R·I·T



PROF741 Lab: Installation Scala and Apache Spark on MacOS

Introduction

Spark Lab is the Lab for the course PROF741.

Lab Goals

1. To be familiar with Scala Functional Programming Language.
2. Practical demo of RDD operations using spark context.

Lab Summary

1. Get Homebrew
2. Install x-code-select
3. Install Spark, Scala
4. Start Spark Shell
5. Setup the $PATH environment variables for easy access to spark (optional)

Activity 1 – Get Homebrew

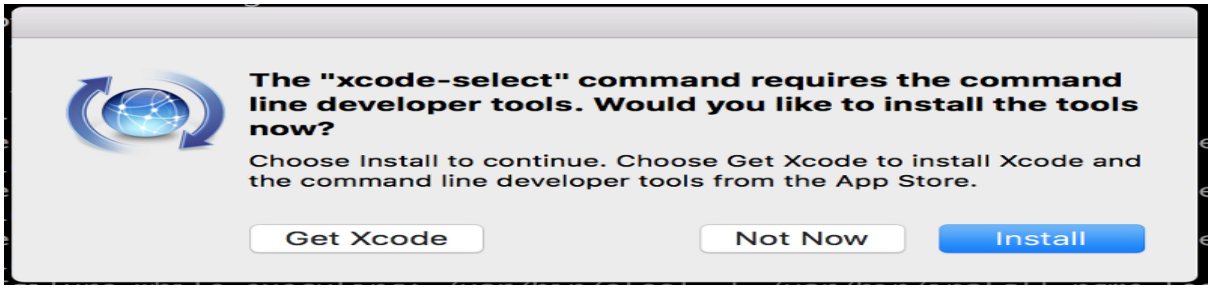
Homebrew makes your work more easier when it comes to installing applications and languages on a Mac OS. You can get Homebrew by following the instructions on it's website.

Activity 2 – Install x-code select

In order to install Java, Scala, and Spark through the command line we will probably need to install xcode-select and command line developer tools. Go to your terminal and type:

xcode-select --install

You will get a prompt that looks something like this:



Go ahead and select install.

Activity 3: Use Homebrew to install Java (if you don't already installed, otherwise skip this step)

Scala is dependent on Java, you may or may not need to install it. The easiest way to install it is to just use HomeBrew:

In your terminal type:

brew install - - cask adoptopenjdk or

brew cask install java

You may need to enter your password at some point to complete the java installation. After running this Homebrew should have taken care of the Java install. Now we can move on to Scala.

Activity 4: Use Homebrew to install Scala

Now with Homebrew installed go to your terminal and type:

brew install scala

Activity 5: Use Homebrew to install Apache Spark

Now with Scala installed go to your terminal and type:

brew install apache-spark

Homebrew will now download and install Apache Spark, it may take some time depending on your internet connection.

Activity 6: Start the Spark Shell

Now try this command:

spark-shell

You should see a flood of text and warnings but eventually see something like this:



scala>

You can confirm that it is working by typing the scala code:



Congratulations! You're all set up!

Part 2: RDD operations

Two operations can be applied in RDD. One is the transformation and another one is action.



Transformations

Transformations are the processes that you perform on an RDD to get a result which is also an RDD. The example would be applying fuctions such as filter(), union(), map(), flatmap(), distinct(), reducebyKey(), and sortby() that would create an another resultant RDD. Lazy evaluation is applied in creation of RDD.

Actions

Actions return results to the driver program or write it in storage and kick off a computation. Some examples are count(), first(), select(), take(), countbyKey(), collectasMap() and reduce().

Transformations will always return RDD whereas actions return some other data type.

Next Lab is a practical for RDD operations. Some of them are as follows:

1) Create RDD
2) Read a result
3) Count
4) Distinct
5) Filter
6) sortBy
7) Reduce
8) Map
9) Union, intersection, and take.

PRACTICE EXERCISES

a) Create RDD

First, let's create a RDD using parallelize() method which is the simplest method

```
scala> val rdd1 =sc.parallelize(List(23,45,67,86,78,27,82,45,67,86))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[3] at parallelize at <console>:23
```

Here sc. Denotes SparkContext and each element is copied to form RDD.

b) Read result
We can read the result generated by RDD by using the collect operation:

rdd1.collect

The results are shown here:

```
scala> rdd1.collect
res2: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86)
```

c) Count
The count action is used to get the total number of elements present in the particular RDD.

rdd1.count

The results are shown here:

```
scala> rdd1.count
res3: Long = 10
```

d) Distinct
Distinct is a type of transformation that is used to get the unique elements in the RDD.

rdd1.distinct.collect

The results are shown here:

```
scala> rdd1.distinct.collect
res4: Array[Int] = Array(45, 82, 86, 78, 27, 23, 67)
```

e) Filter



Filter transformation creates a new dataset by selecting the elements according to the given condition.

rdd1.filter(x=> x<50). collect

The results are shown here:

```
scala> rdd1.filter(x=> x<50).collect
res5: Array[Int] = Array(23, 45, 27, 45)
```

f) sortBy

sortBy operation is used to arrange the elements in ascending order when the condition is true and in descending order when the condition is false.

rdd1.sortBy(x=> x, true). collect

```
scala> rdd1.sortBy(x=>x, true).collect
res6: Array[Int] = Array(23, 27, 45, 45, 67, 67, 78, 82, 86, 86)
```

rdd1.sortBy(x=> x, false). collect

```
scala> rdd1.sortBy(x=>x, false).collect
res7: Array[Int] = Array(86, 86, 82, 78, 67, 67, 45, 45, 27, 23)
```

g) Reduce

Reduce action is used to summarize the RDD based on the given formula:

rdd1. reduce((x,y)=> x+y)

```
scala> rdd1.reduce((x,y)=> x+y)
res8: Int = 606
```

Here, each element is added, and the total sum is printed.

h) Map

Map transformation process each element in the RDD according to the given condition and creates a new RDD.

rdd1.map(x=>x+1).collect

```
scala> rdd1.map (x=>x+1).collect
res9: Array[Int] = Array(24, 46, 68, 87, 79, 28, 83, 46, 68, 87)
```

Here, each element is incremental once.

i) Union, intersection, and cartesian
First, we create a second rdd using parellalize method.
val rdd2= sc.parallelize(List(25,73,97,78,27,82))

```
scala> val rdd2=sc.parallelize(List(25,73,97,78,27,82))
rdd2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[19] at parallelize at <console>:23
```

Union operation combines all the elements of the given two RDDs.

```
scala> rdd1.union(rdd2).collect
res10: Array[Int] = Array(23, 45, 67, 86, 78, 27, 82, 45, 67, 86, 25, 73, 97, 78, 27, 82)
```

Intersection operation forms a new RDD by taking the common elements in the given RDDs.

```
scala> rdd1.intersection(rdd2).collect
res11: Array[Int] = Array(82, 78, 27)
```

Cartesian operation is used to create a cartesian product of the required RDDs.

```
scala> rdd1.cartesian(rdd2).collect
res12: Array[(Int, Int)] = Array((23,25), (45,25), (23,73), (23,97), (45,73), (45,97), (23,78), (45,78), (23,27), (23,82), (45,27), (45,82)
, (67,25), (86,25), (78,25), (67,73), (67,97), (86,73), (86,97), (78,73), (78,97), (67,78), (86,78), (78,78), (67,27), (67,82), (86,27), (8
6,82), (78,27), (78,82), (27,25), (82,25), (27,73), (27,97), (82,73), (82,97), (27,78), (82,78), (27,27), (27,82), (82,27), (82,82), (45,25
), (67,25), (86,25), (45,73), (45,97), (67,73), (67,97), (86,73), (86,97), (45,78), (67,78), (86,78), (45,27), (45,82), (67,27), (67,82), (
86,27), (86,82))
```

j) Take



Take action returns the first n elements in the RDD.

```
scala> rdd1.take(5)
res13: Array[Int] = Array(23, 45, 67, 86, 78)
```

Conclusion

Now, you may notice, that when you do any transformations only copies of existing RDDs are created and the initially created RDD doesn't change. This is because RDDs are immutable. This feature makes RDDs fault-tolerant and the lost data can also be recovered easily.

I hope now you have a basic idea about the RDDs and their role in Apache Spark.

END OF THE LAB