1NF verlangt nur atomare Attribute in der Tabelle. Werte einzeln abrufbar/abfragbar.

2NF verlangt 1NF und das Attribute vom gesamten Primärschlüssel abhängig.

3NF verlangt 2NF und das Nicht-Primärschlüssel-Attribute nicht transitiv abhängig.

Ergebnis:

- Verminderung von Redundanz,
- Vermeidung von Inkonsistenz und Anomalien,
- Übersichtliche kleine Tabellen.

Aber:

- Zu viele kleine Tabellen erschweren Überblick auf Miniwelt.
- Abfragen über mehrere Tabellen sehr aufwendig.

SQL

Allgemeines
`CREATE TABLE`
`DROP TABLE`
`ALTER TABLE`
`INSERT INTO`
`UPADTE`
`DELETE`
`SELECT`

Gute Tabellen sollen nun in eine DB auf den Rechner. Verwenden dazu DBMS *‚SQLite‘*.

Verwenden Datenbanksprache SQL (structured query language). Grobe Aufteilung in vier Bereiche: DDL, DML, TCL, DCL.

Allgemeines

Müssen Tabellen erzeugen, lesen, verändern, löschen (CRUD).

Müssen Datenätze erzeugen, lesen, verändern, löschen (CRUD).

Neue *‚Abfragesprache'*: SQL (structured query language). Aufgeteilt in:

1. Data Definition Language — DDL
   Befehle, um *Tabellen* und verwandte Elemente zu erzeugen, zu ändern oder zu löschen:
   `CREATE, ALTER, DROP` *Machen wir.*
2. Data Manipulation Language — DML
   Befehle, um *Datensätze* auszuwählen, zu erzeugen, zu ändern und zu löschen:
   `SELECT, INSERT, UPDATE, DELETE` *Machen wir.*
3. Transaction Control Language – TCL
   Befehle, um DB-Manipulationen endgültig in die Datenbank zu übernehmen:
   `COMMIT, ROLLBACK` *Machen wir nicht.*
4. Data Control Language — DCL
   Befehle zur Berechtigungsvergabe von Lese- und Schreibrechten:
   `GRANT, REVOKE` *Machen wir nicht.*

SQLite unterstützt *nur* ein Subset von SQL:

ABORT ACTION ADD AFTER ALL ALTER ANALYZE AND AS ASC ATTACH AUTOINCREMENT BEFORE BEGIN BETWEEN BY CASCADE CASE CAST CHECK COLLATE COLUMN COMMIT CONFLICT CONSTRAINT CREATE CROSS CURRENT_DATE CURRENT_TIME CURRENT_TIMESTAMP DATABASE DEFAULT DEFERRABLE DEFERRED DELETE DESC DETACH DISTINCT DROP EACH ELSE END ESCAPE EXCEPT EXCLUSIVE EXISTS EXPLAIN FAIL FOR FOREIGN FROM FULL GLOB GROUP HAVING IF IGNORE IMMEDIATE IN INDEX INDEXED INITIALLY INNER INSERT INSTEAD INTERSECT INTO IS ISNULL JOIN KEY LEFT LIKE LIMIT MATCH NATURAL NO NOT NOTNULL NULL OF OFFSET ON OR ORDER OUTER PLAN PRAGMA PRIMARY QUERY RAISE REFERENCES REGEXP REINDEX RELEASE RENAME REPLACE RESTRICT RIGHT ROLLBACK ROW SAVEPOINT SELECT SET TABLE TEMP TEMPORARY THEN TO TRANSACTION TRIGGER UNION UNIQUE UPDATE USING VACUUM VALUES VIEW VIRTUAL WHEN WHERE

(siehe `http://www.sqlite.org/lang_keywords.html`)

Reicht uns aber. Mehr in MySQL oder PostgreSQL.

Schreibweisen:

- SQL-Befehle: alles in Großbuchstaben,
- Variablen: mit Kleinbuchstaben

CREATE TABLE

Erste Aufgabe: Erzeuge Tabellen.

```
CREATE TABLE table_name (
  column_name  column_type
  [,...]
);
```

`table_name` eindeutig innerhalb der Datenbank. `column_name` eindeutig innerhalb der Tabelle.

`*_name` prinzipiell zusammensetzbar aus *allen* Zeichen auf der Tastatur. Groß- und Kleinschreibung wird *nicht* unterschieden! Leerzeichen einklammern in `[]` oder `""`.

Bsp.:
`aTable`, `Verkaeufer1`, `meine_wichtige_tabelle`, `[my $ table]`, `"my §$!? table"`

Aber! Einfache Namen verwenden. Werden häufig in Abfragen verwendet.

Erzeugt nur Datenschema. Keine Einträge. (Später `INSERT INTO`.)

Spaltendatentyp `column_type` in SQLite:

- ▷ `INTEGER`: ganze Zahl, `1, -42`
- ▷ `REAL`: relle Zahl, `47.11, 0.815`
- ▷ `TEXT`: Zeichenfolge, `'Hallo', ' Liebe Leute!'`
- ▷ `BLOB`: binäre Daten (binary large object).

Andere SQL-Datentypen wie `CHAR(n)`, `VARCHAR(n)`, ...werden darüber ersetzt.

Bsp.:

$$\text{bsp} = \{\text{bsp\_id}: \text{integer},\ \text{titel}: \text{text},\ \text{wert}: \text{real}\}$$

```
CREATE TABLE bsp (
  bsp_id  INTEGER,
  titel   TEXT,
  wert    REAL
);

INSERT INTO bsp VALUES (1, 'Ein Titel', 0.0);
```

Modifikationen: Erzwingen von nichtleeren Einträgen.

```
CREATE TABLE table_name (
  column_name column_type NOT NULL
  [,...]
);
```

Bsp.:

```
CREATE TABLE bsp (
  bsp_id  INTEGER NOT NULL,
  titel   TEXT,
  wert    REAL NOT NULL
);
```

Einträge in die DB *ohne* Angabe von ‚bsp_id' und ‚wert' werden von dem DBMS abgelehnt. ‚titel' kann frei bleiben.

Modifikation: Vorgabe des Primärschlüssels.

```
CREATE TABLE table_name (
  column_name column_type PRIMARY KEY
  [,...]
);
```

Oder

```
CREATE TABLE table_name (
  column_name column_type NOT NULL,
  [,...]
  PRIMARY KEY (column_name[,...])
);
```

`PRIMARY KEY` implizit `NOT NULL`. Legt Primärschlüssel fest.

Garantiert Eindeutigkeit des Datensatzes. Weiterer Datensatz mit gleichem Primärschlüssel wird vom DBMS abgelehnt.

Bsp.:

MatrNr	Name	Fach

VorlNr	Titel	SWS

MatrNr	VorlNr

```
CREATE TABLE student (
  MatrNr INTEGER PRIMARY KEY,
  Name   TEXT NOT NULL,
  Fach   TEXT
);
```

```
CERATE TABLE vorlesung (
  VorlNr INTEGER PRIMARY KEY,
  Titel  TEXT NOT NULL,
  SWS    INTEGER NOT NULL
);
```

```
CERATE TABLE hoeren (
  MatrNr INTEGER NOT NULL,
  VorlNr INTEGER NOT NULL,
  PRIMARY KEY (MatrNr, VorlNr),
  FOREIGN KEY (MatrNr) REFERENCES student(MatrNr),
  FOREIGN KEY (VorlNr) REFERENCES vorlesung(VorlNr)
);
```


Modifikation: Vorgabe von Standardwerten.

```
CREATE TABLE table_name (
  column_name column_type DEFAULT value
  [,...]
);
```

Belegt Eintrag mit Wert ‚value‘, wenn kein Wert angegeben wurde.

Bsp.:

```
CREATE TABLE bsp (
  bsp_id  INTEGER PRIMARY KEY,
  titel   TEXT DEFAULT 'Unbekannt',
  wert    REAL NOT NULL
);

INSERT INTO bsp (bsp_id, wert) VALUES (2, 2.0);
```

Modifikation: Test der Eingabe. (Integritätsbedingungen)

```
CREATE TABLE table_name (
  column_name column_type CHECK check_expr
  [,...]
);
```

Prüft bei Eingaben Wert gegen bool'schen Ausdruck `bool_expr`.

Bsp.:

```
CREATE TABLE bsp (
  id      INTEGER PRIMARY KEY,
  titel   TEXT CHECK (titel <> 'Ein Titel'),
  wert    REAL NOT NULL
);

INSERT INTO bsp VALUES (2, 'Mein Titel', 2.0);
INSERT INTO bsp VALUES (3, 'Ein Titel', 3.0);
```

DBMS verweigert den zweiten Eintrag.

DROP TABLE

Zweite Aufgabe: Lösche Tabelle.

```
DROP TABLE table_name;
```

Löscht gesamte Tabelle ‚table_name'.

Modifikationen:

```
DROP TABLE table_name CASCADE;
```

Löscht alle Verweise in anderen Tabellen.

```
DROP TABLE table_name RESTRICT;
```

Verhindert löschen, wenn noch Verweise in anderen Tabellen.

ALTER TABLE

Man kann eine Tabelle auch verändern.

```
ALTER TABLE table_name alter_action;
```

`alter_action` steuert

- Umbenennen der Tabelle,
- Hinzufügen neuer Attribute/Spalten,
- Löschen von Attributen/Spalten,
- (Ändern von Eigenschaften von Attributen/Spalten),
- (Hinzufügen von Integritätsbedingungen),
- (Löschen von Integritätsbedingungen).

Umbenennen der Tabelle

```
ALTER TABLE table_name RENAME TO new_table_name;
```

Gibt Tabelle neuen Namen.

Bsp.:

```
ALTER TABLE bsp RENAME TO foo;
```

Hinzufügen neuer Attribute/Spalten

```
ALTER TABLE table_name ADD column_name column_type;
```

Ergänzt neue Spalte. Füllt bisherige Datensätze mit ‚leeren' Werten auf.

Bsp.:

```
ALTER TABLE bsp ADD subtitle TEXT;
```


Löschen von Attributen/Spalten

```
ALTER TABLE table_name DROP column_name;
```

Löscht Spalte in Tabelle. Möglicherweise mit Nebenwirkung auf andere Tabellen.

Bsp.:

```
ALTER TABLE bsp DROP subtitle;
```

Nicht in SQLite.

Änderungen problematisch, da

- Relationen zwischen Tabellen,
- Umgang mit inkompatiblen Änderungen,
- Auswirkungen auf externe Anwendungen,
- ...

Guter Datenbankentwurf umso wichtiger.

INSERT INTO

Einfügen von Datensätzen in eine Tabelle. In zwei Formen

1. Einfügen von konstanten Werten,
2. Einfügen von Abfrageergebnissen auf anderen Tabellen. (Über SELECT)

Datenbankschema für Beispiele

```
CREATE TABLE bsp (
  id      INTEGER PRIMARY KEY,
  titel   TEXT DEFAULT 'Unbekannt',
  wert    REAL
);
```

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

Einfügen über Angabe von Spalte und Wert.

Bsp.:

```
INSERT INTO bsp (id, titel, wert) VALUES (1, 'mein titel',
1.0);
```

Fügt den Datensatz `(1, 'mein titel', 1.0)` ein.

```
INSERT INTO bsp (id, wert) VALUES (2, 2.0);
```

Fügt den Datensatz `(2, 'Unbekannt', 2.0)` ein. Default Wert für `titel`.

```
INSERT INTO bsp (id) VALUES (3);
```

Fügt den Datensatz `(3, 'Unbekannt', NULL)` ein.

Bsp.:

```
INSERT INTO bsp (id, titel, wert) VALUES (1, 'noch ein titel',
42.0);
```

Geht schief, da `id` eindeutig sein muss. Wert `id=1` schon vergeben.

Lässt man die Spalten weg, dann muss man alle Werte in der `CREATE` Reihenfolge angeben.

```
INSERT INTO bsp VALUES (4, 'noch ein titel', 42.0);
```

Einfügen von Abfrageergebnissen, wenn wir den SELECT Befehl durchgesprochen haben.

UPADTE

Verändern von Datensätzen in einer Tabelle.

```
UPDATE table_name
  SET column_name  = new_value [,...]
  [WHERE where_expr];
```

In *allen* Tupeln der Tabelle `table_name`, die die `where_expr` erfüllen (falls angegeben), werden die Attributwerte der Spalte `column_name` auf den neuen `new_value` Wert gesetzt.

Dabei kann `new_value` aus einem `SELECT` Befehl kommen. (Später)

Bsp.:

```
UPDATE bsp SET title = 'Mein Titel!!!';
```

Setzt in *allen* Datensätzen den Eintrag `titel` auf den Wert ‚Mein Titel!!!'. Eher nicht erwünscht.

```
UPDATE bsp SET wert = wert + 0.815;
```

Erhöht in *allen* Datensätzen den Eintrag `wert` um 0.815. Wenn nicht NULL.

```
UPDATE bsp SET title = 'foo'
  WHERE id == 1;
```

Setzt für den einen ausgewählten Datensatz den Titel neu.

```
UPDATE bsp SET title = 'Blabla'
  WHERE id > 1;
```

Setzt für die ausgewählten Datensätze den Titel neu.

Also aufpassen, da mehrere Datensätze mit einem `UPDATE` verändert werden können!

DELETE

Löschen von Datensätzen in einer Tabelle.

```
DELETE FROM table_name
  [WHERE where_expr];
```

Löscht *alle* Datensätze aus der Tabelle `table_name`, welche die `where_expr` erfüllen (falls angegeben).

```
DELETE FROM bsp;
```

Löscht *alle* Datensätze aus der Tabelle `bsp`. Eher nicht erwünscht.

```
DELETE FROM bsp WHERE id == 1;
```

Löscht den Datensatz mit `id = 1` aus der Tabelle `bsp`.

SELECT

```
SELECT * FROM table_name;
```

Wählt *alle* Datensätze der `table_name` aus.

Komplizierte Abfragen kommen, wenn wir ‚*Relationenalgebra*' verstanden haben.