Exercise 1: Creating a Complete ETL Pipeline using Delta Live Tables (DLT)

Sample Transactional Data (transaction_data.csv)

TransactionID	TransactionDate	CustomerID	Product	Quantity	Price
1	2024-09-01	C001	Laptop	1	1200
2	2024-09-02	C002	Tablet	2	300
3	2024-09-03	C001	Headphones	5	50
4	2024-09-04	C003	Smartphone	1	800
5	2024-09-05	C004	Smartwatch	3	200

```
dbutils.fs.cp(“file:/Workspace/Shared/transaction_data.csv”,”dbfs:/FileStore/transaction_data.csv”)
```

1. **Create Delta Live Table (DLT) Pipeline**
   - Create a cluster
   - Explore the source data
   - Ingest the raw data
   - Prepare the raw data
   - Query the transformed data
   - Create a job to run the pipeline

2. **Write DLT in Python**

```
import dlt

from pyspark.sql.functions import col


- - Raw Transactions Table: Read data from the CSV file.

@dlt.table(

  comment="Raw transactions ingested from CSV"

)

def raw_transactions():

    return spark.read.format("csv") \

        .option("header", "true") \

        .option("inferSchema", "true") \

        .load("dbfs:/FileStore/transaction_data.csv")
```

- - Transformed Transactions Table: Apply transformations

```
def transformed_transactions():
    return (
        dlt.read("raw_transactions")
        .withColumn("TotalAmount", col("Quantity") * col("Price"))
        .filter(col("Quantity") > 0)
    )
```

3. Write DLT in SQL

```
CREATE OR REPLACE TABLE raw_transactions
COMMENT 'Raw transactions ingested from CSV'
AS SELECT *
FROM csv.`dbfs:/FileStore/transactions.csv`;

CREATE OR REPLACE TABLE transformed_transactions
COMMENT 'Transformed transactions with total amount'
AS
SELECT *,
       Quantity * Price AS TotalAmount
FROM raw_transactions
WHERE Quantity > 0;
```

4. Monitor the pipeline

i. Open the Databricks Delta Live Tables, go to Pipelines tab.
ii. Select the pipeline
iii. Monitor the status of the pipeline

Exercise 2: Delta Lake Operations - Read, Write, Update, Delete, Merge

1. Read Data from Delta Lake

PySpark:

```
df = spark.read.format("delta").load("/delta/transaction_data ")

df.show()
```

SQL:

```
SELECT * FROM delta.`/delta/transaction_data `

LIMIT 5;
```

2. Write Data to Delta Lake

```
new_transactions = [

    (6, "2024-09-06", "C005", "Keyboard", 4, 100),

    (7, "2024-09-07", "C006", "Mouse", 10, 20)

]

new_df = spark.createDataFrame(new_transactions, ["TransactionID", "TransactionDate",
       "CustomerID", "Product", "Quantity", "Price"])

new_df.write.format("delta").mode("append").save("/delta/transaction_data")
```

3. Update Data in Delta Lake

```
from delta.tables import *

delta_table = DeltaTable.forPath(spark, "/delta/transaction_data")

delta_table.update(

  condition="Product = 'Laptop'",

  set={"Price": "1300"}

)
```

4. Delete Data from Delta Lake

PySpark:

```
delta_table.delete(condition="Quantity < 3")
```

SQL:

```
DELETE FROM delta.`/delta/transaction_data`
WHERE Quantity < 3;
```

5. Merge Data into Delta Lake

```
updates = [
    (1, "2024-09-01", "C001", "Laptop", 1, 1250),
    (8, "2024-09-08", "C007", "Charger", 2, 30)
]
updates_df = spark.createDataFrame(updates, ["TransactionID", "TransactionDate",
                  "CustomerID", "Product", "Quantity", "Price"])

delta_table.alias("target").merge(updates_df.alias("source"),
                              "target.TransactionID = source.TransactionID")\
                              .whenMatchedUpdateAll()\
                              .whenNotMatchedInsertAll().execute()
```

Exercise 3: Delta Lake - History, Time Travel, and Vacuum

1. View Delta Table History

PySpark:

```
from delta.tables import *

delta_table = DeltaTable.forPath(spark, "/path/to/delta/table")

history_df = delta_table.history()

history_df.show(truncate=False)
```

SQL:

```
DESCRIBE HISTORY delta.`/delta/transaction_data`;
```

2. **Perform Time Travel**

- - Retrieve the state of the Delta table as it was 5 versions ago.

```
df_time_travel =
    spark.read.format("delta").option("versionAsOf",5).load("/delta/transaction_data")

df_time_travel.show()
```

- - Verify that the table reflects the data before some of the updates and deletions made earlier.

```
df_time_travel = spark.read.format("delta").option("versionAsOf", 5).load("/delta/transaction_data")

df_time_travel.show()
```

- - Perform a query to get the transactions from a specific timestamp

```
df_time_travel = spark.read.format("delta").option("timestampAsOf", "2024-09-
    22T14:00:00Z").load("/delta/transaction_data")

df_time_travel.show()
```

3. **Vacuum the Delta Table**

```
delta_table.vacuum(retentionHours=168)
```

4. **Converting Parquet Files to Delta Files**

- - Create a new Parquet-based table from the raw transactions CSV file.

```
parquet_df = spark.read.format("csv").option("header", "true").option("inferSchema",
    "true").load("dbfs:/FileStore/transaction_data.csv")

parquet_df.write.format("parquet").save("/FileStore/parquet/transaction_table")
```

- - Convert this Parquet table to a Delta table using Delta Lake functionality.

```
spark.read.format("parquet").load("/FileStore/parquet/transaction_table").write.format("delta")\
                                                  .save("/delta/parquet_converted_table")
```

Exercise 4: Implementing Incremental Load Pattern using Delta Lake

1. **Set Up Initial Data**

```
initial_transactions = [
    (1, "2024-09-01", "C001", "Laptop", 1, 1200),
    (2, "2024-09-02", "C002", "Tablet", 2, 300),
    (3, "2024-09-03", "C001", "Headphones", 5, 50)
]
initial_df = spark.createDataFrame(initial_transactions, ["TransactionID", "TransactionDate",
       "CustomerID", "Product", "Quantity", "Price"])
initial_df.write.format("delta").mode("overwrite").save("/delta/transaction_data")
```

2. **Set Up Incremental Data**

```
incremental_transactions = [
    (4, "2024-09-04", "C003", "Smartphone", 1, 800),
    (5, "2024-09-05", "C004", "Smartwatch", 3, 200),
    (6, "2024-09-06", "C005", "Keyboard", 4, 100),
    (7, "2024-09-07", "C006", "Mouse", 10, 20)
]
incremental_df = spark.createDataFrame(incremental_transactions, ["TransactionID",
       "TransactionDate", "CustomerID", "Product", "Quantity", "Price"])
```

3. **Implement Incremental Load**

```
delta_table = DeltaTable.forPath(spark, "/delta/transaction_data")
max_date = delta_table.toDF().agg({"TransactionDate": "max"}).collect()[0][0]
new_transactions = incremental_df.filter(incremental_df.TransactionDate > max_date)
new_transactions.write.format("delta").mode("append").save("/delta/transaction_data")
```

4. **Monitor Incremental Load**

```
history_df = delta_table.history()
history_df.show(truncate=False)
```