Introduction to Spark and Scala

Professor Wei-Min Shen

University of Southern California

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

Outline

- How to write Spark program (syntax)
- Key concepts in Spark
- Speed up your programs
- Install Spark on your machine
- Vocareum.com: use Spark on their terminal window
- Other reading materials and links
- Attachments: (many examples for you to read)
- Quiz logistics

What is Spark?

- **Apache Spark** is an open-source cluster-computing framework

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

Resilient Distributed Dataset (RDD) - the key data structure

- Distributed
  - Data are split into multiple partitions, distributed across nodes to be processed in parallel
- Resilient
  - Spark keeps track of transformations and enable efficient recovery
- Built-in data structure
  - Can be used inside Spark, but can't be accessed directly from outside (such as Python or Pyspark)

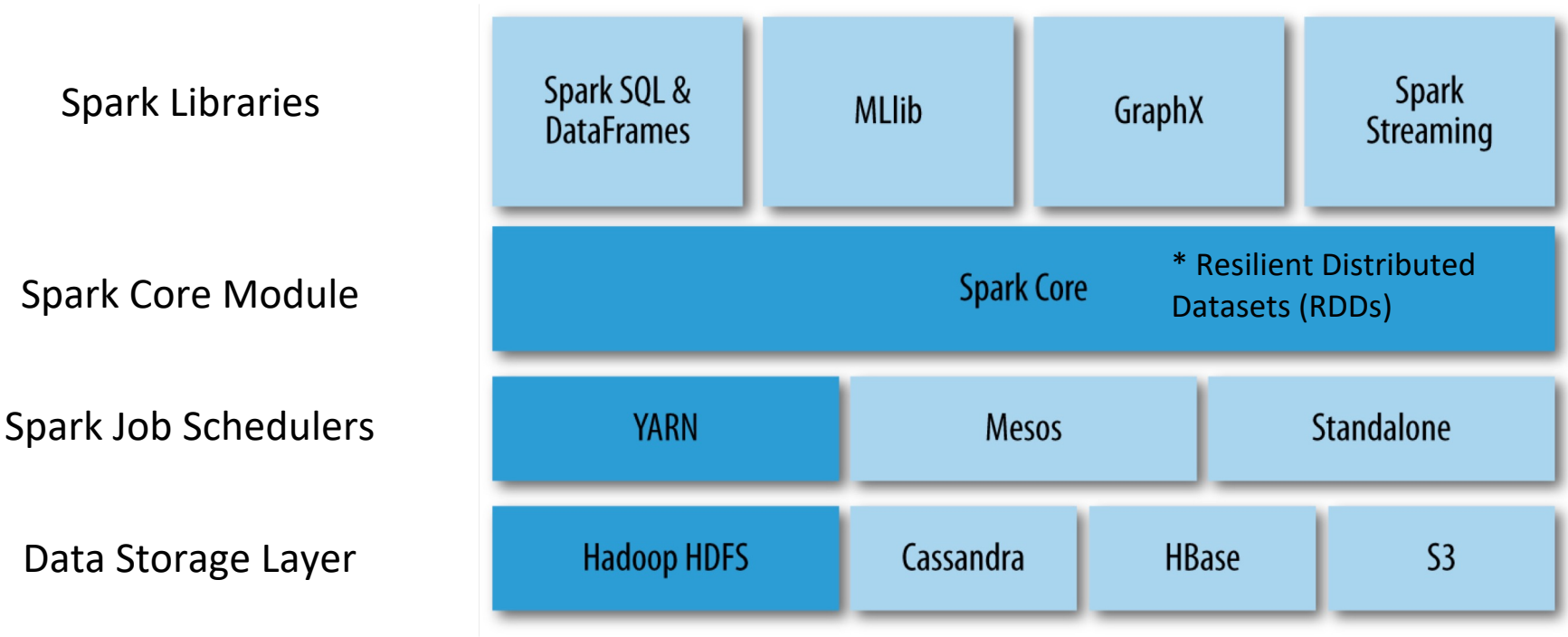
Partitions

- Data are split into multiple partitions by hashing function
- Ensure the partitions are balanced
- Common size of a partition is 64MB, common number of partitions is 2 or 3 times of the #workers
  // many workers at Vocareum
- Less partition sometimes may lead to better performance because of the cost of "partition"
- Repartition(func, num) is a function to use

Shuffling

- Handle shuffling carefully to speed up you program
- Data are essentially re-partitioned during shuffling
  - e.g., Sort by key

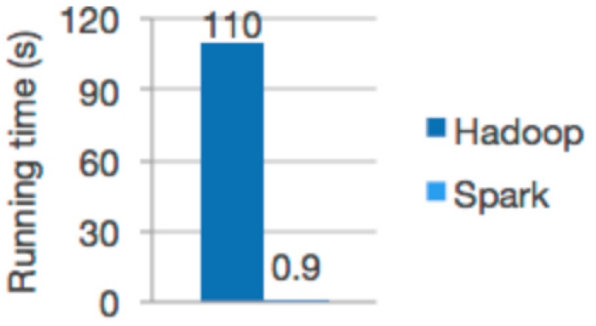
Spark Stack


Spark Libraries
Spark SQL & DataFrames
MLlib
GraphX
Spark Streaming
Spark Core Module
Spark Core
* Resilient Distributed Datasets (RDDs)
Spark Job Schedulers
YARN
Mesos
Standalone
Data Storage Layer
Hadoop HDFS
Cassandra
HBase
S3

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



Logistic regression in Hadoop and Spark

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

[1]

https://spark.apache.org/

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.

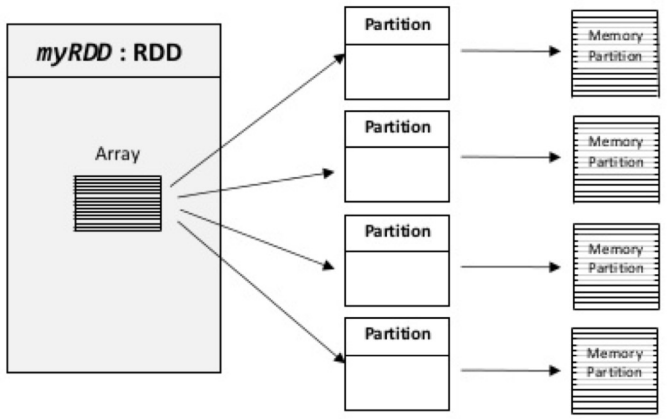
- **Generality**
  - Combine SQL, streaming, and complex analytics

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 are fault-tolerant, parallel data structures that let users
  - Explicitly persist intermediate results in memory
  - Control their partitioning to optimize data placement
  - Manipulate them using a rich set of operators

- RDDs seem a lot like Scala collections
  - RDD[T] and List[T]



How to Create an RDD?

- RDDs can be created in two ways:
  - Transforming from an existing RDD
    - E.g., a call on map on an RDD returns a new RDD
  - From a SparkContext (or SparkSession) object

Start a new Spark job:

```
val spark = new SparkConf()
  .setAppName("WordCount")
  .setMaster("local[2]")
```

The SparkContext object can be thought as your handle to the Spark cluster. It represents the connection between the Spark cluster and your running application

- **parallelize**: convert a local Scala collection to an RDD

```
val RDD = spark.sparkContext.parallelize(ArrayBuffer)
```

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

```
val text = spark.sparkContext.textFile("./text.txt")
```

Two Types of Operations on RDD

- Transformations
  - Return new RDDs as results
  - They are **lazy**, their result RDD is not immediately computed
  - E.g., turning: (left, right, left, right) => ?
- Actions
  - Compute a result based on an RDD, and returned
  - They are **eager**, their result is immediately computed.
  - E.g., turning: (left, right, left, right) => ?

Laziness / Eagerness is how we can limit network communication using the programming model

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 a 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" on RDD

- map(func)
- mapValues(func)
- filter(func)
- flatMap(func)
- reduceByKey(func, [numTasks])
- groupByKey([numTasks])
- sortByKey([asc], [numTasks])
- distinct([numTasks])
- mapPartitions(func)

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” on RDD

- getNumPartitions()
- foreachPartition(func)
- collect()
- take(n)
- count(), sum(), max(), min(), mean()
- reduce(func)
- aggregate(zeroVal, seqOp, combOp)
- countByKey()

Example

- Consider the following example:

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]
val lengthsRDD = wordsRDD.map(_.length) // RDD[Int]
```

sc -> SparkContext

What has happened on the cluster at this point?

?

Example

- Consider the following example:

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]
val lengthsRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?

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

Example

- Consider the following example:

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]
val lengthsRDD = 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

- Consider the following

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]
val lengthsRDD = wordsRDD.map(_.length) // RDD[Int]
val totalChars =lengthsRDD.reduce(_+_)
```

add an action

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(_.contain("ERROR")).take(10)
```

- The execution of *filter* is deferred until the *take* action happens
- Spark leverages this by analyzing and optimizing the chain of operations before executing it.

Spark will not compute intermediate RDDs. Instead, as soon as 10 elements of the filtered RDD have been computed, *logsWithErrors* is done.

Spark saves time and space to compute elements of the unused result of *filter*.

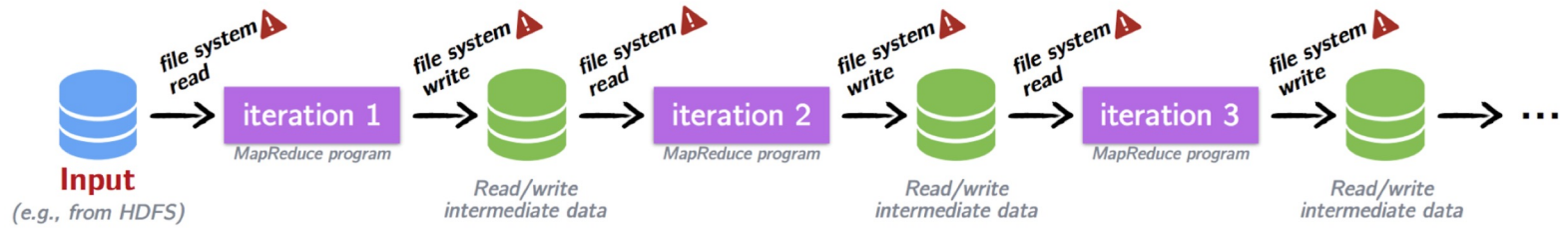
Why Spark is Good for Data Sci

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

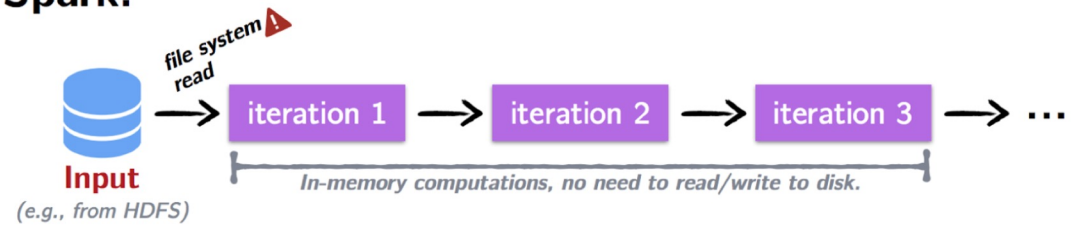
Iteration

- Most data science problems **involve**

Iteration in Hadoop:

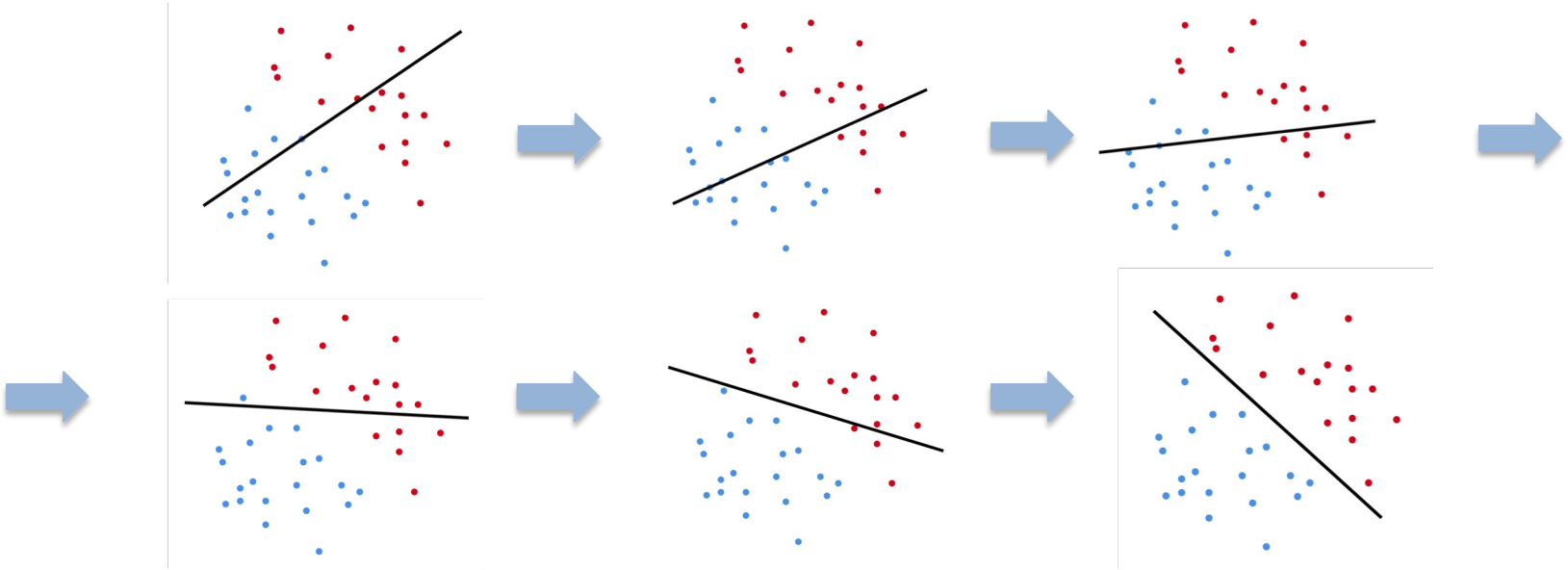


Iteration in Spark:



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

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

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

```
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.

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”

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.

```
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.

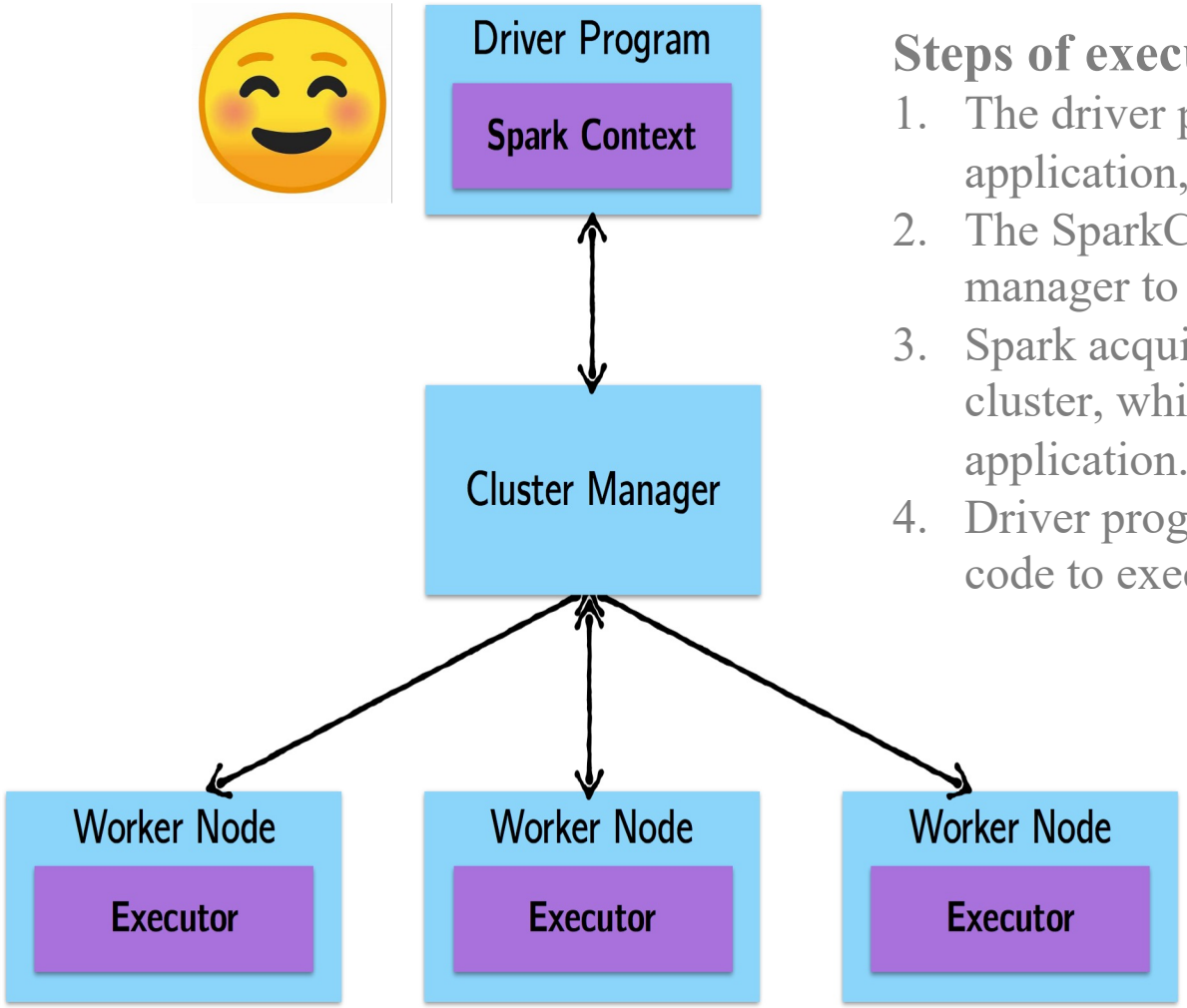
Why Use Spark Operations?

- Technically, one can implement in Python most of operations by map/flatMap by embedding the statement into the input func in Python

- Spark is implemented by Java, runs in jvm, so it is faster than Python

- Spark can analyze and optimize the computing process

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 Topology

- A simple example with *println*

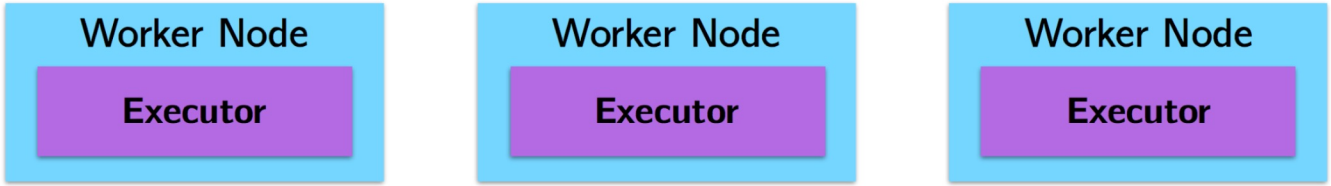
```
case class Person(name: String, age: Int)

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

On the driver: Nothing.

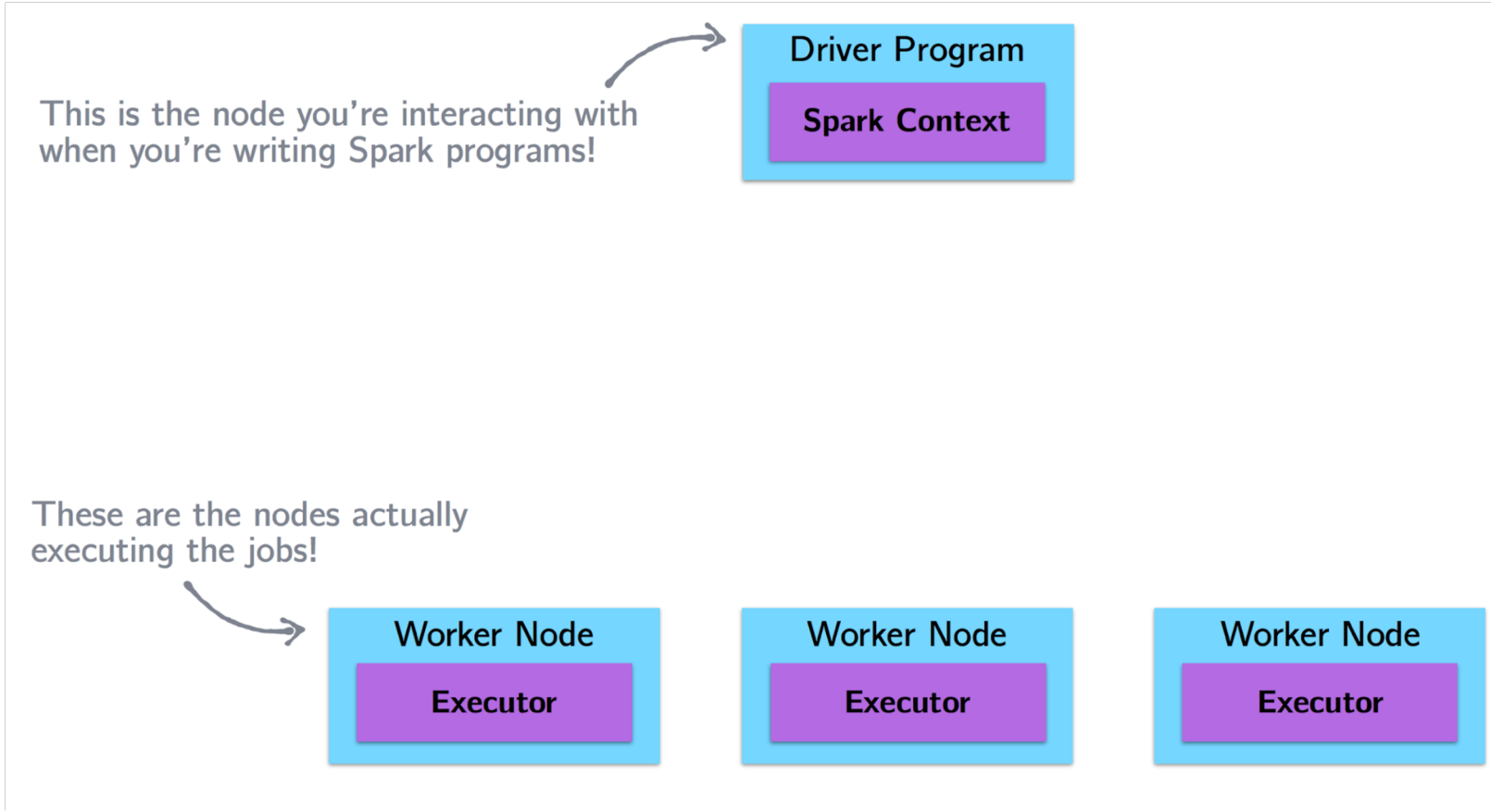
Why? Recall that foreach is **an action**, with **returns type Unit**. Therefore, it will be eagerly executed on the executors. Thus, any calls to *println* are happening on the worker nodes and are not visible in the drive node.

How Spark jobs are Executed?

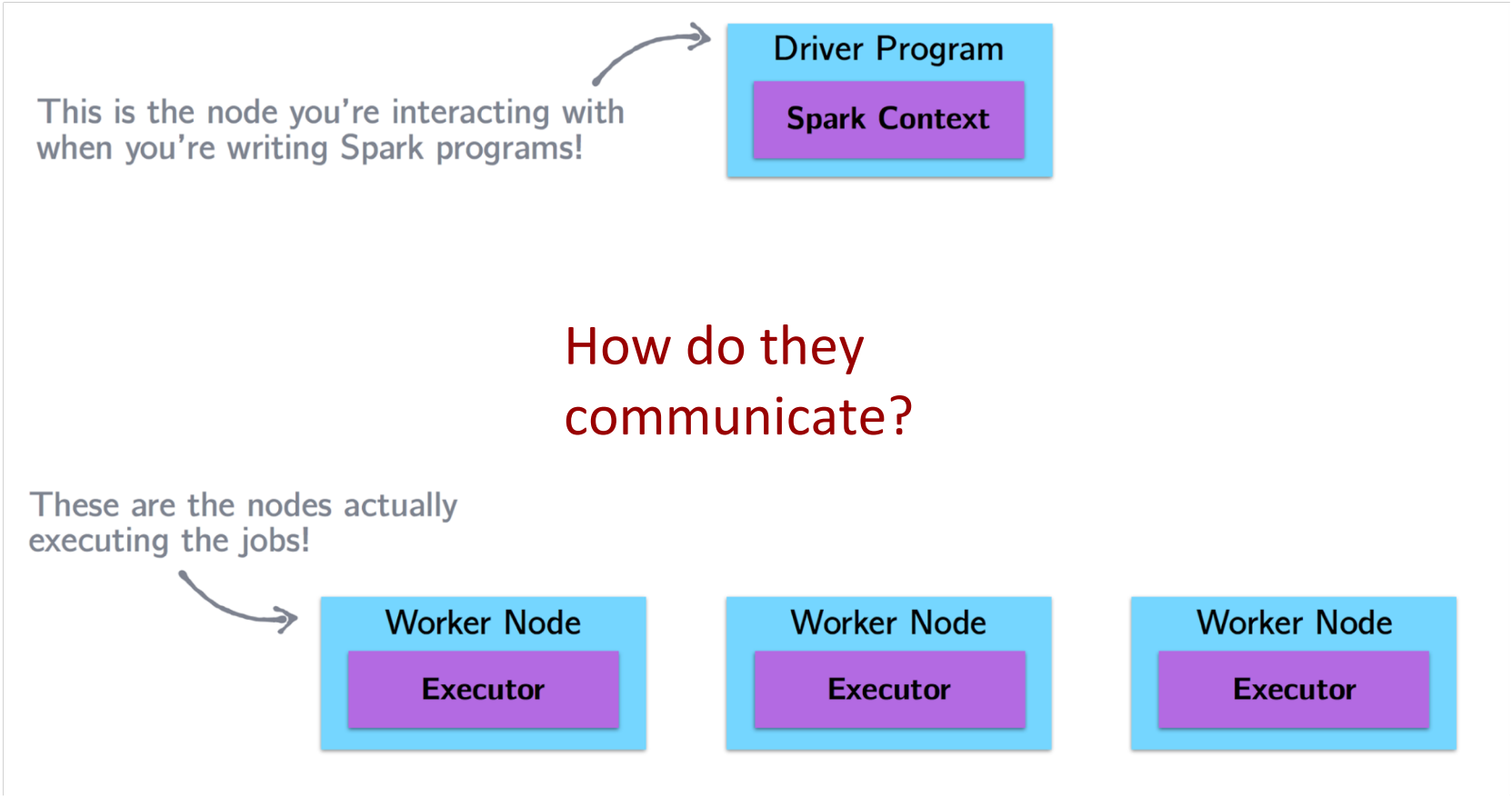
In the context of a Spark program



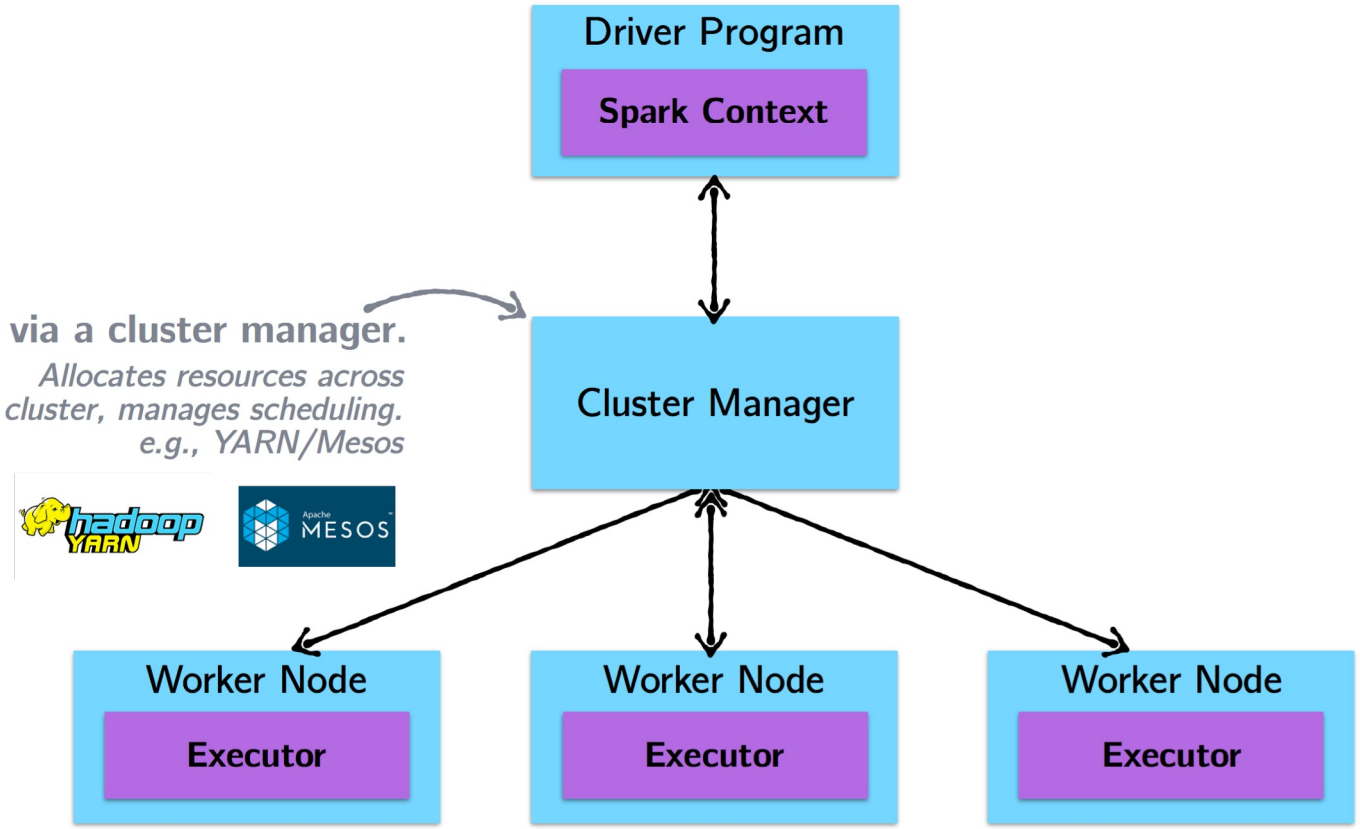
How Spark jobs are Executed



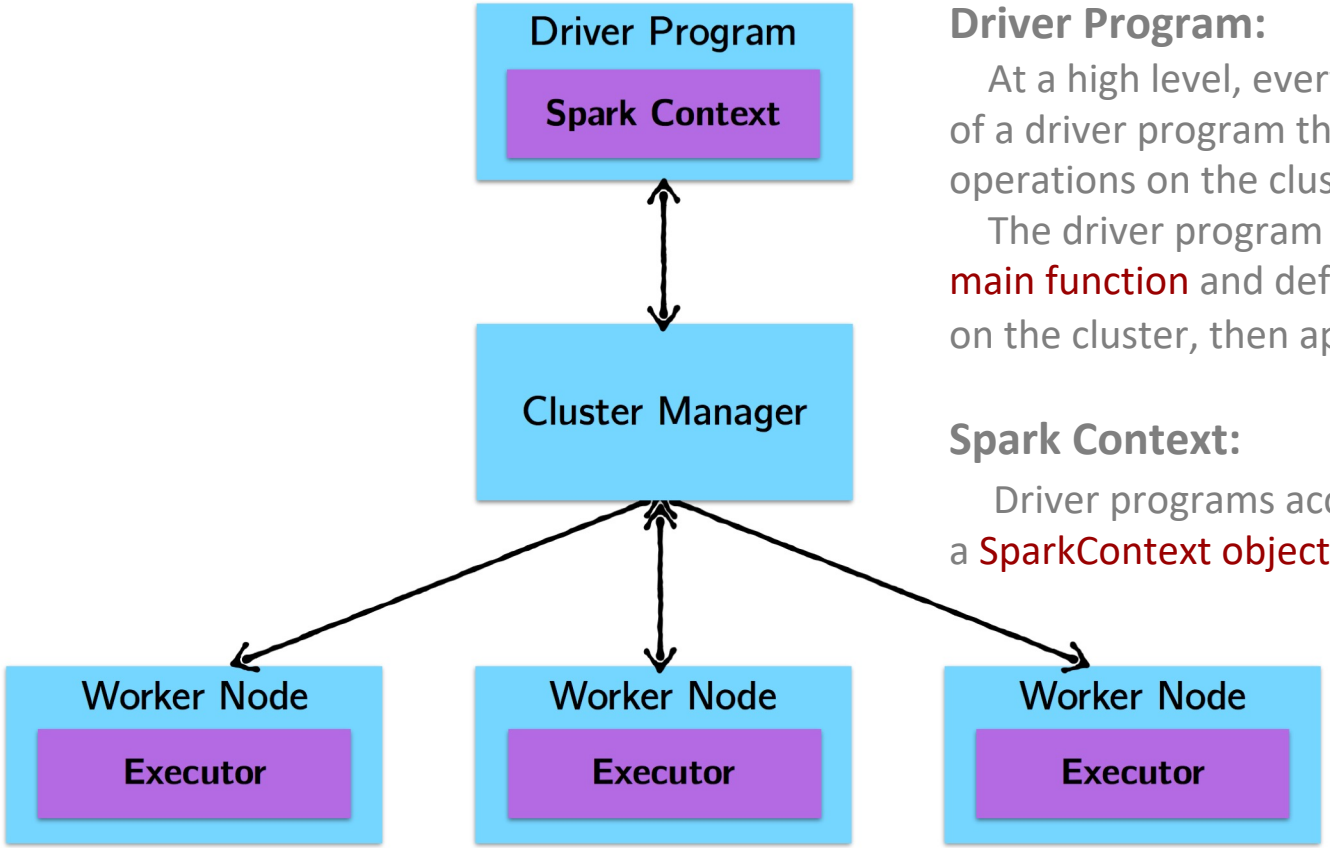
How Spark jobs are Executed



How Spark jobs are Executed



How Spark jobs are Executed



Driver Program:

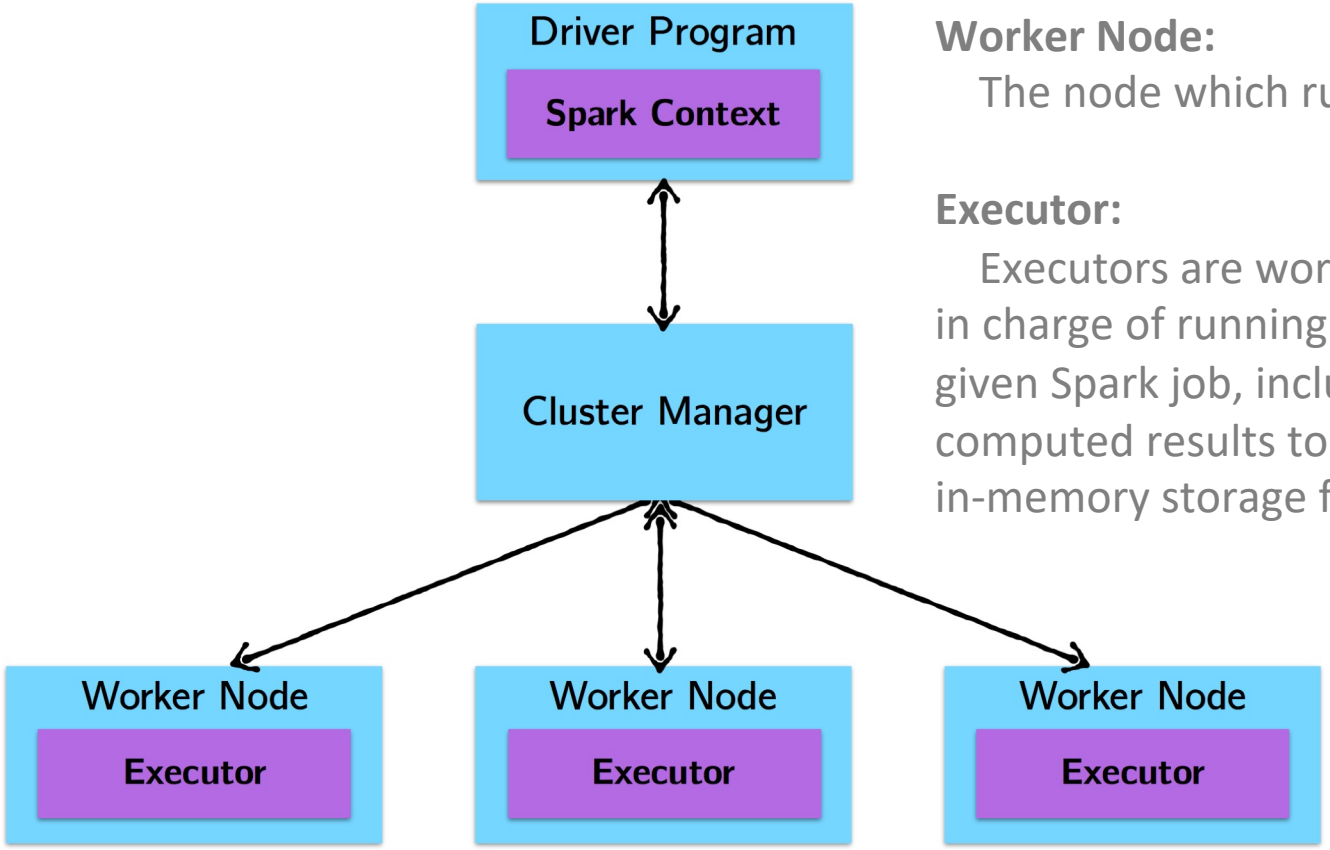
At a high level, every Spark application consists of a driver program that launches various parallel operations on the cluster.

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

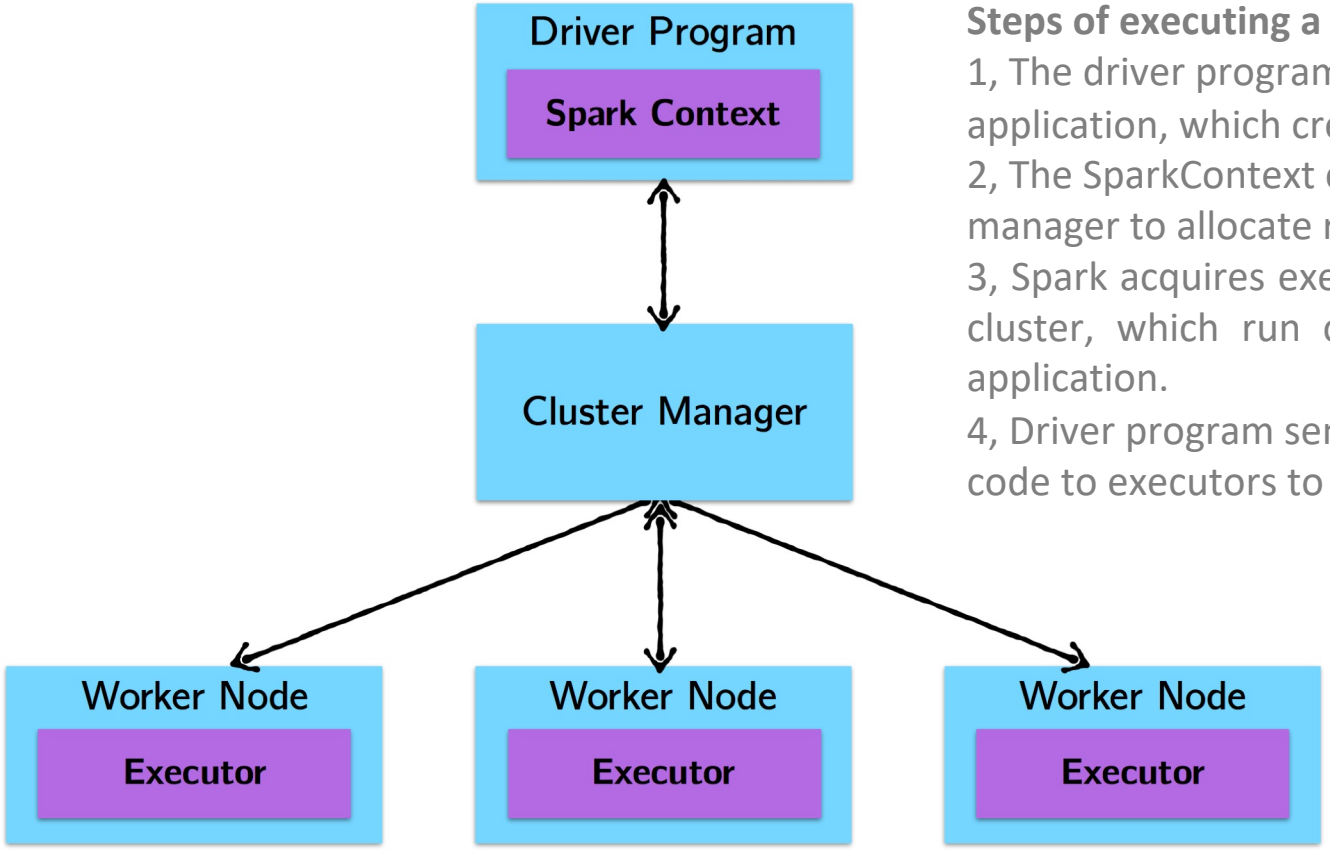
How Spark jobs are Executed



Worker Node:
The node which run applications in a cluster

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 send 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)
```

On the driver: Nothing.

Why? Recall that foreach is **an action**, with return type Unit. Therefore, it is eagerly executed on the executors, not the driver. 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.

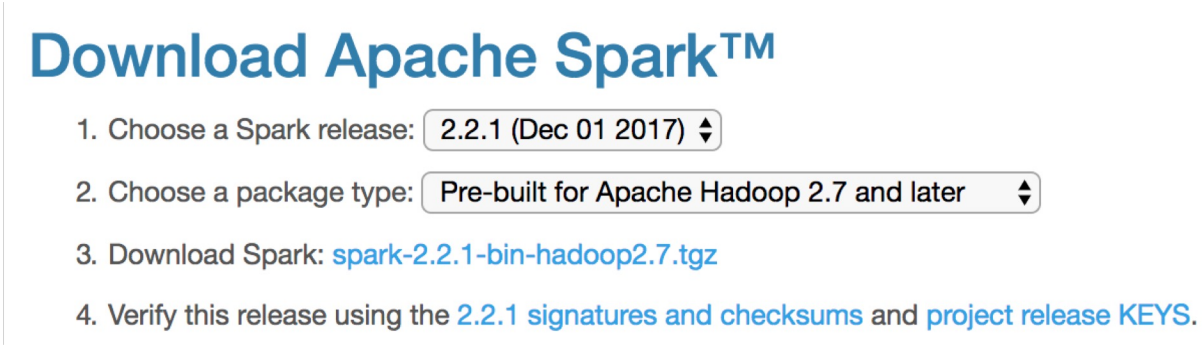
In general, executing an action involves communication between worker nodes and the node running the driver program.

Install Spark

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

  Please search their archive for 3.1.2.

  https://spark.apache.org/releases/spark-release-3-1-2.html



Note: on Vocareum: spark-submit version was 2.4.4. in 2021 Spring, and it may be subject to upgrade to the latest 3.1.2. in 2021 Fall. 2022-2023: We use 3.1.2 that is the latest on Vocareum.com

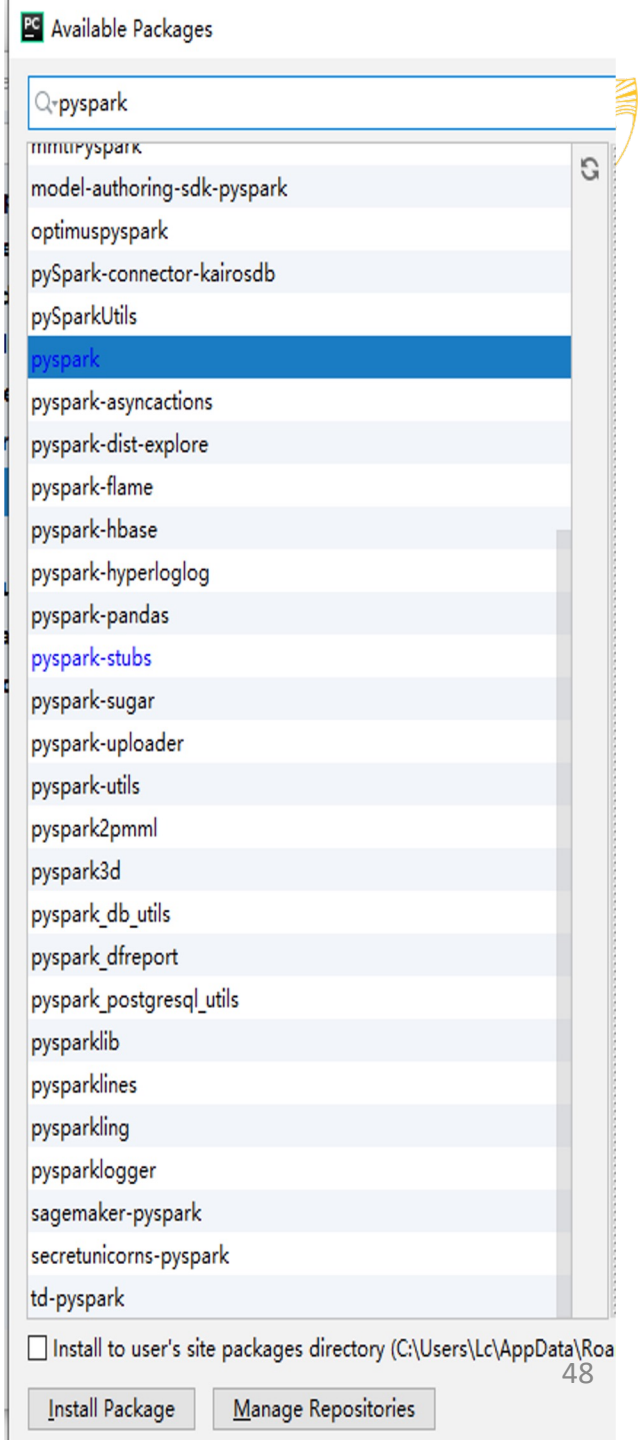
Pyspark

- Pycharm is recommended
- You can use pip install pyspark
- Or config in PyCharm:

- click[Run] -> [Edit Configurations] -> Add[Environment variables]

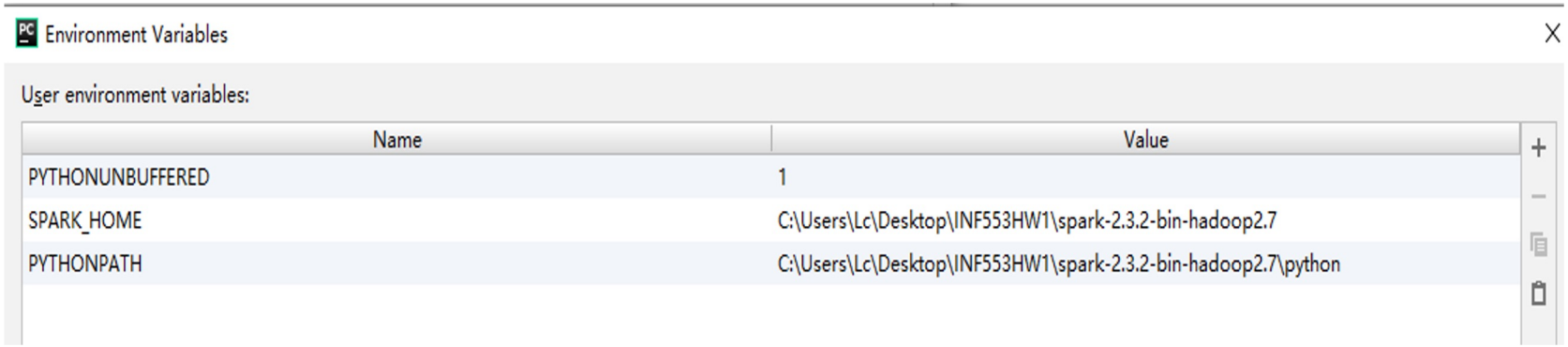
- Install pyspark package

- Make sure keep the same python version for driver and worker



Set environment variables

- SPARK_HOME
  - The root address of your downloaded spark folder
- PYTHONPATH
  - The address of "python" folder under your SPARK_HOME address



Sample Code: *word_count.py*

word_count.py | text.txt

```
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)
17
```

HW0 on Vocareum

Assignment 0 Released

Hello everyone,

We have just published Assignment-0 on Vocareum. This Assignment 0 is not graded. You can try running Wordcount program on Vocareum and get yourself acquainted with Vocareum.

If you are trying to run a script on Vocareum, please do the following two things.

1. Set the python version as python3.6

```
export PYSPARK_PYTHON=python3.6
```

2. Please select JDK 8 by running the command:

export JAVA_HOME=/usr/lib/jvm/java-1.8.0-openjdk-amd64

3. Use the latest Spark

```
/opt/spark/spark-3.1.2-bin-hadoop3.2/bin/spark-submit
```

4. To run your python code with Spark :-

/opt/spark/spark-3.1.2-bin-hadoop3.2/bin/spark-submit --executor-memory 4G --driver-memory 4G <sample.py>

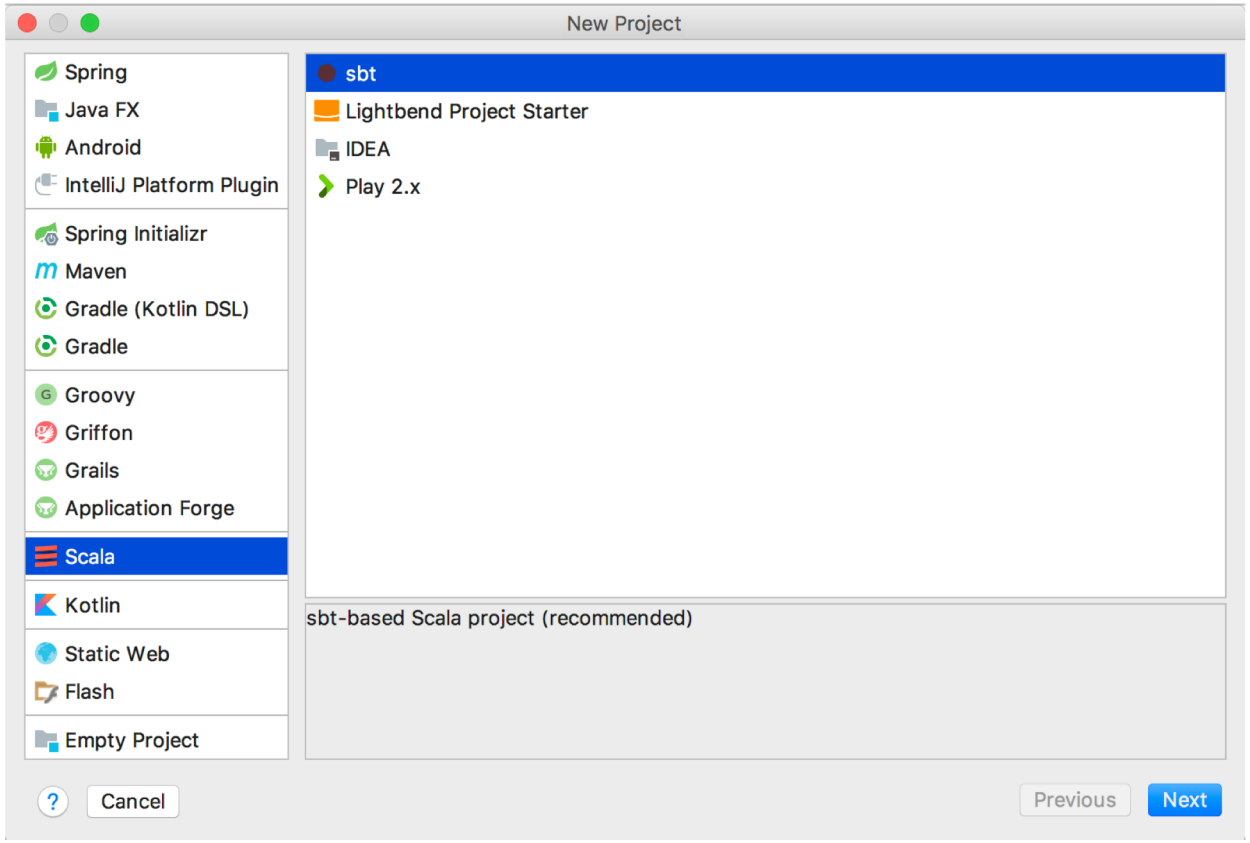
Upload your “word_count.py” and any big text file “text.txt” onto Vocareum HW0, then try them on the terminal window.

Install Scala

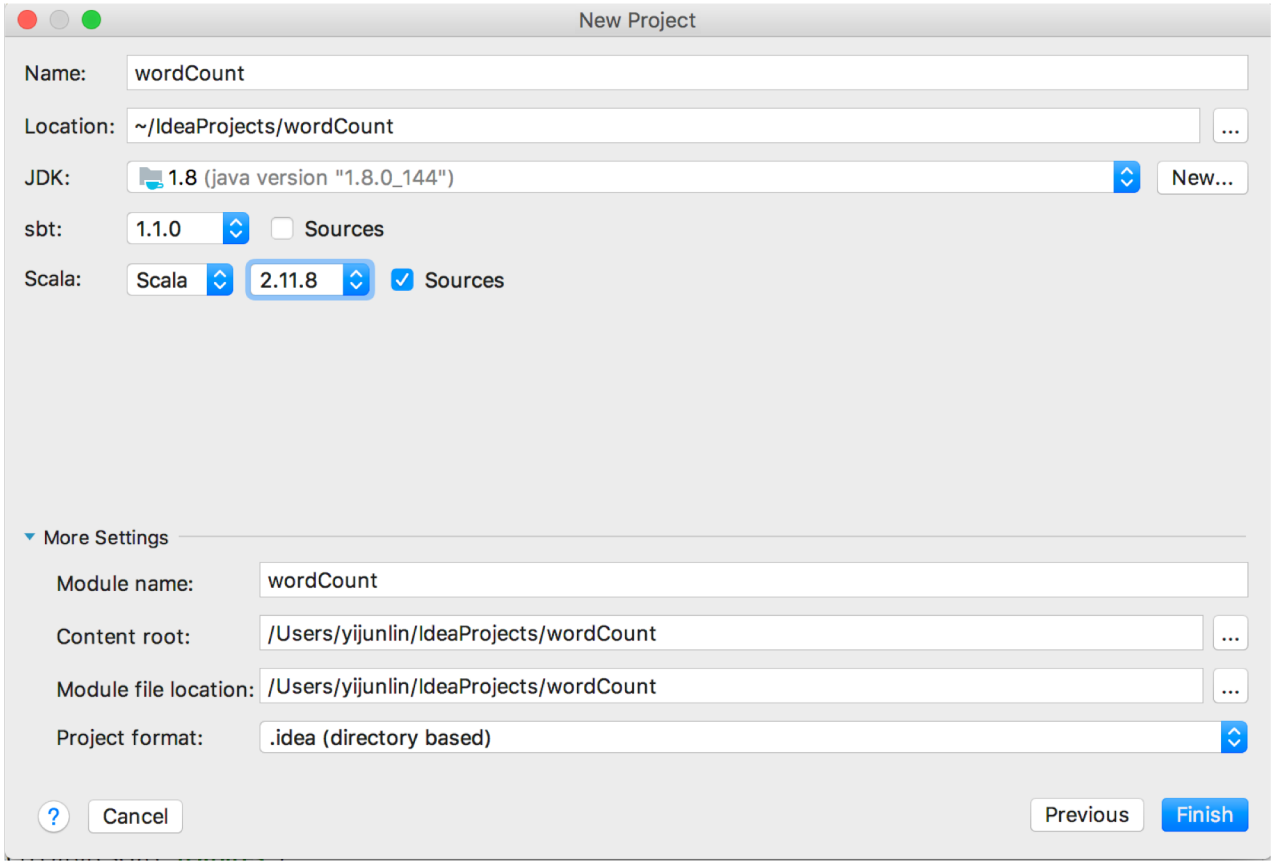
- IntelliJ IDEA, the compiler:
  - https://www.jetbrains.com/idea/#chooseYourEdition

- Install Scala plugin in the compiler
  - Open *Preference* -> Choose [*Plugins]* -> Click [*Install JetBrains plugin] (at bottom)* -> Search *Scala* and Install

Create an SBT project

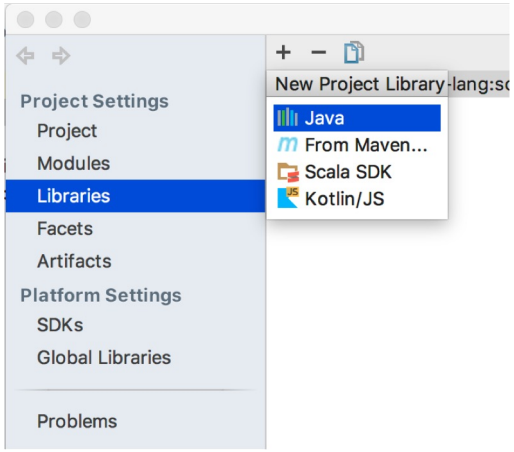


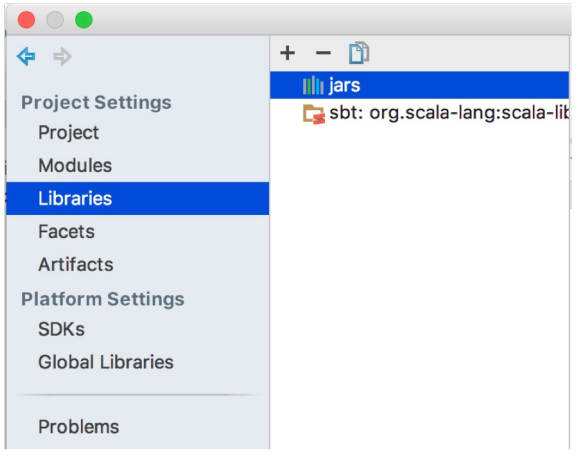
Create an SBT project



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





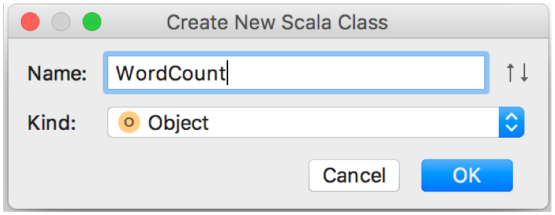
Add Spark Environment

- You can add Spark either in External Libraries or through build.sbt
  - build.sbt

build.sbt

```
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

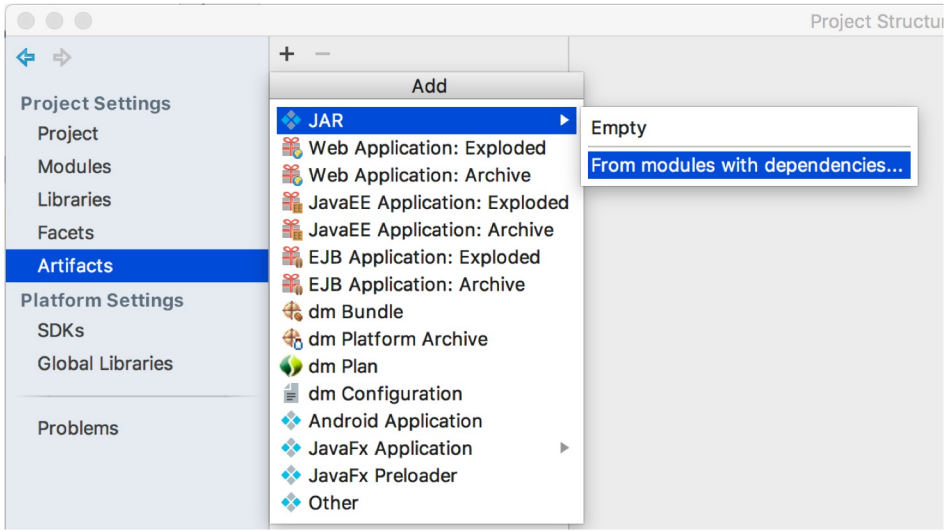
WordCount.scala

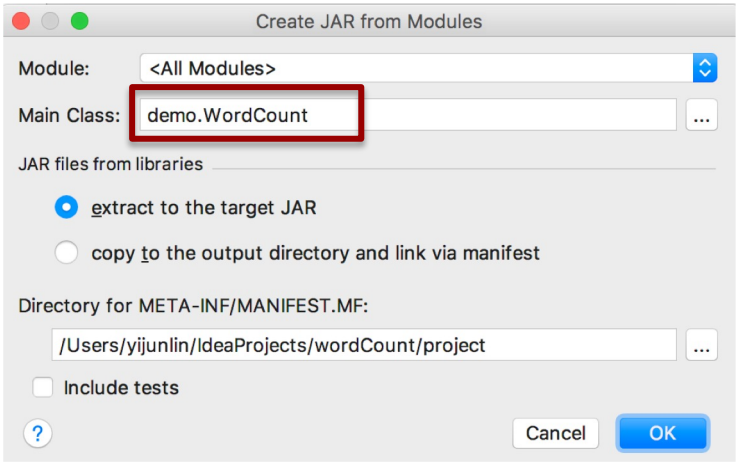
```scala
package demo

import org.apache.spark.{SparkConf, SparkContext}


object WordCount {

    def main(arg: Array[String]): Unit = {

      val spark = new SparkConf().setAppName("WordCount").setMaster("local[2]")
      val sc = new SparkContext(spark)
      val text = sc.textFile("/Users/yijunlin/IdeaProjects/wordCount/text.txt")
      val counts = text.flatMap(line => line.split(" ")).map(word=>(word, 1)).reduceByKey(_+_)

      counts.foreach(println)
    }
}
```

local[2]: start SparkContext with 2 cores
local[*], start SparkContext with all available cores

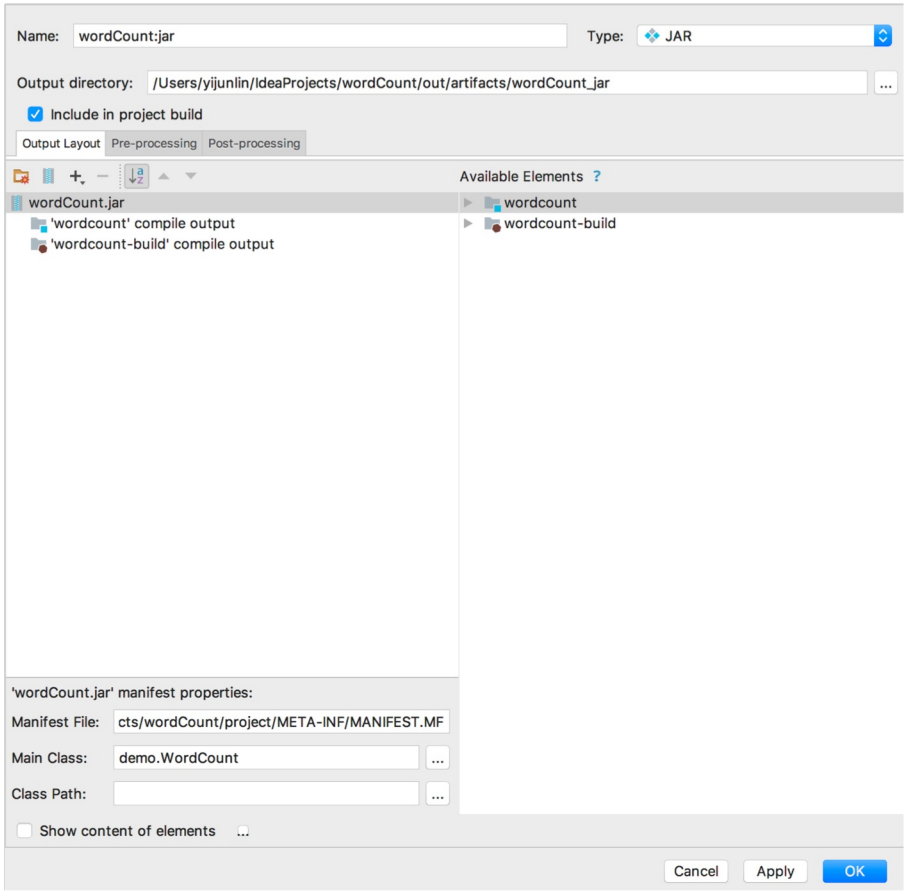
Build jar

- 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 Artifects] -> [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
```

// On Vocareum's terminal:
% bin/spark-submit –class wordcount ./****/wordCount.jar
// please make sure you call version 3.1.2. (its location may change)

Show the Result:

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

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*

- Please read the complete list of operations:

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

- Please read Spark Programming Guide:

https://spark.apache.org/docs/latest/

Attachments

Common Actions

- getNumPartitions()
- foreachPartition(func)
- collect()
- take(n)
- count(), sum(), max(), min(), mean()
- reduce(func)
- aggregate(zeroVal, seqOp, combOp)
- countByKey()

foreachPartition(func)

- What are in each partition?

Iterator for the list of elements in the partition

- def printf(iterator):
  par = list(iterator)
  print 'partition:', par

- sc.parallelize([1, 2, 3, 4, 5], 2).foreachPartition(printf)

=> partition: [3, 4, 5]
partition: [1, 2]

collect()

- Show the entire content of an RDD
- sc.parallelize([1, 2, 3, 4, 5], 2).collect()
- collect()
  - Fetch the entire RDD as a Python list
  - RDD may be partitioned among multiple nodes
  - collect() brings all partitions to the client's node
- Problem:
  - may run out of memory when the data set is large

take(n)

- take(n): collect first n elements from an RDD

- l = [1,2,3,4,5]
- rdd = sc.parallelize(l, 2)
- rdd.take(3)

=>

[1,2,3]

count()

- Return the number of elements in the dataset
  - It first counts in each partition
  - Then sum them up in the client

- l = [1,2,3,4,5]
- rdd = sc.parallelize(l, 2)
- rdd.count()

=> 5

reduce(func)

- Use func to aggregate the elements in RDD
- func(a,b):
  - Takes two input arguments, e.g., a and b
  - Outputs a value, e.g., a + b
- func should be commutative and associative
  - Applied to each partition (like a combiner)

reduce(func)

- func is continually applied to elements in RDD
  - [1, 2, 3]
  - First, compute func(1, 2) => x
  - Then, compute func(x, 3)
- If RDD has only one element x, it outputs x
- Similar to reduce() in Python

Exa mple: finding la rgest integers

- data = [S,4, 4, 1, 2, 3, 3, 1, 2, 5,4, S]
- pdata = sc.parallelize(data)

- pdata .reduce(lambda x,y: max(x,y))

  => 5

aggregate(zeroValue, seqOp, combOp)

But note reduce here is different from that in Python:
zeroValue can have different type than values in p

- For each partition p (values in the partition),
  - "reduce"(seqOp, p, zeroValue)
  - Note if p is empty, it will return zeroValue

- For a list of values, vals, from all partitions, execute:
  - reduce(combOp, vals, zeroValue)

Exa mple

- data = sc.parallelize([1],2)
- data.foreachPa rtition(printf)
  - Pl: []
  - P2: [1]

- data.aggregate(1, add,add)
  - Pl => [1] => after reduction => 1
  - P2 => [1] + [1] = [1,1] => 2
  - fina l: [1] + [1,2] => [1,1,2] => 4

Exa mple

- data.aggregate(2, add, lambda U,v: U * v)
  - Pl => 2
  - P2 => 3
  - Final: [2] + [2,3] => 2 * 2 * 3 = 12

    (w here [2] is zeroVa lue,[2,3] is the list of va lues from partitions)

Common Transformations

- map(func)
- mapValues(func)
- filter(func)
- flatMap(func)
- reduceByKey(func, [numTasks])
- groupByKey([numTasks])
- sortByKey([asc], [numTasks])
- distinct([numTasks])
- mapPartitions(func)

map(func)

- map(func): Apply a function func to each element in input RDD
  - func returns a va lue (could be a list)

- Output the new RDD conta ining the transformed va lues produced by func

Exa mple

- lines = sc.textFile("hello.txt ")

- lineSplit = lines.map(lambda s: s.split())

  => [['hello','world'], ['hello', 'this', 'world']]

- linelengths = lines.map(lambda s: len(s))

  => [11, 16]

ma pPartitions(fu nc)

- Apply transformati on to a partition
  - input to func is an iterator (over the elements in the partition)
  - func must return an itera ble (a list or use yield to return a generator)

- Different from map(func)
  - func in map(func) applies to an element

f l atMap(func)

- flat Map(func):
  - similar to map
  - But func here **must** return a list (or generator) of elements
  - & flatMap merges these lists into a single list

- lines.flatMa p(lambda x: x.split())

=>rdd: ['hello', 'world','hello','this','world']

filter(func)

- filter(func): return a new RDD with elements of existing RDD for which func returns true
- func should be a boolean function
- lines1 = lines.filter(lambda line: "this" in line)
  ⇒ ['hello this world']
- What about: lines.filter(lambda s: len(s) > 11)?

reduceByKey()

- reduceByKey(func)
  - Input: a collection of (k, v) pairs
  - Output: a collection of (k, v') pairs
- v': aggregated value of v's in all (k, v) pairs with the same key k by applying func
- func is the aggregation function
  - Similar to func in the reduce(func, list) in Python

reduceByKey() vs. reduce()

- reduceByKey() returns an RDD
  - Reduce values per key

- reduce() returns a non-RDD value
  - Reduce all values!

groupByKey()

- groupByKey()
  - Similar to reduceByKey(func)
  - But without func & returning (k, Iterable(v)) instead

- rddp.groupByKey()

⇒[(2, <iterable>), (1, …), (3, …)]

Laziness Example

- Consider the following example:

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]    sc -> SparkContext
val lengthsRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?

Laziness Example

- Consider the following example:

```
val largeList: List[String] = ...
val wordsRDD = sc.parallelize(largeList) // RDD[String]
val lengthsRDD = wordsRDD.map(_.length) // RDD[Int]
```

What has happened on the cluster at this point?

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