

Intro to SQL for Querying Databases

Michele Tobias

Nicholas Alonzo

Nick Ulle

2024-04-18

Overview

Structured Query Language (SQL) is a programming language for interacting with relational databases. This workshop covers basic SQL keywords to view, filter, aggregate, and combine tables in a database. SQL is supported by many different database management systems. We'll focus on querying data to get to know a database and answer questions, and joining data from separate tables.

The examples in this workshop use a SQLite database, but most of the keywords are applicable to other database systems as well. The workshop also covers how to use SQLiteStudio, an integrated development environment for SQL code.

Learning Objectives After this workshop learners should be able to:

- Describe the advantages and disadvantages of using SQL with your own data.
- Use SQL queries to view, filter, aggregate, and combine data.
- Combine SQL keywords to develop sophisticated queries.
- Use SQL queries to solve problems with and answer questions about data.
- Identify additional resources for learning more about SQL, such as how to use SQL with the R programming language.

Prerequisites No prior programming experience is necessary. We recommend learners either attend or review the written materials for DataLab's Overview of Databases & Data Storage Technologies workshop.

Before the workshop, learners should:

- Install SQLiteStudio using the install guide and verify that it runs.
- Download the file `2024-04-09_library-data.sqlite` from this link.

NOTE:

If you have a Mac (OSX), you will need to right-click on the SQLiteStudio installer and select open. If you open the installer regularly, the Mac operating system will block the installer from running.

Please see these recommendations for making SQLiteStudio easier to read, particularly for those with low vision and those who use a screen reader.

Concepts

What is a Relational Database?

A **relational database** is a collection of tables (organized in rows and columns of data) that are related to each other in some way.

Database tables are analogous to CSV files, spreadsheets in Excel, or data frames in programming languages like R or Python.

Ideally each table can be connected to another table by a column that is present in both tables. That column may have different numbers of observations in each table, but the values will match up. This column is called a **key**. For example, a key commonly used on campus is your student or employee ID number.

Let's look at an example dataset of fictional student data with data about courses, grades, and employment. Can we say anything about the relationship between course grades and employment based on this data?

Table: Student

ID	Name
123	Jane Smith
456	Maria Martinez
789	Paul Jones

Table: Courses

ID	Course	Grade
123	Calculus	A-
456	Calculus	A
789	Calculus	C+
123	Data Science	A-
456	Data Science	B
789	Data Science	B-

Table: Employment

ID	Position	Employer	HoursPerWeek
123	Student Assistant	University Research Lab	5
456	Customer Service	Alumni Center	5
456	Research Assistant	University Research Lab	15
789	Student Assistant	University Research Lab	10
789	Stock Room	Medical Supplier	20
789	Customer Service	Alumi Center	15

What is SQL?

SQL stands for **structured query language**. SQL is a programming language that allows you to request (query) information from a database using a standard set of keywords. You can pronounce SQL as “ess cue ell” or “sequel”.

What kinds of questions can SQL answer?

SQL excels at extracting and combining information from large datasets. Some questions you might ask with SQL include:

- How many items are there in my data with a specific label?
- What are the unique values in a given column?
- Which records (rows) relate to a specific time period in my data?

What is a Relational Database Management System?

A **relational database management system** (RDBMS) is a software system that creates, updates, and manages relational databases as well as managing user's access to the database. There are many different systems available. For instance, you may have heard of MySQL, Postgres, and Microsoft SQL Server.

For this workshop, we'll use SQLite, which is a simple and widely-used RDBMS. It runs on Windows, MacOS X, and Linux with no setup necessary!



Every RDBMS has its own implementation or “dialect” of SQL. In other words, the set of SQL keywords supported differs slightly from one RDBMS to another, and sometimes queries have to be written differently, but the basics are the same. Details about the supported keywords for a given RDBMS can be found in that system's documentation. How does this impact you working with any given SQL RDBMS? When you're searching for help with syntax, include the version of SQL you're working with. The keywords covered in this workshop are supported by most systems.

Some RDBMS allow you to add functions with extensions. For example, the PostGIS extension adds keywords to PostgreSQL to all you to work with location information to do spatial analysis.

Advantages & Disadvantages of SQL

SQL has major advantages in several areas important to researchers:

- Efficiency
 - Write a few lines of code rather than lots of manual data manipulation
 - SQL is meant for data manipulation
- Reproducibility
 - save queries as a record of your workflow
 - re-run code with updates
- Work with large amounts of data
 - Typically faster to run a process in a database than in a spreadsheet
 - Store lots of data (compare with Excel's row limits)
- Data management
 - One database file is the equivalent to many, many spreadsheet files (like csvs or xlsx files)
 - Write a query instead of making a new files or tabs

What does SQL not do well?

- Most RDBMS do not visualize data, however, you can connect your database to visualization tools to perform these kinds of tasks seamlessly.
- The SQL language is designed for data querying not data analysis. If you want to run statistics on your data you can connect to your database from a programming language like R or python, or from statistical software.
- SQL assumes you work with tabular data. If your data is another type - for example graph data or tree data - you might want to explore other database types.

The Library Checkouts Database

We'll be working with the Library Checkouts Database, a subset of real data from the UC Davis Library. This includes information like:

- Books and their details: title, author, publication year, etc...
- Patrons who check out items: ID number, what user group they belong to, and the date their library card was created
- Checkouts of books by who and when

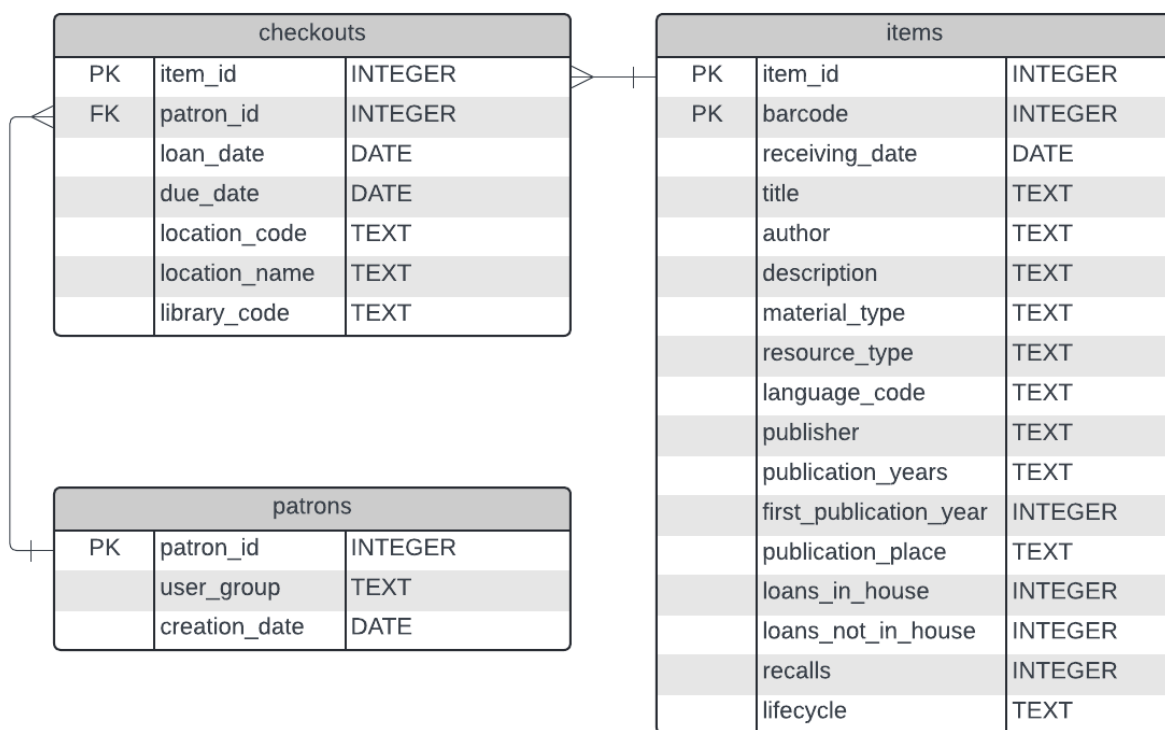
Why would a library want to track this information in a database?

- Data management
 - Centralized location for all data being tracked
 - Computerized and reduces manual processes
- Use SQL to get insight
 - With SQL the library can answer questions like:
 - * Which books are being checked out?
 - * What types of books are being checked out?
 - * Which books are overdue?

Entity Relationship Diagrams

An **entity relationship diagram** ERD is a visual representation of a relational database. ERDs help with understanding what data is available, how the data is stored, and how tables and columns are related. These details are important for determining what types of questions you can answer with SQL!

Here's an ERD for the Library Checkouts database:



Lets break down the components of the ERD:

1. *Entities*, visualized as a rectangle, represent the tables in the database. The name of the table is written at the top in dark gray.
2. *Attributes* represent the columns in the database.
 1. The names of the columns are written in the middle column.

2. *Data Types*: Each attribute is made up of a certain *data type*. The most common data types you'll interact with are numeric, string, date, or boolean. This information is on the right.
3. *Primary Key (PK)*: This is a column(s) that uniquely identifies a row in a table. Key designations are written on the left.
4. *Foreign Key (FK)*: This is a column that references a primary key. It's used to identify a relationship between tables.
3. *Relationships* between tables are represented with lines connecting one entity to another
 1. The symbols at the end of the lines represent *cardinality*, the number of rows between two database tables.

NOTE:

The Library Checkouts ERD was made with the diagramming software Lucidchart. Lucidchart also does an excellent job of breaking down Entity Relationship Diagrams here.

Data Definitions

Below are the data definitions of the tables and columns in the Library Checkouts Database.

Table 4: **patrons**: Users with checkout privileges

column	description	data type
patron_id	unique ID for each user of the library	INTEGER
user_group	the type of borrower - for example, Alumni, Faculty, Undergraduate Students, etc.	TEXT
creation_date	the date lending privileges were created	DATE

Table 5: **items**: All the items (books, etc.) in the library system

column	description	data type
item_id	the unique ID for each item	INTEGER
barcode	the barcode for each item	INTEGER
receiving_date	the date the item became a part of the library's collection	DATE
title	the title of the item	TEXT
author	the names of the authors of the item	TEXT
description	additional information about the title	TEXT
material_type	the type of the item (book)	TEXT
resource_type	the type of resource, for example, Book - Physical, Microforms, etc.	TEXT
language_code	a three letter code indicating the language of publication	TEXT
publisher	the name of the publisher	TEXT
publication_years	the years the item was published	TEXT
first_publication_year	the year the work was first published	INTEGER
publication_place	a list of cities the item was published in	TEXT

column	description	data type
loans_in_house	number of loans internal to the library	INTEGER
loans_not_in_house	number of loans external to the library	INTEGER
recalls	the number of recalls on an item	INTEGER
lifecycle	if a book is available for circulation	TEXT

Table 6: **checkouts**: A log of when a user checks out a book from the library

column	description	data type
item_id	the unique ID for each item	INTEGER
patron_id	unique ID for each user of the library	INTEGER
loan_date	the date the item was checked out	DATE
due_date	the date the item is due	DATE
location_code	a three character code for the location the item can be found	TEXT
location_name	the location the item can be found	TEXT
library_code	a short text code indicating the library that holds the item	TEXT

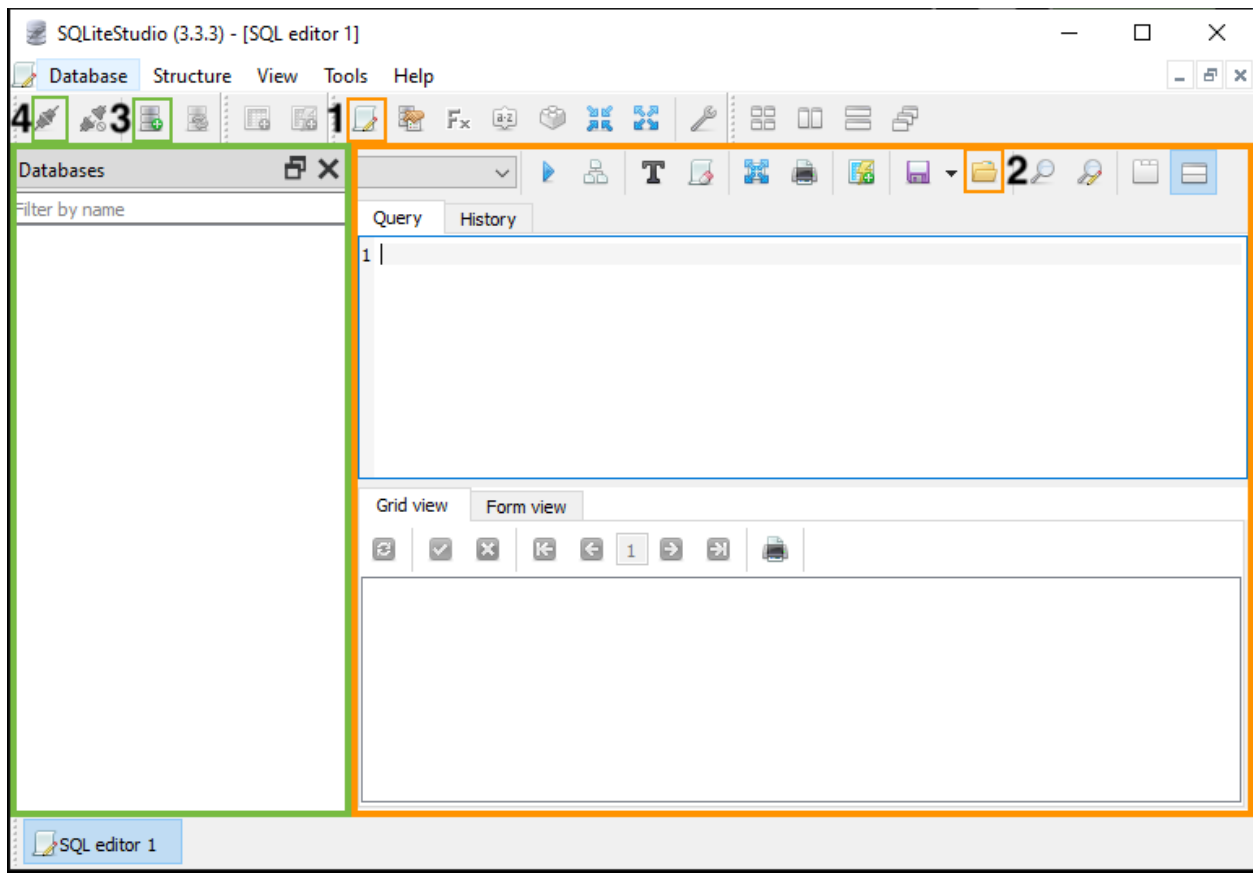
Database Set Up

SQLiteStudio



To work with our database, we'll use SQLiteStudio. It's a free, open source, multi-platform desktop application for browsing SQLite databases and writing SQL queries.

SQLiteStudio has a number of tools and panes to help you interact with your data:



You can view databases in the pane outlined in green. You can also write and run queries in the editor pane outlined in orange.



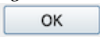
Create a Database

We are going to work with a database that has already been created for us. BUT, To learn about how to create your own database from scratch using data stored in CSV files, see:

- Create a New Database
- Geeks for Geeks' tutorial for how to Import a CSV File Into an SQLite Table - see the "Using SQLiteStudio" section near the middle of the page.

Load the Database

Let's connect to the database that we'll be using for this workshop:

1. Click the "Add a database" icon  and the *Database Dialog Window* will pop up.
2. Click the "Browse for existing database file on your local computer" icon 
 - Your computer file explorer window will pop up
3. Navigate to the `2024-04-09_library-data.sqlite` file on your computer and double-click
 - The *Database Dialog Window* will appear again
4. Click the "OK" button  at the bottom right
 - The database file will load to the left under the *Databases Pane* of the interface (outlined in green)
5. Click the database name under the *Databases Pane* to highlight



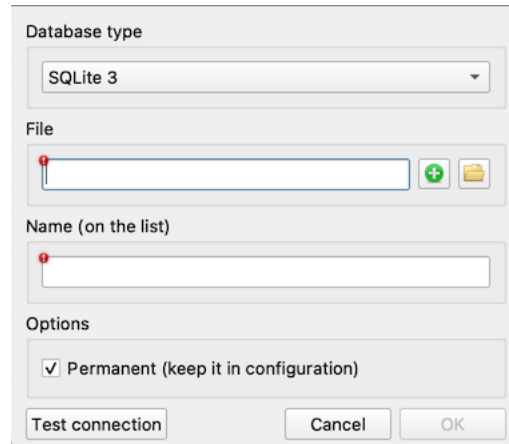



Figure 1: Database Dialog Window

6. Click the “Connect to the database” icon 
 - You are now connected to the database and can execute SQL to the database!

Open the SQL Editor

To write SQL queries, we need to open the SQL Editor. With the database selected (highlighted) in the list of databases on the left side of the screen, click the *Open SQL Editor* icon .

Saving Scripts

You can save a text file with a .sql extension that contains SQL commands to run as a script. Like scripts in other programming languages, the commands run from top to bottom. For today’s workshop, writing a script that we have to execute over and over doesn’t make sense. We’re still learning and we’ll need to run individual queries over and over as we correct our mistakes. For data cleaning tasks that you repeat every time you add new data, for example, these scripts can save a lot of time and make your process repeatable.

Hands-On with SQL Code

We just learned that SQL is a language that allows us to interact with and manage a database. Let’s learn some SQL queries to get some hands-on experience.

Viewing Data

SELECT & FROM

We’re ready to write our first queries! The most common query you’ll use is the SELECT statement. In it’s most basic form, it shows you the data in a table, but you can add option to customize the view you get back. Let’s try! Type this query into the query box:

```
SELECT * FROM items;
```

Now click the *Execute all* button. 

This query asks the database to select everything (* means “everything”) from the table *items*. It ends with a semicolon to tell the database that this is the end of our request.

SQL doesn’t care if you add extra white space (spaces, tabs, or new lines) to your query to make it easier to read. All that matters is that you use the correct keyword structure and end your query with a semicolon (;).

Because of this, the query below does exactly the same thing as the first query we ran.

```
SELECT
*
FROM
items;
```

SQL ignores capitalization, spaces, and new lines in a query. Some tools which use SQL also ignore semicolons. However, it's conventional to:

- Write SQL keywords (`SELECT`, `FROM`, and so on) in UPPERCASE
- Write table and column names in lowercase
- Write a semicolon ; at the end of the query

Selecting Columns

The *items* table has a lot of columns. What if we don't want to see all of the columns in the table? We can ask for just the columns we want to see. Let's get just the item ID, title, and material type.

```
SELECT
    item_id,
    title,
    material_type
FROM items;
```

NOTE:

You can order the columns however you'd like in the `SELECT` statement and select a column multiple times.

Unique Values

What if we now want to know what all the possible languages are in our data set? We could scroll through the results and try to keep track of unique values, but that is tedious - and we'll likely miss some, especially if they are uncommon.

Instead we can use the `SELECT DISTINCT` keywords on one or more columns to show all the unique values.

Let's look at the *items* table again and see which languages our items are published in.

```
SELECT DISTINCT language_code
FROM items;
```

If we wanted to see unique combinations across multiple columns, we just add the columns we're interested in. Let's see which languages each publisher publishes in:

```
SELECT DISTINCT
    language_code,
    publisher
FROM items;
```

Ordering Results

Sometimes sorting data is useful for understanding the output of a query.

With SQL, you can sort on one or more columns with a combination of ascending or descending order using the `ORDER BY` keyword.

Let's sort the *items* table by the year of first publication.

```
SELECT
    title,
```

```

    author,
    first_publication_year
FROM items
ORDER BY first_publication_year;

```

By default, `ORDER BY` sorts in ascending order. We can sort in descending order to get users born more recently by adding `DESC` after the column.

```

SELECT
    title,
    author,
    first_publication_year
FROM items
ORDER BY first_publication_year DESC;

```

CHALLENGE:

How would you sort *items* by author in descending order?

Limiting Number of Rows

Sometimes you'll be working with a large table to analyze with lots of columns and rows. You can use `LIMIT` to reduce the number of rows the query returns to give you a snapshot of the data you're working with. Limiting the output is particularly useful when you are building a complex query on a large amount of data. Limiting makes the query quicker so you can see an example of the results more quickly for troubleshooting. Once your limited query returns the results you want, then you can run the full query.

```

SELECT *
FROM items
LIMIT 10;

```

Commenting

As we're writing queries, sometimes we want to write helpful comments to ourselves and others. There's two ways to write comments so that text won't be interpreted as SQL.

1. Single line comments: text following two dashes `--`: `-- comment here`
2. Multiline comments: text between the characters `/* */`: `/* comment here */`

```

/*
all items in the library system
sorted by first publication year
*/
SELECT
    title,
    author,
    first_publication_year
FROM items
ORDER BY first_publication_year DESC; -- sort most recent to the top

```

Filtering Data

Now that we've seen some ways to view our data, let's learn how we can filter our data. This is really the core of SQL, where we can start to answer our own questions about the data!

We use the `WHERE` clause to filter rows of a query by specifying one or more conditions. `WHERE` in this case does not indicate a location. The table below shows comparison operators that can be used and combined with `WHERE` to create conditions, some of which you may have seen before in other programming languages.

Comparison Operator	Description
=	equals
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
<> or !=	not equal to

In general, the type of data on each side of the operator needs to be the same: compare numbers to numbers, text to text, and so on. When working with **text** or **date** data, it's necessary to wrap values in single quotes. For instance:

- 'text_value' is a text value
- '2000-01-01' is a date

Now let's write a query to find items checked out from Shields Library:

```
SELECT *
FROM checkouts
WHERE library_code = 'SHLDS';
```

NOTE: Some relational database query tools require LIKE for text comparisons rather than =.

SQL also provides a variety of arithmetic operators for working with numeric data:

Arithmetic Operators	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

AND & OR Operators

Often we'll need to filter data based on more than one condition. We can ask **WHERE** to check multiple conditions with the keywords:

- **AND**, meaning a row must satisfy both conditions
- **OR**, meaning a row must satisfy at least one of two conditions

These are logical operators in SQL that evaluate to TRUE or FALSE.

Let's find all the items that have been checked out from Shields Library and the Law Library.

```
SELECT *
FROM checkouts
WHERE library_code = 'SHLDS' AND
      library_code = 'LAW';
```

You might have guessed that this query would return no records because a physical book cannot be checked in to two libraries at once.

Let's find all the items that have been checked out from the New Books shelf *or* Shields Library.

```
SELECT *
FROM checkouts
WHERE library_code = 'SHLDS' OR
      library_code = 'LAW';
```

This query returns many books because we're asking for all the items in both libraries. These last two queries should help illustrate the difference between **AND** and **OR**.

NOTE:

AND & **OR** will follow the order of operations. To use **AND** & **OR** in the same query, wrap parenthesis around the **OR** conditions.

IN Operator

Sometimes we find ourselves writing multiple **OR** conditions on the same column which can make our query look more complex than it really is. In this scenario we can condense multiple **OR** conditions using **IN**.

Let's rewrite our previous query to use **IN**.

```
SELECT *
FROM checkouts
WHERE library_code IN ('SHLDS', 'LAW');
```

BETWEEN Operator

We can also filter using ranges of values with **BETWEEN**. This is handy when you're working with numerical or date values and you don't want to list out all possible values to meet your conditions.

Let's write a query to find all of the due dates for 2020.

```
SELECT *
FROM checkouts
WHERE due_date BETWEEN '2020-01-01' AND '2020-12-31';
```

CHALLENGE:

How many items checked out (loan_date) for 2019? How many in 2020?

LIKE Operator

The **LIKE** keyword tests whether text values match a given pattern. There are two different wildcard characters that you can use in the pattern:

- **_** matches exactly one character
- **%** matches zero or more characters

Let's look at an example to indicate we want to match the beginning of a string, but the end is allowed to vary. To show this, let's find items with a publisher name that starts with "Springer" but can end with anything.

```
SELECT *
FROM items
WHERE publisher LIKE 'Springer%';
```

NOTE:

The wildcard **%** can be used multiple times in one pattern.

You can also use regular expressions in SQLite to match in more complicated situations. Read more about using regular expressions [here](#).

The **LIKE** keyword differs between dialects of SQL, so it's also a good idea to check the documentation for your RDBMS before using **LIKE**.

CHALLENGE:

Write a query that selects all the items whose publisher has the word "University" in their name?

IS NULL Operator

So far we've worked with complete data, but how do we work with missing data? In databases, `NULL` means missing data. `IS NULL` is used to test whether there is missing data in a column.

Let's look at an example to find items where the year of first publication is missing.

```
SELECT *
FROM items
WHERE first_publication_year IS NULL;
```

CHALLENGE:

How would you write a query to find items with a missing author name?

NOT Operator

There will be times where we want to find only the rows that do not satisfy some condition. To do this, use `NOT` combined with other operators:

- `NOT IN`
- `NOT BETWEEN`
- `NOT LIKE`
- `IS NOT NULL`

Below is a query to find items that **do not** have a certain number of recalls - in this case, we're excluding items with 0, 1, or 3 recalls.

```
SELECT *
FROM items
WHERE recalls NOT IN (0, 1, 3);
```

Why would you do a query like this? Our example seems a little arbitrary, but imagine you're trying to troubleshoot an issue with data entry. Maybe you suspect something odd happened with recalls that are equal to 2 or greater than 3.

Aggregating Data

We've just looked at a number of ways to filter data, but now let's look at some ways to aggregate data.

Count

Suppose we want to find out how many items we have in the items table.

We can count the total number of rows in a table using the `COUNT` function. The function takes a column or `*` as an argument, but the argument doesn't actually affect the count.

Here's how we can use `COUNT` to answer our question:

```
SELECT
    COUNT(item_id)
FROM items;
```

NOTE:

You can combine `DISTINCT` with `COUNT` using `COUNT(DISTINCT column_name)` to get a unique count of values in a column when duplicate values exist.

CHALLENGE:

Find the total number of patrons that have checked out a book.

Renaming/Aliasing Columns In the previous query, notice that the name of the column in the result is `COUNT(id)`, which isn't easy to use in subsequent SQL queries or with other data programming tools.

We can use **AS** to rename or **alias** a column in the result of the query. This is handy if you're planning to export the result for future use, especially if you're sending it to someone else.

In our last query, let's rename the column to `total_items`:

```
SELECT
    COUNT(item_id) AS total_items
FROM items;
```

Average

The **AVG** function returns the average value of a numeric column. Let's find the average number of recalls placed on items:

```
SELECT
    AVG(recalls) AS avg_recalls
FROM items;
```

Sum

We can also sum the values in a numeric column with the **SUM** function. Let's find the total number of loans from outside:

```
SELECT
    SUM(loans_not_in_house) AS outside_loans
FROM items;
```

Grouping Data

So now you've seen several functions working on a single column, but we sometimes want to summarize our data in more sophisticated ways. Let's see what grouping can do for our data. Let's write a query that counts the number of books checked out at each library:

```
SELECT
    library_code,
    COUNT(item_id) AS books_checked_out
FROM checkouts
GROUP BY library_code;
```

Notice here how we asked for two columns - the `library_code` and the count of `item_id`.

CHALLENGE:

You can also **GROUP BY** more than one column by listing the columns to group by with each column name separated by a comma. How would you find the total number of times a patron checked out an item at each library? Note that -1 is the missing value for `patron_id`.

Having

HAVING is similar to **WHERE**, but it specifically works with **GROUP BY**. Perhaps we're only interested in days that had more than 100 checkouts. Let's see what that looks like:

```
SELECT
    loan_date,
    COUNT(item_id) AS books_checked_out
FROM checkouts
GROUP BY loan_date
HAVING COUNT(item_id) > 100
```

```
ORDER BY books_checked_out DESC;
```

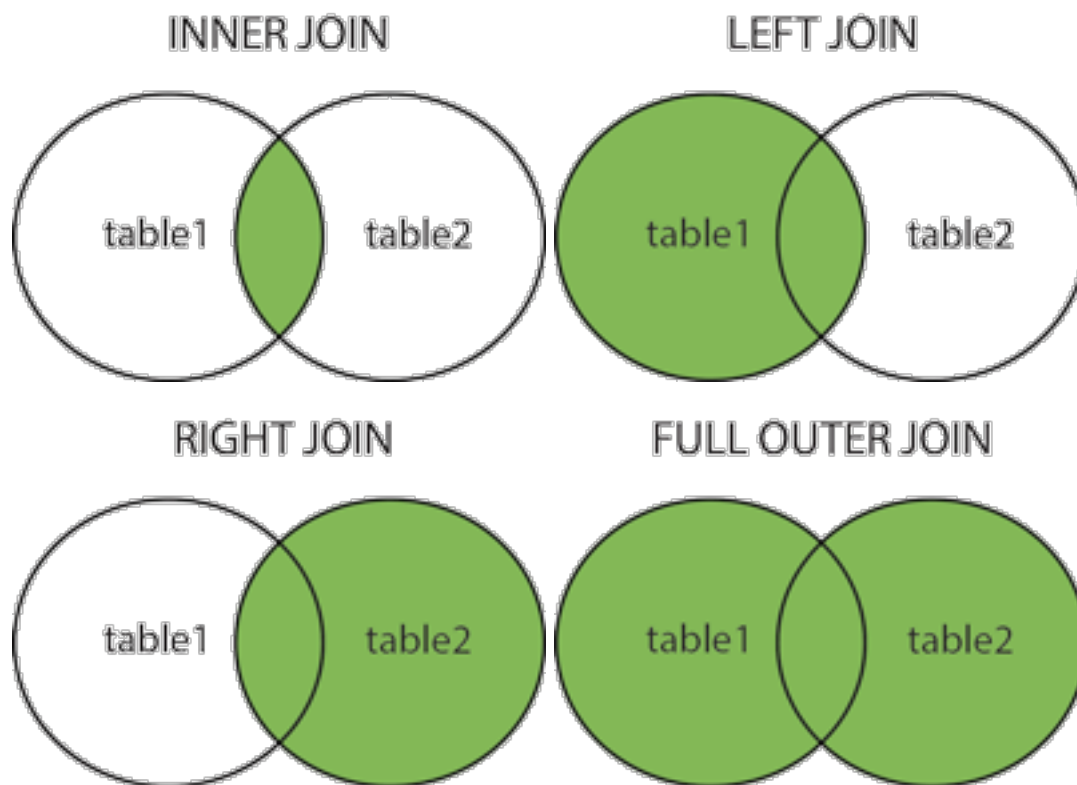
Now we've seen how we can use functions to aggregate data and how grouping data can give us meaningful insights. There are, of course, other functions available in SQL and we can't go over all of them here, but now you've seen how they work and can apply your knowledge to new functions you find.

Joining Data

Joining tables allows us to combine information from more than one table into a new table. The tables need to have a **key** column to be able to link the tables together. A key is a column that contains information that allows it to relate to information in another table. In our Library Checkouts ERD, the *item_id* column in *items* is a key column that links to *item_id* in *checkouts*.

JOIN Types

SQL has 4 main kinds of joins:



NOTE:

The above images come from the W3Schools' SQL join page, an excellent resource for learning more about SQL.

What kinds of joins are there?

- **INNER JOIN:** Returns rows that have matching values in both tables; it gets you what's in the middle of the venn diagram.
- **LEFT JOIN:** Returns all rows from the left table, and the matched rows from the right table; the "left" table is the first table you write in the query.
- **RIGHT JOIN:** Returns all rows from the right table, and the matched rows from the left table; the "right" table is the second table you write in the query or the "join" table.

NOTE:

This is not supported in SQLite, however switching the tables in a `LEFT JOIN` will emulate a `RIGHT JOIN`.

- **FULL OUTER JOIN:** Returns all rows when there is a match in either left or right table; nulls are generated in the table when a row in one table doesn't have a match in the other table.

NOTE:

This is not supported in SQLite, however you can emulate a `FULL OUTER JOIN` described here.

JOIN steps

JOIN queries typically take this form:

```
SELECT
    left_table.column1,
    right_table.column1,
    ...
FROM left_table
    [INNER | LEFT] JOIN right_table ON left_table.key_column = right_table.key_column;
```

Below are the steps for writing a JOIN:

1. **SELECT** the columns we want in the output. Be sure to refer to the columns using their table name (table.column) to disambiguate columns with the same name in two or more tables, like our *item_id* columns.
2. The **FROM** statement indicates which table to start with (this is our “left” table).
3. The **JOIN** statement indicates which table should get joined (this is our “right” table). Before the keyword **JOIN**, you can specify the direction of the join such as **INNER**, **LEFT**, or **RIGHT**.
4. Finally, we indicate which columns the join should be based on with **ON**. The columns you choose here should contain information that allows you to match records (rows) between the two tables.

INNER JOIN

Let's try an **INNER JOIN** to see how this works:

```
SELECT
    items.item_id,
    items.title,
    checkouts.item_id,
    checkouts.due_date
FROM items
INNER JOIN checkouts ON items.item_id = checkouts.item_id;
```

We interpret the **INNER JOIN** query as, “all books that have been checked out.”

LEFT JOIN

Now let's try a **LEFT JOIN**:

```
SELECT
    items.item_id,
    items.title,
    checkouts.item_id,
    checkouts.due_date
FROM items
LEFT JOIN checkouts ON items.item_id = checkouts.item_id;
```


We interpret the `LEFT JOIN` query as, “all books and if they have been checked out or not.”

CHALLENGE:

Can you write a query that contains the title of the books and the ID of the patrons that checked them out?

Subqueries

So far we’ve been working with one `SELECT` statement, but we can actually combine multiple `SELECT` statements using subqueries. Subqueries are nested queries enclosed in parentheses that can be used with other keywords like `JOIN` and `WHERE`. Below are 2 examples of these use cases.

You can think of a subquery as a process where you write a query to create a table, then query the table you just constructed. This can be especially helpful with large complex tables where simplifying helps you understand the query better, or when you need to complete a multi-step query and don’t want to make extra tables or views (something we’ll cover in the next sections).

Let’s first look at a subquery in the `WHERE` clause:

```
-- main query
SELECT *
FROM checkouts
WHERE item_id IN (

    -- subquery

    SELECT item_id
    FROM items
    WHERE resource_type = 'Microforms'
);
```

We retrieved microforms in a subquery. We then use the results of the subquery to filter the *checkouts* table using *checkouts.item_id*. In the end we get, “all checked out microforms.”

NOTE:

When writing a subquery with `WHERE` and `IN`, the subquery must select only one column for `IN` to filter on.

Now let’s look at a subquery with `JOIN`:

```
SELECT
    -- get the columns we want from both tables
    checkouts.patron_id,
    microforms.title,
    checkouts.loan_date,
    checkouts.due_date
FROM checkouts
INNER JOIN (
    -- subquery to get all the items in microform format
    SELECT *
    FROM items
    WHERE resource_type = 'Microforms'
) AS microforms ON checkouts.item_id = microforms.item_id;
```

We retrieved microforms in a subquery, just like before. Next, we write the subquery after `JOIN` and alias it *microforms*. Finally, we join on the keys and return the columns we want in the main `SELECT` statement. Both subquery examples are just different ways to get to the same result, “checked out hardcover books.”

NOTE:

Why would you want to make such a complex query? First, the sub query can help us limit large or complex tables, or join data before we query it. Second, subqueries give you the ability to create a “new” table on the fly, even if that table didn’t exist in the database before, without having to save it.

Saving Query Results

There will be times when we want to save the results of a query so we can reuse it later when needed. Two commands to save a query as a new database object follow:

1. **CREATE TEMPORARY TABLE**

- This is a new table added to the database, just like the tables you’ve been working with, except it is only available in the current session. You typically do this to break down a complex problem into intermediate steps and pass your saved results to the final query.

2. **CREATE VIEW**

- A view is simply a saved query that can be executed when called. The query you save will usually be made up of multiple tables with added conditions if needed. You can use it in pretty much the same way you would a table. The only major difference is that a view, because it is updating from other tables, is not able to be edited.

If we want to create a temporary table, we just need to add **CREATE TEMPORARY TABLE our_new_table_name AS** at the beginning of the query (adding in our own table name, of course). This is what it looks like:

```
CREATE TEMPORARY TABLE mircoform AS
SELECT
    -- get the columns we want from both tables
    checkouts.patron_id,
    microforms.title,
    checkouts.loan_date,
    checkouts.due_date
FROM checkouts
INNER JOIN (
    -- subquery to get all the items in microform format
    SELECT *
    FROM items
    WHERE resource_type = 'Microforms'
) AS microforms ON checkouts.item_id = microforms.item_id;
```

At the time of writing, SQLiteStudio has a bug where accessing temporary tables is complicated. The good news is that for the most part, you probably won’t need this option. You can use subqueries instead or use a View if you know you’ll need continued access to the results of a query.


In much the same way we made the new table, we can make a view:

```
CREATE VIEW mircoform AS
SELECT
    -- get the columns we want from both tables
    checkouts.patron_id,
    microforms.title,
    checkouts.loan_date,
    checkouts.due_date
FROM checkouts
INNER JOIN (
    -- subquery to get all the items in microform format
    SELECT *
    FROM items
```

```
WHERE resource_type = 'Microforms'
) AS microforms ON checkouts.item_id = microforms.item_id;
```

Connecting Databases to Other Analysis Tools

Export Tables SQLiteStudio has the ability to export out query results to CSV files for further analysis.

Click the “Export” icon  at the top of the interface and follow the export dialog.

What do you want to export?

☐ A database
 ☐ A single table
 ☒ Query results

Programming Language Connections Programming languages like R and Python can connect to, read data from, and query your SQL database from your script through additional libraries.

Data Management

Update Tables

You might have noticed at the beginning the *items* table has *NULL* values across different columns. We can fix this fairly easily, but we need to be careful. It’s challenging to undo something in a database so we want to be sure we’re doing it right. Let’s update the *NULL* values in the *items.receiving_date* column to “N/A”.

It’s first helpful to write a query to be sure these are the rows you want to update:

```
SELECT *
FROM items
WHERE receiving_date IS NULL;
```

Once you’ve confirmed, the below statement updates the *NULL* values to “N/A”

```
UPDATE items
SET receiving_date = 'N/A'
WHERE receiving_date IS NULL;
```

The *SET* keyword specifically targets just the *receiving_date* column and replaces *NULL* values with “N/A” when the condition is met in the *WHERE* clause. It leaves the other values alone. If the *WHERE* clause is removed, it will set all values in the whole column to “N/A” overwriting the users address, so proceed with caution!

Add & Populate a Column

Sometimes we want to make a new column and add data into it. Let’s make a new column called *year* in the *patrons* table and populate it with the year parsed from the *creation_date* column.

First we’ll add the column and set the default value to “N/A”:

```
ALTER TABLE patrons
ADD year INTEGER DEFAULT 'N/A';
```

NOTE:

The *DEFAULT* argument is optional, but if you leave it blank, it will make the default value *NULL*.

Now we update all values with the results of a string parsing cution that returns the year from our *created_date* string. SQLite has time-date functions, but the date format of our data is not the format SQLite prefers, so when something like this happens, you need to get a little creative to get the information you want.

```
UPDATE patrons
SET year = substr(creation_date, -4, 4)
WHERE creation_date IS NOT NULL;
```

The function `substr()` creates a substring from a string object - in this case, our *creation_date* string. The second argument, `-4`, indicates the position to start the substring from. Negative values tell the function to start from the right side of the string (or the end of the string) rather than the left. Finally, the third argument indicates how many characters to include. We chose `4` because our date string has a 4 digit year.

Want to see what our update query did?

```
SELECT * FROM patrons;
```

Conclusion

We covered a wide variety of SQL processes you might need in setting up a database and querying data. Did we cover everything you might need to know? Of course not. It's only a 2 hour workshop and SQL is a big language, but we've learned enough terminology and seen enough typical workflows for you to get started. To help you learn more and expand your SQL skills, we've assembled a list of resources in the Resources section of the reader.

Resources

DataLab's Spatial SQL Workshop

W3Schools SQL Materials - This is an excellent reference for SQL syntax with a fun "try it yourself" feature.

Software Carpentry's SQL Novice Workshop

Clark Fitzgeralds & Nick Ulle's SQL Workshop

Clark Fitzgeralds & Nick Ulle's SQL Cheatsheet

Lucidchart's Entity Relationship Diagrams Overview

Working with SQL databases and queries in R:

- RStudio's Database Queries with R
- University of Michigan's Stats 701 Class Notes