

Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2008

Kapitel 3: Datenbanksysteme

Vorlesung: Prof. Dr. Christian Böhm

Übungen: Annahita Oswald, Bianca Wackersreuther

Skript © 2004 Christian Böhm, Peer Kröger

<http://www.dbs.ifi.lmu.de/Lehre/InfoNF>



Überblick

3.1 Einleitung

3.2 Das Relationale Modell

3.3 Die Relationale Algebra

3.4 Mehr zu SQL

3.5 Das E/R-Modell

3.6 Normalformen

Überblick

- Bisher (Kapitel 3.3):
Operatoren der relationalen Algebra in SQL:
 - SELECT *Attribute*
FROM *Relationen*
WHERE *Bedingung*
 - sowie die Mengenoperationen (UNION, EXCEPT, ...)
- In diesem Kapitel:
Erweiterungen, die effektives Arbeiten ermöglichen
 - Verschiedene Formen von Quantoren
 - Aggregationen
 - Sortieren und Gruppieren von Tupeln
 - Sichten

Outer Join

- **Problem:**
Beim gewöhnlichen („inner“) Join gehen diejenigen Tupel verloren, die keine Joinpartner in der jeweiligen anderen Relation haben
- **Beispiel:**
Auflistung aller Kunden mit ihren aktuellen Bestellungen:
`select * from kunde k, auftrag a where k.kname = a.kname`
Kunden ohne aktuellen Auftrag erscheinen nicht.

Kunde:

KName	KAdr	Kto
Maier	Hall	10
Huber	Ibk	25
Geizhals	Ibk	0

Auftrag:

KName	Ware	...
Maier	Brot	
Maier	Milch	
Huber	Mehl	

Kunde ⋈ Auftrag:

KName	KAdr	Kto	Ware	...
Maier	Hall	10	Brot	
Maier	Hall	10	Milch	
Huber	Ibk	25	Mehl	

Geizhals erscheint nicht mehr in der erweiterten Liste

Outer Join

- Ein Outer Join ergänzt das Joinergebnis um die Tupel, die keinen Joinpartner in der anderen Relation haben.
- Das Ergebnis wird mit NULL-Werten aufgefüllt:
select * from kunde natural outer join auftrag

Kunde:

KName	KAdr	Kto
Maier	Hall	10
Huber	Ibk	25
Geizhals	Ibk	0

Auftrag:

KName	Ware	...
Maier	Brot	
Maier	Milch	
Huber	Mehl	

Kunde n.o.j. Auftrag:

KName	KAdr	Kto	Ware	...
Maier	Hall	10	Brot	
Maier	Hall	10	Milch	
Huber	Ibk	25	Mehl	
Geizhals	Ibk	0	NULL	

Outer Join

- Aufstellung aller Möglichkeiten:
 - [inner] join: keine verlustfreie Relation, Normalfall
 - left outer join: die linke Relation ist verlustfrei
 - right outer join: die rechte Relation ist verlustfrei
 - [full] outer join: beide Relationen verlustfrei
- Kombinierbar mit Schlüsselworten **natural**, **on**, ..

L

A	B
1	2
2	3

R

B	C
3	4
4	5

inner

A	B	C
2	3	4

left

A	B	C
1	2	⊥
2	3	4

right

A	B	C
2	3	4
⊥	4	5

full

A	B	C
1	2	⊥
2	3	4
⊥	4	5

Subqueries

- Beispiel für eine Subquery

```
select * from Kunde where exists (select...from...where...)
```

Subquery

- In Where-Klausel der Subquery auch Zugriff auf Relationen/Attribute der Hauptquery
- Eindeutigkeit ggf. durch Aliasnamen für Relationen (wie bei Self-Join):

```
select *
from kunde k1
where exists ( select *
               from Kunde k2
               where k1.Adr=k2.Adr and...
             )
```

Logische Quantoren

- Es gibt in der mathematischen Logik folgende Quantoren:
 - **Existenz-Quantor:**
 - Idee: **mindestens ein** Tupel (einer speziellen Relation, siehe FROM-Klausel) muss von der Subquery zurückgeliefert werden
 - Mathematische Logik: $\exists x: \psi(x)$
 - Bedeutung: wahr wenn es mind. ein x gibt, sodass $\psi(x)$ wahr ist
 - **Allquantor:**
 - Idee: **alle** Tupel (einer speziellen Relation, siehe FROM-Klausel) von der Subquery zurückgeliefert werden sollen/müssen?
 - Mathematische Logik: $\forall x: \psi(x) \Leftrightarrow \neg \exists x: \neg \psi(x)$
 - Bedeutung: für alle möglichen x ist $\psi(x)$ wahr

Dabei: x = Tupel, $\psi(x)$ Bedingung, die x erfüllen sollen

Existenz-Quantor

- Das Schlüsselwort **exists** vor einer Subquery realisiert den Existenz-Quantor
- Der Term **exists** <SUBQUERY> ist wahr genau dann wenn das Ergebnis der Subquery nicht leer ist
- Beispiel:
KAdr der Kunden, zu denen ein Auftrag existiert:

```
select KAdr from Kunde k
where exists
  ( select * from Auftrag a
    where a.KName = k.KName
  )
```

Äquivalent
mit Join ??

Allquantor

- Keine direkte Unterstützung in SQL
- Aber leicht ausdrückbar durch die Äquivalenz:

$$\forall x: \psi(x) \Leftrightarrow \neg \exists x: \neg \psi(x)$$

- Also Notation in SQL:
...where **not exists** (select...from...**where not**...)

- Beispiel:

Die Länder, die von der SPD allein regiert werden

```
select * from Länder L1
```

```
where not exists
```

```
( select * from Länder L2
```

```
  where L1.LName=L2.LName and not L1.Partei='SPD'
```

```
)
```

Direkte Subquery

- An jeder Stelle in der **select**- und **where**-Klausel, an der ein konstanter Wert stehen kann, kann auch eine Subquery (**select...from...where...**) stehen.
- Einschränkungen:
 - Subquery darf nur ein Attribut ermitteln (Projektion)
 - Subquery darf nur ein Tupel ermitteln (Selektion)
- Beispiel: Dollarkurs aus Kurstabelle


```
select Preis,
       Preis * ( select Kurs from Devisen
                where DName = 'US$' ) as USPreis
from Waren where ...
```
- Oft schwierig, Eindeutigkeit zu gewährleisten...

Weitere Quantoren

- Quantoren bei Standard-Vergleichen in WHERE

- Formen:

- $A_i \Theta$ **all** (select...from...where...)
 - $A_i \Theta$ **some** (select...from...where...)
 - $A_i \Theta$ **any** (select...from...where...)
- $\left. \begin{array}{l} \forall\text{-Quantor} \\ \exists\text{-Quantor} \end{array} \right\}$

Vergleichsoperatoren $\Theta \in \{ = , < , \leq , > , \geq , \langle \rangle \}$

- Bedeutung:

- $A_i \Theta$ **all** (Subquery) $\equiv \{ \dots \mid \forall t \in \text{Subquery}: A_i \Theta t \}$
- ist größer als **alle** Werte, die sich aus Subquery ergeben

- Einschränkung bezüglich Subquery:

- Darf nur ein Ergebnis-Attribut ermitteln
 - Aber mehrere Tupel sind erlaubt
- $\left. \begin{array}{l} \text{Menge} \\ \text{nicht Relation} \end{array} \right\}$

Beispiel

- Ermittle den Kunden mit dem höchsten Kontostand

```
select KName, KAdr
from Kunde
where Kto >= all ( select Kto
                  from Kunde
                  )
```

- Äquivalent zu folgendem Ausdruck mit EXISTS:

```
select KName, KAdr
from Kunde k1
where not exists ( select *
                  from Kunde k2
                  where not k1.Kto >= k2.Kto
                  )
```

Subquery mit IN

- Nach dem Ausdruck A_i **[not] in ...** kann stehen:
 - Explizite Aufzählung von Werten: A_i **in** (2,3,5,7,11,13)
 - Eine Subquery:

A_i **in** (**select** wert **from** Primzahlen **where** wert \leq 13)

Auswertung:

- Erst Subquery auswerten
- In explizite Form (2,3,5,7,11,13) umschreiben
- Dann einsetzen
- Zuletzt Hauptquery auswerten

Beispiele

- Gegeben:
 - MagicNumbers (Name: String, Wert: Int)
 - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die prim sind
select * from MagicNumbers where Wert in
(select Zahl from Primzahlen)
- ist äquivalent zu folgender Anfrage mit EXISTS:
select * from MagicNumbers where exists
(select * from Primzahlen where Wert = Zahl)
- und zu folgender Anfrage mit SOME/ANY/ALL:
select * from MagicNumbers where Wert = some
(select Zahl from Primzahlen)

Beispiele

- Gegeben:
 - MagicNumbers (Name: String, Wert: Int)
 - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die *nicht* prim sind
select * from MagicNumbers where Wert not in
(select Zahl from Primzahlen)
- ist äquivalent zu folgender Anfrage mit EXISTS:
select * from MagicNumbers where not exists
(select * from Primzahlen where Wert = Zahl)
- und zu folgender Anfrage mit SOME/ANY/ALL:
select * from MagicNumbers where Wert <> all
(select Zahl from Primzahlen)

Sortieren

- In SQL mit ORDER BY A_1, A_2, \dots
- Bei mehreren Attributen: Lexikographisch

A	B	order by A, B	A	B	order by B, A	A	B
1	1		1	1		1	1
3	1		2	2		3	1
2	2		3	1		4	1
4	1		3	3		2	2
3	3		4	1		3	3

- Steht am Schluss der Anfrage
- Nach Attribut kann man ASC für aufsteigend (Default) oder DESC für absteigend angeben
- Nur Attribute der SELECT-Klausel verwendbar

Beispiel

- Gegeben:
 - MagicNumbers (Name: String, Wert: Int)
 - Primzahlen (Zahl: Int)
- Anfrage: Alle MagicNumbers, die prim sind, sortiert nach dem Wert beginnend mit größtem

```
select * from MagicNumbers where Wert in  
(select Zahl from Primzahlen)  
order by Wert desc
```

- Nicht möglich:

```
select Name from MagicNumbers order by Wert
```

Aggregation

- Berechnet Eigenschaften ganzer Tupel-Mengen
- Arbeitet also Tupel-übergreifend
- Aggregatfunktionen in SQL:
 - **count** Anzahl der Tupel bzw. Werte
 - **sum** Summe der Werte einer Spalte
 - **avg** Durchschnitt der Werte einer Spalte
 - **max** größter vorkommender Wert der Spalte
 - **min** kleinster vorkommender Wert
- Aggregate können sich erstrecken:
 - auf das gesamte Anfrageergebnis
 - auf einzelne Teilgruppen von Tupeln (siehe später)

Aggregation

- Aggregatfunktionen stehen in der Select-Klausel
- Beispiel:
Gesamtzahl und Durchschnitt der Einwohnerzahl aller Länder, die mit 'B' beginnen:

```
select sum (Einw), avg (Einw)
from länder
where LName like 'B%'
```
- Ergebnis ist immer ein einzelnes Tupel:
Keine Mischung aggregierte/nicht aggregierte Attribute
- Aggregate wie **min** oder **max** sind ein einfaches Mittel, um Eindeutigkeit bei Subqueries herzustellen (vgl. S. 10)

Aggregation

- NULL-Werte werden ignoriert (auch bei **count**)
- Eine Duplikatelimination kann erzwungen werden
 - **count** (**distinct** KName) zählt *verschiedene* Kunden
 - **count** (**all** KName) zählt alle Einträge (außer **NULL**)
 - **count** (KName) ist identisch mit **count** (**all** KName)
 - **count** (*) zählt die Tupel des Anfrageergebnisses (macht nur bei NULL-Werten einen Unterschied)

- Beispiel:

Produkt (PName, Preis, ...)

Alle Produkte, mit unterdurchschnittlichem Preis:

```
select *
```

```
from Produkt
```

```
where Preis < (select avg (Preis) from Produkt)
```

Gruppierung

- Aufteilung der Ergebnis-Tupel in Gruppen
- Ziel: Aggregationen
- Beispiel:
Gesamtgehalt und Anzahl Mitarbeiter pro Abteilung

Mitarbeiter

Aggregationen:

<u>PNr</u>	Name	Vorname	Abteilung	Gehalt	Σ Gehalt	COUNT
001	Huber	Erwin	01	2000	6300	3
002	Mayer	Hugo	01	2500		
003	Müller	Anton	01	1800		
004	Schulz	Egon	02	2500	4200	2
005	Bauer	Gustav	02	1700		

- **Beachte: So in SQL nicht möglich!**
Anfrage-Ergebnis soll wieder eine *Relation* sein

Gruppierung

Mitarbeiter

<u>PNr</u>	Name	Vorname	Abteilung	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

- In SQL:
select Abteilung, **sum** (Gehalt), **count** (*)
from Mitarbeiter
group by Abteilung

Abteilung	sum (Gehalt)	count (*)
01	6300	3
02	4200	2

Gruppierung

- Syntax in SQL:

```

select      ...
from       ...
[where     ...]
[group by  A1, A2, ...]
[having    ...]
[order by  ...]
  
```

siehe unten ←

siehe Seite 27 ff. ←

- Wegen Relationen-Eigenschaft des Ergebnisses Einschränkung der **select**-Klausel. Erlaubt sind:
 - Attribute aus der Gruppierungsklausel (incl. arithmetischer Ausdrücke etc.)
 - Aggregationsfunktionen auch über andere Attribute, count (*)
 - in der Regel kein **select * from...**

Gruppierung

- Beispiel: Nicht möglich!!!

Mitarbeiter

<u>PNr</u>	Name	Vorname	Abteilung	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

- ~~select PNr~~, Abteilung, **sum** (Gehalt)
from Mitarbeiter
group by Abteilung

„PNr“	Abteilung	Gehalt
„001,002,003“	01	6300
„004,005“	02	4200

Gruppierung mehrerer Attribute

- Etwa sinnvoll in folgender Situation:

<u>PNr</u>	Name	Vorname	Abteilung	Gruppe	Gehalt
001	Huber	Erwin	01	01	2000
002	Mayer	Hugo	01	02	2500
003	Müller	Anton	01	02	1800
004	Schulz	Egon	02	01	2500
005	Bauer	Gustav	02	01	1700

Debitoren
Kreditoren
Fernsehgeräte
Buchhaltung
Produktion

Gesamtgehalt in jeder Gruppe:

```
select  Abteilung, Gruppe,
        sum(Gehalt)
from    Mitarbeiter
group by Abteilung, Gruppe
```

Abt.	Gr.	Σ Geh.
01	01	2000
01	02	4300
02	01	4200

Gruppierung mehrerer Attribute

- Oft künstlich wegen **select**-Einschränkung:

Mitarbeiter \bowtie Abteilungen

PNr	Name	Vorname	ANr	AName	Gehalt
001	Huber	Erwin	01	Buchhaltung	2000
002	Mayer	Hugo	01	Buchhaltung	2500
003	Müller	Anton	01	Buchhaltung	1800
004	Schulz	Egon	02	Produktion	2500
005	Bauer	Gustav	02	Produktion	1700

- Nicht möglich, obwohl AName von ANr funktional abh.:
select ANr, AName, sum(Gehalt) from ... where ... group by ANr
- Aber ~~wegen der funktionalen Abhängigkeit identisch mit:~~
select ANr, AName, sum(...) from ... where ... group by ANr, AName
- Weitere Möglichkeit (ebenfalls wegen Abhängigkeit):
select ANr, max (AName), sum(...) from ... where ... group by ANr

Die Having-Klausel

- Motivation:
Ermittle das Gesamt-Einkommen in jeder Abteilung, die mindestens 5 Mitarbeiter hat
- In SQL nicht möglich:

```

select      ANr, sum (Gehalt)
from        Mitarbeiter
where     count (*) >= 5
group by   ANr
having     count (*) >= 5

```

GEHT NICHT !
STATT DESSEN:
- Grund: Gruppierung wird erst nach den algebraischen Operationen ausgeführt

Auswertung der Gruppierung

An folgendem Beispiel:

```
select A, sum(D)
from ... where ...
group by A, B
having sum (D) < 10 and max (C) = 4
```

1. Schritt:
from/where

A	B	C	D
1	2	3	4
1	2	4	5
2	3	3	4
3	3	4	5
3	3	6	7

2. Schritt:
Gruppenbildung

A	B	C	D
1	2	3	4
		4	5
2	3	3	4
3	3	4	5
		6	7

3. Schritt:
Aggregation

A	B	sum(D)	max(C)
1	2	9	4
2	3	4	3
3	3	12	6

temporäre „nested relation“

Auswertung der Gruppierung

An folgendem Beispiel:

```
select A, sum(D)
from ... where ...
group by A, B
having sum (D) < 10 and max (C) = 4
```

3. Schritt:

Aggregation

A	B	sum(D)	max(C)
1	2	9	4
2	3	4	3
3	3	12	6

4. Schritt:

having (=Selektion)

A	B	sum(D)	max(C)
1	2	9	4

5. Schritt:

Projektion

A	sum(D)
1	9

Generierung eindeutiger Schlüssel

- Keine Standard-Konstrukte in SQL
- ORACLE: Sequenz als eigenes DB-Objekt
`create sequence SName ;`
`insert into Mitarbeiter values (SName.nextval, 'Müller', ...) ;`
- MySQL: Auto-Increment-Attribute einer Tabelle
`create table Mitarbeiter`
`(PNr integer not null auto_increment,`
`...`
`);`
`insert into Mitarbeiter values (NULL, 'Müller ', ...) ;`
Bei **NULL** wird automatisch Maximalwert+1 gesetzt

Externe Ebene in SQL

- Zur Erinnerung:
 - Drei-Ebenen-Architektur zur Realisierung von
 - **physischer**
 - **und logischer**
 - Datenunabhängigkeit (nach ANSI/SPARC)
- Externe Ebene:
 - Gesamt-Datenbestand ist angepasst, so dass jede Anwendungsgruppe nur die Daten sieht, die sie...
 - sehen will (Übersichtlichkeit)
 - sehen soll (Datenschutz)
 - Logische Datenunabhängigkeit
 - In SQL:
 - Realisiert mit dem Konzept der **Sicht (View)**

Was ist eine Sicht (View)?

- Virtuelle Relation
- Was bedeutet virtuell?
 - Die View sieht für den Benutzer aus wie eine Relation:
 - **select ... from** $View_1, Relation_2, \dots$ **where ...**
 - mit Einschränkung auch: **insert, delete** und **update**
 - Aber die Relation ist nicht real existent/gespeichert;
Inhalt ergibt sich durch **Berechnung** aus anderen Relationen
- Besteht aus zwei Teilen:
 - Relationenschema für die View (nur rudimentär)
 - Berechnungsvorschrift, die den Inhalt festlegt:
SQL-Anfrage mit **select ... from ... where**

Viewdefinition in SQL

- Das folgende DDL-Kommando erzeugt eine View relativ unüblich
create [or replace] view VName [(A₁, A₂, ...)] as select ...
- Beispiel: Eine virtuelle Relation Buchhalter, nur mit den Mitarbeitern der Buchhaltungsabteilung:

```
create view Buchhalter as
select PNr,Name,Gehalt from Mitarbeiter where ANr=01
```

- Die View *Buchhalter* wird erzeugt:

Mitarbeiter

PNr	Name	Vorname	ANr	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

Buchhalter

PNr	Name	Gehalt
001	Huber	2000
002	Mayer	2500
003	Müller	1800

Konsequenzen

- Automatisch sind in dieser View alle Tupel der ***Basisrelation***, die die Selektionsbedingung erfüllen
- An diese können beliebige Anfragen gestellt werden, auch in Kombination mit anderen Tabellen (Join) etc:
select * from Buchhalter where Name like 'B%'
- In Wirklichkeit wird lediglich die View-Definition in die Anfrage eingesetzt und dann ausgewertet:

Buchhalter:

select PNr,Name,Gehalt from Mitarbeiter where ANr=01

select * from Buchhalter where Name like 'B%'

ergibt: select * from (**select PNr, Name, Gehalt
from Mitarbeiter where ANr=01**)
where Name like 'B%'

Konsequenzen

- Bei Updates in der Basisrelation (Mitarbeiter) **ändert sich auch die virtuelle Relation** (Buchhalter)
- Umgekehrt können (mit Einschränkungen) auch Änderungen an der View durchgeführt werden, die sich dann auf die Basisrelation auswirken
- Eine View kann selbst wieder Basisrelation einer neuen View sein (View-Hierarchie)
- Views sind ein wichtiges Strukturierungsmittel für Anfragen und die gesamte Datenbank

Löschen einer View:

drop view *VName*

In Views erlaubte Konstrukte

- Folgende Konstrukte sind in Views erlaubt:
 - Selektion und Projektion
(incl. Umbenennung von Attributen, Arithmetik)
 - Kreuzprodukt und Join
 - Vereinigung, Differenz, Schnitt
 - Gruppierung und Aggregation
 - Die verschiedenen Arten von Subqueries
- Nicht erlaubt:
 - Sortieren

Insert/Delete/Update auf Views

- Logische Datenunabhängigkeit:
 - Die einzelnen Benutzer-/Anwendungsgruppen sollen ausschließlich über das externe Schema (d.h. Views) auf die Datenbank zugreifen (Übersicht, Datenschutz)
 - Insert, Delete und Update auf Views erforderlich
- Effekt-Konformität
 - View soll sich verhalten wie gewöhnliche Relation
 - z.B. nach dem Einfügen eines Tupels muß das Tupel in der View auch wieder zu finden sein, usw.
- Mächtigkeit des View-Mechanismus
 - Join, Aggregation, Gruppierung usw.
 - Bei komplexen Views Effekt-Konformität unmöglich

Insert/Delete/Update auf Views

- Wir untersuchen die wichtigsten Operationen in der View-Definition auf diese Effekt-Konformität
 - Projektion
 - Selektion
 - Join
 - Aggregation und Gruppierung
- Wir sprechen von Projektions-Sichten usw.
 - Änderung auf Projektionssicht muß in Änderung der Basisrelation(en) transformiert werden
- Laufendes Beispiel:
 - MGA (Mitarbeiter, Gehalt, Abteilung)
 - AL (Abteilung, Leiter)

Projektionssichten

- Beispiel:

```
create view MA as  
select Mitarbeiter, Abteilung  
from MGA
```

- Keine Probleme beim Löschen und Update:

```
delete from MA where Mitarbeiter = ...
```

```
→ delete from MGA where Mitarbeiter = ...
```

- Bei Insert müssen wegprojizierte Attribute durch NULL-Werte oder bei der Tabellendefinition festgelegte Default-Werte belegt werden:

```
insert into MA values ('Weber', 001)
```

```
→ insert into MGA values ('Weber', NULL, 001)
```


Projektionssichten

- Problem bei Duplikatelimination (**select distinct**):
Keine eindeutige Zuordnung zwischen Tupeln der View und der Basisrelation:
 - Bei Arithmetik in der Select-Klausel: Rückrechnung wäre erforderlich:
create view P as select 3*x*x*x+2*x*x+x+1 as y from A
 - Der folgende Update wäre z.B. problematisch:
insert into P set y = 0 where ...
 - womit müsste x besetzt werden?
Mit der Nullstelle des Polynoms $f(x) = 3x^3 + 2x^2 + x + 1$
Nullstellensuche kein triviales mathematisches Problem
- Kein **insert/delete/update** bei **distinct**/Arithmetik

Selektionssichten

- Beispiel:

```
create view MG as  
select * from MGA  
where Gehalt >= 20
```

- Beim Ändern (und Einfügen) kann es passieren, dass ein Tupel aus der View verschwindet, weil es die Selektionsbedingung nicht mehr erfüllt:
`update MG set Gehalt = 19 where Mitarbeiter = 'Huber'`
- Huber ist danach nicht mehr in MG
- Dies bezeichnet man als Tupel-Migration:
Tupel verschwindet, taucht aber vielleicht dafür in anderer View auf

Selektionssichten

- Dies ist manchmal erwünscht
 - Mitarbeiter wechselt den zuständigen Sachbearbeiter, jeder Sachbearbeiter arbeitet mit „seiner“ View
- Manchmal unerwünscht
 - Datenschutz
- Deshalb in SQL folgende Möglichkeit:

```
create view MG as
select * from MGA
where Gehalt >= 20
with check option
```
- Die Tupel-Migration wird dann unterbunden
Fehlermeldung bei: `update MG set Gehalt = 19 where ...`

Join-Views

- Beispiel:

```
create view MGAL as
select Mitarbeiter, Gehalt, MGA.Abteilung, Leiter
from MGA, AL
where MGA.Abteilung = AL.Abteilung
```

- Insert in diese View nicht eindeutig übersetzbar:
insert into MGAL values ('Schuster', 30, 001, 'Boss')
→ **insert into MGA values ('Schuster', 30, 001)**
wenn kein Tupel (001, 'Boss') in AL existiert:
→ **insert into AL values (001, 'Boss')**
→ **update AL set Leiter='Boss' where Abteilung=001**
oder Fehlermeldung ?
- Daher: Join-View in SQL nicht updatable

Aggregation, group by, Subquery

- Auch bei Aggregation und Gruppierung ist es nicht möglich, eindeutig auf die Änderung in der Basisrelation zu schließen
- Subqueries sind unproblematisch, sofern sie keinen Selbstbezug aufweisen (Tabelle in from-Klausel der View wird nochmals in Subquery verwendet)

Eine View, die keiner der angesprochenen Problemklassen angehört, heisst ***Updatable View***. Insert, delete und update sind möglich.

Materialisierte View

Hier nur Begriffserklärung:

- Eine sog. materialisierte View ist **keine virtuelle** Relation sondern eine real gespeicherte
- Der Inhalt der Relation wurde aber durch eine Anfrage an andere Relationen und Views ermittelt
- In SQL einfach erreichbar durch Anlage einer Tabelle *MVName* und Einfügen der Tupel mit:
insert into *MVName* (select ... from ... where)
- Bei Änderungen an den Basisrelationen keine automatische Änderung in *MVName* und umgekehrt
- DBS bieten oft auch spezielle Konstrukte zur Aktualisierung (***Snapshot, Trigger***)

Rechtevergabe

- Basiert in SQL auf Relationen bzw. Views

- Syntax:

```
grant Rechteliste  
on Relation           bzw. View  
to Benutzerliste  
[with grant option]
```

- *Rechteliste*:

- all [privileges]
- select, insert, delete (mit Kommas sep.)
- update (optional in Klammern: Attributnamen)

Rechtevergabe

- *Benutzerliste:*
 - Benutzernamen (mit Passwort identifiziert)
 - **to public** (an alle)
- Grant Option:
Recht, das entsprechende Privileg selbst weiterzugeben
- Rücknahme von Rechten:
revoke *Rechteliste*
on *Relation*
from *Benutzerliste*
[restrict] *Abbruch, falls Recht bereits weitergegeben*
[cascade] *ggf. Propagierung der Revoke-Anweisung*