TEIL II: Dies und Das

Teil I: Grundlagen

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

Teil II: Weiteres zum “normalen” SQL

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

Teil III: Prozedurale Konzepte, OO, Einbettung

- PL/SQL: Prozeduren, Funktionen, Trigger
- Objektorientierung
- Embedded SQL, JDBC

Ändern von Schemaobjekten

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

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

Ändern von Tabellen

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

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


Hinzufügen von Spalten zu einer Tabelle

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

Neue Spalten werden mit NULL-Werten aufgefüllt.

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

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

Hinzufügen von Tabellenbedingungen

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

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

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

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

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


Spaltendefinitionen einer Tabelle ändern

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

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

- als `<colConstraint>` sind nur `NULL` und `NOT NULL` erlaubt. Alle anderen Bedingungen müssen mit `ALTER TABLE ... ADD (<tableConstraint>)` hinzugefügt werden.

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

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

ALTER TABLE ... DROP/DISABLE/ENABLE

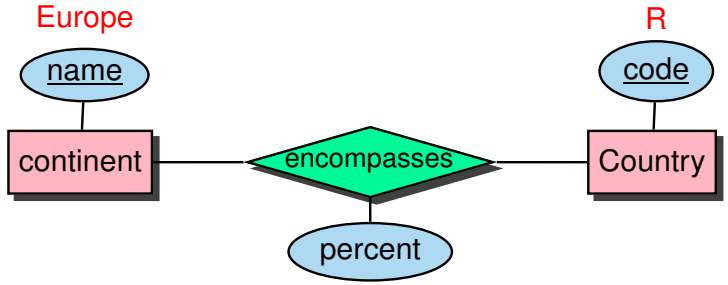
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

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

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

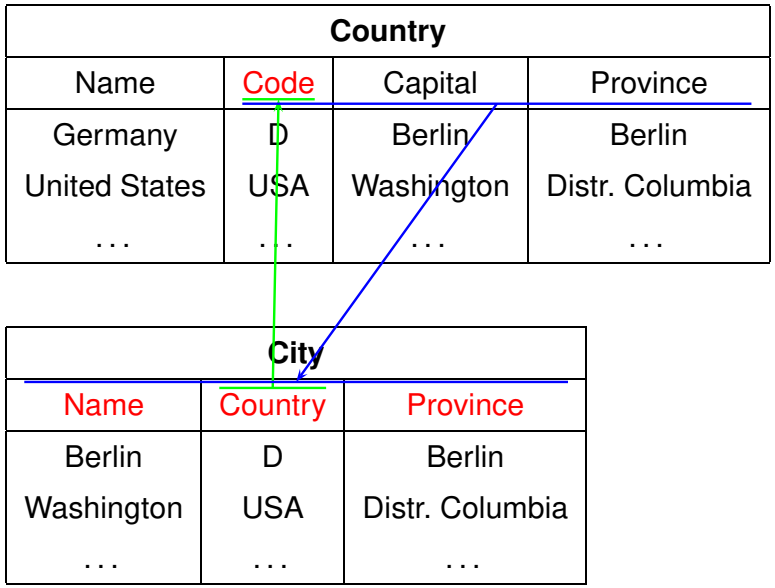
Referentielle Integrität

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




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

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

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

Referentielle Integrität



Country

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

City

Name	Country	Province
Berlin	D	Berlin
Washington	USA	Distr. Columbia
...	...	...

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

- (<attr-list'>) muß Candidate Key der referenzierten Tabelle sein.
- in ORACLE: Primary Key gefordert.

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 Aktionen ausführen, um die referentielle Integrität der Datenbasis zu erhalten
- Ist dies nicht möglich, so werden die gewünschten Operationen nicht ausgeführt, bzw. zurückgesetzt.

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

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

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

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

   Anderenfalls ist es unkritisch:

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

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

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


Referentielle Aktionen im SQL-2-Standard

`NO ACTION:`

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

```
DELETE FROM River;
```

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

`RESTRICT:`

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

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

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

`CASCADE:`

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

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

ändert überall:

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

Referentielle Aktionen im SQL-2-Standard

`SET DEFAULT:`
Die Operation wird ausgeführt und bei den referenzierenden Tupeln wird der entsprechende Fremdschlüsselwert auf die für die entsprechende Spalten festgelegten `DEFAULT`-Werte gesetzt (dafür muß dann wiederum ein entsprechendes Tupel in der referenzierten Relation existieren).

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

`located`: Stadt liegt an Fluss/See/Meer
located(Bremerhaven,Nds.,D,Weser,`NULL`,North Sea)

```
DELETE * FROM River WHERE Name='Weser';
```
located(Bremerhaven,Nds.,D,`NULL`,`NULL`,North Sea)

Referentielle Aktionen im SQL-2-Standard

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

`<columnConstraint>` (für einzelne Spalten)

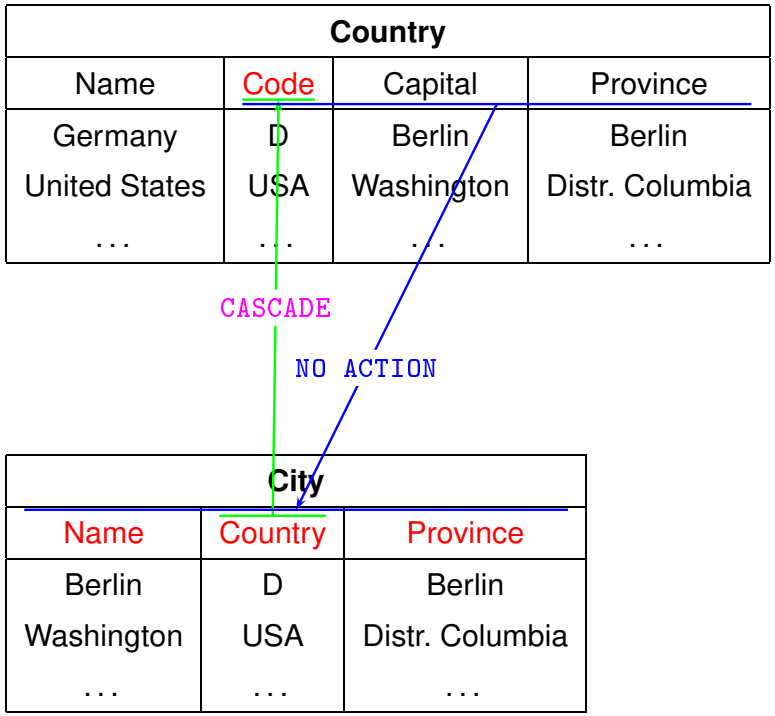
```
<col> <datatype>
  CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ ON DELETE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
  [ ON UPDATE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
```

oder `<tableConstraint>` (für mehrere Spalten)

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

angegeben.

Referentielle Aktionen




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


Referentielle Aktionen in ORACLE:

- ORACLE 8: nur `ON DELETE/UPDATE NO ACTION` und `ON DELETE CASCADE` 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]
```

Syntax als `<tableConstraint>`:

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

Referentielle Aktionen: UPDATE ohne CASCADE

Beispiel: Umbenennung eines Landes:

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

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

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

```
('Yorkshire','GB')
```

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

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

Referentielle Aktionen: UPDATE ohne CASCADE

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

```
ALTER TABLE Province
  DISABLE CONSTRAINT ProvRefsCountry;
```

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

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

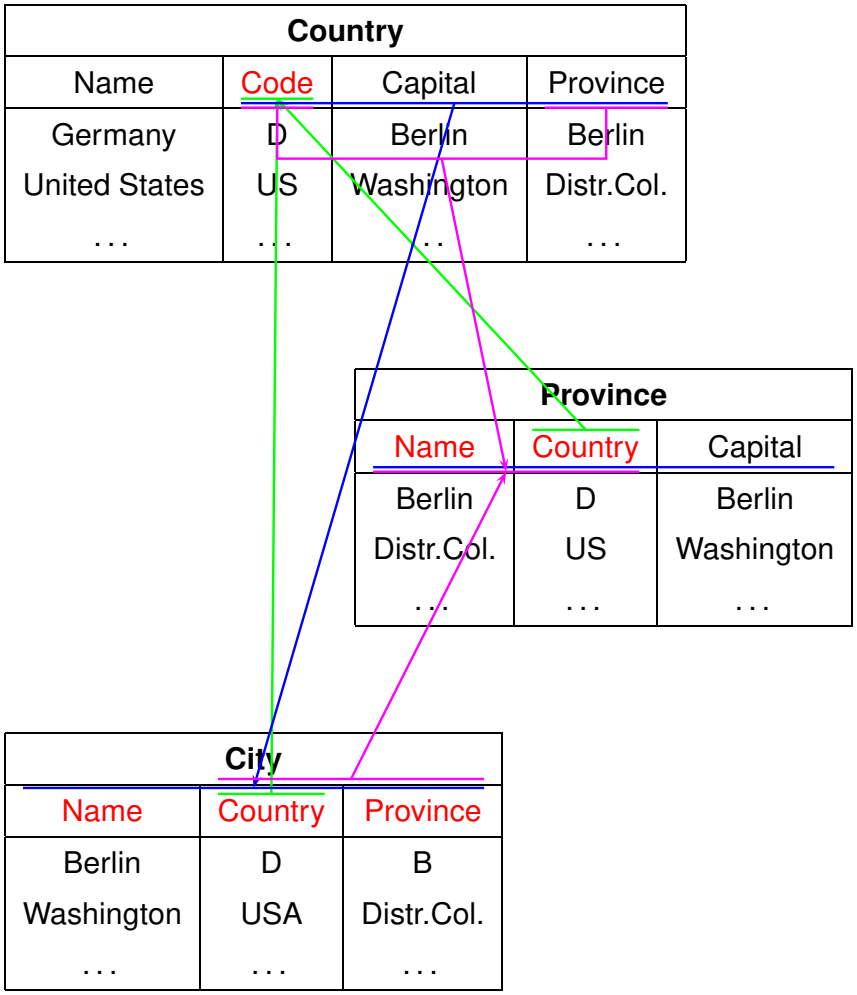
```
ALTER TABLE Province
  ENABLE CONSTRAINT ProvRefsCountry;
```

Referentielle Integritätsbedingungen

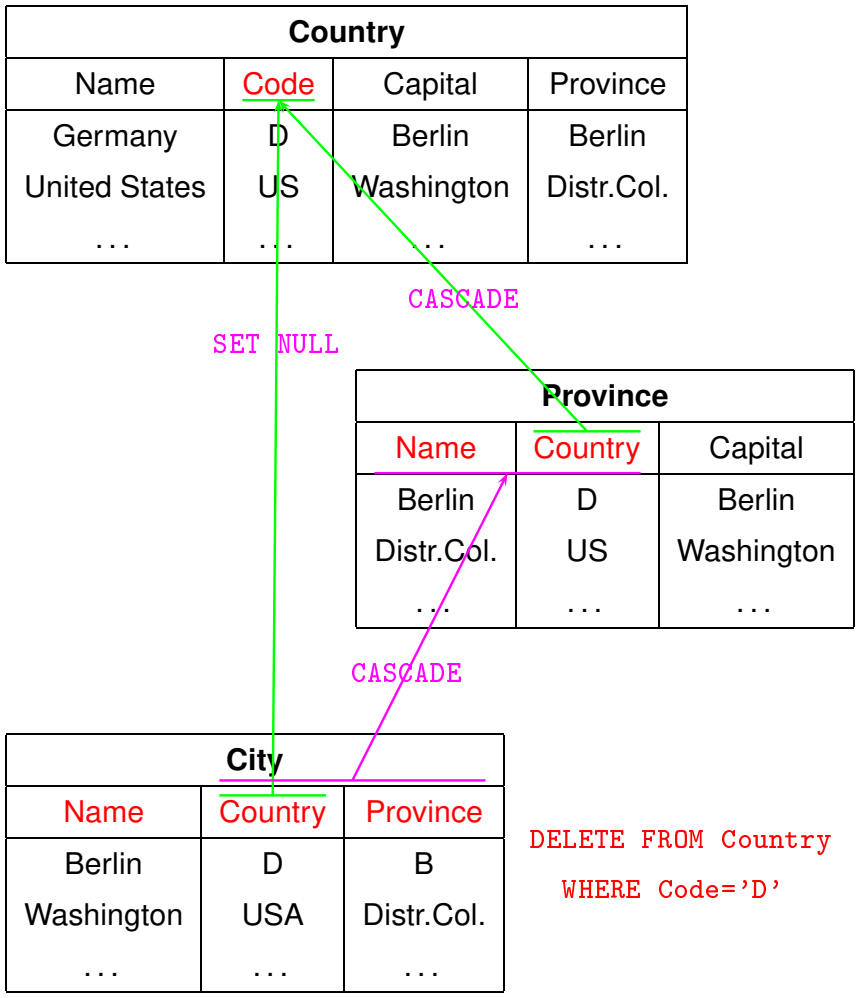
Man kann ein Constraint auch bei der Tabellendefinition mitdefinieren, und sofort disablen:

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


Referentielle Aktionen: Zyklische Referenzen



Referentielle Aktionen: Problematik `ON UPDATE`



Referentielle Aktionen

Im allgemeinen Fall:

- Schon ein einzelnes Update 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.

Arbeiten am Lehrstuhl zu diesem Thema:

- B. Ludäscher, W. May, and G. Lausen: Referential Actions as Logical Rules. In *Proc. 16th ACM Symposium on Principles of Database Systems*, Tucson, Arizona, 1997.
- B. Ludäscher, W. May: Referential Actions: From Logical Semantics to Implementation. In *Proc. 6th Intl. Conf. on Extending Database Technologies*, Valencia, Spain, 1998.
- W. May, B. Ludäscher: Understanding the Global Semantics of Referential Actions using Logical Rules. In *ACM Transactions on Database Systems*, 27(4), 2002.

Views

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

View Updates

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

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

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


View Updates

- abgeleitete Werte können nicht verändert werden:

Beispiel:

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

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

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

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

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

View Updates

Beispiel:

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

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

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

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

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

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

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

View Updates

Beispiel:

- *Country* darf nicht verändert werden:

City	Country
Berlin	Germany
Freiburg	Germany

Umsetzung auf Basistabelle wäre nicht eindeutig:

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

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

View Updates

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

View Updates

Beispiel:

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

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

Country	Name	Population
D	Nordrhein-Westfalen	17816079

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

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

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

Country	Name	Population
D	Bayern	11921944

View Updates

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

Beispiel

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

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

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

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

Materialized Views

- Views werden bei jeder Anfrage neu berechnet.

\+ repräsentieren immer den aktuellen Datenbankzustand.

\- zeitaufwendig, ineffizient bei wenig veränderlichen Daten

$\Rightarrow$ *Materialized Views*

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

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 CLUSTER.`
- 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 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 City
    FOR dbis.City;
  ```

```
SELECT * FROM City;
```

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 recht 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 sind nicht erlaubt.

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

Optimierung der Datenbank

- möglichst wenige Hintergrundspeicherzugriffe
- Daten soweit möglich im Hauptspeicher halten

Datenspeicherung:

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

Anfrageoptimierung:

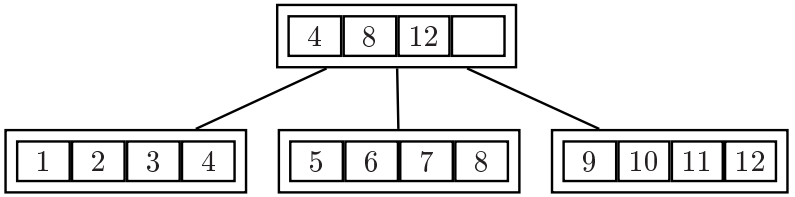
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

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,
- mehrere Indexe für eine Tabelle möglich.

Zugriffspfade: Indexe

Zugriff über indizierte Spalte(n) erheblich effizienter:

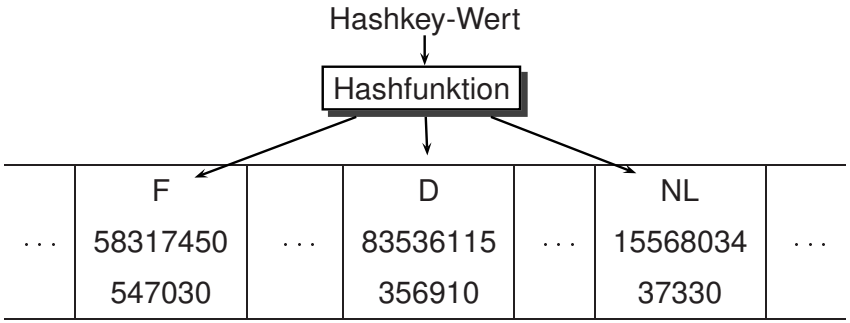
- benötigte Indexknoten aus Hintergrundspeicher holen,
- dann nur ein Zugriff um ein Tupel zu bekommen.

```
CREATE TABLE PLZ
  (City      VARCHAR2(35)
   Country   VARCHAR2(4)
   Province  VARCHAR2(32)
   PLZ    NUMBER)

CREATE INDEX PLZIndex ON PLZ (Country,PLZ);

SELECT *
  FROM plz
  WHERE plz = 79110 AND Country = 'D';
```


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(25));
```

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

Erzeugen eines Clusters in ORACLE

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

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

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

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

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

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

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