Getting to know SQL and mmsql – a tutorial for beginners

This tutorial is for everyone who would like to try databases on the Colour Maximite 2, even without ever having written a line of SQL. In about half an hour you will look at an existing database, ask it questions, build a small database of your own, change data, connect two databases, and export and import data.

What you need: a Colour Maximite 2 with MMBasic **8.00.00b1 or later**, the `sql` folder on the SD card (it contains `computers.db`), and mmsql 1.2 with its files. Copy `demo.sql` into the same folder as mmsql, e.g. `A:/sql/`.

Important: use the unchanged `computers.db` from the firmware zip, so that your results match this tutorial. The tutorial never changes `computers.db`, but from section 4 on it changes the new database `club.db` (for example, a row is deleted), so the steps work exactly as written only on the first run. To start again, quit mmsql and delete `club.db` at the MMBasic prompt:

```
KILL "club.db"
```

Or, as a first small exercise, without leaving mmsql: close the club database with `.close` (if it is open), list the database files with `.files`, and delete `club.db` with `.kill`. mmsql asks before it deletes anything:

```
mmsql> .files
Folder: A:/sql
name          size      modified
------------------------------------------
club.db       16 KB     2026-11-20 15:20
computers.db  12 KB     2026-11-16 18:02
 2 file(s)
mmsql> .kill club.db
Delete club.db (16 KB)? (y/n) y
Deleted: club.db
```

Lines to type are shown like this:

```
.tables
```

At important steps a dark box shows what the screen should look like, so you can check that you are on the right track. Behind the prompt `[...]>` is what you type, below it is the answer from mmsql. Where an input runs over several lines, it is shown once more above the box, ready to copy and paste into the terminal:

```
[computers.db]> .tables
name     type  ncol
------------------
computer table 6
 1 row(s)
```

1. Two minutes of theory

A **database** is a file that holds **tables**. A table looks like a spreadsheet:

id	name	maker	year	cpu	ramkb
1	Apple II	Apple	1977	6502	4
9	Commodore 64	Commodore	1982	6510	64

- Each **row** is one record (here: one computer).
- Each **column** has a name and a type: `INTEGER` (whole number), `REAL` (decimal number), `TEXT` or `BLOB` (binary data).
- The **primary key** (here `id`) is a number that identifies each row.
- An empty field is called **NULL** (“no value”). It is not the same as 0 or an empty text.

SQL is the language for talking to the database. It reads almost like English:

```
SELECT name, year FROM computer WHERE year < 1980;
```

means “show me name and year from the table computer, but only where the year is before 1980”.

mmsql is the tool on the CMM2 where you type these questions – a Swiss army knife for SQLite databases. It has its own commands, which always start with a **dot** (`.tables`, `.help`). Anything without a dot goes to the database as SQL.

2. Start mmsql and look around

```
RUN "mmsql", "A:/sql/computers.db"
```

mmsql starts and opens the sample database that comes with the firmware:

```
mmsql> V1.2 - SQLite manager for the Colour Maximite 2 (G2)
       (c) 2026 Matthias Hollatz - MIT License
       Type .help for help
Opened: A:/sql/computers.db
[computers.db]>
```

Already in mmsql – started without a database, or right after `.kill club.db`? Then simply open it from the prompt:

```
.open A:/sql/computers.db
```

```
mmsql> .open A:/sql/computers.db
Opened: A:/sql/computers.db
[computers.db]>
```

The prompt `[computers.db]>` always shows which database is open. Now look around:

```
.tables
.columns computer
.browse computer
```

- `.tables` lists the tables: there is one, `computer`.
- `.columns computer` shows its columns and their types.
- `.browse computer` shows the rows page by page: **Enter** for the next page, **B** to go back, **Q** to stop.

Useful at any time: `.help` lists all commands, `.help browse` explains one. The **Up** arrow brings back what you typed before – even after a restart.

3. Asking questions with SELECT

Type these lines one by one and press Enter after each. The first one should give:

```
[computers.db]> SELECT name, year FROM computer WHERE year < 1980;
name           year
-----------------
Apple II       1977
PET 2001       1977
TRS-80 Model I 1977
Atari 800      1979
 4 row(s)
```

Now the others:

```
SELECT name, ramkb FROM computer ORDER BY ramkb DESC;
SELECT name, maker FROM computer WHERE cpu = 'Z80';
SELECT name, year FROM computer WHERE name LIKE '%zx%';
```

What happened:

- `SELECT name, year` chooses the **columns**; `SELECT *` would show all of them.
- `WHERE` chooses the **rows**. Texts are written in single quotes: `'Z80'`.
- `ORDER BY ramkb DESC` sorts, largest first (`ASC` or nothing: smallest first).
- `LIKE '%zx%'` searches for part of a text; `%` stands for “anything”. Upper and lower case do not matter here.

Longer questions over several lines: F3 and F2

A question can be spread over several lines, which makes longer ones much easier to read. Type the first line, then press **F3** instead of Enter. The prompt line turns into a status line, and the lines appear numbered below it (or press F3 and paste the block):

```
SELECT name, cpu, ramkb
  FROM computer
  WHERE ramkb >= 64;
```

```
[computers.db]> [ Ln  3/ 3  Col  21/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, cpu, ramkb
 2:   FROM computer
 3:   WHERE ramkb >= 64;
```

After each line press **F3** (or Enter – once there are several lines, Enter also starts a new line). When the question is complete, press **F2** to run it:

```
[computers.db]> [ Ln  3/ 3  Col  21/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, cpu, ramkb
 2:   FROM computer
 3:   WHERE ramkb >= 64;
name          cpu   ramkb
-----------------------
Commodore 64 6510  64
CPC 464      Z80   64
QL           68008 128
Macintosh    68000 128
520ST        68000 512
Amiga 500    68000 512
 6 row(s)
```

Esc throws the whole input away. You can also write SQL in an editor on your PC and paste it into the terminal: mmsql notices the pasted lines and keeps them together; F2 runs them. In a pasted block of several statements, end each statement with `;`.

Counting and calculating

The database can also **count and calculate**:

```
SELECT count(*) FROM computer;
SELECT min(year), max(year), avg(ramkb) FROM computer;
```

And it can count **per group**. `GROUP BY maker` puts all rows with the same maker together, and `count(*)` counts each group. `AS models` gives the result column a name. Commodore wins with 4 models:

```
SELECT maker, count(*) AS models
  FROM computer
  GROUP BY maker
  ORDER BY models DESC, maker;
```

```
[computers.db]> [ Ln  4/ 4  Col  31/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT maker, count(*) AS models
 2:   FROM computer
 3:   GROUP BY maker
 4:   ORDER BY models DESC, maker;
maker       models
------------------
Commodore   4
Sinclair    3
Acorn       2
Apple       2
Atari       2
Amstrad     1
Dragon Data 1
IBM         1
Oric        1
Tandy       1
 10 row(s)
```

A row with many columns is easier to read one field per line:

```
[computers.db]> .mode line
Mode: line
[computers.db]> SELECT * FROM computer WHERE id = 9;
id    = 9
name  = Commodore 64
maker = Commodore
year  = 1982
cpu   = 6510
ramkb = 64
 1 row(s)
[computers.db]> .mode table
Mode: table
```

Try it yourself: Which computers came out in 1984? Which ones have a Z80 and at least 48 KB? (`WHERE cpu = 'Z80' AND ramkb >= 48`)

4. Your own database: the retro club

`demo.sql` describes a small club. Its members own computers from `computers.db`. Create a new database and let mmsql run the file:

```
[computers.db]> .open club.db
File not found: club.db
Create a new database? (y/n) y
Closed: A:/sql/computers.db
Created: club.db
[club.db]> .read demo.sql
Building the retro club ...
Done: tables member and owns, view members_per_city.
 8 statement(s) OK, 0 error(s), 49 line(s) read
```

Answer **y** when mmsql asks whether to create `club.db`. `.read` runs the SQL in the file and reports `8 statement(s) OK`. Open `demo.sql` in an editor to see what it does:

- two `CREATE TABLE` statements build the tables,
- the `INSERT` statements fill them,
- `BEGIN` and `COMMIT` make all inserts one fast **transaction**,
- every statement ends with `;` – that is how mmsql knows where a statement stops, as it may run over several lines,
- `.print` lines show a message.

Look at the result:

```
.tables
.schema member
.browse member
.browse owns
```

- `member` holds 7 members; `owns` holds who owns which computer: `computer_id` is the `id` from `computers.db`.
- `.schema member` shows how the table was built – line by line, just as it is in `demo.sql`.

`members_per_city` is a **view**, a saved question that you can use like a table. Grace has no city: that field is NULL, and mmsql shows it as an empty field.

```
[club.db]> SELECT * FROM members_per_city;
city    members
--------------
        1
Berlin  2
London  2
Sydney  1
Toronto 1
 5 row(s)
```

5. Changing data

Insert a new member, Henry from Dublin, who joined in 2025. mmsql asks for each column; just press Enter for `id`, and the database picks the next number:

```
[club.db]> .insert member
New row in member. Enter = default (or automatic key), NULL or \N = NULL, '' = empty text
  id (INTEGER, PK)
  >
  name (TEXT, NOT NULL)
  > Henry
  city (TEXT)
  > Dublin
  joined (INTEGER)
  > 2025
Insert? (y/n) y
Inserted, rowid 8
```

Edit a row: `.edit member 7` shows each field of member 7 (Grace) with its value. Press Enter to keep a value, or change it – here Grace moves to Paris:

```
[club.db]> .edit member 7
Edit member id = 7. Enter = keep, NULL or \N = NULL, '' = empty text
  id (INTEGER, PK)
  > 7
  name (TEXT, NOT NULL)
  > Grace
  city (TEXT)
  > Paris
  joined (INTEGER)
  > 2024
Save 1 change(s)? (y/n) y
Saved.
```

The same change in SQL would be:

```
UPDATE member SET city = 'Paris' WHERE name = 'Grace';
```

Delete a row: Dave's TRS-80 is broken and goes to the bin. The table `owns` has no primary key, so mmsql uses the hidden row number `rowid`, which every SQLite table has. `.browse` does not show it, so ask for it in SQL:

```
[club.db]> SELECT rowid, * FROM owns WHERE condition = 'broken';
rowid member_id computer_id condition
-----------------------------------
9     4         3           broken
 1 row(s)
[club.db]> .delete owns 9
member_id   = 4
computer_id = 3
condition   = broken
Delete this row? (y/n) y
Deleted: 1 row(s)
```

The `SELECT` shows that the broken TRS-80 has `rowid` 9. mmsql shows the row again and asks before it deletes. In SQL: `DELETE FROM owns WHERE rowid = 9;`.

Be careful: `UPDATE` and `DELETE` without `WHERE` change **every** row of the table! A good habit: first run a `SELECT` with the same `WHERE`, and only then the `UPDATE` or `DELETE`. Or work inside a transaction: `BEGIN`, make the change, look at the result, then `COMMIT` to keep it or `ROLLBACK` to undo it. While a transaction is open, the prompt shows a star: `[club.db]*>`.

6. Connecting two databases

The club only knows computer numbers; the names are in `computers.db`. **ATTACH** opens it next to the club database under a short name:

```
[club.db]> ATTACH 'A:/sql/computers.db' AS retro;
OK
[club.db]> .databases
name  file
------------------------
main  A:/sql/club.db
retro A:/sql/computers.db
 2 row(s)
```

Now `retro.computer` is the computer table. A **JOIN** puts rows from several tables together where the numbers match. This is a longer question, so use F3 after each line and F2 at the end:

```
SELECT m.name AS member, k.name AS computer, k.year
  FROM owns o
  JOIN member m ON m.id = o.member_id
  JOIN retro.computer k ON k.id = o.computer_id
  ORDER BY m.name, k.year;
```

```
[club.db]> [ Ln  5/ 5  Col  27/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT m.name AS member, k.name AS computer, k.year
 2:   FROM owns o
 3:   JOIN member m ON m.id = o.member_id
 4:   JOIN retro.computer k ON k.id = o.computer_id
 5:   ORDER BY m.name, k.year;
member computer          year
---------------------------
Ada    Commodore 64      1982
Ada    ZX Spectrum       1982
Bob    Apple II          1977
Bob    BBC Micro Model B 1981
Bob    Amiga 500         1987
Clara  Commodore 64      1982
Clara  CPC 464           1984
Dave   ZX81              1981
Eve    Macintosh         1984
Eve    520ST             1985
Frank  QL                1984
 11 row(s)
```

Read it like this: take each row of `owns` (short name `o`), find the member whose `id` is `o.member_id` and the computer whose `id` is `o.computer_id`, and show member, computer and year. Dave's TRS-80 is missing: it was deleted in section 5.

Who owns no computer at all? A **LEFT JOIN** keeps the members without a match; their `owns` fields are NULL:

```
SELECT m.name
  FROM member m
  LEFT JOIN owns o ON o.member_id = m.id
  WHERE o.member_id IS NULL;
```

```
[club.db]> [ Ln  4/ 4  Col  29/255 ]__[ F2 = run  Esc = discard ]_____________________________________
 1: SELECT m.name
 2:   FROM member m
 3:   LEFT JOIN owns o ON o.member_id = m.id
 4:   WHERE o.member_id IS NULL;
name
-----
Grace
Henry
 2 row(s)
```

More ideas to try:

```
SELECT k.maker, count(*) AS owned FROM owns o JOIN retro.computer k ON k.id = o.computer_id
GROUP BY k.maker ORDER BY owned DESC;
SELECT m.name, k.name FROM owns o JOIN member m ON m.id = o.member_id JOIN retro.computer k ON
k.id = o.computer_id WHERE k.cpu = 'Z80';
```

To keep a copy of the computer list in the club database, and to close the connection:

```
CREATE TABLE computer AS SELECT * FROM retro.computer;
DETACH retro;
```

7. Export and import

CSV files open in every spreadsheet program on the PC:

```
.mode csv
.headers on
.once members.csv
SELECT * FROM member;
.mode table
```

`.once` sends only the next result into the file. Read the file back into a new table:

```
[club.db]> .import members.csv member_copy
Created table member_copy with 4 column(s)
 8 row(s) imported into member_copy
[club.db]> .browse member_copy
id name  city    joined
----------------------
1  Ada   London  2019
2  Bob   Sydney  2020
3  Clara Berlin  2020
4  Dave  London  2021
5  Eve   Toronto 2022
6  Frank Berlin  2023
7  Grace Paris   2024
8  Henry Dublin  2025
 8 row(s)
```

The first line of the file supplies the column names.

Markdown is handy for posting a table in a forum or on GitHub:

```
[club.db]> .mode markdown
Mode: markdown
[club.db]> SELECT * FROM members_per_city;
| city    | members |
|---------|---------|
| Berlin  | 2       |
| Dublin  | 1       |
| London  | 2       |
| Paris   | 1       |
| Sydney  | 1       |
| Toronto | 1       |
 6 row(s)
[club.db]> .mode table
Mode: table
```

A **dump** is the whole database as SQL text – a good backup that also works with SQLite on a PC:

```
.once club_backup.sql
.dump
```

`.read club_backup.sql` in an empty database rebuilds everything. Even simpler, a ready-to-use copy of the database file:

```
.backup club_copy.db
```

`.restore club_copy.db` puts such a copy back later.

8. Keeping the database healthy

```
.info
.check
.vacuum
```

```
[club.db]> .check
Running quick_check ...
OK - no problems found.
```

`.info` shows size, pages and the number of tables. `.check` tests the file. `.vacuum` makes it compact again after many deletes. Finally:

```
.quit
```

A last tip: with no database open at all, mmsql still answers SQL – it uses a database in memory. Handy as a calculator:

```
mmsql> SELECT 7 * 6 AS answer, date('2026-11-20', '+30 days') AS later;
(No database open - using a database in memory. .open file.db for a file.)
answer later
------------
42     2026-12-20
 1 row(s)
```

SQL cheat sheet

What	Example
Show rows	`SELECT name, year FROM computer WHERE year < 1980 ORDER BY year;`
Count, sum, average	`SELECT count(*), sum(ramkb), avg(ramkb) FROM computer;`
Groups	`SELECT maker, count(*) FROM computer GROUP BY maker;`
Search in text	`WHERE name LIKE '%spectrum%'`
Several conditions	`WHERE year >= 1982 AND cpu = 'Z80'`
Empty fields	`WHERE city IS NULL` (never `= NULL`)
New table	`CREATE TABLE note(id INTEGER PRIMARY KEY, text TEXT);`
New row	`INSERT INTO note(text) VALUES('Hello');`
Change rows	`UPDATE note SET text = 'Hi' WHERE id = 1;`
Delete rows	`DELETE FROM note WHERE id = 1;`
Combine tables	`SELECT ... FROM a JOIN b ON b.id = a.b_id;`
Remove a table	`DROP TABLE note;`

Key	What it does
Enter	run one line
F3	start a new line (several lines)
F2	run several lines
Esc	throw the input away
Up / Down, PgUp / PgDn	earlier input (history)

Where next? `.help` and `.help sql` in mmsql, the mmsql manual (every command, the full SQL course and the limits), the SQL chapter of the Colour Maximite 2 User Manual (how to use SQL from your own BASIC programs), and https://www.sqlite.org/lang.html for the complete SQL language.

Have fun with your data!