CSCC43 UTSC Fall 2022

Assignment 2

Interactive & Embedded SQL Queries

Due date: 5th November at 11:59 pm

Points: 80

Read the note on academic integrity: https://utsc.utoronto.ca/aacc/academic-integrity-matters

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 assignment webpage.
3. Download the file **a2.sql** from the assignment webpage.
4. Download the java skeleton file **Assignment2.java** from the assignment webpage.
5. Submit your work **electronically** using *MarkUs*. 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. Make sure 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.

Part 0: GitHub Repo (For Group Work Only)

While we allow you to work with partner(s), we don't want people to excessively 'split up' the work, or in the worst case, one person just does everything. This defeats the purpose of group work. Each person should attempt the assignment on their own first, then discuss your solutions with your partner(s). It's ok if you don't work on all the queries yourself, but you should do most(>=80%).

For group work, you must create a GitHub repo with the link below. Each member should create their own branch with their name, and work on their branch incrementally. Then, discuss with your partner(s) and merge code to master accordingly. Include your group name as a first comment in the a2.sql file you submit.

If we see disproportionate efforts between group members, or excessive work division, then your mark will be multiplied by 0.5 to 0.9.

https://classroom.github.com/a/FOuaRNMG

Part 1: Interactive SQL Queries [50 marks – 5 for each query]

Introduction

You are a developer for 'Lilmons: Monster Collector', a game where players collect monsters by spending coins and battle with them among other things. You are in charge of writing reports that the higher-ups request in order get data on their players and make business decisions.

Prerequisites & Notes

Download the files **a2.ddl** and **a2.sql** from the assignment webpage. In this section, you will be editing the **a2.sql** file by writing SQL statements that will populate the result tables *Query1, Query2, ..., Query10* with tuples that satisfy the questions below. You can assume that the **a2.ddl** file has been 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 MathLab machine, so be sure to populate your tables with test data and run all your statements on MathLab prior to submission. **We will be running your a2.sql in psql on the mathlab machines using \i. So make sure each statement is delimited with a semicolon and your file is free of errors. A good way to check this is by doing this yourself.**

NOTE: Failure to follow the above instructions may cause your queries to fail when tested and will lead to a deduction in marks.

Make sure you have read and fully understood the schema presented in a2.ddl before continuing. Also, aside from the standard numeric functions (ie: COUNT(), AVG(), etc.) that you may find useful, a few functions and keywords that will certainly be helpful and we encourage you to look at are: CASE, CAST(), CONCAT(), COALESCE(), and NULLIF().

An important concept of the schema to note:

- A (player/guild) only has records in the (Player/Guild) Ratings table starting on the month of the creation of the (player/guild). They then get one record in these tables for every subsequent month. Furthermore, A (player/guild) is deemed inactive for a given month if for that month, their monthly rating is 0; if it is greater than 0, they were active that month. Also note that by design, these ratings tables are only populated at the end of the month, so new players or guilds created in the middle of the current month would not have any records in these tables yet.

Queries

1. We want to classify players as any of ‘whale’, ‘lucky’, and ‘hoarder’ if they fit the respective criteria. A player is a ‘whale’ if their average number of rolls per month they are active is at least 100 (only include players who have been active for at least one month). A player is ‘lucky’ if the number of rarity 5 lilmons they have in their inventory is at least 5% of the number of times they’ve rolled (only include players who have rolled at least once). A player is a ‘hoarder’ if their average number of coins they currently have per month they are active is at least 10,000

(eg: if a player has been active for 5 months, then they are a 'hoarder' if they currently have at least 50,000 coins; only include players who have been active for at least one month). It is possible for a single player to be classified as multiple of these or none of these. Therefore, the way we will handle this is to classify a player as exactly one of the following eight classifications: '--', '--hoarder', '-lucky-', '-lucky-hoarder', 'whale--', 'whale--hoarder', 'whale-lucky-', or 'whale-lucky-hoarder' (HINT: consider using a combination of the functions listed above). Also, you may have noticed that this classification setup resembles a binary number with 3 bits (ie: 000, 001, 010, 011, etc.) but instead of 0s and 1s, it is whether or not the player is classified as 'whale', 'lucky', or 'hoarder' or not.

Output table:	**Query1**		
Attributes:	*p_id*	(player's id)	[INTEGER]
	playername	(player's name)	[VARCHAR]
	email	(player's email)	[VARCHAR]
	classification	(classification as detailed above)	[VARCHAR]
Order by:	*p_id*	ASC	

2. We want to see how popular each lilmon element is. To do this, we associate each element e with a popularity_count. The popularity_count for e is the number of distinct player-lilmon pairs such that lilmon is of element e and player has lilmon favourited and/or has lilmon in their team.

Output table:	**Query2**		
Attributes:	*element*	(lilmon element)	[VARCHAR]
	popularity_count	(element's popularity_count as detailed above)	[INTEGER]
Order by:	*popularity_count*	DESC	

3. We want a report on server issues using our players' number of incomplete games. To do this, we want to find the average over all players of each player's average number of incomplete games per month they are active (if a player was active for 0 months, treat it as they were active for 1 month in this calculation to avoid division by zero).

Output table:	**Query3**		
Attributes:	*avg_ig_per_month_per_player*	(calculated average as detailed above)	[REAL]
Order by:	**N/A**		

4. We want to see how popular each individual lilmon is. To do this, we associate each lilmon with a popularity_count. The popularity_count for a particular lilmon is the number of distinct player-lilmon pairs such that player has lilmon favourited and/or has lilmon in their team.

Output table: **Query4**

Attributes:	*id*	(lilmon's id)	[INTEGER]
	name	(lilmon's name)	[VARCHAR]
	rarity	(lilmon's rarity)	[INTEGER]
	popularity_count	(lilmon's popularity_count as detailed above)	[INTEGER]
Order by:	*popularity_count*	DESC	
	rarity	DESC	
	id	DESC	

5. We want to find the players from North America (Canada, USA, or Mexico) who have performed really well and consistently within the last 6 months. This means that the player must have an all-time rating of at least 2000 at some point within these 6 months and their highest monthly rating (within the 6 months) and lowest monthly rating (within the 6 months) differ by no more than 50. **Note:** The PlayerRatings table or the GuildRatings table could be used to find the last 6 months, but to make things consistent from a testing perspective, please use the PlayerRatings table for this question.

Output table:	**Query5**		
Attributes:	*p_id*	(player's id)	[INTEGER]
	playername	(player's name)	[VARCHAR]
	email	(player's email)	[VARCHAR]
	min_mr	(player's lowest monthly rating in the past 6 months)	[INTEGER]
	max_mr	(player's highest monthly rating in the past 6 months)	[INTEGER]
Order by:	*max_mr*	DESC	
	min_mr	DESC	
	p_id	ASC	

6. We want to classify guilds based on their number of members and their skill level. A guild is considered 'large' if they have at least 500 members, 'medium' if they have 100-499 members, and 'small' if they have less than 100 members. Of course, it would not be fair to give guilds of all sizes the same criteria when judging their skill level; thus, we have created a system where the criteria scales based on guild size when judging a guild's skill level. First of all, we look at the guilds' all-time ratings from the latest month, if this is at least 2000 then they are considered 'elite', if it is 1500-1999 then they are considered 'average', any less than that and they are considered 'casual'; this is for 'large' guilds. For 'medium' guilds, they are considered 'elite' if their latest all-time rating is at least 1750, 'average' if it is between 1250 and 1749, and 'casual' if it is below 1250. 'small' guilds are 'elite' if their latest all-time rating is at least 1500, 'average' if between 1000 and 1499, and 'casual' if under 1000. For guilds that don't have an all-time rating for the latest month, classify them as 'new'. **Note:** The PlayerRatings table or the

GuildRatings table could be used to find the latest month, but to make things consistent from a testing perspective, please use the GuildRatings table for this question.

Output table:	**Query6**		
Attributes:	*g_id*	(guild's id)	[INTEGER]
	guildname	(guild's name)	[VARCHAR]
	tag	(guild's tag)	[VARCHAR(5)]
	leader_id	(guild leader's id)	[INTEGER]
	leader_name	(guild leader's name)	[VARCHAR]
	leader_country	(guild leader's country code)	[CHAR(3)]
	size	(guild size as detailed above)	[VARCHAR]
	classification	(guild classification as detailed above)	[VARCHAR]
Order by:	*g_id*	ASC	

7. We want to find the player retention per country. This is the average over a country's players' number of active months. However, don't include players with 0 active months in the calculation of the averages.

Output table:	**Query7**		
Attributes:	*country_code*	(country's code)	[CHAR(3)]
	player_retention	(country's player retention as detailed above)	[REAL]
Order by:	*player_retention*	DESC	

8. We want to know all players' win rates over their completed games. Also if a player is a member of a guild, we want to also be able to compare their win rate to the guild's members' aggregate win rate over the guild's members' total number of completed games.

Output table:	**Query8**		
Attributes:	*p_id*	(player's id)	[INTEGER]
	playername	(player's name)	[VARCHAR]
	player_wr	(player's win rate as detailed above)	[REAL]
	g_id	(guild's id)	[INTEGER]
	guildname	(guild's name)	[VARCHAR]
	tag	(guild's tag)	[VARCHAR(5)]
	guild_aggregate_wr	(guild's aggregate win rate as detailed above)	[REAL]
Order by:	*player_wr*	DESC	
	guild_aggregate_wr	DESC	

9. For each guild among the top 10 guilds, we want to find the country with the highest number of members within the guild from that country (report ties in separate rows; eg: if a guild has 7 members -- 3 from Canada, 3 from USA, and 1 from Mexico, report Canada and USA in 2 different rows). We also want to compare this with the guild's total number of members. The top 10 guilds should be found as follows: Look at the guild's ratings from the latest month, the first criteria is higher all-time rating; if there is ties between all-time rating then look for higher monthly rating; lastly, if there is also ties between monthly rating then simply look for lower guild id. **Note:** The PlayerRatings table or the GuildRatings table could be used to find the latest month, but to make things consistent from a testing perspective, please use the GuildRatings table for this question.

Output table:	**Query9**		
Attributes:	*g_id*	(guild's id)	[INTEGER]
	guildname	(guild's name)	[VARCHAR]
	monthly_rating	(guild's latest monthly rating)	[INTEGER]
	all_time_rating	(guild's latest all-time rating)	[INTEGER]
	country_pcount	(guild's number of members from country_code)	[INTEGER]
	total_pcount	(guild's total number of members)	[INTEGER]
	country_code	(guild's country with the highest number of members)	[CHAR(3)]
Order by:	*all_time_rating*	DESC	
	monthly_rating	DESC	
	g_id	ASC	

10. We want to find the average member veteran-ness for all guilds. A player's veteran-ness is the total number of months they have recorded in the PlayerRatings table, then divided by 12 (to get a value in years).

Output table:	**Query10**		
Attributes:	*g_id*	(guild's id)	[INTEGER]
	guildname	(guild's name)	[VARCHAR]
	avg_veteranness	(guild's average member veteranness as detailed above)	[REAL]
Order by:	*avg_veteranness*	DESC	
	g_id	ASC	

Part 2: JDBC and Embedded SQL Queries [30 marks – 3 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 in that method. You must also drop it before exiting that method.

Rules:

- Standard input and output must **not** be used.
- The database, username, and password must be passed as parameters, never "hard-coded". We will use your connectDB() method defined below to connect to the database with our own credentials.
- 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:

  o It doesn't throw an SQL exception, and
  o The number of rows to be updated or deleted is correct.

- Don't trim any return values i.e. keep all \n's
- If your code does not compile, you get a 0 on that query

Documentation & Jar:

You may find the official documentation to be of help. Sometimes when you are googling how to do something, it you will lead you straight to the documentation examples. Please keep in mind, the documentation may differ based on different versions. It's important to test stuff out in the MathLab environment versions. Don't use maven please, let's be old fashioned and manually add the jar in.

https://jdbc.postgresql.org/documentation/

https://jdbc.postgresql.org/download/postgresql-42.2.5.jar

Class Name

Class Name	Description
`Assignment2.java`	Allows several interactions with a postgreSQL database

Instance Variables (you may want to add more)

Class Name	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, **false** otherwise.
`boolean disconnectDB()`	Closes the connection. Returns **true** if the closure was successful, **false** otherwise.
`boolean insertPlayer(int id, String playerName, String email, String countryCode)`	Inserts a row into the player table. *id* is the id of the player, *playerName* is the name of the player, *email* is the email of the player, *countryCode* is the country of the player. You must check if the inputs satisfy the given DDL. For example, the *id*, *email*, and *playerName* should not already exist and the *countryCode* should be exactly 3 uppercase characters. Returns **true** if the insertion was successful, **false** otherwise.
`int getMembersCount(int gid)`	Returns the number of players in guild *gid*. Returns **-1** if an error occurs.
`String getPlayerInfo(int id)`	Returns a string with the information of the player with player id *id*. The output is "id:playername:email:countrycode:coins:rolls:wins:losses:total_battles:guild".

	Returns an empty string **“”** if the player does not exist.
`boolean changeGuild(String oldName, String newName)`	Changes the name of the guild with name *oldName* to *newName*. You must check if the inputs satisfy the given DDL. Returns **true** if a guild was successfully changed, **false** otherwise.
`boolean deleteGuild(String guildName)`	Deletes the guild with name *guildName*. Returns **true** if a guild was successfully deleted, **false** otherwise.
`String listAllTimePlayerRatings()`	Returns a string that describes the all-time player ratings from the latest month in the following format: “p1name:p1rating:p2name:p2rating…” where • *piname* is the name of the i-th player • *pirating* is the all-time rating of the i-th player in the latest month • **should be ordered from highest rating to lowest rating** • **do not need to include players who do not have a rating**
`boolean updateMonthlyRatings()`	Recall how the PlayerRatings and GuildRatings tables function (if you don’t remember read **a2.ddl** again). We want to insert records into the PlayerRatings and GuildRatings tables for the month of October 2022. For simplicity, this is how we will calculate each Players’ and Guilds’ new ratings: • For players and guilds who had a record for September 2022, increase their monthly rating by 10% and increase their all-time rating by 10%. • For players and guilds who did not have a record for September 2022 (this means they were a new player or guild

	created during October 2022), set their monthly and all-time rating both to 1000. Returns **true** if the insertion was successful, **false** otherwise.
`boolean createSquidTable()`	Create a table containing all the players that are in the guild “Squid Game” and are from the country Korea. The name of the table is *squidNation* and the attributes are: • *id* INTEGER (player’s id) • *playername* VARCHAR (player’s name) • *email* VARCHAR (player’s email) • *coins* INTEGER (player’s current coins) • *rolls* INTEGER (player’s number of rolls) • *wins* INTEGER (player’s win count) • *losses* INTEGER (player’s loss count) • *total_battles* INTEGER (player’s total number of battles) Returns **true** if the database was successfully updated, **false** otherwise. Store the results in **ASC** order according to the player’s *id*.
`boolean deleteSquidTable()` `(this does not count for marks, but you need to make it… or else…)`	Deletes the Table created above. Returns **true** if the database was successfully updated, **false** otherwise.