SEAS 8515 - Lecture 5

Spark SQL and DataFrames: Interacting with External Data Sources

School of Engineering & Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin

Spark SQL and Apache Hive

- **Overview of Spark SQL:** Spark SQL is a core component of Apache Spark, blending relational processing with Spark's functional programming API. Originating from Shark, Spark SQL has evolved beyond its Hive-based roots to enhance performance and scalability.
- **Evolution from Shark to Spark SQL:** Initially built on the Hive codebase as Shark, Spark SQL evolved to provide the speed of an enterprise data warehouse with the scalability of Hive/MapReduce, moving away from Hive in its implementation in subsequent versions.
- **Capabilities of Spark SQL:** Offers Spark programmers not only faster performance and relational programming benefits such as declarative queries and optimized storage but also the ability to invoke complex analytics libraries, including machine learning.

User-Defined Functions

- **Flexibility of UDFs in Spark:** Apache Spark supports user-defined functions (UDFs), allowing data engineers and scientists to customize and extend the built-in functionalities to meet specific data processing needs.
- **Integration with Spark SQL:** UDFs can be written in PySpark or Scala and integrated directly into Spark SQL, enabling users to utilize custom functions in SQL queries for enhanced data analysis and manipulation.
- **Practical Application:** UDFs are particularly useful in scenarios where data scientists need to apply complex operations, such as wrapping a machine learning model within a UDF to allow data analysts to query model predictions directly in Spark SQL.
- **Session Scope and Persistence:** It's important to note that UDFs are session-specific and do not persist in the Spark metastore, meaning they need to be redefined in each new Spark session where they are needed.

Spark UDF

```scala
// In Scala
// Create cubed function
val cubed = (s: Long) => {
  s * s * s
}

// Register UDF
spark.udf.register("cubed", cubed)

// Create temporary view
spark.range(1, 9).createOrReplaceTempView("udf_test")
```

```python
# In Python
from pyspark.sql.types import LongType

# Create cubed function
def cubed(s):
  return s * s * s

# Register UDF
spark.udf.register("cubed", cubed, LongType())

# Generate temporary view
spark.range(1, 9).createOrReplaceTempView("udf_test")
```

Spark UDF

You can now use Spark SQL to execute either of these `cubed()` functions:

```
// In Scala/Python
// Query the cubed UDF
spark.sql("SELECT id, cubed(id) AS id_cubed FROM udf_test").show()
```

```
+---+--------+
| id|id_cubed|
+---+--------+
|  1|       1|
|  2|       8|
|  3|      27|
|  4|      64|
|  5|     125|
|  6|     216|
|  7|     343|
|  8|     512|
+---+--------+
```

Speeding up and distributing PySpark UDFs with Pandas UDFs

- **Background and Introduction:** Pandas UDFs were introduced in Apache Spark 2.3 to address performance issues with PySpark UDFs, which suffered due to expensive data transfers between the JVM and Python environments.
- **Utilization of Apache Arrow:** By incorporating Apache Arrow for data transfer, Pandas UDFs streamline the interaction between Python and Spark, significantly reducing the overhead associated with data serialization and movement.
- **Vectorized Execution:** Unlike traditional PySpark UDFs that operate row by row, Pandas UDFs work on entire Pandas Series or DataFrames, enabling vectorized operations that are inherently more efficient and faster.

Speeding up and distributing PySpark UDFs with Pandas UDFs

- **Development in Spark 3.0:** The release of Apache Spark 3.0 brought a classification of Pandas UDFs into two categories—Pandas UDFs and Pandas Function APIs—both of which leverage Python type hints to simplify and enhance UDF implementations.
- **Supported UDF Types:** In Spark 3.0, Pandas UDFs support various transformation types, including Series to Series, Iterator of Series to Iterator of Series, and Series to Scalar, allowing for a broad range of analytical operations.
- **Pandas Function APIs**: These APIs enable direct application of local Python functions to PySpark DataFrames, with supported operations such as grouped map and cogrouped map, further bridging the gap between Pandas and Spark for complex data manipulations.

Querying with the Spark SQL Shell, Beeline, and Tableau

- **Querying Options in Apache Spark:** Users can interact with Apache Spark through various tools such as the Spark SQL shell, Beeline CLI utility, and reporting tools like Tableau and Power BI, each offering unique methods to execute Spark SQL queries.
- **Using the Spark SQL Shell:** The Spark SQL shell is a convenient CLI tool located in the $SPARK_HOME folder, ideal for executing SQL queries directly. It interacts with the Hive metastore service in local mode but does not connect to the Spark Thrift Server (STS) for JDBC/ODBC queries.
- **Integration with Reporting Tools:** For visual data exploration and reporting, Spark can be connected with tools like Tableau using specific configurations outlined in the respective tool's documentation. For Power BI and other JDBC/ODBC clients, the Spark Thrift Server facilitates SQL query execution over JDBC and ODBC protocols.

External Data Sources: JDBC and SQL Databases

- **Connecting to JDBC and SQL Databases:** Spark SQL includes a data source API capable of reading data from various databases using JDBC, simplifying the process by returning results in the form of a DataFrame. This integration leverages Spark SQL's performance benefits and its capability to join data with other sources.
- **Setting Up JDBC in Spark**: To connect to a JDBC data source, specify the JDBC driver in the Spark classpath. From the $SPARK_HOME folder, use a command like ./bin/spark-shell --driver-class-path $database.jar --jars $database.jar to start Spark with the necessary driver.
- **Loading Data and Configuring Connections:** Once connected, tables from the remote database can be loaded directly into Spark as a DataFrame or as a temporary Spark SQL view. Users must specify JDBC connection properties in the data source options, with common properties detailed in Spark's documentation.

Common Connection Properties

Property name	Description
`user`, `password`	These are normally provided as connection properties for logging into the data sources.
`url`	JDBC connection URL, e.g., `jdbc:postgresql://localhost/test?user=fred&password=secret`.
`dbtable`	JDBC table to read from or write to. You can't specify the `dbtable` and `query` options at the same time.
`query`	Query to be used to read data from Apache Spark, e.g., `SELECT column1, column2, ..., columnN FROM [table|subquery]`. You can't specify the `query` and `dbtable` options at the same time.
`driver`	Class name of the JDBC driver to use to connect to the specified URL.

The importance of partitioning

- **Importance of Data Partitioning:** When dealing with large data transfers between Spark SQL and JDBC external sources, it's crucial to partition your data. This helps prevent saturation of the single driver connection which can drastically slow down data extraction and burden the source system's resources.
- **Optimizing JDBC Connectivity:** Use specific JDBC properties to manage data loads more efficiently. Although these properties are optional, they are strongly recommended for handling large-scale operations to ensure performance and system stability.
- **Managing System Load:** Proper configuration of JDBC connection properties can prevent overwhelming both the Spark and external database systems, thus maintaining optimal performance and preventing resource exhaustion.

The importance of partitioning

Property name	Description
`numPartitions`	The maximum number of partitions that can be used for parallelism in table reading and writing. This also determines the maximum number of concurrent JDBC connections.
`partitionColumn`	When reading an external source, `partitionColumn` is the column that is used to determine the partitions; note, `partitionColumn` must be a numeric, date, or timestamp column.
`lowerBound`	Sets the minimum value of `partitionColumn` for the partition stride.
`upperBound`	Sets the maximum value of `partitionColumn` for the partition stride.

PostgreSQL

- **Setting Up JDBC for PostgreSQL:** To connect Spark SQL to a PostgreSQL database, first obtain the JDBC driver (e.g., from Maven) and add it to your classpath. This setup is crucial for establishing a successful connection between Spark and PostgreSQL.
- **Starting Spark with PostgreSQL JDBC:** Launch the Spark shell with the PostgreSQL JDBC jar included by using the command: bin/spark-shell --jars postgresql-42.2.6.jar. This command ensures that Spark can interact with the PostgreSQL database through JDBC.
- **Using Spark SQL with PostgreSQL:** Demonstrated by examples, utilize the Spark SQL data source API in Scala to load data from and save data to a PostgreSQL database. This showcases practical usage of JDBC within Spark to manage database operations efficiently.

PostgreSQL – Read Option

```
# In Python
# Read Option 1: Loading data from a JDBC source using load method
jdbcDF1 = (spark
    .read
    .format("jdbc")
    .option("url", "jdbc:postgresql://[DBSERVER]")
    .option("dbtable", "[SCHEMA].[TABLENAME]")
    .option("user", "[USERNAME]")
    .option("password", "[PASSWORD]")
    .load())

  # Read Option 2: Loading data from a JDBC source using jdbc method
  jdbcDF2 = (spark
    .read
    .jdbc("jdbc:postgresql://[DBSERVER]", "[SCHEMA].[TABLENAME]",
            properties={"user": "[USERNAME]", "password": "[PASSWORD]"}))
```

PostgreSQL – Write Option

```
# Write Option 1: Saving data to a JDBC source using save method
(jdbcDF1
  .write
  .format("jdbc")
  .option("url", "jdbc:postgresql://[DBSERVER]")
  .option("dbtable", "[SCHEMA].[TABLENAME]")
  .option("user", "[USERNAME]")
  .option("password", "[PASSWORD]")
  .save())

# Write Option 2: Saving data to a JDBC source using jdbc method
(jdbcDF2
  .write
  .jdbc("jdbc:postgresql:[DBSERVER]", "[SCHEMA].[TABLENAME]",
        properties={"user": "[USERNAME]", "password": "[PASSWORD]"}))
```

MySQL

- **Acquiring the JDBC Driver for MySQL:** To connect Spark SQL to a MySQL database, obtain the JDBC driver by downloading it from MySQL's official site or Maven. This step is essential for integrating the MySQL database with Spark.
- **Launching Spark with MySQL JDBC:** Start the Spark shell by including the MySQL JDBC jar in the command: bin/spark-shell --jars mysql-connector-java_8.0.16-bin.jar. This command configures Spark to interface directly with the MySQL database through JDBC.
- **Interacting with MySQL Using Spark SQL:** Utilize the Spark SQL data source API in Scala to perform data operations, such as loading data from and saving data to a MySQL database. This demonstrates how to effectively manage data transactions between Spark and MySQL using JDBC.

MySQL – Read Option

```
# In Python
# Loading data from a JDBC source using load
jdbcDF = (spark
  .read
  .format("jdbc")
  .option("url", "jdbc:mysql://[DBSERVER]:3306/[DATABASE]")
  .option("driver", "com.mysql.jdbc.Driver")
  .option("dbtable", "[TABLENAME]")
  .option("user", "[USERNAME]")
  .option("password", "[PASSWORD]")
  .load())
```

MySQL – Write Option

```
# Saving data to a JDBC source using save
(jdbcDF
  .write
  .format("jdbc")
  .option("url", "jdbc:mysql://[DBSERVER]:3306/[DATABASE]")
  .option("driver", "com.mysql.jdbc.Driver")
  .option("dbtable", "[TABLENAME]")
  .option("user", "[USERNAME]")
  .option("password", "[PASSWORD]")
  .save())
```

Other External Sources

- **Diverse Data Source Connectivity:** Apache Spark supports integration with a wide array of external data sources beyond traditional SQL databases, enhancing its versatility in handling different types of data workloads.
- **Examples of Supported Data Sources:** Some of the popular data sources that Apache Spark can connect to include Apache Cassandra for distributed database management, Snowflake for cloud data warehousing, and MongoDB for NoSQL database solutions, as well as Microsoft SQL Server.
- **Expanding Data Operations:** This connectivity allows Spark users to perform complex data operations across diverse environments, facilitating scalable and efficient data processing and analytics across multiple platforms.

Higher-Order Functions in DataFrames and Spark SQL

- **Manipulating Complex Data Types:** Complex data types in Spark SQL and DataFrames are often handled through two main methods:
  - Exploding Nested Structures: This technique involves expanding the nested structure into individual rows, applying functions to these rows, and then reassembling them into the original nested format.
  - Creating User-Defined Functions (UDFs): This approach allows for the customization of data processing by building functions tailored to specific data manipulation needs.

Higher-Order Functions in DataFrames and Spark SQL

- **Utility Functions for Data Manipulation:** Several utility functions facilitate these operations, including get_json_object(), from_json(), to_json(), explode(), and selectExpr(). These functions are crucial for effectively transforming and querying nested or complex data structures within Spark.
- **Tabular Format Problem-Solving:** Both methods encourage thinking about data manipulation in a tabular format, simplifying the conceptual approach to handling complex data types within Spark SQL and DataFrames. This tabular perspective helps in visualizing data transformation processes and applying functions more intuitively.

Option 1: Explode and Collect

In this nested SQL statement, we first `explode(values)`, which creates a new row (with the `id`) for each element (`value`) within `values`:

```
-- In SQL
SELECT id, collect_list(value + 1) AS values
FROM  (SELECT id, EXPLODE(values) AS value
        FROM table) x
GROUP BY id
```

While `collect_list()` returns a list of objects with duplicates, the `GROUP BY` statement requires shuffle operations, meaning the order of the re-collected array isn't necessarily the same as that of the original array. As `values` could be any number of dimensions (a really wide and/or really long array) and we're doing a `GROUP BY`, this approach could be very expensive.

Option 2: User-Defined Function

To perform the same task (adding `1` to each element in `values`), we can also create a UDF that uses `map()` to iterate through each element (`value`) and perform the addition operation:

```
-- In SQL
SELECT id, collect_list(value + 1) AS values
FROM  (SELECT id, EXPLODE(values) AS value
        FROM table) x
GROUP BY id
```

We could then use this UDF in Spark SQL as follows:

```
spark.sql("SELECT id, plusOneInt(values) AS values FROM table").show()
```

While this is better than using `explode()` and `collect_list()` as there won't be any ordering issues, the serialization and deserialization process itself may be expensive. It's also important to note, however, that `collect_list()` may cause executors to experience out-of-memory issues for large data sets, whereas using UDFs would alleviate these issues.

Built-in Functions for Complex Data Types

Function/Description	Query	Output
`array_distinct(array<T>): array<T>` Removes duplicates within an array	`SELECT array_distinct(array(1, 2, 3, null, 3));`	`[1,2,3,null]`
`array_intersect(array<T>, array<T>): array<T>` Returns the intersection of two arrays without duplicates	`SELECT array_intersect(array(1, 2, 3), array(1, 3, 5));`	`[1,3]`
`array_union(array<T>, array<T>): array<T>` Returns the union of two arrays without duplicates	`SELECT array_union(array(1, 2, 3), array(1, 3, 5));`	`[1,2,3,5]`
`array_except(array<T>, array<T>): array<T>` Returns elements in `array1` but not in `array2`, without duplicates	`SELECT array_except(array(1, 2, 3), array(1, 3, 5));`	`[2]`
`array_join(array<String>, String[, String]): String` Concatenates the elements of an array using a delimiter	`SELECT array_join(array('hello', 'world'), ' ');`	`hello world`
`array_max(array<T>): T` Returns the maximum value within the array; `null` elements are skipped	`SELECT array_max(array(1, 20, null, 3));`	`20`
`array_min(array<T>): T` Returns the minimum value within the array; `null` elements are skipped	`SELECT array_min(array(1, 20, null, 3));`	`1`

Built-in Functions for Complex Data Types

Function/Description	Query	Output
`array_position(array<T>, T): Long` Returns the (1-based) index of the first element of the given array as a `Long`	`SELECT array_position(array(3, 2, 1), 1);`	`3`
`array_remove(array<T>, T): array<T>` Removes all elements that are equal to the given element from the given array	`SELECT array_remove(array(1, 2, 3, null, 3), 3);`	`[1,2,null]`
`arrays_overlap(array<T>, array<T>): array<T>` Returns `true` if `array1` contains at least one non-`null` element also present in `array2`	`SELECT arrays_overlap(array(1, 2, 3), array(3, 4, 5));`	`true`
`array_sort(array<T>): array<T>` Sorts the input array in ascending order, with null elements placed at the end of the array	`SELECT array_sort(array('b', 'd', null, 'c', 'a'));`	`["a","b","c","d",null]`
`concat(array<T>, ...): array<T>` Concatenates strings, binaries, arrays, etc.	`SELECT concat(array(1, 2, 3), array(4, 5), array(6));`	`[1,2,3,4,5,6]`

Built-in Functions for Complex Data Types

`flatten(array<array<T>>): array<T>` Flattens an array of arrays into a single array	`SELECT flatten(array(array(1, 2), array(3, 4)));`	`[1,2,3,4]`
`array_repeat(T, Int): array<T>` Returns an array containing the specified element the specified number of times	`SELECT array_repeat('123', 3);`	`["123","123","123"]`
`reverse(array<T>): array<T>` Returns a reversed string or an array with the reverse order of elements	`SELECT reverse(array(2, 1, 4, 3));`	`[3,4,1,2]`

`sequence(T, T[, T]): array<T>`
Generates an array of elements from start to stop (inclusive) by incremental step
	`SELECT sequence(1, 5);`
	`SELECT sequence(5, 1);`
	`SELECT sequence(to_date('2018-01-01'), to_date('2018-03-01'), interval 1 month);`
		`[1,2,3,4,5]`
		`[5,4,3,2,1]`
		`["2018-01-01", "2018-02-01", "2018-03-01"]`

`shuffle(array<T>): array<T>` Returns a random permutation of the given array	`SELECT shuffle(array(1, 20, null, 3));`	`[null,3,20,1]`

Built-in Functions for Complex Data Types

Function/Description	Query	Output
`slice(array<T>, Int, Int): array<T>` Returns a subset of the given array starting from the given index (counting from the end if the index is negative), of the specified length	`SELECT slice(array(1, 2, 3, 4), -2, 2);`	`[3,4]`
`array_zip(array<T>, array<U>, ...): array<struct<T, U, ...>>` Returns a merged array of structs	`SELECT arrays_zip(array(1, 2), array(2, 3), array(3, 4));`	`[{"0":1,"1":2,"2":3}, {"0":2,"1":3,"2":4}]`
`element_at(array<T>, Int): T /` Returns the element of the given array at the given (1-based) index	`SELECT element_at(array(1, 2, 3), 2);`	`2`
`cardinality(array<T>): Int` An alias of `size`; returns the size of the given array or a map	`SELECT cardinality(array('b', 'd', 'c', 'a'));`	`4`

Map Functions

Function/Description	Query	Output
`map_form_arrays(array<K>, array<V>): map<K, V>` Creates a map from the given pair of key/value arrays; elements in keys should not be `null`	`SELECT map_from_arrays(array(1.0, 3.0), array('2', '4'));`	`{"1.0":"2", "3.0":"4"}`
`map_from_entries(array<struct<K, V>>): map<K, V>` Returns a map created from the given array	`SELECT map_from_entries(array(struct(1, 'a'), struct(2, 'b')));`	`{"1":"a", "2":"b"}`
`map_concat(map<K, V>, ...): map<K, V>` Returns the union of the input maps	`SELECT map_concat(map(1, 'a', 2, 'b'), map(2, 'c', 3, 'd'));`	`{"1":"a", "2":"c","3":"d"}`
`element_at(map<K, V>, K): V` Returns the value of the given key, or `null` if the key is not contained in the map	`SELECT element_at(map(1, 'a', 2, 'b'), 2);`	`b`
`cardinality(array<T>): Int` An alias of `size`; returns the size of the given array or a map	`SELECT cardinality(map(1, 'a', 2, 'b'));`	`2`

Higher Order Functions

transform()

```
transform(array<T>, function<T, U>): array<U>
```

The `transform()` function produces an array by applying a function to each element of the input array (similar to a `map()` function):

```
// In Scala/Python
// Calculate Fahrenheit from Celsius for an array of temperatures
spark.sql("""
SELECT celsius,
 transform(celsius, t -> ((t * 9) div 5) + 32) as fahrenheit
  FROM tC
""").show()
```

```
+--------------------+--------------------+
|             celsius|          fahrenheit|
+--------------------+--------------------+
|[35, 36, 32, 30, ...|[95, 96, 89, 86, ...|
|[31, 32, 34, 55, 56]|[87, 89, 93, 131,...|
+--------------------+--------------------+
```

Higher Order Functions

filter()

```
filter(array<T>, function<T, Boolean>): array<T>
```

The `filter()` function produces an array consisting of only the elements of the input array for which the Boolean function is `true`:

```
// In Scala/Python
// Filter temperatures > 38C for array of temperatures
spark.sql("""
SELECT celsius,
 filter(celsius, t -> t > 38) as high
  FROM tC
""").show()

+--------------------+--------+
|             celsius|    high|
+--------------------+--------+
|[35, 36, 32, 30, ...|[40, 42]|
|[31, 32, 34, 55, 56]|[55, 56]|
+--------------------+--------+
```

Higher Order Functions

exists()

```
exists(array<T>, function<T, V, Boolean>): Boolean
```

The `exists()` function returns `true` if the Boolean function holds for any element in the input array:

```
// In Scala/Python
// Is there a temperature of 38C in the array of temperatures
spark.sql("""
SELECT celsius,
       exists(celsius, t -> t = 38) as threshold
  FROM tC
""").show()

+--------------------+---------+
|             celsius|threshold|
+--------------------+---------+
|[35, 36, 32, 30, ...|     true|
```

Higher Order Functions

reduce()

```
reduce(array<T>, B, function<B, T, B>, function<B, R>)
```

The `reduce()` function reduces the elements of the array to a single value by merging the elements into a buffer `B` using `function<B, T, B>` and applying a finishing `function<B, R>` on the final buffer:

```
// In Scala/Python
// Calculate average temperature and convert to F
spark.sql("""
SELECT celsius,
       reduce(
          celsius,
          0,
          (t, acc) -> t + acc,
          acc -> (acc div size(celsius) * 9 div 5) + 32
        ) as avgFahrenheit
  FROM tC
""").show()
```

```
+--------------------+-------------+
|             celsius|avgFahrenheit|
+--------------------+-------------+
|[35, 36, 32, 30, ...|           96|
|[31, 32, 34, 55, 56]|          105|
+--------------------+-------------+
```

Common DataFrames and Spark SQL Operations

- **Extensive DataFrame Operations:** Spark SQL supports a wide array of DataFrame operations, enhancing its functionality and flexibility. These operations encompass:
  - Functional Categories: Aggregate, collection, datetime, math, miscellaneous, non-aggregate, sorting, and string functions.
  - Specialized Functions: User-defined functions (UDFs) and window functions.
- **Comprehensive Documentation:** For a detailed understanding of all available functions and operations, users are encouraged to consult the Spark SQL documentation. This resource provides exhaustive information and examples for each type of function.
- **Focus on Relational Operations:** Within the broader context of DataFrame operations, this chapter specifically delves into common relational operations such as unions and joins, windowing techniques, and data modifications. These operations are critical for advanced data manipulation and analysis in Spark SQL.

Windowing

- **Concept of Window Functions:** Window functions use values from rows within a defined window to return values, typically as another row, allowing operations across a group of rows while returning a single value for each input row.
- **Example Function:** We demonstrate the use of the dense_rank() window function, which is used to assign a rank to each row within a partition of a result set, with no gaps in rank values between rows.
- **Versatility of Window Functions:** Besides dense_rank(), Spark SQL supports a variety of other window functions for diverse analytical tasks, enhancing the toolset for detailed data analysis.

Window functions

	SQL	DataFrame API
Ranking functions	`rank()`	`rank()`
	`dense_rank()`	`denseRank()`
	`percent_rank()`	`percentRank()`
	`ntile()`	`ntile()`
	`row_number()`	`rowNumber()`
Analytic functions	`cume_dist()`	`cumeDist()`
	`first_value()`	`firstValue()`
	`last_value()`	`lastValue()`
	`lag()`	`lag()`
	`lead()`	`lead()`

Modifications

- **Immutability of DataFrames:** Although DataFrames themselves are immutable, meaning they cannot be altered once created, modifications can be made through operations that generate new DataFrames with different structures or columns.
- **Underlying RDD Immutability:** The immutability of the underlying Resilient Distributed Datasets (RDDs) ensures data lineage and integrity during Spark operations, facilitating reliable and traceable data transformations.
- **DataFrame Transformations:** Modifications to DataFrames are achieved by applying transformation operations that create new DataFrames, allowing for adjustments in data structure and content without altering the original data source.

Modifications - Example

```
// In Scala/Python
foo.show()
```

```
--------+-----+--------+------+-----------+
|    date|delay|distance|origin|destination|
+--------+-----+--------+------+-----------+
|01010710|   31|     590|   SEA|        SFO|
|01010955|  104|     590|   SEA|        SFO|
|01010730|    5|     590|   SEA|        SFO|
+--------+-----+--------+------+-----------+
```

Modifications - Example

Adding new columns

To add a new column to the `foo` DataFrame, use the `withColumn()` method:

```
// In Scala
import org.apache.spark.sql.functions.expr
val foo2 = foo.withColumn(
               "status",
               expr("CASE WHEN delay <= 10 THEN 'On-time' ELSE 'Delayed' END")
            )

# In Python
from pyspark.sql.functions import expr
foo2 = (foo.withColumn(
          "status",
          expr("CASE WHEN delay <= 10 THEN 'On-time' ELSE 'Delayed' END")
        ))
```

Modifications - Example

The newly created `foo2` DataFrame has the contents of the original `foo` DataFrame plus the additional `status` column defined by the `CASE` statement:

```
// In Scala/Python
foo2.show()
```

```
+--------+-----+--------+------+-----------+-------+
|    date|delay|distance|origin|destination| status|
+--------+-----+--------+------+-----------+-------+
|01010710|   31|     590|   SEA|        SFO|Delayed|
|01010955|  104|     590|   SEA|        SFO|Delayed|
|01010730|    5|     590|   SEA|        SFO|On-time|
+--------+-----+--------+------+-----------+-------+
```

Modifications - Example

Dropping columns

To drop a column, use the `drop()` method. For example, let's remove the `delay` column as we now have a `status` column, added in the previous section:

```
// In Scala
val foo3 = foo2.drop("delay")
foo3.show()
```

```
# In Python
foo3 = foo2.drop("delay")
foo3.show()
```

```
+--------+--------+------+-----------+-------+
|    date|distance|origin|destination| status|
+--------+--------+------+-----------+-------+
|01010710|     590|   SEA|        SFO|Delayed|
|01010955|     590|   SEA|        SFO|Delayed|
|01010730|     590|   SEA|        SFO|On-time|
+--------+--------+------+-----------+-------+
```

Modifications - Example

Renaming columns

You can rename a column using the `rename()` method:

```
// In Scala
val foo4 = foo3.withColumnRenamed("status", "flight_status")
foo4.show()

# In Python
foo4 = foo3.withColumnRenamed("status", "flight_status")
foo4.show()
```

```
+--------+--------+------+-----------+-------------+
|    date|distance|origin|destination|flight_status|
+--------+--------+------+-----------+-------------+
|01010710|     590|   SEA|        SFO|      Delayed|
|01010955|     590|   SEA|        SFO|      Delayed|
|01010730|     590|   SEA|        SFO|      On-time|
+--------+--------+------+-----------+-------------+
```

SEAS 8515

Next Class:

Delta Lake & Spark Structured Streaming

School of Engineering
& Applied Science
THE GEORGE WASHINGTON UNIVERSITY

Dr. Adewale Akinfaderin