SQL - Bootcamp Module

Introduction to SQL

What is SQL?

SQL (Structured Query Language) is a powerful language used to interrogate and manipulate relational databases. It is highly specialised. It is not a general programming language that you can use to write an entire program. However, SQL queries can be embedded in other programming languages to let any program work with databases. There are several different kinds of SQL, but all support the same basic statements that we will be covering today.

Relational databases

Relational databases consist of one or more tables of data. These tables have *fields* (columns) and *records* (rows). Every field has a data *type*. Every value in the same field of each record has the same *type*. These tables can be linked to each when a field in one table can be matched to a field in another table. SQL *queries* are the commands that let you look up data in a database or make calculations based on columns.

Why use SQL

Using SQL lets you keep the data separate from the analysis. There is no risk of accidentally changing data when you are analysing it. If the data is changed, a saved query can be re-run to analyse the new data.

SQL is optimised for handling large amounts of data. Using data types helps with quality control of entries - you will receive an error if you try to enter a word into a field that should contain a number. Understanding the nature of relational databases, and using SQL, will help you in using databases in programming languages and in doing similar things using programming languages such as R or Python.

Why are Librarians well suited to SQL?

Librarianship is about information management. We help sort and organise information and we help people find information. Most of us use mediated queries to help people find the information they need e.g. conducting a search via a library catalogue. With SQL, you can directly construct your database queries without the constraints (e.g. field name or search limitations) imposed by a mediated search interface. Librarians are good at searching information so don't be afraid – constructing queries using SQL is simply a different and more direct way of finding information.

Database Management Systems

There are a number of different database management systems for working with relational data. We're going to use SQLite today, but basically everything we teach you will apply to the other database systems as well (e.g., MySQL, PostgreSQL, MS Access, Filemaker Pro). The only things that will differ are the details of exactly how to import and export data and the datatypediffs.

Database Design

- Every row-column combination contains a single *atomic* value, i.e., not containing parts we might want to work with separately.
- One field per type of information
- No redundant information
    - Split into separate tables with one table per class of information
    - Needs an identifier in common between tables – shared column - to reconnect (foreign key).

Introduction to SQLite Manager

1. Make sure you installed SQLite (download and run the appropriate sqlite-tools executable from https://sqlite.org/download.html)
2. Open Firefox.
3. Add the extension for SQLite Manager (see https://addons.mozilla.org/en-US/firefox/addon/sqlite-manager/)
4. Now go to Firefox > Add-ons > SQLite Manager to start that up.

Import Some Data into a new SQLite Database

1. Download the CSV files from this figshare link.
2. In the interface, start a New Database **Database -> New Database**
3. Start the import **Database -> Import**
4. Select the file to import (articles.csv).
5. Give the table a name that matches the file name (articles, journals, licences, languages publishers), or use the default
6. If the first row has column headings, check the appropriate box
7. Make sure the delimiter and quotation options are appropriate for the CSV files. Ensure 'Ignore trailing Separator/Delimiter' is left *unchecked*.
8. Press **OK**
9. When asked if you want to modify the table, click **OK**
10. Set the data types for each field: choose TEXT for fields with text (e.g. `Title`, `Authors`, `DOI`, etc.) and INT for fields with numbers (e.g. `Citation_Count`, `Author_Count`, `Day`, etc.)

You can also use this same approach to append new data to an existing table.

Adding data to existing tables

1. Browse & Search -> Add
2. Enter data into a csv file and append

Data types

Data type	Description
CHARACTER(n)	Character string. Fixed-length n
VARCHAR(n) or CHARACTER VARYING(n)	Character string. Variable length. Maximum length n
BINARY(n)	Binary string. Fixed-length n
BOOLEAN	Stores TRUE or FALSE values
VARBINARY(n) or BINARY VARYING(n)	Binary string. Variable length. Maximum length n
INTEGER(p)	Integer numerical (no decimal).
SMALLINT	Integer numerical (no decimal).
INTEGER	Integer numerical (no decimal).
BIGINT	Integer numerical (no decimal).
DECIMAL(p,s)	Exact numerical, precision p, scale s.
NUMERIC(p,s)	Exact numerical, precision p, scale s. (Same as DECIMAL)
FLOAT(p)	Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation.
REAL	Approximate numerical
FLOAT	Approximate numerical
DOUBLE PRECISION	Approximate numerical
DATE	Stores year, month, and day values
TIME	Stores hour, minute, and second values
TIMESTAMP	Stores year, month, day, hour, minute, and second values
INTERVAL	Composed of a number of integer fields, representing a period of time, depending on the type of interval
ARRAY	A set-length and ordered collection of elements
MULTISET	A variable-length and unordered collection of elements
XML	Stores XML data

SQL Data Type Quick Reference

Different databases offer different choices for the data type definition.

The following table shows some of the common names of data types between the various database platforms:

Data type	Access	SQLServer	Oracle	MySQL	PostgreSQL
boolean	Yes/No	Bit	Byte	N/A	Boolean
integer	Number (integer)	Int	Number	Int / Integer	Int / Integer
float	Number (single)	Float / Real	Number	Float	Numeric
currency	Currency	Money	N/A	N/A	Money
string (fixed)	N/A	Char	Char	Char	Char
string (variable)	Text (<256) / Memo (65k+)	Varchar	Varchar / Varchar2	Varchar	Varchar
binary object OLE Object Memo Binary (fixed up to 8K)	Varbinary (<8K)	Image (<2GB) Long	Raw Blob	Text Binary	Varbinary

SQL Queries

Writing my first query

Let’s start by using the **articles** table. Here we have data on every article that has been published, including the title of the article, the authors, date of publication, etc.

Let’s write an SQL query that selects only the title column from the articles table.

```
SELECT title
FROM articles;
```

We have capitalized the words SELECT and FROM because they are SQL keywords. SQL is case-insensitive, but it helps for readability, and is good style.

If we want more information, we can add a new column to the list of fields, right after `SELECT`:

```
SELECT title, authors, issns, date
FROM articles;
```

Or we can select all of the columns in a table using the wildcard ‘*’

```
SELECT *
FROM articles;
```

Unique values

If we want only the unique values so that we can quickly see the ISSNs of journals included in the collection, we use `DISTINCT`

```
SELECT DISTINCT issns
FROM articles;
```

If we select more than one column, then the distinct pairs of values are returned

```
SELECT DISTINCT issns, day, month, year
FROM articles;
```

Calculated values

We can also do calculations with the values in a query. For example, if we wanted to look at the relative popularity of an article, so we divide by 10 (because we know the most popular article has 10 citations).

```
SELECT first_author, citation_count/10.0
FROM articles;
```

When we run the query, the expression `citation_count / 10.0` is evaluated for each row and appended to that row, in a new column. Expressions can use any fields, any arithmetic operators (`+`, `-`, `*`, and `/`) and a variety of built-in functions. For example, we could round the values to make them easier to read.

```
SELECT first_author, title, ROUND(author_count/16.0, 2)
FROM articles;
```

Exercise

1. Write a query that returns the title, first_author, citation_count, author_count, month and year

Filtering

Databases can also filter data – selecting only the data meeting certain criteria. For example, let's say we only want data for a specific ISSN for the *Theory and Applications of Mathematics & Computer Science* journal, which has a ISSN code 2067-2764|2247-6202. We need to add a `WHERE` clause to our query:

```
SELECT *
FROM articles
WHERE issns='2067-2764|2247-6202';
```

We can use more sophisticated conditions by combining tests with `AND` and `OR`. For example, suppose we want the data on *Theory and Applications of Mathematics & Computer Science* published after June:

```
SELECT *
FROM articles
WHERE (issns='2067-2764|2247-6202') AND (month > 06);
```

`Note that the parentheses are not needed, but again, they help with readability. They also ensure that the computer combines AND and OR in the way that we intend.`

If we wanted to get data for the *Humanities* and *Religions* journals, which have ISSNs codes `2076-0787` and `2077-1444`, we could combine the tests using OR:

```
SELECT *
FROM articles
WHERE (issns = '2076-0787') OR (issns = '2077-1444');
```

Exercise

Write a query that returns the title, first_author, issns, month and year for all single author papers with more than 4 citations

Building more complex queries

Now, lets combine the above queries to get data for the 3 journals from June on. This time, let's use IN as one way to make the query easier to understand. It is equivalent to saying `WHERE (issns = '2076-0787') OR (issns = '2077-1444') OR (issns = '2067-2764|2247-6202')` , but reads more neatly:

```
SELECT *
FROM articles
WHERE (month > 06) AND (issns IN ('2076-0787', '2077-1444', '2067-2764|2247-6202'));
```

We started with something simple, then added more clauses one by one, testing their effects as we went along. For complex queries, this is a good strategy, to make sure you are getting what you want. Sometimes it might help to take a subset of the data that you can easily see in a temporary database to practice your queries on before working on a larger or more complicated database.

When the queries become more complex, it can be useful to add comments. In SQL, comments are started by `--`, and end at the end of the line. For example, a commented version of the above query can be written as:

```
-- Get post June data on selected journals
-- These are in the articles table, and we are interested in all columns
SELECT * FROM articles
-- Sampling month is in the column `month`, and we want to include
-- everything after June
WHERE (month > 06)
-- selected journals have the `issns` 2076-0787, 2077-1444, 2067-2764|2247-6202
AND (issns IN ('2076-0787', '2077-1444', '2067-2764|2247-6202'));
```

Although SQL queries often read like plain English, it is *always* useful to add comments; this is especially true of more complex queries.

Sorting

We can also sort the results of our queries by using `ORDER BY`. For simplicity, let's go back to the articles table and alphabetize it by issns.

```
SELECT *
FROM articles
ORDER BY issns ASC;
```

The keyword `ASC` tells us to order it in Ascending order. We could alternately use `DESC` to get descending order.

```
SELECT *
FROM articles
ORDER BY first_author DESC;
```

`ASC` is the default.

We can also sort on several fields at once. To truly be alphabetical, we might want to order by genus then species.

```
SELECT *
FROM articles
ORDER BY issns DESC, first_author ASC;
```

Order of execution

Another note for ordering. We don't actually have to display a column to sort by it. For example, let's say we want to order the articles by their ISSN, but we only want to see Authors and Titles.

```
SELECT authors, title
FROM articles
WHERE issns = '2067-2764|2247-6202'
ORDER BY date ASC, first_author ASC;
```

We can do this because sorting occurs earlier in the computational pipeline than field selection.

The computer is basically doing this:

1. Filtering rows according to WHERE
2. Sorting results according to ORDER BY
3. Displaying requested columns or expressions.

Clauses are written in a fixed order: `SELECT`, `FROM`, `WHERE`, then `ORDER BY`. It is possible to write a query as a single line, but for readability, we recommend to put each clause on its own line.

Aggregation

`COUNT` and `GROUP BY`

Aggregation allows us to combine results by grouping records based on value and calculating combined values in groups.

Let’s go to the articles table and find out how many entries there are. Using the wildcard simply counts the number of records (rows)

```
SELECT COUNT(*)
FROM articles;
```

`We can also find out how many authors have participated in these articles.`

```
SELECT COUNT(*), SUM(author_count)
FROM articles;
```

There are many other aggregate functions included in SQL including `MAX`, `MIN`, and `AVG`.

Now, let’s see how many articles were published in each journal. We do this using a `GROUP BY` clause

```
SELECT issns, COUNT( * )
FROM articles
GROUP BY issns;
```

`GROUP BY` tells SQL what field or fields we want to use to aggregate the data. If we want to group by multiple fields, we give `GROUP BY` a comma separated list.

The `HAVING` keyword

In the previous lesson, we have seen the keywords `WHERE`, allowing to filter the results according to some criteria. SQL offers a mechanism to filter the results based on aggregate functions, through the `HAVING` keyword.

For example, we can adapt the last request we wrote to only return information about articles with a 10 or more published articles:

```
SELECT issns, COUNT( * )
FROM articles
GROUP BY issns
HAVING COUNT( * ) >= 10;
```

The `HAVING` keyword works exactly like the `WHERE` keyword, but uses aggregate functions instead of database fields.

If you use `AS` in your query to rename a column, `HAVING` can use this information to make the query more readable. For example, in the above query, we can call the `COUNT(*)` by another name, like `occurrences`. This can be written this way:

```
SELECT issns, COUNT( * ) AS occurrences
FROM articles
GROUP BY issns
HAVING occurrences >= 10;
```

Note that in both queries, `HAVING` comes *after* `GROUP BY`. One way to think about this is: the data are retrieved (`SELECT`), can be filtered (`WHERE`), then joined in groups (`GROUP BY`); finally, we only select some of these groups (`HAVING`).

Ordering aggregated results

We can order the results of our aggregation by a specific column, including the aggregated column. Let’s count the number of articles published in each journal, ordered by the count

```
SELECT issns, COUNT( * )
FROM articles
GROUP BY issns
ORDER BY COUNT( * ) DESC;
```

Saving queries for future use

It is not uncommon to repeat the same operation more than once, for example for monitoring or reporting purposes. SQL comes with a very powerful mechanism to do this: views. Views are a form of query that is saved in the database, and can be used to look at, filter, and even update information. One way to think of views is as a table, that can read, aggregate, and filter information from several places before showing it to you.

Creating a view from a query requires to add `CREATE VIEW viewname AS` before the query itself. For example, if we want to save the query giving the number of journals in a view, we can write

```
CREATE VIEW journal_counts AS
SELECT issns, COUNT(*)
FROM articles
GROUP BY issns;
```

Now, we will be able to access these results with a much shorter notation:

```
SELECT *
FROM journal_counts;
```

Assuming we do not need this view anymore, we can remove it from the database almost as we would a table:

```
DROP VIEW journal_counts;
```

You can also add a view using *Create View* in the *View* menu and see the results in the *Views* tab just like a table.

Exercise

1. Write a query that returns the number of articles published in each journal on each month, sorted from most popular journal to the ones with least publications each month starting from the most recent records. Save this query as a `VIEW`.

Joins and aliases

Joins

To combine data from two tables we use the SQL `JOIN` command, which comes after the `FROM` command.

We also need to tell the computer which columns provide the link between the two tables using the word `ON`. What we want is to join the data with the same journal name.

```
SELECT *
FROM articles
JOIN journals
ON articles.issns = journals.issns;
```

`ON` is like `WHERE`, it filters things out according to a test condition. We use the `table.colname` format to tell the manager what column in which table we are referring to.

Alternatively, we can use the word `USING`, as a short-hand. In this case we are telling the manager that we want to combine `articles` with `journals` and that the common column is `issns`.

```
SELECT *
FROM articles
JOIN journals
USING (issns);
```

We often won't want all of the fields from both tables, so anywhere we would have used a field name in a non-join query, we can use `table.colname`.

For example, what if we wanted information on published articles in different journals, but instead of their ISSN we wanted the actual journal title.

```
SELECT articles.issns, journal_title, title, first_author, citation_count, author_count, month, year
FROM articles
JOIN journals
ON articles.issns = journals.issns;
```

Joins can be combined with sorting, filtering, and aggregation. So, if we wanted average number of authors for articles on different journals, we could do something like

```
SELECT articles.issns, journal_title, ROUND(AVG(author_count), 2)
FROM articles
JOIN journals
ON articles.issns = journals.issns;
GROUP BY articles.issns;
```

Aliases

As queries get more complex names can get long and unwieldy. To help make things clearer we can use aliases to assign new names to things in the query.

We can alias both table names:

```
SELECT ar.title, ar.first_author, jo.journal_title
FROM articles AS ar
JOIN journals  AS jo
ON ar.issns = jo.issns;
```

And column names:

```
SELECT ar.title AS title, ar.first_author AS author, jo.journal_title AS journal
FROM articles AS ar
JOIN journals  AS jo
ON ar.issns = jo.issns;
```

The AS isn't technically required, so you could do

```
SELECT a.title t
FROM articles a;
```

but using AS is much clearer so it is good style to include it.

Exercises:

1. How many plots from each type are there?
2. How many papers have a single author? How many have 2 authors? How many 3? etc?
3. How many articles are published for each language? (Ignore articles where language is unknown).
4. How many articles are published for each licence type, and what is the average number of citations for that licence type
5. Create a view with article, journal and publisher information