

UNIT 0X0A

SQL I

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.

Hintergrund SQL

Ursprung

- Entwickelt bei IBM von Donald Chamberlin und Raymond Boyce 1970er im Zusammenhang mit System-R.
- Ursprünglicher Name SEQUEL (Structured English Query Language).

SQL besteht aus

- Datendefinitions (DDL)-,
- Datenmanipulations (DML)- und der
- Abfrage (Query)-Sprache.

Hintergrund SQL

Standard

- 1986 erstmalig standardisiert durch ANSI/ISO
- 1992 - neue Datentypen, neue JOIN-Syntax, ...
- 1999 - BOOLEAN, Objekt-relationale Erweiterungen, stored procedures
- 2003 - xml, Sequenz-Generatoren
- ...

<https://www.iso.org/committee/45342/x/catalogue/p/1/u/0/w/0/d/0>

© ISO/IEC 9075-1:2016

Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)

© ISO/IEC 9075-2:2016

Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)

© ISO/IEC 9075-2:2016/COR 1:2019

Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation) -

© ISO/IEC 9075-3:2016

Information technology – Database languages – SQL – Part 3: Call-Level Interface (SQL/CLI)

© ISO/IEC 9075-4:2016

Information technology – Database languages – SQL – Part 4: Persistent stored modules (SQL)

© ISO/IEC 9075-4:2016/COR 1:2019

Die verschiedenen DBMS implementieren, trotz "Standardisierung", häufig im Detail unterschiedliche Features. Man sollte generell *nicht* davon ausgehen, dass SQL-Code bzw. -Skripte, die etwa für MySQL laufen, auch auf einem Oracle DBMS laufen... Bestenfalls kann man sie einfach adaptieren.



Hintergrund SQL

Kritik

- Abweichung vom theoretischen Modell
 - Mengen vs. Listen
 - Reihenfolge von Tupeln relevant (z.B. LIMIT)
 - NULL im theoretischen Modell nicht existent
- Inkompatible Implementierungen (trotz Standardisierung!)
 - Darstellung von Zeit / Datum
 - Behandlung von NULL
 - nur PostgreSQL hat den Anspruch standard-nah zu sein, ist aber nicht standard-konform (es fehlt z.B. der Typ BLOB).

Übersicht

Themen

- Datentypen und -strukturen
- Anfragen/Select
- Ausdrücke und Funktionen
- Gruppieren/Aggregatfunktionen
- Optimierung
- Umbenennung
- Mengenoperationen
- Unterabfragen/Subselect
- With
- Korrelierte und unkorrelierte Anfragen
- IN/ALL/ANY/EXISTS
- Fallunterscheidung/CASE

Datentypen und -strukturen

Zeichenketten (*character string types*)

- **character[(n)] | char[(n)]**

Zeichenkette fester Länge n ; ggf. mit Leerzeichen aufgefüllt

- **character varying[(n)] | varchar[(n)] | char varying[(n)]**

Zeichenkette variabler Länge n

- **character large object [(n)] | clob [(n)]**

Zeichenkette variabler Länge n

Wird die Längenangabe weggelassen, so ist die Länge 1

- PostgreSQL, MS-SQL: Typ **text** anstelle von **clob**
- PostgreSQL: **varchar** ohne Längenangabe ist Zeichenkette beliebiger Länge
- Zulässige Maximallänge kann je nach Implementierung variieren
 - z.B. PostgreSQL: **text** max. 2GB; Oracle: **clob** max 4GB;
 - DB2: **varchr** max. 32Kb; **clob** max. 2 GB
- SQL-Typen wie "**national char**, **national varchar**" werden so gut wie von keiner Implementierung unterstützt.

Datentypen und -strukturen

Exakte numerische Typen

Ganze Zahlen / Zahlen mit fester Zahl an Nachkommastellen

- **numeric[p ,s]] | decimal[p ,s]] | integer | int | smallint**
p bezeichnet die Gesamtzahl der Stellen
s die Anzahl der Nachkommastellen, falls weggelassen gilt s=0

- **numeric** und **decimal** sind synonym
- **int** ist Kurzform zu **integer**
- **integer**: typischerweise 4 Byte
- **smallint**: typischerweise 2 Byte
- Maximale Genauigkeit von **decimal** / **numeric** implementierungsabhängig
 - PostgreSQL: 131.072 Vor- und 16.383 Nachkommastellen
 - MySQL: 65 Stellen
 - DB2: 31 Stellen
 - MS-SQL: 38 Stellen
 - Oracle: 38 Stellen - Oracle kennt nur **number**, nicht die SQL'99 Typen!

Datentypen und -strukturen

Annähernd numerische Typen

Fliesskommazahlen (Exponent, Mantisse)

- **real | double precision | float(*n*)**
n ist Genauigkeit in Bits

- **real** typischerweise 4 Byte (PostgreSQL, DB2, MySQL)
- **double precision** typischerweise 8 Byte (PostgreSQL, DB2, MySQL)
- Bereiche für *n*:
Oracle: 1-126
MS-SQL, MySQL,DB2,PostgreSQL:
1-23 Abbildung auf **real**; 24-53 Abbildung auf **double precision**
- Oracle kennt die Typen **real** und **double precision** nicht
real entspricht float(63), **double precision** entspricht float(126)

Datentypen und -strukturen

Bit Strings

Zeichenkette aus Bits ('0' und '1')

- **bits[(n)] | bits varying[(n)]**
n exakte (!) Länge bei **bits**, maximale Länge bei **bits varying**

- Bitstrings sind keine Zeichenketten (z.B.ASCII) - Abbildung auf Folge von Bytes
- kaum unterstützter SQL-Typ:
PostgreSQL, MS-SQL unterstützen Bitstrings,
Oracle, DB2 und MySQL jedoch nicht

Datentypen und -strukturen

Binäre Zeichenketten

- **binary large object[(n)] | blob[(n)]**

n bezeichnet maximale Größe in byte

- Längenangabe kaum unterstützt
- nicht durchgängig unterstützt;
Alternativen:
PostgreSQL: `bytea`
MS-SQL: `binary` bzw. `var binary`
- Maximale Größe implementierungsabhängig
DB2,MS-SQL: 2GB
Oracle: 4GB

Datentypen und -strukturen

Datums- und Zeittypen

- **date**
Jahr, Monat und Tag
- **time[(n)] [with timezone | without timezone]**
Stunde, Minute und Sekunde;
n Nachkommastellen der Sekunden-Komponente
alternativ: Zeitzonenumwandlung in Stunden (zu UTC)
- **timestamp [(n)] [with timezone | without timezone]**
Jahr, Monat, Tag, Stunde, Minute, Sekunde
n Nachkommastellen der Sekunden-Komponente
alternativ: Zeitzonenumwandlung in Stunden (zu UTC)

- with / without timezone kaum unterstützt
PostgreSQL: kompatibel
- Besonderheiten bei Oracle:
`date` entspricht `timestamp(0) without timezone` (also incl. Zeitangabe)
`time` ist nicht bekannt

Datentypen und -strukturen

Zeitintervalle

Zeiträume, Zeitdifferenzen

- **interval range**

range bezeichnet die Genauigkeit

YEAR | MONTH | DAY | HOUR | MINUTE | SECOND[(*n*)]

YEAR TO MONTH, DAY TO HOUR, DAY TO MINUTE, DAY TO SECOND [(*n*)]

- kaum unterstützt
PostgreSQL und Oracle kennen **interval**

Datentypen und -strukturen

Boole'scher Typ

- **boolean**
kann Werte **true** und **false** annehmen

- gut unterstützt
PostgreSQL, Oracle, DB2 und MySQL kennen **boolean**

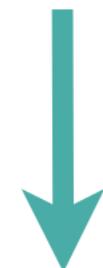
Abfragen / Select

- **select entspricht (fast) der Projektion, where der Selektion**
 - **select PersNr, Name from Professoren where Rang='C4'**
- **Wichtig: SQL eliminiert keine Duplikate**
 - **select Rang from Professoren**
7 Ergebnisse in SQL, da Duplikate nicht eliminiert werden
 - $\Pi_{\text{Rang}}(\text{Professoren})$
liefert hingegen nur 2 Ergebnisse
- **Elimination von Duplikaten mit distinct**
 - **select distinct Rang from Professoren**
liefert (wie gewünscht) 2 Ergebnisse

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

Abfragen / Select

- **Ergebnis kann nach Spalte(n) sortiert werden:**
 - `select PersNr, Name, Rang, Raum
from Professoren
order by Rang desc, Name asc;`



PersNr	Name ↑ ₂	Rang ↓ ₁	Raum
2136	Curie	C4	36
2137	Kant	C4	7
2126	Russel	C4	232
2125	Sokrates	C4	226
2134	Augustinus	C3	309
2127	Kopernikus	C3	310
2133	Popper	C3	52

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

Abfragen / Select

- **select-Klausel darf Ausdrücke (Berechnungen) enthalten**
 - Arithmetische Ausdrücke
 - Ausdrücke über Zeichenketten
 - Ausdrücke über Datum und Zeit
- **Vorgaben aus Standard eher "sparsam"**
 - viele Erweiterungen seitens der Hersteller
 - z.B. keine trigonometrischen Funktionen in SQL'99, SQL'03, aber in MySQL, PostgreSQL, DB2, MS-SQL

Ausdrücke und Funktionen

- **Arithmetische Ausdrücke**

- Operatoren +,- (auch als Präfix-Operator), * und /

```
select 1+1 from table
select 355/113.0 as PI from table
select preis + (preis*19)/100 as Brutto from Waren
```

- vordefinierte Funktionen in SQL'99
 - **abs(x)** Absolutwert
 - **mod(x,y)** Divisionsrest
 - zusätzlich in SQL'03
 - **floor(x)** Abrunden
 - **ceiling(x)** Aufrunden
 - **sqrt(x)** Quadratwurzel
 - **exp(x)** e^x
 - **ln(x)** natürlicher Logarithmus
 - **power(x,y)** x^y

Ausdrücke und Funktionen

- **Ausdrücke über Zeichenketten**

- Operatoren: `||` (Konkatenation)
 - SQL'99 setzt Zeichenkettentypen voraus; viele Implementierungen konvertieren ggf. automatisch

```
select 'An ' || Name from Studenten
```

- vordefinierte Funktionen in SQL'99
 - **position(a in b)** Position von Zeichenkette **a** in Zeichenkette **b**
 - **lower(s)** Zeichenkette **s** in Kleinbuchstaben
 - **upper(s)** Zeichenkette **s** in Grossbuchstaben
 - **substring(s from start [for len])**
Unterzeichenkette von **s** ab Position **start** mit Länge **len**
 - **char_length(s)** Länge der Zeichenkette **s**
 - **trim([leading | trailing | both] [characters] from s)**
Entfernt führende (**leading**), am Ende stehende (**trailing**), oder führende und am Ende stehende (**both**) Zeichen in der Liste **characters** aus der Zeichenkette **s**

Ausdrücke und Funktionen

- **Syntax**
string [NOT] LIKE pattern
- **pattern darf Sonderzeichen (Wildcards) enthalten**
 - '%' als Platzhalter für beliebig viele Zeichen
 - '_' als Platzhalter für ein Zeichen

- **Beispiele:**

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'    true
'abc' LIKE '_b_'   true
'abc' LIKE 'c'     false
```

- **Vorsicht mit führenden Wildcards:**

```
select distinct s.Name
from Vorlesungen as v, hören as h, Studenten as s
where s.MatrNr = h.MatrNr and
      h.VorlNr = v.VorlNr and
      v.Titel like '%thik%' // Kein Index nutzbar!
```

Ausdrücke und Funktionen

- **Ausdrücke über Datum und Zeit**
 - Literale
 - **DATE** '*year-month-day*'
 - **TIME** '*hour:minute:seconds*'
 - **TIMESTAMP** '*year-month-date hour:minute:second*'
 - Konstanten für aktuelles Datum / aktuelle Zeit
 - **CURRENT_DATE**
 - **CURRENT_TIME**
 - **CURRENT_TIMESTAMP**
 - Operatoren + und -

Ausdrücke und Funktionen

- **Automatische Konvertierung**
 - innerhalb der Zeichentypen, der exakten und der annähernd numerischen Typen
- **`cast(x as type)`** konvertiert Wert **x** in Typ **type**

		nach													
		EN	AN	VC	FC	VB	FB	D	T	TS	YM	DT	BO	CL	BL
von	EN	■	■	■	■								■		
	AN	■											■		
	C	■						■					■		
	B						■						■		
	D							■		■			■		
	T								■	■			■		
	TS							■	■	■			■		
	YM										■		■		
	DT											■	■		
	BO											■	■		
	BL	■											■		

EN exakt numerisch

AN annähernd numerisch

VC varchar

FC char

VB bits varying

FB bits

D Date

T time

TS timestamp

YM year-month interval

DT day-time interval

BO boolean

CL clob

BL blob

C character (fixed, variable)

B bit string (fixed, variable)

Ausdrücke und Funktionen

- bei Vergleichen und boole'schen Operationen wird **null** wie ein undefinierter Wert behandelt:

=	0	1	null
0	true	false	null
1	false	true	null
null	null	null	null

^	0	1	null
0	0	0	null
1	0	1	null
null	null	null	null

- **Konsequenz:**
 - Test auf null immer mit **is null** bzw. **is not null!**

Ausdrücke und Funktionen

- **COALESCE(v_1, \dots, v_n)**
gibt den ersten Wert ungleich NULL zurück
- **NULLIF(a, b)**
gibt NULL zurück, falls a und b gleich sind

Gruppierungen / Aggregatfunktionen

- **Aggregatfunktionen fassen alle Werte einer Spalte zusammen**
 - **min([distinct] A)**
Berechnung des Minimalwerts der Spalte **A** (optionales Schlüsselwort **distinct** hier bedeutungslos)
 - **max([distinct] A)**
Berechnung des Maximalwerts der Spalte **A** (optionales Schlüsselwort **distinct** hier bedeutungslos)
 - **avg([distinct] A)**
Berechnung des Durchschnittswerts der Spalte **A** (mit **distinct** gehen gleiche Werte nur einmal in die Berechnung ein)
 - **sum([distinct] A)**
Berechnung der Summe aller Werte in Spalte **A** (mit **distinct** gehen gleiche Werte nur einmal in die Berechnung ein)
 - **count(*)**
Zählen der Tupel der betrachteten Relation
 - **count([distinct] A)**
Zählen der Tupel der betrachteten Relation, bei **distinct** nach Duplikateliminierung bezüglich Spalte **A**

Gruppierungen / Aggregatfunktionen

- **Lediglich count(*) berücksichtigt NULL-Werte**
- **Vor Anwendung einer Aggregatfunktion werden NULL-Werte eliminiert**
 - ggf. soll eine Warnung ausgegeben werden
- **Sollen NULL-Werte berücksichtigt werden, sollte mit COALESCE gearbeitet werden**

B Aggregatfunktionen und NULL

- Gegeben sei die Tabelle **testAgg**

```
select avg(A),avg(B),count(*),count(A),count(B)  
from testAgg;
```



avg(A)	avg(B)	count(*)	count(A)	count(B)
100	133.333	4	4	3

testAgg	
A	B
150	150
0	null
200	200
50	50

Gruppierungen / Aggregatfunktionen

- **Aggregatfunktionen fassen alle Zeilen einer Tabelle zusammen**
- **Häufig sinnvoll: Aggregate über Teilmengen einer Relation berechnen**
 - => alle Tupel mit gleichen Attributwerten “landen im selben Topf”.
 - => Anwendung der Aggregatfunktion pro Topf
 - Beispiel:
 - Es soll gezählt werden, wie viele Studierende sich in jedem Semester befinden
- **Es wird ein Konstrukt benötigt, das**
 - Teilmengen einer (möglicherweise berechneten) Relation zu Gruppen zusammenfasst
 - weitere Operationen auf diesen Gruppen ausführt
 - entstandene Guppen ggf. filtert

Gruppierungen / Aggregatfunktionen

- **Syntax:**

```
select A1,..,An
from R1,...,Rk
where Pw
group by B1,...,Bm
having Ph
```

- **Fasst Zeilen, die in Spalten B₁,...,B_m identische Werte haben, zu Gruppe zusammen**
 - B₁,...,B_m bestimmen, wie 'Töpfe' gebildet werden - Ergebnis der Anfrage sind 'Töpfe', also Zusammenfassungen von Zeilen
 - **Daher:** Spalten, die nicht zur Gruppierung beitragen, dürfen nur im Kontext von Aggregatfunktionen verwendet werden! Aggregatfunktion wirkt dann innerhalb einer Gruppe
- **Gruppen werden abschließend mit Prädikat P_h gefiltert**
 - Beachte: Prädikat P_w der where-Klausel wird vor Gruppierung angewendet!

Gruppierungen / Aggregatfunktionen

B GROUP BY / HAVING

- ohne Angabe von **group by** werden **alle** Zeilen **zusammengefasst**:

```
// Durchschnittliche Semesterzahl aller Studenten  
select avg(Semester) from Studenten
```

- Gruppierung der Vorlesungen nach Professoren, die sie lesen:

```
// Lehrleistung der Professoren  
select gelesenVon,sum(SWS) from Vorlesungen group by gelesenVon
```

- Anschließende Filterung - Professoren, die mindestens 8 SWS leisten

```
// Professoren mit Lehrleistung >= 8 SWS  
select name,sum(SWS) from vorlesungen, professoren  
where (PersNr=gelesenVon)  
group by name  
having sum(SWS)>=8
```

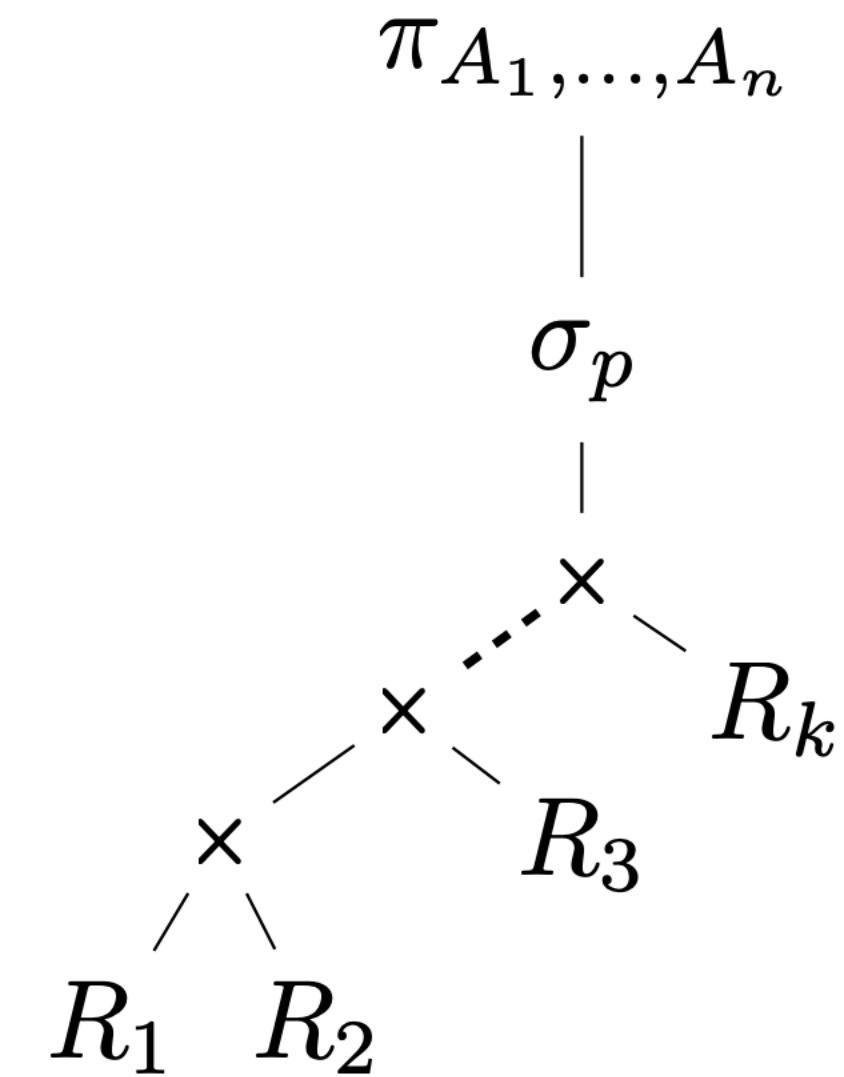
Optimierung

- **SQL-Anfragen können in Ausdrücke der rel. Algebra übersetzt werden:**

- **select** → Projektion
- **where** → Selektion
- **Komma** → Kreuzprodukt

$$\pi_{A_1, \dots, A_n} (\sigma_P (R_1 \times \dots \times R_k))$$

select distinct A_1, \dots, A_n
from R_1, \dots, R_k
where $P;$



Optimierung

- **Abarbeitung der `from`-Klausel**

Kartesisches Produktes über R_1, \dots, R_k :

```
Table cross_product(Table R1, Table R2)
  Table res(R1.cols concat R2.cols);
  foreach row1 in R1 // O(|R1| * |R2| )
    foreach row2 in R2
      res.appendRow( row1 concat row2 )
  return res;
```

```
select distinct A1, ..., An
from R1, ..., Rk
where P;
```

$$O\left(\prod_{i=1}^k |R_i|\right)$$

Laufzeit und Platz!

Zeit- / platzkritische Operation

- **Filterung mit Prädikat in `where`-Klausel (optional):**

```
Table filter(Table R, Predicate P)
  Table res(R.cols);
  foreach row in R // O(|R| )
    if ( P(row) )
      res.appendRow( row )
  return res;
```

$$O\left(\prod_{i=1}^k |R_i|\right)$$

Best case: $O\left(\log\left(\prod_{i=1}^k |R_i|\right)\right)$
(binäre Suche mit Index)

- **Projektion auf Spalten der `select`-Klausel (optional):**

```
Table project(Table R, Columns C)
  Table res(C);
  foreach row in R // O(|R| )
    res.appendRow( πC(row) )
  return res;
```

$$O\left(\prod_{i=1}^k |R_i|\right)$$

Umbenennung

Studenten		
MatrNr	Name	Semester
...

hören	
MatrNr	VorlNr
...	...

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
...

- **Erhöht die Lesbarkeit**

- `select Name, Titel // Welche Studenten hören welche Vorlesungen?
from Studenten, hören, Vorlesungen
where Studenten.MatrNr = hören.MatrNr and
hören.VorlNr = Vorlesungen.VorlNr;`
 - ohne **Kontext** ist unklar, welcher Tabelle die Spalten **Name** und **Titel** zuzuordnen sind.
- `select s.Name, v.Titel
from Studenten as s, hören as h, Vorlesungen as v
where s. MatrNr = h. MatrNr and
h.VorlNr = v.VorlNr`
 - s,h und v entsprechen den Tupelvariablen
 - **Beachte:** implizit verwendet SQL den Namen einer Relation als Tupelvariable (s.o. - hören.VorlNr)

- **Ist bei Doppeldeutigkeiten evtl. notwendig!**

Umbenennung

- **Welche Studenten kennen sich aus Vorlesungen?**
 - geben Sie alle Paare aus!

```
select s1.Name, s2.Name
from Studenten as s1,
     hoeren as h1, hoeren as h2,
     Studenten as s2
where h1.VorlNr = h2.VorlNr and
      h1.MatrNr = s1.MatrNr and
      h2.MatrNr = s2.MatrNr
```

26120 (Fichte),
 27550 (Schopenhauer) und
 29120 (Theophrastos) kennen sich

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

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

Mengenoperationen

- **Kombination von zwei SFW-Abfragen**
- **SQL unterstützt Mengenoperationen**
 - Vereinigung:
 - query₁ **UNION [ALL]** query₂
 - Differenz:
 - query₁ **EXCEPT [ALL]** query₂
 - Durchschnitt:
 - query₁ **INTERSECT [ALL]** query₂
- **Schachtelungen und Verkettungen sind möglich**
 - z.B. query₁ UNION query₂ UNION (query₃ INTERSECT query₄)
- **Durch den Zusatz **ALL** werden Duplikate **nicht** eliminiert**
 - MySQL: **EXCEPT** und **INTERSECT** nicht unterstützt

// Alle Studenten, die keine Hiwis sind:
(select Name from Studenten)
except
(select Name from Hiwis)

Mengenoperationen

- INTERSECT und EXCEPT können leicht simuliert werden:

// Alle Studenten, die gleichzeitig Hiwis sind:

(select Name from Studenten)

intersect

(select Name from Hiwis)

select Name **from** Studenten **join** Hiwis
using Name

// Alle Studenten, die keine Hiwis sind:

(select Name from Studenten)

except

(select Name from Hiwis)

select Name **from** Studenten **left join** Hiwis
using Name
where Hiwis.Name **IS NULL**

Unterabfragen / Subselect

- **Anfragen nicht nur über Mengenoperation kombinierbar**
- **Unterabfragen**
 - in **select**-Klausel
 - in **from**-Klausel
 - in **where**-Klausel
- **Operationen auf Unterabfragen die Mengen zurückliefern**
 - **Existenzquantor:** exists / not exists
 - **Inklusion:** in / not in
 - **Vergleich:** all / any / some

Unterabfragen / Subselect

- Unteranfrage in der select-Klausel
- Für jedes Ergebnistupel wird die Unteranfrage ggf. neu ausgeführt
- Beispiel:

```
// Lehrleistung der Professoren
select PersNr, Name, ( select sum (SWS)
    from Vorlesungen
    where gelesenVon=PersNr ) as Lehrbelastung
from Professoren;
```

Korrelation (später)

Unterabfragen / Subselect

- Unterabfrage kann einen Wert oder eine Liste liefern
- Sofern sie **einen Wert** liefert, kann Sie im where-Ausdruck wie eine Konstante eingesetzt werden:

```
// Prüfungen, die über dem Durchschnitt liegen
select * from prüfen
where Note < ( select avg (Note) from prüfen )
```

Unterabfragen / Subselect

- Verwertung der Ergebnismenge einer Unterabfrage

```
// Studenten, die mehr als zwei Vorlesungen hören
select tmp.MatrNr, tmp.Name, tmp.VorlAnzahl
from (select s.MatrNr, s.Name, count(*) as VorlAnzahl
      from Studenten as s, hören as h
     where s.MatrNr=h.MatrNr
   group by s.MatrNr, s.Name) as tmp
  where tmp.VorlAnzahl > 2
```

Wie viele
Vorlesungen hört ein
Student?



MatrNr	Name	VorlAnzahl
27550	Schopenhauer	2
25403	Jonas	1
28106	Carnap	4
29120	Theophrastos	3
26120	Fichte	1
29555	Feuerbach	2

Unterabfragen / Subselect

B "Marktanteil" der Vorlesungen 1 -> funktionale Dekomposition

- Anzahl der Studenten

```
select count(*) as GesamtAnz from Studenten
```

1

- Anzahl der Hörer pro Vorlesung

```
select VorlNr, count(*) as AnzProVorl from hören  
group by VorlNr
```

2

- Beides durch Kreuzprodukt auf eine Zeile bringen

```
select *  
from (select count(*) as GesamtAnz from Studenten) as g ,  
(select VorlNr, count(*) as AnzProVorl from hören  
group by VorlNr) as h
```

3

- Ergebnis in select berechnen

```
select h.VorlNr, cast(h.AnzProVorl as decimal(6,2)) / g.GesamtAnz as Marktanteil  
from (select count(*) as GesamtAnz from Studenten) as g ,  
(select VorlNr, count(*) as AnzProVorl from hören  
group by VorlNr) as h
```

4

WITH

B "Marktanteil" der Vorlesungen 2 -> Mehr Übersicht mit "with"

Marktanteil der Vorlesung - Zerlegung in Teilaufgaben / Lösungen dazu

- Anzahl der Studenten

```
select count(*) as GesamtAnz from Studenten
```

- Anzahl der Hörer pro Vorlesung

```
select VorlNr, count(*) as AnzProVorl from hören  
group by VorlNr
```

Teillösungen in "with"-Block; Synthese der Gesamtlösung:

```
with SummeStudenten as (select count(*) as GesamtAnz from Studenten),  
HörerProVorlesung as (select VorlNr, count(*) as AnzProVorl from hören  
group by VorlNr)
```

```
select VorlNr, cast( AnzProVorl as decimal(5,2)) / GesamtAnz  
from SummeStudenten,HörerProVorlesung
```

Unterabfragen in "with"-Block werden einmal ausgeführt

WITH

- In with eingeführte Bezeichner können um Schema-Definition erweitert werden
 - Dadurch liest sich eine Tabellendefinition fast wie eine Funktionsdefinition

```
with SummeStudenten as (select count(*) as GesamtAnz from Studenten),
      HörerProVorlesung as (select VorlNr, count(*) as AnzProVorl from hören
                            group by VorlNr)

select VorlNr, cast( AnzProVorl as decimal(5,2)) / GesamtAnz
from SummeStudenten,HörerProVorlesung
```

Typen ergeben sich aus
Ergebnisspalten des zugehörigen
Select!



```
with SummeStudenten(GesamtAnz)
      HörerProVorlesung(VorlNr,AnzProVorl) as (select count(*) from Studenten),
                                      as (select VorlNr, count(*) from hören
                                          group by VorlNr)

select VorlNr, cast( AnzProVorl as decimal(5,2)) / GesamtAnz
from SummeStudenten,HörerProVorlesung
```

WITH

- **Bekanntheitsgrad eines Professors**
 - Studenten kennen einen Professor, wenn Sie eine seiner Vorlesungen besuchen

B Bekanntheitsgrad

with

```
/* Studenten kennen Professor aus der Vorlesung.  
Beachte: distinct, da Student mehrere Vorlesungen eines Professors besuchen kann! */  
kennenSich(ProfessorId,MatrNr) as (select distinct gelesenVon, MatrNr  
from hören natural join Vorlesungen),  
/* Anzahl der unterschiedlichen Studenten, die einen Professor kennen  
Beachte: bereits eingeführter Bezeichner in folgenden with-Definitionen erlaubt */  
anzStudenten(ProfessorId,Anzahl) as (select ProfessorId, count(*)  
from kennenSich group by ProfessorId),  
/* Gesamtzahl der Studenten */  
gesamtStudenten(Gesamt) as (select count(*) from Studenten)  
  
select name, Anzahl *1.0 / Gesamt as Bekanntheitsgrad  
from anzStudenten, gesamtStudenten, Professoren  
where ProfessorId = Professoren.PersNr  
order by Bekanntheitsgrad desc
```

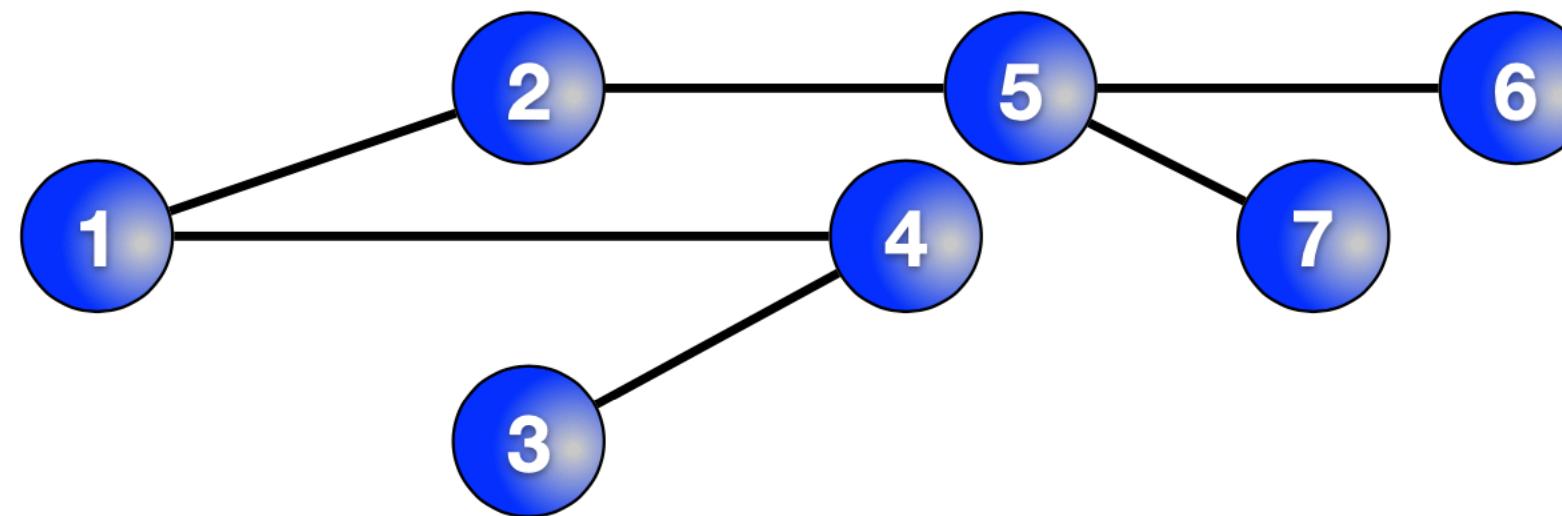
WITH

- **with erlaubt rekursive Definitionen**
- **Syntax:**
**WITH RECURSIVE *non_recursive_term* UNION [ALL | DISTINCT]
 *recursive_term***
- **rekursiver Aufruf nur in *recursive_term* erlaubt**
 - Konsequenz: durch den Zusatz "RECURSIVE" kann auf with-Bezeichner zugegriffen werden, die später im with-Block definiert werden
- **nur ein rekursiver Aufruf ist erlaubt!**
 - Fixpunktsemantik der Rekursion

WITH

- Transitive Hülle eines azyklischen Graphen

- Modellieren des Graphen durch Relation "Kanten"



Kanten	
V	N
1	2
2	5
1	4
4	3
5	6
5	7

- Zwei Knoten A und B sind miteinander verbunden, wenn
 - eine Kante sie direkt verbindet, also Kanten(A,B) existiert
 - es einen Zwischenknoten Z gibt, so dass A mit Z verbunden ist **und** eine Kante von Z nach B existiert

Verbunden(A,B) :- Kanten(A,B)

Verbunden(A,B) :- Verbunden(A,Z) , Kante(Z,B)

WITH

`Verbunden(A,B) :- Kanten(A,B)`

`Verbunden(A,B) :- Verbunden(A,Z) , Kante(Z,B)`

- **Umsetzung in SQL:**

```
with recursive Verbunden(A,B) as ( (select V,N from Kanten)
    union all
    (with V(A,Z) as (select * from Verbunden),
     K(Z,B) as (select * from Kanten)
      select A, B from V, K
      where V.Z = K.Z
    )
  )
select * from Verbunden;
```

WITH

```
with recursive Verbunden(A,B) as ( (select V,N from Kanten)
                                    union all
                                    (with V(A,Z) as (select * from Verbunden),
                                     K(Z,B) as (select * from Kanten)
                                     select A, B from V, K
                                     where V.Z = K.Z
                                    )
                                )
select * from Verbunden;
```

- Operationelle Abarbeitung (vereinfacht):

```
working_table = execute non_recursive_query
while (! working_table.empty) {
    tmp_table = execute recursive_query substitute Verbunden ↪ working_table
    working_table = tmp_table
}
```

- Konsequenz: terminiert nicht für zyklische Graphen

IN / ALL / ANY / EXISTS

- **Können in where-Klausel weiter bearbeitet werden**
 - Existenzquantor: exists / not exists
 - Inklusion: in / not in
 - Vergleich: all / any / some
- **Führen oft zu korrelierten Unterabfragen ...**

IN / ALL / ANY / EXISTS

- **Syntax:**
exists(subquery)
- **unärer Operator**
 - liefert true, falls *subquery* nicht die leere Tabelle liefert
 - *subquery* darf Variablen der umschließenden Anfrage referenzieren
 - dann spricht man von einer **korrelierten Unterabfrage**

```
// Professoren, die Vorlesungen halten  
select p.Name  
from Professoren p  
where exists ( select *  
               from Vorlesungen as v  
               where v.gelesenVon = p.PersNr )
```

Korrelation

Korrelierte Unterabfrage muss

- für jeden Eintrag neu ausgeführt werden
 $O(|\text{Professoren}| * |\text{Vorlesungen}|)$
- nicht vollständig ausgewertet werden
select-Klausel bedeutungslos

Wir werden diese Anfrage gleich optimieren

IN / ALL / ANY / EXISTS

- **Syntax:**

expression in (subquery)

- **binärer Operator**

- prüft, ob Ergebnis von *expression* in *subquery* vorkommt
- Voraussetzung: *subquery* liefert exakt eine Spalte
- **Vorsicht!** Ergebnis ist **null**, falls
 - *expression null* liefert, oder Ergebnis der *subquery null* enthält, denn:

$$a \text{ in } \{v_1, \dots, v_k\} := (a=v_1) \vee \dots \vee (a=v_k)$$

```
// Professoren, die Vorlesungen halten (nicht korreliert!)
```

```
select Name  
from Professoren  
where PersNr in ( select gelesenVon  
                  from Vorlesungen  
                )
```

- nicht korrelierte Unterabfrage wird einmal ausgewertet
evtl. nicht vollständig
- Inklusionstest (bei vorhandenem Index) in konstanter Zeit!
 $O(|\text{Professoren}|)$

IN / ALL / ANY / EXISTS

- **Korrelierte Formulierung**

```
select *  
from Studenten as S  
where exists ( select *  
                from Professoren as P  
                where P.GebDatum > S.GebDatum )  
// Studenten, die älter sind als mind. einer der Professoren
```

$$O(|S| * |P|)$$

- **Schlecht:** Student wird mit jedem Professor verglichen
- **Besser:** Finde zuerst den jüngsten Professor - Elimination der Korrelation

```
select *  
from Studenten as S  
where S.GebDatum < ( select max(GebDatum)  
                      from Professoren )
```

$$O(|S|)$$

IN / ALL / ANY / EXISTS

- Manchmal ist die Beseitigung einer Korrelation nicht ganz so einfach

```
select *  
from Assistenten as A  
where exists ( select *  
    from Professoren as P  
    where A.Chef = P.PersNr and P.GebDatum > A.GebDatum )  
  
// Assistenten, die älter als Ihre Chefs sind
```

- max hilft hier nicht, aber wir könnten einen join versuchen:

```
select *  
from (Assistenten as A join Professoren as p on A.Chef=P.PersNr)  
where P.GebDatum > A.GebDatum
```

- relationale Datenbanken sind für joins hochoptimiert
- join mit Hashtabellen in linearer Zeit möglich

IN / ALL / ANY / EXISTS

- **Syntax:**

expression operator any(subquery)

- **ternärer Operator**

- prüft, ob *subquery* einen Wert *value* enthält, so dass *expression operator value true* ergibt
- *subquery* muss Tabelle mit exakt einer Spalte zum Ergebnis haben
- *operator* muss im Ergebnis einen Wahrheitswert liefern (z.B. =, >=, <>)
- *any* und *some* sind synonym
- 'in' ist äquivalent zu '= any'
- **Vorsicht!** Ergebnis ist **null**, falls *expression* null oder Ergebnis von *subquery* null enthält, denn

$$a \text{ op } \text{any } \{v_1, \dots, v_k\} := (a \text{ op } v_1) \vee \dots \vee (a \text{ op } v_k)$$

IN / ALL / ANY / EXISTS

- **Syntax:**

expression op all(subquery)

- **ternärer Operator**

- prüft, ob *subquery* nur Werte *value* enthält, so dass *expression operator value true* ergibt
- *subquery* muss Tabelle mit exakt einer Spalte zum Ergebnis haben
- *operator* muss im Ergebnis einen Wahrheitswert liefern (z.B. =, >=, <>)
- 'not in' ist äquivalent zu '<> all'
- **Vorsicht!** Ergebnis ist **null**, falls *expression* null oder Ergebnis von *subquery* null enthält.

```
// dienstälteste Studenten
select Name
from Studenten
where Semester >= all ( select Semester
                           from Studenten
                         )
```

IN / ALL / ANY / EXISTS

- **SQL-92 kennt keinen Allquantor**
- **Allquantifizierung muss also durch eine äquivalente Anfrage mit Existenzquantifizierung ausgedrückt werden**

$$\forall x.P(x) \Leftrightarrow \neg\exists x.\neg P(x)$$

IN / ALL / ANY / EXISTS

B Umformung im Tupelkalkül

- **Wer hat alle vierstündigen Vorlesungen gehört?**

$$\{s \mid s \in \text{Studenten} \wedge \forall v \in \text{Vorlesungen} : (v.\text{SWS} = 4 \Rightarrow \exists h \in \text{hören} : (h.\text{VorlNr} = v.\text{VorlNr} \wedge h.\text{MatrNr} = s.\text{MatrNr}))\}$$

- **Elimination \forall**

$$\{s \mid s \in \text{Studenten} \wedge \neg(\exists v \in \text{Vorlesungen} : \neg(v.\text{SWS} = 4 \Rightarrow \exists h \in \text{hören} : (h.\text{VorlNr} = v.\text{VorlNr} \wedge h.\text{MatrNr} = s.\text{MatrNr}))))\}$$

- **Elimination \Rightarrow mit $a \Rightarrow b = \neg a \vee b$**

$$\{s \mid s \in \text{Studenten} \wedge \neg(\exists v \in \text{Vorlesungen} : \neg(\neg(v.\text{SWS} = 4) \vee \exists h \in \text{hören} : (h.\text{VorlNr} = v.\text{VorlNr} \wedge h.\text{MatrNr} = s.\text{MatrNr}))))\}$$

- **Regel von DeMorgan $\neg(a \vee b) = \neg a \wedge \neg b$**

$$\{s \mid s \in \text{Studenten} \wedge \neg(\exists v \in \text{Vorlesungen} : (v.\text{SWS} = 4 \wedge \neg(\exists h \in \text{hören} : (h.\text{VorlNr} = v.\text{VorlNr} \wedge h.\text{MatrNr} = s.\text{MatrNr}))))\}$$

IN / ALL / ANY / EXISTS

B Umsetzung in SQL

$$\{s \mid s \in \text{Studenten} \wedge \neg(\exists v \in \text{Vorlesungen} : (v.SWS = 4 \wedge \neg(\exists h \in \text{hören} : (h.VorlNr = v.VorlNr \wedge h.MatrNr = s.MatrNr))))\}$$

```
// Wer hat alle vierstündigen Vorlesungen gehört?  
select s.*  
from Studenten as s  
where not exists  
  (select *  
   from Vorlesungen as v  
   where v.SWS = 4 and  
         not exists (select * from hören as h  
                      where h.VorlNr = v.VorlNr and  
                      h.MatrNr=s.MatrNr  
    ))  
);
```

IN / ALL / ANY / EXISTS

- Allquantifizierung kann immer auch durch eine count-Aggregation ausgedrückt werden
 - Idee: Zählen aller Elemente, die P(x) erfüllen

B Allquantor und count

- Studenten, die mindestens 20% aller Vorlesungen besuchen

```
select MatrNr  
from hören  
group by MatrNr  
having count (*) >= 0.2*(select count (*) from Vorlesungen)
```

```
with /* Gesamtzahl aller Vorlesungen */  
    GesamtAnzahl(Gesamt) as (select count(*) from Vorlesungen),  
    /* Anzahl Vorlesungen, die ein Student hört (fast...) */  
    AnzahlProStudent(MatrNr,Anzahl) as (select MatrNr,count(*) from hören  
                                         group by MatrNr)  
select MatrNr,Name,Anzahl  
from (Studenten natural join AnzahlProStudent) cross join GesamtAnzahl  
where Anzahl >= 0.2*(select Gesamt from GesamtAnzahl)
```

IN / ALL / ANY / EXISTS

- **Ermitteln Sie die Studenten, die weniger als 10% aus dem Vorlesungsangebot hören**
 - Denken Sie daran: es gibt auch faule Studenten oder solche die Urlaub machen

B Allquantor und count

```
with GesamtAnzahl(Gesamt) as (select count(*) from Vorlesungen),
/* Studenten, die keine Vorlesung besuchen */
StudentenOhne(MatNr) as ((select MatrNr from Studenten)
except
(select MatrNr from hören)),
AnzahlProStudent(MatNr,Anzahl) as ( (select MatrNr,count(*) from hören
group by MatrNr)
union /* Hinzunahme der "Faulen ... */
(select MatrNr,0 from StudentenOhne))
select MatrNr,Name,Anzahl from (Studenten natural join AnzahlProStudent),GesamtAnzahl
where Anzahl <= 0.1*(select Gesamt from GesamtAnzahl)
```

CASE

- **Fallunterscheidung von mit Case**

```
select MatrNr, ( case when Note < 1.5 then 'sehr gut'  
                      when Note < 2.5 then 'gut'  
                      when Note < 3.5 then 'befriedigend'  
                      when Note < 4.0 then 'ausreichend'  
                      else 'nicht bestanden'  
                  end )  
from prüfen
```

- **Achtung:** die **erste** passende when-Klausel wird ausgeführt
- möglichst so formulieren, dass Reihenfolge irrelevant (also nicht, wie oben!)