# Spark, Scala and Project 2

CS 143 Introduction to Database Systems

TA: Jin Wang and Mingda Li

02/28/2020

# Outline

- More tips about Project 2
- Apache Spark
- The Scala Language

# Project 2 Related

- Skeleton codes: CS143Utils, basicOperators, DiskHashedRelation, SpillableAggregate.scala
- Some useful functions are already contained in CS143Utils and DiskHashedRelation. (Understand them and save time!)
- Other useful files: Aggregate.scala, SparkPlan.scala, basicOperators.scala
- Majority of code you need to look at is contained in the package: **sql/core/src/main/scala/org/apache/spark/sql/execution**

# Iterator Interface

- Iterator provides an interface to collections that enforces a specific API: the **next** and **hasNext** functions.
- Some tasks ask you to **implement the function** to generate a new iterator given an iterator as the argument.

```
def generateIterator(args): (Iterator[Row] => Iterator[Row]) = {
    /* Type of input is Iterator[Row] */
    input => {
        new Iterator[Row] {
            /* Use input.next and input.hasNext to help implement the new iterator */
            def hasNext() => …
            def next() => …
        }
    }
}
```

# Task A

- Disk-based hashing
  - Basic idea: refer to contents in week 6 discussion
  - Hint: all three methods can be implemented with the exisitng functions
  - Regarding Task 2
    - Use keyGenerator class
    - Do not forget to close input of each partition
- UDF Caching
  - Iterator Generator: add memberships with type defined in CS143Utils and DiskHashedRelation
  - Understand the way how Iterator works will make it much easier!

# Task B

- Hash based aggregation
  - Basic idea: Hybrid hashing
    - Refer to contents in week 7 discussion
  - Learn from the *Aggregate.scala* file to understand the organization of this class
  - You only need to implement the in-memory part
  - Ignore the functions related to disk operations
    - e.g. initSpills, spillRecord, fetchSpill

# General Tips

- If you use IntelliJ IDE, make sure to configure your environment well
  - Java version
  - Scala plugin
- Try to use the provided APIs instead of creating new ones by yourself
- Generate the .zip file for submission with the script provided by us
  - NEVER manually add files into it!

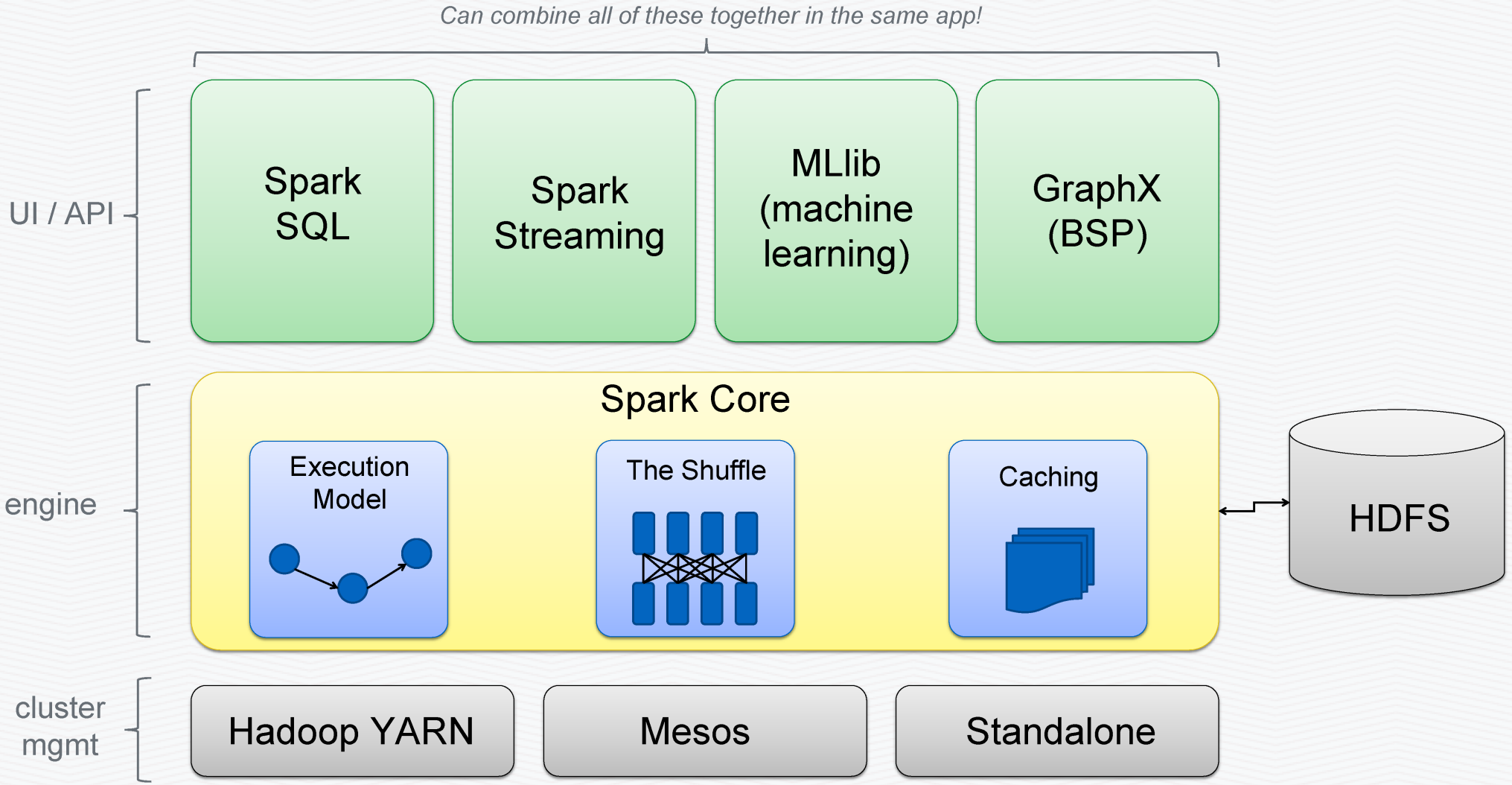
# Outline

- More tips about Project 2
- Apache Spark
- The Scala Language

# Spark Basics

- Spark helps us tackle problems too big for a single machine!
- Traditional distributed computing platforms scale well but have limited APIs (map/reduce)
- On the other hand, Spark has an expressive data focused API that makes writing large scale programs easy.
- Supports built-in APIs in multiple languages:
  - Spark originally written in Scala.
  - Java API added for standalone applications.
  - Python API added recently along with interactive shell.

# Spark Components




Slide taken from: https://www.slideshare.net/thelabdude/apachecon-na-2015-spark-solr-integration

# Resilient Distributed Datasets (RDD) (Key Abstraction of Data Parallelization in Spark)

- RDD is an **immutable (read-only) distributed collection of records**.
- Each dataset in RDD is divided (**partitioned**) into **logical partitions**, which may be computed on different nodes of the cluster.
- RDDs can be created through operations **such as map, filter or reduce on either data on stable storage or other RDDs**.
- RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- RDD is represented by various RDD objects in Spark.

# RDD Example (Word count)

- Spark Program typically starts with the creation of an RDD for data reading (e.g. **HadoopRDD**)
- User specifies a series of **transformations** (e.g. map, reduce, groupBy) on exisiting RDDs to create new RDDs.
- Until an **action** (e.g. count, collect, save) is called, the RDD DAG gets analyzed and optimized, delivered to worker nodes for execution. (**Lazy Evaluation**)

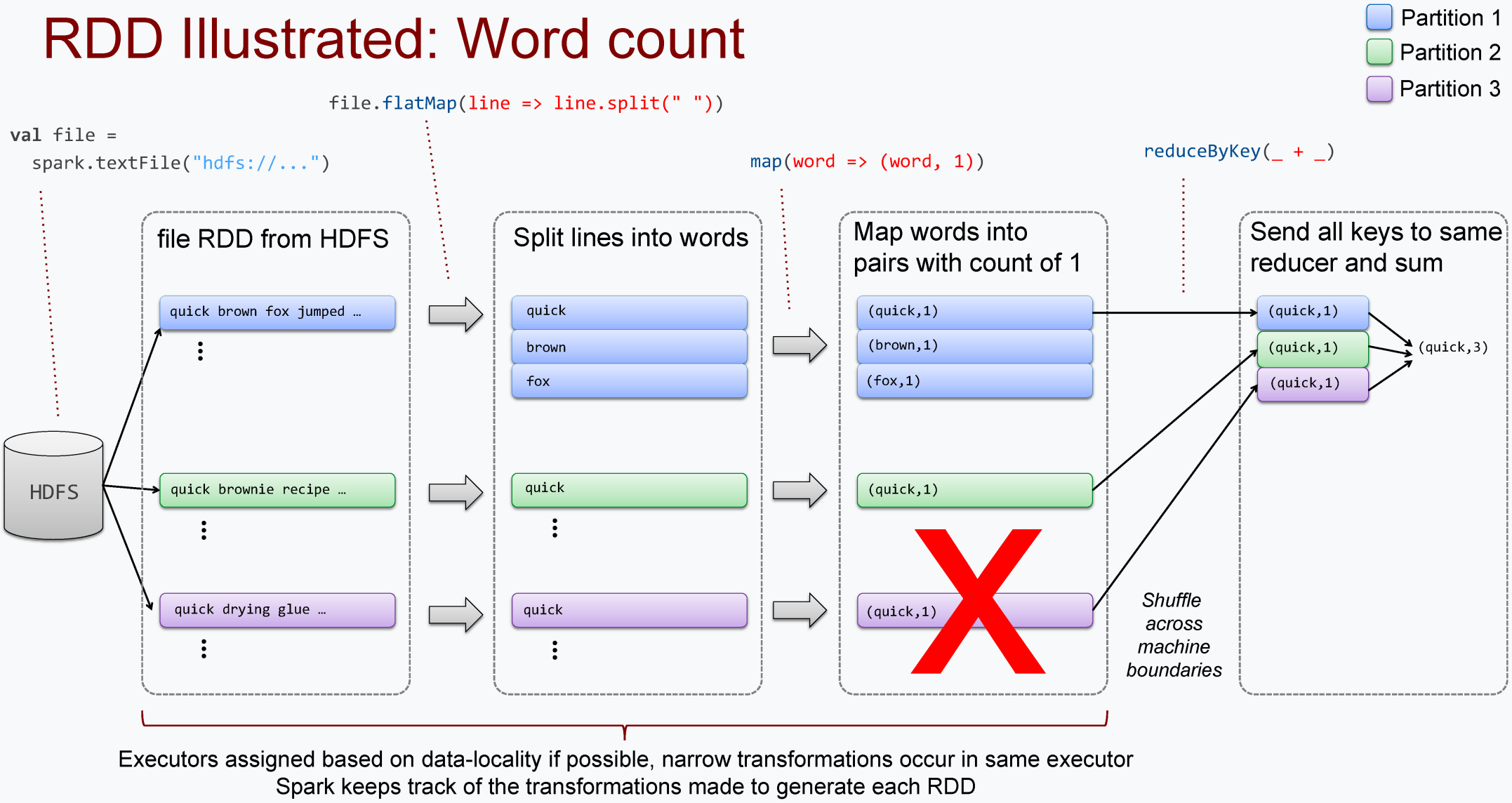
```
// HadoopRDD created
val file = spark.textFile("hdfs://...")


// FlatMappedRDD, MappedRDD, ShuffledRDD created
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)


// Only until an RDD is called, the DAG gets
executed in parallel
counts.saveAsTextFile("hdfs://...")
```

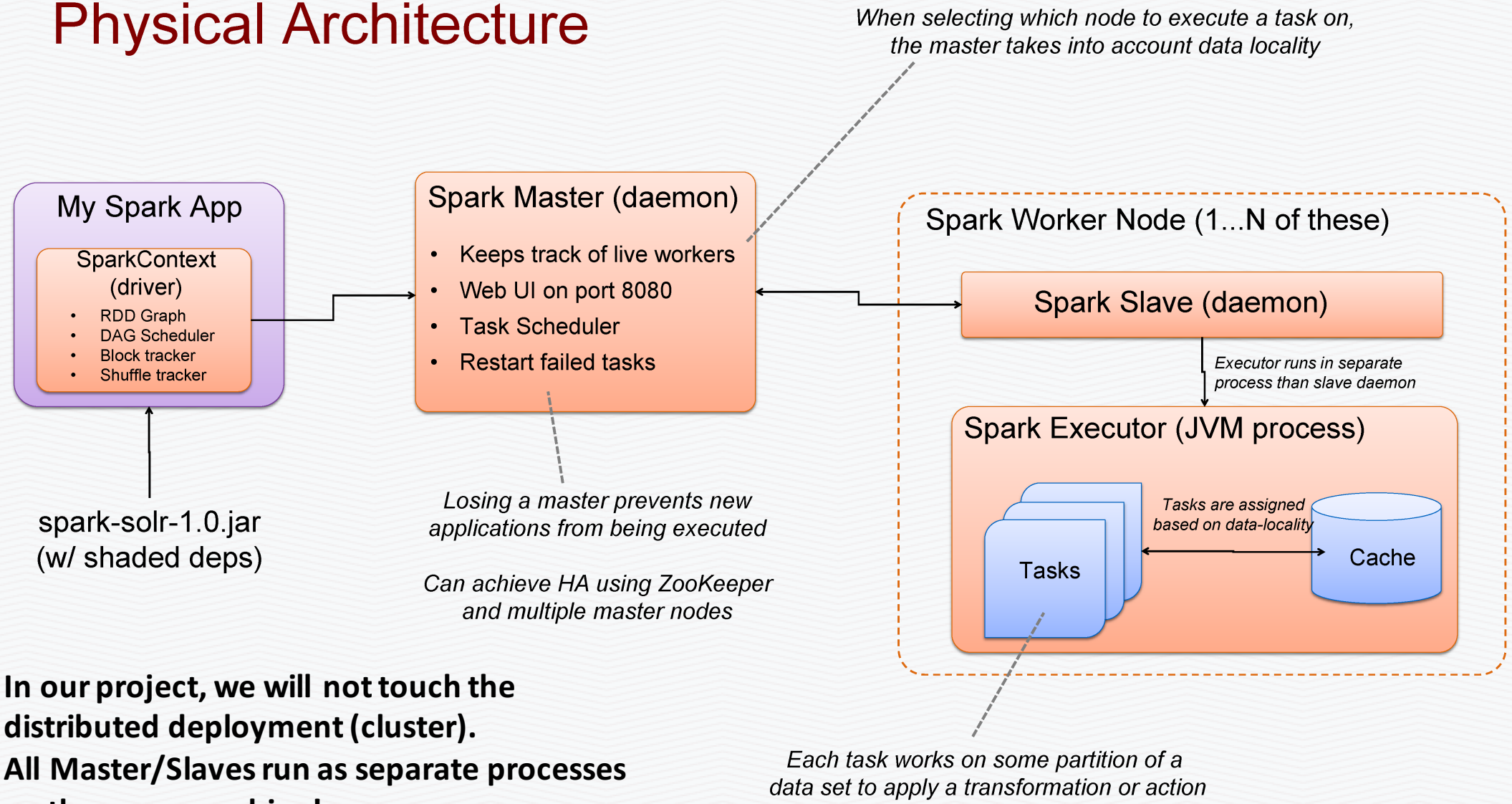
Slide taken from: https://www.slideshare.net/thelabdude/apachecon-na-2015-spark-solr-integration

# RDD Illustrated: Word count




Slide taken from: https://www.slideshare.net/thelabdude/apachecon-na-2015-spark-solr-integration

# Physical Architecture




**In our project, we will not touch the distributed deployment (cluster).**
**All Master/Slaves run as separate processes on the same machine!**

Slide taken from: https://www.slideshare.net/thelabdude/apachecon-na-2015-spark-solr-integration

# Spark SQL / DataFrame

- Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called **DataFrames** and can also act as distributed SQL query engine.

- A **DataFrame (RDD + Schema)** is a **distributed collection of data organized into named columns**. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood.

- DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs.

# DataFrame Operations

```scala
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// Create the DataFrame
val df = sqlContext.jsonFile("examples/src/main/resources/people.json")

// Show the content of the DataFrame
df.show()
// age name
// null Michael
// 30 Andy
// 19 Justin

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show()
// name
// Michael
// Andy
// Justin

// Select everybody, but increment the age by 1
df.select("name", df("age") + 1).show()
// name (age + 1)
// Michael null
// Andy 31
// Justin 20

// Select people older than 21
df.filter(df("name") > 21).show()
// age name
// 30 Andy
// Count people by age

df.groupBy("age").count().show()
// age count
// null 1
// 19 1
// 30 1
```

# Outline

- More tips about Project 2
- Apache Spark
- The Scala Language

# About Scala

- High-level language for the JVM
  - Object oriented + functional programming
- Statically typed
  - Comparable in speed to Java
  - Type inference save us from having to write explicit types most of time
- Interoperates with Java
  - Can use any Java class (inherit from etc.)
  - Can be called from Java code

# Example of Scala Programming

**Declaring variables:**

```
var x: Int = 7
var x = 7 // type inferred
val y = “hi” // read-only
```

**Java equivalent:**

```
int x = 7;

final String y = “hi”;
```

**Functions:**

```
def square(x: Int): Int = x*x
def square(x: Int): Int = {
  x*x
}
def announce(text: String) =
{
  println(text)
}
```

**Java equivalent:**

```
int square(int x) {
  return x*x;
}

void announce(String text) {
  System.out.println(text);
}
```

# More Features of Scala

- Supports lightweight syntax for anonymous functions, higher-order functions, nested functions, currying
- Effective pattern matching
- Flexible control structure
  - Allows defining new control structures without using macros, while maintaining static typing
- Automatic closure construction
  - When method is called, the actual def parameters are not evaluated and a no-argument function is passed

# Scala Typed

```
var str = "hello"
println(str)
//"hello"
str = "hi"
println(str)
//"hi"
str = 1.0
// ERROR !  // Statically typed Lanugage !
```

# Imperative vs functional Style

```
// imperative way
def total(list:List[Int]) = {
    var sum = 0
    for (i <- list)      // type is inferred !
        sum +=i
    sum // No return statement
}
```

```
//functional sense -- assignmentless programming
def totalFunctional(list:List[Int]) = {
    list.foldLeft(0) { (carryOver, e) => carryOver + e}
}

totalFunctional(List(1,2,3,4,5))
```

# More examples for Functional Senses

```
def totalMod(list:List[Int], selector:Int => Boolean) = {
    var sum = 0
    list.foreach {e => if (selector(e)) sum+=e}
    sum
}

totalMod(List(1,2,3,4,5), {e => e%2 == 0})
```

# Object Oriented Feature

```
System.out.println(1.toString()) // 1
System.out.println((new Integer(1)).toString)    //1
println(1.toString())    //1
```

```
class Car {
    def turn(direction:String) = {
        println("Turning " + direction)
    }
}

val car = new Car()
car.turn("left")

//Turning left
```

# Class in Scala

```
// Classes

public class Car {
    final private int year;
    private int miles;

    public int getYear() {

    }

    public int getMiles() { }

    ....
}

//Scala
class Car(val year:Int, var miles:Int) {
    def this(year) = {  //auxillary constructor
        this(year, 0)   //primary constructor -- forced !
    }
}

val ford = new Car(2010, 0)
println(ford.year)
println(ford.miles)
```

# Traits in Scala

```
class A {
    foo()
}

class B extends A{
    foo() { ... }
}

class C extends A{
    foo() { ... }
}

class D extends B, C {
    foo()
}
```

Inheritance relationship

```
// Scala
trait Drawable {
    def draw() {}
    def draw2() { ... }
}

trait Cowboy extends Drawable {
    override def draw() { println("In Cowboy!!")}
}

trait Artist extends Drawable {
    override def draw() { println("In Artist")}
}

class CowboyArtist extends Cowboy with Artist

object Main {
    (new CowboyArtist()).draw()
    // In Artist
}
```

# Trait as types

```
trait X {def foo(s:String)}
trait Y {def bar(i:Int)}}
class A {
    def baz() {println("Hello World!")}
}

class B {
    def barfoobaz(axy: A extends X with Y) = {
        axy.foo("Hi")
        axy.bar(2)
        axy.baz()
    }
}
```

# More about Scala and Spark

- Online Resources
  - http://twitter.github.io/scala_school/index.html
  - https://www.youtube.com/watch?v=pDq06gwJnLk&list=PLS1QulWo1RIYEMepIBinxplXsx9v0zCSf&index=16&t=0s
  - https://jaceklaskowski.gitbooks.io/mastering-spark-sql/