# STAT 19000 Project 10 Examples

In this series of examples we will learn some SQL basics by exploring the read-only sqlite3 database , chinook.db, online, or on scholar:

`/class/datamine/data/spring2020/chinook.db`

**Note:** This database is < 1mb, so don't hesitate to download it onto your own system.

The first step is to log in to scholar, and either connect to the database directly:

`/class/datamine/data/spring2020/chinook.db`

Or, copy and paste the database to a local directory:

`mkdir -p ~/project09 && cp /class/datamine/data/spring2020/chinook.db "$_"`

To connect directly, open a terminal and type the following:

`sqlite3 /class/datamine/data/spring2020/chinook.db`

To connect locally, open a terminal and type the following:

`cd ~/project09; sqlite3 chinook.db`

You should be presented with a simple prompt -- `sqlite>` -- and message with the SQLite version. Great!

---

```
-- First, let's toggle headers -- they are just too useful
.headers on

-- Recall the following from the examples from the previous project:

-- You can do:
SELECT MAX(Total) FROM invoices LIMIT(2);
/*MAX(Total)
25.86*/
-- BUT it still finds the maximum of the entire column, "Total"
-- AVG and SUM work as expected

SELECT AVG(Total) FROM invoices;
/*AVG(Total)
5.65194174757282*/

-- But again, AVGS the whole column
```

```sql
SELECT AVG(Total) FROM invoices LIMIT(2);
/*AVG(Total)
5.65194174757282*/

-- SUM, same behavior
SELECT SUM(Total) FROM invoices;
/*SUM(Total)
2328.6*/

SELECT SUM(Total) FROM invoices LIMIT(2);
/*SUM(Total)
2328.6*/

-- It is quite limited if this is truly all we can do with the functions like:
-- COUNT, AVG, MIN, MAX, SUM

-- Is there a way to perform these operations on subsets of rows? Yes.
-- Let's revisit the statements above:
SELECT MAX(Total) FROM invoices LIMIT(2);
SELECT AVG(Total) FROM invoices LIMIT(2);
SELECT SUM(Total) FROM invoices LIMIT(2);

-- This is really a situation where it seems like the user intends to get those
-- functions to return the summary values: MAX, AVG, and SUM for just the first
-- two rows. The most direct way is to use a subquery. A subquery is where we
-- use the result of one query to feed into another. For example:
SELECT * FROM invoices LIMIT(2);

-- The result of this query is a table with all of the original columns,
-- and only two rows from the original data. We can use our MAX function
-- on the result of this query (which only has two rows), to get what
-- we intended.

SELECT MAX(Total) FROM (SELECT * FROM invoices LIMIT(2));

-- This is called a subquery. The result of the "inside" query is used as our
-- base table for our FROM clause in the "outer" query. This is very useful,
-- and we will explore this more below.

-- With all of that being said, most of the time we won't want to perform an
operation
-- on something as arbitrary as the first X rows, but rather we will want to
-- summarize based on some conditions. This is where the GROUP BY clause comes in
handy:
-- https://www.sqlitetutorial.net/sqlite-group-by/
```

```sql
-- For example, let's get the AVG total for all invoices, but let's get that
total
-- based on the country:
SELECT BillingCountry, AVG(Total) FROM invoices GROUP BY BillingCountry;

-- That's pretty useful. Without GROUP BY, we could only do one country at
-- a time:
SELECT BillingCountry, AVG(Total) FROM invoices WHERE BillingCountry="Brazil";
SELECT BillingCountry, AVG(Total) FROM invoices WHERE BillingCountry="Germany";

-- And it would make for one very giant query if we wanted to use sub queries:
SELECT AVG(g.Total), AVG(b.Total) FROM (SELECT BillingCountry, Total FROM
invoices WHERE BillingCountry="Brazil") AS b, (SELECT BillingCountry, Total FROM
invoices WHERE BillingCountry="Germany") AS g;

-- Yes, GROUP BY is very very useful. In fact, it is even more useful. You can
-- GROUP BY multiple conditions:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices GROUP BY
BillingCountry, BillingState;

-- Very cool. What if we wan't to add conditions like we did before using WHERE?
-- We can still do this:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry IN ("USA", "Canada") GROUP BY BillingCountry, BillingState;

-- Great but what if we wanted only the states where the avg(total) > 5.4? You
may initially
-- think to do something like this:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry IN ("USA", "Canada") AND AVG(Total) > 5.4 GROUP BY BillingCountry,
BillingState;

-- But this won't work. How do you put a condition on a result of a GROUP BY or
some function?
-- This is where HAVING comes in:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry IN ("USA", "Canada") GROUP BY BillingCountry, BillingState HAVING
AVG(Total) > 5.4;

-- With HAVING you must also have GROUP BY. The following will not work:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices HAVING AVG(Total) >
5.4;

-- That doesn't even make any sense, as only a single value would be returns so
the
-- HAVING clause doesn't do anything.
```

```sql
-- WHERE is used to filter data prior to grouping, and HAVING is used to remove
rows after grouping.

-- What if you want to combine the result of two or more queries? There is a
special clause
-- that let's you do this called UNION and UNION ALL:
-- https://www.w3schools.com/sql/sql_union.asp

-- Let's look at the following two results:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry="USA" GROUP BY BillingCountry, BillingState HAVING AVG(Total) >
5.4;

SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry="Canada" GROUP BY BillingCountry, BillingState HAVING AVG(Total) >
5.4;

-- Using UNION we can combine:
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry="USA" GROUP BY BillingCountry, BillingState HAVING AVG(Total) >
5.4
UNION
SELECT BillingCountry, BillingState, AVG(Total) FROM invoices WHERE
BillingCountry="Canada" GROUP BY BillingCountry, BillingState HAVING AVG(Total) >
5.4;

-- Of course, in this instance the original query where we get the results for
both
-- countries makes more sense, but you can see what UNION does.

-- There are some rules listed at the top of this page that you must follow for
UNION to work: https://www.w3schools.com/sql/sql_union.asp

-- There is also UNION ALL which returns all results. If in the first UNION
example,
-- there happened to be two identical rows, one of the rows would have been
removed from
-- the result. UNION ALL, would keep any duplicate rows.

-- At this point there are two more topics to touch on here: wildcards + LIKE,
and joins.

-- We will start with wildcards + LIKE.
```

```sql
-- It's always best to lookup and google each system's wildcards, as they do
vary.
-- Here: https://www.w3schools.com/sql/sql_wildcards.asp is a list of the valid
-- sqlite3 wildcards. Look under the SQL Server list.

-- LIKE is the operator let's you search for a specified pattern in a column.

-- Let's say we wanted to get all album titles that don't start with Mozart:
-- as we aren't looking for his music.
SELECT * FROM albums WHERE Title NOT LIKE "Mozart%";

-- Great, but we don't want bach either:
SELECT * FROM albums WHERE Title NOT LIKE "Mozart%" AND Title NOT LIKE "Bach%";

-- Unfortunately default SQL doesn't have the full power of regex, and each
engine
-- has different levels of regex support. MySQL and postgresql both have either
full POSIX
-- regex support or something very close, and chances are if you are wanting to
do
-- more complex queries, your database will be something that supports it.

-- Finally, the last topic is one of the core utilities of databases: joins.
-- Joins are a way to combine rows from two or more tables based on a related
column.
-- You can find more examples: https://www.w3schools.com/sql/sql_join.asp

-- Depending on your engine, there could be many different types of joins. Many
of the
-- joins can be replicated using some core joins. For this reason, we will only
learn and
-- focus on: left join, and inner join.

-- Inner join combines data where both tables have matching values. Left join
combines
-- data where the first (or left) table is kept in its entirety and supplemented
-- where matching data is found in the second (or right) table. For example,
-- look at the playlists, playlist_track, and tracks tables. playlist is very
simple and
-- gives a playlistid and a name. tracks is also very simple and gives a trackid
and
-- some other info. Playlist_track, is a list of playlist id track id
combinations
-- that together make up a playlist.
```

```
-- To very explicitly demonstrate the differences between and INNER JOIN and LEFT
JOIN
-- we've added tables to the chinook.db database called: students, and advisors.
.tables

SELECT * FROM students;
SELECT * FROM advisors;

-- As you can see, students have advisors, and advisors have advisors as well.

-- If we wanted to get a list of all advisors and their matching students IF they
have
-- a matching student, we could use an INNER JOIN
SELECT * FROM advisors AS a INNER JOIN students AS s ON a.id=s.advisorid;

-- Every time there is a match between the advisor's id and the students'
advisorid,
-- a resulting row is output.

-- What if we wanted a list of all advisors, with matching students if they have
any.
-- In other words we want ALL advisors, and tack on student info if they have
any.
-- For this, a LEFT JOIN is appropriate.
SELECT * FROM advisors AS a LEFT JOIN students AS s ON a.id=s.advisorid;

-- As you can see, advisors without matching students are still listed. If we
were to swap
-- the position of the tables, the result would be different as we would now
want,
-- all students regardless of if they have an advisor.
SELECT * FROM students AS s LEFT JOIN advisors AS a ON a.id=s.advisorid;

-- As you can see, the result is the same as the INNER JOIN, because 100% of the
students
-- have a matching advisor!

-- You can even join a table with itself. In this case, advisors have advisors
too!
SELECT * FROM advisors AS a1 LEFT JOIN advisors AS a2 ON a1.id=a2.advisorid;

-- Here you can see that all advisors are listed regardless of if they advise an
advisor.
-- If we switched to an INNER JOIN, this would change:
SELECT * FROM advisors AS a1 INNER JOIN advisors AS a2 ON a1.id=a2.advisorid;
```

```
-- As you can see, any advisor who advises and advisor appears in this list, but
not advisors who don't advise advisors.

-- We can also perform multiple joins. What if we wanted all advisors who advise
at least 1 student and 1 advisor?
SELECT * FROM (SELECT * FROM advisors AS a INNER JOIN advisors AS ads ON
a.id=ads.advisorid) as sub INNER JOIN students AS s ON sub.id=s.advisorid;

-- Let's create a new table by combining the playlist table and the
playlist_track table.
-- Playlist_track is the longer of the two tables, we just want to supplement
that table
-- With the name of the playlist. We can do this with a left join:
SELECT * FROM playlist_track AS pt LEFT JOIN playlists as p ON
p.PlaylistId=pt.PlaylistId;

-- There are 8715 resulting rows. In this case, every row in playlist_track had a
matching playlist in the playlists table (based on playlistId):
SELECT DISTINCT playlistid FROM playlists;
SELECT DISTINCT playlistid FROM playlist_tracks;

-- BUT, you can see that the reverse is not true. Every playlistid from the
playlists
-- does NOT have matching info from the playlist_track table. There is no track
-- information for playlists with playlistid's: 2, 4, 6, & 7. What this means is
-- if we reversed the positions of the tables in the LEFT JOIN, there would be 4
-- more results as every item in the LEFT table is kept regardless of if there is
-- a match in the right table.
SELECT COUNT(*) FROM playlists AS p LEFT JOIN playlist_track as pt ON
p.PlaylistId=pt.PlaylistId;

-- If we took the above query and made it an inner join, we would go back to
getting our original result as an inner join requires every row to have a match:
SELECT COUNT(*) FROM playlists AS p INNER JOIN playlist_track as pt ON
p.PlaylistId=pt.PlaylistId;

-- This allows us to combine and produce some cool datasets quickly. In addition,
this
-- method of having a table with two id columns (playlist_track), allows us to
reduce
-- the amount of storage used as we don't have to re-store the track's title
every time that
-- track is in a playlist. Very cool! The playlist_track table also exemplifies
what is
-- called a many-to-many relationship. What this means is that a playlist has or
can
```

```
-- have many tracks, and a track can be a part of many playlists.

-- In real life you will want to be mindful when joining and querying a many-to-
many
-- relationship. Depending on what you are doing, the amount of memory and
computational
-- resources to complete your operation can blow up very quick! For example,
-- let's say you have a table with 5 rows, but > 1000 columns of data. You have
another
-- table with billions of rows. Lastly, you have another table that expresses the
-- many to many relationship. If you don't think about what you are doing,
-- you could completely drain the system of resources with a bad query!

-- Another concept to talk about here is foreign keys. The playlist_track table
has
-- two columns, both columns contain foreign keys to other tables. Those id's are
-- "keys" for us to match rows of data from another table, ON that table's
primary key.
-- Tables will have a primary key column that contains a unique number
identifying that
-- row of data. These "key" columns are (usually) how tables are combined.

-- You can also join many tables:
SELECT * FROM playlist_track AS pt
LEFT JOIN playlists as p ON p.PlaylistId=pt.PlaylistId
LEFT JOIN tracks as t ON t.TrackId=pt.TrackId;
```