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 oder sonstwas zwischen diesen Statements!),
- 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: Die Relation *economy* wird um eine Spalte *unemployment* mit Spaltenbedingung erweitert:

```
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 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)) sind nur erlaubt wenn die Spalte leer ist,
- Änderung der Länge von VARCHAR-Spalten ist jederzeit möglich: VARCHAR2(n) $\rightarrow$ VARCHAR2(k).

INTEGRITÄTSBEDINGUNGEN (DE)AKTIVIEREN

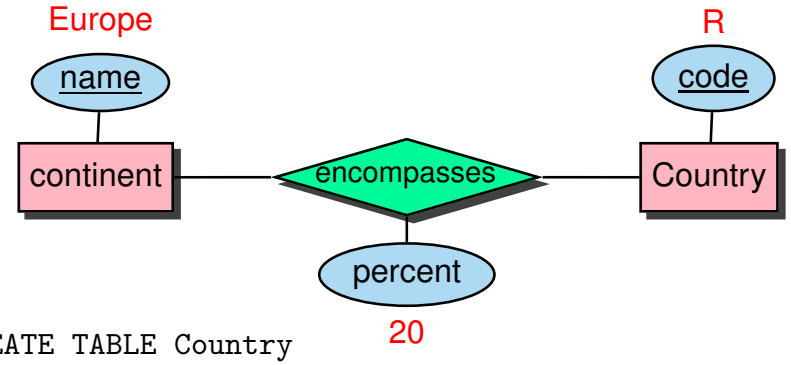
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

```
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 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(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),
  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 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 NULLs 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)

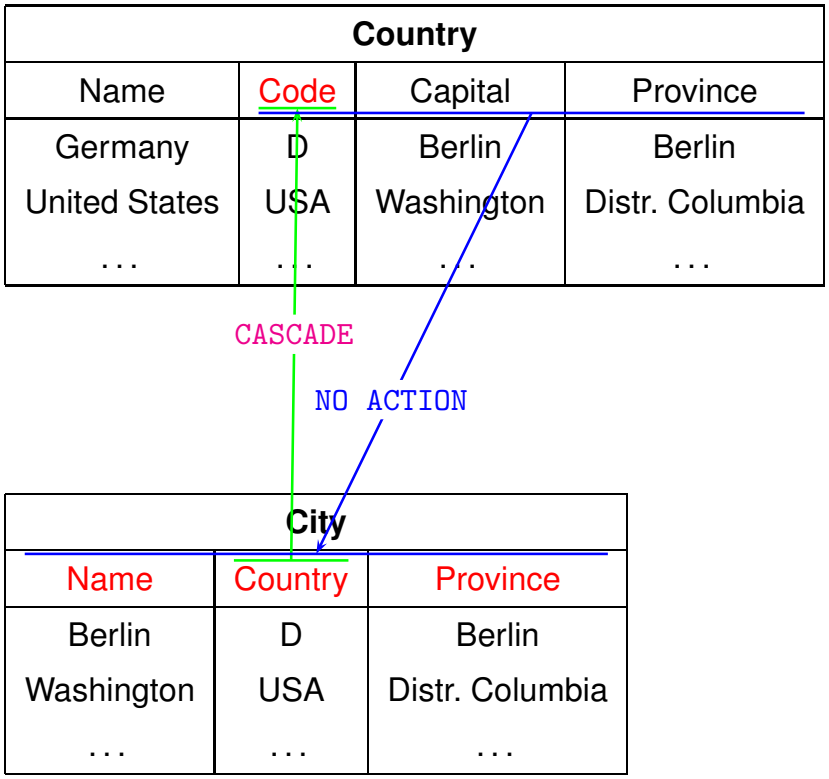
```
<col> <datatype>
  CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ ON DELETE {NO ACTION | RESTRICT | CASCADE |
               SET DEFAULT | SET NULL } ]
  [ ON UPDATE {NO ACTION | RESTRICT | CASCADE |
               SET DEFAULT | SET NULL } ]
```

oder <tableConstraint> (für mehrere Spalten)

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

angegeben.

REFERENTIELLE AKTIONEN




```
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:

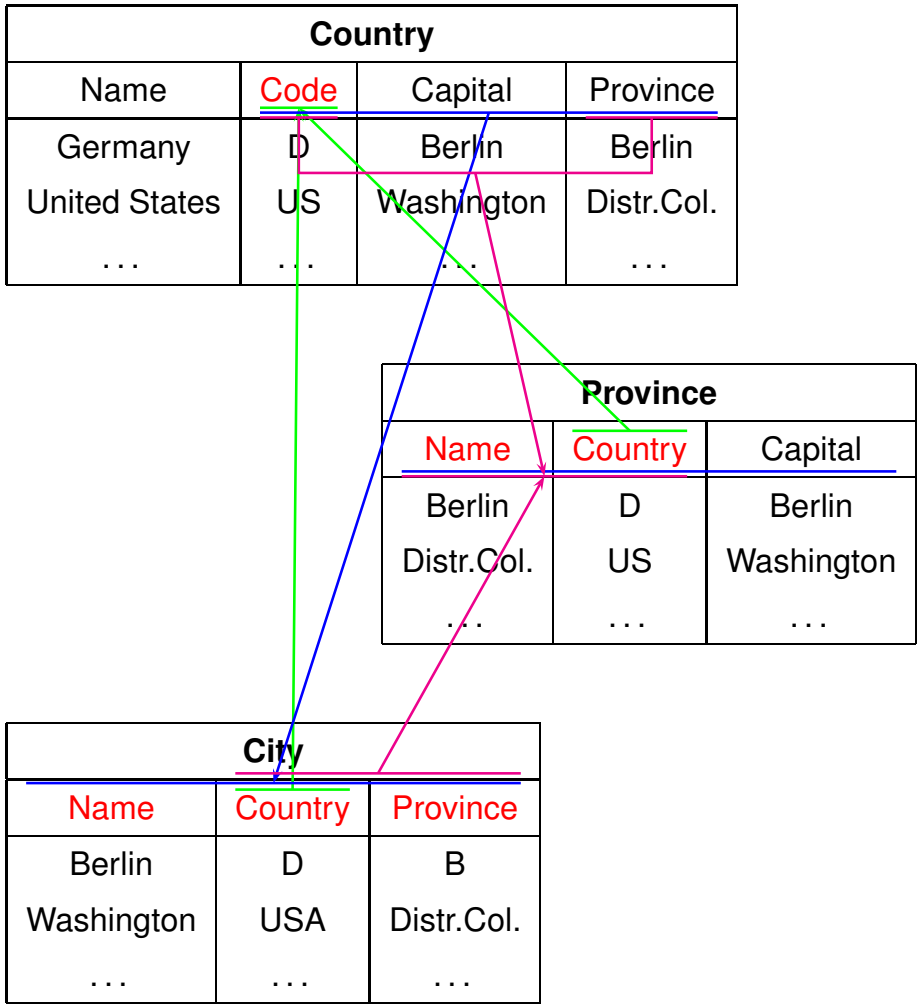
```
CREATE TABLE Country
  ( Name  VARCHAR2(50) NOT NULL UNIQUE,
    Code  VARCHAR2(4) PRIMARY KEY);

('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.

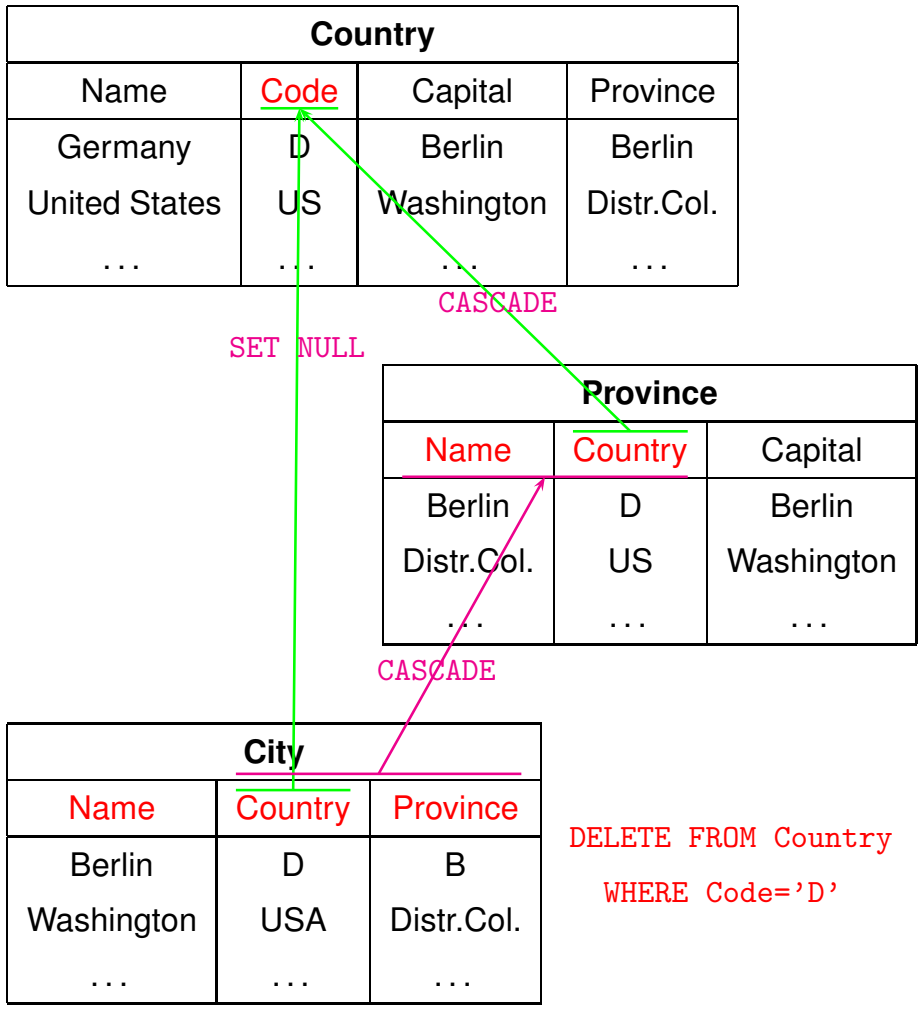
- `UPDATE Country SET Code='UK' WHERE Code='GB';`
  ⇝ "dangling reference" des alten Tupels ('Yorkshire','GB').
- `UPDATE Province SET Code='UK' WHERE Code='GB';`
  ⇝ "dangling reference" des neuen Tupels ('Yorkshire','UK').

Behandlung siehe Folie 154.

REFERENTIELLE AKTIONEN: ZYKLISCHE REFERENZEN



REFERENTIELLE AKTIONEN: PROBLEMATIK



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 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.

⇒ 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 148:
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!

$\Rightarrow$ 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 $\rightarrow$ 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 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.
  (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

$\Rightarrow$ *Materialized Views*

- werden bei der Definition berechnet und
- bei jeder Datenänderung automatisch aktualisiert (u.a. durch *Trigger*).
- $\Rightarrow$ 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

```
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);
```

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	{.,|,.}
NLS_CALENDAR	{GREGORIAN|...}
NLS_DATE_FORMAT	{DD-MON-YYYY|...}
NLS_DATE_LANGUAGE	{AMERICAN|...}
NLS_CHARACTERSET	{AL32UTF8|...}
NLS_SORT	{BINARY|GERMAN}
NLS_LENGTH_SEMANTICS	{BYTE|CHAR}
NLS_RDBMS_VERSION	{11.2.0.1.0|...}

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

- NLS_NUMERIC_CHARACTERS: Dezimalpunkt/komma, z.B. 50.000,00
- NLS_SORT: Behandlung von Umlauten
- NLS_LENGTH_SEMANTICS: Umlaute etc. haben mehrere Bytes
  ('Göttingen' hat unter UTF8 10 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:

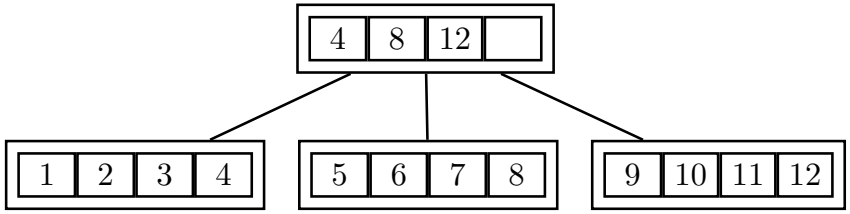
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

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* 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 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);
```

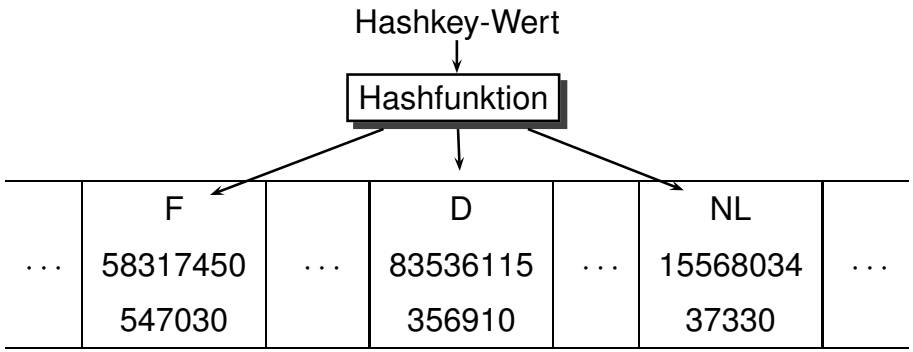
- Ausgabe obiger Anfrage jetzt nach Population geordnet.
  (Blätter des Baums linear durchgehen)

```
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
	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(50));
```

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(50) PRIMARY KEY,
   Depth  NUMBER)
  CLUSTER Cl_Sea (Name);
```

```
CREATE TABLE Cgeo_Sea
  (Province  VARCHAR2(50),
   Country   VARCHAR2(4),
   Sea       VARCHAR2(50))
  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;
```