**DataBricks Assignment (18.09.24)**

**By Amita c**

**Data Governance Using Unity Catalog - Advanced Capabilities**

**Task 1: Set Up Unity Catalog Objects with Multiple Schemas**

1.Create a Catalog:

CREATE CATALOG finance\_data\_catalog;

2. Create Multiple Schemas:

CREATE SCHEMA finance\_data\_catalog.transaction\_data;

CREATE SCHEMA finance\_data\_catalog.customer\_data;

3. Create Tables in Each Schema:

transaction\_data:

CREATE TABLE finance\_data\_catalog.transaction\_data.transactions (

TransactionID STRING,

CustomerID STRING,

TransactionAmount DECIMAL(10, 2),

TransactionDate DATE

);

customer\_data:

CREATE TABLE finance\_data\_catalog.customer\_data.customers (

CustomerID STRING,

CustomerName STRING,

Email STRING,

Country STRING

);

**Task 2: Data Discovery Across Schemas**

1.Explore Metadata:

SHOW TABLES IN finance\_data\_catalog.transaction\_data;

SHOW TABLES IN finance\_data\_catalog.customer\_data;

DESCRIBE TABLE finance\_data\_catalog.transaction\_data.transactions;

DESCRIBE TABLE finance\_data\_catalog.customer\_data.customers;

2. Data Profiling:

Calculate basic statistics

SELECT

MIN(TransactionAmount) AS Min\_TransactionAmount,

MAX(TransactionAmount) AS Max\_TransactionAmount,

AVG(TransactionAmount) AS Avg\_TransactionAmount,

SUM(TransactionAmount) AS Total\_TransactionAmount,

COUNT(\*) AS Total\_Transactions

FROM finance\_data\_catalog.transaction\_data.transactions;

Identify trends by date

SELECT

YEAR(TransactionDate) AS Year,

MONTH(TransactionDate) AS Month,

COUNT(\*) AS Total\_Transactions,

SUM(TransactionAmount) AS Total\_Amount

FROM finance\_data\_catalog.transaction\_data.transactions

GROUP BY

YEAR(TransactionDate), MONTH(TransactionDate)

ORDER BY Year, Month;

Profile the Country distribution

SELECT Country,

COUNT(\*) AS Number\_of\_Customers

FROM

finance\_data\_catalog.customer\_data.customers

GROUP BY

Country

ORDER BY

Number\_of\_Customers DESC;

3. Tagging Sensitive Data:

ALTER TABLE finance\_data\_catalog.customer\_data.customers

ADD TAG ('sensitive' = 'Email');

ALTER TABLE finance\_data\_catalog.transaction\_data.transactions

ADD TAG ('sensitive' = 'TransactionAmount');

**Task 3: Implement Data Lineage and Auditing**

1.Track Data Lineage:

Merge Data:

CREATE OR REPLACE VIEW finance\_data\_catalog.merged\_data AS

SELECT

t.TransactionID,

t.CustomerID,

t.TransactionAmount,

t.TransactionDate,

c.CustomerName,

c.Email,

c.Country

FROM

finance\_data\_catalog.transaction\_data.transactions t

JOIN

finance\_data\_catalog.customer\_data.customers c

ON

t.CustomerID = c.CustomerID;

Query Audit Logs:

SELECT

user\_name,

action\_name,

object\_name,

timestamp

FROM

audit\_logs

WHERE

object\_name = 'finance\_data\_catalog.transaction\_data.transactions'

AND action\_name IN ('READ', 'WRITE', 'MODIFY')

ORDER BY

timestamp DESC;

**Task 4: Access Control and Permissions**

1.Set Up Roles and Groups:

databricks groups create --group-name DataEngineers

databricks groups create --group-name DataAnalysts

GRANT ALL PRIVILEGES ON SCHEMA finance\_data\_catalog.transaction\_data TO

`DataEngineers`;

GRANT ALL PRIVILEGES ON SCHEMA finance\_data\_catalog.customer\_data TO

`DataEngineers`;

GRANT SELECT ON SCHEMA finance\_data\_catalog.customer\_data TO `DataAnalysts`;

REVOKE ALL PRIVILEGES ON SCHEMA finance\_data\_catalog.transaction\_data FROM

`DataAnalysts`;

GRANT SELECT ON TABLE finance\_data\_catalog.transaction\_data.transactions TO

`DataAnalysts`;

2. Row-Level Security:

databricks groups create --group-name HighValueAnalysts

CREATE OR REPLACE VIEW finance\_data\_catalog.transaction\_data.secured\_transactions

AS

SELECT

TransactionID,

CustomerID,

TransactionAmount,

TransactionDate

FROM

finance\_data\_catalog.transaction\_data.transactions

WHERE

(TransactionAmount <= 10000)

OR

(TransactionAmount > 10000 AND CURRENT\_USER() IN ('HighValueAnalysts'));

GRANT SELECT ON VIEW finance\_data\_catalog.transaction\_data.secured\_transactions TO

`DataAnalysts`;

GRANT SELECT ON VIEW finance\_data\_catalog.transaction\_data.secured\_transactions TO

`HighValueAnalysts`;

**Task 5: Data Governance Best Practices**

1.Create Data Quality Rules:

ALTER TABLE finance\_data\_catalog.transaction\_data.transactions

ADD CONSTRAINT chk\_non\_negative\_amount CHECK (TransactionAmount >= 0);

ALTER TABLE finance\_data\_catalog.customer\_data.customers

ADD CONSTRAINT chk\_email\_format CHECK (Email LIKE '%\_@\_\_%.\_\_%');

2. Validate Data Governance:

SELECT \*

FROM finance\_data\_catalog.transaction\_data.transactions

WHERE TransactionAmount < 0;

SELECT \*

FROM finance\_data\_catalog.customer\_data.customers

WHERE Email NOT LIKE '%\_@\_\_%.\_\_%';

Validate Audit Logs:

SELECT user\_name,action\_name,object\_name,timestamp

FROM audit\_logs

WHERE

action\_name IN ('MODIFY', 'ALTER')

AND object\_name IN ('finance\_data\_catalog.transaction\_data.transactions',

'finance\_data\_catalog.customer\_data.customers')

ORDER BY

timestamp DESC;

SELECT user\_name, action\_name,object\_name,timestamp

FROM

audit\_logs

WHERE

object\_name IN ('finance\_data\_catalog.transaction\_data.transactions',

'finance\_data\_catalog.customer\_data.customers')

AND action\_name = 'SELECT'

ORDER BY

timestamp DESC;

**Task 6: Data Lifecycle Management**

1.Implement Time Travel:

SELECT \*

FROM finance\_data\_catalog.transaction\_data.transactions VERSION AS OF 3;

RESTORE TABLE finance\_data\_catalog.transaction\_data.transactions TO VERSION AS OF

3;

RESTORE TABLE finance\_data\_catalog.transaction\_data.transactions TO TIMESTAMP AS

OF '2024-09-15T12:00:00';

2. Run a Vacuum Operation:

VACUUM finance\_data\_catalog.transaction\_data.transactions;

VACUUM finance\_data\_catalog.transaction\_data.transactions RETAIN 168 HOURS;

DESCRIBE HISTORY finance\_data\_catalog.transaction\_data.transactions;

**Advanced Data Governance and Security Using Unity Catalog**

**Task 1: Set Up Multi-Tenant Data Architecture Using Unity Catalog**

1.Create a New Catalog:

CREATE CATALOG IF NOT EXISTS corporate\_data\_catalog;

2. Create Schemas for Each Department:

CREATE SCHEMA IF NOT EXISTS corporate\_data\_catalog.sales\_data;

CREATE SCHEMA IF NOT EXISTS corporate\_data\_catalog.hr\_data;

CREATE SCHEMA IF NOT EXISTS corporate\_data\_catalog.finance\_data;

3. Create Tables in Each Schema:

CREATE TABLE IF NOT EXISTS corporate\_data\_catalog.sales\_data.sales (

SalesID INT,

CustomerID INT,

SalesAmount DECIMAL(10, 2),

SalesDate DATE

);

CREATE TABLE IF NOT EXISTS corporate\_data\_catalog.hr\_data.employees (

EmployeeID INT,

EmployeeName STRING,

Department STRING,

Salary DECIMAL(10, 2)

);

CREATE TABLE IF NOT EXISTS corporate\_data\_catalog.finance\_data.invoices (

InvoiceID INT,

VendorID INT,

InvoiceAmount DECIMAL(10, 2),

PaymentDate DATE

);

**Task 2: Enable Data Discovery for Cross-Departmental Data**

1. Search for Tables Across Departments:

SHOW TABLES IN corporate\_data\_catalog.sales\_data;

SHOW TABLES IN corporate\_data\_catalog.hr\_data;

SHOW TABLES IN corporate\_data\_catalog.finance\_data;

2. Tag Sensitive Information:

ALTER TABLE corporate\_data\_catalog.hr\_data.employees

ADD TAG Sensitive ON COLUMN Salary;

ALTER TABLE corporate\_data\_catalog.finance\_data.invoices

ADD TAG Sensitive ON COLUMN InvoiceAmount;

3. Data Profiling:

SELECT SalesDate, SUM(SalesAmount) AS TotalSales

FROM corporate\_data\_catalog.sales\_data.sales

GROUP BY SalesDate

ORDER BY SalesDate;

SELECT Department, AVG(Salary) AS AvgSalary, MAX(Salary) AS MaxSalary,

MIN(Salary) AS MinSalary

FROM corporate\_data\_catalog.hr\_data.employees

GROUP BY Department;

SELECT PaymentDate, SUM(InvoiceAmount) AS TotalInvoices

FROM corporate\_data\_catalog.finance\_data.invoices

GROUP BY PaymentDate

ORDER BY PaymentDate;

**Task 3: Implement Data Lineage and Data Auditing**

1.Track Data Lineage:

CREATE TABLE corporate\_data\_catalog.reporting.sales\_finance\_report AS

SELECT

s.SalesID,

s.CustomerID,

s.SalesAmount,

s.SalesDate,

f.InvoiceID,

f.InvoiceAmount,

f.PaymentDate

FROM corporate\_data\_catalog.sales\_data.sales AS s

JOIN corporate\_data\_catalog.finance\_data.invoices AS f

ON s.CustomerID = f.VendorID;

2. Enable Data Audit Logs:

ENABLE AUDIT LOGGING ON corporate\_data\_catalog.hr\_data.employees;

ENABLE AUDIT LOGGING ON corporate\_data\_catalog.finance\_data.invoices;

**Task 4: Data Access Control and Security**

1.Set Up Roles and Permissions:

CREATE GROUP SalesTeam;

CREATE GROUP FinanceTeam;

CREATE GROUP HRTeam;

Grant Access

GRANT USAGE ON SCHEMA corporate\_data\_catalog.sales\_data TO SalesTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA corporate\_data\_catalog.sales\_data TO

SalesTeam;

GRANT USAGE ON SCHEMA corporate\_data\_catalog.sales\_data TO FinanceTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA corporate\_data\_catalog.sales\_data TO

FinanceTeam;

GRANT USAGE ON SCHEMA corporate\_data\_catalog.finance\_data TO FinanceTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA corporate\_data\_catalog.finance\_data TO

FinanceTeam;

GRANT USAGE ON SCHEMA corporate\_data\_catalog.hr\_data TO HRTeam;

GRANT SELECT, UPDATE ON TABLE corporate\_data\_catalog.hr\_data.employees TO

HRTeam;

2. Implement Column-Level Security:

CREATE GROUP HRManagers;

ALTER TABLE corporate\_data\_catalog.hr\_data.employees

ALTER COLUMN Salary SET MASKING POLICY 'HRManagersCanViewSalary';

3. Row-Level Security:

CREATE GROUP SalesReps;

CREATE ROW ACCESS POLICY SalesRepCanViewOwnRecords

AS (SalesRepID INT) RETURNS BOOLEAN

-> current\_user() = SalesRepID;

ALTER TABLE corporate\_data\_catalog.sales\_data.sales

SET ROW ACCESS POLICY SalesRepCanViewOwnRecords ON (SalesRepID);

**Task 5: Data Governance Best Practices**

1.Define Data Quality Rules:

SELECT \*

FROM corporate\_data\_catalog.sales\_data.sales

WHERE SalesAmount <= 0;

SELECT \*

FROM corporate\_data\_catalog.hr\_data.employees

WHERE Salary <= 0;

SELECT \*

FROM corporate\_data\_catalog.finance\_data.invoices AS inv

JOIN corporate\_data\_catalog.finance\_data.payments AS pay

ON inv.InvoiceID = pay.InvoiceID

WHERE inv.InvoiceAmount != pay.PaymentAmount;

2. Apply Time Travel for Data Auditing:

DESCRIBE HISTORY corporate\_data\_catalog.finance\_data.invoices;

RESTORE TABLE corporate\_data\_catalog.finance\_data.invoices

TO VERSION AS OF <version\_number>;

Task 6: Optimize and Clean Up Delta Tables

1.Optimize Delta Tables:

OPTIMIZE corporate\_data\_catalog.sales\_data.sales;

OPTIMIZE corporate\_data\_catalog.finance\_data.invoices;

2. Vacuum Delta Tables:

VACUUM corporate\_data\_catalog.sales\_data.sales RETAIN 168 HOURS;

VACUUM corporate\_data\_catalog.finance\_data.invoices RETAIN 168 HOURS;

**Building a Secure Data Platform with Unity Catalog**

**Task 1: Set Up Unity Catalog for Multi-Domain Data Management**

1.Create a New Catalog:

CREATE CATALOG enterprise\_data\_catalog;

2. Create Domain-Specific Schemas:

CREATE SCHEMA enterprise\_data\_catalog.marketing\_data;

CREATE SCHEMA enterprise\_data\_catalog.operations\_data;

CREATE SCHEMA enterprise\_data\_catalog.it\_data;

3. Create Tables in Each Schema:

CREATE TABLE enterprise\_data\_catalog.marketing\_data.campaigns (

CampaignID INT,

CampaignName STRING,

Budget DECIMAL(10, 2),

StartDate DATE

);

CREATE TABLE enterprise\_data\_catalog.operations\_data.orders (

OrderID INT,

ProductID INT,

Quantity INT,

ShippingStatus STRING

);

CREATE TABLE enterprise\_data\_catalog.it\_data.incidents (

IncidentID INT,

ReportedBy STRING,

IssueType STRING,

ResolutionTime INT

);

**Task 2: Data Discovery and Classification**

1.Search for Data Across Schemas:

SHOW TABLES IN enterprise\_data\_catalog;

SELECT table\_catalog, table\_schema, table\_name, column\_name, data\_type

FROM information\_schema.columns

WHERE column\_name IN ('Budget', 'ResolutionTime');

2. Tag Sensitive Information:

ALTER TABLE enterprise\_data\_catalog.marketing\_data.campaigns

ALTER COLUMN Budget SET TAG 'sensitive\_data';

ALTER TABLE enterprise\_data\_catalog.it\_data.incidents

ALTER COLUMN ResolutionTime SET TAG 'sensitive\_data';

3. Data Profiling:

SELECT

AVG(Budget) AS AvgBudget,

MIN(Budget) AS MinBudget,

MAX(Budget) AS MaxBudget

FROM enterprise\_data\_catalog.marketing\_data.campaigns;

SELECT ShippingStatus, COUNT(\*) AS OrderCount

FROM enterprise\_data\_catalog.operations\_data.orders

GROUP BY ShippingStatus;

**Task 3: Data Lineage and Auditing**

1.Track Data Lineage Across Schemas:

SELECT

m.CampaignID,

m.CampaignName,

m.Budget,

o.OrderID,

o.ProductID,

o.Quantity

FROM

enterprise\_data\_catalog.marketing\_data.campaigns m

JOIN

enterprise\_data\_catalog.operations\_data.orders o

ON

m.CampaignID = o.ProductID; -- Assuming CampaignID links to ProductID

2. Enable and Analyze Audit Logs:

SELECT

user\_id,

operation,

table\_name,

timestamp

FROM

audit\_logs

WHERE

table\_name LIKE 'enterprise\_data\_catalog.it\_data%'

ORDER BY

timestamp DESC;

**Task 4: Implement Fine-Grained Access Control**

1.Create User Roles and Groups:

CREATE GROUP MarketingTeam;

CREATE GROUP OperationsTeam;

CREATE GROUP ITSupportTeam;

GRANT USAGE ON SCHEMA enterprise\_data\_catalog.marketing\_data TO

MarketingTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA enterprise\_data\_catalog.marketing\_data

TO MarketingTeam;

GRANT USAGE ON SCHEMA enterprise\_data\_catalog.marketing\_data TO

OperationsTeam;

GRANT USAGE ON SCHEMA enterprise\_data\_catalog.operations\_data TO

OperationsTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA enterprise\_data\_catalog.marketing\_data

TO OperationsTeam;

GRANT SELECT ON ALL TABLES IN SCHEMA enterprise\_data\_catalog.operations\_data

TO OperationsTeam;

GRANT USAGE ON SCHEMA enterprise\_data\_catalog.it\_data TO ITSupportTeam;

GRANT SELECT, UPDATE ON TABLE enterprise\_data\_catalog.it\_data.incidents TO

ITSupportTeam;

2. Implement Column-Level Security:

REVOKE SELECT ON COLUMN Budget FROM ALL;

GRANT SELECT (Budget) ON TABLE enterprise\_data\_catalog.marketing\_data.campaigns

TO MarketingTeam;

3. Row-Level Security:

CREATE OR REPLACE ROW ACCESS POLICY operations\_team\_policy

ON enterprise\_data\_catalog.operations\_data.orders

FOR SELECT

USING (Department = CURRENT\_USER());

ALTER TABLE enterprise\_data\_catalog.operations\_data.orders

SET ROW ACCESS POLICY operations\_team\_policy;

**Task 5: Data Governance and Quality Enforcement**

1.Set Data Quality Rules:

ALTER TABLE enterprise\_data\_catalog.marketing\_data.campaigns

ADD CONSTRAINT chk\_campaign\_budget CHECK (Budget > 0);

ALTER TABLE enterprise\_data\_catalog.operations\_data.orders

ADD CONSTRAINT chk\_shipping\_status CHECK (ShippingStatus IN ('Pending', 'Shipped',

'Delivered'));

ALTER TABLE enterprise\_data\_catalog.it\_data.incidents

ADD CONSTRAINT chk\_resolution\_time CHECK (ResolutionTime >= 0);

2. Apply Delta Lake Time Travel:

DESCRIBE HISTORY enterprise\_data\_catalog.operations\_data.orders;

SELECT \* FROM enterprise\_data\_catalog.operations\_data.orders VERSION AS OF

<version\_number>;

RESTORE TABLE enterprise\_data\_catalog.operations\_data.orders TO VERSION AS OF

<version\_number>;

**Task 6: Performance Optimization and Data Cleanup**

1.Optimize Delta Tables:

OPTIMIZE enterprise\_data\_catalog.operations\_data.orders;

OPTIMIZE enterprise\_data\_catalog.it\_data.incidents;

2. Vacuum Delta Tables:

VACUUM enterprise\_data\_catalog.operations\_data.orders RETAIN 168 HOURS;

VACUUM enterprise\_data\_catalog.it\_data.incidents RETAIN 168 HOURS;

Task 1: Raw Data Ingestion

from pyspark.sql import SparkSession

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

IntegerType

from pyspark.sql.functions import col

import os

spark = SparkSession.builder \

.appName("Weather Data Ingestion") \

.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \

.config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")

\

.getOrCreate()

# Define schema for the weather data

weather\_schema = StructType([

StructField("City", StringType(), True),

StructField("Date", DateType(), True),

StructField("Temperature", FloatType(), True),

StructField("Humidity", IntegerType(), True)

])

# Define file path for the raw data CSV

file\_path = "/path/to/weather\_data.csv"

# Check if the file exists

if not os.path.exists(file\_path):

print(f"File not found: {file\_path}")

# Log the error

with open("/path/to/logs/ingestion\_logs.txt", "a") as log\_file:

log\_file.write(f"Error: Weather data file {file\_path} does not exist\n")

else:

# Proceed to load and process the data

print(f"File found: {file\_path}")

# Load the CSV data with the defined schema

raw\_weather\_data = spark.read.csv(file\_path, header=True, schema=weather\_schema)

# Show a few rows to verify

raw\_weather\_data.show(5)

# Define Delta table path

delta\_table\_path = "/path/to/delta/weather\_data"

# Write data to Delta table

raw\_weather\_data.write.format("delta").mode("overwrite").save(delta\_table\_path)

print(f"Raw data successfully saved to Delta table at {delta\_table\_path}")

Task 2: Data Cleaning

from pyspark.sql import SparkSession

spark = SparkSession.builder \

.appName("Weather Data Cleaning") \

.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \

.config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")

\

.getOrCreate()

# Define the path to the Delta table

delta\_table\_path = "/path/to/delta/weather\_data"

# Load the raw data from the Delta table

raw\_weather\_data = spark.read.format("delta").load(delta\_table\_path)

# Show the raw data

raw\_weather\_data.show(5)

# Remove rows with missing or null values

cleaned\_weather\_data = raw\_weather\_data.dropna()

# Show the cleaned data

cleaned\_weather\_data.show(5)

# Define path for the cleaned Delta table

cleaned\_delta\_table\_path = "/path/to/delta/cleaned\_weather\_data"

# Save the cleaned data to a new Delta table

cleaned\_weather\_data.write.format("delta").mode("overwrite").save(cleaned\_delta\_table\_pat

h)

print(f"Cleaned data successfully saved to Delta table at {cleaned\_delta\_table\_path}")

Task 3: Data Transformation

from pyspark.sql import SparkSession

from pyspark.sql import functions as F

spark = SparkSession.builder \

.appName("Weather Data Transformation") \

.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \

.config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")

\

.getOrCreate()

# Define the path to the cleaned Delta table

cleaned\_delta\_table\_path = "/path/to/delta/cleaned\_weather\_data"

# Load the cleaned data from the Delta table

cleaned\_weather\_data = spark.read.format("delta").load(cleaned\_delta\_table\_path)

# Show the cleaned data

cleaned\_weather\_data.show(5)

# Calculate average temperature and humidity for each city

transformed\_data = cleaned\_weather\_data.groupBy("City").agg(

F.avg("Temperature").alias("Average\_Temperature"),

F.avg("Humidity").alias("Average\_Humidity")

)

# Show the transformed data

transformed\_data.show()

# Define path for the transformed Delta table

transformed\_delta\_table\_path = "/path/to/delta/transformed\_weather\_data"

# Save the transformed data to a new Delta table

transformed\_data.write.format("delta").mode("overwrite").save(transformed\_delta\_table\_pat

h)

print(f"Transformed data successfully saved to Delta table at

{transformed\_delta\_table\_path}")

Task 4: Create a Pipeline to Execute Notebooks

import subprocess

import logging

import os

# Set up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -

%(message)s')

# Define paths to the notebooks

notebooks = {

"Raw Data Ingestion": "/path/to/Raw\_Data\_Ingestion\_Notebook.ipynb",

"Data Cleaning": "/path/to/Data\_Cleaning\_Notebook.ipynb",

"Data Transformation": "/path/to/Data\_Transformation\_Notebook.ipynb"

}

# Function to execute a notebook

def execute\_notebook(notebook\_path):

try:

# Execute the notebook using a command-line tool (e.g., nbconvert or databricks-cli)

result = subprocess.run(['databricks', 'notebook', 'run', '--path', notebook\_path], check=True)

logging.info(f"Successfully executed {notebook\_path}")

return True

except Exception as e:

logging.error(f"Failed to execute {notebook\_path}: {e}")

return False

# Main pipeline execution

def run\_pipeline():

for name, path in notebooks.items():

# Check if the notebook file exists

if not os.path.exists(path):

logging.error(f"Notebook file not found: {path}")

break

# Execute the notebook

success = execute\_notebook(path)

if not success:

logging.error(f"Pipeline failed at step: {name}")

break

else:

logging.info("Pipeline executed successfully!")

if \_\_name\_\_ == "\_\_main\_\_":

run\_pipeline()

Bonus Task: Error Handling

import os

from datetime import datetime

# Function to log errors to a separate file or database

def log\_error(step\_name, error\_message):

error\_log = {

"timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),

"step": step\_name,

"error": error\_message

}

# Log to a file (could also be a database insert)

with open("error\_log.csv", "a") as f:

f.write(f"{error\_log['timestamp']},{error\_log['step']},{error\_log['error']}\n")

logging.error(f"Error logged for {step\_name}: {error\_message}")

Task 1: Raw Data Ingestion

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

IntegerType

from pyspark.sql import SparkSession

import logging

# Set up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -

%(message)s',

handlers=[logging.FileHandler("/dbfs/tmp/raw\_data\_ingestion\_errors.log"),

logging.StreamHandler()])

# Define the schema

schema = StructType([

StructField("City", StringType(), True),

StructField("Date", DateType(), True),

StructField("Temperature", FloatType(), True),

StructField("Humidity", IntegerType(), True)

])

# Define file path

file\_path = "/dbfs/tmp/weather\_data.csv"

try:

# Load the CSV file into a DataFrame

weather\_df = spark.read.csv(file\_path, schema=schema, header=True)

# Log success

logging.info("CSV file loaded successfully.")

# Display the DataFrame (optional)

display(weather\_df)

# Write the DataFrame to a Delta table

delta\_table\_path = "/mnt/delta/weather\_data"

weather\_df.write.format("delta").mode("overwrite").save(delta\_table\_path)

# Log success

logging.info("Data successfully written to Delta table.")

except Exception as e:

# Handle missing file or other errors

error\_message = f"Error loading CSV file from {file\_path}: {str(e)}"

logging.error(error\_message)

# save error logs to a Delta table or file

error\_log\_df = spark.createDataFrame([(error\_message,)], ["Error"])

error\_log\_df.write.format("delta").mode("append").save("/mnt/delta/error\_logs")

Task 2: Data Cleaning

from pyspark.sql import SparkSession

import logging

# Set up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -

%(message)s',

handlers=[logging.FileHandler("/dbfs/tmp/data\_cleaning\_errors.log"),

logging.StreamHandler()])

# Step 1: Load the raw data from Delta table

delta\_table\_path = "/mnt/delta/weather\_data"

raw\_weather\_df = spark.read.format("delta").load(delta\_table\_path)

# Step 2: Remove rows with null values

cleaned\_weather\_df = raw\_weather\_df.na.drop(subset=["Temperature", "Humidity"])

# Step 3: Filter out rows with invalid Temperature and Humidity values

cleaned\_weather\_df = cleaned\_weather\_df.filter((cleaned\_weather\_df.Temperature >= -50)

&

(cleaned\_weather\_df.Temperature <= 60) &

(cleaned\_weather\_df.Humidity >= 0) &

(cleaned\_weather\_df.Humidity <= 100))

# Step 4: Save the cleaned data to a new Delta table

cleaned\_delta\_table\_path = "/mnt/delta/cleaned\_weather\_data"

cleaned\_weather\_df.write.format("delta").mode("overwrite").save(cleaned\_delta\_table\_path)

# Log success

logging.info("Cleaned weather data successfully saved to new Delta table.")

Task 3: Data Transformation

from pyspark.sql.functions import avg

# Calculate the average temperature and humidity for each city

transformed\_weather\_df = cleaned\_weather\_df.groupBy("City").agg(

avg("Temperature").alias("AverageTemperature"),

avg("Humidity").alias("AverageHumidity")

)

# Display the transformed data (optional)

display(transformed\_weather\_df)

# Define the path to the new Delta table for transformed data

transformed\_delta\_table\_path = "/mnt/delta/transformed\_weather\_data"

# Write the transformed data to a new Delta table

transformed\_weather\_df.write.format("delta").mode("overwrite").save(transformed\_delta\_ta

ble\_path)

# Log the successful save operation

import logging

logging.info("Transformed weather data (average temperature and humidity) successfully

saved to Delta table.")

Task 4: Build and Run a Pipeline

# Import necessary libraries

import logging

from databricks import notebook

# Set up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s',

handlers=[logging.FileHandler("/dbfs/tmp/pipeline\_logs.log"), logging.StreamHandler()])

def run\_notebook(notebook\_path):

try:

# Run the notebook

notebook.run(notebook\_path, timeout\_seconds=600)

logging.info(f"Successfully executed notebook: {notebook\_path}")

except Exception as e:

logging.error(f"Error executing notebook {notebook\_path}: {str(e)}")

raise

# Step 1: Run Data Ingestion Notebook

try:

run\_notebook("/path/to/raw\_data\_ingestion\_notebook")

except Exception as e:

logging.error("Data Ingestion Failed. Terminating Pipeline.")

raise

# Step 2: Run Data Cleaning Notebook

try:

run\_notebook("/path/to/data\_cleaning\_notebook")

except Exception as e:

logging.error("Data Cleaning Failed. Terminating Pipeline.")

raise

# Step 3: Run Data Transformation Notebook

try:

run\_notebook("/path/to/data\_transformation\_notebook")

except Exception as e:

logging.error("Data Transformation Failed. Terminating Pipeline.")

raise

logging.info("Pipeline execution completed successfully.")

Task 1: Customer Data Ingestion

from pyspark.sql import SparkSession

from pyspark.sql.utils import AnalysisException

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

# Path to the CSV file

file\_path = "/path/to/customer\_transactions.csv"

try:

# Load CSV file into a DataFrame

customer\_df = spark.read.option("header", True).csv(file\_path)

# Write DataFrame into a Delta table

customer\_df.write.format("delta").mode("overwrite").save("/delta/customer\_transactions")

except AnalysisException as e:

print(f"File not found or error while loading the file: {e}")

Task 2: Data Cleaning

# Drop duplicates

cleaned\_df = customer\_df.dropDuplicates()

# Handle nulls in the TransactionAmount column by filling with 0

cleaned\_df = cleaned\_df.fillna({'TransactionAmount': 0})

# Write cleaned data to a new Delta table

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_customer\_transacti

ons")

Task 3: Data Aggregation Aggregate

aggregated\_df =

cleaned\_df.groupBy("ProductCategory").sum("TransactionAmount").alias("TotalTransaction

Amount")

# Save aggregated data to a Delta table

aggregated\_df.write.format("delta").mode("overwrite").save("/delta/aggregated\_customer\_tra

nsactions")

Task 4: Pipeline Creation

def ingest\_data(file\_path):

try:

customer\_df = spark.read.option("header", True).csv(file\_path)

customer\_df.write.format("delta").mode("overwrite").save("/delta/customer\_transactions")

return customer\_df

except Exception as e:

print(f"Error during ingestion: {e}")

return None

def clean\_data(df):

try:

df\_cleaned = df.dropDuplicates().fillna({'TransactionAmount': 0})

df\_cleaned.write.format("delta").mode("overwrite").save("/delta/cleaned\_customer\_transacti

ons")

return df\_cleaned

except Exception as e:

print(f"Error during cleaning: {e}")

return None

def aggregate\_data(df\_cleaned):

try:

df\_aggregated = df\_cleaned.groupBy("ProductCategory").sum("TransactionAmount")

df\_aggregated.write.format("delta").mode("overwrite").save("/delta/aggregated\_customer\_tra

nsactions")

except Exception as e:

print(f"Error during aggregation: {e}")

# File path to the raw data

file\_path = "/path/to/customer\_transactions.csv"

# Execute the pipeline

df\_raw = ingest\_data(file\_path)

if df\_raw is not None:

df\_cleaned = clean\_data(df\_raw)

if df\_cleaned is not None:

aggregate\_data(df\_cleaned)

Task 5: Data Validation

# Get the total transaction count from raw data

total\_transactions\_raw = df\_cleaned.count()

# Get the total transaction amount from the aggregated data

df\_aggregated = spark.read.format("delta").load("/delta/aggregated\_customer\_transactions")

total\_transactions\_aggregated =

df\_aggregated.selectExpr("sum(TransactionAmount)").collect()[0][0]

if total\_transactions\_raw == total\_transactions\_aggregated:

print("Data validation passed!")

else:

print("Data validation failed!")

Task 1: Product Inventory Data Ingestion

from pyspark.sql import SparkSession

from pyspark.sql.utils import AnalysisException

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

# Path to the CSV file

file\_path = "/path/to/product\_inventory.csv"

try:

# Load CSV into a DataFrame

inventory\_df = spark.read.option("header", True).csv(file\_path)

# Write DataFrame into a Delta table

inventory\_df.write.format("delta").mode("overwrite").save("/delta/product\_inventory")

except AnalysisException as e:

print(f"File not found or error loading file: {e}")

Task 2: Data Cleaning

# Remove rows with negative StockQuantity

cleaned\_inventory\_df = inventory\_df.filter(inventory\_df.StockQuantity >= 0)

# Fill null values in StockQuantity and Price columns

cleaned\_inventory\_df = cleaned\_inventory\_df.fillna({'StockQuantity': 0, 'Price': 0})

# Write cleaned data to a new Delta table

cleaned\_inventory\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_product\_

inventory")

Task 3: Inventory Analysis

from pyspark.sql.functions import col

# Calculate total stock value

inventory\_analysis\_df = cleaned\_inventory\_df.withColumn("TotalStockValue",

col("StockQuantity") \* col("Price"))

# Find products that need restocking

restock\_df = cleaned\_inventory\_df.filter(cleaned\_inventory\_df.StockQuantity < 100)

# Save analysis results to a Delta table

inventory\_analysis\_df.write.format("delta").mode("overwrite").save("/delta/inventory\_analys

is")

restock\_df.write.format("delta").mode("overwrite").save("/delta/restock\_products")

Task 4: Build an Inventory Pipeline

def ingest\_product\_data(file\_path):

try:

inventory\_df = spark.read.option("header", True).csv(file\_path)

inventory\_df.write.format("delta").mode("overwrite").save("/delta/product\_inventory")

return inventory\_df

except Exception as e:

print(f"Error during ingestion: {e}")

return None

def clean\_product\_data(df):

try:

cleaned\_df = df.filter(df.StockQuantity >= 0).fillna({'StockQuantity': 0, 'Price': 0})

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_product\_inventory"

)

return cleaned\_df

except Exception as e:

print(f"Error during cleaning: {e}")

return None

def analyze\_inventory(df\_cleaned):

try:

analysis\_df = df\_cleaned.withColumn("TotalStockValue", col("StockQuantity") \*

col("Price"))

restock\_df = df\_cleaned.filter(df\_cleaned.StockQuantity < 100)

analysis\_df.write.format("delta").mode("overwrite").save("/delta/inventory\_analysis")

restock\_df.write.format("delta").mode("overwrite").save("/delta/restock\_products")

except Exception as e:

print(f"Error during analysis: {e}")

# File path to the raw data

file\_path = "/path/to/product\_inventory.csv"

# Execute the pipeline

df\_raw = ingest\_product\_data(file\_path)

if df\_raw is not None:

df\_cleaned = clean\_product\_data(df\_raw)

if df\_cleaned is not None:

analyze\_inventory(df\_cleaned)

Task 5: Inventory Monitoring

from pyspark.sql import DataFrame

# Load the cleaned product inventory Delta table

df\_inventory = spark.read.format("delta").load("/delta/cleaned\_product\_inventory")

# Find products that need restocking

restock\_alert\_df = df\_inventory.filter(df\_inventory.StockQuantity < 50)

# Send an alert if any product needs restocking

if restock\_alert\_df.count() > 0:

print("ALERT: Some products need restocking!")

else:

print("All products are sufficiently stocked.")

Task 1: Employee Attendance Data Ingestion

from pyspark.sql import SparkSession

from pyspark.sql.utils import AnalysisException

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

# Path to the CSV file

file\_path = "/path/to/employee\_attendance.csv"

try:

# Load CSV into DataFrame

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

# Write the DataFrame into a Delta table

attendance\_df.write.format("delta").mode("overwrite").save("/delta/employee\_attendance")

except AnalysisException as e:

print(f"File not found or error while loading the file: {e}")

Task 2: Data Cleaning

from pyspark.sql.functions import col, unix\_timestamp

# Remove rows with null or invalid CheckInTime or CheckOutTime

cleaned\_df = attendance\_df.filter((col("CheckInTime").isNotNull()) &

(col("CheckOutTime").isNotNull()))

# Calculate HoursWorked based on CheckInTime and CheckOutTime

cleaned\_df = cleaned\_df.withColumn("CalculatedHoursWorked",

(unix\_timestamp("CheckOutTime", 'HH:mm') - unix\_timestamp("CheckInTime", 'HH:mm'))

/ 3600)

# Write cleaned data to a new Delta table

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_employee\_attendan

ce")

Task 3: Attendance Summary

from pyspark.sql.functions import month, sum

# Calculate total hours worked by each employee for the current month

current\_month = 3

summary\_df = cleaned\_df.filter(month(col("Date")) == current\_month) \

.groupBy("EmployeeID") \

.agg(sum("CalculatedHoursWorked").alias("TotalHoursWorked"))

# Find employees who worked overtime (more than 8 hours on any given day)

overtime\_df = cleaned\_df.filter(col("CalculatedHoursWorked") > 8)

# Save the summary and overtime data to Delta tables

summary\_df.write.format("delta").mode("overwrite").save("/delta/employee\_attendance\_sum

mary")

overtime\_df.write.format("delta").mode("overwrite").save("/delta/employee\_overtime")

Task 4: Create an Attendance Pipeline

def ingest\_attendance\_data(file\_path):

try:

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

attendance\_df.write.format("delta").mode("overwrite").save("/delta/employee\_attendance")

return attendance\_df

except Exception as e:

print(f"Error during ingestion: {e}")

return None

def clean\_attendance\_data(df):

try:

cleaned\_df = df.filter((col("CheckInTime").isNotNull()) &

(col("CheckOutTime").isNotNull()))

cleaned\_df = cleaned\_df.withColumn("CalculatedHoursWorked",

(unix\_timestamp("CheckOutTime", 'HH:mm') - unix\_timestamp("CheckInTime", 'HH:mm'))

/ 3600)

cleaned\_df.write.format("delta").mode("overwrite").save("/delta/cleaned\_employee\_attendan

ce")

return cleaned\_df

except Exception as e:

print(f"Error during cleaning: {e}")

return None

def summarize\_attendance(df\_cleaned):

try:

summary\_df =

df\_cleaned.groupBy("EmployeeID").agg(sum("CalculatedHoursWorked").alias("TotalHours

Worked"))

overtime\_df = df\_cleaned.filter(col("CalculatedHoursWorked") > 8)

summary\_df.write.format("delta").mode("overwrite").save("/delta/employee\_attendance\_sum

mary")

overtime\_df.write.format("delta").mode("overwrite").save("/delta/employee\_overtime")

except Exception as e:

print(f"Error during summarization: {e}")

# File path to the raw data

file\_path = "/path/to

Task 5: Time Travel with Delta Lake

# Assuming you want to roll back the attendance logs

spark.sql("CREATE OR REPLACE TABLE attendance\_logs AS SELECT \* FROM

delta.`/mnt/delta/attendance\_logs` VERSION AS OF <version\_number>")

spark.sql("DESCRIBE HISTORY delta.`/mnt/delta/attendance\_logs`").show(truncate=False)