6. Sichten und weiterführende Anfragekonzepte in SQL

Vorlesung "Informationssysteme"

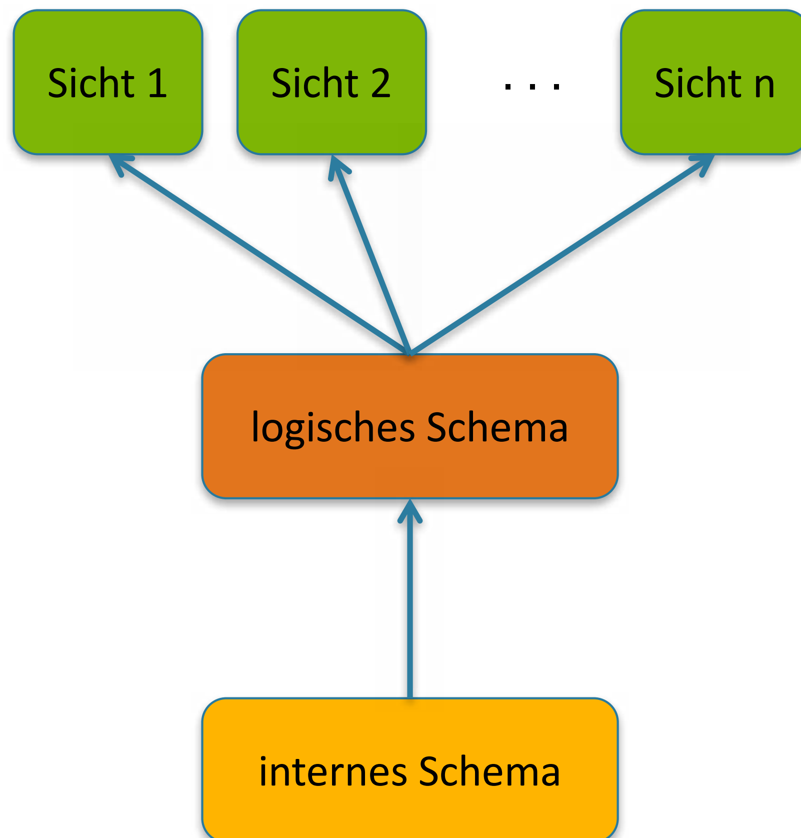
Sommersemester 2017

Technische Universität Kaiserslautern

Überblick

- Sichten (Views)
  - Definition und Eigenschaften
  - Änderbare Sichten
  - "WITH CHECK OPTION"
- WITH-Klausel
- Rekursive Anfragen
- Fensteranfragen

Datenabstraktion



externe Ebene (Sichten)
jedes externe Schema (mit log. DM spezifiziert) beschreibt die Sicht einer Benutzergruppe als Teilmenge des log. Schemas

logische Ebene
DB-Schema (mit log. DM spezifiziert) beschreibt die Struktur der gesamten DB für alle Nutzer

physische Ebene
internes Schema (mit phys. DM beschrieben) legt fest, wie die Daten gespeichert werden

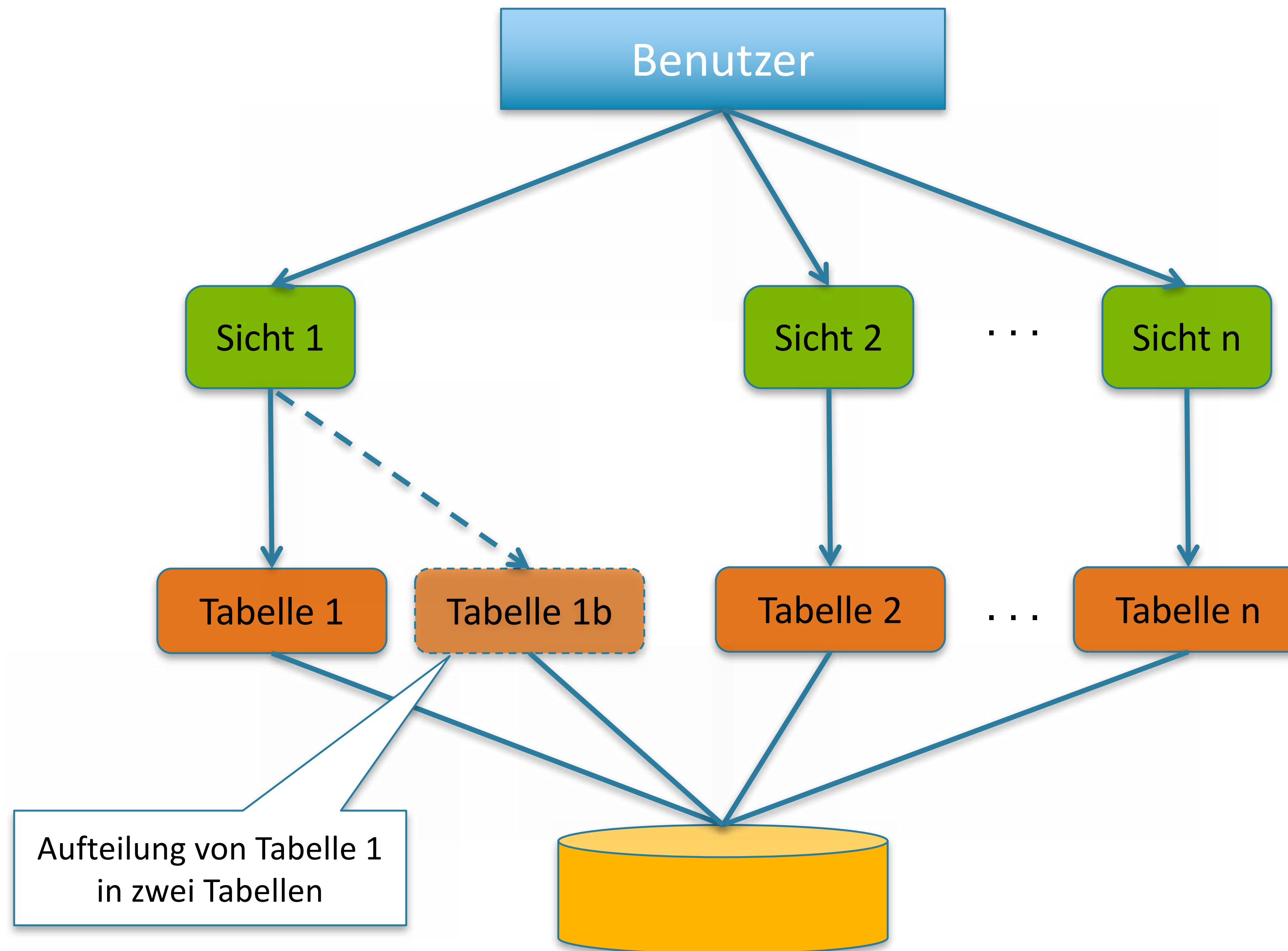
Sichtkonzept

- Sicht (View): mit Namen bezeichnete, aus Tabellen abgeleitete, virtuelle Tabelle (Anfrage)

  ➔ intensionale Relation (s. Kapitel 4)
- Ziel: Festlegung
  - welche Daten Benutzer sehen wollen (Vereinfachung, leichtere Benutzung)
  - welche Daten sie nicht sehen dürfen (Datenschutz)
  - einer zusätzlichen Abbildung (erhöhte Datenunabhängigkeit)
- Korrespondenz zum externen Schema

  Benutzer sieht jedoch i. allg. mehrere Sichten (Views) und Tabellen

Sichten zur Gewährleistung von Datenunabhängigkeit



Sichtdefinition und Beispiele

- Sichtdefinition

```
CREATE VIEW view [(column-commalist)]
    AS table-exp
        [WITH [CASCADED | LOCAL] CHECK OPTION]
```

- D1: Sicht, die alle Programmierer mit einem Gehalt < 30.000 umfasst

```
CREATE VIEW
    ARME_PROGRAMMIERER AS
SELECT PNR, NAME, BERUF, GEHALT, ANR
FROM   PERS
WHERE BERUF= ‘Programmierer’ AND GEHALT < 30 000
```

- D2: Sicht für den Datenschutz

```
CREATE VIEW STATISTIK(BERUF, D_GEHALT) AS
SELECT BERUF, AVG(GEHALT)
FROM PERS
GROUP BY BERUF
```

Eigenschaften von Sichten

- Sicht kann wie eine Tabelle behandelt werden

  z.B.: **SELECT** * **FROM** STATISTIK **WHERE** BERUF='Programmierer'

- Sichtsemantik: „dynamisches Fenster“ auf zugrunde liegende Tabellen
  - Sicht ist eine virtuelle Tabelle, wird immer beim Ausführen von Anfragen auf der Sicht "dynamisch" berechnet
  - Änderungen in den zugrundeliegenden Basistabellen werden automatisch in der Sicht "sichtbar"
- Sichten auf Sichten sind möglich
- Eingeschränkte Änderungen: aktualisierbare und nicht-aktualisierbare Sichten
  - Änderungen auf Sicht werden auf Änderungen der zugrundeliegenden Basistabellen abgebildet
  - "ARME_PROGRAMMIERER" ist änderbar, "STATISTIK" nicht!

Semantik von Sichten – "dynamisches Fenster"

Sicht **ARME_PROGRAMMIERER**

Tabelle **PERS**

GEBDAT	PNR	NAME	BERUF	GEHALT	ANR	W_ORT
12/07/87	1536	Müller	Buchhalter	25000	K21	KL
21/02/76	2864	Schmidt	Programmierer	24000	K45	MA
02/11/59	9746	Maier	Programmierer	28000	K55	LU
14/01/65	7468	Haas	Programmierer	29000	K53	KL
06/05/70	4572	Bauer	Programmierer	35000	K64	F

- Welche Auswirkungen auf die Inhalte der Sicht hat ...
  - **DELETE FROM** PERS **WHERE** PNR=7468
  - **INSERT INTO** PERS **VALUES**
    (01/01/69, 9876, 'Zöller', 'Programmierer', 23000, K22, 'HD')
  - **UPDATE** PERS **SET** GEHALT= GEHALT + 3000
    WHERE BERUF='Programmierer'

Abbildung von Sicht-Anfragen

- Anfragen auf Sicht werden in äquivalente Operationen auf Tabellen umgesetzt (Sicht-Expansion)
  - Anfrage (Sichtreferenz):
    SELECT NAME, GEHALT
    FROM ARME_PROGRAMMIERER
    WHERE ANR = 'K55'
  - Realisierung durch Anfragemodifikation:
    SELECT AP.NAME, AP.GEHALT
    FROM (**SELECT** PNR, NAME, BERUF, GEHALT, ANR
    FROM PERS
    WHERE BERUF= 'Programmierer' **AND** GEHALT < 30000) AP
    WHERE AP.ANR = 'K55'
  - Äquivalent zu:
    SELECT NAME, GEHALT
    FROM PERS
    WHERE ANR='K55' **AND**
    BERUF='Programmierer' **AND** GEHALT< 30000
- Abbildungsprozess auch über mehrere Stufen durchführbar

Änderbare Sichten

Tabelle **PERS**

Sicht **ARME_PROGRAMMIERER**

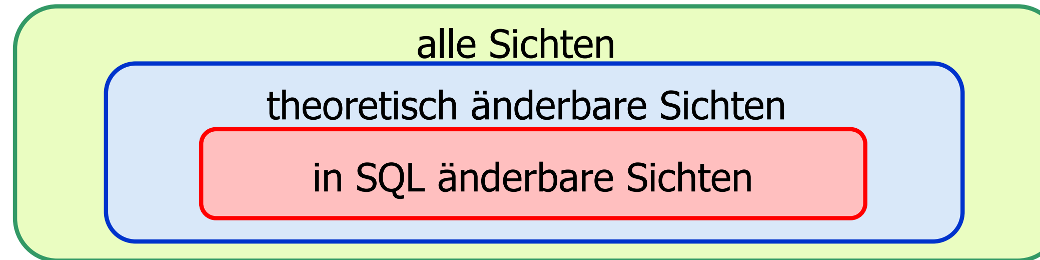
GEBDAT	PNR	NAME	BERUF	GEHALT	ANR	W_ORT
12/07/87	1536	Müller	Buchhalter	25000	K21	KL
21/02/76	2864	Schmidt	Programmierer	24000	K45	MA
02/11/59	9746	Maier	Programmierer	28000	K55	LU
14/01/65	7468	Haas	Programmierer	29000	K53	KL
06/05/70	4572	Bauer	Programmierer	35000	K64	F

- Änderung muss auf zugrundeliegender Tabelle durchgeführt werden!
- Welche Auswirkungen auf die Inhalte der PERS-Tabelle hat ...
  - **DELETE FROM** ARME_PROGRAMMIERER **WHERE** PNR=7468
  - **INSERT INTO** ARME_PROGRAMMIERER **VALUES**
    (9876, 'Zöller', 'Programmierer', 23000, K22)
  - **UPDATE** ARME_PROGRAMMIERER
    SET GEHALT= GEHALT + 3000 **WHERE** GEHALT < 25000

Nicht änderbare Sichten - Beispiele

- Problematisch sind Sichten, für die eine Änderung eines Sicht-Tupels nicht eindeutig auf ein einziges Basistabellen-Tupel abgebildet werden kann
- Beispiele:
  - **CREATE VIEW** STATISTIK(BERUF, D_GEHALT) **AS**
    SELECT BERUF, **AVG**(GEHALT) **FROM** PERS **GROUP BY** BERUF
  - **CREATE VIEW** BERUFE(BERUF, ANR) **AS**
    SELECT DISTINCT BERUF, ANR **FROM** PERS
  - **CREATE VIEW** PERSPROJ **AS**
    SELECT P.*, J.*
    FROM PERS P, PROJEKTMITARBEIT PM, PROJEKT J
    WHERE P.PNR=PM.PNR **AND** PM.JNR=J.JNR
  - i.A. problematisch (warum?):
    CREATE VIEW BERUFE(BERUF, ANR) **AS**
    SELECT BERUF, ANR **FROM** PERS

Änderbare Sichten in SQL



- Einfache, aber restriktive Regeln in SQL-92: Sicht ist änderbar wenn
  - weder DISTINCT noch Aggregatfunktionen, GROUP BY, HAVING verwendet wird,
  - die SELECT-Liste nur eindeutige Attributnamen enthält,
  - die FROM-Klausel nur eine Tabelle oder änderbare Sicht referenziert,
  - und die in der FROM-Klausel referenzierte Tabelle nicht zusätzlich in einer geschachtelten Anfrage erneut in deren FROM-Klausel referenziert wird.
- SQL-99 ist deutlich weniger restriktiv!
- Außerdem zu beachten
  - INSERT in die Sicht belegt alle nicht in der Sicht enthaltenen Attribute der zugrundeliegenden Tabelle mit DEFAULT (bzw. NULL)
  - führt zu Laufzeitfehlern bei Attributen mit NOT NULL, ohne DEFAULT (→ Primärschlüssel, Schlüsselkandidaten)

Sind alle möglichen Änderungen auch erwünscht?

Tabelle **PERS**

Sicht **ARME_PROGRAMMIERER**

GEBDAT	PNR	NAME	BERUF	GEHALT	ANR	W_ORT
12/07/87	1536	Müller	Buchhalter	25000	K21	KL
21/02/76	2864	Schmidt	Programmierer	24000	K45	MA
02/11/59	9746	Maier	Programmierer	28000	K55	LU
14/01/65	7468	Haas	Programmierer	29000	K53	KL
06/05/70	4572	Bauer	Programmierer	35000	K64	F

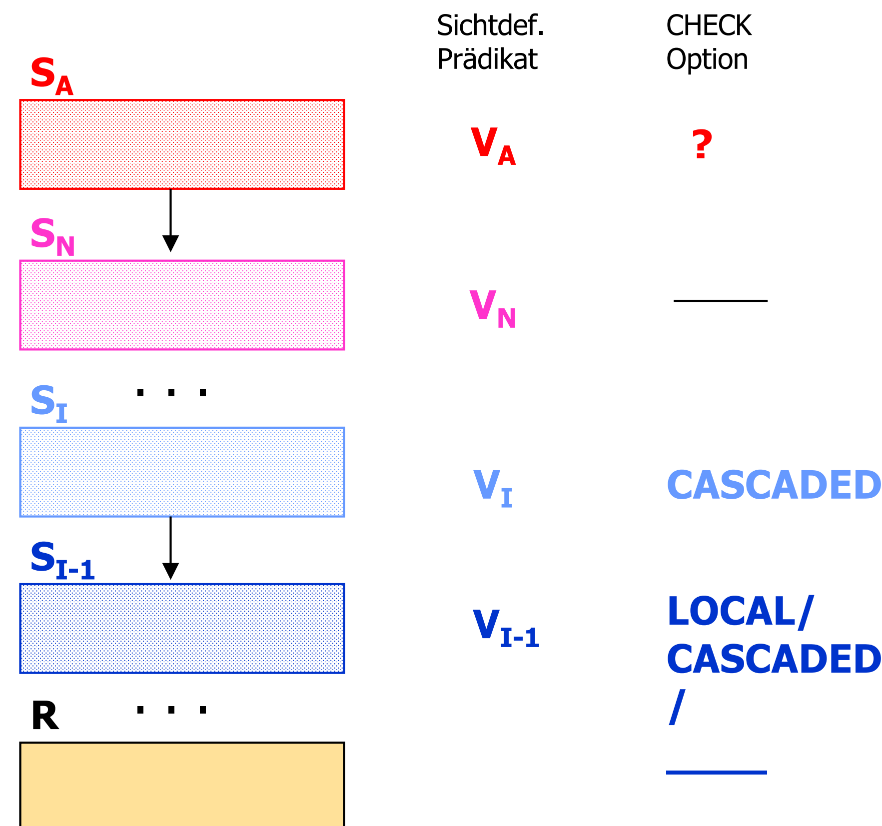
- Sind folgende Änderungen erwünscht/zulässig?
  - **INSERT INTO** ARME_PROGRAMMIERER **VALUES**
    (9876, 'Zöller', 'Programmierer', 32000, K22)
  - **UPDATE** ARME_PROGRAMMIERER
    SET GEHALT= GEHALT + 3000
- Was passiert, wenn neues/geändertes Tupel die Sichtdefinition nicht mehr erfüllt?
  - neues Tupel "verschwindet" aus der Sicht!

Sichtdefinition mit Zusatz WITH CHECK OPTION

- Überprüfung der Sichtdefinition - WITH CHECK OPTION
  - Einfügungen und Änderungen müssen das die Sicht definierende Prädikat erfüllen. Sonst: Zurückweisung
  - nur auf aktualisierbaren Sichten definierbar
- Verfeinerung der CHECK-Option, um Sichten auf Sichten zu kontrollieren
  - WITH LOCAL CHECK OPTION prüft nur die WHERE-Klausel der Sicht, für die die CHECK-Option spezifiziert wurde
  - WITH CASCADED CHECK OPTION (äquivalent zu WITH CHECK OPTION) prüft auch die WHERE-Klauseln aller Sichten die der Definition (evtl. transitiv) zugrundeliegen, unabhängig davon welche Optionen dort definiert wurden

WITH CHECK OPTION - Beispiel

- Annahmen
  - Sicht S_A mit dem die Sicht definierenden Prädikat V_A wird aktualisiert
    - bedeutet implizit auch Aktualisierung von S_N ... S_1, R
  - S_I ist die höchste Sicht im Abstammungspfad von S_A, mit der Option CASCADED
  - Für S_{I+1} – S_N fehlt WITH CHECK OPTION
- Was wird überprüft?
  - keine Check-Option bei S_A
    - Check: $V_I \wedge V_{I-1} \wedge \ldots \wedge V_1$ weil CASCADED bei S_I
    - Tupel können aus S_A verschwinden
  - CASCADED bei S_A
    - Check: $V_A \wedge V_N \wedge \ldots \wedge V_I \wedge \ldots \wedge V_1$
    - keine Tupel können aus S_A verschwinden
  - LOCAL bei S_A
    - Check: $V_A \wedge V_I \wedge \ldots \wedge V_1$
    - Tupel können aus S_A verschwinden, wenn sie auch aus S_N verschwinden
- Empfehlung: immer CASCADED CHECK OPTION verwenden!

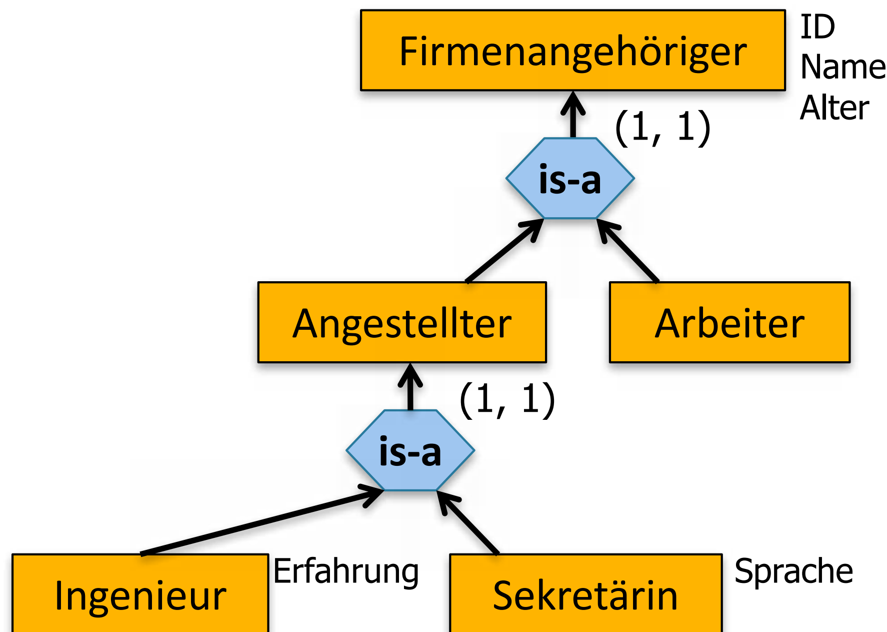


Sichten vs. Materialisierte Sichten

- (Dynamische) Sicht
  - = Makro für Query
  - Ergebnis der Anfrage wird nicht vorberechnet
  - Rechenaufwand zum Zeitpunkt der Anfrage (=query time)
  - erst wenn die Sicht benutzt wird, wird auch das Ergebnis der Sicht berechnet
- Materialisierte Sicht
  - Ergebnis der Sicht wird vorberechnet
  - wenn eine Anfrage die Sicht benutzt, ist das Ergebnis der Sicht bereits vollständig berechnet
  - Rechenaufwand vorher (=index time)
  - Problem bei Updates:
    - Tabellen, die zur Vorberechnung der Sicht benutzt werden, ändern sich
      ➔ Materialisierte Sicht muss (oft) angepasst werden

Generalisierung mit Sichtkonzept

- Simulation einiger Aspekte der Generalisierung durch Einsatz des Sichtkonzeptes




```
CREATE TABLE SEKRETÄRIN
        (ID          INT,
         NAME        CHAR(20),
         ALTER       INT,
         SPRACHE     CHAR(15)
         . . .);
INSERT INTO SEKRETÄRIN
    VALUES (436, 'Daisy', 21, 'Englisch');

CREATE TABLE INGENIEUR
        (ID              INT,
         NAME            CHAR(20),
         ALTER           INT,
         ERFAHRUNG       CHAR(15)
         . . .);
INSERT INTO INGENIEUR
    VALUES (123, 'Donald', 37, 'SUN');
```

```
CREATE VIEW ANGESTELLTER
        AS    (SELECT   ID, NAME, ALTER
               FROM     SEKRETÄRIN)
               UNION
              (SELECT   ID, NAME, ALTER
               FROM     INGENIEUR);

CREATE VIEW FIRMENANGEHÖRIGER
        AS    SELECT    ID, NAME, ALTER
              FROM      ANGESTELLTER
              UNION
              SELECT    ID, NAME, ALTER
              FROM      ARBEITER;
```

Generalisierung mit Sichtkonzept (2)

- Obertypen als Sicht (beim Hausklassenmodell, siehe Beispiel)
  - Union von Ober- und Untertypen
  - Also: bevorzugt Zugriff auf Untertypen
  - Nur bei Zugriff auf Obertyp muss die View ausgeführt werden
- Untertypen als Sicht (bei Partitionierungsmodell)
  - Also Join von Untertyp und Obertyp
  - nur bei Zugriff auf Professoren oder Assistenten muss View ausgeführt werden
  - D.h. Zugriff auf Obertyp ist bevorzugt (günstig)
- Alle Typen als Sicht (bei Hierarchierelation)
  - Nur Projektion bei Wurzeltyp
  - Projektion und Selektion (auf Typindikator) bei allen anderen
  - Keine Joins/Unions notwendig
  - Alle Sichten sind änderbar

WITH-Klausel - Motivation

- "Gib mir Infos zu Vorlesungen mit überdurchschnittlich vielen Hörern"

```
SELECT v.VorlNr, v.Titel, h.AnzHörer
FROM  Vorlesungen v,
      (SELECT VorlNr, COUNT(*) AS AnzHörer
       FROM hören
       GROUP BY VorlNr) AS h
WHERE v.VorlNr = h.VorlNr AND h.AnzHörer >
      (SELECT AVG(AnzHörer)
       FROM (SELECT VorlNr, COUNT(*) AS AnzHörer
             FROM hören
             GROUP BY VorlNr))
```

WITH-Klausel

- Ermöglicht Definition von temporären Sichten/Tabellen innerhalb der Anfrage
  - Sicht/Tabelle ist für nachfolgende Definitionen sichtbar!

```
WITH   hörenAnz AS ( SELECT VorlNr, COUNT(*) AS AnzHörer
                     FROM hören
                     GROUP BY VorlNr),
       hörenSchnitt AS (SELECT AVG(AnzHörer) AS Schnitt
                     FROM hörenAnz)
SELECT v.VorlNr, v.Titel, h.AnzHörer
FROM   Vorlesungen v, hörenAnz h
WHERE  v.VorlNr = h.VorlNr AND h.AnzHörer >
       (SELECT Schnitt FROM hörenSchnitt)
```

Rekursion

- Welche Vorlesungen müssen besucht werden, um die Vorlesung "Der Wiener Kreis" verstehen zu können?
- Wie kann dies in SQL berechnet werden?

voraussetzen	
Vorgaenger	**Nachfolger**
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

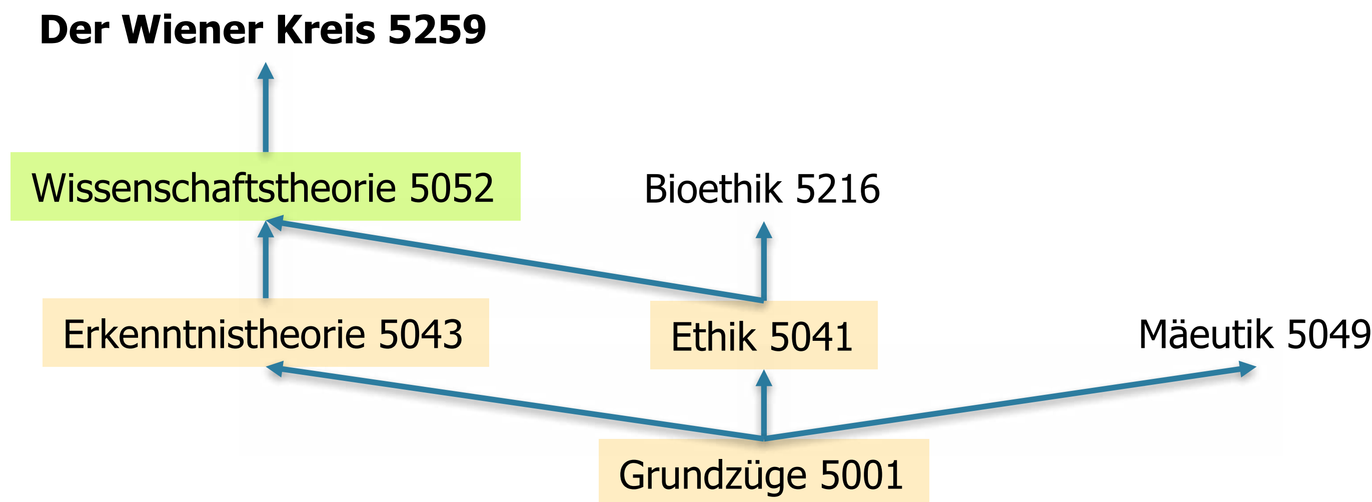
Vorlesungen	
VorlNr	**Titel**
5001	Grundzuege
5041	Ethik
5043	Erkenntnistheorie
5049	Maeeutik
4025	Logik
5052	Wissenschaftstheorie
5216	Bioethik
5259	Der Wiener Kreis
…	…

Vorgänger ...

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

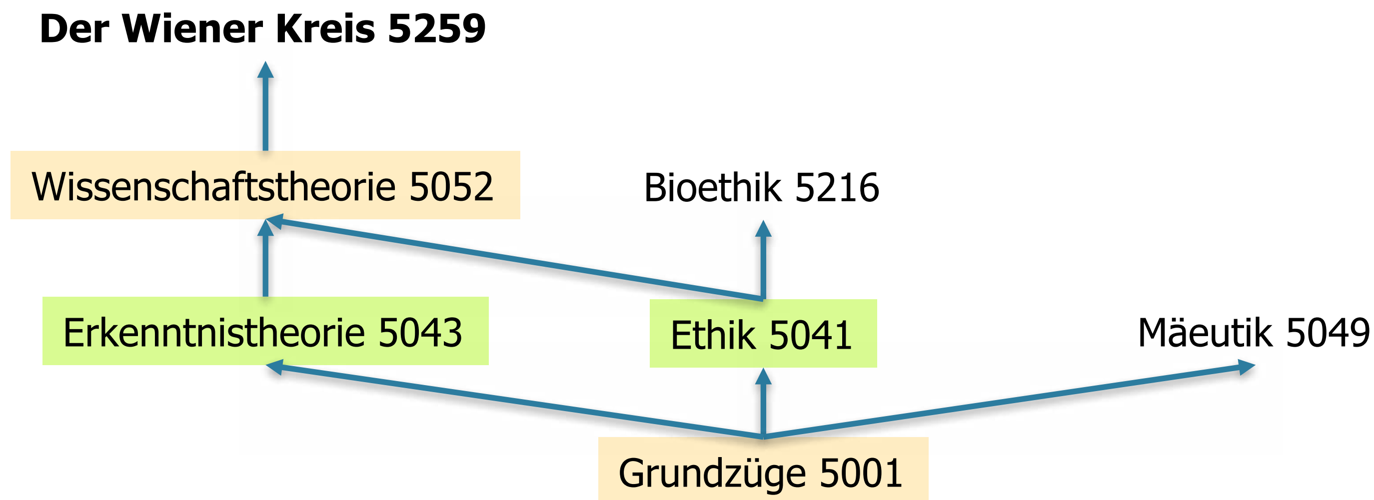
```
SELECT Vorgaenger
FROM voraussetzen, Vorlesungen
WHERE Nachfolger=VorlNr AND Titel = 'Der Wiener Kreis';
```

Hier werden aber nur die direkten Vorgänger berechnet!



… der Tiefe 2 …

```
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';
```



… der Tiefe n

```
SELECT v1.Vorgaenger
FROM  voraussetzen v1,
      voraussetzen v2,
      ...
      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

- Erwünscht: Berechnung der transitiven Hülle

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

$$\begin{gathered} \tau_1.A = \tau_2.B \land \\ \tau_2.A = \tau_3.B \land \\ \dots \\ \tau_{k-1}.A = \tau_k.B \land \\ \tau_1.A = a \land \\ \tau_k.B = b))\} \end{gathered}$$

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

Rekursion in SQL

- WITH RECURSIVE …
  - Definition einer rekursiven, temporären Sicht
  - Basierend auf UNION (ALL) von zwei Anfragen – einer nicht-rekursiven und einer rekursiven

```
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;
```

Auswertung von rekursiven Anfragen

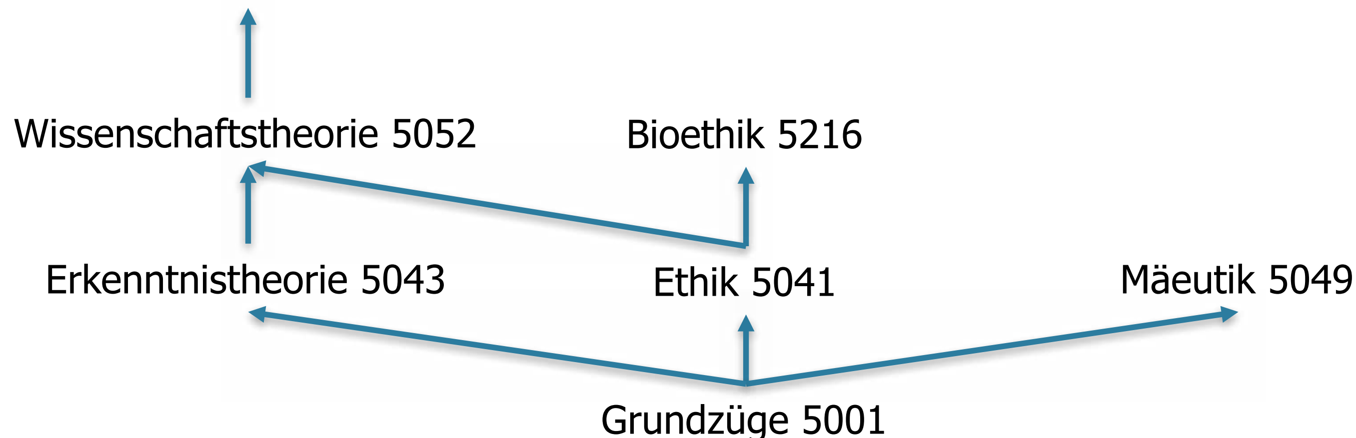
- Schritt 1
  - Evaluiere den nicht-rekursiven Teil. Für UNION (nicht aber für UNION ALL), entferne Duplikate. Alle verbliebenen Tupel werden in Ergebnis hinzugefügt und auch in eine temporäre Tabelle kopiert.
- Schritt 2: Solange temporäre Tabelle nicht leer ist wiederhole:
  a) Evaluiere den rekursiven 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) Ersetze den Inhalt der temporären Tabelle mit dem Inhalt der Zwischenergebnis-Tabelle, leere die temporäre Zwischenergebnis-Tabelle.

Rekursion: Voraussetzungen für "Der Wiener Kreis"

```
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;
```

Variante mit Ausgabe der Rekursionstiefe

```
WITH RECURSIVE TransitivVorl (Vorg, Nachf, Iteration)
AS (
  SELECT Vorgaenger, Nachfolger, 1 AS Iteration
  FROM voraussetzen
UNION ALL
  SELECT DISTINCT t.Vorg, v.Nachfolger, t.Iteration + 1
  FROM TransitivVorl t, voraussetzen v
  WHERE t.Nachf=v.Vorgaenger)
SELECT * FROM TransitivVorl ORDER BY Iteration;
```



Vorg	Nachf	Iter
5001	5041	1
5001	5043	1
5001	5049	1
5041	5216	1
5043	5052	1
5041	5052	1
5052	5259	1
5001	5216	2
5001	5052	2
5041	5259	2
5043	5259	2
5001	5259	3

Rekursion in SQL

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

- Voraussetzungen
  - Die rekursive Anfrage muss monoton sein, d.h. ausgeführt auf einer Instanz V_1 der rekursiven Sicht 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.

Achtung: Terminierung der Rekursion ist nicht immer garantiert!

➔ Anfrage ggf. mit Terminierungsbedingungen absichern.

Ranking, Partitionierung und Fensteranfragen

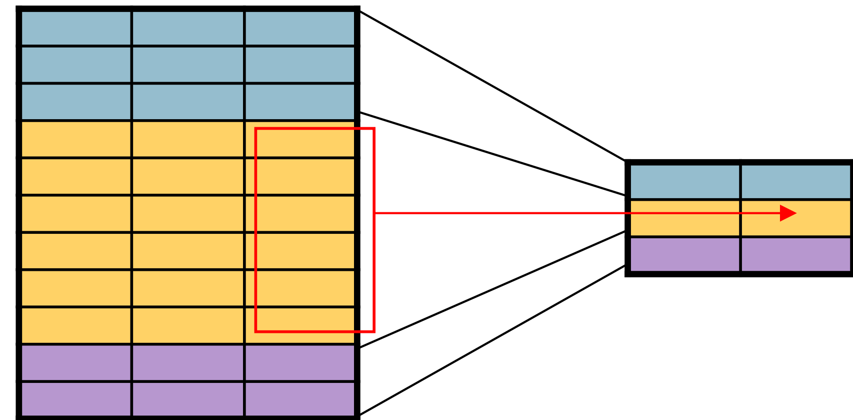
- SQL:2003 Standard
- Ermöglicht das Gruppieren/Partitionieren von Tupeln der Ergebnismenge.
- Und Anwendung einer Aggregat-Funktion auf diese Partitionen
- Z.B. select .... ,

```
        count(*) over (partition by MatrNr) as vlcount
    from ...
```

Konzept im Vergleich zu Gruppierung/Aggregation

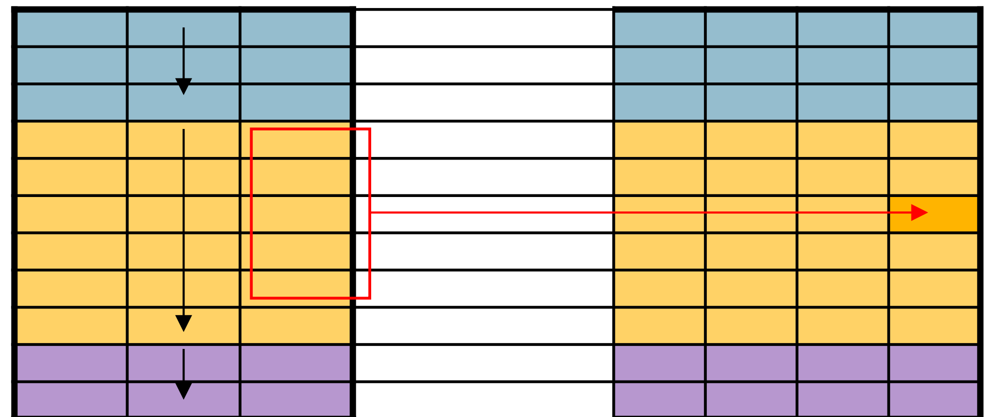
- Gruppierung und Aggregationsfunktion

  SELECT ANR, **AVG**(GEHALT)
  FROM PERS
  GROUP BY ANR



- Fensteranfragen (tupel-basierte Aggregation)

  SELECT ANR, PNR, GEHALT,
  AVG(GEHALT) **OVER**(
  PARTITION BY ANR
  ORDER BY ALTER
  ROWS
  BETWEEN 2 **PRECEEDING**
  AND 2 **FOLLOWING**)
  FROM PERS



Fensteranfragen

```
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

- Für jedes Resultatstupel wird COUNT(*) anhand eines Tupelfensters berechnet.
- Hier ist das Fenster durch eine Partitionierung auf s.matrnr festgelegt
  - Fenster enthält, ähnlich wie bei der Gruppierung, alle Tupel mit der gleichen Matrikelnummer

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

Fensteranfragen (2)

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

Fensteranfragen 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 innerhalb der 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 zwei 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
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 hoeren h;

- keine Partitionierung und Sortierung
- Fenster ist die vollständige Tabelle!
- 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

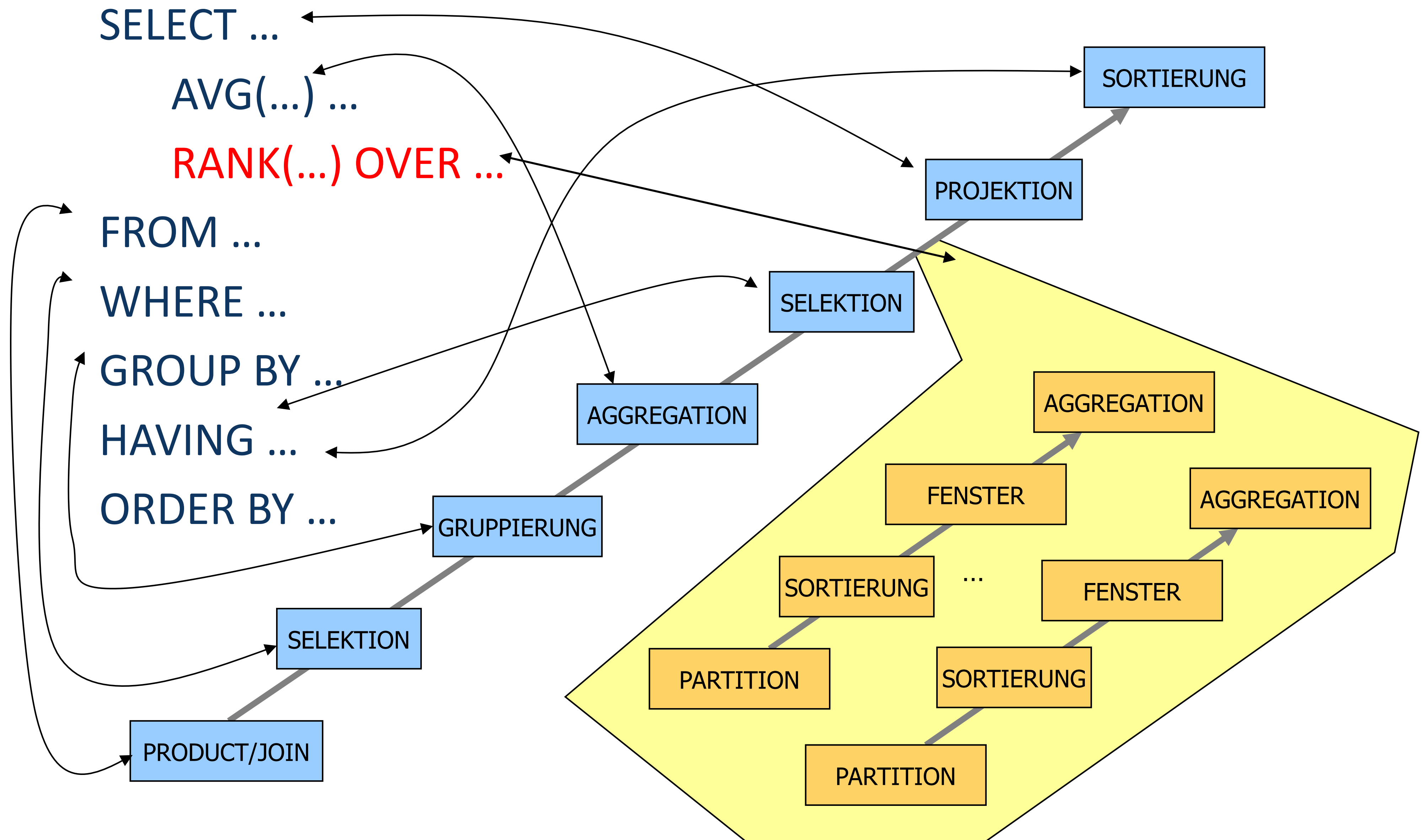
- Nur im SELECT erlaubt
  - Nicht in GROUP BY, HAVING, WHERE
- Warum? Ausführung der Fensterfunktionen passiert logisch nach diesen Statements
- Kann mit normaler Aggregation kombiniert werden:

```
SELECT h.matrnr, COUNT(*) AS countstar,
    COUNT(*) OVER (ORDER BY h.matrnr) AS countpartition
FROM hoeren h
GROUP BY h.matrnr;
```

	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

- countpartition: laufendes Fenster über Ergebnis der Gruppierung!

SQL Auswertungsreihenfolge



Fenster

- **SELECT** func(a) **OVER** (**PARTITION BY** b **ORDER BY** c), ...

  Welche Tupel f sind für ein betrachtetes Tupel t im Fenster?

	mit *PARTITION BY b*	ohne *PARTITION BY b*
mit *ORDER BY c*	$f.b = t.b \wedge f.c \leq t.c$	$f.c \leq t.c$
ohne *ORDER BY c*	$f.b = t.b$	alle

- ORDER BY ermöglicht dynamisches Wachsen des Fensters optional innerhalb einer Partition, und damit sequenzorientierte Analysen
  - Ranking, Kumulation (z.B. laufende Summe)
- Bisher nicht möglich: "gleitendes" Fenster, z.B. für gleitenden Durchschnitt
  - Wird durch benutzerdefinierte *Window Frames* möglich

Benutzerdefinierte Window-Frames

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 2 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 2 PRECEDING AND 2 FOLLOWING)
        AS avg_total_credits
FROM tot_credits;
```

Bereichsbasierte Fenster-Spezikation

- Die Fenster zuvor waren spezifiziert durch Anzahl der Tupel (z.B. 2 preceeding and 2 following)
- Nun: Fenster-Spezikation 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-Spezikation

- Angenommen wir haben nun eine Relation tot_credits_depts(dept_name, year, num_0credits)
- 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;
```

Explizite Fensterdefinition

- Bisher wurden Fenster als Teil der SELECT-Klausel definiert
  - Nachteil: Anwendung mehrerer Funktionen auf dem gleichen Fenster erfordert wiederholte Definition.
- Alternative Synax erlaubt Definition eine Fensters in der WINDOW-Klausel
  - Fenster bekommt einen Namen, der in der OVER-Klausel im SELECT referenziert werden kann.
- Beispiel

```
SELECT dept_name, year,
    AVG (num_credits) OVER FuenfJahre AS avg_total_credits,
    MAX (num_credits) OVER FuenfJahre AS max_total_credits
FROM tot_credits
WINDOW FuenfJahre AS (PARTITION BY dept_name
                ORDER BY year RANGE BETWEEN year - 4 AND year)
```

Zusammenfassung

Sichten/Views

- Einsatz von Sichten in SQL
  - Datenabstraktion: einfache externe Benutzerschemata
  - Datenschutz: "Verbergen" von nicht-zugreifbaren Daten
  - Datenunabhängigkeit: Sicht unabhängig von Datenspeicherung
- Sicht als dynamisches Fenster
  - Sicht ist eine virtuelle Tabelle
  - Berechnung der Sicht zum Anfragezeitpunkt
  - Auswirkungen von Datenänderung in Basistabellen
- Änderungsoperationen auf Sichten
  - Regeln zur Änderbarkeit von Sichten
  - Überprüfung der Sichtprädikate bei Änderungen (WITH CHECK OPTION)

Zusammenfassung (2)

WITH-Klausel

- Ermöglicht Definition von temporären Sichten/Tabellen innerhalb einer Anfrage
- Mehrere Definitionen können aufeinander aufbauen und in der eigentlichen Anfrage beliebig verwendet werden
- Rekursive Anfragen (transitive Hülle)
  - Definition einer rekursiven, temporären Sicht in der WITH-Klausel, basierend auf UNION (ALL) von zwei Anfragen – einer nicht-rekursiven und einer rekursiven.
  - Erlaubt eingeschränkte Klasse von rekursiven Anfragen (monoton, linear).
  - Fixpunktsemantik
  - Vorkehrungen zur Garantie der Terminierung durch Anwender

Zusammenfassung (3)

Fensteranfragen

- Erlauben die Berechnung von neuen Tupelattributen anhand eines "Fensters" (Tupelmenge, die vom aktuellen Tupel abhängt)
  - "OVER"-Klausel als Teil der SELECT-Klausel ermöglicht Partitionierung, Sortierung innerhalb der Partition und präzise Definition des Fensters (basierend auf Bereich oder Tupelzahl)
  - Auswertung von Aggregationsfunktionen und weiteren Funktionen (rank, denserank, …) auf dem Fenster
  - "tupel-basierte Aggregation"
- WINDOW-Klausel ermöglicht wiederholte Verwendung der gleichen (benannten) Fensterdefinition
- Auswertungssemantik: Fensteranfrage wird gemeinsam mit der Projektion (SELECT-Liste) ausgewertet