databricks Databricks-Spark

(https://databricks.com)

Explore data in a dataframe

In this notebook, you'll use Spark in Azure Databricks to explore data in files. One of the core ways in which you work with data in Spark is to load data into a **Dataframe** object, and then query, filter, and manipulate the dataframe to explore the data it contains.

Ingest data

Use the ▸ **Run Cell** menu option at the top-right of the following cell to run it and download data files into the Databricks file system (DBFS).

```
%sh
rm -r /dbfs/data
mkdir /dbfs/data
wget -O /dbfs/data/2019.csv https://raw.githubusercontent.com/MicrosoftLearning/dp-203-azure-data-
engineer/master/Allfiles/labs/24/data/2019.csv
wget -O /dbfs/data/2020.csv https://raw.githubusercontent.com/MicrosoftLearning/dp-203-azure-data-
engineer/master/Allfiles/labs/24/data/2020.csv
wget -O /dbfs/data/2021.csv https://raw.githubusercontent.com/MicrosoftLearning/dp-203-azure-data-
engineer/master/Allfiles/labs/24/data/2021.csv
```

```
rm: cannot remove '/dbfs/data': No such file or directory
--2023-04-02 23:37:31--  https://raw.githubusercontent.com/MicrosoftLearning/dp-203-azure-data-engineer/master/Allfiles/l
abs/24/data/2019.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 122681 (120K) [text/plain]
Saving to: ‘/dbfs/data/2019.csv’

     0K .......... .......... .......... .......... .......... 41% 6.53M 0s
    50K .......... .......... .......... .......... .......... 83% 8.56M 0s
   100K .......... .........                                  100% 15.0M=0.01s

2023-04-02 23:37:31 (8.09 MB/s) - ‘/dbfs/data/2019.csv’ saved [122681/122681]

--2023-04-02 23:37:31--  https://raw.githubusercontent.com/MicrosoftLearning/dp-203-azure-data-engineer/master/Allfiles/l
abs/24/data/2020.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.111.133, 185.199.108.133, 185.199.109.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 283396 (277K) [text/plain]
```

Query data in files

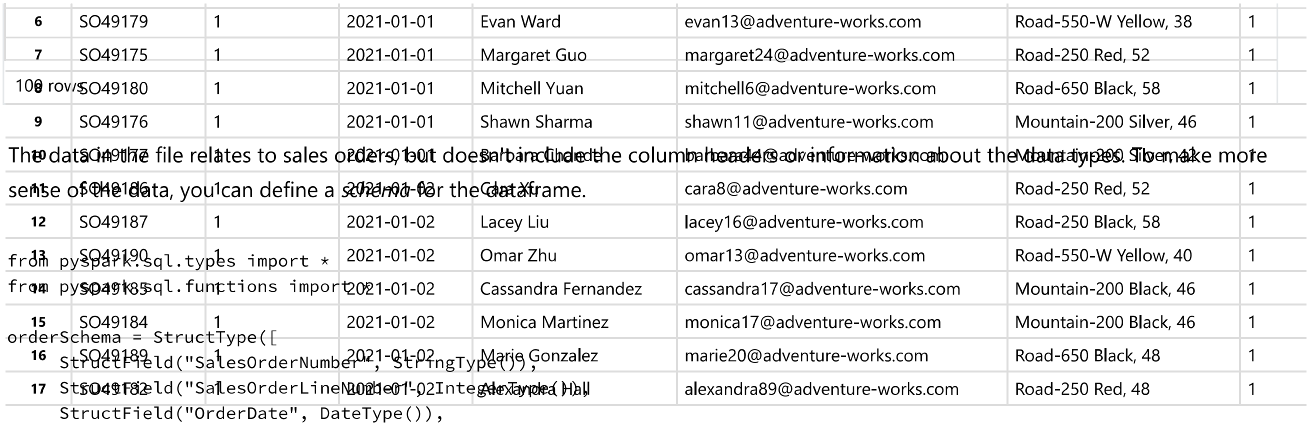
The previous cell downloaded three comma-separated values (CSV) files to the **data** folder in the DBFS storage for your workspace.

Run the following cell to load the data from the file and view the first 100 rows.

```
df = spark.read.load('data/*.csv', format='csv')
display(df.limit(100))
```

Table

	_c0	_c1	_c2	_c3	_c4	_c5	_c6
1	SO49171	1	2021-01-01	Mariah Foster	mariah21@adventure-works.com	Road-250 Black, 48	1
2	SO49172	1	2021-01-01	Brian Howard	brian23@adventure-works.com	Road-250 Red, 44	1
3	SO49173	1	2021-01-01	Linda Alvarez	linda19@adventure-works.com	Mountain-200 Silver, 38	1
4	SO49174	1	2021-01-01	Gina Hernandez	gina4@adventure-works.com	Mountain-200 Silver, 42	1
5	SO49178	1	2021-01-01	Beth Ruiz	beth4@adventure-works.com	Road-550-W Yellow, 44	1

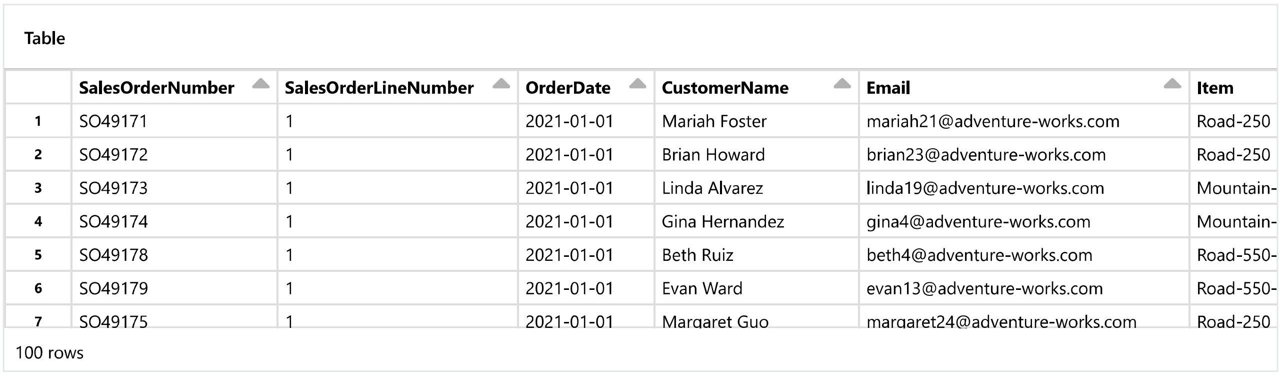


6	SO49179	1	2021-01-01	Evan Ward	evan13@adventure-works.com	Road-550-W Yellow, 38	1
7	SO49175	1	2021-01-01	Margaret Guo	margaret24@adventure-works.com	Road-250 Red, 52	1
8	SO49180	1	2021-01-01	Mitchell Yuan	mitchell6@adventure-works.com	Road-650 Black, 58	1
9	SO49176	1	2021-01-01	Shawn Sharma	shawn11@adventure-works.com	Mountain-200 Silver, 46	1
10	SO49177	1	2021-01-01	Barbara Chande	barbara4@adventure-works.com	Mountain-200 Silver, 42	1
11	SO49186	1	2021-01-02	Cara Xu	cara8@adventure-works.com	Road-250 Red, 52	1
12	SO49187	1	2021-01-02	Lacey Liu	lacey16@adventure-works.com	Road-250 Black, 58	1
13	SO49190	1	2021-01-02	Omar Zhu	omar13@adventure-works.com	Road-550-W Yellow, 40	1
14	SO49185	1	2021-01-02	Cassandra Fernandez	cassandra17@adventure-works.com	Mountain-200 Black, 46	1
15	SO49184	1	2021-01-02	Monica Martinez	monica17@adventure-works.com	Mountain-200 Black, 46	1
16	SO49189	1	2021-01-02	Marie Gonzalez	marie20@adventure-works.com	Road-650 Black, 48	1
17	SO49182	1	2021-01-02	Alexandra Hall	alexandra89@adventure-works.com	Road-250 Red, 48	1

100 rows

The data in the file relates to sales orders, but doesn't include the column headers or information about the data types. To make more sense of the data, you can define a *schema* for the dataframe.

```
from pyspark.sql.types import *
from pyspark.sql.functions import *

orderSchema = StructType([
     StructField("SalesOrderNumber", StringType()),
     StructField("SalesOrderLineNumber", IntegerType()),
     StructField("OrderDate", DateType()),
     StructField("CustomerName", StringType()),
     StructField("Email", StringType()),
     StructField("Item", StringType()),
     StructField("Quantity", IntegerType()),
     StructField("UnitPrice", FloatType()),
     StructField("Tax", FloatType())
 ])

df = spark.read.load('/data/*.csv', format='csv', schema=orderSchema)
display(df.limit(100))
```



Table

	SalesOrderNumber	SalesOrderLineNumber	OrderDate	CustomerName	Email	Item
1	SO49171	1	2021-01-01	Mariah Foster	mariah21@adventure-works.com	Road-250
2	SO49172	1	2021-01-01	Brian Howard	brian23@adventure-works.com	Road-250
3	SO49173	1	2021-01-01	Linda Alvarez	linda19@adventure-works.com	Mountain-
4	SO49174	1	2021-01-01	Gina Hernandez	gina4@adventure-works.com	Mountain-
5	SO49178	1	2021-01-01	Beth Ruiz	beth4@adventure-works.com	Road-550-
6	SO49179	1	2021-01-01	Evan Ward	evan13@adventure-works.com	Road-550-
7	SO49175	1	2021-01-01	Margaret Guo	margaret24@adventure-works.com	Road-250

100 rows

This time the data includes the column headers.

To verify that the appropriate data types have been defined, you van view the schema of the dataframe.

```
 df.printSchema()
```

```
root
 |-- SalesOrderNumber: string (nullable = true)
 |-- SalesOrderLineNumber: integer (nullable = true)
 |-- OrderDate: date (nullable = true)
 |-- CustomerName: string (nullable = true)
 |-- Email: string (nullable = true)
 |-- Item: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- UnitPrice: float (nullable = true)
 |-- Tax: float (nullable = true)
```

Analyze data in a dataframe

The dataframe object in Spark is similar to a *Pandas* dataframe in Python, and includes a wide range of functions that you can use to manipulate, filter, group, and otherwise analyze the data it contains.

Filter a dataframe

Run the following cell to:

- Filter the columns of the sales orders dataframe to include only the customer name and email address.
- Count the total number of order records
- Count the number of distinct customers

```
customers = df['CustomerName', 'Email']
print(customers.count())
print(customers.distinct().count())
display(customers.distinct())
```

```
32718
12427
```

Table

	CustomerName	Email
1	Bridget Andersen	bridget15@adventure-works.com
2	Mya Butler	mya14@adventure-works.com
3	Deanna Hernandez	deanna29@adventure-works.com
4	Ricky Navarro	ricky10@adventure-works.com
5	Omar Ye	omar9@adventure-works.com
6	Kellie Gutierrez	kellie9@adventure-works.com
7	Raymond Rana	raymond13@adventure-works.com

10,000 rows | Truncated data

Observe the following details:

- When you perform an operation on a dataframe, the result is a new dataframe (in this case, a new customers dataframe is created by selecting a specific subset of columns from the df dataframe)
- Dataframes provide functions such as count and distinct that can be used to summarize and filter the data they contain.
- The `dataframe['Field1', 'Field2', ...]` syntax is a shorthand way of defining a subset of column. You can also use **select** method, so the first line of the code above could be written as `customers = df.select("CustomerName", "Email")`

Now let's apply a filter to include only the customers who have placed an order for a specific product:

```
customers = df.select("CustomerName", "Email").where(df['Item']=='Road-250 Red, 52')
print(customers.count())
print(customers.distinct().count())
display(customers.distinct())
```

```
133
133
```

Table

	CustomerName	Email
1	Bridget Andersen	bridget15@adventure-works.com
2	Ricky Navarro	ricky10@adventure-works.com
3	Micah Xu	micah23@adventure-works.com
4	Bryant Raman	bryant11@adventure-works.com
5	Roger Wang	roger6@adventure-works.com
6	Priscilla Yuan	priscilla6@adventure-works.com
7	Lolan Song	lolan1@adventure-works.com

133 rows

Note that you can “chain” multiple functions together so that the output of one function becomes the input for the next - in this case, the dataframe created by the select method is the source dataframe for the where method that is used to apply filtering criteria.

Aggregate and group data in a dataframe

Run the following cell to aggregate and group the order data.

```
productSales = df.select("Item", "Quantity").groupBy("Item").sum()
display(productSales)
```

Table

	Item	sum(Quantity)
1	Mountain-200 Black, 42	388
2	Touring-1000 Yellow, 46	74
3	Touring-1000 Blue, 54	67
4	Short-Sleeve Classic Jersey, S	216
5	Women's Mountain Shorts, S	146
6	Long-Sleeve Logo Jersey, L	234
7	Mountain-400-W Silver, 42	59

130 rows

Note that the results show the sum of order quantities grouped by product. The **groupBy** method groups the rows by *Item*, and the subsequent **sum** aggregate function is applied to all of the remaining numeric columns (in this case, *Quantity*)

Let's try another aggregation.

```
yearlySales = df.select(year("OrderDate").alias("Year")).groupBy("Year").count().orderBy("Year")
display(yearlySales)
```

Table

	Year	count
1	2019	1201
2	2020	2733
3	2021	28784

3 rows

This time the results show the number of sales orders per year. Note that the select method includes a SQL **year** function to extract the year component of the *OrderDate* field, and then an **alias** method is used to assign a columm name to the extracted year value. The data is then grouped by the derived *Year* column and the **count** of rows in each group is calculated before finally the **orderBy** method is used to sort the resulting dataframe.

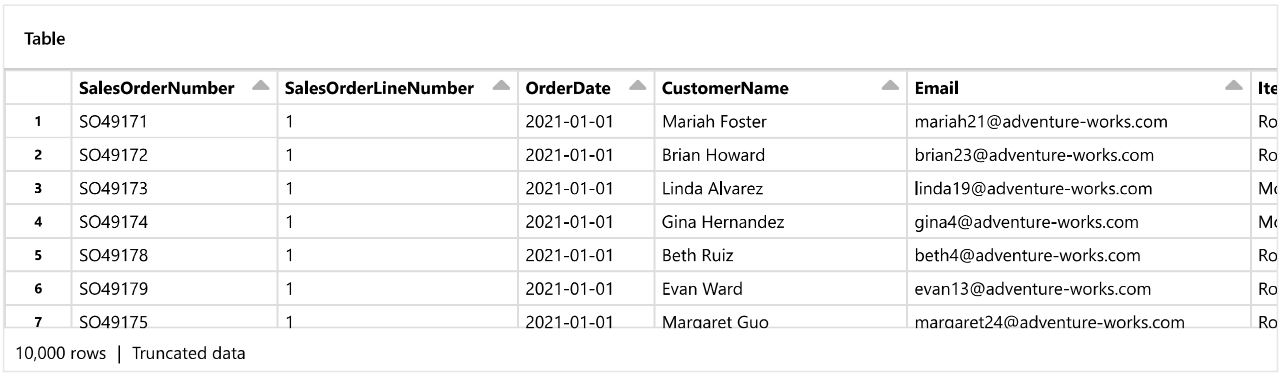
Query data using Spark SQL

As you've seen, the native methods of the dataframe object enable you to query and analyze data quite effectively. However, many data analysts are more comfortable working with SQL syntax. Spark SQL is a SQL language API in Spark that you can use to run SQL statements, or even persist data in relational tables.

Use Spark SQL in PySpark code

The default language in Azure Synapse Studio notebooks is *PySpark*, which is a Spark-based Python runtime. Within this runtime, you can use the **spark.sql** library to embed Spark SQL syntax within your Python code, and work with SQL constructs such as tables and views.

```
df.createOrReplaceTempView("salesorders")

spark_df = spark.sql("SELECT * FROM salesorders")
display(spark_df)
```



Table

	SalesOrderNumber	SalesOrderLineNumber	OrderDate	CustomerName	Email	Ite
1	SO49171	1	2021-01-01	Mariah Foster	mariah21@adventure-works.com	Ro
2	SO49172	1	2021-01-01	Brian Howard	brian23@adventure-works.com	Ro
3	SO49173	1	2021-01-01	Linda Alvarez	linda19@adventure-works.com	Mo
4	SO49174	1	2021-01-01	Gina Hernandez	gina4@adventure-works.com	Mo
5	SO49178	1	2021-01-01	Beth Ruiz	beth4@adventure-works.com	Ro
6	SO49179	1	2021-01-01	Evan Ward	evan13@adventure-works.com	Ro
7	SO49175	1	2021-01-01	Margaret Guo	margaret24@adventure-works.com	Ro

10,000 rows | Truncated data

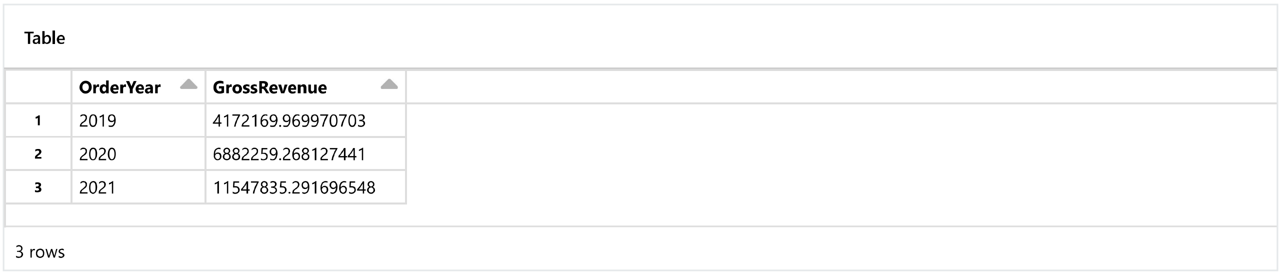
Observe that:

- The code persists the data in the **df** dataframe as a temporary view named **salesorders**. Spark SQL supports the use of temporary views or persisted tables as sources for SQL queries.
- The **spark.sql** method is then used to run a SQL query against the **salesorders** view.
- The results of the query are stored in a dataframe.

Run SQL code in a cell

While it's useful to be able to embed SQL statements into a cell containing PySpark code, data analysts often just want to work directly in SQL.

```
%sql

SELECT YEAR(OrderDate) AS OrderYear,
    SUM((UnitPrice * Quantity) + Tax) AS GrossRevenue
FROM salesorders
GROUP BY YEAR(OrderDate)
ORDER BY OrderYear;
```



Table

	OrderYear	GrossRevenue
1	2019	4172169.969970703
2	2020	6882259.268127441
3	2021	11547835.291696548

3 rows

Observe that:

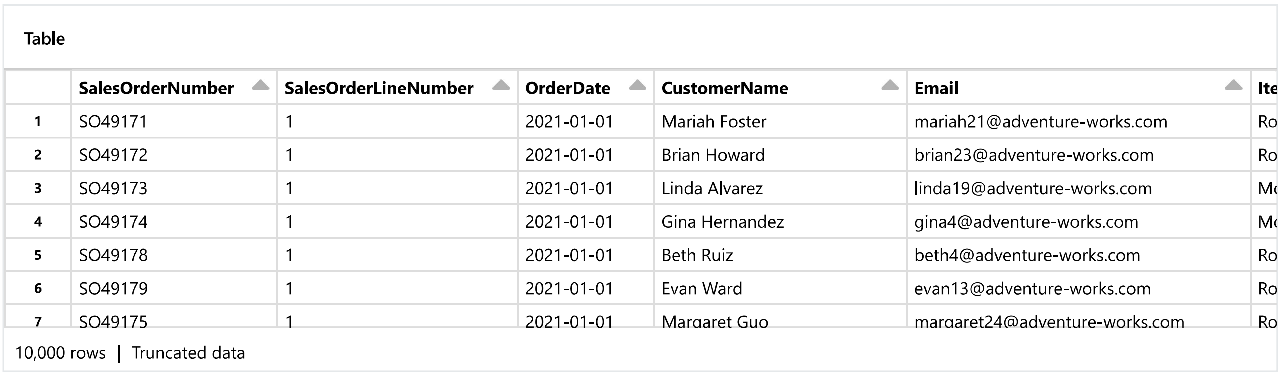
- The ``%sql` line at the beginning of the cell (called a magic) indicates that the Spark SQL language runtime should be used to run the code in this cell instead of PySpark.
- The SQL code references the **salesorder** view that you created previously using PySpark.
- The output from the SQL query is automatically displayed as the result under the cell.

> **Note**: For more information about Spark SQL and dataframes, see the Spark SQL documentation (https://spark.apache.org/docs/2.2.0/sql-programming-guide.html).

Visualize data with Spark

A picture is proverbially worth a thousand words, and a chart is often better than a thousand rows of data. While notebooks in Azure Databricks include support for visualizing data from a dataframe or Spark SQL query, it is not designed for comprehensive charting. However, you can use Python graphics libraries like matplotlib and seaborn to create charts from data in dataframes.

```
%sql

SELECT * FROM salesorders
```



Table

	SalesOrderNumber	SalesOrderLineNumber	OrderDate	CustomerName	Email	Ite
1	SO49171	1	2021-01-01	Mariah Foster	mariah21@adventure-works.com	Ro
2	SO49172	1	2021-01-01	Brian Howard	brian23@adventure-works.com	Ro
3	SO49173	1	2021-01-01	Linda Alvarez	linda19@adventure-works.com	Mo
4	SO49174	1	2021-01-01	Gina Hernandez	gina4@adventure-works.com	Mo
5	SO49178	1	2021-01-01	Beth Ruiz	beth4@adventure-works.com	Ro
6	SO49179	1	2021-01-01	Evan Ward	evan13@adventure-works.com	Ro
7	SO49175	1	2021-01-01	Margaret Guo	margaret24@adventure-works.com	Ro

10,000 rows | Truncated data

Above the table of results, select **+** and then select **Visualization** to view the visualization editor, and then apply the following options: - **Visualization type**: Bar - **X Column**: Item - **Y Column**: *Add a new column and select* **Quantity**. *Apply the* **Sum** *aggregation*.

Save the visualization and then re-run the code cell to view the resulting chart in the notebook.

Get started with matplotlib

You can get mroe control over data visualizations by using graphics libraries.

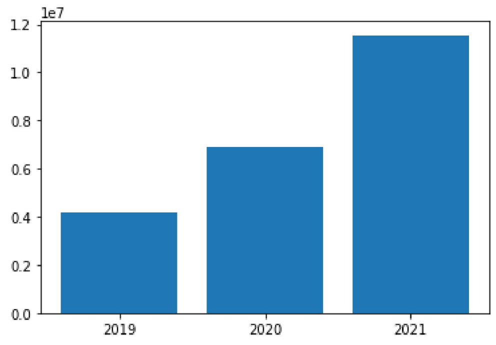
Run the following cell to retrieve some sales order data into a dataframe.

```
sqlQuery = "SELECT CAST(YEAR(OrderDate) AS CHAR(4)) AS OrderYear, \
             SUM((UnitPrice * Quantity) + Tax) AS GrossRevenue \
         FROM salesorders \
         GROUP BY CAST(YEAR(OrderDate) AS CHAR(4)) \
         ORDER BY OrderYear"
df_spark = spark.sql(sqlQuery)
df_spark.show()
```

```
+---------+--------------------+
|OrderYear|        GrossRevenue|
+---------+--------------------+
|     2019|   4172169.969970703|
|     2020|   6882259.268127441|
|     2021|1.1547835291696548E7|
+---------+--------------------+
```

To visualize the data as a chart, we'll start by using the matplotlib Python library. This library is the core plotting library on which many others are based, and provides a great deal of flexibility in creating charts.

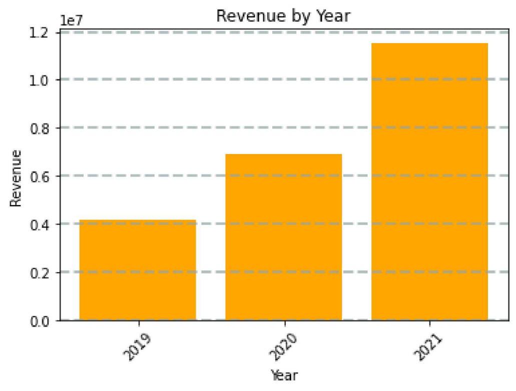
```
from matplotlib import pyplot as plt

# matplotlib requires a Pandas dataframe, not a Spark one
df_sales = df_spark.toPandas()

# Create a bar plot of revenue by year
plt.bar(x=df_sales['OrderYear'], height=df_sales['GrossRevenue'])

# Display the plot
plt.show()
```



Review the results, which consist of a column chart with the total gross revenue for each year. Note the following features of the code used to produce this chart:

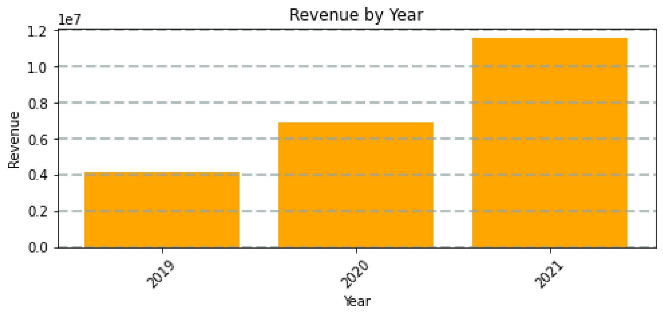
- The **matplotlib** library requires a Pandas dataframe, so you need to convert the Spark dataframe returned by the Spark SQL query to this format.
- At the core of the **matplotlib** library is the **pyplot** object. This is the foundation for most plotting functionality.
- The default settings result in a usable chart, but there's considerable scope to customize it, as you'll see by running the following cell.

```
# Clear the plot area
plt.clf()

# Create a bar plot of revenue by year
plt.bar(x=df_sales['OrderYear'], height=df_sales['GrossRevenue'], color='orange')

# Customize the chart
plt.title('Revenue by Year')
plt.xlabel('Year')
plt.ylabel('Revenue')
plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)
plt.xticks(rotation=45)

# Show the figure
plt.show()
```



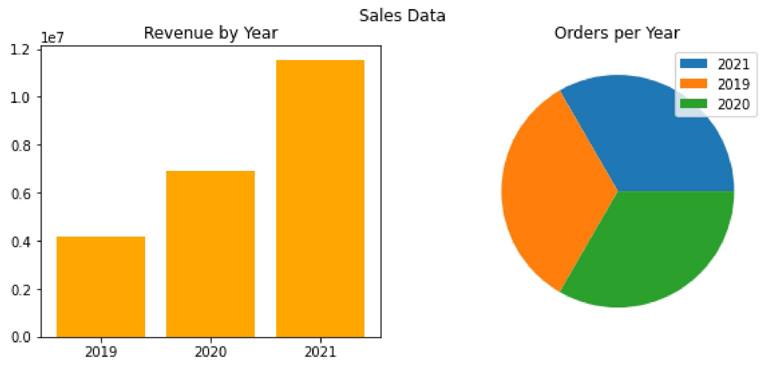
A plot is technically contained with a **Figure**. In the previous examples, the figure was created implicitly for you; but you can create it explicitly.

```
# Clear the plot area
plt.clf()

# Create a Figure
fig = plt.figure(figsize=(8,3))

# Create a bar plot of revenue by year
plt.bar(x=df_sales['OrderYear'], height=df_sales['GrossRevenue'], color='orange')

# Customize the chart
plt.title('Revenue by Year')
plt.xlabel('Year')
plt.ylabel('Revenue')
plt.grid(color='#95a5a6', linestyle='--', linewidth=2, axis='y', alpha=0.7)
plt.xticks(rotation=45)

# Show the figure
plt.show()
```



A figure can contain multiple subplots, each on its own axis.

```
# Clear the plot area
plt.clf()

# Create a figure for 2 subplots (1 row, 2 columns)
fig, ax = plt.subplots(1, 2, figsize = (10,4))

# Create a bar plot of revenue by year on the first axis
ax[0].bar(x=df_sales['OrderYear'], height=df_sales['GrossRevenue'], color='orange')
ax[0].set_title('Revenue by Year')

# Create a pie chart of yearly order counts on the second axis
yearly_counts = df_sales['OrderYear'].value_counts()
ax[1].pie(yearly_counts)
ax[1].set_title('Orders per Year')
ax[1].legend(yearly_counts.keys().tolist())

# Add a title to the Figure
fig.suptitle('Sales Data')

# Show the figure
plt.show()
```

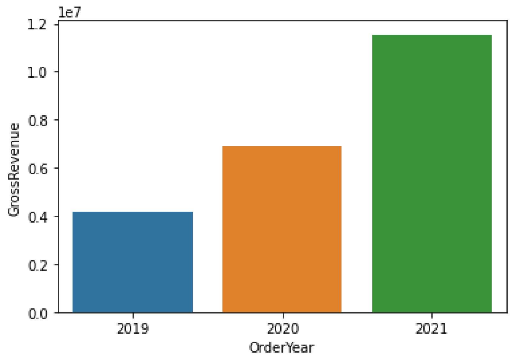


Note: To learn more about plotting with matplotlib, see the matplotlib documentation (https://matplotlib.org/).

Use the seaborn library

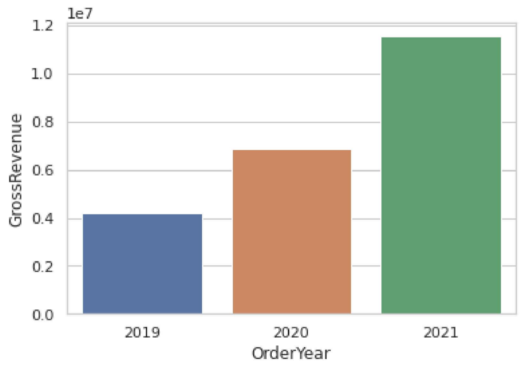
While **matplotlib** enables you to create complex charts of multiple types, it can require some complex code to achieve the best results. For this reason, over the years, many new libraries have been built on the base of matplotlib to abstract its complexity and enhance its capabilities. One such library is **seaborn**.

```
import seaborn as sns

# Clear the plot area
plt.clf()

# Create a bar chart
ax = sns.barplot(x="OrderYear", y="GrossRevenue", data=df_sales)
plt.show()
```



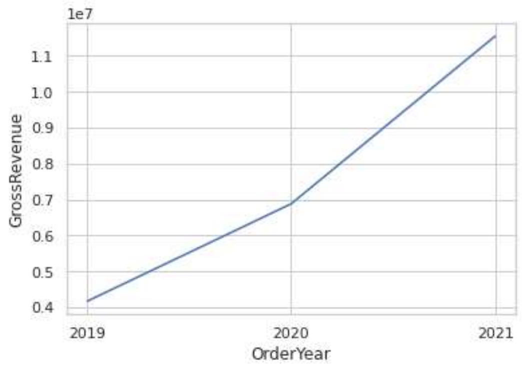
The **seaborn** library makes it simpler to create complex plots of statistical data, and enables you to control the visual theme for consistent data visualizations.

```
# Clear the plot area
plt.clf()

# Set the visual theme for seaborn
sns.set_theme(style="whitegrid")

# Create a bar chart
ax = sns.barplot(x="OrderYear", y="GrossRevenue", data=df_sales)
plt.show()
```



Both **matplotlib** and **seaborn** support multiple charts types. For example, run the following cell to view the yearly sales totals as a line chart.

```
# Clear the plot area
plt.clf()

# Create a bar chart
ax = sns.lineplot(x="OrderYear", y="GrossRevenue", data=df_sales)
plt.show()
```



Note: To learn more about plotting with seaborn, see the seaborn documentation (https://seaborn.pydata.org/index.html).

In this notebook, you've explored some basic techniques for using Spark to explore data in files. To learn more about working with Dataframes in Azure Databricks using PySpark, see Introduction to DataFrames - Python (https://docs.microsoft.com/azure/databricks/spark/latest/dataframes-datasets/introduction-to-dataframes-python) in the Azure Databricks documentation.