databricks

Small summary.

Delta Lake Operations in Databricks

Delta Lake provides ACID transactions and scalable metadata handling for big data. Let's go over common operations on Delta tables, including creating, inserting, selecting, updating, deleting, and merging.

---

1. Create a Delta Table

You can create a Delta table using `CREATE TABLE` in Databricks.

```
-- Create a Delta table from a DataFrame or query
CREATE TABLE delta_table_name (
    id INT,
    name STRING,
    age INT
);
```

- This creates a **Delta table** named `delta_table_name` with `id`, `name`, and `age` columns.
- The `USING DELTA` clause specifies that it is a Delta Lake table.

---

2. Insert into Delta Table

You can use `INSERT INTO` to add data into an existing Delta table.

```
-- Insert values into the Delta table
INSERT INTO delta_table_name (id, name, age)
VALUES (1, 'Alice', 30),
       (2, 'Bob', 25),
       (3, 'Charlie', 35);
```

- This inserts multiple rows into the `delta_table_name` Delta table.

3. Select from Delta Table

To query data from a Delta table, use the `SELECT` statement.

```
-- Select all records from the Delta table
SELECT * FROM delta_table_name;

-- Select specific columns from the Delta table
SELECT id, name FROM delta_table_name WHERE age > 30;
```

4. Update Delta Table

You can update records in a Delta table using the `UPDATE` statement.

```
-- Update records in the Delta table
UPDATE delta_table_name
SET age = 40
WHERE name = 'Charlie';
```

- This updates the `age` to 40 for the record where the name is `Charlie`.

5. Delete from Delta Table

You can remove records from a Delta table using the `DELETE` statement.

```
-- Delete a record from the Delta table
DELETE FROM delta_table_name
WHERE name = 'Bob';
```

- This deletes the record where the name is `Bob`.

6. Drop Delta Table

You can drop a Delta table entirely using the `DROP TABLE` statement.

```
-- Drop the Delta table
DROP TABLE IF EXISTS delta_table_name;
```

7. Merge for Upserts (Update + Insert)

Delta Lake's `MERGE` operation is used for **upserts**, which combines both updates and inserts.

Syntax:

```
MERGE INTO target_table AS t
USING source_table AS s
ON t.id = s.id
WHEN MATCHED THEN
  UPDATE SET t.name = s.name, t.age = s.age
WHEN NOT MATCHED THEN
  INSERT (id, name, age) VALUES (s.id, s.name, s.age);
```

- The `MERGE` statement updates existing records (when there is a match) and inserts new records (when there is no match).
- **WHEN MATCHED**: Specifies what to do when there is a match (perform an update).
- **WHEN NOT MATCHED**: Specifies what to do when there is no match (perform an insert).

Example:

```
-- Assume we have the following target and source tables

-- Target Table (delta_table_name)
-- +---+-------+---+
-- | id|  name |age|
-- +---+-------+---+
-- |  1|  Alice| 30|
-- |  2|   Bob | 25|
-- +---+-------+---+

-- Source Table (source_table_name)
-- +---+--------+---+
-- | id|   name |age|
-- +---+--------+---+
```

```
-- |  2|   Bobby| 26|  -- this will update Bob to Bobby
-- |  3| Charlie| 35|  -- this will insert a new row
-- +---+--------+---+

MERGE INTO delta_table_name AS target
USING source_table_name AS source
ON target.id = source.id
WHEN MATCHED THEN
  UPDATE SET target.name = source.name, target.age = sourc
e.age
WHEN NOT MATCHED THEN
  INSERT (id, name, age) VALUES (source.id, source.name, so
urce.age);
```

- In this example:
  - The `id=2` row is updated (Bob becomes Bobby).
  - The `id=3` row is inserted (Charlie is added).

Summary of Delta Lake Operations:

- **Create**: Use `CREATE TABLE` with `USING DELTA` to create a Delta table.
- **Insert**: Use `INSERT INTO` to add new records.
- **Select**: Query Delta tables using `SELECT`.
- **Update**: Modify existing records using `UPDATE`.
- **Delete**: Remove records using `DELETE`.
- **Merge**: Combine `UPDATE` and `INSERT` operations with the `MERGE` statement for upserts.

These SQL commands allow you to manage Delta Lake tables and perform ACID-compliant operations.

Advanced Delta Lake Operations in Databricks

Delta Lake offers several advanced features that help optimize performance, manage data efficiently, and ensure data consistency. Below are descriptions and examples for **OPTIMIZE**, **Z-ORDER**, **VACUUM**, **Time Travel**, and more.

2.3.1. OPTIMIZE - Compacting Data Files

The `OPTIMIZE` command compacts small data files in a Delta table into larger files, improving read performance.

Syntax:

```
OPTIMIZE delta_table_name;
```

- **Use Case**: When data is written frequently in small files (like streaming data), these files can be merged into larger ones to improve query performance.

2.3.2. Z-ORDER - Data Clustering for Faster Retrieval

`Z-ORDER` helps cluster data in a way that physically colocates similar values on disk, making queries on these columns faster.

Syntax:

```
OPTIMIZE delta_table_name
ZORDER BY (column_name);
```

- **Use Case**: If you frequently filter data by a specific column (like `date` or `user_id`), use `ZORDER BY` to cluster the data for faster retrieval.

Example:

```
OPTIMIZE delta_table_name
ZORDER BY (user_id);
```

- **Z-ORDER** will reorganize data to improve query speed when filtering by `user_id`.

2.3.3. VACUUM - Removing Stale Data

The `VACUUM` command removes old data files that are no longer referenced by Delta Lake. This is necessary for data cleanup.

Syntax:

```
VACUUM delta_table_name RETAIN 7 HOURS;
```

- **Default**: Retains data for 7 days unless specified.
- **Important Parameters**:
  - `spark.databricks.delta.retentionDurationCheck.enabled = false`: Disables the retention duration check if needed for shorter intervals.
  - `spark.databricks.delta.vacuum.logging.enabled = true`: Enables logging of vacuum operations.

Example:

```
-- Vacuum and retain data for only 2 hours
VACUUM delta_table_name RETAIN 2 HOURS;

-- Disable the retention duration check for this vacuum ope
ration
SET spark.databricks.delta.retentionDurationCheck.enabled =
false;
```

- **Caveat**: If Delta Cache is used, data might still be accessible from previous versions after the vacuum. To ensure complete deletion, restart the cluster.

2.4. Time Travel - Querying Historical Data

Delta Lake allows you to **travel back in time** and query previous versions of your data using the **version** or **timestamp**.

2.4.1. Query by Version or Timestamp

- **Query by Version**:

```
SELECT * FROM delta_table_name VERSION AS OF 5;
```

- **Query by Timestamp**:

```
SELECT * FROM delta_table_name TIMESTAMP AS OF '2023-09-
15T00:00:00';
```

2.4.2. Dry Run - Preview Files to be Deleted

Before performing the actual vacuum operation, you can see what files will be deleted with a **dry run**.

```
VACUUM delta_table_name DRY RUN;
```

- **Dry Run** will show you the files marked for deletion without deleting them.

2.4.3. RESTORE - Rolling Back to a Previous Version

If you need to undo changes and revert to a previous version of the Delta table, use the `RESTORE` command.

Syntax:

```
-- Restore a table to a previous version by version number
RESTORE delta_table_name TO VERSION AS OF 3;

-- Restore to a specific timestamp
RESTORE delta_table_name TO TIMESTAMP AS OF '2023-09-15T00:
00:00';
```

- **Use Case**: If a recent write introduced incorrect data, you can roll back to a previous version or timestamp.

Example Workflow: Full Delta Lake Operations

```
-- Step 1: Optimize and Z-ORDER the table for better perfor
mance
```

```
OPTIMIZE delta_table_name
ZORDER BY (user_id);

-- Step 2: Run a dry run before vacuuming to see what will
be deleted
VACUUM delta_table_name DRY RUN;

-- Step 3: Perform a vacuum operation to remove stale data
VACUUM delta_table_name RETAIN 7 DAYS;

-- Step 4: Query historical data using time travel
SELECT * FROM delta_table_name VERSION AS OF 5;

-- Step 5: Roll back the table to a previous version
RESTORE delta_table_name TO VERSION AS OF 3;
```

Summary of Key Features:

- **OPTIMIZE**: Combines small files into larger ones, improving performance.
- **Z-ORDER**: Clusters data based on column values for faster queries.
- **VACUUM**: Deletes old, unused data files based on a retention period.
- **Time Travel**: Allows querying and restoring previous versions of the table.
- **Dry Run**: Shows the files that will be deleted in a vacuum before actual deletion.
- **Restore**: Reverts a Delta table to a previous version or timestamp.

These advanced Delta Lake operations ensure efficient data management and performance optimization in Databricks.

Here's how you can create Views, Temporary Views, and Global Temporary Views using SQL in Databricks:

1. Creating a Standard View

A standard view is available within the session in which it is created. It is accessible only in the current session.

```
sql
Copy code
-- Create a standard view from a table or query
CREATE OR REPLACE VIEW people_view AS
SELECT id, name
FROM some_table;

-- Query the view
SELECT * FROM people_view;
```

- This creates a **view** called `people_view` which is accessible only within the current session. Once the session ends, the view is dropped.

---

2. Creating a Temporary View

A **Temporary View** is accessible only within the current session, and it's specific to the notebook or the cluster session.

```
sql
Copy code
-- Create a temporary view
CREATE OR REPLACE TEMP VIEW people_temp_view AS
SELECT id, name
FROM some_table;

-- Query the temporary view
SELECT * FROM people_temp_view;
```

- The `TEMP VIEW` is scoped to the current session (notebook-specific). It will not be accessible from other notebooks even if they are connected to the same cluster. It will be dropped when the session ends.

---

3. Creating a Global Temporary View

A **Global Temporary View** is accessible across all notebooks within the same cluster as long as the cluster is running. It's stored in the `global_temp` database.

```
sql
Copy code
-- Create a global temporary view
CREATE OR REPLACE GLOBAL TEMP VIEW people_global_view AS
SELECT id, name
FROM some_table;

-- Query the global temporary view
SELECT * FROM global_temp.people_global_view;
```

- Global temp views are stored in the `global_temp` database.
- Any notebook attached to the same cluster can access this view using `global_temp.people_global_view`.
- When the cluster is restarted, global temporary views are dropped.

Key Differences in Databricks SQL:

- **Standard Views**: Available within the session where they are created.
- **Temporary Views**: Session-specific and cannot be shared across notebooks.
- **Global Temporary Views**: Available cluster-wide and can be accessed by any notebook attached to the same cluster, as long as the cluster is running. Stored in the `global_temp` database.

Common Table Expressions (CTEs) in SQL

A **Common Table Expression (CTE)** is a temporary result set that can be referenced within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` query. CTEs improve readability and reusability of complex queries and exist only within the scope of the query in which they are defined.

Key Points About CTEs:

1. **Scope**: The scope of a CTE is limited to the query that defines it.
2. **Reusability**: CTEs allow reusability of complex query logic, making the SQL code cleaner and easier to understand.
3. **Temporary Nature**: Think of a CTE as a **temporary view** that only exists for the duration of the query. Once the query finishes executing, the CTE is no longer accessible.
4. **Difference from Temporary Tables**:
   - CTEs are available only in the current query scope.
   - **Temporary Tables** or **Table Variables** persist for the entire session, allowing multiple DML (Data Manipulation Language) operations.

Syntax for a CTE

```
WITH cte_name AS (
    -- Complex query or logic
    SELECT column1, column2
    FROM some_table
    WHERE some_condition
)
-- Main query using the CTE
SELECT *
FROM cte_name;
```

Example 1: Simple CTE

Here's an example of using a CTE to simplify a complex query in **Databricks SQL**:

```
-- Defining a CTE for reusable logic
WITH avg_age_cte AS (
    SELECT department, AVG(age) AS avg_age
    FROM employees
    GROUP BY department
)
```

```
-- Using the CTE in the main query
SELECT department, avg_age
FROM avg_age_cte
WHERE avg_age > 30;
```

- The CTE `avg_age_cte` calculates the average age for each department.
- The main query selects from this temporary result and applies additional filtering (`avg_age > 30`).

Example 2: Using Multiple CTEs

You can define multiple CTEs within a single query for even more complex logic.

```
WITH department_cte AS (
    SELECT department_id, department_name
    FROM departments
),
employee_cte AS (
    SELECT employee_id, employee_name, department_id
    FROM employees
)
-- Main query using both CTEs
SELECT e.employee_name, d.department_name
FROM employee_cte e
JOIN department_cte d ON e.department_id = d.department_id;
```

- In this example, there are two CTEs: `department_cte` and `employee_cte`.
- The main query joins the results from these two CTEs.

Difference Between CTE and Temporary Table:

Feature	CTE	Temporary Table
Scope	Within the same query	Exists for the duration of the session
Persistence	Disappears after query execution	Remains until session ends or explicitly dropped

DML Operations	Not possible	Can perform multiple DML operations (INSERT, UPDATE, DELETE)

Example 3: Difference from Temporary Table

CTE Example:

```
WITH sales_cte AS (
    SELECT region, SUM(sales) AS total_sales
    FROM sales_data
    GROUP BY region
)
SELECT *
FROM sales_cte
WHERE total_sales > 1000;
```

- The CTE `sales_cte` calculates total sales by region. It exists only during the execution of this query.

Temporary Table Example:

```
-- Creating a temporary table
CREATE TEMPORARY TABLE temp_sales AS
SELECT region, SUM(sales) AS total_sales
FROM sales_data
GROUP BY region;

-- Perform DML operations
SELECT * FROM temp_sales WHERE total_sales > 1000;

-- Drop the temporary table after use
DROP TABLE temp_sales;
```

- In this example, the temporary table `temp_sales` persists throughout the session and allows multiple operations before it's explicitly dropped.

Summary:

- **CTEs** are useful for simplifying complex queries, improving readability, and avoiding repeated logic within a single query.
- **CTEs** are temporary, only available during the execution of the query.
- **Temporary Tables**, by contrast, persist for the duration of the session and allow multiple DML operations.

CTEs are great for breaking down complex queries into manageable parts, while temporary tables are more suited for scenarios where multiple queries or operations need to be performed on the same dataset across a session.

ETL with Spark SQL in Databricks

In Databricks, Spark SQL allows you to efficiently perform ETL (Extract, Transform, Load) operations directly on various file formats or through connections to external data sources. Let's explore these processes in detail.

4.1. Querying Files Directly

Spark SQL allows you to query external data sources like CSV, JSON, Parquet, and more without needing to load them into a Delta Lake. You can query files or entire directories directly.

4.1.1. Supported File Formats

- **CSV**, **JSON**, **PARQUET**, **TEXT**, and **BINARY** formats are supported.
- You can perform SQL operations directly on these files.

4.1.2. Querying a Single File

You can query a single file by specifying its file path.

```
SELECT * FROM parquet.`/path/to/file.parquet`;
```

- Replace `parquet` with the desired format (`csv`, `json`, etc.).
- This allows you to query the contents of a file directly.

4.1.3. Querying a Directory

You can query an entire directory of files if they share the same format and schema.

```
SELECT * FROM parquet.`/path/to/directory/`;
```

- Spark assumes all files in the directory have the same schema.

4.1.4. Important Consideration:

When querying from a directory, ensure all files in the directory follow the same format and schema, or the query might fail.

4.2. Providing Options for External Sources

While querying files directly works well for formats like Parquet, JSON, or CSV, other data sources (like external databases) might require additional configuration, such as schema declaration or connection details.

4.2.1. Options for External Sources

For some data sources, such as SQL databases or external storage systems, you might need to provide specific options like credentials, table names, or schema information.

4.2.2. Registering External Data Sources

You can register external sources (such as object storage locations or non-Delta data) as tables in Databricks. This allows you to query external data directly, but note that performance guarantees from Delta Lake (ACID, caching, etc.) won't apply.

Syntax:

```
CREATE TABLE table_identifier (
    col_name1 col_type1,
    col_name2 col_type2,
    ...
)
USING data_source
```

```
OPTIONS (key1 = val1, key2 = val2, ...)
LOCATION '/path/to/data';
```

- `USING data_source`: Specifies the format of the data, such as `parquet`, `csv`, `json`, or `jdbc` (for databases).
- `OPTIONS`: Specifies additional parameters like credentials, file paths, or database configurations.
- `LOCATION`: Points to the external storage path.

Example:

```
CREATE TABLE my_table (
    name STRING,
    age INT
)
USING parquet
OPTIONS (header = "true")
LOCATION '/mnt/external_storage/my_data';
```

- This creates a table that points to an external Parquet file located in external storage.

Important Note:

- External tables do not offer the same performance and consistency guarantees as Delta tables.
- Use the `REFRESH TABLE` command to ensure the latest state of the external table is reflected in Databricks' cache:

  ```
  REFRESH TABLE my_table;
  ```

4.2.3. Extracting Data from SQL Databases (JDBC)

Databricks can connect to external SQL databases using **JDBC**. This is useful for extracting data from relational databases such as MySQL, SQL Server, PostgreSQL, or even data warehouses.

Syntax:

```
CREATE TABLE table_name
USING JDBC
OPTIONS (
    url = "jdbc:{databaseType}://{hostname}:{port}",
    dbtable = "{database}.table",
    user = "{username}",
    password = "{password}"
);
```

- `url`: JDBC URL to connect to the database, specifying the database type (e.g., MySQL, PostgreSQL), hostname, and port.
- `dbtable`: Specifies the table in the database to query.
- `user` and `password`: Authentication details for accessing the database.

Example:

```
CREATE TABLE sales_data
USING JDBC
OPTIONS (
    url = "jdbc:mysql://my-database-host:3306",
    dbtable = "salesdb.sales",
    user = "myUsername",
    password = "myPassword"
);
```

- This query connects to a MySQL database, retrieves the `sales` table, and registers it in Databricks.

Handling Large Datasets in External SQL Databases

When querying large datasets in external SQL databases, there are two approaches:

1. **Move the data into Databricks**: You can move the entire source table to Databricks using `CREATE TABLE` and then run your transformation logic locally on the Databricks cluster. This is useful when you want to work on the entire dataset at scale.

- **Pros**: Better performance since the data is available locally on Databricks.
- **Cons**: Might require significant data movement, leading to overhead.

2. **Push query logic down to the external database**: Instead of moving the entire table, push down specific query logic (like filtering, aggregating) to the external database. Only the query results are transferred to Databricks, reducing data movement.
   - **Pros**: Less data transferred over the network, reducing latency.
   - **Cons**: Performance is limited by the external database's capacity to process queries.

Example of Query Pushdown:

```
CREATE TABLE filtered_sales_data
USING JDBC
OPTIONS (
    url = "jdbc:postgresql://my-db-host:5432",
    dbtable = "(SELECT * FROM sales WHERE region = 'North A
merica') AS subquery",
    user = "dbuser",
    password = "dbpass"
);
```

- This pushes down the `SELECT` query to the PostgreSQL database, and only the filtered results (i.e., sales in North America) are transferred to Databricks.

Summary of Key Features:

- **Querying Files Directly**: You can directly query external file formats like CSV, Parquet, and JSON using `SELECT` statements in Spark SQL. You can query single files or entire directories.
- **Providing Options for External Sources**: For some external sources, additional configuration (schema, credentials, etc.) is required using the `CREATE TABLE` syntax with `OPTIONS`.

- **JDBC Connections**: Databricks can connect to external SQL databases using JDBC. You can either move data into Databricks or push queries down to the external system.
- **Performance Consideration**: External sources do not offer the same performance guarantees as Delta Lake, so ensure efficient querying by using `REFRESH TABLE` for external tables and consider network overheads for large datasets.

These capabilities allow you to integrate Databricks with a variety of external data sources and file formats, making it versatile for ETL operations.

Creating Delta Tables in Databricks

Delta Lake is a powerful storage layer that provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing. When creating Delta tables in Databricks, several methods and features help optimize data management and query performance.

4.3.1. CTAS (Create Table As Select)

CTAS statements are used to create new tables based on the results of a `SELECT` query. They automatically infer the schema from the query results, making them useful for ingesting external data like Parquet files or other well-structured sources.

Example:

```
CREATE TABLE delta_table_name AS
SELECT *
FROM parquet.`/path/to/file.parquet`;
```

- **Schema Inference**: CTAS automatically infers the schema from the query results.
- **No Schema Declaration**: CTAS does not allow you to manually declare a schema.

- **Limitations**: CTAS does not support additional file options like compression or specific configurations.

4.3.2. Declare Schema with Generated Columns

Generated columns are a special feature introduced in Databricks Runtime (DBR) 8.3. These columns are computed based on other columns in the Delta table, enabling easier table definition and more automation.

Example:

```
CREATE TABLE delta_table_name (
    id INT,
    birth_date DATE,
    age INT GENERATED ALWAYS AS (YEAR(current_date()) - YEA
R(birth_date))
);
```

- The column `age` is automatically calculated based on the difference between the current year and the `birth_date`.
- **Generated Columns** help enforce business logic automatically.

4.3.3. Table Constraints

Databricks supports two types of constraints for Delta tables:

1. **NOT NULL constraints**: Ensure that specific columns do not have `NULL` values.
2. **CHECK constraints**: Enforce conditions on columns to validate data integrity.

Example:

```
CREATE TABLE employees (
    id INT NOT NULL,
    salary DECIMAL(10, 2),
    CHECK (salary > 0)
) USING DELTA;
```

- The `NOT NULL` constraint ensures that the `id` column is always populated.
- The `CHECK` constraint ensures that `salary` is always greater than zero.

4.3.4. Enrich Tables with Additional Options and Metadata

Delta tables can be enriched with additional options such as metadata for tracking, partitioning, and external table declarations.

4.3.4.1. Spark SQL Built-In Functions:

- `current_timestamp()`: Automatically records the timestamp when data is ingested.
- `input_file_name()`: Records the source file from which the data was ingested.

Example:

```
CREATE TABLE enriched_table AS
SELECT
    name,
    age,
    current_timestamp() AS ingest_time,
    input_file_name() AS source_file
FROM parquet.`/path/to/files/`;
```

- This query creates a table where each record is enriched with metadata like the ingestion timestamp and the source file name.

4.3.4.2. CREATE TABLE with Additional Options:

- `COMMENT`: Allows you to add a descriptive comment to the table for easier discovery.
- `LOCATION`: Specifies an external storage location to create an external (unmanaged) table.
- `PARTITION BY`: Organizes the data by partitioning columns, making queries more efficient.

Example:

```
CREATE TABLE external_table
(
    name STRING,
    date DATE
)
USING DELTA
COMMENT 'This table stores external data'
LOCATION '/mnt/external_storage/data'
PARTITIONED BY (date);
```

- This creates a **partitioned external Delta table** stored at the specified location.

4.3.5. Cloning Delta Tables

Delta Lake supports table cloning, which allows you to create copies of Delta tables. There are two types of clones: **Deep Clone** and **Shallow Clone**.

4.3.5.1. Deep Clone:

- Fully copies both data and metadata from the source table to the target table.
- Changes to the source table can be incrementally synced with the target table by re-executing the clone command.
- Ideal for full backups or creating independent versions of the table.

Example:

```
CREATE TABLE target_table DEEP CLONE source_table;
```

- This performs a **Deep Clone**, copying both the data and metadata to a new table `target_table`.

4.3.5.2. Shallow Clone:

- Copies only the Delta transaction logs and metadata but does not move the actual data.

- This is useful for quickly creating a copy of a table for testing changes without modifying the original data.
- The data remains shared between the original and cloned tables.

Example:

```
CREATE TABLE test_clone SHALLOW CLONE original_table;
```

- This **Shallow Clone** creates a copy of the original table for quick testing.

4.3.5.3. CTAS vs. Shallow Clone:

- **CTAS** requires you to re-specify all options (like partitioning, constraints, etc.) and is less robust.
- **Clones** are faster, maintain the original structure (partitioning, constraints, etc.), and work incrementally.
- **Deep Clone** copies all the data and metadata, while **Shallow Clone** copies only the transaction logs.

Summary of Key Features:

- **CTAS (Create Table As Select)**: Automatically infers schema from query results. Ideal for ingesting external data with well-defined schemas.
- **Generated Columns**: Automatically calculate column values based on other columns, reducing the need for manual computation.
- **Table Constraints**: Enforce data integrity using `NOT NULL` and `CHECK` constraints.
- **Metadata Enrichment**: Enhance tables by recording metadata like ingestion timestamps or source file names.
- **Deep and Shallow Cloning**: Create full or partial copies of Delta tables for backup, testing, or versioning purposes.

Delta Lake's flexible table creation and management options offer powerful features for working with data in Databricks. Whether using CTAS, declaring schema with generated columns, or cloning tables, these options provide control, speed, and robustness for ETL and data management workflows.

Writing to Delta Tables in Databricks

Delta Lake provides various methods to write data into tables efficiently, depending on the use case—whether you want to overwrite the entire table, append new rows, or handle updates and deletes with a merge operation.

4.4.1. Summary of Writing Operations:

- **Overwrite** data tables using `INSERT OVERWRITE`.
- **Append** to a table using `INSERT INTO`.
- **Upsert (merge insert, update, and delete)** using `MERGE INTO`.
- **Incremental ingestion** using `COPY INTO`.

4.4.2. Complete Overwrites

Overwriting a table completely is useful when you need to replace all data atomically. There are two methods to accomplish this in Spark SQL:

Benefits of Overwriting:

- Faster than deleting and recreating the table because it doesn't need to delete files individually.
- Time travel is enabled; the previous version of the table still exists.
- Atomic operation ensures consistency.
- ACID guarantees ensure that if overwriting fails, the table remains in its previous state.

4.4.2.1. `CREATE OR REPLACE TABLE`:

This command creates a new table if it doesn't exist or completely replaces the existing table if it does.

Example:

```
CREATE OR REPLACE TABLE delta_table AS
SELECT *
FROM new_data_source;
```

- This will replace all data in `delta_table` with the results from `new_data_source`.

4.4.2.2. `INSERT OVERWRITE`:

- Similar to `CREATE OR REPLACE TABLE` but only works with existing tables.
- Can be used to overwrite entire tables or individual partitions.

Example:

```
INSERT OVERWRITE delta_table
SELECT *
FROM new_data_source;
```

- This overwrites the `delta_table` with data from `new_data_source`.

Key Differences:

- **Schema Enforcement**: `CREATE OR REPLACE TABLE` can redefine the table schema, whereas `INSERT OVERWRITE` will fail if the schema changes unless optional settings are provided.

4.4.3. Appending Rows

4.4.3.1. `INSERT INTO`:

This operation appends new rows to an existing Delta table. It is efficient for adding incremental data but does not prevent duplicate records.

Example:

```
INSERT INTO delta_table
SELECT *
FROM new_data_source;
```

- This appends new rows from `new_data_source` into `delta_table`, but duplicates are not avoided.

4.4.3.2. `MERGE INTO`:

This is the most advanced method for upserting data (inserting new records, updating existing ones, and deleting records). The `MERGE` statement ensures that inserts, updates, and deletes happen in a single transaction.

Key Benefits of `MERGE`:

- Handles inserts, updates, and deletes in one command.
- Avoids duplicating records.
- Can add custom logic for each conditional statement.

Example:

```
MERGE INTO delta_table AS target
USING new_data_source AS source
ON target.id = source.id
WHEN MATCHED THEN
    UPDATE SET target.name = source.name, target.age = sour
ce.age
WHEN NOT MATCHED THEN
    INSERT (id, name, age) VALUES (source.id, source.name,
source.age);
```

- **WHEN MATCHED**: Updates rows where the `id` matches.
- **WHEN NOT MATCHED**: Inserts new rows where there are no matches.

Insert-Only Merge for Deduplication:

```
MERGE INTO delta_table AS target
USING new_data_source AS source
ON target.id = source.id
WHEN NOT MATCHED THEN
    INSERT (id, name, age) VALUES (source.id, source.name,
source.age);
```

- This version avoids duplicates by only inserting non-matching records.

`INSERT INTO` vs. `MERGE INTO`:

- `INSERT INTO`: Simple append operation, but does not handle duplicates.
- `MERGE INTO`: Can avoid duplicates and perform updates/deletes in a single transaction.

4.4.4. Incremental Loading with `COPY INTO`

`COPY INTO` allows incremental ingestion of data from external systems. It is idempotent, meaning it can be executed multiple times without causing duplication or inconsistency, as long as the schema is consistent.

Example:

```
COPY INTO delta_table
FROM '/path/to/data/'
FILEFORMAT = 'parquet'
COPY_OPTIONS ('mergeSchema' = 'true')
PATTERN = '*.parquet';
```

- Ingests new data files into the `delta_table`.
- **File Pattern**: Specifies which files to copy (e.g., all `.parquet` files).
- **Incremental Ingestion**: Only new files are added, making this operation cheaper for large datasets.

Summary of Writing Techniques:

1. **Overwrite Table**:
   - `CREATE OR REPLACE TABLE`: Replace entire table with new data.
   - `INSERT OVERWRITE`: Overwrite existing table or partitions.
2. **Append Data**:
   - `INSERT INTO`: Append new rows but may introduce duplicates.
3. **Upserts (Merge)**:

- `MERGE INTO`: Handle inserts, updates, and deletes in a single operation. Prevents duplicates.

4. **Incremental Ingestion**:
   - `COPY INTO`: Efficiently ingest data incrementally from external systems.

These techniques give you flexible options depending on whether you need to overwrite, append, merge, or load data incrementally into Delta Lake tables.

Data Cleaning in Databricks SQL

Data cleaning is a critical step in data processing, ensuring that your datasets are consistent, free of errors, and structured correctly. Here are some common techniques and considerations for cleaning data using SQL in Databricks:

4.6.1. `COUNT(column)` vs `COUNT(*)`

- `COUNT(column)`: Counts **non-NULL** values in a specific column.
  - NULL values are ignored.
- `COUNT(*)`: Counts **all rows**, including rows where all values are NULL.

Example:

```
-- Count non-null values in a specific column
SELECT COUNT(email) FROM users;

-- Count total rows, including rows with all null values
SELECT COUNT(*) FROM users;
```

- **Result**: `COUNT(email)` skips rows where `email` is NULL, while `COUNT(*)` includes all rows, even those with NULLs in every column.

4.6.2. Counting NULL Values (`count_if` or `WHERE`)

To specifically count rows where a column has NULL values, you can use `count_if` or a `WHERE` clause with `IS NULL`.

Example:

```
-- Using WHERE to count rows where email is NULL
SELECT COUNT(*) FROM users WHERE email IS NULL;

-- Using count_if to count NULLs in the email column
SELECT COUNT_IF(email IS NULL) FROM users;
```

- `count_if`: A more explicit function for counting based on a condition.

4.6.3. `DISTINCT(*)` Behavior with NULLs

When using `DISTINCT(*)`, rows containing NULL values are **not omitted**. However, `DISTINCT(column)` will **ignore** NULL values in the specified column, but the row itself is still considered if other columns contain values.

Example:

```
-- Using DISTINCT on a column, ignoring rows where column i
s NULL
SELECT COUNT(DISTINCT email) FROM users;

-- Using DISTINCT(*) counts unique rows, including rows wit
h NULLs
SELECT COUNT(DISTINCT *) FROM users;
```

- `DISTINCT(*)` includes NULL values for columns but ensures overall row uniqueness.

4.6.4. Spark's Behavior with NULL Values

Spark treats NULL values differently in different contexts:

- **In** `COUNT(column)`: NULLs are ignored.
- **In** `DISTINCT(column)`: Rows with NULL values in that column are excluded.

- **In** `DISTINCT(*)`: All rows are considered, even those with NULL values, when evaluating uniqueness.

4.6.5. Other Important Functions for Data Cleaning

`WHERE`:

- Filters rows based on a condition, often used for identifying missing or invalid data.

Example:

```
-- Find rows where age is missing
SELECT * FROM users WHERE age IS NULL;
```

`GROUP BY`:

- Useful for aggregating data, especially when combined with `COUNT` or `SUM`.

Example:

```
-- Count users by age group
SELECT age, COUNT(*) FROM users
GROUP BY age;
```

`ORDER BY`:

- Sorts data, which can help identify outliers or errors in your data.

Example:

```
-- Order users by age in descending order
SELECT * FROM users ORDER BY age DESC;
```

Data Cleaning Strategies

- **Handle NULL values**: Decide whether to remove, impute, or analyze rows with missing data.

- **De-duplicate records**: Use `DISTINCT`, `GROUP BY`, or deduplication logic like `ROW_NUMBER` for eliminating duplicate entries.
- **Filter invalid data**: Use `WHERE` to identify and exclude rows with invalid values (e.g., negative age or invalid email format).
- **Remove duplicates**: Use `DISTINCT` or deduplication strategies to ensure clean data.

These techniques are useful in ensuring your datasets are cleaned and optimized for analysis or further processing.

Advanced SQL Transformations in Databricks

This section provides an overview of advanced SQL transformations used in Databricks SQL for working with nested data structures, arrays, JSON data, joins, set operators, and higher-order functions.

4.7.1 Interacting with JSON Data

Key Functions:

- `from_json`: Casts a field into a `struct` type, enabling you to parse a JSON string into table columns.
- *JSON. (Star Unpacking)*: Once a JSON string is cast to a `struct`, you can use `*` (wildcard) to unpack the JSON fields into individual columns.
- **Colon `:` Syntax**: Used to extract values from JSON strings.
- **Dot `.` Syntax**: Used to access specific fields within `struct` types.
- `schema_of_json`: Derives the JSON schema from an example JSON string.

Example:

```
-- Parse JSON data and unpack fields
SELECT from_json(json_column, 'struct<id:int, name:string,
```

```
age:int>') AS user_data
FROM users;

-- Access fields using dot notation
SELECT user_data.id, user_data.name, user_data.age
FROM (
  SELECT from_json(json_column, 'struct<id:int, name:strin
g, age:int>') AS user_data
  FROM users
);

-- Extract specific values using colon notation
SELECT json_column:field_name AS extracted_value
FROM users;
```

4.7.1.1 Working with Arrays in Spark SQL

Spark SQL offers several functions to manipulate arrays and their elements.

Key Functions:

- `explode()`: Expands an array, putting each element in its own row.
- `collect_set()`: Collects unique values in an array, eliminating duplicates.
- `flatten()`: Combines multiple arrays into a single array.
- `array_distinct()`: Removes duplicate elements from an array.

Example:

```
-- Explode an array into individual rows
SELECT user_id, explode(interests) AS interest
FROM user_interests;

-- Collect unique values within an array
SELECT collect_set(interest) AS unique_interests
FROM user_interests;

-- Flatten multiple arrays into one
SELECT flatten(array(array1, array2)) AS combined_array
```

```
FROM dataset;

-- Remove duplicates from an array
SELECT array_distinct(interests) AS unique_interests
FROM user_interests;
```

4.7.2 Join Operations

Databricks SQL supports various types of joins, including `INNER`, `OUTER`, `LEFT`, `RIGHT`, `ANTI`, `CROSS`, and `SEMI` joins. These join operations allow combining datasets based on specific conditions.

Example:

```
-- Inner join
SELECT a.*, b.*
FROM table_a a
INNER JOIN table_b b ON a.id = b.id;

-- Left join
SELECT a.*, b.*
FROM table_a a
LEFT JOIN table_b b ON a.id = b.id;

-- Cross join
SELECT a.*, b.*
FROM table_a a
CROSS JOIN table_b b;
```

4.7.3 Set Operators

Set operators combine the results of two or more queries.

- `MINUS`: Returns all rows found in one dataset but not in another.
- `UNION`: Combines results from two queries, returning unique rows.
- `INTERSECT`: Returns only the rows found in both queries.

Example:

```
-- Union of two queries
SELECT * FROM table_a
UNION
SELECT * FROM table_b;

-- Intersection of two queries
SELECT * FROM table_a
INTERSECT
SELECT * FROM table_b;
```

4.7.4 Pivot Tables

The `PIVOT` clause transforms rows into columns, offering a new perspective on the data by aggregating values across specific dimensions.

Example:

```
-- Pivot data by aggregating values across a pivot column
SELECT *
FROM (
  SELECT department, employee, salary
  FROM employee_salaries
) PIVOT (
  SUM(salary) FOR department IN ('HR', 'Engineering', 'Sale
s')
);
```

4.7.5 Pivot Syntax Breakdown

- The first argument is an aggregate function and the column to be aggregated.
- The pivot column is specified in the `FOR` subclause.
- The `IN` operator contains the pivot column values to be transformed into new columns.

4.7.6 Higher-Order Functions in Spark SQL

Higher-order functions allow transformation and manipulation of complex data types like arrays and maps.

Key Higher-Order Functions:

1. `FILTER`: Filters elements from an array based on a condition (lambda function).
2. `EXIST`: Checks if any element in the array meets a condition.
3. `TRANSFORM`: Transforms each element of an array using a lambda function.
4. `REDUCE`: Reduces elements of an array to a single value using two lambda functions.

Examples:

```
-- Filter an array for values greater than 50
SELECT filter(array_column, x -> x > 50) AS filtered_array
FROM dataset;

-- Check if any element in the array is greater than 50
SELECT exists(array_column, x -> x > 50) AS has_values_gt_5
0
FROM dataset;

-- Transform array elements by multiplying them by 2
SELECT transform(array_column, x -> x * 2) AS transformed_a
rray
FROM dataset;

-- Reduce an array to its sum
SELECT reduce(array_column, 0, (acc, x) -> acc + x, acc ->
acc) AS array_sum
FROM dataset;
```

These advanced transformations and functions provide powerful tools to work with structured, semi-structured (JSON), and hierarchical data (arrays and structs) in Databricks SQL, improving the ability to perform complex data processing efficiently.

SQL UDFs and Control Flow in Databricks SQL

This section focuses on how to create, use, and control the flow of SQL User-Defined Functions (UDFs) in Databricks SQL. UDFs allow for custom logic in SQL queries, and when combined with control flow constructs, they help optimize and streamline complex queries.

4.8.1 SQL User-Defined Functions (UDFs)

SQL UDFs allow you to define custom functions that can be reused in queries. A UDF requires a function name, optional parameters, the return type, and some custom logic.

Syntax:

```
CREATE OR REPLACE FUNCTION function_name(parameter_name TYP
E)
RETURNS return_type
RETURN custom_logic;
```

Example:

```
-- Define a UDF that calculates the square of a number
CREATE OR REPLACE FUNCTION square_number(x INT)
RETURNS INT
RETURN x * x;

-- Use the UDF in a query
SELECT square_number(4) AS result; -- Result: 16
```

4.8.2 Scoping and Permissions of SQL UDFs

SQL UDFs are persisted between different execution environments and follow similar access control mechanisms to other objects (e.g., tables, views) in the Databricks metastore.

4.8.2.1 Scope

- SQL UDFs **persist** between execution environments such as notebooks, DBSQL queries, and jobs. This means that once a UDF is created, it can be reused across different queries and environments until explicitly dropped or replaced.

4.8.2.2 Permissions

- SQL UDFs are stored as objects in the metastore, governed by Table ACLs (Access Control Lists).
- **Permissions**:
  - **USAGE**: Required for a user to reference the UDF.
  - **SELECT**: Required to query the UDF.

Users must have both **USAGE** and **SELECT** permissions on the function to execute it.

4.8.2.3 CASE / WHEN Control Flow

The `CASE` / `WHEN` construct in SQL is used for conditional logic, allowing you to evaluate multiple conditions and return different outcomes based on the results.

Syntax:

```
CASE
  WHEN condition_1 THEN result_1
  WHEN condition_2 THEN result_2
  ELSE default_result
END
```

Example:

```
-- Categorize age groups based on age column
SELECT name,
  CASE
    WHEN age < 18 THEN 'Minor'
    WHEN age BETWEEN 18 AND 65 THEN 'Adult'
    ELSE 'Senior'
  END AS age_group
FROM users;
```

4.8.2.4 Simple Control Flow Functions

Combining SQL UDFs with control flow constructs such as `CASE` / `WHEN` can help implement more complex logic and optimize SQL workloads.

Example - Combining UDF with CASE/WHEN:

```
-- Create a UDF to check if a number is even or odd
CREATE OR REPLACE FUNCTION is_even(x INT)
RETURNS STRING
RETURN CASE
  WHEN x % 2 = 0 THEN 'Even'
  ELSE 'Odd'
END;

-- Use the UDF in a query
SELECT number, is_even(number) AS number_type
FROM numbers_table;
```

Benefits of Using SQL UDFs with Control Flow

1. **Modularity**: UDFs allow you to encapsulate logic into reusable functions, making your SQL queries cleaner and more maintainable.
2. **Optimization**: SQL UDFs, when combined with `CASE`/`WHEN` and other control flow constructs, can improve query performance by streamlining execution logic.
3. **Flexibility**: Control flow functions like `CASE` allow for dynamic decision-making within queries, enabling more complex conditions and outputs.

SQL UDFs and control flow constructs enable better management of complex queries and workflows in Databricks SQL, supporting custom logic and enhanced data processing capabilities.

Incremental Data Processing in Databricks: Auto Loader vs COPY INTO

When working with large datasets, efficient and scalable incremental data processing is essential. Databricks provides two key mechanisms for ingesting and processing incremental data: **Auto Loader** and the **COPY INTO** command. Each has its specific use cases and advantages, depending on your workload requirements.

---

1. Auto Loader

Auto Loader is a Databricks feature that simplifies the process of ingesting streaming or batch data incrementally from cloud storage (e.g., AWS S3, Azure Data Lake Storage). It is highly scalable and designed for continuously processing new data files in an efficient manner.

Key Features of Auto Loader:

- **Continuous or Trigger-Based Ingestion**: Auto Loader can detect new files in a directory and process them either continuously (for streaming) or on a scheduled basis (trigger).
- **Schema Inference and Evolution**: Auto Loader can infer the schema of incoming data and automatically evolve the schema to handle changes over time.
- **File Metadata Tracking**: Auto Loader keeps track of which files have already been processed to avoid duplicate processing.
- **Scalability**: It scales to large directories with millions of files by distributing the workload across many machines.

How Auto Loader Works:

Auto Loader listens to a directory for new data files, and as new files are detected, they are incrementally processed and loaded into Delta Lake or another sink.

Example with Auto Loader:

```
# Set up Auto Loader to incrementally ingest data from clou
d storage

df = (spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")  # Input data format
  .option("cloudFiles.schemaLocation", "/mnt/schemas/")  #
Location to store schema
  .load("/mnt/data/input/"))  # Directory to monitor for ne
w files

df.writeStream
  .format("delta")  # Write data to Delta Lake
  .option("checkpointLocation", "/mnt/checkpoints/")
  .outputMode("append")
  .start("/mnt/delta/output/")
```

Benefits of Auto Loader:

- **File Notification System**: Integrates with cloud storage file notification systems, reducing the need for full directory scans and improving efficiency.
- **Fault Tolerance**: Automatically tracks processed files, ensuring that no file is missed or processed multiple times, even in the event of failures.
- **Real-Time Processing**: Ideal for streaming use cases where data arrives incrementally and needs to be processed in near-real-time.
- **Schema Evolution**: Handles changes in data structure without manual intervention.

---

2. COPY INTO

The **COPY INTO** command is another method for incrementally loading data into Delta Lake tables. It is designed for batch data processing and is particularly useful when you want to ingest new data files efficiently but do not require continuous streaming.

Key Features of COPY INTO:

- **Batch File Loading**: Best suited for batch processing, where files are loaded periodically rather than in a continuous stream.
- **File-based Ingestion**: It scans the specified directory for new files and loads them into a Delta table.
- **File Tracking**: Similar to Auto Loader, `COPY INTO` keeps track of which files have been ingested, ensuring no duplication of data.

Example of COPY INTO:

```
COPY INTO delta.`/mnt/delta/output/`
FROM '/mnt/data/input/'
FILEFORMAT = JSON
PATTERN = '.*json'  -- Specify pattern for the files to be
copied
COPY_OPTIONS ('mergeSchema' = 'true');  -- Enable schema ev
olution
```

Benefits of COPY INTO:

- **Simpler Configuration**: Fewer moving parts compared to Auto Loader, making it easier to set up for smaller batch workloads.
- **Efficient for Small/Batch Loads**: Ideal when you don't need the complexity of streaming and can work with periodic batch loading.
- **Schema Evolution**: It can handle schema evolution when using Delta tables with the `mergeSchema` option.
- **File Pattern Matching**: You can specify patterns to load only certain types of files.

Comparison: Auto Loader vs COPY INTO

Feature	**Auto Loader**	**COPY INTO**
Ingestion Mode	Streaming or batch (continuous file detection)	Batch (manual or scheduled execution)
Use Case	Large-scale, real-time, or streaming data processing	Smaller batch loads or periodic updates
File Tracking	Automatic, with file notification system	File tracking based on Delta's metadata
Scalability	Highly scalable, handles millions of files	Suitable for smaller or moderate datasets
Schema Evolution	Supports automatic schema evolution	Supports schema evolution with `mergeSchema`
Fault Tolerance	Built-in fault tolerance with metadata tracking	Metadata-based file tracking for fault tolerance
Configuration Complexity	Higher, requires stream management	Simpler configuration for periodic loads
Efficiency	Optimized for large-scale, real-time streaming	Efficient for batch processing

When to Use Auto Loader:

- **Real-Time or Near Real-Time Data Processing**: When you need continuous data ingestion with low latency.
- **Handling Large Directories**: For directories with millions of files where efficient file management is critical.
- **Schema Evolution**: When dealing with frequently changing data structures.

When to Use COPY INTO:

- **Batch Processing**: When you have periodic data loads and don't need continuous streaming.
- **Simpler Use Cases**: If the workload is relatively small and doesn't need the complexity of a streaming pipeline.
- **Manual File Ingestion**: For occasional, manual ingestion of new data files.

Conclusion

Both **Auto Loader** and **COPY INTO** are powerful tools for incremental data processing in Databricks. **Auto Loader** is designed for high-volume, real-time streaming use cases, while **COPY INTO** is more suited for batch-oriented workflows. The choice between them depends on your use case, the scale of your data, and whether you need continuous or batch processing capabilities.

Medallion Architecture: Bronze, Silver, and Gold Layers

The Medallion Architecture is a data management and organization strategy commonly used in data lakes to structure data in a layered format. This allows for incremental processing and enhances data quality as it moves through different stages. The architecture follows a **Bronze**, **Silver**, and **Gold** pattern to manage raw, cleansed, and refined data for analytics and reporting.

7.1. Bronze Layer – Raw Data Ingestion

The **Bronze** layer is the initial stage of the architecture, where raw data is ingested and stored in its original format. The main goal of this layer is to capture data as-is, often in an append-only mode, without applying any transformations or cleansings. This provides an immutable record of the raw data for future reprocessing or auditing.

Key Features:

- **Raw Data Storage**: All ingested data is stored in its original, raw form.
- **Append-Only Mode**: New data is appended to the existing dataset without any modifications.
- **Minimal Processing**: No transformations or filtering are applied at this stage, just ingestion.

Use Cases:

- **Historical Data**: Stores all historical records as-is, enabling reprocessing or lineage tracking.

- **Auditability**: Raw data is kept for audit purposes to ensure transparency and traceability.

Example:

```
-- Ingest raw data from an external source into the Bronze
layer
CREATE OR REPLACE TABLE bronze.customers
AS SELECT * FROM external_source.customers_raw;
```

7.2. Silver Layer – Cleansed and Enriched Data

The **Silver** layer represents data that has been **cleansed**, **filtered**, and **augmented**. This layer processes the raw data from the Bronze layer to remove duplicates, handle missing values, and apply necessary transformations. The result is a more refined, structured, and accurate dataset ready for business logic or analytics.

Key Features:

- **Data Cleaning**: Handles missing data, outliers, and duplicates.
- **Incremental Processing**: Usually processed in append mode, just like Bronze, but with more refined and enriched data.
- **Schema Alignment**: Ensures that the data schema is well-defined and consistent across different datasets.

Use Cases:

- **Intermediate Data**: Serves as the source for more refined analysis, allowing for the application of business rules.
- **Enhanced Data Quality**: Cleansed data ready for more complex analytics or joins with other datasets.

Example:

```
-- Create a Silver table by cleaning and enriching the Bron
ze data
CREATE OR REPLACE TABLE silver.customers_cleaned
AS
```

```
SELECT DISTINCT
    customer_id,
    name,
    email,
    address
FROM bronze.customers
WHERE email IS NOT NULL;
```

7.3. Gold Layer – Aggregated Business Metrics and Analytics

The **Gold** layer is the final stage where business-level **aggregates**, **metrics**, and **analytics-ready data** are produced. Data in the Gold layer is structured to support reporting, business intelligence, and dashboarding. This layer usually involves more advanced transformations, aggregations, and business logic.

Key Features:

- **Business-Focused**: Designed for business reporting and dashboards.
- **Aggregations and Metrics**: Involves business logic, KPIs, and other calculated metrics.
- **Complete Mode**: Often processed in "complete" mode, which means data is refreshed periodically based on specific business rules.

Use Cases:

- **Reporting**: Provides data for dashboards, reporting tools, or decision-making systems.
- **Business Intelligence**: Enables querying of aggregated business metrics, trends, and KPIs.

Example:

```
-- Create a Gold table by aggregating data from the Silver
layer
CREATE OR REPLACE TABLE gold.customer_sales_metrics
AS
SELECT
    customer_id,
    COUNT(order_id) AS total_orders,
```

```
    SUM(order_amount) AS total_revenue
FROM silver.orders
GROUP BY customer_id;
```

Summary of the Medallion Architecture:

Layer	Purpose	Data Quality	Processing Mode
Bronze	Store raw, unprocessed data from various sources.	Raw and unstructured data	Append-only
Silver	Cleanse, filter, and enrich data for analytical use.	Cleansed and structured	Append or incremental
Gold	Provide business-level aggregates for reporting.	Refined and aggregated data	Complete refresh

When to Use Each Layer:

- **Bronze**: When you need to capture raw data for historical records, auditing, or raw-data exploration.
- **Silver**: When you need to process the data to remove errors, filter out bad records, and structure the data for business logic.
- **Gold**: When you want to build dashboards, reports, or business intelligence insights based on clean, aggregated data.

The **Medallion Architecture** helps create a structured data lake with reliable data pipelines, where each layer increases data quality and usability as it moves from raw to refined.

Delta Live Tables (DLT) Overview

Delta Live Tables (DLT) is a declarative framework in Databricks for building reliable, efficient, and automated data pipelines. It simplifies the creation of ETL processes by providing built-in features for managing data quality, transformations, and continuous or scheduled processing modes.

8.1. Declarative ETL Language

Delta Live Tables uses a **declarative** approach to define ETL (Extract, Transform, Load) pipelines. Instead of writing complex logic in procedural code, users simply define **what** the pipeline should accomplish, and DLT handles **how** the process should be executed.

- **Simplified Syntax**: DLT abstracts the complexities of managing stateful data pipelines, allowing developers to focus on the transformations rather than the infrastructure.
- **Scalability and Performance**: DLT automatically optimizes the pipeline for performance and scalability, handling schema changes, dependencies, and incremental updates.

8.2. LIVE Keyword

In Delta Live Tables, all tables and views are prefixed with the `LIVE` keyword, which ensures that they are managed by DLT and treated as part of a live pipeline.

- **LIVE Tables**: Tables created and maintained by Delta Live Tables that can evolve over time.
- **LIVE Views**: Views are treated as logical representations of data, dynamically updated as source data changes.

Example:

```
-- Define a live table in DLT
CREATE OR REPLACE LIVE TABLE silver_customers AS
SELECT * FROM bronze_customers
WHERE is_active = true;
```

8.3. Two Modes: Continuous vs Triggered Mode

Delta Live Tables can operate in two distinct modes, depending on the desired frequency of data processing:

1. **Continuous Mode**: DLT continuously monitors the source data and updates downstream tables in near real-time as new data arrives. This mode is ideal for streaming or near real-time use cases.

2. **Triggered Mode**: In this mode, DLT pipelines run in batches, only processing new data at specific intervals. This mode is useful for traditional batch-based data processing where updates happen periodically.

- **Continuous Mode Example**: Use this for pipelines that process streaming data, like IoT sensors or real-time logs.
- **Triggered Mode Example**: Use this for nightly batch updates, such as daily sales reports.

8.4. Quality Control

Delta Live Tables has built-in features to manage **data quality** through constraints. The goal is to ensure that invalid or erroneous data does not make its way into downstream tables or dashboards.

8.4.1. CONSTRAINT Keyword

The `CONSTRAINT` keyword allows you to define data validation rules, similar to a `WHERE` clause. It integrates with Delta Live Tables to collect metrics on constraint violations and provides multiple options for handling invalid data.

- **ON VIOLATION**: Specifies the action to take when a record violates a constraint. There are three main behaviors:
  1. **FAIL UPDATE**: The pipeline will fail if a constraint is violated.
  2. **DROP ROW**: The row violating the constraint will be dropped.
  3. **Omitted**: If no action is specified, the pipeline will allow the violation but will log it as part of the metrics.

Example of Constraint:

```
-- Defining a constraint on the age column
CREATE OR REPLACE LIVE TABLE silver_customers
  CONSTRAINT valid_age CHECK (age >= 18) ON VIOLATION DROP
ROW
AS
SELECT * FROM bronze_customers;
```

8.4.2. References to DLT Tables and Views

- When referencing other Delta Live Tables (DLT) or views within a pipeline, you always use the `live.` prefix.
- This ensures that DLT can automatically resolve references between tables at runtime and substitute database names, facilitating easy migration between different environments (DEV, QA, PROD).

Example:

```
-- Reference to another DLT table
CREATE OR REPLACE LIVE TABLE gold_customer_metrics AS
SELECT customer_id, COUNT(order_id) AS total_orders
FROM live.silver_orders
GROUP BY customer_id;
```

8.4.3. References to Streaming Tables

- For streaming tables, use the `STREAM()` function to refer to a DLT table in a streaming context. This ensures that the table is treated as a continuously updating source of data.

Example:

```
-- Streaming reference to another DLT table
CREATE OR REPLACE LIVE TABLE gold_real_time_orders AS
SELECT * FROM STREAM(live.silver_orders);
```

Summary of Key Features

Feature	Description
Declarative Language	Simplifies the creation of ETL pipelines by focusing on the logic, not the infrastructure.
LIVE Keyword	Prefix used to define DLT-managed tables and views.
Continuous vs Triggered Mode	Continuous mode for streaming or real-time updates, triggered mode for periodic batch updates.
CONSTRAINT Keyword	Introduces quality control by enforcing validation rules on data and handling violations with flexible options like `FAIL UPDATE` or `DROP ROW`.

Streaming Tables	Use the `STREAM()` function to reference tables that are continuously updated in real-time streaming scenarios.

Conclusion

Delta Live Tables provide a robust framework for building, managing, and monitoring ETL pipelines. By using a declarative approach with built-in data quality controls and support for both batch and streaming data, DLT simplifies data processing and enhances the reliability of pipelines, ensuring clean and accurate data in production.

Managing Permissions in Databricks

Effective data access control is crucial in a multi-user environment like Databricks. Permissions management ensures that the right users have access to the right data at the right time, with appropriate restrictions. In Databricks, permissions can be managed through the **Data Explorer** interface or by using SQL commands such as `GRANT` and `REVOKE`.

11.1. Data Explorer

The **Data Explorer** is a visual tool within Databricks that simplifies the process of managing permissions and exploring relational data objects like databases, tables, and views.

11.1.1. What is the Data Explorer?

The **Data Explorer** offers the following functionalities:

- **Set and Modify Permissions**: It allows administrators to configure **role-based access control (RBAC)** by assigning or revoking privileges to specific users or groups on databases, tables, or views.
- **Explore Databases, Tables, and Views**: Users can navigate the entire relational structure of a Databricks workspace, viewing details about databases, tables, views, and more.

- **Schema, Metadata, and History**: Users can explore the schema of a table or view metadata like creation time, owner, and recent operations (e.g., updates or modifications).

Key Features of Data Explorer:

- **Permission Management**: Easily assign or remove permissions for users or groups.
- **Object Navigation**: View databases, tables, and views along with their properties and history.
- **Metadata Insight**: Examine schema details and table history for auditing and data governance purposes.

11.1.2. GRANT/REVOKE Syntax

For advanced or programmatic control over permissions, Databricks supports standard SQL syntax for granting or revoking privileges on specific data objects.

GRANT Command:

The `GRANT` command is used to assign specific privileges (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`) on a given object to a user or group.

REVOKE Command:

The `REVOKE` command removes previously granted privileges from a user or group.

Syntax:

```
-- Granting privileges to a user or group
GRANT <<PRIVILEGES>> ON <<OBJECT>> TO <<USER or GROUP>>;

-- Revoking privileges from a user or group
REVOKE <<PRIVILEGES>> ON <<OBJECT>> FROM <<USER or GROUP>>;
```

Common Privileges:

- **SELECT**: Grants the ability to read data from the object (database, table, or view).

- **INSERT**: Grants the ability to insert new data into the object.
- **UPDATE**: Grants the ability to modify existing data in the object.
- **DELETE**: Grants the ability to remove data from the object.
- **USAGE**: Grants the ability to use the object (typically on databases).
- **ALL**: Grants all possible privileges on the object.

Examples:

- **Grant SELECT Privilege on a Table**:

```
GRANT SELECT ON TABLE sales_data TO user123;
```

- **Grant ALL Privileges on a Database**:

```
GRANT ALL PRIVILEGES ON DATABASE marketing_db TO group_data
_analysts;
```

- **Revoke INSERT Privilege from a Table**:

```
REVOKE INSERT ON TABLE sales_data FROM user123;
```

Summary of Permissions Management

Feature	Description
Data Explorer	A visual tool in Databricks for navigating data objects and managing permissions.
Set and Modify Permissions	Easily assign or revoke permissions using the Data Explorer interface or SQL syntax.
GRANT Command	Assigns privileges to users or groups on specific objects (databases, tables, views).
REVOKE Command	Removes previously assigned privileges from users or groups.

Use Cases for Permissions Management:

- **Restrict Data Access**: Ensure that sensitive data is only accessible to authorized users by applying **SELECT** restrictions on tables.

- **Granular Data Control**: Fine-tune permissions for different roles within an organization (e.g., allowing analysts to query data but not modify it).
- **Data Governance**: Use permission management in conjunction with auditing tools to ensure that data access is secure and compliant with regulations.s

By using the **Data Explorer** and the **GRANT/REVOKE** syntax, Databricks administrators can easily manage who can access, modify, or delete data, ensuring the security and integrity of the data lake.

SQL HIGHER-ORDER FUNCTION

Las **funciones de alto orden** (*higher-order functions*) en SQL se refieren a funciones que reciben otras funciones como argumentos o devuelven funciones como resultado. Estas funciones son útiles cuando trabajas con estructuras de datos más complejas, como arrays, mapas o columnas de tipo `struct`, permitiéndote aplicar transformaciones a estos datos.

En **Spark SQL**, las funciones de alto orden son comunes al trabajar con arrays y mapas. Aquí te dejo algunas funciones de alto orden que podrías utilizar en Databricks, junto con ejemplos:

1. `TRANSFORM`

Aplica una función a cada elemento de un array y devuelve un array con los resultados.

- **Sintaxis**: `TRANSFORM(array, x -> función(x))`

Ejemplo:

```
SELECT TRANSFORM(ARRAY(1, 2, 3), x -> x * 2) AS resultado;
-- Resultado: [2, 4, 6]
```

En este caso, se multiplica cada elemento del array por 2.

2. `FILTER`

Filtra los elementos de un array basado en una condición.

- **Sintaxis**: `FILTER(array, x -> condición)`

Ejemplo:

```
SELECT FILTER(ARRAY(1, 2, 3, 4, 5), x -> x % 2 = 0) AS pare
s;
-- Resultado: [2, 4]
```

Este ejemplo devuelve solo los números pares del array.

3. `AGGREGATE`

Aplica una función de agregación sobre un array, permitiendo combinar sus elementos de manera personalizada.

- **Sintaxis**: `AGGREGATE(array, inicial, (acc, x) -> función(acc, x), acc_final -> función_final(acc))`

Ejemplo:

```
SELECT AGGREGATE(ARRAY(1, 2, 3, 4), 0, (acc, x) -> acc + x)
AS suma_total;
-- Resultado: 10
```

Este ejemplo suma los elementos del array, partiendo de un valor inicial de 0.

4. `ZIP_WITH`

Combina dos arrays elemento por elemento usando una función.

- **Sintaxis**: `ZIP_WITH(array1, array2, (x, y) -> función(x, y))`

Ejemplo:

```
SELECT ZIP_WITH(ARRAY(1, 2, 3), ARRAY(4, 5, 6), (x, y) -> x
+ y) AS suma_arrays;
-- Resultado: [5, 7, 9]
```

Aquí se suman los elementos correspondientes de dos arrays.

5. `TRANSFORM_KEYS`

Aplica una función a las llaves de un mapa (*map*).

- **Sintaxis:** `TRANSFORM_KEYS(map, (k, v) -> función(k, v))`

Ejemplo:

```
SELECT TRANSFORM_KEYS(MAP('a', 1, 'b', 2), (k, v) -> CONCAT
(k, '_key')) AS resultado;
-- Resultado: {"a_key": 1, "b_key": 2}
```

En este caso, se modifica cada llave del mapa concatenándole `'_key'`.

6. `TRANSFORM_VALUES`

Aplica una función a los valores de un mapa.

- **Sintaxis:** `TRANSFORM_VALUES(map, (k, v) -> función(k, v))`

Ejemplo:

```
SELECT TRANSFORM_VALUES(MAP('a', 1, 'b', 2), (k, v) -> v *
10) AS resultado;
-- Resultado: {"a": 10, "b": 20}
```

Aquí se multiplica cada valor del mapa por 10.

7. `MAP_FILTER`

Filtra los pares clave-valor de un mapa según una condición.

- **Sintaxis:** `MAP_FILTER(map, (k, v) -> condición)`

Ejemplo:

```
SELECT MAP_FILTER(MAP('a', 1, 'b', 2, 'c', 3), (k, v) -> v
> 1) AS resultado;
-- Resultado: {"b": 2, "c": 3}
```

Este ejemplo devuelve solo los pares cuyo valor es mayor que 1.

8. `ARRAY_SORT`

Ordena los elementos de un array basado en una función de comparación.

- **Sintaxis**: `ARRAY_SORT(array, (x, y) -> función(x, y))`

Ejemplo:

```
SELECT ARRAY_SORT(ARRAY(3, 1, 2), (x, y) -> CASE WHEN x < y
THEN -1 WHEN x > y THEN 1 ELSE 0 END) AS ordenado;
-- Resultado: [1, 2, 3]
```

Este ejemplo ordena un array en orden ascendente.

Ventajas de las Funciones de Alto Orden

- **Flexibilidad**: Puedes realizar operaciones complejas sobre estructuras de datos como arrays y mapas sin necesidad de descomponerlos.
- **Reducción de Código**: Estas funciones permiten realizar operaciones de una manera más concisa, en lugar de escribir código imperativo con bucles.
- **Optimización**: Spark SQL optimiza automáticamente las transformaciones usando estas funciones, lo que mejora el rendimiento.

¿Te gustaría practicar más alguna de estas funciones o tienes algún caso específico en mente para implementar?