SQLite-Tutorial

1 Allgemeines

Gross- und Kleinschreibung

Es ist üblich, SQL-Befehle in Grossbuchstaben zu schreiben. Dennoch unterscheidet SQL nicht zwischen Gross- und Kleinschreibung. Dies betrifft Schlüsselwörter, Tabellennamen und Kolonnennamen.

Bezeichner

Variablennamen sind als Zeichenfolgen einzugeben. Falls sie Leerzeichen oder spezielle Symbole enthalten, sind sie in doppelte Anführungszeichen (`"..."`) einzuschliessen. Aus Kompatibilitätsgründen mit anderen Datenbanken erlaubt SQLite auch, dass Bezeichner von eckigen Klammern (`[...]`) oder Backticks (`` `...` ``) eingeschlossen werden. Bezeichner, die mit dem Präfix `sqlite_` beginnen, sind für SQLite reserviert.

Leerraum, Befehlsende, Kommentare und Aufzählungen

SQL ist unempfindlich gegenüber Leerraum, einschliesslich Zeilenumbrüche.

Die Anweisungen in einer Folge von SQL-Befehlen werden jeweils durch ein Semikolon (`;`) getrennt. Bei einem einzelnen SQL-Befehl, dann dieser auch weggelassen werden.

Zeilenkommentare werden von zwei aufeinander folgenden Bindestrichen (`--`) eingeleitet. SQL unterstützt auch mehrzeilige Kommentare im Stil der C-Syntax (`/* ... */`).

In SQL-Anweisungen kommen oft Aufzählungen vor. Dabei wird ein Komma als Trennzeichen verwendet. Ein Komma am Ende einer Aufzählung ist nicht erlaubt.

Konstanten

- **ganze Zahlen:** `453`
- **Gleitkommazahlen:** `4.53` oder `3.22e-7` (mit Dezimal*punkt*)
- **Text:** `'Hello World'`
- **Binärdaten:** `x'A379C04'` oder `X'A379C04'`
- **Wahrheitswerte:** `0` steht für falsch und `1` steht für wahr.

Um ein einfaches Anführungszeichen innerhalb eines Textes darzustellen, wird es wiederholt (`'C''est quoi?'`). Der SQL-Standard kennt keine Maskierung mit Backslash.

Dreiwertige Logik

SQL verwendet `NULL` als Platzhalter für unbekannte oder fehlende Daten.

`NULL` interagiert nicht in der üblichen Weise mit anderen Werten, da beispielsweise nicht klar ist, welchen Wert der Ausdruck `NULL` > 5 hat.

Um mit diesem Problem umzugehen, verwendet SQL ein Konzept, das *dreiwertige* oder *ternäre Logik* (*three-valued logic, TVL*) genannt wird.

	TVL NOT
TRUE	
FALSE	
NULL	

TVL AND	**TRUE**	**FALSE**	**NULL**
TRUE			
FALSE			
NULL			

TVL OR	**TRUE**	**FALSE**	**NULL**
TRUE			
FALSE			
NULL			

Einfache Operatoren (Auswahl)

- `- +`: unäres Minus und Plus
- `+ - * / %`: binäre arithmetische Operatoren
- `< <= == != > >= %`: Vergleichsoperatoren

  SQL akzeptiert auch `=` anstelle von `==` und `<>` anstelle von `!=`.
- `||`: Textverkettungsoperator
- `NOT AND OR`: logische Operatoren (mit TVL)

Die SQL Datensprachen

SQL ist in vier Sprachteile gegliedert:

- *Data Definition Language (DDL)*: Befehle, welche die Struktur von Tabellen verändern (`CREATE TABLE`, `DROP TABLE`, ...)
- *Data Manipulation Language (DML)*: Befehle, die Daten in den Tabellen einfügen, löschen, verändern und abfragen (`INSERT INTO`, `DELETE`, `ALTER`, `SELECT`)
- *Transaction Control Language (TCL)*: Befehle, welche für die korrekte Ausführung der DDL- und DML-Befehle zuständig sind (`BEGIN`, `COMMIT`, ...).

- *Data Control Language (DCL)*: Regelung von Berechtigungen in Mehrbenutzersystemen (`GRANT`, `REVOKE`).

SQLite unterstützt den Grossteil der DDL- DML- und TCL-Befehle des SQL-Standards. Da SQLite aber weder Benutzernamen noch Logins kennt, sind dort keine DCL-Befehle implementiert.

2 Data Definition Language (DDL)

Tabellen erzeugen

```
CREATE TABLE [IF NOT EXISTS] table-name
(
   column-name column-type [column-constraints],
   [...]

   [table-constraints,
     [...]]
)
```

Datentypen in `column-type`

- NULL (unbekannt)
- INTEGER (ganze Zahlen)
- FLOAT (Gleitkommazahlen)
- TEXT (auch für Datum und Zeit)
- BLOB (Binary Large Objects)

Column-Constraints

- `PRIMARY KEY` (Primärschlüssel)
- `UNIQUE` (Werte müssen verschieden sein)
- `NOT NULL` (keine undefinierten Werte)
- `DEFAULT <value>` (z. B. `0` oder `CURRENT_DATE`)
- `AUTOINCREMENT` (automatisches Hochzählen bei `INTEGER`)
- `CHECK(<expr>)` (benutzerdefinierte Einschränkungen)
- `REFERENCES <table>` (Referenz auf Fremdschlüssel)

Table-Constraints

Die Funktionsweise ist ähnlich wie für Kolonnen, nur dass sich die Einschränkungen auf alle Kolonnen beziehen, sofern die Definitionen dies zulassen. Ferner lässt sich ein zusammengesetzter Primärschlüssel definieren:

- `PRIMARY KEY(a, b, c)`

oder nachträglich Referenzen auf Fremdschlüssel definieren

- `FOREIGN KEY()`

Tabellen löschen

Die Anweisung

```
DROP TABLE [IF EXISTS] table-name;
```

löscht die angegebene Tabelle, sofern sie existiert.

Tabellen ändern

SQLite kennt grundsätzlich zwei Arten, um Kolonnen zu modifizieren:

```
ALTER TABLE <table-name> RENAME TO <new-table-name>;
```

```
ALTER TABLE <table-name> ADD COLUMN <column-def>;
```

Das Löschen von Kolonnen kann nur über das Löschen der gesamten Tabelle und ihrem Neuaufbau realisiert werden.

Aufgabe 1

- Starte den DB Browser für SQLite und öffne eine leere Datenbank mit dem Namen `firma.db3`.
- Erstelle die Tabelle `mitarbeiter` mit einem SQL-Befehl. Verwende passende Datentypen und sorge dafür, dass der Primärschlüssel automatisch hochgezählt wird:

  mitarbeiter

mid	nachname	vorname	gehalt
...	...	...	...

- Speichere die SQL-Anweisung unter `aufgabe-01.sql` ab.

Aufgabe 2

- Erstelle in der soeben erstellten DB die folgenden Tabelle mit einem SQL-Befehl.

  abteilung

aid	bezeichnung
...	...

- Speichere die SQL-Anweisung unter `aufgabe-02.sql` ab.

Aufgabe 3

- Füge mit einem SQL-Befehl in der Tabelle `mitarbeiter` eine weitere Kolonne ein, die einen Fremdschlüssel auf die Tabelle `abteilung` enthält.
- Speichere die SQL-Anweisung unter `aufgabe-03.sql` ab.

3 Data Manipulation Language (DML)

Datensätze einfügen

Mit der Anweisung

```
INSERT INTO table_name (column_name [, ...])
VALUES (value [, ...]);
```

wird ein Datensatz in der angegebenen Tabelle eingefügt. Dabei werden die Werte nach `VALUES` in der Reihenfolge eingefügt, in der sie vor `VALUES` angegeben wurden.

Lässt man Kolonnen mit einer Voreinstellung oder einem automatischen Zähler weg, so werden automatisch die entsprechenden Werte eingefügt. In den anderen Fällen wird NULL eingefügt.

Falls ein Wert im Widerspruch zu den Tabelleneinschränkungen steht, wird die Ausführung mit einer Fehlermeldung verweigert.

Bei Verwendung der alternativen Form

```
INSERT INTO table_name VALUES (value [, ...]);
```

ohne explizite Vorgabe der Kolonnen, müssen die Werte in exakt derselben Reihenfolge wie in der Tabellendefinition eingegeben werden. Es ist dann aber unmöglich, Vorgaben oder automatische Zähler zu verwenden.

Bemerkung: Bei den oben beschriebenen Einfügeoperationen wird für jeden Einfügevorgang eine einzelne Transaktion (siehe TCL) durchgeführt. Dies hat den Vorteil, dass alle bis zum ersten Fehler durchgeführten Transaktionen garantiert ausgeführt wurden. Dafür dauert es länger. Für grössere Datenimporte empfiehlt es sich deshalb, jeweils 1 000 bis 10 000 INSERTs zu einer Transaktion zusammenzufassen.

Aufgabe 4

- Füge mit den entsprechenden SQL-Befehlen die folgenden Daten in der Tabelle `abteilung` ein.

  abteilung

aid	bezeichnung
100	Produktion
200	Verkauf
300	Forschung und Entwicklung
400	Marketing
500	Management

- Speichere die SQL-Anweisung unter `aufgabe-04.sql` ab.

Aufgabe 5

- Füge mit den entsprechenden SQL-Befehlen die folgenden Datensätze in der Tabelle `mitarbeiter` ein.

  mitarbeiter

mid	nachname	vorname	gehalt	aid
1	Abt	Tim	6100	200
2	Muster	Pia	7500	500
3	Kern	Lea	7100	400
4	Liem	Karl	7400	400
5	Honda	Akiro	8100	300
6	Kim	Yao	6800	100

- Speichere die SQL-Anweisung unter `aufgabe-05.sql` ab.

Datensätze ändern

Mit der Anweisung

```
UPDATE table_name SET column_name=value [, ...]
WHERE expression;
```

werden in der angegebene Tabelle und in den vom Ausdruck nach `WHERE` ausgewählten Zeilen, den angegebenen Kolonnen neue Werte zugewiesen.

Achtung: ohne einen `WHERE`-Ausdruck werden *alle* Zeilen der angegebenen Tabelle geändert.

Aufgabe 6

- Schreibe eine SQL-Anweisung, welche den Lohn von Lea Kern in der Tabelle `mitarbeiter` auf CHF 7200.– festsetzt.
- Speichere die SQL-Anweisung unter `aufgabe-06.sql` ab.

Datensätze löschen

Mit der Anweisung

```
DELETE FROM table_name WHERE expression;
```

werden in der angegebene Tabelle alle Zeilen gelöscht, auf die der `WHERE`-Ausdruck zutrifft.

Achtung: ohne einen `WHERE`-Ausdruck werden *alle* Zeilen der angegebenen Tabelle gelöscht.

Aufgabe 7

- Der Mitarbeiter mit der ID 3 hat gekündigt. Schreibe eine SQL-Anweisung, die ihn aus der Mitarbeiter-Tabelle entfernt.
- Speichere die SQL-Anweisung unter `aufgabe-07.sql` ab.

Die SELECT-Pipeline

Die `SELECT`-Anweisung wird dazu verwendet, um Daten aus einer Datenbank auszuwählen und auszugeben. Die allgemeine Syntax lautet:

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

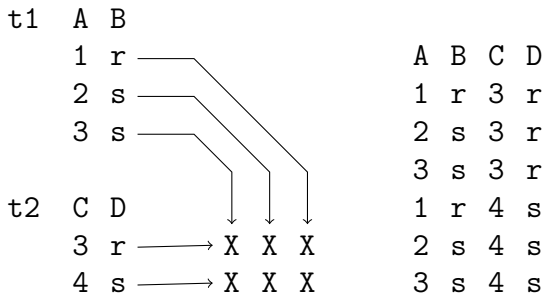
Achtung: Die Reihenfolge, in der die Teilausdrücke intern ausgewertet werden, weicht von der in der Eingabe ab.

(1) `FROM source_tables`

Wählt eine oder mehrere Tabellen aus und kombiniert sie zu einer einzigen grossen Arbeitstabelle.

CROSS JOIN

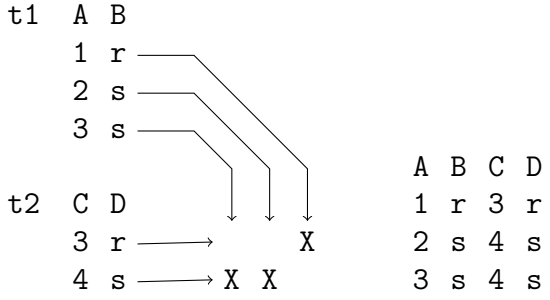
```
t1 CROSS JOIN t2
```



`CROSS JOIN` bildet das Kreuzprodukt der beiden Tabellen. Dabei können unter Umständen sehr grosse Tabellen entstehen.

INNER JOIN

```
t1 [INNER] JOIN t2 ON B=D
```



Es gibt noch weitere Varianten von Kreuzprodukten, die jedoch über den Umfang dieser

Einführung hinausgehen.

Tabellen-Aliase

Mit dem Schlüsselwort `AS` kann einer Tabelle ein neuer Name (Alias) gegeben werden.

- lange Tabellennamen:
  `SELECT ... FROM longname1 AS t1 JOIN longname2 AS t2 ON (t1.x=t2.y)`
- JOINs über mehrere Kolonnen der gleichen Tabelle:
  `SELECT ... FROM tab AS t1 JOIN tab AS t2 on (t1.a=t2.b)`
- Abfragen auf Resultaten von Abfragen (*subqueries*):
  `SELECT ... FROM (SELECT ... ) AS result ...`

In den letzten beiden Fällen kommt man an dem Alias nicht vorbei.

(2) `WHERE filter_expression`

Wählt aus der in (1) erzeugten Tabelle die Ausdrücke aus, auf die `filter_expression` zutrifft. Beispiele:

- `SELECT ... WHERE ort == 'Stans'`
- `SELECT ... WHERE umsatz > 1000`
- `SELECT ... WHERE gehalt > 4000 AND ort == 'Stans'`
- `SELECT ... WHERE name NOT NULL`

Anstelle von `==` und `!=` sind auch `=` bzw. `<>` erlaubt.

Texte können zusätzlich mit `LIKE` verglichen werden und erlauben die folgenden Wildcards (Jokerzeichen):

- `%` eine beliebige Folge von Zeichen
- `_` ein einzelnes Zeichen

(3) `GROUP BY grouping_expression`

Fasst die Zeilen der Tabelle gemäss der `grouping_expression` zusammen.

Beispiel 3.1

Die Anweisung

```
SELECT * FROM mitarbeiter GROUP BY ort;
```

angewendet auf

mitarbeiter

mid	name	ort	gehalt
107	Müller	Stans	7000
42	Kunz	Buochs	7500
228	Kern	Stans	6500
397	Widmer	Stans	8000

liefert die folgende, auf den ersten Blick zweizeilige Tabelle:

mid	name	ort	gehalt
42	Kunz	Buochs	7500
397	Widmer	Stans	8000

Es ist zwar nur einer der drei Datensätze mit `ort='Stans'` sichtbar, jedoch sind die übrigen zwei für Aggregatsfunktionen verfügbar (siehe weiter unten). Dies ist vergleichbar mit einem Stapel Spielkarten, bei der nur die oberste Karte sichtbar ist.

(4) `SELECT select_heading`

Wählt die gewünschten Kolonnen aus, wendet allfällige Aggregatsfunktionen darauf an und setzt bei Bedarf neue Spaltenüberschriften. Ein Stern ($*$) zeigt alle Kolonnen an.

Beispiel 3.2

Wenden wir

```
SELECT gehalt AS "Gehalt", name AS "Nachname";
```

auf die Tabelle `mitarbeiter` von Beispiel 3.1 an, erhalten wir

Gehalt	Nachname
7000	Müller
7500	Kunz
6500	Kern
8000	Widmer

Die doppelten Anführungszeichen sind nur bei Leer- oder Sonderzeichen nötig.

Abhängig vom Datentyp, lassen sich die Werte einzelner Kolonnen vertikal zusammenfassen (aggregieren). Hier eine Auswahl der wichtigsten Aggregatsfunktionen:

- `COUNT(column_name)` (Zeilen zählen)
- `SUM(column_name)`
- `AVG(column_name)` (Durchschnitt bilden)
- `MIN(column_name)`
- `MAX(column_name)`

Beispiel 3.3

Die SQL-Anweisung

```
SELECT ort AS Ort, SUM(gehalt) AS Lohnsumme
FROM mitarbeiter
GROUP BY 'ort';
```

berechnet aus der Tabelle `mitarbeiter` die Lohnsumme der Mitarbeiter nach Orten:

Ort	Lohnsumme
Buochs	7500.0
Stans	21500.0

Beispiel 3.4

Ohne Angabe von Tabellen mit `WHERE` wertet `SELECT` die durch Kommas getrennten Ausdrücke aus und zeigt sie an.

```
SELECT 1+1 AS Summe, 5*32 AS Produkt,
'abc' || 'xyz' AS Text, 1>2 AS Bool;
```

liefert die Resultattabelle:

Summe	Produkt	Text	Bool
2	160	abcxyz	0

(5) `HAVING filter_expression`

Funktioniert wie ein `WHERE`, bei dem die `filter_expression` auf eine Tabelle angewendet wird, die *zuvor* mit `GROUP BY` zusammengefasst wurde.

Im Beispiel 3.3 könnte man, nachdem die Gruppierung und Aggregierung durchgeführt wurde, mit

```
SELECT ... HAVING lohnsumme > 20000 ...
```

die Wohnorte anzeigen lassen, in denen die Lohnsumme der Mitarbeiter CHF 20 000 übersteigt.

(6) `DISTINCT`

Eliminiert identische Zeilen. Im Gegensatz zu `GROUP BY`, lassen sich die entsprechenden Kolonnen nicht aggregieren.

(7) `ORDER BY ordering_expression`

Sortiert die Zeilen gemäss `ordering_expression`.

`SELECT ... ORDER BY ort ASC;` (aufsteigend nach Ort)

`SELECT ... ORDER BY ort DESC;` (absteigend nach Ort)

Es kann zusätzlich eine Vergleichsfunktion (*collation*) angegeben werden, die angibt, wie Textzeichen in anderen Zeichensätzen verglichen werden.

(8) `OFFSET count`

Überspringt `count` Zeilen.

(9) `LIMIT count`

Gibt `count` Zeilen aus.

Beispiel 3.5

`test`

A	B	C
13	z	4.5
29	e	7.6
6	s	5.8
45	a	3.9
77	p	6.1

```
SELECT * FROM test ORDER BY A LIMIT 2;
```

gibt folgende Tabelle zurück:

A	B	C
6	s	5.8
13	z	4.5

Beispiel 3.6

Beispiel: Mit der Tabelle `test` aus Beispiel 3.5 liefert

```
SELECT * FROM test
ORDER BY B DESC
LIMIT 2
OFFSET 1;
```

folgende Tabelle zurück:

A	B	C
6	s	5.8
77	p	6.1

4 Transaction Control Language (TCL)

Transaktionen

Transaktionen fassen Datenbankänderungen zu einem einzigen „atomaren“ Ereignis zusammen.

Dies ist wichtig, wenn man kritische Datenbankmanipulationen wie beispielsweise das Verbuchen von Geldbeträgen in einer Datenbank realisiert. Nach einem Fehler oder einem Computerabsturz könnte die Buchhaltung nicht mehr ausgeglichen sein bzw. Kunden könnten Geld verlieren.

ACID-Kriterien

In einer Datenbank sollten Transaktionen die folgende Eigenschaften haben:

- *Atomic:* Die Transaktionen können nicht in kleinere Einheiten unterteilt werden. Entweder ist die ganze Transaktion erfolgreich oder nicht.
- *Consistent:* Vor und nach jeder Transaktion erfüllt die Datenbank ihre Regeln und Einschränkungen.
- *Isolated:* Die Transaktionen der einzelnen DB-Benutzer sind voneinander abgeschirmt.
- *Durable:* Nach jeder Transaktion muss der Zustand der Datenbank dauerhaft sein. Auch nach Stromausfällen oder Datenverlusten.

Autocommit-Mode

Standardmässig führt SQLite jede Anweisung als separate Transaktion aus, die insgesamt entweder erfolgreich ist oder fehlschlägt.

BEGIN ... COMMIT

Mit

```
BEGIN [TRANSACTION]
  sql-statement-1;
  sql-statement-2;
  ...
COMMIT [TRANSACTION]
```

wird der Autocommit-Modus ausgeschaltet und die angegebenen SQL-Anweisungen werden zu einer Transaktion zusammengefasst.

Dies kann z. B. dann sinnvoll sein, wenn man mehrere `INSERT INTO`-Operationen zu einer Transaktion zusammenfassen möchte, um sie schneller auszuführen. Sollte jedoch dabei ein Fehler auftreten, muss die gesamte Transaktion wiederholt werden.

Üblicherweise werden die Transaktionen verzögert (*deferred*) durchgeführt, um bei mehreren Benutzern eine kooperative Nutzung der Datenbank zu erreichen.

Es gibt auch die Möglichkeit, eine Transaktion *immediate* oder *exclusive* durchzuführen, auf die hier aber nicht eingegangen wird.

Weitere TCL-Befehle

Mit `ROLLBACK [TRANSACTION]` lassen sich Transaktionen wieder Rückgängig machen und mit `SAVEPOINT savepoint-name` kann ein Zustand angegeben werden, zu dem man bei einem `ROLLBACK` allenfalls wieder zurückkehren möchte.

Auch darauf wird nicht im Detail eingegangen.