Introduction to Database Systems

eecs.yorku.ca/course_archive/2020-21/F/3421/project/jeopardy

EECS-3421 (A & B)

York University

Fall 2020

Project #3

Jeopardy: SQL Queries

Honesty

For projects, you are permitted to confer with others, seek advice, and (to a reasonable extent) help. However, remember that copying someone else's queries and claiming them as your own work is *plagiarism*. You must do your own work.

Description

Raccoon Rhapsody is a *quest-based* game.

People who sign up for *Raccoon Rhapsody* — call them *players* — have *accounts*. For each player's account, we keep track of the player's *name*, *address*, the date that they *joined*, a *cc#* (a valid credit card number we keep on file), and a *balance* (how much money in Canadian dollars they presently have in their game account). Each player will also have a unique *login*, which is the account's “name”. It is assumed that a given player will have just one account.

A player may create over time any number of *avatars*. An avatar is an in-game persona. Thus, an avatar *belongs to* a player (an account). An avatar has a *name*, *gender*, *look*, and skill *level*. There is a small set of *looks* available, which may be extended over time; e.g., “wizard”, “faerie”, “footballer”, “shark”, and “raccoon”.

The game's virtual word is divided up into a number of *realms*. On any given *day*, a player may log in (*visit*) as one of his or her avatars into one of the realms. (The player may not then switch avatars or realms for that day.) Of course, some days, a player might not log in at all. Realms will be intricate, amazing virtual worlds unto themselves! But, for the purposes of this database, we just need to know the realm's unique name (*realm*).

The staff of *Questeme* — the gaming house that makes *Raccoon Rhapsody* — creates daily *quests*, usually ensuring that each realm on each day has several *quests* available. Each quest on a given *day* in a *realm* has a different *theme*. Themes are repeated over time. Thus, a given quest is associated with a day, a realm, and a theme. Quests will be intricate, amazing challenges! But, for the purposes of this database, we just need to know the quest's name (*quest*). The quest's name is guaranteed to be unique on its day in its realm. (That is, there will not be two quests named the same thing on a given day within a given realm). And, for a quest, we keep the time it was completed, *if* it was completed.

A player when he or she logs into a realm — as one of his or her avatars — on a given day may just hang out. (Believe me, our realms are amazing! The most amazing realms you've ever hung out in!) Or their avatar can join one of realm's daily quests. We call this *acting* in a quest. When an avatar (the player as the avatar) joins a quest, they choose a *role* to play.

Associated with a quest is *loot*. Each piece of loot is a type of *treasure*. Each treasure type has a unique name (*treasure*), and has a worth associated with it in the game's in-world currency, commonly called “scrip” in gaming parlance. Our in-world currency officially is *sql*, for *Standard Quest Loonie*. (Players can turn in pieces of loot they have received for the *sql*, or keep the loot. The pieces have uses in the game. But this is not modelled so far in the database.)

A quest can have any number of pieces of loot. And, of course, each treasure type can be loot for any number of quests.

If the quest is successfully completed by its “team” — that is, by the avatars signed up for (*acting in*) the quest — then the quest's loot is given to those players. (The most famous treasure in the game is a golden lute. Lute. Loot. Get it? Snickers.)

Each piece of loot from the quest is *given* to just one of the players who participated in the quest (as decided by the game engine), if the quest is *successful*. We need to record who has been given what. *If* a quest was successfully completed, we record the *finish* time. (A quest is just for a given day, so we already know the day.)

In this project, you will work with an existing database, the *Raccoon Rhapsody Database* (RR-DB), a database that tracks participation in an online, multi-player game. Each will install his or her own *private copy* of the database on which to work. You are to implement a number of SQL queries over the RR-DB. The project is 10% of the total grade.

There are *ten* queries that you are to write. Each is worth *one* point, for a total of ten points. I am the manager and I provide you for each one with English specifications in the form of a *statement*, which you will turn into a *question* — an SQL query against the RR-DB, actually — like in the game show *Jeopardy*.

Each query is given a name for bookkeeping. A partial answer table for each is provided, along with how many tuples should appear in the answer table in total. This is so you can check whether you have likely implemented the SQL query correctly, by seeing whether you get the *same* sample tuples when run over the RR-DB database by PRISM's PostgreSQL server, **DB** (*version 11*).

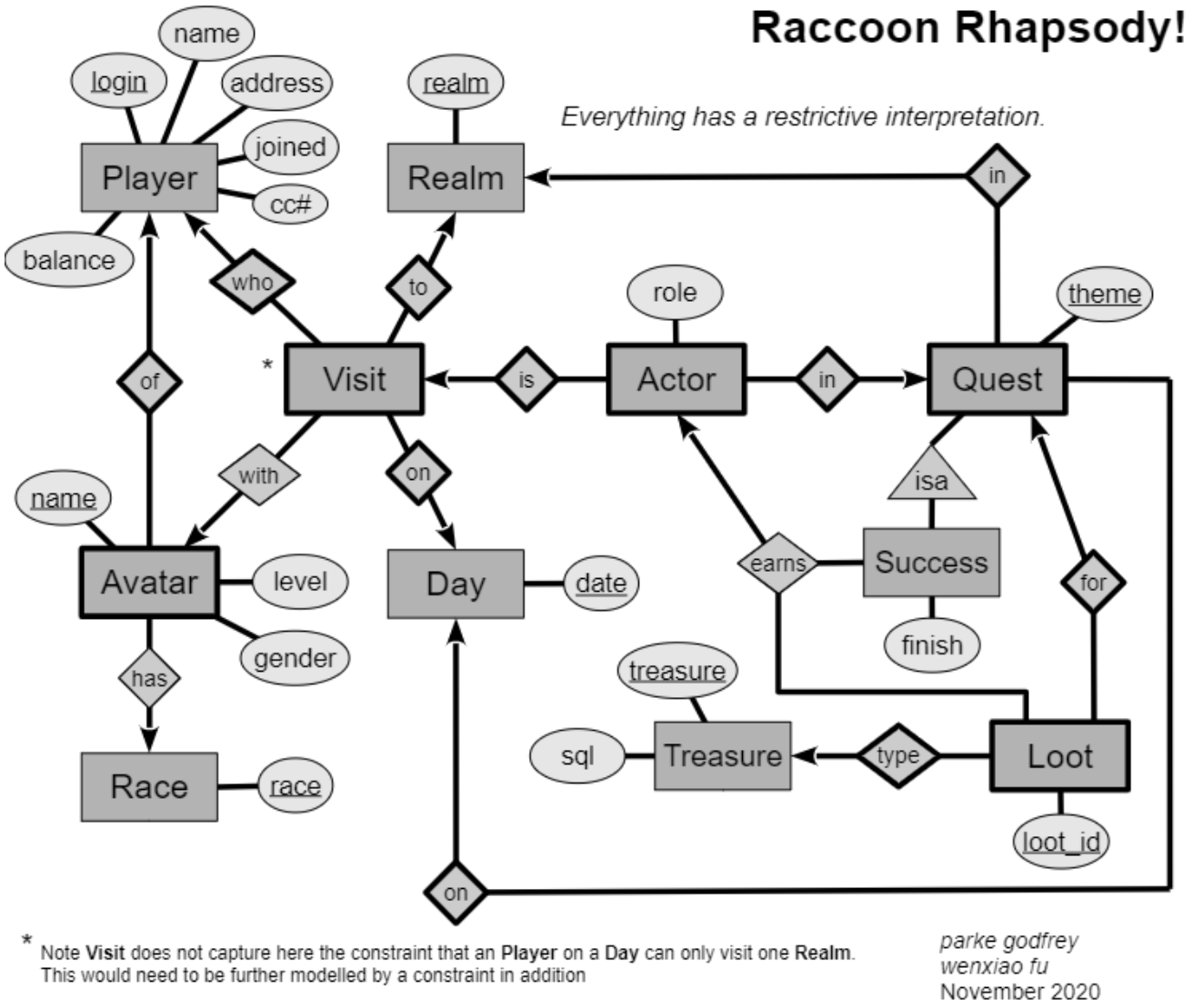
Note that, however, this is not *proof* that your query is entirely correct! A logically incorrect query with respect to the requirements would give the wrong answers on *some* instance of the database. And one could "cheat" by making a simple query on purpose that manages to print the same results on this instance of the database!) Grading per query is *all-or-nothing*. Either your query (always) produces the right results, or it does not. (We likely will test the queries for grading against a *different* instance of the database! That is, the RR-DB database but with different data.

The queries below are arranged very *roughly* in order of increasing difficulty. The ones at the beginning are relatively straightforward, while the ones towards the end are relatively difficult.

Some of the queries will be more involved to write than others. Do not get too discouraged if you are unable to do one right away. For some, you will need to look for features of SQL to help. For others, you will need to think carefully about the logic.

E/R Schema of Raccoon Rhapsody

Here is an E/R schema adaptation of the *description* above.



Relational Schema of Raccoon Rhapsody

We further adapt the E/R design to a *relational schema* and add data. Two scripts are provided for PostgreSQL:

- rrdb-create, and
- rrdb-drop.

The script `rrdb-create` will create the RR-DB relatiomal schema for you, *and* will also populate the tables with the mock data. The script *rrdb-drop* is provided for convenience; It will drop your copy of RR-DB from your schema space. If you mess things up, you can always drop RR-DB and then re-create it easily.

Read the schema definition in `rrdb-create` for RR-DB to understand it fully.

Write an *SQL* query for each of the following with respect to the RR-DB database.

1. myself

List each player whose `login` is part of his or her `name`; i.e., his or her `login` is a substring of his or her `name`. This should be case insensitive; e.g., “thom” is a substring of “Thomas Kane”.

schema: login, name, gender, address, joined
order by login (asc)

answer table: myself

2. golden

List each quest by `realm`, `day`, and `theme` which offered a prize (`treasure`) with “Gold” in the name which was rewarded to some player.

schema: realm, day, theme
order by day, realm, theme

answer table (first 12 records): golden

3. evening

List the quests by `theme`, `day`, and `realm` that were *not* completed *before* 8pm (on the *day* of the quest) with their `succeeded` time (which is *null* if it did not succeed).

schema: theme, day, realm, succeeded
order by theme, day, realm

answer table (first 12 records): evening

4. cheat

Report for each player by `login` and `name` who managed to participate in *more than one* quest on the same day, along with those quests by `day`, `realm`, and `theme`.

schema: login, name, day, realm, theme
order by login, name, day, realm, theme

answer table (first two records): cheat

5. bend

List each player by `login`, `name`, and `gender` who gender swapped at least once with their avatars, along with the count of how many avatars that he or she has (`avatars`).

schema: login, name, gender, avatars
order by login

answer table (first three records): bend

6. successful

Select the themes (`theme`) for which the *quests* were always *successful*, and report the number of successful quests (`quests`) for each such.

schema: theme, quests
order by theme

answer table (first three records): successful

7. frequency

Report the average number of days (as `frequency`) between visits to each given realm for each player. Also show the number of visits (`visits`) to that realm for the player. (Ignore a player in a realm if the player has never visited it or has only visited it once; the frequency is not defined in such cases.)

notes

Cast frequency with precision five and scale two.

schema: login, realm, visits, frequency
order by login, realm

answer table (first 12 records): frequency

8. race

Show each `realm` and `race` (of avatar) with the `gender` whose avatars of that race earned the most scrip (`sql`) collectively from loot rewarded in quests in that realm, along with the what that race and gender collectively earned in quests in the realm (`total`).

In case of ties for most in a region, list all that tied.

schema: realm, race, gender, total
order by realm, race, gender

answer table (first three records): race

9. companions

List each occurrence in which an avatar (by `login` as `companion1` and avatar's `name` as `fname`) whose participation in quests within a given realm has always been together with a second avatar (by `login` as `companion2` and `name` as `lname`) who has participated in exactly those same quests within the realm.

Since each pair of such companions would be shown twice — once with avatar X and avatar Y and once with avatar Y and avatar X — break the tie and show each such pair (per realm) just once; choose such that companion1 is before companion2 in dictionary order.

schema: companion1, fname, realm, companion2, lname
order by realm, companion1, fname, companion2, lname

answer table (first six records): companions

10. potential

Show for each avatar by `login`, avatar's `name`, and `race`, the scrip (`sql`) that the avatar would have earned (earned) if the avatar had been rewarded the prize (*loot*) of *highest* value (and *just* that prize, one piece of loot) for *each* quest in which the avatar participated that was successfully completed, and how many successful quests the avatar has participated in (`quests`).

schema: login, name, race, earned, quests
order by login, name

answer table (first 12 records): potential

Note that this query is asking to report for *each* `login`, (avatar) `name`. Report `0` for `earned` and `quests` for any such avatar who has not participated in any successful quests.

Queries on the York River Bookstore Schema

The York River Bookstore (YRB) schema script is for a small database that is similar to the example discussed in class. (The file `yrb-drop` is a simple file of SQL drop commands that clears out the YRB database.)

See the file `yrb-query` for examples of SQL queries over the YRB schema.

As with **Project #2**, write your queries to work with **PostgreSQL** *version 11*.

Due Date

Due by *11:59pm Tuesday 24 November 2020* by electronic submission.

Materials

For each query, make a *plain text file* with your SQL query text in it with the name that same as the query name above. E.g., your first file for Query #1 would be named “myself”, your second file for Query #2 would be named “golden”, and so forth.

Submission

Use the “ `submit` ” command on a PRISM machine to turn in your queries. Have each query in a plain ASCII (text) file, as above.

Then submit them on PRISM with the `submit` command.

```
% submit 3421A jeopardy myself golden evening cheat bend successful frequency race
companions potential
```

or

```
% submit 3421B jeopardy myself golden evening cheat bend successful frequency race
companions potential
```

depending on whether you are in section **A** or section **B**, respectively.

The *keyword* “jeopardy” in the above command names the project. The following ten words (*myself*, ...) are your query files.

Is the with clause *useful for these queries?*

Yes! The *with clause* is a useful way to organize your query into steps. For some queries, one use no need for this; but for others, it can make the query much easier to compose and read.

One may have more “tables” in the *with clause*, if need be:

```
with
    first (...) as
        (...),
    second (...) as
        (...),
    third (...) as
        (...)
select ...
    ...;
```

Note that you can use the (temporary) view *first* inside the definition for *second*, and the views *first* and *second* inside the definition for *third*, and so forth.

*What is a strategy for **Q9**, companions?*

“For all” cannot be done directly in SQL. And we are asking for pairs of avatars who have gone on *all* of the same *quests* within a given *realm* together.

A strategy might be to find *candidate* avatar pairs first. Say, find the number of quests in a realm each avatar has been on. Then clearly pairs of avatars could only have been on *all* of the same quests together within a region *if* they both have been on the same number of quests in the region. Of course, the “only if” is not *true*. Parke and Jeff maybe both went on

five quests in realm “Buffalo”, but Parke's five quests may not be the same ones as Jeff's. So Parke & Jeff would be a candidate pair, but we still have to check further whether this is an answer to the query.

That further check can be to see whether there is a quest in “Buffalo” that Parke went on but Jeff did not, or vice versa. If so, that candidate — Parke & Jeff — is eliminated; they have not gone on all the same quests. If not, that candidate — Parke & Jeff — is an answer.

*Could you say more on what **Q7**, frequency, is asking?*

First, only players who have been on more than one *quest* in a *realm*. For such a player & realm, we can determine the *number of days* between the earliest quest in that realm in which they participated and the last. Dividing that by the number of quests in that realm in which they participated *minus* one is the measure that we are calling *frequency*; e.g., “Parke quested in region ‘Buffalo’ on average once every thirteen days.”