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
import logging

logging.basicConfig(level=logging.INFO)

spark = SparkSession.builder \
    .appName("VehicleMaintenance") \
    .getOrCreate()

csv_file_path = "content/vehicle_maintenance.csv"
delta_table_path = "/delta/vehicle_maintenance"

try:
    df = spark.read.format("csv").option("header", 
"true").load(csv_file_path)
    df.write.format("delta").mode("overwrite").save(delta_table_path)
    logger.info("Data ingested successfully.")
except FileNotFoundError:
    logger.error("CSV file not found at path: %s", csv_file_path)
```

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.

```
Df = spark.read.format("delta").load(delta_table_path)

cleaned_df = df.filter((df['Cost'] > 0) & (df['Mileage'] 
> 0)).dropDuplicates(['VehicleID', 'Date'])

cleaned_delta_table_path = 
```

```
"/delta/cleaned_vehicle_maintenance"
cleaned_df.write.format("delta").mode("overwrite").save(c
leaned_delta_table_path)
logger.info("Cleaned data saved to Delta table.")
```

Task 3: Vehicle Maintenance Analysis

- Create a notebook to analyze the vehicle maintenance data:
  - Calculate the total maintenance cost for each vehicle.
  - Identify vehicles that have exceeded a certain mileage threshold (e.g., 30,000 miles) and might need additional services.
  - Save the analysis results to a Delta table.

```
cleaned_df = spark.read.format("delta").load(cleaned_delta_table_path)

maintenance cost df = cleaned df.groupBy("VehicleID").agg({"Cost":
"sum"}).withColumnRenamed("sum(Cost)", "TotalCost")

mileage threshold = 30000
vehicles needing service df = cleaned df.filter(cleaned_df['Mileage'] >
mileage_threshold).select("VehicleID").distinct()

analysis delta table path = "/delta/maintenance analysis"
maintenance cost_df.write.format("delta").mode("overwrite").save(analysis_de
lta_table_path)

vehicles needing service df.write.format("delta").mode("overwrite").save("db
fs:/delta/vehicles needing service")
logger.info("Analysis results saved to Delta tables.")
```

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

```
spark.sql(f"VACUUM '{delta_table_path}' RETAIN 168 HOURS")


history_df = spark.sql(f"DESCRIBE HISTORY
delta.`{delta_table_path}`")
history_df.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.

```
from pyspark.sql import SparkSession
import logging
```

```
logging.basicConfig(level=logging.INFO)

spark = SparkSession.builder \
    .appName("MovieRatings") \
    .getOrCreate()

csv_file_path = "content/movie_ratings.csv"
delta_table_path = "delta/movie_ratings"

try:
    df = spark.read.format("csv").option("header", 
"true").load(csv_file_path)
    df.write.format("delta").mode("overwrite").save(delta_table_path)
    logger.info("Data ingested successfully.")
except FileNotFoundError:
    logger.error("CSV file not found at path: %s", csv_file_path)
```

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`).
  - Save the cleaned data to a new Delta table.

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

cleaned_df = df.filter((df['Rating'] >= 1) & (df['Rating'] <=
5)).dropDuplicates(['UserID', 'MovieID'])

cleaned_delta_table_path = "/delta/cleaned_movie_ratings"
cleaned_df.write.format("delta").mode("overwrite").save(cleaned_delta_table_
path)
logger.info("Cleaned data saved to Delta table.")
```

Task 3: Movie Rating Analysis

- Create a notebook to analyze the movie ratings:
  - Calculate the average rating for each movie.
  - Identify the movies with the highest and lowest average ratings.
  - Save the analysis results to a Delta table.

```
cleaned_df =
spark.read.format("delta").load(cleaned_delta_table_path)

average_rating_df =
cleaned_df.groupBy("MovieID").agg({"Rating":
"avg"}).withColumnRenamed("avg(Rating)", "AverageRating")

highest_rating_df =
average_rating_df.orderBy("AverageRating",
ascending=False).limit(1)
lowest_rating_df =
average_rating_df.orderBy("AverageRating",
ascending=True).limit(1)

analysis_delta_table_path = "/delta/movie_rating_analysis"
average_rating_df.write.format("delta").mode("overwrite").
save(analysis_delta_table_path)

highest_rating_df.write.format("delta").mode("overwrite").
save("dbfs:/delta/highest_rating")
lowest_rating_df.write.format("delta").mode("overwrite").s
ave("dbfs:/delta/lowest_rating")
logger.info("Analysis results saved to Delta tables.")
```

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).
  - Roll back to a previous version of the Delta table to retrieve the original ratings.
  - Use `DESCRIBE HISTORY` to view the history of changes to the Delta table.

```
updated_df = cleaned_df.withColumn("Rating", when(cleaned_df['MovieID'] ==
"12", 4.5).otherwise(cleaned_df['Rating']))
updated_df.write.format("delta").mode("overwrite").save(delta_table_path)
logger.info("Movie ratings updated.")

original_df = spark.read.format("delta").option("versionAsOf",
0).load(delta_table_path)
```

```
original_df.write.format("delta").mode("overwrite").save(delta_table_path)
logger.info("Rolled back to the original version of movie ratings.")

history_df = spark.sql(f"DESCRIBE HISTORY delta.`{delta_table_path}`")
history_df.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(f"OPTIMIZE delta.`{delta_table_path}` ZORDER BY
(MovieID)")

spark.sql(f"VACUUM delta.`{delta_table_path}` RETAIN 168 HOURS")
logger.info("Delta table optimized and old versions cleaned
up.")
```

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.

```
from pyspark.sql import SparkSession
import logging

logging.basicConfig(level=logging.INFO)

spark = SparkSession.builder \
    .appName("DataIngestion") \
    .getOrCreate()

csv_file_path = "/content/student_data.csv"
student_df = spark.read.format("csv").option("header",
"true").load(csv_file_path)
logger.info("Student data ingested successfully.")
```

```
json_file_path = "/content/city_data.json"
city_df = spark.read.format("json").load(json_file_path)
logger.info("City data ingested successfully.")

parquet_file_path = "/content/hospital_data.parquet"
hospital_parquet_df =
spark.read.format("parquet").load(parquet_file_path)
logger.info("Hospital data ingested successfully from
Parquet.")

delta_table_path = "/delta/hospital_records"

try:
    hospital_delta_df =
spark.read.format("delta").load(delta_table_path)
    logger.info("Hospital records loaded from Delta table.")
except FileNotFoundException:
    logger.error("Delta table does not exist at path: %s",
delta_table_path)
```

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.

```
student_output_path = "/output/student_data.csv"
student_df.write.format("csv").mode("overwrite").save(student_output_path)
logger.info("Student data written to CSV.")

city_output_path = "/output/city_data.json"
city_df.write.format("json").mode("overwrite").save(city_output_path)
logger.info("City data written to JSON.")

hospital_parquet_output_path = "/output/hospital_data.parquet"
hospital_parquet_df.write.format("parquet").mode("overwrite").save(hospital_
parquet_output_path)
logger.info("Hospital data written to Parquet.")

hospital_delta_output_path = "/delta/hospital_data"
hospital_parquet_df.write.format("delta").mode("overwrite").save(hospital_de
lta_output_path)
logger.info("Hospital data written to Delta table.")
```

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.
2. **Run Notebook B from Notebook A:**
   - Implement the logic to call and run Notebook B from within Notebook A.

```
student_df_cleaned = student_df.dropDuplicates().na.fill({"Score": 0})
cleaned_delta_table_path = "/delta/cleaned_student_data"
student_df_cleaned.write.format("delta").mode("overwrite").save(cleaned_delt
a_table_path)
logger.info("Cleaned student data saved as Delta table.")

cleaned_student_df = spark.read.format("delta").load(delta_table_path)

average_score_df = cleaned_student_df.groupBy("Class").agg({"Score":
"avg"}).withColumnRenamed("avg(Score)", "AverageScore")

average_score_delta_table_path = "delta/average_student_scores"
average_score_df.write.format("delta").mode("overwrite").save(average_score_
delta_table_path)
logger.info("Average student scores saved as Delta table.")
```

Task 4: Databricks Ingestion

1. **Read data from the following sources:**

- CSV file from Azure Data Lake.
- JSON file stored on Databricks FileStore.
- Parquet file from an external data source (e.g., AWS S3).
- Delta table stored in a Databricks-managed database.

```
azure_csv_path = "azure_data_lake_path_of_csv"
azure_student_df =
spark.read.format("csv").option("header",
"true").load(azure_csv_path)
logger.info("Azure CSV data ingested...")

file_store_json_path = "databricks_filestore_of_json "
file_store_city_df =
spark.read.format("json").load(file_store_json_path)
logger.info("Databricks FileStore JSON data ingested...")

parquet_path = "external_datastore_of_parquet"
hospital_df =
spark.read.format("parquet").load(s3_parquet_path)
logger.info("External Parquet data ingested ...")

managed_delta_table_path =
"databricks_managed_database_delta_table"
managed_hospital_df =
spark.read.format("delta").load(managed_delta_table_path)
logger.info("Managed Delta table data ingested...")
```

2. **Write the cleaned data** to each of the formats listed above (CSV, JSON, Parquet, and Delta) after performing some basic transformations (e.g., filtering rows, calculating totals).

```
filtered_hospital_df = hospital_df.filter(hospital_df['Capacity'] > 50)
filtered_hospital_df.write.format("csv").mode("overwrite").save("/output/clea
ned_hospital_data.csv")
filtered_hospital_df.write.format("json").mode("overwrite").save("/output/cle
aned_hospital_data.json")
filtered_hospital_df.write.format("parquet").mode("overwrite").save("/output/
cleaned_hospital_data.parquet")
filtered_hospital_df.write.format("delta").mode("overwrite").save("/delta/cle
aned_hospital_data")
logger.info("Cleaned data written to CSV, JSON, Parquet, and Delta.")
```

Additional Tasks:

- **Optimization Task:** Once the data is written to a Delta table, optimize it using Delta Lake's `OPTIMIZE` command.
- **Z-ordering Task:** Apply Z-ordering on the `CityName` or `Class` columns for faster querying.

  Vacuum Task: Use the `VACUUM`

  table.

```
spark.sql("OPTIMIZE delta.`/delta/cleaned_hospital_data` ZORDER BY
(Capacity)")

spark.sql("VACUUM delta.`dbfs:/delta/cleaned_hospital_data` RETAIN 168
HOURS")
logger.info("Delta table optimized and old versions cleaned up.")
```

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`).

```
import dlt
from pyspark.sql import functions as F

@dlt.table(
    name="raw_transactions",
    comment="Raw customer transaction data"
)
def load_raw_transactions():
    return spark.read.format("csv").option("header",
"true").load("/content/transactions.csv")

@dlt.table(
    name="transformed_transactions",
    comment="Transformed transaction data with total
amount calculated"
)
def transform_transactions():
    raw_df = dlt.read("raw_transactions")
    
    return raw_df.withColumn("TotalAmount",
F.col("Quantity") * F.col("Price"))
```

3. **Write DLT in SQL:**

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

```
CREATE OR REFRESH LIVE TABLE raw_transactions AS
SELECT *
```

```
FROM read_csv('/content/transactions.csv', header = true);


CREATE OR REFRESH LIVE TABLE transformed_transactions AS
SELECT *,
       Quantity * Price AS TotalAmount
FROM live.raw_transactions;
```

4. **Monitor the Pipeline:**
   - Use Databricks' DLT UI to monitor the pipeline and check the status of each step.

     It can be monitored in the Databricks user interface after creating this pipeline.

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.

```
delta_table_path = "/delta/transformed_transactions"
transactions_df = spark.read.format("delta").load(delta_table_path)

transactions_df.show(5)

SELECT *
FROM delta.`/delta/transformed_transactions`
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_transactions_df = spark.createDataFrame(new_data,
["TransactionID","TransactionDate", "CustomerID", "Product", "Quantity", "Price"])

    new_transactions_df.write.format("delta").mode("append").save(delta_table_path)
```

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.

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

SELECT *
FROM delta.`/delta/transformed_transactions`
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.

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

transactions_df_filtered =
transactions_df.filter(transactions_df["Quantity"] >= 3)

transactions_df_filtered.write.format("delta").mode("overwri
te").save(delta_table_path)
```

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.

```
merge_data = [
    (1, "2024-09-01", "C001", "Laptop", 1, 1300),
    (8, "2024-09-09", "C007", "Charger", 2, 30)
]

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

merge_temp_path = "/delta/temp_merge_data"
```

```
merge_df.write.format("delta").mode("overwrite").save(merge_temp_path)

from delta.tables import *

delta_table = DeltaTable.forPath(spark, delta_table_path)

delta_table.alias("a").merge(
    merge_df.alias("b"),
    "a.TransactionID = b.TransactionID"
).whenMatchedUpdate(
    condition="a.TransactionID = b.TransactionID",
    set={
        "TransactionDate": "b.TransactionDate",
        "CustomerID": "b.CustomerID",
        "Product": "b.Product",
        "Quantity": "b.Quantity",
        "Price": "b.Price"
    }
).whenNotMatchedInsertAll().execute()
```

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.

   ```
   delta_table_path = "/delta/transformed_transactions"
   history_df = spark.sql(f"DESCRIBE HISTORY delta.`{delta_table_path}`")
   history_df.show(truncate=False)
   ```

   ```
   DESCRIBE HISTORY delta.`dbfs:/delta/transformed_transactions`;
   ```

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).

```
old_version_df = spark.read.format("delta").option("versionAsOf", 5).load(delta_table_path)
old_version_df.show()

timestamp_query_df = spark.read.format("delta").option("timestampAsOf", "2024-09-03 22:59:59").load(delta_table_path)
timestamp_query_df.show()
```

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.

   ```
   spark.sql(f"VACUUM delta.`{delta_table_path}` RETAIN 168 HOURS")

   current_state_df = spark.read.format("delta").load(delta_table_path)
   current_state_df.show()
   ```

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.

   ```
   parquet_table_path = "/delta/transactions_parquet"
   transactions_df.write.format("parquet").mode("overwrite").save(parquet_table_path)

   spark.read.format("parquet").load(parquet_table_path).write.format("delta").mode("overwrite").save(delta_table_path)

   delta_df = spark.read.format("delta").load(delta_table_path)
   delta_df.show()
   ```

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

delta_table_path = "/delta/incremental_transactions"
initial_transactions_df.write.format("delta").mode("overwrite"
).save(delta_table_path)
```

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", "Monitor", 1, 400),
    (7, "2024-09-07", "C002", "Keyboard", 4, 100)
]

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

incremental_transactions_df.write.format("delta").mode("append")
.save(delta_table_path)
```

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.

Pipeline can be created from the tasks above by connecting them as jobs.

4. **Monitor Incremental Load:**

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

```
history_df = spark.sql(f"DESCRIBE HISTORY
delta.`{delta_table_path}`")
history_df.show(truncate=False)

final_state_df = spark.read.format("delta").load(delta_table_path)
final_state_df.show()
```
