**Lab Exercise 10- Complex Data Processing Pipeline Using MySQL Database in Metaflow**

In this exercise, you will build a more complex data processing pipeline using MySQL as your data source. The flow will involve multiple steps to:

* Retrieve data from a MySQL database.
* Perform data transformations.
* Filter and aggregate data.
* Write the processed data back into the MySQL database.

**Objective**

By the end of this exercise, you will:

* Build a multi-step pipeline that integrates with a MySQL database.
* Process data through various transformations and aggregations.
* Perform database writes in the pipeline.

**Prerequisites**

1. **MySQL Database**: Ensure that you have access to a MySQL instance and a sample table with data.
2. **MySQL Connector**: Install the required Python package to connect to MySQL.

To install the MySQL connector:

pip install mysql-connector-python

**Step 1: Create the Database and Table**

First, create a table in your MySQL database with some sample data. In this case, we will create a table named sales with columns for sales data.

CREATE DATABASE sales\_db;

USE sales\_db;

CREATE TABLE sales (

id INT AUTO\_INCREMENT PRIMARY KEY,

product VARCHAR(50),

quantity INT,

price DECIMAL(10, 2),

sale\_date DATE

);

INSERT INTO sales (product, quantity, price, sale\_date) VALUES

('Laptop', 5, 1200.00, '2023-09-01'),

('Headphones', 20, 100.00, '2023-09-03'),

('Keyboard', 15, 50.00, '2023-09-05'),

('Monitor', 7, 300.00, '2023-09-07'),

('Mouse', 30, 25.00, '2023-09-09');

**Step 2: Build the Complex Pipeline Flow**

Now, let’s build a Metaflow pipeline that:

1. Reads the data from the sales table.
2. Filters out records based on a condition (e.g., sales over a certain amount).
3. Aggregates the data to calculate total sales.
4. Writes the results back into a new table in the database.

Create a file named complex\_mysql\_pipeline.py with the following content:

from metaflow import FlowSpec, step

import mysql.connector

from datetime import datetime

class ComplexMySQLPipelineFlow(FlowSpec):

@step

def start(self):

"""Initialize the database connection and retrieve data."""

print("Connecting to MySQL database...")

# MySQL connection configuration

self.connection\_config = {

'host': 'your\_mysql\_host', # Replace with your MySQL host

'user': 'your\_mysql\_user', # Replace with your MySQL user

'password': 'your\_mysql\_password', # Replace with your MySQL password

'database': 'sales\_db' # Name of the database

}

self.min\_sales\_value = 1000.00 # Filter for sales above this value

self.next(self.query\_data)

@step

def query\_data(self):

"""Retrieve sales data from the MySQL database."""

print("Querying sales data from MySQL...")

# Establish a connection to the database

conn = mysql.connector.connect(\*\*self.connection\_config)

cursor = conn.cursor()

# Query to fetch data from the sales table

cursor.execute("SELECT product, quantity, price, sale\_date FROM sales")

self.sales\_data = cursor.fetchall()

# Close the connection

cursor.close()

conn.close()

print(f"Retrieved {len(self.sales\_data)} records from the sales table.")

self.next(self.filter\_data)

@step

def filter\_data(self):

"""Filter sales records that exceed a minimum sales value."""

print(f"Filtering sales records with value greater than {self.min\_sales\_value}...")

self.filtered\_data = [

record for record in self.sales\_data

if record[1] \* record[2] > self.min\_sales\_value # quantity \* price > min\_sales\_value

]

print(f"Filtered down to {len(self.filtered\_data)} records.")

self.next(self.aggregate\_data)

@step

def aggregate\_data(self):

"""Aggregate data to calculate total sales value."""

print("Aggregating data to calculate total sales...")

self.total\_sales\_value = sum([record[1] \* record[2] for record in self.filtered\_data])

print(f"Total sales value: {self.total\_sales\_value}")

self.next(self.write\_results)

@step

def write\_results(self):

"""Write the filtered and aggregated results back into the MySQL database."""

print("Writing results to the MySQL database...")

conn = mysql.connector.connect(\*\*self.connection\_config)

cursor = conn.cursor()

# Create a new table to store aggregated results

cursor.execute("""

CREATE TABLE IF NOT EXISTS sales\_summary (

id INT AUTO\_INCREMENT PRIMARY KEY,

total\_sales\_value DECIMAL(10, 2),

processed\_at DATETIME

)

""")

# Insert the aggregated results

insert\_query = """

INSERT INTO sales\_summary (total\_sales\_value, processed\_at)

VALUES (%s, %s)

"""

cursor.execute(insert\_query, (self.total\_sales\_value, datetime.now()))

# Commit the changes and close the connection

conn.commit()

cursor.close()

conn.close()

print(f"Results successfully written to the database.")

self.next(self.end)

@step

def end(self):

"""Final step: Complete the flow."""

print("Flow completed.")

print(f"Total sales value written to the database: {self.total\_sales\_value}")

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

ComplexMySQLPipelineFlow()

**Step 3: Run the Flow**

To run the flow, execute the following command:

python complex\_mysql\_pipeline.py run

**Step 4: Explanation of Steps**

1. **start step**: Initializes the database connection and sets the minimum sales value filter.
2. **query\_data step**: Connects to the MySQL database and retrieves data from the sales table.
3. **filter\_data step**: Filters out records where the total sale value (quantity \* price) is below a certain threshold.
4. **aggregate\_data step**: Calculates the total sales value of the filtered records.
5. **write\_results step**: Creates a new table sales\_summary in the database (if it doesn't exist) and writes the aggregated result into this table.
6. **end step**: Outputs the final result and indicates that the flow has completed.

**Step 5: Testing and Extending**

1. **Test the Flow**: You can run the flow multiple times to see how it processes the data and saves the aggregated results into the sales\_summary table.
2. **View Results**: Query the sales\_summary table to view the results:

SELECT \* FROM sales\_summary;

1. **Extend the Flow**:
   * Add additional transformations, such as grouping data by product or month.
   * Implement error handling using Metaflow's @retry decorator for database connection failures.
   * Incorporate parallelism for larger datasets by splitting the data into chunks and processing it in parallel.

**Conclusion**

In this lab exercise, you built a complex data processing pipeline using Metaflow and MySQL. You retrieved sales data, performed filtering and aggregation, and saved the processed results back into the database. This exercise demonstrated how to integrate MySQL with Metaflow for more complex data processing tasks. Feel free to extend this pipeline further based on your requirements!