Getting to know SQL and mmsql – a tutorial for beginners

This tutorial is for everyone who wants to try databases on the Colour Maximite 2, even if you have never written a line of SQL. In about half an hour you will look at an existing database, ask it questions, build your own small database, change data, connect two databases, and export and import data.

You need: a Colour Maximite 2 with MMBasic 8.00.00b1 or later and mmsql 1.1 (with 7.00.00RC6 use mmsql 1.0.1), the `sql` folder on the SD card (it contains `computers.db`), and mmsql 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, `.delete owns 9` removes one row), 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"
```

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 `mmsql[...]>` is what you type, below it is the answer of mmsql:

```
mmsql[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”), which is not the same as 0 or an empty text.

SQL is the language you use to talk 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. It has its own commands, which always start with a **dot** (`.tables`, `.help`). Anything without a dot is sent to the database as SQL.

2. Start mmsql and look around

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

mmsql opens the sample database that comes with the firmware. 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 the commands you typed before.

3. Asking questions with SELECT

Type these lines one by one and look at the result. The first one should give:

```
mmsql[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, year FROM computer WHERE year < 1980
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 a part of a text; `%` stands for “anything”. Upper and lower case do not matter here.

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

```
SELECT count(*) FROM computer
SELECT min(year), max(year), avg(ramkb) FROM computer
SELECT maker, count(*) AS models FROM computer GROUP BY maker ORDER BY models DESC
```

`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:

```
maker       models
------------------
Commodore   4
Sinclair    3
Atari       2
Apple       2
Acorn       2
Tandy       1
Oric        1
IBM         1
Dragon Data 1
Amstrad     1
 10 row(s)
```

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

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

Try it yourself: Which computers have 64 KB or more? (`WHERE ramkb >= 64`)

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:

```
mmsql[computers.db]> .open club.db
File not found: club.db
Create a new database? (y/n) y
Created: club.db
mmsql[club.db]> .read demo.sql
 25 statement(s) OK, 0 error(s), 38 line(s) read
```

Answer **y** when mmsql asks whether to create `club.db`. `.read` runs every line of the file as SQL and reports `25 statement(s) OK`. Open `demo.sql` in an editor to see what it does: two `CREATE TABLE` statements build the tables, the `INSERT` lines fill them, and `BEGIN` and `COMMIT` make all inserts one fast **transaction**.

Look at the result:

```
.tables
.schema member
.browse member
.browse owns
SELECT * FROM members_per_city
```

- `member` holds 7 members, `owns` holds who owns which computer: `computer_id` is the `id` from `computers.db`.
- `members_per_city` is a **view**, a saved question that you can use like a table. Grace has no city: that field is NULL.

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:

```
mmsql[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 in the line editor. Press Enter to keep a value, or change it, e.g. type `Paris` as her city:

```
.edit member 7
```

The same change in SQL:

```
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:

```
mmsql[club.db]> SELECT rowid, * FROM owns WHERE condition = 'broken'
rowid member_id computer_id condition
-------------------------------------
9     4         3           broken
 1 row(s)
mmsql[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`.

6. Connecting two databases

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

```
ATTACH 'A:/sql/computers.db' AS retro
.databases
```

Now `retro.computer` is the computer table. A **JOIN** puts rows from several tables together where the numbers match:

```
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
```

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:

```
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    520ST             1985
Eve    Macintosh         1984
Frank  QL                1984
 11 row(s)
```

Dave's TRS-80 is missing: you deleted it in section 5.

More questions across both databases:

```
SELECT m.name AS member, k.name AS computer 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'
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 FROM member m LEFT JOIN owns o ON o.member_id = m.id WHERE o.member_id IS NULL
```

The first finds the Z80 owners, the second counts the computers per maker in the club, and the last finds members who own no computer at all: Grace and Henry (`LEFT JOIN` keeps members without a match; their `owns` fields are NULL).

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
```

Every SQL statement must fit into 255 characters, the length of an MMBasic string. The JOIN examples above are about 160.

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:

```
mmsql[club.db]> .import members.csv member_copy
Created table member_copy with 4 column(s)
 8 row(s) imported into member_copy
mmsql[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.

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
```

8. Keeping the database healthy

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

```
mmsql[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
```

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`

Where next? `.help` and `.help sql` in mmsql, the `README` of mmsql, 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 full SQL language.

Have fun with your data!