SRM Institute of Science and Technology
Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204
Department of Computer Applications
Circular – 2023-24
BCA DS 3rd Sem

Data Engineering for Enterprise (UDS23301J)

Lab Manual

Lab 1

Title: Setup a Simple Data Engineering Development Infrastructure in MySQL Opensource

Aim: To set up a basic data engineering development environment using MySQL as the database management system.

Procedure:

1. Download and install MySQL Community Server from the official website.
2. Install MySQL Workbench for GUI-based interaction with the server.
3. Create a new database named `DataEngineeringLab`.
4. Set up a user with necessary privileges for the database.

Source Code:

```
CREATE DATABASE DataEngineeringLab;
CREATE USER 'labuser'@'localhost' IDENTIFIED BY 'labpassword';
GRANT ALL PRIVILEGES ON DataEngineeringLab.* TO 'labuser'@'localhost';
FLUSH PRIVILEGES;
```

Input: No input required for setup.

Expected Output:

A new database `DataEngineeringLab` is created.

User `labuser` with full access to the database is created.

Lab 2

Title: Create Tables with Appropriate Columns in MySQL

Aim: To create the Customer, Product, and Sales Order tables in the MySQL database with appropriate schema definitions.

Procedure:

1. Use the `DataEngineeringLab` database.
2. Define the schema for each table with appropriate data types and constraints.
3. Execute the SQL commands to create the tables.

Source Code:

```
USE DataEngineeringLab;

CREATE TABLE Customer (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(100),
    Email VARCHAR(100)
);

CREATE TABLE Product (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    UnitPrice DECIMAL(10, 2)
);

CREATE TABLE SalesOrder (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    ProductID INT,
    Quantity INT,
    OrderDate DATE,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),
    FOREIGN KEY (ProductID) REFERENCES Product(ProductID)
);
```

Input: No runtime input.

Expected Output: Three tables `Customer`, `Product`, and `SalesOrder` created in the database with defined schema.

Lab 3

Title: Load Data from CSV, TEXT File, and Google Drive into Tables

Aim: To load data from external sources such as CSV, TEXT files, and Google Drive into MySQL tables.

Procedure:

1. Prepare CSV/TXT files for each table with appropriate column headers.
2. Use MySQL `LOAD DATA INFILE` or Python scripts to load data into MySQL.
3. Grant necessary file privileges in MySQL.

Source Code:

```
LOAD DATA INFILE '/path/to/customer.csv'
INTO TABLE Customer
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

(Repeat for Product and SalesOrder tables)

Input: CSV or TEXT files containing data.

Expected Output: Customer, Product, and SalesOrder tables are populated with data from the respective files.

Lab 4

Title: Validate Loaded Data in MySQL Tables

Aim: To validate uniqueness, null values, and data types in Customer, Product, and SalesOrder tables.

Procedure:

1. Check if `CustomerID`, `ProductID`, and `OrderID` are unique and not null.
2. Check if data types of Customer Names and Product Names are CHAR or VARCHAR.

Source Code:

```
SELECT CustomerID, COUNT(*) FROM Customer GROUP BY CustomerID HAVING COUNT(*) >
1;
SELECT ProductID, COUNT(*) FROM Product GROUP BY ProductID HAVING COUNT(*) > 1;
SELECT OrderID, COUNT(*) FROM SalesOrder GROUP BY OrderID HAVING COUNT(*) > 1;

SELECT * FROM Customer WHERE CustomerID IS NULL;
SELECT * FROM Product WHERE ProductID IS NULL;
SELECT * FROM SalesOrder WHERE OrderID IS NULL;

-- Data type check is manual: verify using DESCRIBE
DESCRIBE Customer;
DESCRIBE Product;
```

Input: No input required.

Expected Output:

No duplicate or NULL values in IDs.

Data types correctly displayed.

Lab 5

Title: Validate Email Format, Nulls, and Duplicates in Customer Table

Aim: To validate that email addresses in the Customer table are correctly formatted and to check for nulls and duplicates.

Procedure:

1. Check if email format matches a basic pattern.
2. Identify null values in Email field.
3. Identify duplicate Email entries.

Source Code:

```
SELECT * FROM Customer WHERE Email NOT LIKE '%_@__%.__%';
SELECT * FROM Customer WHERE Email IS NULL;
SELECT Email, COUNT(*) FROM Customer GROUP BY Email HAVING COUNT(*) > 1;
```

Input: No input required.

Expected Output:

Email addresses properly formatted.

No null values in Email.

No duplicate email addresses.

Lab 6

Title:
Join Customer and Product Tables using INNER JOIN

Aim:
To write a Python SQL query that merges the Customer and Product tables using INNER JOIN.

Procedure:

1. Connect to MySQL using Python with `sqlalchemy` and `pandas`.
2. Write an INNER JOIN query to merge Customer and Product tables on `CustomerID`.
3. Fetch and display the result.

Source Code:

```
import pandas as pd
from sqlalchemy import create_engine

engine =
create_engine("mysql+pymysql://username:password@localhost/data_engineering_lab"
)
query = """
SELECT * FROM Customer
INNER JOIN SalesOrder ON Customer.CustomerID = SalesOrder.CustomerID
INNER JOIN Product ON SalesOrder.ProductID = Product.ProductID
"""
result = pd.read_sql(query, con=engine)
print(result)
```

Input:
Database with Customer, Product, and SalesOrder tables.

Expected Output:
A joined table showing merged information from all three tables.

Lab 7

Title:
Use LEFT JOIN and RIGHT JOIN in Python SQL Queries

Aim:
To perform LEFT JOIN and RIGHT JOIN operations on the SalesOrder and Product tables using Python.

Procedure:

1. Connect to MySQL using `sqlalchemy`.
2. Perform LEFT JOIN of Customer with SalesOrder.
3. Perform RIGHT JOIN of the result with Product.

Source Code:

```
query = """
SELECT * FROM Customer
LEFT JOIN SalesOrder ON Customer.CustomerID = SalesOrder.CustomerID
RIGHT JOIN Product ON SalesOrder.ProductID = Product.ProductID
"""
result = pd.read_sql(query, con=engine)
print(result)
```

Input:
Data in Customer, Product, and SalesOrder tables.

Expected Output:
Output will show combined data even where SalesOrder entries are missing.

Lab 8

Title:
Update Data in Table Using Python

Aim:
To perform update operations on MySQL tables using Python based on specified logic.

Procedure:

1. Connect to the database.
2. Use SQL `UPDATE` queries in Python to modify the records.

Source Code:

```
with engine.connect() as conn:
    conn.execute("""
        UPDATE Product
        SET Price = 40
        WHERE ProductName = 'Coolers';
    """)

    conn.execute("""
        UPDATE SalesOrder
        SET Quantity = 5
        WHERE CustomerID = (SELECT CustomerID FROM Customer WHERE CustomerName =
'Alfred');
    """)
```

Input:
Product = 'Coolers'; Customer = 'Alfred'

Expected Output:
Updated Unit Price and Quantity as per the logic.

Lab 9

Title:
Calculate Revenue and Apply Discount Using Python

Aim:
To calculate revenue as Unit Price * Quantity and apply a discount for specific products.

Procedure:

1. Use Python with SQL query to calculate revenue.
2. Apply 10% discount where Product = 'Toothpaste'.

Source Code:

```
query = """
SELECT p.ProductName, s.Quantity, (p.Price * s.Quantity) AS Revenue,
CASE
    WHEN p.ProductName = 'Toothpaste' THEN (p.Price * s.Quantity * 0.9)
    ELSE (p.Price * s.Quantity)
END AS DiscountedRevenue
FROM SalesOrder s
JOIN Product p ON s.ProductID = p.ProductID;
"""
result = pd.read_sql(query, con=engine)
print(result)
```

Input:
Product table, SalesOrder table

Expected Output:
Revenue with 10% discount where applicable

Lab 10

Title:
Transform Customer Names Using Python

Aim:
To split and transform Customer names and format them to upper-case properly.

Procedure:

1. Use Python SQL query to extract first and last name.
2. Concatenate and apply formatting.

Source Code:

```
query = """
SELECT
    CONCAT(UPPER(SUBSTRING_INDEX(CustomerName, ' ', -1)), ', ',
           INITCAP(SUBSTRING_INDEX(CustomerName, ' ', 1))) AS FullName
FROM Customer;
"""
result = pd.read_sql(query, con=engine)
print(result)
```

Input:
Customer table with full name field

Expected Output:
Customer names in formatted upper-case (Last, First)

Lab 11: Export table data into CSV format using Python

Aim

Export data from a MySQL table into a CSV file using Python. This involves connecting to the database, running a `SELECT` query, fetching the results, and writing them to CSV (for example using the `csv` module or Pandas' `to_csv()` method)risingwave.com.

Procedure

Connect to MySQL: Use a library like `mysql-connector-python` to connect to the database (`mysql.connector.connect(host, user, password, database)` w3schools.com).

Fetch data: Execute a `SELECT * FROM table_name` query with a cursor. Retrieve all rows (`cursor.fetchall()`) and column names (`cursor.description`).

Write CSV: Open a CSV file in write mode. Write the header row (column names) followed by each data row. You can use Python's built-in `csv` module or Pandas. For example, convert the result to a Pandas DataFrame and use `df.to_csv('output.csv', index=False)` risingwave.com.

Verify: Check the output CSV file to ensure correct formatting and data completeness.

Source Code

```
import mysql.connector
import csv

# 1. Connect to MySQL database
conn = mysql.connector.connect(
    host="localhost", user="user", password="password", database="mydatabase"
)
cursor = conn.cursor()

# 2. Execute query
cursor.execute("SELECT id, name, age FROM students")
rows = cursor.fetchall()
headers = [i[0] for i in cursor.description]  # Get column names

# 3. Write to CSV
with open("students.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(headers)            # write header
    writer.writerows(rows)              # write data rows

conn.close()
print(f"{len(rows)} rows exported to students.csv")
```

Input

A MySQL table **`students`** with sample data. For example:

id	name	age
1	Alice	20

id	name	age
2	Bob	22
3	Charlie	19

Expected Output

A file `students.csv` with the following contents:

```
id,name,age
1,Alice,20
2,Bob,22
3,Charlie,19
```

The program should print something like:

```
3 rows exported to students.csv
```

Lab 12: Create Timestamp using Python

Aim

Generate the current date and time ("timestamp") in Python and format it for use (e.g., inserting into a database). This typically uses the `datetime` module to get `datetime.now()` and format it with `strftime`stackoverflow.com.

Procedure

Import datetime: Use `from datetime import datetime`.

Get current time: Call `datetime.now()` to get the current date and time.

Format timestamp: Use `strftime("%Y-%m-%d %H:%M:%S")` (or similar) to produce a string like `2025-05-19 15:30:00`. This format is commonly used for SQL `DATETIME` fieldsstackoverflow.com.

Use timestamp: The timestamp string can be printed or inserted into a MySQL `TIMESTAMP` column in an `INSERT` statement.

Source Code

```
from datetime import datetime

# Create current timestamp
timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
print("Current Timestamp:", timestamp)
```

Input

No external input; the script uses the current system date/time.

Expected Output

The current date and time in the format `YYYY-MM-DD HH:MM:SS`. For example:

```
Current Timestamp: 2025-05-19 15:30:00
```

This string can then be used in SQL inserts or displayed as neededstackoverflow.com.

Lab 13: Access and insert data from API into MySQL using Python

Aim

Fetch data from a REST API and insert the results into a MySQL table using Python. This involves sending an HTTP GET request (e.g., with `requests`), parsing the JSON response, and using a database connector to insert rows into MySQL.

Procedure

Get API data: Use the `requests` library to send a GET request to the API endpoint (e.g., `requests.get(url)`) and parse JSON via `response.json()`medium.com.

Connect to MySQL: Establish a database connection with `mysql.connector.connect()` or similar.

Create table (if needed): Define a table with columns matching the JSON fields.

Transform data: Optionally process the JSON (e.g., extract needed fields).

Insert data: Use SQL `INSERT` statements or `executemany()` to add records. Construct tuples of values and execute an INSERT query for each record. For multiple rows, use `cursor.executemany(sql, data_tuples)` for efficiencymedium.com.

Commit and close: Commit the transaction and close the connection.

Source Code

```
import requests
import mysql.connector

# 1. Fetch data from API
url = "https://jsonplaceholder.typicode.com/users"
response = requests.get(url)
data = response.json()  # list of dicts

# 2. Connect to MySQL
conn = mysql.connector.connect(
    host="localhost", user="user", password="password", database="mydatabase"
)
cursor = conn.cursor()

# 3. (Re)create table for users
cursor.execute("DROP TABLE IF EXISTS users")
cursor.execute("""
    CREATE TABLE users (
        id INT PRIMARY KEY,
        name VARCHAR(100),
        email VARCHAR(100),
        city VARCHAR(100)
    )
""")
```

```
# 4. Prepare and execute inserts
sql = "INSERT INTO users (id, name, email, city) VALUES (%s, %s, %s, %s)"
values = []
for item in data:
    # Extract fields; handle nested JSON for city
    city = item['address']['city']
    values.append((item['id'], item['name'], item['email'], city))

cursor.executemany(sql, values)  # insert all at
once:contentReference[oaicite:8]{index=8}
conn.commit()

print(f"{cursor.rowcount} records inserted into users")
conn.close()
```

Input

Example JSON data from the API endpoint. For instance, one entry from `https://jsonplaceholder.typicode.com/users`:

```
{
  "id": 1,
  "name": "Leanne Graham",
  "email": "Sincere@april.biz",
  "address": {"city": "Gwenborough", ...}
}
```

Expected Output

A `users` table in MySQL populated with the fetched data.

Console output confirming insertion, e.g.:

```
10 records inserted into users
```

(Assuming the API returned 10 user objects.)

Lab 14: Create Product Category table and insert data using Python

Aim

Create a new MySQL table called `ProductCategory` and insert sample records using Python. This demonstrates executing DDL (`CREATE TABLE`) and DML (`INSERT`) commands from Python.

Procedure

Connect to MySQL: Use `mysql.connector.connect()`.

Create table: Execute a `CREATE TABLE` statement, e.g.:

```
CREATE TABLE ProductCategory (
    id INT AUTO_INCREMENT PRIMARY KEY,
    category_name VARCHAR(50)
)
```

w3schools.com.

Prepare data: Define a list of category names (e.g., `["Electronics", "Clothing", "Furniture"]`).

Insert records: Use a parameterized `INSERT INTO` query. For multiple rows, use `cursor.executemany(sql, values)` where `values` is a list of tuples (each tuple is one row)w3schools.comw3schools.com.

Commit: Commit the transaction and close the connection.

Source Code

```
import mysql.connector

# 1. Connect to MySQL
conn = mysql.connector.connect(
    host="localhost", user="user", password="password", database="mydatabase"
)
cursor = conn.cursor()

# 2. Create ProductCategory table
cursor.execute("DROP TABLE IF EXISTS ProductCategory")
cursor.execute("""
    CREATE TABLE ProductCategory (
        id INT AUTO_INCREMENT PRIMARY KEY,
        category_name VARCHAR(50)
    )
""")

# 3. Insert sample categories
sql = "INSERT INTO ProductCategory (category_name) VALUES (%s)"
categories = [("Electronics",), ("Clothing",), ("Kitchen",)]
cursor.executemany(sql, categories)  # insert multiple
rows:contentReference[oaicite:12]{index=12}
```

```
conn.commit()
print(f"{cursor.rowcount} categories inserted")
conn.close()
```

Input

List of category names to insert, e.g.:

```
[("Electronics",), ("Clothing",), ("Kitchen",)]
```

Expected Output

A new `ProductCategory` table in MySQL with rows:

id	category_name
1	Electronics
2	Clothing
3	Kitchen

Console output:

```
3 categories inserted
```

Lab 15: Write Python SQL query to group, sort, and filter table

Aim

Use Python to execute SQL queries that group, sort, and filter data in a MySQL table. For example, use `SELECT ... GROUP BY ... ORDER BY ...` (and `WHERE` for filtering) to aggregate dataw3schools.com.

Procedure

Connect to MySQL: Use `mysql.connector`.

Write SQL query: Compose a query with `WHERE`, `GROUP BY`, and `ORDER BY`. For example, to count rows per country for a filtered set:

```
SELECT Country, COUNT(*) AS cnt
FROM Customers
WHERE Region = 'WA'
GROUP BY Country
ORDER BY cnt DESC;
```

This counts and lists customers by country in descending orderw3schools.com.

Execute query: Use `cursor.execute(query)`.

Fetch results: Use `cursor.fetchall()` and display or process them.

Optional (Pandas): Alternatively, use `pd.read_sql_query(query, connection)` to get a DataFrame with grouped results.

Source Code

```
import mysql.connector

# Connect to MySQL
conn = mysql.connector.connect(
    host="localhost", user="user", password="password", database="mydatabase"
)
cursor = conn.cursor()

# Execute a grouped, sorted, filtered query
query = ("""
    SELECT Country, COUNT(*) AS cnt
    FROM Customers
    WHERE Region = 'WA'
    GROUP BY Country
    ORDER BY cnt DESC
""")
cursor.execute(query)
results = cursor.fetchall()

# Display results
for country, count in results:
    print(f"{country}: {count}")

conn.close()
```

Input

A **Customers** table with columns including at least `Country` and `Region`. For example:

CustomerID	Country	Region
1	USA	WA
2	USA	WA
3	Canada	BC
4	USA	WA
5	Canada	WA

Expected Output

After filtering by `Region = 'WA'`, grouping by `Country`, and sorting descending by count, the printed output might be:

```
USA: 3
Canada: 1
```

This reflects that there are 3 customers in the USA region “WA” and 1 in Canada, sorted by countw3schools.com.

Citations