Spark DataFrames

Introduction

You've now explored how to perform operations on Spark RDDs for simple MapReduce tasks. This is useful for contexts where the low-level Unstructured API is most appropriate, but now we're going to move on to using a more intuitive and powerful interface: Spark DataFrames!

Objectives

You will be able to:

- Load and manipulate data using Spark SQL DataFrames
- Describe the similarities and differences between RDDs, Spark SQL DataFrames, and pandas DataFrames

Spark SQL DataFrames

From the [Spark SQL docs (https://spark.apache.org/docs/latest/sql-programming-guide.html)](https://spark.apache.org/docs/latest/sql-programming-guide.html):

> Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed.

Spark SQL has both a SQL interface and a DataFrame interface. We will primarily use the DataFrame interface but it's useful to be aware of both.

Understanding SparkSession

In the previous lessons, we were using the Unstructured API and therefore we connected to Spark using a SparkContext. Here, we will be using SparkSession instead ([documentation here (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html)](https://spark.apache.org/docs/latest/api/python/pyspark.sql.html)), which is actually wrapped around a SparkContext under the hood. SparkSession is designed for interacting with high-level Spark SQL data structures (e.g. DataFrames) whereas SparkContext is design with interacting with low-level Spark Core data structures (e.g. RDDs).

A SparkSession is created using a *builder* pattern ([documentation here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html)](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.html)) and its conventional name is `spark` :

In [1]:
```python
from pyspark.sql import SparkSession
spark = SparkSession.builder.master('local').getOrCreate()
```

Creating a Spark SQL DataFrame with PySpark

Now that we have a SparkSession, we can create a DataFrame using the `createDataFrame` method ([documentation here (https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html)](https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.SparkSession.createDataFrame.html))! One way to do this would just be to hard-code the data using built-in Python types:

In [2]:
```python
spark_df = spark.createDataFrame([
    {"color": "red", "number": 4, "count": 1, "valid": True},
    {"color": "blue", "number": 7, "count": 2, "valid": False},
    {"color": "green", "number": 1, "count": 3, "valid": True}
])
spark_df
```

Out[2]:
```
DataFrame[color: string, count: bigint, number: bigint, valid: boolean]
```

When the notebook displays `spark_df` , it is showing the schema between the square brackets -- in other words, it is displaying the column names and data types. We did not specify explicit data types for the columns, so Spark inferred them for us.

You can view a nice print-out of the schema like this:

In [3]:
```python
spark_df.printSchema()
```

```
root
 |-- color: string (nullable = true)
 |-- count: long (nullable = true)
 |-- number: long (nullable = true)
 |-- valid: boolean (nullable = true)
```

Basic Features of Spark SQL DataFrames

RDD Methods

Many of the familiar RDD methods will work with DataFrames.

For example, `.collect()` to load all of the data:

```
In [4]: spark_df.collect()
```

```
Out[4]: [Row(color='red', count=1, number=4, valid=True),
         Row(color='blue', count=2, number=7, valid=False),
         Row(color='green', count=3, number=1, valid=True)]
```

And `.take(n)` to return `n` rows of data:

```
In [5]: spark_df.take(2)
```

```
Out[5]: [Row(color='red', count=1, number=4, valid=True),
         Row(color='blue', count=2, number=7, valid=False)]
```

DataFrame-Specific Methods

In addition to the methods that work on RDDs, there are methods specific to DataFrames.

For example, we can view all of the data in a tabular format using `.show()` :

```
In [6]: spark_df.show()
```

```
+-----+-----+------+-----+
|color|count|number|valid|
+-----+-----+------+-----+
|  red|    1|     4| true|
| blue|    2|     7|false|
|green|    3|     1| true|
+-----+-----+------+-----+
```

(We will add a `.show()` to the end of most of the following examples, since it's easier to read that way.)

If we want to select the data from one or more specific columns, we can use `.select()` :

```
In [7]: spark_df.select("number").show()
```

```
+------+
|number|
+------+
|     4|
|     7|
|     1|
+------+
```

```
In [8]: spark_df.select(["number", "valid"]).show()
```

```
+------+-----+
|number|valid|
+------+-----+
|     4| true|
|     7|false|
|     1| true|
+------+-----+
```

Familiar Techniques for Pandas Developers

There are also several attributes and methods that work very similarly for Spark SQL DataFrames as they do with pandas DataFrames:

```
In [9]: spark_df.columns
```

```
Out[9]: ['color', 'count', 'number', 'valid']
```

```
In [10]: spark_df.describe().show()
```

```
+-------+-----+-----+------+
|summary|color|count|number|
+-------+-----+-----+------+
|  count|    3|    3|     3|
|   mean| null|  2.0|   4.0|
| stddev| null|  1.0|   3.0|
|    min| blue|    1|     1|
|    max|  red|    3|     7|
+-------+-----+-----+------+
```

Loading the Forest Fire Dataset

For this example, we're going to be using the [Forest Fire dataset (https://archive.ics.uci.edu/ml/datasets/Forest+Fires)](https://archive.ics.uci.edu/ml/datasets/Forest+Fires) from UCI, which contains data about the area burned by wildfires in the Northeast region of Portugal in relation to numerous other factors.

We'll use `spark.read.csv` to load in the dataset:

In [11]:
```
fire_df = spark.read.csv("forestfires.csv", header="true", inferSchema="true")
fire_df.show(5)
```

```
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
|  X|  Y|month|day|FFMC| DMC|   DC|ISI|temp| RH|wind|rain|area|
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
|  7|  5|  mar|fri|86.2|26.2| 94.3|5.1| 8.2| 51| 6.7| 0.0| 0.0|
|  7|  4|  oct|tue|90.6|35.4|669.1|6.7|18.0| 33| 0.9| 0.0| 0.0|
|  7|  4|  oct|sat|90.6|43.7|686.9|6.7|14.6| 33| 1.3| 0.0| 0.0|
|  8|  6|  mar|fri|91.7|33.3| 77.5|9.0| 8.3| 97| 4.0| 0.2| 0.0|
|  8|  6|  mar|sun|89.3|51.3|102.2|9.6|11.4| 99| 1.8| 0.0| 0.0|
+---+---+-----+---+----+----+-----+---+----+---+----+----+----+
only showing top 5 rows
```

Spark DataFrame Aggregations

Let's investigate to see if there is any relationship between what month it is and the area of fire.

First we'll group by the `month` column, then aggregate based on the mean of the `area` column for that group:

In [12]:
```
fire_df_months = fire_df.groupBy('month').agg({'area': 'mean'})
fire_df_months
```

Out[12]:
```
DataFrame[month: string, avg(area): double]
```

Notice how the grouped DataFrame is not returned when you call the aggregation method. Remember, this is still Spark! The transformations and actions are kept separate so that it is easier to manage large quantities of data. You can perform the transformation by calling `.collect()` :

In [13]:
```
fire_df_months.collect()
```

Out[13]:
```
[Row(month='jun', avg(area)=5.841176470588234),
 Row(month='aug', avg(area)=12.489076086956521),
 Row(month='may', avg(area)=19.24),
 Row(month='feb', avg(area)=6.275),
 Row(month='sep', avg(area)=17.942616279069753),
 Row(month='mar', avg(area)=4.356666666666667),
 Row(month='oct', avg(area)=6.638),
 Row(month='jul', avg(area)=14.3696875),
 Row(month='nov', avg(area)=0.0),
 Row(month='apr', avg(area)=8.891111111111112),
 Row(month='dec', avg(area)=13.33),
 Row(month='jan', avg(area)=0.0)]
```

Let's show that as a table instead, and also order by the average area of fire:

In [14]:
```
fire_df_months.orderBy("avg(area)").show()
```

```
+-----+------------------+
|month|         avg(area)|
+-----+------------------+
|  jan|               0.0|
|  nov|               0.0|
|  mar| 4.356666666666667|
|  jun| 5.841176470588234|
|  feb|             6.275|
|  oct|             6.638|
|  apr| 8.891111111111112|
|  aug|12.489076086956521|
|  dec|             13.33|
|  jul|        14.3696875|
|  sep|17.942616279069753|
|  may|             19.24|
+-----+------------------+
```

As you can see, there seem to be larger area fires during what would be considered the summer months in Portugal. On your own, practice more aggregations and manipulations that you might be able to perform on this dataset.

In [ ]:

In [ ]:

In [ ]:

Boolean Masking

Boolean masking also works with PySpark DataFrames just like Pandas DataFrames, the only difference being that the `.filter()` method is used in PySpark. To try this out, let's compare the amount of fire in those areas with absolutely no rain to those areas that had rain.

In [15]:
```
no_rain = fire_df.filter(fire_df['rain'] == 0.0)
some_rain = fire_df.filter(fire_df['rain'] > 0.0)
```

Now, to perform calculations to find the mean of a column (without aggregating first), we'll have to import functions from `pyspark.sql`. As always, to read more about them, check out the documentation (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#module-pyspark.sql.functions).

In [16]:
```
from pyspark.sql.functions import mean

print('no rain fire area:')
no_rain.select(mean('area')).show()

print('some rain fire area:')
some_rain.select(mean('area')).show()
```

```
no rain fire area:
+------------------+
|         avg(area)|
+------------------+
|13.023693516699408|
+------------------+

some rain fire area:
+---------+
|avg(area)|
+---------+
|  1.62375|
+---------+
```

Yes there's definitely something there! Unsurprisingly, rain plays in a big factor in the spread of wildfire.

Let's obtain data from only the summer months in Portugal (June, July, and August). We can also do the same for the winter months in Portugal (December, January, February).

In [17]:
```
summer_months = fire_df.filter(fire_df['month'].isin(['jun','jul','aug']))
winter_months = fire_df.filter(fire_df['month'].isin(['dec','jan','feb']))

print('summer months fire area:')
summer_months.select(mean('area')).show()

print('winter months fire area')
winter_months.select(mean('area')).show()
```

```
summer months fire area:
+------------------+
|         avg(area)|
+------------------+
|12.262317596566525|
+------------------+

winter months fire area
+-----------------+
|        avg(area)|
+-----------------+
|7.918387096774193|
+-----------------+
```

Comparison between DataFrames

Although Spark SQL DataFrames and pandas DataFrames have some features in common, they are not the same. We'll demonstrate some similarities and differences below.

First, we'll create a `pandas_df` with the same data as `spark_df` and compare the two:

In [18]:
```python
import pandas as pd

pandas_df = pd.DataFrame([
    {"color": "red", "number": 4, "count": 1, "valid": True},
    {"color": "blue", "number": 7, "count": 2, "valid": False},
    {"color": "green", "number": 1, "count": 3, "valid": True}
])
```

Displaying Data

In [19]:
```python
pandas_df
```

Out[19]:

	color	number	count	valid
0	red	4	1	True
1	blue	7	2	False
2	green	1	3	True

In [20]:
```python
spark_df
```

Out[20]:
```
DataFrame[color: string, count: bigint, number: bigint, valid: boolean]
```

One difference you'll notice immediately, which has been pointed out a couple times in this lesson already, is that a Spark DataFrame loads lazily but a pandas DataFrame does not. Therefore if we just type the name of the variable we see at least a preview of the pandas data, but no preview of the Spark data.

Attributes

In [21]:
```python
pandas_df.dtypes
```

Out[21]:
```
color     object
number     int64
count      int64
valid       bool
dtype: object
```

In [22]:
```python
spark_df.dtypes
```

Out[22]:
```
[('color', 'string'),
 ('count', 'bigint'),
 ('number', 'bigint'),
 ('valid', 'boolean')]
```

Sometimes a Spark DataFrame will have an attribute with a familiar name, but it won't work quite the same way. For example, the `dtypes` attribute is present for both, but in pandas it returns a Series and in Spark SQL it returns a list of tuples.

The actual data types listed are also different, although they correspond to each other. For example, the `object` data type in pandas corresponds to the `string` data type in Spark.

Methods

As mentioned previously, Spark SQL DataFrames have methods related to their underlying RDD data structures. Pandas DataFrames do not have these methods.

Even when the methods have the same name, they might not behave the same way:

In [23]:
```python
pandas_df.corr()
```

Out[23]:

	number	count	valid
number	1.000000	-0.5	-0.866025
count	-0.500000	1.0	0.000000
valid	-0.866025	0.0	1.000000

In [24]:
```python
spark_df.corr("number", "count")
```

Out[24]:
```
-0.5
```

In the example above, both have a `corr` method that is used for computing correlations between columns, but they work fairly differently.

- The pandas method does not require any arguments and returns an entire DataFrame showing the correlations between all numeric variables (including `valid`, which contains booleans).
- The Spark SQL method requires that you specify two column names and returns a single floating point number indicating the correlation between those two columns.

Watch out for distinctions like this! And don't hesitate to read through the Spark SQL documentation (https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#spark-sql) when you're trying out a new method.

Selecting Columns and Boolean Masking

In pandas, you can select the data in a single column and it will be viewable as a Series:

In [25]:
```
pandas_df["color"]
```

Out[25]:
```
0      red
1     blue
2    green
Name: color, dtype: object
```

We can apply a method to that Series to get back a Series of booleans:

In [26]:
```
pandas_df["color"].str.contains("r")
```

Out[26]:
```
0     True
1    False
2     True
Name: color, dtype: bool
```

And then we can also use that Series of booleans to filter the DataFrame using boolean masking:

In [27]:
```
pandas_df[pandas_df["color"].str.contains("r")]
```

Out[27]:

	color	number	count	valid
0	red	4	1	True
2	green	1	3	True

In Spark, the intermediate Column data is not viewable and is only useful for applying the boolean mask.

First we select the data in the `color` column, but if we try to `show()` the result, we get an error:

In [28]:
```
spark_df["color"]
```

Out[28]:
```
Column<'color'>
```

In [29]:
```
try:
    spark_df["color"].show()
except Exception as e:
    print(type(e))
    print(e)
```

```
<class 'TypeError'>
'Column' object is not callable
```

We can chain a `contains` method call onto the Column, but again, if we try to `show()` it, we get an error:

In [30]:
```
spark_df["color"].contains("r")
```

Out[30]:
```
Column<'contains(color, r)'>
```

In [31]:
```
try:
    spark_df["color"].contains("r").show()
except Exception as e:
    print(type(e))
    print(e)
```

```
<class 'TypeError'>
'Column' object is not callable
```

If we want data we can show using this boolean mask, we have to apply the `filter` method:

In [32]:
```
spark_df.filter(spark_df["color"].contains("r"))
```

Out[32]:
```
DataFrame[color: string, count: bigint, number: bigint, valid: boolean]
```

In [33]:
```
spark_df.filter(spark_df["color"].contains("r")).show()
```

```
+-----+-----+------+-----+
|color|count|number|valid|
+-----+-----+------+-----+
|  red|    1|     4| true|
|green|    3|     1| true|
+-----+-----+------+-----+
```

Converting Between DataFrame Types

In general, it is not particularly efficient to convert from one DataFrame type to another. Usually the reason that you are using a Spark SQL DataFrame is that you are working with Big Data and require computational optimization, so it wouldn't make much sense to convert it to a pandas DataFrame.

However you can imagine some specific circumstances where you might want to use pandas for debugging, visualization, or some other task that just isn't working in Spark SQL. If you need to do that, check out this documentation (https://spark.apache.org/docs/latest/api/python/user_guide/sql/arrow_pandas.html).

Stop the SparkSession

In [34]:
```
spark.stop()
```

Summary

In this lesson, you explored Spark SQL DataFrames and their methods for displaying and manipulating data. You created DataFrames from base Python data structures as well as from a CSV file, and performed selection, filtering, and aggregation tasks along the way. Finally, you learned about the similarities and differences between Spark SQL DataFrames, their underlying RDDs, and pandas DataFrames.