An introduction to SQL for journalists

**SQL** (Structured Query Language) – generally pronounced ‘*sequel*‘ – is a widely used language for querying data. It is traditionally used with large databases, but more recently you will also see it used with online sources of data and tools.

For example you can use SQL in the mapping tool Carto to query the data behind a map (and generate different views); and you can use SQL in Scraperwiki to query your scraper’s data (and generate different feeds of live data).

There are three main reasons to use SQL:

\* Firstly, it allows you to query large datasets that you would either not be able to open in spreadsheet software, or which would slow down your computer significantly. This is because SQL queries a basic text or CSV file directly rather than the larger spreadsheet formats like XLS, or memory-heavy tools like Excel.

\* Secondly, and related to that, it is very **fast**: you can perform a number of different queries simultaneously which you might normally have to do separately.

\* Thirdly, as mentioned above, it allows you to generate ‘live’ insights into data where it is hosted in an online tool such as Carto, Scraperwiki, and many APIs.

Thankfully, SQL is not a complicated language, and it is not a very extensive one either - at least for most queries anyway. Most of it uses plain language expressions which will make sense to you from the start.

The biggest barrier is normally finding a tool that allows you to run SQL queries. But it is so widely used that there are actually lots of ways to use it in existing tools. For example, you can get to know SQL in the following websites and tools:

* Data.world
* Franchise: <https://franchise.cloud/app/>
* Morph.io (documentation: <https://morph.io/documentation/api>)
* Google Sheets (using the function QUERY)
* The Firefox extension [SQLite Manager](https://addons.mozilla.org/en-us/firefox/addon/sqlite-manager/).
* If you have a Carto account look for the SQL toggle switch at the bottom of the data view and read about the Carto API: <https://carto.com/docs/carto-engine/sql-api/making-calls/>
* [DB Browser for SQLite](https://sqlitebrowser.org/)

# The most basic query: SELECT everything from the table

A typical SQL query might ask something like the following: ‘*Select all the lines in this data where the first name is Alice and the age is over 25* or: *Group all the data by city, and count how many entries there are in each*‘.

But let’s start with the most basic query: ‘select everything from a particular table’. Here’s how you would write that in SQL:

SELECT \* FROM councilspending

SELECT specifies which fields (column headings) in your table you want to show in the results.

The asterisk means ‘all’. In other words, all fields in that table.

FROM specifies which table or tables you want to query. In the example above the table happens to be called ‘councilspending’. Obviously when adapting the expression above you would change that name to the particular name of *your* table.

We have used capital letters to indicate the SQL commands, but they normally work just as well in lower case. It’s just a convention that makes it easier to read your queries.

As it stands this query doesn’t do anything useful – it only gives us the whole table we had anyway. But as you are more specific in what you select, or add extra commands, you can drill down further.

# Forming the question: SELECT, WHERE and ORDER BY

Three basic commands to start with are SELECT, WHERE and ORDER BY:

* SELECT is used to identify which columns you want to **show** (not necessarily the ones you’re questioning)
* WHERE is used to specify **conditions**, such as ‘Where this column has this value’ or ‘Where that column is below this value’
* ORDER BY is used to **order** the results: for example by a numerical column (highest to lowest, or lowest to highest), or by a text column (alphabetically A to Z or Z to A)

We’ve already used SELECT to select all fields. But often we can use it to filter results to specific fields.

If we only wanted to see the fields ‘vendor*name’ and ‘invoice*amount’ from our table we would specify this by naming them with SELECT like so:

SELECT vendor\_name, invoice\_amount FROM councilspending

Notice, each field is separated by a comma.

If your field names have spaces you should rename them so that they are removed or replaced with an underscore. Otherwise the SQL query will see the two words as two separate fields or commands.

As you can see, most of these clauses use simple language and are pretty intuitive to use.

Next, the WHERE command can be used to filter results by certain criteria like so:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE invoice\_amount > 3000

Again, this extra command WHERE invoice\_amount > 3000 is pretty straightforward: only show results where the invoice amount field is greater than 3000.

You can also use this command to specify certain text entries like so:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE vendor\_name = 'THOMPSONS SOLICITORS'

Notice that when specifying text values:

* You must specify the value as a string in inverted commas
* The match must be exact. If you want to look for partial matches, i.e. ‘contains this string’, then you need to use the LIKE command explained below.

You can combine filter your data by multiple criteria too by separating each criteria with a comma:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE vendor\_name = 'THOMPSONS SOLICITORS', invoice\_amount > 3000

The results will be shown in no particular order. So if you want to show them alphabetically, or from largest to smallest, or vice versa, you need to use ORDER BY with the name of the field you are ordering on, like so:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE invoice\_amount > 3000 ORDER BY invoice\_amount

By default it will order in ascending order – that is, from smallest to largest or A to Z. If you want it to order from largest to smallest or Z to A you will need to add desc like so:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE invoice\_amount > 3000 ORDER BY invoice\_amount DESC

And you can order by multiple columns *in order*. For example:

SELECT vendor\_name, invoice\_amount FROM councilspending WHERE invoice\_amount > 3000 ORDER BY invoice\_amount DESC, vendor\_name ASC

This will order by invoice amount first. But where two invoices are the same value, it will then order within that group of entries by vendor name.

With those three basic commands – SELECT, WHERE and ORDER BY – you can grab subsets of datasets based on simple criteria.

But you may want to perform some calculations on those results such as adding up the total for each recipient, or an average or maximum. For that you will need another command.

# SQL’s answer to pivot tables: GROUP BY

The GROUP BY command is used to **aggregate** results, in the same way as a pivot table might group payments by company. For example you might group results (for example totals or averages) by each country listed in one field, or each category in another.

The group by clause is not quite as simple as other clauses. This is because it needs to be combined with a calculation such as SUM, AVG, MAX, MIN or COUNT.

These are called **aggregation functions** and are very similar to similarly-named functions in Excel or Google Sheets (with some key differences I’ll come on to).

These functions are used at the start of the SQL query when you SELECT your fields, like so:

SELECT SUM(invoice\_amount)

Left alone that would give us a total of all invoice amounts. But when GROUP BY is added we get the total *grouped by* whatever we specify. For example:

SELECT SUM(invoice\_amount) FROM councilspending GROUP BY vendor\_name

Now we should have a list of sums, one for each vendor. But we can’t see the actual vendor names because we haven’t selected them. Adding that field to our SELECT command should solve that:

SELECT SUM(invoice\_amount), vendor\_name FROM councilspending GROUP BY vendor\_name

Clearly you can do more than grouping by sum: you can group by MAX (maximum amount for each country, region, etc.); by MIN (minimum amount for each); COUNT; or average.

Notably, **the average is not calculated using AVERAGE**. Instead you must use AVG, like so:

SELECT AVG(invoice\_amount) FROM councilspending GROUP BY vendor\_name

If you want to filter on the results then you can use [HAVING](https://www.dofactory.com/sql/having) in the same way as WHERE is used on raw data.

# Adding more than one calculation column

Using AVG or SUM will generate a table with a column specifically containing the averages or sums grouped by the column you specified.

But what if you want to do both? Well, you can.

All you have to do is add a **comma** between the two calculations like so:

SELECT MIN(invoice\_amount), MAX(invoice\_amount), vendor\_name FROM councilspending GROUP BY vendor\_name

You can add more and more columns by adding another comma and the calculation you want to use.

# Calculations with text data

Two of these functions – AVG and SUM – only work with numerical data. But curiously the other three – MAX, MIN and COUNT – will work with text data too, unlike the functions of the same name in Excel/Google Sheets, which only work with or count numbers.

When used with text MAX, for example, will give you the entry which is **alphabetically last**. So, ‘zebra’ rather than ‘ant’.

Conversely, MIN will give you the entry which comes first alphabetically: ‘aardvark’ rather than ‘zonkey’, then (a zonkey is a cross between a zebra and a donkey! And don’t get me started on zedonks…)

COUNT will count numbers *or* text. But empty cells may generate an error.

# Finding a partial match on text: LIKE

Earlier I mentioned a limitation with WHERE when dealing with text fields: the match must be exact.

If you want to broaden your criteria to include cells which are a partial match – for example you want postcodes which have ‘B6′ in them but also other characters – then you need to use LIKE.

Here’s an example:

SELECT invoice\_amount, vendor\_name FROM councilspending WHERE vendor\_name LIKE '%Hotcourse%'

Two things to highlight:

* LIKE is used where you would normally use an operator like ‘equals’ or ‘greater than’.
* The percentage sign is placed before and after the string you are looking for.
* The string – including the percentage signs – is placed in single speech marks.

The percentage symbol is a **wild card**: it means ‘none or any characters’. By placing it before and after your string of characters you are specifying that you are looking for matches against ‘none or any characters’ followed by your specific characters (‘Hotcourse’ above), followed by ‘none or any characters’.

You don’t have to put it at both ends. If you want to specify that your string begins with or ends with the string you specify, then omit the percentage symbol. Likewise you can put percentage symbols in the middle. Here are some examples:

* LIKE 'F%' : String must begin with ‘F’ followed by none or any characters
* LIKE '%Ltd' : String can begin with none or any characters, but must end with ‘Ltd’
* LIKE 'Barclays%Ltd' : String must begin with ‘Barclays’ followed by none or any characters, and must end with ‘Ltd’

# Writing queries with multiple or alternative criteria

Until now I’ve only talked about where clauses with only one criterion. But often you want to set multiple criteria, for example:

* You only wanted results in one particular category AND above a certain value.
* You wanted results from this category OR this category OR that category.
* You wanted results from this category OR below a certain value.

If you wanted to run a query like those which sets multiple criteria then you will need to use **parentheses** with the WHERE clause.

Here is an example:

SELECT invoice\_amount, Directorate FROM June\_2012 WHERE (invoice\_amount > 1000000 OR invoice\_amount < 1000)

This will return matches where the invoice amount is greater than 1,000,000 or less than 1,000.

You can use AND in the same way to see matches where two or more criteria are met:

SELECT invoice\_amount, Directorate FROM June\_2012 WHERE (invoice\_amount > 1000000 AND Directorate = 'CYPF')

And you are not limited to just two criteria: add more by adding further OR or AND criteria like so:

SELECT invoice\_amount, Directorate FROM June\_2012 WHERE (invoice\_amount = 500 OR invoice\_amount = 1000 OR invoice\_amount = 2000)

And if you want to combine both, use further parentheses:

SELECT invoice\_amount, Directorate FROM June\_2012 WHERE (invoice\_amount > 1000000 AND (Directorate = 'CYPF' OR Directorate = 'Finance Control'))

Another useful command here is [IN](https://www.sqlitetutorial.net/sqlite-in/). This allows you to specify a number of different matches, e.g. WHERE names IN (‘Paul’,’Fred’)

# Combining tables: UNION and JOIN

The real power of SQL comes when you use it to query *more than one* table. In this case your query combines the data in some way and then queries the resulting combination. There are two broad ways in which it might combine two or more tables:

* If the tables have the **same structure** (the same columns of data) but are for different periods of time or places, then you need to join them **vertically**: in other words, you will be putting each table on top of another, so that you have the same number of columns, but more rows (as you add data for more months, years, or areas)
* If the tables have a **different structure and different information to each other** (for example one has crimes for each town, and another has populations for each town), then you need to join them **horizontally**. In other words, the tables will be joined side-by-side. This means that you won’t have any more rows, but you will have extra columns. To do this the two tables **must have at least one piece of information in common** (the names of the towns, for example, or better still: codes for each town).

## Joining vertically (different periods or areas)

A vertical join can be done by using the UNION or UNION ALL command in SQL. Here’s an example:

SELECT \* FROM datafor2017\_18

UNION ALL

SELECT \* FROM datafor2016\_17

This simply selects all the data from one table, and joins it vertically with all the data from the other.

The difference between UNION and UNION ALL is that UNION will eliminate duplicate rows. UNION ALL will not get rid of any rows. If you think there are likely to be duplicates, use UNION, but bear in mind that rows can have exactly the same data and not be a duplicate, especially if dates are not included: for example you might have a row for your 2016-17 data that says Birmingham had 300 burglaries, and in the 2017-18 data Birmingham just happened to have 300 burglaries again. You want to make sure you still have both rows.

## Joining horizontally (fetching related information)

A horizontal join can be done using one of the JOIN commands, and these are a bit more complex than UNION. The JOIN commands include ([quoted from W3Schools](https://www.w3schools.com/sql/sql_join.asp)):

* (INNER) JOIN: Returns records that have matching values in both tables
* LEFT (OUTER) JOIN: Return all records from the left table, and the matched records from the right table
* RIGHT (OUTER) JOIN: Return all records from the right table, and the matched records from the left table
* FULL (OUTER) JOIN: Return all records when there is a match in either left or right table

The one that you are most likely to use is LEFT JOIN. This is equivalent to VLOOKUP in a spreadsheet. It will look for a value in the ‘left’ (first) table and then fetch any matching values in the other (second) table.

Here’s an example which is joining a table showing expenditure, with another table that describes what the cost centre codes mean. In other words, we have some data on how money is being spent, including cost centre codes, but no explanation of what the cost centre codes mean. And we have another table which has a row for each cost centre code, and a description, but doesn’t tell us how much money is being spent. We need to combine them:

SELECT \* FROM october\_2018\_invoices\_over\_500

LEFT JOIN cost\_centre\_listing ON october\_2018\_invoices\_over\_500.cost\_cente = cost\_centre\_listing.cost\_centre

In this example the SQL:

1. First grabs all fields from the table october\_2018\_invoices\_over\_500.
2. Then it specifies that it wants to LEFT JOIN with another table, cost\_centre\_listing.
3. Finally, it needs to specify *how* to join them, using ON. In other words, which columns can they be matched ON. In this part we have to name *both* the field *and* the table that it comes from, using a period to do so, in this pattern: *tablename.fieldname*. So: october\_2018\_invoices\_over\_500.cost\_cente = cost\_centre\_listing.cost\_centre

Note that in the resulting joined dataset, some column names may be changed by placing the original table name before it, with a period as in the pattern above.

You can extend your query to add further commands and calculations, but check that you are naming the fields correctly - it’s safest to use the *tablename.fieldname* approach detailed above.

You can see some examples of UNION ALL and LEFT JOIN queries in this project: <https://data.world/paulbradshaw/homelessness> - look for the ‘Queries’ section on the right.

# Dealing with dates-as-text and extracting parts of text

Dates are often brought in as text like “05-09-2020” so if you want to filter on those dates you will need to separate parts of it such as the year or month.

There are commands like LEFT (LEFTSTR in SQLite) and RIGHT (RIGHTSTR in SQLite) and [SUBSTR](https://www.sqlitetutorial.net/sqlite-functions/sqlite-substr/) that you can use to do this.

For example, if you wanted to create a new field in your data which extracted the year from the ‘date’ column you might use RIGHTSTR(date, 4) - this takes the last 4 characters from that field.

This query below does that, stores the results as ‘year’ and filters on that new field using AS:

SELECT \*, RIGHTSTR(date, 4) AS year FROM october\_2018\_invoices\_over\_500 WHERE year IS ‘2020’

A more advanced approach is to convert those text values into numbers. This can be done with [CAST](https://www.guru99.com/sqlite-query.html). Below that is used to convert the extracted year to an integer (AS INT) so a numeric comparison (>) can be used.

SELECT \*, CAST(RIGHTSTR(date, 4) AS INT) AS year FROM october\_2018\_invoices\_over\_500 WHERE year > 2019

# Changing your table, and searching for ways to do other things

Once you’ve queried your data you can generally download the results as a new table. This is one of the things that makes it appealing regarding big data: you can run a query on the large datafile in order to generate a subset which is small enough to work with in spreadsheet software if you need to.

Beyond querying your tables, however, you can also use SQL to make changes to the data itself, for example:

* Dropping columns or data from the original file
* Adding new columns of data

If you ever want to do anything in SQL and don’t know how to, just search the web – there are plenty of tutorials and discussions you can pick up from. It’s also worth checking [guides like this one](http://www.1keydata.com/sql/sql-commands.html)