

# Kapitel 4

## SQL-DML und -DDL

## 4.1 Änderungsoperationen im Relationenmodell

Eine Änderungsoperation  $\omega$  (mit weiteren Parametern, z.B. einem Tupel) überführt einen DB-Zustand  $\zeta_{\text{old}}$  in einen DB-Zustand  $\zeta_{\text{new}}$ . Ein DB-Zustand besteht aus den aktuellen Ausprägungen  $\zeta(R)$  aller im DB-Schema angegebenen Relationen  $R$ .

### Definition der Grundoperationen zur Änderung von Relationen

Gegeben sei ein relationales DB-Schema.  $R(A_1: D_1, \dots, A_n: D_n)$  sei ein Relationenschema daraus, und  $t, t_1, t_2$  seien Tupel aus  $|D_1| \times \dots \times |D_n|$ .

1.  $\omega \equiv \text{insert}(R, t)$  — *Einfügen eines Tupels:*

$$\zeta_{\text{new}}(R) := \begin{cases} \zeta_{\text{old}}(R) \cup \{t\}, & \text{falls die Nullwert-, Eindeutigkeits-,} \\ & \text{Schlüssel- und Fremdschlüssel-} \\ & \text{bedingungen von } R \text{ erfüllt bleiben} \\ \zeta_{\text{old}}(R), & \text{sonst (mit Warnung)} \end{cases}$$

$\zeta_{\text{new}}(R') := \zeta_{\text{old}}(R')$  für  $R' \neq R$  (d.h. andere Relationen bleiben unverändert)

## Änderungs-Grundoperationen (Forts.)

2.  $\omega \equiv \text{delete}(\mathbf{R}, \mathbf{t})$  — *Löschen eines Tupels:*

$$\zeta_{\text{new}}(\mathbf{R}) := \begin{cases} \zeta_{\text{old}}(\mathbf{R}) - \{\mathbf{t}\}, & \text{falls die Fremdschlüsselbedingun-} \\ & \text{gen bzgl. } (\rightarrow) \mathbf{R} \text{ erfüllt bleiben} \\ \zeta_{\text{old}}(\mathbf{R}), & \text{sonst (mit Warnung}^1) \end{cases}$$

$$\zeta_{\text{new}}(\mathbf{R}') := \zeta_{\text{old}}(\mathbf{R}') \text{ für } \mathbf{R}' \neq \mathbf{R}^1$$

3.  $\omega \equiv \text{update}(\mathbf{R}, \mathbf{t}_1, \mathbf{t}_2)$  — *Ersetzen:*

$$\text{analog mit } \zeta_{\text{new}}(\mathbf{R}) := \dots \zeta_{\text{old}}(\mathbf{R}) - \{\mathbf{t}_1\} \cup \{\mathbf{t}_2\} \dots$$

(Jedoch sollten theoretisch keine Änderungen der Schlüsselattribute erlaubt sein!)

DB-Sprachen wie SQL bieten diese Operationen an, allerdings nicht für einzelne Tupel, sondern für eine Menge von Tupeln, die typischerweise Ergebnis einer (eingebauten) Anfrage ist.

---

<sup>1</sup>falls keine aktive Einhaltung der Fremdschlüsselbedingung durch kaskadierendes Löschen – in SQL mit **on delete cascade** – verlangt ist

## 4.2 Updates in SQL

### Data Manipulation Language (DML)

- Eine **DML-Anweisung** wird ausgeführt, wenn man
  - eine neue Zeile in eine Tabelle einfügt (**insert**),
  - bestehende Zeilen in einer Tabelle modifiziert (**update**)
  - oder bestehende Zeilen aus einer Tabelle löscht (**delete**).
- Eine **Transaktion** besteht aus einer Folge von DML-Anweisungen, die eine logische Arbeitseinheit bilden.

## Einfügen einer neuen Zeile in eine Tabelle

neue Zeile			
70	Public Relations	100	1700

Tabelle 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

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
70	Public Relations	100	1700

## Syntax der insert-Anweisung

- Die einfache **insert**-Anweisung fügt eine neue Zeile in eine Tabelle ein:

```
insert into Tabelle [(Spalte[, Spalte...])]  
values      (Wert[, Wert...]);
```

- String- und Datumswerte sind wieder in einfache Anführungszeichen zu setzen.

## Verwendung der insert-Anweisung

- Um eine neue Zeile einzufügen, die Werte für jede Spalte enthält, sind die Werte in der Reihenfolge aufzulisten, in der die Spalten in der Tabellendefinition (**create table**) standen.

```
insert into DEPARTMENTS  
values      (70, 'Public Relations', 100, 1700);  
1 row inserted.
```

## Verwendung der insert-Anweisung (Forts.)

- Optional können die Spalten in der **insert**-Klausel explizit angegeben werden, und man folgt bei den Werten dieser Reihenfolge.
- Für weggelassene Spalten wird der Defaultwert (falls definiert) oder der Nullwert (falls erlaubt) verwendet.

**insert into EMPLOYEES**

```
(employee_id, manager_id, first_name, last_name,  
email, phone_number, hire_date, job_id,  
salary, department_id)
```

```
values (113, 205, 'Louis', 'Popp',  
null, '515.124.4567', sysdate, 'AC_ACCOUNT',  
5000*1.19, 100);
```

1 row inserted.

Statt **sysdate** dürfte man auch ein konkretes Datum wie z.B. **to\_date('MAY 28, 2013', 'MON DD, YYYY')** angeben.

## Syntax/Verwendung der insert-Anweisung (Forts.)

- Die **insert**-Anweisung mit einer Unteranfrage (statt der **values**-Klausel) erlaubt das Kopieren von Zeilen aus anderen Tabellen bzw. aus einem Anfrageergebnis; z.B.:

```
insert into SALES_REPS (id, name, salary, commission_pct)
select employee_id, last_name, salary, commission_pct
from EMPLOYEES
where job_id like '%REP%';
4 rows inserted.
```

- Anzahl und Typen der Spalten in der Unteranfrage müssen zu denen in der **insert**-Klausel (implizit oder explizit) passen; die Namen dürfen abweichen.




## Aktualisieren von Daten einer Tabelle

### EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_F
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

60→30



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSIO
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

## Syntax der update-Anweisung

- Die **update**-Anweisung verändert bestimmte Spalten in den mit einer **where**-Klausel ausgewählten Zeilen oder in allen Zeilen einer Tabelle:

**update** *Tabelle*

**set**     *Spalte1 = Spaltenausdruck1 [, Spalte2 = Spaltenausdruck2, ...]*  
**[where** *Bedingung*];

- In der **set**-Klausel können Ausdrücke wie in einer **select**-Liste stehen.

## Verwendung der update-Anweisung

```
update EMPLOYEES
set    department_id = 30
where  department_id = 60;
3 rows updated.
```

```
update COPY_EMP
set department_id = 999;

22 rows updated.
```

## Verwendung der update-Anweisung (Forts.)

Aktualisieren von zwei Spalten, mit Unteranfragen:

Beispiel: *Aktualisiere den Beruf und das Gehalt des Angestellten 114 auf die Werte des Angestellten 205.*

```
update EMPLOYEES
set      (job_id,salary) =
          (select job_id,salary from EMPLOYEES
           where employee_id =205),
where employee_id = 114;
1 row updated.
```

oder:

```
update EMPLOYEES
set      job_id = (select job_id from EMPLOYEES
                   where employee_id =205),
        salary = (select salary from EMPLOYEES
                   where employee_id = 205)
where employee_id = 114;
```

## Verwendung der update-Anweisung (Forts.)

Aktualisieren basierend auf einer anderen Tabelle/Instanz, mit korrelierter Unteranfrage:

Beispiel: *Übernahme Manager-IDs aus DEPARTMENTS nach EMPLOYEES.*

```
update EMPLOYEES e
set     manager_id =
        (select manager_id from DEPARTMENTS
         where department_id = e.department_id);
```

Beispiel: *Ergänze fehlende Gehaltswerte durch Durchschnitt pro Abtlg.*

```
update EMPLOYEES e
set     salary =
        (select avg(salary) from EMPLOYEES d
         where d.department_id = e.department_id);
where  salary is null;
```

## Löschen von Zeilen aus einer Tabelle

### DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
<del>100</del>	<del>Finance</del>		
50	Shipping	124	1500
60	IT	103	1400

**delete from DEPARTMENTS**  
**where department\_name = 'Finance';**  
**1 row deleted.**

## Syntax/Verwendungen der delete-Anweisung

Die **delete**-Anweisung löscht [ausgewählte] Zeilen einer Tabelle:

```
delete [from] Tabelle  
[where Bedingung];
```

```
delete from COPY_EMP;  
22 rows deleted.
```

```
delete from EMPLOYEES  
where department_id =  
      (select department_id from DEPARTMENTS  
       where department_name like '%Public%');  
1 row deleted.
```

```
delete EMPLOYEES e  
where salary >  
      (select salary from EMPLOYEES m  
       where m.employee_id = e.manager_id);  
2 rows deleted.
```

## Ausführen von DML-Anweisungen

Eine DML-Anweisung wertet Berechnungen (für Update-Werte) bzw. die **where**-Bedingung für die Auswahl der zu ändernden/löschenden Tupel im DB-Zustand vor der Anweisung aus. Änderungen von (im Rahmen der Ausführung zufällig) zuerst bearbeiteten Tupeln wirken sich also nicht auf Änderungen von später bearbeiteten Tupeln aus. — *Beispiele:*

```
update ANGEBOT A1  
  set Preis = 2* (select max(Preis)  
                    from ANGEBOT A2 where A2.Ware=A1.Ware)  
  where upper(Ware) like '%GOLD%'
```

⇒ Die Preise schaukeln sich während der Ausführung der Anweisung nicht hoch!

```
delete from ANGEBOT A1  
  where exists (select * from ANGEBOT A2  
                where A2.Ware=A1.Ware and A2.LName<>A1.LName)
```

... sucht alle Angebote, zu denen es ein zweites Angebot mit gleicher Ware gibt, und löscht diese alle

⇒ lässt nur (vorher schon) singuläre Angebote übrig

⇒ falls vorher alle Waren mind. doppelt angeboten wurden, wird ANGEBOT geleert

## Datenbank-Transaktionen

Eine **Datenbank-Transaktion** besteht aus:

- einer Folge von DML-Anweisungen, die eine logisch konsistente Änderung an den Daten bewirken sollen,
- oder einer einzigen DDL-Anweisung

Eine Transaktion endet mit einem der folgenden Ereignisse:

- Eine **commit**-Anweisung zum Freigeben aller Änderungen aus der Transaktion an die Datenbank wird ausgeführt.
- Eine **rollback**-Anweisung zum Zurücksetzen aller Änderungen aus der Transaktion wird ausgeführt (quasi ein Undo).
- Eine DDL-Anweisung wird ausgeführt (automatisches Commit).
- Der Nutzer beendet einen SQL-Client wie SQL\*Plus (autom. Commit).
- Das System stürzt ab (automatisches Rollback).

Die nächste Transaktion beginnt, sobald die nächste DML/DDL-Anweisung ausgeführt wird.



## Commit von Transaktionen

1. (Zusammenhängende) Änderungen vornehmen:

- “isoliert”, d.h. nicht sichtbar für andere Nutzer –

```
insert into DEPARTMENTS
```

```
values      (290, 'Corporate Tax', null, 1700);
```

```
1 row inserted.
```

```
update EMPLOYEES set department_id = 290
```

```
where employee_id=99999;
```

```
1 row updated.
```

```
update DEPARTMENTS set manager_id = 99999
```

```
where department_id = 290;
```

```
1 row updated.
```

2. Änderungen freigeben:

```
commit;
```

```
Commit complete.
```

## Rollback von Transaktionen

```
delete from TEST;  
25000 rows deleted.  
select count(*)||' rows' from TEST;  
0 rows
```

```
rollback;  
Rollback complete.
```

```
delete from TEST where id = 100;  
1 row deleted.  
select count(*)||' rows' from TEST;  
24999 rows
```

```
commit;  
Commit complete.
```

## Zustand der Daten vor einem Commit oder Rollback

- Der vorherige Zustand der Daten kann wiederhergestellt werden.
- Der aktuelle Nutzer kann die Ergebnisse seiner DML-Anweisungen nachprüfen, indem er die **select**-Anweisung verwendet.
- Andere Nutzer können die Ergebnisse der DML-Anweisungen des aktuellen Nutzers *nicht* sehen.
- Die von Änderungen betroffenen Zeilen sind *gesperrt*; andere Nutzer können die Daten der betroffenen Zeilen nicht ändern, sondern müssen ggf. warten.

## Vorteile von Transaktionen

Mit Transaktionen kann man:

- die Konsistenz von Daten durch Gruppieren von Änderungen sicherstellen
- Änderungen an Daten im voraus betrachten, bevor sie permanent gemacht werden

## **Zustand der Daten nach dem Commit**

- Änderungen an Daten werden permanent in die Datenbank übernommen.
- Der vorherige Zustand der Daten ist dauerhaft verloren.
- Alle Nutzer können die Ergebnisse sehen.
- Sperren auf betroffenen Zeilen werden freigegeben; diese Zeilen sind für andere Nutzer verfügbar und können von ihnen geändert werden.
- Alle Sicherungspunkte (s.u.) werden gelöscht.

## **Zustand der Daten nach dem Rollback**

- Alle noch nicht freigegebenen Änderungen an Daten werden zurückgenommen.
- Der vorherige Zustand der Daten (zu dem Zeitpunkt, als die Transaktion begonnen hat) wird wiederhergestellt.
- Sperren auf betroffenen Zeilen werden freigegeben, Sicherungspunkte gelöscht.

## Verwendung von Sicherungspunkten in einer Transaktion

- Mit Hilfe der **savepoint**-Anweisung kann in der aktuellen Transaktion eine Markierung, ein sogenannter **Sicherungspunkt**, gesetzt werden.
- Änderungen müssen nicht bis zum Transaktionsbeginn, sondern können bis zu solch einer dieser Markierung rückgängig gemacht werden; mit der **rollback-to-savepoint**-Anweisung.

**update ...**

**savepoint** update\_done;

Savepoint created.

**insert ...**

**rollback to** update\_done;

Rollback complete.

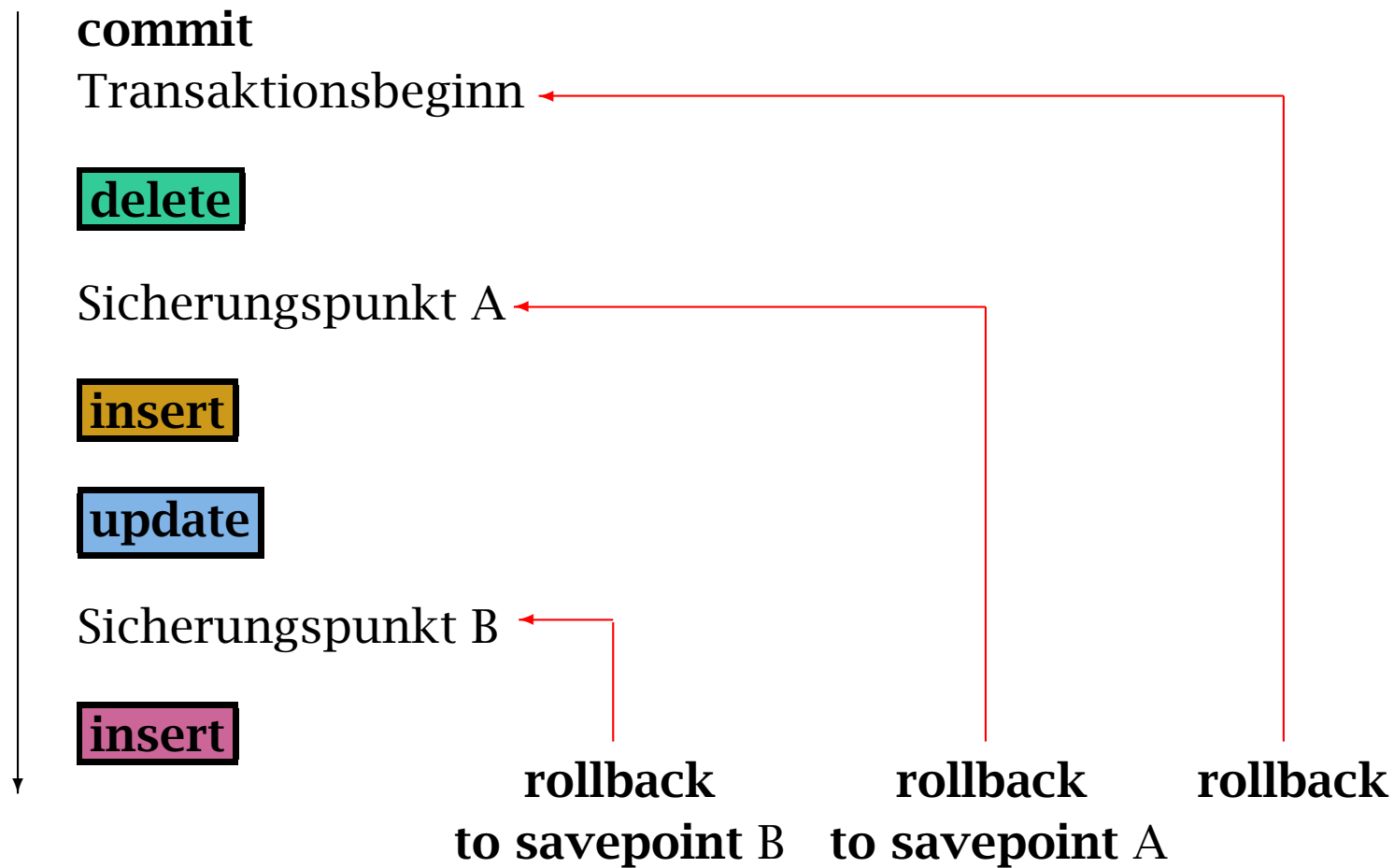
**insert ...**

**commit;**

Commit complete.

## Verwendung von Sicherungspunkten (Forts.)

*Zeit*



## 4.3 Weiteres zu Tabellendefinitionen in SQL

### Erzeugen einer Tabelle

- Beispielanweisung:

```
create table DEPT
  (deptno      number(2) primary key,
   dname       varchar2(14) not null,
   loc         varchar2(13),
   create_date date default sysdate);
```

Table created.

- Beschreibung der erzeugten Tabelle mit: **desc[ribe]** DEPT

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE

## Datentypen in Oracle-SQL

Datentyp	Beschreibung
<b>varchar2(<i>l</i>)</b>	Zeichenkette variabler Länge <i>l</i> (bis 4000 Bytes)
<b>char(<i>l</i>)</b>	Zeichenkette fester Länge <i>l</i> (bis 2000 Bytes)
<b>number(<i>p</i> [, <i>s</i>])</b>	numerische Daten mit max. <i>p</i> Stellen ( $p \leq 38$ ) bis zur <i>s</i> -ten Dezimalstelle ( $-84 \leq s \leq 127$ )
<b>number</b>	numerische Daten variablen Aufbaus
<b>date</b>	Datums- und Zeitwerte
<b>raw(<i>l</i>)</b>	Binärdaten variabler Länge <i>l</i> (bis 2000 Bytes)
<b>blob</b>	“binary large object”; große Binärdaten (früher bis 4 GB)
<b>bfile</b>	Zeiger auf Datei, die große Binärdaten speichert
<b>clob</b>	“character large object”; große Zeichendaten (früher bis 4 GB)
<b>rowid</b>	Adressen von Tabellenzeilen



## Default-Option

- Man kann einen Defaultwert, d.h. einen voreingestellten Wert, für Einfügungen in eine Spalte angeben.

```
create table HIRE_DATES
  (id number(8),
   hire_date date default sysdate);
Table created.
```

- Erlaubt sind Konstanten, SQL-Funktionen und Spaltenausdrücke.
- Der Defaultwert muss zum Datentyp der Spalte passen.

## Löschen einer Tabelle

z.B. **drop table** EMP80;  
Table dropped.

- Alle Daten und Strukturen der Tabelle werden gelöscht.
- Die laufende Transaktion wird committed, so dass die **drop-table**-Anweisung nicht rückgängig gemacht werden kann.
- Alle Constraints und Indexe auf der Tabelle werden gelöscht.

## Löschen einer Sicht (siehe Abschnitt 4.4)

z.B. **drop view** EMPVU80;  
View dropped.

- Da Sichten unterliegende Daten aus Datenbanktabellen nur virtuell darstellen, können Sichten gelöscht werden ohne Daten zu verlieren.

## Ändern von Tabellendefinitionen

Mit der **alter-table**-Anweisung kann man (soweit plausibel) Spaltendefinitionen hinzufügen, modifizieren oder löschen.

**alter table** *Tabelle*

**add** (*Spalte1 Datentyp1* [**default** *Spaltenausdruck*]  
[, *Spalte2 Datentyp2 ...*]);

**alter table** *Tabelle*

**modify** (*Spalte1 Datentyp1* [**default** *Spaltenausdruck*]  
[, *Spalte Datentyp2 ...*]);

**alter table** *Tabelle*

**drop** (*Spalte1*[, *Spalte2 ...*]);      *oder*      **drop column** *Spalte*

## Hinzufügen einer Spalte

- Beispiel für die **add**-Klausel der **alter-table**-Anweisung:

```
alter table EMP80  
add (job_id varchar2(9));  
Table altered.
```

- Die neue Spalte wird hinten angehängt und mit Defaultwerten (falls angegeben), sonst mit Nullwerten gefüllt.

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
145	Russell	14000	01-OCT-96	
146	Partners	13500	05-JAN-97	
147	Errazuriz	12000	10-MAR-97	
148	Cambrault	11000	15-OCT-99	
149	Zlotkey	10500	29-JAN-00	

...

## Modifizieren einer Spalte

- Beispiel für die **modify**-Klausel der **alter-table**-Anweisung:

```
alter table EMP80  
modify (last_name varchar2(35));  
Table altered.
```

- Ändern kann man den Defaultwert einer Spalte, ihren Datentyp (sofern die Spalte leer ist) und ihre Länge (sofern keine zu langen Werte existieren).
- Das Ändern des Defaultwerts hat nur Einfluss auf darauffolgende Einfügungen in die Tabelle.

## Löschen einer Spalte

- Beispiel für die **drop**-Klausel der **alter-table**-Anweisung:

```
alter table EMP80  
drop column job_id;  
Table altered.
```

EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
145	Russell	14000	01-OCT-96
146	Partners	13500	05-JAN-97
147	Errazuriz	12000	10-MAR-97
148	Cambrault	11000	15-OCT-99
149	Zlotkey	10500	29-JAN-00

- Beachte: Auch mit Nicht-Nullwerten gefüllte Spalten können so aus Versehen (da ohne Rückfrage) gelöscht werden.
- Alternatives Vorgehen:

```
alter table EMP80 set unused column job_id;  
und später:  
alter table EMP80 drop unused columns;
```

## Erzeugen einer Tabelle mit Hilfe einer Unteranfrage

- Um eine neue Tabelle zu erstellen und diese gleich mit Inhalt zu füllen, kombiniert man die **create-table**-Anweisung mit einer **as**-Unter-anfrage-Option.

```
create table Tabelle [(Spalte1, Spalte2, ...)]  
    as Unteranfrage;
```

- Die Datentypen/Längen und **not null**-Constraints werden aus der Unteranfrage übernommen. Ohne Angabe von Spaltennamen werden auch die Spaltennamen übernommen; mit deren Angabe müssen die Spaltenanzahlen übereinstimmen.

## Erzeugen einer Tabelle mit Hilfe einer Unteranfrage (Forts.)

**create table** EMP80

**as** select employee\_id, last\_name,  
          salary\*12 ANNSAL, hire\_date  
**from** EMPLOYEES  
**where** department\_id = 80;

Table created.

**describe** EMP80

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE



## Definieren von Integritätsbedingungen (Constraints)

- Syntax:

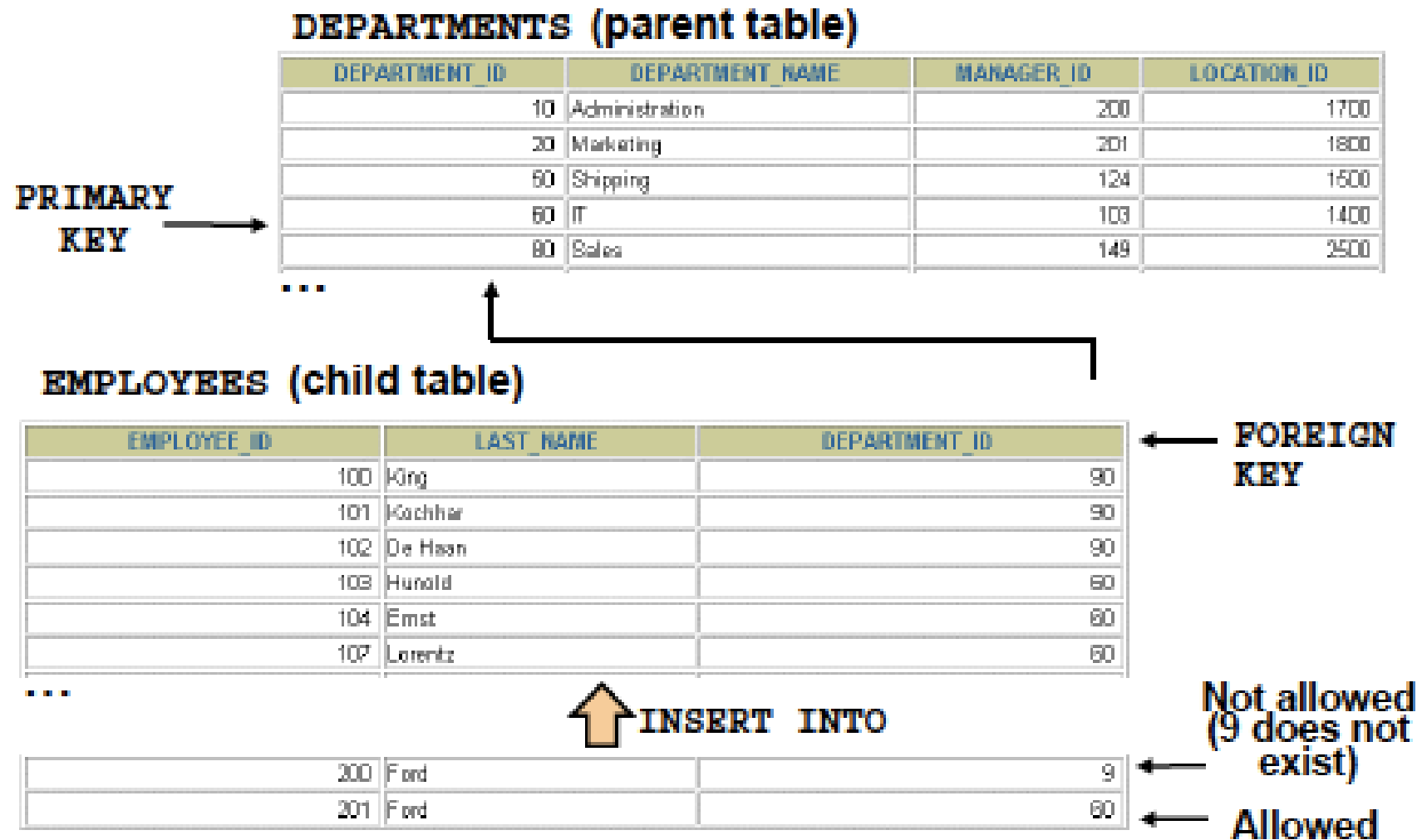
```
create table Tabelle  
    (Spalte Datentyp [default Spaltenausdruck]  
        [Spalten-Constraint],  
    ...,  
    [Tabellen-Constraint] [, ...]);
```

- Spalten-Constraint = Constraint auf einer Spalte  
[**constraint** *Constraint-Name*] *Constraint-Typ*
- Tabellen-Constraint = Constraint über mehrere Spalten)  
[**constraint** *Constraint-Name*] *Constraint-Typ* (*Spalte1*,*Spalte2*,...)
- Constraint-Typen:
  - **not null** (nur auf Spaltenebene)
  - **unique**, **primary key**
  - **foreign key**
  - **check** (s.u.)

## Richtlinien für Constraints

- Constraints erzwingen Regeln für alle Zeilen einer Tabelle.
- Constraints können z.B. Manipulationen von Zeilen verhindern, wenn Abhängigkeiten zu anderen Zeilen der gleichen oder einer anderen Tabelle bestehen.
- Man kann Constraints selbst benennen oder der Oracle-Server generiert einen Namen im SYS\_Cn-Format.
- Constraints können zu einem der folgenden Zeitpunkte definiert werden:
  - zur selben Zeit, zu der auch die Tabelle erzeugt wird  
(in **create table**-Anweisung)
  - wenn die Tabellendefinition geändert wird  
(in **alter table**-Anweisung)
- Constraints können wie Tabellendefinitionen im Data Dictionary eingesehen werden (s.u.).

## zur Erinnerung: foreign key – Constraints



## **zur Erinnerung: foreign key – Constraints** (Forts.)

Beispiel (hier als Tabellen-Constraint auf einer Spalte):

```
create table EMPLOYEES  
  (employee_id number(6) primary key,  
   last_name varchar2(25) not null,  
   email varchar2(25),  
   salary number(8,2),  
   commission_pct number(2,2),  
   hire_date date not null,  
   ...  
   department_id number(4),  
   constraint emp_dept_fk foreign key (department_id)  
     references DEPARTMENTS(department_id),  
   constraint emp_email_uq unique(email) );
```

## foreign key – Constraint: Schlüsselwörter

- **foreign key** (nur auf Tabellenebene): definiert die Fremdschlüssel-Spalte(nkombination) in dieser Tabelle, also der “Kindtabelle”
- **references**: identifiziert die “Elterntabelle” und ggf. explizit die Bezugsspalte(nkombination) darin; der Primärschlüssel ist voreingestellt, es sind aber auch **unique**-Spalte(nkombinationen) erlaubt
- **on delete cascade** – Option: wenn eine Zeile in der Elterntabelle gelöscht wird, werden die abhängigen Zeilen in der Kindtabelle auch gelöscht
- **on delete set null** – Option: setzt in diesem Fall die Fremdschlüssel-Werte in den abhängigen Zeilen auf **null**
- Ohne diese Optionen ist das Löschen von Zeilen in der Elterntabelle nicht erlaubt (Constraintverletzung), solange in der Kindtabelle abhängige Zeilen existieren; s. folgendes Beispiel.

## Verletzungen von foreign key-Constraints in der Elterntabelle

Man kann keine Zeile löschen, die einen Primärschlüssel-Wert enthält, der als Fremdschlüssel-Wert in der Kindtabelle genutzt wird.

```
delete from DEPARTMENTS  
where department_id = 60;
```

→ **delete from DEPARTMENTS**

\*

ERROR at line 1:

```
ORA-02292: integrity constraint (HR.EMP_DEPT_FK) violated  
- child record found
```

## Verletzungen von foreign key-Constraints in der Kindtabelle

Ebenso darf kein Fremdschlüssel-Wert per **update** oder **insert** in der Kindtabelle gesetzt werden, der in der Elterntabelle nicht existiert.

```
update EMPLOYEES  
set    department_id = 55;  
where  department_id = 110;
```

→ **update** EMPLOYEES

\*

ERROR at line 1:

```
ORA-02291: integrity constraint (HR.EMP_DEPT_FK) violated  
- parent key not found
```

Begründung: Es gibt keine Abteilung mit der ID 55.

## check-Constraints

- Ein **check**-Constraint definiert eine Bedingung (syntaktisch fast wie in **where**-Klausel), die von jeder Zeile erfüllt werden muss.
- Die folgenden Ausdrücke sind jedoch nicht erlaubt:
  - Bezüge auf Pseudospalten wie **rownum**
  - **sysdate**-, **uid**-, **user**- oder **userenv**-Funktionen
  - Bezüge auf andere Zeilen der gleichen oder einer anderen Tabelle (deshalb auch keine Tabellennamen oder -alias als Präfixe) !!!
- Beispiele:

```
...,
salary number(2)
    constraint emp_salary_min check (salary > 0),
...,
constraint no_self_mgmt
    check (manager_id is null or manager_id<>employee_id),
...
```



## Ändern von Constraints

Die **alter-table**-Anweisung kann genutzt werden, um:

- beim Hinzufügen einer Spalte (**add**-Klausel) auch zugehörige Spalten-Constraints anzugeben,
- beim Löschen einer Spalte (**drop**-Klausel) auch zugehörige Spalten-Constraints mitzulöschen,
- als Modifikation einer Spaltendefinition (**modify**-Klausel) zugehörige Spalten-Constraints hinzufügen oder **null** ↔ **not null** umzuschalten,
- mit Varianten der **add**/**drop**-Klauseln Tabellen-Constraints hinzuzufügen oder zu löschen,
- Spalten-Constraints wie Tabellen-Constraints zu löschen
- und mit den **disable**/**enable**-Klauseln Constraints zu deaktivieren oder zu aktivieren.

## Ändern von Constraints (Forts.)

```
alter table EMPLOYEES  
modify hire_date unique;
```

```
alter table EMPLOYEES  
add constraint emp_mgr_fk  
foreign key(manager_id) references EMPLOYEES(employee_id);
```

```
alter table EMPLOYEES  
drop constraint emp_dept_fk;
```

```
alter table EMPLOYEES  
add constraint emp_dept_fk2  
foreign key (department_id)  
references DEPARTMENTS on delete cascade;
```

```
alter table EMPLOYEES  
drop unique(hire_date) ;
```

## Ändern von Constraints (Forts.)

- In einer Fremdschlüssel → Primärschlüssel-Situation lässt sich das **primary key**-Constraint nur löschen, wenn das zugehörige **foreign key**-Constraint mitgelöscht wird; dazu gibt es eine **cascade**-Option. (Ebenso bei **unique/foreign key**.)

```
alter table DEPARTMENTS  
drop primary key cascade;
```

- (De-)Aktivieren von Constraints:

```
alter table EMPLOYEES  
disable constraint emp_mgr_fk;  
  
alter table EMPLOYEES  
enable constraint emp_mgr_fk;
```

Voreingestellt ist, dass das Constraint beim Aktivieren auf der Tabelle validiert wird.

- Übrigens: Sobald man ein **unique**- oder **primary key**-Constraint erstellt oder aktiviert, wird automatisch ein “**unique index**” angelegt (vgl. Kap. 7).

## Löschen einer Spalte und damit zusammenhängender Constraints

```
alter table EMP80  
drop column employee_id  
cascade constraints;  
Table altered.
```

- Durch die **cascade constraints**-Option beim Löschen einer Spalte (z.B. einer Primärschlüssel-Spalte) werden alle Fremdschlüsselbedingungen, die (u.a.) diese Spalte referenzieren, und alle Tabellen-Constraints, die (u.a.) auf dieser Spalte definiert sind, mitgelöscht.

## 4.4 Sichten

- **Definition einer Sicht:** **Relationenschema** ( $\hat{=}$  externes DB-Schema) + **Anfrage** ( $\hat{=}$  Abbildung externes  $\rightarrow$  konzeptionelles Schema)
- *Beispiel in SQL:* Bestellungen des Kunden Smith

```
create view  MEINE_BESTELLUNGEN(Artikel,Lieferant,Anzahl)
as select  Ware, LName, Menge
from  BESTELLUNG
where KName='Smith'
```

- Eine gewöhnliche Sicht entspricht einer virtuellen Relation.
- *Benutzung:* wie echte Relation (mit Einschränkungen)
- *Implementierung von Anfragen (bei Übersetzung in die erweiterte Relationenalgebra):* durch Einsetzen des **Definitionsterms** anstelle der **Sicht** in den Anfrageterm (“query modification”) und anschließende Optimierung des Terms, d.h. durch Übersetzung in Anfragen auf Basisrelationen. Es findet keine Materialisierung der Sichten statt!<sup>2</sup>

---

<sup>2</sup>soweit nicht aus Optimierungsgründen erforderlich

## Sichten (Forts.)

- *Beispielanfrage:*

```
select sum(Anzahl) Gesamtanzahl
from MEINE_BESTELLUNGEN
where Artikel like 'PC%'
```

$$\begin{aligned} &\rightarrow \Gamma_{\emptyset} \# \text{sum(Anzahl) Gesamtanzahl } (\sigma_{\text{Artikel like 'PC\%'}} (\text{MEINE\_BESTELLUNGEN})) \\ &= \Gamma_{\emptyset} \# \text{sum(Anzahl) Gesamtanzahl } (\sigma_{\text{Artikel like 'PC\%'}} ( \\ &\quad \pi_{\text{Ware Artikel, LName Lieferant, Menge Anzahl}} (\sigma_{\text{KName='Smith'}} (\text{BESTELLUNG})))) \\ &= \Gamma_{\emptyset} \# \text{sum(Menge) Gesamtanzahl } (\sigma_{\text{Ware like 'PC\%' } \wedge \text{KName='Smith'}} (\text{BESTELLUNG})) \end{aligned}$$

- *Vorteile:*

- logische Datenunabhängigkeit
- Vereinfachung von komplexen Anfragen
- Beschränkung von Zugriffen = Datenschutz (*Abschnitt 4.6*)

- *problematisch:* Änderung von Sichten (*s.u.*)

## Sichten (Forts.)

- *Syntax:*

```
create [or replace] [force|noforce] view Sichtennamen  
    [(Spaltenalias [, Spaltenalias] ...)]
```

```
as Unteranfrage
```

```
[with check option [constraint Constraintname]]
```

-- S.4.51

```
[with read only [constraint Constraintname]];
```

-- S.4.51

- Die Reihenfolge der Spaltenalias in der **create-view**-Klausel muss zu den Ergebnisspalten der Unteranfrage passen.
- Die Unteranfrage kann eine komplexe **select**-Syntax beinhalten, z.B.:

```
create or replace view DEPT_SAL_VU (name, minsal, maxsal, avgsal)
```

```
as select d.department_name, min(e.salary), max(e.salary), avg(e.salary)  
    from EMPLOYEES e join DEPARTMENTS d  
        on (e.department_id = d.department_id)  
    group by d.department_name
```

```
with read only;
```

```
View created.
```

## Änderungen auf Sichten

Beispielrelationen: ESD (EmpName, Salary, Dept), DM (Dept, Mgr)

- **Projektionssicht:**  $ED := \pi_{EmpName, Dept}(ESD)$

**insert into ED values ('Zuse', 'Info')**

→ **insert into ESD values ('Zuse', null, 'Info')**

⇒ kein **insert** auf Sichten, außerhalb derer Nullwerte verboten und keine Defaultwerte angegeben sind

- **Selektionssicht:**  $ES20 := \sigma_{Salary > 20}(\pi_{EmpName, Salary}(ESD))$

**update ES20 set Salary = 18 where EmpName = 'Zuse'**

→ **update ESD set Salary = 18**

**where EmpName = 'Zuse' and Salary > 20**

würde zur Löschung aus Sicht führen  $\xRightarrow{\text{with check option}}$  zurückgewiesen



*Zwei Beispiel-Zustände der Datenbank (mit Beispielsicht ESDM, s.u.):*

ESD	<u>EmpName</u>	Salary	Dept		ESDM	<u>EmpName</u>	Salary	Dept	Mgr
	Ullman	50	Info			Ullman	50	Info	Ullman
	Aho	40	Info			Aho	40	Info	Ullman
	Tanenbaum	30	Info	DM	<u>Dept</u>	Tanenbaum	30	Info	Ullman
	Wirth	45	Info		Info	Wirth	45	Info	Ullman
	Zuse	18	Info		Math	Zuse	18	Info	Ullman
	Hilbert	25	Math		ET	Hilbert	25	Math	Hilbert
	Riese	18	Math			Riese	18	Math	Hilbert
	Ampere	43	ET			Ampere	43	ET	Tesla
	Tesla	51	ET			Tesla	51	ET	Tesla

.....

ESD	<u>EmpName</u>	Salary	Dept		ESDM	<u>EmpName</u>	Salary	Dept	Mgr
	<i>zusätzlich:</i>								
	...			DM					
	Knuth	24	T <sub>E</sub> X		<i>wie oben</i>				
	...								

## Änderungen auf Sichten (Forts.)

- Verbundsicht:  $\text{ESDM} := \text{ESD} \bowtie \text{DM}$

**insert into ESDM values ('Knuth', 30, 'Info', 'Wirth')**

z. B.  $\rightarrow$  **insert into ESD values ('Knuth', 30, 'Info')**

und **update DM set Mgr='Wirth' where Dept='Info'**

Diese Übersetzung passt jedoch nur auf den ersten Beispielzustand. Änderungen auf Verbundsichten sind also nicht eindeutig übersetzbar. Außerdem kann es Seiteneffekte geben, z. B. erscheint nach obigem **insert** auch das Tupel ('Zuse', 18, 'Info', 'Wirth') in ESDM.

$\xRightarrow{\text{in SQL}}$  nur stark eingeschränkte Änderungen auf Verbundsichten

- “Berechnete” Sicht:  $\text{DSum} = \Gamma_{\text{Dept} \# \text{Dept}, \text{sum}(\text{Salary})} \text{SumSal} (\text{ESD})$

**update DSum set SumSal=SumSal\*0.9 where Dept='Info'**

$\rightarrow$  ???

$\xRightarrow{\text{in SQL}}$  keine Änderungen auf Sichten mit berechneten Attributen

*Alternative:* Änderungen mit **instead of**-Triggern ausprogrammieren(s.u.)

## Oracle-Regeln zur Verwendung von DML-Anweisungen auf Sichten

- Auf einfachen Sichten (eine Tabelle, keine Funktionen) dürfen DML-Anweisungen wie gewohnt ausgeführt werden.
- Auf Sichten, in denen jede Zeile von genau einer Zeile einer Basistabelle “abstammt”, funktionieren manche DML-Anweisungen (komplizierte Ausnahmen).
- **delete** ist auf jeden Fall ausgeschlossen, wenn die Sichtdefinition eines der folgenden Konstrukte enthält:
  - Aggregierungsfunktion
  - **group-by**-Klausel
  - das **distinct**-Schlüsselwort
  - das Schlüsselwort **rownum** (Pseudospalte für Ergebniszeilennummer)
- **insert/update** sind bei folgenden Konstrukten in der Sicht ausgeschlossen:
  - Aggregierungsfunktion
  - **group-by**-Klausel
  - das **distinct**-Schlüsselwort
  - das Schlüsselwort **rownum**
  - Spalten, die durch Ausdrücke definiert sind
  - (**insert:**) **not-null**-Spalten in der Basis-Tabelle, die nicht zur Sicht gehören

## Verwendung von Optionen bei Sichten

- Man kann sicherstellen, dass gar keine DML-Anweisungen auf einer Sicht verwendet werden dürfen, indem man die **with-read-only**-Option in der Sichtdefinition verwendet.
- Jeder Versuch, eine DML-Anweisung auf der Sicht auszuführen, führt dann zu einer Fehlermeldung (Constraintverletzung).
- Man kann sicherstellen, dass die Ergebnisse von DML-Anweisungen, die auf einer Sicht ausgeführt werden, im Definitionsbereich der Sicht bleiben, indem man die **with-check-option**-Klausel verwendet (vgl. S. 4.47), z.B.:

```
create or replace view empvu20
as select *
from EMPLOYEES
where department_id = 20
with check option constraint empvu20_ck;
View created.
```

- Jeder Versuch, eine Zeile aus der Sicht zu verschieben, z.B. hier die ID der Abteilung für eine Zeile in der Sicht zu ändern, schlägt fehl (Constraintverletzung).

## Spezielle Sichten

- **Materialisierte Sichten:** Schnappschüsse von Anfrageergebnissen zu einzelnen Zeitpunkten; enthalten bei regelmäßiger **Wartung** (d.h. Fortpflanzung von Updates der Basisrelationen) redundante Daten.

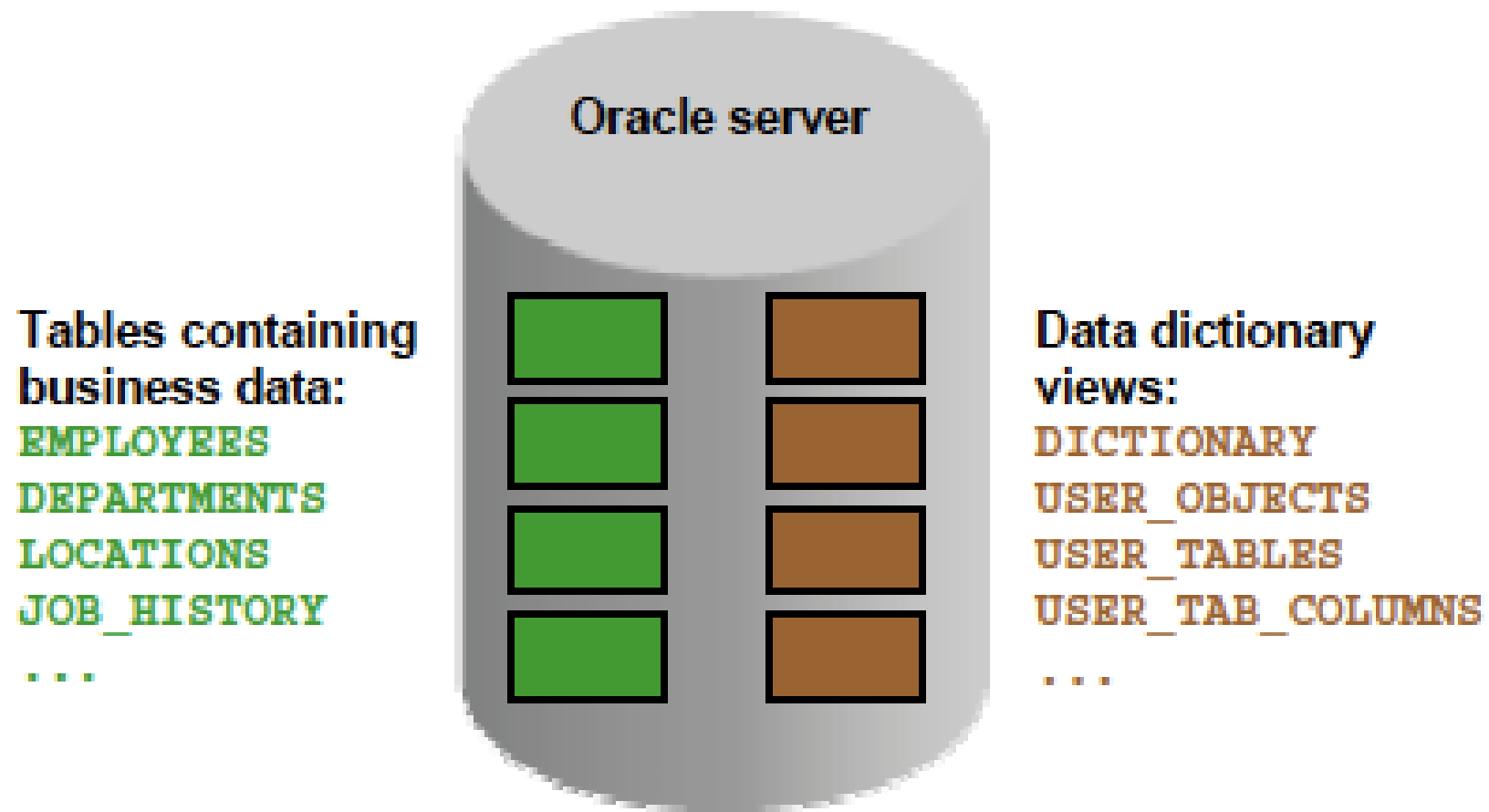
*Implem. der Wartung:* möglichst inkrementell, z. B. mit Triggern

*Vorteile:* Beschleunigung von Anfragen durch Vorausberechnung aufwändiger Zwischenergebnisse, vor allem in Nicht-Standard-Anwendungen oder in Data Warehouses

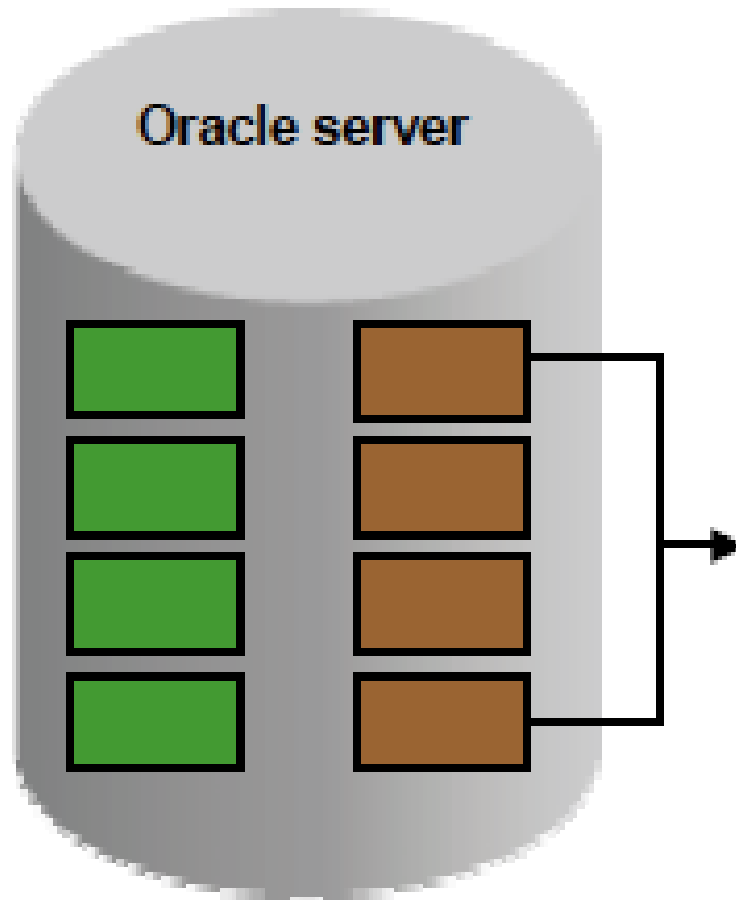
- **“inline views”:** Die in **from**-Klauseln verwendbaren Unteranfragen sind eigentlich unbenannte Sichten! Deshalb sind dort keine Aliase erlaubt, die außerhalb definiert wurden.

*Vorteil:* oft deutliche Vereinfachung von Anfragen (vgl. Beispiele in 3.4)

## 4.5 Das Data Dictionary



## Struktur des Data Dictionary



Das Oracle Data Dictionary besteht aus:

- Basistabellen
- Sichten, auf die der Nutzer zugreifen kann

Der Aufbau kann sich von anderen RDBMS erheblich unterscheiden.

## Struktur des Data Dictionary (Forts.)

Namenskonventionen für Data Dictionary – Sichten  
wie z.B. USER\_TABLES, ALL\_TABLES, DBA\_TAB\_COLUMNS:

Präfix	Zweck
USER	Sicht des Nutzers (Was ist in meinem Schema / was gehört mir?)
ALL	Erweiterte Sicht des Nutzers (Auf was kann ich zugreifen? )
DBA	Sicht des Datenbankadministrators (Was ist in allen Schemata?)
V\$	Performancebezogene Daten

Außerdem gibt es die Sicht DICTIONARY.



## Verwendung der Dictionary-Sichten

DICTIONARY beinhaltet die Namen und Beschreibungen der Tabellen und Sichten des Data Dictionaries.

**describe** DICTIONARY

Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COMMENTS		VARCHAR2(4000)

```
select *  
from DICTIONARY  
where table_name = 'USER_OBJECTS';
```

TABLE_NAME	COMMENTS
USER_OBJECTS	Objects owned by the user

## USER\_OBJECTS-Sicht (Objektinformation)

```
select object_name, object_type, created, status 3  
from USER_OBJECTS  
order by objects_type;
```

OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
REG_ID_PK	INDEX	10-DEC-03	VALID
...			
DEPARTMENTS_SEQ	SEQUENCE	10-DEC-03	VALID
REGIONS	TABLE	10-DEC-03	VALID
LOCATIONS	TABLE	10-DEC-03	VALID
DEPARTMENTS	TABLE	10-DEC-03	VALID
JOB_HISTORY	TABLE	10-DEC-03	VALID
JOB_GRADES	TABLE	10-DEC-03	VALID
EMPLOYEES	TABLE	10-DEC-03	VALID
JOBS	TABLE	10-DEC-03	VALID
COUNTRIES	TABLE	10-DEC-03	VALID
EMP_DETAILS_VIEW	VIEW	10-DEC-03	VALID

---

<sup>3</sup>Name und Typ von DB-Objekten, Erstellungsdatum, Kompilationsstatus der Definition (valid/invalid, insbes. relevant für Sichten und Prozeduren)

## USER\_TABLES-Sicht (Tabelleninformation)

**describe** USER\_TABLES

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)

...

**select** table\_name  
**from** USER\_TABLES;

TABLE_NAME
JOB_GRADES
REGIONS
COUNTRIES
LOCATIONS
DEPARTMENTS

...

## USER\_TAB\_COLUMNS-Sicht (Spalteninformation)

**describe** USER\_TAB\_COLUMNS

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)
COLUMN_ID		NUMBER
DEFAULT_LENGTH		NUMBER
DATA_DEFAULT		LONG

...

## USER\_TAB\_COLUMNS-Sicht (Spalteninformation) (Forts.)

```
select column_name, data_type, data_length,  
        data_precision, data_scale, nullable  
from   USER_TAB_COLUMNS 4  
where table_name = 'EMPLOYEES';
```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NUL
EMPLOYEE_ID	NUMBER	22	6	0	N
FIRST_NAME	VARCHAR2	20			Y
LAST_NAME	VARCHAR2	25			N
EMAIL	VARCHAR2	25			N
PHONE_NUMBER	VARCHAR2	20			Y
HIRE_DATE	DATE	7			N
JOB_ID	VARCHAR2	10			N
SALARY	NUMBER	22	8	2	Y
COMMISSION_PCT	NUMBER	22	2	2	Y
MANAGER_ID	NUMBER	22	6	0	Y
DEPARTMENT_ID	NUMBER	22	4	0	Y

---

<sup>4</sup>In der Übungsumgebung muss es heißen: ... **from** ALL\_TAB\_COLUMNS **where** table\_name = 'EMPLOYEES' **and** owner='HRDB'

## Constraint-Information

- USER\_CONSTRAINTS beschreibt alle Constraints, die auf eigenen Tabellen definiert sind (Tabellen- und Spalten-Constraints).
- USER\_CONS\_COLUMNS gibt die eigenen Tabellenspalten an, die in diese Constraints involviert sind.

### **describe** USER\_CONSTRAINTS

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(30)
SEARCH_CONDITION		LONG
R_OWNER		VARCHAR2(30)
R_CONSTRAINT_NAME		VARCHAR2(30)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)

...

## Constraint-Information (Forts.)

```
select constraint_name, constraint_type, search_condition,  
        r_constraint_name, delete_rule, status 5  
from   USER_CONSTRAINTS  
where table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	CON	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL			ENABLED
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL			ENABLED
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL			ENABLED
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL			ENABLED
EMP_SALARY_MIN	C	salary > 0			ENABLED
EMP_EMAIL_UK	U				ENABLED
EMP_EMP_ID_PK	P				ENABLED
EMP_DEPT_FK	R		DEPT_ID_PK	NO ACTION	ENABLED
EMP_JOB_FK	R		JOB_ID_PK	NO ACTION	ENABLED
EMP_MANAGER_FK	R		EMP_EMP_ID_PK	NO ACTION	ENABLED

---

<sup>5</sup>r\_constraint\_name steht für: referenced constraint name (z.B. das Primärschlüssel-Constraint als Ziel eines Fremdschlüssels)

## Constraint-Information (Forts.)

**describe** USER\_CONS\_COLUMNS

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

```
select constraint_name, column_name  
from USER_CONS_COLUMNS  
where table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_EMAIL_UK	EMAIL
EMP_SALARY_MIN	SALARY
EMP_JOB_NN	JOB_ID
EMP_HIRE_DATE_NN	HIRE_DATE

...



## Sichteninformation

**describe USER\_VIEWS**

Name	Null?	Type
VIEW_NAME	NOT NULL	VARCHAR2(30)
TEXT_LENGTH		NUMBER
TEXT		LONG

**select distinct view\_name  
from USER\_VIEWS;**

VIEW_NAME
EMP_DETAILS_VIEW

**select text  
from USER\_VIEWS  
where view\_name = 'EMP\_DETAILS\_VIEW';**

TEXT
SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.country_id, e.first_name, e.last_name, e.salary, e.commission_pct, d.department_name, j.job_title, l.city, l.state_province, c.country_name, r.region_name FROM employees e, departments d, jobs j, locations l, countries c, regions r WHERE e.department_id = d.department_id AND d.location_id = l.location_id AND l.country_id = c.country_id AND c.region_id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

## Einfügen von Kommentaren

- Man kann mit Hilfe der **comment**-Anweisung Kommentare zu Tabellen oder Spalten hinzufügen:

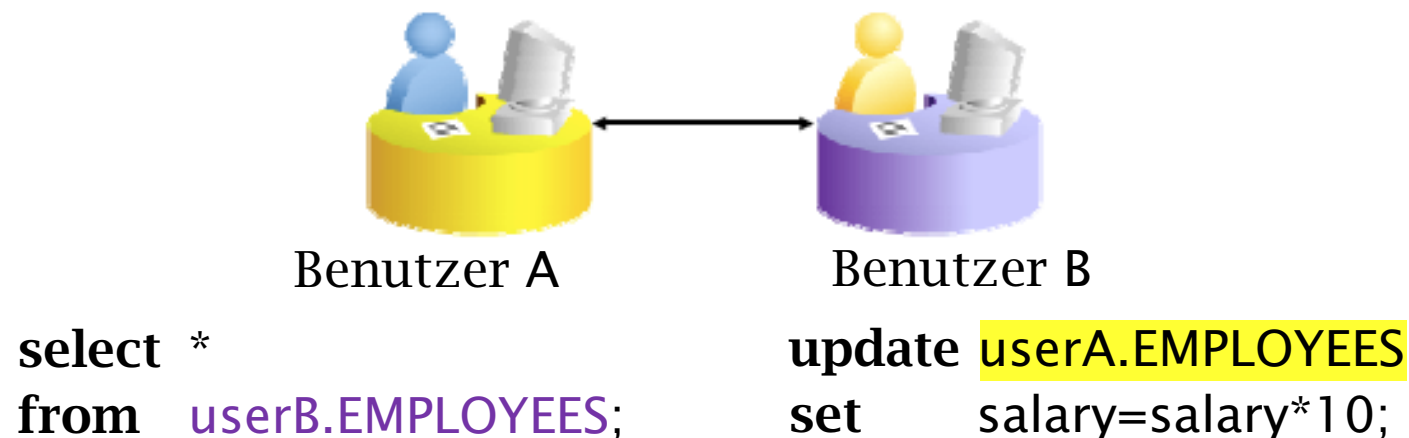
```
comment on table EMPLOYEES  
  is 'Employee Information';  
Comment created.
```

- Kommentare können in den folgenden Data Dictionary – Sichten betrachtet werden:
  - ALL\_TAB\_COMMENTS
  - USER\_TAB\_COMMENTS
  - ALL\_COL\_COMMENTS
  - USER\_COL\_COMMENTS

## 4.6 Datenschutz

### Zugriff auf Tabellen anderer Benutzer

- Im **Schema** (nach Oracle-Terminologie!) eines Benutzers befinden sich alle eigenen, d.h. selbst erzeugten Datenbankobjekte (Tabellen, Sichten u.a.)
- Objekte, die anderen Benutzern gehören, befinden sich nicht im Schema des Benutzers.
- Man kann auf eine fremde Tabelle oder Sicht mit dem Namen des Eigentümers (engl. *owner*) als Präfix zugreifen, sofern man über die entsprechenden Rechte verfügt.



## Arten von Zugriffsrechten (engl. *privileges*)

- **Systemrechte:**  
erlauben den Zugang zur Datenbank und das Erzeugen/Entfernen von Datenbankobjekten
- **Objektrechte:**  
erlauben das Manipulieren der Inhalte von Datenbankobjekten
- Der Datenbankadministrator (DBA) besitzt weitreichende Systemrechte für Aufgaben wie:
  - Erzeugen neuer / Entfernen bestehender Benutzer
  - Vergeben von System-/Objektrechten an Benutzer
  - Entfernen von Tabellen beliebiger Benutzer
  - Backups von Tabellen

## Erzeugung von Benutzern

- Der DBA erzeugt Benutzeraccounts mit Hilfe des **create-user**-Befehls. Dabei initialisiert er das Passwort.

```
create user Benutzer  
identified by Passwort;
```

```
create user scotty  
identified by Yt99Cs;  
User created.
```

## Ändern des Passwortes

- Man kann und soll sein eigenes Passwort mit Hilfe des **alter-user**-Befehls ändern.

```
alter user scotty  
identified by Warp_8;  
User altered.
```

## Vergabe von Systemrechten

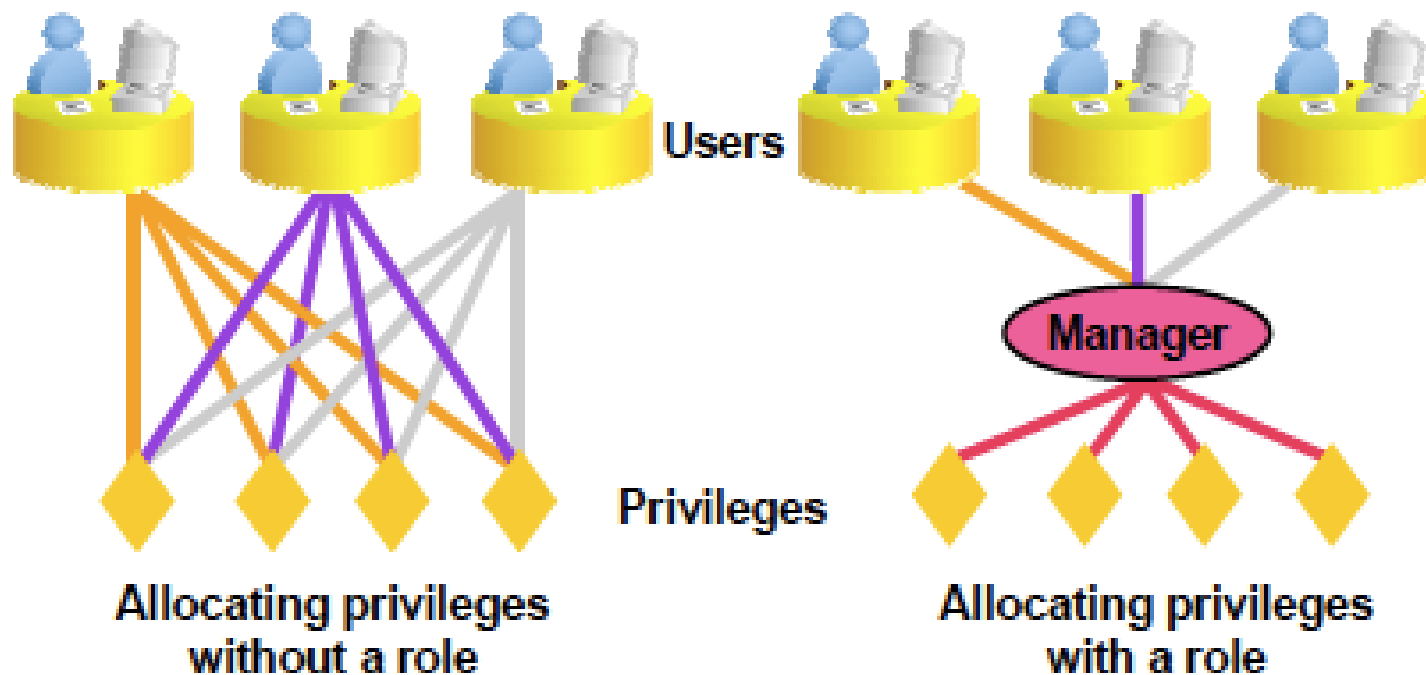
- Nachdem ein Benutzer erzeugt wurde, kann der DBA spezielle Systemrechte vergeben.

```
grant  Recht [, Recht ...]  
to     {Benutzer | Rolle | public} [, {Benutzer | Rolle} ... ];
```

- Z.B. soll der Benutzer scotty Datenbank-Anwendungen entwickeln können:

```
grant  create session, create table, create view,  
        create sequence, create procedure  
to     scotty;  
Grant succeeded.
```

## Was ist eine Rolle?



Eine **Rolle** wie z.B. Manager ist eine Gruppe von Rechten (und evtl. von anderen Rollen), die an Benutzer (oder andere Rollen) vergeben werden kann. Eine Rolle kann aber auch als eine Gruppe von Benutzern (und evtl. von anderen Rollen) aufgefasst werden, an die Rechte (oder andere Rollen) vergeben werden können.

## Verwaltung von Rollen

- Erzeugen einer Rolle:  
**create role** Manager;  
Role created.
- “Vergeben” von Rechten an die Rolle:  
**grant** create table, create view  
**to** Manager;  
Grant succeeded.
- Vergeben der Rolle an Benutzer:  
**grant** Manager  
**to** king, prince;  
Grant succeeded.



## Übersicht über mögliche Objektrechte

Object Privilege	Table	View	Sequence	Procedure
<b>ALTER</b>	✓		✓	
<b>DELETE</b>	✓	✓		
<b>EXECUTE</b>				✓
<b>INDEX</b> <sup>6</sup>	✓			
<b>INSERT</b>	✓	✓		
<b>REFERENCES</b> <sup>6</sup>	✓			
<b>SELECT</b>	✓	✓	✓	
<b>UPDATE</b>	✓	✓		

---

<sup>6</sup>index: Index für diese Tabelle anlegen; references: Fremdschlüsselconstraint (auf einer eigenen Tabelle) bzgl. dieser Tabelle anlegen

## Vergabe von Objektrechten

- Objektrechte variieren von Objekt zu Objekt.
- Der Eigentümer eines Objekts hat alle Rechte daran.
- Der Eigentümer eines Objekts kann für dieses Objekt Rechte mit dem **grant**-Befehl vergeben.

```
grant  Objektrecht [(Spalten)]7 [, ... ]  
on    Objekt  
to    {Benutzer | Rolle | public}[, {Benutzer | Rolle} ... ]  
[with grant option];
```

---

<sup>7</sup>Spaltenangaben nur bei update|insert|references-Rechten

## Vergabe von Objektrechten (Forts.)

- Beispiel: Vergeben des **select**-Rechts auf einer Sicht an mehrere Benutzer:

```
grant select
on    EMPVIEW_NOSAL
to    peter, paul, mary;
Grant succeeded.
```

*Hinweis:* Mit Rechten auf einer Sicht werden entsprechende Rechte auf den unterliegenden Tabellen nicht mehr benötigt.

- Beispiel: Vergeben des **update**-Rechts für spezielle Spalten auf einer Tabelle an Benutzer und Rollen:

```
king: grant update (department_name, location_id)
on    DEPARTMENTS
to    scotty, Manager;
Grant succeeded.
```

## Weitergabe von Rechten

- Beispiel: Vergeben von Rechten mit der Erlaubnis, diese Rechte weiterzugeben:

```
king: grant select, insert
      on    DEPARTMENTS
      to    scotty
      with grant option;
Grant succeeded.
```

- Beispiel: Vergeben von Leserechten an einer Tabelle an alle Benutzer:

```
scotty: grant select
        on    king.DEPARTMENTS
        to    public;
Grant succeeded.
```

```
king: grant select
      on    DEPARTMENTS
      to    public;
Grant succeeded.
```

## Rücknahme von Rechten

- Mit Hilfe des **revoke**-Befehl kann man Rechte zurückzunehmen, die man zuvor vergeben hatte.
- Auch Rechte, die mit **grant option** weitervergeben wurden, werden damit zurückgenommen.

```
revoke { Recht [, Recht ... ] | all }  
on      Objekt  
from    { Benutzer | Rolle | public } [, ... ]
```

z.B.:

```
king: revoke insert  
      on      DEPARTMENTS  
      from    scotty;  
Revoke succeeded.
```

## Informationen über vergebene Rechte im Data Dictionary

Data Dictionary Sicht	Beschreibung
USER_SYS_PRIVS	Systemrechte, die direkt an den Benutzer vergeben wurden
ROLE_SYS_PRIVS	Systemrechte, die an Rollen des Benutzers vergeben wurden
USER_ROLE_PRIVS	Rollen, die direkt an den Benutzer vergeben wurden
ROLE_ROLE_PRIVS	Rollen, die an Rollen des Benutzers vergeben wurden
ROLE_TAB_PRIVS	Objektrechte, die an Rollen des Benutzers vergeben wurden
USER_TAB_PRIVS_RECD	Objektrechte, die direkt an den Benutzer vergeben wurden
USER_TAB_PRIVS_MADE	Objektrechte, die vom Benutzer auf eigenen Objekten direkt an andere Benutzer oder Rollen vergeben wurden
USER_COL_PRIVS_RECD	Objektrechte, die direkt an den Benutzer für bestimmte Spalten vergeben wurden
USER_COL_PRIVS_MADE	Objektrechte, die vom Benutzer für bestimmte Spalten eines eigenen Objekts direkt vergeben wurden

Alle direkt oder indirekt (über Rollen) erhaltenen/vergebenen Rechte zu einem Benutzer bzw. zu einer Tabelle müssen also erst zusammengesucht werden.

## Diskussion von Zugriffsrechten

- Ein Zugriffsrecht auf einem Datenbankobjekt ist offensichtlich dadurch bestimmt,
  - wer (engl. *grantor*)  
→ wem (engl. *grantee*)
    - welche Operation
    - auf welchem Objekterlaubt.
- Aus der Perspektive einer Tabelle kann
  - der Eigentümer  
→ ausgewählten Benutzern oder Benutzergruppen (Rollen)
    - die **select**-, **update**-, **insert**- oder **delete**-Operation
    - auf der ganzen Tabelle, auf einzelnen Spalten (**update**)  
oder auf einer oder mehreren Sichten auf die Tabelleerlauben.

## Diskussion von Zugriffsrechten (Forts.)

- Mit Hilfe von Sichten kann also die Rechtevergabe auf bestimmte Tabellenteile, -auszüge bzw. allgemein bestimmte Anfrageergebnisse beschränkt werden, ohne dass der Rechte-Empfänger entsprechende Rechte an der/den unterliegenden Tabelle/n braucht oder bekommt.

Für **insert/update/delete**-Operationen greifen allerdings oft technische Restriktionen auf Sichten (siehe Abschnitt 4.4).

- Sichten lassen sich sogar so definieren, dass ihre Auswertung von der aktuellen Zeit (**sysdate**) oder vom Benutzernamen (**user**) abhängt.
- Nicht differenzieren lassen sich jedoch Rechte für einzelne Operationen abhängig von deren Ausführungszeit und vor allem nicht von deren Parametern bzw. Effekten:

So könnten z.B. per **update**-Recht auf einer passenden Sicht jedem Kunden Mengenänderungen von eigenen Bestellungen erlaubt, aber nicht auf Mengenerhöhungen beschränkt werden. Für so flexible Steuerungen werden Trigger benötigt (siehe Abschnitt 6.2).

- Explizit vergeben werden können nur Zugriffsrechte, keine Zugriffsverbote.