**Assignment 4(Databricks)**

**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_table_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_transactions")
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_attendance")
```

**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_summary")
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)