# Spark SQL

## What Is SQL?

SQL or Structured Query Language is a domain-specific language for expressing relational operations over data.

Spark SQL: originated as Apache Hive to run on top of Spark and is now integrated with the Spark stack. Apache Hive had certain limitations as mentioned below. Spark SQL was built to overcome these drawbacks and replace Apache Hive.

Limitations with Hive:

* Hive launches MapReduce jobs internally for executing the ad-hoc queries. MapReduce lags in the performance when it comes to the analysis of medium-sized datasets (10 to 200 GB).
* Hive has no resume capability. This means that if the processing dies in the middle of a workflow, you cannot resume from where it got stuck.

.

## Spark SQL Overview

Spark SQL integrates relational processing with Spark’s functional programming. It provides support for various data sources and makes it possible to weave SQL queries with code transformations thus resulting in a very powerful tool.

NOTE: 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 extremely low-latency queries.

## Spark’s Relationship to Hive :

Spark SQL has a great relationship with Hive because it can connect to Hive metastores. The Hive metastore is the way in which Hive maintains table information for use across sessions. With Spark SQL, you can connect to your Hive metastore (if you already have one) and access table metadata to reduce file listing when accessing information. This is popular for users who are migrating from a legacy Hadoop environment and beginning to run all their workloads using Spark.

## The **Hive metastore**

To connect to the Hive metastore, there are several properties that you’ll need. First, you need to set the Metastore version (spark.sql.hive.metastore.version) to correspond to the proper Hive metastore that you’re accessing. By default, this value is 1.2.1. You also need to set spark.sql.hive.metastore.jars

# How to Run Spark SQL Queries

Spark provides several interfaces to execute SQL queries.

## Spark SQL CLI

The Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line. **Note that the Spark SQL CLI cannot communicate with the Thrift JDBC server.**

To start the Spark SQL CLI, run the following in the Spark directory:

./bin/spark-sql

You configure Hive by placing your hive-site.xml, core-site.xml, and hdfs-site.xml files in conf/.

## Spark’s Programmatic SQL Interface

In addition to setting up a server, you can also execute SQL in an ad hoc manner via any of Spark’s language APIs. You can do this via the method **sql** on the SparkSession object. This returns a DataFrame. For example, in Python or Scala, we can run the following:

spark.sql("SELECT 1 + 1").show()

The command spark.sql("SELECT 1 + 1") returns a DataFrame that we can then evaluate programmatically. Just like other transformations, this will not be executed eagerly but lazily.

This is an immensely powerful interface because there are some transformations that are much simpler to express in SQL code than in DataFrames. You can express multiline queries quite simply by passing a multiline string into the function. For example, you could execute something like the following code in Python or Scala:

spark.sql("""SELECT user\_id, department, first\_name FROM professors WHERE department IN (SELECT name FROM department WHERE created\_date >= '2016-01-01')""")

**Even more powerful, you can completely interoperate between SQL and DataFrames, as you see fit. For instance, you can create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame.**

# in Python

spark.read.json("/data/flight-data/json/2015-summary.json")\ .createOrReplaceTempView("some\_sql\_view") # DF => SQL

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 => DF

## Catalog

The highest level abstraction in Spark SQL is the Catalog. The 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.

## 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 sessions.

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.

Spark-Managed Tables

One important note is the concept of managed versus unmanaged tables.

Tables store two important pieces of information.

* The data within the tables
* 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.

**df.write.mode("overwrite").saveAsTable("database.tableName")**

// The created tables are MANAGED.

df.write.saveAsTable("t10")

// The created tables are EXTERNAL

df.write.option("path", "/tmp/tables/t9").saveAsTable("t9")

### Creating Tables

You can create tables from a variety of sources. Something fairly unique to Spark is the capability of reusing the entire Data Source API within SQL. This means that you do not need to define a table and then load data into it; Spark lets you create one on the fly. You can even specify all sorts of sophisticated options when you read in a file. For example, here’s a simple way to read in the flight data we worked with in previous chapters:

CREATE TABLE flights ( DEST\_COUNTRY\_NAME STRING, ORIGIN\_COUNTRY\_NAME STRING, count LONG) USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')

You can also add comments to certain columns in a table, which can help other developers understand the data in the tables:

CREATE TABLE flights\_csv ( DEST\_COUNTRY\_NAME STRING, ORIGIN\_COUNTRY\_NAME STRING COMMENT "remember, the US will be most prevalent", count LONG) USING csv OPTIONS (header true, path '/data/flight-data/csv/2015-summary.csv')

It is possible to create a table from a query as well:

CREATE TABLE flights\_from\_select USING parquet AS SELECT \* FROM flights

## Creating External Tables

Luckily, you can, for the most part, just copy and paste your Hive statements directly into Spark SQL. For example, in the example that follows, we create an unmanaged table. Spark will manage the table’s metadata; however, the files are not managed by Spark at all. You create this table by using the CREATE EXTERNAL TABLE statement. You can view any files that have already been defined by running the following command:

CREATE EXTERNAL TABLE hive\_flights ( DEST\_COUNTRY\_NAME STRING, ORIGIN\_COUNTRY\_NAME STRING, count LONG) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/data/flight-data-hive/'

You can also create an external table from a select clause:

CREATE EXTERNAL TABLE hive\_flights\_2 ROW FORMAT DELIMITED FIELDS TERMINATED BY ','

LOCATION '/data/flight-data-hive/' AS SELECT \* FROM flights

## Inserting into Tables

Insertions follow the standard SQL syntax:

INSERT INTO flights\_from\_select SELECT DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME, count FROM flights LIMIT 20

INSERT INTO partitioned\_flights PARTITION (DEST\_COUNTRY\_NAME="UNITED STATES") SELECT count, ORIGIN\_COUNTRY\_NAME FROM flights WHERE DEST\_COUNTRY\_NAME='UNITED STATES' LIMIT 12

## Describing Table Metadata

**DESCRIBE TABLE flights\_csv**

You can also see the partitioning scheme for the data by using the following (note, however, that this works only on partitioned tables):

**SHOW PARTITIONS partitioned\_flights**

## Refreshing Table Metadata

Maintaining table metadata is an important task to ensure that you’re reading from the most recent set of data. There are two commands to refresh table metadata. 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:

**REFRESH table partitioned\_flights**

Another related command is REPAIR TABLE, which 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 partition manually and the need to repair the table accordingly:

**MSCK REPAIR TABLE partitioned\_flights**

**Dropping Tables**

You cannot delete tables: you can only “drop” them. You can drop a table by using the DROP keyword. If you drop a managed table (e.g., flights\_csv), both the data and the table definition will be removed:

DROP TABLE flights\_csv;

WARNING Dropping a table deletes the data in the table, so you need to be very careful when doing this.

If you try to drop a table that does not exist, you will receive an error. To only delete a table if it already exists, use DROP TABLE IF EXISTS.

DROP TABLE IF EXISTS flights\_csv;

WARNING This deletes the data in the table, so exercise caution when doing this.

Dropping unmanaged tables If you are dropping an unmanaged table (e.g., hive\_flights), no data will be removed but you will no longer be able to refer to this data by the table name.

## Caching Tables

Just like DataFrames, you can cache and uncache tables. You simply specify which table you would like using the following syntax:

CACHE TABLE flights

Here’s how you uncache them:

UNCACHE TABLE flights

## Views

Now that you created a table, another thing that you can define is a view. A view specifies a set of transformations on top of an existing table—basically just saved query plans, which can be convenient for organizing or reusing your query logic. Spark has several different notions of views. Views can be global, set to a database, or per session.

### Creating Views

To an end user, views are displayed as tables, except rather than rewriting all of the data to a new location, they simply perform a transformation on the source data at query time. This might be a filter, select, or potentially an even larger GROUP BY or ROLLUP. For instance, in the following example, we create a view in which the destination is United States in order to see only those flights:

CREATE VIEW just\_usa\_view AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

Like tables, you can create temporary views that are available only during the current session and are not registered to a database:

CREATE TEMP VIEW just\_usa\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

Or, it can be a global temp view. **Global temp views are resolved regardless of database and are viewable across the entire Spark application, but they are removed at the end of the session:**

CREATE GLOBAL TEMP VIEW just\_usa\_global\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

SHOW TABLES

You can also specify that you would like to overwite a view if one already exists by using the keywords shown in the sample that follows. We can overwrite both temp views and regular views:

CREATE OR REPLACE TEMP VIEW just\_usa\_view\_temp AS SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

Now you can query this view just as if it were another table:

SELECT \* FROM just\_usa\_view\_temp

***A view is effectively a transformation and Spark will perform it only at query time***. This means that it will only apply that filter after you actually go to query the table (and not earlier). **Effectively, views are equivalent to creating a new DataFrame from an existing DataFrame.** In fact, you can see this by comparing the query plans generated by Spark DataFrames and Spark SQL. In DataFrames, we would write the following:

val flights = spark.read.format("json") .load("/data/flight-data/json/2015-summary.json")

val just\_usa\_df = flights.where("dest\_country\_name = 'United States'") just\_usa\_df.selectExpr("\*").explain

In SQL, we would write (querying from our view) this:

EXPLAIN SELECT \* FROM just\_usa\_view

Or, equivalently:

EXPLAIN SELECT \* FROM flights WHERE dest\_country\_name = 'United States'

Due to this fact, you should feel comfortable in writing your logic either on DataFrames or SQL— whichever is most comfortable and maintainable for you.

Dropping Views

You can drop views in the same way that you drop tables; you simply specify that what you intend to drop is a view instead of a table. The main difference between dropping a view and dropping a table is that with a view, no underlying data is removed, only the view definition itself:

DROP VIEW IF EXISTS just\_usa\_view;

Databases

Databases are a tool for organizing tables. As mentioned earlier, if you do not define one, Spark will use the default database. Any SQL statements that you run from within Spark (including DataFrame commands) execute within the context of a database. This means that if you change the database, any user-defined tables will remain in the previous database and will need to be queried differently.

**WARNING This can be a source of confusion, especially if you’re sharing the same context or session for your coworkers, so be sure to set your databases appropriately.**

You can see all databases by using the following command:

SHOW DATABASES

Creating Databases Creating databases follows the same patterns you’ve seen previously in this chapter; however, here

you use the CREATE DATABASE keywords:

CREATE DATABASE some\_db

Setting the Database You might want to set a database to perform a certain query. To do this, use the USE keyword followed by the database name:

USE some\_db

After you set this database, all queries will try to resolve table names to this database. Queries that were working just fine might now fail or yield different results because you are in a different database:

SHOW tables

SELECT \* FROM flights -- fails with table/view not found

However, you can query different databases by using the correct prefix:

SELECT \* FROM default.flights

You can see what database you’re currently using by running the following command:

SELECT current\_database()

You can, of course, switch back to the default database:

USE default;

Dropping Databases Dropping or removing databases is equally as easy: you simply use the DROP DATABASE keyword:

DROP DATABASE IF EXISTS some\_db;

# **Advanced Topics**

Now that we defined where data lives and how to organize it, let’s move on to querying it. A SQL query is a SQL statement requesting that some set of commands be run. SQL statements can define manipulations, definitions, or controls. The most common case are the manipulations, which is the focus of this book.

## Complex Types

Complex types are a departure from standard SQL and are an incredibly powerful feature that does not exist in standard SQL. Understanding how to manipulate them appropriately in SQL is essential. There are three core complex types in Spark SQL: structs, lists, and maps.

### **Structs**

Structs are more akin to maps. They provide a way of creating or querying nested data in Spark. To create one, you simply need to wrap a set of columns (or expressions) in parentheses:

CREATE VIEW IF NOT EXISTS nested\_data AS SELECT (DEST\_COUNTRY\_NAME, ORIGIN\_COUNTRY\_NAME) as country, count FROM flights

Now, you can query this data to see what it looks like:

SELECT \* FROM nested\_data

You can even query individual columns within a struct—all you need to do is use dot syntax:

SELECT country.DEST\_COUNTRY\_NAME, count FROM nested\_data

If you like, you can also select all the subvalues from a struct by using the struct’s name and select all of the subcolumns. Although these aren’t truly subcolumns, it does provide a simpler way to think about them because we can do everything that we like with them as if they were a column:

SELECT country.\*, count FROM nested\_data

### Lists

If you’re familiar with lists in programming languages, Spark SQL lists will feel familiar. There are several ways to create an array or list of values. You can use the collect\_list function, which creates a list of values. You can also use the function collect\_set, which creates an array without duplicate values. These are both aggregation functions and therefore can be specified only in aggregations:

SELECT DEST\_COUNTRY\_NAME as new\_name, collect\_list(count) as flight\_counts, collect\_set(ORIGIN\_COUNTRY\_NAME) as origin\_set FROM flights GROUP BY DEST\_COUNTRY\_NAME

You can, however, also create an array manually within a column, as shown here:

SELECT DEST\_COUNTRY\_NAME, ARRAY(1, 2, 3) FROM flights

You can also query lists by position by using a Python-like array query syntax:

SELECT DEST\_COUNTRY\_NAME as new\_name, collect\_list(count)[0] FROM flights GROUP BY DEST\_COUNTRY\_NAME

You can also do things like convert an array back into rows. You do this by using the explode function. To demonstrate, let’s create a new view as our aggregation:

CREATE OR REPLACE TEMP VIEW flights\_agg AS SELECT DEST\_COUNTRY\_NAME, collect\_list(count) as collected\_counts FROM flights GROUP BY DEST\_COUNTRY\_NAME

Now let’s explode the complex type to one row in our result for every value in the array. The DEST\_COUNTRY\_NAME will duplicate for every value in the array, performing the exact opposite of the original collect and returning us to the original DataFrame:

SELECT explode(collected\_counts), DEST\_COUNTRY\_NAME FROM flights\_agg

## Functions

In addition to complex types, Spark SQL provides a variety of sophisticated functions. You can find most of these functions in the DataFrames function reference; however, it is worth understanding how to find these functions in SQL, as well. To see a list of functions in Spark SQL, you use the SHOW FUNCTIONS statement:

SHOW FUNCTIONS

You can also more specifically indicate whether you would like to see the system functions (i.e., those built into Spark) as well as user functions:

SHOW SYSTEM FUNCTIONS

User functions are those defined by you or someone else sharing your Spark environment. These are the same user-defined functions that we talked about in earlier chapters (we will discuss how to create them later on in this chapter):

SHOW USER FUNCTIONS

You can filter all SHOW commands by passing a string with wildcard (\*) characters. Here, we can see all functions that begin with “s”:

SHOW FUNCTIONS "s\*";

Optionally, you can include the LIKE keyword, although this is not necessary:

SHOW FUNCTIONS LIKE "collect\*";

Even though listing functions is certainly useful, often you might want to know more about specific functions themselves. To do this, use the DESCRIBE keyword, which returns the documentation for a specific function. User-defined functions

As we saw in Chapters 3 and 4, Spark gives you the ability to define your own functions and use them in a distributed manner. You can define functions, just as you did before, writing the function in the language of your choice and then registering it appropriately:

def power3(number:Double):Double = number \* number \* number spark.udf.register("power3", power3(\_:Double):Double)

SELECT count, power3(count) FROM flights

You can also register functions through the Hive CREATE TEMPORARY FUNCTION

syntax.https://docs.databricks.com/spark/latest/spark-sql/index.html