# 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);
```

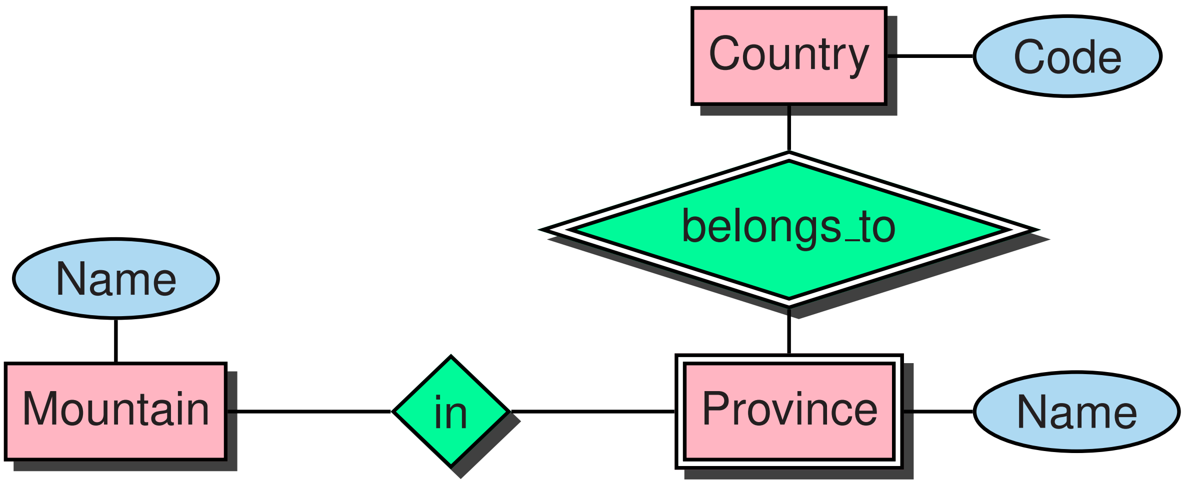
# TABELLENDEFINITION: FOREIGN KEY ...REFERENCES

- 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 201)
  – 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 325) [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';
```

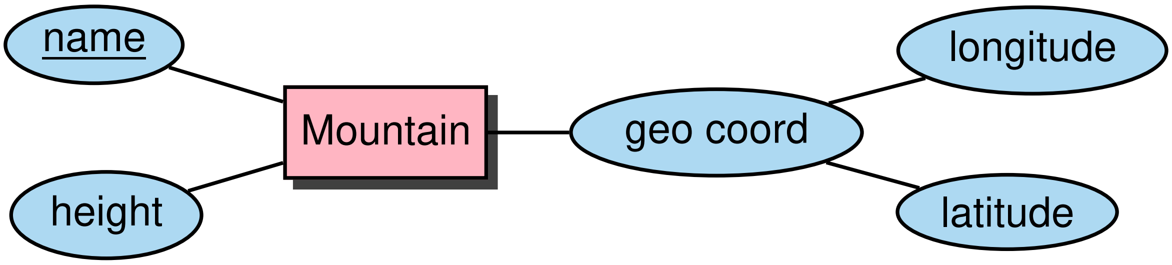
| Land | Datum |
|---|---|
| MC | 01 01 1419 |
| NL | 01 01 1579 |
| E | 01 01 1492 |
| THA | 01 01 1238 |

## 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: $\mathtt{SYSDATE} + 1$ ist morgen, $\mathtt{SYSDATE} + (10/1440)$ ist “in zehn Minuten”.
- $\mathtt{ADD\_MONTHS}(d, n)$ addiert $n$ Monate zu einem Datum $d$.
- $\mathtt{LAST\_DAY}(d)$ ergibt den letzten Tag des in $d$ angegebenen Monats.
- $\mathtt{MONTHS\_BETWEEN}(d_1,d_2)$ gibt an, wieviele Monate zwischen 2 Daten liegen.

# 6.2 Zusammengesetzte Datentypen



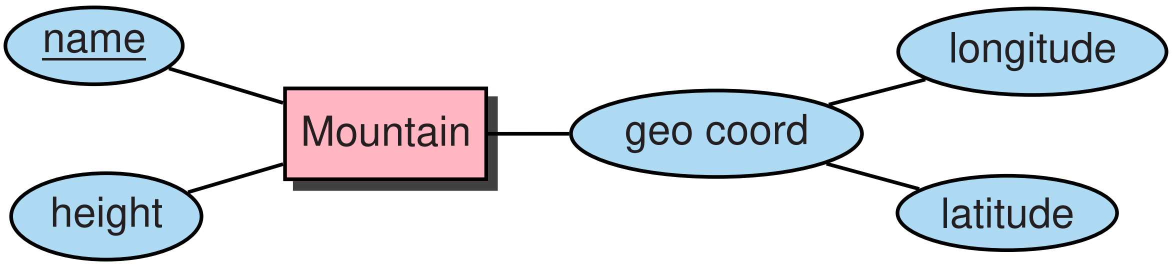

Neue Klasse von Schemaobjekten: `CREATE TYPE`

- ```
  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),<br>Language_T('German', 65),<br>Language_T('Italian', 12),<br>Language_T('Romansch', 1)) |

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

| Languages(Name, Percentage) |
|---|
| Languages_List(Language_T('French', 18),<br>Language_T('German', 65),<br>Language_T('Italian', 12),<br>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
```

# Kapitel 7
# TEIL II: Dies und Das

Teil I: Grundlagen

- ER-Modell und relationales Datenmodell
- Umsetzung in ein Datenbankschema: `CREATE TABLE`
- Anfragen: `SELECT -- FROM -- WHERE`
- Arbeiten mit der Datenbank: `DELETE, UPDATE`

Teil II: Weiteres zum “normalen” SQL

- Änderungen des Datenbankschemas
- Referentielle Integrität
- View Updates
- Zugriffsrechte
- Optimierung

Teil III: Erweiterungen
Prozedurale Konzepte, OO, Einbettung

## 7.1 Ändern von Schemaobjekten

- CREATE-Anweisung
- ALTER-Anweisung
- DROP-Anweisung

- TABLE
- VIEW
- TYPE
- INDEX
- ROLE
- PROCEDURE
- TRIGGER

⋮

# Ändern von Tabellen

- `ALTER TABLE`
- Spalten und Bedingungen hinzufügen,
- bestehende Spaltendeklarationen verändern
- Bedingungen löschen, zeitweise außer Kraft setzen und wieder aktivieren.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ⋮
  DROP <drop-clause>
  DISABLE <disable-clause>
  ⋮
  DISABLE <disable-clause>
  ENABLE <enable-clause>
  ⋮
  ENABLE <enable-clause>;
```

## HINZUFÜGEN VON SPALTEN ZU EINER TABELLE

```
ALTER TABLE <table>
  ADD (<col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
       :
       <col> <datatype> [DEFAULT <value>]
            [<colConstraint> ... <colConstraint>],
       <add table constraints>...)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ... ;
```

Neue Spalten werden mit `NULL`-Werten aufgefüllt.

**Beispiel:** Die Relation *economy* wird um eine Spalte *unemployment* erweitert:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER CHECK (Unemployment > 0));
```

# Hinzufügen von Tabellenbedingungen

```
ALTER TABLE <table>
  ADD (<... add some columns ... >,
       <tableConstraint>,
       ⋮
       <tableConstraint>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ... ;
```

Hinzufügen einer Zusicherung, dass die Summe der Anteile von Industrie, Dienstleistung und Landwirtschaft am Bruttosozialprodukt maximal 100% ist:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER CHECK (Unemployment > 0),
       CHECK (Industry + Service + Agriculture <= 100));
```

- Soll eine Bedingung hinzugefügt werden, die im momentanen Zustand verletzt ist, erhält man eine Fehlermeldung.

```
ALTER TABLE City
  ADD (CONSTRAINT citypop CHECK (Population > 100000));
```

## Spaltendefinitionen einer Tabelle ändern

- Spaltenbedingungen lassen sich nicht durch `ALTER TABLE ... ADD` hinzufügen.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<col> [<datatype>] [DEFAULT <value>]
          [<colConstraint> ... <colConstraint>],
          ⋮
          <col> [<datatype>] [DEFAULT <value>]
          [<colConstraint> ... <colConstraint>])
  DROP <drop-clause>
  ... ;
```

```
ALTER TABLE Country MODIFY (Capital NOT NULL);
ALTER TABLE encompasses
  ADD (PRIMARY KEY (Country,Continent));
ALTER TABLE Desert
  MODIFY Area CONSTRAINT DesertArea CHECK (Area > 10);
```

- Fehlermeldung, falls eine Bedingung formuliert wird, die der aktuelle Datenbankzustand nicht erfüllt.

# ALTER TABLE ... DROP/DISABLE/ENABLE

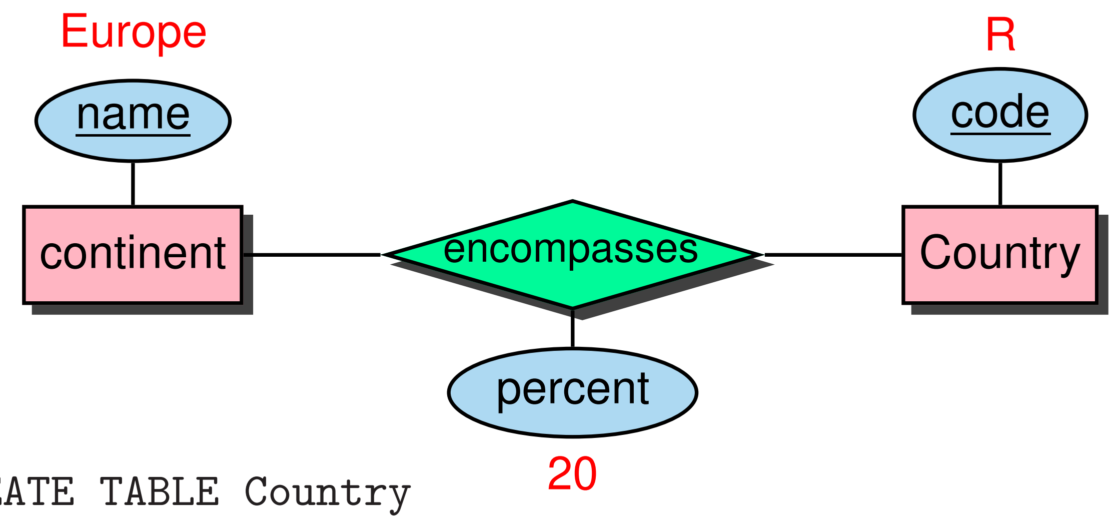
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP    PRIMARY KEY [CASCADE] |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint>
  DISABLE PRIMARY KEY [CASCADE] |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint> | ALL TRIGGERS
  ENABLE  PRIMARY KEY |
          UNIQUE (<column-list>) |
          CONSTRAINT <constraint> | ALL TRIGGERS;
```

- `PRIMARY KEY` darf nicht gelöscht/disabled werden solange `REFERENCES`-Deklaration besteht.
- `DROP PRIMARY KEY CASCADE` löscht/disabled eventuelle `REFERENCES`-Deklarationen ebenfalls.
- `ENABLE`: kaskadierend disable'te Constraints müssen manuell reaktiviert werden.

# 7.2 Referentielle Integrität

Referentielle Integritätsbedingungen treten dort auf, wo bei der Umsetzung vom ER-Modell zum relationalen Modell Schlüsselattribute der beteiligten Entities in Beziehungstypen eingehen (Zusammenhang von Primär- und Fremdschlüsseln):




```
CREATE TABLE Country
 (Name     VARCHAR2(32),
  Code     VARCHAR2(4) PRIMARY KEY,
  ...);

CREATE TABLE Continent
 (Name     VARCHAR2(10) PRIMARY KEY,
  Area     NUMBER(2));

CREATE TABLE encompasses
 (Continent  VARCHAR2(10) REFERENCES Continent(Name),
  Country    VARCHAR2(4) REFERENCES Country(Code),
  Percentage NUMBER);
```

# Referentielle Integrität

| Country | | | |
|---|---|---|---|
| Name | Code | Capital | Province |
| Germany | D | Berlin | Berlin |
| United States | USA | Washington | Distr. Columbia |
| ... | ... | ... | ... |

| City | | |
|---|---|---|
| Name | Country | Province |
| Berlin | D | Berlin |
| Washington | USA | Distr. Columbia |
| ... | ... | ... |

```
FOREIGN KEY (<attr-list>)
REFERENCES <table'> (<attr-list'>)
```

- (<attr-list'>) muss Candidate Key der referenzierten Tabelle sein (NOT NULL UNIQUE).

# Referentielle Integrität

- als Spaltenbedingung:

```
<attr> [CONSTRAINT <name>]
      REFERENCES <table'>(<attr'>)
```

```
CREATE TABLE City
(...
Country VARCHAR2(4)
       CONSTRAINT CityRefsCountry
       REFERENCES Country(Code) );
```

- als Tabellenbedingung:

```
[CONSTRAINT <name>]
  FOREIGN KEY (<attr-list>)
  REFERENCES <table'>(<attr-list'>)
```

```
CREATE TABLE Country
(...
CONSTRAINT CapitalRefsCity
  FOREIGN KEY (Capital,Code,Province)
  REFERENCES City(Name,Country,Province) );
```

# Referentielle Aktionen

- Bei Veränderungen am Inhalt einer Tabelle sollen automatisch Aktionen ausgeführt werden, um die referentielle Integrität der Datenbasis zu erhalten.
- Ist dies nicht möglich, so werden die gewünschten Operationen nicht ausgeführt, bzw. zurückgesetzt.

1. `INSERT` in die referenzierte Tabelle oder `DELETE` aus der referenzierenden Tabelle ist immer unkritisch:

```
INSERT INTO Country
  VALUES ('Lummerland,'LU',...);
DELETE FROM is_member ('D','EU');
```

2. Ein `INSERT` oder `UPDATE` in der referenzierenden Tabelle, darf keinen Fremdschlüsselwert erzeugen, der nicht in der referenzierten Tabelle existiert:

```
INSERT INTO City
  VALUES ('Karl-Marx-Stadt','DDR',...);
```

   Anderenfalls ist es unkritisch:

```
UPDATE City SET Country='A' WHERE Name='Munich';
```

3. `DELETE` und `UPDATE` bzgl. der referenzierten Tabelle: Anpassung der referenzierenden Tabelle durch *Referentielle Aktionen* sinnvoll:

   `UPDATE Country SET Code='UK' WHERE Code='GB';` oder
   `DELETE FROM Country WHERE Code='I';`

# REFERENTIELLE AKTIONEN IM SQL-2-STANDARD

`NO ACTION`:

Die Operation wird zunächst ausgeführt; Nach der Operation wird überprüft, ob “dangling references” entstanden sind und ggf. die Aktion zurückgenommen:

```
DELETE FROM River;
```

Untscheidung zwischen Referenz *River* - *River* und *located* - *River*!

`RESTRICT`:

Die Operation wird nur dann ausgeführt, wenn keine “dangling references” entstehen können:

```
DELETE FROM Organization WHERE ...;
```

Fehlermeldung, wenn eine Organisation gelöscht werden müsste, die Mitglieder besitzt.

`CASCADE`:

Die Operation wird ausgeführt. Die referenzierenden Tupel werden ebenfalls gelöscht bzw. geändert.

```
UPDATE Country SET Code='UK' WHERE Code='GB';
```

ändert überall:

Country: (United Kingdom,GB,. . . ) $\leadsto$
(United Kingdom,UK,. . . )
Province:(Yorkshire,GB,. . . ) $\leadsto$ (Yorkshire,UK,. . . )
City: (London,GB,Greater London,. . . ) $\leadsto$
(London,UK,Greater London,. . . )

# REFERENTIELLE AKTIONEN IM SQL-2-STANDARD

`SET DEFAULT`:

Die Operation wird ausgeführt und bei den referenzierenden Tupeln wird der entsprechende Fremdschlüsselwert auf die für die entsprechende Spalten festgelegten `DEFAULT`-Werte gesetzt (dafür muss dann wiederum ein entsprechendes Tupel in der referenzierten Relation existieren). Falls kein `DEFAULT`-Wert definiert wurde, entspricht das Verhalten `SET NULL` (s.u.).

`SET NULL`:

Die Operation wird ausgeführt und bei den referenzierenden Tupeln wird der entsprechende Fremdschlüsselwert durch `NULL` ersetzt (dazu müssen `NULL`s zulässig sein).

`located`: Stadt liegt an Fluss/See/Meer

located(Bremerhaven,Nds.,D,Weser,NULL,North Sea)

```
DELETE * FROM River WHERE Name='Weser';
```

located(Bremerhaven,Nds.,D,NULL,NULL,North Sea)

## REFERENTIELLE AKTIONEN IM SQL-2-STANDARD

Referentielle Integritätsbedingungen und Aktionen werden bei `CREATE TABLE` und `ALTER TABLE` als

`<columnConstraint>` (für einzelne Spalten)

```
<col> <datatype>
  CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ ON DELETE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
  [ ON UPDATE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
```

oder `<tableConstraint>` (für mehrere Spalten)

```
   CONSTRAINT <name>
     FOREIGN KEY (<attr-list>)
     REFERENCES <table'> (<attr-list'>)
     [ ON DELETE ...]
     [ ON UPDATE ...]
```

angegeben.

# Referentielle Aktionen

| Country | | | |
|---|---|---|---|
| Name | Code | Capital | Province |
| Germany | D | Berlin | Berlin |
| United States | USA | Washington | Distr. Columbia |
| ... | ... | ... | ... |

CASCADE

NO ACTION

| City | | |
|---|---|---|
| Name | Country | Province |
| Berlin | D | Berlin |
| Washington | USA | Distr. Columbia |
| ... | ... | ... |

1. `DELETE FROM City WHERE Name='Berlin';`

2. `DELETE FROM Country WHERE Name='Germany';`

# Referentielle Aktionen in ORACLE:

- ORACLE 9: nur ON DELETE/UPDATE NO ACTION, ON DELETE CASCADE und (seit Oracle 8.1.5) ON DELETE SET NULL implementiert.
- Wird ON ... nicht angegeben, wird NO ACTION als Default verwendet.
- ON UPDATE CASCADE fehlt, was beim Durchführen von Updates ziemlich lästig ist.
- Hat aber so seine Gründe ...

Syntax als <columnConstraint>:

```
CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ON DELETE CASCADE|ON DELETE SET NULL]
```

Syntax als <tableConstraint>:

```
CONSTRAINT <name>
  FOREIGN KEY [ (<attr-list>)]
  REFERENCES <table'> (<attr-list'>)
  [ON DELETE CASCADE|ON DELETE SET NULL]
```

# Referentielle Aktionen: UPDATE ohne CASCADE

**Beispiel:** Umbenennung eines Landes:

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

`('United Kingdom','GB')`

```
CREATE TABLE Province
  ( Name    VARCHAR2(32)
    Country VARCHAR2(4) CONSTRAINT ProvRefsCountry
                        REFERENCES Country(Code));
```

`('Yorkshire','GB')`

Nun soll das Landeskürzel von 'GB' nach 'UK' geändert werden.

- `UPDATE Country SET Code='UK' WHERE Code='GB';`
  $\leadsto$ "dangling reference" des alten Tupels ('Yorkshire','GB').
- `UPDATE Province SET Code='UK' WHERE Code='GB';`
  $\leadsto$ "dangling reference" des neuen Tupels ('Yorkshire','UK').

# REFERENTIELLE AKTIONEN: UPDATE OHNE CASCADE

- referentielle Integritätsbedingung außer Kraft setzen,
- die Updates vornehmen
- referentielle Integritätsbedingung reaktivieren

```
ALTER TABLE Province
  DISABLE CONSTRAINT ProvRefsCountry;

UPDATE Country
  SET Code='UK' WHERE Code='GB';

UPDATE Province
  SET Country='UK' WHERE Country='GB';

ALTER TABLE Province
  ENABLE CONSTRAINT ProvRefsCountry;
```

# REFERENTIELLE INTEGRITÄTSBEDINGUNGEN

Man kann ein Constraint auch bei der Tabellendefinition mitdefinieren, und sofort disablen:

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

# Referentielle Aktionen: Zyklische Referenzen

| Country | | | |
|---|---|---|---|
| Name | Code | Capital | Province |
| Germany | D | Berlin | Berlin |
| United States | US | Washington | Distr.Col. |
| ... | ... | ... | ... |

| Province | | |
|---|---|---|
| Name | Country | Capital |
| Berlin | D | Berlin |
| Distr.Col. | US | Washington |
| ... | ... | ... |

| City | | |
|---|---|---|
| Name | Country | Province |
| Berlin | D | B |
| Washington | USA | Distr.Col. |
| ... | ... | ... |

# Referentielle Aktionen: Problematik

| Country | | | |
|---|---|---|---|
| Name | Code | Capital | Province |
| Germany | D | Berlin | Berlin |
| United States | US | Washington | Distr.Col. |
| ... | ... | ... | ... |

CASCADE

SET NULL

| Province | | |
|---|---|---|
| Name | Country | Capital |
| Berlin | D | Berlin |
| Distr.Col. | US | Washington |
| ... | ... | ... |

CASCADE

| City | | |
|---|---|---|
| Name | Country | Province |
| Berlin | D | B |
| Washington | USA | Distr.Col. |
| ... | ... | ... |

```
DELETE FROM Country
  WHERE Code='D'
```

# Referentielle Aktionen

Im allgemeinen Fall:

- Schon eine einzelne Operation bringt in Verbindung mit `ON DELETE/UPDATE SET NULL/SET DEFAULT` und `ON UPDATE CASCADE` Mehrdeutigkeiten, Widersprüche etc.
- Aufgrund von SQL-Triggern induziert ein User-Update häufig mehrere Datenbank-Updates,
- nichttriviale Entscheidung, welche Updates getriggert werden sollen,
- im Fall von Inkonsistenzen Analyse der Ursache sowie maximal zulässiger Teilmengen der User-Requests,
- Stabile Modelle, exponentieller Aufwand.

... siehe dbis-Webseiten

## 7.3 Views

- Kombination mit der Vergabe von Zugriffsrechten (später)
- Darstellung des realen Datenbestand für Benutzer in einer veränderten Form.

### VIEW UPDATES

- müssen auf Updates der Basisrelation(en) abgebildet werden,
- nicht immer möglich.
- Tabelle USER_UPDATABLE_COLUMNS im Data Dictionary:

```
CREATE VIEW <name> AS ...
```

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = '<NAME>';
```

# VIEW UPDATES

- abgeleitete Werte können nicht verändert werden:

**Beispiel:**

```
CREATE OR REPLACE VIEW temp AS
SELECT Name, Code, Area, Population,
       Population/Area AS Density
FROM Country;
```

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'TEMP';
```

| Table_Name | Column_Name | UPD | INS | DEL |
|---|---|---|---|---|
| temp | Name | yes | yes | yes |
| temp | Code | yes | yes | yes |
| temp | Area | yes | yes | yes |
| temp | Population | yes | yes | yes |
| temp | Density | no | no | no |

```
INSERT INTO temp (Name, Code, Area, Population)
    VALUES ('Lummerland','LU',1,4)
SELECT * FROM temp where Code = 'LU';
```

- analog für Werte die als Ergebnis von Aggregatfunktionen berechnet werden (`COUNT`, `AVG`, `MAX`, ...)

# View Updates

**Beispiel:**

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

```
SELECT * FROM USER_UPDATABLE_COLUMNS
WHERE Table_Name = 'CITYCOUNTRY';
```

| Table_Name | Column_Name | UPD | INS | DEL |
|---|---|---|---|---|
| CityCountry | City | yes | yes | yes |
| CityCountry | Country | no | no | no |

- Städte(namen) können verändert werden:
  direkte Abbildung auf *City*:

  ```
  UPDATE CityCountry
  SET City = 'Wien'
  WHERE City = 'Vienna';
  ```

  ```
  SELECT * FROM City WHERE Country = 'A';
  ```

  | Name | Country | Province | ... |
  |---|---|---|---|
  | Wien | A | Vienna | ... |
  | ⋮ | ⋮ | ⋮ | ⋮ |

# VIEW UPDATES

**Beispiel:**

- *Country* darf nicht verändert werden:

| City | Country |
|---|---|
| Berlin | Germany |
| Freiburg | Germany |

Umsetzung auf Basistabelle wäre nicht eindeutig:

```
UPDATE CityCountry            UPDATE CityCountry
SET Country = 'Poland'        SET Country = 'Deutschland'
WHERE City = 'Berlin';        WHERE Country = 'Germany';
```

Nur in *City* werden die Tupel gelöscht:

```
DELETE FROM CityCountry       DELETE FROM CityCountry
WHERE City = 'Berlin';        WHERE Country = 'Germany';
```

# VIEW UPDATES

- ORACLE: Zulässigkeitsentscheidung durch Heuristiken
- basieren nur auf Schemainformation,
- nicht auf *aktuellem* Datenbankzustand !
- Schlüsseleigenschaften wichtig: Schlüssel einer Basistabelle müssen im View erhalten bleiben.
- Schlüssel einer Basistabelle = Schlüssel des Views: Abbildung möglich.
- Schlüssel einer Basistabelle $\supseteq$ ein Schlüssel des Views: Umsetzung möglich.
  (Eventuell mehrere Tupel der Basistabelle betroffen).
- Schlüssel einer Basistabelle überdeckt keinen Schlüssel des Views: i.a. keine Umsetzung möglich (siehe Aufgaben).

# VIEW UPDATES

**Beispiel:**

```
CREATE OR REPLACE VIEW temp AS
SELECT country, population
FROM Province A
WHERE population = (SELECT MAX(population)
                    FROM Province B
                    WHERE A.Country = B.Country);
```

```
SELECT * FROM temp WHERE Country = 'D';
```

| Country | Name | Population |
|---|---|---|
| D | Nordrhein-Westfalen | 17816079 |

```
UPDATE temp
SET population = 0 where Country = 'D';
SELECT * FROM Province WHERE Name = 'D';
```

Ergebnis: die Bevölkerung der bevölkerungsreichsten Provinz Deutschlands wird auf 0 gesetzt. Damit ändert sich auch das View !

```
SELECT * FROM temp WHERE Country = 'D';
```

| Country | Name | Population |
|---|---|---|
| D | Bayern | 11921944 |

# View Updates

- Tupel können durch Update aus dem Wertebereich des Views hinausfallen.
- Views häufig verwendet, um den "Aktionsradius" eines Benutzers einzuschränken.
- Verlassen des Wertebereichs kann durch `WITH CHECK OPTION` verhindert werden:

**Beispiel**

```
CREATE OR REPLACE VIEW UScities AS
SELECT *
FROM City
WHERE Country = 'USA'
WITH CHECK OPTION;

UPDATE UScities
SET Country = 'D' WHERE Name = 'Miami';
```

```
FEHLER in Zeile 1:
ORA-01402: Verletzung der WHERE-Klausel
           einer View WITH CHECK OPTION
```

Es ist übrigens erlaubt, Tupel aus dem View zu *löschen*.

## MATERIALIZED VIEWS

- Views werden bei jeder Anfrage neu berechnet.

\+ repräsentieren immer den aktuellen Datenbankzustand.

\- zeitaufwendig, ineffizient bei wenig veränderlichen Daten

⇒ *Materialized Views*

- werden bei der Definition berechnet und
- bei jeder Datenänderung automatisch aktualisiert (u.a. durch *Trigger*).
- ⇒ Problem der *View Maintenance*.

# 7.4 Zugriffsrechte

## Benutzeridentifikation

- Benutzername
- Password
- `sqlplus /`: Identifizierung durch UNIX-Account

## Zugriffsrechte innerhalb ORACLE

- Zugriffsrechte an ORACLE-Account gekoppelt
- initial vom DBA vergeben

## Schemakonzept

- Jedem Benutzer ist sein *Database Schema* zugeordnet, in dem "seine" Objekte liegen.
- Bezeichnung der Tabellen *global* durch
  `<username>.<table>`
  (z.B. `dbis.City`),
- im eigenen Schema nur durch `<table>`.

# Systemprivilegien

- berechtigen zu Schemaoperationen
- `CREATE [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte erzeugen,
- `ALTER [ANY] TABLE/TYPE/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte verändern,
- `DROP [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte löschen.
- `SELECT/INSERT/UPDATE/DELETE [ANY] TABLE`: Benutzer darf in Tabellen Tupel lesen/erzeugen/verändern/entfernen.

- `ANY`: Operation in *jedem* Schema erlaubt,
- ohne `ANY`: Operation nur im eigenen Schema erlaubt

Praktikum:

- `CREATE SESSION, ALTER SESSION, CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE PROCEDURE...`
- Zugriffe und Veränderungen an den eigenen Tabellen nicht explizit aufgeführt (`SELECT TABLE`).

## SYSTEMPRIVILEGIEN

```
GRANT <privilege-list>
TO <user-list> | PUBLIC [ WITH ADMIN OPTION ];
```

- `PUBLIC`: jeder erhält das Recht.
- `ADMIN OPTION`: Empfänger darf dieses Recht weiter vergeben.

Rechte entziehen:

```
REVOKE <privilege-list> | ALL
FROM <user-list> | PUBLIC;
```

nur wenn man dieses Recht selbst vergeben hat (im Fall von `ADMIN OPTION` kaskadierend).

**Beispiele:**

- ```
  GRANT CREATE ANY INDEX, DROP ANY INDEX
  TO opti-person WITH ADMIN OPTION;
  ```
  erlaubt `opti-person`, überall Indexe zu erzeugen und zu löschen,
- ```
  GRANT DROP ANY TABLE TO destroyer;
  GRANT SELECT ANY TABLE TO supervisor;
  ```
- ```
  REVOKE CREATE TABLE FROM mueller;
  ```

Informationen über Zugriffsrechte im Data Dictionary:

```
SELECT * FROM SESSION_PRIVS;
```
```

# OBJEKTPRIVILEGIEN

berechtigen dazu, Operationen auf existierenden Objekten auszuführen.

- Eigentümer eines Datenbankobjektes
- Niemand sonst darf mit einem solchen Objekt arbeiten, außer
- Eigentümer (oder DBA) erteilt explizit entsprechende Rechte:

```
GRANT <privilege-list> | ALL [(<column-list>)]
ON <object>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

- `<object>`: `TABLE`, `VIEW`, `PROCEDURE/FUNCTION`, `TYPE`,
- Tabellen und Views: Genauere Einschränkung für `INSERT`, `REFERENCES` und `UPDATE` durch `<column-list>`,
- `<privilege-list>`: `DELETE`, `INSERT`, `SELECT`, `UPDATE` für Tabellen und Views,
  `INDEX`, `ALTER` und `REFERENCES` für Tabellen,
  `EXECUTE` für Prozeduren, Funktionen und `TYPE`n.
- `ALL`: alle Privilegien die man an dem beschriebenen Objekt (ggf. auf der beschriebenen Spalte) hat.
- `GRANT OPTION`: Der Empfänger darf das Recht weitergeben.

# OBJEKTPRIVILEGIEN

Rechte entziehen:

```
REVOKE <privilege-list> | ALL
ON <object>
FROM <user-list> | PUBLIC
[CASCADE CONSTRAINTS];
```

- CASCADE CONSTRAINTS (bei REFERENCES): alle referentiellen Integritätsbedingungen, die auf einem entzogenen REFERENCES-Privileg beruhen, fallen weg.
- Berechtigung von mehreren Benutzern erhalten: Fällt mit dem letzten REVOKE weg.
- im Fall von GRANT OPTION kaskadierend.

Überblick über vergebene/erhaltene Rechte:

```
SELECT * FROM USER_TAB_PRIVS;
```

- Rechte, die man für eigene Tabellen vergeben hat,
- Rechte, die man für fremde Tabellen bekommen hat

```
SELECT * FROM USER_COL_PRIVS;
SELECT * FROM USER_TAB/COL_PRIVS_MADE/RECD;
```

Stichwort: Rollenkonzept

# Synonyme

Schemaobjekt unter einem anderen Namen als ursprünglich abgespeichert ansprechen:

```
CREATE [PUBLIC] SYNONYM <synonym>
  FOR <schema>.<object>;
```

- Ohne `PUBLIC`: Synonym ist nur für den Benutzer definiert.
- `PUBLIC` ist das Synonym systemweit verwendbar. Geht nur mit `CREATE ANY SYNONYM`-Privileg.

**Beispiel:** Benutzer will oft die Relation “City”, aus dem Schema “dbis” verwenden.

- `SELECT * FROM dbis.City;`
- ```
  CREATE SYNONYM DCity
    FOR dbis.City;

  SELECT * FROM DCity;
  ```

Synonyme löschen: `DROP SYNONYM <synonym>;`
  ```

## ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS

- GRANT SELECT kann nicht auf Spalten eingeschränkt werden.
- Stattdessen: Views verwenden.

```
GRANT SELECT [<column-list>]   -- nicht erlaubt
ON <table>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

kann ersetzt werden durch

```
CREATE VIEW <view> AS
  SELECT <column-list>
  FROM <table>;
```

```
GRANT SELECT
ON <view>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

# ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS: BEISPIEL

*pol* ist Besitzer der Relation *Country*, will *Country* ohne Hauptstadt und deren Lage für *geo* les- und schreibbar machen.

View mit Lese- und Schreibrecht für *geo*:

```
CREATE VIEW pubCountry AS
SELECT Name, Code, Population, Area
FROM Country;

GRANT SELECT, INSERT, DELETE, UPDATE
      ON pubCountry TO geo;
```

- Referenzen auf Views müssen separat erlaubt werden:

```
<pol>: GRANT REFERENCES (Code) ON Country TO geo;
<geo>: ... REFERENCES pol.Country(Code);
```

## 7.5 Optimierung der Datenbank

- möglichst wenige Hintergrundspeicherzugriffe
- Daten soweit wie möglich im Hauptspeicher halten

Datenspeicherung:

- Hintergrundspeicherzugriff effizient steuern
  $\longrightarrow$ Zugriffspfade: Indexe, Hashing
- möglichst viele semantisch zusammengehörende Daten mit *einem* Hintergrundspeicherzugriff holen
  $\longrightarrow$ Clustering

Anfrageoptimierung:

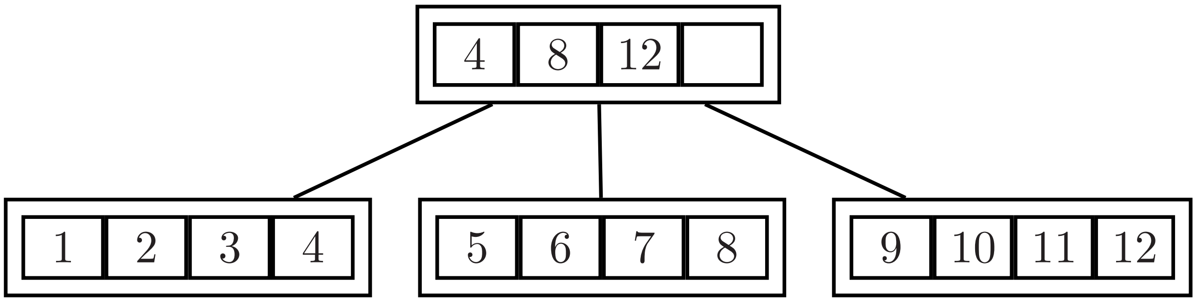
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

Algorithmische Optimierung !

# ZUGRIFFSPFADE: INDEXE

Zugriff über indizierte Spalte(n) erheblich effizienter.

- Baumstruktur; ORACLE: B$^*$-Mehrweg-Baum,
- B$^*$-Baum: Knoten enthalten *nur* Weg-Information, Verzweigungsgrad hoch, Höhe des Baumes klein.




- Suche durch Schlüsselvergleich: logarithmischer Aufwand.
- Schneller Zugriff (logarithmisch) versus hoher Reorganisationsaufwand ($\rightarrow$ Algorithmentechnik),
- bei sehr vielen Indexen auf einer Tabelle kann es beim Einfügen, Ändern und Löschen von Sätzen zu Performance-Verlusten kommen,
- logisch und physikalisch unabhängig von den Daten der zugrundeliegenden Tabelle,
- keine Auswirkung auf die *Formulierung* einer SQL-Anweisung, nur auf die *interne* Auswertung,
- mehrere Indexe für eine Tabelle möglich.

# ZUGRIFFSPFADE: INDEXE

Zugriff über indizierte Spalte(n) erheblich effizienter:

- benötigte Indexknoten aus Hintergrundspeicher holen,
- dann nur ein Zugriff um ein Tupel zu bekommen.

```
SELECT Name, Code FROM Country WHERE Code > 'M';
```

- Ausgabe alphabetisch nach Code geordnet:
  Auf Schlüsselattribut ist automatisch ein Index angelegt und wird verwendet.

```
SELECT Name, Population
FROM Country
WHERE Population > 50000000;
```

- Ausgabe nicht sinnvoll geordnet:
  kein Index vorhanden, linearer Durchlauf ("Scan").

```
CREATE INDEX CountryPopIndex ON Country (Population);
SELECT Name, Population
FROM Country
WHERE Population > 50000000;
```

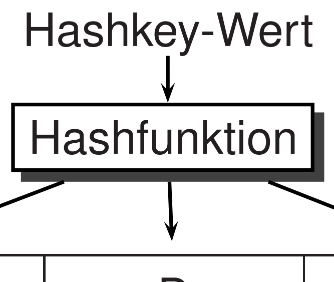
- Ausgabe jetzt nach Population geordnet.
  (Zugriff auf 50000000 über Baum, Rest linear ausgeben)

```
DROP INDEX CountryPopIndex;
```

# HASHING

Aufgrund der Werte einer/mehrerer Spalten (*Hashkey*) wird durch eine *Hashfunktion* berechnet, wo das/die entsprechende(n) Tupel zu finden sind.

- Zugriff in *konstanter* Zeit,
- keine Ordnung.
- gezielter Zugriff auf die Daten über ein bestimmtes Land
  Hashkey: Country.Code




| | | | | | | |
|---|---|---|---|---|---|---|
| ... | F | ... | D | ... | NL | ... |
| | 58317450 | | 83536115 | | 15568034 | |
| | 547030 | | 356910 | | 37330 | |

In ORACLE ist Hashing nur für *Cluster* implementiert.

# CLUSTER

- Zusammenfassung einer Gruppe von Tabellen, die alle eine oder mehrere gemeinsame Spalten (Clusterschlüssel) besitzen, oder
- Gruppierung einer Tabelle nach dem Wert einer bestimmten Spalte (Clusterschlüssel);
- bei einem Hintergrundspeicherzugriff werden semantisch zusammengehörende Daten in den Hauptspeicher geladen.

**Vorteile eines Clusters:**

- geringere Anzahl an Plattenzugriffen und schnellere Zugriffsgeschwindigkeit
- geringerer Speicherbedarf, da jeder Clusterschlüsselwert nur einmal abgespeichert wird

**Nachteile:**

- ineffizient bei häufigen Updates der Clusterschlüsselwerte, da dies eine physikalische Reorganisation bewirkt
- schlechtere Performance beim Einfügen in Cluster-Tabellen

# CLUSTERING

Sea und geo_Sea mit Clusterschlüssel `Sea.Name`:

| Cl_Sea | | |
|---|---|---|
| Mediterranean Sea | Depth | |
| | 5121 | |
| | Province | Country |
| | Catalonia | E |
| | Valencia | E |
| | Murcia | E |
| | Andalusia | E |
| | Languedoc-R. | F |
| | Provence | F |
| | ⋮ | ⋮ |
| Baltic Sea | Depth | |
| | 459 | |
| | Province | Country |
| | Schleswig-H. | D |
| | Mecklenb.-Vorp. | D |
| | Szczecin | PL |
| | ⋮ | ⋮ |

# CLUSTERING

*City* nach (*Province,Country*):

| Country | Province | | | |
|---|---|---|---|---|
| D | Nordrh.-Westf. | City | Population | ... |
| | | Düsseldorf. | 572638 | ... |
| | | Solingen | 165973 | ... |
| USA | Washington | City | Population | ... |
| | | Seattle | 524704 | ... |
| | | Tacoma | 179114 | ... |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

# ERZEUGEN EINES CLUSTERS IN ORACLE

Cluster erzeugen und Clusterschlüssel angeben:

```
CREATE CLUSTER <name>(<col> <datatype>-list)
  [INDEX | HASHKEYS <integer> [HASH IS <funktion>]];
```

```
CREATE CLUSTER Cl_Sea (SeaName VARCHAR2(25));
```

Default: *indexed Cluster*, d.h. die Zeilen werden entsprechend dem Clusterschlüsselwert indiziert und geclustert.
Option: `HASH` mit Angabe einer Hashfunktion, nach der geclustert wird.

↪

# ERZEUGEN EINES CLUSTERS IN ORACLE

Zuordnung der Tabellen mit `CREATE TABLE` unter Angabe des Clusterschlüssels.

```
CREATE TABLE <table>
  (<col> <datatype>,
   ⋮
   <col> <datatype>)
  CLUSTER <cluster>(<column-list>);
```

```
CREATE TABLE CSea
  (Name    VARCHAR2(25) PRIMARY KEY,
   Depth   NUMBER)
  CLUSTER Cl_Sea (Name);
```

```
CREATE TABLE Cgeo_Sea
  (Province  VARCHAR2(32),
   Country   VARCHAR2(4),
   Sea       VARCHAR2(25))
  CLUSTER Cl_Sea (Sea);
```

Erzeugen des Clusterschlüsselindexes:
(Dies muss *vor* dem ersten DML-Kommando geschehen).

```
CREATE INDEX <name> ON CLUSTER <cluster>;
```

```
CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```