



LUDWIG-
MAXIMILIANS-
UNIVERSITY
MUNICH



DEPARTMENT
INSTITUTE FOR
INFORMATICS



DATABASE
SYSTEMS
GROUP

Skript zur Vorlesung:

Einführung in die Informatik: Systeme und Anwendungen

Sommersemester 2016

Kapitel 3: Datenbanksysteme

Vorlesung: Prof. Dr. Christian Böhm

Übungen: Sebastian Goebel

Skript © Christian Böhm

<http://www.dbs.ifi.lmu.de/cms/> Einführung_in_die_Informatik_Systeme_und_Anwendungen

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

3.4 Mehr zu SQL

- Bisher haben wir die Umsetzung der Operatoren der relationalen Algebra in SQL kennen gelernt:
 - `SELECT` *Attribute*
 - `FROM` *Relationen*
 - `WHERE` *Bedingung*
 - sowie die Mengenoperationen (`UNION`, `EXCEPT`, ...)
- In diesem Kapitel lernen wir Erweiterungen von SQL kennen, die effektives Arbeiten ermöglichen, u.a.
 - Aggregationen
 - Sortieren und Gruppieren von Tupeln
 - Sichten
- Darüberhinaus gibt es weitere Konstrukte, die wir im Rahmen dieser Vorlesung nicht genauer vertiefen werden

3.4 Mehr zu SQL

- Subqueries

- An jeder Stelle in der select- und where-Klausel, an der ein konstanter Wert stehen kann, kann auch eine Subquery

(select...from...where...)

stehen.

- Diese Art heißt auch **direkte** Subquery
- 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 ...
    
```

3.4 Mehr zu SQL

- Subquery mit IN
 - Beispiel
 - Gegeben sind die Tabellen
 - MagicNumbers (Name: String, Wert: Int)
 - Primzahlen (Zahl: Int)
 - Anfrage: Alle MagicNumbers, die Primzahlen sind
select * from MagicNumbers where Wert in
(**select Zahl from Primzahlen**)
 - Anfrage: Alle MagicNumbers, die nicht prim sind
select * from MagicNumbers where Wert not in
(**select Zahl from Primzahlen**)

3.4 Mehr zu SQL

- Subquery mit IN (cont.)

- 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 und einsetzen
- Zuletzt Hauptquery auswerten

Existenz-Quantor

- Realisiert mit dem Schlüsselwort **exists**
- Der \exists -quantifizierte Ausdruck wird in einer **Subquery** notiert.
- Term **true** gdw. Ergebnis der Subquery **nicht leer**
- 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** L2.Partei='SPD'

)

3.4 Mehr zu SQL

- 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***

3.4 Mehr zu SQL

– 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
```

3.4 Mehr zu SQL

- 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

3.4 Mehr zu SQL

- 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

3.4 Mehr zu SQL

- 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)
```

3.4 Mehr zu SQL

- 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

3.4 Mehr zu SQL

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

3.4 Mehr zu SQL

- Syntax in SQL:
select ...
from ...
[where ...]
[group by A_1, A_2, \dots
[having ...]
[order by ...]
- 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...**

3.4 Mehr zu SQL

– Beispiel:

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~~

⇒ nicht möglich!!!

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

3.4 Mehr zu SQL

- 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 !

STATTDESSEN:

- Grund: Gruppierung wird erst nach den algebraischen Operationen ausgeführt

3.4 Mehr zu SQL

– Auswertung der Gruppierung an folgendem Beispiel:

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

1. Schritt:
from/where

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



2. Schritt:
Gruppenbildung

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



3. Schritt:
Aggregation

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



4. Schritt:
having (= Selektion)

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



5. Schritt:
Projektion

A	sum(C)
1	9

3.4 Mehr zu 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)**

3.4 Mehr zu SQL

- 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**

3.4 Mehr zu SQL

- Viewdefinition in SQL

- Das folgende DDL-Kommando erzeugt eine View relativ unüblich
create [or replace] view *VName* [(*A*₁, *A*₂, ...)] 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

3.4 Mehr zu SQL

- 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%'
```

3.4 Mehr zu SQL

- 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 (Stichwort: **Effekt-Konformität**, wird hier nicht vertieft)
- 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*

3.4 Mehr zu SQL

- 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

3.4 Mehr zu SQL

- Materialisierte View

- 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 *MV* und Einfügen der Tupel mit:
 - insert into *MV* (select ... from ... where)**
- Bei Änderungen an den Basisrelationen keine automatische Änderung in *MV* und umgekehrt
- DBS bieten oft auch spezielle Konstrukte zur Aktualisierung (***Snapshot, Trigger***)