Here is a script that can be run to enable failover. To begin, you can run a command to show the replication accounts and databases and then enable failover to one or multiple accounts. Db1 in the script below is the primary database. Finally, a secondary (replica) database can be switched to a primary using the last script in this section.

-- show replication accounts and databases

show replication accounts;

show replication databases;

-- enable failover to multiple accounts

alter database db1 enable failover to accounts org.account2, org.account3;

-- change a secondary database to the primary database.

alter database db2 primary;

The following script demonstrates an example of how to begin creating an ADLSgen2 storage integration with Snowflake. It begins by creating the storage integration and a stage to the ADLS gen2 account and container with parquet file format using SAS token credentials. The next part of the code shows how to copy data into a Snowflake table from the ADLS gen2 stage using pattern matching. Finally, the last section of the script shows two options for copying data from the Snowflake table into ADLS gen2.

-- create a storage integration

create storage integration azure\_int

type = external\_stage

storage\_provider = azure

enabled = true

azure\_tenant\_id = 'TENANAT-ID'

storage\_allowed\_locations = ('azure://myaccount.blob.core.windows.net/mycontainer1/mypath1/', 'azure://myaccount.blob.core.windows.net/mycontainer2/mypath2/')

storage\_blocked\_locations = ('azure://myaccount.blob.core.windows.net/mycontainer1/mypath1/sensitivedata/', 'azure://myaccount.blob.core.windows.net/mycontainer2/mypath2/sensitivedata/');

-- create a stage with SAS token and parquet file format

create or replace stage ADLS2

url='azure:// myaccount.blob.core.windows.net/ mycontainer1'

credentials=(azure\_sas\_token='SASTOKEN')

file\_format = PARQUET\_FILE\_FORMAT;

-- copy into Snowflake Table from ADLS2 Stage

copy into SnowflakeTable1

from @ADL2

pattern='.\*sales.\*.parquet';

-- copy into ADLS2 stage and new folder called d1 from Snowflake Table

copy into @ADLS/d1 from SnowflakeTable1;

-- copy into specified ADLS gen2 account, container, and folder from Snowflake Table

copy into azure://myaccount.blob.core.windows.net/data/Table1/ from SnowflakeTable1 storage\_integration = myint;

The following script shows how to create a notification integration using an Azure Event grid subscription. It then moves on to describe how to create a Snowpipe with a number of parameters including the auto\_ingest parameter which determines whether to automatically load data files from the specified external stage when event notifications are received from a configured message service. The last part of the Snowpipe script specifies the COPY INTO commands along with file formats.

-- create a notification integration

CREATE NOTIFICATION INTEGRATION notification\_int

ENABLED = true

TYPE = QUEUE

NOTIFICATION\_PROVIDER = AZURE\_STORAGE\_QUEUE

AZURE\_STORAGE\_QUEUE\_PRIMARY\_URI = '<queue\_URL>'

AZURE\_TENANT\_ID = '<directory\_ID>';

-- create a Snowpipe

create pipe azure\_pipe

auto\_ingest = true

integration = 'notification\_int'

as

copy into db\_snowpipe.public.table

from @db\_snowpipe.public.stage

file\_format = (type = 'JSON');

The script below shows an example of how to read and write data to Snowflake. The script begins by setting the ADLSgen2 context and keys. It goes on to show how to set the various Snowflake options. Finally, it shows how to write a sql query to read from Snowflake and to write 5 rows into Snowflake. Note that the select query can be more complex to accommodate complex joins as well.

-- connecting to ADLS2 account

spark.conf.set(

"fs.azure.account.key.rl001adls2.dfs.core.windows.net",

"ENTER-KEY"

)

-- setting Snowflake options

# snowflake connection options

options = {

"sfUrl": "snow.east-us-2.azure.snowflakecomputing.com",

"sfUser": "accountadmin",

"sfPassword": "Admin123!",

"sfDatabase": "snowflake\_sample\_data",

"sfSchema": "tpcds\_sf10tcl",

"sfWarehouse": "COMPUTE\_WH"

}

-- executing a query to read from Snowflake table

df = spark.read

.format("snowflake")

.options(\*\*options)

.option("query", "select \* from snowflake\_sample\_data.tpcds\_sf10tcl.item").load()

df.show()

-- writing to Snowflake

spark.range(5).write

.format("snowflake")

.options(\*\*options2)

.option("dbtable", "TEST\_DEMO")

.save()

An example of transforming data while loading it using the COPY INTO command can be seen in this script shown below which basically replaces the ~ character from the columns before loading it from ADLS gen2 into Snowflake.

-- Transforming data while loading it into Snowflake from ADLS gen2

COPY INTO AUTHOR (AUTHOR\_CODE, DESCRIPTION)

FROM

(SELECT REPLACE($1,'~',''),REPLACE($2, '~','')

FROM @MY\_ADLS2\_ACCOUNT/raw/AUTHOR.txt)

FILE\_FORMAT = AUTHOR\_FILE\_FORMAT;

The following script shows an example of flattening JSON and XML data. It begins by showing you how to create a table with VARIANT data type, along with scripts for creating both JSON and XML file formats. It then provides sample XML, JSON, and nested JSON data along with the corresponding SQL statements containing the FLATTEN commands. These SQL statements can also be used in the COPY INTO command to transform and FLATTEN data while loading it into a Snowflake table from your ADLS gen2 account.

-- Creating a new Snowflake table with VARIANT column

CREATE TABLE "DEMO\_DB"."PUBLIC"."FLATTEN\_RAW\_JSON\_TABLE" ("FLATTEN\_RAW\_COL" VARIANT)

COMMENT = 'Store JSON Data';

-- Creating a JSON file format

CREATE FILE FORMAT "DEMO\_DB"."PUBLIC".JSON

TYPE = 'JSON'

COMPRESSION = 'AUTO'

ENABLE\_OCTAL = FALSE

ALLOW\_DUPLICATE = FALSE

STRIP\_OUTER\_ARRAY = TRUE

STRIP\_NULL\_VALUES = FALSE

IGNORE\_UTF8\_ERRORS = FALSE;

-- Creating an XML file format

CREATE FILE FORMAT "DEMO\_DB"."PUBLIC".XML

TYPE = 'XML'

COMPRESSION = 'AUTO'

PRESERVE\_SPACE = FALSE

STRIP\_OUTER\_ELEMENT = TRUE

DISABLE\_SNOWFLAKE\_DATA = FALSE

DISABLE\_AUTO\_CONVERT = FALSE

IGNORE\_UTF8\_ERRORS = FALSE;

-- Sample XML data

<?xml version='1.0' encoding='UTF-8'?>

<dataset>

<AUTHOR AUTHOR\_UID = 1>

<FIRST\_NAME>Ron</FIRST\_NAME>

<MIDDLE\_NAME/>

<LAST\_NAME>LEsteve</LAST\_NAME>

</AUTHOR>

<AUTHOR AUTHOR\_UID = 2>

<FIRST\_NAME>Sam</FIRST\_NAME>

<MIDDLE\_NAME>Smith</MIDDLE\_NAME>

<LAST\_NAME>Broadwhick</LAST\_NAME>

</AUTHOR>

<AUTHOR AUTHOR\_UID = 3>

<FIRST\_NAME>Kathy</FIRST\_NAME>

<MIDDLE\_NAME>L</MIDDLE\_NAME>

<LAST\_NAME>Salisbery</LAST\_NAME>

</AUTHOR>

<AUTHOR AUTHOR\_UID = 4>

<FIRST\_NAME>Levi</FIRST\_NAME>

<MIDDLE\_NAME/>

<LAST\_NAME>Bastille</LAST\_NAME>

</AUTHOR>

<AUTHOR AUTHOR\_UID = 5>

<FIRST\_NAME>John</FIRST\_NAME>

<MIDDLE\_NAME/>

<LAST\_NAME>Doe</LAST\_NAME>

</AUTHOR>

<AUTHOR AUTHOR\_UID = 6>

<FIRST\_NAME>Kelly</FIRST\_NAME>

<MIDDLE\_NAME/>

<LAST\_NAME>Jacobs</LAST\_NAME>

</AUTHOR>

</dataset>

-- Query to FLATTEN the XML data

SELECT

FLATTEN\_RAW\_COL:"@AUTHOR\_UID" as AUTHOR\_ID

,XMLGET(FLATTEN\_RAW\_COL, 'FIRST\_NAME'):"$"::STRING as FIRST\_NAME

,XMLGET(FLATTEN\_RAW\_COL, 'MIDDLE\_NAME'):"$"::STRING as MIDDLE\_NAME

,XMLGET(FLATTEN\_RAW\_COL, 'LAST\_NAME'):"$"::STRING as LAST\_NAME

FROM FLATTEN\_RAW\_XML\_TABLE;

-- Sample JSON data

{

"id": 55388352846278,

"inventory\_quantity": 19,

"sku": "sku6"

},

{

"id": 98388391387998,

"inventory\_quantity": 37,

"sku": "sku4"

},

{

"id": 93394420142283,

"inventory\_quantity": 16,

"sku": "sku2"

},

{

"id": 95794426007123,

"inventory\_quantity": 28,

"sku": "sku7"

},

{

"id": 89794429022894,

"inventory\_quantity": 32,

"sku": "sku9"

},

{

"id": 45694431414982,

"inventory\_quantity": 28,

"sku": "sku6"

},

{

"id": 23594455597765,

"inventory\_quantity": 76,

"sku": "sku8"

},

{

"id": 45694459856987,

"inventory\_quantity": 10,

"sku": "sku1"

}

-- Query to FLATTEN the JSON data

SELECT FLATTEN\_RAW\_COL:id::varchar AS ID,

FLATTEN\_RAW\_COL:sku::varchar AS SKU,

FLATTEN\_RAW\_COL:inventory\_quantity AS INVENTORY\_QUANTITY

FROM FLATTEN\_RAW\_JSON\_TABLE,

LATERAL FLATTEN(input => FLATTEN\_RAW\_JSON\_TABLE.FLATTEN\_RAW\_COL);

-- Sample nested JSON data

[{

"book\_title":"The Definitive Guide to Azure Data Engineering",

"year\_published":2021,

"authors": [

{

"first\_name":"Ron",

"middle\_name":null,

"last\_name":"LEsteve"

},

{

"first\_name":"Sam",

"middle\_name":"Smith",

"last\_name":"Broadwhick"

}

]

}

{

"book\_title":"Baby Talks",

"year\_published":2021,

"authors":

[{

"first\_name":"Kathy",

"middle\_name":"L",

"last\_name":"Salisbery"

}

]

}

{

"book\_title":"BBQ Recipes",

"year\_published":2021,

"authors":

[{

"first\_name":"Levi",

"middle\_name":null,

"last\_name":"Bastille"

}

]

}

{

"book\_title":"Game Of Tech",

"year\_published":2020,

"authors":

[{

"first\_name":"John",

"middle\_name":null,

"last\_name":"Doe"

}

]

}

{

"book\_title":"Corgi Dreams",

"year\_published":2021,

"authors":

[{

"first\_name":"Kelly",

"middle\_name":null,

"last\_name":"Jacobs"

}

]

}

-- Query to FLATTEN the nested JSON data

SELECT

value:first\_name::VARCHAR AS FIRST\_NAME,

value:middle\_name::VARCHAR AS MIDDLE\_NAME,

value:last\_name::VARCHAR AS LAST\_NAME,

FLATTEN\_RAW:book\_title::VARCHAR AS BOOK\_TITLE,

FLATTEN\_RAW:year\_published::VARCHAR AS YEAR\_PUBLISHED

FROM FLATTEN\_RAW\_NESTEDJSON\_TABLE

,LATERAL FLATTEN(input => FLATTEN\_RAW:authors);

Here is a sample script which details how to create masking policies. Within the External Tokenization script, creditcard\_decrypt is the external function needed for unprotecting the tokenized data. When creating masking policies, regex, timestamps, UDF, and custom decryption scripts are supported.

-- create a custom masking policy admin role

create role masking\_policy\_admin;

-- grant privileges to masking\_policy\_admin role.

grant create masking policy on schema <schema\_name> to role masking\_policy\_admin;

grant apply masking policy on account to role masking\_policy\_admin;

-- Dynamic Data Masking

create masking policy mask\_creditcard as (val string) returns string ->

case

when current\_role() in ('FINANCE') then val

else '\*\*\*\*\*\*'

end;

-- External Tokenization

create masking policy detokenize\_creditcard as (val string) returns string ->

case

when current\_role() in ('FINANCE') then decrypt\_creditcard(val)

else val -- sees tokenized data

end;

-- Apply Masking Policy to Column in Table

create or replace table tbl\_customer\_data (creditcard string masking policy mask\_creditcard);

-- Apply Masking Policy to Column in View

create or replace view vw\_customer\_data (creditcard string masking policy mask\_creditcard) as select \* from tbl\_customer\_data;

Here is a sample row access policy to allow Regional Managers the ability to view only their region’s sales based on a mapping table called region\_manager\_mapping. If the manager’s current role is in the mapping table, they will only be able to see data for their respective region. Additionally, the leadership\_role is a custom role that would supersede the managers role in terms or privilege, therefore this role would not be bound to a particular region and would have full access to all data for their subordinate Regional Managers.

-- Create manager and leadership roles and grant access to sales table

create role manager\_role;

create role leadership\_role;

grant select on table sales to role manager\_role;

grant select on table sales to role leadership\_role;

-- Create row access policy

create or replace row access policy security.sales\_policy as (sales\_region varchar) returns boolean ->

'leadership\_role' = current\_role()

or exists (

select 1 from region\_manager\_mapping

where manager = current\_role()

and region = region

)

;

-- Applying newly create row access policy to existing sales table

alter table sales add row access policy security.sales\_policy on (region);

Here are some sample queries that can be run using the access\_history view.

-- Query to obtain user access history

select user\_name

, query\_id

, query\_start\_time

, direct\_objects\_accessed

, base\_objects\_accessed

from access\_history

-- Query to determine who accessed a sensitive table(object\_id) in the last 30 days

select distinct user\_name

from access\_history

, lateral flatten(base\_objects\_accessed) f1

where f1.value:"objectId"::int=<fill\_in\_object\_id>

and f1.value:"objectDomain"::string='Table'

and query\_start\_time >= dateadd('day', -30, current\_timestamp())

;

A tag is a schema level object that can be associated with another Snowflake object. Tags are stored as key-value pairs (e.g. cost\_center = ‘data’). The script below shows how to create custom tags.

-- create a custom tag\_admin role

use role useradmin;

create role tag\_admin;

use role accountadmin;

grant create tag on schema <schema\_name> to role tag\_admin;

grant apply tag on account to role tag\_admin;

-- create a new tag

use role tag\_admin;

use schema my\_db.my\_schema;

create tag cost\_center;

-- create a tag on a new warehouse

use role tag\_admin;

create warehouse mywarehouse with tag cost\_center = 'data';

-- create a tag on an existing table’s column

alter table <table\_name> modify column <column\_name> set tag <tag\_key> = '<tag\_value>' [ , <tag\_key> = ’<tag\_value>’ , ... ];

alter table <table\_name> modify column <column\_name> unset <tag\_key> [ , <tag\_key> , ... ];

-- view all available tags

select \* from snowflake.account\_usage.tags

order by tag\_name;

Here is a sample script which demonstrates just how easy it is to create a share, grant usage to database objects, and then allow specified accounts access to the share. Shares can be even more complex to allow sharing of multiple databases across regions and cloud providers. The last script in this section shows how to create a Dynamic Secure View that can then be shared via account-to-account or reader account access. This method enforces row-level security by only allowing consumers access to data that they are permitted to view based on their current account. A well-defined mapping table and appropriate key across tables will be needed for this approach to be set up. Once successfully set up, it allows for seamless data sharing across vendors and clients.

-- create a Share on a database and schema

create share share\_data;

grant usage on database db\_data to share share\_data;

grant usage on schema db\_data.dim to share share\_data;

grant select on table db\_data.dim.customers to share share\_data;

-- show grants to the Share

show grants to share share\_data;

-- add accounts to the Share

alter share share\_data add accounts=Account1, Account2;

-- create a Dynamic Secure View

CREATE or replace SECURE VIEW SHARED\_VIEW AS

SELECT

NAME,

SALES,

PRODUCT,

ID

FROM FACT\_SALES fs

JOIN DIM\_CUSTOMER dc

ON fs.ID = dc.ID

AND UPPER(SNOWFLAKE\_ACCNT) = CURRENT\_ACCOUNT();

Here is the script that will need to be run in Snowflake to create the Security Integration. Note that only the ACCOUNTADMIN or a role with global CREATE INTEGRATION permissions will be allowed to run this script.

The <AZURE\_AD\_ISSUER> value for the external\_oauth\_issuer parameter includes the TENANT\_ID (eg: <https://sts.windows.net/TENANT_ID/>) and can be found in the About section in your Power BI tenant. Once this script executes successfully, be sure to test connectivity from Power BI to Snowflake using SSO to ensure that you can authenticate and connect successfully.

-- create Power Bi Security Integration

create security integration powerbi

type = external\_oauth

enabled = true

external\_oauth\_type = azure

external\_oauth\_issuer = '<AZURE\_AD\_ISSUER>'

external\_oauth\_jws\_keys\_url = 'https://login.windows.net/common/discovery/keys'

external\_oauth\_audience\_list = ('https://analysis.windows.net/powerbi/connector/Snowflake')

external\_oauth\_token\_user\_mapping\_claim = 'upn'

external\_oauth\_snowflake\_user\_mapping\_attribute = 'login\_name'

external\_oauth\_any\_role\_mode = 'ENABLE';

The following code can be used to generate this manifest file with a SQL Query within the Databricks workspace.

GENERATE snowflake\_delta\_manifest FOR TABLE DeltaDimCustomer

Create a stage using code similar to the following from Snowflake. A Snowflake Stage is used to reference inside and outside of Snowflake. For external Stages, Snowflake supports ADLSgen2, AWS S3, and GCP buckets.

create or replace stage delta\_stage\_table url='<path-to-delta-table>'

Once this Stage is created, you can create an external table using code as follows.

CREATE OR REPLACE EXTERNAL TABLE delta\_DimCustomer(

filename VARCHAR AS split\_part(VALUE:c1, '/', -1)

)

WITH LOCATION = @delta\_stage\_table/snowflake\_delta\_manifest/

FILE\_FORMAT = (TYPE = CSV)

PATTERN = '.\*[/]manifest'

AUTO\_REFRESH = true;

For more information on Snowflake to Delta Lake integration, please see the following Snowflake article: [https://docs.databricks.com/delta/snowflake-integration.html](about:blank)

The samples shown in the following GitHub Repository demonstrates that models can be pushed inside Snowflake UDFs to run the model as part of SQL: <https://github.com/avadhoot-agasti/snowflake-demos/blob/master/ml_classifier_lending_club.ipynb>

For more details on getting started with Snowflake, check out the robust ‘Getting Started’ documentation from Snowflake which can be found here: https://docs.snowflake.com/en/user-guide-getting-started.html