# SPARK&SCALA

## NEW MATERIAL

# NARESH

# TECHNOLOGIES


## MANOJ XEROX

Gayarti nager,Behind-Huda-Ameerpet

SOFT WARE INSTITUTES MATERIAL AVAILABLE

CELL :9542556141

# What is Spark?

Spark

## Fastest Big Data Analytics Framework


Spark

hadoop

cassandra

mongoDB

## Spark?

1. General purpose framework

2. Fast data processing

3. Flexible and easy to use

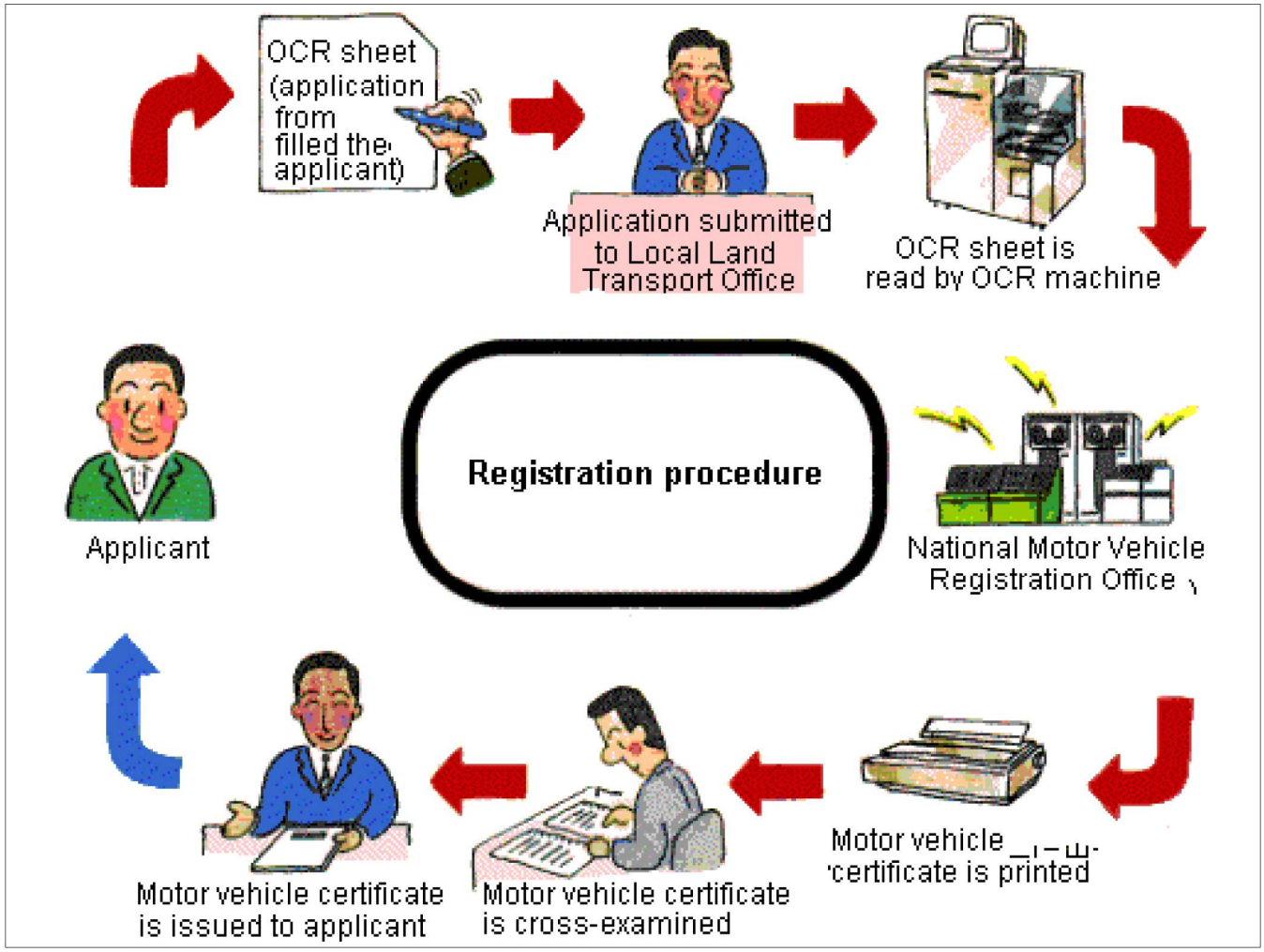
## 1. General Purpose Framework

- Allows different methods of data processing
  - Batch

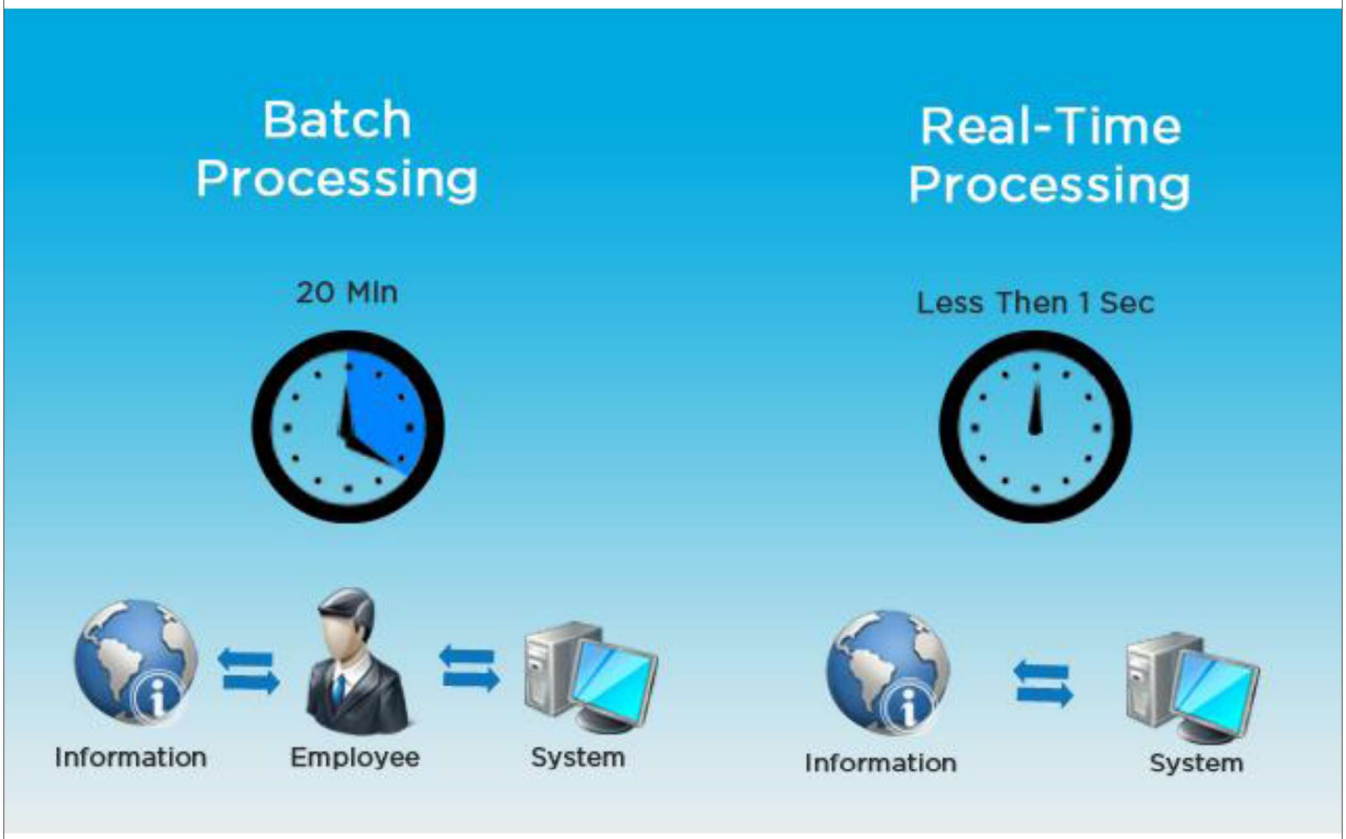
Batch VS NRT

Zillions of data once

Gigabyte data per second




## Real time Processing




## 1. General Purpose Framework

- Allows different methods of data processing
  - Batch
  - Interactive Ad-Hoc
  - Streaming

## 1. General Purpose Framework

- Allows different methods of data processing
  - Batch
  - Interactive Ad-Hoc
  - Streaming
- Wide range of workloads covered under one system
  - Machine learning


## 1. General Purpose Framework

- Allows different methods of data processing
  - Batch
  - Interactive Ad-Hoc
  - Streaming
- Wide range of workloads covered under one system
  - Machine learning

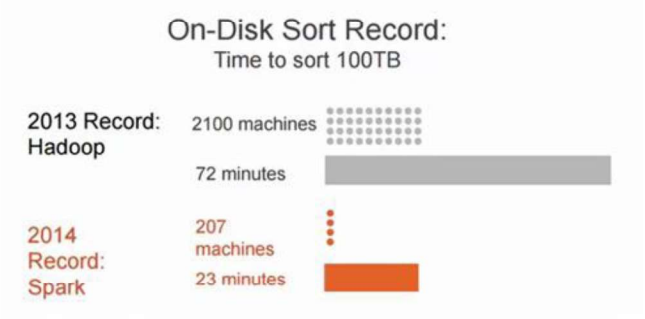
  - Text processing

## 1. General Purpose Framework

- Allows different methods of data processing
  - Batch
  - Interactive Ad-Hoc
  - Streaming
- Wide range of workloads covered under one system
  - Machine learning
  - Text processing
  - Batch data processing
- Hadoop requires different tools for the various workloads
  - MapReduce can only run batch jobs

## 2. Fast Data Processing

- Proven to be 10-100 times faster than Hadoop for specific use cases
- Speed achieved by leveraging in-memory computations across a cluster




Source: Presentation by Matei Zaharia in 2015

## 3. Flexible and Easy to Use

- Can be deployed stand-alone or integrated with Hadoop
- Offers a growing set of APIs
- Scala development is straightforward and efficient

## Easy: High productivity language support

### Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

### Scala

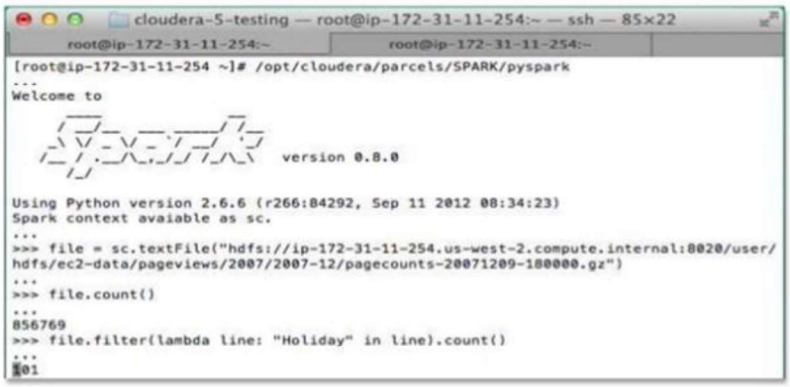
```
val lines = sc.textFile(...)
lines.filter(s => s.contains("ERROR")).count()
```

### Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>()
{
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

- Native support for multiple languages with identical APIs
- Use of closures, iterations and other common language constructs to minimize code

# Easy: Use Interactively




- Interactive exploration of data for data scientists – no need to develop "applications"
- Developers can prototype application on live system as they build application

# Easy: Expressive API

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip
- sample
- take
- first
- partitionBy
- mapWith
- pipe
- save

# Easy: Example – Word Count (M/R)

```
public static class WordCountMapClass
extends MapReduceBase implements
Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one =
  new IntWritable(1);
  private Text word = new Text();

  public void map(
    LongWritable key, Text value,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer itr
      = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

```
public static class WorkdCountReduce
extends MapReduceBase implements
Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(
    Text key, Iterator<IntWritable> values,
    OutputCollector<Text, IntWritable> output,
    Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

# Easy: Example – Word Count (Spark)

```
val spark = new SparkContext(master, appName, [sparkHome], [jars])
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

## Easy: Out of the Box Functionality

- **Hadoop Integration**
  - Works with Hadoop Data
  - Runs With YARN
- **Libraries**
  - MLlib
  - Spark Streaming
  - GraphX (alpha)
- **Language support:**
  - Improved Python support
  - SparkR
  - Java 8
  - Schema support in Spark's APIs

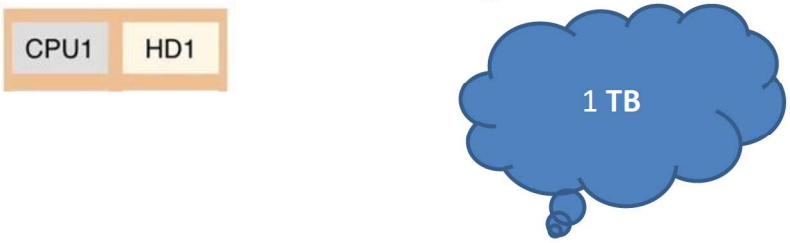
# The Problem: Big Data?

**Big data** is a term applied to data sets whose size is beyond the ability of commonly used software tools to capture, manage, and process the data within a tolerable elapsed time.




## Traditional Approach

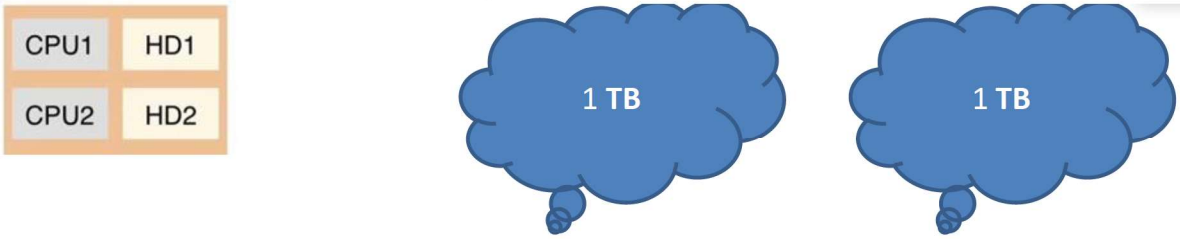
### Scale Up




Up

## Traditional Approach

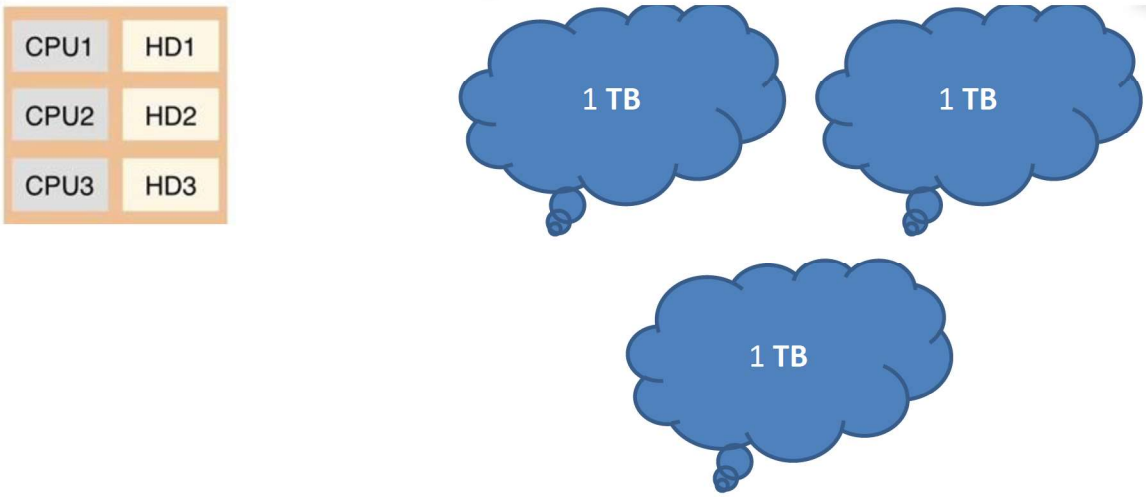
### Scale Up




Up

## Traditional Approach

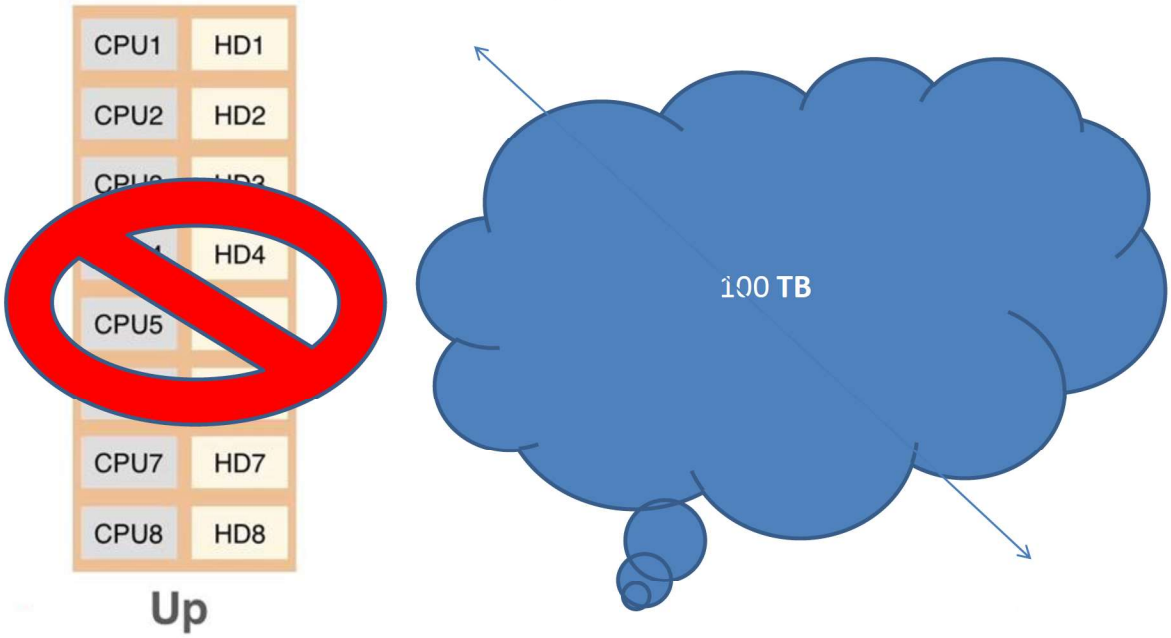
### Scale Up




Up

## Limit

### Scale Up

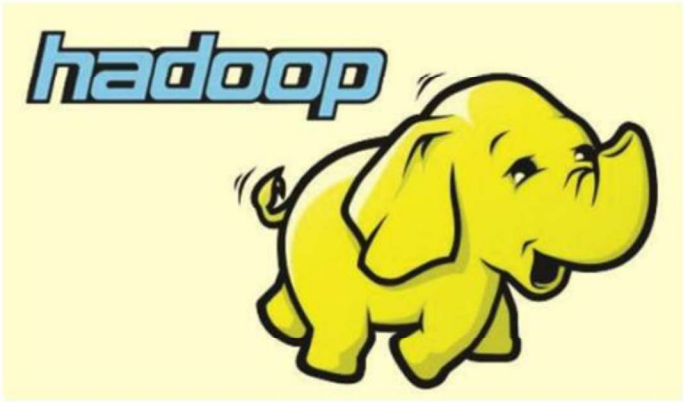



Up

## What is Hadoop?

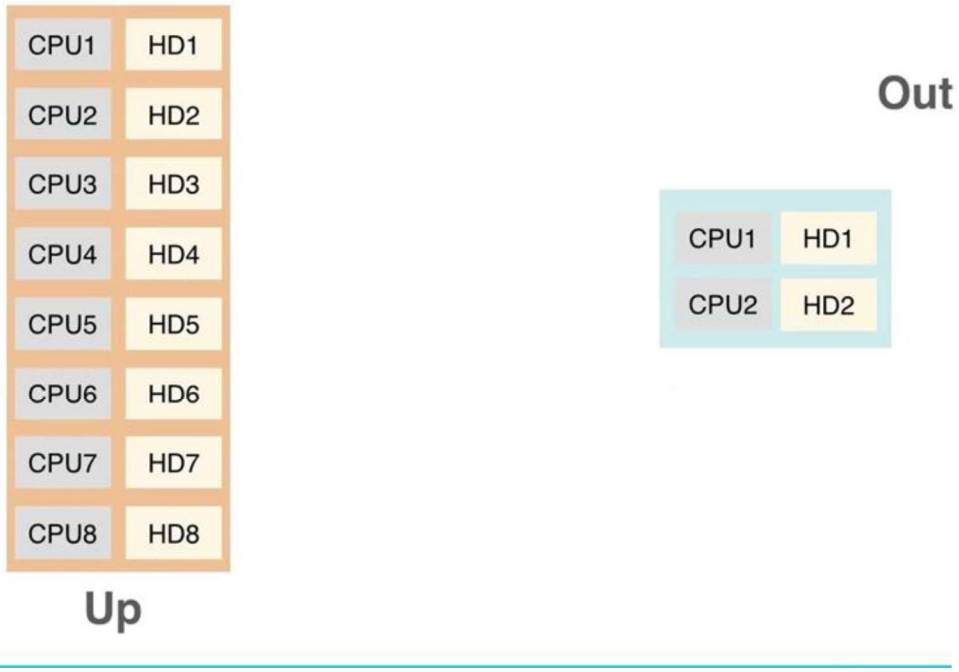
→ Apache Hadoop is a framework that allows for the distributed processing of large data sets across clusters of commodity computers using a simple programming model.

→ It is an Open-source Data Management with scale-out storage and distributed processing.



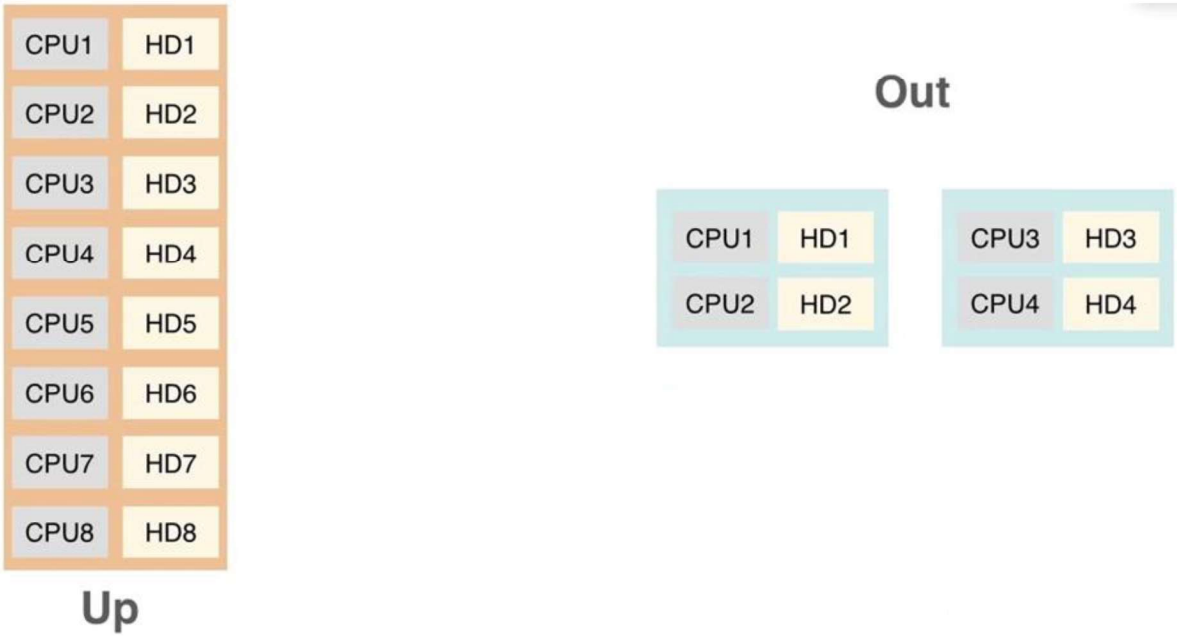
## New Approach

### Scale Up versus Out



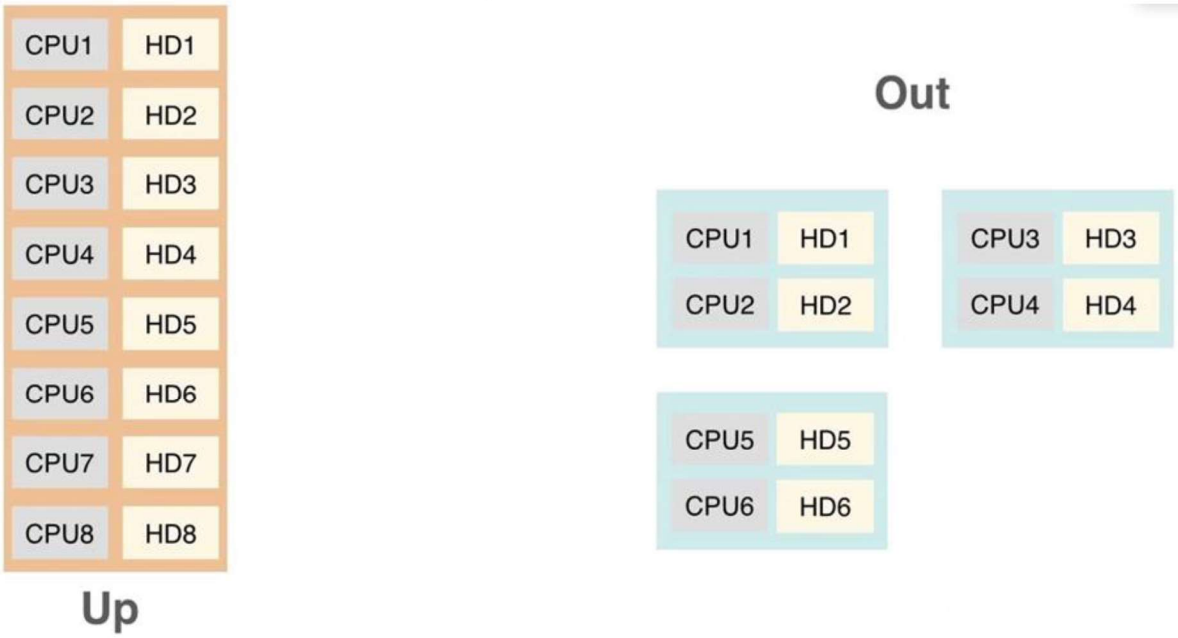

## New Approach

### Scale Up versus Out



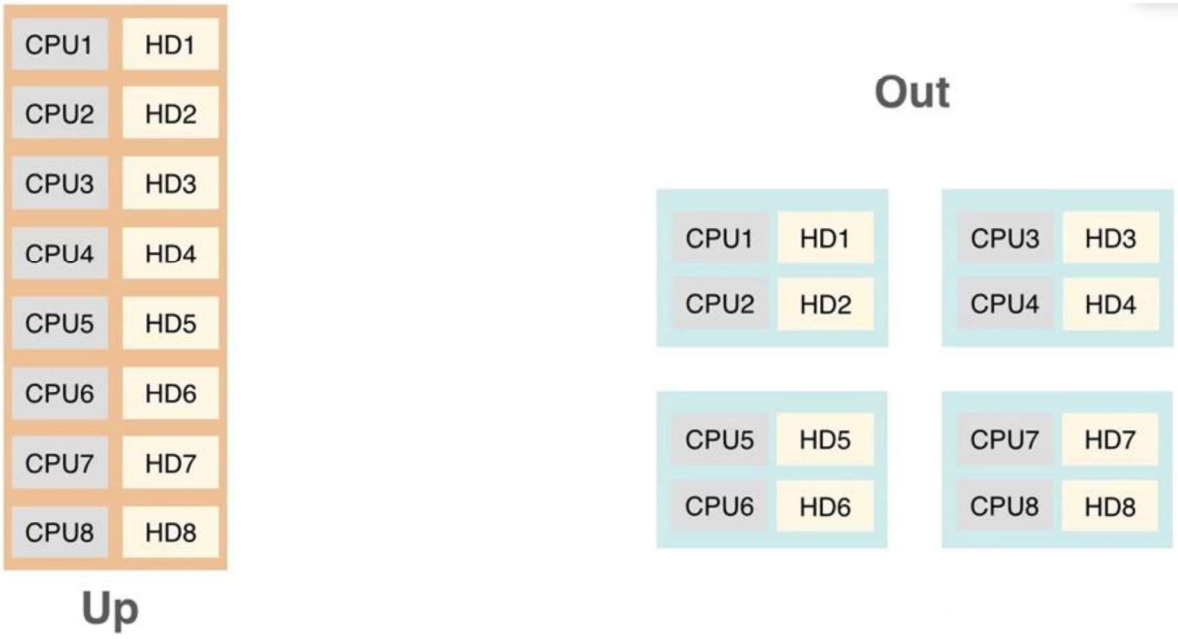
## New Approach

### Scale Up versus Out

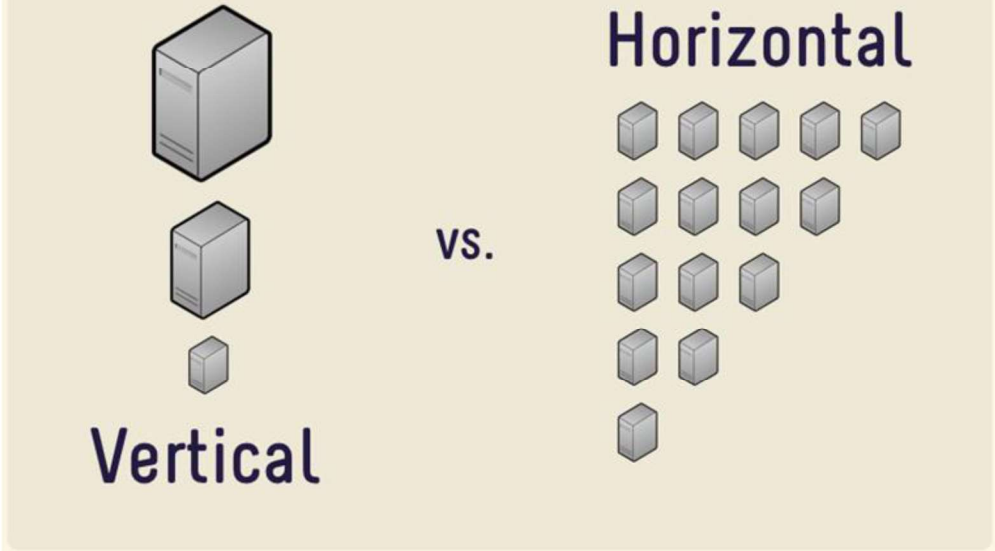



## No Limit

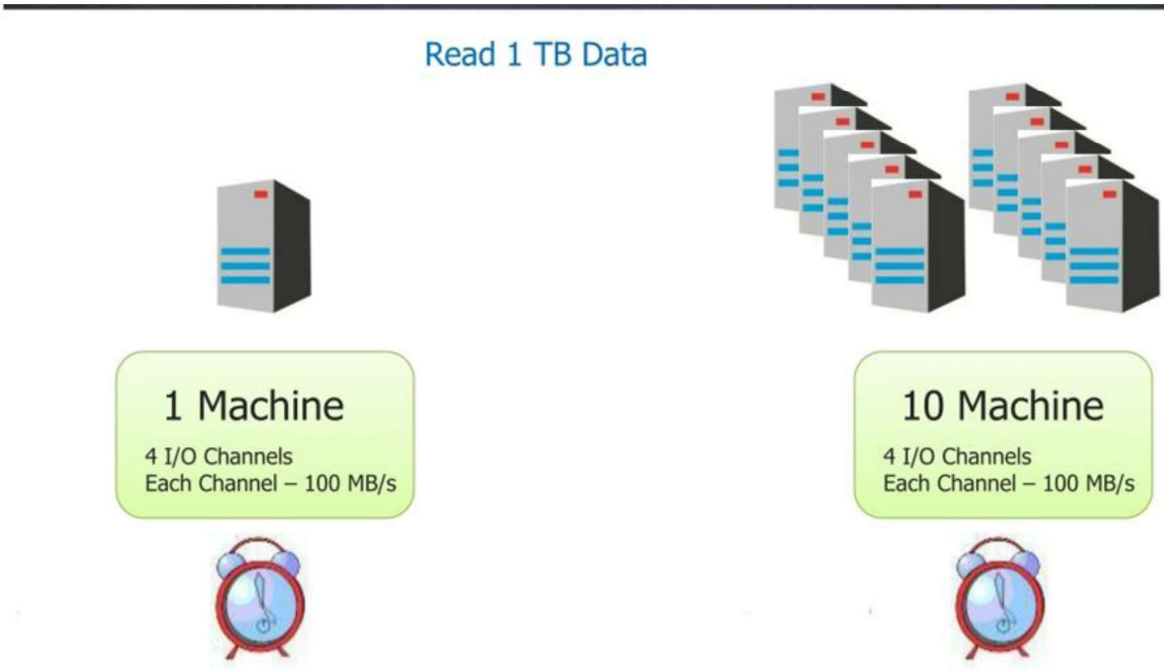
### Scale Up versus Out



## Traditional Approach vs New

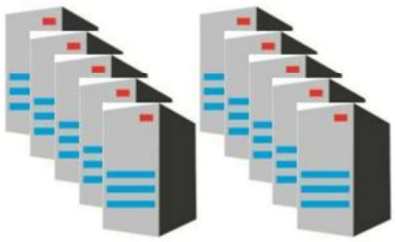



## What is Hadoop?



## Why DFS?

Read 1 TB Data



| 1 Machine | 10 Machine |
| --- | --- |
| 4 I/O Channels<br>Each Channel – 100 MB/s | 4 I/O Channels<br>Each Channel – 100 MB/s |
| 43 Minutes | 4.3 Minutes |

## Replication

**Block A :** 

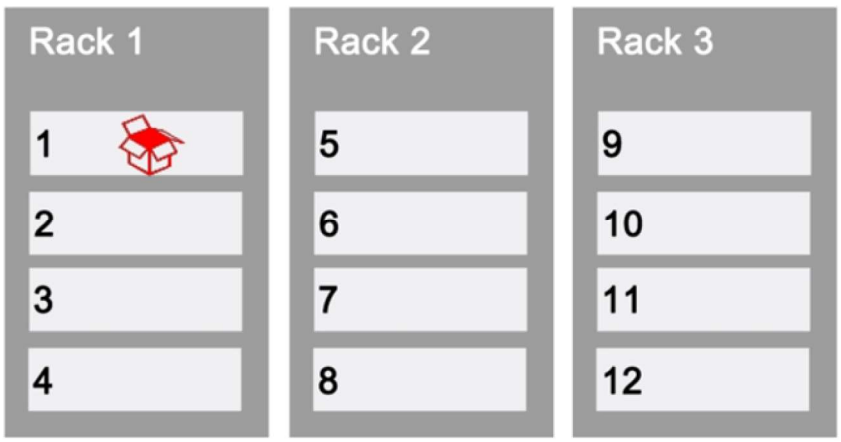
**Block B :** 

**Block C :** 

| Rack 1 | Rack 2 | Rack 3 |
| --- | --- | --- |
| 1 | 5 | 9 |
| 2 | 6 | 10 |
| 3 | 7 | 11 |
| 4 | 8 | 12 |

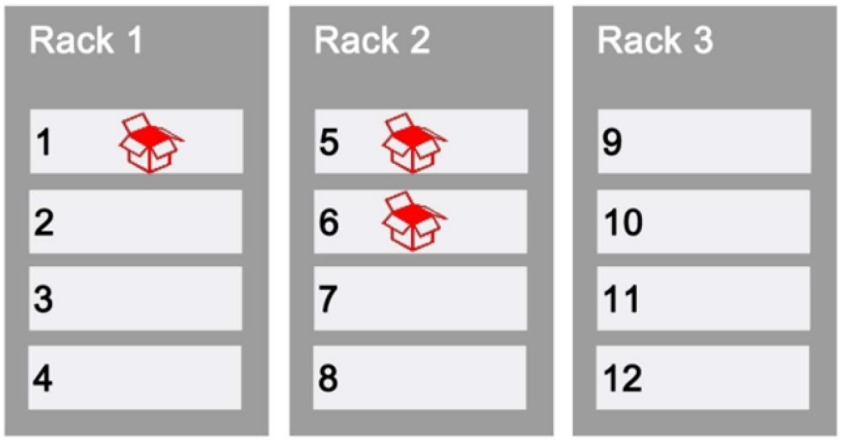
## Replication

Block A :
Block B :
Block C :



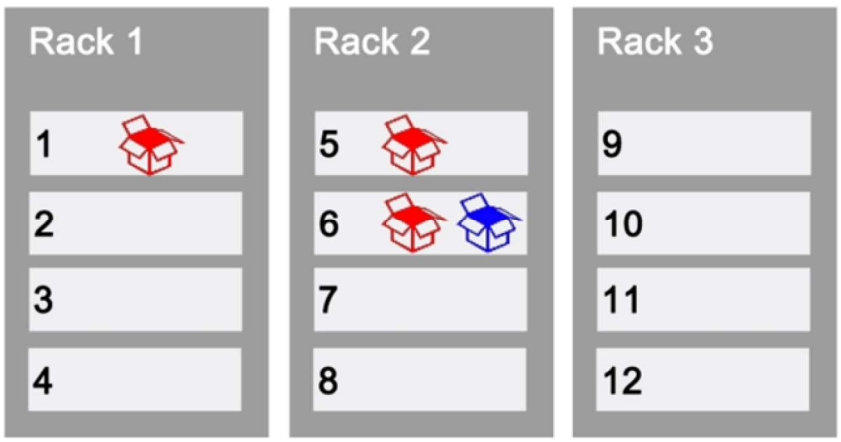

## Replication

Block A :
Block B :
Block C :



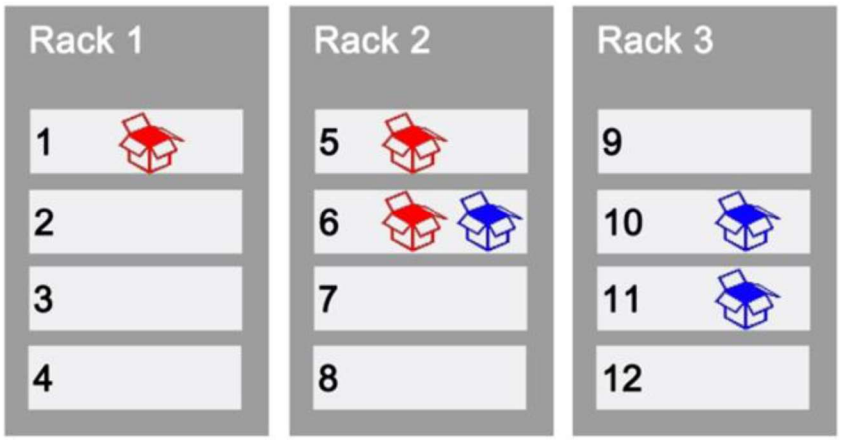
## Replication

Block A :
Block B :
Block C :



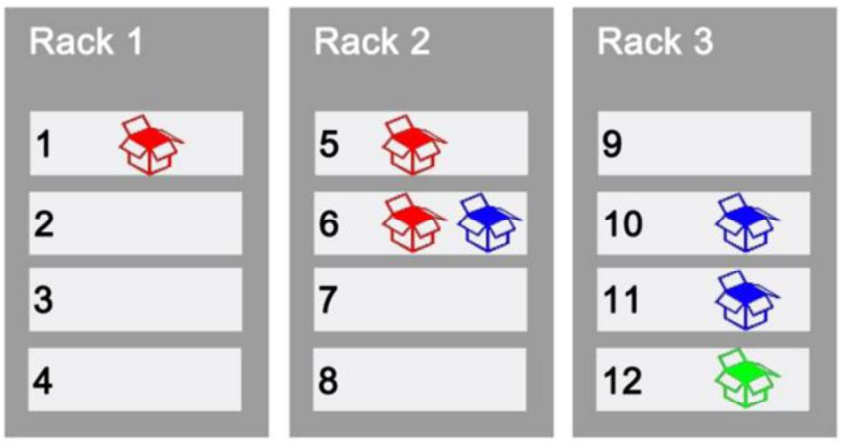

## Replication

Block A :
Block B :
Block C :



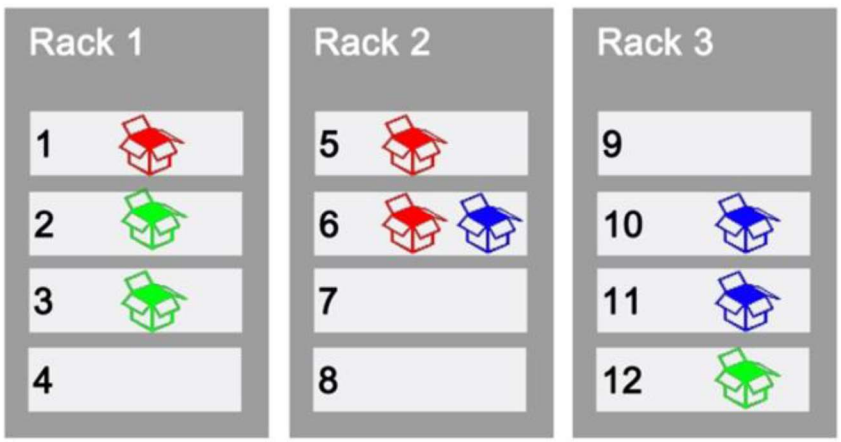
## Replication

Block A :
Block B :
Block C :




## Replication

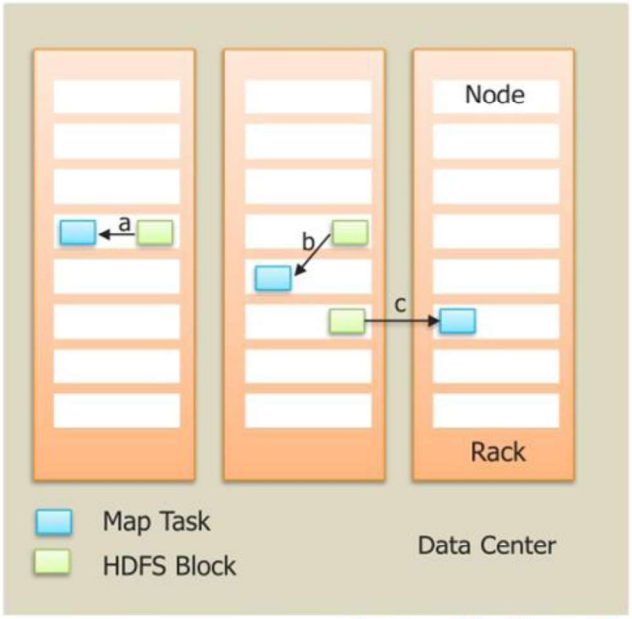
Block A :
Block B :
Block C :



## Data Locality

» Taking processing to the data

» Processing data in parallel

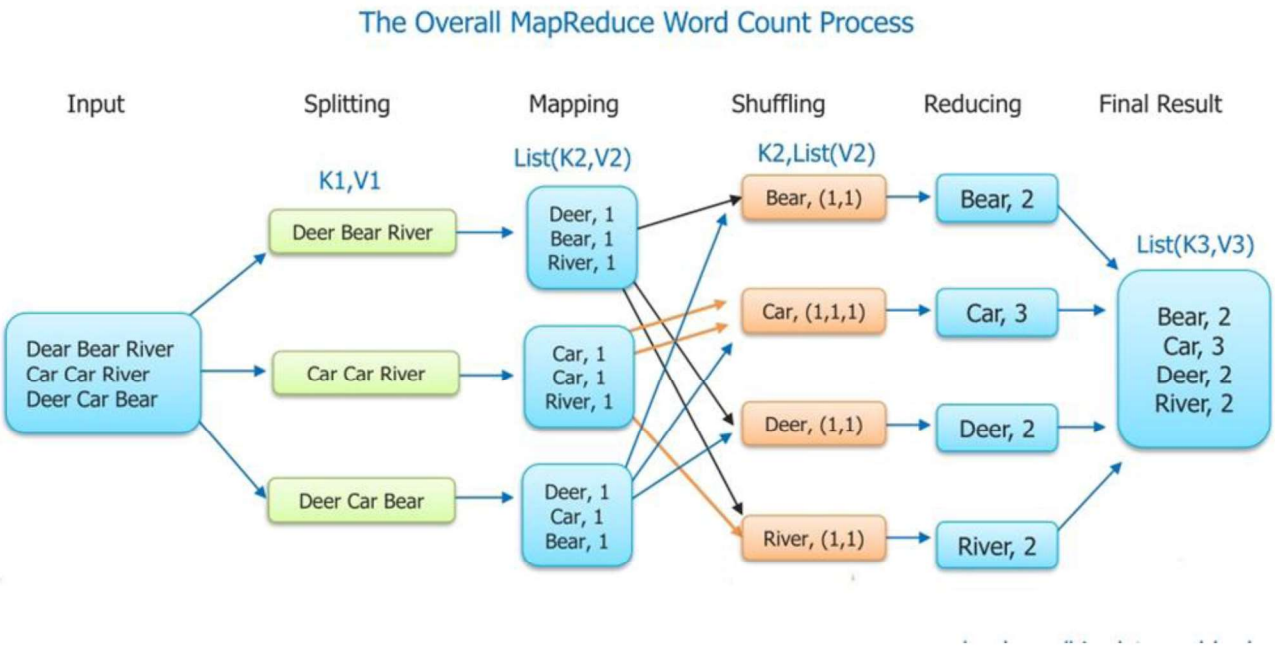



# Before Spark: MapReduce

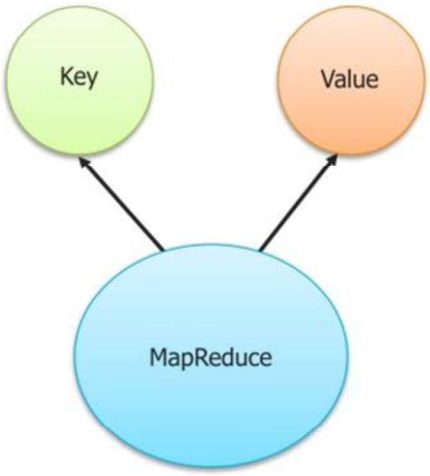
•Hadoop Map-Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

- Programming model used by Google
- A combination of the **Map** and **Reduce** models with an associated implementation
- MapReduce is highly scalable and can be used across many computers and answers all problems mentioned earlier.
- Many small machines can be used to process jobs that normally could not be processed by a large machine.
- Programming Languages Supported: Java
  - Hadoop Streaming: Python, Ruby, Perl etc
  - Hadoop Pipes: C++

# MapReduce Paradigm

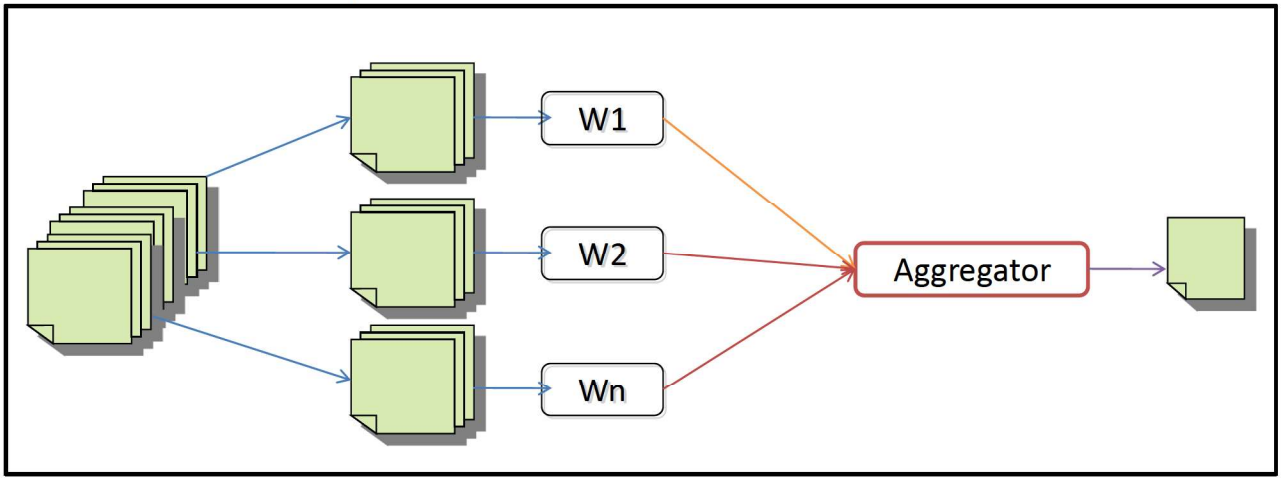



# Anatomy of a MapReduce Program



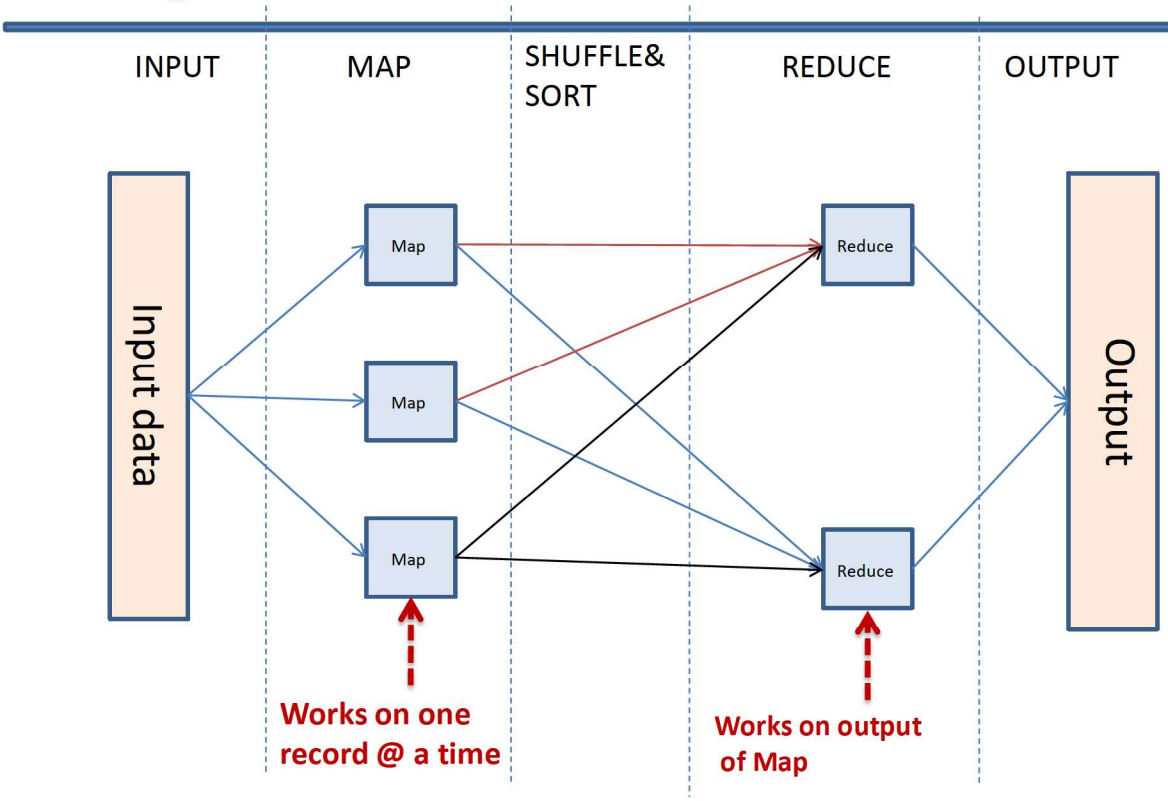

# Word Count – Distributed Solution

- Problem:
  - Large quantity of documents
  - Count the number of times each distinct word occurs in the documents
- Solution:
  - Divide and Conquer






# Map Reduce Flow



# Log processing

10.77.25.72 [16/Feb/2009:14:29:27] http://www.google.com/
10.77.25.72 [16/Feb/2009:14:29:27] http://www.rediff.com/
…
10.88.45.101 [16/Feb/2009:16:58:13] http://www.cnn.com/
10.88.45.101 [16/Feb/2009:16:58:13] http://www.yahoo.com/
10.77.45.82 [17/Feb/2009:09:35:42] http://news.cnet.com/
10.88.45.101 [18/Feb/2009:16:03:07] http://www.cnn.com/
….
10.88.45.101 [18/Feb/2009:16:03:07] http://www.yahoo.com/
10.88.88.69 [09/Jun/2009:15:43:50] http://www.cnn.com/
10.88.88.69 [11/Jun/2009:16:11:25] http://news.cnet.com/
10.88.88.69 [11/Jun/2009:16:11:25] http://www.cnn.com/
….
10.77.224.66 [12/Jun/2009:14:41:54] http://www.rediff.com/
10.77.6.54 [12/Jun/2009:15:05:31] http://www.yahoo.com/
10.88.88.69 [19/Jun/2009:16:31:11] http://www.cnn.com/
10.77.222.22 [26/Jun/2009:15:07:46] http://www.yahoo.com/
10.77.222.22 [26/Jun/2009:15:07:50] http://www.cnn.com/
10.77.222.22 [26/Jun/2009:15:11:56] http://www.yahoo.com/
10.88.88.69 [26/Jun/2009:15:48:15] http://www.cnn.com/
10.88.88.69 [02/Jul/2009:16:11:51] http://www.yahoo.com/
…..
….

Log file may be in Terabytes

Find Url Frequencies



# Log processing

**Key:** Offset into the file
**Value:** 10.88.45.101 [16/Feb/2009:16:58:13] http://www.cnn.com/

Input to Mapper

**Key:** http://www.cnn.com/
**Value:** 1

Output of Mapper

**Key:** http://www.cnn.com/
**Value:** [1,1,1,1,1/……] or [n1, n2, n3 … ] if combiner is used

Input to Reducer

**Key1:** http://www.cnn.com/
**Value1:** 1000

**Key2:** http://www.yahoo.com/
**Value2:** 5000

Output from Reducer

# Map Reduce

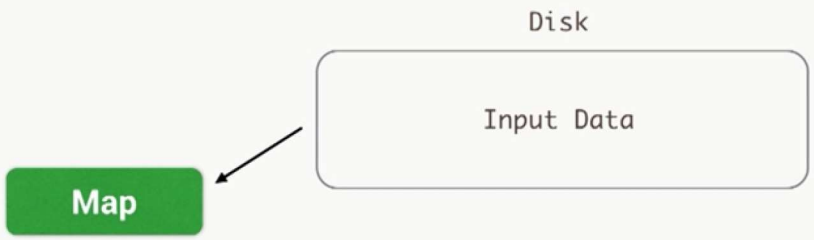
- Difficult—MapReduce is difficult to program and needs abstractions.
- Interactive Mode—There is no built in interactive mode except for pig & hive
- Streaming—Used for generating batch reports that help in finding insights to historical queires.
- Performance— Does not leverage the memory of the cluster.
- Latency—It is disk oriented completely.
- Ease of Coding - Writing data pipelines is complex & lengthy

Problem

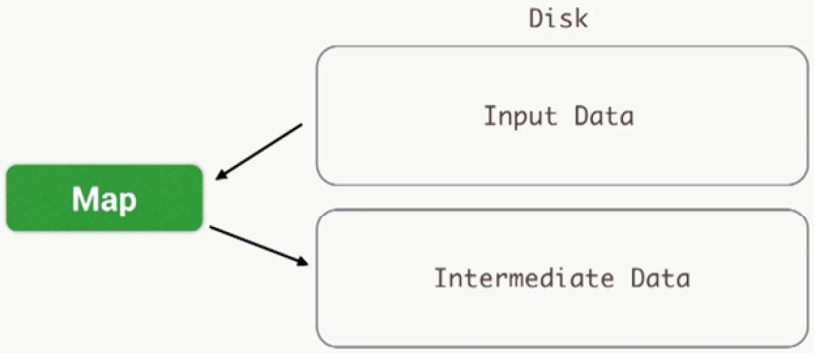


# Map Reduce

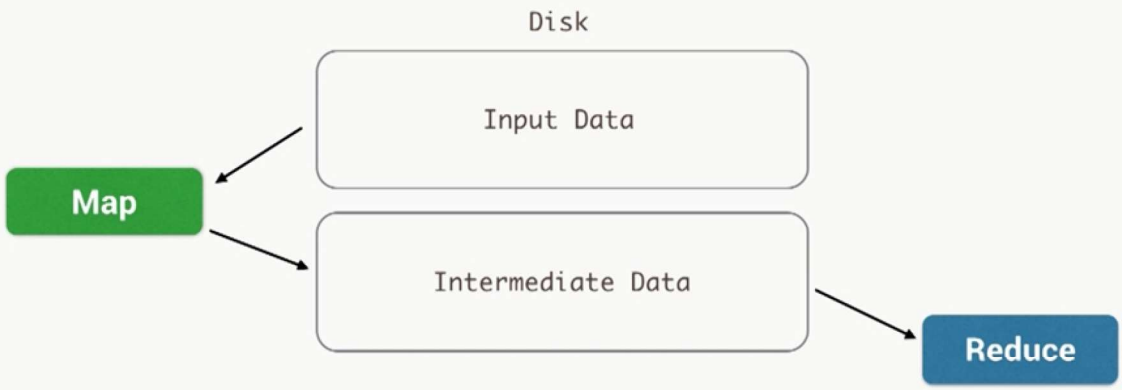
# Map Reduce



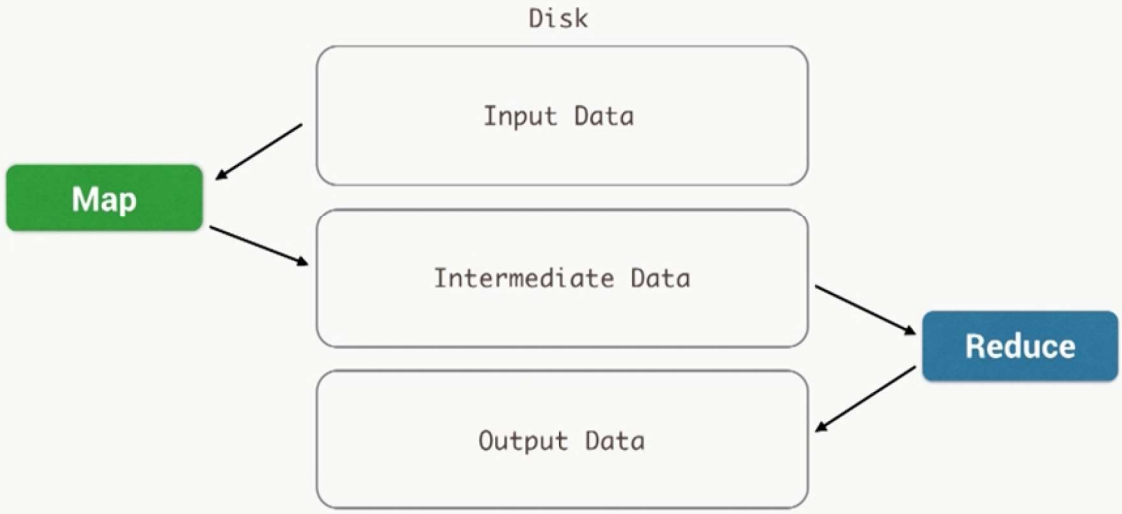

# Map Reduce



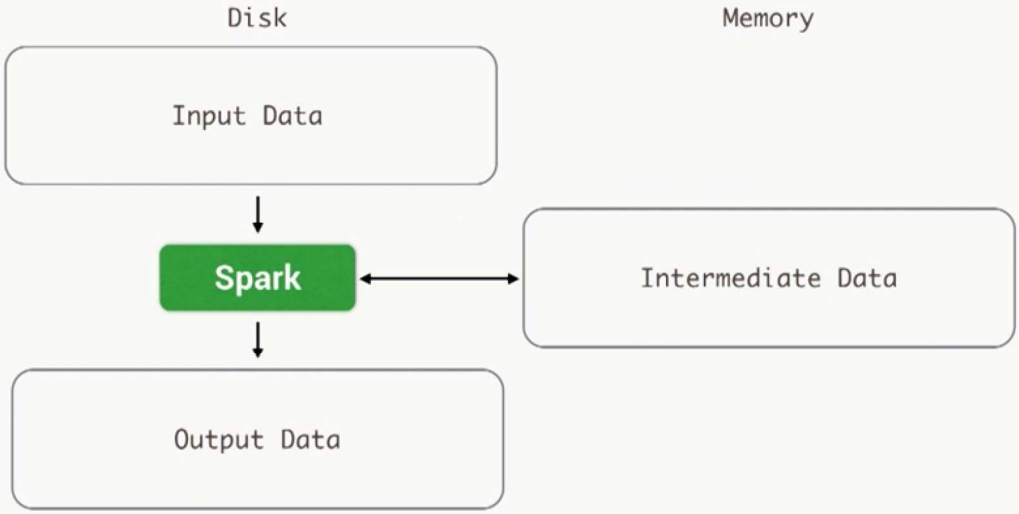
# Map Reduce




# Map Reduce



# In memory




# Map Reduce Problem

- Difficult—MapReduce is difficult to program and needs abstractions.
- Interactive Mode—There is no built in interactive mode except for pig & hive
- Streaming—Used for generating batch reports that help in finding insights to historical queires.
- Performance— Does not leverage the memory of the cluster.
- Latency—It is disk oriented completely.
- Ease of Coding - Writing data pipelines is complex & lengthy

Problem

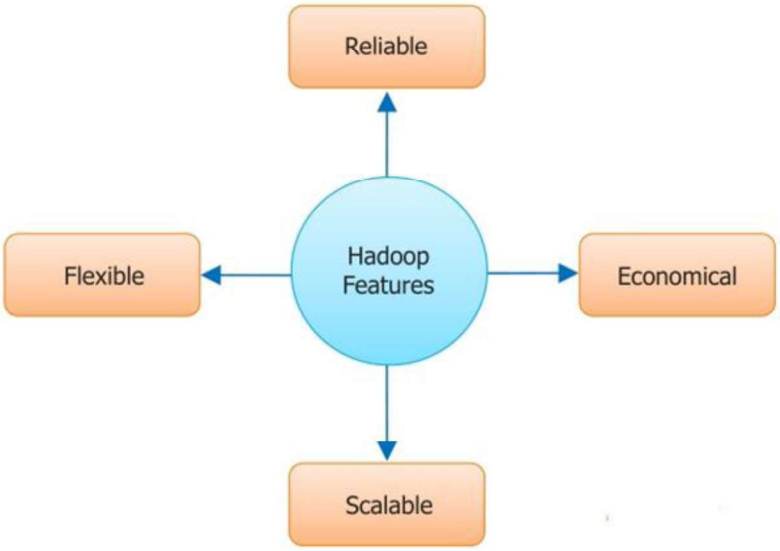
# Spark's Solution

■ Difficult—Spark is easy to program and compared to MR.
■ Interactive Mode— Spark is Interactive
■ Streaming— Streaming is accomplished in a simple way and also provides batch processing and machine learning.
■ Performance— 100 times faster than Hadoop.
■ Latency—In memory computing ensures low latency and provides caching abilities across distributed workers.
■ Ease of Coding - Simpler & less compact compared to MR

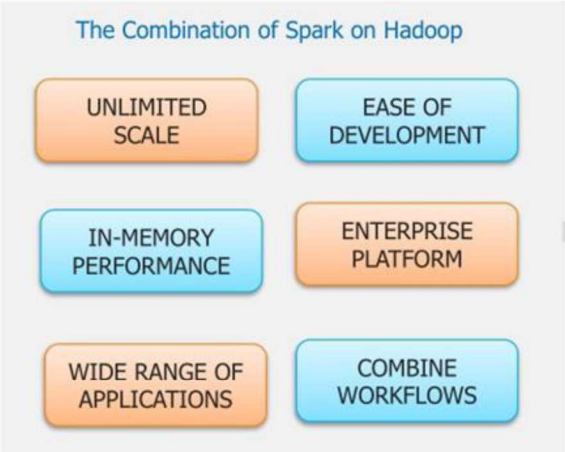
Solution



## Hadoop Key Characteristics

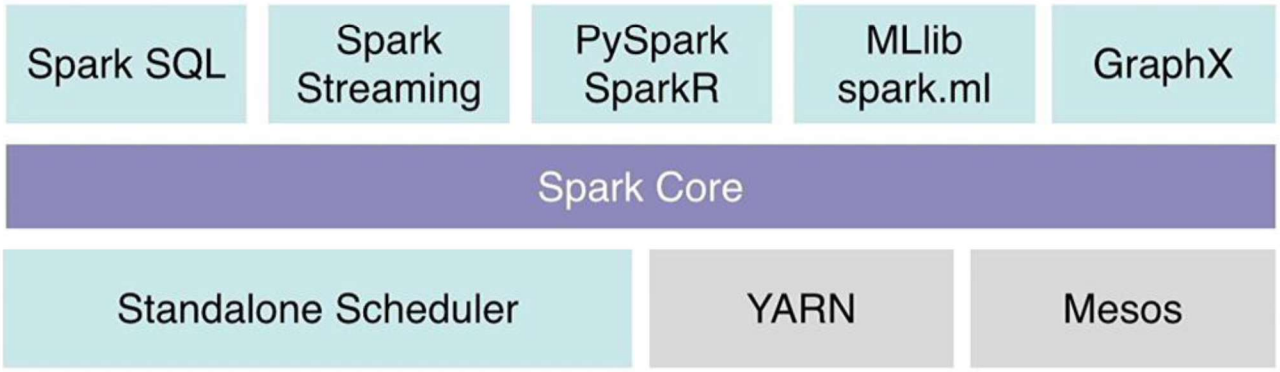


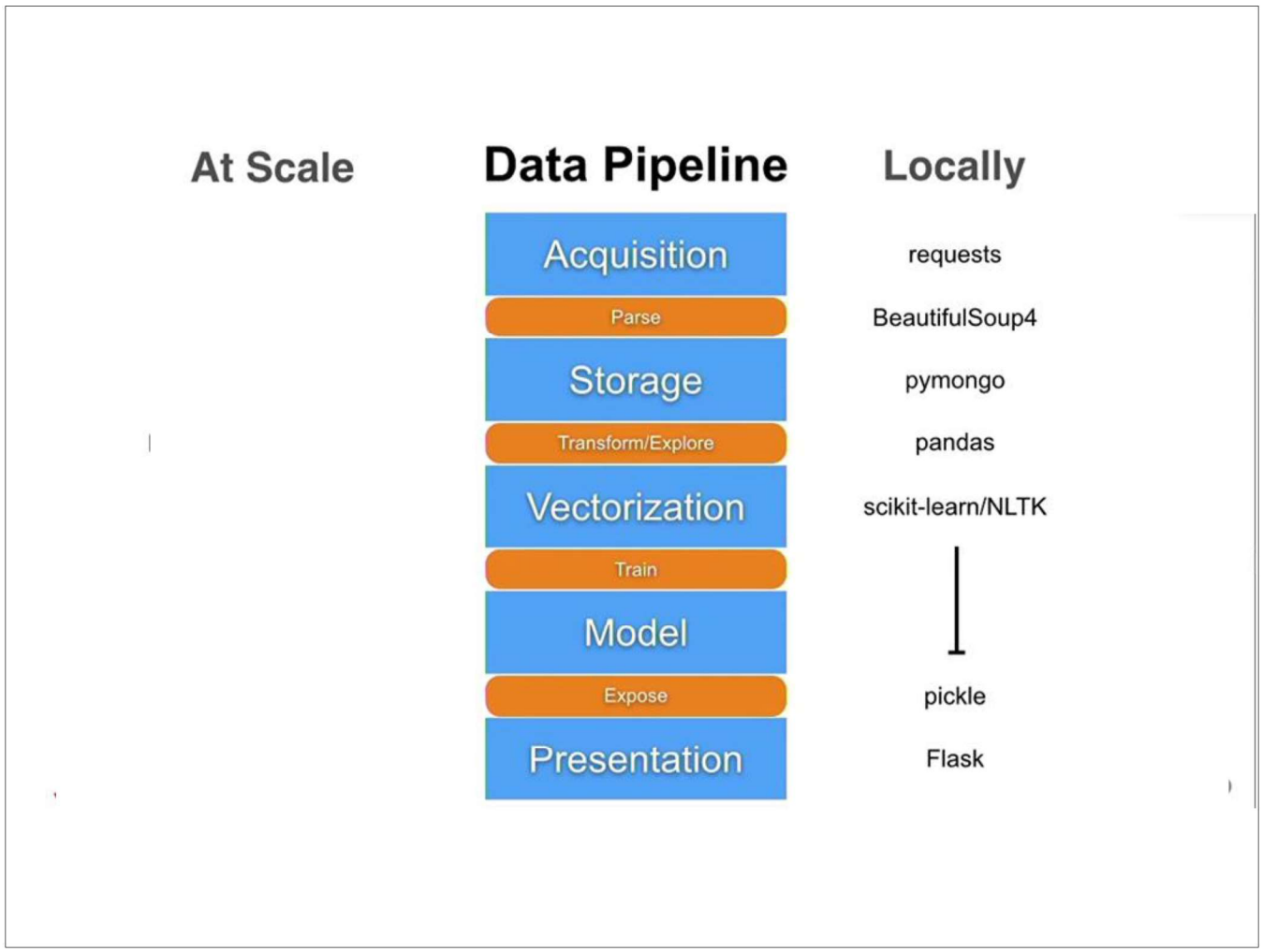
## Spark + Hadoop

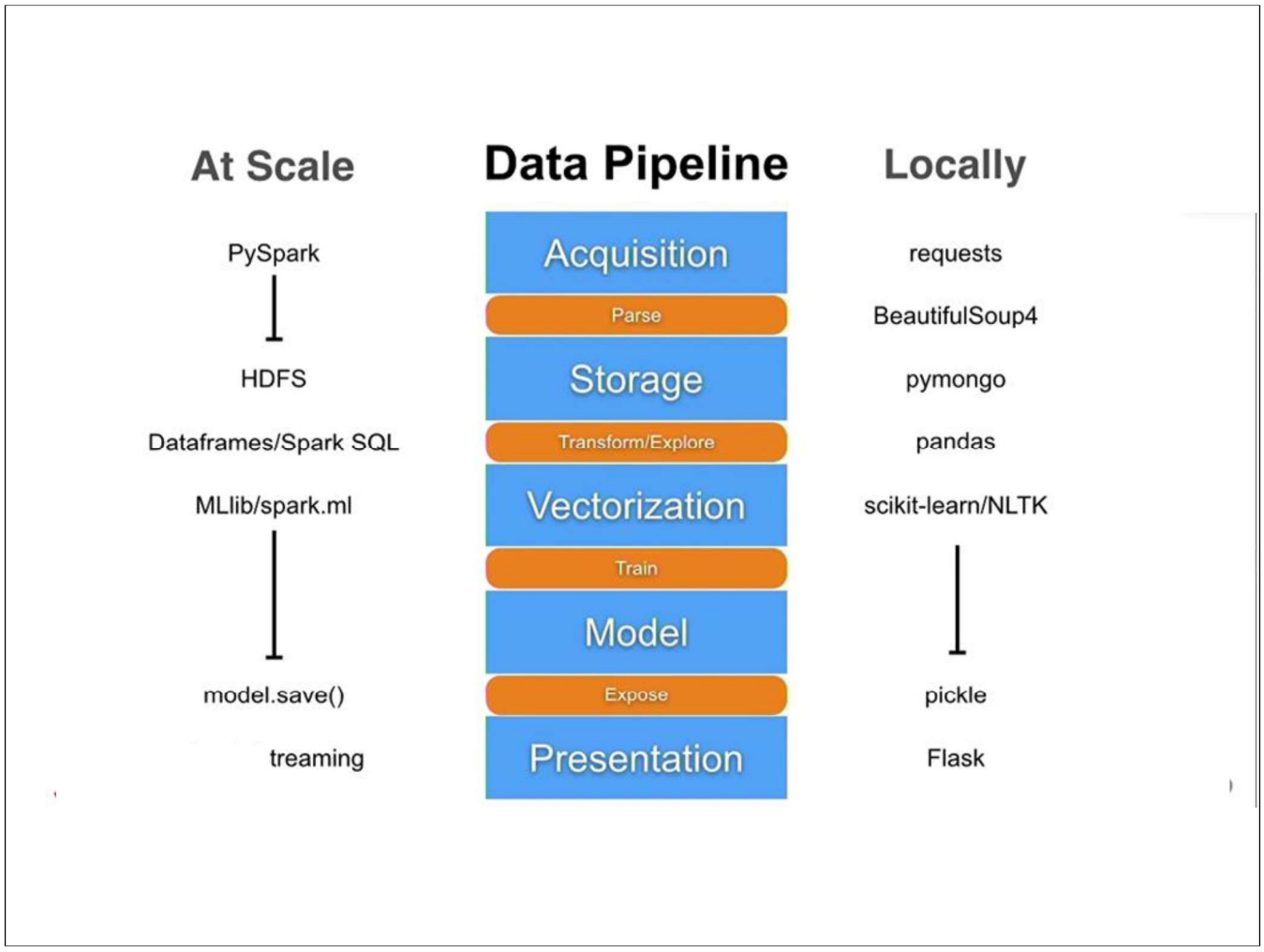


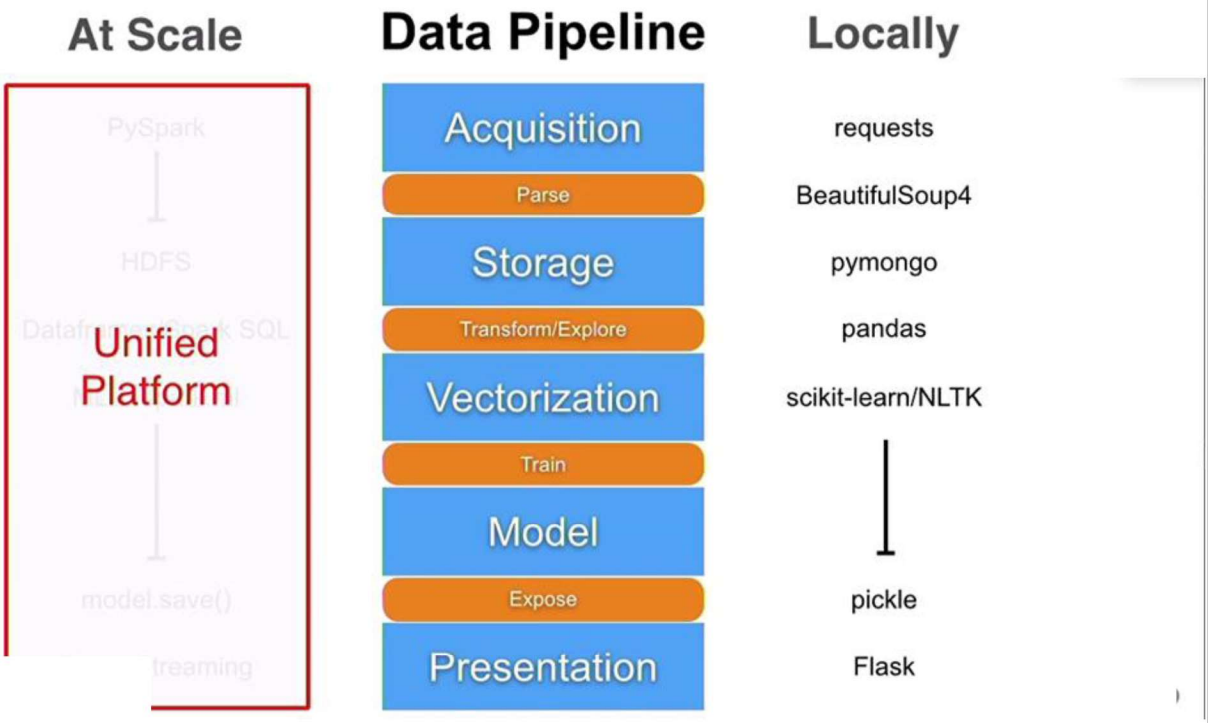

# Spark Components

**Unified Platform**




At Scale
Data Pipeline
Locally
Acquisition
Parse
Storage
Transform/Explore
Vectorization
Train
Model
Expose
Presentation
requests
BeautifulSoup4
pymongo
pandas
scikit-learn/NLTK
pickle
Flask


At Scale
Data Pipeline
Locally
PySpark
HDFS
Dataframes/Spark SQL
MLlib/spark.ml
model.save()
treaming
Acquisition
Parse
Storage
Transform/Explore
Vectorization
Train
Model
Expose
Presentation
requests
BeautifulSoup4
pymongo
pandas
scikit-learn/NLTK
pickle
Flask




## Spark Framework – Unified Stack

- Advantages of "Unified Stack" approach
  - Enables various workloads to be combined
  - Additional components can be integrated easily
  - Lower level stack enhancements impact higher level stack components
  - Cost efficient
  - Allows leveraging the different workloads provided by Spark

Resilient Distributed Dataset

# RDD

**Are**

- Immutable
- Partitioned
- Reusable

# RDD

**Are**

- Immutable
- Partitioned
- Reusable

**and Have**

**Tranformations**

- Produces new RDD
- Calls: `filter, flatmap, map, distinct, groupBy, union, zip, reduceByKey, subtract`

**Actions**

- Start cluster computing operations
- Calls: `collect: Array[T], count, fold, reduce..`

# API

map reduce

# API

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin

reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip

sample
take
first
partitionBy
mapWith
pipe
save
...

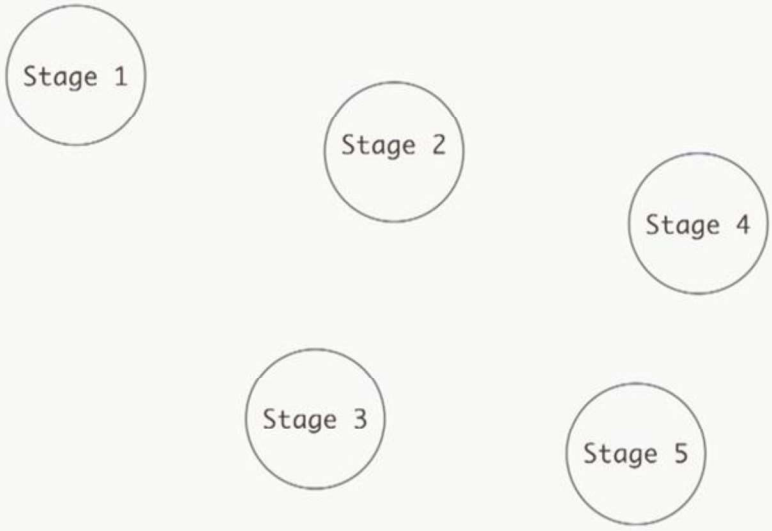
Directed Acyclic Graph

Resilient Distributed Dataset

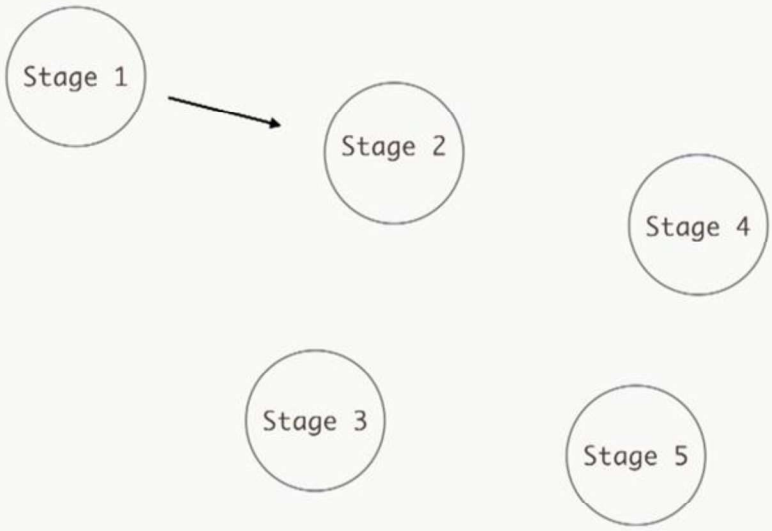
DAG

RDD

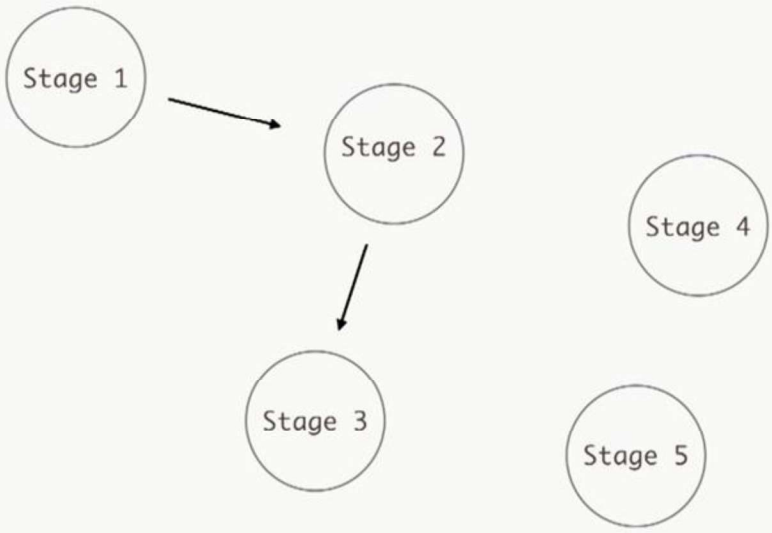
# DAG


Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

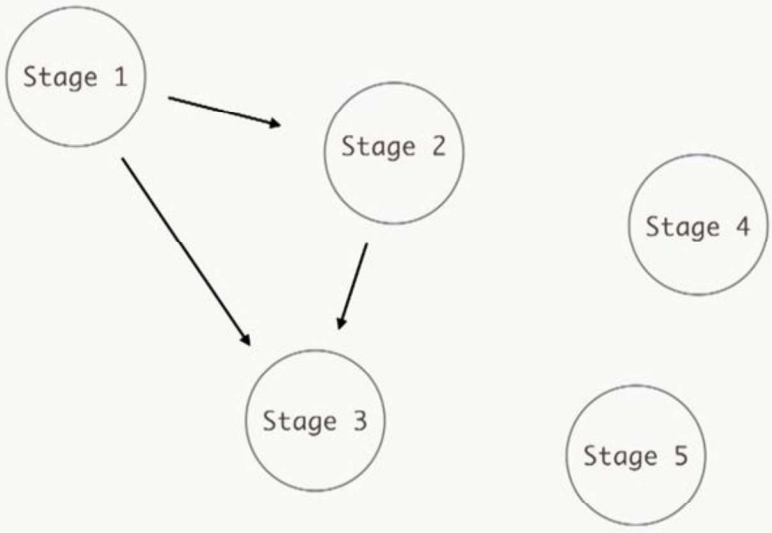
# DAG


Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

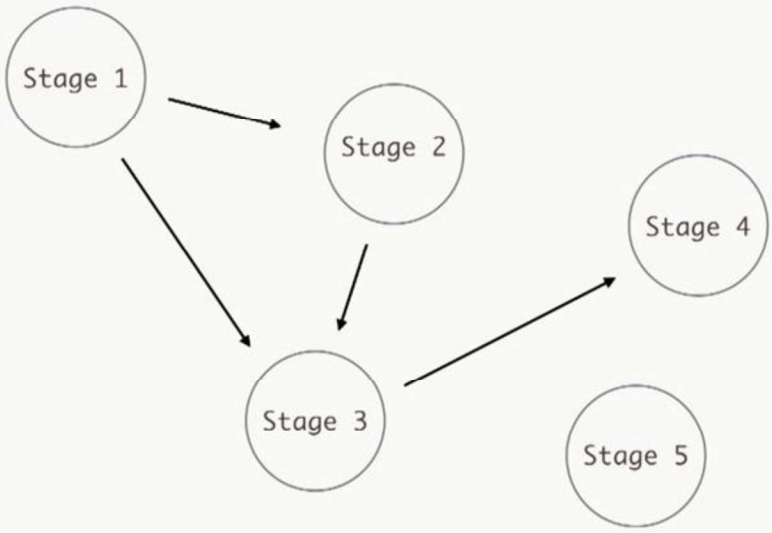
# DAG


Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

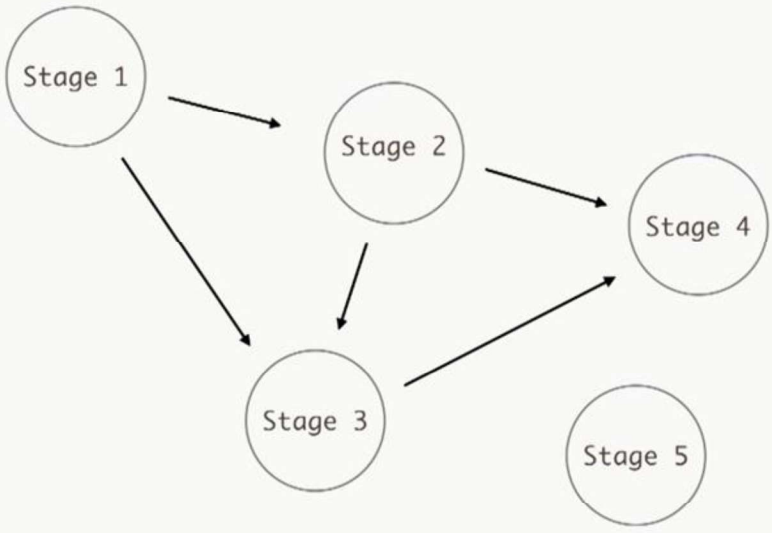
# DAG


Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

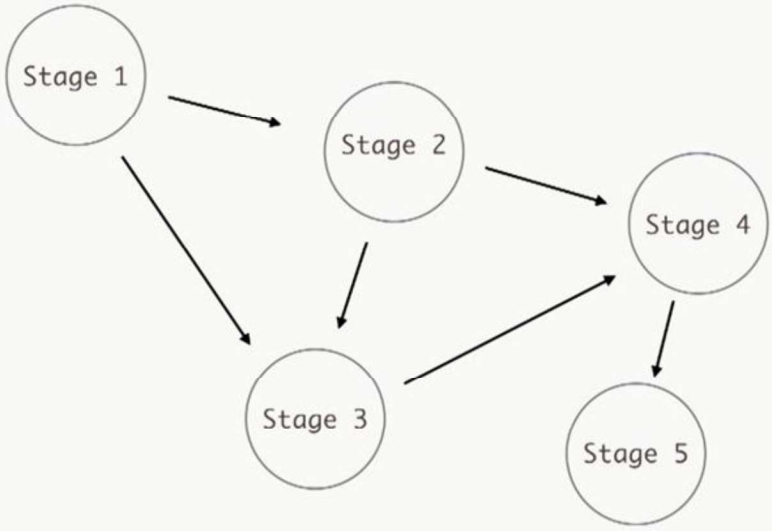
# DAG


Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

# DAG

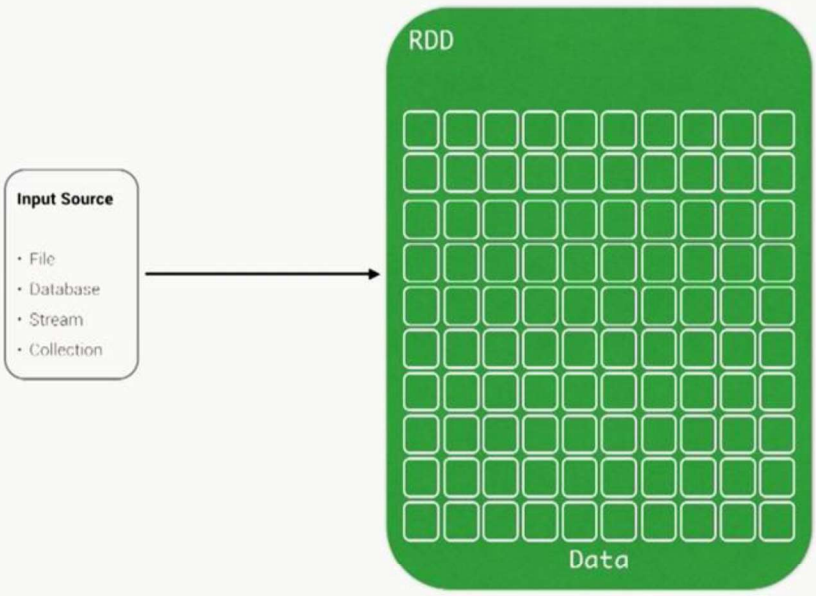

Stage 1
Stage 2
Stage 4
Stage 3
Stage 5

# DAG

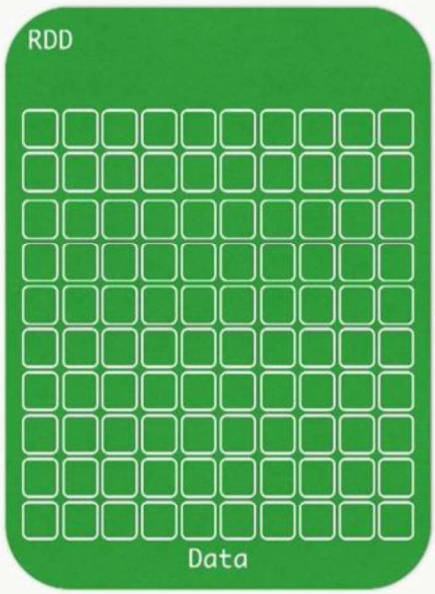



# RDD

# RDD

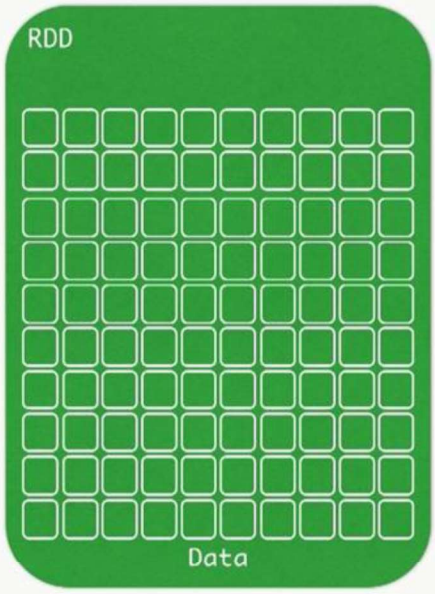



# RDD

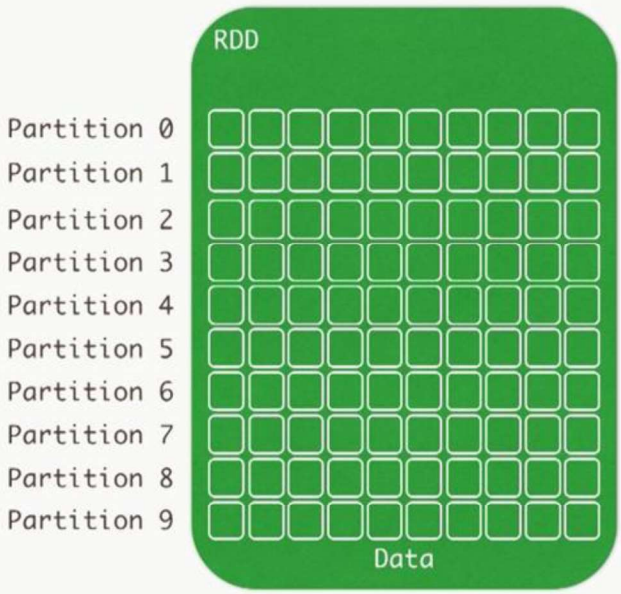



.count() -> 100

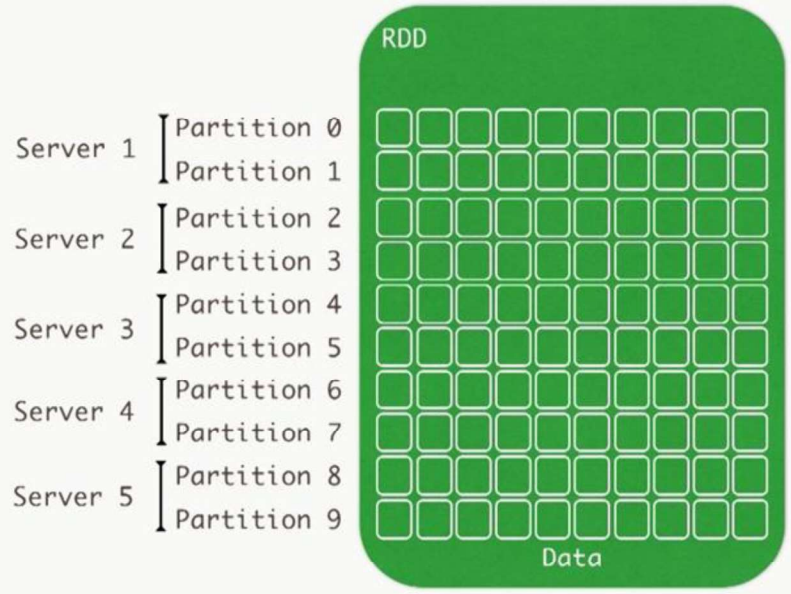
# Partitions



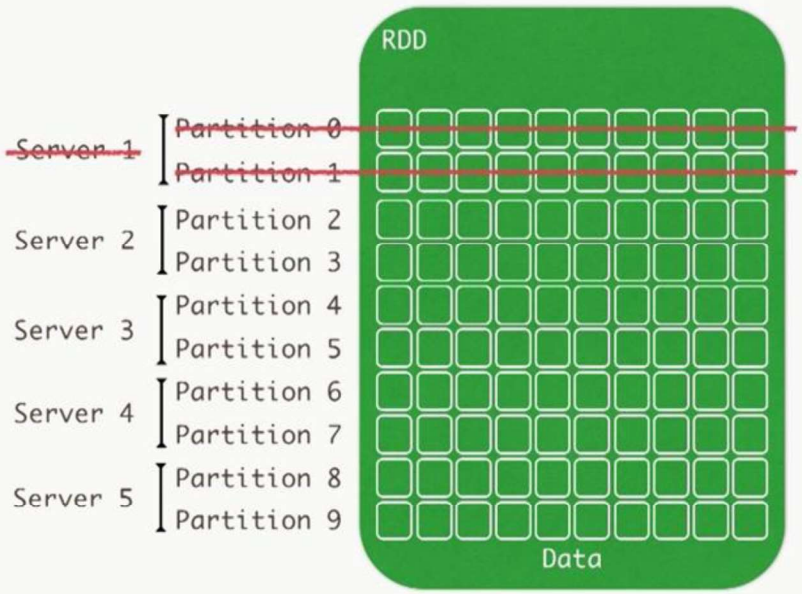

# Partitions



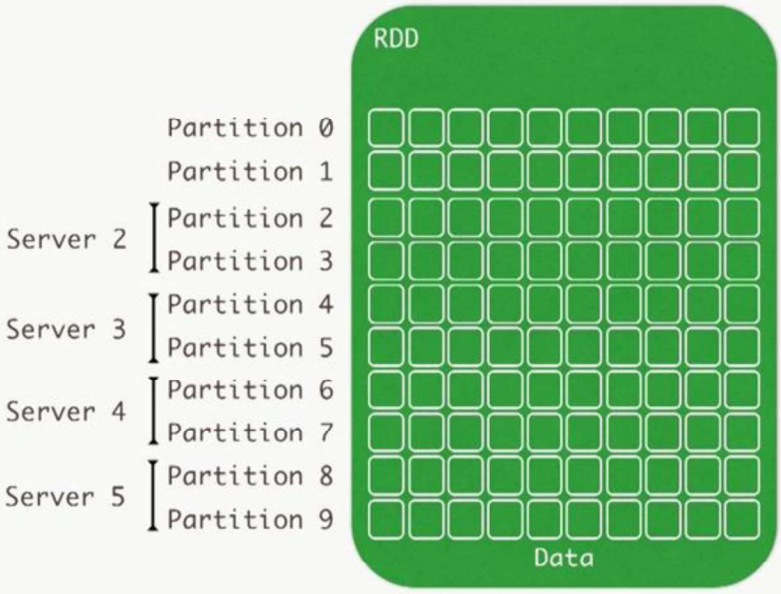
# Partitions



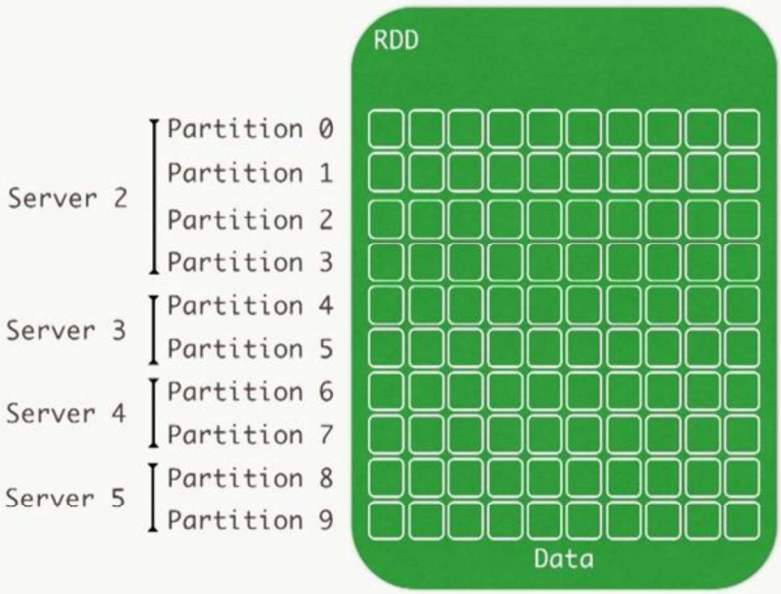

# Partitions



# Partitions




# Partitions

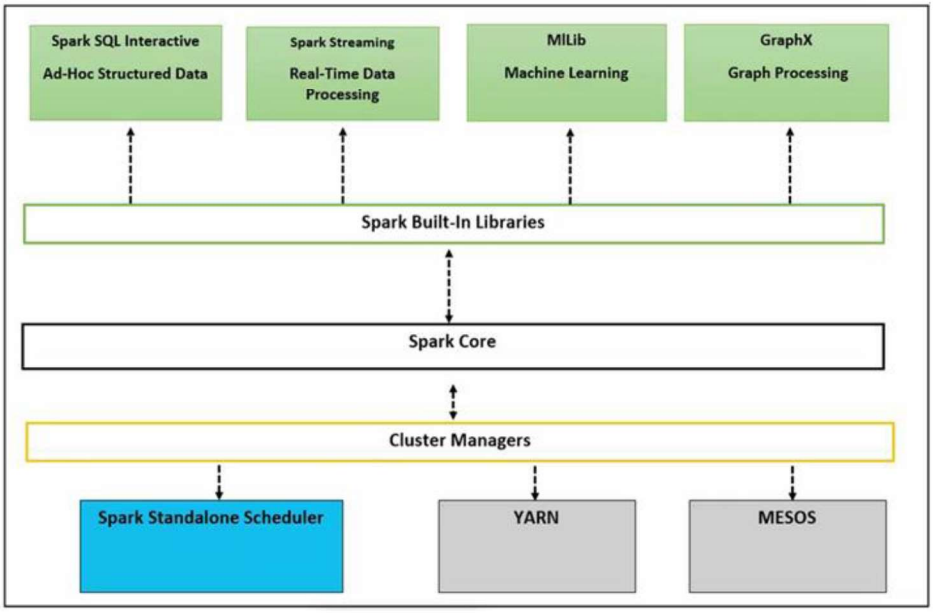


# Spark Framework

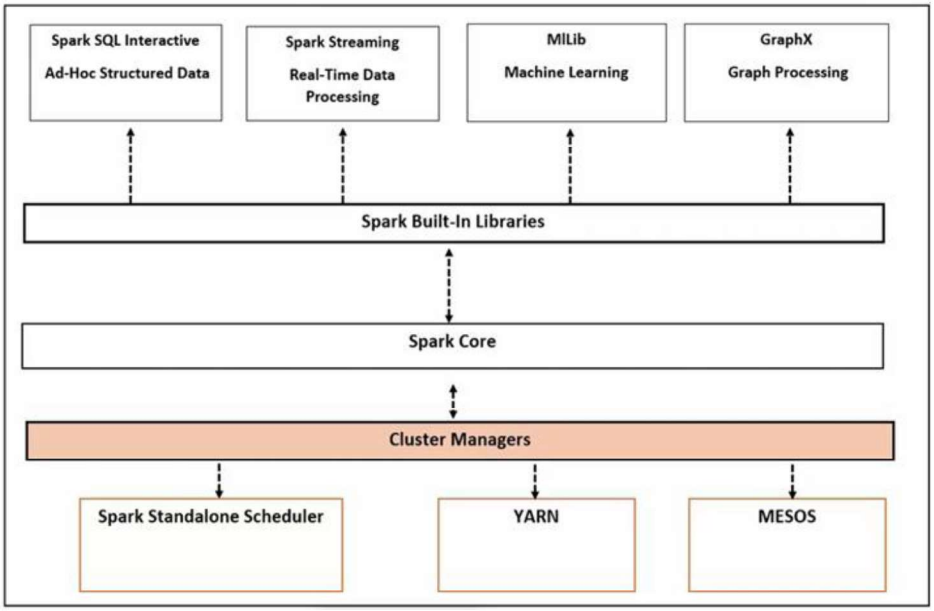
- Spark components are designed to be efficient and integrated
- The various workloads can be leveraged either independently or in combination
- Spark framework consists of 3 main components

1. Cluster Managers
2. Spark Core
3. Spark Built-In Libraries

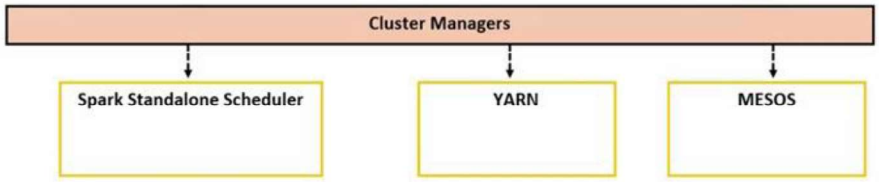
# Spark Framework



# Cluster Managers



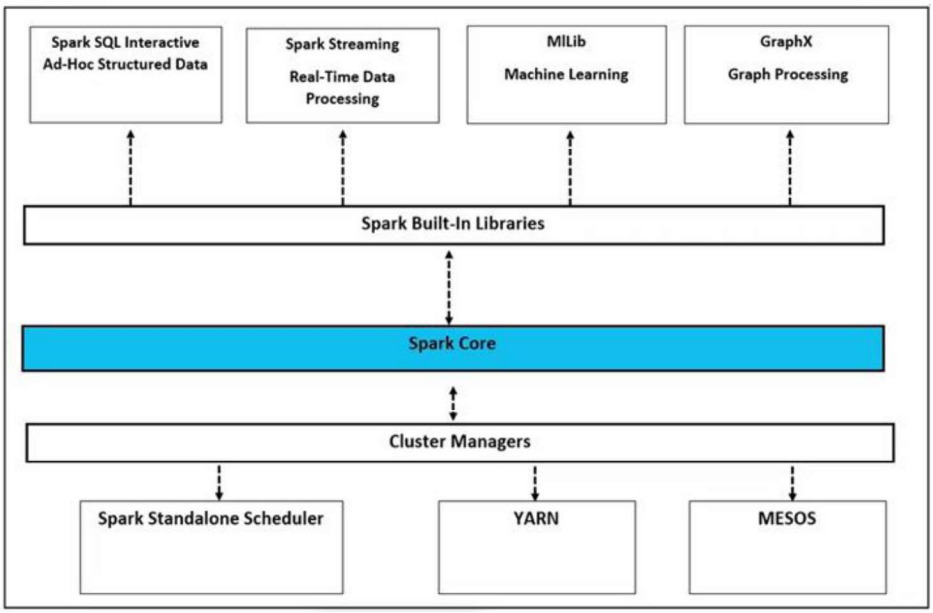

# Cluster Managers




Cluster Managers

- Spark provides efficient and effective scaling
  - Can operate on one to several thousand nodes
  - Standalone Scheduler cluster manager included with Spark
  - YARN and Mesos cluster managers used to deploy Spark on a shared cluster

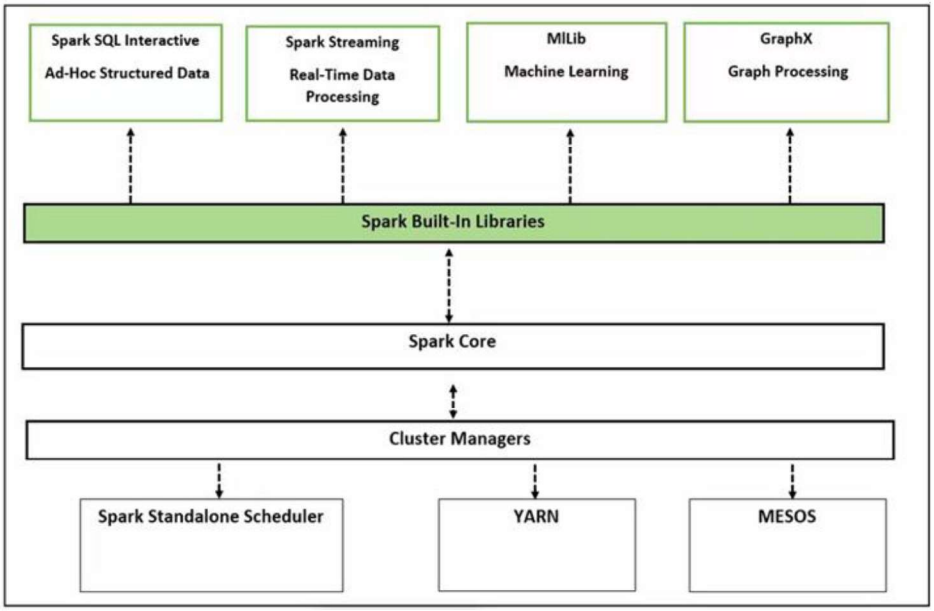
## Spark Core



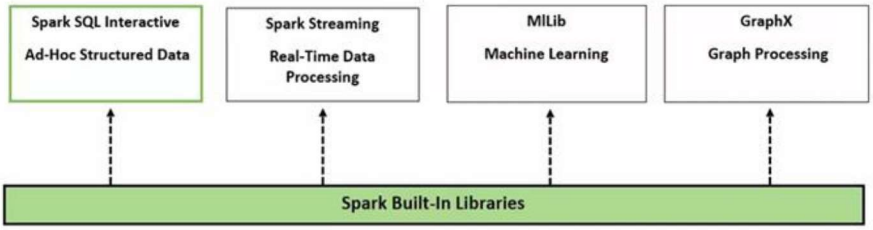

## Spark Core


- Encompasses essential functionality
  - Task scheduling
  - Memory management
  - Fault recovery
  - Interacting with storage systems e.g. HDFS, Hive etc.
  - Contains the API that defines resilient distributed datasets

## Spark Built-in Libraries



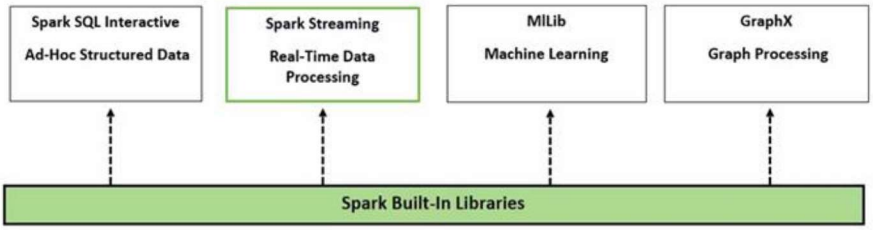

## Spark SQL




- Enables SQL querying on structured data
- Can query with SQL or Hive Query Language (HQL)
- Support exists for various data types e.g. JSON, Hive tables, Parquet
- SQL queries can be mixed with data manipulations supported by RDDs
- Spark SQL replaces the older Shark (Spark on Hive) component

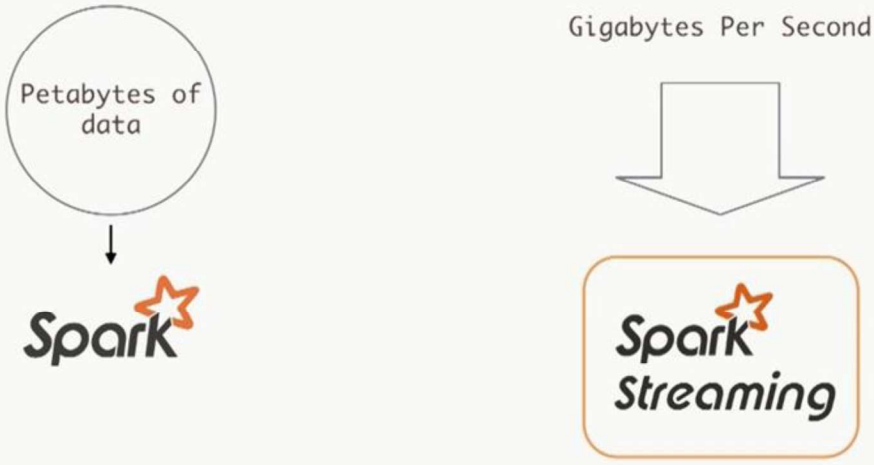
# SparkSQL


# Spark Streaming

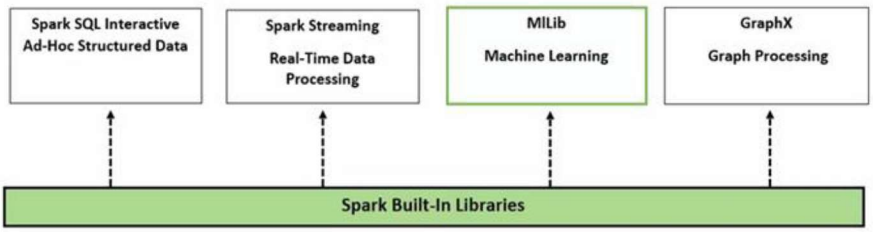



- Enables Spark to process data in real-time as it is ingested
- Data can be ingested:
  - From Hadoop tools e.g. Flume, Kafka
  - Directly into Spark
  - From custom data sources
- Examples of real-time data streams:
  - Social Media feeds (Twitter, Facebook etc.)
  - Log files generated by web servers
  - E-commerce payment transactions

# Spark Streaming



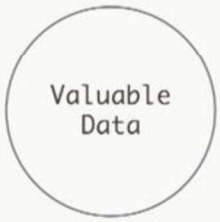

## MlLib



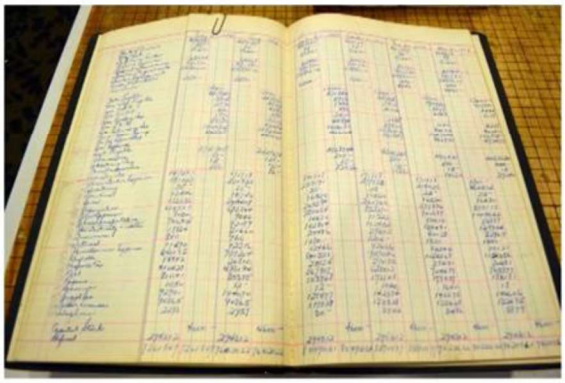

- Provides summary statistics on data, and can perform operations such as:
  - Correlations
  - Hypothesis testing
  - Several machine learning algorithms e.g. clustering, classification, regression
- As is the case with Spark, MlLib continues to evolve
- Designed to efficiently scale across nodes
- Supports various functionalities such as:
  - Data Import
  - Model Evaluation

# MLLib

?



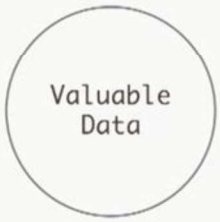

# Time Series Data?



| Step 1 | Step 2 | Step 3 |
| --- | --- | --- |
| Collect Data | ? | Profit! |

# MLLib

?

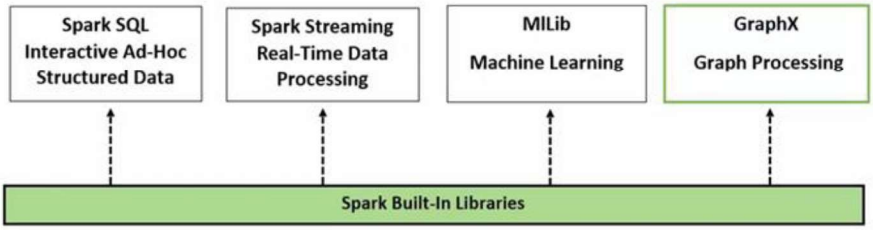



**Statistics** - About my data
- Summaries
- Correlations
- Hypothesis Testing
- Sampling

**Classifiers** - Identify a thing
- Linear Models
- Naive Bayes
- Decision trees

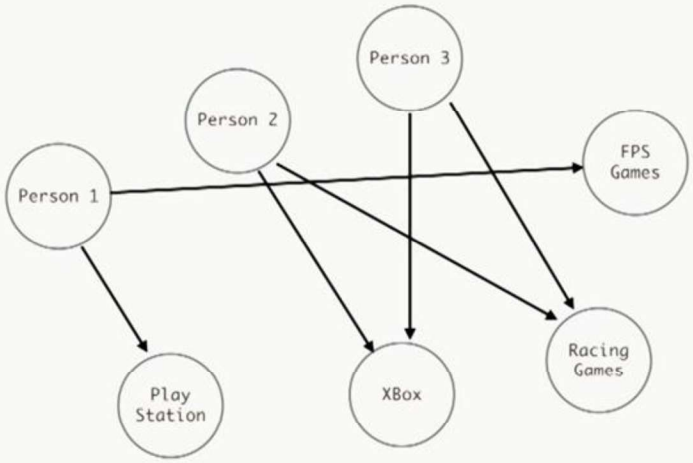
**Clustering** - Grouping similar things
- K-Means

# GraphX

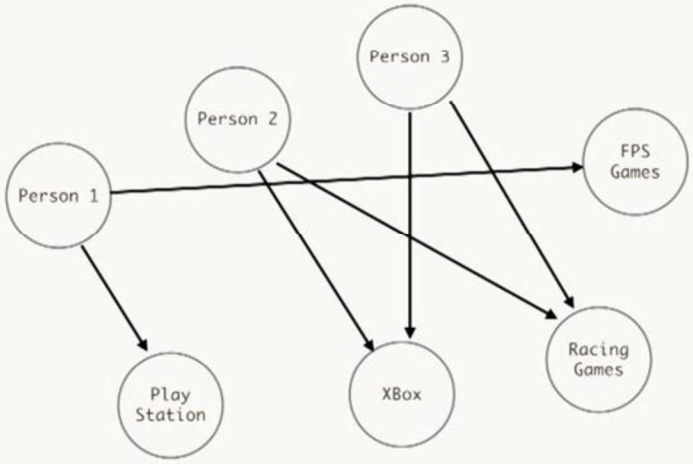



- Provides capability to analyze and manipulate graph data and perform graph computations
- Allows integration of exploratory analysis, Iterative graph computation and ETL
- Can view data as either a graph or a collection
- Several graph algorithms are supported including:
  - Connected Components
  - PageRank
  - Label Propagation
  - Triangle Count

# GraphX

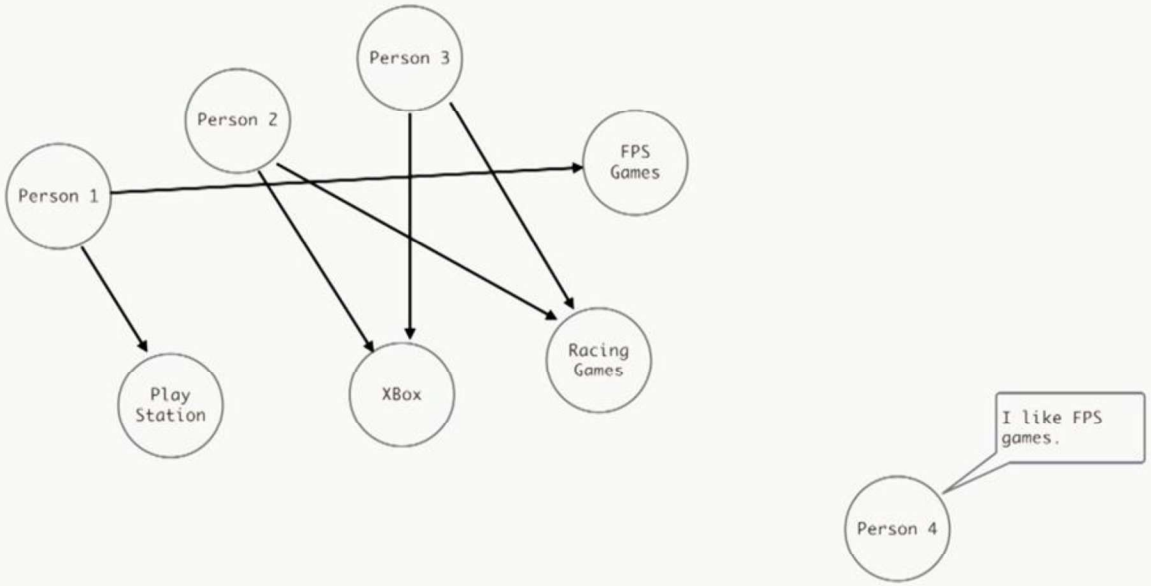



# GraphX




Person 4

# GraphX




## Spark Deployment Modes

- Spark can be deployed in 3 main modes
  1. Local stand-alone
  2. Stand-alone cluster
  3. Shared cluster

- Criteria for choosing a deployment mode:
  - Resources available
  - Type of data processing required and how resource intensive it is
  - Available technology infrastructure
  - Whether or not Cloud services will be utilized

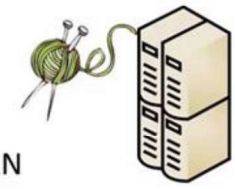
## 1. Local Stand-Alone Deployment

- Everything run locally on one machine
- Ideal for learning Spark and becoming familiar with the Spark shell interface
- Not recommended for production systems in an organization even if deployed on a powerful machine
  - Real power of Spark is realized with a cluster

## 2. Stand-alone Cluster

- A separate cluster is dedicated to Spark
- Jobs distributed across various nodes and implemented in parallel on the cluster
- Optimizes CPU and memory usage

## 3. Shared Cluster



- Run on an existing cluster with Mesos or YARN
- Can coexist with other applications e.g. Flume, Hbase etc.
- Spark jobs can directly read
  - Data previously loaded with Pig
  - Data stored in Hive

## Introduction to Scala

- Scala stands for **Sca**lable **La**nguage
- Created by Martin Odersky
- Recently introduced multi-paradigm development language
  - Enables concise and elegant code development
  - An object-oriented language with functional programming features

Scala

# **Sca**lable **la**nguage

Scala

# Scala - James Gosling

James Gosling

The father of the Java programming language

"If I were to pick a language to use today other than Java, it would be Scala"

## Scala

- Runs on the Java Virtual Machine
  - This allows the use of Scala with Java or as an alternative to Java
  - Can leverage existing Java libraries and tools
- Can be used for applications of varying scale

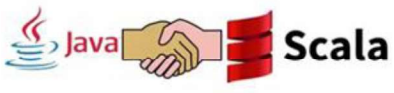

- Combines the strengths of functional and object-oriented languages
  - Functional constructs enable design of new code from simple components
  - OO constructs facilitate the structuring of larger systems and provide flexibility

## Scala Features

1. Compatible with Java
2. Concise
3. High-level
4. Statically typed

## 1. Compatible with Java

- Seamlessly integrates with Java
- Scala code compiles to JVM bytecodes
- Run-time performance very similar to Java's
- Scala code can call Java methods, access fields, inherit classes and implement interfaces



## 2. Concise Programming Language

- Lines of code can be half to a factor of ten less than Java
- Conciseness is due to:
  - Avoiding "boilerplates" that are mandatory in Java (e.g. semicolons)
  - Powerful libraries can be defined and re-used

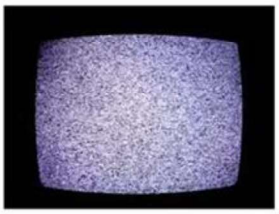

## 3. High-Level Language

- Allows developers to manage code complexity by elevating the level of abstraction in the interfaces used
  - E.g. Java code deals with strings as low-level entities and goes through each character using a loop

## 4. Statically Typed

- A "static typed" environment classifies **Variables** and **Expressions** based on the type of values they contain and/or compute

- Provides a robust and powerful mechanism to manage static types. Developers can:
  - Parameterize types with *generics*
  - Hide details using *abstract types*



## Enterprises using Scala Applications

| | |
|---|---|
| LinkedIn | Twitter |
| Électricité de France Trading | Novell |
| Xerox | Sony |
| Siemens | Reaktor |
| AppJet | Thatcham |
| OPOWER | GridGain |
| FourSquare | Xebia |
| The Guardian | ... |

**Scala in the Enterprise**
http://www.scala-lang.org/old/node/1658

# Enterprises using Scala Applications

Twitter:

- Scala is fast, functional, expressive, statically typed, object-oriented, concurrent, and java compatible
- **Actors** keep concurrency simple
- **Immutability** keeps concurrency predictable
- **Type inference** keeps the code clean
- **Traits** enable a high level of code reuse

Électricite de France Trading:

- Scala brings all the features of other programming languages together

OPOWER:

- Java and Scala are statically typed languages vs. Ruby and Clojure are dynamically typed
- Java and Ruby: Object Oriented, Clojure: Functional Programming
- Scala: both Object-Oriented and Functional Programming

LinkedIn:

- Scala is Multithreaded and asynchronous programming language Using actors in Scala makes this much easier

**Scala in the Enterprise**
http://www.scala-lang.org/old/node/1658

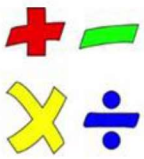
# Main advantages of Scala:

- Concise, Clean, Professional
- Agile, scalable, fast, and interoperable with Java
- Suitable to handle lots of Data and handle it in real-time
- Suitable for parallel execution using multi-threaded and asynchronous programming features
- Enables code re-use with traits
- Used in many open source and Big Data projects
- The language for Spark, the most on demand and so far the best framework for big data

## Base Scala Types


- Integer types: Byte, Short, Int, Long
  - 8,16,32,64 bit signed Integer types
- Float types: Float, Double
  - 32,64 bit float types
- Char – 8 bit unsigned Integer
- String – Sequence of characters
  - Like in Java, maximum String length is $2^{31}$ characters
- Boolean – logical value true / false
- Every data element is an object, including base types
  - E.g. 5 is an object of type Int and Int is the class name of the type

## Basic Scala Operators



- Arithmetic operators: + - * / %
- Relational operators: == != > < >= <=
- Logical operators: && || !
- Bitwise operators: & (and) | (or) ^ (XOR) ~ (binary ones complement) << (left shift) >> (right shift) >>> (right shift zero fill)
- Assignment operators = += -= *= /= %= <<= >>= &= |= ^=

## Scala Variables

- Scala has two ways of defining variables:
  - val – defines a constant (immutable variable)
  - var – defines a variable which can change its value

- Variable definition
  - var|val <name> [: <type>] = <value>
    - <name> - identifier name of the variable
    - <type> - class name of the variable type, optional if = <value> is provided
    - <value> - value assigned to the variable

## Immutability

```
val a = 1

a = 2 // does not work

a = a + 1 // does not work

var b = 1
b = 2
b = b + 1
```

"val" is immutable in the sense once you bind a value to a symbol defined as "val", the binding can't be changed

# The Scala REPL

```
scala> 2*3
res0: Int = 6

scala> val a = 10
a: Int = 10

scala> val b = a + 1
b: Int = 11

scala> a = a + 1
<console>:8: error: reassignment to val
       a = a + 1
         ^

scala>
```

# Basic Types and Type inference

Types: Byte, Short, Int, Long, Float, Double, Char, String, Boolean

Variables: val and var

| Example | Notes |
|---|---|
| val x = 1 | Define x as immutable value of 1 Int |
| x = 2 | Gives "error: reassignment to val" |
| var y = 1 | Define y as mutable value of 1 Int |
| y = 2 | No error |

In above examples, Scala uses type interfence to determine the variable x and y are of type Int.
Note types begin with an uppercase letter. The keywords val and var begin with a lowercase letter.

# Basic Types and Type inference

| Example | Notes |
| --- | --- |
| val a: Double = 10<br>val num = Array(1, 2, 3)<br>val matrix = Array(Array(), Array())<br>val num: List[Int] = List(1, 2, 3) | val Name: Type = Value<br><br>If the type is left out, Scala uses type inference. |
| val a = new Array[String](3)<br>a(0) = "abc"<br>a(1) = "def"<br>a(2) = "gh"<br>a | Arrays contain mutable objects with index starting at zero. The contents of a cell can be changed after declaration. |
| val b = Array("abc", "def", "gh")<br>b(2) = "ghi"<br>b | Using type inference to define an array of strings |

## Function Definition

```
def add(a:Int, b:Int):Int = a + b

val m:Int = add(1, 2)

println(m)
```

Note the use of the type declaration "Int". Scala is a "statically typed" language. We define "add" to be a function which accepts two parameters of type Int and returns a value of type Int. Similarly, "m" is defined as a variable of type Int.

## Function Definition

```
def fun(a: Int):Int = {
  a + 1
  a - 2
  a * 3
}

val p:Int = fun(10)
println(p)
```

**Note!**

There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

## Type Inference

```
def add(a:Int, b:Int) = a + b

val m = add(1, 2)

println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

## Type Inference

```
def add(a, b) = a + b

val m = add(1, 2)

println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML. Scala is said to do local, "flow-based" type inference while Haskell/ML do Hindley-Milner type inference

## Expression Oriented Programming

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"

println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

## Expression Oriented Programming

```
def errorMsg(errorCode: Int) = errorCode match {
  case 1 => "File not found"
  case 2 => "Permission denied"
  case 3 => "Invalid operation"
}

println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern.

## Lazy Evaluation

- Do things as late as possible and never do them twice
- Lazy values provide a way to delay initialization of a value until the first time it is accessed
- When dealing with values whose computational cost is significant, lazy evaluation may avoid the computation of the values which is not needed in that particular execution flow
- Scala does not use lazy by default

## Lazy Vals

```
def hello() = {
  println("hello")
  10
}

val a = hello()
```

The program prints "hello" once, as expected. The value of the val "a" will be 10.

## Lazy Vals

```
def hello() = {
  println("hello")
  10
}

lazy val a = hello()
```

Strange, the program does NOT print "hello"! Why? The expression which assigns a value to a "lazy" val is executed only when that lazy val is used somewhere in the code!

## Lazy Vals

```
def hello() = {
  println("hello")
  10
}

lazy val a = hello()

println(a + a)
```

# Built-In Control Structures

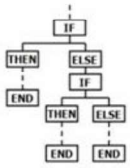
- if expression
- while loop
- for comprehension
- match - case
- Exception handling with try expression
- Function Calls in Scala

Control structures almost always produce a value.

The keywords begin with a lowercase letter.

# If Expression

- Evaluation based on a condition:
  if ( <condition> ) <expressionTrue> [ else <expressionFalse> ]
  <condition> - logical expression
  <expressionTrue> - result when <condition> is true
  <expressionFalse> - result when <condition> is false



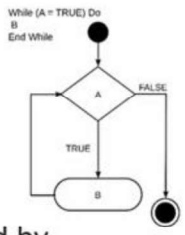

- If expression and its result are both objects
  - Unity (no return value) is allowed as a result
    - E.g. println function returns Unity

# Built-In Control Structures: if

Scala's if works just like many other languages but it always produces a value.

| Example | Notes |
|---|---|
| val a = 1<br>if (a < 2) true else false | Produces Boolean value true |
| val b = 3<br>if (b < 0) {<br>  -1<br>} else if (b == 0) {<br>  0<br>} else {<br>  +1<br>} | Blocks are enclosed by curly braces {}<br><br>Scala has else if similar to many other languages |
| val fn = if (a < b) "a" else "b" | We can assign a variable to the result of a control structure |

# While Loop Expression




- Evaluates an expression while a condition is met
  - while ( <condition> ) <expression>
    - <condition> - a Boolean expression
    - <expression> - evaluated while <condition> is true
- Generally, <condition> contains a variable modified by <expression>
  - Otherwise <expression> is never evaluated or is evaluated infinitely
- The <expression> does not need to return a value. Often a construct {<expression>;<expression>…} is used to simulate a while-loop in Java or similar languages.

# Built-In Control Structures: while

| Example | Notes |
|---|---|
| var sum = 0<br>var i = 10<br>while (i < 20) {<br>  sum = sum + i<br>  i = i + 1<br>}<br>sum | A while loop executes until the condition is false. It does not execute if the condition begins as false. |
| var sum = 0<br>var i = 10<br>do {<br>  sum = sum + i<br>  i = i + 1<br>} while (i < 10)<br>Sum | A do-while loop executes once when the condition begins as false. |

# For Loop Expression

- Evaluates an expression for each element of a collection

  for ( <variable> <- <collection> ) <expression>

  <variable> - control variable

  <collection> - control variable is sequentially assigned the value of each member of the <collection>

  <expression> - evaluated for each member of the <collection>

- Range syntax <from> to <to> [by <step>] is commonly used as a collection, where both <from> and <to> are inclusive

# Built-In Control Structures: for

| Example | Notes |
|---|---|
| val c = Array("abc", "def", "ghi")<br>for (i <- c) println(i) | Iterate over a collection using <- symbol |
| for (i <- 1 to 3) println(i) | The syntax 1 to 3 gives the collection: (1, 2, 3) |
| for (i <- 1 until 3) println(i) | Use until to omit the right hand end point: (1, 2) |
| for {<br>i <- 1 until 9<br>if i > 6<br>} yield i | Here is a for that produces a Vector in the range 1 until 9 which are greater than 6. Use yield to extract the desired values into a collection. |

# Match–Case Expression

- Extended conditional expression with multiple conditions

```
<expression> match {
    case <value1> => <result1>
    case <value2> => <result2>
          …..
    [ case _ => <defaultResult> ]
}
```

- A case <value> must exactly match the value of <expression> to apply the corresponding <result>
  - i.e. <expression> == <value> must evaluate to true
- Default value _ is used when no other match is made

# Built-In Control Structures: match - case

Similar to switch statement in Java but again Scala produces a value.

```
x match {
        case PATTERN1 => R1
        case PATTERN2=>R2
        case _ => Rn
}
```

| Example | Notes |
|---|---|
| val country = "Honduras"<br>val currency =<br>country match {<br>case "Canada" => "Dollar"<br>case "Honduras" => "Lempira"<br>case "Thailand" => "Baht"<br>case _ => "Unknown"<br>} | The underscore _ is used for matching the default case. |

# Exception handling with try expression

**Example**

```
import java.util.NoSuchElementException
val country = "France"
val currency =
 country match {
  case "Canada" => "Dollar"
  case "Honduras" => "Lempira"
  case "Thailand" => "Baht"
  case _ => throw new NoSuchElementException()
 }
```

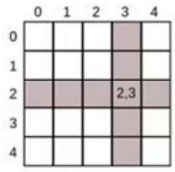
```
import java.io._
val data = try {
 val fp = new FileReader("missing.txt")
} catch {
 case e: FileNotFoundException => "no file"
 case e: IOException => "IO exception"
 case _: Throwable => "unexpected error"
}
```

# Scala Collections

→ Scala has a rich library of Collections, they are:

- » Array
- » ArrayBuffers
- » Maps
- » Tuples
- » Lists

## Arrays




- Arrays are mutable lists of elements
  - Elements can be changed after creation
- Definition

  val <name> = Array[<type>](<size>)

  val <name> = Array(<list-of-elements>)
- Common Array operations

  <name>(<index>) – array element access (<index> is zero-based)

  Array.concat(<array1>, <array2> ...) – concatenate arrays

  Array.range(<from>,<to>[,<step>]) – create an array containing a range of numeric elements
- Scala does not directly support multidimensional arrays

## Scala Arrays

→ Fixed Length Arrays:
→ Examples:

```
val n = new Array[Int](10)
val s = new Array[String](10)
val st = Array("Hello", "World")
```

# Scala ArrayBuffers

→ Variable Length Arrays (Array Buffers)
→ Similar to Java ArrayLists

```
import scala.collection.mutable.ArrayBuffer
val a = ArrayBuffer[Int]()
a +=1
a+= (2,3,5)
a++=Array(6,7,8)
```

# ArrayBuffers

→ Common Operations:

```
a.trimEnd(2)   //Removes last 2 elements
a.insert(2, 9)  // Adds element at 2nd index
a.insert (2,10,11,12)  //Adds a list
a.remove(2)   //Removes an element
a.remove(2,3)  //Removes three elements from index 2
```

→ Traversing and Transformation:

```
for (el <- a)
        println(el)

        for (el <-a if el%2 == 0) yield (2*el)
```

# Arrays & ArrayBuffers

→ Common Operations:

```
Array(1,2,3,4).sum
Array(1,5,9,8).max
val a = Array(1,7,2,9)
scala.util.Sorting.quickSort(a)
a.mkString( " ** ")
```

# Collections

- Scala defines a rich set of collections, iterable sets of elements:

List – linked list of elements of the same type
Set - set of pairwise different elements (each element can be contained only once)
Map – collection of key-value pairs
Tuple – collection of objects of various types (unlike List or Array)
Option – zero or one element of given type
Range – collection of values from a defined range

# Scala Maps

→ In Scala, a map is a collection of Pair
→ A pair is a group of two values ( Not necessarily of same type)

```
val mapping = Map("Vishal" -> "Kumar", "Vijay" -> "Verma")
val mapping = scala.collection.mutable.Map("Vishal" -> "K", "Vijay" -> "V")
```

# Scala Maps cotd.

→ Accessing Maps:

```
val x = mapping("Vishal")
val x = mapping.getOrElse("Vish", 0)
mapping -= "Vishal"
mapping += ("Ajay" -> "Sharma")
```

→ Iterating Maps:

```
for ((k,v) <- mapping) yield (v,k)
```

# Scala Tuples

→ Tuple is more generalized form of pair

→ Tuple has more than two values of potentially different types

```
val a = (1,4, "Bob", "Jack")
```

→ Accessing the tuple elements:

```
a._2  or a _2//Returns 4
```

→ In tuples the offset starts with 1 and NOT from 0

→ Tuples are typically used for the functions which return more than one value:

```
"New Delhi India".partition(_.isUpper)
```

# Lists

- Ordered immutable list of elements
  - Elements cannot be changed once the list is created
- Definition:

  val <name> : List [<type>] = List(<element>,...)
- Basic List operations:

  <element> :: <list> - add an element to the beginning (pronounced cons)
  <list> ::: <list> - concatenate two lists
  List.concat(<list1>,<list2>...) – concatenate lists
  <list>.head – first member of the list
  <list>.tail – a list containing all but the first member
  <list>.isEmpty – false if the list contains at least one member
  Nil – empty list
  <list>.size – number of members in the list
  <list>(<index>) – reference a list member, <index> is zero-based

# Lists

→ List is either Nil or a combination of head and tail elements where tail is again a List
→ Example:

```
val lst = List(1,2)
lst.head = 1
lst.tail = List(2)
```

→ :: operator adds a new List from given head and tail

```
2 :: List(4,5)
List[Int] = List(2, 4, 5)
```

# Lists cotd.

→ We can use iterator to iterate over a list, but recursion is a preferred practice in Scala

→ Example:

```
def sum(l :List[Int]):Int = {if (l == Nil) 0 else l.head + sum(l.tail)}
val y = sum(lst)
```

## Some List Operations

```
val a = List(1,2,3)
val b = Nil
val c = List()
val d = 0::a
val e = 0::b

println(b)
println(c)
println(d) // List(0,1,2,3)
println(e) // List(0)
```

- Nil and List() are both "empty" lists
- a::b returns a new list with "a" as the first item (the "head") and remaining part b (called the "tail")

## Pattern Matching

```
def fun(a: List[Int]) = a match {
  case List(0, p, q) => p + q
  case _  => -1
}

println(fun(List(0, 10, 20)))
println(fun(List(0, 1, 2, 3)))
println(fun(List(1, 10, 20)))
```

Pattern matching helps you "pull-apart" complex datastructures and analyse their components in an elegant way. In the above case, the function will return the sum of the second and third elements for any three element List of integers starting with a 0 - the value -1 is returned for any other list

## Pattern Matching

```
def fun(a: List[Int]) = a match {
  case List(0, p, q) => p + q
  case List() => -1
}
```

The compiler warns us when it detects that the match is not "exhaustive"? Why do we get a warning in the above case?

## Pattern Matching

```
def length(a: List[Int]):Int = a match {
  case Nil => 0
  case h::t => 1 + length(t)
}

println(length(List()))
println(length(List(10,20,30,40)))
```

A simple recursive routine for computing length of a list. If list is empty, length is 0, otherwise, it is 1 + length of remaining part of the list.

If we have a List(1,2,3) and if we match it with a "case h::t", "h" will be bound to 1 and "t" will be bound to the "tail" of the list, ie, List(2,3)

## Scala Functions

- Group of statements which perform a task and can return a value
- Defining a function

```
def <name> (<list-of-parameters>) [: <result-type>|Unit] = {
              <statements>
       return <expression>
}
```

- Invoking a function

  <name> (<list-of-arguments>)

- Function that returns no result (Unit) is called procedure

# Functional Programming

- One of the main strengths of Scala is the Functional paradigm
- Functional paradigm helps to handle programs when they get larger:
  - By dividing them into smaller, more manageable pieces
- In Scala, functions have no "return" statement, the value of last expression is returned
- One of the first places you'll start to use functional constructs is with the collections:
  - Considering the collection as a list of items, the idea is to apply a function to each item in the list in some way

Example

```
object FunctionDefinition {
def add (a:Int, b:Int) : Int =a+b                    //> add: (a: Int, b: Int)Int
val m:Int = add(1,2)                                 //> m  : Int = 3
println(m)                                           //> 3
```

# Function Calls in Scala

Functions in Scala are first-class values:

- they can be passed as a parameter
- they can be returned as a result

Functions which use a function as a parameter or as a result is a **Higher Order Function.**

Functions always result in a value. The final line of a function becomes the result.

In the previous examples, we have used operators such as +, <, ==, ...

Operators in Scala are actually infix notation calls to functions. A function in Scala taking a single parameter can be invoked using either the infix notation or the familiar syntax with parenthesis.

| Example | Notes |
|---|---|
| myObject.foo(42) | Using parenthesis to call a function |
| myObject foo 42 | Using infix notation to call a function |
| myObject + 42 | Using infix notation to call + function on myObject with argument 42 |

# Function Calls in Scala

**Example**

```
def plus(x: Int, y: Int): Int = {
  println("Sum of " + x + " and " + y + " is " + (x+y))
  x + y
}
plus(1,2)
```

```
def sumInts(a: Int, b: Int): Int = {
  if (a > b) 0 else a + sumInts(a + 1, b)
}
sumInts(1,5)
```

```
def square(x: Int) = x * x
def sumSquares(a: Int, b: Int): Int = {
  if (a > b) 0 else square(a) + sumSquares(a + 1, b)
}
sumSquares(1,5)
```

# Function Calls in Scala

**Example**

```
def powerOfTwo(x: Int): Int = {
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)
}
powerOfTwo(1)
powerOfTwo(2)
powerOfTwo(3)
powerOfTwo(4)
```

```
def sumP2(a: Int, b: Int): Int = {
  if (a > b) 0
  else powerOfTwo(a) + sumP2(a + 1, b)
}
sumP2(1,3)
sumP2(1,4)
sumP2(3,8)
```

# Function Calls in Scala

**Example**

```
def square(x: Int) = x * x
def powerOfTwo(x: Int): Int = {
  if (x == 0) 1 else 2 * powerOfTwo(x - 1)
}
def hoSum(f: Int => Int, a: Int, b: Int): Int = {
  if (a > b) 0
  else f(a) + hoSum(f, a + 1, b)
}
def sumSquares(a: Int, b: Int): Int =
  hoSum(square, a, b)
def sumP2(a: Int, b: Int): Int =
  hoSum(powerOfTwo, a, b)

sumSquares(1,5)
sumP2(3,8)
```

# Higher Order Functions

→ Functional languages treat functions as first-class values

→ This means that, like any other value, a function can be passed as a parameter and returned as a result

→ This provides a flexible way to compose programs

→ Functions that take other functions as parameters or that return functions as results are called higher order functions

# Examples

→ Take the sum of the integers between a and b:

```
def sumInts(a: Int, b: Int): Int =
if (a > b) 0 else a + sumInts(a + 1, b)
```

→ Take the sum of the cubes of all the integers between a and b :

```
def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

# Examples cotd.

→ Take the sum of the factorials of all the integers between a and b:

```
def sumFactorials(a: Int, b: Int): Int =
if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)
```

→ These are special cases of below for different values of f

$$\sum_{n=a}^{b} f(n)$$

→ Can we factor out the common pattern?

# Examples cotd.

→ Let's define:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
if (a > b) 0
else f(a) + sum(f, a + 1, b)
```

→ We can then write:

```
def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

→ Where

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if (x == 0) 1 else fact(x - 1)
```

# Commonly used HOFs

→ Map

```
(1 to 9).map(0.1 * _)
```

→ foreach

```
(1 to 9).map("*" * _).foreach(println _)
```

# Commonly used HOFs cotd.

→ filter

```
(1 to 9).filter(_ % 2 == 0)
```

→ reduceLeft

```
(1 to 9).reduceLeft(_ * _)
```

→ Split, sortWith

```
"Mary had a little lamb".split(" ").sortWith(_.length < _.length)
```

# Anonymous Functions


- Function used on-the-fly, without prior definition
  - Typically passed as an argument to other functions
- Definition

  (<list-of-arguments>) => <function body>
- Usage
  - When a function expects another function as argument

```
def myhofunction( f:Int => Int): Int = {
            return f(1) + f(2) + f(3)
        }
```

  - myhofunction ((a:Int)=> a*a) = 14 (1*1 + 2*2 + 3*3 = 14)
  - myhofunction ((a:Int)=> a+2) = 12 (1+2 + 2+2 + 3+2 = 12)

# Anonymous Functions

→ Passing functions as parameters leads to the creation of many small functions
  » Sometimes it is tedious to have to define (and name) these functions using def

→ Compare to strings: We do not need to define a string using def

→ Instead of

```
def str = "abc"; println(str)
```

→ We can directly write

```
println("abc")
```

→ Because strings exist as literals. Analogously we would like function literals, which let us write a function without giving it a name

→ These are called anonymous functions

# Anonymous Functions cotd.

→ Example: A function that raises its argument to a cube:

```
(x: Int) => x * x * x
```

→ Here, (x: Int) is the parameter of the function, and x * x * x is it's body.

  » The type of the parameter can be omitted if it can be inferred by the compiler from the context

→ If there are several parameters, they are separated by commas:

```
(x: Int, y: Int) => x + y
```

# Anonymous Functions and syntactic sugar

→ An anonymous function (x1 : T1; :::; xn : Tn) ) E can always be expressed using def as follows:

```
def f(x1 : T1; :::; xn : Tn) = E; f
```

→ where f is an arbitrary, fresh name (that's not yet used in the program)

→ One can therefore say that anonymous functions are syntactic sugar

→ Using anonymous functions, we can write sums in a shorter way:

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)

def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

# Closures

→ In Scala, the functions can be defined anywhere, in a package or class or inside another function or method

→ We can access the variables of enclosing scope from the function

→ Thus the functions which access out of bound variables (variables not in scope) are called Closures

→ Example:

```
def mulBy(factor : Double) = (x : Double) => factor * x
```

→ Now consider following:

```
val triple = mulBy(3)
val half = mulBy(0.5)
println(triple(14) + " " + half(14))
```

→ Here, each of the returned function has its own setting for factor

→ This is called Closure. So, a Closure comprises of code along with the definition of non-local variables used by the code

# Currying

→ Currying is the process of converting a function which takes two arguments to the function which takes one argument

→ That function returns a function, which consumes the second argument

→ Example:

```
def mul(x: Int, y: Int) = x * y
```

→ The above function takes two arguments

→ The same function can be re-written as:

```
def mulOneAtATime(x: Int) = (y: Int) => x * y
```

→ Now the modified function takes an argument, yielding a function that takes another argument

→ Two multiply two numbers, we call:

```
mulOneAtATime(6)(7)
```

→ Under the hood, the result of mulOneAtATime(6) is the function (y: Int) => 6*y. This function now is applied to 7, which yields 42

# Currying cotd.

→ Scala provides a shortcut for such curried functions:

```
def mulOneAtATime(x: Int)(y: Int) = x * y
```

# Call-By-Value vs. Call-By-Name

- Call-By-Value:
  - Avoids repeated evaluation of arguments

- Call-By-Name:
  - Avoids evaluation of arguments when the parameter is not used at all by the function

- Call-By-Value is usually more efficient but it might loop continuously whereas a Call-By-Name evaluation would eventually terminate

- In Scala usually Call-By-Value is used

# Call-By-Value vs. Call-By-Name

Example

```
def evaluate() = {
  println("Function Calls")
  20 // return value
}                                          //> something: ()Int

def callByValue(x: Int) = {
  println("x1=" + x)
  println("x2=" + x)
}                                          //> callByValue: (x: Int)Unit

def callByName(x: => Int) = {
  println("x1=" + x)
  println("x2=" + x)
}                                          //> callByName: (x: => Int)Unit
callByValue(evaluate())                    //> Function Calls
                                           //| x1=20
                                           //| x2=20


callByName(evaluate())                     //> Function Calls
                                           //| x1=20
                                           //| Function Calls
```

# Recursion

*Recursion: A function can call itself repeatedly*

| Notes | Example |
|---|---|
| If we have a function called factorial: | def factorial(x: BigInt): BigInt =<br>if (x == 0) 1 else x * factorial(x - 1) |
| factorial(5)<br>The evaluation steps are as shown in details =><br>The out put is 120 | **if** (5 == 0) 1 **else** 5 * factorial(5-1)<br>5 * factorial(5-1)<br>5 * factorial(4)<br>5 * (4 * factorial(3))<br>5 * (4 * (3 * factorial(2)))<br>5 * (4 * (3 * (2 * factorial(1))))<br>5 * (4 * (3 * (2 * (1 * factorial(0)))))<br>5 * (4 * (3 * (2 * (1 * 1)))) |

# Tail Recursion

| Example | |
|---|---|
| If we have a function called gcd: | def gcd(a: Int, b: Int): Int =<br>if (b == 0) a else gcd(b, a % b) |
| gcd(14,21)<br>The evaluation steps are as shown<br>In details =><br>The output is 7 | **if** (21 == 0) 14 **else** gcd(21, 14 % 21)<br>gcd(21, 14 % 21)<br>gcd(21, 14)<br>**if** (14 == 0) 21 **else** gcd(14, 21 % 14)<br>gcd(14, 21 % 14)<br>gcd(14, 7)<br>**if** (7 == 0) 14 **else** gcd(7, 14 % 7)<br>gcd(7, 14 % 7)<br>gcd(7, 0)<br>**if** (0 == 0) 7 **else** gcd(0, 7 % 0) |

# Nested Functions

## Example

```
object NestedFunctions {
  println("This program filters out the numbers smaller than  (5) and prints them")
  //> This program filters out the numbers smaller than tereshold (5) and prints them
  def filter(xs: List[Int], threshold: Int) = {
    def process(ys: List[Int]): List[Int] =
      if (ys.isEmpty) ys
      else if (ys.head < threshold) ys.head :: process(ys.tail)
      else process(ys.tail)
    process(xs)
  } //> filter: (xs: List[Int], threshold: Int)List[Int]
  println(filter(List(1, 9, 2, 8, 3, 7, 4), 5)) //> List(1, 2, 3, 4)
}
```

# Common Scala Functions

- Map
  - Apply a function to each element in the list
- Flatten
  - Combines two lists
- FlatMap
  - Combines Mapping and Flattening
- Filter
  - Filter out some elements of list based on some criteria
- fold - foldLeft – foldRight
  - Function fold takes data in one format and gives it back in another format
  - The fold method for a List takes two arguments; the start value and a function to apply on list's elements
- Zip-Unzip
  - Zip is used to aggregate the contents of two lists into a single list of pairs
  - Unzip is used to partition a list

# Scala Functions: zip, find, drop, dropwhile, foldLeft,

Example

```
//zip function: aggregates the contents of two lists into a single list of pairs
List(1, 2, 3).zip(List("a", "b", "c"))          //> res0: List[(Int, String)] = List((1,a), (2,b), (3,c))

//partition function: splits a list based on the condition provided
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                                                //> numbers  : List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
println(numbers.partition(_ % 2 == 0))          //> (List(2, 4, 6, 8, 10),List(1, 3, 5, 7, 9))

//find function:  returns the first element of a collection that matches a predicate function
numbers.find((i: Int) => i > 5)                 //> res1: Option[Int] = Some(6)

//drop function: Drops the first i elements
numbers.drop(5)                                 //> res2: List[Int] = List(6, 7, 8, 9, 10)

//dropwhile function: removes the first elements that match a predicate function
numbers.dropWhile(_ % 2 != 0)                   //> res3: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)

//foldLeft & foldRight
numbers.foldLeft(0)((m: Int, n: Int) => m + n)  //> res4: Int = 55

numbers.foldLeft(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
                                                //> m: 0 n: 1
                                                //| m: 1 n: 2
                                                //| m: 3 n: 3
                                                //| m: 6 n: 4
                                                //| m: 10 n: 5
                                                //| m: 15 n: 6
                                                //| m: 21 n: 7
                                                //| m: 28 n: 8
                                                //| m: 36 n: 9
                                                //| m: 45 n: 10
                                                //| res5: Int = 55
```

# Scala Functions: foldRight, Map, filter

## Example

```
numbers.foldRight(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
                                                                                    //> m: 10 n: 0
                                                                                    //| m: 9 n: 10
                                                                                    //| m: 8 n: 19
                                                                                    //| m: 7 n: 27
                                                                                    //| m: 6 n: 34
                                                                                    //| m: 5 n: 40
                                                                                    //| m: 4 n: 45
                                                                                    //| m: 3 n: 49
                                                                                    //| m: 2 n: 52
                                                                                    //| m: 1 n: 54
                                                                                    //| res6: Int = 55


//Map

val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)
                                                  //> extensions  : scala.collection.immutable.Map[String,Int] = Map(steve -> 100
                                                  //| , bob -> 101, joe -> 201)
//filter the names that have id less than 200:
extensions.filter((namePhone: (String, Int)) => namePhone._2 < 200)
                                                  //> res7: scala.collection.immutable.Map[String,Int] = Map(steve -> 100, bob ->
                                                  //|  101)
// or by using case
extensions.filter({ case (name, extension) => extension < 200 })
                                                  //> res8: scala.collection.immutable.Map[String,Int] = Map(steve -> 100, bob ->
```

# Scala Functions: flatMap

## Example

```
//Map function: applying a function to each element in the list.

val l=List(1,2,3,4,5)                            //> l  : List[Int] = List(1, 2, 3, 4, 5)
l.map(x=>x*2)                                    //> res9: List[Int] = List(2, 4, 6, 8, 10)

//flatten: collapses one level of nested structure
                                                  //> res10: List[Int] = List(1, 2, 3, 4)
List(List(1, 2), List(3, 4)).flatten

//flatMap: a frequently used combinator that combines mapping and flattening.
//flatMap takes a function that works on the nested lists and then concatenates the results back together

val nestedNumbers = List(List(1, 2), List(3, 4))//> nestedNumbers  : List[List[Int]] = List(List(1, 2), List(3, 4))
nestedNumbers.flatMap(x => x.map(_ * 2))         //> res11: List[Int] = List(2, 4, 6, 8)
```

# Scala Functions: foldLeft

## Example

```
object fold {
  println("Exmaple for FoldLeft")                 //> Exmaple for FoldLeft
  class fold(val name: String, val age: Int, val sex: Symbol)
  def apply(name: String, age: Int, sex: Symbol) = new fold(name, age, sex)
                                                  //> apply: (name: String, age: Int, sex: Symbol)fold.fold
  val NameList = apply("James Jass", 25, 'male) :: apply("John Dickus", 30, 'female) :: Nil
                                                  //> NameList  : List[fold.fold] = List(fold$$anonfun$main$1$fold$1@59a6e353, fol
                                                  //| d$$anonfun$main$1$fold$1@7a0ac6e3)
  val stringList = NameList.foldLeft(List[String]()) { (z, f) =>

    val title = f.sex match {
      case 'male   => "Mr."
      case 'female => "Ms."
    }

    z :+ s"$title ${f.name} ,${f.age}"
  }                                               //> stringList  : List[String] = List(Mr. James Jass ,25, Ms. John Dickus ,30)
  println(stringList(0) + "\n" + stringList(1))   //> Mr. James Jass ,25
                                                  //| Ms. John Dickus ,30
}
```

## Example

```
//finding last item in an array:

val ns=Array(1,2,3,4,5,6)                         //> ns  : Array[Int] = Array(1, 2, 3, 4, 5, 6)
```

# Scala Functions: Map, flatMap

## Example

```
object functions {
//Map and FlatMap

//Map works by applying a function to each element in the list

    val l=List(1,2,3,4,5)                         //> l  : List[Int] = List(1, 2, 3, 4, 5)
    l.map(x=>x*2)                                 //> res0: List[Int] = List(2, 4, 6, 8, 10)


//If you want to return a sequence or a list from the function, for example an option:

def f(x:Int)=if (x>2) Some (x) else None          //> f: (x: Int)Option[Int]
l.map(x=>f(x))                                    //> res1: List[Option[Int]] = List(None, None, Some(3), Some(4), Some(5))
l.flatMap(x=>f(x))                                //> res2: List[Int] = List(3, 4, 5)


//Map can be thought of a sequence of tuples, with a key and a value

val m=Map(1->2,2->4,3->6)                         //> m  : scala.collection.immutable.Map[Int,Int] = Map(1 -> 2, 2 -> 4, 3 -> 6)


//so if we want to use map and flatmap on our map, we should apply the function to the values

m.mapValues(v=>v*2)                               //> res3: scala.collection.immutable.Map[Int,Int] = Map(1 -> 4, 2 -> 8, 3 -> 12)
                                                  //|
m.mapValues(v=>f(v))                              //> res4: scala.collection.immutable.Map[Int,Option[Int]] = Map(1 -> None, 2 ->
                                                  //| Some(4), 3 -> Some(6))


//flamtmap doesn't work the same as mapValues on the maps:
m.flatMap(e=>List(e._2))                          //> res5: scala.collection.immutable.Iterable[Int] = List(2, 4, 6)
```

# Scala Functions: flatMap, filter

## Example

```
//Function f is defined to filter out the None's :

def h(k:Int,v:Int) =if (v>2) Some(k->v) else None //> h: (k: Int, v: Int)Option[(Int, Int)]

//and we can call it as follows:

m.flatMap(e =>h(e._1,e._2))                    //> res6: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)

//if we want to get rid of _1,_2,... we can write:

m.flatMap{ case (k,v) => h(k,v) }              //> res7: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)


// filter: There is an alternative to do the same task with filter method
                                               //> res8: List[Int] = List(1, 2, 3, 4, 5)
m.filter(e=> f(e._2)!=None)                    //> res9: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)

m.filter {case(k,v) =>f(v)!=None}              //> res10: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)
m.filter {case(k,v) => f(v).isDefined}         //> res11: scala.collection.immutable.Map[Int,Int] = Map(2 -> 4, 3 -> 6)
}
```

# For-Expression

- Sometimes the level of abstraction required by some of the higher-order functions in Scala makes programs a bit hard to understand

- Alternative solution is using For Expressions:

  - Querying with For Expression
  - Translation of For Expressions to map, flatMap and filter
  - Translation of map, flatMap and filter to For Expression

# For-Expression: Querying

## Example

```
case class Book(title: String, authors: String*)
  val books: List[Book] =
  List(
  Book(
    "Structure and Interpretation of Computer Programs",
    "Abelson, Harold", "Sussman, Gerald J."),
  Book(
    "Principles of Compiler Design",
    "Aho, Alfred", "Ullman, Jeffrey"),
  Book(
    "Programming in Modula-2",
    "Wirth, Niklaus"),
  Book(
    "Elements of ML Programming",
    "Ullman, Jeffrey"),
  Book(
    "The Java Language Specification", "Gosling, James",
    "Joy, Bill", "Steele, Guy", "Bracha, Gilad"))
```

```
//to find the titles of all books whose author's last name is "Gosling":
for (b <- books; a <- b.authors if a startsWith "Gosling")
  yield b.title

//to find the titles of all books that have the string "Program" in their title:
for (b <- books if (b.title indexOf "Program") >= 0)
  yield b.title

// to find the names of all authors that have written at least two books in the database:
for (
  b1 <- books; b2 <- books if b1 != b2;
  a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
  yield a1
```

# For-Expression: Translation

- map
  - for (x <- expr_1) yield expr_2
  - expr_1.map(x => expr_2)
- flatMap
  - for (x <- expr_1; y <- expr_2; seq) yield expr_3
  - expr_1.flatMap(x => for (y <- expr_2; seq) yield expr_3)
- filter
  - for (x <- expr_1 if expr_2) yield expr_3
  - for (x <- expr_1 filter (x => expr_2)) yield expr_3

# For-Expression: Translation

## Example

```
case class per(name: String, isMale: Boolean, children: per*)
val Laura = per("Laura", false)
val Julie = per("Julie", false)
val Bob = per("Bob", true, Laura, Julie)
val pers = List(Laura, Bob, Julie)


//to find out the names of all pairs of father and their children in that list

  for (p <- pers; if p.isMale; c <- p.children)
       yield (p.name, c.name)                    //> res0: List[(String, String)] = List((Bob,Laura), (Bob,Julie))

//Using map, flatMap and filter it can be formulated by the following query:

  pers filter (p => p.isMale) flatMap (p =>
       (p.children map (c => (p.name, c.name))))//> res1: List[(String, String)] = List((Bob,Laura), (Bob,Julie))
```

# For-Expression: Translation

## Example

```
case class Book(title: String, authors: String*)
val books: List[Book] =
  List(
    Book(
      "Structure and Interpretation of Computer Programs",
      "Abelson, Harold", "Sussman, Gerald J."),
    Book(
      "Principles of Compiler Design",
      "Aho, Alfred", "Ullman, Jeffrey"),
    Book(
      "Programming in Modula-2",
      "Wirth, Niklaus"),
    Book(
      "Elements of ML Programming",
      "Ullman, Jeffrey"),
    Book(
      "The Java Language Specification", "Gosling, James",
      "Joy, Bill", "Steele, Guy", "Bracha, Gilad"))

  //to find all authors who have published at least two books
    for (b1 <- books; b2 <- books if b1 != b2;
         a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
         yield a1 //> res0: List[String] = List(Ullman, Jeffrey, Ullman, Jeffrey)

  //This query translates to the following map/flatMap/filter combination:
  books flatMap (b1 =>
  books filter (b2 => b1 != b2) flatMap (b2 =>
    b1.authors flatMap (a1 =>
      b2.authors filter (a2 => a1 == a2) map (a2 =>
```

# For-Expression: Translation

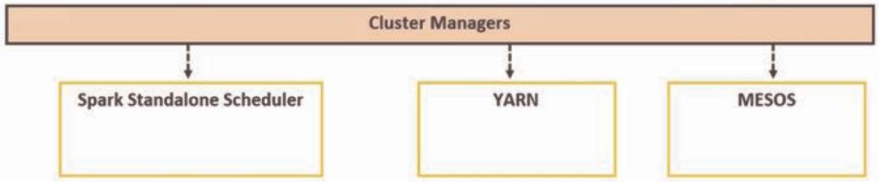
- Example of an application that translates from map, flatMap and filter to For-Expressions:

```
def map[A, B](xs: List[A], f: A => B): List[B] =
                                        for (x <- xs) yield f(x)

def flatMap[A, B](xs: List[A], f: A => List[B]): List[B] =
                                        for (x <- xs; y <- f(x)) yield y

def filter[A](xs: List[A], p: A => Boolean): List[A] =
                                        for (x <- xs if p(x)) yield x
```

# Spark Internals & Architecture ...



## Cluster Managers




Cluster Managers

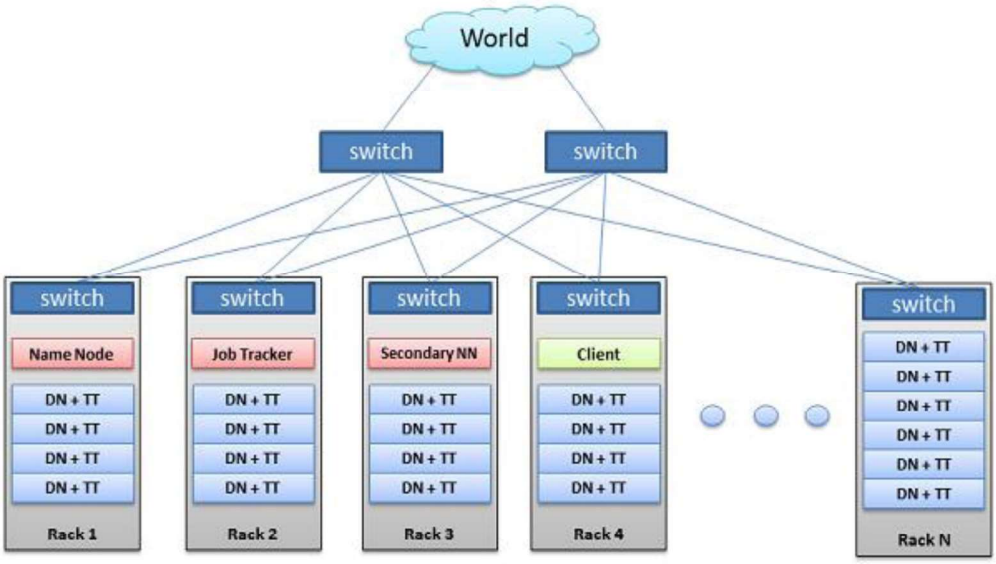
- Spark provides efficient and effective scaling
  - Can operate on one to several thousand nodes
  - Standalone Scheduler cluster manager included with Spark
  - YARN and Mesos cluster managers used to deploy Spark on a shared cluster

# What is YARN?

- Yet Another Resource Negotiator
- Part of core Hadoop
- A multi-node cluster has an aggregate pool of compute resources
- CPU and memory
- Sounds like resources that an operating system would manage
  - AKA the Hadoop Operating System
- Recall that disk resources are managed by HDFS
- Now we have total infrastructure resource management by Hadoop!

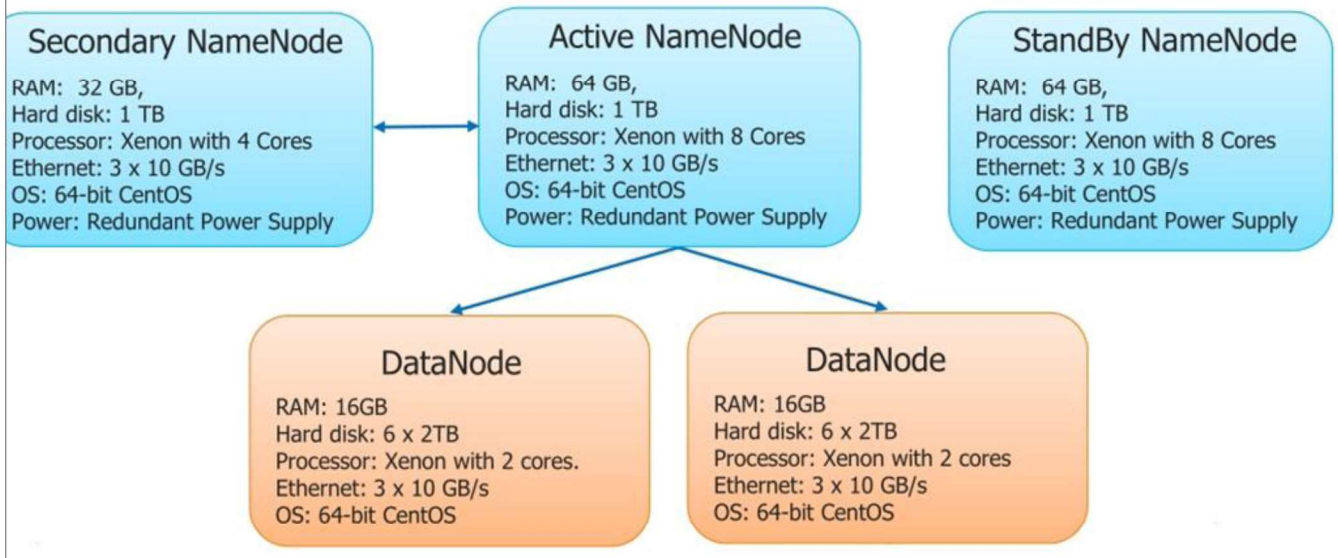
# Understanding Hadoop Cluster

## Hadoop Cluster

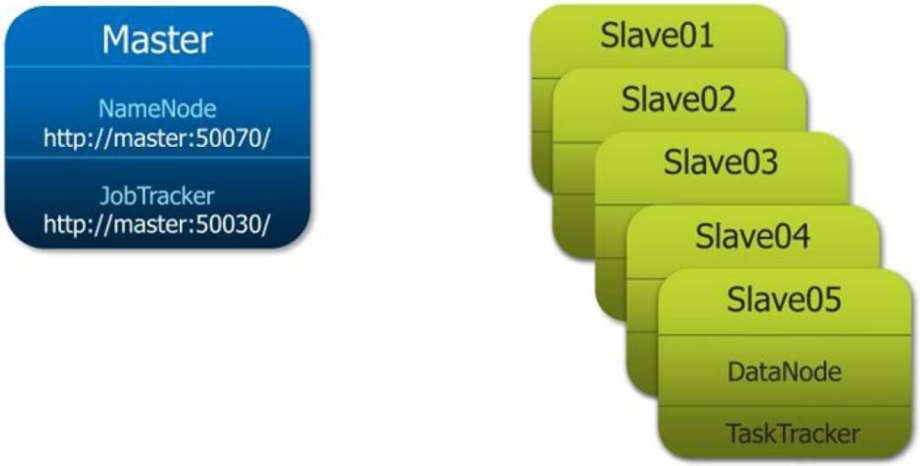



This is the typical architecture of a Hadoop cluster

## Hadoop Cluster: A Typical Use Case




## Hadoop 1.0: Cluster Architecture



# Main Components of HDFS

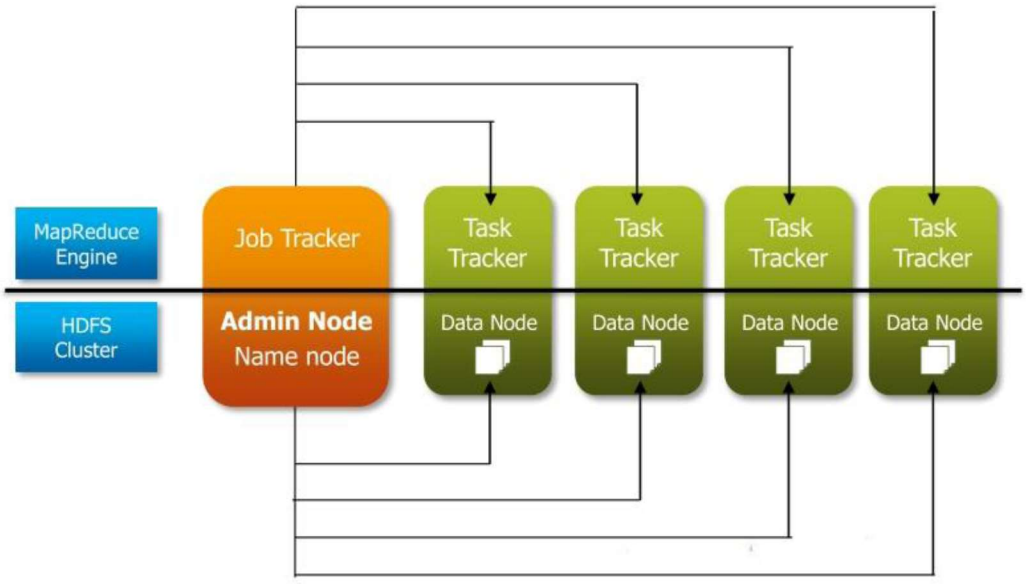
→ NameNode:

» Master of the system
» Maintains and manages the blocks which are present on the DataNodes

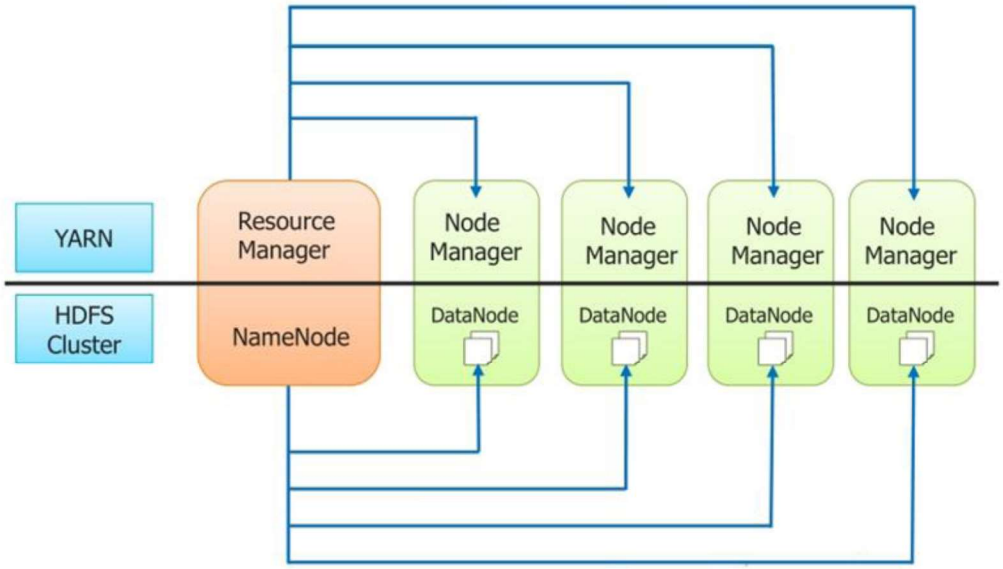
→ DataNodes:

» Slaves which are deployed on each machine and provide the actual storage

# Hadoop Core Components



# Hadoop 2.x Core Components ( Contd.)




# RDD (Resilient Distributed Datasets)

→ Resilient Distributed Datasets (RDDs), a distributed memory abstraction which lets programmers perform in-memory computations on large clusters in a fault-tolerant manner

→ RDDs can be created from any data source eg. Scala collection, local file system, Hadoop, Amazon S3, HBase table etc.

→ Spark supports text files, SequenceFiles, and any other Hadoop InputFormat, and can also take a directory or a glob (e.g. /ip/2014*)

→ Even though the RDDs are defined, they don't contain any data

→ The computation to create the data in a RDD is only done when the data is referenced; e.g. caching results or writing out the RDD

→ It is a very important feature for development process, as code can be written, compiled and even run on development environment without actually loading the original data!

# Data Loading in RDD

→ RDD is re-computed every time when it is materialized

→ So it is a good idea to improve performance by caching a RDD if it is accessed frequently

→ One of the easiest way to load data in RDD is to load from a Scala collection:

```
val dataRDD = sc.parallelize(List(1,2,4))
```

→ SparkContext provides parallelize function, which converts the Scala collection into the RDD of the same type:

```
dataRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12
```

# Data Loading in RDD (Contd.)

→ Another simple way of loading data is loading text from a file

→ In local mode (single node operation), you just need to specify the file location

→ In distributed mode, the pre-requisite is the availability of the file in all nodes of the cluster

→ Spark's addFile functionality is used to copy the file to designated location to all machines in the cluster

```
import spark.SparkFiles;
                ...
                sc.addFile("abc.dat")
                val inFile = sc.textFile(SparkFiles.get("abc.dat"))
```

# Manipulating RDDs

→ Manipulating RDDs is quite similar to Scala collections manipulation

→ Most of the standard list functions are available directly on Spark RDDs

→ Programmer need not to worry about the RDDs being executed on same machine or multiple machines

→ The hallmark functions of map and reduce are available by default

→ The map and other Spark functions DO NOT transform the existing elements, they always create a new RDD with transformed elements (Immutability in Action!)

» Map is a higher order function, which takes in a function, applies to each individual element of RDD and produces a new RDD

» Reduce also takes in a function, which operates in pairs to combine all the data. The reducer function needs to be commutative and associative, i.e.

f(a,b) == f(b,a) and f(a, f(b,c)) == f(f(a,b), c)

# Scala RDD Functions

| Function | Parameter options | Explanation | Return Type |
|---|---|---|---|
| foldByKey | (zeroValue)<br>(func(V,V)=>V) | Merges the values using the provided function. Unlike a traditional fold function over a list, the zeroValue can be added an arbitrary number of times. | RDD[K,V] |
| reduceByKey | (func(V,V)=>V, numTasks) | Parallel version of reduce that merges the values for each key using the provided function and returns an RDD. | RDD[K,V] |
| groupByKey | (numPartitions) | Groups elements together by key. | RDD[K,Seq[V]] |

# Pair RDD Functions

| Function | Parameter options | Explanation | Return Type |
|---|---|---|---|
| lookup | (key: K) | Looks up a specific element in the RDD. Uses the RDD's partitioner to figure out which partition(s) to look at. | Seq[V] |
| mapValues | (f: v=>u) | A specialized version of map for PairRDD when you only want to change the value of the key-value pair. If you need to make your changes based on both key and value, you must use one of the normal RDD Map functions. | RDD[K,U] |
| collectAsMap | () | Takes an RDD and returns a concrete map. Your RDD must be able to fit into the memory. | Map[K, V] |

# Pair RDD Functions (Contd.)

| Function | Parameter options | Explanation | Return Type |
|---|---|---|---|
| countByKey | () | Counts the number of elements for each key in RDD. | Map[K, Long] |
| partitionBy | (partitioner:P, mapSideCombine: Boolean) | Returns a new RDD with the same data but partitioned by the new Partitioner, and mapSideCombine controls Spark group values with the same key together before repartitioning. Defaults to false. | RDD[K,V] |
| flatMapValues | (f:V => TraversableOnce[U]) | Similar to mapValues. A specialized version of flatMap for PairRDDs when you only want to change the value of the key-value pair. Takes the provided Map function and applies it to the value. The resulting sequence is then "flattened". | RDD[K, U] |

# Double RDD Functions

→Spark provides a number of Utility functions for the RDDs if they consist of double data type

| Function | Argument | Returns |
|---|---|---|
| mean | () | Average of RDDs elements |
| Sum | () | Sum of Elements |
| variance | () | Variance of RDD elements |
| Stats | () | Mean, Variance and Count as Stats Counter |

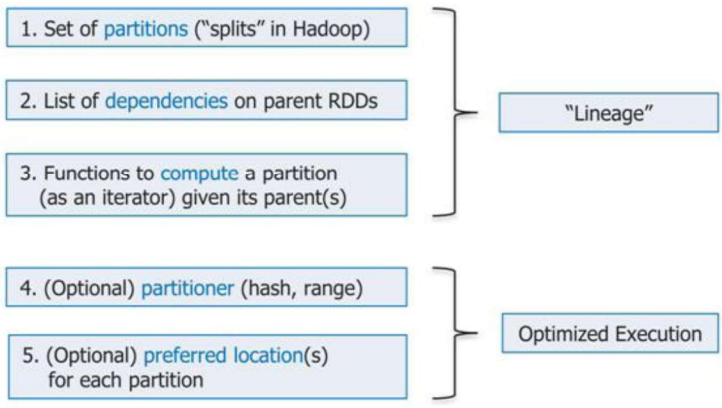
# General RDD Functions

| Function | Argument | Returns |
|---|---|---|
| cache | () | Caches an RDD reused without re-computing. Same as persist(StorageLevel. MEMORY_ONLY). |
| collect | () | An array of element in RDD. |
| count | () | Number of elements in RDD. |
| countByValue | () | A map of value to the number of times the value occurs. |
| distinct | () | RDD containing only distinct elements. |
| filter | (f: T => Boolean) | RDD containing elements only matching f. |
| foreach | (f: T => Unit) | Applies the function f to each element. |
| persist | () ,(newLevel: StorageLevel) | Sets the RDD storage level, which can cause the RDD to be stored after it is computed. Different StorageLevels can be seen in StorageLevel. |

# General RDD Functions

| Function | Argument | Returns |
|---|---|---|
| sample | (fraction: double) | RDD of that fraction. |
| toDebugString | () | A handy function that outputs the recursive deps of the RDD. |
| count | () | Number of elements in RDD. |
| unpersist | () | Remove all the persistent blocks of the RDD from the memory/disk. |
| union | (other: RDD[T]) | An RDD containing elements of both RDDs. Duplicates are not removed. |

# RDD Deep Dive: What is it?



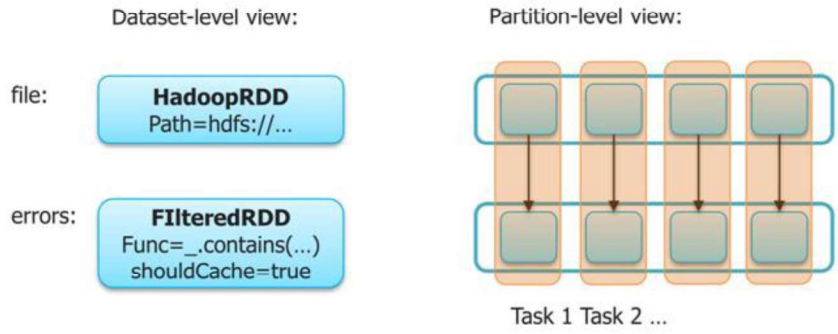
# HadoopRDD

| Example: HadoopRDD | | |
|---|---|---|
| partition | = | One per HDFS block |
| dependencies | = | none |
| compute (part) | = | read corresponding block |
| preferred location(part) | = | HDFS block location |
| partitioner | = | none |

# FilteredRDD

| Example: FilteredRDD | | |
|---|---|---|
| partition | = | Same as parent RDD |
| dependencies | = | "one-to-one" on parent |
| compute (part) | = | Compute parent and filter it |
| preferred location(part) | = | None (ask parent) |
| partitioner | = | none |

# RDD Graph




# JoinedRDD

| | | |
|---|---|---|
| Example: JoinedRDD | | |
| partition | = | one per reduce task |
| dependencies | = | "shuffle" on each parent |
| compute (partition) | = | read and join shuffle data |
| preferred location(part) | = | none |
| partitioner | = | HashPartitioner(numTasks) |

Spark will now know this data is hashed!

# Transformations

→ Transformations create a new dataset from an existing one

→ All transformations in Spark are lazy: they do not compute their results right away

→ instead they remember the transformations applied to some base dataset

→ This helps in:

» Optimizing the required calculations

» Recover from lost data partitions

# Transformations (Contd.)

| Transformation | Meaning |
|---|---|
| map(func) | Return a new distributed dataset formed by passing each element of the source through a function func |
| filter(func) | Return a new dataset formed by selecting those elements of the source on which func returns true |
| intersection(otherDataset) | Return a new RDD that contains the intersection of elements in the source dataset and the argument |
| groupByKey([numTasks]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs |
| reduceByKey(func, [numTasks]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V |

# Transformations (Contd.)

| Transformation | Meaning |
|---|---|
| join(otherDataset, [numTasks]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key |
| cartesian(otherDataset) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements) |
| intersection(otherDataset) | Return a new RDD that contains the intersection of elements in the source dataset and the argument |
| coalesce(numPartitions) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset |
| cogroup(otherDataset, [numTasks]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples |

# Actions

→Spark forces the calculations for execution only when actions are invoked on the RDDs

| Action | Meaning |
|---|---|
| reduce(func) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel |
| first() | Return the first element of the dataset (similar to take(1)) |
| takeOrdered(n, [ordering]) | Return the first *n* elements of the RDD using either their natural order or a custom comparator |
| count() | Return the number of elements in the dataset |

## Actions (Contd.)

| Action | Meaning |
|---|---|
| Collect() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data |
| saveAsSequenceFile(path) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local file system, HDFS or any other Hadoop-supported file system |
| foreach(func) | Run a function func on each element of the dataset. This is usually done for side effects such as updating an accumulator variable |
| countByKey() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key |

## A Day in the Life of a Spark Application

1. Determine RDD lineage: construct DAG
2. Create execution plan for DAG: determine stages
3. Schedule and execute tasks

```
file_rdd.map(line => line.split(""))
      .map(split => (split(0), split(1).toInt))
      .groupByKey()
      .mapValues(iter => iter.reduce(_ + _)).collect()
```

**dummy.txt**

```
jon 2
mary 3
anna 1
jon 1
jesse 3
mary 5
```

# What's in a Task?

```
sc.textFile('file:///dummy.txt')
```

↓

```
map(line => line.split(" "))
```

↓

```
map(split => (split(0), split(1).toInt))
```

↓

```
groupByKey()
```

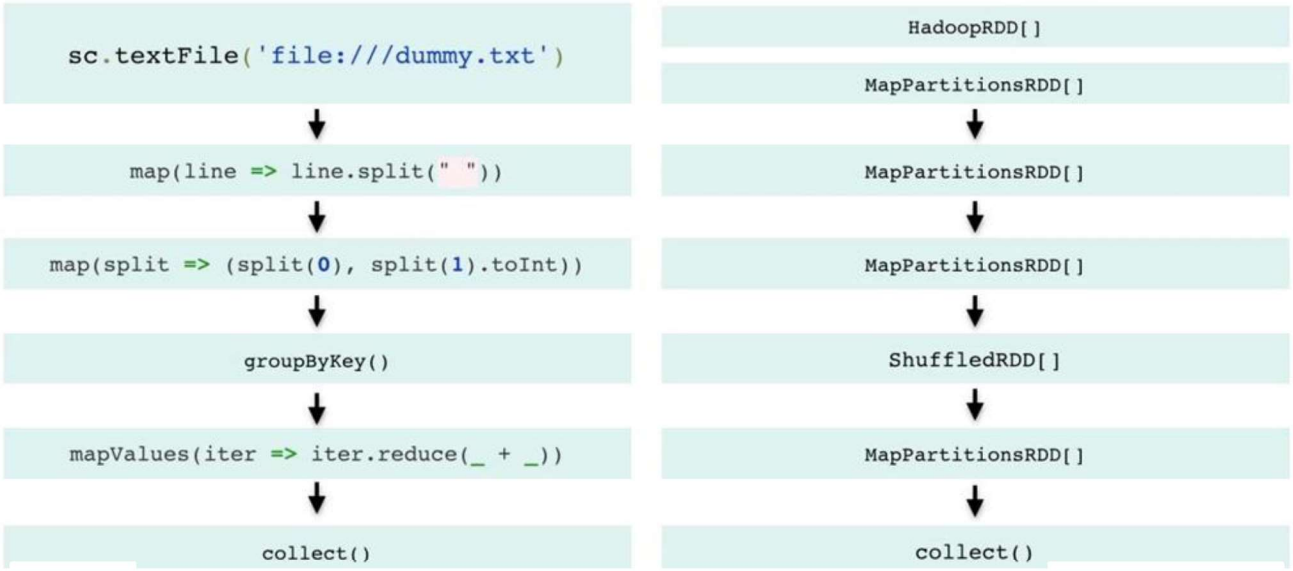
↓

```
mapValues(iter => iter.reduce(_ + _))
```

↓

```
collect()
```

# 1. Create RDDs



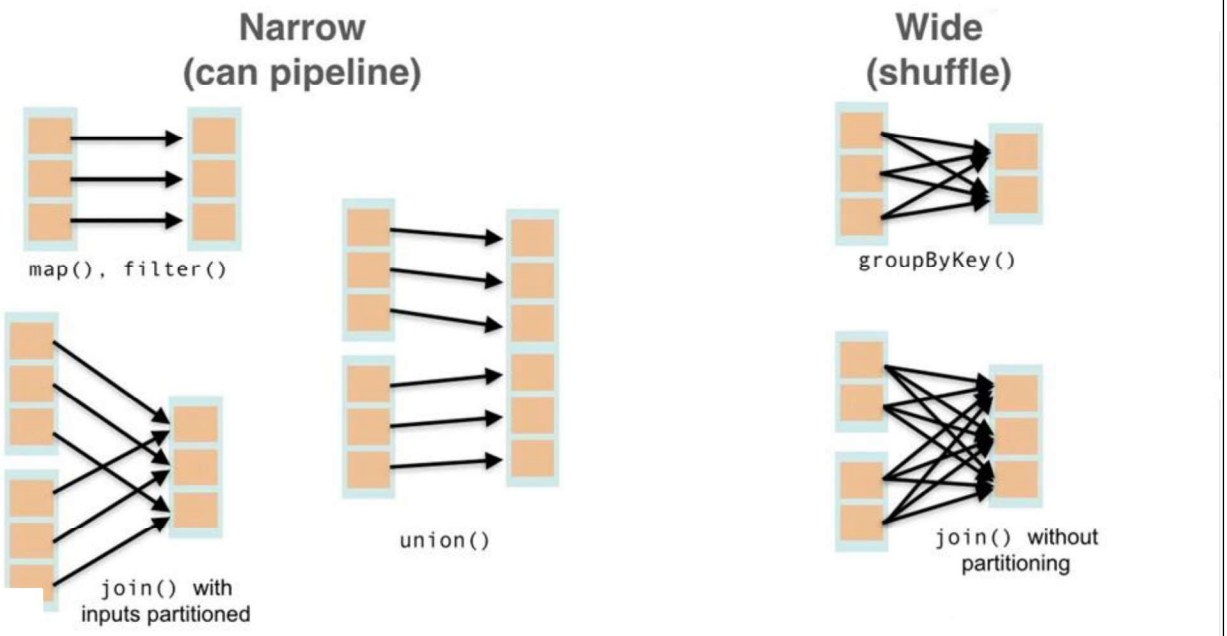

# RDD as an interface

| *Operation* | *Meaning* |
|---|---|
| `partitions()` | Return a list of `Partition` objects |
| `dependencies()` | Return a list of dependencies |
| `compute(p, parent)` | Compute the elements of `Partition` ***p*** given its ***parent*** `Partitions` |
| `partitioner()` | Return metadata specifying whether this RDD is hash/range partitioned |
| `preferredLocations(p)` | List nodes where `Partition` ***p*** can be accessed quicker due to data locality |

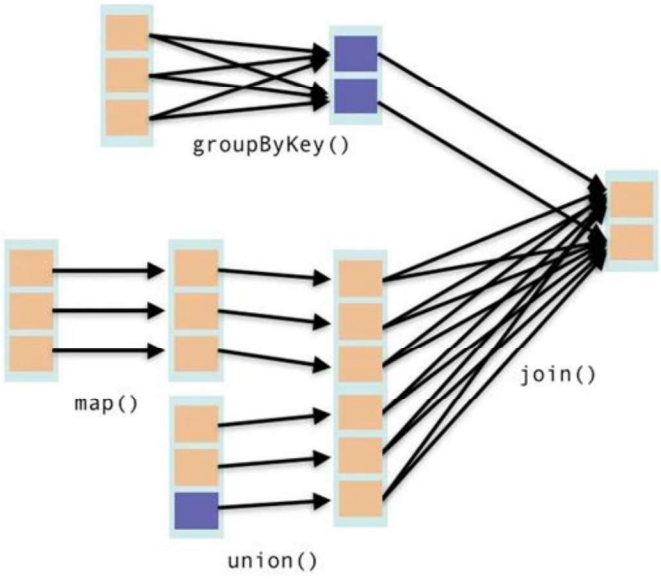
# Functions Revisited

```
import random
flips = 1000000

# local sequence
coins = xrange(flips)

# distributed sequence
coin_rdd = sc.parallelize(coins)
flips_rdd = coin_rdd.map(lambda i: random.random())
heads_rdd = flips_rdd.filter(lambda r: r < 0.51)

# local value
head_count = heads_rdd.count()
```

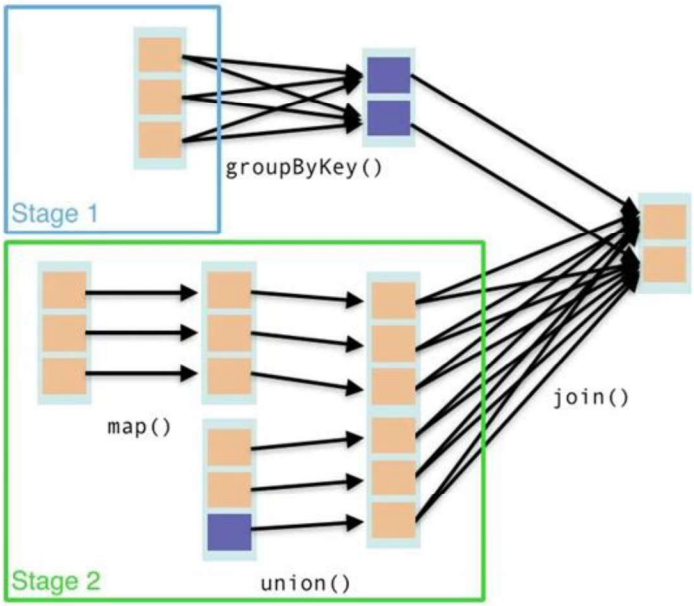
# Partition Dependencies



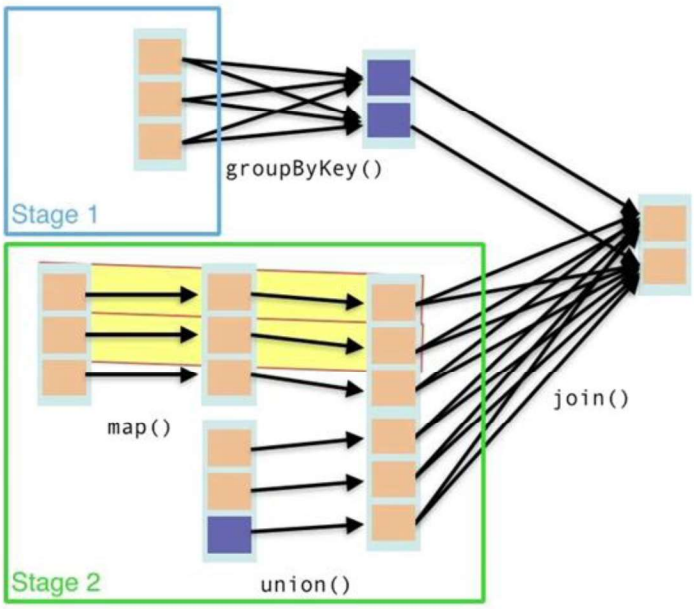
# Partition Dependencies



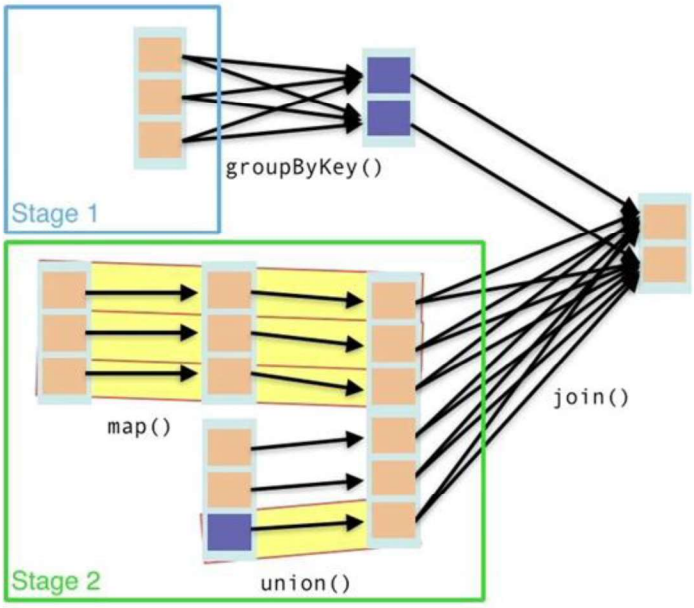

# Partition Dependencies



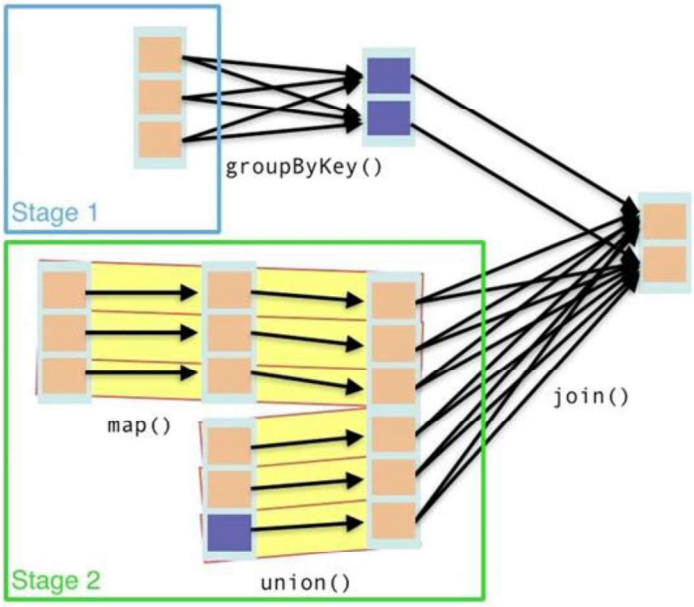
# Partition Dependencies



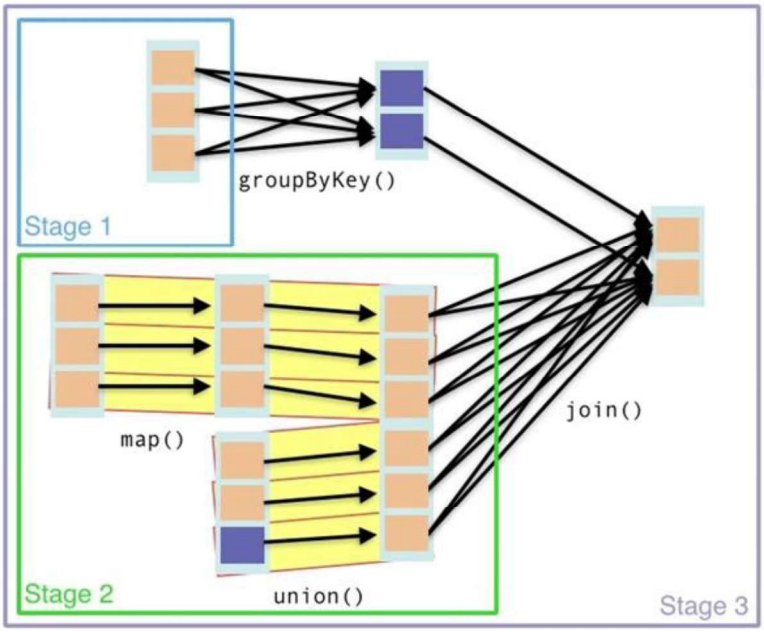

# Partition Dependencies

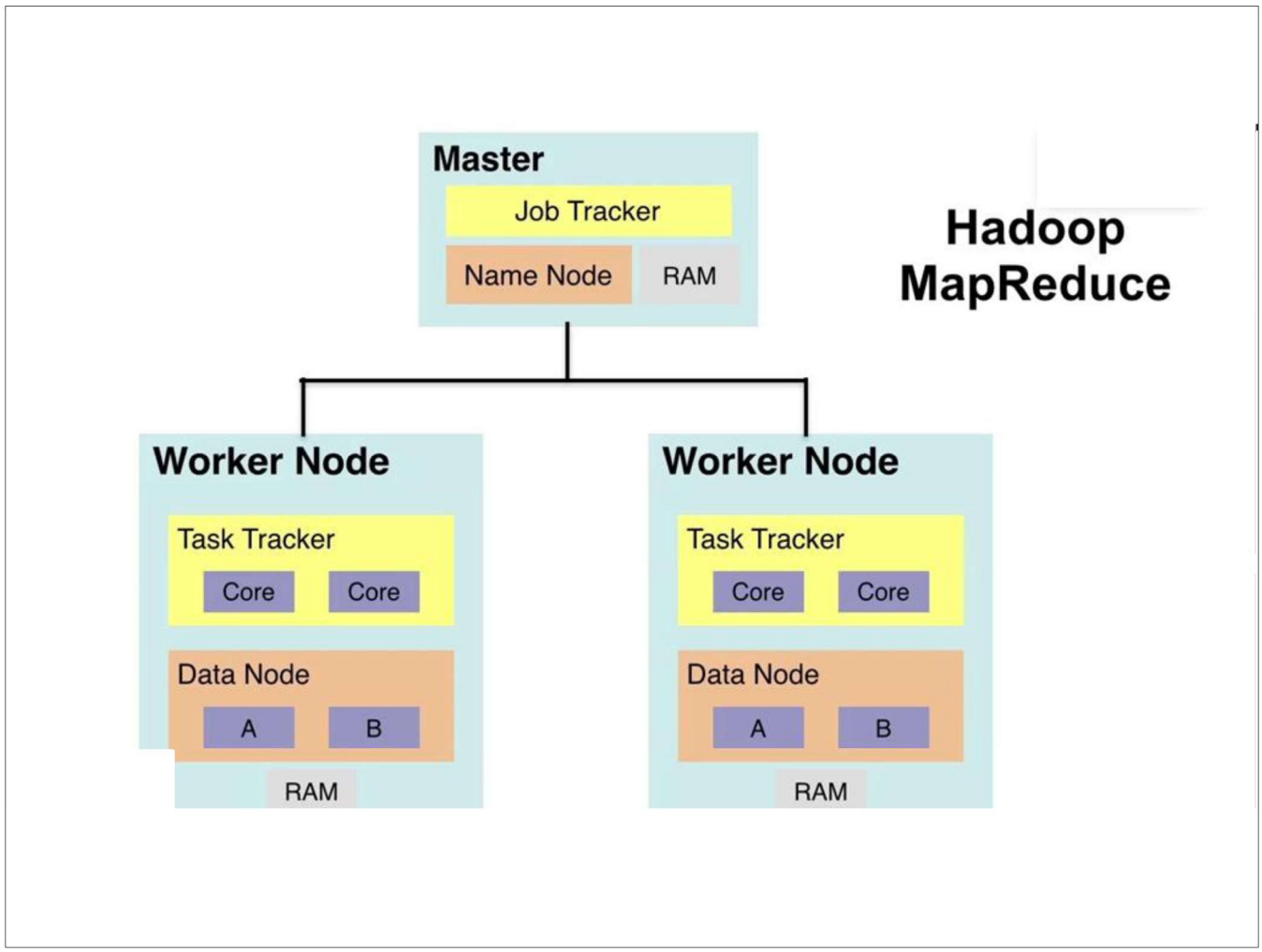


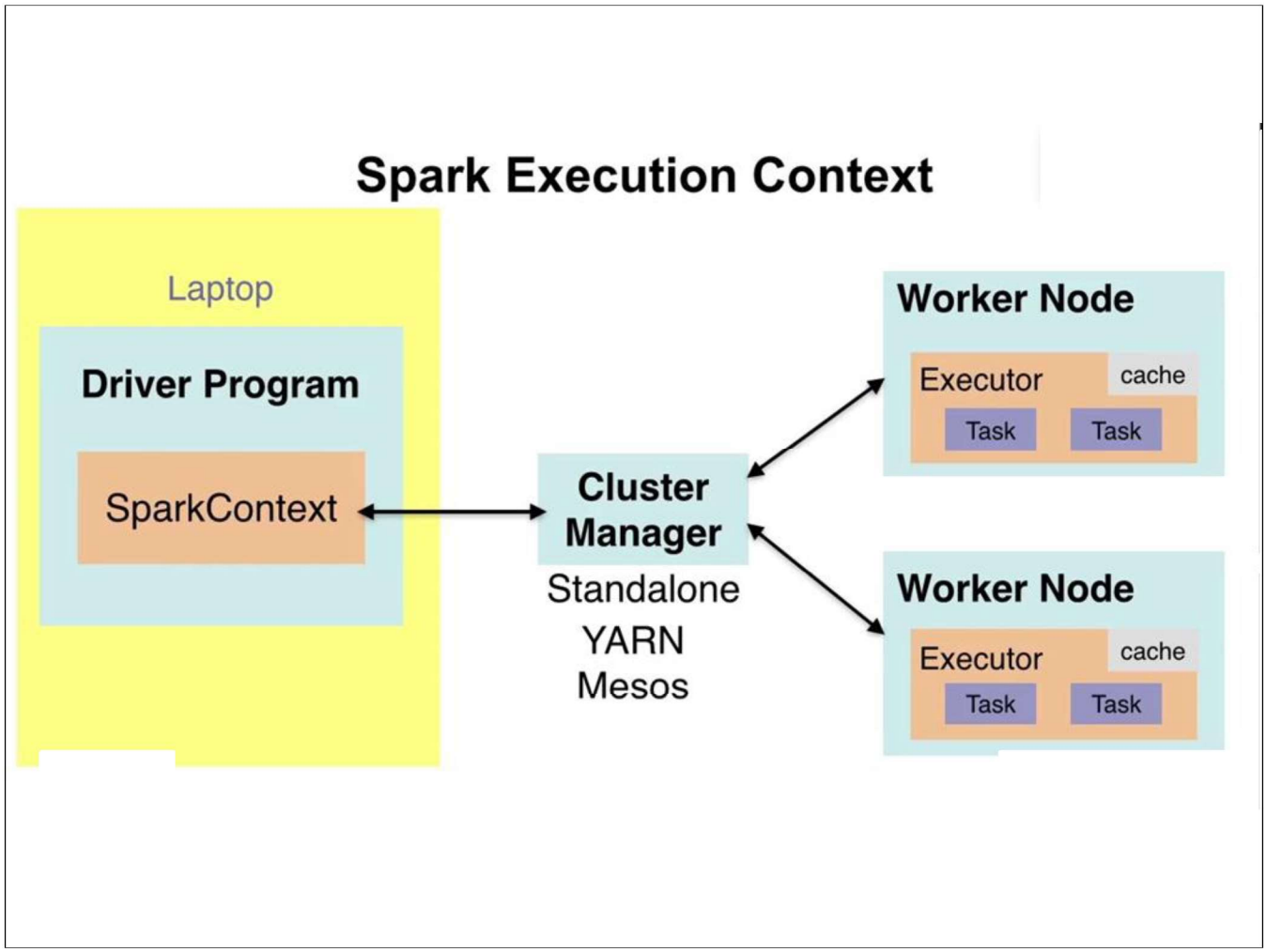
# Partition Dependencies




# Partition Dependencies




Master
Job Tracker
Name Node
RAM
Hadoop
MapReduce
Worker Node
Task Tracker
Core
Core
Data Node
A
B
RAM
Worker Node
Task Tracker
Core
Core
Data Node
A
B
RAM


Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster
Manager
Standalone
YARN
Mesos
Worker Node
Executor
cache
Task
Task
Worker Node
Executor
cache
Task
Task

## Spark Execution Context

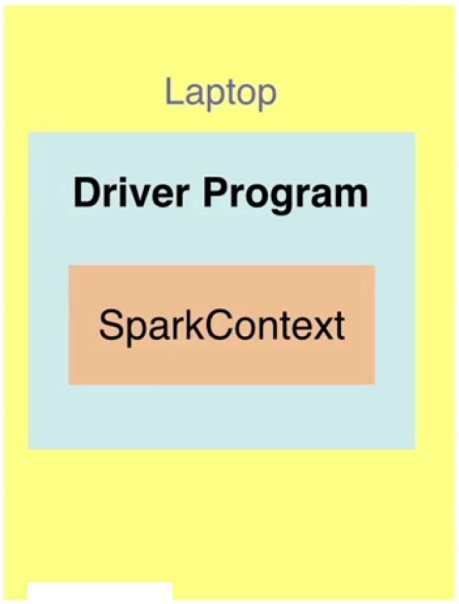
Driver Program

## Spark Execution Context

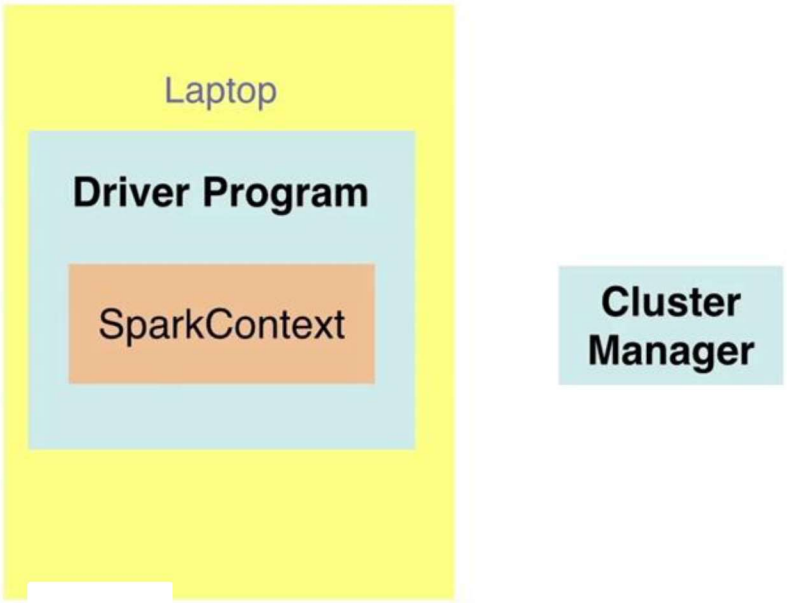
Driver Program

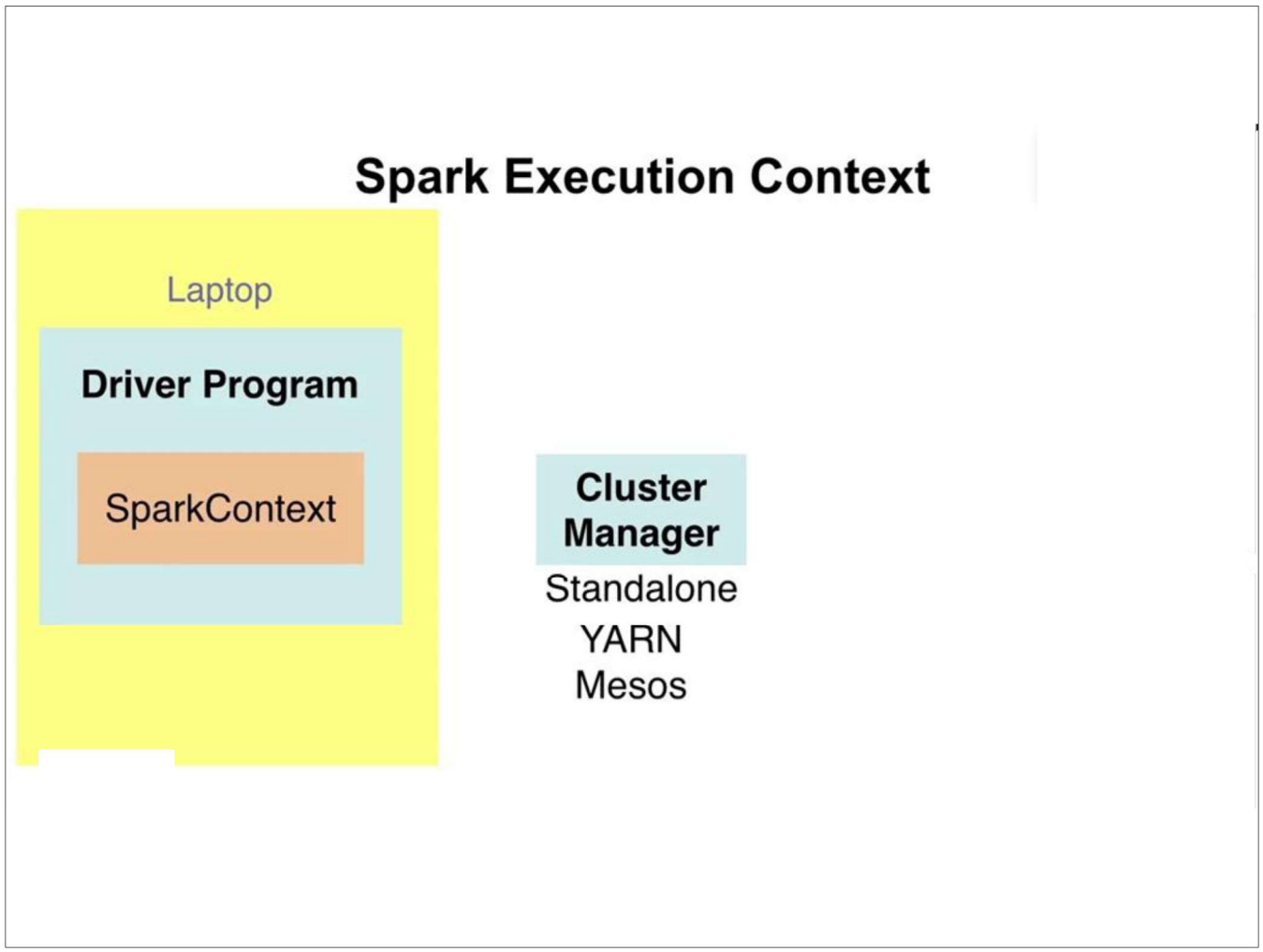
SparkContext

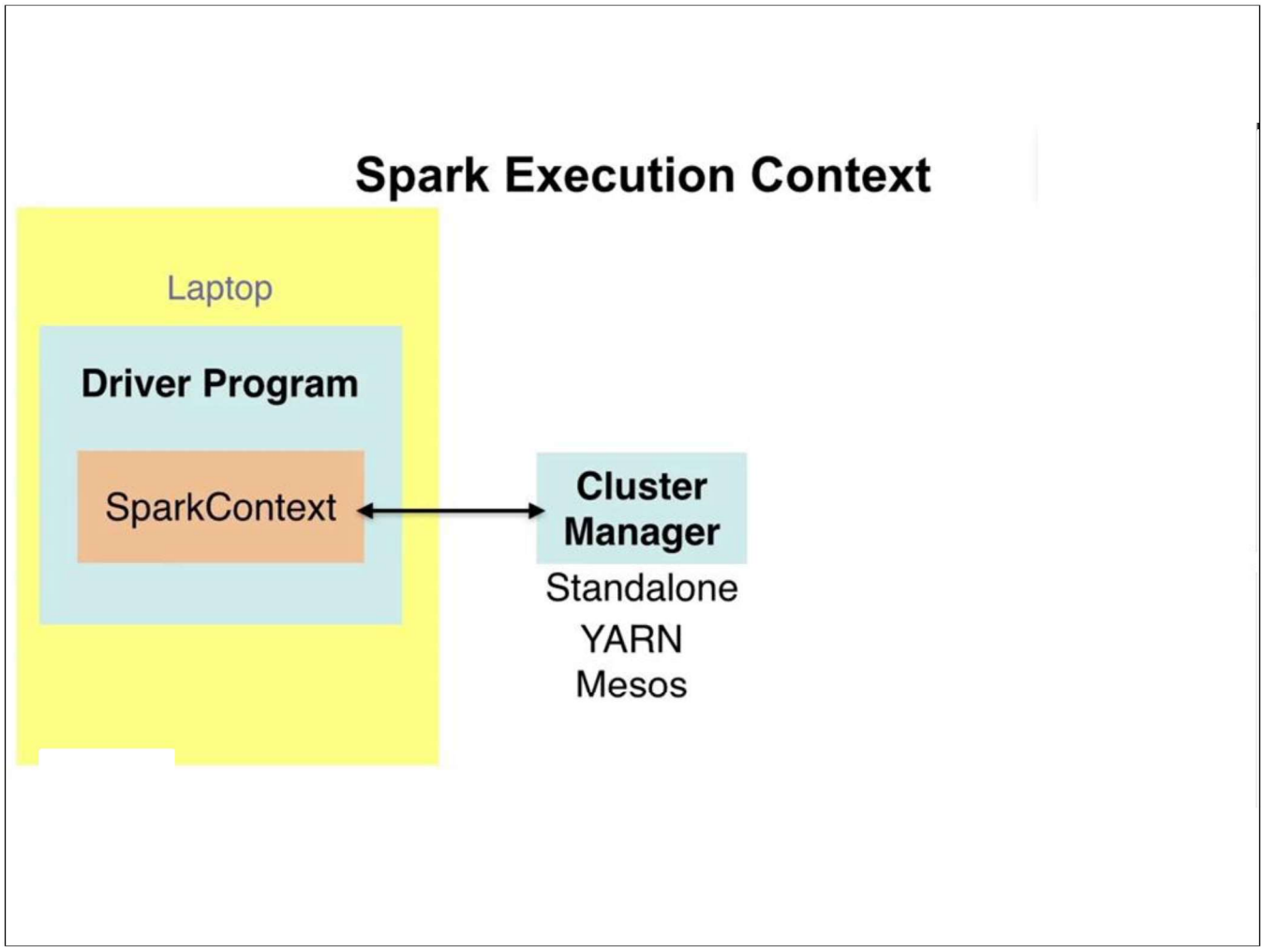
## Spark Execution Context

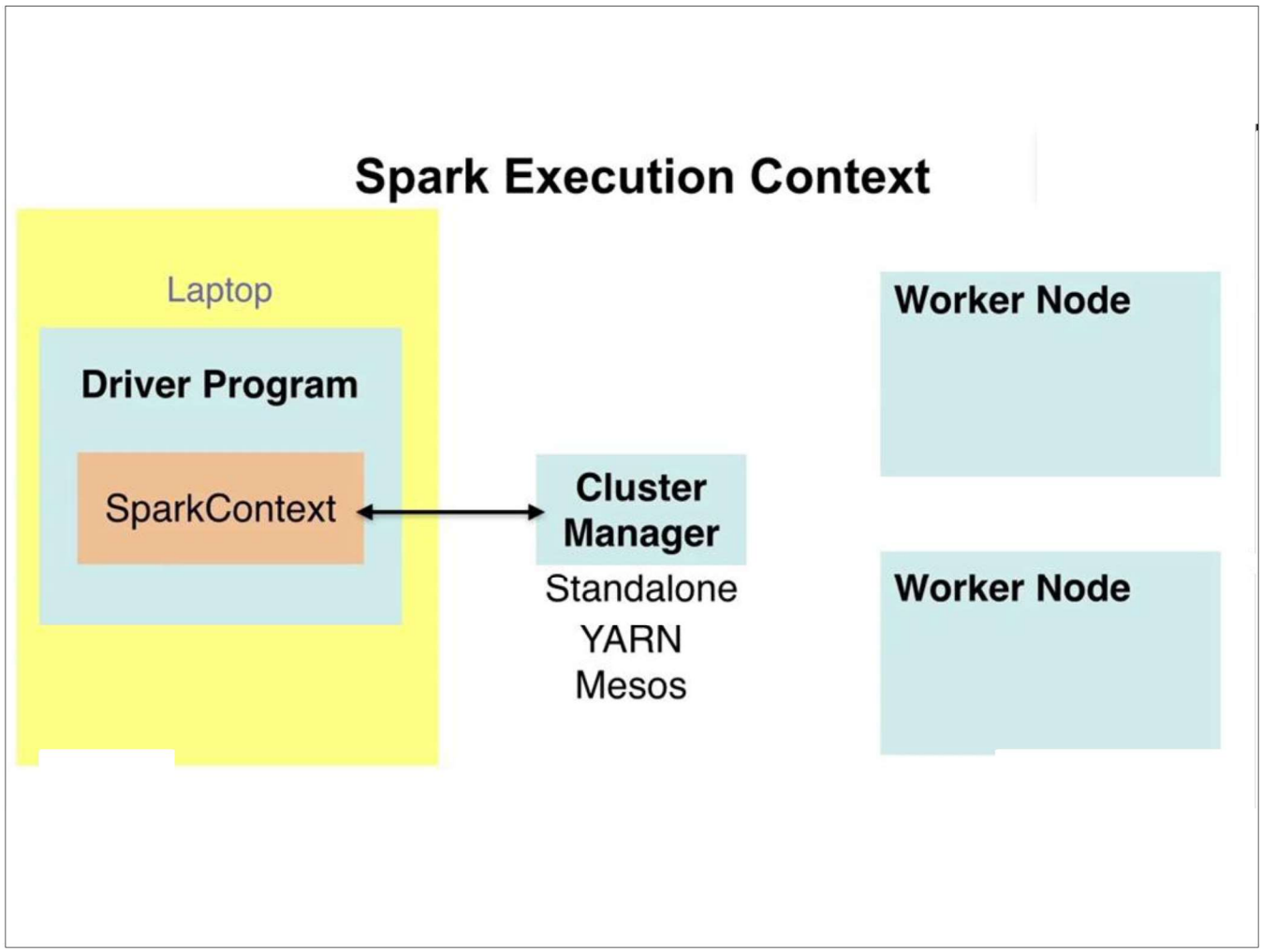


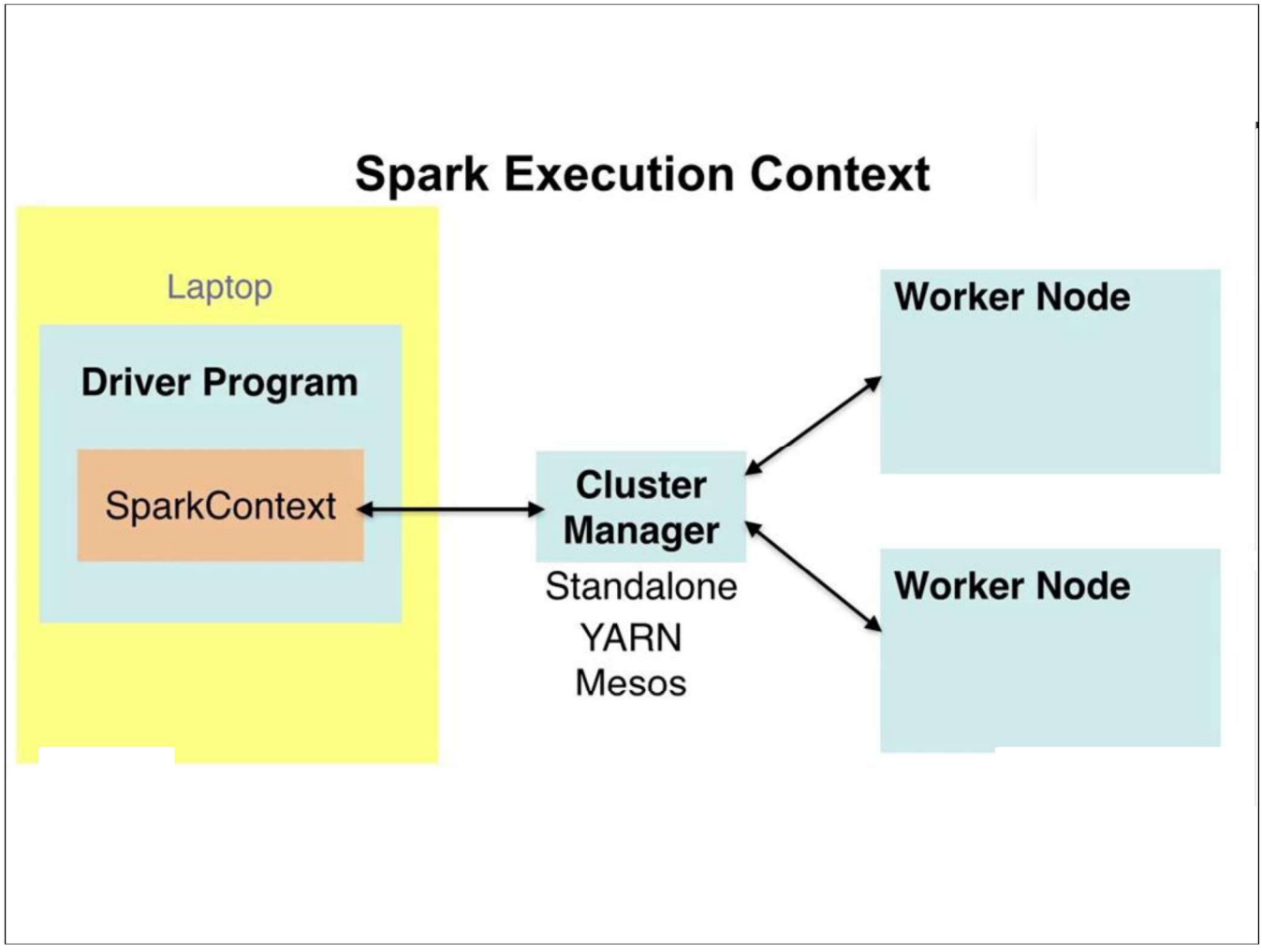

## Spark Execution Context

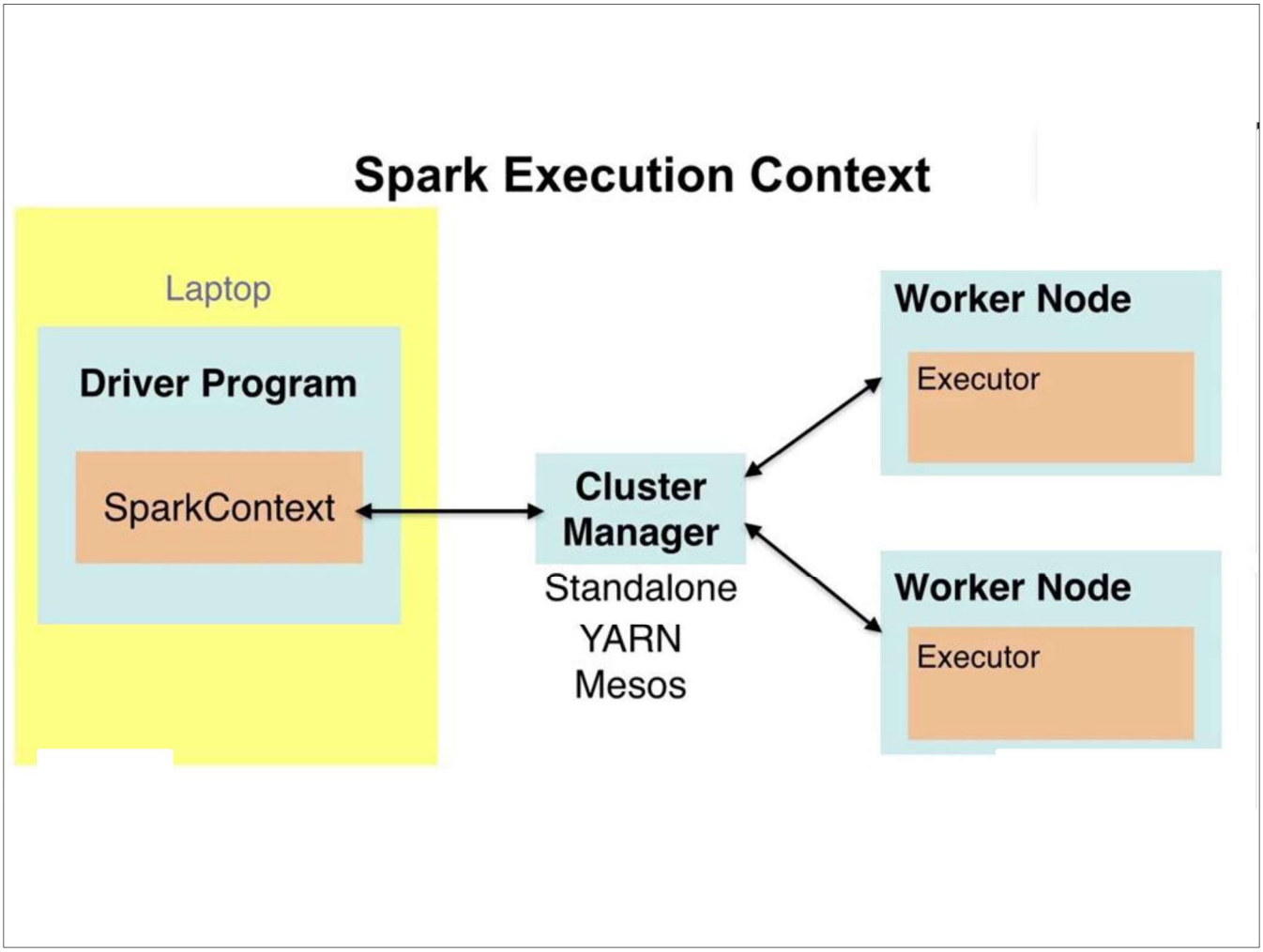


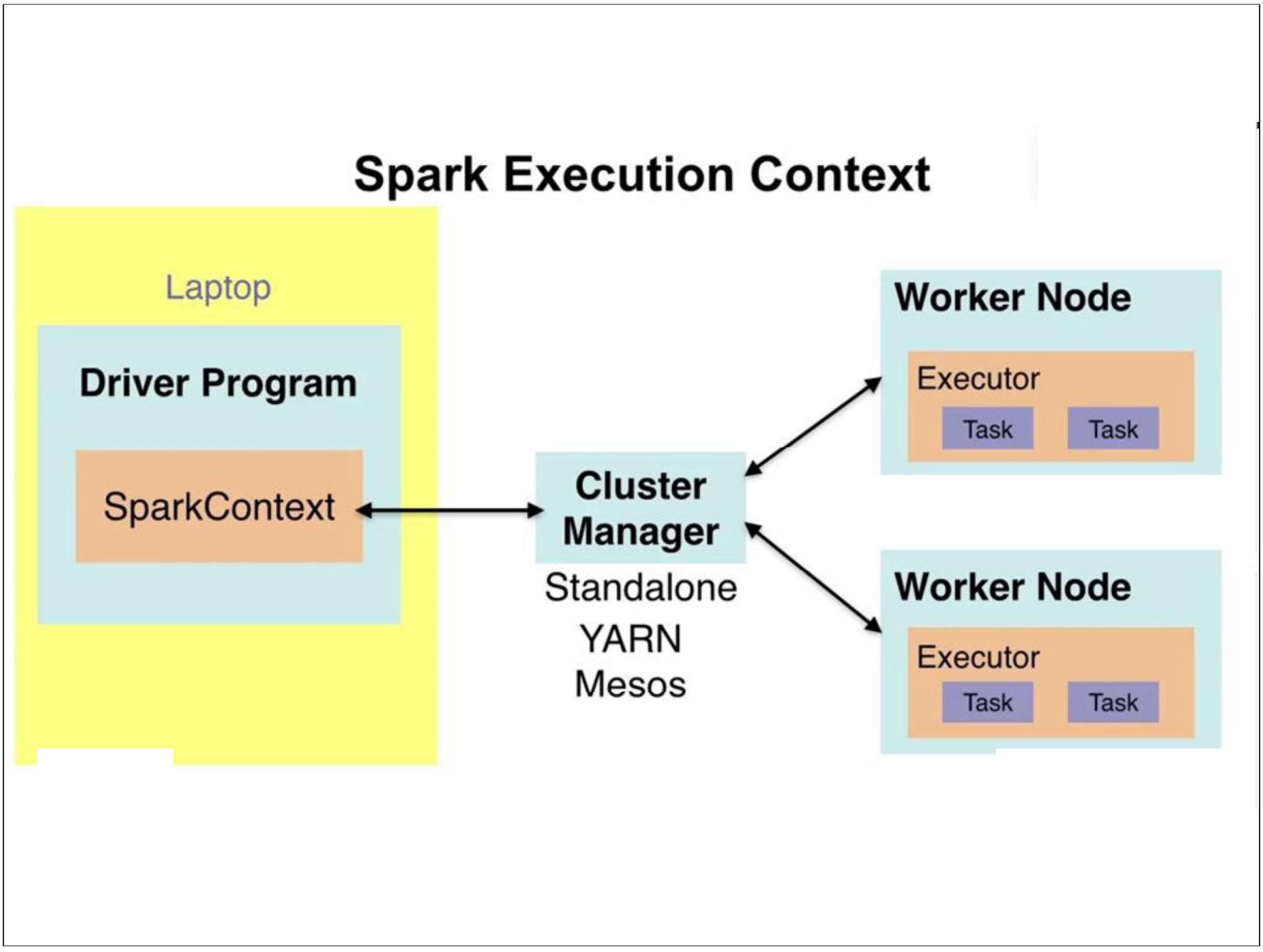

Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos


Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos

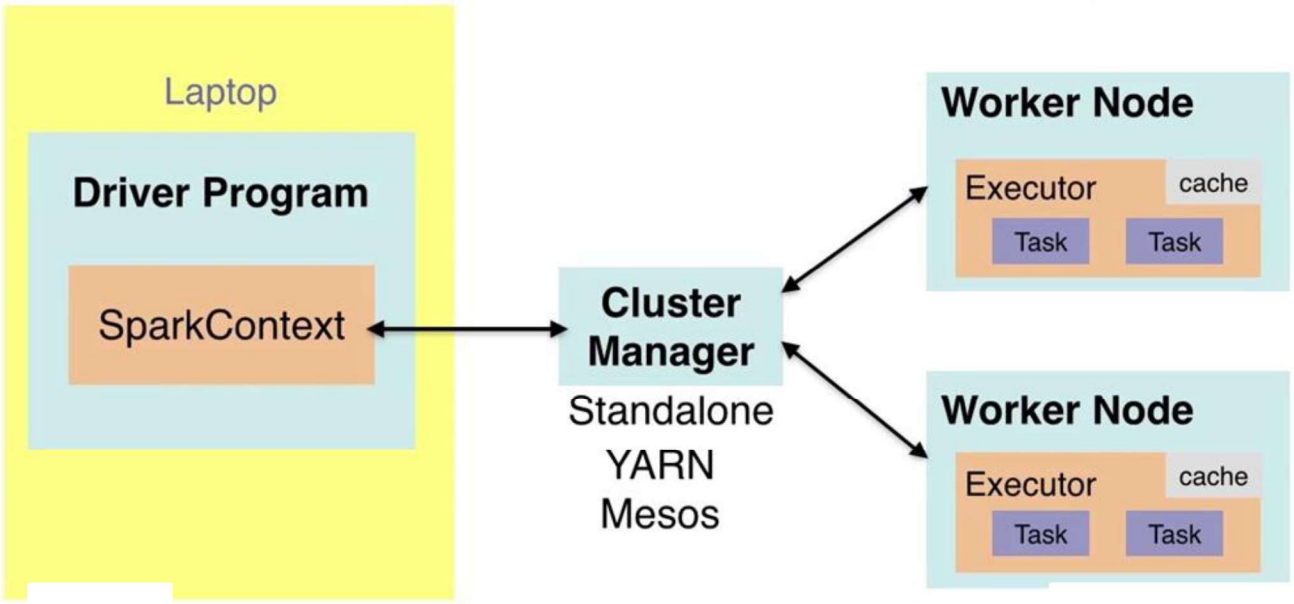

Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos
Worker Node
Worker Node


Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos
Worker Node
Worker Node


Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos
Worker Node
Executor
Worker Node
Executor


Spark Execution Context
Laptop
Driver Program
SparkContext
Cluster Manager
Standalone
YARN
Mesos
Worker Node
Executor
Task
Task
Worker Node
Executor
Task
Task

## Spark Execution Context




## Terminology

| *Term* | *Meaning* |
|---|---|
| Driver | *Process that contains the SparkContext* |
| Executor | *Process that executes one or more Spark tasks* |
| Master | *Process that manages applications across the cluster* |
| Worker | *Process that manages executors on a particular node* |

# Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = xrange(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()
```

Python Generator

Create RDD

Transformations

Action (materialize result)

# Functions Deconstructed

```
import random
flips = 1000000

# lazy eval
coins = xrange(flips)

# lazy eval, nothing executed
heads = sc.parallelize(coins) \
          .map(lambda i: random.random()) \
          .filter(lambda r: r < 0.51) \
          .count()

# create a closure with the lambda function
# apply function to data
```

Closures

# Functions Revisited

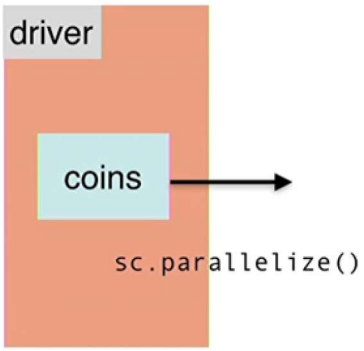
```
import random
flips = 1000000

# local sequence
coins = xrange(flips)

# distributed sequence
coin_rdd = sc.parallelize(coins)
flips_rdd = coin_rdd.map(lambda i: random.random())
heads_rdd = flips_rdd.filter(lambda r: r < 0.51)

# local value
head_count = heads_rdd.count()
```

# RDD Lineage

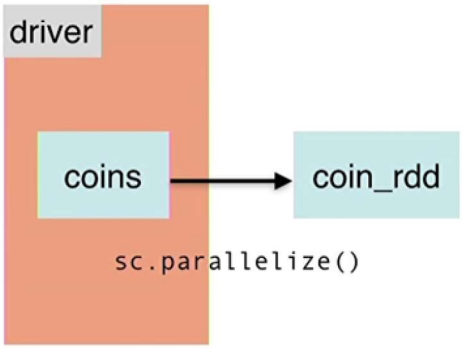
coins

# RDD Lineage

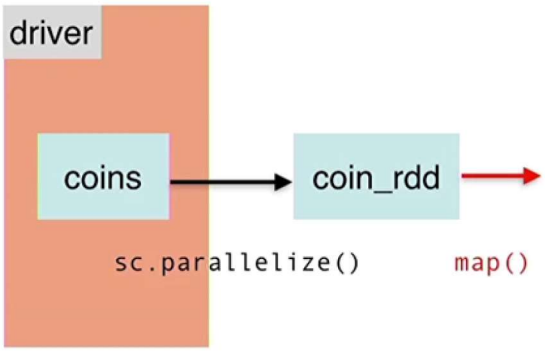

# RDD Lineage



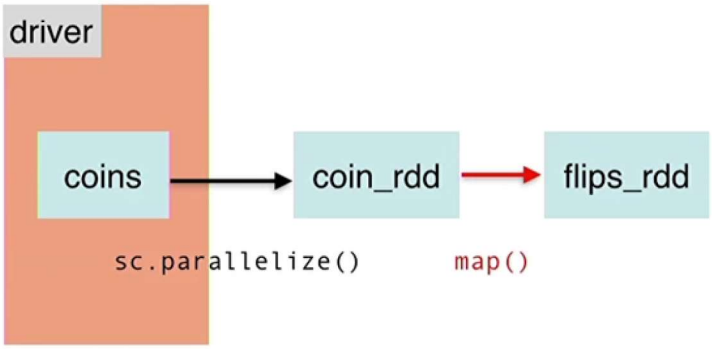
# RDD Lineage



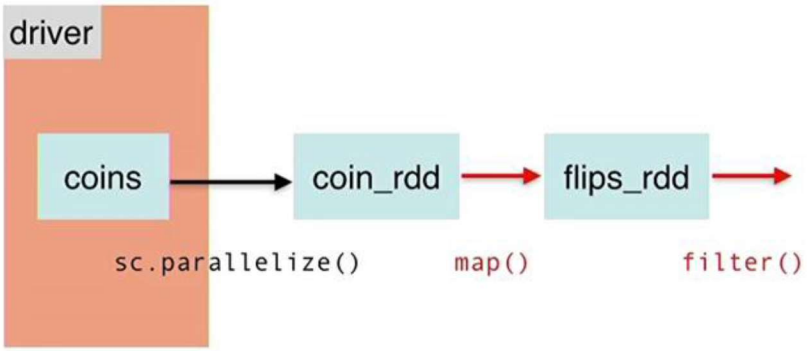

# RDD Lineage



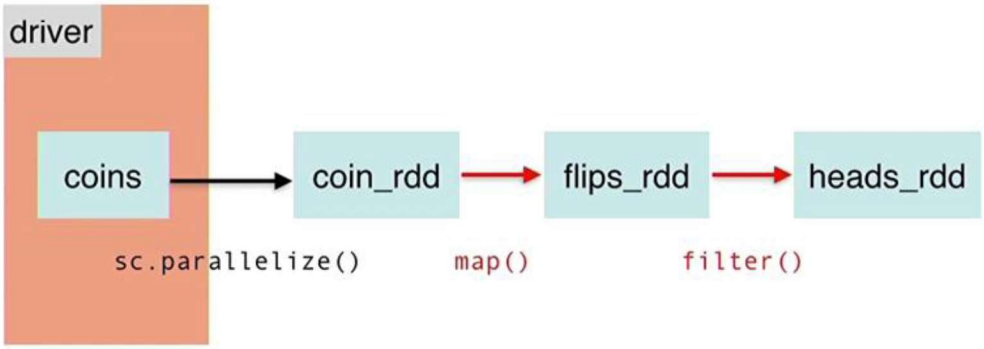
# RDD Lineage



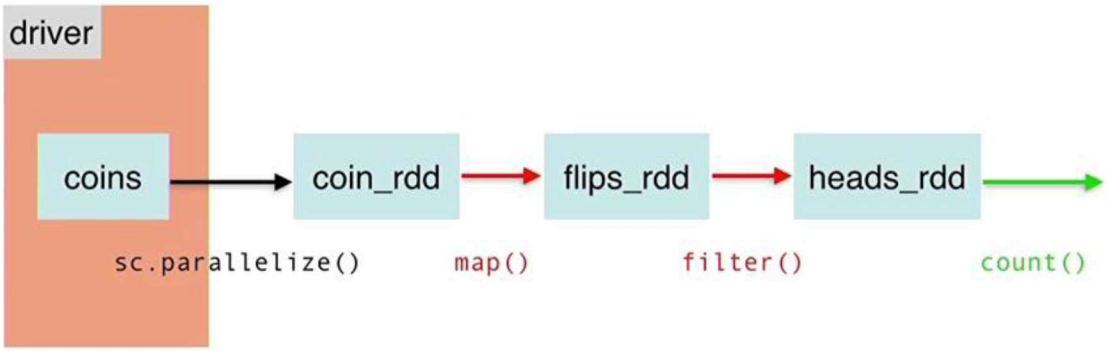

# RDD Lineage



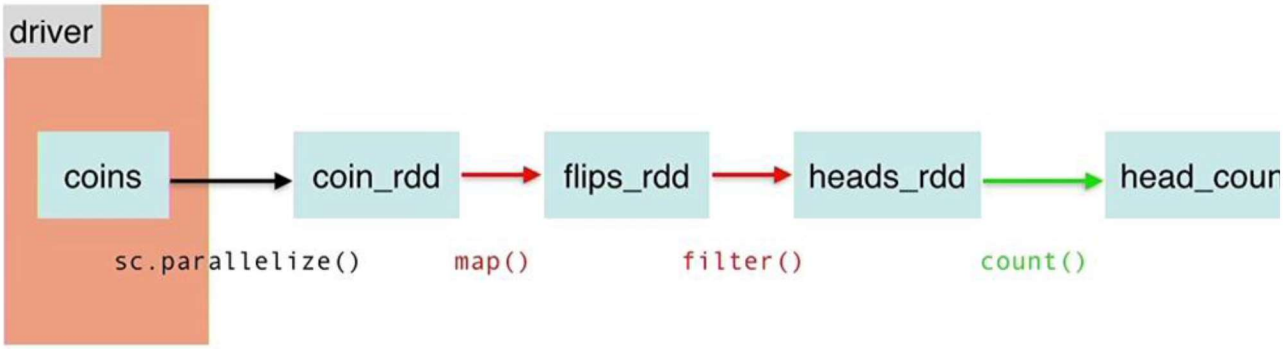
# RDD Lineage



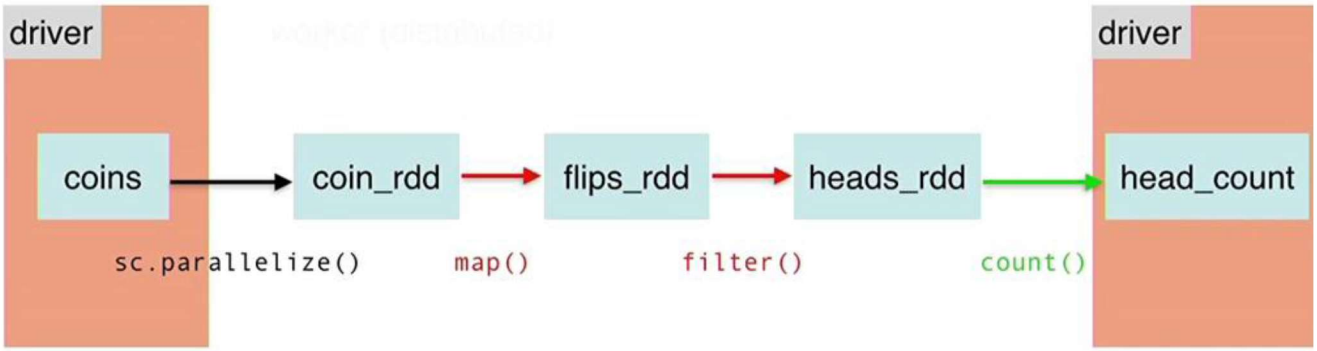

# RDD Lineage



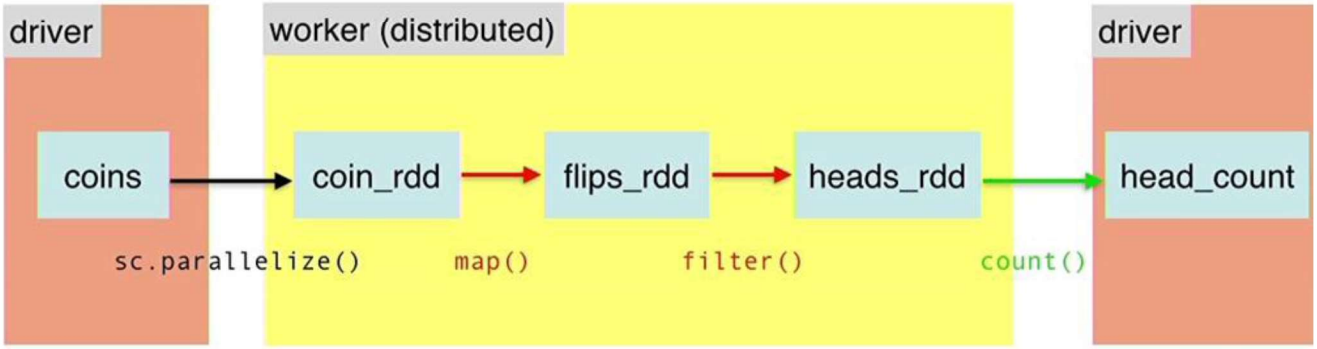
# RDD Lineage




# RDD Lineage



# RDD Lineage




# Functions Revisited

```
import random
flips = 1000000

# lazy eval
coins = xrange(flips)
```

nothing runs here

```
# lazy eval, nothing executed
heads_rdd = sc.parallelize(coins) \
              .map(lambda i: random.random()) \
              .filter(lambda r: r < 0.51)
```

```
head_count = heads_rdd.count()
```

Everything runs here

# SparkSQL

## Overview

- A library built on Spark Core that supports SQL like data and operations
- Make it easy for traditional RDBMS developers to transition to big data
- Works with "structured" data that has a schema
- Seamlessly mix SQL queries with Spark programs.
- Supports JDBC
- Helps "mix" n "match" different RDBMS and NoSQL Data sources

## DataFrame

- A distributed collection of data organized as rows and columns
- Has a schema – column names, data types
- Built upon RDD, Spark optimizes better since it knows the schema
- Can be created from and persisted to a variety of sources
  - CSV
  - Database tables
  - Hive / NoSQL tables
  - JSON
  - RDD

## Operations supported by Data Frames

- `filter` – filter data based on a condition
- `join` – join two Data Frames based on common column
- `groupBy` – group data frames by specific column values
- `agg` – compute aggregates like sum, average.
- `registerAsTempTable` – register the Data Frame as a table withi SQLContext
- Operations can be nested.

## SQLContext

- All functionality for Spark SQL accessed through a SQLContext
- Derived from SparkContext
- Data Frames are created through SQLContext
- Provides a standard interface to work across different data sources
- Can register Data Frames as temp table and then run SQL queries or them

# Schema Inference

## Structure RDD into Rows & Columns

**Summary of RDD transform**

```
RDD[String] -> RDD[ Row(Seq[Field]) ] -> DataFrame
```

- **Create a Schema programmatically**
  Similar to a "CREATE TABLE" by defining explicitly the name and datatype of each column

OR

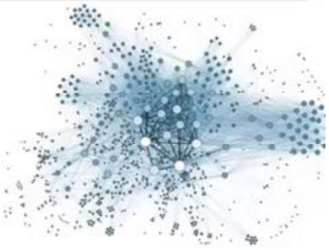
- **Via a case class**
  Create a case class and infer the schema via reflection

# RDD[String] from CSV

```
val elementsRdd = sc.parallelize(List(
   "17,Cl,Chlorine,35.453,1774,true,false",
  "38,Sr,Strontium,87.62,1790,false,false",
  "60,Nd,Neodymium,144.242,1885,false,false",
  "86,Rn,Radon,222.0176,1900,true,true",
  "115,Uuo,Ununoctium,294,2006,,true" ))
```

Unstructured data:

- Hard to interpret
- Queries using RDD methods are difficult to read and maintain



# DataFrame Schema created Programmatically (1/4)

**"Column Definition" of the Schema**

A collection of Fields to form a row: `StructType(fields: Seq[StructField])`

```
import org.apache.spark.sql.types._
val elemSchema = StructType(
  StructField(name="AtomicNumber", dataType=IntegerType, nullable=false) ::
  StructField("Symbol"       ,  StringType, false) ::
  StructField("ElementName"  ,  StringType, false) ::
  StructField("AtomicMass"   ,   FloatType, false) ::
  StructField("YearDiscovery", IntegerType, false) ::
  StructField("isGas"        , BooleanType, true ) ::
  StructField("isRadioactive", BooleanType, false) :: Nil)
```

# DataFrame, Schema created Programmatically (2/4)

**"Row Definition" of the Schema**
Transform RDD[String] → RDD[Row]

- A Row must be initialized by field values
- Field values: mirror the `StructField` declaration as defined the in the `StructType` previously

```
import org.apache.spark.sql.Row

val elemRows = elementsRdd.map(_.split(",")).
  map(line => Row(line(0).toInt, line(1), line(2),
      line(3).toFloat, line(4).toInt,
      if (line(5).isEmpty) null else line(5).toBoolean,
      line(6).toBoolean))
```

# DataFrame, Schema created Programmatically 3/4

DataFrame = Schema applied to RDD of Rows

```
val dfPeriodicTable = sqlContext.createDataFrame(elemRows, elemSchema)

dfPeriodicTable.printSchema

root
 |-- AtomicNumber: integer (nullable = false)
 |-- Symbol: string (nullable = false)
 |-- ElementName: string (nullable = false)
 |-- AtomicMass: float (nullable = false)
 |-- YearDiscovery: integer (nullable = false)
 |-- isGas: boolean (nullable = true)
 |-- isRadioactive: boolean (nullable = false)
```

# DataFrame, Schema created Programmatically 4/4

```
dfPeriodicTable.show
```

```
+------------+------+-----------+----------+-------------+-----+-------------+
|AtomicNumber|Symbol|ElementName|AtomicMass|YearDiscovery|isGas|isRadioactive|
+------------+------+-----------+----------+-------------+-----+-------------+
|          17|    Cl|   Chlorine|    35.453|         1774| true|        false|
|          38|    Sr|  Strontium|     87.62|         1790|false|        false|
|          60|    Nd|  Neodymium|   144.242|         1885|false|        false|
|          86|    Rn|      Radon|  222.0176|         1900| true|         true|
|         115|   Uuo| Ununoctium|     294.0|         2006| null|         true|
+------------+------+-----------+----------+-------------+-----+-------------+
```

# Schema inferred from Case Class 1/2

Schema inference via Case Class requires:

- A case class with the constructor parameters matching the columns of the desired schema
- An RDD of case class instances

```
case class MiniPeriodic(AtomicNumber: Int, Symbol: String, ElementName: String,
  AtomicMass: Double, YearDiscovery: Int,
  isGas: Option[Boolean], isRadioactive: Boolean)

val miniPeriodObjRDD = elementsRdd.map(_.split(",")).
  map(line => MiniPeriodic(line(0).toInt, line(1), line(2),
      line(3).toDouble, line(4).toInt,
      if (line(5).isEmpty) None else Some(line(5).toBoolean), line(6).toBoolean))
```

# Schema inferred from Case Class 2/2

- toDF() method converts an RDD of case class instances into a DataFrame.
- The inferred schema inference is not always perfectly equivalent to the "programmatic" schema.

```
// import sqlContext.implicits._ // automatic by spark-shell
val dfPeriodicTab2 = miniPeriodObjRDD.toDF

dfPeriodicTab2.printSchema
```

```
root
 |-- AtomicNumber: integer (nullable = false)
 |-- Symbol: string (nullable = true)      // string is always nullable
 |-- ElementName: string (nullable = true) // string is always nullable
 |-- AtomicMass: double (nullable = false)
 |-- YearDiscovery: integer (nullable = false)
 |-- isGas: boolean (nullable = true)
 |-- isRadioactive: boolean (nullable = false)
```

# Data Query
# SELECT

# SELECT (using API) 1/2

```
// 1) The ubiquitous SELECT * FROM …
dfPeriodicTable.show

// 2) SELECT by Column names (column addressed as string, Expr like Col1 +4 or Alias is not possible)
dfPeriodicTable.select("AtomicNumber", "ElementName", "AtomicMass", "YearDiscovery", "isGas").show

// 3) SELECT by Column names (column referred as org.apache.spark.sql.Column object)
dfPeriodicTable.select(dfPeriodicTable("AtomicNumber"), dfPeriodicTable("ElementName"),
                       dfPeriodicTable("YearDiscovery")).show

// 4) SELECT by Column names (v3: shortcut notation for Column object)
dfPeriodicTable.select($"AtomicNumber", $"ElementName", $"AtomicMass", $"YearDiscovery",
$"isGas").show

// 5) SELECT with Column Alias
dfPeriodicTable.select( $"ElementName", $"AtomicMass".cast("Int") as "MassRounded").show

// 6) SELECT with column Expression
dfPeriodicTable.select( $"Symbol", $"AtomicMass" * 2.0 as "Mass_x2").show
```

# SELECT (using API) 2/2

```
// 7) SELECT using built-in function from org.apache.spark.sql.functions
dfPeriodicTable.select(concat_ws("-", $"Symbol", $"Symbol") as "DoubleSymbol",
  upper($"ElementName") as "UpperName", format_number(log2($"AtomicMass"),3) as "Log2_Mass",
  when($"isGas", "Gas").otherwise("Solid") as "Phase", when($"isRadioActive", "Yes").otherwise("No") as "RadioActive"
).show

// 8) SELECT using SQL expressions as a string
dfPeriodicTable.selectExpr("upper(ElementName) as UpperName",
    "round(log2(AtomicMass),3) as Log2_Mass").show

// 9) SELECT ... LIMIT N (TOP N Rows)
dfPeriodicTable.limit(2).show
dfPeriodicTable.select($"AtomicNumber", $"ElementName", $"AtomicMass",
    $"YearDiscovery").show(numRows = 2)

// 10) Random Sample 50% of Rows
dfPeriodicTable.sample(withReplacement=false, fraction=0.50, seed=20151109).show
dfPeriodicTable.sample(false, 0.50).select($"Symbol", $"ElementName").show // No Seed
```

# WHERE (using API)

```
// 11) WHERE on numeric column
dfPeriodicTable.where("AtomicMass > 100").show  // using Condition Expression
dfPeriodicTable.where($"AtomicMass" > 100).show // using Column and Column operator
dfPeriodicTable.filter("AtomicMass < 100").show // filter() & where() are identical

// 12) WHERE on String column
dfPeriodicTable.where($"Symbol" === "Nd").show // Equality
dfPeriodicTable.where($"Symbol" !== "Nd").show // INequality
dfPeriodicTable.where($"ElementName" > "Radon").show // < and > also possible with string

// 13) AND conditions
dfPeriodicTable.where($"YearDiscovery" > 1899 && $"ElementName".contains("ium")).show

// 14) OR conditions. NOTE: rlike("[^t]ium") means contains 'ium' EXCEPT 'tium'
dfPeriodicTable.where($"isGas".isNull || $"ElementName".rlike("[^t]ium")).show
```

# SELECT (using SQL) 1/2

- Register DataFrame as a temporary table
- Same lifetime than the underlying SQLContext

```
// Mandatory to allow query in plain SQL syntax
dfPeriodicTable.registerTempTable("MiniPeriodic")


// 1) Simple SELECT
sqlContext.sql("SELECT * FROM MiniPeriodic").show

sqlContext.sql("SELECT AtomicNumber, ElementName, YearDiscovery, " +
  "isGas, isRadioactive FROM MiniPeriodic").show
```

# SELECT (using SQL) 2/2

```
// 2) Expressions
sqlContext.sql("SELECT concat_ws('-', Symbol, Symbol) AS DoubleSymbol, " +
  "upper(ElementName) AS UpperElemName, " +
  "format_number(log2(AtomicMass),3) AS Log2_Mass, " +
  "CASE WHEN isGas THEN 'Gas' ELSE 'Solid' END AS Phase, " +
  "CASE WHEN isRadioActive THEN 'Yes' ELSE 'No' END AS RadioActive FROM MiniPeriodic").show

// 3) TOP N Rows
sqlContext.sql("SELECT * FROM MiniPeriodic LIMIT 2").show

// 4) Random Sample
sqlContext.sql("SELECT * FROM MiniPeriodic TABLESAMPLE(50 PERCENT)").show
sqlContext.sql("SELECT * FROM MiniPeriodic TABLESAMPLE(3 ROWS)").show
```

# WHERE (using SQL)

```
// 5) WHERE on Numeric data type
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE AtomicNumber % 2 == 0 AND AtomicMass > 100").show

// 6) WHERE on String data type
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE Symbol  = 'Nd'").show // Equality
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE Symbol != 'Nd'").show // INequality
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE ElementName > 'Radon'").show

// 7) AND Condition, instr(ElementName, 'ium'): ElementName contains 'ium'
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE YearDiscovery > 1899 AND
instr(ElementName, 'ium') > 0").show

// 8) isGas == NULL OR ElementName contains 'ium' EXCEPT 'tium'
sqlContext.sql("SELECT * FROM MiniPeriodic WHERE isGas IS NULL OR
regexp_extract(ElementName, '[^t]ium', 0) != ''").show
```

# JSON Format (1/2)

- RDD[`string`] of JSON strings or JSON file
- Each record must be a complete JSON object
- Schema automatically inferred from the JSON object. (nested structures and arrays are supported)


# JSON Format (2/2)

Example: A book info is described by

```
+-------------+-----+-----+---------------+----------------+
|        Title|Price|eBook|         Author|        Language|
+-------------+-----+-----+---------------+----------------+
|  3D Printing|12.75|false|[5748,Joe Spark]|[EN, FR, ES, DE]|
|Scala Express|26.45| true|[6154,Jane Dora]|        [EN, FR]|
+-------------+-----+-----+---------------+----------------+
```

- Author is a nested structure having 2 members: Author.**ID** and Author.**Name**
- Language is an unbound array, with 0 to N elements

# JSON Sourced from RDD[String]

- RDD of Strings: one string per row
- Each string is a complete JSON object

```
val bookJsonRDD = sc.parallelize(List(
  """{"Title":"3D Printing", "Price":12.75, "eBook":false, "Author": {"Name":"Joe Spark", "ID":5748}, "Language": ["EN", "FR", "ES", "DE"] }"
  """{"Title":"Scala Express", "Price":26.45, "eBook":true, "Author": {"Name":"Jane Dora", "ID":6154}, "Language": ["EN", "FR"] }"""))

// Create DataFrame from RDD of JSON strings
val dfBookJsonRDD =
sqlContext.read.json(bookJsonRDD)

dfBookJsonRDD.printSchema
```

```
root
 |-- Author: struct (nullable = true)
 |    |-- ID: long (nullable = true)
 |    |-- Name: string (nullable = true)
 |-- Language: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- Price: double (nullable = true)
 |-- Title: string (nullable = true)
 |-- eBook: boolean (nullable = true)
```

# Query the DataFrame

```
// 1) API: SELECT all
dfBookJsonRDD.select("Title", "Price", "eBook", "Author", "Language").show

// 2) API: Extract individual value from a complex column type
dfBookJsonRDD.select($"Title", $"Author.Name" as "AuthorName",
  $"Author.ID" as "AuthorID", $"Language".apply(1) as "2ndLang").show

// 3) SQL: Same as #2
dfBookJsonRDD.registerTempTable("Book")
sqlContext.sql("SELECT Title, Author.Name, Author.ID, Language[1] AS 2ndLang FROM Book").show
```

```
+-------------+----------+--------+-------+
|        Title|AuthorName|AuthorID|2ndLang|
+-------------+----------+--------+-------+
|  3D Printing| Joe Spark|    5748|     FR|
|Scala Express| Jane Dora|    6154|     FR|
+-------------+----------+--------+-------+
```

# Source from JSON File

Create a file name MyLibrary.json containing 2 lines:

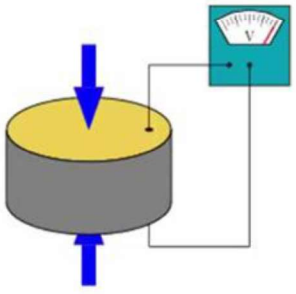
```
{"Title":"3D Printing", "Price":12.75, "eBook":false, "Author":{"Name":"Joe Spark", "ID":5748}, "Language":["EN", "FR", "ES", "DE"
{"Title":"Scala Express", "Price":26.45, "eBook":true, "Author":{"Name":"Jane Dora", "ID":6154}, "Language":["EN", "FR"]}
```

Read the JSON file using the DataFrameReader

```
val dfBookJsonFile = sqlContext.read.json("MyLibrary.json")
dfBookJsonFile.printSchema
dfBookJsonFile.select(etc.).show
```

# JSON - Force an explicit Schema (1/2)

- The Schema inferred from JSON is rather conservative:
  - Column is always nullable
  - Long is inferred for any Integer or Long
  - Double is inferred for any Float or Double



# JSON - Force an explicit Schema (2/2)

```
import org.apache.spark.sql.types._

val bookSchema = StructType(
  StructField(name="Title", dataType=StringType, nullable=false) ::
  StructField("Price"   ,  DoubleType, false) ::
  StructField("eBook"   , BooleanType, true ) ::
  StructField("Author"  , StructType(
              StructField("Name", StringType, false) ::
              StructField("ID", IntegerType, false) :: Nil ), false) ::
  StructField("Language", ArrayType(StringType), false) :: Nil)

val dfBookJsonRDD  = sqlContext.read.schema(bookSchema).json(bookJsonRDD)
val dfBookJsonFile = sqlContext.read.schema(bookSchema).json("MyLibrary.json")
```

# Data Query
# INNER JOINs

# Loading the MiniStore Dataset

```
val dfCustomer = sqlContext.read.json("<path                    >/Customers.json")
val dfProduct  = sqlContext.read.json("<path                    >/Products.json")
val dfOrder    = sqlContext.read.json("<path                    >/Orders.json")
```

# MiniStore - Customer table

```
dfCustomer.count // 18 records
dfCustomer.printSchema
dfCustomer.registerTempTable("Customer")
sqlContext.sql("SELECT * FROM Customer LIMIT 3").show
```

```
root
 |-- CustomerID: long (nullable = true)
 |-- Name: string (nullable = true)
 |-- DoB: string (nullable = true)
 |-- Rating: double (nullable = true)
 |-- Address: struct (nullable = true)
 |    |-- City: string (nullable = true)
 |    |-- Country: string (nullable = true)
 |    |-- State: string (nullable = true)
 |    |-- Street: string (nullable = true)
 |    |-- ZipCode: string (nullable = true)
 |-- PaymentMethods: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

```
+----------+----------------+----------+------+--------------------+--------------------+
|CustomerID|            Name|       DoB|Rating|             Address|      PaymentMethods|
+----------+----------------+----------+------+--------------------+--------------------+
|        16|   Dexter Gadget|1992-02-27|   3.5|[Reinfeld,null,MB...|[Debit Visa Paypa...|
|        27| Sebastian Gowda|1965-01-14|   4.5|[San Francisco,US...|[Cash GWallet Pay...|
|        33|     Chris Fuzzy|1956-09-11|   0.5|[St Féréol des-Ch...|[Paypal GWallet B...|
+----------+----------------+----------+------+--------------------+--------------------+
```

# MiniStore - Product table

```
dfProduct.count // 20 records

dfProduct.printSchema

dfProduct.registerTempTable("Product")

sqlContext.sql("SELECT * FROM Product LIMIT 3").show
```

```
root
 |-- ProductCode: string (nullable = true)
 |-- ProductName: string (nullable = true)
 |-- Category: string (nullable = true)
 |-- Price: double (nullable = true)
 |-- Unit: string (nullable = true)
 |-- isPerishable: boolean (nullable = true)
 |-- LastModified: string (nullable = true)
 |-- MiniOrderQty: long (nullable = true)
 |-- Origin: struct (nullable = true)
 |    |-- Country: string (nullable = true)
 |    |-- Manufacturer: string (nullable = true)
```

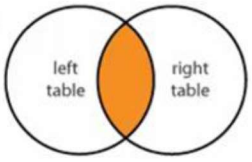
| ProductCode | ProductName | Category | Unit | Price | isPerishable | LastModified | MiniOrderQty | Origin |
|---|---|---|---|---|---|---|---|---|
| TO5674 | High Altitude Tom... | Food | kg | 3.75 | true | 2015-07-18 | 60 | [Peru,Machu Picch... |
| BB1509 | Bamboo Hardwood | Home HW | meter | 18.2 | true | 2015-03-29 | 50 | [Mongolia,Panda F... |
| ND8742 | Neutron Detector | HiTech | piece | 952.99 | false | 2014-12-13 | 1 | [Canada,SubAtomic... |

# INNER JOIN

SELECT ... FROM TableL L **INNER JOIN** TableR R ON L.Key = R.Key

Means:

- Equivalent to an intersection
- Select records from **L** and **R** having matching join keys
- The output records contains columns from both tables

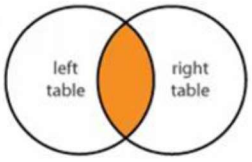


# INNER JOIN 2 Tables

```
// 1) API Syntax (columns referred as string name)
dfOrder.join(dfCustomer, "CustomerID").
  select("OrderID", "OrderDate", "ProductCode", "Qty", "CustomerID",
         "Name", "Rating", "Address").show(5)

// 2) SQL Syntax
dfCustomer.registerTempTable("Customer")
sqlContext.sql("SELECT OrderID, OrderDate, ProductCode, Qty, " +
  "O.CustomerID, Name, Rating, Address " +
  "FROM Order O INNER JOIN Customer C ON C.CustomerID = O.CustomerID LIMIT 5").show()
```

# INNER JOIN 3 tables + Column Expression (1/3)

- Join 3 tables
- Calculated Expression (Qty * Price)
- Use of built-in UDF (`format_number`, `upper`)
- Reference members of the complex structure "Address"
- Reference an element of the array "PaymentMethods"



# INNER JOIN 3 tables + Column Expression (2/3)

```
// 3) API Syntax (columns referred as Column datatype)
dfOrder.join(dfCustomer, "CustomerID").join(dfProduct, "ProductCode").
  select($"OrderID", $"OrderDate", $"ProductName", $"Qty", format_number($"Price",2) as "Price",
  format_number($"Qty" * $"Price", 2) as "QtyXPrice",
  $"CustomerID", $"Name", upper($"Address.City") as "CityName", $"PaymentMethods".apply(1) as "2ndPmtMethod").
  sample(withReplacement=false, fraction=0.02, seed=20151109).show


// 4) SQL Syntax
sqlContext.sql("SELECT OrderID, OrderDate, ProductName, Qty, " +
  "format_number(Price,2) AS Price, format_number(Qty * Price, 2) AS QtyXPrice, " +
  "O.CustomerID, Name, upper(Address.City) AS CityName, PaymentMethods[1] AS 2ndPmtMethod " +
  "FROM Order O " +
  "INNER JOIN Customer C ON C.CustomerID  = O.CustomerID " +
  "INNER JOIN Product  P ON P.ProductCode = O.ProductCode ").
  sample(withReplacement=false, fraction=0.02, seed=20151109).show
```

# AGGREGATES & SORTING (1/3)

## Sales Report:

- Grouped By Product
- Count the total orders per Product
- Compute Min, Max, Average Quantity
- Compute Total Sale Dollar amount
- Sort results by Total Sale Dollar amount in descending order

# AGGREGATES & SORTING (2/3)

```
// 1) API, NOTE: The GroupBy key is automatically selected in the output
dfOrder.join(dfProduct, "ProductCode").
  groupBy("ProductCode", "ProductName").
  agg(count($"OrderID") as "OrderCount",
      min($"Qty") as "MinQty", max($"Qty") as "MaxQty", mean($"Qty").cast("int") as "AvgQty",
      sum($"Qty") as "SumQty", format_number(sum($"Qty" * $"Price"), 2) as "TotalSales").
  orderBy(sum($"Qty" * $"Price").desc).show

// 2) SQL Language
sqlContext.sql("SELECT P.ProductCode, P.ProductName, count(*) AS OrderCount, " +
  "min(Qty) AS MinQty, max(Qty) AS MaxQty, cast(avg(Qty) as int) AS AvgQty, sum(Qty) AS SumQty, "
  "format_number(sum(Qty * Price),2) AS TotalSales " +
  "FROM Order O " +
  "INNER JOIN Product P ON P.ProductCode = O.ProductCode " +
  "GROUP BY P.ProductCode, P.ProductName " +
  "ORDER BY sum(Qty * Price) DESC").show
```

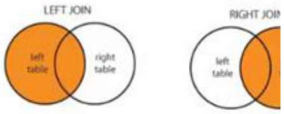
# Data Query
# OUTER JOINs, SEMI-JOIN

# OUTER JOINs (1/4)

SELECT ... FROM TableL L **LEFT OUTER JOIN** TableR R ON L.Key = R.Key

Means:

- All records of TableL are included (declared at the **LEFT** side of the JOIN operator)
- Select only records having matching join keys of TableR
- The resultset combines all columns of tables **L** and **R**.
  Rows having no matching keys with TableR will have all **R**.Columns = NULL
- A "RIGHT OUTER JOIN" is identical to LEFT OUTER JOIN.
  Except that the roles of the tables around the JOIN operator are reversed.




# OUTER JOINs (2/4)

An outer join is used when we want the content of an entire table even if some rows have no matching join key.

Example:

- List of ALL customers and the Total number of Purchases made per customer.
- Include customers who made zero purchase

"customers who made zero purchase" = the customers who have zero recoı in the Order table.

# OUTER JOINs (3/4)

```
// 1) Dataframe API
dfCustomer.join(dfOrder, dfOrder("CustomerID") === dfCustomer("CustomerID"), "left_outer").
  groupBy(dfCustomer("CustomerID"), dfCustomer("Name")).
  agg(count($"OrderID") as "OrderCount").orderBy(dfCustomer("CustomerID")).show

// 2) SQL Language
sqlContext.sql("SELECT C.CustomerID, C.Name, count(O.CustomerID) AS OrderCount " +
  "FROM Customer C " +
  "LEFT OUTER JOIN Order O ON C.CustomerID = O.CustomerID " +
  "GROUP BY C.CustomerID, C.Name " +
  "ORDER BY C.CustomerID").show
```

Stricter **API** Syntax:

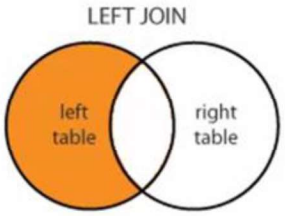
- join() method with JoinType parameter must reference columns via Column type
- Identical column names must be distinguished by their respective DataFrame (CustomerID in our example)
- The GroupBy key is automatically selected in the output

# OUTER JOINs (4/4)

```
+----------+------------------+----------+
|CustomerID|              Name|OrderCount|
+----------+------------------+----------+
|        16|      Dexter Gadget|        15|
|        27|    Sebastian Gowda|        15|
|        33|        Chris Fuzzy|        21|
|        55|    Antoine Lapêche|        13|
|        88|    Katya Goldfield|        11|
. . . Etc . . .
|       747|       Helena Mills|        14|
|       839|  Edward McGuinness|        17|
|       911|Satyendra Nath Bose|         0|  ← zero purchase
|       921|  Erwin Schrödinger|         0|  ← zero purchase
+----------+------------------+----------+
```

# OUTER JOINs - (not in) - 1/3

- "NOT IN": SELECT the "non-intersect" part between 2 tables
- LEFT OUTER JOIN: non-intersect part comes from LEFT table
- Typical business cases:
  - Customers who never made any purchase?
  - Any invalid Products referenced in the Order table?




# OUTER JOINs - (not in) - 2/3

```
// 3) API: Customers who never made any purchase
dfCustomer.join(dfOrder, dfOrder("CustomerID") === dfCustomer("CustomerID"), "left_outer").
  where(dfOrder("CustomerID").isNull).
  select(dfCustomer("CustomerID"), dfCustomer("Name"), $"Address").show


// 4) SQL
sqlContext.sql("SELECT C.CustomerID, C.Name, C.Address " +
  "FROM Customer C " +
  "LEFT OUTER JOIN Order O ON O.CustomerID = C.CustomerID " +
  "WHERE O.CustomerID IS NULL").show
```

```
+----------+------------------+--------------------+
|CustomerID|              Name|             Address|
+----------+------------------+--------------------+
|       911|Satyendra Nath Bose|[Kolkata,India,We...|
|       921|  Erwin Schrödinger|[Vienna,Austria,n...|
+----------+------------------+--------------------+
```

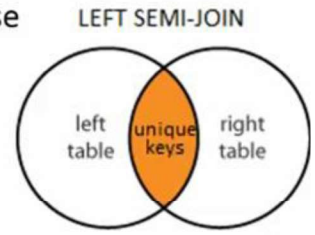
# OUTER JOINs - (not in) - 3/3

```
// 5) API: Invalid Products in the Order table
dfOrder.join(dfProduct, dfOrder("ProductCode") === dfProduct("ProductCode"), "left_outer").
  where(dfProduct("ProductCode").isNull).
  select($"OrderID", $"OrderDate", $"CustomerID", dfOrder("ProductCode"), dfProduct("ProductName"
dfProduct("Price"), $"Qty").show

// 6) SQL
sqlContext.sql("SELECT OrderID, OrderDate, O.CustomerID, O.ProductCode, P.ProductName, P.Price, O.Qty " +
  "FROM Order O LEFT OUTER JOIN Product P ON P.ProductCode = O.ProductCode " +
  "WHERE P.ProductCode IS NULL").show
```

```
+-------+----------+----------+-----------+-----------+-----+---+
|OrderID| OrderDate|CustomerID|ProductCode|ProductName|Price|Qty|
+-------+----------+----------+-----------+-----------+-----+---+
|   1000|2015-09-16|       601|     Çàéèìô|       null| null|  4|
|   1002|2015-10-13|       144|     ~~ZZZ~|       null| null| 12|
|   1004|2015-10-14|       401|     ^^VV^^|       null| null|  8|
|   1005|2015-10-14|        55|     **$$**|       null| null|  4|
|   1006|2015-10-15|       201|     ??????|       null| null|  1|
|   1008|2015-10-15|       295|     (?¿¡!)|       null| null|  5|
|   1010|2015-10-15|       295|    ÖåØÁæÉ?|       null| null|  1|
+-------+----------+----------+-----------+-----------+-----+---+
```

# SEMI-JOIN (1/2)

- A "LEFT SEMI-JOIN" returns rows from the LEFT table if there is one or mo matches in the RIGHT table.
- Duplicates are eliminated in the output
- Example of business case:
  List of Customers who have made at least 1 purchase
- Only the columns of the LEFT table are selectable.



# SEMI-JOIN (2/2)

```
// Left Semi-Join: Customers who have purchased at least 1 product
dfCustomer.join(dfOrder, dfCustomer("CustomerID") === dfOrder("CustomerID"), "leftsemi").
  select("CustomerID", "Name").show

sqlContext.sql("SELECT C.CustomerID, C.Name " +
  "FROM Customer C " +
  "LEFT SEMI JOIN Order O ON O.CustomerID = C.CustomerID").show
```

```
+----------+------------------+
|CustomerID|              Name|
+----------+------------------+
|        16|     Dexter Gadget|
|        27|   Sebastian Gowda|
|        33|      Chris Fuzzy|
|        55|   Antoine Lapêche|
|        88|   Katya Goldfield|
|       127|    William Hadley|
|       144|      Doug Quixote|
| . . Etc . .
|       363|         Xin Trien|
|       452|     Joel Beringer|
|       567|      Lars Neumann|
|       624|Krzysztof Geilswyk|
|       747|      Helena Mills|
|       839| Edward McGuinness|
+----------+------------------+
```

# Custom UDF
# (User Defined Function)

# Custom UDF

Built-in UDFs are not always sufficient

Examples of complex transforms on columns

- Convert WGS84 coordinates from [48°51'24.12"N, 2°21'2.88"E] to [48.8567, 2.3508]
- Calculate the weighted average of an array of numerical values
- Data integrity check (Postal Code, Credit Card number, etc.)
- Small Lookups

# UDF 1 – String manipulation

- Upper case a string
- Drop all vowels
- Replace space by dash
- Example: "Toronto Pearson Airport" becomes "TRNT-PRSN-RPRT"

```
def dropVowels(origTxt: String) = {
  val regexPtrn = """(?i)[AEIOU]""".r // (?i) case-insensitive match
  val noVowel = regexPtrn.replaceAllIn(origTxt, "").toUpperCase
  noVowel.replaceAll(" ", "-")
}
```

# UDF 2 – Exchange Rates

Convert an amount from US dollars to another currency

```
// Connect to a financial service, acquire exchange rates US Dollar -> Currency X
// Example:
// val mapUSDtoX = realtimeXchangeRate; mapUSDtoX("CAD")
// res1: Double = 1.3327 : means 1 USD = 1.3327 Canadian Dollars
def realtimeXchangeRate: Map[String, Double] = {
  Map(
    "AUD" -> 1.4085,  // Australian Dollar
    "BRL" -> 3.8112,  // Brazilian Real
    "CAD" -> 1.3327,  // Canadian Dollar
    "CHF" -> 1.0158,  // Swiss Franc
    "CNY" -> 6.3845,  // Chinese Yuan
    "EUR" -> 0.9402,  // Euro
    "GBP" -> 0.6580,  // British Pound
    "INR" -> 66.0815, // Indian Rupee
    "JPY" -> 123.4275,// Japanese Yen
    "MXN" -> 16.7473, // Mexican Peso
    "ZAR" -> 14.2949  // South African Rand
  )
}
```

# UDF registered as SQLContext.udf (1/2)

```
// store result of the "service call" in a variable for re-use
val mapUSDtoX = realtimeXchangeRate

sqlContext.udf.register("dropVowels", dropVowels(_:String))
sqlContext.udf.register("XChgUSDtoCAD",
  (priceUSD:Double, currencySymbol:String) => priceUSD * mapUSDtoX(currencySymbol))

sqlContext.sql("SELECT ProductName, dropVowels(ProductName) AS CrypticName, " +
  "format_number(Price,2) AS Price_USD, " +
  "format_number(XChgUSDtoCAD(Price, 'CAD'),2) as Price_CAD, " +
  "format_number(XChgUSDtoCAD(Price, 'EUR'),2) as Price_EUR, " +
  "format_number(XChgUSDtoCAD(Price, 'JPY'),2) as Price_JPY " +
  "FROM Product").show
```

# UDF registered as SQLContext.udf (2/2)

| ProductName | CrypticName | Price_USD | Price_CAD | Price_EUR | Price_JPY |
|---|---|---|---|---|---|
| High Altitude Tom... | HGH-LTTD-TMTS | 3.75 | 5.00 | 3.53 | 462.85 |
| Bamboo Hardwood | BMB-HRDWD | 18.20 | 24.26 | 17.11 | 2,246.38 |
| Neutron Detector | NTRN-DTCTR | 952.99 | 1,270.05 | 896.00 | 117,625.17 |
| Cubic 3D Memory | CBC-3D-MMRY | 3,584.50 | 4,777.06 | 3,370.15 | 442,425.87 |
| Portable FMRI | PRTBL-FMR | 1,460.00 | 1,945.74 | 1,372.69 | 180,204.15 |
| Pickles IceCream | PCKLS-CCRM | 26.50 | 35.32 | 24.92 | 3,270.83 |
| Chilli Jam | CHLL-JM | 9.95 | 13.26 | 9.35 | 1,228.10 |
| Ultrared Wasabi | LTRRD-WSB | 50.80 | 67.70 | 47.76 | 6,270.12 |
| Invisible Curtain | NVSBL-CRTN | 28.45 | 37.92 | 26.75 | 3,511.51 |
| Peach and Cream Corn | PCH-ND-CRM-CRN | 1.99 | 2.65 | 1.87 | 245.62 |
| Volcanic Basalmic... | VLCNC-BSLMC-VNGR | 16.95 | 22.59 | 15.94 | 2,092.10 |
| Mountain Lila Honey | MNTN-LL-HNY | 156.70 | 208.83 | 147.33 | 19,341.09 |
| Gorilla Glass 5 | GRLL-GLSS-5 | 875.80 | 1,167.18 | 823.43 | 108,097.80 |
| Soja Packed Powder | SJ-PCKD-PWDR | 66.77 | 88.98 | 62.78 | 8,241.25 |
| Fakir Mattress | FKR-MTTRSS | 487.30 | 649.42 | 458.16 | 60,146.22 |
| Butter Squash | BTTR-SQSH | 6.45 | 8.60 | 6.06 | 796.11 |
| Snowboarding Gloves | SNWBRDNG-GLVS | 12.85 | 17.13 | 12.08 | 1,586.04 |
| Stradiviarius Rep... | STRDVRS-RPLC | 650.40 | 866.79 | 611.51 | 80,277.25 |
| Pebble Neck Lace | PBBL-NCK-LC | 23.45 | 31.25 | 22.05 | 2,894.37 |
| Psychedelic Disco... | PSYCHDLC-DSC-LD | 248.62 | 331.34 | 233.75 | 30,686.55 |
| Jeans Full of Holes | JNS-FLL-F-HLS | 365.99 | 487.75 | 344.10 | 45,173.23 |
| Gold Plated Rotar... | GLD-PLTD-RTRY-PHN | 4,235.35 | 5,644.45 | 3,982.08 | 522,758.66 |

# UDF registered for DataFrame API

```
import org.apache.spark.sql.functions.udf

val dropVowelsUDF = udf(dropVowels(_:String))

// store result of the "service call" in a variable for re-use
val mapUSDtoX = realtimeXchangeRate

// NOTE: variable mapUSDtoX is usable directly, no UDF needed
dfProduct.select($"ProductName", dropVowelsUDF($"ProductName") as "CrypticName",
  format_number($"Price",2) as "Price_USD",
  format_number($"Price" * mapUSDtoX("CAD"), 2) as "Price_CAD",
  format_number($"Price" * mapUSDtoX("EUR"), 2) as "Price_EUR",
  format_number($"Price" * mapUSDtoX("JPY"), 2) as "Price_JPY"
).show
```

# Legacy Java Custom UDF for Hive

## Highly recommended to rewrite the UDFs into native Spark code

- Internal code optimization performed by Spark behind the scene
- Simpler code maintenance

## Hive custom Java UDFs are still usable

- The JAR of the Hive custom UDFs must be submitted to Spark using the --jars argument
- Must use HiveContext
- Declare and use the UDF with the same syntax as in Hive

```
hiveCtx = HiveContext(sc)
hiveCtx.sql("CREATE TEMPORARY FUNCTION MyCustomHiveUDF AS
   'edu.scalauniversity.MyClass.MyMethodName'")
hiveCtx.sql("SELECT Col1, MyCustomHiveUDF(Col2) AS NewCol2 FROM MyTable").show
```

# Shared Variables - Broadcast Variables

→ Normally a function passed over a Spark transformation or Action works on separate copies of the variables of the function

→ The variables are local to each machine and updates to these variables are not propagated back to the driver program

→ Spark provides two types of shared variables:- Broadcast variables and Accumulators

→ Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

→ For example, to give every node a copy of a large input dataset efficiently Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

# Shared Variables - Accumulators

→ Accumulators are variables that can only be "added" to through an associative operation

→ Used to implement counters and sums, efficiently in parallel

→ Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

→ Only the driver program can read an accumulator's value, not the tasks