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

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

TABELLENDEFINITION

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(!)  Default ist CAPITALS (siehe auch alle Ausgaben
                                    von Anfragen an das Data Dictionary)
```

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	**Org.**	**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 156 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);
```

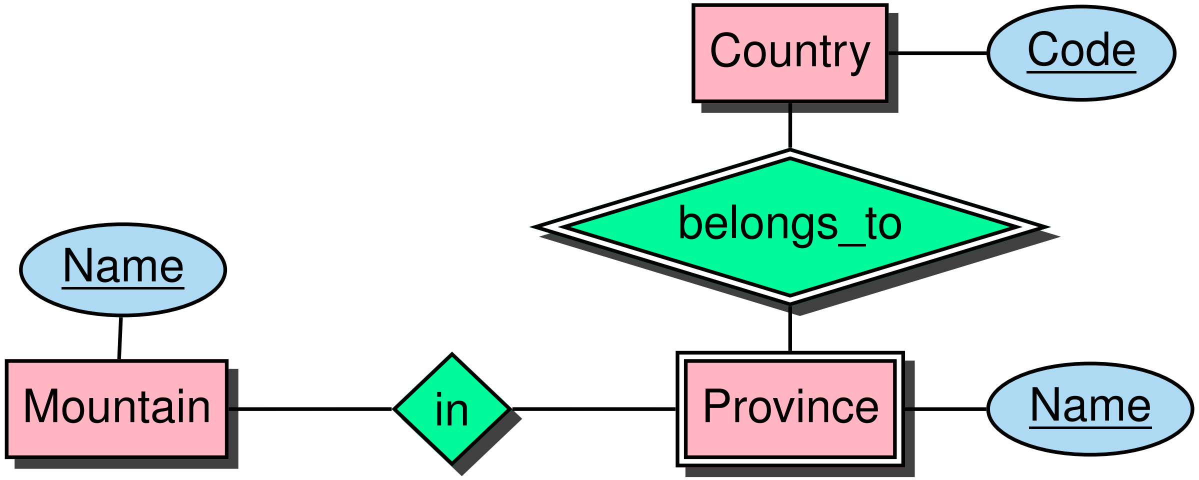
TABELLENDEFINITION: FOREIGN KEY ...REFERENCES

- `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 146 ff.

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

Tabellendefinition: Fremdschlüssel

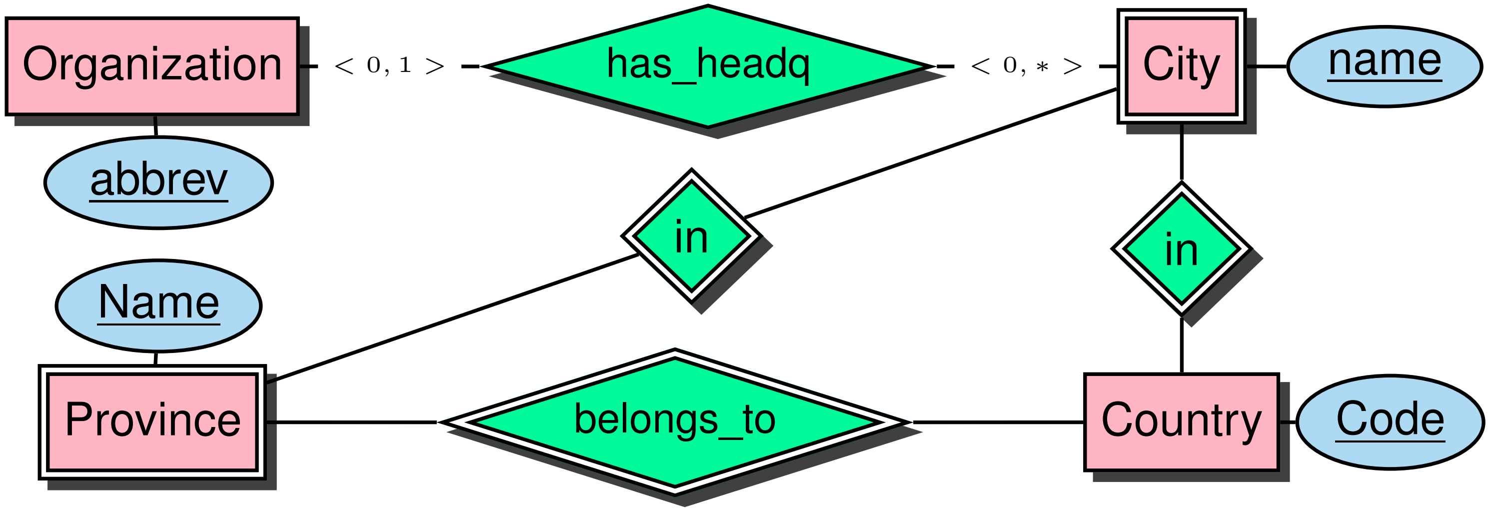
Ein Berg liegt in einer Provinz eines Landes:




```
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:



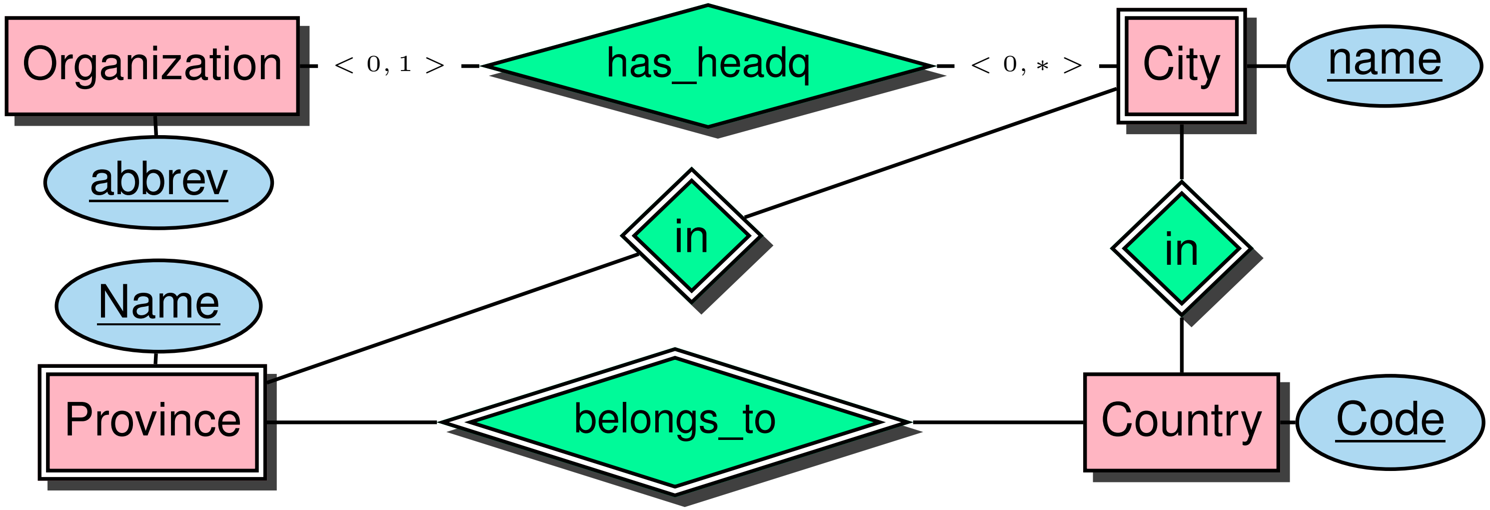
- Organization(Abbrev., Name, City, Country, Province, ...)

```
CREATE TABLE Organization ( ... ,
  CONSTRAINT orgrefshq
      FOREIGN KEY (City, Country, Province)
      REFERENCES City (Name, Country, Province) )
```

- Einzelne Nullwerte würden die FK-Bedingung nicht verletzen:
- `INSERT INTO Organization`
  `VALUES ('XX','xx','Clausthal','ZZ',NULL,NULL)`

Tabellendefinition

- Die meisten Organisationen haben ihren Sitz in einer Stadt:



- Organization(Abbrev., Name, City, Country, Province, ...)
- Zusammenhang des Fremdschlüssels erzwingen:

```
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;
```

```
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

Siehe Folie 138 ff.

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 droppen und nicht im Recyclebin sichern:

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

- Tabelle wieder holen:

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

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);
```

Einfügen von Daten: komplette Tabelle

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;
```

- Im Gegensatz zu einem View wird diese Tabelle beim Aufruf gefüllt, und ändert sich bei Änderungen an der Tabelle City nicht automatisch mit.

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,
    Latitude = NULL,
    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>    - Tabellenname, Subquery, oder DUAL
  ON (<condition>)           - über <source_relation> und <target_table>
  WHEN MATCHED THEN UPDATE
       SET <col_1> = <expr_1>, ..., <col_n> = <expr_n>
  WHEN NOT MATCHED THEN
       INSERT (<col'_1>,...,<col'_m>)
       VALUES (<expr'_1>,...,<expr'_m>);
```

- `<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 mit neuen Werten für Einwohnerzahlen (evtl. auch neue Länder)

```
CREATE TABLE NewCountryPops (name VARCHAR2(50), code VARCHAR2(4), pop 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.pop
  WHEN NOT MATCHED THEN
       INSERT (name, code, population)
       VALUES (n.name, n.code, n.pop);
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 156).

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 bei Abbruch eines Benutzerprozesses.

Kapitel 6
Spezialisierte Datentypen

- (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 254)
  - 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 409) [seit Oracle 9i-2/2002]
  - Ergänzungen “Extensions” (Spatial Data (seit Oracle 8i/8.1.5) etc.)

6.1 Datums- und Zeitangaben

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

- Oracle: 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');
```

Beispiel: 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	08 01 1297
E	20 01 1479
NL	26 07 1581
THA	01 01 1238

Verwendung von Zeitangaben

- SYSDATE liefert das aktuelle Datum (in Oracle-SQL).

```
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';
SELECT to_char(to_date('24.12.2002')) FROM dual;      -- wird erkannt
SELECT to_char(to_date('24 JUN 2002')) FROM dual;     -- wird erkannt
SELECT to_char(to_date('JUN 24 2002')) FROM dual;     -- wird nicht erkannt
-- 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
```

Constraints und Sonstiges mit Zeitangaben

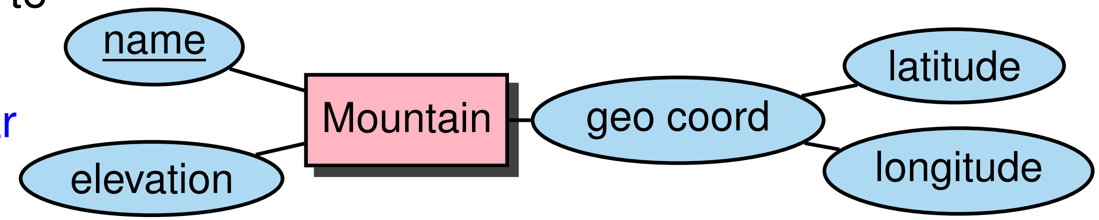
- Wenn man ein Datum in einem CREATE TABLE constraint verwenden muss, muss es zu dem zu diesem Zeitpunkt aktiven Format passen.
- produktabhängig: SYSDATE (in Oracle), CURRENT_DATE (in Postgres) ...
- In Oracle darf SYSDATE in constraints nicht verwendet werden, "da es veränderlich ist und somit das Constraint zur Laufzeit ohne ein Update verletzt werden könnte".
- in Postgres ist es erlaubt:

```
create table test(a date check(a <= current_date),
                  b date check(b >= current_date));
insert into text values(current_date,current_date);
```

Das Constraint wird nur zum Eingabe/Updatezeitpunkt geprüft.
Das Tupel ist am nächsten Tag auch noch da und selektierbar.

6.2 Zusammengesetzte Datentypen

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



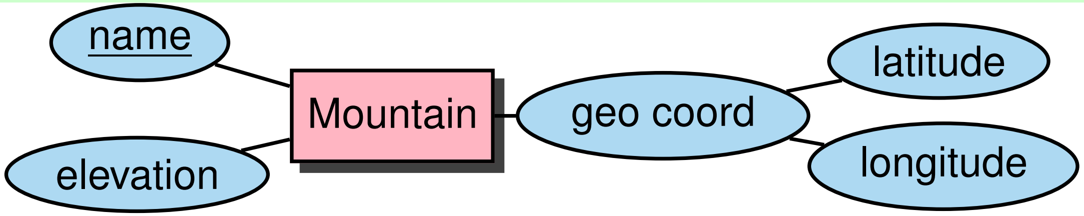
Neue Klasse von Schemaobjekten: `CREATE TYPE`

- ```
  CREATE [OR REPLACE] TYPE <name> AS OBJECT
    ( <attr_1> <datatype_1>,
        :
      <attr_n> <datatype_n>);
  /
  ```
  ← *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*).

- definiert automatisch eine *Konstruktormethode* `<type>( arg_1, ..., arg_n)`.
  ```

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);

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 können *nicht* gleich bei dem Datentypen definiert werden, sondern erst bei *(jeder!)* Tabelle, in der man ihn benutzt:

```
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 elegante 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	$\emptyset$
:	:	:	:

- 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 komplexeren Konzeptes “Nested Tables” (siehe Folie 128 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', . . .)

Tabellen mit Collections anfragen

- Collections und sie sind eigentlich kleine, sehr einfache Tabellen.
- 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
:	:

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.

Tabellen mit Collections vergleichen

- Instanzen von Collections 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;
```

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 nur 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_Spoken		
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>

$\Rightarrow$ Generische Syntax

Generische Syntax:

```
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 Spoken_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Spoken_list
  AS TABLE OF Spoken_T;
/
CREATE TABLE Nested_Spoken
  ( Country VARCHAR2(4),
    Languages Spoken_list)
  NESTED TABLE Languages
    STORE AS Lang_nested;
```

Einfügen mit Konstruktormethoden

```
INSERT INTO Nested_Spoken
VALUES ('CH', Spoken_list (Spoken_T('German',65), Spoken_T('French',18),
                           Spoken_T('Italian',12), Spoken_T('Romansch',1)));
```

[Filename (alles): nestedspoken.sql]

Geschachtelte Tabellen

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

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

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

Languages(Name, Percentage)
Spoken_List(Spoken_T('French', 18), Spoken_T('German', 65), Spoken_T('Italian', 12), Spoken_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 Nested_Spoken
            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 Nested_Spoken
 (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 Nested_Spoken -- syntaktisch zulässig, macht aber etwas falsches !!!!
 (SELECT Country,
        CAST(MULTISET(SELECT Name, Percentage
                      FROM Language
                      WHERE Country = A.Country)
             AS Spoken_List)
 FROM Language A); jedes Tupel (Land, dessen Sprachenliste) n-mal
  (n = Anzahl Sprachen in diesem Land), weil jedes Land n Tupel in 'Language' hat)
```

Füllen von Geschachtelten Tabellen

... also erst Tupel erzeugen und dann die geschachtelten Tabellen mit einer korrelierten SET-Subquery hinzufügen:

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

UPDATE Nested_Spoken B
SET Languages =
    CAST(MULTISET(SELECT Name, Percentage
                  FROM Language A
                  WHERE B.Country = A.Country)
         AS Spoken_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 Nested_Spoken
WHERE 'German' IN (SELECT name
                   FROM TABLE (Languages));
```

- TABLE(‹attr›) als korreliertes Join:

```
SELECT Country, nl1.*
FROM Nested_Spoken nl, TABLE(nl.Languages) nl1;
```

Komplexe Datentypen

```
SELECT * FROM USER_TYPES
```

Type_name	Type_oid	Typecode	Attrs	Meths
GeoCoord	_	Object	2	0
Spoken_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 Spoken_T`

"Typ mit abhängigen Typen oder Tabellen kann nicht gelöscht oder ersetzt werden"

`DROP TYPE Spoken_T FORCE` löscht `Spoken_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
- Transaktionen und Integritätsbedingungen
- View Updates
- Zugriffsrechte
- Anpassung der Datenbank an Sprache, Zeichensatz etc.
- 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,
- Spalten löschen,
- Bedingungen löschen, zeitweise außer Kraft setzen und wieder aktivieren.

```
ALTER TABLE <table>
  ADD <add-clause>
  MODIFY <modify-clause>
  DROP <drop-clause>
  DISABLE <disable-clause>
  ENABLE <enable-clause>
  RENAME TO <new-table-name>
```

- jede der obigen Zeilen kann beliebig oft vorkommen (keine Kommas dazwischen!),
- eine solche Zeile enthält eine oder mehrere Änderungs-Spezifikationen, z.B.

```
  MODIFY <modify-item>
  MODIFY (<modify-item>, ..., <modify-item>)
```

- Syntaxvielfalt nützlich wenn die Statements automatisch generiert werden.

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>...);
```

Neue Spalten werden mit `NULL`-Werten aufgefüllt.

Beispiel: Erweitern der Relation *economy* um eine Spalte *unemployment* mit Spaltenbedingung:

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

Entfernen von Spalten

```
ALTER TABLE <table>
  DROP (<column-name-list>);
ALTER TABLE <table>
  DROP COLUMN <column-name>;
```

HINZUFÜGEN VON TABELLENBEDINGUNGEN

```
ALTER TABLE <table>
  ADD (<... add some columns ... >,
       <tableConstraint>,
       ⋮
       <tableConstraint>);
```

Hinzufügen einer unemployment-Spalte und der 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 <= 102));
```

- 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

```
ALTER TABLE <table>
  MODIFY (<col> [<datatype>] [DEFAULT <value>]
           [<colConstraint> ... <colConstraint>],
          ⋮
          <col> [<datatype>] [DEFAULT <value>]
           [<colConstraint> ... <colConstraint>]);
```

```
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));
ALTER TABLE isMember
  MODIFY (type VARCHAR2(10))   - change maximal length;
```

- Hinzufügen von Spaltenbedingungen – Fehlermeldung, falls eine Bedingung formuliert wird, die der aktuelle Datenbankzustand nicht erfüllt.
- Datentypänderungen (z.B. NUMBER zu VARCHAR2(n)) nur erlaubt wenn die Spalte leer ist,
- Änderung der Länge von VARCHAR-Spalten ist jederzeit möglich: VARCHAR2(n) $\rightarrow$ VARCHAR2(k).

Integritätsbedingungen (de)aktivieren

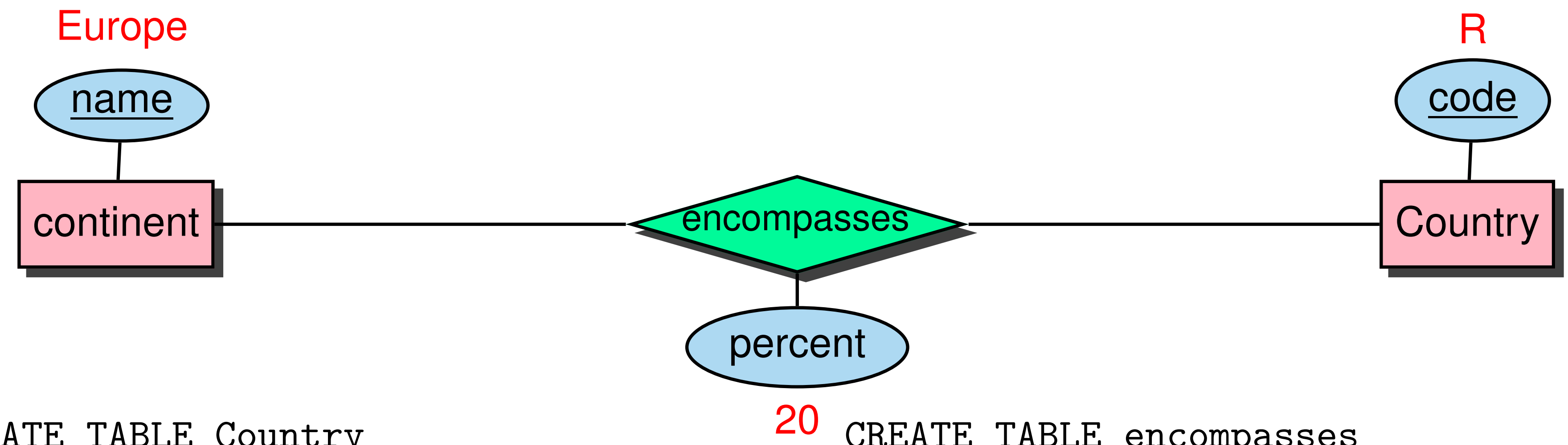
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

```
ALTER TABLE <table>
  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 `REFERENCES`-Deklarationen darauf 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(50),
  code      VARCHAR2(4) PRIMARY KEY,
  ...);

CREATE TABLE Continent
 (name      VARCHAR2(20) PRIMARY KEY,
  area      NUMBER(2));
```

```
CREATE TABLE encompasses
  ( Continent VARCHAR2(20)
       REFERENCES Continent(name),
    Country   VARCHAR2(4)
       REFERENCES Country(code),
    percentageNUMBER);
```

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 isMember WHERE country='D';
   ```

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, ...)	$\rightsquigarrow$	(United Kingdom, UK, ...)
Province:	(Yorkshire, GB, ...)	$\rightsquigarrow$	(Yorkshire, UK, ...)
City:	(London, GB, Greater London, ...)	$\rightsquigarrow$	(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,Bremen,D,Weser,NULL,North Sea)

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

located(Bremerhaven,Bremen,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 9-11: 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]
```

- Hinweis: MS SQL Server unterstützt `ON UPDATE CASCADE`, jedoch kein `ON DELETE/UPDATE CASCADE` auf rekursiven Schemata (z.B. River/flows_into) (Stand 2013)!

Referentielle Aktionen: UPDATE ohne CASCADE

Beispiel: Umbenennung eines Landes:

```
CREATE TABLE Country
  ( Name   VARCHAR2(50) NOT NULL UNIQUE,
    Code   VARCHAR2(4) PRIMARY KEY);
```

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

```
CREATE TABLE Province
  ( Name     VARCHAR2(50)
    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').

Behandlung siehe Folie 158.

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: Semantische und Operationale Probleme

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 Transaktionen und Integritätsbedingungen

- Theorie: Integritätsbedingungen werden (erst) zum Ende der Transaktion überprüft (und nicht nach jedem einzelnen Statement)
- Praxis: Es ist effizienter (und manchmal benutzerfreundlicher), sie nach jedem einzelnen Statement zu überprüfen.

$\Rightarrow$ konfigurierbar:

`IMMEDIATE` vs. `DEFERRED`

Syntax

- Bei Definition (CREATE TABLE und ALTER TABLE) von Constraints kann DEFERRABLE bzw. NOT DEFERRABLE (Default) und im ersten Fall IMMEDIATE DEFERRED (sonst automatisch immediate) angegeben werden.

```
ALTER TABLE isMember
ADD CONSTRAINT MemberCRef
               FOREIGN KEY (Country) REFERENCES Country(Code)
               ON DELETE CASCADE
               DEFERRABLE IMMEDIATE DEFFERED
ADD CONSTRAINT MemberOrgRef
               FOREIGN KEY (Organization) REFERENCES Organization(Abbreviation)
               DEFERRABLE IMMEDIATE DEFFERED;
```

- (Oracle): Das Setting kann vom Benutzer (lokal) geändert werden (falls durch DEFERRABLE erlaubt):
  - Für eine einzelne Transaktion:

    ```
    SET CONSTRAINT[S] {<name>|ALL} {IMMEDIATE|DEFERRED}
    ```

  - Für die ganze Session:

    ```
    ALTER SESSION
    SET CONSTRAINT[S] {<name>|ALL} {IMMEDIATE|DEFERRED}
    ```

Beispiel: UPDATE ohne CASCADE

Situation von Folie 152:
Änderung des Landescodes “GB” auf “UK” innerhalb der gesamten Datenbank:

```
SET constraints all deferred;
UPDATE Country SET Code='UK' WHERE Code='GB';
UPDATE Province SET Country='UK' WHERE Country='GB';
  -- weitere Updates --
COMMIT;
```

- wenn jetzt eines vergessen wurde, wird allerdings ein komplettes Rollback ausgeführt, und man darf von vorne anfangen!

⇒ mit Skript arbeiten.

Vor- und Nachteile von DEFERRED/IMMEDIATE

- `DEFERRED` manchmal notwendig bei Updates,
- Bei großen Mengen von Updates ist `DEFERRED` effizienter (1x alle prüfen statt mehrmals dasselbe tun – mengenorientiert statt tupelorientiert),
- Eigentlich Unique-Indexe auf Keys können bei `DEFERRED` zeitweise verletzt sein
  $\rightarrow$ DB arbeitet ggf. mit Multiindexen
  $\rightarrow$ ineffizienter beim Lesen
  (deshalb ist `IMMEDIATE` Default)
- Bei `DEFERRED` bekommt man Fehlermeldungen erst am Transaktionsende.
- Hinweis: MS SQL Server unterstützt keine `DEFERRED`-Constraints (Stand 2013).

Einzelanalyse DEFERRED/IMMEDIATE

- `CHECK`-Constraints: `IMMEDIATE` meistens sinnvoller,
- `PRIMARY KEY, UNIQUE`: `IMMEDIATE` meistens sinnvoller,
- `NOT NULL`: `DEFERRED` praktisch um ein eingefügtes Tupel erst später zu vervollständigen,
- Referentielle Integritätsbedingungen:
  - wenn man die Einfügereihenfolge nicht weiß, ist `DEFERRED` sinnvoll,
  - bei zyklischen Strukturen (Country/Capital/City) sollte mindestens ein Constraint `DEFERRED` sein.

DEFERRED/IMMEDIATE vs. DISABLE/ENABLE

- `DISABLE/ENABLE` erfordert `ALTER TABLE`-Rechte,
  `DEFERRABLE` wird einmal vom Admin gesetzt, und dann kann jeder Benutzer in seiner Session entscheiden, ob er `DEFERRED` setzen möchte
- Mehrbenutzerbetrieb: Ausschalten der Constraints gilt für alle. Beim Wiedereinschalten hat man die Fehler der anderen mit am Hals.
- bei sehr komplexen Updates will man mit Transaktionen arbeiten und Bedingungsverletzungen zeitweise erlauben
  → alleine, Constraints disablen, schrittweise committen.

7.4 View Updates

Views

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

View Updates

- müssen intern operational 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

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.
  (bei $\subsetneq$ sind eventuell mehrere Tupel der Basistabelle betroffen).
- Schlüssel einer Basistabelle überdeckt keinen Schlüssel des Views: i.a. keine Umsetzung möglich (siehe Aufgaben).
- die Heuristik ist nicht immer so ganz korrekt (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*.

7.5 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

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

- `PUBLIC`: jeder erhält das Recht.
- `WITH 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;` darf überall Tabellen löschen
  `GRANT SELECT ANY TABLE TO supervisor;` darf alle Tabellen aller Benutzer lesen
- `REVOKE CREATE TABLE FROM mueller;` darf keine Tabellen mehr erzeugen.

Informationen über eigene Systemprivilegien: `SELECT * FROM SESSION_PRIVS;`

Stichwort: Rollenkonzept – fasst eine Menge von Rechten zusammen

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>`: eine Instanz von `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 Objekt (ggf. auf der angegebenen Spalte) hat.
- `GRANT OPTION`: Der Empfänger darf das Recht weitergeben.

Objektprivilegien (Forts.)

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;
```

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);
```

DURCHGRIFF AUF ANDERE BENUTZER/REINE SCHEMA-ACCOUNTS

- Man (Admin!) kann einem Benutzer erlauben, “sich” mit dem Rechten eines anderen Accounts (und dessen Schema) einzuloggen.
- Dabei kann man auch User anlegen, die nur als “Schema-Accounts” existieren, und sich selber garnicht einloggen können:

  ```
  ALTER USER <data-owner> GRANT CONNECT THROUGH <user>
    (zu lesen als ‘connect by’, nicht als ‘connect as’)
  ALTER USER dbis GRANT CONNECT THROUGH may;
  ```

- und dann:

  ```
  may@login> sqlplus
  SQL> connect may[dbis]/may-passwort
  SQL> select * from continent
     greift automatisch auf dbis.continent zu,
     hat alle Rechte, die dbis hat.
  ```

- Die “realen-Personen”-Benutzer benötigen dann nicht einmal ein eigenes Schema, und können auch keine eigenen (privaten) Tabellen in der “Firmen”-Datenbank anlegen.

7.6 Anpassung der Datenbank an Sprache, Zeichensatz etc.

- Alle Benutzer arbeiten (“session”) auf demselben Datenbestand (“system”, “database”, “instance”),
- Lokale Anpassungen: Sprache für Fehlermeldungen, Darstellung von Datum, Dezimalkomma/punkt, Zeichensatz, ...
- Oracle NLS: Natural Language Support
  - NLS_DATABASE_PARAMETERS: bei Erzeugung der Datenbank gesetzt
  - NLS_SESSION_PARAMETERS: bei Beginn der Session gesetzt

Anpassungs-Parameter

```
SELECT * FROM NLS_{SESSION|DATABASE}_PARAMETERS;
```

Parameter	Value	
NLS_LANGUAGE	{AMERICAN|...}	
NLS_NUMERIC_CHARACTERS	{.,|,.}	Dezimalpunkt/komma, z.B. 50.000,00
NLS_CALENDAR	{GREGORIAN|...}	
NLS_DATE_FORMAT	{DD-MON-YYYY|...}	
NLS_DATE_LANGUAGE	{AMERICAN|...}	
NLS_CHARACTERSET	{AL32UTF8|...}	
NLS_SORT	{BINARY|GERMAN}	Behandlung von Umlauten
NLS_LENGTH_SEMANTICS	{BYTE|CHAR}	
NLS_RDBMS_VERSION	{11.1.2.0.1.0|19.0.0.0|. . .}	

```
ALTER {SESSION|SYSTEM} SET <parameter> = <value>;
```

- NLS_LENGTH_SEMANTICS: Umlaute etc. haben mehrere Bytes ('Göttingen' hat unter UTF8 10 Bytes/Zeichen)

7.7 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:

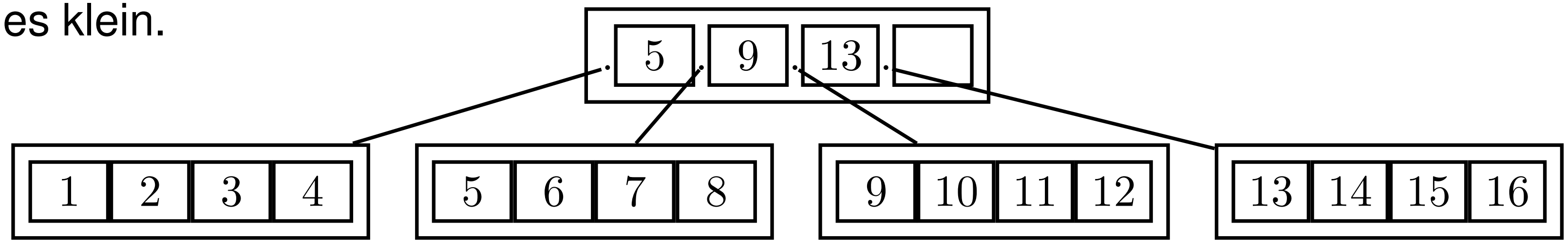
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

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* algebrabasierte 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.

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

- Ausgabe alphabetisch nach Code geordnet:
  Auf Schlüsselattributen sind automatisch Indexe angelegt und werden 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);
```

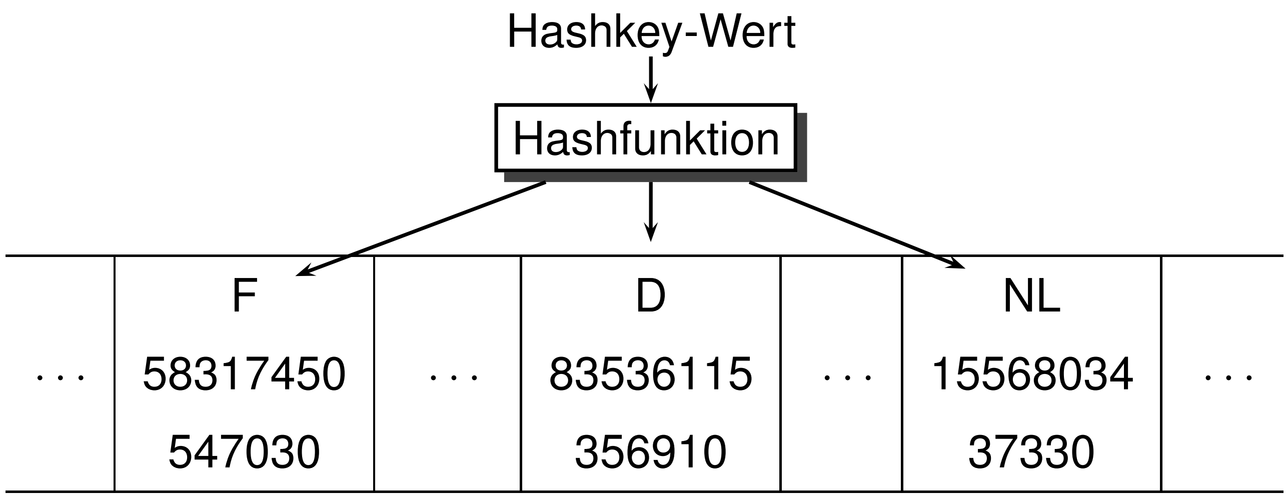
- Ausgabe obiger Anfrage jetzt nach Population geordnet.
  (Blätter des Baums linear durchgehen)

```
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



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
	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	...
⋮	⋮	⋮	⋮	⋮