summary Metrics for 956 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0 ms	40 s	47 s	52 s	3.7 min
GC Time	0.3 s	0.8 s	1 s	4 s	3.2 min
Output Size / Records	0.0 B / 0	16.1 MB / 3219340	16.3 MB / 3242100	16.5 MB / 3263540	17.0 MB / 3323340
Shuffle Read Size / Records	317.2 MB / 1940846	320.7 MB / 1957433	321.6 MB / 1961754	322.4 MB / 1965647	666.7 MB / 44831864
Shuffle spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	672.0 MB
Shuffle spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	91.5 MB

# Minimize Data Scans (Persistence)

- Persistence
  - Not Free
- Repetition
  - SQL Plan



`df.cache == df.persist(StorageLevel.MEMORY_AND_DISK)`

- Types

- Default (MEMORY\_AND\_DISK)
  - Deserialized
- Deserialized = Faster = Bigger
- Serialized = Slower = Smaller
- \_2 = Safety = 2X bigger
- MEMORY\_ONLY
- DISK\_ONLY

Don't Forget To Cleanup!  
`df.unpersist`

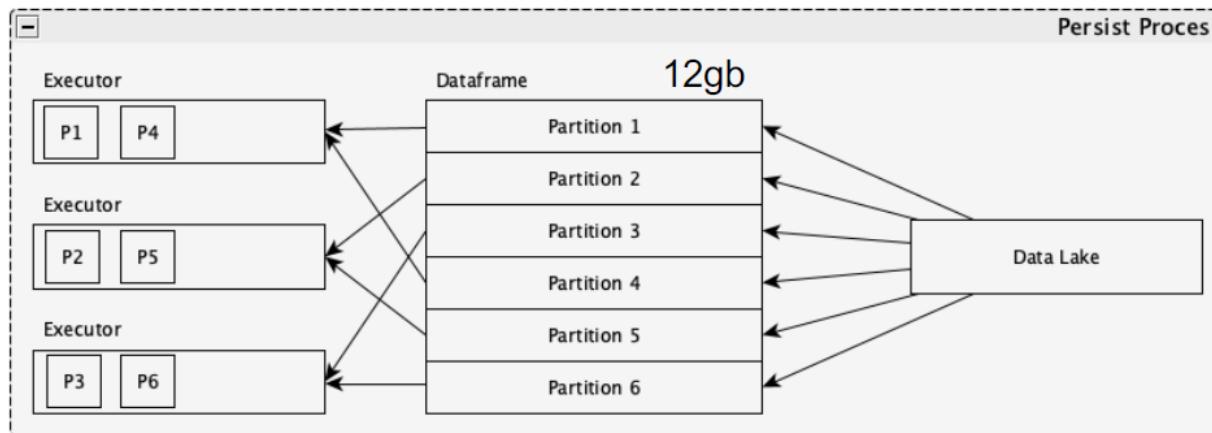
# Join Optimization

- SortMergeJoins (Standard)
- Broadcast Joins (Fastest)
- Skew Joins
- Range Joins
- BroadcastedNestedLoop Joins (BNLJ)

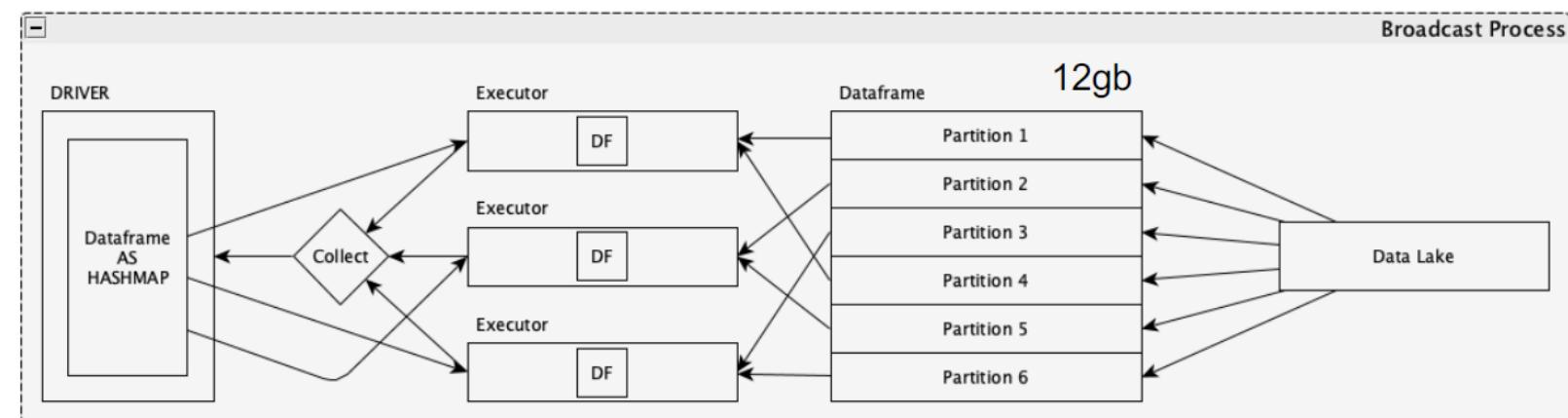
# Join Optimization

- SortMerge Join – Both sides are large
- Broadcast Joins – One side is small
  - Automatic If:  
(one side < `spark.sql.autoBroadcastJoinThreshold`) (default 10m)
  - Risks
    - Not Enough Driver Memory
    - DF > `spark.driver.maxResultSize`
    - DF > Single Executor Available Working Memory
  - Prod – Mitigate The Risks
    - Validation Functions

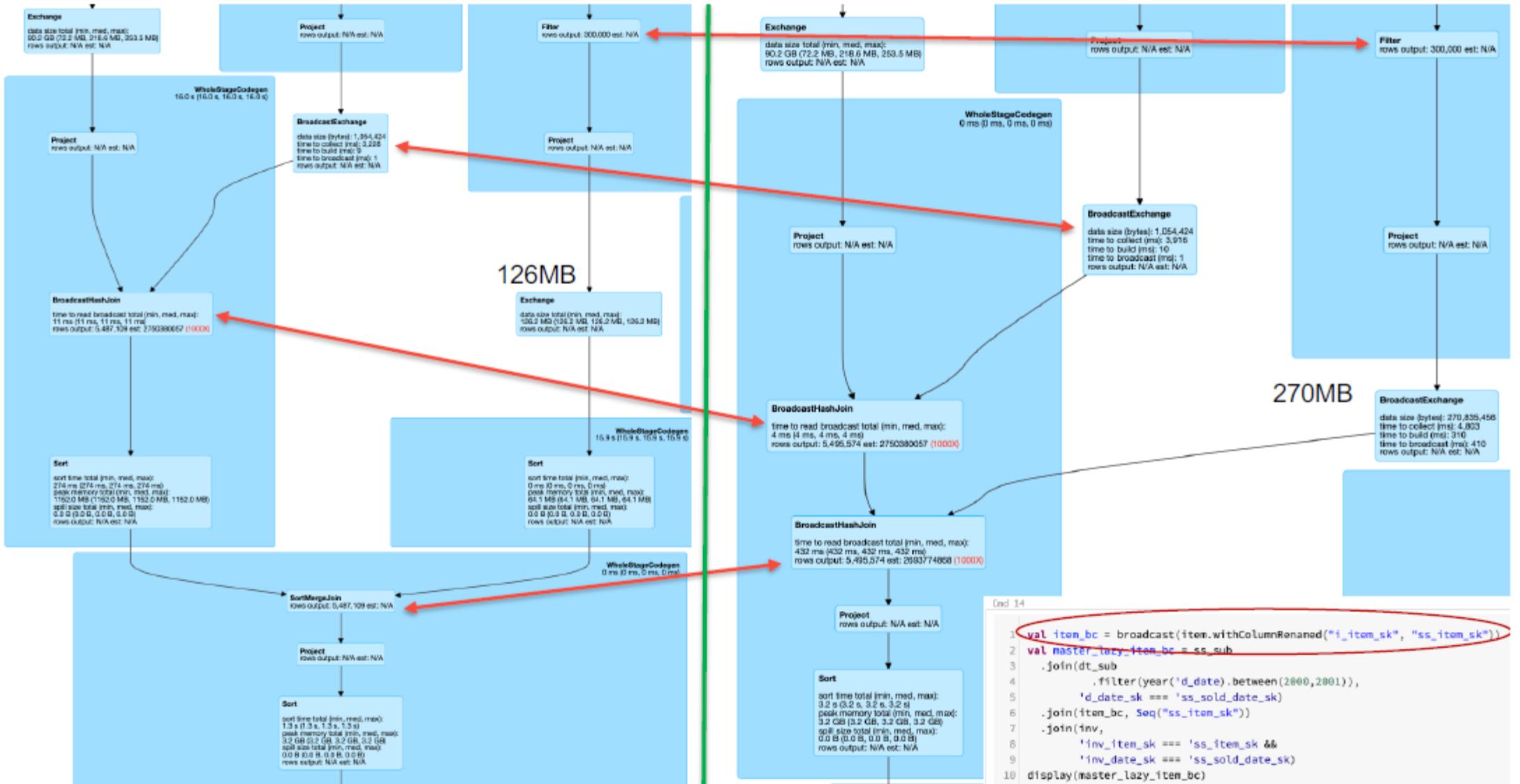
# Persistence Vs. Broadcast



Attempt to send compute to the data



Data availability guaranteed ->  
each executor has entire dataset



From 6h and barely started  
TO 8m → Lazy Loading  
TO 2.5m → Broadcast

Cmd 31 (+)

```
1 val master_lazy = ss_sub
2 .join(dt_sub
3 .filter(year('d_date').between(2000,2001)),
4 'd_date_sk === 'ss_sold_date_sk')
5 .joinbroadcast(item.withColumnRenamed("i_item_sk", "ss_item_sk")), Seq("ss_item_sk"))
6 .join(inv,
7 'inv_item_sk === 'ss_item_sk &&
8 'inv_date_sk === 'ss_sold_date_sk)
```

▼Completed Stages (3)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
36	4260198380812558228	master_lazy.write.format("parquet").mode("overw... save at command-885896:1 +details	2019/04/19 17:42:45	2.5 min	<b>200/200</b>		21.8 GB	55.6 GB	
35	4260198380812558228	master_lazy.write.format("parquet").mode("overw... save at command-885896:1 +details	2019/04/19 17:42:02	43 s	<b>452/452</b>	14.7 GB			47.1 GB
34	4260198380812558228	master_lazy.write.format("parquet").mode("overw... save at command-885896:1 +details	2019/04/19 17:42:02	21 s	<b>261/261</b>	3.8 GB			8.6 GB

**SQL Example:**

SELECT /\*+ BROADCAST(customers) \*/ \* FROM customers, orders WHERE o\_custId = c\_custId

# Skew Join Optimization

Code 5

```
1 val skewedVals = invWSkew.withColumn("key", concat('inv_item_sk, 'inv_date_sk))
2 .groupBy("key", "inv_item_sk", "inv_date_sk")
3 .agg(count("key").alias("cnt"))
4 .orderBy('cnt.desc)
```

Code 6

key	inv_item_sk	inv_date_sk	cnt
924052452275	92405	2452275	42872883
924042452275	92404	2452275	42867283
924072452275	92407	2452275	42857248
924062452275	92406	2452275	42854886
924032452275	92403	2452175	42853282
924012452275	92401	2452275	42862880
924022452275	92402	2452475	42851558
565032452341	56503	2452341	20
119012452341	119011	2452341	20
2928412452341	292841	2452341	20
2283962452341	228396	2452341	20

- OSS Fix - Salting

- Add Column to each side with random int between 0 and spark.sql.shuffle.partitions – 1 to both sides
- Add join clause to include join on generated column above
- Drop temp columns from result

- Databricks Fix (Skew Join)

```
val skewedKeys = List("id1", "id200", "id-99")
df.join(
 skewDF.hint("SKEW", "skewKey", skewedKeys),
 Seq(keyCol), "inner")
```

# Oh My God!! Is my Data Skewed ?



# Skewed Aggregates

```
df.groupBy("city", "state").agg(<f(x)>).orderBy(col.desc)
```

```
val saltVal = random(0, spark.conf.get(org...shuffle.partitions) - 1)
```

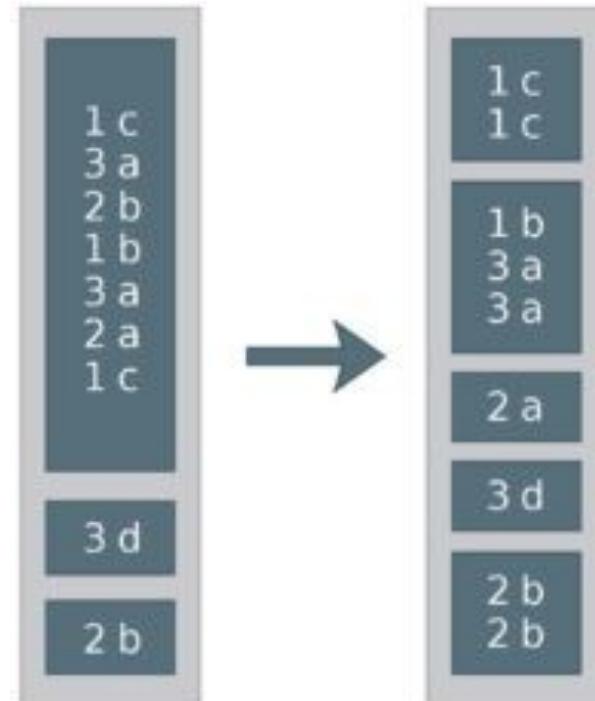
```
df.withColumn("salt", lit(saltVal))
 .groupBy("city", "state", "salt")
 .agg(<f(x)>)
 .drop("salt")
 .orderBy(col.desc)
```

# Few ways to solve:-

- Repartition
- Salting Technique
- Isolating Salting
- Isolating Map Join
- Iterative broadcast Join

# Re-partition

Repartition helps us in fighting the skewness issue to some extent but not completely.



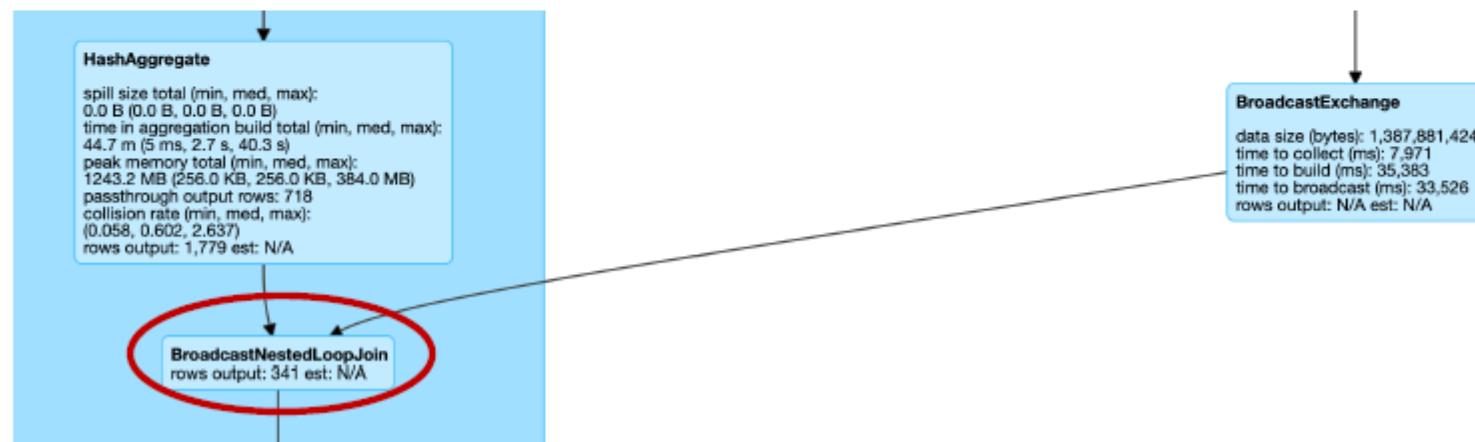
# Salting Technique: we have salted the key to make it salted

- consider you have datasets consist of 4 unique keys out of which 1st key is having 2000 records ,2nd key is having 500 records ,3rd and 4th key is having 50 and 30 records respectively.

*Normal Key: “Foo”*

*Salted Key: “Foo” + Random Integer*

# BroadcastNestedLoopJoin (BNLJ)



```

1 SELECT distinct sys_cd_val , PI_NUM FROM retail_sales
2 WHERE loadTS IN (SELECT MAX(loadTS) FROM retail_sales)
3 AND PI_NUM NOT IN (SELECT distinct PI_13 FROM product_master)
4 AND sys_cd_val <> 'PWG'

```

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
47	3925260204415598680	SELECT distinct SRC_SYS_CD , PIN_NUM FROM edl... collectResult at OutputAggregator.scala:136 +details	2019/04/22 18:56:06	7.1 min	557/563 (6)	7.8 GB				Job 17 cancelled part of cancelled job 3925260204415598680_50868365920

```

1 SELECT distinct sys_cd_val , PI_NUM
2 FROM retail_sales
3 WHERE loadTS IN (
4 SELECT MAX(loadTS) FROM retail_sales
5)
6 AND NOT exists (
7 SELECT 1
8 FROM product_master sub
9 where sub.PI_13 <= a.PI_NUM
0)
1 AND sys_cd_val <> 'PWG'

```

Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
019/04/22 19:04:34	0.1 s	1/1			7.1 KB	
019/04/22 19:04:06	28 s	200/200			674.0 MB	1062.5 KB
019/04/22 19:03:51	16 s	569/569	6.6 GB			148.8 MB
019/04/22 19:03:50	14 s	12/12				525.2 MB

# Range Join Optimization

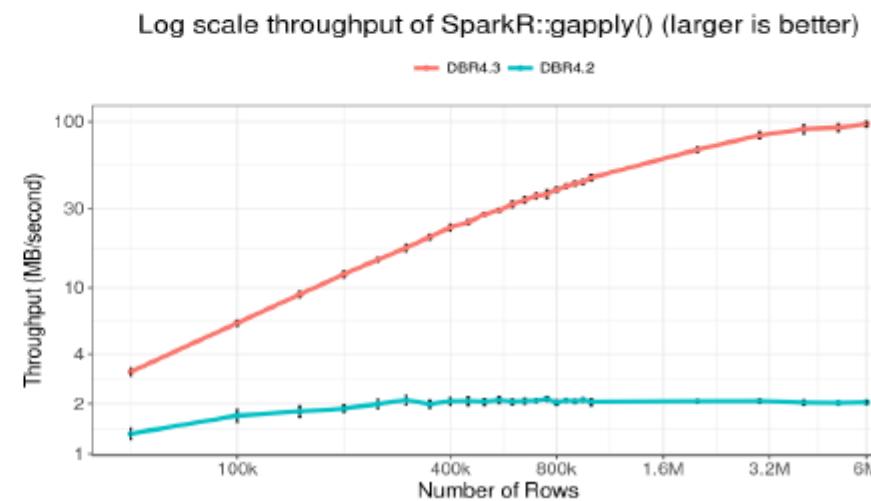
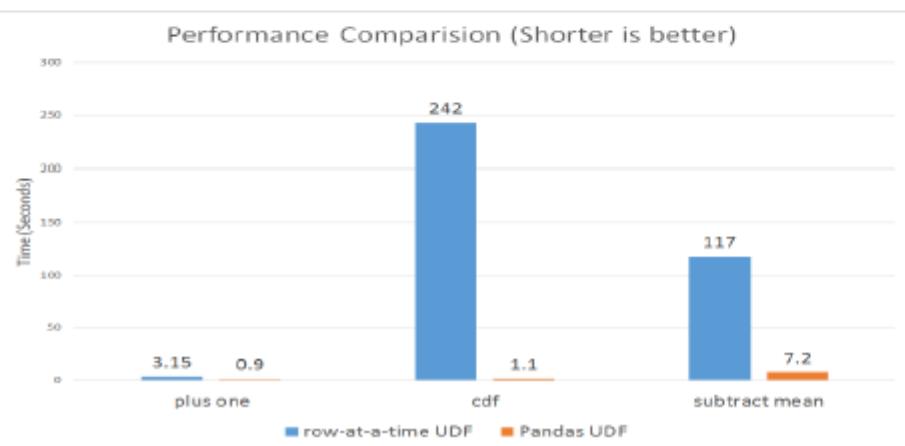
- Range Joins Types
  - Point In Interval Range Join
    - Predicate specifies value in one relation that is between two values from the other relation
  - Interval Overlap Range Join
    - Predicate specifies an overlap of intervals between two values from each relation

# Omit Expensive Ops

- Repartition
  - Use Coalesce or Shuffle Partition Count
- Count – Do you really need it?
- DistinctCount
  - use approxCountDistinct()
- If distincts are required, put them in the right place
  - Use dropDuplicates
  - dropDuplicates BEFORE the join
  - dropDuplicates BEFORE the groupBy

# UDF Penalties

- Traditional UDFs cannot use [Tungsten](#)
  - Use [org.apache.spark.sql.functions](#)
  - Use [PandasUDFs](#)
    - Utilizes [Apache Arrow](#)
  - Use [SparkR UDFs](#)



# Data Locality

Data locality is how close data is to the code processing it. There are several levels of locality based on the data's current location. In order from closest to farthest:

**PROCESS\_LOCAL:** data is in the same JVM as the running code. This is the best locality possible

**NODE\_LOCAL:** data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS\_LOCAL because the data has to travel between processes

**NO\_PREF** data: is accessed equally quickly from anywhere and has no locality preference

**RACK\_LOCAL:** data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch

**ANY:** data is elsewhere on the network and not in the same rack

# On-Heap memory and Off-Heap memory

Executor acts as a JVM process, and its memory management is based on the JVM. So JVM memory management includes two methods:

- On-Heap memory management: Objects are allocated on the JVM heap and bound by GC.
- Off-Heap memory management: Objects are allocated in memory outside the JVM by serialization, managed by the application, and are not bound by GC. This memory management method can avoid frequent GC, but the disadvantage is that you have to write the logic of memory allocation and memory release.

# On-Heap memory and Off-Heap memory

In general, the objects' read and write speed is:

on-heap > off-heap > disk

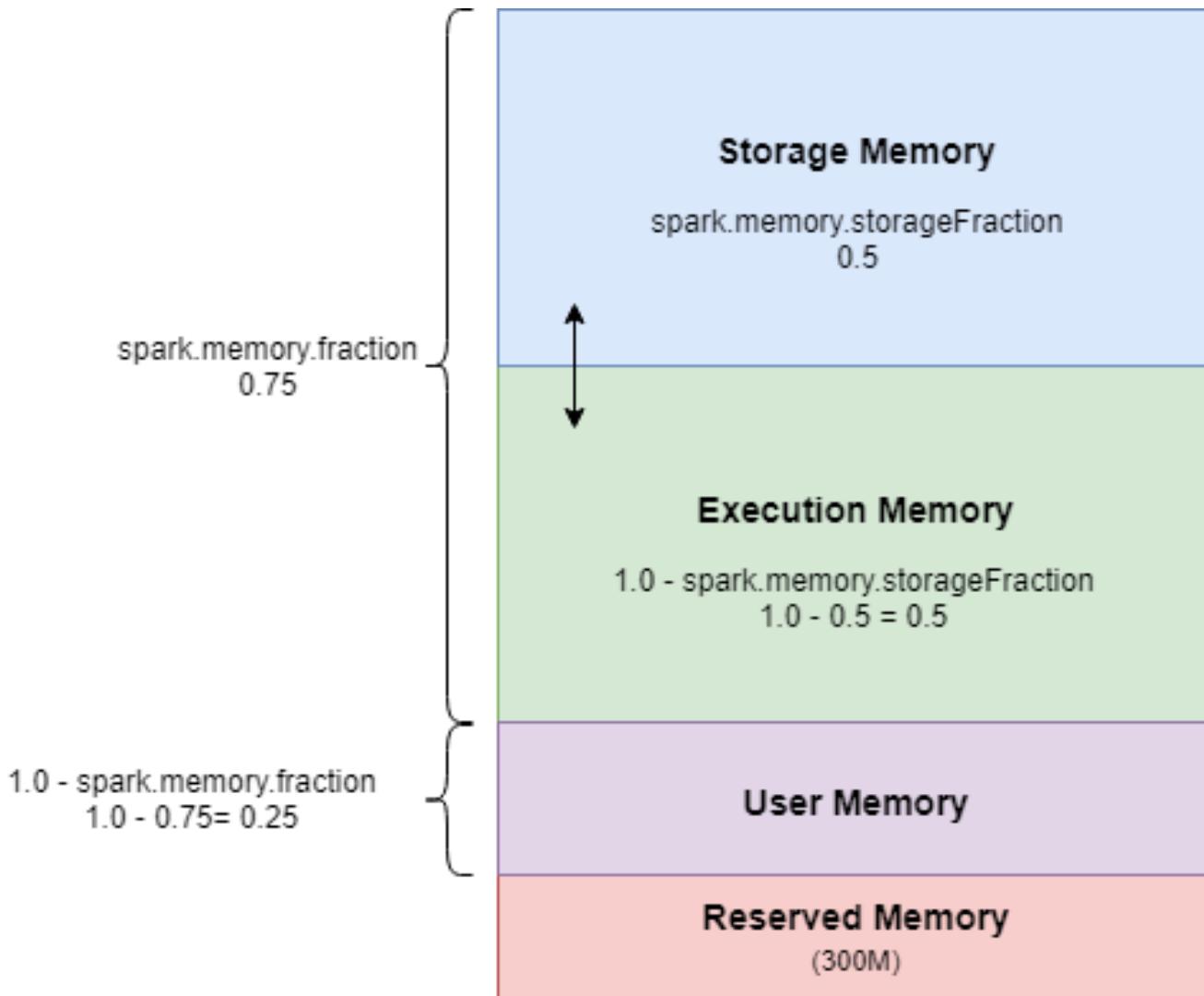
# On-heap model

- The size of the On-heap memory is configured by the --executor-memory or spark.executor.memory parameter when the Spark Application starts.

The On-heap memory area in the Executor can be roughly divided into the following four blocks:

1. Storage Memory
2. Execution Memory
3. User Memory
4. Reserved Memory

# Memory Distribution



# *m4.2xlarge instance (32GB Memory)*

## **YARN**

*yarn.nodemanager.resource.memory-mb* (24.5 GB)

### **YARN Container 1**

#### **Container Memory**

*spark.yarn.executor.memoryOverhead*  
= max(384, 0.1 \* Container Memory)

*spark.executor.memory*  
= 0.9 \* Container Memory

#### **Usable Memory**

**Execution Memory**      **Storage Memory**

#### **User Memory**

**Reserved Memory**  
(300MB hardcoded)

**Spark Executor 1**

### **YARN Container 2**

#### **Container Memory**

*spark.yarn.executor.memoryOverhead*  
= max(384, 0.1 \* Container Memory)

*spark.executor.memory*  
= 0.9 \* Container Memory

#### **Usable Memory**

**Execution Memory**      **Storage Memory**

#### **User Memory**

**Reserved Memory**  
(300MB hardcoded)

**Spark Executor 2**

**Core 1**

**Core 2**

# Advanced Parallelism

- Spark's Three Levels of Parallelism
  - Driver Parallelism
  - Horizontal Parallelism
  - Executor Parallelism

```
1 val origHist_1y = spark.read.table(origHistTable)
2
3 spark.conf.set("spark.sql.shuffle.partitions", 500)
4
5 private val taskSupport = new ForkJoinTaskSupport(new ForkJoinPool(parallelism))
6 case class monthPart(year: Int, month: Int)
7
8 private val parts: ArrayBuffer[monthPart] = ArrayBuffer[monthPart]()
9 Range(2018,2019).toArray.foreach(yr => {
10 Range(1,13).toArray.foreach(mnth => parts.append(monthPart(yr, mnth)))
11 })
12
13 val parMonth = parts.toArray.par
14
15 parMonth.tasksupport = taskSupport
16 parMonth.foreach(part =>
17 try {
18 origHist_1y.filter('end_yr === part.year && 'end_mnth === part.month)
19 .write.format("delta")
20 .mode("append")
21 .partitionBy("end_cptr_dt")
22 .save(histPerfPath_1y)
23 } catch {
24 case e: Throwable => println(s"ERROR: Failed on ${part.year}-${part.month} with, $e")
25 }
26 println(s"Completed ${part.year}-${part.month}")
27 })|
```