**CODING CHALLENGE**

**1.**

**Task 1: Vehicle Maintenance Data Ingestion**

csv\_data = """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"""

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName("DLT Exercise") \

.getOrCreate()

file\_path = "dbfs:/FileStore/vehicle\_maintenance.csv"

try:

# Attempt to read the CSV

attendance\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Save as Delta table

attendance\_df.write.format("delta").mode("overwrite").save("/delta/vehicle\_maintenance\_raw")

except Exception as e:

print(f"Error encountered: {str(e)}")

**Task 2: Data Cleaning**

# Load raw data from Delta table

maintenance\_raw\_df = spark.read.format("delta").load("/delta/vehicle\_maintenance\_raw")

# Filter valid rows (ServiceCost and Mileage must be positive)

cleaned\_df = maintenance\_raw\_df.filter((maintenance\_raw\_df.ServiceCost > 0) &

(maintenance\_raw\_df.Mileage > 0))

# Remove duplicate records based on VehicleID and Date

cleaned\_df = cleaned\_df.dropDuplicates(["VehicleID", "Date"])

# Save cleaned data to a new Delta table

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_vehicle\_maintenance")

**Task 3: Vehicle Maintenance Analysis**

from pyspark.sql.functions import sum

# Load cleaned data from Delta table

cleaned\_df = spark.read.format("delta").load("/delta/cleaned\_vehicle\_maintenance")

# Calculate the total maintenance cost for each vehicle

total\_cost\_df =

cleaned\_df.groupBy("VehicleID").agg(sum("ServiceCost").alias("TotalMaintenanceCost"))

# Identify vehicles with mileage > 30,000

high\_mileage\_df = cleaned\_df.filter(cleaned\_df.Mileage > 30000)

# Save analysis results to Delta tables

total\_cost\_df.write.format("delta").mode("overwrite").save("/delta/total\_maintenance\_cost")

high\_mileage\_df.write.format("delta").mode("overwrite").save("/delta/high\_mileage\_vehicles")

**Task 5: Data Governance with Delta Lake**

# Use VACUUM to remove old data versions

spark.sql("VACUUM `/delta/cleaned\_vehicle\_maintenance` RETAIN 7 HOURS")

# Use DESCRIBE HISTORY to check table history

spark.sql("DESCRIBE HISTORY `/delta/cleaned\_vehicle\_maintenance`").show(truncate=False)

**2.**

**Task 1: Movie Ratings Data Ingestion**

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName("DLT Exercise") \

.getOrCreate()

file\_path = "dbfs:/FileStore/movie\_ratings.csv"

try:

# Attempt to read the CSV

ratings\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Save as Delta table

attendance\_df.write.format("delta").mode("overwrite").save("/delta/movie\_ratings\_raw")

except Exception as e:

print(f"Error encountered: {str(e)}")

**Task 2: Data Cleaning**

# Load raw data from Delta table

ratings\_raw\_df = spark.read.format("delta").load("/delta/movie\_ratings\_raw")

# Filter valid ratings (between 1 and 5)

cleaned\_df = ratings\_raw\_df.filter((ratings\_raw\_df.Rating >= 1) & (ratings\_raw\_df.Rating <= 5))

# Remove duplicates (same UserID and MovieID)

cleaned\_df = cleaned\_df.dropDuplicates(["UserID", "MovieID"])

# Save cleaned data to a new Delta table

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_movie\_ratings")

**Task 3: Movie Rating Analysis**

from pyspark.sql.functions import avg

# Load cleaned data from Delta table

cleaned\_df = spark.read.format("delta").load("/delta/cleaned\_movie\_ratings")

# Calculate the average rating for each movie

avg\_rating\_df = cleaned\_df.groupBy("MovieID").agg(avg("Rating").alias("AvgRating"))

# Identify the movies with the highest and lowest average ratings

highest\_rated\_df = avg\_rating\_df.orderBy("AvgRating", ascending=False).limit(1)

lowest\_rated\_df = avg\_rating\_df.orderBy("AvgRating", ascending=True).limit(1)

# Save analysis results to Delta table

avg\_rating\_df.write.format("delta").mode("overwrite").save("/mnt/delta/movie\_ratings\_analysis")

**Task 4: Time Travel and Delta Lake History**

# Update some ratings

updated\_df = cleaned\_df.withColumn("Rating", when(cleaned\_df.MovieID == "M001",

5).otherwise(cleaned\_df.Rating))

# Save the updated data to Delta table

updated\_df.write.format("delta").mode("overwrite").save("/mnt/delta/cleaned\_movie\_ratings")

# Rollback to a previous version using time travel (e.g., version 0)

previous\_version\_df = spark.read.format("delta").option("versionAsOf",

0).load("/mnt/delta/cleaned\_movie\_ratings")

# View history of changes

spark.sql("DESCRIBE HISTORY '/delta/cleaned\_movie\_ratings'").show(truncate=False)

**Task 5: Optimize Delta Table**

# Perform Z-ordering on the MovieID column

spark.sql("OPTIMIZE '/delta/cleaned\_movie\_ratings' ZORDER BY (MovieID)")

# Use VACUUM to clean up old data versions

spark.sql("VACUUM '/delta/cleaned\_movie\_ratings' RETAIN 0 HOURS")

**3.**

**Task 1: Data Ingestion - Reading Data from Various Formats**

# Ingest CSV Data (Student Information):

student\_csv\_path = "/FileStore/student\_info.csv"

student\_df = spark.read.csv(student\_csv\_path, header=True, inferSchema=True)

student\_df.show()

# Ingest JSON Data (City Information):

city\_df = spark.read.json("/Filestore/city\_info.json")

city\_df.show()

# Ingest Parquet Data (Hospital Information):

hospital\_df = spark.read.parquet("/path/to/hospital\_info.parquet")

hospital\_df.show()

# Ingest Delta Table (Hospital Records):

try:

hospital\_delta\_df = spark.read.format("delta").load("/delta/hospital\_records")

hospital\_delta\_df.show()

except Exception as e:

print(f"Error reading Delta table: {str(e)}")

**Task 2: Writing Data to Various Formats**

# Write Student Data to CSV:

student\_df.write.csv("/path/to/output/student\_data.csv", header=True, mode="overwrite")

# Write City Data to JSON:

city\_df.write.json("/Filestore/output/city\_data.json", mode="overwrite")

# Write Hospital Data to Parquet:

hospital\_df.write.parquet("/Filestore/output/hospital\_data.parquet", mode="overwrite")

# Write Hospital Data to Delta Table:

hospital\_df.write.format("delta").mode("overwrite").save("/delta/hospital\_data")

**Task 3: Running One Notebook from Another**

# Notebook A - Ingest CSV data, clean it, and save to Delta

student\_df = spark.read.csv("/Filestore/student\_data.csv", header=True)

# Clean data (remove duplicates and handle missing values)

cleaned\_student\_df = student\_df.dropDuplicates().na.drop()

# Save cleaned data to Delta table

cleaned\_student\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_student\_data")

# Run Notebook B from Notebook A

dbutils.notebook.run("/path/to/notebook\_b", 60)

# Notebook B - Perform analysis on the cleaned data from Delta table

cleaned\_student\_df = spark.read.format("delta").load("/mnt/delta/cleaned\_student\_data")

# Calculate average score

avg\_score\_df = cleaned\_student\_df.groupBy("Class").agg({"Score": "avg"})

# Save analysis results to Delta table

avg\_score\_df.write.format("delta").mode("overwrite").save("/delta/student\_avg\_score")

**Task 4: Databricks Ingestion**

# Read CSV File from Azure Data Lake:

azure\_csv\_path =

"abfss://<file\_system>@<storage\_account>.dfs.core.windows.net/student\_info.csv"

student\_df = spark.read.csv(azure\_csv\_path, header=True)

# Read JSON File from Databricks FileStore:

json\_path = "dbfs:/FileStore/city\_info.json"

city\_df = spark.read.json(json\_path)

# Read Parquet File from AWS S3:

s3\_parquet\_path = "s3a://bucket-name/path/to/hospital\_info.parquet"

hospital\_df = spark.read.parquet(s3\_parquet\_path)

# Read Delta Table from Databricks-managed Database:

hospital\_delta\_df = spark.read.format("delta").load("/mnt/delta/hospital\_data")

# Write Cleaned Data to Various Formats After Transformations:

# Filtering rows where Class >= 10

transformed\_student\_df = student\_df.filter(student\_df.Class >= 10)

# Write to CSV

transformed\_student\_df.write.csv("/Filestore/output/transformed\_student\_data.csv", header=True)

# Write to JSON

city\_df.write.json("/Filestore/output/transformed\_city\_data.json")

# Write to Parquet

hospital\_df.write.parquet("/Filestore/output/transformed\_hospital\_data.parquet")

# Write to Delta

hospital\_df.write.format("delta").save("/delta/transformed\_hospital\_data")

**# Additional Tasks:**

# Optimization Task: Optimize Delta Table

spark.sql("OPTIMIZE '/delta/hospital\_data'")

# Z-ordering Task: Apply Z-ordering on the CityName or Class Column

spark.sql("OPTIMIZE '/delta/hospital\_data' ZORDER BY (CityName)")

# Vacuum Task: Clean up Old Versions of the Delta Table

spark.sql("VACUUM '/delta/hospital\_data' RETAIN 0 HOURS")

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

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

**Task 2: Write DLT in Python**

import dlt

from pyspark.sql.functions import col

# Step 1: Create the Raw Transactions Table

@dlt.table

def raw\_transactions():

return (

spark.read.format("csv")

.option("header", True)

.option("inferSchema", True)

.load("/path/to/transactions.csv")

)

# Step 2: Create the Transformed Transactions Table

@dlt.table

def transformed\_transactions():

return (

dlt.read("raw\_transactions")

.withColumn("TotalAmount", col("Quantity") \* col("Price"))

)

# Step 3: Write to Delta Table

@dlt.table

def final\_transactions():

return dlt.read("transformed\_transactions")

**Task 3: Write DLT in SQL**

-- Step 1: Create Raw Transactions Table

CREATE OR REPLACE LIVE TABLE raw\_transactions AS

SELECT \* FROM read\_csv('/path/to/transactions.csv');

-- Step 2: Transform Data by Calculating TotalAmount

CREATE OR REPLACE LIVE TABLE transformed\_transactions AS

SELECT \*, Quantity \* Price AS TotalAmount

FROM raw\_transactions;

-- Step 3: Store Final Transformed Data into Delta Table

CREATE OR REPLACE LIVE TABLE final\_transactions AS

SELECT \* FROM transformed\_transactions;

**Task 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**

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

# Read Data using PySpark:

delta\_table\_path = "/path/to/transactions/delta\_table"

df = spark.read.format("delta").load(delta\_table\_path)

# Display first 5 rows of the data

df.show(5)

# Read Data using SQL:

SELECT \* FROM delta.`/path/to/transactions/delta\_table`

LIMIT 5;

**Task 2: Write Data to Delta Lake (Append New Transactions)**

# Create a DataFrame with the new transactions

new\_data = [(6, "2024-09-06", "C005", "Keyboard", 4, 100),

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

columns = ["TransactionID", "TransactionDate", "CustomerID", "Product", "Quantity", "Price"]

new\_df = spark.createDataFrame(new\_data, columns)

# Append the new data to the Delta table

new\_df.write.format("delta").mode("append").save(delta\_table\_path)

**Task 3: Update Data in Delta Lake (Update the Price of 'Laptop')**

-- SQL to update the price of 'Laptop' to 1300

UPDATE delta.`/path/to/transactions/delta\_table`

SET Price = 1300

WHERE Product = 'Laptop';

-- Verify the update

SELECT \* FROM delta.`/path/to/transactions/delta\_table`

WHERE Product = 'Laptop';

**Task 4: Delete Data from Delta Lake (Delete Transactions with Quantity < 3)**

# Delete rows where Quantity is less than 3

delta\_table.delete("Quantity < 3")

# Verify the deletion

delta\_table.toDF().show()

-- SQL to delete rows where Quantity is less than 3

DELETE FROM delta.`/path/to/transactions/delta\_table`

WHERE Quantity < 3;

-- Verify the deletion

SELECT \* FROM delta.`/path/to/transactions/delta\_table`;

**Task 5: Merge Data into Delta Lake (Insert and Update Transactions)**

# New data for merge operation

merge\_data = [(1, "2024-09-01", "C001", "Laptop", 1, 1250),

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

merge\_df = spark.createDataFrame(merge\_data, columns)

# Perform the merge operation

delta\_table.alias("target").merge(

merge\_df.alias("source"),

"target.TransactionID = source.TransactionID"

).whenMatchedUpdateAll(

).whenNotMatchedInsertAll(

).execute()

# Verify the merge operation

delta\_table.toDF().show()

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

**Task 1: View Delta Table History**

# View History using PySpark:

from delta.tables import \*

# Load the Delta table

delta\_table = DeltaTable.forPath(spark, "/path/to/transactions/delta\_table")

# View the history of the Delta table

history\_df = delta\_table.history()

# Display the last 10 operations

history\_df.show(10)

# View History using SQL:

-- SQL to view the history of the Delta table

DESCRIBE HISTORY delta.`/path/to/transactions/delta\_table`;

-- View the last 10 operations

SELECT \* FROM delta.`/path/to/transactions/delta\_table` VERSION AS OF 10;

**Task 2: Perform Time Travel**

# Load the Delta table as it was 5 versions ago

df\_time\_travel = spark.read.format("delta").option("versionAsOf",

5).load("/path/to/transactions/delta\_table")

# Show the data as it was 5 versions ago

df\_time\_travel.show()

# Load the Delta table as it was at a specific timestamp (YYYY-MM-DD HH:MM:SS)

df\_time\_travel = spark.read.format("delta").option("timestampAsOf", "2024-09-01

14:30:00").load("/path/to/transactions/delta\_table")

# Show the data from that timestamp

df\_time\_travel.show()

-- SQL to retrieve the state of the table 5 versions ago

SELECT \* FROM delta.`/path/to/transactions/delta\_table` VERSION AS OF 5;

-- SQL to retrieve the state of the table at a specific timestamp

SELECT \* FROM delta.`/path/to/transactions/delta\_table` TIMESTAMP AS OF '2024-09-01 14:30:00';

**Task 3: Vacuum the Delta Table**

-- SQL to vacuum the Delta table with a retention period of 7 days

VACUUM delta.`/path/to/transactions/delta\_table` RETAIN 168 HOURS;

-- Verify that the current table state is intact

SELECT \* FROM delta.`/path/to/transactions/delta\_table`;

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

# Load raw transaction CSV data

csv\_file\_path = "/path/to/raw\_transactions.csv"

df\_csv = spark.read.option("header", "true").csv(csv\_file\_path)

# Save as a Parquet table

parquet\_table\_path = "/path/to/parquet\_transactions"

df\_csv.write.format("parquet").save(parquet\_table\_path)

# Convert the Parquet table to a Delta table

parquet\_df = spark.read.format("parquet").load(parquet\_table\_path)

parquet\_df.write.format("delta").save("/path/to/delta\_transactions")

-- SQL to convert Parquet table to Delta

CONVERT TO DELTA parquet.`/path/to/parquet\_transactions`;

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

**Task 1: Set Up Initial Data**

**Load initial transactions data (from 2024-09-01 to 2024-09-03)**

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)

]

# Define the schema

schema = ['TransactionID', 'TransactionDate', 'CustomerID', 'Product', 'Quantity', 'Price']

# Create DataFrame

initial\_df = spark.createDataFrame(initial\_data, schema=schema)

# Write the initial data to a Delta table

initial\_df.write.format("delta").mode("overwrite").save("/path/to/delta/transactions")

**Task 2: Set Up Incremental Data**

# New transactions data (from 2024-09-04 to 2024-09-07)

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)

]

# Create DataFrame for incremental data

incremental\_df = spark.createDataFrame(incremental\_data, schema=schema)

**Task 3: Implement Incremental Load**

# Read the existing Delta table (initial data from 2024-09-01 to 2024-09-03)

existing\_df = spark.read.format("delta").load("/path/to/delta/transactions")

# Show the existing data

existing\_df.show()

# Append new transactions (from 2024-09-04 to 2024-09-07) to the Delta table

incremental\_df.write.format("delta").mode("append").save("/path/to/delta/transactions")

# Verify that the new transactions have been appended without overwriting

updated\_df = spark.read.format("delta").load("/path/to/delta/transactions")

updated\_df.show()

**Task 4: Monitor Incremental Load**

-- SQL to view the version history of the Delta table

DESCRIBE HISTORY delta.`/path/to/delta/transactions`;