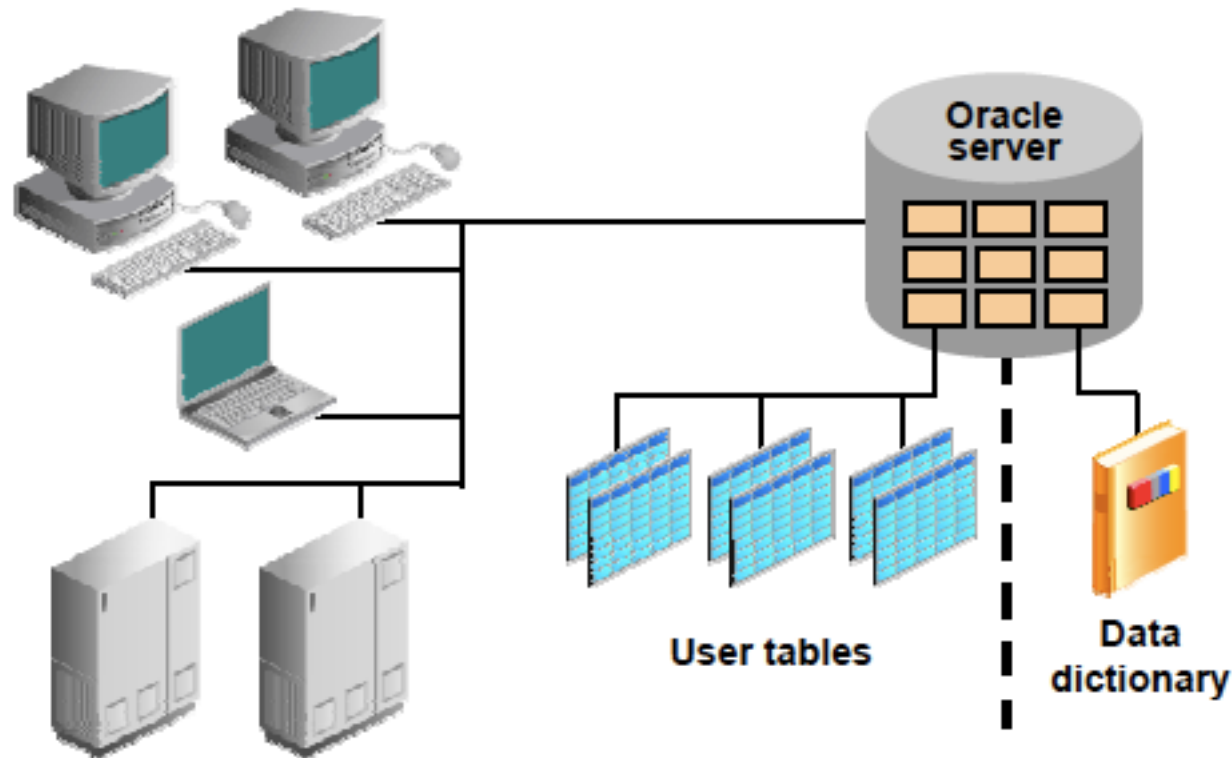


## 3.2 Der Kern der Anfragesprache SQL ("Structured Query Language")

### Verwendete Plattform: Oracle RDBMS

(Relational Database Management System) <sup>2</sup>



<sup>2</sup>Die meisten Folien in den reinen SQL- bzw. PL/SQL-Teilen der Vorlesung sind Übersetzungen und Überarbeitungen von Materialien der Oracle Academic Initiative.

## Kommunikation mit dem RDBMS über SQL

SQL-Anfrage wird eingegeben.

Anfrage wird an den Oracle-Server gesendet.

```
select department_name  
from DEPARTMENTS;
```



DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

8 rows selected.

Ergebnis wird als Tabelle ausgegeben.

## Verwendete Tabellen

### EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	25
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	105
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	110
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	86 ...

...

## Verwendete Tabellen (Forts.)

### DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

### JOB\_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

### 3.2.1 Anfragen an eine einzelne Tabelle mit Hilfe der select-Anweisung

#### Minimale select-Anweisungen

```
select * | [distinct] {{Spalte | Ausdruck} [Alias], ... }  
from  Tabelle;
```

- Die **select**-Klausel benennt die auszugebenden Spalten oder Spaltenberechnungen (Projektion).
- “\*” wählt alle Spalten aus.
- Die **from**-Klausel benennt die Tabelle, die die Spalten enthält.

## Ausgabe aller/spezieller Spalten

```
select *  
from DEPARTMENTS;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

```
select department_id, location_id  
from DEPARTMENTS;
```

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

## Spalten-Aliase und Ausdrücke

```
select last_name as name, commission_pct comm
from EMPLOYEES;
```

NAME	COMM
King	
Kochhar	
De Haan	

...

```
select last_name, salary, salary + 300
from EMPLOYEES
```

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300

...

## Spalten-Aliase und Ausdrücke (Forts.)

```
select last_name "Name", salary*12 "Annual Salary"  
from EMPLOYEES;
```

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

...

```
select last_name || ' is a ' || job_id as "Employee Details"  
from EMPLOYEES;
```

Employee Details
King is a AD_PRES
Kochhar is a AD_VP
De Haan is a AD_VP
Hunold is a IT_PROG
Ernst is a IT_PROG
Lorentz is a IT_PROG
Mourgos is a ST_MAN
Rajs is a ST_CLERK

...

<sup>3</sup>In String-Ausdrücken ist "||" der Konkatenationsoperator.



## Zum Schreiben von SQL-Anweisungen

- SQL-Anweisungen sind im allgemeinen nicht case-sensitiv.
- SQL-Anweisungen können aus einer oder mehreren Zeilen bestehen.
- Schlüsselwörter dürfen nicht abgekürzt oder durch einen Zeilenumbruch aufgetrennt werden.
- Klauseln werden normalerweise in eigene Zeilen geschrieben. Zur besseren Lesbarkeit nutzt man Einrückungen.
- In manchen SQL-Benutzerschnittstellen müssen/können SQL-Anweisungen mit einem Semikolon (;) abgeschlossen werden. Semikolons sind notwendig, wenn man mehrere SQL-Anweisungen ausführt.

## Oracle-SQL-Identifizier (z.B. Tabellennamen, Spaltenalias) ...

- (a) ... bestehen aus 1 bis 30 Zeichen,
- (b) dürfen keinen Namen, der bereits für ein anderes Objekt (Tabelle, Sicht u.a.) desselben Benutzers verwendet wird, duplizieren,
- (c1) beginnen mit einem Buchstaben,
- (c2) enthalten Buchstaben|Ziffern|'\_'|'#'|'\$', aber sind nicht case-sensitiv (intern in Großbuchstaben gespeichert),
- (c3) und dürfen kein reserviertes Wort sein (u.a. aus SQL wie z.B. **select**),
- (d) bzw. sind bei Abweichungen von (c1)–(c3) in Anführungsstriche zu setzen.

## Spalten und Ausdrücke mit Nullwerten

```
select last_name, job_id, salary, commission_pct
from EMPLOYEES;
```

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	
...			
Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2
...			
Gietz	AC_ACCOUNT	8300	

```
select last_name, 12*salary*commission_pct
from EMPLOYEES;
```

LAST_NAME	12*SALARY*COMMISSION_PCT
King	
Kochhar	
...	
Zlotkey	25200
Abel	39600
Taylor	20640
...	
Gietz	

## Zeilenduplikate

In der Standardausgabe von Anfragen werden alle Zeilen angezeigt, inklusive doppelter Zeilen. **distinct** eliminiert Zeilenduplikate.

```
select department_id  
from EMPLOYEES;
```

DEPARTMENT_ID
90
90
90

...

20 rows selected.

```
select distinct department_id  
from EMPLOYEES;
```

DEPARTMENT_ID
10
20
50

...

8 rows selected.

## Übliche select-Anweisungen

```
select  * | [distinct] {{Spalte | Ausdruck} [Alias], ... }  
from    Tabelle  
[where Bedingung ]  
[order by {{Spalte | Ausdruck} [ASC|DESC], ... }];
```

- Durch die **where** - Klausel lässt sich die Ausgabe auf diejenigen Zeilen einschränken, welche die angegebene Bedingung erfüllen (Selektion).
- Durch die **order**-Klausel lässt sich die Ausgabe nach den angegebenen Spalten oder Ausdrücken sortieren.

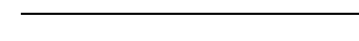
## Verwendung der where-Klausel

### EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

...

*“Gib alle Angestellten aus Abteilung 90 aus”*



EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

```
select employee_id, last_name, job_id, department_id
from   EMPLOYEES
where  department_id = 90;
```

## Wie erstellt man where-Bedingungen?

### Vergleichsoperatoren für einfache Bedingungen

Operator	Bedeutung
=	gleich
>	größer als
>=	größer/gleich
<	kleiner als
<=	kleiner/gleich
<>	ungleich
[not] between ... and ...	[nicht] zwischen zwei Werten (inklusive untere/obere Grenze)
[not] in (Werteliste)	kommt in Liste [nicht] vor
[not] like String-Muster	passt [nicht] auf das Muster
is [not] null	ist [k]ein Nullwert

## Verwendung von Vergleichsoperatoren

```
select last_name, salary
from EMPLOYEES
where salary <= 3000;
```

LAST_NAME	SALARY
Matos	2600
Vargas	2500

```
select last_name, salary
from EMPLOYEES
where salary between 2500 and 3500;
```

LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

## Verwendung von Vergleichsoperatoren (Forts.)

```
select employee_id, last_name, salary, manager_id  
from EMPLOYEES  
where manager_id in (100, 101, 201);
```

EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100



## Verwendung von Vergleichsoperatoren (Forts.)

```
select last_name, job_id
from EMPLOYEES
where job_id not in ('IT_PROG', 'ST_CLERK', 'SA_REP');
```

LAST_NAME	JOB_ID
King	AD_PRE
Kochhar	AD_V
De Haan	AD_V
Mourgos	ST_MAN
Zlotkey	SA_MAN
Whalen	AD_ASST
Hartstein	MK_MAN
Fay	MK_REP
Higgins	AC_MGR
Gietz	AC_ACCOUNT

## Einschub: Zeichenketten und Kalenderdaten

- Zeichenketten (Strings) und Kalenderdaten werden in einfache Anführungszeichen '...' gesetzt.
- String-Werte sind case-sensitiv, Datums-Werte sind format-sensitiv. Es gibt ein Standardformat für Daten (siehe weiter unten).
- In Suchmustern für Strings dürfen Wildcards verwendet werden:
  - % steht für kein, ein oder mehrere beliebige Zeichen
  - \_ steht für ein Zeichen
- Wildcards lassen sich beliebig miteinander und mit normalen Zeichen kombinieren.
- Um nach den Zeichen '%', '\_' zu suchen, stellt man ein selbstdefinier-tes Escape-Zeichen voran.

## Verwendung der like-Bedingung

```
select first_name, last_name  
from EMPLOYEES  
where first_name like 'El%n%';
```

```
select distinct last_name  
from EMPLOYEES  
where last_name like '_o%';
```

```
select distinct job_id  
from EMPLOYEES  
where job_id like '%!_MA%' escape '!';
```

## Verwendung der is-null-Bedingung

Mit dem **is-null**-Operator prüft man, ob ein Wert ein Nullwert (**null**) ist.

```
select last_name, manager_id  
from   EMPLOYEES  
where  manager_id is null;
```

LAST_NAME	MANAGER_ID
King	

## Logische Verknüpfung von Vergleichsbedingungen

Operator	Erfüllung der verknüpften Bedingung
... <b>and</b> ...	wahr, wenn <i>beide</i> Bedingungen wahr sind
... <b>or</b> ...	wahr, wenn mind. <i>eine</i> der beiden Bedingungen wahr ist
<b>not</b> ...	wahr, wenn die folgende Bedingung falsch ist

## Präzedenz von Operatoren

Stufe	Operatoren
1	+, − (unär)
2	*, /
3	+, − (binär),
4	alle Vergleichsoperatoren
	...
	...
7	logische Verknüpfung <b>not</b>
8	logische Verknüpfung <b>and</b>
9	logische Verknüpfung <b>or</b>

Mit Klammersetzungen kann man die Präzedenzen außer Kraft setzen.

## Verwendung des and- und des or-Operators

```
select employee_id, last_name, job_id, salary
from EMPLOYEES
where salary >= 10000 and job_id like '%MAN%';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

```
select employee_id, last_name, job_id, salary
from EMPLOYEES
where salary >= 10000 or job_id like '%MAN%';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

## Auswirkung bzw. Überschreiben von Präzedenzen

```
select last_name, job_id, salary
from EMPLOYEES
where job_id = 'SA_REP' or job_id = 'AD_PRES' and salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

```
select last_name, job_id, salary
from EMPLOYEES
where (job_id = 'SA_REP' or job_id = 'AD_PRES') and salary > 15000;
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

## Verwendungen der order-by-Klausel

- Die auszugebenden Zeilen werden aufsteigend sortiert (**asc**=ascending, voreingestellt):

```
select last_name, job_id, department_id, hire_date
from   EMPLOYEES
order by hire_date [asc];
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

...



## Verwendungen der order-by-Klausel (Forts.)

- In absteigender Reihenfolge sortieren (**desc**=descending):

```
select last_name, job_id, department_id, hire_date
from   EMPLOYEES
order  by hire_date desc;
```

- Nach Spaltenalias sortieren:

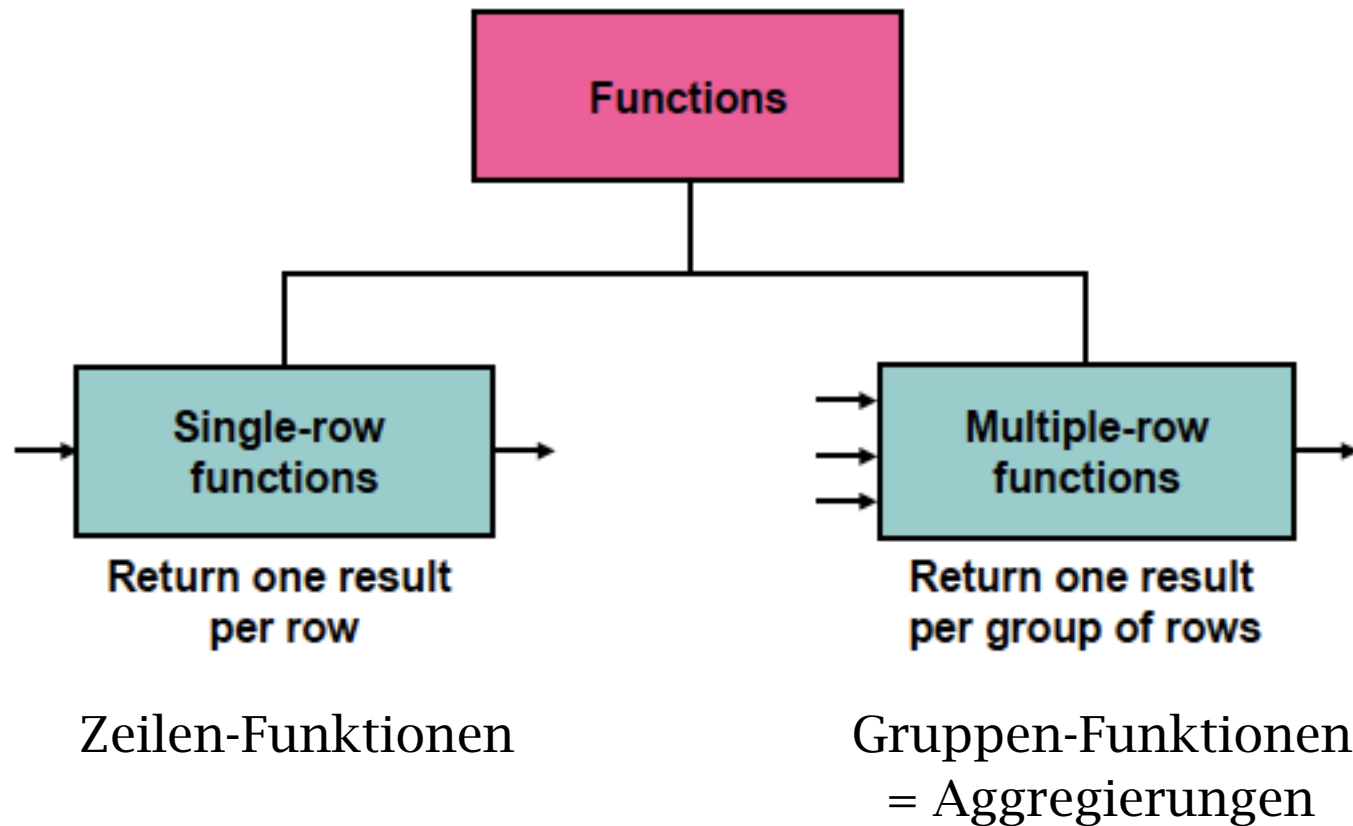
```
select employee_id, last_name, salary*12 annsal
from   EMPLOYEES
order  by annsal;
```

- Nach mehreren (vielleicht versteckten) Spalten sortieren:

```
select department_id, last_name
from   EMPLOYEES
order by department_id, salary desc;
```

## 3.2.2 Zeilen-Funktionen

### Arten von SQL-Funktionen



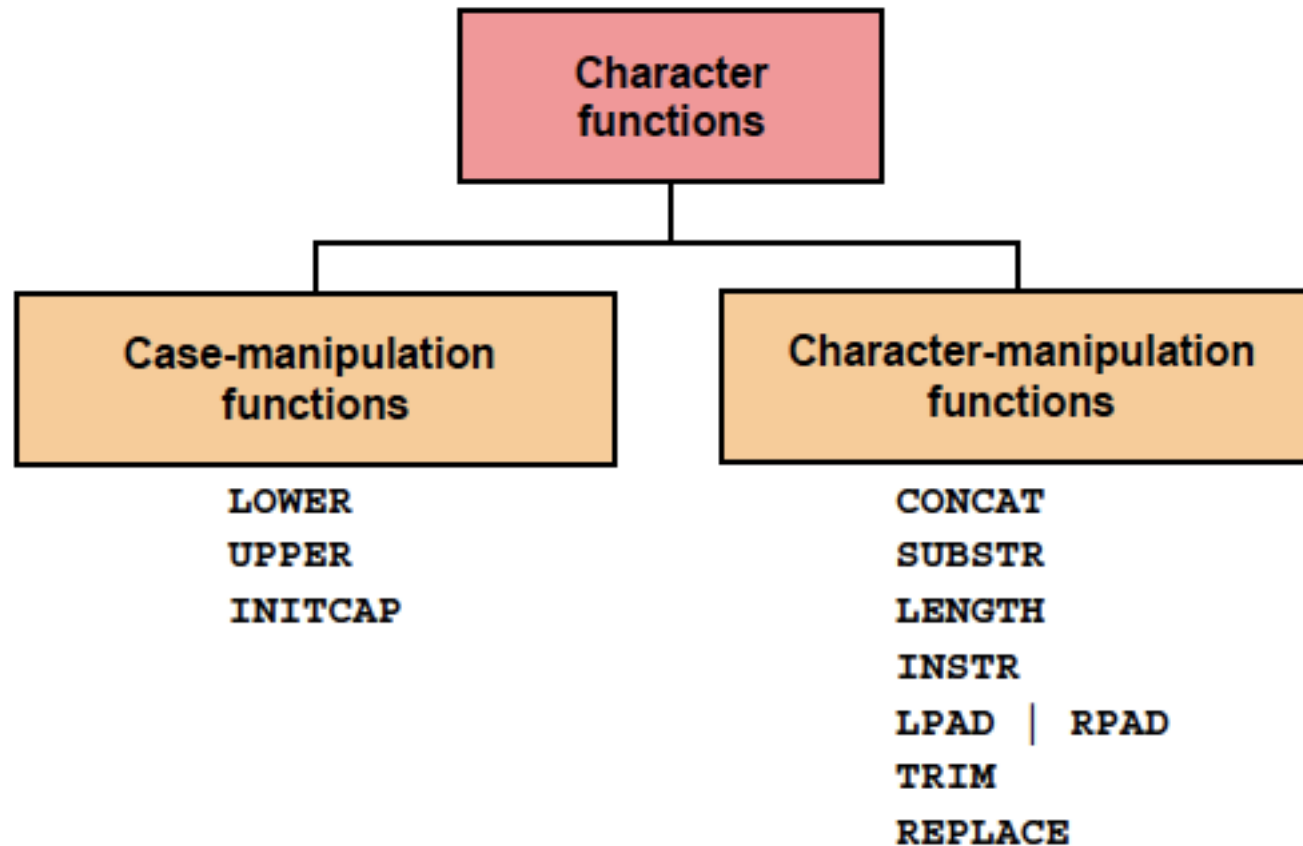
## Arten von SQL-Funktionen (Forts.)

- bearbeiten jede Zeile, die in der jeweiligen Klausel bearbeitet werden soll
- geben für jede solche Zeile ein Ergebnis zurück
- akzeptieren Argumente und geben einen Wert zurück

*Funktionsname* [ (*arg1* , *arg2* , ... ) ]

- akzeptieren als Argumente  
Konstanten, Spaltennamen und Ausdrücke
- manipulieren Daten
- können den Datentyp modifizieren
- können verschachtelt sein

## Funktionen auf Strings (Zeichenketten, Datentyp varchar2)



## Case-Manipulation

Diese Funktionen manipulieren die Groß- und Kleinschreibung von Strings:

Funktion	Ergebnis
<b>lower</b> ('SQL Kurs')	sql kurs
<b>upper</b> ('SQL Kurs')	SQL KURS
<b>initcap</b> ('SQL Kurs') <sup>4</sup>	Sql Kurs

Verwendung:

```
select employee_id, last_name, department_id
from EMPLOYEES
where lower(last_name) = 'higgins';
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

---

<sup>4</sup>Als Worttrenner dienen nicht nur Leerzeichen, sondern auch alle Sonderzeichen.

## String-Manipulation

Funktion	Ergebnis
<b>concat</b> ('Hello', 'World') <i>alternativ:</i> 'Hello'    'World'	HelloWorld HelloWorld
<b>substr</b> ('HelloWorld', 1, 5)	Hello
<b>substr</b> ('HelloWorld', 6)	World
<b>substr</b> ('HelloWorld', -4, 2)	or
<b>length</b> ('HelloWorld')	10
<b>instr</b> ('HelloWorld', 'Wo')	6
<b>instr</b> ('HelloWorld', 'No')	0
<b>rpad</b> ('Hello', 10, '*')	Hello*****
<b>lpad</b> (2400, 7, '*')	***2400
<b>replace</b> ('JACK and JUE', 'J', 'BL')	BLACK and BLUE
<b>trim</b> ('H' FROM 'HelloWorld')	elloWorld

## Verwendung von Funktionen zur String-Manipulation

```
select employee_id, concat(first_name, last_name) name,  
       job_id, length(last_name),  
       instr(last_name, 'a') "Contains 'a'?"  
from EMPLOYEES  
where substr(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

```
select last_name, upper(concat(substr(last_name, 1, 8), 'US'))  
from EMPLOYEES  
where department_id = 60;
```

LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8
Hunold	HUNOLD_US
Ernst	ERNST_US
Lorentz	LORENTZ_US

## Funktionen auf Zahlen (Datentyp number)

- arithmetische Operatoren +, −, \*, /
- **round**: rundet Wert auf angegebene Dezimalstelle
- **trunc**: schneidet Wert nach der angegebenen Dezimalstelle ab
- **mod**: gibt den Rest bei ganzzahliger Division zurück

Funktion	Ergebnis
<b>round</b> (45.926, 2)	45.93
<b>trunc</b> (45.92, 0) = <b>trunc</b> (45.92)	45
<b>mod</b> (1600, 300)	100

z.B.

```
select round(45.923, 2), round(45.923, 0), round(45.923, -1)  
from DUAL;5
```

ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
45.92	46	50

<sup>5</sup>DUAL ist eine Dummytabelle, die genutzt werden kann, um Ergebnisse von Funktionen und Kalkulationen anzeigen zu lassen.



## Umgang mit Kalender- bzw. Zeitdaten (Datentyp date)

- Die Oracle-Datenbank speichert Datumsangaben in einem internen numerischen Format: Jahrhundert, Jahr, Monat, Tag, Stunden, Minuten und Sekunden.
- Das Standardformat hier ist DD-MON-RR, in den Übungen DD.MM.RR.
  - RR erlaubt den Gebrauch von Daten der zweiten Hälfte des 20. Jahrhundert (1950-1999) im 21. Jahrhundert durch Angabe der letzten zwei Zeichen des Jahres.
  - Und es erlaubt den Gebrauch von Daten der ersten Hälfte des 21. Jahrhundert (2000-2049) auf die gleiche Weise.

Verwendung:

```
select last_name, hire_date
from EMPLOYEES
where hire_date < '01-FEB-10';
```

LAST_NAME	HIRE_DATE
King	17-JUN-87
Whalen	17-SEP-87

## Anzeigen von Kalenderdaten als Strings

`to_char(Datum [, 'Formatmodell' ])`

Typische Elemente des Formatmodells:

Element	Ergebnis
YYYY	Komplettes Jahr als Nummer
YY	Letzte zwei Ziffern des Jahres
YEAR	Gesprochene Version des Jahres (in Englisch)
MM	Zweistelliger Wert des Monats
MONTH	Vollständiger Name des Monats
MON	Dreistellige Abkürzung des Monats
DD	Zweistelliger Wert für den Tag im Monat
DAY	Vollständiger Name des Wochentags
DY	Dreistellige Abkürzung für den Wochentag
DD	Zweistelliger Wert für den Tag im Monat
fm	Modifizier vor einem anderen Element: unterdrückt zusätzliche Leerzeichen oder führende Nullen

- Das Formatmodell kombiniert solche Elemente mit Sonderzeichen und Strings.
- Lässt man es weg, wird das Standardformat verwendet.

## Anzeigen von Kalenderdaten als Strings (Forts.)

```
select last_name,  
       to_char(hire_date, 'fmDD Month YYYY') as hiredate  
from   EMPLOYEES;
```

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

Weitere Formatelemente:

- Zeitelemente formatieren den Teil mit der Uhrzeit im Datum, z.B.:  
HH24:MI:SS AM      ~>      15:45:32 PM
- Strings werden in doppelten Anführungszeichen eingefügt:  
DD "of" MONTH      ~>      12 of OCTOBER
- Zahlsuffixe liefern gesprochene Versionen der Zahl:  
ddspth      ~>      fourteenth

## Arithmetik auf Kalenderdaten

- Addition oder Subtraktion einer Zahl (Anzahl von Tagen) zu oder von einem Datum berechnet ein neues Datum.
- Subtraktion zweier Daten berechnet die Anzahl von Tagen zwischen den beiden Daten.
- Stunden lassen sich addieren usw., indem die Anzahl von Stunden durch 24 dividiert wird (Tagesanteile).

z.B.

```
select last_name, (sysdate - hire_date) / 7 as weeks  
from EMPLOYEES  
where department_id = 90;
```

LAST_NAME	WEEKS
King	744.245395
Kochhar	626.102538
De Haan	453.245395

## Datumsfunktionen

Funktion	Ergebnis
<b>months_between</b>	berechnet Anzahl von Monaten zwischen zwei Daten
<b>add_months</b>	addiert angeg. Zahl von Monaten zum Datum
<b>next_day</b>	liefert das nächste Datum eines angeg. Wochentags
<b>last_day</b>	letzter Tag des Monats
<b>round/trunc</b>	rundet das Datum / "schneidet es ab"
<b>sysdate</b>	das aktuelle Datum bzw. die aktuelle Zeit

Beispiele:

Funktion	Ergebnis
<b>months_between('01-SEP-95', '11-JAN-94')</b>	19.6774194
<b>add_months('11-JAN-94', 6)</b>	'11-JUL-94'
<b>next_day('01-SEP-95', 'FRIDAY')</b>	'08-SEP-95'
<b>last_day('01-FEB-96')</b>	'29-FEB-96'
<b>round('25-JUL-03', 'MONTH')</b>	01-AUG-03
<b>round('25-JUL-03', 'YEAR')</b>	01-JAN-04
<b>trunc('25-JUL-03', 'MONTH')</b>	01-JUL-03

## Anzeigen von Zahlen als Strings

`to_char(Zahl [, 'Formatmodell' ])`

Format-Element	Ergebnis
9	repräsentiert eine Ziffer
0	auch eine Ziffer; erzwingt aber das Anzeigen einer führenden / abschließenden Null
\$	(führendes) Dollarzeichen
L	lokal eingestelltes Währungszeichen
.	Dezimalpunkt
,	Komma als Separator von Tausendergruppen

z.B.

```
select to_char(salary, '$99,999.00') salary
from   EMPLOYEES
where  last_name = 'Ernst';
```

SALARY
\$6,000.00

## Konversion von Strings in Zahlen bzw. Daten

Umgekehrt zu **to\_char** konvertieren die Funktionen **to\_number** und **to\_date** einen String, der ein angegebenes passendes Format hat, in eine Zahl bzw. in ein Kalenderdatum:

**to\_number**(*String* [, '*Formatmodell*'])

**to\_date**(*String* [, '*Formatmodell*'])

Oft erfolgen Typkonversionen aber auch implizit ohne eine der o.g. Funktionen.

## Allgemeine Funktionen

- Die **decode**-Funktion<sup>6</sup> und **case**-Ausdrücke stellen eine **if-then-else**-Logik für Ausdrücke innerhalb einer SQL-Anfrage zur Verfügung; sie ermöglichen also (auf äquivalente Weise) bedingte Ausdrücke.

**decode**( {*Spalte* | *Ausdruck*},  
          *Vergleichsausdruck1* , *Rückgabebausdruck1*  
          [, *Vergleichsausdruck2* , *Rückgabebausdruck2* , ...]  
          [, *Standardausdruck*] )

**case** {*Spalte* | *Ausdruck*}  
      **when** *Vergleichsausdruck1* **then** *Rückgabebausdruck1*  
      [ **when** *Vergleichsausdruck2* **then** *Rückgabebausdruck2*  
      ... ]  
      [ **else** *Standardausdruck* ]  
**end**

- Wenn kein Standardausdruck angegeben ist, wird **null** angenommen.

---

<sup>6</sup>Oracle-spezifisches SQL, d.h. nicht SQL-Standard



## Verwendung von decode-Funktionen/case-Ausdrücken

```
select last_name, job_id, salary,  
       decode(job_id, 'IT_PROG', 1.10*salary,  
                'ST_CLERK', 1.15*salary,  
                'SA_REP', 1.20*salary,  
                salary)  
       revised_salary  
from   EMPLOYEES;
```

```
select last_name, job_id, salary,  
       case job_id when 'IT_PROG' then 1.10*salary  
                 when 'ST_CLERK' then 1.15*salary  
                 when 'SA_REP' then 1.20*salary  
                 else salary  
       end revised_salary  
from   EMPLOYEES;
```

## Verwendung von decode-Funktionen/case-Ausdrücken (Forts.)

Anfrage:

Gehaltsabhängige Steuersätze für die Angestellten aus Abteilung 80?

```
select last_name, salary,  
       decode(trunc(salary/2000,0),  
              0, 0.00,  
              1, 0.09,  
              2, 0.20,  
              3, 0.30,  
              4, 0.40,  
              5, 0.42,  
              6, 0.44,  
              0.45) tax_rate  
from   EMPLOYEES  
where  department_id = 80;
```

## Entschlüsseln von Nullwerten

Die folgenden Funktionen spezialisieren **decode** für **null**-Ausdrücke:

- **nvl**(*expr1*, *expr2*) = **decode**(*expr1*, **null**, *expr2*, *expr1*)  
— ersetzt *expr1* durch *expr2*, wenn *expr1* **is null**
- **nvl2**(*expr1*, *expr2*, *expr3*) = **decode**(*expr1*, **null**, *expr3*, *expr2*)  
— ersetzt *expr1* durch *expr2*, sofern *expr1* **is not null**,  
sonst durch *expr3*
- **coalesce**(*expr1*, *expr2*, ..., *exprN*)  
— gibt den ersten **not null**-Ausdruck *expr...* oder *exprN* zurück;  
für N=3:  
= **decode**(*expr1*, **null**, **decode**(*expr2*, **null**, *expr3*, *expr2*), *expr1*)

Achtung: Die Datentypen müssen passen.

## Verwendung der nvl- und nvl2-Funktion

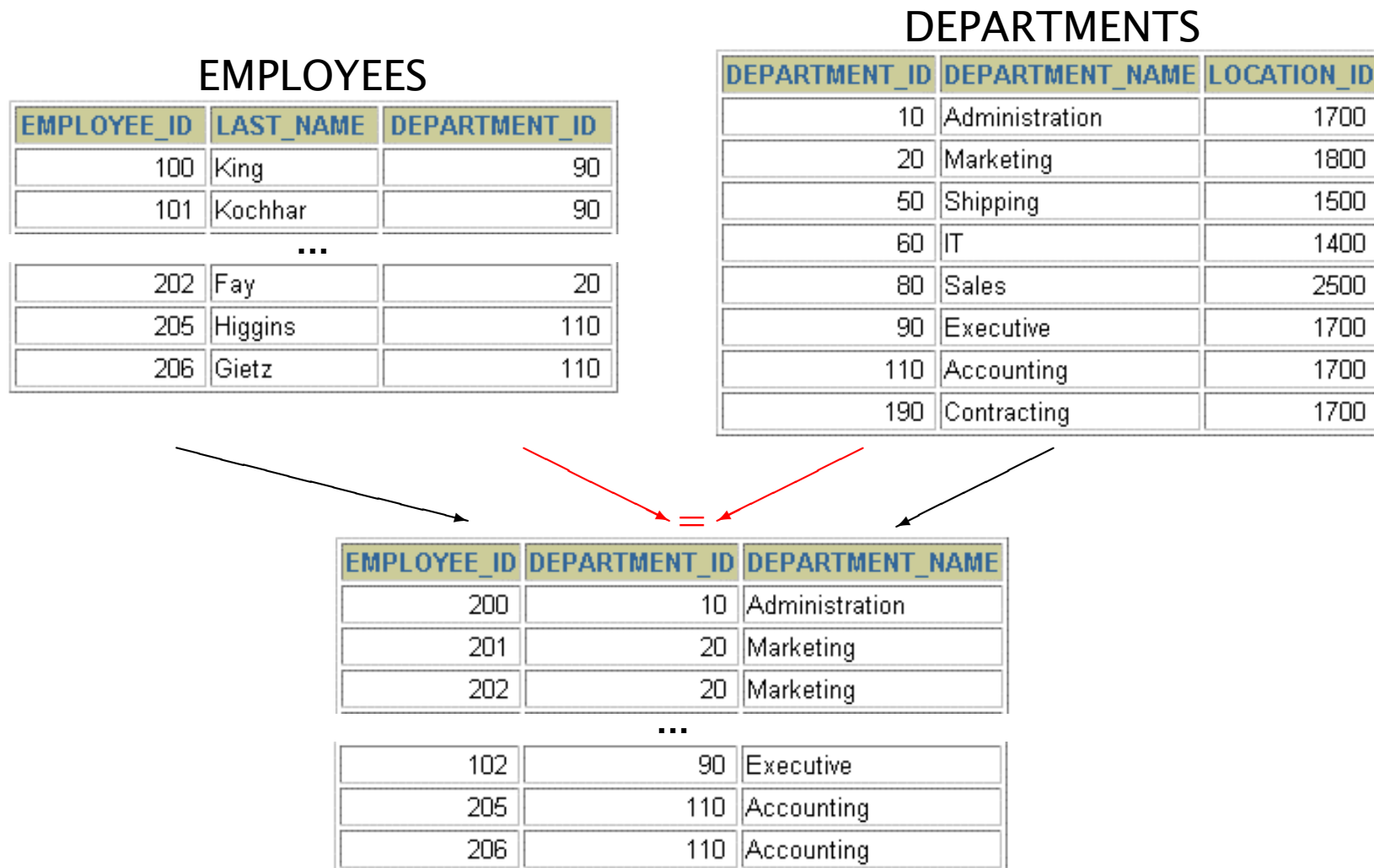
```
select last_name, salary, nvl(commission_pct, 0),  
       (salary*12) + (salary*12*nvl(commission_pct, 0)) an_sal  
from   EMPLOYEES;
```

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
...			

```
select last_name, salary, commission_pct  
       nvl2(commission_pct, 'SAL+COMM', 'SAL') income  
from   EMPLOYEES where department_id in (50, 80);
```

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
Zlotkey	10500	.2	SAL+COMM
Abel	11000	.3	SAL+COMM
Taylor	8600	.2	SAL+COMM
Mourgos	5800		SAL
Rajs	3500		SAL
Davies	3100		SAL
Matos	2600		SAL
Vargas	2500		SAL

### 3.2.3 Anfragen an mehrere Tabellen



## Equijoins (Gleichheitsverbunde) – Klassische Syntax (SQL-86)

Mit Joins kann man Daten aus mehreren Tabellen anfragen:

```
select Tabelle1.Spalte, Tabelle2.Spalte  
from   Tabelle1, Tabelle2  
where Tabelle1.Spalte1 = Tabelle2.Spalte2;
```

- Die Join-Bedingung wird in die **where**-Klausel geschrieben.
- Wenn es denselben Spaltennamen in mehr als einer Tabelle gibt, wird der Tabellename als Präfix vor den Spaltennamen gesetzt.

## Equijoins (Gleichheitsverbunde) – Klassische Syntax (SQL-86) (Forts.)

```
select EMPLOYEES.employee_id, EMPLOYEES.last_name,  
       EMPLOYEES.department_id, DEPARTMENTS.department_id,  
       DEPARTMENTS.location_id  
from   EMPLOYEES, DEPARTMENTS  
where  EMPLOYEES.department_id = DEPARTMENTS.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

...

## Zuordnung mehrdeutiger Spaltennamen

- Um Spaltennamen, die in mehreren Tabellen vorkommen, eindeutig zuzuordnen, nutzt man Tabellennamen oder Tabellenalias als Präfixe<sup>7</sup>.
- Solche Präfixe verbessern die Performance.
- Tabellenalias vereinfachen Anfragen.

```
select e.employee_id, e.last_name,  
       e.department_id, d.department_id,  
       d.location_id  
from   EMPLOYEES e, DEPARTMENTS d  
where  e.department_id = d.department_id;
```

---

<sup>7</sup>Deshalb dürfen im Beispiel alle Präfixe außer für department\_id weggelassen werden.



## Equijoins – SQL:1999-Syntax

```
select Tabelle1.Spalte, Tabelle2.Spalte  
from Tabelle1  
    [natural join Tabelle2] |  
    [join Tabelle2 using (Spaltenname)] |  
    [join Tabelle2 on Tabelle1.Spaltenname = Tabelle2.Spaltenname] |  
    [{ left | right | full } outer join Tabelle2  
        on Tabelle1.Spaltenname = Tabelle2.Spaltenname] |  
    [{ cross join | , } Tabelle2];
```

## Natural Joins

- Die **natural-join**-Klausel bezieht sich auf alle Spalten, die in den beiden Tabellen dieselben Namen besitzen.
- Ein Natural Join kombiniert die Zeilen von beiden Tabellen, die gleiche Werte in den gleichnamigen Spalten haben.
- Besitzen zwei Spalten denselben Namen, jedoch verschiedene Datentypen, so wird ein Fehler gemeldet.

Verwendung:

```
select department_id, department_name, location_id, city  
from DEPARTMENTS natural join LOCATIONS;
```

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

...

## Equijoins mit Hilfe der using-Klausel

- Besitzen mehrere Spalten den gleichen Namen, aber es sollen/können nicht alle genutzt werden, so kann die **join**-Klausel mit der **using**-Klausel modifiziert werden. Diese spezifiziert die Spalten, die für einen Equijoin genutzt werden sollen.
- Für diese Spalten dürfen keine Präfixe genutzt werden.
- Natural Join und **using**-Klausel schließen sich gegenseitig aus.

Verwendung:

```
select EMPLOYEES.employee_id, EMPLOYEES.last_name,  
        DEPARTMENTS.location_id, department_id  
from   EMPLOYEES join DEPARTMENTS using (department_id);
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
200	Whalen	1700	10
201	Hartstein	1800	20
202	Fay	1800	20
124	Mourgos	1500	50
141	Rajs	1500	50
142	Davies	1500	50
144	Vargas	1500	50
143	Matos	1500	50

...

## Joins mit Hilfe der on-Klausel

- Die Join-Bedingung für den Natural Join war im wesentlichen ein Equijoin auf allen Spalten mit demselben Namen.
- Die **on**-Klausel wird genutzt, um beliebige Joinbedingungen anzugeben. – Durch die Trennung von der **where**-Klausel erleichtert es die **on**-Klausel, die Anfrage zu verstehen.

Verwendung:

```
select e.employee_id, e.last_name,  
       e.department_id, d.department_id, d.location_id  
from   EMPLOYEES e join DEPARTMENTS d  
       on e.department_id = d.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

## Joins mit Hilfe der on-Klausel (Forts.)

Beispiel für einen Zweifach-Join:

```
select employee_id, city, department_name
from EMPLOYEES e
  join DEPARTMENTS d
    on e.department_id = d.department_id
  join LOCATIONS l
    on d.location_id = l.location_id;
```


EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

...

## Self-Joins (Selbstverbunde)

Beispiel:

EMPLOYEES (worker)			EMPLOYEES (manager)		
EMPLOYEE_ID	LAST_NAME	MANAGER_ID	EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King		100	King	
101	Kochhar	100	101	Kochhar	100
102	De Haan	100	102	De Haan	100
103	Hunold	102	103	Hunold	102
104	Ernst	103	104	Ernst	103
107	Lorentz	103	107	Lorentz	103
124	Mourgos	100	124	Mourgos	100
...			...		



Die MANAGER\_ID in der worker-Ausprägung der Tabelle EMPLOYEES soll übereinstimmen mit der EMPLOYEE\_ID in der manager-Ausprägung derselben Tabelle.

## Self-Join (Selbstverbund) mit Hilfe der on-Klausel

```
select worker.last_name || ' works for ' || manager.last_name
from EMPLOYEES worker join EMPLOYEES manager
on (worker.manager_id = manager.employee_id);
```

### ... in der klassischen Syntax

```
select worker.last_name || ' works for ' || manager.last_name
from EMPLOYEES worker, EMPLOYEES manager
where worker.manager_id = manager.employee_id;
```

WORKER.LAST_NAME  'WORKSFOR'  MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

...

## Nicht-Equijoins

Beispiel:

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

...

JOB\_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000



Um die korrespondierenden Gehaltsstufen zu finden, muss das Gehalt in der Tabelle EMPLOYEES zwischen dem niedrigsten und dem höchsten Gehalt in der Tabelle JOB\_GRADES liegen.



## Nicht-Equi Joins (Forts.)

```
select e.last_name, e.salary, j.gra  
from EMPLOYEES e join JOB_GRADES j  
on e.salary between j.lowest_sal and j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

...

## Outer Joins

Motivation:

DEPARTMENTS

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

...

In Abteilung 190 gibt es keine Angestellten. Trotzdem soll diese beim Join mit ausgegeben werden.

## Inner Join versus Outer Join

- In SQL:1999 nennt man den Join von zwei Tabellen, der nur passende Zeilen zurückliefert, **Inner Join**.
- Den Join zwischen zwei Tabellen, der die Werte eines Inner Joins sowie die nicht passenden Zeilen von der linken (oder rechten) Tabelle zurückliefert, nennt man **linker (oder rechter) Outer Join**.
- Den Join zwischen zwei Tabellen, der die Werte eines Inner Joins sowie die Ergebnisse eines linken und rechten Outer Joins zurückliefert, nennt man **vollen Outer Join**.
- Fehlende Werte werden mit **null** aufgefüllt.

## Rechter Outer Join

```
select e.last_name, d.department_id, d.department_name
from EMPLOYEES e right outer join DEPARTMENTS d
on (e.department_id = d.department_id);
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
Davies	50	Shipping
...		
Kochhar	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
	190	Contracting

## Linker Outer Join

```
select e.last_name, e.department_id8, d.department_name
from EMPLOYEES e left outer join DEPARTMENTS d
on (e.department_id = d.department_id);
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

---

<sup>8</sup>department\_id ist zwar Fremdschlüssel von EMPLOYEES bzgl. DEPARTMENTS, darf aber auch Nullwerte annehmen.

## Voller Outer Join

```
select e.last_name, d.department_id, d.department_name
from EMPLOYEES e full outer join DEPARTMENTS d
on (e.department_id = d.department_id);
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
	190	Contracting

## Linke bzw. rechte Outer Joins in der klassischen Syntax

Operator für den linken bzw. rechten Outer Join ist das Pluszeichen (+). Es steht jedoch auf der Seite, auf welcher die Nullwerte hinzugefügt werden können.

```
select Tabelle1.Spalte, Tabelle2.Spalte  
from   Tabelle1, Tabelle2  
where Tabelle1.Spalte = Tabelle2.Spalte (+);
```

```
select Tabelle1.Spalte, Tabelle2.Spalte  
from   Tabelle1, Tabelle2  
where Tabelle1.Spalte (+) = Tabelle2.Spalte;
```

## Linke bzw. rechte Outer Joins in der klassischen Syntax (Forts.)

```
select e.last_name, e.department_id9, d.department_name
from   EMPLOYEES e, DEPARTMENTS d
where  e.department_id (+) = d.department_id;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
...		
Gietz	110	Accounting
		Contracting

<sup>9</sup>Beachte: department\_id stammt hier aus EMPLOYEES.



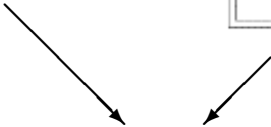
## Bildung eines Kartesischen Produkts (Kreuzprodukt)

EMPLOYEES (20 Zeilen)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS (8 Zeilen)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700

...

20 x 8 = 160 Zeilen

## Bildung eines Kartesischen Produkts (Kreuzprodukt) (Forts.)

```
select last_name, department_name  
from   EMPLOYEES cross join DEPARTMENTS;
```

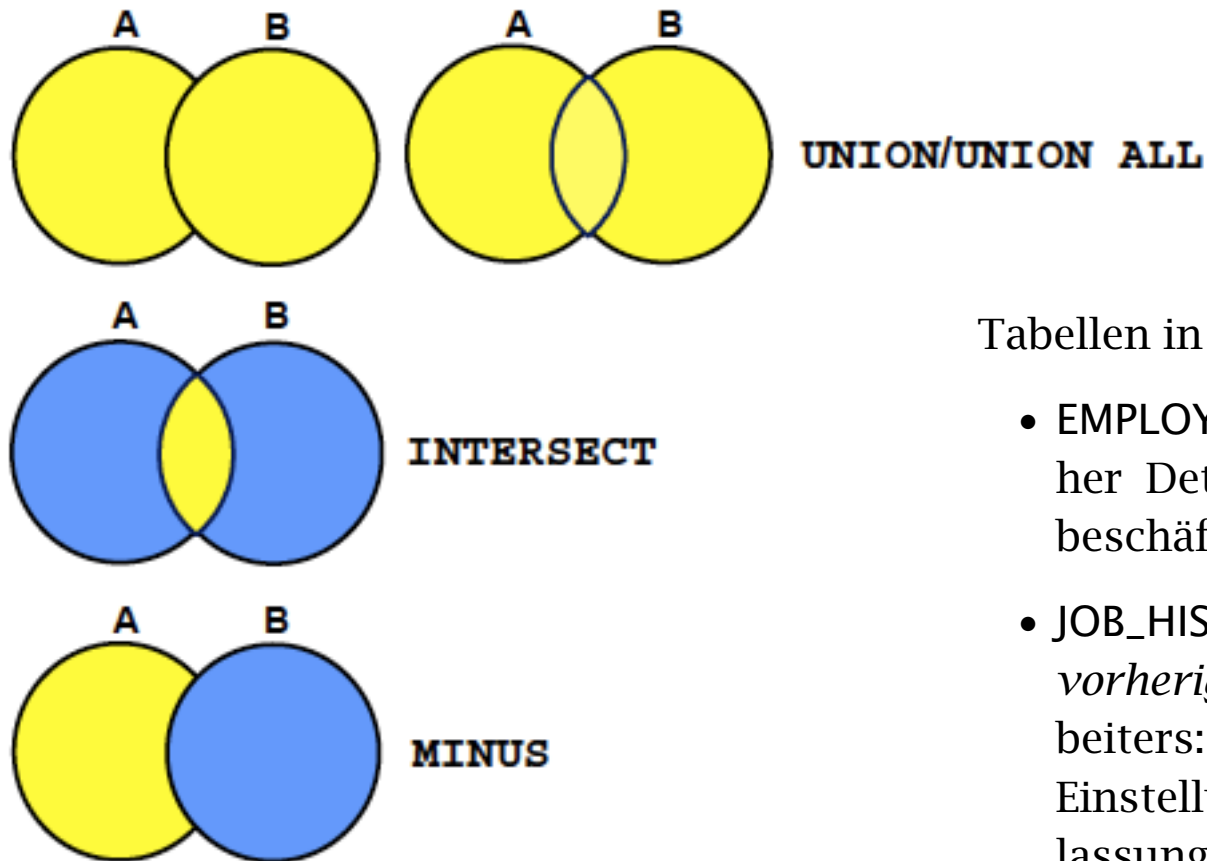
oder:

```
select last_name, department_name  
from   EMPLOYEES , DEPARTMENTS;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

...

### 3.2.4 Mengen-Operatoren



Tabellen in diesem Abschnitt:

- **EMPLOYEES** beinhaltet wie bisher Details über alle *aktuell* beschäftigten Mitarbeiter
- **JOB\_HISTORY** speichert alle *vorherigen* Jobs eines Mitarbeiters: die Mitarbeiter-ID, das Einstellungsdatum, das Entlassungsdatum, die Berufsbezeichnung und die Abteilungsnummer des Jobs.

## Verwendung des union-Operators

Der **union**-Operator gibt die Ergebnisse von zwei Teilanfragen zurück, nachdem er alle Duplikate entfernt hat. (**union all** liefert auch Duplikate.)

Beispiel: *Zeige die aktuellen und früheren Employee/Job/Department-IDs von allen Mitarbeitern an; dabei soll jede Kombination nur einmal ausgegeben werden.*

```
select employee_id, job_id, department_id from EMPLOYEES
```

**union**

```
select employee_id, job_id, department_id from JOB_HISTORY  
order by employee_id, job_id, department_id;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	AD_PRES	90
101	AD_VP	90
...		
200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90

...

## Verwendung des intersect-Operators

Bsp.: Zeige die aktuelle Employee/Job-ID von jedem Mitarbeiter an, der momentan eine Job-ID hat, welche identisch mit einer alten Job-ID ist.

```
select employee_id, job_id from EMPLOYEES  
intersect  
select employee_id, job_id from JOB_HISTORY  
order by employee_id;
```

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

## Verwendung des minus-Operators

Beispiel: *Zeige die aktuellen Employee/Job-IDs von Mitarbeitern an, die einen Job ausüben, den sie nie zuvor ausgeübt haben.*

```
select employee_id, job_id from EMPLOYEES
```

**minus**

```
select employee_id, job_id from JOB_HISTORY  
order by employee_id;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AD_VP
102	AD_VP
103	IT_PROG
...	
201	MK_MAN
202	MK_REP
205	AC_MGR
206	AC_ACCOUNT

## Regeln für die Verwendung von Mengen-Operatoren

- Die Ausdrücke in der **select**-Liste müssen bezüglich Anzahl und Datentyp übereinstimmen.
- Klammern können genutzt werden, um die Ausführungsreihenfolge zu ändern.
- Zeilenduplikate werden von allen Mengen-Operatoren automatisch eliminiert, mit Ausnahme von **union all**.
- Die **order-by**-Klausel:
  - kann nur am Ende des gesamten Statements genutzt werden.
  - akzeptiert nur Spaltennamen/-alias aus der ersten **select**-Anweisung oder die Positionsnotation.
- Auch im Ergebnis erscheinen die Spaltennamen/-alias aus der ersten **select**-Liste.

## Angleichung heterogener select-Listen

Beispiel: Nutze den **union**-Operator, um *alle Department-IDs, deren Orts-IDs, sowie die Einstellungsdaten der Mitarbeiter anzuzeigen.*

```
select department_id, to_number(null) location, hire_date
from EMPLOYEES
union
select department_id, location_id, to_date(null)
from DEPARTMENTS
order by department_id;
```

DEPARTMENT_ID	LOCATION	HIRE_DATE
10	1700	
10		17-SEP-87
20	1800	
20		17-FEB-96
110	1700	
110		07-JUN-94
190	1700	
		24-MAY-99

...



## Angleichung heterogener select-Listen (Forts.)

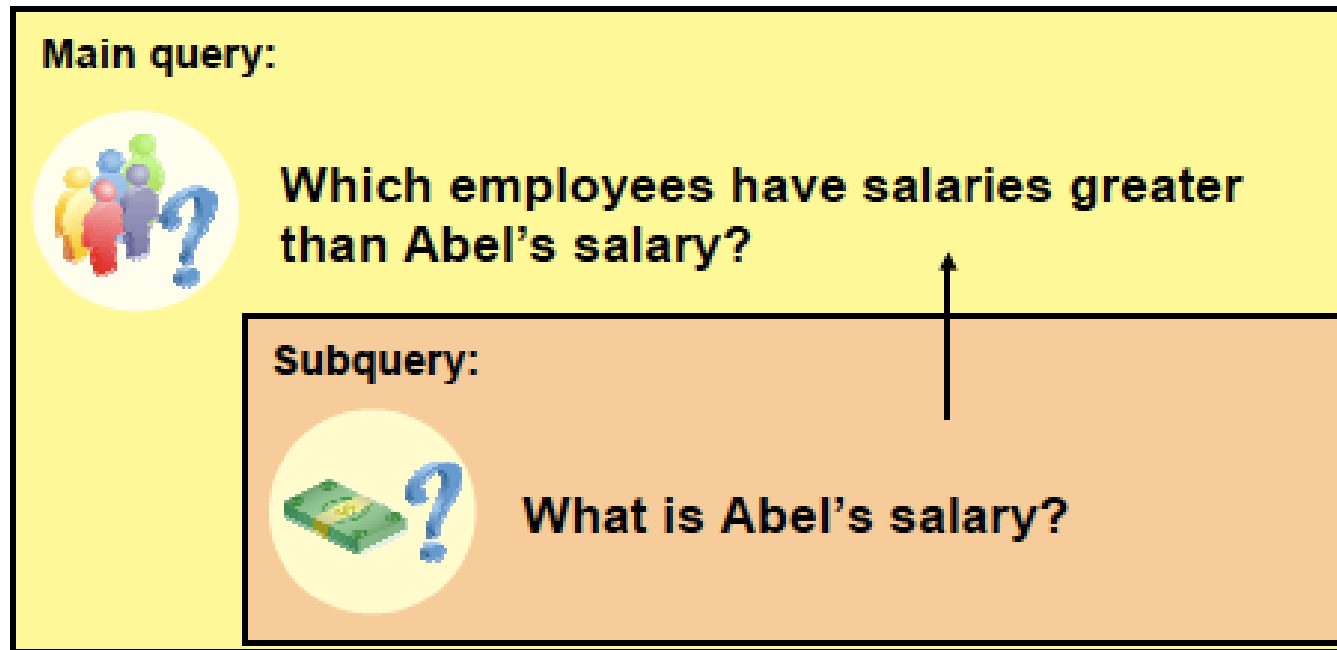
Beispiel: Nutze den **union**-Operator, um *die Employee/Job-ID und das Gehalt jedes aktuellen und früheren Mitarbeiters* auszugeben.

```
select employee_id, job_id, salary
from   EMPLOYEES
union
select employee_id, job_id, 0
from   JOB_HISTORY
order  by employee_id;
```

EMPLOYEE_ID	JOB_ID	SALARY
100	AD_PRES	24000
101	AC_ACCOUNT	0
101	AC_MGR	0
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

### 3.2.5 Anfragen mit Unteranfragen

Beispiel: *Wer hat ein Gehalt, das größer als das von Abel ist?*



## Syntax für den Anschluss von Unteranfragen

```
select Select-Liste  
from  Tabelle(n)  
where Ausdruck Vergleichsoperator  
      (select Select-Liste  
       from Tabelle  
       ...);
```

## Verwendung einer Unteranfrage

```
select last_name  
from EMPLOYEES  
where salary >
```

```
(select salary  
from EMPLOYEES  
where last_name = 'Abel');
```

→ 11.000

LAST_NAME
King
Kochhar
De Haan
Hartstein
Higgins

## Zur Syntax und Ausführung von Unteranfragen

- Unteranfragen müssen in Klammern gesetzt werden.
- Eine Unteranfrage muss auf der rechten Seite des Vergleichsoperators stehen.
- In der Regel wird die **order-by**-Klausel in Unteranfragen nicht benötigt.

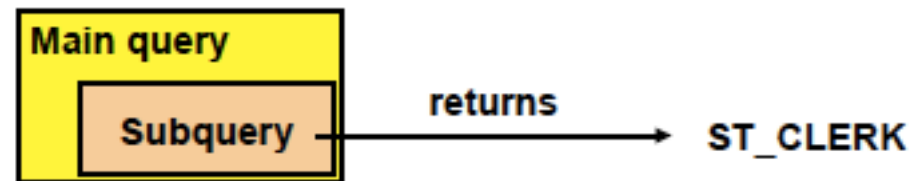
Sofern die Unteranfrage (innere Anfrage) unabhängig von der Hauptanfrage (äußere Anfrage) ist, d. h. nicht mit ihr “korreliert” ist:

- Die Unteranfrage wird einmalig vor der Hauptanfrage ausgeführt.
- Das Ergebnis der Unteranfrage wird bei der Auswertung der Hauptanfrage genutzt.

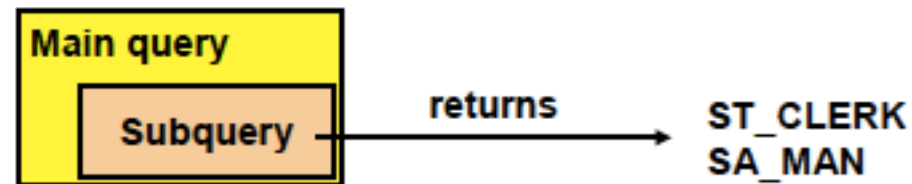
Anders bei korrelierten Unterfragen; s. u.

## Typen von Unteranfragen

- Einzeilige Unteranfragen: geben nur eine einzige Zeile zurück



- Mehrzeilige Unteranfragen



- Einzeilige Vergleichsoperatoren (=, >, >=, <, <=, <>) können nur mit einzeiligen Unteranfragen genutzt werden, (>, >=, <, <=) sogar nur mit einwertigen, d.h. einzeiligen einspaltigen Unteranfragen.

## Verwendung einzeliger Unteranfragen

```
select last_name, job_id, salary
from EMPLOYEES
where job_id =
    (select job_id
     from EMPLOYEES
     where employee_id = 141)
and salary >
    (select salary
     from EMPLOYEES
     where employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

## Verwendung einzeliger Unteranfragen (Forts.)

Was ist an dieser Anfrage falsch?

```
select employee_id, last_name  
from EMPLOYEES  
where salary =  
    (select salary  
     from EMPLOYEES  
     where last_name = 'Grant');
```

→ ERROR at line 4:

ORA-01427: single-row subquery returns more than one row

Einzeiliger Operator mit mehrzeiliger Unteranfrage!



## Verwendung einzeliger Unteranfragen (Forts.)

Gibt diese Anfrage Zeilen zurück?

```
select last_name, job_id
from EMPLOYEES
where job_id =
    (select job_id
     from EMPLOYEES
     where last_name = 'Lipeck');
```

→ no rows selected

Die Unteranfrage liefert keine Werte;  
das leere Ergebnis wird wie **null** behandelt.

## Mehrzeilige Unteranfragen

- geben mehr als eine Zeile zurück
- können nur mit mehrzeiligen Vergleichsoperatoren genutzt werden:

Operator	Bedeutung
<b>in</b>	stimmt mit irgendeinem Wert der Ergebnisliste überein
<b>... any</b>	Vergleichswert muss mit <b>mindestens einem</b> Ergebniswert der Unteranfrage “vergleichen”
<b>... all</b>	Vergleichswert muss mit <b>jedem</b> Ergebniswert der Unteranfrage “vergleichen”

## Verwendung des any-Operators in mehrzeiligen Unteranfragen

```
select employee_id, last_name, job_id, salary
from EMPLOYEES
where salary < any10
    (select salary
     from EMPLOYEES
     where job_id = 'IT_PROG')
and job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

...

<sup>10</sup> ... kleiner als mind. ein Ergebniswert der Unteranfrage, die z.B. (9000,4200,6000) liefert

## Verwendung des all-Operators in mehrzeiligen Unteranfragen

```
select employee_id, last_name, job_id, salary
from EMPLOYEES
where salary < all11
    (select salary
     from EMPLOYEES
     where job_id = 'IT_PROG')
and job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

<sup>11</sup>.... kleiner als jeder Ergebniswert der Unteranfrage (9000,4200,6000)

## Nullwerte in Unteranfragen

```
select last_name  
from EMPLOYEES  
where employee_id not in  
      (select manager_id from EMPLOYEES);
```

→ no rows selected ?!<sup>12</sup>

besser:

```
... not in  
(select nvl(manager_id, 0) from EMPLOYEES);
```

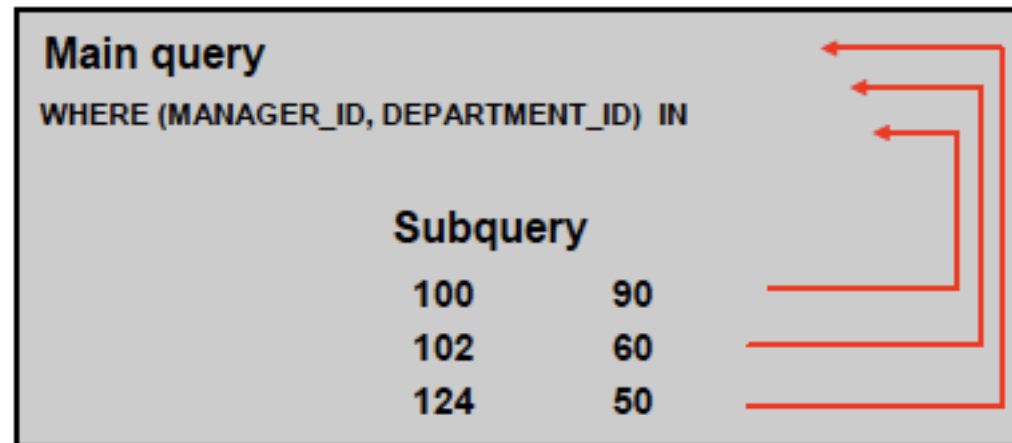
oder:

```
... not in  
(select manager_id from EMPLOYEES  
  where manager_id is not null)
```

---

<sup>12</sup>Beachte:  $x \text{ not in } \{x_1, \dots, x_k\} \hat{=} x <> x_1 \wedge \dots \wedge x <> x_k$ ;  $(x <> \perp) \equiv \perp$ ;  $(\text{true} \wedge \perp) \equiv \perp$ ; **where**  $\perp \hat{=} \text{where false}$

## Mehrspaltige Unteranfragen



Jede Zeile der Hauptanfrage wird mit Ergebniszeilen einer mehrzeiligen mehrspaltigen Unteranfrage verglichen.

## Mehrsplaltige Unteranfragen (Forts.)

Beispiel einer Unteranfrage mit paarweisem Vergleich:

*Gib die Details der Angestellten aus, die vom selben Manager gemanagt werden **und** in derselben Abteilung arbeiten **wie** einer der Angestellten mit ID 199 oder ID 174.*

```
select employee_id, manager_id, department_id
from EMPLOYEES
where (manager_id, department_id) in
      (select manager_id, department_id
       from EMPLOYEES
       where employee_id in (199, 174))
and   employee_id not in (199, 174);
```

## Mehrspaltige Unteranfragen (Forts.)

Beispiel einer Unteranfrage ohne paarweisen Vergleich:

*Gib die Details der Angestellten aus, die vom selben Manager gemanagt werden **wie** einer der Angestellten mit 174 oder 199, **und** die in derselben Abteilung arbeiten **wie** einer der Angestellten mit denselben IDs.*

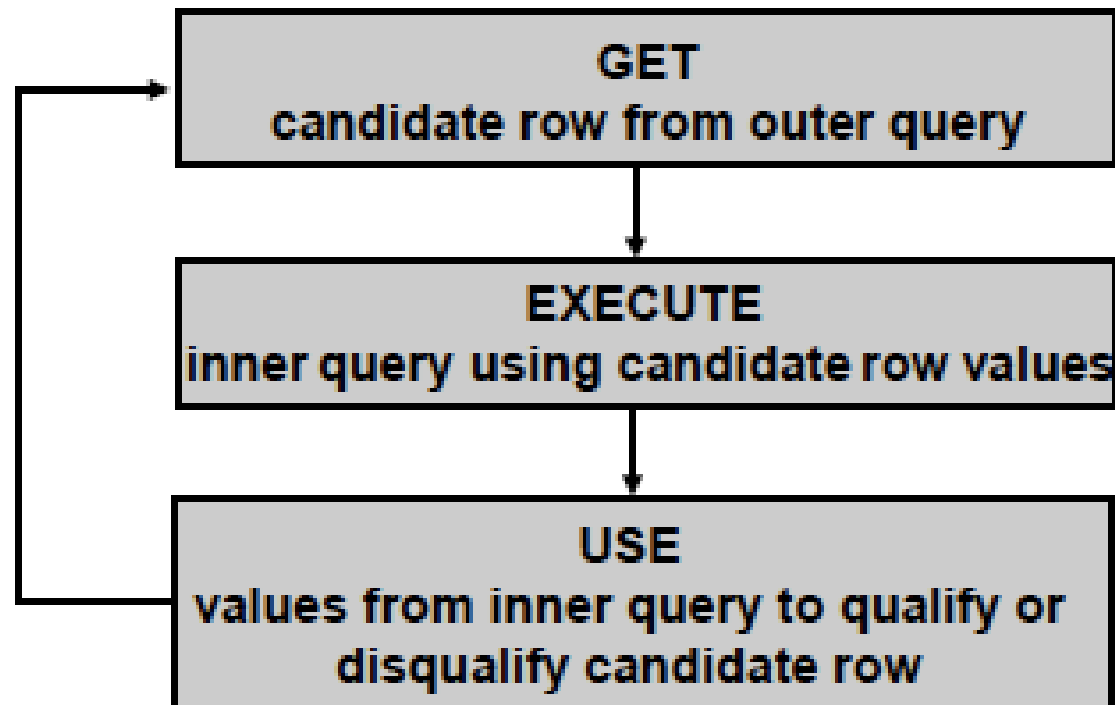
**Anderes Ergebnis !**

```
select employee_id, manager_id, department_id
from EMPLOYEES
where manager_id in
    (select manager_id
     from EMPLOYEES
     where employee_id in (174, 199))
and department_id in
    (select department_id
     from EMPLOYEES
     where employee_id in (174, 199))
and employee_id not in (174, 199);
```



## Korrelierte Unteranfragen

Eine korrelierte Unteranfrage bezieht sich auf die äußere Anfrage. Sie wird so ausgeführt, als ob sie für jede Zeile der äußeren Anfrage einzeln ausgewertet würde.



## Korrelierte Unteranfragen (Forts.)

In der Unteranfrage werden ein oder mehrere Spalten aus einer oder mehreren Tabellen der äußeren Anfrage benutzt.

Schematisches Beispiel mit Tabellenalias (falls nötig):

```
select Spalte1a, Spalte1b, ...  
from Tabelle1 aussen  
where Spalte1 Operator  
      (select Spalte2a, Spalte2b  
       from Tabelle2  
       where Ausdruck2 = aussen.Ausdruck1);
```

Bezeichner werden von innen nach außen aufgelöst; z. B. werden Spaltennamen ohne Präfix der innersten passenden Tabelle zugeordnet.

## Korrelierte Unteranfragen (Forts.)

Beispiel: *Finde alle Angestellten, die mehr verdienen als alle anderen Angestellten ihrer jeweiligen Abteilung.*

```
select last_name, salary, department_id
from   EMPLOYEES e
where  salary > all
      (select salary
       from   EMPLOYEES es
       where  es.employee_id <> e.employee_id
       and    es.department_id = e.department_id);
```

Quasi jedes Mal, wenn eine Zeile der äußeren Anfrage bearbeitet wird, wird die innere Anfrage ausgewertet.

## Verwendung des [not] exists – Operators

Beispiel: *Finde alle Angestellten, die mindestens eine Person managen.*

```
select employee_id, last_name, job_id, department_id
from EMPLOYEES e
where exists (select *
              from EMPLOYEES
              where manager_id = e.employee_id);
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
108	Greenberg	FI_MGR	100
114	Raphaely	PU_MAN	30
120	Weiss	ST_MAN	50
121	Fripp	ST_MAN	50
122	Kaufling	ST_MAN	50

...

## Verwendung des [not] exists – Operators (Forts.)

Beispiel: *Finde alle Abteilungen ohne Angestellte.*

```
select department_id, department_name
from DEPARTMENTS d
where not exists (select *
                  from EMPLOYEES
                  where department_id = d.department_id);
```

DEPARTMENT_ID	DEPARTMENT_NAME
120	Treasury
130	Corporate Tax
140	Control And Credit
150	Shareholder Services
160	Benefits
170	Manufacturing
...	
260	Recruiting
270	Payroll

## Verwendung des [not] exists – Operators (Forts.)

- Der **exists**-Operator prüft die Existenz von Zeilen in der Ergebnismenge der Unteranfrage.
- Wenn in der Unteranfrage eine Zeile gefunden wird, dann
  - wird die Bedingung als **true** markiert
  - und die Suche in der inneren Anfrage wird nicht fortgesetzt.
- Wenn in der Unteranfrage keine Zeile gefunden wird, dann liefert die Bedingung **false**.

## Unterabfragen außerhalb der where-Klausel

- Unterabfrage in der **select**-Klausel, hier mehrwertig / unkorreliert:

```
select employee_id, last_name,  
        (case  
          when department_id = any  
            (select department_id  
             from DEPARTMENTS  
             where location_id = 1800)  
          then 'Canada' else 'USA' end) location  
from EMPLOYEES;
```

- Unterabfrage in der **order-by**-Klausel, hier einwertig / korreliert:

```
select employee_id, last_name,  
from EMPLOYEES e  
order by (select department_name  
           from DEPARTMENTS d  
           where e.department_id = d.department_id);
```

### 3.3 Die Semantik des SQL-Kerns in der Relationenalgebra

#### Beispiel-DB-Schema “Warenmarkt”:

KUNDE (KNr, KName, KAdr, Kto)

WARE (WBez, . . .)

LIEFERANT (LName, LAdr)

ANGEBOT (LName → LIEFERANT, Ware → WARE, Preis)

BESTELLUNG (KNr → KUNDE, LName → LIEFERANT,  
Ware → WARE, Menge;  
(LName, Ware) → ANGEBOT)

Zusätzliche Annahmen:

- Nur **not null**-Attribute.
- Die Attributkombination (KName, KAdr) in KUNDE ist eindeutig.



## Typische SQL-Beispielanfragen: Grundform

(a) *Welche Kunden haben ihr Konto überzogen ? (Namen, Adressen)*

<b>select</b> KName, KAdr	[Zielliste]
<b>from</b> KUNDE	[Herkunft]
<b>where</b> Kto < 0	[Qualifikation]
<b>order by</b> KName	[Sortierung]

→ äquivalente Anfrage in der Relationenalgebra:

$$\pi_{k.KName, k.KAdr} (\sigma_{k.Kto < 0} (KUNDE \ k))$$

*Hinweis:* In der Relationenalgebra lassen sich nur Anfragen erklären, die keine Duplikate liefern, da Relationen nur Mengen (aber nicht Multimengen) von Tupeln sind.

## SQL-Anfragen (Forts.): Grundform mit zusammengesetzter Bedingung

(b) *Welche Kunden aus 30159 Hannover haben bei einem Lieferanten mehrere PC2050 bestellt ?*

```
select distinct KName, KAdr
from   KUNDE, BESTELLUNG
where  KAdr like '%30159 Hannover%'
and    KUNDE.KNr = BESTELLUNG.KNr
and    Ware = 'PC2050' and Menge > 1
```

→  $\pi_{k.KName, k.KAdr} (\sigma_{\varphi} (KUNDE \ k \times BESTELLUNG \ b))$

mit  $\varphi \equiv (k.KNr = b.KNr \wedge \psi)$ ,

$\psi \equiv ((\text{like}(k.KAdr, '%30159 Hannover\%'))$   
 $\wedge b.Ware = 'PC2050' \wedge b.Menge > 1)$

$=^{14} \pi_{k.KName, k.KAdr} (\sigma_{\psi} (KUNDE \ k \underset{k.KNr=b.KNr}{\bowtie} BESTELLUNG \ b))$

## SQL-Anfragen (Forts.): Unteranfrage

nochmals (b) *Welche Kunden aus ... haben ... mehrere PC2050 bestellt ?*

```
select KName, KAdr
from KUNDE
where KAdr like '%30159 Hannover%'
and   KNr = any                                oder: KNr in ...
      (select KNr
       from BESTELLUNG
       where Ware = 'PC2050' and Menge > 1 )
```

Das semantische Gegenstück zur Schachtelung in SQL ist in der Relationenalgebra der Semijoin-Operator (oder bei Negation der Antijoin, s.u.)<sup>13</sup>:

$$\begin{aligned} &\rightarrow \pi_{k.KName, k.KAdr}(\sigma_{\text{like}(k.KAdr, \dots)}(\text{KUNDE } k \\ &\quad \bowtie_{k.KNr=b.KNr} \pi_{b.KNr}(\sigma_{(b.Ware='PC2050' \wedge b.Menge > 1)}(\text{BESTELLUNG } b)))) \\ &= {}^{14}\pi_{k.KName, k.KAdr}(\sigma_{\psi}(\text{KUNDE } k \bowtie_{k.KNr=b.KNr} \text{BESTELLUNG } b)) \end{aligned}$$

*Also ist diese geschachtelte SQL-Anfrage äquivalent zur obigen ungeschachtelten SQL-Anfrage.*

<sup>13</sup> ... nicht etwa eine irgendwie geartete Schachtelung innerhalb der Selektion. Selektionen beziehen sich immer nur auf jedes Tupel einzeln.

<sup>14</sup>Die Umformungen basieren auf den Ableitungen und Gesetzen aus Kap. 3.1, hier Proj.(viii) für  $\pi(\dots \bowtie \dots)$  und Sel.(iv)' für  $\sigma(\dots) \bowtie \sigma(\dots)$ .

**SQL-Anfragen** (Forts.): Unteranfrage mit Rückbezug (korrelierte Unteranfrage)

(c) *Stelle eine Liste von Kunden und Lieferanten aller Bestellungen zusammen.*

```
select distinct KName,KAdr, LName,LAdr
from   KUNDE, LIEFERANT
where exists
      (select *
       from   BESTELLUNG
       where KUNDE.KNr = KNr
       and    LIEFERANT.LName = LName)
```

$$\begin{aligned} &\rightarrow {}^{15}\pi_{k.KName,\dots}((KUNDE\ k \times LIEFERANT\ l) \\ &\quad \bowtie_{k.KNr=b.KNr \wedge l.LName=b.LName} BESTELLUNG\ b)) \\ &= \pi_{k.KName,\dots}(KUNDE\ k \bowtie_{k.KNr=b.KNr} (LIEFERANT\ l \bowtie_{l.LName=b.LName} BESTELLUNG\ b)) \end{aligned}$$

*also in SQL entschachtelbar zu .....*

---

<sup>15</sup>Die Korrelationsbedingung gehört an den Semijoin!

## SQL-Anfragen (Forts.)

(c) ... *entschachtelbar zu folgenden äquivalenten SQL-Formulierungen:*

```
select distinct KName,KAdr, LName,LAdr  
from   KUNDE, BESTELLUNG, LIEFERANT  
where KUNDE.KNr = BESTELLUNG.KNr  
and   BESTELLUNG.LName = LIEFERANT.LName  
  
= select distinct KName,KAdr, LName,LAdr  
from   KUNDE natural join BESTELLUNG natural join LIEFERANT
```

Hinweis zur (gedachten!) Ausführung von SQL-Anfragen:  
[Operationale Semantik]

Für jedes Tupel *t* des Produkts der äußeren **from**-Relationen  
wird die äußere **where**-Klausel ausgewertet,  
einschließlich evtl. Unteranfragen (wiederum nach dieser Vorschrift);  
dabei dienen **Rückbezüge** auf äußere Relationen als Rückbezüge auf das Tupel *t*.

## SQL-Anfragen (Forts.): Mehrfache Bezüge auf die gleiche Relation

(d) *Welche Lieferanten bieten mindestens eine Ware an, die auch Miller&Co anbietet? (Namen)*

```
select distinct LName from ANGEBOT ANG
where exists (select * from ANGEBOT
              where ANG.Ware = Ware and LName = 'Miller&Co')
```

→  $\pi_{ang.LName}$

(  $\text{ANGEBOT } ang \bowtie_{ang.Ware=angm.Ware} \sigma_{angm.LName='Miller\&Co'} (\text{ANGEBOT } angm) )$

=  $\pi_{ang.LName} (\sigma_{angm.LName='Miller\&Co'}$

(  $\text{ANGEBOT } ang \bowtie_{ang.Ware=angm.Ware} \text{ANGEBOT } angm ) )$

→ **select distinct** ANG.LName

**from** ANGEBOT ANG, ANGEBOT ANGM

**where** ANG.Ware = ANGM.Ware **and** ANGM.LName = 'Miller&Co'

## SQL-Anfragen (Forts.): not exists-Unteranfrage

(e) *Welche Kunden haben keine Waren bestellt ?*

```
select KName, KAdr
from KUNDE
where not exists
      (select *
       from BESTELLUNG
       where KUNDE.KNr = KNr )
```

→  $\pi_{k.KName}(\text{KUNDE } k \bowtie_{k.KNr=b.KNr} \text{BESTELLUNG } b)$

*select...from...where...-Anfragen ohne Unteranfragen lassen sich mit den Grundoperationen Projektion, Selektion und Produkt erklären. Die obige Anfrage kann also in SQL nicht entschachtelt werden<sup>16</sup>, weil der Anti-Semijoin die Differenz als weitere Grundoperation enthält.*

<sup>16</sup>— ausser man würde Outer Joins und Tests auf Nullwerte verwenden (*bleibt zur praktischen Übung*); wie oft in der relationalen Datenbanktheorie seien hier die besonderen Effekte von Nullwerten vernachlässigt.

## SQL-Anfragen (Forts.): Verwandtschaft **all** – **not exists**

(f) *Welche Lieferanten bieten welche Waren billiger (d. h. nicht teurer) an als alle Lieferanten ?*

```
select LName, Ware
from   ANGEBOT ANG
where  Preis <= all
      (select Preis from ANGEBOT VGL where VGL.Ware = ANG.Ware)
```

→ Semantik durch äquivalente Umformulierung:

```
select LName, Ware
from   ANGEBOT ANG
where  not exists
      (select * from ANGEBOT VGL
       where VGL.Ware = ANG.Ware and ANG.Preis > VGL.Preis)
```

→  $\pi_{ang, \dots}((ANGEBOT \text{ ang}) \bowtie_{ang.Ware=vgl.Ware \wedge ang.Preis > vgl.Preis} (ANGEBOT \text{ vgl}))$



## SQL-Anfragen (Forts.): Verwandtschaft **any** - **exists**

(g) *Welche Lieferanten bieten welche Waren echt billiger an als irgendein Lieferant ?*

```
select LName, Ware
from   ANGEBOT ANG
where  Preis < any
      (SELECT Preis from ANGEBOT where Ware = ANG.Ware)
```

```
→ select LName, Ware
   from   ANGEBOT ANG
   where  exists
        (select * from ANGEBOT VGL
         where VGL.Ware = ANG.Ware and ANG.Preis < VGL.Preis)
```

```
→ select LName, Ware
   from   ANGEBOT ANG, ANGEBOT VGL
   where  VGL.Ware = ANG.Ware and ANG.Preis < VGL.Preis
```

## SQL-Anfragen (Forts.): Teilmengenvergleich ...

(h) *Welche Lieferanten liefern mind. alle Waren, die Miller&Co liefert?*

Sei  $W[x] \equiv \pi_{\text{Ware}}(\sigma_{\text{LName}=x}(\text{ANGEBOT}))$  .

— *Welche Waren liefert der Lieferant mit Name x ?*

Gesucht ist:  $\pi_{\text{LName}}(\sigma_{\text{''}W[x/\text{'Miller\&Co'}] \subseteq W[x/\text{ang.LName}] \text{'}}(\text{ANGEBOT ang}))$

Wir verwenden hier einen *parametrisierten* Term der Relationenalgebra, den Term  $W$  mit formalem Parameter  $x$ ;  $W[x/\text{'Smith'}]$  steht dann für den Term, in dem  $x$  durch den aktuellen Parameter 'Smith' substituiert wird. *(Diese Notation dient nur zur Motivation; sie ist in der Relationenalgebra nicht erlaubt.)*

Es gilt:  $W[x/\text{'Miller\&Co'}] \subseteq W[x/\text{ang.LName}]$

$$\Leftrightarrow (W[x/\text{'Miller\&Co'}] - W[x/\text{ang.LName}]) = \emptyset$$

Dies führt uns zu folgender SQL-Anfrage:

**SQL-Anfragen** (Forts.): Teilmengenvergleich erfordert doppelte Schachtelung.

```
→ select distinct LName
   from  ANGEBOT ANG
   where not exists
         (select Ware from ANGEBOT
          where LName = 'Miller&Co'
          and   Ware not in
                (select Ware from ANGEBOT
                 where LName = ANG.LName))
```

Alternativ kann man direkt eine Mengenoperation (−) verwenden:

```
select distinct LName
   from  ANGEBOT ANG
   where not exists
         ((select Ware from ANGEBOT
          where LName = 'Miller&Co')
          minus (select Ware from ANGEBOT
                 where LName = ANG.LName))
```

## SQL-Anfragen (Forts.): Operatoren aus der Relationenalgebra

(i) *Welche Waren sind bestellt oder werden angeboten?*

(select Ware from BESTELLUNG)

**union**

(select Ware from ANGEBOT)

→  $\pi_{\text{Ware}}(\text{BESTELLUNG}) \cup \pi_{\text{Ware}}(\text{ANGEBOT})$

Die Anfrage wäre ohne **union** nicht in SQL ausdrückbar.

Weitere relationenalgebraische Operationen als SQL-Sprachkonstrukte<sup>17</sup>:

- **intersect**
- **except** (synonym zu **minus**)
- **join**, auch **natural join**, **[left] outer join** u.a.

Diese sind allerdings auch mit anderen Anfragemöglichkeiten von SQL ausdrückbar.

---

<sup>17</sup>Leider gibt es in SQL eine syntaktische Ungleichbehandlung: Während die **join**-Operatoren auch Basisrelationen in der **from**-Klausel verknüpfen dürfen, z. B. **select ... from (R join S on R.A=S.B) ...**, dürfen **union**, **minus** und **intersect** nur ganze *Anfragen* wie im Beispiel oben verknüpfen.