University of Windsor

School of Computer Science
Faculty of Science
COMP-3150: Database Management System (Fall 2022)

Lab#	Date	Title	Due Date	Grade Release Date
Lab4	Week 07	**Data Manipulating Language (DML)**	Two-Week Lab November 16, 2022, Wednesday Midnight EDT	November 21, 2022

The objective of this lab to is work with Data Manipulation Language (DML) of the SQL including `INSERT`, `DELETE`, `UPDATE`, and `SELECT`. This will give us an understanding of how data are stored, retrieved and modified within tables of relational databases.

Step 1. Insert New Information

Given that you have already designed the tables for your project in Lab3, we will focus on working with the data that is to be stored in these tables. I will continue to work on my sample movie project (`Moviesion`) in this lab as well. You will need to perform the following steps for your own project.

Now that we have set up a database we can put some information in it. Suppose we want to add the following movies to the database:

Director	Title	Genre
Hitchcock	*Psycho*	*Horror*
Hitchcock	*The Birds*	*Horror*
Scorsese	*Raging Bull*	*Drama*
Kurosawa	*Seven Samurai*	*Samurai*
Wilder	*Sunset Boulevard*	*Film Noir*
Spielberg	*Schindler's List*	*Drama*
Welles	*Citizen Kane*	*Film Noir*

Each movie is going to require two entries - one in the `Movie` table, and one in the `Director` table. However, we don't want to input each director's information more than once, and we need to know the director's `Id` before we can insert the movie. This means that there are three stages to inserting a movie:

1. Check the `Director` table and see if the director for the movie already exists or not
2. If the director is already stored in the `Director` table, just retrieve his/her `Id`
3. If the director is not stored in the `Director` table, insert it and retrieve its `Id`
4. Insert the movie into the `Movie` table and for `DirectorId`, use the `Id` that was retrieved in step 2 or 3.

Now, to add a new director to `Director` table, I will use the following command:

```
INSERT INTO Director(Id, Name) VALUES (1, 'Hitchcock');
```

You can even ignore entering the `Id` and the database will create an `AUTOINCREMENT` Id for you:

```
INSERT INTO Director(Name) VALUES ('Hitchcock');
```

If you try the same insert command again, you will violate the primary key constraint and receive error message as follows:

```
Error: UNIQUE constraint failed: Director.Id
```

If you try the same insert command again, this time you will violate the unique key constraint and receive error message as follows:

```
Error: UNIQUE constraint failed: Director.Name
```

To check whether data was properly inserted into the table, you can check the values in the table:

```
SELECT *
FROM Director;
```

Now, in the four steps above, I wanted to first check whether a director was already in the table or not, I can simply check this by extending the above select statement:

```
SELECT Id
FROM Director
WHERE Name = 'Hitchcock';
```

This will return the `Id` for `Hitchcock` if it exists; otherwise it will return nothing. As we already insert this director into `Director` table with `Id=1`, we can insert movies which directed by him into `Movie` table given his id as follows:

```
INSERT INTO Movie(Title, Genre, DirectorId) VALUES ('Psycho', 'Horror', 1);
INSERT INTO Movie(Title, Genre, DirectorId) VALUES ('The Birds', 'Horror', 1);
```

If you insert a movie with a `DirectorId` value which does not exist in `Director` table (e.g., `DirectorId=2`), you violate the foreign key constraint and receive the following error message:

```
INSERT INTO Movie(Title, Genre, DirectorId) VALUES ('The Birds', 'Horror', 2);

Error: FOREIGN KEY constraint failed
```

Make sure you enter 5 entries for each of your tables and also ensure that all your foreign keys are respected when data are entered in each table.

Step 2. Retrieving Information

Let's suppose now we want to find all the movies that have the same genre as the movie called `Inception`. For this we would need to know about the row for `Inception` from the `Movie` table (to know its genre) and other rows at the same time (to see if they have same genre). We can do this by using aliases to make two copies or references to the movie table. One copy, `MovieInception`, we'll use to see `Inception`'s genre, and the other, `MovieOther`, will let us compare this genre to other movies. This leads to the query:

```
SELECT Movie_Other.Title
FROM Movie AS[1] MovieInception, Movie AS MovieOther
WHERE MovieInception.Title = 'Inception'
  AND MovieInception.Genre = MovieOther.Genre
```

In the above example, I am assuming that I have columns called `Genre` and `Title` in my `Movie` table.

Combining tables, and copies of tables using aliases, allows us to build up very complex queries, but they can get quite difficult to understand. Subqueries provide a way of passing the results of one query into the `WHERE` clause of another. This lets us break the problem down into smaller parts, and then write queries for each part. For example, if we want to find all the `Movie` that have the same genre as `Inception`, we can write:

```
SELECT Title
FROM Movie
WHERE Genre = (
               SELECT Genre
               FROM Movie
               WHERE Title = 'Inception'
              )
```

[1] While optional in SQLite, some other DBMS might require you to use `AS` keyword to define aliases.

Note in this query the `Genre` in the subquery refers to the genre of `Inception`, while the `Genre` outside of the subquery refers to the genre of the results. If you need to be able to use both together, then you can use aliases to do so.

Let us now look at a more complex query for finding a list of the titles of all movies that have the same genre as any movie produced by Martin Scorsese. The query to find the genre of movies produced by `Scorsese` is:

```
SELECT Genre
FROM Movie, Director
WHERE Movie.DirectorId = Director.Id
  AND Directors.Name = 'Scorsese'
```

This will typically return more than one result, so we can't use the query in the following way with the `=` operator. So, the following query will not work:

```
SELECT Title
FROM Movie
WHERE Genre = (
                SELECT Genre
                FROM Movie, Director
                WHERE Movie.DirectorId = Director.Id
                  AND Directors.Name = 'Scorsese'
              )
```

In SQLite the above query works as it selects the first row of the subquery! But it may not be what you want in the final results. Given there may be more than one value from the subquery, we cannot compare a single value to a set with the `=` operator. Instead, we can use the `IN` operator for checking whether a value is within a set as follows:

```
SELECT Title
FROM Movie
WHERE Genre IN (
                 SELECT Genre
                 FROM Movie, Director
                 WHERE Movie.DirectorId = Director.Id
                   AND Directors.Name = 'Scorsese'
               )
```

It is also possible to delete rows of information from each table. If we decide to delete the information for a specific director from our director table, we can use the following statement to achieve this:

```
DELETE FROM Director WHERE Name = 'Hitchcock';
```

Sometimes rows cannot be removed because of foreign key constraints. If, for example, we were to remove the entry for `Hitchcock` from the `Director` table, then the Id's for any movies that have this director would become invalid, and *referential integrity* violated. Because a foreign key has been set up between `Movie` and `Director`, SQLite can detect this and stop you from deleting directors who have movies in the `Movie` table. As we already had inserted two movies for Hitchcock, SQLite raise the following error for the above delete command:

```
Error: foreign key mismatch
```

Step 2. Update Information

It is also possible to update the information within a table. To do this, we employ the update statements. Let's suppose that we had entered `Hitchcock`'s name incorrectly and we need to fix this error:

```
UPDATE Director SET Name = 'Hitchcock' WHERE Name = 'Hitchkok';
```

It is very important to get the `WHERE` clause on an `UPDATE` statement right. Running an `UPDATE` without a `WHERE` clause will change all of the rows in a table. For example, (*don't do this!*) the command:

```
UPDATE Director SET Name = 'Welles';
```

will change all the directors' name to `Welles`. If you want to check the `WHERE` clause, you can try it out with a `SELECT` statement first, to make sure that it returns the rows you expect. For example:

```
SELECT *
FROM Director
WHERE Name = 'Welles';
```

Now, you are expected to go back to your submission for Lab1 and find the list of tasks that you wanted to perform for your project. For each of the tasks present the list of one or more SQL statements that you need to successfully perform. Use `INSERT` if you need to add data, `DELETE` if you need to remove some information and `UPDATE` if you need to modify data. For instance, If you are developing a course registration application and one of the tasks is to register a student, you'd need an `INSERT` statement that adds a new student into a course and you also need to make sure the foreign keys are respected. If a student drops a course, then you'd need to use `DELETE` or `UPDATE` to handle the situation. You will need to provide one or more SQL statements that will allow you to achieve each of your intended tasks.

Final Step. Deliverables

You should complete the steps described above and submit the following items in one single `zip` file `Lab4_uwinid.zip`:

- (70%) `Lab4_uwinid.zip`
  - (20%) `DatabaseName.db` → The database file whose tables have at least 5 rows. Instead of `DatabaseName`, use your own database name, e.g., mine is Movision, so, `Movision.db`.
  - (20%) `DatabaseName.sql` → The series of all SQL commands (script).
  - (30%) `Report.pdf` → Lab report including name, student id, and the list of at least 10 tasks that your project does and the list of SQL statements that you would need to accomplish each task. Remember, you might need more than one SQL statement for one task. For each task, briefly explain how the one or more SQL statements will allow you to perform that task. Respect code convention when writing SQL statements as we did above, i.e., SQL keywords should be UPPERCASE, each part of the statement should be in new line, and etc.
  - (Optional) `ReadMe.txt` → optional additional information that helps lab instructor or grader to evaluate
- (30%) Naming (PascalCase) and Formats

Instead of `uwinid`, use your own account name, e.g., mine is hfani@uwindsor.ca, so, `Lab3_hfani.zip`