Task 1: Vehicle Maintenance Data Ingestion

Use the following CSV data representing vehicle maintenance records:

```
VehicleID,Date,ServiceType,ServiceCost,Mileage
V001,2024-04-01,Oil Change,50.00,15000
V002,2024-04-05,Tire Replacement,400.00,30000
V003,2024-04-10,Battery Replacement,120.00,25000
V004,2024-04-15,Brake Inspection,200.00,40000
V005,2024-04-20,Oil Change,50.00,18000
```

Ingest this CSV data into a Delta table in Databricks.
Add error handling for cases where the file is missing or contains incorrect data, and log any such issues.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Create a Spark session
spark = SparkSession.builder.appName("vehicle_maintenance").getOrCreate()

# Reading CSV data
try:
    maintenance_df =
spark.read.csv("dbfs:/FileStore/vehicle_maintenance/vehicle_maintenance.csv"
, header=True, inferSchema=True)
    maintenance_df.show()
except Exception as e:
    print(f"Error reading CSV file: {e}")

# Write data to Delta table
try:

maintenance_df.write.format("delta").mode("overwrite").save("/delta/vehicle_
maintenance")
except Exception as e:
    print(f"Error writing to Delta table: {e}")
```

Task 2: Data Cleaning

Clean the vehicle maintenance data:

- Ensure that the ServiceCost and Mileage columns contain valid positive values.
- Remove any duplicate records based on VehicleID and Date .

Save the cleaned data to a new Delta table.

```
# Filter out rows with non-positive values in ServiceCost or Mileage
cleaned_df = maintenance_df.filter((col("ServiceCost") > 0) &
(col("Mileage") > 0))
cleaned_df = cleaned_df.dropDuplicates(["VehicleID", "Date"])
cleaned_df.write.format("delta").mode("overwrite").save("/delta/cleaned_vehicle_maintenance")
```

Task 3: Vehicle Maintenance Analysis

Create a notebook to analyze the vehicle maintenance data:

Calculate the total maintenance cost for each vehicle.

```
total_cost_df =
cleaned_df.groupBy("VehicleID").sum("ServiceCost").withColumnRenamed("sum(ServiceCost)", "TotalMaintenanceCost")
total_cost_df.show()
```

Identify vehicles that have exceeded a certain mileage threshold (e.g., 30,000 miles) and might need additional services.

```
high_mileage_df = cleaned_df.filter(col("Mileage") > 30000)
high_mileage_df.show()
```

Save the analysis results to a Delta table.

```
total_cost_df.write.format("delta").mode("overwrite").save("/delta/vehicle_maintenance_cost")
high_mileage_df.write.format("delta").mode("overwrite").save("/delta/vehicle_high_mileage")
```

Task 5: Data Governance with Delta Lake

Enable Delta Lake's data governance features:

Use VACUUM to clean up old data from the Delta table.

```
spark.sql("VACUUM '/delta/vehicle_maintenance' RETAIN 0 HOURS")
```

Use DESCRIBE HISTORY to check the history of updates to the maintenance records.

```
spark.sql("DESCRIBE HISTORY
'/delta/vehicle_maintenance'").show(truncate=False)
```

Task 1: Movie Ratings Data Ingestion

Use the following CSV data to represent movie ratings by users:

```
UserID,MovieID,Rating,Timestamp
U001,M001,4,2024-05-01 14:30:00
U002,M002,5,2024-05-01 16:00:00
U003,M001,3,2024-05-02 10:15:00
```

U001,M003,2,2024-05-02 13:45:00
U004,M002,4,2024-05-03 18:30:00

Ingest this CSV data into a Delta table in Databricks.
Ensure proper error handling for missing or inconsistent data, and log errors accordingly.

```
# Reading CSV data
try:
    ratings_df = spark.read.csv("dbfs:/FileStore/movies/ratings.csv", header=True, inferSchema=True)
    ratings_df.show()
except Exception as e:
    print(f"Error reading CSV file: {e}")

# Write data to Delta table
try:
    ratings_df.write.format("delta").mode("overwrite").save("/delta/movie_ratings")
except Exception as e:
    print(f"Error writing to Delta table: {e}")
```

Task 2: Data Cleaning

Clean the movie ratings data:

Ensure that the Rating column contains values between 1 and 5. Remove any duplicate entries (same UserID and MovieID).

```
cleaned_ratings_df = ratings_df.filter((col("Rating") >= 1) & (col("Rating") <= 5))
cleaned_ratings_df = cleaned_ratings_df.dropDuplicates(["UserID", "MovieID"])
```

Save the cleaned data to a new Delta table.

```
cleaned_ratings_df.write.format("delta").mode("overwrite").save("/delta/cleaned_movie_ratings")
```

Task 3: Movie Rating Analysis

Create a notebook to analyze the movie ratings:

Calculate the average rating for each movie.

```
avg_ratings_df = cleaned_ratings_df.groupBy("MovieID").avg("Rating").withColumnRenamed("avg(Rating)", "AverageRating")
```

```
avg_ratings_df.show()
```

Identify the movies with the highest and lowest average ratings. Save the analysis results to a Delta table.

```
highest_rating_df =
avg_ratings_df.orderBy(col("AverageRating").desc()).limit(1)
lowest_rating_df =
avg_ratings_df.orderBy(col("AverageRating").asc()).limit(1)

highest_rating_df.show()
lowest_rating_df.show()
```

Task 4: Time Travel and Delta Lake History

Implement Delta Lake's time travel feature:

Perform an update to the movie ratings data (e.g., change a few ratings).

```
updated_ratings_df = cleaned_ratings_df.withColumn("Rating",
    when((col("UserID") == "U001") & (col("MovieID") == "M001"),
5).otherwise(col("Rating")))

updated_ratings_df.write.format("delta").mode("overwrite").save("/del
ta/cleaned_movie_ratings")
```

Roll back to a previous version of the Delta table to retrieve the original ratings.

```
previous_version_df =
spark.read.format("delta").option("versionAsOf",
0).load("/delta/cleaned_movie_ratings")
previous_version_df.show()
```

Use DESCRIBE HISTORY to view the history of changes to the Delta table.

```
spark.sql("DESCRIBE HISTORY
'/delta/cleaned_movie_ratings'").show(truncate=False)
```

Task 5: Optimize Delta Table

Apply optimizations to the Delta table:

Implement Z-ordering on the MovieID column to improve query performance.

Use the OPTIMIZE command to compact the data and improve performance. Use VACUUM to clean up older versions of the table.

```
spark.sql("OPTIMIZE '/delta/cleaned_movie_ratings' ZORDER BY (MovieID)")
spark.sql("OPTIMIZE '/delta/cleaned_movie_ratings'")
spark.sql("VACUUM '/delta/cleaned_movie_ratings' RETAIN 0 HOURS")
```

Task 1: Data Ingestion - Reading Data from Various Formats 1. **Ingest data from different formats** (CSV, JSON, Parquet, Delta table): **CSV Data**: Use the following CSV data to represent student information:

```
StudentID,Name,Class,Score
S001,Anil Kumar,10,85
S002,Neha Sharma,12,92
S003,Rajesh Gupta,11,78
```

JSON Data: Use the following JSON data to represent city information:

```
[
  {"CityID": "C001", "CityName": "Mumbai", "Population": 20411000}, {"CityID": "C002", "CityName": "Delhi", "Population": 16787941}, {"CityID": "C003", "CityName": "Bangalore", "Population": 8443675} ]
```

Parquet Data: Use a dataset containing data about hospitals stored in Parquet format. Write code to load this data into a DataFrame. **Delta Table**: Load a Delta table containing hospital records, ensuring you include proper error handling in case the table does not exist.

```
# Reading CSV data

student_df = spark.read.csv("dbfs:/FileStore/students/student_data.csv", header=True, inferSchema=True)

student_df.show()

# Reading JSON data

city_df = spark.read.json("dbfs:/FileStore/cities/city_data.json")

city_df.show()

# Reading Parquet data

hospital_df = spark.read.parquet("dbfs:/FileStore/hospitals/hospital_data.parquet")

hospital_df.show()

# Reading Delta table with error handling

try:

    delta_hospital_df = spark.read.format("delta").load("/delta/hospital_records")

    delta_hospital_df.show()
```

```
except Exception as e:

    print(f"Error loading Delta table: {e}")
```

Task 2: Writing Data to Various Formats

1. **Write data from the following DataFrames to different formats**:
   CSV: Write the student data (from Task 1) to a CSV file.
   JSON: Write the city data (from Task 1) to a JSON file.
   Parquet: Write the hospital data (from Task 1) to a Parquet file. **Delta Table**: Write the hospital data to a Delta table.

```
# Writing student data to CSV
student_df.write.csv("/output/students_data.csv", mode="overwrite",
header=True)
# Writing city data to JSON
city_df.write.json("/output/cities_data.json", mode="overwrite")
# Writing hospital data to Parquet
hospital_df.write.parquet("/output/hospitals_data.parquet",
mode="overwrite")
# Writing hospital data to Delta table
hospital_df.write.format("delta").mode("overwrite").save("/delta/hospital_da
ta_delta")
```

Task 3: Running One Notebook from Another

1. **Create two notebooks**:

   Notebook A: Ingest data from a CSV file, clean the data (remove duplicates, handle missing values), and save it as a Delta table. Notebook B: Perform analysis on the Delta table created in Notebook A (e.g., calculate the average score of students) and write the results to a new Delta table.

```
# Reading CSV and cleaning data

student_df = spark.read.csv("dbfs:/FileStore/students/student_data.csv",
header=True, inferSchema=True)

cleaned_student_df = student_df.dropDuplicates().na.fill({"Score": 0})



# Writing cleaned data to Delta table

cleaned_student_df.write.format("delta").mode("overwrite").save("/delta/clea
ned_students")
```

2. **Run Notebook B from Notebook A**:

```python
# Reading cleaned data from Delta table
cleaned_student_df =
spark.read.format("delta").load("/delta/cleaned_students")


# Calculating average score
avg_score_df =
cleaned_student_df.groupBy("Class").avg("Score").withColumnRenamed("avg(Score)", "AverageScore")
avg_score_df.show()


# Writing results to a new Delta table
avg_score_df.write.format("delta").mode("overwrite").save("/delta/student_avg_scores")
```

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

Objective:

Learn how to create an end-to-end ETL pipeline using Delta Live Tables.

Tasks:

1. **Create Delta Live Table (DLT) Pipeline**:

   Set up a DLT pipeline for processing transactional data. Use sample data representing daily customer transactions.
   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

   Define the pipeline steps:
   Step 1: Ingest raw data from CSV files.
   Step 2: Apply transformations (e.g., calculate total transaction amount).
   Step 3: Write the final data into a Delta table.

2. **Write DLT in Python**:

   Implement the pipeline using **DLT in Python**. Define the following tables: **Raw Transactions Table**: Read data from the CSV file.
   Transformed Transactions Table: Apply transformations (e.g., calculate total amount: `Quantity * Price`).

```python
import dlt
```

```python
from pyspark.sql.functions import col

# Step 1: Raw Transactions Table
@dlt.table
def raw_transactions():
    return (spark.read.option("header",
"true").csv("dbfs:/FileStore/transactions/transaction_data.csv"))

# Step 2: Transformed Transactions Table
@dlt.table
def transformed_transactions():
    return (dlt.read("raw_transactions")
                .withColumn("TotalAmount", col("Quantity") * col("Price")))

# Step 3: Writing Transformed Data to Delta Table
@dlt.table
def final_table():
    return dlt.read("transformed_transactions")
```

3. **Write DLT in SQL**:

   Implement the same pipeline using **DLT in SQL**. Use SQL syntax to define tables, transformations, and outputs.

```sql
%sql
-- Step 1: Raw Transactions Table

CREATE OR REPLACE LIVE TABLE raw_transactions

AS SELECT * FROM csv.`dbfs:/FileStore/transactions/transaction_data.csv`;



-- Step 2: Transformed Transactions Table

CREATE OR REPLACE LIVE TABLE transformed_transactions

AS SELECT *, Quantity * Price AS TotalAmount FROM LIVE.raw_transactions;
```

```
-- Step 3: Final Table

CREATE OR REPLACE LIVE TABLE final_table

AS SELECT * FROM LIVE.transformed_transactions;
```

4. **Monitor the Pipeline**:

   Use Databricks' DLT UI to monitor the pipeline and check the status of each step.

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

Objective:

Work with Delta Lake to perform read, write, update, delete, and merge operations using both PySpark and SQL.

Tasks:

1. **Read Data from Delta Lake**:

   Read the transactional data from the Delta table you created in the first exercise using PySpark and SQL.
   Verify the contents of the table by displaying the first 5 rows.

```
# Reading from Delta table
df = spark.read.format("delta").load("/delta/final_table")
df.show(5)
```

```
%sql

SELECT * FROM delta.`/delta/final_table` LIMIT 5;
```

2. **Write Data to Delta Lake**:

   Append new transactions to the Delta table using PySpark.
   Example new transactions:

   6,2024-09-06,C005,Keyboard,4,100
   7,2024-09-07,C006,Mouse,10,20

```
new_data = [(6, "2024-09-06", "C005", "Keyboard", 4, 100),
            (7, "2024-09-07", "C006", "Mouse", 10, 20)]


new_df = spark.createDataFrame(new_data, schema=["TransactionID",
"TransactionDate", "CustomerID", "Product", "Quantity", "Price"])
new_df.write.format("delta").mode("append").save("/delta/final_table")
```

3. **Update Data in Delta Lake**:

   Update the Price of Product = 'Laptop' to 1300 .

Use PySpark or SQL to perform the update and verify the results.

```python
# Updating product price for 'Laptop'
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, "/delta/final_table")
deltaTable.update(
    condition="Product = 'Laptop'",
    set={"Price": "1300"}
)
```

```sql
UPDATE delta.`/delta/final_table`
SET Price = 1300
WHERE Product = 'Laptop';
```

4. **Delete Data from Delta Lake**:

    Delete all transactions where the Quantity is less than 3.
    Use both PySpark and SQL to perform this deletion.

```python
# Deleting rows where Quantity < 3
deltaTable.delete(condition="Quantity < 3")
```

```sql
DELETE FROM delta.`/delta/final_table`
WHERE Quantity < 3;
```

5. **Merge Data into Delta Lake**:

    Create a new set of data representing updates to the existing transactions. Merge the following new data into the Delta table:

    TransactionID,TransactionDate,CustomerID,Product,Quantity,Price
    1,2024-09-01,C001,Laptop,1,1250 -- Updated Price
    8,2024-09-08,C007,Charger,2,30 -- New Transaction

    Use the Delta Lake **merge** operation to insert the new data and update the existing records.

```python
merge_data = [(1, "2024-09-01", "C001", "Laptop", 1, 1250),

              (8, "2024-09-08", "C007", "Charger", 2, 30)]



merge_df = spark.createDataFrame(merge_data, schema=["TransactionID",
"TransactionDate", "CustomerID", "Product", "Quantity", "Price"])
```

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

Understand how to use Delta Lake features such as versioning, time travel, and data cleanup with vacuum.

Tasks:

1. **View Delta Table History**:

    Query the **history** of the Delta table to see all changes (inserts, updates, deletes) made in the previous exercises.
    Use both PySpark and SQL to view the history.

```
# Viewing history
deltaTable.history().show(truncate=False)
```

```
DESCRIBE HISTORY delta.`/delta/final_table`;
```

2. **Perform Time Travel**:

    Retrieve the state of the Delta table as it was **5 versions ago**. Verify that the table reflects the data before some of the updates and deletions made earlier.
    Perform a query to get the transactions from a specific timestamp (e.g., just before an update).

```
# Querying table as it was 5 versions ago
df_time_travel = spark.read.format("delta").option("versionAsOf",
5).load("/delta/final_table")
df_time_travel.show()
```

```
SELECT * FROM delta.`/delta/final_table` VERSION AS OF 5;
```

3. **Vacuum the Delta Table**:

    Clean up old data using the **VACUUM** command.
    Set a retention period of 7 days and vacuum the Delta table.
    Verify that old versions are removed, but the current table state is intact.

```
# Vacuuming the Delta table (7-day retention)
spark.sql("VACUUM delta.`/delta/final_table` RETAIN 7 HOURS")
```

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

    Create a new Parquet-based table from the raw transactions CSV file. Convert this Parquet table to a Delta table using Delta Lake
    functionality.

```
# Converting Parquet to Delta
parquet_df = spark.read.parquet("/parquet/transactions")
parquet_df.write.format("delta").save("/delta/converted_transactions")
```

Exercise 4: Implementing Incremental Load Pattern using Delta Lake Objective:

Learn how to implement incremental data loading with Delta Lake to avoid reprocessing old data.

Tasks:

1. **Set Up Initial Data**:

   Use the same transactions data from previous exercises, but load only transactions from the first three days (2024-09-01 to 2024-09-03) into the Delta table.

```
initial_data = [(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_data, schema=["TransactionID",
"TransactionDate", "CustomerID", "Product", "Quantity", "Price"])

initial_df.write.format("delta").save("/delta/incremental_table")
```

2. **Set Up Incremental Data**:

   Add a new set of transactions representing the next four days (2024-09- 04 to 2024-09-07).
   Ensure that these transactions are loaded incrementally into the Delta table.

```
incremental_data = [(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_data,
schema=["TransactionID", "TransactionDate", "CustomerID", "Product",
"Quantity", "Price"])
```

3. **Implement Incremental Load**:

   Create a pipeline that reads new transactions only (transactions after 2024-09-03) and appends them to the Delta table without overwriting existing data.
   Verify that the incremental load only processes new data and does not duplicate or overwrite existing records.

```
# Incrementally loading data after 2024-09-03
incremental_df.write.format("delta").mode("append").save("/delta/incremental
_table")
```

4. **Monitor Incremental Load**:

   Check the Delta Lake version history to ensure only the new transactions are added, and no old records are reprocessed.

```
# Check version history
```

```
deltaTable.history().show(truncate=False)
```