

UNIT 0X0B

SQL II

Hintergrund

Einordnung der Kapitel SQL I und II

- Die praktischen Übungen zu SQL und etwas Hintergrund dazu gibt es regulär im SQL-Praktikum.
- Die Folien hier greifen einzelne Aspekte und Variationen etwas breiter auf, dennoch sind sie kein Nachschlagewerk. Dazu bemühe man die aktuelle Dokumentation der jeweiligen DBMS.

Übersicht

Themen

- Tabellen anlegen und löschen
- Bedingungen in Tabellen (Constraints)
- Domains
- Daten einfügen, modifizieren und löschen
- Sichten
- Generalisierung
- Datenintegrität
- Kaskadierende Operationen
- Check
- Assertions
- Stored Procedures
- Trigger
- Beispieldatenbank Kemper/Eickler

Anlegen von Tabellen

- Umfangreiche Möglichkeiten, hier nur einzelne Beispiele.
- Bedingungen/Constraints können an Spalten oder an die Tabelle gestellt werden.

```
create table table_name [ if not exists ] (
    { column_name data_type {column_constraint}* }
    {, column_name data_type {column_constraint}* }*
);
```

B Anlegen der Tabelle Professor

- Wir beschränken uns zunächst auf die korrekten Datentypen

```
create table Professoren (
    PersNr integer,
    Name varchar(50),
    Rang char(2),
    Raum integer
);
```

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Anlegen von Tabellen – Constraints

Bedingungen an eine Spalte

- primary key
Spalte ist Primärschlüssel, impliziert unique und not NULL.
- references ref_table [(ref_column)]
Spalte ist Fremdschlüssel, verweist auf Spalte ref_column in Tabelle ref_table.
- not null
NULL ist nicht erlaubt.
- unique
Spaltenwert einmalig, NULL ist erlaubt.
- check (expression)
Für einzufügende Werte muss expression true oder NULL liefern.
- default default_expr
Wird beim Einfügen einer neuen Zeile kein Wert für diese Spalte angegeben, so wird sie mit default_expr vorbelegt.

Anlegen von Tabellen – Beispiel

B Eine verbesserte Professorentabelle

- **PersNr** ist der Primärschlüssel
- Für **Name** ist NULL nicht erlaubt
- Nur ein Professor pro **Raum**
- **Rang** ist C2,C3 oder C4 - default ist C2

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

```
create table Professoren (
    PersNr integer primary key,
    Name varchar(50) not null,
    Rang char(2) check( Rang in ['C2','C3','C4']) default 'C2',
    Raum integer unique
);
```

Anlegen von Tabellen – Beispiel

B Die Tabelle "Vorlesungen"

- **VorlNr** ist der Primärschlüssel
- Für **Titel** ist NULL nicht erlaubt
- **SWS** liegt zwischen 1 und 8
- **gelesen_von** ist Fremdschlüssel in Professoren-Tabelle

Vorlesungen			
VorlNr	Titel	SWS	gelesen_von
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

```
create table Vorlesungen (
    VorlNr integer primary key,
    Titel varchar(50) not null,
    SWS integer check( SWS between 1 and 8 ),
    gelesen_von integer references Professoren // (PersNr) optional, da Primärschlüssel
);
```

Anlegen von Tabellen - Table Constraints

Motivation

- Zusammengesetzte Schlüssel definieren.
- check Constraint über mehrere Spalten.
- unique für Tupel.

- **Syntax**

```
create table table_name [ if not exists ] (
    { column_name data_type {column_constraint}* }
    {, column_name data_type {column_constraint}* }*
    {table_constraint}
);
```

Anlegen von Tabellen – Beispiel Table Constraints

B Die Tabelle "Vorlesungen" mit *table constraints*

```
create table Vorlesungen (
    VorlNr integer primary key,
    Titel varchar(50) not null,
    SWS integer check( SWS between 1 and 8 ),
    gelesen_von integer references Professoren // (PersNr) optional, da Primärschlüssel
);
```

```
create table Vorlesungen (
    VorlNr integer,
    Titel varchar(50),
    SWS integer,
    gelesen_von integer,
    // table constraints:
    primary key( VorlNr ),
    foreign key( gelesen_von ) references Professoren,
    check( SWS between 1 and 8 )
);
```

Anlegen von Tabellen – Beispiel Table Constraints

B Die Tabelle "voraussetzen"

- **(Vorgänger,Nachfolger)** ist zusammengesetzter Primärschlüssel
- **Vorgänger** und **Nachfolger** sind Fremdschlüsselelemente mit Verweis in die Tabelle Vorlesungen

voraussetzen	
<u>Vorgänger</u>	<u>Nachfolger</u>
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

```
create table voraussetzen (
    Vorgänger integer references Vorlesungen, // column constraint
    Nachfolger integer references Vorlesungen, // column constraint
    primary key (Vorgänger, Nachfolger) // table constraint
);
```

Löschen von Tabellen

- **Syntax zum Löschen**

`drop table table_name;`

PostgreSQL: `drop table [if exists] table_name;`

Löscht eine Tabelle und alle zugehörigen Indizes

- **Syntax zum Leeren - Tabelle selbst bleibt erhalten**

`delete from table_name;`

PostgreSQL: `truncate table_name;` als schnellere Alternative

Domain

- Eine Domain beschreibt die Einschränkung eines Basistyps durch eine oder mehrere Bedingungen.
- Kann z.B. bei `create table` überall dort eingesetzt werden, wo Typ erwartet wird. Aber: definiert keinen echten neuen Typ, sondern entspricht vom Typ her immer dem Basistyp `data_type`.

- **Syntax:**

```
create domain domain_name as data_type  
[ default expression ]  
[{not null | null | check( expression )}]
```

Wert in `expression` wird mit **value** angesprochen

Beispiel Domain

B Professorentabelle mit eigener domain

- **PersNr** ist der Primärschlüssel
- Für **Name** ist NULL nicht erlaubt
- Nur ein Professor pro **Raum**
- **Rang** ist C2,C3 oder C4 - default ist C2

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

```
create domain ProfRang as char(2)
default 'C2'
check( value in ['C2','C3','C4'] )

create table Professoren (
    PersNr integer primary key,
    Name varchar(50) not null,
    Rang ProfRang,
    Raum integer unique
);
```

Einfügen von Daten

- **Einfügen einer Zeile:**
 - `insert into Studenten (MatrNr, Name)
values (28121, 'Archimedes');`
 - nicht angegebene Spalten werden mit *default*-Wert belegt (siehe **create table**, **create domain**), falls für zugehörigen Typ keiner bekannt mit NULL.
 - `insert into Studenten default values; // Neuer Student aus Vorgabewerten`
- **Einfügen mehrerer Zeilen**
 - `insert into Studenten // ohne Spaltenangabe: Werte für alle Spalten erforderlich
values (28121, 'Archimedes', 12), (25403,'Jonas',18), ...;`
- **Einfügen einer Tabelle:**
 - `insert into hören
select MatrNr, VorlNr
from Studenten, Vorlesungen
where Titel= 'Logik' ;`

Löschen und Ändern von Daten

- **Löschen**
 - **delete from Studenten where MatrNr = 197403** // Löschen eines Studenten
 - **delete from Studenten where Semester > 13;** // Löschen aller Studenten ab dem 14. Semester

- **Ändern**
 - **update Studenten set Nachname='von Neumann' where MatrNr=196704** // Student mit MatrNr.=196704 erhält neuen Namen
 - **update Studenten set Semester= Semester + 1;** // Alle Studenten erreichen nächstes Semester

Sichten

Idee

- Abstrahieren eine bestimmte 'Sicht' aus einer Rolle oder einem Anwendungsfall, d.h. in der Regel Ausschnitt oder bestimmte Aggregation aus Daten bzw. Modell.
- Nicht einfach zu entscheiden ist, ob Sicht besser in externer Business Logik oder im DBMS aufgehoben ist.
- Realität ist noch komplexer, z.B. Elasticsearch mit Beats.
- Manche Projekte verlangen ausschliesslich Sichten.

Sichten

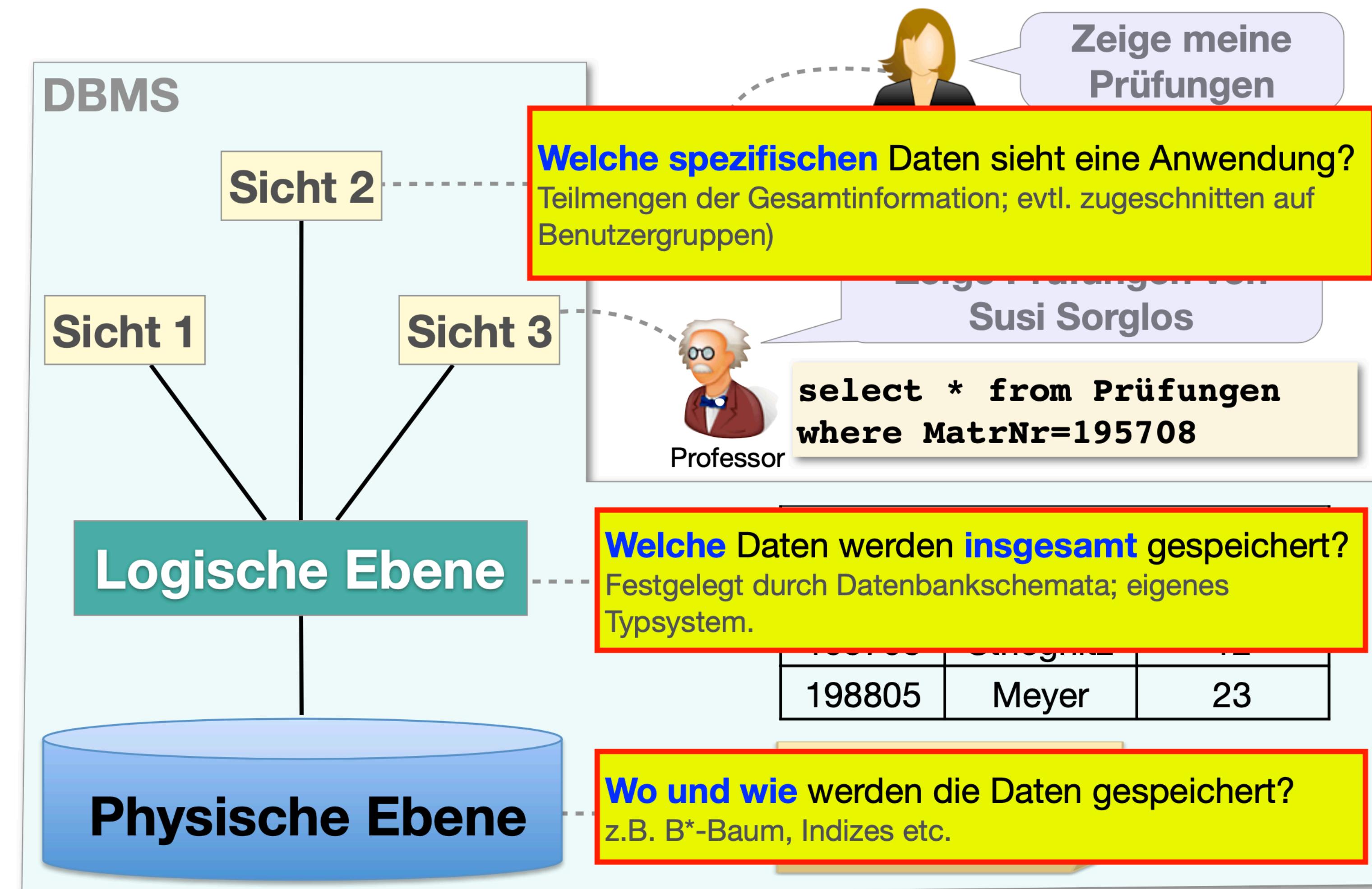
Temporäre Sichten

- im Kontext von with-Definitionen
- überdauern die Anfrage nicht

Statische Sichten

- Datenschutz
- Vereinfachung von Aufgaben
- Darstellung der Generalisierung

Sichten und Abstraktionsebenen



Sichten

Idee

Oftmals benötigen Anwender speziell zugeschnittenen Ausschnitt der Daten.

Externe Sichte

- sind Bestandteil der 3-Ebenen-Architektur.
- ermöglichen die Modellierung der Daten in einer für den Benutzer wünschenswerten Form.
- arbeiten auf den Daten der logischen Gesamtsicht. Somit sind entsprechende Transformationsregeln notwendig.

Sichten - Beispiel

- **Datenbank für Kinobetriebe: häufig abgefragt werden die Filme des aktuellen Tages**
- **In Datenbank sind pro Kino alle Filme mit Angabe von Titel, Saal und Uhrzeit abgelegt**
- **Es kann sinnvoll sein, den Zugriff über eine spezielle Tabelle**

**PROGRAMM_HEUTE (Name, Adresse, Telefon, Uhrzeit, Titel,
Regisseur)**

zu realisieren

Sichten - Beispiel

- **Möglichkeit:**
 - Deklaration einer **separaten Tabelle** und Befüllen mit der gewünschten Information:

```
create table PROGRAMM_HEUTE (
    Name  varchar(30) ,
    Adresse varchar(30),
    Telefon varchar(15),
    Uhrzeit timestamp,
    Titel  varchar(100) primary key,
    Saal   varchar(20)
)

insert into PROGRAMM_HEUTE
( select F.Name, F.Adresse, F.Telefon, F.Uhrzeit, F.Titel, F.Saal
  from  (PROGRAMM natural join KINO natural join FILM) as F
  where F.Datum = CURRENT_DATE )
)
```

- **Nachteile:** Redundante Datenhaltung, Tägliche Änderung der Tabelle

Sichten - Beispiel

- **Sinnvolle Alternative:**
 - Vereinbarung der „Tabelle“ PROGRAMM_HEUTE als **Sicht** (View)
- **Sicht entspricht Berechnung aus tatsächlich gespeicherten Relationen:**

```
create or replace view PROGRAMM_HEUTE as (
  select F.Name, F.Adresse, F.Telefon, F.Uhrzeit, F.Titel, F.Saal
  from   (PROGRAMM natural join KINO natural join FILM) as F
  where  F.Datum = CURRENT_DATE
)
```

Sichten

Effekt einer Sichtenvereinbarung

- DBMS speichert die Sichtdefinition, wertet sie aber nicht aus.
- Der Name der Sicht kann in Anfragen wie eine mit CREATE TABLE erzeugte Tabelle verwendet werden.
- Bei Zugriff auf die Sicht wird der Sichtename automatisch durch die mit der Sicht assoziierten Anfrage ersetzt. Die Sichtdefinition wirkt wie ein Makro

Sichten - Beispiel

- **Anfrageauswertung für die Sicht PROGRAMM_HEUTE:**

Anfrage eines Nutzers auf die Sicht PROGRAMM_HEUTE: „Welcher Film läuft im Atrium-Saal?“:

```
select Name  
from PROGRAMM_HEUTE  
where Saal = ,Atrium'
```

Tatsächliche Anfrage nach Auswertung der Sichtendefinition:

```
select PROGRAMM_HEUTE.Name  
from ( select F.Name, F.Adresse, F.Telefon, F.Uhrzeit, F.Titel, F.Name  
      from (PROGRAMM natural join KINO natural join FILM) as F  
      where F.Datum = CURDATE()  
    ) as PROGRAMM_HEUTE  
where Saal = ,Atrium';
```

Sichten - Beispiel

- **Beliebiges Anfrage zur Definition einer Sicht erlaubt**
 - Sichten können **mehrere Tabellen** verknüpfen
 - Sichten können Aggregatfunktionen nutzen
 - Sichten können auch Gruppierungen nutzen

```
create view PersonalkostenProStandort as
( select SUM(m.Gehalt) as Gehaltsbudget, m.Abteilungsnummer, a.Standort
  from Mitarbeiter as m natural join Abteilungen as a
  group by m.Abteilungsnummer, a.Standort
)
```

Sichten

Problem

- Änderungen müssen immer noch auf die Basisrelationen abgebildet werden. Dies ist je nach Sichtdefinition nicht immer möglich.

Beispiel

- Änderung der Uhrzeit einer Vorstellung in PROGRAMM_HEUTE ist unproblematisch: Änderung kann direkt in die benutzten Relationen eingebracht werden.
- Änderung des Datums eines Filmes in PROGRAMM_HEUTE entfernt Film aus der Sicht.

Allgemein

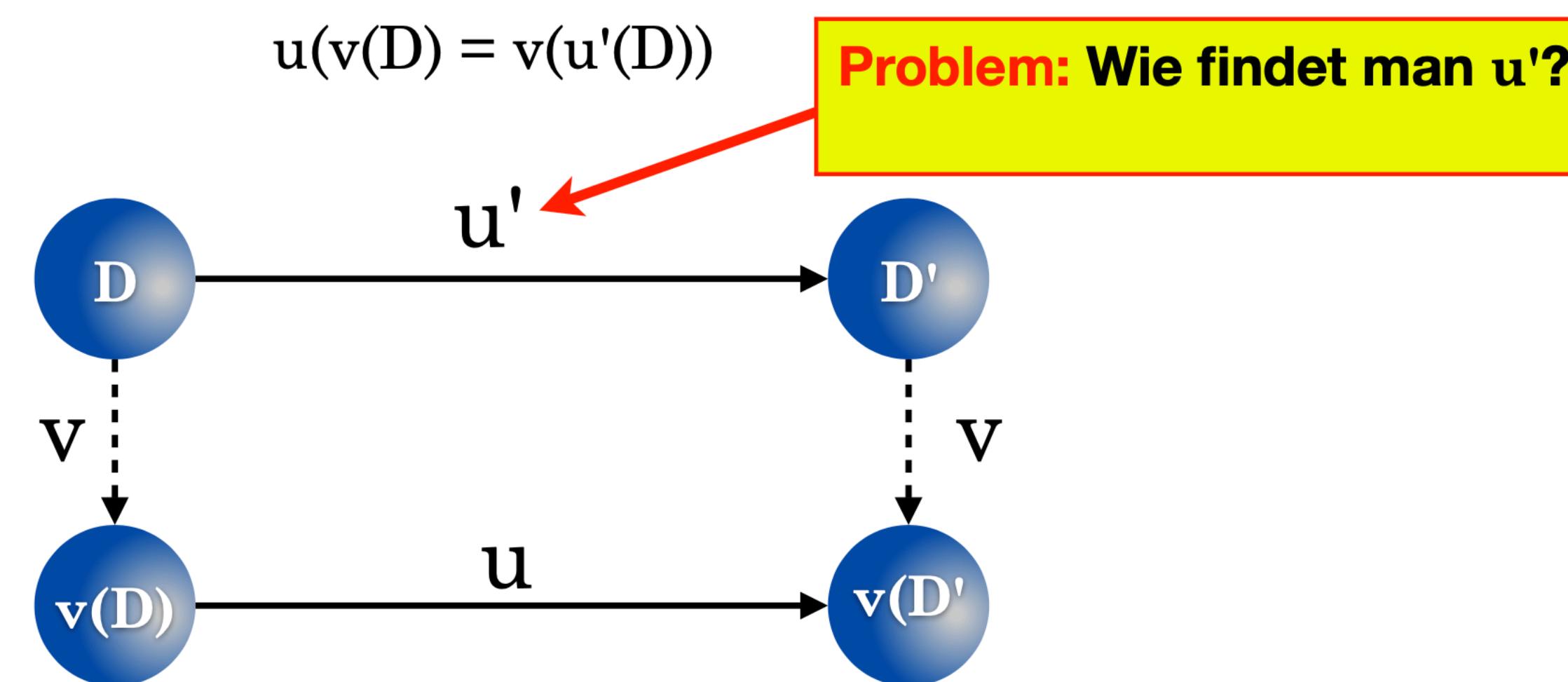
- Änderungen von Tupeln können ein „Löschen“ des Tupels aus der entsprechenden Sicht bewirken (Tupelmigration).

Sichten

- Sei D die Menge aller Belegungen einer Datenbank, dann können Sichten und Änderungen als partielle Funktionen betrachtet werden:

$$v:D \rightarrow D \text{ bzw. } u:D \rightarrow D$$

- wobei v eigentlich keine Änderung vornimmt
- Eine Sicht v ist durch ein Update u änderbar, wenn es ein äquivalentes Update u' für den Ausgangsdatenbestand $D \in D$ gibt, so dass



Sichten

- **Die eben genannte Definition reicht zur Entscheidungsfindung nicht aus, denn es fehlt an Kenntnis über u'**
 - von DBMS ist u' nicht automatisch bestimmbar
- **Stattdessen: definiere syntaktische Einschränkungen für änderbare Sichten**
 - Syntax und statische Semantik von DBMS automatisch überprüfbar
 - Einschränkungen u.U. nicht vollständig

Regeln für Sichten

- Die Sicht muss durch eine einzelne select-Anweisung definiert sein, d.h., kein join, union, etc.
- Die select-Klausel darf nur Attributnamen enthalten und jeden Namen nur einmal, keine Aggregatfunktionen, berechnete oder konstante Ausdrücke; ebenso kein distinct.
- Die from-Klausel darf nur einen einzigen Relationsnamen enthalten. Dieser muss eine Basisrelation oder eine änderbare Sicht bezeichnen.
- Falls die where-Klausel geschachtelte Anfragen beinhaltet, darf in deren from-Klauseln der Relationsname aus "from" nicht auftauchen, d.h. keine korrelierten Unterabfragen

Beispiel Sichten

- **Gegeben seien die beiden Tabellen:**
 - MGA(Mitarbeiter, Gehalt, Abteilung)
 - AL(Abteilung, Leiter)
- **MGA speichert Daten über Zugehörigkeit von Mitarbeitern zu Abteilungen und deren jeweiliges Gehalt**
- **AL gibt für jede Abteilung den Abteilungsleiter an**

Beispiel Sichten – Projektion

- **Wir definieren eine Sicht MA auf MGA, bei der das Gehalt ausgeblendet wird:**

```
create or replace view MA as (
    select Mitarbeiter, Abteilung
    from MGA
)
```

- **Diese Sicht gilt als änderbar, kann aber Probleme bereiten**
 - **insert into MA values ('Zuse','Informatik')** wird umgesetzt zu
 - **insert into MGA values ('Zuse',**null**, 'Informatik')** **null** evtl. in Schema ausgeschlossen
- **Beachte:**
 - implizite constraint-Verletzungen beim Einfügen Projektionssichten
 - auf default-Werte achten

Beispiel Sichten – Selektion

- **Wir definieren eine Sicht Top von MGA, welche die Top-Verdiener enthält**

```
create view Top as (
    select Mitarbeiter, Gehalt
    from MGA
    where Gehalt > 20
)
```

- **Sicht gilt als änderbar, aber**
 - **update Top set Gehalt=15 where Mitarbeiter='Zuse'** bewegt Tupel aus der Sicht heraus und ist logisch somit fragwürdig
 - **check option:** Einfügungen und Änderungen an View müssen zu dessen Definition passen, ansonsten werden sie abgelehnt

```
create view Top as (
    select Mitarbeiter, Gehalt
    from MGA
    where Gehalt > 20
) with check option
```

Beispiel Sichten – Verbund/Join

- **Wir definieren MGAL als Verbund:**

```
create or replace view MGAL as (
    select Mitarbeiter, Gehalt, MGA.Abteilung, Leiter
    from MGA natural join AL
)
```

- **View gilt als nicht änderbar!**
 - Grund: zwei Relationen in **from**-Klausel
- **Sinnvoll, da es sonst zu Inkonsistenzen kommen kann**
 - **insert into MGAL values('Müller', 3000, 001, 'Boss')** zieht nach sich
 - **insert into MGL values('Müller',3000, 001)**
 - Aber was soll in AL geschehen, wenn Tupel (001,'Boss') nicht existiert?
 - update, insert, Fehlermeldung?

Beispiele

Welche der folgenden Views sind änderbar?

CREATE VIEW Aenderbar(...) AS ...

SELECT MAX (gehalt)
FROM Personal

SELECT abtnr, budget * 1,71
FROM Projekt

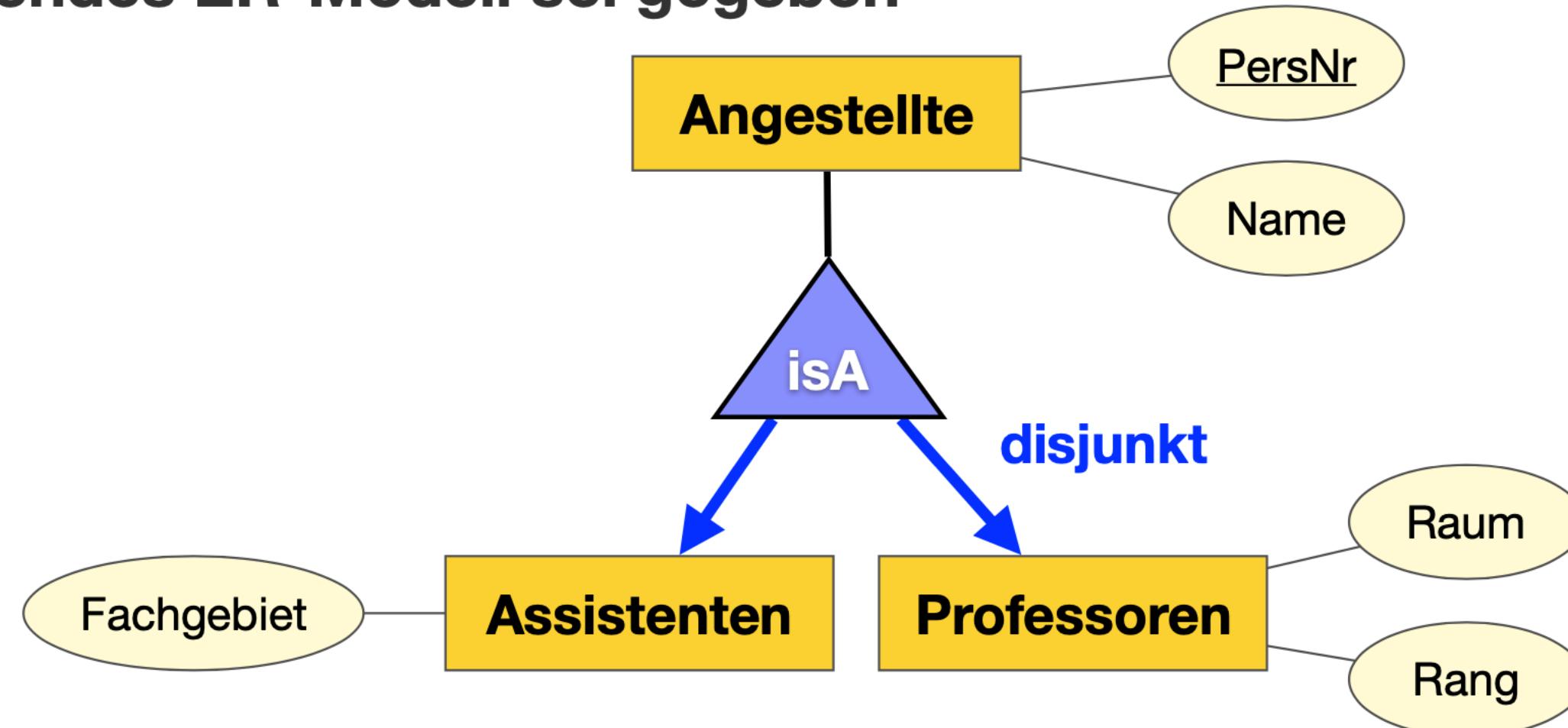
SELECT persnr, name
FROM Personal
WHERE name LIKE 'M%'

SELECT Personal.name, Projekt.name **FROM Personal **JOIN** Projekt**
USING (projnr)

SELECT gehalt
FROM Personal
WHERE gehalt > 5000

Views und Generalisierung – Variante 1

- Folgendes ER-Modell sei gegeben



- **Variante 1: Sichten für Unterklassen**
- **geerbte Attribute werden nicht repliziert:**
 - Angestellte: {[PersNr, Name]}
 - Professoren: {[PersNr, Rang, Raum]}
 - Assistenten: {[PersNr, Fachgebiet]}

Views und Generalisierung - SQL 1

- Nur Oberklasse als echte Tabelle, Unterklassen als Sichten

```
create table Angestellte (
    PersNr integer not null,
    Name varchar (30) not null );
```

// Δ Professoren - Angestellte

```
create table ProfDaten (
    PersNr integer not null,
    Rang character(2),
    Raum integer );
```

// Δ Assistenten - Angestellte

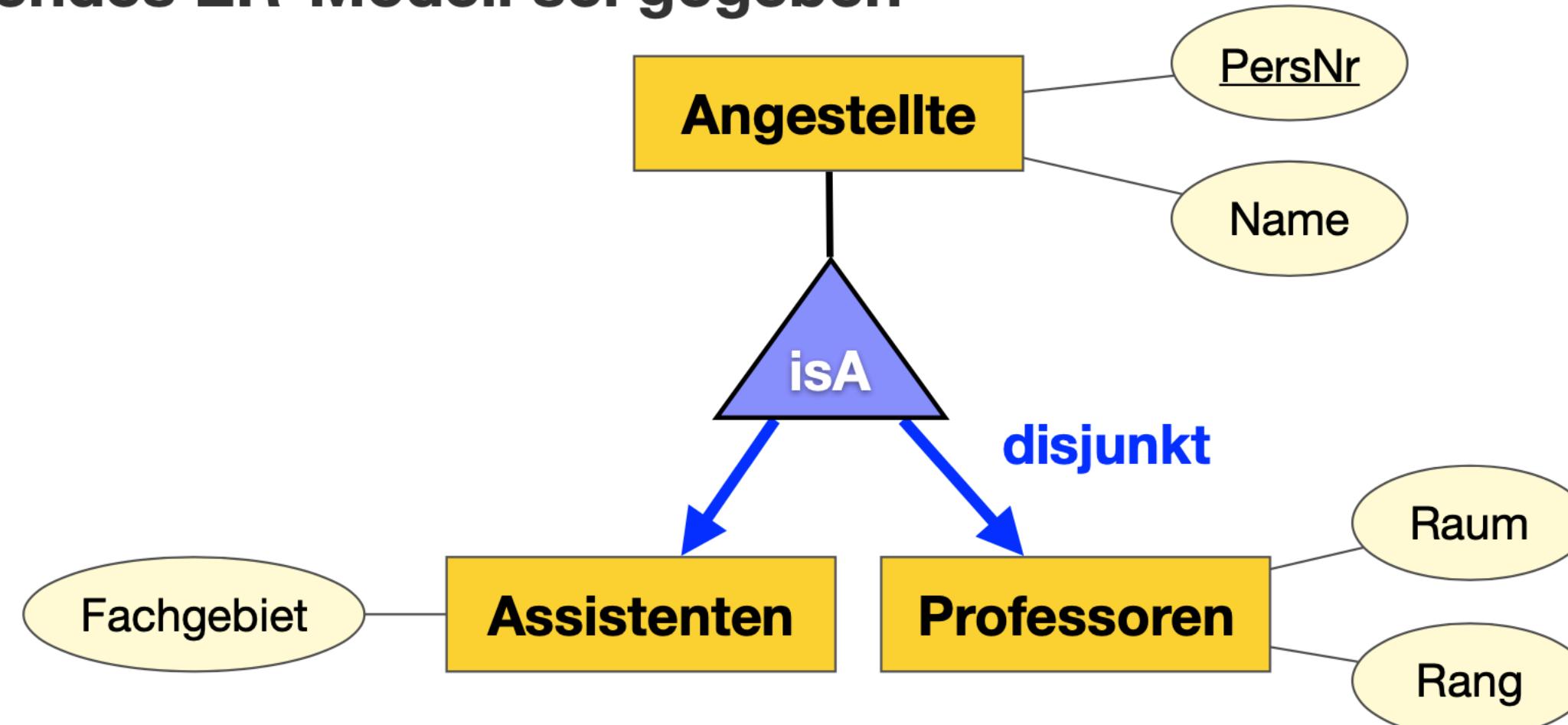
```
create table AssiDaten (
    PersNr      integer not null,
    Fachgebiet  varchar(30),
    Boss        integer );
```

```
create view Professoren as (
    select *
    from Angestellte a, ProfDaten d
    where a.PersNr=d.PersNr );
```

```
create view Assistenten as (
    select *
    from Angestellte a, AssiDaten d
    where a.PersNr=d.PersN )r;
```

Views und Generalisierung – Variante 2

- Folgendes ER-Modell sei gegeben



- **Variante 2: Sicht für Oberklasse**
- **geerbte Attribute werden repliziert:**
 - AndereAngestellte: {[PersNr, Name]}
 - Professoren: {[PersNr, Name, Rang, Raum]}
 - Assistenten: {[PersNr, Name, Fachgebiet]}

Views und Generalisierung - SQL 2

- Nur Oberklasse als echte Tabelle, Unterklassen als Sichten

```
create table Professoren (
    PersNr  integer not null,
    Name    varchar (30) not null,
    Rang    character (2),
    Raum    integer );
```

```
create table Assistenten (
    PersNr  integer not null,
    Name    varchar (30) not null,
    Fachgebiet  varchar (30),
    Boss    integer );
```

```
create table AndereAngestellte (
    PersNr  integer not null,
    Name    varchar (30) not null );
```

```
create view Angestellte as (
    ( select PersNr, Name
        from Professoren )
    union
    ( select PersNr, Name
        from Assistenten )
    union
    ( select*
        from AndereAngestellte);
)
```

Zusammenhang Objekt-Relationale-Mapper

Idee

- Abbildung von Vererbungshierarchien.
- Je nach Anwendungsfall mehrere sinnvolle Möglichkeiten der Implementierung.
- Wer entscheidet, wie modelliert wird?
- Probleme erscheinen u.a. auch beim Weiterentwickeln der Software und der Synchronisation von Klassen und Daten im DBMS.

Datenintegrität

Idee

- Daten sollen in sich schlüssig, konsistent sein.
- Auch hier die Frage: Besser in externer Business Logik oder im DBMS?
- Wir benötigen Mittel, um Funktionalität zu modellieren.
- Was passiert im Fehlerfall?

Integrität

Bisher: Column- und Table-Constraints

- primary key Festlegen der Primärschlüsselattribute
 - references Fremdschlüssel
 - unique Schlüsselkandidaten
 - check Einschränken gültiger Wert im Kontext einer Zeile
 - not NULL Wert "Null" verbieten

Neu

- Komplexe Check-Constraints Über mehrere Tabellen mit Hilfe von Triggern.
 - Referentielle Integrität Vermeiden von Fremdschlüsselverweisen "ins Leere".

Referentielle Integrität

Fremdschlüssel

- verweisen auf Tupel einer Relation
- z.B. "gelesenVon" in "Vorlesungen" verweist auf Tupel in "Professoren"

referentielle Integrität

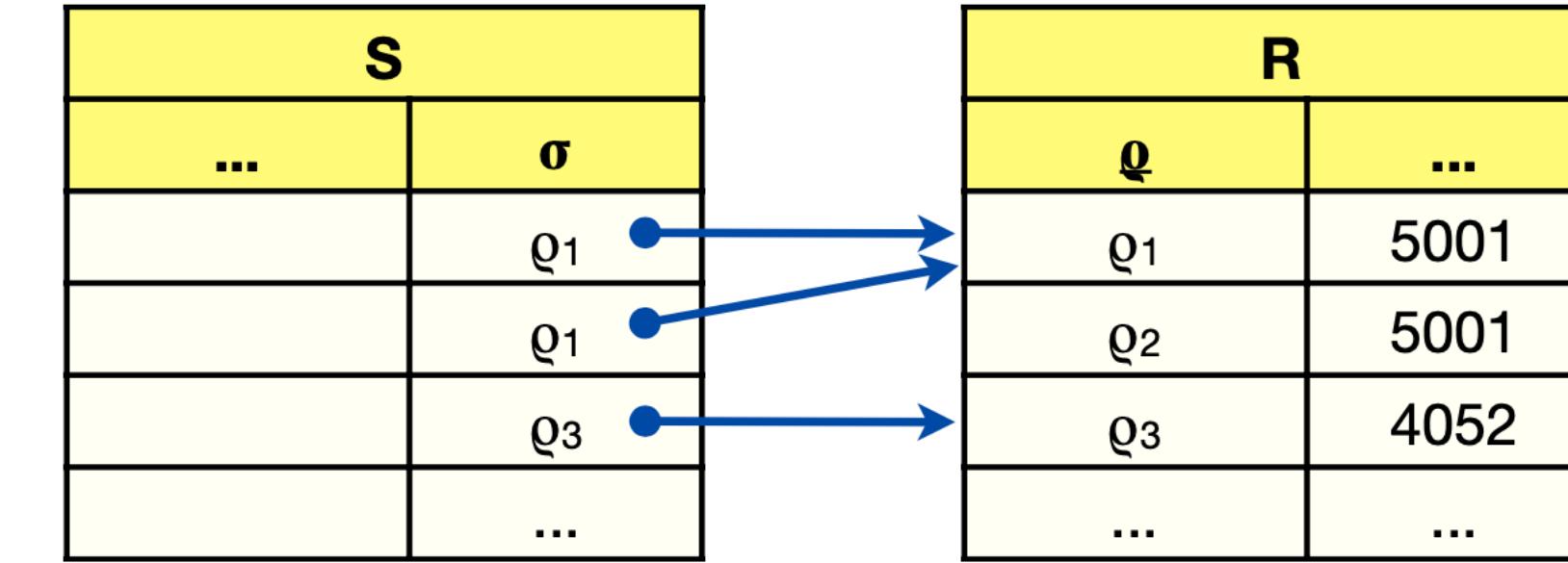
- Fremdschlüssel müssen auf existierende Tupel verweisen oder den Wert "NULL" haben

Zugehöriger Eintrag in Tabelle
"Studenten" muss existieren!

hören	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
...	...

Referentielle Integrität

```
// Tabelle R mit Schlüssel q
create table R (
    q integer primary key
    ...
)
// Tabelle S mit Schlüssel σ
create table S (
    σ integer references R
    ...
)
```



- In S dürfen für σ nur Werte eingetragen werden, für die es in R eine passende Zeile gibt (oder σ hat den Wert NULL)**
- In R dürfen keine Zeilen gelöscht werden, für die es noch Verweise aus S heraus gibt**
- Ein Schlüssel in R darf nur dann geändert werden, wenn es keinen Verweis darauf in S gibt**

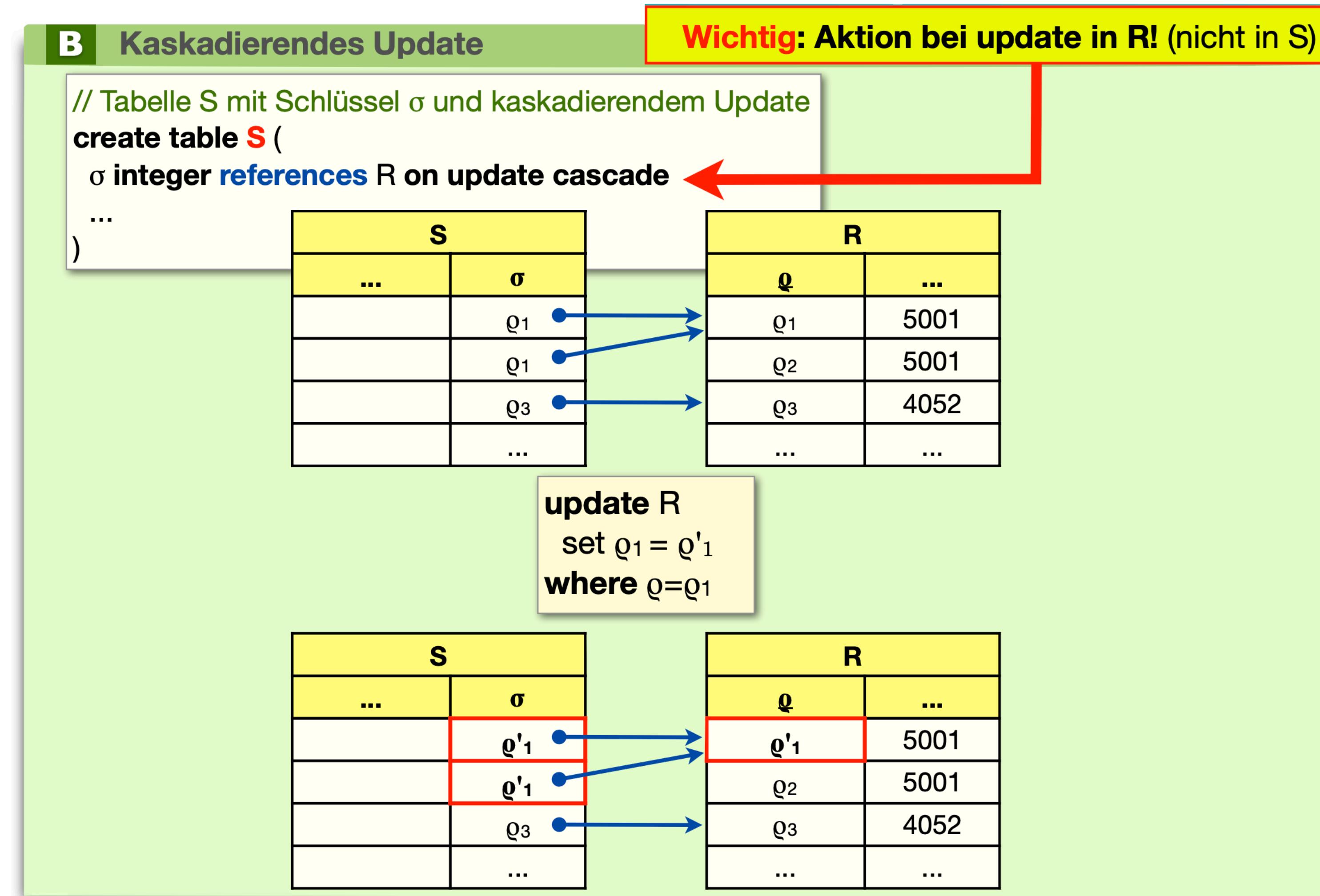
DBMS erzeugt Fehler, falls eine der Bedingungen verletzt ist!

Kaskadierende Operationen

Für Tabellen können gesonderte Regeln für festgelegt werden:

- Durch die Klausel **on update cascade** werden Veränderungen des Primärschlüssels auf den Fremdschlüssel propagiert
- Durch die Klausel **on delete cascade** zieht das Löschen eines Tupels in R das Entfernen des auf ihn verweisenden Tupels in S nach sich
- Durch die Klauseln **on update set null** und **on delete set null** erhalten beim Ändern bzw. Löschen eines Tupels in R die entsprechenden Tupel in S einen Nulleintrag

Kaskadierende Operationen



Kaskadierende Operationen

B Kaskadierendes Update

```
// Tabelle S mit Schlüssel σ und kaskadierendem Löschen  
create table S (  
    σ integer references R on delete cascade
```

```
...  
)
```

S		R	
...	σ	Q	...
	Q1	Q1	5001
	Q1	Q2	5001
	Q3	Q3	4052

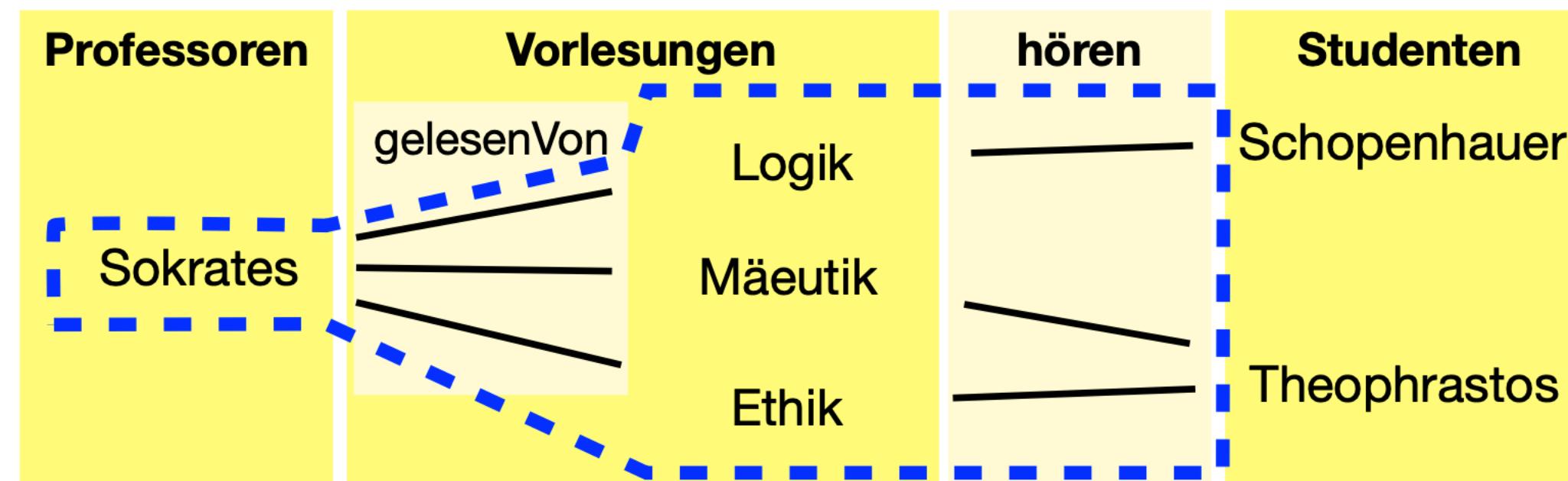

```
delete from R  
where Q=Q1
```

S		R	
...	σ	Q	...
	Q3	Q2	5001
	...	Q3	4052

Kaskadierende Operationen

Es ist durchaus erlaubt mehrere Klauseln zu verwenden:

```
// Tabelle S mit Schlüssel σ und kaskadierendem Löschen
create table S (
    σ integer references R on delete cascade
        on update cascade
    ...
)
```



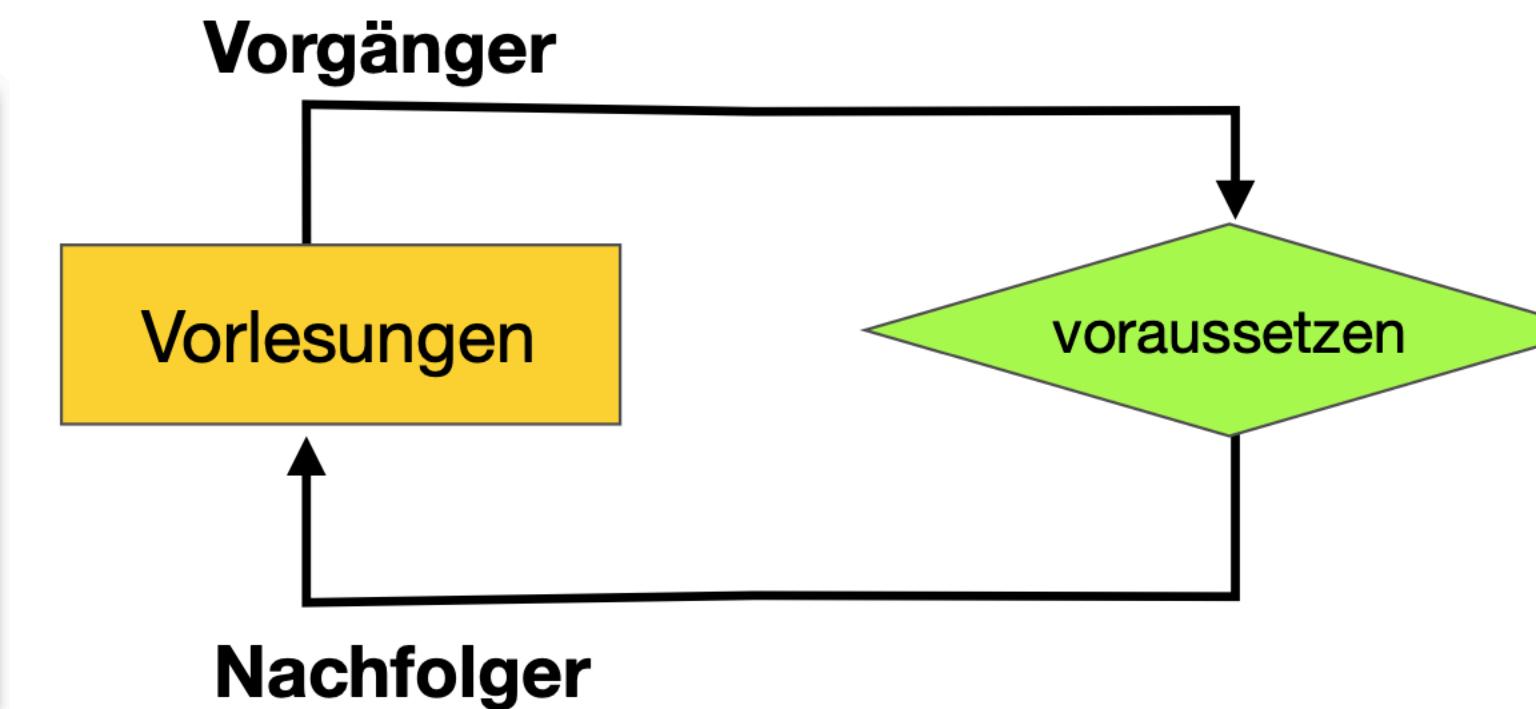
Vorsicht bei kaskadierendem Löschen!



Kaskadierende Operationen

Klausel "on ..." bezieht sich immer auf Vorgänge in Tabelle mit Primärschlüssel!

```
create table voraussetzen (
    Vorgänger references Vorlesungen
        on delete cascade,
    Nachfolger references Vorlesungen
        on delete cascade,
    primary key (Vorgänger,Nachfolger)
)
```



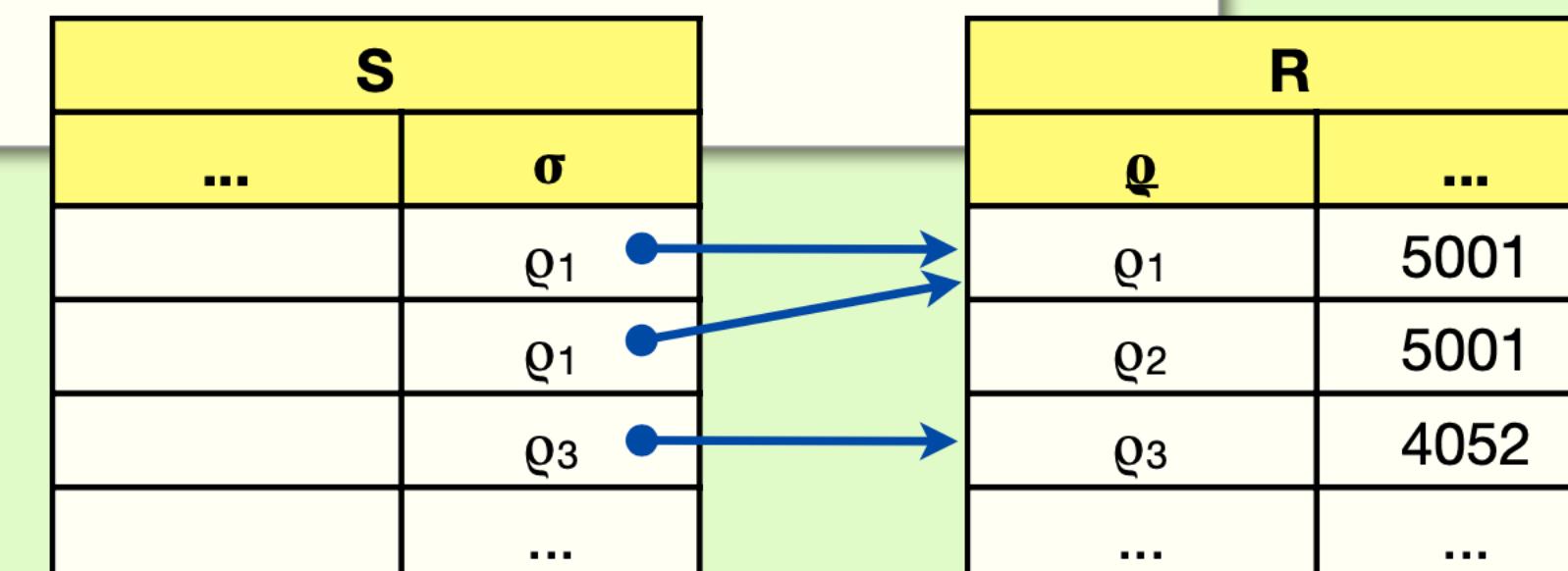
- Es wird nicht innerhalb von voraussetzen kaskadiert!

Kaskadierende Operationen

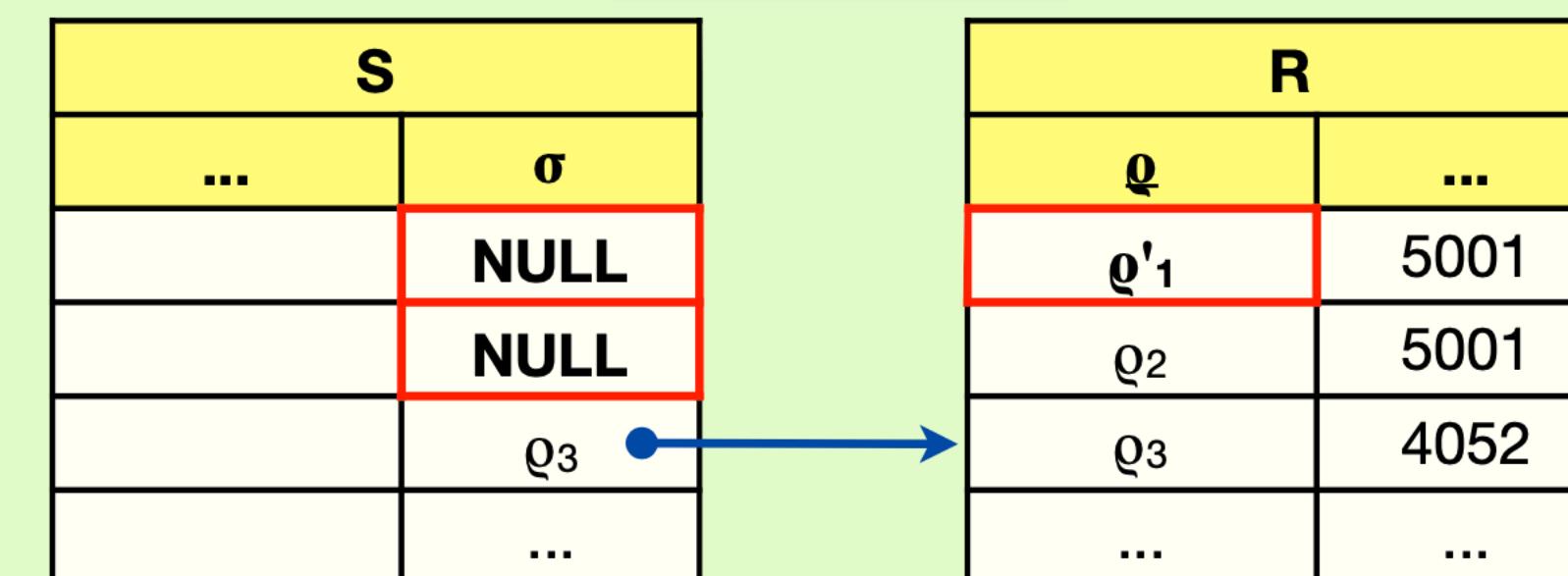
B Kaskadierendes Update

// Tabelle S mit Schlüssel σ und kaskadierendem Update

```
create table S (
     $\sigma$  integer references R on update set null
    ...
)
```



```
update R
set Q1 = Q'1
where q=Q1
```



Check

- **Falls als *table constraint* realisiert, kann Konsistenz einer Zeile geprüft werden**
 - Rückgriff nur auf Attribute dieser Zeile
- **Nicht weit verbreitet: Komplexere check-constraints**
 - Beispiel: Student darf nur Vorlesungen prüfen, die er besucht hat

```
create table prüfen (
    MatrNr integer references Studenten on update cascade
                                            on delete cascade,
    VorlNr integer references Vorlesungen on update cascade ,
    PersNr integer references Professoren,
    Note numeric (2,1) check (Note between 1.0 and 5.0),
    primary key (MatrNr, VorlNr),
    check( exists ( select * from hören as h
                    where h.VorlNr = prüfen.VorlNr and h.MatNR = prüfen.MatNr))
```

- **Problem:** *constraint* kann verletzt werden, wenn sich hören-Tabelle ändert!
- **Daher:** in Tupel und Spalten-*constraint* immer nur auf die Attribute der Zeile zugreifen!
 - nur dann: **verlässliche constraints!**

Assertions

- **Relationsübergreifende Integritätsbedingungen**
 - nicht auf Operationen bzgl. einer Tabelle beschränkt
 - **Konsequenz:** muss bei jeder Änderung geprüft werden

- **Syntax:**

```
create assertion Name check( expression )
drop assertion Name
```

- **Werden von kaum einem DBMS unterstützt!**
 - Alternative: Auf Trigger zurückgreifen

Beispiel Assertions

B Einfache Stored Procedure als SQL-Kommando

- Student darf nur Vorlesung prüfen, wenn er diese gehört hat

```
create assertion check_prüfen
check( exists ( select * from hören as h
                where h.VorlNr = prüfen.VorlNr and h.MatNR = prüfen.MatNr)
      );
```

Beachte:

- **nur boole'sche Ausdrücke möglich!**
→ einfach in der Handhabung - *constraint programming*
- **Alle Operationen, die Assertion verletzen müssen abgelehnt werden!**
→ **Problem:** Welche Operationen sind das?
→ Für DBMS nicht einfach herauszufinden, daher kaum umgesetzt
Alternative: Trigger - Programmierer legt fest, was wann geprüft werden muss

Stored Procedures

Idee

- Funktionalität, z.B. Geschäftslogik oder Massnahmen zur Erhaltung der Datenintegrität, im DBMS.

Q&A

- DBMS oder Business-Logik im Programm?
- Vor- und Nachteile DBMS?

Stored Procedures

- Grundlage für Trigger
- Mit Stored Procedures können Datenbankabläufe als definierte Prozessabläufe ohne Benutzerinteraktion hinterlegt werden
 - werden im Server in übersetzter Form gespeichert
 - werden vom Server ausgeführt (Vermeidung von Datentransporten zum Client)
 - Vorgefertigte Abläufe können das Sicherheitsniveau verbessern
(z.B. Vermeidung von SQL-Injections)
- Neben SQL kommen auch andere Sprachen zum Einsatz
 - neben speziellen "Makrosprachen" z.B. auch - JAVA (z.B. Oracle, DB2), Perl, PHP (PostgreSQL), .NET-Sprachen (Microsoft) oder JAVA
 - wenige Implementierungen orientieren sich am SQL/PSM-Standard
(PSM=Persistent Stored Modules), MySQL, DB2, PostgreSQL (mit zusätzlichem Modul PL/pgPSM)

Stored Procedures

B Einfache Stored Procedure als SQL-Kommando

- Funktion zur Berechnung des Durchschnitts zweier Zahlen

```
create or replace function average(IN numeric,IN numeric) returns numeric AS
  'select ($1 + $2) / 2.0;'    // Beachte den Makro-Character!
language SQL
immutable                      // Hinweis an Optimizer: keine Änderung an Tabellen
returns null on null input; // Umgang mit NULL-Werten

select average(3,5);
```

- Alle Vorlesungen nebst Dozent und SWS, die ein Student hört

```
create or replace function alleVorlesungen(IN integer) returns table (titel varchar,
                                                               name varchar,
                                                               sws integer) AS

'select titel,name,sws from (hören natural join Vorlesungen)
                           join Professoren on PersNr=gelesenVon
                           where MatrNr = $1;'

language SQL;

select * from alleVorlesungen(28106);
```

Stored Procedures

B Einfache Stored Procedures in anderen Sprachen

- Funktion zur Berechnung des Durchschnitts zweier Zahlen in PL/PGSQL

```
create or replace function average(IN a numeric,IN b numeric) returns numeric AS $$  
begin  
    return (a + b) / 2.0;  
end  
$$  
language plpgsql  
immutable  
returns null on null input;
```

Start- bzw. End-Tag

- Berechnung des Maximums in PL/PHP

```
create or replace function php_max(integer, integer) returns integer AS $$  
if ($args[0] > $args[1]) {  
    return $args[0];  
} else {  
    return $args[1];  
}  
$$  
strict // Kurzform für set null on null input  
language 'plphp';
```

Trigger

- Vereinbarung einer Stored Procedure und der sie auslösenden Ereignisse, etwa
 - Events Durch Änderung ausgelöstes Ereignis.
 - Conditions Filtern und Kategorisieren der Änderung.
 - Actions Durch das DBMS durchgeführte Anweisungen.
 - Ein Trigger kann vor oder nach einer Operation ausgelöst werden.
 - Vor einer Aktion: Unterdrücken möglich; z.B. wenn neue Werte bestimmten Bedingungen nicht genügen -> komplexe check-constraints.
 - Mit einem Trigger können komplexe Aktionen mit Ereignissen kombiniert werden, z.B. besondere Maßnahmen bei bestimmten Fehlersituationen.
 - Häufig werden mit einem Trigger auch nicht Datenbank-spezifische Abläufe in einem Unternehmen angesprochen (falls das DBMS entsprechendes unterstützt), wie das Versenden einer SMS oder E-Mail bei Störfällen.

Trigger

B Trigger als komplexes check-constraint (PostgreSQL)

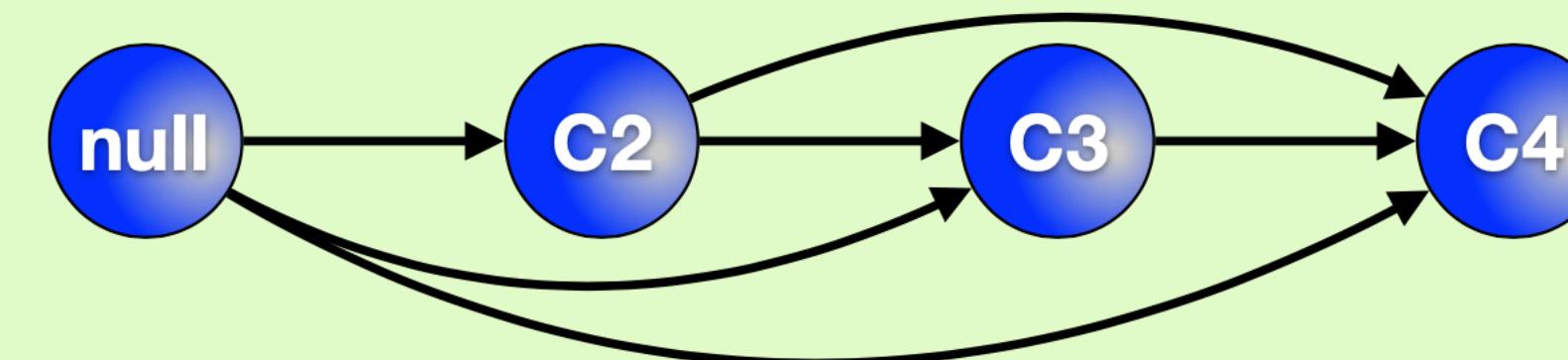
- Student darf nur Vorlesung prüfen, wenn er diese gehört hat

```
create or replace function mussVorlesungHoeren() returns trigger as $$  
declare hatGehoert boolean;  
    name varchar;  
    vorlesung varchar;  
begin  
    hatGehoert = exists (select * from hören as h  
                          where h.VorlNr = NEW.VorlNr AND h.MatrNr = NEW.MatrNr);  
    if (not hatGehoert) then  
        select s.name into name from studenten as s where MatrNr = NEW.MatrNr;  
        select v.titel into vorlesung from vorlesungen as v where VorlNr = NEW.VorlNr;  
        raise notice '% hat die Vorlesung % nicht gehört',name,vorlesung;  
        return OLD;  
    end if;  
    return NEW;  
end  
$$ LANGUAGE plpgsql;  
  
create trigger mussVorlesungHoeren  
before insert on prüfen  
for each row execute procedure mussVorlesungHoeren();
```

Trigger

B Trigger als komplexes check-constraint 2 (PostgreSQL)

- Rang eines Professors nicht beliebig änderbar
 - Degradierungen sind nicht erlaubt



```
create or replace function keineDegradierung() returns trigger AS $$  
begin  
    if (OLD.Rang is not null) then  
        if (NEW.Rang is NULL) then NEW.Rang = OLD.Rang;  
        elsif (NEW.Rang < OLD.Rang) then  
            raise exception '% darf nicht von % auf % degradiert werden',  
                NEW.name, OLD.Rang, NEW.Rang;  
        end if;  
    end if;  
    return NEW;  
end  
$$ language plpgsql;
```

```
create trigger keineDegradierung before update on Professoren  
for each row execute procedure keineDegradierung();
```

Trigger

Wann sind Trigger sinnvoll?

- Bedingung kann nicht einfach über SQL-Constraints formuliert werden, bzw. als Ersatz für Assertions.
- Die entsprechende Überprüfung müsste über verschiedene Anwendungsprogramme erfolgen.
- Es sollen nicht datenbankspezifische Aktionen getroffen werden.

Anmerkungen

- Trigger sollten nur dann verwendet werden, wenn die Reaktion nicht mittels Check und Assertion realisiert werden kann.
- Vorsicht: Trigger können wieder weitere Trigger auslösen!
- Bei reinen 3-Ebenen-Architekturen werden diese Mechanismen oft in der mittleren Ebene implementiert (Geschäftsprozesslogik).

Beispieldatenbank Kemper/Eickler

Tabellen werden u.a. bei Prüfungen im SQL-Teil verwendet.

Dabei geht es allerdings um die SQL-Kommandos, nicht um das Ergebnis. Das bedeutet, primär die Struktur der Tabellen ist relevant, nicht der Inhalt.

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

prüfen			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Studenten			
MatrNr	Name	Semester	
24002	Xenokrates	18	
25403	Jonas	12	
26120	Fichte	10	
26830	Aristoxenos	8	
27550	Schopenhauer	6	
28106	Carnap	3	
29120	Theophrastos	2	
29555	Feuerbach	2	

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

Assistenten			
PersNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126