BIG DATA MANAGEMENT

Assignment III

Submitted To:

Dr. Dip Shankar Banerjee

Associate Professor

School Of Artificial Intelligence and Data Engineering

Indian Institute of Technology Jodhpur – Rajasthan

Submitted By:

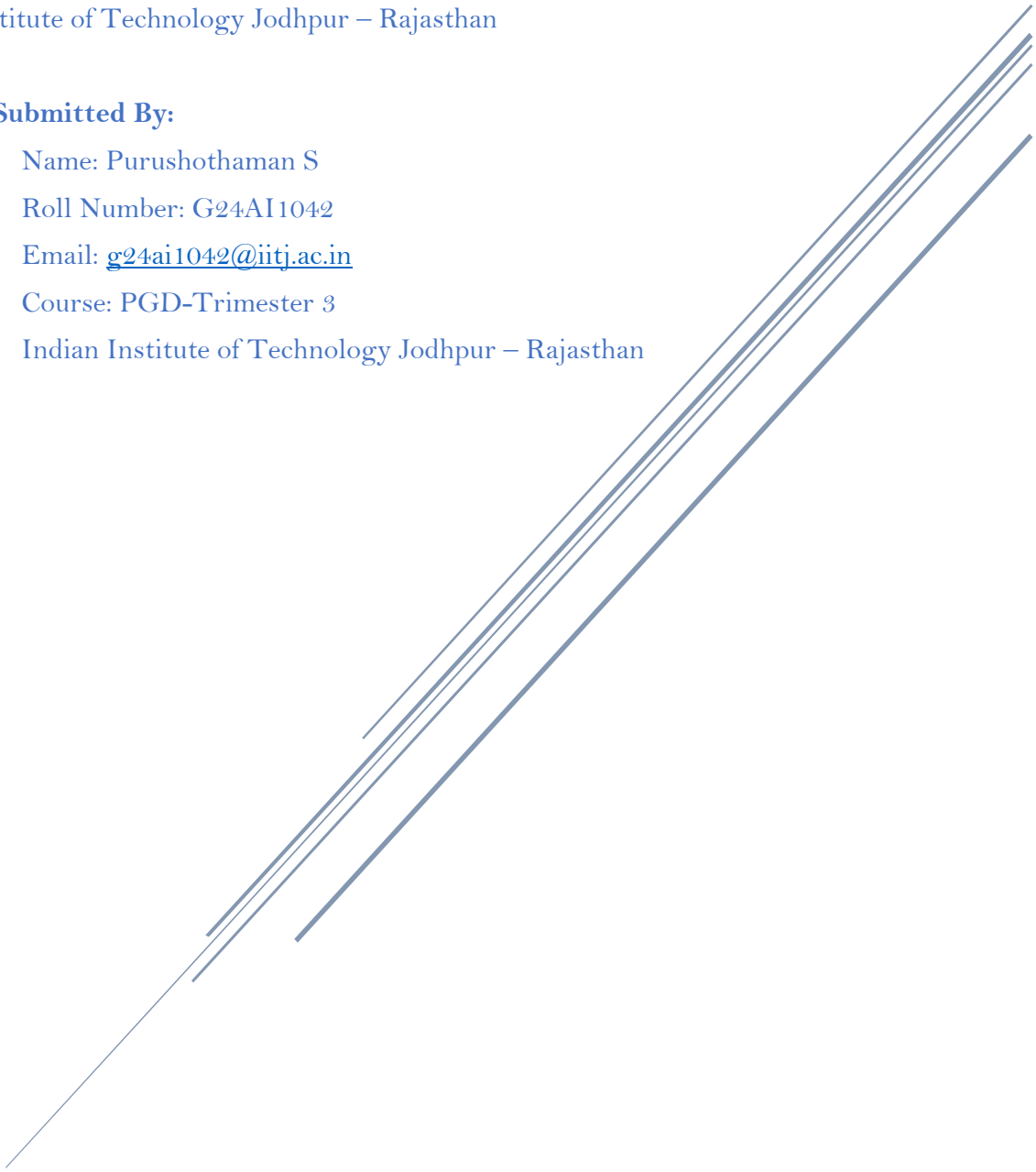
Name: Purushothaman S

Roll Number: G24AI1042

Email: g24ai1042@iitj.ac.in

Course: PGD-Trimester 3

Indian Institute of Technology Jodhpur – Rajasthan



Index

Introduction

This assignment focuses on building a Java-based interface for interacting with an Amazon RDS (Relational Database Service) instance using SQL operations. It aims to enhance understanding of database integration in enterprise applications by implementing standard database operations such as table creation, data insertion and deletion, and execution of analytical queries. Two tables, company and stock price, are used to simulate real-world data management in the financial sector, particularly tracking corporate profiles and daily stock movements.

What is RDS

RDS (Relational Database Service) is a **managed database service** provided by Amazon Web Services (AWS) that makes it easy to set up, operate, and scale relational databases in the cloud.

Key Features of RDS:

1. **Managed Service:**
   AWS handles tasks like provisioning, backups, patching, monitoring, and hardware scaling.
2. **Supports Multiple Database Engines:**
   You can choose from several popular database systems, including:
   - MySQL
   - PostgreSQL
   - MariaDB
   - Oracle
   - Microsoft SQL Server
   - Amazon Aurora (AWS's own optimized engine)
3. **Scalability:**
   You can easily scale up the compute and storage capacity without downtime.
4. **High Availability:**
   RDS supports automatic failover and multi-AZ (availability zone) deployments for resilience and uptime.
5. **Backups and Snapshots:**
   It provides automated daily backups, manual snapshots, and point-in-time recovery.
6. **Security:**
   Supports encryption at rest and in transit, VPC integration, and IAM-based access.

7. **Monitoring:**
   Integrates with Amazon CloudWatch for tracking performance and usage metrics.

How It is Used:

In your assignment, RDS is used to host a **MySQL database**. Your Java code connects to the RDS instance to:

- Create tables (company and stockprice)
- Insert test data
- Run SQL queries and data analysis

Why Use RDS?

- No need to manage database hardware or software manually
- Automated scaling, backups, and failover
- High performance with less operational overhead

Establishment of Connection Between RDS and Java Editor

The establishment of a connection between a Java application and an Amazon RDS (Relational Database Service) instance involves several key steps using JDBC (Java Database Connectivity). Here's a detailed explanation of the connection process as used in your assignment:

1. Load the JDBC Driver

Before any database interaction can take place, the application must load the JDBC driver for the target database (in this case, MySQL):

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- This line ensures that the Java application can communicate with the MySQL database by loading the MySQL JDBC driver into memory.
- The com.mysql.cj.jdbc.Driver is the class provided by MySQL Connector/J, the official JDBC driver for MySQL 8.0+.

2. Construct the JDBC URL

The JDBC URL defines the path and credentials for connecting to the database:

```
String jdbcUrl = "jdbc:mysql://" + url + "/mydb?user=" + uid + "&password=" + pw;
```

- url contains the RDS endpoint and port (e.g., iitjdb.cpmqoqquutp0.us-east-1.rds.amazonaws.com:3306).

- uid and pw are the username and password (e.g., admin, iitj@12345).
- mydb is the target database within the RDS instance.
- This URL combines everything to tell the JDBC driver where and how to connect.

3. Establish the Connection

The actual connection to the database is made using DriverManager.getConnection():

```
con = DriverManager.getConnection(jdbcUrl);
```

- The DriverManager uses the previously loaded JDBC driver to establish a TCP/IP connection to the Amazon RDS instance.
- Upon successful connection, a Connection object (con) is returned, which is then used for executing SQL statements.

4. Confirmation and Debugging

Print statements are added to confirm the connection status:

```
System.out.println("Connecting to database");
System.out.println("Connection Successful");
```

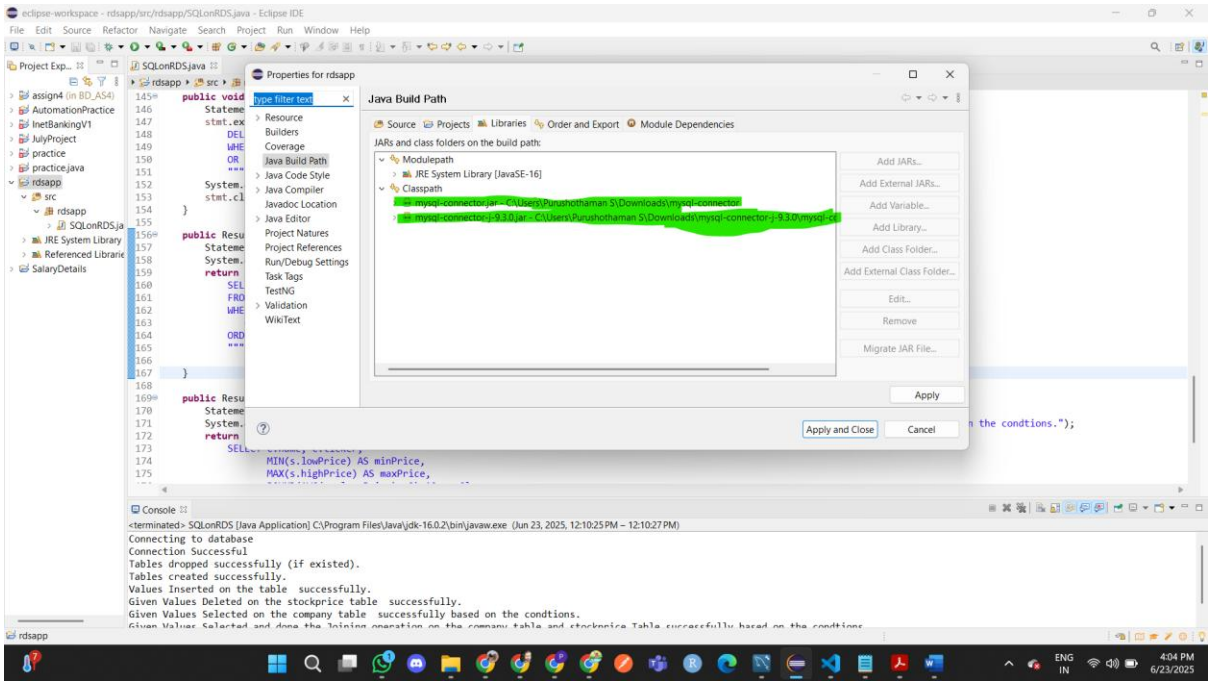
These messages provide real-time feedback to the user or developer to verify whether the connection was successfully established or if troubleshooting is needed.

5. Exception Handling

The entire process is enclosed in a try-catch block to handle potential exceptions:

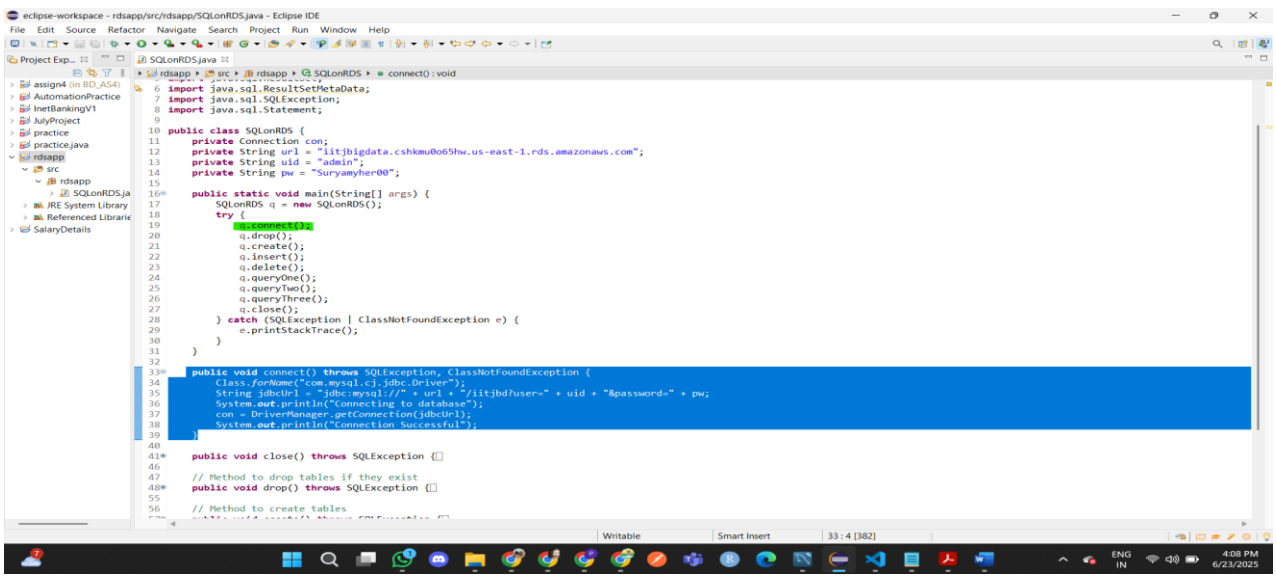
```
catch (SQLException | ClassNotFoundException e) {
    e.printStackTrace();
}
```

- ClassNotFoundException occurs if the JDBC driver class is not found in the classpath.
- SQLException handles errors related to database access, such as invalid credentials or unavailable RDS instance.

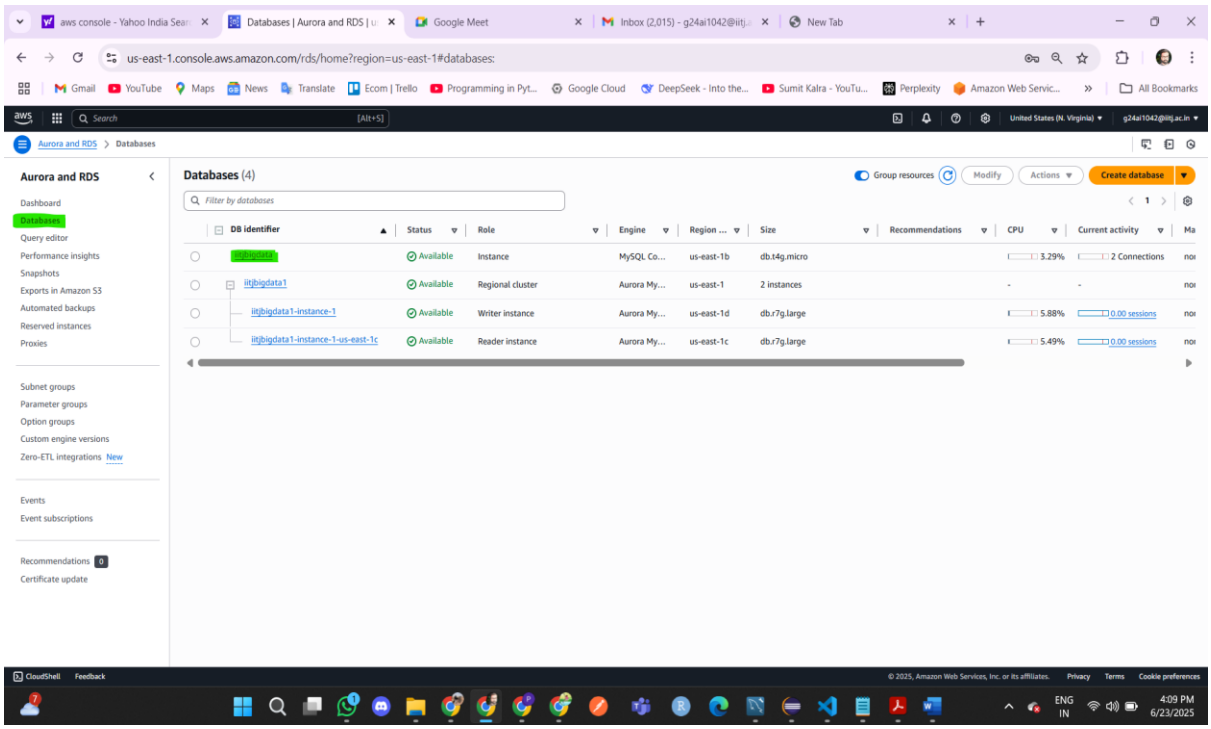


Script to connect

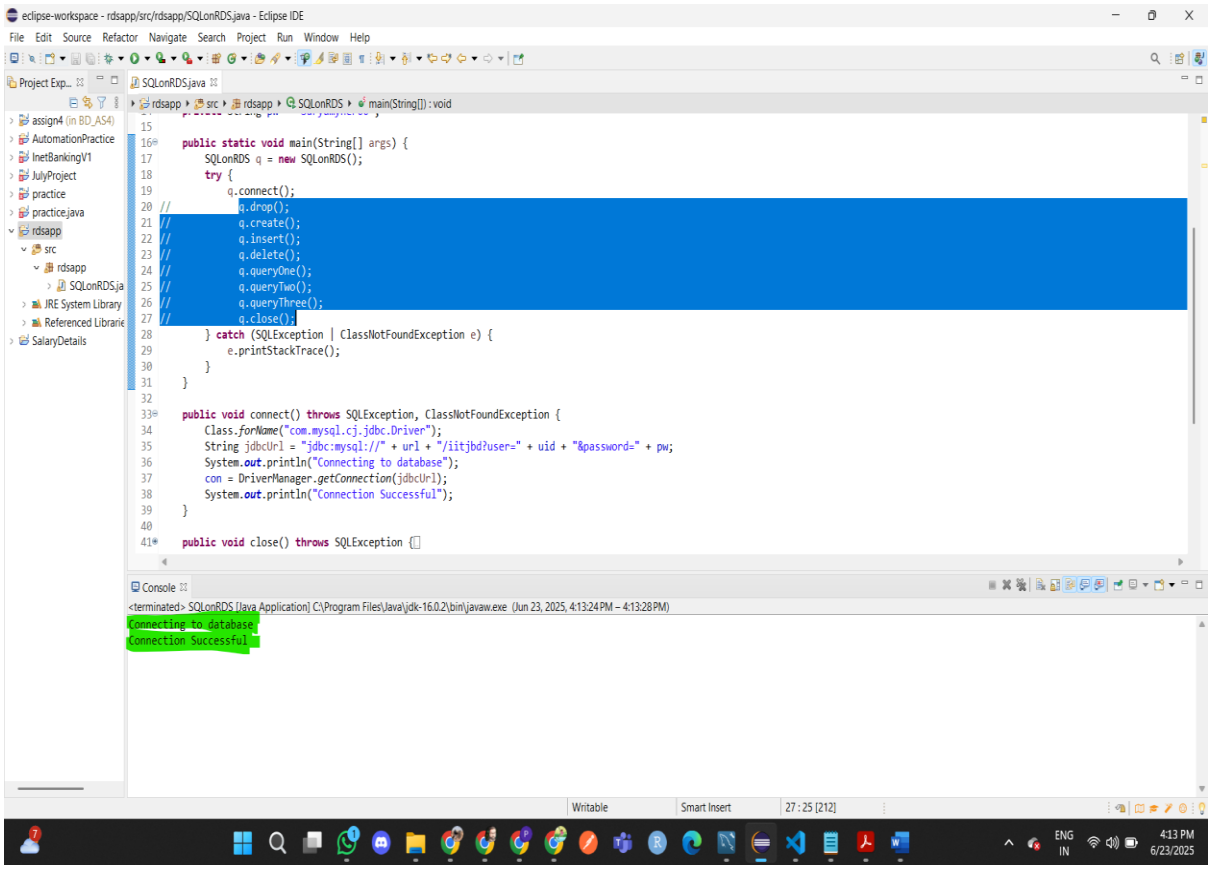
```
public void connect() throws SQLException, ClassNotFoundException {
    Class.forName("com.mysql.cj.jdbc.Driver");
    String jdbcUrl = "jdbc:mysql://" + url + "/iitjbd?user=" + uid + "&password=" + pw;
    System.out.println("Connecting to database");
    con = DriverManager.getConnection(jdbcUrl);
    System.out.println("Connection Successful");
}
```



On AWS - RDS



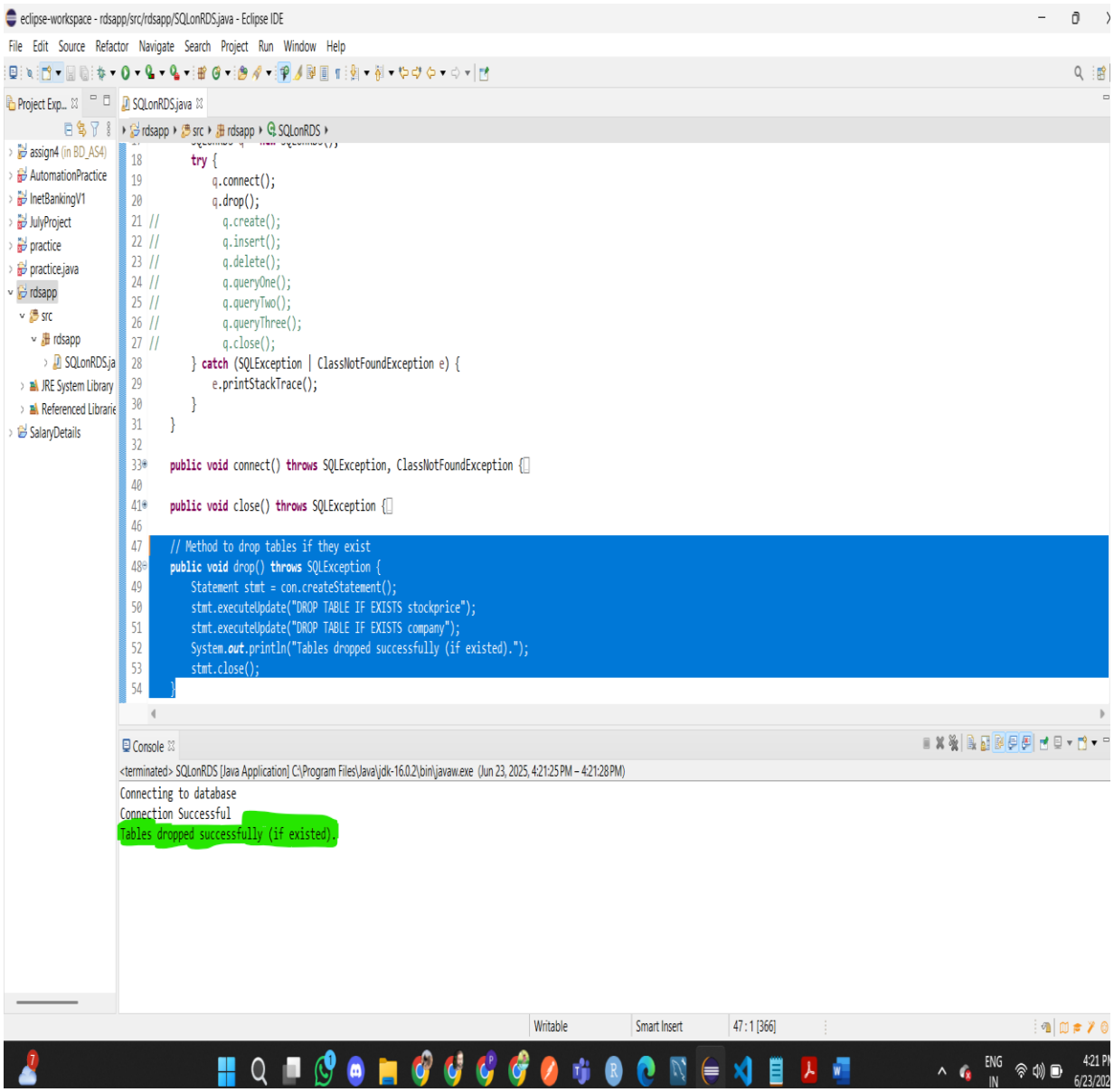
On Execution



Script for Drop

```
// Method to drop tables if they exist
public void drop() throws SQLException {
    Statement stmt = con.createStatement();
    stmt.executeUpdate("DROP TABLE IF EXISTS stockprice");
    stmt.executeUpdate("DROP TABLE IF EXISTS company");
    System.out.println("Tables dropped successfully (if existed).");
    stmt.close();
}
```

On Script



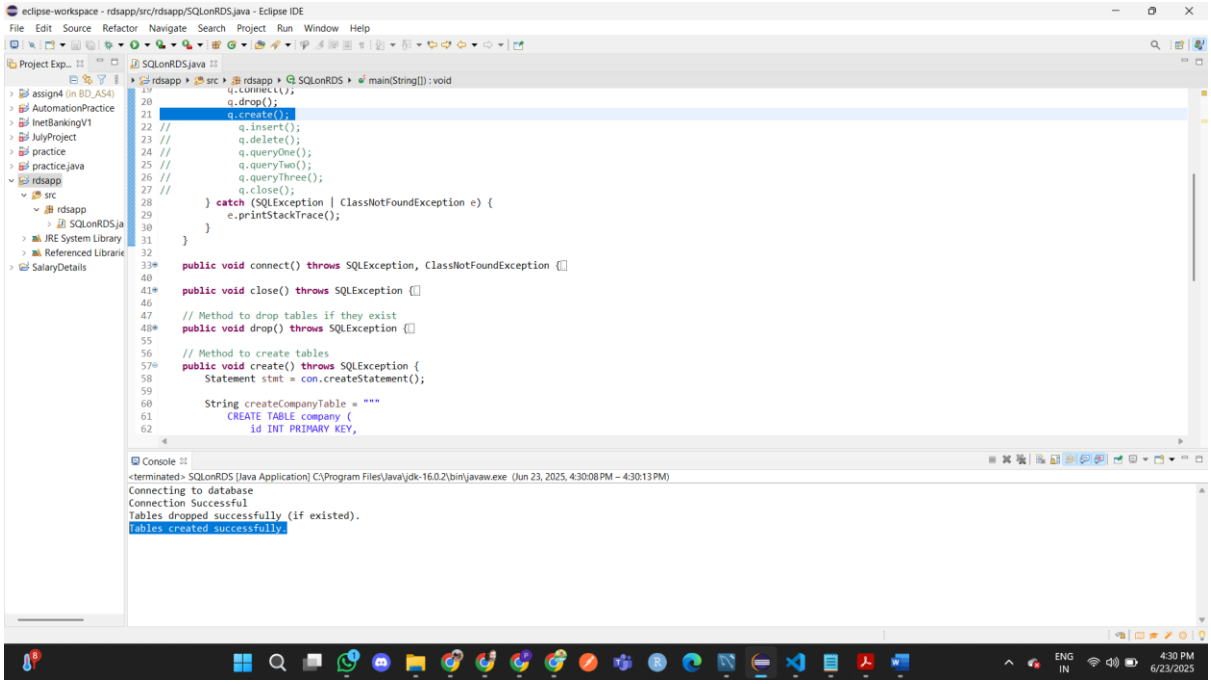
Script for create: create () creates the following table in the database

create (): Creates two tables—company and stockprice—with specific schemas and constraints, including primary and foreign keys.

Script

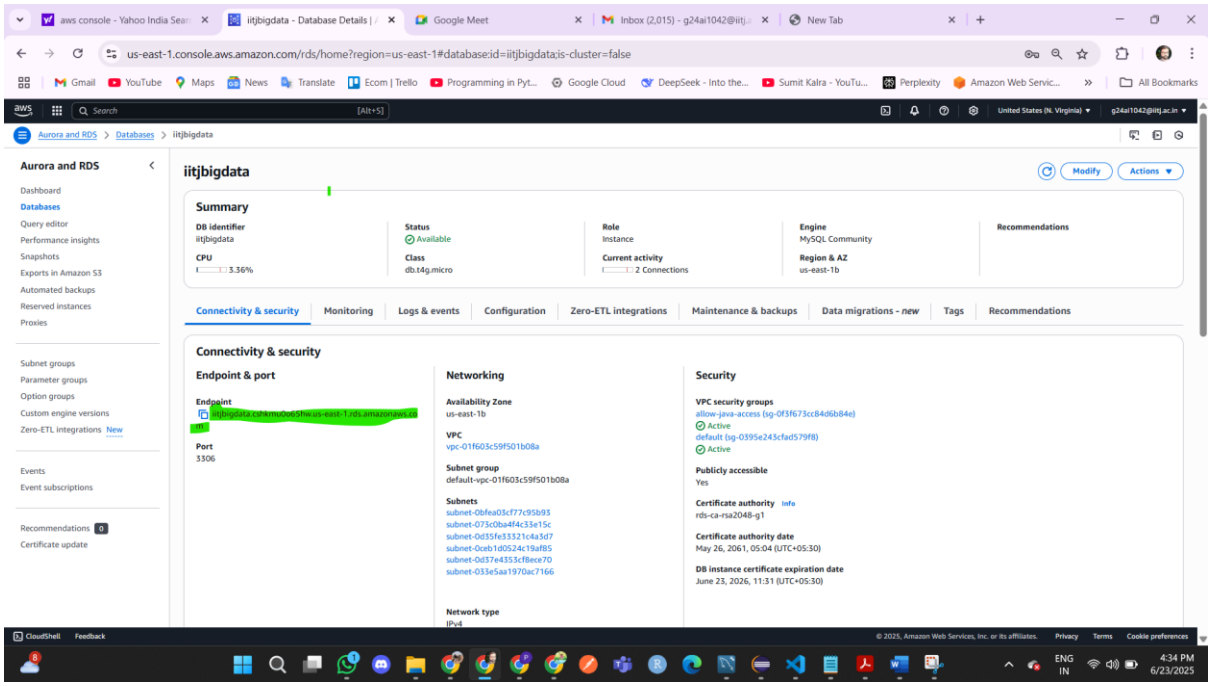
```
public void create() throws SQLException {
    Statement stmt = con.createStatement();
    String createCompanyTable = """
      CREATE TABLE company (
        id INT PRIMARY KEY,
        name VARCHAR(50),
        ticker CHAR(10),
        annualRevenue DECIMAL(15,2),
        numEmployees INT
      )
      """;
    String createStockPriceTable = """
      CREATE TABLE stockprice (
        companyId INT,
        priceDate DATE,
        openPrice DECIMAL(10,2),
        highPrice DECIMAL(10,2),
        lowPrice DECIMAL(10,2),
        closePrice DECIMAL(10,2),
        volume INT,
        PRIMARY KEY (companyId, priceDate),
        FOREIGN KEY (companyId) REFERENCES company(id)
      )
      """;
    stmt.executeUpdate(createCompanyTable);
    stmt.executeUpdate(createStockPriceTable);
    System.out.println("Tables created successfully.");
    stmt.close();
}
```

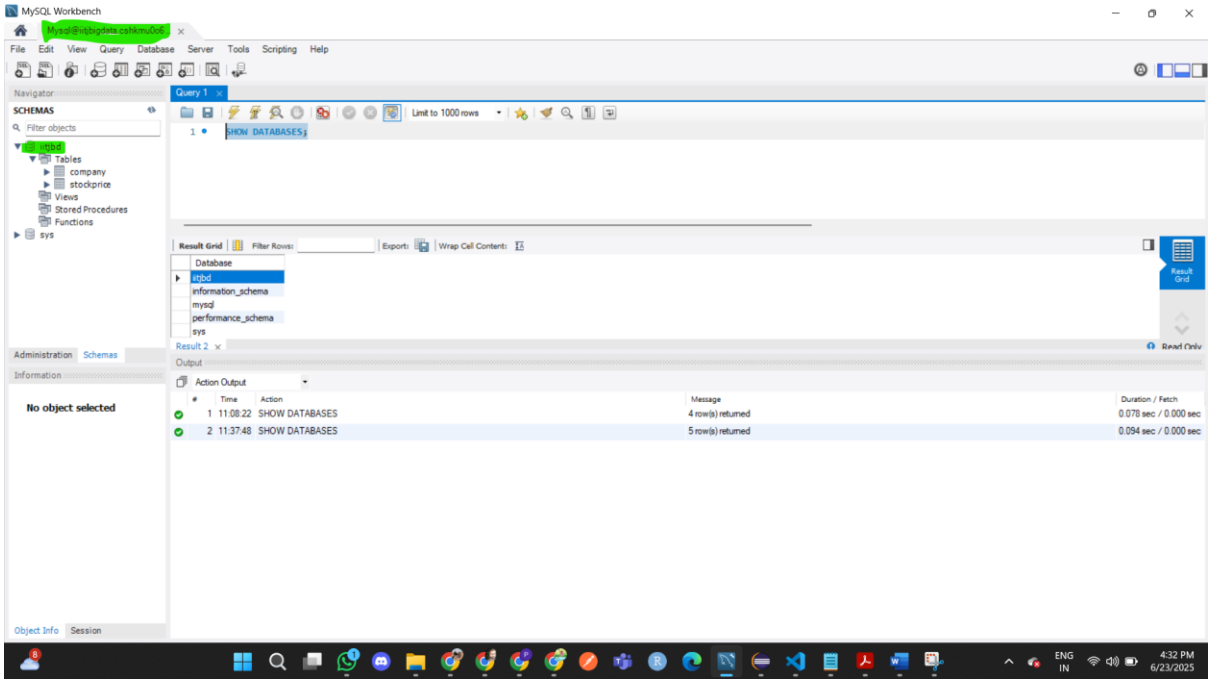
Execution



Verification on Work bench

The MySql Work bench and the RDS has been connected using Instance Id and UID and PW





Insert the Data into the Table

insert(): Inserts predefined sample data into both tables.

Script

```
 public void insert() throws SQLException {
Statement stmt = con.createStatement();
// Insert into company
stmt.executeUpdate("INSERT INTO company VALUES " +
        "(1, 'Apple', 'AAPL', 387540000000.00 , 154000)," +
        "(2, 'GameStop', 'GME', 611000000.00, 12000)," +
        "(3, 'Handy Repair', NULL, 2000000, 50)," +
        "(4, 'Microsoft', 'MSFT', 198270000000.00 , 221000)," +
        "(5, 'StartUp', NULL, 50000, 3)");
// Insert into stockprice
stmt.executeUpdate("""
   INSERT INTO stockprice (companyId, priceDate, openPrice, highPrice, lowPrice, closePrice, volume) VALUES
   (1, '2022-08-15', 171.52, 173.39, 171.35, 173.19, 54091700),
   (1, '2022-08-16', 172.78, 173.71, 171.66, 173.03, 56377100),
   (1, '2022-08-17', 172.77, 176.15, 172.57, 174.55, 79542000),
```

(1, '2022-08-18', 173.75, 174.90, 173.12, 174.15, 62290100),

(1, '2022-08-19', 173.03, 173.74, 171.31, 171.52, 70211500),

(1, '2022-08-22', 169.69, 169.86, 167.14, 167.57, 69026800),

(1, '2022-08-23', 167.08, 168.71, 166.65, 167.23, 54147100),

(1, '2022-08-24', 167.32, 168.11, 166.25, 167.53, 53841500),

(1, '2022-08-25', 168.78, 170.14, 168.35, 170.03, 51218200),

(1, '2022-08-26', 170.57, 171.05, 163.56, 163.62, 78823500),

(1, '2022-08-29', 161.15, 162.90, 159.82, 161.38, 73314000),

(1, '2022-08-30', 162.13, 162.56, 157.72, 158.91, 77906200),

(2, '2022-08-15', 39.75, 40.39, 38.81, 39.68, 5243100),

(2, '2022-08-16', 39.17, 45.53, 38.60, 42.19, 23602800),

(2, '2022-08-17', 42.18, 44.36, 40.41, 40.52, 9766400),

(2, '2022-08-18', 39.27, 40.07, 37.34, 37.93, 8145400),

(2, '2022-08-19', 35.18, 37.19, 34.67, 36.49, 9525600),

(2, '2022-08-22', 34.31, 36.20, 34.20, 34.50, 5798600),

(2, '2022-08-23', 34.70, 34.99, 33.45, 33.53, 4836300),

(2, '2022-08-24', 34.00, 34.94, 32.44, 32.50, 5620300),

(2, '2022-08-25', 32.84, 32.89, 31.50, 31.96, 4726300),

(2, '2022-08-26', 31.50, 32.38, 30.63, 30.94, 4289500),

(2, '2022-08-29', 30.48, 32.75, 30.38, 31.55, 4292700),

(2, '2022-08-30', 31.62, 31.87, 29.42, 29.84, 5060200),

(4, '2022-08-15', 291.00, 294.18, 290.11, 293.47, 18085700),

(4, '2022-08-16', 291.99, 294.04, 290.42, 292.71, 18102900),

(4, '2022-08-17', 289.74, 293.35, 289.47, 291.32, 18253400),

(4, '2022-08-18', 290.19, 291.91, 289.08, 290.17, 17186200),

(4, '2022-08-19', 288.90, 289.25, 285.56, 286.15, 20557200),

(4, '2022-08-22', 282.08, 282.46, 277.22, 277.75, 25061100),

(4, '2022-08-23', 276.44, 278.86, 275.40, 276.44, 17527400),

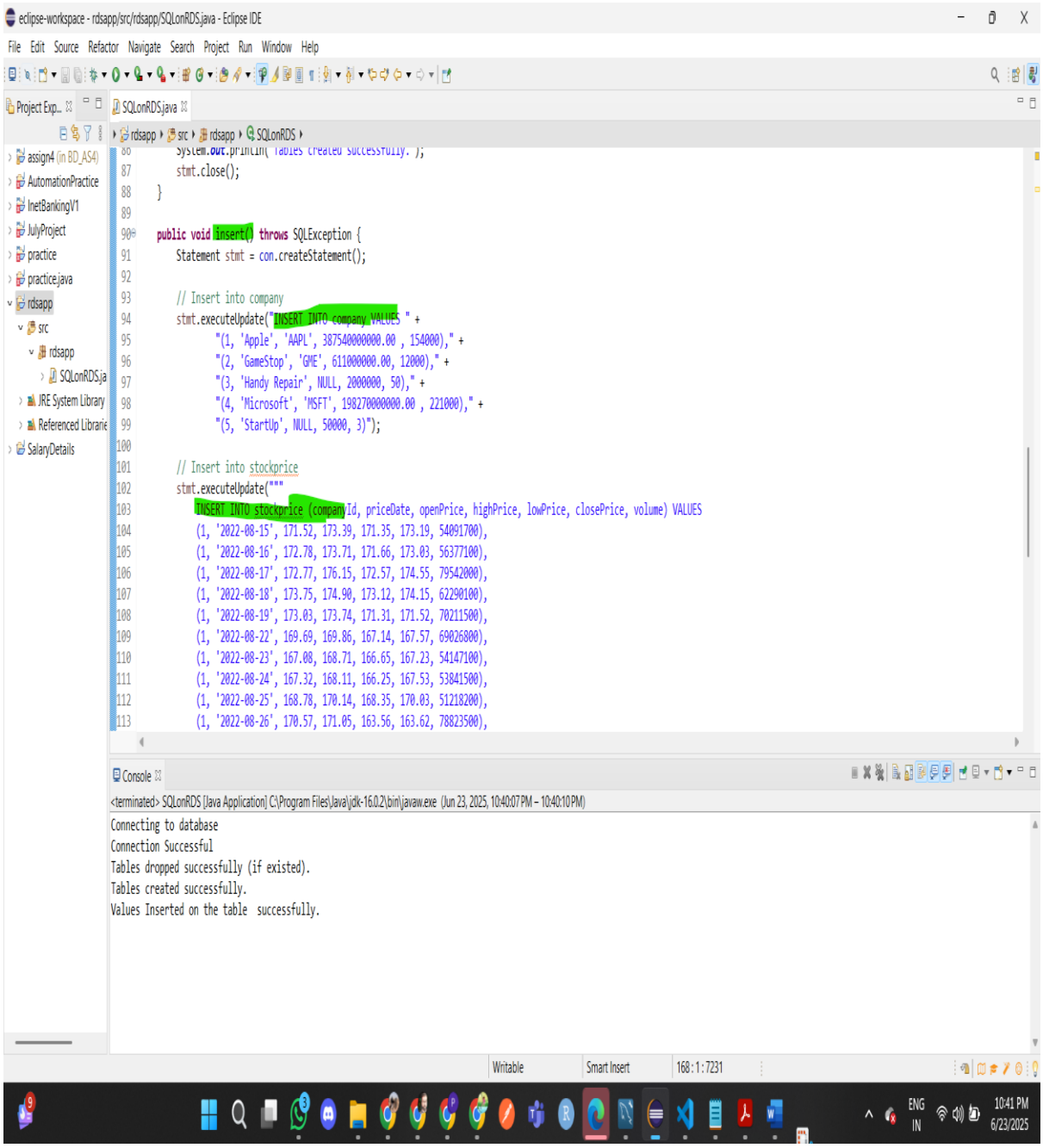
(4, '2022-08-24', 275.41, 277.23, 275.11, 275.79, 18137000),

(4, '2022-08-25', 277.33, 279.02, 274.52, 278.85, 16583400),

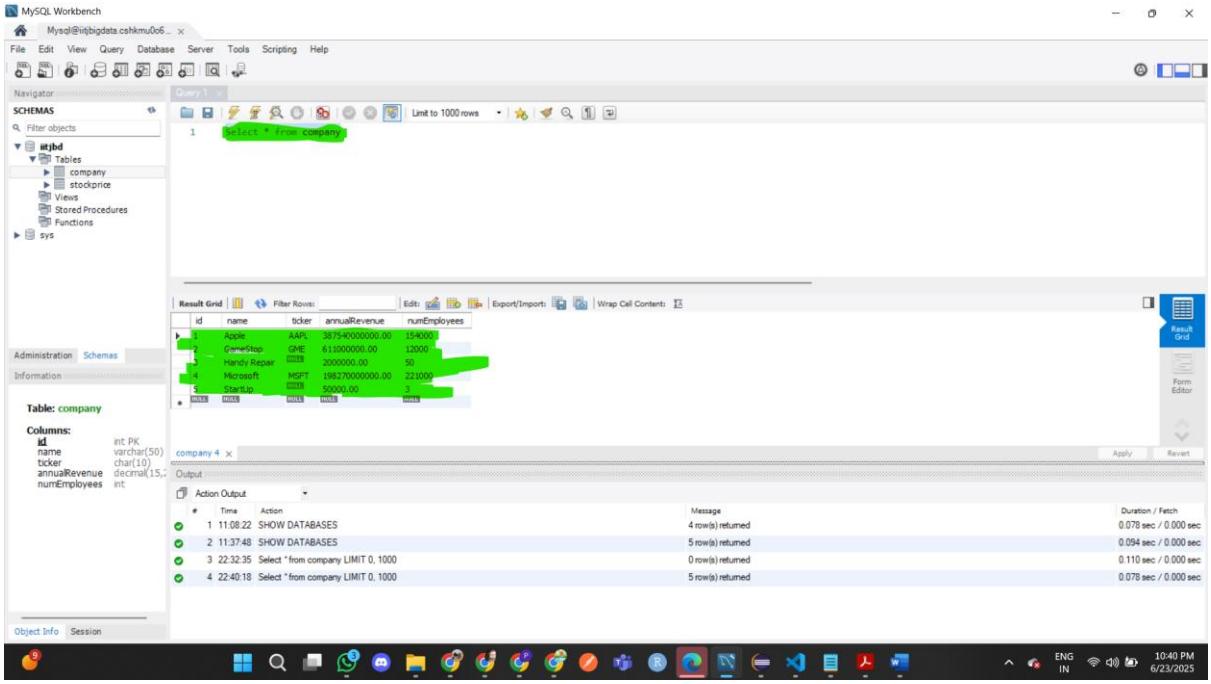
(4, '2022-08-26', 279.08, 280.34, 267.98, 268.09, 27532500),

```
        (4, '2022-08-29', 265.85, 267.40, 263.85, 265.23, 20338500),
        (4, '2022-08-30', 266.67, 267.05, 260.66, 262.97, 22767100)
        """);
    System.out.println("Values Inserted on the table  successfully.");
    stmt.close();
  }
```

On Script



Verification on the MySQL Work Bench



Delete Handling

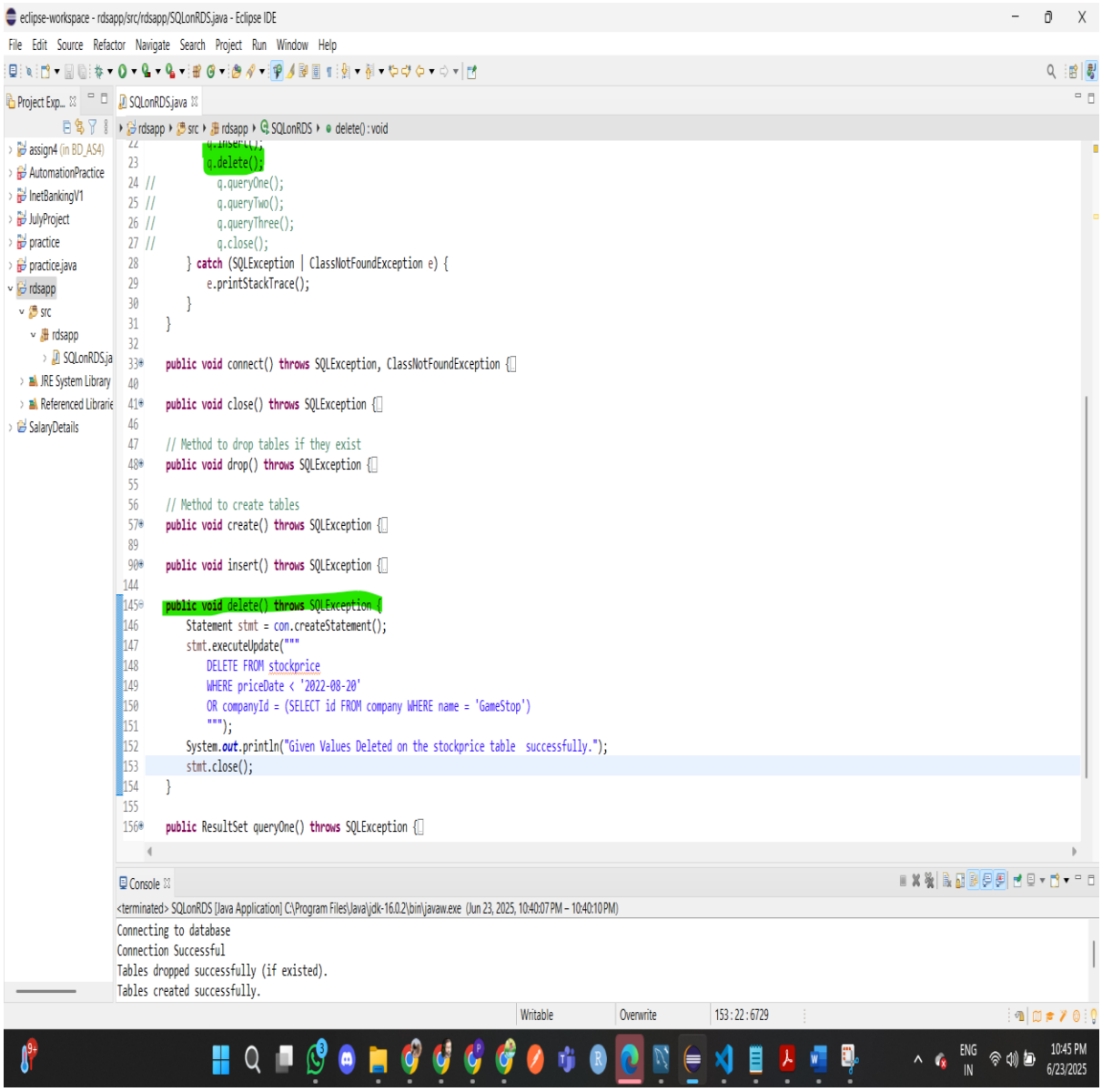
delete(): Deletes records based on specific conditions, such as date thresholds and company identity.

Script

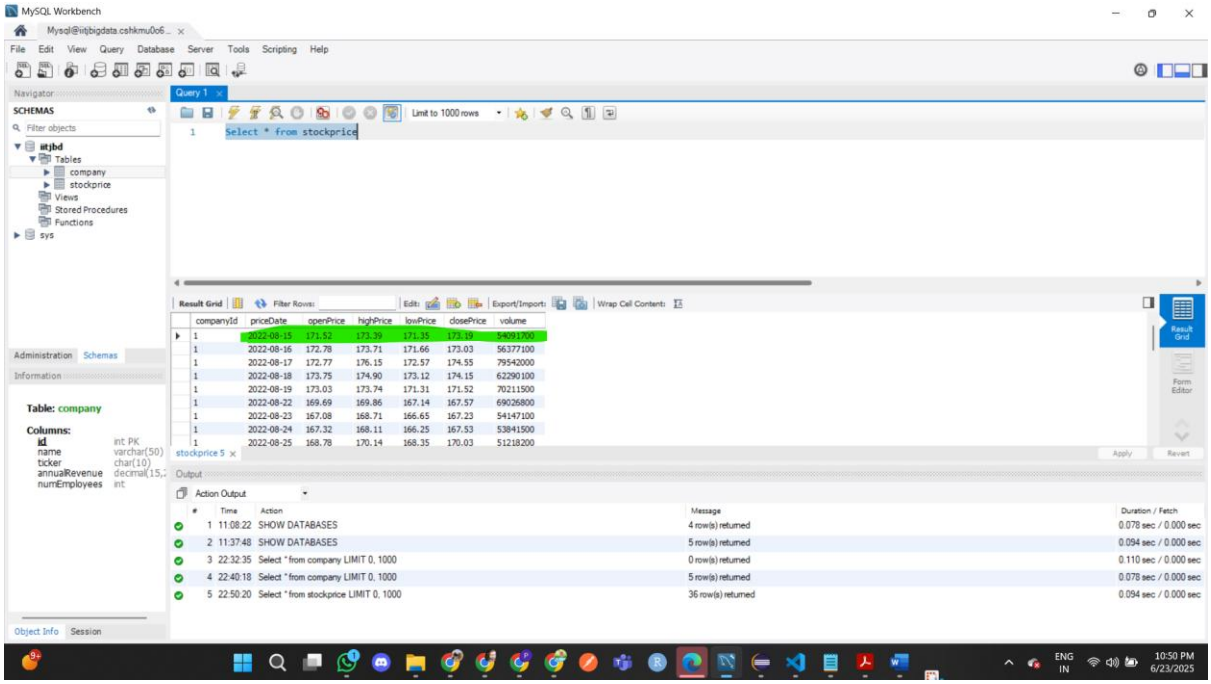
```
public static void main(String[] args) {
    SQLonRDS q = new SQLonRDS();
    try {
      q.connect();
       q.drop();
       q.create();
       q.insert();
       q.delete();
    } catch (SQLException | ClassNotFoundException e) {
       e.printStackTrace();
    }
}
```

```
public void delete() throws SQLException {
    Statement stmt = con.createStatement();
    stmt.executeUpdate("""
       DELETE FROM stockprice
       WHERE priceDate < '2022-08-20'
       OR companyId = (SELECT id FROM company WHERE name = 'GameStop')
       """);
    System.out.println("Given Values Deleted on the stockprice table  successfully.");
    stmt.close();
  }
```

On Script



Verification on MySQL Work Bench after the delete the data based on the condition



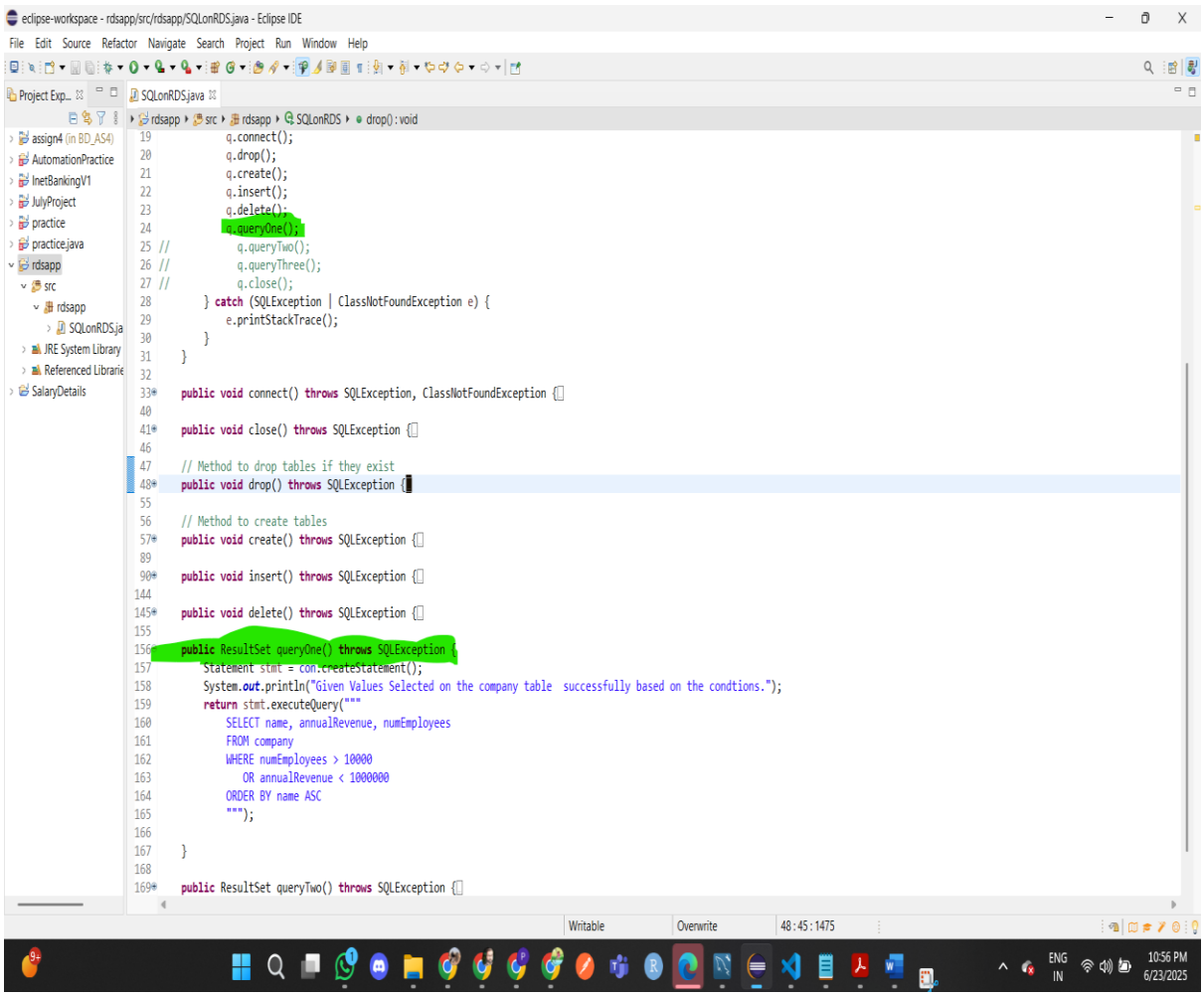
QueryOne: QueryOne() Query returns company info (name, revenue, employees) that have more than 10000 employees or annual revenue less that 1 million dollars. Order by company name ascending.

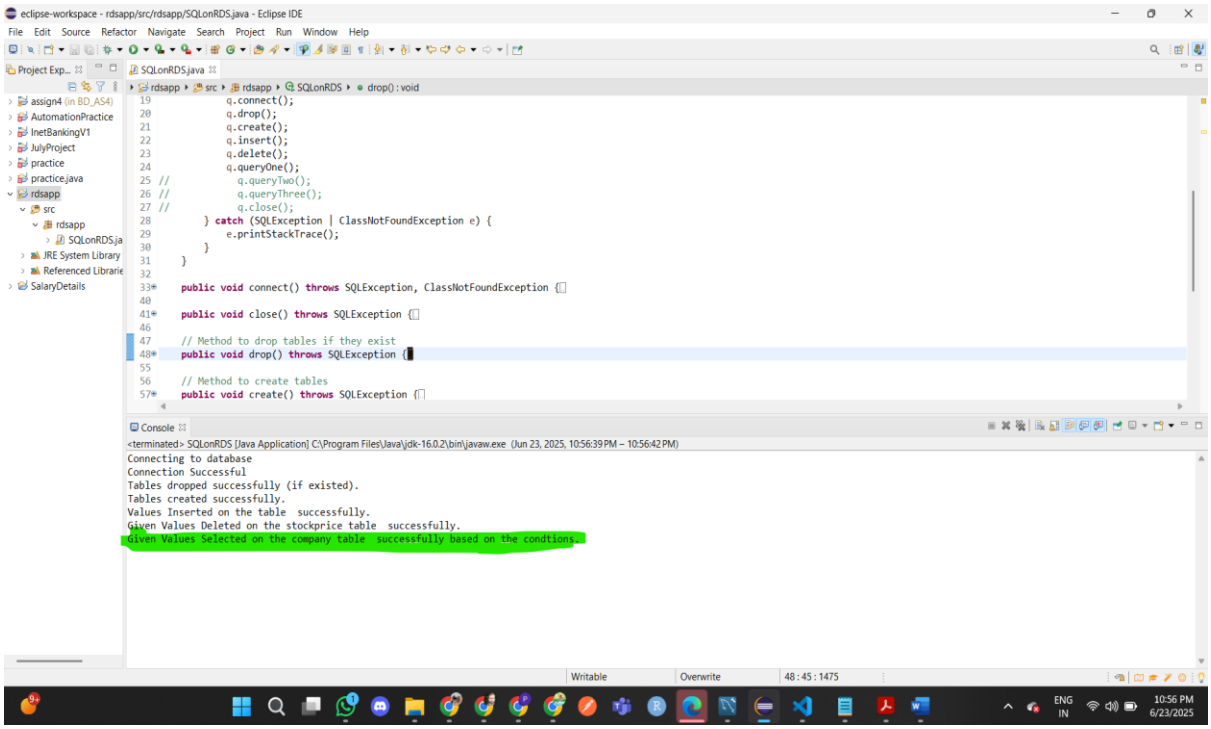
Script

```
public static void main(String[] args) {
    SQLonRDS q = new SQLonRDS();
    try {
        q.connect();
        q.drop();
        q.create();
        q.insert();
        q.delete();
        q.queryOne();
//          q.queryTwo();
//          q.queryThree();
//          q.close();
    } catch (SQLException | ClassNotFoundException e) {
        e.printStackTrace();
```

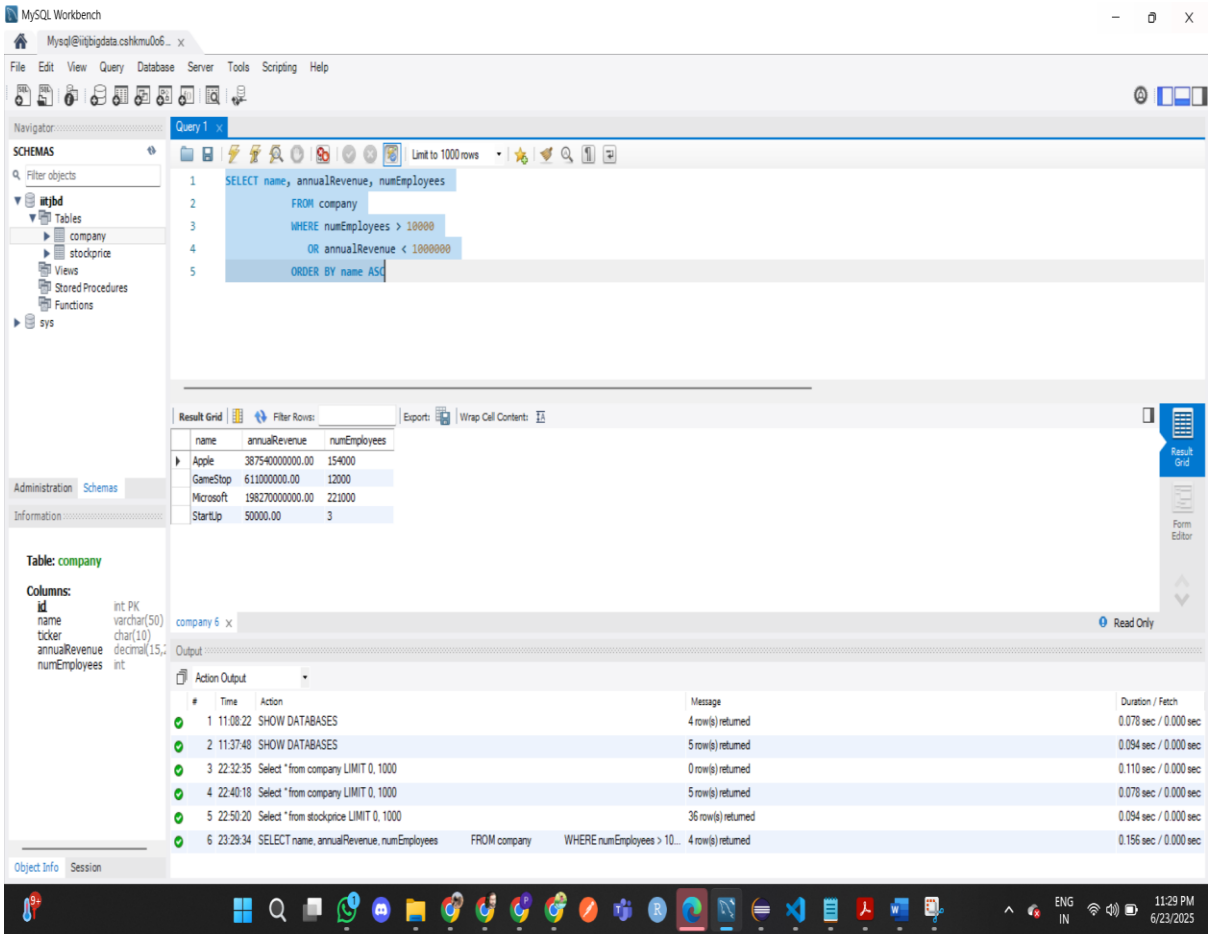
```
    }
  }
public ResultSet queryOne() throws SQLException {
    Statement stmt = con.createStatement();
    System.out.println("Given Values Selected on the company table  successfully based on the condtions.");
    return stmt.executeQuery("""
      SELECT name, annualRevenue, numEmployees
      FROM company
      WHERE numEmployees > 10000
        OR annualRevenue < 1000000
      ORDER BY name ASC
      """);
  }
```

On Script





Verify in MySQL Work Bench



QueryTwo()for returning ResultSet Query returns the company name and ticker and calculates the lowest price, highest price,average closing price, and average volume in the week of August 22nd to 26th inclusive. Order by average volume descending

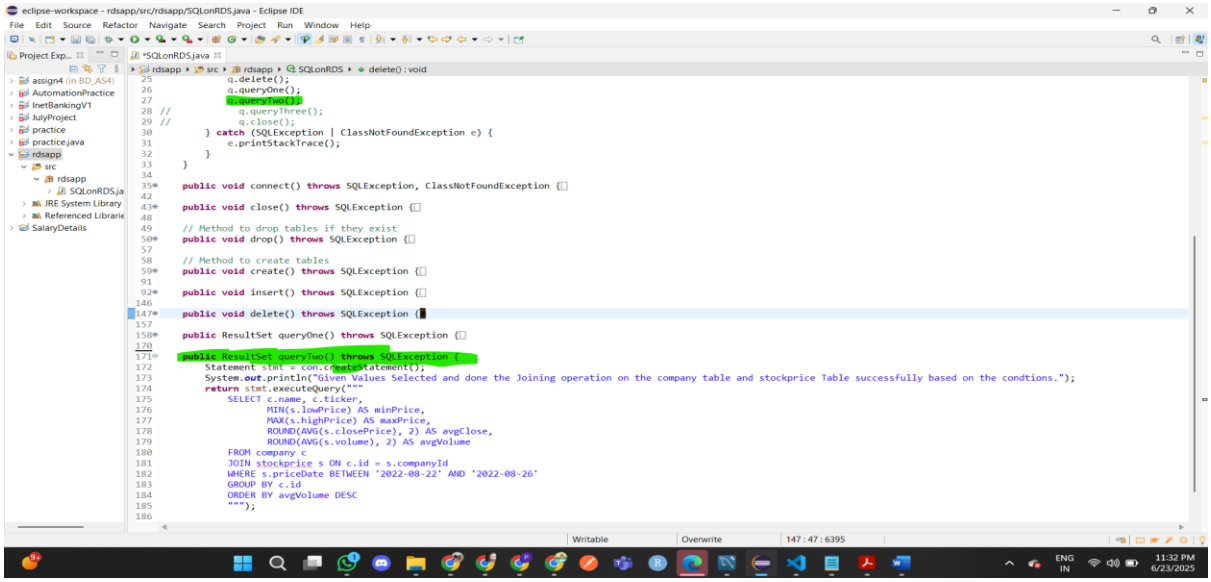
Script

```
    public static void main(String[] args) {
    SQLonRDS q = new SQLonRDS();
    try {
      q.connect();
      q.drop();
      q.create();
      q.insert();
      q.delete();
      q.queryOne();
      q.queryTwo();
//        q.queryThree();
//        q.close();
    } catch (SQLException | ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
public ResultSet queryTwo() throws SQLException {
    Statement stmt = con.createStatement();
    System.out.println("Given Values Selected and done the Joining operation on the company table and stockprice Table successfully based on the condtions.");
    return stmt.executeQuery("""
      SELECT c.name, c.ticker,
          MIN(s.lowPrice) AS minPrice,
          MAX(s.highPrice) AS maxPrice,
          ROUND(AVG(s.closePrice), 2) AS avgClose,
          ROUND(AVG(s.volume), 2) AS avgVolume
      FROM company c
      JOIN stockprice s ON c.id = s.companyId
```

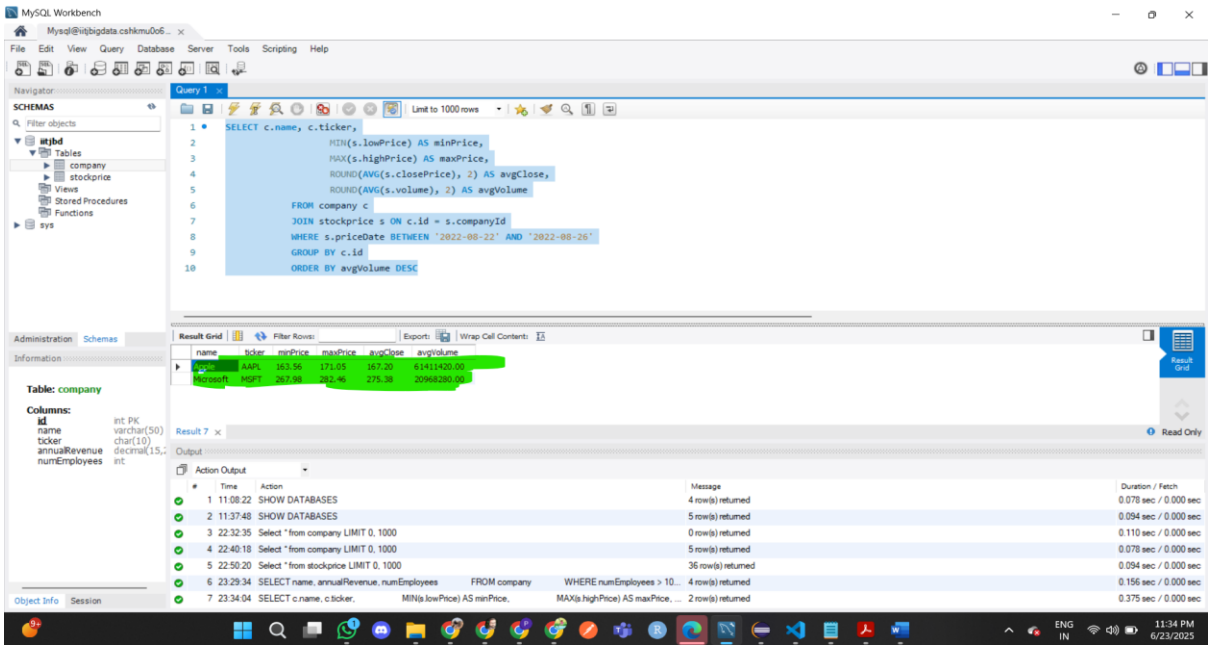
```
        WHERE s.priceDate BETWEEN '2022-08-22' AND '2022-08-26'
        GROUP BY c.id
        ORDER BY avgVolume DESC
        """);

    }
```

On Script



Verification on the MySQL Work Bench



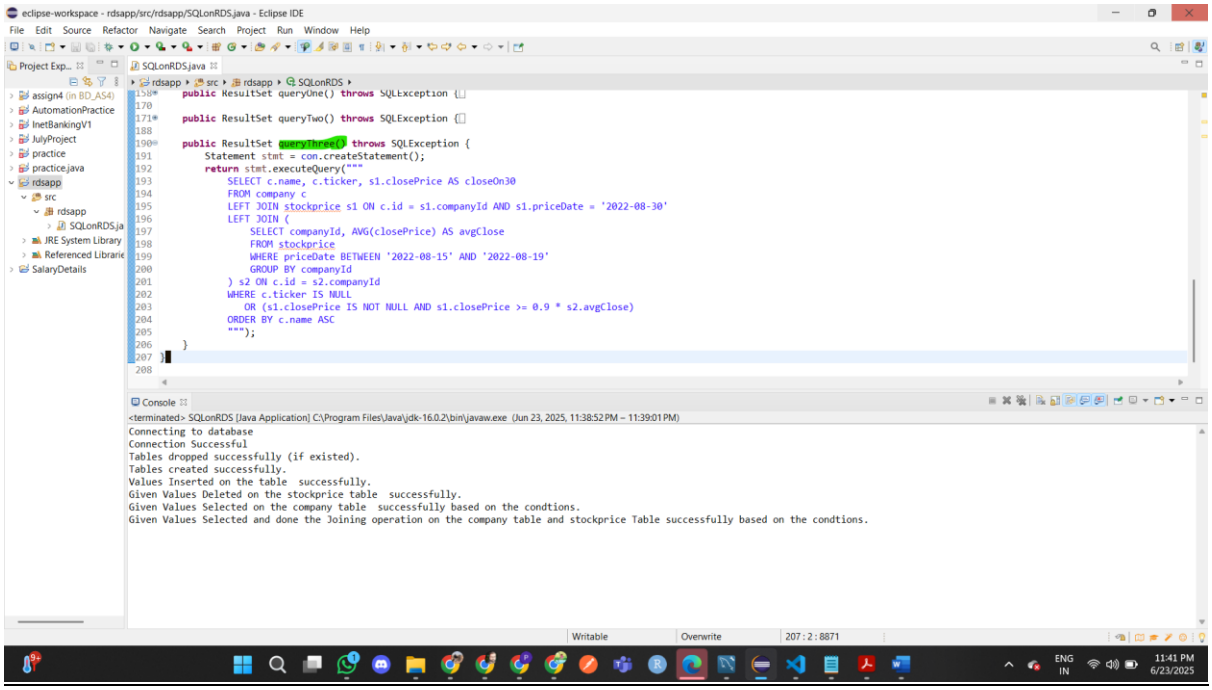
QueryThree(): Query returns a list of all companies that displays their name, ticker, and closing stock price on August 30, 2022 (if exists). Only show companies where their closing stock price on August 30, 2022 is no more than 10% below the closing average for the week of August 15th to 19th inclusive. That is, if closing price is currently 100, the average closing price must be <= 110. Companies without a stock ticker should always be shown in the list. Order by company name ascending.

Script

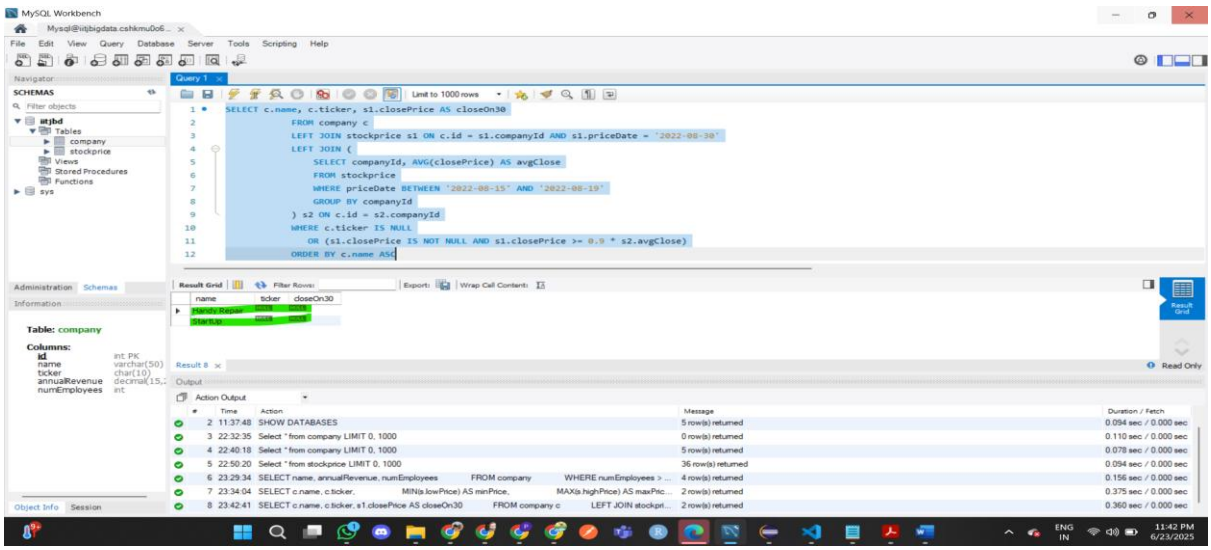
```
    public static void main(String[] args) {
    SQLonRDS q = new SQLonRDS();
    try {
      q.connect();
      q.drop();
      q.create();
      q.insert();
      q.delete();
      q.queryOne();
      q.queryTwo();
      q.queryThree();
//        q.close();
    } catch (SQLException | ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
public ResultSet queryThree() throws SQLException {
    Statement stmt = con.createStatement();
    return stmt.executeQuery("""
      SELECT c.name, c.ticker, s1.closePrice AS closeOn30
      FROM company c
      LEFT JOIN stockprice s1 ON c.id = s1.companyId AND s1.priceDate = '2022-08-30'
      LEFT JOIN (
        SELECT companyId, AVG(closePrice) AS avgClose
        FROM stockprice
```

```
            WHERE priceDate BETWEEN '2022-08-15' AND '2022-08-19'
            GROUP BY companyId
        ) s2 ON c.id = s2.companyId
        WHERE c.ticker IS NULL
           OR (s1.closePrice IS NOT NULL AND s1.closePrice >= 0.9 * s2.avgClose)
        ORDER BY c.name ASC
        """);
    }
```

On Script



Verification on the MySQL Work Bench



Conclusion

Through this assignment, students gain practical experience in integrating Java applications with cloud-based relational databases using JDBC. By designing and implementing structured queries and managing relational data, the exercise helps solidify concepts of SQL operations, schema design, and data analysis within a cloud environment. It also reinforces best practices in data manipulation and extraction essential for real-world analytics and financial data systems.

Git Link: https://github.com/PurushothamanShanmugam/Big-Data