Bootcamp Project 2

Transactions and Loan Data for a Customer

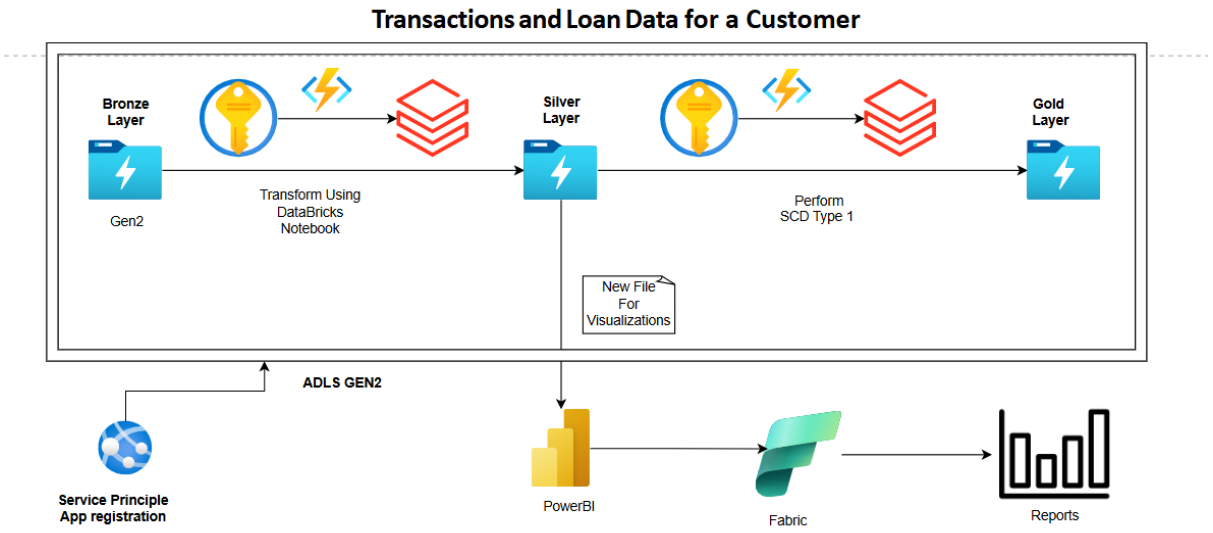
Table of Contents

Objective

The project aims to design and implement a robust and scalable data pipeline to process customer account data. The pipeline includes the following:

- Ingesting data from Azure Data Lake Storage Gen2 (ADLS Gen2) Bronze Layer.
- Transforming and cleaning the data in Databricks Notebooks for the Silver Layer.
- Writing the transformed data into the Gold Layer using SCD Type 1 technique.
- Visualizing data in Power BI using Delta format files from the Gold Layer.

Architecture Overview



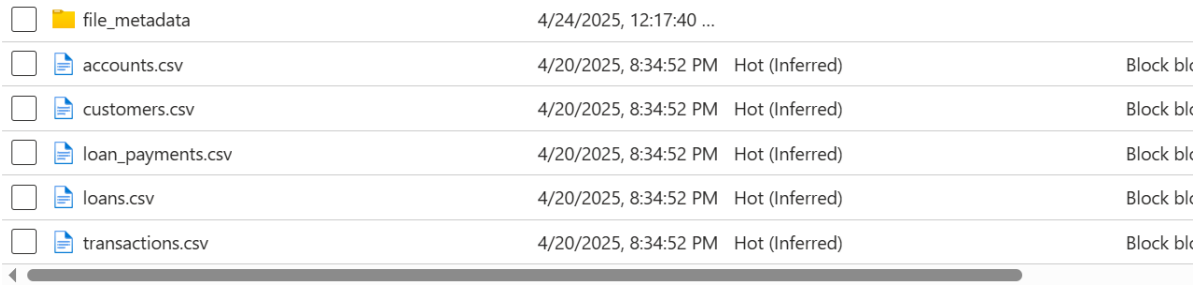
Components Used:

- **Azure Data Lake Storage Gen2 (Bronze, Silver, Gold Layers)**
- **Azure Key Vault**
- **Service Principle**
- **Azure Databricks**
- **Power BI**

Project Steps

Step 1: Data Ingestion (Bronze Layer)

- Source Location: Backend Team Storage Account (Simulated with Kaggle Dataset)
    - accounts.csv
    - customers.csv
    - loan_payments.csv
    - loans.csv
    - transactions.csv
- Sink Location: Your ADLS Gen2 Storage Account (Bronze Layer Folder)

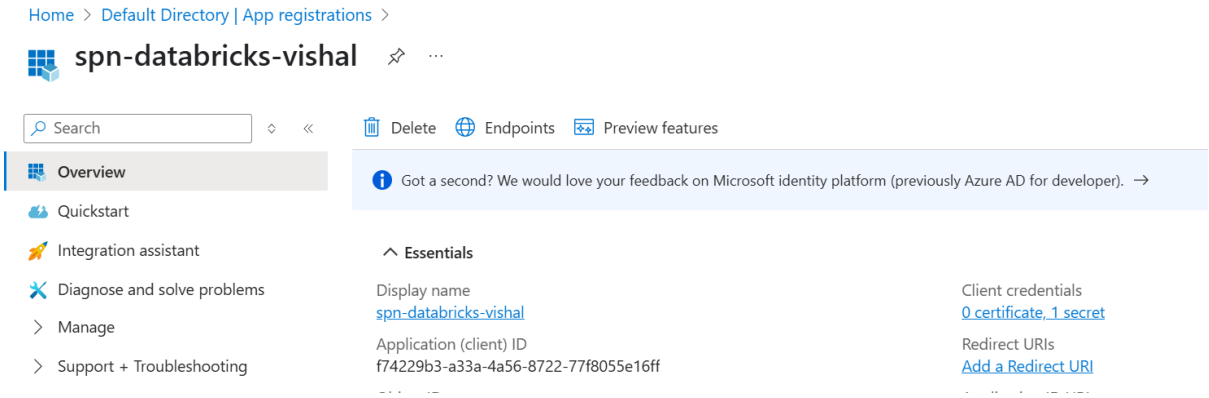


file_metadata	4/24/2025, 12:17:40 ...		
accounts.csv	4/20/2025, 8:34:52 PM	Hot (Inferred)	Block bl
customers.csv	4/20/2025, 8:34:52 PM	Hot (Inferred)	Block bl
loan_payments.csv	4/20/2025, 8:34:52 PM	Hot (Inferred)	Block bl
loans.csv	4/20/2025, 8:34:52 PM	Hot (Inferred)	Block bl
transactions.csv	4/20/2025, 8:34:52 PM	Hot (Inferred)	Block bl

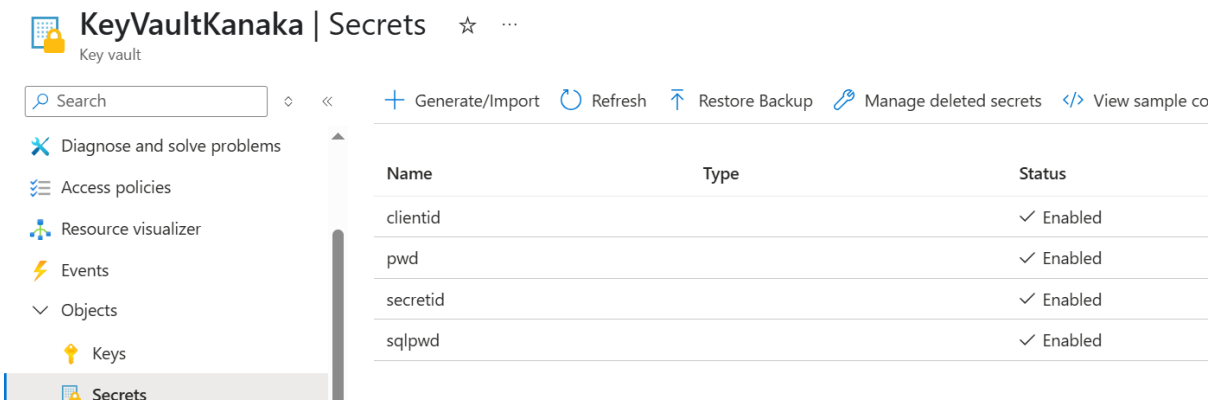
Step 2: Data Cleaning and Transformation in Databricks (Silver Layer)

Tasks:

- **Create a secret scope using Azure Key Vault and Service Principle**



    - Stored appid and secret in Azure Key Vault and gave storage contributor access to service principle.



- **Mount ADLS Gen2 using Databricks**

  Secrets List

3 days ago (1s) 7

```
dbutils.secrets.list("adlsconnection")
```

```
[SecretMetadata(key='clientid'),
 SecretMetadata(key='pwd'),
 SecretMetadata(key='secretid'),
 SecretMetadata(key='sqlpwd')]
```

Mount adls Gen2using secrets

22 hours ago (17s) 8

```
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": dbutils.secrets.get(scope="adlsconnection", key="clientid"),
  "fs.azure.account.oauth2.client.secret": dbutils.secrets.get(scope="adlsconnection", key="secretid"),
  "fs.azure.account.oauth2.client.endpoint": "https://login.microsoftonline.com/fcee7905-be7c-4a7c-b3f6-7c94700f97cb/
  oauth2/token"
}

dbutils.fs.mount(
  source = "abfss://input@adlsgen2vishal.dfs.core.windows.net/",
  mount_point = "/mnt/input1",
  extra_configs=configs)
```

```
True
```

- **Get the Modified Files Only**
  - Get the filenames and modified_time

08:42 AM (5s) 1 Python

```python
from pyspark.sql.functions import input_file_name
import os

bronze_path = "/mnt/input//Bronze"

files_info = dbutils.fs.ls(bronze_path)

bronze_files = [
    {"file_path": f.path, "file_name": os.path.basename(f.path), "modified_time": f.modificationTime}
    for f in files_info if f.path.endswith(".csv")  # or .parquet, etc.
]
```

  - Create a Meatdata Delta file to store filenames, modifiedtimes, primarycolumns

23 hours ago (10s) 2

```python
from pyspark.sql.functions import current_timestamp
from pyspark.sql.types import StructType, StructField, StringType, TimestampType



metadata_table_path = "/mnt/input/Bronze/file_metadata"

file_names = ["accounts.csv", "customers.csv", "loan_payments.csv", "loans.csv", "transactions.csv"]
primary_columns = ["account_id", "customer_id", "payment_id", "loan_id", "transaction_id"]

data = [(str(i), file_names[i], None, primary_columns[i]) for i in range(len(file_names))]

schema = StructType([
    StructField("file_id", StringType(), False),
    StructField("file_name", StringType(), True),
    StructField("modified_time", TimestampType(), True),
    StructField("primary_column", StringType(), True)
])
```

```python
df = spark.createDataFrame(data, schema)

df = df.withColumn("modified_time", lit("1900-01-01T00:00:00.000+00:00").cast(TimestampType()))

df.write.format("delta").mode("overwrite").save(metadata_table_path)

print(f" Metadata Delta table created at: {metadata_table_path}")
```

- Compare and get the modified files Info

```python
import pandas as pd
metadata_df = spark.read.format("delta").options( inferSchema="true").load("/mnt/input/Bronze/file_metadata")

# Step 1: Convert bronze files and metadata to Pandas
bronze_df = pd.DataFrame(bronze_files)
metadata_pd = metadata_df.toPandas()

# Ensure 'modified_time' is in datetime format
bronze_df["modified_time"] = pd.to_datetime(bronze_df["modified_time"])
metadata_pd["modified_time"] = pd.to_datetime(metadata_pd["modified_time"])

# Step 2: Merge on file name
merged = pd.merge(bronze_df, metadata_pd, on="file_name", how="left", suffixes=("", "_old"))

# Step 3: Replace NaT with a known early date for safe comparison
merged["modified_time_old"] = pd.to_datetime(merged["modified_time_old"])
merged["modified_time_old"].fillna(pd.Timestamp("1900-01-01"), inplace=True)
```

```python
# Step 4: Filter files that are new or modified
modified_files = merged[merged["modified_time"] > merged["modified_time_old"]]
modified_files.display()
```

- Update the Meatdata Delta File with new modifiedtimes

```python
from delta.tables import DeltaTable
from pyspark.sql.functions import col, lit
import datetime

modified_spark_df = spark.createDataFrame(modified_files[["file_name", "modified_time"]])

delta_table = DeltaTable.forPath(spark, "/mnt/input/Bronze/file_metadata")

for row in modified_spark_df.collect():
    file_name = row["file_name"]
    new_time = row["modified_time"]

    delta_table.update(
        condition = f"file_name = '{file_name}'",
        set = { "modified_time": lit(new_time) }
    )
```

- **Read CSV files from the Bronze Layer**

  Apply the following transformations:

  - **Remove duplicates using window functions.**
  - **Handle missing values.**

23 hours ago (<1s) 9 Python

```python
from pyspark.sql import DataFrame


def TransformBronzeFile(file_name: str,primary_key_column: str, bronze_path: str = "/mnt/input/Bronze") -> DataFrame:

    file_path = f"{bronze_path}/{file_name}"

    df = spark.read.option("inferSchema", True).option("header", True).csv(file_path)

    df = df.dropDuplicates()
    df = df.dropna(subset=[primary_key_column])

    return df
```

Generate (Ctrl + I)

- **Convert formats: output the combined file as Delta, others as Parquet.**

23 hours ago (47s) 10 Python

```python
silver_base_path = "/mnt/input/Silver"
modified_file_list = modified_files[["file_name", "primary_column"]].to_dict(orient="records")


for file_info in modified_file_list:
    file_name = file_info["file_name"]
    primary_column = file_info["primary_column"]

    # Transform the file
    cleaned_df = transform_bronze_file(file_name)

    base_name = file_name.replace(".csv", "")
    silver_path = f"{silver_base_path}/{base_name}"

    cleaned_df.write.mode("overwrite").option("header", "true").parquet(silver_path)
```

- **Join the datasets using appropriate keys:**
  - **Join columns: Account ID, Transaction ID, Customer ID, Loan ID, Payment ID, Amount, Dates**

08:43 AM (<1s) 14

```python
joined_df =(
    accounts
.join(customers, on="customer_id", how="inner")
.join(loans, on="customer_id", how="inner")
.join(loanpayments, on="loan_id", how='inner')
.join(transactions, on="account_id", how="inner")
)
```

joined_df: pyspark.sql.dataframe.DataFrame = [account_id: integer, loan_id: integer ... 19 more

08:47 AM (<1s) 15 Python

```python
final_joined_df=joined_df.select("account_id","loan_id","customer_id","balance","first_name","last_name","city",
"state","loan_amount","loan_term","payment_id","payment_date","payment_amount","transaction_id","transaction_date",
"transaction_amount",)
```

final_joined_df: pyspark.sql.dataframe.DataFrame = [account_id: integer, loan_id: integer ... 14 more fields]

- **Write the final output file in Delta format to Silver Layer.**

09:01 AM (11s) 16

```python
final_joined_df.write.mode("overwrite").option("header", "true").format("delta").save("/mnt/input/Silver/final")
display(spark.read.format("delta").load("/mnt/input/Silver/final"))
```

(10) Spark Jobs

Table

	account_id	loan_id	customer_id	balance	first_name	last_name	city
1	50	50	31	5100.5	David	Sanchez	North Bay
2	33	33	85	150.25	John	Harrison	Temagami
3	85	85	65	800.25	Daniel	Bryant	Elmvale
4	21	21	53	300.25	James	Jenkins	Queensville
5	3	3	78	1500	Abigail	Cole	Sundridge
6	4	4	34	3000.25	Olivia	Reed	Orillia
7	12	12	81	2700	Michael	Owens	Mattawa
8	56	56	28	5700	Emily	Edwards	Brantford

Step 3: SCD Type 1 in Databricks (Gold Layer)

- **Accounts**

2 days ago (<1s)

```python
accounts_src_path='/mnt/input/Silver/accounts.parquet'
accounts_tgt_path='/mnt/input/Gold/accounts'
```

```
%sql
create table if not exists accounts
(
  account_id int,
  customer_id int,
  account_type string,
  balance double,
  hashkey bigint,
  createdby string,
  createdDate timestamp,
  updatedby string,
  updatedDate timestamp
)
using delta
location '/mnt/input/Gold/accounts'
```

- **Read the file and generate hashkey column**

```
5
df_accounts=spark.read.format("parquet").option("header", "true").load(accounts_src_path)
```

```
6
%python
from pyspark.sql.functions import crc32, concat
df_accounts_hash=df_accounts.withColumn("hashkey",crc32(concat(*df_accounts.columns)))
```

- **Read the delta table and compare with src for new records**

7 Pyt

```python
from delta.tables import *
dtable_accounts = DeltaTable.forPath(spark, accounts_tgt_path)
dtable_accounts.toDF().show()
```

8

```python
%python
from pyspark.sql.functions import col

df_src_accounts = df_accounts_hash.alias("src").join(
    dtable_accounts.toDF().alias("tgt"),
    (col("src.account_id") == col("tgt.account_id")) & (col("src.hashkey") == col("tgt.hashkey")),
    "anti"
).select("src.*")
```

Perform Merge for SCD Implentation

9

```python
from pyspark.sql.functions import *
dtable_accounts.alias("tgt").merge(df_src_accounts.alias("src"),((col("src.account_id") == col("tgt.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.hashkey",
        "tgt.updatedDate":current_timestamp(),
        "tgt.updatedby":lit("databricks-update")
    })\
        .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.hashkey",
```

Customers

2 days ago (5s) 11

```
%sql
use catalog hive_metastore;
create table if not exists customers
(
    customer_id int,
    first_name string,
    last_name string,
    address string,
    state string,
    city string,
    zip string,
    hashkey bigint,
    createdby string,
    createdDate timestamp,
    updatedby string,
    updatedDate timestamp
)
using delta
location '/mnt/input/Gold/customers'
```

2 days ago (<1s) 16

```
%python
from pyspark.sql.functions import col

df_src_customers = df_customers_hash.alias("src").join(
    dtable_customers.toDF().alias("tgt"),
    (col("src.customer_id") == col("tgt.customer_id")) & (col("src.hashkey") == col("tgt.hashkey")),
    "anti"
).select("src.*")
```

2 days ago (9s) 17 Python

```
from pyspark.sql.functions import *
dtable_customers.alias("tgt").merge(df_src_customers.alias("src"),((col("src.customer_id") == col("tgt.customer_id"))))\
    .whenMatchedUpdate(set={
        "tgt.customer_id":"src.customer_id",
        "tgt.first_name":"src.first_name",
        "tgt.last_name":"src.last_name",
        "tgt.address":"src.address",
        "tgt.city":"src.city",
        "tgt.state":"src.state",
        "tgt.zip":"src.zip",
        "tgt.hashkey":"src.hashkey",
        "tgt.updatedDate":current_timestamp(),
        "tgt.updatedby":lit("databricks-update")
    })\
        .whenNotMatchedInsert(values={
       "tgt.customer_id":"src.customer_id",
        "tgt.first_name":"src.first_name",
        "tgt.last_name":"src.last_name",
```

Loan_payments

Yesterday (22s) 20

```
%sql
use catalog hive_metastore;
create table if not exists loan_payments
(
    payment_id int,
    loan_id int,
    payment_date timestamp,
    payment_amount decimal,
    hashkey bigint,
    createdby string,
    createdDate timestamp,
    updatedby string,
    updatedDate timestamp
)
using delta
location '/mnt/input/Gold/loan_payments'
```

OK

Yesterday (<1s) 24

```
%python
from pyspark.sql.functions import col

df_src_loan_payments = df_loan_payments_hash.alias("src").join(
    dtable_loan_payments.toDF().alias("tgt"),
    (col("src.payment_id") == col("tgt.payment_id")) & (col("src.hashkey") == col("tgt.hashkey")),
    "anti"
).select("src.*")
```

df_src_loan_payments: pyspark.sql.dataframe.DataFrame = [payment_id: integer, loan_id: integer ... 3 more fields]

Yesterday (18s) 25 Python

```
from pyspark.sql.functions import *
dtable_loan_payments.alias("tgt").merge(df_src_loan_payments.alias("src"),((col("src.payment_id") == col("tgt.
payment_id"))))\
    .whenMatchedUpdate(set={
        "tgt.payment_id":"src.payment_id",
        "tgt.loan_id":"src.loan_id",
        "tgt.payment_date":"src.payment_date",
        "tgt.payment_amount":"src.payment_amount",
        "tgt.hashkey":"src.hashkey",
        "tgt.updatedDate":current_timestamp(),
        "tgt.updatedby":lit("databricks-update")
    })\
        .whenNotMatchedInsert(values={
       "tgt.payment_id":"src.payment_id",
        "tgt.loan_id":"src.loan_id",
        "tgt.payment_date":"src.payment_date",
        "tgt.payment_amount":"src.payment_amount",
```

Loans

Yesterday (21s) 28

```
%sql
use catalog hive_metastore;
create table if not exists loans
(
    loan_id int,
    customer_id int,
    loan_amount decimal,
    interest_rate decimal,
    loan_term int,
    hashkey bigint,
    createdby string,
    createdDate timestamp,
    updatedby string,
    updatedDate timestamp
)
using delta
location '/mnt/input/Gold/loans'
```

Yesterday (<1s) 32

```
%python
from pyspark.sql.functions import col

df_src_loans = df_loans_hash.alias("src").join(
    dtable_loans.toDF().alias("tgt"),
    (col("src.loan_id") == col("tgt.loan_id")) & (col("src.hashkey") == col("tgt.hashkey")),
    "anti"
).select("src.*")
```

df_src_loans: pyspark.sql.dataframe.DataFrame = [loan_id: integer, customer_id: integer ... 4 more fields]

Yesterday (20s) 33 Python

```
from pyspark.sql.functions import *
dtable_loans.alias("tgt").merge(df_src_loans.alias("src"),((col("src.loan_id") == col("tgt.loan_id"))))\
    .whenMatchedUpdate(set={
        "tgt.loan_id":"src.loan_id",
        "tgt.customer_id":"src.customer_id",
        "tgt.loan_amount":"src.loan_amount",
        "tgt.interest_rate":"src.interest_rate",
        "tgt.loan_term":"src.loan_term",
        "tgt.hashkey":"src.hashkey",
        "tgt.updatedDate":current_timestamp(),
        "tgt.updatedby":lit("databricks-update")
    })\
        .whenNotMatchedInsert(values={
       "tgt.loan_id":"src.loan_id",
        "tgt.customer_id":"src.customer_id",
        "tgt.loan_amount":"src.loan_amount",
        "tgt.interest_rate":"src.interest_rate",
```

Transactions

Yesterday (6s) 36

```
%sql
use catalog hive_metastore;
create table if not exists transactions
(
    transaction_id int,
    account_id int,
    transaction_date timestamp,
    transaction_amount decimal,
    transaction_type string,
    hashkey bigint,
    createdby string,
    createdDate timestamp,
    updatedby string,
    updatedDate timestamp
)
using delta
location '/mnt/input/Gold/transactions'
```

Yesterday (<1s) 40

```
%python
from pyspark.sql.functions import col

df_src_transactions = df_transactions_hash.alias("src").join(
    dtable_transactions.toDF().alias("tgt"),
    (col("src.transaction_id") == col("tgt.transaction_id")) & (col("src.hashkey") == col("tgt.hashkey")),
    "anti"
).select("src.*")
```

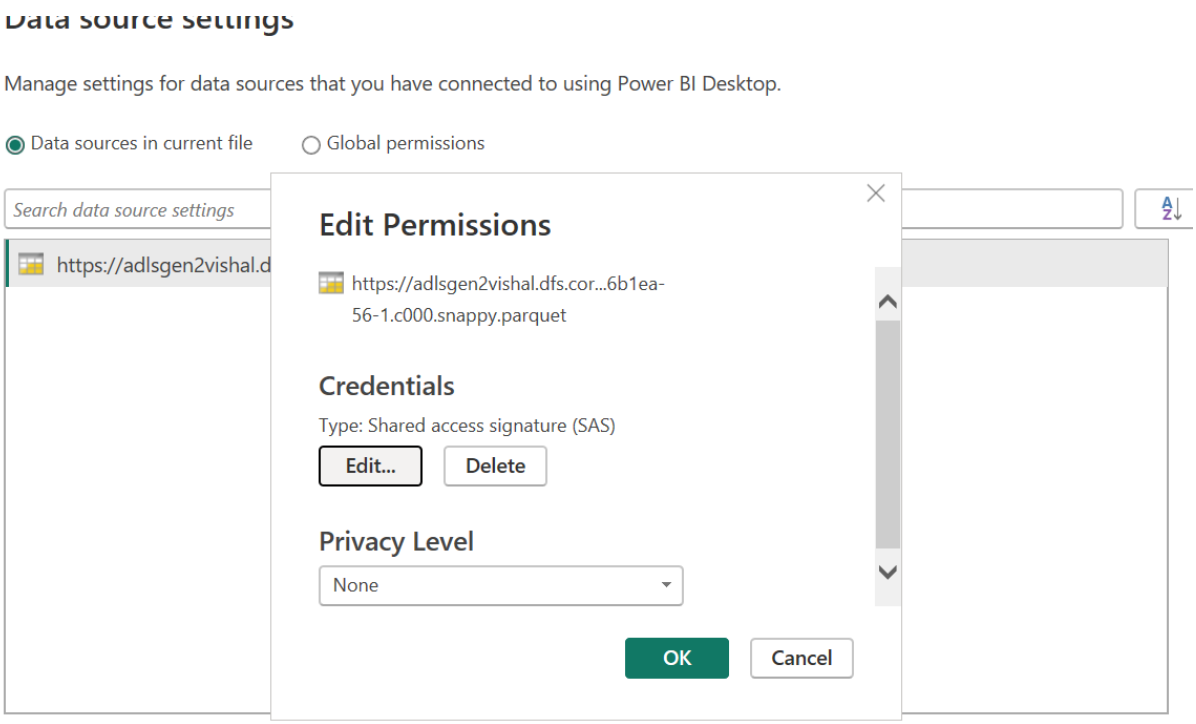
df_src_transactions: pyspark.sql.dataframe.DataFrame = [transaction_id: integer, account_id: integer ... 4 more fields]

Yesterday (9s) 41 Python

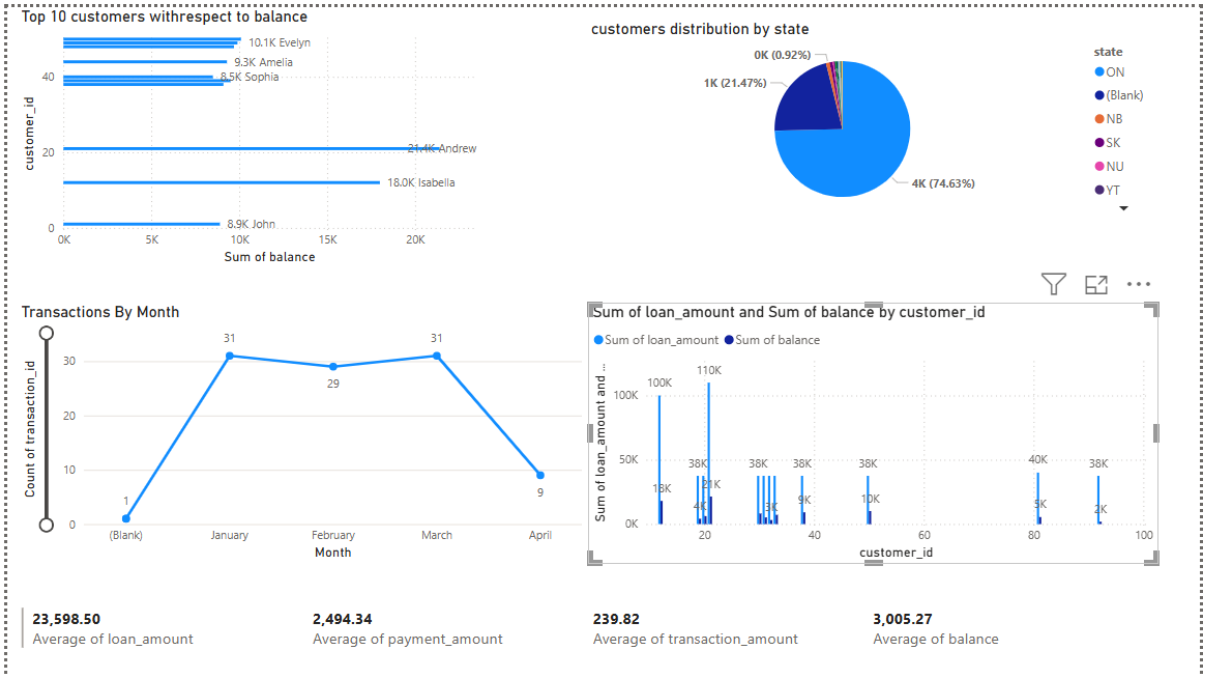
```
from pyspark.sql.functions import *
dtable_transactions.alias("tgt").merge(df_src_transactions.alias("src"),((col("src.transaction_id") == col("tgt.
transaction_id"))))\
    .whenMatchedUpdate(set={
        "tgt.transaction_id":"src.transaction_id",
        "tgt.account_id":"src.account_id",
        "tgt.transaction_date":"src.transaction_date",
        "tgt.transaction_amount":"src.transaction_amount",
        "tgt.transaction_type":"src.transaction_type",
        "tgt.hashkey":"src.hashkey",
        "tgt.updatedDate":current_timestamp(),
        "tgt.updatedby":lit("databricks-update")
    })\
        .whenNotMatchedInsert(values={
      "tgt.transaction_id":"src.transaction_id",
        "tgt.account_id":"src.account_id",
        "tgt.transaction_date":"src.transaction_date",
```

Step 4: Data Visualization in Power BI

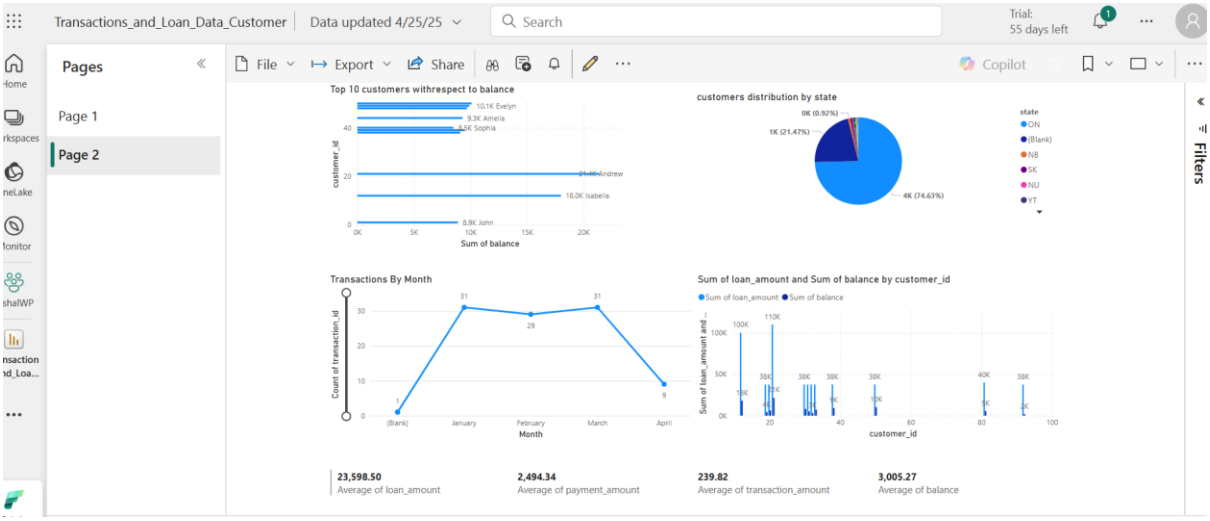
- **Connect Power BI to ADLS Gen2 using Azure Data Lake Gen2 Connector**



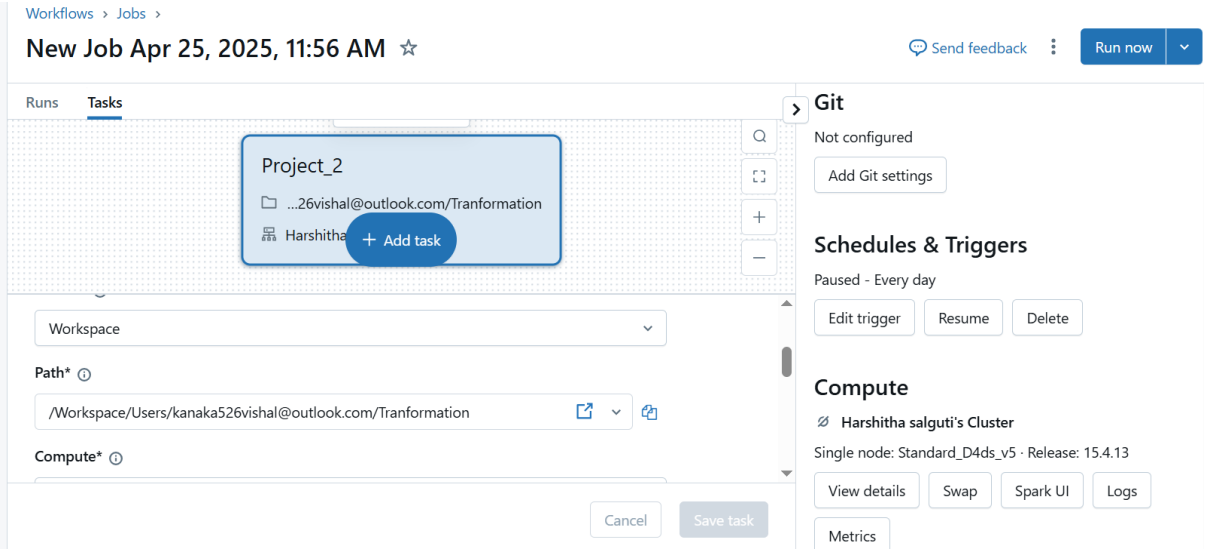
- **Build Visualizations:**



- **Publish to Fabric Workspace**



Step 5: Automation and scheduling the Notebooks



Deliverables

Git Repository

https://github.com/VishalKanaka/Data-Tranformation-and-SCD-Using-DataBricks.git