

# BIG DATA ANALYTICS

## Spark SQL

# Spark SQL: *Outline*

- Spark SQL Overview 
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- Select Statements

# Spark SQL: Overview

- One of Spark's most powerful features.
- Enables running SQL queries on views or tables organized in databases.
- Supports system and user-defined functions.
- Integrates with DataFrame and Dataset APIs.
- SQL queries and DataFrame operations compile into the same underlying code.
- Helps optimize workloads by analyzing query plans.

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL? 
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- Select Statements

# Spark SQL: What Is SQL?

- SQL (Structured Query Language) is a domain-specific language for working with relational data.
- Used in all relational databases and many NoSQL (Not Only SQL) databases.
- Despite predictions of its decline, SQL remains a key tool for businesses.
- Spark supports a subset of the ANSI SQL:2003 (American National Standards Institute) standard, enabling compatibility with many databases.
- Spark can run the popular TPC-DS (Transaction Processing Performance Council - Decision Support) benchmark due to this support.

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL 
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- Select Statements

# Spark SQL | Big Data and SQL: Spark SQL

## Spark SQL Overview

- Spark SQL supports both ANSI-SQL and HiveQL, providing a powerful tool for data processing.
- It integrates with DataFrames and the Dataset API, allowing SQL and DataFrame code to compile into the same underlying code.
- Spark is widely adopted, with companies like Facebook achieving major performance improvements (4.5–6x CPU, 3–4x resource usage, ~5x lower latency) by switching from Hive to Spark.

# Spark SQL | Big Data and SQL: Spark SQL

## Spark's Relationship to Hive

- Spark SQL can connect to Hive metastore, making it easy to access table metadata, which is useful for users transitioning from Hadoop environments.
- This connection reduces the need for file listing when retrieving data.

## The Hive Metastore

To connect Spark to the Hive metastore, configure these properties:

- Set the metastore version (``spark.sql.hive.metastore.version``), default is `1.2.1`.
- Define metastore jars (``spark.sql.hive.metastore.jars``) if needed.
- Provide shared class prefixes (``spark.sql.hive.metastore.sharedPrefixes``) for database communication.



# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- **Creating DataFrames** 
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- Select Statements

# Spark SQL | Creating DataFrames

## What Are DataFrames?

- A **DataFrame** in **Spark SQL** is a distributed collection of data organized into named columns, similar to a table in a relational database or a data frame in Python's **pandas** or **R**.

## Features:

- Provides a higher-level abstraction than **RDDs**.
- Optimized for large-scale data processing using **Spark SQL's Catalyst Optimizer**.
- Supports a variety of data sources like **JSON, CSV, Parquet, JDBC, and HDFS**.

## Advantages:

- Rich APIs for transformations and actions.
- Easier integration with SQL queries.
- Schema enforcement ensures structured processing.

# Spark SQL | Creating DataFrames

## Creating DataFrames

### From a Collection:

- DataFrames can be created from in-memory collections like sequences or lists.

### From External Data Sources:

- DataFrames can be loaded from external sources like CSV, JSON, or Parquet files.

# Spark SQL | Creating DataFrames

## Creating DataFrames

### Example: scala

```
// Create DataFrame from a collection
val data = Seq(("Alice", 29), ("Bob", 35))
val df = spark.createDataFrame(data).toDF("Name", "Age")
```

```
+-----+-----+
| Name|Age|
+-----+-----+
|Alice| 29|
|  Bob| 35|
+-----+-----+
```

```
// Read CSV into a DataFrame
```

```
val dfFromCSV = spark.read.option("header", "true").csv("path/to/data.csv")
```

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations 
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- Select Statements

# Spark SQL | Dataframes Operations

## Operations on DataFrames

### (a) Transformations

- Transformations are operations that transform one DataFrame into another. These are **lazy operations**, meaning they don't execute until an action is called.

### Example Transformations:

- **Filtering:** Select rows based on a condition.
- **Mapping:** Apply transformations to columns.
- **Grouping:** Aggregate data by groups.

# Spark SQL | Dataframes Operations

## Operations on DataFrames

### (a) Transformations- Example (scala)

```
// Filter rows where Age > 30  
val filteredDF = df.filter($"Age" > 30)  
filteredDF.show()
```

```
+-----+-----+  
| Name | Age |  
+-----+-----+  
|  Bob |  35 |  
+-----+-----+
```

# Spark SQL | Dataframes Operations

## Operations on DataFrames

### (b) Actions

Actions are operations that trigger execution and return results to the driver or write them to an external store.

### Examples of Actions:

- `show()`: Displays the DataFrame's content.
- `collect()`: Collects all data into the driver program.
- `write()`: Saves the DataFrame to storage.



# Spark SQL | Dataframes Operations

## Operations on DataFrames

### (b) Actions- Example (scala)

```
// Collect data and print  
filteredDF.collect().foreach(println)
```

# Spark SQL | Dataframes Operations

## DataFrame Operations Examples

### (a) SQL Queries on DataFrames

- DataFrames integrate seamlessly with Spark SQL.
- We can register a DataFrame as a temporary view and run SQL queries.

// Example

```
df.createOrReplaceTempView("people")  
  
val sqlDF = spark.sql("SELECT Name, Age FROM people  
WHERE Age > 30")  
  
sqlDF.show()
```

# Spark SQL | Dataframes Operations

## DataFrame Operations Examples

### (b) Joining DataFrames

- Joins are used to combine two DataFrames based on a common column.

Example:

```
// Create another DataFrame
val data2 = Seq((29, "Engineer"), (35, "Doctor"))
val df2 = spark.createDataFrame(data2).toDF("Age",
"Profession")

// Perform an inner join
val joinedDF = df.join(df2, "Age")
joinedDF.show()
```

# Spark SQL | Dataframes Operations

## DataFrame Operations Examples

### (b) Joining DataFrames

Example: Output

```
+----+-----+-----+
|Age| Name| Profession|
+----+-----+-----+
| 29| Alice| Engineer|
| 35|  Bob|   Doctor|
+----+-----+-----+
```

# Spark SQL | Dataframes Operations

## DataFrame Operations Examples

### (c) Aggregations

- Aggregations are used to perform computations like sum, average, and count on grouped data.

Example:

```
val avgAgeDF = df.groupBy("Name").avg("Age")  
avgAgeDF.show()
```

# Spark SQL | Dataframes Operations

## Benefits of DataFrames in Spark

- **High-level Abstraction:** Makes working with structured data simpler than using RDDs.
- **Optimized Execution:** Catalyst Optimizer ensures efficient query execution.
- **Integration with SQL:** Leverage SQL-like queries for complex data analysis.
- **Interoperability:** Supports multiple data sources like HDFS, Hive, and NoSQL

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries 
- Tables
- Views
- Databases
- Select Statements

# Spark SQL | How to Run Spark SQL Queries

Spark provides several interfaces to execute SQL queries.

## 1. Spark SQL CLI (Command Line Interface)

- A tool for basic Spark SQL queries in local mode.
- To start the Spark SQL CLI, run the following in the Spark directory:

```
./bin/spark-sql
```

- Place *hive-site.xml*, *core-site.xml*, and *hdfs-site.xml* files in *conf/* for Hive configuration.

- For more options, run:

```
./bin/spark-sql --help
```



# Spark SQL | How to Run Spark SQL Queries

## 2. Spark's Programmatic SQL Interface

- SQL queries can be executed **ad hoc** through **Spark's** language APIs.
- **Example** in Python/Scala:

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

- **Multi-line SQL** example:

```
//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') """)
```

# Spark SQL | How to Run Spark SQL Queries

## 2. Spark's Programmatic SQL Interface cont'd.

- We can switch between SQL and DataFrames:
- In Scala:

```
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
```

# Spark SQL | How to Run Spark SQL Queries

## 2. Spark's Programmatic SQL Interface cont'd.

- We can switch between SQL and DataFrames:
- 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
```

# Spark SQL | How to Run Spark SQL Queries

## 3. SparkSQL Thrift JDBC/ODBC Server

- Spark provides a JDBC (Java Database Connectivity) interface for running Spark SQL queries.
- Commonly used for connecting tools like Tableau to Spark for data analysis.
- The Thrift JDBC/ODBC (Open Database Connectivity) server is compatible with HiveServer2 (Hive 1.2.1).
- We can test the JDBC server using the Beeline script included with Spark or Hive.

# Spark SQL | How to Run Spark SQL Queries

## 3. SparkSQL Thrift JDBC/ODBC Server cont'd.

- To **start** the JDBC/ODBC server:

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

- To **test** the **server** with Beeline:

```
./bin/beeline
```

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

# Spark SQL | How to Run Spark SQL Queries

## 3. SparkSQL Thrift JDBC/ODBC Server cont'd.

- 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>
  ...
```

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables 
- Views
- Databases
- Select Statements

# Spark SQL | Tables

- Tables: Essential for working with Spark SQL.
- Tables are similar to DataFrames but exist within a database.
- Tables support operations like:
  - Join
  - Filter
  - Aggregation
  - Other data manipulations
- Key difference:
  - DataFrames are defined in a programming language.
  - Tables are defined in a database (default if not specified).
- In Spark 2.X, tables always contain data.
  - No temporary tables, only views (which don't hold data).
  - Dropping a table can result in data loss.



# Spark SQL | Tables

## 1. Spark-Managed Tables

- Spark-Managed vs Unmanaged Tables:
  - **Managed tables:** Spark manages both the data and metadata.
  - **Unmanaged tables:** Only the metadata is managed by Spark, data remains in external files.
- Creating tables:
  - Use `saveAsTable` on a DataFrame to create a managed table.
  - **Unmanaged tables** are created from external files.
- Storage location:
  - Managed tables are saved in the default Hive warehouse (`/user/hive/warehouse`).
  - We can customize the location using `spark.sql.warehouse.dir`
- To see tables in a specific database, use `SHOW TABLES IN databaseName`

# Spark SQL | Tables

## 2. Creating Tables

- **Creating Tables from Various Sources:**
  - **Spark** can create tables on the fly using the **Data Source API**, no need to predefine tables.
  - **Example:** Create a table from a **JSON** file:

```
// sql
CREATE TABLE flights (
  DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')
```

# Spark SQL | Tables

## 2. Creating Tables cont'd.

- **Creating Tables from Various Sources:**
  - We can add comments to table columns for clarity:

```
// sql
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')
```

# Spark SQL | Tables

## 2. Creating Tables cont'd.

- **Creating Tables from Various Sources:**

- Create a table from a query:

```
// sql
```

```
CREATE TABLE flights_from_select USING parquet AS SELECT * FROM  
flights
```

- Create a table if it doesn't already exist:
- Partition data when writing out:

```
// sql
```

```
CREATE TABLE partitioned_flights USING parquet PARTITIONED BY  
(DEST_COUNTRY_NAME) AS SELECT DEST_COUNTRY_NAME,  
ORIGIN_COUNTRY_NAME, count FROM flights LIMIT 5
```

- **Temporary tables** do not exist; create a temporary view instead.

# Spark SQL | Tables

## 3. Creating External Tables

- **Compatibility with Hive:** Spark SQL fully supports HiveQL, allowing easy migration of Hive SQL statements to Spark SQL.
- **External Tables:** These tables are particularly useful for managing legacy data where the table metadata is handled by Spark, but the data files themselves are not managed by Spark.
- **Creating an External Table:** Simply use the CREATE EXTERNAL TABLE statement to define the schema and location of your data. This approach maintains our data's original location and format without duplicating it within Spark's managed environment.

# Spark SQL | Tables

## 3. Creating External Tables cont'd.

- We 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/'
```

- We 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
```

## 4. 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
```

We can optionally provide a **partition** specification if we want to write only into a certain partition.

**Note** that a write will respect a **partitioning scheme**, as well (which may cause the above query to run quite slowly); however, it will add additional files only into the end partitions:

```
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
```

# Spark SQL | Tables

## 5. Describing Table Metadata

- We can add a **comment** when **creating a table**.
- We can **view** this by **describing the table metadata**, which will show us the relevant comment:

```
DESCRIBE TABLE flights_csv
```

- We 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
```



# Spark SQL | Tables

## 6. Refreshing Table Metadata

- Maintaining **table metadata** is an important task to ensure that we'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 // MSCK stands for Metastore Check
```

# Spark SQL | Tables

## 7. Dropping Tables

- We cannot delete tables: we can only “drop” them.
- We can drop a table by using the `DROP` keyword.
- If we drop a managed table (e.g., `flights_csv`), both the data and the table definition will be removed:

```
DROP TABLE flights_csv;
```

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

```
DROP TABLE IF EXISTS flights_csv;
```

## Dropping unmanaged tables

- If we are dropping an unmanaged table (e.g., `hive_flights`), no data will be removed but we will no longer be able to refer to this data by the table name.

# Spark SQL | Tables

## 8. Caching Tables

- Just like DataFrames, we can cache and uncache tables.
- The CACHE TABLE command is used to load the contents of a specified table into memory, which helps speed up query performance on this data by avoiding disk reads on subsequent queries. When a table is cached, Spark SQL stores its data as an in-memory columnar format.
- We simply specify which table we would like using the following syntax:

```
CACHE TABLE flights
```

- The UNCACHE TABLE command is used to remove the table from memory. This is useful when we no longer need quick access to the table data or if memory resources are constrained and we need to clear up space that was previously occupied by cached tables.
- Here's how we uncache them:

```
UNCACHE TABLE FLIGHTS
```

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views 
- Databases
- Select Statements

# Spark SQL | Views

- 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 our query logic.
- Spark has several different notions of views.
- Views can be global, set to a database, or per session.

# Spark SQL | Views

## Creating Views

- Views allow us to transform source data at query time without rewriting it to a new location.
- For example, we can create a view to filter data for flights to the United States:

```
// sql
```

```
CREATE VIEW just_usa_view AS
```

```
SELECT * FROM flights WHERE dest_country_name = 'United  
States'
```

# Spark SQL | Views

## Creating Views cont'd.

### Temporary and Global Views

- **Temporary views** exist only for the session and are not registered in the database:

```
// sql
CREATE TEMP VIEW just_usa_view_temp AS
SELECT * FROM flights WHERE dest_country_name = 'United
States'
```

- **Global temporary views** are accessible across the **Spark** application but removed at the end of the session:

```
// sql
CREATE GLOBAL TEMP VIEW just_usa_global_view_temp AS
SELECT * FROM flights WHERE dest_country_name = 'United
States'

SHOW TABLES
```

# Spark SQL | Views

## Creating Views cont'd.

### Overwriting Views

- We can overwrite existing views, whether temporary or regular:

```
// sql
```

```
CREATE OR REPLACE TEMP VIEW just_usa_view_temp AS  
SELECT * FROM flights WHERE dest_country_name = 'United  
States'
```

- Query the view just like a table:

```
// sql
```

```
SELECT * FROM just_usa_view_temp
```

- Views act as transformations that Spark applies only when queried, similar to creating new DataFrames.



# Spark SQL | Views

## Creating Views cont'd.

### Example in DataFrames vs SQL

- **In DataFrames:**

```
// scala
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:**

```
// sql
EXPLAIN SELECT * FROM just_usa_view
```

Or, equivalently:

```
EXPLAIN SELECT * FROM flights WHERE dest_country_name = 'United
States'
```

- Both methods are equivalent, so choose the approach that best fits your workflow.

# Spark SQL | Views

## Dropping Views

- We can **drop views** in the same way that we **drop tables**; we simply specify that what we 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:

```
// sql
```

```
DROP VIEW IF EXISTS just_usa_view;
```

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases 
- Select Statements

# Spark SQL | Databases

- Databases are a **tool** for **organizing tables**.
- If we do not define one, **Spark** will use the **default database**.
- Any **SQL** statements that we run from within **Spark** (including **DataFrame** commands) execute within the context of a database. This means that if we change the database, any **user-defined tables** will remain in the **previous database** and will need to be queried differently.
- We can see **all databases** by using the following command:

```
SHOW DATABASES
```

# Spark SQL | Databases

## Creating Databases

- **Creating databases** we use the `CREATE DATABASE` keywords:

```
CREATE DATABASE some_db
```

## Setting the Database

- We 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 we 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 we are in a different database:

# Spark SQL | Databases

## Setting the Database cont'd.

- After we **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 we are in a **different database**:

```
SHOW tables
```

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

- However, we can **query different databases** by using the **correct prefix**:

```
SELECT * FROM default.flights
```

- We can see what **database** we're currently using by running the following command:

```
SELECT current_database()
```

- We can **switch back** to the **default database**:

```
USE default;
```

## Dropping Databases

- **Dropping or removing databases** is equally as easy: we simply use the `DROP DATABASE` keyword::

```
DROP DATABASE IF EXISTS some_db;
```

# Spark SQL: *Outline*

- Spark SQL Overview
- What Is SQL?
- Big Data and SQL: Spark SQL
  - Spark's Relationship to Hive
- Creating DataFrames
- Dataframes Operations
- How to Run Spark SQL Queries
- Tables
- Views
- Databases
- **Select Statements**





# Spark SQL | Select Statements

- **Queries** in **Spark** support the following **ANSI SQL** requirements (here we list the layout of the **SELECT** expression):

```
SELECT [ALL|DISTINCT] named_expression[, named_expression, ...]  
    FROM relation[, relation, ...]  
    [lateral_view[, lateral_view, ...]]  
    [WHERE boolean_expression]  
    [aggregation [HAVING boolean_expression]]  
    [ORDER BY sort_expressions]  
    [CLUSTER BY expressions]  
    [DISTRIBUTE BY expressions]  
    [SORT BY sort_expressions]  
    [WINDOW named_window[, WINDOW named_window, ...]]  
    [LIMIT num_rows]
```

## **named\_expression:**

```
: expression [AS alias]
```

## **relation:**

```
| join_relation  
| (table_name|query|relation) [sample] [AS alias]  
: VALUES (expressions)[, (expressions), ...]  
[AS (column_name[, column_name, ...])]
```

## **expressions:**

```
: expression[, expression, ...]
```

## **sort\_expressions:**

```
: expression [ASC|DESC][, expression [ASC|DESC], ...]
```

# Spark SQL | Select Statements

## case...when...then Statements

- Oftentimes, we might need to **conditionally replace values** in our SQL queries.
- We can do this by using a `case...when...then...end` style statement.
- This is essentially the equivalent of programmatic `if` statements:

```
SELECT
    CASE WHEN DEST_COUNTRY_NAME = 'UNITED STATES' THEN 1
         WHEN DEST_COUNTRY_NAME = 'Egypt' THEN 0
         ELSE -1 END
FROM partitioned_flights
```