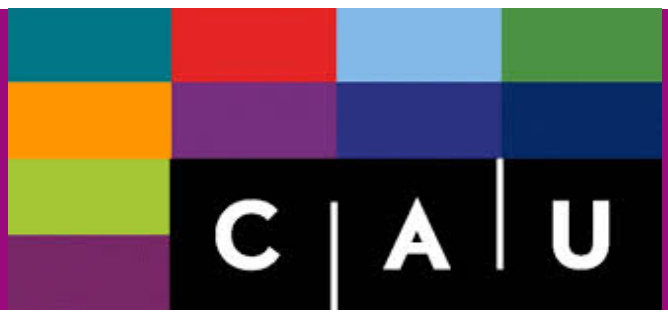
Informationssyteme[1]

Kapitel 4: Mehr SQL: Sortieren, Gruppieren und Views

Wintersemester 2020/21

Prof. Dr. Peer Kröger
Institut für Informatik
Arbeitsgruppe Informationssysteme und Data Mining



[1] Die folgenden Folien basieren in großen Teilen auf Material zur Vorlesung "Datenbanksysteme", die ich in meiner Zeit an der LMU München mehrmals gehalten habe. Das Material ist von den damaligen Kollegen maßgeblich (mit-)gestaltet worden, insbes. von Prof. Dr. C. Böhm.

Übersicht

1. Sortieren und Aggregieren
2. Gruppierung
3. Views

Agenda

1. Sortieren und Aggregieren
2. Gruppierung
3. Views

Sortieren

- In SQL wird mit dem Befehl

  `ORDER BY` A_1, A_2, ...

  nach den Attributen A_1 (zuerst), A_2 (bei "Unentschieden" in A_1), etc. sortiert
- Beispiel:

A	B
1	1
3	1
2	2
4	1
3	3

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

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

- `ORDER BY` steht am Ende der Anfrage
- Nur auf Attribute in `SELECT`-Klausel anwendbar

Sortieren

- Schlüsselwort `ASC` für aufsteigende (Default!) bzw. `DESC` für absteigende Sortierung

Beispiel: Sortieren

Gegeben:

MagicNumbers(Name: String, Wert: Int)

Primzahlen(Zahl: Int)

- Berechne alle MagicNumbers, die Primzahlen sind, sortiert nach dem Wert beginnend mit größtem (also absteigend)

```
SELECT * FROM MagicNumbers WHERE Wert IN
   (SELECT Zahl FROM Primzahlen)
   ORDER BY Wert DESC
```

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 der Spalte

- Aggregate können sich erstrecken auf ...
  - ... das gesamte Anfrageergebnis
  - ... einzelne Teilgruppen von Tupeln (siehe später)

Aggregation

- Aggregatfunktionen stehen in der `SELECT`-Klausel
- Ergebnis ist immer ein einzelnes Tupel: Keine Mischung aggregierte/nicht aggregierte Attribute möglich
- Aggregate wie `MIN` oder `MAX` sind ein einfaches Mittel, um Eindeutigkeit bei Subqueries herzustellen

Beispiel: Aggregation

Gesamtzahl und Durchschnitt der Einwohnerzahl aller Länder, die mit "B" beginnen:

```
SELECT SUM(Einw), AVG(Einw)
FROM Länder
WHERE LName LIKE 'B%'
```

Aggregation

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

Beispiel: Aggregation

Schema: Produkt(**PName**, Preis, ...)

Anfrage: Alle Produkte, mit unterdurchschnittlichem Preis

```
SELECT *
FROM Produkt
WHERE Preis < (SELECT AVG(Preis) FROM Produkt)
```

Agenda

1. Sortieren und Aggregieren
2. Gruppierung
3. Views

Intuition

- Aufteilung der Ergebnis-Tupel in Gruppen
- Ziel: Aggregationen
- Intuition an einem Beispiel:

  Gesucht: Gesamtgehalt und Anzahl der Mitarbeiter pro Abteilung

Mitarbeiter

PNr	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

Aggregationen:

Σ Gehalt	COUNT
6300	3
4200	2

- Das ist so in SQL aber nicht möglich, das Anfrageergebnis soll wieder eine Relation (mit flachen/atomaren Attributen) sein

Gruppierung

- Syntax der Gruppierung (`GROUP BY`-Klausel):

```
SELECT ...
FROM ...
[ WHERE ...  ]
GROUP BY A1, A2, ...
[ HAVING ...]
[ ORDER BY ...  ]
```

- Wegen Relationen-Eigenschaft des ErgebnissesEinschränkung der `SELECT`-Klausel: erlaubt sind
  - Attribute aus der Gruppierungsklausel (incl. arithmetischer Ausdrücke etc.)
  - Aggregationsfunktionen auch über andere Attribute, z.B. `COUNT (*)`
- Im Allgemeinen also `SELECT * FROM ... GROUP BY ...` nicht erlaubt

Gruppierung

Beispiel: Gruppierung

Gesucht (wie oben): Gesamtgehalt und Anzahl der Mitarbeiter pro Abteilung

Gegeben: Mitarbeiter-Tabelle

Mitarbeiter

PNr	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 Abteilung, SUM(Gehalt), COUNT(*)
FROM Mitarbeiter
GROUP BY Abteilung
```

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

Gruppierung

Beispiel: Gruppierung

Und so eben NICHT!!!

Mitarbeiter

PNr	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

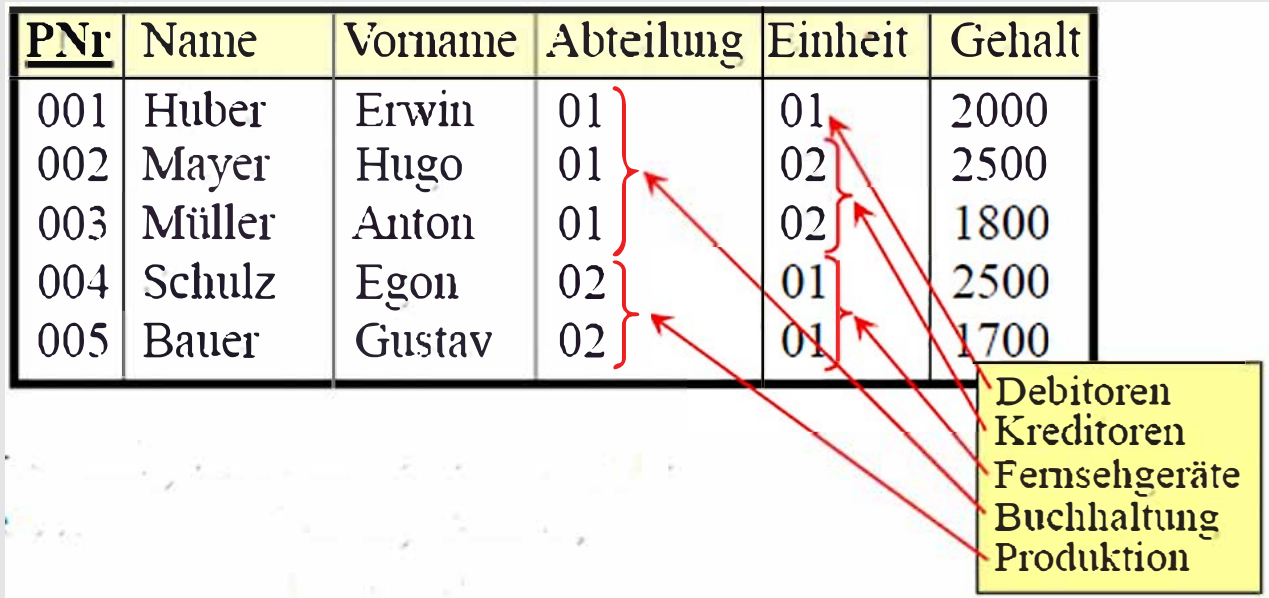
```
SELECT PNr, Abteilung, SUM(Gehalt)
FROM Mitarbeiter
GROUP BY Abteilung
```

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

Gruppierung mehrerer Attribute

Beispiel: Gruppierung mehrerer Attribute

PNr	Name	Vorname	Abteilung	Einheit	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



Gesucht: Gesamtgehalt in jeder Einheit:

```
SELECT Abteilung, Einheit, SUM(Gehalt)
FROM Mitarbeiter
GROUP BY Abteilung, Einheit
```

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

Die HAVING-Klausel

- Spezifikation von Eigenschaften der Gruppen
- Beispiel:
  Ermittle das Gesamt-Einkommen in jeder Abteilung, die mindestens 5 Mitarbeiter hat
- 1. Versuch:

```
SELECT ANr, SUM(Gehalt)
FROM Mitarbeiter
WHERE COUNT(*) >= 5
GROUP BY ANr
```

- Das ist so leider nicht möglich!
- Grund: Gruppierung wird erst nach den SELECT-FROM-WHERE-Operationen ausgeführt
- Stattdessen spezifiziert man diese Einschränkungen in der HAVING-Klausel

Die HAVING-Klausel

- Das Beispiel jetzt richtig:
  "Ermittle das Gesamt-Einkommen in jeder Abteilung, die mindestens 5 Mitarbeiter hat":

```
SELECT   ANr, SUM(Gehalt)
FROM     Mitarbeiter
GROUP BY ANr
HAVING COUNT(*) >= 5
```

- In diesem Zusammenhang: wie werden Gruppierungen eigentlich ausgewertet? – siehe nächste Folie ...

Auswertung der Gruppierung

- Dazu betrachten wir auch wieder ein 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“

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

Agenda

1. Sortieren und Aggregieren
2. Gruppierung
3. Views

Motivation

- Wir erinnern uns an das Thema Datenunabhängigkeit und deren Sicherstellung durch die spezielle 3-Ebenen-Architektur
- Jetzt geht es um die logische Datenunabhängigkeit: diese wurde durch die Entkopplung der logischen Ebene (Gesamtschau aller Daten, log. Schema) von der externen Ebene (Benutzersichten, externe Schemas) erreicht
- Außerdem: jede Anwendungsgruppe sieht nur die Daten, die sie ...
  - sehen will (Übersichtlichkeit)
  - sehen soll (Datenschutz)
- In SQL gibt es ein eigenes Konzept zur Realisierung der verschiedene externen Schemas: sog. Views

Was ist eine View

View (Sicht)

Eine View ist eine virtuelle Relation mit einem eigenen Schema deren Inhalt durch eine SQL Anfrage festgelegt wird.

- Was bedeutet virtuell?
  - Die View sieht für den Benutzer aus wie eine Relation
    - Man kann sie in Anfragen wie eine "normale" Relation verwenden:
      `SELECT ... FROM` $View_x$, $Relation_y$, ... `WHERE ...`
    - Mit Einschränkungen kann man auch `INSERT`, `DELETE`, und `UPDATE` auf ihr ausführen
  - Aber die Relation ist nicht real existent/gespeichert
  - Der Inhalt ergibt sich durch Berechnung aus anderen Relationen ("on-the-fly")

View in SQL definieren

- Syntax des DDL-Kommandos:

```
CREATE [/REPLACE] VIEW Name [(A_1, A_2, ...)]  AS
   SELECT ...  FROM ...  WHERE ...
```

- Dabei ist es eher unüblich, das Schema `(A_1, A_2, ...)` direkt anzugeben, da die `SELECT`-Klausel der Berechnungsvorschrift dies ja (implizit) bestimmt
- Views können natürlich auch gelöscht werden:

```
DROP VIEW Name
```

View in SQL definieren

Beispiel: View-Definition in SQL

Eine virtuelle Relation Buchhalter, nur mit den Mitarbeitern der Buchhaltungsabteilung

```
CREATE VIEW Buchhalter AS
   SELECT PNr, Name, Gehalt
   FROM Mitarbeiter WHERE ANr = 01
```

Bildlich:

Mitarbeiter

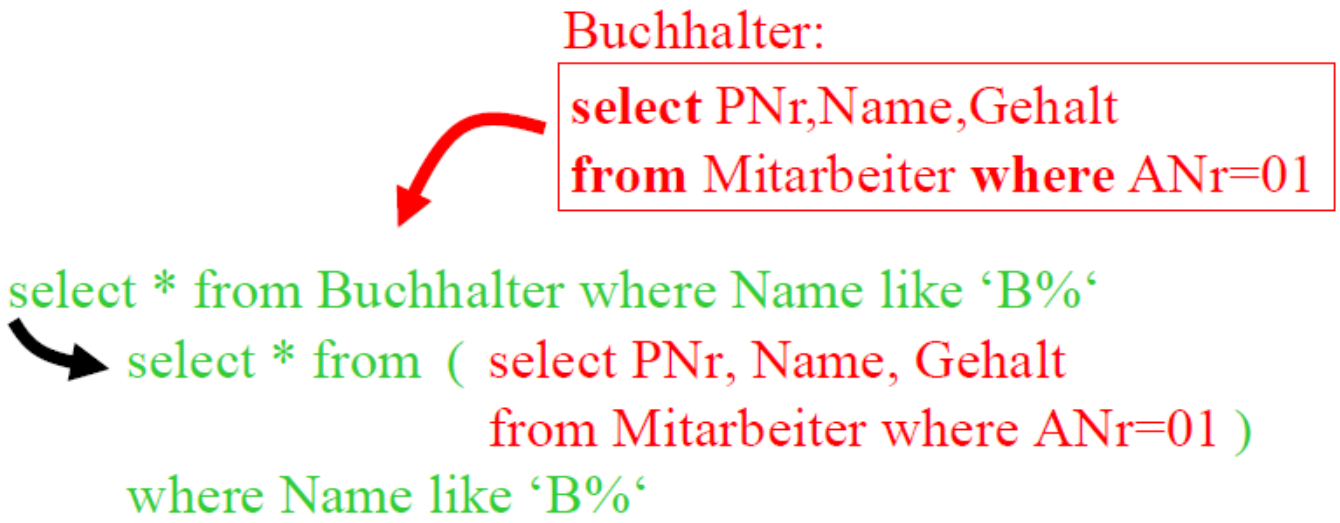
PNr	Name	Vorname	ANr	Gehalt
001	Huber	Erwin	01	2000
002	Mayer	Hugo	01	2500
003	Müller	Anton	01	1800
004	Schulz	Egon	02	2500
005	Bauer	Gustav	02	1700

Buchhalter

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

Virtuelle Relation

- Automatisch sind in der View `Buchhalter` alle Tupel der Basisrelation `Mitarbeiter`, 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:



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 (insbs. eben der externen Ebene)

View-Definition: erlaubte Konstrukte

- Folgende Konstrukte sind in Views erlaubt:
  - Selektion und Projektion (inkl. Umbenennung von Attributen, Arithmetik)
  - Kreuzprodukt und Join
  - Vereinigung, Differenz, Schnitt
  - Gruppierung und Aggregation
  - Die verschiedenen Arten von Subqueries
- Nicht erlaubt ist 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)
  - Dazu: 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. (d.h. die Daten des Tupel müssen entsprechend in die Basisrelationen eingefügt werden ...)
- Problem: 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 wenn Änderung auf Projektionssicht in Änderung der Basisrelation(en) transformiert werden
- Analog Selektions-Sichten, etc.
- Running Example: Basisrelationen
  - MGA (Mitarbeiter, Gehalt, Abteilung)
  - AL (Abteilung, Leiter)

Projektions-Sichten

- Beispiel View:

```
CREATE VIEW MA AS
    SELECT Mitarbeiter, Abteilung FROM MGA
```

- Keine Probleme beim Löschen und Update; z.B. Löschen:

```
DELETE FROM MA WHERE Mitarbeiter = ...
```

  wird zu

```
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 ('Werner', 001)
```

  wird zu

```
INSERT INTO MGA VALUES ('Werner', NULL, 001)
```

Projektions-Sichten

- 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:

```
UPDATE P SET y = 0 WHERE ...
```

- Was wäre der Wert von x in A?

  Die Nullstelle des Polynoms $y = 3x^3 + 2x^2 + x + 1$, die zu berechnen alles andere als trivial ist
- **Konsequenz**: Kein insert/delete/update bei `DISTINCT` und/oder Arithmetik in einer Projektions-Sicht

Selektions-Sichten

- Beispiel View:

```
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

Selektions-Sichten

- Tupel-Migration ist manchmal erwünscht: z.B.
  - Mitarbeiter wechselt den zuständigen Sachbearbeiter
  - Jeder Sachbearbeiter arbeitet mit der eigenen View
- Manchmal aber auch unerwünscht (z.B. Datenschutz)
- Deshalb gibt es in SQL folgende Möglichkeit:

```
CREAT VIEW MG AS
    SELECT * FROM MGA WHERE Gehalt >= 20
    WITH CHECK OPTION
```

- Tupel-Migration wird dann (mit Fehlermeldung) unterbunden

Join-Sichten

- Beispiel View:

```
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, 'Boss01')
```

daraus wird

```
INSERT INTO MGA VALUES ('Schuster', 30, 001)
```

und was ist mit AL?

  - Wenn kein (001, 'Boss01') in AL existiert:

```
INSERT INTO AL VALUES (001, 'Boss01')
```

  - Wenn schon aber anderes (001, 'BossAlt') in AL existiert:

```
UPDATE AL SET Leiter='Boss01' WHERE Abteilung = 001
```

  - Oder besser Fehlermeldung?

- **Daher:** Join-Sichten sind in SQL nicht updatable!

Was gibt es noch zu beachten?

- 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 Sichten

- Eine materialisierte View (materialized 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 ...`
- Bei Änderungen an den Basisrelationen keine automatische Änderung in MVName und umgekehrt
- Materialisierte Views sind ein Werkzeug des DB-Tunings: häufig angefragte, aber teuere Joins können in einer MV vorberechnet werden (Look-Up deutlich schneller als Join)
- DBS bieten oft auch spezielle Konstrukte zur Aktualisierung (sog. "Snapshot", "Trigger"), kein Standard-SQL

Rechtevergabe

- Basiert in SQL auf Relationen bzw. Views
- Syntax: Vergabe

```
GRANT Rechteliste
ON Relation
TO Benutzerliste
[WITH GRANT OPTION]
```

- Syntax: Rücknahme

```
REVOKE Rechteliste
ON Relation
FROM Benutzerliste
[RESTRICT]
[CASCADE]]
```

Rechtevergabe

- Rechteliste:
  - `ALL [PRIVILEGES]`
  - `SELECT, INSERT, DELETE` (ggfls. mit Komma getrennt)
  - `UPDATE` (optional in Klammern: Attributnamen)
- Benutzerliste:
  - Benutzername
  - `PUBLIC` (an alle)
- Grant Option ist das Recht, das entsprechende Privileg selbst weiterzugeben
- `RESTRICT` (bei Rücknahme) bedeutet Abbruch, falls Recht bereits weitergegeben
- `CASCADE` (bei Rücknahme) bedeutet ggfls. Propagierung der Revoke-Anweisung