**Worked in Google Colab**

**#Assignment: Delta Lake Concepts**

**#Task 1: Creating Delta Table using Three Methods**

#1. Load the given CSV and JSON datasets into Databricks.

df\_employees = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/employees.csv")

df\_products = spark.read.format("json").load("dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/products.json")

# Load the Employees CSV file into a DataFrame and cache it

df\_employees = spark.read.format("csv") \

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

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

.load("dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/employees.csv").cache()

# Load the Products JSON file into a DataFrame and cache it

df\_products = spark.read.format("json") \

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

.load("dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/products.json").cache()

#2. Create a Delta table using the following three methods:

#Create a Delta table from a DataFrame.

df\_employees.write.format("delta").mode("overwrite").save("/delta/employees")

# Create temporary view for Employees DataFrame

df\_employees.createOrReplaceTempView("df\_employees")

# Create temporary view for Products DataFrame

df\_products.createOrReplaceTempView("df\_products")

%sql

--Use SQL to create a Delta table.

-- Employees

CREATE TABLE employees\_delta

USING delta

AS SELECT \* FROM df\_employees;

#Convert both the CSV and JSON files into Delta format.

df\_employees.write.format("delta").mode("overwrite").save("/delta/employees\_converted")

df\_products.write.format("delta").mode("overwrite").save("/delta/products\_converted")

**#Task 2: Merge and Upsert (Slowly Changing Dimension - SCD)**

#1. Load the Delta table for employees created in Task 1.

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

#2. Merge the new employee data into the employees Delta table.

new\_employee\_data = [

(102, "Alice", "Finance", "2023-02-15", 75000),  # Updated Salary

(106, "Olivia", "HR", "2023-06-10", 65000)       # New Employee

]

columns = ["EmployeeID", "EmployeeName", "Department", "JoiningDate", "Salary"]

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

#3. If an employee exists, update their salary. If the employee is new, insert their details.

#new\_employees\_df.createOrReplaceTempView("new\_employees\_view")

spark.sql("""

MERGE INTO delta.`/delta/employees` AS target

USING new\_employees\_view AS source

ON target.EmployeeID = source.EmployeeID

WHEN MATCHED THEN

UPDATE SET target.Salary = source.Salary

WHEN NOT MATCHED THEN

INSERT (EmployeeID, EmployeeName, Department, JoiningDate, Salary)

VALUES (source.EmployeeID, source.EmployeeName, source.Department, source.JoiningDate, source.Salary);

""")

# Verify the updated Delta table

updated\_employees\_df = spark.read.format("delta").load("/delta/employees")

updated\_employees\_df.show()

**#Task 3: Internals of Delta Table**

#1. Explore the internals of the employees Delta table using Delta Lake features.

# Describe the Delta table to see its metadata and internals

spark.sql("DESCRIBE DETAIL delta.`/delta/employees`").show(truncate=False)

# Show the schema of the Delta table

spark.sql("DESCRIBE delta.`/delta/employees`").show()

# Check the number of files in the Delta table

spark.sql("DESCRIBE HISTORY delta.`/delta/employees`").show()

#2. Check the transaction history of the table.

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

%sql

--3. Perform Time Travel and retrieve the table before the previous merge operation.

-- Retrieve the Delta table using a version number

SELECT \* FROM delta.`/delta/employees` VERSION AS OF 1;

-- Retrieve the Delta table using a valid timestamp

SELECT \* FROM delta.`/delta/employees` TIMESTAMP AS OF '2024-09-17T04:30:00.000Z';

**#Task 4: Optimize Delta Table**

#1. Optimize the employees Delta table for better performance.

spark.sql("""

OPTIMIZE delta.`/delta/employees`;

""")

#2. Use Z-ordering on the Department column for improved query performance.

spark.sql("""

OPTIMIZE delta.`/delta/employees` ZORDER BY (Department);

""")

**#Task 5: Time Travel with Delta Table**

#1. Retrieve the employees Delta table as it was before the last merge.

spark.sql("""

DESCRIBE HISTORY delta.`/delta/employees`;

""")

#2. Query the table at a specific version to view the older records.

spark.sql("""

SELECT \* FROM delta.`/delta/employees` VERSION AS OF 2;

""")

**#Task 6: Vacuum Delta Table**

#1. Use the vacuum operation on the employees Delta table to remove old versions and free up disk space.

spark.sql("""

VACUUM delta.`/delta/employees`;

""")

#2. Set the retention period to 7 days and ensure that old files are deleted.

spark.sql("""

VACUUM delta.`/delta/employees` RETAIN 168 HOURS;

""")

**#Assignment: Structured Streaming and Transformations on Streams**

**#Task 1: Ingest Streaming Data from CSV Files**

streaming\_folder\_path = "/mnt/streaming\_csv\_data"

dbutils.fs.mkdirs(streaming\_folder\_path)

#2. Set up a structured streaming source to continuously read CSV data from this folder.

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DateType

schema = StructType([

StructField("TransactionID", StringType(), True),

StructField("TransactionDate", DateType(), True),

StructField("ProductID", StringType(), True),

StructField("Quantity", IntegerType(), True),

StructField("Price", IntegerType(), True)

])

streaming\_df = spark.readStream \

.schema(schema) \

.csv(streaming\_folder\_path)

#3. Ensure that the streaming query reads the data continuously in append mode and displays the results in the console.

query = streaming\_df.writeStream \

.outputMode("append") \

.format("console") \

.option("truncate", False) \

.start()

#query.awaitTermination()

**#Task 2: Stream Transformations**

#Add a new column for the TotalAmount ( Quantity \* Price ).

#Filter records where the Quantity is greater than 1.

from pyspark.sql.functions import col

transformed\_df = streaming\_df \

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

.filter(col("Quantity") > 1)

#2. Write the transformed stream to a memory sink to see the updated results continuously.

query = transformed\_df.writeStream \

.outputMode("append") \

.format("memory") \

.queryName("transformed\_stream") \

.start()

#query.awaitTermination()

# Display the results from the memory table

display(spark.table("transformed\_stream"))

**#Task 3: Aggregations on Streaming Data**

#1. Implement an aggregation on the streaming data:

#Group the data by ProductID and calculate the total sales for each product

from pyspark.sql.functions import col, sum as sum\_

# Aggregate data: sum of TotalAmount grouped by ProductID

aggregated\_df = transformed\_df \

.groupBy("ProductID") \

.agg(

sum\_("TotalAmount").alias("TotalSales")

)

# 2. Ensure the stream runs in update mode, so only updated results are output to the sink.

query = aggregated\_df.writeStream \

.outputMode("update") \

.format("memory") \

.queryName("aggregated\_sales\_stream") \

.start()

#query.awaitTermination()

display(spark.table("aggregated\_sales\_stream"))

**#Task 4: Writing Streaming Data to File Sinks**

#1. After transforming and aggregating the data, write the streaming results to a Parquet sink.

#2. Ensure that you configure a checkpoint location to store progress and ensure recovery in case of failure.

parquet\_sink\_path = "/mnt/streaming\_parquet\_data"

checkpoint\_location = "/mnt/checkpoints/aggregated\_sales"

query = aggregated\_df.writeStream \

.outputMode("update") \

.format("parquet") \

.option("path", parquet\_sink\_path) \

.option("checkpointLocation", checkpoint\_location) \

.start()

query.awaitTermination()

**#Task 5: Handling Late Data using Watermarks**

#1. Introduce a watermark on the TransactionDate column to handle late data arriving in the stream.

#2. Set the watermark to 1 day to allow late data within a 24-hour period and discard data that is older.

from pyspark.sql.functions import col, to\_timestamp, sum as sum\_

transformed\_df\_with\_timestamp = transformed\_df \

.withColumn("TransactionDate", to\_timestamp(col("TransactionDate")))

watermarked\_df = transformed\_df\_with\_timestamp \

.withWatermark("TransactionDate", "1 day")

aggregated\_df = watermarked\_df \

.groupBy("ProductID") \

.agg(

sum\_("TotalAmount").alias("TotalSales")

)

parquet\_sink\_path = "/mnt/streaming\_parquet\_data"

checkpoint\_location = "/mnt/checkpoints/aggregated\_sales"

query = aggregated\_df.writeStream \

.outputMode("append") \

.format("parquet") \

.option("path", parquet\_sink\_path) \

.option("checkpointLocation", checkpoint\_location) \

.start()

query.awaitTermination()

**#Task 6: Streaming from Multiple Sources**

#1. Simulate a scenario where two streams of data are being ingested:

#Stream 1: Incoming transaction data (same as Task 1).

#Stream 2: Product information (CSV with columns: ProductID, ProductName, Category).

from pyspark.sql.types import StructType, StructField, StringType, IntegerType, TimestampType

transaction\_schema = StructType([

StructField("TransactionID", StringType(), True),

StructField("TransactionDate", TimestampType(), True),

StructField("ProductID", StringType(), True),

StructField("Quantity", IntegerType(), True),

StructField("Price", IntegerType(), True)

])

product\_schema = StructType([

StructField("ProductID", StringType(), True),

StructField("ProductName", StringType(), True),

StructField("Category", StringType(), True)

])

transaction\_stream\_df = spark.readStream \

.schema(transaction\_schema) \

.csv("/mnt/streaming\_csv\_data")  # Update with the path to your CSV folder

product\_stream\_df = spark.readStream \

.schema(product\_schema) \

.csv("/mnt/streaming\_csv\_data")  # Update with the path to your CSV folder

#2. Perform a join on the two streams using the ProductID column and display the combined stream results.

from pyspark.sql.functions import col

joined\_stream\_df = transaction\_stream\_df \

.join(product\_stream\_df, on="ProductID", how="inner")

display\_df = joined\_stream\_df \

.select(

col("TransactionID"),

col("TransactionDate"),

col("ProductID"),

col("ProductName"),

col("Category"),

col("Quantity"),

col("Price"),

(col("Quantity") \* col("Price")).alias("TotalAmount")

)

query = display\_df.writeStream \

.outputMode("append") \

.format("console") \

.start()

#query.awaitTermination()

**#Task 7: Stopping and Restarting Streaming Queries**

#1. Stop the streaming query and explore the results.

query.stop()

print("Streaming Query Stopped.")

#2. Restart the query and ensure that it continues from the last processed data by utilizing the checkpoint.

checkpoint\_location = "/mnt/checkpoints/aggregated\_sales"

query = aggregated\_df.writeStream \

.outputMode("append") \

.format("parquet") \

.option("path", parquet\_sink\_path) \

.option("checkpointLocation", checkpoint\_location) \

.start()

query.awaitTermination()

**#Assignment: Creating a Complete ETL Pipeline using Delta Live Tables (DLT)**

**#Task 1: Create an ETL Pipeline using DLT (Python)**

#1. Create a Delta Live Table pipeline using PySpark to perform the following:

#Read the source data from a CSV or Parquet file.

#Transform the data by performing the following:

#Add a new column for TotalAmount which is the result of

#multiplying Quantity by Price .

#Filter records where the Quantity is greater than 1.

#Load the transformed data into a Delta table.

#2. Ensure the pipeline is repeatable and can handle incremental loads by re-running with new data.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

spark = SparkSession.builder \

.appName("ETL Pipeline with DLT") \

.getOrCreate()

source\_df = spark.read.csv("dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/orders.csv", header=True, inferSchema=True)

transformed\_df = source\_df \

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

.filter(col("Quantity") > 1)

transformed\_df.write.format("delta").mode("append").saveAsTable("delta\_table\_orders")

existing\_df = spark.read.format("delta").table("delta\_table\_orders")

incremental\_df = transformed\_df.join(

existing\_df.select("OrderID"),

on="OrderID",

how="left\_anti"

)

incremental\_df.write.format("delta").mode("append").saveAsTable("delta\_table\_orders")

# Stop Spark Session

spark.stop()

**#Task 2: Create an ETL Pipeline using DLT (SQL)**

#1. Create a similar Delta Live Table pipeline using SQL:

#Use SQL to read the source data, perform the same transformations (as above), and write the data into a Delta table.

spark.sql("""

CREATE TABLE IF NOT EXISTS source\_orders (

OrderID INT,

OrderDate DATE,

CustomerID STRING,

Product STRING,

Quantity INT,

Price DECIMAL(10, 2)

)

USING DELTA;

""")

%sql

-- Load data from CSV into a temporary view

CREATE OR REPLACE TEMPORARY VIEW temp\_source\_orders

USING csv

OPTIONS (

path 'dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/orders.csv',

header 'true',

inferSchema 'true'

);

-- Insert data from the temporary view into the Delta table

INSERT INTO source\_orders

SELECT \* FROM temp\_source\_orders;

%sql

-- Create or replace the target Delta table for transformed data

CREATE OR REPLACE TABLE transformed\_orders AS

SELECT

OrderID,

OrderDate,

CustomerID,

Product,

Quantity,

Price,

Quantity \* Price AS TotalAmount

FROM source\_orders

WHERE Quantity > 1;

%sql

--Ensure the pipeline can process incremental data without losing records or creating duplicates.

-- Upsert new records into the transformed Delta table

MERGE INTO transformed\_orders AS target

USING (

SELECT

OrderID,

OrderDate,

CustomerID,

Product,

Quantity,

Price,

Quantity \* Price AS TotalAmount

FROM source\_orders

WHERE Quantity > 1

) AS source

ON target.OrderID = source.OrderID

WHEN MATCHED THEN

UPDATE SET

target.OrderDate = source.OrderDate,

target.CustomerID = source.CustomerID,

target.Product = source.Product,

target.Quantity = source.Quantity,

target.Price = source.Price,

target.TotalAmount = source.TotalAmount

WHEN NOT MATCHED THEN

INSERT (OrderID, OrderDate, CustomerID, Product, Quantity, Price, TotalAmount)

VALUES (source.OrderID, source.OrderDate, source.CustomerID, source.Product, source.Quantity, source.Price, source.TotalAmount);

%sql

**--Task 3: Perform Read, Write, Update, and Delete Operations on Delta Table (SQL + PySpark)**

--1. Read the data from the Delta table created in Task 1 and Task 2.

SELECT \* FROM transformed\_orders;

from pyspark.sql import SparkSession

# Initialize Spark Session

spark = SparkSession.builder \

.appName("Delta Table Operations") \

.getOrCreate()

# Read data from the Delta table using PySpark

df = spark.read.format("delta").table("transformed\_orders")

df.show()

%sql

--2. Update the table by changing the price of a product (e.g., increase the price of laptops by 10%).

UPDATE transformed\_orders

SET Price = Price \* 1.10

WHERE Product = 'Laptop';

#3. Delete rows from the Delta table where the quantity is less than 2.

spark.sql("""

DELETE FROM transformed\_orders

WHERE Quantity < 2;

""")

#4. Insert a new record into the Delta table using PySpark or SQL.

spark.sql("""

INSERT INTO transformed\_orders

VALUES (106, '2024-01-06', 'C006', 'Keyboard', 3, 50, 150);

""")

**#Task 4: Merge Data (Slowly Changing Dimension - SCD Type 2)**

#1. Create a new dataset representing updated orders with new prices and products.

spark.sql("""

CREATE OR REPLACE TEMPORARY VIEW updated\_orders AS

SELECT \* FROM (

VALUES

(101, '2024-01-10', 'C001', 'Laptop', 2, 1200),

(106, '2024-01-12', 'C006', 'Keyboard', 3, 50)

) AS updated\_orders(OrderID, OrderDate, CustomerID, Product, Quantity, Price);

""")

#Implement a MERGE operation to simulate a Slowly Changing Dimension Type 2 (SCD2) scenario. Ensure that:

#The Quantity , Price , and TotalAmount columns are updated if there is a match on OrderID .

#If no match is found, insert the new record into the Delta table.

spark.sql("""

MERGE INTO transformed\_orders AS target

USING updated\_orders AS source

ON target.OrderID = source.OrderID

WHEN MATCHED THEN

UPDATE SET

target.Quantity = source.Quantity,

target.Price = source.Price,

target.TotalAmount = source.Quantity \* source.Price,

target.OrderDate = source.OrderDate,

target.CustomerID = source.CustomerID,

target.Product = source.Product

WHEN NOT MATCHED THEN

INSERT (OrderID, OrderDate, CustomerID, Product, Quantity, Price, TotalAmount)

VALUES (source.OrderID, source.OrderDate, source.CustomerID, source.Product, source.Quantity, source.Price, source.Quantity \* source.Price);

""")

**#Task 5: Explore Delta Table Internals**

#1. Inspect the Delta table's transaction logs and explore the metadata using SQL queries:

#Display the history of changes to the Delta table using the DESCRIBE HISTORY command.

spark.sql("""

DESCRIBE HISTORY transformed\_orders;

""")

#Check the file size and modification times using DESCRIBE DETAIL .

spark.sql("""

DESCRIBE DETAIL transformed\_orders;

""")

**#Task 6: Time Travel in Delta Tables**

#1. Use time travel to query the Delta table as it existed at a previous point in time.

#Query the table as it existed before the last merge operation.

spark.sql("""

DESCRIBE HISTORY transformed\_orders;

""")

#Demonstrate time travel by using both the version of the table and the timestamp.

spark.sql("""

SELECT \* FROM transformed\_orders VERSION AS OF 5;

""")

spark.sql("""

SELECT \* FROM transformed\_orders TIMESTAMP AS OF '2024-09-17T08:56:56';

""")

**#Task 7: Optimize Delta Table**

#1. Optimize the Delta table for faster queries using Z-Ordering.

#Optimize the table on the Product column to reduce I/O and improve query performance.

spark.sql("""

OPTIMIZE transformed\_orders

ZORDER BY (Product);

""")

#2. Use vacuum to remove any old files that are no longer necessary after the optimization process.

spark.sql("""

VACUUM transformed\_orders;

""")

**#Task 8: Converting Parquet Files to Delta Format**

#1. You are provided with Parquet files containing historical order data. Convert these files into a Delta table format using either PySpark or SQL.

# Save DataFrame as Parquet

source\_df.write.format("parquet").mode("overwrite").save("/mnt/delta/historical\_orders\_parquet")

source\_df.write.format("delta").mode("overwrite").saveAsTable("historical\_orders\_delta")

#Perform a simple query on the converted Delta table to verify the conversion.

spark.sql("SELECT \* FROM historical\_orders\_delta").show()

**#Assignment: Creating and Scheduling a Job on Databricks using Notebooks**

**#Task 1: Prepare Your Notebook**

#1. Create a new Notebook in your Databricks workspace.

#Use PySpark for data processing.

#In the notebook, read a CSV file (use the provided sample data), perform a transformation, and write the transformed data into a Delta table.

#The transformation should include:

#Adding a new column ( TotalAmount ) which is the product of Quantity and Price .

#Filtering rows where Quantity is greater than 5.

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

spark = SparkSession.builder.appName("TransformData").getOrCreate()

csv\_file\_path = "dbfs:/FileStore/shared\_uploads/varshinie.1006@gmail.com/orders.csv"

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

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

df\_filtered = df\_transformed.filter(col("Quantity") > 5)

df\_filtered.show()

delta\_table\_path = "/mnt/delta/transformed\_orders\_delta"

df\_filtered.write.format("delta").mode("overwrite").save(delta\_table\_path)

spark.sql("""

CREATE TABLE IF NOT EXISTS transformed\_orders\_delta

USING DELTA

LOCATION '{}'

""".format(delta\_table\_path))

spark.sql("SELECT \* FROM transformed\_orders\_delta").show()