Datenbankanwendung

Wintersemester 2014/15

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Ausblick auf kommende Vorlesungen

- Weiterführende SQL Konzepte
- Views
- JDBC: Java Database Connectivity
- PL/pgSQL
- Trigger

Zur Erinnerung: Die relationale Uni-DB

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

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	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

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

hören	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
29120	5041
29120	5049
29555	5022
25403	5022

Assistenten			
PerslNr	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

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

Anfragen über mehrere Relationen

Welche Studenten hören welche Vorlesungen?

```
select Name, Titel
from Studenten, hören, Vorlesungen
where Studenten.MatrNr=hören.MatrNr and
      hören.VorlNr = Vorlesungen.VorlNr;
```

Alternativ:

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

Sortierung

```
select    PersNr, Name, Rang
from      Professoren
order by  Rang desc, Name asc;
```

- **asc**: aufsteigend (Englisch: ascending)
- **desc**: absteigend (Englisch: descending)

Warum ist dies in relationaler Algebra nicht möglich?

Duplikateliminierung

```
select Rang
from Professoren;
```

Rang
C4
C3
C3
C4
C4
C3
C4

```
select distinct Rang
from Professoren
```

Rang
C4
C3

Aggregatfunktion und Gruppierung

Aggregatfunktionen **avg, max, min, count, sum**.

select avg (Semester)
from Studenten;

Gruppierung:

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

Bedeutung:

1. alle Tupel, die den gleichen Wert für Attribut gelesenVon haben, werden zu einer Gruppe zusammengefasst
2. für jede dieser Gruppen wird dann die Summe gebildet

Beispiel: Was ist OK und was erzeugt Fehler.

	select			
group by		rang, name	rang	name
	rang, name	OK	OK	OK
	rang	ERROR	OK	ERROR
	name	ERROR	ERROR	OK

select .... **from** Professoren ... **group by**;

- Alle im select aufgeführten Attribute, über die nicht aggregiert wird, müssen im group by aufgeführt werden.
- ABER: Nicht alle im group by aufgeführten Attribute müssen im select aufgeführt sein!

Weitere Beispiele zur Gruppierung

- **select** count(*)
  from Vorlesungen
  group by gelesenVon
- OK: Ergebnis = Anzahl der Elemente pro Gruppe

- **select** gelesenVon, count(*)
  from Vorlesungen
  group by SWS
- ERROR: gelesenVon muss ins group by!

- **select** gelesenVon, count(*)
  from Vorlesungen
  group by SWS, gelesenVon
- OK.

Aggregatfunktion und Gruppierung

```
select gelesenVon, Name, sum(SWS)
from Vorlesungen, Professoren
where gelesenVon=PersNr and Rang = 'C4'
group by gelesenVon, Name
         having avg (SWS) >= 3;
```

1. Gruppieren nach gelesenVon, Name
2. Welche Gruppen erfüllen die Bedingung in **having**
3. Im select wird dann das Aggregat pro Gruppe berechnet, hier: die Summe der SWS.

Weiterführende SQL Konzepte

Fensteranfragen

- SQL:2003 Standard
- Ermöglicht das Gruppieren/Partitionieren von Tupeln der Ergebnismenge.
- Und Anwendung einer Aggregat-Funktion auf diese Partitionen
- Z.B. **select** .... , count(*) **over** (**partition by** MatrNr) as vlcount **from** ...

Rekursion

- Beispiel: Gegeben eine Relation `setztVoraus`, in der für eine Vorlesung jeweils die direkten Vorgänger (welche vorausgesetzt werden) angegeben sind
- Wie können dann rekursiv (beliebig tief) Vorlesungen gefunden werden, auf denen eine Vorlesung aufbaut?

```
select s.name, count(*) over
(partition by s.matrnr) as vlcount,
h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

	name character varying(30)	vlcount bigint	vorlnr integer
1	Carnap	4	5259
2	Carnap	4	5216
3	Carnap	4	5052
4	Carnap	4	5041
5	Feuerbach	2	5001
6	Feuerbach	2	5022
7	Fichte	1	5001
8	Jonas	1	5022
9	Schopenhauer	2	5001
10	Schopenhauer	2	4052
11	Theophrastos	3	5049
12	Theophrastos	3	5041
13	Theophrastos	3	5001

Im Vergleich dazu:

```
select s.name, count(*) as vlcount,
h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
group by s.name, h.vorlnr
order by s.name asc;
```

	name character varying(30)	vlcount bigint	vorlnr integer
1	Carnap	1	5041
2	Carnap	1	5052
3	Carnap	1	5216
4	Carnap	1	5259
5	Feuerbach	1	5001
6	Feuerbach	1	5022
7	Fichte	1	5001
8	Jonas	1	5022
9	Schopenhauer	1	4052
10	Schopenhauer	1	5001
11	Theophrastos	1	5001
12	Theophrastos	1	5041
13	Theophrastos	1	5049

Im Vergleich dazu:

```
select s.name, count(*) as vlcount
from hoeren h, studenten s
where h.matrnr=s.matrnr
group by s.name
order by s.name asc;
```

	name character varying(30)	vlcount bigint
1	Carnap	4
2	Feuerbach	2
3	Fichte	1
4	Jonas	1
5	Schopenhauer	2
6	Theophrastos	3

Fensterfunktionen mit mehreren Partitionen

```
select s.name, count(*) over (partition by s.matrnr) as vlcount,
count(*) over (partition by h.vorlnr) as studentcount, h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

- **vlcount:** Anzahl Tupel mit demselben Namen
- **studentcount:** Anzahl Tupel mit derselben VorlNr

	name character varying(30)	vlcount bigint	studentcount bigint	vorlnr integer
1	Carnap	4	1	5052
2	Carnap	4	2	5041
3	Carnap	4	1	5216
4	Carnap	4	1	5259
5	Feuerbach	2	2	5022
6	Feuerbach	2	4	5001
7	Fichte	1	4	5001
8	Jonas	1	2	5022
9	Schopenhauer	2	4	5001
10	Schopenhauer	2	1	4052
11	Theophrastos	3	1	5049
12	Theophrastos	3	2	5041
13	Theophrastos	3	4	5001

rank()

```
select s.name, count(*) over (partition by s.matrnr) as vlcount,
rank() over (partition by s.name order by h.vorlnr) as rank, h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

- **rank:** Position in Gruppe wenn partitioniert nach s.name

	name character varying(30)	vlcount bigint	rank bigint	vorlnr integer
1	Carnap	4	1	5041
2	Carnap	4	2	5052
3	Carnap	4	3	5216
4	Carnap	4	4	5259
5	Feuerbach	2	1	5001
6	Feuerbach	2	2	5022
7	Fichte	1	1	5001
8	Jonas	1	1	5022
9	Schopenhauer	2	1	4052
10	Schopenhauer	2	2	5001
11	Theophrastos	3	1	5001
12	Theophrastos	3	2	5041
13	Theophrastos	3	3	5049

window frames

select h.matrnr, count(h.matrnr) over ()
from hören h;

- keine Partitionierung
- alle Werte gleich

	matrnr integer	count bigint
1	26120	13
2	27550	13
3	27550	13
4	28106	13
5	28106	13
6	28106	13
7	28106	13
8	29120	13
9	29120	13
10	29120	13
11	29555	13
12	25403	13
13	29555	13

select h.matrnr, count(h.matrnr) over (order by h.matrnr)
from hören h;

- keine Partitionierung
- ABER: **laufendes Fenster**
- Fenster enthält
  - alle Werte bis hierhin
  - einschließlich derjenigen mit **derselben** matrnr

	matrnr integer	count bigint
1	25403	1
2	26120	2
3	27550	4
4	27550	4
5	28106	8
6	28106	8
7	28106	8
8	28106	8
9	29120	11
10	29120	11
11	29120	11
12	29555	13
13	29555	13

Regeln für Fensterfunktion

- Nur im SELECT und ORDER BY erlaubt
- Nicht in **group by**, **having** und **where**
- Warum? Ausführung der Fensterfunktionen passiert logisch nach diesen Statements
- kann mit normaler Aggregation kombiniert werden:

```
select h.matrnr, count(*) as countstar, count(*) over (order by h.matrnr)
as countpartition
from hören h
group by h.matrnr;
```

- **countpartition:** laufendes Fenster über Ergebnis der Gruppierung!

	matrnr integer	countstar bigint	countpartition bigint
1	25403	1	1
2	26120	1	2
3	27550	2	3
4	28106	4	4
5	29120	3	5
6	29555	2	6

```
select h.matrnr, count(*) as countstar, count(*) over () as countpartition
from hören h
group by h.matrnr;
```

- keine Partitionierung!
- kein laufendes Fenster!

Auswertungsreihenfolge:

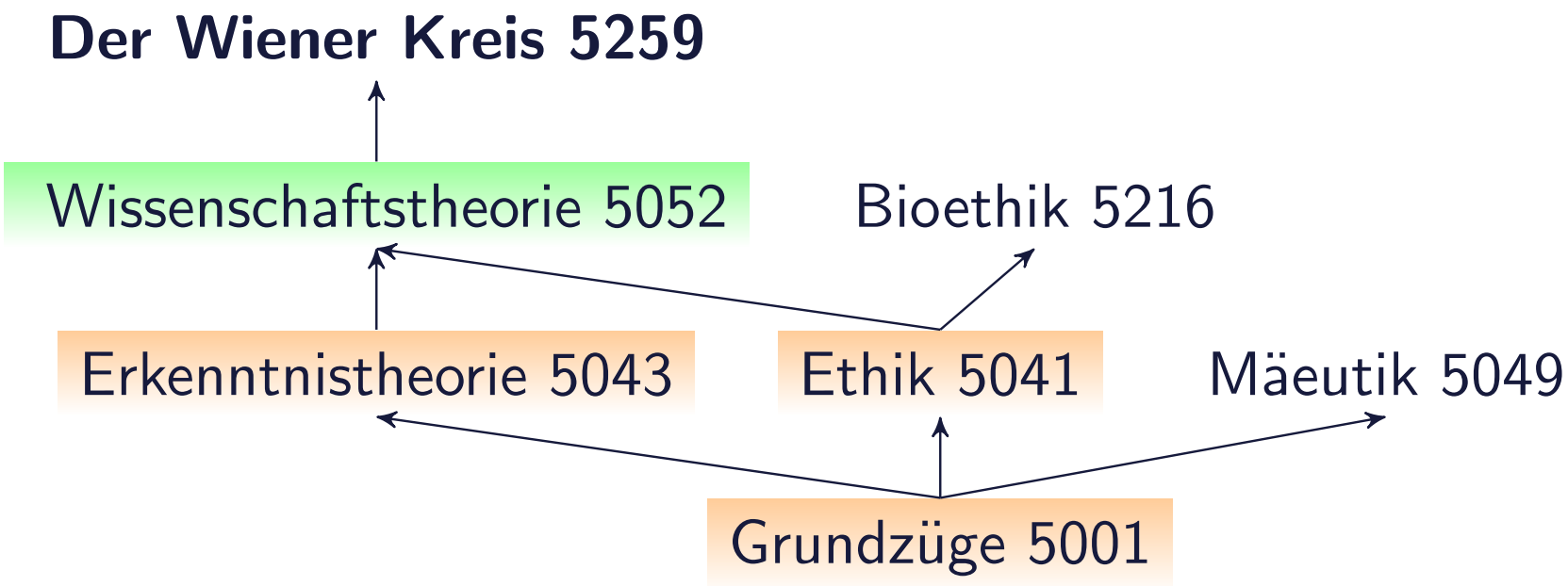
- erst **where**
- dann **group by**
- dann **having**
- dann **Fenster**

Rekursion

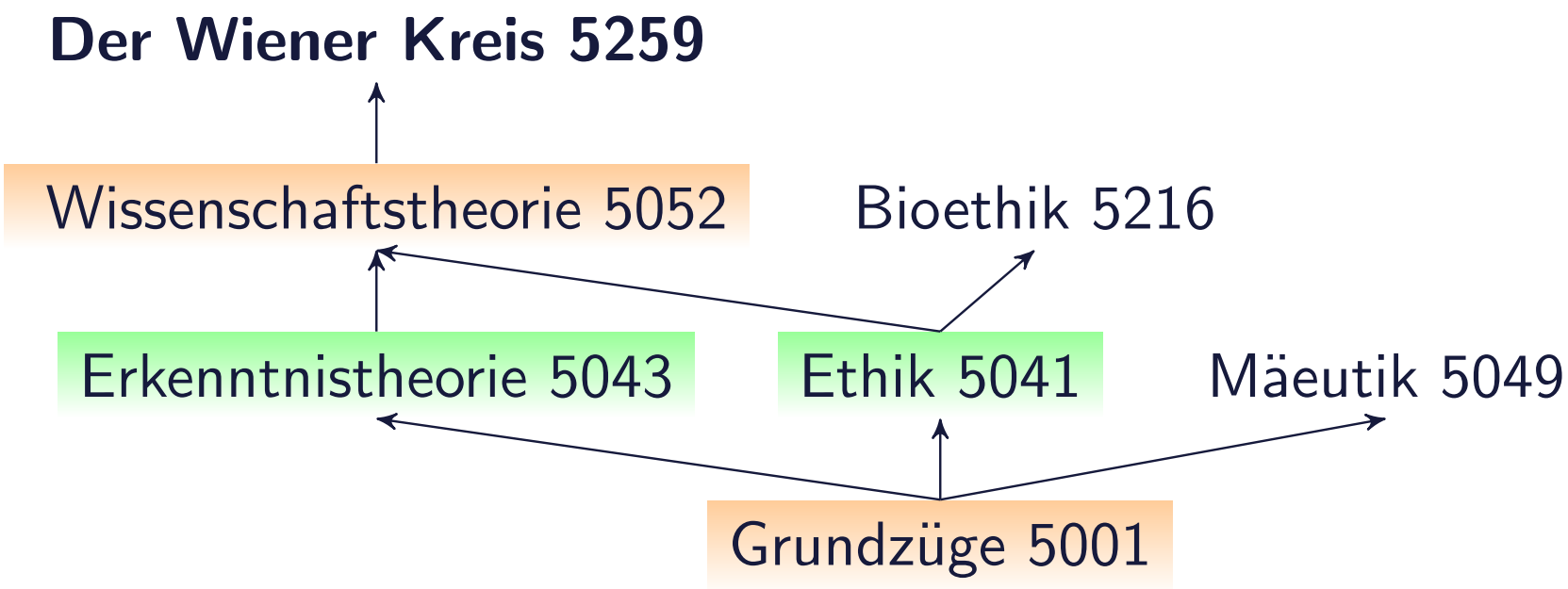
Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?

```
select Vorgaenger
from voraussetzen, Vorlesungen
where Nachfolger=VorlNr and
      Titel = 'Der Wiener Kreis';
```

Hier werden allerdings nur die direkten Vorgänger berechnet. Ergebnis: Vorlesung mit VorlNr=5052.


Der Wiener Kreis 5259
Wissenschaftstheorie 5052
Bioethik 5216
Erkenntnistheorie 5043
Ethik 5041
Mäeutik 5049
Grundzüge 5001

```
select v1.Vorgaenger
from voraussetzen v1, voraussetzen v2, Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger and
v2.Nachfolger= v.VorlNr and
v.Titel='Der Wiener Kreis';
```



Vorgänger der Tiefe n

```
select v1.Vorgaenger
from voraussetzen v1
        ...
        voraussetzen vn_minus_1
        voraussetzen vn,
        Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger and
        ...
        vn_minus_1.Nachfolger= vn.Vorgaenger and
        vn.Nachfolger = v.VorlNr and
        v.Titel='Der Wiener Kreis';
```

Alle Vorgänger?

Tiefe 1 union Tiefe 2 union Tiefe 3 union ...

Transitive Hülle

Was hier sehr hilfreich wäre: Berechnung der **transitiven Hülle**.

$$trans_{A,B}(R) = \{(a,b)|\exists k \in \mathbb{N}(\exists \tau_1, \tau_2, ...\tau_k \in R($$

$$\tau_1.A = \tau_2.B \wedge$$

$$\tau_2.A = \tau_3.B \wedge$$

$$...$$

$$\tau_{k-1}.A = \tau_k.B \wedge$$

$$\tau_1.A = a \wedge$$

$$\tau_k.B = b))\}$$

Enthält alle Paare, für die ein "Pfad" beliebiger Länge k in R existiert.

Values Statement

- erzeugt konstante Tabelle
- **values** (1, 'eins'), (2, 'zwei'), (3, 'drei');
- ist äquivalent zu:

```
    select 1 as column1, 'eins' AS column2
union all
    select 2, 'zwei'
union all
    select 3, 'drei';
```

Verwendung z.B. in Select Statement (auch in Joins) als normale Tabelle

```
select * from
(values (1,'eins'), (2,'zwei') ) as table1 (nummer, wert);
```

Vergleiche hierzu:

```
with t(n) as(
    values (1)
)
select n+1
from t;
```

Rekursionen in SQL

```
with recursive t(n) as (
        values(1)
    union all
        select n+1
        from t
        where n < 100
)
select sum(n)
from t;
```

nicht rekursiver Teil
union all

rekursiver Teil:
nur dieser darf t(n) referenzieren

eigentlicher Aufruf

Ergebnis: 5050

Achtung: Es ist Ihre Aufgabe darauf zu achten, dass die Rekursion terminiert!

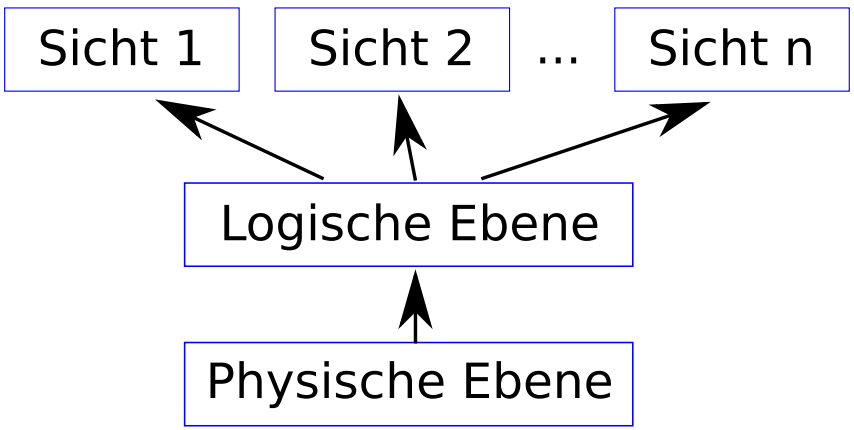
Rekursion: Anfrage für Voraussetzungen

```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select *
from TransitivVorl
order by (vorg,nachf) asc;
```

Rekursion: Anfrage für Voraussetzungen

```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select v2.titel
from TransitivVorl tv, vorlesungen v1, vorlesungen v2
where tv.nachf=v1.vorlnr and v1.titel='Der Wiener Kreis'
    and vorg=v2.vorlnr;
```

Abstraktionsebenen eines Datenbanksystems



Teilmenge des Datenbankschemas, z.B., verschiedene Rollen

Datenbankschema, z.B. Menge von Tabellen

Geräte, Dateien, Dateiformate, z.B., Dateien, Festplatten

Datenunabhängigkeit:

- physische Unabhängigkeit
- logische Datenunabhängigkeit $\rightarrow$ Sichten (Views)

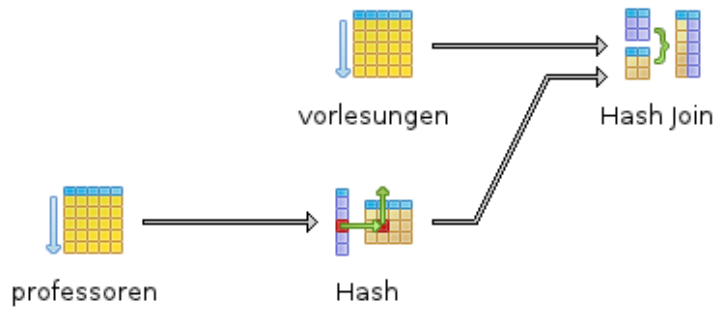
Beispiele für Sichten

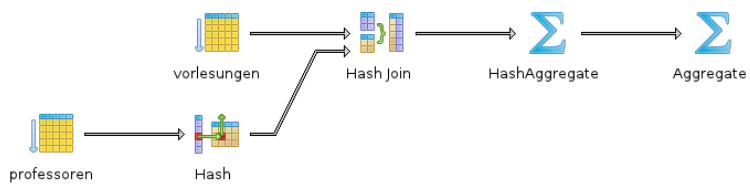
```
create view ProfsUndIhreVorlesungen as
    select v.titel, p.name
    from Professoren p, Vorlesungen v
    where p.persnr=v.gelesenvon;

select * from  ProfsUndIhreVorlesungen;

create view AnzahlSWSProProf as
    select p.name, sum(v.sws)
    from professoren p, vorlesungen v
    where p.persnr=v.gelesenvon
    group by p.name;

select sum(sum) from AnzahlSWSProProf;
```





Beispiele für Sichten

```
create or replace view
        ProfsUndIhreVorlesungen as
    select v.titel, p.name
    from Professoren p, Vorlesungen v
    where p.persnr=v.gelesenvon;
```

Achtung:

replace view erwartet dieselben Spalten in derselben Reihenfolge mit denselben Typen.

```
drop view AnzahlSWSProProf;
create view AnzahlSWSProProf as
    select p.name, p.persnr, sum(v.sws)
    from professoren p, vorlesungen v
    where p.persnr=v.gelesenvon
    group by p.name, p.persnr;
```

Alternative:

erst löschen, dann neu erzeugen. oder: Postgres SQL-Erweiterung **alter view**

Beispiele für Sichten

Relation **pruefen**: ([MatrNr, VorlNr, PersNr, Note])

```
create view pruefenSicht as
    select MatrNr, VorlNr, PersNr
    from pruefen;
```

Was ist der Unterschied zu **pruefen**?

```
create view pruefenSicht2 (M,V,P) as
    select MatrNr, VorlNr, PersNr
    from pruefen;
```

Was ist der Unterschied zu **pruefenSicht**?

```
create view pruefenSicht3 (M,V) as
    select MatrNr, VorlNr, PersNr
    from pruefen;
```

Was ist der Unterschied zu **pruefenSicht2**?

Sichten vs. Materialisierte Sichten

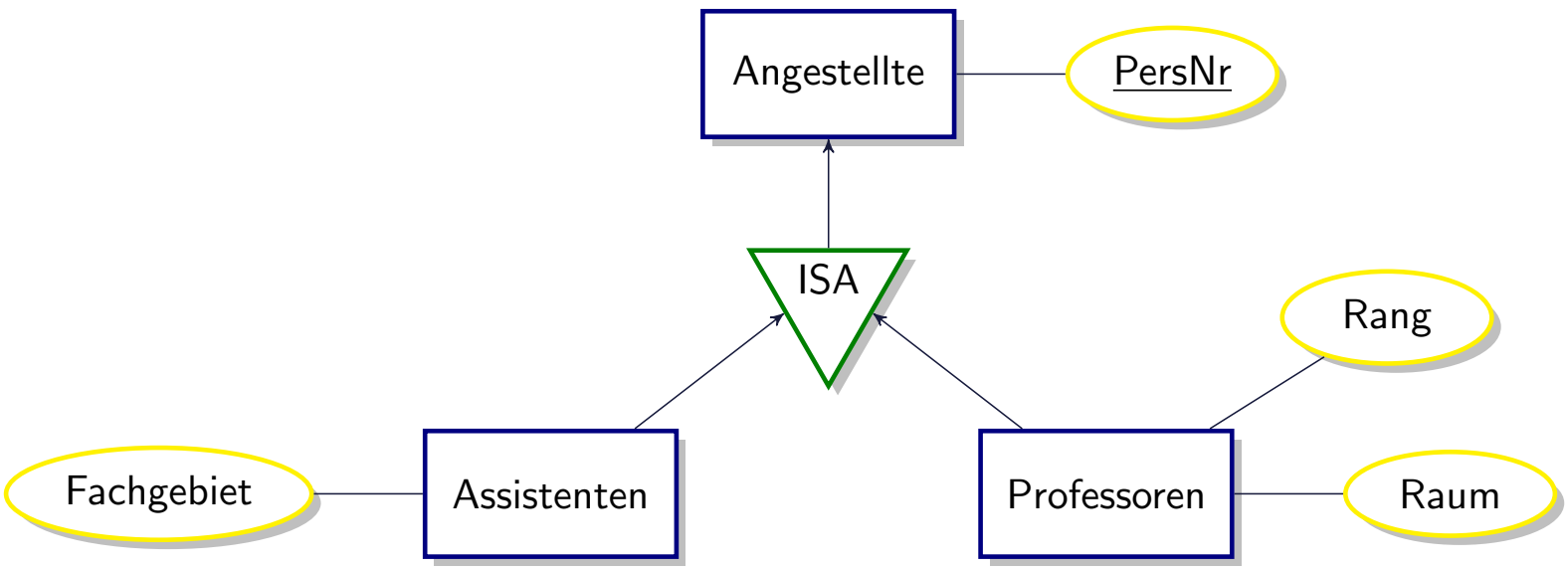
(Dynamische) Sicht

- =Makro für Query
- Ergebnis der Anfrage wird nicht vorberechnet
- Rechenaufwand zum Zeitpunkt der Anfrage (=query time)
- erst wenn die Sicht benutzt wird, wird auch das Ergebnis der Sicht berechnet

Materialisierte Sicht

- Ergebnis der Sicht wird vorberechnet
- wenn eine Anfrage die Sicht benutzt, ist das Ergebnis der Sicht bereits vollständig berechnet
- Rechenaufwand vorher (=index time)
- Problem bei Updates:
  Tabellen, die zur Vorberechnung der Sicht benutzt werden, ändern sich $\Rightarrow$ Materialisierte Sicht muß (oft) angepaßt werden

Wiederholung: Generalisierung



Sichten zur Modellierung von Generalisierung

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

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

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

```
create view Professoren as
        select *
        from Angestellte A join ProfDaten D
        on A.PersNr=D.PersNr;

create view Assistenten as
        select *
        from Angestellte A join AssistentenDaten D
        on A.PersNr=D.PersNr;
```

Untertypen als Sicht

- Also Join von Untertyp und Obertyp
- nur bei Zugriff auf Professoren oder Assistenten muss View ausgeführt werden
- D.h. Zugriff auf Obertyp ist bevorzugt (günstig).

Alternative:

```
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 Andere Angestellte
        (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);
```

Obertypen als Sicht

- Union von Ober- und Untertypen
- Also: bevorzugt Zugriff auf Untertypen
- Nur bei Zugriff auf Obertyp muss die View ausgeführt werden

Updates in SQL

```
update studenten
    set semester=semester+1
```

kombiniert mit **where**:

```
update professoren
    set rang='C4'
where name='Kopernikus'
```

Änderbarkeit von Sichten

Beispiel für nicht veränderbare Sichten

```
create view WieHartAlsPruefer as
    select PersNr, avg(Note)
    from pruefen
    group by PersNr;

update WieHartAlsPruefer
    set Durchschnittsnote=1.0
    where PersNr=(select PersNr
                  from Professoren
                  where Name='Sokrates');
```

Änderbarkeit von Sichten

Beispiel für nicht veränderbare Sichten

```
create view VorlesungenSicht as
    select Titel, SWS, Name
    from Vorlesungen, Professoren
    where gelesenVon=PersNr;

insert into VorlesungenSicht
    values ('Nihilismus', '2', 'Nobody');
```

Wie soll das DBMS dieses Tupel den Tabellen Professoren und Vorlesungen zuordnen?

Änderbarkeit von Sichten in SQL

Daten in einer View sind veränderbar (update/insert/delete), wenn ...

in SQL-92

- nur eine Tabelle
- nur Selektion und Projektion
- keine Aggregatfunktionen, Gruppierung und Duplikateliminierung

in SQL-99

- wie SQL-92
- oder: mehrere Tabellen UND Felder, auf die sich das Update bezieht, können mit Hilfe des Primärschlüssels eindeutig zugeordnet werden.
- D.h. auch bei einem Join kann man manchmal ändern.

Änderbarkeit von Sichten in Postgres

```
create view Studis as
    select *
    from studenten;

insert into studis values (42, 'mueller', 11);
```

ERROR: cannot insert into a view.
HINT: You need an unconditional ON INSERT DO INSTEAD rule.

Views in Postgres sind **nicht** änderbar!

- Aber: jede view kann einzeln "freigeschaltet" werden.
- Das passiert mit Hilfe von Regeln (rules).

Rules

create [**or replace**] **rule** name
as on event **to** table [**where** condition]
do [also | instead] action

- event: update/insert/delete
- action:
  - nothing: mache nichts
  - command: eine Aktion
  - command1; command2; ... : mache mehrere Aktionen
- also
- instead

`http://www.postgresql.org/docs/9.3/static/rules-update.html`

View update mit rule **instead**

```
create rule StudiInsert
as on insert to studis
do instead
    insert into studenten
        values (NEW.matrnr, NEW.name, NEW.semester);
```

Jetzt OK:

```
insert into studis values (42,'mueller',11);
```

View update mit rule **also**

```
create or replace rule StudiInsert
as on insert to studis
do also
    insert into studenten
        values (NEW.matrnr, NEW.name, NEW.semester);
```

```
insert into studis values (42,'mueller',11);
```

ERROR: cannot insert into a view.
HINT: You need an unconditional ON INSERT DO INSTEAD rule.

Endlosrekursion

```
create or replace rule StudentenInsert
as on insert to studenten
do  also
    insert into studenten
        values (NEW.matrnr, NEW.name, NEW.semester);
```

```
insert into studenten values (777,'mueller',11);
```

ERROR: infinite recursion detected in rules for relation "studenten"

Ohne Rekursion

```
create or replace rule StudentenInsert
as on insert to studenten
do also
    insert into hoeren
    values (New.matrnr, 5001);

select count(*) from studenten;

insert into studenten values (777, 'mueller', 11);

select count(*) from studenten;

select * from hoeren where matrnr=777;
```