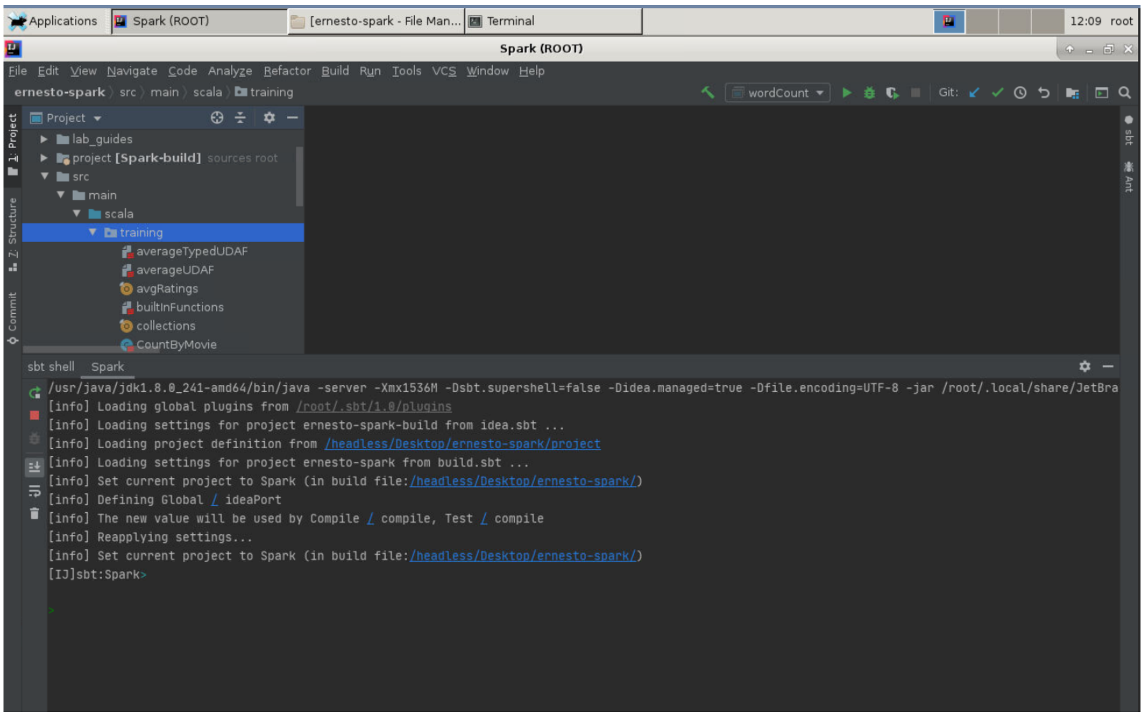
Lab : Apache Spark Accumulators Custom

Pre-reqs:

- Google Chrome (Recommended)

Note:

- The supplied commands in the next steps MUST be run from your `~/Desktop/ernesto-spark` directory.
- Final code was already cloned from github for this lab. You can just understand the application code in the next steps and run it using the instructions.
- Start IntelliJ IDE and open `~/work/ernesto-spark/src/main/scala/training/CountByMovie.scala` to view scala file.


Applications
Spark (ROOT)
[ernesto-spark - File Man...
Terminal
12:09 root
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
ernesto-spark › src › main › scala › training
wordCount
Git:
Project
lab_guides
project [Spark-build] sources root
src
main
scala
training
averageTypedUDAF
averageUDAF
avgRatings
builtInFunctions
collections
CountByMovie
sbt shell
Spark

```
/usr/java/jdk1.8.0_241-amd64/bin/java -server -Xmx1536M -Dsbt.supershell=false -Didea.managed=true -Dfile.encoding=UTF-8 -jar /root/.local/share/JetBra
[info] Loading global plugins from /root/.sbt/1.0/plugins
[info] Loading settings for project ernesto-spark-build from idea.sbt ...
[info] Loading project definition from /headless/Desktop/ernesto-spark/project
[info] Loading settings for project ernesto-spark from build.sbt ...
[info] Set current project to Spark (in build file:/headless/Desktop/ernesto-spark/)
[info] Defining Global / ideaPort
[info] The new value will be used by Compile / compile, Test / compile
[info] Reapplying settings...
[info] Set current project to Spark (in build file:/headless/Desktop/ernesto-spark/)
[IJ]sbt:Spark>
```

We will cover following topics in this scenario.

- Implementing Custom Accumulators

Prerequisites

We need following packages to perform the lab exercise:

- Java Development Kit
- SBT

JAVA

Verify the installation with: `java -version`

You'll see the following output:

```
java version "1.8.0_201"
Java(TM) SE Runtime Environment (build 1.8.0_201-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.201-b09, mixed mode)
```

SBT

Verify your sbt installation version by running the following command.

`sbt sbtVersion`

You will get following output. If you get an error first time, please run the command again.

```
[info] Loading project definition from /headless/Desktop/ernesto-spark/project
[info] Loading settings for project apache-spark from build.sbt ...
[info] Set current project to Spark (in build file:/headless/Desktop/ernesto-spark/)
[info] 1.3.2
```

Implementing Custom Accumulators V2

Let us implement a Custom Accumulator which counts the number of times each movie has been rated. We can achieve the same using the map transformation and then applying reduceByKey action. But then this will have the data shuffled across the nodes of the cluster. However, when we use Accumulators, the data is not shuffled across the clusters as each executor processes data locally and has its own local accumulator. The only data shuffled across the cluster will be the count from each local accumulator, which will only be a few bytes. The local count from all the executors is aggregated by the global accumulator in the driver, thus providing the final result.

Step 1: Download the ratings.csv file from the URL below. This file contains four columns: userId, movieID, rating and timestamp.

Ratings_head.csv - http://bit.ly/2X3r2wb

Note: We already have cloned a github repository which contains a required file. Open `~/work/ernesto-spark/Files/chapter_6` to view file.

Custom Accumulators...

Step 2: Start IntelliJ IDE and open `~/work/ernesto-spark/src/main/scala/training/CountByMovie.scala` to view scala file.

```
import org.apache.spark.util.AccumulatorV2
import scala.collection.mutable.HashMap
```

The first import is version two of Accumulator. The second import is an mutable HashMap as we will be storing our movies and total number of ratings as key and value respectively. We need to explicitly import the HashMap collection or else we would end up having an immutable HashMap when we declare one.

Step 3: We now have to extend our class to inherit AccumulatorV2 and then specify the input and output. The input to our Accumulator would be a tuple (movieId and count), processed by each task (local accumulator) on executors and the output is a HashMap which will be aggregated by the global accumulator on driver.

```
class CountByMovie extends  AccumulatorV2[(Int, Int), HashMap[Int, Int]]{
```

Step 4: Let us now declare a private HashMap variable called movieCount which will hold the final count of our CountByMovie Accumulator.

```
private val movieCount = new HashMap[Int, Int]()
```

We have to implement a reset method available in the AccumulatorV2 class to reset the accumulator value to zero.

```
def reset() = {
  movieCount.clear()
}
```

Step 5: We now have to implement the add method to specify the aggregation logic for local accumulators, i.e., the tasks which run on executors. All the tasks running on executors will run the method to aggregate data locally.

```
def add(tuple: (Int, Int)): Unit = {
  val movieId = tuple._1
  val updatedCount = tuple._2 + movieCount.getOrElse (movieId, 0)

  movieCount += ((movieId, updatedCount))
}
```

The add method takes two arguments as key and value. The key, which is the first argument, is the movieId, and the second argument, the count of the movieId, is value. We simply extract them into their respective variables and add them to the movieCount Hashmap. The getOrElse method is used to get the value of count if it exists or set a value for that movie as zero and add them with the current count and previous count to get the updated count.

Step 6: The next step is to implement the merge method which actually aggregates all the values from executors and provides with the final count.

```
def merge(tuples: AccumulatorV2[(Int, Int), HashMap[Int, Int]]): Unit = {

  tuples.value.foreach(add)

}

def value() =  movieCount
```

When all the tasks complete executing, the final results from all the executors is then sent to the driver where the merging happens. The merge method takes an AccumulatorV2 as an argument which takes a tuple and returns a HashMap as output. The merge method is called on all the local accumulators from the tasks processed in the exectors. Therefore, we use the add method inside the foreach function. Since we declared the HashMap as private, we can only access it through the value method. The value method is used to simply get the current value in our accumulator.

To summarize, the merge method takes an accumulator as an argument and merges all the local accumulators, which were processed in the executors by tasks, based on the logic in add method, into the global accumulator. The value method is used to get the current value of the HashMap variable movieCount.

Step 7: Next, there are a couple of methods required to complete our implementation of custom accumulator. Ther are the isZero method and the copy method.

```
def isZero(): Boolean = {
  movieCount.isEmpty
}
```

```
def copy() = new CountByMovie
```

The isZero method returns a Boolean by checking if the accumulator value is zero or not. The copy method is used to create a new copy of our accumulator object.

These are the abstract methods which must be implemented in our code as they are defined in the base class. These methods will will be used while aggregating the value in the accumulator.

The error for the class name should be gone now. With this we have successfully implemented our Accumulator V2. We now have to use this custom accumulator in our main program.

Step 8: Start IntelliJ IDE and open `~/work/ernesto-spark/src/main/scala/training/countByMovieMain.scala` to view scala file.

```
import org.apache.spark.sql.SparkSession
```

Let us first create a case class with all our fields for input data outside the object as shown in the screenshot below.

```
case class Movies(userId: Int, movieId: Int, rating : Double, timeStamp: String)
```

Step 9: Let us now write our main function and create a SparkSession object.

```
def main(args: Array[String]) {

  val sparkSession = SparkSession.builder.
    master("local[*]")
    .appName("Count By movieId")
    .getOrCreate()
```

Next, let us create the Accumulator object and register it using the register method as shown below. We have to register out Accumulator since it is custom accumulator. You will not have to register for the built-in accumulators.

```
val countByMovie = new CountByMovie()
sparkSession.sparkContext.register(countByMovie)
```

Step 10: Let us now write some code to read the input data. We will also need to import the implicits.

```
import sparkSession.implicits._

val options = Map("header" -> "true", "inferSchema" -> "true")

val data = sparkSession.read.format("com.databricks.spark.csv")
.options(cvsOptions)
.load(input)
.as[User]
```

Do not worry if this code doesn't make sense. Just think this as a way to read the input data, as we used to do with SparkContext object in the previous exercises. Everything will start to make sense once we cover the next couple of chapters.

Step 11: Let us now apply our custom accumulator in the foreach higher order function and print the result to console.

```
data.foreach(record => {
      countByMovie.add((record.location, 1))
    })

    println(countByMovie.value.toList)

  }
}
```

Here, we are passing our data through the foreach function, where our custom accumulator countByMovie is applied with the add method. We specify the movieId as the field for which the aggregations to be done on. Finally, we can access the result by calling value method on our custom accumulator and convert it to a List.

Step 12: Let us now run the program. You should see the output with count for each movie in a List collection as shown in the screenshot below.

To run this program from the terminal, simply run the following command. The program will the then be compiled and executed.

Run solution using intelliJ IDEA. You can also run using sbt CLI:

```
sbt "runMain training.countByMovieMain"
```

Please note caution while using accumulators. If the output generated from the accumulator is huge data, you should not use the accumulators. Instead, you should use the transformations as required. In this we case, the result of accumulator is just movies and their counts. It is not a huge data. We have achieved our result without shuffling the data across the network, which is usually the case with transformations.

Task is complete!