

End-to-End Azure Data Engineering Project Pipeline with DataBricks

Here's a clean and engaging using Azure Data tech stack those tools 👏🔥

🚀 Modern Data Engineering Stack in Azure

Building scalable data platforms? These are the tools that power the journey 👏

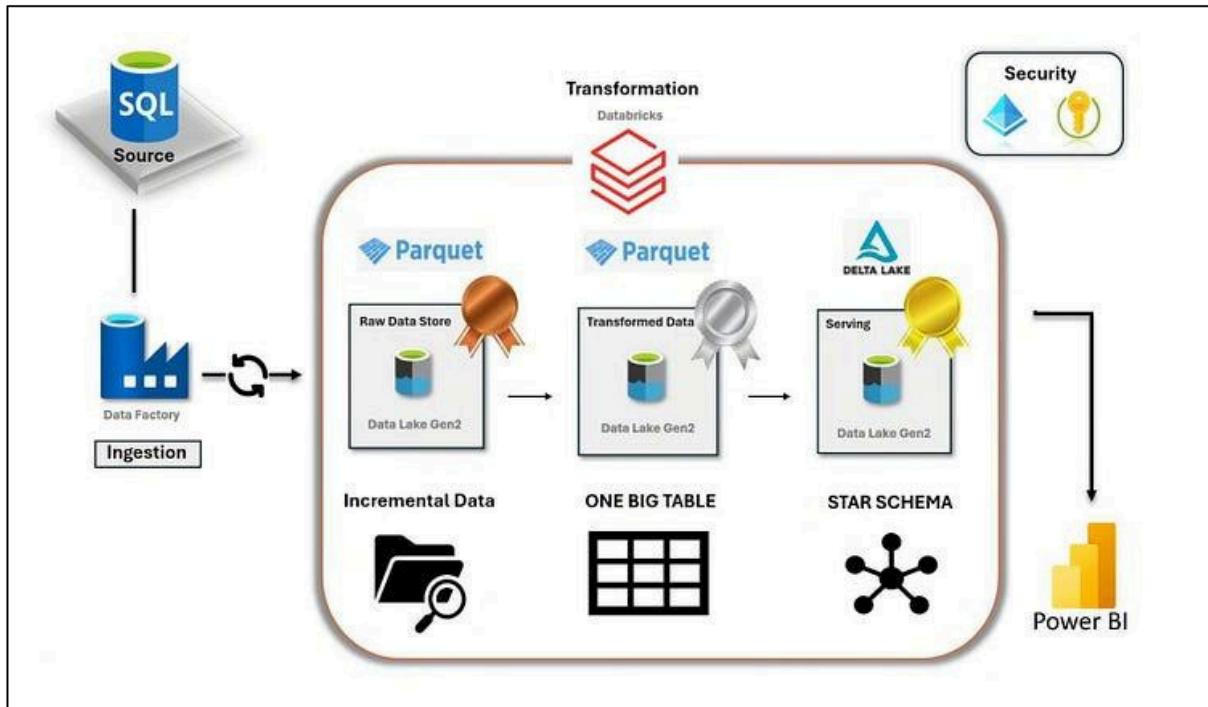
- ◆ **Azure SQL Database** – Store structured data with high availability and performance.
- ◆ **Azure Data Factory** – Orchestrate your pipelines, automate data movement, and integrate multiple sources.
- ◆ **Databricks** – Process big data, run distributed computing, and build machine learning workflows.
- ◆ **Delta Lake** – Enable ACID transactions, versioning, and high-quality data lakes with reliability.
- ◆ **Power BI** – Visualize insights, build dashboards, and empower decision-makers.

💡 Together, these tools enable:

- ✨ ETL/ELT workflows
- ✨ Real-time analytics
- ✨ Data governance & lineage
- ✨ Enterprise-grade analytics

📌 **Azure ecosystem = scalable, secure & future-ready.**

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



PROJECT ARCHITECTURE

In this project you will learn the following concepts:

- **Data Modelling**— Star Schema (Fact & Dimensions Modelling).
- Slowly Changing Dimensions handling & Change Data Capture (CDC).
- **Data Design Pattern:** Medallion 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 organisation, 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. At the top, there are tabs for 'Basics', 'Tags', and 'Review + create'. Below the tabs, a description explains what a resource group is: 'A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization.' A 'Learn more' link is provided. The 'Project details' section includes a 'Subscription' dropdown set to 'Azure subscription 1' and a 'Resource group' dropdown set to 'RG_Azure_Car_Sales_Project'. The 'Resource details' section includes a 'Region' dropdown set to '(Europe) Germany West Central'. The 'Copilot' button is visible in the top right corner.

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 top navigation bar shows the path: Home > Resource groups > RG_Azure_Car_Sales_Project > Marketplace > Storage account. The page title is 'Create a storage account'. It includes a note: 'manage your storage account together with other resources.' The 'Subscription' dropdown is set to 'Azure subscription 1' and the 'Resource group' dropdown is set to 'RG_Azure_Car_Sales_Project' with a 'Create new' option available. The 'Instance details' section includes fields for 'Storage account name' (set to 'carsalesdatalake'), 'Region' (set to '(Europe) Germany West Central'), 'Primary service' (set to 'Azure Blob Storage or Azure Data Lake Storage Gen 2'), 'Performance' (radio button selected for 'Standard: Recommended for most scenarios (general-purpose v2 account)'), and 'Redundancy' (set to 'Locally-redundant storage (LRS)'). At the bottom, there are 'Previous' and 'Next' buttons, and the 'Review + create' button is highlighted in blue.

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.

The screenshot shows the 'Create Data Factory' wizard in the Microsoft Azure portal. The 'Basics' tab is selected. The 'Subscription' dropdown is set to 'Azure subscription 1'. The 'Resource group' dropdown is set to 'RG_Azure_Car_Sales_Project'. Under 'Instance details', the 'Name' is 'df-car-sales', 'Region' is 'Germany West Central', and 'Version' is 'V2'. At the bottom, there are 'Previous' and 'Next' buttons, and a highlighted 'Review + create' button.

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.

The screenshot shows the 'Create SQL Database Server' wizard in the Microsoft Azure portal. It's the second step of a multi-step process. The 'Authentication method' section has 'Use both SQL and Microsoft Entra authentication' selected. The 'Set Microsoft Entra admin' section shows an email address 'ribipersonal@gmail.com#EXT#@ribipersonal@gmail.onmicrosoft.com' and an 'Admin Object/App ID: 4aa909f8-ce9b-413b-b770-b0016b08d14d'. A 'Set admin' button is present. Below, 'Server admin login' is 'sql-admin', 'Password' is masked, and 'Confirm password' is also masked. At the bottom right, there is a 'Create a Server' button.

Continue with the steps to create a managed Azure SQL Database

The screenshot shows the 'Create SQL Database' step in the Azure portal. On the left, under 'Product details', it says 'SQL database by Microsoft'. Under 'Estimated cost', it shows 'Storage cost 5.70 USD / month + Compute cost 0.000159 USD / vCore second'. The 'Terms' section contains legal text. The 'Basics' section lists database details: Subscription (Azure subscription 1), Resource group (RG_Azure_Car_Sales_Project), Region (Germany West Central), Database name (car-sales-sql-db), Server (new car-sales--db-server), and Authentication method (SQL and Microsoft Entra authentication). On the right, a 'Cost summary' panel shows 'General Purpose (GP_S_Gen5_1)' with a cost of 0.14 USD per GB and 41.6 GB selected. It also displays 'ESTIMATED STORAGE COST / MONTH' at 5.70 USD and 'COMPUTE COST / VCORE SECOND' at 0.000159 USD. A note states that serverless databases are billed in vCore seconds based on a combination of CPU and memory utilization.

In networking choose a public endpoint

Step 5: Create Containers in the Data Lake

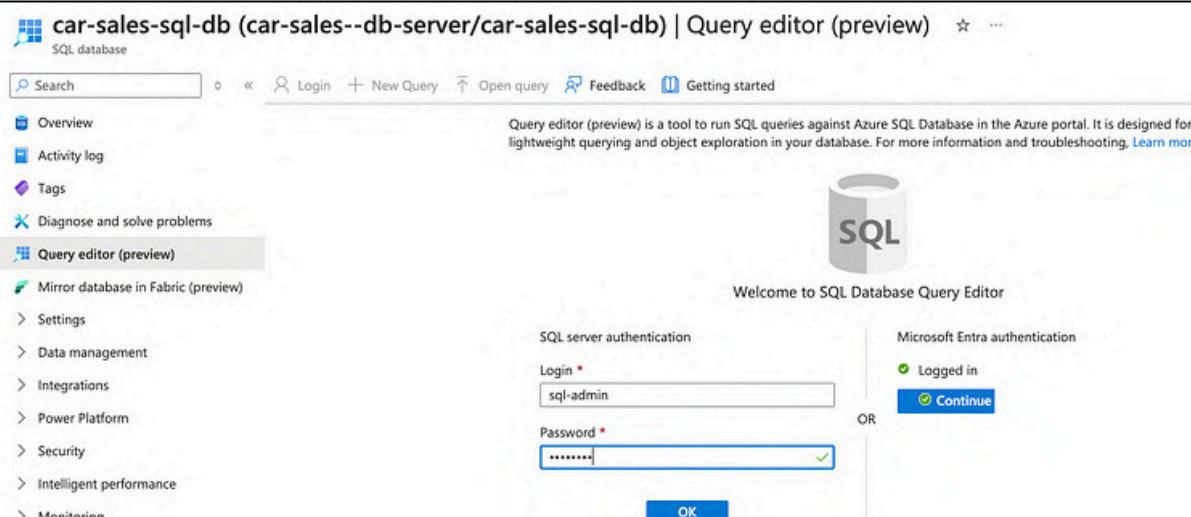
As we are following the Medallion Data Design pattern, create three containers:

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

The screenshot shows the 'Containers' blade for the 'datalakecarsales' storage account. The sidebar includes options like 'Diagnose and solve problems', 'Access Control (IAM)', 'Data migration', 'Events', 'Storage browser', 'Storage Mover', 'Partner solutions', 'Data storage', 'Containers' (selected), 'File shares', 'Queues', and 'Tables'. The main area shows a list of existing containers: '\$logs', 'bronze', 'gold', and 'silver', all set to 'Private'. To the right, a 'New container' dialog is open, prompting for a 'Name' (with a red asterisk) and 'Anonymous access level' (set to 'Private (no anonymous access)'). A note indicates that the access level is set to private and disabled on this storage account. An 'Advanced' section is partially visible.

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.



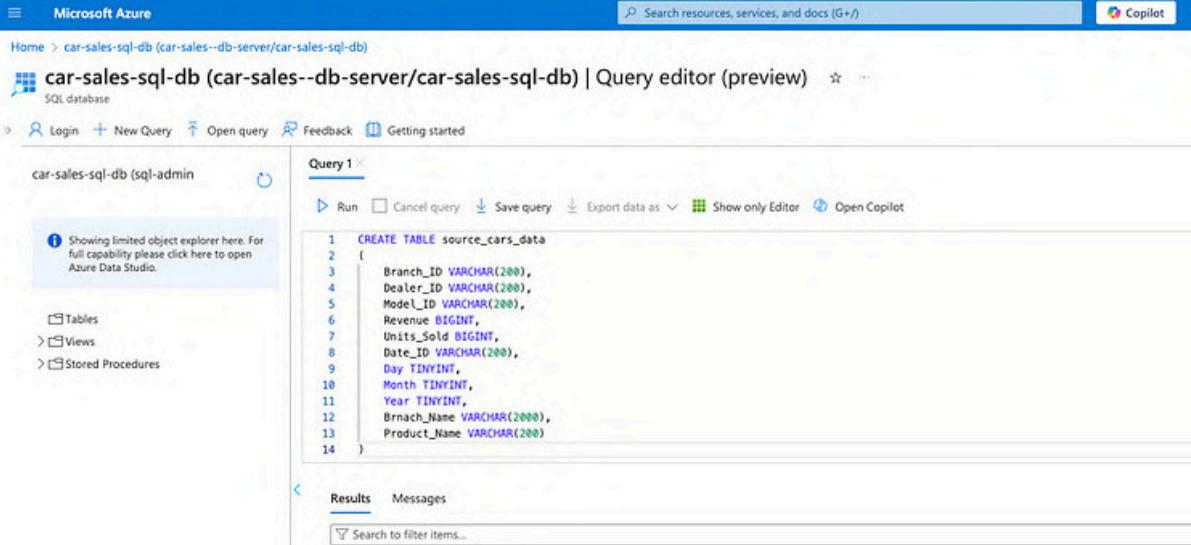
The screenshot shows the Azure SQL Database Query editor (preview) interface. The left sidebar has a 'Query editor (preview)' section selected. The main area displays a 'Welcome to SQL Database Query Editor' message and a login form. The 'SQL server authentication' section contains fields for 'Login' (sql-admin) and 'Password'. The 'Microsoft Entra authentication' section is shown below it. A large 'OK' button is at the bottom right of the form.

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	9664267	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 datastructure

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



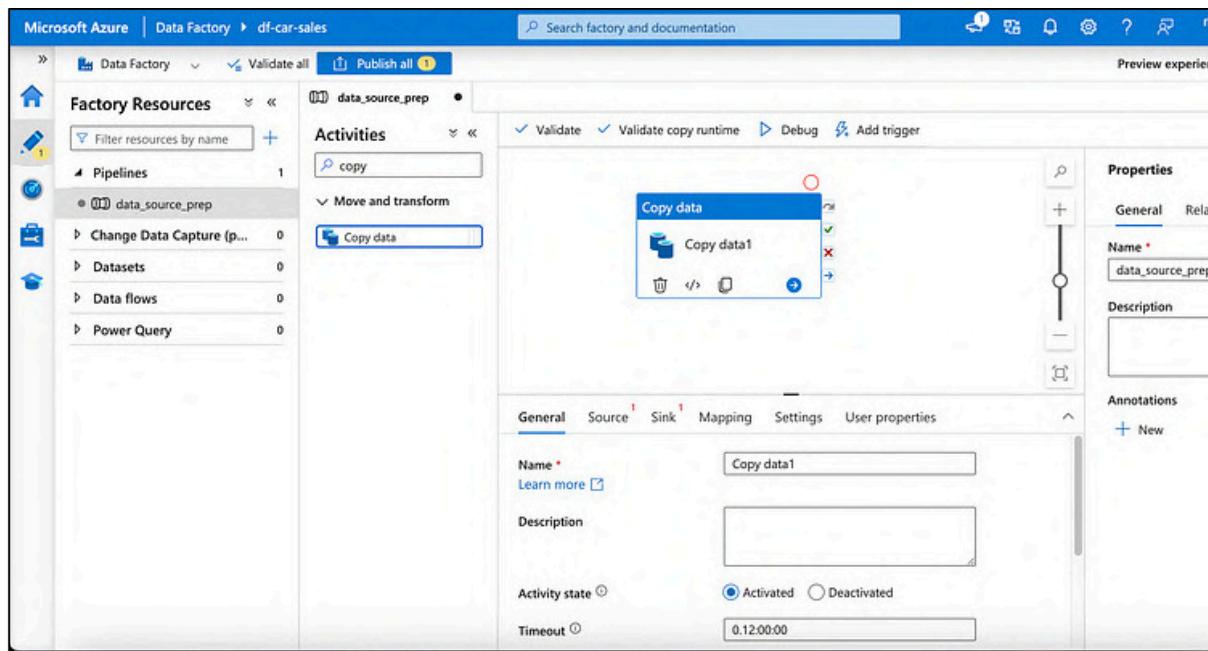
The screenshot shows the Azure SQL Database Query editor (preview) interface with a new query window titled 'Query 1'. The code pane contains the following 'CREATE TABLE' statement:

```
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    BranchName VARCHAR(200),
13    Product_Name VARCHAR(200)
14 )
```

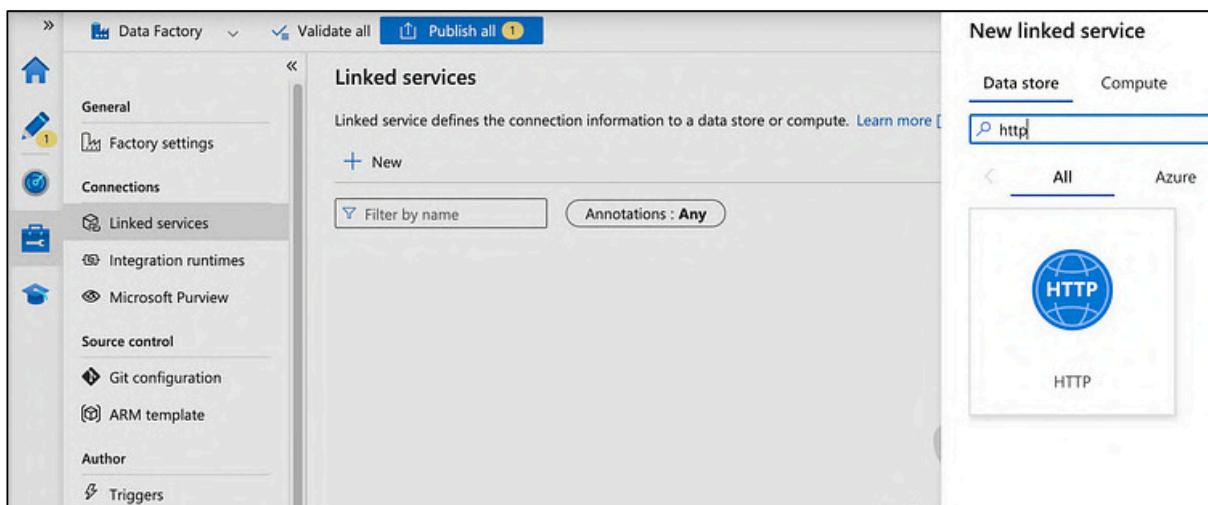
Create Table query

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

The screenshot shows the 'Linked services' section of the Azure Data Factory interface. On the left, a sidebar lists various settings like General, Connections, Source control, and Author. The 'Connections' section is expanded, showing 'Linked services'. A 'New' button is visible. The main pane displays a 'New linked service' configuration for an 'HTTP' type. The 'Name' field is set to 'ls_github'. The 'Base URL' field contains 'https://raw.githubusercontent.com/'. A warning message states: 'Information will be sent to the URL specified. Please ensure you trust the URL entirely'. Under 'Authentication type', 'Anonymous' is selected. The 'Create' button is at the bottom right, and a 'Test connection' link is also present.

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.

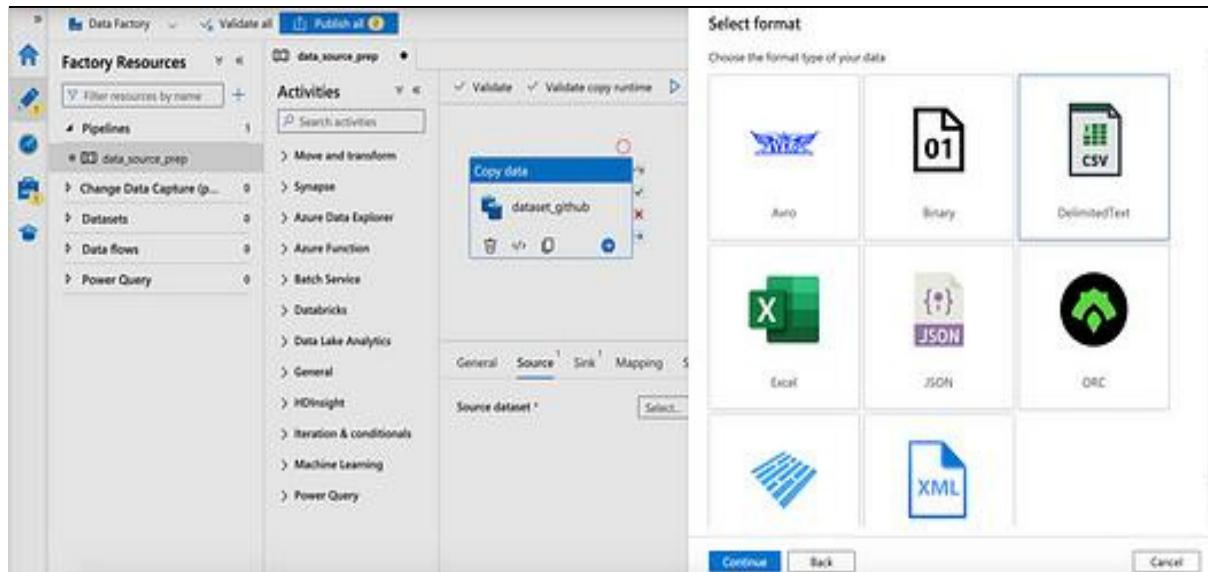
When you test the connection, you might get a connection error because of the firewall to protect the access to the database. To fix that error, navigate to the networking settings in the SQL server and click on “Enable Azure service to access this server” which you should find at the end of the page.

The screenshot shows the 'Linked services' section of the Azure Data Factory interface. The 'Connections' section is expanded, showing 'Linked services'. A 'New' button is visible. The main pane displays a 'New linked service' configuration for an 'Azure SQL Database' type. The 'Account selection method' is set to 'From Azure subscription'. The 'Azure subscription' dropdown shows 'Azure subscription 1 (acfdd59e-5b83-407d-bf67-033b48ee5c19)'. The 'Server name' is 'car-sales-db-server', and the 'Database name' is 'car-sales-sql-db'. The 'Authentication type' is 'SQL authentication'. The 'User name' is 'sql-admin', and the 'Password' field is filled with '*****'. The 'Create' button is at the bottom right, and a 'Test connection' link is also present.

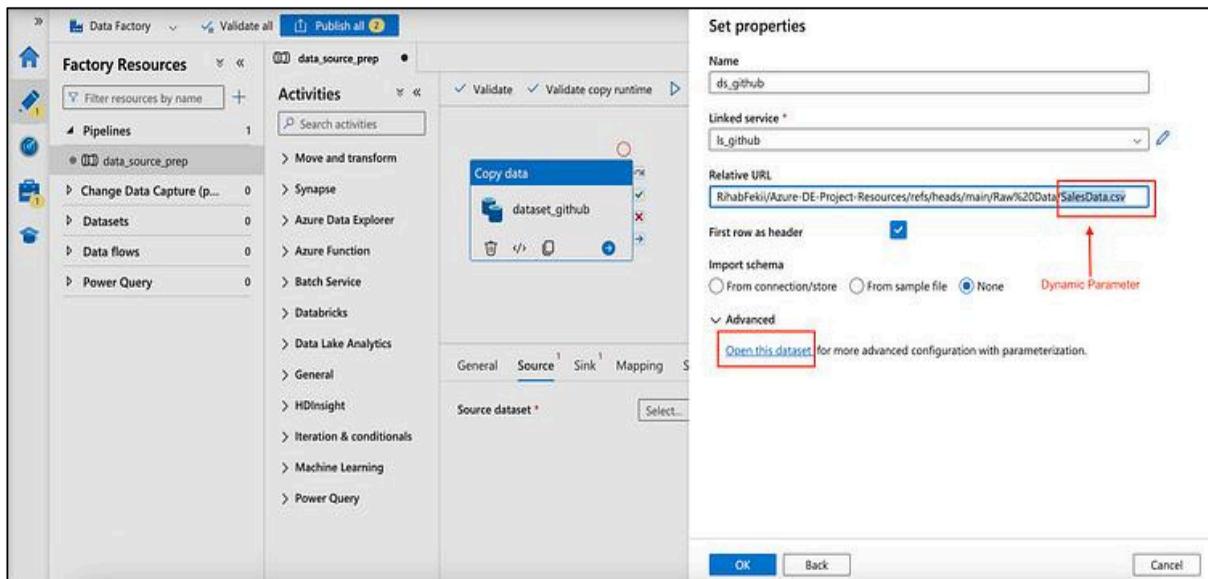
Dynamic ETL: parametrized dataset

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

The screenshot shows the 'Factory Resources' pane on the left with 'Datasets' selected. In the main area, 'ds_github' is selected. The 'Parameters' tab is active, showing a table with one row: 'file_name' (String type, Default value: Value). A tooltip indicates 'Added the parameter 'file_name''.

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.

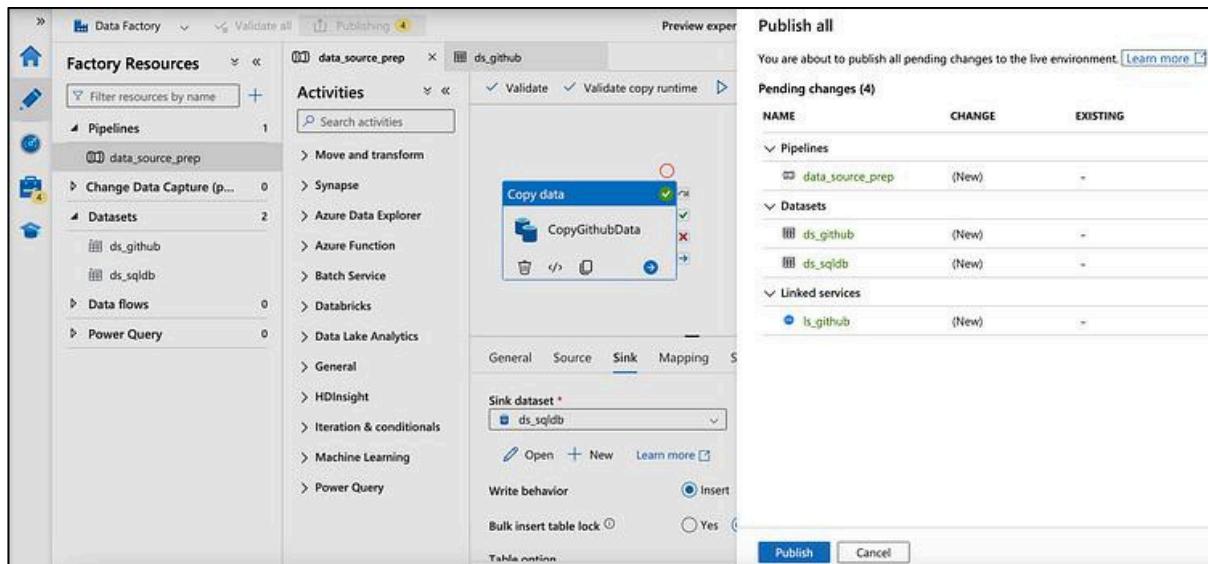
The screenshot shows the 'Factory Resources' pane on the left with 'Datasets' selected. In the main area, 'ds_github' is selected. The 'Connection' tab is active, showing the 'Linked service' dropdown set to 'ls_github'. The 'Relative URL' field contains the expression \${dataset().file_name}, which is highlighted with a red box. The 'Pipeline expression builder' dialog is open, showing the expression and a parameters section with 'file_name' selected.

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.

The screenshot shows the 'Factory Resources' pane on the left with 'Pipelines' selected. In the main area, 'data_source_prep' is selected. The 'Activities' tab is active, showing a list of activities including 'Move and transform', 'Synapse', 'Azure Data Explorer', etc. A 'Copy data' activity named 'CopyGitHubData' is selected. The 'Sink' tab of the activity configuration is open, showing the 'Sink dataset' dropdown set to 'Select...'. To the right, a 'New dataset' panel is open, showing options for selecting a data store, with 'azure sql' selected. Below it, 'All' is selected in a tab bar, and two options are shown: 'Azure SQL Database' and 'Azure SQL Database Managed Instance'.

Sinkconfig

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.

The screenshot shows the Azure Data Studio Query editor (preview) interface. The left sidebar includes options like Overview, Activity log, Tags, Diagnose and solve problems, Query editor (preview) (which is selected), Mirror database in Fabric (preview), Settings, Data management, Integrations, Power Platform, Security, Intelligent performance, Monitoring, Automation, and Help. The main area has a 'Tables' section showing the 'dbo.source_cars_data' table with columns: Branch_ID, Dealer_ID, Model_ID, and Revenue. A query 'Query 1' is running, displaying the following SQL code:

```
1 select * from [dbo].[source_cars_data]
```

The 'Results' tab shows the query output:

Branch_ID	Dealer_ID	Model_ID	Revenue
BR0001	DLR0001	BMW-M1	13363978
BR0003	DLR0228	Hon-M218	17376468
RR/NN4	DR/NN0R	Tat-M188	9664767

A message at the bottom indicates 'Query succeeded | 0s'.

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 portal interface for a SQL database named 'car-sales-sql-db'. The left sidebar has 'Query editor (preview)' selected. In the main area, 'Query 2' is active, displaying the following T-SQL code:

```
1 CREATE TABLE watermark_table
2 (
3     last_load Varchar(200)
4 )
5
6 SELECT min(Date_ID) FROM [dbo].[source_cars_data]
7
8 INSERT INTO [dbo].[watermark_table]
9 VALUES('DT0000')
```

The results pane shows the output 'DT0000'.

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

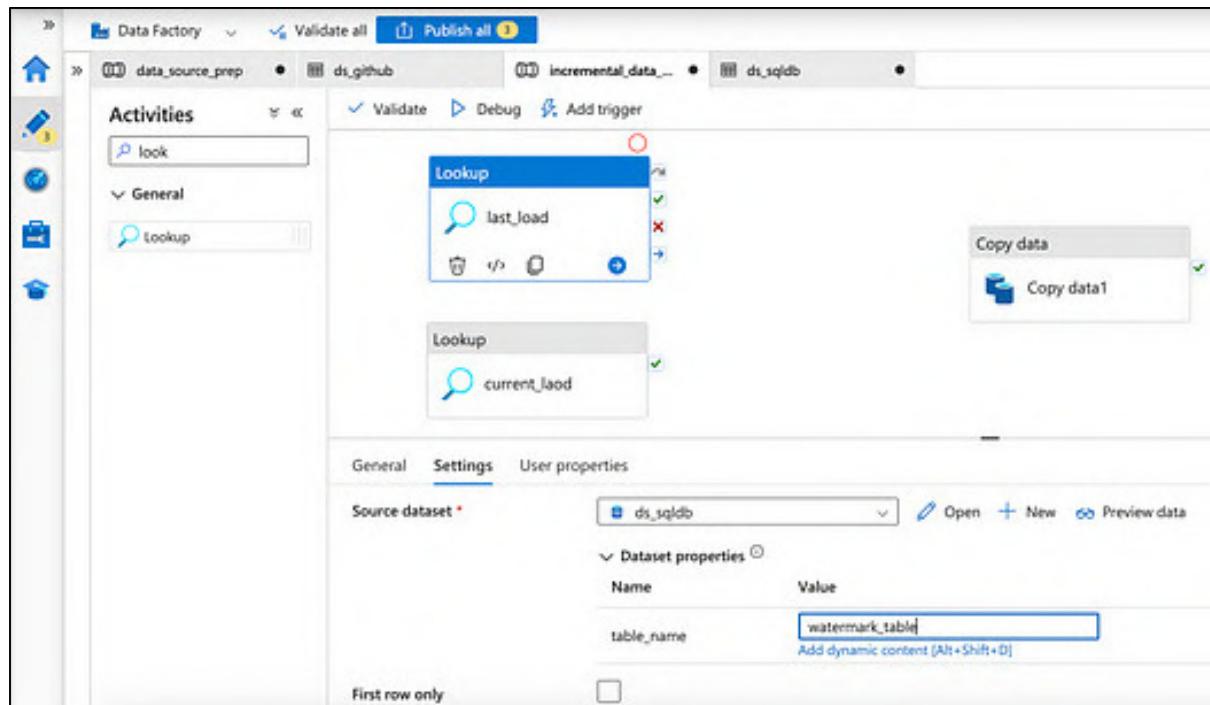
Now create a stored procedure to update this value again & again in the watermark table.

The screenshot shows the Azure portal interface for the same SQL database. The left sidebar has 'Query editor (preview)' selected. In the main area, 'Query 2' is active, displaying the following T-SQL code:

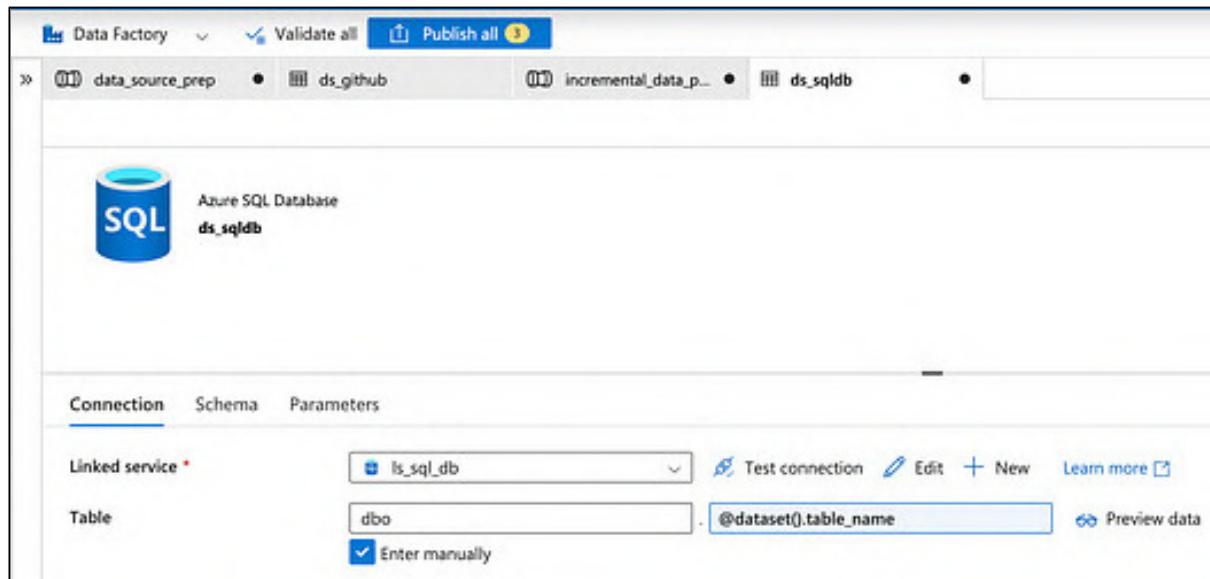
```
1 CREATE PROCEDURE UpdateWatermarkTable
2     @lastload VARCHAR(200)
3 AS
4 BEGIN
5     -- Start the transaction
6     BEGIN TRANSACTION;
7
8     -- Update the incremental column in the table
9     UPDATE watermark_table
10    SET last_load = @lastload
11
12    COMMIT TRANSACTION;
13 END;
```

The results pane shows the output 'Query succeeded: Affected rows: 0'.

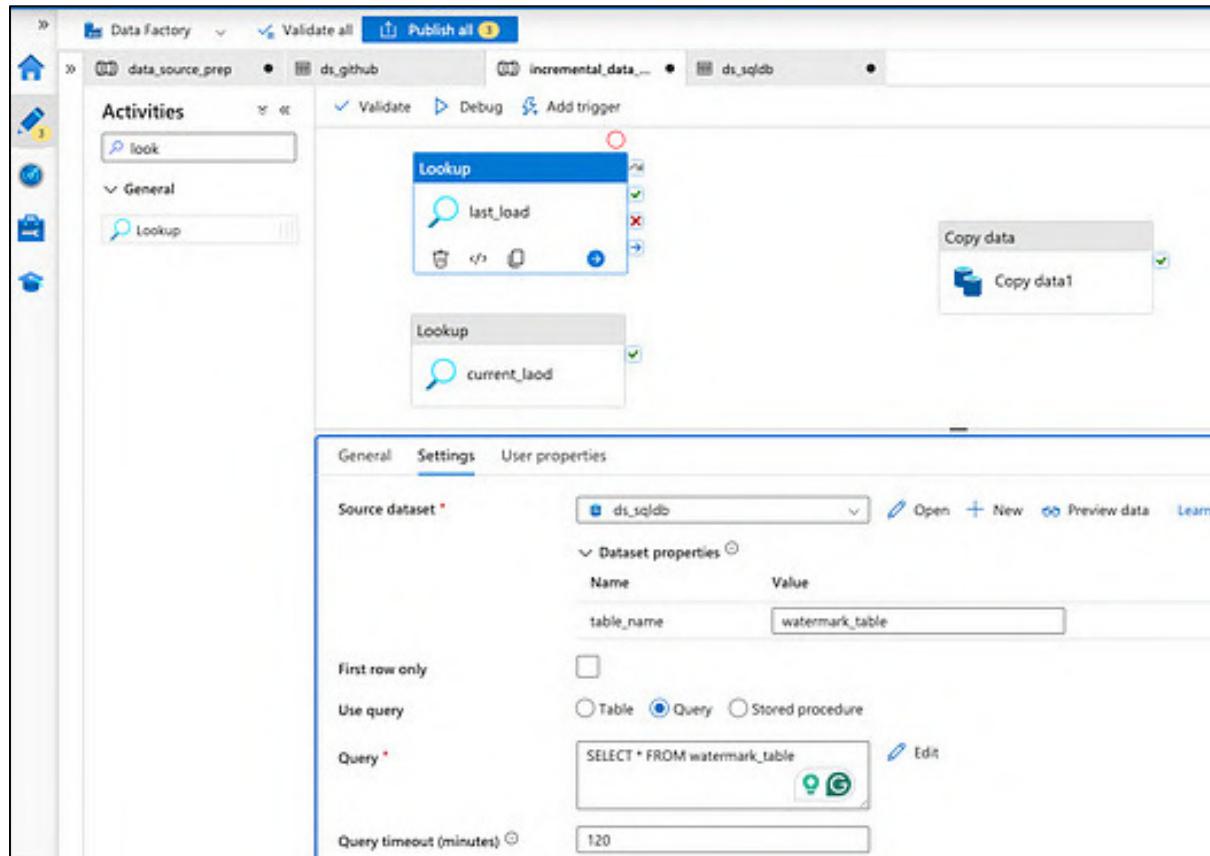
The next step is to add two activities “Lookup”, one that captures the last_load date and the other that captures the current_load date.



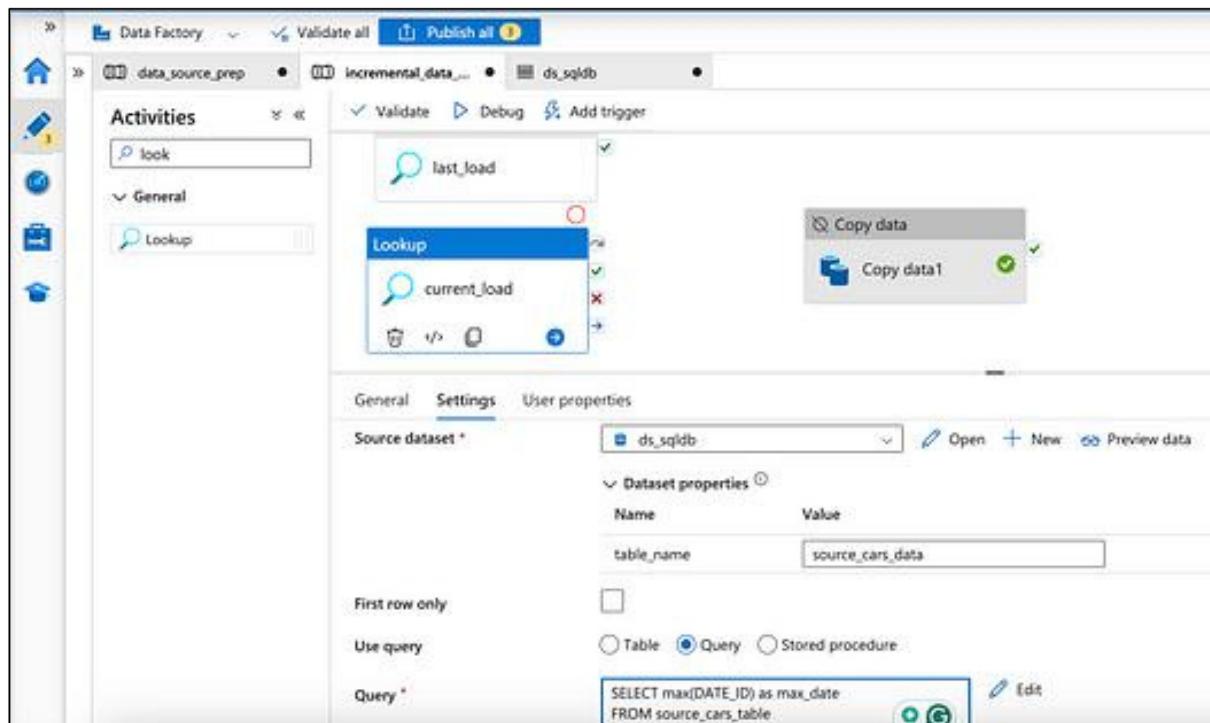
In the configuration of the lookup, select the dataset in the settings tab and then create a parameter for the table_name (which will be used in both Lookups).



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

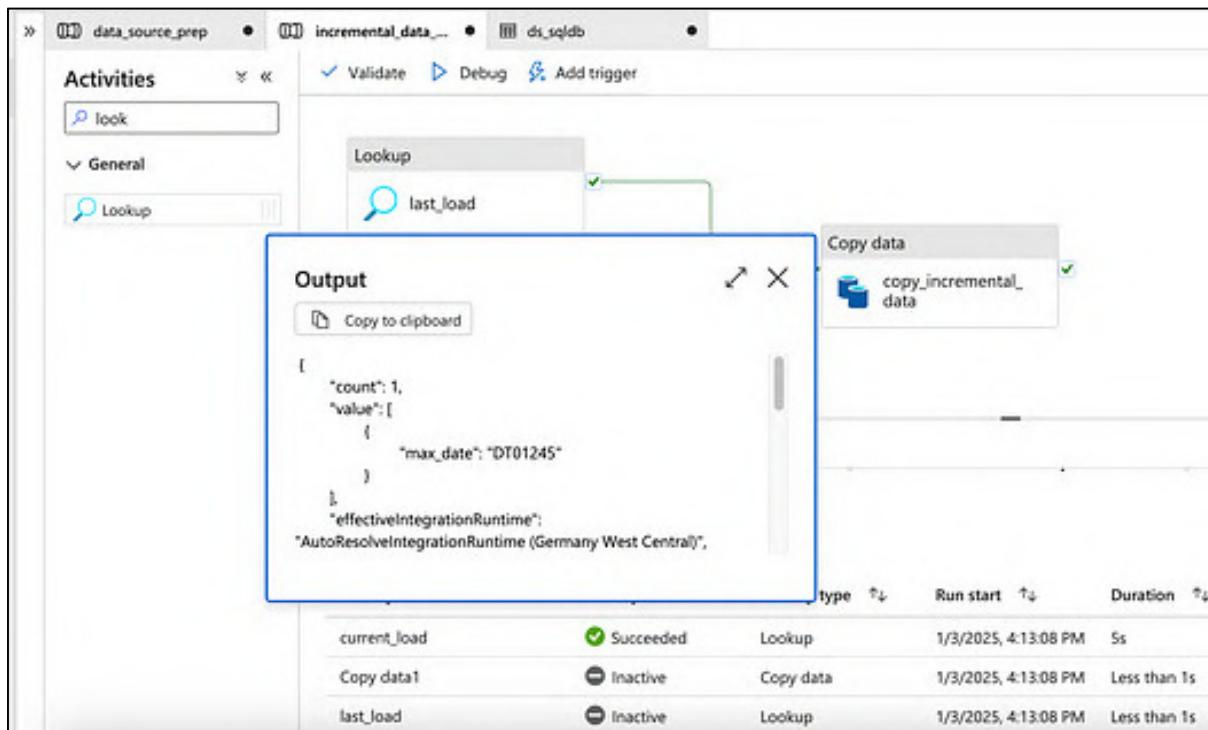


Then, configure the settings of the Lookup "current_load" as shown below.

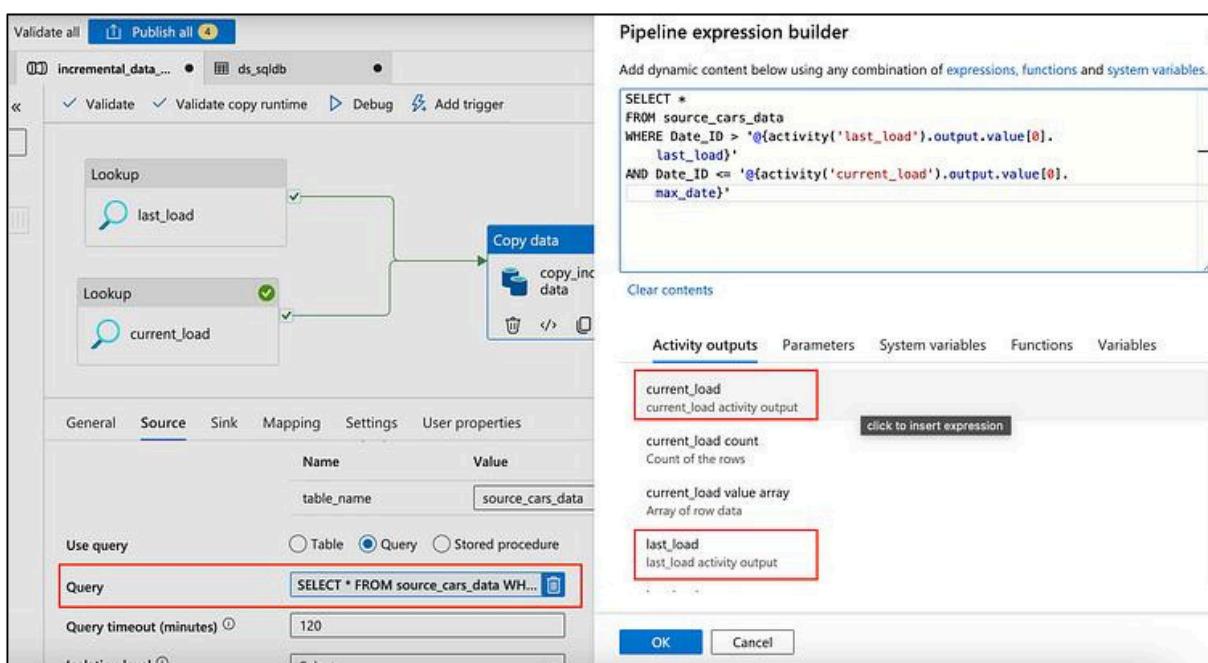


Then deactivate the other Activities and click on Debug to check the output of that activity. Afterward, connect the two lookups to the copy data activity on success by dragging the green check button.

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:

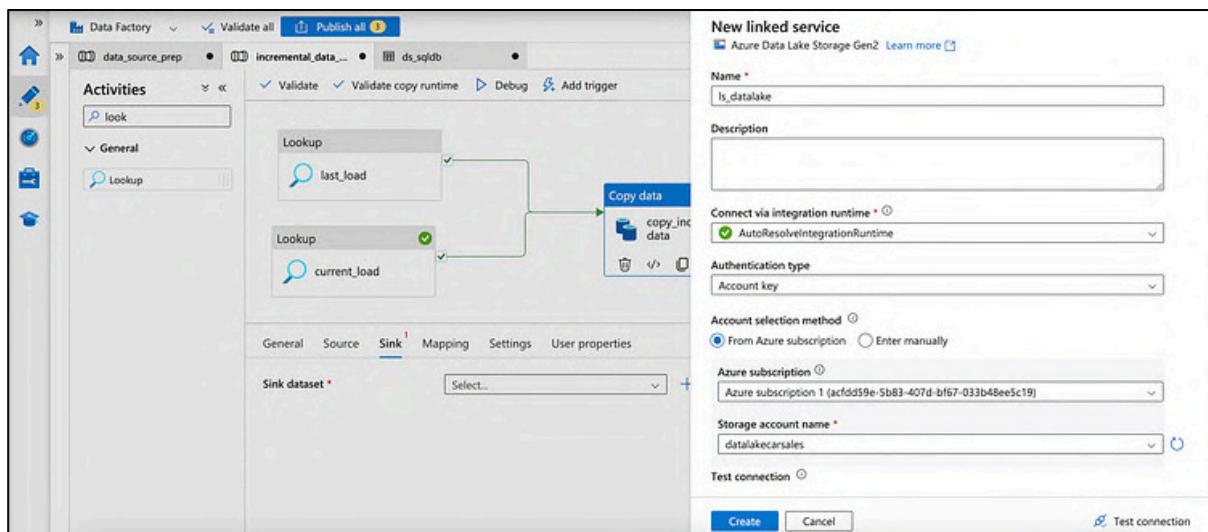


To get the max_date we need to write output.value[0].max_date and we do the same with output.value[0].last_load to be replaced in the parameter.

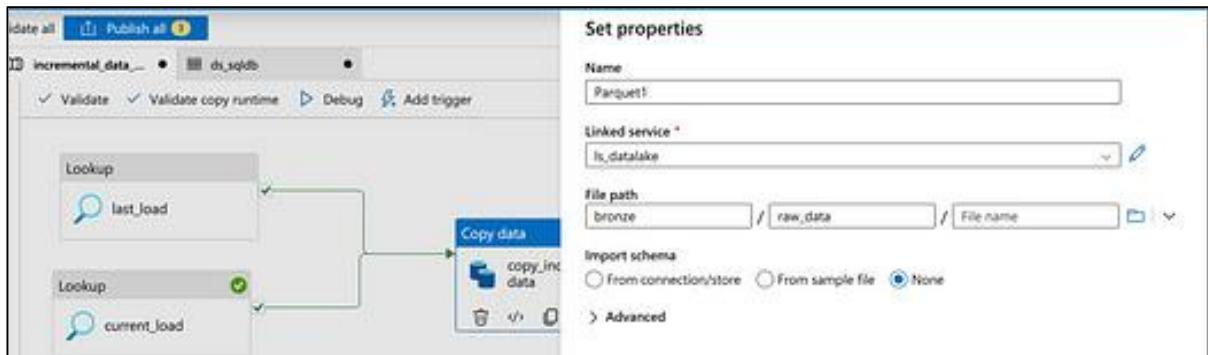


The next step is to add the sink information and at this level, we want to save the data to the Datalake (Bronze layer).

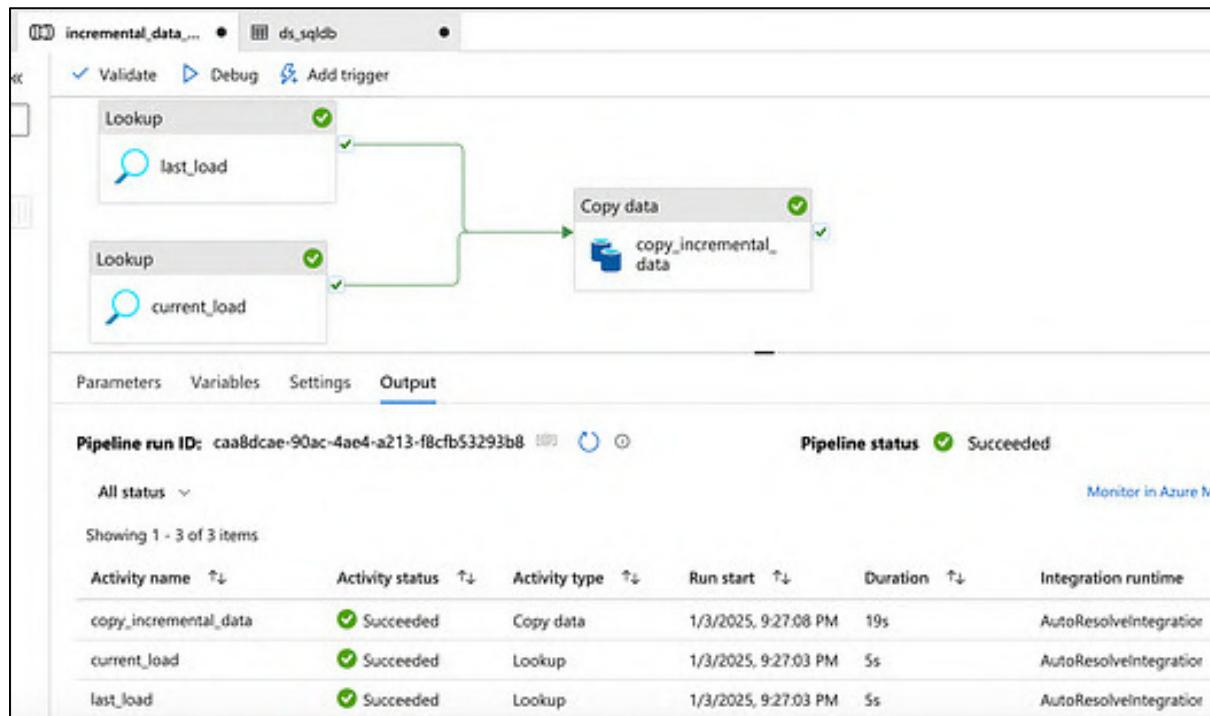
The data source would be Azure Datalake Storage Gen 2. Choose the format Parquet to save the data. Call the dataset ds_bronze create a linked service and link the Datalake we created at the beginning.



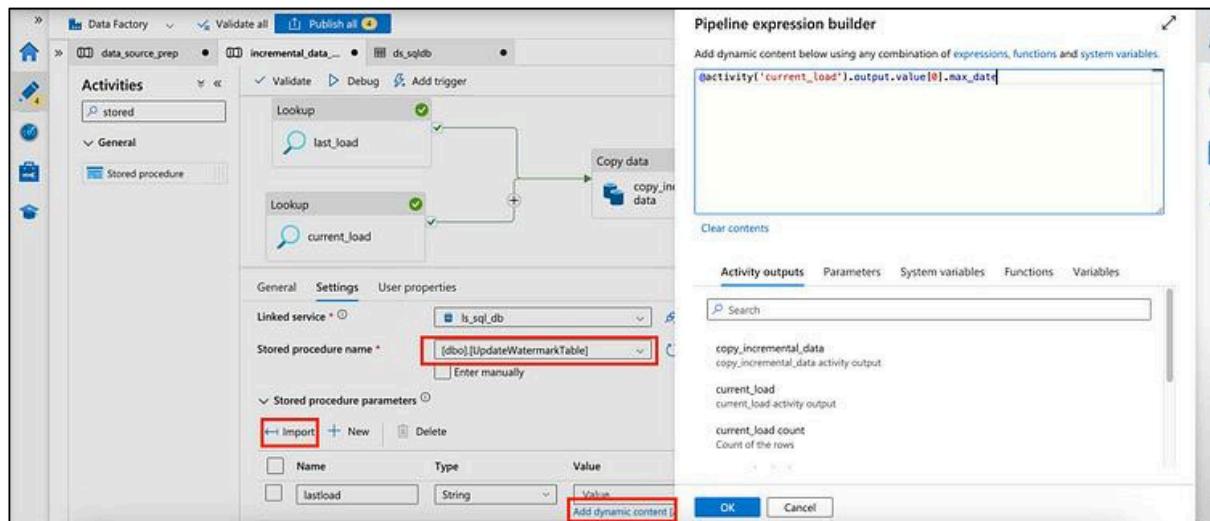
Then the last step in the sink is to set the properties, as follows:



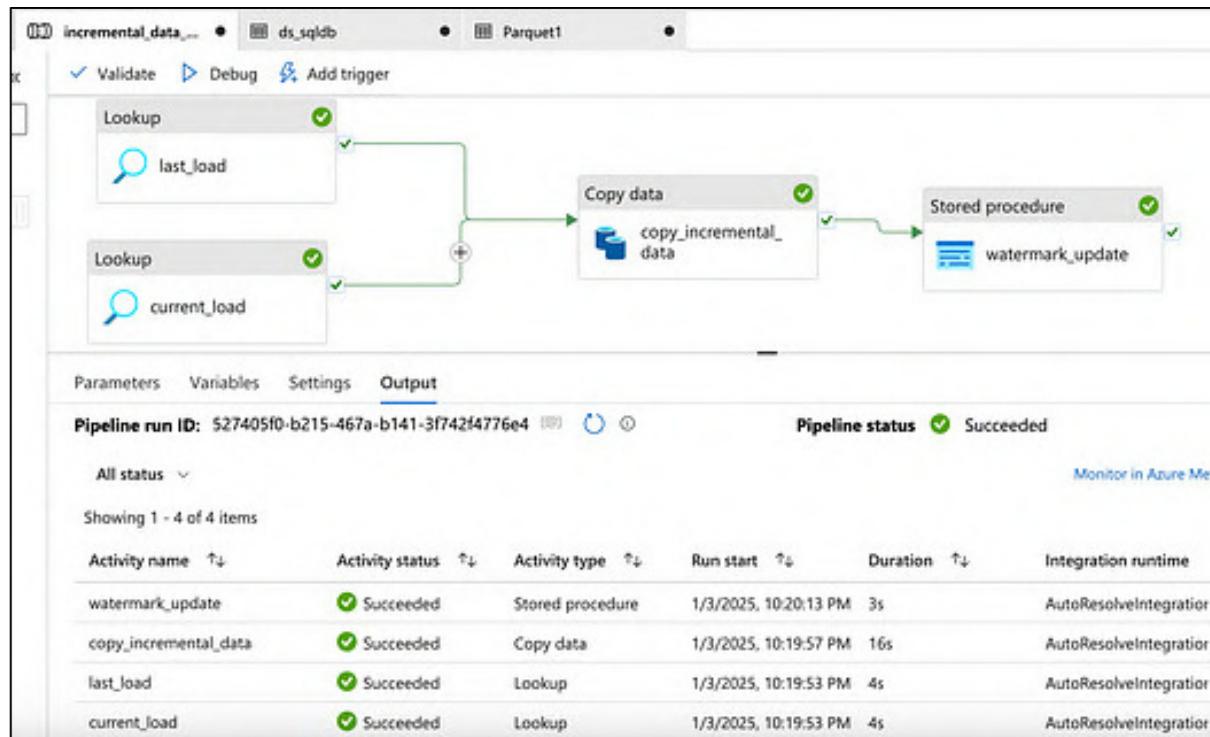
Then click on Debug to test the pipeline.



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.



The screenshot shows the Azure SQL Database Query editor. The left sidebar lists database objects: 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 shows the database 'car-sales-sql-db' with a connection to 'car-sales-sql-db (sql-admin)'. A message indicates a limited object explorer; clicking it opens Azure Data Studio. The 'Tables' section shows 'dbo.source_cars_data', 'dbo.watermark_table' (with a note about 'last_load'), and 'Views'. The 'Query 1' tab contains the SQL query: 'select * from [dbo].[watermark_table]'. The results pane shows the output:

last_load
DT01245

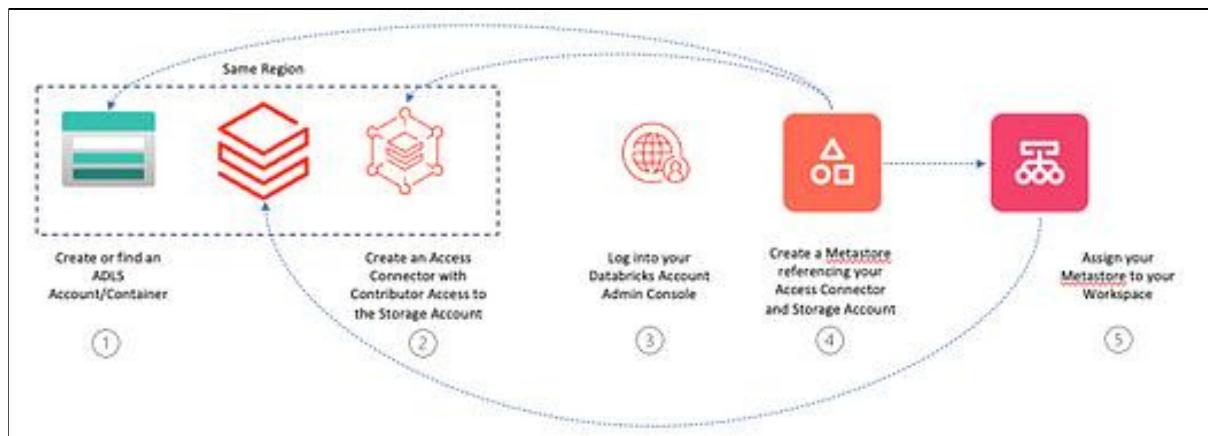
At last, make sure to publish the pipelines in order to save the work after we validated all the steps.

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.

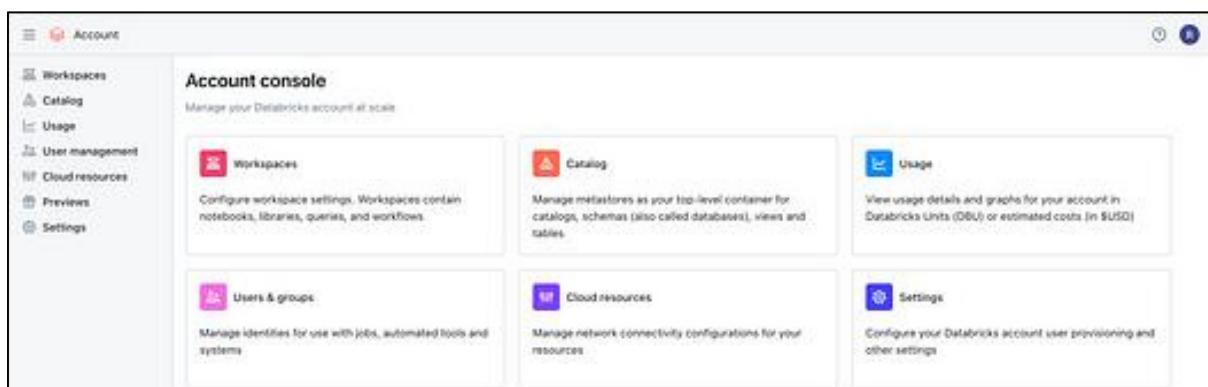
Step 1: Create a Unity Metastore

We will be following these steps:



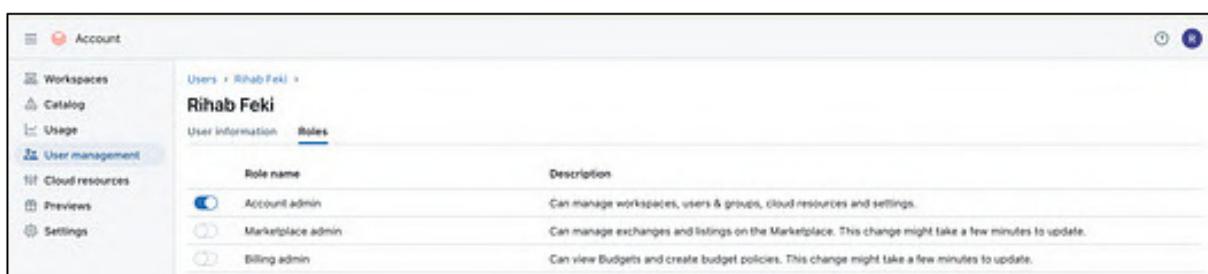
To create Compute, we must attach the Databricks workspace to the Unity Catalog. But to be able to create a Unity Metastore, we need to do that from the admin console.

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.azure.databricks.net/> (by resetting the password).



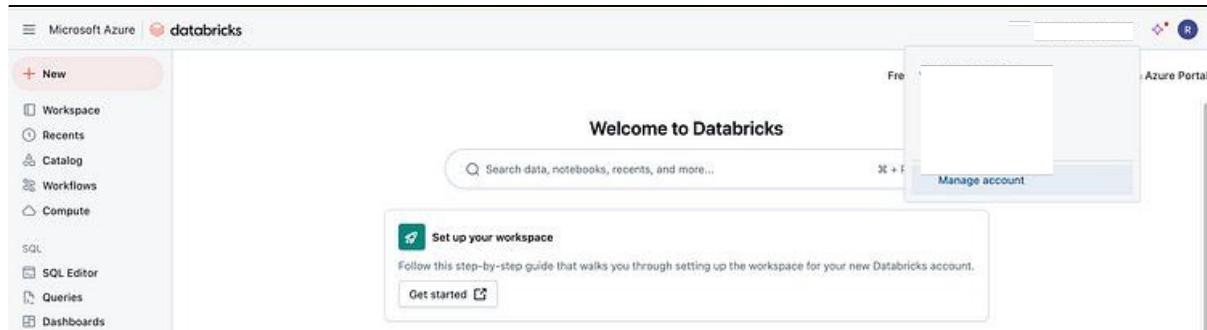
The Databricks admin console

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



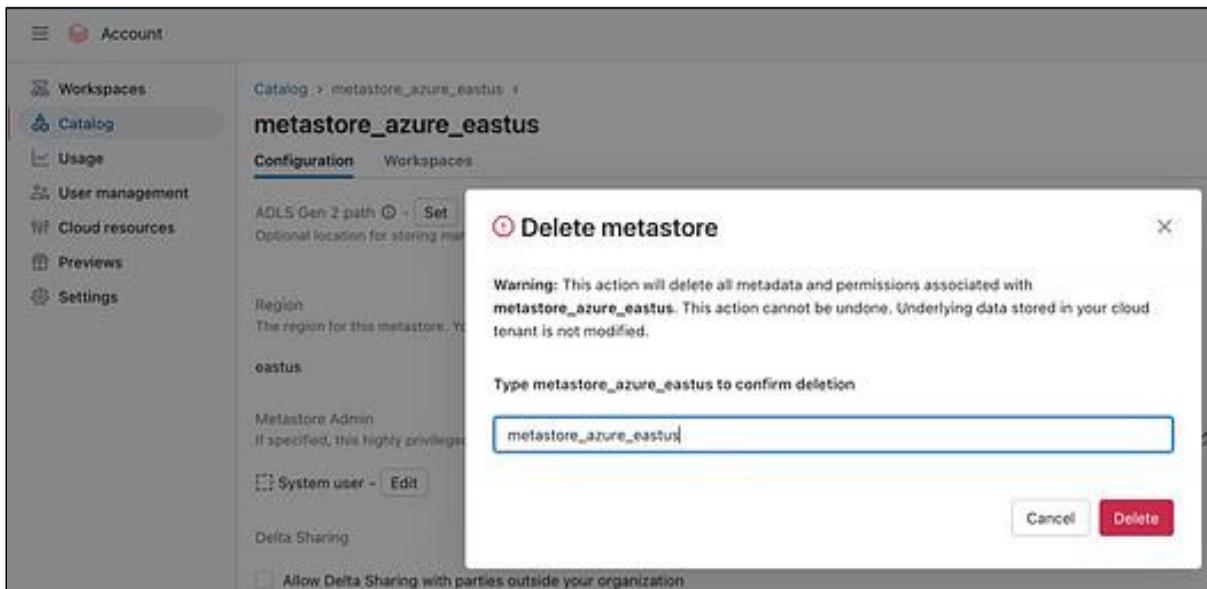
click in Account admin

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.

The screenshot shows the 'Create metastore' wizard in the Databricks Admin Console. The left sidebar has a 'Catalog' tab selected. The main area shows two steps: '1 Create metastore' and '2 Assign to workspaces'. Step 1 is active. It includes fields for 'Name' (set to 'carsalesmetastore'), 'Region' (set to 'westus'), 'ADLS Gen 2 path (optional)' (set to 'unity-metastore@datalakecarsale.dfs.core.windows.net/'), and 'Access Connector Id' (set to '/subscriptions/acfdd59e-5b83-407d-bf67-033b48ee5c19/resourceGroups/RG_Azure_Car_Sa'). A note below the region says: 'Select the region for your metastore. You will only be able to assign workspaces in this region to this metastore.'

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

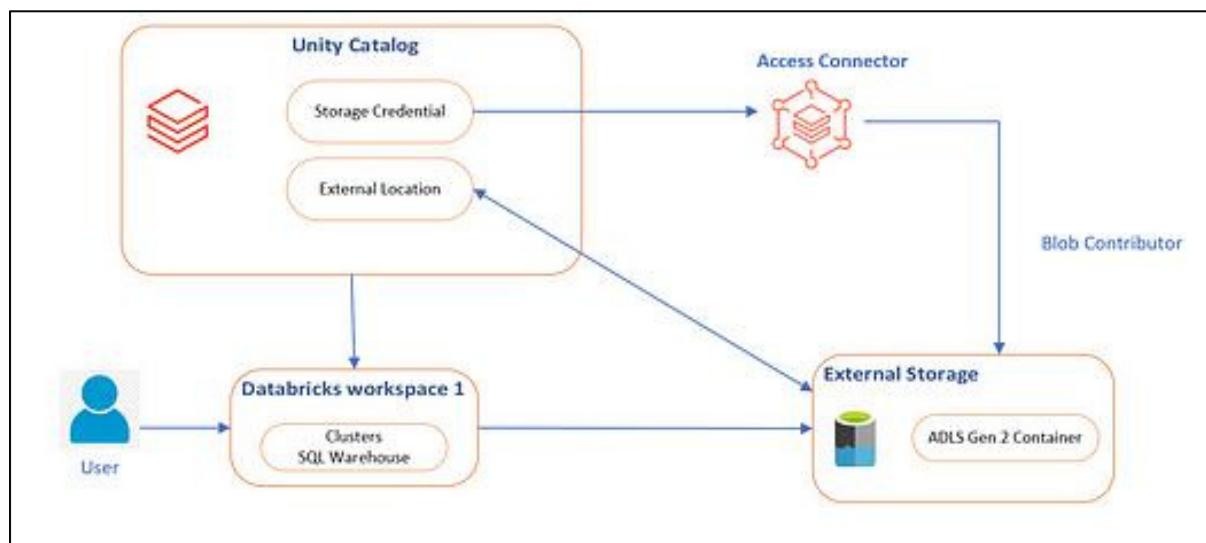
<container_name>@<storage_account_name>.dfs.core.windows.net/<path>

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

The screenshot shows the 'Containers' blade in the Azure Storage Account 'datalakecarsale'. The left sidebar has a 'Containers' tab selected. The main area lists existing containers: 'Slogs', 'bronze', 'gold', and 'silver'. A 'New container' dialog is open on the right, showing a 'Name' field set to 'unity-metastore'. A note below it says: 'The access level is set to private because anonymous access is disabled on this storage account.' An 'Advanced' section is partially visible at the bottom of the dialog.

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 * ⓘ

Resource group * ⓘ

Instance details

Name * ⓘ

Region * ⓘ

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.

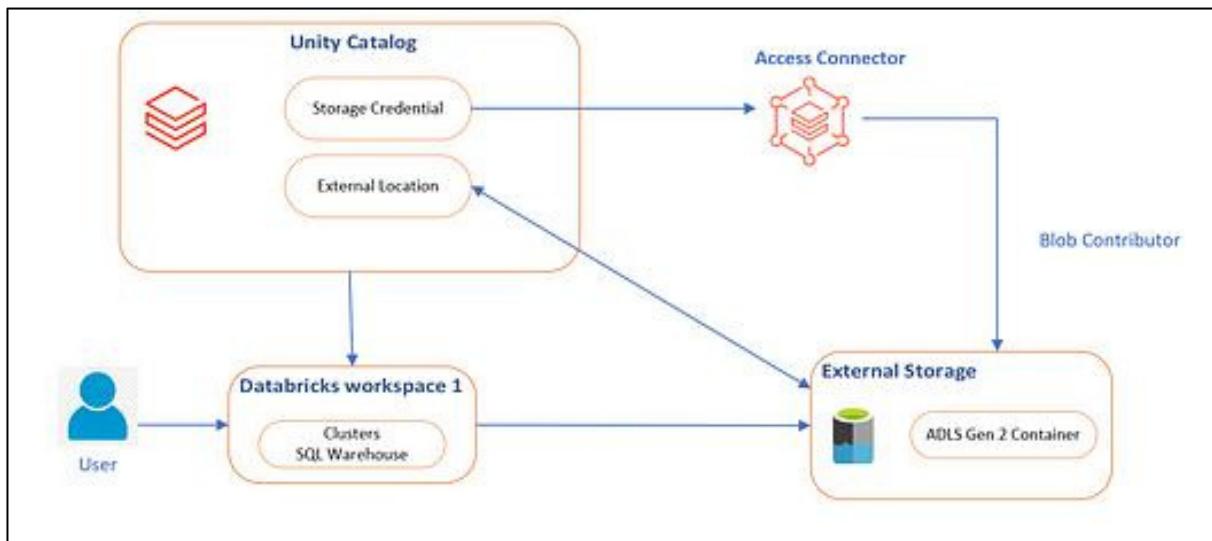
Name	Status	Pricing tier	Resource group	Region	Created	Metastore
AzureDatabricksWS	Running	Premium	RG_Databricks	eastus	last Saturday at 2:59...	metastore_azu...
carsdatabricks	Running	Premium	RG_Azure_Car_Sale...	germanywestcentral	last Saturday at 9:51...	metastore_azu...

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.

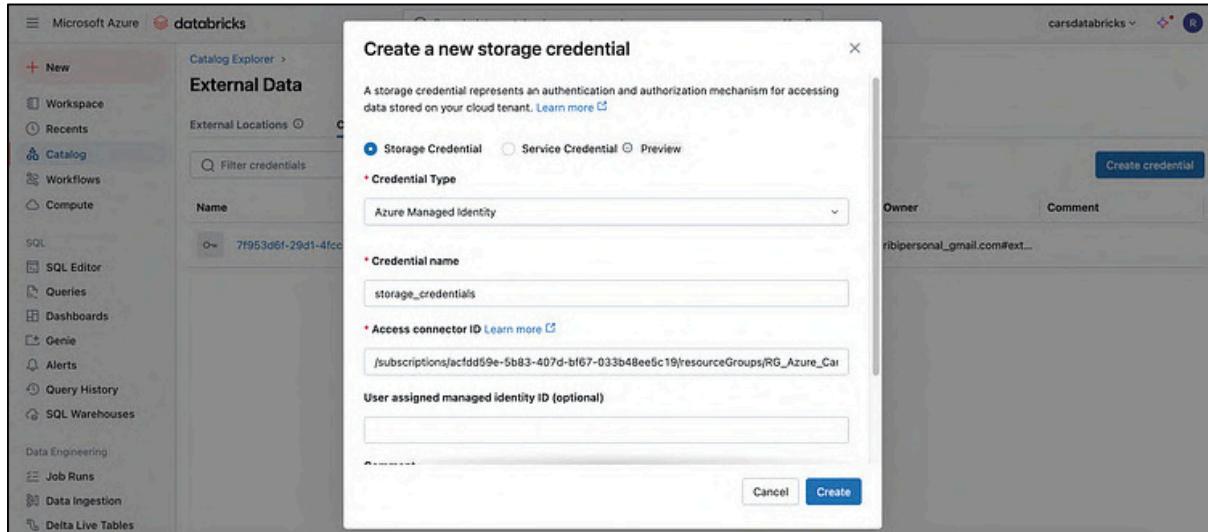
Step 2: Create Compute

The screenshot shows the 'Compute > New compute' interface. On the left sidebar, 'Compute' is selected. The main area is titled 'Rihab Feki's Cluster'. It includes sections for 'Policy' (set to 'Personal Compute'), 'Single user access' (set to 'Rihab Feki'), 'Performance' (Runtime: '15.4 LTS [Scala 2.12, Spark 3.5.0]', Node type: 'Standard_DS3_v2'), and 'Tags'. A summary box on the right shows '1 Driver', '14 GB Memory, 4 Cores', 'Runtime 15.4.x-scala2.12', 'Unity Catalog Standard_DS3_v2', and '0.75 DBU/h'. At the bottom are 'Create compute' and 'Cancel' buttons.

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”.

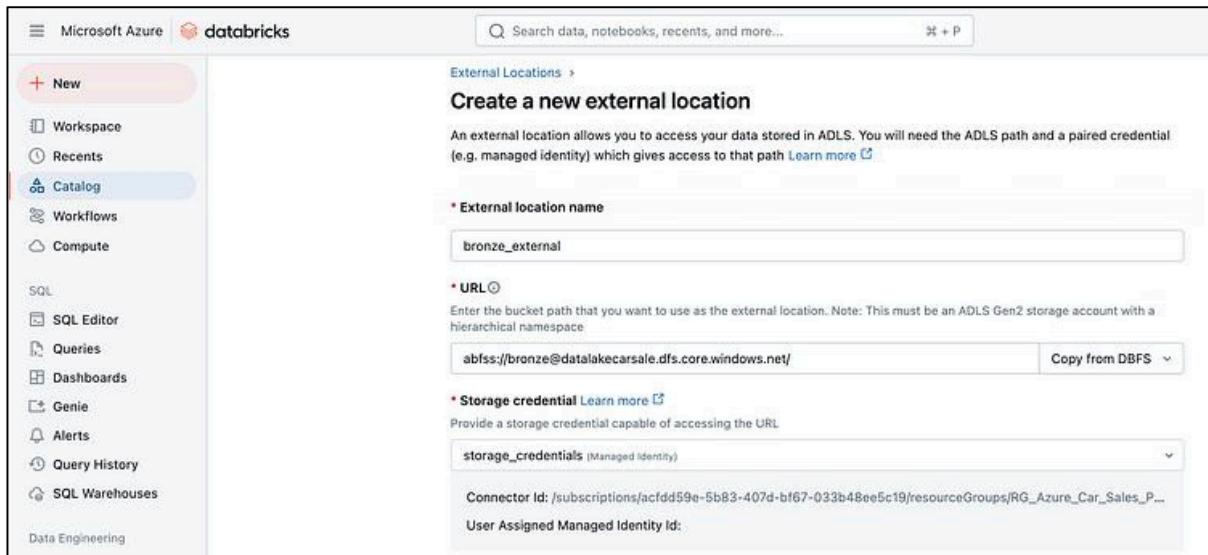


to create an External Location, you need to start by creating credentials. So, navigate to Databricks Workspace and click on Catalog > External Data > Credentials.



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)

The screenshot shows the Databricks Admin Console with the 'Catalog' tab selected. A modal window titled 'Edit Metastore Admin' is open. It contains instructions for Metastore admins and a search bar where 'rihab' is typed. Below the search bar, results for 'rihab' are listed, showing two entries: 'Rihab Feki' with the email 'ribipersonal@gmail.com/test#@ribipersonalgmail.onmicrosoft.com' and another entry for 'Rihab Feki' with the email 'ribipersonal@gmail.com'. A note at the bottom right of the modal says 'Rihab Feki (ribipersonal@gmail.com)'.

The screenshot shows the Databricks Catalog Explorer with the 'External Locations' section selected. A specific external location named 'bronze_external' is being configured. The 'Overview' tab is active. The 'Credential' field is set to 'storage_credentials'. The 'URL' field contains 'abfss://bronze@datalakecarsale.dfs.core.windows.net/'. The 'Limit to read-only use' field is set to 'Disabled'. On the right side, under 'About this external location', the 'Owner' is listed as 'Rihab Feki'.

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

The screenshot shows the Databricks Catalog Explorer with the 'External Data' section selected. The 'External Locations' tab is active. There are three external locations listed: 'bronze_external', 'gold_external', and 'silver_external'. Each location is associated with the 'storage_credentials' credential and a specific URL: 'abfss://bronze@datalakecarsale.dfs.core.windows.net/' for bronze, 'abfss://gold@datalakecarsale.dfs.core.windows.net/' for gold, and 'abfss://silver@datalakecarsale.dfs.core.windows.net/' for silver.

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.

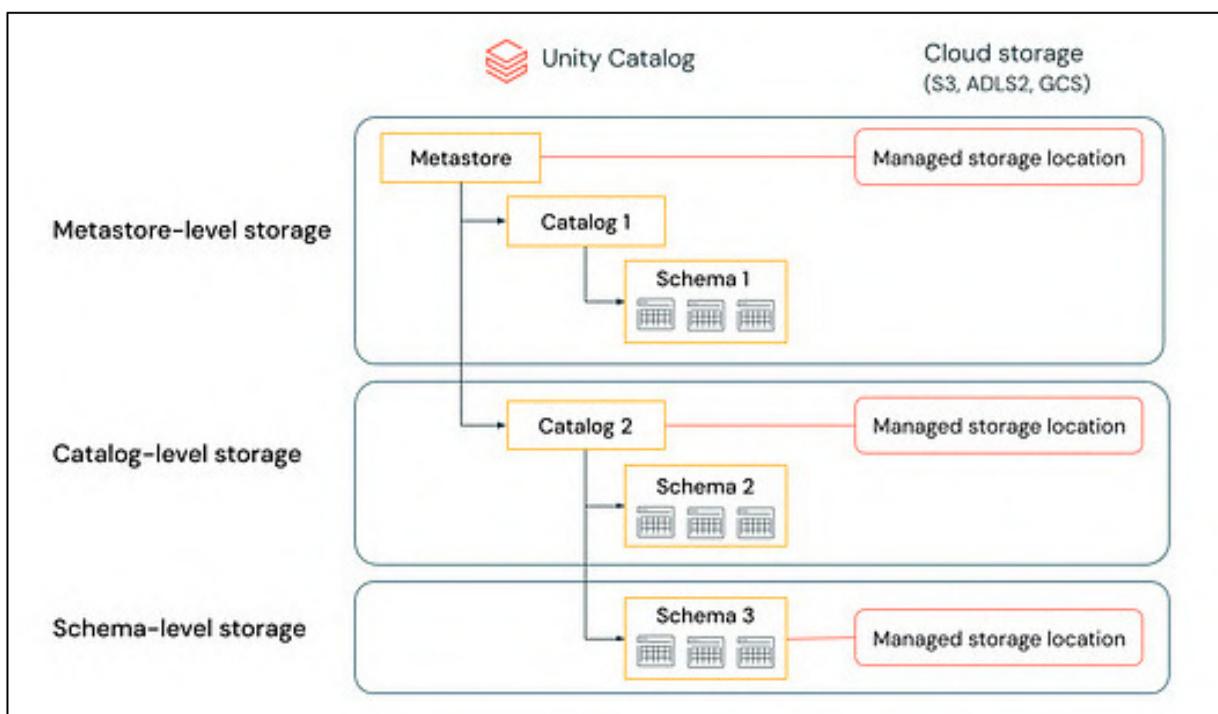
Step 4: Create a Workspace Click on Workspace in the sidebar, then in the Workspace folder, create a new project folder, and inside it create a Notebook to define the catalogs and schemas..

The screenshot shows the Microsoft Azure Databricks workspace interface. On the left, there's a sidebar with options like 'New', 'Workspace', 'Recents', 'Catalog', 'Workflows', 'Compute', 'SQL', and 'Data Flow'. The main area is titled 'CarsProject' and shows a tree view with 'Home', 'Workspace', 'Repos', 'Shared', 'Users', and 'CarsProject'. Inside 'CarsProject', there's a 'Repos (Legacy)' folder and some 'Favorites' and 'Trash' items. A table below lists a single item: 'Untitled Notebook 2025-01-07 08:58:30' of type 'Notebook' created by 'Rihab Feki' on '01/07/2025, 08:58:30 AM'.

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

The screenshot shows a Databricks notebook titled 'database_notebook' in Python mode. The left sidebar shows a 'Catalog' section with a search bar and a list of available catalogs: 'My organization' (containing 'system', 'cars_catalog', 'main', 'samples', 'Legacy', and 'hive_metastore'), 'Shared', and 'Legacy'. The main area has two sections: 'Create Catalog' and 'Create Schemas'. In 'Create Catalog', a cell runs the SQL command 'CREATE CATALOG cars_catalog;'. In 'Create Schemas', two cells run 'CREATE SCHEMA cars_catalog.silver;' and 'CREATE SCHEMA cars_catalog.gold;', both returning 'OK'. A note at the bottom of the schemas section says 'This result is stored as _sqldf and can be used in other Python cells.'

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.

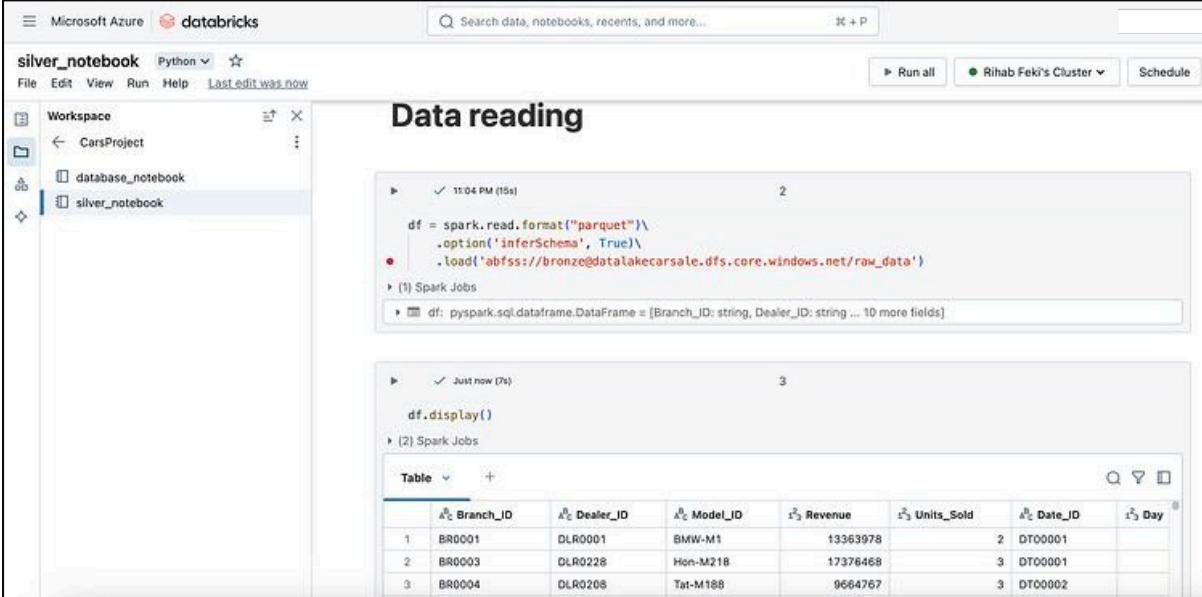
Unity Catalog allows the creation of managed and external tables.

- **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)

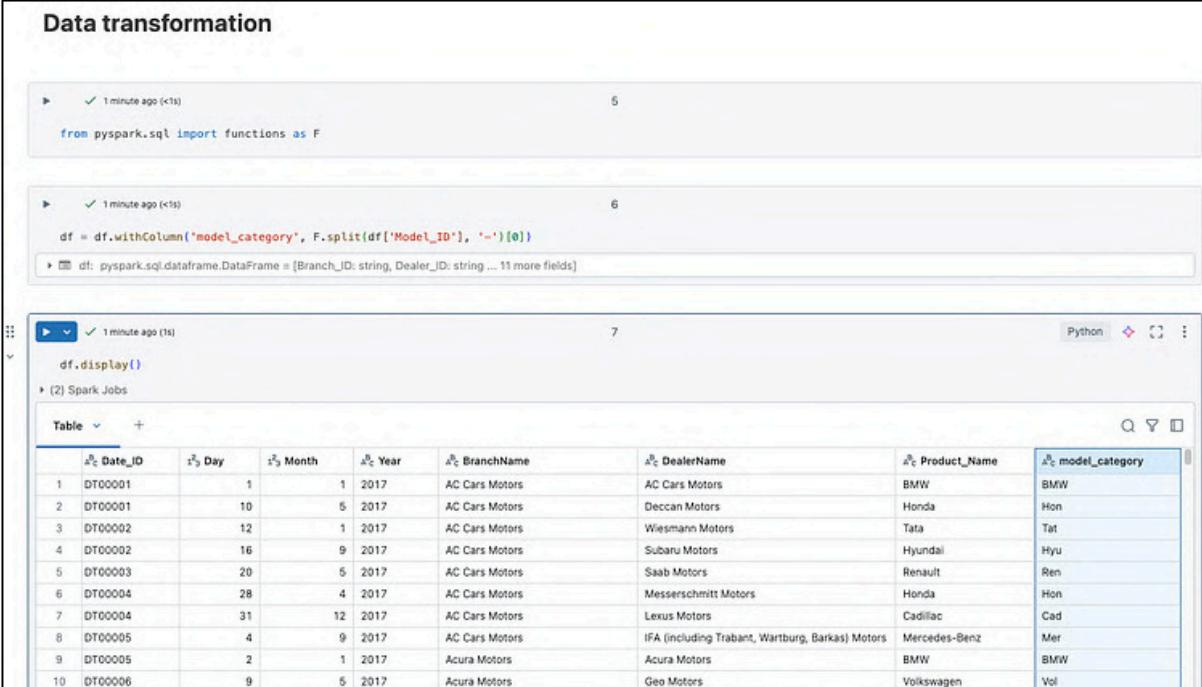
We will use PySpark API to read the data and one thing to note here is the ‘inferSchema’ option which helps to derive the schema from the raw data in parquet file format.



The screenshot shows a Databricks notebook titled "silver_notebook" in a workspace named "CarsProject". The notebook contains two code cells. The first cell, run at 11:04 PM, reads a parquet file from a bronze data lake into a DataFrame named df. The second cell, run just now, displays the DataFrame. The resulting table has columns: Branch_ID, Dealer_ID, Model_ID, Revenue, Units_Sold, Date_ID, and Day. The data shows three rows of sales records.

	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 a Databricks notebook titled "Data transformation". It includes three code cells. The first cell imports functions from pyspark.sql. The second cell adds a new column "model_category" by splitting the "Model_ID" column at the '-' character. The third cell displays the transformed DataFrame. The resulting table includes the original columns plus the new "model_category" column, which categorizes the models based on their names.

	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	Gro Motors	Volkswagen	Vol

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

```
▶ ✓ Just now (4s) 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 |
| --- | --- | --- | --- |
| AC Cars Motors | BMW | BMW | 6681989 |
| Deccan Motors | Honda | Hon | 5792156 |
| Wiesmann Motors | Tata | Tat | 3221689 |
| Subaru Motors | Hyundai | Hyu | 1841768 |
| Saab Motors | Renault | Ren | 4323696 |
| 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.

```
▶ ✓ 2 minutes ago (4s) 11
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

Table ▾ + 🔍 ⌂ ⌂

| Year | BranchName | Total_Units_Sold |
| --- | --- | --- |
| 2017 | Alpine Motors | 72 |
| 2017 | Aston Martin Motors | 69 |
| 2017 | Bristol Motors | 69 |
| 2017 | Acura Motors | 69 |
| 2017 | BMW Motors | 69 |
| 2017 | Ariel Motors | 63 |
| 2017 | Gilbern Motors | 63 |
```

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

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.

```
from pyspark.sql.functions import sum as F_sum\n\ndf.groupby('Year', 'BranchName').agg(\n    F_sum('Units_Sold').alias('Total_Units_Sold'))\n    .sort('Year', 'Total_Units_Sold', ascending=[True, False]).display()
```

(2) Spark Jobs

Table Visualization 1

New charts: ON

Visualization

Data Profile

Total_Units_Sold

Year

- 2018
- 2017
- 2019
- 2020

Year	Percentage
2018	29.99%
2017	29.34%
2019	28.76%
2020	11.91%

Then we write the transformed data to the silver storage container

silver_notebook

File Edit View Run Help Lastedit:now

Run all Rhab Feki's Cluster Schedule

Workspace

- CarsProject
- database_notebook
- silver_notebook

Data writing

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

(1) Spark Jobs

Then check that the parquet files were saved on Azure.

Home > Resource groups > RG_Azure_Car_Sales_Project > datalakecarsale | Containers >

silver Container

Upload Add Directory Refresh Remove Delete Change tier Acquire 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	Block type	Size	Lease state
_SUCCESS	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	0 B	Available
part-00000-bd-40509111653782272...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available
part-00001-bd-405091116537822...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available
part-00002-bd-405091116537822...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	60.81 KiB	Available
_committed_4050911165378227272...	1/8/2025, 12:02:06 PM	Hot (Inferred)		Block blob	318.8	Available
_started_4050911165378227273...	1/8/2025, 12:02:05 PM	Hot (Inferred)		Block blob	0 B	Available

To query the data, we can use SQL:

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

Querying Silver Data

The screenshot shows a Jupyter Notebook cell with the following content:

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

Below the code, it says '(3) Spark Jobs' and then shows a DataFrame named '_seldf'. The DataFrame has the following schema:

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

At the bottom, there is a table view of the data:

	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 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 bbl('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 interface. The title bar says "Microsoft Azure | databricks" and the notebook name is "gold_dim_model" in Python. The left sidebar shows a catalog tree under "Catalog" with categories like "cars_catalog", "gold", and "dim_model". The main area contains a code cell with the following SQL query:

```
%sql  
SELECT * FROM cars_catalog.gold.dim_model
```

Below the code cell is a table with the following data:

	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 Databricks workspace interface. On the left, there is a sidebar with a tree view of notebooks: "CarsProject" > "database_notebook", "gold_dim_model" (which is selected), and "silver_notebook". A modal dialog box titled "Clone 'gold_dim_model'" is open in the center. It has a "New name*" input field containing "gold_dim_branch", a "Clone to" dropdown set to "Workspace/CarsProject", and two buttons at the bottom: "Cancel" and "Clone".

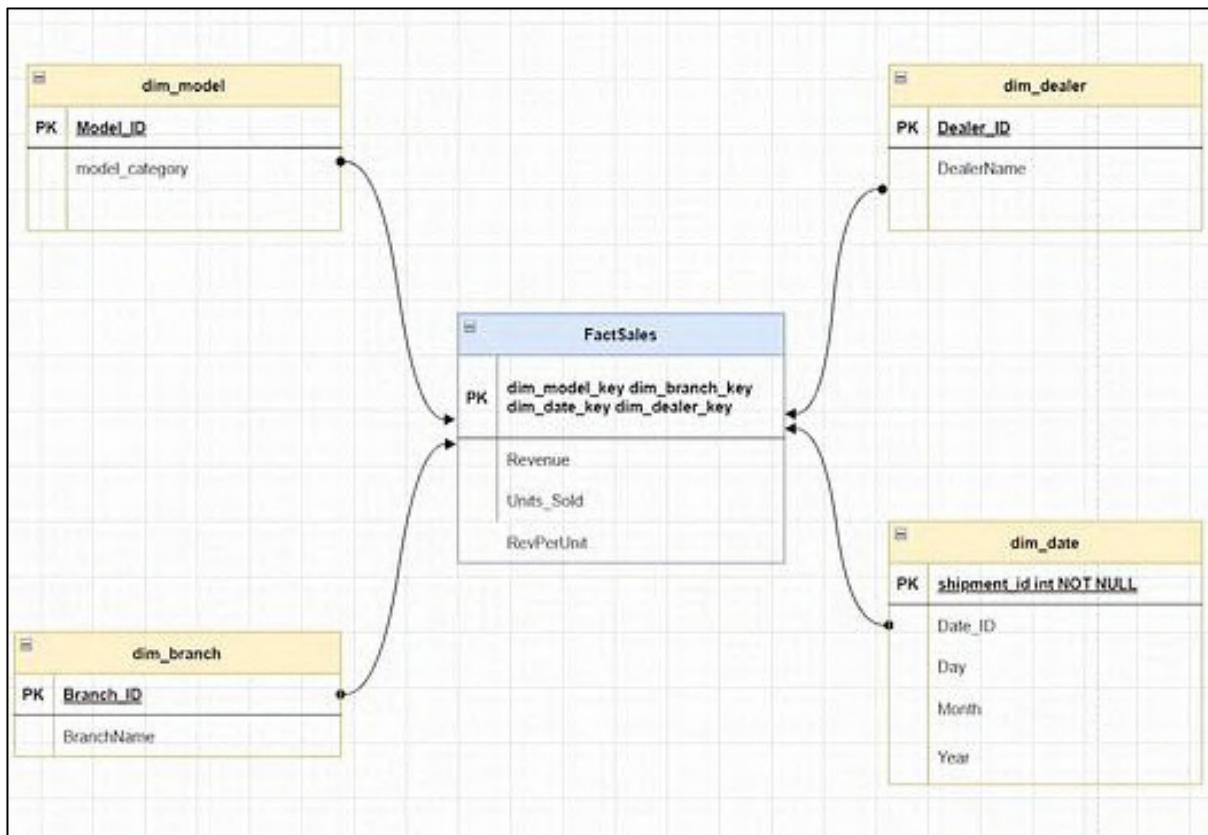
The process repeats for all the dimensions which are the dim_branch and dim_dealer.

The screenshot shows a 'Catalog' interface with a sidebar containing icons for 'Catalog', 'File', 'Sort', and 'Star'. The main area has a search bar with placeholder 'Type to search...' and filter buttons for 'For you' and 'All'. A tree view lists the following structures:

- cars_catalog
 - default
- gold
 - dim_branch
 - dim_date
 - dim_dealer
 - dim_model
 - information_schema
 - silver
- main

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')

```

Step8: 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.

The screenshot shows the Databricks Workflow creation interface. At the top, there is a header with 'Microsoft Azure' and 'databricks'. Below it, a search bar says 'Search data, notebooks, recents, and more...'. The main area is titled 'Data-Modek' with a star icon. It has two tabs: 'Runs' and 'Tasks'. Under 'Tasks', there is a box labeled 'Silver_Data' which contains a sub-task '...space/CarsProject/silver_notebook' and a cluster 'Rihab Feki's Cluster'. Below this, there are several configuration fields:

- 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)

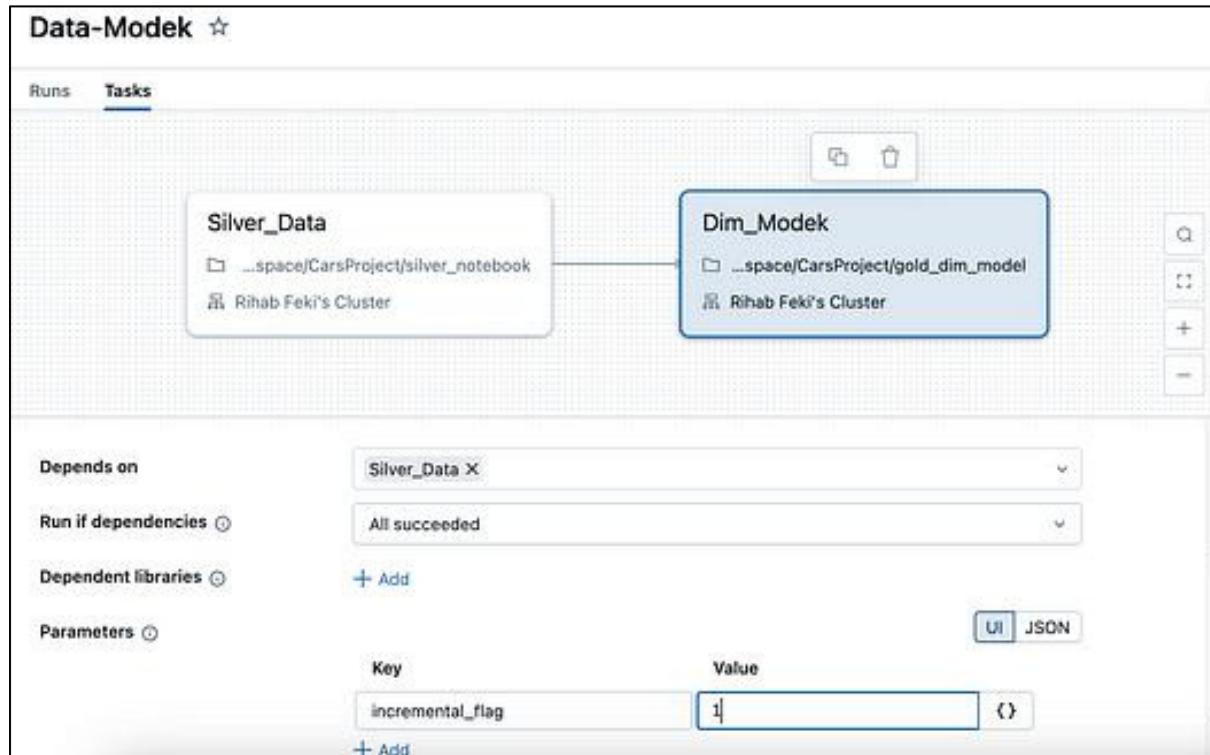
At the bottom right of the task configuration are 'Cancel' and 'Create task' buttons.

Add more tasks in this manner:

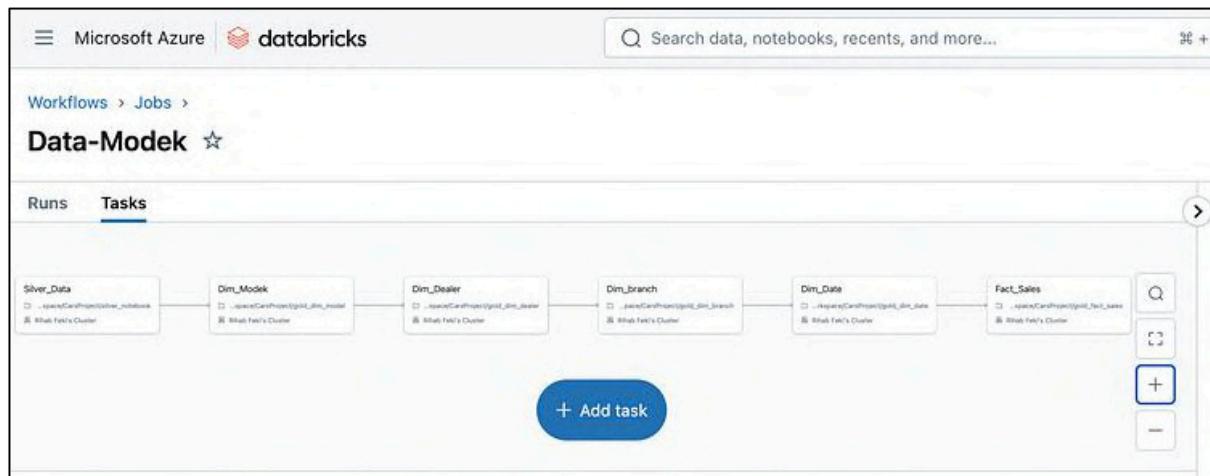
The screenshot shows the same Databricks Workflow creation interface. The 'Tasks' tab is selected. A blue button labeled '+ Add task' is highlighted. A dropdown menu is open, listing the following task types:

- 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:



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 Databricks Data Modeler interface. The top navigation bar says 'Workflows > Jobs > Data-Modek'. The main area has tabs for 'Runs' and 'Tasks'. Under 'Tasks', there's a preview of a task graph with nodes like 'Dim_Date', 'Dim_Dealer', 'Dim_Modek', 'Dim_branch', and 'Fact_Sales'. A blue button '+ Add task' is visible. Below the graph, there are input fields: 'Source*' (Silver_Data), 'Path*' (Dim_Dealer, Dim_Modek, Dim_branch, Fact_Sales), 'Cluster*' (Rihab Feki's Cluster), 'Depends on' (Silver_Data), and 'Run if dependencies' (All succeeded). At the bottom right 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.

The screenshot shows the Databricks Data Model run interface. The top navigation bar says 'Workflows > Jobs > Data-Model > Data-Model run'. The main area has tabs for 'Graph' and 'Timeline'. The 'Graph' tab shows a pipeline with five tasks: 'Silver_Data', 'Dim_Date', 'Dim_Dealer', 'Dim_Modek', and 'Fact_Sales'. Each task is shown as a green box with a success status (e.g., 'Succeeded - 14s'). Arrows indicate the flow from Silver_Data to Dim_Date, Dim_Dealer, Dim_Modek, and Dim_branch, and from Dim_branch to Fact_Sales. To the right, the 'Job run details' panel provides specific information about the run: Job ID (266926568184946), Job run ID (34504658038297), Launched (Manually), Started (Jan 16, 2025, 12:22 PM), Ended (Jan 16, 2025, 12:23 PM), Duration (1m 20s), Queue duration (-), Status (Succeeded), and Lineage (5 upstream tables, 2 downstream tables). There's also a 'Compute' section indicating a single node: Standard_DS3_v2 - DBR: 15.4 LTS (includes Apache).

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 Microsoft Azure Databricks interface. On the left, the sidebar includes options like Workspace, Recents, Catalog, Workflows, Compute, SQL, and SQL Editor (which is selected). The main area has a 'Catalog' sidebar with sections for 'For you' and 'All'. Under 'All', it lists 'My organization' (system, cars_catalog, default, gold), 'cars_branch', 'dim_branch', 'dim_date', 'dim_dealer', 'dim_model', 'factsales', 'information_schema', 'silver', 'main', 'Shared', 'samples', and 'Legacy'. A 'hive_metastore' option is also present. The central workspace contains a 'New Query 2025-01-16 12:33pm' tab with the SQL command: 'SELECT * FROM cars_catalog.gold.factsales'. Below the query is a 'Raw results' table with 9 rows of data:

	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 Microsoft Azure Databricks Compute page. The sidebar includes options like Workspace, Recents, Catalog, Workflows, Compute (selected), SQL, and SQL Editor. The main area displays a table titled 'Compute' under the 'All-purpose compute' tab. It shows a single cluster named 'Rihab Feki's Cluster' with a 'Personal Co...' policy, 15.4 runtime, and active status. The table columns include State, Name, Policy, Runtime, Active m..., Active co..., Active DB..., Source, Creator, Notebooks, and a 'Start' button.

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