Spark Program

CHAPTER 4: SPARK SQL

Chapter Objectives

In this chapter, we will:

- Introduce Spark SQL
- Use SQL on DataFrames
- Discuss Hive integration

Spark SQL

- Spark 2.0 introduced not only DataFrames, but the ability to query them in two different ways
  - Using methods on the DataFrame variable
    - Syntax is programmatic, so more familiar to programmers not already well versed in SQL
    - Many more methods exist than the SQL language supports
  - Using SQL queries
    - Uses familiar syntax based on HQL (Hive Query Language)
    - Allows data analysts to leverage already existing queries and knowledge to get results quicker
- Can combine the two methods together to mix and match where appropriate for solving a query

Hive Integration

- Spark 2.0 can directly query a table in the Hive catalog if you add Hive support when making the Spark session object

```
spark=SparkSession.builder.enableHiveSupport().getOrCreate()
```

- Then using the SQL method, you can run any Hive query

```
   regions = spark.sql('select * from regions')
   regions.show()
```

- This will simply use the metadata definition in the Hive catalog and do all the processing in Spark
  – Hive is not doing the work, Spark is

Temporary View

- A Hive table is nothing more than a stored definition in the Hive catalog
- Even in Hive, the table is virtual and only exists during processing
- The `createDataFrame` method can be called on an existing DataFrame to temporarily put it into a local copy of the Hive catalog for the current Spark session and treat it as if it were a Hive table

```
x1 = spark.createDataFrame(x, schema = ['ID', 'Name'])
x1.createOrReplaceTempView('MyTable')
```

- Once created, this virtual table or view can be queried with Spark SQL using the HQL dialect of SQL

```
x2 = spark.sql('select * from MyTable')
x2.show()
```

Writing Results To Table

- Not only could you read from a Hive table, but you could also write the results of a query to a Hive table
- This has the effect of writing the output to HDFS and keeping a copy of the schema information in the Hive catalog
- Could use the `saveAsTable` method

```
x1.write.saveAsTable('Table1', mode = 'overwrite')
```

- Or using `CREATE TABLE AS` syntax of SQL

```
spark.sql('CREATE TABLE Table2 AS SELECT * FROM
MyTable')
```

- `x1 & x2` are DataFrames, `MyTable` is a temporary view, `Table1 & Table2` become permanent entries in the Hive catalog

Methods vs. SQL

- Once you have a temporary view or Hive table, you can use a combination of SQL queries and method calls
- These two queries are equivalent, although SQL tends to be not case sensitive whereas Spark & Python are, so sometimes you need to watch out for case

```
sql = """select r.regionid, r.regionname, t.territoryid,
t.territoryname
from regions as r
join territories as t on r.regionid = t.regionid
order by r.regionid, t.territoryid"""
rt = spark.sql(sql)
```

```
tr = regions.join(territories, regions.regionid ==
territories.RegionID).select('regions.regionid',
'regionname', 'TerritoryID', 'TerritoryName')
```

Interacting with SQL Servers

- Data in any SQL server such as Oracle, MySQL, Sybase, or Microsoft SQL Server is stored in one central server and can be processed using that server's dialect of SQL
  - Does not run in a cluster or scale
  - SQL is used to store and process
- Spark can read and write data stored in a SQL Server into a Spark DataFrame and process it
  - Uses the cluster and scales
  - Only uses SQL as storage

Interacting with SQL Servers

- Spark can connect to any JDBC or ODBC data source
  - Write a DataFrame to a new SQL table

```
regions.write.format("jdbc").options(url="jdbc:mysql:/
/localhost/northwind", driver='com.mysql.jdbc.Driver',
dbtable='regions', user='test', password = "password",
mode = "append", useSSL = "false").save()
```

  - Read a SQL table into a DataFrame

```
regions2 = spark.read.format("jdbc").
options(url="jdbc:mysql://localhost/northwind",
driver="com.mysql.jdbc.Driver", dbtable= "regions",
user="test", password="password").load()
regions2.show()
```

Calculated Columns

- Creating the `regions2` DataFrame does not execute anything yet
- Making DataFrame into a Temp View then running a Spark SQL query, tells Spark to read the SQL data into an RDD in the Spark Cluster
- Once in the cluster, it can be processed using Spark methods or Spark SQL

```
regions2.createOrReplaceTempView('regions2')
spark.sql('select * from regions2
           where regionid < 3').show()
```

Mixing Python and SQL

- HQL is predefined to have most of the standard functions in the SQL language
- Sometimes Python has a function that SQL does not
- You could always use Spark methods to call a Python function as we showed in the last chapter

```
from pyspark.sql.functions import expr, udf
from pyspark.sql.types import *
t2 = spark.sql('select * from territories')
t2.printSchema()
t2 = t2.withColumn('upperName', expr('UPPER(TerritoryName)'))
t2.show(5)
t2 = t2.withColumn('titleName', udf(lambda x : x.title(),
StringType())(t2.upperName))
t2.show(5)
```

User Defined Functions

- To make it easier to use, you could take a Python function and turn it into a UDF
- This is similar to how you make a DataFrame into a Temp View
- Registering the Python function makes it a UDF callable within Spark SQL queries

```
def reverseString(x):
    return x[::-1]

spark.udf.register('reverse', reverseString, StringType())

spark.sql('select *, reverse(TerritoryName) as Reversed
from Territories').orderBy('Reversed').show()
```

Complex Data Types

- In addition to the typical primitive SQL datatypes, Spark has a few complex types that are similar to Python collection types and Hive's complex datatypes
  - `ArrayType` is similar to a list
  - `MapType` is similar to a dictionary
  - `StructType` & `StructField` allow you to build a table structured object
- Can be used to create more complex shaped objects than is possible in traditional SQL relational model
- Using the `ArrayType`, multiple children values can be embedded into a column instead of using a related table
- You could create a dynamic on the fly collection of Key Value pairs using `MapType`
- Using `StructType` & `StructField`, you could create a table structure that could embed an entire sub-table into a field

Use Case

- Typically `Join`, `Group`, and `Aggregate` operations tend to be more expensive because they are wide transformations that require a lot of shuffling of data around the nodes
- An alternative data modeling technique would be to embed children records inside the parent so that in a sense they are pre-joined to the parent
- This reduces the need to do joins and to shuffle data when doing aggregates

Collect_List

- Spark has two aggregate functions called `collect_list` and `collect_set` that can be used to group together elements into either a list or a unique set

```
from pyspark.sql.functions import collect_list
territories.groupBy(territories.RegionID). \
agg(collect_list(territories.TerritoryName)).show()
```

- HQL has a similar function and is much easier to use

```
tr1 = spark.sql("SELECT RegionID,
collect_list(TerritoryName) AS TerritoryList
FROM Territories
GROUP BY RegionID")
tr1.show()
tr1.printSchema()
print(tr1.take(1))
```

Complex Collect_List

- While it could be done using pure Spark methods, it is much easier to use HQL's `NAMED_STRUCT` function to create a Spark array of structs

```
SELECT r.RegionID, r.RegionName,
COLLECT_SET(NAMED_STRUCT("TerritoryID", TerritoryID,
"TerritoryName", TerritoryName)) AS TerritoryList
FROM Regions AS r
JOIN Territories AS t ON r.RegionID = t.RegionID
GROUP BY r.RegionID, r.RegionName
ORDER BY r.RegionID
```

```
DataFrame[RegionID: int, RegionName: string,
TerritoryList:
array<struct<TerritoryID:string,TerritoryName:string>>]
```

Exploding

- Sometimes you get data that is nested in this form and you want to flatten it out
- This could happen if you read from a complex data source like JSON, XML, or a Hive table that is structured this way
- To flatten out an embedded array of values, there is a Spark function called explode

```
from pyspark.sql.functions import explode
tr1.select('RegionID', explode('TerritoryList')).show()
```

- HQL has a similar function

```
tr1.createOrReplaceTempView('RegionTerritories')
sql = """SELECT RegionID, TerritoryName
FROM RegionTerritories
LATERAL VIEW EXPLODE(TerritoryList)
EXPLODED_TABLE AS TerritoryName
ORDER BY RegionID, TerritoryName"""
spark.sql(sql).show()
```

Complex Exploding

- If the object you are exploding contains a structure, you simply use dot syntax to drill down into the subelements

```
sql = """SELECT RegionID, RegionName,
Territory.TerritoryID AS TerritoryID,
Territory.TerritoryName AS TerritoryName
FROM RegionTerritories
LATERAL VIEW EXPLODE(TerritoryList) EXPLODED_TABLE AS
Territory"""
spark.sql(sql).show()
```

User Defined Functions

- Sometimes it is necessary to write complex functions using Python
- Import the helper functions in pyspark.sql

```
from pyspark.sql.functions import udf
from pyspark.sql.types import *
from pyspark.sql.functions import to_date
```

- Write whatever custom function you need

```
def city(x):
    return x[:x.find(',')]
def country(x):
    return x[x.find(',') + 1 :]
```

- Call the built-in function or use the `udf` function to wrap and call your UDF

```
df4.withColumn('City', udf(city, StringType())(df4.CityCountry)) \
.withColumn('Country', udf(country, StringType())(df4.CityCountry)) \
.withColumn('Date', to_date(df4.Date, 'dd-MMM-yy')) \
.drop(df4.CityCountry)
```

Closing Thoughts

- Ultimately, Spark is just a processing engine that can read data from almost any source
  - Local files would be read in and distributed into the cluster
  - HDFS files are already in a cluster and could be parallel loaded into Spark nodes simultaneously
  - Data in a SQL table would be similar to a local file and would be fed into the cluster as it retrieves it from the SQL server
  - Data in a NoSQL cluster is already partitioned in a cluster and like HDFS could be parallel loaded
- You will get the fastest load performance for both reading and writing when the data is already stored in a clustered environment
- There are many built-in data sources
  - JSON
  - CSV
  - Parquet
  - ORC

Closing Thoughts (continued)

- You can also use JDBC and `format` to load in custom formats
  - AVRO
  - Cassandra
  - Mongo
  - HBase
- The `os.environ` setting is used to add the custom jars necessary to support these features
- Here is an example of starting a Spark session which could read and write data in a Cassandra cluster

```
import os
os.environ['PYSPARK_SUBMIT_ARGS'] = '--packages
com.datastax.spark:spark-cassandra-connector_2.11:2.3.0 --conf
spark.cassandra.connection.host=192.168.0.123,192.168.0.124
pyspark-shell'
table_df = sqlContext.read \
.format("org.apache.spark.sql.cassandra") \
.options(table=table_name, keyspace=keys_space_name) \
.load()
```

Chapter Summary

In this chapter, we have:

- Introduced Spark SQL
- Used SQL on DataFrames
- Discussed Hive integration