SparkSQL in Scala

Lab03: SparkSQL Case Classes and Schemas

This lab will demonstrate how Spark SQL can use Hive tables. If you have already created the `employees` and `stocks` Hive tables as part of the Spark Hive Fundamentals module, you may skip directly to the section labeled **Querying Hive Tables from Spark SQL**. However, we recommend you read through this next section anyway because interacting with Hive is a powerful feature of Spark. Further, Spark's intelligence simplifies some aspects of table manipulation over Hive's SQL-based approach.

Creating Hive Tables

The Hive exercises create two tables for later use:

1. `employees` is a very simple table of employees that uses complex data types such as maps and structs
2. `stocks` is a Hive table of roughly 2 million stock prices partitioned by stock exchange and stock symbol.

We'll use two different techniques to create these tables

Employees

We'll create employees from a text file by using Spark SQL statements to create the schema just as we would in Hive. Unlike in the Hive example, we'll save some time by creating this table as an *external* table (i.e., one where we simply provide the location of the file that backs the table).

We'll build out the SQL string first and then invoke the `sql` method using our `spark` session variable.

```
val emptable = """CREATE EXTERNAL TABLE IF NOT EXISTS employees
       (name string,
        salary float,
        subordinates array<string>,
        deductions map<string, float>,
        address struct<street:string, city:string, state:string, zip:int>)
row format delimited
lines terminated by '\n'
stored as textfile location '/data/employees/input/'"""

spark.sql(emptable)
spark.sql("select * from employees").show
```

You should see the results:

name	salary	subordinates	deductions	address
John Doe	100000.0	[Mary Smith, Todd...	Map(Federal Taxes...	[1 Michigan Ave.,...
Mary Smith	80000.0	[Bill King]	Map(Federal Taxes...	[100 Ontario St.,...
Todd Jones	70000.0	[]	Map(Federal Taxes...	[200 Chicago Ave....
Bill King	60000.0	[]	Map(Federal Taxes...	[300 Obscure Dr.,...
Boss Man	200000.0	[John Doe, Fred F...	Map(Federal Taxes...	[1 Pretentious Dr...
Fred Finance	150000.0	[Stacy Accountant]	Map(Federal Taxes...	[2 Pretentious Dr...
Stacy Accountant	60000.0	[]	Map(Federal Taxes...	[300 Main St.,Nap...

Stocks

To keep things simple, we're not going to expect that all our partitions for the stocks table have already been created. Instead, we're going to read in a flat input file and have Spark create a partitioned table in Hive from that file.

Here are the HDFS input files we are going to read:

```
/data/stocks-flat/input/NASDAQ_daily_prices_A.csv
/data/stocks-flat/input/NASDAQ_daily_prices_I.csv
/data/stocks-flat/input/NYSE_daily_prices_G.csv
/data/stocks-flat/input/NYSE_daily_prices_I.csv
```

Read these in using Spark's native .csv reader, which was added to the distribution in Spark 2.0.0. We'll then add column names as arguments to the `toDF` function

```
val stocks = spark.read.format("csv").load("/data/stocks-flat/input/").
  toDF("exchg",
    "symbol",
    "ymd",
    "price_open",
    "price_high",
    "price_low",
    "price_close",
    "volume",
    "price_adj_close")
```

Now we'll write this table into Hive. In the process, we'll specify that it should be partitioned by `exchg` and `symbol`. We'll then ask Spark to describe the table for us.

```
spark.sql("drop table stocks") // delete if already exists
stocks.write.partitionBy("exchg", "symbol").saveAsTable("stocks")
spark.sql("describe stocks").show(50)
```

You should see the following description showing that the table ha been properly partitioned.

col_name	data_type	comment
ymd	string	null
price_open	string	null
price_high	string	null
price_low	string	null
price_close	string	null
volume	string	null
price*adj*close	string	null
exchg	string	null
symbol	string	null
# Partition Information		
# col_name	data_type	comment
exchg	string	null
symbol	string	null

SQL Queries

With the tables all set up, we can now do normal SQL queries on our tables. So let's get to it.

First, let's find those employees who live in Zip Code 60500.

```
spark.sql("SELECT name FROM employees WHERE address.zip = 60500").show()
```

You should see

name
Boss Man
Fred Finance

That was a piece of cake. Let's now transition to `stocks`, which is a bit more of a Big Data dataset at more than 2 million rows. Actually, let's count how many rows there actually are.

```
val stks = spark.read.table("stocks")
stks.count
```

The value should be 2,131,092.

One of the nice things about the SQL interface is that types get converted on the fly based on context. If we look at the descrioption of the stock table above, every column was a string type. We're now going to do some numeric comparisons.

Up until this point, we've invoked sql as a method on our Spark Session. However, SQL is used so commonly that you can leave off the spark session reference.

```
sql("""SELECT ymd, price_open, price_close FROM stocks
WHERE symbol = 'AAPL' AND exchg = 'NASDAQ' LIMIT 20""").show()
```

Output should be

ymd	price_open	price_close
2015-06-22	127.489998	127.610001
2015-06-19	127.709999	126.599998
2015-06-18	127.230003	127.879997
2015-06-17	127.720001	127.300003
2015-06-16	127.029999	127.599998
2015-06-15	126.099998	126.919998
2015-06-12	128.190002	127.169998
2015-06-11	129.179993	128.589996
2015-06-10	127.919998	128.880005
2015-06-09	126.699997	127.419998
2015-06-08	128.899994	127.800003
2015-06-05	129.5	128.649994
2015-06-04	129.580002	129.360001
2015-06-03	130.660004	130.119995
2015-06-02	129.860001	129.960007
2015-06-01	130.279999	130.539993
2015-05-29	131.229996	130.279999
2015-05-28	131.860001	131.779999
2015-05-27	130.339996	132.039993
2015-05-26	132.600006	129.619995

Now let's see

```
sql("SELECT year(s.ymd), avg(s.price_close) FROM stocks s  WHERE s.symbol = 'AAPL' AND s.exchg
```

You should now see

year(CAST(ymd AS DATE))	avg(CAST(price_close AS DOUBLE))
1990	37.56175417786561
2003	18.54476169444443
2007	128.2739047848606
2015	124.3063555169492
2013	472.63488080952385
1997	17.966775490118593
1988	41.53902472332016
1994	34.08054893650793
2014	295.4023412182538
2004	35.52694387301588
1982	19.142774332015808
1996	24.919478582677176
1989	41.65872438095236
1998	30.564851150793665
1985	20.195169592885374
2012	576.0497200880001
2009	146.81412911904766
1995	40.54017670238094
1980	30.442332307692308
2001	20.219431697580646