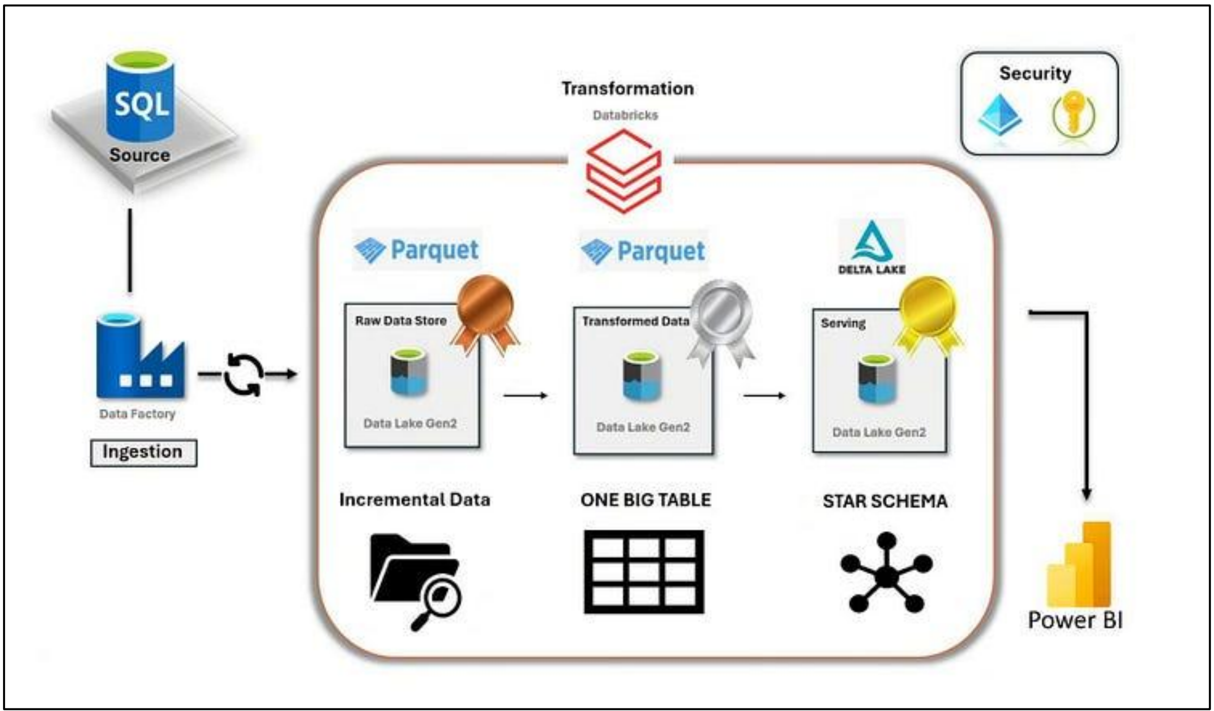
End-to-End Azure Data Engineering Pipeline

Using Azure SQL DB, Azure Data Factory, Databricks, Delta Lake, Power BI



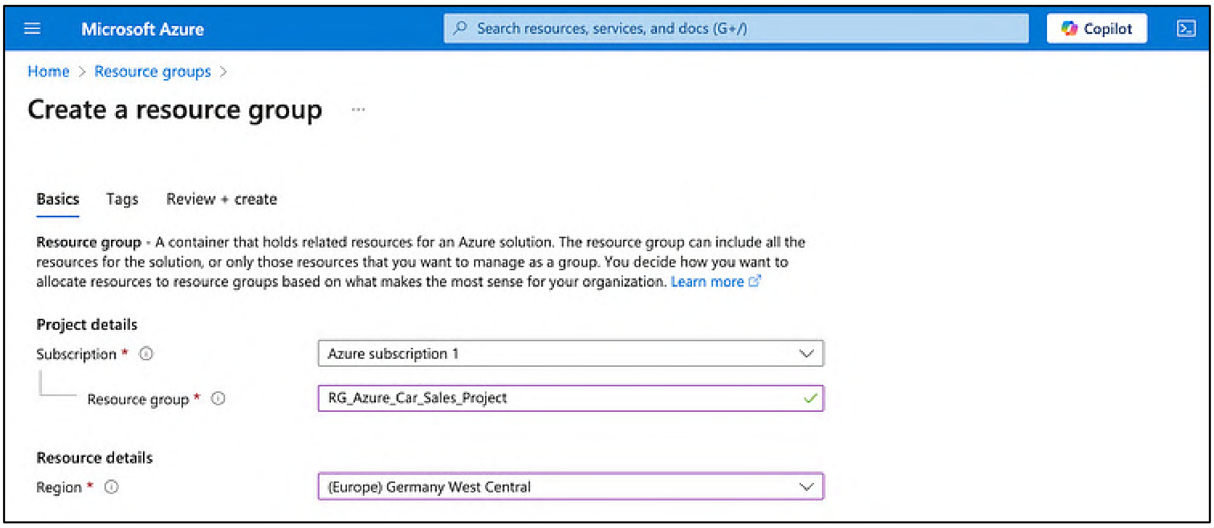
Project architecture

In this project you will learn the following concepts:

- Data modeling — star schema (Fact & Dimensions modeling)
- Slowly changing dimensions handling & Change Data Capture (CDC)
- Data Design Pattern: Medialion Architecture
- Azure Services for Data Engineering

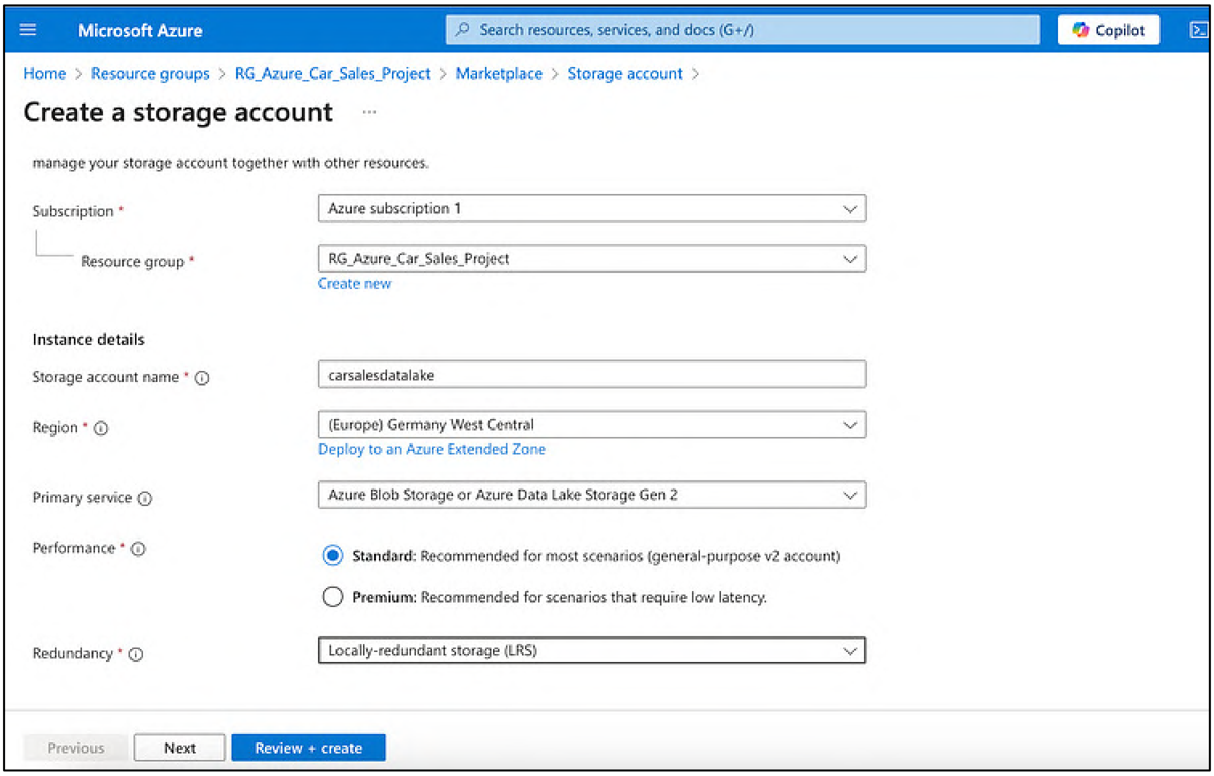
Step 1: Create a Resource Group

Starting with a resource group in Azure is not just a best practice but a foundational step toward effective cloud resource management. Resource groups enhance organization, improve security through access control, facilitate cost tracking, enable consistent deployments, and allow for environment isolation.



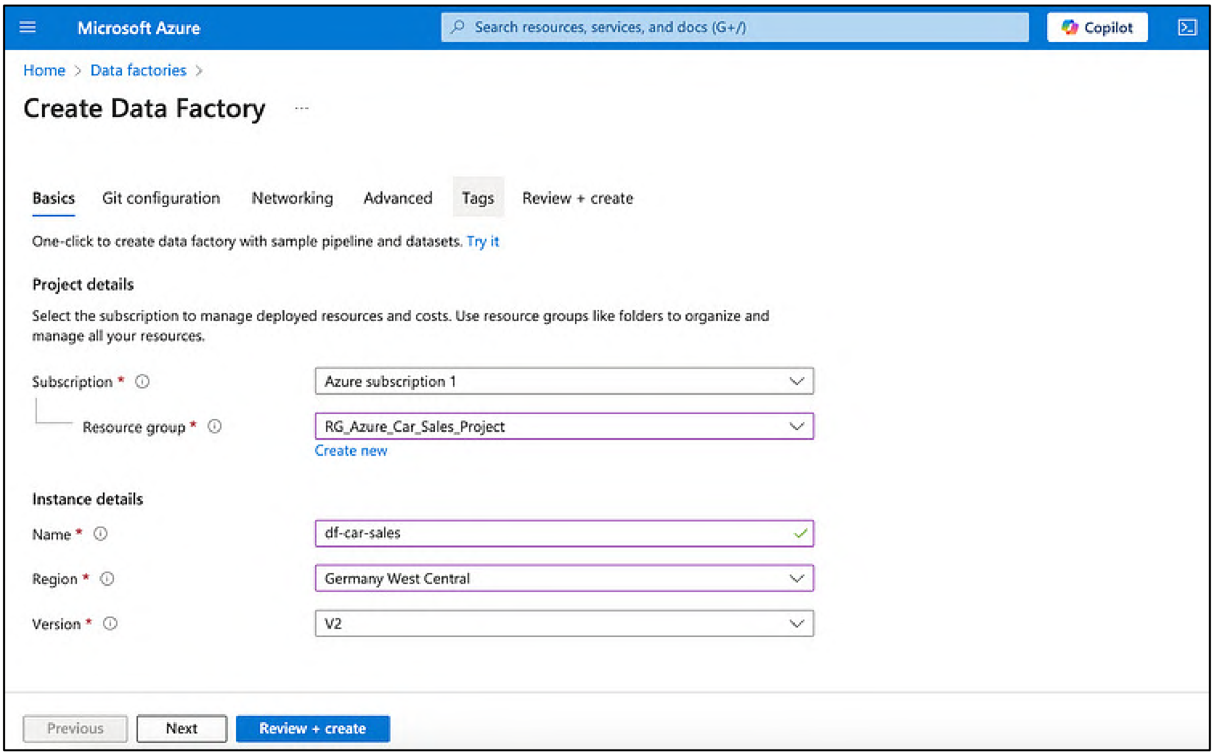
Step 2: Create a Storage Account (Datalake)

An Azure storage account contains all your Azure Storage data objects: blobs, files, queues, and tables. The storage account provides a unique namespace for your Azure Storage data accessible from anywhere in the world over HTTP or HTTPS.



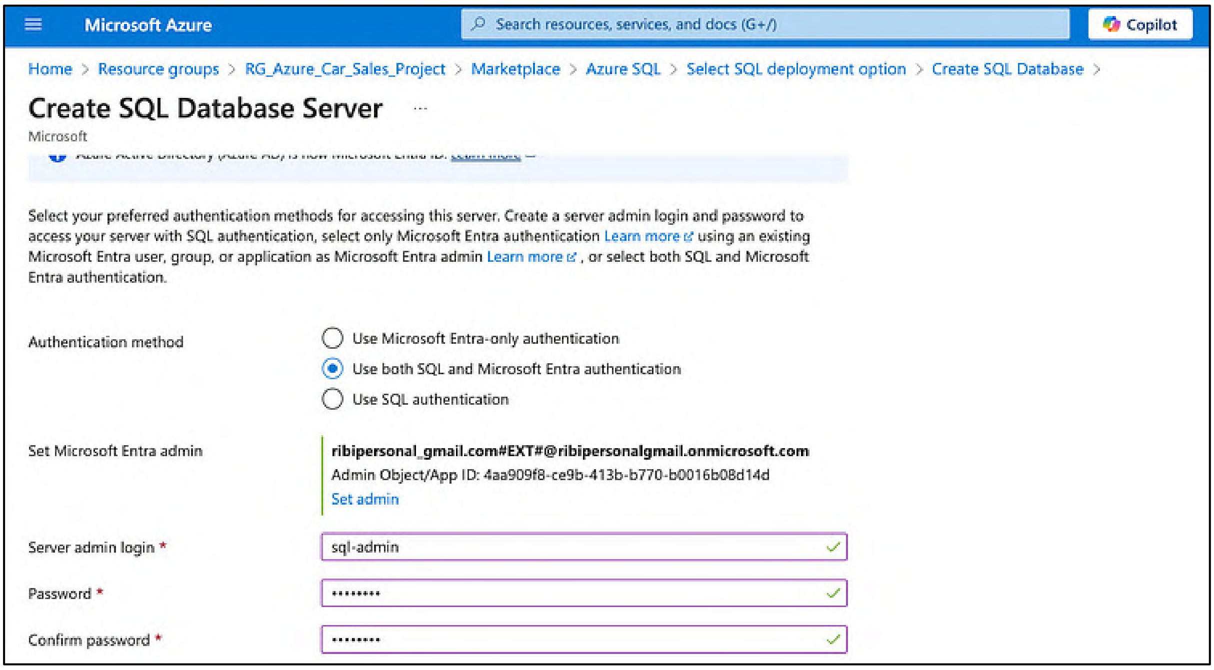
Step 3: Create a Data Factory

Data Factory provides a data integration and transformation layer and you can use it to create ETL and ELT pipelines.



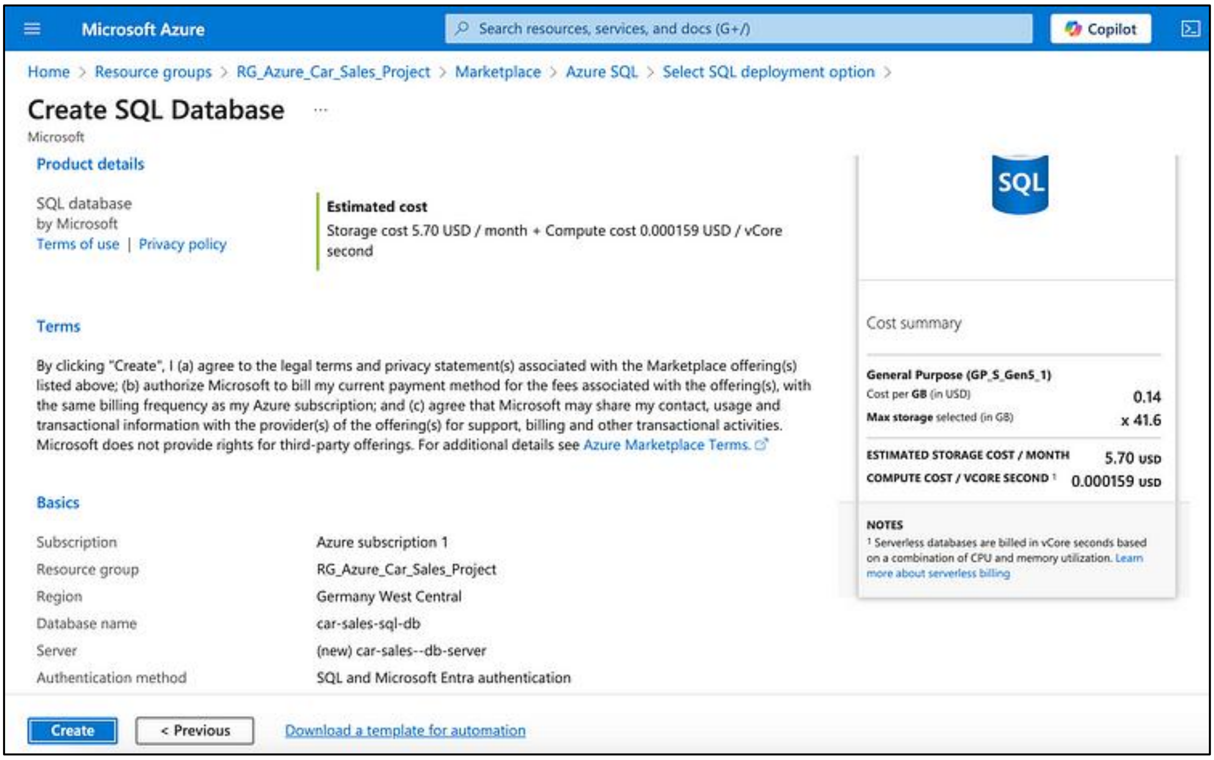
Step 4: Create an Azure SQL Database

Azure SQL allows you to create and manage your SQL Server resources from a single view, ranging from fully managed PaaS databases to IaaS virtual machines with direct OS and database engine access.



Create a Server

Continue with the steps to create a managed Azure SQL Database

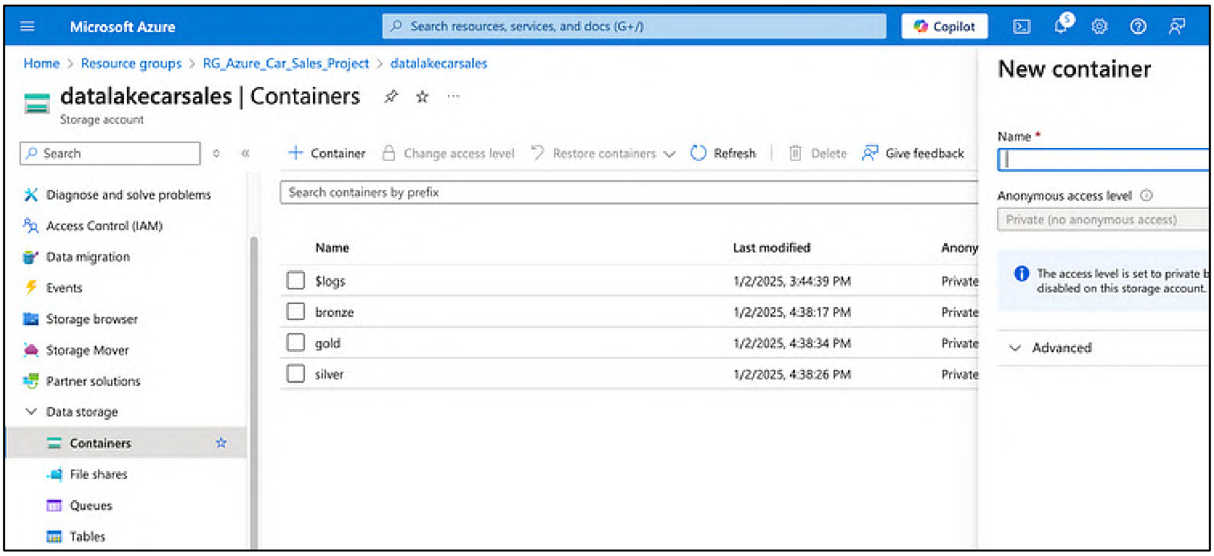


In networking choose a public endpoint

Step 5: Create Containers in the Data Lake

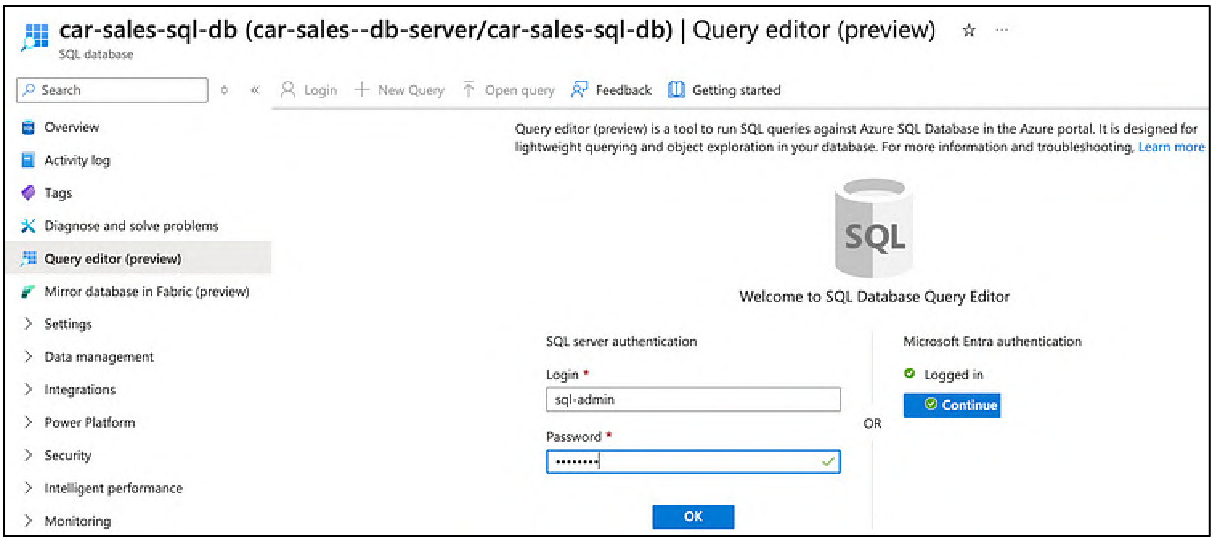
As we are following the Medallion Data Design pattern, create three containers:

- **Bronze** for the raw data
- **Silver** for the transformed data
- **Gold** for the aggregated data

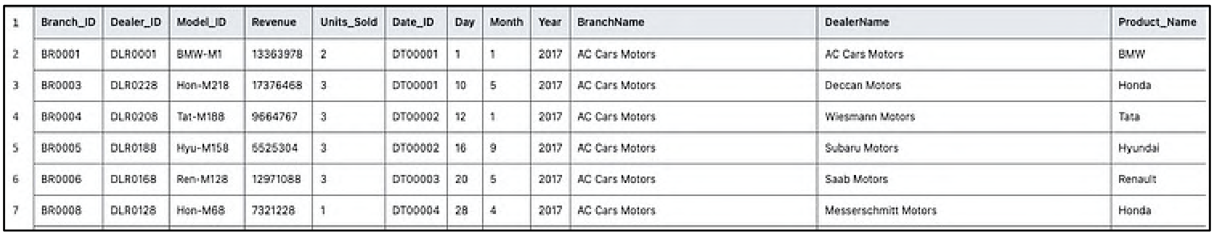


Step 6: Create a Table Schema in the Database

Navigate to the created database and click on Query Editor, you will be forwarded to the login interface, where you need to specify your admin credentials.



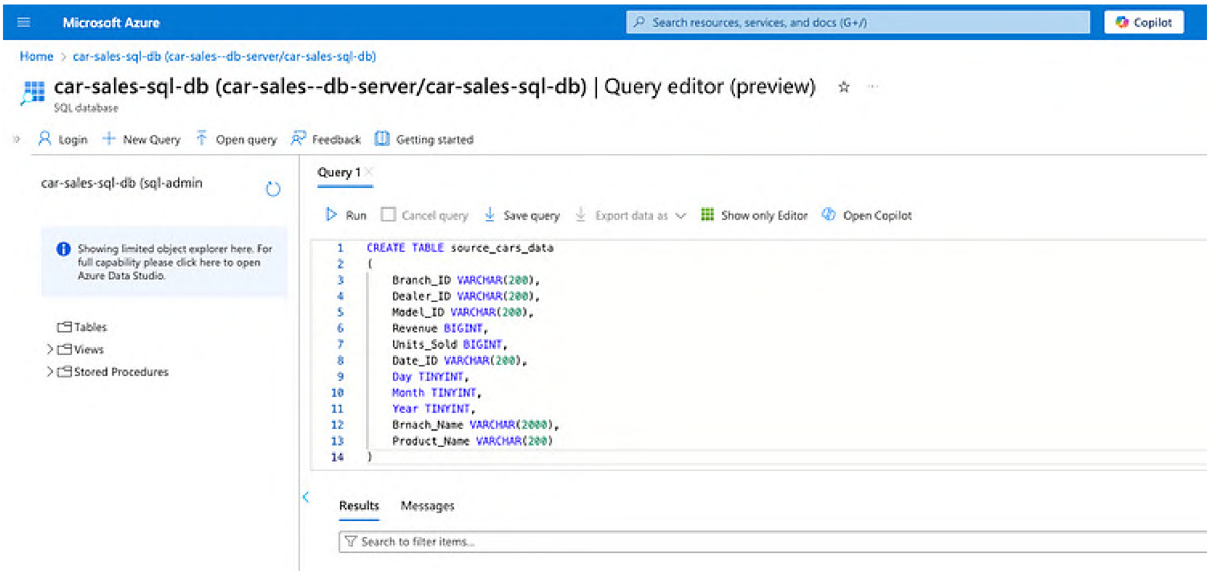
login to the database



1	Branch_ID	Dealer_ID	Model_ID	Revenue	Units_Sold	Date_ID	Day	Month	Year	BranchName	DealerName	Product_Name
2	BR0001	DLR0001	BMW-M1	13363978	2	DT00001	1	1	2017	AC Cars Motors	AC Cars Motors	BMW
3	BR0003	DLR0228	Hon-M218	17376468	3	DT00001	10	5	2017	AC Cars Motors	Deccan Motors	Honda
4	BR0004	DLR0208	Tat-M188	9664767	3	DT00002	12	1	2017	AC Cars Motors	Wiesmann Motors	Tata
5	BR0005	DLR0188	Hyu-M158	6525304	3	DT00002	16	9	2017	AC Cars Motors	Subaru Motors	Hyundai
6	BR0006	DLR0168	Ren-M128	12971088	3	DT00003	20	5	2017	AC Cars Motors	Saab Motors	Renault
7	BR0008	DLR0128	Hon-M68	7321228	1	DT00004	28	4	2017	AC Cars Motors	Messerschmitt Motors	Honda

Raw data structure

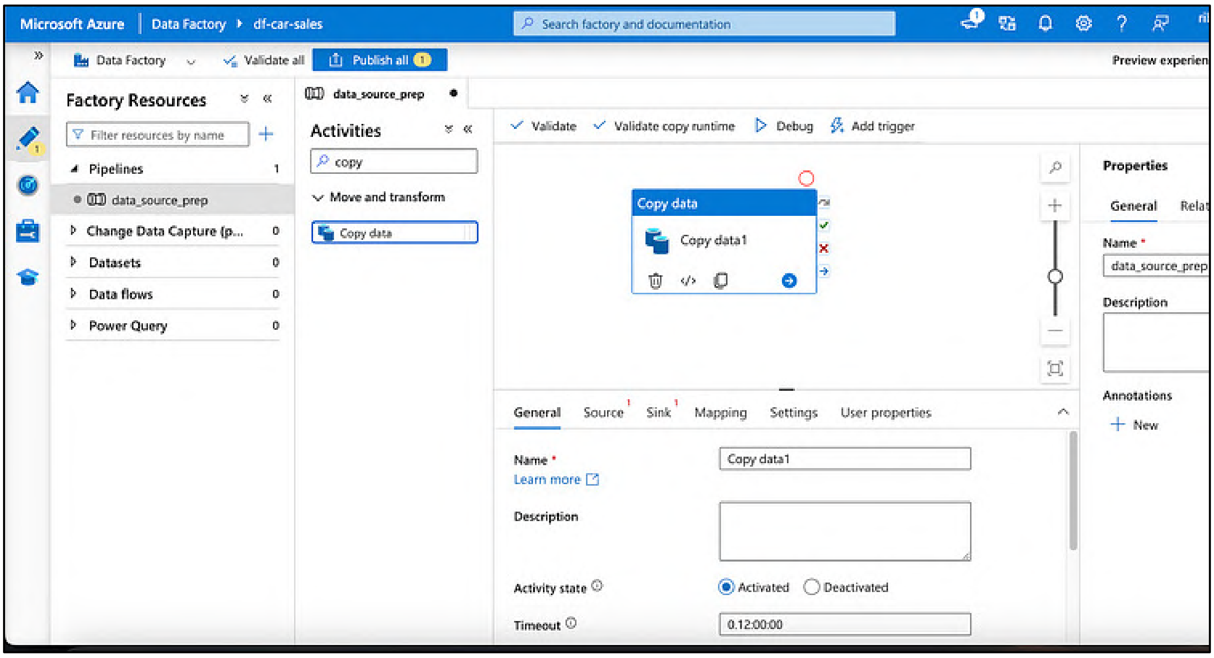
The next step is to create a table with the appropriate schema for the source raw data by creating a new query.



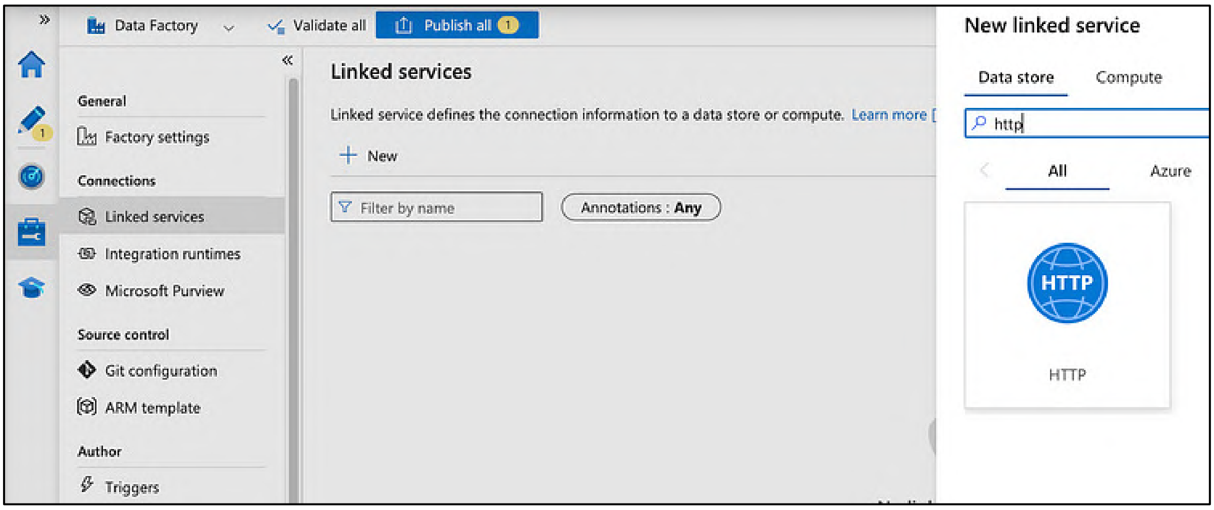
Create Table query

Step 7: Ingest raw data with Data Factory

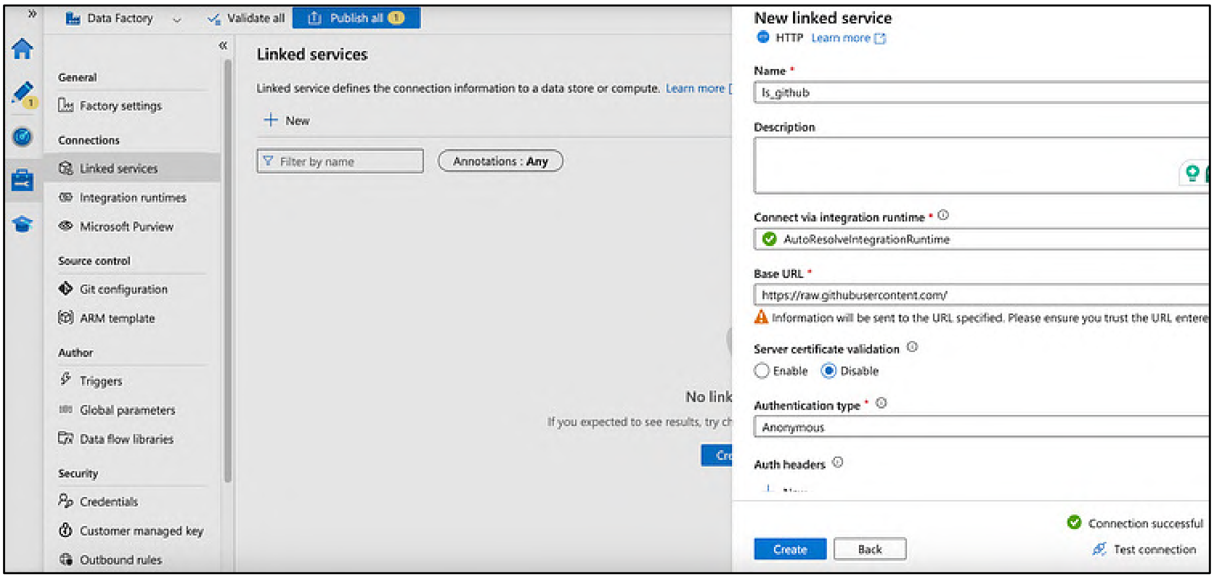
Launch Data Factory navigate to the Author tab and create a pipeline called ‘data_source_prep’ What we will do is copy data from GitHub to Azure SQL DB using Data Factory.



Then navigate to the Manage tab and click on Linked Services to create an HTTP connection to GitHub (where we have the source data) and this can be any other website from which you read raw data.



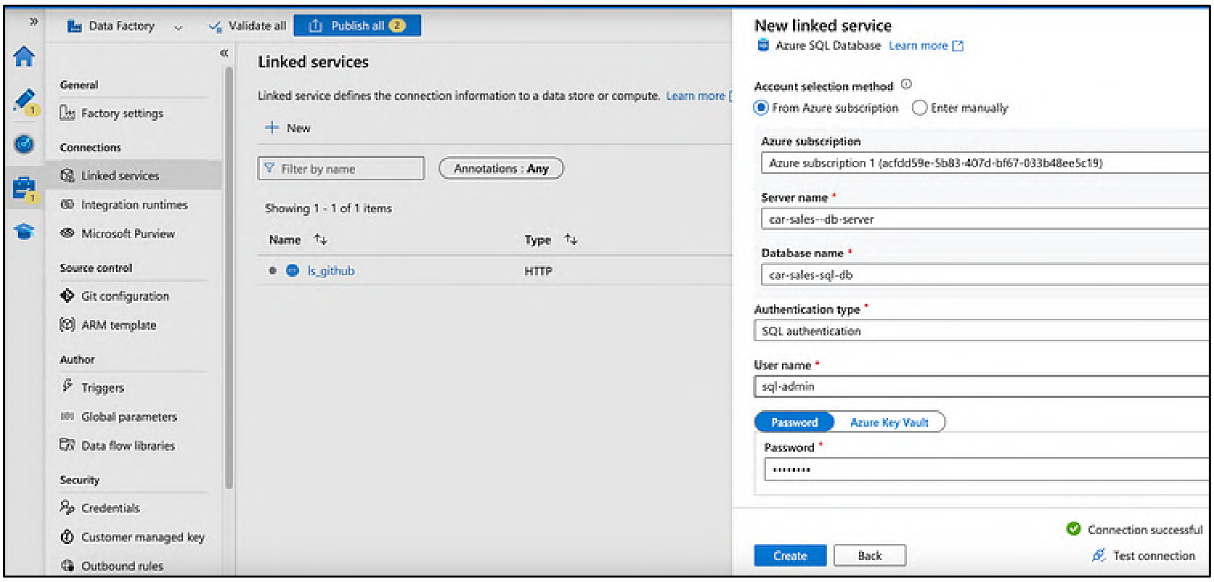
http linked service to connect to github



make sure to test the connection before clicking on create

The second linked connection will be the Azure SQL database to write the ingested data to the SQL database.

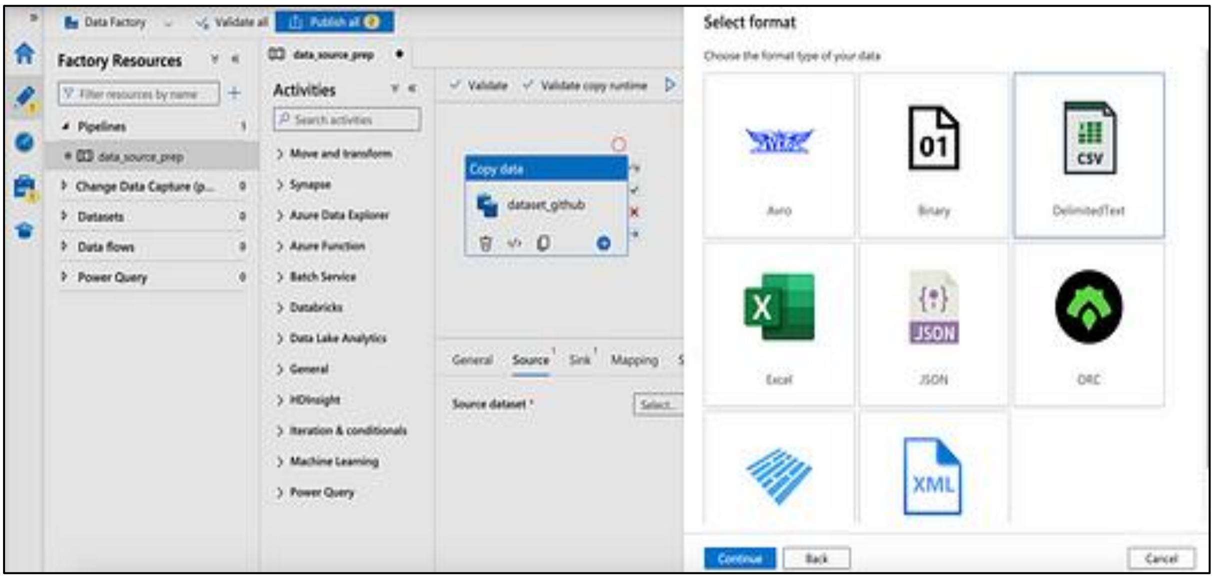
When you test the connection, you might get a connection error because of the firewall to protect the access to the database. To fix that error, navigate to the networking settings in the SQL server and click on “Enable Azure service to access this server” which you should find at the end of the page.



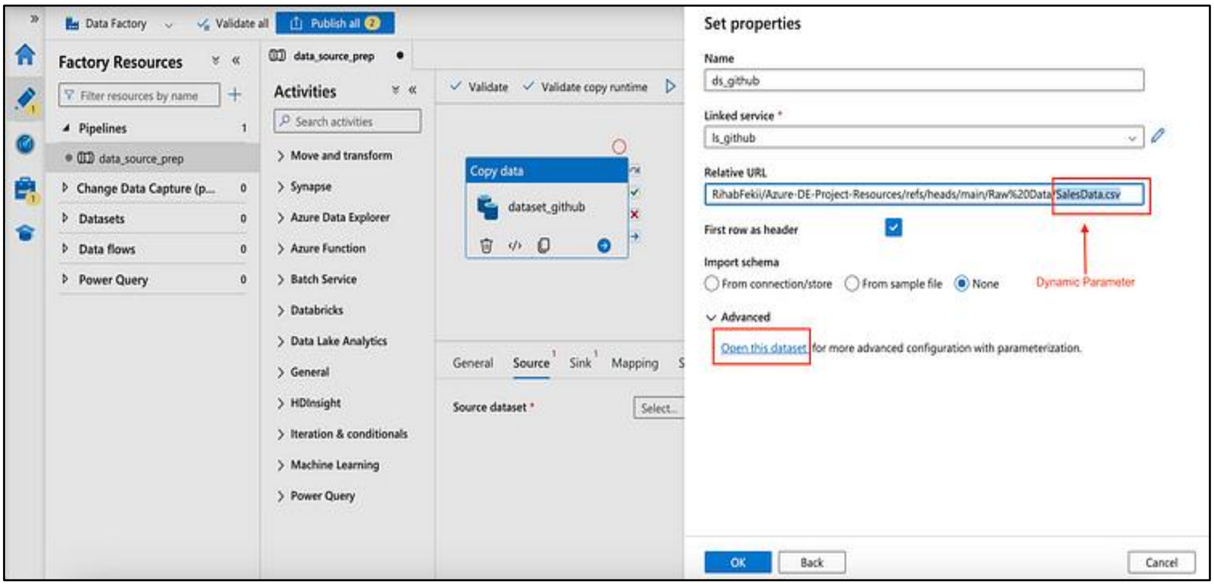
Dynamic ETL: parametrized dataset

After setting up the Linked Services, it is time to configure the data pipeline to create a dynamic dataset with a parameter of the file name.

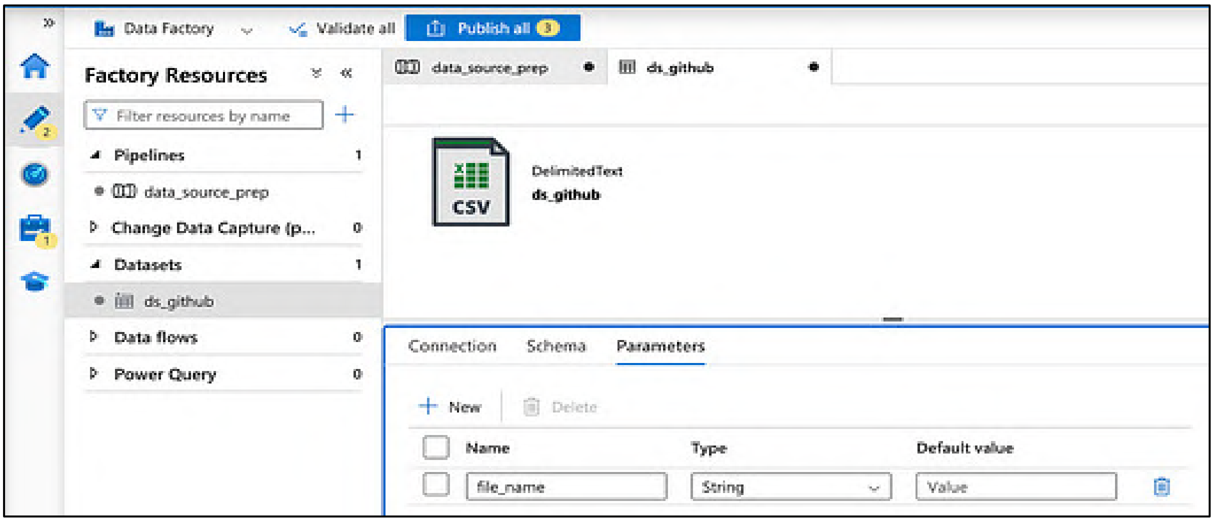
To do that, go to the Author tab and start configuring the Copy Data activity by first adding a new dataset in the source section, typing new, selecting an HTTP data store, and since the data is a CSV, selecting 'Delimited Text'.



Then we need to create the parameter for the dataset (file name)

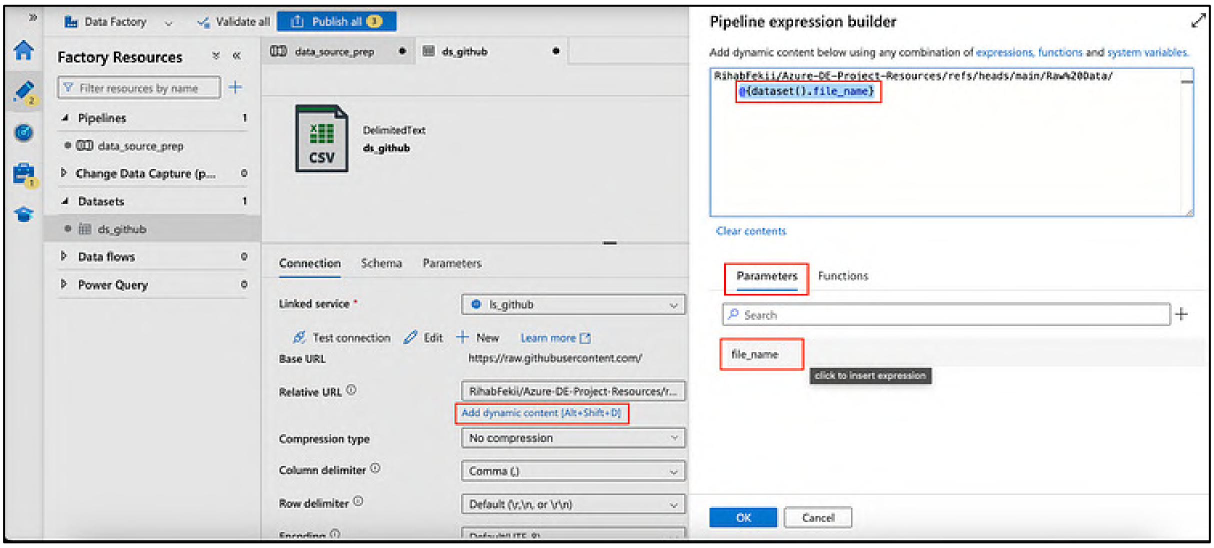


click on Open this dataset for more advanced configurations to add the parameter

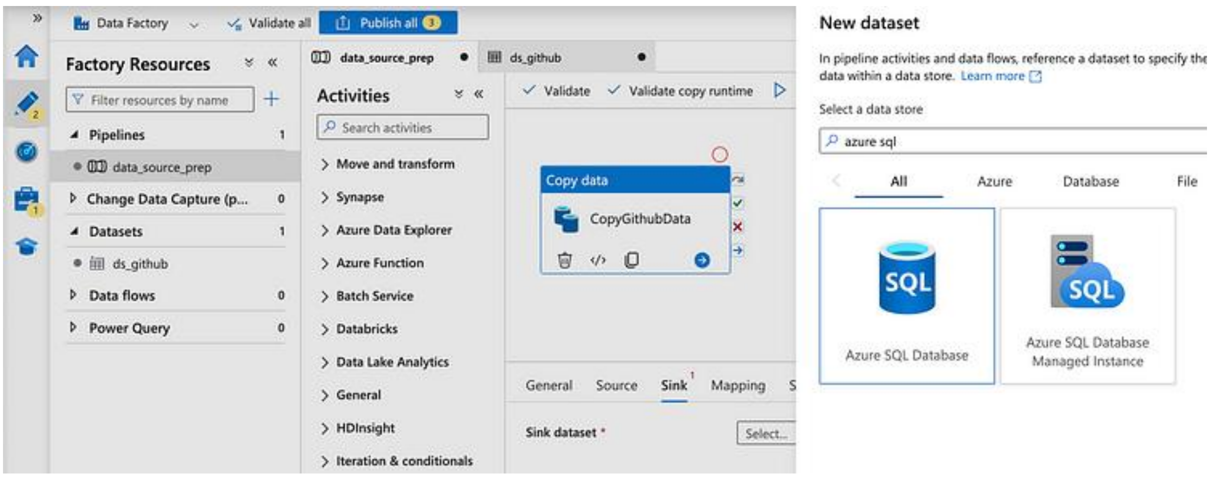


Added the parameter 'file_name'

Then navigate to the connection tab and click on A**dd dynamic content** to edit the relative URL and add the parameter that was just created.

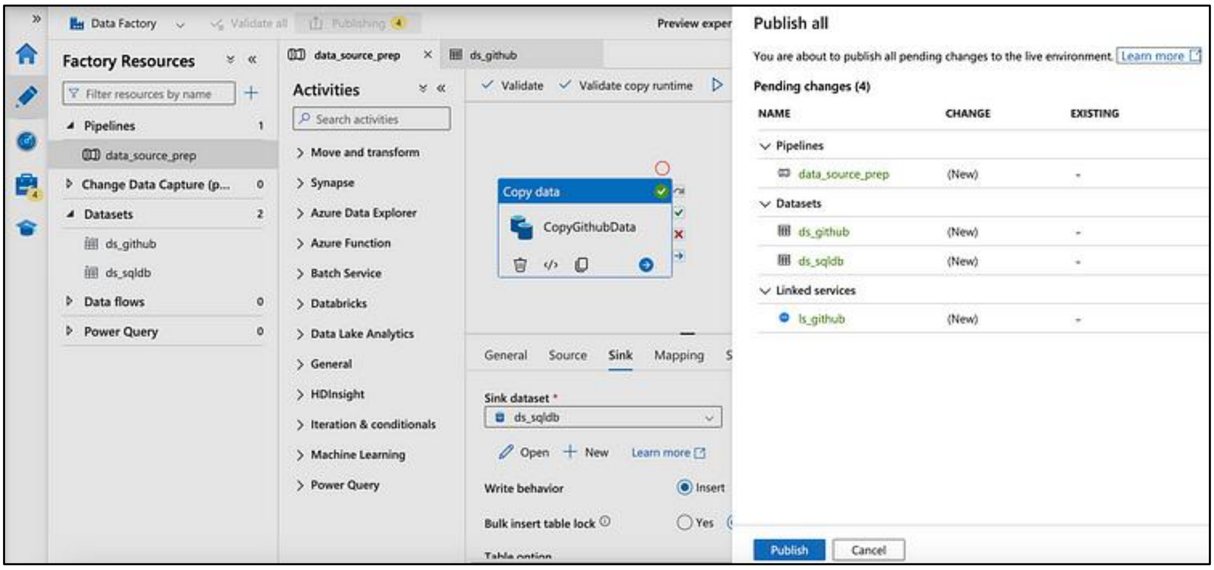


This way we created the dynamic data source, the next step is to configure the **sink** which is the destination where we will load the data in Azure SQL.

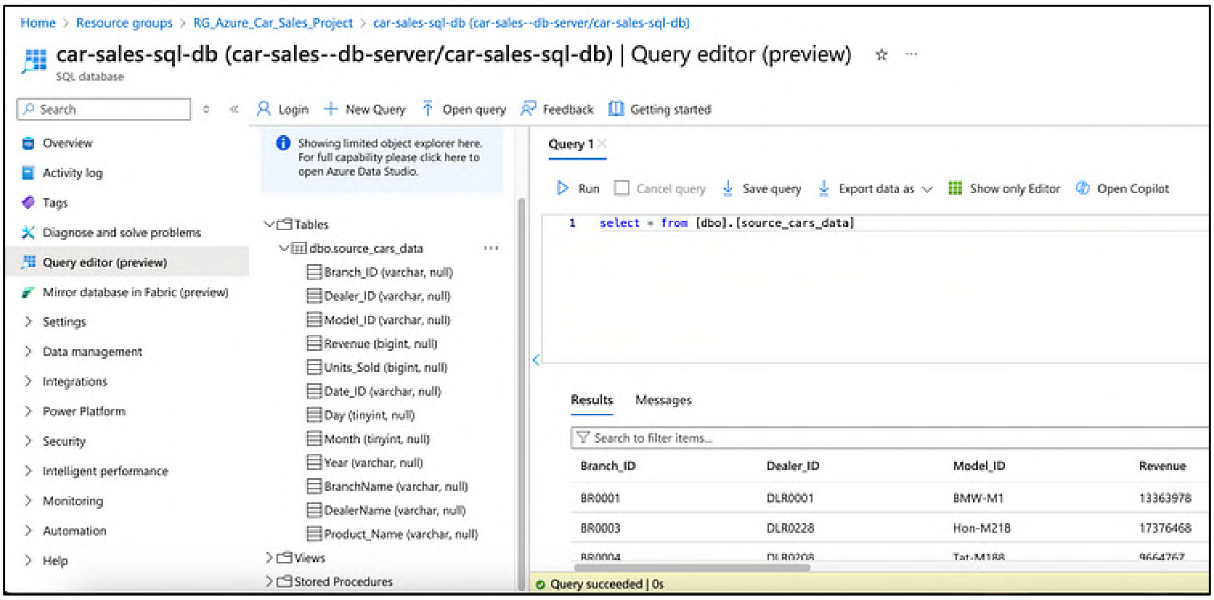


Sink config

After configuring the sink, click on Debug to run the pipeline and then click on Publish All to save the work.



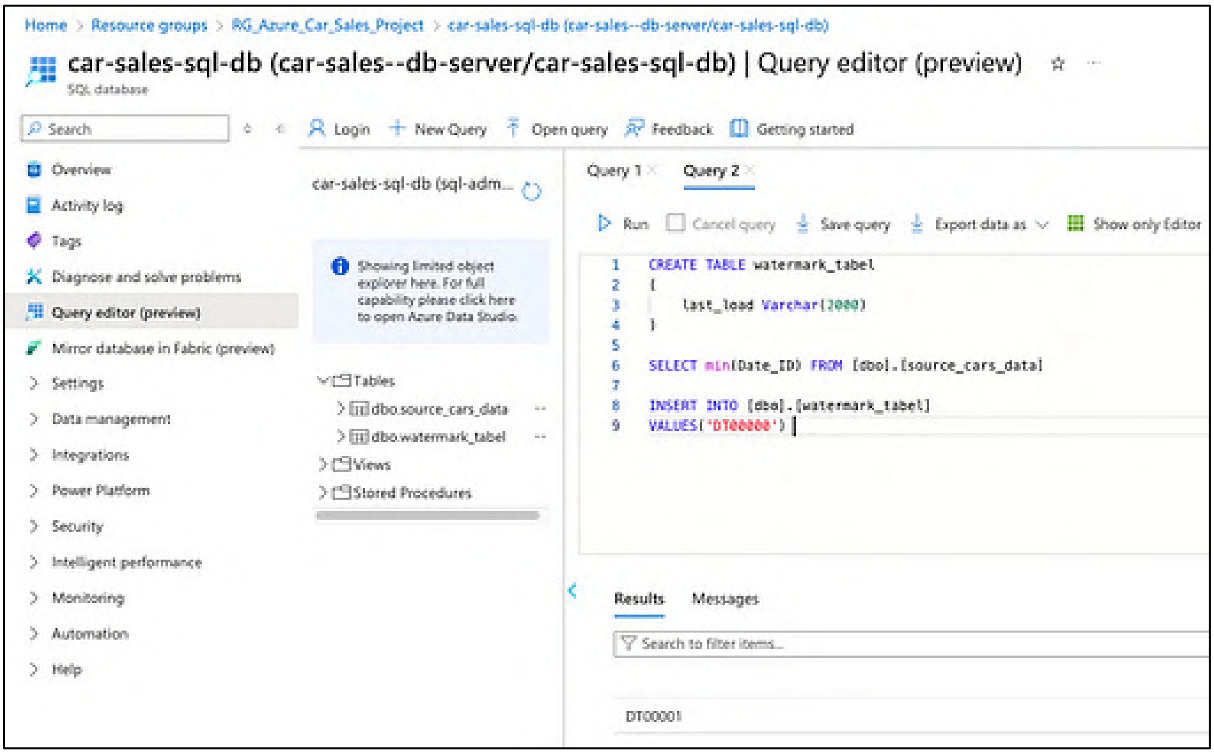
Then test that the copy data pipeline worked correctly by querying the data in the dataset.



Step 8: Incremental data Loading

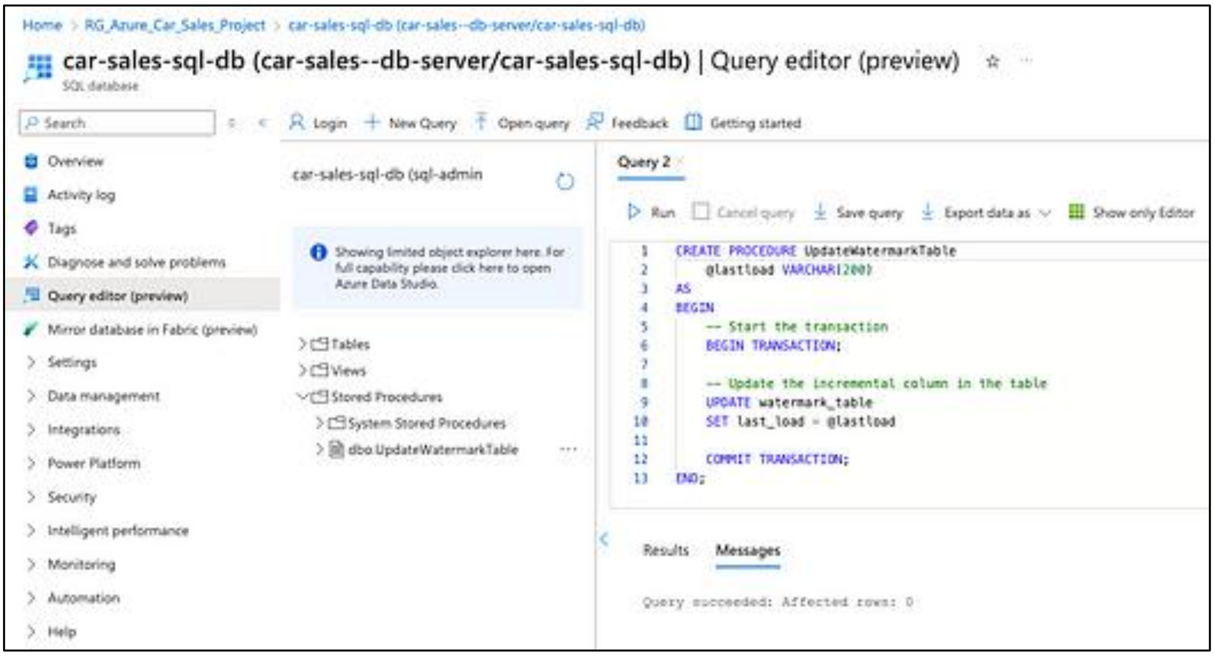
In this step, we need to load new data incrementally and automatically. To do that we will need to create two pipelines. One for the initial load and one for the incremental load and we create two parameters to save the current load data and the last load date.

Create a Watermark table to store the last load date identifier.

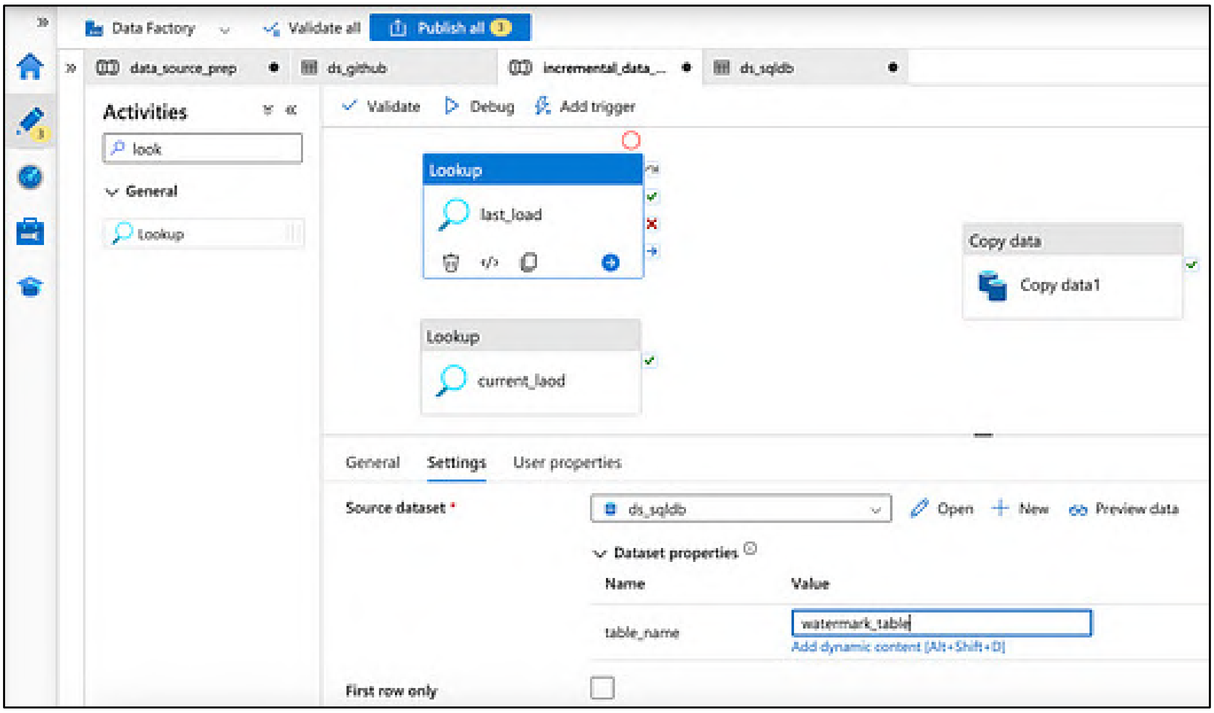


Anything above that DT0000 date identifier will insert all the data.

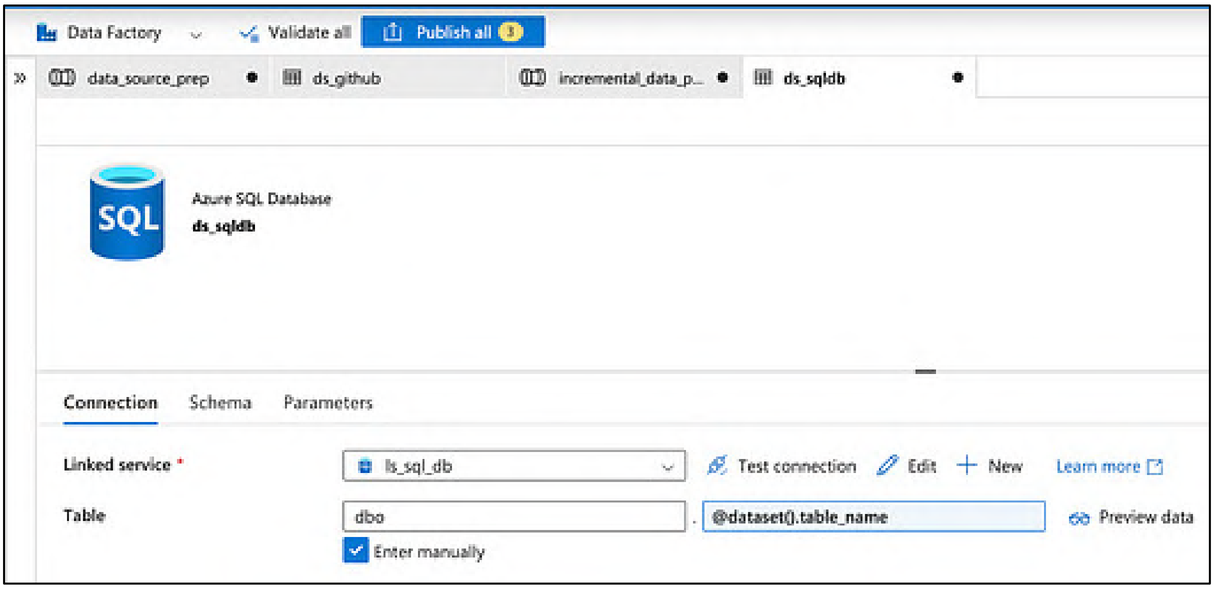
Now create a stored procedure to update this value again & again in the watermark table.



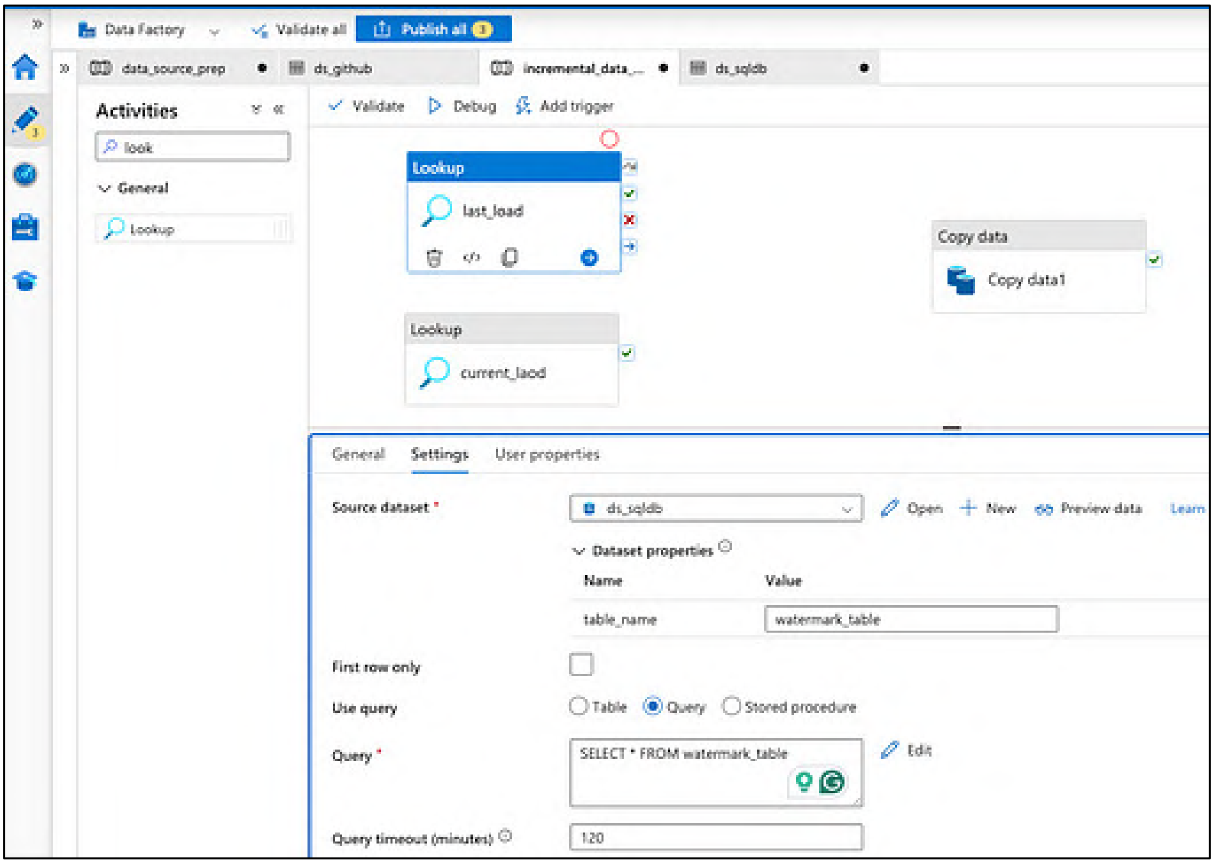
The next step is to add two activities "Lookup", one that captures the last_load date and the other that captures the current_load date.



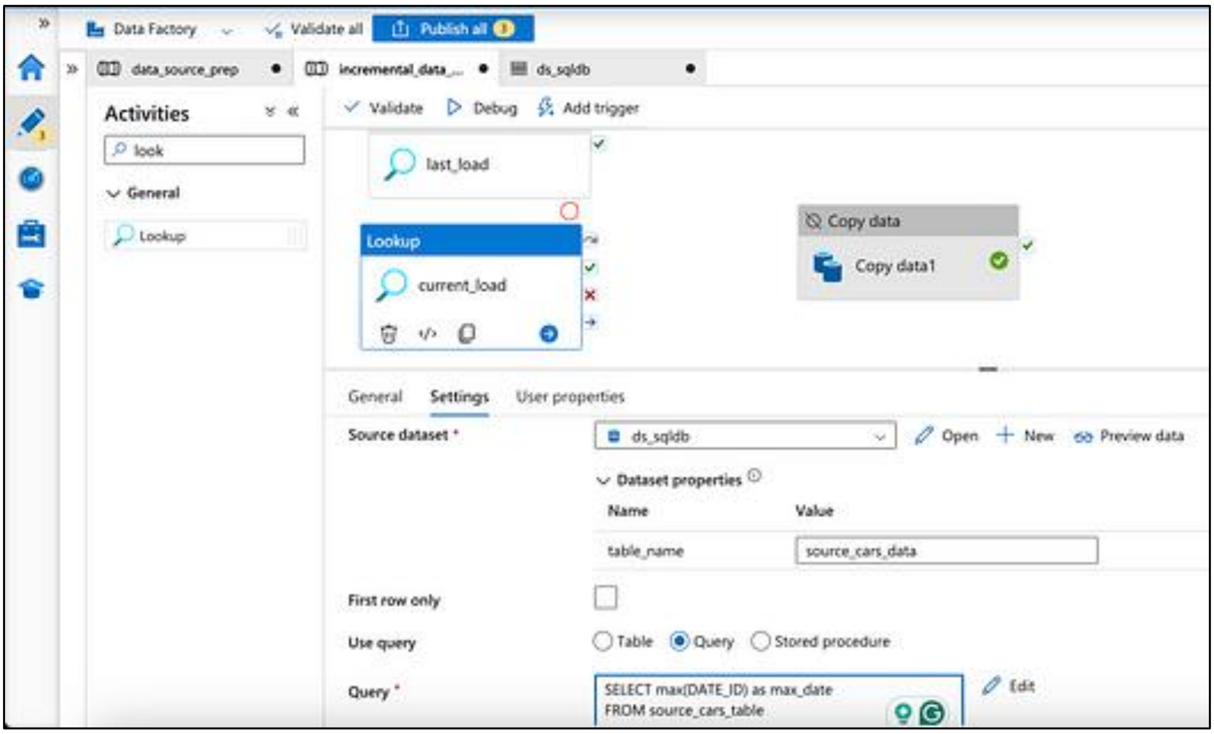
In the configuration of the lookup, select the dataset in the settings tab and then create a parameter for the table_name (which will be used in both Lookups).



Let's continue with the configuration of the Lookup activity 'last_load' to get the last load value via a query.



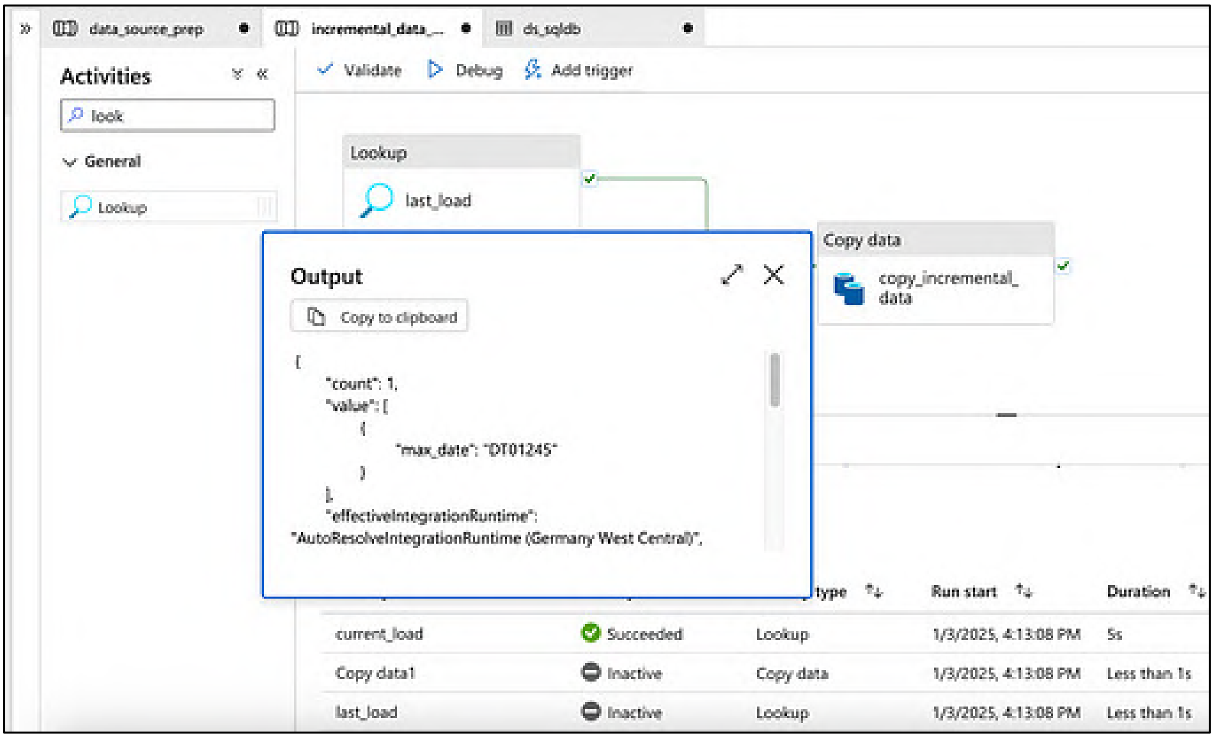
Then, configure the settings of the Lookup "current_load" as shown below.



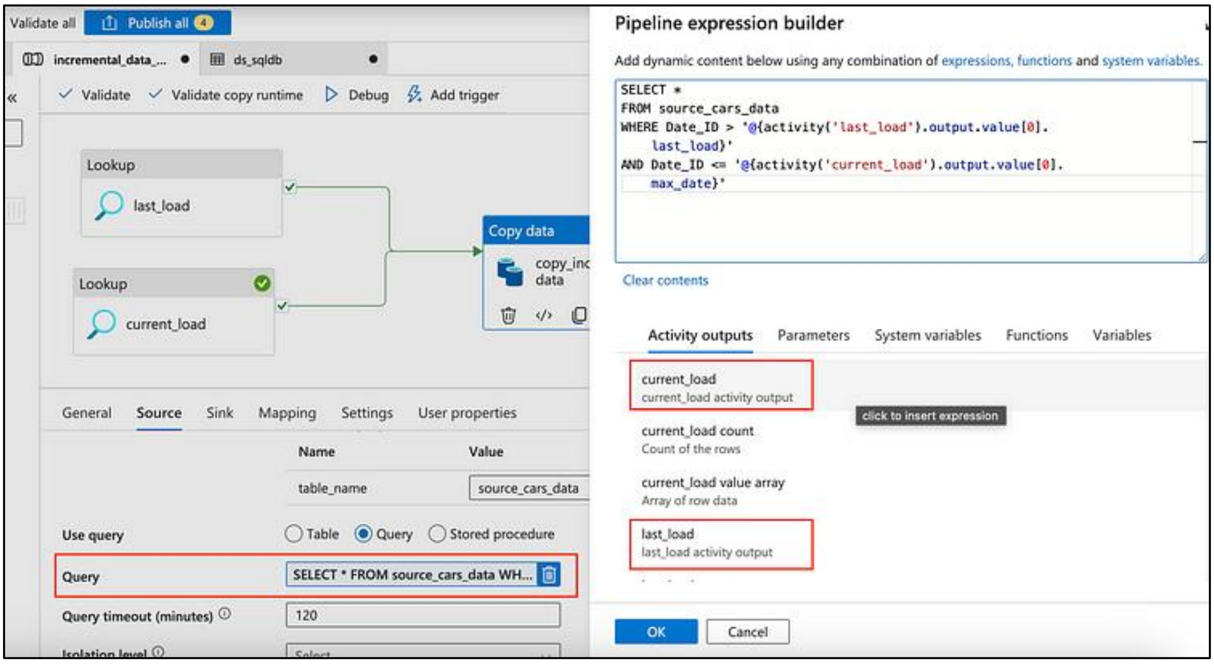
Then deactivate the other Activities and click on Debug to check the output of that activity. Afterward, connect the two lookups to the copy data activity on success by dragging the green check button.

Then we start to configure the Copy Data activity in which we will set up the copy_incremental_data. Then add the needed info in the source tab and then add dynamic content to add the parameters of the activity outputs.

since the output of the lookup activities is as follows:

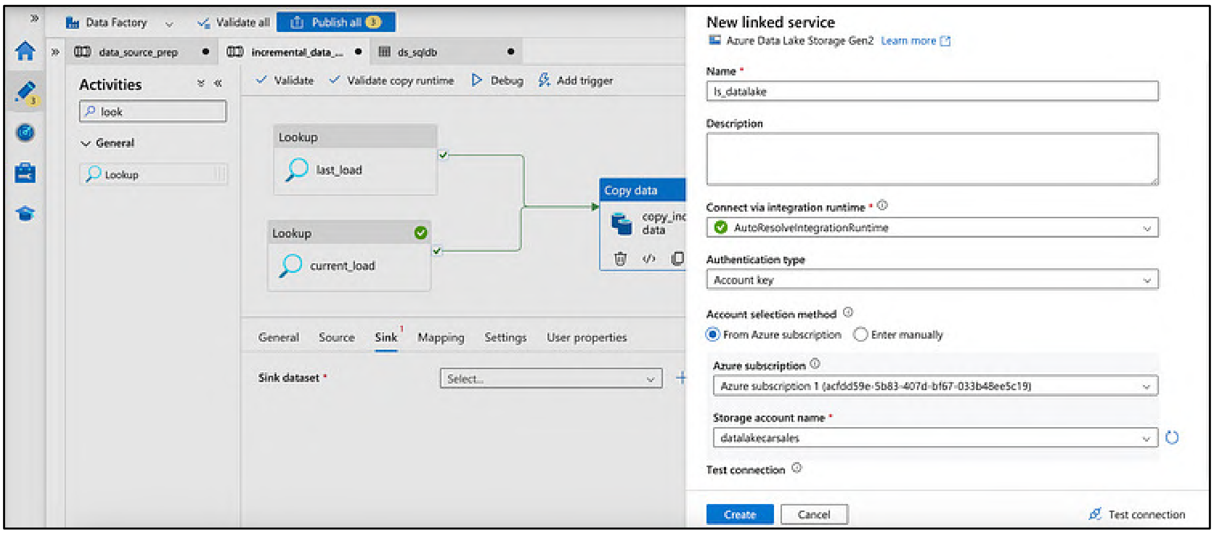


To get the max_date we need to write output.value[0].max_date and we do the same with output.value[0].last_load to be replaced in the parameter.

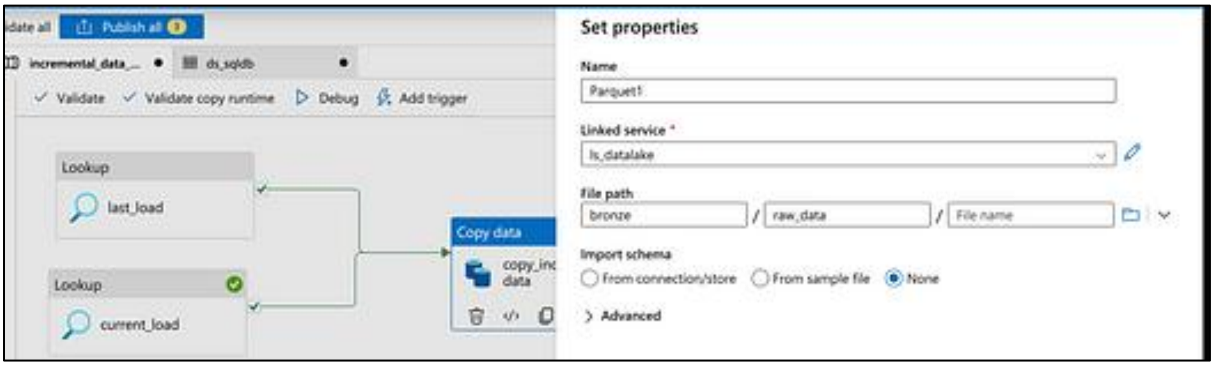


The next step is to add the sink information and at this level, we want to save the data to the Datalake (Bronze layer).

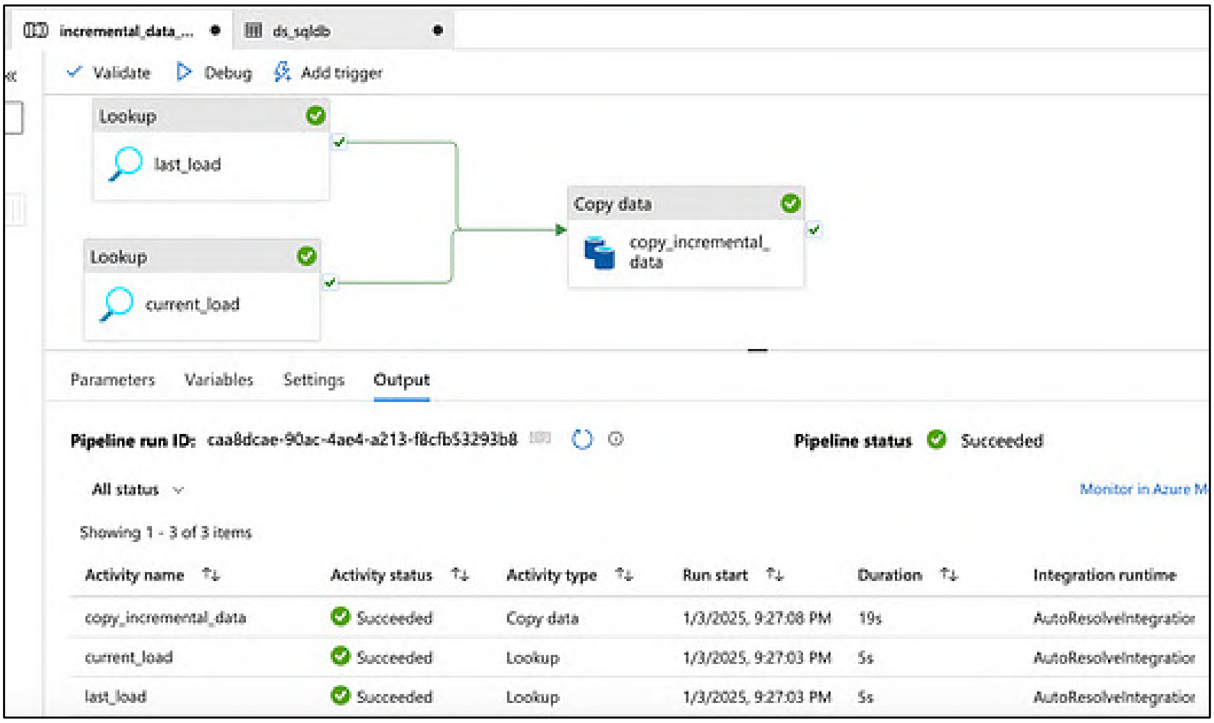
The data source would be Azure Datalake Storage Gen 2. Choose the format Parquet to save the data. Call the dataset ds_bronze create a linked service and link the Datalake we created at the beginning.



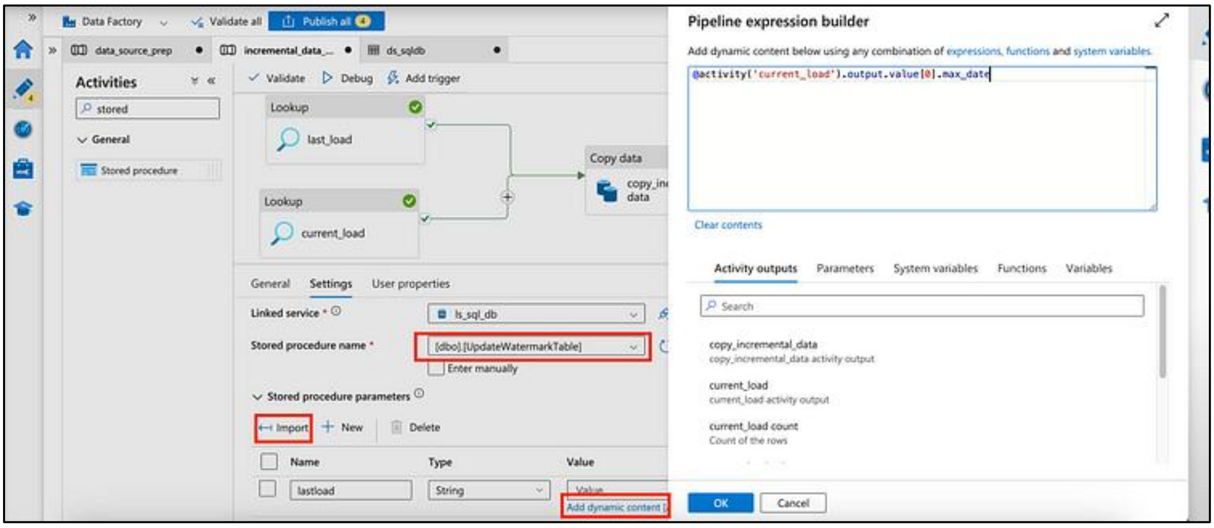
Then the last step in the sink is to set the properties, as follows:



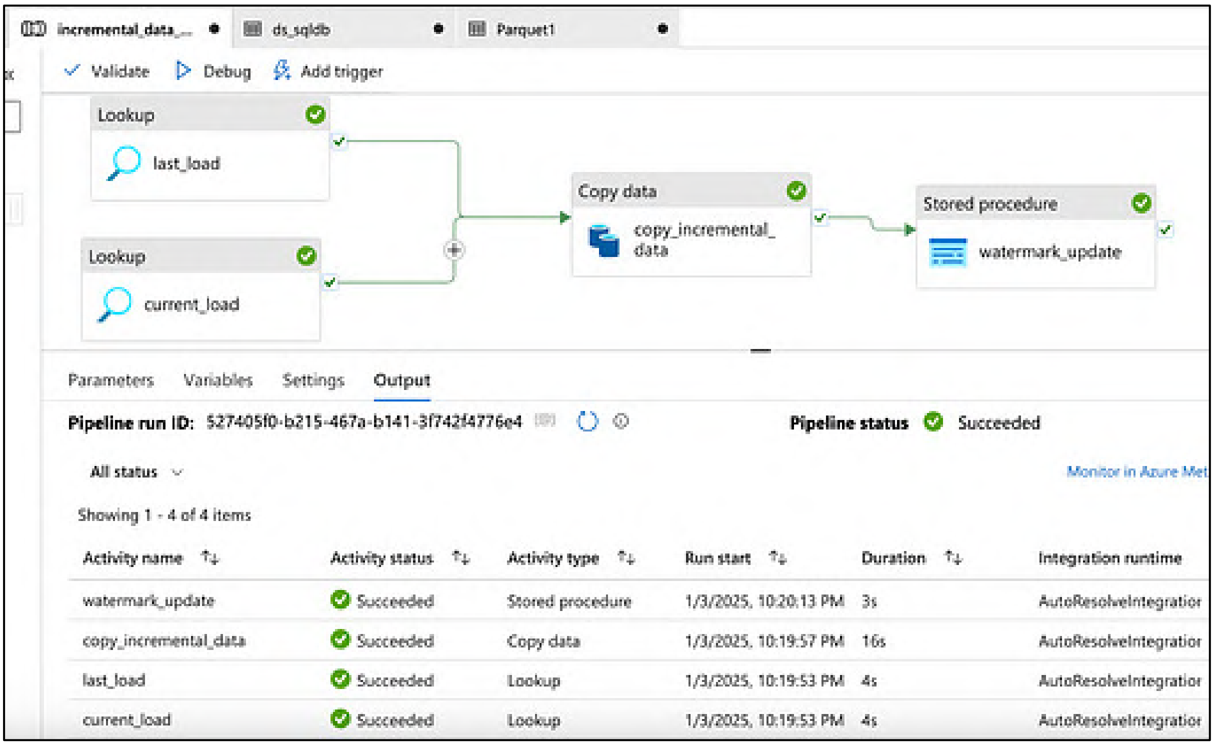
Then click on Debug to test the pipeline.

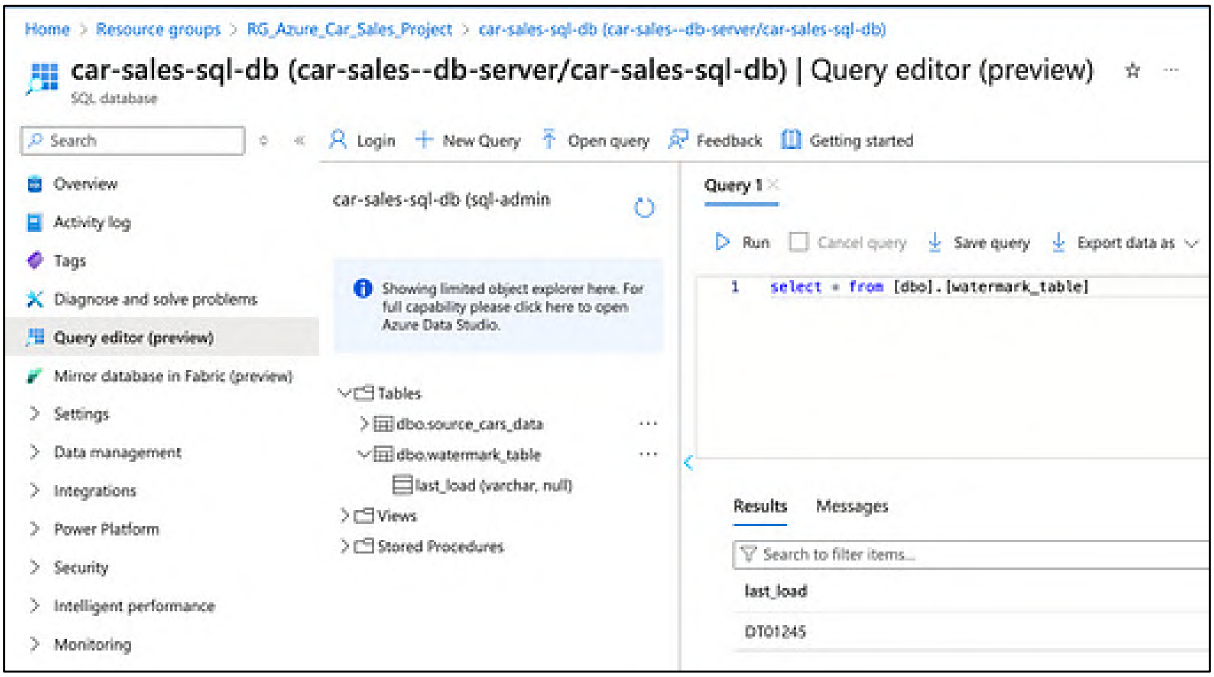


At this stage, we can move to the next step by adding a stored procedure activity to the pipeline to update the watermark_table with the last_load with the current_load (max_date) once the data is incremented.



After configuring the stored procedure activity, we debug the whole pipeline which dynamically increments data load.





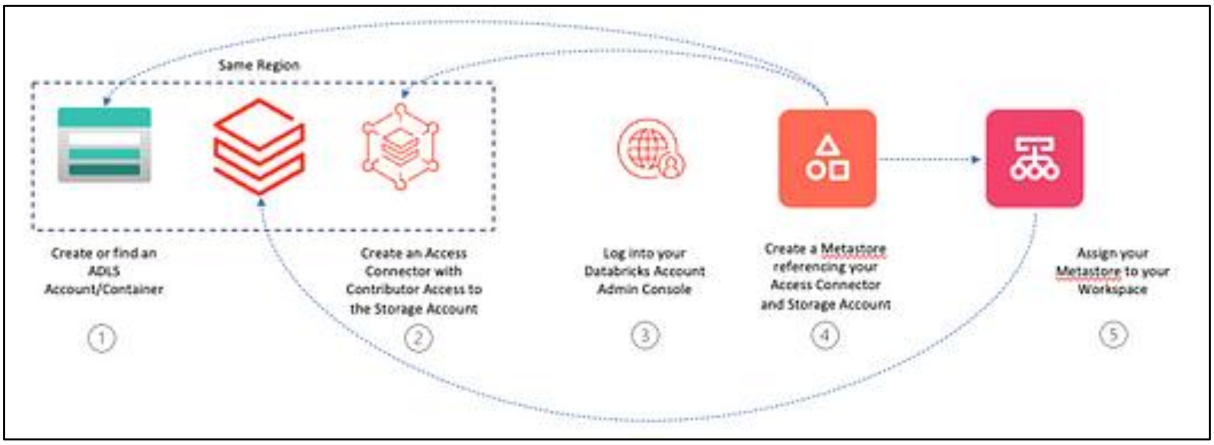
At last, make sure to publish the pipelines in order to save the work after we validated all the steps.

This was it for the first part of the project of ingesting the data from the source (GitHub) into Data Factory and creating a data pripline to dynamically load incremental data and saved to to the bronze database.

In this part of the project, we will focus on using Databricks to implement the Medallion Architecture which supports data quality by refining data incrementally at each layer. Bronze layer captures raw data, Silver layer cleans and transforms it and Gold layer aggregates and enriches it for business use.

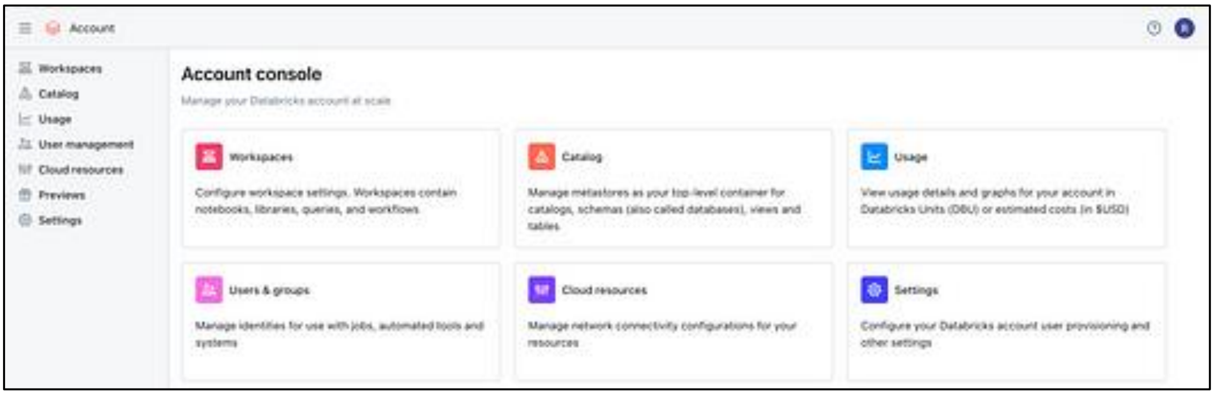
Step 1: Create a Unity Metastore

We will be following these steps:



To create Compute, we must attach the Databricks workspace to the Unity Catalog. But to be able to create a Unity Metastore, we need to do that from the admin console.

All you need to do is navigate to Azure > Microsoft Intra ID > users, copy the User principal name, and log in to the console https://accounts.azuredatabricks.net/ (by resetting the password).

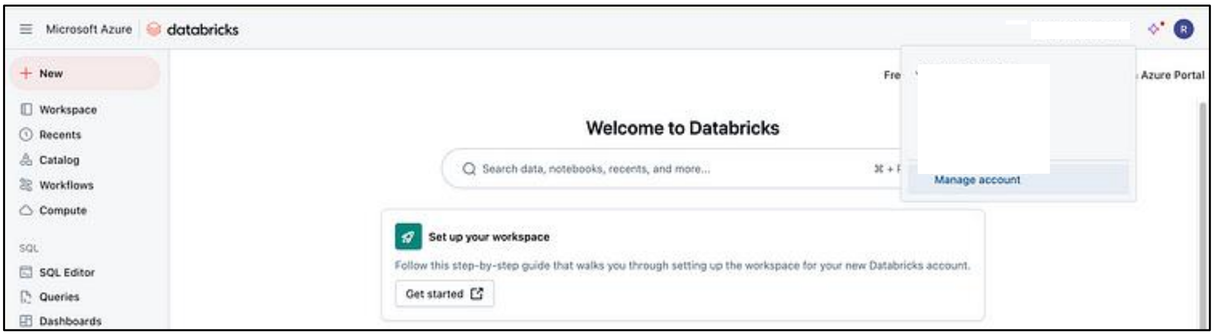


The Databricks admin console

Then all you need to do is assign the admin role to your email address which you used in your Azure account.

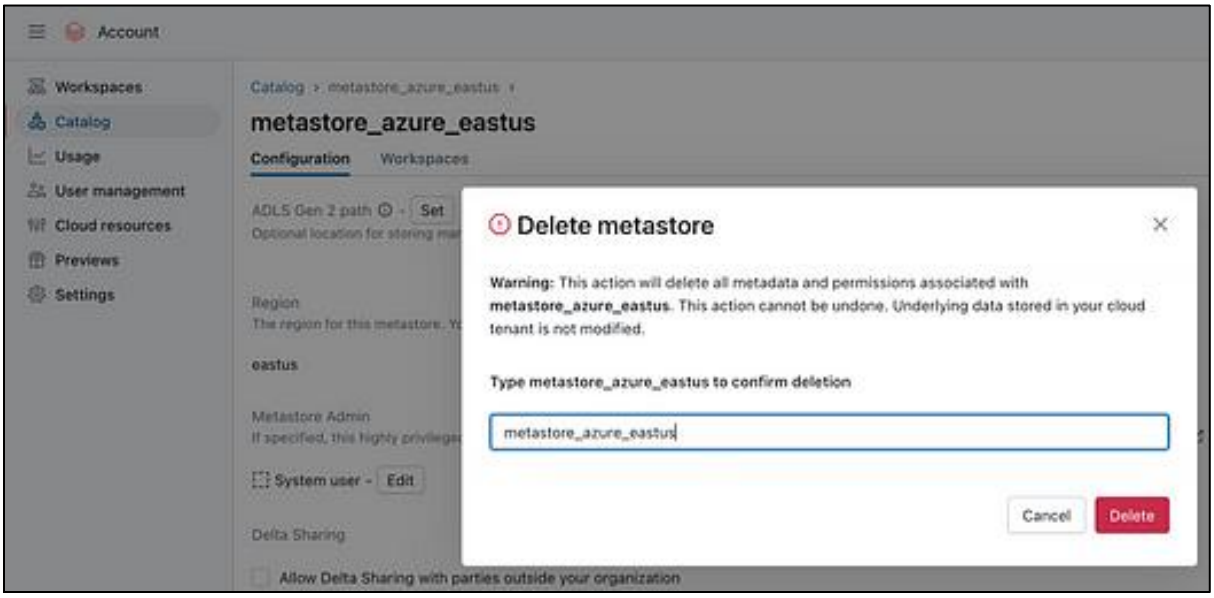
click in Account admin

Then go back to the Databricks workspace & refresh the page and you should see the 'Manage account' button.



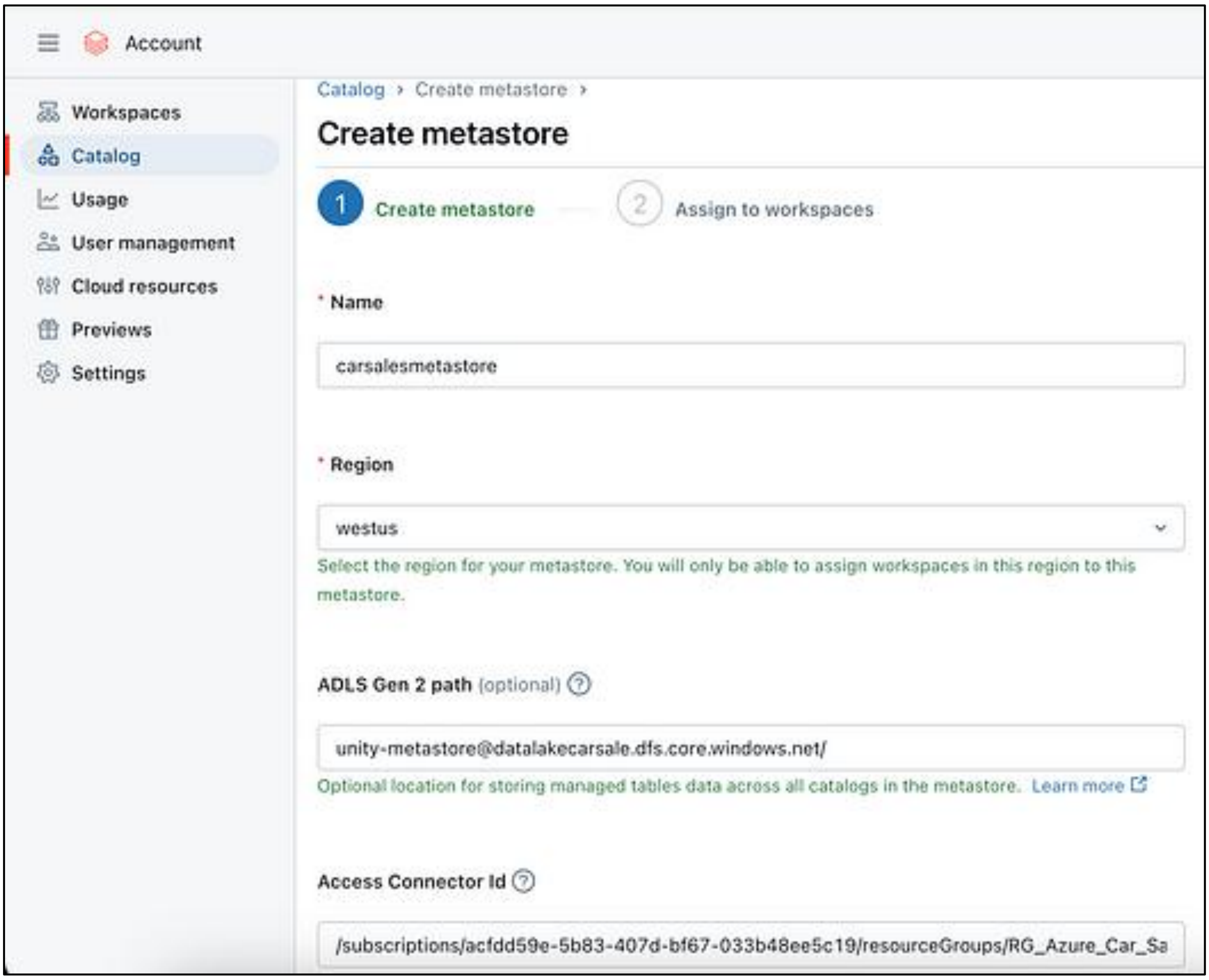
Notes to keep in mind:

- It is only possible to create one Metastore per region.
- Databricks creates default Metastores (to be deleted)



delete the default metastore

Now, in the Databricks admin console in the Catalog tab, click on Create metastore.

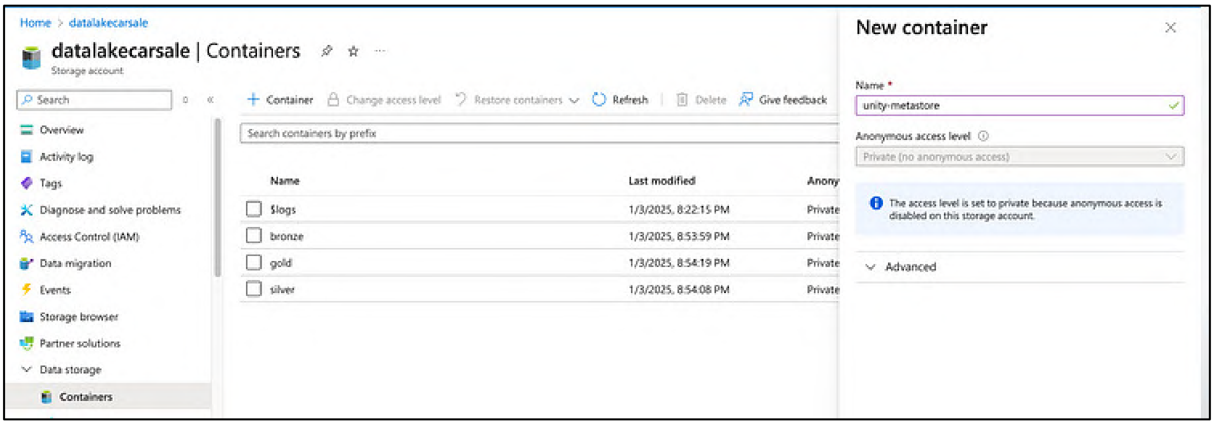


Add a name, select the region and provide the ADLS Gen 2 path (Azure Datalake Storage) following this convention:

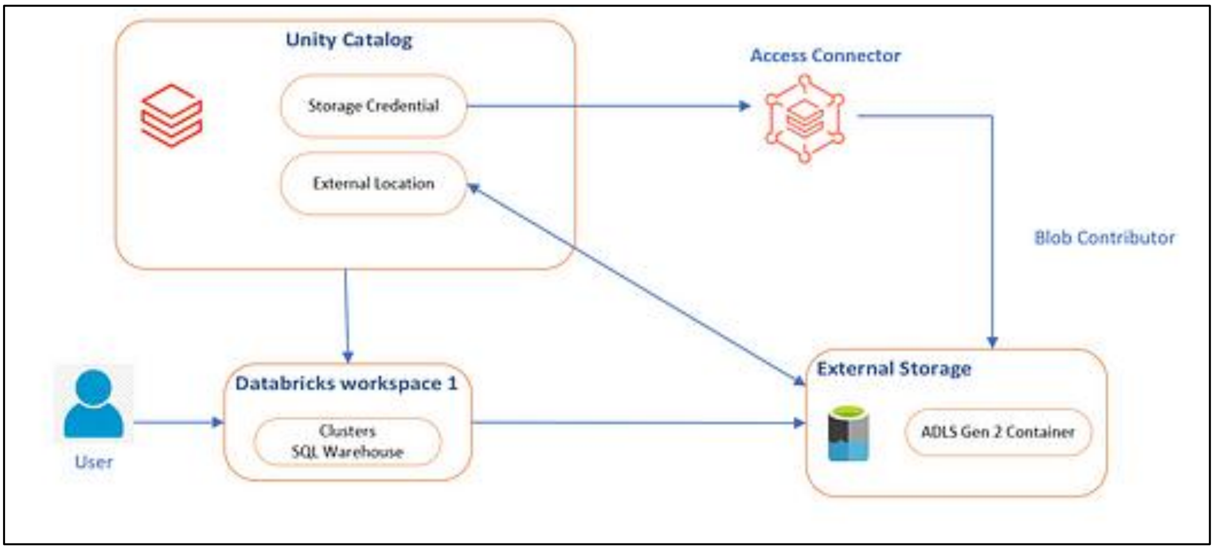
<container_name>@<storage_account_name>.dfs.core.windows.net/<path>

Example: unity-metastore@datalakecarsale.dfs.core.windows.net/

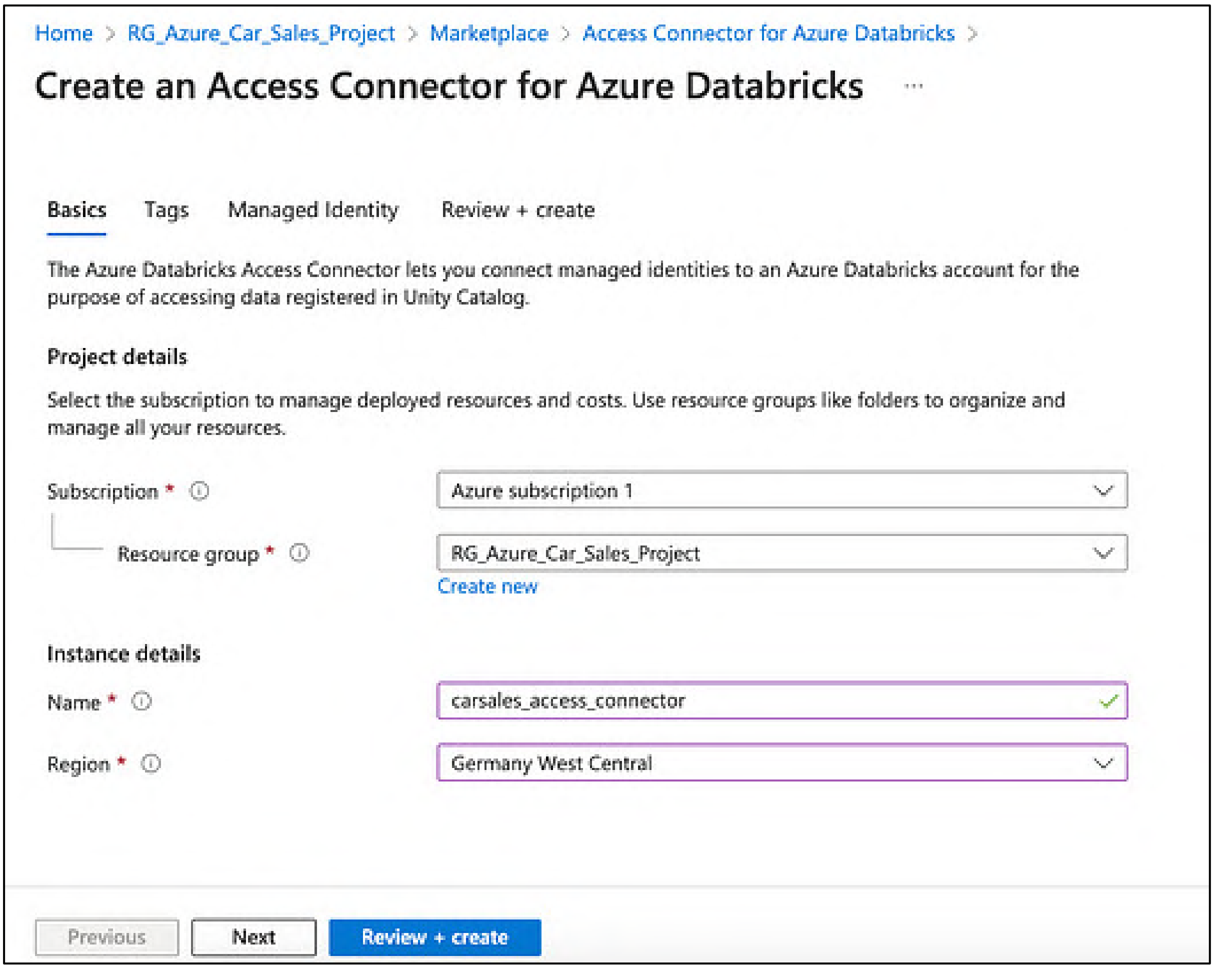
This storage account will be used to store the **default data e.g. metadata**. To create this ADLS storage, navigate to the Azure portal > our project resource group > account storage > containers



About the Access Connector ID which is required to create the metastore, we need to create a Databricks Access Connector which will connect the Databricks workspace and the ADLS Gen 2 storage (more detail in the below graph)

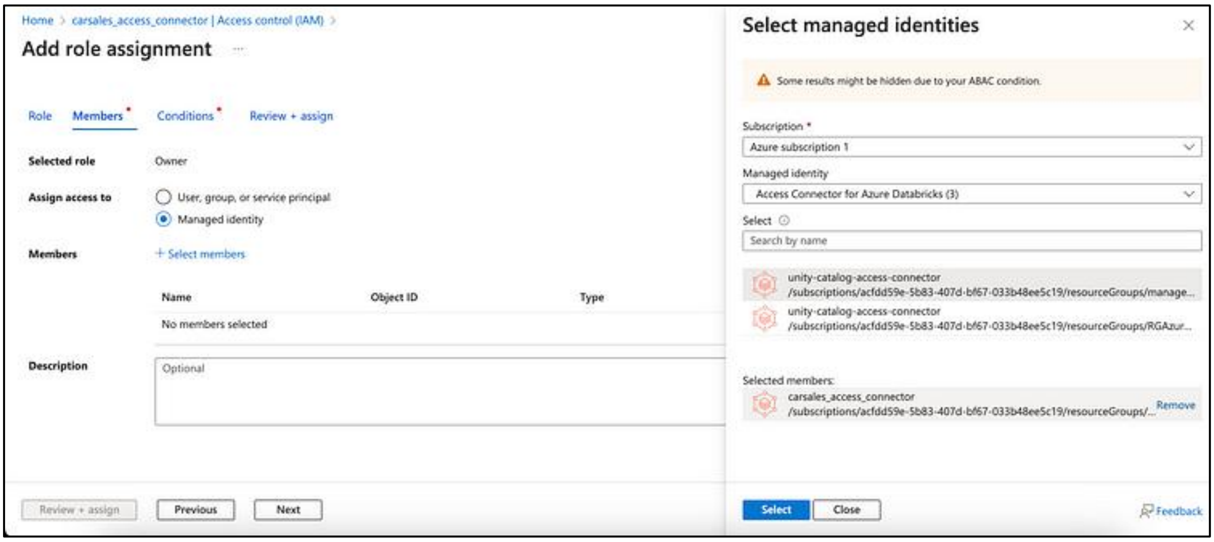


Then in the resource group, add the access connector for Databricks

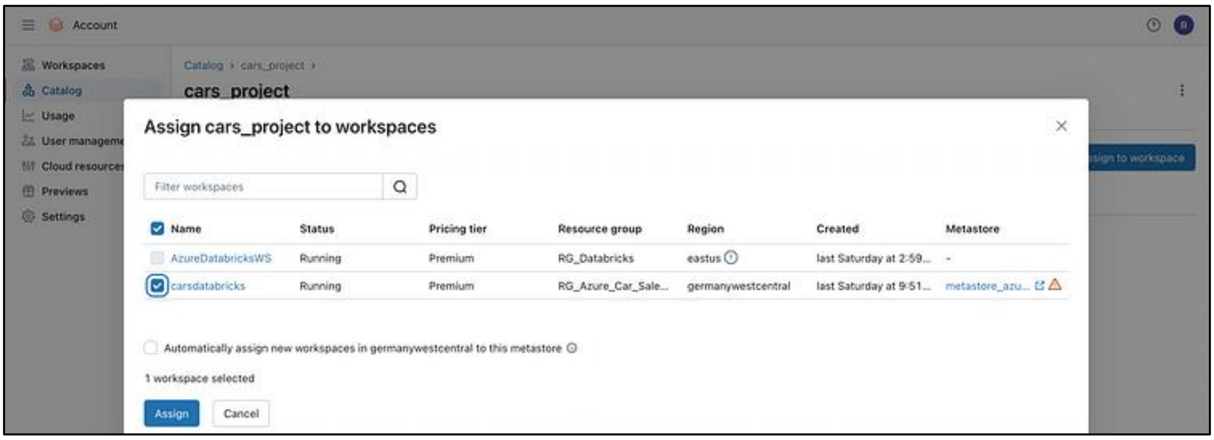


Then, we need to assign to the access connector the role of "storage blob contributor" to be able to contribute to the datalake (storage account). To do that, click on the access connector > Access Control (IAM) > Add role.

After configuring the role, move to assign the managed identity members as shown in the screenshot below:

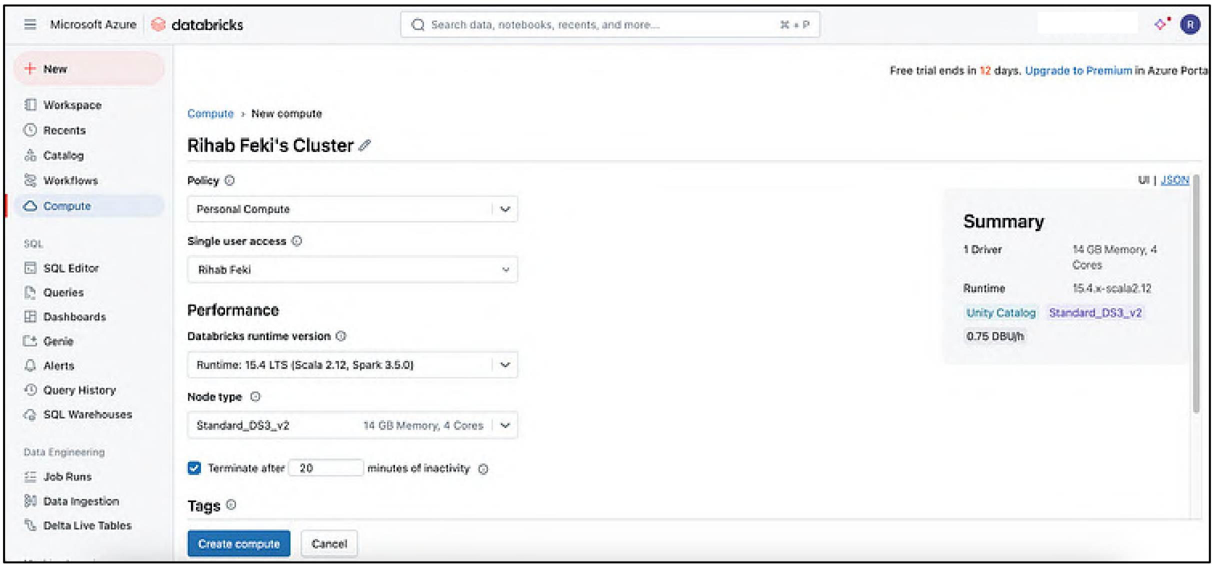


After creating the needed resources, finish filling the form to creating the metastore and Finally assign the Workspace to the metastore.



After completing this step, we successfully created a metastore and attached it to the Databricks workspace. Now we get back to the Databricks workspace and continue to create Compute.

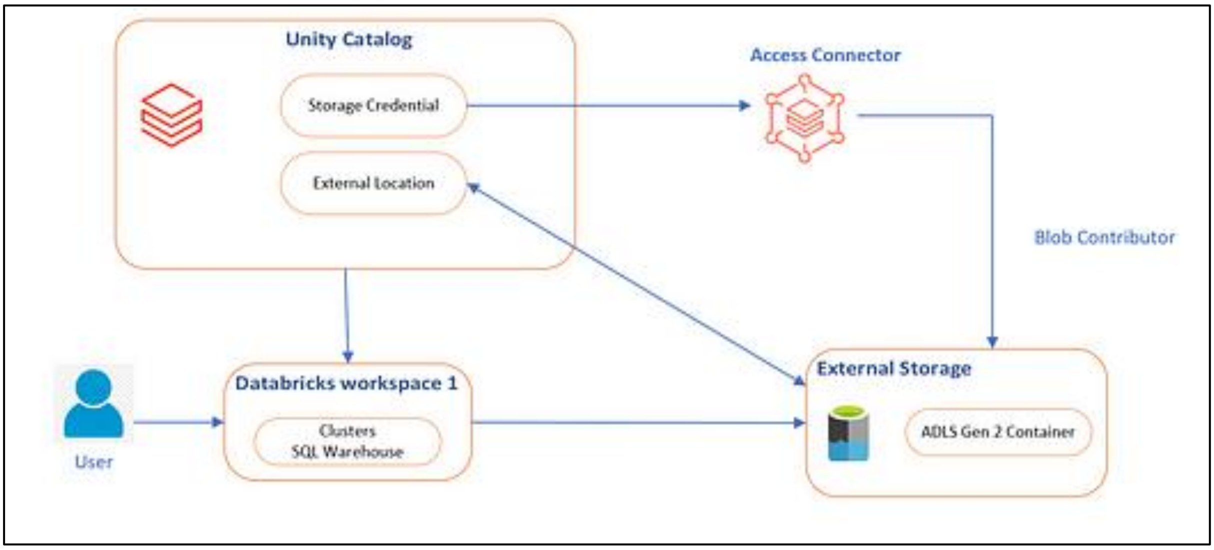
Step 2: Create Compute



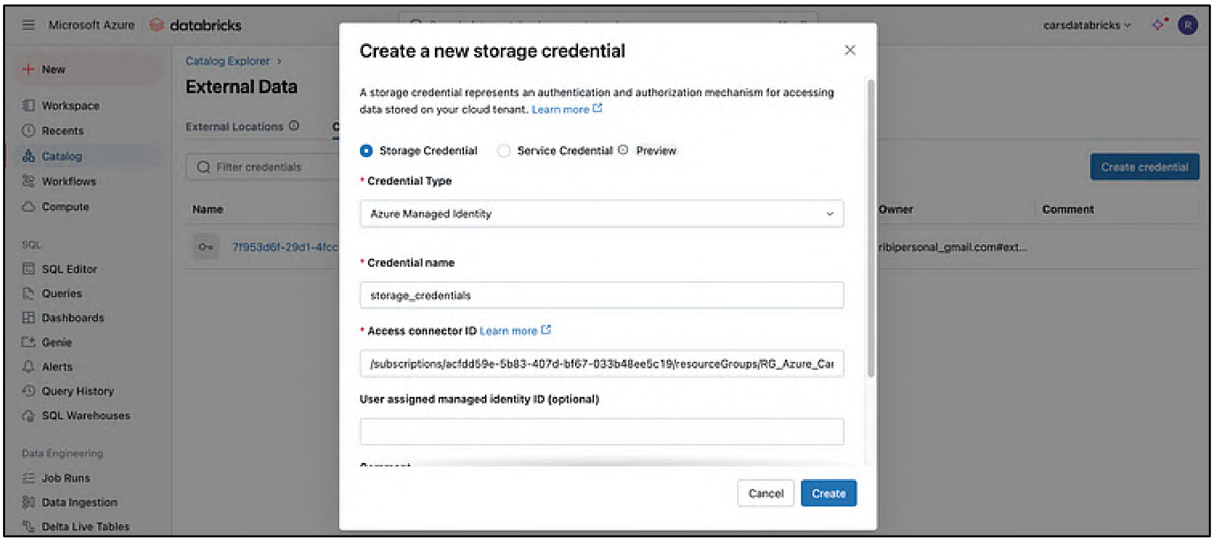
Step 3: Create External locations

At the current state, we have the raw data on Azure datalake in the bronze container, now, we need to create 3 External Locations (bronze, silver, and gold) because we need to read & write data between these containers, so we should have an external location for it.

To create an external location, we should have "storage credentials".

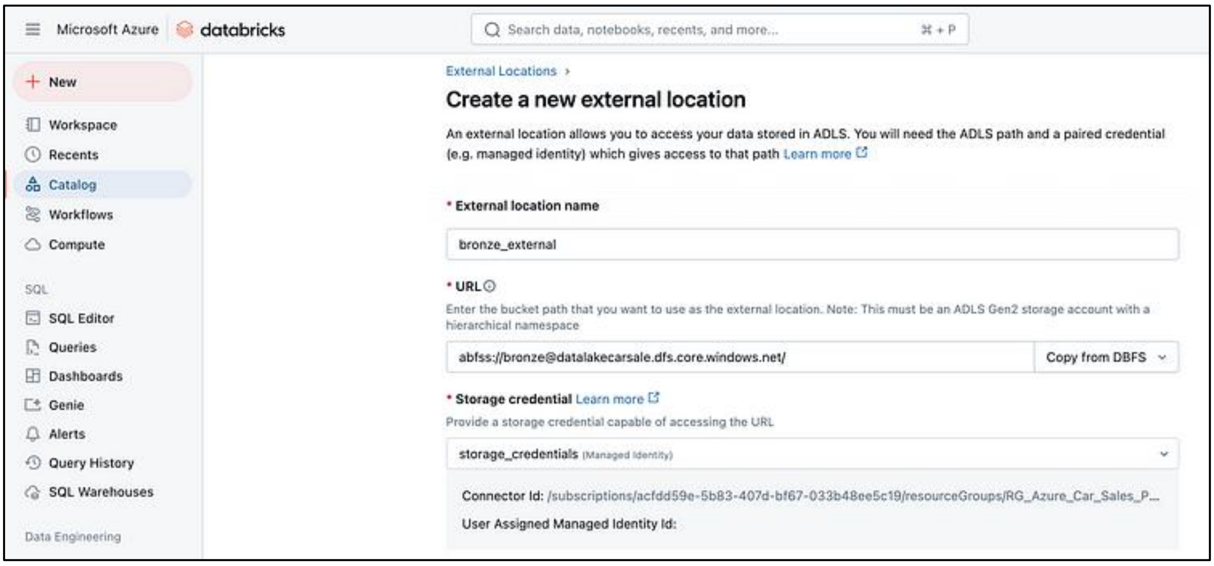


to create an External Location, you need to start by creating credentials. So, navigate to Databricks Workspace and click on Catalog > External Data > Credentials.



start by creating the credential, then the external storage

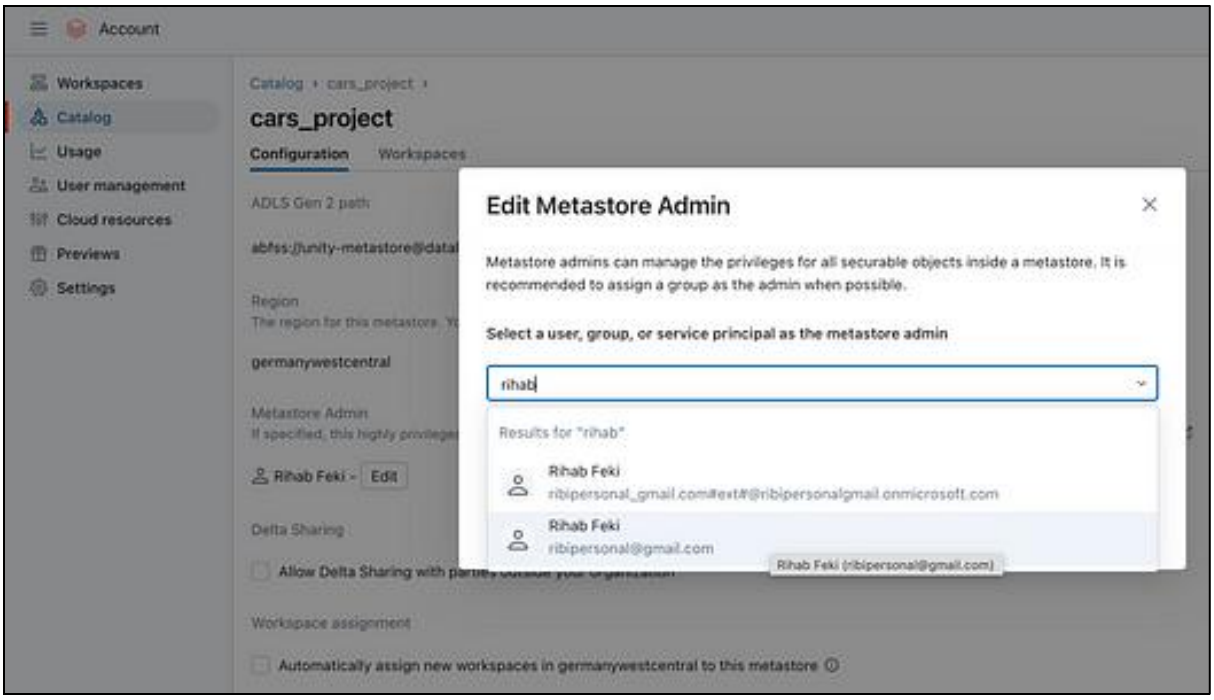
After creating the credentials, click again on Catalog > **External location** And provide a URL following this structure: **abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<path>**



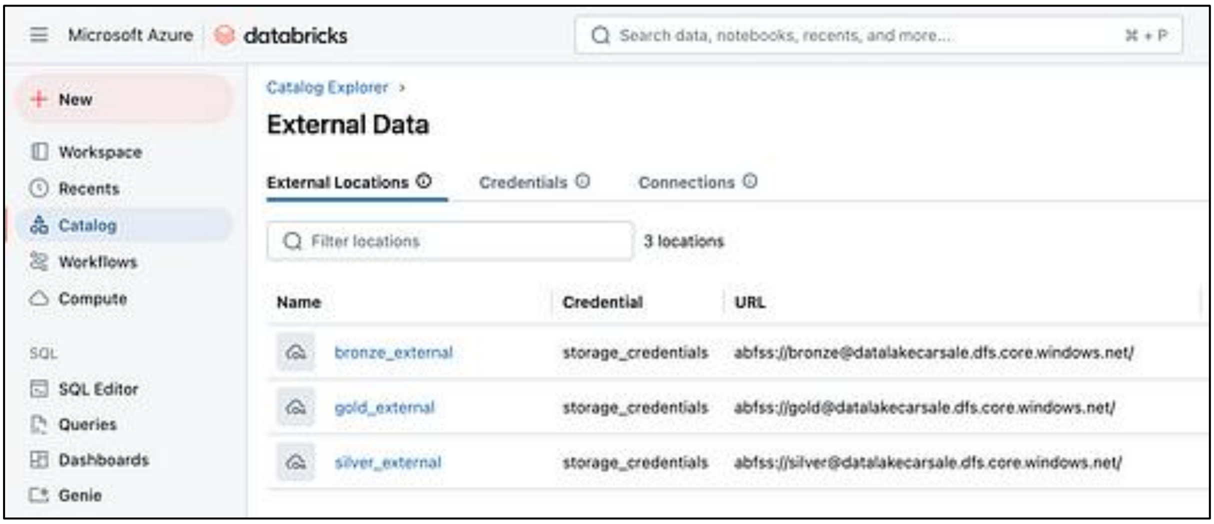
When clicking on create, you will have the following error message:

User does not have CREATE EXTERNAL LOCATION on Metastore 'cars_project'.

To fix it, go to the Databricks admin console (https://accounts.azuredatabricks.net/) and edit the Metastore admin to make it your user account (not the intra ID email)



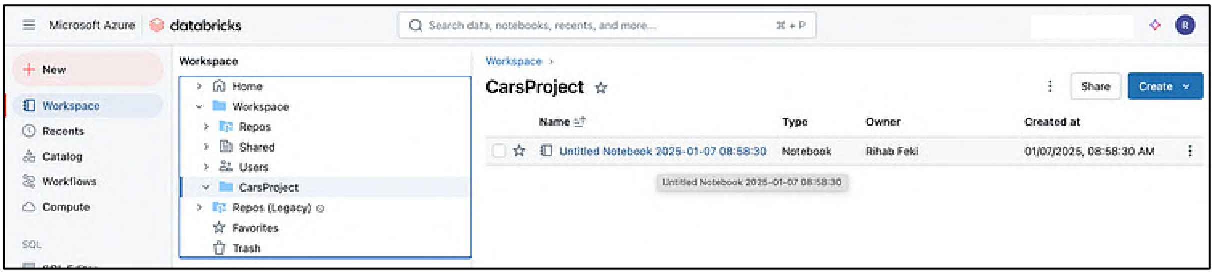
after creating the external location for the bronze layer, do the same work for the silver and gold layers.



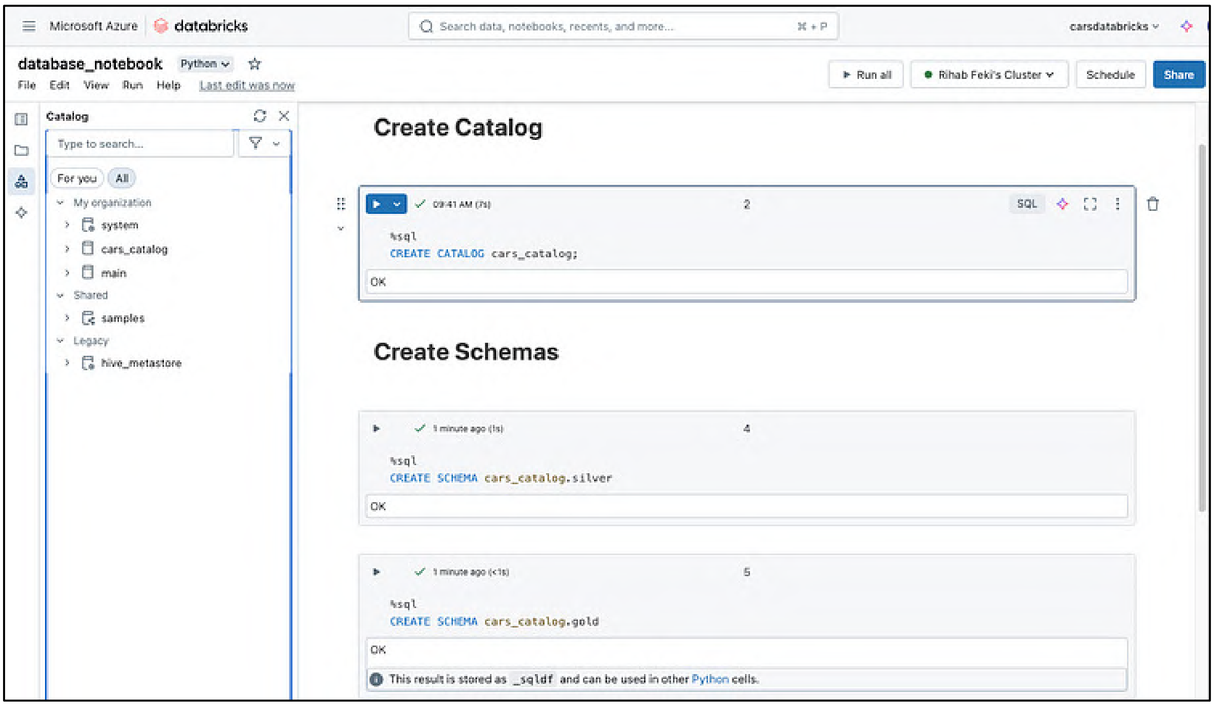
Now we are all set to pull the data from bronze, we need to apply transformation and we need to store that data in the silver container.

Step 4: Create a Workspace

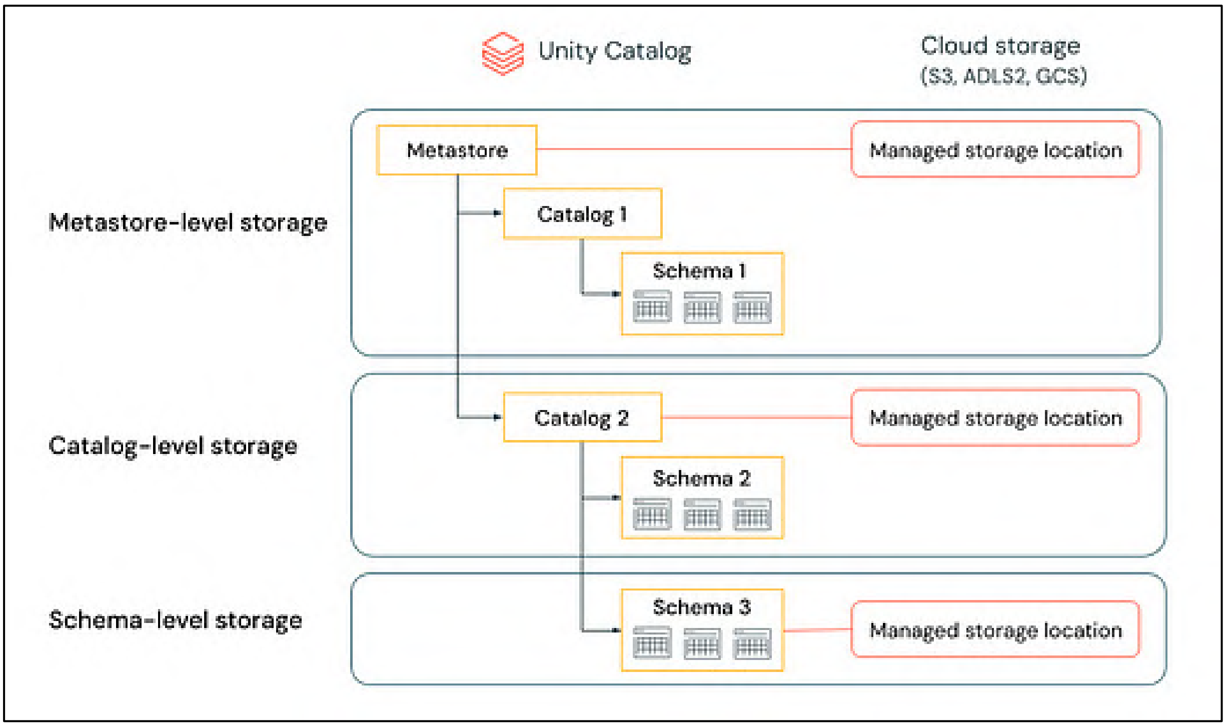
Click on Workspace in the sidebar, then in the Workspace folder, create a new project folder, and inside it create a Notebook to define the catalogs and schemas.



Within the notebook, we create one catalog and two schemas.



To better understand the concept of Unity Catalog hierarchical architecture, check the following graph:



- **Catalog**: Catalogs are the highest level in the data hierarchy (catalog > schema > table/view/volume) managed by the Unity Catalog metastore. They are intended as the primary unit of data isolation in a typical Databricks data governance model.
- **Schema**: Schemas, or databases, are logical groupings of tabular data (tables and views), non-tabular data (volumes), functions, and machine learning models.
- **Tables**: They contain rows of data.

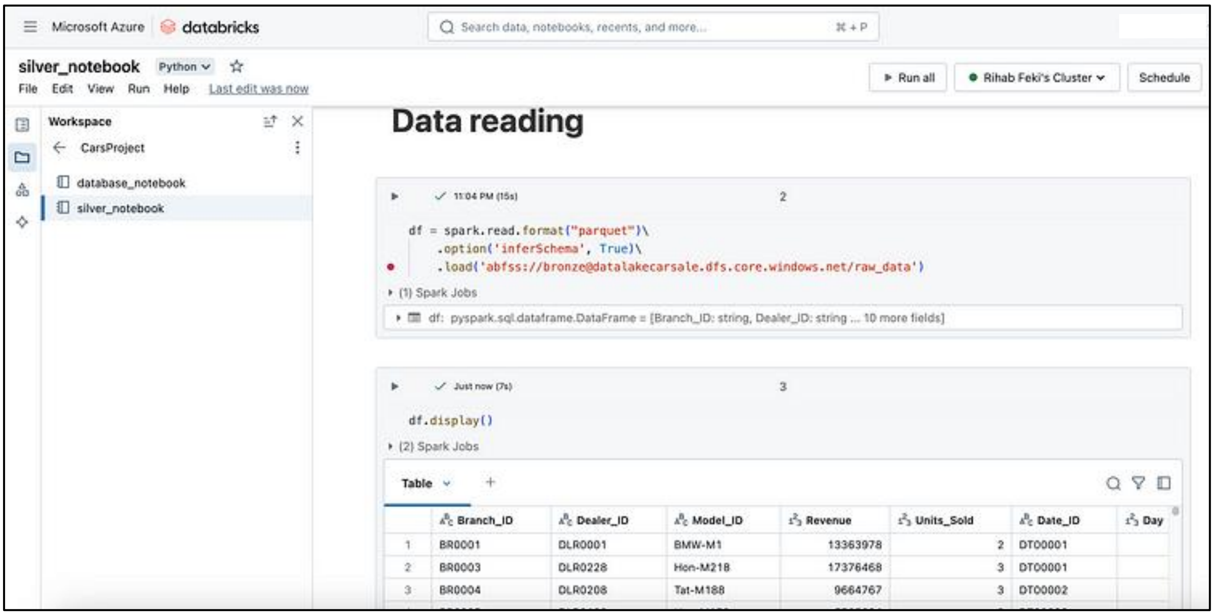
Unity Catalog allows the creation of managed and external tables.

- **Managed tables** are fully managed by Unity Catalog, including their lifecycle and storage
- **External tables** rely on cloud providers for data management, making them suitable for integrating existing datasets and allowing write access from outside Databricks.

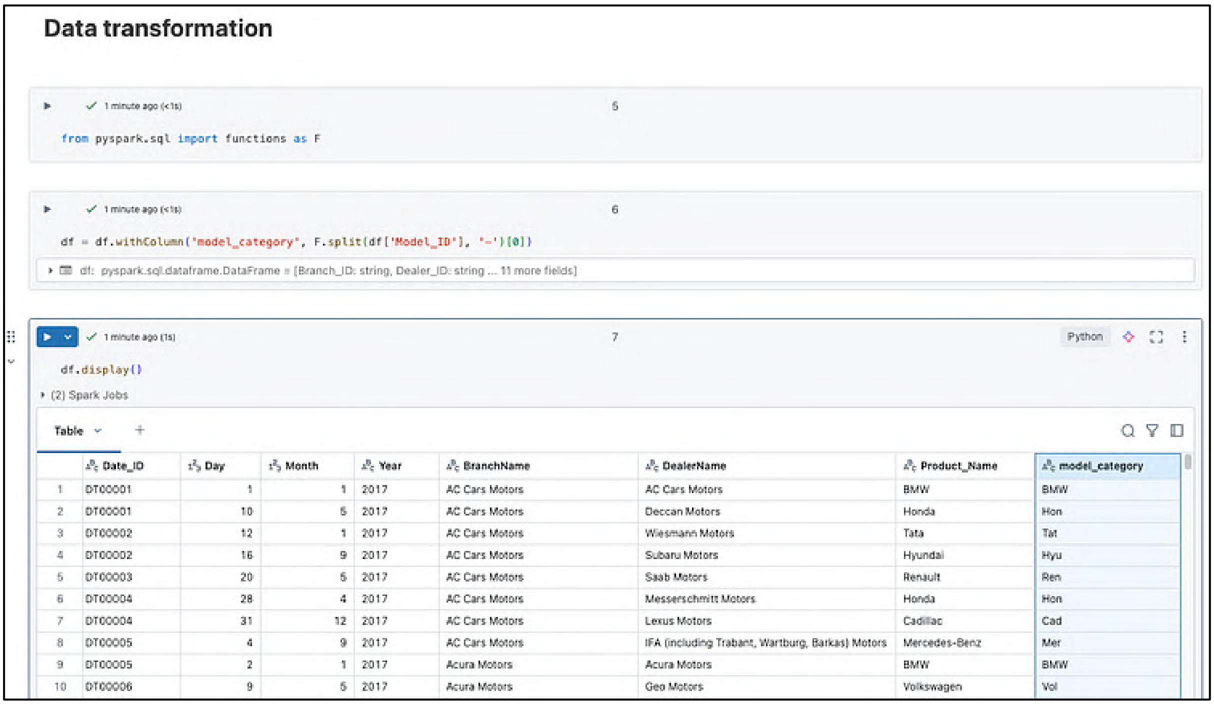
Now after creating the first Notebook in which we created the catalog and the schemas, we create a second Notebook to read the data & transform it.

Step 5: Silver layer — data transformation (one big table)

We will use PySpark API to read the data and one thing to note here is the 'inferSchema' option which helps to derive the schema from the raw data in parquet file format.



Then, after reading the dataset, we will do some column transformation, to split the Model_ID and make the part before the '-' as model_category.



Then we created an additional column to calculate the revenue per unit this can be useful for the analytics.

Just now (<1s) 8

```
df = df.withColumn('revenue_per_unit', df['Revenue']/df['Units_Sold'])
```

df: pyspark.sql.dataframe.DataFrame = [Branch_ID: string, Dealer_ID: string ... 12 more fields]

Just now (1s) 9

```
df.display()
```

(2) Spark Jobs

Table

	DealerName	Product_Name	model_category	revenue_per_unit
1	AC Cars Motors	BMW	BMW	6681989
2	Deccan Motors	Honda	Hon	5792156
3	Wiesmann Motors	Tata	Tat	3221589
4	Subaru Motors	Hyundai	Hyu	1841768
5	Saab Motors	Renault	Ren	4323696
6	Messerschmitt Motors	Honda	Hon	7321228

withColumn will create a new column if the name does not exists, if if does it will modify the column

AD-HOC analysis (data aggregation)

How many units were sold of each branch every year. To know which branch is doing good and which is doing bad.

2 minutes ago (4s) 11 Python

```
from pyspark.sql.functions import sum as F_sum

df.groupBy('Year', 'BranchName').agg(
    F_sum('Units_Sold').alias('Total_Units_Sold')
).sort('Year', 'Total_Units_Sold', ascending=[True, False]).display()
```

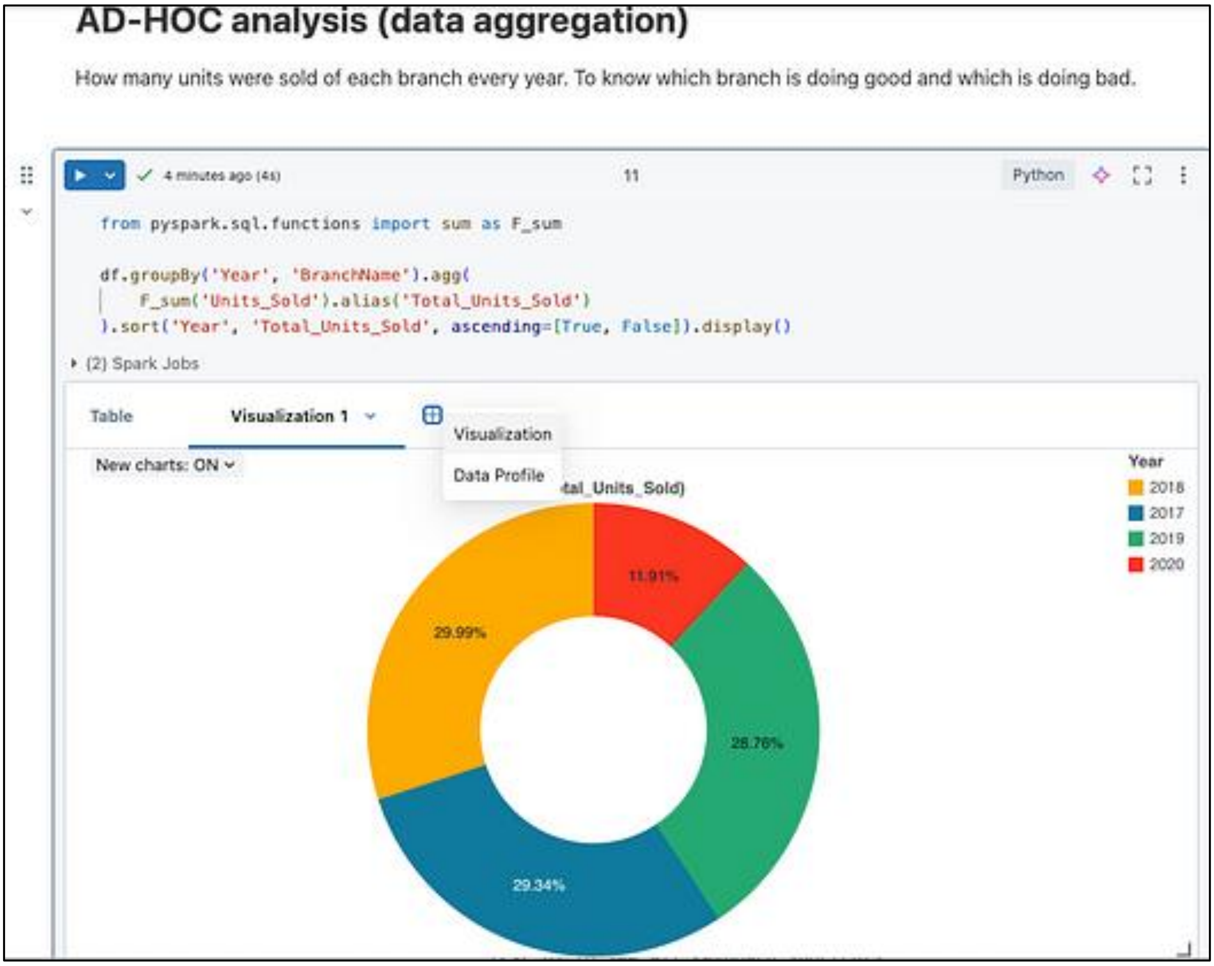
Generate (⌘ + I)

(2) Spark Jobs

Table

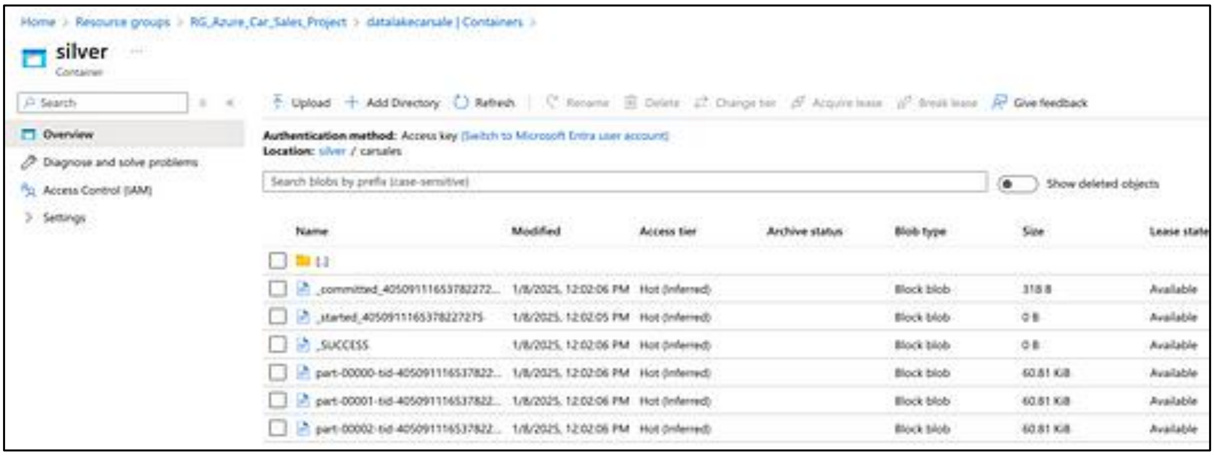
	Year	BranchName	Total_Units_Sold
1	2017	Alpine Motors	72
2	2017	Aston Martin Motors	69
3	2017	Bristol Motors	69
4	2017	Acura Motors	69
5	2017	BMW Motors	69
6	2017	Ariel Motors	63
7	2017	Gilbern Motors	63

You can also create visualizations by clicking on the + button near Table.



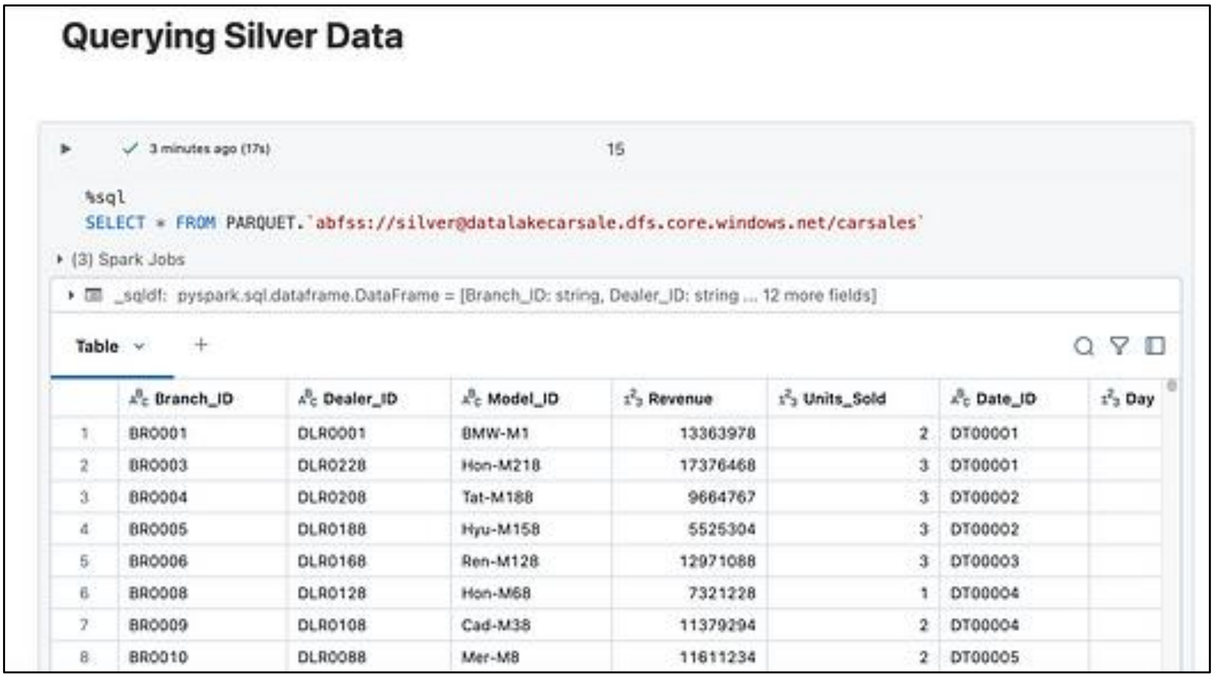
Then we write the transformed data to the silver storage container

Then check that the parquet files were saved on Azure.



To query the data, we can use SQL:

SELECT * FROM 'abfss://<container>@<storageaccount>.dfs.core.windows.net/<path>/<file>'



Querying Silver Data

3 minutes ago (17s) 15

```
%sql
SELECT * FROM PARQUET.`abfss://silver@datalakecarsale.dfs.core.windows.net/carsales`
```

(3) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [Branch_ID: string, Dealer_ID: string ... 12 more fields]

Table

	Branch_ID	Dealer_ID	Model_ID	Revenue	Units_Sold	Date_ID	Day
1	BR0001	DLR0001	BMW-M1	13363978	2	DT00001	
2	BR0003	DLR0228	Hon-M218	17376468	3	DT00001	
3	BR0004	DLR0208	Tat-M188	9664767	3	DT00002	
4	BR0005	DLR0188	Hyu-M158	5525304	3	DT00002	
5	BR0006	DLR0168	Ren-M128	12971088	3	DT00003	
6	BR0008	DLR0128	Hon-M68	7321228	1	DT00004	
7	BR0009	DLR0108	Cad-M38	11379294	2	DT00004	
8	BR0010	DLR0088	Mer-M8	11611234	2	DT00005	

Step 6: Gold layer (Dimension Model)

The main goal of transitioning data from the silver to the gold layer is to prepare data for high-level business intelligence and reporting. This involves modeling the data e.g. following the start schema, to ensure it is ready for consumption by end-users, analysts, and decision-makers.

Silver layer: doesn't maintain historical changes — it's more about reflecting the current, cleaned state of incoming data.

Gold layer: Moving to the Gold layer with a focus on dimensional modeling and implementing SCD, the strategy needs to capture and store historical changes for analysis*.*

Slowly Changing Dimension — Type 1

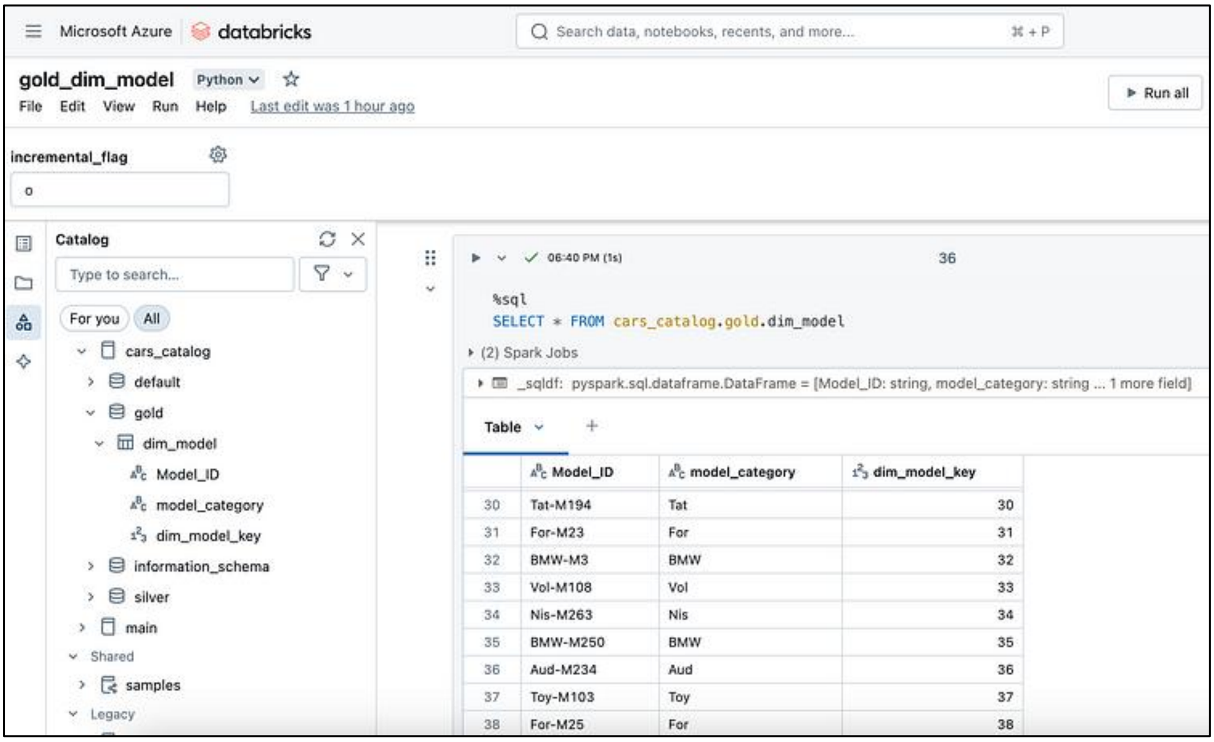
In this part of the tutorial, we will dive into the steps to implement the incremental data update of the dim_model table to create the dimension in the gold layer.

A detailed step-by-step guide is in this Databricks Notebook (PySpark)

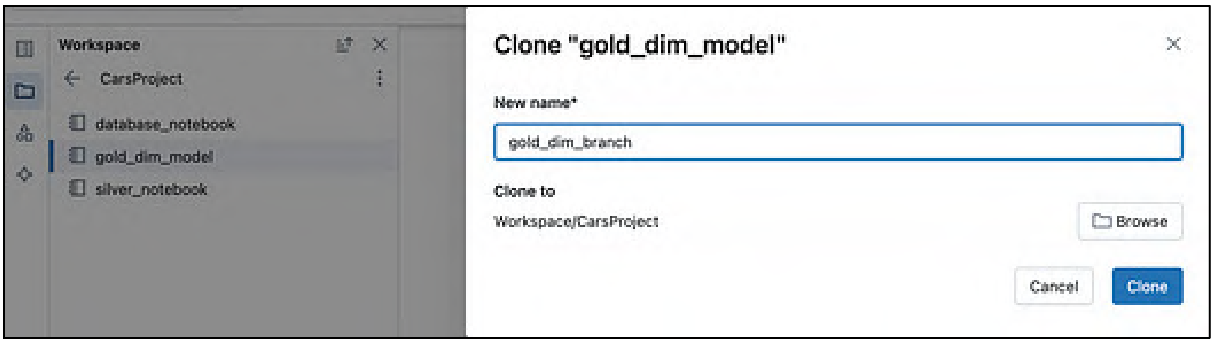
One of the most important functions is the following:

```
# Incremental RUN
if spark.catalog.tableExists('cars_catalog.gold.dim_model'):
  delta_table = DeltaTable.forPath(spark,
"abfss://gold@datalakecarsale.dfs.core.windows.net/dim_model")
  # update when the value exists
  # insert when new value
  delta_table.alias("target").merge(df_final.alias("source"), "target.dim_model_key =
source.dim_model_key")\
    .whenMatchedUpdateAll()\
    .whenNotMatchedInsertAll()\
    .execute()

# Initial RUN
else: # no table exists
  df_final.write.format("delta")\
    .mode("overwrite")\
    .option("path", "abfss://gold@datalakecarsale.dfs.core.windows.net/dim_model")\
    .saveAsTable("cars_catalog.gold.dim_model")
```

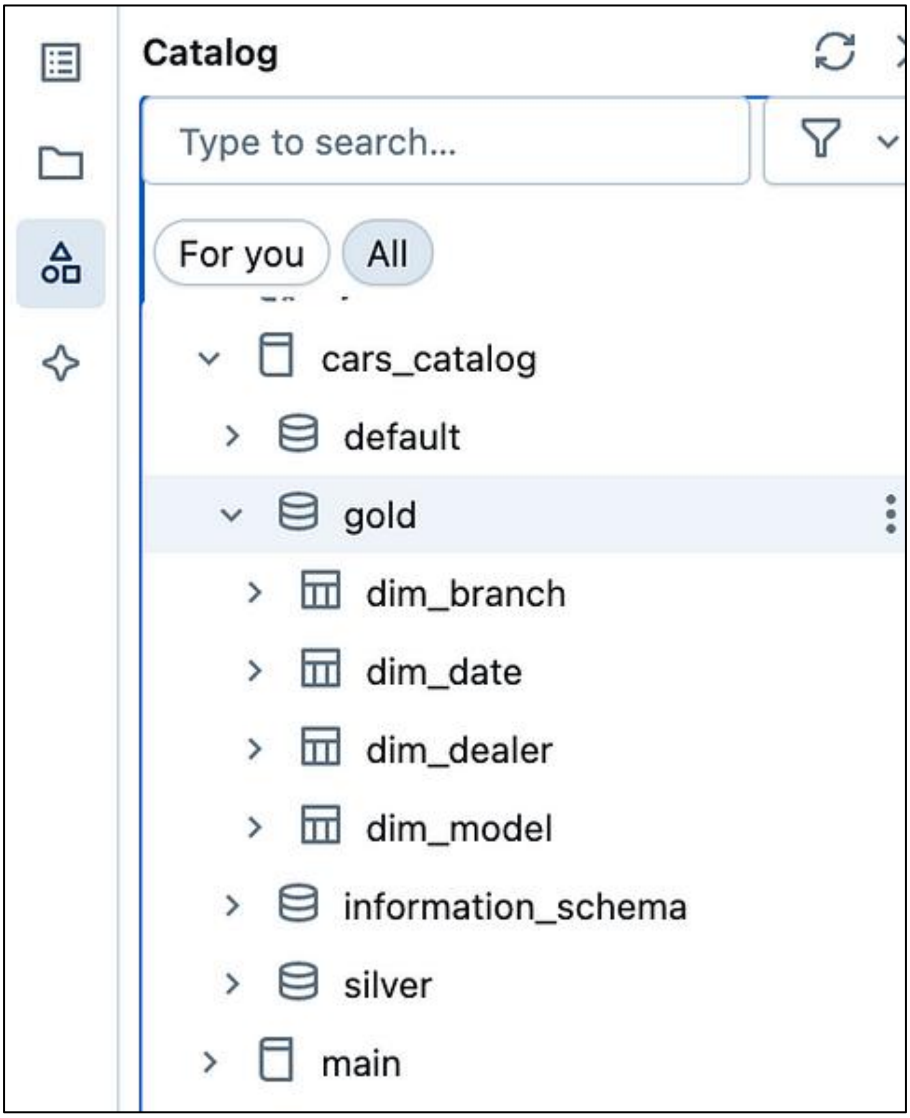
The final result should look like this in the dimension table:



Then to create the rest of the dimensions, you can simply clone the same notebook and just rename it with the new dimension name, and make the necessary changes, like the relative columns and table name.

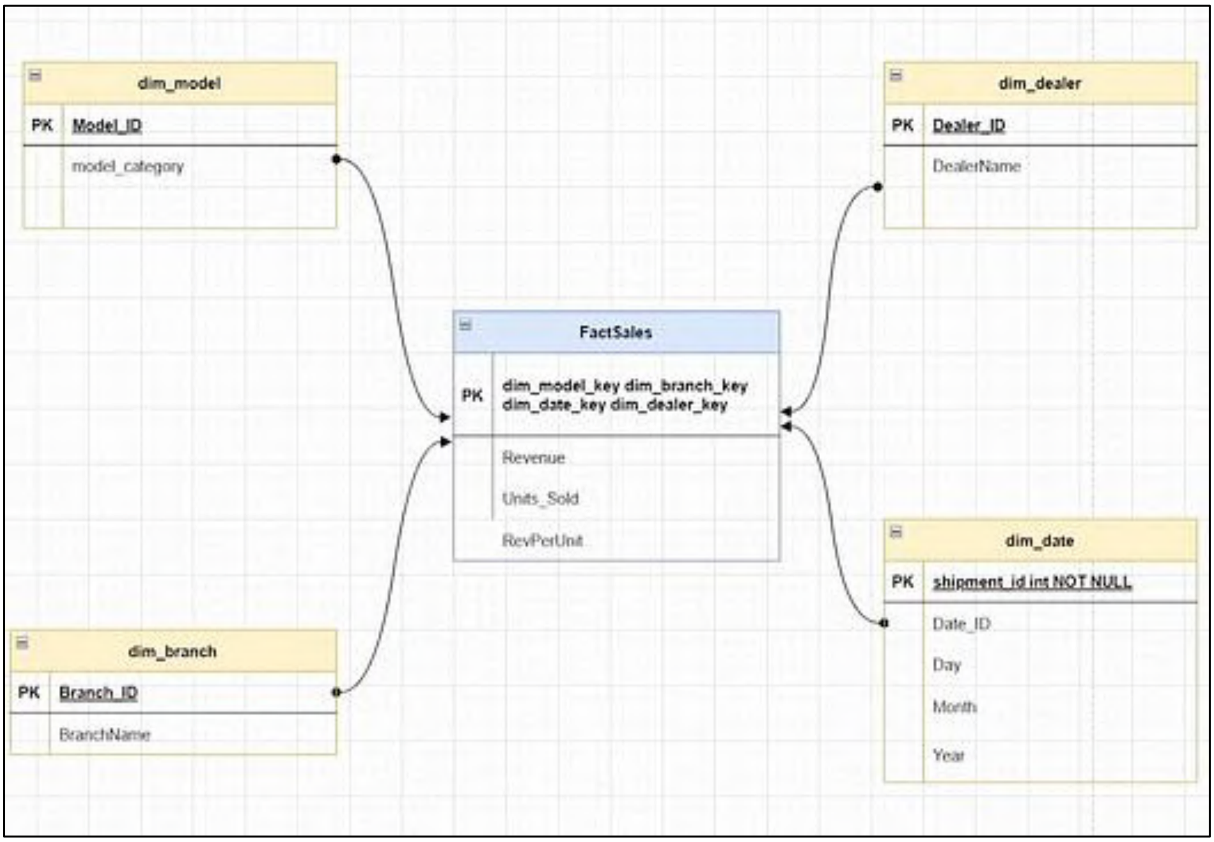


The process repeats for all the dimensions which are the dim_branch and dim_dealer.



Step 7: Create the Fact Table

In data warehousing, a fact table consists of a business process's measurements, metrics, or facts. It is located at the center of a star schema or a snowflake schema surrounded by dimension tables.



The Fact Table is created after the Dim tables are made. So the first step is reading the Revenue, Units_Sold, and RevPerUnit from the silver layer and then joining the Dim tables with the fact table. Then, we add the keys to the created dimensions.

The following is the code snippet to make the left join of the fact table with the dimension tables and also to bring the rest of the columns from the silver table and the surrogate keys we created for the dimensions.

```
df_fact = df_silver.join(df_branch, df_silver.Branch_ID==df_branch.Branch_ID, how='left') \
  .join(df_dealer, df_silver.Dealer_ID==df_dealer.Dealer_ID, how='left') \
  .join(df_model, df_silver.Model_ID==df_model.Model_ID, how='left') \
  .join(df_date, df_silver.Date_ID==df_date.Date_ID, how='left')\
  .select(df_silver.Revenue, df_silver.Units_Sold, df_branch.dim_branch_key,
  df_dealer.dim_dealer_key, df_model.dim_model_key, df_date.dim_date_key)
```

and then we need to write the resultant fact sales table in the gold layer, using this code snippet:

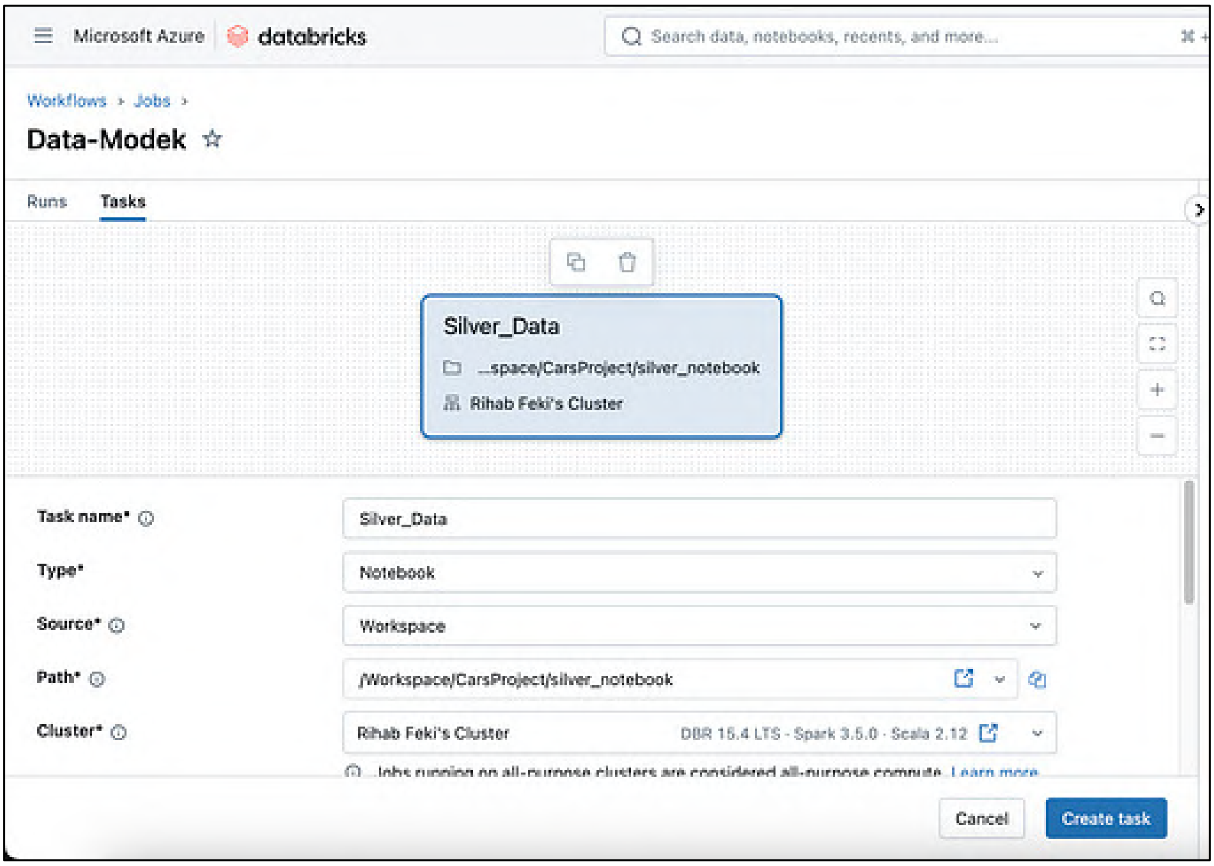
```
if spark.catalog.tableExists('factsales'):
  deltatable = DeltaTable.forName(spark, 'cars_catalog.gold.factsales')

  deltatable.alias('trg').merge(df_fact.alias('src'), 'trg.dim_branch_key = src.dim_branch_key
and trg.dim_dealer_key = src.dim_dealer_key and trg.dim_model_key = src.dim_model_key and
trg.dim_date_key = src.dim_date_key')\
    .whenMatchedUpdateAll()\
    .whenNotMatchedInsertAll()\
    .execute()
```

```
else:
  df_fact.write.format('delta')\
      .mode('Overwrite')\
      .option("path", "abfss://gold@datalakecarsale.dfs.core.windows.net/factsales")\
      .saveAsTable('cars_catalog.gold.factsales')
```

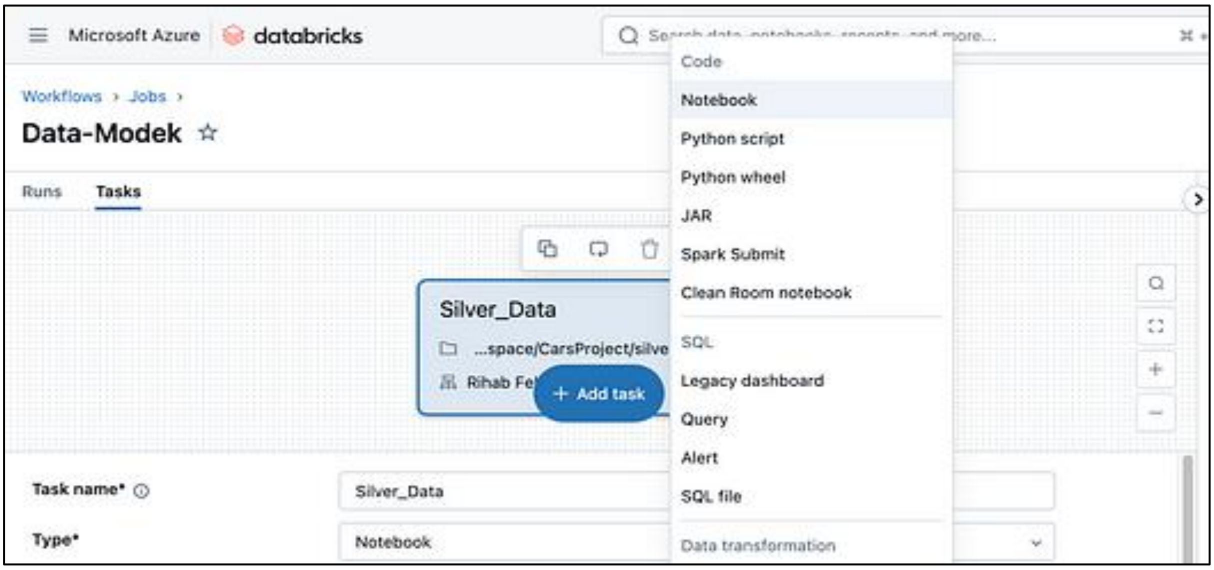
Step 8: Databricks Workflows (End-to-end Pipeline)

We can automate this whole pipeline with Azure Data Factory, but we will opt for using Databricks.

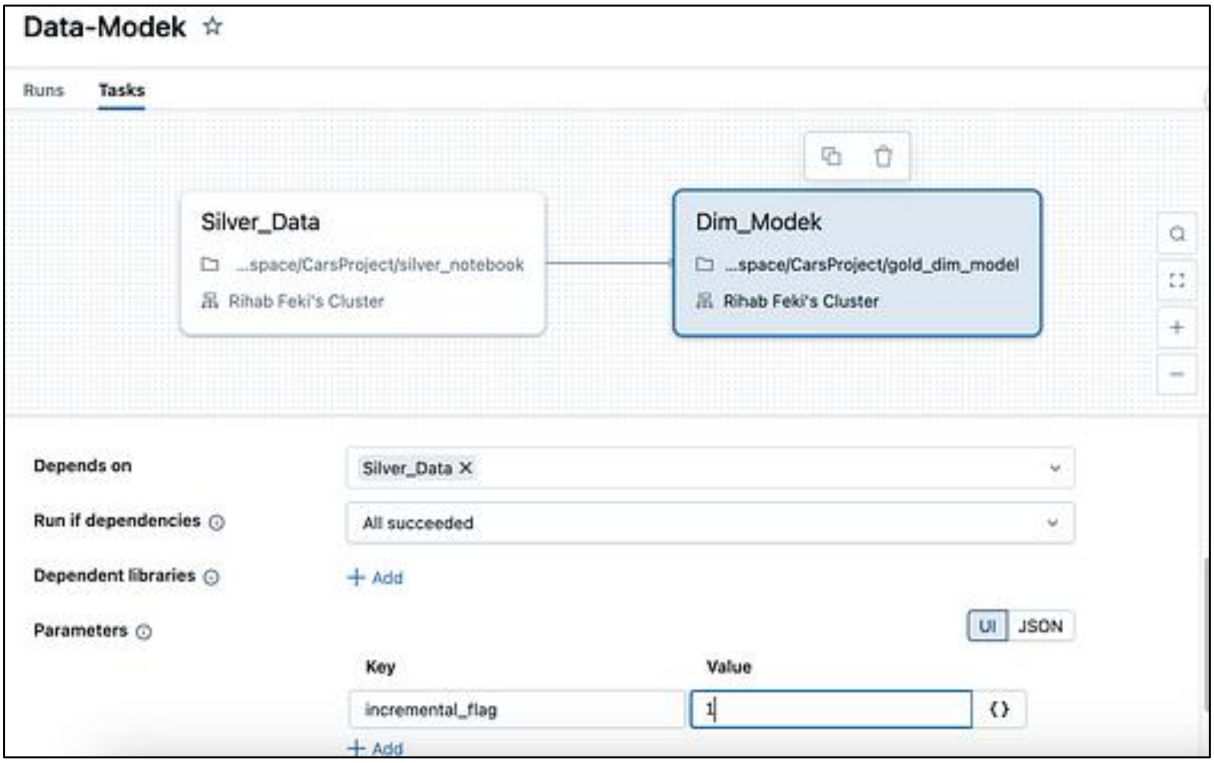
To do that, navigate to Workflows on Databricks workspace and click on ‘create job’ and then fill in the needed info as shown below attach the silver_notebook and the cluster, and finally click on create task.



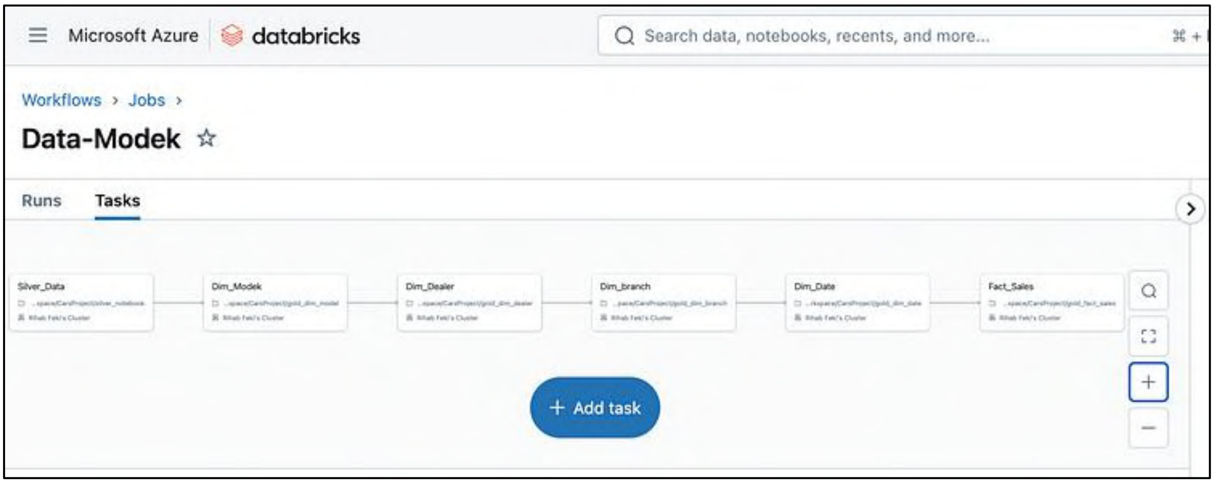
Add more tasks in this manner:



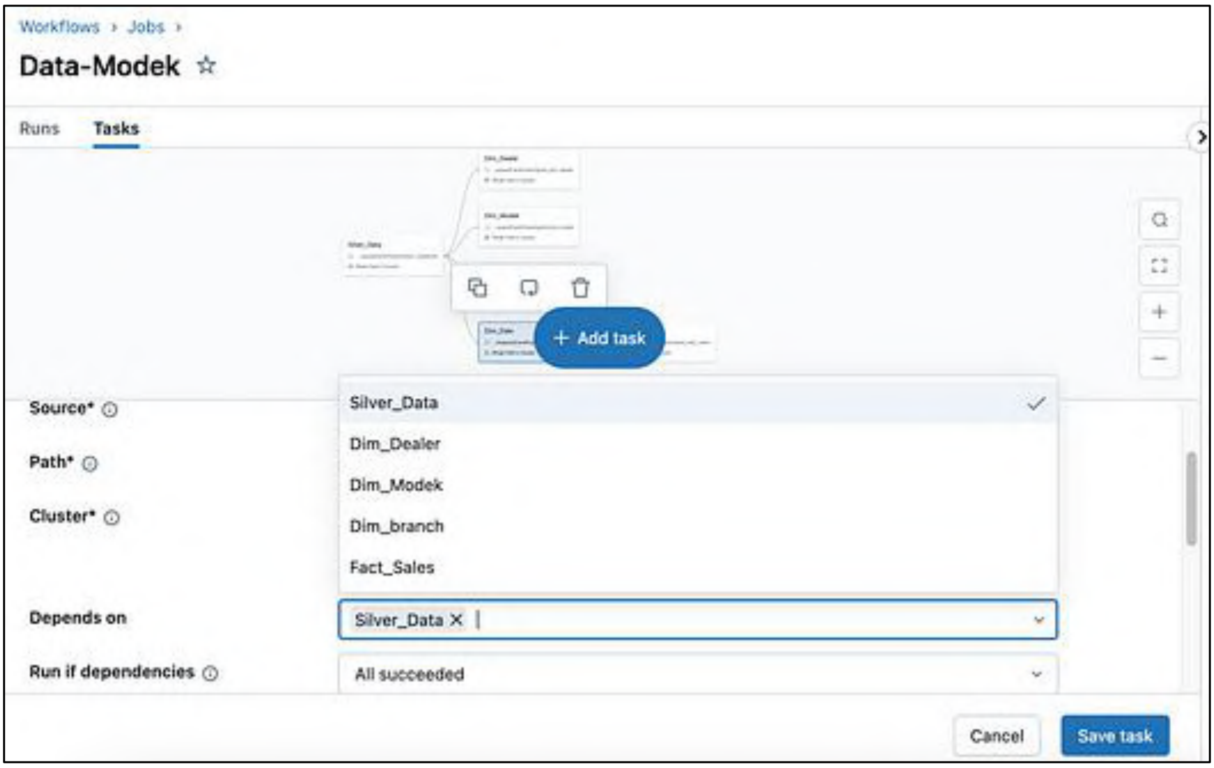
For the dimension model, make sure to configure a parameter of the incremental_flag at the stage of creating the task, as shown below:



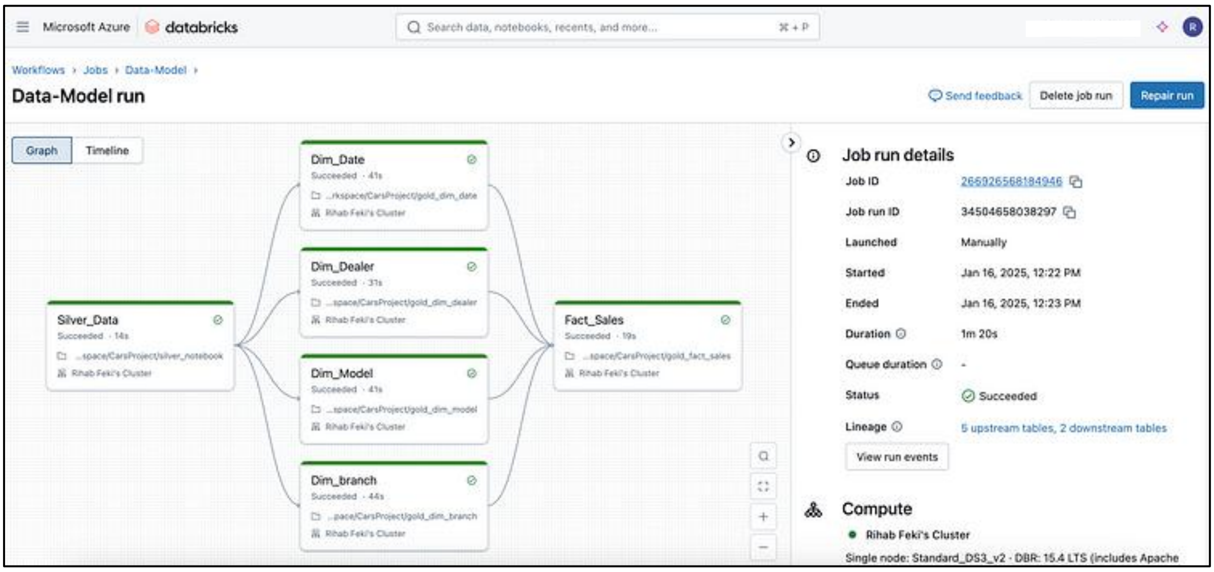
after adding all the dimensions tasks and the fact table task, you will end up having a sequential pipeline like the following:



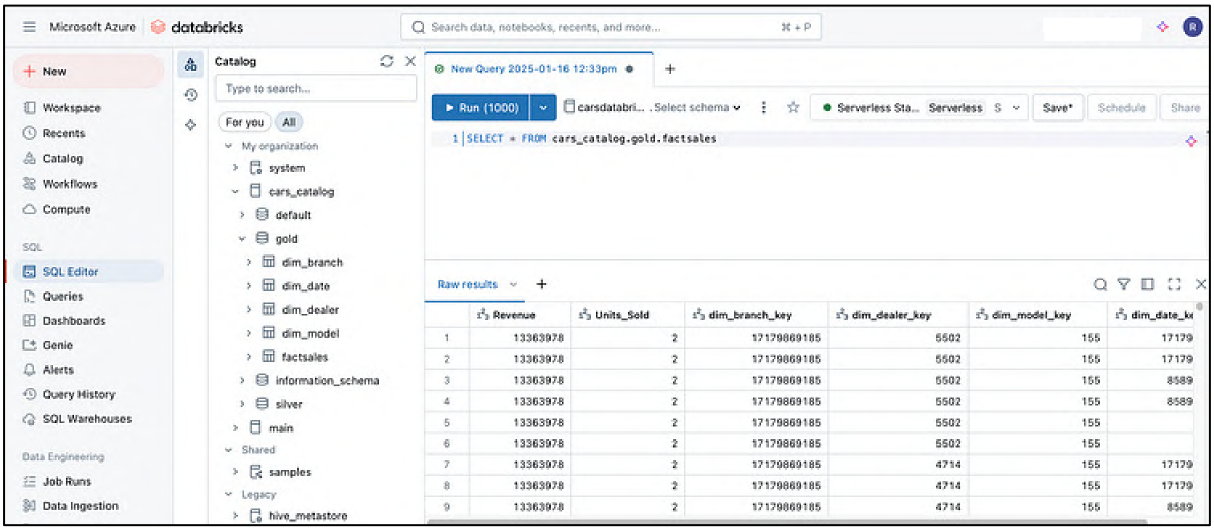
But to enhance the performance, we need to make all the DIM_tables tasks depend on the silver table and then make the fact_table depend on all the dimension tables by editing the Depends on option in the form.



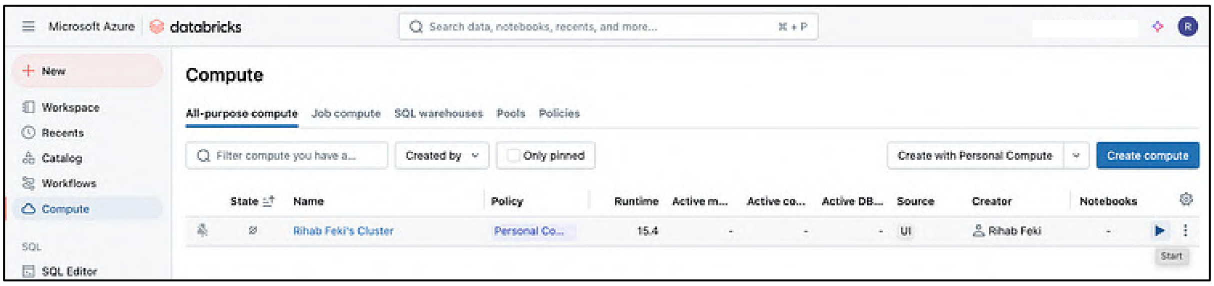
Now that we have all the tasks organized, click on 'Run now' to test the pipeline. Some steps of the pipeline could throw an error, in that case, click on the task highlight the error fix it in the Notebooks in the workspace, and re-run the workflow until it all succeeds.



After creating the Fact tables and the dimensions in the Gold layer, the Data Analyst can now use this data to make SQL queries via the SQL Editor



Make sure to turn off the compute once you are done with it.



To test the functioning of the whole pipeline, navigate to the data factory, choose the incremental pipeline, run it again, and verify the count of the rows to verify the results (via the query editor in databricks)

At this stage we finished the whole end to end pipeline using Azure and Databricks