Digging Deep Into Data with SQL

Ronald Campbell
NBC Owned Television Stations
ron.campbell@nbcuni.com
@campbellronaldw

T. Christian Miller
ProPublica
t.christian.miller@propublica.org
@txtianmiller

Thomas Thoren
The Lens
tthoren@thelensnola.org
@thomasjthoren

SQL: The language of databases

Structured Query Language, SQL for short and pronounced “sequel,” is the secret sauce of relational databases. Whether you’re talking high cost database engines such as Oracle and SQL Server or open-source products such as MySQL and PostgreSQL, they all rely on dialects of this same powerful language. Some of them have complex features that take months or years to master. But at their core is something remarkably simple: SQL.

Put it this way: If a foreign language were as easy to understand as SQL, the typical high school Spanish curriculum would look something like this:

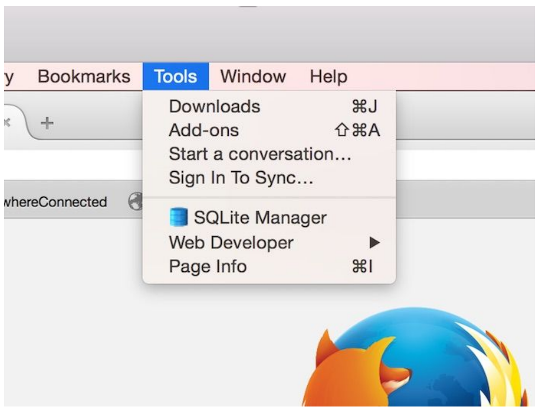
- 1st semester: Master Spanish vocabulary and grammar.
- 2nd semester: Read and discuss *One Hundred Years of Solitude* (1967) by Gabriel Garcia Marquez in the original Spanish.
- 3rd semester: Read and discuss *Don Quixote* (1605) by Miguel de Cervantes in the original Spanish.
- 4th semester: Write a series of short stories or a novel in Spanish.

In this series of three classes we’re going to do the equivalent of that first semester. You’ll learn the basic vocabulary and grammar of SQL, using SQLite, by using an open-source database extension for Firefox called SQLite Manager.

And if you want to learn more, enough to do advanced work and, eventually the journalistic equivalent of a novel -- a major project -- why not go to the NICAR conference for some training? The next conference will be March 2-5, 2017, in Jacksonville, Fla.

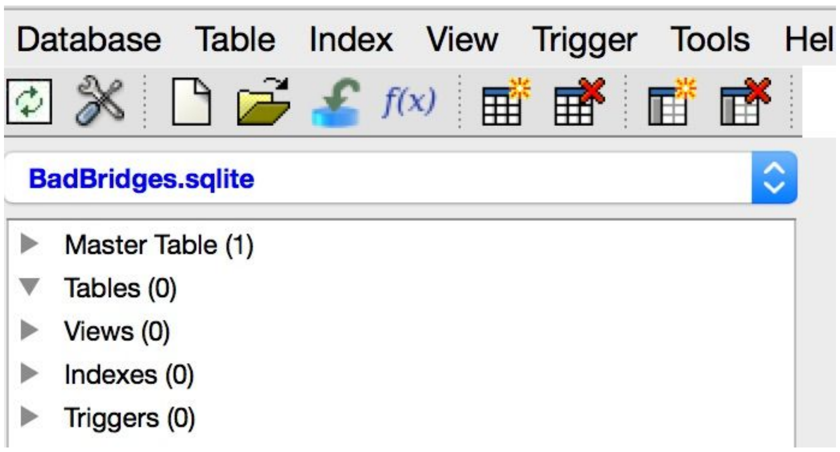
Installation

First we'll open Firefox, click on Tools and then on SQLite Manager. (You must first install SQLite Manager, which again is a free extension. To install, click on "Tools" then click on "Add-ons." On the Add-ons page, search for "SQLite Manager" and then click the "Install" button. You will then need to restart Firefox to finish the installation.)



We'll click on the "New database" icon and name the database (I chose "BadBridges"); the program appends the name with the extension ".sqlite".

Now it's time to get some data for our new database. In the menu bar, you'll see several icons; look for the blue disk with the green arrow pointing into it -- -- that's the "import" icon.



When you click on the blue import disk, a pane opens to the right with options for importing data. A button toward the upper right says “Select file.” Click on that button and browse until you find your file called **bridge15_la.csv**. This is an extract of the National Bridge Inventory. Whenever a bridge collapses -- as occurs with alarming frequency -- data journalists use this database to see what engineers knew about the bridge before the collapse and how many bridges in similar condition exist in their state. The complete National Bridge Inventory contains more than 700,000 records and is 150 columns wide. For this class we’re just looking at 14,069 bridges in Louisiana, and we’re limiting ourselves to about 50 columns of data on each of those bridges.

The standard import settings in SQLite -- first row contains column names, fields separated by comma, fields enclosed by double-quotes if necessary -- work fine with this table. Go ahead and hit “OK” at the bottom of the pane; on the next panel select the data types for each of the 50 or so columns (latmap, longmap and suffrtno all are float; avdayno, brdgimno, rdimpno, totcstno and futdlyno are integer; all other fields are varchar) and hit OK. Then hit OK a third time to import. The table should import in a few moments.

Now it’s time to put away your data mechanic hat and put on your data journalist hat. Time to interview the data.

Selecting your data

SELECT and FROM

The two most basic SQL commands are SELECT and FROM. For every SQL query, you will always need to state which columns you want and the table where those columns are located. Select all columns from the bridge15_la table. The * is a wildcard character, which means everything in this context.

```
SELECT *
FROM bridge15_la;
```

Now let's select specific columns from the bridge15_la table. These columns are the bridge status, sufficiency rating number, feature, structure, average daily traffic, year the bridge was built and the most recent inspection date. The table includes many more columns, but these are the columns that we will focus our attention on.

```
SELECT stat, suffrtno, feature, strcture, avdayno, year, inspdate
FROM bridge15_la;
```

If the column names aren't clear or are difficult to remember, you can rename the column headers to make your output easier to understand. Use the AS command to rename a column.

```
SELECT feature,
        stat AS status,
        year AS year_built,
        avdayno AS average_daily_traffic,
        suffrtno AS sufficiency_rating_number,
        strcture AS structure,
        inspdate AS most_recent_inspection_date
FROM bridge15_la;
```

We'll stick with the spreadsheet's default column names, but remember this trick if you ever get bogged down in too many abbreviations.

Try searching for various columns to explore the data. You can rearrange the order of column names in the SELECT statement to your liking.

Formatting

The capitalization of SQL syntax words is not necessary, but helps to differentiate between SQL commands and other information. I find it easier to scan this way too. This is also why I include the new lines for each successive SQL command. They make reading easier but are not necessary.

The semicolon at the end of each command is not required by all SQL software, but it is by many so it's a good habit to get into.

Filtering the data

WHERE

Now we're going to look at one of the most useful parts of SQL. The WHERE command lets you filter your data based on any number of criteria. It a row matches the given criteria, that row is returned.

For example, you could limit the rows to only those that are in Orleans Parish using the following query. The output will include all rows where the statement cnty = '071' is true.

```
SELECT *
FROM bridge15_la
WHERE cnty = '071';
```

In addition to equals (=), other common comparison operators include does not equal (!=), greater than (>), less than (<), greater than or equal (>=) and less than or equal (<=). Of course, your data must be numeric in order to use mathematical operators such as > or <. That wouldn't make much sense when comparing two words or phrases.

```
SELECT *
FROM bridge15_la
WHERE suffrtno < 50;
```

LIKE

Perhaps you only know a part of the text that you are seeking. SQL offers a useful command that lets you search by pieces of text. The % here acts as a wildcard character, meaning it can represent zero or more characters. In other words, any feature cells starting with "I-10" will match this filter.

```
SELECT *
FROM bridge15_la
WHERE feature LIKE 'I-10%';
```

You could also use it to only show features that end in "I-10."

```
SELECT *
FROM bridge15_la
WHERE feature LIKE '%I-10';
```

And if you wanted to find "I-10" anywhere, you could use the wildcard % at both the start and the end.

```
SELECT *
FROM bridge15_la
WHERE feature LIKE '%I-10%';
```

AND and OR

You might have noticed that other features refer to I-10 as "I10" or "I 10." To capture those rows as well, we can add additional filters using the OR condition.

```
SELECT *
FROM bridge15_la
WHERE feature LIKE '%I-10%'
      OR feature LIKE '%I 10%'
      OR feature LIKE '%I10%';
```

As long as the feature value matches at least one of those patterns, that row will be returned. With OR, rows are returned as long as they match at least one of the filters.

According to the data's documentation, a bridge with stat equal to 1 is structurally deficient and a bridge with statequal to 2 is functionally obsolete. Let's find those bridges using two filters combined with the OR operator.

```
SELECT feature, stat, suffrtno
FROM bridge15_la
WHERE stat = '1' OR stat = '2';
```

You can accomplish the same query using the IN operator, as shown below. This query says that as long as the statvalue is somewhere in that list of values, the row is a match and will be returned. I find this syntax a little bit easier to manage once you start searching for more than a few values. The downside to using IN is that you can no longer make use of LIKE, so only use IN when you have full-text matches.

```
SELECT feature, stat, suffrtno
FROM bridge15_la
WHERE stat IN ('1', '2');
```

The data documentation also states that, of those bridges with stat equal to 1 or 2, any bridge that also has a sufficiency rating (suffrtno) less than 50 is eligible for replacement or rehabilitation. These are the worst of the worst bridges.

To check if a bridge is structurally deficient or functionally obsolete and has a low sufficiency rating, we can use the ANDoperator.

```
SELECT feature, stat, suffrtno
FROM bridge15_la
WHERE (stat = '1' OR stat = '2') AND suffrtno < 50;
```

This means that a row will only be returned if the stat field is either "1" or "2", while also having a sufficiency rating below 50. The OR operator only requires one true value, but the AND operator requires true values from all comparisons.

Notice the use of parentheses around the OR operator in the above query. This groups the result of that comparison, which is then used in the AND comparison. The parentheses help to stay organized.

Sorting your data

ORDER BY

Since we're exploring the worst bridges, it might help to rank those bridges from worst to best. This is where the ORDER BY operator helps. You can select the column which will determine the order of the rows.

This query orders the results in descending order, based on the suffrtno values.

```
SELECT feature, stat, suffrtno
FROM bridge15_la
WHERE (stat = '1' OR stat = '2') AND suffrtno < 50
ORDER BY suffrtno DESC;
```

The default setting for ORDER BY is ascending order. In the above query, ascending order could be achieved by writing either “...ORDER BY suffrtno ASC” or “...ORDER BY suffrtno”.

LIMIT

The LIMIT command forces your query to only return the specified number of rows. This is commonly used in conjunction with ORDER BY to show a small set of ranked rows ("The 10 worst bridges in Louisiana").

This query orders the results in ascending order (the default order), based on the suffrtno values, and only returns the first 10 values. Because we are sorting from low to high (bad to good) sufficiency rating numbers, and limiting the results to the first 10, this query returns the 10 worst bridges (at least according to these columns).

```
SELECT feature, stat, suffrtno
FROM bridge15_la
WHERE (stat = '1' OR stat = '2') AND suffrtno < 50
ORDER BY suffrtno
LIMIT 10;
```

Aggregate functions

SQLite offers built-in functions to perform basic calculations on your data. COUNT, MAX, MIN, and AVG are some common ones. You can read more about them here.

COUNT

Return the number of rows matching your query. This is especially useful when combined with WHERE statements to understand how many rows match your filters.

```
SELECT COUNT(*)
FROM bridge15_la;
WHERE cnty = '071';
```

AVG

Return the average value for the column specified.

```
SELECT AVG(suffrtno)
FROM bridge15_la;
```

MAX

Return the greatest value for the column specified.

```
SELECT MAX(suffrtno)
FROM bridge15_la;
```

MIN

Return the smallest value for the column specified.

```
SELECT MIN(suffrtno)
FROM bridge15_la;
```

A few more notes

Comments

As your queries grow more and more complex, it might help to write comments within your SQL code to note what a particular line does or explain why you are writing a query in the first place. Your future self with be grateful when you revisit your code.

If you are familiar with other programming languages, then you are probably familiar with the idea of comments in your code. These are lines that are not executed and only exist for people reading the code.

In SQL, you can write comments in two ways. For a single line, you can use two hyphens (--) to begin your comment. For example:

```
SELECT stat, suffrtno  -- suffrtno stands for "sufficiency rating number"
FROM bridge15_la;
```

Another more flexible way to write comments is using the /* Comments here. */ syntax. These can be used for a single line or multiple lines. For example:

```
/*
Everything inside here is a comment and won't be executed in the SQL query.
This query accomplishes two things:
- Filters the data to only include structurally deficient or functionally obsolete bridges.
- Filters the data to only include bridges that also have sufficiency ratings below 50.
*/
SELECT feature, stat, suffrtno  -- suffrtno is the sufficiency rating number
FROM bridge15_la
WHERE (stat = '1' OR stat = '2') AND suffrtno < 50;  /* These are some bad bridges. */
```

Data types

Stay aware of the different types of data in your tables. Common types include integers (whole numbers), floats (numbers with decimals), booleans (True or False), text and dates.

This is very important when you have data containing a leading zero (e.g. zip code 07712). If you were to convert that to an integer (7,712), it would lose its meaning. Conversely, you should make sure numeric data is stored as numbers and not text so that you can make use of mathematical operators such as =, < and >.

Differences in SQL syntaxes

The various flavors of SQL (SQLite, MySQL, PostgreSQL, SQL Server, etc.) all have slightly different syntaxes, but they are mostly the same when it comes to basic usage. This can be annoying when switching between the SQL languages, but the good news is that they all have been around for decades. That means most syntax fixes are well-documented and only a quick Google search away.

NULL

One confusing point with SQL and programming languages in general is the idea of NULL. In databases, you can declare whether or not a column allows NULL entries, meaning whether or not they can lack any values. This is a subtle but significant difference between an empty value. An empty value means the emptiness is reported, whereas a NULL value means nothing is reported at all. It is the lack of anything.

You can filter based on whether a cell is NULL or not using IS or IS NOT as the comparison operators, instead of = or !=. Again, this is because NULL is not really equal to anything; it's the absence of any value.

```
SELECT *
FROM bridge15_la
WHERE stat IS NULL;
```

Heavy analysis: Grouping and summing

In the earlier class, you got to see some of the basics of SQL, the language of databases. Now let's learn some power moves.

SQL is a great language for exploring data. It can use WHERE statements to select certain bridges (the ones that we care most about, i.e., the ones that are about to fall down). It can rank bridges, to show the truly terrible ones via the ORDER BY statement. And it can count which counties in Louisiana have the worst bridges using the COUNT statement.

But there's so much more! Let's say you wanted to know how much it was going to cost to repair all the bridges in a county? Or which counties had the highest average cost per square foot (perhaps the grafty bridge officials pad their estimates?)

That's where grouping and summing, or aggregate functions, come in. These are basic math functions that will help you get a better picture of your data — and will lead you to better stories. The list of basic functions in SQLite are found here.

Before we start, let’s get a handle on the idea of grouping. It’s one of the most powerful features of SQL, but it can be a little difficult to conceptualize at times. A walk through an imaginary grocery store aisle can help.

Let’s say you’re in the produce section. The most basic way to group the stuff there is by type: there are fruits and there are vegetables. Everything in the produce aisle fits into one of those two categories.

Another is by color: let’s say you were on a new diet that allowed you to only eat yellow stuff. There could be lemons, but also squash. You’ve created a new group: things that are yellow. It will include both fruits and vegetables. But not cherries or beets. (Unless you’re at one of those artisinal places that sells yellow cherries.)

You can also create groups within groups: think of all citrus fruits. There are lemons, limes, grapefruits. Or vegetables grown below ground: peanuts, potatoes and radishes. And vegetables grown above ground: corn, lettuce, and artichokes.

The point of this extended culinary metaphor is that everything in the produce section has a series of characteristics that can be used to place them into categories. The same applies to data.

Grouping is the way you cut data to explore it.

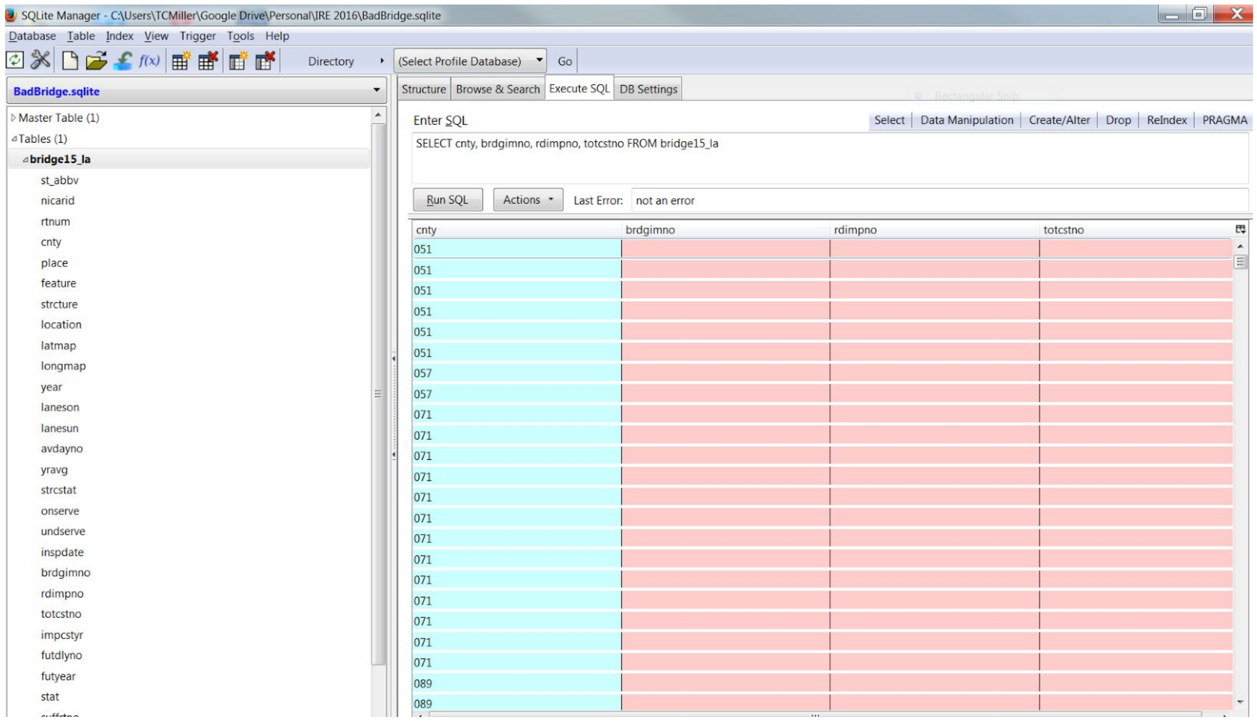
Back to the bridges. You can look at bridges by county. Or by the average traffic per day. Or by the year built. Let’s take a look at why this can get interesting.

There are three fields in bridge15_la that have to do with the cost of replacing these mouldering bridges. The first is brdgimno — that’s the cost in thousands of dollars to replace the bridge itself. The second is rdimpno — that’s the cost to improve the road. And then there’s totcstno — the whole shebang, which includes the bridge, the road and everything else associated with fixing up the bridge.

(Parenthetically, the cost numbers that come in the raw data from the Department of Transportation are tricky to understand. That’s why it’s crucial to always get a data dictionary, or record layout, every time that you obtain data. In this case, the kind folks at IRE have converted the data for you. Details on how the DOT tracks costs in the raw data can be found here.)

Your reporter instincts fire. Money. Follow the money. Each county submits these costs to the government. Which county submits the highest estimates? Which the lowest? Let’s do our select statement:

SELECT cnty, brdgimno, rdimpno, totcstno FROM bridge15_la;



```
SELECT cnty, brdgimno, rdimpno, totcstno FROM bridge15_la
```

Last Error: not an error

cnty	brdgimno	rdimpno	totcstno
051			
051			
051			
051			
051			
051			
057			
057			
071			
071			
071			
071			
071			
071			
071			
071			
071			
071			
071			
071			
071			
089			
089			

You get something that looks like this. It’s a list of every bridge in every county, with the cost to repair estimates. The first thing you’ll notice is a bunch of empty rows — red is the color that SQLite uses to tell you there is no data, or NULL.

But scroll down. Soon, you’ll see numbers begin to appear. The quick way to do this is to click **twice** on the column header — the word that says totcstno. That automatically sorts the database by the highest to lowest cost. (One click sorts it from lowest to highest.) We can see that county 103 — St. Tammany parish, right on the shores of Lake Pontchartrain — has some expensive bridges to fix.

But each row that you see right now is an individual bridge. Let’s see you wanted to see the TOTAL cost to fix all the bridges in each county.

Here’s where we group — we’re going to group the bridges together by county, just like we group together all the lemons into a group of fruit, or yellow things, or citrus.

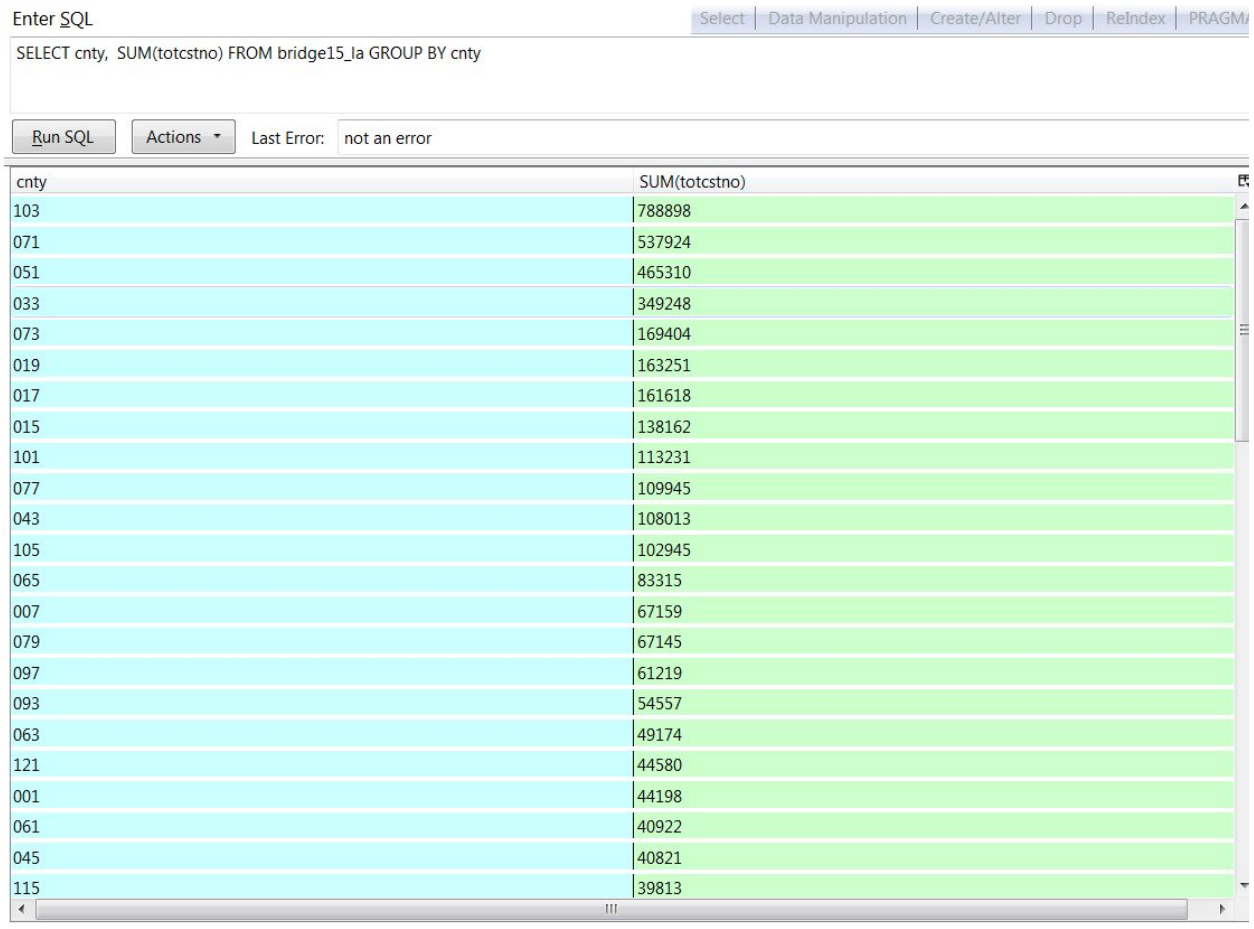
First, we'll use the SQL command GROUP BY. SQL only has seven basic words, and they always come in this order:

SELECT
FROM
JOIN
WHERE
GROUP BY
HAVING
ORDER BY

GROUP BY is almost always used in combination with an aggregate function like COUNT or SUM or AVG (average) after the SELECT statement. In this case, we want to tell SQLite to GROUP all the bridges together by county, and then SUM up the cost of fixing all the bridges in the county. Like this:

```
SELECT cnty,  SUM(totcstno) FROM bridge15_la GROUP BY cnty
```

In English, this line tells SQLite, "Get each county. Add up the costs to fix all the bridges in the county and tell me the total cost for each county. You should get something like this. Click twice on the column labeled SUM(totcstno) to sort the table by highest to lowest cost.



```
SELECT cnty,  SUM(totcstno) FROM bridge15_la GROUP BY cnty
```

Last Error: not an error

cnty	SUM(totcstno)
103	788898
071	537924
051	465310
033	349248
073	169404
019	163251
017	161618
015	138162
101	113231
077	109945
043	108013
105	102945
065	83315
007	67159
079	67145
097	61219
093	54557
063	49174
121	44580
001	44198
061	40922
045	40821
115	39813

Looks like St. Tammany Parish is still the champion. Their total bill is 788,898 — but remember this is in thousands of dollars so that means 788,898,000 — nearly a billion dollars to fix their bridges. Number 71, Orleans Parish isn't even close.

But can we write a story saying that St. Tammany Parish is filled with money grubbing engineers who are inflating the bill to fix their bridges? Nope. Why? Because we don't know how many bridges are in the parish. Maybe it has tons of bridges. What to do?

More GROUP BY! And SUMMING.

Remember the COUNT function from lesson one? We want to tell SQLite to COUNT how many bridges are in each county. We do that thusly:

SELECT cnty, COUNT(*), SUM(totcstno) FROM bridge15_la GROUP BY cnty

The GROUP BY command controls how the aggregate functions in that follow the SELECT command work. That's why you usually see GROUP BY always with aggregate functions. They work together.

Now double click again on the SUM(totcstno) column and see how many bridges there are in St. Tammany. A lot — 390. But we still really can't compare St. Tammany to other parishes. We need to see the average cost per bridge. This gets a little complicated, but hold on. Remember, your competitor is trying to do all this by hand. You can do it in seconds with SQL.

What we want to do is tell SQLite: tell me how many bridges there are in each county. Tell me the total bill for the county. Then divide the total bill by the number of bridges — that's the average cost per bridge.

This will create an apples-to-apples (there's that fruit stuff again) comparison. Which county has the highest per bridge repair bill. Is St. Tammany really inflating their estimates to get more federal money, when compared to other counties?

Before we run the SQL command, there's one more catch. Remember those red boxes from the first query? Those were bridges that had no cost estimates, or didn't need repair. So we don't want to include those in our count of bridges in each county. We only want to look at bridges WHERE there is a price tag to fix them.

The final SQL command looks like this:

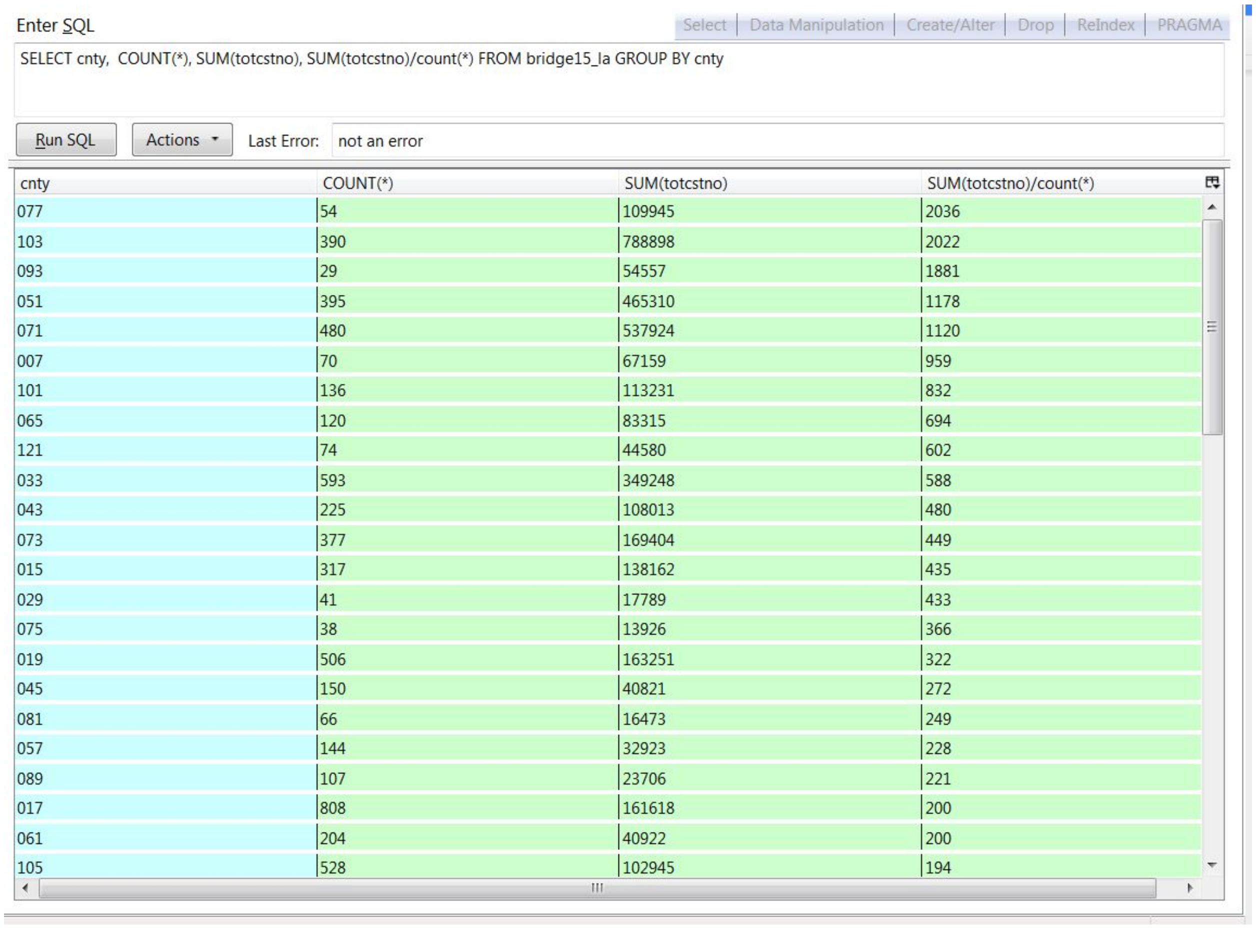
SELECT cnty, COUNT(*), SUM(totcstno), **SUM(totcstno)/count(*)**, AVG(totcstno) FROM bridge15_la WHERE totcstno > 0 GROUP BY cnty

Let's break that down.

This bit — **SUM(totcstno)/count(*)** — uses the forward slash, which is what SQLite uses for "divide by". So it's telling SQLite, take the total cost for each county, and divide by the number of bridges in the county. Then show me how much the average cost per bridge is.

The WHERE totcsatno > 0 tell SQL to ignore those bridges where there is no repair cost. We don't want to include perfectly fine bridges.

Run the query. Now click twice on the column labeled SUM(totcstno)/count(*). This will show you the county with the highest average cost per bridge. You'll get something like:


Enter SQL
Select | Data Manipulation | Create/Alter | Drop | ReIndex | PRAGMA
Run SQL
Actions
Last Error:

```
SELECT cnty,  COUNT(*), SUM(totcstno), SUM(totcstno)/count(*) FROM bridge15_la GROUP BY cnty
```

not an error

cnty	COUNT(*)	SUM(totcstno)	SUM(totcstno)/count(*)
077	54	109945	2036
103	390	788898	2022
093	29	54557	1881
051	395	465310	1178
071	480	537924	1120
007	70	67159	959
101	136	113231	832
065	120	83315	694
121	74	44580	602
033	593	349248	588
043	225	108013	480
073	377	169404	449
015	317	138162	435
029	41	17789	433
075	38	13926	366
019	506	163251	322
045	150	40821	272
081	66	16473	249
057	144	32923	228
089	107	23706	221
017	808	161618	200
061	204	40922	200
105	528	102945	194

Now which parish comes out on top? St. Tammany Parish is still on top, with an average cost of $7,043,000 million per bridge. Bur right behind is Number 093, or tiny St. James Parish, right between Baton Rouge and New Orleans. They've only got 8 bridges but their cost per bridge is $6,819,000 — that's right behind St. Tammany Parish, but much higher than the third place county, New Orleans.

It's not a huge difference. But it also shows that those two parishes, St. James and St. Tammany, have much higher costs per bridge than any other parish. And that's something you never would have known unless you'd crunched the numbers with the database. It's not something that you could detect by simply running your eye over a list of 14,000 bridges — or even a list of bridges per county.

So now you have two parishes to investigate — one large, and one small. That's the power of grouping and summing.

Playing well with others: Joining data

So far we've just worked with Bridge15_la, the Louisiana bridge table. But the real power of relational databases -- only the second time we've used that word, *relational* -- is that they can analyze relationships and reveal hidden patterns. We're going to start with something simple and move on to something more complicated.

First let's take a closer look at the bridge table:

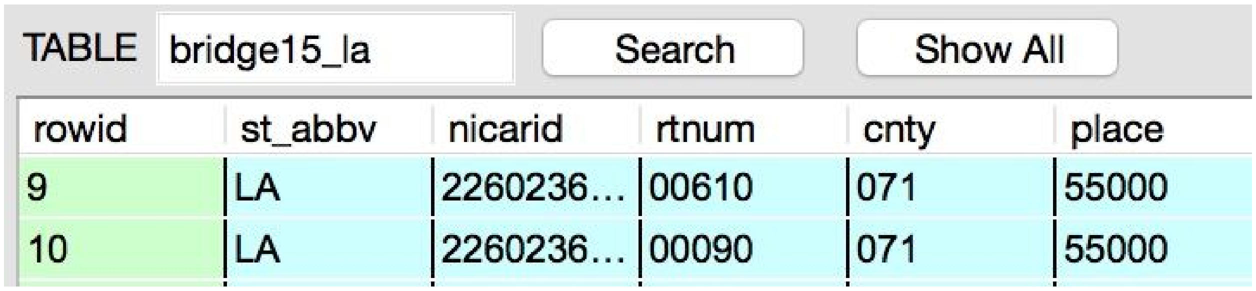


TABLE bridge15_la Search Show All

rowid	st_abbv	nicarid	rtnum	cnty	place
9	LA	2260236…	00610	071	55000
10	LA	2260236…	00090	071	55000

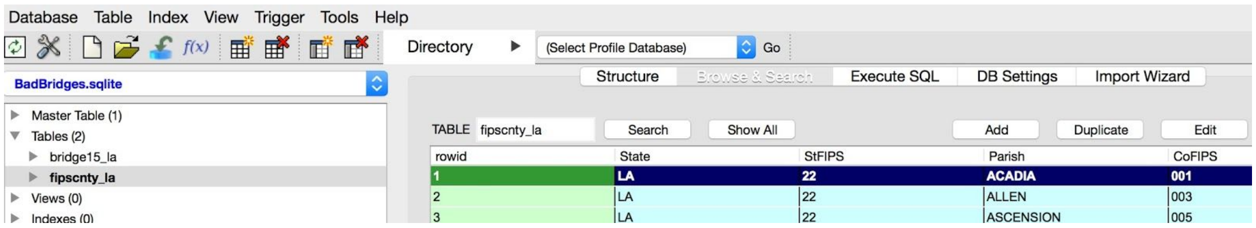
The st_abbv is obvious: "LA" is Louisiana. The "sternum" most likely refers to "route numbers" and "nicarid" is, per the documentation, an ID added by the folks at NICAR.

But what are those "cnty" and "place" numbers? Those are FIPS codes -- Federal Information Processing System codes. Spend enough time with federal databases, and you will know the FIPS codes for your local area by heart. The rationale for FIPS codes is simple: Not only do they save space (a necessity way, way back when computer hard drives were expensive), but they also prevent error. Think how many ways someone could name a particular place in a computer: *"Orleans", "Orleans Parish", "Orleans Par."* Or they could simply enter the three-digit code *"071"* which, together with the two-digit state code *"22"* can only mean *"Orleans Parish, Louisiana"*.

We're going to import a table that already has all the FIPS codes for Louisiana counties. Then we'll merge, or *join*, it with the bridges table so that we can say where the bridges are located.

First click on the Import icon to bring up the import pane, click on the “Select File” button, navigate to the correct folder and then click on **fipscnty_la.csv** and click OK to import; all four fields are varchar, short for variable character fields. You’ll get 64 records, one for each county in Louisiana.

If you click on fipscnty_la in the left panel and the Browse & Search button on the right, this is what you’ll see:



rowid	State	StFIPS	Parish	CoFIPS
1	LA	22	ACADIA	001
2	LA	22	ALLEN	003
3	LA	22	ASCENSION	005

The CoFIPS field in fipscnty_la matches the cnty field in bridge15_la. Knowing that, we can join the two tables and put the parish name in records where the cnty number now appears. We’ll begin by clicking on the Execute SQL button.

Until now we’ve just worked with the bridge15_la table. Now we’re going to work with fipscnty_la as well. To keep things simple, we’ll refer to the two tables by *aliases*.

We’ll start with this statement:

```
SELECT parish, feature, strcture, avdayno, suffrtno
FROM bridge15_la a, fipscnty_la b
WHERE a.cnty = b.cofips
```

This returns 14,069 rows, a subset of the entire database, but this time with the parishes identified. Next we’ll limit our selection to highways:

```
and onserve = '1'
```

That reduces the selection to 12,895. If we set a lower threshold on average daily traffic, at least 1,000 cars per day, we’ll further limit our selection:

```
and avdayno >= 1000
```

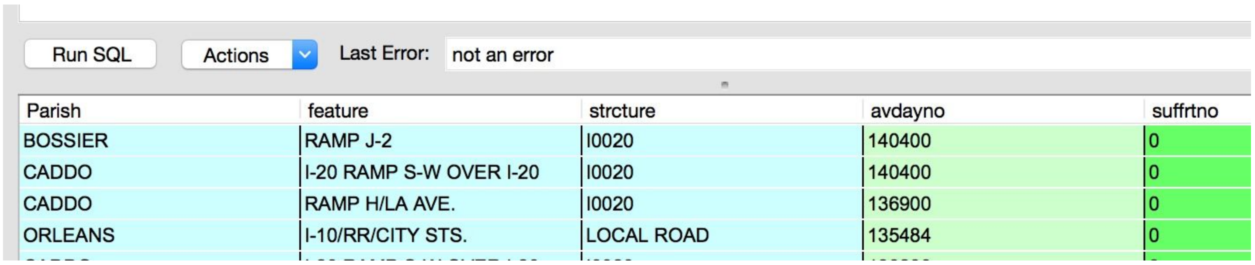
That makes a big difference: just 6,893 bridges -- and now we're starting to see some of the same parishes over and over again. Finally we'll look for bridges with a sufficiency rating at or below 50, and specify that we want the table ordered by sufficiency rating -- meaning that the worst bridges will appear at the top.

```
and suffrtno <= 50
order by suffrtno
```

Since there are dozens of bridges rated 0, we'll add a second tie-breaker to the order, avdayno (traffic). We'll specify that we want that, unlike sufficiency to be ranked in descending order.

```
order by suffrtno, avdayno DESC
```

This means that SQLite will rank bridges first by their sufficiency rating but that, in case of ties, bridges with more traffic will rank higher. And we have a couple of "winners":



Parish	feature	strcture	avdayno	suffrtno
BOSSIER	RAMP J-2	I0020	140400	0
CADDO	I-20 RAMP S-W OVER I-20	I0020	140400	0
CADDO	RAMP H/LA AVE.	I0020	136900	0
ORLEANS	I-10/RR/CITY STS.	LOCAL ROAD	135484	0

But database software lets us do much more than just look at one database in isolation. The National Bridge Inventory was designed by engineers to answer one question: whether bridges are being properly maintained. But as journalists we sometimes ask broader questions: For example, why are some bridges better maintained than others?

Rather than just look for answers to that question inside the National Bridge Inventory, let's turn to the Census Bureau and the American Community Survey, specifically to data on median household income by county. There's evidence that politicians cater to the affluent when they write the laws. Is there a broader pattern? Does it extend as far as better maintained bridges in affluent counties?

We'll import another file called **HHInc_LA.csv**. Click on the blue import icon, then the "Select File" button, navigate to the correct folder, click to open and hit OK to begin the import. The CoFIPS and Geography fields both are varchar; the Household_Inc field is integer. There are 64 records, one for each county.

We’re going to compare household income in each parish with the average sufficiency rating. Again we’ll use aliases to make things less confusing. Here’s the query:

```
SELECT b.geography, avg(suffrtno) as Sufficiency, Household_Inc
FROM bridge15_la a, hhinc_la b
WHERE a.cnty = b.cofips
GROUP BY b.geography
ORDER BY sufficiency
```

The result is not very satisfying. The parish with the lowest average sufficiency rating, East Baton Rouge, has a median household income of $48,535. The parish with the highest rating, Tensas, gets by on $26,178. Get fancy and run a correlation on the results, and the correlation is a *negative* 0.389, meaning that bridges tend to be slightly better maintained in poorer parishes.

Of course we’ve tried just one possible explanation for the sorry condition of bridges in Louisiana. We could try using place instead of county as our geographic variable. We could try race or education levels instead of income as our explanatory variable, or we could look deeper into the Inventory, at the age of the bridges.

Or we could simply go to the state line and post a sign on every highway:

Now entering Louisiana.
Watch for falling bridges.