Lab 4 - Explore, transform, and load data into the Data Warehouse using Apache Spark

This lab teaches how to explore data stored in a data lake, transform the data, and load data into a relational data store. You will explore Parquet and JSON files and use techniques to query and transform JSON files with hierarchical structures. Then you will use Apache Spark to load data into the data warehouse and join Parquet data in the data lake with data in the dedicated SQL pool.

After completing this lab, you will be able to:

- Perform Data Exploration in Synapse Studio
- Ingest data with Spark notebooks in Azure Synapse Analytics
- Transform data with DataFrames in Spark pools in Azure Synapse Analytics
- Integrate SQL and Spark pools in Azure Synapse Analytics

Lab setup and pre-requisites

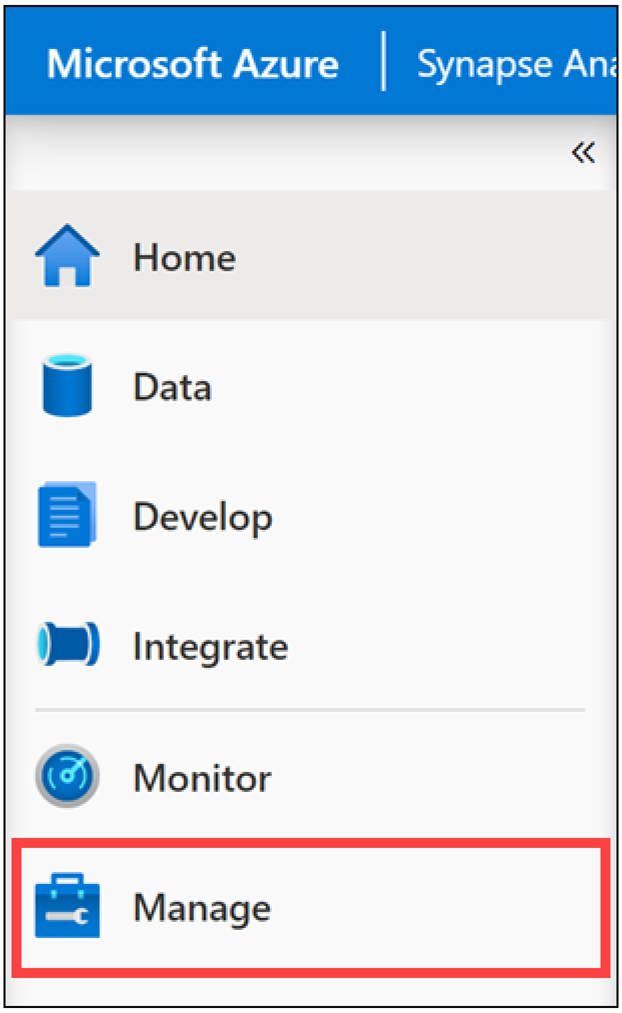
Before starting this lab, ensure you have successfully completed the setup steps to create your lab environment. Then complete the following setup tasks to create a dedicated SQL pool.

> **Note**: The setup tasks will take around 6-7 minutes. You can continue the lab while the script runs.

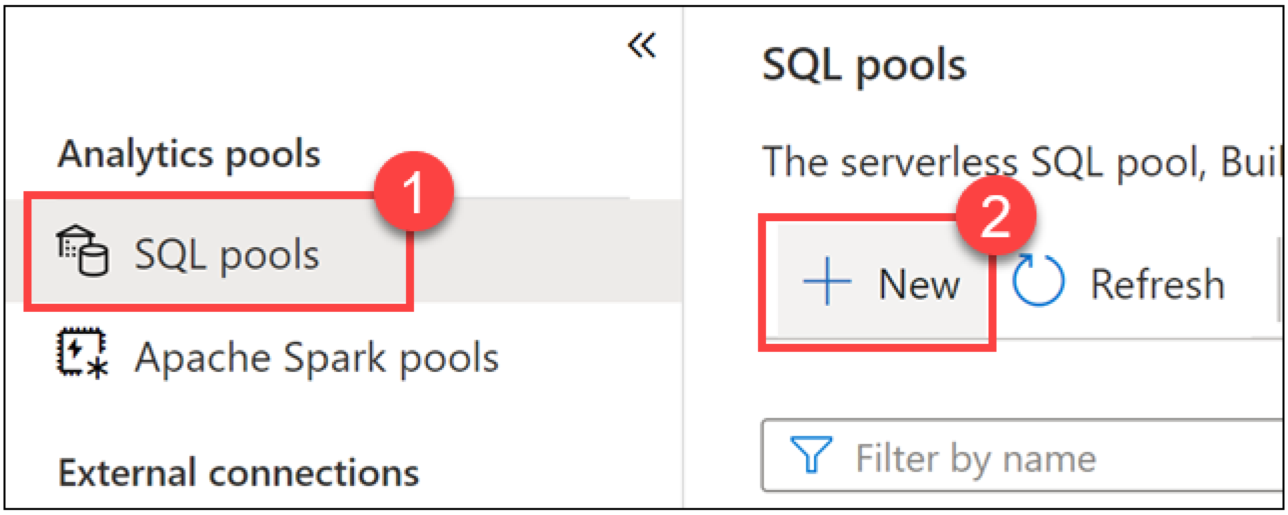
Task 1: Create dedicated SQL pool

1. Open Synapse Studio (https://web.azuresynapse.net/).

2. Select the **Manage** hub.



3. Select **SQL pools** in the left-hand menu, then select **+ New**.



4. In the **Create dedicated SQL pool** page, enter `SQLPool01` (You must use this name exactly as displayed here) for the pool name, and then set the performance level to **DW100c** (move the slider all the way to the left).

5. Click **Review + create**. Then select **Create** on the validation step.

6. Wait until the dedicated SQL pool is created.

> **Important:** Once started, a dedicated SQL pool consumes credits in your Azure subscription until it is paused. If you take a break from this lab, or decide not to complete it; follow the instructions at the

> end of the lab to **pause your SQL pool**

Task 2: Execute PowerShell Script

1. In the hosted VM environment provided for this course, open Powershell in administrator mode, and execute the following to set the execution policy to Unrestricted so you can run the local PowerShell script file:

   ```
   Set-ExecutionPolicy Unrestricted
   ```

   > **Note**: If you receive a prompt that you are installing the module from an untrusted repository, select **Yes to All** to proceed with the setup.

2. Change directories to the root of this repo within your local file system.

   ```
   cd C:\dp-203\data-engineering-ilt-
   deployment\Allfiles\00\artifacts\environment-setup\automation\
   ```

3. Enter the following command to run a PowerShell script that creates objects in the SQL pool:

   ```
   .\setup-sql.ps1
   ```

4. When prompted to sign into Azure, and your browser opens; sign in using your credentials. After signing in, you can close the browser and return to Windows PowerShell.

5. When prompted, sign into your Azure account again (this is required so that the script can manage resources in your Azure subscription - be sure you use the same credentials as before).

6. If you have more than one Azure subscription, when prompted, select the one you want to use in the labs by entering its number in the list of subscriptions.

7. When prompted, enter the name of the resource group containing your Azure Synapse Analytics workspace (such as **data-engineering-synapse-*xxxxxxx***).

8. **Continue on to Exercise 1** while this script is running.

Exercise 1 - Perform Data Exploration in Synapse Studio

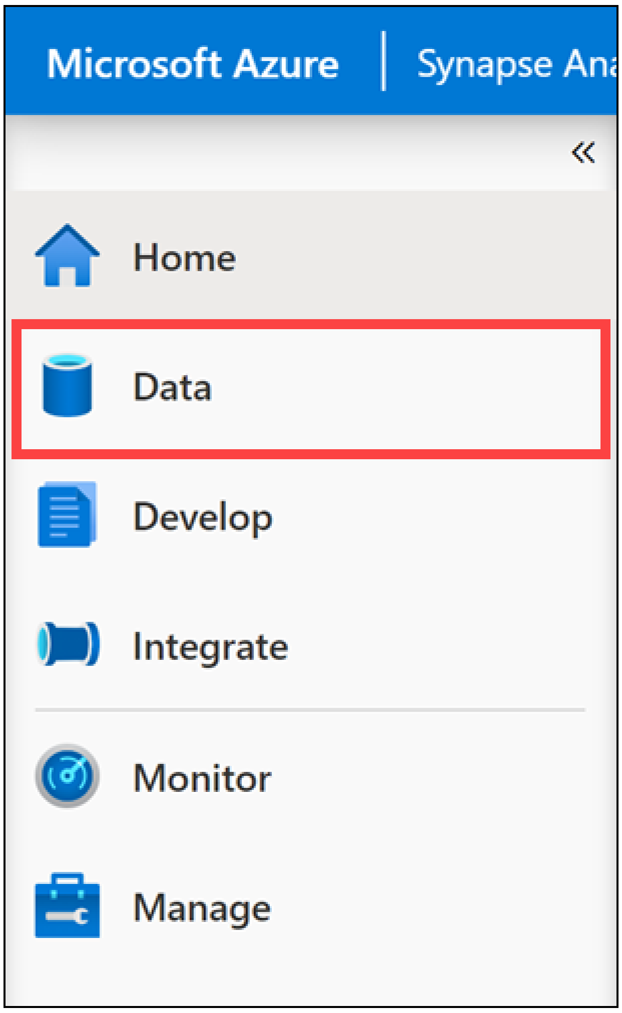
One of the first data engineering tasks typically performed during data ingestion is to explore the data that is to be imported. Data exploration allows engineers to understand better the contents of files being ingested. This process helps to identify any potential data quality issues that might hinder automated ingestion processes. Through exploration, we can gain insights into data types, data quality, and whether any processing needs to be performed on the files prior to importing the data into your data lake or using it for analytics workloads.

The engineers at Tailspin Traders have run into issues ingesting some of their sales data into the data warehouse, and have requested assistance in understanding how Synapse Studio can be used to help them resolve these issues. As the first step of this process, you need to explore the data to understand what is causing the issues they've encountered, and then provide them with a solution.

Task 1: Exploring data using the data previewer in Azure Synapse Studio

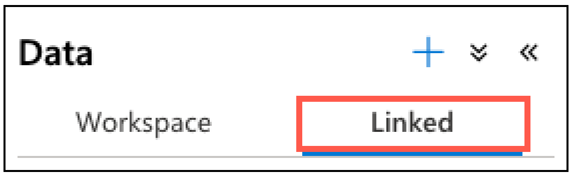
Azure Synapse Studio provides numerous ways to explore data, from a simple preview interface to more complicated programmatic options using Synapse Spark notebooks. In this exercise, you will learn how to use these features to explore, identify, and fix problematic files. You will be exploring CSV files stored in the **wwi-02/sale-poc** folder of the data lake and learning about how to identify and fix issues.

1. In Azure Synapse Studio, navigate to the **Data** hub.

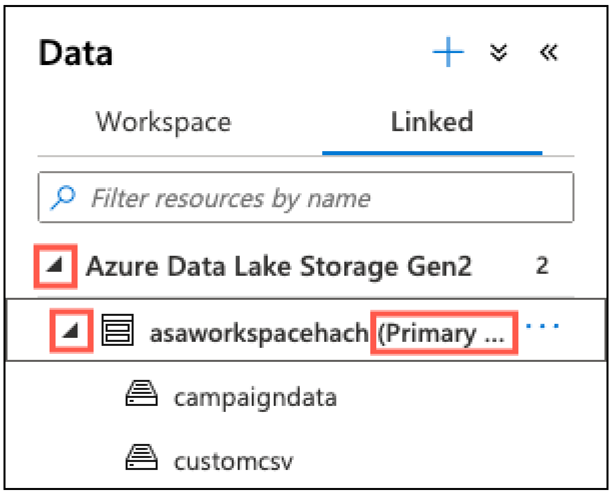
   

   > The Data hub is where you access your provisioned SQL pool databases and SQL serverless databases in your workspace, as well as external data sources, such as storage accounts and other linked services.

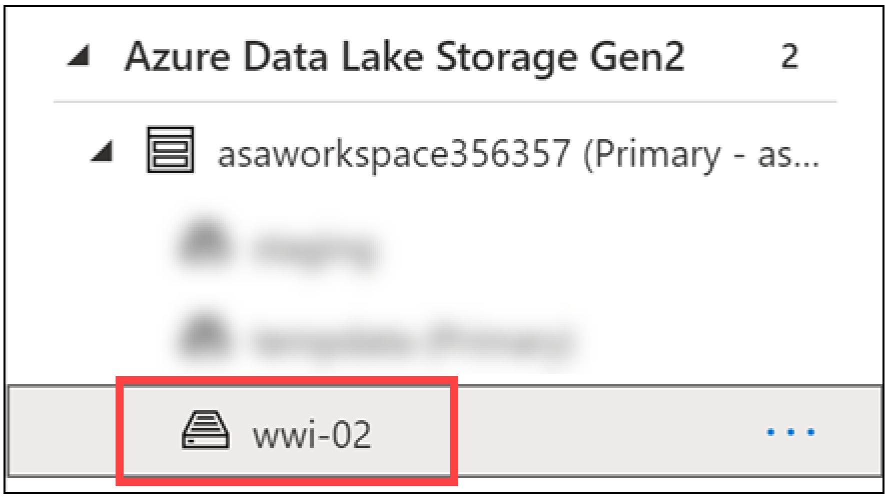
2. We want to access files stored in the workspace's primary data lake, so select the **Linked** tab within the Data hub.

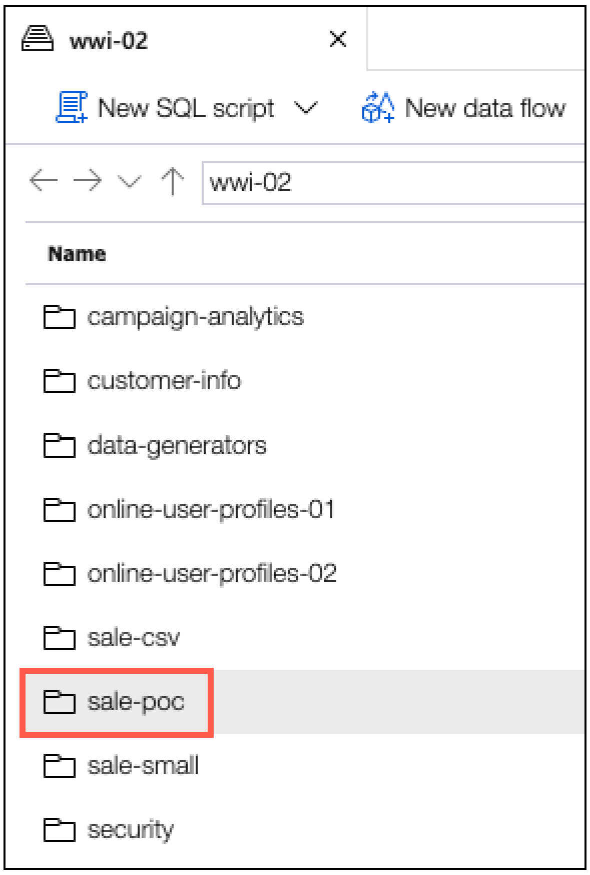
3. On the Linked tab, expand **Azure Data Lake Storage Gen2** and then expand the **Primary** data lake for the workspace.



4. In the list of containers within the primary data lake storage account, select the **wwi-02** container.

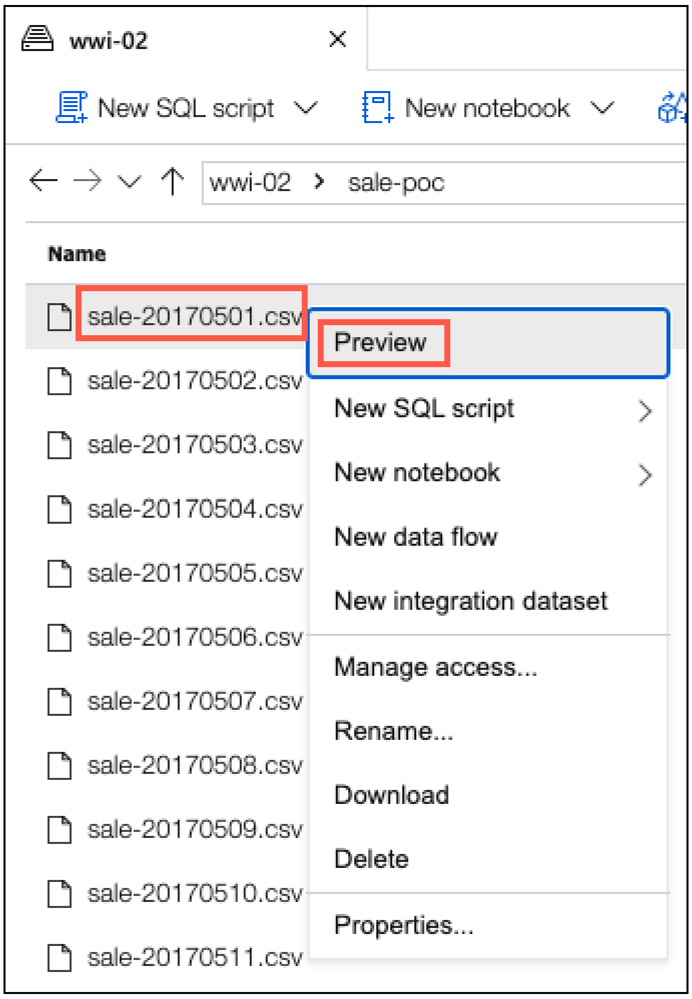

Azure Data Lake Storage Gen2 2
asaworkspace356357 (Primary - as...
wwi-02

5. In the container explorer window, browse to the **sale-poc**.

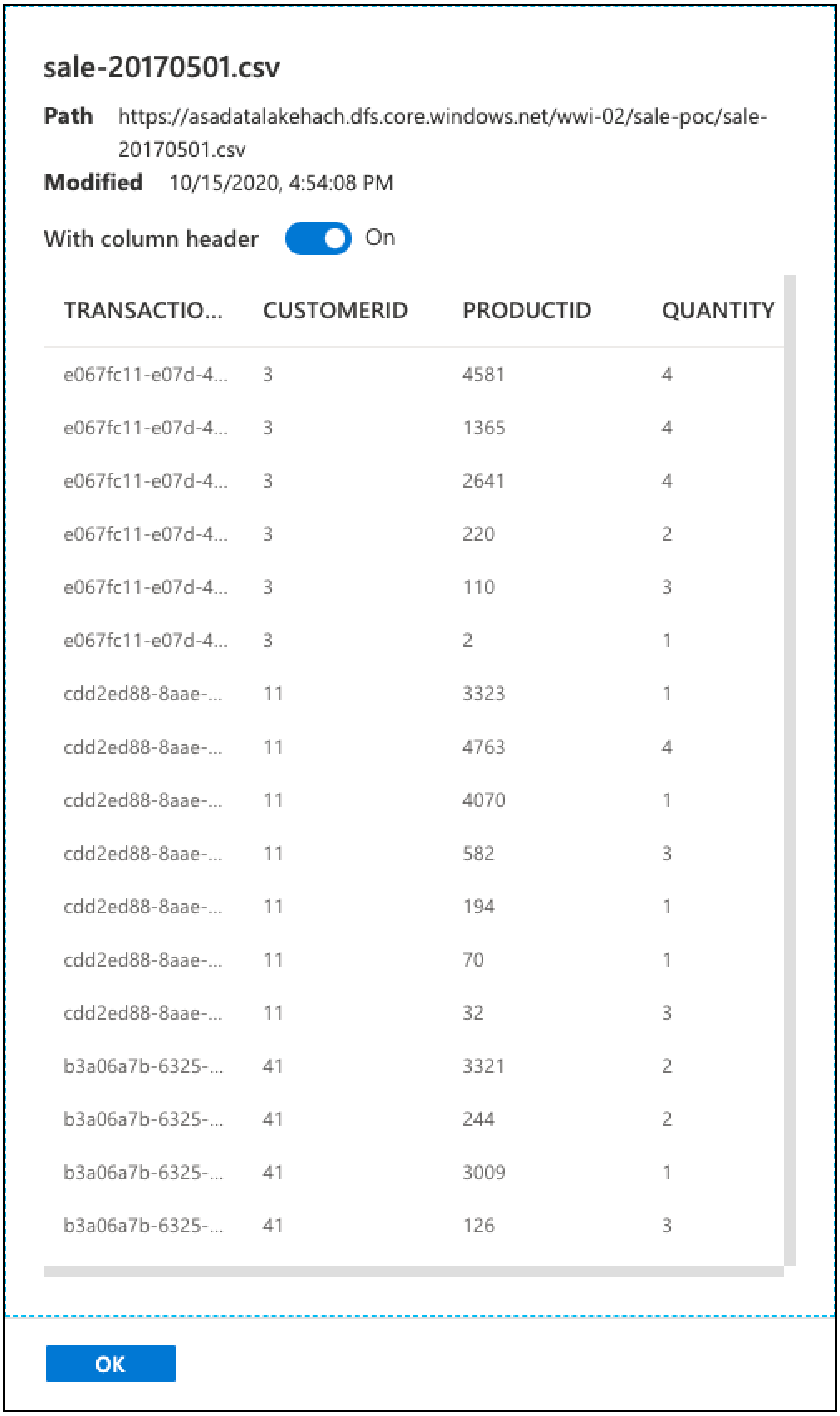


6. The **sale-poc** contains sales data for the month of May, 2017. There are some files in the folder. These files were imported by a temporary process to account for an issue with Tailspin's import process. Let's now take a few minutes to explore some of the files.

7. Right-click the first file in the list, **sale-20170501.csv**, and select **Preview** from the context menu.



8. The preview functionality in Synapse Studio provides an quick and code-free way to examine the contents of a file. This is an effective way of getting a basic understanding of the features (columns) and types of data stored within them for an individual file.



sale-20170501.csv

Path https://asadatalakehach.dfs.core.windows.net/wwi-02/sale-poc/sale-20170501.csv

Modified 10/15/2020, 4:54:08 PM

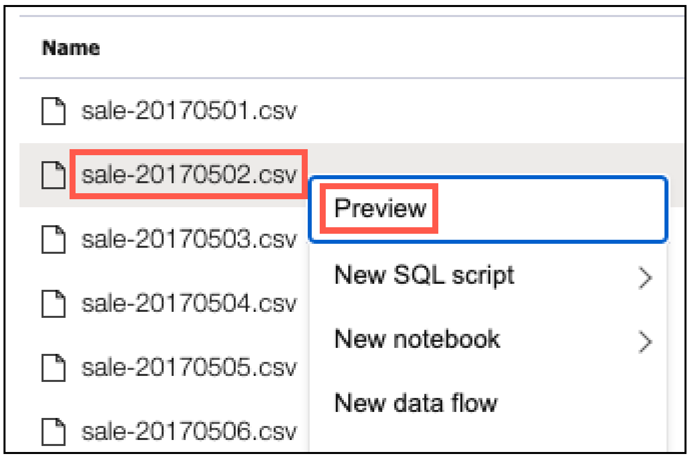
With column header: On

TRANSACTIO...	CUSTOMERID	PRODUCTID	QUANTITY
e067fc11-e07d-4...	3	4581	4
e067fc11-e07d-4...	3	1365	4
e067fc11-e07d-4...	3	2641	4
e067fc11-e07d-4...	3	220	2
e067fc11-e07d-4...	3	110	3
e067fc11-e07d-4...	3	2	1
cdd2ed88-8aae-...	11	3323	1
cdd2ed88-8aae-...	11	4763	4
cdd2ed88-8aae-...	11	4070	1
cdd2ed88-8aae-...	11	582	3
cdd2ed88-8aae-...	11	194	1
cdd2ed88-8aae-...	11	70	1
cdd2ed88-8aae-...	11	32	3
b3a06a7b-6325-...	41	3321	2
b3a06a7b-6325-...	41	244	2
b3a06a7b-6325-...	41	3009	1
b3a06a7b-6325-...	41	126	3

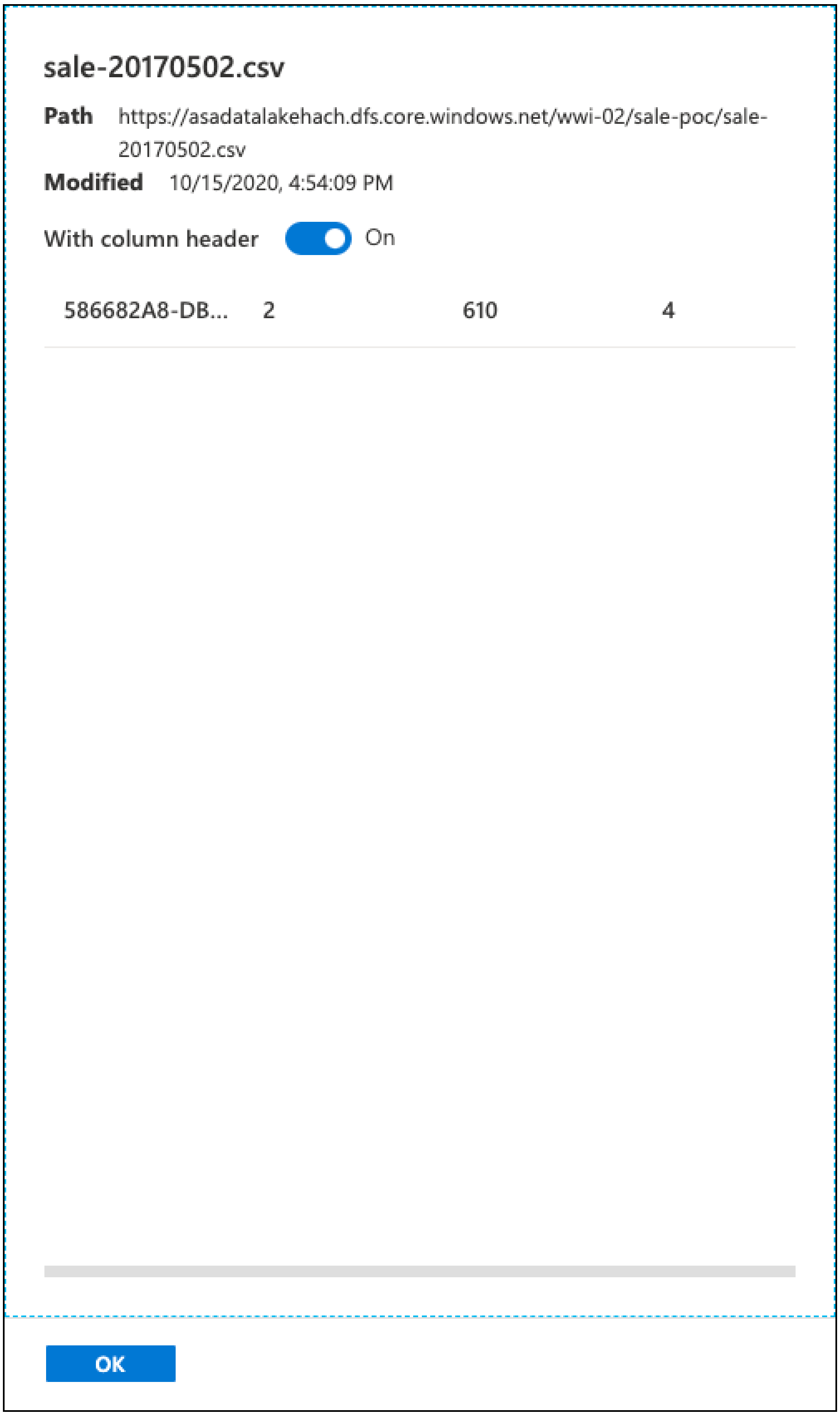
> While in the Preview dialog for **sale-20170501.csv**, take a moment to scroll through the file preview. Scrolling down shows there are a limited number of rows included in the preview, so this is just a glimpse into the structure of the file. Scrolling to the right allows you to see the number and names of the columns contained in the file.

9. Select **OK** to close the preview.

10. When performing data exploration, it is important to look at more than just one file, as it helps to get a more representative sample of the data. Let's look at the next file in the folder. Right-click the **sale-20170502.csv** file and select **Preview** from the context menu.



11. Notice the structure of this file is different from the **sale-20170501.csv** file. No data rows appear in the preview and the column headers appear to contain data and not field names.



sale-20170502.csv

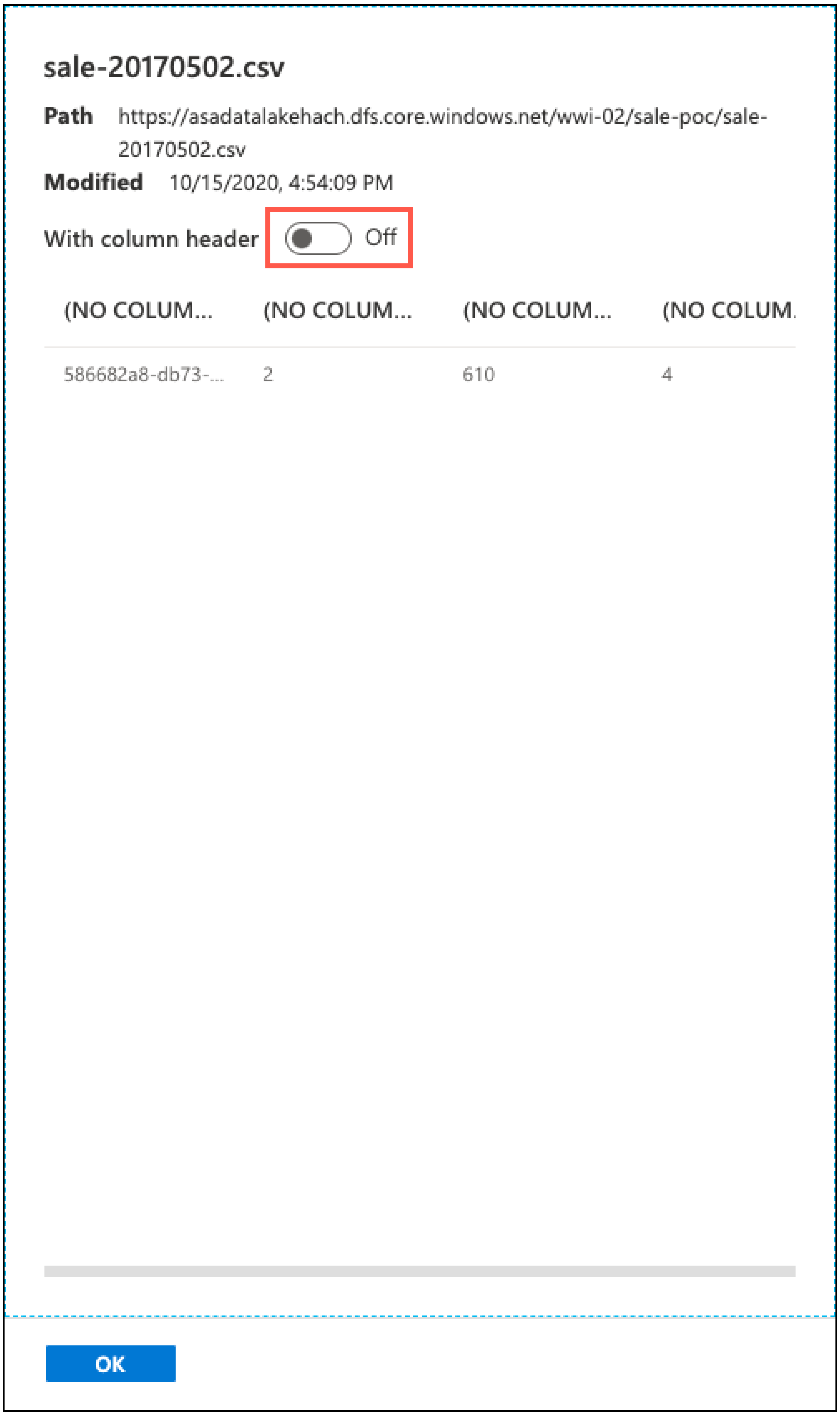
Path https://asadatalakehach.dfs.core.windows.net/wwi-02/sale-poc/sale-20170502.csv

Modified 10/15/2020, 4:54:09 PM

With column header: On

586682A8-DB...	2	610	4

12. It appears this file does not contain column headers, so set the **With column header** option to **off** (which may take a while to change) and inspect the results.



sale-20170502.csv

Path https://asadatalakehach.dfs.core.windows.net/wwi-02/sale-poc/sale-20170502.csv

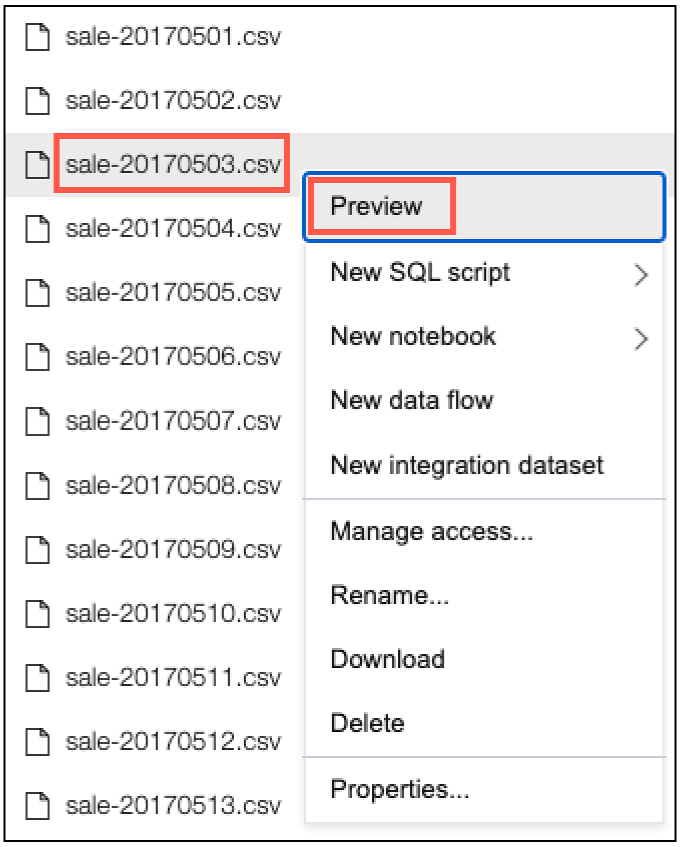
Modified 10/15/2020, 4:54:09 PM

With column header: Off

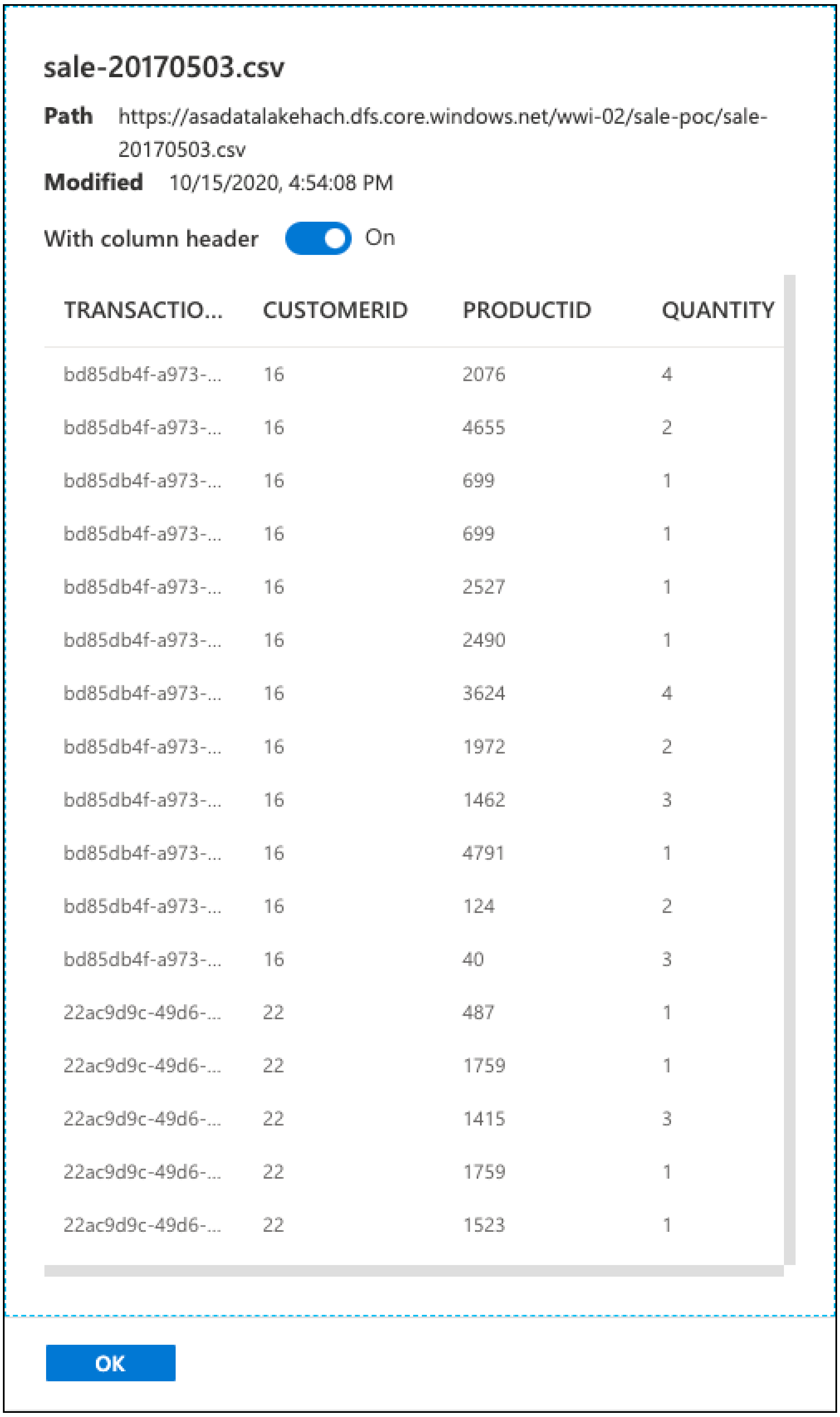
(NO COLUM...	(NO COLUM...	(NO COLUM...	(NO COLUM.
586682a8-db73-...	2	610	4

> Setting the **With column headers** to off verifies that the file does not contain column headers. All columns have "(NO COLUMN NAME)" in the header. This setting moves the data down appropriately, and it appears this is only a single row. By scrolling to the right, you will notice that while there appears to only be a single row, there are many more columns than what we saw when previewing the first file. That file contained 11 columns.

13. Since we have seen two different file structures, let's look at another file to see if we can learn which format is more typical of the files within the **sale-poc** folder. Close the preview of **sale-20170502.csv**, and then open a preview of **sale-20170503.csv**.



14. Verify that the **sale-20170503.csv** file appears to have a structure similar to that found in **20170501.csv**.



sale-20170503.csv

Path https://asadatalakehach.dfs.core.windows.net/wwi-02/sale-poc/sale-20170503.csv

Modified 10/15/2020, 4:54:08 PM

With column header: On

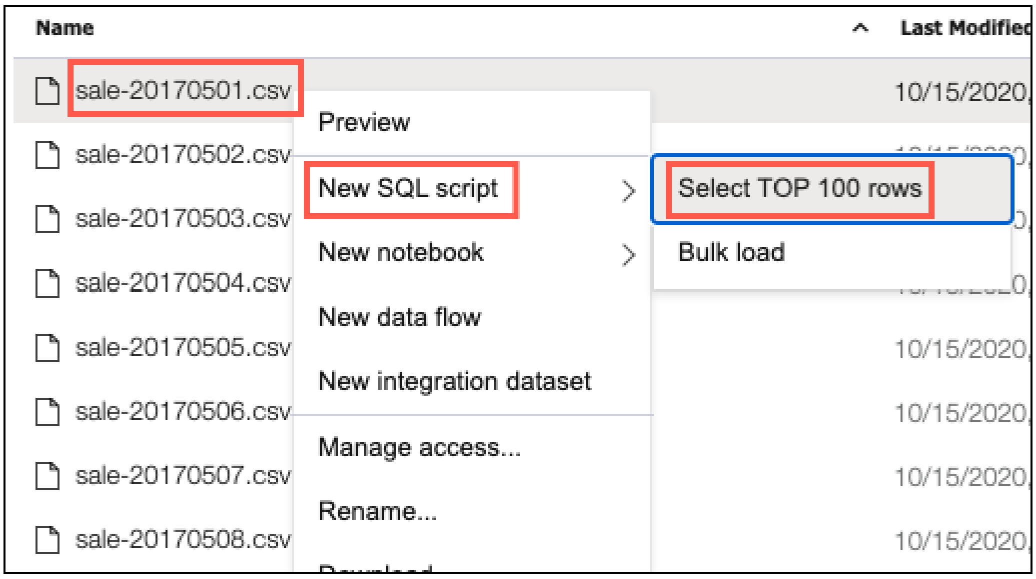
TRANSACTIO...	CUSTOMERID	PRODUCTID	QUANTITY
bd85db4f-a973-...	16	2076	4
bd85db4f-a973-...	16	4655	2
bd85db4f-a973-...	16	699	1
bd85db4f-a973-...	16	699	1
bd85db4f-a973-...	16	2527	1
bd85db4f-a973-...	16	2490	1
bd85db4f-a973-...	16	3624	4
bd85db4f-a973-...	16	1972	2
bd85db4f-a973-...	16	1462	3
bd85db4f-a973-...	16	4791	1
bd85db4f-a973-...	16	124	2
bd85db4f-a973-...	16	40	3
22ac9d9c-49d6-...	22	487	1
22ac9d9c-49d6-...	22	1759	1
22ac9d9c-49d6-...	22	1415	3
22ac9d9c-49d6-...	22	1759	1
22ac9d9c-49d6-...	22	1523	1

15. Select **OK** to close the preview.

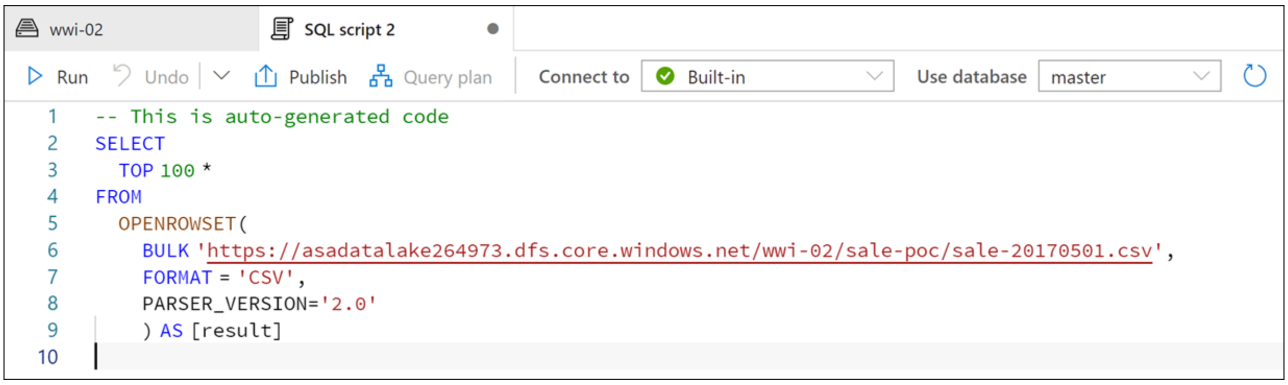
Task 2: Using serverless SQL pools to explore files

The preview functionality in Synapse Studio enables quick exploration of files, but doesn't allow us to look deeper into the data or gain much in the way of insights into files with issues. In this task, we will use the **serverless SQL pools (built-in)** functionality of Synapse to explore these files using T-SQL.

1. Right-click the **sale-20170501.csv** file again, this time selecting **New SQL Script** and **Select TOP 100 rows** from the context menu.



2. A new SQL script tab will open in Synapse Studio containing a SELECT statement to read the first 100 rows of the file. This provides another way to examine the contents of files. By limiting the number of rows being examined, we can speed up the exploration process, as queries to load all the data within the files will run slower.

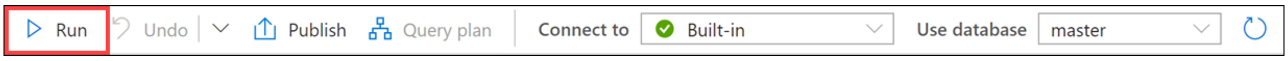


```sql
-- This is auto-generated code
SELECT
  TOP 100 *
FROM
  OPENROWSET(
    BULK 'https://asadatalake264973.dfs.core.windows.net/wwi-02/sale-poc/sale-20170501.csv',
    FORMAT = 'CSV',
    PARSER_VERSION='2.0'
    ) AS [result]
```

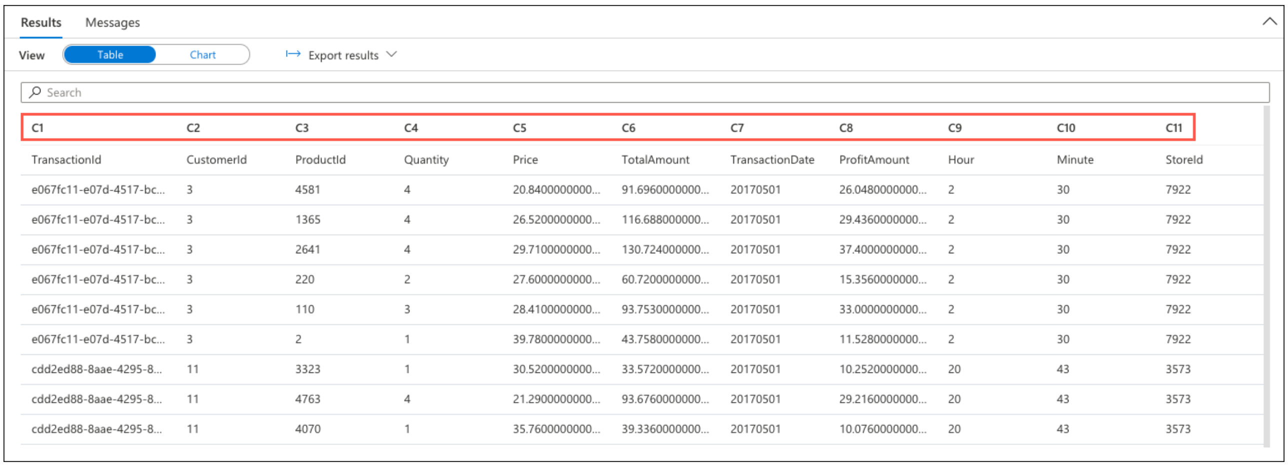
> **Tip**: Hide the script's **Properties** pane to make it easier to see the script.

T-SQL queries against files stored in the data lake leverage the OPENROWSET function, which can be referenced in the FROM clause of a query as if it were a table. It supports bulk operations through a built-in BULK provider that enables data from a file to be read and returned as a rowset. To learn more, you can review the to OPENROWSET documentation.

3. Now, select **Run** on the toolbar to execute the query.



4. In the **Results** pane, observe the output.

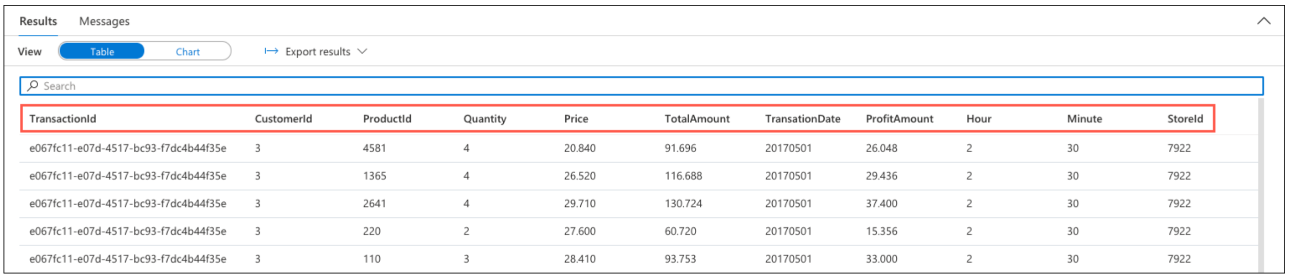


C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
TransactionId	CustomerId	ProductId	Quantity	Price	TotalAmount	TransactionDate	ProfitAmount	Hour	Minute	StoreId
e067fc11-e07d-4517-bc...	3	4581	4	20.8400000000...	91.6960000000...	20170501	26.0480000000...	2	30	7922
e067fc11-e07d-4517-bc...	3	1365	4	26.5200000000...	116.688000000...	20170501	29.4360000000...	2	30	7922
e067fc11-e07d-4517-bc...	3	2641	4	29.7100000000...	130.724000000...	20170501	37.4000000000...	2	30	7922
e067fc11-e07d-4517-bc...	3	220	2	27.6000000000...	60.7200000000...	20170501	15.3560000000...	2	30	7922
e067fc11-e07d-4517-bc...	3	110	3	28.4100000000...	93.7530000000...	20170501	33.0000000000...	2	30	7922
e067fc11-e07d-4517-bc...	3	2	1	39.7800000000...	43.7580000000...	20170501	11.5280000000...	2	30	7922
cdd2ed88-8aae-4295-8...	11	3323	1	30.5200000000...	33.5720000000...	20170501	10.2520000000...	20	43	3573
cdd2ed88-8aae-4295-8...	11	4763	4	21.2900000000...	93.6760000000...	20170501	29.2160000000...	20	43	3573
cdd2ed88-8aae-4295-8...	11	4070	1	35.7600000000...	39.3360000000...	20170501	10.0760000000...	20	43	3573

> In the results, you will notice that the first row, containing the column headers, is rendered as a data row, and the columns are assigned names **C1** - **C11**. You can use the FIRSTROW parameter of the OPENROWSET function to specify the number of the first fow of the file to display as data. The default value is 1, so if a file contains a header row, the value can be set to 2 to skip the column headers. You can then specify the schema associated with the file using the `WITH` clause.

5. Modify the query as shown below to skip the header row and specify the names of the columns in the resultset; replacing *SUFFIX* with the unique resource suffix for your storage account:

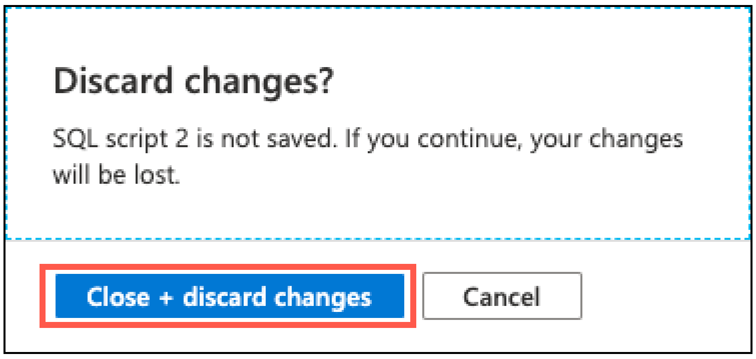
```
SELECT
    TOP 100 *
FROM
    OPENROWSET(
        BULK 'https://asadatalakeSUFFIX.dfs.core.windows.net/wwi-02/sale-
poc/sale-20170501.csv',
        FORMAT = 'CSV',
        PARSER_VERSION='2.0',
        FIRSTROW = 2
    ) WITH (
        [TransactionId] varchar(50),
        [CustomerId] int,
        [ProductId] int,
        [Quantity] int,
        [Price] decimal(10,3),
        [TotalAmount] decimal(10,3),
        [TransactionDate] varchar(8),
        [ProfitAmount] decimal(10,3),
        [Hour] int,
        [Minute] int,
        [StoreId] int
    ) AS [result]
```



TransactionId	CustomerId	ProductId	Quantity	Price	TotalAmount	TransationDate	ProfitAmount	Hour	Minute	StoreId
e067fc11-e07d-4517-bc93-f7dc4b44f35e	3	4581	4	20.840	91.696	20170501	26.048	2	30	7922
e067fc11-e07d-4517-bc93-f7dc4b44f35e	3	1365	4	26.520	116.688	20170501	29.436	2	30	7922
e067fc11-e07d-4517-bc93-f7dc4b44f35e	3	2641	4	29.710	130.724	20170501	37.400	2	30	7922
e067fc11-e07d-4517-bc93-f7dc4b44f35e	3	220	2	27.600	60.720	20170501	15.356	2	30	7922
e067fc11-e07d-4517-bc93-f7dc4b44f35e	3	110	3	28.410	93.753	20170501	33.000	2	30	7922

> Using the OPENROWSET function, you can now use T-SQL syntax to further explore your data. For example, you can use a WHERE clause to check various fields for *null* or other values that might need to be handled prior to using the data for advanced analytics workloads. With the schema specified, you can refer to fields by name to make this processes easier.

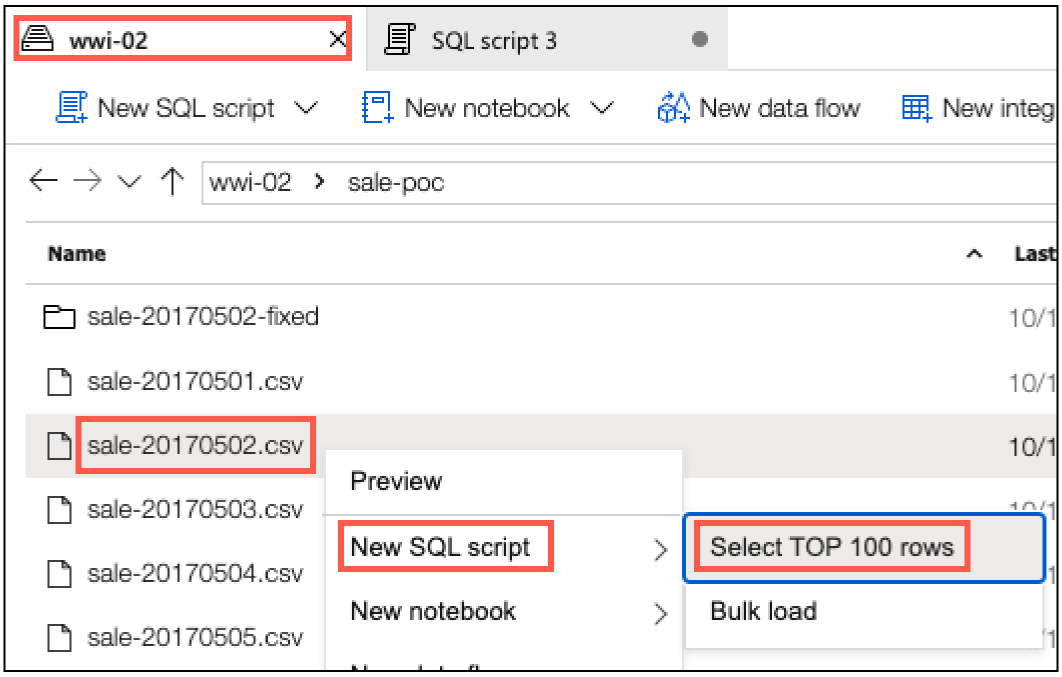
6. Close the SQL script tab. If prompted, select **Close + discard changes**.



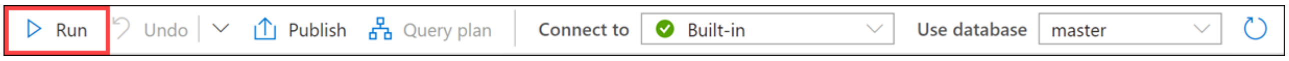
Discard changes?

SQL script 2 is not saved. If you continue, your changes will be lost.

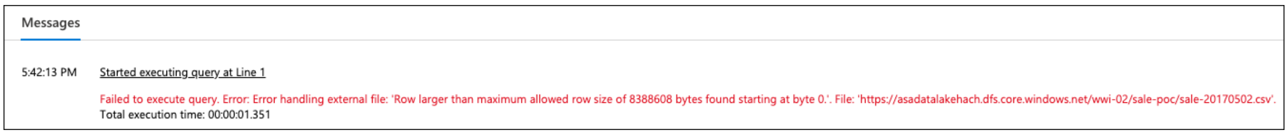
7. We saw while using the **Preview** functionality that the **sale-20170502.csv** file is poorly formed. Let's see if we can learn more about the data in this file using T-SQL. In the **wwi-02** tab, right-click the **sale-20170502.csv** file and select **New SQL script** and **Select TOP 100 rows**.



8. Run the automatically generated query.



9. Observe that the query results in the error, *Error handling external file: 'Row larger than maximum allowed row size of 8388608 bytes found starting at byte 0.'*.


```
Messages

5:42:13 PM   Started executing query at Line 1
             Failed to execute query. Error: Error handling external file: 'Row larger than maximum allowed row size of 8388608 bytes found starting at byte 0.'. File: 'https://asadatalakehach.dfs.core.windows.net/wwi-02/sale-poc/sale-20170502.csv'.
             Total execution time: 00:00:01.351
```

> This error aligns with what we saw in the preview window for this file. In the preview we saw the data being separated in to columns, but all of the data was in a single row. This implies the data is being split into columns using the default field delimiter (comma). What appears to be missing, however, is a row terminator, `\r`.

10. Close the query tab, discarding the changes, and in the **wwi-02** tab, right-click the **sale-20170502.csv** file and select **Download**. This downloads the file and opens it in the browser.

11. Review the data in the browser, and note that there are no line-terminators; all of the data is in a single line (which wraps in the browser display).

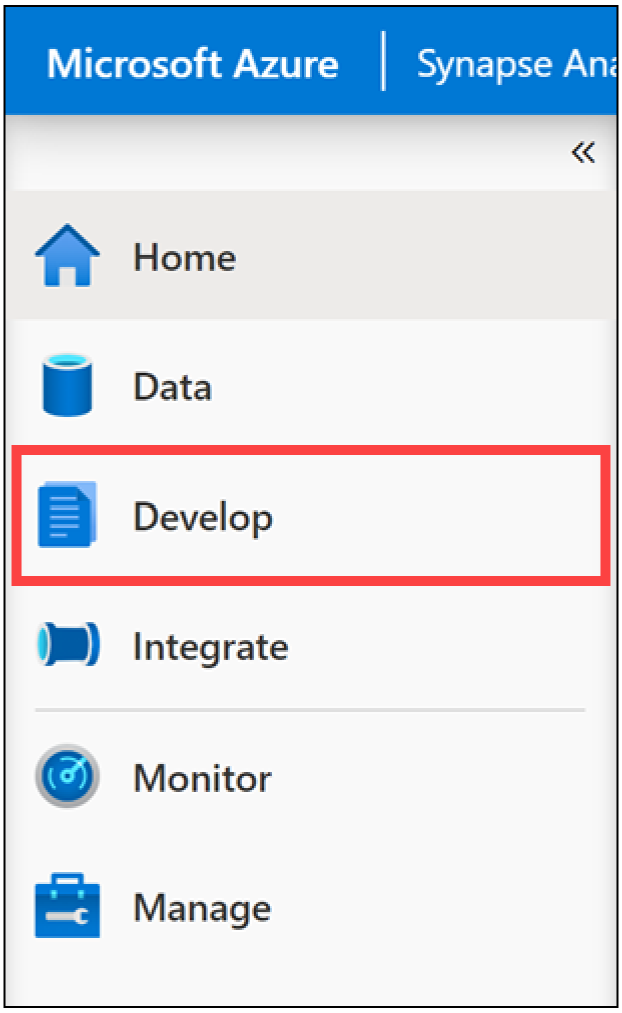
12. Close the browser tab containing the contents of the **sale-20170502.csv** file.

    To fix the file, we need to use code. T-SQL and Synapse Pipelines do not have the ability to efficiently handle this type of issue. To address the problems with this file, we need to use a Synapse Spark notebook.

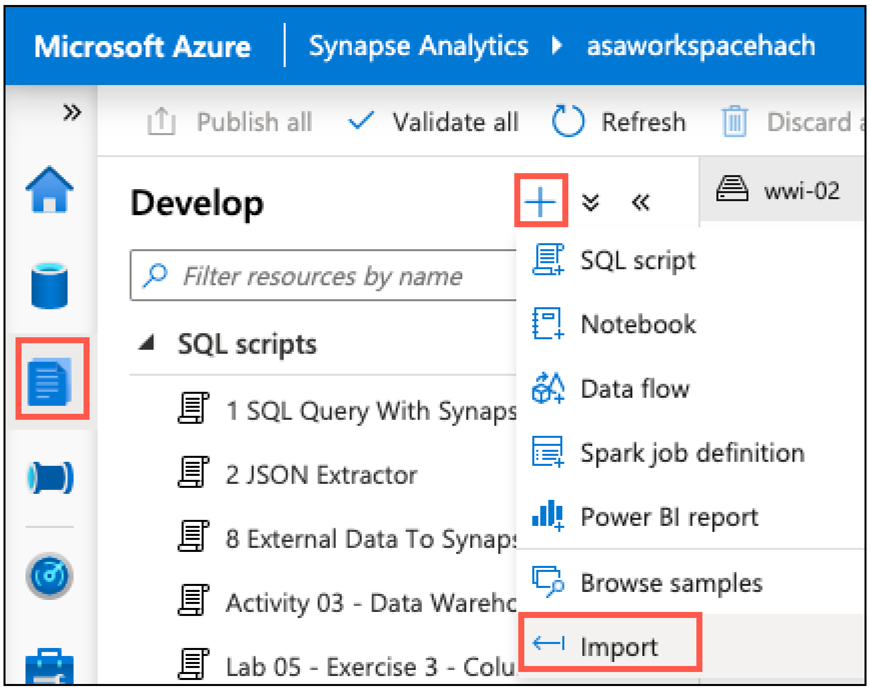
Task 3: Exploring and fixing data with Synapse Spark

In this task, you will use a Synapse Spark notebook to explore a few of the files in the **wwi-02/sale-poc** folder in the data lake. You will also use Python code to fix the issues with the **sale-20170502.csv** file, so all the files in the directory can be ingested using a Synapse Pipeline later.

1. In Synapse Studio, open the **Develop** hub.



2. In the **+** menu, select **Import**.

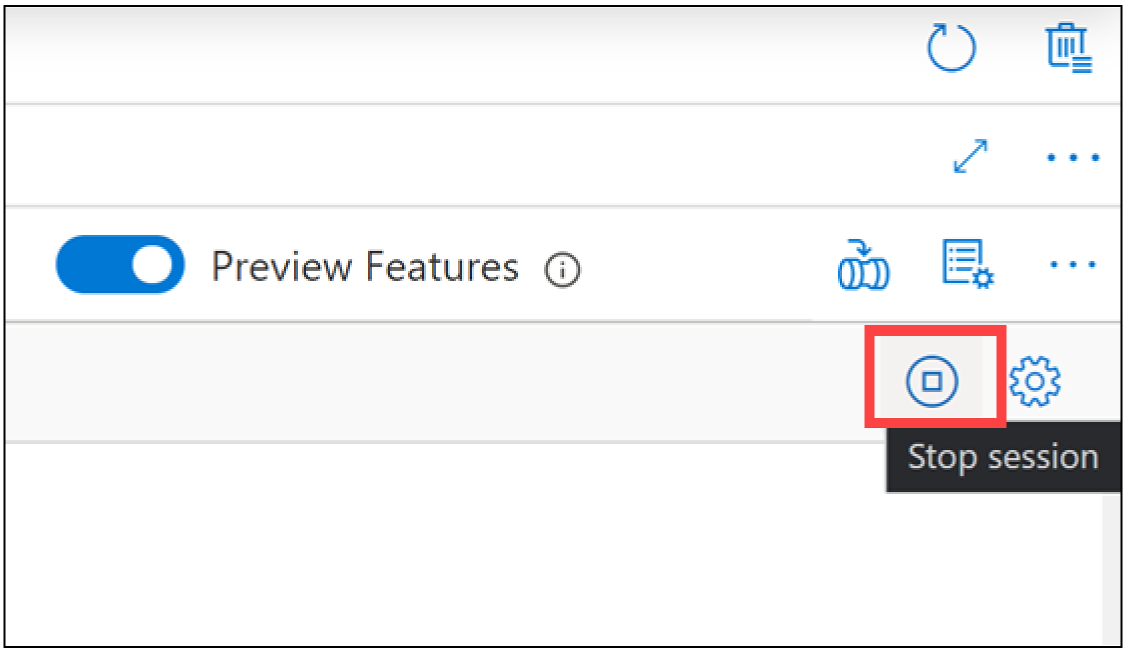


3. Import the **Explore with Spark.ipynb** notebook in the C:\dp-203\data-engineering-ilt-deployment\Allfiles\synapse-apache-spark-notebooks folder.

4. Follow the instructions contained within the notebook to complete the remainder of this task, attaching it to the **SparkPool01** Spark pool. Note that the first cell may take some time to run as the Spark pool

must be started.

5. When you have completed the **Explore with Spark** notebook, select on the **Stop Session** button on the right hand side of the notebook toolbar to release the Spark cluster for the next exercise.



Exercise 2 - Ingesting data with Spark notebooks in Azure Synapse Analytics

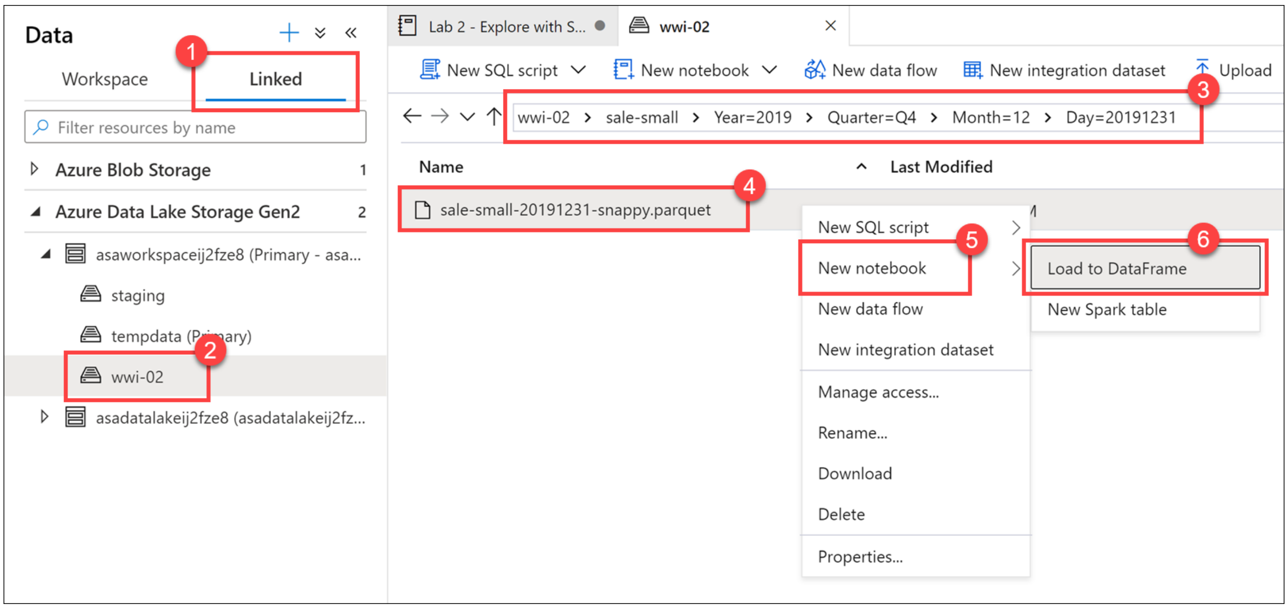
Tailwind Traders has unstructured and semi-structured files from various data sources. Their data engineers want to use their Spark expertise to explore, ingest, and transform these files.

You recommend using Synapse Notebooks, which are integrated in the Azure Synapse Analytics workspace and used from within Synapse Studio.

Task 1: Ingest and explore Parquet files from a data lake with Apache Spark for Azure Synapse

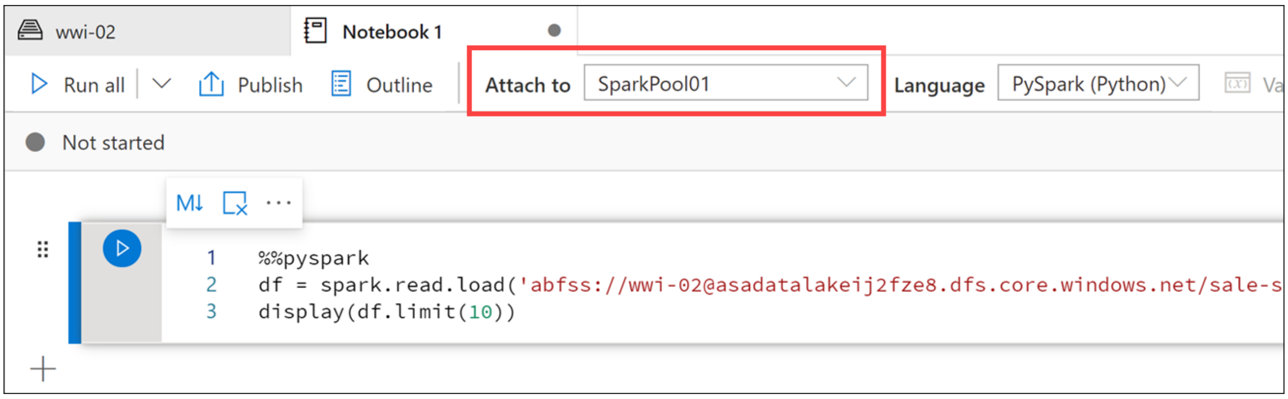
1. In Azure Synapse Studio, select the **Data** hub.

2. On the **Linked** tab, in the **wwi-02** container, browse to the *sale-small/Year=2019/Quarter=Q4/Month=12/Day=20191231* folder. Then right-click the Parquet file, select **New notebook**, and then select **Load to DataFrame**.



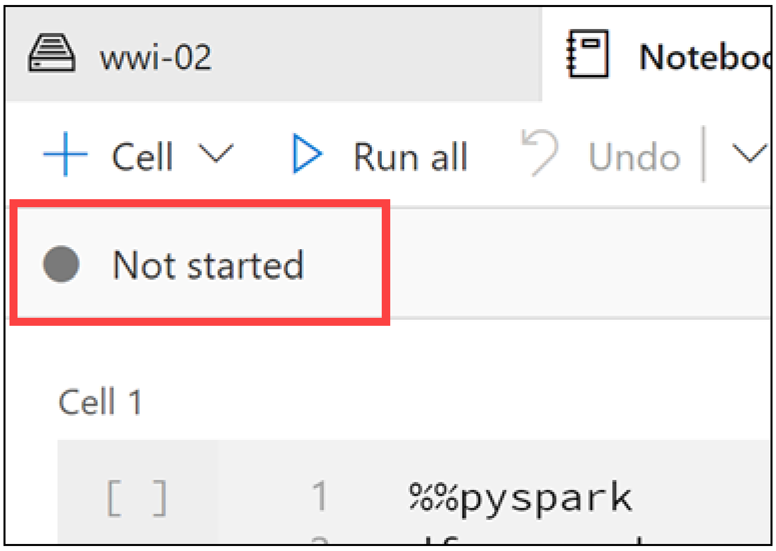
This generates a notebook with PySpark code to load the data in a Spark dataframe and display 10 rows with the header.

3. Attach the **SparkPool01** Spark pool to the notebook, but **do not run/execute the cell at this stage** - you need to create a variable for the name of your data lake first.



```
%%pyspark
df = spark.read.load('abfss://wwi-02@asadatalakeij2fze8.dfs.core.windows.net/sale-s
display(df.limit(10))
```

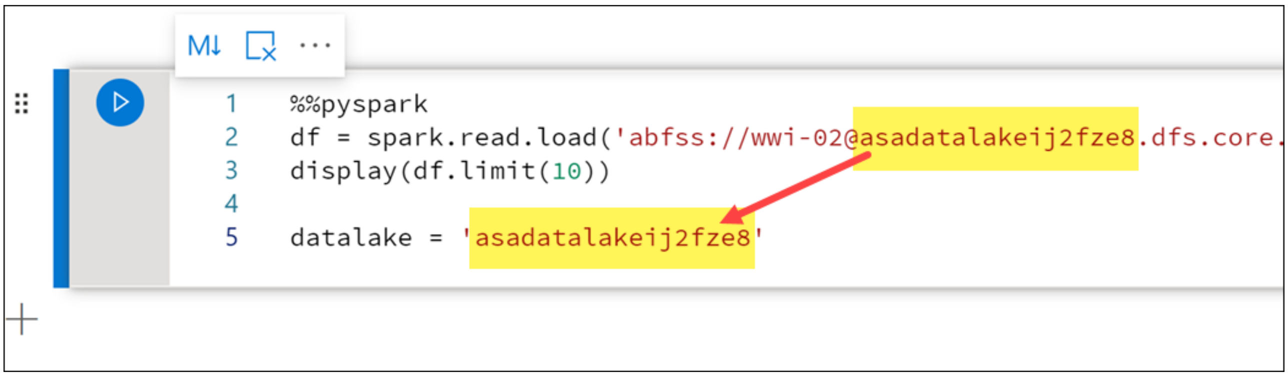
The Spark pool provides the compute for all notebook operations. If you look under the notebook toolbar, you'll see that the pool has not started. When you run a cell in the notebook while the pool is idle, the pool will start and allocate resources. This is a one-time operation until the pool auto-pauses from being idle for too long.



> The auto-pause settings are configured on the Spark pool configuration in the Manage hub.

4. Add the following beneath the code in the cell to define a variable named **datalake** whose value is the name of the primary storage account (replace *SUFFIX* with the unique suffix for your data store):

```
datalake = 'asadatalakeSUFFIX'
```

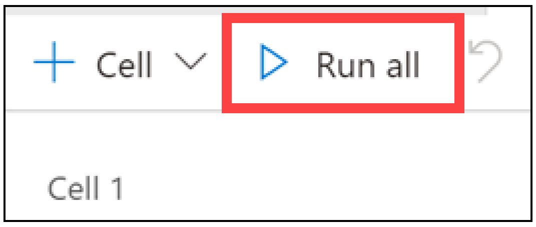


```
%%pyspark
df = spark.read.load('abfss://wwi-02@asadatalakeij2fze8.dfs.core.
display(df.limit(10))

datalake = 'asadatalakeij2fze8'
```

This variable will be used in subsequent cells.

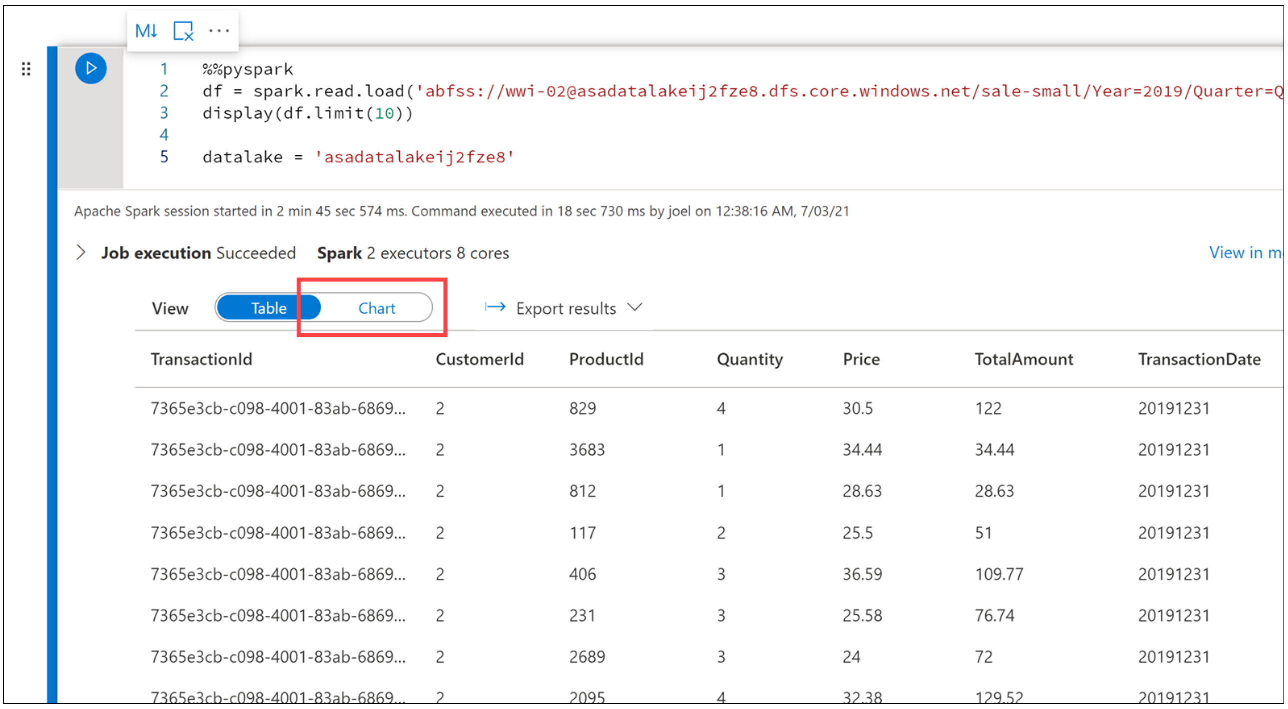
5. Select **Run all** on the notebook toolbar to execute the notebook.



> **Note:** The first time you run a notebook in a Spark pool, Azure Synapse creates a new session. This can take approximately 2-3 minutes.

> **Note:** To run just the cell, either hover over the cell and select the *Run cell* icon to the left of the cell, or select the cell then type **Ctrl+Enter** on your keyboard.

6. After the cell run is complete, change the View to **Chart** in the cell output.



```
%%pyspark
df = spark.read.load('abfss://wwi-02@asadatalakeij2fze8.dfs.core.windows.net/sale-small/Year=2019/Quarter=Q
display(df.limit(10))

datalake = 'asadatalakeij2fze8'
```

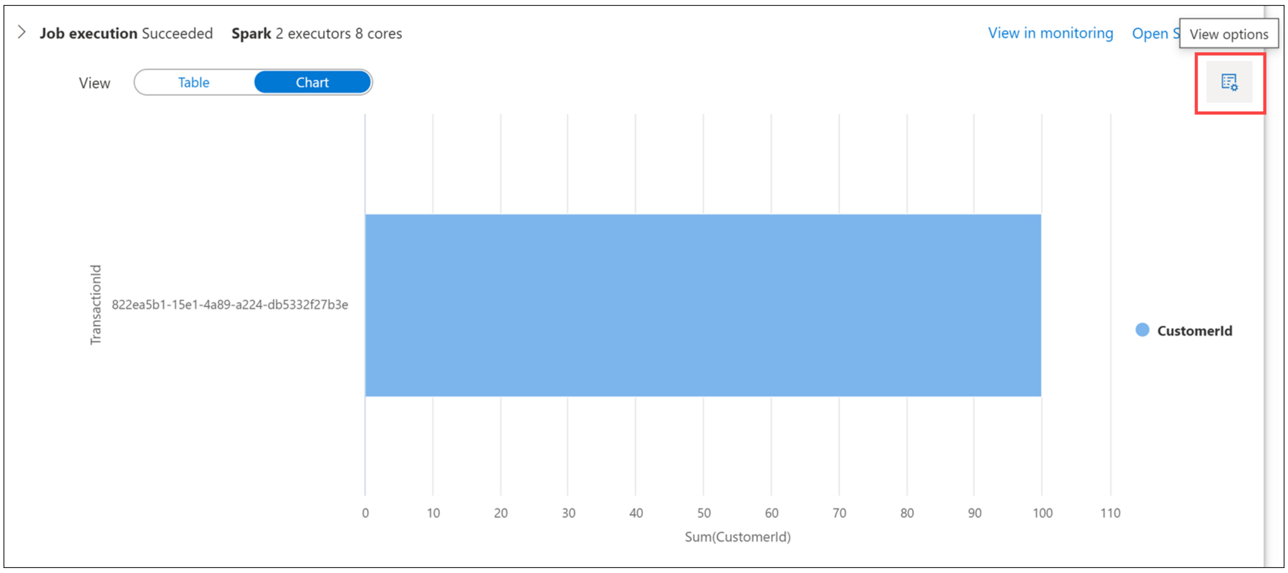
Apache Spark session started in 2 min 45 sec 574 ms. Command executed in 18 sec 730 ms by joel on 12:38:16 AM, 7/03/21

Job execution Succeeded **Spark** 2 executors 8 cores

TransactionId	CustomerId	ProductId	Quantity	Price	TotalAmount	TransactionDate
7365e3cb-c098-4001-83ab-6869...	2	829	4	30.5	122	20191231
7365e3cb-c098-4001-83ab-6869...	2	3683	1	34.44	34.44	20191231
7365e3cb-c098-4001-83ab-6869...	2	812	1	28.63	28.63	20191231
7365e3cb-c098-4001-83ab-6869...	2	117	2	25.5	51	20191231
7365e3cb-c098-4001-83ab-6869...	2	406	3	36.59	109.77	20191231
7365e3cb-c098-4001-83ab-6869...	2	231	3	25.58	76.74	20191231
7365e3cb-c098-4001-83ab-6869...	2	2689	3	24	72	20191231
7365e3cb-c098-4001-83ab-6869...	2	2095	4	32.38	129.52	20191231

By default, the cell outputs to a table view when we use the **display()** function. We see in the output the sales transaction data stored in the Parquet file for December 31, 2019. Let's select the **Chart** visualization to see a different view of the data.

7. Select the **View options** button to the right.



Job execution Succeeded **Spark** 2 executors 8 cores

8. Set **Key** to **ProductId** and **Values** to **TotalAmount**, then select **Apply**.

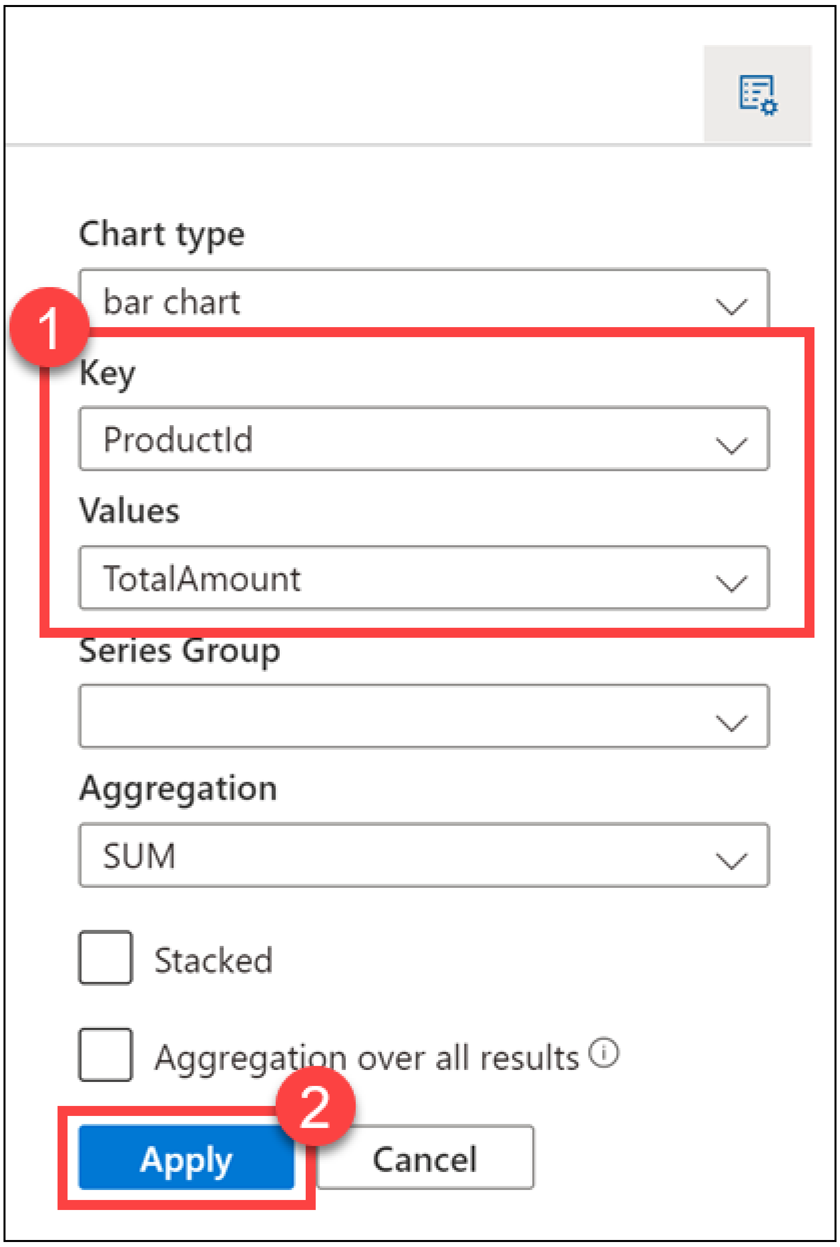


Chart type: bar chart

Key: ProductId

Values: TotalAmount

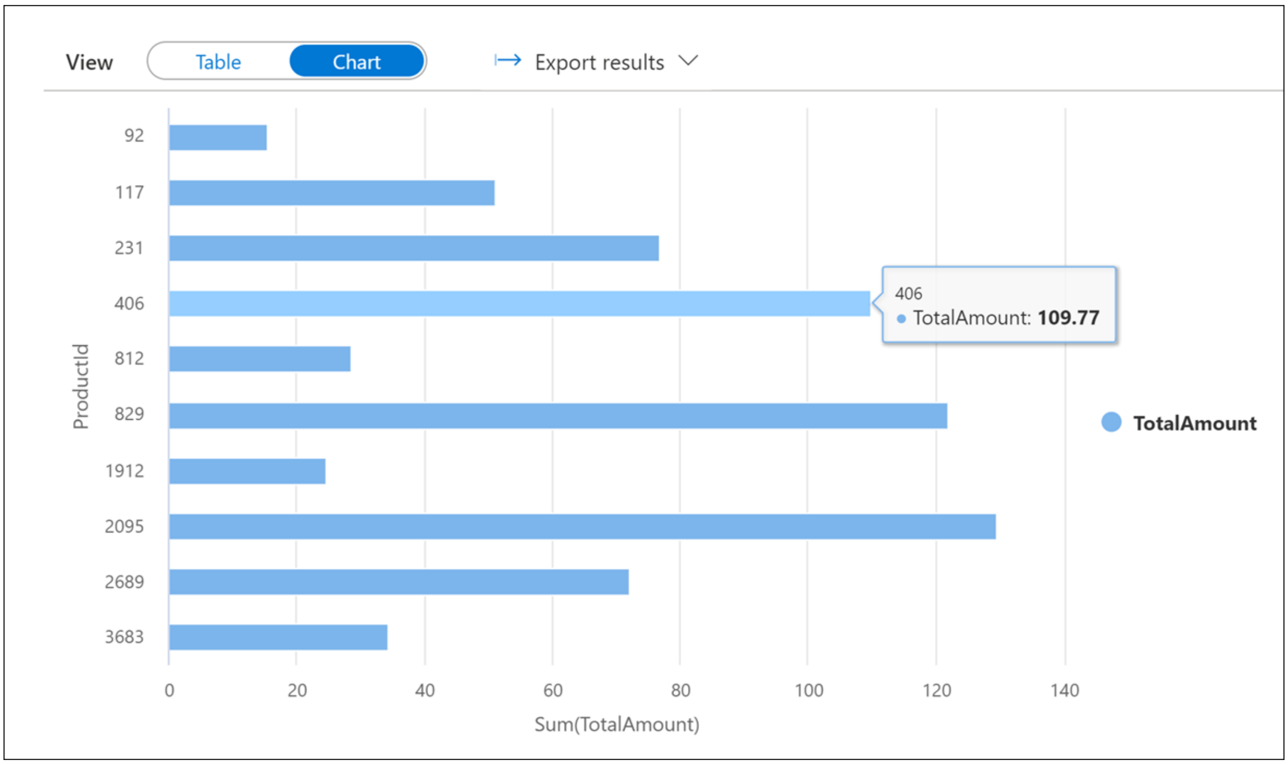
Series Group:

Aggregation: SUM

[ ] Stacked

[ ] Aggregation over all results

9. The chart visualization is displayed. Hover over the bars to view details.



10. Create a new cell underneath by selecting **+ Code**.

11. The Spark engine can analyze the Parquet files and infer the schema. To do this, enter the following in the new cell and run it:

```
df.printSchema()
```

Your output should look like the following:

```
root
 |-- TransactionId: string (nullable = true)
 |-- CustomerId: integer (nullable = true)
 |-- ProductId: short (nullable = true)
 |-- Quantity: byte (nullable = true)
 |-- Price: decimal(38,18) (nullable = true)
 |-- TotalAmount: decimal(38,18) (nullable = true)
 |-- TransactionDate: integer (nullable = true)
 |-- ProfitAmount: decimal(38,18) (nullable = true)
 |-- Hour: byte (nullable = true)
 |-- Minute: byte (nullable = true)
 |-- StoreId: short (nullable = true)
```

Spark evaluates the file contents to infer the schema. This automatic inference is usually sufficient for data exploration and most transformation tasks. However, when you load data to an external resource like a SQL table, sometimes you need to declare your own schema and apply that to the dataset. For now, the schema looks good.

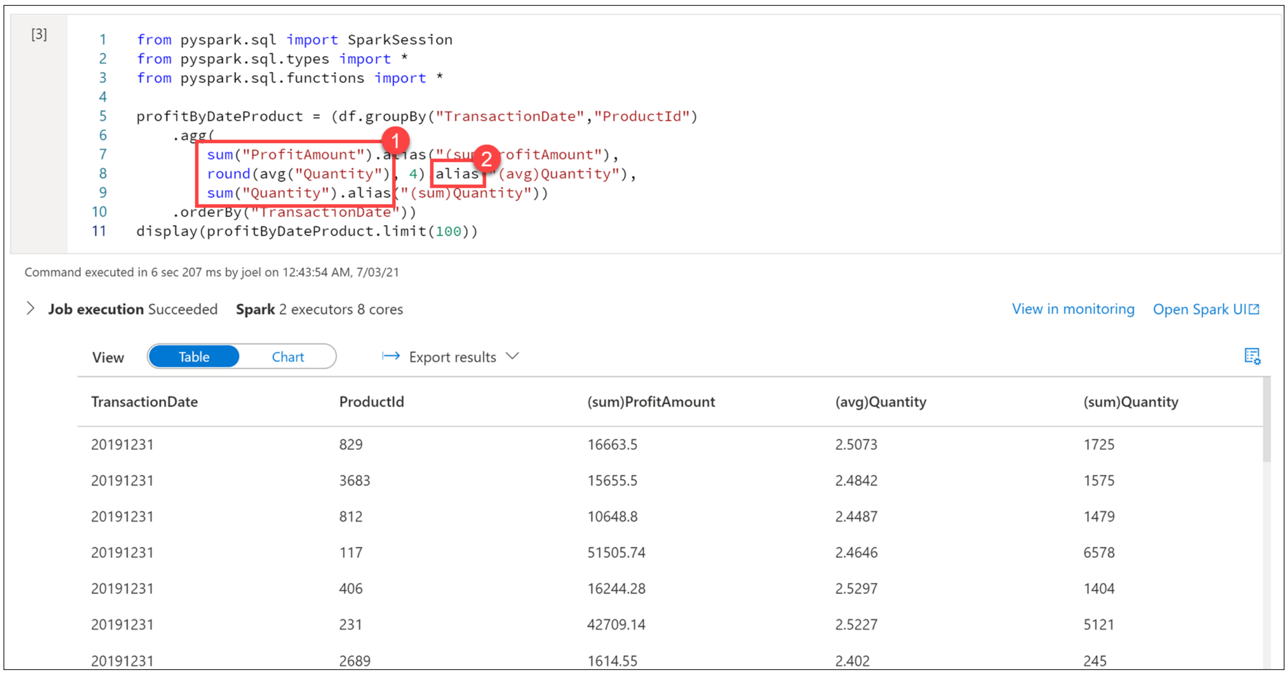
12. Now let's use aggregates and grouping operations to better understand the data. Create a new code cell and enter the following, then run the cell:

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

profitByDateProduct = (df.groupBy("TransactionDate","ProductId")
    .agg(
        sum("ProfitAmount").alias("(sum)ProfitAmount"),
        round(avg("Quantity"), 4).alias("(avg)Quantity"),
        sum("Quantity").alias("(sum)Quantity"))
    .orderBy("TransactionDate"))
display(profitByDateProduct.limit(100))
```

> We import required Python libraries to use aggregation functions and types defined in the schema to successfully execute the query.

The output shows the same data we saw in the chart above, but now with **sum** and **avg** aggregates. Notice that we use the **alias** method to change the column names.



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

profitByDateProduct = (df.groupBy("TransactionDate","ProductId")
    .agg(
        sum("ProfitAmount").alias("(sum)ProfitAmount"),
        round(avg("Quantity"), 4).alias("(avg)Quantity"),
        sum("Quantity").alias("(sum)Quantity"))
    .orderBy("TransactionDate"))
display(profitByDateProduct.limit(100))
```

Command executed in 6 sec 207 ms by joel on 12:43:54 AM, 7/03/21

Job execution Succeeded **Spark** 2 executors 8 cores

TransactionDate	ProductId	(sum)ProfitAmount	(avg)Quantity	(sum)Quantity
20191231	829	16663.5	2.5073	1725
20191231	3683	15655.5	2.4842	1575
20191231	812	10648.8	2.4487	1479
20191231	117	51505.74	2.4646	6578
20191231	406	16244.28	2.5297	1404
20191231	231	42709.14	2.5227	5121
20191231	2689	1614.55	2.402	245

13. Keep the notebook open for the next excercise.

Exercise 3 - Transforming data with DataFrames in Spark pools in Azure Synapse Analytics

In addition to the sales data, Tailwind Traders has customer profile data from an e-commerce system that provides top product purchases for each visitor of the site (customer) over the past 12 months. This data is stored within JSON files in the data lake. They have struggled with ingesting, exploring, and transforming these JSON files and want your guidance. The files have a hierarchical structure that they want to flatten before loading into relational data stores. They also wish to apply grouping and aggregate operations as part of the data engineering process. You recommend using Synapse Notebooks to explore and apply data transformations on the JSON files.

Task 1: Query and transform JSON data with Apache Spark for Azure Synapse

1. Create a new code cell in the Spark notebook, enter the following code and run the cell:

   ```
   df = (spark.read \
           .option('inferSchema', 'true') \
           .json('abfss://wwi-02@' + datalake + '.dfs.core.windows.net/online-
   user-profiles-02/*.json', multiLine=True)
       )

   df.printSchema()
   ```

   > The **datalake** variable you created in the first cell is used here as part of the file path.

   Your output should look like the following:

   ```
   root
   |-- topProductPurchases: array (nullable = true)
   |    |-- element: struct (containsNull = true)
   |    |    |-- itemsPurchasedLast12Months: long (nullable = true)
   |    |    |-- productId: long (nullable = true)
   |-- visitorId: long (nullable = true)
   ```

   > Notice that we are selecting all JSON files within the **online-user-profiles-02** directory. Each JSON file contains several rows, which is why we specified the **multiLine=True** option. Also, we set the **inferSchema** option to **true**, which instructs the Spark engine to review the files and create a schema based on the nature of the data.

2. We have been using Python code in these cells up to this point. If we want to query the files using SQL syntax, one option is to create a temporary view of the data within the dataframe. Run the following code in a new code cell to create a view named **user_profiles**:

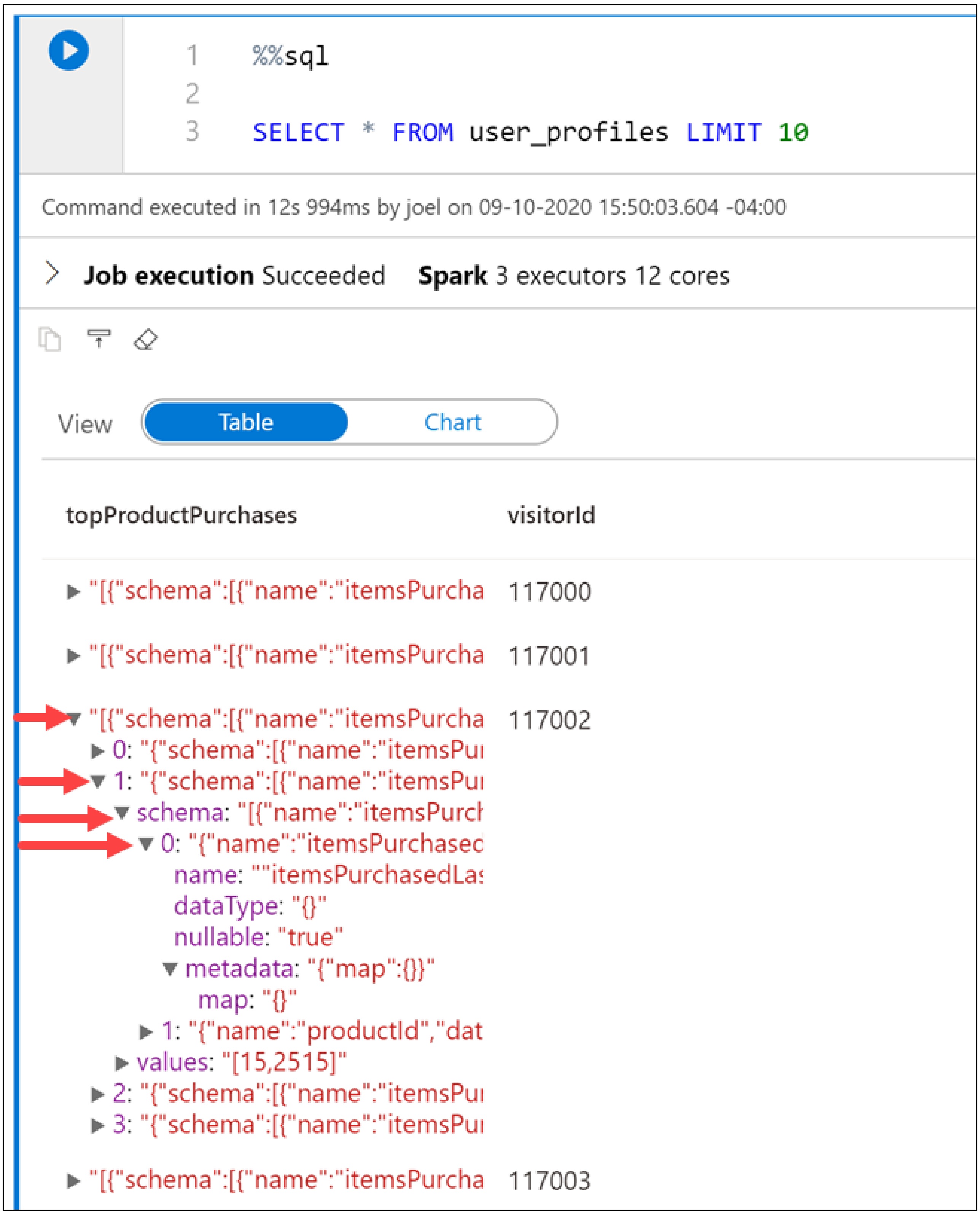
   ```
   # create a view called user_profiles
   df.createOrReplaceTempView("user_profiles")
   ```

3. Create a new code cell. Since we want to use SQL instead of Python, we use the **%%sql** *magic* to set the language of the cell to SQL. Execute the following code in the cell:

   ```
   %%sql

   SELECT * FROM user_profiles LIMIT 10
   ```

Notice that the output shows nested data for **topProductPurchases**, which includes an array of **productId** and **itemsPurchasedLast12Months** values. You can expand the fields by clicking the right triangle in each row.



```
%%sql

SELECT * FROM user_profiles LIMIT 10
```

Command executed in 12s 994ms by joel on 09-10-2020 15:50:03.604 -04:00

Job execution Succeeded Spark 3 executors 12 cores

View Table Chart

```
topProductPurchases                              visitorId
▶ "[{"schema":[{"name":"itemsPurcha              117000
▶ "[{"schema":[{"name":"itemsPurcha              117001
▼ "[{"schema":[{"name":"itemsPurcha              117002
  ▶ 0: "{"schema":[{"name":"itemsPu
  ▼ 1: "{"schema":[{"name":"itemsPu
    ▼ schema: "[{"name":"itemsPurch
      ▼ 0: "{"name":"itemsPurchasec
          name: ""itemsPurchasedLas
          dataType: "{}"
          nullable: "true"
        ▼ metadata: "{"map":{}}"
            map: "{}"
      ▶ 1: "{"name":"productId","dat
    ▶ values: "[15,2515]"
  ▶ 2: "{"schema":[{"name":"itemsPu
  ▶ 3: "{"schema":[{"name":"itemsPu
▶ "[{"schema":[{"name":"itemsPurcha              117003
```

This makes analyzing the data a bit difficult. This is because the JSON file contents look like the following:

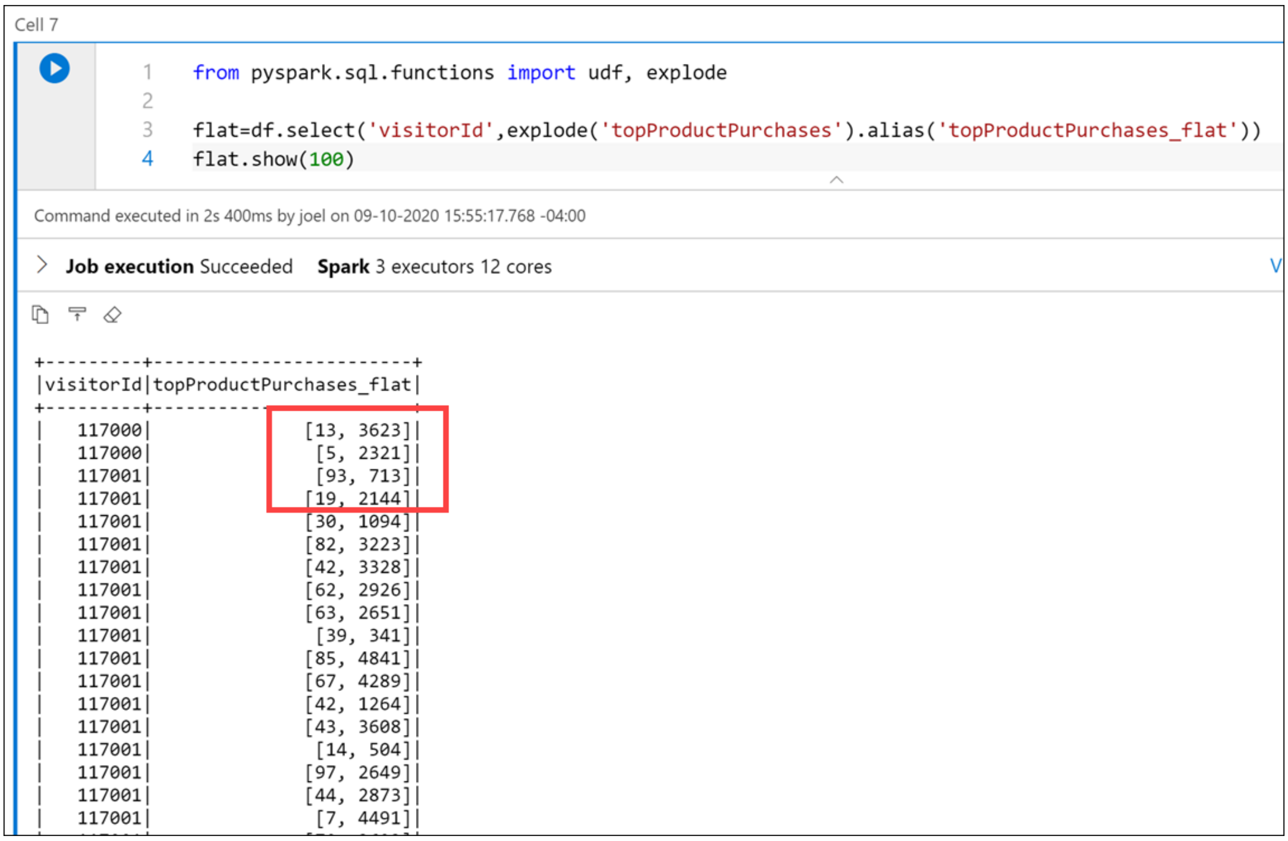
```
[
    {
        "visitorId": 9529082,
        "topProductPurchases": [
            {
                "productId": 4679,
                "itemsPurchasedLast12Months": 26
            },
            {
                "productId": 1779,
                "itemsPurchasedLast12Months": 32
            },
            {
                "productId": 2125,
                "itemsPurchasedLast12Months": 75
            },
            {
                "productId": 2007,
                "itemsPurchasedLast12Months": 39
            },
            {
                "productId": 1240,
                "itemsPurchasedLast12Months": 31
            },
            {
                "productId": 446,
                "itemsPurchasedLast12Months": 39
            },
            {
                "productId": 3110,
                "itemsPurchasedLast12Months": 40
            },
            {
                "productId": 52,
                "itemsPurchasedLast12Months": 2
            },
            {
                "productId": 978,
                "itemsPurchasedLast12Months": 81
            },
            {
                "productId": 1219,
                "itemsPurchasedLast12Months": 56
            },
            {
                "productId": 2982,
                "itemsPurchasedLast12Months": 59
            }
        ]
    },
    {
        ...
    },
    {
        ...
        }
    ]
```

4. PySpark contains a special **explode** function, which returns a new row for each element of the array. This will help flatten the **topProductPurchases** column for better readability or for easier querying. Run the following in a new code cell:

   ```
   from pyspark.sql.functions import udf, explode

   flat=df.select('visitorId',explode('topProductPurchases').alias('topProductP
   urchases_flat'))
   flat.show(100)
   ```

   In this cell, we created a new dataframe named **flat** that includes the **visitorId** field and a new aliased field named **topProductPurchases_flat**. As you can see, the output is a bit easier to read and, by extension, easier to query.



Cell 7

```
from pyspark.sql.functions import udf, explode

flat=df.select('visitorId',explode('topProductPurchases').alias('topProductPurchases_flat'))
flat.show(100)
```

Command executed in 2s 400ms by joel on 09-10-2020 15:55:17.768 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

```
+---------+------------------------+
|visitorId|topProductPurchases_flat|
+---------+------------------------+
|   117000|              [13, 3623]|
|   117000|               [5, 2321]|
|   117001|               [93, 713]|
|   117001|              [19, 2144]|
|   117001|              [30, 1094]|
|   117001|              [82, 3223]|
|   117001|              [42, 3328]|
|   117001|              [62, 2926]|
|   117001|              [63, 2651]|
|   117001|               [39, 341]|
|   117001|              [85, 4841]|
|   117001|              [67, 4289]|
|   117001|              [42, 1264]|
|   117001|              [43, 3608]|
|   117001|               [14, 504]|
|   117001|              [97, 2649]|
|   117001|              [44, 2873]|
|   117001|               [7, 4491]|
```

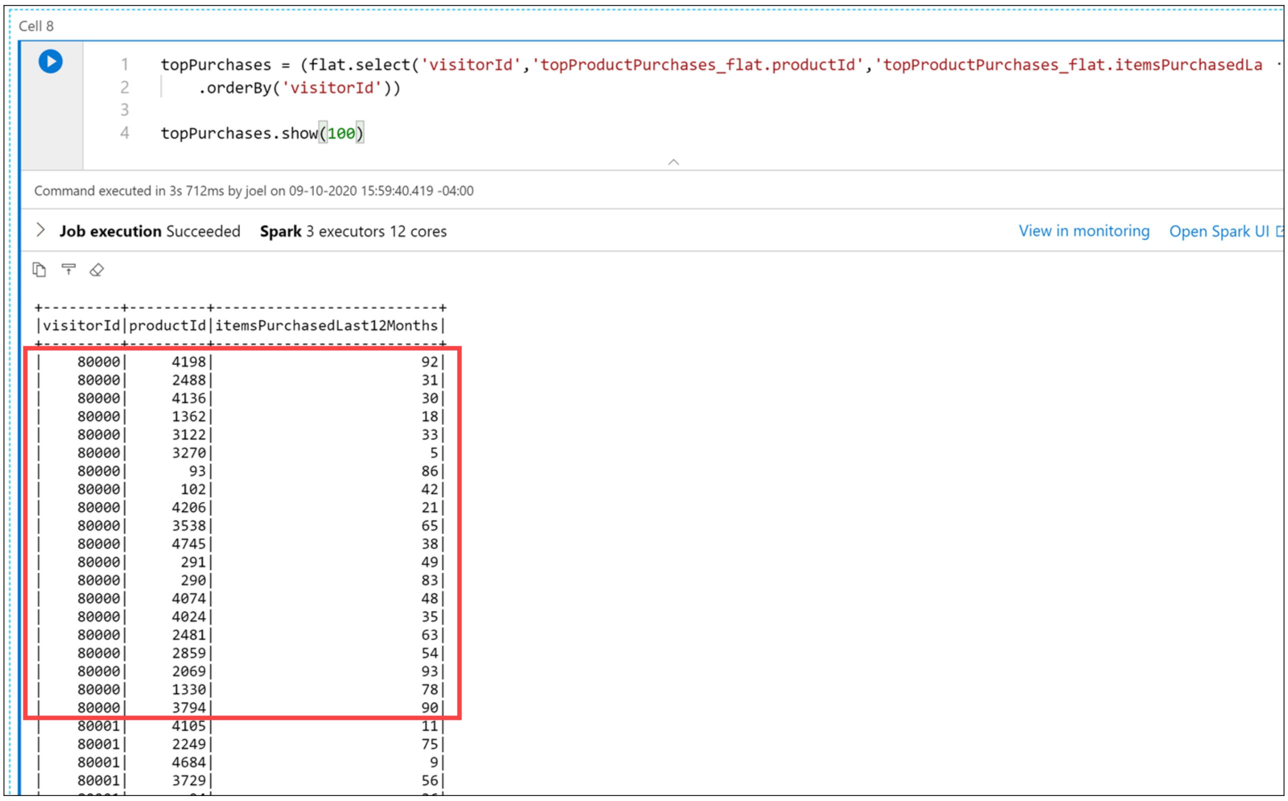
5. Create a new cell and execute the following code to create a new flattened version of the dataframe that extracts the **topProductPurchases_flat.productId** and **topProductPurchases_flat.itemsPurchasedLast12Months** fields to create new rows for each data combination:

   ```
   topPurchases =
   (flat.select('visitorId','topProductPurchases_flat.productId','topProductPur
   chases_flat.itemsPurchasedLast12Months')
   ```

```
        .orderBy('visitorId'))

    topPurchases.show(100)
```

In the output, notice that we now have multiple rows for each **visitorId**.



Cell 8

```
topPurchases = (flat.select('visitorId','topProductPurchases_flat.productId','topProductPurchases_flat.itemsPurchasedLa
    .orderBy('visitorId'))

topPurchases.show(100)
```

Command executed in 3s 712ms by joel on 09-10-2020 15:59:40.419 -04:00

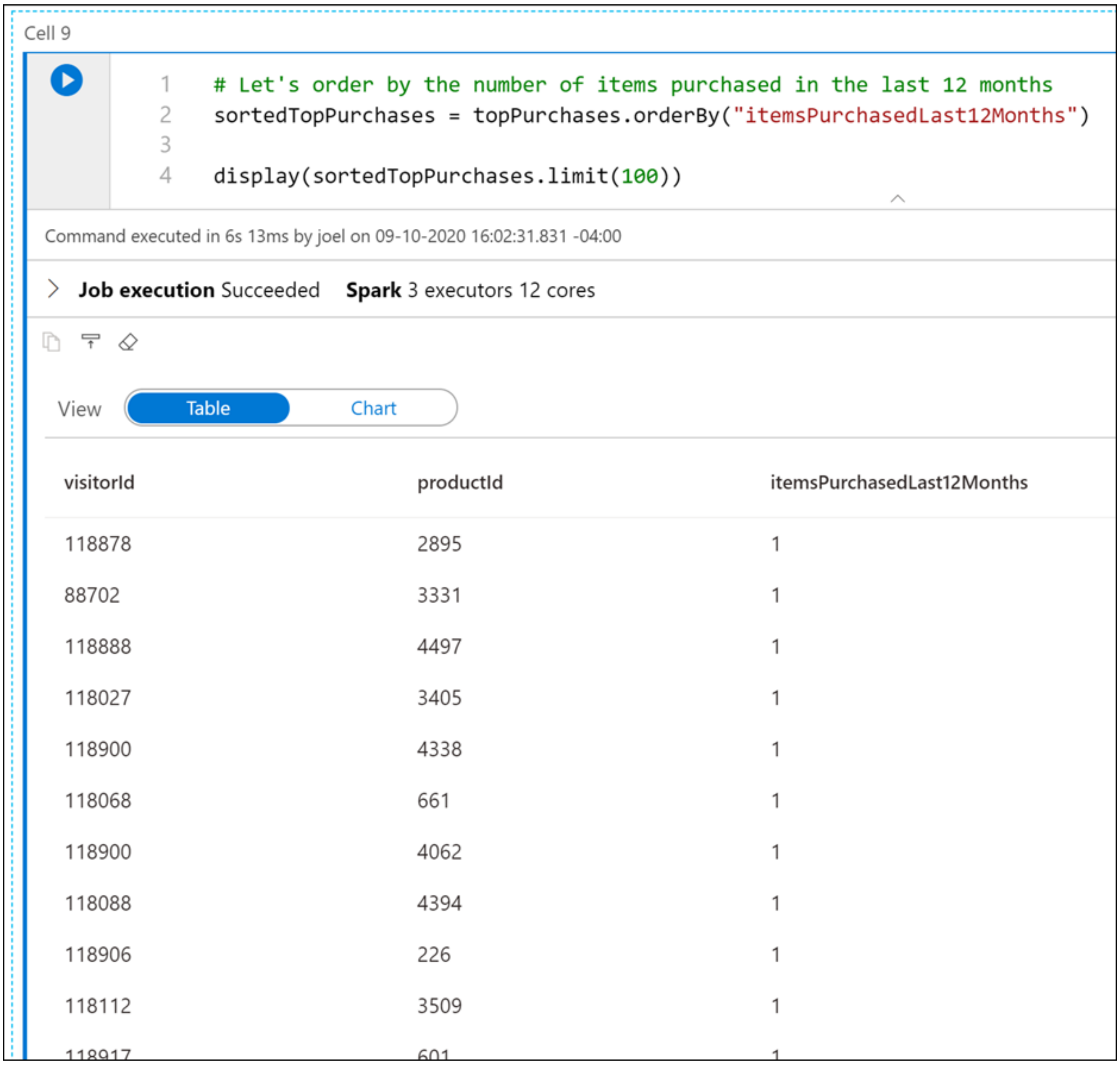
Job execution Succeeded **Spark** 3 executors 12 cores View in monitoring Open Spark UI

```
+---------+---------+------------------------+
|visitorId|productId|itemsPurchasedLast12Months|
+---------+---------+------------------------+
|    80000|     4198|                      92|
|    80000|     2488|                      31|
|    80000|     4136|                      30|
|    80000|     1362|                      18|
|    80000|     3122|                      33|
|    80000|     3270|                       5|
|    80000|       93|                      86|
|    80000|      102|                      42|
|    80000|     4206|                      21|
|    80000|     3538|                      65|
|    80000|     4745|                      38|
|    80000|      291|                      49|
|    80000|      290|                      83|
|    80000|     4074|                      48|
|    80000|     4024|                      35|
|    80000|     2481|                      63|
|    80000|     2859|                      54|
|    80000|     2069|                      93|
|    80000|     1330|                      78|
|    80000|     3794|                      90|
|    80001|     4105|                      11|
|    80001|     2249|                      75|
|    80001|     4684|                       9|
|    80001|     3729|                      56|
```

6. Let's order the rows by the number of items purchased in the last 12 months. Create a new code cell and execute the following code:

```
# Let's order by the number of items purchased in the last 12 months
sortedTopPurchases = topPurchases.orderBy("itemsPurchasedLast12Months")

display(sortedTopPurchases.limit(100))
```



Cell 9

```
# Let's order by the number of items purchased in the last 12 months
sortedTopPurchases = topPurchases.orderBy("itemsPurchasedLast12Months")

display(sortedTopPurchases.limit(100))
```

Command executed in 6s 13ms by joel on 09-10-2020 16:02:31.831 -04:00

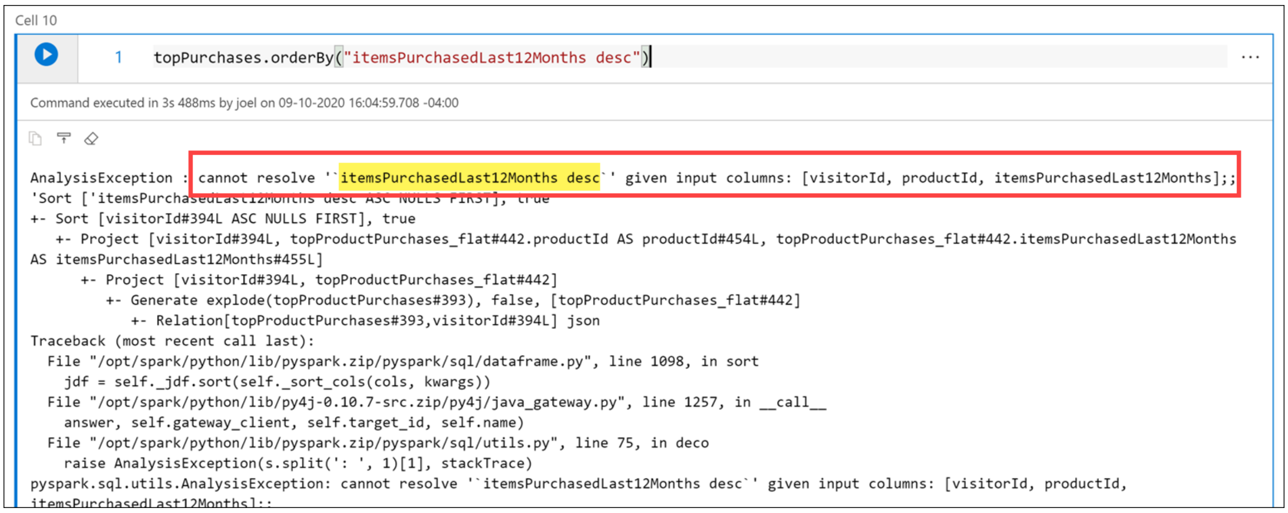
Job execution Succeeded **Spark** 3 executors 12 cores

View Table Chart

visitorId	productId	itemsPurchasedLast12Months
118878	2895	1
88702	3331	1
118888	4497	1
118027	3405	1
118900	4338	1
118068	661	1
118900	4062	1
118088	4394	1
118906	226	1
118112	3509	1
118917	601	1

7. How do we sort in reverse order? One might conclude that we could make a call like this: *topPurchases.orderBy("itemsPurchasedLast12Months desc")*. Try it in a new code cell:

```
topPurchases.orderBy("itemsPurchasedLast12Months desc")
```



Cell 10

```
topPurchases.orderBy("itemsPurchasedLast12Months desc")
```

Command executed in 3s 488ms by joel on 09-10-2020 16:04:59.708 -04:00

```
AnalysisException : cannot resolve '`itemsPurchasedLast12Months desc`' given input columns: [visitorId, productId, itemsPurchasedLast12Months];;
'Sort ['itemsPurchasedLast12Months desc ASC NULLS FIRST], true
+- Sort [visitorId#394L ASC NULLS FIRST], true
   +- Project [visitorId#394L, topProductPurchases_flat#442.productId AS productId#454L, topProductPurchases_flat#442.itemsPurchasedLast12Months
AS itemsPurchasedLast12Months#455L]
      +- Project [visitorId#394L, topProductPurchases_flat#442]
         +- Generate explode(topProductPurchases#393), false, [topProductPurchases_flat#442]
            +- Relation[topProductPurchases#393,visitorId#394L] json
Traceback (most recent call last):
  File "/opt/spark/python/lib/pyspark.zip/pyspark/sql/dataframe.py", line 1098, in sort
    jdf = self._jdf.sort(self._sort_cols(cols, kwargs))
  File "/opt/spark/python/lib/py4j-0.10.7-src.zip/py4j/java_gateway.py", line 1257, in __call__
    answer, self.gateway_client, self.target_id, self.name)
  File "/opt/spark/python/lib/pyspark.zip/pyspark/sql/utils.py", line 75, in deco
    raise AnalysisException(s.split(': ', 1)[1], stackTrace)
pyspark.sql.utils.AnalysisException: cannot resolve '`itemsPurchasedLast12Months desc`' given input columns: [visitorId, productId,
itemsPurchasedLast12Months];;
```

Notice that there is an **AnalysisException`error, because**itemsPurchasedLast12Months desc** does not match up with a column name.

Why does this not work?

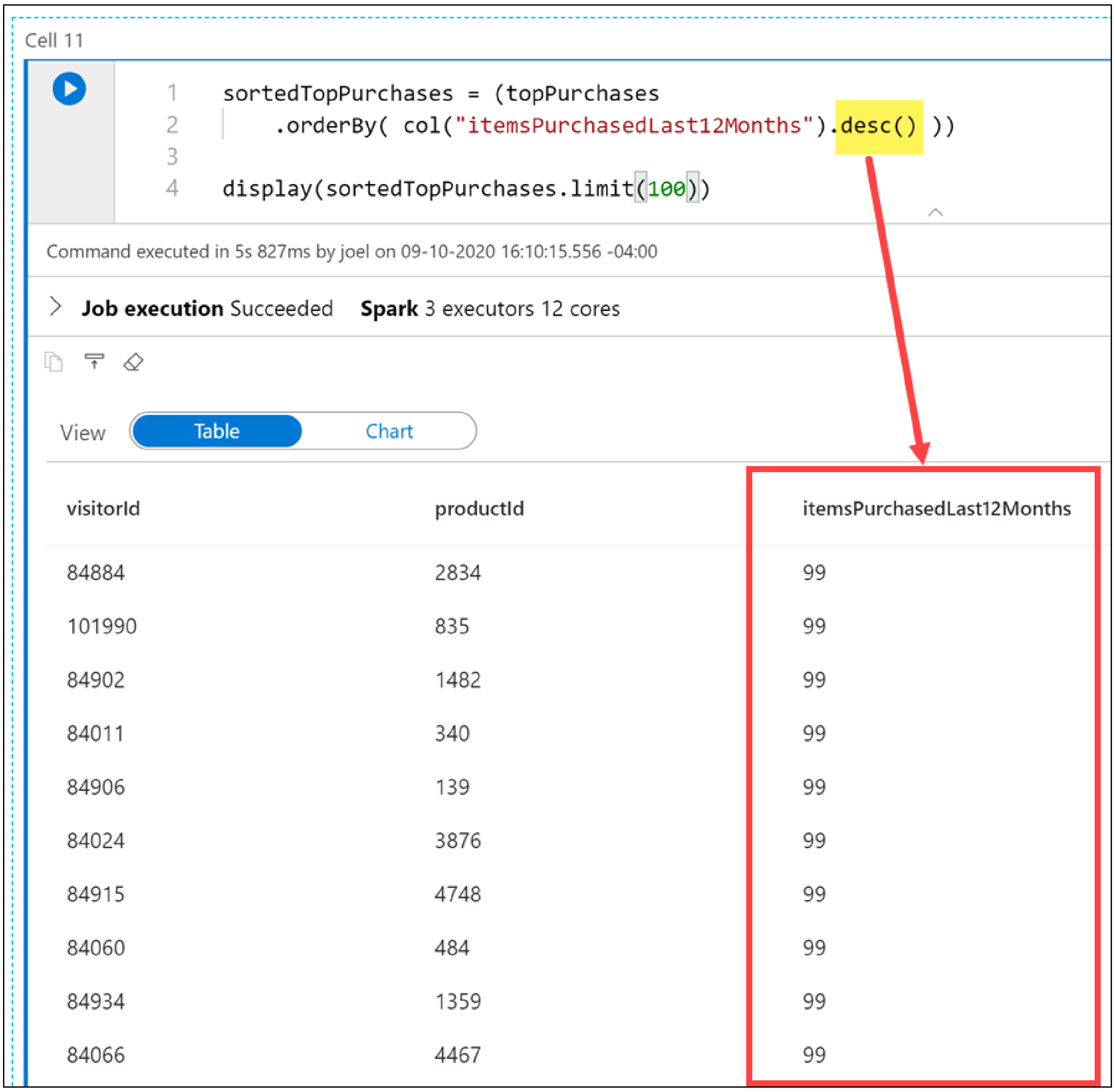
- The **DataFrames** API is built upon an SQL engine.
- There is a lot of familiarity with this API and SQL syntax in general.
- The problem is that **orderBy(..)** expects the name of the column.
- What we specified was an SQL expression in the form of **requests desc**.
- What we need is a way to programmatically express such an expression.
- This leads us to the second variant, **orderBy(Column)**,` and more specifically, the class **Column**.

8. The **Column** class is an object that encompasses more than just the name of the column, but also column-level-transformations, such as sorting in a descending order. Replace the code that failed previously with the following code and run it:

```
sortedTopPurchases = (topPurchases
    .orderBy( col("itemsPurchasedLast12Months").desc() ))

display(sortedTopPurchases.limit(100))
```

Notice that the results are now sorted by the **itemsPurchasedLast12Months** column in descending order, thanks to the **desc()** method on the **col** object.


Cell 11

```
sortedTopPurchases = (topPurchases
    .orderBy( col("itemsPurchasedLast12Months").desc() ))

display(sortedTopPurchases.limit(100))
```

Command executed in 5s 827ms by joel on 09-10-2020 16:10:15.556 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

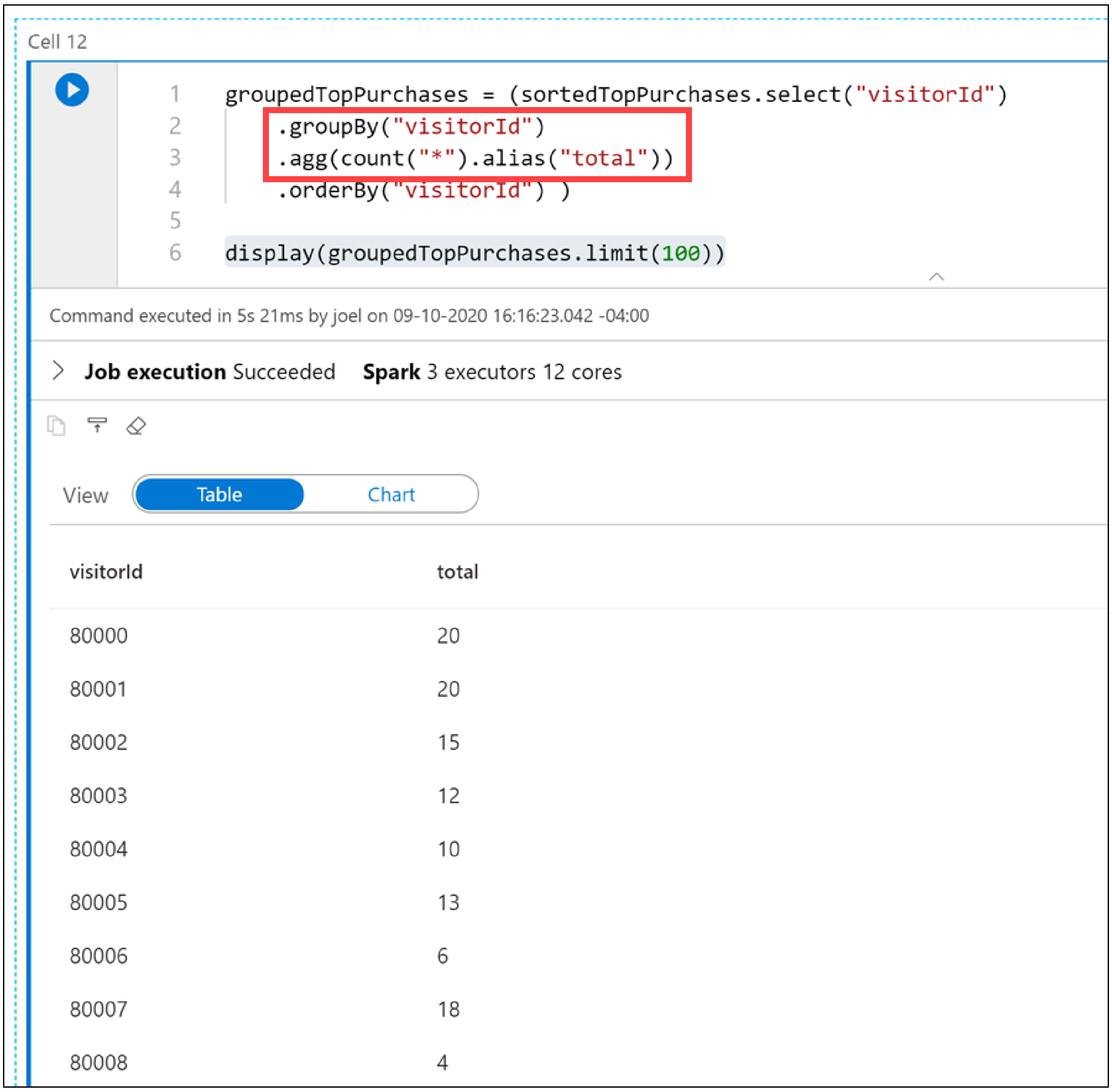
visitorId	productId	itemsPurchasedLast12Months
84884	2834	99
101990	835	99
84902	1482	99
84011	340	99
84906	139	99
84024	3876	99
84915	4748	99
84060	484	99
84934	1359	99
84066	4467	99

9. How many *types* of products did each customer purchase? To figure this out, we need to group by **visitorId** and aggregate on the number of rows per customer. Run the following code in a new code cell:

```
groupedTopPurchases = (sortedTopPurchases.select("visitorId")
    .groupBy("visitorId")
    .agg(count("*").alias("total"))
    .orderBy("visitorId") )

display(groupedTopPurchases.limit(100))
```

Notice how we use the **groupBy** method on the **visitorId** column, and the **agg** method over a count of records to display the total for each customer.


Cell 12

```
groupedTopPurchases = (sortedTopPurchases.select("visitorId")
    .groupBy("visitorId")
    .agg(count("*").alias("total"))
    .orderBy("visitorId") )

display(groupedTopPurchases.limit(100))
```

Command executed in 5s 21ms by joel on 09-10-2020 16:16:23.042 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

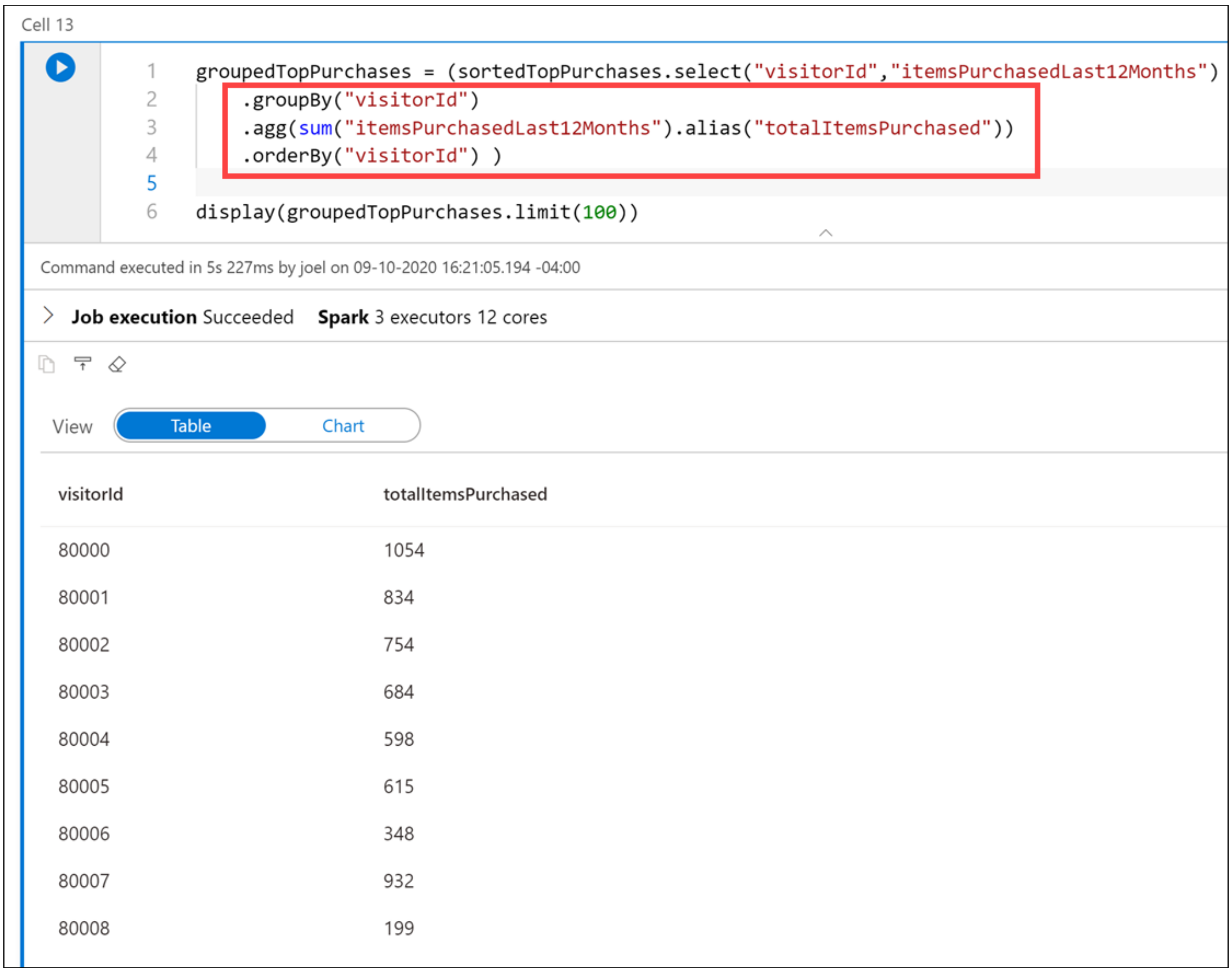
visitorId	total
80000	20
80001	20
80002	15
80003	12
80004	10
80005	13
80006	6
80007	18
80008	4

10. How many *total items* did each customer purchase? To figure this out, we need to group by **visitorId** and aggregate on the sum of **itemsPurchasedLast12Months** values per customer. Run the following code in a new code cell:

```
groupedTopPurchases =
(sortedTopPurchases.select("visitorId","itemsPurchasedLast12Months")
    .groupBy("visitorId")
    .agg(sum("itemsPurchasedLast12Months").alias("totalItemsPurchased"))
    .orderBy("visitorId") )

display(groupedTopPurchases.limit(100))
```

Here we group by **visitorId** once again, but now we use a **sum** over the **itemsPurchasedLast12Months** column in the **agg** method. Notice that we included the **itemsPurchasedLast12Months** column in the **select** statement so we could use it in the **sum**.



```
groupedTopPurchases = (sortedTopPurchases.select("visitorId","itemsPurchasedLast12Months")
    .groupBy("visitorId")
    .agg(sum("itemsPurchasedLast12Months").alias("totalItemsPurchased"))
    .orderBy("visitorId") )

display(groupedTopPurchases.limit(100))
```

Command executed in 5s 227ms by joel on 09-10-2020 16:21:05.194 -04:00

Job execution Succeeded Spark 3 executors 12 cores

visitorId	totalItemsPurchased
80000	1054
80001	834
80002	754
80003	684
80004	598
80005	615
80006	348
80007	932
80008	199

11. Keep the notebook open for the next exercise.

Exercise 4 - Integrating SQL and Spark pools in Azure Synapse Analytics

Tailwind Traders wants to write to the SQL database associated with dedicated SQL pool after performing data engineering tasks in Spark, then reference that SQL database as a source for joining with Spark dataframes that contain data from other files.

You decide to use the Apache Spark to Synapse SQL connector to efficiently transfer data between Spark databases and SQL databases in Azure Synapse.

Transferring data between Spark databases and SQL databases can be done using JDBC. However, given two distributed systems such as Spark pools and SQL pools, JDBC tends to be a bottleneck with serial data transfer.

The Apache Spark pool to Synapse SQL connector is a data source implementation for Apache Spark. It uses the Azure Data Lake Storage Gen2 and PolyBase in dedicated SQL pools to efficiently transfer data between the Spark cluster and the Synapse SQL instance.

Task 1: Update notebook

1. We have been using Python code in these cells up to this point. If we want to use the Apache Spark pool to Synapse SQL connector, one option is to create a temporary view of the data within the dataframe. Run the following in a new code cell to create a view named **top_purchases**:

```
# Create a temporary view for top purchases so we can load from Scala
topPurchases.createOrReplaceTempView("top_purchases")
```

We created a new temporary view from the **topPurchases** dataframe that we created earlier and which contains the flattened JSON user purchases data.

2. We must run code that uses the Apache Spark pool to Synapse SQL connector in Scala. To do this, we add the **%%spark** magic to the cell. Run the following in a new code cell to read from the **top_purchases** view:

```
%%spark
// Make sure the name of the dedcated SQL pool (SQLPool01 below) matches the
name of your SQL pool.
val df = spark.sqlContext.sql("select * from top_purchases")
df.write.synapsesql("SQLPool01.wwi.TopPurchases", Constants.INTERNAL)
```

> **Note**: The cell may take over a minute to execute. If you have run this command before, you will receive an error stating that "There is already and object named.." because the table already exists.

After the cell finishes executing, let's take a look at the list of SQL tables to verify that the table was successfully created for us.

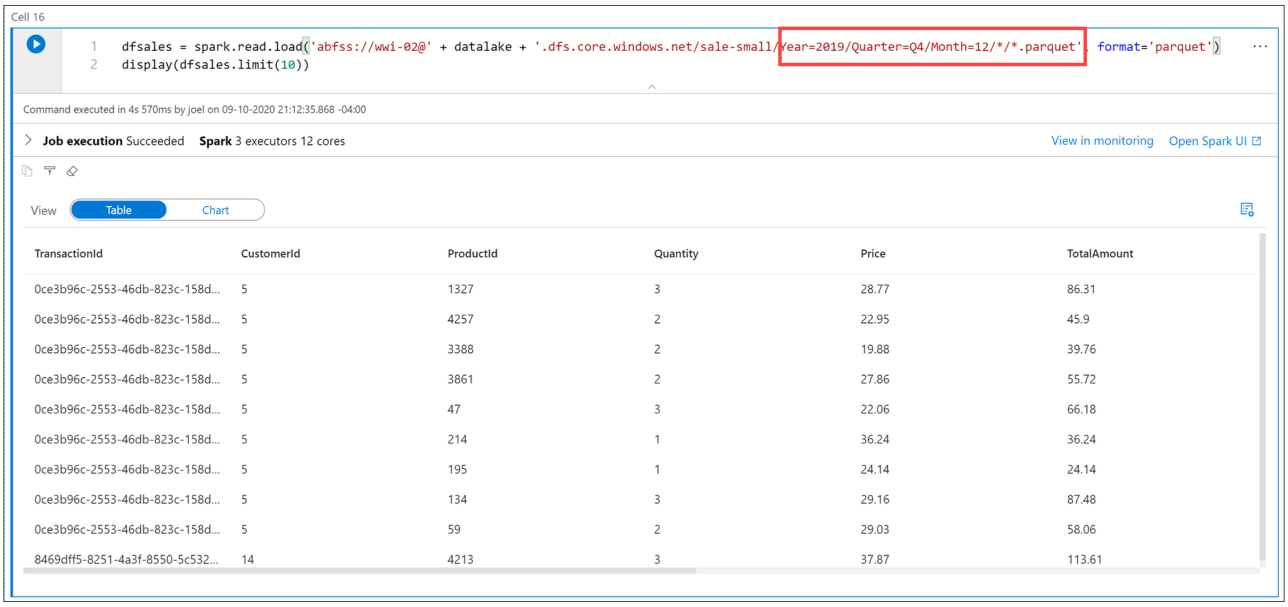
3. **Leave the notebook open**, then navigate to the **Data** hub (if not already selected).

4. Select the **Workspace** tab, In the **ellipses (...)** menu for **Databases**, select **Refresh**. Then expand the **SQLPool01** database and its **Tables** folder, and expand the **wwi.TopPurchases** table and its columns.

The **wwi.TopPurchases** table was automatically created for us, based on the derived schema of the Spark dataframe. The Apache Spark pool to Synapse SQL connector was responsible for creating the table and efficiently loading the data into it.

5. Return to the notebook and run the following code in a new code cell to read sales data from all the Parquet files located in the *sale-small/Year=2019/Quarter=Q4/Month=12/* folder:

```
dfsales = spark.read.load('abfss://wwi-02@' + datalake +
'.dfs.core.windows.net/sale-
small/Year=2019/Quarter=Q4/Month=12/*/*.parquet', format='parquet')
display(dfsales.limit(10))
```



```
dfsales = spark.read.load('abfss://wwi-02@' + datalake + '.dfs.core.windows.net/sale-small/Year=2019/Quarter=Q4/Month=12/*/*.parquet', format='parquet')
display(dfsales.limit(10))
```

Command executed in 4s 570ms by joel on 09-10-2020 21:12:35.868 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

TransactionId	CustomerId	ProductId	Quantity	Price	TotalAmount
0ce3b96c-2553-46db-823c-158d...	5	1327	3	28.77	86.31
0ce3b96c-2553-46db-823c-158d...	5	4257	2	22.95	45.9
0ce3b96c-2553-46db-823c-158d...	5	3388	2	19.88	39.76
0ce3b96c-2553-46db-823c-158d...	5	3861	2	27.86	55.72
0ce3b96c-2553-46db-823c-158d...	5	47	3	22.06	66.18
0ce3b96c-2553-46db-823c-158d...	5	214	1	36.24	36.24
0ce3b96c-2553-46db-823c-158d...	5	195	1	24.14	24.14
0ce3b96c-2553-46db-823c-158d...	5	134	3	29.16	87.48
0ce3b96c-2553-46db-823c-158d...	5	59	2	29.03	58.06
8469dff5-8251-4a3f-8550-5c532...	14	4213	3	37.87	113.61

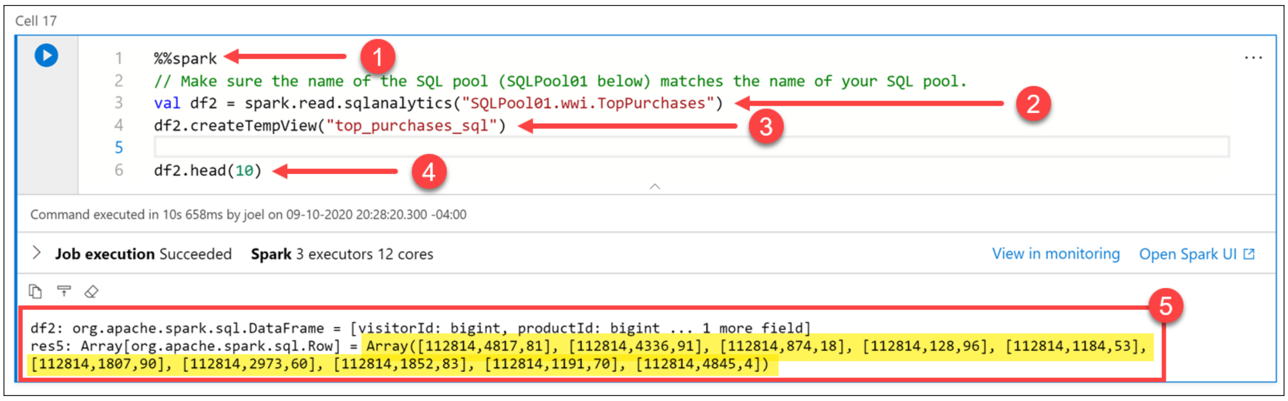
Compare the file path in the cell above to the file path in the first cell. Here we are using a relative path to load **all December 2019 sales** data from the Parquet files located in **sale-small**, vs. just December 31, 2019 sales data.

Next, let's load the **TopSales** data from the SQL table we created earlier into a new Spark dataframe, then join it with this new **dfsales** dataframe. To do this, we must once again use the **%%spark** magic on a new cell since we'll use the Apache Spark pool to Synapse SQL connector to retrieve data from the SQL database. Then we need to add the dataframe contents to a new temporary view so we can access the data from Python.

6. Run the following code in a new cell to read from the **TopSales** SQL table and save it to a temporary view:

```
%%spark
// Make sure the name of the SQL pool (SQLPool01 below) matches the name of your SQL pool.
val df2 = spark.read.synapsesql("SQLPool01.wwi.TopPurchases")
df2.createTempView("top_purchases_sql")

df2.head(10)
```



```
%%spark
// Make sure the name of the SQL pool (SQLPool01 below) matches the name of your SQL pool.
val df2 = spark.read.sqlanalytics("SQLPool01.wwi.TopPurchases")
df2.createTempView("top_purchases_sql")

df2.head(10)
```

Command executed in 10s 658ms by joel on 09-10-2020 20:28:20.300 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

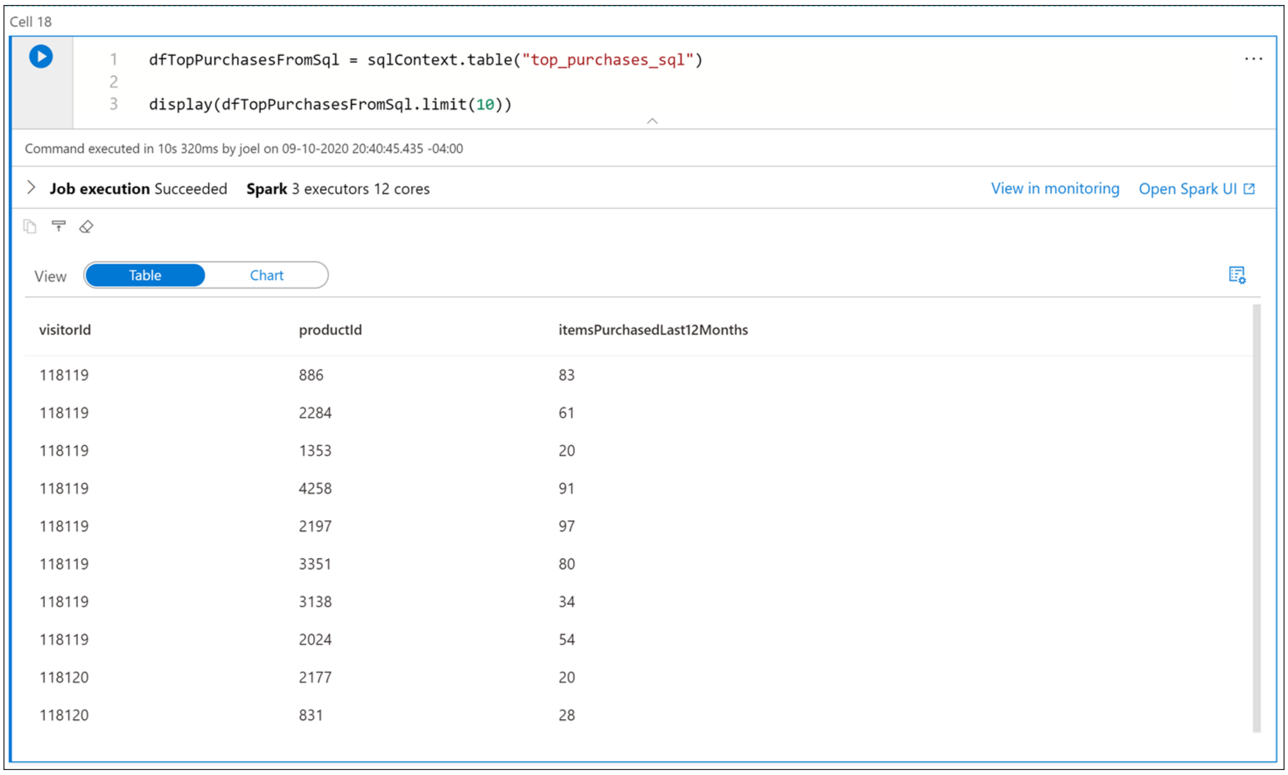
```
df2: org.apache.spark.sql.DataFrame = [visitorId: bigint, productId: bigint ... 1 more field]
res5: Array[org.apache.spark.sql.Row] = Array([112814,4817,81], [112814,4336,91], [112814,874,18], [112814,128,96], [112814,1184,53],
[112814,1807,90], [112814,2973,60], [112814,1852,83], [112814,1191,70], [112814,4845,4])
```

The cell's language is set to Scala by using the **%%spark** magic at the top of the cell. We declared a new variable named **df2** as a new DataFrame created by the **spark.read.synapsesql** method, which reads from the **TopPurchases** table in the SQL database. Then we populated a new temporary view named **top_purchases_sql**. Finally, we showed the first 10 records with the **df2.head(10))** line. The cell output displays the dataframe values.

7. Run the following code in a new code cell to create a new dataframe in Python from the **top_purchases_sql** temporary view, then display the first 10 results:

```
dfTopPurchasesFromSql = sqlContext.table("top_purchases_sql")

display(dfTopPurchasesFromSql.limit(10))
```



Cell 18

```
dfTopPurchasesFromSql = sqlContext.table("top_purchases_sql")

display(dfTopPurchasesFromSql.limit(10))
```

Command executed in 10s 320ms by joel on 09-10-2020 20:40:45.435 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

visitorId	productId	itemsPurchasedLast12Months
118119	886	83
118119	2284	61
118119	1353	20
118119	4258	91
118119	2197	97
118119	3351	80
118119	3138	34
118119	2024	54
118120	2177	20
118120	831	28

8. Run the following code in a new code cell to join the data from the sales Parquet files and the **TopPurchases** SQL database:

```
inner_join = dfsales.join(dfTopPurchasesFromSql,
    (dfsales.CustomerId == dfTopPurchasesFromSql.visitorId) &
(dfsales.ProductId == dfTopPurchasesFromSql.productId))

inner_join_agg =
(inner_join.select("CustomerId","TotalAmount","Quantity","itemsPurchasedLast
12Months","top_purchases_sql.productId")
    .groupBy(["CustomerId","top_purchases_sql.productId"])
    .agg(
        sum("TotalAmount").alias("TotalAmountDecember"),
        sum("Quantity").alias("TotalQuantityDecember"),
sum("itemsPurchasedLast12Months").alias("TotalItemsPurchasedLast12Months"))
    .orderBy("CustomerId") )

display(inner_join_agg.limit(100))
```

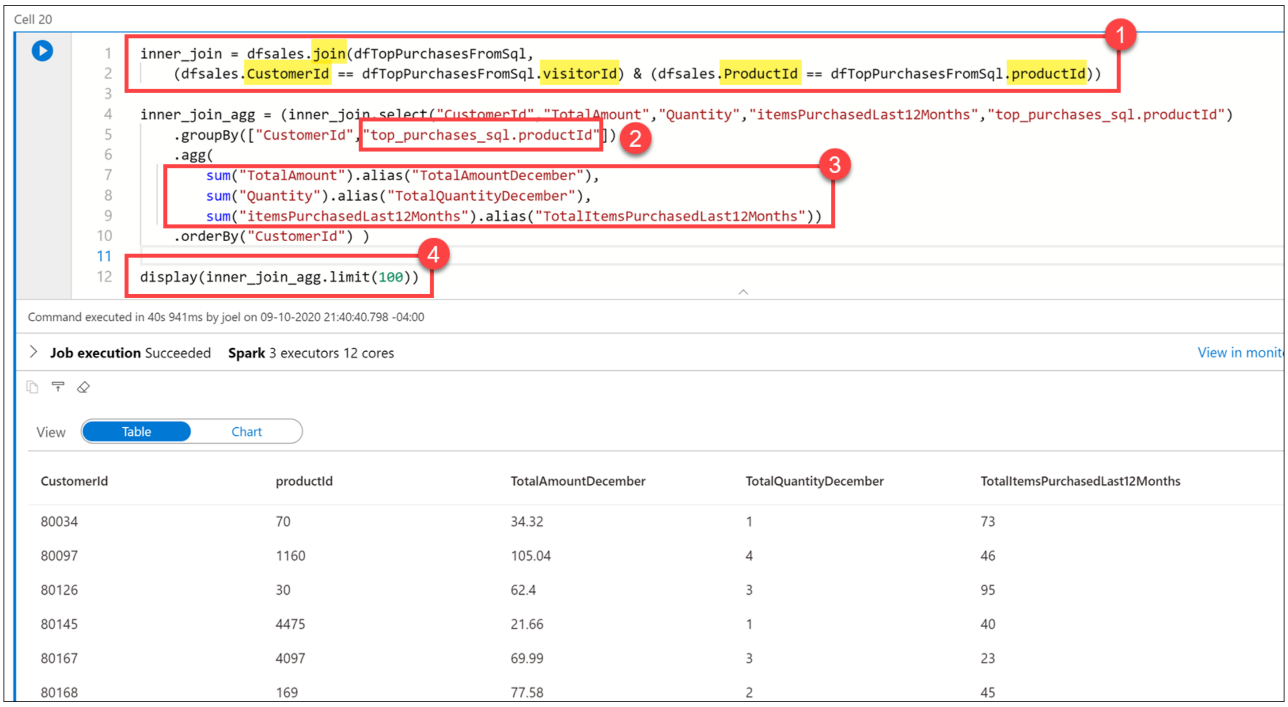
In the query, we joined the **dfsales** and **dfTopPurchasesFromSql** dataframes, matching on **CustomerId** and **ProductId**. This join combined the **TopPurchases** SQL table data with the December 2019 sales Parquet data.

We grouped by the **CustomerId** and **ProductId** fields. Since the **ProductId** field name is ambiguous (it exists in both dataframes), we had to fully-qualify the **ProductId** name to refer to the one in the **TopPurchases** dataframe.

Then we created an aggregate that summed the total amount spent on each product in December, the total number of product items in December, and the total product items purchased in the last 12 months.

Finally, we displayed the joined and aggregated data in a table view.

> **Note**: You can click the column headers in the Table view to sort the result set.



```
inner_join = dfsales.join(dfTopPurchasesFromSql,
    (dfsales.CustomerId == dfTopPurchasesFromSql.visitorId) & (dfsales.ProductId == dfTopPurchasesFromSql.productId))

inner_join_agg = (inner_join.select("CustomerId","TotalAmount","Quantity","itemsPurchasedLast12Months","top_purchases_sql.productId")
    .groupBy(["CustomerId","top_purchases_sql.productId"])
    .agg(
        sum("TotalAmount").alias("TotalAmountDecember"),
        sum("Quantity").alias("TotalQuantityDecember"),
        sum("itemsPurchasedLast12Months").alias("TotalItemsPurchasedLast12Months"))
    .orderBy("CustomerId") )

display(inner_join_agg.limit(100))
```

Command executed in 40s 941ms by joel on 09-10-2020 21:40:40.798 -04:00

Job execution Succeeded **Spark** 3 executors 12 cores

CustomerId	productId	TotalAmountDecember	TotalQuantityDecember	TotalItemsPurchasedLast12Months
80034	70	34.32	1	73
80097	1160	105.04	4	46
80126	30	62.4	3	95
80145	4475	21.66	1	40
80167	4097	69.99	3	23
80168	169	77.58	2	45

9. At the top right of the notebook, use the **Stop Session** button to stop the notebook session.

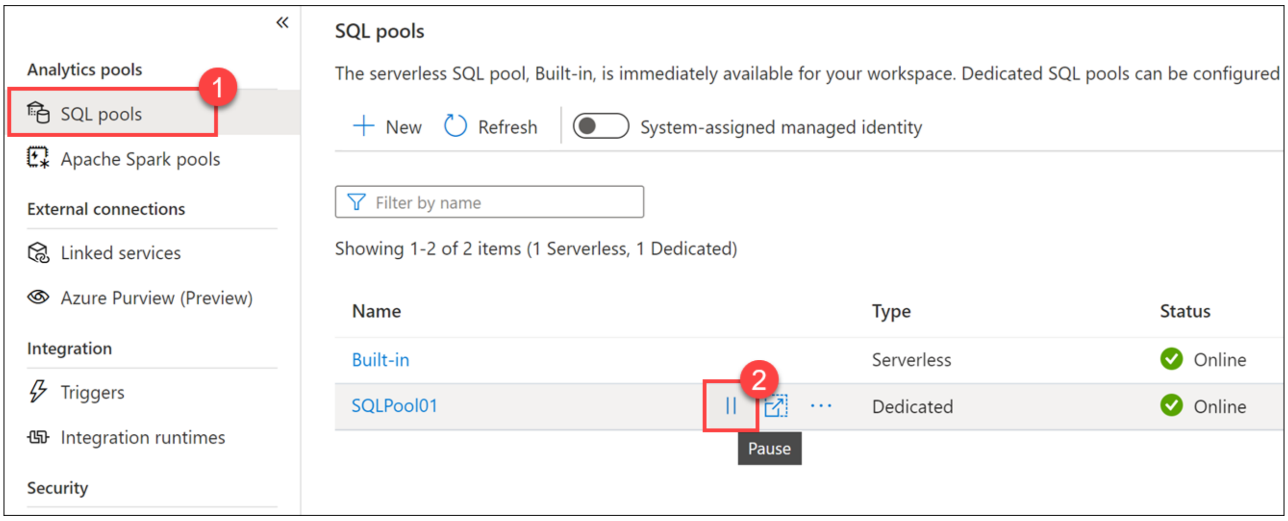
10. Publish the notebook if you want to review it again later. Then close it.

Important: Pause your SQL pool

Complete these steps to free up resources you no longer need.

1. In Synapse Studio, select the **Manage** hub.

2. Select **SQL pools** in the left-hand menu. Hover over the **SQLPool01** dedicated SQL pool and select **||**.



SQL pools

The serverless SQL pool, Built-in, is immediately available for your workspace. Dedicated SQL pools can be configured

Showing 1-2 of 2 items (1 Serverless, 1 Dedicated)

Name	Type	Status
Built-in	Serverless	Online
SQLPool01	Dedicated	Online

3. When prompted, select **Pause**.