mmsql 1.2 – Manual
SQLite databases on the Colour Maximite 2

```
mmsql> V1.2 - SQLite manager for the Colour Maximite 2 (G2)
       (c) 2026 Matthias Hollatz - MIT License
       Type .help for help
mmsql>
```

For my son Konrad.

© 2026 Matthias Hollatz – MIT License (see LICENSE.txt)

Tested on a Colour Maximite 2 (Generation 2) with MMBasic **8.00.00b1** and **8.00.00b2**.

What is mmsql?

mmsql is the Swiss army knife for SQLite databases on the Colour Maximite 2. One program to create databases, look into them, query, change, check, back up, import and export them – right on the CMM2, without writing a line of BASIC. It works like `sqlite3`, the tool the SQLite developers provide for the PC, and adds a lot that makes life easier on a small computer without a mouse: an editor for long statements, paging through large tables, forms for editing rows, help in English and German. mmsql itself is written in MMBasic and is free (MIT License).

> **And what is SQLite?** Its developers describe it on sqlite.org as a library that contains a complete SQL database – self-contained, serverless, needing no setup, and with transactions. They call SQLite the most used database engine in the world; it is in every smartphone and every web browser. Their motto: *“Small. Fast. Reliable. Choose any three.”* Since firmware 7, SQLite is in the CMM2 too – and mmsql is the key to it.

Foreword

This manual is for everyone who wants to work with data on the Colour Maximite 2: sorting a games collection, keeping measurements, managing the members of a club, or simply understanding what lies behind the word “database”. **No knowledge of SQL is needed.** Everything is explained step by step, with one example to follow along.

Readers who have worked with databases before will find the differences to MySQL, MariaDB, MS SQL Server & Co. in chapter 12, all technical limits in chapter 13 and an honest comparison with the model, `sqlite3`, in chapter 14. MMBasic programmers will find tips from the mmsql source code in chapter 15 – to read and to reuse in their own programs.

How this manual works

It is best read at the CMM2, typing along. The examples build on each other: chapter 3 creates the database `retro.db`, which all later chapters use.

What to type is shown like this:

```
SELECT name FROM computer;
```

What the screen looks like afterwards is shown in a dark box. The first line starts with the **prompt** – the input line, here `[retro.db]>`. After it comes what was typed, below it the answer from mmsql:

```
[retro.db]> SELECT name FROM computer WHERE year = 1987;
name
---------
Amiga 500
 1 row(s)
```

So it is always easy to check that everything is on the right track.

Where an input runs over several lines, it is shown once more above the box as a plain block – ready to copy and paste into the terminal.

Boxes headed **New in 1.2** mark what was not yet in mmsql 1.1.

Contents

- What is mmsql?
- New in 1.2 – the highlights

1. What is a database?
2. Installing and starting
3. A ten-minute tour
4. Input: editor, keys and blocks
5. SQL step by step
6. Looking around without SQL
7. Editing without SQL
8. Output and export
9. Data in and out: import, dump, scripts
10. Safety and maintenance
11. Settings
12. For users of MySQL, MariaDB, MS SQL Server & Co.
13. Technical data and limits
14. mmsql and sqlite3 – an honest comparison
15. Under the bonnet – mmsql as a source of ideas
16. Messages and troubleshooting

- Appendix A: All commands at a glance
- Appendix B: Keys
- Appendix C: Options
- Appendix D: SQL cheat sheet
- Appendix E: Changes since 1.1
- Appendix F: Glossary

New in 1.2 – the highlights

A real editor at the prompt. Until now every statement had to fit into one line of at most 255 characters. Now SQL can run over up to 20 lines, as clearly laid out as wanted: F3 starts a new line, F2 runs the statement. A status line shows where the cursor is. SQL written on the PC can simply be pasted into the terminal – mmsql notices and keeps the lines together. → Chapter 4

The limit of 20 columns is gone. `.import` reads CSV files with any number of columns, `.insert` fills tables with up to 64 columns. Fields may be longer than 255 characters. → Chapters 7 and 9

Long queries show their result. A SELECT of more than 255 characters works like a short one: as a table to page through, or in any output mode – at the prompt, in a block and in scripts. → Chapter 5

Scripts as in sqlite3. `.read` understands statements over several lines, shows the results of queries and knows dot commands such as `.print`, `.mode` and `.once`. So dumps and scripts from the PC run too. → Chapter 9

One rule for input. One line runs with Enter. Several lines are a block: each statement ends with `;`, dot commands run along, the first error stops the rest. → Chapter 4

The history remembers everything. Long statements over many lines come back with Up or PgUp – even after a restart. → Chapter 4

Start straight away without a file. With no database open, mmsql works in a database in memory (`:memory:`). → Chapter 9

New output: `.mode markdown` for tables in texts and forums, `.mode json` for other programs, `.width` for fixed column widths, `.print` for headings. → Chapter 8

`.dump` tells BLOB and text apart reliably – after reading the dump back, every value has exactly its type again. → Chapter 9

More new commands: `.restore` (put a backup back), `.version`, search patterns for `.tables`, `.schema`, `.indexes` and `.triggers`, `.set screen` (screen mode at the start) and `.set width 0` (width follows the screen). → Chapters 6, 10 and 11

> **Note, a change that affects old scripts:** `.read` now needs a `;` at the end of every statement, as `sqlite3` does. Scripts with one statement per line **without** `;` no longer run correctly. Adding a `;` at the end of each line is enough. Dumps from mmsql have always had it. Details in chapter 9.

1. What is a database?

Tables, rows, columns

A **database** is a file in which data is kept in good order in **tables**. A table looks like a sheet in a spreadsheet program. The firmware package of the CMM2 comes with a small sample database, `computers.db`. It contains the table `computer` with 18 home computers; here is part of it:

id	name	maker	year	cpu	ramkb
1	Apple II	Apple	1977	6502	4
9	Commodore 64	Commodore	1982	6510	64
18	Amiga 500	Commodore	1987	68000	512

- Each **row** is one record – here, one computer.
- Each **column** has a name and a **type**: `name` holds text, `year` a whole number.
- The column `id` is the **primary key**: a number that identifies each row. Two computers may have the same name, but never the same `id`.
- An empty field is called **NULL** – "no value". NULL is not the same as 0 or an empty text: for a computer whose year is unknown, `year` is NULL, not 0.

A database can hold many tables that belong together. In chapter 5 a table `makers` joins `computer`, and the two are connected.

What can a database do that a text file cannot?

Of course the list of computers could also be kept in a text file. But a database can do more:

- **Answer questions:** "Which computers with a Z80 came out before 1983?" – one line of SQL, and the answer is there, sorted as wished.
- **Calculate:** count, add up, average, minimum, maximum, grouped by maker or year.
- **Connect:** bring together data from several tables without storing it twice.
- **Keep watch:** required fields, unique keys, rules that are checked when data is saved.
- **Change safely:** several changes either all take effect or none at all (a *transaction*). If the power fails in the middle, the database stays intact.

SQL – the language of the database

Databases are spoken to in **SQL** (*Structured Query Language*). SQL reads almost like English:

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

means: "Select name and year from the table computer, only those before 1980, sorted by year." Upper and lower case do not matter for SQL words; this manual writes them in capitals so that they stand out. Chapter 5 explains SQL step by step.

SQLite and mmsql

The CMM2 firmware contains **SQLite**, the most used database in the world – it also runs in every smartphone and every web browser. An SQLite database is **one single file**. There is no server, no users, no passwords. The file can be copied, sent to the PC by XMODEM and opened there with `sqlite3` or *DB Browser for SQLite* – and the other way round.

BASIC programs use SQLite with commands such as `SQL OPEN`, `SQL EXEC` and `SQL QUERY`. **mmsql** is the tool for it: a program with which databases can be looked at, queried, changed, checked, backed up, exported and imported right on the CMM2 – without a line of BASIC. It works like the command-line tool `sqlite3` on the PC and is itself written entirely in MMBasic.

There are two kinds of input in mmsql:

- **Commands for mmsql** start with a **dot**: `.tables`, `.help`, `.quit`.
- **Everything else is SQL** and goes to the database: `SELECT * FROM computer;`.

mmsql runs on the screen of the CMM2 and on a serial terminal on the PC (e.g. Tera Term) at the same time. Both show the same, and both can be typed on.

2. Installing and starting

What is needed

Computer	Colour Maximite 2, Generation 1, 2 or 3
Firmware	MMBasic **8.00.00b1** or later
SQLite module	`A:/sql/sql.mod` from the firmware package
Sample database	`A:/sql/computers.db` from the firmware package
for firmware 7.00.00RC6	use mmsql **1.0.1** instead

If mmsql 1.2 is started on older firmware, it says so, points to mmsql 1.0.1 and ends.

The files

File	Purpose
`mmsql.bas`	the program
`mmsql_en.hlp`	help texts, English
`mmsql_de.hlp`	help texts, German
`LICENSE.txt`	MIT License
`demo.sql`	sample data for the tutorial
PDF files	this manual, the German Handbuch, README and tutorial

All files go into the same folder. The easiest is `A:/sql`, where `sql.mod` and `computers.db` already are – then the sample database is right at hand. mmsql creates two more files in its folder:

File	Contents
`mmsql.cfg`	the settings (`.set`), as a readable text file
`mmsql.his`	the history of the input

Both may be deleted at any time; mmsql then starts with the default settings or an empty history.

Copying to the CMM2

Either put the SD card into the PC and copy the files, or send them over the serial cable with XMODEM. In Tera Term: *File* → *Transfer* → *XMODEM* → *Send*, first on the CMM2:

```
CHDIR "A:/sql"
XMODEM RECEIVE "mmsql.bas"
```

Repeat this for each file.

Setting up the terminal

mmsql also works without a PC, with just the screen and a USB keyboard. With a terminal on the PC many things are easier, above all pasting SQL from a text editor. For Tera Term these settings are recommended:

Setting in Tera Term	Value	what for
Setup → Serial port → Speed	as on the CMM2 (usually 115200)	connection
Setup → Additional settings → Copy and Paste → Paste delay per line	e.g. 200 ms	pasting several lines (chapter 4)
Setup → Additional settings → Copy and Paste → Trim trailing new line	on	no empty Enter at the end of pasted blocks

At the start mmsql sets the terminal window to the same size as the screen (e.g. 50 rows × 100 columns). So both show exactly the same.

Starting

At the MMBasic prompt:

```
CHDIR "A:/sql"
RUN "mmsql"
```

mmsql greets like this:

```
mmsql> V1.2 - SQLite manager for the Colour Maximite 2 (G2)
       (c) 2026 Matthias Hollatz - MIT License
       Type .help for help
mmsql>
```

The first line looks as if `.version` had been typed at the prompt: version, generation of the CMM2 (G1, G2 or G3) and licence. Below it the prompt `mmsql>` waits for input. It is called that as long as no database is open.

A database can be given straight away:

```
RUN "mmsql", "computers.db"
```

Help

mmsql has built-in help:

Input	shows
`.help`	the overview of all commands in groups
`.help command`	details of one command, e.g. `.help import`
`.help keys`	the keys of the editor
`.help sql`	notes on SQL in mmsql
`.help memory`	the database in memory
`.help utf8`	umlauts and other special characters

```
[retro.db]> .help
.help name shows details, e.g. .help import  (also: .help keys/sql/utf8)

Database  : .open .close .info .backup .restore .vacuum .files .kill
Look      : .tables .indexes .triggers .schema .columns .browse .databases
Data      : .insert .edit .delete .import .read
Structure : .create .alter .reindex
Check     : .check .fkcheck .fk
Output    : .mode .headers .separator .nullvalue .width .once .output .dump
Settings  : .set .timer .bail .echo .print .cls .version .help .quit
Short forms: .o .t .ix .s .c .b .db .ver .h .q

Commands start with a dot. Anything else is SQL. Enter runs one line;
F3 starts a new line for a statement over several lines, F2 runs it
(see .help keys).
```

The help is also available in German: `.set helpfile mmsql_de.hlp` (and back with `.set helpfile mmsql_en.hlp`). The setting is saved in `mmsql.cfg`.

> **Tip: help in another language.** The help files are plain text files. For help in French, Dutch or any other language, have `mmsql_en.hlp` translated on the PC by an AI or with DeepL (deepl.com) and save the result next to mmsql, e.g. as `mmsql_fr.hlp`. Switch it on with `.set helpfile mmsql_fr.hlp`. Three things to keep in mind when translating:
>
> - **Lines starting with `#` stay unchanged.** They hold the names that `.help` looks for (`#open o`, `#keys`).
> - **Commands, options and SQL stay as they are** (`.open`, `.set paste`, `SELECT`).
> - **Use plain letters only.** The CMM2 font has no accented letters; the German help writes “ae”, “oe”, “ue” and “ss” instead. Lines should be at most 79 characters long, so that they fit on any screen.

Quitting

`.quit` (short `.q`, or `.exit`) closes the database and ends mmsql. After that, the MMBasic prompt is back.

3. A ten-minute tour

This chapter opens the sample database, asks it the first questions and makes a copy of its own to practise on. Start mmsql in the folder `A:/sql` (chapter 2).

Step 1: open the database and look inside

`.open` opens a database, `.tables` shows which tables are in it:

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

The prompt is now `[computers.db]>` – so it is always clear which database is in use. There is one table `computer` (`type` = table) with 6 columns (`ncol`). The last line `1 row(s)` says how many rows the answer had – here one.

Step 2: the first query

Now the first SQL statement. `SELECT * FROM computer;` means: “Show all columns (`*`) of the table computer.”

```
[computers.db]> SELECT * FROM computer;
id name              maker       year cpu   ramkb
-------------------------------------------------
1  Apple II          Apple       1977 6502  4
2  PET 2001          Commodore   1977 6502  4
3  TRS-80 Model I    Tandy       1977 Z80   4
4  Atari 800         Atari       1979 6502  8
5  ZX81              Sinclair    1981 Z80   1
6  BBC Micro Model B Acorn       1981 6502  32
7  IBM PC            IBM         1981 8088  16
8  VIC-20            Commodore   1981 6502  5
9  Commodore 64      Commodore   1982 6510  64
10 ZX Spectrum       Sinclair    1982 Z80   48
11 Dragon 32         Dragon Data 1982 6809  32
12 Oric-1            Oric        1983 6502  48
13 Electron          Acorn       1983 6502  32
14 CPC 464           Amstrad     1984 Z80   64
15 QL                Sinclair    1984 68008 128
16 Macintosh         Apple       1984 68000 128
17 520ST             Atari       1985 68000 512
18 Amiga 500         Commodore   1987 68000 512
 18 row(s)
```

mmsql lines up the columns, puts the column names on top and counts the rows at the end. Were the table longer than one screen page, Enter would page on (chapter 6).

Step 3: asking a question

Only the computers before 1980, and only name and year:

```
[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)
```

This is the heart of SQL: one describes **what** one wants to know – not how the computer should search for it. `WHERE` filters the rows, the list after `SELECT` chooses the columns.

Step 4: a database to practise on

`computers.db` belongs to the firmware package and should stay unchanged. Everything else uses a copy:

```
[computers.db]> .backup retro.db
OK - copy written to retro.db
[computers.db]> .open retro.db
Closed: computers.db
Opened: retro.db
[retro.db]>
```

`.backup` writes a clean copy of the open database to a new file. `.open` closes the old database (“Closed”) and opens the new one. From now on all examples in this manual use **`retro.db`**. To start again some time: `.quit`, at the MMBasic prompt `KILL "retro.db"`, start mmsql again and repeat step 4.

Step 5: forgot the dot? No problem

Commands for mmsql start with a dot. If it is missing, mmsql helps:

```
mmsql> open computers.db
Did you mean .open? Commands start with a dot.
```

The same goes for a comma typed by mistake instead of the dot (`,help`).

Step 6: quitting

```
[retro.db]> .quit
Closed: retro.db
>
```

Next time, type `RUN "mmsql", "retro.db"` or open the file with `.open retro.db`.

That was the tour: a database opened, its tables looked at, two questions asked and a backup copy made. The next chapter shows how to type longer statements comfortably; after that, SQL really gets going.

4. Input: editor, keys and blocks

Everything typed at the prompt goes into a small editor. It works the same on the screen and in the terminal.

One line: type and press Enter

For most input one line is enough. Enter runs it:

```
[retro.db]> SELECT name, year FROM computer WHERE maker = 'Atari';
name      year
--------------
Atari 800 1979
520ST     1985
 2 row(s)
```

A `;` at the end is allowed but not needed. These keys help while typing:

Key	Effect
← →	move the cursor
Home / End	to the start / end of the line
Backspace	delete the character left of the cursor
Del	delete the character at the cursor
Esc	throw the input away
↑ ↓	bring back earlier input from the history
Enter	run

If the line is longer than the screen is wide, it scrolls sideways. A line holds at most **255 characters** – the length of a string in MMBasic.

Several statements in one line are separated by `;`. They run one after the other, each with its own result:

```
[retro.db]> SELECT count(*) FROM makers; SELECT count(*) FROM computer;
Error: SQL: no such table: makers
  Statement ( 27 chars): SELECT count(*) FROM makers
Hint: .tables lists the tables
Not run: 1 more statement(s) of this line.
```

If one of them fails, the rest does not run: “Not run: n more statement(s) of this line.”

Several lines: F3 and F2

> **New in 1.2:** statements over several lines.

Longer queries are easier to read when spread over several lines. Two keys do this:

Key	Effect
F3	start a new line (or split the line at the cursor)
F2	run the whole statement

To try it, type `SELECT name, year, cpu` and press **F3**. The display changes at once: the prompt line becomes a **status line**, and the lines appear numbered below it. Go on typing, F3 after each line:

```
SELECT name, year, cpu
  FROM computer
  WHERE cpu = 'Z80'
  ORDER BY year;
```

```
[retro.db]> [ Ln  4/ 4  Col  17/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, year, cpu
 2:   FROM computer
 3:   WHERE cpu = 'Z80'
 4:   ORDER BY year;
```

The status line shows:

- the prompt `[retro.db]>`, so that it is clear which database is in use,
- `[ Ln  4/ 4  Col  17/255 ]` – the cursor is in line 4 of 4, column 17 of at most 255,
- `[ F2 = run  Esc = discard ]` – as a reminder: F2 runs, Esc throws away.

The underscores fill the line up to the right edge. Now **F2**:

```
[retro.db]> [ Ln  4/ 4  Col  17/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, year, cpu
 2:   FROM computer
 3:   WHERE cpu = 'Z80'
 4:   ORDER BY year;
name           year cpu
----------------------
TRS-80 Model I 1977 Z80
ZX81           1981 Z80
ZX Spectrum    1982 Z80
CPC 464        1984 Z80
 4 row(s)
```

As soon as there are several lines, **Enter** starts a new line – just as in a text editor. Only F2 runs. So nothing can start by accident while still writing.

Tip: on an empty prompt, F3 switches straight to several lines without creating an empty line 1. So when it is going to be long, press F3 first.

Moving and changing in several lines

Key	Effect
← →	move the cursor, also across line ends
↑ ↓	one screen row up / down; in the first or last row: history
Home / End	start / end of the screen row
Backspace at the start of a line	join the line to the one above
Del at the end of a line	pull up the next line
Enter or F3	new line, or split the line at the cursor
F2	run
Esc	throw everything away (“(discarded)”)

Each line may be up to 255 characters long. If it does not fit the screen width, it goes on in the next screen row, marked with `+`. While typing, the text flows as in a word processor:

```
[retro.db]> [ Ln  2/ 2  Col  43/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: INSERT INTO computer(name, maker, year, cpu, ramkb) VALUES('Spectrum +2', 'Amstrad', 1986, 'Z80
  + ', 128),
 2: ('CPC 6128', 'Amstrad', 1985, 'Z80', 128);
```

Up to **20 lines** are possible. The statement is stored with its line breaks exactly as it was typed – `.schema` later shows a `CREATE TABLE` in the same form.

Always with F2? With `.set enter line`, Enter **always** starts a new line, even the first time; only F2 runs. `.set enter run` (default) switches back.

The history

Every input that has run goes into the **history**, also long ones over many lines. It is saved in `mmsql.his` and is still there after a restart.

Key	Effect
↑ / ↓	previous / next input (in several lines: only at the top or bottom edge)
PgUp / PgDn	previous / next input, wherever the cursor is

What has just been started but not yet run is not lost: paging all the way down brings it back. The option `history` (default 50) sets how many entries mmsql keeps.

Pasting from the PC

On the PC, SQL can comfortably be written in a text editor and then pasted into the terminal. The terminal then sends the lines as if they were typed very fast – with an Enter after each line. So that not every line runs on its own, mmsql watches out: **if more characters come straight after an Enter, it is pasted text.** mmsql then keeps the lines in the editor, and F2 runs everything.

How long “straight after” is, is set with `.set paste` (default 30 milliseconds). Some terminals wait after each pasted line; in Tera Term this is called *Paste delay per line*. Then mmsql’s paste time must be longer than this pause:

Tera Term: Paste delay per line	mmsql
0 (off)	`.set paste 30` (default)
200 ms	`.set paste 250`

The price: an Enter on a single line only runs after this time – at 250 ms that is hardly noticeable. To be quite sure, press F3 before pasting. If characters go missing while pasting, a small delay per character helps in Tera Term (*Setup → Serial port → Transmit delay*).

Here the table `makers` and its data were pasted from a PC editor and run with F2 (this table is needed from chapter 5 on – best paste it exactly like this):

```
CREATE TABLE makers(name TEXT PRIMARY KEY, country TEXT);
INSERT INTO makers VALUES
  ('Apple', 'USA'), ('Commodore', 'USA'), ('Tandy', 'USA'),
  ('Atari', 'USA'), ('IBM', 'USA'), ('Sinclair', 'UK'),
  ('Acorn', 'UK'), ('Amstrad', 'UK'), ('Dragon Data', 'UK'),
  ('Texas Instruments', 'USA');
```

```
[retro.db]> [ Ln  6/ 6  Col  32/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: CREATE TABLE makers(name TEXT PRIMARY KEY, country TEXT);
 2: INSERT INTO makers VALUES
 3:   ('Apple', 'USA'), ('Commodore', 'USA'), ('Tandy', 'USA'),
 4:   ('Atari', 'USA'), ('IBM', 'USA'), ('Sinclair', 'UK'),
 5:   ('Acorn', 'UK'), ('Amstrad', 'UK'), ('Dragon Data', 'UK'),
 6:   ('Texas Instruments', 'USA');
OK
OK, 10 row(s) changed, last ID 10
```

The rule for blocks

> **New in 1.2:** one single rule for everything that has more than one line.

As soon as an input has several lines – typed with F3 or pasted – it is a **block**. A block runs as if its lines had been typed one by one:

1. **Every SQL statement ends with `;` at the end of a line.** Only then does mmsql know that the statement is complete. The last one may do without.
2. **Dot commands run along**, when they stand in a line of their own between two statements.
3. **The output is the same as at the prompt.**
4. **The first SQL error stops the rest:** “Not run: the rest of the block.”

A block with dot commands:

```
.mode line
SELECT * FROM makers WHERE name = 'Acorn';
.mode table
```

```
[retro.db]> [ Ln  3/ 3  Col  12/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: .mode line
 2: SELECT * FROM makers WHERE name = 'Acorn';
 3: .mode table
Mode: line
name    = Acorn
country = UK
 1 row(s)
Mode: table
```

A block with a typing error in line 2 – line 3 does not run any more:

```
UPDATE makers SET country = 'UK' WHERE name = 'Acorn';
SELECT nam FROM computer WHERE id = 5;
SELECT name, ramkb FROM computer WHERE id = 5;
```

```
[retro.db]> [ Ln  3/ 3  Col  47/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: UPDATE makers SET country = 'UK' WHERE name = 'Acorn';
 2: SELECT nam FROM computer WHERE id = 5;
 3: SELECT name, ramkb FROM computer WHERE id = 5;
OK, 1 row(s) changed
Error: SQL: no such column: nam
  Statement ( 37 chars): SELECT nam FROM computer WHERE id = 5
Not run: the rest of the block.
```

Why the `;` matters so much: without `;` the next line still belongs to the same statement. That is intended, because only so can a statement run over several lines. If it is missing, mmsql joins two statements and SQLite reports a syntax error. Even a line starting with a dot then still belongs to the statement, just as in `sqlite3`.

In short:

Input	Rule
one line	runs with Enter, `;` not needed
several lines (block)	every statement ends with `;` at the end of a line
script with `.read`	like a block (chapter 9)

Paging through results

mmsql shows longer results page by page. At the end of a page:

Key	Effect
Enter or N	next page
B	previous page
Q or Esc	stop

How many rows go on a page is set with `.set rows` (default 20).

Ctrl-C

If an SQL statement runs too long, **Ctrl-C** stops it, and mmsql carries on. What happens to the data is explained in chapter 10.

Umlauts and other special characters

SQLite stores text as UTF-8, with all umlauts, accents and special characters. The CMM2 font does not know these characters. So mmsql shows each of them as **one** placeholder, a middle dot `·` (in the terminal it appears as `ú`). That keeps the columns straight. In the database and in files the real text is kept.

Setting	Effect
`.set utf8 dot`	placeholder (default)
`.set utf8 raw`	send the bytes unchanged, for a terminal set to UTF-8
`.set dotchar 46`	another placeholder, here the full stop `.`

Such characters cannot be typed in mmsql. To use them in SQL, write their codes: `'M' || char(252) || 'ller'` gives “Müller”.

5. SQL step by step

Now things really get going: open `retro.db` (chapter 3) and type along. Anyone who has not yet created the table `makers` in chapter 4 should do so now (section “Pasting from the PC”) – it is needed from section 5.4 on.

A few ground rules first:

- SQL words such as `SELECT` or `WHERE` may be written in upper or lower case.
- **Texts** go in single quotes: `'Apple'`. In texts, case matters: `'apple'` finds nothing.
- **Numbers** go without quotes: `1982`.
- Names of tables and columns are simply written as they are: `computer`, `year`.

5.1 Asking questions: SELECT

The basic form of every query:

```
SELECT columns FROM table WHERE condition ORDER BY sorting;
```

Only `SELECT … FROM …` is required; the rest is added as needed.

Choosing columns and filtering: name, processor and memory of the Sinclair computers:

```
[retro.db]> SELECT name, cpu, ramkb FROM computer WHERE maker = 'Sinclair';
name        cpu   ramkb
----------------------
ZX81        Z80   1
ZX Spectrum Z80   48
QL          68008 128
 3 row(s)
```

`SELECT *` shows all columns, a list such as `name, cpu, ramkb` only these, in this order.

Sorting and limiting: `ORDER BY` sorts, `DESC` turns the order round (largest first); without it, the order is ascending. After the comma comes what decides a tie. `LIMIT 5` shows only the first five:

```
[retro.db]> SELECT name, year FROM computer ORDER BY year DESC, name LIMIT 5;
name      year
--------------
Amiga 500 1987
520ST     1985
CPC 464   1984
Macintosh 1984
QL        1984
 5 row(s)
```

The five newest computers – three of them came out in 1984 and are sorted by name.

Combining conditions: with `AND` both conditions must be true, with `OR` one is enough. `NOT` turns a condition round.

```
[retro.db]> SELECT name, cpu, ramkb FROM computer WHERE cpu = '6502' AND ramkb >= 32;
name              cpu  ramkb
---------------------------
BBC Micro Model B 6502 32
Oric-1            6502 48
Electron          6502 32
 3 row(s)
```

The comparisons at a glance:

Comparison	Meaning	Example
`=`	equal	`maker = 'Apple'`
`<>` or `!=`	not equal	`cpu <> 'Z80'`
`<`, `<=`, `>`, `>=`	less, greater	`year >= 1984`
`BETWEEN a AND b`	from a to b (both included)	`year BETWEEN 1981 AND 1982`
`IN (…)`	one of the list	`maker IN ('Acorn', 'Oric')`
`LIKE`	text pattern	`name LIKE 'ZX%'`
`IS NULL`, `IS NOT NULL`	no value / a value	`year IS NULL`

Text patterns with LIKE: `%` stands for any number of characters (also none), `_` for exactly one. For English letters `LIKE` does not tell upper and lower case apart.

```
[retro.db]> SELECT name FROM computer WHERE name LIKE 'ZX%';
name
-----------
ZX81
ZX Spectrum
 2 row(s)
```

A list with IN. Longer queries are best written over several lines (F3 after each line, F2 at the end – chapter 4):

```
SELECT name, maker FROM computer
  WHERE maker IN ('Acorn', 'Sinclair')
  ORDER BY maker, name;
```

```
[retro.db]> [ Ln  3/ 3  Col  24/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, maker FROM computer
 2:   WHERE maker IN ('Acorn', 'Sinclair')
 3:   ORDER BY maker, name;
name              maker
------------------------
BBC Micro Model B Acorn
Electron          Acorn
QL                Sinclair
ZX Spectrum       Sinclair
ZX81              Sinclair
 5 row(s)
```

A range with BETWEEN:

```
SELECT name, year FROM computer
  WHERE year BETWEEN 1981 AND 1982
  ORDER BY year, name;
```

```
[retro.db]> [ Ln  3/ 3  Col  23/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, year FROM computer
 2:   WHERE year BETWEEN 1981 AND 1982
 3:   ORDER BY year, name;
name              year
----------------------
BBC Micro Model B 1981
IBM PC            1981
VIC-20            1981
ZX81              1981
Commodore 64      1982
Dragon 32         1982
ZX Spectrum       1982
 7 row(s)
```

Each value only once with DISTINCT: which processors are there at all?

```
[retro.db]> SELECT DISTINCT cpu FROM computer ORDER BY cpu;
cpu
-----
6502
6510
68000
68008
6809
8088
Z80
 7 row(s)
```

> **Remember: NULL is different.** A field without a value is NULL. Nothing can be compared with NULL: `year = NULL` is **never** true, not even for empty fields. The right way is `year IS NULL`. By default mmsql shows NULL as an empty field; with `.nullvalue NULL` the word `NULL` appears (section 5.4).

5.2 Calculating and converting

The column list may also calculate. `AS` gives the result a name:

```
[retro.db]> SELECT name, ramkb, ramkb * 1024 AS bytes FROM computer WHERE ramkb >= 128;
name      ramkb bytes
----------------------
QL        128   131072
Macintosh 128   131072
520ST     512   524288
Amiga 500 512   524288
 4 row(s)
```

SQLite has many functions for texts, for example `upper()` (capitals), `length()` and `substr(text, from, length)` (a piece):

```
[retro.db]> SELECT upper(name), length(name), substr(name, 1, 3) FROM computer WHERE maker = 'Atari';
upper(name) length(name) substr(name, 1, 3)
-------------------------------------------
ATARI 800   9            Ata
520ST       5            520
 2 row(s)
```

`CASE` sorts values into classes – like `SELECT CASE` in BASIC:

```
SELECT name, ramkb,
  CASE WHEN ramkb < 16  THEN 'small'
       WHEN ramkb < 128 THEN 'medium'
       ELSE 'large' END AS size
  FROM computer
  WHERE maker = 'Commodore';
```

```
[retro.db]> [ Ln  6/ 6  Col  29/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, ramkb,
 2:   CASE WHEN ramkb < 16  THEN 'small'
 3:        WHEN ramkb < 128 THEN 'medium'
 4:        ELSE 'large' END AS size
 5:   FROM computer
 6:   WHERE maker = 'Commodore';
name         ramkb size
------------------------
PET 2001     4     small
VIC-20       5     small
Commodore 64 64    medium
Amiga 500    512   large
 4 row(s)
```

More useful functions: `lower()`, `trim()`, `replace(text, old, new)`, `round(number, places)`, `abs()`, `coalesce(a, b)` (the first one that is not NULL), `printf('%05d', number)` and `||`, which joins two texts: `name || ' (' || maker || ')'`.

5.3 Counting and summarising

How many computers are in the table?

```
[retro.db]> SELECT count(*) FROM computer;
count(*)
--------
18
 1 row(s)
```

Aggregate functions sum up many rows in one value: `count()` counts, `min()` and `max()` find the smallest and largest value, `avg()` the average, `sum()` the total.

```
[retro.db]> SELECT min(year), max(year), avg(ramkb), sum(ramkb) FROM computer;
min(year) max(year) avg(ramkb)       sum(ramkb)
-----------------------------------------------
1977      1987      91.2222222222222 1642
 1 row(s)
```

Grouping with GROUP BY: how many computers does each maker have? `GROUP BY maker` forms a group for each maker, `count(*)` counts in each group:

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

```
[retro.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)
```

This is one of the moments that show what a database can do: a question that would take a while with pencil and list is answered at once by one line of SQL.

Filtering groups with HAVING: `WHERE` filters rows **before** grouping, `HAVING` filters groups **afterwards**. Which processors are in at least three computers?

```
SELECT cpu, count(*) AS models
  FROM computer
  GROUP BY cpu
  HAVING count(*) >= 3
  ORDER BY models DESC;
```

```
[retro.db]> [ Ln  5/ 5  Col  24/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT cpu, count(*) AS models
 2:   FROM computer
 3:   GROUP BY cpu
 4:   HAVING count(*) >= 3
 5:   ORDER BY models DESC;
cpu   models
------------
6502  7
Z80   4
68000 3
 3 row(s)
```

The order of the parts is always the same: `SELECT … FROM … WHERE … GROUP BY … HAVING … ORDER BY … LIMIT …`

5.4 Connecting two tables: JOIN

So far everything was in one table. Chapter 4 created the table `makers` – each maker with its country:

```
[retro.db]> SELECT * FROM makers;
name              country
------------------------
Apple             USA
Commodore         USA
Tandy             USA
Atari             USA
IBM               USA
Sinclair          UK
Acorn             UK
Amstrad           UK
Dragon Data       UK
Texas Instruments USA
 10 row(s)
```

Why not simply put the country in another column of `computer`? Because Commodore would then have “USA” four times. If something changes, it would have to be changed in four places, and sooner or later one of them would be wrong. In a database each fact is stored **once**, and the tables are **connected** in the query.

JOIN connects two tables by a condition. Here a row of `makers` belongs to a computer when its `name` is the computer’s `maker`:

```
SELECT computer.name, computer.maker, makers.country
  FROM computer
  JOIN makers ON makers.name = computer.maker
  WHERE year < 1982
  ORDER BY year, computer.name;
```

```
[retro.db]> [ Ln  5/ 5  Col  32/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT computer.name, computer.maker, makers.country
 2:   FROM computer
 3:   JOIN makers ON makers.name = computer.maker
 4:   WHERE year < 1982
 5:   ORDER BY year, computer.name;
name              maker     country
---------------------------------
Apple II          Apple     USA
PET 2001          Commodore USA
TRS-80 Model I    Tandy     USA
Atari 800         Atari     USA
BBC Micro Model B Acorn     UK
IBM PC            IBM       USA
VIC-20            Commodore USA
ZX81              Sinclair  UK
 8 row(s)
```

How to read it:

- `FROM computer JOIN makers` – take both tables,
- `ON makers.name = computer.maker` – and put together the rows that match,
- `computer.name` – as both tables have a column `name`, the table name in front says which one is meant.

Shorter with alias names: a short substitute name may follow the table name, here `c` and `h`. That saves typing:

```
SELECT c.name, h.country
  FROM computer c
  JOIN makers h ON h.name = c.maker
  WHERE h.country = 'UK'
  ORDER BY c.name;
```

```
[retro.db]> [ Ln  5/ 5  Col  19/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT c.name, h.country
 2:   FROM computer c
 3:   JOIN makers h ON h.name = c.maker
 4:   WHERE h.country = 'UK'
 5:   ORDER BY c.name;
name              country
------------------------
BBC Micro Model B UK
CPC 464           UK
Dragon 32         UK
Electron          UK
QL                UK
ZX Spectrum       UK
ZX81              UK
 7 row(s)
```

Who is missing? LEFT JOIN. A normal JOIN only shows rows that have a partner in **both** tables. The maker “Oric” is missing in `makers` – with `JOIN` the Oric-1 would simply disappear. `LEFT JOIN` keeps all rows of the left table (the one before `LEFT JOIN`) and fills in NULL where the partner is missing. To make NULL visible, set `.nullvalue NULL` first:

```
SELECT c.name, c.maker, h.country
  FROM computer c
  LEFT JOIN makers h ON h.name = c.maker
  WHERE c.maker IN ('IBM', 'Oric')
  ORDER BY c.name;
```

```
[retro.db]> .nullvalue NULL
NULL is shown as: [NULL]
[retro.db]> [ Ln  5/ 5  Col  19/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT c.name, c.maker, h.country
 2:   FROM computer c
 3:   LEFT JOIN makers h ON h.name = c.maker
 4:   WHERE c.maker IN ('IBM', 'Oric')
 5:   ORDER BY c.name;
name   maker country
--------------------
IBM PC IBM   USA
Oric-1 Oric  NULL
 2 row(s)
```

The other way round: which makers have the fewest computers – even none? `count(c.id)` only counts rows in which there is a computer:

```
SELECT h.name, count(c.id) AS computer
  FROM makers h
  LEFT JOIN computer c ON c.maker = h.name
  GROUP BY h.name
  ORDER BY computer, h.name
  LIMIT 3;
```

```
[retro.db]> [ Ln  6/ 6  Col  11/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT h.name, count(c.id) AS computer
 2:   FROM makers h
 3:   LEFT JOIN computer c ON c.maker = h.name
 4:   GROUP BY h.name
 5:   ORDER BY computer, h.name
 6:   LIMIT 3;
name             computer
-------------------------
Texas Instruments 0
Amstrad           1
Dragon Data       1
 3 row(s)
```

Texas Instruments is in `makers` but has no computer in the list – a clear case for a TI-99/4A.

Connecting and grouping: how many computers come from which country, and since when?

```
SELECT h.country, count(*) AS models, min(c.year) AS since
  FROM computer c
  JOIN makers h ON h.name = c.maker
  GROUP BY h.country;
```

```
[retro.db]> [ Ln  4/ 4  Col  22/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT h.country, count(*) AS models, min(c.year) AS since
 2:   FROM computer c
 3:   JOIN makers h ON h.name = c.maker
 4:   GROUP BY h.country;
country models since
--------------------
UK      7      1981
USA     10     1977
 2 row(s)
```

(Oric is missing here, because a normal JOIN does not find it. It is added in chapter 7.)

A query inside a query: which computers have more memory than the average? The inner query in brackets works out the average, the outer one compares:

```
SELECT name, ramkb FROM computer
  WHERE ramkb > (SELECT avg(ramkb) FROM computer);
```

```
[retro.db]> [ Ln  2/ 2  Col  51/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT name, ramkb FROM computer
 2:   WHERE ramkb > (SELECT avg(ramkb) FROM computer);
name      ramkb
---------------
QL        128
Macintosh 128
520ST     512
Amiga 500 512
 4 row(s)
```

5.5 Changing data: INSERT, UPDATE, DELETE

Insert a row with `INSERT INTO table(columns) VALUES(values);`. The `id` is left out – SQLite gives the next free number itself. mmsql reports how many rows changed, and the new number (`last ID`):

```
[retro.db]> INSERT INTO computer(name, maker, year, cpu, ramkb) VALUES('CPC 6128', 'Amstrad', 1985,
'Z80', 128);
OK, 1 row(s) changed, last ID 19
[retro.db]> SELECT * FROM computer WHERE id = 19;
id name     maker   year cpu ramkb
---------------------------------
19 CPC 6128 Amstrad 1985 Z80 128
 1 row(s)
```

Change values with `UPDATE table SET column = value WHERE condition;`. Our ZX81 has been given a 16K memory expansion:

```
[retro.db]> UPDATE computer SET ramkb = 16 WHERE name = 'ZX81';
OK, 1 row(s) changed
[retro.db]> SELECT name, ramkb FROM computer WHERE name = 'ZX81';
name ramkb
----------
ZX81 16
 1 row(s)
```

Delete rows with `DELETE FROM table WHERE condition;`, for example `DELETE FROM computer WHERE id = 19;`.

> **Careful:** `UPDATE` and `DELETE` **without** `WHERE` affect **all** rows of the table! First check with a `SELECT` with the same condition which rows will be hit.

When unsure, work inside a **transaction** (section 5.7): first `BEGIN`, then change and look, then `COMMIT` (keep) or `ROLLBACK` (undo everything). Here a `DELETE` without `WHERE` goes wrong – and is rescued:

```
[retro.db]> BEGIN;
OK
[retro.db]*> DELETE FROM computer;
OK, 19 row(s) changed
[retro.db]*> SELECT count(*) FROM computer;
count(*)
--------
0
 1 row(s)
[retro.db]*> ROLLBACK;
OK
[retro.db]> SELECT count(*) FROM computer;
count(*)
--------
19
 1 row(s)
```

Note the star in the prompt: `[retro.db]*>` shows that a transaction is open.

5.6 Creating tables: CREATE TABLE

Now a table of our own: which computers from the list are in the collection, in what condition and since when. For a `CREATE TABLE`, writing over several lines is especially worthwhile:

```
CREATE TABLE owned(
  id          INTEGER PRIMARY KEY,
  computer_id INTEGER NOT NULL,
  condition   TEXT DEFAULT 'working',
  bought      INTEGER
);
```

```
[retro.db]> [ Ln  6/ 6  Col   3/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: CREATE TABLE owned(
 2:   id          INTEGER PRIMARY KEY,
 3:   computer_id INTEGER NOT NULL,
 4:   condition   TEXT DEFAULT 'working',
 5:   bought      INTEGER
 6: );
OK
[retro.db]> INSERT INTO owned(computer_id, bought) VALUES(9, 1984);
OK, 1 row(s) changed, last ID 1
[retro.db]> SELECT * FROM owned;
id computer_id condition bought
------------------------------
1  9           working   1984
 1 row(s)
```

The building blocks of a column:

Part	Meaning
`INTEGER`, `REAL`, `TEXT`, `BLOB`	type: whole number, decimal number, text, binary data
`PRIMARY KEY`	key of the table; `INTEGER PRIMARY KEY` counts up by itself
`NOT NULL`	required field – there must be a value
`DEFAULT value`	used when no value is given on insert (here `'working'`)
`UNIQUE`	each value may occur only once
`REFERENCES table(column)`	foreign key: the value must exist in the other table (chapter 10)

> **Aha: SQLite is generous with types.** Unlike most databases, SQLite checks types only loosely. If a value fits the column, it is converted; otherwise it is stored as it comes. Try it – `typeof()` shows how a value is stored:

```
[:memory:]> CREATE TABLE t(num INTEGER, txt TEXT);
OK
[:memory:]> INSERT INTO t VALUES('42', 42), ('abc', 7);
OK, 2 row(s) changed, last ID 2
[:memory:]> SELECT num, typeof(num), txt, typeof(txt) FROM t;
num typeof(num) txt typeof(txt)
------------------------------
42  integer     42  text
abc text        7   text
 2 row(s)
```

The text `'42'` became a number in the INTEGER column, the number `42` became text in the TEXT column, and `'abc'` sits as text in the INTEGER column. (This example ran in a database in memory, chapter 9 – it does not change `retro.db`.) For strict types, write `STRICT` after the closing bracket of the column list; then SQLite rejects wrong types.

Changing and removing tables:

Statement	Effect
`ALTER TABLE t ADD COLUMN col TEXT;`	add a column
`ALTER TABLE t RENAME COLUMN old TO new;`	rename a column
`ALTER TABLE t DROP COLUMN col;`	remove a column
`ALTER TABLE t RENAME TO new;`	rename the table
`DROP TABLE t;`	delete the table with all data – no questions asked!

SQLite cannot change the type of a column directly. One creates a new table and copies the data: `CREATE TABLE new(…);`, `INSERT INTO new SELECT * FROM old;`, `DROP TABLE old;`, `ALTER TABLE new RENAME TO old;`. Without SQL, `.create` and `.alter` help (chapter 7).

5.7 Transactions: all or nothing

A **transaction** groups several changes: either all take effect, or none. It starts with `BEGIN` and ends with `COMMIT` (keep) or `ROLLBACK` (throw away):

```
[retro.db]> BEGIN;
OK
[retro.db]*> INSERT INTO owned(computer_id, condition, bought) VALUES(10, 'keyboard faulty', 1985);
OK, 1 row(s) changed, last ID 2
[retro.db]*> INSERT INTO owned(computer_id, bought) VALUES(18, 1989);
OK, 1 row(s) changed, last ID 3
[retro.db]*> COMMIT;
OK
[retro.db]> SELECT * FROM owned;
id computer_id condition       bought
-----------------------------------
1  9           working         1984
2  10          keyboard faulty 1985
3  18          working         1989
 3 row(s)
```

Transactions have two advantages:

1. **Safety:** if something stops halfway – an error, Ctrl-C, the power – the database stays as it was before `BEGIN`.
2. **Speed:** without a transaction, SQLite writes to the SD card after **every** change. In a transaction only once, at `COMMIT`. A hundred `INSERT`s in one transaction are much faster than a hundred single ones.

While a transaction is open, the prompt shows the **star**. If the `COMMIT` is missing on quitting, mmsql asks (chapter 10). `SAVEPOINT name`, `RELEASE name` and `ROLLBACK TO name` set intermediate points inside a transaction.

5.8 Index, view and trigger

An index makes searching faster, like the index of a book. With 18 rows nothing shows, with 18,000 a lot. An index is worthwhile on columns that are often searched or used to connect tables:

```
[retro.db]> CREATE INDEX computer_maker ON computer(maker);
OK
[retro.db]> .indexes
name                     tbl_name cols
-------------------------------------
computer_maker           computer maker
computer_year            computer year
sqlite_autoindex_makers~ makers   name
 3 row(s)   ~ = shortened (.mode line shows all)
```

`computer_year` was already in `computers.db`. The index with the long name `sqlite_autoindex_makers_1` was created by SQLite itself, because `name` is the primary key of `makers`. SQLite uses an index on its own when it helps; nothing changes in the query itself.

A view is a saved query that can be used like a table:

```
[retro.db]> CREATE VIEW computer_country AS SELECT c.name, c.year, h.country FROM computer c JOIN
makers h ON h.name = c.maker;
OK
[retro.db]> SELECT * FROM computer_country WHERE country = 'UK' AND year >= 1983;
name     year country
--------------------
Electron 1983 UK
CPC 464  1984 UK
QL       1984 UK
CPC 6128 1985 UK
 4 row(s)
```

The view stores no data, only the question. When `computer` or `makers` change, it shows the new state at once.

A trigger is a rule that does something by itself when data changes. Here the database records every change to a computer's memory. A trigger contains statements with `;` inside and ends with `END;` – best typed over several lines with F3:

```
CREATE TABLE ram_log(computer_id INTEGER, old_kb INTEGER, new_kb INTEGER);
CREATE TRIGGER ram_changed AFTER UPDATE OF ramkb ON computer
BEGIN
  INSERT INTO ram_log VALUES(old.id, old.ramkb, new.ramkb);
END;
```

```
[retro.db]> [ Ln  5/ 5  Col   5/255 ]__[ F2 = run  Esc = discard ]_____________________________________
 1: CREATE TABLE ram_log(computer_id INTEGER, old_kb INTEGER, new_kb INTEGER);
 2: CREATE TRIGGER ram_changed AFTER UPDATE OF ramkb ON computer
 3: BEGIN
 4:   INSERT INTO ram_log VALUES(old.id, old.ramkb, new.ramkb);
 5: END;
OK
OK
[retro.db]> UPDATE computer SET ramkb = 32 WHERE name = 'VIC-20';
OK, 1 row(s) changed
[retro.db]> SELECT * FROM ram_log;
computer_id old_kb new_kb
-------------------------
8           5      32
 1 row(s)
```

In the trigger, `old` and `new` are the row before and after the change. The trigger ran by itself with the `UPDATE`. `DROP TRIGGER ram_changed;` removes it again.

5.9 Long statements

> **New in 1.2:** statements over 255 characters, including queries with results.

In the editor each line has at most 255 characters, but a statement has up to 20 lines. If together it is longer than 255 characters, mmsql hands it to SQLite as a whole – queries show their result as usual, for paging and in every output mode. The same goes for pasted blocks and scripts with `.read`. The upper limit for one statement is the option `maxtext` (default 32,768 characters, chapter 11).

5.10 Date and time

SQLite works with date and time as text in the form `YYYY-MM-DD HH:MM:SS`:

Expression	Result
`date('now')`	today’s date from the CMM2 clock
`datetime('now')`	date and time
`date('2026-11-20', '+100 days')`	2027-02-28
`strftime('%Y', '2026-11-20')`	2026
`julianday('2026-12-24') - julianday('now')`	days until Christmas Eve

The time comes from the CMM2 clock (`TIME$` and `DATE$` in MMBasic). The SQLite module knows no time zones: the modifiers `localtime` and `utc` are missing.

5.11 When something goes wrong

On an error mmsql shows SQLite’s message, the statement and its length – and often a hint:

```
[retro.db]> SELECT * FROM computers;
Error: SQL: no such table: computers
  Statement ( 23 chars): SELECT * FROM computers
Hint: .tables lists the tables
```

The table is called `computer`, not `computers` – `.tables` would have shown it. The most common mistake mmsql catches earlier: a command without its dot.

```
[retro.db]> tables
Did you mean .tables? Commands start with a dot.
[retro.db]> schema computer
Did you mean .schema? Commands start with a dot.
```

If such a word starts a longer input, mmsql tries it as SQL. If SQLite reports the syntax error exactly at the first word (`near "tables": syntax error`), mmsql adds a reminder: “Hint: commands start with a dot”. More messages are explained in chapter 16.

5.12 What the SQLite module cannot do

The SQLite of the CMM2 (version 3.47.0) is trimmed down for the memory of the computer. Missing are:

missing	consequence
maths functions	no `sqrt()`, `sin()`, `power()`, `ln()` etc. – the four basic operations and `round()`, `abs()` are there
`EXPLAIN`, `ANALYZE`	no query plans or statistics
WAL mode	databases from the PC in WAL mode cannot be opened (see below)
extensions	no `load_extension()`
time zones	no `localtime` and `utc`

What is WAL mode? SQLite protects changes with a *journal*, so that the file stays intact after a crash. The default is the journal mode `delete`: during a change a file `…-journal` appears briefly and is deleted afterwards. In the mode `wal` (*Write-Ahead Logging*), changes first go into an extra file `…-wal` and are moved into the database later. Some PC programs switch this mode on, because it is faster there. The SQLite of the CMM2 cannot handle it. Switch such a database over on the PC before copying it: `PRAGMA journal_mode=delete;`

Available are, among others, JSON functions, window functions (`OVER`), `WITH` (common table expressions, also recursive), views, triggers, foreign keys, `UPSERT` (`ON CONFLICT`), `RETURNING`, `STRICT` tables, `VACUUM INTO` and `ATTACH`.

6. Looking around without SQL

For many questions about the database itself – which tables are there, how are they built? – mmsql has commands of its own. They start with a dot and need no SQL.

.tables – which tables are there?

`.tables` lists tables and views with the number of their columns. With a search pattern (as for `LIKE`: `%` = any text, `_` = one character) only the matching ones:

```
[retro.db]> .tables
name             type  ncol
--------------------------
computer         table 6
computer_country view  3
makers           table 2
owned            table 4
ram_log          table 3
 5 row(s)
[retro.db]> .tables h%
(no rows)
```

After the examples in chapter 5, `retro.db` has four tables and one view.

> **New in 1.2:** search patterns for `.tables`, `.schema`, `.indexes` and `.triggers`.

.schema – how was a table built?

`.schema` shows the `CREATE` statements – the blueprint. Without a name for all objects, with a name for one, with a pattern for all that match:

```
[retro.db]> .schema makers
CREATE TABLE makers(name TEXT PRIMARY KEY, country TEXT);
[retro.db]> .schema %log%
CREATE TABLE ram_log(computer_id INTEGER, old_kb INTEGER, new_kb INTEGER);
```

A table created over several lines (like `owned` in chapter 5) is shown just like that, line by line.

.columns – the columns of a table

```
[retro.db]> .columns makers
cid name    type notnull dflt_value pk
-------------------------------------
0   name    TEXT 0       NULL       1
1   country TEXT 0       NULL       0
 2 row(s)
```

Column	Meaning
`cid`	number of the column, from 0
`name`, `type`	name and type
`notnull`	1 = required field (`NOT NULL`)
`dflt_value`	default value (`DEFAULT`)
`pk`	1 = part of the primary key

.indexes and .triggers

`.indexes` (short `.ix`) shows all indexes with their columns, `.indexes computer` only those of one table. `.triggers` shows the triggers and their tables. Examples are in section 5.8.

.browse – paging through a table

`.browse table` (short `.b`) shows all rows page by page. `WHERE` and `ORDER BY` may follow. Here with 10 rows per page (`.set rows 10`, takes effect after a restart):

```
[retro.db]> .browse computer ORDER BY year DESC
id name        maker       year cpu   ramkb
-----------------------------------------
18 Amiga 500   Commodore   1987 68000 512
19 CPC 6128    Amstrad     1985 Z80   128
17 520ST       Atari       1985 68000 512
16 Macintosh   Apple       1984 68000 128
15 QL          Sinclair    1984 68008 128
14 CPC 464     Amstrad     1984 Z80   64
13 Electron    Acorn       1983 6502  32
12 Oric-1      Oric        1983 6502  48
11 Dragon 32   Dragon Data 1982 6809  32
10 ZX Spectrum Sinclair    1982 Z80   48
Rows 1 - 10   [Enter/N] next  [B] back  [Q] stop
id name              maker     year cpu  ramkb
---------------------------------------------
9  Commodore 64      Commodore 1982 6510 64
8  VIC-20            Commodore 1981 6502 32
7  IBM PC            IBM       1981 8088 16
6  BBC Micro Model B Acorn     1981 6502 32
5  ZX81              Sinclair  1981 Z80  16
4  Atari 800         Atari     1979 6502 8
3  TRS-80 Model I    Tandy     1977 Z80  4
2  PET 2001          Commodore 1977 6502 4
1  Apple II          Apple     1977 6502 4
Rows 11 - 19 (end)   [B] back  [Q] stop
```

At the end of a page: Enter or N goes on, B back, Q or Esc stops.

A few signs in tables are good to know:

Display	Meaning
value ends with `~`	shortened – a column is at most 24 characters wide (wider with `.width`, chapter 8)
`<40000 bytes>~!`	the value is longer than `maxtext` and was not read (chapter 11)
`(+2 column(s) not shown)`	more columns than fit on the screen – `.mode line` shows them all

If `~` or `~!` occurs, a line below the table explains it.

.info – the database at a glance

```
[retro.db]> .info
File:          retro.db
SQLite:        3.47.0
Page size:      4096 bytes
Pages:          8  (32 KB)
Free pages:     0  (0 B, VACUUM frees them)
Journal:       delete
Encoding:      UTF-8
Auto vacuum:   0
user_version:  0
Tables:        4
Views:         1
Indexes:       2
Triggers:      1
```

Line	Meaning
Page size, Pages	SQLite stores in pages; size × number = file size
Free pages	free pages after deletions – `.vacuum` gives them back (chapter 10)
Journal	how SQLite protects changes; on the CMM2 always `delete`
Encoding	how texts are stored, usually UTF-8
user_version	a number programs may use freely (`PRAGMA user_version = 2;`)

.databases and ATTACH – several databases at once

One database is always open. More can be attached with `ATTACH`, up to ten. Their tables are then reached with the name in front:

```
[retro.db]> ATTACH 'computers.db' AS orig;
OK
[retro.db]> .databases
name file
------------------------
main A:/sql/retro.db
orig A:/sql/computers.db
 2 row(s)
[retro.db]> SELECT count(*) FROM orig.computer;
count(*)
--------
18
 1 row(s)
[retro.db]> DETACH orig;
OK
```

This makes it easy to copy tables between files: `CREATE TABLE copy AS SELECT * FROM orig.computer;`. `main` is always the open database, `temp` an area for temporary tables.

7. Editing without SQL

For single rows and for the structure of tables there are commands that ask step by step. They show the SQL statement they build – a good way to learn along the way.

For all input:

Input	Meaning
Enter (empty)	with `.insert`: default value or new key; with `.edit`: keep the value
`NULL` or `\N`	NULL (no value)
`''`	empty text
`\NULL`	the text “NULL” (in case someone is really called that)

.insert – adding a row

The maker Oric is still missing in the table `makers` (section 5.4). `.insert` asks for each column and shows type, required field, default value and primary key:

```
[retro.db]> .insert makers
New row in makers. Enter = default (or automatic key), NULL or \N = NULL, '' = empty text
  name (TEXT, PK)
  > Oric
  country (TEXT)
  > UK
Insert? (y/n) y
Inserted, rowid 11
```

Now the JOIN from section 5.4 finds the Oric-1 too.

> **New in 1.2:** `.insert` fills tables with up to 64 columns (Generation 1: 32); until now it was at most 20.

.edit – changing a row

`.edit table key` fetches a row by its primary key. Each field appears in the editor with its value: Enter keeps it, otherwise it is changed like any input. Here the ZX81 gets its full name:

```
[retro.db]> .edit computer 5
Edit computer id = 5. Enter = keep, NULL or \N = NULL, '' = empty text
  id (INTEGER, PK)
  > 5
  name (TEXT)
  > Sinclair ZX81
  maker (TEXT)
  > Sinclair
  year (INTEGER)
  > 1981
  cpu (TEXT)
  > Z80
  ramkb (INTEGER)
  > 16
Save 1 change(s)? (y/n) y
Saved.
```

Saving only happens after the question, and all changes together in one transaction. Texts over 255 characters and BLOBs (binary data) appear as `(long text, n bytes)` or `(BLOB, n bytes)` and stay unchanged; those are changed with `UPDATE`.

If a table has no single-column primary key, mmsql uses the `rowid` – the internal row number every SQLite table has. `SELECT rowid, * FROM table;` makes it visible.

.delete – deleting a row

`.delete table key` shows the row and deletes it after a question. Texas Instruments has no computer in the list – out it goes:

```
[retro.db]> .delete makers Texas Instruments
name = Texas Instruments
country = USA
Delete this row? (y/n) y
Deleted: 1 row(s)
```

.create – creating a table

`.create` asks for name, type, primary key, required field and default of each column. An empty name ends the list. Here a wish list:

```
[retro.db]> .create wishlist
New table wishlist. Empty column name = done.
  column1 name: id
    type (INTEGER, TEXT, REAL, BLOB, NUMERIC, Enter = none): INTEGER
    primary key? (y/n) y
  column2 name: name
    type (INTEGER, TEXT, REAL, BLOB, NUMERIC, Enter = none): TEXT
    NOT NULL? (y/n) y
    default (Enter = none):
  column3 name: price
    type (INTEGER, TEXT, REAL, BLOB, NUMERIC, Enter = none): REAL
    NOT NULL? (y/n) n
    default (Enter = none): 0
  column4 name:
CREATE TABLE "wishlist"("id" INTEGER PRIMARY KEY,"name" TEXT NOT NULL,"price" REAL DEFAULT 0)
Create? (y/n) y
Created: wishlist
```

The finished statement is shown before it runs – a good way to learn `CREATE TABLE`. mmsql puts the names in quotes, so that names with spaces work too. The statement may be at most 255 characters long; bigger tables are created with SQL (section 5.6).

.alter – rebuilding a table

`.alter` shows the columns and offers four changes:

Key	Change
A	add a column
R	rename a column
D	remove a column
T	rename the table

```
[retro.db]> .alter wishlist
Columns of wishlist:
  id (INTEGER, PK)
  name (TEXT, NOT NULL)
  price (REAL, default 0)
[A] add column  [R] rename column  [D] drop column  [T] rename table  [Q] quit A
  new column name: note
    type (INTEGER, TEXT, REAL, BLOB, NUMERIC, Enter = none): TEXT
    NOT NULL? (y/n) n
    default (Enter = none):
ALTER TABLE "wishlist" ADD COLUMN "note" TEXT
Run? (y/n) y
Done.
```

8. Output and export

How mmsql shows the result of a query is decided by the **output mode**. It is set with `.mode` and applies to all following queries (it is saved in `mmsql.cfg`). Back to the usual table is always `.mode table`.

All examples use the same query:

```
SELECT name, maker, year FROM computer WHERE year < 1979;
```

The output modes

Mode	Output	good for
`table`	aligned columns, page by page (default)	reading on screen
`line`	one field per line	wide rows, long texts
`csv`	separated by commas, as RFC 4180	spreadsheets, data exchange
`list`	with a separator, without quotes	simple lists, own programs
`insert`	`INSERT` statements	moving data into another database
`markdown`	Markdown table	forums, README files, documentation
`json`	JSON list	web pages, Python, other programs

line – one field per line, the rows separated by an empty line. Long texts appear in full here, line breaks in them indented under the `=`:

```
[retro.db]> .mode line
Mode: line
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
name  = Apple II
maker = Apple
year  = 1977

name  = PET 2001
maker = Commodore
year  = 1977

name  = TRS-80 Model I
maker = Tandy
year  = 1977
 3 row(s)
```

csv – *comma-separated values*. A field is put in quotes if it contains a comma, a quote or a line break; quotes inside the text are doubled. The first line holds the column names (`.headers on`, default). Excel, LibreOffice & Co. open such files directly:

```
[retro.db]> .mode csv
Mode: csv
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
name,maker,year
Apple II,Apple,1977
PET 2001,Commodore,1977
TRS-80 Model I,Tandy,1977
 3 row(s)
[retro.db]> SELECT 'A "quote", with a comma' AS text;
text
"A ""quote"", with a comma"
 1 row(s)
```

list – like csv, but never with quotes. The separator is set with `.separator` (1 to 3 characters, `\t` for tab), the header line with `.headers on|off`. Here with a semicolon and without header – and everything set back afterwards:

```
[retro.db]> .mode list
Mode: list
[retro.db]> .separator ;
Separator: ;
[retro.db]> .headers off
Headers: off
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
Apple II;Apple;1977
PET 2001;Commodore;1977
TRS-80 Model I;Tandy;1977
 3 row(s)
[retro.db]> .separator ,
Separator: ,
[retro.db]> .headers on
Headers: on
[retro.db]> .mode table
Mode: table
```

insert – one `INSERT` statement for each row. The table name after `.mode insert` goes into the statements. So rows can be moved to another database:

```
[retro.db]> .mode insert computer
Mode: insert  (INSERT INTO computer)
[retro.db]> SELECT * FROM computer WHERE year < 1979;
INSERT INTO "computer" VALUES(1,'Apple II','Apple',1977,'6502',4);
INSERT INTO "computer" VALUES(2,'PET 2001','Commodore',1977,'6502',4);
INSERT INTO "computer" VALUES(3,'TRS-80 Model I','Tandy',1977,'Z80',4);
 3 row(s)
```

markdown – a table in Markdown format, as shown by GitHub, many forums and wikis. The columns are aligned so that they are readable as text too. For this, the query runs twice: the first run measures the widths. A `|` in the text becomes `\|`, a line break `<br>`.

```
[retro.db]> .mode markdown
Mode: markdown
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
| name           | maker     | year |
|----------------|-----------|------|
| Apple II       | Apple     | 1977 |
| PET 2001       | Commodore | 1977 |
| TRS-80 Model I | Tandy     | 1977 |
 3 row(s)
```

json – a list of objects. Numbers stay numbers, NULL becomes `null`, texts are escaped by the JSON rules. BLOBs appear as hex text or, with `.set blob base64`, in Base64:

```
[retro.db]> .mode json
Mode: json
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
[{"name":"Apple II","maker":"Apple","year":1977},
{"name":"PET 2001","maker":"Commodore","year":1977},
{"name":"TRS-80 Model I","maker":"Tandy","year":1977}]
 3 row(s)
```

> **New in 1.2:** `.mode markdown` and `.mode json`.

How NULL appears: .nullvalue

`.nullvalue text` sets what is shown for NULL – in `table`, `line`, `csv`, `list` and `markdown`. `.nullvalue NULL` makes NULL visible, `.nullvalue ''` shows it empty again (default). In `insert` it is always `NULL`, in `json` always `null`. The setting is saved.

Column widths: .width

> **New in 1.2:** `.width`.

In mode `table` a column is as wide as its longest value, at most 24 characters. `.width` sets widths from the left; `0` means “automatic”. Longer values are shortened with `~`:

```
[retro.db]> .width 10
Widths: 10  (others auto)
[retro.db]> SELECT name, maker, year FROM computer WHERE maker = 'Acorn';
name       maker year
--------------------
BBC Micro~ Acorn 1981
Electron   Acorn 1983
 2 row(s)   ~ = shortened (.mode line shows all)
[retro.db]> .width
Widths: auto
```

`.width 0 40` makes the second column 40 characters wide – good for long descriptions. A column may be as wide as the output width minus 1. `.width` lasts until mmsql ends; it is not saved.

Writing to a file: .once and .output

`.once file` sends the **next** query into a file; after that it goes back to the screen. Together with `.mode csv` this makes a file for a spreadsheet:

```
[retro.db]> .mode csv
Mode: csv
[retro.db]> .once old.csv
The next query goes to old.csv
[retro.db]> SELECT name, maker, year FROM computer WHERE year < 1979;
 3 row(s) written to old.csv
[retro.db]> .mode table
Mode: table
```

The file `old.csv` contains:

```
name,maker,year
Apple II,Apple,1977
PET 2001,Commodore,1977
TRS-80 Model I,Tandy,1977
```

`.output file` sends **all** following queries into the file, until `.output` without a file name switches back to the screen. `.print` writes headings in between. In mode `table` a file is written as a list with `|`:

```
[retro.db]> .output list.txt
Output goes to list.txt  (.output alone = back to screen)
[retro.db]> .print Computers by Acorn:
[retro.db]> SELECT name, year FROM computer WHERE maker = 'Acorn';
 2 row(s) written to list.txt
[retro.db]> .output
Closed: list.txt
Output: screen
```

The file `list.txt` contains:

```
Computers by Acorn:
name|year
BBC Micro Model B|1981
Electron|1983
```

A few things apply to all files:

- An existing file is overwritten.
- Lines end with CR LF, as usual on Windows.
- Texts and BLOBs are written in full, even very long ones. If a value is longer than `maxtext` (chapter 11), the export stops with a clear message instead of quietly writing a shortened value:

```
Stopped: a text of 40000 bytes (row 3, column note) is longer than
maxtext (32768). Use .set maxtext 40000 and try again.
notes.csv is incomplete.
```

What becomes of the values in exports

Value	csv / list	insert / `.dump`	json
NULL	text from `.nullvalue`	`NULL`	`null`
whole number	as stored (64 bit)	as stored	number
decimal number	15 significant digits	17 significant digits (exact)	17 digits
text	unchanged (UTF-8)	`'text'`, `'` doubled	`"text"`, escaped
text with line break	in quotes	`'a'||char(10)||'b'` – in one line	`"a\nb"`
BLOB	the raw bytes	`X'0102A0'`	hex or Base64

9. Data in and out: import, dump, scripts

Importing CSV files: .import

Many programs can save tables as CSV files: Excel, LibreOffice, address books, measuring devices. `.import` reads such a file into a table. Here is the file `members.csv`:

```
name,city,since
Ada,London,2019
Bob,Sydney,2020
"Clara ""CJ""",Berlin,2020
Dave,London,2021
```

The third line shows a special feature of CSV: a field containing quotes is itself put in quotes, and the inner ones are doubled. mmsql understands that, as well as commas and line breaks inside a quoted field (RFC 4180).

Into a new table: if the table does not exist yet, `.import` creates it. The first line supplies the column names, all columns get the type TEXT:

```
[retro.db]> .import members.csv member
Created table member with 3 column(s)
 4 row(s) imported into member
[retro.db]> SELECT * FROM member;
name       city   since
----------------------
Ada        London 2019
Bob        Sydney 2020
Clara "CJ" Berlin 2020
Dave       London 2021
 4 row(s)
```

Into an existing table: for columns with other types, create the table first. Then **all** lines of the file are data; `--skip 1` skips the header line:

```
[retro.db]> CREATE TABLE member2(name TEXT, city TEXT, since INTEGER);
OK
[retro.db]> .import --skip 1 members.csv member2
 4 row(s) imported into member2
```

Good to know:

- The separator comes from `.separator` and must be exactly one character here. For files with semicolons: `.separator ;` before the import.
- The import runs in one transaction – fast, and clean when there are problems: after ten bad records (wrong number of fields, too long) **nothing** is imported. Single bad records are reported with their number and skipped.
- Ctrl-C stops the import; then nothing is imported either.
- CSV has no NULL. An empty field (`,,` or `""`) becomes an empty text.
- Empty lines are skipped. mmsql recognises padding bytes at the end of the file (from XMODEM) as the end.

> **New in 1.2:** `.import` reads any number of columns (until now at most 20), and any number of fields may be longer than 255 characters. A record may be up to `maxtext` long.

The whole database as text: .dump

`.dump` writes a database as SQL text: first the blueprint of each table, then an `INSERT` for each row, finally indexes, views and triggers. With a table name only that table:

```
[retro.db]> .dump makers
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE makers(name TEXT PRIMARY KEY, country TEXT);
INSERT INTO "makers" VALUES('Apple','USA');
INSERT INTO "makers" VALUES('Commodore','USA');
INSERT INTO "makers" VALUES('Tandy','USA');
INSERT INTO "makers" VALUES('Atari','USA');
INSERT INTO "makers" VALUES('IBM','USA');
INSERT INTO "makers" VALUES('Sinclair','UK');
INSERT INTO "makers" VALUES('Acorn','UK');
INSERT INTO "makers" VALUES('Amstrad','UK');
INSERT INTO "makers" VALUES('Dragon Data','UK');
INSERT INTO "makers" VALUES('Oric','UK');
COMMIT;
```

Such a dump is the best backup there is: readable, can be opened with any text editor, read back with `.read` and also used on the PC with `sqlite3`. `.once` writes it to a file:

```
[retro.db]> .once retro.sql
The next query goes to retro.sql
[retro.db]> .dump
Dump written to retro.sql
```

About the format:

- Every statement ends with `;`. Line breaks in texts are written as `char(10)`, so that every `INSERT` statement stays on one line: `'first'||char(10)||'second'`.
- BLOBs (binary data) appear as `X'0A1B…'`.
- Decimal numbers are written with 17 digits, so that they are exactly the same after reading back. A decimal number with nothing after the point is written as `2.0`, so that it stays a decimal number, and infinity as `1e999`.

> **New in 1.2:** `.dump` asks SQLite for the real type of each value (`typeof`). Until now a BLOB without control characters became text after reading back, and a text with control characters became a BLOB. Now every value has exactly its type again after `.dump` and `.read`.

Running SQL scripts: .read

`.read file` runs SQL from a text file – a **script**. This can rebuild databases from a dump, load sample data, or do recurring evaluations with one command. A script is written on the PC with a text editor or on the CMM2 with `EDIT`.

A script follows the same rules as a block at the prompt (chapter 4):

1. **Every statement ends with `;` at the end of a line.** It may run over any number of lines.
2. Empty lines and lines starting with `--` (comments) are skipped.
3. Queries show their result as at the prompt.
4. These dot commands also run in a script: `.print` `.echo` `.bail` `.mode` `.headers` `.separator` `.nullvalue` `.timer` `.width` `.once` `.output`. There they give no confirmation, and their settings are not saved in `mmsql.cfg`. Other dot commands are reported with the line number and skipped.

An example, the file `ram.sql`:

```
-- ram.sql: RAM statistics per processor
.print === RAM per processor ===
SELECT cpu,
       count(*)   AS models,
       max(ramkb) AS max_kb
FROM computer
GROUP BY cpu
ORDER BY max_kb DESC;

.print === Done ===
```

```
[retro.db]> .read ram.sql
=== RAM per processor ===
cpu   models max_kb
------------------
68000 3      512
Z80   5      128
68008 1      128
6510  1      64
6502  7      48
6809  1      32
8088  1      16
 7 row(s)
=== Done ===
 1 statement(s) OK, 0 error(s), 10 line(s) read
```

At the end `.read` reports how many statements ran without error, how many errors there were and how many lines were read. Errors name the line in which the faulty statement **starts**.

Setting	Effect
`.bail on`	stop at the first error (otherwise after ten)
`.echo on`	show each statement before it runs – good for finding errors

More details:

- At the end of the file a last statement also runs without `;`.
- A `CREATE TRIGGER` ends with `END;`. Inside a trigger, write `CASE … END` without `;` directly after it, otherwise mmsql (like `sqlite3`) takes it for the end of the trigger. Safest: `(CASE … END)`.
- Statements over 255 characters run as a whole, up to `maxtext`; still longer ones are skipped, and `.read` carries on after their `;`.
- If a transaction is still open after the script (a `BEGIN` without `COMMIT`), mmsql points it out.
- `.open` and `.read` themselves do not run in a script. To run several scripts or databases one after the other, paste the commands as a block at the prompt (chapter 4) – all dot commands run there.

> **Note when moving from 1.1:** in mmsql 1.1 every line was a statement of its own, and no `;` was needed. Since 1.2 a statement only ends at the `;` – as in `sqlite3` and every other database. An old script without `;` now runs as one single, wrong statement.

```
-- old.sql (mmsql 1.1, without ;)
CREATE TABLE t(a)
INSERT INTO t VALUES(1)
```

```
[retro.db]> .read old.sql
Line 2: SQL: near "INSERT": syntax error
 0 statement(s) OK, 1 error(s), 3 line(s) read
```

The first statement starts in line 2 (line 1 is a comment) and runs to the end of the file – SQLite trips over the `INSERT`. The cure: put a `;` at the end of every line. Dumps from mmsql have always had it.

> **New in 1.2:** statements over several lines, query results and dot commands in scripts.

Without a file: the database in memory

> **New in 1.2:** `:memory:`.

If no database is open, mmsql uses a **database in memory** for SQL, `.read` and `.import`. It is there at once and fast, but is not saved anywhere. Good for trying things out, calculating and testing:

```
mmsql> SELECT 6 * 7 AS answer, date('2026-11-20', '+100 days') AS due;
(No database open - using a database in memory. .open file.db for a file.)
answer due
----------------
42     2027-02-28
 1 row(s)
[:memory:]> CREATE TABLE note(text TEXT);
OK
[:memory:]> .quit
The database in memory has 1 table(s) - they are lost (.backup file.db saves them).
Go on? (y/n) n
Cancelled - the database stays open.
[:memory:]> .backup notes.db
OK - copy written to notes.db
```

- The prompt shows `[:memory:]>`.
- All commands work, except `.restore`.
- If the database in memory has tables, `.open`, `.close` and `.quit` ask first, because its contents get lost. `.backup file.db` saves it as a file.
- `.open :memory:` opens it on purpose, `RUN "mmsql", ":memory:"` right at the start.

Exchange with the PC

- Transfer files by XMODEM or read the SD card on the PC.
- Databases from the CMM2 open on the PC with `sqlite3`, *DB Browser for SQLite* and any other SQLite program.
- Databases from the PC must be saved in journal mode `delete`, not `wal` (section 5.12). On the PC first: `PRAGMA journal_mode=delete;`
- XMODEM pads the last block (Tera Term with Ctrl-Z, the CMM2 with NUL bytes). `.read` and `.import` recognise that as the end of the file.
- A dump from `sqlite3` on the PC can be read with `.read`, as long as it uses no functions of a newer SQLite version (such as `unistr()` from SQLite 3.50).

10. Safety and maintenance

mmsql watches open transactions

If a transaction started with `BEGIN` has not been ended, `.quit`, `.close` and `.open` ask what should happen to it:

```
[retro.db]*> .quit
A transaction is open (BEGIN without COMMIT).
[C] commit  [R] rollback  [Esc] cancel C
Committed.
Closed: retro.db
```

Key	Effect
C	*commit* – keep the changes
R	*rollback* – throw the changes away
Esc	cancel – the database stays open, and so does the transaction

After an error inside a transaction, mmsql asks SQLite whether the transaction is still open. If SQLite has ended it itself, mmsql reports: “Note: SQLite has undone the open transaction.”

Ctrl-C: stopping without damage

An SQL statement that runs too long is stopped with **Ctrl-C**. mmsql then carries on and says exactly what SQLite has undone:

```
[retro.db]*> UPDATE big SET n = n + 1;
Interrupted (Ctrl-C).
The statement was undone. The whole transaction since BEGIN was undone.
[retro.db]>
```

What was running?	What is the state afterwards?
a query (`SELECT`)	nothing changed; an open transaction stays open
a change without a transaction	the change is completely undone
a change inside a transaction	the **whole** transaction since `BEGIN` is undone

`.read`, `.import` and `.dump` stop cleanly on Ctrl-C: `.import` then imports nothing, `.dump` reports the file as incomplete.

Elsewhere – at the prompt or while mmsql writes a file – Ctrl-C ends mmsql like any BASIC program. Even then the database stays intact: an open transaction is undone, nothing of it is written. With `OPTION BREAK 0`, Ctrl-C is switched off; mmsql then points out at the start that SQL cannot be stopped.

Backing up and restoring: .backup and .restore

`.backup file.db` writes a clean, compact copy of the open database. If the file exists already, mmsql asks before overwriting it.

> **New in 1.2:** `.restore`.

`.restore file.db` does the opposite: it replaces the open database with a backup.

```
[retro.db]> .backup retro_backup.db
OK - copy written to retro_backup.db
[retro.db]> DELETE FROM makers;
OK, 10 row(s) changed
[retro.db]> .restore retro_backup.db
Replace retro.db (44 KB) with retro_backup.db (44 KB)?
The present content of retro.db is lost. (y/n) y
Closed: retro.db
Restored: retro.db from retro_backup.db
[retro.db]> SELECT count(*) FROM makers;
count(*)
--------
10
 1 row(s)
```

`.restore` goes about it carefully:

1. It checks that the file really is an SQLite database.
2. It asks, because the present contents get lost (with an open transaction it asks like `.close`).
3. It first copies the backup next to the database (`retro.db-restore`). Only when this copy is complete is the database replaced. If the SD card is full, everything stays as it was.
4. A left-over `retro.db-journal` belongs to the old contents and is deleted.

The backup itself stays unchanged – it can be restored as often as wanted. A dump (chapter 9) is the second kind of backup: as readable text, also for the PC.

Checking: .check

Command	checks
`.check`	the most important things, fast (`PRAGMA quick_check`)
`.check full`	everything, including the indexes (`PRAGMA integrity_check`)

```
[retro.db]> .check
Running quick_check ...
OK - no problems found.
[retro.db]> .check full
Running integrity_check ...
OK - no problems found.
```

With large databases this takes a while, on Generation 1 even minutes (Ctrl-C stops). If `.check` finds problems, it lists them. Then rescue with `.backup new.db` what SQLite can still read, and carry on with the copy.

Tidying up: .vacuum

Deleted rows leave free pages in the file; it does not get smaller by itself. `.vacuum` rebuilds the file and shows the size before and after:

```
[retro.db]> .vacuum
Needs free space on the card of about the database size.
OK - 44 KB before, 44 KB after.
```

For this, SQLite briefly needs about as much free space on the card as the database is big. `.reindex` rebuilds all indexes (or those of one table, or one particular index).

Foreign keys: .fk and .fkcheck

A foreign key (`REFERENCES` in `CREATE TABLE`, section 5.6) says: this value must exist in another table. But SQLite only checks that when it is switched on – for the session, until mmsql ends:

```
[retro.db]> .fk
Foreign key enforcement: off
[retro.db]> .fk on
Foreign key enforcement: on
[retro.db]> .fkcheck
OK - no foreign key violations.
```

`.fkcheck` finds rows that break such a rule – even if checking was off when they were written.

Files on the card: .files and .kill

`.files` shows files with size and date; without anything else the `*.db` files of the current folder, otherwise by pattern (`.files *.sql`, `.files *`, `.files A:/data/*.db`):

```
[retro.db]> .files
Folder: A:/sql
name             size      modified
-----------------------------------------------
computers.db     12 KB     2026-11-20 09:12
notes.db         8192 B    2026-11-20 10:41
retro.db         44 KB     2026-11-20 10:52
retro_backup.db  44 KB     2026-11-20 10:49
 4 file(s)
```

Sizes below 10,000 bytes are shown exactly in bytes, larger ones in KB.

`.kill file` deletes a file after a question – always exactly one, without `*`. The open database cannot be deleted this way (`.close` first).

```
[retro.db]> .kill notes.db
Delete notes.db (8192 B)? (y/n) y
Deleted: notes.db
```

A file `…-journal` next to a database is the remains of an interrupted transaction. SQLite tidies it up itself the next time the database is opened. When a database is deleted with `.kill`, mmsql offers to delete the journal too.

11. Settings

.set – all options at a glance

`.set` alone shows all options with their value and a short explanation, sorted by topic: output, input, characters and texts, other.

```
[retro.db]> .set
mode      table         output mode of queries
headers   1             column names in csv/list (1 = on)
separator ,             separator for csv/list
nullvalue [NULL]        how NULL is shown
blob      hex           BLOBs in .mode json
width     100           output width (0 = as wide as the screen)
rows      20            rows per page (after restart)

enter     run           Enter on one line runs it (F3 = new line)
paste     30            ms after Enter: more keys = pasted (0 = off)
history   50            lines kept in the history (after restart)

utf8      dot           UTF-8 characters shown as ·
dotchar   250           placeholder for utf8 dot (code)
maxtext   32768         longest text read, in bytes (memory)

screen    0             MODE at the start (0 = as it is)
timer     0             show the run time (1 = on)
helpfile  mmsql_en.hlp  help file (e.g. mmsql_de.hlp)
File: A:/sql/mmsql.cfg
```

`.set name value` changes an option, saves it in `mmsql.cfg` at once and shows the changed line:

```
[retro.db]> .set nullvalue NULL
nullvalue [NULL]        how NULL is shown
[retro.db]> .set width 0
width     0 (99)        output width (0 = as wide as the screen)
```

`mmsql.cfg` sits next to the program and is a plain text file with lines `name=value`. It can also be edited or deleted on the PC (then the defaults apply again). The complete list with all allowed values is in appendix C.

The options explained

Output

- `mode`, `headers`, `separator`, `nullvalue` – as `.mode`, `.headers`, `.separator` and `.nullvalue` (chapter 8).
- `blob hex|base64` – how `.mode json` writes binary data.
- `width` – how wide mmsql prints tables, 40 to 250 characters. The default is the width of the screen (100 characters in MODE 1 with the standard font). If a table does not fit, mmsql leaves out columns on the right and says so. `width 0` means: always as wide as the screen, also after a change of MODE; `.set` then shows the actual width in brackets, e.g. `0 (99)`. The last column of the screen stays free, so that no line wraps by accident.
- `rows` – rows per page when paging, 5 to 100. Takes effect after a restart.

Input

- `enter run|line` – whether Enter on a single line runs it (`run`) or starts a new line (`line`), chapter 4.
- `paste` – the paste time in milliseconds, 0 to 1000 (0 = off), chapter 4.
- `history` – how many inputs the history keeps, 0 to 200. Takes effect after a restart.

Characters and texts

- `utf8 dot|raw` and `dotchar` – how special characters are shown, chapter 4.
- `maxtext` – the longest text that mmsql reads or writes in one go, in bytes (1024 to 1,048,576, default 32,768). mmsql keeps two memory areas of this size. On Generation 2 and 3 even 1 MB is no problem; on Generation 1 it is better to stay small. If there is not enough memory, mmsql sets the value back to 32,768.

Other

- `screen` – the screen mode (see below).
- `timer 1` – show the run time after each statement (the same as `.timer on`).
- `helpfile` – the help file, e.g. `mmsql_de.hlp`.

Screen mode: .set screen

> **New in 1.2:** `.set screen` and `.set width 0`.

The CMM2 knows many screen modes (`MODE 1` to `MODE 19`), from 320 × 200 to 1920 × 1080 pixels. The higher the resolution, the more rows and columns fit on the screen – a big gain for wide tables.

`.set screen n` changes the MODE at once and remembers it: at every start mmsql switches to this mode, on quitting back to the mode and font from before. `.set screen 0` (default) leaves the screen as it is.

MODE	Resolution	Characters (standard font)	Note
1	800 × 600	100 × 50	default after switching on
9	1024 × 768	128 × 64	
11	1280 × 720	160 × 60	widescreen
15	1280 × 1024	160 × 85	
16	1920 × 1080	240 × 90	widescreen, Generation 2 only

Which modes a monitor shows is described in the CMM2 manual under the command `MODE`. Together with `.set width 0` the output width follows the mode. mmsql sets the terminal window to the new size at every change.

More small commands

Command	Effect
`.timer on` / `.timer off`	show the run time after each statement
`.cls`	clear screen and terminal; also sets the terminal window back to the screen size
`.print text`	print one line (useful in scripts and with `.output`)
`.version` (`.ver`)	the versions of mmsql, SQLite and firmware

```
[retro.db]> .version
V1.2 - SQLite manager for the Colour Maximite 2 (G2)
(c) 2026 Matthias Hollatz - MIT License
SQLite 3.47.0, firmware 8.00.00b1
```

These three lines belong in every bug report.

Help on one command

`.help command` shows the help on one command, short and to the point:

```
[retro.db]> .help width
.width [n1 n2 ...]
Column widths in table mode (also .browse), from the first column on:
0 = automatic (as wide as the values, up to 24), others up to the output
width (.set width) - 1. Longer values are shortened (~).
  .width 0 40     2nd column 40 wide, the others automatic
  .width          all automatic again
Not saved: it holds until mmsql ends.
```

12. For users of MySQL, MariaDB, MS SQL Server & Co.

Anyone who has worked with MySQL, MariaDB, Microsoft SQL Server, Oracle, PostgreSQL or Sybase knows SQL – and will still be surprised in a few places. **Here there is only SQLite: one file, one user, no server.** That makes many things simpler and some things different.

What does not exist

from server databases	in SQLite
server, connection, users, `GRANT`, passwords	none of it – the database is a file; whoever has the file has the data
stored procedures and functions (`CREATE PROCEDURE`, PL/SQL, T-SQL blocks)	do not exist; logic belongs in the BASIC program or in triggers
`GO` as a separator (SQL Server, Sybase)	every statement ends with `;`
`USE database`	`.open file.db` – or `ATTACH` for several at once
sequences, `AUTO_INCREMENT`, `IDENTITY`	`INTEGER PRIMARY KEY` counts up by itself
own date and time types	date as text `'2026-11-20'` or as a number; functions `date()`, `datetime()`, `strftime()`
`BOOLEAN` as a type of its own	0 and 1 (`TRUE` and `FALSE` are understood)
`ALTER TABLE … MODIFY / ALTER COLUMN`	changing a type only via a new table (section 5.6)
maths functions in the SQLite of the CMM2	missing (section 5.12)

BEGIN is a transaction

In T-SQL and PL/SQL, `BEGIN … END` starts a block of statements. In SQLite, `BEGIN` always means “start a transaction” – like `BEGIN TRAN` in SQL Server or `START TRANSACTION` in MySQL. `BEGIN … END` as a block exists only in the body of a trigger:

```
CREATE TRIGGER t AFTER INSERT ON a
BEGIN
  UPDATE b SET n = n + 1;
END;
```

Same idea, different spelling

MySQL / MariaDB	SQL Server / Sybase	Oracle	SQLite
`LIMIT 10`	`TOP 10`	`FETCH FIRST 10 ROWS ONLY`	`LIMIT 10` (also `LIMIT 10 OFFSET 20`)
`CONCAT(a, b)`	`a + b`	`a || b`	`a || b`
`NOW()`	`GETDATE()`	`SYSDATE`	`datetime('now')`
`IFNULL(a, b)`	`ISNULL(a, b)`	`NVL(a, b)`	`ifnull(a, b)` or `coalesce(a, b)`
`` `name` ``	`[name]`	`"name"`	`"name"` – `` `name` `` and `[name]` work too
`SHOW TABLES`	`sp_tables`	`SELECT … FROM user_tables`	`.tables` or `SELECT name FROM sqlite_master;`
`DESCRIBE t`	`sp_help t`	`DESC t`	`.columns t` or `PRAGMA table_info(t);`
`INSERT … ON DUPLICATE KEY UPDATE`	`MERGE`	`MERGE`	`INSERT … ON CONFLICT(…) DO UPDATE`

Types are recommendations

SQLite stores whatever comes in each column. The column type is a **recommendation** (*type affinity*): if a value fits, it is converted, otherwise it is stored as it is (example in section 5.6). `VARCHAR(20)` is allowed, but the 20 is not checked. For strict types, create the table with `STRICT`:

```
CREATE TABLE t(id INTEGER PRIMARY KEY, name TEXT NOT NULL) STRICT;
```

What else stands out

- **Foreign keys** are only checked when switched on: `.fk on` (or `PRAGMA foreign_keys = ON;`), each time after opening.
- `LIKE` does not tell upper and lower case apart for English letters; `GLOB` does, and uses `*` and `?` as wildcards.
- **Texts in single quotes**, names in double quotes. `"Apple"` without a matching column is taken as text if need be – better not to rely on it.
- **Every table has a** `rowid`, an internal row number. A column `INTEGER PRIMARY KEY` is this number.
- **Locks apply to the whole file.** On the CMM2 with a single user, that does not matter.
- `RIGHT JOIN` **and** `FULL OUTER JOIN` exist since SQLite 3.39, so on the CMM2 too.

What mmsql does differently from the tools of the big databases

- At the prompt **one line** runs without `;` – as in many graphical tools. Several lines are a block, and there `;` ends every statement as in `mysql`, `sqlcmd` or `sqlplus`.
- Dot commands (`.tables`, `.schema`) replace commands such as `SHOW` or `DESCRIBE`; they come from the model, `sqlite3`.
- No `GO`, no `/` at the start of a line as in `sqlplus`.

13. Technical data and limits

All values come from the source code of mmsql 1.2. Most limits follow from a single rule of MMBasic: **a string is at most 255 characters long.** Where mmsql can do more, it uses LONGSTRING buffers – memory areas for texts of any length (appendix F, tip 4 in chapter 15). Their size is the option `maxtext`, default 32,768 characters.

Besides, SQLite itself has limits. The most important one: **a table has at most 2000 columns.** This is SQLite's default value; it has not been tried on the CMM2. Texts and BLOBs may be up to 1 GB in SQLite – on the CMM2 the memory sets a limit long before that.

Input

What	Limit	What does it mean?
one line in the editor	255 characters	That is how long an MMBasic string may be. Spread longer statements over several lines (F3).
lines of one input	20	More lines do not fit into the editor. Even longer statements belong in a script (`.read`).
one statement	`maxtext`, default 32,768 characters	applies at the prompt, in a block and in scripts. 20 lines of 255 characters are about 5,100 characters – so in the editor there is always room; only scripts reach the limit.
statements in one line	32	A single line does not take more statements separated by `:` .
history	0 to 200 entries, default 50	Long entries over many lines count as one.
value with `.insert` and `.edit`	255 characters	Enter longer values with `INSERT` or `UPDATE` in SQL.
paste time	0 to 1000 ms	see chapter 4

Output

What	Limit	What does it mean?
columns in `table` and `.browse`	64 (Generation 1: 32)	One page of the table is held in memory, so the number of columns is fixed. Wider results are shown in full by `.mode line`, `csv` and `json`.
columns in `csv`, `list`, `line`, `insert`, `json`	as SQLite, so up to 2000	These modes write value by value and need no buffer.
columns in `markdown`	2000	that many column widths are kept by mmsql
width of a column in `table`	automatically at most 24 characters, with `.width` up to the output width − 1	Longer values end with `~`.
length of a column name	64 characters	longer names are shortened on display
one text or BLOB	up to `maxtext` (1,024 to 1,048,576 bytes)	On screen a longer value appears as `~!`; an export stops with a message instead of writing the value shortened. Cure: set `.set maxtext` higher.
output width (`width`)	40 to 250 characters	
rows per page (`rows`)	5 to 100	
names with `.schema` and `.files`	200	With more matches, give a search pattern.
decimal numbers	15 digits on screen, 17 in `insert`, `.dump`, `json`	With 17 digits a number is exactly the same after reading back.
whole numbers	64 bit	up to about ±9.2 quintillion, without rounding

Commands

What	Limit	What does it mean?
columns with `.insert` and `.edit`	64 (Generation 1: 32)	up to mmsql 1.1 it was 20
columns with `.import`	as SQLite, so up to 2000	mmsql itself does not count (up to mmsql 1.1: 20).
one CSV line with `.import`	about `maxtext` − 300 characters	mmsql turns every CSV line into a complete statement `INSERT INTO "table" VALUES('value','value',…)` and hands it to SQLite. This statement must fit into the buffer, which is `maxtext` big; 300 characters stay free for the beginning and end of the statement. With the default, a CSV line may be a good 32,000 characters long. If it is longer, `.import` reports “Record n: longer than maxtext (.set maxtext) - skipped” – then `.set maxtext 100000` and try again.
separator	1 to 3 characters, exactly one with `.import`	For **output** the separator may be longer, e.g. `.separator " | "` for a nicely readable list. For **reading in**, mmsql checks the file character by character; there only a single character such as `,`, `;` or tab (`\t`) works.
statement of `.create` and `.alter`	255 characters	Create bigger tables with `CREATE TABLE` in SQL.
errors before stopping (`.read`, `.import`)	10	`.import` then takes nothing at all.
`.nullvalue`	20 characters	
attached databases (`ATTACH`)	10	limit of SQLite
size of a database	as big as the SD card allows	

Memory

What	Value (measured on Generation 2, 8.00.00b1)
program	about 100 KB, 4269 lines after loading – 19 % of program memory
data	about 0.7 MB, of which twice `maxtext` for the LONGSTRING buffers
`maxtext` 1 MB	about 2 MB more – no problem on Generation 2 and 3

Files

File	Contents
`mmsql.cfg`	settings, one line `name=value` per option
`mmsql.his`	history, one line per entry. If an entry has several lines, the further ones start with the control character `CHR$(30)`; a line of exactly 255 characters is continued with `CHR$(31)`. Files of mmsql 1.1 stay readable.
exports	line end CR LF, as on Windows

File numbers: mmsql uses #7 (`.read`, `.import`, `.restore`, help), #8 (`.once`, `.output`) and #9 (`mmsql.cfg`, `mmsql.his`). When taking parts of mmsql into your own programs, do not use these numbers there at the same time.

What mmsql sets at the start

Setting	why	afterwards
`OPTION CRLF CR`	otherwise Enter gives CR and LF in firmware 8 – mmsql would see two keys	only while mmsql runs
terminal window = screen size	terminal and screen show exactly the same	stays
`MODE` from `.set screen`	only if set	back at quitting

14. mmsql and sqlite3 – an honest comparison

The model for mmsql is `sqlite3`, the command-line tool maintained by the SQLite developers themselves. It is written in C, runs on every PC and has more than twenty years of development behind it. mmsql is written in MMBasic and made for the CMM2. Here, honestly, is what is the same, what mmsql does on top and what it lacks.

The same or nearly the same

Area	as in sqlite3
SQL	the same SQLite language – it is the same database, after all
dot commands	`.open` `.tables` `.schema` `.indexes` `.databases` `.dump` `.read` `.import` `.mode` `.headers` `.separator` `.nullvalue` `.width` `.once` `.output` `.print` `.bail` `.echo` `.timer` `.backup` `.restore` `.help` `.quit`
search patterns	with `.tables`, `.schema`, `.indexes`, `.triggers`
output modes	`csv`, `list`, `line`, `insert`, `markdown`, `json`
scripts	`;` ends a statement, statements over several lines, triggers up to `END;` – with the same limit for `CASE … END;` inside a trigger
files	databases and dumps can be exchanged between both

What only mmsql does

Feature	what for
`.browse` with paging forwards and back	reading large tables on screen
`.insert`, `.edit`, `.delete`	editing rows by questions, without SQL
`.create`, `.alter`	creating and rebuilding tables step by step
`.columns`, `.info`, `.check`, `.fk`, `.fkcheck`	structure and state of the database without knowing the PRAGMAs
`.files`, `.kill`	files on the SD card, without leaving mmsql
`.set` with `mmsql.cfg`	settings stay over restarts
one line runs without `;`	more comfortable at the prompt
editor with status line, history over restarts	pleasant even without a PC keyboard
recognising pasted blocks	simply copy SQL from the PC into the terminal
star in the prompt, question about an open transaction	no change gets lost by accident
Ctrl-C with an exact message of what was undone	safe stopping
“Did you mean .tables?”	helps with a forgotten dot
`:memory:` by itself when no file is open	start straight away
screen and terminal at once, 1:1	work on the CMM2 and on the PC
help in English and German	
`.set screen`	screen mode of the CMM2

What sqlite3 does and mmsql does not

Feature of sqlite3	in mmsql
more output modes: `box`, `column`, `html`, `tcl`, `quote`, `tabs`, `ascii`, `qbox`	not there; `table` in mmsql is about what `column` is in sqlite3 (sqlite3 draws frames in `table`)
`.parameter` (named values for `?` and `:name`)	not there
`.recover` (rescuing data from damaged files)	not possible – needs an extension; mmsql has `.check` and `.backup`
`.eqp`, `.expert`, `.scanstats` (query plans)	not possible – `EXPLAIN` is missing in the SQLite module
`.load` (extensions)	not possible
`.shell`, `.system`, `.excel`, output to programs (`.output |cmd`)	not there – the CMM2 has no operating system with a command line, only the MMBasic prompt; other programs cannot be started from mmsql
`.import` with `--csv`, `--schema`, `--ascii`	only `--skip`; the separator comes from `.separator`
`.dump` with filters, `--data-only`, `--nosys`	only the whole database or one table
`.changes`, `.stats`, `.show`, `.limit`, `.lint`, `.sha3sum`, `.archive`, `.cd` …	not there; mmsql reports changed rows after every statement by itself
typing and showing umlauts and accents	not possible on the CMM2 (font); mmsql shows placeholders, the data stays real
speed	sqlite3 is C on a PC – many times faster. On the CMM2, SQLite itself also works in C; only the display is BASIC. An import of 500 rows × 8 columns takes about 3 seconds on Generation 2.

Where they behave differently

- **The semicolon at the prompt:** sqlite3 waits for more text after a line without `;`. mmsql runs a single line with Enter. In blocks and scripts both behave the same.
- **Long texts:** sqlite3 shows everything. mmsql shortens to the column width in `table` (`~`) and does not show values over `maxtext` at all (`~!`); `.mode line` shows everything up to `maxtext`.
- **Line breaks in a dump:** sqlite3 writes newer dumps with `replace(…)` or `unistr(…)`, mmsql with `char(10)`. Both read mmsql's dump; mmsql reads a dump of a newer sqlite3 as long as it uses no function the SQLite of the CMM2 does not know.
- **Error messages:** sqlite3 reports "Parse error near line 3", mmsql "Line 3: SQL: …" and shows the statement with its length.

Conclusion

For everyday work with databases on the CMM2, mmsql has everything important from sqlite3, plus a lot that makes life easier on a small computer without a mouse. What is missing is mainly what the SQLite module does not offer (query plans, extensions, rescue with `.recover`), what does not exist on the CMM2 (an operating system's command line, Excel) – and rare special commands. Anyone who needs one of those takes the file to the PC: it can be read there at once.

15. Under the bonnet – mmsql as a source of ideas

mmsql is written entirely in MMBasic, about 5000 lines, and the source code is open in `mmsql.bas` (MIT License: reusing it is expressly allowed). Anyone writing programs with SQLite will find ready-made answers in it to problems one otherwise stumbles over. This chapter is a map: what is where, and why it is built that way.

The structure: three layers

Prefix	Layer	Contents
`MS_`	core	everything that talks to the database – **without screen output**. These routines can be taken into your own programs almost unchanged.
`CLI_`	command line	the dot commands, output, files, history, settings
`ED_`	editor	the editor at the prompt

Every routine has a comment above it that says what it does and what it returns. Global variables carry the same prefix as their layer. With `OPTION EXPLICIT` every variable must be declared; typing errors in names show up at once.

Tip 1: catching SQL errors without the program stopping

An SQL error is a normal error in MMBasic: without precautions the program stops. So all database access in mmsql follows this pattern (`MS_Exec%`):

```
FUNCTION MS_Exec%(s$)
  ON ERROR CLEAR
  ON ERROR SKIP
  SQL EXEC s$
  IF MM.ERRNO THEN MS_SetErr s$ : MS_Exec% = -1 : EXIT FUNCTION
  ON ERROR CLEAR
  MS_Exec% = 0
END FUNCTION
```

`ON ERROR SKIP` applies only to the **next** command. Afterwards `MM.ERRNO` says whether there was an error, and `MM.ERRMSG$` holds the message, e.g. `SQL: no such table: computers`. The `ON ERROR CLEAR` before it makes sure no old error is left. Ctrl-C during an SQL statement is such an error too (“SQL: stopped by Ctrl-C”) – this is how mmsql stops queries without ending itself (`MS_SetErr`, `CLI_BrkMsg`).

Tip 2: is a transaction open?

SQLite does not say so directly. `MS_TxOpen%` asks with a trick: a `BEGIN` inside a transaction is an error (“cannot start a transaction within a transaction”). If the `BEGIN` works, none was open, and a `ROLLBACK` closes the empty transaction again at once.

Tip 3: reading a value, whatever its type

`SQL(n)` returns the value of column n – as a string if it is text, as a number if it is a number. But a program that shows any table does not know beforehand what comes. `MS_ValText$` tries it as a string first; if MMBasic reports “Expected a string”, it is a number. Then it reads the value as a decimal number **and** as a whole number and compares: if both are equal, it is a whole number (important for 64-bit values, which would be inexact as decimal numbers). `SQL(ISNULL n)` recognises NULL.

Tip 4: texts over 255 characters

An MMBasic string has at most 255 characters, SQLite texts can be megabytes. The bridge is the **LONGSTRING** array: an INTEGER array that holds texts of any length.

- **Reading:** `SQL(LONGSTRING n, buffer%())` fetches column n into the buffer and returns its length (`MS_LongRead%`). If it is longer than the buffer, the firmware reports an error with the length needed – mmsql reads it from the message and can thus say what `maxtext` should be set to.
- **Writing:** `SQL EXEC buffer%()` and `SQL QUERY buffer%()` take the whole statement as a LONGSTRING (`MS_ExecL%`). This allows statements of any length, for example in the import (`CLI_CsvRec%`, `CLI_ExecSB%`) and in `.insert` (`CLI_LBQuote`).
- **Printing piece by piece:** `LGETSTR$(buffer%(), from, length)` fetches a piece as a string (`CLI_LPiece$`, which does not cut a UTF-8 character in half).

> **Important for your own programs:** in firmware 8.00.00b1 (and 7.00.00RC6), `LGETSTR$` with exactly 255 characters writes one byte too many into memory. This can damage the open database (“disk I/O error” at `COMMIT`). So mmsql never asks for more than **254** characters. Found while building mmsql 1.1, fixed by Peter Mather in 8.00.00b2.

Long texts from `sqlite_master` (the blueprint of a table) are read by `MS_SqlPart$` without any LONGSTRING: with `substr(sql, ?, 200)` in pieces of 200 characters.

Tip 5: paging through large results

A query can return thousands of rows, but an array in memory has a fixed size. So `MS_Fetch%` and `CLI_Grid` always fetch only one page – and one row more, to know whether there is more:

```
SELECT * FROM (the user's query) LIMIT ? OFFSET ?
```

With `SQL QUERY q$, MS_PR% + 1, off%` the two `?` go along as values. A line break before the closing bracket makes sure that a `--` comment at the end of the query does not also “comment out” the bracket.

Tip 6: BLOB or text – asking for the real type

MMBasic returns BLOBs as strings, like texts. So how to recognise a BLOB? `.dump` asks SQLite itself: `CLI_DumpTab` builds a query for each table that returns the type of each column as well:

```
SELECT "a","b",typeof("a"),typeof("b") FROM "t"
```

The columns at the end of the row then say for each value `integer`, `real`, `text`, `blob` or `null`. `CLI_NumLit$` uses this to write a REAL such as 2.0 as `2.0`, so that it stays a REAL.

Tip 7: reading files with tabs

`LINE INPUT #` turns tabs into spaces – that is how MMBasic wants it. For data that is fatal. So `CLI_RdLine%` reads files in blocks with `INPUT$(255, #7)` and splits the lines at the LF itself; a CR before it is removed. Lines over 255 characters go into a LONGSTRING. For `.import`, `CLI_GetC$` reads character by character from the same buffer.

Tip 8: controlling screen and terminal at once

The console of the CMM2 writes to the screen **and** to the terminal. Escape sequences for the terminal would appear on the screen as gibberish. mmsql solves it like this:

- **Moving the cursor:** `PRINT @(x, y)` sets the position on the screen and sends the same position to the terminal (`ED_Up`). This works because the terminal window is exactly as big as the screen (`CLI_TermInit`).
- **Terminal only:** `CLI_TermOnly` switches briefly to `OPTION CONSOLE SERIAL`, writes, and switches back – but only at the start and with `.cls`. While typing, that would be dangerous: switching spoils characters that are just arriving over the serial line.
- **Cursor on the screen:** while `INKEY$` waits, the CMM2 shows no cursor. mmsql draws the character under the cursor inverted with `TEXT` (`ED_CursorOn`).

Tip 9: recognising pasted text

A person does not type on within 30 milliseconds after an Enter, a terminal pasting text does. `ED_Pasted%` waits briefly after an Enter (`.set paste`) and looks whether `INKEY$` already has the next character. The character is kept and not lost (`ED_Peek$`, `ED_Key$`). When many characters arrive at once, the editor only draws when none are waiting any more (`ED_Flush`) – that makes pasting fast.

Tip 10: taking SQL apart

Where does a statement end? Not at every `;` – not in `'texts'`, `"names"`, `[names]`, `` `names` ``, not in `-- comments` and `/* comments */`, and in a trigger only at `END;`. `CLI_Split%` splits one line; `CLI_Scan` with its helpers (`CLI_ScWord`, `CLI_ScNext%`) checks piece by piece for `.read` and blocks whether a statement is complete. The state (open quotes, comment) is kept across line and piece boundaries.

More finds

Routine	What it shows
`CLI_FwVer$`	the firmware version from `MM.VER` as text: `8.000001` → `8.00.00b1` (b1–b50 beta, RC from 51, 99 release)
`CLI_PutB64`	Base64 with `MATH(BASE64 ENCODE …)` in pieces of 189 bytes (divisible by 3: no padding in the middle)
`CLI_PutLit`	texts with control characters as SQL: `'a'||char(10)||'b'`
`CLI_HPut`, `CLI_LoadHis`	saving a list of entries over several lines in a text file, backwards compatible
`CLI_ScrSet%`, `CLI_ScrEnd`	changing `MODE` and restoring it at the end – with colour depth from `MM.INFO(MODE)` (e.g. 1.8) and `MM.INFO(FONT)`
`CLI_Restore`	replacing a file safely: copy first, check the size, then rename
`CLI_Show$`	counting UTF-8 characters and replacing them with placeholders
`MS_QId$`	putting names in `"…"`, so that names with spaces work too

A reminder of MMBasic peculiarities

- **Keywords cannot be variable names**, not even with a type suffix: `pos%`, `inv%`, `rnd%` do not work.
- **`AND` and `OR` work bitwise.** `IF a% AND b$ = "" THEN` is wrong when `a%` is, say, 2 (2 AND 1 = 0). Safe: `IF a% > 0 AND b$ = "" THEN`.
- **Both sides of `AND` are evaluated.** `IF i% <= LEN(s$) AND MID$(s$, i%, 1) = "x"` calls `MID$` even when `i%` is already too large.
- **Strings have at most 255 characters** – lists and help texts inside the program too. mmsql splits long lists into two strings (`CLI_IsCmd%`).
- **Never read a FUNCTION's own name inside it** – that is a recursive call. Use a local variable for values in between.
- **Do not call a FUNCTION like a SUB.** That only gives "Unknown command" when the program runs.
- **`LEFT$(s$, 0)` and `MID$` past the end** need care; mmsql has `CLI_Left$` and `CLI_From$` for this.

16. Messages and troubleshooting

Errors from SQLite

Errors from SQLite start with `Error: SQL:`, followed by the statement and its length. For long statements mmsql shows the first 200 characters and the real length.

Message	Meaning and cure
`no such table: x`	The table does not exist – a typing error? `.tables` lists them all.
`no such column: x`	The column does not exist – `.columns table` shows them all.
`near "x": syntax error`	SQL written wrongly, just before or at `x`. Often a missing comma, a missing bracket, a forgotten `;` in a block – or a dot command without its dot.
`UNIQUE constraint failed`	The value exists already but may only occur once (e.g. the primary key).
`NOT NULL constraint failed`	A required field was not filled.
`FOREIGN KEY constraint failed`	The value is missing in the connected table (only with `.fk on`).
`file is not a database`	not an SQLite file, encrypted, or saved in WAL mode on a PC (on the PC: `PRAGMA journal_mode=delete;`).
`unable to open database file`	check the file name, the folder and the SD card.
`disk I/O error`	SD card full or faulty.
`attempt to write a readonly database`	file or SD card write-protected.
`database is locked`	a transaction is still open – `COMMIT` or `ROLLBACK`.
`cannot commit - no transaction is active`	No transaction is running – perhaps it was already ended after an error or Ctrl-C.
`stopped by Ctrl-C`	stopped, see chapter 10.

Messages from mmsql

Message	Meaning
`Did you mean .xxx? Commands start with a dot.`	dot forgotten.
`Hint: commands start with a dot`	The syntax error is at the first word – probably a command without its dot.
`Hint: .tables lists the tables`	after `no such table`.
`Not run: n more statement(s) of this line.`	After an error the rest of the line did not run.
`Not run: the rest of the block.`	After an error the rest of the block did not run.
`(No database open - using a database in memory. …)`	No file was open; mmsql uses `:memory:` (chapter 9).
`No database open (.open file.db)`	The command needs an open database.
`Line n: …`	`.read`: error in the statement that starts in line n.
`Line n: .xxx is not run by .read`	This dot command does not run in scripts (chapter 9).
`Record n: … - skipped`	`.import`: record with the wrong number of fields or too long.
`Too many errors - stopped.` / `- nothing imported.`	`.read` or `.import` stopped after ten errors; `.import` has then taken nothing.
`Note: SQLite has undone the open transaction.`	After an error SQLite has ended the transaction.
`A transaction is still open: COMMIT or ROLLBACK`	after `.read`: the script left a transaction open.
`Stopped: a text of n bytes … is longer than maxtext`	export stopped – `.set maxtext n` and try again (chapter 8).
`(discarded)`	input thrown away with Esc.

When something does not work

Problem	Solution
When pasting from the PC every line runs on its own	paste time too short: `.set paste 250` with a 200 ms delay in the terminal – or press F3 before pasting (chapter 4).
Characters go missing when pasting	set a small delay per character in the terminal.
A pasted block gives a syntax error	Is the `;` missing at the end of a statement? In a block the next line otherwise still belongs to it.
An old script no longer runs	since 1.2, `.read` needs a `;` at the end of every statement (chapter 9).
Umlauts and accents appear as `·` (in the terminal `ú`)	intended – the CMM2 font has no such characters (chapter 4).
Columns are missing on the right	`(+n column(s) not shown)`: choose a wider screen mode (`.set screen`), make `.width` smaller, or use `.mode line`.
Terminal and screen show different things	`.cls` sets the terminal window back to the screen size.
Enter acts twice	This only happens outside mmsql; mmsql sets `OPTION CRLF CR` at the start.
Ctrl-C does not stop SQL	`OPTION BREAK 0` is set (mmsql points it out at the start). With `OPTION BREAK 4`, Ctrl-D takes over the role of Ctrl-C.
A database from the PC cannot be opened	on the PC `PRAGMA journal_mode=delete;` and save again.

Notes on the firmware

OPTION CRLF: without `OPTION PICOMITE` the Enter key sends CR and LF in firmware 8. mmsql sets `OPTION CRLF CR` at the start; this lasts only while mmsql runs.

LGETSTR$ in 8.00.00b1: writes one byte too many with exactly 255 characters (chapter 15); fixed in 8.00.00b2. mmsql avoids the bug on every firmware.

LINE INPUT # and tabs: `LINE INPUT #` turns tabs into spaces; so mmsql reads files differently (chapter 15).

MM.DEVICE$ under OPTION PICOMITE: returns the PicoMite name then; Generation 3 cannot be recognised this way, mmsql shows “G2/G3”.

Reporting a bug

Please report bugs and wishes on the *TheBackShed* forum in the mmsql thread. Please include:

- the three lines of `.version`,
- what was typed and what appeared,
- if possible a small database or script with which it can be repeated.

Appendix A: All commands at a glance

`[ ]` = may be left out, `a|b` = a or b. Short forms in brackets. File names without drive and folder refer to the current folder of MMBasic.

Command	Effect	Chapter
`.alter table`	add, rename, remove a column; rename the table	7
`.backup file.db`	write a clean copy of the database	10
`.bail on|off`	stop `.read` at the first error	9
`.browse table [WHERE …] [ORDER BY …]` (`.b`)	show rows page by page	6
`.check [full]`	check the database	10
`.close`	close the database	10
`.cls`	clear screen and terminal	11
`.columns table` (`.c`)	columns of a table	6
`.create table`	create a table step by step	7
`.databases` (`.db`)	open and attached databases	6
`.delete table key`	delete one row	7
`.dump [table]`	database as SQL text	9
`.echo on|off`	`.read` shows each statement	9
`.edit table key`	change one row	7
`.files [pattern]`	files with size and date	10
`.fk [on|off]`	show or switch foreign key checking	10
`.fkcheck [table]`	find foreign key violations	10
`.headers on|off`	header line in csv and list	8
`.help [command]` (`.h`)	help	2
`.import [--skip N] file.csv table`	import a CSV file	9
`.indexes [table]` (`.ix`, `.indices`)	indexes with their columns	6
`.info`	the database at a glance	6
`.insert table`	add one row	7
`.kill file`	delete one file	10
`.mode table|csv|list|insert [table]|line|markdown|json`	output mode	8
`.nullvalue text`	how NULL is shown	8
`.once file`	next query into a file	8
`.open file.db|:memory:` (`.o`)	open or create a database	3, 9
`.output [file]`	all queries into a file, without a file back	8
`.print text`	print one line	8, 9
`.quit` (`.q`, `.exit`)	end mmsql	2

`.read file.sql`	run an SQL script	9
`.reindex [name]`	rebuild indexes	10
`.restore file.db`	replace the database with a backup	10
`.schema [name|pattern]` (`.s`)	blueprint of the objects	6
`.separator x`	separator for csv and list	8
`.set [name value]`	show or change settings	11
`.tables [pattern]` (`.t`)	tables and views	6
`.timer on|off`	show the run time	11
`.triggers [table]`	triggers and their tables	6
`.vacuum`	make the file smaller	10
`.version` (`.ver`)	versions of mmsql, SQLite, firmware	11
`.width [n1 n2 …]`	column widths in `table`	8

The help on the CMM2 sorts the commands into groups:

Group	**Commands**
Database	`.open` `.close` `.info` `.backup` `.restore` `.vacuum` `.files` `.kill`
Look	`.tables` `.indexes` `.triggers` `.schema` `.columns` `.browse` `.databases`
Data	`.insert` `.edit` `.delete` `.import` `.read`
Structure	`.create` `.alter` `.reindex`
Check	`.check` `.fkcheck` `.fk`
Output	`.mode` `.headers` `.separator` `.nullvalue` `.width` `.once` `.output` `.dump`
Settings	`.set` `.timer` `.bail` `.echo` `.print` `.cls` `.version` `.help` `.quit`

Appendix B: Keys

At the prompt (editor)

Key	one line	several lines
Enter	run (with `.set enter line`: new line)	new line / split the line
F3	new line / split; on an empty prompt: start several lines	new line / split
F2	run	run
Esc	throw away	throw away
← →	cursor	cursor, also across line ends
Home / End	start / end of the line	start / end of the screen row
↑ ↓	history	one screen row; at the edge: history
PgUp / PgDn	history	history
Backspace	delete left	at the start of a line: join to the line above
Del	delete at the cursor	at the end of a line: pull up the next line

When paging: Enter or N = on, B = back, Q or Esc = stop.

At questions: y = yes, any other key = no. With an open transaction: C = commit, R = rollback, Esc = cancel.

Ctrl-C: stops a running SQL statement (chapter 10).

Appendix C: Options

All options are changed with `.set name value` and saved in `mmsql.cfg` at once.

Option	Default	allowed	Meaning
`mode`	table	table, csv, list, insert, line, markdown, json	output mode
`headers`	1	on/off or 1/0	header line in csv and list
`separator`	`,`	1 to 3 characters, `\t` or `tab`	separator for csv and list
`nullvalue`	(empty)	up to 20 characters, `''` = empty	how NULL is shown
`blob`	hex	hex, base64	BLOBs in `.mode json`
`width`	screen width	40 to 250, 0 = follows the screen	output width
`rows`	20	5 to 100	rows per page (after restart)
`enter`	run	run, line	what Enter does on one line
`paste`	30	0 to 1000 (ms), 0 = off	recognising pasted lines
`history`	50	0 to 200	entries in the history (after restart)
`utf8`	dot	dot, raw	how UTF-8 characters are shown
`dotchar`	250	code 33 to 255 or one character	placeholder for `utf8 dot`
`maxtext`	32768	1024 to 1048576	longest text or BLOB in bytes
`screen`	0	0 to 19	MODE at the start, 0 = as it is
`timer`	0	0, 1	show the run time
`helpfile`	mmsql_en.hlp	file name	help file, e.g. mmsql_de.hlp

Appendix D: SQL cheat sheet

Queries

```
SELECT column, column FROM table WHERE condition ORDER BY column [DESC] LIMIT n;
SELECT DISTINCT column FROM table;
SELECT column, count(*) FROM table GROUP BY column HAVING count(*) > 1;
SELECT a.x, b.y FROM a JOIN b ON b.id = a.b_id;
SELECT a.x, b.y FROM a LEFT JOIN b ON b.id = a.b_id;
SELECT x FROM t WHERE y > (SELECT avg(y) FROM t);
```

Conditions: `=` `<>` `<` `<=` `>` `>=` `AND` `OR` `NOT` `BETWEEN a AND b` `IN (a, b)` `LIKE 'A%'` `IS NULL` `IS NOT NULL`

Functions: `count()` `sum()` `avg()` `min()` `max()` · `length()` `upper()` `lower()` `substr()` `trim()` `replace()` `round()` `abs()` `coalesce()` `ifnull()` `typeof()` · `date()` `time()` `datetime()` `strftime()` `julianday()` · `||` joins texts

Changing

```
INSERT INTO t(a, b) VALUES(1, 'x');
INSERT INTO t(a, b) VALUES(1, 'x'), (2, 'y');
UPDATE t SET b = 'z' WHERE a = 1;
DELETE FROM t WHERE a = 1;
```

Building

```
CREATE TABLE t(id INTEGER PRIMARY KEY, name TEXT NOT NULL, n INTEGER DEFAULT 0);
ALTER TABLE t ADD COLUMN note TEXT;
DROP TABLE t;
CREATE INDEX t_name ON t(name);
CREATE VIEW v AS SELECT …;
CREATE TRIGGER tr AFTER UPDATE ON t BEGIN … ; END;
```

Transactions: `BEGIN` … `COMMIT` or `ROLLBACK`

Useful PRAGMAs: `PRAGMA table_info(t);` · `PRAGMA foreign_keys = ON;` · `PRAGMA integrity_check;` · `PRAGMA user_version;`

Appendix E: Changes since 1.1

New

- Editor over up to 20 lines with a status line (F3 new line, F2 run), options `enter` and `paste`, recognising pasted lines.
- One rule for input: one line runs with Enter, several lines are a block with `;` at the end of every statement.
- Statements and queries over 255 characters, with results.
- History of statements of any length, also after a restart; PgUp/PgDn.
- `.read` with statements over several lines, query results and dot commands.
- Database in memory (`:memory:`).
- `.mode markdown`, `.mode json`, `.set blob`.
- `.width`, `.print`, `.version`, `.restore`, `.set screen`, `.set width 0`.
- Search patterns for `.tables`, `.schema`, `.indexes`, `.triggers`.
- `.import` without limits for columns and long fields; `.insert` up to 64 columns.
- `.dump` with the real type of every value.
- Shorter prompt: `[retro.db]>` instead of `mmsql[retro.db]>`.
- Terminal and screen show exactly the same.
- `.set` sorted by topic.

Changed

- **`.read` needs `;` at the end of every statement** (as sqlite3). Scripts without `;` no longer run correctly.
- Error messages show the statement as it was typed, over several lines line by line.
- The hint “commands start with a dot” only appears when the error is at the first word.
- `--` comments end at the end of the line.

Fixed

- A SELECT of more than 239 characters could end mmsql with an error.
- `.schema` shows statements over several lines without a “staircase” in the terminal.
- `.dump` writes a REAL such as 2.0 as `2.0` (it stays REAL) and infinity as `1e999` (SQLite can read it back).
- Very large integers (above 2^{53}) are shown and exported exactly, no longer rounded as a float.

Appendix F: Glossary

Aggregate function – sums up many rows in one value: `count()`, `sum()`, `avg()`, `min()`, `max()`. With `GROUP BY` for each group separately (section 5.3).

Alias – a substitute name in a query, written with `AS` or simply after it: `count(*) AS models`, `FROM computer c` (section 5.4).

ATTACH – attaches another database file to the open one; its tables are then called `name.table` (chapter 6).

BLOB – *Binary Large Object*: binary data such as pictures or program code, as opposed to text. In exports mmsql shows them as hex (`X'0A1B'`) or Base64.

Block – an input of several lines, typed with F3 or pasted. Every statement in it ends with `;` (chapter 4).

CSV – *comma-separated values*: a text file with one table row per line, the values separated by commas. Spreadsheets read and write this format (chapters 8 and 9).

Database – a file in which data is kept in good order in tables. With SQLite a database is exactly one file (chapter 1).

Dot command – a command to mmsql itself, not to the database. It starts with a dot: `.tables`.

Dump – the whole database as SQL text: all `CREATE` and `INSERT` statements with which it can be rebuilt (chapter 9).

Foreign key – a column whose value must exist in another table, created with `REFERENCES`. SQLite only checks it after `.fk on` (chapter 10).

Index – a directory for a column with which SQLite searches faster, like the index of a book (section 5.8).

JOIN – connects two tables in a query by a condition. `LEFT JOIN` also keeps the rows without a partner (section 5.4).

Journal – a helper file with which SQLite protects changes: if something stops, it is used to restore the old state. On the CMM2 it is called `file.db-journal` and only exists during a change (chapter 10).

JSON – a text format for data that web pages and many programs understand (chapter 8).

LONGSTRING – in MMBasic a memory area (an INTEGER array) for texts over 255 characters. mmsql uses LONGSTRINGs for long statements and values (chapter 15, tip 4).

Markdown – a simple way of writing headings, lists and tables in text, which forums and GitHub display nicely (chapter 8).

maxtext – mmsql option: the longest text that mmsql reads or writes in one go, default 32,768 characters (chapter 11).

:memory: – a database that exists only in memory and disappears on closing (chapter 9).

NULL – “no value”. Not the same as 0 or an empty text; tested with `IS NULL` (chapter 1 and section 5.1).

PRAGMA – an SQLite command that shows or changes settings of the database, for example `PRAGMA foreign_keys = ON;`. Many dot commands of mmsql use PRAGMAs internally.

Primary key – the column that identifies each row, usually a number `id` (chapter 1 and section 5.6).

Prompt – the input line, in mmsql `mmsql>` or `[file.db]>`. A star `*` in it shows an open transaction.

Query – a question to the database, in SQL a `SELECT`. The answer is a table of rows and columns (section 5.1).

Record – one row of a table, for example one computer.

rowid – the internal row number that every SQLite table has. A column `INTEGER PRIMARY KEY` is this number (chapter 7).

Schema – the blueprint of a database: which tables with which columns, indexes, views and triggers there are. `.schema` shows it (chapter 6).

Script – a text file with SQL statements, run with `.read` (chapter 9).

SQL – *Structured Query Language*, the language for talking to databases (chapter 5).

SQLite – the database in the CMM2 firmware: a library without a server, one database per file (chapter 1).

Table – data in rows and columns, like a sheet in a spreadsheet.

Terminal – a program on the PC (e.g. Tera Term) that is connected to the CMM2 by the serial cable and shows the same as the screen (chapter 2).

Transaction – several changes that only take effect together: `BEGIN` … `COMMIT` or `ROLLBACK` (section 5.7).

Trigger – a rule that runs further statements by itself when a table changes (section 5.8).

Type – the kind of a value: whole number (`INTEGER`), decimal number (`REAL`), text (`TEXT`), binary data (`BLOB`) or `NULL`. In SQLite a recommendation, not a rule (section 5.6).

UTF-8 – the usual way of storing texts with accents, umlauts and other special characters. The CMM2 font does not know these characters; mmsql shows a placeholder (chapter 4).

VACUUM – rebuilds the database file and gives back the space of deleted rows (`.vacuum`, chapter 10).

View – a saved query that can be used like a table (section 5.8).

WAL – *Write-Ahead Logging*, a journal mode of SQLite in which changes first go into an extra file `…-wal`. The SQLite of the CMM2 cannot handle it; switch databases from the PC over first with `PRAGMA journal_mode=delete;` (section 5.12).

XMODEM – a way of transferring files over the serial cable between PC and CMM2 (chapter 2).

Thanks

To Peter Mather for SQLite on the CMM2 and for quick answers and fixes, to Geoff Graham for MMBasic, and to the community on TheBackShed.