By writing SQL code similar to the following script, you could also create tables using parquet files stored within your ADLSgen2 account.

CREATE TABLE nyc\_taxi.yellowcab

USING parquet

LOCATION '/data/nyc\_taxi/parquet/yellowcab.parquet

There are a variety of Databricks datasets that come mounted with DBFS and can be accessed through the following Python code: display(dbutils.fs.ls('/databricks-datasets')).

To create the secret scope in Databricks, navigate to https://<DATABRICKS-INSTANCE>#secrets/createScope and replace <DATABRICKS-INSTANCE> with your own Databricks URL instance.

Paste the following code into your Python Databricks notebook and replace the adlsAccountName, adlsContainerName, adlsFolderName, and mountpoint with your own ADLS gen2 values. Also ensure that the ClientId, ClientSecret, and TenantId match the secret names that you provided in your Key Vault in Azure portal.

# Python code to mount and access Azure Data Lake Storage Gen2 Account from Azure Databricks with Service Principal and OAuth

# Define the variables used for creating connection strings

adlsAccountName = "adlsg2v001"

adlsContainerName = "data"

adlsFolderName = "raw"

mountPoint = "/mnt/raw"

# Application (Client) ID

applicationId = dbutils.secrets.get(scope="akv-0011",key="ClientId")

# Application (Client) Secret Key

authenticationKey = dbutils.secrets.get(scope="akv-0011",key="ClientSecret")

# Directory (Tenant) ID

tenandId = dbutils.secrets.get(scope="akv-0011",key="TenantId")

endpoint = "https://login.microsoftonline.com/" + tenandId + "/oauth2/token"

source = "abfss://" + adlsContainerName + "@" + adlsAccountName + ".dfs.core.windows.net/" + adlsFolderName

# Connecting using Service Principal secrets and OAuth

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": applicationId,

"fs.azure.account.oauth2.client.secret": authenticationKey,

"fs.azure.account.oauth2.client.endpoint": endpoint}

# Mount ADLS Storage to DBFS only if the directory is not already mounted

if not any(mount.mountPoint == mountPoint for mount in dbutils.fs.mounts()):

dbutils.fs.mount(

source = source,

mount\_point = mountPoint,

extra\_configs = configs)

Run the following command to list the content on your mounted store.

dbutils.fs.ls('mnt/raw')

Use the %fs magic command to view the same list in tabular format.

#dbutils.fs.ls('mnt/raw')

%fs

ls "mnt/raw"

By running this code, you will notice an error. I deliberately added the commented out #dbutils.fs.ls code to show that if you happen to have comments in the same code block as the %fs command, you will receive an error.

To get around this error, remove any comments in the code block which contains the %fs command.

%fs

ls "mnt/raw"

Read the data from the mount point by simply creating a data frame to read the file by using the spark.read command.

df = spark.read.json("/mnt/raw/Customer1.json")

display(df)

Run the following command to list the mounts that have been created on this account.

display(dbutils.fs.mounts())

Run the following command to unmount the mounted directory.

# Unmount only if directory is mounted

if any(mount.mountPoint == mountPoint for mount in dbutils.fs.mounts()):

dbutils.fs.unmount(mountPoint)

GRANT SELECT ON tbl\_nyctaxidata TO datadevs

GRANT SELECT (mileage, city, state, zip) ON nyctaxidata TO dataanalysts

The Unity Catalog also supports SQL view-based access control which allows you to create complex aggregated views for certain users or groups.

CREATE VIEW sum\_nyctaxi\_mileage AS

SELECT date, country, city, state SUM(\*) AS Total\_Miles FROM tbl\_nyctaxidata

GROUP BY date, country, city, state

GRANT SELECT ON sum\_nyctaxi\_mileage TO dataanalysts

With the Unity Catalog’s attribute-based controls allows, you can tag columns as PII and manage access to all columns tagged as PII in a single rule:

ALTER TABLE tbl\_nyctaxidata ADD ATTRIBUTE pii ON credit\_card\_number

ALTER TABLE tbl\_nyctaxicustomers ADD ATTRIBUTE pii ON email

GRANT SELECT ON TABLE tbl\_nyctaxidata

HAVING ATTRIBUTE NOT IN (pii)

TO datamanagers

Note that attribute-based controls can also be applied to MLflow models using the following sample SQL command.

GRANT EXECUTE ON MODELS HAVING ATTRIBUTE (midwest\_churn)

TO midwest\_regionalmanagers

Delta Live Tables can be created using the syntax shown in the code below and can be chained to other scripts in the form of dependencies to develop ELT scripts for multiple stages (raw, staging, curated). These pipelines which capture lineage and dependencies can be visually tracked, tested, re-started, and maintained both manually and automatically.

CREATE live TABLE nyc\_taxi

COMMENT "Raw Table for nyc\_taxi"

AS

SELECT \* FROM cloud\_files("/data/nyc\_taxi", "parquet")

The Databricks release pipeline tasks shown in Figure 3-45 requires the installation of the [Databricks Script Deployment Task by Data Thirst from the Visual Studio Marketplace](https://marketplace.visualstudio.com/items?itemName=DataThirstLtd.databricksDeployScriptsTasks). The marketplace link can be found here <https://marketplace.visualstudio.com/items?itemName=DataThirstLtd.databricksDeployScriptsTasks>.

The following Python script demonstrates just how easy it is to read, write, and transform data in Synapse Analytics Dedicated SQL Pools directly from a Databricks notebook.

# Read data from Synapse SQL table.

df = spark.read \

.format("com.databricks.spark.sqldw") \

.option("url", "jdbc:sqlserver://<connection-string>") \

.option("tempDir", " https://adlsg2v001.dfs.core.windows.net/data/raw") \

.option("forwardSparkAzureStorageCredentials", "true") \

.option("dbTable", "<table-name>") \

.load()

# Read Azure Synapse query and load into Databricks Dataframe.

df = spark.read \

.format("com.databricks.spark.sqldw") \

.option("url", "jdbc:sqlserver://<connection-string>") \

.option("tempDir", "https://adlsg2v001.dfs.core.windows.net/data/raw") \

.option("forwardSparkAzureStorageCredentials", "true") \

.option("query", "select \* from table group by n") \

.load()

# Apply data transformations and write back to a Synapse SQL Table.

df.write \

.format("com.databricks.spark.sqldw") \

.option("url", "jdbc:sqlserver://<connection-string>") \

.option("forwardSparkAzureStorageCredentials", "true") \

.option("dbTable", "<table-name>") \

.option("tempDir", " https://adlsg2v001.dfs.core.windows.net/data/raw ") \

.save()

The function works by simply writing a SQL statement similar to the following code by specifying what to encrypt as well as a 32bit encryption key. SELECT aes\_encrypt(‘MY SECRET’, ‘mykey’). To decrypt your data, simply run the aes\_decrypt () function with code that is similar to the following, which wraps your encrypted value and key within the decrypt function and casts it as a string: SELECT CAST(aes\_decrypt(unbase64(‘encryptedvalue’), ‘mykey’) as string)

The is\_Member function can be used as a filter in a SQL statement to only retrieve rows for which the current user has access to. The following SQL code creates a view on tbl\_encrypt to only show groups that a user has access to.

CREATE VIEW vw\_decrypt\_data

AS

SELECT \* FROM tbl\_encrypt where is\_Member(group\_name)

You could write another SQL query similar to the following to decrypt your data dynamically based on the group that you have row-level access to. Additionally, if an encryption key is ever deleted from tbl\_encrypt, then the following query will return ‘nulls’ for the rows that you may still have access to but since the encryption is no longer part of the lookup table, you will not be able to decrypt the data and will only see ‘nulls’ in place of the actual data. This pattern prevents the need from having to store duplicate versions of data by dynamically decrypting row-level data based on group-based membership.

SELECT

cast(aes\_decrypt(a.name, DA.encryption\_key) as string) Name,

cast(aes\_decrypt(a.email, DA.encryption\_key) as string) Email,

cast(aes\_decrypt(a.ssn, DA.encryption\_key) as string) SSN,

FROM

tbl\_data a

LEFT JOIN vw\_decrypt\_data b ON a.group\_name = b.group\_name

Here is the sample SQL Query which I ran in the Databricks SQL Analytics Workspace:

SELECT

SUM(l\_extendedprice\* (1 - l\_discount)) AS revenue

FROM

samples.tpch.lineitem,

samples.tpch.part

WHERE

(

p\_partkey = l\_partkey

AND p\_brand = 'Brand#12'

AND p\_container IN ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')

AND l\_quantity >= 1 AND l\_quantity <= 1 + 10

AND p\_size BETWEEN 1 AND 5

AND l\_shipmode IN ('AIR', 'AIR REG')

AND l\_shipinstruct = 'DELIVER IN PERSON'

)

OR

(

p\_partkey = l\_partkey

AND p\_brand = 'Brand#23'

AND p\_container IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')

AND l\_quantity >= 10 AND l\_quantity <= 10 + 10

AND p\_size BETWEEN 1 AND 10

AND l\_shipmode IN ('AIR', 'AIR REG')

AND l\_shipinstruct = 'DELIVER IN PERSON'

)

OR

(

p\_partkey = l\_partkey

AND p\_brand = 'Brand#34'

AND p\_container IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')

AND l\_quantity >= 20 AND l\_quantity <= 20 + 10

AND p\_size BETWEEN 1 AND 15

AND l\_shipmode IN ('AIR', 'AIR REG')

AND l\_shipinstruct = 'DELIVER IN PERSON'

)

For the NOT NULL constraint, you can add this within the create table statement’s schema definition, as shown in the code below.

CREATE TABLE Customers (

id INT NOT NULL,

FirstName STRING,

MiddleInitial STRING NOT NULL,

LastName STRING,

RegisterDate DATETIME

) USING DELTA;

## You can also drop a NOT NULL constraint or add a new NOT NULL constraint by using an ALTER TABLE command, as shown in the code below.

ALTER TABLE Customers CHANGE COLUMN MiddleInitial DROP NOT NULL;

ALTER TABLE Customers CHANGE COLUMN FirstName SET NOT NULL;

## The CHECK constraint can be added using an ALTER TABLE ADD CONTRAINT and ALTER TABLE DROP CONTRAINT commands. This will ensure that all rows meet the desired constraint conditions prior to adding it to the table, as shown in the code below.

ALTER TABLE Customers ADD CONSTRAINT ValidDate CHECK (RegisterDate > '1900-01-01');

You will need to begin by obtaining the max value of the CustomerID column by running the following PySpark code:

maxCustomerID = spark.sql(“select max(CustomerID) Customer ID from Customer”).first()[0]

The next block of code will use the maxCustomerID created in the previous code and it will apply a unique id to each record

Id = (

Id.withColumn(“CustomerID, maxCustomerID= monotonically\_increasing\_id())

)

Yet another method uses the row\_number().over() partition function which leads to slow performance. This pattern will give you the sequentially increasing numbers for the identity columns which were limitation in the previous function, but with huge cost and performance implications due to the sort that will need to happen. The PySpark code to achieve this would be as follows:

window = Window.orderBy(“CustomerFirstName”)

Id = (

Id.withColumn(“CustomerID”, maxCustomeID=row\_number().over(window))

)

The following CREATE TABLE SQL code shows how this works in practice. The ALWAYS option prevents users from inserting their own identity columns. ALWAYS can be replaced with BY DEFAULT which will allow uses to specify the Identity values. START WITH 0 INCREMENT BY 1 can be altered and customized as needed.

CREATE TABLE Customer

(

CustomerID bigint GENERATED ALWAYS AS IDENTITY (START WITH 0 INCREMENT BY 1),

CustomerFirstName string,

CustomerLastName string

)

Delta Live Tables (CDC) will need to be enabled within the pipeline settings of each pipeline by simply adding to the following configuration settings.

{

"configuration": {

"pipelines.applyChangesPreviewEnabled": "true"

}

}

Notice that the code shown below does not accept the MERGE command and that there is no reference to the MERGE command at the surface. Since this feature provides an abstraction layer which converts your code for you and handles the MERGE functionality behind the scenes, it simplifies the inputs that are required. Notice from the code below, that KEYS are accepted to simplify the JOIN command by joining on the specified keys for you. Also, changes will only be applied WHERE a condition is met. There is also an option to make no updates if nothing has changed with IGNORE NULL UPDATES. Like applying changes into a target table, you can APPLY AS DELETE WHEN a condition has been met rather than upserting the data. SEQUENCE BY will define the logical order of the CDC events to handle data which arrives out of order.

APPLY CHANGES INTO LIVE.tgt\_DimEmployees

FROM src\_Employees

KEYS (keys)

[WHERE condition]

[IGNORE NULL UPDATES]

[APPLY AS DELETE WHEN condition]

SEQUENCE BY orderByColumn

[COLUMNS {columnList | \* EXCEPT (exceptColumnList)}]

DLT’s CDC features can be integrated with streaming cloudfile sources, where we would begin by defining the cloudfile configuration details within a data frame, as shown in the Python code below which can be run in a Python Databricks notebook. This was also covered in greater detail in Chapter 17 – Auto Loader.

cloudfile = {

"cloudFiles.subscriptionID": subscriptionId,

"cloudFiles.connectionString": queueconnectionString,

"cloudFiles.format": "json",

"cloudFiles.tenantId": tenantId,

"cloudFiles.clientId": clientId,

"cloudFiles.clientSecret": clientSecret,

"cloudFiles.resourceGroup": resourceGroup,

"cloudFiles.useNotifications": "true",

"cloudFiles.schemaLocation": "/mnt/raw/Customer\_stream/\_checkpoint/",

"cloudFiles.schemaEvolutionMode": "rescue",

"rescueDataColumn":"\_rescued\_data"

}

Once the cloudfile configuration details are defined within a data frame, create a dlt view for the Bronze zone, as shown in the code below. This code will read the streaming cloudfile source data and incrementally maintain the most recent updates to the source data within the Bronze DLT view.

@dlt.view(name=f"bronze\_SalesLT\_DimCustomer")

def incremental\_bronze();

df = (spark

.readStream

.format"cloudFiles")

.options(\*\*cloudfile)

.load("/mnt/raw/Customer\_stream/\_checkpoint/")

return df

After the Bronze view is created, you’ll also need to create the target Silver table with the following Python code.

dlt.create\_target\_table("silver\_SalesLT\_DimCustomer")

The next block of Python code will apply the DLT changes from the DimCustomer Bronze source view to the DimCustomer Silver target table with CustomerID as the commonly identified join key on both tables and with UpdateDate as the sequence\_by identifier.

dlt.apply\_changes(

target = “silver\_SalesLT\_DimCustomer”,

source = “bronze\_SalesLT\_DimCustomer”,

keys = [“CustomerID”],

sequence\_by = col(“UpdateDate”)

)

For more information on Databricks best practices, please read the contents of the following GitHub Repository: <https://github.com/Azure/AzureDatabricksBestPractices/blob/master/toc.md>