SparkSQL in Scala

Lab02: SparkSQL Case Classes and Schemas

This lab will help us understand how to use case classes to define a DataFrame's schema.

Case Classes

We're going to work with a similar file to the one we did in the previous lab. This time the file is at `/data/spark-resources-data/people.txt` and instead of being JSON, it's a comma-delimited file.

To create the schema, we're going to define a case class in Scala that provides names and types for each column.

Please note: Case classes in Scala 2.10, the version of Scala used with Spark versions before 2.0, can support only up to 22 fields. You can use custom classes that implement the Product interface to work around that limit. Alternatively, you may wish to simply upgrade to Spark 2.0 with Scala 2.11, which does not impose this limitation on case classes.

Before we begin our Scala program, we must import a couple of spark libraries so that we can create Spark sessions and convert from RDDs to DataFrames and back again.

```
import org.apache.spark.sql._
import spark.implicits._
```

With that note out of the way, let's define a `Person` case class with two columns, a name and an age.

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

Encoders

For those of you who are Object Oriented Programming gurus, you'll be interested to know that Encoders are automatically created for case classes. That means I can create a case class and then immediately convert it to a DataSet or DataFrame like this:

```
scala> val caseClassDF = Seq(Person("Andy", 32)).toDF()
caseClassDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> caseClassDF.show()
+----+---+
|name|age|
+----+---+
|Andy| 32|
+----+---+
```

And in fact, encoders are provided for most common object types. You can use those encoders by importing `spark.implicits._` as we did at the beginning of this lab. That allows us to do implicit mappings from Sequences, Arrays, and Lists to DataFrames and DataSets.

```
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

Populating Our Case Class

Once the case class is defined, we can now read in an RDD and use `map` to assign various bits of text to the columns. Then we'll convert our `Person` object to a DataFrame. Finally, we'll register our DataFrame as a temporary SQL table named `people` so that we can query it using—you guessed it—SQL.

Note we've put the dots at the ends of lines instead of the more traditional position at the beginning so that you can simply copy and paste this code into the spark-shell.

```
// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("hdfs:///data/spark-resources-data/people.txt").
  map(_.split(",")).                        // split by commas
  map(p => Person(p(0), p(1).trim.toInt)). // first field is the person string, second is an i
  toDF()

people.registerTempTable("people")
```

Now if our data already has a structure that maps to our case class, we can save some time by simply telling the `spark.read` command to read the file as that case class type.

If you recall, we had a file called `people.json` that maps nicely to our case class. Therefore we can just say:

```
val path = "hdfs:///data/spark-resources-data/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
```

And we now have a People DataFrame that looks like this:

age	name
null	Michael
30	Andy
19	Justin

SQL Statements

SQL statements can be run by using the sql methods provided by sqlContext.

```
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

The results of SQL queries are DataFrames and support all the normal RDD operations. The columns of a row in the result can be accessed by number.

```
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

That should produce `Name: Justin`

The following statements produce the same output. In the first case we reference the column name by ordinal, while in the second, we reference it by field name.

```
teenagers.map(teenager => "Name: " + teenager(0)).show()
teenagers.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
```

Both statements should produce.

value
Name: Justin

This step concludes the lab.