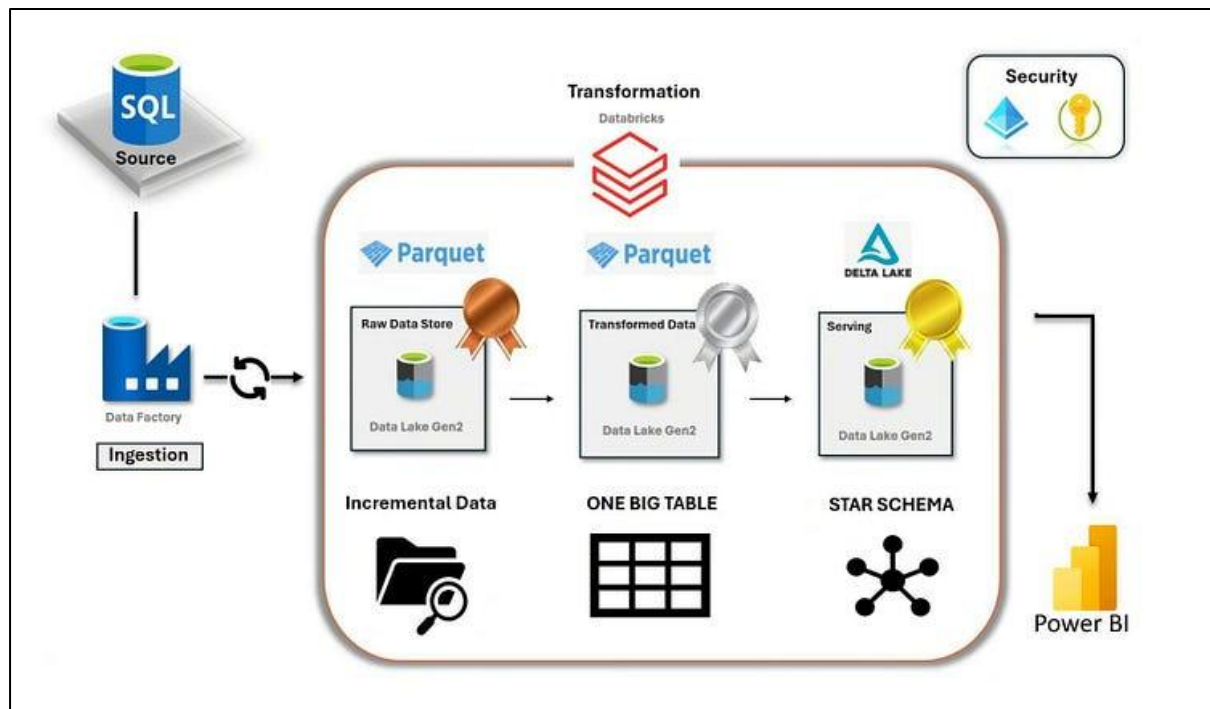


End-to-End Azure Data Engineering Pipeline



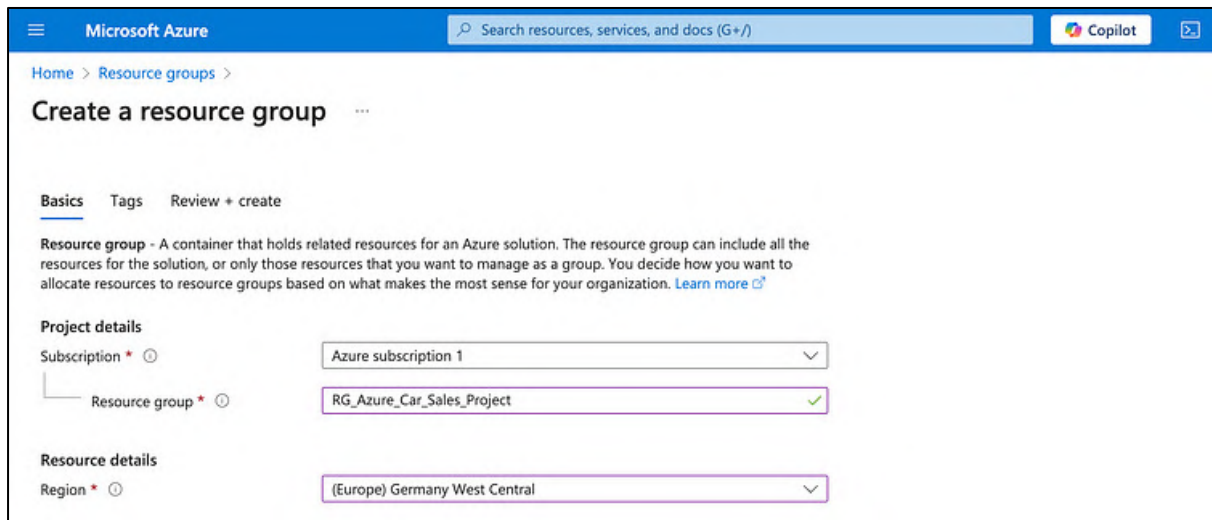
Project architecture

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: Mediation Architecture
- Azure Services for Data Engineering

Step 1: Create a Resource Group

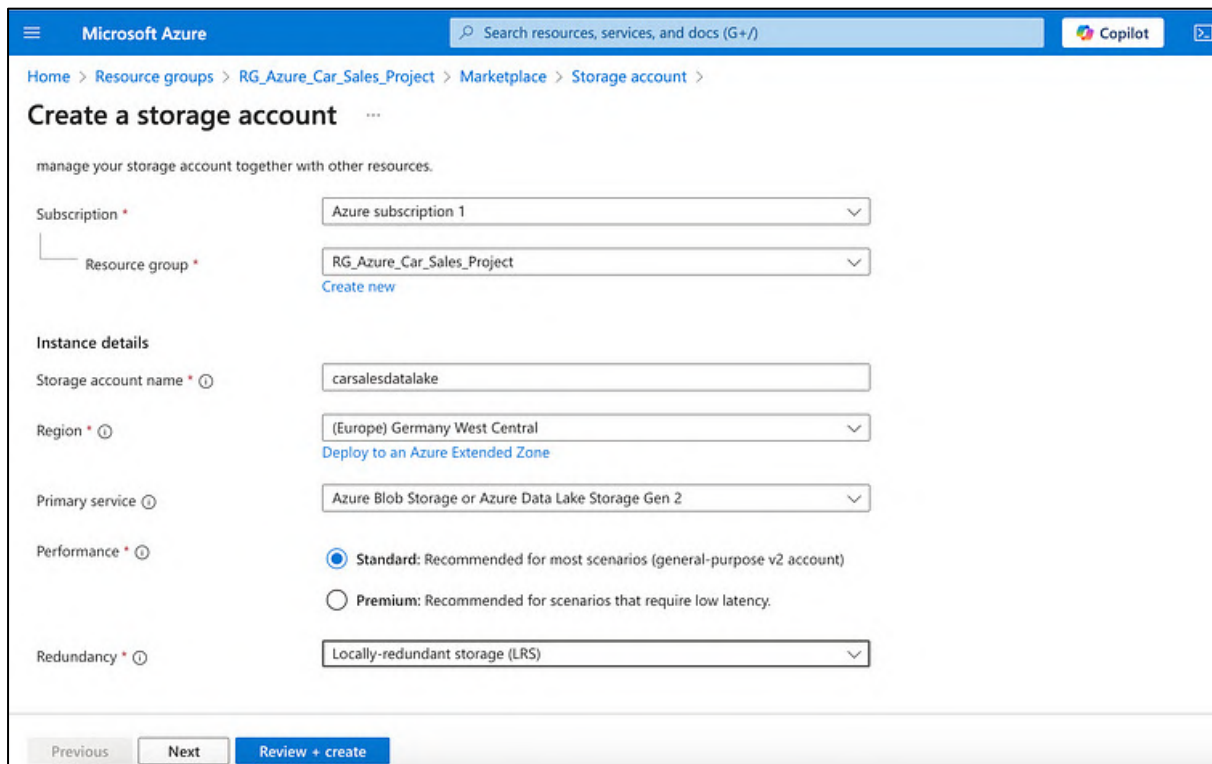
Starting with a resource group in Azure is not just a best practice but a foundational step toward effective cloud resource management. Resource groups enhance organization, improve security through access control, facilitate cost tracking, enable consistent deployments, and allow for environment isolation.



The screenshot shows the 'Create a resource group' page in the Microsoft Azure portal. The page has a blue header with the Microsoft Azure logo, a search bar, and a Copilot button. The breadcrumb trail is 'Home > Resource groups >'. The main heading is 'Create a resource group'. Below the heading are three tabs: 'Basics' (selected), 'Tags', and 'Review + create'. A descriptive paragraph explains that a resource group is a container for related resources. The 'Project details' section contains two dropdown menus: 'Subscription' (set to 'Azure subscription 1') and 'Resource group' (set to 'RG_Azure_Car_Sales_Project' with a green checkmark). The 'Resource details' section contains one dropdown menu: 'Region' (set to '(Europe) Germany West Central').

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.



The screenshot shows the 'Create a storage account' page in the Microsoft Azure portal. The page has a blue header with the Microsoft Azure logo, a search bar, and a Copilot button. The breadcrumb trail is 'Home > Resource groups > RG_Azure_Car_Sales_Project > Marketplace > Storage account >'. The main heading is 'Create a storage account'. Below the heading is a sub-heading 'manage your storage account together with other resources.' The 'Subscription' dropdown is set to 'Azure subscription 1'. The 'Resource group' dropdown is set to 'RG_Azure_Car_Sales_Project' with a 'Create new' link below it. The 'Instance details' section contains three dropdown menus: 'Storage account name' (set to 'carsalesdatalake'), 'Region' (set to '(Europe) Germany West Central' with a 'Deploy to an Azure Extended Zone' link below it), and 'Primary service' (set to 'Azure Blob Storage or Azure Data Lake Storage Gen 2'). The 'Performance' section has two radio buttons: 'Standard: Recommended for most scenarios (general-purpose v2 account)' (selected) and 'Premium: Recommended for scenarios that require low latency.' The 'Redundancy' dropdown is set to 'Locally-redundant storage (LRS)'. At the bottom are three buttons: 'Previous', 'Next', and 'Review + create'.

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.

Microsoft Azure

Search resources, services, and docs (G+/I)

Copilot

Home > Data factories >

Create Data Factory

Basics Git configuration Networking Advanced Tags Review + create

One-click to create data factory with sample pipeline and datasets. [Try it](#)

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Azure subscription 1

Resource group * ⓘ RG_Azure_Car_Sales_Project [Create new](#)

Instance details

Name * ⓘ df-car-sales ✓

Region * ⓘ Germany West Central

Version * ⓘ V2

Previous Next Review + create

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.

Microsoft Azure

Search resources, services, and docs (G+/I)

Copilot

Home > Resource groups > RG_Azure_Car_Sales_Project > Marketplace > Azure SQL > Select SQL deployment option > Create SQL Database >

Create SQL Database Server

Microsoft

Select your preferred authentication methods for accessing this server. Create a server admin login and password to access your server with SQL authentication, select only Microsoft Entra authentication [Learn more](#) using an existing Microsoft Entra user, group, or application as Microsoft Entra admin [Learn more](#), or select both SQL and Microsoft Entra authentication.

Authentication method

☐ Use Microsoft Entra-only authentication

☒ Use both SQL and Microsoft Entra authentication

☐ Use SQL authentication

Set Microsoft Entra admin

ribipersonal_gmail.com#EXT#@ribipersonalgmail.onmicrosoft.com

Admin Object/App ID: 4aa909f8-ce9b-413b-b770-b0016b08d14d

[Set admin](#)

Server admin login * sql-admin ✓

Password * ✓

Confirm password * ✓

Create a Server

Continue with the steps to create a managed Azure SQL Database

Microsoft Azure

Search resources, services, and docs (G+)

Copilot

Home > Resource groups > RG_Azure_Car_Sales_Project > Marketplace > Azure SQL > Select SQL deployment option >

Create SQL Database

Microsoft

Product details

SQL database
by Microsoft
[Terms of use](#) | [Privacy policy](#)

Estimated cost

Storage cost 5.70 USD / month + Compute cost 0.000159 USD / vCore second

Cost summary

General Purpose (GP_S_Gen5_1)

Cost per GB (in USD)0.14

Max storage selected (in GB)x 41.6

ESTIMATED STORAGE COST / MONTH5.70 USD

COMPUTE COST / VCORE SECOND¹0.000159 USD

NOTES

¹ Serverless databases are billed in vCore seconds based on a combination of CPU and memory utilization. [Learn more about serverless billing](#)

Terms

By clicking "Create", I (a) agree to the legal terms and privacy statement(s) associated with the Marketplace offering(s) listed above; (b) authorize Microsoft to bill my current payment method for the fees associated with the offering(s), with the same billing frequency as my Azure subscription; and (c) agree that Microsoft may share my contact, usage and transactional information with the provider(s) of the offering(s) for support, billing and other transactional activities. Microsoft does not provide rights for third-party offerings. For additional details see [Azure Marketplace Terms](#).

Basics

SubscriptionAzure subscription 1

Resource groupRG_Azure_Car_Sales_Project

RegionGermany West Central

Database namecar-sales-sql-db

Server(new) car-sales--db-server

Authentication methodSQL and Microsoft Entra authentication

Create

< Previous

[Download a template for automation](#)

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

Microsoft Azure

Search resources, services, and docs (G+)

Copilot

Home > Resource groups > RG_Azure_Car_Sales_Project > datalakecarsales

Search

Container

Change access level

Restore containers

Refresh

Delete

Give feedback

Search containers by prefix

Name	Last modified	Anony
<input type="checkbox"/> \$logs	1/2/2025, 3:44:39 PM	Private
<input type="checkbox"/> bronze	1/2/2025, 4:38:17 PM	Private
<input type="checkbox"/> gold	1/2/2025, 4:38:34 PM	Private
<input type="checkbox"/> silver	1/2/2025, 4:38:26 PM	Private

diagnose-and-solve-problems

Access Control (IAM)

Data migration

Events

Storage browser

Storage Mover

Partner solutions

Data storage

Containers

File shares

Queues

Tables

data lake carsales | Containers

Storage account

New container

Name *

Anonymous access level

Private (no anonymous access)

The access level is set to private to disabled on this storage account.

Advanced

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.

The screenshot shows the 'Query editor (preview)' interface for a database named 'car-sales-sql-db'. On the left is a sidebar with navigation options: Overview, Activity log, Tags, Diagnose and solve problems, Query editor (preview) (selected), Mirror database in Fabric (preview), Settings, Data management, Integrations, Power Platform, Security, Intelligent performance, and Monitoring. The main area displays a 'Welcome to SQL Database Query Editor' message. Below this, there are two authentication sections. The 'SQL server authentication' section has fields for 'Login' (containing 'sql-admin') and 'Password' (masked with dots), with an 'OK' button below. The 'Microsoft Entra authentication' section shows a 'Logged in' status with a green checkmark and a 'Continue' button. A large 'SQL' logo is also visible.

login to the database

	Branch_ID	Dealer_ID	Model_ID	Revenue	Units_Sold	Date_ID	Day	Month	Year	BranchName	DealerName	Product_Name
1	BR0001	DLR0001	BMW-M1	13363978	2	DT00001	1	1	2017	AC Cars Motors	AC Cars Motors	BMW
2	BR0003	DLR0228	Hon-M218	17376468	3	DT00001	10	5	2017	AC Cars Motors	Deccan Motors	Honda
3	BR0004	DLR0208	Tat-M188	9664767	3	DT00002	12	1	2017	AC Cars Motors	Wiesmann Motors	Tata
4	BR0005	DLR0188	Hyu-M158	5525304	3	DT00002	16	9	2017	AC Cars Motors	Subaru Motors	Hyundai
5	BR0006	DLR0168	Ren-M128	12971088	3	DT00003	20	5	2017	AC Cars Motors	Saab Motors	Renault
6	BR0008	DLR0128	Hon-M68	7321228	1	DT00004	28	4	2017	AC Cars Motors	Messerschmitt Motors	Honda
7												

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.

The screenshot shows the 'Query editor (preview)' interface with a new query being created. The query text is as follows:

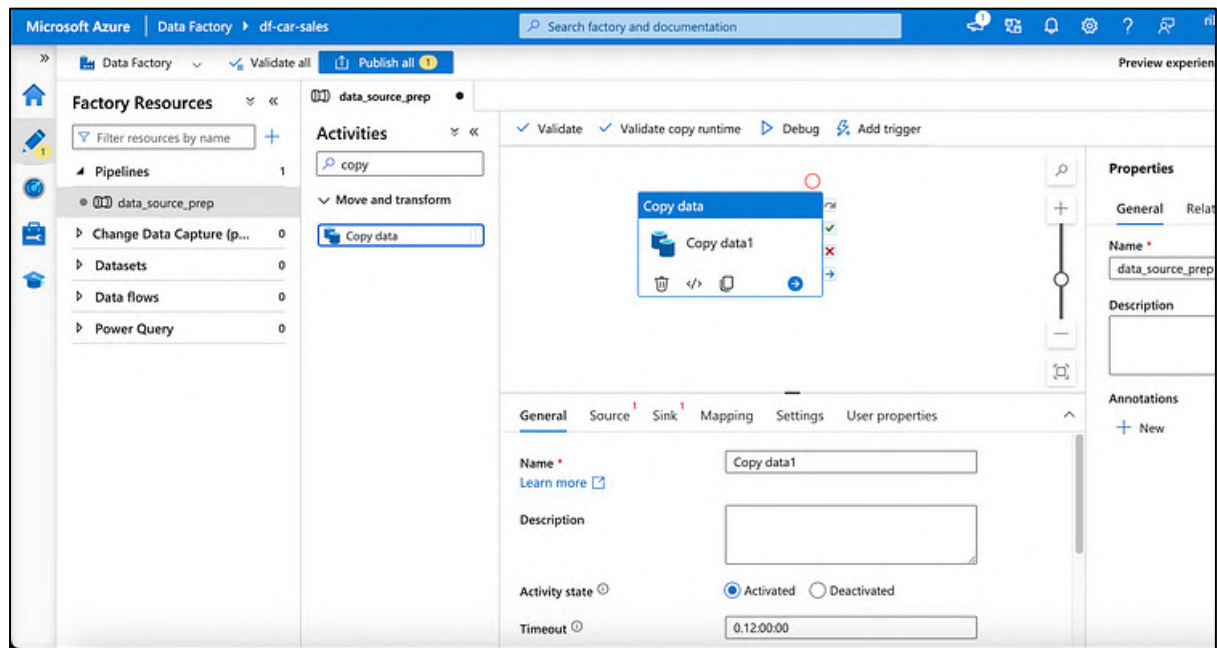
```
1 CREATE TABLE source_cars_data
2 {
3     Branch_ID VARCHAR(200),
4     Dealer_ID VARCHAR(200),
5     Model_ID VARCHAR(200),
6     Revenue BIGINT,
7     Units_Sold BIGINT,
8     Date_ID VARCHAR(200),
9     Day TINYINT,
10    Month TINYINT,
11    Year TINYINT,
12    Branch_Name VARCHAR(2000),
13    Product_Name VARCHAR(200)
14 }
```

The interface includes a sidebar with 'Tables', 'Views', and 'Stored Procedures' options. The main area has buttons for 'Run', 'Cancel query', 'Save query', 'Export data as', 'Show only Editor', and 'Open Copilot'. Below the query editor, there are tabs for 'Results' and 'Messages'.

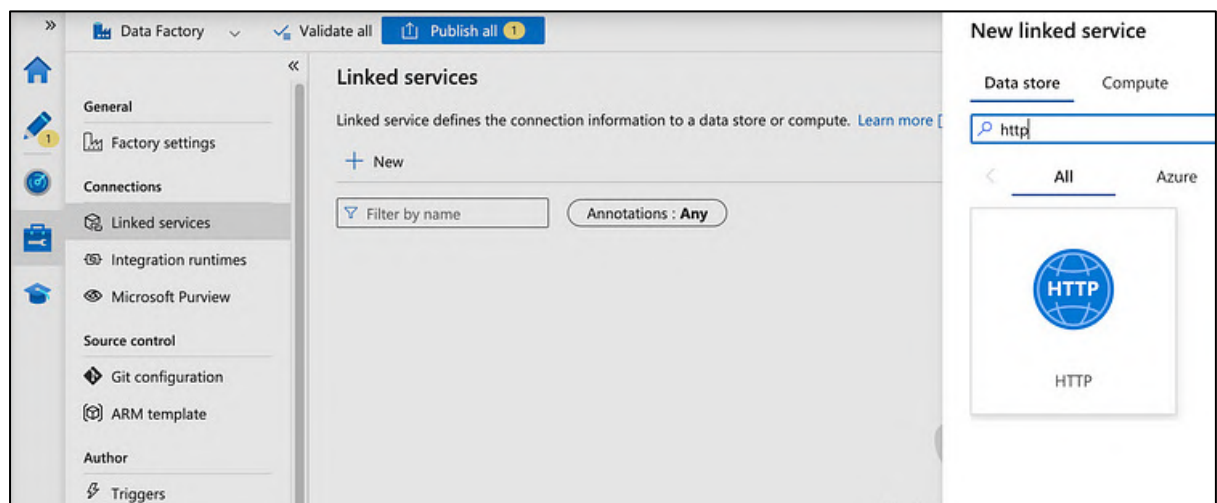
Create Table query

Step 7: Ingest raw data with Data Factory

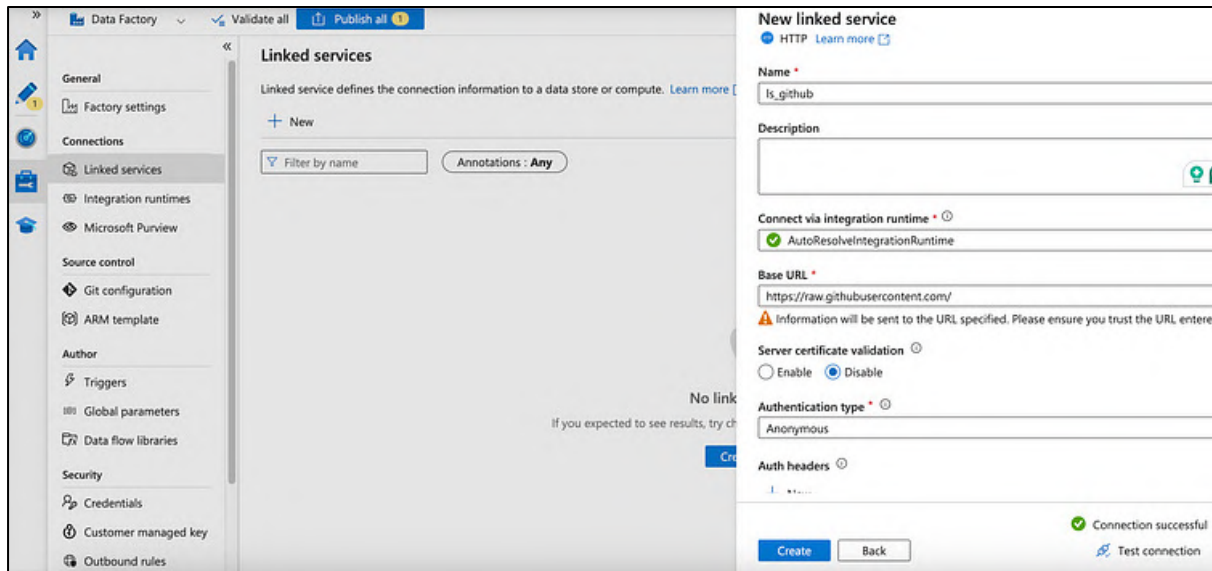
Launch Data Factory navigate to the Author tab and create a pipeline called 'data_source_prep'. What we will do is copy data from GitHub to Azure SQL DB using Data Factory.



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.



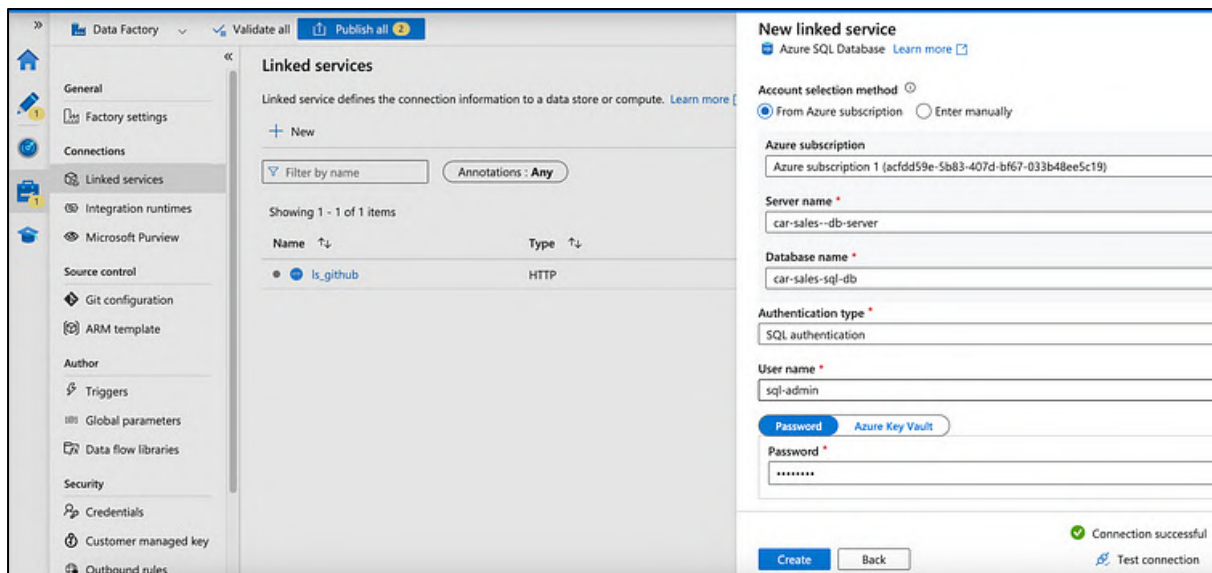
http linked service to connect to github



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.

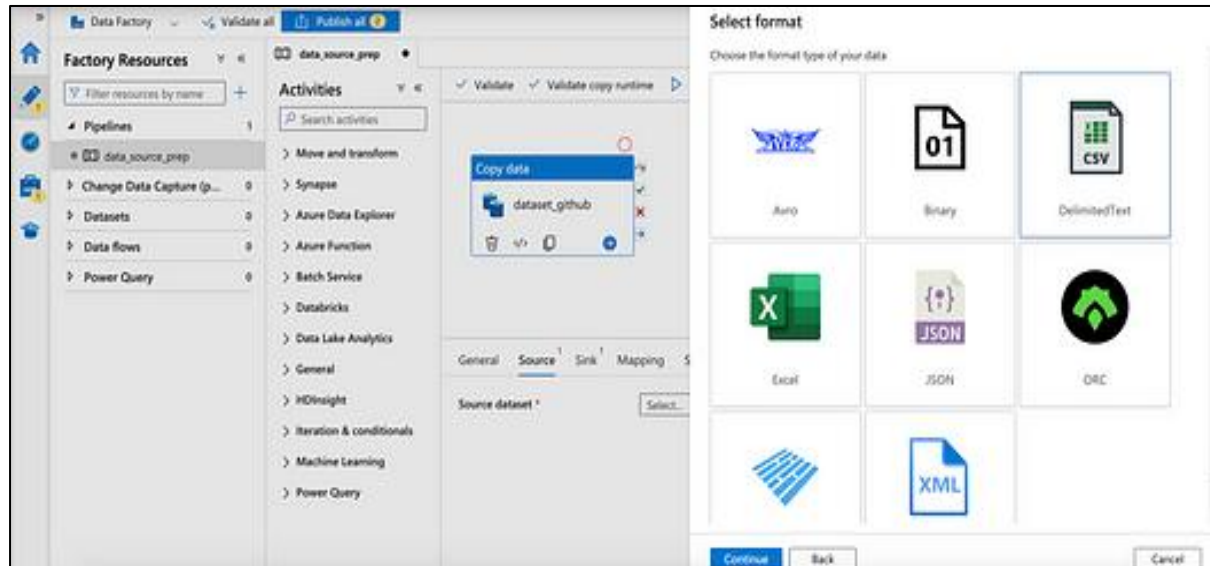
When you test the connection, you might get a connection error because of the firewall to protect the access to the database. To fix that error, navigate to the networking settings in the SQL server and click on “Enable Azure service to access this server” which you should find at the end of the page.



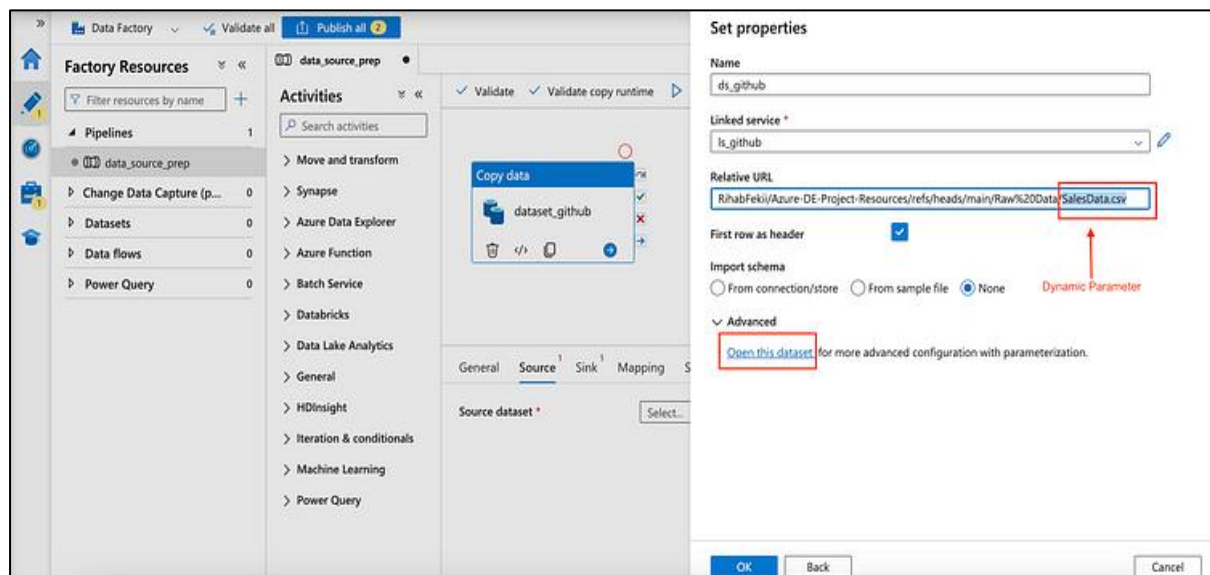
Dynamic ETL: parametrized dataset

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.

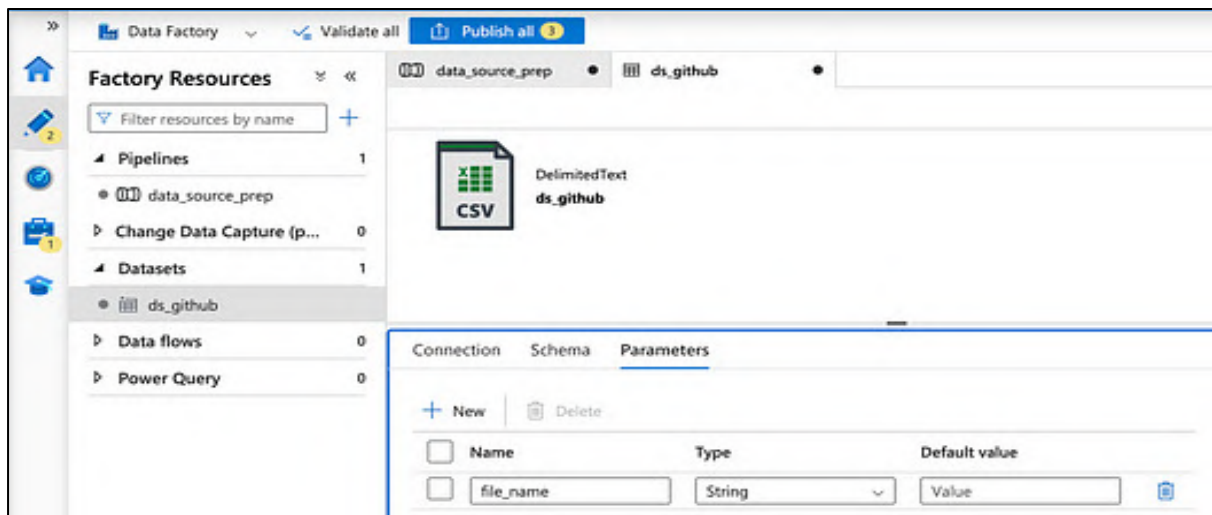
To do that, go to the Author tab and start configuring the Copy Data activity by first adding a new dataset in the source section, typing new, selecting an HTTP data store, and since the data is a CSV, selecting 'Delimited Text'.



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

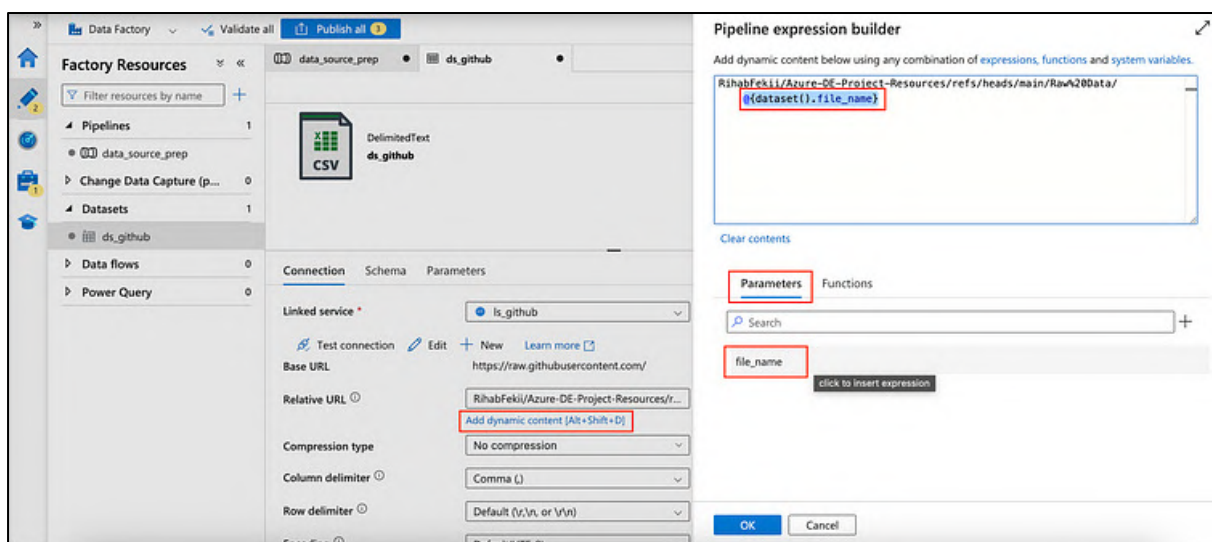


click on Open this dataset for more advanced configurations to add the parameter

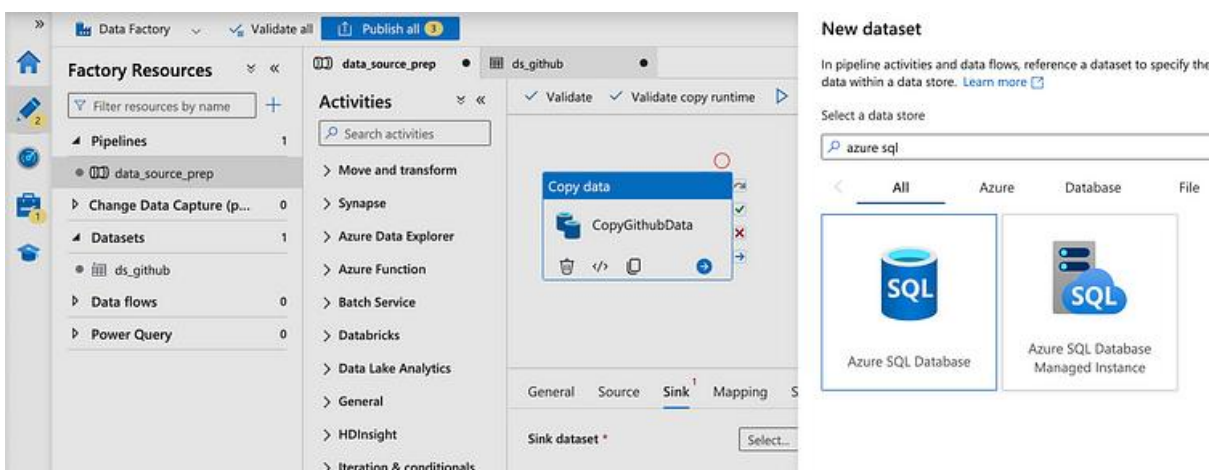


Added the parameter 'file_name'

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

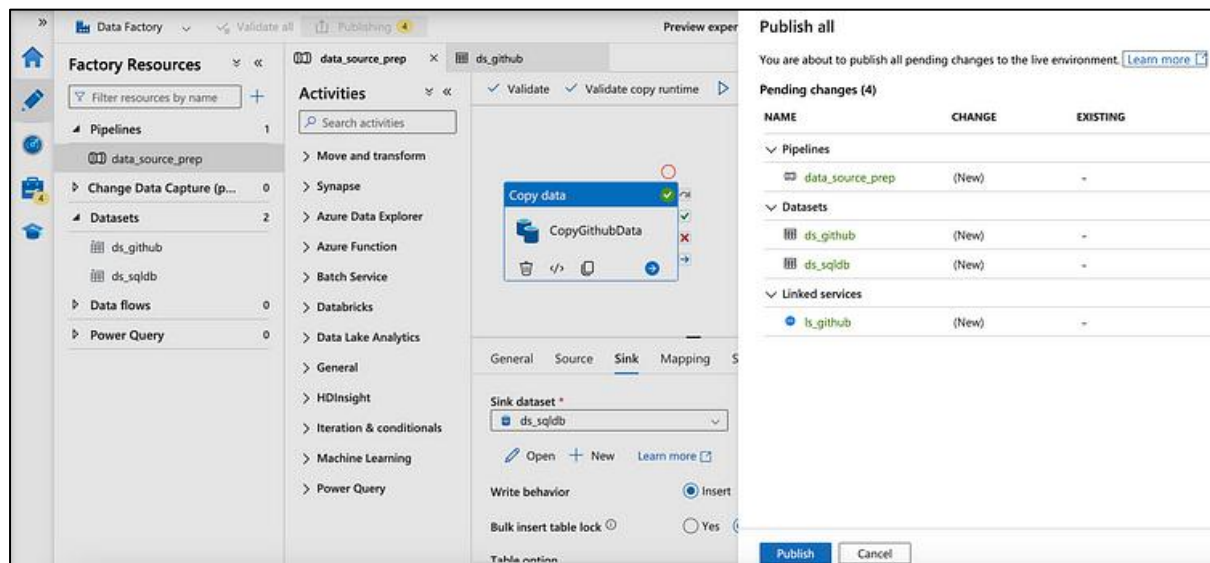


This way we created the dynamic data source, the next step is to configure the **sink** which is the destination where we will load the data in Azure SQL.

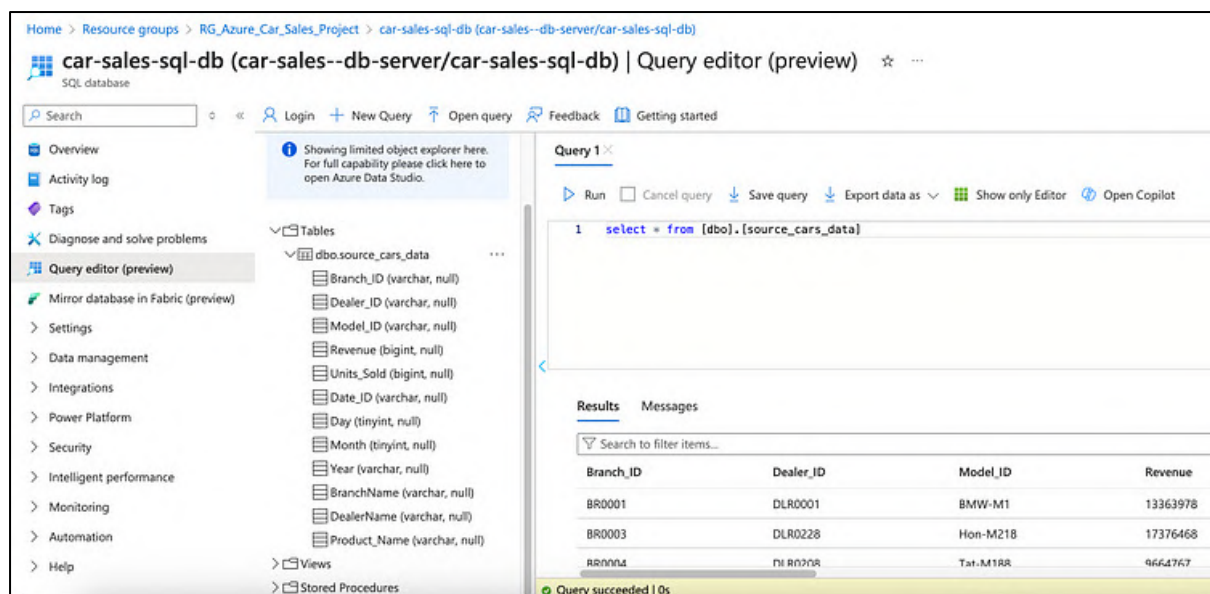


Sink config

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



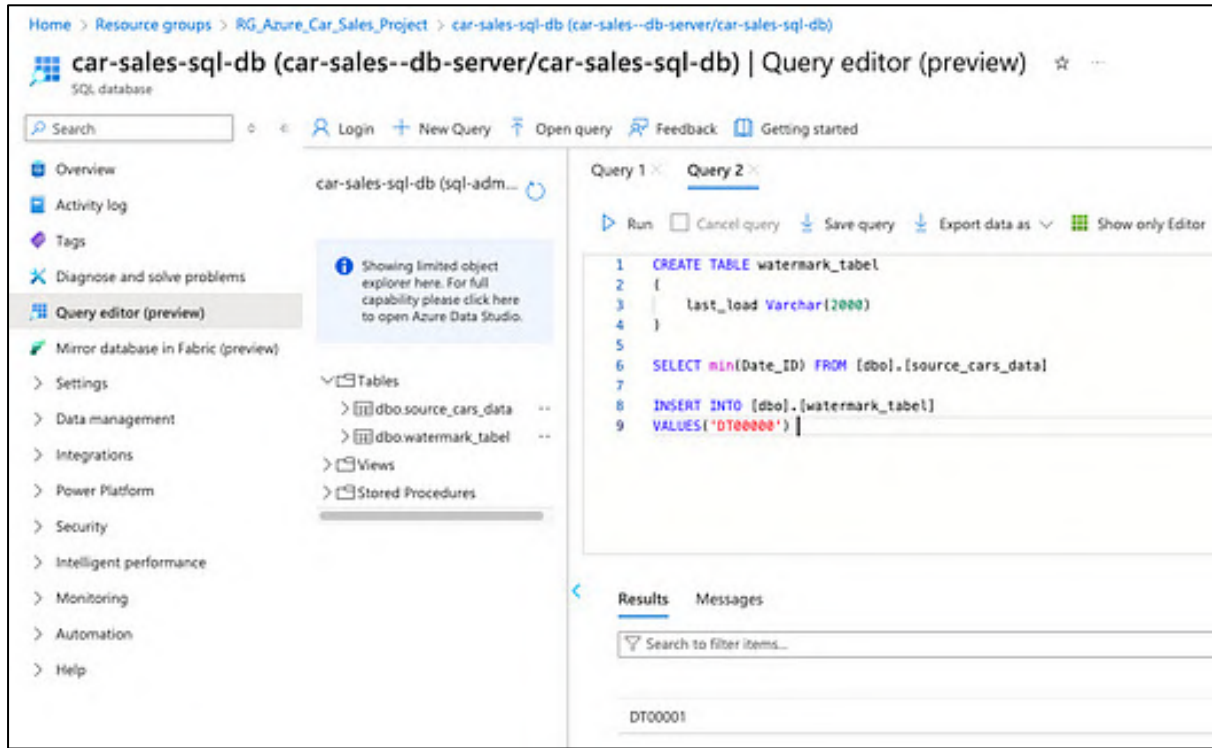
Then test that the copy data pipeline worked correctly by querying the data in the dataset.



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.



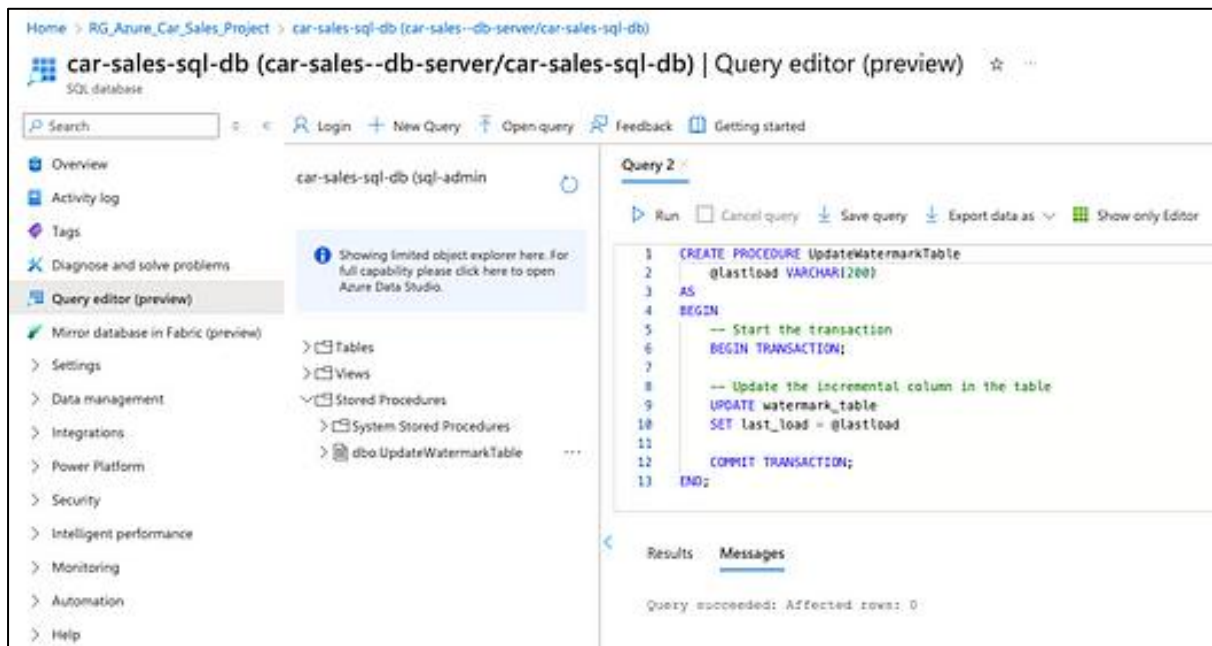
The screenshot shows the Azure Data Studio interface for the 'car-sales-sql-db' database. The left sidebar displays the 'Query editor (preview)' tab. The main pane shows a SQL query in 'Query 2' that creates a 'watermark_table' with a 'last_load' column of type 'Varchar(2000)'. The query also includes a 'SELECT' statement to find the minimum 'Date_ID' from the 'source_cars_data' table and an 'INSERT INTO' statement to store this value in the 'watermark_table'.

```
1 CREATE TABLE watermark_table
2 (
3     last_load Varchar(2000)
4 )
5
6 SELECT min(Date_ID) FROM [dbo].[source_cars_data]
7
8 INSERT INTO [dbo].[watermark_table]
9 VALUES('DT00000')
```

The 'Results' pane at the bottom shows the output of the query, displaying the value 'DT00001'.

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.

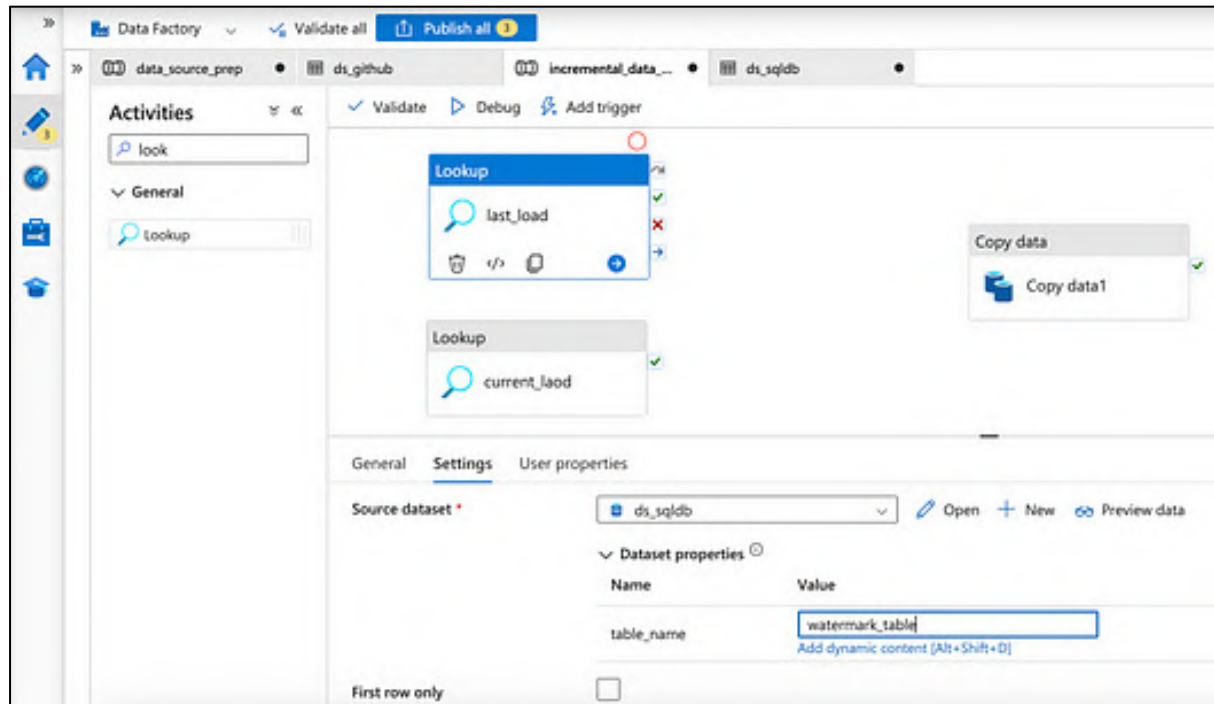


The screenshot shows the Azure Data Studio interface for the 'car-sales-sql-db' database. The left sidebar displays the 'Query editor (preview)' tab. The main pane shows a SQL query in 'Query 2' that creates a stored procedure named 'UpdateWatermarkTable'. The procedure takes a parameter '@lastload' of type 'VARCHAR(200)' and updates the 'last_load' column in the 'watermark_table' with the value of '@lastload'.

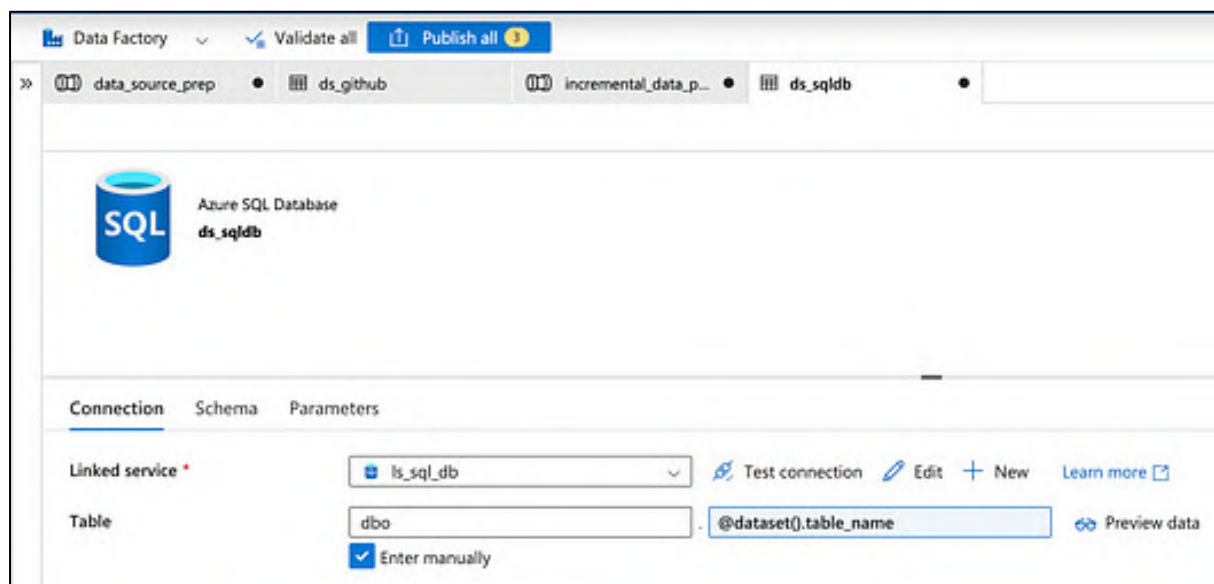
```
1 CREATE PROCEDURE UpdateWatermarkTable
2     @lastload VARCHAR(200)
3 AS
4 BEGIN
5     -- Start the transaction
6     BEGIN TRANSACTION;
7
8     -- Update the incremental column in the table
9     UPDATE watermark_table
10    SET last_load = @lastload
11
12    COMMIT TRANSACTION;
13 END;
```

The 'Results' pane at the bottom shows the output of the query, displaying the message 'Query succeeded: Affected rows: 0'.

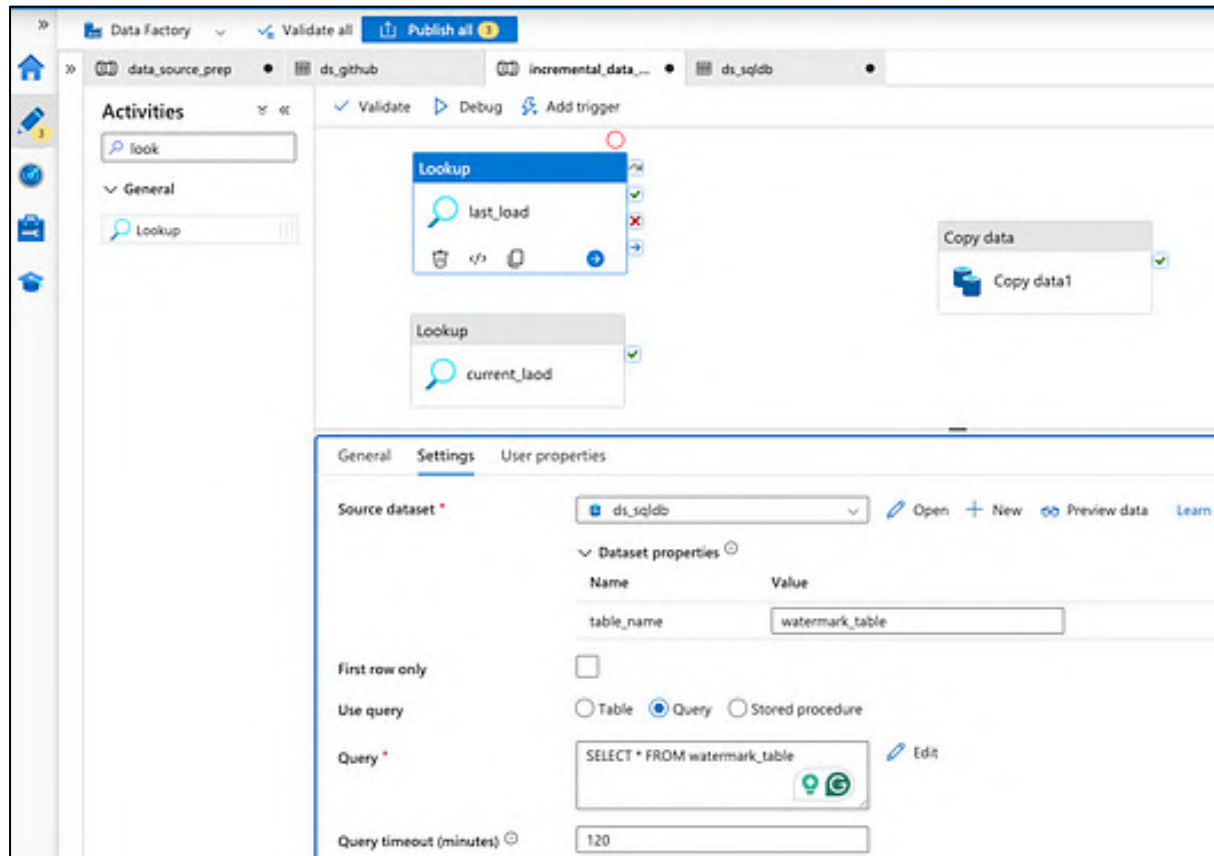
The next step is to add two activities “Lookup”, one that captures the last_load date and the other that captures the current_load date.



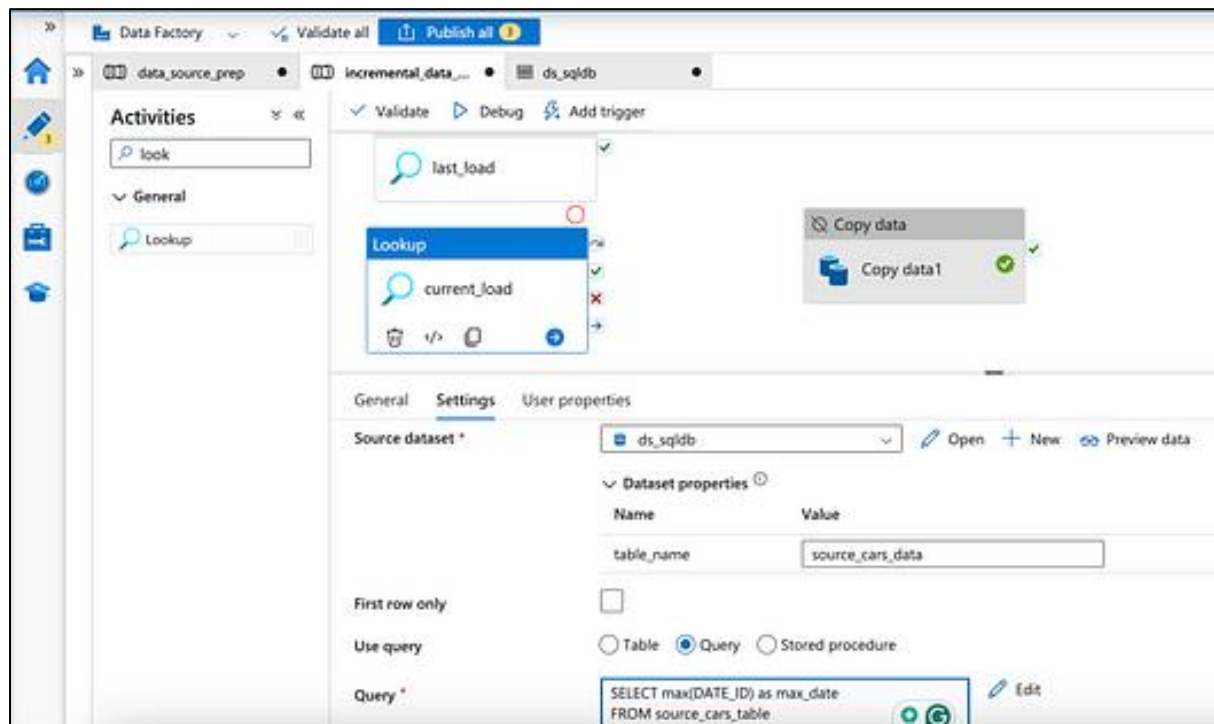
In the configuration of the lookup, select the dataset in the settings tab and then create a parameter for the table_name (which will be used in both Lookups).



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:

The screenshot shows the 'Output' window for the 'last_load' activity. The output is a JSON object:

```

{
  "count": 1,
  "value": [
    {
      "max_date": "DT01245"
    }
  ],
  "effectiveIntegrationRuntime": "AutoResolveIntegrationRuntime (Germany West Central)"
}

```

Below the output window, a table shows the execution status of activities:

Activity	Status	Type	Run start	Duration
current_load	Succeeded	Lookup	1/3/2025, 4:13:08 PM	5s
Copy data1	Inactive	Copy data	1/3/2025, 4:13:08 PM	Less than 1s
last_load	Inactive	Lookup	1/3/2025, 4:13:08 PM	Less than 1s

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 screenshot shows the 'Copy data' activity configuration in the 'Source' tab. The 'Query' is set to:

```

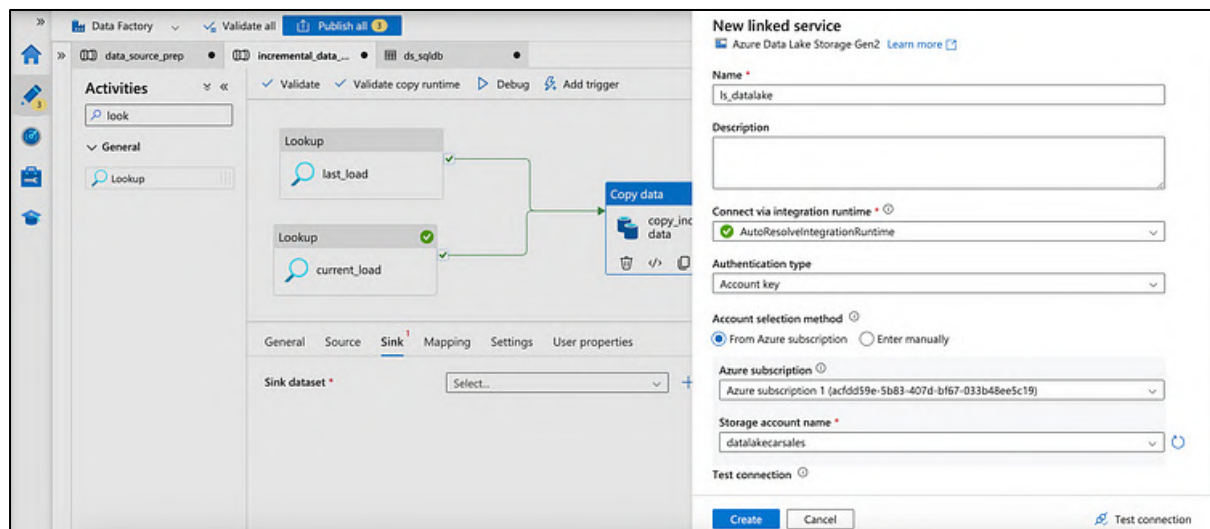
SELECT * FROM source_cars_data
WHERE Date_ID > @activity('last_load').output.value[0].last_load
AND Date_ID <= @activity('current_load').output.value[0].max_date

```

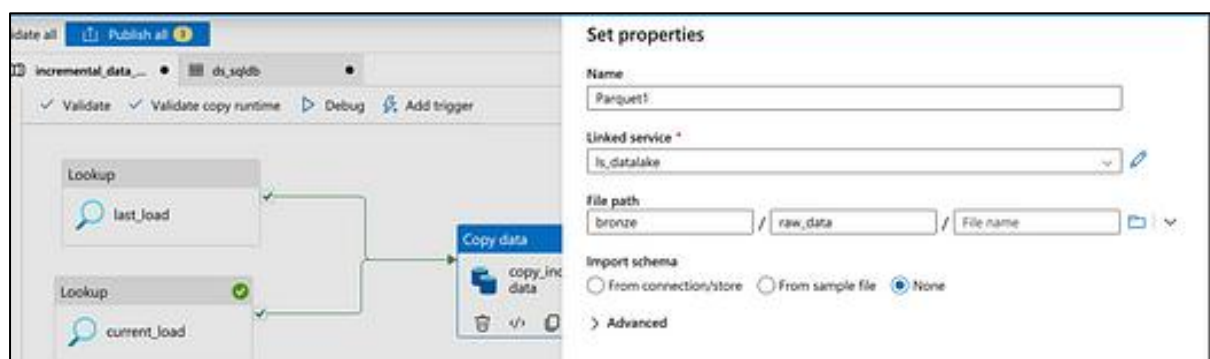
The 'Pipeline expression builder' window is open, showing the 'Activity outputs' tab. The 'current_load' and 'last_load' activity outputs are highlighted.

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.

Pipeline run ID: caa8dcae-90ac-4ae4-a213-f8cfb53293b8 **Pipeline status:** Succeeded

Showing 1 - 3 of 3 items

Activity name	Activity status	Activity type	Run start	Duration	Integration runtime
copy_incremental_data	Succeeded	Copy data	1/3/2025, 9:27:08 PM	19s	AutoResolveIntegration
current_load	Succeeded	Lookup	1/3/2025, 9:27:03 PM	5s	AutoResolveIntegration
last_load	Succeeded	Lookup	1/3/2025, 9:27:03 PM	5s	AutoResolveIntegration

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.

Stored procedure name: [dbo].[UpdateWatermarkTable]

Import (highlighted)

Pipeline expression builder: @activity('current_load').output.value[0].max_date

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

The screenshot shows the Azure Data Factory pipeline editor for a pipeline named 'incremental_data'. The pipeline is composed of four activities: two 'Lookup' activities ('last_load' and 'current_load'), a 'Copy data' activity ('copy_incremental_data'), and a 'Stored procedure' activity ('watermark_update'). The pipeline is connected to a 'ds_sqldb' data store. The 'Pipeline run ID' is '527405f0-b215-467a-b141-3f742f4776e4' and the 'Pipeline status' is 'Succeeded'. Below the pipeline diagram, the 'Output' tab shows a table of activity details.

Activity name	Activity status	Activity type	Run start	Duration	Integration runtime
watermark_update	Succeeded	Stored procedure	1/3/2025, 10:20:13 PM	3s	AutoResolveIntegration
copy_incremental_data	Succeeded	Copy data	1/3/2025, 10:19:57 PM	16s	AutoResolveIntegration
last_load	Succeeded	Lookup	1/3/2025, 10:19:53 PM	4s	AutoResolveIntegration
current_load	Succeeded	Lookup	1/3/2025, 10:19:53 PM	4s	AutoResolveIntegration

The screenshot shows the Azure Data Studio Query editor for a database named 'car-sales-sql-db'. The query editor is displaying a query: 'select * from [dbo].[watermark_table]'. The query has been executed successfully, and the results are shown in the 'Results' tab. The results table has two columns: 'last_load' and 'DT01245'. The 'last_load' column contains the value 'DT01245'.

last_load	DT01245
DT01245	

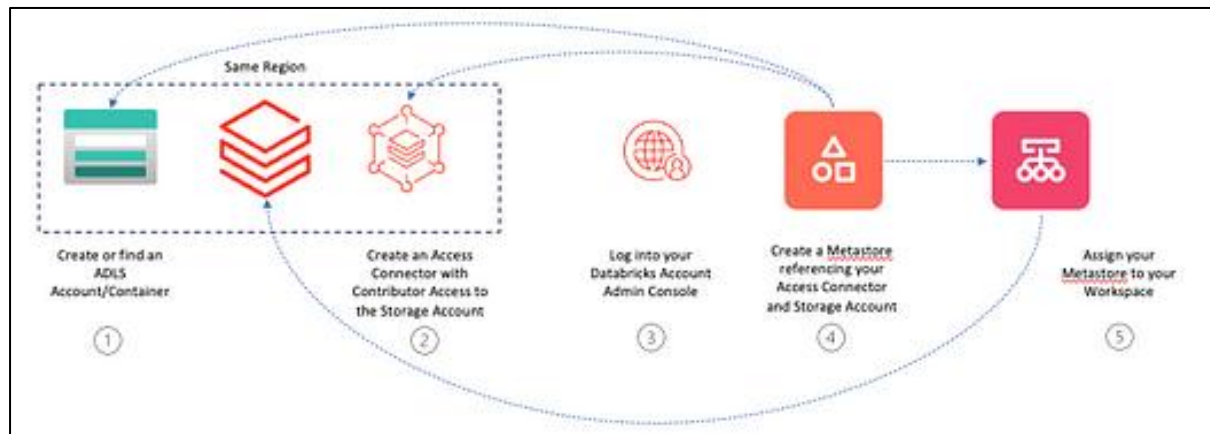
At last, make sure to publish the pipelines in order to save the work after we validated all the steps.

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 pipeline to dynamically load incremental data and saved 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.

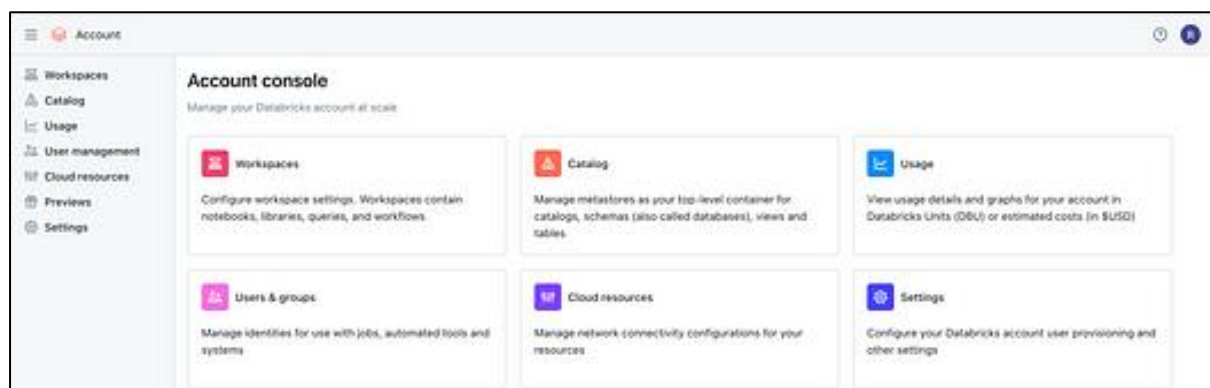
Step 1: Create a Unity Metastore

We will be following these steps:



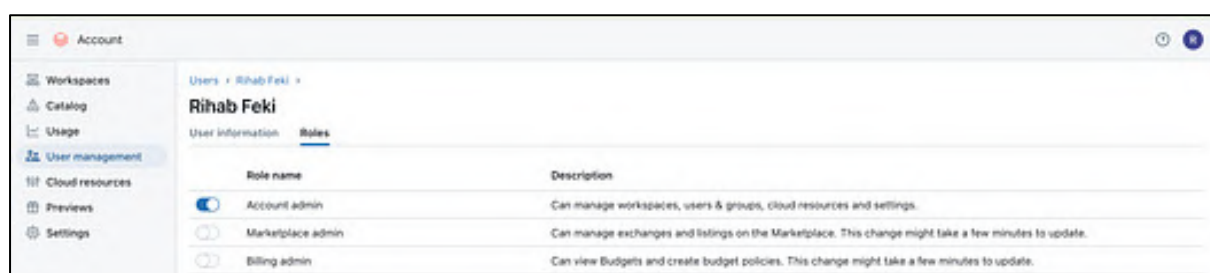
To create Compute, we must attach the Databricks workspace to the Unity Catalog. But to be able to create a Unity Metastore, we need to do that from the admin console.

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).



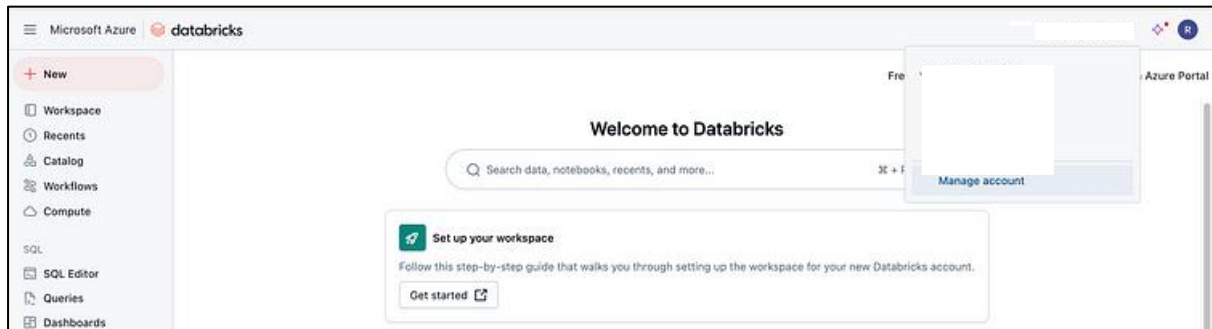
The Databricks admin console

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



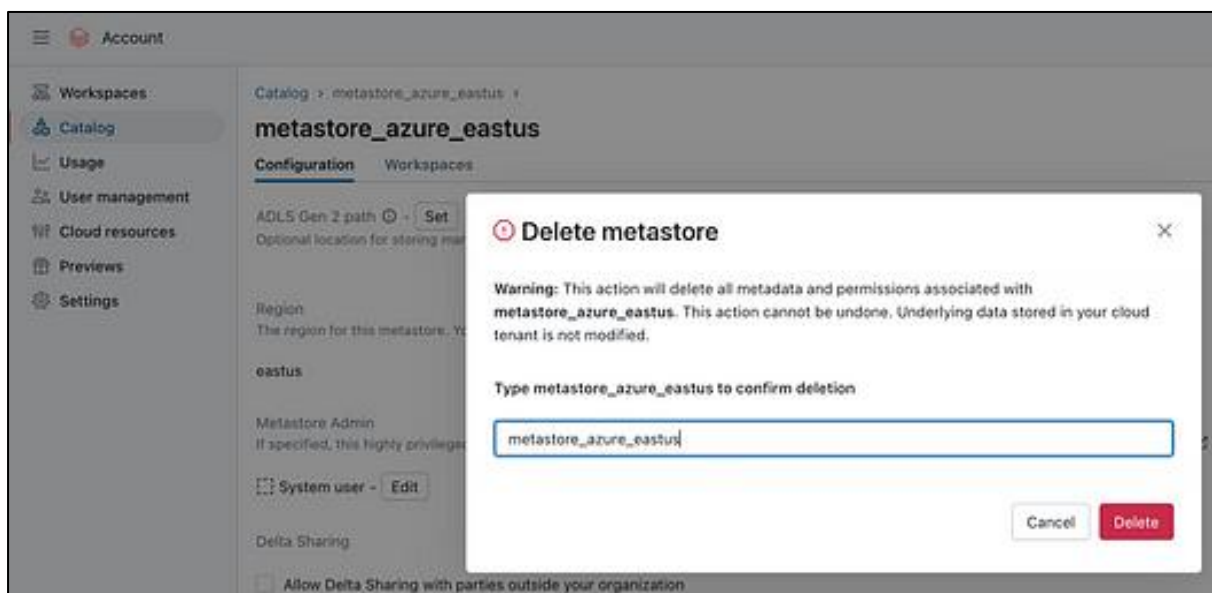
click in Account admin

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.

Account

Workspaces

Catalog

Usage

User management

Cloud resources

Previews

Settings

Catalog > Create metastore >

Create metastore

1 Create metastore — 2 Assign to workspaces

* Name

carsalesmetastore

* Region

westus

Select the region for your metastore. You will only be able to assign workspaces in this region to this metastore.

ADLS Gen 2 path (optional) ?

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

Optional location for storing managed tables data across all catalogs in the metastore. [Learn more](#)

Access Connector Id ?

/subscriptions/acfdd59e-5b83-407d-bf67-033b48ee5c19/resourceGroups/RG_Azure_Car_Sa

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

Home > datalakecarsale

datalakecarsale | Containers

Storage account

Search

+ Container Change access level Restore containers Refresh Delete Give feedback

Search containers by prefix

Name	Last modified	Anony
<input type="checkbox"/> \$logs	1/3/2025, 8:22:15 PM	Private
<input type="checkbox"/> bronze	1/3/2025, 8:53:59 PM	Private
<input type="checkbox"/> gold	1/3/2025, 8:54:19 PM	Private
<input type="checkbox"/> silver	1/3/2025, 8:54:08 PM	Private

New container

Name *

unity-metastore

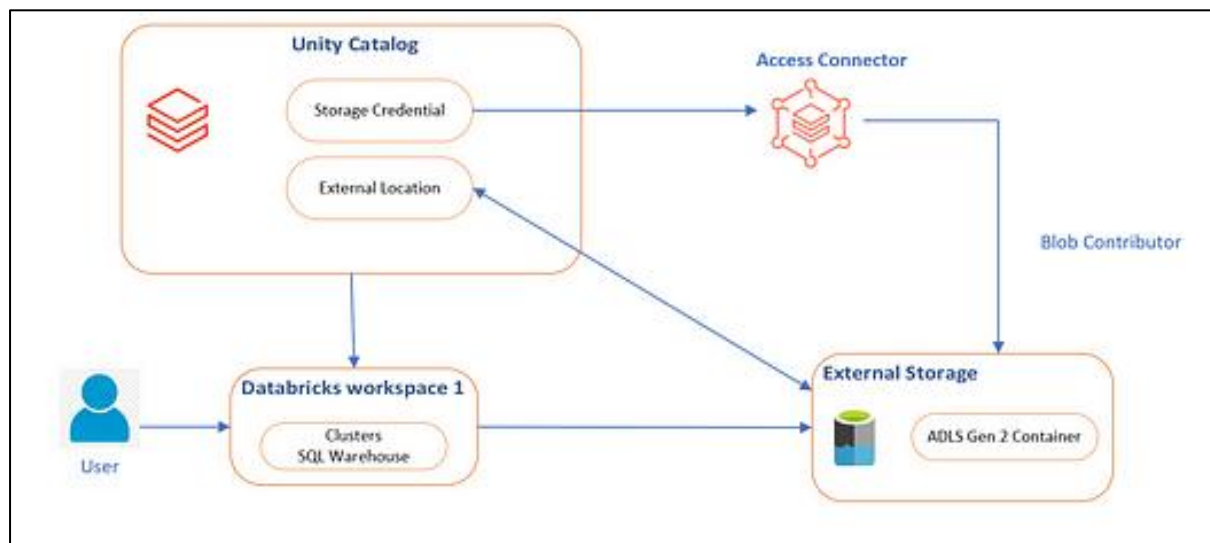
Anonymous access level ?

Private (no anonymous access)

The access level is set to private because anonymous access is disabled on this storage account.

Advanced

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

Home > RG_Azure_Car_Sales_Project > Marketplace > Access Connector for Azure Databricks >

Create an Access Connector for Azure Databricks

Basics | Tags | Managed Identity | Review + create

The Azure Databricks Access Connector lets you connect managed identities to an Azure Databricks account for the purpose of accessing data registered in Unity Catalog.

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * ⓘ Azure subscription 1

Resource group * ⓘ RG_Azure_Car_Sales_Project

[Create new](#)

Instance details

Name * ⓘ carsales_access_connector ✓

Region * ⓘ Germany West Central

[Previous](#) [Next](#) [Review + create](#)

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:

Home > carsales_access_connector | Access control (IAM) >

Add role assignment

Role: **Members** | Conditions: | Review + assign

Selected role: Owner

Assign access to: ☐ User, group, or service principal ☒ Managed identity

Members: + Select members

Name	Object ID	Type
No members selected		

Description: Optional

Review + assign Previous Next

Select managed identities

Some results might be hidden due to your ABAC condition.

Subscription: Azure subscription 1

Managed identity: Access Connector for Azure Databricks (3)

Select: Search by name

- unity-catalog-access-connector /subscriptions/acfd59e-5b83-407d-bf67-033b48ee5c19/resourceGroups/manage...
- unity-catalog-access-connector /subscriptions/acfd59e-5b83-407d-bf67-033b48ee5c19/resourceGroups/RGAzur...

Selected members: carsales_access_connector /subscriptions/acfd59e-5b83-407d-bf67-033b48ee5c19/resourceGroups/... Remove

Select Close

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

Account

Workspaces Catalog > cars_project >

Assign cars_project to workspaces

Filter workspaces

Name	Status	Pricing tier	Resource group	Region	Created	Metastore
AzureDatabricksWS	Running	Premium	RG_Databricks	eastus	last Saturday at 2:59...	-
<input checked="" type="checkbox"/> carsdatabricks	Running	Premium	RG_Azure_Car_Sale...	germanywestcentral	last Saturday at 9:51...	metastore_azur...

☐ Automatically assign new workspaces in germanywestcentral to this metastore

1 workspace selected

Assign Cancel

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

The screenshot shows the Databricks 'New compute' configuration page. The left sidebar contains navigation links: New, Workspace, Recents, Catalog, Workflows, Compute, SQL, SQL Editor, Queries, Dashboards, Genie, Alerts, Query History, SQL Warehouses, Data Engineering, Job Runs, Data Ingestion, and Delta Live Tables. The main area is titled 'Rihab Feki's Cluster' and includes the following settings:

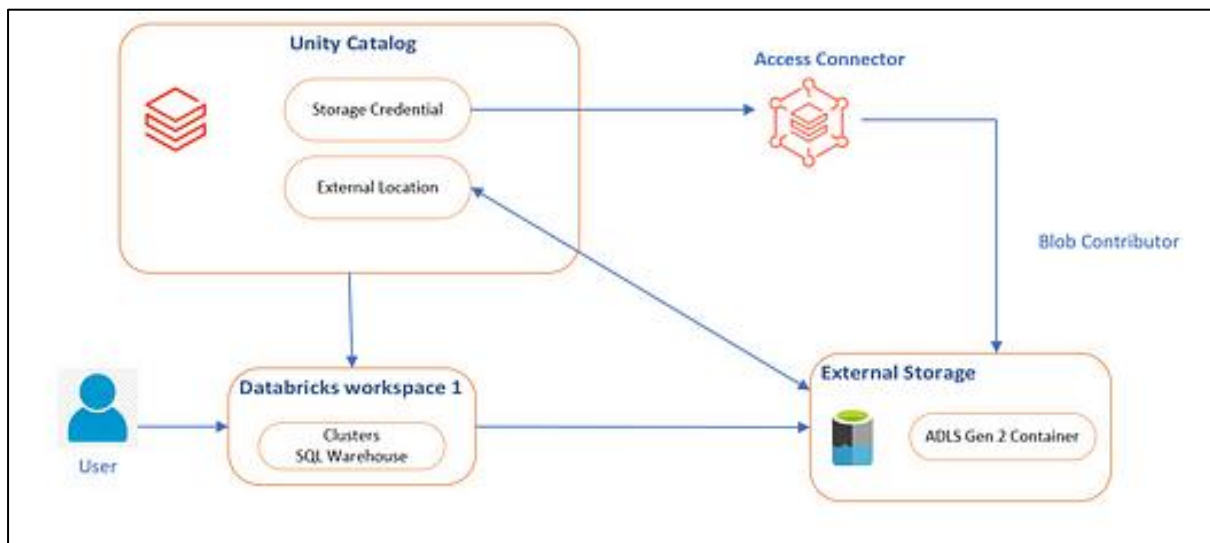
- Policy:** Personal Compute
- Single user access:** Rihab Feki
- Performance:**
 - Databricks runtime version:** Runtime: 15.4 LTS (Scala 2.12, Spark 3.5.0)
 - Node type:** Standard_DS3_v2 (14 GB Memory, 4 Cores)
 - ☒ Terminate after 20 minutes of inactivity
- Tags:** (empty)

At the bottom are 'Create compute' and 'Cancel' buttons. A 'Summary' panel on the right shows: 1 Driver, 14 GB Memory, 4 Cores; Runtime: 15.4 x-scala2.12; Unity Catalog: Standard_DS3_v2; 0.75 DBU/h. A top banner indicates 'Free trial ends in 12 days. Upgrade to Premium in Azure Portal'.

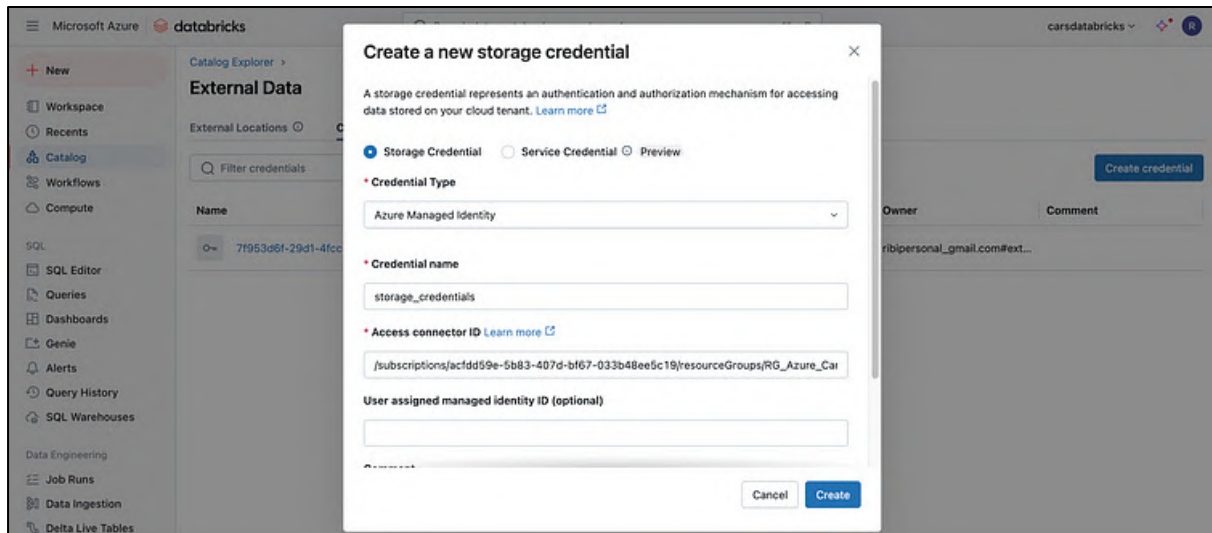
Step 3: Create External locations

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”.

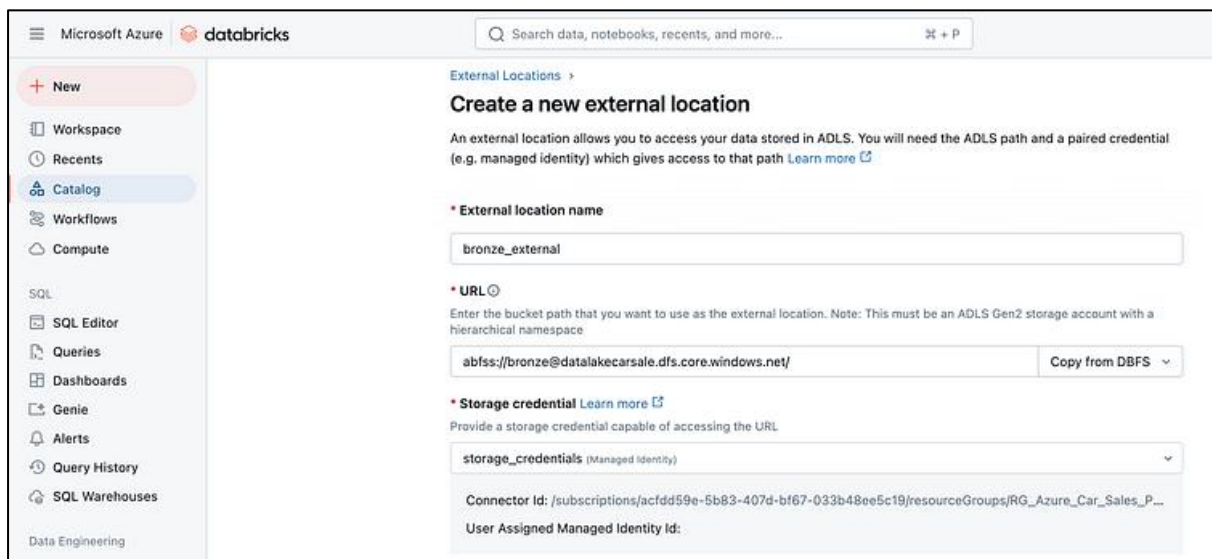


to create an External Location, you need to start by creating credentials. So, navigate to Databricks Workspace and click on Catalog > External Data > Credentials.



start by creating the credential, then the external storage

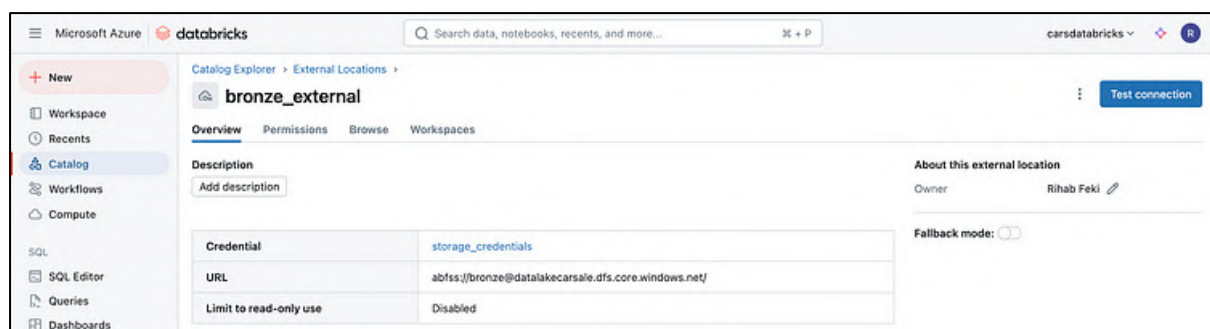
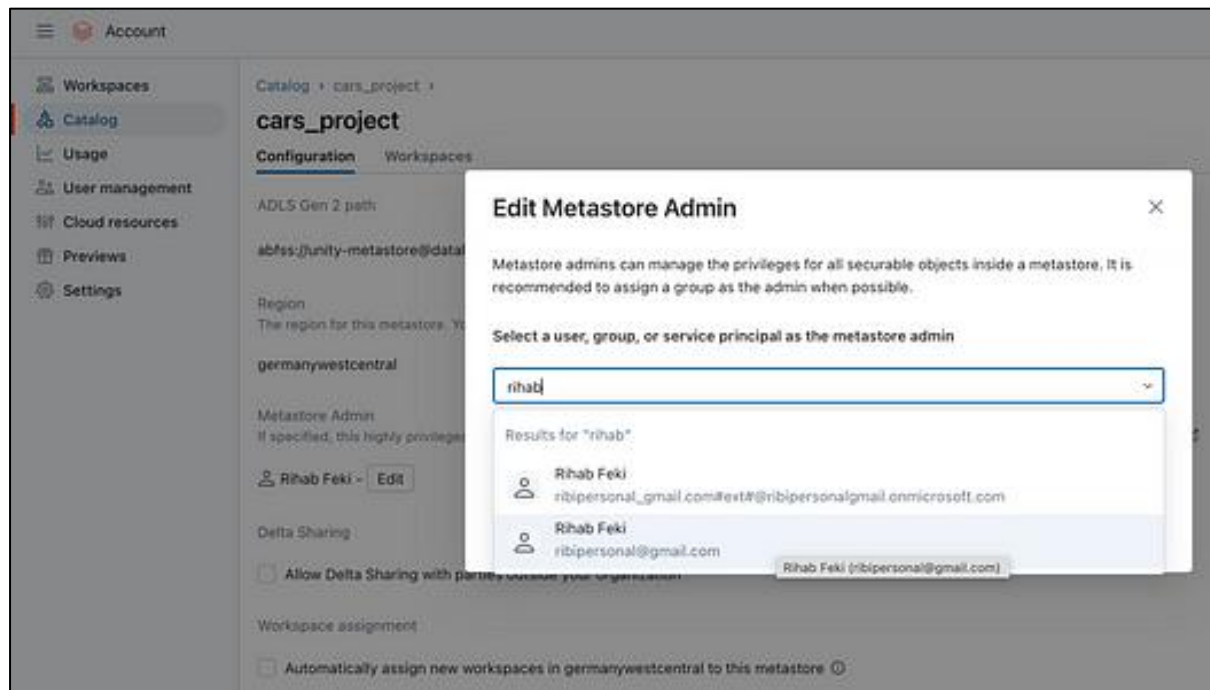
After creating the credentials, click again on Catalog > **External location** And provide a URL following this structure: **abfss://<container_name>@<storage_account_name>.dfs.core.windows.net/<path>**



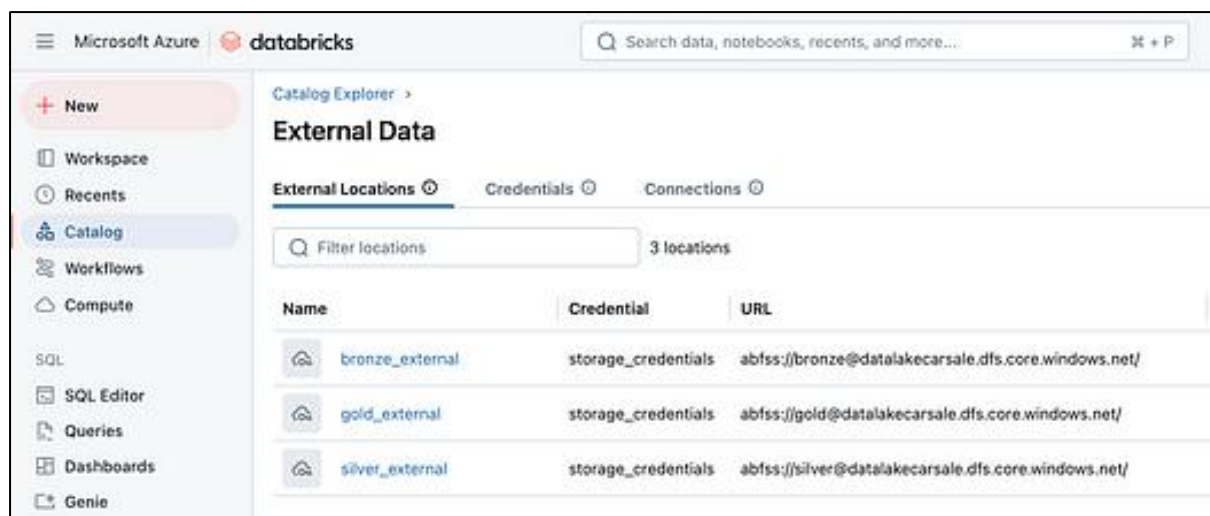
When clicking on create, you will have the following error message:

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)



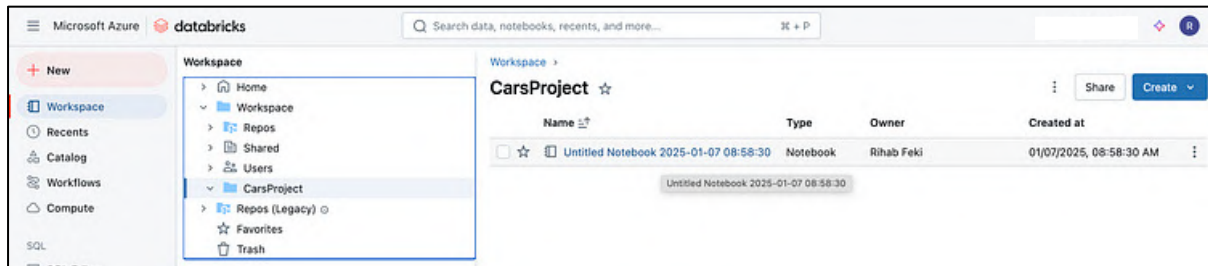
after creating the external location for the bronze layer, do the same work for the silver and gold layers.



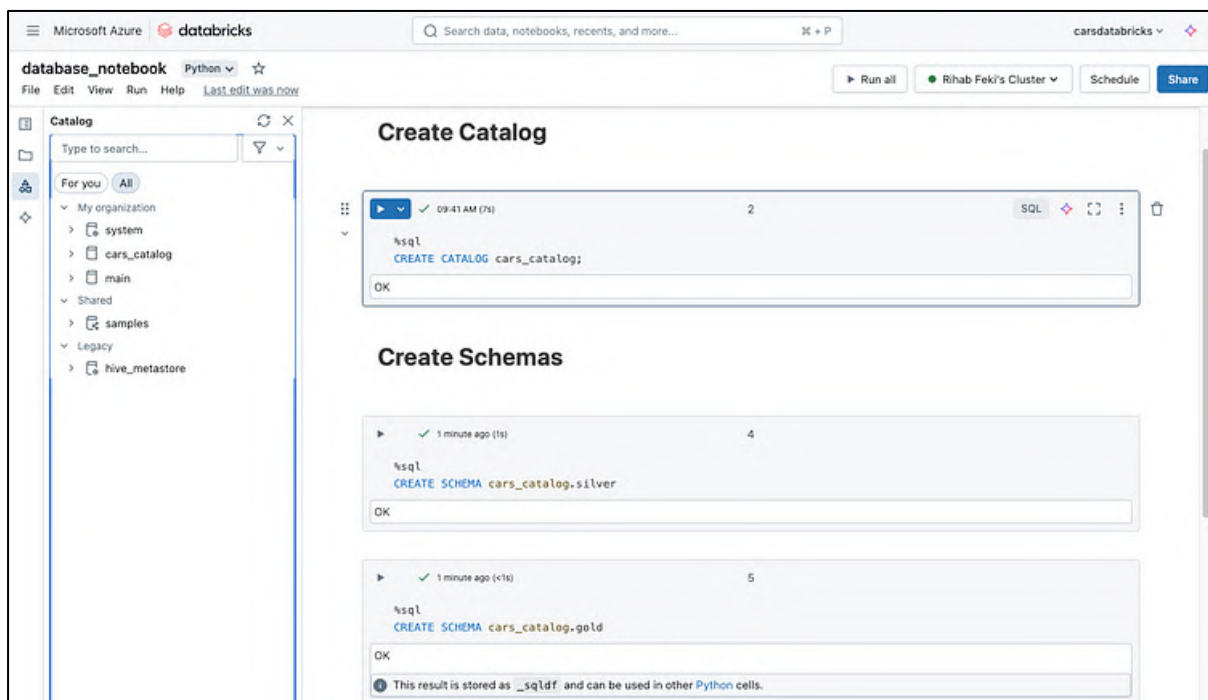
Now we are all set to pull the data from bronze, we need to apply transformation and we need to store that data in the silver container.

Step 4: Create a Workspace

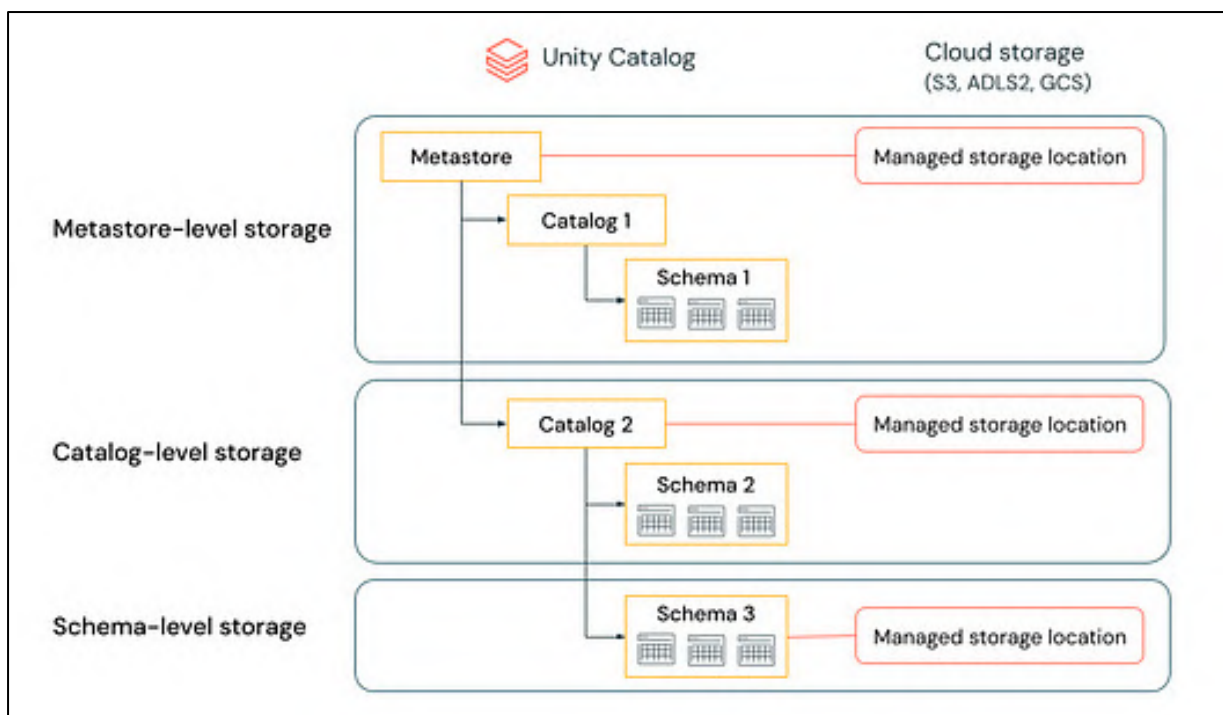
Click on Workspace in the sidebar, then in the Workspace folder, create a new project folder, and inside it create a Notebook to define the catalogs and schemas.



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.

The screenshot shows the Databricks interface for a notebook named 'silver_notebook'. The workspace sidebar on the left shows a folder 'CarsProject' containing 'database_notebook' and 'silver_notebook'. The main area is titled 'Data reading' and contains two code blocks. The first block, executed at 11:04 PM (15s), reads a Parquet file from a specific path using PySpark's read.format() method with the inferSchema option. The second block, executed just now (7s), displays the resulting DataFrame. The output is a table with 8 columns: Branch_ID, Dealer_ID, Model_ID, Revenue, Units_Sold, Date_ID, and Day. The first three rows of data are visible.

```
df = spark.read.format("parquet")\
    .option('inferSchema', True)\
    .load('abfss://bronze@datalakecarsale.dfs.core.windows.net/raw_data')
```

```
df.display()
```

	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	

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.

The screenshot shows the Databricks interface for the same notebook, now titled 'Data transformation'. It contains three code blocks. The first block imports functions from pyspark.sql. The second block uses the withColumn method to split the Model_ID column into a new column named 'model_category' using the F.split function. The third block displays the transformed DataFrame. The output is a table with 9 columns: Date_ID, Day, Month, Year, BranchName, DealerName, Product_Name, and model_category. The first 10 rows of data are visible.

```
from pyspark.sql import functions as F
```

```
df = df.withColumn('model_category', F.split(df['Model_ID'], '-')[0])
```

```
df.display()
```

	Date_ID	Day	Month	Year	BranchName	DealerName	Product_Name	model_category
1	DT00001	1	1	2017	AC Cars Motors	AC Cars Motors	BMW	BMW
2	DT00001	10	5	2017	AC Cars Motors	Deccan Motors	Honda	Hon
3	DT00002	12	1	2017	AC Cars Motors	Wiesmann Motors	Tata	Tat
4	DT00002	16	9	2017	AC Cars Motors	Subaru Motors	Hyundai	Hyu
5	DT00003	20	5	2017	AC Cars Motors	Saab Motors	Renault	Ren
6	DT00004	28	4	2017	AC Cars Motors	Messerschmitt Motors	Honda	Hon
7	DT00004	31	12	2017	AC Cars Motors	Lexus Motors	Cadillac	Cad
8	DT00005	4	9	2017	AC Cars Motors	IFA (including Trabant, Wartburg, Barkas) Motors	Mercedes-Benz	Mer
9	DT00005	2	1	2017	Acura Motors	Acura Motors	BMW	BMW
10	DT00006	9	5	2017	Acura Motors	Geo Motors	Volkswagen	Vol

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

The screenshot shows a Databricks notebook with two cells. The first cell (8) contains the code to create a new column 'revenue_per_unit' by dividing 'Revenue' by 'Units_Sold'. The second cell (9) displays the resulting DataFrame.

```
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]

```
df.display()
```

(2) Spark Jobs

	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 it 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.

The screenshot shows a Databricks notebook with a single cell (11) containing code to aggregate units sold by year and branch name. The resulting DataFrame is displayed below.

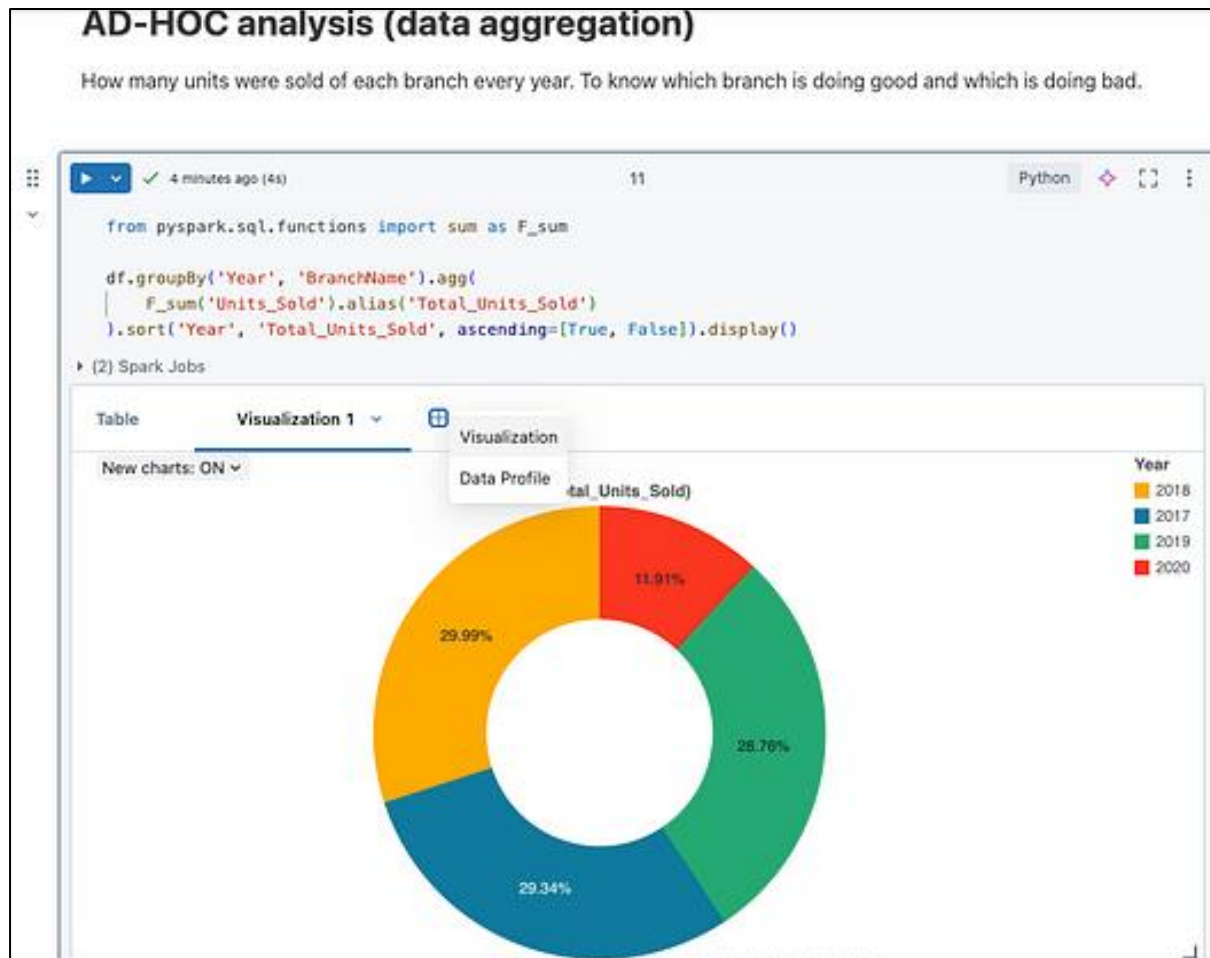
```
from pyspark.sql.functions import sum as F_sum

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

(2) Spark Jobs

	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

silver_notebook

File Edit View Run Help Last edited: 1/8/2025

Workspace

- CarsProject
- database_notebook
- silver_notebook

Data writing

```
df.write.format('parquet')\
    .mode('append')\
    .option('path', 'abfss://silver@datalakecarsale.dfs.core.windows.net/carsales')\
    .save()
```

(7) Spark Jobs

Then check that the parquet files were saved on Azure.

Home > Resource groups > RG_Azure_Car_Sales_Project > datalakecarsale | Containers >

silver

Container

Search

Upload Add Directory Refresh Rename Delete Change tier Acquire lease Break lease Give feedback

Authentication method: Access key (Switch to Microsoft Entra user account)

Location: silver / carsales

Search blobs by prefix (case-sensitive)

Show deleted objects

Name	Modified	Access tier	Archive status	Blob type	Size	Lease state
1						
committed_40509111653782272...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	318 B	Available
_started_4050911165378227275	1/8/2025, 12:02:05 PM	Hot (Inferred)		Block blob	0 B	Available
_SUCCESS	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	0 B	Available
part-00000-tid-405091116537822...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available
part-00001-tid-405091116537822...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available
part-00002-tid-405091116537822...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available

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]

	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	9684767	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 star 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.

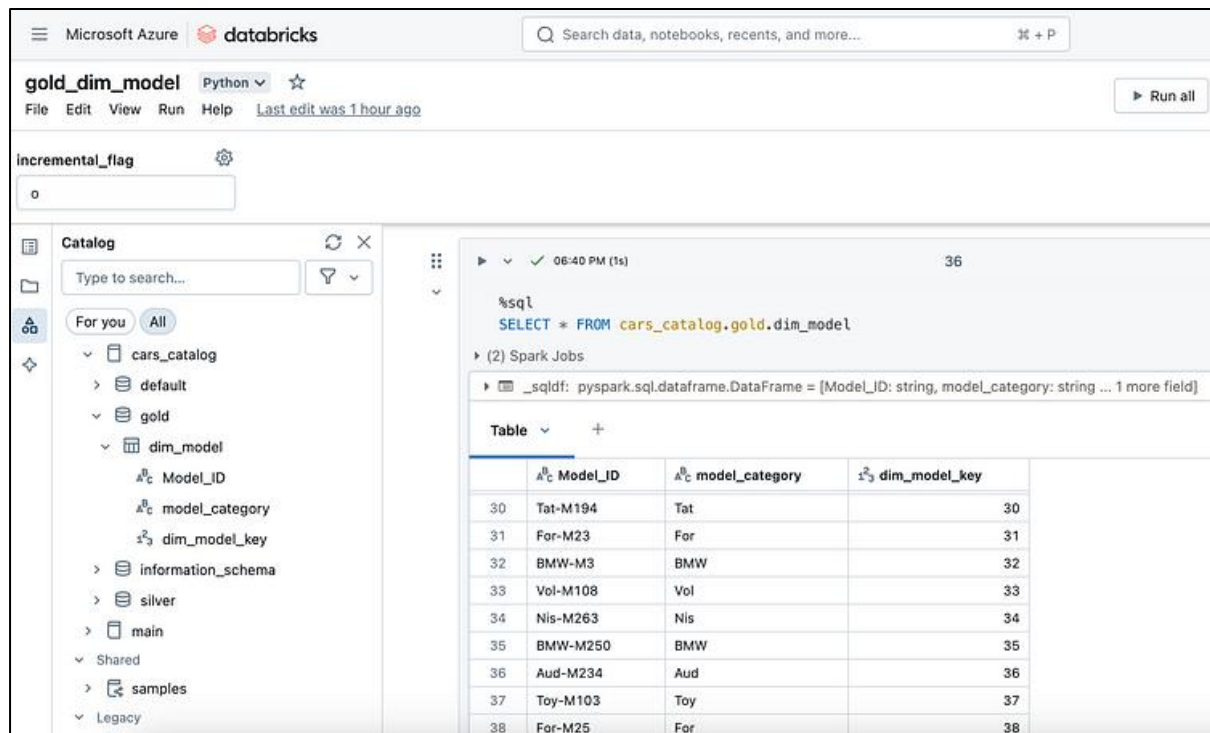
A detailed step-by-step guide is in this [Databricks Notebook \(PySpark\)](#)

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")
```

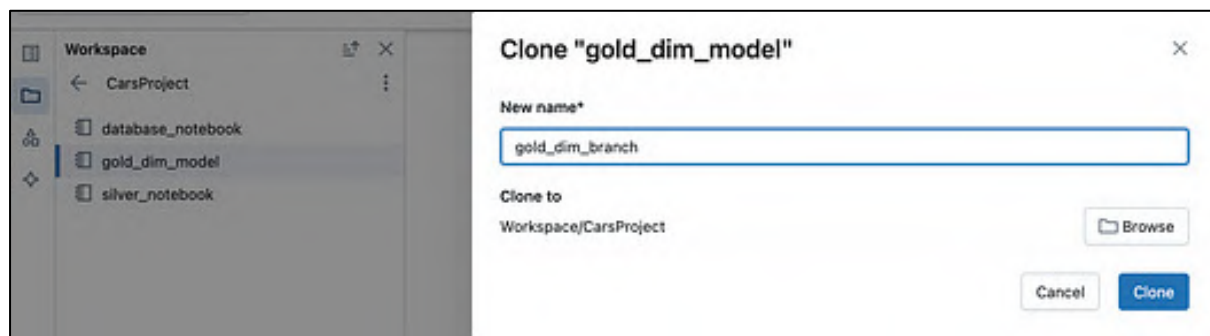
The final result should look like this in the dimension table:



The screenshot shows a Databricks notebook titled "gold_dim_model" in a workspace. The notebook is in Python mode and has a search bar at the top. The left sidebar shows a catalog with a tree view containing "cars_catalog", "default", "gold", "dim_model", "information_schema", "silver", "main", "Shared", "samples", and "Legacy". The "dim_model" table is selected, showing its schema with columns: "Model_ID", "model_category", and "dim_model_key". The main area displays a SQL query: `SELECT * FROM cars_catalog.gold.dim_model`. Below the query, it shows "(2) Spark Jobs" and a table of results. The table has 36 rows and 3 columns: "Model_ID", "model_category", and "dim_model_key".

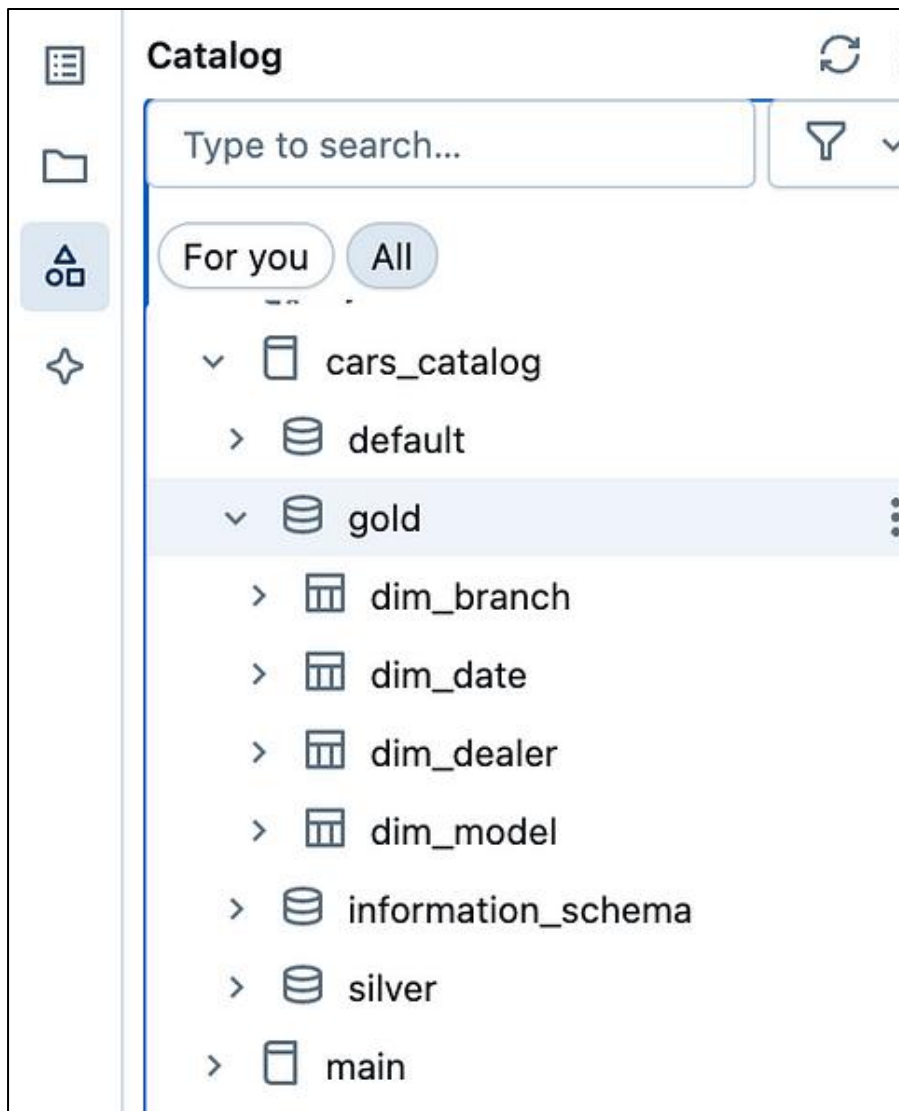
	Model_ID	model_category	dim_model_key
30	Tat-M194	Tat	30
31	For-M23	For	31
32	BMW-M3	BMW	32
33	Vol-M108	Vol	33
34	Nis-M263	Nis	34
35	BMW-M250	BMW	35
36	Aud-M234	Aud	36
37	Toy-M103	Toy	37
38	For-M25	For	38

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.



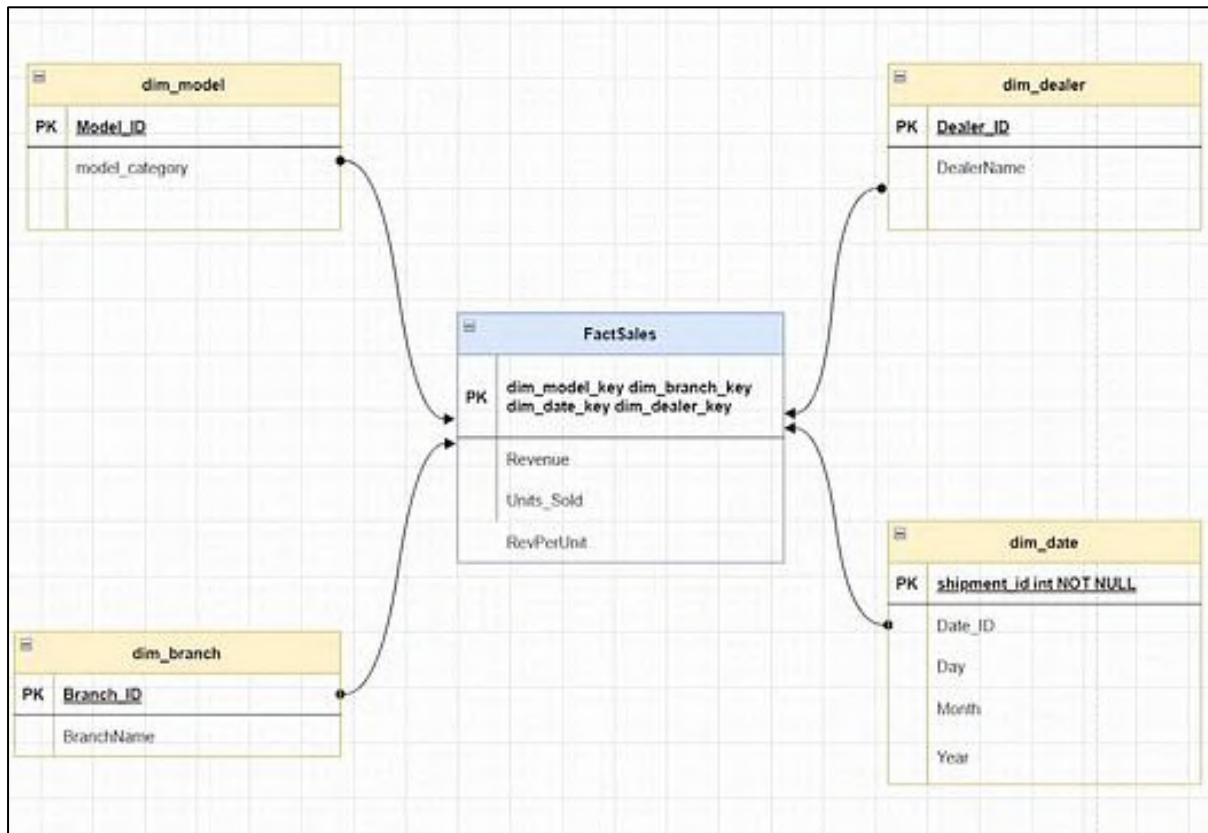
The screenshot shows the "Clone 'gold_dim_model'" dialog box in Databricks. The dialog has a "New name*" field with the value "gold_dim_branch". The "Clone to" field shows "Workspace/CarsProject". There are "Browse", "Cancel", and "Clone" buttons.

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'):
    deltable = DeltaTable.forName(spark, 'cars_catalog.gold.factsales')

    deltable.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')\n    .mode('Overwrite')\n    .option("path", "abfss://gold@datalakecarsale.dfs.core.windows.net/factsales")\n    .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.

Microsoft Azure | databricks

Search data, notebooks, recents, and more...

Workflows > Jobs >

Data-Modek ☆

Runs Tasks

Silver_Data

...space/CarsProject/silver_notebook

Rihab Feki's Cluster

Task name* Silver_Data

Type* Notebook

Source* Workspace

Path* /Workspace/CarsProject/silver_notebook

Cluster* Rihab Feki's Cluster DBR 15.4 LTS - Spark 3.5.0 - Scala 2.12

Jobs running on all-purpose clusters are considered all-purpose compute. [Learn more](#)

Cancel Create task

Add more tasks in this manner:

Microsoft Azure | databricks

Search data, notebooks, recents, and more...

Workflows > Jobs >

Data-Modek ☆

Runs Tasks

Silver_Data

...space/CarsProject/silver_notebook

Rihab Feki's Cluster

Task name* Silver_Data

Type* Notebook

Code

Notebook

Python script

Python wheel

JAR

Spark Submit

Clean Room notebook

SQL

Legacy dashboard

Query

Alert

SQL file

Data transformation

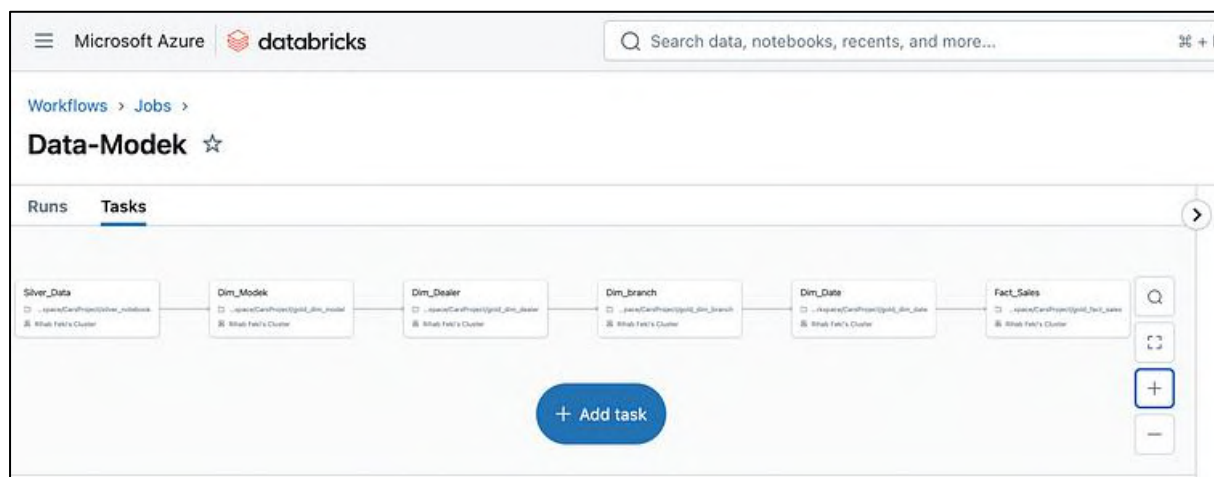
For the dimension model, make sure to configure a parameter of the incremental_flag at the stage of creating the task, as shown below:

The screenshot shows the 'Tasks' configuration page for a task named 'Dim_Modelk'. The task is a notebook located at '...space/CarsProject/gold_dim_model' and is running on 'Rihab Feki's Cluster'. It depends on the 'Silver_Data' task. The 'Run if dependencies' are set to 'All succeeded'. The 'Parameters' section is configured with a key-value pair: 'incremental_flag' with a value of '1'. The 'UI' and 'JSON' tabs are visible at the bottom right of the parameters section.

```
graph LR; Silver_Data[Silver_Data] --> Dim_Modelk[Dim_Modelk];
```

Key	Value
incremental_flag	1

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



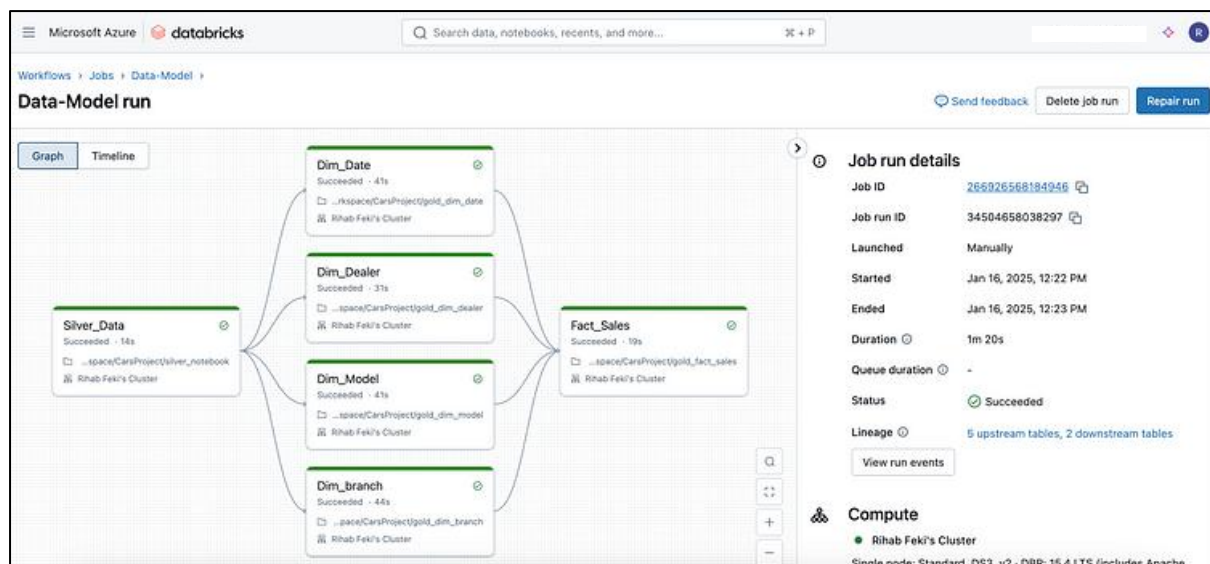
But to enhance the performance, we need to make all the DIM_tables tasks depend on the silver table and then make the fact_table depend on all the dimension tables by editing the Depends on option in the form.

The screenshot shows the 'Add task' dialog in the Databricks interface. The dialog is titled 'Data-Model' and has tabs for 'Runs' and 'Tasks'. The 'Tasks' tab is active. The dialog contains the following fields:

- Source***: A dropdown menu with the following options: Silver_Data, Dim_Dealer, Dim_Model, Dim_branch, and Fact_Sales. 'Silver_Data' is selected.
- Path***: A text input field.
- Cluster***: A dropdown menu.
- Depends on**: A dropdown menu with the following options: Silver_Data X, All succeeded, and All failed. 'Silver_Data X' is selected.
- Run if dependencies**: A dropdown menu with the following options: All succeeded, All failed, and All succeeded or failed. 'All succeeded' is selected.

At the bottom of the dialog are 'Cancel' and 'Save task' buttons.

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

The screenshot shows the Databricks SQL Editor interface. On the left is a sidebar with navigation options: New, Workspace, Recents, Catalog, Workflows, Compute, SQL, SQL Editor (selected), Queries, Dashboards, Genie, Alerts, Query History, SQL Warehouses, Data Engineering, Job Runs, and Data Ingestion. The main area displays a query editor with the following SQL query:

```
1 | SELECT * FROM cars_catalog.gold.factsales
```

Below the query editor, the 'Raw results' section shows a table with 9 rows and 7 columns. The columns are: Revenue, Units_Sold, dim_branch_key, dim_dealer_key, dim_model_key, and dim_date_key. The data is as follows:

	Revenue	Units_Sold	dim_branch_key	dim_dealer_key	dim_model_key	dim_date_key
1	13363978	2	17179869185	5502	155	17179
2	13363978	2	17179869185	5502	155	17179
3	13363978	2	17179869185	5502	155	8589
4	13363978	2	17179869185	5502	155	8589
5	13363978	2	17179869185	5502	155	
6	13363978	2	17179869185	5502	155	
7	13363978	2	17179869185	4714	155	17179
8	13363978	2	17179869185	4714	155	17179
9	13363978	2	17179869185	4714	155	8589

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

The screenshot shows the Databricks Compute interface. The left sidebar is the same as in the previous screenshot. The main area is titled 'Compute' and shows a table of clusters. The table has columns: State, Name, Policy, Runtime, Active m..., Active co..., Active DB..., Source, Creator, and Notebooks. The only cluster listed is 'Rihab Feki's Cluster' with a state of 'Idle' and a policy of 'Personal Co...'. The 'Start' button is visible at the bottom right of the cluster row.

State	Name	Policy	Runtime	Active m...	Active co...	Active DB...	Source	Creator	Notebooks
Idle	Rihab Feki's Cluster	Personal Co...	15.4	-	-	-	UI	Rihab Feki	-

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