Creating Delta Table Methods

Bavatharani S

Method 1 – Create Delta Table Using SQL

```
-- Drop if already exists

DROP TABLE IF EXISTS products_sql;

-- Create directly via SQL

CREATE TABLE products_sql (

  product_id INT,

  product_name STRING,

  category STRING,

  price DOUBLE

) USING DELTA;

-- Insert some sample rows

INSERT INTO products_sql VALUES

(1, "Laptop CE", "Electronics", 1200.00),

(2, "Tablet CE", "Electronics", 500.00),

(3, "Earbuds CE", "Accessories", 90.00);


select * from products_sql;
```

Table

	product_id	product_name	category	price
1	1	Laptop CE	Electronics	1200
2	2	Tablet CE	Electronics	500
3	3	Earbuds CE	Accessories	90

3 rows | 7.63s runtime

Refreshed now

This result is stored as `_sqldf` and can be used in other Python and SQL cells.

Method 2 – Convert Existing Parquet Data to Delta Table

```
# Create sample Parquet file first

data = [(10, "Camera CE", "Electronics", 800.00),

      (11, "Headphones CE", "Accessories", 150.00)]

cols = ["product_id", "product_name", "category", "price"]


df = spark.createDataFrame(data, cols)

display(df)


# Save as managed Delta table in Unity Catalog

df.write.format("delta").mode("overwrite").saveAsTable("catalog.schema.products_ce")
```

Table

	product_id	product_name	category	price
1	10	Camera CE	Electronics	800
2	11	Headphones CE	Accessories	150

Method 3 – Create Delta Table by Writing DataFrame (PySpark)

```
# Sample data

data2 = [

    (20, "Monitor CE", "Electronics", 300.00),

    (21, "Keyboard CE", "Accessories", 75.00)

]

cols = ["product_id", "product_name", "category", "price"]

df2 = spark.createDataFrame(data2, cols)


# Save as a managed Delta table (no /tmp path needed in CE)

df2.write.format("delta").mode("overwrite").saveAsTable("products_df")
```

Verify the data

spark.sql("SELECT * FROM products_df").show()

```
▸ df2: pyspark.sql.connect.dataframe.DataFrame = [product_id: long, product_name: string ... 2 more fields]
+----------+-----------+-----------+-----+
|product_id|product_name|   category|price|
+----------+-----------+-----------+-----+
|        20|  Monitor CE|Electronics|300.0|
|        21| Keyboard CE|Accessories| 75.0|
+----------+-----------+-----------+-----+
```

Delta Lake Merge & Upsert (SCD)

Step 1: Create Initial Delta Table

```
# Base data
base_data = [
  (101, "Laptop Air", "Electronics", 999.99),
  (102, "Tablet Plus", "Electronics", 499.00),
  (103, "Wireless Earbuds", "Accessories", 89.50)
]
cols = ["product_id", "product_name", "category", "price"]
base_df = spark.createDataFrame(base_data, cols)

# Save as a managed Delta table (works in CE)
base_df.write.format("delta").mode("overwrite").saveAsTable("products_merge")

# Verify table creation
spark.sql("SELECT * FROM products_merge").show()
```

```
▸ base_df: pyspark.sql.connect.dataframe.DataFrame = [product_id: long, product_name: string ... 2 more fields]
+----------+----------------+-----------+------+
|product_id|    product_name|   category| price|
+----------+----------------+-----------+------+
|       101|      Laptop Air|Electronics|999.99|
|       102|     Tablet Plus|Electronics| 499.0|
|       103|Wireless Earbuds|Accessories|  89.5|
+----------+----------------+-----------+------+
```

Step 2: Prepare Incoming Updates

```
update_data = [
    (102, "Tablet Plus", "Electronics", 550.00),  # Updated price
    (104, "Smartwatch CE", "Wearables", 220.00)   # New product
]
update_df = spark.createDataFrame(update_data, cols)
update_df.show()
```

```
▸ update_df: pyspark.sql.connect.dataframe.DataFrame = [product_id: long, product_name: string ... 2 more fields]
+----------+------------+-----------+-----+
|product_id| product_name|   category|price|
+----------+------------+-----------+-----+
|       102|  Tablet Plus|Electronics|550.0|
|       104|Smartwatch CE|  Wearables|220.0|
+----------+------------+-----------+-----+
```

Step 3: Perform Merge (Upsert)

```
from delta.tables import DeltaTable


# Load Delta table from metastore
deltaTable = DeltaTable.forName(spark, "products_merge")
deltaTable.alias("old").merge(
    update_df.alias("new"),
    "old.product_id = new.product_id"
).whenMatchedUpdate(set={
    "product_name": "new.product_name",
    "category": "new.category",
    "price": "new.price"
}).whenNotMatchedInsert(values={
    "product_id": "new.product_id",
    "product_name": "new.product_name",
    "category": "new.category",
    "price": "new.price"
```

```
}).execute()
```

```
DataFrame[num_affected_rows: bigint, num_updated_rows: bigint, num_deleted_rows: bigint, num_inserted_rows: bigint]
```

Internals of a Delta Table

1. Storage Format

Delta Lake stores its underlying data in Parquet files, which provide efficient columnar storage and compression. Alongside these files, Delta also maintains a special folder called _delta_log, where all operations and metadata are tracked. This combination allows Delta to handle both the raw data and the transactional consistency layer together.

2. Transaction Log

Every change in a Delta table is recorded in the transaction log, located in the _delta_log directory. The log contains ordered JSON files, where each file represents a transaction such as adding or removing data files, updating schemas, or modifying metadata. Periodically, checkpoint Parquet files are also created to improve performance by reducing the need to replay the entire log during queries.

3.Metadata

Metadata is stored in the transaction log and contains important information such as the table schema, column types, partitioning strategy, and table properties. This enables Delta Lake to enforce schema consistency, while also allowing schema evolution when changes are required in the data structure.

4.ACID Transactions

Delta Lake provides full support for ACID transactions. This means that all operations are Atomic (they either succeed completely or fail without partial changes), Consistent (data adheres to schema rules), Isolated (concurrent operations do not interfere with each other), and Durable (once committed, data is reliably stored). These guarantees make Delta suitable for mission-critical workloads.

5. Versioning & Time Travel

One of the powerful features of Delta Lake is time travel. Every transaction increases the version number of the table, and users can query historical data using commands like VERSION AS OF or TIMESTAMP AS OF. This allows rollback to previous states, auditing of historical records, or reproducing past analyses with exact data consistency.

6. Compaction & Cleanup

Over time, frequent writes can lead to the accumulation of many small Parquet files, which slows down query performance. Delta Lake addresses this with the OPTIMIZE command, which compacts small files into larger ones for efficiency. Additionally, the VACUUM operation removes obsolete files that are no longer referenced in the transaction log, helping free up storage space and keeping the table clean.