Create a medallion architecture in a Microsoft Fabric lakehouse

In this exercise you will build out a medallion architecture in a Fabric lakehouse using notebooks. You will create a workspace, create a lakehouse, upload data to the bronze layer, transform the data and load it to the silver Delta table, transform the data further and load it to the gold Delta tables, and then explore the semantic model and create relationships.

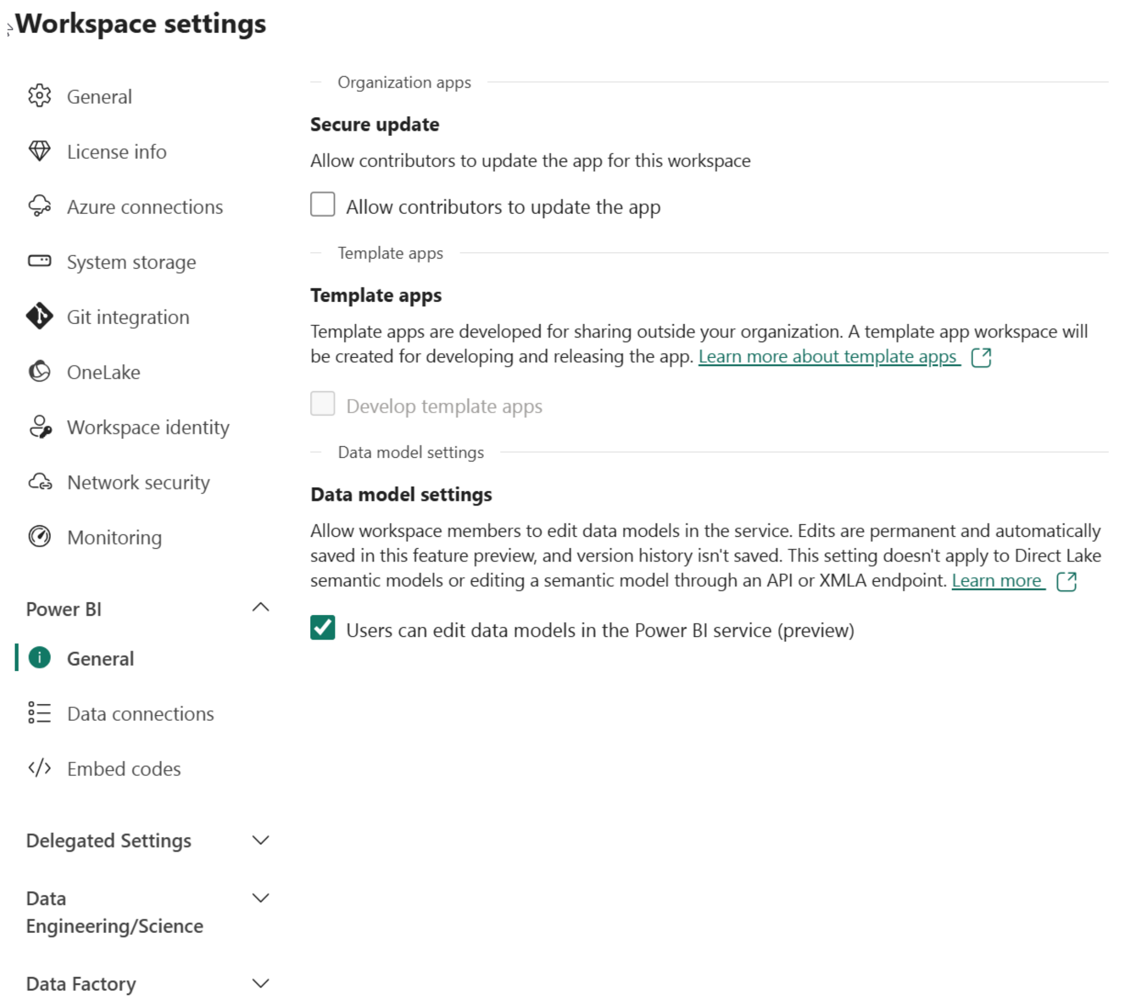
This exercise should take approximately **45** minutes to complete

Note: You need a Microsoft Fabric trial to complete this exercise.

Create a workspace

Before working with data in Fabric, create a workspace with the Fabric trial enabled.

1. Navigate to the Microsoft Fabric home page at `https://app.fabric.microsoft.com/home?experience=fabric` in a browser, and sign in with your Fabric credentials.
2. In the menu bar on the left, select **Workspaces** (the icon looks similar to 🗇).
3. Create a new workspace with a name of your choice, selecting a licensing mode that includes Fabric capacity (*Trial*, *Premium*, or *Fabric*).
4. When your new workspace opens, it should be empty.
5. Navigate to the workspace settings and enable the **Data model editing** preview feature. This will enable you to create relationships between tables in your lakehouse using a Power BI semantic model.


Workspace settings

General
License info
Azure connections
System storage
Git integration
OneLake
Workspace identity
Network security
Monitoring
Power BI
General
Data connections
Embed codes
Delegated Settings
Data Engineering/Science
Data Factory

Organization apps

Secure update

Allow contributors to update the app for this workspace

☐ Allow contributors to update the app

Template apps

Template apps

Template apps are developed for sharing outside your organization. A template app workspace will be created for developing and releasing the app. Learn more about template apps

☐ Develop template apps

Data model settings

Data model settings

Allow workspace members to edit data models in the service. Edits are permanent and automatically saved in this feature preview, and version history isn't saved. This setting doesn't apply to Direct Lake semantic models or editing a semantic model through an API or XMLA endpoint. Learn more

☑ Users can edit data models in the Power BI service (preview)

Note: You may need to refresh the browser tab after enabling the preview feature.

Create a lakehouse and upload data to bronze layer

Now that you have a workspace, it’s time to create a data lakehouse for the data you’re going to analyze.

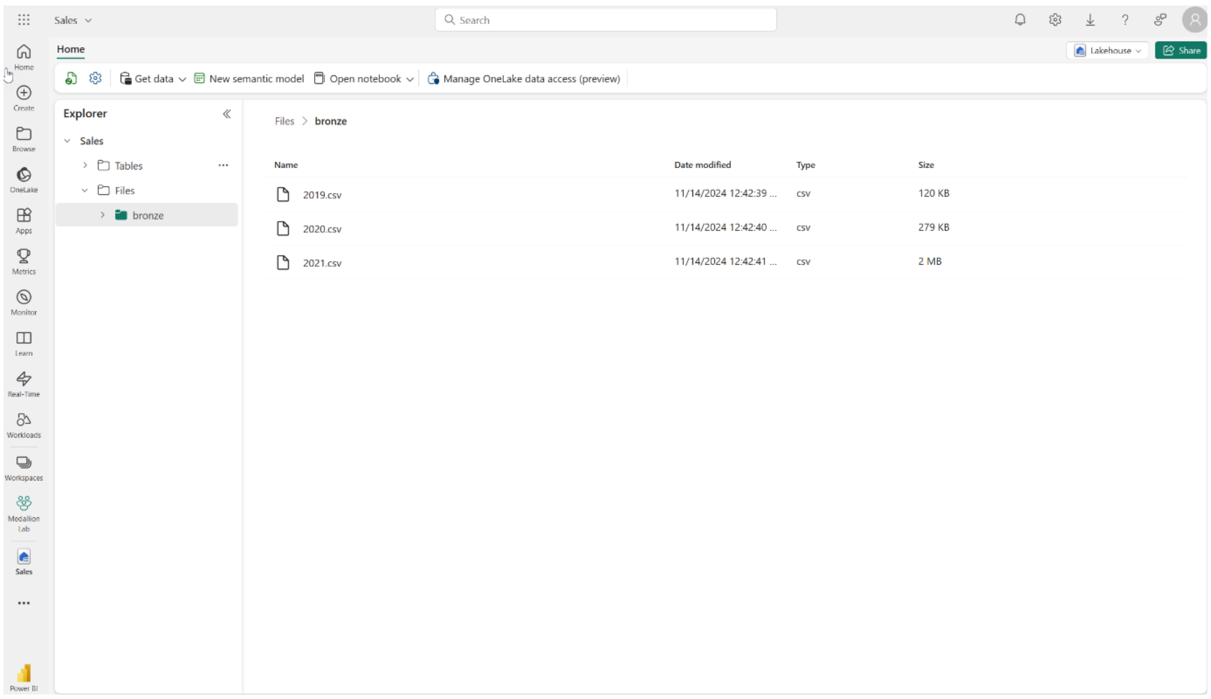
1. In the workspace you just created, create a new **Lakehouse** named **Sales** by clicking the **+ New item** button.

   After a minute or so, a new empty lakehouse will be created. Next, you’ll ingest some data into the data lakehouse for analysis. There are multiple ways to do this, but in this exercise you’ll simply download a text file to your local computer (or lab VM if applicable) and then upload it to your lakehouse.

2. Download the data file for this exercise from `https://github.com/MicrosoftLearning/dp-data/blob/main/orders.zip`. Extract the files and save them with their original names on your local

computer (or lab VM if applicable). There should be 3 files containing sales data for 3 years: 2019.csv, 2020.csv, and 2021.csv.

3. Return to the web browser tab containing your lakehouse, and in the **...** menu for the **Files** folder in the **Explorer** pane, select **New subfolder** and create a folder named **bronze**.
4. In the **...** menu for the **bronze** folder, select **Upload** and **Upload files**, and then upload the 3 files (2019.csv, 2020.csv, and 2021.csv) from your local computer (or lab VM if applicable) to the lakehouse. Use the shift key to upload all 3 files at once.
5. After the files have been uploaded, select the **bronze** folder; and verify that the files have been uploaded, as shown here:



Name	Date modified	Type	Size
2019.csv	11/14/2024 12:42:39 ...	csv	120 KB
2020.csv	11/14/2024 12:42:40 ...	csv	279 KB
2021.csv	11/14/2024 12:42:41 ...	csv	2 MB

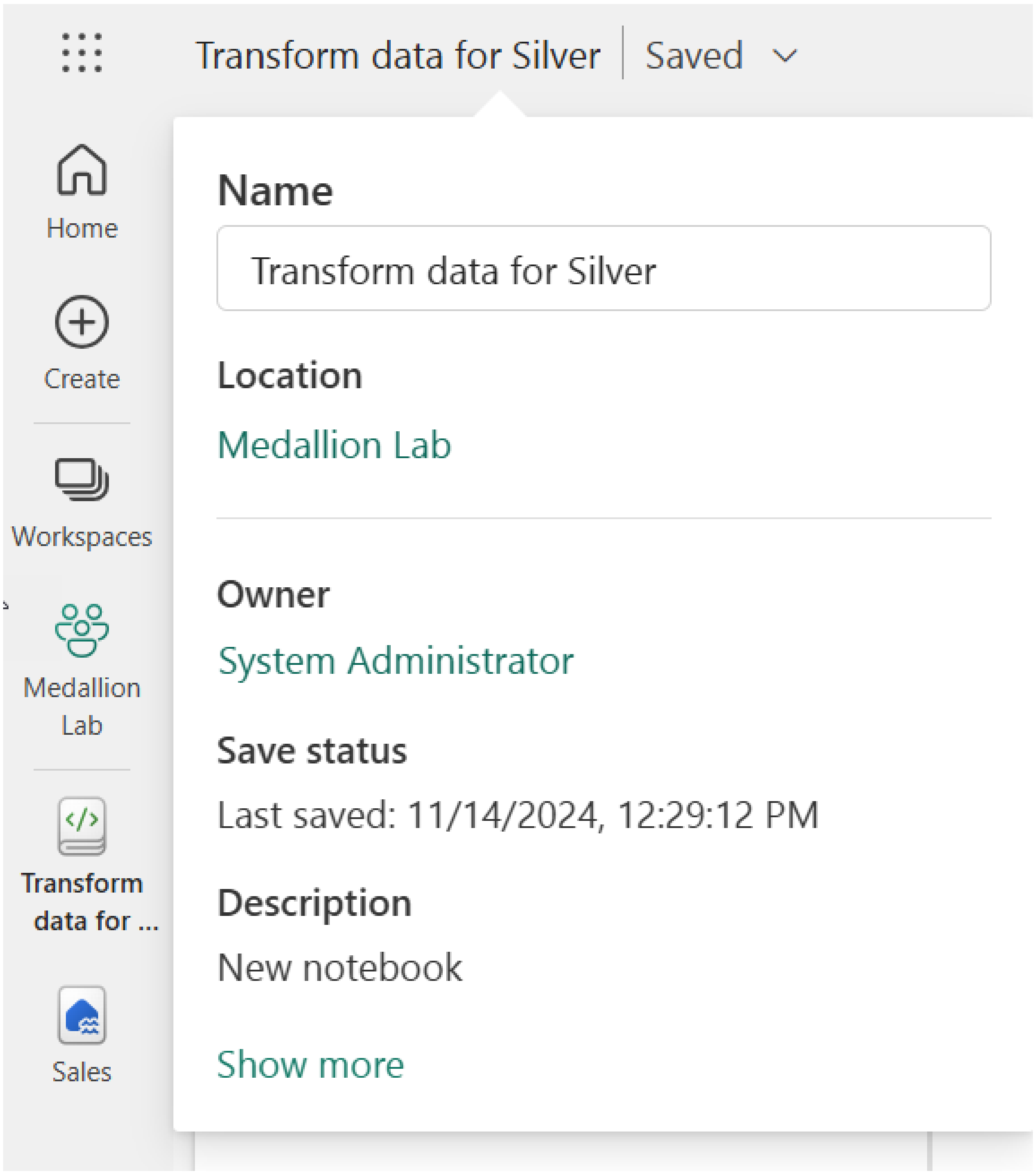
Transform data and load to silver Delta table

Now that you have some data in the bronze layer of your lakehouse, you can use a notebook to transform the data and load it to a delta table in the silver layer.

1. On the **Home** page while viewing the contents of the **bronze** folder in your data lake, in the **Open notebook** menu, select **New notebook**.

   After a few seconds, a new notebook containing a single *cell* will open. Notebooks are made up of one or more cells that can contain *code* or *markdown* (formatted text).

2. When the notebook opens, rename it to **Transform data for Silver** by selecting the **Notebook xxxx** text at the top left of the notebook and entering the new name.


Transform data for Silver | Saved

Name
Transform data for Silver

Location
Medallion Lab

Owner
System Administrator

Save status
Last saved: 11/14/2024, 12:29:12 PM

Description
New notebook

Show more

3. Select the existing cell in the notebook, which contains some simple commented-out code. Highlight and delete these two lines - you will not need this code.

   Note: Notebooks enable you to run code in a variety of languages, including Python, Scala, and SQL. In this exercise, you'll use PySpark and SQL. You can also add markdown cells to provide formatted text and images to document your code.

4. **Paste** the following code into the cell:

codeCopy

```
from pyspark.sql.types import *

# Create the schema for the table
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())
    ])

# Import all files from bronze folder of lakehouse
df = spark.read.format("csv").option("header",
"true").schema(orderSchema).load("Files/bronze/*.csv")

# Display the first 10 rows of the dataframe to preview your data
display(df.head(10))
```

5. Use the **▷ (*Run cell*)** button on the left of the cell to run the code.

   Note: Since this is the first time you've run any Spark code in this notebook, a Spark session must be started. This means that the first run can take a minute or so to complete. Subsequent runs will be quicker.

6. When the cell command has completed, **review the output** below the cell, which should look similar to this:

Index	SalesOrderNumber	SalesOrderLineNumber	OrderDate	CustomerN
1	SO49172	1	2021-01-01	Brian Howar
2	SO49173	1	2021-01-01	Linda Alvare
…	…	…	…	…

7. The code you ran loaded the data from the CSV files in the **bronze** folder into a Spark dataframe, and then displayed the first few rows of the dataframe.

8. **Note**: You can clear, hide, and auto-resize the contents of the cell output by selecting the **...** menu at the top left of the output pane.

9. Now you'll **add columns for data validation and cleanup**, using a PySpark dataframe to add columns and update the values of some of the existing columns. Use the + button to **add a new code block** and add the following code to the cell:

   codeCopy

```
 from pyspark.sql.functions import when, lit, col, current_timestamp, 
input_file_name

 # Add columns IsFlagged, CreatedTS and ModifiedTS
 df = df.withColumn("FileName", input_file_name()) \
     .withColumn("IsFlagged", when(col("OrderDate") < '2019-08-
01',True).otherwise(False)) \
     .withColumn("CreatedTS", current_timestamp()).withColumn("ModifiedTS", 
current_timestamp())

 # Update CustomerName to "Unknown" if CustomerName null or empty
 df = df.withColumn("CustomerName", when((col("CustomerName").isNull() | 
(col("CustomerName")=="")),lit("Unknown")).otherwise(col("CustomerName")))
```

   The first line of the code imports the necessary functions from PySpark. You're then adding new columns to the dataframe so you can track the source file name, whether the order was flagged as being a before the fiscal year of interest, and when the row was created and modified.

   Finally, you're updating the CustomerName column to "Unknown" if it's null or empty.

10. Run the cell to execute the code using the **▷ (*Run cell*)** button.
11. Next, you'll define the schema for the **sales_silver** table in the sales database using Delta Lake format. Create a new code block and add the following code to the cell:

   codeCopy

```
 # Define the schema for the sales_silver table

 from pyspark.sql.types import *
 from delta.tables import *

 DeltaTable.createIfNotExists(spark) \
     .tableName("sales.sales_silver") \
     .addColumn("SalesOrderNumber", StringType()) \
     .addColumn("SalesOrderLineNumber", IntegerType()) \
     .addColumn("OrderDate", DateType()) \
     .addColumn("CustomerName", StringType()) \
     .addColumn("Email", StringType()) \
     .addColumn("Item", StringType()) \
        .addColumn("Quantity", IntegerType()) \
        .addColumn("UnitPrice", FloatType()) \
        .addColumn("Tax", FloatType()) \
        .addColumn("FileName", StringType()) \
        .addColumn("IsFlagged", BooleanType()) \
        .addColumn("CreatedTS", DateType()) \
        .addColumn("ModifiedTS", DateType()) \
        .execute()
```

12. Run the cell to execute the code using the **▷ (*Run cell*)** button.
13. Select the **...** in the Tables section of the lakehouse explorer pane and select **Refresh**. You should now see the new **sales_silver** table listed. The ▲ (triangle icon) indicates that it's a Delta table.

    Note: If you don't see the new table, wait a few seconds and then select **Refresh** again, or refresh the entire browser tab.

14. Now you're going to perform an **upsert operation** on a Delta table, updating existing records based on specific conditions and inserting new records when no match is found. Add a new code block and paste the following code:

    codeCopy

```
# Update existing records and insert new ones based on a condition defined
by the columns SalesOrderNumber, OrderDate, CustomerName, and Item.

from delta.tables import *

deltaTable = DeltaTable.forPath(spark, 'Tables/sales_silver')

dfUpdates = df

deltaTable.alias('silver') \
  .merge(
    dfUpdates.alias('updates'),
    'silver.SalesOrderNumber = updates.SalesOrderNumber and
silver.OrderDate = updates.OrderDate and silver.CustomerName =
updates.CustomerName and silver.Item = updates.Item'
  ) \
   .whenMatchedUpdate(set =
    {

    }
  ) \
 .whenNotMatchedInsert(values =
    {
      "SalesOrderNumber": "updates.SalesOrderNumber",
      "SalesOrderLineNumber": "updates.SalesOrderLineNumber",
      "OrderDate": "updates.OrderDate",
      "CustomerName": "updates.CustomerName",
      "Email": "updates.Email",
      "Item": "updates.Item",
      "Quantity": "updates.Quantity",
      "UnitPrice": "updates.UnitPrice",
        "Tax": "updates.Tax",
        "FileName": "updates.FileName",
        "IsFlagged": "updates.IsFlagged",
        "CreatedTS": "updates.CreatedTS",
        "ModifiedTS": "updates.ModifiedTS"
      }
    ) \
    .execute()
```

15. Run the cell to execute the code using the **▷ (*Run cell*)** button.

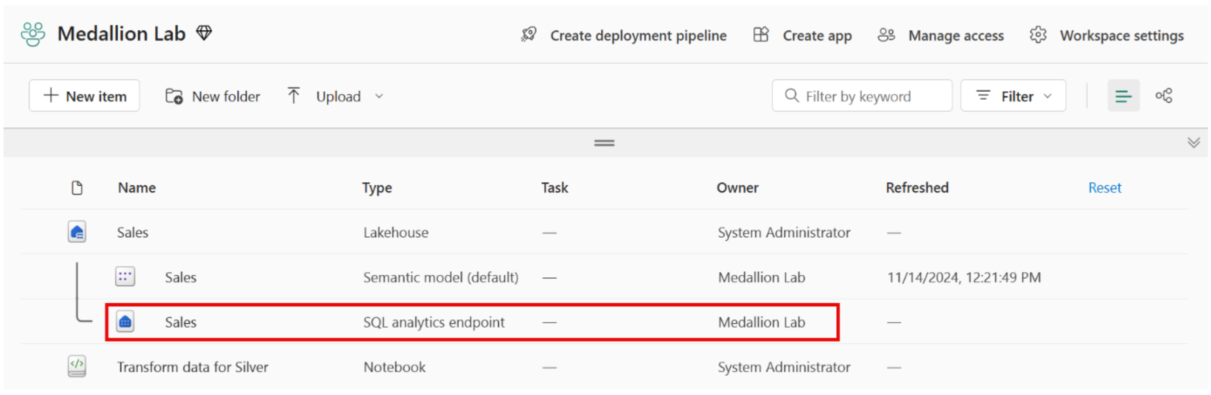
    This operation is important because it enables you to update existing records in the table based on the values of specific columns, and insert new records when no match is found. This is a common requirement when you're loading data from a source system that may contain updates to existing and new records.

You now have data in your silver delta table that is ready for further transformation and modeling.

Explore data in the silver layer using the SQL endpoint

Now that you have data in your silver layer, you can use the SQL analytics endpoint to explore the data and perform some basic analysis. This is useful if you're familiar with SQL and want to do some basic exploration of your data. In this exercise we're using the SQL endpoint view in Fabric, but you can use other tools like SQL Server Management Studio (SSMS) and Azure Data Explorer.

1. Navigate back to your workspace and notice that you now have several items listed. Select the **Sales SQL analytics endpoint** to open your lakehouse in the SQL analytics endpoint view.



Name	Type	Task	Owner	Refreshed
Sales	Lakehouse	—	System Administrator	—
Sales	Semantic model (default)	—	Medallion Lab	11/14/2024, 12:21:49 PM
Sales	SQL analytics endpoint	—	Medallion Lab	—
Transform data for Silver	Notebook	—	System Administrator	—

2. Select **New SQL query** from the ribbon, which will open a SQL query editor. Note that you can rename your query using the **...** menu item next to the existing query name in the lakehouse explorer pane.

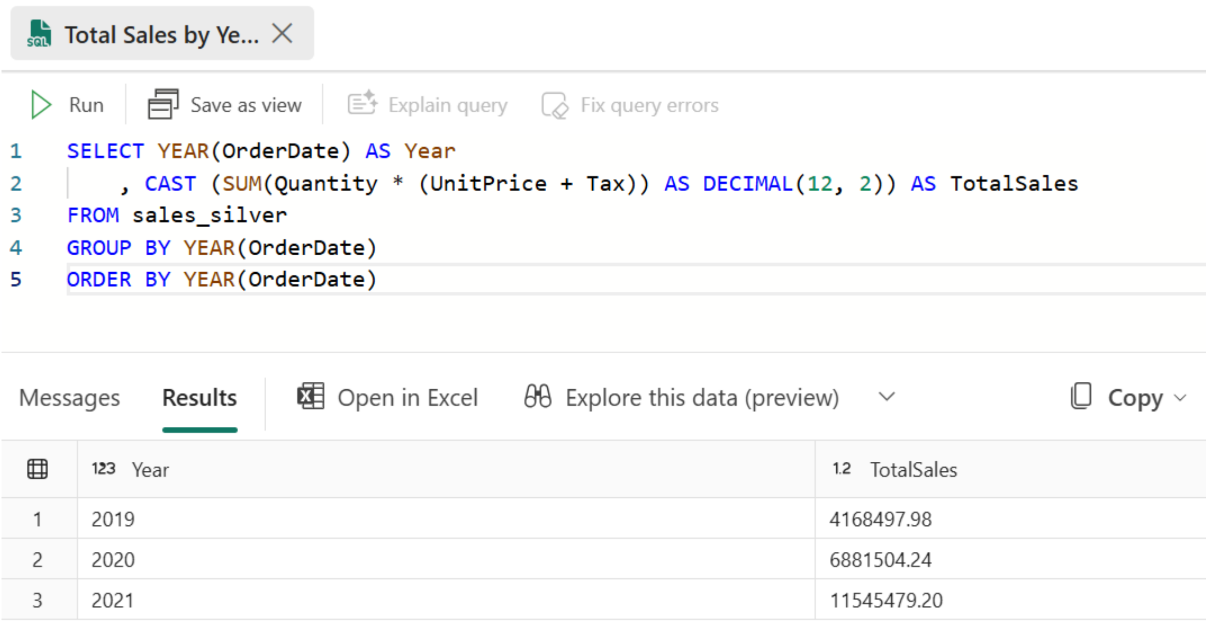
   Next, you'll run two sql queries to explore the data.

3. Paste the following query into the query editor and select **Run**:

   sqlCopy

```sql
 SELECT YEAR(OrderDate) AS Year
     , CAST (SUM(Quantity * (UnitPrice + Tax)) AS DECIMAL(12, 2)) AS
TotalSales
 FROM sales_silver
 GROUP BY YEAR(OrderDate)
 ORDER BY YEAR(OrderDate)
```

   This query calculates the total sales for each year in the sales_silver table. Your results should look like this:



```sql
SELECT YEAR(OrderDate) AS Year
    , CAST (SUM(Quantity * (UnitPrice + Tax)) AS DECIMAL(12, 2)) AS TotalSales
FROM sales_silver
GROUP BY YEAR(OrderDate)
ORDER BY YEAR(OrderDate)
```

	Year	TotalSales
1	2019	4168497.98
2	2020	6881504.24
3	2021	11545479.20

4. Next you'll review which customers are purchasing the most (in terms of quantity). Paste the following query into the query editor and select **Run**:

   sqlCopy

```sql
 SELECT TOP 10 CustomerName, SUM(Quantity) AS TotalQuantity
 FROM sales_silver
 GROUP BY CustomerName
```

```
ORDER BY TotalQuantity DESC
```

This query calculates the total quantity of items purchased by each customer in the sales_silver table, and then returns the top 10 customers in terms of quantity.

Data exploration at the silver layer is useful for basic analysis, but you'll need to transform the data further and model it into a star schema to enable more advanced analysis and reporting. You'll do that in the next section.

Transform data for gold layer

You have successfully taken data from your bronze layer, transformed it, and loaded it into a silver Delta table. Now you'll use a new notebook to transform the data further, model it into a star schema, and load it into gold Delta tables.

You could have done all of this in a single notebook, but for this exercise you're using separate notebooks to demonstrate the process of transforming data from bronze to silver and then from silver to gold. This can help with debugging, troubleshooting, and reuse.

1. Return to the workspace home page and create a new notebook called **Transform data for Gold**.
2. In the lakehouse explorer pane, add your **Sales** lakehouse by selecting **Add** and then selecting the **Sales** lakehouse you created earlier. In the **Add Lakehouse** window, select **Existing Lakehouse without Schema**. You should see the **sales_silver** table listed in the **Tables** section of the explorer pane.
3. In the existing code block, remove the commented text and **add the following code** to load data to your dataframe and start building your star schema, then run it:

   codeCopy

```
# Load data to the dataframe as a starting point to create the gold layer
df = spark.read.table("Sales.sales_silver")
```

4. **Add a new code block** and paste the following code to create your date dimension table and run it:

   codeCopy

```
 from pyspark.sql.types import *
 from delta.tables import*

 # Define the schema for the dimdate_gold table
 DeltaTable.createIfNotExists(spark) \
     .tableName("sales.dimdate_gold") \
     .addColumn("OrderDate", DateType()) \
     .addColumn("Day", IntegerType()) \
     .addColumn("Month", IntegerType()) \
     .addColumn("Year", IntegerType()) \
     .addColumn("mmmyyyy", StringType()) \
     .addColumn("yyyymm", StringType()) \
     .execute()
```

Note: You can run the `display(df)` command at any time to check the progress of your work. In this case, you'd run 'display(dfdimDate_gold)' to see the contents of the dimDate_gold dataframe.

5. In a new code block, **add and run the following code** to create a dataframe for your date dimension, **dimdate_gold**:

codeCopy

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

 # Create dataframe for dimDate_gold

 dfdimDate_gold = df.dropDuplicates(["OrderDate"]).select(col("OrderDate"),
\
         dayofmonth("OrderDate").alias("Day"), \
         month("OrderDate").alias("Month"), \
         year("OrderDate").alias("Year"), \
         date_format(col("OrderDate"), "MMM-yyyy").alias("mmmyyyy"), \
         date_format(col("OrderDate"), "yyyyMM").alias("yyyymm"), \
     ).orderBy("OrderDate")

 # Display the first 10 rows of the dataframe to preview your data

 display(dfdimDate_gold.head(10))
```

6. You're separating the code out into new code blocks so that you can understand and watch what's happening in the notebook as you transform the data. In another new code block, **add and run the following code** to update the date dimension as new data comes in:

codeCopy

```
 from delta.tables import *
```

```
deltaTable = DeltaTable.forPath(spark, 'Tables/dimdate_gold')

dfUpdates = dfdimDate_gold

deltaTable.alias('gold') \
  .merge(
    dfUpdates.alias('updates'),
    'gold.OrderDate = updates.OrderDate'
  ) \
   .whenMatchedUpdate(set =
    {

    }
  ) \
 .whenNotMatchedInsert(values =
    {
      "OrderDate": "updates.OrderDate",
      "Day": "updates.Day",
      "Month": "updates.Month",
      "Year": "updates.Year",
      "mmmyyyy": "updates.mmmyyyy",
      "yyyymm": "updates.yyyymm"
    }
  ) \
  .execute()
```

The date dimension is now set up. Now you'll create your customer dimension.

7. To build out the customer dimension table, **add a new code block**, paste and run the following code:

codeCopy

```
from pyspark.sql.types import *
from delta.tables import *

# Create customer_gold dimension delta table
DeltaTable.createIfNotExists(spark) \
    .tableName("sales.dimcustomer_gold") \
    .addColumn("CustomerName", StringType()) \
    .addColumn("Email",  StringType()) \
    .addColumn("First", StringType()) \
    .addColumn("Last", StringType()) \
    .addColumn("CustomerID", LongType()) \
    .execute()
```

8. In a new code block, **add and run the following code** to drop duplicate customers, select specific columns, and split the "CustomerName" column to create "First" and "Last" name columns:

codeCopy

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

 # Create customer_silver dataframe

 dfdimCustomer_silver = 
df.dropDuplicates(["CustomerName","Email"]).select(col("CustomerName"),col(
"Email")) \
     .withColumn("First",split(col("CustomerName"), " ").getItem(0)) \
     .withColumn("Last",split(col("CustomerName"), " ").getItem(1))

 # Display the first 10 rows of the dataframe to preview your data

 display(dfdimCustomer_silver.head(10))
```

Here you have created a new DataFrame dfdimCustomer_silver by performing various transformations such as dropping duplicates, selecting specific columns, and splitting the "CustomerName" column to create "First" and "Last" name columns. The result is a DataFrame with cleaned and structured customer data, including separate "First" and "Last" name columns extracted from the "CustomerName" column.

9. Next we'll **create the ID column for our customers**. In a new code block, paste and run the following:

codeCopy

```
 from pyspark.sql.functions import monotonically_increasing_id, col, when, 
coalesce, max, lit

 dfdimCustomer_temp = spark.read.table("Sales.dimCustomer_gold")

 MAXCustomerID = 
dfdimCustomer_temp.select(coalesce(max(col("CustomerID")),lit(0)).alias("MA
XCustomerID")).first()[0]

 dfdimCustomer_gold = 
dfdimCustomer_silver.join(dfdimCustomer_temp,(dfdimCustomer_silver.Customer
Name == dfdimCustomer_temp.CustomerName) & (dfdimCustomer_silver.Email == 
dfdimCustomer_temp.Email), "left_anti")

 dfdimCustomer_gold = 
dfdimCustomer_gold.withColumn("CustomerID",monotonically_increasing_id() + 
MAXCustomerID + 1)

 # Display the first 10 rows of the dataframe to preview your data

 display(dfdimCustomer_gold.head(10))
```

Here you're cleaning and transforming customer data (dfdimCustomer_silver) by performing a left anti join to exclude duplicates that already exist in the dimCustomer_gold table, and then generating unique CustomerID values using the monotonically_increasing_id() function.

10. Now you'll ensure that your customer table remains up-to-date as new data comes in. **In a new code block**, paste and run the following:

codeCopy

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, 'Tables/dimcustomer_gold')

dfUpdates = dfdimCustomer_gold

deltaTable.alias('gold') \
  .merge(
    dfUpdates.alias('updates'),
    'gold.CustomerName = updates.CustomerName AND gold.Email = 
updates.Email'
  ) \
   .whenMatchedUpdate(set =
    {

    }
  ) \
 .whenNotMatchedInsert(values =
    {
      "CustomerName": "updates.CustomerName",
      "Email": "updates.Email",
      "First": "updates.First",
      "Last": "updates.Last",
      "CustomerID": "updates.CustomerID"
    }
  ) \
  .execute()
```

11. Now you'll **repeat those steps to create your product dimension**. In a new code block, paste and run the following:

codeCopy

```
from pyspark.sql.types import *
from delta.tables import *

DeltaTable.createIfNotExists(spark) \
    .tableName("sales.dimproduct_gold") \
    .addColumn("ItemName", StringType()) \
    .addColumn("ItemID", LongType()) \
    .addColumn("ItemInfo", StringType()) \
    .execute()
```

12. **Add another code block** to create the **product_silver** dataframe.

codeCopy

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

# Create product_silver dataframe

dfdimProduct_silver = df.dropDuplicates(["Item"]).select(col("Item")) \
    .withColumn("ItemName",split(col("Item"), ", ").getItem(0)) \
    .withColumn("ItemInfo",when((split(col("Item"), ",
").getItem(1).isNull() | (split(col("Item"), ",
").getItem(1)=="")),lit("")).otherwise(split(col("Item"), ",
").getItem(1)))

# Display the first 10 rows of the dataframe to preview your data

display(dfdimProduct_silver.head(10))
```

13. Now you'll create IDs for your **dimProduct_gold table**. Add the following syntax to a new code block and run it:

codeCopy

```
from pyspark.sql.functions import monotonically_increasing_id, col, lit,
max, coalesce

#dfdimProduct_temp = dfdimProduct_silver
dfdimProduct_temp = spark.read.table("Sales.dimProduct_gold")

MAXProductID =
dfdimProduct_temp.select(coalesce(max(col("ItemID")),lit(0)).alias("MAXItem
ID")).first()[0]

dfdimProduct_gold =
dfdimProduct_silver.join(dfdimProduct_temp,(dfdimProduct_silver.ItemName ==
dfdimProduct_temp.ItemName) & (dfdimProduct_silver.ItemInfo ==
dfdimProduct_temp.ItemInfo), "left_anti")

dfdimProduct_gold =
dfdimProduct_gold.withColumn("ItemID",monotonically_increasing_id() +
MAXProductID + 1)

# Display the first 10 rows of the dataframe to preview your data

display(dfdimProduct_gold.head(10))
```

This calculates the next available product ID based on the current data in the table, assigns these new IDs to the products, and then displays the updated product information.

14. Similar to what you've done with your other dimensions, you need to ensure that your product table remains up-to-date as new data comes in. **In a new code block**, paste and run the following:

    codeCopy

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, 'Tables/dimproduct_gold')

dfUpdates = dfdimProduct_gold

deltaTable.alias('gold') \
  .merge(
        dfUpdates.alias('updates'),
        'gold.ItemName = updates.ItemName AND gold.ItemInfo =
updates.ItemInfo'
        ) \
        .whenMatchedUpdate(set =
        {

        }
        ) \
        .whenNotMatchedInsert(values =
         {
          "ItemName": "updates.ItemName",
          "ItemInfo": "updates.ItemInfo",
          "ItemID": "updates.ItemID"
          }
          ) \
          .execute()
```

Now that you have your dimensions built out, the final step is to create the fact table.

15. **In a new code block**, paste and run the following code to create the **fact table**:

    codeCopy

```
from pyspark.sql.types import *
from delta.tables import *

DeltaTable.createIfNotExists(spark) \
    .tableName("sales.factsales_gold") \
    .addColumn("CustomerID", LongType()) \
    .addColumn("ItemID", LongType()) \
    .addColumn("OrderDate", DateType()) \
    .addColumn("Quantity", IntegerType()) \
    .addColumn("UnitPrice", FloatType()) \
    .addColumn("Tax", FloatType()) \
    .execute()
```

16. **In a new code block**, paste and run the following code to create a **new dataframe** to combine sales data with customer and product information include customer ID, item ID, order date, quantity, unit price, and tax:

    codeCopy

```
from pyspark.sql.functions import col

dfdimCustomer_temp = spark.read.table("Sales.dimCustomer_gold")
dfdimProduct_temp = spark.read.table("Sales.dimProduct_gold")

df = df.withColumn("ItemName",split(col("Item"), ", ").getItem(0)) \
    .withColumn("ItemInfo",when((split(col("Item"), ",
").getItem(1).isNull() | (split(col("Item"), ",
").getItem(1)=="")),lit("")).otherwise(split(col("Item"), ",
").getItem(1))) \


# Create Sales_gold dataframe

dffactSales_gold =
df.alias("df1").join(dfdimCustomer_temp.alias("df2"),(df.CustomerName ==
dfdimCustomer_temp.CustomerName) & (df.Email == dfdimCustomer_temp.Email),
"left") \
        .join(dfdimProduct_temp.alias("df3"),(df.ItemName ==
dfdimProduct_temp.ItemName) & (df.ItemInfo == dfdimProduct_temp.ItemInfo),
"left") \
    .select(col("df2.CustomerID") \
        , col("df3.ItemID") \
        , col("df1.OrderDate") \
        , col("df1.Quantity") \
        , col("df1.UnitPrice") \
        , col("df1.Tax") \
    ).orderBy(col("df1.OrderDate"), col("df2.CustomerID"),
col("df3.ItemID"))

# Display the first 10 rows of the dataframe to preview your data

display(dffactSales_gold.head(10))
```

17. Now you'll ensure that sales data remains up-to-date by running the following code in a **new code block**:

    codeCopy

```
from delta.tables import *

deltaTable = DeltaTable.forPath(spark, 'Tables/factsales_gold')

dfUpdates = dffactSales_gold

deltaTable.alias('gold') \
  .merge(
    dfUpdates.alias('updates'),
    'gold.OrderDate = updates.OrderDate AND gold.CustomerID =
updates.CustomerID AND gold.ItemID = updates.ItemID'
  ) \
   .whenMatchedUpdate(set =
    {

    }
  ) \
 .whenNotMatchedInsert(values =
    {
      "CustomerID": "updates.CustomerID",
      "ItemID": "updates.ItemID",
      "OrderDate": "updates.OrderDate",
      "Quantity": "updates.Quantity",
      "UnitPrice": "updates.UnitPrice",
      "Tax": "updates.Tax"
    }
  ) \
  .execute()
```

Here you're using Delta Lake's merge operation to synchronize and update the factsales_gold table with new sales data (dffactSales_gold). The operation compares the order date, customer ID, and item ID between the existing data (silver table) and the new data (updates DataFrame), updating matching records and inserting new records as needed.

You now have a curated, modeled **gold** layer that can be used for reporting and analysis.

Create a semantic model

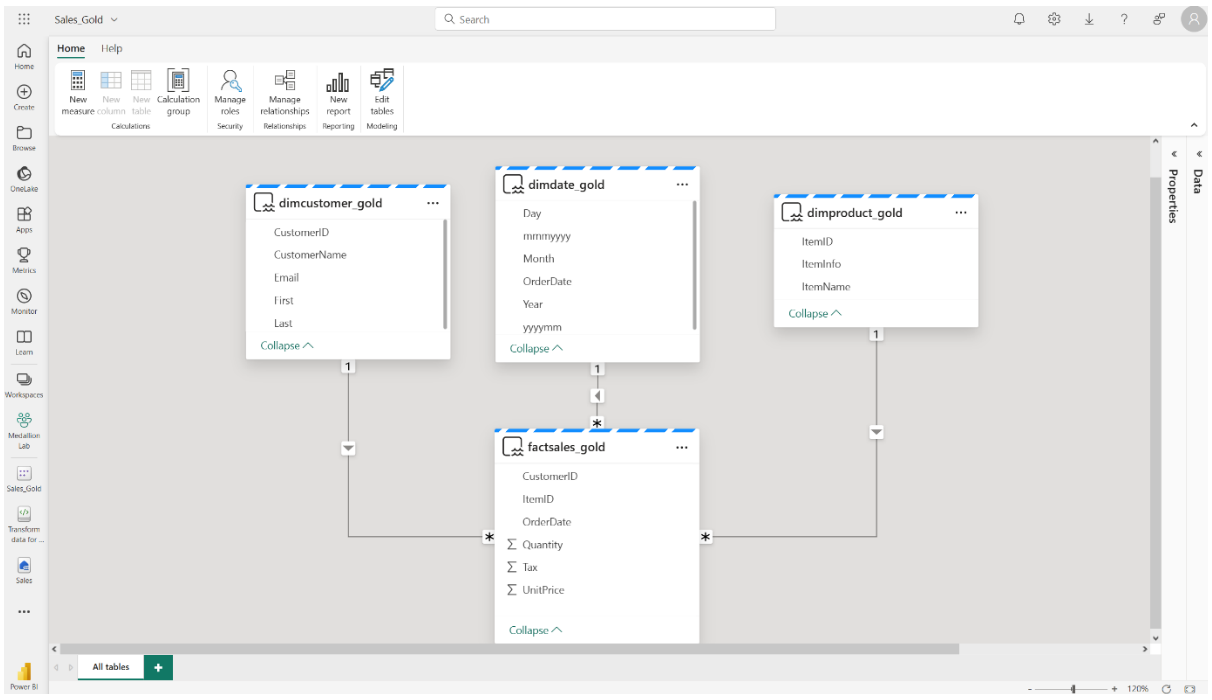
In your workspace, you can now use the gold layer to create a report and analyze the data. You can access the semantic model directly in your workspace to create relationships and measures for reporting.

Note that you can't use the **default semantic model** that is automatically created when you create a lakehouse. You must create a new semantic model that includes the gold tables you created in this exercise, from the lakehouse explorer.

1. In your workspace, navigate to your **Sales** lakehouse.

2. Select **New semantic model** from the ribbon of the lakehouse explorer view.
3. Assign the name **Sales_Gold** to your new semantic model.
4. Select your transformed gold tables to include in your semantic model and select **Confirm**.
   - dimdate_gold
   - dimcustomer_gold
   - dimproduct_gold
   - factsales_gold

   This will open the semantic model in Fabric where you can create relationships and measures, as shown here:



From here, you or other members of your data team can create reports and dashboards based on the data in your lakehouse. These reports will be connected directly to the gold layer of your lakehouse, so they'll always reflect the latest data.

Clean up resources

In this exercise, you've learned how to create a medallion architecture in a Microsoft Fabric lakehouse.

If you've finished exploring your lakehouse, you can delete the workspace you created for this exercise.

1. In the bar on the left, select the icon for your workspace to view all of the items it contains.

2. In the **...** menu on the toolbar, select **Workspace settings**.
3. In the **General** section, select **Remove this workspace**.