EECS3421, Summer 2020
Assignment 2
Interactive & Embedded SQL Queries

Due date: *Sat, Jun 13 at 11:59pm*

Instructions

1. Read this assignment thoroughly before you proceed. Failure to follow instructions can affect your grade.
2. Download the database schema **a2.ddl** from the course website.
3. Download the file **a2.sql** from the course website.
4. Download the java skeleton file **Assignment2.java** from the course website.
5. Submit your work **electronically** using UNIX *submit*. Your submission must include the following files:

a) a2.sql	your queries for the interactive SQL part of the assignment (can include any view creation statement). If you define any **views** for a question, you must drop them after you have populated the answer table for that question.
b) Assignment2.java	your java code for the embedded SQL part of the assignment. Be careful to submit the .java file (**not** the .class file.). To get you started, we provide a skeleton of this file that you must download from the assignment webpage. Also, make sure that your java file **does not** include any `main()` function.
c) team.txt	a file that includes information about the team members (first name, last name, student ID, login, yorku email).

How to Submit

When you have completed the assignment, move or copy your files in a directory (e.g., assignment2), and use the following command to electronically submit your files:

```
% submit 3421 a2 a2.sql Assignment2.java team.txt
```

You can also submit the files individually after you complete each part of the assignment– simply execute the *submit* command and give the filename that you wish to submit. Make sure you name your files **exactly** as stated (including lower/upper case letters). Failure to do so will result in a mark of 0 being assigned. You may check the status of your submission using the command:

```
% submit -l 3421 a2
```

Interactive SQL Queries [50 marks]

In this section, you must edit the file **a2.sql** and add SQL statements that can **be run in *psql* on Prism machines.** Your SQL statements can create views and queries that will populate the result tables *query1, query2, …, query10* with tuples that satisfy the questions below. In order to ensure that everything is run in the correct order (by the markers or the automarker), you should add all your SQL statements in the file ***a2.sql*** that we have provided. This file can be read and executed using the *psql* command:

\i <FILENAME>

You can assume that the **a2.ddl** file has been read and executed in psql, before your **a2.sql** file is executed.

Follow these rules:

- The output of each query must be stored in a result table. We provide the definitions of these tables in the **a2.ddl** file (*query1, query2, …, query10*).
- For each of the queries below, your final statement should populate the respective answer table (*queryX*) with the correct tuples. It should look something like:

  "INSERT INTO queryX (SELECT … <complete your SQL query here> …)"

  where X is the correct index [1, …,10].
- In order to answer each of the questions, you are encouraged to create virtual **views** that can keep intermediate results and can be used to build your final INSERT INTO *queryX* statement. Do not create actual tables. Remember that you have to drop the views you have created, after each INSERT INTO *queryX* statement (i.e., after you have populated the result table).
- Your tables **must** match the output tables specified for each query. The attribute names **must** be **identical** to those specified in italics, and they must be in the specified order. Also, make sure to sort the results according to the attributes and ordering we specify in each question.
- We are not providing a sample database to test your answers, but you are encouraged to create one. We will test the validity of your queries against our own test database.
- All of your statements must run on PostgreSQL on the Prism machines, so be sure to populate your tables with test data and run all your statements on Prism prior to submission.

NOTE: Failure to do follow the instructions may cause your queries to fail when (automatically) tested, and you will lose marks.

Express the following queries in SQL.

1. [5 **marks**] For each country, find its neighbor country with the highest elevation point. Report the id and name of the country and the id and name of its neighboring country.

Output Table:	**query1**		
Attributes:	*c1id*	(country id)	[INTEGER]
	c1name	(country name)	[VARCHAR(20)]
	c2id	(neighbor country id)	[INTEGER]
	c2name	(neighbor country name)	[VARCHAR(20)]
Order by:	*c1name*	ASC	

2. [5 **marks**] Find the landlocked countries. A landlocked country is a country entirely enclosed by land (e.g., Switzerland). Report the id(s) and name(s) of the landlocked countries.

Output Table:	**query2**		
Attributes:	*cid*	(landlocked country id)	[INTEGER]
	cname	(landlocked country name)	[VARCHAR(20)]
order by:	cname	ASC	

3. [5 **marks**] Find the landlocked countries which are surrounded by exactly one country. Report the id and name of the landlocked country, followed by the id and name of the country that surrounds it.

Output Table:	**query3**		
Attributes:	*c1id*	(landlocked country id)	[INTEGER]
	c1name	(landlocked country name)	[VARCHAR(20)]
	c2id	(surrounding country id)	[INTEGER]
	c2name	(surrounding country name)	[VARCHAR(20)]
Order by:	*c1name*	ASC	

4. [5 **marks**] Find the accessible ocean(s) of each country. An ocean is accessible by a country if either the country has an ocean coastline itself (direct access to the ocean) or the country is neighboring another country that has an ocean coastline (indirect access). Report the name of the country and the name of the accessible ocean(s).

Output Table:	**query4**		
Attributes:	*cname*	(country name)	[VARCHAR(20)]
	oname	(ocean name)	[VARCHAR(20)]
Order by:	*cname*	ASC,	
	oname	DESC	

5. [5 **marks**] Find the top-10 countries with the highest average Human Development Index (HDI) over the 5-year period of 2009-2013 (inclusive).

Output Table:	**query5**		
Attributes:	*cid*	(country id)	[INTEGER]
	cname	(country name)	[VARCHAR(20)]
	avg*hdi*	(country's average HDI)	[REAL]
Order by:	avg*hdi*	DESC	

6. [5 **marks**] Find the countries that their Human Development Index (HDI) is constantly increasing over the 5-year period of 2009-2013 (inclusive). Constantly increasing means that from year to year there is a positive change (increase) in the country's HDI.

Output Table:	**query6**		
Attributes:	*cid*	(country id)	[INTEGER]
	cname	(country name)	[VARCHAR(20)]
Order by:	*cname*	ASC	

7. [5 **marks**] Find the total number of people in the world that follow each religion. Report the id of the religion, the name of the religion and the respective number of people that follow it.

Output Table:	**query7**		
Attributes:	rid	(religion id)	[INTEGER]
	rname	(religion name)	[VARCHAR(20)]
	followers	(number of followers)	[INTEGER]
Order by:	*followers*	DESC	

8. [5 **marks**] Find all the pairs of neighboring countries that have the same most popular language. For example, *<Canada, USA, English>* is one example tuple because in both countries, English is the most popular language; <Chile, Argentina, Spanish> can be another tuple, and so on. Report the names of the countries and the name of the language.

Output Table:	**query8**		
Attributes:	c1name	(country name)	[VARCHAR(20)]
	c2name	(neighboring country name)	[VARCHAR(20)]
	lname	(language name)	[VARCHAR(20)]
Order by:	lname	ASC,	
	c1name	DESC	

9. [5 **marks**] Find the country with the larger span between the country's highest elevation point and the depth of its deepest ocean. If a country has no direct access to an ocean, you should assume that its ocean's depth is 0. Report the name of the country and the total span.

Output Table:	**query9**		
Attributes:	cname	(country name)	[VARCHAR(20)]
	totalspan	(total span)	[INTEGER]

10. [5 **marks**] Find the country with the longest borders (with all its neighboring countries). Report the country and the total length of its borders.

Output Table:	**query10**		
Attributes:	cname	(country name)	[VARCHAR(20)]
	borderslength	(length of borders)	[INTEGER]

Embedded SQL Queries [50 marks – 5 for each method]

For this part of the assignment, you will create the class **Assignment2.java** which will allow you to process queries using JDBC. We will use the standard tables provided in the **a2.ddl** for this assignment. If you feel you need an intermediate **view** to execute a query in a method, you must create it inside that method. You must also drop it before exiting that method.

Rules:

- Standard input and output must **not** be used. This will halt the "automarker" and you will probably end up with a zero.
- The *database*, *username*, and *password* must be passed as parameters, never "hard-coded".
- Be sure to close all unused statements and result sets.
- All return values will be *String*, *boolean* or *int* values.
- A successful action (Update, Delete) is when:
  - It doesn't throw an SQL exception, and
  - The number of rows to be updated or deleted is correct.

Class name	**Description**
Assignment2.java	Allows several interactions with a postgreSQL database.

Instance Variables (you may want to add more)

Type	**Description**
Connection	The database connection for this session.

Methods (you may want to add helper methods.)

Constructor	**Description**
Assignment2()	Identifies the postgreSQL driver using Class.forName method.

Method	**Description**
boolean connectDB(String URL, String username, String password)	Using the String input parameters which are the *URL*, *username*, and *password* respectively, establish the Connection to be used for this session. Returns **true** if the connection was successful.
boolean disconnectDB()	Closes the connection. Returns **true** if the closure was successful.
boolean insertCountry(int cid, String name, int height, int population)	Inserts a row into the country table. *cid* is the name of the country*, name* is the name of the country, *height* is the highest elevation point and *population* is the population of the newly inserted country. You have to check if the country with id *cid* exists. Returns **true** if the insertion was successful, **false** otherwise.
int getCountriesNext ToOceanCount(int oid)	Returns the number of countries in table "oceanAccess" that are located next to the ocean with id o*id*. Returns **-1** if an error occurs.
String getOceanInfo(int oid)	Returns a string with information of an ocean with id *oid*. The output should be formatted as "oid:oname:depth". Returns an empty string "" if the ocean does not exist.

Method	**Description**
boolean chgHDI(int cid, int year, float newHDI)	Changes the HDI value of the country c*id* for the year *year* to the HDI value supplied (newHDI). Returns **true** if the change was successful, **false** otherwise.
boolean deleteNeighbour(int c1id, c2id)	Deletes the neighboring relation between two countries. Returns **true** if the deletion was successful, **false** otherwise. You can assume that the neighboring relation to be deleted exists in the database. Remember that if c2 is a neighbor of c1, c1 is also a neighbour of c2.
String listCountryLanguages(int cid)	Returns a string with all the languages that are spoken in the country with id *cid*. The list of languages should follow the contiguous format described below, and contain the following attributes in the order shown: (**NOTE**: before creating the string order your results by *population*). "l1id:l1lname:l1population**\n**l2id:l2lname:l2population**\n**... " where: • *lid* is the id of the language. • *lname* is name of the language. • *population* is the number of people in a country that speak the language, note that you will need to compute this number, as it is not readily available in the database. Returns an empty string "" if the country does not exist.
boolean updateHeight(int cid, int decrH)	Decreases the height of the country with id *cid.* This decrease might happen due to the natural erosion (i.e. height - decrH). Returns **true** if the update was successful, **false** otherwise.
boolean updateDB()	Create a table containing all the countries which have a population over 100 million. The name of the table should be *mostPopulousCountries* and the attributes should be: • *cid* INTEGER (country id) • *cname* VARCHAR(20) (country name) Returns **true** if the database was successfully updated, **false** otherwise. Store the results in **ASC** order according to the country id (cid).