Datenbank & SQLite

Christoph Rinne*

23. Dezember 2021

Inhaltsverzeichnis

1 Grundlegende Information

SQL steht für *structured query language* und dient dem Erstellen, Nutzen und Modifizieren von relationalen Datenbanken. Ob Sie eine SQL-Datenbank verwenden sollten hängt von der Komplexität Ihrer Daten und der geplanten Nutzung ab. Nachfolgend spielen Datenstrukturen und -modelle vorderhand keine zentrale Rolle. Sie sollten sich damit aber vor dem Erstellen einer Datenbank befassen.

SQL ist weit verbreitet, Sie finden SQL-Syntax z.B. in QGIS für das Filtern eines Datebestandes. Die grundlegenden Befehle und die zugehörige Syntax sind weitgehend einheitlich, es gibt aber auch zahlreiche Erweiterungen für einzelne Datenbanksysteme (MySQL, PostgreSQL, ORACLE etc). SQL Datenbanksysteme sind fast ausnahmslos Server-Systeme, eine Ausnahme ist SQLite.

SQL ist eine Sprache und keine Programm für eine Nutzeroberfläche wie es MS Access oder Libre Office Base bieten.

SQL ist die *lingua franca* für relationale Datenbanken. Auch deshalb finden Sie viele andere Einführungen im Internet.

Für SQLite ist das System und wird *a priori* auf der Befehlszeile ausgeführt. Es gibt aber diverse grafische Nutzeroberflächen, ich verwende aktuell gerne DB Browser for SQLite (s.u.).

*Christian-Albrechts Universität zu Kiel, crinne@ufg.uni-kiel.de

1.1 SQLite in R / RStudio

Da ich diesen Text in R-Markdown mit RStudio schreibe verwende ich neben SQLite zudem das Paket RSQLite, um die Befehle in sogenannten *code-chunks* darstellen und ausführen zu können. Das Paket zielt aber auf eine schnelle Integration von Daten aus einer Datenbank in die Analyse in R.

Sofern noch nicht vorhanden wird mit dem folgenden Code das Paket RSQLite in R installiert und dann geladen.

```
require (RSQLite) || install.packages("RSQLite")
```

```
## [1] TRUE
```

```
library(RSQLite)
```

Dann wird eine neue SQLite Datenbank im Arbeitsspeicher angelegt und zugleich eine Verbindung "t1db" hergestellt. In einem anderen Programm müssen Sie ebenfalls eine neue Datenbank erstellen die dann auch automatisch geöffnet würde. Um in R auf eine vorhandene Datenbank zuzugreifen müssten Sie 'dbname = "" ' schreiben.

```
t1db<-dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

Egal mit welcher Software, Sie sollten nun eine leere Datenbank geöffnet haben.

2 SQL Grundlagen

Im folgenden Abschnitt wird eine Tabelle erstellt und Daten in diese eingetragen, abgefragt als auch modifiziert.

2.1 Tabelle erstellen

Mit den folgenden Befehlen wird die Tabelle tbl_features ggf. gelöscht, also rigoros aufgeräumt, und dann erstellt.

```
DROP TABLE IF EXISTS tbl_features;
```

```
CREATE TABLE "tbl_features" (
    "feature_id"    INTEGER PRIMARY KEY,
    "feature_nr"    INTEGER NOT NULL,
    "f_name"    TEXT,
    "f_type"    TEXT,
    "f_dating"  TEXT,
    "discoverdate"  TEXT
);
```

Die englisches Befehle und Parameter müssen wohl nicht erläutert werden. Jede Anweisung kann zur besseren Lesbarkeit über mehrere Zeilen laufen und wird mit einem ";" beendet. Die Großbuchstaben sind die traditionelle Hervorheben der Syntax und nicht notwendig. Klammern gliedern nach allgemein gültigen Regeln die Bestandteile. Gesondert erwähnt sei: **primary key** definiert feature_id als Primärschlüssel wodurch Eindeutigkeit und auch eine automatische Zählung impliziert wird. Im Kern ist es ein dezidiert zugewiesener Name für die stets im Hintergrund geführte Zeilennummer (rowid). Mit **not null** wird die Angabe einer Befundnummer zur Pflicht.

Folgende **Datentypen** können gesetzt werden: integer (Ganzzahl), real (reelle Zahl), text (Text) und blob (binary long objct, Binärobjekt). Gängige Datentypen in SQL sind dann noch date (Datum mit Uhrzeit) und boolean (Ja/Nein). Zudem wird oft zwischen "klein" und "groß" oder expliziter Dimension unterschieden: integer, long integer, char, varchar etc. SQLite ist hier etwas anders als Sie es ggf. aus anderen Systemen schon kennen (Stichwort type affinity). Das Thema ist hier jetzt nicht weiter relevant.

Weiteres zur Tabellendefinition später.

2.2 Daten eintragen : INSERT

Datenbanken sollten schon bei der Erhebung der Daten durch entsprechende Strukturen die Qualität der Daten gewährleisten, z.B. durch übersichtliche und aufgabenorientierte Formulare mit Datenkontrolle im Hintergrund. Die nachfolgende Syntax ist also *a priori* nicht für die Handarbeit gedacht.

```
INSERT INTO tbl_features (feature_nr, f_name, f_type, f_dating, discoverdate)
VALUES
(1001, 1001, 'posthole', 'not dated', '2019-05-22'),
(1002, 1002, 'pit', 'neolithic', '2019-05-22'),
(1001, 1001, 'grave', 'Corded Ware', '2019-05-22')
;
```

Die vorangehende Syntax nenn erst die Tabelle und die zu belegenden Felder, danach folgen eingeleitet durch *values* die Daten (*recordset*). Alle Werte werden duch "," getrennt, Text muss mit Anführungszeichen (" oder ') versehen werden und die Werte einer Datenzeile (*record*) stehen jeweils in Klammern. Beachten Sie bitte, die Anzahl und Reihenfolge der Felder und nachfolgend aufgeführten Werte müssen sich entsprechen.

Die zentrale Syntax von *INSERT*ist also einfach:

```
INSERT INTO table-name (attribute1, attr..) VALUES (value1, val...);
```

Im Vergleich zur vorgefertigten Anfügeabfrage erkennen Sie: 1. Wir brauchen Kenntnis über die Feldparameter (Zahl, Text, Schlüsselfeld, nicht Null, etc) und müssen entsprechend handeln sonst erfolgt eine Fehlermeldung. Allerdings können wir Zahlen auch ohne Anführungszeichen in ein Textfeld schreiben. 2. Sie können nicht nur eine Datenzeile sondern auch mehrere Zeilen anfügen. Wenn Sie stets nur eine Datenzeile (atomar) anfügen wissen Sie sofort wann ein Fehler auftritt. Dafür dauert es meist etwas länger, allerdings nicht bei SQLite. In der folgenden Anweisung fehlt z.B. die notwendige Angabe zur Befundnummer in dem Feld feature_nr.

```
INSERT INTO tbl_features (f_name) VALUES (1005);
```

Die folgende Anweisung nennt keine Felder, deshalb müssen alle vorhandenen Felder in der korrekten Reihenfolge bedient werden. Für das zu ergänzende Datum wird die Funktion *date()* genutz, *date('now')* liefert das Systemdatum im Standardformat 'YYYY-MM-DD' (weitere Infos).

```
INSERT INTO tbl_features VALUES (4, 1004, '1004', 'grave', 'Corded Ware', date('now'));
```

Und noch einige Daten zum Üben.

```
INSERT INTO tbl_features (feature_nr, f_type, f_dating, discoverdate)
VALUES
(1005, 'pit', 'Late Neolithic', DATE('now')),
(1006, 'posthole', 'Neolithic', DATE('now')),
(1007, 'posthole', 'not dated', DATE('now'));
```

2.3 Daten abfragen : SELECT

SELECT ist eine der wichtigsten Anweisungen und es kann alles abgefragt werden was existiert. Wie zuvor erwähnt steht Text in Anführungszeichen, beachten Sie also das Ergebnis der folgenden Abfrage.

```
SELECT 'col 1', 2, '3', 1+3, pi() AS magic;
```

Tabelle 1: 1 records

'col 1'	2	'3'	1+3	magic
col 1	2	3	4	3.141593

Die zuvor in der Tabelle tbl_features eingetragenen Daten können wir nun durch anfügen des Tabellennamens

als Quelle abfragen wobei wir statt einzelner Felder mit * schlicht alle abfragen.

```
SELECT * FROM tbl_features;
```

Tabelle 2: 7 records

feature_id	feature_nr	f_name	f_type	f_dating	discoverdate
1	1001	1001	posthole	not dated	2019-05-22
2	1002	1002	pit	neolithic	2019-05-22
3	1001	1001	grave	Corded Ware	2019-05-22
4	1004	1004	grave	Corded Ware	2021-12-23
5	1005	NA	pit	Late Neolithic	2021-12-23
6	1006	NA	posthole	Neolithic	2021-12-23
7	1007	NA	posthole	not dated	2021-12-23

2.3.1 Filtern

Natürlich gibt es zahlreiche Möglichkeiten, die Abfrage anzupassen um die gewünschte Information zu erhalten. Das ist vor allem die Bedingung mit *WHERE*. Für Zahlen können folgende Operatoren verwendet werden: <, >, = , >=, <=, <>.

```
SELECT feature_nr, f_type, f_dating FROM tbl_features
WHERE feature_nr > 1005;
```

Tabelle 3: 2 records

feature_nr	f_type	f_dating
1006	posthole	Neolithic
1007	posthole	not dated

Text muss weiterhin in Anführungszeichen stehen und als Operatoren werden erneut = und <> verwendet, dazu *like* für unscharfe Suchen in Ergänzung mit % innerhalb der Anführungszeichen und *is not* für die Negation eines Operators. Mit Klammern, *and* und *or* können komplexere Bedingungen erstellt werden. Beachten Sie bitte später auch Unterschiede zwischen *NULL* und " " als Hinweis auf eine leere Zeichenkette.

```
SELECT feature_nr, f_type, f_dating FROM tbl_features
WHERE ((f_type is not null) and (f_dating not like "n%"));
```

Tabelle 4: 3 records

feature_nr	f_type	f_dating
1001	grave	Corded Ware
1004	grave	Corded Ware
1005	pit	Late Neolithic

2.3.2 Sortieren

Die Rückgabe einer Abfrage kann nach einem oder mehreren Sortierfeldern, dann durch "," getrennt, absteigend (*desc*) oder aufsteigend (*asc*) sortiert werden.

```
SELECT feature_nr, f_name, f_type, f_dating FROM tbl_features
ORDER BY f_type DESC, feature_nr ASC;
```

Tabelle 5: 7 records

feature_nr	f_name	f_type	f_dating
1001	1001	posthole	not dated
1006	NA	posthole	Neolithic
1007	NA	posthole	not dated
1002	1002	pit	neolithic
1005	NA	pit	Late Neolithic
1001	1001	grave	Corded Ware
1004	1004	grave	Corded Ware

2.3.3 Gruppieren bzw. Aggregieren

Typischerweise wollen sie sich eine Überblick über die Daten verschaffen, z.B. wieviele Befunde habe ich je Epoche und welche Befundtypen sind vertreten? Die folgende Anweisung erledigt dies umgehend. Beachten Sie:

- GROUP BY : Aggregiert den ausgewählten Datensatz (FROM) nach den genannten Feldern.
- count() : Die Funktion liefert die Anzahl der aggregierten Datenzeilen.
- group_concat() : Die Funktion ist SQLite spezifisch und liefert die verketten Feldwerte mit dem genannten Trennzeichen.
- AS : Jeder Rückgabewert kann mit einem Etikett (*label*) versehen werden (s.o.), statt der Anweisung nach *SELECT* wird dann dieses als Spaltenüberschrift oder für Verweise verwendet. Achten Sie auf die Anführungszeichen wegen "." und "-" in "Bef.-Nr.".

```
SELECT f_dating, count(feature_nr) AS n, group_concat(feature_nr, ", ") AS 'Bef.-Nr.'
FROM tbl_features
GROUP BY f_dating
```

Tabelle 6: 5 records

f_dating	n	Bef.-Nr.
Corded Ware	2	1001, 1004
Late Neolithic	1	1005
Neolithic	1	1006
neolithic	1	1002
not dated	2	1001, 1007

Wollen Sie die aggregierten Daten filtern wird statt WHERE die Anweisung HAVING nach GROUP BY gesetzt. Als Beispiel hier die Suche nach doppelten Fundnummern.

```
SELECT feature_nr as Dubletten FROM tbl_features
GROUP BY feature_nr
HAVING count(feature_nr) > 1;
```

Tabelle 7: 1 records

Dubletten
1001

Es gibt deutlich mehr Bedingungen und Optionen in SELECT-Abfrage aber es reicht für den Anfang.

2.4 Daten modifizieren : UPDATE

Die letzten Abfragen offenbahren einen gravierenden und auch weitere Fehler: Die Befundnummer 1001 ist doppelt vergeben, f_name fehlt in einigen Fällen und *Neolithic* ist einmalig klein geschrieben. Letzteres zuerst:

```
UPDATE tbl_features SET f_dating = 'Neolithic'
WHERE f_dating='neolithic';
```

Beachten Sie bitte die missliche Zweideutigkeit in SQL von "=", einmal als "ist gleich" und einmal als Zuweisung. Auch wird das lehre Feld f_name mit f_name=" " nicht gefunden, hier müssen wir korrekt auf *IS NULL* filtern. Da NULL nichts ist, nichteinmal "", ist"=" auch kein zulässiger Operator.

```
UPDATE tbl_features SET f_name = feature_nr
WHERE f_name IS NULL;
```

2.5 Unterabfragen

Für die Dublette nutzen wir die oben erstellte Abfrage nach den Dubletten als Filter bzw. Unterabfrage. Diese Unterabfrage wird mit "*IN*" übergeben und die Unterabfrage als ganzes in Klammern gesetzt. Im Ergebnis erhalten wir eine Liste aller Felder der Dubletten, um eine Entscheidung treffen zu können. Ich würde den Datensatz mit feature_id=3 auf die bisher fehlende Fundnummer 1003 ändern.

```
SELECT * FROM tbl_features
WHERE feature_nr IN (SELECT feature_nr FROM tbl_features
GROUP BY feature_nr
HAVING count(feature_nr) > 1);
```

Tabelle 8: 2 records

feature_id	feature_nr	f_name	f_type	f_dating	discoverdate
1	1001	1001	posthole	not dated	2019-05-22
3	1001	1001	grave	Corded Ware	2019-05-22

```
UPDATE tbl_features
SET feature_nr = 1003, f_name = '1003'
WHERE feature_id = 3;
```

2.6 Löschabfragen

Löschabfragen sind recht einfach und leider irreversibel, also vorher prüfen. Oder Sie multiplizieren bei den zu löschenden Datensätzen die ID mit -1 und filtern immer auf positive, intendiert gültige IDs.

```
DELETE FROM tbl_features WHERE ((feature_nr <> f_name) AND (feature_id < 0));
```

3 Tabellen, Daten und Relationen

Wie oben erwähnt: die Strukturierung der Daten in diversen Tabellen und deren Felder sollte nicht unüberlegt erfolgen. Investieren Sie etwas Zeit, lesen Sie etwas, skizzieren Sie Ihre Struktur in einem sogenannten Entity-Relationship-Model (ERM) und fragen Sie um Rat.

3.1 Tabelle erstellen

Die folgende Tabelle soll Informationen zu Fundkontexten (Fundzettel) beinhalten. Dabei ist hier wichtig, dass ein Fund nicht ohne Befund existieren soll und deshalb auf diesen als Fremdschlüssel verweist.

```
DROP TABLE IF EXISTS "tbl_findcontexts";
```

```
CREATE TABLE "tbl_findcontexts" (
    "findcontext_id"    INTEGER PRIMARY KEY,
    "find_nr" INTEGER NOT NULL,
    "feature_nr"    INTEGER NOT NULL,
    "finddate"  TEXT,
    "description"   TEXT,
    "created" TEXT DEFAULT current_date,
    FOREIGN KEY (feature_nr) REFERENCES tbl_features(feature_nr),
    CONSTRAINT "feature_find" UNIQUE (feature_nr, find_nr)
);
```

Wie oben erwähnt, die grundlegende Syntax ist einfach: `CREATE TABLE <table name> (<attribute> data type[, <attribute> data type]);` Der Tabellenname und mindestens ein Feld mit nachfolgendem Datentyp sind Pflicht.

3.2 Datentypen

In Ergänzung zu den oben genannten Datentypen *integer*, *real*, *text* und *blob* hier ergänzend Informatione zu:

- *boolean* : Sind nicht vorgesehen, sondern werden als 1 (WAHR) oder 0 (FALSCH) gespeichert. Das ist übrigens Standard, schreiben Sie in MS Excel mal in eine Zelle `=GANZZAHL(WAHR)` oder `=GANZZAHL(FALSCH)`.
- *date* oder *time* : Auch diese Datenformat sind so nicht vorhanden, sondern werden als Text oder Zahl gespeichert. Dazu die folgende Abfrage.

```
select date('now'), date('now','+1 day'), datetime('now'),
  strftime('%s', 'now') AS 'UNIX time', julianday('now');
```

Tabelle 9: 1 records

date('now')	date('now','+1 day')	datetime('now')	UNIX time	julianday('now')
2021-12-23	2021-12-24	2021-12-23 14:19:17	1640269157	2459572

Datum und Zeit werden als Text mit definierten Elementen gespeichert (YYYY-MM-DD HH:MM). Das ist im Grunde aber nur eine Repräsentation, denn Sie können mit dem Datum auch rechnen und die Rückgabe von *date()* auch multiplizieren. Die UNIX Zeit sind die Sekunden ab dem 1.1.1970 und das Julianische Datum ist eine Tageszählung seit -4713 (4713 v. Chr.) mit der Uhrzeit in Nachkommastellen.

Datentypen sind in SQLite insgesamt ein sehr eigenes Thema, im Unterschied zu anderen SQL-Systemen werden diese sehr dynamisch in Klassen umgesetzt (Datatypes). Dazu in Anlehnung an das Beispiel aus dem vorangehenden Link.

```
create table test (i integer, t text, d integer);
```

```
insert into test values ('123', 123, datetime('now'));
```

```
select * from test;
```

Tabelle 10: 1 records

i	t	d
123	123	2021-12-23 14:19:17

```
DROP TABLE IF EXISTS test;
```

Durch die Datenklassen werden unterschiedliche Datentypen zusammengefasst und durch eine intern definierte Hierarchie auch ineinander konvertiert. Dieser Vorgang wird als Datenaffinität (*type affinity*) bezeichnet und gibt mehr Freiheit beim Entwurf der Datenbank.

Feldparameter: Standardwerte, Referenzen und Schlüsselfelder

Den einzelnen Feldern können nach dem Datentyp bzw. der Datenklasse weitere Parameter zugewiesen werden. All dies soll die Datenintegrität sicherstellen:

- *PRIMARY KEY* : Setzt den Primärschlüssel und damit die für jede Tabelle mitgeführte *rowid* auf dieses Feld. Das impliziert eine automatische Zählung und Eindeutigkeit allerdings mit der Ausnahme, dass NULL erlaubt ist und mehrfach vorkommen kann. Sie können also nachträglich die automatisch vergebenen ID's löschen es sei denn, Sie ergänzen noch NOT NULL. Es kann nur einen Primärschlüssel je Tabelle geben.

- *UNIQUE* : Die Werte müssen eindeutig sein wobei auch hier NULL erlaubt ist, es sei denn, Sie ergänzen *NOT NULL*.
- *NOT NULL* : Ein Wert muss angegeben werden.
- *DEFAULT* '' : Weist dem Feld einen Standardwert zu. Im Beispiel das Systemdatum.
- *AUTOINCREMENT* : Ist möglich, die SQLite-Dokumentation mahnt aber zur Vorsicht: Es wird stets erhöht, einmal gelöschte Werte werden nicht erneut vergeben (SQLite Autoincrement).

3.3 Schlüsselfelder und Indices

Jede Tabelle und deren Felder kann weiteren Einschränkungen (*constraints*) unterliegen. Jede Einschränkung benötigt einen Namen und danach eine Anweisung, z.B. einen Werte zu prüfen. Für die Tabelle der Fundkontexte soll die Kombination aus Befundnummer und Fundnummer eindeutig sein. Die entsprechende Einschränkung lautet `CONSTRAINT "feature_find" UNIQUE (feature_nr, find_nr)`. Wäre das Feld findkontext_id nicht schon Primärschlüsselfeld wäre mit `CONSTRAINT "feature_find" PRIMARY KEY (feature_nr, find_nr)` oder kurz `PRIMARY KEY (feature_nr, find_nr)` etwas sehr Ähnliches erzielt worden.

Indices, auch mit dezidierter Eindeutigkeit, können nach der Tabellendefinition erstellt werden. Im vorgenannten Fall wäre die Anweisung dann:

```
CREATE UNIQUE INDEX "feature_find" ON tbl_findcontexts (feature_nr, find_nr).
```

3.4 Externe Refferenz und Fremdschlüssel

Die Funkontexte in der Tabelle sollen sich auf einen Befund der Befundtabelle beziehen, dies wird mit einer Referenz *FOREIGN KEY* erzielt (1:n). Eine Eins-zu-Viele-Referenz ist der Standard in SQL und die Zuweisung von keinem bis vielen Fundkontexten (Fundzettel) zu einem Befund ist ein typisch Fall. Der übergeordnete (Eltern-) Datensatz (*record*) muss eindeutig sein. Meist ist es das Primärschlüsselfeld es kann aber auch ein Feld mit dem Parameter *UNIQUE* sein. Ein Fremdschlüssel kann sich auch auf einen eindeutigen Schlüssel aus mehreren Feldern beziehen, z.B. die einzelnen Funde auf die Kombination von Befund-Nr. und Fundzettel-Nr. (feature_nr, find_nr). Für das vorliegende Beispiel der Befundnummer der Fundzettel lautet die Syntax:

```
FOREIGN KEY (feature_nr) REFERENCES tbl_features(feature_nr)
```

3.5 Trigger

Trigger führen automatisch Anweisungen aus wenn bestimmte Ereignisse auftreten. Mit Blick auf unsere bisherige Datenbank darf es z.B. keine Funde aus einem Befund geben der gelöscht wird. Die hier notwendige Syntax wird zunehmend komplexer und wird hier nicht ausgeführt. Bitte lesen Sie hierzu die originale Dokumentation (*CREATE TRIGGER*).

Fügen wir zum Abschluss doch noch einige Daten in die Tabelle der Fundkontexte ein.

```
INSERT INTO tbl_findcontexts (find_nr, feature_nr, finddate, description)
VALUES
(1, 1005, DATE('now'), 'surface finds'),
(2, 1005, DATE('now', '+2 day'), '1. layer'),
(3, 1005, DATE('now', '+1 month'), 'bottom layer');
```

```
Select * FROM tbl_findcontexts;
```

Tabelle 11: 3 records

findcontext_id	find_nr	feature_nr	finddate	description	created
1	1	1005	2021-12-23	surface finds	2021-12-23
2	2	1005	2021-12-25	1. layer	2021-12-23
3	3	1005	2022-01-23	bottom layer	2021-12-23

4 Daten kombinieren : JOIN