CITS1402 Project — Draft Only

Gordon Royle

2023 Semester Two

Deadlines

Regular Deadline: 5pm Sunday 15th October (the end of Week 11)
Hard Deadline: 5pm Sunday 22nd October (the end of Week 12)

Submissions will *not be accepted* after the hard deadline, and submissions made between the regular and hard deadlines will be treated as follows:

- The default action is that a late penalty will be applied according to the UWA policy, which is a 5% deduction per day or part of day for at most 7 days. Submissions more than 7 days late are not accepted. In this case, `cssubmit` will indicate the penalty that will be imposed based on your final submission date.
- If you have a UAAP that entitles you to an automatic extension of X days, then any late penalties will only start accruing after those days have elapsed. However `cssubmit` does not know about UAAPs, so it will still indicate the default late-penalty, but rest assured that I have everyone's details and I will manually override the late penalty.

  However, the *hard deadline* (Sunday 22 October) still applies for *all students*, so if you cannot make that deadline then you must apply for Special Consideration. In this case, the marks will be reallocated to the other assessments.
- In cases of sickness or misadventure, you should contact your Student Advising Office and apply for Special Consideration. Do not send me medical certificates or long stories about car crashes etc because I cannot evaluate them and I am not allowed to influence the granting or otherwise of Special Consideration. Depending on how serious the condition / accident is, you may be granted an extension for up to 7 days for minor things, or the marks may be reallocated to the other assessments if the condition / accident is so severe that you cannot submit by the hard deadline.

Remember that you can submit any number of files at any time, and replace them as often as you need. Submit each file *as you complete it* so that `cssubmit` always has a record of your completed work. If you wait until the last minute before submitting your files, then there is always a risk that something (laptop failure, Internet down, car crash on way to Uni etc.) will prevent you from submitting them by the deadline.

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.

A database designer builds the database schema and possibly enters the initial data, but over time the database 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 look up the names of certain SQLite functions or learn how certain operations are implemented. The official documentation is located at

`https://www.sqlite.org/docs.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 "giving the answer" or debugging your own supplying code, 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. Use `help1402` effectively

   The number of questions on `help1402` has been fairly low during the first half of semester this year, but usually ramps up considerably during the project.

   To avoid overloading us, it helps if you take a couple of extra minutes before posting to run through the following checklist:

   - Is this showing too much code?

     You may post tiny snippets of code showing what is causing an error, but do not post anything longer.
   - Is the question new?

     Try not to ask a question that has already been answered very recently in another thread. Even if you do not read `help1402` regularly, just glance down at the most recent 5-10 threads and see if the answer is already there.

- Do I actually need help? . Even though the error messages from SQLite (and other versions of SQL) are often cryptic, they do usually explain the root cause of the problem. With this information and Google, it is often possible to resolve the problem much more quickly than waiting for an answer.
- Is the question 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".
  Asking a precise question often clarifies in your own mind exactly what the problem is, and often you will end up resolving it while you are actually typing the question.
- Does the question include 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.)

KwikiPhone Rentals

KwikiPhone Rentals is a mobile phone rental business that rents iPhones, usually for a few weeks or months, to both individual and business clients.

They have a simple SQLite database that they use to keep track of the fleet of phones they own, their clients and the rentals, but they have noticed that it has some problems. Some of the data is clearly incorrect, while the data in some tables is inconsistent with the data in others. They are consulting you to see if you have some recommendations for improving the long-term integrity of the database.

The existing database has four tables, namely `PhoneModel`, `Phone`, `rental` and `Customer`, which have the following structure.

The table `Phone` contains data about a physical phone

The table `Phone` has data for the *individual phones* owned by KwikiPhone Rentals , and it has the following schema:

```
CREATE TABLE Phone (
  modelNumber TEXT,
  modelName TEXT,
  IMEI TEXT);
```

A typical row in this table would be something like:

```
('MPXL3LL/A', 'iPhone 14', '526457946034482')
```

The first field (`modelNumber`) is the code that Apple uses to distinguish between all the *variants* of their phones, such as their different colours and different storage sizes. For example, the code `'MPXL3LL/A'` refers to a blue iPhone 14 with 512Gb storage, while `MQ063LL/A` refers to a gold iPhone 14 Pro with 128Gb storage. The model name is the commonly used name for that particular type of phone.

The IMEI (International Mobile Equipment Identity) is the *unique identifier* for this specific phone. By

maintaining lists of the IMEIs of lost or stolen phones, network operators can prevent unauthorised use of the phone. The IMEI is a 15-digit number, where the last digit is a *check digit* that makes it easier to detect if a number is typed incorrectly.

KwikiPhone Rentals may have many phones with the same `modelName` and `modelNumber` but the IMEI is different for each and every phone.

The table `PhoneModel` describes each different model of phone

The table `PhoneModel` contains the information about that particular *model* of iPhone including its name, how much storage it has (in Gb) and the rate that KwikiPhone Rentals charges for that particular model.

```
CREATE TABLE PhoneModel (
  modelNumber TEXT,
  modelName TEXT,
  storage INTEGER,
  colour TEXT,
  baseCost REAL,
  dailyCost REAL
);
```

Each model is uniquely identified by its `modelNumber` and the rest of the columns contain information such as the common name for the phone (e.g., `'iPhone 14 Pro'`), its colour and storage size. The amounts that KwikiPhone Rentals charge for rentals depend on the phone model (not the individual phone) and so they are stored in this table.

The table `rentalContract` has data for each rental

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

```
CREATE TABLE rentalContract (
    customerId INTEGER,
    IMEI  TEXT,
    dateOut TEXT,
    dateBack TEXT,
    rentalCost REAL);
```

A new tuple is entered into the table `rental` at the time that the customer picks up the phone. The customer and phone are identified by `customerId` and `IMEI` respectively, `dateOut` records the date on which the rental starts and the remaining two fields are set to `NULL`, because these values are not yet known.

For example

```
INSERT INTO rentalContract VALUES
  (12, '526457946034482', '2023-07-10', NULL, NULL);
```

When the phone is returned, the clerk issues an `UPDATE` statement changing `dateBack` to the date that the phone is returned, using a statement such as:

```
UPDATE rentalContract
```

```
SET dateBack = '2023-09-31'
WHERE IMEI = '526457946034482'
AND dateBack IS NULL;
```

The rental is now completed and the rental cost can be calculated from the length of the rental and the costs for that particular phone model. At this stage, the clerk is meant to calculate the total cost of this rental and update the `rentalCost` column of this row with the correct value, but this does not always happen.

You may assume (without checking the input) that all the information about rentals is sensible. In particular, the testing process will not attempt to create rentals that end before they start, or that overlap with other rentals for the same phone, and so you *do not need to check* for this. Similarly, when the phone is returned and the clerk updates the relevant row, you may assume that the value of `dateBack` that is entered will be sensible, so you do not need to verify this.

The table `Customer` has data for each customer

This table records the details for each KwikiPhone Rentals customer.

```
CREATE TABLE Customer (
  customerId INTEGER,
  customerName TEXT,
  customerEmail TEXT);
```

Each customer has a unique id, and KwikiPhone Rentals only keeps the name and email address of their customers.

The tasks

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

You should submit the following three files to `cssubmit`.

```
createTables.sql
createTrigger.sql
createView.sql
```

according to the following specifications:

1. `cssubmit createTables.sql` DDL statements (2 + 3 + 4 = 9 marks as specified below)

   You should prepare a file called `createTables.sql` that contains four separate DDL statements to create the four tables `Phone`, `PhoneModel`, `rentalContract` and `Customer`. These tables should have *exactly the same* column names and data types as described above, but with *additional features* as described below.

   To avoid errors in typing the names of the tables or attributes, you should simply copy the supplied file `createTables.sql`, add the extra features, and then upload the revised file.

You should only include the DDL statements (the statements that *create* the tables) but not populate the tables with data. You should test your code with some sample data while you are writing the project, but make sure that the files you submit *do not include* any statements that insert data into the tables.

My testing code will first run your `createTables.sql` file to create empty tables and then enter my own testing data to make sure that the database behaves according to the specification.

The additional features that you should incorporate into the tables are as follows:

(a) Primary Keys (2 marks)

Add suitable `PRIMARY KEY` declarations to all of the tables to enforce the constraints given in the project description explaining which columns (or combinations of columns) uniquely identify a row of each table. These prevent the accidental creation of two rows referring to the same entity. For example, a `Customer` is uniquely identified by the `customerId` and so the system should prevent the creation of two rows with the same `customerId`.

All of the tables should have suitable `PRIMARY KEY` definitions.

(b) Referential integrity (3 marks)

One problem is that the desk clerk often enters a new tuple into `rentalContract` in a hurry, and mistypes either the `IMEI` or the `customerId`. If the `IMEI` 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. Similarly the clerk sometimes gets the name of the phone wrong when entering a new phone into the table `Phone`, for example, mistyping `'iPhone 13 Pro'` instead of `'iPhone 13 Pro Max'`.

Incorporate suitable referential integrity (i.e., `FOREIGN KEY`) constraints into the DDL statements that create the `rentalContract` table and the `Phone` table. In the `rentalContract` table, these should ensure that the `IMEI` and `customerId` fields refer to a valid row of `Phone` and a valid row of `Customer` respectively. In the `Phone` table, the referential integrity constraint should ensure that the combination of `modelNumber` and `modelName` refers to a valid combination of these fields from a row of `PhoneModel`.

The IMEI for a phone can never change, but phones are deleted from the database when they have reached their end-of-life, in which case the corresponding IMEI entry in the `rentalContract` should automatically be set to `NULL`.

(c) IMEI validation (4 marks)

A phone's `IMEI` is very important for any and all paperwork, such as lease agreements, insurance details, etc. However it is very easy to mistype a long sequence of characters, and so we'd like to add some *data entry validation* that will pick up some of these mistakes.

We will be using a simplified version of a real IMEI, so first we need to understand how these work. A real IMEI is a 15-digit string using only numeric characters (i.e., 0-9) that must satisfy the following condition:

If the IMEI has digits

$$d_1 d_2 d_3 \dots d_{14} d_{15}$$

then calculate the following value:

- Calculate the sum of the digits in the *odd-numbered* positions

$$d_1 + d_3 + \cdots + d_{13} + d_{15}$$

- Calcuate the sum of the *Luhn codes* of the digits in the *even-numbered* positions (the Luhn codes are given in Table 1).
- Add these two sums to get a final value and check whether it is a multiple of 10

If this final value is a multiple of 10, then the string is a valid IMEI, while if the final value is *not* a multiple of 10, then the string is not a valid IMEI.

Here is an illustration of this calculation, using the string `'526457946034482'`.

Digit	Luhn Code	Digit	Luhn Code
0	0	5	1
1	2	6	3
2	4	7	5
3	6	8	7
4	8	9	9

Table 1: Luhn Codes

Position	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}
Digit	5	2	6	4	5	7	9	4	6	0	3	4	4	8	2
Odd Position	5		6		5		9		6		3		4		2
Luhn Codes		4		8		5		8		0		8		7	

If we add up all the digits in the odd positions, we get

$$5 + 6 + 5 + 9 + 6 + 3 + 4 + 2 = 40$$

and we add up all the Luhn codes of the digits in the even positions we get

$$4 + 8 + 5 + 8 + 0 + 8 + 7 = 40$$

and if add these two values then we get

$$40 + 40 = 80$$

which is a multiple of 10, and so we conclude that `'526457946034482'` is a valid IMEI. (It is purely an accident that these values are both equal to 40.)

The point of this condition is that if a data entry operator mistypes one digit (which is the most common typing error) then the resulting value is not a valid IMEI, and if the system detects this immediately, then the error can be corrected before it can do any damage. (Of course if the operator makes more than one mistake, then they may accidentally type an incorrect, but still valid, IMEI.)

In a procedural language such as Python, it would be easy to write a loop to calculate these values, but SQLite does not have any way of doing this. So the addition must be done by explicitly extracting each digit from the code and performing the calculation, which means at least 15 statements involving `substr` (to get the digits) along with various other manipulations.

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.

You should implement a `CHECK` constraint for the table `Phone` that will validate a correct 15-digit IMEI (and reject any string that is not a valid 15-digit IMEI).

You will have to decide how to calculate the Luhn code of a digit — the easiest way to do this is to calculate it, based on the fact that it is just obtained by multiplying the digit by 2 and adding the digits of the result together. Because

$$8 \times 2 = 16 \quad \text{and} \quad 1 + 6 = 7$$

the Luhn code of 8 is 7.

There is no really elegant way to do this — you literally have to extract the digits one-by-one add up the associated values (either the raw digit itself for d_1, d_3, d_5 etc. or the Luhn code for d_2, d_4, d_6 etc.) and verify that the final sum is divisible by 10. This ends up as a large and rather ugly expression.

2. `cssubmit createTrigger.sql` A *trigger* to improve data consistency (3 marks)

   The cost of renting a particular phone from KwikiPhone Rentals is calculated according to the `baseCost` and `dailyCost` of that particular type of phone. Obviously, it is more expensive to rent an `'iPhone 14 Pro Max'` than an `iPhone 11`, and obviously a phone rented for 60 days costs more than one rented for 30 days.

   The length of the rental is calculated as the difference between the `dateOut` and `dateBack` *including* the day the phone was taken out and the day it was returned. So a phone taken out on `2023-09-14` and returned on `2023-09-15` counts as a *two day rental*.

   The rental cost is the `baseCost` plus the `dailyCost` multiplied by the number of days. You may find the SQLite function `julianday()` useful in performing the date calculations.

   You tell KwikiPhone Rentals that you can solve another one of their problems by using a *trigger* to automatically enter the correct `rentalCost` when the phone is returned.

   In the file `createTrigger.sql`, write the code to *create a trigger* on the table `rentalContract`. This trigger should fire whenever a row is *updated* (which only happens when the clerk enters the `dateBack` value), then compute the cost of the rental and issue another `UPDATE` statement entering this value into the `rentalCost` field.

   You will need to be careful to distinguish the clerk's `UPDATE` statement from the trigger's `UPDATE` statement or the trigger will keep firing itself in an infinite loop. You can tell the difference because the `dateBack` field will be `NULL` when the update comes from the clerk, but it will not be `NULL` when the update comes from the trigger itself.

   Do not have the trigger re-calculate the rental cost for all the rows— KwikiPhone Rentals frequently alters the costs of renting each type of phone, but this change should only affect *new rentals* and should not alter the calculation for completed rentals. In other words, the trigger should alter *only the current row* of `rentalContract` and not any others.

3. `cssubmit createView.sql` A *view* to improve usability (3 marks)

   For tax purposes, customers often want a list of all of their rentals along with the tax year in which they incurred the expense. The necessary SQL command to extract this information in the right format is a little complicated for the clerk because it involves joining several tables and creating slightly complicated expressions.

   Write the SQL code that defines a *view* named `CustomerSummary` with the following schema:

```
CustomerSummary (
  customerId INTEGER,
  modelName INTEGER,
  daysRented INTEGER,
  taxYear TEXT,
  rentalCost REAL);
```

   Each row should summarise the information for one rental, where the fields `customerId`, `modelName` and `rentalCost` have the obvious meanings, `daysRented` is the number of days for that particular rental and `taxYear` is a string representing the *tax year* when the expense was incurred.

   In Australia, the tax year runs from 1 July to 30 June, and so the period from 1 July 2022 - 30 June 2023 is the 2022/23 tax year. So if a rental *finished* between 1 July 2022 and 30 June 2023, then the entry in the column `taxYear` should be `'2022/23'`, and similarly for the other years. (You may assume that all rentals occurred after the year 2000, so you do not have to consider the change from 1999 to 2000.)

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