Introduction

Today we will learn more about how to work with Spark and Spark DataFrames using its Python API—PySpark. It gives us the capability to process petabyte-scale data, but also implements machine learning (ML) algorithms at petabyte scale in real time. This chapter will focus on the data processing part using Spark DataFrames in PySpark.

Note:

We will be using the term DataFrame quite frequently during this chapter. This will explicitly refer to the Spark DataFrame, unless mentioned otherwise. Please do not confuse this with the pandas DataFrame.

Spark DataFrames are a distributed collection of data organized as named columns. They are inspired from R and Python DataFrames and have complex optimizations at the backend that make them fast, optimized, and scalable.

The DataFrame API was developed as part of Project Tungsten and is designed to improve the performance and scalability of Spark. It was first introduced with Spark 1.3.

Spark DataFrames are much easier to use and manipulate than their predecessor RDDs (Resilient Distributed Dataset). They are immutable, like RDDs, and support lazy loading, which means no transformation is performed on the DataFrames unless an action is called. The execution plan for the DataFrames is prepared by Spark itself and hence is more optimized, making operations on DataFrames faster than those on RDDs.

Getting Started with Spark DataFrames

To get started with Spark DataFrames, we will have to create something called a SparkContext first. SparkContext configures the internal services under the hood and facilitates command execution from the Spark execution environment. The following code snippet is used to create SparkContext:

```
sc = SparkContext()
```

We also need to create an SQLContext before we can start working with DataFrames. SQLContext in Spark is a class that provides SQL-like functionality within Spark. We can create SQLContext using SparkContext:

```
from pyspark.sql import SQLContext

sqlc = SQLContext(sc)
```

There are three different ways of creating a DataFrame in Spark:

1. We can programmatically specify the schema of the DataFrame and manually enter the data in it. However, since Spark is generally used to handle big data, this method is of little use, apart from creating data for small test/sample cases.
2. Another method to create a DataFrame is from an existing RDD object in Spark. This is useful, because working on a DataFrame is way easier than working directly with RDDs.
3. We can also read the data directly from a data source to create a Spark DataFrame. Spark supports a variety of external data sources, including CSV, JSON, parquet, RDBMS tables, and Hive tables

Exercise 24: Specifying the Schema of a DataFrame

In this exercise, we will create a small sample DataFrame by manually specifying the schema and entering data in Spark. Even though this method has little application in a practical scenario, it will be a good starting point in getting started with **Spark DataFrames:**

1. Import the necessary libraries and create **SQLContext**

```
from pyspark import SparkContext
sc= SparkContext()
from pyspark.sql import SQLContext
sqlc = SQLContext(sc)
```

2. Import SQL utilities from the PySpark module and specify the schema of the sample DataFrame:

```
from pyspark.sql import *
na_schema = Row("Name","Age")
```

3. Create rows for the DataFrame as per the specified schema:

```
row1 = na_schema("Ankit", 23)
row2 = na_schema("Tyler", 26)
row3 = na_schema("Preity", 36)
```

4. Combine the rows together to create the DataFrame:

```
na_list = [row1, row2, row3]
df_na = sqlc.createDataFrame(na_list)
type(df_na)
```

```
pyspark.sql.dataframe.DataFrame
```

5. Now, show the DataFrame using the following command:

```
df_na.show()
```

```
+------+---+
|  Name|Age|
+------+---+
| Ankit| 23|
| Tyler| 26|
|Preity| 36|
+------+---+
```

Exercise 25: Creating a DataFrame from an Existing RDD (Resilient Distributed Dataset)

In this exercise, we will create a small sample DataFrame from an existing RDD object in Spark:

1. Create an RDD object that we will convert into DataFrame.
When Spark **parallelize** method is applied on a Collection (with elements), a new distributed data set is created with specified number of partitions and the elements of the collection are copied to the distributed dataset (RDD)

```
data = [("Ankit",23),("Tyler",26),("Preity",36)]
data_rdd = sc.parallelize(data)
type(data_rdd)
```

```
pyspark.rdd.RDD
```

2. Convert the RDD object into a DataFrame:

```
data_sd = sqlc.createDataFrame(data_rdd)
```

3. Now, show the DataFrame using the following command:

```
data_sd.show()
```

```
+------+---+
|    _1| _2|
+------+---+
| Ankit| 23|
| Tyler| 26|
|Preity| 36|
+------+---+
```

Exercise 26: Creating a DataFrame Using a CSV File

1. Read the data from the CSV file into the Spark DataFrame:

```
df = sqlc.read.format('com.databricks.spark.csv').
options(header='true',inferschema='true').load('cars_headers.csv')
type(df)
```

```
pyspark.sql.dataframe.DataFrame
```

2. Now, show the DataFrame using the following command:

```
df.show(5)
```

Exercise 27: Converting a Spark DataFrame to a Pandas DataFrame

Spark gives us the ability to write the data stored in Spark DataFrames into a local pandas DataFrame, or write them into external structured file formats such as CSV. However, before converting a Spark DataFrame into a local pandas DataFrame, make sure that the data would fit in the local driver memory. In the following exercise, we will explore how to convert the Spark DataFrame to a pandas DataFrame.

In this exercise, we will use the pre-created Spark DataFrame of the dataset from the previous exercise, and convert it into a local pandas DataFrame. We will then store this DataFrame into a CSV file. Perform the following steps:

1. Convert the Spark DataFrame into a pandas DataFrame using the following command:

```
import pandas as pd
df.toPandas()
```

	_c0	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	name
0	0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite
3	3	16.0	8	304.0	150.0	3433.0	12.0	70	1	amc rebel sst
4	4	17.0	8	302.0	140.0	3449.0	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...	...
393	393	27.0	4	140.0	86.00	2790.0	15.6	82	1	ford mustang gl
394	394	44.0	4	97.0	52.00	2130.0	24.6	82	2	vw pickup
395	395	32.0	4	135.0	84.00	2295.0	11.6	82	1	dodge rampage
396	396	28.0	4	120.0	79.00	2625.0	18.6	82	1	ford ranger
397	397	31.0	4	119.0	82.00	2720.0	19.4	82	1	chevy s-10

398 rows × 10 columns

2. Now use the following command to write the pandas DataFrame to a CSV file:

```
df.toPandas().to_csv('carsMyPandasCopy.csv')
```

3. You can also write the contents of a Spark DataFrame to CSV file, using spark-csv package:

```
df.write.csv('carsMySparkCopy.csv')
```

Exploring Spark DataFrames

One of the major advantages that the Spark DataFrames offer over the traditional RDDs is the ease of data use and exploration. The data is stored in a more structured tabular format in the DataFrames and hence is easier to make sense of. We can compute basic statistics such as the number of rows and columns, look at the schema, and compute summary statistics such as mean and standard deviation.

Exercise 28: Displaying Basic DataFrame Statistics

In this exercise, we will show basic DataFrame statistics of the first few rows of the data, and summary statistics for all the numerical DataFrame columns and an individual DataFrame column.

1. Look at the DataFrame schema. The schema is displayed in a tree format on the console:

```
df.printSchema()
```

2. Now, use the following command to print the column names of the Spark DataFrame:

```
df.schema.names
```

```
['_c0',
 'mpg',
 'cylinders',
 'displacement',
 'horsepower',
 'weight',
 'acceleration',
 'year',
 'origin',
 'name']
```

2. To retrieve the number of rows and columns present in the Spark DataFrame, use the following command:

```
df.count()
```

```
398
```

```
len(df.columns)
```

```
10
```

4. Let's fetch the first n rows of the data. We can do this by using the **head()** method. However, we use the **show()** method as it displays the data in a better format:

```
df.show(5)
```

5. Now, compute the summary statistics, such as mean and standard deviation, for all the numerical columns in the DataFrame:

```
df.describe().show()
```

6. To compute the summary statistics for an individual numerical column of a Spark DataFrame, use the following command:

```
df.describe('acceleration').show()
```

```
+-------+-----------------+
|summary|     acceleration|
+-------+-----------------+
|  count|              398|
|   mean|15.568090452261291|
| stddev| 2.757688929812676|
|    min|              8.0|
|    max|             24.8|
+-------+-----------------+
```

Activity 9: Getting Started with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections and create a Spark DataFrame using all three methods. We will also compute DataFrame statistics, and finally, write the same data into a CSV file. Feel free to use any open source dataset for this activity.

1. Create a sample DataFrame by manually specifying the schema.
2. Create a sample DataFrame from an existing RDD.

3. Create a sample DataFrame by reading the data from a CSV file.
4. Print the first seven rows of the sample DataFrame read in step 3.
5. Print the schema of the sample DataFrame read in step 3.
6. Print the number of rows and columns in the sample DataFrame.
7. Print the summary statistics of the DataFrame and any 2 individual numerical columns.
8. Write the first 7 rows of the sample DataFrame to a CSV file using both methods mentioned in the exercises.

Data Manipulation with Spark DataFrames

Data manipulation is a prerequisite for any data analysis. To draw meaningful insights from the data, we first need to understand, process, and massage the data. But this step becomes particularly hard with the increase in the size of data. Due to the scale of data, even simple operations such as filtering and sorting become complex coding problems. Spark DataFrames make data manipulation on big data a piece of cake. Manipulating the data in Spark DataFrames is quite like working on regular pandas DataFrames. Most of the data manipulation operations on Spark DataFrames can be done using simple and intuitive one-liners. We will use the Spark DataFrame containing the dataset that we created in previous exercises for these data manipulation exercises.

Exercise 29: Selecting and Renaming Columns from the DataFrame

In this exercise, we will first rename the column using the withColumnRenamed method and then select and print the schema using the select method. Perform the following steps.

1. Rename the columns of a Spark DataFrame using the withColumnRenamed() method:

```
df=df.withColumnRenamed('cylinders','cyl')
```

2. Select a single column or multiple columns from a Spark DataFrame using the select method:

```
df.select('cyl','mpg').show(5)
```

```
+---+----+
|cyl| mpg|
+---+----+
|  8|18.0|
|  8|15.0|
|  8|18.0|
|  8|16.0|
|  8|17.0|
+---+----+
only showing top 5 rows
```

Exercise 30: Adding and Removing a Column from the DataFrame

In this exercise, we will add a new column in the dataset using the withColumn method, and later, using the drop function, will remove it. Now, let's perform the following steps:

1. Add a new column in a Spark DataFrame using the withColumn method:

```
df = df.withColumn('weight_tons', df['weight']/1000.0)
```

```
df.show(5)
```

2. Now, to remove a column in a Spark DataFrame, use the drop method illustrated here:

```
df = df.drop('weight_tons')
```

```
df.show(5)
```

```
+---+----+---+------------+----------+------+------------+----+------+--------------------+
|_c0| mpg|cyl|displacement|horsepower|weight|acceleration|year|origin|                name|
+---+----+---+------------+----------+------+------------+----+------+--------------------+
|  0|18.0|  8|       307.0|     130.0|3504.0|        12.0|  70|     1|chevrolet chevell...|
|  1|15.0|  8|       350.0|     165.0|3693.0|        11.5|  70|     1|   buick skylark 320|
|  2|18.0|  8|       318.0|     150.0|3436.0|        11.0|  70|     1|  plymouth satellite|
|  3|16.0|  8|       304.0|     150.0|3433.0|        12.0|  70|     1|       amc rebel sst|
|  4|17.0|  8|       302.0|     140.0|3449.0|        10.5|  70|     1|         ford torino|
+---+----+---+------------+----------+------+------------+----+------+--------------------+
only showing top 5 rows
```

Exercise 31: Displaying and Counting Distinct Values in a DataFrame

To display the distinct values in a DataFrame, we use the distinct().show() method. Similarly, to count the distinct values, we will be using the distinct().count() method. Perform the following procedures to print the distinct values with the total count:

1. Select the distinct values in any column of a Spark DataFrame using the distinct method, in conjunction with the select method:

```
df.select('cyl').distinct().show()
```

```
+---+
|cyl|
+---+
|  6|
|  3|
|  5|
|  4|
|  8|
+---+
```

2. To count the distinct values in any column of a Spark DataFrame, use the count method, in conjunction with the distinct method:

```
df.select('cyl').distinct().count()
```

```
5
```

Exercise 32: Removing Duplicate Rows and Filtering Rows of a DataFrame

In this exercise, we will learn how to remove the duplicate rows from the dataset, and later, perform filtering operations on the same column.

1. Remove the duplicate values from a DataFrame using the dropDuplicates() method:

```
df.select('cyl').dropDuplicates().show()
```

```
+---+
|cyl|
+---+
|  6|
|  3|
|  5|
|  4|
|  8|
+---+
```

Note, that while distinct() works with whole records, dropDuplicates() can be applied to chosen columns:

```
#df.dropDuplicates(["cyl"]).show()
df.dropDuplicates(["cyl","displacement"]).show()
```

2. Filter the rows from a DataFrame using one or multiple conditions. These multiple conditions can be passed together to the DataFrame using Boolean operators such as and (&), or |, similar to how we do it for pandas DataFrames:

```
# Filtering using a single condition
df.filter(df.cyl == 3).show()
```

```
+---+----+---+------------+----------+------+------------+----+------+---------------+
|_c0| mpg|cyl|displacement|horsepower|weight|acceleration|year|origin|           name|
+---+----+---+------------+----------+------+------------+----+------+---------------+
| 71|19.0|  3|        70.0|     97.00|2330.0|        13.5|  72|     3|mazda rx2 coupe|
|111|18.0|  3|        70.0|     90.00|2124.0|        13.5|  73|     3|      maxda rx3|
|243|21.5|  3|        80.0|     110.0|2720.0|        13.5|  77|     3|     mazda rx-4|
|334|23.7|  3|        70.0|     100.0|2420.0|        12.5|  80|     3|  mazda rx-7 gs|
+---+----+---+------------+----------+------+------------+----+------+---------------+
```

3. Now, to filter the column using multiple conditions, use the following command:

```
df.filter((df.cyl==8) & (df.horsepower>200)).show()
```

```
+---+----+---+------------+----------+------+------------+----+------+--------------------+
|_c0| mpg|cyl|displacement|horsepower|weight|acceleration|year|origin|                name|
+---+----+---+------------+----------+------+------------+----+------+--------------------+
|  6|14.0|  8|       454.0|     220.0|4354.0|         9.0|  70|     1|    chevrolet impala|
|  7|14.0|  8|       440.0|     215.0|4312.0|         8.5|  70|     1|   plymouth fury iii|
|  8|14.0|  8|       455.0|     225.0|4425.0|        10.0|  70|     1|    pontiac catalina|
| 13|14.0|  8|       455.0|     225.0|3086.0|        10.0|  70|     1|buick estate wago...|
| 25|10.0|  8|       360.0|     215.0|4615.0|        14.0|  70|     1|           ford f250|
| 27|11.0|  8|       318.0|     210.0|4382.0|        13.5|  70|     1|          dodge d200|
| 67|11.0|  8|       429.0|     208.0|4633.0|        11.0|  72|     1|     mercury marquis|
| 94|13.0|  8|       440.0|     215.0|4735.0|        11.0|  73|     1|chrysler new york...|
| 95|12.0|  8|       455.0|     225.0|4951.0|        11.0|  73|     1|buick electra 225...|
|116|16.0|  8|       400.0|     230.0|4278.0|         9.5|  73|     1|  pontiac grand prix|
+---+----+---+------------+----------+------+------------+----+------+--------------------+
```

Exercise 33: Ordering Rows in a DataFrame

In this exercise, we will explore how to sort the rows in a DataFrame in ascending and descending order. Let's perform these steps:

1. Sort the rows in a DataFrame, using one or multiple conditions, in ascending or descending order:

```
df.orderBy(df.cyl).show()
```

2. To sort the rows in descending order, use the following command:

```
df.orderBy(df.cyl.desc()).show()
```

Exercise 34: Aggregating Values in a DataFrame

1. To calculate the mean horsepower for each cylinders count value, use the following command:

```
df.groupby('cyl').agg({'horsepower':'mean'}).show()
#df.groupby('cyl').agg({'horsepower':'mean'}).orderBy(df.cyl).show()
```

```
+---+-----------------+
|cyl|  avg(horsepower)|
+---+-----------------+
|  6|101.50602409638554|
|  3|            99.25|
|  5| 82.33333333333333|
|  4| 78.28140703517587|
|  8| 158.3009708737864|
+---+-----------------+
```

2. Now, let's calculate the number of rows for each lylinders value by using the following command:

```
df.groupby('cyl').count().show()
```

```
+---+-----+
|cyl|count|
+---+-----+
|  6|   84|
|  3|    4|
|  5|    3|
|  4|  204|
|  8|  103|
+---+-----+
```

Activity 10: Data Manipulation with Spark DataFrames

In this activity, we will use the concepts learned in the previous sections to manipulate the data in the Spark DataFrame created using the Iris dataset. We will perform basic data manipulation steps to test our ability to work with data in a Spark DataFrame. Feel free to use any open source dataset for this activity. Make sure the dataset you use has both numerical and categorical variables:

1. Rename any five columns of the DataFrame. If the DataFrame has more than columns, rename all the columns.

2. Select two numeric and one categorical column from the DataFrame.
3. Count the number of distinct categories in the categorical variable.
4. Create two new columns in the DataFrame by summing up and multiplying together the two numerical columns.
5. Drop both the original numerical columns.
6. Sort the data by the categorical column.
7. Calculate the mean of the summation column for each distinct category in the categorical variable.
8. Filter the rows with values greater than the mean of all the mean values calculated in step 7.
9. De-duplicate the resultant DataFrame to make sure it has only unique records.

Graphs in Spark

The ability to effectively visualize data is of paramount importance. Visual representations of data help the user develop a better understanding of data and uncover trends that might go unnoticed in text form. There are numerous types of plots available in Python, each with its own context.

Exercise 35: Creating a Bar Chart

In this exercise, we will try to plot the number of records available for each cylinders using a bar chart. We will have to first aggregate the data and count the number of records for each cylinders. We can then convert this aggregated data into a regular pandas DataFrame and use Matplotlib and Seaborn packages to create any kind of plots of it that we wish.

1. First, calculate the number of rows for each flower species and convert the result to a pandas DataFrame:

```
data = df.groupby('cyl').count().toPandas()
data.head()
```

	cyl	count
0	6	84
1	3	4
2	5	3
3	4	204
4	8	103

2. Now, create a bar plot from the resulting pandas DataFrame:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.barplot( x = data['cyl'], y = data['count'])
plt.xlabel('Cylinders')
plt.ylabel('count')
plt.title('Number of rows per cylinders')
```

Exercise 36: Creating a Linear Model Plot

In this exercise, we will plot the data points of two different variables and fit a straight line on them. This is similar to fitting a linear model on two variables and can help identify correlations between the two variables:

```
data = df.toPandas()
sns.lmplot(x = "displacement", y = "weight", data = data)
```

Exercise 37: Creating a KDE Plot and a Boxplot

In this exercise, we will create a kernel density estimation (KDE) plot, followed by a boxplot. Follow these instructions:

1. First, plot a KDE plot that shows us the distribution of a variable.

```
import seaborn as sns
data = df.toPandas()
sns.kdeplot(data.weight, shade = True)
#plt.show()
```

2. Now, plot the boxplots for the Iris dataset using the following command:

```
sns.boxplot(x = "cyl", y = "displacement", data = data)
```

Boxplots are a good way to look at the data distribution and locate outliers. They represent the distribution using the 1st quartile, the median, the 3rd quartile, and the interquartile range (25th to 75th percentile).