Kapitel 4
Schema-Definition

- das Datenbankschema umfasst alle Informationen über die Struktur der Datenbank,
- Tabellen, Views, Constraints, Indexe, Cluster, Trigger ...
- objektrelationale DB: Datentypen, ggf. Methoden
- wird mit Hilfe der DDL (Data Definition Language) manipuliert,
- `CREATE`, `ALTER` und `DROP` von Schemaobjekten,
- Vergabe von Zugriffsrechten: `GRANT`.

ERZEUGEN VON TABELLEN

```
CREATE TABLE <table>
  (<col> <datatype>,
   ⋮
   <col> <datatype>)
```

`CHAR(`n`)`: Zeichenkette fester Länge n.

`VARCHAR2(`n`)`: Zeichenkette variabler Länge $\leq n$.
$||$: Konkatenation von Strings.

`NUMBER`: Zahlen. Auf `NUMBER` sind die üblichen Operatoren $+$, $-$, $*$ und $/$ sowie die Vergleiche $=$, $>$, $>=$, $<=$ und $<$ erlaubt. Außerdem gibt es `BETWEEN x AND y`. Ungleichheit: $!=$, $\wedge=$, $\neg=$ oder $<>$.

`DATE`: Datum und Zeiten: Jahrhundert – Jahr – Monat – Tag – Stunde – Minute – Sekunde. U.a. wird auch Arithmetik für solche Daten angeboten.

weitere Datentypen findet man im Manual.

Andere DBMS verwenden in der Regel andere Namen für dieselben oder ähnliche Datentypen!

TABELLENDEFINITION

Das folgende SQL-Statement erzeugt z.B. die Relation *City* (noch ohne Integritätsbedingungen):

```
CREATE TABLE City
       ( Name           VARCHAR2(35),
         Country        VARCHAR2(4),
         Province       VARCHAR2(32),
         Population     NUMBER,
         Longitude      NUMBER,
         Latitude       NUMBER );
```


TABELLENDEFINITION: CONSTRAINTS

Mit den Tabellendefinitionen können Eigenschaften und Bedingungen an die jeweiligen Attributwerte formuliert werden.

- Bedingungen an ein einzelnes oder mehrere Attribute:
- Wertebereichseinschränkungen,
- Angabe von Default-Werten,
- Forderung, dass ein Wert angegeben werden muss,
- Angabe von Schlüsselbedingungen,
- Prädikate an Tupel.

```
CREATE TABLE <table>
  (<col> <datatype> [DEFAULT <value>]
       [<colConstraint> ... <colConstraint>],
  ⋮
  <col> <datatype> [DEFAULT <value>]
       [<colConstraint> ... <colConstraint>],
  [<tableConstraint>,]
  ⋮
  [<tableConstraint>])
```

- `<colConstraint>` betrifft nur *eine* Spalte,
- `<tableConstraint>` kann mehrere Spalten betreffen.

TABELLENDEFINITION: DEFAULT-WERTE

```
DEFAULT <value>
```

Ein Mitgliedsland einer Organisation wird als volles Mitglied angenommen, wenn nichts anderes bekannt ist:

```
CREATE TABLE is_member
      ( Country       VARCHAR2(4),
        Organization  VARCHAR2(12),
        Type          VARCHAR2(30)
                      DEFAULT 'member')

INSERT INTO is_member VALUES
    ('CZ', 'EU', 'membership applicant');
INSERT INTO is_member (Land, Organization)
    VALUES ('D', 'EU');
```

Country	Organization	Type
CZ	EU	membership applicant
D	EU	member
⋮	⋮	⋮

TABELLENDEFINITION: CONSTRAINTS

Zwei Arten von Bedingungen:

- Eine Spaltenbedingung `<colConstraint>` ist eine Bedingung, die nur *eine* Spalte betrifft (zu der sie definiert wird)
- Eine Tabellenbedingung `<tableConstraint>` kann mehrere Spalten betreffen.

Jedes `<colConstraint>` bzw. `<tableConstraint>` ist von der Form

```
[CONSTRAINT <name>] <bedingung>
```

TABELLENDEFINITION: BEDINGUNGEN (ÜBERBLICK)

Syntax:

```
[CONSTRAINT <name>] <bedingung>
```

Schlüsselwörter in `<bedingung>`:

1. `CHECK (<condition>)`: Keine Zeile darf `<condition>` verletzen. `NULL`-Werte ergeben dabei ggf. ein *unknown*, also *keine Bedingungsverletzung*.
2. `[NOT] NULL`: Gibt an, ob die entsprechende Spalte Nullwerte enthalten darf (nur als `<colConstraint>`).
3. `UNIQUE (<column-list>)`: Fordert, dass jeder Wert nur einmal auftreten darf.
4. `PRIMARY KEY (<column-list>)`: Deklariert die angegebenen Spalten als Primärschlüssel der Tabelle.
5. `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE|ON DELETE SET NULL]`:
   gibt an, dass eine Menge von Attributen Fremdschlüssel ist.

TABELLENDEFINITION: SYNTAX

```
[CONSTRAINT <name>] <bedingung>
```

Dabei ist `CONSTRAINT <name>` optional (ggf. Zuordnung eines systeminternen Namens).

- `<name>` wird bei `NULL`-, `UNIQUE`-, `CHECK`- und `REFERENCES`-Constraints benötigt, wenn das Constraint irgendwann einmal geändert oder gelöscht werden soll,
- `PRIMARY KEY` kann man ohne Namensnennung löschen und ändern.

Da bei einem `<colConstraint>` die Spalte implizit bekannt ist, fällt der `(<column-list>)` Teil weg.

TABELLENDEFINITION: CHECK CONSTRAINTS

- als Spaltenconstraints: Wertebereichseinschränkung

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   ...);
```

- Als Tabellenconstraints: beliebig komplizierte Integritätsbedingungen an ein Tupel.

TABELLENDEFINITION: PRIMARY KEY, UNIQUE UND NULL

- PRIMARY KEY (<column-list>): Deklariert diese Spalten als Primärschlüssel der Tabelle.
- Damit entspricht PRIMARY KEY der Kombination aus UNIQUE und NOT NULL.
- UNIQUE wird von NULL-Werten *nicht* unbedingt verletzt, während PRIMARY KEY NULL-Werte *verbietet*.

Eins	Zwei
a	b
a	NULL
NULL	b
NULL	NULL

erfüllt UNIQUE (Eins,Zwei).

- Da auf jeder Tabelle nur ein PRIMARY KEY definiert werden darf, wird NOT NULL und UNIQUE für Candidate Keys eingesetzt.

Relation *Country*: Code ist PRIMARY KEY, Name ist Candidate Key:

```
CREATE TABLE Country
  ( Name         VARCHAR2(32) NOT NULL UNIQUE,
    Code         VARCHAR2(4) PRIMARY KEY);
```

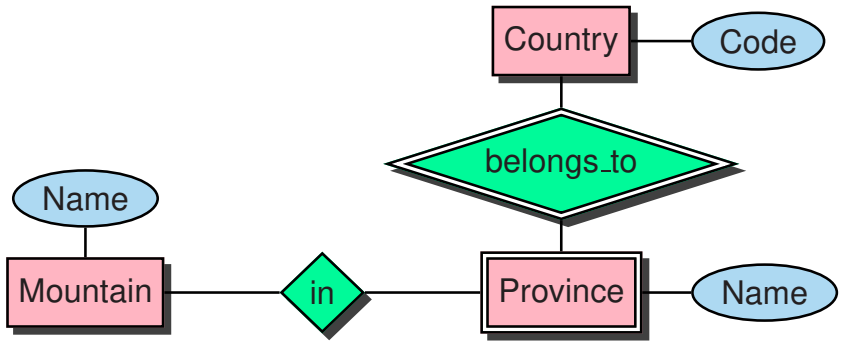
TABELLENDEFINITION: FOREIGN KEY ...REFERENCES

- `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE|ON DELETE SET NULL]`: gibt an, dass das Attributtupel `<column-list>` der Tabelle ein Fremdschlüssel ist und das Attributtupel `<column-list2>` der Tabelle `<table>` referenziert.
- Das referenzierte Attributtupel `<table>(<column-list2>)` muss ein *Candidate Key* von `<table>` sein.
- Eine `REFERENCES`-Bedingung wird durch `NULL`-Werte nicht verletzt.
- `ON DELETE CASCADE|ON DELETE SET NULL`: Referentielle Aktion (später).

```
CREATE TABLE is_member
  (Country       VARCHAR2(4)
        REFERENCES Country(Code),
   Organization  VARCHAR2(12)
        REFERENCES Organization(Abbreviation),
   Type          VARCHAR2(30) DEFAULT 'member');
```


TABELLENDEFINITION: FREMDSCHLÜSSEL

Ein Berg liegt in einer Provinz eines Landes:




```
CREATE TABLE geo_Mountain
  ( Mountain VARCHAR2(20)
                REFERENCES Mountain(Name),
    Country VARCHAR2(4) ,
    Province VARCHAR2(32) ,
    CONSTRAINT GMountRefsProv
                FOREIGN KEY (Country,Province)
                REFERENCES Province (Country,Name));
```

TABELLENDEFINITION

Vollständige Definition der Relation *City* mit Bedingungen und Schlüsseldeklaration:

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Country VARCHAR2(4)
     REFERENCES Country(Code),
   Province VARCHAR2(32),      -- + <tableConstraint>
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   Longitude NUMBER CONSTRAINT CityLong
     CHECK ((Longitude >= -180) AND (Longitude <= 180)),
   Latitude NUMBER CONSTRAINT CityLat
     CHECK ((Latitude >= -90) AND (Latitude <= 90)),
   CONSTRAINT CityKey
     PRIMARY KEY (Name, Country, Province),
   FOREIGN KEY (Country,Province)
     REFERENCES Province (Country,Name));
```

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`-Klausel enthält, erstellt wird, muss `<table>` bereits definiert und `<column-list>` dort als `PRIMARY KEY` deklariert sein.

VIEWS (=SICHTEN)

- Virtuelle Tabellen
- nicht zum Zeitpunkt ihrer Definition berechnet, sondern
- jedesmal berechnet, wenn auf sie zugegriffen wird.
- spiegeln also stets den aktuellen Zustand der ihnen zugrundeliegenden Relationen wieder.
- Änderungsoperationen nur in eingeschränktem Umfang möglich.

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
<select-clause>;
```

Beispiel: Ein Benutzer benötigt häufig die Information, welche Stadt in welchem Land liegt, ist jedoch weder an Landeskürzeln noch Einwohnerzahlen interessiert.

```
CREATE VIEW CityCountry (City, Country) AS
  SELECT City.Name, Country.Name
  FROM City, Country
  WHERE City.Country = Country.Code;
```

Wenn der Benutzer nun nach allen Städten in Kamerun sucht, so kann er die folgende Anfrage stellen:

```
SELECT *
FROM CityCountry
WHERE Country = 'Cameroon';
```

Löschen von Tabellen und Views

- Tabellen bzw. Views werden mit `DROP TABLE` bzw. `DROP VIEW` gelöscht:

```
DROP TABLE <table-name> [CASCADE CONSTRAINTS];
DROP VIEW <view-name>;
```

- Tabellen müssen nicht leer sein, wenn sie gelöscht werden sollen.
- Eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, kann mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden.
- Mit `DROP TABLE <table> CASCADE CONSTRAINTS` wird eine Tabelle mit allen auf sie zeigenden referentiellen Integritätsbedingungen gelöscht und die referenzierenden Tupel werden entfernt.

Ändern von Tabellen und Views

später.

Kapitel 5
Einfügen und Ändern von Daten

- Einfügen (in existierende Tabellen):
  - Tupel (als Konstanten)
  - Mengen (Ergebnisse von Anfragen)
- Ändern: Einfache Erweiterung des SELECT-FROM-WHERE-Statements.

5.1 Einfügen von Daten

- INSERT-Statement.
- Daten einzeln von Hand einfügen,

```
INSERT INTO <table>[(<column-list>)]
VALUES (<value-list>);
```

- Ergebnis einer Anfrage einfügen:

```
INSERT INTO <table>[(<column-list>)]
<subquery>;
```

- Rest wird ggf. mit Nullwerten aufgefüllt.

So kann man z.B. das folgende Tupel einfügen:

```
INSERT INTO Country (Name, Code, Population)
VALUES ('Lummerland', 'LU', 4);
```

Eine Tabelle *Metropolis (Name, Country, Population)* kann man z.B. mit dem folgenden Statement füllen:

```
INSERT INTO Metropolis
SELECT Name, Country, Population
FROM City
WHERE Population > 1000000;
```

Es geht auch noch kompakter (implizite Tabellendefinition):

```
CREATE TABLE Metropolis AS
SELECT Name, Country, Population
FROM City WHERE Population > 1000000;
```


5.2 Löschen von Daten

Tupel können mit Hilfe der DELETE-Klausel aus Relationen gelöscht werden:

```
DELETE FROM <table>
WHERE <predicate>;
```

Dabei gilt für die WHERE-Klausel das für SELECT gesagte.

Mit einer leeren WHERE-Bedingung kann man z.B. eine ganze Tabelle abräumen (die Tabelle bleibt bestehen, sie kann mit DROP TABLE entfernt werden):

```
DELETE FROM City;
```

Der folgende Befehl löscht sämtliche Städte, deren Einwohnerzahl kleiner als 50.000 ist.

```
DELETE FROM City
WHERE Population < 50000;
```

5.3 Ändern von Tupeln

```
UPDATE <table>
SET <attribute> = <value> | (<subquery>),
                    ⋮
    <attribute> = <value> | (<subquery>),
    (<attribute-list>) = (<subquery>),
                    ⋮
    (<attribute-list>) = (<subquery>)
WHERE <predicate>;
```

Beispiel:

```
UPDATE City
SET Name = 'Leningrad',
    Population = Population + 1000,
WHERE Name = 'Sankt Peterburg';
```

Beispiel: Die Einwohnerzahl jedes Landes wird als die Summe der Einwohnerzahlen aller Provinzen gesetzt:

```
UPDATE Country
SET Population = (SELECT SUM(Population)
                  FROM Province
                  WHERE Province.Country=Country.Code);
```


5.4 Referentielle Integrität – A First Look

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`-Klausel enthält, erstellt wird, muss `<table>` bereits definiert und `<column-list>` dort ein *Candidate Key* sein.
- Beim Einfügen von Daten müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.
- Beim Löschen eines referenzierten Tupels muss die referentielle Integrität erhalten bleiben.
- Eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, wird mit `DROP TABLE <table> CASCADE CONSTRAINTS` gelöscht.

5.5 Transaktionen in ORACLE

Beginn einer Transaktion

```
SET TRANSACTION READ [ONLY | WRITE];
```

Sicherungspunkte setzen

Für eine längere Transaktion können zwischendurch Sicherungspunkte gesetzt werden:

```
SAVEPOINT <savepoint>;
```

Ende einer Transaktion

- `COMMIT`-Anweisung, macht alle Änderungen persistent,
- `ROLLBACK [TO <savepoint>]` nimmt alle Änderungen [bis zu `<savepoint>`] zurück,
- DDL-Anweisung (z.B. `CREATE, DROP, RENAME, ALTER`),
- Benutzer meldet sich von ORACLE ab,
- Abbruch eines Benutzerprozesses.

Kapitel 6
Spezialisierte Datentypen

- (einfache) Built-In-Typen: Zeitangaben
- Möglichkeit, zusammengesetzte Datentypen selber zu definieren
  (z.B. Geo-Koordinaten aus Länge, Breite) [seit Oracle 8i/1997]
- Verlassen der 1. Normalform: Mengenwertige Einträge – Geschachtelte Tabellen [seit Oracle 8i/8.1.5/1997]
- selbstdefinierte Objekttypen (Siehe Folie 204)
  - Objekte an Stelle von Tupeln und Attributwerten
  - mit Objektmethoden
  - basierend auf PL-SQL [seit Oracle 8.0/1997/1998]
  - mit Java-Methoden [seit Oracle 8i/8.1.5/1999]
  - Objekttypen basierend auf Java-Klassen, Vererbung [seit Oracle 9i/2001]
- Built-In-Typen mit festem Verhalten
  - XMLType (siehe Folie 330) [seit Oracle 9i-2/2002]
  - Ergänzungen durch “DataBlades”, “Extensions” (Spatial Data (seit Oracle 8i/8.1.5) etc.)

6.1 Zeitangaben

Der Datentyp `DATE` speichert Jahrhundert, Jahr, Monat, Tag, Stunde, Minute und Sekunde.

- Eingabe-Format mit `NLS_DATE_FORMAT` setzen,
- Default: `'DD-MON-YY'` eingestellt, d.h. z.B. `'20-Oct-97'`.

```
CREATE TABLE Politics
  ( Country VARCHAR2(4),
    Independence DATE,
    Government VARCHAR2(120));

ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';

INSERT INTO politics VALUES
  ('B','04 10 1830','constitutional monarchy');
```

Alle Länder, die zwischen 1200 und 1600 gegründet wurden:

```
SELECT Country, Independence
FROM Politics
WHERE Independence BETWEEN
'01 01 1200' AND '31 12 1599';
```

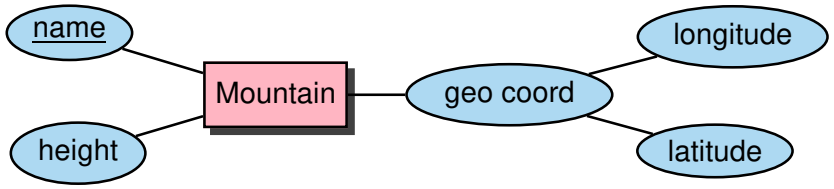
Land	Datum
MC	01 01 1419
NL	01 01 1579
E	01 01 1492
THA	01 01 1238

ZEITANGABEN

ORACLE bietet einige Funktionen um mit dem Datentyp `DATE` zu arbeiten:

- `SYSDATE` liefert das aktuelle Datum.
- Addition und Subtraktion von Absolutwerten auf `DATE` ist erlaubt, Zahlen werden als Tage interpretiert: `SYSDATE` $+ 1$ ist morgen, `SYSDATE` $+ (10/1440)$ ist “in zehn Minuten”.
- `ADD_MONTHS`(d, n) addiert n Monate zu einem Datum d.
- `LAST_DAY`(d) ergibt den letzten Tag des in d angegebenen Monats.
- `MONTHS_BETWEEN`(d_1,d_2) gibt an, wieviele Monate zwischen 2 Daten liegen.

6.2 Zusammengesetzte Datentypen



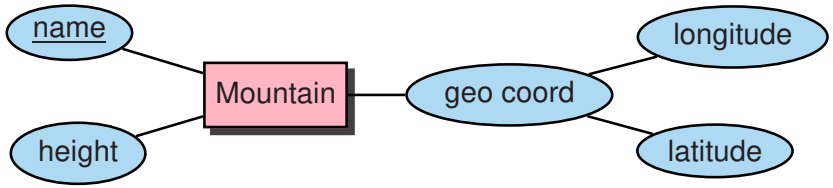
Neue Klasse von Schemaobjekten: `CREATE TYPE`

- `CREATE [OR REPLACE] TYPE <name> AS OBJECT`
  `(<attr> <datatype>,`
  `⋮`
  `<attr> <datatype>);`
- Bei “echten” Objekten kommt noch ein `CREATE TYPE BODY ...` dazu, in dem die Methoden in PL/SQL definiert werden ... später.

  Ohne Body bekommt man einfache komplexe Datentypen (ähnlich wie *Records*).

Zusammengesetzte Datentypen

Geographische Koordinaten:




```
CREATE TYPE GeoCoord AS OBJECT
  ( Longitude NUMBER,
    Latitude  NUMBER);
/

CREATE TABLE Mountain
      ( Name          VARCHAR2(20),
        Height        NUMBER,
        Coordinates   GeoCoord);
```

`CREATE TYPE <type> AS OBJECT (...)`
definiert automatisch eine *Konstruktormethode* `<type>`:

```
INSERT INTO Mountain
  VALUES ('Feldberg', 1493, GeoCoord(8,48));
SELECT * FROM Mountain;
```

Name	Height	Coordinates(Longitude, Latitude)
Feldberg	1493	GeoCoord(8,48)

ZUSAMMENGESETZTE DATENTYPEN

Zugriff auf einzelne Komponenten von komplexen Attributen in der bei Records üblichen *dot*-Notation.
Hierbei muss der Pfad mit dem Alias einer Relation beginnen (Eindeutigkeit!):

```
SELECT Name, B.Coordinates.Longitude,
     B.Coordinates.Latitude
FROM Mountain B;
```

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	8	48

Constraints in zusammengesetzten Datentypen:

```
CREATE TABLE Mountain
  (Name          VARCHAR2(20),
  Height         NUMBER,
  Coordinates    GeoCoord,
CHECK ((Coordinates.Longitude >= -180) AND
  (Coordinates.Longitude <= 180)),
CHECK ((Coordinates.Latitude >= -90) AND
  (Coordinates.Latitude <= 90)));
```


6.3 Geschachtelte Tabellen

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

- mengenwertige Attribute (“Kollektion”, Tabellen als *Datentypen*)

⇒ tabellenwertige Attribute

- Generischer Typ `TABLE OF <inner_type>`

⇒ Generische Syntax

[Dieser Abschnitt kann übersprungen werden.]

GESCHACHTELTE TABELLEN

```
CREATE [OR REPLACE] TYPE <inner_type>
  AS OBJECT (...);
/
CREATE [OR REPLACE] TYPE <inner_table_type> AS
  TABLE OF <inner_type>;
/
CREATE TABLE <table_name>
  (... ,
   <table-attr> <inner_table_type> ,
   ... )
NESTED TABLE <table-attr> STORE AS <name >;
```

Beispiel

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Lang_nested;
```


GESCHACHTELTE TABELLEN

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Lang_nested;
```

Wieder: Konstruktormethoden

```
INSERT INTO NLanguage
VALUES( 'SK',
        Languages_list
          (  Language_T('Slovak',95),
             Language_T('Hungarian',5)));
```

GESCHACHTELTE TABELLEN

```
SELECT *
FROM NLanguage
WHERE Country='CH';
```

Country	Languages(Name, Percentage)
CH	Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

```
SELECT Languages
FROM NLanguage
WHERE Country='CH';
```

Languages(Name, Percentage)
Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

ANFRAGEN AN GESCHACHTELTE TABELLEN

Inhalt von inneren Tabellen:

```
THE (SELECT <table-attr> FROM ...)
```

```
SELECT ...
FROM THE (<select-statement>)
WHERE ... ;

INSERT INTO THE (<select-statement>)
VALUES ... / SELECT ... ;

DELETE FROM THE (<select-statement>)
WHERE ... ;
```

```
SELECT Name, Percentage
FROM THE (SELECT Languages
          FROM NLanguage
          WHERE Country='CH');
```

Name	Percentage
German	65
French	18
Italian	12
Romansch	1

KOPIEREN VON GESCHACHTELTEN TABELLEN

Geschachtelte Tabelle “am Stück” einfügen: Menge von Tupeln wird als Kollektion strukturiert:

```
  CAST(MULTISET(SELECT ...) AS <nested-table-type>)
```

```
INSERT INTO NLanguage -- zulässig, aber falsch !!!!
 (SELECT Country,
         CAST(MULTISET(SELECT Name, Percentage
                       FROM Language
                       WHERE Country = A.Country)
              AS Languages_List)
 FROM Language A);
```

jedes Tupel (Land, Sprachenliste) n-mal
(n = Anzahl Sprachen in diesem Land) !!

```
INSERT INTO NLanguage (Country)
 (SELECT Country
  FROM Language);
```

```
UPDATE NLanguage B
SET Languages =
  CAST(MULTISET(SELECT Name, Percentage
                FROM Language A
                WHERE B.Country = A.Country)
       AS Languages_List);
```


GESCHACHTELTE TABELLEN

Liefert eine Anfrage bereits eine Tabelle, kann man diese als ganzes einfügen:

```
INSERT INTO <table>
VALUES (..., THE ( SELECT <attr>
                   FROM <table'>
                   WHERE ...) );
```

```
INSERT INTO NLanguage VALUES
 ('CHXX', THE (SELECT Languages from NLanguage
               WHERE Country='CH'));
```

ARBEITEN MIT GESCHACHTELTEN TABELLEN

Nicht ganz einfach ...

- Unterabfrage darf nur *eine einzige* geschachtelte Tabelle zurückgeben.
  ⇒ nicht möglich in Abhängigkeit von dem Tupel der äußeren Tabelle die jeweils passende innere Tabelle auszuwählen:

Alle Länder, in denen Deutsch gesprochen wird:

```
SELECT Country -- UNZULAESSIG !!!!
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE Name='German');
```


ARBEITEN MIT GESCHACHTELTEN TABELLEN

```
TABLE ([<table>.]<attr>)
```

kann in *Unterabfrage* verwendet werden:

```
SELECT Country
FROM NLanguage
WHERE EXISTS
  (SELECT *
   FROM TABLE (Languages) -- zu dem aktuellen Tupel
   WHERE Name='German');
```

Country
A
B
CH
D
NAM

Aber: Attribute der inneren Tabelle können nicht im äußeren SELECT-Statement ausgewählt werden.

⇒ Ausgabe des prozentualen Anteils in den verschiedenen Ländern nicht möglich.

ARBEITEN MIT GESCHACHTELTEN TABELLEN

CURSOR-Operator:

Beispiel:

```
SELECT Country,
       CURSOR (SELECT *
               FROM TABLE (Languages))
FROM NLanguage;
```

Country	**CURSOR(SELECT...)**
CH	CURSOR STATEMENT : 2
NAME	**PERCENTAGE**
French	18
German	65
Italian	12
Romansch	1

⇒ Cursore etc. in PL/SQL.

ARBEITEN MIT GESCHACHTELTEN TABELLEN

```
SELECT Country, Name -- UNZULÄSSIG !!
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country);
```

```
SELECT Country, Name
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE A.Country = 'CH'; -- jetzt zulässig.
```

Mit Tabelle *All_Languages*, die alle Sprachen enthält:

```
SELECT Country, Name
FROM NLanguage, All_Languages
WHERE Name IN
  (SELECT Name
   FROM TABLE (Languages));
```

Fazit: Wertebereich von geschachtelten Tabellen muss in *einer* Tabelle zugreifbar sein.

KOMPLEXE DATENTYPEN

```
SELECT * FROM USER_TYPES
```

Type_name	Type_oid	Typecode	Attributes	Methods	Pre	Inc
GeoCoord	_	Object	2	0	NO	NO
Language_T	_	Object	2	0	NO	NO
Languages_List	_	Collection	0	0	NO	NO

Löschen: `DROP TYPE [FORCE]`
Mit `FORCE` kann ein Typ gelöscht werden, dessen Definition von anderen Typen noch gebraucht wird.

Szenario von oben:
`DROP TYPE Language_T`
“Typ mit abhängigen Typen oder Tabellen kann nicht gelöscht oder ersetzt werden”

`DROP TYPE Language_T FORCE` löscht `Language_T`, allerdings

```
SQL> desc Languages_List;
FEHLER: ORA-24372: Ungültiges Objekt für Beschreibung
```