Introduction to Spark and Scala

INF 553

University of Southern California

Outline

- How to write Spark program (syntax)
- Key concepts in spark
- Speed program up
- Install Spark on your machine
- Vocareum.com, use Spark on their terminal window
- Reading materials and links

Resilient Distributed Dataset (RDD)

- 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
  - You can't access the value directly in 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 partition is 2 or 3 times of the #workers
- Less partition sometimes may lead to better performance because of the cost of “partition”
- Repartition(func, num)

Shuffling

- Handle shuffling carefully to speed you program up!
- Data are essentially repartitioned during shuffling.
- E.g. sort by key

RDD Operations

- Transformations
  - They are lazy, the result is not immediately computed
- Actions
  - They are eager, the result is immediately computed

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*.

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”

Common "Transformations" on RDD

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

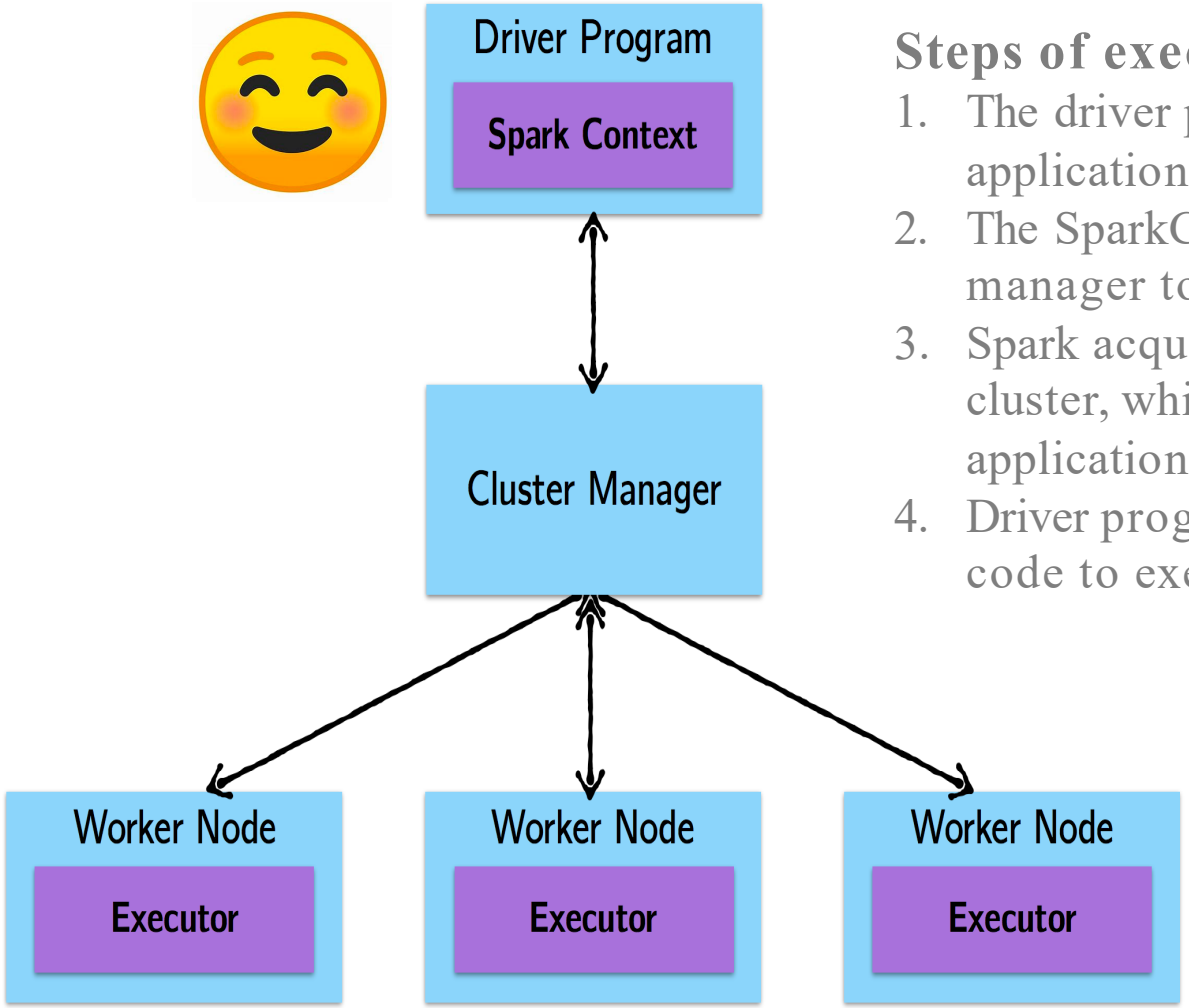
Common ”Actions” on RDD

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

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*

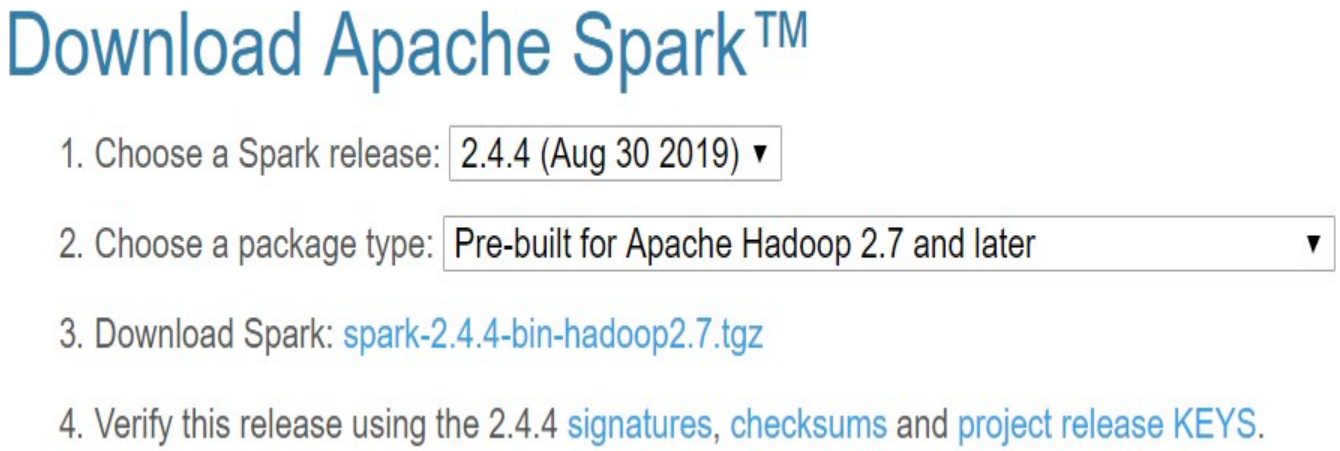
```
case class Person(name: String, age: Int)

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.

Install Spark

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



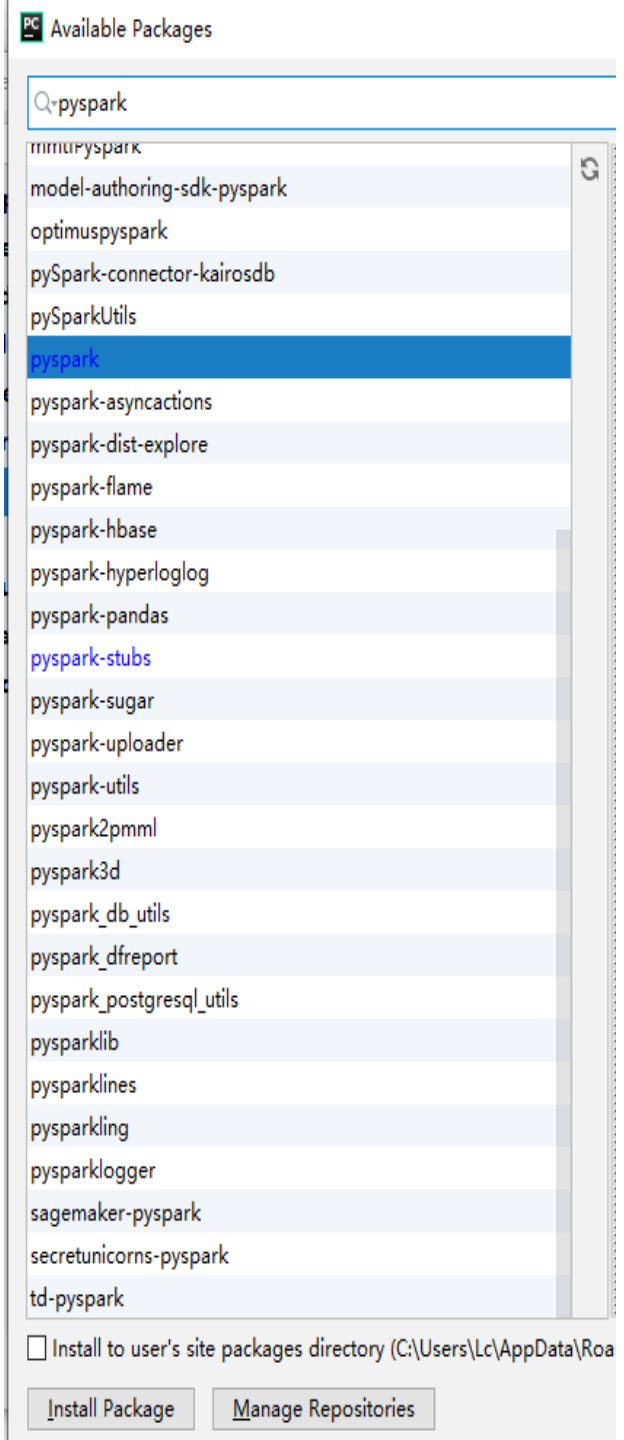
Pyspark

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

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

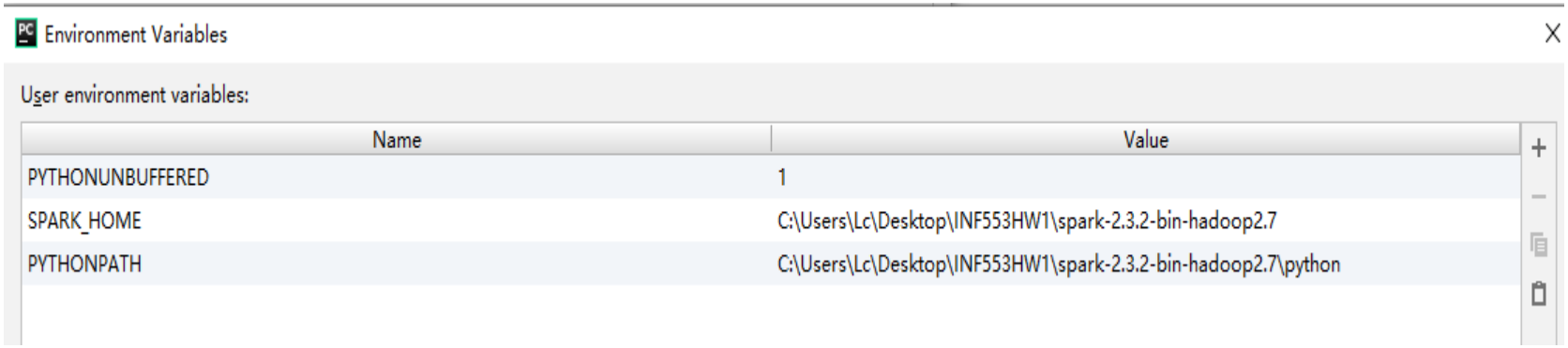
- Install pyspark package

- 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 | 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
```

Run Python on Spark command line

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

  - On the command line, you can do:
  - bin/spark-submit ./word_count.py

- Try this on a terminal window
  - On your local machine
  - On Vocareum terminal window

Run Spark on Vocareum

- Login to Vocareum.com
- Got to the terminal window, and type:

/home/local/latest/spark/bin/spark-submit ./word_count.py

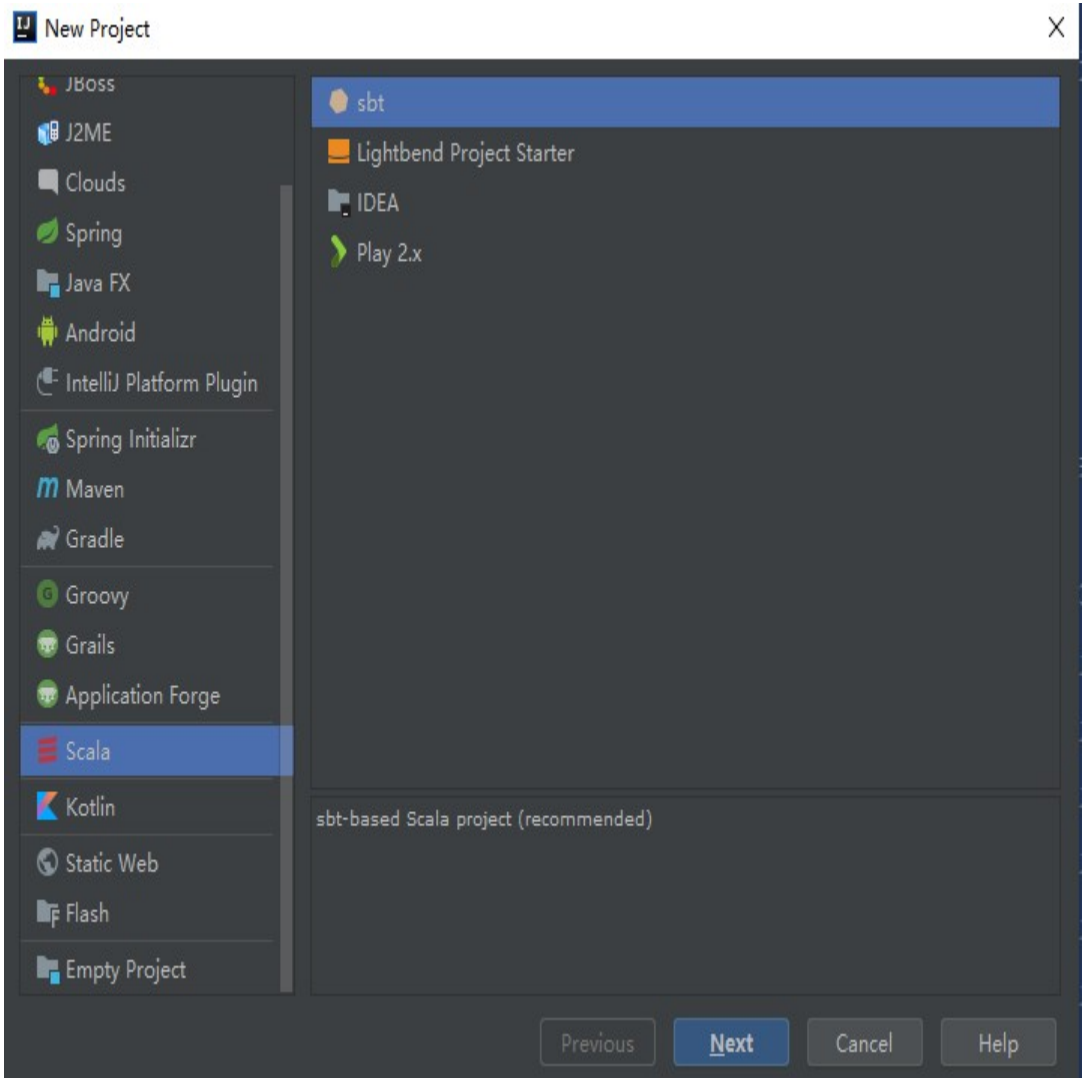
Install Scala

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

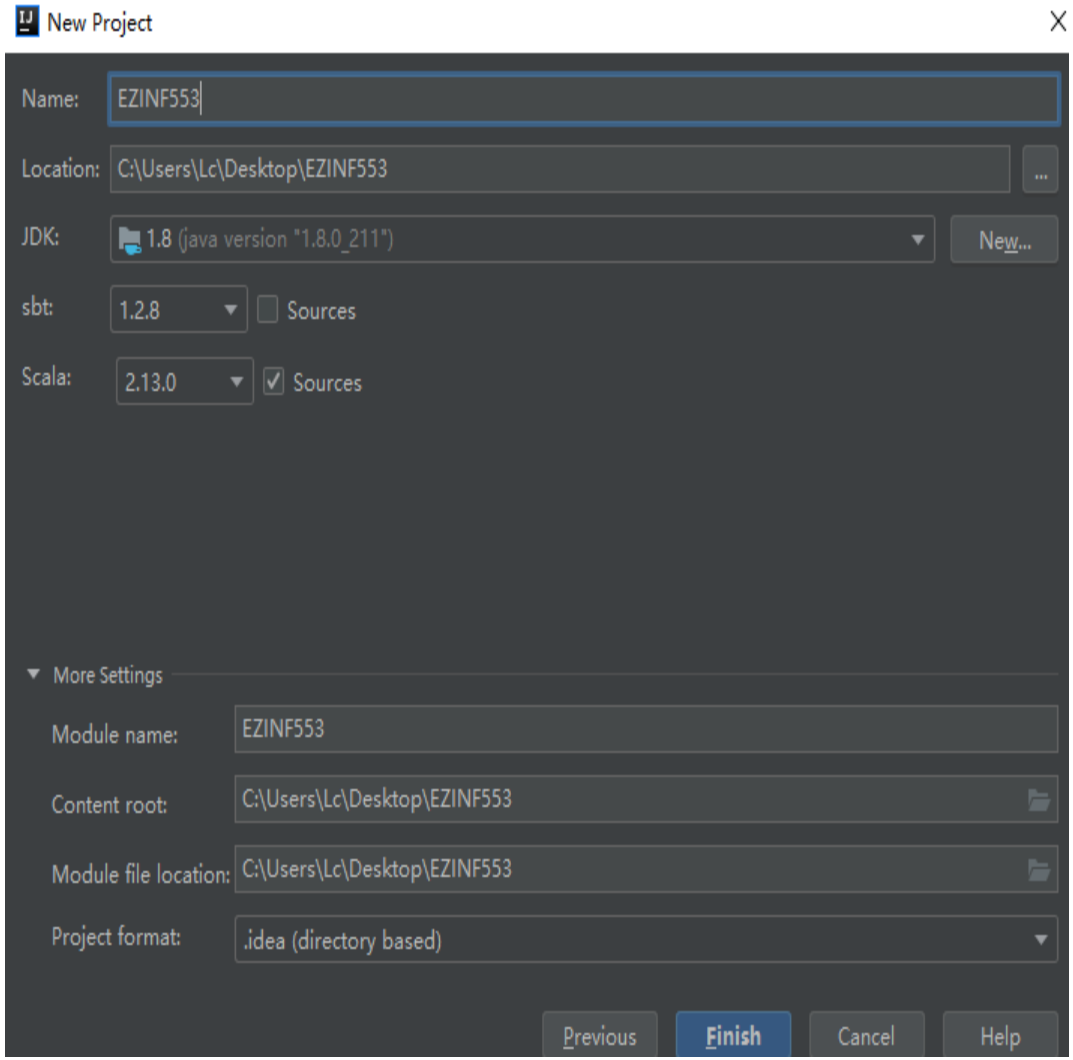
- Install Scala plugin in the compiler

  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*

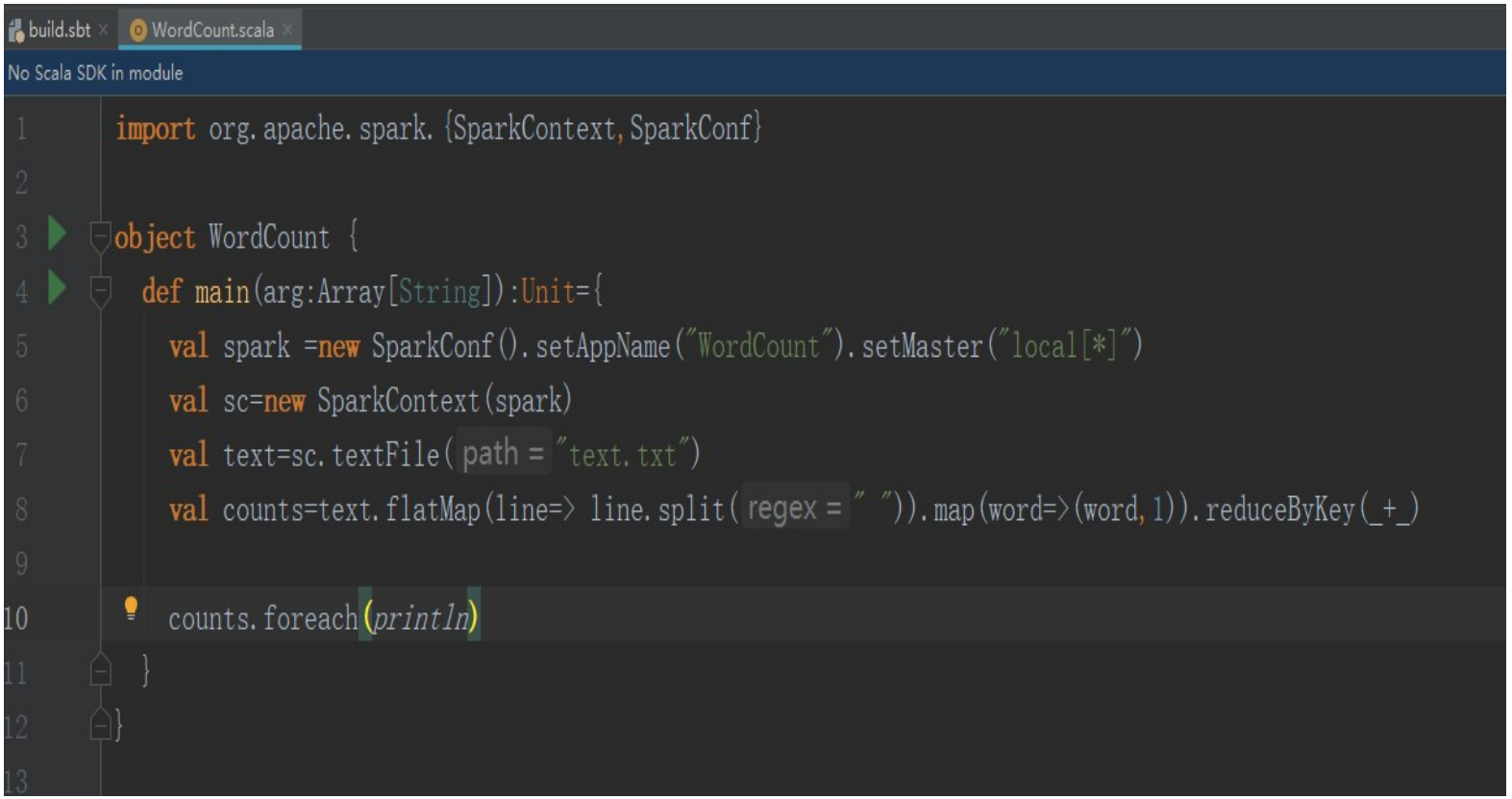
Add Spark Environment

build.sbt

```
1   name := "EZINF553"
2
3   version := "0.1"
4
5   scalaVersion := "2.13.0"
6
7   val sparkVersion = "2.4.4"
8   resolvers ++= Seq("apache-snapshots" at "http://repository.apache.org/snapshots/")
9
10
11  libraryDependencies ++= Seq(
12    "org.apache.spark" %% "spark-core" %sparkVersion,
13    "org.apache.spark" %% "spark-sql" % sparkVersion
14  )
```

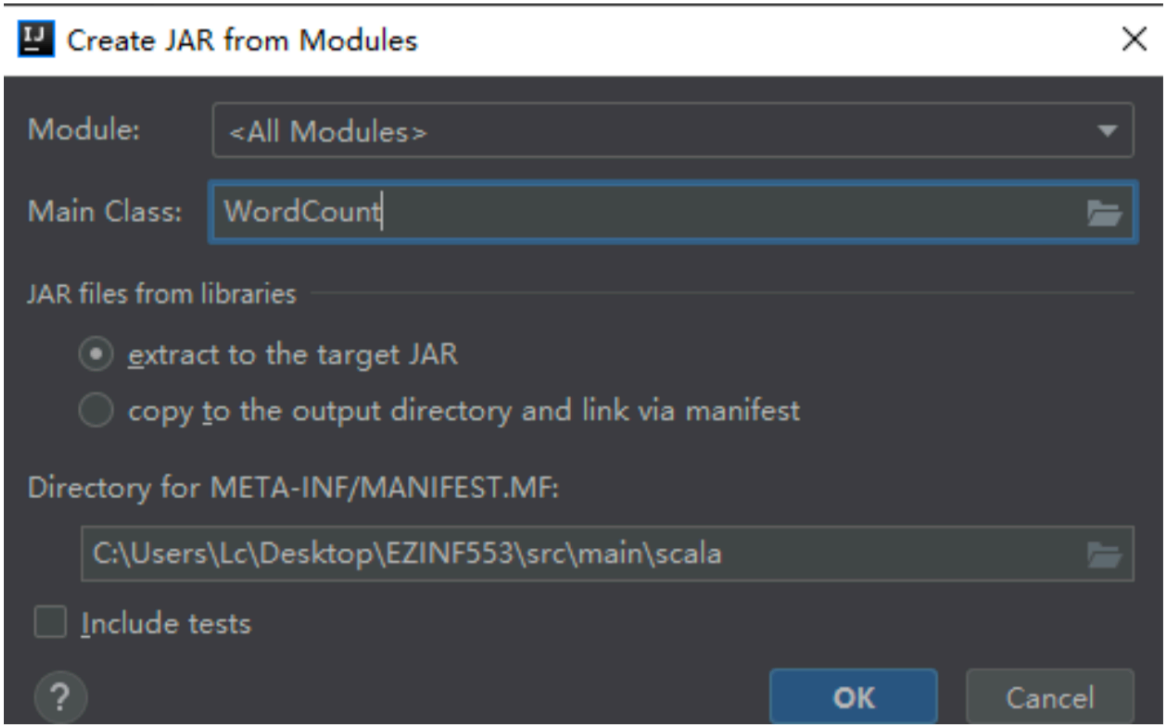
Write Scala Code

- On the Project panel on the left, expand src => main
- Right- click scala and select New => Scala Object



Build jar

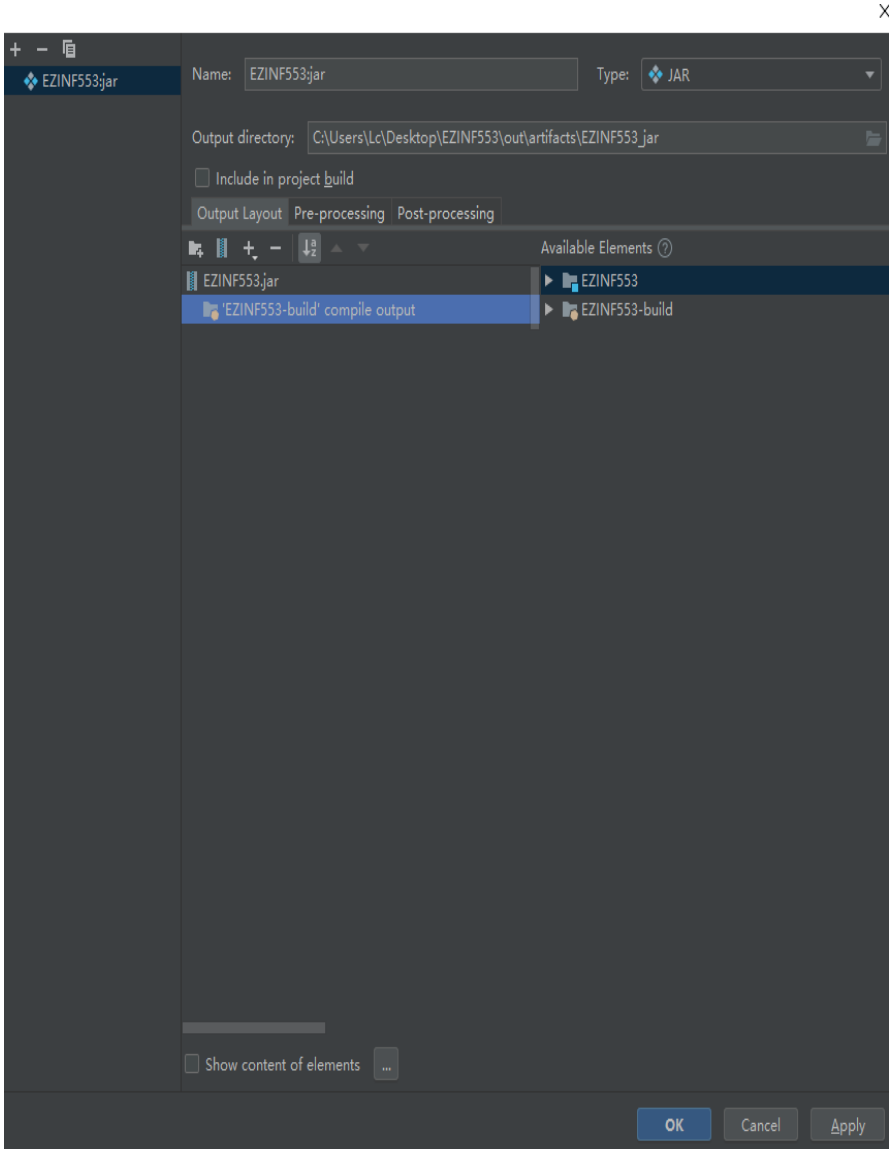
- Click [File] -> [Project Structure] -> [Artifacts] -> [+] JAR -> [From modules with dependencies] -> Put the Main Class of your code -> [OK]



Build jar

- You need to delete all the spark libraries or other unrelated libraries that you would not use in your program, otherwise your jar will be huge. Click[OK]

- Click [Build] -> [Build Artifacts] -> [Build] -> produce the package out, the jar file is in it!



Run jar on Spark

- bin/spark- submit –class wordcount ./****/wordcount.jar

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)
```

Links

- 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?

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

Iterator for the list of elements in the partition

- 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

Example: finding largest integers

- data = [5, 4, 4, 1, 2, 3, 3, 1, 2, 5, 4, 5]
- 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)

Example

- data = sc.parallelize([1], 2)
- data.foreachPartition(printf)
  - P1: []
  - P2: [1]

- data.aggregate(1, add, add)
  - P1 => [1] => after reduction => 1
  - P2 => [1] + [1] = [1, 1] => 2
  - final: [1] + [1, 2] => [1, 1, 2] => 4

Example

- data.aggregate(2, add, lambda U, v: U * v)
  - P1 => 2
  - P2 => 3
  - Final: [2] + [2, 3] => 2 * 2 * 3 = 12

    (where [2] is zeroValue, [2,3] is the list of values 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 value (could be a list)

- Output the new RDD containing the transformed values produced by func

Example

- 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]

mapPartitions(func)

- Apply transformation to a partition
  - input to func is an iterator (over the elements in the partition)
  - func must return an iterable (a list or use yield to return a generator)

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

flatMap(func)

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

- lines.flatMap(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)
  $\Rightarrow$ ['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.