CSE132A Winter 2019

JDBC Assignment

Due February 22, 11:59pm

The goal of the second project will be the development of a Java application that interacts with a local SQLite database using the **Java Database Connectivity** (JDBC) interface. **This is an individual project. The usual criteria of academic integrity apply. If you have any doubts on what is allowed, talk to the instructor or a TA.**

Computing Environment

- Your application will be built using SQLite
- The programming language will be Java. Please make sure JDK is correctly installed https://www.oracle.com/technetwork/java/javase/downloads/index.html
- You will use JDBC to connect to the database.
- A UNIX environment is recommended (such as ieng6)
- Please contact the TA for all questions about computing environment, connection to the database, and JDBC issues. However, TAs will not resolve for you Java programming issues.

Database Schema and Data Set

The database will hold data about direct flights between cities on given airlines. The database has one table, **Flight**, with attributes Airline, Origin, Destination (all 3 attributes are non-null character strings).

In addition to these tables, you may create any additional tables you need. Please make sure to drop at the end of the program all such additional helper tables you have created.

What you need to do

Write a Java program which, given a database with the above schema, outputs a table **Connected** with columns **Airline, Origin, Destination** (of type char(32)) and **Stops** (of type int). The result should contain the tuples <A, city1, city2, N> where city1 and city2 are distinct cities and N is the minimum number of stops needed to reach city2 from city1 by flights on airline A (if city2 is not reachable at all from city1 by flights on airline A then no such tuple should be produced).

Algorithm

Your algorithm should adapt the computation of transitive closure using semi-naïve evaluation, by additionally taking into account the airline and keeping track of the number of stops. You should **not**

use a recursive SQL query. Instead, the control needed for the recursion (i.e., the loop) should be implemented on the Java side.

The Java program is to be used primarily to send SQL commands via JDBC for execution on the database server. There should be no transfer of data between the database server and the client running Java except for Boolean values testing emptiness of relations on the server side or a single integer value per query denoting successful rows modified. The above restriction on data transfer means that the entire computation must be carried out using SQL commands on the server side. This will be checked explicitly in grading.

Input and Output

Your Java program must be in a file named **PA2.java** and this is the only file to be submitted on Gradescope.

The program should create a table in the database called **Connected**, with columns **Airline, Origin, Destination** (of type char(32) and **Stops** (of type int). No extra output to stdout is needed (but can be helpful for your debugging). We will compare the table **Connected** with the expected solution table directly on the database. Sample test cases are also provided.

Additional information

Local Database setup

The first step is to create local SQLite database called "pa2.db". Once you create the database pa2.db, your Java program can access pa2.db.

Steps to create local SQLite database "pa2.db"

You are given two input files (input1.sql and input2.sql). You can use either of the files to create the database. Commands (in **bold**) and outputs are given below:

This command removes the old database if any.

$ **rm pa2.db** (for windows use : **del pa2.db**) // continue even if it fails

This command creates an empty database with name pa2.db

$ **sqlite3 pa2.db**

SQLite version 3.7.9 2011-11-01 00:52:41

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

This command reads the input.sql, creates flight table and populates the table.

sqlite> **.read input1.sql** // This command might take one min.

This command saves the database on your machine with name pa2.db

sqlite> **.exit**

This command tell you the size of database

$ **ls -l pa2.db** (for windows use : **dir pa2.db**)

-rw-r--r-- 1 ben staff 16384 Nov 1 17:08 pa2.db

Your database is now ready to be used by your Java program. If you want to verify the content of the database or during debugging, you can access pa2.db through SQLite3's shell:

$ **sqlite3 pa2.db**

Enter ".help" for instructions

Enter SQL statements terminated with a ";"

This command tells you the tables in your database

sqlite> **.tables**

Flight

sqlite> **select count(*) from Flight;**

JDBC

In order to access your database using Java code, you will use JDBC. In order to use JDBC with SQLite you will need to download a JDBC driver which is freely available from the web site (see JDBC Resources below for more information on the driver and JDBC in general). This driver is simply a jar file (in this case named *sqlite-jdbc-3.8.7.jar*) that you will need to have on your machine. Keep this jar file and your PA2.java in the same directoty.

A complete working example is provided in the JDBC Sample section below.

JDBC Resources

- Oracle's JDBC Basic Tutorial

(https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html.)

Read all chapters up to Using Joins. This material is enough to carry out the project. Read the rest of the chapters if you want to learn about some more advanced features of JDBC.

- JDBC programming reference for Java 8.0

(https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html)

JDBC Sample: Connecting, Querying, and Processing the Results

The example below registers a JDBC driver, connects to the database, creates a Statement object, executes a query, and processes the result set. Remember when executing the program to add to the classpath the JDBC jar file (say, sqlite-jdbc-3.8.7.jar) you downloaded from SQLite-JDBC.

- Download the SQLite-JDBC jar: https://bitbucket.org/xerial/sqlite-jdbc/downloads/sqlite-jdbc-3.8.7.jar
- SQLite-JDBC also has a nice tutorial https://bitbucket.org/xerial/sqlite-jdbc/wiki/Home

Steps to compile and execute the Java program:

Use case #1: Command line for Unix and Mac machines

You will have to declare it as part of the classpath to the java command when you execute your compiled .class file (i.e. java -cp .:sqlite-jdbc-3.8.7.jar YourClass, notice the ".:" adds also the current directory! Notice the classname is without any suffix).

Go to the directory where PA2.java and sqlite-jdbc-3.8.7.jar are stored and run these commands.

$ **javac PA2.java**

$ **java -cp .:sqlite-jdbc-3.8.7.jar PA2** # PA2 is without .java or .class!

For Windows machine, run this:

$ **javac PA2.java**

$ **java -cp ".;sqlite-jdbc-3.8.7.jar" PA2**

Your output table “connected” will be stored in your local db pa2.db. You can verify the content of your local database using steps mentioned above.

Use case #2: Eclipse

Choose your project → File → Properties → (left) Java Build Path → Libraries tab → Add External JARs → choose the SQLite-JDBC jar → OK.

In order to avoid any issues, **we recommend that you keep all of the following files in the same directory:**

pa2.db
sqlite-jdbc-3.8.7.jar
PA2.java
input1.sql
input2.sql
sqlite3.exe (in case you are using Windows)

```
/**
 * This Java program exemplifies the basic usage of JDBC.
 * Requirements:
 *   (1) JDK 8.0+
 *   (2) SQLite3.
 *   (3) SQLite3 JDBC jar (https://bitbucket.org/xerial/sqlite-
jdbc/downloads/sqlite-jdbc-3.8.7.jar).
 */

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class PA2 {

  public static void main(String[] args) {

    Connection conn = null;  // Database connection.

    try {
      // Load the JDBC class.
      Class.forName("org.sqlite.JDBC");
      // Get the connection to the database.
      // - "jdbc" : JDBC connection name prefix.
      // - "sqlite" : The concrete database implementation
      // (e.g., sqlserver, postgresql).
      // - "pa2.db" : The name of the database. In this project,
      // we use a local database named "pa2.db".  This can also
```

```
 // be a remote database name.
 conn = DriverManager.getConnection("jdbc:sqlite:pa2.db");
 System.out.println("Opened database successfully.");

 // Use case #1: Create and populate a table.
 // Get a Statement object.
 Statement stmt = conn.createStatement();
 stmt.executeUpdate("DROP TABLE IF EXISTS Student;");
// Student table is being created just as an example. You
// do not need Student table in PA2
stmt.executeUpdate(
     "CREATE TABLE Student(FirstName, LastName);");
 stmt.executeUpdate(
"INSERT INTO Student VALUES('F1','L1'),('F2','L2');");

 // Use case #2: Query the Student table with Statement.
 // Returned query results are stored in a ResultSet
 // object.
 ResultSet rset = stmt.executeQuery("SELECT * from Student;");

 // Print the FirstName and LastName columns.
 System.out.println ("\nStatement result:");
 // This shows how to traverse the ResultSet object.
 while (rset.next()) {
   // Get the attribute value.
   System.out.print(rset.getString("FirstName"));
   System.out.print("---");
   System.out.println(rset.getString("LastName"));
 }

 // Use case #3: Query the Student table with
 // PreparedStatement (having wildcards).
 PreparedStatement pstmt = conn.prepareStatement(
     "SELECT * FROM Student WHERE FirstName = ?;");
 // Assign actual value to the wildcard.
 pstmt.setString (1, "F1");
 rset = pstmt.executeQuery ();

 System.out.println ("\nPrepared statement result:");
 while (rset.next()) {
   System.out.print(rset.getString("FirstName"));
```

```
        System.out.print("---");
        System.out.println(rset.getString("LastName"));
      }

      // Close the ResultSet and Statement objects.
      rset.close();
      stmt.close();

    } catch (Exception e) {
      throw new RuntimeException("There was a runtime problem!", e);
    } finally {
      try {
        if (conn != null) conn.close();
      } catch (SQLException e) {
        throw new RuntimeException(
          "Cannot close the connection!", e);
      }
    }
  }
}
```