Using the SQLite Manager for Firefox

By MaryJo Webster

@MaryJoWebster

[Mjwebster71@gmail.com](mailto:Mjwebster71@gmail.com)

Created: March 2015

Download and install Firefox

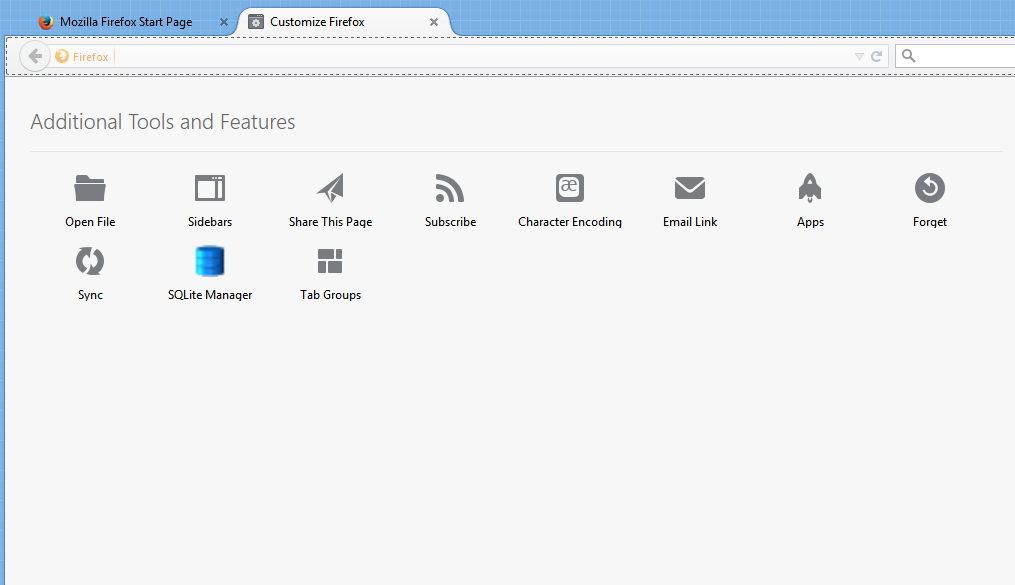
While in Firefox, go to this site to get the SQLite Manager (click “Download Now” button):  
<https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/>

Click “install now.” You will need to restart Firefox.

**If using Mac OS….**

Go to the Tools menu and you should find SQLite Manager (then jump to bottom of this page where you create a new database)

**If using Windows…..**

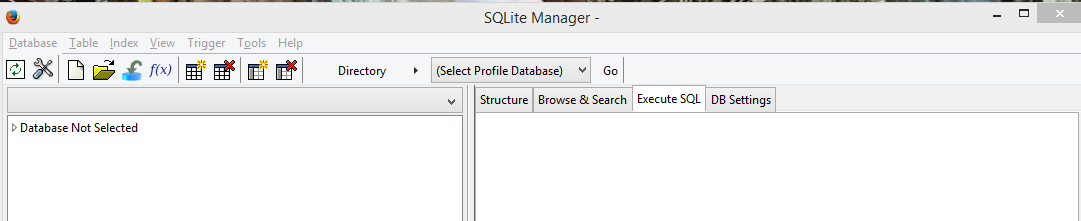
Right-mouse click along the top of Firefox and choose “Customize”

In this box that comes up (like the one in the picture), you’ll find “SQLite Manager”

Drag that icon into your toolbar area to permanently position it on the toolbar.

Then push the “Exit customize” button (green) in the lower right

Launch SQLite Manager (using that button you just installed in the toolbar)



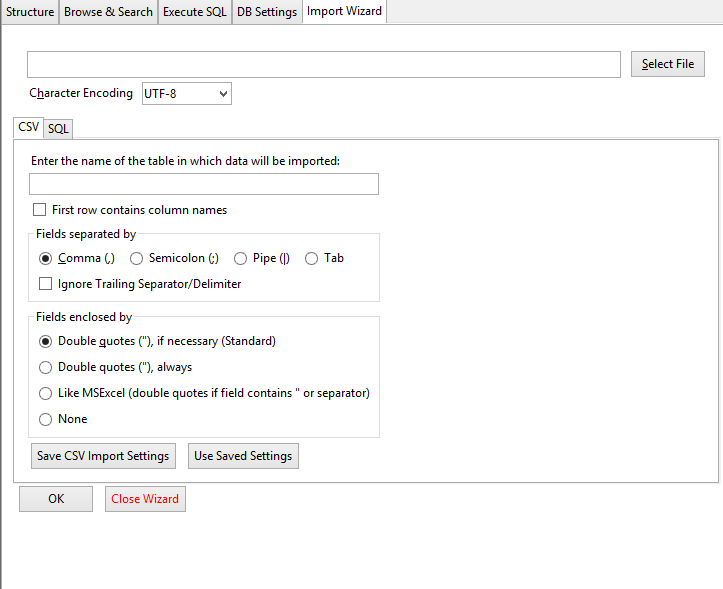
Use the button that looks like a blank piece of paper to create a new database

In the dialog box that pops up, give your database a name. In this case, we’ll call it “Bridges”

That will launch the Import Wizard tab on the right side of the screen.

Note: SQLite can only import text files. So if you have your data in Excel or some other format, you first need to export the data as a delimited text file

Click the “Select File” button to go get your text file



Then fill out the bottom portion :  
1) Give your table a name (the default will be the name of your text file)  
2) Indicate whether or not the first row contains column names  
3) Indicate the delimiter (comma, semi-colon, etc)  
4) Indicate if there are text qualifiers (where it says “fields enclosed by”)

You can save these settings, if you want.

**Then push OK**

It will then ask if you want to “modify” the table. This gives you a chance to set the field properties – whether the field should be text, date, number, etc. YES, YOU DO!!

Here are the datatype options that come up (stars in front of the ones you’ll use most often):

(\*)INTEGER: Used for small numbers (less than 10)  
BOOL: Boolean – 0 for no, 1 for yes

(\*) DOUBLE: number with 2 decimal places

FLOAT: automatically assigns decimal places; use this if you have numbers with more than 2 digits on the right side of the decimal point (highly unlikely that you’d use this one)

REAL: The value is a floating point value

CHAR: Small text field

TEXT: Very large text field

(\*) VARCHAR: This is the one to use for most text fields – it’s “variable” to accommodate different sizes of text fields (but can’t exceed 255 bytes)

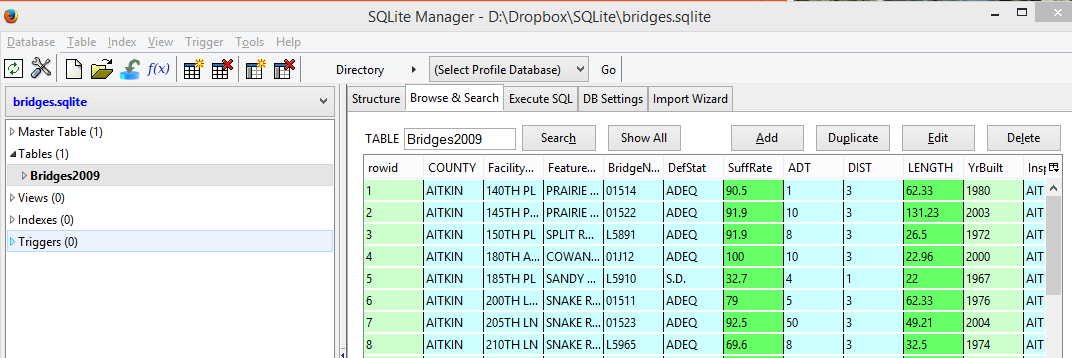
BLOB: Stores the data exactly as it came in. No format

(\*) NUMERIC: Field you’d want to use for most numbers (unless there are decimal points)

(\*) DATETIME: use for dates or date/time fields

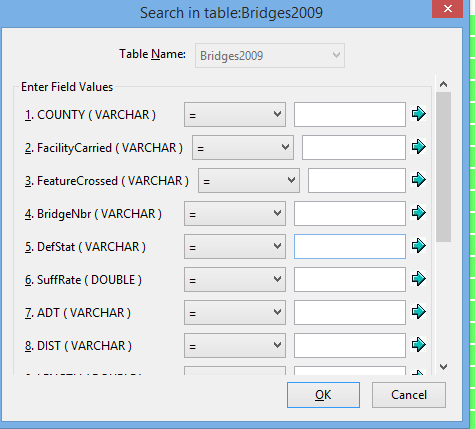
Then it will say “are you sure you want to import?” Say Ok and it will create your table.

You’ll see your table listed under “Tables” in the left-side navigation pane.



Now look at the tabs in the right pane – Structure, Browse&Search, Execute SQL, DB Settings, Import Wizard.

That’s where the real work gets done. We’ve already used the Import Wizard one.

Click on Browse & Search and you’ll see the first 100 rows of your data (like in the picture above).   
  
You can push the Search button to pull up a sort of Filter option.

NOTE: SQLite is quite picky in both this search and in a SQL query when looking for something. **It is case sensitive.**

For example, if we want to find all the bridges in Ramsey county, we have to put “RAMSEY” in all capital letters in the search box, because in the data the name of the county is in all capital letters.

RECOMMENDATION: Before bringing your data into SQLite, run it through OpenRefine and uppercase all of the text fields.

After using the search, push the “Show All” button to undo the filter.

The “Add” button allows you to add a new row.

If you select a row, you can push “Edit” to edit any of the content in that row. Or you can push “duplicate” to make a new row that is a copy of the one you selected. Or you can push “delete” to get rid of a row permanently.

Under the “Structure” tab, you can see the SQL used to create your table and the list of fields and their datatypes (essentially your record layout)

The “Execute SQL” tab is where we’re going to do our analysis work.

We’re going to use Structured Query Language (SQL), which is used by pretty much every database manager program out there. The syntax is sometimes a little different from one to another. SQLite seems to mimic SQL Server and MySQL, two very common programs.

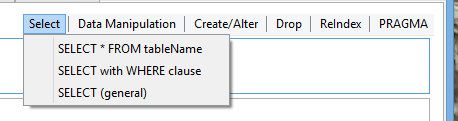
About SQL:

SQL is essentially English boiled down into keywords. You need a minimum of 2 lines of SQL to run a basic query and could have a maximum of 5 or 6 lines. Here are the primary SQL codes we’re going to use for running a query:

SELECT  
FROM  
WHERE  
GROUP BY  
ORDER BY

Here’s a good SQL tutorial: <http://databases.about.com/od/sql/a/sql.htm>

The main thing to remember right now is that the computer requires you to be precise – just like with formulas in Excel. If you type something just a little bit wrong, you’ll either get an error message or get the wrong answer.

One of the nifty things about the SQLite Manager is that it has some “cheaters” for you. In the upper right when you have Execute SQL open, you’ll see a few tabs, starting with “Select.” If you pull one of them down, you’ll see potential query syntax that you can use to get started.

-------------------------------------------

The data we are working with contains one record for each bridge in Minnesota that is 20 feet or longer in length. These are the ones that bridge inspectors focus on. This data was obtained from the Minnesota Department of Transportation in June 2009 and reflects the inspection status of bridges as of that date. The data doesn’t include the inspection date or have any history beyond this one year, so it doesn’t let us look at trends over time or determine whether inspectors are looking at these bridges frequently enough. But it does provide good insight into the current condition of bridges in the state.

The DOT sends this information, along with more detailed fields and data on smaller bridges, to the federal Department of Transportation which compiles the National Bridge Inventory database (available from the NICAR Database Library)

Here is the Record Layout:

|  |  |
| --- | --- |
| COUNTY | County where bridge is located |
| FacilityCarried | Road or railroad that runs on top of the bridge |
| FeatureCrossed | Road, river, lake, etc that the bridge goes over |
| BridgeNbr | Bridge number |
| DefStat | Deficiency status (see codes) |
| SuffRate | Sufficiency rating |
| ADT | Average daily traffic |
| DIST | DOT District it's located in |
| LENGTH | Length (in feet) |
| YrBuilt | Year constructed |
| InspectAgency | Agency responsible for inspections |
| OWNER | Bridge owner |

Here are some codes you will need to know:

DefStat:

ADEQ = adequate

S.D. = structurally deficient. This means it needs to be replaced due to safety concerns. See “Suffrate” field for more detail about how serious this is.

F.O. = functionally obsolete --essentially that the design of the bridge is not appropriate for the current traffic needs or that there is a flaw in the design that causes traffic problems

SuffRate:

This is a number from 0 to 100 that represents the overall rating for the bridge that was calculated using a complex mathematical formula. Anything less than 50% is eligible for replacement funds. Anything between 50% and 80% is eligible for rehabilitation funds only. The fact that a bridge is eligible for funds doesn’t mean the funds will be available immediately. This rating just gets them on the list for future funding.

Owner:

TWP= township

COUNTY = county where it is located

OTHER = other entity (might be privately owned)

STATE = state government

---------------------------------------------------------------------------------------------

Let’s get back to SQL. There are 2 lines of SQL that are required for every query – SELECT and FROM

SELECT tells the program which fields to show in your answer

FROM tells the program what table to pull from (of course, this is most crucial when you have a database that has multiple tables)

So a very basic SQL query could look like this:

SELECT \*  
FROM Bridges2009

The asterisk (\*) is simply a wildcard that, in this case, means return all the fields.

We can get a little more specific with a query like this:

SELECT county, defstat, suffrate  
FROM Bridges2009

This one tells the program to only display three fields in the answer. You can put the fields in whatever order you like (it doesn’t have to match the order they appear in the original table)

But that doesn’t tell us much about the data, does it?

So we need to start using some of the optional parts of SQL to get a little deeper.

The ORDER BY line allows us to sort our answer. So let’s add that to our last query, sorting it so that the bridge with the highest suffrate is at the top. ORDER BY is always the last line of SQL.

SELECT county, defstat, suffrate  
FROM Bridges2009  
ORDER BY suffrate desc

There are two ways to write the ORDER BY portion of the query. In the example above, I named the field I wanted and then put “desc” for descending order. (the default is ascending, so you would just leave it as “ORDER BY suffrate” if you want ascending; or you can tack on “ASC”)

The second way is to reference the column number. This means the column number in your answer. So in the above query we have 3 columns in our answer – county, defstat and suffrate. The field we want to sort by is the 3rd column. So we could write it like this and get the same result:

SELECT county, defstat, suffrate  
FROM Bridges2009  
ORDER BY 3 desc

Let’s get a little fancier. I want to see the name of the bridge and the feature that it crosses (usually it’s a river, but not always)

SELECT county, facilitycarried, featurecrossed, defstat, suffrate

FROM Bridges2009

ORDER BY 5 desc

(NOTE: I had to change the order by to “5” because I added fields to my SELECT line)

Now let’s find all the bridges that are in Hennepin County.

For that, we need to add another line of SQL – the WHERE line. Think of this as your filter. This is how you tell SQLite that you want to limit your answer to only certain records that match whatever criteria you put in the WHERE line.

WHERE goes right after the FROM line.

Also, this is an important note: SQLite is case sensitive. So, in this case, the county field in our data has the county names in all uppercase. So we need to write it that way in our query (other database managers are not case sensitive)

SELECT county, facilitycarried, featurecrossed, defstat, suffrate

FROM Bridges2009

WHERE county = “HENNEPIN”

ORDER BY 5 desc

Note that I put double quote marks around the county name (“HENNEPIN”). SQLite will also work if you use single quotes. I did this because the criteria I’m trying to filter on – in this case the county name, Hennepin – is in a field that is formatted as text.

The syntax for the WHERE line differs depending on the format of the field you are looking in.

Here’s a little cheat sheet:

**TEXT** – use single or double quotes. Example: Where county= “Hennepin”

**DATE or TIME** – use hashtags (#). Example: where mydatefield = #01/01/2014# or mytimefield = #2:00 PM#

**NUMBER** – No special symbols are needed. Example: where myagefield = 10

Some fancier things you might want to do with your filter (my examples don’t necessarily pertain to this data)……

**Does not equal** can be written two different ways.

Example 1: Where County <> “Hennepin”

Example 2: Where County is not “Hennepin”

On number, date and time fields you might want to find something that is greater than, less than or some variation of that:

**Greater than** (this means it will NOT include the value in your criteria):

Where suffrate >50

Where mytimefield > #2:00 PM#

Where mydatefield >#1/1/2014#

**Less than** (this means it will NOT include the value in your criteria):  
Where yrBuilt<2000  
Where mytimefield<#2:00 PM#

Where mydatefield <#1/1/2014#

**Greater than or equal to** (this means it will include the value in your criteria):

Where yrBuilt >=2000

Where mytimefield > =#2:00 PM#

Where mydatefield >=#1/1/2014#

**Less than or equal to** (this means it will include the value in your criteria):  
Where Suffrate<=50  
Where mytimefield<=#2:00 PM#

Where mydatefield <=#1/1/2014#

You can combine some of these to get ranges. For example, in our bridge data we might want to find all the bridges that were built between 1980 and 1989 (but not including 1990):

SELECT \*

FROM Bridges2009

WHERE YrBuilt>=1980 and YrBuilt<1990

You might also want to find all records where a particular field does not have a value in it. This is referred to as having a NULL value. You will run across this in most datasets. Surprisingly, this bridge data doesn’t have any blank or NULL cells.

But if you were looking for NULL values, here’s how you’d do it:

SELECT \*

FROM Bridges2009

WHERE YrBuilt is null

Or you can look for all the records where that field is NOT NULL:

SELECT \*

FROM Bridges 2009

WHERE YrBuilt is not null

You can put as many filters into your WHERE line as you’d like – just know that the more you filter, the fewer records you’re going to get back.

Doing multiple filters requires you to use Boolean Logic. This is what Google and other search engines use. The most basic Boolean phrases are “AND” and “OR”

AND is more inclusive – it’s saying each record must contain BOTH of the criteria you are listing.

OR is broader – it’s saying it can have one or the other or both.

So let’s say we want to find all the bridges in Hennepin County that are also structurally deficient. Here’s the query we’d use:

SELECT county, defstat, suffrate, facilitycarried, featurecrossed

FROM Bridges2009

WHERE county = “HENNEPIN” and defstat = “S.D.”

(note that the field Defstat has periods in the abbreviation for structurally deficient)

When you run this, you’ll see you get 54 records back (in SQLite look along the bottom of the screen for “Number of rows returned”)

If we change that query to say where county = Hennepin OR defstat = structurally deficient, we get a much different answer:

SELECT county, defstat, suffrate, facilitycarried, featurecrossed

FROM Bridges2009

WHERE county = “HENNEPIN” or defstat = “S.D.”

Now we get 2,008 records and you’ll see that not all the bridges are in Hennepin County. So what we got was everything in Hennepin County and any structurally deficient bridges that are in other counties.

A better way to use OR is if we want to find all the bridges in several counties:

SELECT county, defstat, suffrate, facilitycarried, featurecrossed

FROM Bridges2009

WHERE county = “HENNEPIN” or county = “RAMSEY” or county = “WASHINGTON”

What would happen if we put AND in place of each “or” in that previous query? You wouldn’t get any records returned because SQLite would be looking for records where the county field says Hennepin, Ramsey and Washington – all in the same record. And that doesn’t exist in our data.

Now, let’s say we want to find all the bridges that cross the Minnesota River and are located in either Brown or Chippewa counties. Now we need to give SQLite a little assistance in deciding how to make sense of this – we need to tell it to look for (Brown or Chippewa) and Minnesota River. Here’s the syntax:

SELECT county, defstat, suffrate, facilitycarried, featurecrossed

FROM Bridges2009

WHERE (county = “BROWN” or county = “CHIPPEWA”) and featurecrossed=”MINNESOTA RIVER”

The other thing you might need where filtering using WHERE is a wildcard. In the example above, we looked for all bridges crossing the Minnesota River.

But we didn’t get all the records

The problem is that some of the records say “MINNESOTA RIVER”, but others say “MINNESOTA RIVER (DAM)” or “MINN R” OR “MINN R & BLACKDOG RD”, etc.

The percent sign (%) is what SQLite uses for a wildcard in the WHERE line. (Microsoft Access uses an asterisk; most other database managers use the percent sign)

We can use that percent sign a couple different ways, depending what you’re looking for.

If you want all records where the name of the featurecrossed STARTS WITH “Minn”, then we would put the percent sign at the end, like this:

SELECT county, facilitycarried, featurecrossed, defstat, suffrate

FROM Bridges2009

WHERE featurecrossed LIKE “MINN%”

ORDER BY featurecrossed

Also note that I changed the equal sign to the word “LIKE”

If we want all the records where the name of the featurecrossed includes “Minn” anywhere in the field, you put the percent sign on both sides, like this:

SELECT county, facilitycarried, featurecrossed, defstat, suffrate

FROM Bridges2009

WHERE featurecrossed LIKE “%MINN%”

ORDER BY featurecrossed

Try out those two queries and see all the variations that you get.

Notice that the second query (with 2 percent signs) casts a much wider net – catching “NB/MINNESOTA RIVER” and “SB/MINNESOTA RIVER” that the first query didn’t catch.

However, in both queries we’re getting things we don’t want like “MINNEHAHA CREEK” or “MINNEOPA CREEK”

There are a couple different ways you could deal with this, depending what you’re planning to do with this data.

If you are just doing a quick lookup trying to find all the bridges on the Minnesota River, then you’d be best to run that broad query and then either export to Excel and winnow out the ones you want there, or refine your query like this:

SELECT county, facilitycarried, featurecrossed, defstat, suffrate

FROM Bridges2009

WHERE featurecrossed like “MINN R%” or featurecrossed like “%MINNESOTA%”

ORDER BY featurecrossed

If you need to do bigger analysis on this data and want to look at all rivers or all features crossed, then you might want to first clean up (standardize) the names of the features. How to do that is best left for a more advanced exercise – look at the exercises for using OpenRefine on my Github page: mjwebster.github.io/DataJ

Now that we’ve got some basics of SQL behind us, let’s start asking our data some actual questions…..

1) How many bridges are included in this data?

2) How many bridges are located in the metro district?

3) How many bridges statewide are considered structurally deficient? What would that be in terms of the percentage of all bridges included in this data?

4) How many bridges, built before 1970, are considered structurally deficient?

5) How many bridges cross over the Little Fork River or the Little Knife River in St. Louis County?

6) How many bridges in Ramsey County have sufficiency ratings low enough to make them eligible for replacement?

7) How many bridges statewide are eligible for replacement money? What would that be in terms of the percentage of all bridges included in this data?

8) How many bridges are owned by the state?

9) What is the oldest bridge and where is it located?

HERE ARE THE ANSWERS:

**1) How many bridges are included in this data?**

-- 13,048

This can be found by simply looking at the total number of records in the table.

You could also run a query like this:

Select count(\*)

From bridges2009

**2) How many bridges are located in the metro district?**

--1975

SELECT \*

FROM Bridges2009

WHERE Dist = “METRO”

**3) How many bridges statewide are considered structurally deficient? What would that be in terms of the percentage of all bridges included in this data?**

--1170 (9% of all bridges)

SELECT \*

FROM Bridges2009

WHERE defstat='S.D.'

**4) How many bridges, built before 1970, are considered structurally deficient?**

--1,076

(yrbuilt is a number field so you don’t need the quotes)

SELECT \*

FROM Bridges2009

where yrbuilt<1970 and defstat="S.D."

**5) How many bridges cross over the Little Fork River or the Little Knife River in St. Louis County?**

19 bridges (note: there are some bridges crossing these rivers in another county)

Select \*

From bridges2009

Where county= “St Louis” and (featuredcrossed = “Little Fork River” or featurecrossed = “Little Knife river”

**6) How many bridges in Ramsey County have sufficiency ratings low enough to make them eligible for replacement?**

--18 in total

SELECT \*

FROM Bridges2009

where county='RAMSEY' and suffrate<50

**7) How many bridges statewide are eligible for replacement money? What would that be in terms of the percentage of all bridges included in this data?**

--674 (5% of all bridges)

SELECT \*

FROM Bridges2009

where suffrate<50

**8) How many bridges are owned by the state?**

3,678 bridges

Select \*

From bridges2009

Where owner= “STATE”

**9) What is the oldest bridge and where is it located?**

Bridge number 5721, located in Koochiching County; carries trunk highway 65 across the Little Fork River

SELECT COUNTY, FacilityCarried, FeatureCrossed, BridgeNbr, YrBuilt

FROM Bridges2009

ORDER BY YrBuilt

**Group By Queries:**

So far we’ve merely filtered our data to get answers to our questions, but most of the time you’ll want to ask a question that requires you to summarize either all of your data or part of it. This is where you need to do what’s called a group by query. It’s essentially the same thing as a pivot table in Excel, but is immensely more powerful and flexible.

Earlier in this exercise, remember how we asked it to show us all the bridges in Hennepin County. Wouldn’t it be great to see all the counties listed and the number of bridges in each one?

Let’s step back a minute and talk about when we would need a group by query:

Typically the records in your database will each represent one thing (1 bridge, 1 hunting accident, 1 baseball player, 1 crime incident, 1 campaign finance donation) and you may want to count them or summarize them or “group” them and compare the groups.

So questions that might require a group by query would be like:

Bridges – Which county has the greatest number of bridges?

Hunting accidents -- Which county had the greatest number of accidents? Or How many victims were women, compared to how many were men?

Baseball salaries – Which team paid the most in salaries last year? Or What are the average salaries for each position?

Crime incidents – What type of crime occurred most often last year? Did they occur more often at night than during the day?

Campaign finance – Which donor gave the most money? Or which candidate received the most money? Or which candidate received the greatest number of contribution checks?

I’ve underlined the key words that should tip you off that you might need a group by query to get the answer. Notice how many of these questions ask you for one thing in your answer – i.e. “which team paid the post”. Although you’re asking a question that implies you need just one thing in your answer, you really will need to see ALL the teams to put them in order based on the total money they paid out. Then whichever team lands at the top of your list, is the answer to your question.

Before we do a group by query, let’s look at some of the FUNCTIONS in SQL that we are going to use as part of our group by queries.

The main ones that you will use are:

COUNT

SUM

AVG

The COUNT function will count the number of records.

The SUM will total all the values in whatever field you’ve selected.

AVG will average all the values in whatever field you’ve selected.

These are primarily used in the SELECT line of your query (although there are more advanced analyses you might do where you might use them in the WHERE line)

These functions can be used stand-alone, or in conjunction with “groups”.

For example, we can count the number of structurally deficient bridges like this:

SELECT count(\*)  
FROM Bridges2009  
Where Defstat = “S.D.”

Or we can get the average length of all the bridges in our database like this:

SELECT avg(Length)  
FROM Bridges2009

But using these functions in conjunction with GROUP BY is where you get the real power of SQL.

GROUP BY is an optional line of SQL and it goes after the FROM line, but before ORDER BY.

SELECT  
FROM  
WHERE  
GROUP BY  
ORDER BY

You use Group by when you want to summarize your data based on “groups.” So in this case, let’s say we want to count how many bridges there are in each county. So our answer would be a list of the counties (1 record for each county) and a second field showing the number of bridges.

The first thing you want to do is think about what you want your groups to be comprised of. In this case we want it based on the name of the county – so then we would find the field that has the name of the county (in this case, it’s simply called “county”) and you put that in both your SELECT line and your GROUP BY line.

Then you think about the math you want to do. Do you want to count records? Or total up values that are stored in the data? Or generate an average based on values in the data? Depending on the answer, you then put COUNT, SUM OR AVG in the SELECT line.

SELECT county, COUNT(\*)  
FROM Bridges2009  
GROUP BY county

The above query counts the number of records (in this case, bridges) in each county.

Now, let’s say we want to find out which county has the most traffic crossing all the bridges in its county. In that case, we would want to add together the ADT (average daily traffic) field, using the SUM function.

SELECT county, SUM(adt)

FROM bridges2009

GROUP BY county

ORDER BY SUM(adt) desc

Finally, let’s say we want to see the average length of bridges in each county, then we need to use the AVG function on the field called “length”

SELECT county, AVG(length)

FROm Bridges2009

GROUP BY county

ORDER BY 2 DESC

The best advice I can give you about creating queries – regardless of whether you are using GROUP BY – is to first think about what you want your answer to look like. Maybe even draw it out on a piece of paper. And think about whether you want to summarize your data, or if you just want to filter it. If you want to see each individual bridge, but maybe only ones that meet a certain criteria, then you don’t need a group by. But if you want to have each record in your answer represent multiple bridges, then you need group by.

So for example, I’d like to know how many bridges fall into each of the deficiency status (“Defstat”) categories.

SELECT defstat, count(\*)  
FROM Bridges2009

GROUP BY defstat

Date and time functions:

<https://www.sqlite.org/lang_datefunc.html>