Teil II

Erweiterte Konzepte innerhalb SQL

8 REFERENTIELLE INTEGRITÄT

Referentielle Integritätsbedingungen entstehen aus dem Zusammenhang zwischen Primär- und Fremdschlüsseln. Eine referentielle Integritätsbedingung

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

definiert eine Inklusionsabhängigkeit: Zu jedem (Fremdschlüssel)wert der Attribute `(<attr-list>)` der *referenzierenden* Tabelle (*C- (Child-) Table*) `<table>` muss ein entsprechender Schlüsselwert von `(<attr-list>)` in der *referenzierten* Tabelle `<table>` (*P- (Parent-) Table*) existieren.

Dabei muss `(<attr-list'>)` ein Candidate Key der referenzierten Tabelle sein. In ORACLE kann nur der deklarierte `PRIMARY KEY` referenziert werden, damit ist die Angabe der Spalten optional.

Referentielle Integritätsbedingungen treten immer auf, wenn bei der Umsetzung vom ER-Modell zum relationalen Modell Schlüsselattribute der beteiligten Entities in Beziehungstypen eingehen:

```
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);
```

Aufgabe *referentieller Aktionen* bzgl. einer C-Tabelle ist, bei Veränderungen am Inhalt der P-Tabelle Aktionen auf der C-Tabelle auszuführen, um die referentielle Integrität der Datenbasis zu erhalten. Ist dies nicht möglich, so werden die gewünschten `DELETE/UPDATE`-Operationen nicht ausgeführt, bzw. zurückgesetzt.

8.1 Referentielle Aktionen im SQL-2 Standard

Nach dem *SQL-2-Standard* werden referentielle Integritätsbedingungen werden bei `CREATE TABLE` und `ALTER TABLE` als `<columnConstraint>` (für einzelne Spalten)

```
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) angegeben:

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

Die Klauseln `ON DELETE` und `ON UPDATE` geben an, welche referentiellen Aktionen bei einem `DELETE` bzw. `UPDATE` auf die referenzierte Tabelle ausgeführt werden sollen, um die referentielle Integrität der Datenbasis zu gewährleisten.

1. Ein `INSERT` bzgl. der referenzierten Tabelle oder ein `DELETE` bzgl. der referenzierenden Tabelle ist immer unkritisch.
   `INSERT INTO Country VALUES ('Lummerland,'LU',...);`
   `DELETE FROM isMember ('D','EU');`
2. Ein `INSERT` oder `UPDATE` bzgl. der referenzierenden Tabelle, das einen Fremdschlüsselwert erzeugt, zu dem kein Schlüssel in der referenzierten Tabelle existiert, ist immer unzulässig:
   `INSERT INTO City VALUES ('Karl-Marx-Stadt','DDR',...);`
   anderenfalls ist es unkritisch:
   `UPDATE City SET Country='A' WHERE Name='Munich';`
3. Notwendig sind damit nur referentielle Aktionen für `DELETE` und `UPDATE` bzgl. der referenzierten Tabelle:
   `UPDATE Country SET Code='UK' WHERE Code='GB';` oder
   `DELETE FROM Country WHERE Code='I';`

`NO ACTION:`
Die `DELETE/UPDATE`-Operation auf der P-Tabelle wird zunächst ausgeführt; *Nach der Operation* wird überprüft, ob "dangling references" in der C-Tabelle entstanden sind. Falls ja, war die Operation verboten und wird zurückgenommen.

```
CREATE TABLE River
(Name VARCHAR2(20) CONSTRAINT RiverKey PRIMARY KEY,
 River VARCHAR2(20) REFERENCES River(Name),
 Lake VARCHAR2(20) REFERENCES Lake(Name),
 Sea VARCHAR2(25) REFERENCES Sea(Name),
 Length NUMBER);

DELETE FROM River;
```

Unter der Annahme, dass es keine weiteren Referenzen auf *River* gäbe (was in MONDIAL nicht der Fall ist, da *located River* referenziert), würden dabei alle Flüsse gelöscht. Referenzen durch Nebenflüsse, die ebenfalls gelöscht werden, sind dabei kein Hindernis (da der Datenbankzustand *nach* der kompletten Operation betrachtet wird).

`RESTRICT:`
Die `DELETE/UPDATE`-Operation auf der P-Tabelle wird nur dann ausgeführt, wenn keine "dangling references" in der C-Tabelle entstehen können:

```
DELETE FROM Organization;
```

wird abgebrochen, falls es eine Organisation gibt, die Mitglieder hat (Referenz von *isMember* auf *Organization*).

`CASCADE:`

Die `DELETE/UPDATE`-Operation auf der P-Tabelle wird ausgeführt. Die referenzierenden Tupel der C-Tabelle werden ebenfalls mittels `DELETE` entfernt, bzw. mittels `UPDATE` geändert. Ist die C-Tabelle selbst P-Tabelle bzgl. einer anderen Bedingung, so wird das `DELETE/UPDATE` bzgl. der dort festgelegten Lösch/Änderungs-Regel weiter behandelt:

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

wird am sinnvollsten dadurch ausgeführt, dass die Ersetzung für referenzierende Tupel ebenfalls durchgeführt wird:

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

`SET DEFAULT:`

Die `DELETE/UPDATE`-Operation auf der P-Tabelle wird ausgeführt und bei den referenzierenden Tupeln der C-Tabelle wird der entsprechende Fremdschlüsselwert auf die für die betroffenen Spalten festgelegten `DEFAULT`-Werte gesetzt. Dafür muss dann wiederum ein entsprechendes Tupel in der referenzierten Relation existieren. Beispiel: Eine Firmendatenbank, in denen jedes Projekt einem Mitarbeiter zugeordnet ist. Fällt dieser aus, werden seine Projekte zur Chefsache gemacht.

`SET NULL:`

Die `DELETE/UPDATE`-Operation auf der P-Tabelle wird ausgeführt. In der C-Tabelle wird der entsprechende Fremdschlüsselwert durch `NULL` ersetzt. Voraussetzung ist hier, dass `NULL`s zulässig sind.

Beispiel: Die Relation *located* gibt an, an welchem Fluss/See/Meer eine Stadt liegt, z. B.

located(Bremerhaven,Nds.,D,Weser,`NULL`,Nordsee)

Wird nun mit `DELETE * FROM River WHERE Name='Weser';`
das Tupel 'Weser' in der Relation River gelöscht, sollte die Information, dass Bremerhaven an der Nordsee liegt, erhalten bleiben:

located(Bremerhaven,Nds.,D,`NULL`,`NULL`,Nordsee)

8.2 Referentielle Aktionen in ORACLE

In ORACLE 9 sind nur `ON DELETE/UPDATE NO ACTION`, `ON DELETE CASCADE` und `ON DELETE SET NULL` implementiert `:-(`. Als Default wird `NO ACTION` angenommen; damit ist nur optional anzugeben, falls `ON DELETE CASCADE` oder `ON DELETE SET NULL` verwendet werden soll:

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

für `<columnConstraint>` bzw.

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

für `<tableConstraint>` (für mehrere Spalten).

Insbesondere die Tatsache, dass `ON UPDATE CASCADE` fehlt, ist beim Durchführen von Updates ziemlich lästig:

Beispiel 9 (Umbenennung eines Landes) Für die Tabelle *Country* ist das Kürzel *Code* als `PRIMARY KEY` definiert. *Code* wird u. a. in *Province* referenziert, d.h. ist dort *Fremdschlüssel*:

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

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

Die beiden Tabellen enthalten unter anderem die Tupel ('United Kingdom','GB') und ('Yorkshire','GB'). Nun soll das Landeskürzel von 'GB' nach 'UK' geändert werden.

- `UPDATE Country SET Code='UK' WHERE Code='GB'` führt zu einer "dangling reference" des Tupels ('Yorkshire','GB').
- will man zuerst `UPDATE Province SET Code='UK' WHERE Code='GB'` ändern, gibt es kein zu referenzierendes Tupel für ('Yorkshire','UK').

Damit muss man zuerst die referentielle Integritätsbedingung außer Kraft setzen, dann die Updates vornehmen, und danach die referentielle Integritätsbedingung wieder aktivieren:

```
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;
```
□

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 ...;
```

In dem oben beschriebenen Fall stellt "nur" das Ändern von Daten ein Problem dar – das mit `ON UPDATE CASCADE` gelöst werden könnte. Ein analoges Problem ergibt sich aus gegenseitigen und zyklischen Referenzen zwischen verschiedenen Tabellen:

```
CREATE TABLE Country
   ( Name VARCHAR2(32),
     Code VARCHAR2(4) PRIMARY KEY,
     Capital VARCHAR2(35),
     Province VARCHAR2(32),
     ⋮
     CONSTRAINT CountryCapRefsCity
        FOREIGN KEY (Capital,Code,Province)
        REFERENCES City(Name,Country,Province));

CREATE TABLE Province
   ( Name VARCHAR2(32),
     Country VARCHAR2(4),
     Capital VARCHAR2(35),
     ⋮
     PRIMARY KEY (Name, Country),
     CONSTRAINT ProvRefsCountry
        FOREIGN KEY (Country)
        REFERENCES Country(Name)),
     CONSTRAINT ProvCapRefsCity
        FOREIGN KEY (Capital,Country,Name)
        REFERENCES City(Name,Country,Prov));

CREATE TABLE City
   ( Name VARCHAR2(35),
     Country VARCHAR2(4),
     Province VARCHAR2(32),
     ⋮
     PRIMARY KEY (Name, Country, Province),
     CONSTRAINT CityRefsProv
        FOREIGN KEY (Country, Province)
        REFERENCES Province(Country, Name));
```

Ein Einfügen von Daten in diese Tabellen ist so nicht möglich, da in der jeweils anderen Tabelle noch nichts existiert. In diesem Fall wird man z. B. `CityRefsProv` bei der Tabellendefinition disablen, dann die Relationen *City*, *Province* und *Country* "von unten her" füllen, und danach das Constraint durch

```
ALTER TABLE Country
  ENABLE CONSTRAINT CityRefsProv;
```

aktivieren.

Wie bereits in Abschnitt 2.3 gesagt, können Tabellen, auf die noch eine referentielle Integritätsbedingung zeigt, mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden. Mit

```
DROP TABLE <table> CASCADE CONSTRAINTS;
```

wird eine Tabelle mit allen auf sie zeigenden referentielle Integritätsbedingungen gelöscht.

9 VIEWS – TEIL 2

9.1 View Updates

Views werden häufig (in Kombination mit der Vergabe von Zugriffsrechten, siehe Abschnitt10) dazu benutzt, den realen Datenbestand für Benutzer in einer veränderten Form darzustellen. Damit ein Benutzer in seiner Sicht(weise) Updates ausführen kann, müssen diese auf die Basisrelationen abgebildet werden. ORACLE verwendet Heuristiken, um aufgrund des Schemas festzustellen, ob eine solche Abbildung eindeutig möglich ist.

Über die Tabelle `USER_UPDATABLE_COLUMNS` des Data Dictionary kann der Benutzer abfragen, welche (View-)Spalten updatable sind (in dieser Abfrage ist es wichtig, den Tabellennamen in Großbuchstaben anzugeben!). In dem folgenden View können alle Spalten außer *Density* verändert werden. Da *Density* ein abgeleiteter Wert ist, kann diese Spalte natürlich nicht direkt verändert werden.

```
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';
```

Name	Code	Area	population	Density
Lummerland	LU	1	4	4

Da das View bei der Ausgabe aus der (durch das `INSERT` veränderten) aktuellen Basistabelle *Country* neu berechnet wird, enthält es auch den Wert für *Density*.

Werte, die durch Aggregatfunktionen berechnet wurden, sind in Views ebenfalls nicht veränderbar. Diese Fälle sind relativ einfach damit zu begründen sind, dass berechnete Werte

nicht geändert werden können.

```
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

Über diese Sicht können also Städte(namen) verändert werden. das Einfügen von Tupeln ist nicht möglich, da das Attribut *Country* nicht verändert/inserted werden darf, andererseits *Country* aber als Teil des Schlüssels einer neu eingefügten Stadt angegeben werden müsste.

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

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

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

Aber entgegen der oben angegebenen Werte können Daten in *CityCountry* gelöscht werden:

```
SQL> delete from CityCountry where country='Austria';
9 Zeilen wurden gelöscht.
```

Dieser Befehl löscht die betroffenen Städte aus *City*, löscht jedoch *Austria* nicht aus *Country*.

Ein spezielles Problem stellen Join-Views dar, bei denen mehrere Basistabellen verknüpft werden. Generell erlaubt ORACLE 8 nicht, dass ein View Update *mehrere* Basistabellen gleichzeitig verändert. Außerdem kommt es auch häufig vor, dass Werte zwar unverändert aus Basistabellen übernommen werden, und es trotzdem nicht möglich ist, eine eindeutige Abbildung der Änderungen auf die Basistabelle zu garantieren. Die ORACLE-Heuristiken basieren *nur* auf Schema-Informationen, betrachten also nicht, ob in der *gegebenen Datenbankinstanz* eine eindeutige Umsetzung möglich ist. Dabei spielen Schlüsseleigenschaften eine wichtige Rolle:

- Ist der Schlüssel einer Basistabelle auch gleichzeitig Schlüssel des Views, ist eine eindeutige Umsetzung möglich.
- umfasst der Schlüssel einer Basistabelle einen Schlüssel des Views, ist eine Umsetzung möglich (wobei eine Veränderung/Löschung an einem Tupel des Views möglicherweise mehrere Tupel der Basistabelle beeinflusst):

```
CREATE OR REPLACE VIEW temp AS
SELECT country, name, 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 Country = 'D';
```

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

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

Country	Name	Population
D	Bayern	11921944

- Ein Einfügen in das folgende View ist nicht möglich, da der Schlüssel der Basisrelation (*Province.Name* und *Province,Country* nicht komplett in den Attributen des Views enthalten ist.

```
CREATE OR REPLACE VIEW CountryProvPop AS
SELECT country, population
FROM Province A;
```

- Umfasst der Schlüssel einer Basistabelle keinen Schlüssel des Views komplett, ist keine eindeutige Umsetzung mehr möglich (siehe Aufgaben).

Bei Join-Views hat man allgemein das Problem, das man meistens einen Equi-Join über Schlüsselattribute bildet, wobei nur eines der Attribute dann in dem View auftritt – das andere (das zwar mit dem anderen übereinstimmt, aber eben formal nicht dasselbe Attribut ist wird dann nicht auf die darunterliegende Basistabelle umgesetzt.

In dem obigen Beispiel wurde ein Tupel eines Views so modifiziert, dass es aus dem Wertebereich des Views hinausfiel. Da Views häufig verwendet werden, um den "Aktionsradius" eines Benutzers einzuschränken, ist dies in vielen Kontexten unerwünscht und kann durch `WITH CHECK OPTION` verhindert werden:

Beispiel 10 Ein Benutzer soll *nur* mit US-amerikanischen Städte arbeiten.

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

UPDATE UScities
```

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

liefert die Fehlermeldung

```
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*. □

9.2 Materialized Views; View Maintenance

Views werden bei jeder Anfrage neu berechnet. Dies hat den Vorteil, dass sie immer den aktuellen Datenbankzustand repräsentieren. Bei der Verwendung großer Views über teilweise nur selten veränderten Daten – wie sie im realen Leben häufig vorkommen – ist eine ständige komplette Neuberechnung jedoch ineffizient. Zu diesem Zweck können *Materialized Views* eingesetzt werden, die bei jeder Datenänderung automatisch aktualisiert werden (dies kann u. a. durch Trigger (vgl. Abschnitt 14 geschehen). Materialized Views werden im Praktikum nicht behandelt. Die Probleme, die sich aus der Aktualisierung ergeben werden unter dem Stichwort *View Maintenance* zusammengefasst.

Sowohl *View Updates* als auch *View Maintenance* sind aktuelle Forschungsthemen (Theorie und Implementierung).

10 ZUGRIFFSRECHTE

Jeder Benutzer weist sich gegenüber ORACLE durch seinen Benutzernamen und sein Passwort aus.[1] Dem Benutzernamen werden vom Datenbankadministrator (DBA) Zugriffsrechte erteilt. Der DBA hat somit eine Schlüsselfunktion in der Verwaltung der Datenbank. In der Praxis fallen u. a. folgende Tätigkeiten in seinen Aufgabenbereich: Einrichten der Datenbank, Verwaltung der Benutzerrechte, Performance-Tuning der Datenbank, Durchführung von Datensicherungsmaßnahmen und "Beaufsichtigung" der Benutzer (Auditing).

Bei der Vergabe von Zugriffsrechten an die Benutzer wird meist nach der Regel verfahren, das jeder Benutzer so wenige Rechte wie möglich bekommt - damit er möglichst wenig Schaden in der Datenbank anrichten kann - und so viele Rechte wie nötig um seine Anwendungen durchführen zu können.

Schemakonzept. Die Organisation von Tabellen in ORACLE basiert auf dem *Schemakonzept*: Jedem Benutzer ist sein *Database Schema* zugeordnet, in dem er sich defaultmäßig aufhält und in dem er Objekte erzeugt und Anfragen an sie stellt. Die Bezeichnung der Tabellen geschieht somit global durch `<username>.<table>`, wird bei einer Anfrage durch einen Benutzer das Schema nicht angegeben, wird automatisch das entsprechende Objekt aus dem eigenen Schema angesprochen.

Die Zugriffsrechte teilen sich in zwei Gruppen: "Systemprivilegien", die z. B. zu Schemaoperationen berechtigen, sowie "Objekt"rechte, d.h., was man dann mit den einzelnen Objekten machen kann.

Systemprivilegien. In ORACLE existieren über 80 verschiedene Systemberechtigungen. Jede Systemberechtigung erlaubt einem Benutzer die Ausführung einer bestimmten Datenbankoperation oder Klasse von Datenbankoperationen. Auch bestimmte Schema-Objekte, wie z. B. Cluster, Indices und Trigger werden nur durch Systemberechtigungen gesteuert. Da Systemberechtigungen oftmals weitreichende Konsequenzen haben, darf sie in der Regel nur der DBA vergeben.

- `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/TYPE/VIEW/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.

[1] Im Rahmen des Praktikums wird durch den Aufruf von `sqlplus /` der UNIX-Account zur Autorisierung verwendet.

ANY berechtigt dazu die entsprechende Operation in *jedem* Schema auszuführen, fehlt ANY, so bleibt die Ausführung auf das eigene Schema beschränkt. Die Teilnehmer des Praktikums sollten etwa mit den Privilegien CREATE SESSION, ALTER SESSION, CREATE TABLE, CREATE TYPE, CREATE CLUSTER, CREATE SYNONYM, CREATE VIEW ausgestattet sein. Dabei sind die Zugriffe und Veränderungen an den eigenen Tabellen nicht explizit aufgeführt (würde logischerweise SELECT TABLE heißen).

Will man auf eine Tabelle zugreifen, deren Zugriff man nicht gestattet ist, etwa SELECT * FROM Kohl.Private, bekommt man nur die Meldung ORA-00942: Tabelle oder View nicht vorhanden.

Systemprivilegien werden mit Hilfe der GRANT-Anweisung erteilt:

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

<privilege-list> ist dabei eine Aufzählung der administrativen Privilegien, die man jemandem erteilen will. Mit der <user-list> gibt man an, wem diese Privilegien erteilt werden; mit PUBLIC erhält jeder das Recht. Wird ein Systemprivileg zusätzlich mit ADMIN OPTION vergeben, so kann der Empfänger es auch weiter vergeben.

Informationen bzgl. der Zugriffsrechte werden in Tabellen des Data Dictionary gespeichert.[2] Mit SELECT * FROM SESSION_PRIVS erfährt man, welche generellen Rechte man besitzt.

Objektprivilegien. Jedes Datenbankobjekt hat einen Eigentümer, dies ist meistens derjenige der es angelegt hat (wozu er allerdings auch wieder das Recht haben muss). Prinzipiell darf niemand sonst mit einem solchen Objekt arbeiten, außer der Eigentümer oder der DBA erteilt ihm explizit entsprechende Rechte:

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

Hierbei wird das betreffende Objekt durch <object> beschrieben und kann von der Art TABLE, VIEW, PROCEDURE/FUNCTION/PACKAGE oder TYPE sein. <privilege-list> ist eine Aufzählung der Privilegien, die man erteilen will. Mögliche Privilegien sind CREATE, ALTER und DROP für die verschiedenen Arten von Schemaobjekten, DELETE, INSERT, SELECT und UPDATE für Tabellen[3] und Views, außerdem INDEX, und REFERENCES für Tabellen. Für INSERT, REFERENCES und UPDATE kann optional angegeben werden, für welche Spalten das Recht vergeben wird (bei DELETE ist das natürlich nicht erlaubt und resultiert auch in einer Fehlermeldung[4]. Für Prozeduren/Funktionen/Packages und Typen kann man EXECUTE vergeben. Mit ALL gibt man alle Privilegien die man an dem beschriebenen Objekt hat, weiter.

- SELECT, UPDATE, INSERT, DELETE klar, die entsprechenden Zugriffs- und Änderungsoperationen,

[2] Zur Erinnerung: alle Tabellen, die im Data Dictionary enthalten sind, können mit SELECT * FROM DICTIONARY erfragt werden.

[3] dabei kann man mit UPDATE-Recht weder löschen noch einfügen.

[4] für Oracle 8.0.3 getestet.

- `REFERENCES`: das Recht, Spalten einer Tabelle als Fremdschlüssel in einer `CREATE TABLE`-Anweisung zu deklarieren und referentielle Aktionen anzugeben,
- `INDEX` das Recht, Indexe auf dieser Tabelle zu erstellen.
- `EXECUTE` das Recht, eine Prozedur, Funktion oder einen Trigger auszuführen, oder einen Typ zu benutzen.

Mit der `<user-list>` gibt man an, wem diese Privilegien erteilt werden; mit `PUBLIC` erhält jeder das Recht. Wird ein Recht mit `GRANT OPTION` vergeben, kann derjenige, der es erhält, es auch weiter vergeben.

Rechte entziehen. Gelegentlich ist es erforderlich, jemandem ein Recht (oder nur die `GRANT OPTION`) zu entziehen; dies kann man natürlich nur machen, wenn man dieses Recht selbst vergeben hat:

Administrative Rechte:

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

Objekt-Rechte:

```
REVOKE <privileg-list> | ALL
ON TABLE <table>
FROM <user-list> | PUBLIC [CASCADE CONSTRAINTS];
```

`CASCADE CONSTRAINTS` sorgt dafür, dass alle referentiellen Integritätsbedingungen, die auf einem entzogenen `REFERENCES`-Privileg beruhen, wegfallen.

- Hat ein Benutzer eine Berechtigung von mehreren Benutzern erhalten, behält er sie, bis sie ihm jeder einzelne entzogen hat.
- Hat ein Benutzer eine Berechtigung mit `ADMIN OPTION` oder `GRANT OPTION` weitergegeben und bekommt sie entzogen, so wird sie automatisch auch allen Benutzern entzogen, an die er sie weitergegeben hat.

Mit

```
SELECT * FROM USER_TAB_PRIVS;
```

kann man sich die Informationen über Tabellen ausgegeben lassen, die einem gehören und auf die man irgendwelche Rechte vergeben oder die jemandem anderem gehören und man Rechte dafür bekommen hat. Hat man Rechte nur für bestimmte Spalten einer Tabelle vergeben oder erhalten, erfährt man dies mit

```
SELECT * FROM USER_COL_PRIVS;
```

Die Tabellen `USER_TAB/COL_PRIVS_MADE` und `USER_TAB/COL_PRIVS_RECD` enthalten im einzelnen die vergebenen oder erhaltenen Rechte für Tabellen bzw. Spalten.

Rollenkonzept. Neben der direkten Vergabe von Privilegien an einzelne Beutzer kann die Vergabe von Privilegien in Oracle auch durch *Rollen* geregelt werden. Benutzer die über die gleichen Rechte verfügen sollen werden dann einer bestimmten Rolle zugeordnet. Das Rollenkonzept vereinfacht somit auch die Rechtevergabe für den DBA, da die Rechte nur einmal der Rolle zugeordnet werden, anstatt dies für jeden einzelnen Benutzer zu wiederholen.

Rollen können hierarchisch strukturiert sein. Dabei muss darauf geachtet werden, dass keine Zyklen bei der Zuweisung entstehen. Die Zuordnung von Benutzern zu einer Rolle erhält man (falls man die DBA-Tabellen lesen darf) über folgende Anfrage

```
SELECT * FROM DBA_ROLE_PRIVS;
```

11 SYNONYME

Synonyme können verwendet werden, um ein Schemaobjekt unter einem anderen Namen als ursprünglich abgespeichert anzusprechen. Dies ist insbesondere sinnvoll, wenn man eine Relation, die in einem anderen Schema liegt, nur unter ihrem Namen ansprechen will.

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

Ohne den Zusatz `PUBLIC` ist das Synonym nur für den Benutzer definiert, der es definiert hat. Mit `PUBLIC` ist das Synonym systemweit verwendbar. Das dazu notwendige `CREATE ANY SYNONYM`-Privileg haben i.a. nur die SysAdmins.

Beispiel: Wenn ein Benutzer keine eigene Relation "City" besitzt, sondern stattdessen immer die Relation "City", aus dem Schema "dbis" verwendet, müsste er diese immer mit `SELECT * FROM dbis.City` ansprechen. Definiert man

```
CREATE SYNONYM City
  FOR dbis.City;
```

so kann man sie auch über `SELECT * FROM City` erreichen.

Man kann übrigens ein Synonym auch für eine Tabelle definieren, die (noch) nicht existiert und/oder auf die man noch kein Zugriffsrecht hat. Wird die Tabelle dann definiert und erhält man das Zugriffsrecht, tritt das Synonym in Kraft.

Synonyme werden mit `DROP SYNONYM <synonym>` gelöscht.

12 ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS

Bei der Aufstellung der Objektprivilegien im vorhergehenden Abschnitt lässt sich feststellen, dass bei `GRANT SELECT` der Zugriff nicht auf Spalten eingeschränkt werden kann. Dies kann allerdings einfach über Views (vgl. Abschnitt 2.2) geschehen:

Der *nicht existierende* Befehl

```
GRANT SELECT [<column-list>]
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 ];
```

Beispiel: Der Benutzer `pol` ist Besitzer der Relation "Country". Er entscheidet sich, seinem Kollegen `geo` die Daten über die Hauptstadt und ihre Lage nicht zur Verfügung zu stellen. Alle restlichen Daten sollen von `geo` aber gelesen *und* geschrieben werden können.

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

GRANT SELECT ON pubCountry TO geo;
```

Da das View über Basis *genau einer* Basisrelation definiert ist und der von Basisrelation und View übereinstimmt, sind Updates und Einfügungen erlaubt.

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

Zusätzlich wird `geo` sich ein Synonym definieren:

```
CREATE SYNONYM Country FOR pol.pubCountry;
```

Ärgerlich ist in diesem Zusammenhang nur, dass man keine Referenzen auf Views legen kann (diese haben keinen Primary Key): Um Country.Code zu referenzieren, muss `geo` trotzdem die *Basistabelle* `pol.pubCountry` verwenden:

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

13 OPTIMIERUNG DER DATENBANK

Dieser Abschnitt behandelt die in ORACLE vorhandenen Konstrukte, um Speicherung und Zugriff einer Datenbank zu optimieren. Diese Konstrukte dienen dazu, (a) effizienter bestimmte Einträge zu finden (Indexe) und (b) physikalisch schneller zugreifen zu können (Cluster). (a) ist dabei ein algorithmisches Problem, betreffend die Suche nach Werten; (b) soll die Zugriffe auf den Hintergrundspeicher minimieren.

Diese Strukturen haben keine Auswirkung auf die Formulierung der restlichen SQL-Anweisungen!

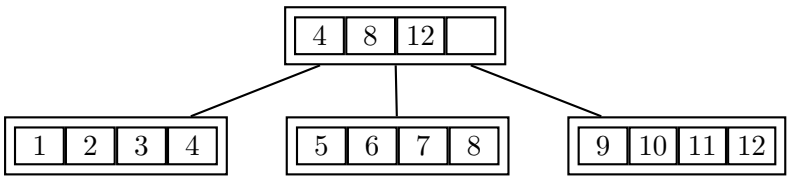
13.1 Indexe

Indexe unterstützen den Zugriff auf eine Tabelle über einen gegebenen Spaltenwert:

```
CREATE TABLE PLZ
  (City    VARCHAR2(35)
   Country VARCHAR2(4)
   ProvinceVARCHAR2(32)
   PLZ     NUMBER)
```

- Indexe in ORACLE als B-Baum organisiert,
- logisch und physikalisch unabhängig von den Daten der zugrundeliegenden Tabelle,
- keine Auswirkung auf die Formulierung einer SQL-Anweisung,
- mehrere Indexe für eine Tabelle möglich,
- bei sehr vielen Indexen auf einer Tabelle kann es beim Einfügen, Ändern und Löschen von Sätzen zu Performance-Verlusten kommen.

B-Baum:



- Logarithmische Höhe des Baumes,
- B-Baum: Knoten enthalten auch die Werte der Tupel,
- B*-Baum: Knoten enthalten *nur* die Weg-Information, daher ist der Verzweigungsgrad sehr hoch, und die Höhe des Baumes auch absolut gesehen gering.
- Schneller Zugriff (logarithmisch) versus hoher Reorganisationsaufwand (→ Algorithmentechnik-Vorlesung).

Indexarten:

eindeutige Indexe:
- keine gleichen Werte in der Spalte des Indexe erlaubt,
- bei einer `PRIMARY KEY`-Deklaration erzeugt ORACLE automatisch einen Index über diese Spalte(n).

mehrdeutige Indexe:
- gleiche Werte sind erlaubt.

zusammengesetzte Indexe:
- Index besteht aus mehreren Spalten (Attributen) einer Tabelle
- sinnvoll, wenn häufig `WHERE`-Klauseln über mehrere Spalten formuliert werden (z. B. (L_ID,LT_ID)).

```
CREATE [UNIQUE] INDEX <name> ON <table>(<column-list>);
```

Im obigen Beispiel wäre ein Index über L_ID, PLZ sinnvoll:

```
CREATE INDEX PLZIndex ON PLZ (Country,PLZ);
SELECT *
  FROM plz
  WHERE plz = 79110 AND Country = 'D';
```

Indexe werden mit `DROP INDEX` gelöscht.

Wichtig: Indexe sind physikalische Strukturen, die in der Datenbank gespeichert sind, während Schlüssel ein rein logisches Konzept sind. Damit gehören Indexe *nicht* zur Modellierung (ER- und relationale Modellierung), sondern werden erst betrachtet, wenn das Datenbankschema entworfen wird.

13.2 Bitmap-Indexe

Die Motivation der oben genannten Indexe (insbesondere als `UNIQUE INDEX`) liegt in erster Linie darin, den Zugriff zu einzelnen Elementen zu verbessern indem nicht mehr die gesamte Relation linear durchsucht werden muss. Erst auf den zweiten Blick wird erkennbar, dass damit auch alle Werte mit demselben Indexwert effizient zugegriffen werden können.

Bei `BITMAP`-Indexen liegt das Hauptinteresse darauf, zu bestimmten Attributen, die nur wenige verschiedene Wert annehmen, *alle* Tupel mit einem bestimmten Attributwert optimal selektieren zu können: Für die entsprechenden Attribute wird eine Bitmap angelegt, die für jeden vorkommenden Wert ein Bit enthält. Dieses wird für ein Tupel genau dann gesetzt, wenn das Tupel den entsprechenden Attributwert enthält:

```
CREATE BITMAP INDEX <name> ON <table>(<column-list>);
```

Beispiel 11 (Bitmap-Index) Die Relation *encompasses* kann vorteilhaft mit einem Bitmap-Index über Kontinent erweitert werden:

```
CREATE BITMAP INDEX continents_index ON encompasses(continent);
```

Country	**Perc.**
D	100
CH	100
USA	100
R	20
R	80
ZA	100
NZ	100

EUR	AS	AM	AF	AUS
×				
×				
		×		
×				
	×			
			×	
				×

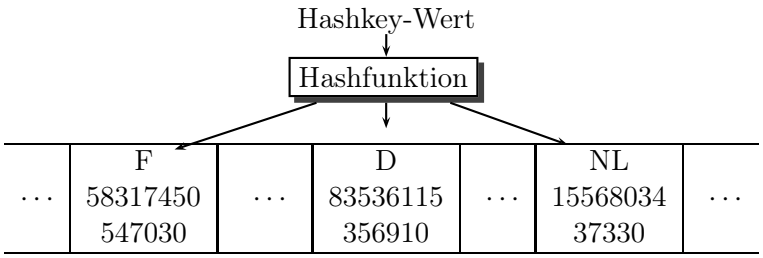
□

Bitmap-Indexe haben insbesondere dann Performanz-Vorteile, wenn eine `WHERE`-Klausel mehrere solche Spalten auswertet und damit auf eine Verknüpfung auf logischer (Bit)-Ebene auf diesen Indexen reduziert werden kann. Bitmap-Indexe sind damit maßgeschneidert für Data-Warehousing-Anwendungen.

13.3 Hashing

Hashing unterstützt Zugriff auf eine Tabelle über bestimmte Attributwerte (eben die, die als *Hashkey* definiert sind). Dabei wird aufgrund dieser Attributwerte *in konstanter Zeit* berechnet, wo die entsprechenden Tupel zu finden sind: Eine Hash-Funktion bildet jeden Wert des angegebenen Hashkeys auf eine Zahl innerhalb eines festgelegten Hash-Bereiches ab. Die Vorgehensweise ist damit ähnlich zu einem Index.

Z.B. wäre ein Hash-Zugriff sinnvoll, wenn um auf die Daten über ein bestimmtes Land gezielt zugreifen zu können: Hashkey ist *Country.Code*.



Vorteile der Hash-Methode.

- falls Anzahl der Hash-Werte ausreichend groß, genügt ein Blockzugriff, um einen Datensatz zu finden.
- kein Zugriff auf Indexblöcke erforderlich.

In Oracle 8 ist Hashing nur für Cluster implementiert.

13.4 Cluster

Cluster unterstützen Zugriff auf mehrere Tabellen mit gemeinsamen Spalten durch physikalisches Zusammenfassen der Tabellen nach diesen Spalten, um jeweils bei einem Hintergrundspeicherzugriff semantisch zusammengehörende Daten in den Hauptspeicher zu bringen.

- Zusammenfassung einer Gruppe von Tabellen, die alle eine oder mehrere gemeinsame Spalten (Clusterschlüssel) besitzen,

- sinnvoll bei referentiellen Integritätsbedingungen (Fremdschlüsselbeziehungen) oder bei häufigen `JOIN`-Anweisungen über dieselbe Spalte(n),
- auch zu einer einzelnen Tabelle kann ein Cluster erzeugt werden, z. B. um Städte nach Landesteilen geordnet abzuspeichern.

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

Wird ein Cluster erstellt, muss zuerst dessen Cluster-Schlüssel angegeben werden, um die in diesem Cluster gespeicherten Tabellen geeignet zu organisieren.

Erzeugen eines Clusters:

1. Cluster und Clusterschlüssel definieren,
2. Tabellen erzeugen und mit der Angabe der Spalten für den Clusterschlüssel in den Cluster einfügen,
3. Clusterindex über den Clusterschlüssel-Spalte(n) definieren. Dies muss *vor* dem ersten DML-Kommando geschehen.

In Mondial gibt es einige Relationen (*Mountain*, *Lake*, ...), in denen zusammengehörende Daten über auf zwei Relationen verteilt sind: Z.B. enthält *Sea* die topographischen Informationen über ein Meer, während *geo_Sea* dessen Lage in Bezug auf geopolitische Begriffe bezeichnet. Diese könnte man clustern. Ebenso ist es sinnvoll, die Relation *City* nach (*Country, Province*) zu clustern. Beide Cluster sind in Abb. 13.1 und 13.2 gezeigt.

Diese Speicherstruktur erreicht man folgendermaßen:

Zuerst wird ein Cluster erzeugt, und angegeben, welche Spalten (und ihre Datentypen) den Clusterschlüssel bilden sollen:

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

Als Default wird ein *indexed Cluster* erstellt, d.h. die Zeilen werden entsprechend ihren Clusterschlüsselwerten indiziert und geclustert.

Gibt man `HASH` und eine Hashfunktion an wird aus den Zeilen eines Tupels dessen Hashwert (mit `<funktion>` falls angegeben, sonst intern) berechnet, der dann modulo `<integer>` genommen wird. Nach diesem Wert wird dann geclustert (wobei vollkommen unzusammenhängende Zeilen zusammen geclustert werden können).

Die Tabellen werden durch ein weiteren Befehl in der `CREATE TABLE`-Definition einem Cluster zugeordnet. Dabei muss die Zuordnung der Spalten zum Clusterschlüssel angegeben werden:

```
CREATE TABLE <table>
```

Sea		
Mediterranean Sea	**Depth**	
	5121	
	Province	**Country**
	Catalonia	E
	Balearic Isl.	E
	Valencia	E
	Murcia	E
	Andalusia	E
	Corse	F
	Languedoc-R.	F
	Provence	F
	⋮	⋮
Baltic Sea	**Depth**	
	459	
	Province	**Country**
	Schleswig-H.	D
	Mecklenb.-Vorp.	D
	Szczecin	PL
	Koszalin	PL
	Gdansk	PL
	Olsztyn	PL
	⋮	⋮

Abbildung 13.1: Geclusterte Tabellen

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

Zum Schluss muss noch der Clusterschlüsselindex erzeugt werden:

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

Mit den folgenden Befehlen kann man den oben beschriebenen Cluster *Cl_Sea* generieren:

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

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

CREATE TABLE Cgeo_Sea
  (ProvinceVARCHAR2(35),
```

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

Abbildung 13.2: Geclusterte Tabellen

```
  Country VARCHAR2(4),
  Sea     VARCHAR2(25))
CLUSTER Cl_Sea (Sea);

CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```

Alle drei Methoden erfordern relativ hohen Aufwand, wenn sie reorganisiert werden müssen (Überlaufen von Indexknoten, Cluster-Bereichen, Hash-Speicherbereichen oder des Wertebereichs der Hash-Funktion).