CITS1402 Project

Gordon Royle and Michael Stewart

2021 Semester Two

So far in this unit, the labs have been primarily focussed on writing SQL *queries* thereby learning how the SQL "row-processing-machine" can be used to select, manipulate and summarise data contained in multiple relational tables.

This short project is going to focus on some of the other aspects of databases and database design.

A database designer builds the database schema and possibly enters the initial data, but over time the data evolves as rows are inserted, updated and deleted during the day-to-day use of the database. End-users will often make data entry errors or forget to complete a sequence of database actions, risking the integrity and internal consistency of the database. While the database designer cannot totally protect the database from future misuse, there are many ways in which decisions made by the database designer can make the database significantly more robust to data corruption caused by careless users.

This project explores some of the steps that a database designer can take to enhance the long-term integrity of the database.

The questions may require you to undertake your own research into how certain SQLite features are implemented. The official documentation is located at `https://www.sqlite.org/docs.html/index.html`, and there are numerous SQLite tutorial sites with examples.

Project Rules

For the duration of the project, different (stricter) rules apply for obtaining help from the facilitators and `help1402` for the duration of the project.

1. Absolutely no *"pre-marking"* requests

   Do not show your code to a facilitator and say "Is this right?"

   Firstly, this is not fair to the facilitator, who is there to provide *general assistance* about SQL and not to judge whether code meets the specifications.

   Secondly, from previous experience, such requests often degenerate into the situation where the facilitator "helps out" with the first line of code, then the student returns five minutes later and asks for help with the second line of code, and so on, until the final query is mostly written line-by-line by the facilitator and not the student.

   Facilitators are there to gently nudge you in the right direction, not by just "giving the answer" and supplying code that works, but by making general suggestions on SQL features, reminders about what concepts might be useful, and advice on how you might investigate and resolve problems yourself.

2. No *validation* requests for your submission

   Please do not ask the facilitators anything about the mechanics of making a valid submission such as file names, due dates etc. This is not their job and it leads to awkward situations where

a student submits something that is obviously incorrect, but then claims that "the facilitator said it was ok".

You are responsible for writing, testing, formatting and submitting your code correctly, and if you have any doubts about what is required, then please ask on `help1402`.

3. Avoid *low-quality* `help1402` posts

   Monitoring `help1402` is one of the most time-consuming activities in this unit. It is, however, one of the most effective ways of helping many students at a time, particularly when we answer a high-quality question seeking guidance on a difficult concept or a general principle or tricky syntax.

   Towards the project submission date however, there is often a reduction in high-quality questions, but a flood of low-quality questions. In order to maintain the current high "signal-to-noise" ratio, please ensure that:

   - Your question is actually new

     Don't ask a question that has already been answered in another thread. You can either monitor `help1402` daily so you always know what has been discussed, or use the search facility.

   - You actually need external help

     Quite a few posts have asked for confirmation that the output of a SQL query is "correct", even though it would be straightforward for the user to check this themselves.

     Given access to an actual database, you should normally be able to tell how many rows of output there should be by using SQLiteStudio to examine the data directly or by manually running a few simpler queries.

     So just make sure that you have made reasonable efforts to test your query yourself before posting to `help1402`

   - Your question is precise

     Please don't post *vague or overly general* requests for assistance such as: "I tried using `<random SQL>` but it didn't work. Any help".

     All coding starts by forming a logical plan for extracting the required information from the database. Of course you have to keep the general overall structure of an SQL query in mind in terms of the sorts of things that SQL can and cannot do, but try to get a clear idea of what you want to do before you start actually coding it.

     While forming the plan, you may notice that you need a table or a value that is not actually stored in the existing tables, but needs to be computed. This is when you think about how you can use subqueries to create the table or compute the value.

     When it is time to implement your plan in SQL, remember that very few people can just sit down and code an entire complicated SQL query from first line to last line, partly because the order in which the keywords occur is not the order in which the actual steps of the row-processing are conducted. So write and test small portions of the code separately and then put them together.

     Finally, remember that *you are in control* — you are the coder, and the computer is doing *exactly* what you tell it to do. If you accidentally tell it to do the wrong thing, then work out *why* it is doing the wrong thing (by mentally going through the process) and change it.

     While coding certainly requires experimentation and testing, it should be a systematic process. In other words, just randomly changing one SQL keyword to another or shuffling around the lines of code is not an effective method of coding.

     sd

   - Your question includes no (or minimal) actual code

> As usual, don't post actual code to `help1402`, instead giving just a verbal description or posting a redacted screenshot (i.e., with key parts blurred or otherwise obscured). So far this has been done well.

BRICKBUSTER **Videogame Rentals™**

BRICKBUSTER Videogame Rentals™ is a bricks-and-mortar videogame rental store. The owners of the store, Greg and Matt, set up shop a few years ago with the dream of providing gamers with an alternative to shelling out considerable amounts of money to buy games that they may only play for a few days.

Greg and Matt, who are somewhat familiar with but by no means experts in SQL, have implemented a SQLite database themselves that keeps track of BRICKBUSTER Videogame Rentals™ 's games, game licenses, rentals and customers. However, they have noticed some problems with their database schema. Some data is clearly incorrect, while the data in some tables is inconsistent with the data in others.

Greg and Matt employ you to examine the current schema and propose an enhanced database design according to their requirements. The current database has four tables, namely `GameTitle`, `GameLicense`, `gameRental`, and `Gamer`. The structure of this database is outlined below.

The table `GameTitle` has general data for games on a particular platform

This table stores data about *game titles on a particular gaming platform* (not about the hard copies of those games on the Brickbuster shelves).

```
CREATE TABLE GameTitle (
title TEXT,
release_year INTEGER,
platform TEXT,
price REAL
);
```

Each row captures the title (i.e. name) of the game, the year the game was released, the platform on which it is played (e.g. PlayStation 5, PC, Xbox, Nintendo Switch) and the recommended retail price of the game (in Australian dollars) were you to purchase it outright. A typical row in this table would be something like:

```
('Stardew Valley', 2016, 'PC', 16.99)
```

which would indicate that the game titled "Stardew Valley" was released on PC in 2016 and has a recommended retail price of $16.99 AUD.

Greg and Matt indicate that the *combination* of `title`, `release_year` and `platform` uniquely determines a `GameTitle`, and that the price depends only on these three columns.

Note that BRICKBUSTER Videogame Rentals™ may list games in `GameTitle` even if they do not stock any actual hard copies (i.e. a `GameLicense`) of those games.

The table `GameLicense` has data for actual games

Each `GameLicense` represents a single hard copy of a game that BRICKBUSTER Videogame Rentals™ has in its inventory, and that is potentially available for a gamer to rent.

```
CREATE TABLE GameLicense (
title TEXT,
release_year INTEGER,
platform TEXT,
license_id TEXT
);
```

Every individual `GameLicense` has its own unique identifier called `license_id`. This 5-character string is comprised of four digits followed by a "check digit" (which is explained further in Question 2c).

BRICKBUSTER Videogame Rentals™ may have multiple copies for popular games, and so there may be multiple `GameLicense` records referring to the same `GameTitle`.

Rows from the `GameLicense` table might look similar to the following:

```
('Stardew Valley', 2016, 'PC', 12137)
('Stardew Valley', 2016, 'PC', 12238)
('Stardew Valley', 2016, 'Nintendo Switch', 12339)
```

... which indicates that there are 3 copies of the 2016 game "Stardew Valley" in BRICKBUSTER Videogame Rentals™ (two of which are for PC, and one of which is for the Nintendo Switch). Those three copies have the license id of `12137`, `12238` and `12339` respectively.

The table `gameRental` has data for each rental

Each row of this table records the details for a single rental to a customer of a particular `GameLicense`.

```
CREATE TABLE gameRental (
gamer_id INTEGER,
license_id TEXT,
date_out TEXT,
date_back TEXT,
rental_cost REAL
);
```

Each rental involves a particular `Gamer`, identified by a unique `gamer_id` renting a specific `GameLicense` (identified by the `license_id`).

When a gamer rents a license, a new tuple is entered into the `gameRental` table by the Brickbuster clerk. The tuple contains the ids of the gamer and game license, and the field `date_out` is set to the current date and time, while both `date_back` and `rental_cost` are set to `NULL`. Although the columns are called `date_out` and `date_back` they actually store what are called *datetime* values, which represent the exact date and time down to the second as a text string in the format `YYYY-MM-YY HH:MM:SS`.

When the game is returned some time later, the clerk updates the row for that rental by setting the `date_back` to the new current date and time, and calculates the value for `rental_cost`. Rentals are charged at a flat $3 fee, plus a percentage (5%) of the recommended retail price of that particular game title, multiplied by the number of days in the rental (including fractions of the day). For example, if the game "Stardew Valley" was rented for 9.25 days (i.e. 9 days and 6 hours), the total cost would be:

$3 + 16.99 \times 0.05 \times 9.25 = \$10.857875$

The `rental_cost` column is then updated according to the calculated value as per the above.

The table `Gamer` has data for gamers (i.e. customers)

The `Gamer` table is responsible for storing the data about an individual customer of BRICKBUSTER Videogame Rentals™ . Each gamer is assigned a unique numeric gamer id. The table stores their first name, last name and email address.

```
CREATE TABLE Gamer (
gamer_id INTEGER,
first_name TEXT,
last_name TEXT,
email TEXT
);
```

A typical row in this table would be something like

```
(11, 'Bobby', 'Simpson', 'bobby@mail.com')
```

indicating that the `Gamer` with `gamer_id=11` is called Bobby Simpson, whose email address is `bobby@mail.com`.

The tasks

As a database developer, you have been called in to improve the integrity of the database. You will not be changing any of the column names or data types of the tables, but just *adding* database features to improve the integrity and usability of the database.

You are asked to submit four files

```
ERD.png
DB.sql
DBTrigger.sql
DBView.sql
```

according to the following specifications:

1. An entity-relationship diagram `Submit to cssubmit: ERD.png` (5 marks)

   The first task is to get a visual representation of the existing database. This requires you to "reverse engineer" the actual database to produce a corresponding entity-relationship diagram.

   Do not invent additional entities or attributes in the ERD, but also remember that—in certain situations—not *all* of the relations in the ERD will be represented as tables in the database. In this situation, you *will* need to name a relationship in the ERD that is not present as a table in the relational schema. Video 31 should clarify what is required.

   You *must* use `ERDPlus.com` to prepare your ERD and then use the "Export Image" selection from the "Menu" button at the top-left of a diagram to save it to a PNG file. The file will be saved under some generic name like `image.png`, but you should rename it to `ERD.png` and submit it as the first file to `cssubmit`.

Include all of the relevant cardinality and participation constraints of the *improved database* according to the specifications above, using your real-world knowledge of how game platforms work for anything not explicitly specified.

Once again, *do not submit anything* that is produced by a different ER diagramming tool, or produced as a figure in Microsoft Word, or drawn in a drawing/painting program, or is hand-drawn and photographed/scanned.

(The reason for this is that there are literally hundreds of diagramming tools / conventions, and it would be impossible for the markers to know them all.)

To keep things simple, for both you and the project markers, please make sure that your ERD includes *only* entities, relationships, attributes (with underlines for key fields) and cardinality and participation constraints.

In particular,

- Only use *regular entities*, and do not use weak, associative or supertype entities
- Only use *attributes* or *unique attributes* and do not use multivalued, optional, composite or derived attributes. (If you need a composite key, then just underline all of the attributes involved.)
- You may use any of the cardinality/participation constraints available in ERDPlus.

2. A database schema `Submit to cssubmit: DB.sql` (2 + 2 + 2 = 6 marks as specified below)

   You should prepare a file called `DB.sql` that creates an improved database. It should contain code to create the four tables `GameTitle`, `GameLicense`, `gameRental` and `Gamer`, with *exactly the same* attributes and data types as described above, but with additional features (as described below).

   You should only include the DDL statements (the statements that create the tables), so make sure you *do not include* any statements to insert data into the tables.

> WARNING: Your file `DB.sql` must create the four tables with *exactly the same* columns as specified above. Your code will be tested by using your `DB.sql` file to create a database, into which sample data will be inserted, and various
>
> If you change the name of the tables, or of the names of the columns, say, `release_year` to `releaseYear` then the testing code *will not work*.

   Of course, you should *test* your improved database by populating it your own synthetic (made-up) sample data, and running various insert, update and delete commands, but do not include this in your submission.

   The additional features you should incorporate into `DB.sql` are:

   (a) Key columns and uniqueness constraints (2 marks)

   The tables written by Greg and Matt contain no information about keys, so nothing prevents the accidental insertion of inconsistent data, such as:

   - Two rows in `GameTitle` with the same combination of `title`, `release_year` and `platform`.
   - Two rows in `GameLicense` with the same `license_id`.
   - Two rows in `Gamer` with the same `gamer_id`.

   Give improved `CREATE TABLE` statements for the tables `GameTitle`, `GameLicense` and `Gamer`, ensuring that the uniqueness constraints specified above are enforced by the database.

   (b) Referential integrity (2 marks)

   One problem for Greg and Matt is that the desk clerk often enters a new tuple into `gameRental` in a hurry, and mistypes either the `license_id` or the `gamer_id`. If the `license_id` is incorrect, then it is impossible to calculate the cost of a rental, and if the `gamer_id` is incorrect, then it is impossible to know which customer to charge, so this is a major problem.

Give an improved `CREATE TABLE gameRental` statement to incorporate *referential integrity constraints* ensuring that the `license_id` refers to an actual row in `GameLicense` and `gamer_id` refers to an actual row in `Gamer`.

Greg and Matt tell you that a gamer is never deleted from the table `Gamer`, but occasionally a `gamer_id` might change (via an `UPDATE` statement). If this happens, then the tuples in the `gameRental` table for this gamer's previous rentals should automatically be altered to reflect this change.

Greg and Matt tell you that the `license_id` for a `GameLicense` can never change, and that a GameLicense is never deleted from the database, therefore you need not include code that will prevent or deal with this situation.

You also do need to include referential integrity in any table other than `gameRental` (it won't hurt if you do, but it will be ignored).

(c) Data entry validation (2 marks)

Each of the games at BRICKBUSTER Videogame Rentals™ have a sticker on the back of the box that displays the `license_id` of that particular game. A `license_id` is a 5-digit string of digits, where the last digit is a"check digit" based on the previous digits. Typing any sequence of numbers into a computer is error-prone, and often a single digit will be mistyped or two digits transposed. The point of the check digit is to ensure that these common typing errors will result in a string with the wrong check digit, and hence an invalid `license_id`.

Greg and Matt would therefore like you to find a way to automatically validate a `license_id` at the moment the tuple is entered into the database, so that any such errors can be immediately detected and prevented.

A string of characters

$$d_1 d_2 d_3 d_4 d_5$$

is a valid `license_id` if

- The first four characters are digits, i.e. 0–9.
- The fifth character, i.e., the *check digit*, is the *last digit* of the value

$$(1 \times d_1 + 3 \times d_2 + 1 \times d_3 + 3 \times d_4) \tag{1}$$

For example, given that the first four characters of a `license_id` are `1234`, then the check digit is the last digit of

$$(1 \times 1 + 3 \times 2 + 1 \times 3 + 3 \times 4) = 22 \tag{2}$$

resulting in a check digit of $d_5 = 2$, and hence the `license_id` of `12342` is valid, whereas `12341`, `12343`, `12347` are not.

(This approach to computing a check digit is very similar to the calculation of the check digit in the ISBN13 codes for books.)

Checking validity of input in this fashion is accomplished by using SQLite *check constraints*. A check constraint is a boolean expression associated with a single column using the keyword `CHECK`. Every time the value in that column is altered (or inserted) the system will *check* that the boolean expression is still true with the new value.

For example, consider a table `BankAccount` for an account where the balance is never allowed to drop below 0. This could be defined with

```
CREATE TABLE BankAccount(
  accountNumber INTEGER,
  accountBalance REAL CHECK(accountBalance >= 0));
```

The system will then *check the condition* when any `UPDATE` statement is attempted, and prohibit the operation if the changed value violates the condition.

Add a `CHECK` constraint to the `GameLicense` table to ensure that the `license_id` always meets the basic requirements above. You may need to look up the documentation for `CHECK` on `sqlite.org` to double-check the exact syntax.

Although the `license_id` is a string that contains only the digits `0` to `9`, it is not itself a number, but rather it is entered as a *string* in a column of type `TEXT`. So you should use string functions such `substr` to extract characters from the string, combined with operators such as `IN` and/or `BETWEEN` to make sure they are allowable characters. You will need to cast the digits to integers to be able to perform the addition and multiplication — so it is recommended you take a look at the `CAST` operator in the SQLite documentation: `https://sqlite.org/lang_expr.html`.

3. A trigger to improve data consistency `Submit to cssubmit: DBTrigger.sql` (2 marks)

   Greg and Matt constantly have problems keeping the `rental_cost` field in the `gameRental` table consistent with the actual amount a gamer owes upon returning a game.

   As mentioned previously, when the gamer rents a `GameLicense`, the clerk enters a new tuple in the `gameRental` table. This tuple contains the gamer id, the license id, and the current date and time in the `date_out` column, while the other fields are set to `NULL`. When the gamer returns the `GameLicense`, the clerk updates the tuple by setting the `date_back` column to the date and time.

   At this point, the clerk is *also* meant to update the `rental_cost` field in the `gameRental` table to indicate that the gamer who returned the game now owes a certain amount of money. As previously mentioned, this amount owed is based on a flat $3 fee, plus a percentage (5%) of the recommended retail price of the game title, multiplied by the number of days the gamer rented it for (including fractions of the day). As an equation, it looks as follows:

   $$\texttt{rental\_cost} = 3 + \texttt{price} \times 0.05 \times \texttt{days\_rented} \tag{3}$$

   where `price` is the recommended retail price of the game (the `price` column in the `GameTitle` table), and `days_rented` is the total number of days that the game was rented for (as a real number to accommodate fractions of a day).

   However, relying on the desk clerk to calculate this value correctly when busy helping customers is not realistic. Performing this calculation manually is prone to user error and Greg and Matt wonder if there is a way to do this automatically somehow.

   You realise that this is an ideal situation for the use of *triggers*.

   Write the code for a trigger on the table `gameRental` as follows:

   - When the desk clerk *updates* a tuple in `gameRental` (because the gamer has returned the game) and he or she updates the `date_back` field, a trigger should intercept this operation and update the `rental_cost` column according to the calculation in equation 3 above.

   This ensures that the desk clerk cannot accidentally miscalculate the total cost of the rental as the calculation will happen automatically.

   Datetimes are given in the `YYYY-MM-DD HH:MM:SS` string format used by SQLite, and you may need to look at the date and time functions supplied by SQLite, which are documented in `https://www.sqlite.org/lang_datefunc.html`. I'd look closely at `julianday()` to start with.

4. A view to improve usability `Submit to cssubmit: DBView.sql` (2 marks)

   Gamers often ask Brickbuster for a summary of the games they have rented, along with the number of rentals and the total amount of money that they have spent on each game. The necessary SQL command to extract this information in the right format is a little complicated, and too easy for Greg and Matt to get wrong.

   Write SQLite code that *defines a view* named `GamerRentalSummary` that should behave as though it were a table with the following schema:

```
GamerRentalSummary (
  gamer_id INTEGER,
  gamer_name TEXT,
  game_title TEXT,
```

```
  game_release_year INTEGER,
  game_platform TEXT,
  times_rented INTEGER,
  total_rental_cost REAL
);
```

Note that `gamer_name` should be the full name of the gamer, i.e. their first name and last name separated by a space. An example row from this view might look as follows:

`(11, 'Bobby Simpson', 'Stardew Valley', 2016, 'PC', 3, 24.50)`

... indicating that Bobby Simpson (whose `gamer_id=11`) has rented "Stardew Valley" for the PC three times and has spent a total of $24.50 across these three rentals.

Write the code to *create the view* `GamerRentalSummary` (note it is **Gamer**RentalSummary and not **game**RentalSummary) with the specifications as above.

Remember that you should not include *incomplete game rentals* in this data; these can be identified by the fact that the field `gameRental.date_back` is `NULL`.

Brickbuster is a portmanteau of Blockbuster, a now defunct video rental store, and Breakout, a popular arcade game from the 1970's that involved breaking a set of bricks by bouncing a ball into them with a paddle.