

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: \neq , \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(50),
    Country       VARCHAR2(4),
    Province      VARCHAR2(50),
    Population    NUMBER,
    Latitude      NUMBER,
    Longitude     NUMBER );
```

Die so erzeugten Tabellen- und Spaltennamen sind case-insensitive.

Randbemerkung: case-sensitive Spaltennamen

Falls man case-sensitive Spaltennamen benötigt, kann man dies mit doppelten Anführungszeichen erreichen:

```
CREATE TABLE "Bla"
  ("a" NUMBER,
   "A" NUMBER);
desc "Bla";
insert into "Bla" values(1,2);
select "a" from "Bla";    -> 1
select "A" from "Bla";    -> 2
select a from "Bla";     -> 2(!)
```

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 isMember
  ( Country      VARCHAR2(4),
    Organization VARCHAR2(12),
    Type         VARCHAR2(50)
                DEFAULT 'member')
```

```
INSERT INTO isMember VALUES
  ('CH', 'EU', 'membership applicant');
INSERT INTO isMember (Land, Organization)
  VALUES ('R', 'EU');
```

| Country | Organization | Type |
|---------|--------------|----------------------|
| CH | EU | membership applicant |
| R | 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.

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

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.
- Angabe von DEFERRABLE: siehe Folie 148 ff.

TABELLENDEFINITION: CHECK CONSTRAINTS

- als Spaltenconstraints: Wertebereichseinschränkung

```
CREATE TABLE City
( Name VARCHAR2(50),
  Population NUMBER CONSTRAINT CityPop
    CHECK (Population >= 0),
  ...);
```

- Als Tabellenconstraints: beliebig komplizierte Integritätsbedingungen an ein Tupel.
 - Economy(Country, GDP, Agriculture, Service, Industry, ...):

```
CREATE TABLE Economy ( ...
  CONSTRAINT gdpcheck
  CHECK (industry + service + agriculture <= 102));
```

- zusammengesetzte Fremdschlüssel: Zusammenhang erzwingen (einzelne NULL-Werte würden die Bedingung nicht verletzen):

Organization(Abbrev., Name, City, Country, Province, ...):

```
CREATE TABLE Organization ( ...
  CONSTRAINT hq
  CHECK ( (City IS NULL and Country IS NULL
    and Province IS NULL)
    OR (City IS NOT NULL and Country IS NOT NULL
    and Province IS NOT NULL)))
```

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(50) NOT NULL UNIQUE,
  Code VARCHAR2(4) PRIMARY KEY);
```

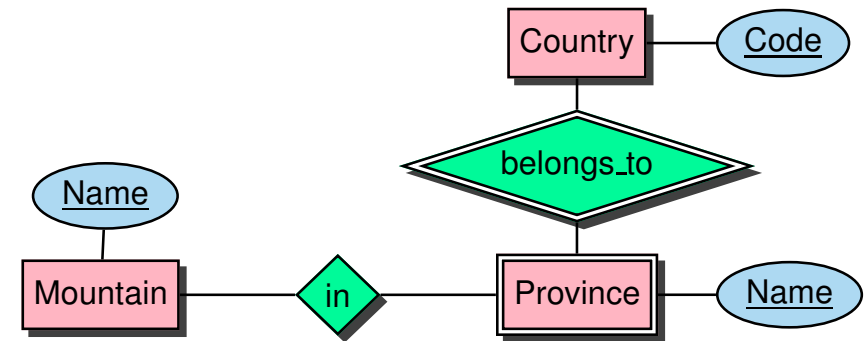
**TABELLENDEFINITION: FOREIGN KEY
...REFERENCES**

- 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 Aktionen, siehe Folie 138 ff.

```
CREATE TABLE isMember
(Country          VARCHAR2(4)
  REFERENCES Country(Code),
 Organization     VARCHAR2(12)
  REFERENCES Organization(Abbreviation),
 Type             VARCHAR2(60) DEFAULT 'member');
```

Tabellendefinition: Fremdschlüssel

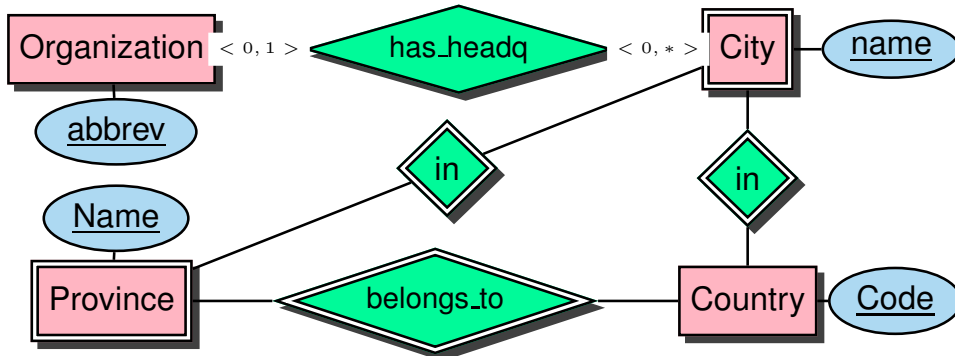
Ein Berg liegt in einer Provinz eines Landes:



```
CREATE TABLE geo_Mountain
( Mountain VARCHAR2(50)
  REFERENCES Mountain(Name),
  Country VARCHAR2(4) ,
  Province VARCHAR2(50) ,
  CONSTRAINT GMountRefsProv
  FOREIGN KEY (Country,Province)
  REFERENCES Province (Country,Name));
```

Tabellendefinition

- Die meisten Organisationen haben ihren Sitz in einer Stadt:



- Organization(Abbrev., Name, City, Country, Province, ...):
- Zusammenhang des Fremdschlüssels erzwingen
- Einzelne Nullwerte würden die FK-Bedingung nicht verletzen:
- INSERT INTO Organization
VALUES ('XX', 'xx', 'Clausthal', 'ZZ', NULL, NULL)

```

CREATE TABLE Organization ( ... ,
    CONSTRAINT orgrefshq
        FOREIGN KEY (City, Country, Province)
            REFERENCES City (Name, Country, Province),
    CONSTRAINT hq
        CHECK ( (City IS NULL and Country IS NULL
                and Province IS NULL)
            OR (City IS NOT NULL and Country IS NOT NULL
                and Province IS NOT NULL)))
    
```

TABELLENDEFINITION

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

```

CREATE TABLE City
( Name VARCHAR2(50),
  Country VARCHAR2(4)
    REFERENCES Country(Code),
  Province VARCHAR2(50),      -- + <tableConstraint>
  Population NUMBER CONSTRAINT CityPop
    CHECK (Population >= 0),
  Latitude NUMBER CONSTRAINT CityLat
    CHECK ((Latitude >= -90) AND (Latitude <= 90)),
  Longitude NUMBER CONSTRAINT CityLong
    CHECK ((Longitude > -180) AND (Longitude <= 180)),
  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.

PAPIERKORB/RECYCLEBIN

Seit Version 11 besitzt Oracle einen *Recyclebin*, wo alles reinfällt, was gedroppt wurde:

- Vorteil: man kann es wiederholen
- Nachteil: es braucht weiterhin Platz im Tablespace.
- Inhalt anschauen (vgl. Data Dictionary: all_objects)

```
SELECT type, object_name, original_name  
FROM RECYCLEBIN;
```

- (Etwas aus) Recyclebin löschen:

```
PURGE RECYCLEBIN;  
PURGE TABLE <tablename>;
```

- Tabelle dropen und nicht im Recyclebin sichern:

```
DROP <tablename> PURGE;
```

- Tabelle wieder holen:

```
FLASHBACK TABLE <tablename>  
TO {BEFORE DROP | TIMESTAMP <timestamp>}  
[ RENAME TO <name>] ;
```

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,
    Longitude = NULL
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 Insert/Update: Merge (Upsert)

Ziel: Wert einer oder mehrerer Spalten setzen, wenn nicht bekannt ist, ob das Tupel (d.h. der Schlüsselwert) bereits existiert.

- falls es existiert: Spalteninhalt setzen,
 - falls es nicht existiert: neues Tupel anlegen.
- ⇒ kann nicht mit einfachen SQL Updates ausgedrückt werden,
- ⇒ kombiniertes Statement "MERGE" (auch als "UPSERT" bezeichnet) seit SQL 2003.

```
MERGE INTO <target_table>
USING <source_relation>
ON (<condition>)
WHEN MATCHED THEN UPDATE
    SET <col1> = <expr1>, ..., <coln> = <exprn>
WHEN NOT MATCHED THEN
    INSERT (<col'1>, ..., <col'm>)
    VALUES (<expr'1>, ..., <expr'm>);
```

- <source_relation> ist DUAL, wenn Konstanten eingesetzt werden sollen,
- <expr_i>, <expr'_i> sind Konstanten oder Ausdrücke über den Spaltennamen von <source_relation>.

Merge: mit konstanten Werten

```

MERGE INTO country
  USING DUAL
  ON (code = 'WAN')
  WHEN MATCHED THEN UPDATE
    SET population = 152217341
  WHEN NOT MATCHED THEN
    INSERT (name, code, population)
    VALUES ('Nigeria', 'WAN', 152217341);

```

Merge: aus anderer Tabelle

- Tabelle NewCountryPops enthält aktuelle Werte für Einwohnerzahlen (evtl. auch neue Länder)

```

CREATE TABLE NewCountryPops (name VARCHAR2(50),
  code VARCHAR2(4), population NUMBER);
INSERT INTO NewCountryPops VALUES('Nigeria', 'WAN', 152217341);
INSERT INTO NewCountryPops VALUES('Lummerland', 'LU', 4);

```

```

MERGE INTO country c
  USING newCountryPops n
  ON (c.code = n.code)
  WHEN MATCHED THEN UPDATE
    SET population = n.population
  WHEN NOT MATCHED THEN
    INSERT (name, code, population)
    VALUES (n.name, n.code, n.population);
SELECT * FROM country WHERE code IN ('LU','WAN');

```

- <source relation> kann eine Tabelle oder eine Subquery sein.
- Die in der ON-Klausel angegebenen Attribute müssen eindeutig ein Tupel der Quell- (logisch, sonst wäre nicht klar welcher Wert eingesetzt werden muss) und Zieltabelle (wäre nicht notwendig) spezifizieren, sonst:
ORA-30926: unable to get a stable set of rows in the source tables

5.5 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.
- Eine Tabelle, auf die noch eine REFERENCES-Deklaration zeigt, wird mit DROP TABLE <table> CASCADE CONSTRAINTS gelöscht.
- Beim Einfügen, Löschen oder Verändern eines referenzierten Tupels muss die referentielle Integrität gewährleistet sein.
(Weiteres dazu später, siehe Folie 148).

5.6 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, COMMIT scheitert, wenn Integritätsbedingungen verletzt sind (dann wird automatisch ein ROLLBACK ausgeführt).
- ROLLBACK [TO <savepoint>] nimmt alle Änderungen [bis zu <savepoint>] zurück,
- Auto-COMMIT in folgenden Situationen:
 - DDL-Anweisung (z.B. CREATE, DROP, RENAME, ALTER),
 - Benutzer meldet sich von ORACLE ab.
- Auto-ROLLBACK in folgenden Situationen:
 - Abbruch eines Benutzerprozesses.

Kapitel 6 Spezialisierte Datentypen

6.1 Datums- und Zeitangaben

- (einfache) Built-In-Typen: Zeitangaben
- zusammengesetzte benutzerdefinierte Datentypen (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 241)
 - 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 393) [seit Oracle 9i-2/2002]
 - Ergänzungen durch “DataBlades”, “Extensions” (Spatial Data (seit Oracle 8i/8.1.5) etc.)

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';
```

| Country | Independence |
|---------|--------------|
| MC | 01 01 1419 |
| NL | 01 01 1579 |
| E | 01 01 1492 |
| THA | 01 01 1238 |

Verwendung von Zeitangaben

- SYSDATE liefert das aktuelle Datum.

```
ALTER SESSION SET NLS_DATE_FORMAT = "hh:mi:ss";
SELECT SYSDATE FROM DUAL;
```

| |
|----------------|
| SYSDATE |
| 10:50:43 |

- Funktion

```
EXTRACT (
    { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
    | { TIMEZONE_HOUR | TIMEZONE_MINUTE }
    | { TIMEZONE_REGION | TIMEZONE_ABBR }
FROM { datevalue | intervalvalue } )
```

Beispiel: Alle Länder, die zwischen 1988 und 1992 gegründet wurden:

```
SELECT Country, EXTRACT(MONTH FROM Independence),
       EXTRACT(YEAR FROM Independence)
FROM Politics
WHERE EXTRACT(YEAR FROM Independence)
      BETWEEN 1988 AND 1992;
```

| Country | EXTR... | EXTR... |
|---------|---------|---------|
| MK | 9 | 1991 |
| SLO | 6 | 1991 |
| : | : | : |

Rechnen mit Datumswerten

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

- 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 zwei Daten liegen.

```
SELECT MONTHS_BETWEEN(LAST_DAY(D1), LAST_DAY(D2))
FROM (SELECT independence as D1 FROM politics
      WHERE country='R'),
      (SELECT independence as D2 FROM politics
      WHERE country='UA');
```

| |
|----------------------------|
| MONTHS_BETWEEN(...) |
| -4 |

Formattoleranz

- NLS_date_format ist verbindlich für das Ausgabeformat
- für das Eingabeformat wendet Oracle zusätzlich Heuristiken an:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';
-- die folgenden beiden werden erkannt:
SELECT to_char(to_date('24.12.2002')) FROM dual;
SELECT to_char(to_date('24 JUN 2002')) FROM dual;
-- das wird nicht erkannt:
SELECT to_char(to_date('JUN 24 2002')) FROM dual;
-- ORA-01858: a non-numeric character was found
-- where a numeric was expected
```

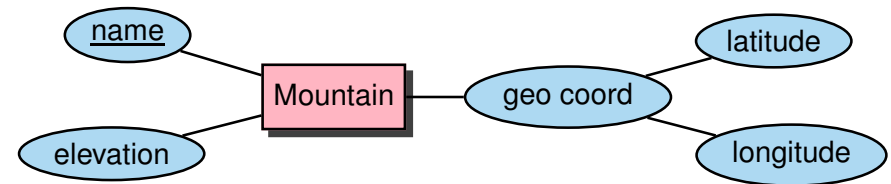
```
ALTER SESSION SET NLS_DATE_FORMAT = 'MON DD YYYY';
SELECT to_char(to_date('JUN 24 2002')) FROM dual;
```

Explizite Formatvorgabe im Einzelfall

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';
SELECT to_char(to_date('JUN 24 2002','MON DD YYYY'))
FROM dual;
-- 24 06 2002
SELECT to_char(to_date('JUN 24 2002','MON DD YYYY'),
              'MM/DD-YYYY')
FROM dual;
-- 06/24-2002
```

6.2 Zusammengesetzte Datentypen

- “First Normal Form”: nur atomare Werte
- Erweiterung I: Strukturierte Werte
- Syntaktisch elegant in SQL umsetzbar



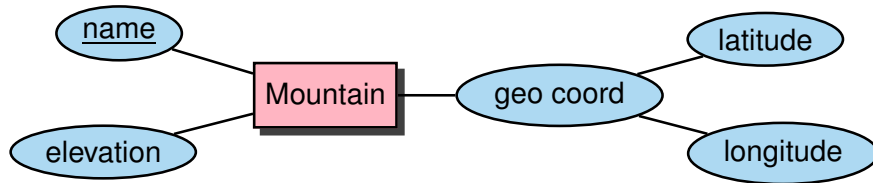
Neue Klasse von Schemaobjekten: CREATE TYPE

- CREATE [OR REPLACE] TYPE <name> AS OBJECT
 (<attr> <datatype>,
 :
 <attr> <datatype>);
 / ← dieser Slash ist unbedingt notwendig!

- 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
```

```
( Latitude NUMBER,
  Longitude NUMBER);
```

/ ← dieser Slash ist unbedingt notwendig!

```
CREATE TABLE Mountain
( Name          VARCHAR2(50),
  Elevation     NUMBER,
  Coordinates   GeoCoord);
```

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

```
INSERT INTO Mountain
VALUES ('Feldberg', 1493, GeoCoord(47.5, 7.5));
SELECT * FROM Mountain;
```

| Name | Elevation | Coordinates(Latitude, Longitude) |
|----------|-----------|----------------------------------|
| Feldberg | 1493 | GeoCoord(47.5, 7.5) |

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

| Name | Coordinates.Latitude | Coordinates.Longitude |
|----------|----------------------|-----------------------|
| Feldberg | 47.5 | 7.5 |

Constraints in zusammengesetzten Datentypen:

```
CREATE TABLE Mountain
( Name          VARCHAR2(50),
  Elevation     NUMBER,
  Coordinates   GeoCoord,
  CHECK ((Coordinates.Latitude >= -90) AND
        (Coordinates.Latitude <= 90)),
  CHECK ((Coordinates.Longitude > -180) AND
        (Coordinates.Longitude <= 180)));
```

6.3 Collections

- “First Normal Form”: nur atomare Werte
- Erweiterung II: Collections:
Wert eines Attributs ist eine Menge
- ... es geht, aber die Syntax dafür wird umständlich und durchbricht die eleganten Einfachheit von SQL und fällt in eine häßliche Programmiersprachenebene.

| NestedPolitics | | | |
|----------------|--------------|------|---------------------|
| Country | Independence | Dep. | Memberships |
| D | 18-JAN-1871 | NULL | EU, NATO, OECD, ... |
| GBJ | NULL | GB | ∅ |
| ⋮ | ⋮ | ⋮ | ⋮ |

- Collection kann durch Aggregation aus einem GROUP-BY gebildet werden:

```
SELECT country, collect(organization)
FROM isMember
GROUP BY country;
```
- Ergebnis z.B. SYSTPkEqWcRtkgT/gQEyGzFEpmA==('EU', 'NATO', 'OECD', ...)
- erzeugt ad-hoc einen systemeigenen Typ “SYSTP...”, der die Collection aufnimmt.

Tabellen mit Collections erzeugen

Verwendet eine einfache Form des etwas komplexeren Konzeptes “Nested Tables” (siehe Folie 119 ff.)

```
CREATE [OR REPLACE] TYPE <collection_type> AS
    TABLE OF <basic_type>;
/
CREATE TABLE <table_name>
    (... ,
        <collection_attr> <collection_type> ,
        ... )
    NESTED TABLE <collection_attr> STORE AS <name >;
```

TABLE-Typ MON_ORGLIST definieren:

```
CREATE OR REPLACE
    TYPE MON_ORGLIST AS TABLE OF VARCHAR2(12);
/
CREATE TABLE NestedPolitics
    ( country VARCHAR2(4) PRIMARY KEY,
      independence DATE,
      dependent VARCHAR2(4), -- REFERENCES Country(Code)
      memberships MON_ORGLIST)
    NESTED TABLE memberships STORE AS o_list;
```

Tabellen mit Collections füllen (1)

- explizit unter Verwendung der entsprechenden Konstruktormethode:

```
INSERT INTO NestedPolitics
VALUES('BAV', '01-APR-2010',
      NULL, MON_ORGLIST('EU','OECD'));
INSERT INTO NestedPolitics
VALUES('SYLT', NULL, 'D', MON_ORGLIST());
```

- eine leere Tabelle ist etwas anderes als NULL.
- ⇒ damit wird es schwieriger, herauszufinden welche Länder nirgends Mitglied sind!
- man kann keine Bedingungen für die in einer Collection erlaubten Werte formulieren (insb. keine REFERENCES).

Tabellen mit Collections füllen (2)

- collect(...) erzeugt eine Instanz eines ad-hoc-Typs, der Zeichenketten (oder Zahlen oder DATE) enthält,
- man muss (leider) explizit mitteilen, dass diese in den Zieltyp (hier MON_ORGLIST) **gecastet** werden muss:
CAST(<instanz-eines-typs> AS <kompatibler typ>)

```
INSERT INTO NestedPolitics
( SELECT p.country, p.independence, p.dependent,
  CAST(collect(i.organization) AS MON_ORGLIST)
  FROM Politics p LEFT OUTER JOIN isMember i
  ON p.country = i.country
  GROUP BY p.country, p.independence, p.dependent);
SELECT country, memberships
FROM NestedPolitics
WHERE country = 'D';
```

| Country | Organizations |
|---------|--|
| 'D' | MON_ORGLIST('EU', 'NATO', 'OECD', ...) |

- Solche Instanzen können mit “=” verglichen werden

```
SELECT a.country, b.country, a.memberships
FROM NestedPolitics a, NestedPolitics b
WHERE a.country < b.country
      AND a.memberships = b.memberships;
```
- ... und sie sind eigentlich kleine, sehr einfache Tabellen ...

Tabellen mit Collections anfragen

Mit [THE|TABLE] (<collection-wertiger Wert>) kann man die Collection wie eine Tabelle verwenden.
(THE ist die schon länger gebräuchliche Syntax)

```
SELECT * FROM TABLE(SELECT memberships
                      FROM NestedPolitics
                      WHERE country = 'D');
```

| COLUMN_VALUE |
|--------------|
| EU |
| NATO |
| OECD |

- Test: mit Konstanten ist nur TABLE, nicht THE erlaubt:
`SELECT * FROM TABLE(MON_ORGLIST('EU', 'NATO'));`
- eine Spalte, die nur den Namen COLUMN_VALUE hat,
- oft als `SELECT column_value as <alias>`.
- Hinweis:
`SELECT * FROM TABLE(SELECT memberships
 FROM NestedPolitics);`
ist nicht zulässig, da es ja mehrere Tabellen wären:
⇒ `single-row subquery returns more than one row`

Tabellen mit Collections anfragen

Mit TABLE(<attrname>) kann auch innerhalb eines Tupels ein collection-wertiges Attribut als Tabelle zugreifbar gemacht werden:
(hier ist THE nicht erlaubt)

- in Subqueries:
`SELECT country
FROM NestedPolitics
WHERE EXISTS (SELECT *
 FROM TABLE(memberships)
 WHERE column_value = 'NATO');`
- oder auch als *korreliertes Join* in der FROM-Zeile:
jede umgebende Zeile mit *ihrer* geschachtelten Tabelle joinen und ausmultiplizieren:
`SELECT country, m.*
 -- oder m.column_value as membership
FROM NestedPolitics, TABLE(memberships) m;`

| Country | COLUMN_VALUE (bzw. membership) |
|---------|--------------------------------|
| D | EU |
| D | NATO |
| D | OECD |
| : | : |

Vergleich mit 1:n- bzw. m:n-Beziehungen als separate Tabelle

- Man sieht relativ einfach, dass die nested table o_list ähnlich der bestehenden “flachen” Tabelle isMember gespeichert ist, und dass

```
SELECT p.country, p.independence, im.organization
FROM Politics p, isMember im
WHERE p.country = im.country;
```

```
SELECT p.country, p.independence, i.organization
FROM Politics p,
-- korreliertes Join, waere z.B. in OQL zulaessig
(SELECT * FROM isMember where country = p.country) i
äquivalent ist.
```

- Anmerkung:
korreliertes Join: i -te Relation in Abhängigkeit von $i - 1$ ter berechnen
 - in SQL nicht erlaubt
 - in Sprachen zu Datenmodellen, die Referenzen/Objektwertige Attribute, mengen-/mehrwertige Attribute oder baumartige Hierarchien besitzen, üblicherweise erlaubt (OQL, XML/XQuery; Forschungs-Sprachen aus 1995-2000: OEM, F-Logic)
 - daher auch für SQL mit Collections naheliegend.

Collection im Ganzen kopieren

```
UPDATE NestedPolitics
SET memberships = (SELECT memberships
                   FROM NestedPolitics
                   WHERE country = 'D')
WHERE country='BAV';
-- optional THE (SELECT ...)
```

Einfügen, Ändern und Löschen mit THE

- Man kann immer nur eine Collection gleichzeitig anfassen, und muss diese mit einer SELECT-Anfrage auswählen (also nicht 'XXX' in alle Mitgliedschaftslisten einfügen, oder überall 'EU' durch 'EWG' ersetzen)

```
INSERT INTO THE (SELECT memberships
                FROM NestedPolitics
                WHERE country = 'D')
VALUES('XXX');
DELETE FROM THE (SELECT memberships
                 FROM NestedPolitics
                 WHERE country = 'D')
WHERE column_value = 'XXX';
UPDATE THE (SELECT memberships
            FROM NestedPolitics
            WHERE country = 'D')
SET column_value = 'XXX'
WHERE column_value = 'EU';
```

6.4 Geschachtelte Tabellen

... zeigen endgültig, wie häßlich die Syntax einer eigentlich schönen Sprache wird, wenn man unbedingt etwas machen will, was im zugrundeliegenden Datenmodell (1. Normalform) nicht möglich ist.

| Nested Languages | | |
|------------------|-----------|---------|
| Country | Languages | |
| | Name | Percent |
| D | German | 100 |
| CH | German | 65 |
| | French | 18 |
| | Italian | 12 |
| | Romansch | 1 |
| FL | NULL | |
| F | French | 100 |
| ⋮ | ⋮ | |

- Tabellenwertige Attribute
- Generischer Typ TABLE OF <inner_type>

⇒ Generische Syntax

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 |

FÜLLEN VON GESCHACHTELTEN TABELLEN

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

collect() über mehrspaltige Tupel nicht erlaubt

```
-- nicht erlaubt:
INSERT INTO NLanguage
  (SELECT country, collect(name,percentage)
   FROM language
   GROUP BY country)
-- PLS-306: wrong number or types of arguments in
-- call to 'SYS_NT_COLLECT'
```

... also anders: Tupelmenge als Tabelle casten

```
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) !!

Füllen von Geschachtelten Tabellen

... also erst Tupel erzeugen und dann die geschachtelten Tabellen hinzufügen:

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

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

ARBEITEN MIT GESCHACHTELTEN TABELLEN

Mit THE und TABLE wie für Collections beschrieben:

- Kopieren ganzer eingebetteter Tabellen mit


```
INSERT INTO ... VALUES(..., THE(SELECT ...),...);
INSERT INTO ... (SELECT ..., THE (SELECT ...)...);
INSERT INTO THE (...) ...;
DELETE FROM THE ( ) ...;
UPDATE THE (...) ...;
```
- TABLE(<attr>) in Unterabfrage:


```
SELECT Country
FROM NLanguage
WHERE 'German' IN (SELECT name
                  FROM TABLE (Languages));
```
- TABLE(<attr>) als korreliertes Join:


```
SELECT Country, n1.*
FROM NLanguage n1, TABLE(n1.Languages) n11;
```

KOMPLEXE DATENTYPEN

```
SELECT * FROM USER_TYPES
```

| Type_name | Type_oid | Typecode | Attrs | Meths |
|----------------|----------|------------|-------|-------|
| GeoCoord | - | Object | 2 | 0 |
| Language_T | - | Object | 2 | 0 |
| Mon_Orglist | - | Collection | 0 | 0 |
| Languages_List | - | Collection | 0 | 0 |

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
```