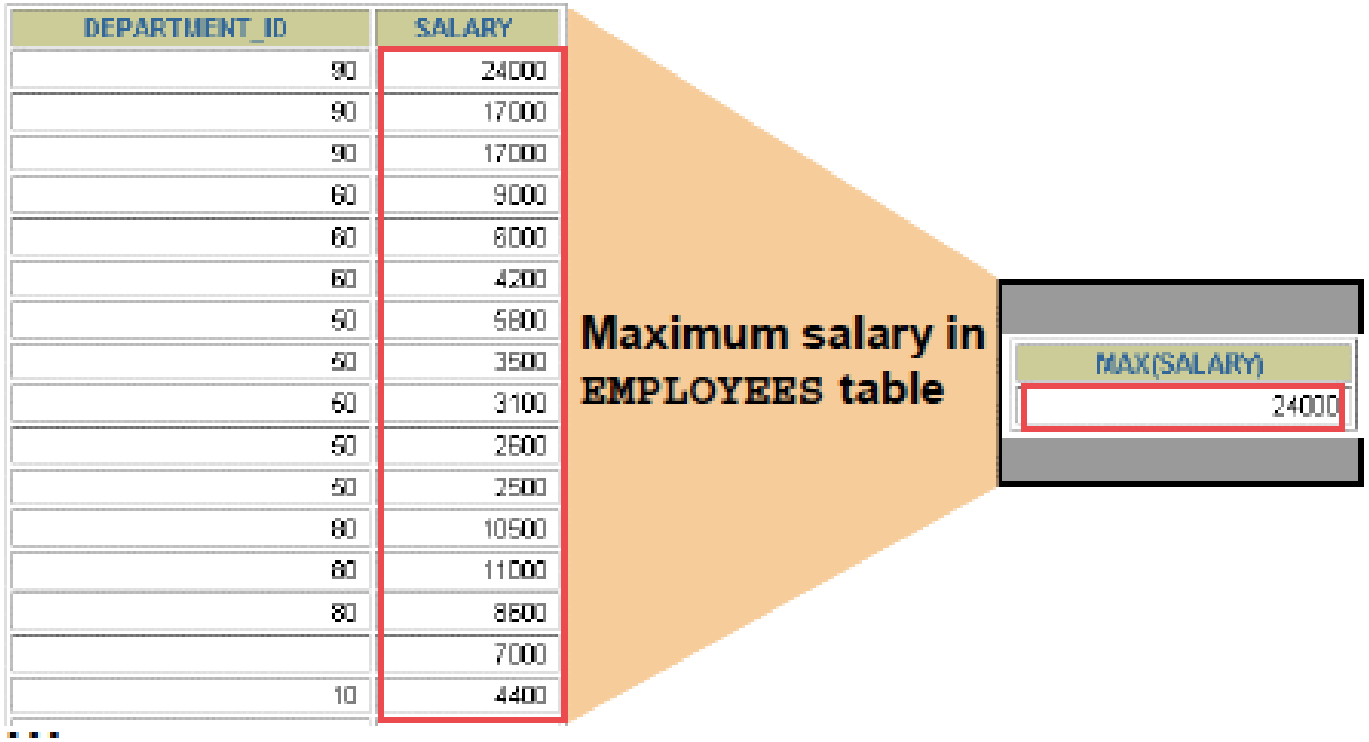
# 3.4 Gruppierungen und Aggregierungen in SQL und in der Relationenalgebra

Aggregierungsfunktionen operieren auf einer Menge von Zeilen und geben je ein Ergebnis pro Gruppe aus; hier für die Gruppe aller Zeilen:

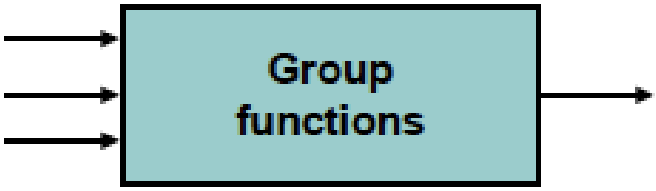


## Syntax von Aggregierungen

**select** *Aggregierungsfunktion*({*Spalte*|*Spaltenausdruck*}), ...
**from** *Tabelle(n)*
[**where** *Bedingung*]
[**order by** ...];

## Arten von Aggregierungsfunktionen (Funktionen auf Zeilengruppen)

- **avg**: Durchschnitt
- **count**: Anzahl
- **max**: Maximum
- **min**: Minimum
- **stddev**: Standardabweichung
- **sum**: Summe
- **variance**: Varianz



## Verwendung von Aggregierungsfunktionen

**avg** und **sum** können für numerische Daten genutzt werden.

```
select  avg(salary), max(salary), min(salary), sum(salary)
from    EMPLOYEES
where job_id like '%REP%';
```

| AVG(SALARY) | MAX(SALARY) | MIN(SALARY) | SUM(SALARY) |
|---|---|---|---|
| 8150 | 11000 | 6000 | 32600 |

**min** und **max** können für numerische Daten, Strings und Datumswerte genutzt werden.

```
select  min(hire_date), max(hire_date)
from    EMPLOYEES;
```

| MIN(HIRE_ | MAX(HIRE_ |
|---|---|
| 17-JUN-87 | 29-JAN-00 |

## Verwendung von Aggregierungsfunktionen (Forts.)

**count**(*) gibt die Anzahl der Zeilen einer Tabelle zurück:

```
select  count(*)
from    EMPLOYEES
where department_id = 50;
```

| COUNT(*) |
|---|
| 5 |

**count**(*expr*) gibt die Anzahl der Zeilen zurück, die für *expr* einen Wert ungleich **null** besitzen:

```
select  count(commission_pct)
from    EMPLOYEES
where department_id = 80;
```

| COUNT(COMMISSION_PCT) |
|---|
| 3 |

## Verwendung von distinct in Aggregierungsfunktionen

**count**(**distinct** *expr*) gibt die Anzahl der unterschiedlichen Werte für *expr* zurück, die ungleich **null** sind.

Beispiel: Um die Anzahl der unterschiedlichen Abteilungen in der Tabelle EMPLOYEES auszugeben, schreibt man:

```
select  count(distinct department_id)
from    EMPLOYEES;
```

| COUNT(DISTINCTDEPARTMENT_ID) |
|---|
| 7 |

## Behandlung von Nullwerten durch Aggregierungsfunktionen

Aggregierungsfunktionen ignorieren Nullwerte in der Spalte:

```
select  avg(commission_pct)
from    EMPLOYEES;
```

| AVG(COMMISSION_PCT) |
|---|
| .2125 |

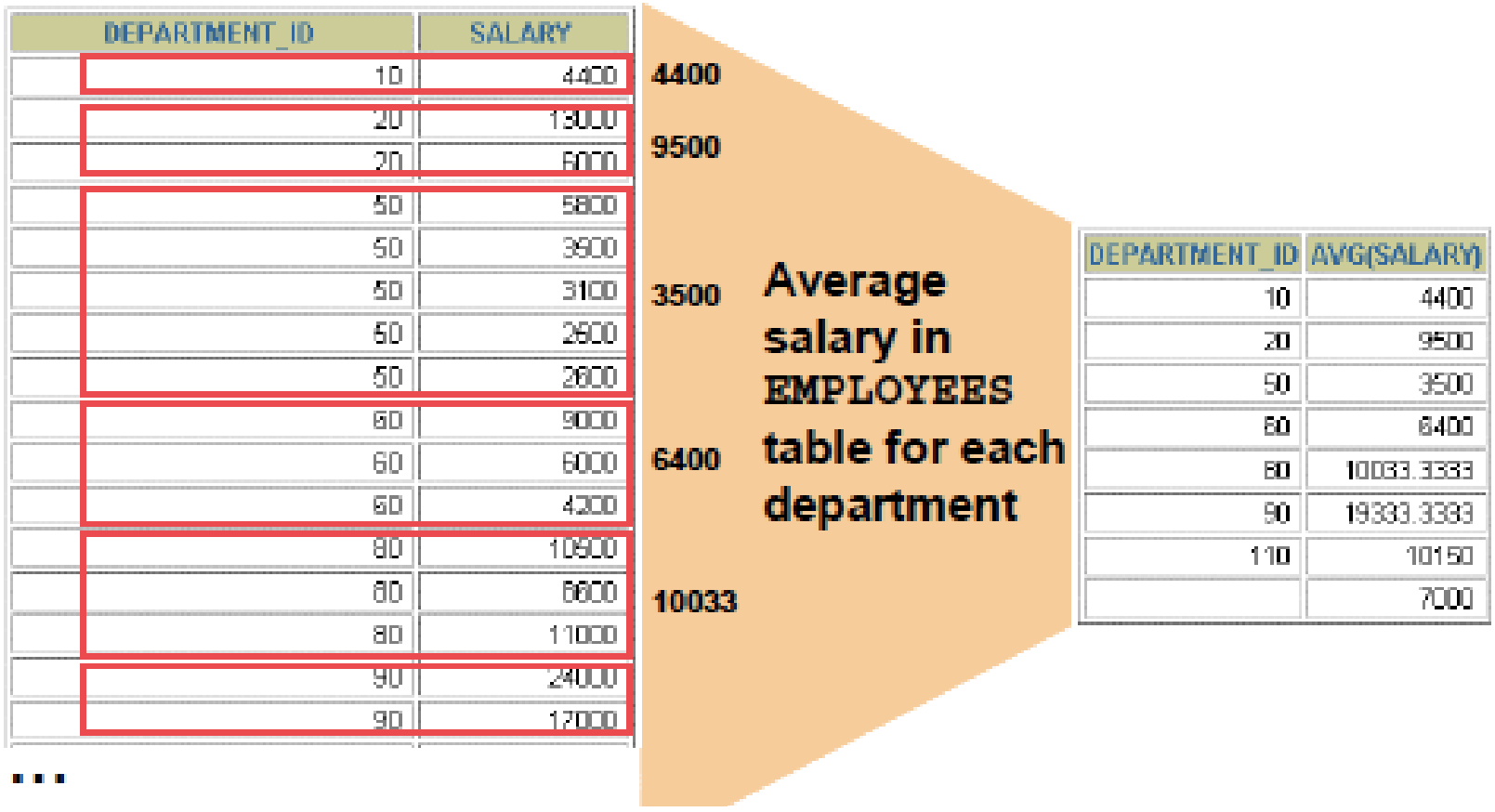
Durch Vorschalten der **nvl**-Funktion lassen sich auch Nullwerte einbeziehen:

```
select  avg(nvl(commission_pct, 0))
from    EMPLOYEES;
```

| AVG(NVL(COMMISSION_PCT,0)) |
|---|
| .0425 |

## Gruppieren

Man kann die Zeilen einer Tabelle in mehrere Gruppen unterteilen, und diese jeweils agggregieren:



## Gruppieren: Syntax von select-Anweisungen mit group-by-Klausel

**select** *Spalte...* , *Aggregierungsfunktion*(...), ...
**from** *Tabelle(n)*
[**where** *Bedingung*]
[**group by** *Gruppenausdruck*, ...]
[**order by** ...];

- Ein Gruppenausdruck (Gruppierungskriterium) kann eine Spalte oder ein Spaltenausdruck sein.
- In der **select**-Klausel dürfen dann nur noch
  - in der **group by**-Klausel erwähnte Spalten oder Spaltenausdrücke
  - oder aber Aggregierungsausdrücke über beliebigen Spalten(ausdrücken) stehen,

  die also pro Gruppe einen Wert bestimmen.
- Aggregierungen ohne **group by**-Klausel beziehen sich auf den Spezialfall der (einen) Gruppe aller Zeilen.

## Verwendung der group-by-Klausel

```
select department_id, avg(salary) from EMPLOYEES
group by department_id;
```

| DEPARTMENT_ID | AVG(SALARY) |
|---|---|
| 10 | 4400 |
| 20 | 9500 |
| 50 | 3500 |
| 60 | 6400 |
| 80 | 10033.3333 |
| 90 | 19333.3333 |
| 110 | 10150 |
| | 7000 |

```
select avg(salary) [18] from EMPLOYEES
group by department_id;
```

| AVG(SALARY) |
|---|
| 4400 |
| 9500 |
| 3500 |
| 6400 |
| 10033.3333 |
| 19333.3333 |
| 10150 |
| 7000 |

[18] Die **group by**-Spalte muss nicht in der **select**-Liste stehen.

## Verbotene Anfragen

Jede Spalte und jeder Ausdruck der **select**-Liste, der keine Aggregierung ist, muss in der **group-by**-Klausel stehen:

```
select  department_id, count(last_name)
from    EMPLOYEES
group by manager_id;
```

→
```
select  department_id, count(last_name)
        *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

# Gruppieren nach mehr als einer Spalte

| DEPARTMENT_ID | JOB_ID | SALARY |
|---|---|---|
| 90 | AD_PRES | 24000 |
| 90 | AD_VP | 17000 |
| 90 | AD_VP | 17000 |
| 60 | IT_PROG | 9000 |
| 60 | IT_PROG | 6000 |
| 60 | IT_PROG | 4200 |
| 50 | ST_MAN | 5800 |
| 50 | ST_CLERK | 3500 |
| 50 | ST_CLERK | 3100 |
| 50 | ST_CLERK | 2600 |
| 50 | ST_CLERK | 2500 |
| 80 | SA_MAN | 10500 |
| 80 | SA_REP | 11000 |
| 80 | SA_REP | 8600 |
| ... | | |
| 20 | MK_REP | 6000 |
| 110 | AC_MGR | 12000 |
| 110 | AC_ACCOUNT | 8300 |

**Add the salaries in the EMPLOYEES table for each department and job.**

| DEPARTMENT_ID | JOB_ID | SUM(SALARY) |
|---|---|---|
| 10 | AD_ASST | 4400 |
| 20 | MK_MAN | 13000 |
| 20 | MK_REP | 6000 |
| 50 | ST_CLERK | 11700 |
| 50 | ST_MAN | 5800 |
| 60 | IT_PROG | 19200 |
| 80 | SA_MAN | 10500 |
| 80 | SA_REP | 19600 |
| 90 | AD_PRES | 24000 |
| 90 | AD_VP | 34000 |
| 110 | AC_ACCOUNT | 8300 |
| 110 | AC_MGR | 12000 |
| | SA_REP | 7000 |

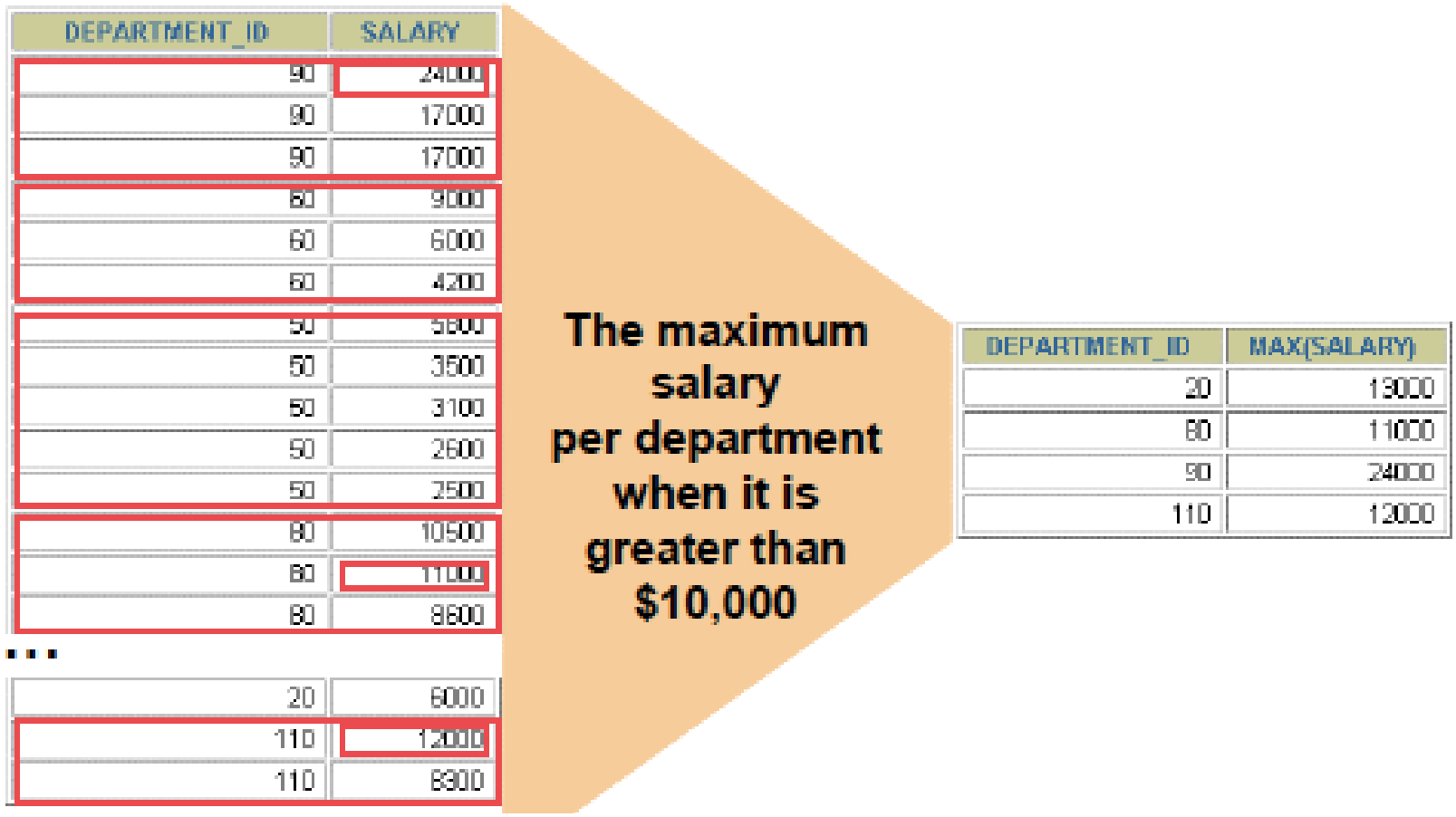
13 rows selected.

## Verwendung der group-by-Klausel mit mehreren Spalten

```
select  department_id dept_id, job_id, sum(salary)
from    EMPLOYEES
group by department_id, job_id;
```

| DEPT_ID | JOB_ID | SUM(SALARY) |
|---|---|---|
| 10 | AD_ASST | 4400 |
| 20 | MK_MAN | 13000 |
| 20 | MK_REP | 6000 |
| 50 | ST_CLERK | 11700 |
| 50 | ST_MAN | 5800 |
| 60 | IT_PROG | 19200 |
| 80 | SA_MAN | 10500 |
| 80 | SA_REP | 19600 |
| 90 | AD_PRES | 24000 |
| 90 | AD_VP | 34000 |
| 110 | AC_ACCOUNT | 8300 |
| 110 | AC_MGR | 12000 |
|  | SA_REP | 7000 |

# Auswahl von Gruppen



## Verbotene Anfragen

```
select  department_id, max(salary)
from    EMPLOYEES
where max(salary) > 10000
group  by department_id;
```

→
```
where max(salary) > 10000
      *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

- Man kann die **where**-Klausel nicht nutzen, um Gruppen auszuwählen. Die **where**-Klausel wählt Tupel aus.
- Deshalb darf man die Aggregierungsfunktionen nicht in der **where**-Klausel benutzen.
- Zur Auswahl von Gruppen muss man stattdessen die **having**-Klausel nutzen.

## Verwendung der having-Klausel

```
select  department_id, max(salary)
from    EMPLOYEES
group  by department_id
having      max(salary) > 10000;
```

| DEPARTMENT_ID | MAX(SALARY) |
|---|---|
| 20 | 13000 |
| 80 | 11000 |
| 90 | 24000 |
| 110 | 12000 |

## Verwendung der having-Klausel und der where-Klausel

```
select  job_id, sum(salary) PAYROLL
from    EMPLOYEES
where job_id not like '%REP%'
group  by job_id
having      sum(salary) > 13000
order   by sum(salary);
```

1. Auswahl der Zeilen
2. Gruppieren
3. Auswahl der Gruppen

| JOB_ID | PAYROLL |
|---|---|
| IT_PROG | 19200 |
| AD_PRES | 24000 |
| AD_VP | 34000 |

## Syntax und Auswertung der select-Anweisung mit having-Klausel

**select** *Spalte ...*, *Aggregierungsfunktion(...)*, ...
**from** *Tabelle(n)*
[**where** *Zeilenbedingung*]
[**group by** *Gruppenausdruck*, ...]
[**having** *Gruppenbedingung* ]
[**order by** ...];

Anfragen mit einer **having**-Klausel werden wie folgt ausgewertet:

1. Mit der **where**-Klausel ausgewählte Zeilen werden mit der **group by**-Klausel gruppiert.

Für jede Gruppe:

2. Die Gruppenbedingung in der **having**-Klausel – inkl. darin enthaltener Aggregierungen – wird ausgewertet.
3. Wenn die Gruppe die **having**-Bedingung erfüllt, wird sie ausgewählt.
4. Die Gruppenspalten bzw. Aggregierungsausdrücke der **select**-Klausel werden für die Gruppe ausgewertet und ausgegeben.

## Verschachtelte Aggregierungsfunktionen

Gib das maximale der Durchschnittsgehälter pro Abteilung aus:

```
select  max(avg(salary))
from    EMPLOYEES
group  by department_id;
```

| MAX(AVG(SALARY)) |
|---|
| 19333.3333 |

## Verwendung von Aggregierungen in Unteranfragen[19]

```
select  last_name, job_id, salary
from    EMPLOYEES
where salary = (select  min(salary)
                from    EMPLOYEES);
```

| LAST_NAME | JOB_ID | SALARY |
|---|---|---|
| Vargas | ST_CLERK | 2500 |

```
select  department_id, min(salary)
from    EMPLOYEES
group  by department_id
having     max(salary)  < (select  min(salary)
                           from    EMPLOYEES
                           where department_id = 50);
```

[19] Auch solche Unteranfragen dürfen korreliert sein.

## Zugehörige Erweiterung der Relationenalgebra

- **Gruppierungs- und Aggregierungsoperator** $\Gamma_{\overline{G}\#\overline{F}}(R)$

  $\Gamma$ bildet aus $R$ zunächst eine *Menge von Gruppen* von solchen Tupeln, die jeweils *die gleichen* $\overline{G}$*-Attributwerte* haben; dann wendet $\Gamma$ auf jede Tupelgruppe die Aggregierungsfunktionen $\overline{F}$ an und bildet daraus *jeweils ein Ergebnistupel*.

  $\overline{G}$ ist eine Menge[20] von $R$-Attributtermen für die Gruppierung; $\overline{G}$ kann auch leer sein ($\overline{G} = \emptyset$), so dass nur eine einzige Tupelgruppe gebildet wird.

  $\overline{F}$ ist eine Liste von mit Aliasen versehbaren "Aggregierungstermen", d. h. *gruppeninvarianten* $R$-Attributtermen, insb. aus $\overline{G}$, oder aggregierenden Funktionen wie count, countd[istinct], sum, avg, min, max usw.[21], jeweils auf einen Attributterm über $R$ angewendet[22].

---

[20] ohne Mengenklammmern notiert

[21] soweit für jeweilige Datentypen sinnvoll

[22] count, sum, avg (Summe, Durchschnitt) beziehen sich auf alle Werte des Attributterms in einer Gruppe, berücksichtigen also auch mehrfach vorkommende Werte. countd zählt nur verschiedene Werte !!! (vgl. in SQL: "**count distinct**") — count und countd können aber auch auf eine Kombination von Attributtermen, evtl. auf ganze Tupel "count(∗)" angewendet werden, und so auch alle Tupel einer Gruppe zählen.

## **Erweiterung der Relationenalgebra** (Forts.)

*Beispiel:*

R

| A | B | C | D |
|---|---|---|---|
| 991 | 1 | 2 | 6 |
| 995 | 1 | 2 | 8 |
| 994 | 1 | 4 | 6 |
| 996 | 1 | 4 | 9 |
| 993 | 3 | 7 | 8 |
| 992 | 3 | 7 | 8 |
| 998 | 3 | 7 | 8 |
| 997 | 5 | 11 | 2 |

$\Gamma_{\overline{G}\#\overline{F}}(R)$

$\Gamma_{B,C\#B,C,\,\mathrm{sum}(D)\,\mathrm{Summe}}(R)$

| B | C | Summe |
|---|---|---|
| 1 | 2 | 14 |
| 1 | 4 | 15 |
| 3 | 7 | 24 |
| 5 | 11 | 2 |

*entspricht in SQL:*

**select** B, C, sum(D) Summe **from** R
**group by** B, C

## Semantik von SQL-Anfragen mit Gruppierungen u. Aggregierungen:

Aggregierung über das gesamte Anfrageergebnis

(a) *Anzahl aller Kunden ?*

```
select  count(*) Anzahl
from    KUNDE
```

→ $\Gamma_{\emptyset\,\#\,\text{count}(*)\ \text{Anzahl}}(\text{KUNDE})$

(b) *Anzahl der versch. Lieferanten, die PC2050 oder MacG12 anbieten ?*

```
select  count(distinct LName) Anz
from    ANGEBOT
where Ware = 'PC2050' or Ware = 'MacG12'
```

→ $\Gamma_{\emptyset\,\#\,\text{countd}(\text{LName})\ \text{Anz}}(\sigma_{\text{Ware}='\text{PC2050}'\,\vee\,\text{Ware}='\text{MacG12}'}(\text{ANGEBOT}))$

## Semantik von SQL-Anfragen mit Gruppierungen u. Aggregierungen

(Forts.): Gruppenweise Aggregierung

(c) *Summe der Bestellmengen von Kunden bis Nr. 1000, je Ware ?*

```
select  Ware, sum(Menge) Summe
from    BESTELLUNG
where   KNr <= 1000
group by Ware
```

→ $\Gamma_{\text{Ware}\,\#\,\text{Ware},\ \text{sum(Menge)}\ \text{Summe}}\,(\sigma_{\text{KNr}\leq 1000}(\text{BESTELLUNG}))$

(d) *Gesamtwert der Bestellungen je Kunde ?*

```
select  KNr, sum(Menge * Preis) Wert
from    BESTELLUNG  natural join  ANGEBOT
group by KNr
```

→ $\Gamma_{\text{KNr}\,\#\,\text{KNr},\ \text{sum(Menge}*\text{Preis)}\ \text{Wert}}\,(\text{BESTELLUNG} \bowtie \text{ANGEBOT})$

## Semantik von SQL-Anfragen mit Gruppierungen u. Aggregierungen

(Forts.): Mehrere Aggregierungen, Bedingungen an Gruppen

(e) *Waren von mehr als einem Anbieter (mit Preisspektrum) ?*

```
select  Ware, min(Preis) Min, max(Preis) Max
from    ANGEBOT
group by Ware
having     count(LName) > 1
```

→ $\pi_{\text{Ware,Min,Max}}( \sigma_{\text{C}>1} ($
$\Gamma_{\text{Ware}\,\#\,\text{Ware},\ \text{min(Preis)}\ \text{Min},\ \text{max(Preis)}\ \text{Max},\ \text{count(LName)}\ \text{C}}$ (ANGEBOT) ) )

(f) *Durchschnitt aller Gesamtbestellmengen von Waren ?*

```
select  avg(sum(Menge)) Mittel
from    BESTELLUNG
group by Ware
```

→ $\Gamma_{\varnothing\,\#\,\text{avg(Gesamt)}\ \text{Mittel}}$ ($\Gamma_{\text{Ware}\,\#\ \text{Ware},\ \text{sum(Menge)}\ \text{Gesamt}}$ (BESTELLUNG))

Eine direkte Schachtelung von Aggregierungen über mehr als 2 Stufen ist in SQL offensichtlich nicht vorgesehen.

## (Gedachte) Ausführung von allgemeinen SQL-Anfragen

*Format:*

**select** ...
**from** ...
**where** ⟨Zeilenbedingung⟩
**group by** ⟨Gruppierungsattribute⟩
**having** ⟨Gruppenbedingung⟩
**order by** ...

*Ausführungsreihenfolge (ohne Optimierung):*

1. **from**: Produkt der angegebenen Relationen (Tabellen)
2. **where**: Selektion von Tupeln (Zeilen) gemäß **where**-Bedingung
   (korrel. Unteranfrage: so als ob äußere Anfrage tupelweise ausgewertet wird)
3. **group by**: Einteilung in Gruppen
4. **having**: Selektion von Gruppen gemäß **having**-Bedingung
5. **select**: Projektion einschl. evtl. tupel- bzw. gruppenweisen Berechnungen
   (falls **group by**: nur Ausdrücke über **group by**-Spalten und Aggregierungen)
6. **order by**: Sortierung des Ergebnisses
   (Sortierung nach Ausdrücken über Spalten der **select**-Liste sowie,
   falls nicht **distinct**, nicht **group by**: über weitere Spalten der FROM-Tabellen,
   bzw. falls **group by**: über weitere Gruppenausdrücke)

**Weitere SQL-Anfragen:** Aggregierungen in korrelierten Unteranfragen

(g) Ein einfaches Beispiel: *Für jede Ware: Anbieter mit minimalem Preis ?*

```
select  Ware, LName
from    ANGEBOT A
where   Preis =
        ( select  min(Preis) from ANGEBOT
          where A.Ware = Ware )
```

$\rightarrow \pi_{a.Ware,a.LName}(ANGEBOT\ a \underset{a.Preis=g.MinPr \wedge a.Ware=g.Ware}{\rtimes} (\Gamma_{Ware\#Ware,\ min(Preis)\ MinPr}(ANGEBOT)\ g))$

Die relationenalgebraische Formulierung benötigt einen Teilterm, der für jede Ware den minimalen Preis vorausberechnet. Dieser Teilterm lässt sich auch direkt in die SQL-Anfrage einbauen, und zwar als "Unteranfrage" in der **from**-Klausel:

```
select  A.Ware, A.LName
from    ANGEBOT A,
        (select  Ware, min(Preis) MinPr from ANGEBOT
         group  by Ware) G
where   A.Preis=G.MinPr and A.Ware=G.Ware
```

**Weitere SQL-Anfragen:** Aggregierungen in korrelierten Unteranfragen (Forts.)

(h) *Für jeden Kunden zu jeder Ware: Welche Anbieter sind deutlich billiger (<50%) als die billigsten lokalen Anbieter (<10 km Entfernung)?*
Die Funktion distance berechne die Entfernung von zwei Adressen in km.

```
select  KName, Ware, LName
from    KUNDE K, ANGEBOT A
where   Preis < 0.5 *
          (select  min(Preis) from ANGEBOT A2, LIEFERANT L2
           where A2.LName = L2.LName
           and     A.Ware = A2.Ware
           and     distance(K.KAdr,L2.LAdr)<10)
```

$\rightarrow \pi_{k.KName,\, a.Ware,\, a.LName}$ (KUNDE k $\times$ ANGEBOT a $\overset{a.Preis < 0.5 * g.MinPr \,\wedge\, a.Ware = g.Ware \,\wedge\, k.KAdr = g.KAdr}{\rtimes}$

$(\Gamma_{param1.Ware,\, param2.KAdr \;\#\; param1.Ware\ Ware,\ param2.KAdr\ KAdr,\ min(Preis)\ MinPr}$

$(\sigma_{a2.LName=l2.LName \,\wedge\, param1.Ware=a2.Ware \,\wedge\, distance(param2.KAdr,\, l2.LAdr)<10}$

$((\pi_{Ware}(ANGEBOT)\ param1) \times (\pi_{KAdr}(KUNDE)\ param2)$

$\times (ANGEBOT\ a2) \times (LIEFERANT\ l2))\ g))$

In der Relationenalgebra wird ein Teilterm benötigt, der in Abhängigkeit von allen möglichen "Parametern" (Korrelationen) Ware/KAdr die gewünschte Aggregierung min(Preis) vorausberechnet und für die Semijoin-Verknüpfung bereitstellt.

## Weitere SQL-Anfragen: (Forts. des Beispiels (h))

Die relationenalgebraische Unteranfrage lässt sich wieder direkt in SQL formulieren, und zwar als "Unteranfrage" in der **from**-Klausel:

```
select  distinct K.KName, A.Ware, A.LName
from    KUNDE K, ANGEBOT A,
   (select  PARAM1.Ware Ware, PARAM2.KAdr KAdr, min(Preis) MinPr
    from    ANGEBOT PARAM1, KUNDE PARAM2, ANGEBOT A2, LIEFERANT L2
    where A2.LName = L2.LName
    and     PARAM1.Ware=A2.Ware
    and     distance(PARAM2.KAdr, L2.LAdr) < 10
    group by PARAM1.Ware, PARAM2.KAdr)  G
where A.Preis<0.5*G.MinPr
and     A.Ware=G.Ware and K.KAdr=G.KAdr
```

In **from**-Unteranfragen, die anstelle von Relationennamen stehen, sind allerdings keine Aliase erlaubt, die außerhalb definiert wurden (d. h. Unteranfragen mit Rückbezügen sind hier unzulässig)[23].

[23] **from**-Unteranfragen sollten zwar benutzt werden, um Anfragen ggf. übersichtlicher und verständlicher aufzubauen, sollten aber auch zugunsten klassischer Formulierungsmöglichkeiten wie geschachtelter **where**-Klauseln sparsam benutzt werden, um Anfragen möglichst automatisch optimierbar zu halten.

## **Weitere SQL-Anfragen**: Beispiel für hilfreiche **from**-Unteranfragen

(i) *Geg. seien* KUNDE(KNr,...,Ort,...), BESTELLUNG(...,KNr,...))
*Welcher Anteil von Kunden pro Ort hat nichts bestellt ?*

```
select OHNE.Ort, (OHNE.Anzahl/ALLE.Anzahl*100) Anteil
from   (select count(*) Anzahl, Ort
        from   KUNDE
        group by Ort) ALLE,
       (select count(*) Anzahl, Ort
        from   KUNDE
        where not exists
                (select * from BESTELLUNG
                 where KNr=KUNDE.KNr)
        group by Ort) OHNE
where  OHNE.Ort=ALLE.Ort
```

## Weitere SQL-Anfragen: (Forts. des Beispiels (i))

(j) Nachstehend eine zu (i) äquivalente Anfrage ohne **from**-Unteranfragen. Diese ist nicht nur schwerer zu verstehen *(zum Selbststudium),* sondern wird auch typischerweise viel viel langsamer ausgeführt.

Bei z.B. 10 Orten mit je 1000 Kunden, davon 100 ohne Bestellungen, wird für (j) vor allem eine Verbund/Produkt-Zwischenrelation mit $10 * 100 * 1000 = 10^6$ Tupeln gebildet, während (i) nur mit Relationen bis $10^4$ Tupeln und einem Verbund mit 10 Tupeln (Orte) arbeitet.

```
select  K1.Ort,
        (count(distinct K1.KNr) /
         count(distinct K2.KNr)*100)  Anteil
from    KUNDE K1, KUNDE K2
where K1.Ort=K2.Ort
and     not exists (select * from BESTELLUNG
                    where KNr=K1.KNr)
group by K1.Ort
```