mmsql 1.2 – Handbuch SQLite-Datenbanken auf dem 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>
```

Für meinen Sohn Konrad.

© 2026 Matthias Hollatz – MIT-Lizenz (siehe LICENSE.txt)

Getestet auf einem Colour Maximite 2 (Generation 2) mit MMBasic **8.00.00b1** und **8.00.00b2**.

Was ist mmsql?

mmsql ist das Schweizer Taschenmesser für SQLite-Datenbanken auf dem Colour Maximite 2. Ein einziges Programm, um Datenbanken anzulegen, anzusehen, abzufragen, zu ändern, zu prüfen, zu sichern, zu importieren und zu exportieren – direkt am CMM2, ohne eine Zeile BASIC zu schreiben. Es arbeitet wie `sqlite3`, das Werkzeug der SQLite-Entwickler für den PC, und bringt dazu vieles mit, was auf einem kleinen Rechner ohne Maus das Leben leichter macht: einen Editor für lange Anweisungen, Blättern in großen Tabellen, Formulare zum Bearbeiten von Zeilen, Hilfe auf Deutsch. mmsql ist selbst in MMBasic geschrieben und frei verfügbar (MIT-Lizenz).

> **Und was ist SQLite?** Die Entwickler beschreiben es auf sqlite.org so: eine Bibliothek, die eine vollständige SQL-Datenbank enthält – eigenständig, ohne Server, ohne Einrichtung und mit Transaktionen. Sie nennen SQLite die meistgenutzte Datenbank der Welt; sie steckt in jedem Smartphone und jedem Webbrowser. Ihr Motto: *„Small. Fast. Reliable. Choose any three.“* Seit der Firmware 7 steckt SQLite auch im CMM2 – und mmsql ist der Schlüssel dazu.

Vorwort

Dieses Handbuch ist für alle, die auf dem Colour Maximite 2 mit Daten arbeiten wollen: die Spielesammlung ordnen, Messwerte aufheben, die Mitglieder des Vereins verwalten oder einfach verstehen, was hinter dem Wort „Datenbank“ steckt. **SQL-Kenntnisse sind nicht nötig.** Alles, was man wissen muss, wird hier Schritt für Schritt erklärt, an einem Beispiel zum Mitmachen.

Wer schon mit Datenbanken gearbeitet hat, findet in Kapitel 12 die Unterschiede zu MySQL, MariaDB, MS SQL Server & Co., in Kapitel 13 alle technischen Grenzen und in Kapitel 14 einen ehrlichen Vergleich mit dem Vorbild `sqlite3`. Wer in MMBasic programmiert, findet in Kapitel 15 Tipps aus dem Quelltext von mmsql – zum Nachlesen und Übernehmen in eigene Programme.

Wie dieses Handbuch gedacht ist

Am besten liest man es am CMM2 und tippt mit. Die Beispiele bauen aufeinander auf: Kapitel 3 legt die Datenbank `retro.db` an, mit der alle weiteren Kapitel arbeiten.

Was einzutippen ist, steht so da:

```
SELECT name FROM computer;
```

Wie der Bildschirm danach aussieht, zeigt ein dunkler Kasten. Die erste Zeile beginnt mit dem **Prompt** – das ist die Eingabeaufforderung, hier `[retro.db]>`. Dahinter steht, was getippt wurde, darunter die Antwort von mmsql:

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

So lässt sich jederzeit vergleichen, ob alles auf dem richtigen Weg ist. mmsql antwortet auf Englisch, wie MMBasic selbst; dieses Handbuch erklärt jede Meldung auf Deutsch. Die Hilfe am CMM2 gibt es auch auf Deutsch (Kapitel 2).

Geht eine Eingabe über mehrere Zeilen, steht sie vor dem Kasten noch einmal als einfacher Block – zum Kopieren und Einfügen ins Terminal.

Kästen mit der Überschrift **Neu in 1.2** markieren, was es in mmsql 1.1 noch nicht gab.

Inhalt

- Was ist mmsql?
- Neu in 1.2 – die Höhepunkte

1. Was ist eine Datenbank?
2. Installation und Start
3. Rundgang in zehn Minuten
4. Die Eingabe: Editor, Tasten und Blöcke
5. SQL zum Mitmachen
6. Ansehen ohne SQL
7. Bearbeiten ohne SQL
8. Ausgabe und Export
9. Daten rein und raus: Import, Dump, Skripte
10. Sicherheit und Pflege
11. Einstellungen
12. Für Kenner von MySQL, MariaDB, MS SQL Server & Co.
13. Technische Daten und Grenzen
14. mmsql und sqlite3 – ein ehrlicher Vergleich
15. Unter der Haube – mmsql als Fundgrube für eigene Programme
16. Meldungen und Problemlösung

- Anhang A: Alle Befehle auf einen Blick
- Anhang B: Tasten
- Anhang C: Optionen
- Anhang D: SQL-Spickzettel

- Anhang E: Änderungen gegenüber 1.1
- Anhang F: Kleines Lexikon

Neu in 1.2 – die Höhepunkte

Ein richtiger Editor am Prompt. Bisher musste jede Anweisung in eine Zeile mit höchstens 255 Zeichen passen. Jetzt geht SQL über bis zu 20 Zeilen, so übersichtlich wie gewünscht: F3 beginnt eine neue Zeile, F2 führt aus. Eine Statuszeile zeigt, wo der Cursor steht. Was am PC geschrieben wurde, lässt sich einfach ins Terminal einfügen – mmsql erkennt das und behält die Zeilen zusammen. → Kapitel 4

Die Grenze von 20 Spalten ist weg. `.import` liest CSV-Dateien mit beliebig vielen Spalten, `.insert` füllt Tabellen mit bis zu 64 Spalten. Felder dürfen länger als 255 Zeichen sein. → Kapitel 7 und 9

Lange Abfragen zeigen ihr Ergebnis. Ein SELECT mit mehr als 255 Zeichen läuft wie ein kurzes: als Tabelle zum Blättern oder in jedem Ausgabemodus – am Prompt, im Block und in Skripten. → Kapitel 5

Skripte wie bei sqlite3. `.read` versteht Anweisungen über mehrere Zeilen, zeigt die Ergebnisse von Abfragen und kennt Punkt-Befehle wie `.print`, `.mode` oder `.once`. Damit laufen auch Dumps und Skripte vom PC. → Kapitel 9

Eine Regel für die Eingabe. Eine Zeile läuft mit Enter. Mehrere Zeilen sind ein Block: jede Anweisung endet mit `;`, Punkt-Befehle laufen mit, der erste Fehler stoppt den Rest. → Kapitel 4

Der Verlauf merkt sich alles. Auch lange Anweisungen über viele Zeilen kommen mit ↑ oder Bild↑ wieder – auch nach einem Neustart. → Kapitel 4

Sofort loslegen ohne Datei. Ohne offene Datenbank rechnet mmsql in einer Datenbank im Speicher (`:memory:`). → Kapitel 9

Neue Ausgaben: `.mode markdown` für Tabellen in Texten und Foren, `.mode json` für andere Programme, `.width` für feste Spaltenbreiten, `.print` für Überschriften. → Kapitel 8

`.dump` erkennt BLOB und Text sicher – nach dem Zurücklesen hat jeder Wert wieder genau seinen Typ. → Kapitel 9

Weitere neue Befehle: `.restore` (Sicherung zurückspielen), `.version`, Suchmuster bei `.tables`, `.schema`, `.indexes` und `.triggers`, `.set screen` (Bildschirm-Modus beim Start) und `.set width 0` (Breite folgt dem Bildschirm). → Kapitel 6, 10 und 11

> **Achtung, eine Änderung, die alte Skripte betrifft:** `.read` braucht jetzt am Ende jeder Anweisung ein `;`, wie `sqlite3`. Skripte mit einer Anweisung pro Zeile **ohne** `;` laufen nicht mehr richtig. Ein `;` am Zeilenende ergänzen genügt. Dumps von mmsql haben es schon immer. Einzelheiten in Kapitel 9.

1. Was ist eine Datenbank?

Tabellen, Zeilen, Spalten

Eine **Datenbank** ist eine Datei, in der Daten ordentlich in **Tabellen** liegen. Eine Tabelle sieht aus wie ein Blatt in einer Tabellenkalkulation. Das Firmware-Paket des CMM2 bringt eine kleine Beispiel-Datenbank mit, `computers.db`. Sie enthält die Tabelle `computer` mit 18 Heimcomputern; hier ein Ausschnitt:

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

- Jede **Zeile** ist ein Datensatz – hier ein Computer.
- Jede **Spalte** hat einen Namen und einen **Typ**: `name` enthält Text, `year` eine ganze Zahl.
- Die Spalte `id` ist der **Primärschlüssel** (englisch *primary key*): eine Nummer, die jede Zeile eindeutig kennzeichnet. Zwei Computer können gleich heißen, aber nie dieselbe `id` haben.
- Ein leeres Feld heißt **NULL** – „kein Wert“. NULL ist nicht dasselbe wie 0 oder ein leerer Text: Bei einem Computer mit unbekanntem Baujahr ist `year` NULL, nicht 0.

Eine Datenbank kann viele Tabellen enthalten, die zusammenhängen. In Kapitel 5 kommt neben `computer` eine Tabelle `hersteller` dazu, und beide werden verbunden beide – das ist die eigentliche Stärke einer Datenbank.

Was kann eine Datenbank, was eine Textdatei nicht kann?

Natürlich könnte man die Computerliste auch in eine Textdatei schreiben. Eine Datenbank kann aber mehr:

- **Fragen beantworten:** „Welche Computer mit Z80 kamen vor 1983 heraus?“ – eine Zeile SQL, und die Antwort ist da, sortiert nach Wunsch.
- **Rechnen:** zählen, summieren, Durchschnitt, Minimum, Maximum, gruppiert nach Hersteller oder Jahr.
- **Verbinden:** Daten aus mehreren Tabellen zusammenführen, ohne sie doppelt zu speichern.
- **Aufpassen:** Pflichtfelder, eindeutige Schlüssel, Regeln, die beim Speichern geprüft werden.
- **Sicher ändern:** Mehrere Änderungen gelten entweder alle oder gar nicht (*Transaktion*). Fällt mittendrin der Strom aus, bleibt die Datenbank heil.

SQL – die Sprache der Datenbank

Mit der Datenbank spricht man in **SQL** (*Structured Query Language*). SQL liest sich fast wie Englisch:

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

heißt: „Wähle Name und Jahr aus der Tabelle computer, nur die vor 1980, sortiert nach Jahr.“ Groß- und Kleinschreibung ist bei den SQL-Wörtern egal; dieses Handbuch schreibt sie groß, damit man sie erkennt. Kapitel 5 erklärt SQL Schritt für Schritt.

SQLite und mmsql

Auf dem CMM2 steckt in der Firmware **SQLite**, die meistgenutzte Datenbank der Welt – sie läuft auch in jedem Smartphone und jedem Webbrowser. Eine SQLite-Datenbank ist **eine einzige Datei**. Es gibt keinen Server, keine Benutzer, keine Passwörter. Die Datei lässt sich kopieren, per XMODEM zum PC schicken und dort mit `sqlite3` oder dem *DB Browser for SQLite* öffnen – und umgekehrt.

BASIC-Programme nutzen SQLite mit Befehlen wie `SQL OPEN`, `SQL EXEC` und `SQL QUERY`. **mmsql** ist das Werkzeug dazu: ein Programm, mit dem sich Datenbanken direkt am CMM2 ansehen, abfragen, ändern, prüfen, sichern, exportieren und importieren lassen – ohne eine Zeile BASIC zu schreiben. Es arbeitet wie das Kommandozeilen-Werkzeug `sqlite3` am PC und ist selbst vollständig in MMBasic geschrieben.

In mmsql gibt es zwei Sorten von Eingaben:

- **Befehle für mmsql** beginnen mit einem **Punkt**: `.tables`, `.help`, `.quit`.
- **Alles andere ist SQL** und geht an die Datenbank: `SELECT * FROM computer;`.

mmsql läuft gleichzeitig auf dem Bildschirm des CMM2 und auf einem seriellen Terminal am PC (z. B. Tera Term). Beide zeigen dasselbe, an beiden kann man tippen.

2. Installation und Start

Was nötig ist

Rechner	Colour Maximite 2, Generation 1, 2 oder 3
Firmware	MMBasic **8.00.00b1** oder neuer
SQLite-Modul	`A:/sql/sql.mod` aus dem Firmware-Paket
Beispiel-Datenbank	`A:/sql/computers.db` aus dem Firmware-Paket
für Firmware 7.00.00RC6	stattdessen mmsql **1.0.1**

Startet mmsql 1.2 auf einer älteren Firmware, meldet es sich mit einem Hinweis auf mmsql 1.0.1 und beendet sich.

Die Dateien

Datei	Zweck
`mmsql.bas`	das Programm
`mmsql_en.hlp`	Hilfetexte, Englisch
`mmsql_de.hlp`	Hilfetexte, Deutsch
`LICENSE.txt`	MIT-Lizenz
`demo.sql`	Beispieldaten für das Tutorial
PDF-Dateien	dieses Handbuch, das englische Manual, README und Tutorial

Alle Dateien kommen in denselben Ordner. Am einfachsten ist der Ordner `A:/sql`, in dem schon `sql.mod` und `computers.db` liegen – dann ist die Beispiel-Datenbank sofort zur Hand. mmsql legt in seinem Ordner zwei weitere Dateien an:

Datei	Inhalt
`mmsql.cfg`	die Einstellungen (`.set`), als lesbare Textdatei
`mmsql.his`	der Verlauf der Eingaben

Beide dürfen jederzeit gelöscht werden; mmsql beginnt dann mit den Standardwerten bzw. einem leeren Verlauf.

Übertragen auf den CMM2

Entweder die SD-Karte in den PC stecken und die Dateien kopieren, oder sie über das serielle Kabel mit XMODEM schicken. In Tera Term: *Datei → Übertragung → XMODEM → Senden*, am CMM2 vorher:

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

Das für jede Datei wiederholen.

Das Terminal einrichten

mmsql läuft auch ohne PC, nur mit Bildschirm und USB-Tastatur. Mit einem Terminal am PC geht vieles bequemer, vor allem das Einfügen von SQL aus einem Texteditor. Für Tera Term empfehlen sich:

Einstellung in Tera Term	Wert	wofür
Setup → Serial port → Speed	wie am CMM2 (meist 115200)	Verbindung
Setup → Additional settings → Copy and Paste → Paste delay per line	z. B. 200 ms	Einfügen mehrerer Zeilen (Kapitel 4)
Setup → Additional settings → Copy and Paste → Trim trailing new line	an	kein leeres Enter am Ende eingefügter Blöcke

Beim Start stellt mmsql das Terminalfenster auf dieselbe Größe wie den Bildschirm (z. B. 50 Zeilen × 100 Spalten). So zeigen beide genau dasselbe.

Starten

Am MMBasic-Prompt:

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

mmsql meldet sich so:

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

Die erste Zeile sieht aus, als hätte man am Prompt `.version` getippt: Version, Generation des CMM2 (G1, G2 oder G3) und Lizenz. Darunter wartet der Prompt `mmsql>` auf eine Eingabe. Er heißt so, solange keine Datenbank offen ist.

Man kann gleich eine Datenbank mitgeben:

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

Die Hilfe auf Deutsch

mmsql hat eine eingebaute Hilfe. Damit sie deutsch spricht, einmal eingeben:

```
.set helpfile mmsql_de.hlp
```

Die Einstellung wird in `mmsql.cfg` gespeichert und gilt ab sofort, auch nach jedem Neustart.

> **Tipp: Hilfe in einer anderen Sprache.** Die Hilfedateien sind einfache Textdateien. Wer die Hilfe auf Französisch, Niederländisch oder in einer anderen Sprache möchte, lässt `mmsql_en.hlp` am PC von einer KI oder mit DeepL (deepl.com) übersetzen und speichert das Ergebnis neben mmsql, z. B. als `mmsql_fr.hlp`. Aktivieren mit `.set helpfile mmsql_fr.hlp`. Beim Übersetzen drei Dinge beachten:
>
> - **Zeilen, die mit `#` beginnen, bleiben unverändert.** Sie enthalten die Namen, nach denen `.help` sucht (`#open o`, `#keys`).
> - **Befehle, Optionen und SQL bleiben, wie sie sind** (`.open`, `.set paste`, `SELECT`).
> - **Nur einfache Buchstaben verwenden.** Die Schrift des CMM2 hat keine Umlaute und Akzente; die deutsche Hilfe schreibt deshalb „ae“, „oe“, „ue“ und „ss“. Zeilen sollten höchstens 79 Zeichen lang sein, damit sie auf jeden Bildschirm passen.

Eingabe	zeigt
`.help`	die Übersicht aller Befehle nach Gruppen
`.help befehl`	Einzelheiten zu einem Befehl, z. B. `.help import`
`.help keys`	die Tasten des Editors
`.help sql`	Hinweise zu SQL in mmsql
`.help memory`	die Datenbank im Speicher
`.help utf8`	Umlaute und andere Sonderzeichen

```
[retro.db]> .help
.help name zeigt Einzelheiten, z.B. .help import  (auch: .help keys/sql/utf8)

Datenbank : .open .close .info .backup .restore .vacuum .files .kill
Ansehen   : .tables .indexes .triggers .schema .columns .browse .databases
Daten     : .insert .edit .delete .import .read
Struktur  : .create .alter .reindex
Pruefen   : .check .fkcheck .fk
Ausgabe   : .mode .headers .separator .nullvalue .width .once .output .dump
Optionen  : .set .timer .bail .echo .print .cls .version .help .quit
Kurzformen: .o .t .ix .s .c .b .db .ver .h .q

Befehle beginnen mit einem Punkt. Alles andere ist SQL. Enter fuehrt eine
Zeile aus; F3 beginnt eine neue Zeile fuer eine mehrzeilige Anweisung, F2
fuehrt sie aus (.help keys).
```

Beenden

`.quit` (kurz `.q`, oder `.exit`) schließt die Datenbank und beendet mmsql. Danach geht es am MMBasic-Prompt weiter.

3. Rundgang in zehn Minuten

In diesem Kapitel wird die Beispiel-Datenbank geöffnet, bekommt ihre ersten Fragen gestellt, und es entsteht eine eigene Kopie zum Üben. Dazu mmsql im Ordner `A:/sql` starten (Kapitel 2).

Schritt 1: Datenbank öffnen und hineinschauen

`.open` öffnet eine Datenbank, `.tables` zeigt, welche Tabellen darin sind:

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

Der Prompt heißt jetzt `[computers.db]>` – so ist immer zu sehen, mit welcher Datenbank gerade gearbeitet wird. Es gibt eine Tabelle `computer` (`type` = table) mit 6 Spalten (`ncol`). Die letzte Zeile `1 row(s)` sagt, wie viele Zeilen die Antwort hatte – hier eine.

Schritt 2: Die erste Abfrage

Jetzt die erste SQL-Anweisung. `SELECT * FROM computer;` heißt: „Zeige alle Spalten (`*`) aus der Tabelle 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 richtet die Spalten aus, schreibt die Spaltennamen darüber und zählt am Ende die Zeilen. Wäre die Tabelle länger als eine Bildschirmseite, ginge es mit Enter weiter (Kapitel 6).

Schritt 3: Eine Frage stellen

Nur die Computer vor 1980, und nur Name und Jahr:

```
[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)
```

Das ist der Kern von SQL: Man beschreibt, **was** man wissen will – nicht, wie der Rechner es suchen soll. `WHERE` filtert die Zeilen, die Liste hinter `SELECT` wählt die Spalten.

Schritt 4: Eine eigene Übungs-Datenbank

`computers.db` gehört zum Firmware-Paket und soll unverändert bleiben. Für alles Weitere dient eine Kopie:

```
[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` schreibt eine saubere Kopie der offenen Datenbank in eine neue Datei. `.open` schließt die alte Datenbank („Closed“) und öffnet die neue. Ab jetzt arbeiten alle Beispiele in diesem Handbuch mit **`retro.db`**. Um einmal neu anzufangen: `.quit`, am MMBasic-Prompt `KILL "retro.db"`, mmsql wieder starten und Schritt 4 wiederholen.

Schritt 5: Punkt vergessen? Kein Problem

Befehle für mmsql beginnen mit einem Punkt. Fehlt er, hilft mmsql weiter:

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

Das gilt auch für ein versehentlich getipptes Komma statt des Punktes (`,help`).

Schritt 6: Beenden

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

Beim nächsten Start `RUN "mmsql", "retro.db"` eingeben oder die Datei mit `.open retro.db` öffnen.

Das war der Rundgang: eine Datenbank geöffnet, ihre Tabellen angesehen, zwei Fragen gestellt und eine Sicherungskopie angelegt. Das nächste Kapitel zeigt, wie man bequem längere Anweisungen eingibt; danach geht es mit SQL richtig los.

4. Die Eingabe: Editor, Tasten und Blöcke

Alles, was am Prompt getippt wird, landet in einem kleinen Editor. Er arbeitet am Bildschirm und am Terminal gleich.

Eine Zeile: tippen und Enter

Für die meisten Eingaben reicht eine Zeile. Enter führt sie aus:

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

Ein `;` am Ende ist erlaubt, aber nicht nötig. Beim Tippen helfen diese Tasten:

Taste	Wirkung
← →	Cursor bewegen
Pos1 / Ende	an den Anfang / das Ende der Zeile
Backspace	Zeichen links vom Cursor löschen
Entf	Zeichen am Cursor löschen
Esc	Eingabe verwerfen
↑ ↓	frühere Eingaben aus dem Verlauf holen
Enter	ausführen

Ist die Zeile länger als der Bildschirm breit, scrollt sie seitlich. Eine Zeile hat höchstens **255 Zeichen** – so lang darf ein Text in MMBasic sein.

Mehrere Anweisungen in einer Zeile werden mit `;` getrennt. Sie laufen nacheinander, jede mit ihrem eigenen Ergebnis:

```
[retro.db]> SELECT count(*) FROM hersteller; SELECT count(*) FROM computer;
Error: SQL: no such table: hersteller
  Statement ( 31 chars): SELECT count(*) FROM hersteller
Hint: .tables lists the tables
Not run: 1 more statement(s) of this line.
```

Gibt eine davon einen Fehler, laufen die übrigen nicht mehr: „Not run: n more statement(s) of this line.“

Mehrere Zeilen: F3 und F2

> **Neu in 1.2:** Anweisungen über mehrere Zeilen.

Längere Abfragen werden übersichtlicher, wenn man sie auf mehrere Zeilen verteilt. Dafür gibt es zwei Tasten:

Taste	Wirkung
F3	neue Zeile beginnen (oder die Zeile am Cursor teilen)
F2	die ganze Anweisung ausführen

Zum Ausprobieren `SELECT name, year, cpu` tippen und **F3** drücken. Sofort ändert sich die Anzeige: Die Prompt-Zeile wird zur **Statuszeile**, und die Zeilen erscheinen nummeriert darunter. Weiter tippen, nach jeder Zeile F3:

```
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;
```

Die Statuszeile zeigt:

- den Prompt `[retro.db]>`, damit klar ist, in welcher Datenbank es weitergeht,
- `[ Ln  4/ 4  Col  17/255 ]` – der Cursor steht in Zeile 4 von 4, Spalte 17 von höchstens 255,
- `[ F2 = run  Esc = discard ]` – zur Erinnerung: F2 führt aus, Esc verwirft.

Die Unterstriche füllen die Zeile bis zum rechten Rand. Jetzt **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)
```

Sobald mehrere Zeilen da sind, beginnt **Enter** eine neue Zeile – wie in einem Texteditor. Ausgeführt wird nur mit F2. So kann nichts aus Versehen loslaufen, während noch geschrieben wird.

Tipp: Auf einem leeren Prompt schaltet F3 gleich in den mehrzeiligen Modus, ohne eine leere Zeile 1 zu erzeugen. Wer also weiß, dass es länger wird, drückt zuerst F3.

Bewegen und Ändern in mehreren Zeilen

Taste	Wirkung
← →	Cursor bewegen, auch über Zeilenenden hinweg
↑ ↓	eine Bildschirmzeile hoch / runter; in der ersten bzw. letzten: Verlauf
Pos1 / Ende	Anfang / Ende der Bildschirmzeile
Backspace am Zeilenanfang	die Zeile an die vorige anhängen
Entf am Zeilenende	die nächste Zeile heranholen
Enter oder F3	neue Zeile, bzw. die Zeile am Cursor teilen
F2	ausführen
Esc	alles verwerfen („(discarded)“)

Jede Zeile darf bis zu 255 Zeichen lang sein. Passt sie nicht in die Bildschirmbreite, läuft sie in der nächsten Bildschirmzeile weiter, markiert mit `+`. Beim Tippen fließt der Text um wie in einer Textverarbeitung:

```
[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);
```

Bis zu **20 Zeilen** sind möglich. Die Anweisung wird mit Zeilenumbrüchen genau so gespeichert, wie sie getippt wurde – `.schema` zeigt ein `CREATE TABLE` später in derselben Form.

Lieber immer mit F2? Mit `.set enter line` beginnt Enter **immer** eine neue Zeile, auch bei der ersten; ausgeführt wird nur mit F2. `.set enter run` (Standard) stellt es zurück.

Der Verlauf

Jede ausgeführte Eingabe kommt in den **Verlauf**, auch lange über viele Zeilen. Er wird in `mmsql.his` gespeichert und ist nach einem Neustart wieder da.

Taste	Wirkung
↑ / ↓	vorige / nächste Eingabe (in mehreren Zeilen: erst am oberen bzw. unteren Rand)
Bild↑ / Bild↓	vorige / nächste Eingabe, egal wo der Cursor steht

Was gerade angefangen und noch nicht ausgeführt ist, geht dabei nicht verloren: Ganz nach unten geblättert, ist es wieder da. Die Option `history` (Standard 50) legt fest, wie viele Einträge mmsql behält.

Einfügen vom PC

Am PC lässt sich SQL bequem im Texteditor schreiben und dann ins Terminal einfügen. Das Terminal schickt die Zeilen dann so, als würden sie sehr schnell getippt – mit einem Enter nach jeder Zeile. Damit nicht jede Zeile einzeln losläuft, passt mmsql auf: **Kommen nach einem Enter sofort weitere Zeichen, ist das eingefügter Text.** mmsql behält die Zeilen dann im Editor, F2 führt alles aus.

Wie lange „sofort“ ist, stellt `.set paste` ein (Standard 30 Millisekunden). Manche Terminals warten nach jeder eingefügten Zeile; in Tera Term heißt das *Paste delay per line*. Dann muss die Paste-Zeit von mmsql länger sein als diese Pause:

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

Der Preis: Ein Enter auf einer einzelnen Zeile führt erst nach dieser Zeit aus – bei 250 ms merkt man das kaum. Wer ganz sicher gehen will, drückt vor dem Einfügen F3. Fehlen beim Einfügen Zeichen, hilft in Tera Term eine kleine Verzögerung pro Zeichen (*Setup → Serial port → Transmit delay*).

Hier wurden die Tabelle `hersteller` und ihre Daten aus einem PC-Editor eingefügt und mit F2 ausgeführt (diese Tabelle wird ab Kapitel 5 gebraucht – am besten genau so einfügen):

```
CREATE TABLE hersteller(name TEXT PRIMARY KEY, land TEXT);
INSERT INTO hersteller 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 hersteller(name TEXT PRIMARY KEY, land TEXT);
 2: INSERT INTO hersteller 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
```

Die Regel für Blöcke

> **Neu in 1.2:** eine einzige Regel für alles, was mehr als eine Zeile hat.

Sobald eine Eingabe mehrere Zeilen hat – getippt mit F3 oder eingefügt –, ist sie ein **Block**. Ein Block läuft so, als wären seine Zeilen einzeln nacheinander eingegeben worden:

1. **Jede SQL-Anweisung endet mit `;` am Zeilenende.** Erst dann weiß mmsql, dass die Anweisung fertig ist. Bei der letzten darf es fehlen.
2. **Punkt-Befehle laufen mit**, wenn sie zwischen zwei Anweisungen in einer eigenen Zeile stehen.
3. **Die Ausgabe ist wie am Prompt.**
4. **Der erste SQL-Fehler stoppt den Rest:** „Not run: the rest of the block.“

Ein Block mit Punkt-Befehlen:

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

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

Ein Block mit einem Tippfehler in Zeile 2 – Zeile 3 läuft nicht mehr:

```
UPDATE hersteller SET land = '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 hersteller SET land = '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.
```

Warum das `;` so wichtig ist: Ohne `;` gehört die nächste Zeile noch zur selben Anweisung. Das ist gewollt, denn nur so kann eine Anweisung über mehrere Zeilen gehen. Fehlt es, hängt mmsql zwei Anweisungen zusammen, und SQLite meldet einen Syntaxfehler. Auch eine Zeile mit Punkt gehört dann noch zur Anweisung, genau wie bei `sqlite3`.

Kurz gesagt:

Eingabe	Regel
eine Zeile	läuft mit Enter, `;` nicht nötig
mehrere Zeilen (Block)	jede Anweisung endet mit `;` am Zeilenende
Skript mit `.read`	wie ein Block (Kapitel 9)

Blättern in Ergebnissen

Längere Ergebnisse zeigt mmsql seitenweise. Am Seitenende:

Taste	Wirkung
Enter oder N	nächste Seite
B	vorige Seite
Q oder Esc	aufhören

Wie viele Zeilen auf eine Seite kommen, stellt `.set rows` ein (Standard 20).

Ctrl-C

Läuft eine SQL-Anweisung zu lange, bricht **Ctrl-C** sie ab, und mmsql läuft weiter. Was dabei mit den Daten passiert, erklärt Kapitel 10.

Umlaute

SQLite speichert Text als UTF-8, mit allen Umlauten und Sonderzeichen. Die Schrift des CMM2 kennt aber keine Umlaute. Deshalb zeigt mmsql jedes solche Zeichen als **einen** Platzhalter, einen mittigen Punkt `·` (im Terminal erscheint er als `ú`). So bleiben die Spalten gerade. In der Datenbank und in Dateien bleibt der echte Text erhalten.

Einstellung	Wirkung
`.set utf8 dot`	Platzhalter (Standard)
`.set utf8 raw`	Bytes unverändert senden, wenn das Terminal auf UTF-8 steht
`.set dotchar 46`	anderer Platzhalter, hier der Punkt `.`

Eintippen lassen sich Umlaute in mmsql nicht. Wer sie in SQL braucht, schreibt ihre Codes: `'M' || char(252) || 'ller'` ergibt „Müller“.

5. SQL zum Mitmachen

Jetzt geht es richtig los: `retro.db` öffnen (Kapitel 3) und mittippen. Wer in Kapitel 4 die Tabelle `hersteller` noch nicht angelegt hat, holt das jetzt nach (Abschnitt „Einfügen vom PC“) – ab Abschnitt 5.4 wird sie gebraucht.

Ein paar Grundregeln vorweg:

- SQL-Wörter wie `SELECT` oder `WHERE` dürfen groß oder klein geschrieben werden.
- **Texte** stehen in einfachen Anführungszeichen: `'Apple'`. Bei Texten zählt Groß- und Kleinschreibung: `'apple'` findet nichts.
- **Zahlen** stehen ohne Anführungszeichen: `1982`.
- Namen von Tabellen und Spalten schreibt man einfach hin: `computer`, `year`.

5.1 Fragen stellen: SELECT

Die Grundform jeder Abfrage:

```
SELECT spalten FROM tabelle WHERE bedingung ORDER BY sortierung;
```

Nur `SELECT … FROM …` ist Pflicht, der Rest kommt nach Bedarf dazu.

Spalten wählen und filtern: Name, Prozessor und Speicher der Sinclair-Computer:

```
[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 *` zeigt alle Spalten, eine Liste wie `name, cpu, ramkb` nur diese, in dieser Reihenfolge.

Sortieren und begrenzen: `ORDER BY` sortiert, `DESC` dreht die Reihenfolge um (größte zuerst), ohne Angabe wird aufsteigend sortiert. Hinter dem Komma steht, wonach bei Gleichstand sortiert wird. `LIMIT 5` zeigt nur die ersten fünf:

```
[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)
```

Die fünf jüngsten Computer – drei davon kamen 1984 heraus und stehen nach Namen sortiert.

Bedingungen verknüpfen: Mit `AND` müssen beide Bedingungen zutreffen, mit `OR` genügt eine. `NOT` dreht eine Bedingung um.

```
[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)
```

Die Vergleiche im Überblick:

Vergleich	Bedeutung	Beispiel
`=`	gleich	`maker = 'Apple'`
`<>` oder `!=`	ungleich	`cpu <> 'Z80'`
`<`, `<=`, `>`, `>=`	kleiner, größer	`year >= 1984`
`BETWEEN a AND b`	von a bis b (beide eingeschlossen)	`year BETWEEN 1981 AND 1982`
`IN (…)`	einer aus der Liste	`maker IN ('Acorn', 'Oric')`
`LIKE`	Textmuster	`name LIKE 'ZX%'`
`IS NULL`, `IS NOT NULL`	kein Wert / ein Wert	`year IS NULL`

Textmuster mit LIKE: `%` steht für beliebig viele Zeichen (auch keins), `_` für genau ein Zeichen. `LIKE` unterscheidet bei englischen Buchstaben nicht zwischen groß und klein.

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

Eine Liste mit IN. Längere Abfragen schreibt man am besten über mehrere Zeilen (F3 nach jeder Zeile, F2 am Ende – Kapitel 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)
```

Ein Bereich mit 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)
```

Jeden Wert nur einmal mit DISTINCT: Welche Prozessoren gibt es überhaupt?

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

> **Merke: NULL ist anders.** Ein Feld ohne Wert ist NULL. Mit NULL kann man nicht vergleichen: `year = NULL` ist **nie** wahr, auch nicht bei leeren Feldern. Richtig heißt es `year IS NULL`. In mmsql erscheint NULL standardmäßig als leeres Feld; mit `.nullvalue NULL` steht dort sichtbar `NULL` (Abschnitt 5.4).

5.2 Rechnen und umformen

In der Spaltenliste darf auch gerechnet werden. `AS` gibt dem Ergebnis einen Namen:

```
[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 kennt viele Funktionen für Texte, zum Beispiel `upper()` (in Großbuchstaben), `length()` (Länge) und `substr(text, ab, länge)` (ein Stück):

```
[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)
```

Mit `CASE` lassen sich Werte einteilen – wie ein `SELECT CASE` in BASIC:

```
SELECT name, ramkb,
  CASE WHEN ramkb < 16  THEN 'klein'
       WHEN ramkb < 128 THEN 'mittel'
       ELSE 'gross' END AS klasse
  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 'klein'
 3:        WHEN ramkb < 128 THEN 'mittel'
 4:        ELSE 'gross' END AS klasse
 5:   FROM computer
 6:   WHERE maker = 'Commodore';
name         ramkb klasse
------------------------
PET 2001     4     klein
VIC-20       5     klein
Commodore 64 64    mittel
Amiga 500    512   gross
 4 row(s)
```

Weitere nützliche Funktionen: `lower()`, `trim()`, `replace(text, alt, neu)`, `round(zahl, stellen)`, `abs()`, `coalesce(a, b)` (das erste, das nicht NULL ist), `printf('%05d', zahl)` und `||`, das zwei Texte verbindet: `name || ' (' || maker || ')'`.

5.3 Zählen und zusammenfassen

Wie viele Computer sind in der Tabelle?

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

Aggregatfunktionen fassen viele Zeilen zu einem Wert zusammen: `count()` zählt, `min()` und `max()` finden den kleinsten und größten Wert, `avg()` bildet den Durchschnitt, `sum()` die Summe.

```
[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)
```

Gruppieren mit GROUP BY: Wie viele Computer hat jeder Hersteller? `GROUP BY maker` bildet für jeden Hersteller eine Gruppe, `count(*)` zählt in jeder Gruppe:

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

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

Das ist einer der Momente, in denen man merkt, was eine Datenbank kann: Eine Frage, die mit Bleistift und Liste eine Weile dauern würde, beantwortet eine Zeile SQL sofort.

Gruppen filtern mit HAVING: `WHERE` filtert Zeilen **vor** dem Gruppieren, `HAVING` filtert Gruppen **danach**. Welche Prozessoren stecken in mindestens drei Computern?

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

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

Die Reihenfolge der Teile ist immer gleich: `SELECT … FROM … WHERE … GROUP BY … HAVING … ORDER BY … LIMIT …`

5.4 Zwei Tabellen verbinden: JOIN

Bisher stand alles in einer Tabelle. In Kapitel 4 entstand die Tabelle `hersteller` – zu jedem Hersteller sein Land:

```
[retro.db]> SELECT * FROM hersteller;
name              land
---------------------
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)
```

Warum steht das Land nicht einfach als weitere Spalte in `computer`? Weil Commodore dann viermal „USA“ hätte. Ändert sich etwas, müsste man es an vier Stellen ändern, und irgendwann steht es an einer Stelle falsch. In einer Datenbank speichert man jede Angabe **einmal** und **verbindet** die Tabellen bei der Abfrage.

JOIN verbindet zwei Tabellen über eine Bedingung. Hier gehört eine Zeile aus `hersteller` zu einem Computer, wenn ihr `name` gleich dem `maker` des Computers ist:

```
SELECT computer.name, computer.maker, hersteller.land
  FROM computer
  JOIN hersteller ON hersteller.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, hersteller.land
 2:   FROM computer
 3:   JOIN hersteller ON hersteller.name = computer.maker
 4:   WHERE year < 1982
 5:   ORDER BY year, computer.name;
name              maker     land
------------------------------
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)
```

So liest sich das:

- `FROM computer JOIN hersteller` – nimm beide Tabellen,
- `ON hersteller.name = computer.maker` – und füge die Zeilen zusammen, die zueinander passen,
- `computer.name` – weil beide Tabellen eine Spalte `name` haben, sagt der Tabellenname davor, welche gemeint ist.

Kürzer mit Aliasnamen: Hinter dem Tabellennamen darf ein kurzer Ersatzname stehen, hier `c` und `h`. Das spart Tipparbeit:

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

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

Wer fehlt? LEFT JOIN. Ein normaler JOIN zeigt nur Zeilen, die in **beiden** Tabellen einen Partner haben. Der Hersteller „Oric“ fehlt in `hersteller` – mit `JOIN` würde der Oric-1 einfach verschwinden. `LEFT JOIN` behält alle Zeilen der linken Tabelle (der vor `LEFT JOIN`) und setzt NULL ein, wo der Partner fehlt. Damit NULL sichtbar wird, vorher `.nullvalue NULL` einstellen:

```
SELECT c.name, c.maker, h.land
  FROM computer c
  LEFT JOIN hersteller 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.land
 2:   FROM computer c
 3:   LEFT JOIN hersteller h ON h.name = c.maker
 4:   WHERE c.maker IN ('IBM', 'Oric')
 5:   ORDER BY c.name;
name   maker land
----------------
IBM PC IBM   USA
Oric-1 Oric  NULL
 2 row(s)
```

Andersherum gefragt: Welche Hersteller haben die wenigsten Computer – auch keinen? `count(c.id)` zählt nur Zeilen, in denen es einen Computer gibt:

```
SELECT h.name, count(c.id) AS computer
  FROM hersteller 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 hersteller 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 steht in `hersteller`, hat aber keinen Computer in der Liste – ein klarer Fall für einen TI-99/4A.

Verbinden und gruppieren: Wie viele Computer kommen aus welchem Land, und seit wann?

```
SELECT h.land, count(*) AS anzahl, min(c.year) AS ab
  FROM computer c
  JOIN hersteller h ON h.name = c.maker
  GROUP BY h.land;
```

```
[retro.db]> [ Ln  4/ 4  Col  19/255 ]__[ F2 = run  Esc = discard ]_____________________________
 1: SELECT h.land, count(*) AS anzahl, min(c.year) AS ab
 2:   FROM computer c
 3:   JOIN hersteller h ON h.name = c.maker
 4:   GROUP BY h.land;
land anzahl ab
----------------
UK   7      1981
USA  10     1977
 2 row(s)
```

(Oric fehlt hier, weil ein normaler JOIN ihn nicht findet. In Kapitel 7 wird er nachgetragen.)

Eine Abfrage in der Abfrage: Welche Computer haben mehr Speicher als der Durchschnitt? Die innere Abfrage in Klammern rechnet den Durchschnitt aus, die äußere vergleicht:

```
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 Daten ändern: INSERT, UPDATE, DELETE

Eine Zeile einfügen mit `INSERT INTO tabelle(spalten) VALUES(werte);`. Die `id` bleibt weg – SQLite vergibt die nächste freie Nummer selbst. mmsql meldet, wie viele Zeilen geändert wurden, und die neue Nummer (`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)
```

Werte ändern mit `UPDATE tabelle SET spalte = wert WHERE bedingung;`. Unser ZX81 hat eine 16K-Speichererweiterung bekommen:

```
[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)
```

Zeilen löschen mit `DELETE FROM tabelle WHERE bedingung;`, zum Beispiel `DELETE FROM computer WHERE id = 19;`.

> **Vorsicht:** `UPDATE` und `DELETE` **ohne** `WHERE` betreffen **alle** Zeilen der Tabelle! Vorher mit einem `SELECT` mit derselben Bedingung prüfen, welche Zeilen getroffen werden.

Wer unsicher ist, arbeitet in einer **Transaktion** (Abschnitt 5.7): Erst `BEGIN`, dann ändern und nachsehen, dann `COMMIT` (behalten) oder `ROLLBACK` (alles zurück). Hier geht ein `DELETE` ohne `WHERE` schief – und wird gerettet:

```
[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)
```

Auf den Stern im Prompt achten: `[retro.db]*>` zeigt, dass eine Transaktion offen ist.

5.6 Eigene Tabellen anlegen: CREATE TABLE

Jetzt eine eigene Tabelle: welche Computer aus der Liste in der eigenen Sammlung stehen, in welchem Zustand und seit wann. Bei einem `CREATE TABLE` lohnt sich die Schreibweise über mehrere Zeilen besonders:

```
CREATE TABLE besitz(
  id          INTEGER PRIMARY KEY,
  computer_id INTEGER NOT NULL,
  zustand     TEXT DEFAULT 'ok',
  gekauft     INTEGER
);
```

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

Die Bausteine einer Spalte:

Angabe	Bedeutung
`INTEGER`, `REAL`, `TEXT`, `BLOB`	Typ: ganze Zahl, Kommazahl, Text, Binärdaten
`PRIMARY KEY`	Schlüssel der Tabelle; `INTEGER PRIMARY KEY` zählt automatisch hoch
`NOT NULL`	Pflichtfeld – ein Wert muss da sein
`DEFAULT wert`	Vorgabe, wenn beim Einfügen kein Wert angegeben ist (hier `'ok'`)
`UNIQUE`	jeder Wert darf nur einmal vorkommen
`REFERENCES tabelle(spalte)`	Fremdschlüssel: der Wert muss in der anderen Tabelle vorkommen (Kapitel 10)

> **Aha: SQLite ist großzügig mit Typen.** Anders als die meisten Datenbanken prüft SQLite den Typ nur locker. Passt ein Wert zur Spalte, wird er umgewandelt; sonst wird er so gespeichert, wie er kommt. Zum Ausprobieren: `typeof()` zeigt, als was ein Wert gespeichert ist:

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

Der Text `'42'` wurde in der INTEGER-Spalte zur Zahl, die Zahl `42` in der TEXT-Spalte zum Text, und `'abc'` steht als Text in der INTEGER-Spalte. (Dieses Beispiel lief in einer Datenbank im Speicher, Kapitel 9 – es verändert `retro.db` nicht.) Wer es streng will, schreibt hinter die Klammer der Spaltenliste `STRICT`; dann lehnt SQLite falsche Typen ab.

Tabellen ändern und löschen:

Anweisung	Wirkung
`ALTER TABLE t ADD COLUMN spalte TEXT;`	Spalte anhängen
`ALTER TABLE t RENAME COLUMN alt TO neu;`	Spalte umbenennen
`ALTER TABLE t DROP COLUMN spalte;`	Spalte löschen
`ALTER TABLE t RENAME TO neu;`	Tabelle umbenennen
`DROP TABLE t;`	Tabelle mit allen Daten löschen – ohne Rückfrage!

Den Typ einer Spalte ändern kann SQLite nicht direkt. Man legt eine neue Tabelle an und kopiert die Daten: `CREATE TABLE neu(…);`, `INSERT INTO neu SELECT * FROM alt;`, `DROP TABLE alt;`, `ALTER TABLE neu RENAME TO alt;`. Ohne SQL helfen `.create` und `.alter` (Kapitel 7).

5.7 Transaktionen: alles oder nichts

Eine **Transaktion** fasst mehrere Änderungen zusammen: Entweder gelten alle, oder keine. Sie beginnt mit `BEGIN` und endet mit `COMMIT` (behalten) oder `ROLLBACK` (verwerfen):

```
[retro.db]> BEGIN;
OK
[retro.db]*> INSERT INTO besitz(computer_id, zustand, gekauft) VALUES(10, 'Tastatur defekt', 1985);
OK, 1 row(s) changed, last ID 2
[retro.db]*> INSERT INTO besitz(computer_id, gekauft) VALUES(18, 1989);
OK, 1 row(s) changed, last ID 3
[retro.db]*> COMMIT;
OK
[retro.db]> SELECT * FROM besitz;
id computer_id zustand         gekauft
-------------------------------------
1  9           ok              1984
2  10          Tastatur defekt 1985
3  18          ok              1989
 3 row(s)
```

Transaktionen haben zwei Vorteile:

1. **Sicherheit:** Bricht etwas mittendrin ab – ein Fehler, Ctrl-C, der Strom –, bleibt die Datenbank im Zustand vor `BEGIN`.
2. **Tempo:** Ohne Transaktion schreibt SQLite nach **jeder** Änderung auf die SD-Karte. In einer Transaktion nur einmal, beim `COMMIT`. Hundert `INSERT` in einer Transaktion sind viel schneller als hundert einzelne.

Solange eine Transaktion offen ist, zeigt der Prompt den **Stern**. Fehlt das `COMMIT` beim Beenden, fragt mmsql nach (Kapitel 10). `SAVEPOINT name`, `RELEASE name` und `ROLLBACK TO name` setzen Zwischenstände innerhalb einer Transaktion.

5.8 Index, View und Trigger

Ein Index macht das Suchen schneller, wie das Stichwortverzeichnis eines Buches. Bei 18 Zeilen merkt man nichts, bei 18 000 sehr viel. Sinnvoll ist ein Index auf Spalten, nach denen oft gesucht oder verbunden wird:

```
[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_herste~ hersteller name
 3 row(s)   ~ = shortened (.mode line shows all)
```

`computer_year` war schon in `computers.db`. Der Index mit dem langen Namen `sqlite_autoindex_hersteller_1` (gekürzt angezeigt, `~`) hat SQLite selbst angelegt, weil `name` der Primärschlüssel von `hersteller` ist. SQLite benutzt einen Index von allein, wenn er hilft; in der Abfrage selbst ändert sich nichts.

Eine View ist eine gespeicherte Abfrage, die sich wie eine Tabelle benutzen lässt:

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

Die View speichert keine Daten, nur die Frage. Ändern sich `computer` oder `hersteller`, zeigt sie sofort den neuen Stand.

Ein Trigger ist eine Regel, die bei einer Änderung von selbst etwas tut. Hier merkt sich die Datenbank jede Änderung am Speicher eines Computers. Ein Trigger enthält Anweisungen mit `;` im Inneren und endet mit `END;` – am besten mit F3 über mehrere Zeilen eingeben:

```
CREATE TABLE ram_log(computer_id INTEGER, alt INTEGER, neu 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, alt INTEGER, neu 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 alt neu
------------------
8           5   32
 1 row(s)
```

`old` und `new` sind im Trigger die Zeile vor und nach der Änderung. Der Trigger lief beim `UPDATE` von allein. `DROP TRIGGER ram_changed;` entfernt ihn wieder.

5.9 Lange Anweisungen

> **Neu in 1.2:** Anweisungen über 255 Zeichen, auch Abfragen mit Ergebnis.

Im Editor hat jede Zeile höchstens 255 Zeichen, eine Anweisung aber bis zu 20 Zeilen. Wird sie zusammen länger als 255 Zeichen, gibt mmsql sie als Ganzes an SQLite – Abfragen zeigen ihr Ergebnis wie gewohnt, zum Blättern und in jedem Ausgabemodus. Das gilt genauso für eingefügte Blöcke und für Skripte mit `.read`. Die Obergrenze für eine Anweisung ist die Option `maxtext` (Standard 32 768 Zeichen, Kapitel 11).

5.10 Datum und Uhrzeit

SQLite rechnet mit Datum und Uhrzeit als Text im Format `JJJJ-MM-TT HH:MM:SS`:

Ausdruck	Ergebnis
`date('now')`	das heutige Datum aus der Uhr des CMM2
`datetime('now')`	Datum und Uhrzeit
`date('2026-11-20', '+100 days')`	2027-02-28
`strftime('%Y', '2026-11-20')`	2026
`julianday('2026-12-24') - julianday('now')`	Tage bis Heiligabend

Die Zeit kommt von der Uhr des CMM2 (`TIME$` und `DATE$` in MMBasic). Zeitzonen kennt das SQLite-Modul nicht: Die Angaben `localtime` und `utc` fehlen.

5.11 Wenn etwas schiefgeht

Bei einem Fehler zeigt mmsql die Meldung von SQLite, die Anweisung und ihre Länge – und oft einen Hinweis:

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

Die Tabelle heißt `computer`, nicht `computers` – `.tables` hätte es gezeigt. Den häufigsten Fehler fängt mmsql schon vorher ab: einen Befehl ohne Punkt.

```
[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.
```

Steht ein solches Wort am Anfang einer längeren Eingabe, versucht mmsql sie als SQL. Meldet SQLite den Syntaxfehler genau beim ersten Wort (`near "tables": syntax error`), erinnert mmsql zusätzlich: „Hint: commands start with a dot“. Weitere Meldungen erklärt Kapitel 16.

5.12 Was das SQLite-Modul nicht kann

Das SQLite des CMM2 (Version 3.47.0) ist für den Speicher des Rechners verkleinert. Es fehlen:

fehlt	Folge
Mathe-Funktionen	kein `sqrt()`, `sin()`, `power()`, `ln()` usw. – die Grundrechenarten und `round()`, `abs()` gibt es
`EXPLAIN`, `ANALYZE`	keine Abfragepläne und Statistiken
WAL-Modus	Datenbanken vom PC im WAL-Modus lassen sich nicht öffnen (Kapitel 16)
Erweiterungen	kein `load_extension()`
Zeitzonen	kein `localtime` und `utc`

Was ist der WAL-Modus? SQLite sichert Änderungen mit einem *Journal* ab, damit die Datei bei einem Absturz heil bleibt. Standard ist der Journal-Modus `delete`: Während einer Änderung entsteht kurz eine Datei `…-journal`, danach wird sie gelöscht. Im Modus `wal` (*Write-Ahead Logging*) landen Änderungen dagegen zuerst in einer Zusatzdatei `…-wal` und werden später in

die Datenbank übertragen. Manche PC-Programme stellen diesen Modus ein, weil er dort schneller ist. Das SQLite des CMM2 kann ihn nicht. Eine solche Datenbank am PC vor dem Kopieren umstellen: `PRAGMA journal_mode=delete;`

Vorhanden sind unter anderem JSON-Funktionen, Fensterfunktionen (`OVER`), `WITH` (Common Table Expressions, auch rekursiv), Views, Trigger, Fremdschlüssel, `UPSERT` (`ON CONFLICT`), `RETURNING`, `STRICT`-Tabellen, `VACUUM INTO` und `ATTACH`.

6. Ansehen ohne SQL

Für viele Fragen zur Datenbank selbst – welche Tabellen gibt es, wie sind sie aufgebaut? – hat mmsql eigene Befehle. Sie beginnen mit einem Punkt und brauchen kein SQL.

.tables – welche Tabellen gibt es?

`.tables` listet Tabellen und Views mit der Zahl ihrer Spalten. Mit einem Suchmuster (wie bei `LIKE`: `%` = beliebiger Text, `_` = ein Zeichen) nur die passenden:

```
[retro.db]> .tables
name          type  ncol
----------------------
besitz        table 4
computer      table 6
computer_land view  3
hersteller    table 2
ram_log       table 3
 5 row(s)
[retro.db]> .tables h%
name       type  ncol
--------------------
hersteller table 2
 1 row(s)
```

Nach den Beispielen aus Kapitel 5 hat `retro.db` vier Tabellen und eine View.

> **Neu in 1.2:** Suchmuster bei `.tables`, `.schema`, `.indexes` und `.triggers`.

.schema – wie wurde eine Tabelle angelegt?

`.schema` zeigt die `CREATE`-Anweisungen – also den Bauplan. Ohne Angabe von allen Objekten, mit Namen von einem, mit Muster von allen passenden:

```
[retro.db]> .schema hersteller
CREATE TABLE hersteller(name TEXT PRIMARY KEY, land TEXT);
[retro.db]> .schema %log%
CREATE TABLE ram_log(computer_id INTEGER, alt INTEGER, neu INTEGER);
```

Wurde eine Tabelle über mehrere Zeilen angelegt (wie `besitz` in Kapitel 5), zeigt `.schema` sie genau so, Zeile für Zeile.

.columns – die Spalten einer Tabelle

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

Spalte	Bedeutung
`cid`	Nummer der Spalte, ab 0
`name`, `type`	Name und Typ
`notnull`	1 = Pflichtfeld (`NOT NULL`)
`dflt_value`	Vorgabewert (`DEFAULT`)
`pk`	1 = gehört zum Primärschlüssel

.indexes und .triggers

`.indexes` (kurz `.ix`) zeigt alle Indizes mit ihren Spalten, `.indexes computer` nur die einer Tabelle. `.triggers` zeigt die Trigger und ihre Tabellen. Beispiele stehen in Kapitel 5.8.

.browse – durch eine Tabelle blättern

`.browse tabelle` (kurz `.b`) zeigt alle Zeilen seitenweise. Dahinter dürfen `WHERE` und `ORDER BY` stehen. Hier mit 10 Zeilen pro Seite (`.set rows 10`, gilt nach einem Neustart):

```
[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
```

Am Seitenende: Enter oder N blättert weiter, B zurück, Q oder Esc hört auf.

Ein paar Zeichen in Tabellen sind gut zu kennen:

Anzeige	Bedeutung
Wert endet mit `~`	gekürzt – eine Spalte ist höchstens 24 Zeichen breit (mit `.width` breiter, Kapitel 8)
`<40000 bytes>~!`	der Wert ist länger als `maxtext` und wurde nicht gelesen (Kapitel 11)
`(+2 column(s) not shown)`	mehr Spalten, als auf den Bildschirm passen – `.mode line` zeigt alle

Kommen `~` oder `~!` vor, erklärt eine Zeile unter der Tabelle, was sie bedeuten.

.info – Steckbrief der Datenbank

```
[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
```

Zeile	Bedeutung
Page size, Pages	SQLite speichert in Seiten; Größe × Anzahl = Dateigröße
Free pages	freie Seiten nach Löschungen – `.vacuum` gibt sie frei (Kapitel 10)
Journal	wie SQLite Änderungen absichert; auf dem CMM2 immer `delete`
Encoding	wie Texte gespeichert sind, meist UTF-8
user_version	eine Zahl, die Programme frei nutzen dürfen (`PRAGMA user_version = 2;`)

.databases und ATTACH – mehrere Datenbanken zugleich

Es ist immer eine Datenbank offen. Weitere lassen sich mit `ATTACH` anhängen, bis zu zehn. Ihre Tabellen erreicht man dann mit dem Namen davor:

```
[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
```

So lassen sich zum Beispiel Tabellen zwischen Dateien kopieren: `CREATE TABLE kopie AS SELECT * FROM orig.computer;`. `main` ist immer die offene Datenbank, `temp` ein Bereich für vorübergehende Tabellen.

7. Bearbeiten ohne SQL

Für einzelne Zeilen und für den Aufbau von Tabellen gibt es Befehle, die Schritt für Schritt fragen. Sie zeigen dabei die SQL-Anweisung, die sie bauen – so lernt man nebenbei.

Bei allen Eingaben gilt:

Eingabe	Bedeutung
Enter (leer)	bei `.insert`: Vorgabewert bzw. neuer Schlüssel; bei `.edit`: Wert behalten
`NULL` oder `\N`	NULL (kein Wert)
`''`	leerer Text
`\NULL`	der Text „NULL“ (falls jemand wirklich so heißt)

.insert – eine Zeile einfügen

Der Hersteller Oric fehlt noch in der Tabelle `hersteller` (Kapitel 5.4). `.insert` fragt jede Spalte ab und zeigt dabei Typ, Pflichtfeld, Vorgabewert und Primärschlüssel:

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

Jetzt findet auch der JOIN aus Kapitel 5.4 den Oric-1.

> **Neu in 1.2:** `.insert` füllt Tabellen mit bis zu 64 Spalten (Generation 1: 32), bisher waren es höchstens 20.

.edit – eine Zeile ändern

`.edit tabelle schlüssel` holt eine Zeile über ihren Primärschlüssel. Jedes Feld steht mit seinem Wert im Editor: Enter behält ihn, sonst wird er geändert wie jede Eingabe. Hier bekommt der ZX81 seinen vollen Namen:

```
[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.
```

Gespeichert wird erst nach der Rückfrage, und zwar alle Änderungen zusammen in einer Transaktion. Texte über 255 Zeichen und BLOBs (Binärdaten) erscheinen als `(long text, n bytes)` bzw. `(BLOB, n bytes)` und bleiben unverändert; die ändert man mit `UPDATE`.

Hat eine Tabelle keinen einspaltigen Primärschlüssel, nimmt mmsql die `rowid` – die interne Zeilennummer, die jede SQLite-Tabelle hat. Sie wird mit `SELECT rowid, * FROM tabelle;` sichtbar.

.delete – eine Zeile löschen

`.delete tabelle schlüssel` zeigt die Zeile und löscht sie nach einer Rückfrage. Texas Instruments hat keinen Computer in der Liste – weg damit:

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

.create – eine Tabelle anlegen

`.create` fragt für jede Spalte Name, Typ, Primärschlüssel, Pflichtfeld und Vorgabe ab. Ein leerer Name beendet die Liste. Hier eine Wunschliste:

```
[retro.db]> .create wunsch
New table wunsch. 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: preis
    type (INTEGER, TEXT, REAL, BLOB, NUMERIC, Enter = none): REAL
    NOT NULL? (y/n) n
    default (Enter = none): 0
  column4 name:
CREATE TABLE "wunsch"("id" INTEGER PRIMARY KEY,"name" TEXT NOT NULL,"preis" REAL DEFAULT 0)
Create? (y/n) y
Created: wunsch
```

Die fertige Anweisung wird vor dem Ausführen gezeigt – ein guter Weg, `CREATE TABLE` zu lernen. Die Namen setzt mmsql in Anführungszeichen, damit auch Namen mit Leerzeichen gehen. Die Anweisung darf höchstens 255 Zeichen lang sein; größere Tabellen entstehen mit SQL (Kapitel 5.6).

.alter – eine Tabelle umbauen

`.alter` zeigt die Spalten und bietet vier Umbauten an:

Taste	Umbau
A	Spalte hinzufügen
R	Spalte umbenennen
D	Spalte löschen
T	Tabelle umbenennen

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

8. Ausgabe und Export

Wie mmsql das Ergebnis einer Abfrage zeigt, bestimmt der **Ausgabemodus**. Er wird mit `.mode` eingestellt und gilt für alle folgenden Abfragen (er wird in `mmsql.cfg` gespeichert). Zurück zur gewohnten Tabelle geht es immer mit `.mode table`.

Alle Beispiele nutzen dieselbe Abfrage:

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

Die Ausgabemodi

Modus	Ausgabe	gut für
`table`	ausgerichtete Spalten, seitenweise (Standard)	Lesen am Bildschirm
`line`	ein Feld pro Zeile	breite Zeilen, lange Texte
`csv`	durch Komma getrennt, nach RFC 4180	Tabellenkalkulation, Datenaustausch
`list`	mit Trennzeichen, ohne Anführungszeichen	einfache Listen, eigene Programme
`insert`	`INSERT`-Anweisungen	Daten in eine andere Datenbank übertragen
`markdown`	Markdown-Tabelle	Foren, README-Dateien, Dokumentation
`json`	JSON-Liste	Webseiten, Python, andere Programme

line – ein Feld pro Zeile, die Zeilen durch eine Leerzeile getrennt. Lange Texte erscheinen hier vollständig, Zeilenumbrüche darin eingerückt unter dem `=`:

```
[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*. Ein Feld kommt in Anführungszeichen, wenn es ein Komma, ein Anführungszeichen oder einen Zeilenumbruch enthält; Anführungszeichen im Text werden verdoppelt. Die erste Zeile enthält die Spaltennamen (`.headers on`, Standard). Excel, LibreOffice & Co. öffnen solche Dateien direkt:

```
[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 'Ein "Zitat", mit Komma' AS text;
text
"Ein ""Zitat"", mit Komma"
 1 row(s)
```

list – wie csv, aber nie mit Anführungszeichen. Das Trennzeichen stellt `.separator` ein (1 bis 3 Zeichen, `\t` für Tab), die Kopfzeile `.headers on|off`. Hier mit Semikolon und ohne Kopfzeile – und danach alles wieder zurückgestellt:

```
[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 – für jede Zeile eine `INSERT`-Anweisung. Hinter `.mode insert` steht der Tabellenname, der in die Anweisungen kommt. So lassen sich Zeilen in eine andere Datenbank übertragen:

```
[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 – eine Tabelle im Markdown-Format, wie sie GitHub, viele Foren und Wikis darstellen. Die Spalten sind ausgerichtet, damit sie auch als Text lesbar bleiben. Dafür läuft die Abfrage zweimal: Der erste Durchlauf misst die Breiten. Ein `|` im Text wird zu `\|`, ein Zeilenumbruch zu `<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 – eine Liste von Objekten. Zahlen bleiben Zahlen, NULL wird `null`, Texte werden nach JSON-Regeln maskiert. BLOBs erscheinen als Hex-Text oder, mit `.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)
```

> **Neu in 1.2:** `.mode markdown` und `.mode json`.

Wie NULL erscheint: .nullvalue

`.nullvalue text` legt fest, was bei NULL steht – in `table`, `line`, `csv`, `list` und `markdown`. `.nullvalue NULL` macht NULL sichtbar, `.nullvalue ''` zeigt es wieder leer (Standard). In `insert` steht immer `NULL`, in `json` immer `null`. Die Einstellung wird gespeichert.

Spaltenbreiten: .width

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

Im Modus `table` ist eine Spalte so breit wie ihr längster Wert, höchstens 24 Zeichen. `.width` legt Breiten von links an fest; `0` heißt „automatisch“. Längere Werte werden mit `~` gekürzt:

```
[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` macht die zweite Spalte 40 Zeichen breit – gut für lange Beschreibungen. Eine Spalte darf bis zur Ausgabebreite minus 1 breit sein. `.width` gilt, bis mmsql endet; gespeichert wird es nicht.

In eine Datei schreiben: .once und .output

`.once datei` schickt die **nächste** Abfrage in eine Datei, danach geht es wieder auf den Bildschirm. Zusammen mit `.mode csv` entsteht so eine Datei für die Tabellenkalkulation:

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

Die Datei `alt.csv` enthält:

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

`.output datei` schickt **alle** folgenden Abfragen in die Datei, bis `.output` ohne Dateinamen wieder auf den Bildschirm schaltet. Mit `.print` lassen sich Überschriften dazwischen schreiben. Im Modus `table` wird eine Datei als Liste mit `|` geschrieben:

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

Die Datei `liste.txt` enthält:

```
Computer von Acorn:
name|year
BBC Micro Model B|1981
Electron|1983
```

Ein paar Dinge gelten für alle Dateien:

- Eine vorhandene Datei wird überschrieben.
- Die Zeilen enden mit CR LF, wie unter Windows üblich.
- Texte und BLOBs werden vollständig geschrieben, auch sehr lange. Ist ein Wert länger als `maxtext` (Kapitel 11), bricht der Export mit einer klaren Meldung ab, statt still einen gekürzten Wert zu schreiben:

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

Was in Exporten aus den Werten wird

Wert	**csv / list**	**insert / `.dump`**	**json**
NULL	Text aus `.nullvalue`	`NULL`	`null`
ganze Zahl	wie gespeichert (64 Bit)	wie gespeichert	Zahl
Kommazahl	15 gültige Stellen	17 gültige Stellen (verlustfrei)	17 Stellen
Text	unverändert (UTF-8)	`'Text'`, `'` verdoppelt	`"Text"`, maskiert
Text mit Zeilenumbruch	in Anführungszeichen	`'a'||char(10)||'b'` – in einer Zeile	`"a\nb"`
BLOB	die rohen Bytes	`X'0102A0'`	Hex oder Base64

9. Daten rein und raus: Import, Dump, Skripte

CSV-Dateien importieren: .import

Viele Programme können Tabellen als CSV-Datei speichern: Excel, LibreOffice, Adressbücher, Messgeräte. `.import` liest so eine Datei in eine Tabelle. Hier die Datei `mitglieder.csv`:

```
name,ort,seit
Ada,London,2019
Bob,Sydney,2020
"Clara ""CJ""",Berlin,2020
Dave,London,2021
```

Die dritte Zeile zeigt eine Besonderheit von CSV: Ein Feld mit Anführungszeichen steht selbst in Anführungszeichen, und die inneren werden verdoppelt. mmsql versteht das, ebenso Kommas und Zeilenumbrüche innerhalb eines Feldes in Anführungszeichen (RFC 4180).

In eine neue Tabelle: Gibt es die Tabelle noch nicht, legt `.import` sie an. Die erste Zeile liefert die Spaltennamen, alle Spalten bekommen den Typ TEXT:

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

In eine vorhandene Tabelle: Sollen die Spalten andere Typen haben, wird die Tabelle vorher selbst angelegt. Dann sind **alle** Zeilen der Datei Daten; `--skip 1` überspringt die Kopfzeile:

```
[retro.db]> CREATE TABLE mitglied2(name TEXT, ort TEXT, seit INTEGER);
OK
[retro.db]> .import --skip 1 mitglieder.csv mitglied2
 4 row(s) imported into mitglied2
```

Gut zu wissen:

- Das Trennzeichen kommt aus `.separator` und muss hier genau ein Zeichen sein. Für Dateien mit Semikolon: `.separator ;` vor dem Import.
- Der Import läuft in einer Transaktion – schnell, und bei Problemen sauber: Nach zehn fehlerhaften Datensätzen (falsche Zahl von Feldern, zu lang) wird **nichts** importiert. Einzelne fehlerhafte Datensätze werden mit ihrer Nummer gemeldet und übersprungen.
- Ctrl-C bricht ab, dann wird ebenfalls nichts importiert.
- In CSV gibt es kein NULL. Ein leeres Feld (`,,` oder `""`) wird ein leerer Text.
- Leere Zeilen werden übersprungen. Füllbytes am Dateiende (von XMODEM) erkennt mmsql als Ende.

> **Neu in 1.2:** `.import` liest beliebig viele Spalten (bisher höchstens 20), und beliebig viele Felder dürfen länger als 255 Zeichen sein. Ein Datensatz darf bis zu `maxtext` lang sein.

Die ganze Datenbank als Text: .dump

`.dump` schreibt eine Datenbank als SQL-Text: erst den Bauplan jeder Tabelle, dann ein `INSERT` für jede Zeile, zum Schluss Indizes, Views und Trigger. Mit einem Tabellennamen nur diese Tabelle:

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

So ein Dump ist die beste Sicherung, die es gibt: lesbar, mit jedem Texteditor zu öffnen, mit `.read` wieder einzulesen und auch am PC mit `sqlite3` zu verwenden. In eine Datei schreibt ihn `.once`:

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

Zum Format:

- Jede Anweisung endet mit `;`. Zeilenumbrüche in Texten werden als `char(10)` geschrieben, damit jede `INSERT`-Anweisung in einer Zeile bleibt: `'erste'||char(10)||'zweite'`.
- BLOBs (Binärdaten) erscheinen als `X'0A1B…'`.
- Kommazahlen werden mit 17 Stellen geschrieben, damit sie nach dem Zurücklesen exakt gleich sind. Eine Kommazahl ohne Nachkommastellen erscheint als `2.0`, damit sie eine Kommazahl bleibt, und „unendlich“ als `1e999`.

> **Neu in 1.2:** `.dump` fragt SQLite für jeden Wert nach seinem echten Typ (`typeof`). Bisher wurde ein BLOB ohne Steuerzeichen nach dem Zurücklesen zu Text, und ein Text mit Steuerzeichen zu einem BLOB. Jetzt hat jeder Wert nach `.dump` und `.read` wieder genau seinen Typ.

SQL-Skripte ausführen: .read

`.read datei` führt SQL aus einer Textdatei aus – ein **Skript**. Damit lassen sich Datenbanken aus einem Dump wiederherstellen, Beispieldaten einspielen oder immer wiederkehrende Auswertungen mit einem Befehl erledigen. Ein Skript entsteht am PC mit einem Texteditor oder am CMM2 mit `EDIT`.

Ein Skript folgt denselben Regeln wie ein Block am Prompt (Kapitel 4):

1. **Jede Anweisung endet mit `;` am Zeilenende.** Sie darf über beliebig viele Zeilen gehen.
2. Leere Zeilen und Zeilen mit `--` am Anfang (Kommentare) werden übersprungen.
3. Abfragen zeigen ihr Ergebnis wie am Prompt.

4. Diese Punkt-Befehle laufen auch im Skript: `.print` `.echo` `.bail` `.mode` `.headers` `.separator` `.nullvalue` `.timer` `.width` `.once` `.output`. Sie geben dort keine Bestätigung aus, und ihre Einstellungen werden nicht in `mmsql.cfg` gespeichert. Andere Punkt-Befehle meldet mmsql mit der Zeilennummer und überspringt sie.

Ein Beispiel, die Datei `ram.sql`:

```
-- ram.sql: RAM-Statistik nach Prozessor
.print === RAM nach Prozessor ===
SELECT cpu,
       count(*)   AS anzahl,
       max(ramkb) AS max_kb
FROM computer
GROUP BY cpu
ORDER BY max_kb DESC;

.print === Fertig ===
```

```
[retro.db]> .read ram.sql
=== RAM nach Prozessor ===
cpu   anzahl 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)
=== Fertig ===
 1 statement(s) OK, 0 error(s), 10 line(s) read
```

Am Ende meldet `.read`, wie viele Anweisungen fehlerfrei liefen, wie viele Fehler es gab und wie viele Zeilen gelesen wurden. Fehler nennen die Zeile, in der die fehlerhafte Anweisung **beginnt**.

Einstellung	Wirkung
`.bail on`	beim ersten Fehler aufhören (sonst nach zehn)
`.echo on`	jede Anweisung vor dem Ausführen zeigen – gut zur Fehlersuche

Weitere Einzelheiten:

- Am Dateiende läuft eine letzte Anweisung auch ohne `;`.
- Ein `CREATE TRIGGER` endet mit `END;`. Innerhalb eines Triggers `CASE … END` ohne `;` direkt dahinter schreiben, sonst hält mmsql (wie `sqlite3`) es für das Ende des Triggers. Am sichersten: `(CASE … END)`.
- Anweisungen über 255 Zeichen laufen als Ganzes, bis `maxtext`; noch längere werden übersprungen, und `.read` macht nach ihrem `;` weiter.
- Ist nach dem Skript eine Transaktion offen (ein `BEGIN` ohne `COMMIT`), weist mmsql darauf hin.
- `.open` und `.read` selbst laufen in einem Skript nicht. Wer mehrere Skripte oder Datenbanken nacheinander braucht, fügt die Befehle als Block am Prompt ein (Kapitel 4) – dort laufen alle Punkt-Befehle.

> **Achtung beim Umstieg von 1.1:** In mmsql 1.1 war jede Zeile eine eigene Anweisung, ein `;` war nicht nötig. Seit 1.2 endet eine Anweisung erst am `;` – wie bei `sqlite3` und jeder anderen Datenbank. Ein altes Skript ohne `;` läuft jetzt als eine einzige, falsche Anweisung.

```
-- alt.sql (mmsql 1.1, ohne ;)
CREATE TABLE t(a)
INSERT INTO t VALUES(1)
```

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

Die erste Anweisung beginnt in Zeile 2 (Zeile 1 ist ein Kommentar) und reicht bis zum Dateiende – SQLite stolpert über das `INSERT`. Abhilfe: an jedes Zeilenende ein `;` setzen. Dumps von mmsql haben es schon immer.

> **Neu in 1.2:** Anweisungen über mehrere Zeilen, Abfragen mit Ergebnis und Punkt-Befehle in Skripten.

Ohne Datei: die Datenbank im Speicher

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

Ist keine Datenbank offen, nimmt mmsql für SQL, `.read` und `.import` eine **Datenbank im Speicher**. Sie ist sofort da und schnell, wird aber nirgends gespeichert. Gut zum Ausprobieren, Rechnen und Testen:

```
mmsql> SELECT 6 * 7 AS antwort, date('2026-11-20', '+100 days') AS termin;
(No database open - using a database in memory. .open file.db for a file.)
antwort termin
------------------
42      2027-02-28
 1 row(s)
[:memory:]> CREATE TABLE notiz(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 notizen.db
OK - copy written to notizen.db
```

- Der Prompt zeigt `[:memory:]>`.
- Alle Befehle funktionieren, außer `.restore`.
- Hat die Datenbank im Speicher Tabellen, fragen `.open`, `.close` und `.quit` vorher nach, denn ihr Inhalt geht verloren. `.backup datei.db` speichert sie als Datei.
- `.open :memory:` öffnet sie gezielt, `RUN "mmsql", ":memory:"` gleich beim Start.

Austausch mit dem PC

- Dateien per XMODEM übertragen oder die SD-Karte am PC lesen.
- Datenbanken vom CMM2 öffnen am PC `sqlite3`, der *DB Browser for SQLite* und jedes andere SQLite-Programm.
- Datenbanken vom PC müssen im Journal-Modus `delete` gespeichert sein, nicht `wal` (Kapitel 5.12). Am PC vorher: `PRAGMA journal_mode=delete;`
- XMODEM füllt den letzten Block auf (Tera Term mit Ctrl-Z, der CMM2 mit NUL-Bytes). `.read` und `.import` erkennen das als Dateiende.
- Ein Dump von `sqlite3` am PC lässt sich mit `.read` einlesen, solange er keine Funktionen einer neueren SQLite-Version benutzt (etwa `unistr()` ab SQLite 3.50).

10. Sicherheit und Pflege

mmsql passt auf offene Transaktionen auf

Ist eine mit `BEGIN` begonnene Transaktion noch nicht beendet, fragen `.quit`, `.close` und `.open` nach, was damit geschehen soll:

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

Taste	Wirkung
C	*commit* – Änderungen behalten
R	*rollback* – Änderungen verwerfen
Esc	abbrechen – die Datenbank bleibt offen, die Transaktion auch

Nach einem Fehler innerhalb einer Transaktion fragt mmsql bei SQLite nach, ob die Transaktion noch offen ist. Hat SQLite sie selbst beendet, meldet mmsql: „Note: SQLite has undone the open transaction.“

Ctrl-C: abbrechen, ohne Schaden

Eine SQL-Anweisung, die zu lange läuft, bricht **Ctrl-C** ab. mmsql läuft danach weiter und sagt genau, was SQLite rückgängig gemacht hat:

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

Was lief?	Was ist danach?
eine Abfrage (`SELECT`)	nichts geändert; eine offene Transaktion bleibt offen
eine Änderung ohne Transaktion	die Änderung ist ganz rückgängig gemacht
eine Änderung in einer Transaktion	die **ganze** Transaktion seit `BEGIN` ist rückgängig gemacht

`.read`, `.import` und `.dump` brechen bei Ctrl-C sauber ab: `.import` importiert dann nichts, `.dump` meldet die Datei als unvollständig.

An anderen Stellen – am Prompt oder während mmsql eine Datei schreibt – beendet Ctrl-C mmsql wie jedes BASIC-Programm. Auch dann bleibt die Datenbank heil: Eine offene Transaktion wird rückgängig gemacht, nichts davon wird geschrieben. Mit `OPTION BREAK 0` ist Ctrl-C abgeschaltet; dann weist mmsql beim Start darauf hin, dass sich SQL nicht abbrechen lässt.

Sichern und zurückspielen: .backup und .restore

`.backup datei.db` schreibt eine saubere, kompakte Kopie der offenen Datenbank. Gibt es die Datei schon, fragt mmsql vor dem Überschreiben.

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

`.restore datei.db` macht das Gegenteil: Es ersetzt die offene Datenbank durch eine Sicherung.

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

`.restore` geht vorsichtig vor:

1. Es prüft, ob die Datei wirklich eine SQLite-Datenbank ist.
2. Es fragt, denn der jetzige Inhalt geht verloren (bei offener Transaktion fragt es wie `.close`).
3. Es kopiert die Sicherung zuerst neben die Datenbank (`retro.db-restore`). Erst wenn diese Kopie vollständig ist, wird die Datenbank ersetzt. Ist die SD-Karte voll, bleibt alles, wie es war.
4. Eine übrig gebliebene `retro.db-journal` gehört zum alten Inhalt und wird gelöscht.

Die Sicherung selbst bleibt unverändert – sie lässt sich beliebig oft zurückspielen. Ein Dump (Kapitel 9) ist die zweite Art der Sicherung: als lesbarer Text, auch für den PC.

Prüfen: .check

Befehl	prüft
`.check`	schnell das Wichtigste (`PRAGMA quick_check`)
`.check full`	alles, auch die Indizes (`PRAGMA integrity_check`)

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

Bei großen Datenbanken dauert das, auf Generation 1 auch Minuten (Ctrl-C bricht ab). Findet `.check` Probleme, listet es sie auf. Dann mit `.backup neu.db` retten, was SQLite noch lesen kann, und mit der Kopie weiterarbeiten.

Aufräumen: .vacuum

Gelöschte Zeilen hinterlassen freie Seiten in der Datei; sie wird nicht von allein kleiner. `.vacuum` baut die Datei neu auf und zeigt die Größe vorher und nachher:

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

Dafür braucht SQLite kurzzeitig etwa so viel freien Platz auf der Karte, wie die Datenbank groß ist. `.reindex` baut alle Indizes neu auf (oder die einer Tabelle, oder einen bestimmten).

Fremdschlüssel: .fk und .fkcheck

Ein Fremdschlüssel (`REFERENCES` in `CREATE TABLE`, Kapitel 5.6) sagt: Dieser Wert muss in einer anderen Tabelle vorkommen. SQLite prüft das aber nur, wenn man es einschaltet – das gilt für die Sitzung, bis mmsql endet:

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

`.fkcheck` findet Zeilen, die eine solche Regel verletzen – auch wenn die Prüfung aus war, als sie geschrieben wurden.

Dateien auf der Karte: .files und .kill

`.files` zeigt Dateien mit Größe und Datum, ohne Angabe die `*.db`-Dateien des aktuellen Ordners, sonst nach Muster (`.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
notizen.db     8192 B    2026-11-20 10:41
retro.db       44 KB     2026-11-20 10:52
retro_sich.db  44 KB     2026-11-20 10:49
 4 file(s)
```

Größen unter 10 000 Bytes stehen genau in Bytes, größere in KB.

`.kill datei` löscht eine Datei nach Rückfrage – immer genau eine, ohne `*`. Die offene Datenbank lässt sich so nicht löschen (erst `.close`).

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

Liegt neben einer Datenbank eine Datei `…-journal`, ist das der Rest einer abgebrochenen Transaktion. Beim nächsten Öffnen räumt SQLite sie selbst auf. Wird eine Datenbank mit `.kill` gelöscht, bietet mmsql an, das Journal mitzulöschen.

11. Einstellungen

.set – alle Optionen auf einen Blick

`.set` allein zeigt alle Optionen mit Wert und kurzer Erklärung, nach Themen geordnet: Ausgabe, Eingabe, Zeichen und Texte, Sonstiges.

```
[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_de.hlp  help file (e.g. mmsql_de.hlp)
File: A:/sql/mmsql.cfg
```

`.set name wert` ändert eine Option, speichert sie sofort in `mmsql.cfg` und zeigt die geänderte Zeile:

```
[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` liegt neben dem Programm und ist eine einfache Textdatei mit Zeilen `name=wert`. Man kann sie auch am PC bearbeiten oder löschen (dann gelten wieder die Standardwerte). Die vollständige Liste mit allen erlaubten Werten steht in Anhang C.

Die Optionen erklärt

Ausgabe

- `mode`, `headers`, `separator`, `nullvalue` – wie `.mode`, `.headers`, `.separator` und `.nullvalue` (Kapitel 8).
- `blob hex|base64` – wie `.mode json` Binärdaten schreibt.
- `width` – wie breit mmsql Tabellen ausgibt, 40 bis 250 Zeichen. Standard ist die Breite des Bildschirms (im MODE 1 mit der Standardschrift 100 Zeichen). Passt eine Tabelle nicht, lässt mmsql rechte Spalten weg und sagt es. `width 0` heißt: immer so breit wie der Bildschirm, auch nach einem Wechsel des MODE; `.set` zeigt dann in Klammern die tatsächliche Breite, z. B. `0 (99)`. Die letzte Spalte des Bildschirms bleibt dabei frei, damit keine Zeile ungewollt umbricht.
- `rows` – Zeilen pro Seite beim Blättern, 5 bis 100. Gilt nach einem Neustart.

Eingabe

- `enter run|line` – ob Enter auf einer einzelnen Zeile ausführt (`run`) oder eine neue Zeile beginnt (`line`), Kapitel 4.
- `paste` – die Paste-Zeit in Millisekunden, 0 bis 1000 (0 = aus), Kapitel 4.

- `history` – wie viele Eingaben der Verlauf behält, 0 bis 200. Gilt nach einem Neustart.

Zeichen und Texte

- `utf8 dot|raw` und `dotchar` – die Anzeige von Umlauten, Kapitel 4.
- `maxtext` – der längste Text, den mmsql auf einmal liest oder schreibt, in Bytes (1024 bis 1 048 576, Standard 32 768). Dafür hält mmsql zwei Speicherbereiche dieser Größe bereit. Auf Generation 2 und 3 ist auch 1 MB kein Problem; auf Generation 1 bleibt man besser klein. Reicht der Speicher nicht, setzt mmsql den Wert auf 32 768 zurück.

Sonstiges

- `screen` – der Bildschirm-Modus (siehe unten).
- `timer 1` – nach jeder Anweisung die Laufzeit zeigen (dasselbe wie `.timer on`).
- `helpfile` – die Hilfedatei, z. B. `mmsql_de.hlp`.

Bildschirm-Modus: .set screen

> **Neu in 1.2:** `.set screen` und `.set width 0`.

Der CMM2 kennt viele Bildschirm-Modi (`MODE 1` bis `MODE 19`), von 320 × 200 bis 1920 × 1080 Punkten. Je höher die Auflösung, desto mehr Zeilen und Spalten passen auf den Bildschirm – für breite Tabellen ein großer Gewinn.

`.set screen n` stellt den MODE sofort um und merkt ihn sich: Bei jedem Start schaltet mmsql in diesen Modus, beim Beenden zurück in den Modus und die Schrift von vorher. `.set screen 0` (Standard) lässt den Bildschirm, wie er ist.

MODE	Auflösung	Zeichen (Standardschrift)	Hinweis
1	800 × 600	100 × 50	Standard beim Einschalten
9	1024 × 768	128 × 64	
11	1280 × 720	160 × 60	Breitbild
15	1280 × 1024	160 × 85	
16	1920 × 1080	240 × 90	Breitbild, nur Generation 2

Welche Modi ein Monitor zeigt, steht im Handbuch des CMM2 beim Befehl `MODE`. Zusammen mit `.set width 0` passt sich die Ausgabebreite dem Modus an. Das Terminalfenster stellt mmsql bei jedem Wechsel auf die neue Größe ein.

Weitere kleine Befehle

Befehl	Wirkung
`.timer on` / `.timer off`	Laufzeit nach jeder Anweisung zeigen
`.cls`	Bildschirm und Terminal löschen; stellt auch das Terminalfenster wieder auf die Bildschirmgröße
`.print text`	eine Zeile ausgeben (in Skripten und mit `.output` nützlich)
`.version` (`.ver`)	die Versionen von mmsql, SQLite und 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
```

Diese drei Zeilen gehören zu jeder Fehlermeldung.

Hilfe zu einem Befehl

`.help befehl` zeigt die Hilfe zu einem Befehl, kurz und auf Deutsch (mit `.set helpfile mmsql_de.hlp`):

```
[retro.db]> .help width
.width [n1 n2 ...]
Spaltenbreiten im table-Modus (auch .browse), ab der ersten Spalte:
0 = automatisch (so breit wie die Werte, bis 24), sonst bis zur
Ausgabebreite (.set width) - 1. Laengere Werte werden gekuerzt (~).
  .width 0 40     2. Spalte 40 breit, die anderen automatisch
  .width          wieder alle automatisch
Wird nicht gespeichert: gilt, bis mmsql endet.
```

12. Für Kenner von MySQL, MariaDB, MS SQL Server & Co.

Wer schon mit MySQL, MariaDB, Microsoft SQL Server, Oracle, PostgreSQL oder Sybase gearbeitet hat, kennt SQL – und wird sich trotzdem an ein paar Stellen wundern. **Hier gibt es nur SQLite: eine Datei, ein Benutzer, kein Server.** Das macht vieles einfacher und manches anders.

Was es nicht gibt

aus Server-Datenbanken	in SQLite
Server, Verbindung, Benutzer, `GRANT`, Passwörter	nichts davon – die Datenbank ist eine Datei; wer die Datei hat, hat die Daten
gespeicherte Prozeduren und Funktionen (`CREATE PROCEDURE`, PL/SQL, T-SQL-Blöcke)	gibt es nicht; Logik gehört ins BASIC-Programm oder in Trigger
`GO` als Trennzeichen (SQL Server, Sybase)	jede Anweisung endet mit `;`
`USE datenbank`	`.open datei.db` – oder `ATTACH` für mehrere zugleich
Sequenzen, `AUTO_INCREMENT`, `IDENTITY`	`INTEGER PRIMARY KEY` zählt von selbst hoch
eigene Datums- und Uhrzeittypen	Datum als Text `'2026-11-20'` oder Zahl; Funktionen `date()`, `datetime()`, `strftime()`
`BOOLEAN` als eigener Typ	0 und 1 (`TRUE` und `FALSE` werden verstanden)
`ALTER TABLE … MODIFY / ALTER COLUMN`	Typ ändern geht nur über eine neue Tabelle (Kapitel 5.6)
Mathe-Funktionen im SQLite des CMM2	fehlen (Kapitel 5.12)

BEGIN ist eine Transaktion

In T-SQL und PL/SQL leitet `BEGIN … END` einen Anweisungsblock ein. In SQLite heißt `BEGIN` immer „Transaktion beginnen“ – wie `BEGIN TRAN` bei SQL Server oder `START TRANSACTION` bei MySQL. `BEGIN … END` als Block gibt es nur im Rumpf eines Triggers:

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

Gleiche Idee, andere Schreibweise

MySQL / MariaDB	SQL Server / Sybase	Oracle	SQLite
`LIMIT 10`	`TOP 10`	`FETCH FIRST 10 ROWS ONLY`	`LIMIT 10` (auch `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)` oder `coalesce(a, b)`
`` `name` ``	`[name]`	`"name"`	`"name"` – `` `name` `` und `[name]` gehen auch
`SHOW TABLES`	`sp_tables`	`SELECT … FROM user_tables`	`.tables` oder `SELECT name FROM sqlite_master;`
`DESCRIBE t`	`sp_help t`	`DESC t`	`.columns t` oder `PRAGMA table_info(t);`
`INSERT … ON DUPLICATE KEY UPDATE`	`MERGE`	`MERGE`	`INSERT … ON CONFLICT(…) DO UPDATE`

Typen sind Empfehlungen

SQLite speichert in jeder Spalte, was kommt. Der Spaltentyp ist eine **Empfehlung** (*type affinity*): Passt ein Wert, wird er umgewandelt, sonst so gespeichert, wie er ist (Beispiel in Kapitel 5.6). `VARCHAR(20)` ist erlaubt, die 20 wird aber nicht geprüft. Wer strenge Typen will, legt die Tabelle mit `STRICT` an:

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

Was sonst auffällt

- **Fremdschlüssel** werden nur geprüft, wenn sie eingeschaltet sind: `.fk on` (bzw. `PRAGMA foreign_keys = ON;`), jedes Mal nach dem Öffnen.
- `LIKE` unterscheidet bei englischen Buchstaben nicht zwischen groß und klein; `GLOB` tut es und nimmt `*` und `?` als Platzhalter.
- **Texte in einfachen Anführungszeichen**, Namen in doppelten. `"Apple"` ohne passende Spalte versteht SQLite zur Not als Text – verlassen sollte man sich darauf nicht.
- **Jede Tabelle hat eine** `rowid`, eine interne Zeilennummer. Eine Spalte `INTEGER PRIMARY KEY` ist diese Nummer.
- **Sperren gelten für die ganze Datei.** Auf dem CMM2 mit einem einzigen Benutzer spielt das keine Rolle.
- `RIGHT JOIN` **und** `FULL OUTER JOIN` gibt es seit SQLite 3.39, also auch auf dem CMM2.

Was mmsql anders macht als die Werkzeuge der großen Datenbanken

- Am Prompt läuft **eine Zeile** ohne `;` – wie bei vielen grafischen Werkzeugen. Mehrere Zeilen sind ein Block, und dort beendet `;` jede Anweisung wie in `mysql`, `sqlcmd` oder `sqlplus`.
- Punkt-Befehle (`.tables`, `.schema`) ersetzen Befehle wie `SHOW` oder `DESCRIBE`; sie kommen vom Vorbild `sqlite3`.
- Kein `GO`, kein `/` am Zeilenanfang wie in `sqlplus`.

13. Technische Daten und Grenzen

Alle Werte stammen aus dem Quelltext von mmsql 1.2. Die meisten Grenzen folgen aus einer einzigen Regel von MMBasic: **Ein String ist höchstens 255 Zeichen lang.** Wo mmsql mehr kann, benutzt es LONGSTRING-Puffer – das sind Speicherbereiche für beliebig lange Texte (Anhang F, Tipp 4 in Kapitel 15). Ihre Größe ist die Option `maxtext`, Standard 32 768 Zeichen.

Daneben hat SQLite selbst Grenzen. Die wichtigste: **Eine Tabelle hat höchstens 2000 Spalten.** Das ist der Standardwert von SQLite; auf dem CMM2 wurde er nicht ausprobiert. Texte und BLOBs dürfen in SQLite bis zu 1 GB groß sein – auf dem CMM2 setzt der Speicher früher eine Grenze.

Eingabe

Was	Grenze	Was heißt das?
eine Zeile im Editor	255 Zeichen	So lang darf ein MMBasic-String sein. Längere Anweisungen auf mehrere Zeilen verteilen (F3).
Zeilen einer Eingabe	20	Mehr Zeilen passen nicht in den Editor. Noch längere Anweisungen gehören in ein Skript (`.read`).
eine Anweisung	`maxtext`, Standard 32 768 Zeichen	gilt am Prompt, im Block und in Skripten. 20 Zeilen à 255 Zeichen sind rund 5 100 Zeichen – im Editor ist also immer Platz, nur Skripte stoßen an die Grenze.
Anweisungen in einer Zeile	32	Mehr Anweisungen, mit `;` getrennt, nimmt eine einzelne Zeile nicht.
Verlauf	0 bis 200 Einträge, Standard 50	Auch lange Einträge über viele Zeilen zählen als einer.
Wert bei `.insert` und `.edit`	255 Zeichen	Längere Werte mit `INSERT` oder `UPDATE` in SQL eingeben.
Paste-Zeit	0 bis 1000 ms	siehe Kapitel 4

Ausgabe

Was	Grenze	Was heißt das?
Spalten in `table` und `.browse`	64 (Generation 1: 32)	Eine Seite der Tabelle liegt im Speicher, dafür ist die Spaltenzahl fest. Breitere Ergebnisse zeigen `.mode line`, `csv` und `json` vollständig.
Spalten in `csv`, `list`, `line`, `insert`, `json`	wie SQLite, also bis 2000	Diese Modi schreiben Wert für Wert und brauchen keinen Puffer.
Spalten in `markdown`	2000	so viele Spaltenbreiten merkt sich mmsql
Breite einer Spalte in `table`	automatisch höchstens 24 Zeichen, mit `.width` bis zur Ausgabebreite − 1	Längere Werte enden mit `~`.
Länge eines Spaltennamens	64 Zeichen	längere Namen werden in der Anzeige gekürzt
ein Text oder BLOB	bis `maxtext` (1 024 bis 1 048 576 Bytes)	Am Bildschirm erscheint ein längerer Wert als `~!`; ein Export bricht mit einer Meldung ab, statt den Wert gekürzt zu schreiben. Abhilfe: `.set maxtext` größer stellen.
Ausgabebreite (`width`)	40 bis 250 Zeichen	
Zeilen pro Seite (`rows`)	5 bis 100	
Namen bei `.schema` und `.files`	200	Bei mehr Treffern ein Suchmuster angeben.
Kommazahlen	15 Stellen am Bildschirm, 17 in `insert`, `.dump`, `json`	Mit 17 Stellen ist eine Zahl nach dem Zurücklesen exakt gleich.
ganze Zahlen	64 Bit	bis etwa ±9,2 Trillionen, ohne Rundung

Befehle

Was	Grenze	Was heißt das?
Spalten bei `.insert` und `.edit`	64 (Generation 1: 32)	bis mmsql 1.1 waren es 20
Spalten bei `.import`	wie SQLite, also bis 2000	mmsql selbst zählt nicht mit (bis mmsql 1.1: 20).
eine CSV-Zeile bei `.import`	etwa `maxtext` − 300 Zeichen	mmsql baut aus jeder CSV-Zeile eine fertige Anweisung `INSERT INTO "tabelle" VALUES('wert','wert',…)` und schickt sie an SQLite. Diese Anweisung muss in den Puffer passen, der `maxtext` groß ist; 300 Zeichen bleiben für den Anfang und das Ende der Anweisung frei. Mit dem Standard darf eine CSV-Zeile also gut 32 000 Zeichen lang sein. Ist sie länger, meldet `.import` „Record n: longer than maxtext (.set maxtext) - skipped“ – dann `.set maxtext 100000` und noch einmal.
Trennzeichen	1 bis 3 Zeichen, bei `.import` genau eins	Bei der **Ausgabe** darf das Trennzeichen länger sein, etwa `.separator " | "` für eine gut lesbare Liste. Beim **Einlesen** prüft mmsql die Datei Zeichen für Zeichen; dort geht nur ein einzelnes Zeichen wie `,`, `;` oder Tab (`\t`).
Anweisung von `.create` und `.alter`	255 Zeichen	Größere Tabellen mit `CREATE TABLE` in SQL anlegen.
Fehler bis zum Abbruch (`.read`, `.import`)	10	`.import` übernimmt dann gar nichts.
`.nullvalue`	20 Zeichen	
angehängte Datenbanken (`ATTACH`)	10	Grenze von SQLite
Größe einer Datenbank	so groß, wie die SD-Karte erlaubt	

Speicher

Was	Wert (gemessen auf Generation 2, 8.00.00b1)
Programm	ca. 100 KB, 4269 Zeilen nach dem Laden – 19 % des Programmspeichers
Daten	ca. 0,7 MB, davon zweimal `maxtext` für die LONGSTRING-Puffer
`maxtext` 1 MB	etwa 2 MB zusätzlich – auf Generation 2 und 3 kein Problem

Dateien

Datei	Inhalt
`mmsql.cfg`	Einstellungen, eine Zeile `name=wert` je Option
`mmsql.his`	Verlauf, eine Zeile je Eintrag. Hat ein Eintrag mehrere Zeilen, beginnen die weiteren mit dem Steuerzeichen `CHR$(30)`; eine Zeile mit genau 255 Zeichen wird mit `CHR$(31)` fortgesetzt. Dateien von mmsql 1.1 bleiben lesbar.
Exporte	Zeilenende CR LF, wie unter Windows

Dateinummern: mmsql benutzt #7 (`.read`, `.import`, `.restore`, Hilfe), #8 (`.once`, `.output`) und #9 (`mmsql.cfg`, `mmsql.his`). Wer Teile von mmsql in eigene Programme übernimmt, sollte diese Nummern dort nicht gleichzeitig verwenden.

Was mmsql beim Start einstellt

Einstellung	warum	danach
`OPTION CRLF CR`	Enter liefert in Firmware 8 sonst CR und LF – mmsql sähe zwei Tasten	gilt nur, solange mmsql läuft
Terminalfenster = Bildschirmgröße	Terminal und Bildschirm zeigen 1:1 dasselbe	bleibt
`MODE` aus `.set screen`	nur wenn eingestellt	beim Beenden zurück

14. mmsql und sqlite3 – ein ehrlicher Vergleich

Das Vorbild von mmsql ist `sqlite3`, das Kommandozeilen-Werkzeug, das die Entwickler von SQLite selbst pflegen. Es ist in C geschrieben, läuft auf jedem PC und hat über zwanzig Jahre Entwicklung hinter sich. mmsql ist in MMBasic geschrieben und für den CMM2 gemacht. Hier steht ehrlich, was gleich ist, was mmsql mehr kann und was ihm fehlt.

Gleich oder fast gleich

Bereich	wie bei sqlite3
SQL	dieselbe SQLite-Sprache – es ist ja dieselbe Datenbank
Punkt-Befehle	`.open` `.tables` `.schema` `.indexes` `.databases` `.dump` `.read` `.import` `.mode` `.headers` `.separator` `.nullvalue` `.width` `.once` `.output` `.print` `.bail` `.echo` `.timer` `.backup` `.restore` `.help` `.quit`
Suchmuster	bei `.tables`, `.schema`, `.indexes`, `.triggers`
Ausgabemodi	`csv`, `list`, `line`, `insert`, `markdown`, `json`
Skripte	`;` beendet eine Anweisung, Anweisungen über mehrere Zeilen, Trigger bis `END;` – mit derselben Grenze bei `CASE … END;` im Trigger
Dateien	Datenbanken und Dumps sind zwischen beiden austauschbar

Was nur mmsql kann

Funktion	wofür
`.browse` mit Blättern vor und zurück	große Tabellen am Bildschirm lesen
`.insert`, `.edit`, `.delete`	Zeilen per Abfrage bearbeiten, ohne SQL
`.create`, `.alter`	Tabellen Schritt für Schritt anlegen und umbauen
`.columns`, `.info`, `.check`, `.fk`, `.fkcheck`	Aufbau und Zustand der Datenbank ohne PRAGMA-Kenntnisse
`.files`, `.kill`	Dateien auf der SD-Karte, ohne mmsql zu verlassen
`.set` mit `mmsql.cfg`	Einstellungen bleiben über Neustarts erhalten
eine Zeile läuft ohne `;`	bequemer am Prompt
Editor mit Statuszeile, Verlauf über Neustarts	auch ohne PC-Tastatur angenehm
Erkennung eingefügter Blöcke	SQL vom PC einfach ins Terminal kopieren
Stern im Prompt, Rückfrage bei offener Transaktion	keine Änderung geht aus Versehen verloren
Ctrl-C mit genauer Meldung, was rückgängig gemacht wurde	sicheres Abbrechen
„Did you mean .tables?“	hilft beim vergessenen Punkt
`:memory:` von selbst, wenn keine Datei offen ist	sofort loslegen
Bildschirm und Terminal zugleich, 1:1	am CMM2 und am PC arbeiten
Hilfe auf Deutsch	
`.set screen`	Bildschirm-Modus des CMM2

Was sqlite3 kann und mmsql nicht

Funktion von sqlite3	in mmsql
weitere Ausgabemodi: `box`, `column`, `html`, `tcl`, `quote`, `tabs`, `ascii`, `qbox`	nicht vorhanden; `table` von mmsql entspricht etwa `column` von sqlite3 (sqlite3 zeichnet bei `table` Rahmen)
`.parameter` (benannte Werte für `?` und `:name`)	nicht vorhanden
`.recover` (Daten aus kaputten Dateien retten)	nicht möglich – braucht eine Erweiterung; mmsql hat `.check` und `.backup`
`.eqp`, `.expert`, `.scanstats` (Abfragepläne)	nicht möglich – `EXPLAIN` fehlt im SQLite-Modul
`.load` (Erweiterungen)	nicht möglich
`.shell`, `.system`, `.excel`, Ausgabe in Programme (`.output |cmd`)	nicht vorhanden – der CMM2 hat kein Betriebssystem mit Kommandozeile, nur den MMBasic-Prompt; aus mmsql heraus lassen sich keine anderen Programme starten
`.import` mit `--csv`, `--schema`, `--ascii`	nur `--skip`; Trennzeichen über `.separator`
`.dump` mit Filtern, `--data-only`, `--nosys`	nur ganze Datenbank oder eine Tabelle
`.changes`, `.stats`, `.show`, `.limit`, `.lint`, `.sha3sum`, `.archive`, `.cd` …	nicht vorhanden; geänderte Zeilen meldet mmsql nach jeder Anweisung von selbst
Umlaute tippen und anzeigen	am CMM2 nicht möglich (Schrift); mmsql zeigt Platzhalter, die Daten bleiben echt
Geschwindigkeit	sqlite3 ist C auf einem PC – vielfach schneller. Auf dem CMM2 arbeitet SQLite selbst in C; nur die Anzeige ist BASIC. Ein Import von 500 Zeilen × 8 Spalten dauert auf Generation 2 etwa 3 Sekunden.

Wo sie sich anders verhalten

- **Das Semikolon am Prompt:** sqlite3 wartet bei einer Zeile ohne `;` auf mehr Text. mmsql führt eine einzelne Zeile mit Enter aus. In Blöcken und Skripten verhalten sich beide gleich.
- **Lange Texte:** sqlite3 zeigt alles. mmsql kürzt in `table` auf die Spaltenbreite (`~`) und zeigt Werte über `maxtext` gar nicht (`~!`); `.mode line` zeigt alles bis `maxtext`.
- **Zeilenumbrüche im Dump:** sqlite3 schreibt neuere Dumps mit `replace(…)` oder `unistr(…)`, mmsql mit `char(10)`. Beide lesen den Dump von mmsql; einen Dump eines neueren sqlite3 liest mmsql, solange er keine Funktion benutzt, die das SQLite des CMM2 nicht kennt.
- **Fehlermeldungen:** sqlite3 meldet „Parse error near line 3“, mmsql „Line 3: SQL: …“ und zeigt die Anweisung mit ihrer Länge.

Fazit

Für die tägliche Arbeit mit Datenbanken auf dem CMM2 hat mmsql alles Wichtige von sqlite3 und dazu vieles, was auf einem kleinen Rechner ohne Maus das Leben leichter macht. Was fehlt, sind vor allem Dinge, die das SQLite-Modul nicht hergibt (Abfragepläne, Erweiterungen, Rettung mit `.recover`), die es auf dem CMM2 nicht gibt (Kommandozeile eines Betriebssystems, Excel) – und seltene Spezialbefehle. Wer davon etwas braucht, nimmt die Datei mit zum PC: Sie ist dort sofort lesbar.

15. Unter der Haube – mmsql als Fundgrube für eigene Programme

mmsql ist vollständig in MMBasic geschrieben, etwa 5000 Zeilen, und der Quelltext liegt offen in `mmsql.bas` (MIT-Lizenz: übernehmen ausdrücklich erlaubt). Wer selbst Programme mit SQLite schreibt, findet darin fertige Lösungen für Probleme, über die man sonst stolpert. Dieses Kapitel ist eine Landkarte dazu: was wo steht und warum es so gebaut ist.

Der Aufbau: drei Schichten

Vorsilbe	Schicht	Inhalt
`MS_`	Kern	alles, was mit der Datenbank spricht – **ohne Bildschirmausgabe**. Diese Routinen lassen sich fast unverändert in eigene Programme übernehmen.
`CLI_`	Befehlszeile	die Punkt-Befehle, die Ausgabe, Dateien, Verlauf, Einstellungen
`ED_`	Editor	der Editor am Prompt

Jede Routine hat einen Kommentar darüber, der sagt, was sie tut und was sie zurückgibt. Globale Variablen tragen dieselbe Vorsilbe wie ihre Schicht. Mit `OPTION EXPLICIT` muss jede Variable deklariert sein; Tippfehler in Namen fallen so sofort auf.

Tipp 1: SQL-Fehler abfangen, ohne dass das Programm stehen bleibt

Ein SQL-Fehler ist in MMBasic ein normaler Fehler: Ohne Vorkehrung hält das Programm an. Alle Datenbankzugriffe in mmsql folgen deshalb diesem Muster (`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` gilt nur für den **nächsten** Befehl. Danach steht in `MM.ERRNO`, ob es einen Fehler gab, und in `MM.ERRMSG$` die Meldung, etwa `SQL: no such table: computers`. Das `ON ERROR CLEAR` davor sorgt dafür, dass kein alter Fehler übrig ist. Auch Ctrl-C während einer SQL-Anweisung ist so ein Fehler („SQL: stopped by Ctrl-C“) – so bricht mmsql Abfragen ab, ohne selbst zu enden (`MS_SetErr`, `CLI_BrkMsg`).

Tipp 2: Ist eine Transaktion offen?

SQLite sagt das nicht direkt. `MS_TxOpen%` fragt mit einem Trick: Ein `BEGIN` innerhalb einer Transaktion ist ein Fehler („cannot start a transaction within a transaction“). Klappt das `BEGIN`, war keine offen, und ein `ROLLBACK` macht die leere Transaktion gleich wieder zu.

Tipp 3: Einen Wert lesen, egal welchen Typ er hat

`SQL(n)` liefert den Wert der Spalte n – als String, wenn es ein Text ist, als Zahl, wenn es eine Zahl ist. Ein Programm, das beliebige Tabellen zeigt, weiß aber vorher nicht, was kommt. `MS_ValText$` versucht es erst als String; meldet MMBasic „Expected a string“, ist es eine Zahl. Dann liest es den Wert als Kommazahl **und** als ganze Zahl und vergleicht: Sind beide gleich, ist es eine ganze Zahl (wichtig für 64-Bit-Werte, die als Kommazahl ungenau würden). NULL erkennt `SQL(ISNULL n)`.

Tipp 4: Texte über 255 Zeichen

Ein MMBasic-String hat höchstens 255 Zeichen, SQLite-Texte können Megabytes groß sein. Die Brücke sind **LONGSTRING**-Felder: INTEGER-Arrays, die beliebig lange Texte aufnehmen.

- **Lesen:** `SQL(LONGSTRING n, puffer%())` holt die Spalte n in den Puffer und gibt ihre Länge zurück (`MS_LongRead%`). Ist sie länger als der Puffer, meldet die Firmware einen Fehler mit der nötigen Länge – mmsql liest sie aus der Meldung und kann so sagen, auf welchen Wert man `maxtext` stellen muss.
- **Schreiben:** `SQL EXEC puffer%()` und `SQL QUERY puffer%()` nehmen die ganze Anweisung als LONGSTRING (`MS_ExecL%`). So entstehen Anweisungen beliebiger Länge, zum Beispiel beim Import (`CLI_CsvRec%`, `CLI_ExecSB%`) und bei `.insert` (`CLI_LBQuote`).
- **Stückweise ausgeben:** `LGETSTR$(puffer%(), ab, länge)` holt ein Stück als String (`CLI_LPiece$`, das dabei kein UTF-8-Zeichen zerschneidet).

> **Wichtig für eigene Programme:** In Firmware 8.00.00b1 (und 7.00.00RC6) schreibt `LGETSTR$` mit genau 255 Zeichen ein Byte zu viel in den Speicher. Das kann die offene Datenbank beschädigen („disk I/O error“ beim `COMMIT`). mmsql fragt deshalb nie mehr als **254** Zeichen ab. Gefunden beim Bau von mmsql 1.1, von Peter Mather in 8.00.00b2 behoben.

Lange Texte aus `sqlite_master` (der Bauplan einer Tabelle) liest `MS_SqlPart$` ganz ohne LONGSTRING: mit `substr(sql, ?, 200)` in Stücken von 200 Zeichen.

Tipp 5: Blättern in großen Ergebnissen

Eine Abfrage kann Tausende Zeilen liefern, ein Array im Speicher hat aber eine feste Größe. `MS_Fetch%` und `CLI_Grid` holen deshalb immer nur eine Seite – und eine Zeile mehr, um zu wissen, ob es weitergeht:

```
SELECT * FROM (die Abfrage des Benutzers) LIMIT ? OFFSET ?
```

Mit `SQL QUERY q$, MS_PR% + 1, off%` laufen die beiden `?` als Werte mit. Ein Zeilenumbruch vor der schließenden Klammer sorgt dafür, dass ein `--`-Kommentar am Ende der Abfrage nicht auch die Klammer „auskommentiert“.

Tipp 6: BLOB oder Text – den echten Typ erfragen

MMBasic liefert BLOBs als String, wie Texte. Woran erkennt man also ein BLOB? `.dump` fragt SQLite selbst: `CLI_DumpTab` baut für jede Tabelle eine Abfrage, die zu jeder Spalte zusätzlich ihren Typ liefert:

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

Die Spalten am Ende der Zeile sagen dann für jeden Wert `integer`, `real`, `text`, `blob` oder `null`. `CLI_NumLit$` schreibt damit eine Kommazahl wie 2.0 als `2.0`, damit sie REAL bleibt.

Tipp 7: Dateien mit Tabs lesen

`LINE INPUT #` wandelt Tabs in Leerzeichen um – so will es MMBasic. Für Daten ist das fatal. `CLI_RdLine%` liest Dateien deshalb blockweise mit `INPUT$(255, #7)` und trennt die Zeilen selbst am LF; ein CR davor wird entfernt. Zeilen mit mehr als 255 Zeichen wandern dabei in einen LONGSTRING. `CLI_GetC$` liest für `.import` Zeichen für Zeichen aus demselben Puffer.

Tipp 8: Bildschirm und Terminal gleichzeitig steuern

Die Konsole des CMM2 schreibt auf den Bildschirm **und** an das Terminal. Escape-Folgen für das Terminal würden am Bildschirm als Zeichensalat erscheinen. mmsql löst das so:

- **Cursor bewegen:** `PRINT @(x, y)` setzt die Position auf dem Bildschirm und schickt dem Terminal dieselbe Position (`ED_Up`). Das klappt, weil das Terminalfenster genau so groß ist wie der Bildschirm (`CLI_TermInit`).
- **Nur ans Terminal:** `CLI_TermOnly` schaltet kurz auf `OPTION CONSOLE SERIAL`, schreibt, und schaltet zurück – aber nur beim Start und bei `.cls`. Während getippt wird, wäre das gefährlich: Das Umschalten verdirbt Zeichen, die gerade über die serielle Leitung ankommen.
- **Cursor am Bildschirm:** Während `INKEY$` wartet, zeigt der CMM2 keinen Cursor. mmsql zeichnet das Zeichen unter dem Cursor mit `TEXT` invertiert (`ED_CursorOn`).

Tipp 9: Eingefügten Text erkennen

Ein Mensch tippt nach einem Enter nicht innerhalb von 30 Millisekunden weiter, ein Terminal beim Einfügen schon. `ED_Pasted%` wartet nach einem Enter kurz (`.set paste`) und schaut, ob mit `INKEY$` schon das nächste Zeichen da ist. Das Zeichen wird aufbewahrt und nicht verloren (`ED_Peek$`, `ED_Key$`). Kommen viele Zeichen auf einmal, zeichnet der Editor erst, wenn keine mehr warten (`ED_Flush`) – das macht das Einfügen schnell.

Tipp 10: SQL zerlegen

Wo endet eine Anweisung? Nicht an jedem `;` – nicht in `'Texten'`, `"Namen"`, `[Namen]`, `` `Namen` ``, nicht in `-- Kommentaren` und `/* Kommentaren */`, und in einem Trigger erst bei `END;`. `CLI_Split%` zerlegt eine Zeile, `CLI_Scan` mit seinen Hilfen (`CLI_ScWord`, `CLI_ScNext%`) prüft für `.read` und Blöcke Stück für Stück, ob eine Anweisung fertig ist. Der Zustand (offene Anführungszeichen, Kommentar) bleibt dabei über Zeilen- und Stückgrenzen hinweg erhalten.

Weitere Fundstücke

Routine	Was sie zeigt
`CLI_FwVer$`	die Firmware-Version aus `MM.VER` als Text: `8.000001` → `8.00.00b1` (b1–b50 Beta, RC ab 51, 99 Release)
`CLI_PutB64`	Base64 mit `MATH(BASE64 ENCODE …)` in Stücken von 189 Bytes (durch 3 teilbar: kein Auffüllen in der Mitte)
`CLI_PutLit`	Texte mit Steuerzeichen als SQL: `'a'||char(10)||'b'`
`CLI_HPut`, `CLI_LoadHis`	eine Liste mit Einträgen über mehrere Zeilen in einer Textdatei speichern, abwärtskompatibel
`CLI_ScrSet%`, `CLI_ScrEnd`	`MODE` wechseln und am Ende wiederherstellen – mit Farbtiefe aus `MM.INFO(MODE)` (z. B. 1.8) und `MM.INFO(FONT)`
`CLI_Restore`	eine Datei sicher ersetzen: erst kopieren, Größe prüfen, dann umbenennen
`CLI_Show$`	UTF-8-Zeichen zählen und durch Platzhalter ersetzen
`MS_QId$`	Namen in `"…"` setzen, damit auch Namen mit Leerzeichen gehen

Erinnerung an MMBasic-Eigenheiten

- **Schlüsselwörter sind als Variablennamen verboten**, auch mit Typzeichen: `pos%`, `inv%`, `rnd%` gehen nicht.
- **`AND` und `OR` rechnen bitweise.** `IF a% AND b$ = "" THEN` ist falsch, wenn `a%` zum Beispiel 2 ist (2 AND 1 = 0). Sicher: `IF a% > 0 AND b$ = "" THEN`.
- **Beide Seiten von `AND` werden ausgewertet.** `IF i% <= LEN(s$) AND MID$(s$, i%, 1) = "x"` ruft `MID$` auch dann auf, wenn `i%` schon zu groß ist.
- **Strings haben höchstens 255 Zeichen** – auch Listen und Hilfstexte im Programm. Lange Listen teilt mmsql auf zwei Strings auf (`CLI_IsCmd%`).
- **In einer FUNCTION nie den eigenen Namen lesen** – das ist ein rekursiver Aufruf. Für Zwischenwerte eine lokale Variable nehmen.
- **Eine FUNCTION nicht wie eine SUB aufrufen.** Das gibt erst zur Laufzeit „Unknown command“.
- **`LEFT$(s$, 0)` und `MID$` hinter dem Ende** brauchen Sorgfalt; mmsql hat dafür `CLI_Left$` und `CLI_From$`.

16. Meldungen und Problemlösung

Fehler von SQLite

Fehler von SQLite beginnen mit `Error: SQL:`, danach folgt die Anweisung mit ihrer Länge. Bei langen Anweisungen zeigt mmsql die ersten 200 Zeichen und die tatsächliche Länge.

Meldung	Bedeutung und Abhilfe
`no such table: x`	Die Tabelle gibt es nicht – Tippfehler? `.tables` listet alle.
`no such column: x`	Die Spalte gibt es nicht – `.columns tabelle` zeigt alle.
`near "x": syntax error`	SQL falsch geschrieben, kurz vor oder bei `x`. Oft ein fehlendes Komma, eine fehlende Klammer, ein vergessenes `;` in einem Block – oder ein Punkt-Befehl ohne Punkt.
`UNIQUE constraint failed`	Der Wert ist schon vorhanden, darf aber nur einmal vorkommen (z. B. Primärschlüssel).
`NOT NULL constraint failed`	Ein Pflichtfeld wurde nicht gefüllt.
`FOREIGN KEY constraint failed`	Der Wert fehlt in der verbundenen Tabelle (nur mit `.fk on`).
`file is not a database`	keine SQLite-Datei, verschlüsselt oder am PC im WAL-Modus gespeichert (am PC: `PRAGMA journal_mode=delete;`).
`unable to open database file`	Dateiname, Ordner oder SD-Karte prüfen.
`disk I/O error`	SD-Karte voll oder fehlerhaft.
`attempt to write a readonly database`	Datei oder SD-Karte schreibgeschützt.
`database is locked`	eine Transaktion ist noch offen – `COMMIT` oder `ROLLBACK`.
`cannot commit - no transaction is active`	Es läuft keine Transaktion – vielleicht wurde sie nach einem Fehler oder Ctrl-C schon beendet.
`stopped by Ctrl-C`	abgebrochen, siehe Kapitel 10.

Meldungen von mmsql

Meldung	Bedeutung
`Did you mean .xxx? Commands start with a dot.`	Punkt vergessen.
`Hint: commands start with a dot`	Der Syntaxfehler liegt beim ersten Wort – vermutlich ein Befehl ohne Punkt.
`Hint: .tables lists the tables`	nach `no such table`.
`Not run: n more statement(s) of this line.`	Nach einem Fehler wurde der Rest der Zeile nicht ausgeführt.
`Not run: the rest of the block.`	Nach einem Fehler wurde der Rest des Blocks nicht ausgeführt.
`(No database open - using a database in memory. …)`	Es war keine Datei offen; mmsql nimmt `:memory:` (Kapitel 9).
`No database open (.open file.db)`	Der Befehl braucht eine offene Datenbank.
`Line n: …`	`.read`: Fehler in der Anweisung, die in Zeile n beginnt.
`Line n: .xxx is not run by .read`	Dieser Punkt-Befehl läuft nicht in Skripten (Kapitel 9).
`Record n: … - skipped`	`.import`: Datensatz mit falscher Feldzahl oder zu lang.
`Too many errors - stopped.` / `- nothing imported.`	`.read` bzw. `.import` nach zehn Fehlern abgebrochen; `.import` hat dann nichts übernommen.
`Note: SQLite has undone the open transaction.`	Nach einem Fehler hat SQLite die Transaktion beendet.
`A transaction is still open: COMMIT or ROLLBACK`	nach `.read`: Das Skript hat eine Transaktion offen gelassen.
`Stopped: a text of n bytes … is longer than maxtext`	Export abgebrochen – `.set maxtext n` und noch einmal (Kapitel 8).
`(discarded)`	Eingabe mit Esc verworfen.

Wenn etwas nicht klappt

Problem	Lösung
Beim Einfügen vom PC läuft jede Zeile einzeln los	Paste-Zeit zu kurz: `.set paste 250` bei 200 ms Verzögerung im Terminal – oder vor dem Einfügen F3 drücken (Kapitel 4).
Beim Einfügen fehlen Zeichen	im Terminal eine kleine Verzögerung pro Zeichen einstellen.
Ein eingefügter Block gibt einen Syntaxfehler	Fehlt am Ende einer Anweisung das `;`? Im Block gehört die nächste Zeile sonst noch dazu.
Ein altes Skript läuft nicht mehr	seit 1.2 braucht `.read` ein `;` am Ende jeder Anweisung (Kapitel 9).
Umlaute erscheinen als `·` (im Terminal `ú`)	gewollt – die Schrift des CMM2 hat keine Umlaute (Kapitel 4).
Spalten fehlen rechts	`(+n column(s) not shown)`: breiteren Bildschirm-Modus wählen (`.set screen`), `.width` verkleinern oder `.mode line`.
Terminal und Bildschirm zeigen Unterschiedliches	`.cls` stellt das Terminalfenster wieder auf die Bildschirmgröße.
Enter wirkt doppelt	Das passiert nur außerhalb von mmsql; mmsql setzt beim Start `OPTION CRLF CR`.
Ctrl-C bricht SQL nicht ab	`OPTION BREAK 0` ist gesetzt (mmsql weist beim Start darauf hin). Mit `OPTION BREAK 4` übernimmt Ctrl-D die Rolle von Ctrl-C.
Eine Datenbank vom PC lässt sich nicht öffnen	am PC `PRAGMA journal_mode=delete;` und neu speichern.

Hinweise zur Firmware

OPTION CRLF: Ohne `OPTION PICOMITE` schickt die Enter-Taste in Firmware 8 CR und LF. mmsql setzt beim Start `OPTION CRLF CR`; das gilt nur, solange mmsql läuft.

LGETSTR$ in 8.00.00b1: schreibt bei genau 255 Zeichen ein Byte zu viel (Kapitel 15); in 8.00.00b2 behoben. mmsql umgeht den Fehler auf jeder Firmware.

LINE INPUT # und Tabs: `LINE INPUT #` wandelt Tabs in Leerzeichen; mmsql liest Dateien deshalb anders (Kapitel 15).

MM.DEVICE$ unter OPTION PICOMITE: liefert dann den PicoMite-Namen; die Generation 3 lässt sich so nicht erkennen, mmsql zeigt „G2/G3“.

Einen Fehler melden

Fehler und Wünsche bitte im Forum *TheBackShed* im Thema zu mmsql melden. Dazu gehören:

- die drei Zeilen von `.version`,
- was eingegeben wurde und was erschien,
- wenn möglich eine kleine Datenbank oder ein Skript, mit dem es sich wiederholen lässt.

Anhang A: Alle Befehle auf einen Blick

`[ ]` = darf fehlen, `a|b` = a oder b. Kurzformen in Klammern. Dateinamen ohne Laufwerk und Ordner beziehen sich auf den aktuellen Ordner von MMBasic.

Befehl	Wirkung	Kapitel
`.alter tabelle`	Spalte hinzufügen, umbenennen, löschen; Tabelle umbenennen	7
`.backup datei.db`	saubere Kopie der Datenbank schreiben	10
`.bail on|off`	`.read` beim ersten Fehler stoppen	9
`.browse tabelle [WHERE …] [ORDER BY …]` (`.b`)	Zeilen seitenweise zeigen	6
`.check [full]`	Datenbank prüfen	10
`.close`	Datenbank schließen	10
`.cls`	Bildschirm und Terminal löschen	11
`.columns tabelle` (`.c`)	Spalten einer Tabelle	6
`.create tabelle`	Tabelle Schritt für Schritt anlegen	7
`.databases` (`.db`)	offene und angehängte Datenbanken	6
`.delete tabelle schlüssel`	eine Zeile löschen	7
`.dump [tabelle]`	Datenbank als SQL-Text	9
`.echo on|off`	`.read` zeigt jede Anweisung	9
`.edit tabelle schlüssel`	eine Zeile ändern	7
`.files [muster]`	Dateien mit Größe und Datum	10
`.fk [on|off]`	Fremdschlüssel-Prüfung zeigen oder schalten	10
`.fkcheck [tabelle]`	Verletzungen von Fremdschlüsseln finden	10
`.headers on|off`	Kopfzeile bei csv und list	8
`.help [befehl]` (`.h`)	Hilfe	2
`.import [--skip N] datei.csv tabelle`	CSV-Datei importieren	9
`.indexes [tabelle]` (`.ix`, `.indices`)	Indizes mit ihren Spalten	6
`.info`	Steckbrief der Datenbank	6
`.insert tabelle`	eine Zeile einfügen	7
`.kill datei`	eine Datei löschen	10
`.mode table|csv|list|insert [tabelle]|line|markdown|json`	Ausgabemodus	8
`.nullvalue text`	Anzeige von NULL	8
`.once datei`	nächste Abfrage in eine Datei	8
`.open datei.db|:memory:` (`.o`)	Datenbank öffnen oder anlegen	3, 9
`.output [datei]`	alle Abfragen in eine Datei, ohne Datei zurück	8

`.print text`	eine Zeile ausgeben	8, 9
`.quit` (`.q`, `.exit`)	mmsql beenden	2
`.read datei.sql`	SQL-Skript ausführen	9
`.reindex [name]`	Indizes neu aufbauen	10
`.restore datei.db`	Datenbank durch eine Sicherung ersetzen	10
`.schema [name|muster]` (`.s`)	Bauplan der Objekte	6
`.separator x`	Trennzeichen für csv und list	8
`.set [name wert]`	Einstellungen zeigen oder ändern	11
`.tables [muster]` (`.t`)	Tabellen und Views	6
`.timer on|off`	Laufzeit zeigen	11
`.triggers [tabelle]`	Trigger und ihre Tabellen	6
`.vacuum`	Datei verkleinern	10
`.version` (`.ver`)	Versionen von mmsql, SQLite, Firmware	11
`.width [n1 n2 …]`	Spaltenbreiten in `table`	8

Die Hilfe am CMM2 ordnet die Befehle nach Gruppen:

Gruppe	**Befehle**
Datenbank	`.open` `.close` `.info` `.backup` `.restore` `.vacuum` `.files` `.kill`
Ansehen	`.tables` `.indexes` `.triggers` `.schema` `.columns` `.browse` `.databases`
Daten	`.insert` `.edit` `.delete` `.import` `.read`
Struktur	`.create` `.alter` `.reindex`
Prüfen	`.check` `.fkcheck` `.fk`
Ausgabe	`.mode` `.headers` `.separator` `.nullvalue` `.width` `.once` `.output` `.dump`
Optionen	`.set` `.timer` `.bail` `.echo` `.print` `.cls` `.version` `.help` `.quit`

Anhang B: Tasten

Am Prompt (Editor)

Taste	eine Zeile	mehrere Zeilen
Enter	ausführen (mit `.set enter line`: neue Zeile)	neue Zeile / Zeile teilen
F3	neue Zeile / teilen; am leeren Prompt: mehrzeilig beginnen	neue Zeile / teilen
F2	ausführen	ausführen
Esc	verwerfen	verwerfen
← →	Cursor	Cursor, auch über Zeilenenden
Pos1 / Ende	Anfang / Ende der Zeile	Anfang / Ende der Bildschirmzeile
↑ ↓	Verlauf	eine Bildschirmzeile; am Rand: Verlauf
Bild↑ / Bild↓	Verlauf	Verlauf
Backspace	links löschen	am Zeilenanfang: an die vorige Zeile hängen
Entf	am Cursor löschen	am Zeilenende: nächste Zeile heranholen

Beim Blättern: Enter oder N = weiter, B = zurück, Q oder Esc = aufhören.

Bei Rückfragen: y = ja, jede andere Taste = nein. Bei offener Transaktion: C = commit, R = rollback, Esc = abbrechen.

Ctrl-C: bricht eine laufende SQL-Anweisung ab (Kapitel 10).

Anhang C: Optionen

Alle Optionen werden mit `.set name wert` geändert und sofort in `mmsql.cfg` gespeichert.

Option	Standard	erlaubt	Bedeutung
`mode`	table	table, csv, list, insert, line, markdown, json	Ausgabemodus
`headers`	1	on/off bzw. 1/0	Kopfzeile bei csv und list
`separator`	`,`	1 bis 3 Zeichen, `\t` oder `tab`	Trennzeichen für csv und list
`nullvalue`	(leer)	bis 20 Zeichen, `''` = leer	Anzeige von NULL
`blob`	hex	hex, base64	BLOBs in `.mode json`
`width`	Bildschirmbreite	40 bis 250, 0 = folgt dem Bildschirm	Ausgabebreite
`rows`	20	5 bis 100	Zeilen pro Seite (nach Neustart)
`enter`	run	run, line	was Enter auf einer Zeile tut
`paste`	30	0 bis 1000 (ms), 0 = aus	Erkennung eingefügter Zeilen
`history`	50	0 bis 200	Einträge im Verlauf (nach Neustart)
`utf8`	dot	dot, raw	Anzeige von UTF-8-Zeichen
`dotchar`	250	Code 33 bis 255 oder ein Zeichen	Platzhalter für `utf8 dot`
`maxtext`	32768	1024 bis 1048576	längster Text bzw. BLOB in Bytes
`screen`	0	0 bis 19	MODE beim Start, 0 = wie er ist
`timer`	0	0, 1	Laufzeit anzeigen
`helpfile`	mmsql_en.hlp	Dateiname	Hilfedatei, z. B. mmsql_de.hlp

Anhang D: SQL-Spickzettel

Abfragen

```
SELECT spalte, spalte FROM tabelle WHERE bedingung ORDER BY spalte [DESC] LIMIT n;
SELECT DISTINCT spalte FROM tabelle;
SELECT spalte, count(*) FROM tabelle GROUP BY spalte 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);
```

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

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

Ändern

```
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;
```

Aufbauen

```
CREATE TABLE t(id INTEGER PRIMARY KEY, name TEXT NOT NULL, n INTEGER DEFAULT 0);
ALTER TABLE t ADD COLUMN notiz 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;
```

Transaktionen: `BEGIN` … `COMMIT` oder `ROLLBACK`

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

Anhang E: Änderungen gegenüber 1.1

Neu

- Editor über bis zu 20 Zeilen mit Statuszeile (F3 neue Zeile, F2 ausführen), Optionen `enter` und `paste`, Erkennung eingefügter Zeilen.
- Eine Regel für die Eingabe: eine Zeile läuft mit Enter, mehrere Zeilen sind ein Block mit `;` am Ende jeder Anweisung.
- Anweisungen und Abfragen über 255 Zeichen, mit Ergebnis.
- Verlauf beliebig langer Anweisungen, auch nach Neustart; Bild↑/Bild↓.
- `.read` mit Anweisungen über mehrere Zeilen, Abfrage-Ergebnissen und Punkt-Befehlen.
- Datenbank im Speicher (`:memory:`).
- `.mode markdown`, `.mode json`, `.set blob`.
- `.width`, `.print`, `.version`, `.restore`, `.set screen`, `.set width 0`.
- Suchmuster bei `.tables`, `.schema`, `.indexes`, `.triggers`.
- `.import` ohne Grenze bei Spalten und langen Feldern; `.insert` bis 64 Spalten.
- `.dump` mit dem echten Typ jedes Wertes.
- Kürzerer Prompt: `[retro.db]>` statt `mmsql[retro.db]>`.
- Terminal und Bildschirm zeigen 1:1 dasselbe.
- `.set` nach Themen geordnet.

Geändert

- **`.read` braucht `;` am Ende jeder Anweisung** (wie sqlite3). Skripte ohne `;` laufen nicht mehr richtig.
- Fehlermeldungen zeigen die Anweisung so, wie sie getippt wurde, mehrzeilige Zeile für Zeile.
- Der Hinweis „commands start with a dot“ erscheint nur noch, wenn der Fehler beim ersten Wort liegt.
- `--`-Kommentare enden am Zeilenende.

Behoben

- Ein SELECT mit mehr als 239 Zeichen konnte mmsql mit einem Fehler beenden.
- `.schema` zeigt Anweisungen über mehrere Zeilen ohne „Treppe“ im Terminal.
- `.dump` schreibt eine Kommazahl wie 2.0 als `2.0` (sie bleibt REAL) und „unendlich“ als `1e999` (SQLite kann es wieder lesen).
- Sehr große Ganzzahlen (über 2^{53}) werden genau angezeigt und exportiert, nicht mehr gerundet als Kommazahl.

Anhang F: Kleines Lexikon

Abfrage – eine Frage an die Datenbank, in SQL ein `SELECT`. Die Antwort ist eine Tabelle aus Zeilen und Spalten (Kapitel 5.1).

Aggregatfunktion – fasst viele Zeilen zu einem Wert zusammen: `count()`, `sum()`, `avg()`, `min()`, `max()`. Mit `GROUP BY` für jede Gruppe einzeln (Kapitel 5.3).

Alias – ein Ersatzname in einer Abfrage, mit `AS` oder einfach dahinter geschrieben: `count(*) AS anzahl`, `FROM computer c` (Kapitel 5.4).

ATTACH – hängt eine weitere Datenbank-Datei an die offene an; ihre Tabellen heißen dann `name.tabelle` (Kapitel 6).

BLOB – *Binary Large Object*: Binärdaten wie Bilder oder Programmcode, im Gegensatz zu Text. mmsql zeigt sie in Exporten als Hex (`X'0A1B'`) oder Base64.

Block – eine Eingabe aus mehreren Zeilen, getippt mit F3 oder eingefügt. Jede Anweisung darin endet mit `;` (Kapitel 4).

CSV – *comma-separated values*: eine Textdatei mit einer Tabellenzeile pro Zeile, die Werte durch Kommas getrennt. Tabellenkalkulationen lesen und schreiben das Format (Kapitel 8 und 9).

Datenbank – eine Datei, in der Daten geordnet in Tabellen liegen. Bei SQLite ist eine Datenbank genau eine Datei (Kapitel 1).

Datensatz – eine Zeile einer Tabelle, zum Beispiel ein Computer.

Dump – die ganze Datenbank als SQL-Text: alle `CREATE`- und `INSERT`-Anweisungen, mit denen sie sich neu aufbauen lässt (Kapitel 9).

Fremdschlüssel – eine Spalte, deren Wert in einer anderen Tabelle vorkommen muss, angelegt mit `REFERENCES`. SQLite prüft das nur nach `.fk on` (Kapitel 10).

Index – ein Verzeichnis zu einer Spalte, mit dem SQLite schneller sucht, wie das Stichwortverzeichnis eines Buches (Kapitel 5.8).

JOIN – verbindet zwei Tabellen in einer Abfrage über eine Bedingung. `LEFT JOIN` behält auch die Zeilen ohne Partner (Kapitel 5.4).

JSON – ein Textformat für Daten, das Webseiten und viele Programme verstehen (Kapitel 8).

Journal – eine Hilfsdatei, mit der SQLite Änderungen absichert: Bricht etwas ab, wird mit ihrer Hilfe der alte Stand wiederhergestellt. Auf dem CMM2 heißt sie `datei.db-journal` und ist nur während einer Änderung da (Kapitel 10).

LONGSTRING – in MMBasic ein Speicherbereich (ein INTEGER-Array) für Texte über 255 Zeichen. mmsql nutzt LONGSTRINGs für lange Anweisungen und Werte (Kapitel 15, Tipp 4).

Markdown – eine einfache Textschreibweise für Überschriften, Listen und Tabellen, die Foren und GitHub schön darstellen (Kapitel 8).

maxtext – Option von mmsql: der längste Text, den mmsql auf einmal liest oder schreibt, Standard 32 768 Zeichen (Kapitel 11).

:memory: – eine Datenbank, die nur im Arbeitsspeicher liegt und beim Schließen verschwindet (Kapitel 9).

NULL – „kein Wert“. Nicht dasselbe wie 0 oder ein leerer Text; abgefragt mit `IS NULL` (Kapitel 1 und 5.1).

PRAGMA – ein SQLite-Befehl, der Einstellungen der Datenbank zeigt oder ändert, zum Beispiel `PRAGMA foreign_keys = ON;`. Viele Punkt-Befehle von mmsql benutzen intern PRAGMAs.

Primärschlüssel – die Spalte, die jede Zeile eindeutig kennzeichnet, meist eine Nummer `id` (Kapitel 1 und 5.6).

Prompt – die Eingabeaufforderung, bei mmsql `mmsql>` oder `[datei.db]>`. Ein Stern `*` darin zeigt eine offene Transaktion.

Punkt-Befehl – ein Befehl an mmsql selbst, nicht an die Datenbank. Er beginnt mit einem Punkt: `.tables`.

rowid – die interne Zeilennummer, die jede SQLite-Tabelle hat. Eine Spalte `INTEGER PRIMARY KEY` ist diese Nummer (Kapitel 7).

Schema – der Bauplan einer Datenbank: welche Tabellen mit welchen Spalten, Indizes, Views und Trigger es gibt. `.schema` zeigt ihn (Kapitel 6).

Skript – eine Textdatei mit SQL-Anweisungen, ausgeführt mit `.read` (Kapitel 9).

SQL – *Structured Query Language*, die Sprache, in der man mit Datenbanken spricht (Kapitel 5).

SQLite – die Datenbank, die in der Firmware des CMM2 steckt: eine Bibliothek ohne Server, eine Datenbank pro Datei (Kapitel 1).

Tabelle – Daten in Zeilen und Spalten, wie ein Blatt einer Tabellenkalkulation.

Terminal – ein Programm am PC (z. B. Tera Term), das über das serielle Kabel mit dem CMM2 verbunden ist und dasselbe zeigt wie der Bildschirm (Kapitel 2).

Transaktion – mehrere Änderungen, die nur zusammen gelten: `BEGIN` … `COMMIT` oder `ROLLBACK` (Kapitel 5.7).

Trigger – eine Regel, die bei einer Änderung an einer Tabelle von selbst weitere Anweisungen ausführt (Kapitel 5.8).

Typ – die Art eines Wertes: ganze Zahl (`INTEGER`), Kommazahl (`REAL`), Text (`TEXT`), Binärdaten (`BLOB`) oder `NULL`. In SQLite eine Empfehlung, keine Vorschrift (Kapitel 5.6).

UTF-8 – die übliche Art, Texte mit Umlauten und anderen Sonderzeichen zu speichern. Die Schrift des CMM2 kennt diese Zeichen nicht; mmsql zeigt einen Platzhalter (Kapitel 4).

VACUUM – baut die Datenbank-Datei neu auf und gibt den Platz gelöschter Zeilen frei (`.vacuum`, Kapitel 10).

View – eine gespeicherte Abfrage, die sich wie eine Tabelle benutzen lässt (Kapitel 5.8).

WAL – *Write-Ahead Logging*, ein Journal-Modus von SQLite, bei dem Änderungen erst in eine Zusatzdatei `…-wal` gehen. Das SQLite des CMM2 kann ihn nicht; Datenbanken vom PC vorher mit `PRAGMA journal_mode=delete;` umstellen (Kapitel 5.12).

XMODEM – ein Verfahren, Dateien über das serielle Kabel zwischen PC und CMM2 zu übertragen (Kapitel 2).

Dank

An Peter Mather für SQLite auf dem CMM2 und für schnelle Antworten und Korrekturen, an Geoff Graham für MMBasic und an die Community auf TheBackShed.