Lab: Query and Transform the data using spark pool

Duration: 45 minutes

The main task for this exercise is as follows:

1. Use Spark to explore data
2. Analyze data in a dataframe
3. Query data using Spark SQL
4. Visualize data with Spark
5. Transform data by using Spark

Task 1: Use Spark to explore data

1. Select any of the files in the **sales/csv** folder, and then in the **New notebook** list on the toolbar, select **Load to DataFrame**. A dataframe is a structure in Spark that represents a tabular dataset.

2. In the new **Notebook 1** tab that opens, in the **Attach to** list, select your Spark pool (**spark*xxxxxxxx***). Then use the ▷ **Run all** button to run all of the cells in the notebook (there's currently only one!).

   Since this is the first time you've run any Spark code in this session, the Spark pool must be started. This means that the first run in the session can take a few minutes. Subsequent runs will be quicker.

3. While you are waiting for the Spark session to initialize, review the code that was generated; which looks similar to this:

```
 df = spark.read.load('abfss://files@datalakexxxxxxx.dfs.core.windows.net/sales/csv/2019.csv',
format='csv'
 ## If header exists uncomment line below
 ##, header=True
 )
 display(df.limit(10))
```

4. When the code has finished running, review the output beneath the cell in the notebook. It shows the first ten rows in the file you selected, with automatic column names in the form **_c0**, **_c1**, **_c2**, and so on.

5. Modify the code so that the **spark.read.load** function reads data from all of the CSV files in the folder, and the **display** function shows the first 100 rows. Your code should look like this (with *datalakexxxxxxx* matching the name of your data lake store):

```
df = spark.read.load('abfss://files@datalakexxxxxxx.dfs.core.windows.net/sales/*.csv',
format='csv'
)
display(df.limit(100))
```

6. Use the ▷ button to the left of the code cell to run just that cell, and review the results.

The dataframe now includes data from all of the files, but the column names are not useful. Spark uses a "schema-on-read" approach to try to determine appropriate data types for the columns based on the data they contain, and if a header row is present in a text file it can be used to identify the column names (by specifying a **header=True** parameter in the **load** function). Alternatively, you can define an explicit schema for the dataframe.

7. Modify the code as follows (replacing *datalakexxxxxxx*), to define an explicit schema for the dataframe that includes the column names and data types. Rerun the code in the cell.

```
%%pyspark
 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('abfss://files@datalakexxxxxxx.dfs.core.windows.net/sales/*.csv',
format='csv', schema=orderSchema, header=True)
 display(df.limit(100))
```

8. Under the results, use the **+ Code** button to add a new code cell to the notebook. Then in the new cell, add the following code to display the dataframe's schema:

```
 df.printSchema()
```

9. Run the new cell and verify that the dataframe schema matches the **orderSchema** you defined. The **printSchema** function can be useful when using a dataframe with an automatically inferred schema.

Task 2: 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

1. Add a new code cell to the notebook, and enter the following code in it:

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

2. Run the new code cell, and review the results. 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")`
3. Modify the code as follows:

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

4. Run the modified code to view the customers who have purchased the *Road-250 Red, 52* product. 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

1. Add a new code cell to the notebook, and enter the following code in it:

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

2. Run the code cell you added, and 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*)

3. Add another new code cell to the notebook, and enter the following code in it:

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

4. Run the code cell you added, and note that 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.

Task 3: 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.

1. Add a new code cell to the notebook, and enter the following code in it:

   CodeCopy

   ```
   df.createOrReplaceTempView("salesorders")

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

2. Run the cell and review the results. 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.

1. Add a new code cell to the notebook, and enter the following code in it:

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

2. Run the cell and review the results. 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.

Task 3: 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 Synapse Analytics include a built in chart view for data that is displayed 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.

View results as a chart

1. Add a new code cell to the notebook, and enter the following code in it:

```
%%sql
 SELECT * FROM salesorders
```

2. Run the code and observe that it returns the data from the **salesorders** view you created previously.
3. In the results section beneath the cell, change the **View** option from **Table** to **Chart**.
4. Use the **View options** button at the top right of the chart to display the options pane for the chart. Then set the options as follows and select **Apply**:
    - **Chart type**: Bar chart
    - **Key**: Item
    - **Values**: Quantity
    - **Series Group**: *leave blank*
    - **Aggregation**: Sum
    - **Stacked**: *Unselected*

Get started with matplotlib

1. Add a new code cell to the notebook, and enter the following code in it:

```
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()
```

2. Run the code and observe that it returns a Spark dataframe containing the yearly revenue.

   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.

3. Add a new code cell to the notebook, and add the following code to it:

```
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()
```

4. Run the cell and 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

5. Modify the code to plot the chart as follows:

```
# 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()
```

6. Re-run the code cell and view the results. The chart now includes a little more information.

Task 4: Transform data by using Spark

Apache Spark provides a distributed data processing platform that you can use to perform complex data transformations at scale.

Load source data

Let's start by loading some historical sales order data into a dataframe.

Review the code in the cell below, which loads the sales order from all of the csv files within the **data** directory. Then click the ▷ button to the left of the cell to run it.
Note: The first time you run a cell in a notebook, the Spark pool must be started; which can take several minutes.

```
order_details =
spark.read.csv(''abfss://files@asastoragepc.dfs.core.windows.net/sales/csv /*.csv',
header=True, inferSchema=True)

display(order_details.limit(5))
```

Transform the data structure

The source data includes a **CustomerName** field, that contains the customer's first and last name. Let's modify the dataframe to separate this field into separate **FirstName** and **LastName** fields.

```
from pyspark.sql.functions import split, col

# Create the new FirstName and LastName fields

transformed_df = order_details.withColumn("FirstName", split(col("CustomerName"), "
").getItem(0)).withColumn("LastName", split(col("CustomerName"), " ").getItem(1))

# Remove the CustomerName field

transformed_df = transformed_df.drop("CustomerName")

display(transformed_df.limit(5))
```

The code above creates a new dataframe with the **CustomerName** field removed and two new **FirstName** and **LastName** fields.

You can use the full power of the Spark SQL library to transform the data by filtering rows, deriving, removing, renaming columns, and applying any other required data modifications.

Save the transformed data

After making the required changes to the data, you can save the results in a supported file format.

Note: Commonly, *Parquet* format is preferred for data files that you will use for further analysis or ingestion into an analytical store. Parquet is a very efficient format that is supported by most large scale data analytics systems. In fact, sometimes your data transformation requirement may simply be to convert data from another format (such as CSV) to Parquet!

Use the following code to save the transformed dataframe in Parquet format (Overwriting the data if it already exists).

```
transformed_df.write.mode("overwrite").parquet('/transformed_data/orders.parquet')

print ("Transformed data saved!")
```

In the **files** tab (which should still be open above), navigate to the root **files** container and verify that a new folder named **transformed_data** has been created, containing a file named **orders.parquet**. Then return to this notebook.

Partition data

A common way to optimize performance when dealing with large volumes of data is to partition the data files based on one or more field values. This can significant improve performance and make it easier to filter data.

Use the following cell to derive new **Year** and **Month** fields and then save the resulting data in Parquet format, partitioned by year and month.

```
from pyspark.sql.functions import year, month, col

dated_df = transformed_df.withColumn("Year", year(col("OrderDate"))).withColumn("Month", month(col("OrderDate")))

display(dated_df.limit(5))

dated_df.write.partitionBy("Year","Month").mode("overwrite").parquet("/partitioned_data")

print ("Transformed data saved!")
```

In the **files** tab (which should still be open above), navigate to the root **files** container and verify that a new folder named **partitioned_data** has been created, containing a hierachy of folders in the format **Year=*NNNN*** / **Month=*N***, each containing a .parquet file for the orders placed in the corresponding year and month. Then return to this notebook.

You can read this data into a dataframe from any folder in the hierarchy, using explicit values or wildcards for partitioning fields. For example, use the following code to get the sales orders placed in 2020 for all months.

```
orders_2020 = spark.read.parquet('/partitioned_data/Year=2020/Month=*')

display(orders_2020.limit(5))
```

Note that the partitioning columns specified in the file path are omitted in the resulting dataframe.

Use SQL to transform data

Spark is a very flexible platform, and the **SQL** library that provides the dataframe also enables you to work with data using SQL semantics. You can query and transform data in dataframes by using SQL queries, and persist the results as tables - which are metadata abstractions over files.

First, use the following code to save the original sales orders data (loaded from CSV files) as a table. Technically, this is an *external* table because the **path** parameter is used

to specify where the data files for the table are stored (an *internal* table is stored in the system storage for the Spark metastore and managed automatically).

```
order_details.write.saveAsTable('sales_orders', format='parquet', mode='overwrite', path='/sales_orders_table')

print ("Source table saved.")
```

In the **files** tab (which should still be open above), navigate to the root **files** container and verify that a new folder named **sales_orders_table** has been created, containing parquet files for the table data. Then return to this notebook.

Now that the table has been created, you can use SQL to transform it. For example, the following code derives new Year and Month columns and then saves the results as a partitioned external table.

```
sql_transform = spark.sql("SELECT *, YEAR(OrderDate) AS Year, MONTH(OrderDate) AS Month FROM sales_orders")

display(sql_transform.limit(5))

sql_transform.write.partitionBy("Year","Month").saveAsTable('transformed_orders', format='parquet', mode='overwrite', path='/transformed_orders_table')

print ("Transformed table saved.")
```

In the **files** tab (which should still be open above), navigate to the root **files** container and verify that a new folder named **transformed_orders_table** has been created, containing a hierachy of folders in the format **Year=*NNNN*** / **Month=*N***, each containing a .parquet file for the orders placed in the corresponding year and month. Then return to this notebook.

Essentially you've performed the same data transformation into partitioned parquet files as before, but by using SQL instead of native dataframe methods.

You can read this data into a dataframe from any folder in the hierarchy as before; but because the data files are also abstracted by a table in the metastore, you can query the data directly using SQL.

```
%%sql

SELECT * FROM transformed_orders

WHERE Year = 2021

    AND Month = 1;
```

Because these are *external* tables, you can drop the tables from the metastore without deleting the files - so the transfomed data remains available for other downstream data analytics or ingestion processes.

```
%%sql

DROP TABLE transformed_orders;

DROP TABLE sales_orders;
```