



INF 553: Foundations and Applications of Data Mining (Summer 2019)



Introduction to Spark and Scala

Thanks for source slide and material to: Weiwei Duan and Dr. Heather Miller
<https://www.coursera.org/learn/scala-spark-big-data/home/welcome>



What is Scala?

Object-oriented programming language

```
class Point(val xc: Int, val yc: Int) {  
    var x: Int = xc  
    var y: Int = yc  
  
    def move(dx: Int, dy: Int) {  
        x = x + dx  
        y = y + dy  
        println ("Point x location : " + x);  
        println ("Point y location : " + y);  
    }  
  
}  
  
class Location(override val xc: Int, override val yc: Int,  
    val zc :Int) extends Point(xc, yc){  
    var z: Int = zc  
  
    def move(dx: Int, dy: Int, dz: Int) {  
        x = x + dx  
        y = y + dy  
        z = z + dz  
        println ("Point x location : " + x);  
        println ("Point y location : " + y);  
        println ("Point z location : " + z);  
    }  
}
```

```
object Demo {  
    def main(args: Array[String]) {  
        val loc = new Location(10, 20, 15);  
  
        // Move to a new location  
        loc.move(10, 10, 5);  
    }  
}
```

(Singleton) Object:
A class that can have only one instance.

Usually we use object to call the main function



What is Scala?

Functional programming language

```
def addInt1( a:Int, b:Int ) : Int = {  
    var sum = a + b  
    return sum  
}  
var res1=addInt1(1,2)
```

- **Anonymous Functions**

```
def addInt2=(a: Int, b: Int) => a+b  
var res2=addInt2(1,2)
```

- **Higher-Order Functions**

Functions that take other functions as parameters

```
def output(f: Int => String, v: Int) = f(v)  
def layout[A](x: A) = "[" + x.toString() + "]"  
println(output( layout, res2))
```

Final output:
[3]

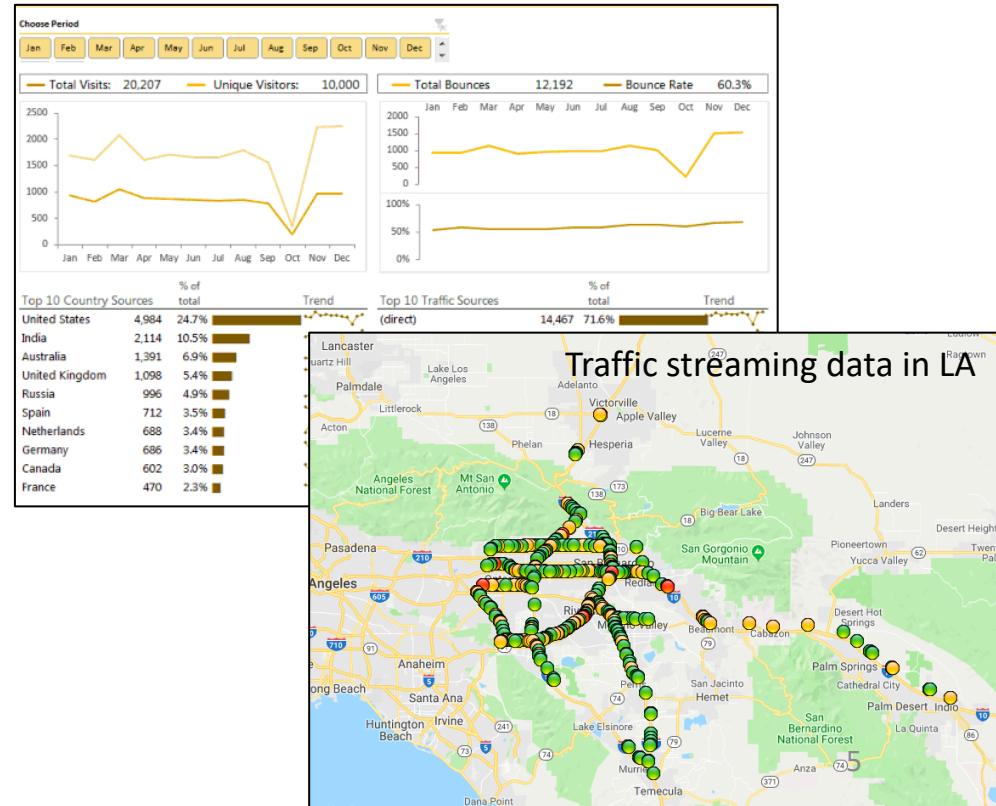


What is Spark?



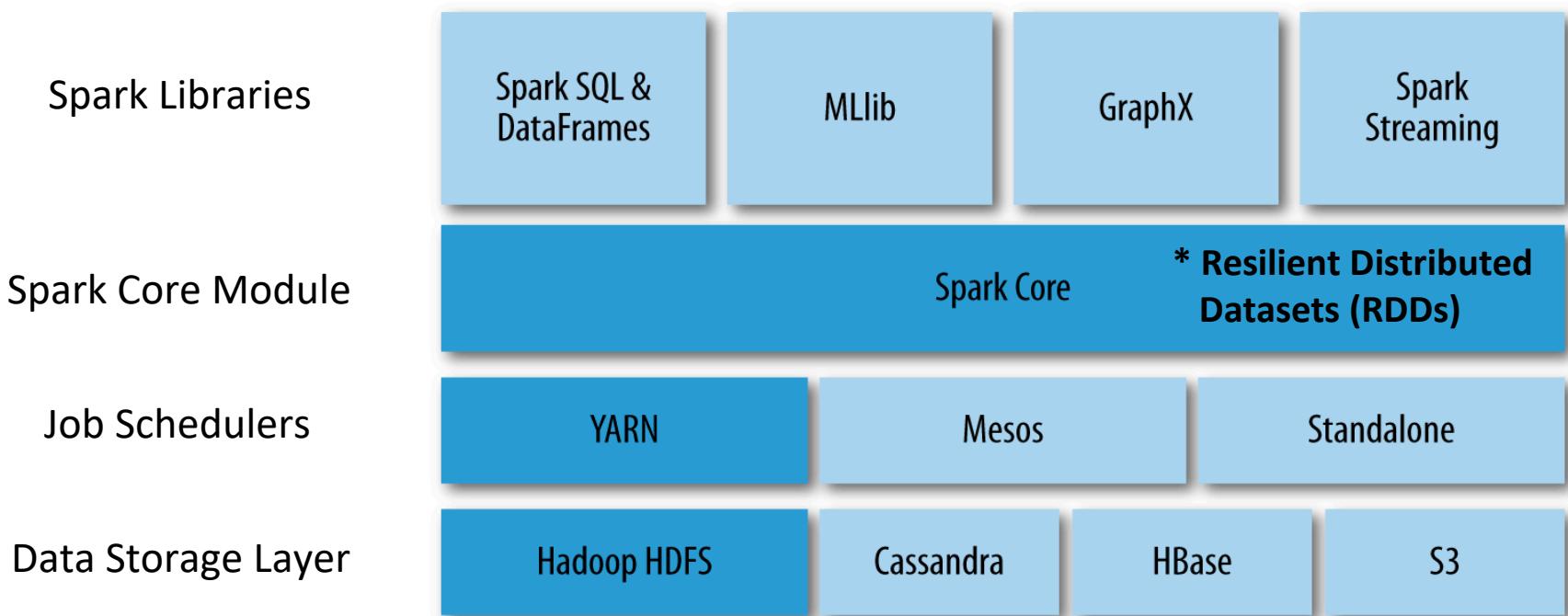
- **Apache Spark** is a unified analytics engine for large-scale data processing

- Application areas
 - Interactive Data Query
 - Real-time Data Analysis
 - Streaming Data Processing





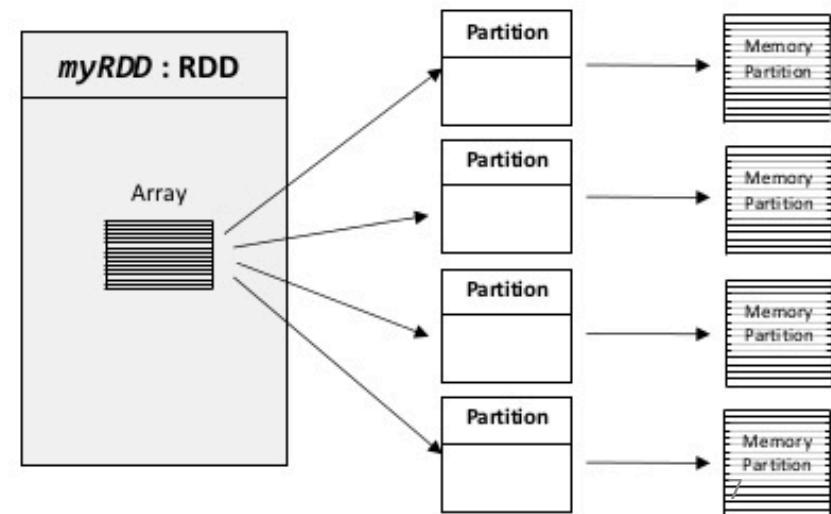
Spark Stack





Resilient Distributed Datasets (RDDs)

- An RDD is an **immutable, in-memory collection** of objects
- Each RDD can be split into multiple partitions, which in turn are computed on different nodes of the cluster
- RDDs seem a lot like Scala collections
 - `RDD[T]` and `List[T]`





How to create an RDD

- RDDs can be created in two ways:
 - Creating from a SparkContext object
 - Transforming from an existing RDD



How to create an RDD (Cont.)

- Creating from a **SparkContext** object
 - Can be thought as your handle to the Spark cluster
 - Represents the connection to a Spark cluster

```
val ss = SparkSession // Configure your Spark job here
  .builder()
  .appName("wordCount")
  .config("spark.master", "local[*]")
  .getOrCreate()

val sc = ss.sparkContext
```



How to create an RDD (Cont.)

- Creating from a **SparkContext** object
 - **parallelize**: convert a local Scala collection to an RDD

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]
```



How to create an RDD (Cont.)

- Creating from a **SparkContext** object
 - **parallelize**: convert a local Scala collection to an RDD

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]
```

- **textFile**: read a file from HDFS or local file system

```
val textRDD = sc.textFile("./text.txt")
```



How to create an RDD (Cont.)

- Transforming from an existing RDD
 - E.g., calling a *map operation* on an existing RDD, it will return a new RDD

```
// call a map operation on wordsRDD
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```



RDD Operations

- Transformations
 - E.g., map, filter, ...

```
// call a map operation on wordsRDD  
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```

- Actions
 - E.g., collect, reduce ...

```
val wordsColl = wordsRDD.collect() // RDD -> Collection  
print(wordsColl.toList) // List("you", "jump", "I", "jump", "")
```



Transformations VS Actions

- Transformations
 - Return new RDDs as results
 - They are **lazy**, the result RDD is not immediately computed
- Actions
 - Compute a result based on an RDD, and returned
 - They are **eager**, the result is **immediately computed**



Transformations VS Actions

- Transformations
 - Return new RDDs as results
 - They are **lazy**, the result RDD is not immediately computed

```
// call a map operation on wordsRDD
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```

- Actions
 - Compute a result based on an RDD, and returned
 - They are **eager**, the result is **immediately computed**

```
val wordsColl = wordsRDD.collect() // RDD -> Collection
print(wordsColl.toList) // List("you", "jump", "I", "jump", "")
```



Example

- Consider the following example:

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]  
  
// call a map operation on wordsRDD  
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?





Example (Cont.)

- Consider the following example:

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]  
  
// call a map operation on wordsRDD  
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?

Nothing. Execution of map (a transformation) is deferred.



Example (Cont.)

- Consider the following example:

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]  
  
// call a map operation on wordsRDD  
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?

Nothing. Execution of *map* (a transformation) is deferred.

How to ensure this computation is done on the cluster?



Example (Cont.)

- Consider the following example:

```
val aList: List[String] = List("you", "jump", "I", "jump", "")  
// Create an RDD from a list  
val wordsRDD = sc.parallelize(aList) // RDD[String]  
  
// call a map operation on wordsRDD  
val lengthRDD = wordsRDD.map(_.length) // RDD[Int]  
val totalChars = lengthRDD.reduce(_+_) // 12
```

add an action, *reduce*

Spark starts the execution when an action is called

Return the total number of characters in the entire RDD of strings



Benefits of Laziness

- Another example:

```
val logs: RDD[String] = ...
val logsWithErrors = logs.filter(_.contains("ERROR")).take(10)
```



Benefits of Laziness

- Another example:

```
val logs: RDD[String] = ...
val logsWithErrors = logs.filter(_.contains("ERROR")).take(10)
```

- The execution of *filter* is **deferred** until the *take* action happens
 - Spark will not compute intermediate RDDs. As soon as 10 elements of the filtered RDD have been computed, *logsWithErrors* is done.



Benefits of Laziness

- Another example:

```
val logs: RDD[String] = ...
val logsWithErrors = logs.filter(_.contains("ERROR")).take(10)
```

- The execution of *filter* is **deferred** until the *take* action happens
 - Spark will not compute intermediate RDDs. As soon as 10 elements of the filtered RDD have been computed, *logsWithErrors* is done
- Spark leverages this by analyzing and optimizing the **chain of operations** before executing it
 - Spark saves time and space to compute elements of the unused result of the *filter operation*



Common Transformations

map **map[T](f: A=>B): RDD[T]**

Apply function to each element in the RDD and return an RDD of the result.

flatmap **flatmap[T](f: A=>B): RDD[T]**

Apply function to each element in the RDD and return an RDD of the result, but output is flattened.

filter **filter[T](pred: A=>Boolean): RDD[T]**

Apply predicate function, pred, to each element in the RDD and return an RDD of elements that passed the condition.

distinct **distinct():RDD[T]**

Return an RDD with duplicates removed



Common Transformations

flatmap flatmap[T](f: A=>B): RDD[T]

Apply function to each element in the RDD and return an RDD of the result, but output is flattened.

```
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward", "")  
val textRDD = sc.parallelize(text)  
  
val splitText = textRDD.flatMap(phase => phase.split(" ")) // Flatten the output  
  
val splitTextColl = splitText.collect()  
splitTextColl.foreach(println) // "you", "me", "jump", "and", "run", "I", "love", "you", "jump", "forward"
```



Common Transformations

distinct **distinct():RDD[T]**

Return an RDD with duplicates removed

```
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward", "")  
val textRDD = sc.parallelize(text)  
  
val splitText = textRDD.flatMap(phase => phase.split(" ")) // Flatten the output  
val textDist = splitText.distinct() // Get the distinct words  
  
val textDistColl = textDist.collect()  
textDistColl.foreach(println) // "me", "I", "love", "run", "forward", "jump", "you", "and"
```



Common Actions

collect **collect: Array[T]**

Return all elements from RDD.

count **count(): Long**

Return the number of elements in the RDD.

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

Return the first num elements of the RDD.

reduce **reduce(op: (A, A) => A): A**

Combine the elements in the RDD together using op function and return result.

foreach **foreach(f: A => Unit): Unit**

Apply function to each element in the RDD, and return Unit.



Common Actions

count **count(): Long**

Return the number of elements in the RDD.

```
val text: List[String] = List("you and me", "jump and run", "I love you", "jump forward", "")  
val textRDD = sc.parallelize(text)  
  
val splitText = textRDD.flatMap(phase => phase.split(" ")) // Flatten the output  
val textDist = splitText.distinct() // Get the distinct words  
val counts = textDist.count() // return 8
```



Common Actions

foreach **foreach(f: A => Unit): Unit**

Apply function to each element in the RDD, and return Unit.

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)
```



Cluster Topology

- A simple example with *println*

```
case class Person(name: String, age: Int)  
  
val people: RDD[Person]  
people.foreach(println)
```

What happens?



How Spark jobs are Executed

Master-Worker
Topology





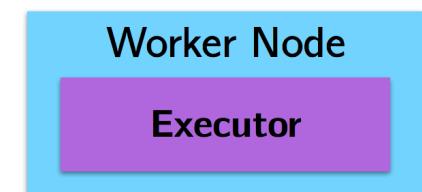
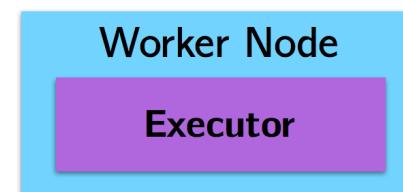
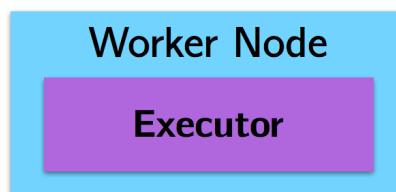
How Spark jobs are Executed

This is the node you're interacting with when you're writing Spark programs!



In the context of a Spark program

These are the nodes actually executing the jobs!





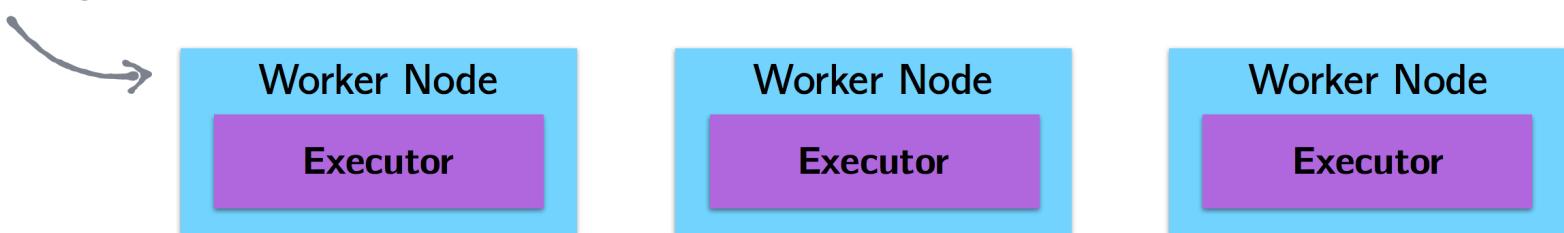
How Spark jobs are Executed

This is the node you're interacting with when you're writing Spark programs!



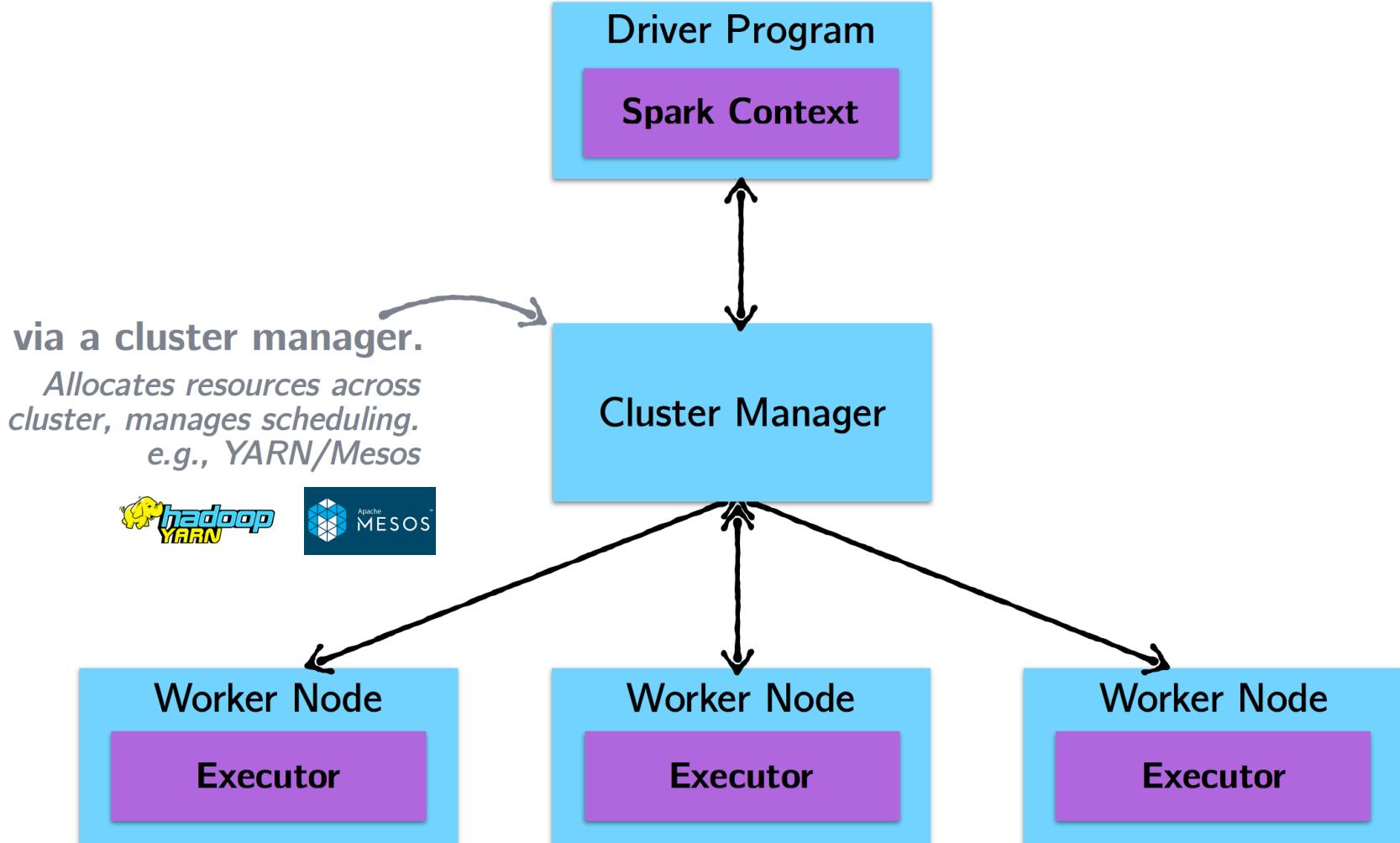
How do they communicate?

These are the nodes actually executing the jobs!



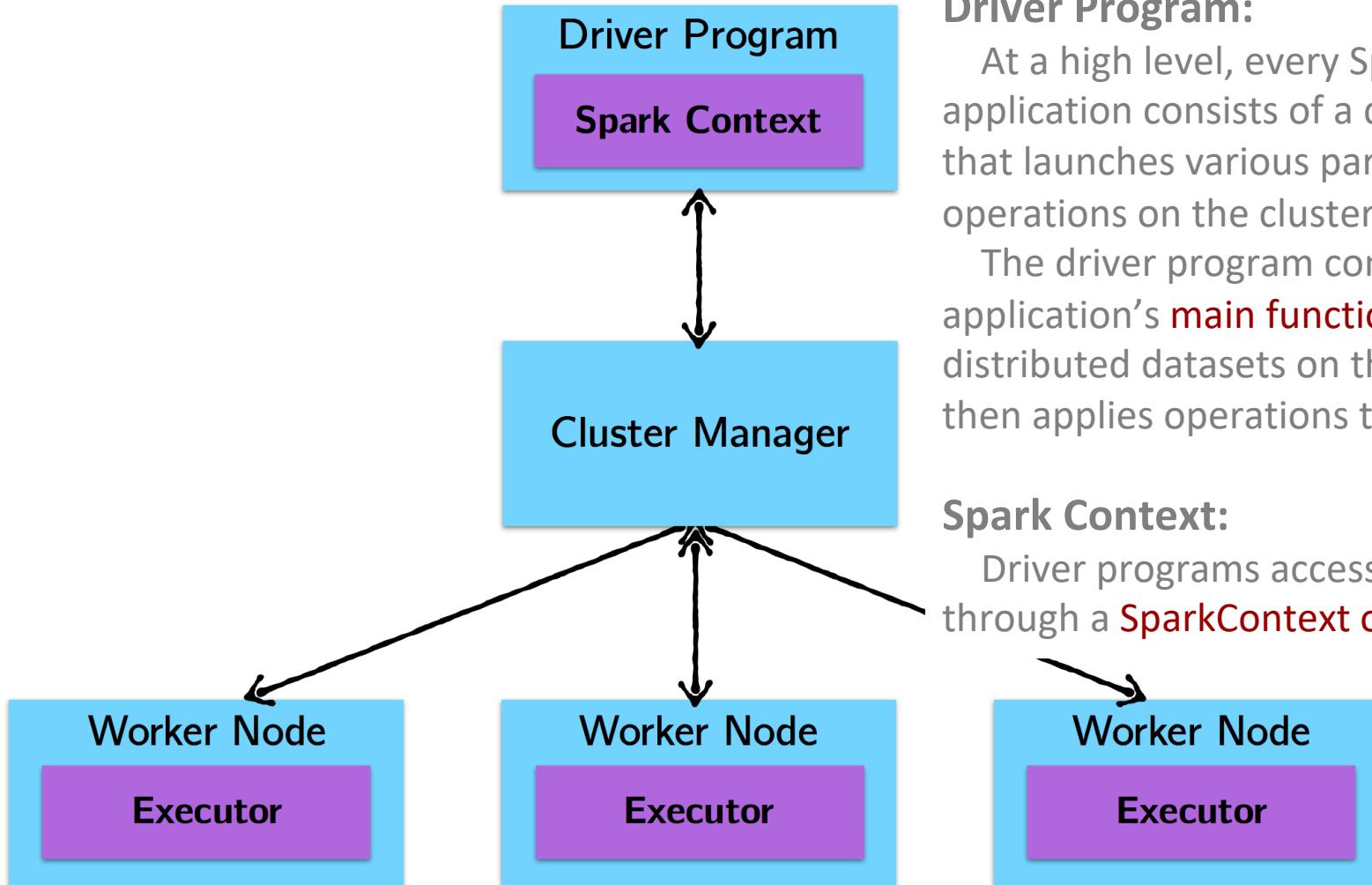


How Spark jobs are Executed





How Spark jobs are Executed



Driver Program:

At a high level, every Spark application consists of a driver program that launches various parallel operations on the cluster.

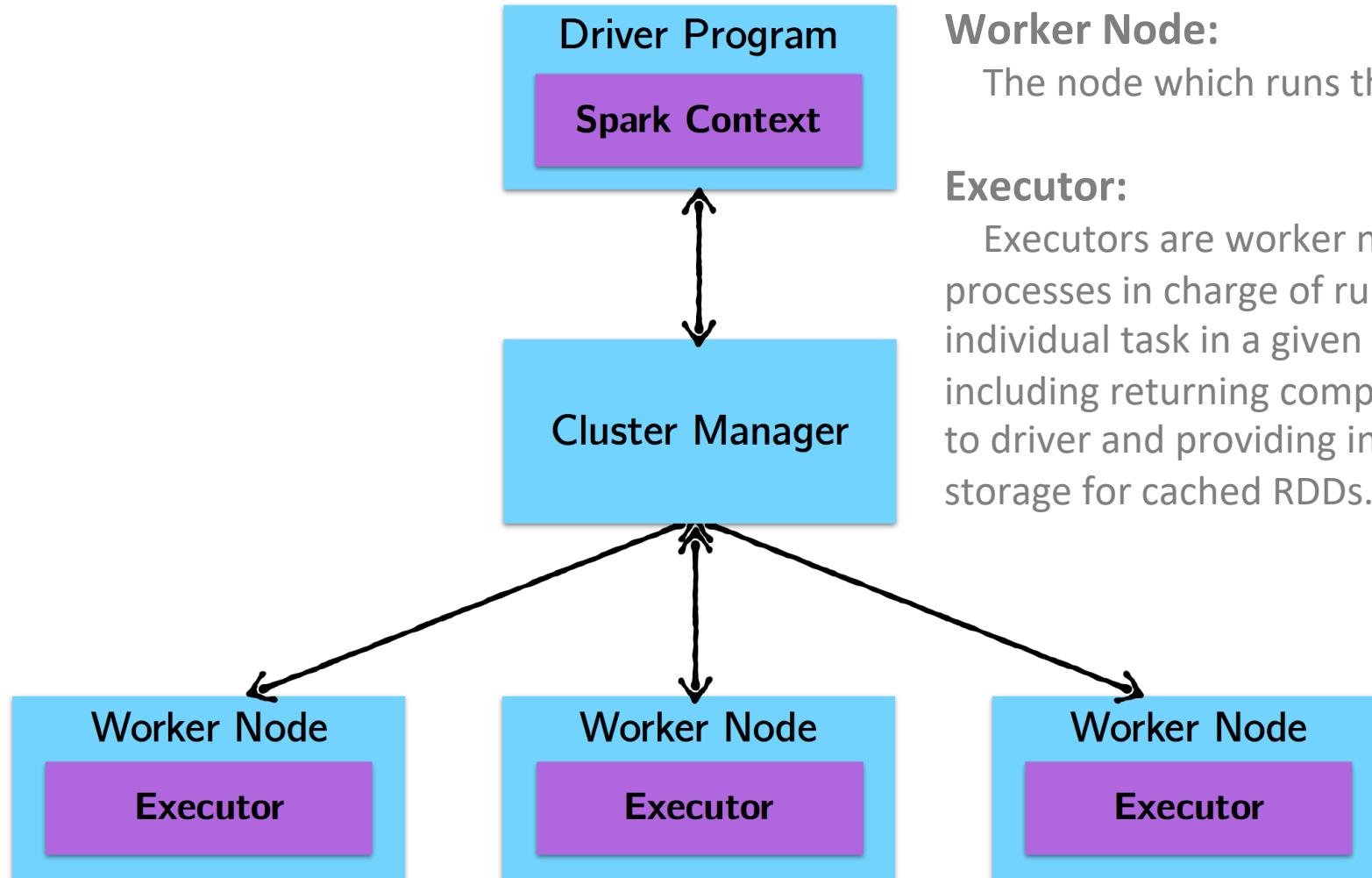
The driver program contains your application's **main function** and defines distributed datasets on the cluster, then applies operations to them.

Spark Context:

Driver programs access Spark through a **SparkContext object**



How Spark jobs are Executed



Worker Node:

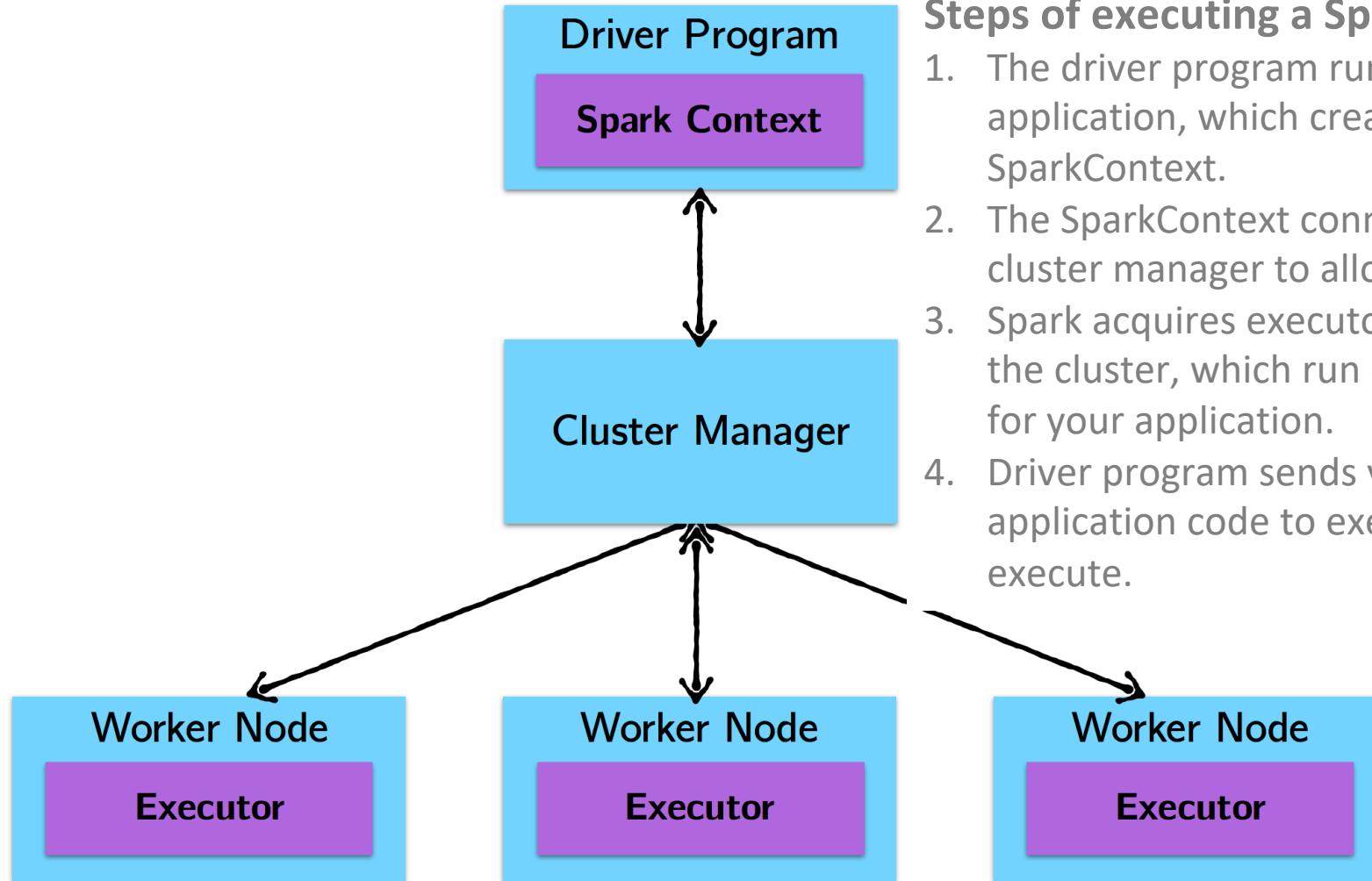
The node which runs the application

Executor:

Executors are worker nodes' processes in charge of running individual task in a given Spark job, including returning computed results to driver and providing in-memory storage for cached RDDs.



How Spark jobs are Executed



Steps of executing a Spark program:

1. The driver program runs the Spark application, which creates a `SparkContext`.
2. The `SparkContext` connects to a cluster manager to allocate resources.
3. Spark acquires executors on nodes in the cluster, which run computations for your application.
4. Driver program sends your application code to executors to execute.



Cluster Topology

- A simple example with *println*

```
case class Person(name: String, age: Int)  
  
val people: RDD[Person]  
people.foreach(println)
```

What happens?



Cluster Toplogy

- A simple example with *println*

```
case class Person(name: String, age: Int)  
  
val people: RDD[Person]  
people.foreach(println)
```

The driver node: Nothing.

The worker node: Print results.

Why? Recall that `foreach` is **an action**, with **return type Unit**. Therefore, it will be eagerly executed on the executors. Thus, any calls to *println* are happening on the worker nodes and are not visible in the drive node.



Cluster Topology

- Another simple example with *take*

```
case class Person(name: String, age: Int)  
  
val people: RDD[Person]  
people.foreach(println)  
  
val first10 = people.take(10)
```

Where will *first10* end up?



Cluster Topology

- Another simple example with *take*

```
case class Person(name: String, age: Int)

val people: RDD[Person]
people.foreach(println)

val first10 = people.take(10)
```

Where will *first10* end up? The driver program.

In general, executing an action involves communication between worker nodes and the node running the driver program.



Why Spark is Good for Data Sci

- In-memory computation
- RDD operations
 - Transformations: **Lazy**, deferred
 - Actions: **Eager**, kick off staged transformations
- Why Spark is good for data science?
 - Machine learning algorithms

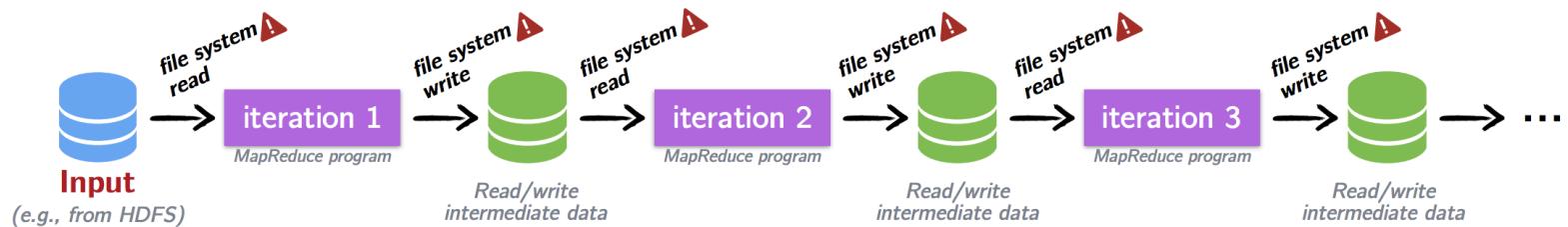




Iteration

- Most data science problems **involve iterations**

Iteration in Hadoop:

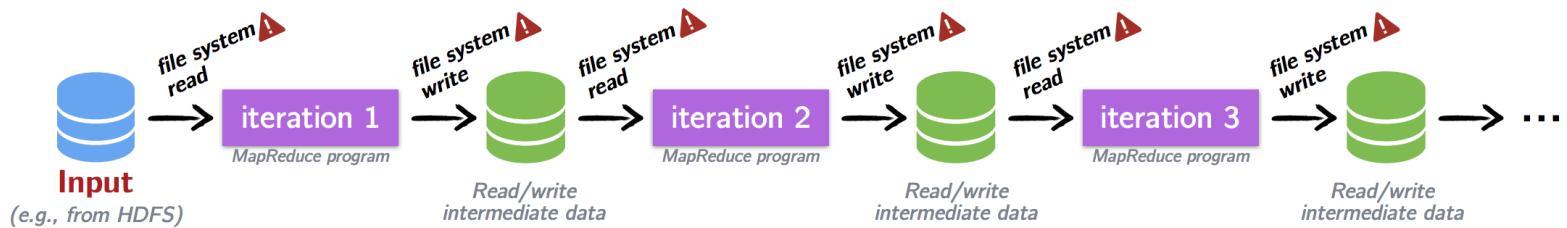




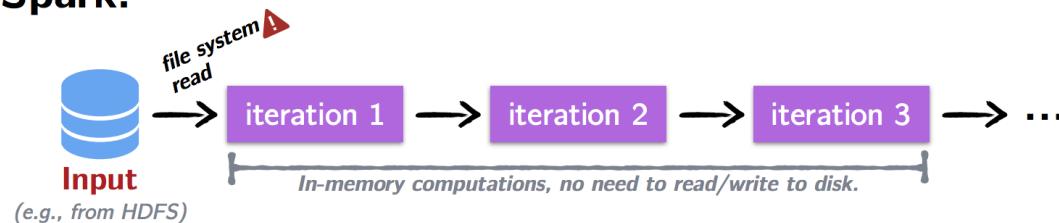
Iteration

- Most data science problems **involve iteration**

Iteration in Hadoop:



Iteration in Spark:

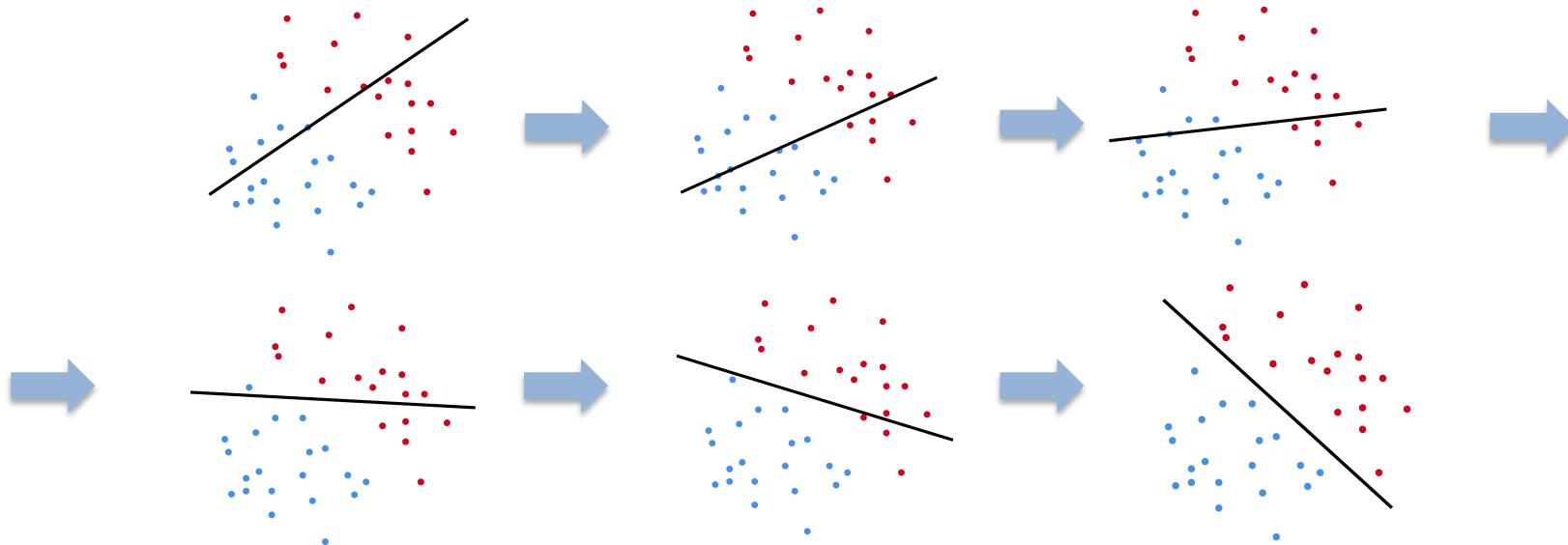




Iteration

Example: Logistic Regression

- Logistic regression is an **iterative algorithm** typically used for classification. Like most classification algorithms, it **updates weights iteratively** base on the training data.





Iteration

Example: Logistic Regression

- Logistic regression is an **iterative algorithm** typically used for classification. Like most classification algorithms, it **updates weights iteratively** base on the training data.

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$



Iteration

Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
    val gradient = points.map {p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_+_)
    w -= alpha * gradient
}
```

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$



Iteration

Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
    val gradient = points.map {p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_+_)
    w -= alpha * gradient
}
```

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^n g(w; x_i, y_i)$$

What is the weakness for this code?



Iteration

Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
    val gradient = points.map {p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_+_)
    w -= alpha * gradient
}
```

Spark starts the execution when the action *reduce* is applied



Iteration

Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint) // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
    val gradient = points.map {p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_+_)
    w -= alpha * gradient
}
```

points is being re-evaluated upon every iteration!
Unnecessary!



Caching and Persistence

- By default, RDDs are recomputed each time you run an action on them. This can be expensive (time-consuming) if you need to use a dataset more than once.

**Spark allows you to control what is cached in memory
use *persist()* or *cache()***

`cache()` : using the default storage level

`persist()`: can pass the storage level as a parameter,
e.g., “**MEMORY_ONLY**”, “**MEMORY_AND_DISK**”
default



Iteration

Example: Logistic Regression

- Logistic regression sample code:

```
val points = sc.textFile(...).map(parsePoint).persist() // case class Point(x: Double, y: Double)
var w = Vector.zero(d)
for(i <- 1 to numIterations) {
    val gradient = points.map {p =>
        g(p) // Apply the function of logistic regression
    }.reduce(_+_)
    w -= alpha * gradient
}
```

points is evaluated once and is cached in memory.
It can be re-used on each iteration.



Why Spark is Good for Data Sci

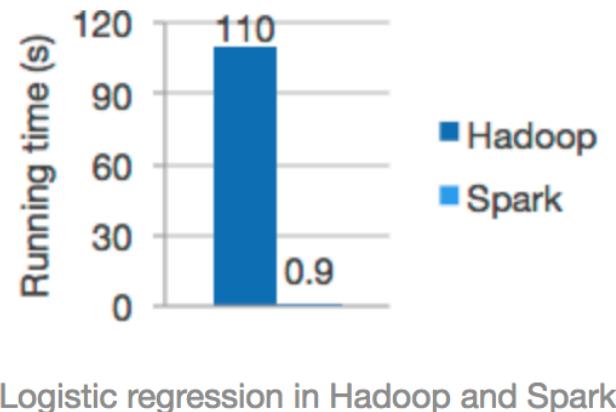
- **The lazy semantics** of RDD transformation operations help improve the performance.
- One of the most common performance bottlenecks for newcomers to Spark arises from unknowingly re-evaluating several transformations when caching could be used.



Spark vs. Hadoop

- **Spark is Faster**
 - When the output of an operation needs to be fed into another operation, Spark passes the data directly without writing to persistent storage
 - Better for some iterative algorithms
e.g. machine learning algorithms

Spark runs programs up to 100x faster than Hadoop MapReduce in memory. [1]





Other advantages

- **Easy to use**
 - Write applications quickly in Java, Scala, Python
- **Runs Everywhere**
 - Spark runs on Hadoop, standalone, or in the cloud
 - It can access diverse data sources including HDFS, Cassandra, HBase, and S3
- **Generality**
 - Combine SQL, streaming, and complex analytics





Install Spark

- Download Spark from official website:
 - <http://spark.apache.org/downloads.html>

Download Apache Spark™

1. Choose a Spark release: 2.3.1 (Jun 08 2018)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Download Spark: [spark-2.3.1-bin-hadoop2.7.tgz](#)
4. Verify this release using the [2.3.1 signatures and checksums](#) and [project release KEYS](#).

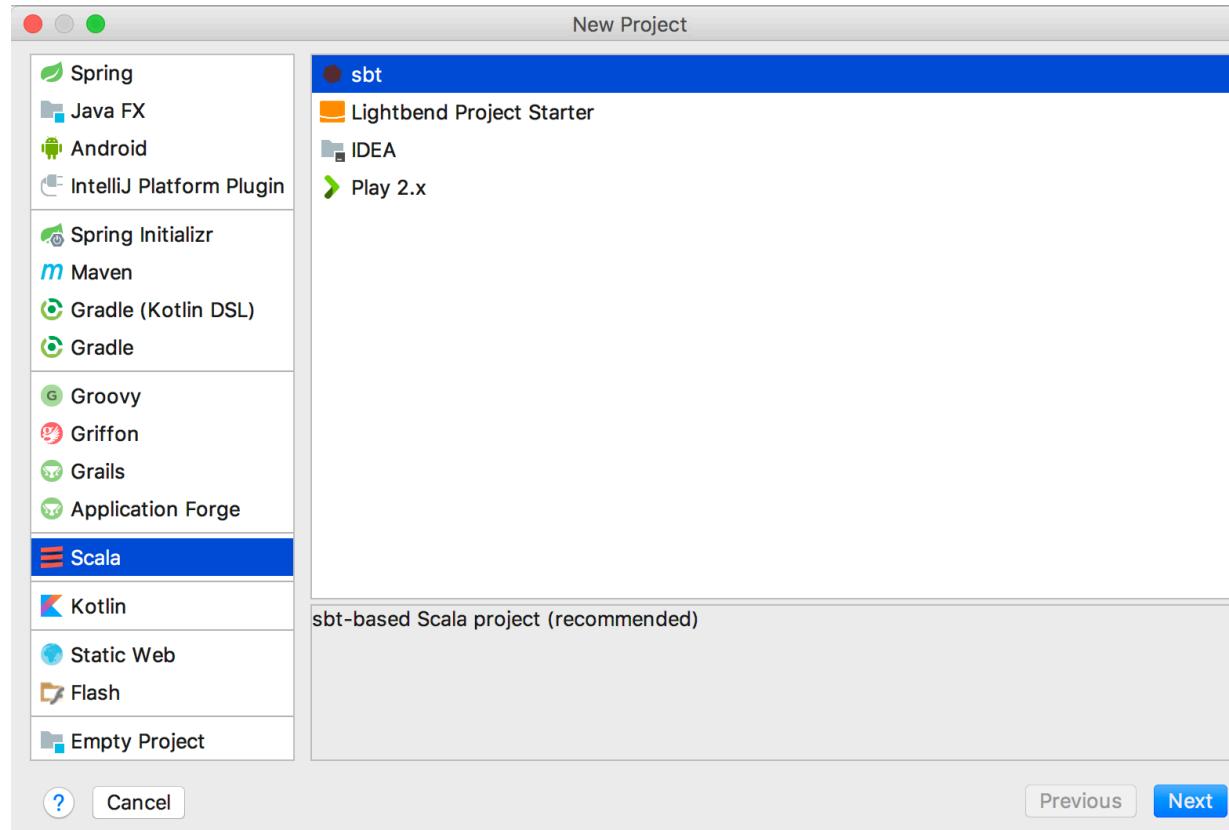


Install Scala

- IntelliJ IDEA, the compiler:
 - <https://www.jetbrains.com/idea/#chooseYourEdition>
- Install Scala plugin in the compiler
 - Open *Preference* -> Choose [*Plugins*] -> Click [*Install JetBrains plugin*] (at bottom) -> Search *Scala* and Install



Create an SBT project





Create an SBT project

New Project

Name: wordCount

Location: ~/IdeaProjects/wordCount

JDK: 1.8 (java version "1.8.0_144")

sbt: 1.1.0 Sources

Scala: Scala Sources

More Settings

Module name: wordCount

Content root: /Users/yijunlin/IdeaProjects/wordCount

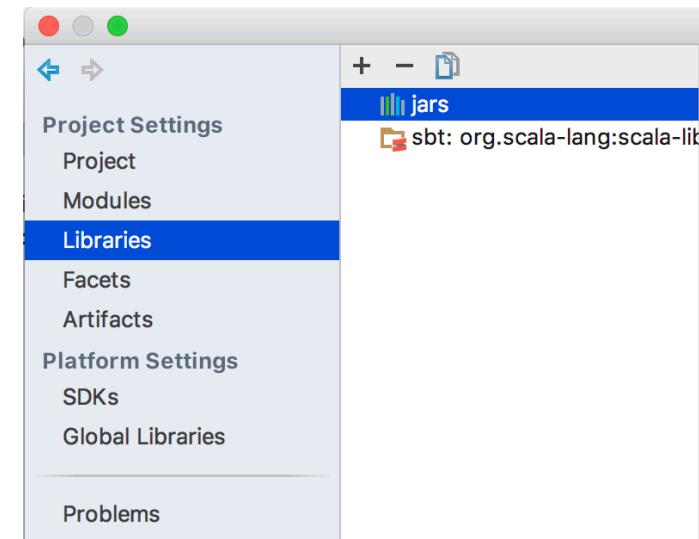
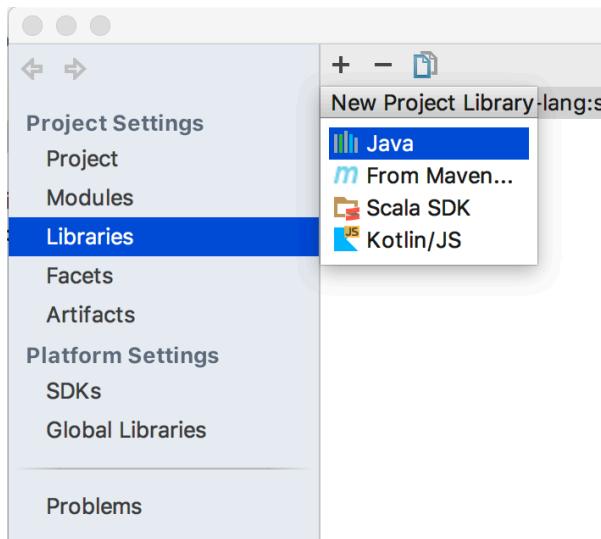
Module file location: /Users/yijunlin/IdeaProjects/wordCount

Project format: .idea (directory based)



Add Spark Environment

- You can add Spark either in **External Libraries** or through build.sbt
 - External Libraries:
 - Click [File] -> [Project Structure] -> [Libraries] -> [+] Java library -> Add the **jar package/.jar file** from the Spark you download





Add Spark Environment

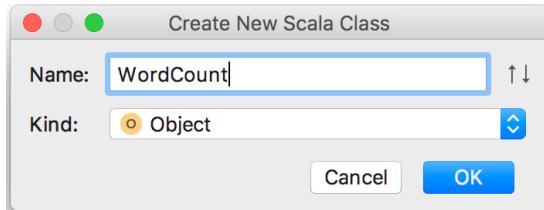
- You can also add Spark **through build.sbt**
 - build.sbt

A screenshot of a code editor window titled "build.sbt x". The code is written in Scala and defines a project configuration. It sets the project name to "wordCount", version to "0.1", and Scala version to "2.11.8". It specifies a sparkVersion of "2.1.0" and adds a resolver for Apache snapshots. The library dependencies section lists several org.apache.spark packages with their respective versions and sparkVersion constraints.

```
1  name := "wordCount"
2
3  version := "0.1"
4
5  scalaVersion := "2.11.8"
6
7  val sparkVersion = "2.1.0"
8
9  resolvers ++= Seq(
10    "apache-snapshots" at "http://repository.apache.org/snapshots/"
11  )
12
13  libraryDependencies ++= Seq(
14    "org.apache.spark" %% "spark-core" % sparkVersion,
15    "org.apache.spark" %% "spark-sql" % sparkVersion,
16    "org.apache.spark" %% "spark-mllib" % sparkVersion,
17    "org.apache.spark" %% "spark-streaming" % sparkVersion,
18    "org.apache.spark" %% "spark-hive" % sparkVersion
19  )
20
```



Write Scala Code



Create new Scala object

```
import org.apache.log4j.{Level, Logger}
import org.apache.spark.sql.SparkSession

object wordCount {

  def main(args:Array[String]): Unit = {
    Logger.getLogger("org").setLevel(Level.INFO)

    val ss = SparkSession // Configure your Spark job here
      .builder()
      .appName("wordCount")
      .config("spark.master", "local[*]")
      .getOrCreate()

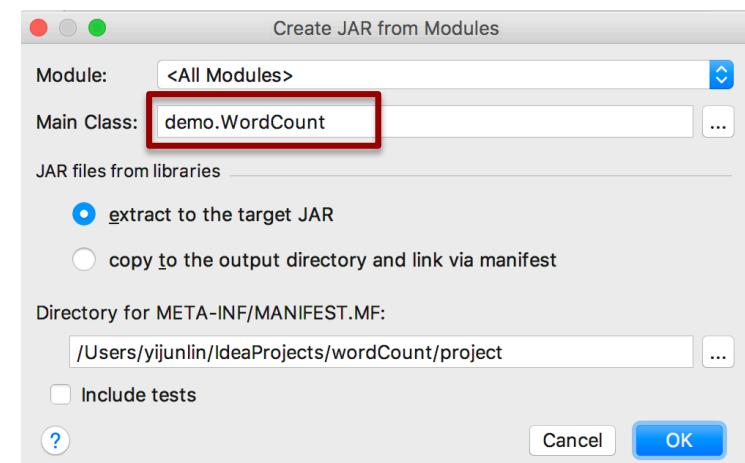
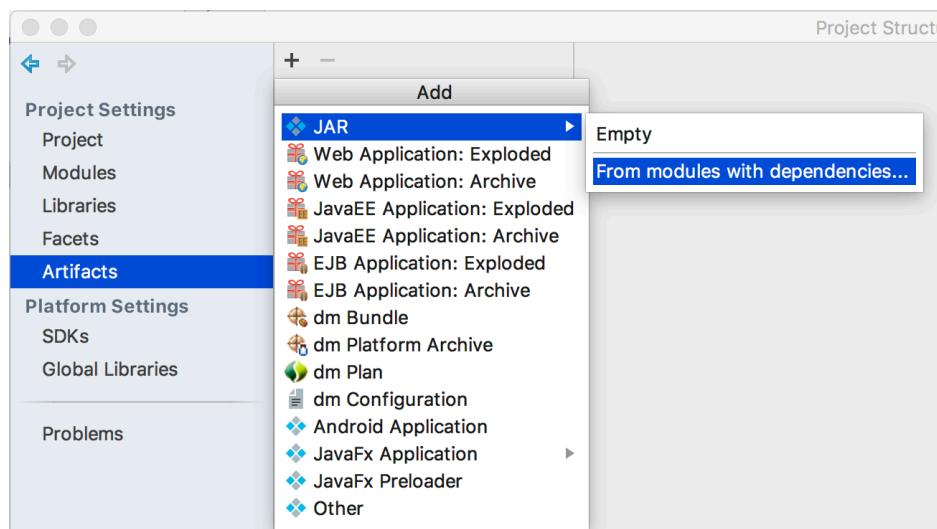
    val sc = ss.sparkContext
    val textRDD = sc.textFile("./text.txt")
    val counts = textRDD.flatMap(line => line.split(" ")).map(x => (x, 1)).reduceByKey(_+_).collect()

    counts.foreach(println)
  }
}
```



Build jar

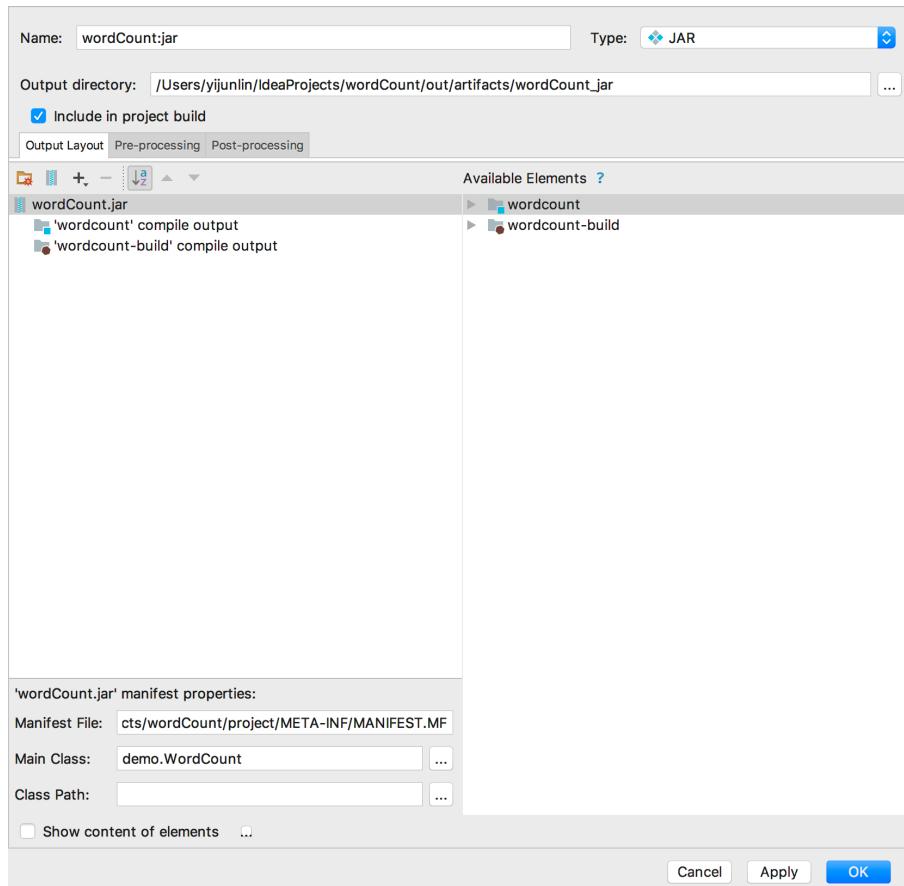
- Click [File] -> [Project Structure] -> [Artifacts] -> [+] JAR -> [From modules with dependencies] -> Put the Main Class of your code -> [OK]





Build jar

- You need to **delete all the spark libraries** or other unrelated libraries that you would not use in your program; Click [OK]
- Click [Build] -> [Build Artifacts] -> [Build] -> produce the package *out*, the jar file is in it!





Run jar on Spark - command line

```
[Yijuns-MacBook-Pro:~ yijunlin$ Tools/spark-2.1.0-bin-hadoop2.7/bin/spark-submit ]  
--class demo.WordCount --master local[2] ./IdeaProjects/wordCount/out/artifacts/  
wordCount_jar/wordCount.jar
```

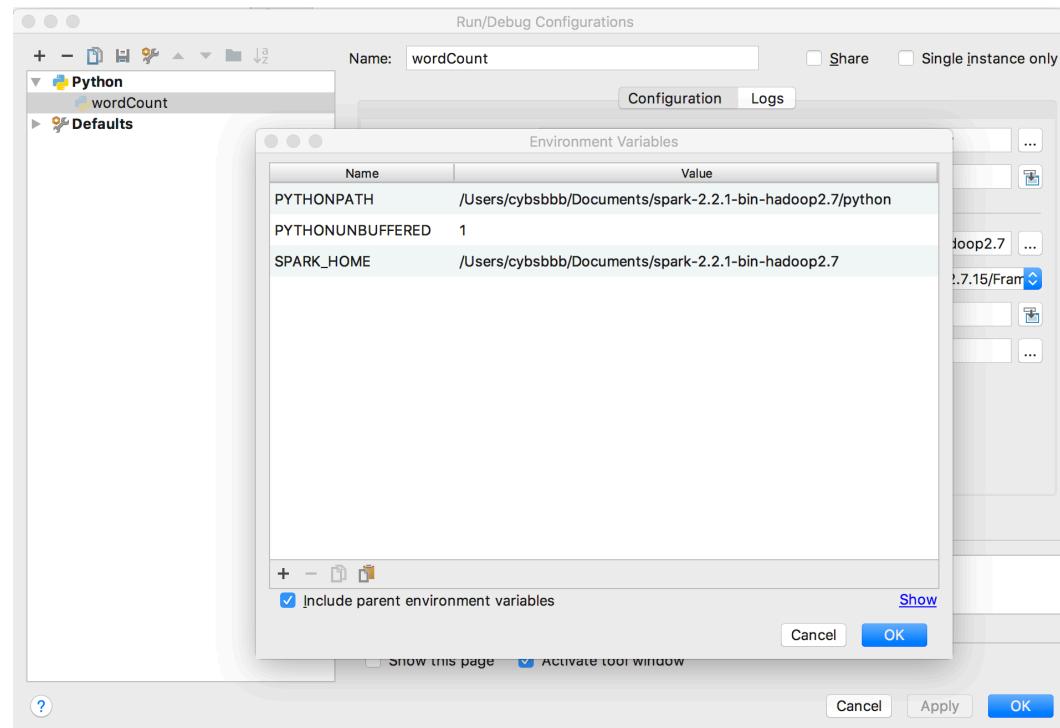
Show the Result:

(event,3)	(conditions.,1)
(customized,1)	(satellite,1)
(rate,1)	(geographical,1)
(video,2)	(for,3)
(Figure,1)	(decision-making,1)
(range,,1)	(detecting,1)
(integrating,1)	(Nearest,1)
((4),1)	(meter:,1)
(Event:,1)	((e.g.,,4)
(content,2)	(5,400,1)
(demonstrates,1)	(buses,1)



pyspark

- You can use pip install pyspark
- Or config in PyCharm:
 - click[Run] -> [Edit Configurations] -> Add[Environment variables]





pyspark

- Make sure keep the same python version for driver and worker
- Try the word_count.py to test the environment

The screenshot shows a code editor with two tabs: 'word_count.py' and 'text.txt'. The 'word_count.py' tab contains the following Python code:

```
1 from pyspark import SparkContext
2 import os
3
4 os.environ['PYSPARK_PYTHON'] = '/usr/local/bin/python3.6'
5 os.environ['PYSPARK_DRIVER_PYTHON'] = '/usr/local/bin/python3.6'
6
7 sc = SparkContext('local[*]', 'wordCount')
8
9 input_file_path = './text.txt'
10 textRDD = sc.textFile(input_file_path)
11
12 counts = textRDD.flatMap(lambda line: line.split(' '))
13 .map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b).collect()
14
15 for each_word in counts:
16     print(each_word)
```

The 'text.txt' tab shows the output of the word count program, which is a list of words and their counts:

```
('Apache', 1)
('Spark', 2)
('is', 1)
('general-purpose', 1)
('It', 2)
('provides', 1)
('high-level', 1)
('APIs', 1)
('in', 1)
('Scala,', 1)
('Java,', 1)
('Python', 1)
('make', 1)
('parallel', 1)
('write,', 1)
('an', 1)
('optimized', 1)
('engine', 1)
('supports', 2)
('computation', 1)
```



Run python on Spark

- command line

- Make sure keep the same python version for driver and worker
 - Edit ./conf/spark-env.sh (copy from ./conf/spark-env.sh.template)
 - Add environment variables

```
export PYSPARK_PYTHON=/usr/local/bin/python3.6
export PYSPARK_DRIVER_PYTHON=/usr/local/bin/python3.6
```

```
vpn-052-143:spark-2.3.1-bin-hadoop2.7 yijunlin$ bin/spark-submit ../../PycharmProjects/inf553spring2019/word_count.py
```



Configuration on Linux and Windows

- Install python version
 - You can find many tutorial on Google
- Install JDK
 - Download jdk and config PATH in bash file
- Download Spark and config PATH
 - SPARK_HOME and PYTHONPATH



If you want to learn more...

- Official documentation
 - <http://spark.apache.org/docs/latest/>
- Online course
 - Coursera: Big Data Analysis with Scala and Spark
- Books
 - *Learning Spark, O'Reilly*
 - Advanced Analytics with Spark: Patterns for Learning from Data at Scale, *O'Reilly*
 - *Machine Learning with Spark, Packt*