Introduction to PySpark (Data Manipulation)

Updating Columns

In this chapter, you'll learn how to use the methods defined by Spark's `DataFrame` class to perform common data operations.

Let's look at performing column-wise operations. In Spark you can do this using the `.withColumn()` method, which takes two arguments. First, a string with the name of your new column, and second the new column itself.

The new column must be an object of class `Column`. Creating one of these is as easy as extracting a column from your DataFrame using `df.colName`.

Updating a Spark DataFrame is somewhat different than working in `pandas` because the Spark DataFrame is *immutable*. This means that it can't be changed, and so columns can't be updated in place.

Thus, all these methods return a new DataFrame. To overwrite the original DataFrame you must reassign the returned DataFrame using the method like so:

```
df = df.withColumn("newCol", df.oldCol + 1)
```

The above code creates a DataFrame with the same columns as `df` plus a new column, `newCol`, where every entry is equal to the corresponding entry from `oldCol`, plus one.

To overwrite an existing column, just pass the name of the column as the first argument!

Remember, a `SparkSession` called `spark` is already in your workspace.

```
# Create the DataFrame flights
flights = spark.table('flights')

# Show the head
flights.show()
```

```
6
7 # Add duration_hrs
8 flights = flights.withColumn("duration_hrs", flights.air_time/60)
```

Filtering Data

Now that you have a bit of SQL know-how under your belt, it's easier to talk about the analogous operations using Spark DataFrames.

Let's take a look at the `.filter()` method. As you might suspect, this is the Spark counterpart of SQL's `WHERE` clause. The `.filter()` method takes either an expression that would follow the `WHERE` clause of a SQL expression as a string, or a Spark Column of boolean (`True` / `False`) values.

For example, the following two expressions will produce the same output:

```
flights.filter("air_time > 120").show()
flights.filter(flights.air_time > 120).show()
```

Notice that in the first case, we pass a *string* to `.filter()` . In SQL, we would write this filtering task as `SELECT * FROM flights WHERE air_time > 120` . Spark's `.filter()` can accept any expression that could go in the `WHERE` clause of a SQL query (in this case, `"air_time > 120"`), as long as it is passed as a string. Notice that in this case, we do not reference the name of the table in the string -- as we wouldn't in the SQL request.

In the second case, we actually pass a *column of boolean values* to `.filter()` . Remember that `flights.air_time > 120` returns a column of boolean values that has `True` in place of those records in `flights.air_time` that are over 120, and `False` otherwise.

Remember, a `SparkSession` called `spark` is already in your workspace, along with the Spark DataFrame `flights` .

```
1 # Filter flights by passing a string
2 long_flights1 = flights.filter("distance > 1000")
3
4 # Filter flights by passing a column of boolean values
5 long_flights2 = flights.filter(flights.distance > 1000)
6
7 # Print the data to check they're equal
8 print(long_flights1.show())
9 print(long_flights2.show())
```

Selecting and Filter

The Spark variant of SQL's `SELECT` is the `.select()` method. This method takes multiple arguments - one for each column you want to select. These arguments can either be the column name as a string (one for each column) or a column object (using the `df.colName` syntax). When you pass a column object, you can perform operations like addition or subtraction on the column to change the data contained in it, much like inside `.withColumn()` .

The difference between `.select()` and `.withColumn()` methods is that `.select()` returns only the columns you specify, while `.withColumn()` returns all the columns of the DataFrame in addition to the one you defined. It's often a good idea to drop columns you don't need at the beginning of an operation so that you're not dragging around extra data as you're wrangling. In this case, you would use `.select()` and not `.withColumn()` .

Remember, a SparkSession called `spark` is already in your workspace, along with the Spark DataFrame `flights` .

```
# Select the first set of columns
selected1 = flights.select('tailnum', 'origin', 'dest')

# Select the second set of columns
temp = flights.select(flights.origin, flights.dest, flights.carrier)

# Define first filter
filterA = flights.origin == "SEA"

# Define second filter
filterB = flights.dest == "PDX"

# Filter the data, first by filterA then by filterB
selected2 = temp.filter(filterA).filter(filterB)
```

Similar to SQL, you can also use the `.select()` method to perform column-wise operations. When you're selecting a column using the `df.colName` notation, you can perform any column operation and the `.select()` method will return the transformed column. For example,

```
flights.select(flights.air_time/60)
```

returns a column of flight durations in hours instead of minutes. You can also use the `.alias()` method to rename a column you're selecting. So if you wanted to `.select()` the column `duration_hrs` (which isn't in your DataFrame) you could do

```
flights.select((flights.air_time/60).alias("duration_hrs"))
```

The equivalent Spark DataFrame method `.selectExpr()` takes SQL expressions as a string:

```
flights.selectExpr("air_time/60 as duration_hrs")
```

with the SQL `as` keyword being equivalent to the `.alias()` method. To select multiple columns, you can pass multiple strings.

Remember, a `SparkSession` called `spark` is already in your workspace, along with the Spark DataFrame `flights` .

```
# Define avg_speed
avg_speed = (flights.distance/(flights.air_time/60)).alias("avg_speed")

# Select the correct columns
speed1 = flights.select("origin", "dest", "tailnum", avg_speed)

# Create the same table using a SQL expression
speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air_time/60) as
avg_speed")
```

Aggregating

All of the common aggregation methods, like `.min()` , `.max()` , and `.count()` are `GroupedData` methods. These are created by calling the `.groupBy()` DataFrame method. You'll learn exactly what that means in a few exercises. For now, all you have to do to use these functions is call that method on your DataFrame. For example, to find the minimum value of a column, `col` , in a DataFrame, `df` , you could do

```
df.groupBy().min("col").show()
```

This creates a `GroupedData` object (so you can use the `.min()` method), then finds the minimum value in `col` , and returns it as a DataFrame.

Now you're ready to do some aggregating of your own!

A `SparkSession` called `spark` is already in your workspace, along with the Spark DataFrame `flights` .

```
# Find the shortest flight from PDX in terms of distance
flights.filter(flights.origin == 'PDX').groupBy().min('distance').show()

# Find the longest flight from SEA in terms of air time
flights.filter(flights.origin == 'SEA').groupBy().max('air_time').show()

# Average duration of Delta flights
flights.filter(flights.carrier == 'DL').filter(flights.origin ==
'SEA').groupBy().avg('air_time').show()

# Total hours in the air
flights.withColumn("duration_hrs", flights.air_time/60).groupBy().sum('duration_hrs').show()
```

Grouping and Aggregating

Part of what makes aggregating so powerful is the addition of groups. PySpark has a whole class devoted to grouped data frames: `pyspark.sql.GroupedData`, which you saw in the last two exercises.

You've learned how to create a grouped DataFrame by calling the `.groupBy()` method on a DataFrame with no arguments.

Now you'll see that when you pass the name of one or more columns in your DataFrame to the `.groupBy()` method, the aggregation methods behave like when you use a `GROUP BY` statement in a SQL query!

Remember, a `SparkSession` called `spark` is already in your workspace, along with the Spark DataFrame `flights`.

```
# Group by tailnum
by_plane = flights.groupBy("tailnum")

# Number of flights each plane made
by_plane.count().show()

# Group by origin
by_origin = flights.groupBy("origin")

# Average duration of flights from PDX and SEA
by_origin.avg("air_time").show()
```

In addition to the `GroupedData` methods you've already seen, there is also the `.agg()` method. This method lets you pass an aggregate column expression that uses any of the aggregate functions from the `pyspark.sql.functions` submodule.

This submodule contains many useful functions for computing things like standard deviations. All the aggregation functions in this submodule take the name of a column in a `GroupedData` table.

Remember, a `SparkSession` called `spark` is already in your workspace, along with the Spark DataFrame `flights`. The grouped DataFrames you created in the last exercise are also in your workspace.

```
# Import pyspark.sql.functions as F
import pyspark.sql.functions as F

# Group by month and dest
by_month_dest = flights.groupBy('month', 'dest')

# Average departure delay by month and destination
by_month_dest.avg('dep_delay').show()

# Standard deviation of departure delay
by_month_dest.agg(F.stddev('dep_delay')).show()
```

Joining

In PySpark, joins are performed using the DataFrame method `.join()`. This method takes three arguments. The first is the second DataFrame that you want to join with the first one. The second argument, `on`, is the name of the key column(s) as a string. The names of the key column(s) must be the same in each table. The third argument, `how`, specifies the kind of join to perform. In this course we'll always use the value `how="leftouter"`.

The `flights` dataset and a new dataset called `airports` are already in your workspace.

```
# Examine the data
print(airports.show())

# Rename the faa column
airports = airports.withColumnRenamed('faa', 'dest')

# Join the DataFrames
flights_with_airports = flights.join(airports, on = 'dest', how = 'leftouter')

# Examine the new DataFrame
print(flights_with_airports.show())
```