Taksh Shah

Project: Financial Data Analysis

Objective:

The project aims to design and implement a robust data pipeline for processing customer account data. This includes copying data from ADLS GEN2 (Bronze layer) and transforming the data in the Silver layer using Data bricks Notebooks and storing the data into Gold layer using SCDType 1 Delta Table in ADLS GEN2. The pipeline aims to ensure efficient, accurate, and scalable data processing to support downstream analytics and reporting needs.

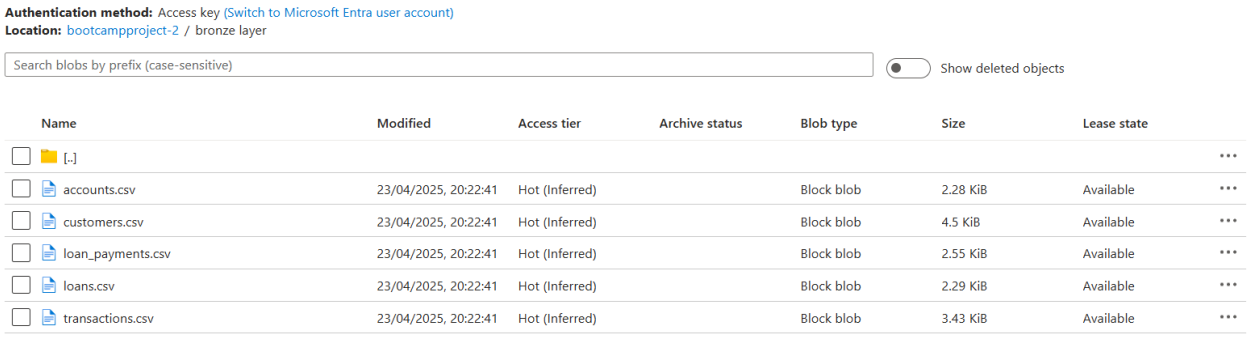
Dataset used: https://www.kaggle.com/datasets/varunkumari/ai-bank-dataset

Tools required:

- Azure Data Lake Gen 2 Storage
- Azure Databricks
- App registration – service principal
- Azure Keyvault
- Pyspark
- Power BI Desktop
- Microsoft Fabric
- Draw.io – for architecture diagram

Bronze layer:

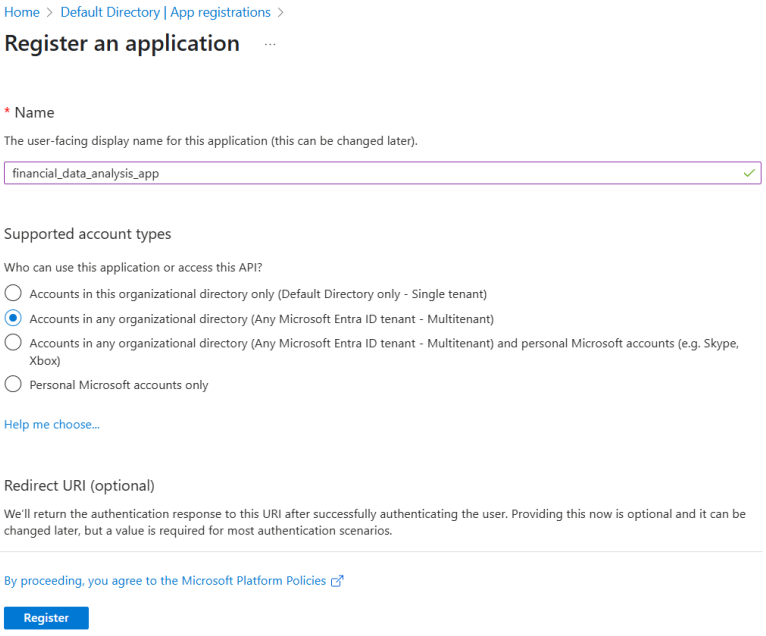
First, create 3 folders in your adlsgen2 container – bronze layer, silver layer and gold layer – and upload the dataset files into bronze folder.



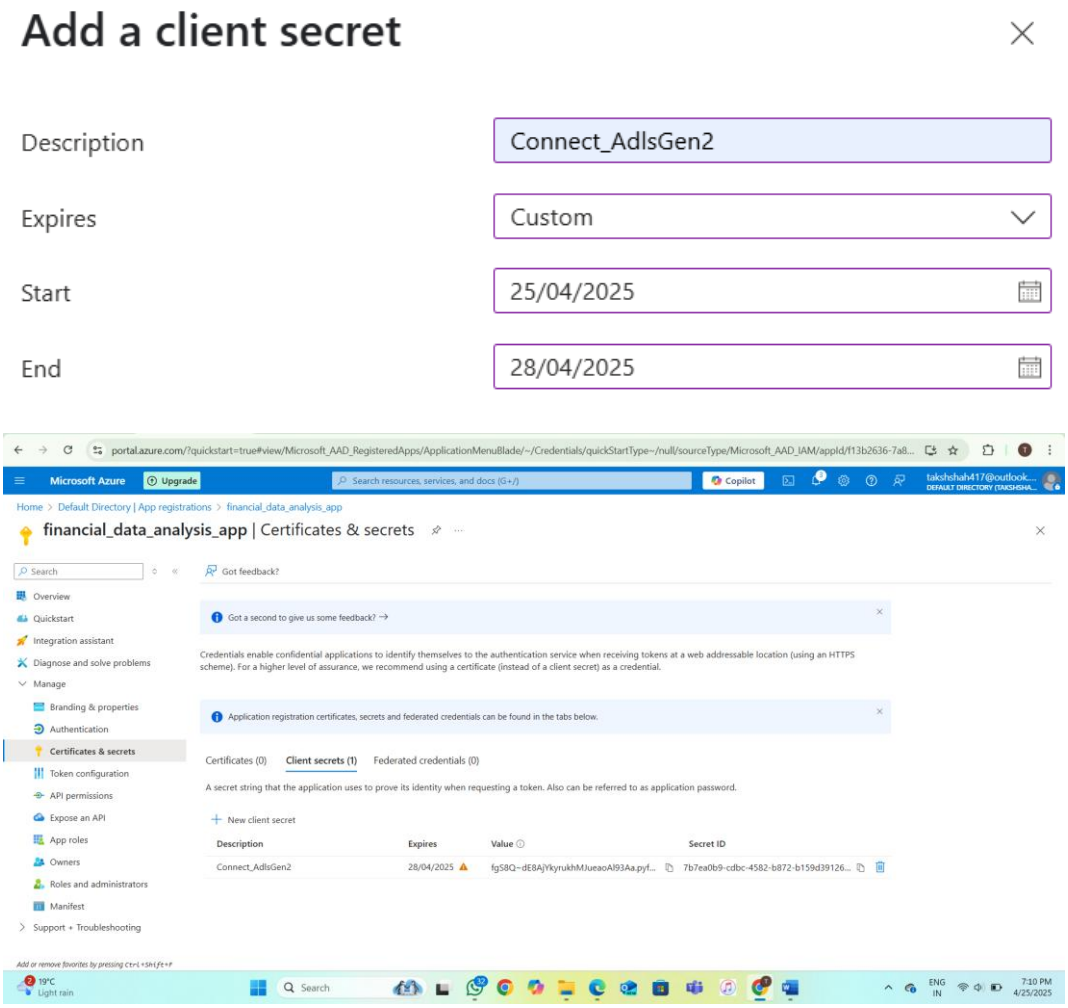
Now, in order to give databricks access to the adlsgen2 storage account, we'll have to create a mount for adlsgen2, so let's use service principal to accomplish this task.

Go to Microsoft Entra ID in azure portal -> manage -> app registrations -> select "new registration"

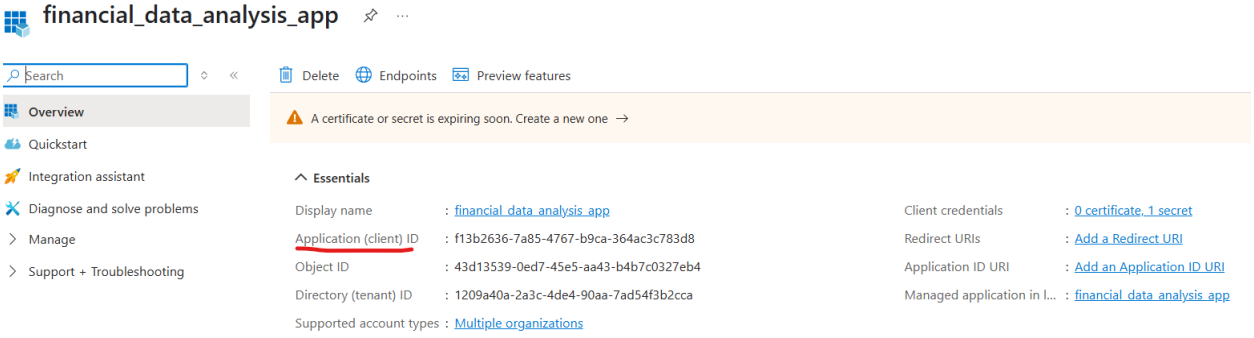
Give a name for the application. For supported account types, select "accounts in any organizational directory (Any Microsoft Entra ID tenant - Multitenant)" and register the application.



Copy the tenant ID, click on “client credentials” -> click on “new client secret” and add details as per the image given below and click “add”:

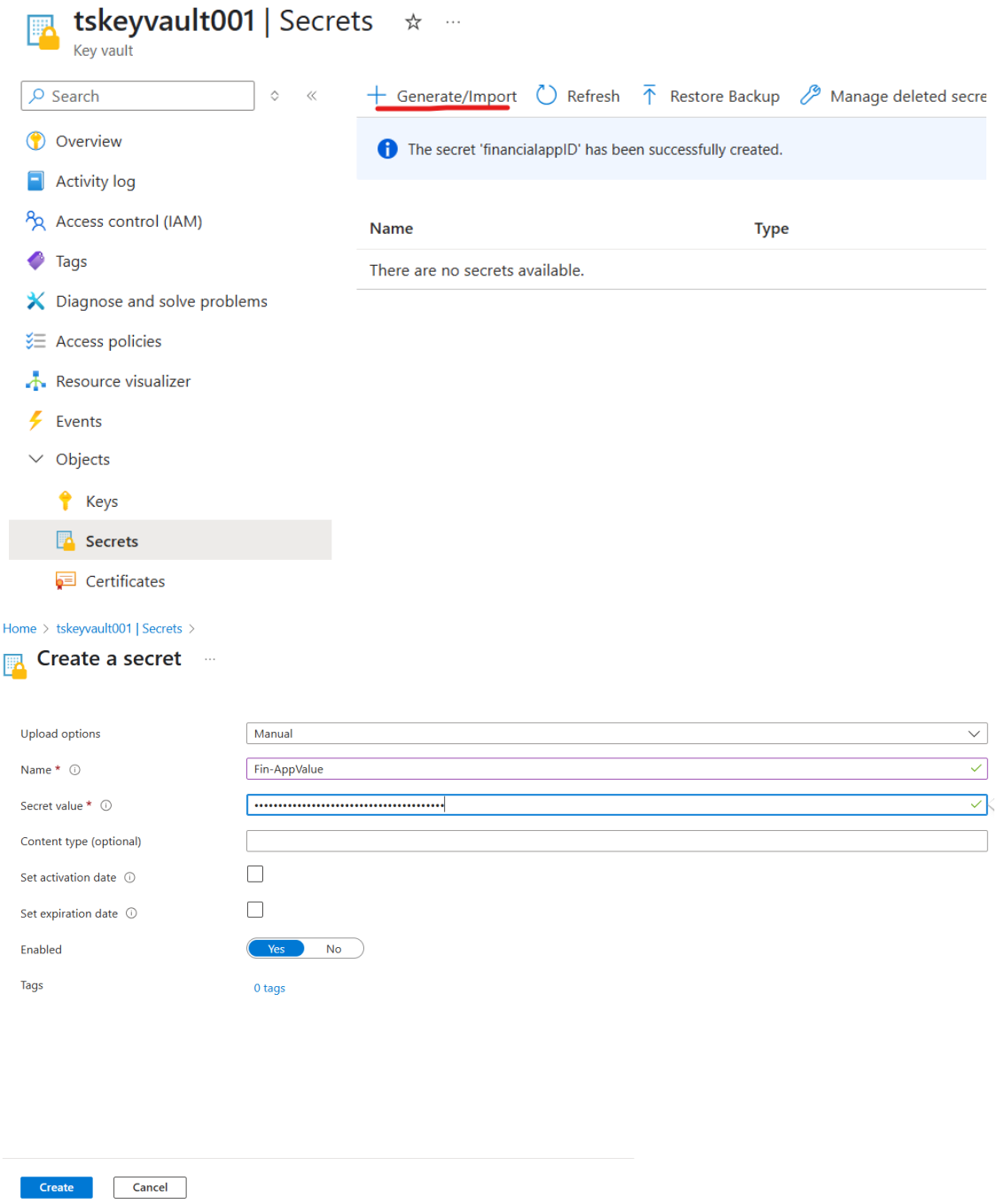


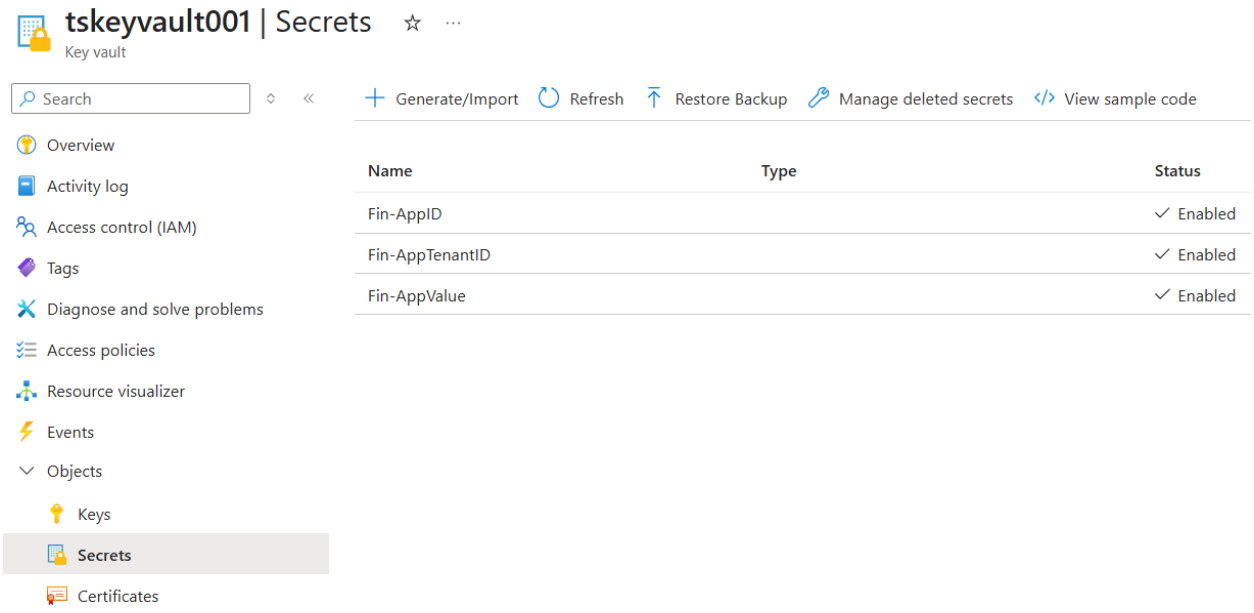
In the above image, make sure you copy the “value” and “Secret ID” and keep it stored for future use. Now, go to the overview section of the application and copy the application ID as shown in the image below:



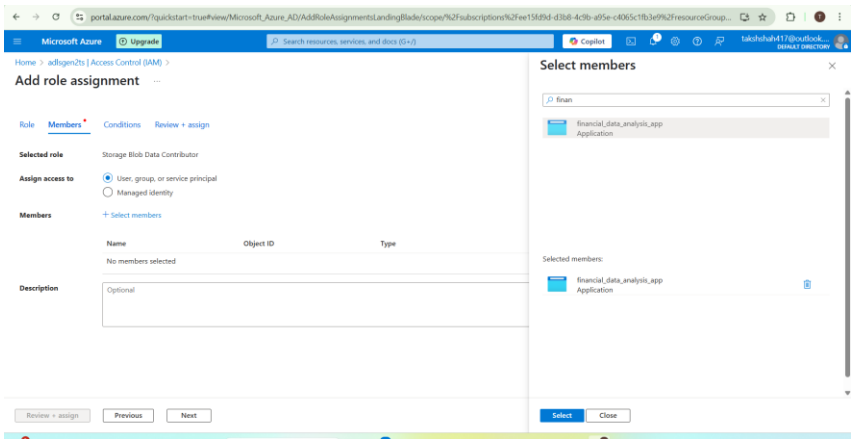
Now, we'll go to the keyvault and create a new secret for your databricks application ID, application value and application secret ID. So, let's say we're creating a secret for appID first, then we'll give the name of the secret key as "Fin-AppID" for convenience and "secret value" will be the application ID which we had copied earlier. Similarly, create a secret for "Fin-AppValue", where the "secret value" field will be the "value" from client secrets for our app which we had copied earlier. Finally, create a secret for tenant ID as well.

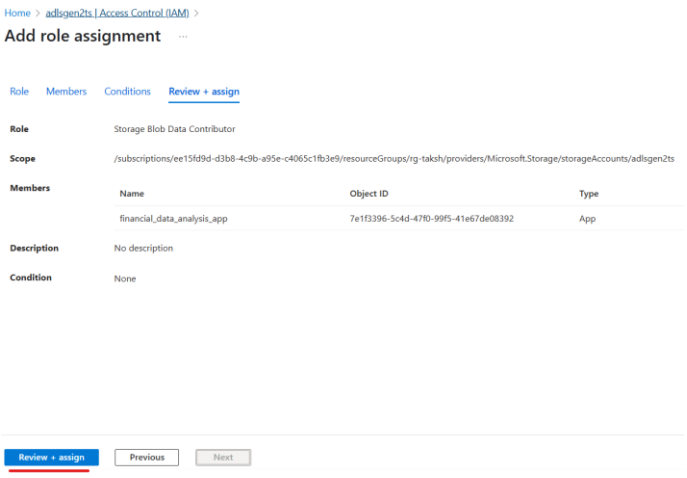
Go to your keyvault -> secrets -> click on generate/import





Make sure you give the “storage blob data contributor” role to your application (financial_data_analysis_app aka your service principal) under access management control section in your adlsgen2 storage account.



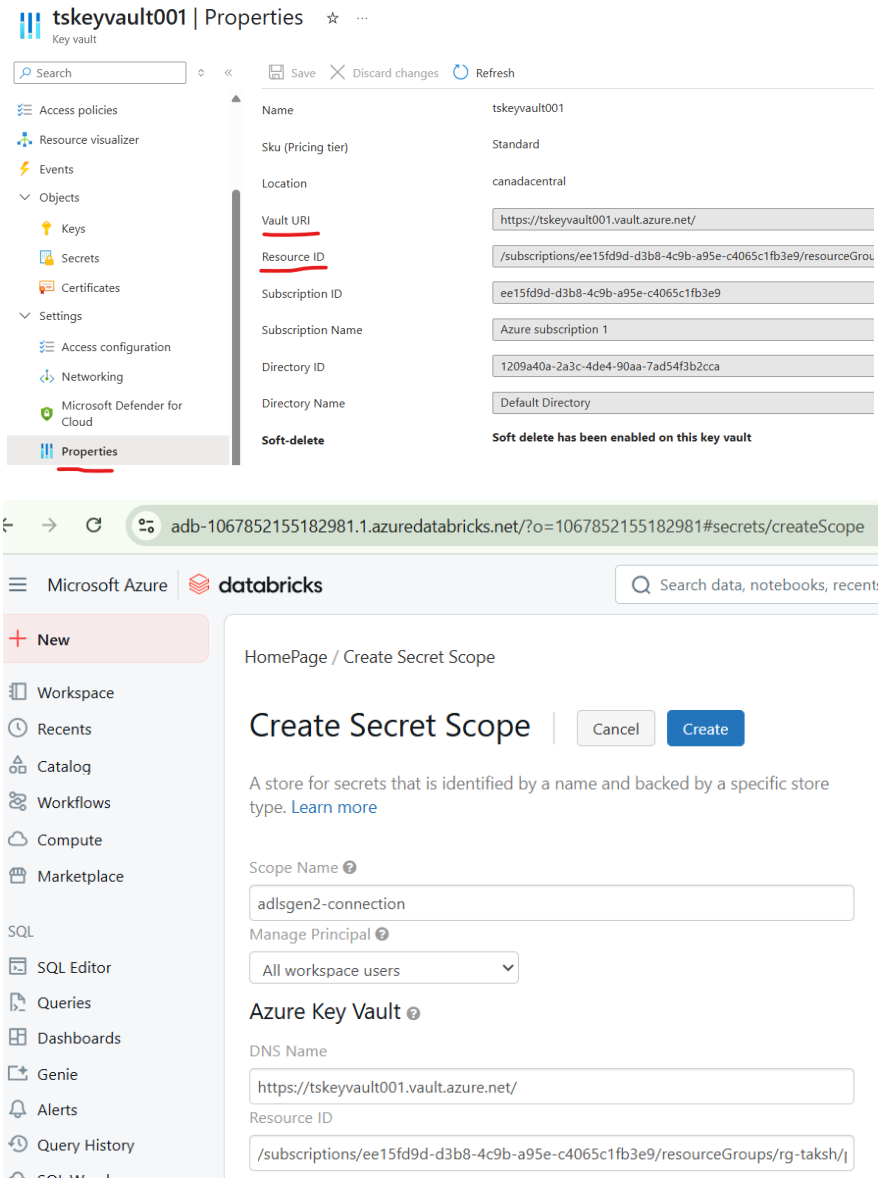


Now, let's create a scope in our databricks workspace for adlsgen2 storage account. For that, modify the databricks web browser URL by suffixing "/#secrets/createScope" after databricks.net part in the URL.

Original URL: https://adb-1067852155182981.1.azuredatabricks.net/editor/notebooks/2396847816085068?o=1067852155182981#command/6249167138378354

Modified URL: https://adb-1067852155182981.1.azuredatabricks.net/#secrets/createScope

After landing on the page, fill out all the details as per the image provided below and create the scope. DNS Name (aka vault URL) and Resource ID can be found in the properties section of your keyvault.



Now, visit the following link: https://phv2705.medium.com/mount-adls-gen2-to-databricks-file-system-using-service-principal-oauth-2-0-47527e339178

From here, copy the following code from “step-5” in the article to mount adlsgen2 to databricks using service principal:

```
configs = {"fs.azure.account.auth.type": "OAuth",
"fs.azure.account.oauth.provider.type":
"org.apache.hadoop.fs.azurebfs.oauth2.ClientCredsTokenProvider",
"fs.azure.account.oauth2.client.id": "<application-id>",
"fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope="<scope-name>",key="<service-credential-key-name>"),
"fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/<directory-id>/oauth2/token"}

dbutils.fs.mount(
source = "abfss://<file-system-name>@<storage-account-name>.dfs.core.windows.net/",
mount_point = "/mnt/<mount-name>",
extra_configs = configs)
```

Paste this code in a new notebook. Now, in the place of <application ID>, we will enter the actual application ID but we have stored our appID in our keyvault, so instead of directly providing the appID, we’ll fetch it from the keyvault. To do that, let’s first replace <application-id> with dbutils.secrets.get(scope="*<scope-name>*",key="*<service-credential-key-name>*") [the same line of code that’s below <application-id> in the code itself]. Similarly, we’ll provide appValue as well. Now, we’ll start editing the code as per our requirement. Following are the changes to be made:

<scope-name>: adlsgen2-connection

appID: Fin-AppID (name as per secret)

appValue: Fin-AppValue (name as per secret)

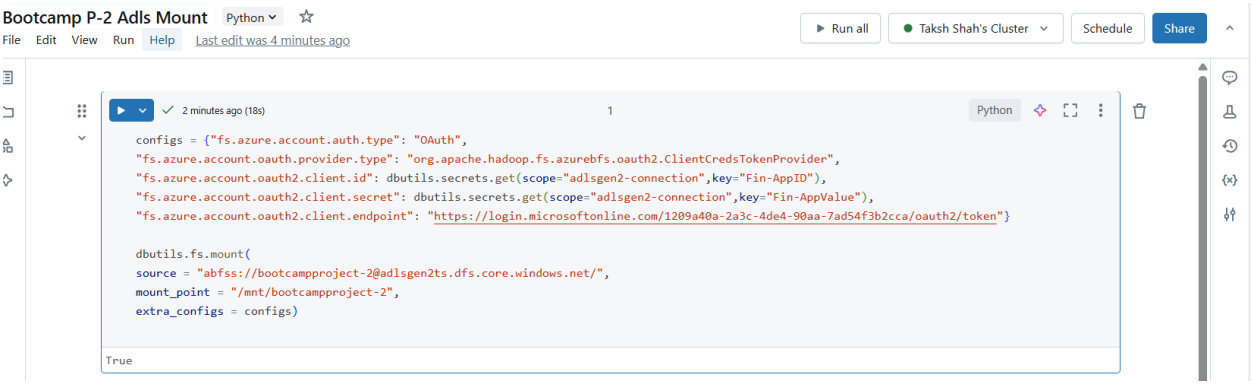
<directory-id>: tenant ID which we had copied previously

File-system name: this means adlsgen2 container-name = bootcampproject-2

Storage-account name: adlsgen2ts

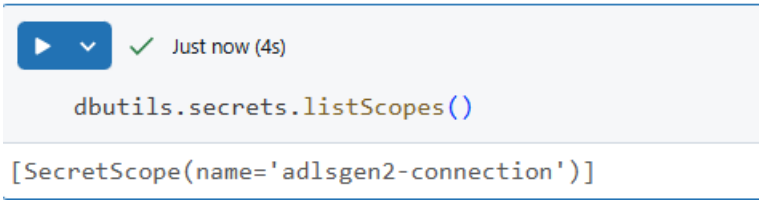
Mount-name: same as container-name = bootcampproject-2

The final code after all the changes looks like this:

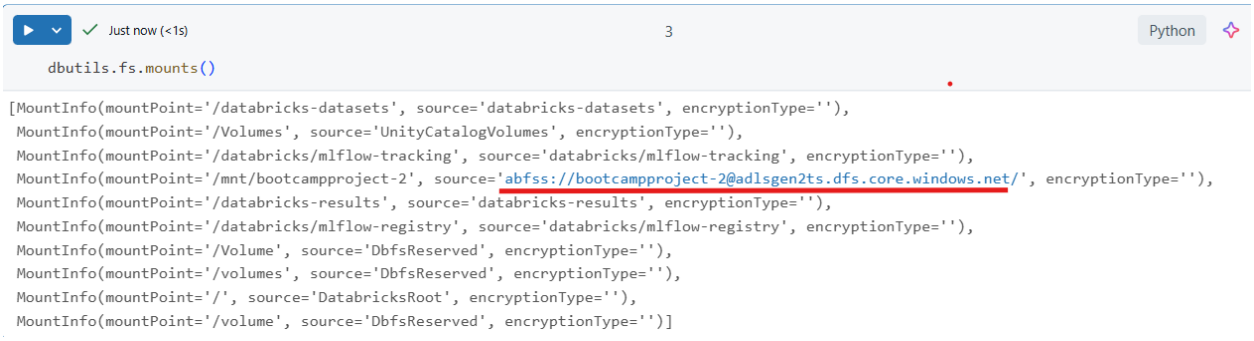


In order to check if your scope was created or not, you can run the following command in your notebook:

dbutils.secrets.listScopes()



Great, so now if we run the command “dbutils.fs.mounts()” in our notebook, it’ll list all the mounts that we have and we can see that a mount for adlsgen2 exists now.



Silver Layer:

We'll use pyspark code to read data from bronze layer, remove duplicates and null values and store the cleaned data into silver layer in parquet format as per the project's requirement.

First, let's create a function called "clean_and_save_data" using the code given below:

```
from pyspark.sql.functions import col

def clean_and_save_data (

    input_path: str,
    output_path: str,
    file_format: str = "csv",
    output_format: str = "parquet",
    header: bool = True

):

    """

    Reads a CSV file from ADLS Gen2, removes duplicates and nulls, then saves it in Parquet format.

    Parameters:

    - input_path: ADLS Gen2 path to the input CSV file (e.g., abfss://container@storageaccount.dfs.core.windows.net/folder/input.csv)
    - output_path: ADLS Gen2 path where cleaned file will be saved
    - file_format: Format of input file (default is 'csv')
    - output_format: Format to save cleaned data (default is 'parquet')
    - header: Whether the CSV has a header row (default True)

    """

    # Read CSV from ADLS Gen2

    df = spark.read.format(file_format) \

        .option("header", str(header).lower()) \

        .load(input_path)
```

```
# Remove nulls and duplicates
df_cleaned = df.dropna().dropDuplicates()

# Save as Parquet to new folder
df_cleaned.write.mode("overwrite").format(output_format).save(output_path)

print(f"Cleaned data saved to {output_path}")
```

```
from pyspark.sql.functions import col

def clean_and_save_data(
    input_path: str,
    output_path: str,
    file_format: str = "csv",
    output_format: str = "parquet",
    header: bool = True
):
    """
    Reads a CSV file from ADLS Gen2, removes duplicates and nulls, then saves it in Parquet format.
    Parameters:
    - input_path: ADLS Gen2 path to the input CSV file (e.g., abfss://container@storageaccount.dfs.core.windows.net/folder/input.csv)
    - output_path: ADLS Gen2 path where cleaned file will be saved
    - file_format: Format of input file (default is 'csv')
    - output_format: Format to save cleaned data (default is 'parquet')
    - header: Whether the CSV has a header row (default True)
    """
```

```
# Read CSV from ADLS Gen2
df = spark.read.format(file_format) \
    .option("header", str(header).lower()) \
    .load(input_path)

# Remove nulls and duplicates
df_cleaned = df.dropna().dropDuplicates()

# Save as Parquet to new folder
df_cleaned.write.mode("overwrite").format(output_format).save(output_path)

print(f"Cleaned data saved to {output_path}")
```

This function reads a csv file from adlsgen2 storage account, removes duplicates and nulls, then saves it in parquet format. Using this function, we'll fetch all the data from the bronze layer folder, clean it and store it in silver layer folder.

```
✓ 1 hour ago (20s)                                                    2

input_path = "/mnt/bootcampproject-2/bronze layer/day1/accounts.csv"
output_path = "/mnt/bootcampproject-2/silver layer/accounts_cleaned"

clean_and_save_data(input_path, output_path)

▸ (3) Spark Jobs
```

```
Cleaned data saved to /mnt/bootcampproject-2/silver layer/accounts_cleaned
```

```
✓ 1 hour ago (33s)                                                    3

#customers.csv
input_path = "/mnt/bootcampproject-2/bronze layer/day1/customers.csv"
output_path = "/mnt/bootcampproject-2/silver layer/customers_cleaned"
clean_and_save_data(input_path, output_path)

#loan_payments.csv
input_path = "/mnt/bootcampproject-2/bronze layer/day1/loan_payments.csv"
output_path = "/mnt/bootcampproject-2/silver layer/loan_payments_cleaned"
clean_and_save_data(input_path, output_path)

#loans.csv
input_path = "/mnt/bootcampproject-2/bronze layer/day1/loans.csv"
output_path = "/mnt/bootcampproject-2/silver layer/loans_cleaned"
clean_and_save_data(input_path, output_path)

#transactions.csv
input_path = "/mnt/bootcampproject-2/bronze layer/day1/transactions.csv"
output_path = "/mnt/bootcampproject-2/silver layer/transactions_cleaned"
clean_and_save_data(input_path, output_path)

▸ (12) Spark Jobs
```

```
Cleaned data saved to /mnt/bootcampproject-2/silver layer/customers_cleaned
Cleaned data saved to /mnt/bootcampproject-2/silver layer/loan_payments_cleaned
Cleaned data saved to /mnt/bootcampproject-2/silver layer/loans_cleaned
Cleaned data saved to /mnt/bootcampproject-2/silver layer/transactions_cleaned
```

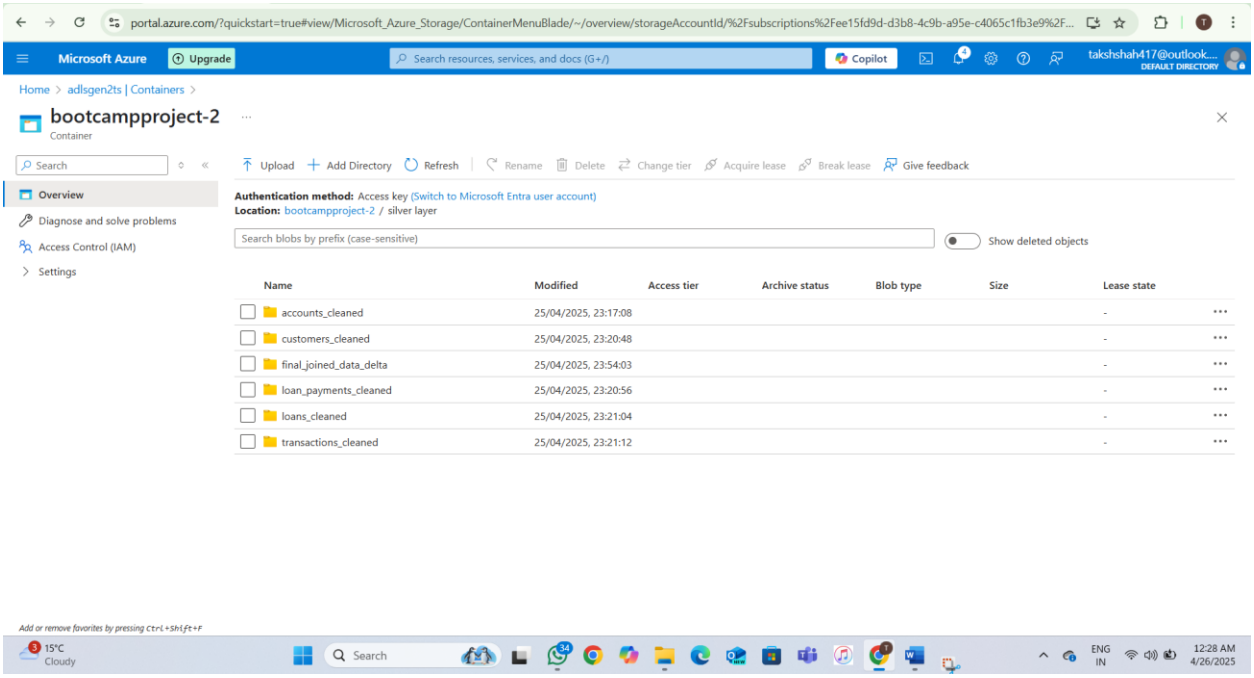
In addition to cleaning the data, our project requires us to join all the cleaned data files into a single file, select certain columns from each file and store in delta format in the silver layer folder. Use the code provided in the images below:

32 minutes ago (37s) 4

```
from pyspark.sql.functions import col

# Read all cleaned silver data
accounts_df = spark.read.parquet("/mnt/bootcampproject-2/silver layer/accounts_cleaned")
customers_df = spark.read.parquet("/mnt/bootcampproject-2/silver layer/customers_cleaned")
loan_payments_df = spark.read.parquet("/mnt/bootcampproject-2/silver layer/loan_payments_cleaned")
loans_df = spark.read.parquet("/mnt/bootcampproject-2/silver layer/loans_cleaned")
transactions_df = spark.read.parquet("/mnt/bootcampproject-2/silver layer/transactions_cleaned")

# Step 1: Join accounts with customers on customer_id
acc_cust_df = accounts_df.join(customers_df, on="customer_id", how="left")

# Step 2: Join with transactions on account_id
acc_cust_trans_df = acc_cust_df.join(transactions_df, on="account_id", how="left")

# Step 3: Join with loans on account_id
acc_cust_trans_loans_df = acc_cust_trans_df.join(loans_df, on="customer_id", how="left")

# Step 4: Join with loan_payments on loan_id
final_df = acc_cust_trans_loans_df.join(loan_payments_df, on="loan_id", how="left")
```

```
# Final: Drop duplicates and select required columns
final_df = final_df.dropDuplicates().select(
    col("account_id").cast("int"),
    col("customer_id").cast("int"),
    col("transaction_id").cast("int"),
    col("loan_id").cast("int"),
    col("payment_id").cast("int"),
    col("balance").cast("float"),
    col("transaction_date").cast("timestamp"),
    col("transaction_amount").cast("float"),
    col("loan_amount").cast("float"),
    col("payment_date").cast("timestamp"),
    col("payment_amount").cast("float")
)

# Write the result as a Delta table to the silver layer
final_df.write.format("delta") \
    .mode("overwrite") \
    .save("/mnt/bootcampproject-2/silver layer/final_joined_data_delta")
```

```
(17) Spark Jobs
acc_cust_df:  pyspark.sql.dataframe.DataFrame = [customer_id: string, account_id: string ... 8 more fields]
acc_cust_trans_df:  pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 12 more fields]
acc_cust_trans_loans_df:  pyspark.sql.dataframe.DataFrame = [customer_id: string, account_id: string ... 16 more fields]
accounts_df:  pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 2 more fields]
customers_df:  pyspark.sql.dataframe.DataFrame = [customer_id: string, first_name: string ... 5 more fields]
final_df:  pyspark.sql.dataframe.DataFrame = [account_id: integer, customer_id: integer ... 9 more fields]
loan_payments_df:  pyspark.sql.dataframe.DataFrame = [payment_id: string, loan_id: string ... 2 more fields]
loans_df:  pyspark.sql.dataframe.DataFrame = [loan_id: string, customer_id: string ... 3 more fields]
transactions_df:  pyspark.sql.dataframe.DataFrame = [transaction_id: string, account_id: string ... 3 more fields]
```



Silver layer folder containing cleaned data and the joined data file

Gold Layer:

Here, we have to bring cleaned data from silver folder to gold layer using scdtype-1 logic.

We'll discuss accounts_cleaned file. Use the following code for all the 5 files by making necessary changes:

We'll create our target table first.

accounts table scdtype-1

11:29 AM (11s)

```
%sql
--gold layer begins now
-- we'll do accounts_cleaned file first
CREATE TABLE IF NOT EXISTS hive_metastore.default.accounts (
    account_id INT,
    customer_id INT,
    account_type STRING,
    balance DOUBLE,
    created_by string,
    created_date timestamp,
    updated_by string,
    updated_date timestamp,
    hashkey bigint
)
USING DELTA
LOCATION '/mnt/bootcampproject-2/gold layer/accounts'
```

OK

Then we provide source path (from where we'll take the cleaned file) and target path (where we'll store our file).

11:34 AM (<1s) 7

```
src_path_accounts="/mnt/bootcampproject-2/silver layer/accounts_cleaned"
print(src_path_accounts)
tgt_path_accounts="/mnt/bootcampproject-2/gold layer/accounts"
print(tgt_path_accounts)
```

```
/mnt/bootcampproject-2/silver layer/accounts_cleaned
/mnt/bootcampproject-2/gold layer/accounts
```

The below code is just to check if data is present.

11:35 AM (3s) 8

```
#reading the file just to check if data is present
df_src=spark.read.format("parquet").option("header", "true").option("inferSchema", "true").load(src_path_accounts)
display(df_src)
```

(2) Spark Jobs

df_src: pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 2 more fields]

Table +

	account_id	customer_id	account_type	balance
1	1	45	Savings	1000.50
2	48	6	Checking	4900.00
3	21	53	Savings	300.25
4	29	58	Savings	75.25

Next, we add a hashkey.

11:34 AM (<1s) 9

```
#add hashkey
from pyspark.sql.functions import *
df_hash_accounts=df_src.withColumn("hash_key",crc32(concat(*df_src.columns)))
display(df_hash_accounts)
```

(1) Spark Jobs

df_hash_accounts: pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 3 more fields]

Table +

	account_id	customer_id	account_type	balance	hash_key
1	1	45	Savings	1000.50	4261402674
2	48	6	Checking	4900.00	3216528439
3	21	53	Savings	300.25	2800704470
4	29	58	Savings	75.25	3637748452
5	34	41	Checking	3500.50	868746038

The following code is to convert the delta table object to data frame so we can display it.

```
11:39 AM (3s)                                                    10

#converting delta table object to dataframe so we can display it
from delta.tables import DeltaTable
dbtable_accounts = DeltaTable.forPath(spark, tgt_path_accounts)
dbtable_accounts.toDF().show()
```

```
+----------+-----------+------------+-------+----------+------------+----------+------------+-------+
|account_id|customer_id|account_type|balance|created_by|created_date|updated_by|updated_date|hashkey|
+----------+-----------+------------+-------+----------+------------+----------+------------+-------+
+----------+-----------+------------+-------+----------+------------+----------+------------+-------+
```

Now, we'll join the source and target tables (similar to lookup activity in scdtype-1 dataflow in azure synapse)

```
11:42 AM (2s)                                                    11                                    Python

df_hash_accounts=df_hash_accounts.alias("src").join(dbtable_accounts.toDF().alias("tgt"), ((col("src.account_id") == col("tgt.account_id")) &
(col("src.hash_key") == col("tgt.hashkey"))), "anti").select(col("src.*"))
df_hash_accounts.show()

(1) Spark Jobs

df_hash_accounts:  pyspark.sql.dataframe.DataFrame = [account_id: string, customer_id: string ... 3 more fields]
|        21|         53|     Savings| 300.25|2800704470|
|        29|         58|     Savings|  75.25|3637748452|
|        34|         41|    Checking|3500.50| 868746038|
|        35|         62|     Savings| 175.75|3427479153|
|         5|         56|     Savings| 500.00|2866974084|
|        78|          4|    Checking|7900.50|2650882798|
|        38|         15|    Checking|3900.50|2020350540|
|        33|         85|     Savings| 150.25| 655204919|
|        36|         27|    Checking|3700.00|4060537623|
|        45|         68|     Savings| 300.25|2343563725|
|        51|         72|     Savings| 375.75|1266528029|
|        28|          7|    Checking|2900.00|2088520729|
|        15|         47|     Savings| 700.75|1744154229|
|        27|         94|     Savings|  50.75|3045508432|
|        31|         71|     Savings| 125.75| 352326581|
|        98|         49|    Checking|9900.50|4068827442|
|         3|         78|     Savings|1500.00|3519015933|
|        83|         82|     Savings| 775.75| 293251303|
+----------+-----------+------------+-------+----------+
only showing top 20 rows
```

Next comes the scdtype-1 split condition and mapping.

```
11:47 AM (14s)                                                    12

dbtable_accounts.alias("tgt").merge(df_hash_accounts.alias("src"),"tgt.account_id = src.account_id")\
    .whenMatchedUpdate(
        set={
            "tgt.account_id": "src.account_id",
            "tgt.customer_id": "src.customer_id",
            "tgt.account_type": "src.account_type",
            "tgt.balance": "src.balance",
            "tgt.hashkey": "src.hash_key",
            "tgt.updated_date": current_timestamp(),
            "tgt.updated_by": lit("databricks-updated")
        }
    )\
    .whenNotMatchedInsert(
        values={
            "tgt.account_id": "src.account_id",
            "tgt.customer_id": "src.customer_id",
            "tgt.account_type": "src.account_type",
            "tgt.balance": "src.balance",
            "tgt.hashkey": "src.hash_key",
            "tgt.created_date": current_timestamp(),
            "tgt.created_by": lit("databricks"),
            "tgt.updated_date": current_timestamp(),
            "tgt.updated_by": lit("databricks")
        }
    ).execute()

(7) Spark Jobs
```

Finally, display the final data.

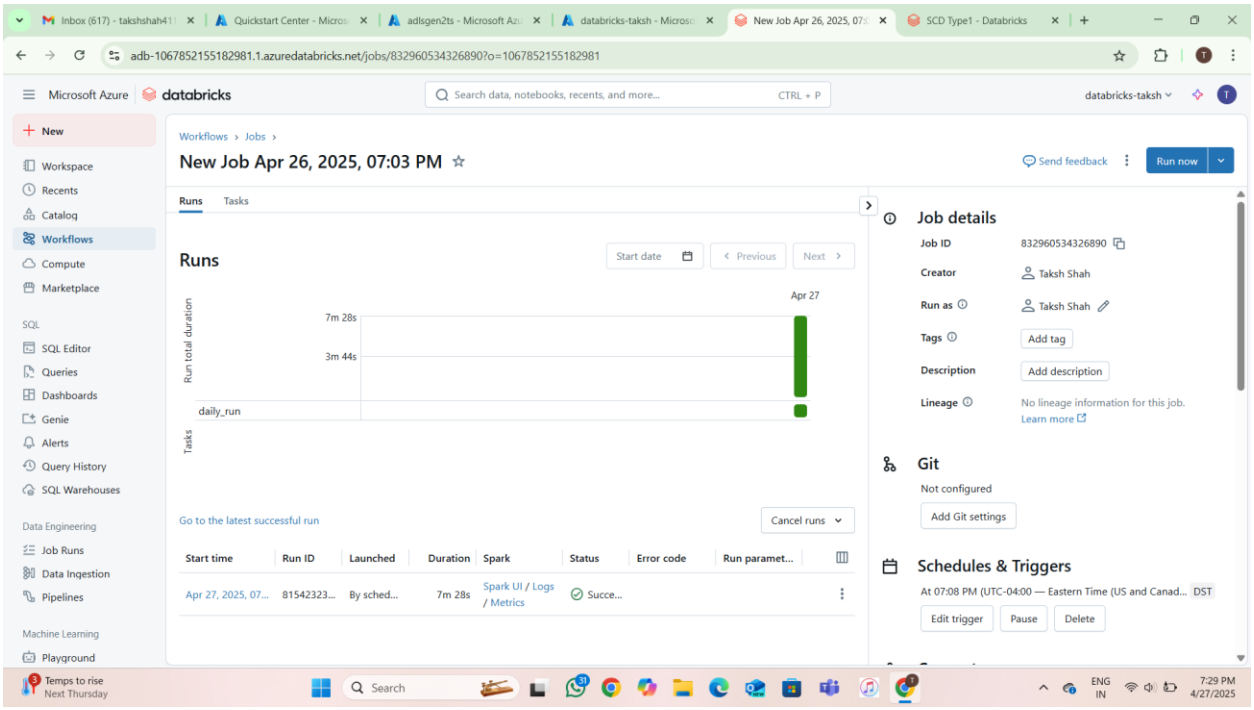
```
11:48 AM (2s)                                                     13                                    Python

display(spark.read.format("delta").option("header","true").load(tgt_path_accounts))

(1) Spark Jobs
```

Table +

	account_type	balance	created_by	created_date	updated_by	updated_date	hashkey
1	Savings	1000.5	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	4261402674
2	Checking	4900	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	3216528439
3	Savings	300.25	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	2800704470
4	Savings	75.25	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	3637748452
5	Checking	3500.5	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	868746038
6	Savings	175.75	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	3427479153
7	Savings	500	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	2866974084
8	Checking	7900.5	databricks	2025-04-26T15:47:21.135+00:...	databricks	2025-04-26T15:47:21.135+00:...	2650882798

Furthermore, create a scheduled workflow in databricks and provide a time at which the databricks notebook should run every day. Make sure to create a separate cluster to run this workflow, so we'll keep our main cluster only for the main code and a different cluster which will just be used for the workflow.

The image provided below shows a successful run after scheduling the trigger for 7:08 PM EST



Note that we are manually replacing the original dataset files with the new test files (or day2 files – which have new data) in the "dataset location" **BEFORE** this scheduled workflow triggers the pipeline, so when the pipeline is triggered, it'll get all the new data into our cloud folders (bronze, silver, gold).

Now, “final_joined_data”delta” present in silver layer is the file that we’ll create a Power BI report on, as per our project requirement. First, let’s assign the “storage blob data reader role” to ourself in the access control management section of our adlsgen2 account. Doing so will allow Power BI to access the adlsgen2 storage account when we connect them.

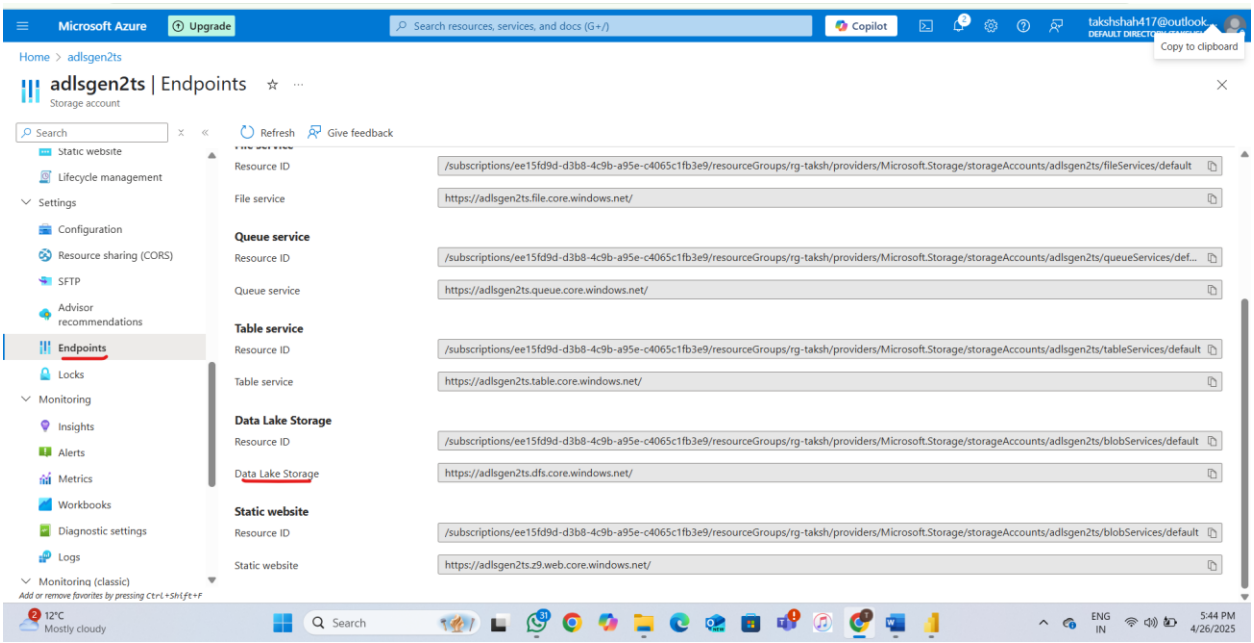
Home > adlsgen2ts | Access Control (IAM) >

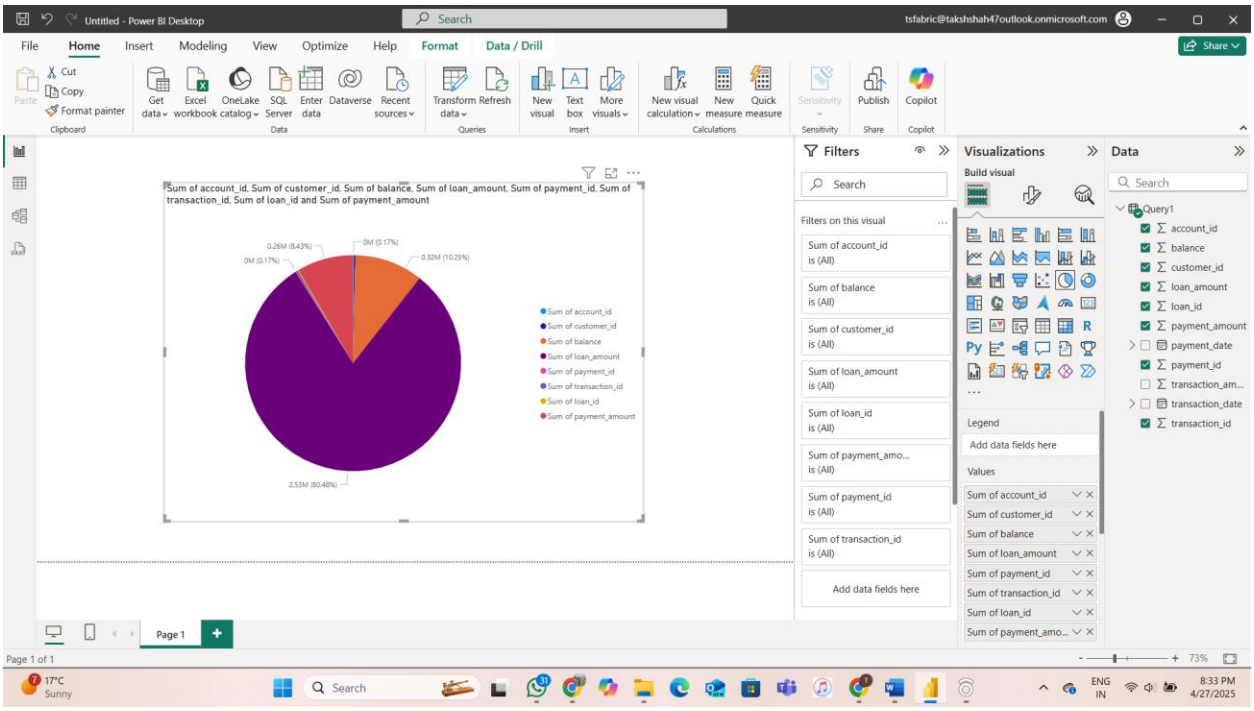
Add role assignment ...

Role | Members | Conditions | Review + assign

Role	Storage Blob Data Reader
Scope	/subscriptions/ee15fd9d-d3b8-4c9b-a95e-c4065c1fb3e9/resourceGroups/rg-taksh/providers/Microsoft.Storage/storageAccounts/adlsgen2ts
Members	Name: Taksh Shah(Guest) | Object ID: 385da847-977c-4c02-a5ae-79137cf48527 | Type: User
Description	No description
Condition	None

When connecting the adlsgen2 account to power bi, it’ll ask for a URL for adlsgen2, which can be found in the endpoints section of the settings in adlsgen2.





Power BI Report

Finally, this report can be published to Microsoft Fabrics workspace (make sure you turn on fabric capacity to finish this task).