ORIE 3120
Industrial Data and Systems Analysis
Recitation 1
Relational Database Concepts
Peter Jackson
Edited for SQLiteStudio 3 by L. Seligman, V. Sung, and P. Frazier

Basic Concepts

Your teaching assistant will explain or review the following concepts:

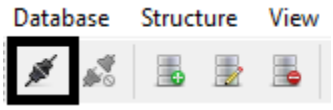
- Table (data presented in rows and columns)
- Database table (data presented in rows using fixed, pre-specified columns)
- Fields (columns of a database table, with pre-specified attributes)
- Records (rows of data in a database table)
- Data types (typical field types: text, integer, Boolean, single-precision, double-precision, etc.)
- Key (a collection of fields that uniquely define a record)

You should be provided with the following files:

- Rec01_StudentAnswerSheet.txt
- Chinook_Sqlite.sqlite
- MediaType.csv

Getting Started

1. To receive credit for attendance, fill in the student answer sheet text file with the answers to the questions asked in this recitation and upload to Gradescope. You may work by yourself, or with one other person, but please do not work in groups larger than two. If you work with another person, note this on your submitted solutions.
2. Create a directory to store your work. If you use the lab machine, be aware that the directory (and all your work) will disappear when the machine is rebooted. We recommend using a USB flash-drive.
3. Open SQLiteStudio 3. Screenshots and descriptions in this recitation are based on SQLiteStudio v3.1.1 on Windows 10. SQLiteStudio is a little bit different on Mac, so be aware that the interface will be a bit different on that operating system. If you are using your own computer and haven't installed it yet, you can install it from http://sqlitestudio.pl/.
4. Open an existing database by selecting the option "Add a database" under the "Databases" tab. Download and select the file "Chinook_Sqlite.sqlite" from blackboard. You can leave checked the box "Permanent (keep it in configuration)". Then, click the "Connect to the database" icon as outlined in this screenshot:



5. Your TA will briefly explain the following sections of your database file:

- Tables
- Views

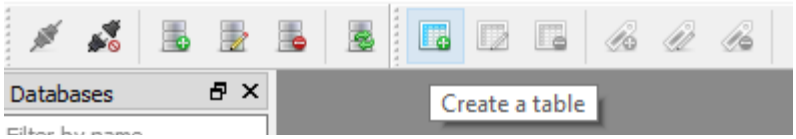
Creating Tables and Manipulating Records

We will consider two ways in which you can import data into your database:

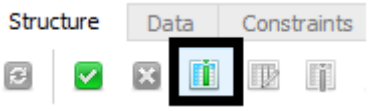
1. Define a table and enter data manually
2. Import data from a CSV (the kind that can be created in Microsoft Excel, or in many other ways)

Define a table and enter data manually:

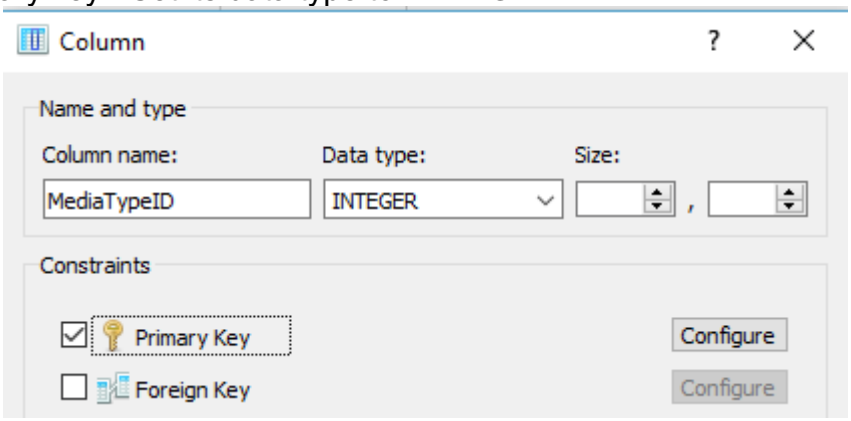
1. Click on the “Create a table” icon as highlighted in the screen shot below:



2. Name your table “MediaType2”. Add fields by clicking the button with tooltip “Add Column (Ins)” above the table name you just entered.



3. Create a column named “MediaTypeID” and check the box to make it the “Primary key”. Set its data type to “INTEGER”.



4. The other boxes (“Foreign Key”, “Unique”, “Not NULL” etc.) allow you to specify other properties you would like to require of the data in this column. For example, if you clicked “Not NULL”, the database would check that all every row entered in this table would have a value (i.e., would not be NULL). We will leave these other boxes unchecked. Go ahead and click “OK” to finish adding this column.
5. Create a column named “Name”, set its data type to “TEXT”, and click “OK”.

6. You do not need to add any constraints. Click on the button with the green check mark with the tooltip “Commit structure changes”. Here, “Commit” refers to the fact that we have described some changes to the structure of the database (we want to create a new table) but we haven’t actually “committed” or applied on them yet. SQLiteStudy will then show the SQL command that will be executed to create the table that you have described using the GUI. This query is:

```
CREATE TABLE MediaType2 (
        MediaTypeID INTEGER PRIMARY KEY,
        Name TEXT
);
```

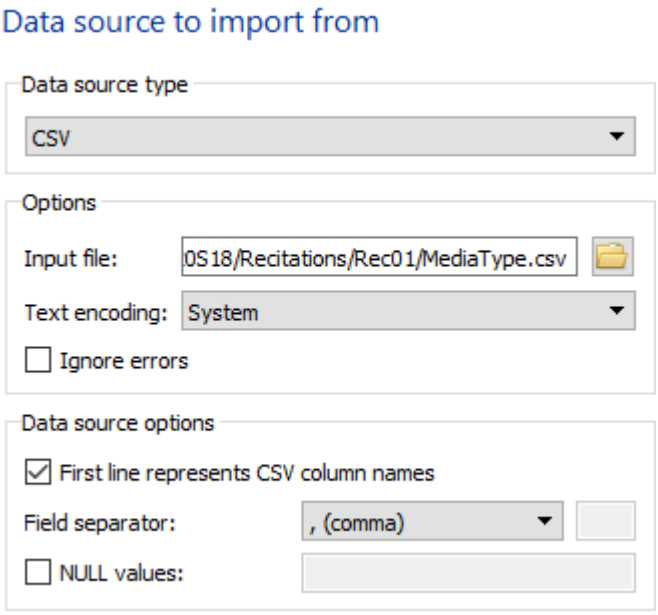
   You could have also created this table by typing this command instead of using the GUI to specify what you wanted to do.
7. Double click “MediaType2” under tables to view the contents of the table. The table should have no data in it. Make sure you are viewing the table in the “Data” tab in “Grid View” as shown in the screen shot below:

8. You can add data manually data to the table “MediaType2” by clicking the green plus sign to add records, filling in fields, and clicking the green check mark to commit the changes when you are done. Play around with adding records and filling in values, but do not commit your changes. Instead delete the rows you have added by clicking the red minus sign so that the table is still empty. If you accidentally commit new records, you can remove them and commit your change.

Import data from a CSV:

1. We will now add data from the “MediaType.csv” file you have been provided. Download this file and take note of the folder you put it in.
2. Click the “Import data to table” icon circled in the screen shot below:

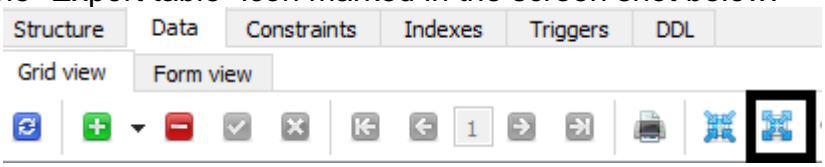
3. Confirm that the table into which data will be imported is MediaType2 from the database Chinook_SQlite. Then click “Next”.
4. Set the input file field to contain the location of the file you downloaded, “MediaType.csv”. Make sure “First line represents CSV column names” is clicked, as in the screen shot below. Then click “Finish”.



5. Verify that the data is correctly imported. You should see a blue status message in the bottom of the SQLiteStudio window saying, “Imported data to the table ‘MediaType2’ successfully” and 5 records in the table, with different audio file formats, e.g., “Purchased AAC audio file.” If the column names were incorrectly imported as a record, delete it and commit the change.
6. Note that this same method of importing data works regardless of what program we use to create .csv file.

Copying data to a spreadsheet:

1. Open table “Invoice” and view the data in “Grid View”.
2. Click the “Export table” icon marked in the screen shot below:



3. Make sure the “Table” field is populated by “Invoice”. Ensure that “Export table data” is checked. We will export to a csv and so it doesn’t matter whether the other options (for triggers and indices) are checked. Click “Next”.

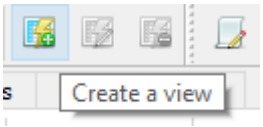
4. For the “File” field, select a folder and filename into which SQLiteStudio will place your data. Check “Column names in first row”. Make sure “Export format” is CSV. Click “Finish”.
5. Find the file in the folder you specified in the previous step and make sure it contains the desired data, along with the column names.

Using the *SELECT* Query

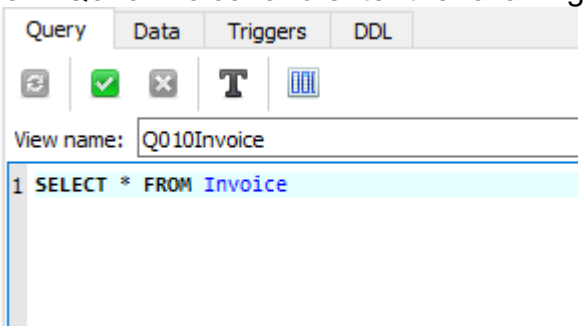
The purpose of the *SELECT* Query:

Defn: The ***SELECT*** query is the basic tool for getting the data you need from a relational database. Whether you are using SQL for data mining, data manipulation, data analysis, or data reporting, your work starts with the SELECT query.

1. Let’s create our first query. We want to create a query that will allow us to generate a view of the entries of the “Invoice” table. We can either create this query using the SQL query editor (which can be accessed using the “pencil & paper” icon on the top menu bar), or by creating a view. We will create a view.
   *Note the difference between a **query** and a **view**.*
   Defn: A ***query*** is SQL code executed to pull rows from our database.
   Defn: A ***view*** is a saved query. By making a query into a view, we can refer to its results in other queries.
2. Click on the “New view” icon in the toolbar.

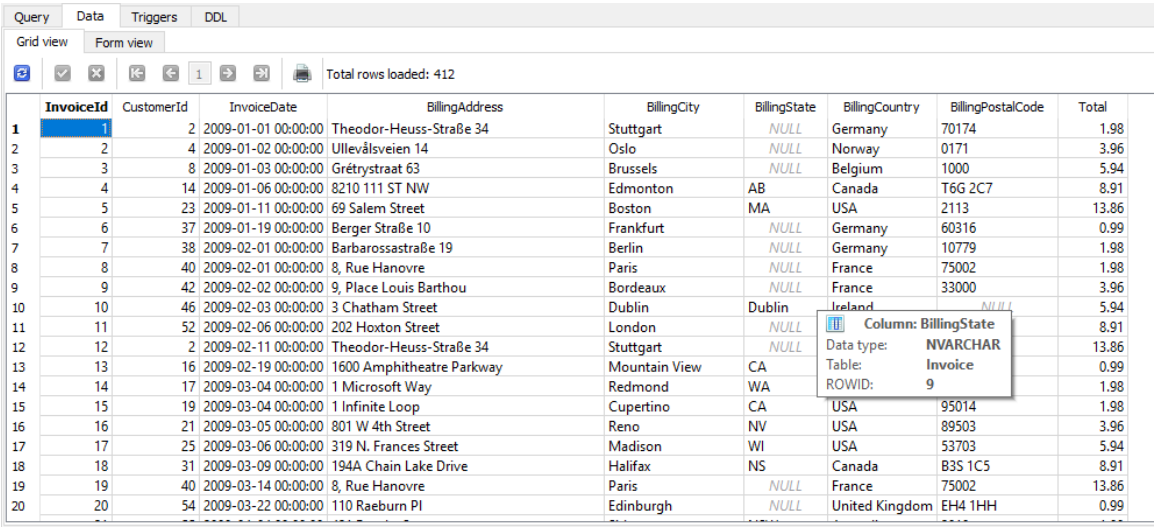


3. Let’s name the view “Q010Invoice” and enter the following SQL code:



Note the syntax used in the SQL code.
The ***asterisk “*”*** used in the SELECT query gets all the rows and columns from the table after the FROM keyword. In other words, the above SQL code “selects all rows and columns from the table called ‘Invoice’.” Although SQL commands like “SELECT” and “FROM” are often written in all caps, SQL does not require this, and your query would run if you used different capitalization.

4. Click the green check mark and then click “OK” to save the view. You should see a message indicating success in the status window: “Committed changes for view ‘Q010Invoice’ (named before “) successfully.”
5. Click on the “Data” tab to see the results from the view.


Query | Data | Triggers | DDL

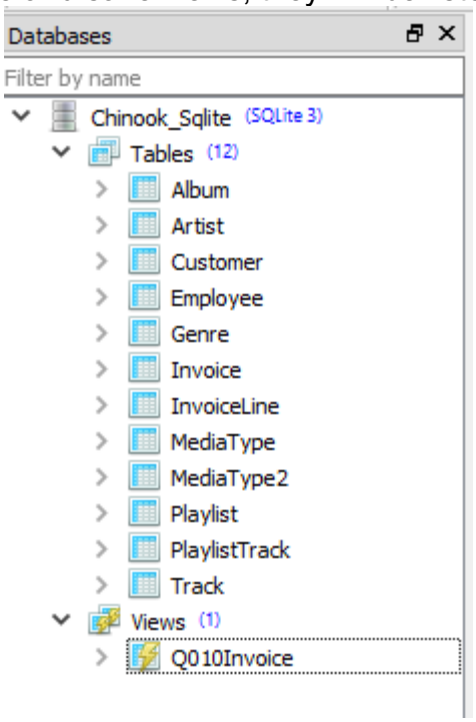
Grid view | Form view

Total rows loaded: 412

	InvoiceId	CustomerId	InvoiceDate	BillingAddress	BillingCity	BillingState	BillingCountry	BillingPostalCode	Total
1	1	2	2009-01-01 00:00:00	Theodor-Heuss-Straße 34	Stuttgart	NULL	Germany	70174	1.98
2	2	4	2009-01-02 00:00:00	Ullevålsveien 14	Oslo	NULL	Norway	0171	3.96
3	3	8	2009-01-03 00:00:00	Grétrystraat 63	Brussels	NULL	Belgium	1000	5.94
4	4	14	2009-01-06 00:00:00	8210 111 ST NW	Edmonton	AB	Canada	T6G 2C7	8.91
5	5	23	2009-01-11 00:00:00	69 Salem Street	Boston	MA	USA	2113	13.86
6	6	37	2009-01-19 00:00:00	Berger Straße 10	Frankfurt	NULL	Germany	60316	0.99
7	7	38	2009-02-01 00:00:00	Barbarossastraße 19	Berlin	NULL	Germany	10779	1.98
8	8	40	2009-02-01 00:00:00	8, Rue Hanovre	Paris	NULL	France	75002	1.98
9	9	42	2009-02-02 00:00:00	9, Place Louis Barthou	Bordeaux	NULL	France	33000	3.96
10	10	46	2009-02-03 00:00:00	3 Chatham Street	Dublin	Dublin	Ireland	NULL	5.94
11	11	52	2009-02-06 00:00:00	202 Hoxton Street	London	NULL	United Kingdom	NW1 5LH	8.91
12	12	2	2009-02-11 00:00:00	Theodor-Heuss-Straße 34	Stuttgart	NULL	Germany	70174	13.86
13	13	16	2009-02-19 00:00:00	1600 Amphitheatre Parkway	Mountain View	CA	USA	94043	0.99
14	14	17	2009-03-04 00:00:00	1 Microsoft Way	Redmond	WA	USA	98052	1.98
15	15	19	2009-03-04 00:00:00	1 Infinite Loop	Cupertino	CA	USA	95014	1.98
16	16	21	2009-03-05 00:00:00	801 W 4th Street	Reno	NV	USA	89503	3.96
17	17	25	2009-03-06 00:00:00	319 N. Frances Street	Madison	WI	USA	53703	5.94
18	18	31	2009-03-09 00:00:00	194A Chain Lake Drive	Halifax	NS	Canada	B3S 1C5	8.91
19	19	40	2009-03-14 00:00:00	8, Rue Hanovre	Paris	NULL	France	75002	13.86
20	20	54	2009-03-22 00:00:00	110 Raeburn Pl	Edinburgh	NULL	United Kingdom	EH4 1HH	0.99

Column: BillingState
Data type: NVARCHAR
Table: Invoice
ROWID: 9

6. If you go and look at other tables, you can come back to this view by clicking “Q010Invoice” under “Views” in the left pane of the window. Whenever you create and save views, they will be listed here.



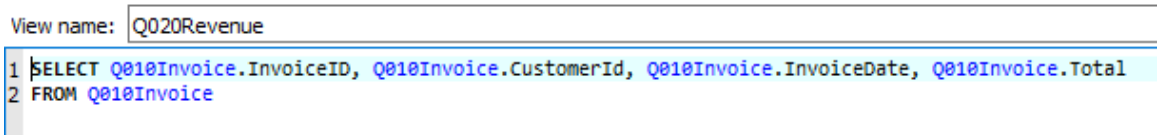
7. SQL is a non-procedural language. In a procedural language like Java or C, you open a file, read the records one at a time using a FOR loop and display or print them. In SQL, you specify what you want the result to look like and the looping is done for you automatically. With SQL, you end up

thinking about your data at a more abstract level and you don't have to be a programmer to do useful things with data.

Being selective with the *SELECT* query:

Being selective in what fields we view:

1. One of the powerful features of SQL is that you can base SELECT queries on other saved SELECT queries (views) and in this way build up your queries in simple steps. In other words, we can *build queries on queries*.
2. We will demonstrate this by being selective in what fields we view. Let's be selective about what fields we display in our view "Q010Invoice."
3. We want to exclude the *BillingAddress*, *BillingCity*, *BillingState*, *BillingCountry*, and *BillingPostalCode* fields from our new view because we feel that they aren't relevant to our analysis. In other words, we want to select only the following fields: *InvoiceId*, *CustomerId*, *InvoiceDate*, *Total*.
4. Create a new view called "Q020Revenue" with the following SQL code:



   Note that the general syntax used above is "Table/View"."Field". While this gives us the view that we want, there is a shortcut we could have used. Because we are selecting these fields from only one table, and thus there is no ambiguity*, we can drop the prefix, which specifies which Table/View we are pulling data from. In other words, we could have used the language "SELECT InvoiceId, CustomerId... FROM Invoice"

   **In the next recitation, you will create queries that pull data from multiple views and tables that have the same field names. Because of this ambiguity—SQLite won't know which table/view to retrieve the data from—you need to include this prefix.*
5. Edit "Q020Revenue" so that the prefixes are deleted from the field names.
   a. Select "Q020Revenue" in the left pane where all the table and view names are listed.
   b. In the toolbar, select "Edit view."

   c. Delete the prefixes from the field names.
6. View the results of your edited query. Your query should generate the same view as in Step 3.

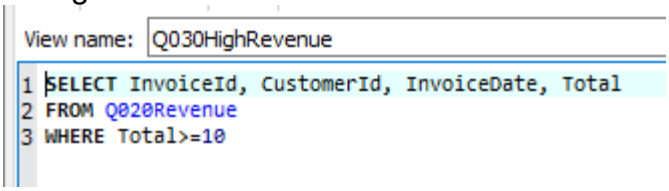
*Being selective in what records we view using the **WHERE** clause:*

Defn: The ***WHERE*** clause allows us to specify criteria that dictates which records to display. This clause must be written after the FROM clause.

1. Now, let's be selective in what records we retrieve. Let's look for orders that generated higher revenue, say at least $10. In order words, we want

to see records where the field “Total” has values greater than or equal to 10.

2. Create the following view.



3. How many of the invoices satisfy this criteria? Write this in your student answer sheet.
4. Furthermore, we want to see only orders with revenues that are at least $10 and were placed in the year 2013 or later. We want to specify a criterion for the *InvoiceDate* field. This is a bit tricky because SQL doesn’t recognize “1/1/2013” as a date. We will need to use **DATETIME(’2013-01-01 00:00:00’)** as our criterion format.
5. Create a new query Q040RecentHighRevenue in which only invoices that were created in 2013 that generated at least $10 in revenue are listed. *(Hint: build this query off of Q030HighRevenue).*
6. Write the WHERE clause you used in your student answer sheet.
7. How many orders satisfy both the criteria (created in 2013 or later, and greater than $10)? Write this in your student answer sheet.

Sorting data:

Defn*: The *ORDER BY clause allows records to be sorted by either descending or ascending order. This clause must come last in the SQL code. The syntax for this clause is:

ORDER BY FieldName ***DESC/ASC***

Note: ***DESC*** indicates descending order, ***ASC*** indicates ascending order.

1. We want to view the most recent from “Q040RecentHighRevenue” first.
2. Edit “Q040RecentHighRevenue” and write the resulting SQL code in your student answer sheet.

Using the *SELECT* query as a calculator:

Defn*: The *AS keyword allows the user to specify the name of a calculated field in the SELECT clause. The syntax for this clause is:

SELECT Field1, Field2, EXPRESSION ***AS*** FieldName

1. In addition to selecting fields, we can use the SELECT clause to calculate new fields based on the other fields in the field list.
2. We want to create a new query called “Q050RevenueByInvLine” based on the “InvoiceLine” table to calculate how much revenue we generated from each invoice line based on the price and the quantity of each track the customer purchased.
3. Start writing the SQL code by indicating that we want to include the following fields from “InvoiceLine”: *InvoiceLineId*, *InvoiceId*, *TrackId*, *UnitPrice*, *Quantity*.

4. Extend the list of fields by putting a comma "," after *Quantity* and typing "*UnitPrice*Quantity* AS *TotalRevenue*".
5. The full SELECT statement should be as follows:

View name: Q050RevenueByInvLine

```
SELECT InvoiceLineId, InvoiceId, TrackId, UnitPrice, Quantity, UnitPrice*Quantity AS TotalRevenue
FROM InvoiceLine
```

6. Look at the view generated by the query. Unfortunately, all the invoice lines required only a quantity of 1 of each TrackId, so this isn't the most interesting data to analyze.

Using other useful functions:

Defn: The ***CASE WHEN*** function is the IF-THEN-ELSE function in SQLite. The syntax for ***CASE WHEN*** is as follows:

CASE WHEN [condition] ***THEN*** [result if true] ***ELSE*** [result if false] ***END***

1. We want to categorize the revenues listed in "Invoice" as "High" if they are at least $10, "Medium" if they are at least $5 or "Low."
2. Let's first categorize only the "High" revenues. Create a new view "Q060RevenueClass" that lists the fields *InvoiceId*, *CustomerId*, *InvoiceDate*, *Total*, and a new field called *RevenueClass*, which will label our invoices as "High," "Medium," or "Low" using the CASE WHEN function.

View name: Q060RevenueClass

```
SELECT InvoiceId, CustomerId, InvoiceDate, Total,
CASE WHEN Total >= 10 THEN 'High' ELSE '' END AS RevenueClass
FROM Invoice
```

3. In order to label our "Medium" and "Low" revenues, we will need to use a nested CASE WHEN function. In the view "Q060RevenueClass," edit the ELSE statement in our CASE WHEN function as follows:

View name: Q060RevenueClass

```
SELECT InvoiceId, CustomerId, InvoiceDate, Total,
CASE WHEN Total >= 10 THEN 'High' ELSE CASE WHEN Total >= 5 THEN 'Medium' ELSE 'Low' END END AS RevenueClass
FROM Invoice
```

4. Create a query based on "Q060RevenueClass" that selects all fields but only records with *RevenueClass*='Medium.'
5. Write the SQL statement for this query in your student answer sheet.
6. How many 'Medium' invoices are there? Write this in your student answer sheet.

Defn: The ***IFNULL(x,y)*** function checks if there is missing data in field x and replaces it with default value y if the value is null.

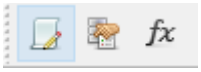
1. Create a query "Q070Customers" that checks for missing data in the field *Company* of the "Customer" table. If the record is null, replace it with "Student."

2. Write the SQL statement for this query in your student answer sheet.

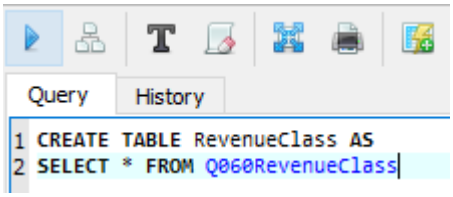
Using Executable Commands

Create a new table using *CREATE TABLE … AS*:

1. A query, without the results it generates, does not take up very much space, since it is only storing text. However, advanced queries on large datasets may take a long time to run and retrieve the resulting dataset. One way to speed things up is to store your results in a table, so that the query will not need to run every time you wish to view your results. This takes up more space than just storing the query, but it can be faster.
2. Let us now store the results of "Q060RevenueClass" into a table titled "RevenueClass" using an executable query.
3. Click the pencil & paper icon shown in the screen shot below to "Open SQL Query Editor".



4. Execute the following command by typing in te following SQL code and clicking on the "play button" icon outlined in the screen shot below:



5. Make sure that this table was created correctly and populated with the dataset resulting from the query "Q060RevenueClass". The table should have 412 rows.

Delete records with the *DELETE* command:

1. Let's delete all records from "Track" with a *Genre* of "Sci Fi & Fantasy", or a *GenreId* of 20.
2. View the contents of the "Track" table and note the number of rows.
3. Open the SQL Query Editor by clicking on the pencil & paper icon.
4. Execute the following SQL to view the number of rows you should delete:

```
SELECT * FROM Track WHERE GenreId=20
```

5. Now execute the following SQL to delete the appropriate rows:

```
DELETE FROM Track WHERE GenreId=20
```

6. Re-open the "Track" table and make sure it has 26 fewer rows.

Change individual records with the *UPDATE* command:

1. We can use SQL to update the records in a table.
2. Suppose we have decided that we do not like that there are two records in the "Playlist" table with name "Music". We would like to change the name of Playlist 8 to "Music 2" to avoid confusion.

3. Open the SQL Query Editor and execute the SQL in the screenshot below. Note this SQL ends in a semicolon. In many SQL-based software applications, you are required to end your commands with a semicolon. In SQLiteStudio you can include it if you want, as we have done in this screenshot, or you can drop it, as we have done in the rest of the recitation.

```
UPDATE Playlist SET Name="Music 2" WHERE
PlayListId=8;
```

4. Open the table “Playlist”. Verify that this change has been made correctly by viewing the data in grid view.

Review

1. Congratulations! You now know the basics of structured query language (SQL) as implemented by SQLiteStudio. You can now pick up and learn other database packages that use SQL with this basic knowledge.
2. At this stage, you are familiar with the following keywords:
    - ***SELECT***
    - ***FROM***
    - ***WHERE***
    - ***ORDER BY***
    - ***AS***
    - ***CASE WHEN…THEN…ELSE…END***
    - ***IFNULL***
    - ***CREATE***
    - ***DELETE***
    - ***UPDATE***
    - ***SET***