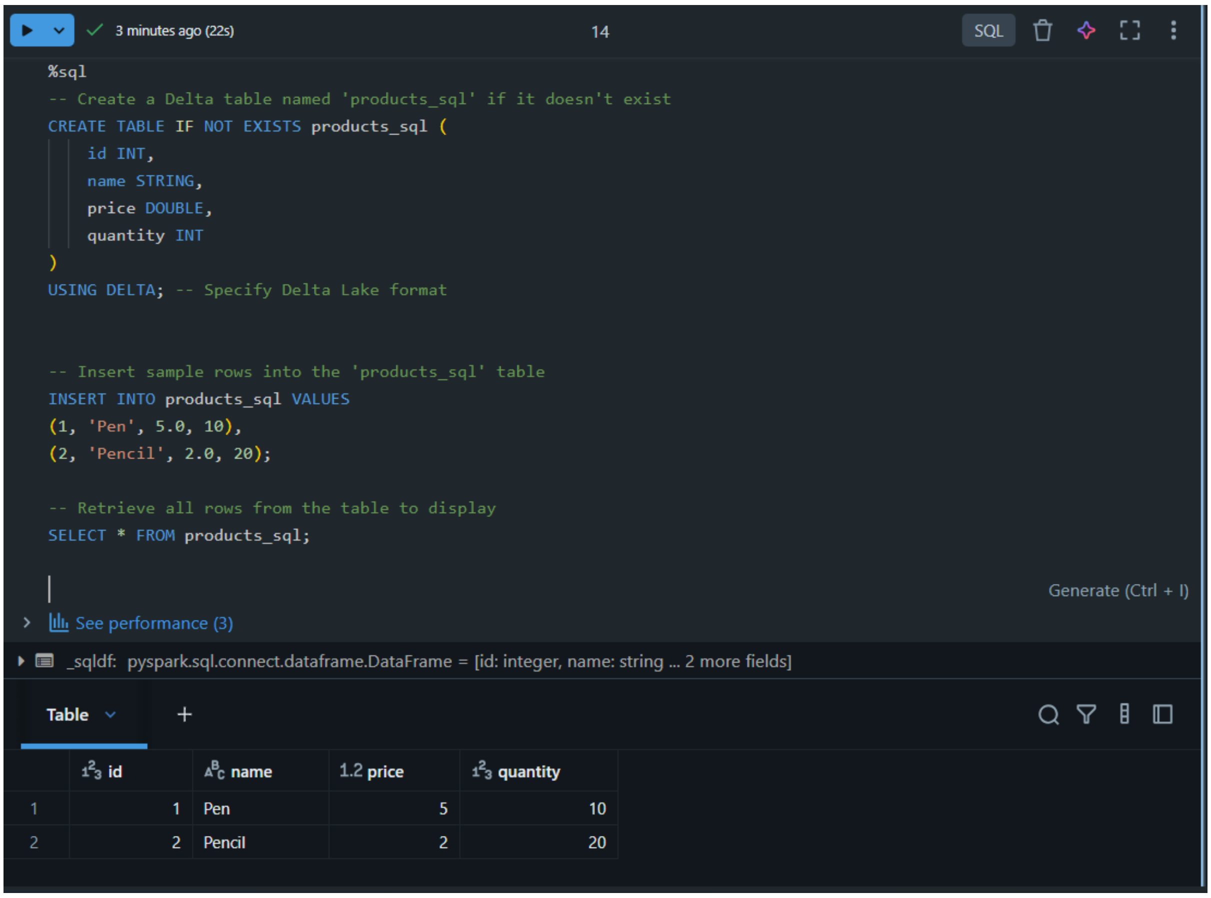
11.8.25 – Assignment

Creating Delta Tables: 3 Methods

Method 1: Create Delta Table Using SQL

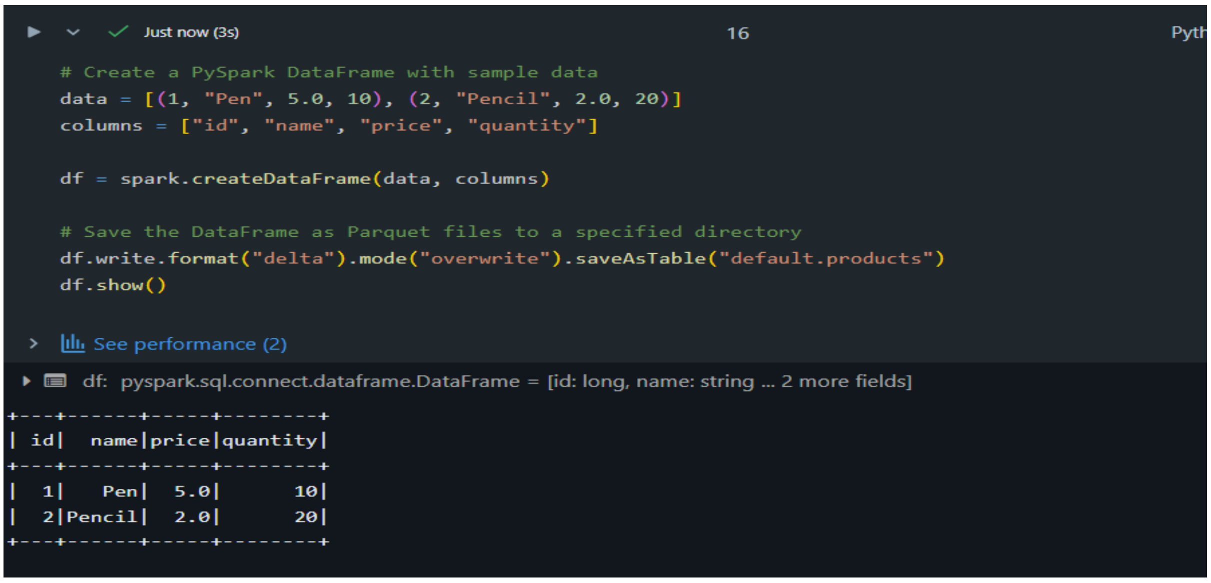


```
%sql
-- Create a Delta table named 'products_sql' if it doesn't exist
CREATE TABLE IF NOT EXISTS products_sql (
    id INT,
    name STRING,
    price DOUBLE,
    quantity INT
)
USING DELTA; -- Specify Delta Lake format


-- Insert sample rows into the 'products_sql' table
INSERT INTO products_sql VALUES
(1, 'Pen', 5.0, 10),
(2, 'Pencil', 2.0, 20);

-- Retrieve all rows from the table to display
SELECT * FROM products_sql;
```

```
_sqldf: pyspark.sql.connect.dataframe.DataFrame = [id: integer, name: string ... 2 more fields]
```

	id	name	price	quantity
1	1	Pen	5	10
2	2	Pencil	2	20

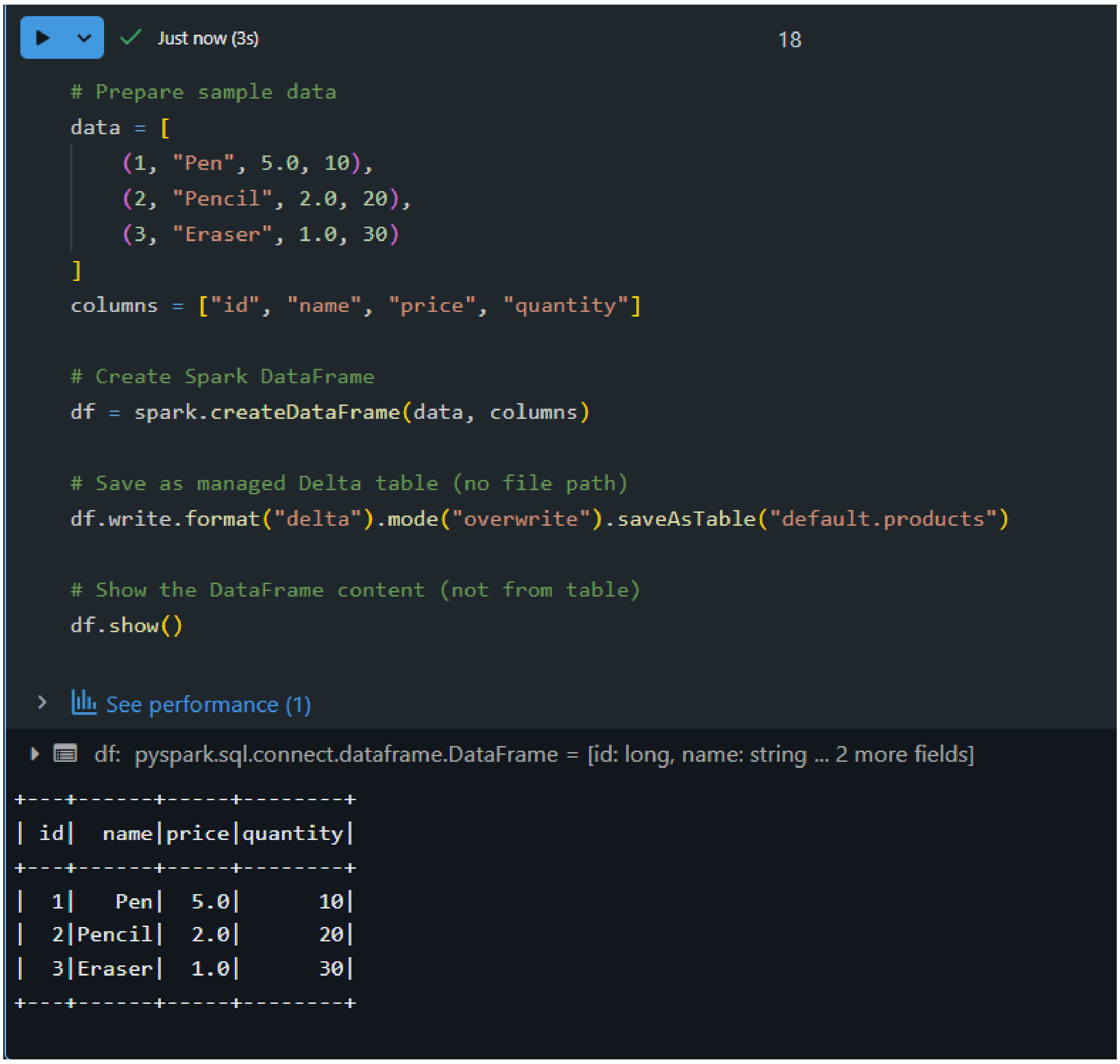
Method 2: Convert Existing Parquet Data to Delta Table



```
# Create a PySpark DataFrame with sample data
data = [(1, "Pen", 5.0, 10), (2, "Pencil", 2.0, 20)]
columns = ["id", "name", "price", "quantity"]

df = spark.createDataFrame(data, columns)

# Save the DataFrame as Parquet files to a specified directory
df.write.format("delta").mode("overwrite").saveAsTable("default.products")
df.show()
```

```
df: pyspark.sql.connect.dataframe.DataFrame = [id: long, name: string ... 2 more fields]
+---+------+-----+--------+
| id|  name|price|quantity|
+---+------+-----+--------+
|  1|   Pen|  5.0|      10|
|  2|Pencil|  2.0|      20|
+---+------+-----+--------+
```

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


Just now (3s)

18

```python
# Prepare sample data
data = [
    (1, "Pen", 5.0, 10),
    (2, "Pencil", 2.0, 20),
    (3, "Eraser", 1.0, 30)
]
columns = ["id", "name", "price", "quantity"]

# Create Spark DataFrame
df = spark.createDataFrame(data, columns)

# Save as managed Delta table (no file path)
df.write.format("delta").mode("overwrite").saveAsTable("default.products")

# Show the DataFrame content (not from table)
df.show()
```

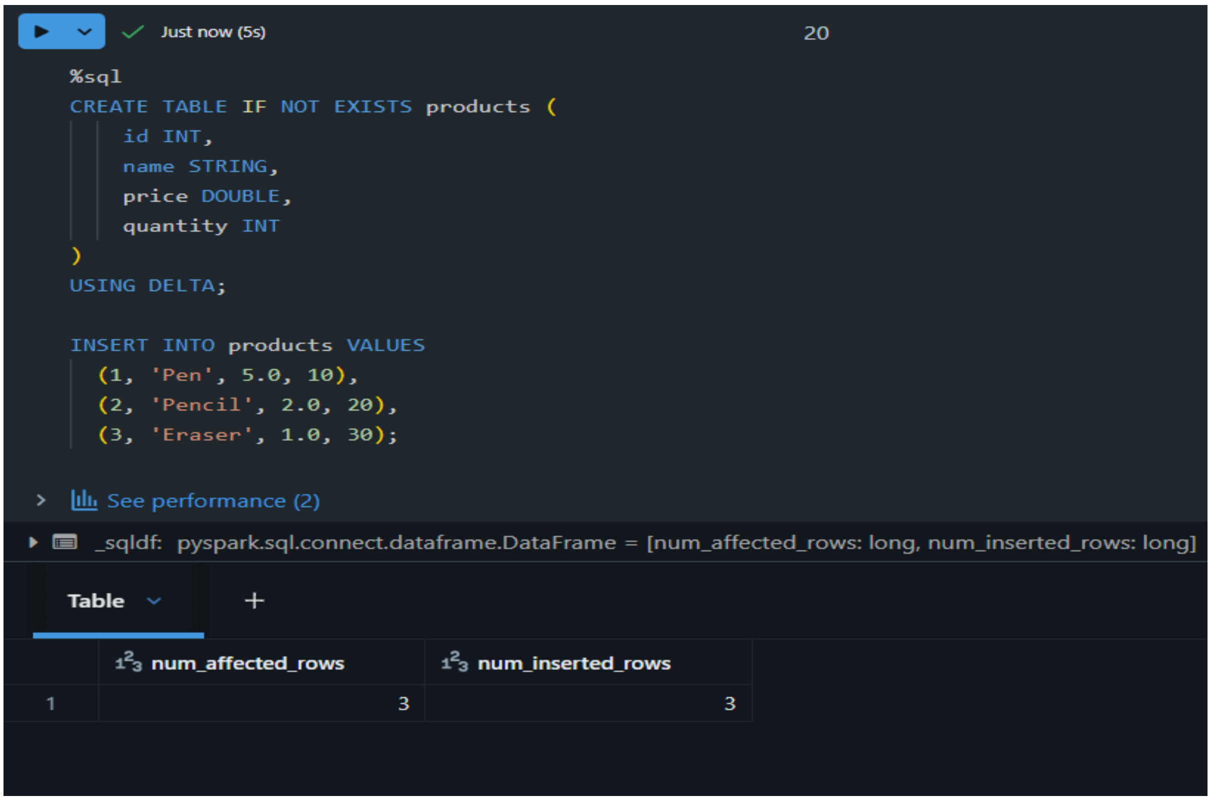
See performance (1)

df: pyspark.sql.connect.dataframe.DataFrame = [id: long, name: string ... 2 more fields]

```
+---+------+-----+--------+
| id|  name|price|quantity|
+---+------+-----+--------+
|  1|   Pen|  5.0|      10|
|  2|Pencil|  2.0|      20|
|  3|Eraser|  1.0|      30|
+---+------+-----+--------+
```

Delta Lake Merge & Upsert (SCD)

Create initial Delta table (existing data)

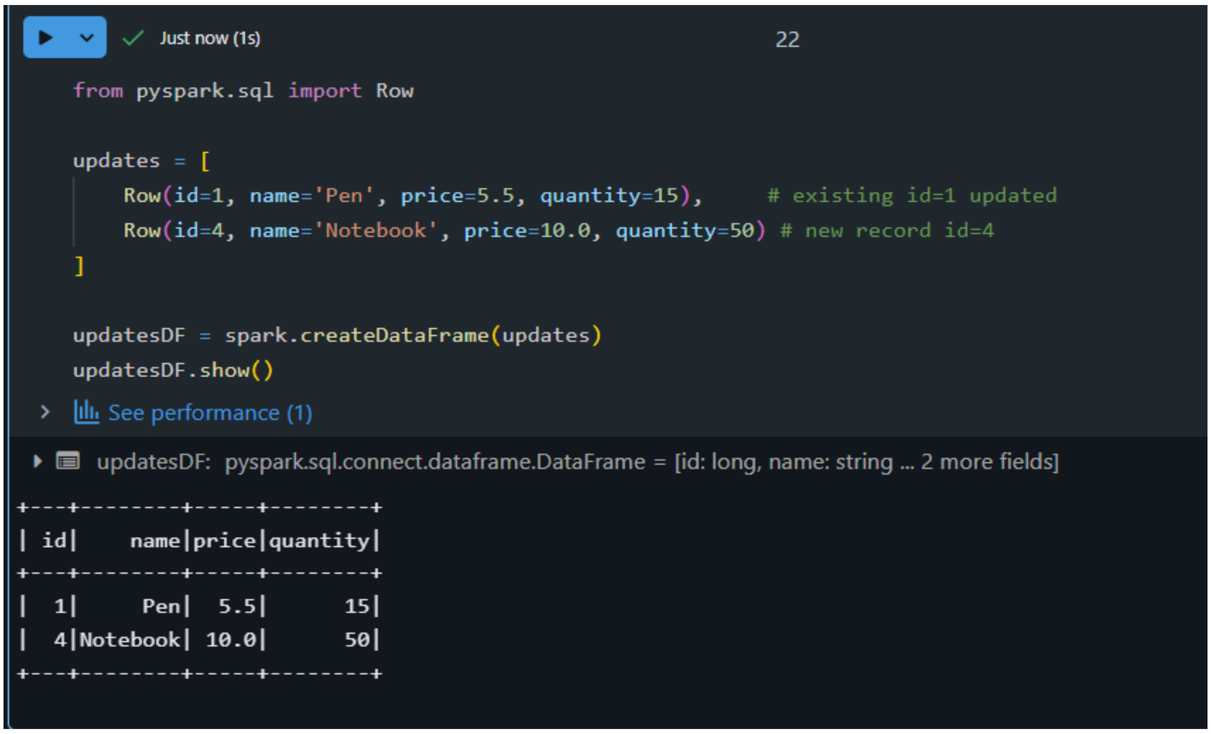


```
%sql
CREATE TABLE IF NOT EXISTS products (
    id INT,
    name STRING,
    price DOUBLE,
    quantity INT
)
USING DELTA;

INSERT INTO products VALUES
  (1, 'Pen', 5.0, 10),
  (2, 'Pencil', 2.0, 20),
  (3, 'Eraser', 1.0, 30);
```

```
_sqldf: pyspark.sql.connect.dataframe.DataFrame = [num_affected_rows: long, num_inserted_rows: long]
```

	num_affected_rows	num_inserted_rows
1	3	3

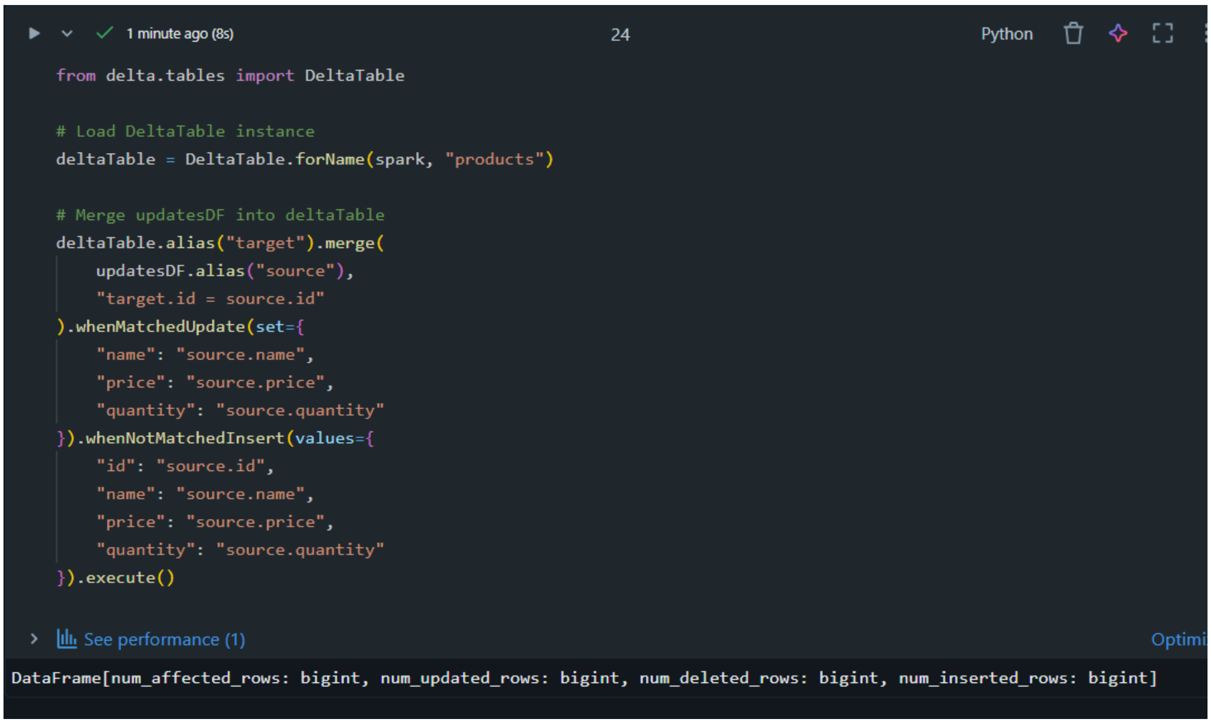
Prepare incoming updates as a DataFrame (new & updated records)



```
from pyspark.sql import Row

updates = [
    Row(id=1, name='Pen', price=5.5, quantity=15),      # existing id=1 updated
    Row(id=4, name='Notebook', price=10.0, quantity=50) # new record id=4
]

updatesDF = spark.createDataFrame(updates)
updatesDF.show()
```

```
updatesDF: pyspark.sql.connect.dataframe.DataFrame = [id: long, name: string ... 2 more fields]

+---+--------+-----+--------+
| id|    name|price|quantity|
+---+--------+-----+--------+
|  1|     Pen|  5.5|      15|
|  4|Notebook| 10.0|      50|
+---+--------+-----+--------+
```

Use Delta Table.merge to upsert



```python
from delta.tables import DeltaTable

# Load DeltaTable instance
deltaTable = DeltaTable.forName(spark, "products")

# Merge updatesDF into deltaTable
deltaTable.alias("target").merge(
    updatesDF.alias("source"),
    "target.id = source.id"
).whenMatchedUpdate(set={
    "name": "source.name",
    "price": "source.price",
    "quantity": "source.quantity"
}).whenNotMatchedInsert(values={
    "id": "source.id",
    "name": "source.name",
    "price": "source.price",
    "quantity": "source.quantity"
}).execute()
```

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

Internals of a Delta Table

Internals of a Delta Table

Delta Lake is an open-source storage layer that brings ACID transactions, scalable metadata handling, and unified batch & streaming data processing on top of existing data lakes (like Parquet files on cloud storage).

1. Storage Format

- Data Files: Actual data is stored as Parquet files on distributed storage (e.g., S3, Azure Blob Storage, HDFS).
- Transaction Log: Delta Lake maintains a transaction log which records all changes (writes, updates, deletes) as a sequence of JSON files.
- The transaction log ensures ACID compliance and consistent reads.

2. Transaction Log (_delta_log folder)

- Located at the root of the Delta table in the _delta_log directory.
- Contains ordered JSON files (e.g., 00000000000000000000.json, 00000000000000000001.json, etc.) — each represents a commit or transaction.
- Log files record:
  - Files added or removed.
  - Metadata changes (schema updates, partitions).
  - Operation type (write, delete, update).
  - Transaction timestamps.
- Periodically, a checkpoint Parquet file (e.g., 00000000000000001000.checkpoint.parquet) is created to optimize log replay during reads.

3. Metadata

- Stored in the transaction log and includes:
  - Table schema (column names, data types).
  - Partitioning information.
  - Table properties and configurations.
- Supports schema enforcement and schema evolution.

4. Data Consistency & Isolation

- Delta Lake provides ACID Transactions:
  - Atomicity: Entire transactions succeed or fail as a whole.
  - Consistency: Schema and data remain consistent after transactions.
  - Isolation: Concurrent transactions do not interfere.

    - Durability: Committed data is safely stored.
- The transaction log enables snapshot isolation — consistent reads of a stable version of the data.

5. Versioning and Time Travel

- Every transaction increments the table version.
- Delta Lake can query previous versions of data by reading the transaction log up to a chosen version.
- Enables time travel queries to access historical data snapshots or rollbacks.

6. Compaction & Cleanup

- Over time, many small Parquet files accumulate, impacting performance.
- Delta Lake supports:
    - OPTIMIZE: Compacts small files into larger ones for faster reads.
    - VACUUM: Cleans up obsolete files that are no longer referenced by the transaction log, freeing storage space.