Informationssysteme

Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

SQL

SQL

- **Deklarative** Anfragesprache ("was" nicht "wie")
- **Inspiriert durch das relationale Tupelkalkül**
- Setzte sich aus drei Sprachen zusammen:
- **Datendefinitions (DDL)-Sprache:**
  - erstellt/ändert das Schema
  - create, alter, drop
- **Datenmanipulations (DML)-Sprache**
  - ändert Ausprägungen
  - insert, update, delete
- **Anfragesprache:**
  - berechnet Anfragen auf den Ausprägungen
  - select * from where ...

Relationen vs. Tabellen

- In SQL betrachten wir aber Tabellen
  - Tabellen können Duplikate enthalten
  - Tabellen können eine Ordnung haben
  - Tabellen haben nicht unbedingt einen Schlüssel

Es war einmal ..... die Geschichte von SQL

- 1986 erste SQL-Norm durch ANSI verabschiedet
- SQL-92 (SQL 2)
- SQL-99 (SQL 3): rekursive Anfragen, Trigger, OO
- SQl-2003: XML, Fensteranfragen
- SQL-2006: XML, XQuery
- SQL-2008: Verschiedenes
- Standards von verschiedenen DBMS nicht vollständig implementiert
- DBMS haben oft zusätzlich eigene SQL-Erweiterungen

Die Datenbanksysteme (nicht vollständig)

- 1970: Relationales Modell, Codd
- seit 70er Jahren System R, IBM (SEQUEL $\rightarrow$ SQL)
- 1979: Oracle 2 (erste Version von Oracle)
- 1982: IBM DB2
- 1984: Teradata
- 1985: Informix
- 1987: Sybase
- 1989: Postgres
- 1989: Microsoft SQL Server
- 1996: MySQL
- 2004: Apache Derby
- 2004: MonetDB
- 2007: H-Store
- 2010: HyPerDB
- 2012: VoltDB ...

In dieser Vorlesung wird Postgresql verwendet

... in Beispielen in den Folien sowie in den Übungen

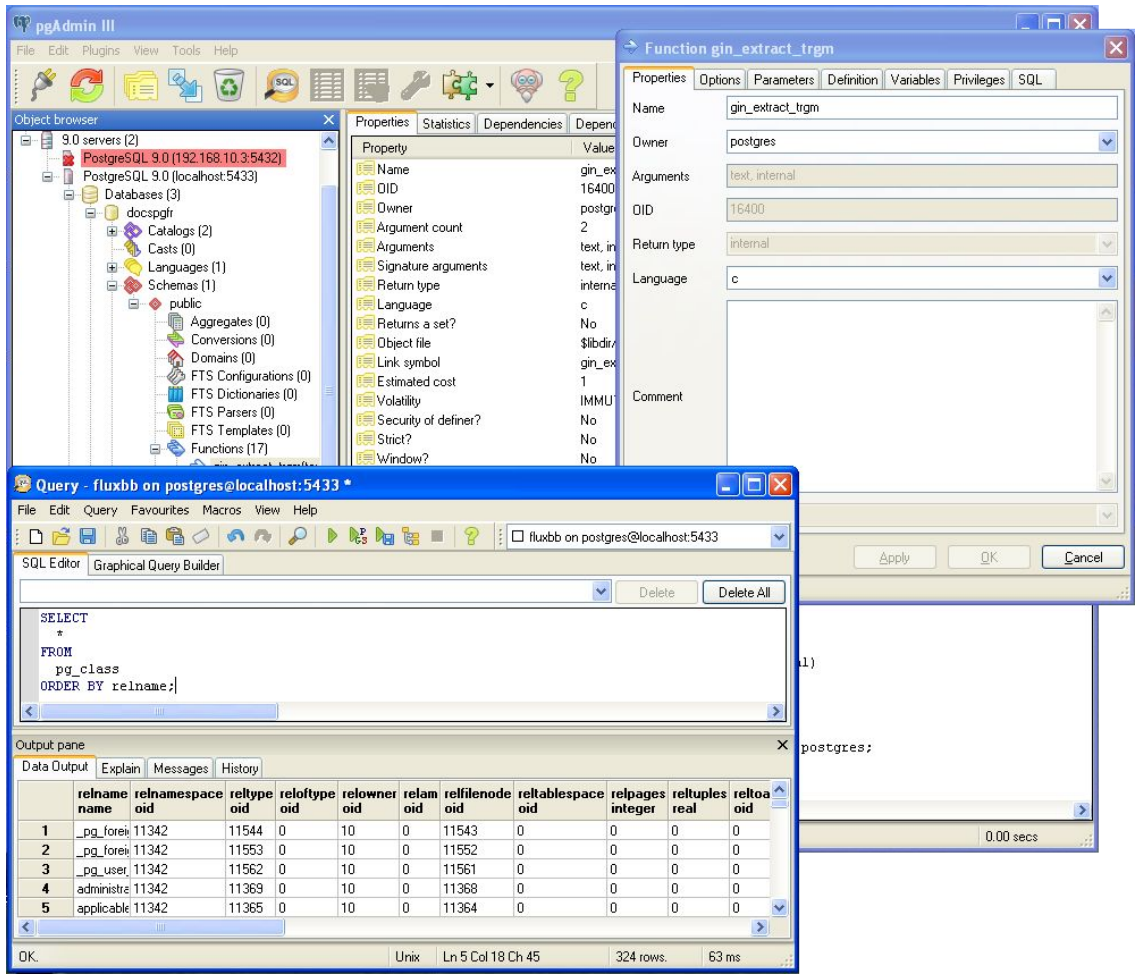
Postgresql

- "The world's most advanced open source database"
- Einfach zu Installieren
- Gute SQL Unterstützung
- Transaktionen
- Gute Dokumentation (es gibt auch Bücher dazu)

http://www.postgresql.org/

PGAdmin3

User Interface: Download unter http://www.pgadmin.org/



Einfache Datendefinitionen in SQL

Datentypen

- **character**(n), **char** (n)
- **character varying** (n), **varchar** (n)
- **numeric**(p,s), **integer**
- **blob** oder **raw** für sehr große binäre Daten
- **clob** für sehr große String-Attribute
- **date** für Datumsangaben
- ...

`http://www.postgresql.org/docs/9.4/static/datatype.html`
`http://www.postgresql.org/docs/9.4/static/sql-syntax.html`

Datendefinitions (DDL) Sprache

Tabellen erstellen:

```
create table Professoren(
        PersNr      integer,
        Name        varchar (10)
        Rang        character (2));
```

Tabellen verändern:

```
alter table Professoren
      add (Raum integer);

alter table Professoren
      modify (Name varchar(30));
```

`http://www.postgresql.org/docs/9.4/static/ddl-alter.html`

Datendefinitions (DDL) Sprache

Tabellen löschen:

drop table Professoren;

Datenmanipulationssprache: insert

```
insert into Studenten (MatrNr, Name)
            values (28121, 'Archimedes');
```

```
insert into Studenten (Name)
            values ('Meier');
```

Ohne Angabe der Spaltennamen ist die Reihenfolge der Attribute relevant!

```
insert into hören
            select MatrNr, VorlNr
            from Studenten, Vorlesungen
            where Titel = 'Logik';
```

`http://www.postgresql.org/docs/9.4/static/dml.html`

Anfragesprache

`http://www.postgresql.org/docs/9.4/static/queries.html`

select <Liste von Spalten>
from <Liste von Tabellen>
where <Bedingung>;

select * wählt alle verfügbaren Spalten aus!

Relationale Algebra $\rightarrow$ SQL

- Projektion $\pi \rightarrow$ select
- Kreuzprodukt $\times \rightarrow$ from
- Selektion $\sigma \rightarrow$ where

Select from where ...

select PersNr, Name
from Professoren
where Rang = 'C4';

Professoren	
PersNr	**Name**
2125	Sokrates
2126	Russel
2136	Curie
2137	Kant

Professoren			
PersNr	**Name**	**Rang**	**Raum**
7 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

Anmerkung:

SQL legt nicht fest in welcher Reihenfolge Selektion, Projektion oder Join ausgeführt werden.

Values Statement

- Erzeugt konstante Tabelle
- **values** (1, 'eins'), (2, 'zwei'), (3, 'drei');
- Dann z.B. select * from (values (1,'eins')) s;
- Achtung: in Postgres braucht values in FROM einen Alias (hier s)
- Values Statement 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);
```

Prädikate in der **where** Klausel

Beschreibt Bedingungen die für Ergebnistupel gelten müssen.

- Bedingungen in der where-Klausel können logisch kombiniert werden mit: **and, or, not**
- Vergleichsoperatoren sind u.a.: $=, <, \leq, >, \geq, like, between$

Between und Like

select Name, MatrNr
from Studenten
where semester **between** 1 **and** 4

Könnte auch mit Vergleichsoperatoren ausgedrückt werden ...

select Name, Matr
from Studenten
where Name **like** 'M%'

% steht für eine beliebige Zeichenkette (auch Länge 0).
Es gibt auch _
für EIN beliebiges Zeichen.

Sortierung (Order By) und Top Ergebnisse (Limit)

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

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

Möchte man nur die ersten k (hier =5) Ergebnisse haben: Limit Anweisung

```
select    Name, Semester
from      Studenten
order by  Semester desc
limit     5
```

Limit hat leicht andere Syntax in versch. DBS, z.B. Oracle, SQL Server.

Duplikateliminierung

```
select Rang
from Professoren;
```

Rang
C4
C3
C3
C4
C4
C3
C4

```
select distinct Rang
from Professoren
```

Rang
C4
C3

Anfragen über mehrere Relationen

Welcher Professor liest "Mäeutik"?

select Name, Titel
from Professoren, Vorlesungen
where PersNr = gelesenVon **and** Titel = 'Mäeutik'

In der relationalen Algebra sieht das wie folgt aus:

$$\pi_{Name,Titel}(\sigma_{PersNr=gelesenVon \wedge Titel='Mäeutik'}(Professoren \times Vorlesungen))$$

Und im Tupelkalkül....?

Anfragen über mehrere Relationen

Professoren			
PersNr	**Name**	**Rang**	**Raum**
2125	Sokrates	C4	226
2126	Russel	C4	232
...	...	...	...
2137	Kant	C4	7

Vorlesungen			
VorlNr	**Titel**	**SWS**	**gelesenVon**
5001	Grundzüge	4	2137
5041	Ethik	4	2125
...	...	...	...
5049	Mäeutik	2	2125
...	...	...	...
4630	Die 3 Kritiken	4	2137

Verknüpfung durch Kreuzprodukt

Ergebnis des Kreuzprodukts:

PersNr	Name	Rang	Raum	VorlNr	Titel	SWS	gelesenVon
2125	Sokrates	C4	226	5001	Grundzüge	4	2137
1225	Sokrates	C4	226	5041	Ethik	4	2125
...	...	...	...	...	...	...	...
2125	Sokrates	C4	226	5049	Mäeutik	2	2125
...	...	...	...	...	...	...	...
2126	Russel	C4	232	5001	Grundzüge	4	2137
2126	Russel	C4	232	5041	Ethik	4	2125
...	...	...	...	...	...	...	...
2317	Kant	C4	7	4630	Die 3 Kritiken	4	2137

Was fällt auf?

Kreuzprodukt erzeugt VIELE unnötige Kombinationen!

Nun: für die Auwahl der interessanten Tupel brauchen wir die Selektion $\rightarrow$

Nach Anwenden der Selektion "PersNr=gelesenVon" und "Titel= Mäeutik" erhalten wir:

PersNr	Name	Rang	Raum	VorlNr	Titel	SWS	gelesenVon
2125	Sokrates	C4	226	5049	Mäeutik	2	2125

Weiter mit der Projektion auf Name und Titel:

Name	Titel
Sokrates	Mäeutik

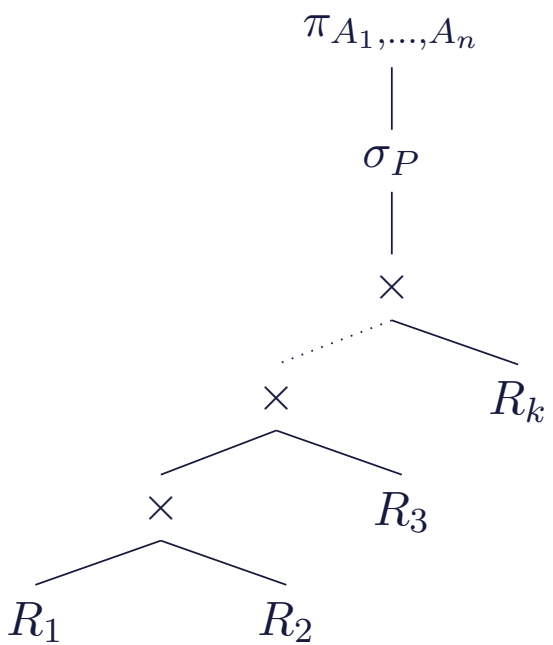
Übersetzung von SQL in die relationale Algebra

Allgemeine Form einer (ungeschachtelten) SQL-Anfrage:

select A_1, ..., A_n
from R_1, ..., R_k
where P;

Übersetzung in die relationale Algebra:

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



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;
```

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

Was es bei der Aggregation zu Beachten gilt

- SQL erzeugt pro Gruppe ein Ergebnistupel
- Also **müssen** alle in der **select**-Klausel aufgeführten Attribute - außer den aggregierten - auch in der **group by**-Klausel aufgeführt werden.

So geht es also nicht:

select Rang, count(PersNr), Name
from Professoren
group by Rang

Welcher Name sollte hier auch für einen bestimmten Rang zurückgegeben werden?
Es muss sichergestellt werden, dass sich das Attribut innerhalb einer Gruppe nicht ändert.

Was es bei der Aggregation zu Beachten gilt

- Aber: **umgekehrt** muss dies **nicht** notwendigerweise gelten.
- D.h. wenn ein Attribut im group by steht, muss es nicht unbedingt im select aufgeführt werden!

Beispiel:

select Name
from Professoren
group by Name, Rang

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**.

Ausführen einer Anfrage mit group by

Vorlesung × Professoren							
VorlNr	**Titel**	**SWS**	**gelesenVon**	**PersNr**	**Name**	**Rang**	**Raum**
5001	Grundzüge	4	2137	2125	Sokrates	C4	226
5041	Ethik	4	2125	2125	Sokrates	C4	226
...	...	...	...	...	...	...	...
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓

where

Vorlesung × Professoren							
VorlNr	**Titel**	**SWS**	**gelesenVon**	**PersNr**	**Name**	**Rang**	**Raum**
5001	Grundzüge	4	2137	2137	Kant	C4	7
5041	Ethik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnisth.	3	2126	2126	Russel	C4	232
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5052	Wissenschaftsth.	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

group by

Vorlesung × Professoren							
VorlNr	**Titel**	**SWS**	**gelesenVon**	**PersNr**	**Name**	**Rang**	**Raum**
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnisth.	3	2126	2126	Russel	C4	232
5052	Wissenschaftsth.	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

having-Bedingung

Vorlesung × Professoren							
VorlNr	**Titel**	**SWS**	**gelesenVon**	**PersNr**	**Name**	**Rang**	**Raum**
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

Aggregation **sum** und Projektion

gelesenVon	**Name**	**sum(SWS)**
2125	Sokrates	10
2137	Kant	8

Mengenoperationen: union

```
(select Name
from Assistenten)
union
(select Name
from Professoren);
```

```
(select Name
from Assistenten)
union
(select Name
from Assistenten);
```

```
(select Name
from Assistenten)
union all
(select Name
from Assistenten);
```

union eliminiert Duplikate, union all hingegen nicht.

Mengenoperationen: intersection und minus bzw. except

A
x
1
2
3

B
x
2
3
4

```
(select * from A)
intersect
(select * from B);
```

A
x
2
3

```
(select * from A)
except
(select * from B);
```

A
x
1

intersect all, except all

Wie bei union muss man bei intersect und except ein **all** dazuschreiben, wenn man Multimengensemantik möchte.

Zum Beispiel ergibt

```
(values (1),(1),(1),(1))
      except
(values (1),(1))
```

eine leere Tabelle, aber mit **except all** kommt eine Tabelle mit zwei Tupeln der Form $\langle 1 \rangle$ heraus

where .... in: Beispiel Unkorrelierte Unteranfragen

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

Vergleich von Wert mit Menge von Werten

where exists: Beispiel Korrelierte Unteranfragen

```
select Name
from Professoren
where exists (select *
              from Vorlesungen v
              where v.gelesenVon = p.PersNr );
```

Was passiert, wenn man das select der Unteranfrage ändert?

where exists: Beispiel Korrelierte Unteranfragen (2)

```
select Name
from Professoren p
where exists (select 42
              from Vorlesungen v
              where v.gelesenVon = p.PersNr );
```

Ist das Ergebnis ein anderes?

Ist die innere Menge leer?

where ..... any

Vergleich von Wert mit Menge von Werten

```
select Name, Semester
from Studenten
where  Semester > any (select Semester
                       from Studenten);
```

where ..... all

Vergleich von Wert mit Menge von Werten

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

Was passiert bei Vergleich mit $>$ anstelle von $>=$?

Allquantifizierung

SQL hat keinen Allquantor

- **Allquantifizierung muss durch eine äquivalente Anfragen mit Existenzquantifizierung ausgedrückt werden.**
- Tupel-Kalkül-Formulierung der Anfrage: Wer hat alle vierstündigen Vorlesungen gehört?

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

Eliminierung durch $\Rightarrow$ und $\forall$ durch Äquivalenzen

- $\forall t \in R(P(t)) = \neg(\exists t \in R(\neg P(t)))$
- $X \Rightarrow Y = \neg X \vee Y$

Umformung des Kalkül-Ausdrucks ...

Elimination $\forall$:

$$\{s|s \in Studenten \wedge \neg(\exists v \in Vorlesungen\ \neg(v.SWS = 4 \Rightarrow \exists h \in hören$$

$$(h.VorlNr = vl.VorlNr \wedge h.MatrNr = s.MatrNr)))\}$$

Elimination $\Rightarrow$:

$$\{s|s \in Studenten \wedge \neg(\exists v \in Vorlesungen\ \neg(\neg(v.SWS = 4) \vee \exists h \in hören$$

$$(h.VorlNr = vl.VorlNr \wedge h.MatrNr = s.MatrNr)))\}$$

Durch Anwendung von DeMorgan erhalten wird:

$$\{s|s \in Studenten \wedge \neg(\exists v \in Vorlesungen(v.SWS = 4 \wedge \neg(\exists h \in hören$$

$$(h.VorlNr = vl.VorlNr \wedge h.MatrNr = s.MatrNr))))\}$$

Umsetzung in SQL:

$$\{s|s \in Studenten \wedge \neg(\exists v \in Vorlesungen(v.SWS = 4 \wedge \neg(\exists h \in hoeren$$

$$(h.VorlNr = vl.VorlNr \wedge h.MatrNr = s.MatrNr))))\}$$

```
select s.*
from Studenten s
where not exists
  (select *
  from Vorlesungen v
  where v.SWS=4 and not exists (
    select *
    from hoeren h
    where h.VorlNr=v.VorlNr and h.MatrNr=s.MatrNr ));
```

Allquantifizierung durch count-Aggregation

Allquantifizierung kann auch immer durch eine count-Aggregation ausgedrückt werden:

```
select h.MatrNr
from hoeren h, vorlesungen v
where h.vorlNr = v.vorlNr and v.sws = 4
group by h.MatrNr
having count(*) = (select count(*)
                   from Vorlesungen
                   where sws=4);
```

Unteranfragen in where-Klausel

```
select *
from pruefen
where Note < (select avg(Note)
              from pruefen);
```

Vergleich von Wert mit Wert

Unteranfragen in select-Klausel

- **Für jedes Ergebnistupel wird die Unteranfrage ausgeführt**
- Unteranfrage ist korreliert (greift auf Attribute der umschließenden Anfrage zu)

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

Pro äußerer Zeile wird ein Wert berechnet

Unkorrelierte vs. korrelierte Unteranfragen

Korrelierte Formulierung

```
select s.*
from Studenten s
where exists
        (select p.*
        from Professoren p
        where p.GebDatum > s.GebDatum);
```

Unkorrelierte vs. korrelierte Unteranfragen

Äquivalente unkorrelierte Formulierung

```
select s.*
from Studenten s
where s.GebDatum <
        (select max(p.GebDatum)
        from Professoren p);
```

- **Vorteil: Eregebnis der Unteranfrage kann materialisiert werden**
- **Unteranfrage braucht nur einmal ausgewertet zu werden**

Entschachtelung korrelierter Unteranfragen

```
select a.*
from Assistenten a
where exists
            (select p.*
            from Professoren p
            where a.Boss = p.PersNr and
                  p.GebDatum > a.GebDatum);
```

Entschachtelung durch Join

```
select a.*
from Assistenten a, Professoren p
where a.Boss=p.PersNr and p.GebDatum > a.GebDatum);
```

WITH statements

```
with AnzahlStudentenJeVL as (
select VorlNr, count(*) as AnzProVorl
from hoeren
group by VorlNr
),
GesamtanzahlStudenten as (
select count (*) as GesamtAnz
from Studenten
)
select h.VorlNr, h.AnzProVorl, g.GesamtAnz,
h.AnzProVorl/cast(g.GesamtAnz as float) as Marktanteil
from AnzahlStudentenJeVL h,
GesamtanzahlStudenten g;
```

Erzeugt temporäre Tabelle innerhalb der Anfrage

Null Werte

- **null entspricht "unbekannter Wert", "nicht definiert"**
- **Nullwerte können auch im Zuge der Anfrageauswertung entstehen (z.B. äußere Joins)**
- Manchmal sehr überraschende Anfrageergebnisse, wenn Nullwerte vorkommen. **Beispiel:**

  select count (*)
  from Studenten **where** Semester $<$ 13 **or** Semester $>=$ 13;
- Wenn es Studenten gibt, deren Semester-Attribut den Wert null hat, werden diese nicht mitgezählt.
- Der Grund liegt in folgenden Regeln für den Umgang mit null- Werten begründet.

COUNT und Null Werte

Bei count(spaltenname) werden Null Werte nicht mitgezählt, ebenso wie bei count(distinct spaltenname).

Bei count(*) werden hingegen auch Tupel, die nur aus NULL Werten bestehen mitgezählt.

“select count(*) from (select null as x) as y” liefert also 1 und
“select count(x) from (select null as x) as y” liefert 0

Auswertung bei Null-Werten

- In arithmetischen Ausdrücken werden Nullwerte propagiert, d.h. sobald ein Operand null ist, wird auch das Ergebnis null. Dementsprechend wird z.B. null + 1 zu null ausgewertet-aber auch null * 0 wird zu null ausgewertet.
- **SQL hat eine dreiwertige Logik**, die nicht nur **true** und **false** kennt, sondern auch einen dritten Wert **unknown**. Diesen Wert liefern Vergleichsoperationen zurück, wenn mindestens eines ihrer Argumente null ist. Beispielsweise wertet SQL das Prädikat (PersNr=...) immer zu unknown aus, wenn die PersNr des betreffenden Tupels den Wert null hat.
- Logische Ausdrücke werden nach den folgenden Tabellen berechnet:

Auswertung bei Null-Werten

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

Das case-Konstrukt

```
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 pruefen;
```

Die erste qualifizierende when-Klausel wird ausgeführt.

Joins

- **cross join**: Kreuzprodukt
- **natural join:** natürlicher Join
- **join oder inner join:** Theta-Join
- **left, right** oder **full outer join**: äußerer Join

select *
from R_1, R_2
where $R_1.A = R_2.B$;

kann wie folgt geschrieben werden:

select *
from R_1 **join** R_2 **on** $R_1.A = R_2.B$;

Right Outer Join

select *
from pruefen f **right outer join** Studenten s **on**
f.MatrNr=s.MatrNr;

	matrnr integer	vorlnr integer	persnr integer	note numeric(2,1)	matrnr integer	name character varying(30)	semester integer
1					24002	Xenokrates	18
2	25403	5041	2125	2.0	25403	Jonas	12
3					26120	Fichte	10
4					26830	Aristoxenos	8
5	27550	4630	2137	2.0	27550	Schopenhauer	6
6	28106	5001	2126	1.0	28106	Carnap	3
7					29120	Theophrastos	2
8					29555	Feuerbach	2
9					42	mueller	

Dreiwege Right Outer Join

```
select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr,
s.MatrNr, s.Name
from Professoren p right outer join
                    (pruefen f right outer join Studenten s on
                    f.MatrNr=s.MatrNr)
                    on p.PersNr=f.PersNr;
```

	persnr integer	name character varying(30)	persnr integer	note numeric(2,1)	matrnr integer	matrnr integer	name character varying(30)
1						24002	Xenokrates
2	2125	Sokrates	2125	2.0	25403	25403	Jonas
3						26120	Fichte
4						26830	Aristoxenos
5	2137	Kant	2137	2.0	27550	27550	Schopenhauer
6	2126	Russel	2126	1.0	28106	28106	Carnap
7						29120	Theophrastos
8						29555	Feuerbach
9						42	mueller

Dreiwege Left Outer Join

```
select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr,
s.MatrNr, s.Name
from Professoren p left outer join
     (pruefen f left outer join Studenten s on f.MatrNr=
     s.MatrNr)
     on p.PersNr=f.PersNr;
```

	persnr integer	name character varying(30)	persnr integer	note numeric(2,1)	matrnr integer	matrnr integer	name character varying(30)
1	2125	Sokrates	2125	2.0	25403	25403	Jonas
2	2126	Russel	2126	1.0	28106	28106	Carnap
3	2127	Kopernikus					
4	2133	Popper					
5	2134	Augustinus					
6	2136	Curie					
7	2137	Kant	2137	2.0	27550	27550	Schopenhauer

Full Outer Join

select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr, s.MatrNr, s.Name
from Professoren p **full outer join**
(pruefen f **full outer join** Studenten s **on**
f.MatrNr= s.MatrNr)
on p.PersNr=f.PersNr;

	persnr integer	name character varying(30)	persnr integer	note numeric(2,1)	matrnr integer	matrnr integer	name character varying(30)
1	2125	Sokrates	2125	2.0	25403	25403	Jonas
2	2126	Russel	2126	1.0	28106	28106	Carnap
3	2127	Kopernikus					
4	2133	Popper					
5	2134	Augustinus					
6	2136	Curie					
7	2137	Kant	2137	2.0	27550	27550	Schopenhauer
8						26830	Aristoxenos
9						29120	Theophrastos
10						42	mueller
11						29555	Feuerbach
12						24002	Xenokrates
13						26120	Fichte