CITS1402 Project

Gordon Royle

2020 Semester Two

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

This project, really a mini-project, is going to focus on some of the other aspects of databases.

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. An important role of the database designer is to make the database resistant 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

   Before the mid-semester test, there was a huge spike of activity on `help1402`. While I encourage thoughtful questions and try to answer them promptly, the sheer volume of questions almost overwhelmed me.

   Even without `help1402`, this unit is already consuming far more time than I am meant to spend on it, so I have to cut back. A lot of my time earlier in semester was spent dealing with *repeated* or *low-quality* questions, so I'd like to eliminate (or at least reduce) these. Ideally `help1402` should be a lower-volume but higher-quality forum.

   So before you post, 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 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. For example, if counting parking tickets for black and white cars has to be done for every state, then first write the query for just one state and one of black options, and then gradually extend it.

     Finally, remember that *you are in control* — you are the coder and the machine 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.

   - 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).

(Actually, almost everyone is *already* doing the right thing with obscuring their posted code, so this is just a reminder to keep doing it properly rather than a change in policy.)

Dodgey Brothers Auto Rentals

Wayne and Arthur Dodgey run a car rental business called Dodgey Brothers Auto Rentals and want a database to keep a record of their cars, customers and rentals.

They have implemented a SQLite database themselves that is adequate, but after a few months use they have noticed some problems. Some data is clearly incorrect, while the data in some tables is inconsistent with the data in others.

You are given the schema of the current database and discuss the requirements with Wayne and Arthur. The database has four tables, namely `Car`, `Vehicle`, `rental` and `Customer` which have the following structure:

The table `Car` has data for types of car

The table stores data about *types of car* (not individual vehicles).

```
CREATE TABLE Car (carMake TEXT,
     carModel TEXT,
     carYear INTEGER,
     dailyCost INTEGER,
     kmCost REAL )
```

A typical row in this table would be something like:

```
('Hyundai','i30',2020,30,0.10)
```

The first three fields describe a type of car, in this case a 2020 Hyundai i30, and the last two fields indicate that Dodgey Brothers Auto Rentals rents a car of this type for $30 per day plus $0.10 per km.

Wayne and Arthur indicate that the *combination* of make, model and year uniquely determines a car type, and that the daily and per-km costs depend only on this car type.

If a customer rents a 2020 Hyundai i30 for 3 days and drives 200km, then the cost of this rental will be

$$3 \times 30 + 200 \times 0.10 = 110.$$

Wayne and Arthur may not have actual vehicles of every type listed in `Car`.

The table `Vehicle` has data for actual vehicles

This table stores data about the individual vehicles in the Dodgey Brothers Auto Rentals fleet.

```
CREATE TABLE Vehicle (carMake TEXT,
     carModel TEXT,
     carYear INTEGER,
     VIN TEXT,
     odometer INTEGER)
```

A typical row in this table would be something like

```
('Hyundai','i30',2020,'WDCGG5GB8AF429863', 15199)
```

The fields `carMake`, `carMode` and `carYear` have the same meaning as in `Car` while `VIN` is the car's Vehicle Identification Number which is a unique code stamped onto a metal plate and riveted to the car's frame by its manufacturer. The code is a 17-digit string containing letters and numbers in a format similar to the example above. Dodgey Brothers Auto Rentals may have several cars of the same type, but it is impossible for two different vehicles to have the same VIN.

The `odometer` field lists the number of kilometres on this vehicle's odometer, so this particular vehicle has been driven for a total of 15199 kilometres since it was new.

The table `rental` has data for each rental

This records the details for each individual rental of a vehicle.

```
CREATE TABLE rental (customerId INTEGER,
    VIN  TEXT,
    odo_out INTEGER,
    odo_back INTEGER,
    date_out TEXT,
    date_back TEXT)
```

A rental is made by a customer, identified by a unique customer ID. The customer rents a specific vehicle (identified by the VIN).

A new tuple is entered into the table `rental` at the time that the customer picks up the vehicle. The fields `odo_out` and `date_out` record the odometer reading on the vehicle, and the date. The fields `odo_back` and `date_back` are set to `NULL` (because these values will not be known until the car is returned.)

When the car is returned, an `UPDATE` statement completes the tuple by setting `odo_back` and `date_back` to the actual odometer reading on the car and the actual date that the car is returned.

This rental is now completed and the rental cost can be calculated from the costs for that type of car, the number of days in the rental (including *both* the start day and finish day of the rental), and number of kilometres travelled (the value `odo_back - odo_out`).

Dates are given in the `YYYY-MM-DD` 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.

At this stage, the rental desk clerk is meant to update the `odometer` field in the tuple in the `Vehicle` table for this particular car, but sometimes the clerk is busy, puts this off until later, and then forgets to do it.

A vehicle can be entered into the database before it is rented to anyone, and obviously over its lifetime a vehicle will be involved in many different rentals.

You may assume without testing that the information about rentals is sensible. The test data will not create rentals that end before they start, or that overlap with other rentals for the same car, and so you *do not need to check* this.

The table `Customer` has data for each customer

This table records the details for each Dodgey Brothers Auto Rentals customer.

```
CREATE TABLE Customer (
  id INTEGER,
  name TEXT,
  email TEXT);
```

Each customer has a unique id, and Dodgey Brothers Auto Rentals only keeps the name and email address of their customers. An account can be created for a customer before they rent a car.

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 `cssubmit ERD.png` (5 marks)

   The first task is to get a visual representation of the database. This requires you to "reverse engineer" the actual database to produce the 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 car rentals 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.)

2. A database schema `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 `Car`, `Vehicle`, `rental` and `Customer`, 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, views and triggers) but *do not include* any statements to insert data into the tables.

   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 (2 marks)

   The tables written by Wayne and Arthur Dodgey contain no information about keys, so nothing prevents the accidental insertion of inconsistent data (for example, two different vehicles with the same VIN).

   Give improved `CREATE TABLE` statements for the tables `Car`, `Vehicle` and `Customer`, ensuring that the uniqueness constraints specified above are enforced by the database.

   (b) Referential integrity (2 marks)

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

   Give an improved `CREATE TABLE rental` statement to incorporate *referential integrity constraints* ensuring that the `VIN` and `customerId` refer to actual vehicles and customers in the `Vehicle` and `Customer` tables.

   Wayne and Arthur tell you that a customer is never deleted from the table, but occasionally a `customerId` might change (via an `UPDATE` statement). If this happens, then the tuples in the `rental` table for this customer's previous rentals should automatically be altered to reflect this change.

   For vehicles, Wayne and Arthur tell you that the `VIN` for a vehicle can never change, and a vehicle is never deleted from the database.

   (c) Data entry validation (2 marks)

   A vehicle's `VIN` is very important for any and all paperwork, such as lease agreements, insurance details, servicing schedule etc.

   However it is easy to mistype a long sequence of characters, and so we'd like to add some validation to ensure that anything entered into this field at least has the right format to be a VIN.

   - A VIN is a string of exactly 17 characters
   - Each character in a VIN is a *digit* or an *uppercase letter*
   - A VIN can contain any of the digits `0` to `9`
   - A VIN can contain any uppercase letter except `I`, `O` and `Q`
   - The 9th character of a VIN is either a digit from `0` to `9` or the letter `X`.

   (These are all true facts about a VIN, but in real VINs the 9th character acts a check digit and must satisfy an equation involving the other 16 characters.)

   The code needed to check a 17-digit VIN can be very cumbersome. So instead, we'll use a *simplifed version* of a VIN for this project, which is defined as follows:

   - A VIN is a string of exactly 5 characters
   - Each character in a VIN is a *digit* or an *uppercase letter*
   - A VIN can contain any of the digits `0` to `9`

- A VIN can contain any uppercase letter except `I`, `O` and `Q`
- The 3rd character of a VIN is either a digit from `0` to `9` or the letter `X`.

SQLite implements SQL *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 table `Vehicle` to ensure that the `VIN` 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.

This can be done in a naive way using string functions such as `substr` to extract characters from the string that is meant to be a VIN, combined with operators such as `IN` and/or `BETWEEN` to make sure they are allowable characters. Or it can be done in a more sophisticated way using operators such as `GLOB`. If you choose to investigate `GLOB` then you will need to carefully read the documentation and understand (or teach yourself) the basic principles underlying pattern matching, UNIX wildcards and regular expressions). This is only recommended for confident students.

3. Triggers to improve data consistency `cssubmit DBTrigger.sql` (2 marks)

Wayne and Arthur constantly have problems keeping the `odometer` fields in `Vehicle` and `rental` consistent.

As mentioned previously, when the customer rents a vehicle, a tuple is created in the `rental` table. At this point, the clerk checks the actual vehicle's odometer and enters this value into `odo_out`. When the customer returns the vehicle, the clerk again checks the vehicle's odometer, and enters this value into the `odo_back` field for this rental.

At this point, the clerk is *also* meant to update the `odometer` field in the `Vehicle` table, so that both `Vehicle.odometer` and `rental.odo_back` have the same value.

However, relying on the desk clerk to transfer values correctly when busy helping customers is not realistic. You advise Wayne and Arthur that having the same data stored in two different places is poor relational database design. Wayne and Arthur say that they are unwilling to change the schema because too many other systems rely on it, and ask if you can work around this design flaw some other way.

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

Write the code for two triggers on the table `rental` that maintains consistency between the two `odometer` fields in the following manner:

- When the desk clerk *inserts* a new tuple into `rental`, he or she enters the actual values for the `VIN`, `customerId` and `date_out`, but enters `NULL` for the other three values.

  A trigger should intercept this operation, look up the `odometer` reading for this car in the `Vehicle` table, and enter this value into the `odo_out` column for the newly-created tuple in `rental`.
- When the desk clerk *updates* a tuple in `rental` (because the customer has returned the car) he or she updates the `date_back` and `odo_back` fields with the current date and the actual reading on the vehicle's odometer.

  A second trigger should intercept this operation and update the correct row of `Vehicle` with the new odometer reading.

This ensures that the desk clerk cannot accidentally enter an incorrect `odo_out` value at the start of the rental, and cannot forget to update the odometer reading in `Vehicle` at the end of the rental.

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

   For tax purposes, customers often want a list of all of their rentals together with the cost of each rental. The necessary SQL command to extract this information in the right format is a little complicated and too easy for Wayne and Arthur to get wrong.

   Write the SQL code that defines a *view* name `CustomerSummary` that should behave as though it were a table with each row containing just the essential information about a *completed rental*.

   So the view should have the following schema:

```
CustomerSummary (
  customerId INTEGER,
  rental_date_out TEXT,
  rental_date_back TEXT,
  rental_cost REAL);
```

   Write the code to *create the view* `CustomerSummary` with the specifications as above.

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

Wayne and Arthur Dodgey, also known as "The Dodgey Brothers" were characters on the Australian TV Comedy Show "Australia You're Standing In It" from the early 1990s.