STAT 19000 Project 9 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.

Relational databases and SQL

Relational databases are digital databases that store data into tables of columns and rows where a unique key identifies each row.

Typically, users use SQL (Structured Query Language) for managing (adding, deleting, updating, etc.) data inside the database.

SQLite

SQLite is a database engine that allows us to connect to the database and use SQL to manage the data. There are a variety of database engines, SQLite is one brand that happens to have a low learning curve when it comes to database-engine-specific tooling. It's simple to set up, connect to a database, and use.

From https://sqlite.org:

> SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day.

The "self-contained" feature is particularly useful! From https://sqlite.org:

> SQLite is "stand-alone" or "self-contained" in the sense that it has very few dependencies. It runs on any operating system, even stripped-down bare-bones embedded operating systems. SQLite uses no external libraries or interfaces (other than a few standard C-library calls described below). The entire SQLite library is encapsulated in a single source code file that requires no special facilities or tools to build.

Normally, a database engine has it's own set of special dependencies that need to be properly installed and managed, making the time to get "up-and-running" higher.

In addition, the database itself (for sqlite), is typically stored in a single `.db` file, making it very convenient to transport, locate, and open.

SQLite does have its disadvantages, mainly its inability to have multiple simultaneous writers to the database. With that being said, we will only be reading from the database, for now, so it suits our needs well. Plus, other than database-engine unique syntax, and a few omitted SQL features, the syntax is the same as any other relational database.

Enough is enough, let's get started.

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 and message with the SQLite version. Great!

---

w3schools.com has a good introduction to SQL, including some exercises. For now, we are going to show some examples of the following SQL statements: `SELECT`, `FROM`, `WHERE`, `AS`, `SELECT DISTINCT`, `AND`, `OR`, `NOT`, `ORDER BY`, `LIMIT`, `MIN`, `MAX`, `COUNT`, `AVG`, `SUM`, `OFFSET`, `DESC`, `ASC` and `BETWEEN`. Additionally, we are going to show you how to make comments like we do in R using `#`.

```
1  -- Let's start by talking about comments. This line is a single-line
   comment,
2  -- and so is this one, because they start with two dashes "--".
3
4  /* A more appropriate solution when writing multiple lines is to use the
5  multi-line comments. Multi-line comments start with "/*" and end with
6  the two characters reversed, so "*" and then "/". Anything in between these
   sets of symbols is recognized by sqlite as a comment, great! */
7
8  -- All of the commands given here are to be run in the sqlite prompt.
9  -- Once you've started the sqlite prompt you should be able to see
10 -- something like:
11 --  sqlite>
12
13 /* We've loaded the chinook database. Databases can be pretty complicated,
   but, in general, when dealing with a relational database, you really need
   to think about tables and indexes.
14
15 Think of tables like an excel spreadsheet. The spreadsheet has a header row
   with the names of the columns, and each remaining row contains one datum
   per column that conforms to a certain type (i.e. in a spreadsheet, one
   column may have floating point integers, another may have integers, another
   may have strings, etc.). In the same way, a table has columns (or fields),
   with headers. Tables have rows of data that conform to a certain, pre-
   defined data type.
16
```

```
Great. Now what about indexes. Think of an index just like an index you
would find in a text book. Those last 10-20 pages of a 800 page book that
allows you to find some term quickly (because it's in alphabetical order),
and immediately flip to the page in the book where the topic is discussed.
If you didn't have the index you'd have to flip through every page to find
what you are looking for. The same concept applies to databases. Sometimes,
if you have a table with large amounts of data, creating an index on a
certain column, or combination of columns will greatly increase query
performance. Great, so why don't we create indexes for everything? Indexes
take up storage space, if the table is small an index won't necessarily
speed things up, if the table gets updated frequently, the indexes will
need updated too, which takes time.

Great, now let's actually do something. */

-- What are the tables within the database
.tables
/*albums          employees       invoices        playlists
artists         genres          media_types     tracks
customers       invoice_items   playlist_track*/

-- Are there any indexes?
.indices

/*IFK_AlbumArtistId                 IFK_PlaylistTrackTrackId
IFK_CustomerSupportRepId          IFK_TrackAlbumId
IFK_EmployeeReportsTo             IFK_TrackGenreId
IFK_InvoiceCustomerId             IFK_TrackMediaTypeId
IFK_InvoiceLineInvoiceId          sqlite_autoindex_playlist_track_1
IFK_InvoiceLineTrackId*/

-- Let's turn on headers so it is easy to tell the data from our queries
.headers on

/* Let's test it out by selecting all of the data, every column (*), from
the first employee only. */
SELECT * FROM employees LIMIT(1);
/*EmployeeId|LastName|FirstName|Title|ReportsTo|BirthDate|HireDate|Address
|City|State|Country|PostalCode|Phone|Fax|Email
1|Adams|Andrew|General Manager||1962-02-18 00:00:00|2002-08-14
00:00:00|11120 Jasper Ave NW|Edmonton|AB|Canada|T5K 2N1|+1 (780) 428-
9482|+1 (780) 428-3457|andrew@chinookcorp.com */

-- Great, we can easily see the headers: EmployeeId, LastName, Title, etc.

-- Let's break this query down into parts:

-- The SELECT command tells sqlite that we want to get (or "select") data
from the database.
-- The * immediately following the SELECT tells sqlite to select all of the
columns.

-- Instead of * you could do:
SELECT FirstName, LastName FROM employees LIMIT(1);
/*FirstName|LastName
Andrew|Adams*/
```

```sql
-- Note, that this will also work as column names and sql commands are not case sensitive.
select firstname, lastname from employees limit(1);

-- With that being said, capitalizing the commands makes things easier to read.
-- The FROM command tells sqlite from what table do we want to get every column of
-- information?
-- The "employees" part following FROM is the table.
-- LIMIT(1) tells sqlite to limit the output to a single row.
-- LIMIT(2) would limit the output to two rows, etc.
-- Lastly, the semi-colon tells sqlite that this is the end of the statement.

-- Let's try that LIMIT(2) thing out...
SELECT FirstName, LastName FROM employees LIMIT(2);
/*FirstName|LastName
Andrew|Adams
Nancy|Edwards*/

-- Great, and if we look they are indeed the first two:
SELECT FirstName, LastName FROM employees;
/*FirstName|LastName
Andrew|Adams
Nancy|Edwards
Jane|Peacock
Margaret|Park
Steve|Johnson
Michael|Mitchell
Robert|King
Laura|Callahan*/

-- What if we wanted the LastName and then the FirstName?
SELECT LastName, FirstName FROM employees;
/*LastName|FirstName
Adams|Andrew
Edwards|Nancy
Peacock|Jane
Park|Margaret
Johnson|Steve
Mitchell|Michael
King|Robert
Callahan|Laura*/

-- Cool, so you just put them in whatever order you want them.
-- What if we, for some reason wanted something repeated like James, James Bond
SELECT FirstName, FirstName, LastName FROM employees;
/*FirstName|FirstName|LastName
Andrew|Andrew|Adams
Nancy|Nancy|Edwards
Jane|Jane|Peacock
Margaret|Margaret|Park
Steve|Steve|Johnson
Michael|Michael|Mitchell
Robert|Robert|King
Laura|Laura|Callahan*/
```

```
110
111 -- Nice.
112
113 -- What if instead of the first two names, we wanted to get the first two
    names
114 -- that occur after the 5th name?
115 SELECT FirstName, LastName FROM employees LIMIT(2) OFFSET(5);
116 /*FirstName|LastName
117 Michael|Mitchell
118 Robert|King*/
119
120 -- What if we wanted the last two names? Well, for this database we could
    do:
121 SELECT FirstName, LastName FROM employees LIMIT(2) OFFSET(6);
122
123 -- BUT, as soon as the database adds a new row, this won't work.
124 -- So, first of all, how is the order curently determined? It's a
125 -- small table, so let's look:
126 SELECT * FROM employees;
127 /*EmployeeId|LastName|FirstName|Title|ReportsTo|BirthDate|HireDate|Address
    |City|State|Country|PostalCode|Phone|Fax|Email
128 1|Adams|Andrew|General Manager||1962-02-18 00:00:00|2002-08-14
    00:00:00|11120 Jasper Ave NW|Edmonton|AB|Canada|T5K 2N1|+1 (780) 428-
    9482|+1 (780) 428-3457|andrew@chinookcorp.com
129 2|Edwards|Nancy|Sales Manager|1|1958-12-08 00:00:00|2002-05-01 00:00:00|825
    8 Ave SW|Calgary|AB|Canada|T2P 2T3|+1 (403) 262-3443|+1 (403) 262-
    3322|nancy@chinookcorp.com
130 3|Peacock|Jane|Sales Support Agent|2|1973-08-29 00:00:00|2002-04-01
    00:00:00|1111 6 Ave SW|Calgary|AB|Canada|T2P 5M5|+1 (403) 262-3443|+1 (403)
    262-6712|jane@chinookcorp.com
131 4|Park|Margaret|Sales Support Agent|2|1947-09-19 00:00:00|2003-05-03
    00:00:00|683 10 Street SW|Calgary|AB|Canada|T2P 5G3|+1 (403) 263-4423|+1
    (403) 263-4289|margaret@chinookcorp.com
132 5|Johnson|Steve|Sales Support Agent|2|1965-03-03 00:00:00|2003-10-17
    00:00:00|7727B 41 Ave|Calgary|AB|Canada|T3B 1Y7|1 (780) 836-9987|1 (780)
    836-9543|steve@chinookcorp.com
133 6|Mitchell|Michael|IT Manager|1|1973-07-01 00:00:00|2003-10-17
    00:00:00|5827 Bowness Road NW|Calgary|AB|Canada|T3B 0C5|+1 (403) 246-
    9887|+1 (403) 246-9899|michael@chinookcorp.com
134 7|King|Robert|IT Staff|6|1970-05-29 00:00:00|2004-01-02 00:00:00|590
    Columbia Boulevard West|Lethbridge|AB|Canada|T1K 5N8|+1 (403) 456-9986|+1
    (403) 456-8485|robert@chinookcorp.com
135 8|Callahan|Laura|IT Staff|6|1968-01-09 00:00:00|2004-03-04 00:00:00|923 7
    ST NW|Lethbridge|AB|Canada|T1H 1Y8|+1 (403) 467-3351|+1 (403) 467-
    8772|laura@chinookcorp.com*/
136
137 -- Clearly, the table is ordered by EmployeeId. We will dig into that a
    little bit later.
138 -- So, one way to do this would be to use DESC and ORDER BY
139 SELECT EmployeeId, FirstName, LastName FROM employees ORDER BY EmployeeId
    DESC;
140 /*FirstName|LastName
141 Laura|Callahan
142 Robert|King
143 Michael|Mitchell
144 Steve|Johnson
145 Margaret|Park
146 Jane|Peacock
```

```
Nancy|Edwards
Andrew|Adams*/

-- Great, just add the LIMIT back now
SELECT FirstName, LastName FROM employees ORDER BY EmployeeId DESC
LIMIT(2);
/*FirstName|LastName
Laura|Callahan
Robert|King*/

-- By default, ORDER BY goes in ascending order, but you can specify using
ASC.
-- You can also sort by other columns:
SELECT FirstName FROM employees ORDER BY FirstName ASC;
/*FirstName
Andrew
Jane
Laura
Margaret
Michael
Nancy
Robert
Steve*/

-- Let's look at the Title column
SELECT Title FROM employees;
/*Title
General Manager
Sales Manager
Sales Support Agent
Sales Support Agent
Sales Support Agent
IT Manager
IT Staff
IT Staff*/

/* This result is a little bit redundant. Imagine if we just wanted a list
of titles but there were 10000 employees! This is where SELECT DISTINCT can
be useful. SELECT DISTINCT returns only unique results or unique
combinations depending on what column(s) you SELECT DISTINCT.*/
SELECT DISTINCT Title FROM employees;
/*Title
General Manager
Sales Manager
Sales Support Agent
IT Manager
IT Staff*/

-- But if you wanted FirstName/Title combos (of which all are unique),
you'd get:
SELECT DISTINCT FirstName, Title FROM employees;
/*FirstName|Title
Andrew|General Manager
Nancy|Sales Manager
Jane|Sales Support Agent
Margaret|Sales Support Agent
Steve|Sales Support Agent
Michael|IT Manager
```

```
Robert|IT Staff
Laura|IT Staff*/

-- SQL also has some built in functions including but not limited to: MIN, MAX, COUNT, SUM, AVG
SELECT MIN(Total) FROM invoices;
/*MIN(Total)
0.99*/

-- As you can see the invoice table has what appears to be a variety of different types.
-- Let's check them out.
.schema invoices
/*
CREATE TABLE IF NOT EXISTS "invoices"
(
    [InvoiceId] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    [CustomerId] INTEGER  NOT NULL,
    [InvoiceDate] DATETIME  NOT NULL,
    [BillingAddress] NVARCHAR(70),
    [BillingCity] NVARCHAR(40),
    [BillingState] NVARCHAR(40),
    [BillingCountry] NVARCHAR(40),
    [BillingPostalCode] NVARCHAR(10),
    [Total] NUMERIC(10,2)  NOT NULL,
    FOREIGN KEY ([CustomerId]) REFERENCES "customers" ([CustomerId])
        ON DELETE NO ACTION ON UPDATE NO ACTION
);
CREATE INDEX [IFK_InvoiceCustomerId] ON "invoices" ([CustomerId]);
*/

/* Interesting! This is the SQL code that is used to build the invoices table. This is a lot to take in, but for now, the main takeaway should be that each column has a type explicitly declared.

For example, InvoiceId is an INTEGER, InvoiceDate is a DATETIME, many of these are NVARCHAR(X) where X is the length of the NVARCHAR, and Total is NUMERIC. If interested, you can read about SQLite types here: https://www.sqlite.org/datatype3.htmls

For now, ignore the rest, but note that using .schema can give you a pretty good idea what the data will look like.
*/

SELECT MAX(Total) FROM invoices;
/*MAX(Total)
25.86*/

-- You can still use LIMIT...
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
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*/

-- COUNT is useful too, and returns a count of the number of rows when
-- COUNT(*), otherwise if it is COUNT(column_header), it returns the count
-- of non null values.

SELECT COUNT(*) FROM invoices;
/*COUNT(*)
412*/

-- What if we want to add a condition to our queries. For example we want
only invoices
-- for Brazil? Well, the WHERE clause let's us do this!
SELECT * FROM invoices WHERE BillingCountry="Brazil";

-- And it can be combined with function too, finding the total for Brazil
invoices
-- is simple then:
SELECT SUM(Total) FROM invoices WHERE BillingCountry="Brazil";
/*SUM(Total)
190.1*/

-- And of course, you can find the average and sum at the same time:
SELECT AVG(Total), SUM(Total) FROM invoices WHERE BillingCountry="Brazil";
/*AVG(Total)|SUM(Total)
5.43142857142857|190.1*/

-- BETWEEN can be useful as well, and selects the rows that are between two
values,
-- inclusively (which means if we want numbers between 1-10 we would
-- include 1 and 10 in the list).

-- Let's get invoices that total between 13-14.
SELECT * FROM invoices WHERE Total BETWEEN 13 AND 14;

-- BETWEEN works on numbers, dates, and strings.

-- Date example: get invoices between 1/1/2013 and 2/1/2013
SELECT * FROM invoices WHERE InvoiceDate BETWEEN '2013-1-1' AND '2013-2-
1';

-- Uh oh! This is wrong. Why? Becuase you need to zero pad your month and
day values:
```

```sql
SELECT * FROM invoices WHERE InvoiceDate BETWEEN '2013-01-01' AND '2013-02-01';

-- Great! How many of those were in the UK?
SELECT * FROM invoices WHERE InvoiceDate BETWEEN '2013-01-01' AND '2013-02-01' AND BillingCountry="United Kingdom";
/*335|53|2013-01-15 00:00:00|113 Lupus St|London||United Kingdom|SW1V 3EN|0.99
336|54|2013-01-28 00:00:00|110 Raeburn Pl|Edinburgh ||United Kingdom|EH4 1HH|1.98*/

-- Excellent! Although not explicitly mentioned earlier, AND, OR, and NOT function just like
-- normal logical operators in any language. You can use parenthesis to force order of
-- operations as well.

SELECT * FROM invoices WHERE Total > 20 OR Total < 5 AND BillingCountry="Brazil";

-- Uh oh! We wanted invoices where the Total was either greater than 20 or less
-- than 5 and from Brazil. Due to the fact that AND has precedence over OR (in the
-- same way multiplication has precedence over addition and subtraction),
-- what this statement did was equivalent to:
SELECT * FROM invoices WHERE Total > 20 OR (Total < 5 AND BillingCountry="Brazil");

-- To fix, add parenthesis:
SELECT * FROM invoices WHERE (Total > 20 OR Total < 5) AND BillingCountry="Brazil";

-- Now the last topic we will touch on is called aliasing (using AS). I'd recommend reading:
-- https://www.w3schools.com/sql/sql_alias.asp

-- You can alias a column...
SELECT BillingCountry AS Country FROM invoices LIMIT(1);
/*Country
Germany*/

-- or multiple columns
SELECT BillingCountry AS Country, Total AS DollarValue FROM invoices LIMIT(1);
/*Country|DollarValue
Germany|1.98*/

-- You can alias a table, and access that tables indiviual values
SELECT inv.BillingCountry FROM invoices AS inv LIMIT(1);
/*BillingCountry
Germany*/

-- Which allows you to query multiple tables at once
SELECT i.BillingCountry, c.FirstName, c.LastName FROM invoices AS i, customers AS c WHERE i.CustomerId = c.CustomerId;

```

```
-- That is pretty cool. We printed the invoices billing country, and customer first and last
-- name every place that the customer id matches in the tables.

-- You can also "combine" columns into a single string. Note that the syntax varies
-- depending on the database engine. In sqlite, we use || to concatenate.
SELECT InvoiceId, BillingCity || ', ' || BillingCountry AS Place FROM invoices;

-- If the alias name contains space you need to use ""
SELECT InvoiceId AS "Invoice ID" FROM invoices LIMIT(1);
/*Invoice ID
98*/

/* A cool feature of sqlite is it is really easy to export a query result into some file using the .mode and .once dot commands.*/

-- Check what mode we are currently in:
.mode

-- Export entire invoices table to a csv
.mode csv
.output invoices.csv
SELECT * FROM invoices;

-- To go back to using | separator
.separator |
.mode list

-- Very cool! By defaul the .mode is "list". Some other useful modes include "tabs"
-- and "html". This is a very reasonable way to go from sqlite database -> analyzing a
-- table in R.

-- Feel free to explore SQL and sqlite3 features in more depth!
```

You can see a full list of dot commands by typing `.help`:

```
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail ON|OFF           Stop after hitting an error.  Default OFF
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
.explain ?ON|OFF?      Turn output mode suitable for EXPLAIN on or off.
                         With no args, it turns EXPLAIN on.
.header(s) ON|OFF      Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indices ?TABLE?       Show names of all indices
                         If TABLE specified, only show indices for tables
```

```
                          matching LIKE pattern TABLE.
.load FILE ?ENTRY?     Load an extension library
.log FILE|off          Turn logging on or off.  FILE can be stderr/stdout
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                          csv      Comma-separated values
                          column   Left-aligned columns.  (See .width)
                          html     HTML <table> code
                          insert   SQL insert statements for TABLE
                          line     One value per line
                          list     Values delimited by .separator string
                          tabs     Tab-separated values
                          tcl      TCL list elements
.nullvalue STRING      Use STRING in place of NULL values
.output FILENAME       Send output to FILENAME
.output stdout         Send output to the screen
.print STRING...       Print literal STRING
.prompt MAIN CONTINUE  Replace the standard prompts
.quit                  Exit this program
.read FILENAME         Execute SQL in FILENAME
.restore ?DB? FILE     Restore content of DB (default "main") from FILE
.schema ?TABLE?        Show the CREATE statements
                          If TABLE specified, only show tables matching
                          LIKE pattern TABLE.
.separator STRING      Change separator used by output mode and .import
.show                  Show the current values for various settings
.stats ON|OFF          Turn stats on or off
.tables ?TABLE?        List names of tables
                          If TABLE specified, only list tables matching
                          LIKE pattern TABLE.
.timeout MS            Try opening locked tables for MS milliseconds
.trace FILE|off        Output each SQL statement as it is run
.vfsname ?AUX?         Print the name of the VFS stack
.width NUM1 NUM2 ...   Set column widths for "column" mode
.timer ON|OFF          Turn the CPU timer measurement on or off
```