Computergestützte Methoden

Praktische Datenhaltung und -verarbeitung

Meik Teßmer

...

Inhalt der Veranstaltung

- Entwurf einer Datenbank-Struktur
- Einführung in die Sprache SQL
- Beispielanwendung

Datenbankmanagementsystem (DBMS)

- Menge aller Datensätze == *Datenbank*
- DBMS verwaltet eine oder mehrere Datenbanken:
  - Datenbank einrichten
  - Tabellen erzeugen/modifizieren/löschen
  - Datensätze hinzufügen/modifizieren/löschen
  - Datensätze abfragen
- bei relationalen DBMS: Kontakt erfolgt über *Structured Query Language* (SQL)

Datenbankmanagementsystem

Beispiel-DBMS: SQLite

- „eingebettetes“ DBMS, keine Konfiguration nötig
- keine Client-Server-Struktur
- nutzt normale Dateien für die Speicherung
- eine Datenbank entspricht *einer* Datei
- lizenzfrei (Public Domain)
- läuft auf vielen Plattformen
- Einsatzbereiche:
  - Firefox, Safari
  - iTunes, iPhone/iPod
  - Skype, Dropbox

Entwurf einer Datenbank-Struktur

Vorüberlegungen zu den zu speichernden Daten:

- Tabellen
- Attribute dieser Tabellen
- Schlüsselkandidaten
- Fremdschlüssel-Attribute

Vorüberlegungen

Datum	*Station*	*count*
2022-01-01	10th & E St NW	1
2022-01-01	10th & Florida Ave NW	24
2022-01-01	10th & G St NW	11
...	...	...

Hilfsmittel: Markierungen von Schlüsselattributen und Schlüsseln

- Schlüsselkandidaten der Tabelle Verleih: explizite Id, Kombination Datum udn Station
- Idee für Entwurf: Stationen in separate Tabelle auslagern und über Fremdschlüssel in Verleih-Tabelle anbinden

Vorüberlegungen: Entwurf

Struktur einer Tabelle heißt auch *Schema*

Schemata für Mitglieder und Status:

- **Verleih** (Id#, Datum, StationId#, count)
- **Stationen** (Id#, Name)

Vorüberlegungen: Normalformen

- 1. Normalform (1NF): Trennung nicht-atomarer Attribute, bspw. Name $\rightarrow$ passt
- 2. Normalform (2NF): Auftrennung in mehrere Tabellen + Fremdschlüssel-Beziehungen mit passenden Abhängigkeiten

Normalisierung 2NF

ist grundsätzlich schon gegeben, eine Auslagerung der Stationsnamen erleichtert aber spätere Anpassungen

Id	Datum	StationId	count
1	2022-01-01	1	11
2	2022-01-01	2	8
3	2022-01-02	2	4
4	2022-01-01	3	9
.	...	...	...

Id	Station
1	10th & G St NW
2	10th & K St NW
3	10th & Monroe St NE
.	...

Nächster Schritt: Übersetzung in SQL

- SQL (Structured Query Language): verbreitete Abfragesprache für relationale Datenbanken
- Groß-/Kleinschreibung ist bei SQL-Anweisungen egal
- Anweisungen mit einem Semikolon abschließen
- SQL besteht aus zwei Teilen:
  - DML: Data Manipulation Language
  - DDL: Data Definition Language
- Kommentarzeilen: `-- Kommentar...`

Schritt 1: Übersetzung in SQL

- DDL-Teil:
  - `CREATE DATABASE` - Datenbank erzeugen
  - `ALTER DATABASE` - Datenbank modifizieren
  - `CREATE TABLE` - Tabelle erzeugen
  - `ALTER TABLE` - Tabelle modifizieren
  - `DROP TABLE` - Tabelle löschen
  - `CREATE INDEX` - Index erzeugen
  - `DROP INDEX` - Index löschen
- DML-Teil:
  - `SELECT` - Datensätze suchen
  - `UPDATE` - Datensätze modifizieren
  - `DELETE` - Datensätze löschen
  - `INSERT INTO` - Datensätze einfügen
  - `UPSERT` - Datensätze einfügen/aktualisieren

Schritt 1: Übersetzung in SQL

Vorgehensweise:

1. Datenbank erzeugen
2. Tabellen erzeugen
3. Datensätze einfügen
4. Abfragen erstellen

Schritt 1.1: Datenbank erzeugen

Datenbanken werden erstellt mit dem Befehl

```
CREATE DATABASE database_name;
```

Schritt 2.1: Tabellen erzeugen

Tabelle werden erzeugt mit

```
CREATE TABLE tabellen_name (
    attribut1 datentyp,
    attribut2 datentyp,
    attribut3 datentyp,
    ...
);
```

Datentypen: INTEGER, REAL, TEXT, BLOB, ...

Schritt 2.2: Tabelle Mitglieder erzeugen

```
CREATE TABLE verleih (
  id            INTEGER,
  datum         TEXT,
  station       TEXT,
  count         INTEGER,
  ...
);
```

Wie erzeugt man Schlüssel und stellt deren Eigenschaft sicher?

Schritt 2.2: Tabellen und Schlüssel

Attribut zum Primärschlüssel machen: `PRIMARY KEY`

→ für Mitglieder und Status heißt das:

```
CREATE TABLE stationen (
    id        INTEGER PRIMARY KEY,
    name      TEXT
);

CREATE TABLE verleih (
    id            INTEGER PRIMARY KEY,
    datum         TEXT,
    stationid     INTEGER,
    count         INTEGER,
    ...
);
```

Schritt 1.2: Tabellen und Fremdschlüssel

- Tabelle `verleih` nutzt Fremdschlüssel `stationid` → DBMS darüber „informieren“, um *Integrität* sicherzustellen
- Syntax ist abhängig von DBMS

```
CREATE TABLE stationen (
    id          INTEGER PRIMARY KEY,
    name        TEXT
);

CREATE TABLE verleih (
    id              INTEGER PRIMARY KEY,
    datum           TEXT,
    stationid       INTEGER,
    count           INTEGER,

    FOREIGN KEY(stationid) REFERENCES stationen(id)
);
```

→ *referenzielle Integrität* wird sichergestellt

Schritt 2.2: Integritätssicherung aktivieren

→ verhindert versehentliches Löschen von schlüsselrelevanten Attributwerten

```
PRAGMA foreign_keys = ON;              ← Aktivierung
CREATE TABLE stationen (
    id        INTEGER PRIMARY KEY,
    name      TEXT
);

CREATE TABLE verleih (
    id            INTEGER PRIMARY KEY,
    datum         TEXT,
    stationid     INTEGER,
    count         INTEGER,

    FOREIGN KEY(stationid) REFERENCES stationen(id)
);
```

Was ist mit leeren Attributwerten?

Schritt 2.2: Verhindern von leeren Attributwerten

Constraints beschreiben erlaubte Attributwerte

→ Datum und Stationsname dürfen nicht leer sein:

```
CREATE TABLE verleih (
    id              INTEGER PRIMARY KEY,
    datum           TEXT NOT NULL,
    ...

);
```

Zwischenschritt: Datenbank anlegen

- auf der Kommandozeile: `sqlite3 verleih.db`
- Informationen erhalten
  - Datenbank: `.databases`
  - Tabellen: `.tables`
  - Schema einer Tabelle: `.schema`
- DBMS verlassen: `.quit`

Schritt 2: Tabellen erzeugen

Variante 1: per Hand

Tabellen per Hand erzeugen

Variante 2: aus Datei lesen

Tabellen aus SQL-Datei erzeugen lassen:

```
.read verleih.sql
```

Schritt 3: Datensätze eingeben

SQL-Befehl lautet `INSERT INTO...`

- kompletter Datensatz:

```
INSERT INTO table_name
  VALUES (value1, value2, value3,...);
```

- teilweise: Attribute müssen angegeben werden

```
INSERT INTO table_name (column1, column2,...)
  VALUES (value1, value2,...)
```

→ `INSERT INTO stationen VALUES(1, '10th & E St NW');`

Schritt 3: Datensätze eingeben

- per Hand mühsam
- Alternative: Daten liegen im CSV-Format vor:

```
1,'10th & E St NW'
2,'10th & Florida Ave NW'
...
```

- Einlesen: `.import FILE TABLE`
- `.show` zeigt das gewählte Trennzeichen an (Default: |)
- ändern mit `.separator ,`

```
.separator ,
.import stationen.csv stationen
```

wichtig: ID-Werte müssen hierbei mit angegeben werden!

Schritt 4: Abfragen formulieren

- Syntax von Abfragen in SQL

```
SELECT * from <TABLE>; -- komplette Tabelle ausgeben
```

- Beispiel: `SELECT * FROM status;`
- schönere Ausgabe (SQLite):

```
.mode column
.headers on
```

- Liste aller Stationen mit allen hinterlegten Attributen:

```
SELECT * FROM stationen;
```

Schritt 4: Abfragen aus dem relationalen Modell umsetzen

- Projektion:

```
SELECT (attribut, ...) FROM table;
```

- Beispiel: Liste aller Stationsnamen

```
SELECT name FROM stationen;
```

- Restriktion (`WHERE`-Klausel):

```
SELECT * FROM table
  WHERE attributname=wert;
```

- Beispiel: Liste aller Stationen mit `count>5`:

```
SELECT * FROM verleih
  WHERE count>5;
```

Schritt 4: Abfragen aus dem relationalen Modell formulieren

- mehrfach auftretende Datumsangaben bei Projektion:

```
SELECT datum  FROM verleih;
```

- Lösung: `SELECT DISTINCT`:

```
SELECT DISTINCT datum FROM verleih;
```

- Projektion und Restriktion können kombiniert werden

```
SELECT DISTINCT datum, count FROM verleih
  WHERE count>5;
```

Schritt 4: Abfragen formulieren - Verbesserungen

- unschön: IDs im Ergebnis an Stelle des Stationsnamens
- Lösung: *Join* zweier Tabellen
- per Hand
  - Angabe des Tabellennamens bei Attributen
  - Angabe der beteiligten Tabellen
- Beispiel: Liste mit Nachname und Status:

```
SELECT verleih.datum, stationen.name,verleih.count
    FROM verleih, stationen
    WHERE verleih.stationid=stationen.id;
```

- heißt auch *Inner Join*

Schritt 4: Abfragen formulieren - Verbesserungen

- Inner Joins werden von den meisten DBMS direkt unterstützt
- Abfrage mit `INNER JOIN`:

```
SELECT verleih.datum, stationen.name,verleih.count
    FROM (verleih INNER JOIN stationen
      ON verleih.stationnr=stationen.id)
WHERE verleih.count>5;
```

- Vorteil: potenzielle WHERE-Klausel ist besser zu formulieren
- Natural Join?
  - ist quasi ein Inner Join, aber mit möglichen Seiteneffekten
  - besser: explizite Angabe zu den Join-Attributen als automatisch mit Natural Join

Schritt 4: Abfragen formulieren - Vereinfachungen

- hilfreich: Abfragen können vorbereitet und gespeichert werden
  $\rightarrow$ *View*
- sehen aus wie eine Tabelle $\rightarrow$ `.tables`
- erstellen mit

```
CREATE VIEW 'Stationen auflisten' AS
  SELECT name FROM stationen;
```

- Restriktionen und alle anderen Anweisungen sind möglich

Export von Daten

- Ausgabe in Datei umlenken:

```
.output FILENAME
```

- Dump der gesamten Datenbank (Backup):

```
.output backup.sql
.dump
```

→ in `backup.sql` steht nun alles

Zugriff auf SQLite-Datenbanken mit Python

- Modul `sqlite3`
- Verbindung herstellen:

```
import sqlite3
conn = sqlite3.connect("/tmp/verleih.db")
# Verbindung schließen
conn.close()
```

- SQL-Anweisungen ausführen:

```
cursor = conn.cursor()
# Tabelle erzeugen
cursor.execute("""CREATE TABLE stationen (
  id integer primary key,
  name)""")
# Änderungen in der DB speichern
conn.commit()
```

Zugriff auf SQLite-Datenbanken mit Python

- Daten einfügen:

```
cursor.execute("""INSERT INTO stationen
  VALUES(1,'10th & E St NW')""")
cursor.execute("""INSERT INTO "stationen"
  VALUES(2,'10th & Florida Ave NW')""")
```

- parametrisierte Form:

```
id_, station = 1, "10th & E St NW"
cursor.execute("""INSERT INTO stationen (id,name)
  VALUES('%d',%s)""" % (id_, name))
```

→ klassische String-Ersetzung; gefährlich!

Angriff mittels SQL-Injection

auch das ist eine mögliche Eingabe:

```
10th & E St NW'); DROP TABLE verleih; --
```

Ergebnis:

```
cursor.execute("""INSERT INTO stationen (id, name)
  VALUES('10th & E St NW'); DROP TABLE verleih; --')""")
```

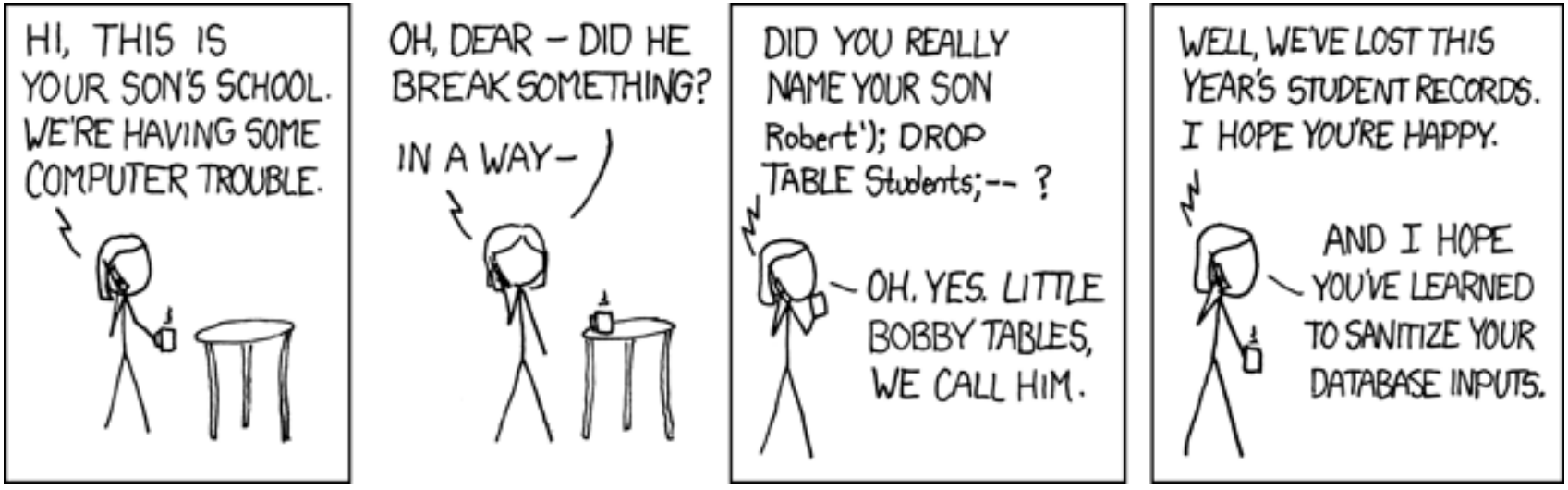

Figure 1: XKCD-Comic

SQL-Injection vermeiden

- Parameter-Substitution der DB-Schnittstelle nutzen:

```
cursor.execute("""INSERT INTO "stationen"
  VALUES(?, ?)""" % (id_, station))
```

- Daten abrufen und verarbeiten:

```
cursor.execute("SELECT * FROM stationen")
for row in cursor:
  print(row)
```

Zusammenfassung

- Entwurf und Übertragung der Daten in eine Datenbank
- technische Umsetzung des Relationenmodells
- programmierter Zugriff auf Daten

Ende

- Informationen zu SQLite: Hipp (n.d.)
- SQL-Tutorial: “SQL-Tutorial” (n.d.)

Literatur

Hipp, Richard D. n.d. “SQLite Home Page.” Accessed November 14, 2017. https://sqlite.org/index.html.

“SQL-Tutorial.” n.d. 1keydata.com. Accessed November 14, 2017. https://www.1keydata.com/de/sql/.