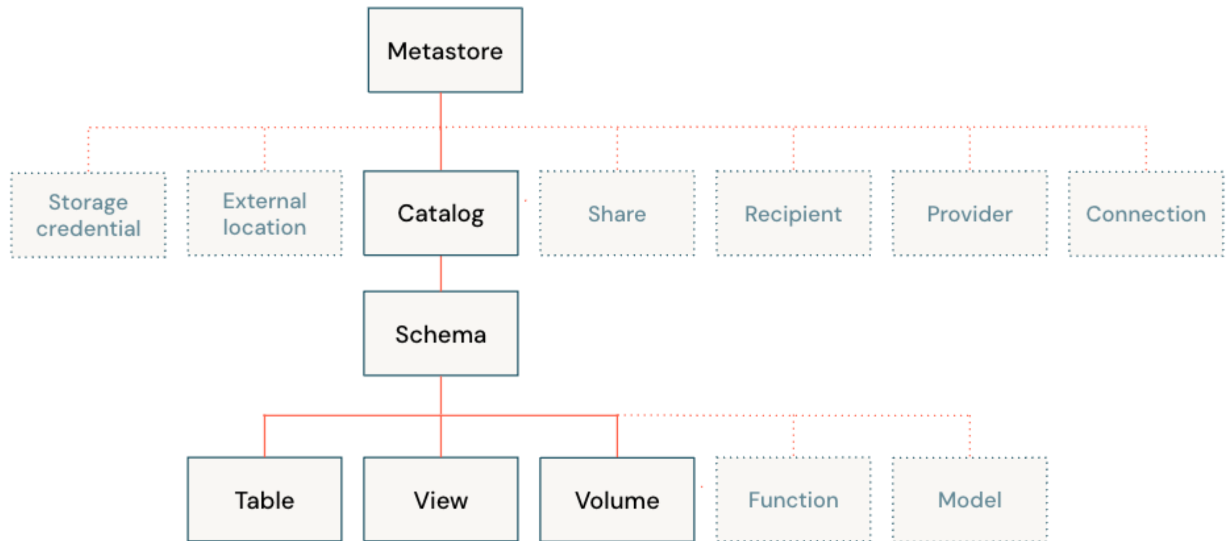
Best practices for ingestion partners using Unity Catalog volumes

Context

The goal of this doc is to outline how Databricks ingestion partners can bring data into Databricks using Unity Catalog volumes as staging locations for data.

Glossary

- ISV
    - Independent Software Vendors (aka ISVs or technology partners) are software companies who have business and technical relationships with Databricks.
- Partner Connect
    - Partner Connect lets you create accounts with select Databricks technology partners and connect your Databricks workspace to partner solutions from the Databricks UI.
- User
    - Part of the Databricks identity management model. User identities recognized by Databricks and represented by email addresses.
- Service principal
    - Part of the Databricks identity management model. Identities for use with jobs, automated tools, and systems such as scripts, apps, and CI/CD platforms.
- Unity Catalog
    - Unity Catalog provides centralized access control, auditing, lineage, and data discovery capabilities across Databricks workspaces.
- Unity Catalog object model
    - The hierarchy of primary data objects in Unity Catalog. See graph below:



- Catalog
  - Part of the Unity Catalog object model. The first layer of the object hierarchy, used to organize your data assets (for example schemas, tables, and volumes)
- Schema
  - Part of the Unity Catalog object model. Also known as databases, schemas are the second layer of the object hierarchy and contain tables and views.
- Table
  - Part of the Unity Catalog object model. A table resides in the third layer of Unity Catalog's three-level namespace. It contains rows of data.

What are Volumes

Volumes are Unity Catalog objects representing a logical volume of storage in a cloud object storage location, with governance based on unity catalog.

Volumes provide capabilities for accessing, storing, governing, and organizing files. While tables provide governance over tabular datasets, volumes add governance over non-tabular datasets. You can use volumes to store and access files in any format, including structured, semi-structured, and unstructured data.

From an administrative perspective volumes are similar to tables: they both live in schemas inside a catalog.

For more information, please refer to volume public documentation.

High level steps

For Independent Software Vendors (ISVs) building a partner integration, below are the recommended high-level steps to use volumes to build an integration with Databricks to support ingest/transform use cases.

1. **Partners** to ***document*** for their customers using ISV integration: permissions required to create volumes. Refer to the section on prerequisites in “How to create Volumes”.
2. **Partners’ customers** ***create a new catalog*** and ***grant permission*** to **Partners**.
3. **Partners** use APIs (SQL) to ***create a schema*** in the provided catalog.
4. **Partners** use APIs (SQL) to ***create a volume***.
    a. Databricks recommends creating one volume per destination schema.
5. **Partners** ***write files*** to the volume using APIs (SQL).
    a. For recommended file size, see “File sizing best practices”.
    b. For file format, Databricks recommends that partners use Parquet files when uploading to volumes.
    c. For directories inside the volume, Databricks recommends that partners use table specific prefixes to separate out the workflows between different tables.
6. **Partners** use ***COPY INTO*** to load data into Delta Lake.
    a. If there are transient errors (e.g., network failures), you can re-run COPY INTO. COPY INTO is a retriable and idempotent operation—files in the volume that have already been loaded are skipped.
7. **Partners** **delete files** in the volumes once the COPY INTO command finishes successfully.

See the following sections of this doc to learn more about the details of each step.

How to create Volumes

On a high level, the volume creation process requires following steps:

1. **Partners’ customers** follow the “**Prerequisite**” section below and provide the following 3 items to the **partners.**
    a. Personal access token (PAT)
    b. Catalog
    c. Compute resource
2. **Partners** follow the “Create Schemas via SQL” section to create a schema.
3. **Partners** follow the “Different approaches to create Volumes” section to create a volume.

Prerequisites

Partner's customer should collect the following information and send it to the partner in a secure manner (e.g., Partner can provide an App where data transferred from customer is encrypted).

Prerequisite 1: Personal Access Token (PAT)

The steps described in this section are expected to be completed by partners' customers. Alternatively, if partners integrate with Partner Connect, Partner Connect will automate the steps in this section.

Create a Personal Access Token (PAT) for a service principal.

- To provision a Databricks service principal, Databricks recommends that you create a Databricks service principal in your Databricks account. Then you add that Databricks service principal to your target Databricks workspace and give the Databricks service principal workspace permissions. Finally, you generate a Databricks personal access token, which you can use to represent the Databricks service principal programmatically in calls from tools, scripts, and SDKs. Follow this quickstart to complete these steps.

Prerequisite 2: Create a Catalog

The steps described in this section are expected to be completed by partners' customers. Alternatively, if partners integrate with Partner Connect, Partner Connect will automate the steps in this section.

Create a catalog to organize your data assets (for example schemas, tables, and volumes).

It is NOT expected that partners will create the catalog when building the integration. Partners should either:

- Ask customers to create the destination catalog, or
- Use Partner Connect which will set up the catalog.

This section is added so that partners can show customers how to create a destination catalog.

Requirements

- You must be a Databricks metastore admin or have been granted the CREATE CATALOG privilege on the metastore
- Your Databricks account must be on the Premium plan or above.
- You must have a Unity Catalog metastore linked to the workspace where you perform the catalog creation.
- The compute resource that you use to run the notebook or Databricks SQL to create the catalog must be using a Unity Catalog compliant access mode. Right now the supported modes are "Assigned" or "Shared". More details can be found at: What is cluster access mode?

Create a catalog using the Databricks UI

1. Log in to a workspace that is linked to the metastore.
2. Click **Data**.
3. Click the **Create Catalog** button.
4. (Optional) Specify the location where data for managed tables in the catalog will be stored. Specify a location here only if you do not want managed tables in this catalog to be stored in the default root storage location that was configured for the metastore. See Create a Unity Catalog metastore.
   The path that you specify must be defined in an external location configuration, and you must have the CREATE MANAGED STORAGE privilege on that external location. You can also use a subpath of that path. See Manage external locations and storage credentials.
5. Click **Create**.
6. (Optional) Specify the workspace that the catalog is bound to.
   By default, the catalog is shared with all workspaces attached to the current metastore. If the catalog contains data that should be restricted to specific workspaces, go to the Workspaces tab and add those workspaces.
   For more information, see (Optional) Assign a catalog to specific workspaces.
7. **Grant USE CATALOG** and **CREATE SCHEMA** permission to partner's service principal. See Unity Catalog privileges and securable objects for more information.
   - To assign the permissions in SQL, you can run:
     - ```
       GRANT USE CATALOG ON CATALOG test_catalog TO
       partner_service_principal;
       ```
       ```

            - E.g, `GRANT USE CATALOG ON CATALOG test_catalog TO `partner_job_bot`;`
        - `GRANT CREATE SCHEMA ON CATALOG test_catalog TO partner_service_principal;`
            - E.g, `GRANT CREATE SCHEMA ON CATALOG test_catalog TO `partner_job_bot`;`

For more information on how to create catalog (e.g., requirements, using programming languages such as SQL/Python/R/Scala), see create catalogs.

Prerequisite 3: Compute Resource

The steps described in this section are expected to be completed by partners' customers. Alternatively, if partners integrate with Partner Connect, Partner Connect will automate the steps in this section.

Choose the SQL warehouse (recommended) or cluster that you want to use.

[Recommended] For SQL warehouse:

- The server hostname of the SQL warehouse. You can get this from the Server Hostname value in the Connection Details tab for your SQL warehouse.
- The HTTP path of the SQL warehouse. You can get this from the HTTP Path value in the Connection Details tab for your SQL warehouse.

For Cluster:

- The server hostname of the cluster. You can get this from the Server Hostname value in the Advanced Options > JDBC/ODBC tab for your cluster.
- The HTTP path of the cluster. You can get this from the HTTP Path value in the Advanced Options > JDBC/ODBC tab for your cluster.

Create Schemas via SQL

The steps described in this section are expected to be completed by partners.

A schema (also known as databases) resides in a catalog. You need a schema to organize tables and volumes.

Requirements

- You must have the USE CATALOG and CREATE SCHEMA data permissions on the schema's parent catalog. Either a metastore admin or the owner of the catalog can grant you these privileges. If you are a metastore admin, you can grant these privileges to yourself.
  All users have the USE CATALOG permission on the main catalog by default.
- Your Databricks account must be on the Premium plan or above.
- You must have a Unity Catalog metastore linked to the workspace where you perform the schema creation.
- The compute resource that you use to run the notebook or Databricks SQL to create the catalog must be using a Unity Catalog compliant access mode. Right now the supported modes are "Assigned" or "Shared". More details can be found at: What is cluster access mode?

Steps

1. Use one of the following approaches to configure the SQL connection:
   - Configure the Databricks SQL Connector for Python.
   - Configure the Databricks JDBC driver.
   - Configure the Databricks ODBC driver.
2. Run the following SQL commands. Items in brackets are optional. Replace the placeholder values:
   a. <catalog-name>: The name of the parent catalog for the schema.
   b. <schema-name>: A name for the schema.
   c. <comment>: Optional description or other comment.
   d. <property-key> = <property-value> [, ...]: Optional. Spark SQL properties and values to set for the schema.

For parameter descriptions, see CREATE SCHEMA.

```
USE CATALOG <catalog>;
CREATE { DATABASE | SCHEMA } [ IF NOT EXISTS ] <schema-name>
    [ COMMENT <comment> ]
    [ WITH DBPROPERTIES ( <property-key = property_value [ , ... ]> ) ];
```

Example:

```
USE CATALOG test_catalog;
CREATE SCHEMA IF NOT EXISTS test_schema
```

You can optionally omit the USE CATALOG statement and replace <schema-name> with <catalog-name>.<schema-name>.

Example:
`CREATE SCHEMA IF NOT EXISTS test_catalog.test_schema`

Create volumes via SQL

The steps described in this section are expected to be carried out by partners.

It is recommended that partners use one volume per schema.

Steps

1. Use one of the following approaches to configure the SQL connection:
   - Configure the Databricks SQL Connector for Python.
   - Configure the Databricks JDBC driver.
   - Configure the Databricks ODBC driver.
2. Use one of the approaches listed above, run the following command to create the volume.
   - CREATE VOLUME <catalog_name>.<schema_name>.<volume_name>;
   - Example: CREATE VOLUME test_catalog.test_schema.test_volume;

How to use Volumes

The steps described in this section are expected to be completed by partners.

For instructions on how to create a volume, please see the above "How to create Volumes" section.

Step 1: Upload files to Volume

Use the provided APIs to upload files to the volume. We support uploading via SQL.
1. Note: SQL API support is available on DBR versions 13.2 and above.

Example of using the SQL API

Here is a simple example of how to use the SQL API to upload files. You need to set up the connection following Managing Volumes with SQL API.

```
PUT 'home/bob/data.csv' INTO
'/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/file1.csv' OVERWRITE
```

File format recommendations

Supported source formats include CSV, JSON, Avro, ORC, Parquet, text, and binary files.
Databricks recommends that partners upload files in the format of Parquet.

Path setup recommendations

The path management within the volume is entirely up to the partner. ***Databricks recommends that partners use table specific prefixes to separate out the workflows between different tables.***

E.g., you can set up the full volume path like this:
"/Volumes/<catalog_name>/<schema_name>/<volume_name>/<table_name_tmp>/"

Replace the placeholder values:

- <catalog-name>: The name of the catalog.
- <schema-name>: The name of the schema.
- <volume-name>: The name of the volume.
- <table_name_tmp>: The name of the table. You can add a "_tmp" suffix to indicate this is used only during the file uploading process.

For example, suppose you have 2 upload sessions. In session 1, you uploaded files: `expense_file_1` and `revenue_file_1` for tables: "test_table_expense" and "test_table_revenue". In session 2, you added `expense_file_2` for "test_table_expense". Then the paths could look like following:

- "/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/expense_file_1"
- "/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/revenue_file_1"
- "/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/expense_file_2"

File sizing best practices

Databricks recommends that the file size should be roughly 128MB.

- For file formats such as CSV/JSON/Text/BINARY FILE, this is the data size uncompressed.
- For file formats such as Parquet/Avro/ORC, this is the data size compressed.

Step 2: Create a target table in Delta

COPY INTO must target an existing Delta table.

Syntax

Run the following command to create the table.

```
CREATE TABLE IF NOT EXISTS catalog_name.schema_name.table_name
[(col_1 col_1_type, col_2 col_2_type, ...)]
[COMMENT <table-description>]
[TBLPROPERTIES (<table-properties>)];
```

Replace the placeholder values:

- <catalog-name>: The name of the catalog.
- <schema-name>: The name of the schema.
- <table-name>: The name of the table.
- <table-description>: Optional comment describing the table.
- <table-properties>: Optionally sets one or more user defined properties.

Databricks recommends that partners define the list of columns, their types, properties, descriptions, and column constraints.

Example: Create a Delta table with Schema

You can also set the schema explicitly:

```
CREATE TABLE test_catalog.test_schema.test_table_with_schema (
  loan_id BIGINT,
  funded_amnt INT,
  paid_amnt DOUBLE,
  addr_state STRING
);
```

Step 3: Use the COPY INTO command to move files from Volumes into Delta Lake.

COPY INTO provides following features when ingesting data:

1. COPY INTO is a re-triable and idempotent operation. Files in the source location that have already been loaded are skipped.
2. COPY INTO supports concurrent invocations against the same table. As long as COPY INTO is invoked concurrently on distinct sets of input files, each invocation should eventually succeed. More details can be found here.

Syntax

```
COPY INTO target_table
  FROM { source_clause |
         ( SELECT expression_list FROM source_clause ) }
  FILEFORMAT = data_source
  [ VALIDATE [ ALL | num_rows ROWS ] ]
  [ FILES = ( file_name [, ...] ) | PATTERN = glob_pattern ]
  [ FORMAT_OPTIONS ( { data_source_reader_option = value } [, ...] ) ]
  [ COPY_OPTIONS ( { copy_option = value } [, ...] ) ]

source_clause
  source [ WITH ( [ CREDENTIAL { credential_name |
                                 (temporary_credential_options) } ]
                  [ ENCRYPTION (encryption_options) ] ) ]
```

Parameters

- target_table :
  - An existing Delta table
- source:
  - The file location to load the data from. Files in this location must have the format specified in FILEFORMAT. The location is provided in the form of a URI.
  - You don't need to provide credentials if the path is already defined as a volume path that you have permissions to use.
  - Accepted encryption options are:
    - TYPE = 'AWS_SSE_C', and MASTER_KEY for AWS S3
- SELECT expression_list:
  - Selects the specified columns or expressions from the source data before copying into the Delta table. The expressions can be anything you use with SELECT statements, including window operations. You can use aggregation expressions only for global aggregates–you cannot GROUP BY on columns with this syntax.
- FILEFORMAT = data_source:
  - The format of the source files to load. One of CSV, JSON, AVRO, ORC, PARQUET, TEXT, BINARYFILE.
- VALIDATE
  - The data that is to be loaded into a table is validated but not written to the table. These validations include:
    - Whether the data can be parsed.

        - Whether the schema matches that of the table or if the schema needs to be evolved.
        - Whether all nullability and check constraints are met.
    - The default is to validate all of the data that is to be loaded. You can provide a number of rows to be validated with the ROWS keyword, such as VALIDATE 15 ROWS. The COPY INTO statement returns a preview of the data of 50 rows or less, when a number of less than 50 is used with the ROWS keyword.
- FILES
  A list of file names to load, with length up to 1000. Cannot be specified with PATTERN.
- PATTERN
  A glob pattern that identifies the files to load from the source directory. Cannot be specified with FILES.
- FORMAT_OPTIONS
  Options to be passed to the Apache Spark data source reader for the specified format. See Format options for each file format.
- COPY_OPTIONS
  Options to control the operation of the COPY INTO command.
    - force: boolean, default false. If set to true, idempotency is disabled and files are loaded regardless of whether they've been loaded before.
    - mergeSchema: boolean, default false. If set to true, the schema can be evolved according to the incoming data.

More details on the parameters can be found in the COPY INTO docs.

Example: Load data with schema defined

This example assumes that you've created a table with schema following "Example: Create a Delta table with Schema".

Below is an example of loading a single Parquet file:

```
COPY INTO test_catalog.test_schema.test_table_with_schema
FROM '/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/file1.parquet'
FILEFORMAT = PARQUET;
```

Below is an example of loading multiple parquet files from one folder:

```
COPY INTO test_catalog.test_schema.test_table_with_schema
FROM '/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/'
FILEFORMAT = PARQUET;
```

COPY INTO and schema evolution

For advanced users who'd like to use schema evolution, please follow the section "Appendix: COPY INTO and schema evolution".

Step 4: Clean up files in volume

Databricks recommends that partners delete files in the staging directory once the data is ingested into Delta Lake. Partners can use the [SQL REMOVE API] to delete files from the staging location. You can find detailed information about APIs in the Appendix.

The delete operation has to be done on a file level. You can NOT delete a folder and its contents recursively.

E.g., you can use SQL API to remove the file

```
REMOVE '/Volumes/<catalog_name>/<schema_name>/<volume_name>/<table_name_tmp>/<file_name>'
```

To find all the files in a folder use LIST command:

```
LIST '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/'
```

Suppose you have 2 files in the "test_table_revenue_tmp" folder. You can delete the files by running following command:

```
REMOVE '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/file1.csv'
REMOVE '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/file2.csv'
```

Best practices on failure recovery

File upload failures

If you encountered failures while uploading files, it is safe to retry the upload operations.

While retrying, you want to make sure you do not accidentally put duplicated copies of data in volume. This might lead to duplicated data in the target table.

Here an example where you can have ***duplicated data*** in the target table:

1. You received a timeout error when uploading `file1.csv` to '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/**file1.parquet**.
2. Since the first attempt failed, you issued a 2nd attempt, however, you uploaded to a different path:'/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/**file1_retry.parquet**'.
3. It turned out that the 1st upload actually succeeded on volume, but the response was lost during network transmission.
4. Now you issue the COPY INTO command on the folder: '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/`. This will ingest both **`file1.parquet`** and **`file1_retry.parquet`** and you will have duplicated data.

A simple fix is to always use the same file path when retrying the file upload.

COPY INTO failures

COPY INTO is a re-triable and idempotent operation. Files in the source location that have already been loaded are skipped. This means when you see COPY INTO failures, you can safely retry it without worrying about duplicating data.

Best practices on uploading multiple files to one target table

[Recommended] Coordinator and worker architecture

If you have multiple files that you'd like to ingest to a single target table, Databricks recommends following a coordinator and worker architecture. You can have 1 coordinator and many workers.

Here's a high level summary of how to do it:

1. Each worker node is responsible for writing files to volume. All the worker nodes should write to the same parent folder for a target table. E.g.
    a. Worker A writes to '/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/file1.parquet
    b. Worker B writes to '/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/file2.parquet/

2. After all the worker nodes are done uploading, the coordinator can issue a COPY INTO command and load all the files in the folder. You only need 1 COPY INTO command for the table.
    a. E.g,

```
COPY INTO test_catalog.test_schema.test_table_with_schema
FROM '/Volumes/test_catalog/test_schema/test_volume/test_table_expense_tmp/'
FILEFORMAT = PARQUET;
```

3. After the COPY INTO is done, the coordinator can list all the files in the folder and delete them.

Run COPY INTO concurrently

If you can NOT follow the "[Recommended] Coordinator and worker architecture", you can run COPY INTO concurrently against the same table. As long as COPY INTO is invoked concurrently on distinct sets of input files, each invocation should eventually succeed. More details can be found here.

Running too many concurrent COPY INTO commands can cause transaction conflicts and eventually the COPY INTO command to fail. While there isn't a one size fits all recommendation for how many COPY INTO calls can be made concurrently, you can try adding parallelism while latencies are consistent and you don't get a COPY_INTO_ROCKSDB_MAX_RETRY_EXCEEDED error.

Note this will be less performant than the recommended approach as each COPY INTO invocations involves overhead.

FAQ

Question: How many volumes can be created per schema?
Answer: You can create up to 10,000 volumes per schema, but it is recommended that you create 1 volume per schema.

Question: What's the ideal file size?
Answer: Please check out the "File sizing best practices".

Question: Instead of having **partners** creating a new schema, can **partners** reuse an existing schema which is provided by the **customer**?
Answer: Yes, **partners** can reuse an existing schema provided by the **customer**. The set up steps in this case would be slightly different from creating a new schema.

Here're the high level steps if you choose to have **partners** creating a **new** schema:

1. **Customers** create a catalog.
2. **Customers** grant **USE CATALOG** and **CREATE SCHEMA** to the **partner's** service principal on the catalog.
3. **Customers** send the **catalog** information to the **partners**.

Mode details can be found at "How to Create Volumes::Prerequisite 2: Create a Catalog".

Here're the high level steps if you choose to have **partners** reuse an **existing** schema:

1. **Customers** create a catalog.
2. **Customers** grant **USE CATALOG** to the **partner's** service principal on the catalog.
    a. To assign the permissions in SQL, you can run:
        i. `GRANT USE CATALOG ON CATALOG test_catalog TO partner_service_principal;`
            1. `E.g, GRANT USE CATALOG ON CATALOG test_catalog TO `partner_job_bot`;`
3. **Customers** create a schema.
    a. For more information on how to create schemas, please checkout the documentation on create schema.
4. **Customers** grant **USE SCHEMA, CREATE TABLE, CREATE VOLUME** to the **partner's** service principal on the schema.
    a. To assign the permissions in SQL, you can run:
        i. `GRANT USE SCHEMA, CREATE TABLE, CREATE VOLUME ON SCHEMA test_schema TO partner_service_principal;`
            1. `E.g, GRANT USE CATALOG ON CATALOG test_catalog TO `partner_job_bot`;`
    b. See Unity Catalog privileges and securable objects for more information.
5. **Customers** send the **catalog and schema** information to the **partners**.

Appendix

Managing Volumes with SQL API

We provide SQL APIs below to enable our partners to put/get/remove items into the volume for staging purposes.

Prerequisites

To use the SQL API, please follow these steps:

1. If you are using a cluster, make sure the cluster has DBR version 13.2 or above.
2. Follow [Databricks SQL Connector for Python]() to get started. Specifically you need to:
    a. Gather the information for the cluster or SQL warehouse that you want to use.
    b. Install the Databricks SQL Connector for Python library on your development machine by running: pip install databricks-sql-connector.
3. When initializing the connection by calling sql.connect(), set `staging_allowed_local_path`. For security, local files must be contained within, or descended from, staging_allowed_local_path of the connection.

After the above steps, you should be able to use the SQL API to ingest files into volumes and also load the files into delta via COPY INTO. You can find [an example script here.]()

SQL Syntax

```
PUT <localfile> INTO '/Volumes/<catalog_name>/<schema_name>/<volume_name>/<path>' [OVERWRITE]

GET '/Volumes/<catalog_name>/<schema_name>/<volume_name>/<path>' TO <localfile>

REMOVE '/Volumes/<catalog_name>/<schema_name>/<volume_name>/<path>'

LIST '/Volumes/<catalog>/<schema>/<volume>/<path>/'
```

SQL Example

```
PUT '/home/bob/data.csv' INTO
'/Volumes/test_catalog/test_schema/test_volume/2023/06/file1.csv' OVERWRITE

GET '/Volumes/test_catalog/test_schema/test_volume/2023/06/file1.csv' TO '/home/bob/data.csv'

REMOVE '/Volumes/test_catalog/test_schema/test_volume/2023/06/file1.csv'

LIST '/Volumes/catalog_demo/schema_demo/volume_demo/2023/06/'
```

Example

Below is an example Python script that demonstrates how to use ingestion commands to PUT/GET/REMOVE files and how to use COPY INTO to load the file into Delta.

See comment in the file on how to run it.

```
from databricks import sql
import os
```

```
"""
Databricks supports data ingestion of local files via a cloud staging location.
Ingestion commands will work on DBR >13.2 And you must include a staging_allowed_local_path kwarg when
calling sql.connect().

Use databricks-sql-connector to PUT files into the staging location where Databricks can access them:

    put '/path/to/local/data.csv' into
'/volumes/test_catalog/test_schema/test_volume/examples/sales.csv' [overwrite]

and you can delete with a REMOVE command:

    REMOVE '/Volumes/test_catalog/test_schema/test_volume/examples/sales.csv'

Ingestion queries are passed to cursor.execute() like any other query. For PUT commands, a local file
will be read. For security, this local file must be contained within, or descended from, a
staging_allowed_local_path of the connection.

After file is written to staging location, you can issue COPY INTO commands to loads the file into a
delta table:
    COPY INTO my_table
    FROM '/path/to/files'
    FILEFORMAT = <format>
    FORMAT_OPTIONS ('inferSchema' = 'true')
    COPY_OPTIONS ('mergeSchema' = 'true');

To run this script:

1. Set the environment variables for secrets:

export DATABRICKS_HOST=********.databricks.com
export DATABRICKS_HTTP_PATH=/sql/1.0/endpoints/****************
export DATABRICKS_TOKEN=dapi********************************

2. Set the CATALOG, SCHEMA, VOLUME and TABLE constants to your catalog, schema, volume and table.
3. Set the FILEPATH constant to the path of a file that will be uploaded (this example assumes its a CSV
file)
4. Run this file

"""

CATALOG = "test_catalog"
SCHEMA = "test_schema"
TABLE = "test_table"
VOLUME = "test_volume"

FILEPATH = "example.csv"

# FILEPATH can be relative to the current directory.
# Resolve it into an absolute path
_complete_path = os.path.realpath(FILEPATH)

if not os.path.exists(_complete_path):

    # It's easiest to save a file in the same directory as this script. But any path to a file will
work.
    raise Exception(
        "You need to set FILEPATH in this script to a file that actually exists."
    )

# Set staging_allowed_local_path equal to the directory that contains FILEPATH
staging_allowed_local_path = os.path.split(_complete_path)[0]
```

```
host = os.getenv("DATABRICKS_HOST")
http_path = os.getenv("DATABRICKS_HTTP_PATH")
access_token = os.getenv("DATABRICKS_TOKEN")

connection = sql.connect(
  server_hostname=host,
  http_path=http_path,
  access_token=access_token,
  staging_allowed_local_path=staging_allowed_local_path,
)

cursor = connection.cursor()

# Ingestion commands are executed like any other SQL.
# Here's a sample PUT query. You can remove OVERWRITE at the end to avoid silently overwriting data.
query = f"PUT '{_complete_path}' INTO '/Volumes/{CATALOG}/{SCHEMA}/{VOLUME}/pysql_examples/demo.csv'
OVERWRITE"

print(f"Uploading {FILEPATH} to staging location")
cursor.execute(query)
print("Upload was successful")

# use COPY INTO to create a delta table from the uploaded file
# first, create the target table
print(f"Create table: {CATALOG}.{SCHEMA}.{TABLE} so we can run COPY INTO")
create_table_query = f"CREATE TABLE IF NOT EXISTS {CATALOG}.{SCHEMA}.{TABLE};"
cursor.execute(create_table_query)
print("Create table was successful")

copy_into_command = f"""
COPY INTO  {CATALOG}.{SCHEMA}.{TABLE}
FROM '/Volumes/{CATALOG}/{SCHEMA}/{VOLUME}/pysql_examples/demo.csv'
FILEFORMAT = CSV
FORMAT_OPTIONS ('mergeSchema' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');
"""

print("Load data into delta using COPY INTO")
# copy into
cursor.execute(copy_into_command)
print("COPY INTO was successful")

# Here's a sample REMOVE query. It cleans up the the demo.csv created in our first query
query = f"REMOVE '/Volumes/{CATALOG}/{SCHEMA}/{VOLUME}/pysql_examples/demo.csv'"

print("Removing demo.csv from staging location")
cursor.execute(query)
print("Remove was successful")

# clean up
cursor.close()
connection.close()
```

COPY INTO and schema evolution

Example: Create a Schemaless Delta table

If your data format supports schema evolution, you can create a schemaless delta table, and then let COPY INTO infer the schema for you.

```
CREATE TABLE IF NOT EXISTS test_catalog.test_schema.test_table_schemaless;
```

Example: Load data with schema inference

This example assumes that you've created a *schemaless* table following "Example: Create a Schemaless Delta table".

Below is an example of loading a CSV file with headers.

```
COPY INTO test_catalog.test_schema.test_table_schemaless
FROM '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/file1.csv'
FILEFORMAT = CSV
FORMAT_OPTIONS ('mergeSchema' = 'true', 'header' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');
```

Below is an example of loading multiple CSV files with headers. The CSV files are located in:
`'/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/'`

```
COPY INTO test_catalog.test_schema.test_table_schemaless_from_multiple_files
FROM '/Volumes/test_catalog/test_schema/test_volume/test_table_revenue_tmp/'
FILEFORMAT = CSV
FORMAT_OPTIONS ('mergeSchema' = 'true', 'header' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');
```

Migrating from PSL to Volumes

If you previously used Personal Staging Locations (PSLs), please follow the instructions in this doc and migrate to volumes.

Main differences between PSLs and volumes from partner integration's view:

- For volumes, you'll need to create them first before being able to write data to it. Please refer to the "How to create Volumes" section on how to do this.
- The file path in volume is different from the PSL. The volume file path will look like this:
    - "/Volumes/<catalog_name>/<schema_name>/<volume_name>/<path>