

SQL

Jo Hallett

November 24, 2025



Last time

We did some database theory, and drew some pretty doodles

This time

Lets write code!

SQLite 3

The database we're going to be using in this course is SQLite3

```
sqlite3
```

```
SQLite version 3.44.2 2023-11-24 11:41:44
```

```
Enter ".help" for usage hints.
```

```
sqlite>
```

(Do check out the `.help` command... there are some useful things to make output more readable!)

```
.headers on
```

```
.mode column
```

Why I like SQLite...

- ▶ It's fast
- ▶ Very little set up
 - ▶ Just point it at your database go
 - ▶ (Do ask the TAs about the hell of setting up MariaDB ; -)
- ▶ It's relatively unsurprising
- ▶ The docs are really good!
- ▶ It's installed almost everywhere

Query Language Understood by SQLite — Mozilla Firefox

File Edit View History Bookmarks Tools Help

Query Language Under

https://www.sqlite.org/lang.html

Home About Documentation Download License Support Purchase Search

SQL As Understood By SQLite

SQLite understands most of the standard SQL language. But it does [omit some features](#) while at the same time adding a few features of its own. This document attempts to describe precisely what parts of the SQL language SQLite does and does not support. A list of [SQL keywords](#) is also provided. The SQL language syntax is described by [syntax diagrams](#).

The following syntax documentation topics are available:

aggregate functions	DELETE	ON CONFLICT clause
ALTER TABLE	DETACH DATABASE	PRAGMA
ANALYZE	DROP INDEX	REINDEX
ATTACH DATABASE	DROP TABLE	RELEASE_SAVEPOINT
BEGIN TRANSACTION	DROP TRIGGER	REPLACE
comment	DROP VIEW	RETURNING clause
COMMIT TRANSACTION	END TRANSACTION	ROLLBACK TRANSACTION
core functions	EXPLAIN	SAVEPOINT
CREATE INDEX	expression	SELECT
CREATE TABLE	INDEXED BY	UPDATE
CREATE TRIGGER	INSERT	UPSERT
CREATE VIEW	JSON functions	VACUUM
CREATE VIRTUAL TABLE	keywords	window functions
date and time functions	math functions	WITH clause

The routines `sqlite3_prepare_v2()`, `sqlite3_prepare()`, `sqlite3_prepare16()`, `sqlite3_prepare16_v2()`, `sqlite3_exec()`, and `sqlite3_get_table()` accept an SQL statement list (sql-stmt-list) which is a semicolon-separated list of statements.

sql-stmt-list:

Each SQL statement in the statement list is an instance of the following:

sql-stmt:

SQL

To query most relational database we use a language called SQL
Query language for asking questions about databases from 1974

- ▶ Standardized in 1986 in the US and 1987 everywhere else
- ▶ Still the dominant language for queries today

Not a general purpose programming language

- ▶ Not Turing complete
- ▶ Weird English-like syntax

Standardized?

You would be so lucky!

- ▶ In theory, yes
- ▶ In practice, absolutely not

Every database engine has small differences...

- ▶ Some have quite big ones too!

Lots have differences in performance

- ▶ SQLite is good with strings, most others prefer numbers

Managing these differences used to be an entire degree/job in its own right!

- ▶ Now we just manage databases badly!

I'll try and stick to relatively conventional syntax...

- ▶ But if you find something doesn't work, check your database's documentation

Convention says SQL keywords are written in CAPITALS

- ▶ But it doesn't actually matter... you do you.

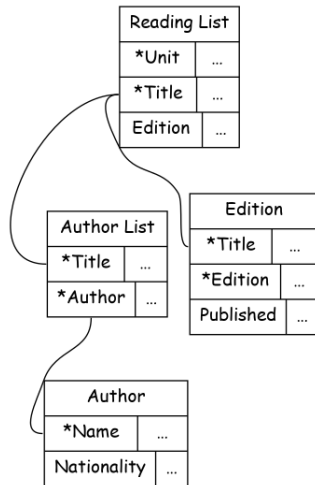
More than one indentation style too

- ▶ I like the Haskell-esque one
 - ▶ Makes it easy to delete/add lines and not have to fix up commas as you go
 - ▶ As with C, you do you (but if you write it all on one line... eww).

Reading lists

Last time we were modelled a reading list database...

- Let's code it in SQL



CREATE TABLE

Each of the entities we described is going to become a separate table

- ▶ (We'll insert rows later to create the actual data)

Lets start with the author table

```
CREATE TABLE author  
( name TEXT  
  , nationality TEXT  
) ;
```


DROP TABLE

What about if we want to delete it?

```
DROP TABLE author;
```

But it'll throw an error if you haven't already created the table.

```
DROP TABLE IF EXISTS author;  
  
CREATE TABLE IF NOT EXISTS author  
( name TEXT  
  , nationality TEXT  
) ;
```

And all the rest...

```
CREATE TABLE IF NOT EXISTS readinglist
( unit TEXT
, title TEXT
, edition INTEGER
);

CREATE TABLE IF NOT EXISTS authorlist
( title TEXT
, author TEXT
);

CREATE TABLE IF NOT EXISTS edition
( title TEXT
, edition INTEGER
, published DATE
);

CREATE TABLE IF NOT EXISTS author
( name TEXT
, nationality TEXT
);
```

Data in SQL tables has a type which says what sort of data it is

INT / INTEGER whole numbers

TEXT strings

BLOB binary blobs of data (could be anything)

REAL floating point numbers

DECIMAL(5,2) a decimal number of 5 digits (2 of which are after the decimal point)

CHARACTER(10) / VARCHAR(10) A string of 10 characters or upto 10 characters.

DATE / DATETIME a timestamp

BOOLEAN true or false

Others exist. Some are more efficient than others on different databases.

So now what?

Lets start adding the data!

```
INSERT INTO author(name, nationality)
VALUES ('Michael_W._Lucas', 'American')
      , ('Brian_W._Kernighan', 'Canadian')
      , ('P.J._Plaugher', 'American')
;
```

And can we read it back out?

```
SELECT *
FROM author
;
```

name	nationality
Michael W. Lucas	American
Brian W. Kernighan	Canadian
P.J. Plaugher	American

What about if we don't know an author's nationality?

SQL has a special value called NULL to indicate you don't know something.

```
INSERT INTO author(name)
VALUES ('Elonka_Dunin');
```

```
SELECT * FROM author;
```

name	nationality
Michael W. Lucas	American
Brian W. Kernighan	Canadian
P.J. Plaugher	American
Elonka Dunin	

NULL causes problems

Whilst you can leave blanks in your database with NULL

- ▶ Not generally a good idea
- ▶ NULL wreaks havoc when you come to JOIN tables to each other
- ▶ And when you do statistics...

Best to add a constraint to say that a field can never be NULL.

```
CREATE TABLE IF NOT EXISTS author  
( name TEXT NOT NULL,  
  , nationality TEXT NOT NULL,  
);
```

What other constraints do we have

I've said NOT NULL exists, what others are there?

NOT NULL this can never be blank

UNIQUE no other row can have the same value here

CHECK () run a check that a value meets a condition

PRIMARY KEY implies NOT NULL and UNIQUE (and that it's a primary key!)

Keys...

We can declare a single field PRIMARY KEY, but what if we want a composite key?

- ▶ What if I want a FOREIGN KEY
 - ▶ (Check that this column lines up with something in another table)

```
CREATE TABLE IF NOT EXISTS authorlist
( title TEXT
, author TEXT
, PRIMARY KEY (title, author)
, FOREIGN KEY(author) REFERENCES author(name)
);
```

You don't have to declare primary key, or foreign key relationships...

- ▶ But it might make your database faster (indexes)
- ▶ You can do advanced tricks for how data should change as other tables change
- ▶ Helps to keep you sane!

Lets move on from books

I'm bored of books, and don't want to populate an entire database for the sake of a lecture...
Lets ~~steal~~ borrow a music library database and have an explore!

```
.tables
```

Album	Employee	InvoiceLine	PlaylistTrack
Artist	Genre	MediaType	Track
Customer	Invoice	Playlist	

```
SELECT *  
FROM Artist  
;
```

ArtistId	Name
1	AC/DC
2	Accept
3	Aerosmith
4	Alanis Morissette
5	Alice In Chains
6	Antônio Carlos Jobim
7	Apocalyptica
8	Audioslave
9	BackBeat

Lets try that again

```
SELECT Name  
FROM Artist  
LIMIT 5  
;
```

Name
AC/DC
Accept
Aerosmith
Alanis Morissette
Alice In Chains

```
SELECT COUNT(Name) AS Number  
FROM Artist  
;
```

Number
275

I wonder who else's music is in here?

```
SELECT Name
FROM Artist
WHERE Name IS "Chappell_Roan"
;
```

```
SELECT Name
FROM Artist
WHERE Name LIKE "Avril_%"
;
```

```
SELECT Name
FROM Artist
WHERE Name LIKE "%_zep%l%in%"
;
```

What about Albums?

```
SELECT *  
FROM Album  
LIMIT 10  
;
```

AlbumId	Title	ArtistId
1	For Those About To Rock We Salute You	1
2	Balls to the Wall	2
3	Restless and Wild	2
4	Let There Be Rock	1
5	Big Ones	3
6	Jagged Little Pill	4
7	Facelift	5
8	Warner 25 Anos	6
9	Plays Metallica By Four Cellos	7
10	Audioslave	8

What about Led Zeppelin albums?

```
SELECT Album.Title, Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Artist.Name LIKE "Led_Zeppelin"
;
```

Title	Name
BBC Sessions [Disc 1] [Live]	Led Zeppelin
Physical Graffiti [Disc 1]	Led Zeppelin
BBC Sessions [Disc 2] [Live]	Led Zeppelin
Coda	Led Zeppelin
Houses Of The Holy	Led Zeppelin
In Through The Out Door	Led Zeppelin
IV	Led Zeppelin
Led Zeppelin I	Led Zeppelin
Led Zeppelin II	Led Zeppelin
Led Zeppelin III	Led Zeppelin
Physical Graffiti [Disc 2]	Led Zeppelin
Presence	Led Zeppelin
The Song Remains The Same (Disc 1)	Led Zeppelin
The Song Remains The Same (Disc 2)	Led Zeppelin

Okay, so how many Led Zeppelin albums?

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Artist.Name LIKE "Led%"
GROUP BY Artist.Name
;
```

Whose got the most albums?

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
GROUP BY Artist.Name
ORDER BY Albums DESC
LIMIT 10
;
```

Albums	Name
21	Iron Maiden
14	Led Zeppelin
11	Deep Purple
10	U2
10	Metallica
6	Ozzy Osbourne
5	Pearl Jam
4	Various Artists
4	Van Halen
4	Lost

Lets just list the artists with more than 5 albums

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Albums >= 5
GROUP BY Artist.Name
ORDER BY Albums DESC
LIMIT 10
;
```

Parse error near line 2: misuse of aggregate: COUNT()

Aggregates are tricky

When you use an aggregate function like `COUNT()` you can't filter on it with `WHERE`.

- Instead you need to use `HAVING`...

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
GROUP BY Artist.Name
HAVING Albums >= 5
ORDER BY Albums DESC
;
```

Albums	Name
21	Iron Maiden
14	Led Zeppelin
11	Deep Purple
10	U2
10	Metallica
6	Ozzy Osbourne
5	Pearl Jam

Aggregates to be aware of:

`COUNT()` counts number of rows

`SUM()` adds values in rows

`MAX()` gives biggest value

`MIN()` gives minimum value

`AVG()` gives the average of the rows

Whose got the least?

```
SELECT COUNT(Album.Title) AS Albums
      , Artist.Name
FROM Album
JOIN Artist
ON Album.ArtistId = Artist.ArtistId
GROUP BY Artist.Name
ORDER BY Albums ASC
LIMIT 5
;
```

Albums	Name
1	Aaron Copland & London Symphony Orchestra
1	Aaron Goldberg
1	Academy of St. Martin in the Fields & Sir Neville Marriner
1	Academy of St. Martin in the Fields Chamber Ensemble & Sir Neville Marriner
1	Academy of St. Martin in the Fields, John Birch, Sir Neville Marriner & Sylvia McNair

BUT WHAT ABOUT AVRIL LAVIGNE?

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name  
FROM Album  
JOIN Artist  
ON Album.ArtistId = Artist.ArtistId  
WHERE Artist.Name LIKE "Avril%"  
;
```

When we do a JOIN in SQL its technically an INNER JOIN.

- ▶ That means we need something on both sides
- ▶ If there are no albums to join onto artists...
 - ▶ Then they are not included in the results.

If we want to include them we need an OUTER JOIN

OUTER JOIN

Three variants:

- a **LEFT OUTER JOIN** b if there's something in a but nothing in b to join it to... then leave a NULL for the missing values.
- a **RIGHT OUTER JOIN** b if there's something in b but nothing in a to join it to... then leave a NULL for the missing values.
- a **FULL OUTER JOIN** b include all the entries in a and b adding NULL s as needed

```
SELECT "Albums_with_no_artist" AS Description
      , COUNT(Album.Title) AS Count
FROM Album
LEFT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Artist.Name IS NULL
;
```

```
SELECT "Artists_with_no_album" AS Description
      , COUNT(Artist.Name) AS Count
FROM Album
RIGHT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Album.Title IS NULL
;
```

Seems a little excessive...

```
SELECT "Artists_with_no_album" AS Description
      , COUNT(Artist.Name) AS Count
FROM Album
RIGHT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Album.Title IS NULL
UNION
SELECT "Albums_with_no_artist" AS Description
      , COUNT(Album.Title) AS Count
FROM Album
LEFT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
WHERE Artist.Name IS NULL
;
```

(Use UNION ALL if you want to insist that the tables you're joining have the same fields).

Why IS?

NULL means I don't know which means that the logic gets a bit funky.

- ▶ How would you know if an unknown equaled a value?
- ▶ How would you know if a value was unknown?
- ▶ What is an unknown equal to?

Lets not get bogged down in philosophy!

- ▶ If you want to test whether a value IS NULL use IS not =
- ▶ Otherwise your joins will sometimes be weird

Sub Queries

Lets try and find the average number of albums per artist in the database!

- ▶ To do this we'll need a sub-query

A sub-query is when we use the results of one SQL query as part of the input for a second. We can get the number of albums each artist has had with:

```
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
RIGHT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
GROUP BY Artist.Name
LIMIT 5;
```

Albums	Name
0	A Cor Do Som
2	AC/DC
1	Aaron Copland & London Symphony Orchestra
1	Aaron Goldberg
1	Academy of St. Martin in the Fields & Sir Neville Marriner

The naming of tables is a difficult matter...

We could turn our query into a separate table...

- ▶ CREATE TEMPORARY TABLE ensures our table isn't saved to the database
- ▶ Will persist through the current session though
- ▶ Will not update if Artist or Album changes though

```
CREATE TEMPORARY TABLE AlbumsPerArtist AS
SELECT COUNT(Album.Title) AS Albums, Artist.Name
FROM Album
RIGHT OUTER JOIN Artist
ON Album.ArtistId = Artist.ArtistId
GROUP BY Artist.Name
;

SELECT SUM(Albums) as Albums
      , COUNT(Name) AS Artists
      , AVG(Albums) as "Albums_per_Artist"
FROM AlbumsPerArtist;
```

Albums	Artists	Albums per Artist
347	275	1.26181818181818

Subqueries avoid naming things

But if we know we're never going to use it again we can create a subquery by wrapping our first query in ().

- ▶ Useful if you're rubbish at thinking of names
- ▶ Subquery rerun every time query run

```
SELECT SUM(Albums) as Albums
      , COUNT(Name) AS Artists
      , AVG(Albums) as "Albums_per_Artist"
FROM (
  SELECT COUNT(Album.Title) AS Albums, Artist.Name
  FROM Album
  RIGHT OUTER JOIN Artist
  ON Album.ArtistId = Artist.ArtistId
  GROUP BY Artist.Name
);
```

Albums	Artists	Albums per Artist
347	275	1.26181818181818

Thats the basics of SQL

We've covered:

- ▶ CREATE TABLE and DROP TABLE
- ▶ INSERT-ing data
- ▶ Data types
- ▶ SELECT-ing data WHERE there is are conditions
- ▶ Various JOIN-s
- ▶ NULL
- ▶ Aggregate queries
- ▶ Subqueries

That's 99% of all you'll ever need

- ▶ Lets play with different queries in the lab

Now...

- ▶ What other things can we ask this database?