CITS1402 Project - FIRST DRAFT

Gordon Royle & Michael Stewart

2022 Semester Two

This project is a *2-week project* that is due at 1700 Sunday 16th October (the Sunday at the end of Week 11). Submissions will continue to be accepted, subject to a late penalty, up until 1700 Sunday 23rd October. This is the *hard deadline*, and *no submissions* will be accepted after that time.

If your submission is *late* due to sickness or misadventure, then you must apply for Special Consideration from your Student Office, and if it is granted then the late penalty will be waived.

If your end up making *no submission*, due to an extended period of sickness or serious misadventure, then you must apply for Special Consideration from your Student Office. If Special Consideration is granted, then the marks will be reallocated to the other assessments in proportion to their original weights.

(If you are a UAAP/AEA student entitled to a routine extension of assessment deadlines, then late penalties will only start to accrue after your guaranteed extension expires. I have a list of all students to whom this applies and I will do this calculation automatically, so you do not need to do anything to make this happen.)

This project is an *individual assessment*, and your submissions will be compared and any statistically-unlikely similarities investigated. In this case, you may be asked to explain your answer or demonstrate that you can write a similar query on-the-spot. In most cases there is a simple explanation that requires no further action.

You may make multiple submissions to `cssubmit` with the more recent submissions over-writing the earlier ones. I suggest that you submit each piece of work as you complete it, because a consistent record of a developing project with multiple submissions over the entire 2-week period is strong evidence of authorship.

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 short project is going to focus on some of the other aspects of databases, in particular features that involve *data integrity*.

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 incorporate data integrity features into a database to make it more resistant to errors caused by careless user

This project explores some of the fundamental data integrity measures that can be taken by the database designer.

The questions may require you to look up 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

   This year, I have noticed that `help1402` has been used less than in previous years, and so it has been quite manageable.

   In previous years, the volume of posts increases dramatically during the two weeks of the Project, but the volume of low-quality posts also increases. I'd rather spend time giving full explanations to good questions, so please try to follow these guidelines before you post.

   - Try to check that your question is actually new

     Try not to ask 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.

   - Only post if you 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`.

   - Ask your questions as precisely as possible

     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. This is at least 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 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.

- Don't post 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).

  If you are really stuck and you need to see why a particular piece code is not working, then email `cits1402-pmc@uwa.edu.au` with the details.

Senior Citizens Mobile Library (SCML)

The local council runs a Senior Citizens Mobile Library that travels to various aged-care residences lending books to the elderly residents.

Currently, all the lending records are kept in an Excel spreadsheet, but as SCML is expanding, this is becoming increasingly cumbersome and so SCML is moving to an SQLite database.

They have already started their implementation and have designed some of the tables they will need, but they need advice on data integrity and on techniques to keep the database internally consistent.

Currently, the database has four tables, namely `BookEdition`, `BookCopy`, `loan` and `Client` which have the following structure:

The table `BookEdition`

The table stores data about *editions* of a book (not individual physical copies of a book).

```
CREATE TABLE BookEdition (ISBN TEXT,
   title TEXT,
   author TEXT,
   publicationDate TEXT,
   genre TEXT)
```

Each row of BookEdition is uniquely identified by its ISBN and contains the following additional information about the book edition: the title, author, publication date and main genre.

An example of a tuple might be:

```
('9780593607695', 'Any other family', 'Eleanor Brown', 2022, 'Family')
```

indicating that the ISBN 9780593607695 refers to a 2022 edition of the book *Any Other Family* written by *Eleanor Brown* and that it is in the genre *Family*.

(Real ISBNs are often displayed in a stylised form with hyphens or spaces separating various groups of digits, but it is only the digits that count.)

The table BookCopy

This table stores data about the actual physical books in the SCML collection.

```
CREATE TABLE BookCopy (ISBN TEXT,
      copyNumber INTEGER,
      daysLoaned INTEGER)
```

As their clientele are spread across many residences, SCML tends to have many copies of their most popular books. So if SCML has, say, 10 copies of a book from the same edition, then the 10 books are numbered from 1 to 10, and this "copy number" is stamped inside the front cover of the book.

Each individual book is then represented by a row in the database containing the book's ISBN and copy number. Therefore a row of BookCopy is uniquely determined by the combination of ISBN and copyNumber.

SCML realises that they buy too many copies of some books, and too few of others, and so they want to keep statistics on how often the books are actually on loan. They have added an additional column daysLoaned to the table BookCopy, to keep track of the total number of days on which this individual book copy has been out on loan. They wish to analyse this data in the future and hope to keep these values updated every time a book is loaned out and returned.

The table BookEdition contains many more books than SCML can possibly buy, and so SCML may have zero, one or more actual copies of each of the books listed in BookEdition.

The table loan

Each row of loan stores details for a single loan of a particular physical copy of a book to a particular client.

```
CREATE TABLE loan (clientId INTEGER,
    ISBN TEXT,
    copyNumber INTEGER,
    dateOut TEXT,
    dateBack TEXT)
```

The information stored about the loan is the clientId identifying the client loaning the book, the two fields ISBN, copyNumber needed to identify the physical book, and then two fields dateOut and dateBack, giving the dates (in YYYY-MM-DD form) that the book went out to the client and came back from the client.

A new tuple is entered into the table `loan` on the day that the book is loaned out, with the value `NULL` assigned to the column `dateBack` (because this value is not known).

When the book is returned, the row is *updated* by a statement of the form:

```
UPDATE loan
SET dateBack = '2022-08-15'
WHERE ISBN = '9780593607695' AND copyNumber = 5 AND dateOut IS NULL;
```

At this stage, the volunteer is meant to calculate the number of days in this loan and update the `daysLoaned` column in the table `BookCopy`.

The table `Client`

This table records the details for each SCML client.

```
CREATE TABLE Client (
  clientId INTEGER,
  name TEXT,
  residence TEXT)
```

Each client has a unique `clientId` and SCML also keeps information about the client's name and their aged-care residence. A sample tuple in `Client` might be

```
(112, 'Jill Borowicz', 'Shady Acres Subiaco')
```

indicating that Jill Borowicz is client number 112 and that she lives in the *Shady Acres Subiaco* aged-care residence.

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 tables or columns, but just *adding* database features to improve the integrity and usability of the database.

You are asked to submit seven files:

```
ERD.png
BookEdition.sql
BookCopy.sql
loan.sql
Client.sql
loanTrigger.sql
ReadingHistory.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 *existing structure* of the database, in order to facilitate further discussions with SCML. Your ERD should have

(a) Entity sets, attributes, relations and relationship attributes that reflect the *current structure* of the database.

(b) The key, participation and cardinality constraints that should be added according to the descriptions of the four tables given above.

The intention is for you to produce a feasible ERD that, when translated to a relational schema, ends up with the existing structure—in other words, to "reverse engineer" the existing database.

Please remember that this part of the project is *not about* designing a better schema, but about *describing* an existing schema. So *do not* incorporate entities or attributes into the ERD that are not in the database (even if you think you can make improvements) or you will lose marks. There is one exception to this, because when an ERD is converted to a relational schema, some of the relationships in the ERD are not implemented as a table, but via foreign keys instead. In this situation, the name of the relationship in the ERD does not appear in the relational schema at all, and so you will have to make up just one name.

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 via `cssubmit`.

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 new database schema (2 + 2 + 2 = 6 marks as specified below)

   You should prepare files called

   ```
   BookEdition.sql
   BookCopy.sql
   loan.sql
   Client.sql
   ```

   that each contain a DDL statement (i.e., a `CREATE TABLE` statement) that will create the tables `BookEdition`, `BookCopy`, `loan` and `Client` respectively. Each of the statements should create a table with *exactly the same* columns and data types as in the current database, but with *additional data integrity features* as described below.

   Each of your files will be *tested individually* — for example, when marking `loan.sql` we will create a database that uses our model solutions to create `BookEdition`, `BookCopy` and `Client`, but your submission to create `loan`, and then test that it works properly.

   Your files should *only* create the *empty* tables — we will use our own simulated data to test that your tables behave according to the specification.

   (Of course you may want to test your own tables by making up some data of your own, but you should not leave any trace of this in your submitted files.)

   Marks will be allocated for the following additional features that you should incorporate into your improved schema.

   (a) Primary key constraints in `BookEdition`, `BookCopy` and `Client` (2 marks)

   The description of the table `Client` indicated that a client should be uniquely identified by the `clientId` column. However at the moment, there is nothing to prevent two different rows being entered into the table with the same client number.

   The sections describing `BookEdition` and `BookCopy` also identified columns or combinations of columns that would uniquely identify a row in those tables.

Add key constraints to the DDL statements that you submit to create these three tables, so that SQLite will enforce these constraints on these three tables.

(Do not add any primary key constraints to the table `loan`.)

(b) Referential integrity in `loan` (2 marks)

One problem faced by SCML is that it is quite busy when they visit an aged-care residence and it is very easy to mistype numbers when entering rows into the table `loan`. This results in rows of the table `loan` that refer to clients or books that don't actually exist, and it takes them some time to correct these errors.

SCML realises that the best time to catch these typing errors is at the moment that the row is entered into the table `loan` by having the system check that any new rows actually refer to genuine clients and actual books.

Your file `loan.sql` should create the table including *referential integrity constraints* ensuring that the combination of `ISBN` and `copyNumber` refers to an actual book, and that `clientId` refers to an actual client.

You are given some further information about how SCML's operations: SCML wants to be able to change a client's id (in the table `Client`) and to have that change automatically reflected in the table `loan`. Due to the nature of their business, they frequently lose clients, in which case they delete the corresponding row from `Client`. If this happens, then the loans involving that client should remain in the table `loan` (for the purposes of analysing the data on books loaned out), but the client id should be set to `NULL`.

You are told to assume that no rows of `BookCopy` will be updated or deleted, so *do not* specify any particular action to deal with changes on this table.

(c) Data entry validation for `BookEdition`

The 13th digit of a real ISBN is a *check digit* calculated from the first 12 digits by taking the last digit of the sum

$$(d_1 + d_3 + d_5 + d_7 + d_9 + d_{11}) + 3 \times (d_2 + d_4 + d_6 + d_8 + d_{10} + d_{12}).$$

For example, if we consider the book with ISBN `9780593607695` then the calculation gives

$$9 + 8 + 5 + 3 + 0 + 6 + 3 \times (7 + 0 + 9 + 6 + 7 + 9) = 145$$

and indeed the 13th digit of the ISBN is 5.

The most common error when typing a number is to get one digit incorrect. If this happens while typing an ISBN, then the *actual* final digit will not match the *required* final digit. Therefore systems dealing with ISBNs should automatically check that it is valid.

We will not use 13-digit ISBN for this project because it is too cumbersome, but instead we will imagine that ISBNs are only 5 digits long, and assume that the check digit of the ISBN is obtained by taking the final digit of the value

$$3 \times (d_1 + d_3) + 7 \times (d_2 + d_4).$$

Your file `BookEdition.sql` should create the table `BookEdition` with an additional constraint to prevent the insertion of any rows with an invalid ISBN. More precisely, your constraint should ensure that the ISBN field satisfies the following conditions:

- The ISBN is exactly 5 characters long.
- The 5 characters are all numeric digits (from `0` to `9`)
- The 5th digit is determined by the first 4 digits according to the calculation given above.

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

When a book is returned, the row for that book in the table `loan` is updated by an `UPDATE` statement changing the value of `dateBack` from `NULL` to the return date.

At this point, the person updating the database is also meant to calculate the number of days that the book has been out on loan (using `dateOut` and `dateBack`) and then add this value to the `daysLoaned` column in the corresponding row of `BookCopy`.

However because it is busy, this step is often not done, or the calculation of the days is done incorrectly, so it would be much better if these values could be updated *automatically*.

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

Your file `loanTrigger.sql` should *create a trigger* on the table `loan` that fires when a row is updated, calculates the number of days from this loan, and adds the value to the `daysLoaned` column of the corresponding row in `BookCopy`.

When calculating the number of days, include *both* of the days specified in `dateOut` and `dateBack`. So if a book is loaned out on `2022-08-25` and returned on `2022-08-29` then it has been out for 5 days. You will find the built-in SQLite function `julianday()` useful — this takes a string such as `'2022-08-25'` representing a date and calculates how many days have elapsed from the beginning of the Julian period (a fixed day in the year 4714 BC) to the given date.

Note that it would be easy to simply make your trigger update every row in the table `Book`, rather than update only the row that is being changed, but this is not allowed because it defeats the purpose of the question. In particular, we will check that your trigger only affects one row of `BookCopy`.

4. A view `cssubmit ReadingHistory.sql` (2 marks)

   Write the SQLite code to *create a view* of the data called `ReadingHistory` that has the following schema:

```
ReadingHistory(
  clientId INTEGER,
  yr INTEGER,
  genre TEXT,
  numLoans INTEGER);
```

   For each combination of client, year and genre, the view lists the clientId, the year, the genre and the *number of loans* of books of that genre made to the client in that year. Use the `dateOut` column when deciding what year a loan belongs to (so a book borrowed on December 25 2021 and returned on January 5 2022 counts towards the 2021 total).

   For example, if Jill Borowicz had borrowed 10 adventure novels, 4 fantasy novels and 6 biographies in 2022, then the command

```
SELECT * FROM ReadingHistory
WHERE clientId = 112 AND yr = 2022;
```

   should produce the following output:

```
112 2022 'fantasy'    4
112 2022 'adventure' 10
112 2022 'biography'  6
```

   (If Jill borrows the same book three times, then this counts as three loans.)

   We will be testing your code by creating the database tables with our model solution, and then creating *just the view* with your code, prior to testing it with simulated data.

   So it is *absolutely imperative* that every column has *exactly* the correct name. You are free to alter the way a name is capitalised (so `numloans` and `numLoans` are the same), but apart from that you cannot add or remove even a single letter. For example, `numLoan` instead of `numloans` will cause the testing to fail.

   Make sure that your file starts with

```
CREATE VIEW ReadingHistory AS
```