BIG DATA MANAGEMENT

POST GRADUATE DIPLOMA
IN DATA ENGINEERING

ASSIGNMENT - 6

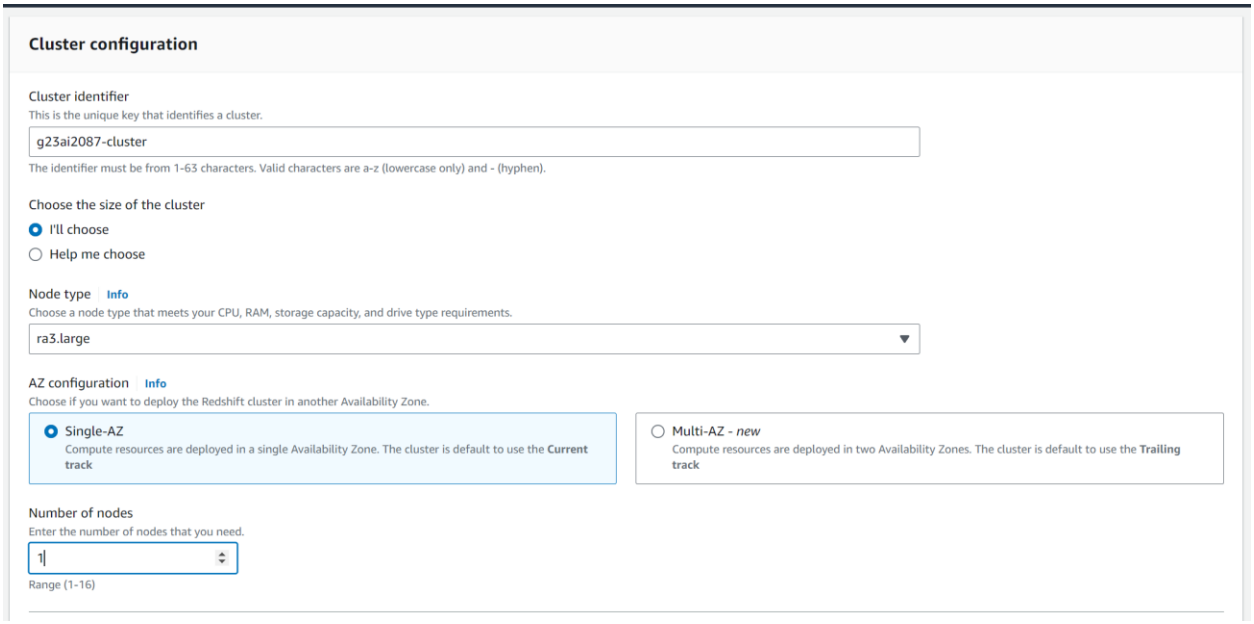
SUBMITTED BY:

NIRAJ BHAGCHANDANI [G23AI2087]

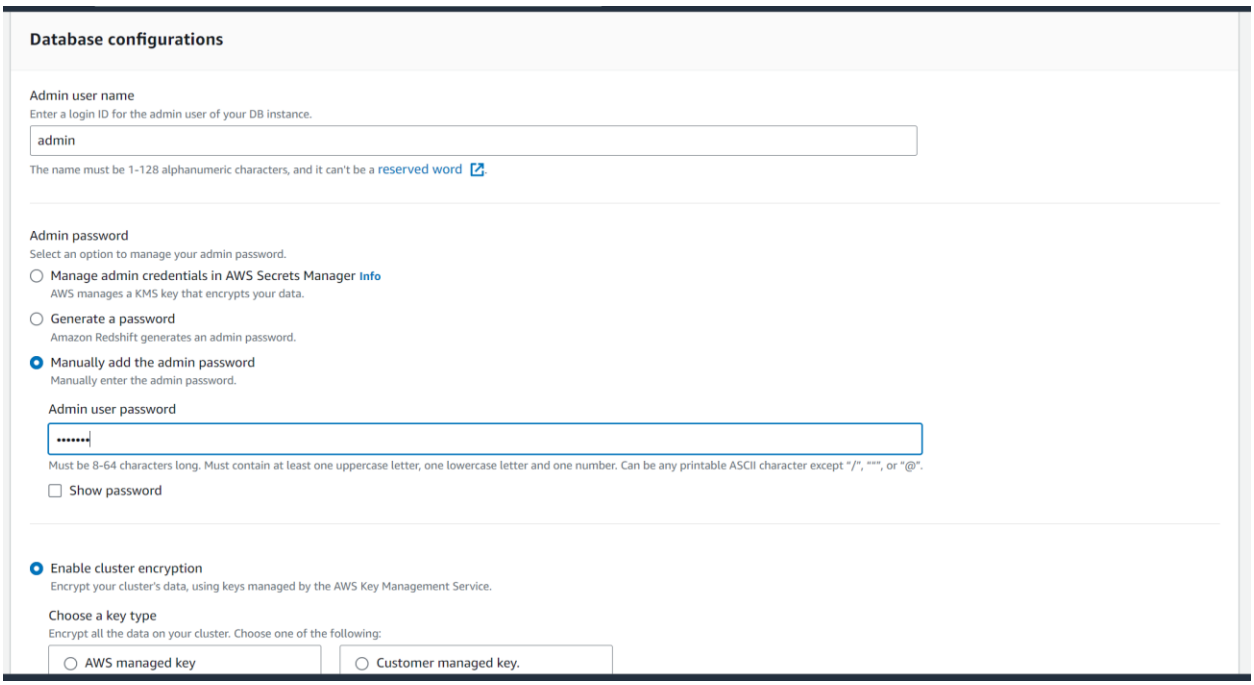
SUBMISSION DATE: 15th December, 2024

DEPARTMENT OF AIDE
INDIAN INSTITUTE OF TECHNOLOGY, JODHPUR

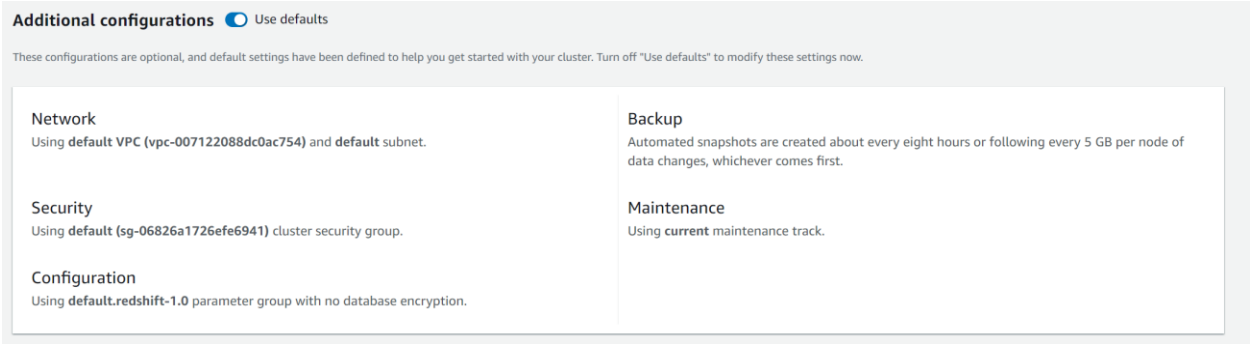
Step – 1: Set the Cluster Identifier, choose Node Type (ra3.large), select Single-AZ, and specify the Number of Nodes (1).



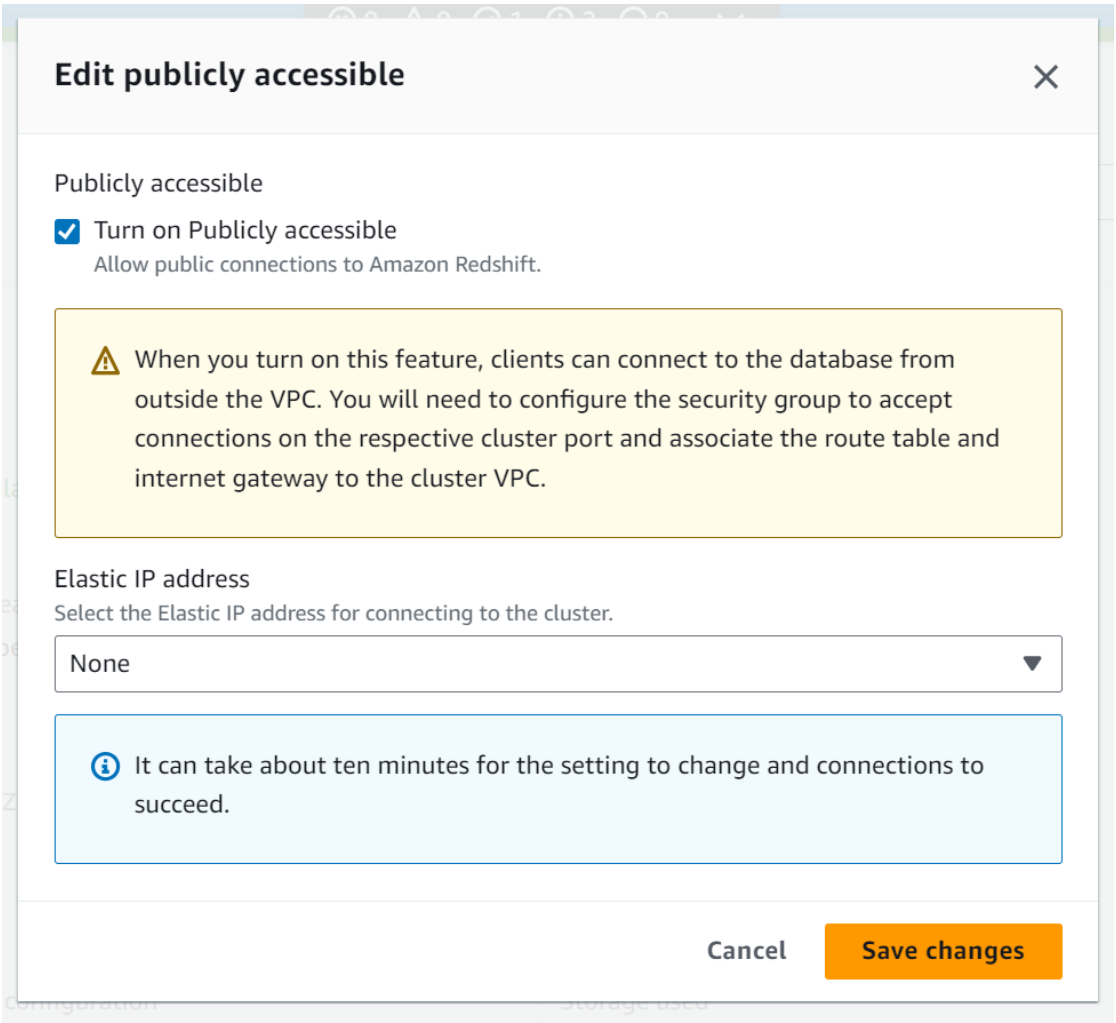
Step – 2: Set the Admin User Name (e.g., "admin"), manually add the password, and optionally enable cluster encryption with AWS or Customer managed keys.



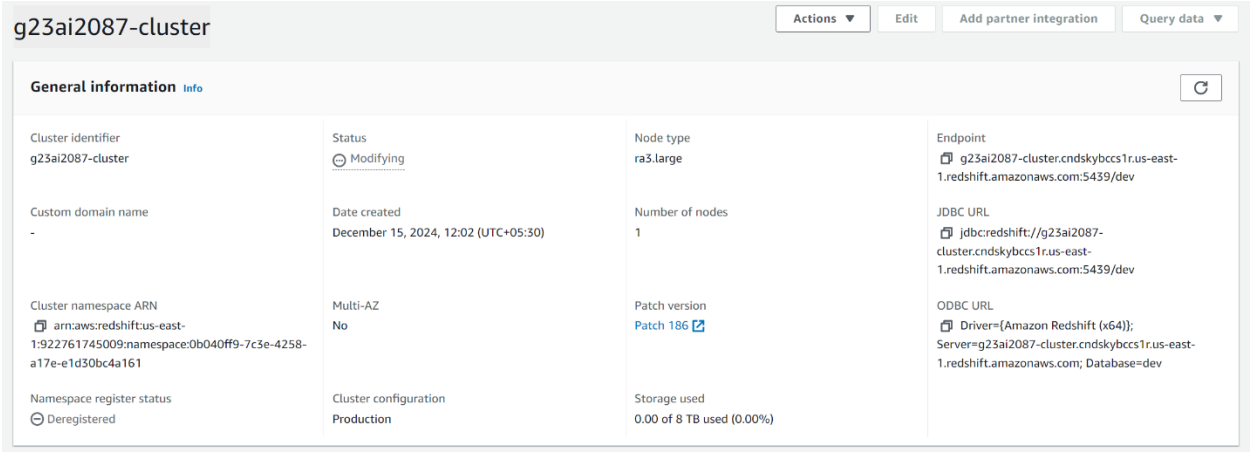
Step -3: Use default settings for Network, Security, Configuration, Backup, and Maintenance for the cluster.



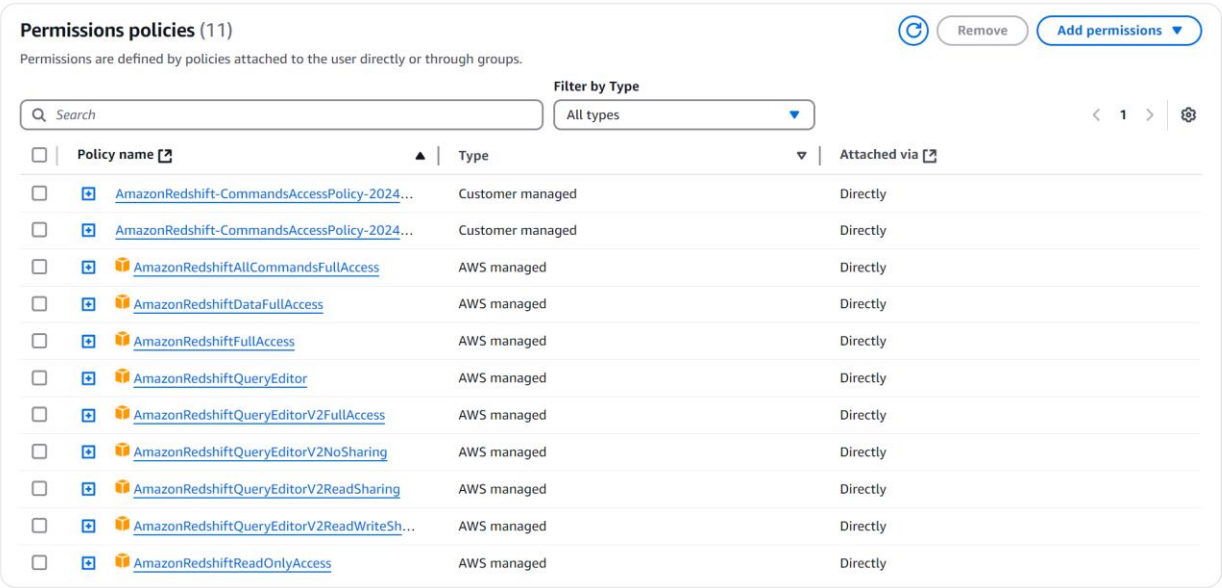
Step – 4: Enable Publicly accessible to allow public connections, configure the Elastic IP address if needed, and note the changes may take about ten minutes to apply.



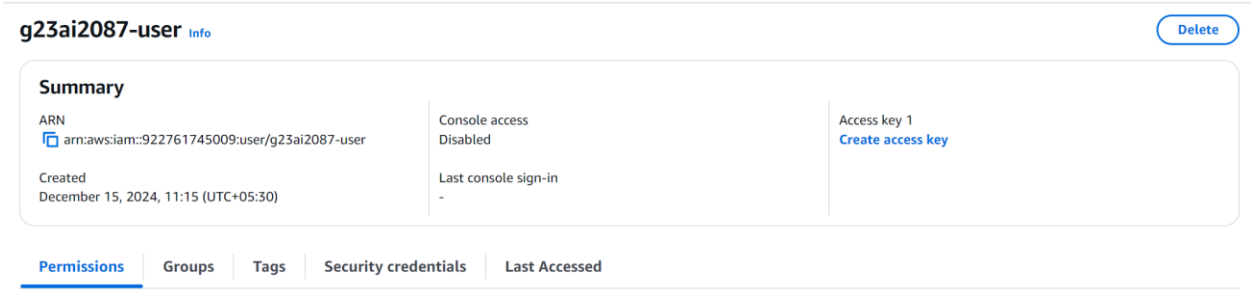
Step – 5: The g23ai2087-cluster is in a Modifying state with a Node type of ra3.large, 1 node, and is configured for Production. The cluster endpoint, JDBC URL, and ODBC URL are provided for connecting to the database.



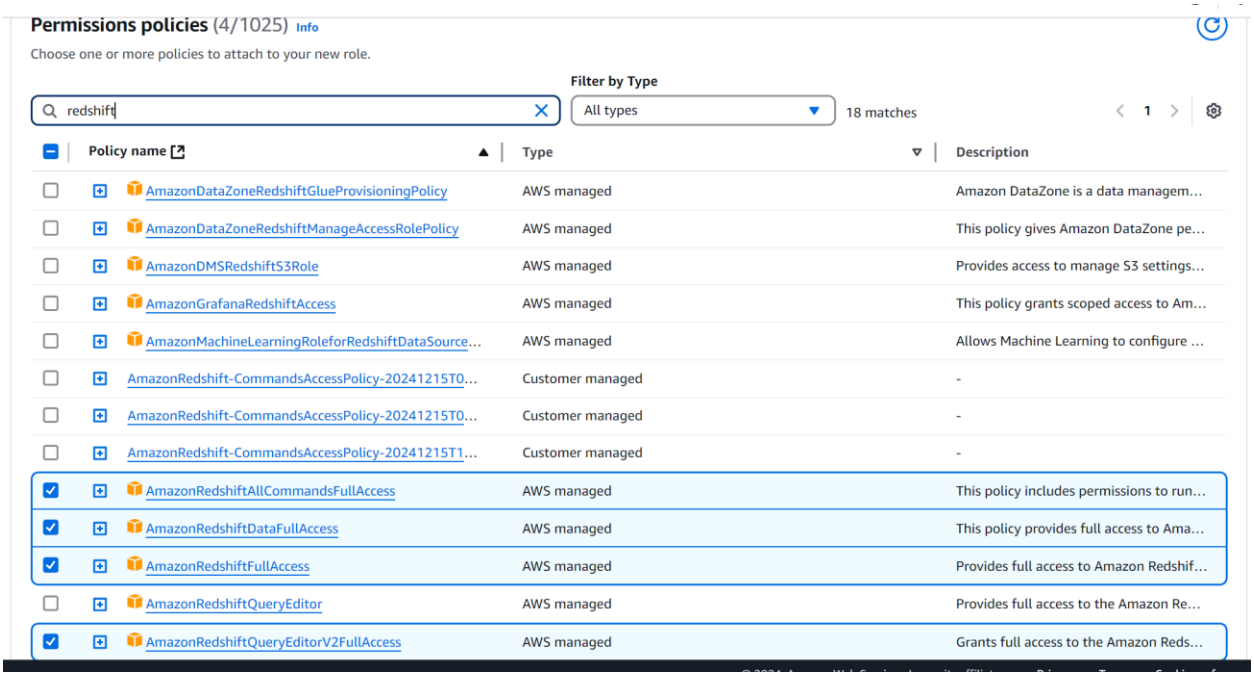
Step – 6: The Permissions policies list contains 11 policies attached directly to the user, including AWS managed policies like AmazonRedshiftFullAccess, AmazonRedshiftQueryEditorV2FullAccess, and AmazonRedshiftReadOnlyAccess, alongside Customer managed policies.



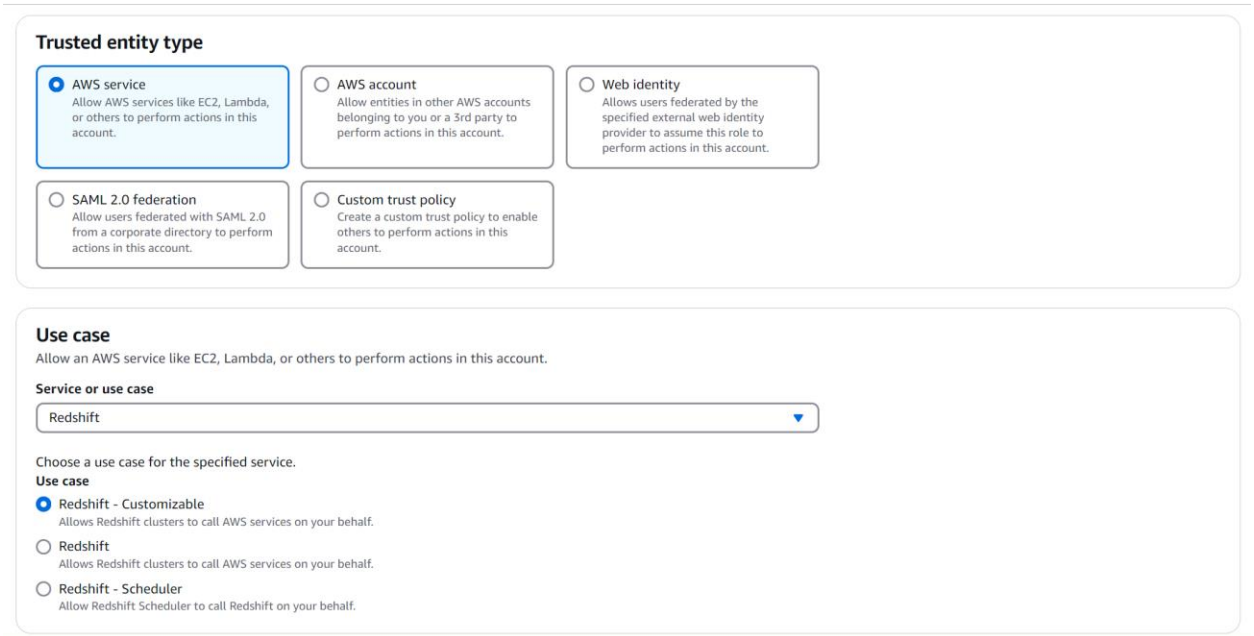
Step – 7: The g23ai2087-user has console access disabled and no recorded last console sign-in. An Access key can be created for programmatic access. The ARN and creation date (December 15, 2024) are also provided.



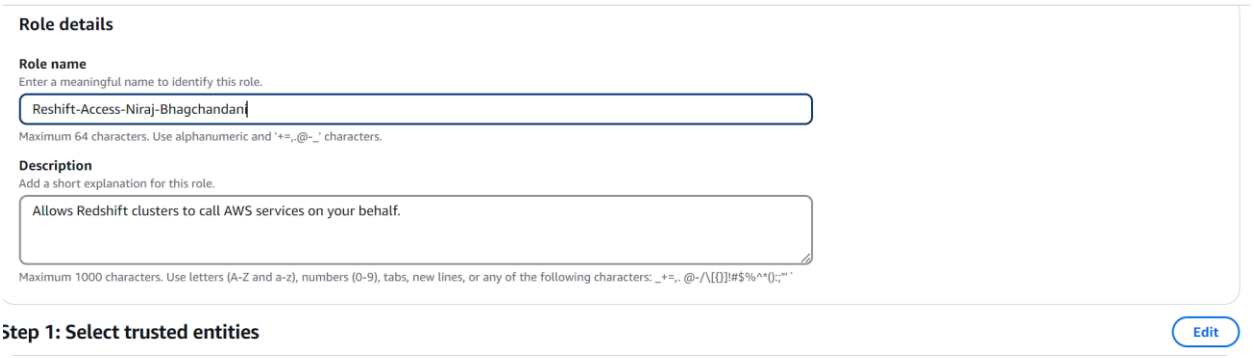
Step – 8: The Permissions policies list displays 18 matches for "Redshift", including AWS managed policies like AmazonRedshiftAllCommandsFullAccess, AmazonRedshiftDataFullAccess, and AmazonRedshiftFullAccess, with some already selected for the role.



Step – 9: The Trusted entity type is set to AWS service, with the Service selected as Redshift. The chosen Use case is Redshift - Customizable, allowing Redshift clusters to call AWS services on your behalf.



Step – 10: The Role name is set to Redshift-Access-Niraj-Bhagchandani, with a Description that states, *"Allows Redshift clusters to call AWS services on your behalf."*



Now do the following tasks with the help of the starter code provided below

1. Write the method connect() to make a connection to the database. [5]

Code:

```
public Connection connect() {
        try {
            Class.forName("com.amazon.redshift.jdbc42.Driver");
            Properties properties = new Properties();
```

```
            properties.setProperty("user", masterUsername);
            properties.setProperty("password", password);

            this.con = DriverManager.getConnection(redshiftUrl, properties);

            System.out.println("Connection to Redshift established successfully.");
        } catch (ClassNotFoundException e) {
            System.out.println("Error: Redshift JDBC driver not found.");
            e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Failed to connect to the database.");
            e.printStackTrace();
        }
        return con;
    }
```

Output:

Fig. 6.1 *Role configuration details for Redshift access, specifying the role name and description for AWS service permissions.*

2. Method close() to close the connection to the database. [5]

Code:

```
public void close() {
        System.out.println("Closing database connection.");
        try {
            if (con != null && !con.isClosed()) {
                con.close();
            }
        } catch (SQLException e) {
            System.out.println("Error closing the connection.");
        }
    }
```

Output:

Fig. 6.2 *Amazon Redshift Menu displaying database operations, including options to connect, execute queries, and close the connection.*

3. Method drop() to drop all the tables from the database. Note: The database schema name will be dev. [5]

Code:

```
public void drop() {
        System.out.println("Dropping all tables in the 'dev' schema...");
        String dropQuery = "SELECT tablename FROM pg_tables WHERE schemaname = 'dev'";

        try (Statement stmt = con.createStatement()) {
            ResultSet rs = stmt.executeQuery(dropQuery);
            while (rs.next()) {
                String tableName = rs.getString("tablename");
                String dropTableQuery = "DROP TABLE IF EXISTS dev." + tableName;
                stmt.executeUpdate(dropTableQuery);
                System.out.println("Dropped table: " + tableName);
            }
        } catch (SQLException e) {
            System.out.println("Error dropping tables: " + e.getMessage());
        }
    }
```

Output:

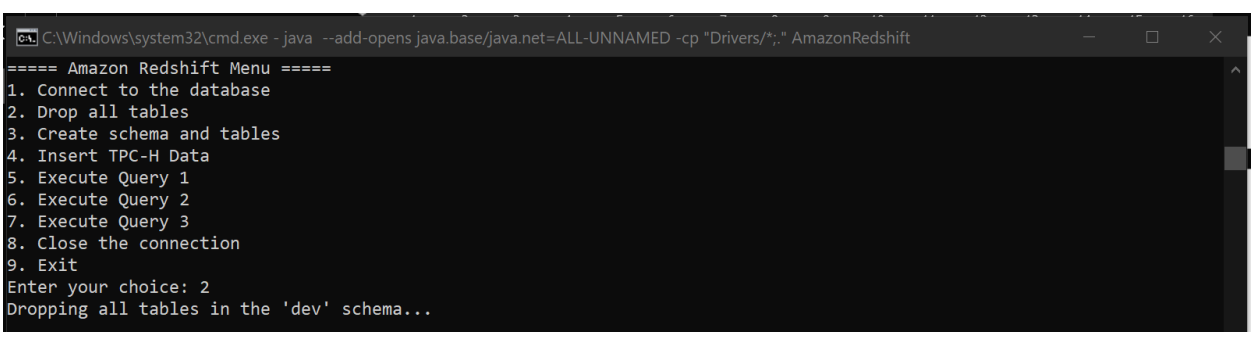


Fig. 6.3 *Amazon Redshift Menu performing the Drop all tables operation in the 'dev' schema after selecting option 2.*

4. Method create() to create the database dev and the tables. [5]

Code:

```
public void create() throws SQLException {
        System.out.println("Creating the 'dev' schema and tables...");
        String createSchemaQuery = "CREATE SCHEMA IF NOT EXISTS dev";
        try (Statement stmt = con.createStatement()) {
            stmt.executeUpdate(createSchemaQuery);
            System.out.println("Schema 'dev' created.");
        } catch (SQLException e) {
            System.out.println("Error creating schema: " + e.getMessage());
        }
        File ddlFolder = new File("ddl");
        File[] ddlFiles = ddlFolder.listFiles((dir, name) -> name.endsWith(".sql"));
        if (ddlFiles != null) {
            for (File ddlFile : ddlFiles) {
                try {
                    String ddlQuery = new String(Files.readAllBytes(ddlFile.toPath()));
                    if (ddlFile.getName().equals("tpch_create.sql")) {
                        // Ensure the 'tpch_create.sql' contains CREATE TABLE queries
                        if (ddlQuery.toUpperCase().contains("CREATE TABLE")) {
                            try (Statement stmt = con.createStatement()) {
                                stmt.executeUpdate(ddlQuery);
                                System.out.println("Created table(s) from " + ddlFile.getName());
                            }
                        } else {
                            System.out.println("No CREATE TABLE queries found in tpch_create.sql.");
                        }
                    } else {
                        System.out.println("Skipping non-CREATE TABLE SQL file: " + ddlFile.getName());
                    }
                } catch (IOException e) {
                    System.out.println("Error reading DDL file: " + ddlFile.getName() + " - " + e.getMessage());
                } catch (SQLException e) {
                    System.out.println("Error executing DDL query for file " + ddlFile.getName() + " - " + e.getMessage());
                }
            }
        } else {
            System.out.println("No DDL files found in 'ddl' folder.");
        }
    }
```

Output:

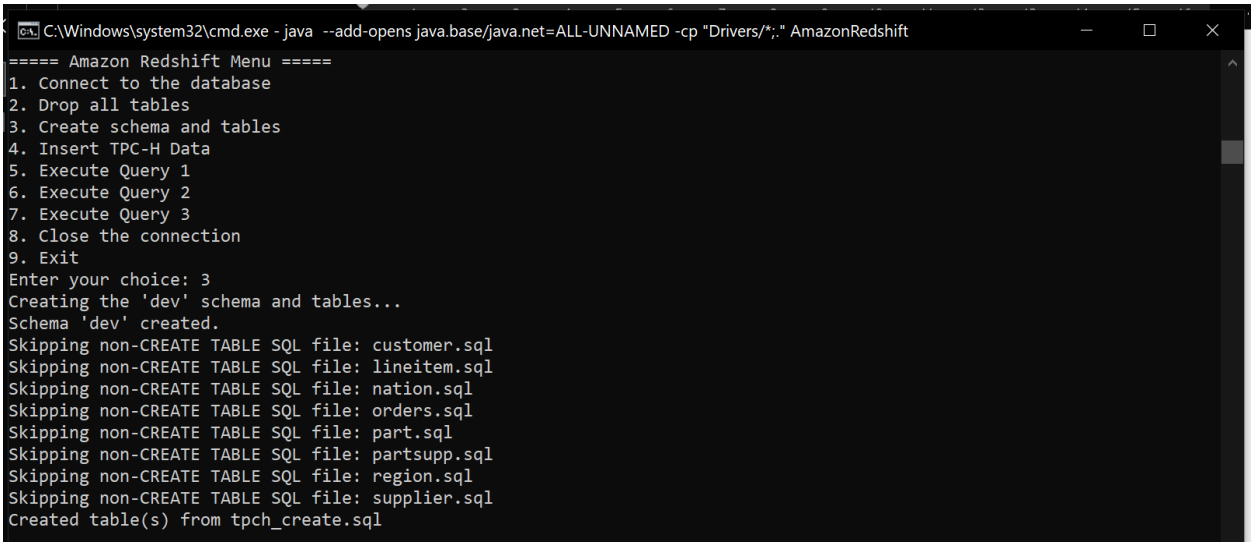


Fig. 6.4 *Amazon Redshift Menu executing the* ***Create schema and tables*** *operation, successfully creating the 'dev' schema and tables from SQL scripts.*

5. Write the method insert() to add the standard TPC-H data into the database. The DDL files are in the ddl folder. Hint: Files are designed so can read entire file as a string and execute it as one statement. May need to divide up into batches for large files. [10]

Code:

```
public void insert() {
        File dataFolder = new File("ddl");
        File[] dataFiles = dataFolder.listFiles((dir, name) -> name.endsWith(".sql") && !name.equals("tpch_create.sql"));

        if (dataFiles != null) {
            // Use a CountDownLatch to wait for all insertions to complete
            CountDownLatch latch = new CountDownLatch(dataFiles.length);

            for (File dataFile : dataFiles) {
                executorService.submit(() -> {
                    processFile(dataFile);
                    latch.countDown();  // Decrease latch count when each file is processed
                });
            }
            try {
                // Wait until all insert tasks are completed
                latch.await();
                System.out.println("All data insertions completed.");
            } catch (InterruptedException e) {
                System.out.println("Error waiting for insertions to complete: " + e.getMessage());
            }
        } else {
            System.out.println("No data files found in the folder.");
```

```java
        }
    }
    private void processFile(File dataFile) {
        try {
            // Read the SQL query from the file
            String sqlQuery = new String(Files.readAllBytes(dataFile.toPath()));

            // Log the start of the process
            System.out.println("Processing file: " + dataFile.getName());

            // Open the Statement for executing the insert query
            try (Statement stmt = con.createStatement()) {
                // Split the SQL query into individual statements (assuming multiple INSERT statements in the file)
                String[] insertStatements = sqlQuery.split(";");
                int totalStatements = insertStatements.length; // Total number of statements in the file

                int batchCount = 0;
                int recordCount = 0;

                // Process each INSERT statement
                for (String statement : insertStatements) {
                    if (statement.trim().isEmpty()) continue;

                    // Add the statement to the batch
                    stmt.addBatch(statement.trim());
                    batchCount++;

                    // Execute the batch every 500 records
                    if (batchCount % 500 == 0) {
                        stmt.executeBatch(); // Execute the batch
                        recordCount += 500;

                        // Calculate and display progress
                        double percentageCompleted = (recordCount / (double) totalStatements) * 100;
                        int remainingRecords = totalStatements - recordCount;
                        System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
                                dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
                    }
                }
                if (batchCount > 0) {
                    stmt.executeBatch();
                    recordCount += batchCount;
                    double percentageCompleted = (recordCount / (double) totalStatements) * 100;
                    int remainingRecords = totalStatements - recordCount;
                    System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
                            dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
```

```
                }

                System.out.println("Insertion completed for file: " + dataFile.getName());

            } catch (SQLException e) {
                System.out.println("Error executing statement: " + e.getMessage());
                e.printStackTrace();
            }

        } catch (IOException e) {
            System.out.println("Error reading file: " + dataFile.getName() + " - " + e.getMessage());
            e.printStackTrace(); // Optional: log the stack trace for more details
        }
    }
```

Output:

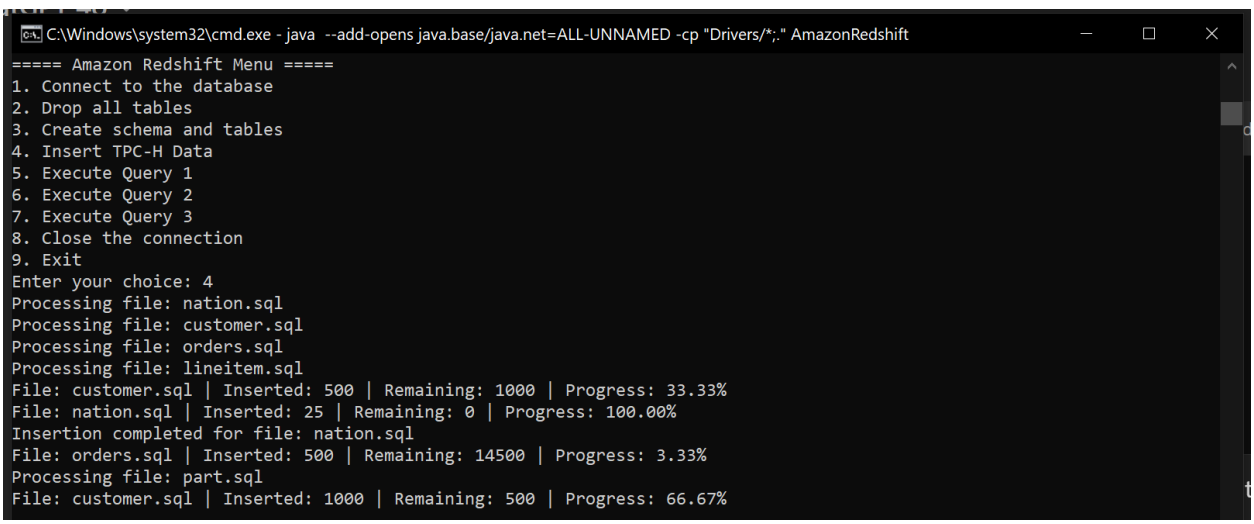


Fig. 6.5 *Amazon Redshift Menu executing the* ***Insert TPC-H Data*** *operation, processing multiple SQL files and showing insertion progress for each file.*

6. Write the method query1() that returns the most recent top 10 orders with the total sale and the date of the order for customers in America. [5]

Code:

```
public ResultSet query1() throws SQLException {
        System.out.println("Executing query #1.");
        String query = " SELECT o.O_ORDERKEY AS order_key, " +
                    " SUM(l.L_EXTENDEDPRICE) AS total_sale, " +
                    " o.O_ORDERDATE AS order_date " +
                    " FROM ORDERS o " +
```

```
                        " JOIN LINEITEM l ON o.O_ORDERKEY = l.L_ORDERKEY " +
                        " JOIN CUSTOMER c ON o.O_CUSTKEY = c.C_CUSTKEY " +
                        " JOIN NATION n ON c.C_NATIONKEY = n.N_NATIONKEY " +
                        " WHERE n.N_NAME = 'UNITED STATES' " +
                        " GROUP BY o.O_ORDERKEY, o.O_ORDERDATE " +
                        " ORDER BY o.O_ORDERDATE DESC " +
                        " LIMIT 10 ";
        if (con == null) {
            throw new SQLException("Connection is null. Please connect to the database first.");
        }

        Statement stmt = con.createStatement();
        return stmt.executeQuery(query);
    }
```

Output:

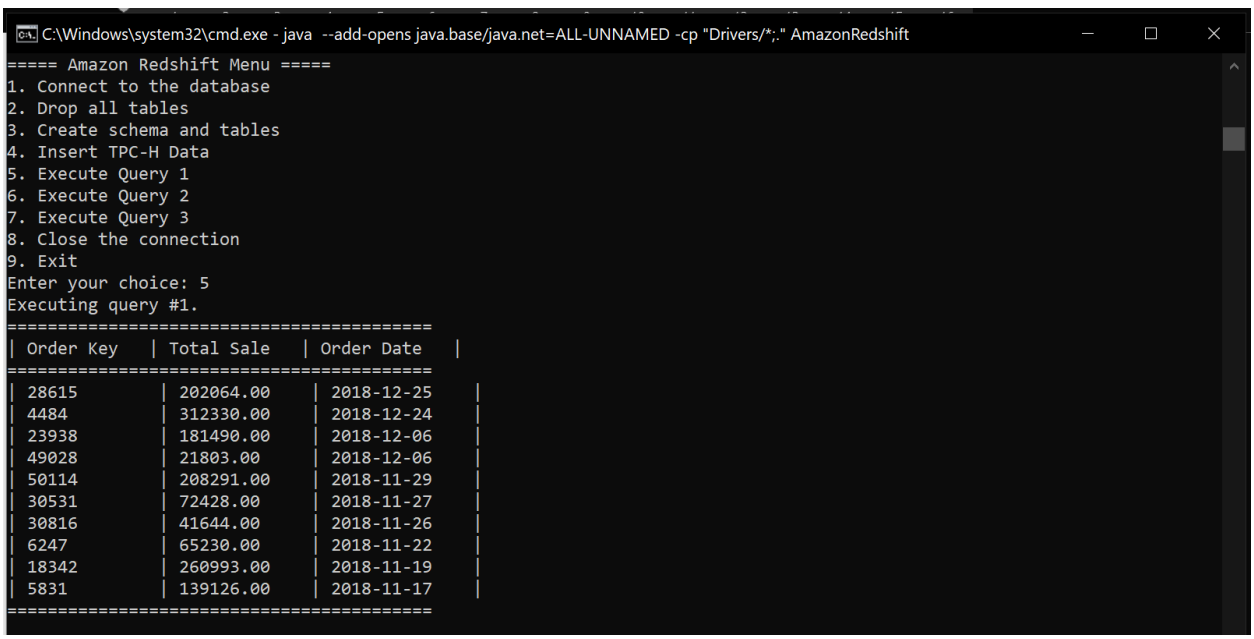


Fig. 6.6 *Amazon Redshift Menu executing* ***Query 1****, displaying a table with* ***Order Key****,* ***Total Sale****, and* ***Order Date*** *columns as query results.*

7. Write the method query2() that returns the customer key and the total price a customer spent in descending order, for all urgent orders that are not failed for all customers who are outside Europe and belong to the largest market segment. The largest market segment is the market segment with the most customers. [10]

Code:

```
public ResultSet query2() throws SQLException {
        System.out.println("Executing query #2.");
        String segmentQuery = "SELECT C_MKTSEGMENT " +
```

```
                                "FROM CUSTOMER " +
                                "GROUP BY C_MKTSEGMENT " +
                                "ORDER BY COUNT(C_CUSTKEY) DESC " +
                                "LIMIT 1";

        Statement stmt = con.createStatement();
        ResultSet segmentResult = stmt.executeQuery(segmentQuery);
        String largestSegment = null;
        if (segmentResult.next()) {
            largestSegment = segmentResult.getString("C_MKTSEGMENT");
        }
        if (largestSegment != null) {
            String query = "WITH NonEuropeanCustomers AS ( " +
                           "    SELECT C.C_CUSTKEY " +
                           "    FROM CUSTOMER C " +
                           "    JOIN NATION N ON C.C_NATIONKEY = N.N_NATIONKEY " +
                           "    JOIN REGION R ON N.N_REGIONKEY = R.R_REGIONKEY " +
                           "    WHERE R.R_NAME != 'EUROPE' " +
                           "), " +
                           "FilteredCustomers AS ( " +
                           "    SELECT C.C_CUSTKEY " +
                           "    FROM CUSTOMER C " +
                           "    WHERE C.C_MKTSEGMENT = ? " +
                           "    AND C.C_CUSTKEY IN (SELECT C_CUSTKEY FROM NonEuropeanCustomers) " +
                           "), " +
                           "UrgentOrders AS ( " +
                           "    SELECT O.O_CUSTKEY AS CustomerKey, SUM(L.L_EXTENDEDPRICE) AS TotalSpent " +
                           "    FROM ORDERS O " +
                           "    JOIN LINEITEM L ON O.O_ORDERKEY = L.L_ORDERKEY " +
                           "    WHERE O.O_ORDERPRIORITY = '1-URGENT' " +
                           "      AND O.O_ORDERSTATUS != 'F' " +
                           "      AND O.O_CUSTKEY IN (SELECT C_CUSTKEY FROM FilteredCustomers) " +
                           "    GROUP BY O.O_CUSTKEY " +
                           ") " +
                           "SELECT U.CustomerKey, U.TotalSpent " +
                           "FROM UrgentOrders U " +
                           "ORDER BY U.TotalSpent DESC";

            PreparedStatement preparedStatement = con.prepareStatement(query);
            preparedStatement.setString(1, largestSegment); // Set the largest market segment dynamically
            return preparedStatement.executeQuery();
        }
        System.out.println("No largest market segment found.");
        return null;
    }
```

Output:

```
===== Amazon Redshift Menu =====
1. Connect to the database
2. Drop all tables
3. Create schema and tables
4. Insert TPC-H Data
5. Execute Query 1
6. Execute Query 2
7. Execute Query 3
8. Close the connection
9. Exit
Enter your choice: 6
Executing query #2.
Query #2 Results:
========================================
Customer Key    Total Spent
========================================
962             934276.00
1052            833829.00
103             772835.00
1279            740936.00
1061            734090.00
664             658041.00
1331            654336.00
1415            622849.00
334             620177.00
1316            611696.00
1334            601828.00
1144            601810.00
1345            574253.00
340             570314.00
1013            555097.00
1027            545509.00
694             541472.00
1253            538708.00
818             529872.00
1124            526399.00
229             522166.00
380             513735.00
835             511995.00
575             504550.00
1214            503437.00
1268            489049.00
188             475074.00
767             457100.00
995             456724.00
932             453079.00
134             438663.00
1486            437668.00
1075            425282.00
512             422890.00
1               421274.00
329             413687.00
662             404588.00
844             403041.00
649             402030.00
674             400317.00
508             395928.00
1223            386810.00
```

814	382438.00
1046	374573.00
803	368071.00
592	366214.00
938	365671.00
185	363520.00
580	359714.00
1414	355840.00
709	355246.00
968	353066.00
553	344943.00
298	334711.00
1163	334638.00
1100	327751.00
1009	327475.00
1115	319143.00
805	316412.00
568	312407.00
1433	308212.00
1237	308206.00
1202	308009.00
1091	306579.00
1400	305537.00
77	304452.00
1295	299133.00
860	297132.00
1430	296181.00
1235	292461.00
1453	292215.00
220	290881.00
610	284676.00
152	283341.00
476	279524.00
722	275900.00
986	271477.00
73	259207.00
428	257247.00
116	248579.00
1318	244310.00
223	244030.00
1208	243368.00
211	242005.00
1405	238524.00
1475	237839.00
944	237457.00
40	235070.00
475	234605.00
790	233068.00
280	232699.00
1201	230542.00
392	224661.00
523	224453.00
1183	220834.00
1460	217412.00
98	215161.00
1312	212232.00
653	210170.00
400	204820.00
1040	198800.00
1114	194242.00
1006	193509.00

224	191248.00
278	191091.00
1085	190796.00
1180	189196.00
1412	187654.00
865	183851.00
766	182136.00
763	175047.00
170	171616.00
1196	169535.00
1447	167667.00
1184	166621.00
64	164606.00
350	162400.00
200	159039.00
784	146507.00
113	146103.00
826	144242.00
670	143332.00
515	142196.00
448	141324.00
328	139529.00
802	138073.00
1396	130749.00
811	128303.00
347	117633.00
1358	117503.00
905	116995.00
548	115519.00
1330	112946.00
1357	109970.00
1370	108208.00
1261	106892.00
562	105994.00
904	101642.00
1385	101477.00
296	100987.00
542	99021.00
1082	96900.00
728	96476.00
793	93905.00
647	91988.00
535	91529.00
859	90163.00
518	89866.00
602	89745.00
13	86102.00
419	81138.00
121	77998.00
1277	76457.00
109	74685.00
557	73613.00
205	71835.00
32	64622.00
1468	58353.00
623	53127.00
785	48653.00
890	43337.00
1033	41460.00
386	33790.00
221	32520.00

```
1292            28556.00
478             10756.00
```

Fig. 6.7 *Amazon Redshift Menu executing* ***Query 2****, displaying the query results with* ***Customer Key*** *and corresponding* ***Total Spent*** *values in descending order.*

8. Write the method query3() that returns a count of all the line items that were ordered within the six years starting on April 1st, 1997 group by order priority. Make sure to sort by order priority in ascending order. [10]

Code:

```
public ResultSet query3() throws SQLException {
        System.out.println("Executing query #3.");
        String query = "SELECT o.O_ORDERPRIORITY, COUNT(l.L_LINENUMBER) AS lineitem_count " +
                       "FROM orders o " +
                       "JOIN lineitem l ON o.O_ORDERKEY = l.L_ORDERKEY " +
                       "WHERE o.O_ORDERDATE >= '1997-04-01' " +
                       "AND o.O_ORDERDATE < DATEADD(year, 6, '1997-04-01') " +
                       "GROUP BY o.O_ORDERPRIORITY " +
                       "ORDER BY o.O_ORDERPRIORITY ASC";
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);

        return rs;
    }
```

Output:

Fig. 6.8 *Amazon Redshift Menu executing* ***Query 3****, displaying the results with* ***Order Priority*** *and corresponding* ***Line Item Count*** *values.*

Full Code:

```
1. import java.math.BigDecimal;
2. import java.nio.file.Files;
3. import java.sql.Connection;
4. import java.sql.DriverManager;
5. import java.sql.PreparedStatement;
6. import java.sql.ResultSet;
7. import java.sql.ResultSetMetaData;
8. import java.sql.SQLException;
9. import java.sql.Statement;
10. import java.util.Arrays;
11. import java.util.List;
12. import java.util.Properties;
13. import java.util.Scanner;
14. import java.io.File;
15. import java.io.IOException;
16. import java.util.concurrent.CountDownLatch;
17. import java.util.concurrent.ExecutorService;
18. import java.util.concurrent.Executors;
19.
20. /**
21.  * Performs SQL DDL and SELECT queries on a MySQL database hosted on AWS RDS.
22.  *
23.  * java --add-opens java.base/java.net=ALL-UNNAMED -cp "Drivers/*;."
    AmazonRedshift
24.  */
25. public class AmazonRedshift {
26.     /**
27.      * Connection to database
28.      */
29.     static final String redshiftUrl = "jdbc:redshift://g23ai2087-
    cluster.cndskybccs1r.us-east-1.redshift.amazonaws.com:5439/dev";
30.     static final String masterUsername = "admin"; // Replace with your Redshift
    admin username
31.     static final String password = "IITj1234"; // Replace with your Redshift
    password
32.     private static ExecutorService executorService;
33.     /**
34.      * Main method is only used for convenience. Use JUnit test file to verify
    your
35.      * answer.
36.      *
37.      * @param args
38.      *              none expected
39.      * @throws SQLException
40.      *                      if a database error occurs
41.      */
42.     public static void main(String[] args) {
43.         Scanner scanner = new Scanner(System.in);
44.         AmazonRedshift q = new AmazonRedshift();
45.         // Initialize the executor service for parallel execution
46.         executorService = Executors.newFixedThreadPool(4);
47.         ResultSet rs;
```

```
        while (true) {
            // Display menu
            System.out.println("\n===== Amazon Redshift Menu =====");
            System.out.println("1. Connect to the database");
            System.out.println("2. Drop all tables");
            System.out.println("3. Create schema and tables");
            System.out.println("4. Insert TPC-H Data");
            System.out.println("5. Execute Query 1");
            System.out.println("6. Execute Query 2");
            System.out.println("7. Execute Query 3");
            System.out.println("8. Close the connection");
            System.out.println("9. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            try {
                switch (choice) {
                    case 1:
                        q.connect();
                        break;
                    case 2:
                        q.drop();
                        break;
                    case 3:
                        q.create();
                        break;
                    case 4:
                        q.insert();
                        break;

                    case 5:
                        if (q.con == null) {
                            System.out.println("Please connect to the database first.");
                        } else {
                            try {
                                rs = q.query1();

                                // Display header for the table
                                System.out.println("==============================================");
                                System.out.println("| Order Key    | Total Sale    | Order Date    |");
                                System.out.println("==============================================");

                                // Process and display each row in the ResultSet
                                while (rs.next()) {
                                    System.out.printf("| %-12d | %-12.2f | %-13s |\n",
                                                      rs.getInt("order_key"),  // Column alias from the query
                                                      rs.getDouble("total_sale"),  // Column alias from the query
```

```
                                                        rs.getDate("order_date"));
      // Column alias from the query
                                }

                                System.out.println("==============================
   =============");
                            } catch (SQLException e) {
                                    System.out.println("Error processing result
   set: " + e.getMessage());
                                }
                            }
                            break;
                        case 6:
                            try {
                                rs = q.query2();
                                if (rs != null) {
                                    System.out.println("Query #2 Results:");
                                    System.out.println("==========================
   ================");
                                    System.out.printf("%-15s %-20s%n", "Customer
   Key", "Total Spent");
                                    System.out.println("==========================
   ================");
                                    while (rs.next()) {
                                        System.out.printf("%-15d %-20.2f%n",
                                                rs.getInt("CustomerKey"),
                                                rs.getDouble("TotalSpent"));
                                    }
                                    System.out.println("==========================
   ================");
                                } else {
                                    System.out.println("No results found for Query
   #2.");
                                }
                            } catch (SQLException e) {
                                System.out.println("Error executing Query #2: " +
   e.getMessage());
                            }
                            break;
                        case 7:
                        if (q.con == null) {
                            System.out.println("Please connect to the database
   first.");
                        } else {
                            rs = q.query3();
                            System.out.println("==================================
   ========");
                            System.out.println("| Order Priority   | Line Item
   Count     |");
                            System.out.println("==================================
   ========");
                            while (rs.next()) {
                                String orderPriority =
   rs.getString("O_ORDERPRIORITY");
```

```
                                int lineItemCount = rs.getInt("lineitem_count");
                                System.out.printf("| %-16s | %-18d |\n", orderPriority, lineItemCount);
                            }
                            System.out.println("==========================================");
                        }
                            break;
                        case 8:
                            q.close();
                            break;
                        case 9:
                            System.out.println("Exiting program...");
                            scanner.close();
                            System.exit(0);  // Exit the program
                            break;
                        default:
                            System.out.println("Invalid choice. Please try again.");
                            break;
                    }
                } catch (SQLException e) {
                    System.out.println("Error: " + e.getMessage());
                } catch (Exception e) {
                    System.out.println("An unexpected error occurred: " + e.getMessage());
                }
            }
        }

        /**
         * Makes a connection to the database and returns connection to caller.
         *
         * @return
         *         connection
         * @throws SQLException
         *                        if an error occurs
         */

        // Redshift connection details


        private Connection con;

        public Connection connect() {
            try {
                Class.forName("com.amazon.redshift.jdbc42.Driver");
                Properties properties = new Properties();
                properties.setProperty("user", masterUsername);
                properties.setProperty("password", password);

                this.con = DriverManager.getConnection(redshiftUrl, properties);

```

```
                System.out.println("Connection to Redshift established successfully.");
            } catch (ClassNotFoundException e) {
                System.out.println("Error: Redshift JDBC driver not found.");
                e.printStackTrace();
            } catch (SQLException e) {
                System.out.println("Failed to connect to the database.");
                e.printStackTrace();
            }
            return con;
        }
        /**
         * Closes connection to database.
         */
        public void close() {
            System.out.println("Closing database connection.");
            try {
                if (con != null && !con.isClosed()) {
                    con.close();
                }
            } catch (SQLException e) {
                System.out.println("Error closing the connection.");
            }
        }

        public void drop() {
            System.out.println("Dropping all tables in the 'dev' schema...");
            String dropQuery = "SELECT tablename FROM pg_tables WHERE schemaname = 'dev'";

            try (Statement stmt = con.createStatement()) {
                ResultSet rs = stmt.executeQuery(dropQuery);
                while (rs.next()) {
                    String tableName = rs.getString("tablename");
                    String dropTableQuery = "DROP TABLE IF EXISTS dev." + tableName;
                    stmt.executeUpdate(dropTableQuery);
                    System.out.println("Dropped table: " + tableName);
                }
            } catch (SQLException e) {
                System.out.println("Error dropping tables: " + e.getMessage());
            }
        }

        public void create() throws SQLException {
            System.out.println("Creating the 'dev' schema and tables...");
            String createSchemaQuery = "CREATE SCHEMA IF NOT EXISTS dev";
            try (Statement stmt = con.createStatement()) {
                stmt.executeUpdate(createSchemaQuery);
                System.out.println("Schema 'dev' created.");
            } catch (SQLException e) {
                System.out.println("Error creating schema: " + e.getMessage());
            }
            File ddlFolder = new File("ddl");
```

```
            File[] ddlFiles = ddlFolder.listFiles((dir, name) -> name.endsWith(".sql"));
            if (ddlFiles != null) {
                for (File ddlFile : ddlFiles) {
                    try {
                        String ddlQuery = new String(Files.readAllBytes(ddlFile.toPath()));
                        if (ddlFile.getName().equals("tpch_create.sql")) {
                            // Ensure the 'tpch_create.sql' contains CREATE TABLE queries
                            if (ddlQuery.toUpperCase().contains("CREATE TABLE")) {
                                try (Statement stmt = con.createStatement()) {
                                    stmt.executeUpdate(ddlQuery);
                                    System.out.println("Created table(s) from " + ddlFile.getName());
                                }
                            } else {
                                System.out.println("No CREATE TABLE queries found in tpch_create.sql.");
                            }
                        } else {
                            System.out.println("Skipping non-CREATE TABLE SQL file: " + ddlFile.getName());
                        }
                    } catch (IOException e) {
                        System.out.println("Error reading DDL file: " + ddlFile.getName() + " - " + e.getMessage());
                    } catch (SQLException e) {
                        System.out.println("Error executing DDL query for file " + ddlFile.getName() + " - " + e.getMessage());
                    }
                }
            } else {
                System.out.println("No DDL files found in 'ddl' folder.");
            }
        }
        public void insert() {
            File dataFolder = new File("ddl");
            File[] dataFiles = dataFolder.listFiles((dir, name) -> name.endsWith(".sql") && !name.equals("tpch_create.sql"));

            if (dataFiles != null) {
                // Use a CountDownLatch to wait for all insertions to complete
                CountDownLatch latch = new CountDownLatch(dataFiles.length);

                for (File dataFile : dataFiles) {
                    executorService.submit(() -> {
                        processFile(dataFile);
                        latch.countDown();  // Decrease latch count when each file is processed
                    });
                }
                try {
                    // Wait until all insert tasks are completed
```

```
                    latch.await();
                    System.out.println("All data insertions completed.");
                } catch (InterruptedException e) {
                    System.out.println("Error waiting for insertions to complete: " + e.getMessage());
                }
            } else {
                System.out.println("No data files found in the folder.");
            }
        }
        private void processFile(File dataFile) {
            try {
                // Read the SQL query from the file
                String sqlQuery = new String(Files.readAllBytes(dataFile.toPath()));

                // Log the start of the process
                System.out.println("Processing file: " + dataFile.getName());

                // Open the Statement for executing the insert query
                try (Statement stmt = con.createStatement()) {
                    // Split the SQL query into individual statements (assuming multiple INSERT statements in the file)
                    String[] insertStatements = sqlQuery.split(";");
                    int totalStatements = insertStatements.length; // Total number of statements in the file

                    int batchCount = 0;
                    int recordCount = 0;

                    // Process each INSERT statement
                    for (String statement : insertStatements) {
                        if (statement.trim().isEmpty()) continue;

                        // Add the statement to the batch
                        stmt.addBatch(statement.trim());
                        batchCount++;

                        // Execute the batch every 500 records
                        if (batchCount % 500 == 0) {
                            stmt.executeBatch(); // Execute the batch
                            recordCount += 500;

                            // Calculate and display progress
                            double percentageCompleted = (recordCount / (double) totalStatements) * 100;
                            int remainingRecords = totalStatements - recordCount;
                            System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
                                    dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
                        }
                    }
                    if (batchCount > 0) {
```

```
                        stmt.executeBatch();
                        recordCount += batchCount;
                        double percentageCompleted = (recordCount / (double) totalStatements) * 100;
                        int remainingRecords = totalStatements - recordCount;
                        System.out.printf("File: %s | Inserted: %d | Remaining: %d | Progress: %.2f%%%n",
                                dataFile.getName(), recordCount, remainingRecords, percentageCompleted);
                    }

                    System.out.println("Insertion completed for file: " + dataFile.getName());

                } catch (SQLException e) {
                    System.out.println("Error executing statement: " + e.getMessage());
                    e.printStackTrace();
                }

            } catch (IOException e) {
                System.out.println("Error reading file: " + dataFile.getName() + " - " + e.getMessage());
                e.printStackTrace(); // Optional: log the stack trace for more details
            }
        }


        /**
         * Query returns the most recent top 10 orders with the total sale and the date
         * of the
         * order in `America`.
         *
         * @return
         *         ResultSet
         * @throws SQLException
         *                     if an error occurs
         */
        public ResultSet query1() throws SQLException {
            System.out.println("Executing query #1.");
            String query = " SELECT o.O_ORDERKEY AS order_key, " +
                        " SUM(l.L_EXTENDEDPRICE) AS total_sale, " +
                        " o.O_ORDERDATE AS order_date " +
                        " FROM ORDERS o " +
                        " JOIN LINEITEM l ON o.O_ORDERKEY = l.L_ORDERKEY " +
                        " JOIN CUSTOMER c ON o.O_CUSTKEY = c.C_CUSTKEY " +
                        " JOIN NATION n ON c.C_NATIONKEY = n.N_NATIONKEY " +
                        " WHERE n.N_NAME = 'UNITED STATES' " +
                        " GROUP BY o.O_ORDERKEY, o.O_ORDERDATE " +
                        " ORDER BY o.O_ORDERDATE DESC " +
                        " LIMIT 10 ";
            if (con == null) {
```

```
                throw new SQLException("Connection is null. Please connect to the database first.");
            }

            Statement stmt = con.createStatement();
            return stmt.executeQuery(query);
        }



        /**
         * Query returns the customer key and the total price a customer spent in
         * descending
         * order, for all urgent orders that are not failed for all customers who are
         * outside Europe belonging
         * to the highest market segment.
         *
         * @return
         *         ResultSet
         * @throws SQLException
         *                      if an error occurs
         */
        public ResultSet query2() throws SQLException {
            System.out.println("Executing query #2.");
            String segmentQuery = "SELECT C_MKTSEGMENT " +
                                  "FROM CUSTOMER " +
                                  "GROUP BY C_MKTSEGMENT " +
                                  "ORDER BY COUNT(C_CUSTKEY) DESC " +
                                  "LIMIT 1";

            Statement stmt = con.createStatement();
            ResultSet segmentResult = stmt.executeQuery(segmentQuery);
            String largestSegment = null;
            if (segmentResult.next()) {
                largestSegment = segmentResult.getString("C_MKTSEGMENT");
            }
            if (largestSegment != null) {
                String query = "WITH NonEuropeanCustomers AS ( " +
                               "    SELECT C.C_CUSTKEY " +
                               "    FROM CUSTOMER C " +
                               "    JOIN NATION N ON C.C_NATIONKEY = N.N_NATIONKEY " +
                               "    JOIN REGION R ON N.N_REGIONKEY = R.R_REGIONKEY " +
                               "    WHERE R.R_NAME != 'EUROPE' " +
                               "), " +
                               "FilteredCustomers AS ( " +
                               "    SELECT C.C_CUSTKEY " +
                               "    FROM CUSTOMER C " +
                               "    WHERE C.C_MKTSEGMENT = ? " +
                               "    AND C.C_CUSTKEY IN (SELECT C_CUSTKEY FROM NonEuropeanCustomers) " +
                               "), " +
```

```
                                "UrgentOrders AS ( " +
                                "    SELECT O.O_CUSTKEY AS CustomerKey, SUM(L.L_EXTENDEDPRICE) AS TotalSpent " +
                                "    FROM ORDERS O " +
                                "    JOIN LINEITEM L ON O.O_ORDERKEY = L.L_ORDERKEY " +
                                "    WHERE O.O_ORDERPRIORITY = '1-URGENT' " +
                                "      AND O.O_ORDERSTATUS != 'F' " +
                                "      AND O.O_CUSTKEY IN (SELECT C_CUSTKEY FROM FilteredCustomers) " +
                                "    GROUP BY O.O_CUSTKEY " +
                                ") " +
                                "SELECT U.CustomerKey, U.TotalSpent " +
                                "FROM UrgentOrders U " +
                                "ORDER BY U.TotalSpent DESC";

                PreparedStatement preparedStatement = con.prepareStatement(query);
                preparedStatement.setString(1, largestSegment); // Set the largest market segment dynamically
                return preparedStatement.executeQuery();
            }
            System.out.println("No largest market segment found.");
            return null;
        }

        /**
         * Query returns all the lineitems that was ordered within the six years from
         * January 4th,
         * 1997 and the orderpriority in ascending order.
         *
         * @return
         *         ResultSet
         * @throws SQLException
         *                      if an error occurs
         */
        public ResultSet query3() throws SQLException {
            System.out.println("Executing query #3.");
            String query = "SELECT o.O_ORDERPRIORITY, COUNT(l.L_LINENUMBER) AS lineitem_count " +
                           "FROM orders o " +
                           "JOIN lineitem l ON o.O_ORDERKEY = l.L_ORDERKEY " +
                           "WHERE o.O_ORDERDATE >= '1997-04-01' " +
                           "AND o.O_ORDERDATE < DATEADD(year, 6, '1997-04-01') " +
                           "GROUP BY o.O_ORDERPRIORITY " +
                           "ORDER BY o.O_ORDERPRIORITY ASC";
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(query);

            return rs;
        }


        /*
```

```
     * Do not change anything below here.
     */
    /**
     * Converts a ResultSet to a string with a given number of rows displayed.
     * Total rows are determined but only the first few are put into a string.
     *
     * @param rst
     *                ResultSet
     * @param maxrows
     *                maximum number of rows to display
     * @return
     *         String form of results
     * @throws SQLException
     *                      if a database error occurs
     */
    public static String resultSetToString(ResultSet rst, int maxrows) throws SQLException {
        StringBuffer buf = new StringBuffer(5000);
        int rowCount = 0;
        ResultSetMetaData meta = rst.getMetaData();
        buf.append("Total columns: " + meta.getColumnCount());
        buf.append('\n');
        if (meta.getColumnCount() > 0)
            buf.append(meta.getColumnName(1));
        for (int j = 2; j <= meta.getColumnCount(); j++)
            buf.append(", " + meta.getColumnName(j));
        buf.append('\n');
        while (rst.next()) {
            if (rowCount < maxrows) {
                for (int j = 0; j < meta.getColumnCount(); j++) {
                    Object obj = rst.getObject(j + 1);
                    buf.append(obj);
                    if (j != meta.getColumnCount() - 1)
                        buf.append(", ");
                }
                buf.append('\n');
            }
            rowCount++;
        }
        buf.append("Total results: " + rowCount);
        return buf.toString();
    }

    /**
     * Converts ResultSetMetaData into a string.
     *
     * @param meta
     *             ResultSetMetaData
     * @return
     *         string form of metadata
     * @throws SQLException
     *                      if a database error occurs
     */
```

```
521.      public static String resultSetMetaDataToString(ResultSetMetaData meta)
   throws SQLException {
522.          StringBuffer buf = new StringBuffer(5000);
523.          buf.append(meta.getColumnName(1) + " (" + meta.getColumnLabel(1) + ",
   " +
524.                  meta.getColumnType(1) + "-" + meta.getColumnTypeName(1) + ", "
   +
525.                  meta.getColumnDisplaySize(1) + ", " + meta.getPrecision(1) +
   ", " +
526.                  meta.getScale(1) + ")");
527.          for (int j = 2; j <= meta.getColumnCount(); j++) {
528.              buf.append(", " + meta.getColumnName(j) + " (" +
   meta.getColumnLabel(j) + ", " +
529.                      meta.getColumnType(j) + "-" + meta.getColumnTypeName(j) +
   ", " +
530.                      meta.getColumnDisplaySize(j) + ", " + meta.getPrecision(j)
   + ", " +
531.                      meta.getScale(j) + ")");
532.          }
533.          return buf.toString();
534.      }
535.  }
```