Ch. 10 - Spark SQL

Spark runs the popular benchmark **TPC-DS** - subset of ANSI SQL:2003

Apache Hive

- Before Spark, Hive was the de facto big data SQL access layer
- developed at Facebook
- Hive in many ways it helped propel Hadoop into different industries because analysts could run SQL queries
- Spark began as a general processing engine with *Resilient Distributed Datasets* (RDDs), many now use Spark SQL
- With Spark 2.0, its authors created a superset of Hive's support
  - Native SQL parser that supports both ANSI-SQL as well as HiveQL queries
  - unique interoperability with DataFrames
- 2016, Facebook announced that it had begun running Spark workloads
  - significant performance improvements

Advantages of Spark SQL

- analysts can use Thrift Server or Spark's SQL interface
- data engineers and scientists can use Spark SQL where appropriate in any data flow
- unifying API allows for data to be extracted with SQL, manipulated as a DataFrame, passed into one of Spark MLlibs' large-scale machine learning algorithms, and written out to another data source

Spark SQL is intended to operate as an **online analytic processing** (OLAP) database, not an **online transaction processing** (OLTP) database. This means that it is not intended to perform e**xtremely low-latency queries**.

Spark SQL can connect to Hive metastores

- Hive metastore is the way in which Hive maintains table information for use across sessions
- Spark SQL, you can connect to your Hive metastore (if you already have one)
- access table metadata to reduce file listing when accessing information. T
- popular for migrating from a legacy Hadoop environment

Spark SQL CLI

- ./bin/spark-sql
- Spark SQL CLI cannot communicate with the Thrift JDBC server
- configure Hive by placing your hive-site.xml, core-site.xml, and hdfs-site.xml files in conf
- ./bin/spark-sql --help

Programmatic SQL Interface

- can also execute SQL in an ad hoc manner via any of Spark's language APIs via the method **sql** on the SparkSession object, returning a DataFrame

- The command

```
- spark.sql("SELECT 1 + 1")
```

returns a DataFrame that we can then evaluate programmatically.

- will not be executed eagerly but *lazily*
- some transformations are much simpler to express in SQL code than in DataFrames
- can express multiline queries quite simply by passing a multiline string into the function

```
spark.sql("""SELECT user_id, department, first_name FROM professors  WHERE department IN    (SELECT name FROM department WHERE created_date >= '2016-01-01')""")
```

- can completely interoperate between SQL and DataFrames
- can create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame

```
spark.sql("""SELECT DEST_COUNTRY_NAME, sum(count)FROM some_sql_view GROUP BY
DEST_COUNTRY_NAME""")\.where("DEST_COUNTRY_NAME like 'S%'").where("`sum(count)` > 10")\.count()#
SQL => D
```

Spark provides a *Java Database Connectivity* (JDBC) interface

- connects to the Spark driver in order to execute Spark SQL queries
- a common use case might be a for a business analyst to connect business intelligence software like Tableau to Spark
- The Thrift JDBC/Open Database Connectivity (ODBC) server implemented here corresponds to the HiveServer2 in Hive 1.2.1.
- You can test the JDBC server with the **beeline** script that comes with either Spark or Hive 1.2.1.
- To start the JDBC/ODBC server, run the following in the Spark directory:

```
./sbin/start-thriftserver.sh
```

- This script accepts all bin/spark-submit command-line options. To see all available options for configuring this Thrift Server, run

```
./sbin/start-thriftserver.sh --help
```

- By default,the server listens on localhost:10000
- You can override this through environmental variables or system properties
- For environment configuration, use this

```
:export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
--master <master-uri> \
```

For system properties:

```
./sbin/start-thriftserver.sh \
--hiveconf hive.server2.thrift.port=<listening-port> \
--hiveconf hive.server2.thrift.bind.host=<listening-host> \
```

```
--master <master-uri>
```

You can then test this connection by running the following commands:

```
./bin/beeline
beeline> !connect jdbc:hive2://localhost:10000
```

Beeline will ask you for a username and password. In nonsecure mode, simply type the username on your machine and a blank password

Catalog

- The highest level abstraction in Spark SQL
- Catalog is an abstraction for the storage of metadata about the data stored in your tables as well as other helpful things like databases, tables, functions, and views
- The catalog is available in the *org.apache.spark.sql.catalog.Catalog* package and contains a number of helpful functions for doing things like listing tables, databases, and functions.

For these examples keep in mind that you need to wrap everything in a spark.sql function call to execute the relevant code

Tables

- to do anything useful with Spark SQL, you first need to define tables
- *Tables are logically equivalent to a DataFrame* in that they are a structure of data against which you run commands
- We can join tables, filter them, aggregate them, and perform different manipulations that we saw in previous chapters.
- The core difference between tables and DataFrames is this: **you define DataFrames in the scope of a programming language, whereas you define tables within a database.**
- This means that when you create a table (assuming you never changed the database), it will belong to the default database.
- in Spark 2.X, tables always contain data. There is no notion of a temporary table, only a view, which does not contain data. This is important because if you goto drop a table, you can risk losing the data when doing so.

managed versus **unmanaged** tables

- tables store two important pieces of information. The data **within** the tables as well as the data **about** the tables; that is, the metadata.
- You can have Spark **manage** the metadata for a set of files as well as for the data.
- When you define a table from files on disk, you are defining an **unmanaged** table.
- When you use **saveAsTable** on a DataFrame, you are creating a managed table for which Spark will track of all of the relevant information.
- This will read your table and write it out to a new location in Spark format - to the default Hive warehouse

location.

- You can set this by setting the *spark.sql.warehouse.dir* configuration to the directory of your choosing when you create your SparkSession.
- By defaultSpark sets this to */user/hive/warehouse*:
- Can also see tables in a specific database by using the query

```
show tables IN databaseName,
```

where **databaseName** represents the name of the database that you want to query.

- If you are running on a new cluster or local mode, this should return zero results

Creating Tables

- capability of *reusing the entire Data Source API within SQL*
- Do not need to define a table and then load data into it; Spark lets you create one on the fly
- Can even specify all sorts of sophisticated options when you read in a file.

USING and STORED AS

- the specification of the **USING** syntax - if you do not specify the format, Spark will default to a **Hive SerDe** configuration.
- This has performance implications for future readers and writers because Hive SerDes are much slower than Spark's native serialization.
- Use the **STORED AS** syntax to specify that this should be a Hive table

Partition specification

- if you want to write only into a certain partition, optionally
- a write will respect a partitioning scheme, as well (which may cause the above query to run quite slowly)
- will add additional files only into the end partitions

Describing Metadata

- Can add a comment when creating a table by *describing the table metadata*, which will show us the relevant comment:

```
DESCRIBE TABLE flights_csv
```

- See the partitioning scheme for the data by using the following (note, however, that this works only on partitioned tables):

```
SHOW PARTITIONS partitioned_flights
```

Important to ensure that you're reading from the most recent set of data:

REFRESH TABLE refreshes all cached entries (essentially, files) associated with the table. If the table were previously cached, it would be cached lazily the next time it is scanned

REPAIR TABLE refreshes the partitions maintained in the catalog for that given table

- this command's focus is on collecting new partition information—an example might be writing out a new this command's focus is on collecting new partition information - an example might be writing out a new partition manually and the need to repair the table accordingly

DROP TABLE IF EXISTS...

Caching Tables

Just like DataFrames, you can cache and uncache tables by specifying a table:

```
CACHE TABLE flights
```

```
UNCACHE TABLE FLIGHTS
```