# 5 SQLite

## 5.1 Relationales Datenbank Managment System

### RDBMS

- Für die praktische Arbeit mit den Daten muss das *relationale Datenmodell* als *relationale Datenbank* implementiert werden.
- Ein Datenbankprogramm, das nicht nur das Speichern und Abfragen sondern auch das Verwalten der Daten und der Benutzer ermöglicht, wird *Relationales Datenbank Management System (RDBMS)* genannt.

### Aufgaben eines RDBMS

- Datenbanken anlegen
- Tabellen erstellen und löschen
- Datensätze in Tabellen einfügen
- Datensätze aus Tabellen löschen
- Datenbankabfragen tätigen
- DB-Benutzer verwalten (Passwörter, Berechtigungen, ...)
- Transaktionsüberwachung ($\rightarrow$ ACID)

## 5.2 Structured Query Language

In der Praxis existieren viele verschiedene RDBMS. Ihre Anwendungsschnittstelle ist in der Regel die Datenbanksprache *Structured Query Language (SQL)*.

Auch wenn der SQL-Sprachkern standardisiert ist, gibt es je nach Datenbank(-Hersteller) verschiedene SQL-Dialekte:

- IBM DB2 (kommerziell)
- Oracle Database (kommerziell und eingeschränkte Version für kostenlose Nutzung)
- Microsoft SQL Server (kommerziell)
- MySQL (Open Source und kommerziell)
- PostgreSQL (Open Source)
- SQLite (Open Source)
- ...

## 5.3 SQLite

SQLite ist ein kostenloses leichtgewichtiges RDBMS, das im Gegensatz zu seinen grossen Geschwistern

- keinen Server benötigt,
- ohne Konfiguration auskommt,
- auf vielen Plattformen verfügbar ist,
- in einer einzelnen Datei integriert ist,
- sparsam mit Speichplatz umgeht.

Dennoch unterstützt SQLite einen grossen Teil des SQL92-Standards und ist sehr zuverlässig.

### 5.3.1 Information und Download

`http://www.sqlite.org/`

### 5.3.2 GUI

Für SQLite gibt es auch grafische Benutzerschnittstellen (GUI), die die Arbeit mit dem RDBMS vereinfachen.

Wir werden dafür das Firefox Add-On *SQLite Manager* verwenden. Dieses Programm verwendet den Browser als GUI und erlaubt das Speichern der Datenbank im lokalen Dateisystem.

### 5.3.3 Datentypen

Jeder Wert, der in einer SQLite-DB gespeichert oder von ihr verändert wird, gehört zu einer der folgenden Datentypen:

- **NULL** der NULL-Wert ( $\neq 0$)
- **INTEGER** eine ganze Zahl mit Vorzeichen (je nach Grössenordnung 1, 2, 3, 4, 6, oder 8 Bytes)
- **REAL** Fliesskommazahl (8 Byte, IEEE 754-Format)
- **TEXT** Zeichenkette (je nach DB-Kodierung UTF-8, UTF-16BE oder UTF-16LE)
- **BLOB** (Binary Large Object) exakte Kopie des Inputs

### 5.3.4 Datum und Zeit

Datum- und Zeitangaben werden am einfachsten als `TEXT` gespeichert. Es sind auch Darstellungen als `REAL` möglich, die jedoch Umrechnungen erfordern.

Hier einige Darstellungen, die von SQLite erkannt werden:

- *YYYY-MM-DD*
- *YYYY-MM-DD HH:MM*
- *YYYY-MM-DD HH:MM:SS*
- *YYYY-MM-DD HH:MM:SS.sss*
- *HH:MM*
- *HH:MM:SS*
- *HH:MM:SS.sss*
- ...

### 5.3.5 CREATE TABLE

#### Beispiel 1

```
CREATE TABLE personal (
   pid       INTEGER PRIMARY KEY AUTOINCREMENT,
   nachname  TEXT NOT NULL,
   vorname   TEXT CHECK(vorname != 'Konstanze'),
   eintritt  TEXT NOT NULL DEFAULT CURRENT_DATE,
   garderobe INTEGER UNIQUE
);
```

#### Beispiel 2

```
CREATE TABLE wohnungen (
   wohnungsnummer INTEGER NOT NULL,
   gebaeudenummer INTEGER NOT NULL,
   PRIMARY KEY (wohnungsnummer, gebaeudenummer)
);
```

### 5.3.6 DROP TABLE

DROP TABLE [IF EXISTS] *db_name.table_name*;

#### Beispiel

```
DROP TABLE IF EXISTS personal;
```

### 5.3.7 INSERT INTO

```
INSERT INTO db_name.tab_name (col1, col2, ...)
VALUES (val1, val2, ...)
```

#### Beispiel 1

```
INSERT INTO personal (pid, nachname, vorname, garderobe)
VALUES (11, 'Meier', 'Andreas', '04-06-2009', 17);
```

#### Beispiel 1

```
INSERT INTO personal (eintritt, nachname, garderobe, vorname)
VALUES ('25-11-2018', 'Bernasconi', 39, 'Maria');
```

### 5.3.8 ALTER TABLE

Übliche Verwendung:

```
ALTER TABLE db_name.table_name
RENAME TO new_table_name
```

und

```
ALTER TABLE db_name.table_name
ADD COLUMN column_def
```

### 5.3.9 DELETE

```
DELETE FROM db_name.table_name;
```

oder

```
DELETE FROM db_name.table_name
WHERE condition;
```

### 5.3.10 ACID-Transaktionen

- **A**tomic: *Die Transaktion darf nicht in kleinere Teile zerlegt werden.*
- **C**onsistent: *Die Transaktion muss die DB in einem konsistenten (widerspruchsfreien) Zustand belassen.*
- **I**solated: *Die Transaktion muss von denen anderer Clients abgeschirmt werden.*
- **D**urable: *Nach dem erfolgreichen Abschluss der Transaktion muss diese ein permanenter und unwiderruflichr Teil der DB werden.*

#### BEGIN/COMMIT TRANSACTION

Normalerweise befindet sich SQLite im *autocommit*-Modus: jede Eingabe wird implizit als einzelne Transaktion verarbeitet.

Problem: *Verlangsamt die Ausführung von SQL-Befehlen*

```
BEGIN [DEFERRED|IMMEDIATE|EXCLUSIVE] [TRANSACTION]
DB-Befehle
COMMIT [TRANSACTION] oder END [TRANSACTION]
```

### 5.3.11 Die SELECT-Pipeline

#### Überblick

```
SELECT [DISTINCT] select_heading
FROM source_tables
WHERE filter_expression
GROUP BY grouping_expressions
HAVING filter_expression
ORDER BY ordering_expressions
LIMIT count
OFFSET count
```

#### Auswertungsreihenfolge

1. `FROM` *source_tables*

   Legt eine oder mehrere Ausgangstabellen fest und kombiniert sie zu einer grossen Arbeitstabelle.

2. `WHERE` *filter_expressions*

   Filtert bestimmte Zeilen aus der Arbeittstabelle heraus.

3. `GROUP BY` *grouping_expressions*

   Fasst Zeilen zusammen, die in bestimmten Werten übereinstimmen.

4. `SELECT` *select_heading*

   Definiert die Kolonnen der Resultatmenge sowie die gruppierenden Funktionen (Aggregatsfunktionen), sofern diese anwendbar sind.

5. `HAVING` *filter_expression*

   Filtert bestimmte Zeilen aus der gruppierten Tabelle. Verlangt ein `GROUP BY`.

6. `DISTINCT`

   Eliminiert identische Zeilen.

7. `ORDER BY` *ordering_expressions*

   Sortiert die Zeilen der Resultatmenge.

8. `OFFSET` *count*

   Überspringt *count* Zeilen am anfang der Resultatmenge. Verlangt ein `LIMIT`.

9. `LIMIT` *`count`*

   Begrenzt die Anzahl der auszugebenden Zeilen der Resultatmenge auf *`count`*.

### 5.3.12 Der FROM-Abschnitt

Für die Kombination von Ausgangstabellen gibt es folgende Möglichkeiten:

#### CROSS JOIN

Ein ungefiltertes Kreuzprodukt der Tabellen. *Achtung!*

| a | b |
|---|---|
| 1 | 2 |
| 3 | 7 |
| 4 | 5 |

$\times$

| c | d |
|---|---|
| 2 | 7 |
| 1 | 2 |

$=$

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 2 | 7 |
| 1 | 2 | 1 | 2 |
| 3 | 7 | 2 | 7 |
| 3 | 7 | 1 | 2 |
| 4 | 5 | 2 | 7 |
| 4 | 5 | 1 | 2 |

```
SELECT * FROM tab1 CROSS JOIN tab2;
```

#### [INNER] JOIN

Ein Kreuzprodukt, das durch eine Bedingung gefiltert wird, die nach dem Schlüsselowrt `ON` steht.

| a | b |
|---|---|
| 1 | 2 |
| 3 | 7 |
| 4 | 5 |

$\bowtie_{b=c}$

| c | d |
|---|---|
| 2 | 7 |
| 1 | 2 |

$=$

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 2 | 7 |

```
SELECT * FROM tab1 INNER JOIN tab2 ON b=c;
```

Da ein `INNER JOIN` häufig verwendet wird, kann man dafür die Abkürzung `JOIN` verwenden.

#### NATURAL JOIN

Filtert im Kreuzprodukt automatisch nach Kolonnen mit identischen Kolonnenamen. Dies ist bequem, kann aber gefährlich werden, wenn man seine Kolonnen unsorgfältig benennt.

| a | b |
|---|---|
| 1 | 2 |
| 3 | 7 |
| 4 | 5 |

$\bowtie$

| c | b |
|---|---|
| 2 | 7 |
| 1 | 2 |

$=$

| a | b | c |
|---|---|---|
| 1 | 2 | 1 |
| 3 | 7 | 2 |

```
SELECT * FROM tab1 NATURAL JOIN tab2;
```

Ein `NATURAL JOIN` führt Kolonnen mit gemeinsamen Merkmalsnamen nur einmal auf.

#### LEFT OUTER JOIN

Ergänzt einen `INNER JOIN` um diejenigen Zeilen in der ersten Tabelle, zu denen es keine

passenden Zeilen in der zweiten Tabelle gibt und füllt die Kollonnen mit NULL-Werten auf.

| a | b |
|---|---|
| 1 | 2 |
| 3 | 7 |
| 4 | 5 |

$\rtimes_{b=d}$

| c | d |
|---|---|
| 2 | 7 |
| 1 | 2 |

=

| a | b | c | d |
|---|---|---|---|
| 1 | 2 | 1 | 2 |
| 3 | 7 | 2 | 7 |
| 4 | 5 | NULL | NULL |

```
SELECT * FROM tab1 LEFT OUTER JOIN tab2 on b=d;
```

### 5.3.13 Der WHERE-Abschnitt

Nach `WHERE` kann man eine (oder mehrere) Bedingung(en) angeben, nach denen man die Resultattabelle filtert (*Selektion*). Beispielsweise mit den Vergleichsoperatoren (`=`, `!=`, `<`, `<=`, `>`, `>=`, `BETWEEN ... AND ...`), den logischen Operatoren (`AND`, `OR`, `NOT`) oder mit `LIKE`.

```
SELECT a, c FROM tab WHERE b >= 5;
```

| a | b | c |
|---|---|---|
| 3 | 5 | 7 |
| 9 | 3 | 1 |
| 2 | 6 | 4 |

⇒

| a | c |
|---|---|
| 3 | 7 |
| 2 | 4 |

### 5.3.14 Der GROUP BY-Abschnitt

`GROUP BY` fasst Zeilen zusammen, die in bestimmten Werten übereinstimmen.

Üblicherweise wendet man auf die gruppierten Zeilen im SELECT-Teil eine der Aggregatsfunktionen `count(...)`, `sum(...)`, `min(...)`, `max(...)` oder `avg(...)` auf eines der Attribute an.

```
SELECT a, b, sum(c) FROM tab GROUP BY a;
```

| a | b | c |
|---|---|---|
| 3 | 12 | 7 |
| 1 | 27 | 5 |
| 3 | 30 | 4 |
| 2 | 14 | 9 |
| 2 | 19 | 8 |
| 1 | 32 | 0 |

⇒

| a | b | c |
|---|---|---|
| 3 | 12 | 7 |
| 3 | 30 | 4 |
| 1 | 27 | 5 |
| 1 | 32 | 0 |
| 2 | 14 | 9 |
| 2 | 19 | 8 |

⇒

| a | b | c |
|---|---|---|
| 1 | 32 | 5 |
| 2 | 19 | 17 |
| 3 | 30 | 11 |

Die mittlere Tabelle stellt einen Zwischenschritt in der Berechnung dar und wird nicht ausgegeben.

**Achtung:** Einträge, die weder von der Zusammenfassung noch von der Aggregatsfunktion betroffen sind (hier: Attribut *b*), erhalten jeweils den Eintrag der letzten Zeile in der Gruppe. Sie sind von der Reihenfolge der Daten abhängig und damit nicht vorhersehbar.

### 5.3.15 Die SELECT-Kopfzeile

Die SELECT-Kopfzeile dient dazu, das Format und den Inhalt der Resultattabelle zu definieren.

`SELECT` *`expression`* `[AS` *`column_name`*`] [,...]`

Mit dem Optionalen Argument `AS` kann ein Alias definiert werden, der dann in Resultat-

tabelle erscheint. Wenn der Aliasname Leerzeichen enthält, so muss er von Anführungs- und Schlusszeichen eingeschlossen sein.

Als *expression* kann auch ein Stern (*) verwendet werden, der jede Kolonne im Resultat anzeigt.

### 5.3.16 Der HAVING-Abschnitt

Die HAVING-Klausel ist im Grunde identisch mit der WHERE-Klausel.

Der Unterschied besteht darin, dass die WHERE-Klausel *vor* einem GROUP BY und die HAVING-Klausel nach einem GROUP BY ausgewertet wird.

Deshalb sollte man die HAVING-Klausel nur als Filterinstrument nach einem GROUP BY verwenden.

#### Beispiel

```
SELECT a, sum(c) AS summe FROM tab
GROUP BY a
HAVING summe > 10;
```

| a | b | c |
|---|---|---|
| 3 | 12 | 7 |
| 1 | 27 | 5 |
| 3 | 30 | 4 |
| 2 | 14 | 9 |
| 2 | 19 | 8 |
| 1 | 32 | 0 |

⇒

| a | summe |
|---|---|
| 3 | 11 |
| 1 | 5 |
| 2 | 17 |

⇒

| a | summe |
|---|---|
| 2 | 17 |
| 3 | 11 |

### 5.3.17 Das DISTINCT-Schlüsselwort

Mit dem Schlüsselwort DISTINCT werden in der Resultatabelle identischen Zeilen allfällige duplizierte Zeilen entfernt.

*Beachte:* Da SQLite alle Zeilen paarweise auf Identität prüfen muss, ist es eine *teure* Operation, die bei grossen Tabellen einen grossen Zeitaufwand verursachen kann.

#### Beispiel

```
SELECT DISTINCT a from tab;
```

| a | b | c |
|---|---|---|
| 3 | 6 | 9 |
| 1 | 4 | 8 |
| 3 | 5 | 7 |
| 2 | 4 | 9 |
| 2 | 6 | 7 |
| 1 | 5 | 8 |

⇒

| a |
|---|
| 3 |
| 1 |
| 2 |

### 5.3.18 Der ORDER BY-Abschnitt

Die `ORDER BY`-Klausel dient dazu, die Zeilen der Resultattabelle zu sortieren bzw. zu ordnen.

`ORDER BY` *`expression`*

`[COLLATE` *`collation_name`*`] [ASC|DESC] [,...]`

`COLLATE` bezeichnet die vorgegebene Ordnung in der die Symbole des Zeichensatzes verglichen werden. Beispielsweise werden mit `COLLATE NOCASE` alle Grossbuchstaben in Kleinbuchstaben verwandelt. [kein Prüfungsstoff]

Ohne Angabe einer Sortierreihenfolge wird standardmässig `ASC` (aufsteigend) verwendet.

**Beispiel**

```
SELECT * FROM tab
ORDER BY a ASC, b DESC;
```

| a | b | c |
|---|---|---|
| 3 | 6 | 9 |
| 1 | 4 | 8 |
| 3 | 5 | 7 |
| 2 | 4 | 9 |
| 2 | 6 | 7 |
| 1 | 5 | 8 |

⇒

| a | b | c |
|---|---|---|
| 1 | 5 | 8 |
| 1 | 4 | 8 |
| 2 | 6 | 7 |
| 2 | 4 | 9 |
| 3 | 6 | 9 |
| 3 | 5 | 7 |

### 5.3.19 Die LIMIT- und OFFSET-Abschnitte

`LIMIT` und `OFFSET` ermöglichen eine spezielle Teilmenge der Resultattabelle anzuzeigen.

`LIMIT` beschränkt die Anzahl der ausgegebenen Zeilen.

`OFFSET` sagt, wie viele Zeilen vor der ersten Ausgabe *weggelassen* werden sollen.

**Beispiele:**

| | |
|---|---|
| `LIMIT 10` | gibt die Zeilen 1–10 aus |
| `LIMIT 10 OFFSET 3` | gibt die Zeilen 4–13 aus |
| `LIMIT 3 OFFSET 20` | gibt die Zeilen 21–23 aus |

**Achtung:** Es gibt noch eine Abkürzung, bei der OFFSET implizit an erster Stelle von `LIMIT` steht (kein Prüfungsstoff):

`LIMIT 3, 20` gibt die Zeilen 4–23 aus

### 5.3.20 Aggregatsfunktionen

- `count(`*`col`*`)`: Anzahl der Zeilen in *`col`*
- `min(`*`col`*`)`: kleinster Wert in *`col`*
- `max(`*`col`*`)`: grösster Wert in *`col`*
- `sum(`*`col`*`)`: Summe aller Werte in *`col`*

- `avg(col)`: Durchschnitt der Werte in *col*

Darüber hinaus kann man die üblichen Operatoren `+`, `-`, `*` und `/` zum Operieren auf den Kolonnen verwenden.

### 5.3.21 Subqueries (Unterabfragen)

Anstelle einer gegebenen Tabelle (nach `FROM`) darf auch ein weiteres `SELECT`-Statement stehen, das in runde Klammern eingeschlossen werden muss. (kein Prüfungsstoff)

```
SELECT ... FROM (SELECT ...) WHERE ...;
```

### 5.3.22 Mengenoperationen

Die Resultate von jeweils zwei `SELECT`-Ausdrücken können durch die folgenden Mengenoperationen im Sinne der Mengenalgebra verknüpft werden.

- `SELECT ... UNION SELECT ...`

  Bildet die Vereinigungsmenge ($\cup$) der Resultate der beiden `SELECT`-Ausdrücke.

- `SELECT ... INTERSECT SELECT ...`

  Bildet die Durchschnittsmenge ($\cap$) der Resultate der beiden `SELECT`-Ausdrücke.

- `SELECT ... EXCEPT SELECT ...`

  Bildet die Mengendifferenz ($\setminus$) von der ersten und der zweiten Resultatmenge.