Informationssysteme

Sommersemester 2016

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

Ausblick auf kommende Vorlesungen

- **Weiterführende SQL Konzepte**
  - Fensteranfragen
  - Rekursion in SQL
- **Anwendungsprogrammierung mit JDBC**
- **PL/pgSQL**
  - Sprache/Konzept um prozeduralen Code zu schreiben und direkt in der Datenbank (nicht im Client mit Java/JDBC) auszuführen.
- **Trigger**
  - Beschreibe welche Anweisungen ausgeführt werden sollen, wenn bestimmte Bedingung erfüllt ist. Z.B. füge Tupel in Relation Bestellungen ein, wenn ein Produkt im Lager weniger als 10 Mal vorrätig ist.

Zur Erinnerung: Die relationale Uni-DB

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

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

Vorlesungen			
VorlNr	Titel	SWS	gelesen von
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

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
25403	5022

Assistenten			
PerslNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2126

prüfen			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

Weiterführende SQL Konzepte

Ranking, Partitionierung und Fensteranfragen

- SQL:2003 Standard
- Ermöglicht das Gruppieren/Partitionieren von Tupeln der Ergebnismenge.
- Und Anwendung einer Aggregat-Funktion auf diese Partitionen
- Z.B. **select** .... , count(*) **over** (**partition by** MatrNr) as vlcount **from** ...

Rekursion

- Beispiel: Gegeben eine Relation `setztVoraus`, in der für eine Vorlesung jeweils die direkten Vorgänger (welche vorausgesetzt werden) angegeben sind
- Wie können dann rekursiv (beliebig tief) Vorlesungen gefunden werden, auf denen eine Vorlesung aufbaut?

```
select s.name, count(*) over
(partition by s.matrnr) as vlcount,
h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

	name character varying(30)	vlcount bigint	vorlnr integer
1	Carnap	4	5259
2	Carnap	4	5216
3	Carnap	4	5052
4	Carnap	4	5041
5	Feuerbach	2	5001
6	Feuerbach	2	5022
7	Fichte	1	5001
8	Jonas	1	5022
9	Schopenhauer	2	5001
10	Schopenhauer	2	4052
11	Theophrastos	3	5049
12	Theophrastos	3	5041
13	Theophrastos	3	5001

Im Vergleich dazu:

```
select s.name, count(*) as vlcount,
h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
group by s.name, h.vorlnr
order by s.name asc;
```

	name character varying(30)	vlcount bigint	vorlnr integer
1	Carnap	1	5041
2	Carnap	1	5052
3	Carnap	1	5216
4	Carnap	1	5259
5	Feuerbach	1	5001
6	Feuerbach	1	5022
7	Fichte	1	5001
8	Jonas	1	5022
9	Schopenhauer	1	4052
10	Schopenhauer	1	5001
11	Theophrastos	1	5001
12	Theophrastos	1	5041
13	Theophrastos	1	5049

Im Vergleich dazu:

```
select s.name, count(*) as vlcount
from hoeren h, studenten s
where h.matrnr=s.matrnr
group by s.name
order by s.name asc;
```

	name character varying(30)	vlcount bigint
1	Carnap	4
2	Feuerbach	2
3	Fichte	1
4	Jonas	1
5	Schopenhauer	2
6	Theophrastos	3

.... mit mehreren Partitionen

```
select s.name, count(*) over (partition by s.matrnr) as vlcount,
count(*) over (partition by h.vorlnr) as studentcount, h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

- **vlcount:** Anzahl Tupel mit demselben Namen
- **studentcount:** Anzahl Tupel mit derselben VorlNr

	name character varying(30)	vlcount bigint	studentcount bigint	vorlnr integer
1	Carnap	4	1	5052
2	Carnap	4	2	5041
3	Carnap	4	1	5216
4	Carnap	4	1	5259
5	Feuerbach	2	2	5022
6	Feuerbach	2	4	5001
7	Fichte	1	4	5001
8	Jonas	1	2	5022
9	Schopenhauer	2	4	5001
10	Schopenhauer	2	1	4052
11	Theophrastos	3	1	5049
12	Theophrastos	3	2	5041
13	Theophrastos	3	4	5001

rank()

```
select s.name, count(*) over (partition by s.matrnr) as vlcount,
rank() over (partition by s.name order by h.vorlnr) as rank, h.vorlnr
from hoeren h, studenten s
where h.matrnr=s.matrnr
order by s.name asc;
```

- **rank:** Position in Gruppe wenn partitioniert nach s.name

	name character varying(30)	vlcount bigint	rank bigint	vorlnr integer
1	Carnap	4	1	5041
2	Carnap	4	2	5052
3	Carnap	4	3	5216
4	Carnap	4	4	5259
5	Feuerbach	2	1	5001
6	Feuerbach	2	2	5022
7	Fichte	1	1	5001
8	Jonas	1	1	5022
9	Schopenhauer	2	1	4052
10	Schopenhauer	2	2	5001
11	Theophrastos	3	1	5001
12	Theophrastos	3	2	5041
13	Theophrastos	3	3	5049

rank() vs. dense_rank()

Berechne den Rang von Studenten basierend auf GPA.

select ID, **rank() over (order by GPA desc)** as student_rank
from student_grades
order by student_rank;

- Was passiert, wenn zwei Studenten den gleichen GPA haben?
- Z.B. wenn bei Studenten den höchsten GPA haben bekommen beide Rang 1
- die nächsten Studenten bekommen dann Rang 3, etc.
- Wenn es drei Studenten geben würde mit dem zweithöchsten GPA, dann würde der nächste zu vergebene Rang 6 sein.
- Bei Funktion **dense_rank** gibt es keine fehlenden Ränge, d.h. die Tupel mit dem zweithöchsten Wert würden Rang 2 bekommen, etc.

Beispiel ohne rank() berechnet

Die Anfrage von zuvor kann auch wie folgt ausgedrückt werden:

```
select ID, (1+(select count(*)
               from student_grades B
               where B.GPA > A.GPA)) as student_rank
from student_grades A
order by student_rank;
```

- **Die naive Ausführung dieser Anfrage hat quadratischen Aufwand: Für jeden Studenten wird der Rang berechnet durch linearen Aufwand** (laufen über alle anderen Studenten).

- **In der vorhergehenden Notation mit rank() kann das Datenbanksystem geschickter vorgehen**: Relation sortieren und den Rang einfach berechnen.

window frames

select h.matrnr, count(h.matrnr) over ()
from hören h;

- keine Partitionierung
- alle Werte gleich

	matrnr integer	count bigint
1	26120	13
2	27550	13
3	27550	13
4	28106	13
5	28106	13
6	28106	13
7	28106	13
8	29120	13
9	29120	13
10	29120	13
11	29555	13
12	25403	13
13	29555	13

select h.matrnr, count(h.matrnr) **over (order by** h.matrnr)
from hoeren h;

- keine Partitionierung
- ABER: **laufendes Fenster**
- Fenster enthält
  - alle Werte bis hierhin
  - einschließlich derjenigen mit **derselben** matrnr

	matrnr integer	count bigint
1	25403	1
2	26120	2
3	27550	4
4	27550	4
5	28106	8
6	28106	8
7	28106	8
8	28106	8
9	29120	11
10	29120	11
11	29120	11
12	29555	13
13	29555	13

Regeln für Fensterfunktion

- **Nur im SELECT und ORDER BY erlaubt**
- Nicht in **group by**, **having** und **where**
- Warum? Ausführung der Fensterfunktionen passiert logisch nach diesen Statements
- **kann mit normaler Aggregation kombiniert werden:**

```
select h.matrnr, count(*) as countstar, count(*) over (order by h.matrnr)
as countpartition
from hoeren h
group by h.matrnr;
```

- **countpartition:** laufendes Fenster über Ergebnis der Gruppierung!

	matrnr integer	countstar bigint	countpartition bigint
1	25403	1	1
2	26120	1	2
3	27550	2	3
4	28106	4	4
5	29120	3	5
6	29555	2	6

select h.matrnr, count(*) as countstar, count(*) over () as countpartition
from hoeren h
group by h.matrnr;

- keine Partitionierung!
- kein laufendes Fenster!

Auswertungsreihenfolge:

- erst **where**
- dann **group by**
- dann **having**
- dann **Fenster**

Angenommen wir haben eine Tabelle tot_credits, in der die Summe aller von Studenten erreichten Credit Points stehen, pro Jahr, also ein Eintrag pro Jahr.

```
select year, avg(num_credits)
             over (order by year rows 3 preceding)
             as avg_total_credits
from tot_credits;
```

- Diese Anfrage berechnet die durchschnittliche Anzahl von Credits über je 3 Tupel, in der durch order by angegebenen Reihenfolge.
- Wir haben also z.B. für das Jahr 2009 einen Durchschnitt, der über die Jahre 2007, 2008 und 2009 berechnet wurde.
- Ähnlich mit Angaben von following:

```
select year, avg(num_credits)
             over (order by year rows
                 between 3 preceding and  2  following)
             as avg_total_credits
from tot_credits;
```

Beispiel

Wetter	
Zeit	**Temperatur**
1	15
2	16
3	17
4	17
5	16
6	18
7	20
8	21
9	22
10	21
11	20
12	23

```
select zeit,
avg(temperatur) over
(order by zeit rows
between 1 preceding
and 1 following) as
avg_temp from wetter
order by zeit
```

Zeit	**avg_temp**
1	15.5
2	16
3	16.66
4	16.66
5	17
6	18
...	...

Beispiel (2)

Mit der Aggregationsfunktion array_agg können wir zur Veranschaulichung schauen welche Werte dieses laufende Fenster umfasst:

select zeit, array_agg(temperatur) **over** (**order by** zeit **rows between 1 preceding and 1 following**) **as** info from wetter **order by** zeit

Zeit	**info**
1	{15,16}
2	{15,16,17}
3	{16,17,17}
4	{17,17,16}
5	{17,16,18}
6	{16,18,20}
...	...

Bereichsbasierte Fenster-Spezifikation

- Die Fenster zuvor waren spezifiziert durch Anzahl der Tupel (z.B. 3 preceeding and 2 following)
- Nun: Fenster-Spezifikation berücksichtigt Bereich (Range) von Daten

```
select year, avg(num_credits)
             over (order by year range between year-4 and year)
             as avg_total_credits
from tot_credits;
```

Bereichsbasierte Fenster-Spezifikation

- Angenommen wir haben nun eine Relation tot_credits_depts(dept_name, year, num_credits)
- Also wie zuvor bloß per Department
- Dann können wir nun noch anhand von Department partitionieren:

```
select dept_name,year, avg(num_credits)
                    over (partition by dept_name
                    order by year range between year-4 and year)
                    as avg_total_credits
from tot_credits_dept;
```

Hinweis: Einige Beispiele sind zum Teil direkt übernommen aus dem Buch "Datebase System Concepts" von Silberschatz, Korth, Sudarshan.

Übersicht Syntax (in Postgresql)

http://www.postgresql.org/docs/9.3/static/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

```
function_name ([expression [, expression ... ]]) OVER window_name
function_name ([expression [, expression ... ]]) OVER ( window_definition )
function_name ( * ) OVER window_name
function_name ( * ) OVER ( window_definition )
```

where window_definition has the syntax

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS
                                                { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

and the optional frame_clause can be one of

```
{ RANGE | ROWS } frame_start
{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

where frame_start and frame_end can be one of

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

Voraussetzen	
vorgaenger	**nachfolger**
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

Vorlesungen	
vorlnr	**titel**
5001	Grundzuege
5041	Ethik
5043	Erkenntnistheorie
5049	Maeeutik
4052	Logik
5052	Wissenschaftstheorie
5216	Bioethik
5259	Der Wiener Kreis
5022	Glaube und Wissen
4630	Die 3 Kritiken

Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?

Wie kann dies in SQL berechnet werden?

Rekursion

Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?

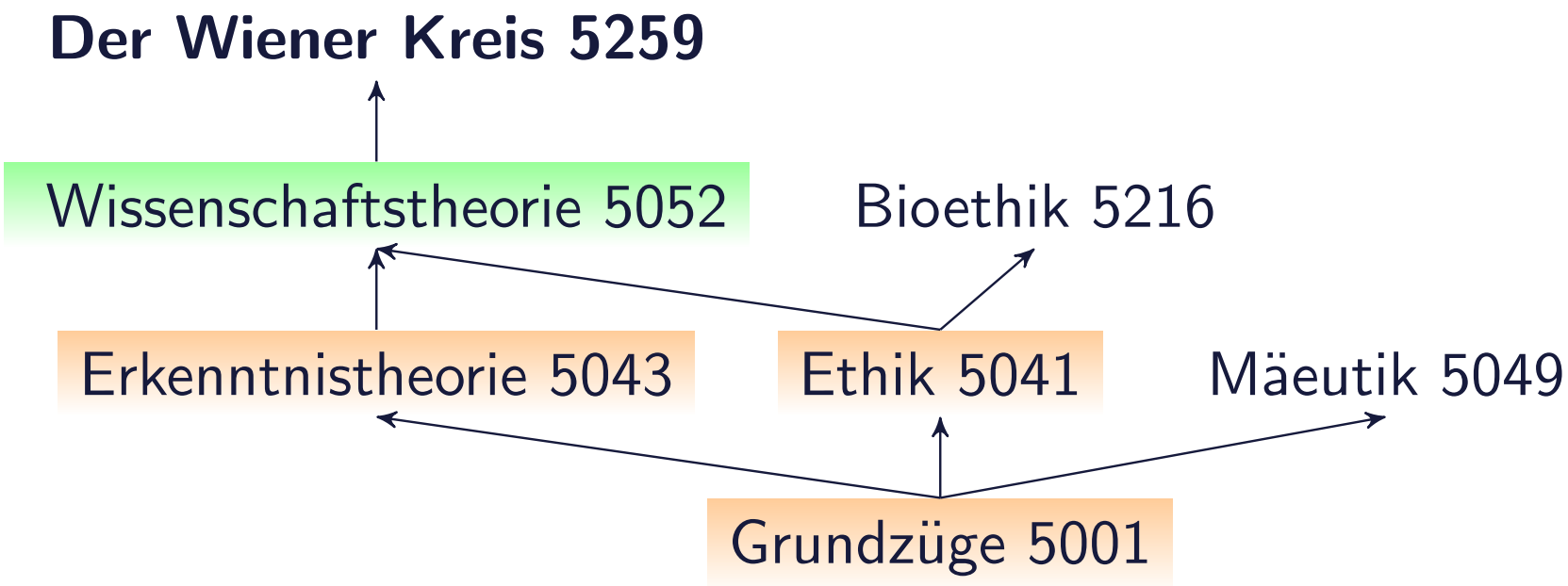
```
select Vorgaenger
from voraussetzen, Vorlesungen
where Nachfolger=VorlNr and
      Titel = 'Der Wiener Kreis';
```

Rekursion

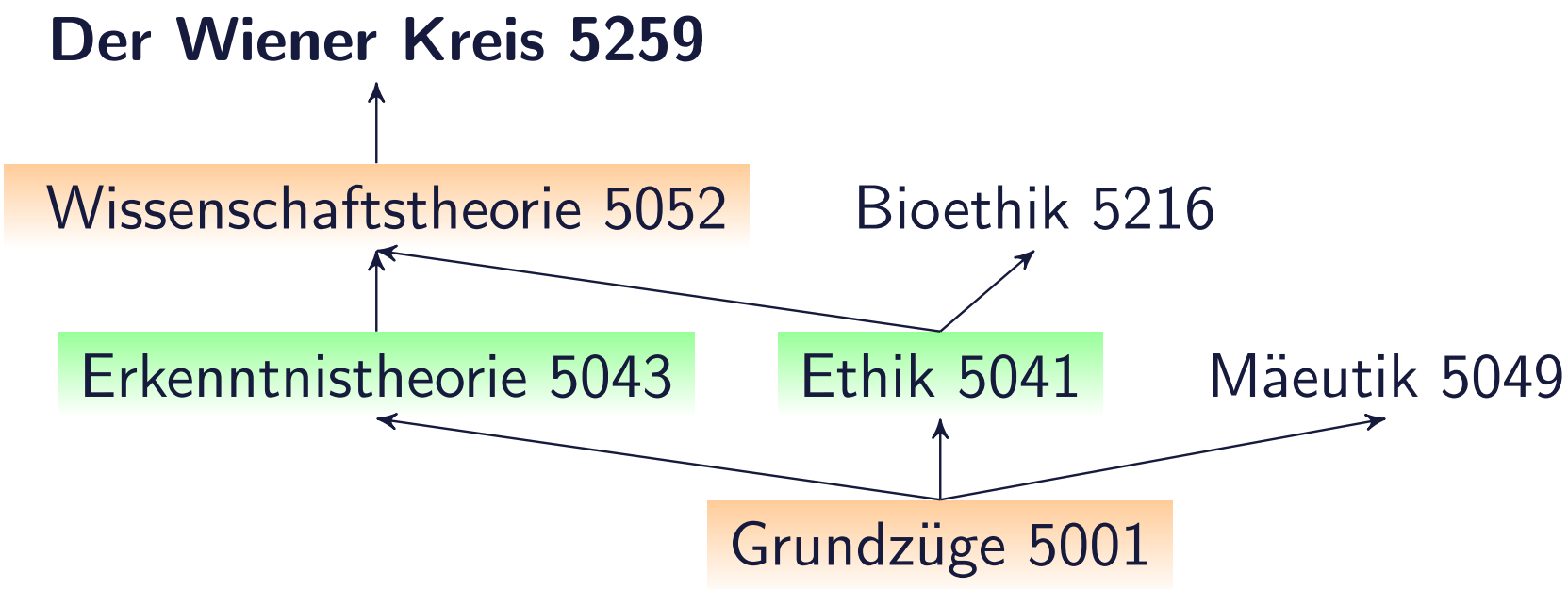
Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?

```
select Vorgaenger
from voraussetzen, Vorlesungen
where Nachfolger=VorlNr and
      Titel = 'Der Wiener Kreis';
```

Hier werden allerdings nur die direkten Vorgänger berechnet. Ergebnis: Vorlesung mit VorlNr=5052.


Der Wiener Kreis 5259
Wissenschaftstheorie 5052
Bioethik 5216
Erkenntnistheorie 5043
Ethik 5041
Mäeutik 5049
Grundzüge 5001

select v1.Vorgaenger
from voraussetzen v1, voraussetzen v2, Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger **and**
v2.Nachfolger= v.VorlNr **and**
v.Titel='Der Wiener Kreis';



Vorgänger der Tiefe n

```
select v1.Vorgaenger
from voraussetzen v1
        ...
        voraussetzen vn_minus_1
        voraussetzen vn,
        Vorlesungen v
where v1.Nachfolger= v2.Vorgaenger and
        ...
        vn_minus_1.Nachfolger= vn.Vorgaenger and
        vn.Nachfolger = v.VorlNr and
        v.Titel='Der Wiener Kreis';
```

Wie kann man alle Vorgänger finden?

Tiefe 1 union Tiefe 2 union Tiefe 3 union ...

Transitive Hülle

Was hier sehr hilfreich wäre: **Berechnung der transitiven Hülle**.

$$trans_{A,B}(R) = \{(a,b)|\exists k \in \mathbb{N}(\exists \tau_1, \tau_2, ...\tau_k \in R($$

$$\tau_1.A = \tau_2.B \wedge$$

$$\tau_2.A = \tau_3.B \wedge$$

$$...$$

$$\tau_{k-1}.A = \tau_k.B \wedge$$

$$\tau_1.A = a \wedge$$

$$\tau_k.B = b))\}$$

Enthält alle Paare, für die ein "Pfad" beliebiger Länge k in R existiert.

Values Statement

- Erzeugt konstante Tabelle
- **values** (1, 'eins'), (2, 'zwei'), (3, 'drei');

  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);
```

Vergleiche hierzu:

```
with mytable(mycolumn) as(
    values (1)
)
select mycolumn+1
from mytable;
```

Rekursionen in SQL

Definieren eine rekursive "View", basierend auf Union (all) von Zwei Anfragen: Einer nicht-rekursiven Anfrage und einer rekursiven.

```
with recursive mytable(mycolumn) as (
        values(1)                          nicht rekursiver Teil
    union all                              union all
        select mycolumn+1
        from mytable                       rekursiver Teil:
        where mycolumn < 100               nur dieser darf mytable referenzieren
)
select sum(mycolumn)                       eigentlicher Aufruf
from mytable;
```

Ergebnis: 5050

Achtung: Darauf achten, dass die Rekursion terminiert!

Auswertung von rekursiven Anfragen

Schritt 1

- Evaluiere den nicht-rekursiven Teil. Für UNION (nicht aber für UNION ALL), entferne Duplikate. Alle verbliebenen Tupel werden in Ergebnis hinzugefügt und auch in eine temporäre Tabelle kopiert.

Schritt 2: Solange temporäre Tabelle nicht leer ist wiederhole:

(a) Evaluiere den rekursiven Teil, wobei die Eigenreferenz durch die temporäre Tabelle ersetzt wird. Für UNION (aber nicht UNION ALL), entferne Duplikate und auch Duplikate zu vorherigen Ergebnissen. Füge verbliebene Tupel in Ergebnis hinzu und ebenso in eine temporäre Zwischenergebnis-Tabelle.

(b) Ersetzt Inhalt der temporären Tabelle mit dem Inhalt der Zwischenergebnis-Tabelle, leere die temporäre Zwischenergebnis-Tabelle.

Rekursion: Anfrage für Voraussetzungen

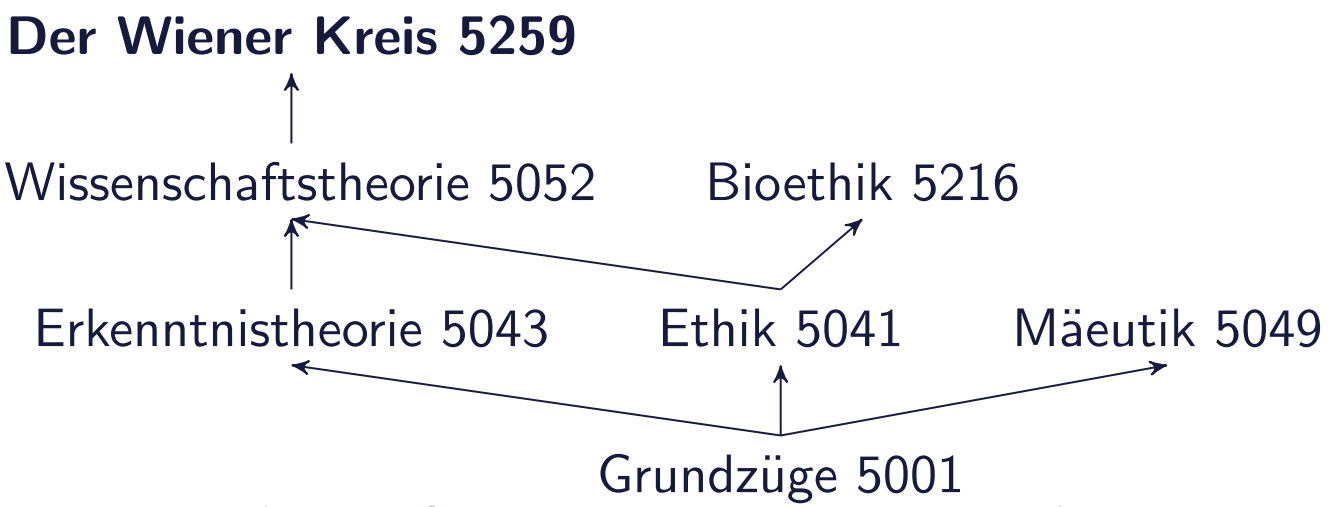
```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select *
from TransitivVorl
order by (vorg,nachf) asc;
```

Rekursion: Anfrage für Voraussetzungen

```
with recursive TransitivVorl (Vorg, Nachf)
as (
    select Vorgaenger, Nachfolger
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select v2.titel
from TransitivVorl tv, vorlesungen v1, vorlesungen v2
where tv.nachf=v1.vorlnr and v1.titel='Der Wiener Kreis'
    and vorg=v2.vorlnr;
```

```
with recursive TransitivVorl (Vorg, Nachf, iteration)
as (
    select Vorgaenger, Nachfolger, 1 as iteration
    from voraussetzen
union all
    select distinct t.Vorg, v.Nachfolger, 1+ t.iteration
    from TransitivVorl t, voraussetzen v
    where t.Nachf=v.Vorgaenger
)
select * from TransitivVorl order by iteration;
```

	vorg integer	nachf integer	iteration integer
1	5001	5041	1
2	5001	5043	1
3	5001	5049	1
4	5041	5216	1
5	5043	5052	1
6	5041	5052	1
7	5052	5259	1
8	5043	5259	2
9	5041	5259	2
10	5001	5052	2
11	5001	5216	2
12	5001	5259	3



Nur zur Info: User-defined Function (UDF), die für eine bestimmte Vorlesung alle Vorgänger berechnet. UDFs betrachten wir später noch.

```
CREATE OR REPLACE FUNCTION alleVorgaenger(id integer)
RETURNS TABLE (VorlNr integer) AS $$
BEGIN
     -- temporaere Tabellen
    CREATE TEMP TABLE IF NOT EXISTS vorgaengerTabelle(VorlNr integer);
    CREATE TEMP TABLE IF NOT EXISTS neu_vorgaengerTabelle(VorlNr integer);
    CREATE TEMP TABLE IF NOT EXISTS temp_Tabelle(VorlNr integer);
    DELETE FROM vorgaengerTabelle;
    DELETE FROM temp_Tabelle;
    DELETE FROM neu_vorgaengerTabelle;

    --los geht es mit Suche der direkten Vorgaenger
    INSERT INTO neu_vorgaengerTabelle (select v.vorgaenger from voraussetzen v where v.nachfolger = id);
    LOOP
        INSERT INTO vorgaengerTabelle (select r.vorlnr from neu_vorgaengerTabelle r);
        INSERT INTO temp_Tabelle   --suche neue Vorgaenger
            (SELECT v.vorgaenger
            FROM neu_vorgaengerTabelle r, voraussetzen v
            WHERE v.nachfolger = r.vorlnr
            ) --aber nur Neue
            EXCEPT (SELECT r.vorlnr FROM vorgaengerTabelle r);
        DELETE from neu_vorgaengerTabelle;
        INSERT into neu_vorgaengerTabelle (select * from temp_Tabelle);
        DELETE FROM temp_Tabelle;
        IF NOT EXISTS (SELECT * FROM neu_vorgaengerTabelle) THEN
            EXIT; --exit aus Schleife, falls keine weiteren Vorgaenger gefunden
        END IF;
    END LOOP;
    RETURN QUERY SELECT  r.vorlnr FROM vorgaengerTabelle r;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM alleVorgaenger(5259)
```

Rekursion in SQL

- Iteration wird so lange ausgeführt bis keine neuen Tupel hinzugefügt werden.
- Also ein Fixpunkt erreicht ist.

Voraussetzungen

- Die rekursive Anfrage muss **monoton** sein, d.h. ausgeführt auf einer Instanz V_1 der rekursiven View muss Ergebnis eine Obermenge von den Ergebnissen auf Instanz V_2 sein, falls V_1 eine Obermenge von V_2 ist.
- D.h. wenn mehr Tupel zur View hinzugefügt werden muss die rekursive Anfrage mindestens die gleichen Tupel wie zuvor liefern.
- z.B. sollte die rekursive Anfrage also kein NOT EXISTS enthalten.