Relationale Anfragesprachen

Relationale Anfragesprachen

Geschichte

- Anfang 70er Jahre: IBM Prototyp System R, Anfragesprache SEQUEL
- Später Umbenennung in SQL
- Anfang 80er Jahre: SQL/DS
- Später: IBM DB2, Oracle, MS SQL-Server, ...
- 1986: ANSI, erste SQL Spec
- 1989: Revidierung
- 1992: SQL-92 = SQL2
- Ende 90er Jahre: Erweiterung SQL-99 = SQL3
- Spracherweiterung SQL-2003 um XML-Integration
- ...
- SQL-2023: Property graph queries, JSON

- 3 fundamantale Datentypen: Zahlen, Zeichenketten und Datumstypen
- Viele Schreibweisen
- **date**: Format JJJJ-MM-TT
- **character(n)**: Feste Größe n, aufgefüllt mit Leerzeichen
- **char varying(n)**: Dynamische Größe, aber n ist Maximum
- Abkürzung zu **char** oder **varchar**
- **numeric(p, s)**: Allg. Zahlentyp, (p, s) optional, p = Gesamtzahl Stellen, s = Anzahl Nachkommastellen
- Innerhalb der geg. Präzision ist numeric exakt
- **integer** oder **int**: Zahlen ohne Nachkommastellen
- **float**: Angenäherte Zahlen
- **blob**, **raw**: große binäre Daten, vom DBMS nicht zu interpretieren
- **xml**: XML-Dokumente

Definition einer neuen Tabelle mit **create table**

```
create table Dozenten
(   PersNr    integer not null,
    Name      varchar(10) not null,
    Rang      character(2)
);
```

Hier ist die Angabe eines Primärschlüssels nicht notwendig, da PersNr und der Name nicht null sein dürfen. Somit ist die Integrität gewahrt.

Definition einer neuen Tabelle mit **create table** und **Primärschlüssel** in **MySql**:

```
create table Dozenten
(   PersNr    integer not null,
    Name      varchar(10) not null,
    Rang      character(2),
    primary key (PersNr)
);
```

In **SQLite, MS SQL-Server, Oracle, MS Access**:

```
create table Dozenten
(   PersNr    integer not null primary key,
    Name      varchar(10) not null,
    Rang      character(2)
);
```

Ändern einer Tabelle mit **alter table**:

```
alter table Dozenten
    add (Raum integer);

alter table Dozenten
    modify (Name varchar(30));
```

In **SQL-92**:

```
alter table Dozenten
    add column Raum integer;

alter table Dozenten
    alter Column Name varchar(30);
```

SQLite unterstützt modify nicht, daher muss man sich mit anderen Mitteln behelfen:

```
alter table Dozenten
    drop column Name;

alter table Dozenten
    add column Name varchar(30);
```

Mit **insert into** können Zeilen (Tupel) in Tabellen eingefügt werden:

```
insert into Dozenten
    values (11235813, 'Fibonacci', 'C4', 112);
```

- Die Werte müssen in der exakten Reihenfolge der Schemadefinition eingegeben werden
- Natürlich können Datensätze auch bedingt gefiltert und eingefügt werden.
- Dazu muss aber erst das **select**-Statement besprochen werden

Allgemeine Form:

```
select <Spalte>,<Spalte>, ... from
       <Tabelle>,<Tabelle>,... where
       <Bedingung>;
```

- Bedingung = Selektionsprädikat
- Falls mehrere Tabellen angegeben werden, wird das Kreuzprodukt durchgeführt und
- die Daten müssen sinnvoll gefiltert werden.
- Faustregel: Bei n Tabellen sollten mindestens n-1 Filterkriterien gefunden werden.[1]
- Beispiel: ... where pruefung.teilnehmer = studierender.matrnr;

[1] Erfahrungswert des Dozenten

Allgemeine Form mit Sortierung:

```
select <Spalte>,<Spalte>, ... from
       <Tabelle>,<Tabelle>,... where
       <Bedingung>
       order by <Spalte> asc | desc;
```

- asc = Aufsteigend, desc = Absteigend
- Es können auch mehrere Sortierkriterien angegeben werden
- Beispiel: ... order by Name desc, Rang asc;

Relationale Anfragesprachen
SQL Anfragen - Beispiel

```
select Name, Titel
from Dozenten, Vorlesungen
where PersNr = gelesenVon
    and Titel = 'Mathematik␣I';
```

1. Bilden des Kreuzprodukts der beiden Tabellen
2. Jede Zeile wird auf Erfüllen der Bedingung aus dem where-Teil überprüft und bei Zutreffen ausgewählt
3. Projektion auf die im select-Teil angegebenen Attribute

Ausdruck in Relationenalgebra:

$$\Pi_{Name, Titel}(\sigma_{PersNr=gelesenVon \wedge Titel='MathematikI'}(Dozenten \times Vorlesungen))$$

Tupelvariablen:

```
select s.Name, v.Titel
from Studierende s, Vorlesungen v, hoeren h
where s.MatrNr = h.MatrNr
    and h.VorlNr = v.VorlNr;
```

- Zur Vereinfachung und bei mehrfacher Verwendung von Tabellen nützlich

Aggregatfunktionen und Gruppierung:

- Aggregatfunktionen führen Operationen auf Tupelmengen durch und
- komprimieren eine Menge von Werten zu einem einzelnen Wert
- Typische Vertreter: avg, sum, min, max, count

```
select avg(Semester)
from Studierende;

select gelesenVon, sum(SWS)
from Vorlesungen
group by gelesenVon;
```

- Das erste Beispiel berechnet die durchschnittliche Semesterzahl aller Studierender
- Unten wird die Gesamtzahl der SWS aller Vorlesungen eines Dozenten gebildet

Aggregatfunktionen und Gruppierung:

- Mit der `having` Klausel kann bei `group by` noch eine zusätzliche Bedingung gestellt werden.

```
select gelesenVon, sum(SWS)
from Vorlesungen
group by gelesenVon
    having avg(SWS) >= 3;
```

- Es werden alle Tupel mit geleichem Wert im angegebenem Attribut `gelesenVon` zusammengefasst und für
- jede der entstandenen Gruppen die Summe der SWS gebildet
- Die having-Klausel filtert alle Dozenten mit längeren Vorlesungen

Aggregatfunktionen und Gruppierung:

- Der unterschied zwischen `where` und `having`

```
select gelesenVon, Name, sum(SWS)
from Vorlesungen, Dozenten
where gelesenVon = PersNr and Rang = 'Prof.'
group by gelesenVon, Name
    having avg(SWS) >= 3;
```

- Bei `select` (mit group by!) düfen nur Aggregatfunktionen oder Attribute, nach denen gruppiert wurde, verwendet werden. Aus diesem Grund steht der Name mit in der `group by`-Klausel

Geschachtelte Anfragen:

- Suchen aller Prüfungen, die schlechter als der Durchschnitt sind

```
select *
from pruefen
where Note > (select avg(Note)
              from pruefen);
```

- * wird verwendet, wenn alle Attribute ausgegeben werden sollen
- $=, >, <$ wird nur verwendet, wenn die Anfrage einen skalaren Wert liefert
- SQL schreibt vor, dass Unteranfragen immer geklammert werden müssen

Geschachtelte Anfragen:

- Unteranfragen in der Attributangabe sind möglich

```
select PersNr, Name, (select sum(SWS) as Lehrlast
                      from Vorlesungen
                      where gelesenVon = PersNr)
from Dozenten;
```

- Lehrbelastung wird zum Attribut in der Ausgabetabelle
- PersNr (aus Dozenten) ist innerhalb des inneren Selects gültig

Exists (korrelierte Unteranfrage):

- Ein/e Studierende/r ist älter als ein Dozent, wenn das Geburtsdatum des Dozenten größer ist (kurz drüber nachdenken...)

```
select s.*
from Studierende s
where exists
    (select d.*
     from Dozenten d
     where d.gebDatum > s.gebDatum);
```

- Der exists-Operator liefert true, wenn die Unteranfrage mindestens ein Ergebnistupel beinhaltet
- Das bedeutet, für jeden/jede Studierende/n wird die Unteranfrage separat ausgewertet
- Die Anfrage liefert alle Studierenden, die älter als der jüngste Dozent sind

Frage:

Ist die obige Anfrage effizienter/schlechter als folgendes Konstrukt?

```
select s.*
from Studierende s
where s.gebDatum <
    (select max(d.gebDatum)
     from Dozenten d);
```

Diese Anfrage ist unkorreliert

with-Klausel:

- Generieren von temporären Sichten für die Dauer der Anfragebearbeitung

```
with h as (
       select VorlNr, count(*) as AnzProVorl
       from hoeren
       group by VorlNr),

     g as (
       select count(*) as GesamtAnz
       from Studierende)

select h.VorlNr, h.AnzProVorl, g.GesamtAnz,
       cast(h.AnzProVorl as decimal(8,2)) /
       g.GesamtAnz
       as Marktanteil
from h, g;
```