Lab 4 – Manipulating Data

This is a two-week lab

The objective of this lab is to work with data manipulation statements of the SQL language including insert, delete, and update. This will give us an understanding of how data are stored, retrieved and modified within tables of relational databases.

Given that you have already designed the tables for your project in Lab 3, 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 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 that we are going to insert is already in our director table or not
2. If the director is already stored in the director table then, just retrieve her dirID
3. If the directory is not stored in the director table then insert it and retrieve its dirID
4. Insert the movie into the movie table and for the dirID value, use the dirID that was retrieved

The final problem is generating IDs. One way to do this is to just make them up, which is fine for a very small database maintained by hand, but for even moderately sized databases, or any sort of automation this becomes a problem. Most DBMSs offer some way to generate IDs. In fact, there is a functionality that would create incremental IDs automatically for you. This is called autoincrement.

In SQLite, a column with type INTEGER PRIMARY KEY is an alias for the ROWID, which is always a 64-bit signed integer. On an INSERT, if the ROWID or INTEGER PRIMARY KEY column is not explicitly given a value, then it will be filled automatically with an unused integer, usually one more than the largest ROWID currently in use. This is true regardless of whether or not the AUTOINCREMENT keyword is used.

Now in order to add a new director to my director table, I will use the following command:

INSERT INTO director (dirID, dirName) VALUES (1, 'Hitchcock');

You can even ignote entering the dirID and the database will create an autoincremented ID for you.

INSERT INTO director (dirName) VALUES ('Hitchcock');

*Make sure you enter 10 entries for each of your tables and also ensure that all your foreign keys are respected when data are entered in each table.*

In order 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 dirID FROM director WHERE dirName='Hitchcock';

This will return the dirID for Hitchcock if it exists; otherwise it will return nothing.

Lets suppose now we want to find all the movies that have the same price as the movie called Inception. For this we would need to know about the row for Inception from the Movie table (to know how much it cost) and other rows at the same time (to see if they cost the same). We can do this by using aliases to make two copies or references to the movie table. One copy, Movie\_Inception, we'll use to see how much Inception costs, and the other, Movie\_Other, will let us compare this price to other movies. This leads to the query:

SELECT Movie\_Other.movieTitle FROM   
 Movie Movie\_Inception, Movie Movie\_Other  
 WHERE Movie\_Inception.movieTitle = ‘Inception’  
 AND Movie\_Inception.cdPrice = Movie\_Other.cdPrice

Some database version would require you to use the ‘as’ keyword to define aliases:

SELECT Movie\_Other.movieTitle FROM   
 Movie Movie\_Inception, Movie Movie\_Other  
 WHERE Movie\_Inception.movieTitle = ‘Inception’  
 AND Movie\_Inception.cdPrice = Movie\_Other.cdPrice

In the above example, I am assuming that I have columns called cdPrice and movieTitle 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 Movies that have the same price as Inception, we could write a statement like

SELECT movieTitle FROM movie WHERE cdPrice =   
 (SELECT cdPrice FROM movie WHERE movieTitle = ‘Inception’)

Note in this query the 'cdPrice' in the subquery refers to the price of Inception, while the 'cdPrice' outside of the subquery refers to the price 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 movieGenre FROM Movie, Director   
 WHERE Movie.dirID = Director.dirID  
 AND Director.dirName = '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 movieTitle FROM Movie  
 WHERE movieGenre **=** (SELECT movieGenre FROM Movie, Director   
 WHERE Movie.dirID = Director.dirID  
 AND Director.dirName = 'Scorsese')

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 movieTitle FROM Movie  
 WHERE movieGenre **IN** (SELECT movieGenre FROM Movie, Director   
 WHERE Movie.dirID = Director.dirID  
 AND Director.dirName = 'Scorsese')

It is also possible to delete rows of information from each table. Assuming that 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 dirName = '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 dirID'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.

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

UPDATE Director SET dirName = 'Hitchcock' WHERE dirName = 'Hichkok';

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 dirName = 'Welles';

will change all of the Director's names 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 dirName = 'Wells';

Now, you are expected to go back to your submission for Lab 1 and find the list of 10 tasks (you were expected to have at least 10) that you wanted to perform for your project. For instance, I had included:

* How many movies did I watch in a certain year or certain month of a year?
* What were the highest rated movies or lowest rated movies that I watched?
* Who were the actors of the movies that I really enjoyed?
* Where there actors that appeared in all the movies that I really liked or disliked?

For each of the tasks present the list of one or more SQL statements that you need to successfully perform the task. For instance, use INSERT if you need to add data, DELETE if you need to remove some information and update if you need to modify data.

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.

**Deliverables**

You should complete the steps described above. Then, you will prepare and submit the results in one single zip file (**YourName\_Lab4.zip**) containing the following item:

* A lab report document: The lab report should be prepared using a word processor, and should be stored as a single PDF file. This PDF document should be named as follows: **YourName\_Lab4\_Report.pdf**. This is what should be included in your PDF file:

1. Your name and student number
2. A dump of your tables that would include the structure and data of your tables. Make sure each of your tables have 10 rows in them. You can do this by performing the following command on SQLite:

sqlite3 dbName .dump

You need to replace dbName with the name of your database.

1. A list of at least 10 tasks that you want your project to do 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.

Use the following command to complete the submission: