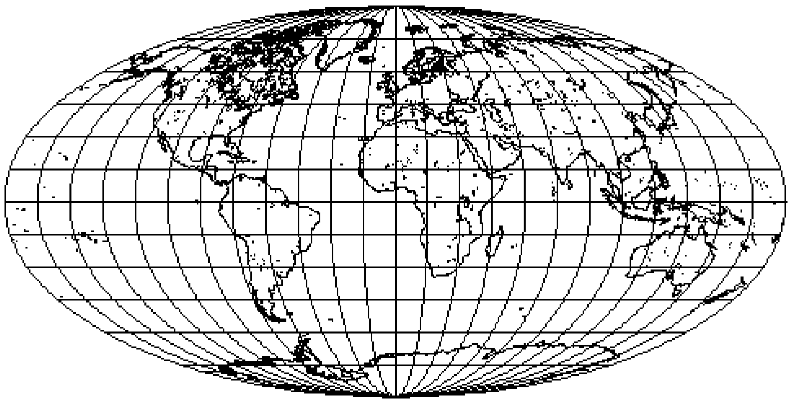
Praktikum: Datenbankprogrammierung in SQL/ORACLE



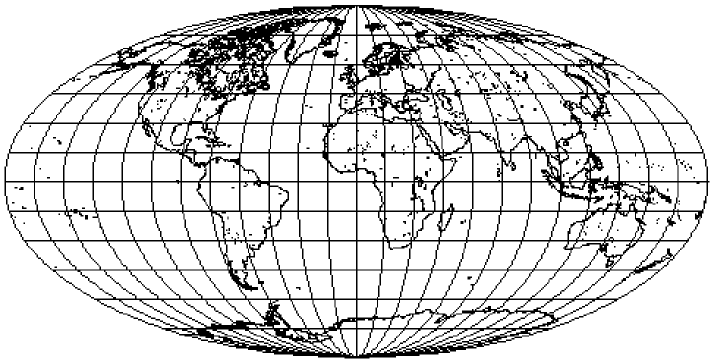
Wolfgang May

Termin: Mittwochs, 14 Uhr ct. – $\leq$ 17.00 Uhr,
Geb. 51, Raum 00-006.

SQL-3 Standard/ORACLE 8:

- ER-Modellierung
- Schemaerzeugung
- Anfragen
- Views
- Komplexe Attribute, geschachtelte Tabellen
- Optimierung
- Zugriffskontrolle
- Transaktionen
- Updates, Schemaänderungen
- Referentielle Integrität
- PL/SQL: Trigger, Prozeduren, Funktionen
- Objektrelationale Features
- Embedded SQL
- JDBC (Einbindung in Java)

Diskurswelt: MONDIAL



- Kontinente
- Länder
- Landesteile
- Städte
- Organisationen
- Berge
- Flüsse
- Seen
- Meere
- Wüsten
- Wirtschaft
- Bevölkerung
- Sprachen
- Religionen
- Ethn. Gruppen

- CIA World Factbook
- "Global Statistics": Länder, Landesteile, Städte
- TERRA-Datenbasis des Instituts für Programmstrukturen und Datenorganisation der Universität Karlsruhe
- ... einige weitere WWW-Seiten
- Datenintegration mit *FLORID*

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Organisation

- Skripte zum Erstellen der Datenbasis:

  /home/dbis/MondialDBP/
  Lesen für alle, Schreiben für Betreuer.

- Gruppenarbeit:

  `/home/dbis/db-prakt/gr1/`
  `.../home/dbis/db-prakt/grn/`
  Eigene SQL-Skripte.
  Lesen und Schreiben für alle Gruppenmitglieder.

- ORACLE-Datenbank:

  Jeder hat seinen eigenen ORACLE-Account mit einer kompletten Datenbank.

  ⇒ Updates problemlos möglich.

  Vergabe von Lese-/Schreibrechten an Gruppenmitglieder.
  Eigene Datenbasis: `SELECT FROM CITY;`
  Fremde Datenbasis: `SELECT FROM <name>.CITY;`

Literatur

- Lehrbücher zu Datenbanken (In Deutsch):

  A. Kemper, A. Eickler: Datenbanksysteme - Eine Einführung, Oldenbourg, 1996

  G. Vossen: Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme. Addison-Wesley, 1994.

- Lehrbuch zu SQL (In Deutsch):
  G. Matthiessen and M. Unterstein: Relationale Datenbanken und SQL: Konzepte der Entwicklung und Anwendung. Addison-Wesley, 1997.

- Das Buch zum DB-Praktikum der Uni KA mit TERRA:
  M. Dürr and K. Radermacher: Einsatz von Datenbanksystemen. Springer Verlag, 1990.

- Das Buch zum SQL-2 Standard:
  C. Date and H. Darwen: A guide to the SQL standard: a user's guide to the standard relational language SQL. Addison-Wesley, 1994.

- Lehrbücher über Relationale Datenbanken und SQL:

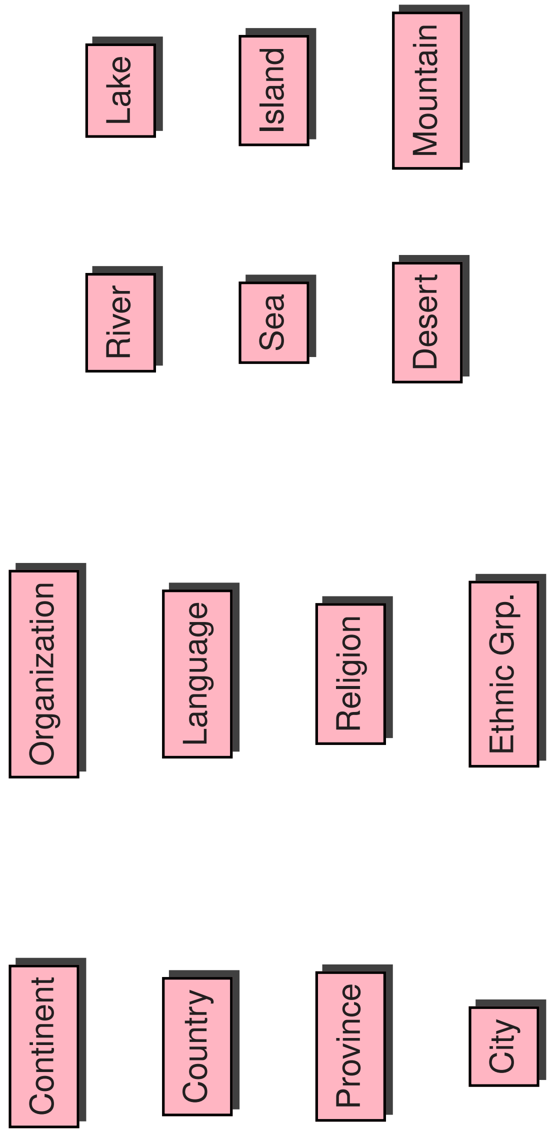
  H. F. Korth and A. Silberschatz: Database System Concepts. McGraw-Hill, 1991.

  J. Ullman and J. Widom: A First Course in Database Systems. Prentice Hall, 1997.

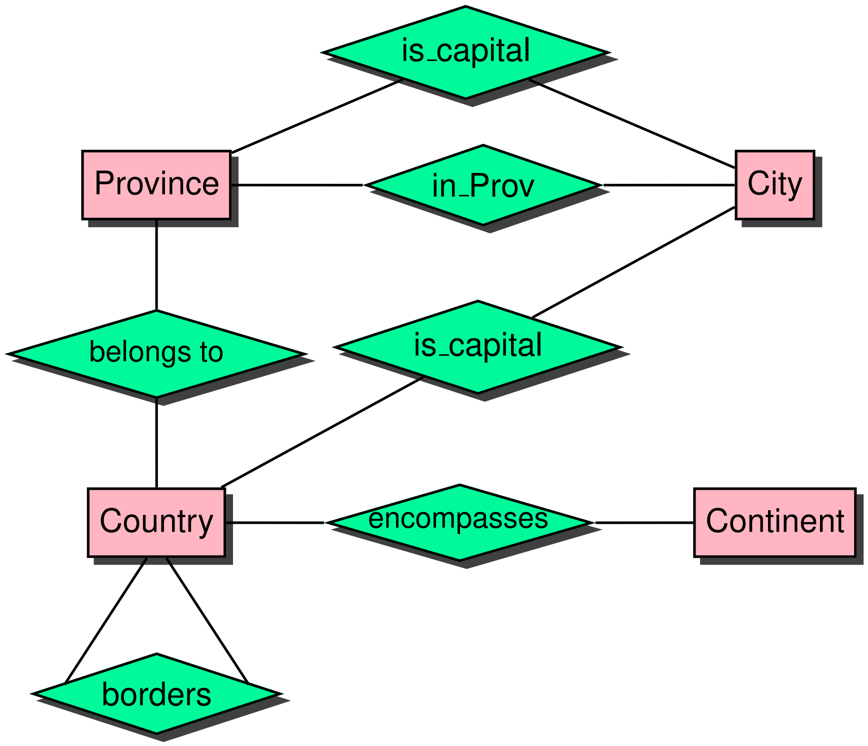
Semantische Modellierung:
Entity-Relationship Modell (ERM; Chen, 1976)

Strukturierungskonzepte zur Abfassung eines Schemas im ERM:

- Entitäts– (entity) Typen (≡ Objekttypen) und
- Beziehungs– (relationship) Typen



Entities und Beziehungen



Entities

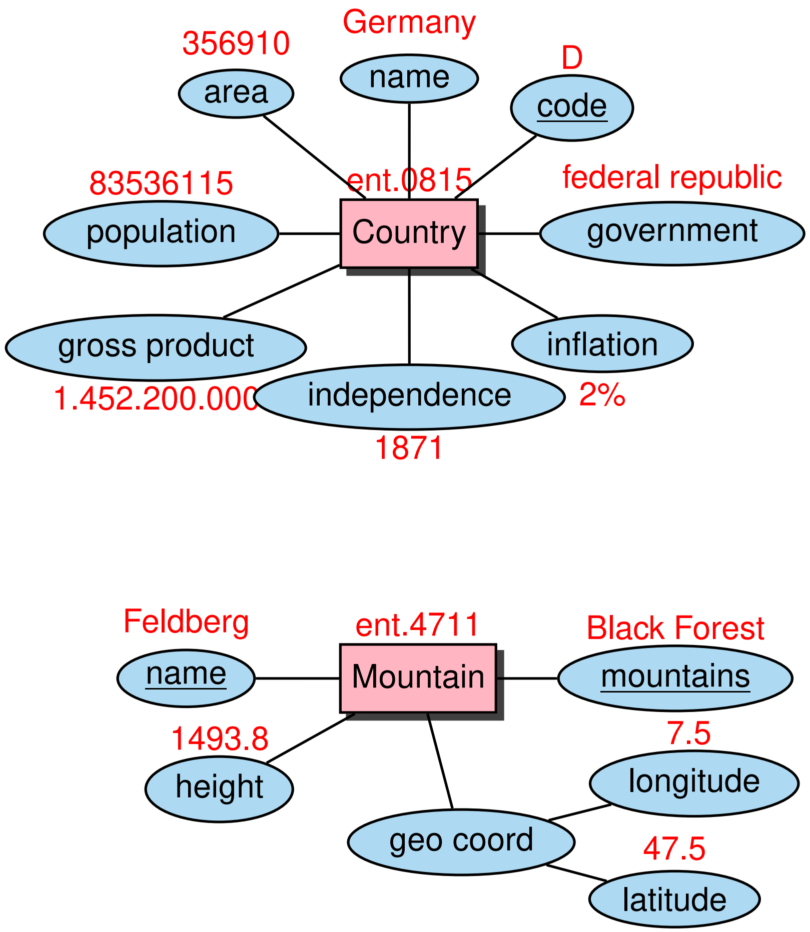
Entitätstyp ist durch ein Paar $(E, \{A_1, \ldots, A_n\})$ gegeben, wobei E der Name und $\{A_1, \ldots, A_n\}$, $n \geq 0$, die Menge der Attribute des Typs ist.

Attribut: Relevante Eigenschaft der Entitäten eines Typs. Jedes Attribut kann *Werte* aus einem bestimmten *Wertebereich (domain)* annehmen.

Entität: besitzt zu jedem Attribut ihres Entitätstyps E einen Wert.

Schlüsselattribute: Ein Schlüssel ist eine Menge von Attributen eines Entitätstyps, deren Werte zusammen eine eindeutige Identifizierung der Entitäten eines Zustands gewährleisten soll (siehe auch *Schlüsselkandidaten, Primärschlüssel*).

Entities:



Beziehungen

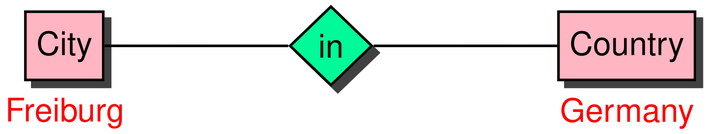
Beziehungstyp: Menge gleichartiger Beziehungen zwischen Entitäten; ein Beziehungstyp ist durch ein Tripel $(B, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_n\})$ gegeben, wobei B der Name, $\{RO_1, \ldots, RO_k\},\ k \geq 2$, die Menge der sog. Rollen, $\{E_1, \ldots, E_k\}$ die den Rollen zugeordnete Entitätstypen, und $\{A_1, \ldots, A_n\}, n \geq 0$, die Menge der Attribute des Typs sind.

Rollen sind paarweise verschieden - die ihnen zugeordneten Entitätstypen nicht notwendigerweise. Falls $E_i = E_j$ für $i \neq j$, so liegt eine **rekursive** Beziehung vor.

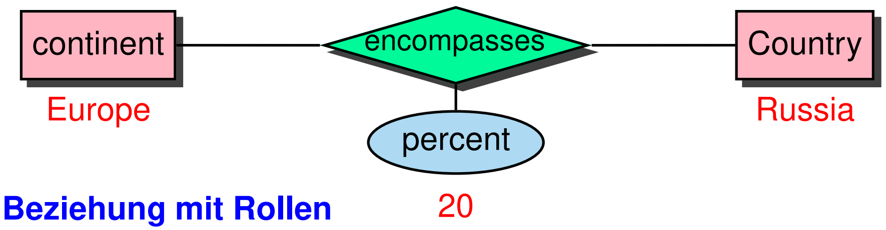
Attribut: Relevante Eigenschaft der Beziehungen eines Typs.

Beziehung: eines Beziehungstyps B ist definiert durch die beteiligten Entitäten gemäß den B zugeordneten Rollen; zu jeder Rolle existiert genau eine Entität und zu jedem Attribut von B genau ein Wert.

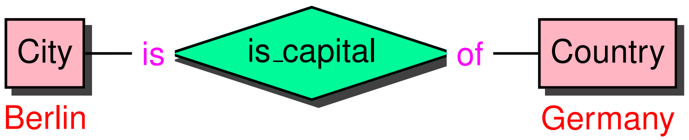
Beziehungen



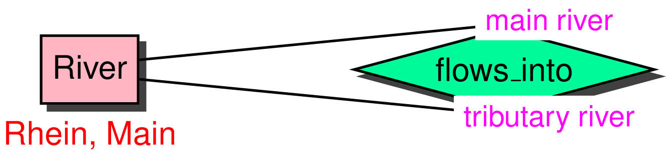
attributierte Beziehung



Beziehung mit Rollen



rekursive Beziehung



Praktikum: Datenbankprogrammierung in SQL/ORACLE

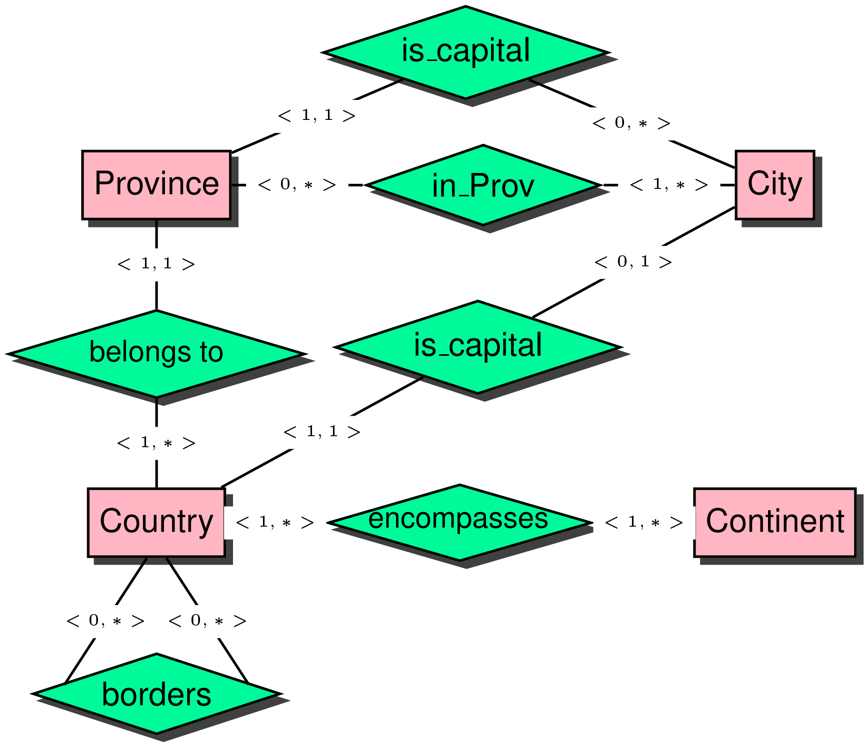
Beziehungskomplexitäten

Jedem Beziehungstyp ist eine Beziehungskomplexität zugeordnet, die die Mindest- und Maximalzahl von Beziehungen ausgedrückt, in denen eine Entität eines Typs unter einer bestimmten Rolle in einem Zustand beteiligt sein darf.

Ein **Komplexitätsgrad** eines Beziehungstyps B bzgl. einer seiner Rollen RO ist ein Ausdruck der Form (min, max).

Eine Menge b von Beziehungen erfüllt den Komplexitätsgrad (min, max) einer Rolle RO, wenn für jedes e des entsprechenden Entity-Typs gilt: es existieren mindestens min und maximal max Beziehungen in b, in denen e unter der Rolle RO auftritt.

Beziehungen

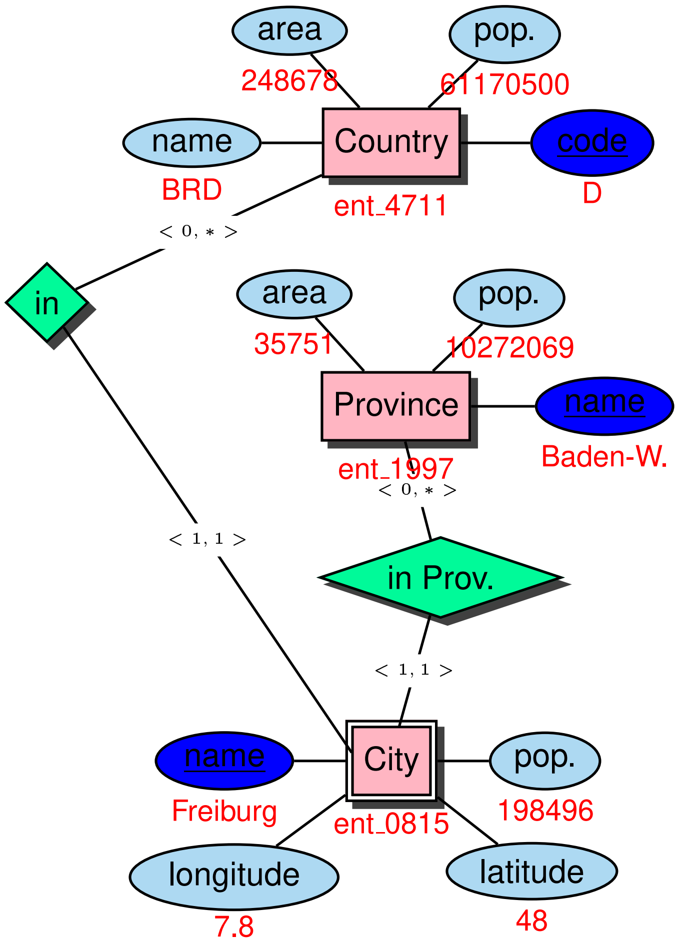


Schwache Entitätstypen

Ein schwacher Entitätstyp ist ein Entitätstyp ohne Schlüssel.

- Schwache Entitätstypen müssen mit mindestens einem (starken) Entitätstyp in einer $n : 1$-Beziehung stehen (auf der 1-Seite steht der starke Entitätstyp).
- Sie müssen einen **lokalen** Schlüssel besitzen, d.h. Attribute, die erweitert um den Primärschlüssel des betreffenden (starken) Entitätstyps einen Schlüssel des schwachen Entitätstyps ergeben (**Schlüsselvererbung**).

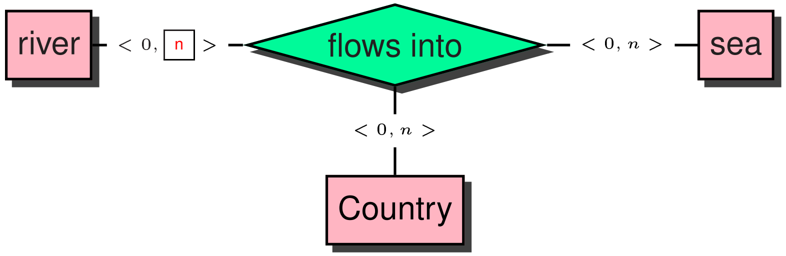
Schwache Entitätstypen



Es gibt z.B. noch ein Freiburg/CH
und Freiburg/Elbe, Niedersachsen

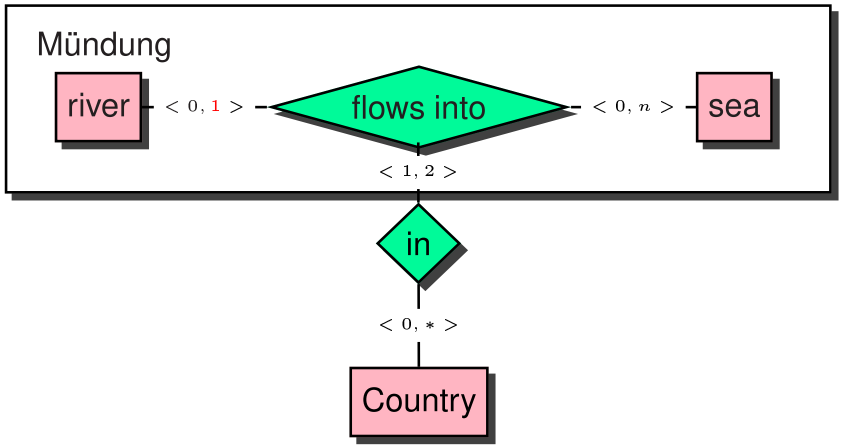
Mehrstellige Beziehungen

Ein Fluß mündet in ein Meer/See/Fluß; genauer kann dieser Punkt durch die Angabe eines oder zweier Länder beschrieben werden.



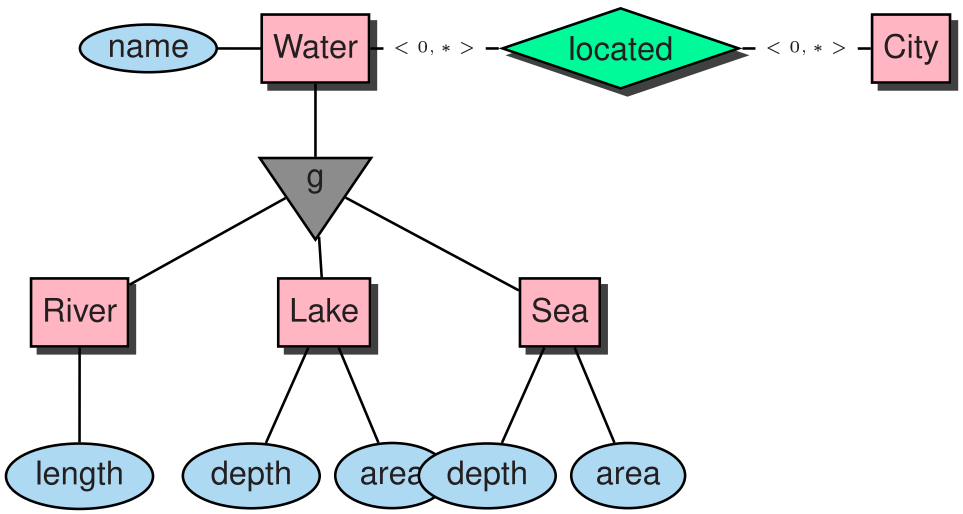
Aggregation

Sinnvoll, einen *Aggregattyp Mündung* einzuführen:



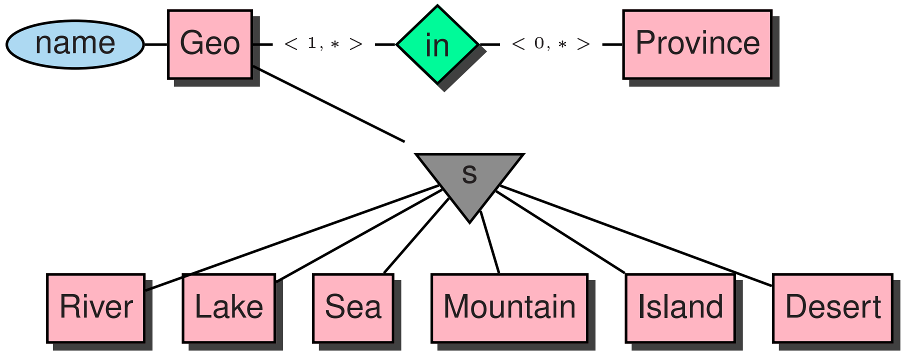
Generalisierung/Spezialisierung

- Generalisierung: Flüsse, Seen und Meere bilden die Menge der Gewässer. Diesen können z.B. mit Städten in einer *liegt-an*-Beziehung stehen:



Generalisierung/Spezialisierung:

- Spezialisierung: MONDIAL enthält nicht alle geographischen Merkmale, sondern nur Flüsse, Seen, Meere, Berge, Wüsten und Inseln (keine Tiefländer, Hochebenen, Steppengebiete, Moore etc). Allen geo-Merkmalen gemeinsam ist, dass sie in einer *in*-Beziehung zu Landesteilen stehen:



Das relationale Modell

- nur ein einziges Strukturierungskonzept *Relation* für Entitytypen *und* Beziehungstypen,
- Relationenmodell von Codd (1970): mathematisch fundierte Grundlage: Mengentheorie
- ein Relationsschema besteht aus einem Namen sowie einer Menge von Attributen,
  Continent: Name, Area
- Jedes Attribut besitzt einen Wertebereich, als *Domain* bezeichnet. Oft können Attribute auch *Nullwerte* annehmen.
  Continent: Name: VARCHAR(25), Area: NUMBER
- Die Elemente einer Relation werden als *Tupel* bezeichnet.
  (Asia,4.5E7)

Ein **(relationales) Datenbank-Schema** $\mathbf{R}$ ist gegeben durch eine (endliche) Menge von (Relations-)Schemata.
Continent: ...; Country: ...; City: ...

Ein **(Datenbank)-Zustand** ordnet den Relationsschemata eines betrachteten konzeptuellen Schemas jeweils eine **Relation** zu.

Abbildung ERM in RM

Seien E_{ER} ein Entitätstyp und B_{ER} ein Beziehungstyp im ERM.

1. Entitätstypen: $(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$,
2. Beziehungstypen:
   $(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
   $B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots,$
   $E_k\_K_{k1}, \ldots, E_k\_K_{kp_k},\ A_1, \ldots, A_m)$,
   wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
   Falls B_{ER} Rollenbezeichnungen enthält, so wird durch die Hinzunahme der Rollenbezeichnung die Eindeutigkeit der Schlüsselattribute im jeweiligen Beziehungstyp erreicht.
   Für $k = 2$ können im Falle einer (1,1)-Beziehungskomplexität das Relationsschema des Beziehungstyps und das Schema des Entitätstyps zusammengefaßt werden.
3. Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.
4. Aggregattypen können unberücksichtigt bleiben, sofern der betreffende Beziehungstyp berücksichtigt wurde.

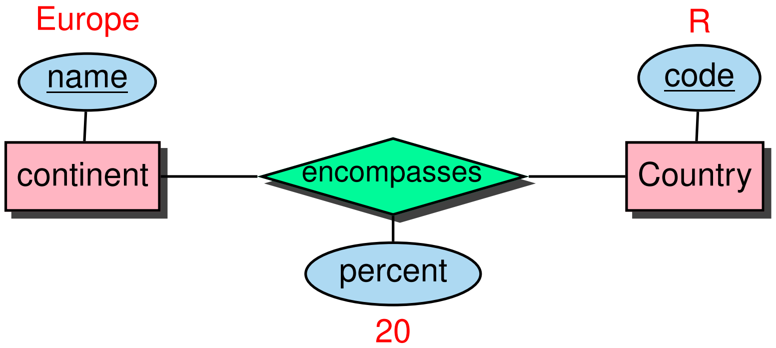
Entitätstypen

$$(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$$

Continent	
Name	**Area**
VARCHAR(20)	NUMBER
Europe	9562489.6
Africa	3.02547e+07
Asia	4.50953e+07
America	3.9872e+07
Australia	8503474.56

Beziehungstypen

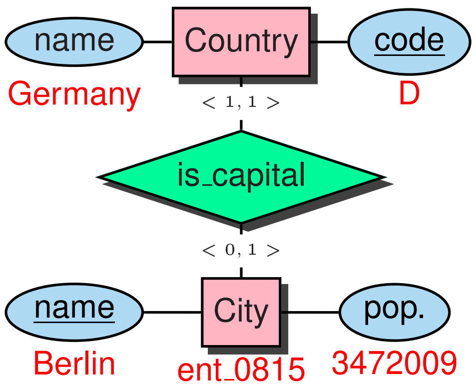
$(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
$B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots,$
$E_k\_K_{k1}, \ldots, E_k\_K_{kp_k}, \; A_1, \ldots, A_m),$
wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
(man darf aber umbenennen, z.B. *Country* für *Country.Code*)



encompasses		
Country	**Continent**	**Percent**
VARCHAR(4)	VARCHAR(20)	NUMBER
R	Europe	20
R	Asia	80
D	Europe	100
...	...	...

Beziehungstypen

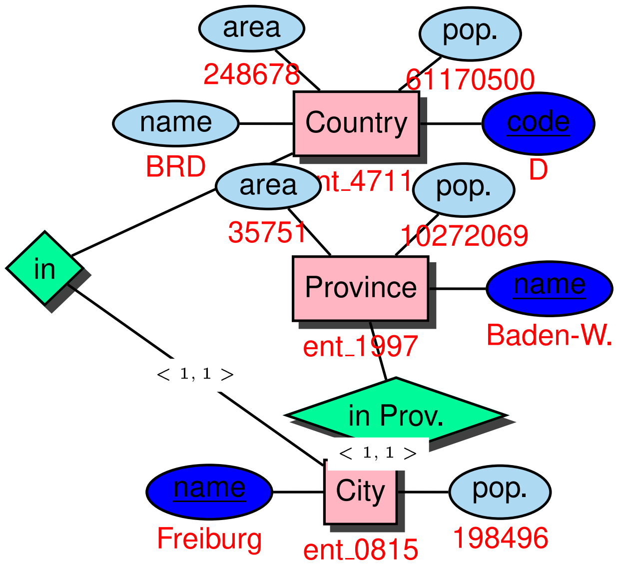
Für zweistellige Beziehungstypen können im Falle einer (1,1)-Beziehungskomplexität das Relationsschema des Beziehungstyps und das Schema des Entitätstyps zusammengefaßt werden:



Country					
Name	**code**	**Population**	**Capital**	**Province**	**...**
Germany	D	83536115	Berlin	Berlin	
Sweden	S	8900954	Stockholm	Stockholm	
Canada	CDN	28820671	Ottawa	Quebec	
Poland	PL	38642565	Warsaw	Warszwaskie	
Bolivia	BOL	7165257	La Paz	Bolivia	
..	..	..	..	..	

Schwache Entitätstypen

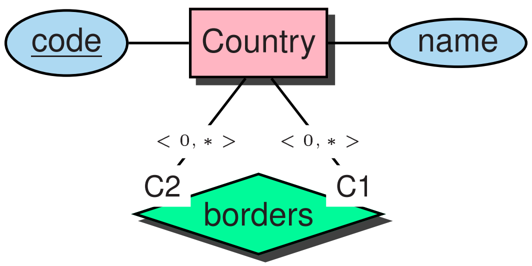
Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.



City				
Name	**Country**	**Province**	**Population**	**...**
Freiburg	D	Baden-W.	198496	..
Berlin	D	Berlin	3472009	..
..	..	..	..	..

Beziehungstypen

Falls B_{ER} Rollenbezeichnungen enthält, so wird durch die Hinzunahme der Rollenbezeichnung die Eindeutigkeit der Schlüsselattribute im jeweiligen Beziehungstyp erreicht.



borders	
Country1	Country2
D	F
D	CH
CH	F
..	..

Praktikum: Datenbankprogrammierung in SQL/ORACLE

SQL = Structured **Q**uery **L**anguage

- Standard-Anfragesprache
- Standardisierung: SQL-89, SQL-2 (1992), SQL-3 (1996)
- SQL-2 in 3 Stufen, entry, intermediate und full level eingeführt.
- SQL-3: Objektorientierung
- deskriptive Anfragesprache
- Ergebnisse immer Mengen von Tupeln (Relationen)
- Implementierung: ORACLE 8

- SQL ist case-insensitive, d.h. CITY=city=City=cItY.
- Innerhalb von Quotes ist SQL nicht case-insensitive, d.h. City='Berlin' $\neq$ City='berlin'.
- jeder Befehl wird mit einem Strichpunkt ";" abgeschlossen
- Kommentarzeilen werden in $/* \ldots */$ eingeschlossen, oder mit `--` oder `rem` eingeleitet.

Data Dictionary: Enthält *Metadaten* über die Datenbank.

Datenbanksprache:

DDL: **D**ata **D**efinition **L**anguage zur Definition der Schemata

- Tabellen
- Sichten
- Indexe
- Integritätsbedingungen

DML: **D**ata **M**anipulation **L**anguage zur Verarbeitung von DB–Zuständen

- Suchen
- Einfügen
- Verändern
- Löschen

Data Dictionary

Besteht aus Tabellen und Views, die *Metadaten* über die Datenbank enthalten.

Mit `SELECT * FROM DICTIONARY` (kurz `SELECT * FROM DICT`) erklärt sich das Data Dictionary selber.

TABLE_NAME
COMMENTS
ALL_ARGUMENTS
Arguments in objects accessible to the user
ALL_CATALOG
All tables, views, synonyms, sequences accessible to the user
ALL_CLUSTERS
Description of clusters accessible to the user
ALL_CLUSTER_HASH_EXPRESSIONS
Hash functions for all accessible clusters
⋮

Data Dictionary

`ALL_OBJECTS`: Enthält alle Objekte, die einem Benutzer zugänglich sind.

`ALL_CATALOG`: Enthält alle Tabellen, Views und Synonyme, die einem Benutzer zugänglich sind.

`ALL_TABLES`: Enthält alle Tabellen, die einem Benutzer zugänglich sind.

Analog für diverse andere Dinge (`select * from ALL_CATALOG where TABLE_NAME LIKE 'ALL%';`).

`USER_OBJECTS`: Enthält alle Objekte, die einem Benutzer gehören.

Analog für die anderen, meistens existieren für `USER_...` auch Abkürzungen, etwa `OBJ` für `USER_OBJECTS`.

`ALL_USERS`: Enthält Informationen über alle Benutzer der Datenbank.


```
SELECT table_name FROM tabs;
```

Table_name	Table_name
BORDERS	ISLAND
CITY	LAKE
CONTINENT	LANGUAGE
COUNTRY	LOCATED
DESERT	IS_MEMBER
ECONOMY	MERGES_WITH
ENCOMPASSES	MOUNTAIN
ETHNIC_GROUP	ORGANIZATION
GEO_DESERT	POLITICS
GEO_ISLAND	POPULATION
GEO_LAKE	PROVINCE
GEO_MOUNTAIN	RELIGION
GEO_RIVER	RIVER
GEO_SEA	SEA

28 Zeilen wurden ausgewählt.

Die Definition einzelner Tabellen und Views wird mit `DESCRIBE <table>` oder kurz `DESC <table>` abgefragt:

```
DESC City;
```

Name	NULL?	Typ
NAME	NOT NULL	VARCHAR2(25)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(35)
POPULATION		NUMBER
LONGITUDE		NUMBER
LATITUDE		NUMBER

Anfragen: SELECT-FROM-WHERE

Anfragen an die Datenbank werden in SQL ausschließlich mit dem `SELECT`-Befehl formuliert. Dieser hat prinzipiell eine sehr einfache Grundstruktur:

`SELECT`	Attribute
`FROM`	Relation(en)
`WHERE`	Bedingung

Einfachste Form: alle Spalten und Zeilen einer Relation

```
SELECT * FROM City;
```

Name	C.	Province	Pop.	Long.	Lat.
⋮	⋮	⋮	⋮	⋮	⋮
Vienna	A	Vienna	1583000	16,3667	48,25
Innsbruck	A	Tyrol	118000	11,22	47,17
Stuttgart	D	Baden-W.	588482	9.1	48.7
Freiburg	D	Germany	198496	NULL	NULL
⋮	⋮	⋮	⋮	⋮	⋮

3114 Zeilen wurden ausgewählt.

Projektionen: Auswahl von Spalten

```
SELECT <attr-list>
FROM <table>;
```

Gebe zu jeder Stadt ihren Namen und das Land, in dem sie liegt, aus.

```
SELECT Name, Country
FROM City;
```

Name	COUNTRY
Tokyo	J
Stockholm	S
Warsaw	PL
Cochabamba	BOL
Hamburg	D
Berlin	D
..	..

Praktikum: Datenbankprogrammierung in SQL/ORACLE

DISTINCT

```
SELECT * FROM Island;
```

Name	Islands	Area	...
⋮	⋮	⋮	⋮
Jersey	Channel Islands	NULL	...
Mull	Inner Hebrides	910	...
Montserrat	Antilles	106	...
Grenada	Antilles	NULL	...
⋮	⋮	⋮	⋮

```
SELECT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
Antilles
⋮

```
SELECT DISTINCT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
⋮

Duplikateliminierung

- Duplikateliminierung nicht automatisch:
  - Duplikateliminierung teuer (Sortieren + Eliminieren)
  - Nutzer will Duplikate sehen
  - später: Aggregatfunktionen auf Relationen mit Duplikaten
- Duplikateliminierung: `DISTINCT`-Klausel
- später: Duplikateliminierung automatisch bei Anwendung der Mengenoperatoren `UNION`, `INTERSECT`, ...

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Selektionen: Auswahl von Zeilen

```
SELECT <attr-list>
FROM <table>
WHERE <predicate>;
```

<predicate> kann dabei die folgenden Formen annehmen:

- <attribute> <op> <value> mit $op \in \{=, <, >, <=, >=\}$,
- <attribute> `[NOT] LIKE` <string>, wobei underscores im String ein Zeichen repräsentieren und Prozentzeichen null bis beliebig viele Zeichen darstellen,
- <attribute> `IN` <value-list>, wobei <value-list> entweder von der Form ('val1',...,'valn') ist, oder durch eine Subquery bestimmt wird,
- `[NOT] EXISTS <subquery>`
- `NOT` (<predicate>),
- <predicate> `AND` <predicate>,
- <predicate> `OR` <predicate>.

Beispiel:

```
SELECT Name, Country, Population
FROM City
WHERE Country = 'J';
```

Name	Country	Population
Tokyo	J	7843000
Kyoto	J	1415000
Hiroshima	J	1099000
Yokohama	J	3256000
Sapporo	J	1748000
⋮	⋮	⋮

Beispiel:

```
SELECT Name, Country, Population
FROM City
WHERE Country = 'J' AND Population > 2000000
```

Name	Country	Population
Tokyo	J	7843000
Yokohama	J	3256000

Beispiel:

```
SELECT Name, Country, Population
FROM City
WHERE Country LIKE '%J_%';
```

Name	Country	Population
Kingston	JA	101000
Amman	JOR	777500
Suva	FJI	69481
⋮	⋮	⋮

Die Forderung, daß nach dem J noch ein weiteres Zeichen folgen muß, führt dazu, daß die japanischen Städte nicht aufgeführt werden.

ORDER BY

```
SELECT Name, Country, Population
FROM City
WHERE Population > 5000000
ORDER BY Population DESC; (absteigend)
```

Name	Country	Population
Seoul	ROK	10.229262
Mumbai	IND	9.925891
Karachi	PK	9.863000
Mexico	MEX	9.815795
Sao Paulo	BR	9.811776
Moscow	R	8.717000
⋮	⋮	⋮

Praktikum: Datenbankprogrammierung in SQL/ORACLE

ORDER BY, Alias

```
SELECT Name, Population/Area AS Density
FROM Country
ORDER BY 2 ; (Default: aufsteigend)
```

Name	Density
Western Sahara	,836958647
Mongolia	1,59528243
French Guiana	1,6613956
Namibia	2,03199228
Mauritania	2,26646745
Australia	2,37559768

Aggregatfunktionen

- `COUNT (*| [DISTINCT] <attribute>)`
- `MAX (<attribute>)`
- `MIN (<attribute>)`
- `SUM ([DISTINCT] <attribute>)`
- `AVG ([DISTINCT] <attribute>)`

Beispiel: Ermittle die Zahl der in der DB abgespeicherten Städten.

```
SELECT Count (*)
FROM City;
```

Count(*)
3114

Beispiel: Ermittle die Anzahl der Länder, für die Millionenstädte abgespeichert sind.

```
SELECT Count (DISTINCT Country)
FROM City
WHERE Population > 1000000;
```

Count(DISTINCT(Country))
68

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Aggregatfunktionen

Beispiel: Ermittle die Gesamtsumme aller Einwohner von Städten Österreichs sowie Einwohnerzahl der größten Stadt Österreichs.

```
SELECT SUM(Population), MAX(Population)
FROM City
WHERE Country = 'A';
```

SUM(Population)	MAX(Population)
2434525	1583000

Und was ist, wenn man diese Werte für *jedes* Land haben will??

Gruppierung

`GROUP BY` berechnet für jede Gruppe *eine Zeile*, die Daten enthalten kann, die mit Hilfe der Aggregatfunktionen über mehrere Zeilen berechnet werden.

```
SELECT <expr-list>
FROM <table>
WHERE <predicate>
GROUP BY <attr-list>;
```

gibt für jeden Wert von `<attr-list>` *eine* Zeile aus. Damit darf `<expr-list>` nur

- Konstanten,
- Attribute aus `<attr-list>`,
- Attribute, die für jede solche Gruppe nur *einen* Wert annehmen (etwa `Code`, wenn `<attr-list>` *Country* ist),
- *Aggregatfunktionen*, die dann über alle Tupels in der entsprechenden Gruppe gebildet werden,

enthalten.

Die `WHERE`-Klausel `<predicate>` enthält dabei nur Attribute der Relationen in `<table>` (also *keine* Aggregatfunktionen).

Gruppierung

Beispiel: Gesucht sei für jedes Land die Gesamtzahl der Einwohner, die in den gespeicherten Städten leben.

```
SELECT Country, Sum(Population)
FROM City
GROUP BY Country;
```

Country	SUM(Population)
A	2434525
AFG	892000
AG	36000
AL	475000
AND	15600
⋮	⋮

Bedingungen an Gruppierungen

Die HAVING-Klausel ermöglicht es, Bedingungen an die durch GROUP BY gebildeten Gruppen zu formulieren:

```
SELECT <expr-list>
FROM <table>
WHERE <predicate1>
GROUP BY <attr-list>
HAVING <predicate2>;
```

- WHERE-Klausel: Bedingungen an einzelne Tupel *bevor* gruppiert wird,
- HAVING-Klausel: Bedingungen, nach denen die *Gruppen* zur Ausgabe ausgewählt werden. In der HAVING-Klausel dürfen neben Aggregatfunktionen nur Attribute vorkommen, die *explizit* in der GROUP BY-Klausel aufgeführt wurden.

Bedingungen an Gruppierungen

Beispiel: Gesucht ist für jedes Land die Gesamtzahl der Einwohner, die in den gespeicherten Städten mit mehr als 10000 Einwohnern leben. Es sollen nur solche Länder ausgegeben werden, bei denen diese Summe größer zehn Millionen ist.

```
SELECT Country, SUM(Population)
FROM City
WHERE Population > 10000
GROUP BY Country
HAVING SUM(Population) > 10000000;
```

Country	SUM(Population)
AUS	12153500
BR	77092190
CDN	10791230
CO	18153631
⋮	⋮

Mengenoperationen

SQL-Anfragen können über Mengenoperatoren verbunden werden:

```
<select-clause> <mengen-op> <select-clause>;
```

- `UNION [ALL]`
- `MINUS [ALL]`
- `INTERSECT [ALL]`
- automatische Duplikateliminierung (kann verhindert werden mit ALL)

Beispiel: Gesucht seien diejenigen Städtenamen, die auch als Namen von Ländern in der Datenbank auftauchen.

```
(SELECT Name
 FROM City)
INTERSECT
(SELECT Name
 FROM Country);
```

Name
Armenia
Djibouti
Guatemala
⋮

Join-Anfragen

Eine Möglichkeit, mehrere Relationen in eine Anfrage einzubeziehen, sind *Join*-Anfragen.

```
SELECT <attr-list>
FROM <table-list>
WHERE <predicate>;
```

Prinzipiell kann man sich einen Join als kartesisches Produkt der beteiligten Relationen vorstellen (Theorie: siehe Vorlesung).

- Attributmenge: Vereinigung aller Attribute
- ggf. durch `<table>.<attr>` qualifiziert.
- Join "mit sich selbst" – Aliase.

Beispiel: Alle Länder, die weniger Einwohner als Tokyo haben.

```
SELECT Country.Name, Country.Population
FROM City, Country
WHERE City.Name = 'Tokyo'
AND Country.Population < City.Population;
```

Name	Einwohner
Albania	3249136
Andorra	72766
Liechtenstein	31122
Slovakia	5374362
Slovenia	1951443
⋮	⋮

Equijoin

Beispiel: Es soll für jede politische Organisation festgestellt werden, in welchem Erdteil sie ihren Sitz hat.

encompasses: Country, Continent, Percentage.

Organization: Abbreviation, Name, City, Country, Province.

```
SELECT Continent, Abbreviation
FROM encompasses, Organization
WHERE encompasses.Country = Organization.Country;
```

Name	Organization
America	UN
Europe	UNESCO
Europe	CCC
Europe	EU
America	CACM
Australia/Oceania	ANZUS
⋮	⋮

Verbindung einer Relation mit sich selbst

Beispiel: Ermittle alle Städte, die in anderen Ländern Namensvettern haben.

```
SELECT A.Name, A.Country, B.Country
FROM City A, City B
WHERE A.Name = B.Name
AND A.Country < B.Country;
```

A.Name	A.Country	B.Country
Alexandria	ET	RO
Alexandria	ET	USA
Alexandria	RO	USA
Barcelona	E	YV
Valencia	E	YV
Salamanca	E	MEX
⋮	⋮	⋮

Subqueries

In der WHERE-Klausel können Ergebnisse von Unterabfragen verwendet werden:

```
SELECT <attr-list>
FROM <table>
WHERE <attribute> (<op> [ANY|ALL] | IN) <subquery>;
```

- <subquery> ist eine SELECT-Anfrage (*Subquery*),
- für <op> $\in \{=, <, >, <=, >=\}$ muß <subquery> eine einspaltige Ergebnisrelation liefern,
- für IN <subquery> sind auch mehrspaltige Ergebnisrelationen erlaubt (seit ORACLE 8).
- für <op> ohne ANY oder ALL muß das Ergebnis von <subquery> einzeilig sein.

Unkorrelierte Subquery

- unabhängig von den Werten des in der umgebenden Anfrage verarbeiteten Tupels,
- wird vor der umgebenden Anfrage *einmal* ausgewertet,
- das Ergebnis wird bei der Auswertung der `WHERE-Klausel` der äußeren Anfrage verwendet,
- streng sequentielle Auswertung, daher ist eine Qualifizierung mehrfach vorkommender Attribute *nicht erforderlich*.

Beispiel: Bestimme alle Länder, in denen es eine Stadt namens Victoria gibt:

```
SELECT Name
FROM Country
WHERE Code IN
  (SELECT Country
   FROM City
   WHERE Name = 'Victoria');
```

Country.Name
Canada
Seychelles

Unkorrelierte Subquery mit `IN`

Beispiel: Alle Städte, von denen bekannt ist, daß die an einem Fluß, See oder Meer liegen:

```
SELECT *
FROM CITY
WHERE (Name,Country,Province)
  IN (SELECT City,Country,Province
      FROM located);
```

Name	Country	Province	Population	...
Ajaccio	F	Corse	53500	...
Karlstad	S	Värmland	74669	...
San Diego	USA	California	1171121	...
⋮	⋮	⋮		

Subquery mit ALL

Beispiel: `ALL` ist z.B. dazu geeignet, wenn man alle Länder bestimmen will, die kleiner als alle Staaten sind, die mehr als 10 Millionen Einwohner haben:

```
SELECT Name,Area,Population
FROM Country
WHERE Area < ALL
  (SELECT Area
   FROM Country
   WHERE Population > 10000000);
```

Name	Area	Population
Albania	28750	3249136
Macedonia	25333	2104035
Andorra	450	72766
⋮	⋮	⋮

Korrelierte Subquery

- Subquery ist von Attributwerten des gerade von der umgebenden Anfrage verarbeiteten Tupels abhängig,
- wird *für jedes Tupel der umgebenden Anfrage* einmal ausgewertet,
- Qualifizierung der *importierten* Attribute erforderlich.

Beispiel: Es sollen alle Städte bestimmt werden, in denen mehr als ein Viertel der Bevölkerung des jeweiligen Landes wohnt.

```
SELECT Name, Country
FROM City
WHERE Population * 4 >
  (SELECT Population
   FROM Country
   WHERE Code = City.Country);
```

Name	Country
Copenhagen	DK
Tallinn	EW
Vatican City	V
Reykjavik	IS
Auckland	NZ
⋮	⋮

Der EXISTS-Operator

`EXISTS` bzw. `NOT EXISTS` bilden den Existenzquantor nach.

```
SELECT <attr-list>
FROM <table>
WHERE [NOT] EXISTS
(<select-clause>);
```

Beispiel: Gesucht seien diejenigen Länder, für die Städte mit mehr als einer Million Einwohnern in der Datenbasis abgespeichert sind.

```
SELECT Name
FROM Country
WHERE EXISTS
      ( SELECT *
        FROM City
        WHERE Population > 1000000
        AND City.Country = Country.Code) ;
```

Name
Serbia and Montenegro
France
Spain
Austria
⋮

Umformung EXISTS, Subquery, Join

Äquivalent dazu sind die beiden folgenden Anfragen:

```
SELECT Name
FROM Country
WHERE Code IN
      ( SELECT Country
        FROM City
        WHERE City.Population > 1000000);
```

```
SELECT DISTINCT Country.Name
FROM Country, City
WHERE City.Country = Country.Code
AND City.Population > 1000000;
```

Beispielanfrage

Ein Land, in dem mehr als 10 Prozent der Bevölkerung in Großstädten leben, gilt als stark urbanisiert. Großstädte sind Städte mit mehr als 500000 Einwohnern. Welche Länder der EU sind stark urbanisiert?

```
SELECT Country.Name
FROM Country, City, is_member
WHERE Organization = 'EU'
AND is_member.Country = Country.Code
AND is_member.Type = 'member'
AND City.Population > 500000
AND City.Country = Country.Code
GROUP BY Country.Name, Country.Population
HAVING (SUM(City.Population)/Country.Population) > 0.1;
```

Name
Austria
Denmark
Germany
Ireland
Italy
Netherlands
Spain
United Kingdom

Subqueries in der FROM-Zeile

```
SELECT <attr-list>
FROM <table/subquery-list>
WHERE <condition>;
```

Werte, die auf unterschiedliche Weise aus einer oder mehreren Tabellen gewonnen werden sollen in Beziehung zueinander gestellt werden.

Beispiel: Gesucht ist die Zahl der Menschen, die nicht in den gespeicherten Städten leben.

```
SELECT Population-Urban_Residents
FROM
  (SELECT SUM(Population) AS Population
   FROM Country),
  (SELECT SUM(Population) AS Urban_Residents
   FROM City);
```

Population-Urban_Residents
4620065771

Subqueries in der FROM-Zeile

Dies ist inbesondere geeignet, um geschachtelte Berechnungen mit Aggregatfunktionen durchzuführen:

Beispiel: Berechnen Sie die Anzahl der Menschen, die in der größten Stadt ihres Landes leben.

```
SELECT sum(pop_biggest)
FROM (SELECT country, max(population) as pop_biggest
   FROM City
   GROUP BY country);
```

sum(pop_biggest)
273837106

Schema-Definition

- das Datenbankschema umfaßt alle Informationen über die Struktur der Datenbank,
- Tabellen, Views, Constraints, Indexe, Cluster, Trigger ...
- ORACLE 8: Datentypen, ggf. Methoden
- wird mit Hilfe der DDL (Data Definition Language) manipuliert,
- CREATE, ALTER und DROP von Schemaobjekten,
- Vergabe von Zugriffsrechten: GRANT.

Erzeugen von Tabellen

```
CREATE TABLE <table>
  (<col> <datatype>,
    ⋮
   <col> <datatype>)
```

`CHAR(`n`)`: Zeichenkette fester Länge n.

`VARCHAR2(`n`)`: Zeichenkette variabler Länge $\leq n$.
$||$: Konkatenation von Strings.

`NUMBER`: Zahlen. Auf `NUMBER` sind die üblichen Operatoren $+$, $-$, $*$ und $/$ sowie die Vergleiche $=$, $>$, $>=$, $<=$ und $<$ erlaubt. Außerdem gibt es `BETWEEN x AND y`. Ungleichheit: $! =$, $\wedge =$, $\neg =$ oder $<>$.

`DATE`: Datum und Zeiten: Jahrhundert – Jahr – Monat – Tag – Stunde – Minute – Sekunde. U.a. wird auch Arithmetik für solche Daten angeboten.

weitere Datentypen findet man im Manual.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Tabellendefinition

Das folgende SQL-Statement erzeugt z.B. die Relation *City* (noch ohne Integritätsbedingungen):

```
CREATE TABLE City
      ( Name         VARCHAR2(35),
        Country      VARCHAR2(4),
        Province     VARCHAR2(32),
        Population   NUMBER,
        Longitude    NUMBER,
        Latitude     NUMBER );
```

Tabellendefinition: Constraints

Mit den Tabellendefinitionen können Eigenschaften und Bedingungen an die jeweiligen Attributwerte formuliert werden.

- Bedingungen an ein einzelnes oder mehrere Attribute:
- Wertebereichseinschränkungen,
- Angabe von Default-Werten,
- Forderung, daß ein Wert angegeben werden muß,
- Angabe von Schlüsselbedingungen,
- Prädikate an Tupel.

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

- `<colConstraint>` betrifft nur *eine* Spalte,
- `<tableConstraint>` kann mehrere Spalten betreffen.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Tabellendefinition: Default-Werte

```
DEFAULT <value>
```

Ein Mitgliedsland einer Organisation wird als volles Mitglied angenommen, wenn nichts anderes bekannt ist:

```
CREATE TABLE is_member
      ( Country       VARCHAR2(4),
        Organization  VARCHAR2(12),
        Type          VARCHAR2(30)
                      DEFAULT 'member')

INSERT INTO is_member VALUES
    ('CZ', 'EU', 'membership applicant');
INSERT INTO is_member (Land, Organization)
    VALUES ('D', 'EU');
```

Country	Organization	Type
CZ	EU	membership applicant
D	EU	member
⋮	⋮	⋮

Tabellendefinition: Constraints

Zwei Arten von Bedingungen:

- Eine Spaltenbedingung `<colConstraint>` ist eine Bedingung, die nur *eine* Spalte betrifft (zu der sie definiert wird)
- Eine Tabellenbedingung `<tableConstraint>` kann mehrere Spalten betreffen.

Jedes `<colConstraint>` bzw. `<tableConstraint>` ist von der Form

```
[CONSTRAINT <name>] <bedingung>
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Tabellendefinition: Bedingungen (Überblick)

Syntax:

```
[CONSTRAINT <name>] <bedingung>
```

Schlüsselwörter in `<bedingung>`:

1. `CHECK (<condition>)`: Keine Zeile darf `<condition>` verletzen. `NULL`-Werte ergeben dabei ggf. ein *unknown*, also *keine Bedingungsverletzung*.
2. `[NOT] NULL`: Gibt an, ob die entsprechende Spalte Nullwerte enthalten darf (nur als `<colConstraint>`).
3. `UNIQUE (<column-list>)`: Fordert, daß jeder Wert nur einmal auftreten darf.
4. `PRIMARY KEY (<column-list>)`: Deklariert die angegebenen Spalten als Primärschlüssel der Tabelle.
5. `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE]`: gibt an, daß eine Menge von Attributen Fremdschlüssel ist.

Tabellendefinition: Syntax

```
[CONSTRAINT <name>] <bedingung>
```

Dabei ist `CONSTRAINT <name>` optional (ggf. Zuordnung eines systeminternen Namens).

- `<name>` wird bei `NULL`-, `UNIQUE`-, `CHECK`- und `REFERENCES`-Constraints benötigt, wenn das Constraint irgendwann einmal geändert oder gelöscht werden soll,
- `PRIMARY KEY` kann man ohne Namensnennung löschen und ändern.

Da bei einem `<colConstraint>` die Spalte implizit bekannt ist, fällt der `(<column-list>)` Teil weg.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Tabellendefinition: `CHECK` Constraints

- als Spaltenconstraints: Wertebereichseinschränkung

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   ...);
```

- Als Tabellenconstraints: beliebig komplizierte Integritätsbedingungen an ein Tupel.

Tabellendefinition: PRIMARY KEY, UNIQUE und NULL

- PRIMARY KEY (<column-list>): Deklariert diese Spalten als Primärschlüssel der Tabelle.
- Damit entspricht PRIMARY KEY der Kombination aus UNIQUE und NOT NULL.
- UNIQUE wird von NULL-Werten *nicht* unbedingt verletzt, während PRIMARY KEY NULL-Werte *verbietet*.

Eins	Zwei
a	b
a	NULL
NULL	b
NULL	NULL

erfüllt UNIQUE (Eins,Zwei).

- Da auf jeder Tabelle nur ein PRIMARY KEY definiert werden darf, wird NOT NULL und UNIQUE für Candidate Keys eingesetzt.

Relation *Country*: Code ist PRIMARY KEY, Name ist Candidate Key:

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

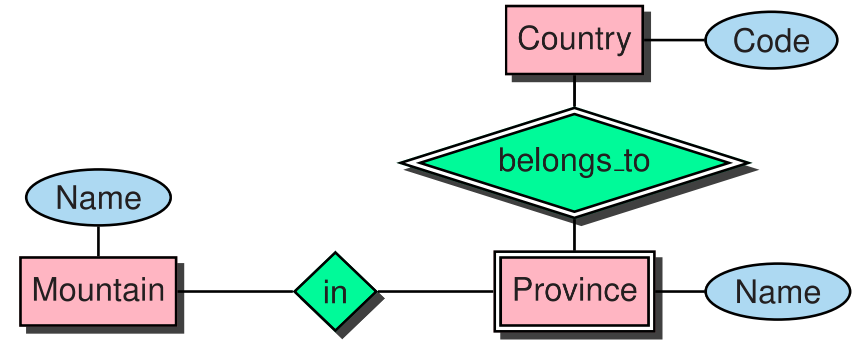

Tabellendefinition: FOREIGN KEY ...REFERENCES

- FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE]: gibt an, daß das Attributtupel <column-list> der Tabelle ein Fremdschlüssel ist und das Attributtupel <column-list2> der Tabelle <table> referenziert.
- Das referenzierte Attributtupel <table>(<column-list2>) muß in der Definition von <table> als PRIMARY KEY deklariert sein.
- Eine REFERENCES-Bedingung wird durch NULL-Werte nicht verletzt.
- ON DELETE CASCADE: Referentielle Aktion (später).

```
CREATE TABLE is_member
  (Country        VARCHAR2(4)
        REFERENCES Country(Code),
   Organization   VARCHAR2(12)
        REFERENCES Organization(Abbreviation),
   Type           VARCHAR2(30) DEFAULT 'member');
```

Tabellendefinition: Fremdschlüssel

Ein Berg liegt in einer Provinz eines Landes:




```
CREATE TABLE geo_Mountain
  ( Mountain VARCHAR2(20)
                REFERENCES Mountain(Name),
    Country VARCHAR2(4) ,
    Province VARCHAR2(32) ,
    CONSTRAINT GMountRefsProv
                FOREIGN KEY (Country,Province)
                REFERENCES Province (Country,Name));
```


Tabellendefinition

Vollständige Definition der Relation *City* mit Bedingungen und Schlüsseldeklaration:

```
CREATE TABLE City
 ( Name VARCHAR2(35),
   Country VARCHAR2(4)
     REFERENCES Country(Code),
   Province VARCHAR2(32)     -- + <tableConstraint>
   Population NUMBER CONSTRAINT CityPop
     CHECK (Population >= 0),
   Longitude NUMBER CONSTRAINT CityLong
     CHECK ((Longitude >= -180) AND (Longitude <= 180)),
   Latitude NUMBER CONSTRAINT CityLat
     CHECK ((Latitude >= -90) AND (Latitude <= 90)),
   CONSTRAINT CityKey
     PRIMARY KEY (Name, Country, Province),
   FOREIGN KEY (Country,Province)
     REFERENCES Province (Country,Name));
```

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES` `<table>(<column-list>)`–Klausel enthält, erstellt wird, muß `<table>` bereits definiert und `<column-list>` dort als `PRIMARY KEY` deklariert sein.

Views (=Sichten)

- Virtuelle Tabellen
- nicht zum Zeitpunkt ihrer Definition berechnet, sondern
- jedesmal berechnet, wenn auf sie zugegriffen wird.
- spiegeln also stets den aktuellen Zustand der ihnen zugrundeliegenden Relationen wieder.
- Änderungsoperationen nur in eingeschränktem Umfang möglich.

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
<select-clause>;
```

Beispiel: Benutzer benötigt häufig die Information, welche Stadt in welchem Land liegt, ist jedoch weder an Landeskürzeln noch Einwohnerzahlen interessiert.

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

Wenn der Benutzer nun nach allen Städten in Kamerun sucht, so kann er die folgende Anfrage stellen:

```
SELECT *
FROM CityCountry
WHERE Country = 'Cameroon';
```


Löschen von Tabellen und Views

- Tabellen bzw. Views werden mit `DROP TABLE` bzw. `DROP VIEW` gelöscht:

  ```
  DROP TABLE <table-name> [CASCADE CONSTRAINTS];
  DROP VIEW <view-name>;
  ```

- Tabellen müssen nicht leer sein, wenn sie gelöscht werden sollen.
- Es ist nicht möglich, eine Tabelle zu löschen, die referenzierte Tupel enthält.
- eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, kann mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden.
- Mit `DROP TABLE <table> CASCADE CONSTRAINTS` wird eine Tabelle mit allen auf sie zeigenden referentiellen Integritätsbedingungen gelöscht.

Ändern von Tabellen und Views

später.

Einfügen von Daten

- `INSERT`-Statement.
- Daten einzeln von Hand einfügen,

  ```
  INSERT INTO <table>[(<column-list>)]
  VALUES (<value-list>);
  ```

  oder
- Ergebnis einer Anfrage.

  ```
  INSERT INTO <table>[(<column-list>)]
  <subquery>;
  ```

- Rest wird ggf. mit Nullwerten aufgefüllt.

So kann man z.B. das folgende Tupel einfügen:

```
INSERT INTO Country (Name, Code, Population)
VALUES ('Lummerland', 'LU', 4);
```

Eine Tabelle *Metropolis (Name, Country, Population)* kann man z.B. mit dem folgenden Statement füllen:

```
INSERT INTO Metropolis
SELECT Name, Country, Population
FROM City
WHERE Population > 1000000;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Löschen von Daten

Tupel können mit Hilfe der `DELETE`-Klausel aus Relationen gelöscht werden:

```
DELETE FROM <table>
WHERE <predicate>;
```

Dabei gilt für die `WHERE`-Klausel das für `SELECT` gesagte.

Mit einer leeren `WHERE`-Bedingung kann man z.B. eine ganze Tabelle abräumen (die Tabelle bleibt bestehen, sie kann mit `DROP TABLE` entfernt werden):

```
DELETE FROM City;
```

Der folgende Befehl löscht sämtliche Städte, deren Einwohnerzahl kleiner als 50.000 ist.

```
DELETE FROM City
WHERE Population < 50000;
```

Ändern von Tupeln

```
UPDATE <table>
SET <attribute> = <value> | (<subquery>),
                         ⋮
    <attribute> = <value> | (<subquery>),
    (<attribute-list>) = (<subquery>),
                         ⋮
    (<attribute-list>) = (<subquery>)
WHERE <predicate>;
```

Beispiel:

```
UPDATE City
SET Name = 'Leningrad',
    Population = Population + 1000,
WHERE Name = 'Sankt-Peterburg';
```

Beispiel: Die Einwohnerzahl jedes Landes wird als die Summe der Einwohnerzahlen aller Provinzen gesetzt:

```
UPDATE Country
SET Population = (SELECT SUM(Population)
                  FROM Province
                  WHERE Province.Country=Country.Code);
```


Zeitangaben

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

- Eingabe-Format mit `NLS_DATE_FORMAT` setzen,
- Default: `'DD-MON-YY'` eingestellt, d.h. z.B. `'20-Oct-97'`.

```
CREATE TABLE Politics
  ( Country VARCHAR2(4),
    Independence DATE,
    Government VARCHAR2(120));

ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';

INSERT INTO politics VALUES
  ('B','04 10 1830','constitutional monarchy');
```

Alle Länder, die zwischen 1200 und 1600 gegründet wurden:

```
SELECT Country, Independence
FROM Politics
WHERE Independence BETWEEN
'01 01 1200' AND '31 12 1599';
```

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

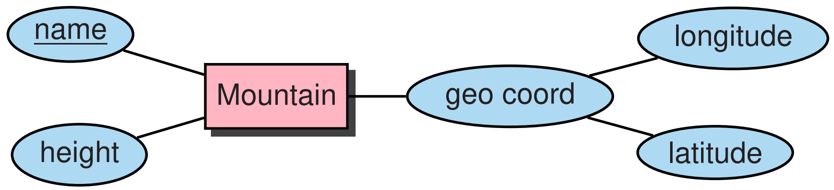
Zeitangaben

ORACLE bietet einige Funktionen um mit dem Datentyp `DATE` zu arbeiten:

- `SYSDATE` liefert das aktuelle Datum.
- Addition und Subtraktion von Absolutwerten auf `DATE` ist erlaubt, Zahlen werden als Tage interpretiert: `SYSDATE` $+ 1$ ist morgen, `SYSDATE` $+ (10/1440)$ ist “in zehn Minuten”.
- `ADD_MONTHS`(d, n) addiert n Monate zu einem Datum d.
- `LAST_DAY`(d) ergibt den letzten Tag des in d angegebenen Monats.
- `MONTHS_BETWEEN`(d_1, d_2) gibt an, wieviele Monate zwischen 2 Daten liegen.

Objektorientierung in ORACLE 8

- Komplexe Datentypen:



- Geschachtelte Tabellen:

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

- Objekte, Methoden, Objekttabellen, Objektreferenzen ... (später)

Erzeugung von Datentypen

Neue Klasse von Schemaobjekten: `CREATE TYPE`

- ```
  CREATE [OR REPLACE] TYPE <name> AS OBJECT
   (<attr> <datatype>,
    ⋮
    <attr> <datatype>);
  ```
  Bei "echten" Objekten kommt noch ein `CREATE TYPE BODY ...` dazu, in dem die Methoden in PL/SQL definiert werden ... später.

  Ohne Body bekommt man einfache komplexe Datentypen (ähnlich wie *Records*).

- ```
  CREATE [OR REPLACE] TYPE <name>
    AS TABLE OF <datatype>
  ```
  ("Kollektion", Tabellen als *Datentypen*)

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Komplexe Datentypen

Geographische Koordinaten:

```
CREATE TYPE GeoCoord AS OBJECT
  ( Longitude NUMBER,
    Latitude  NUMBER);
/

CREATE TABLE Mountain
      ( Name          VARCHAR2(20),
        Height        NUMBER,
        Coordinates   GeoCoord);
```

`CREATE TYPE <type> AS OBJECT (...)` definiert automatisch eine *Konstruktormethode* `<type>`:

```
INSERT INTO Mountain
  VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

```
SELECT * FROM Mountain;
```

Name	Height	Coordinates(Longitude, Latitude)
Feldberg	1493	GeoCoord(8,48)

Komplexe Datentypen

Zugriff auf einzelne Komponenten von komplexen Attributen in der bei Records üblichen *dot*-Notation.
ORACLE 8.0: *nur mit Qualifizierung*:

```
SELECT Name, B.Coordinates.Longitude,
       B.Coordinates.Latitude
FROM Mountain B;
```

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	8	48

Geschachtelte Tabellen

```
CREATE [OR REPLACE] TYPE <inner_type>
  AS OBJECT (...);
/
CREATE [OR REPLACE] TYPE <inner_table_type> AS
  TABLE OF <inner_type>;
/
CREATE TABLE <table_name>
  (... ,
   <table-attr> <inner_table_type> ,
   ... )
NESTED TABLE <table-attr> STORE AS <name >;
```

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

Geschachtelte Tabellen

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

Wieder: Konstruktormethoden

```
INSERT INTO NLanguage
VALUES( 'SK',
        Languages_list
          (  Language_T('Slovak',95),
             Language_T('Hungarian',5)));
```


Geschachtelte Tabellen

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

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

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

Languages(Name, Percentage)
Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Anfragen an Geschachtelte Tabellen

Inhalt von inneren Tabellen:

```
THE (SELECT <table-attr> FROM ...)
```

```
SELECT ...
FROM THE (<select-statement>)
WHERE ... ;
```

```
INSERT INTO THE (<select-statement>)
VALUES ... / SELECT ... ;
```

```
DELETE FROM THE (<select-statement>)
WHERE ... ;
```

```
SELECT Name, Percentage
FROM THE( SELECT Languages
          FROM NLanguage
          WHERE Country='CH');
```

Name	Percentage
German	65
French	18
Italian	12
Romansch	1

Kopieren von Geschachtelten Tabellen

Geschachtelte Tabelle “am Stück” einfügen: Menge von Tupeln wird als Kollektion strukturiert:

```
CAST(MULTISET(SELECT ...) AS <nested-table-type>)
```

```
INSERT INTO NLanguage -- zulaessig, aber falsch !!!!
 (SELECT Country,
         CAST(MULTISET(SELECT Name, Percentage
                       FROM Language
                       WHERE Country = A.Country)
              AS Languages_List)
 FROM Language A);
```

jedes Tupel (Land, Sprachenliste) n-mal
(n = Anzahl Sprachen in diesem Land) !!

```
INSERT INTO NLanguage (Country)
 (SELECT Country
  FROM Language);
```

```
UPDATE NLanguage B
SET Languages =
  CAST(MULTISET(SELECT Name, Percentage
                FROM Language A
                WHERE B.Country = A.Country)
       AS Languages_List);
```

Geschachtelte Tabellen

Liefert eine Anfrage bereits eine Tabelle, kann man diese als ganzes einfügen:

```
INSERT INTO <table>
VALUES (...,  THE ( SELECT <attr>
                    FROM <table'>
                    WHERE ...)      );
```

```
INSERT INTO NLanguage VALUES
 ('CHXX', THE (SELECT Languages from NLanguage
                WHERE Country='CH'));
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Arbeiten mit geschachtelten Tabellen

Nicht ganz einfach ... (ORACLE 8.0)

- Unterabfrage darf nur *eine einzige* geschachtelte Tabelle zurückgeben.
  ⇒ nicht möglich in Abhängigkeit von dem Tupel der äußeren Tabelle die jeweils passende innere Tabelle auszuwählen:

Alle Länder, in denen Deutsch gesprochen wird:

```
SELECT Country -- UNZULAESSIG !!!!
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE Name='German');
```

Arbeiten mit geschachtelten Tabellen

```
TABLE ([<table>.]<attr>)
```

kann in *Unterabfrage* verwendet werden:

```
SELECT Country
FROM NLanguage
WHERE EXISTS
  (SELECT *
   FROM TABLE (Languages) -- zu dem aktuellen Tupel
   WHERE Name='German');
```

Country
A
B
CH
D
NAM

Aber: Attribute der inneren Tabelle können nicht im äußeren `SELECT`-Statement ausgewählt werden.

⇒ Ausgabe des prozentualen Anteils in den verschiedenen Ländern nicht möglich.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Arbeiten mit geschachtelten Tabellen

`CURSOR`-Operator:

Beispiel:

```
SELECT Country,
       CURSOR (SELECT *
               FROM TABLE (Languages))
FROM NLanguage;
```

Country	**CURSOR(SELECT...)**
CH	CURSOR STATEMENT : 2
NAME	**PERCENTAGE**
French	18
German	65
Italian	12
Romansch	1

⇒ Cursore etc. in PL/SQL.

Arbeiten mit geschachtelten Tabellen

```
SELECT Country, Name -- UNZULAESSIG !!
FROM NLanguage A,
    THE ( SELECT Languages
          FROM NLanguage B
          WHERE B.Country=A.Country);
```

```
SELECT Country, Name
FROM NLanguage A,
    THE ( SELECT Languages
          FROM NLanguage B
          WHERE B.Country=A.Country)
WHERE A.Country = 'CH'; -- jetzt zulaessig.
```

Mit Tabelle *All_Languages*, die alle Sprachen enthält:

```
SELECT Country, Name
FROM NLanguage, All_Languages
WHERE Name IN
  (SELECT Name
   FROM TABLE (Languages));
```

Fazit: Wertebereich von geschachtelten Tabellen muß in *einer* Tabelle zugreifbar sein.

Komplexe Datentypen

```
SELECT * FROM USER_TYPES
```

Type_name	Type_oid	Typecode	Attributes	Methods	Pre	Inc
GeoCoord	_	Object	2	0	NO	NO
Language_T	_	Object	2	0	NO	NO
Languages_List	_	Collection	0	0	NO	NO

Löschen: `DROP TYPE [FORCE]`

Mit `FORCE` kann ein Typ gelöscht werden, dessen Definition von anderen Typen noch gebraucht wird.

Szenario von oben:

`DROP TYPE Language_T`

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

`DROP TYPE Language_T FORCE` löscht `Language_T`, allerdings

```
SQL> desc Languages_List;
FEHLER:
ORA-24372: Ungültiges Objekt für Beschreibung
```

Transaktionen in ORACLE

Beginn einer Transaktion

```
SET TRANSACTION READ [ONLY | WRITE];
```

Sicherungspunkte setzen

Für eine längere Transaktion können zwischendurch Sicherungspunkte gesetzt werden:

```
SAVEPOINT <savepoint>;
```

Ende einer Transaktion

- `COMMIT`-Anweisung, macht alle Änderungen persistent,
- `ROLLBACK [TO <savepoint>]` nimmt alle Änderungen [bis zu `<savepoint>`] zurück,
- DDL-Anweisung (z.B. `CREATE`, `DROP`, `RENAME`, `ALTER`),
- Benutzer meldet sich von ORACLE ab,
- Abbruch eines Benutzerprozesses.

Referentielle Integrität – A First Look

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`–Klausel enthält, erstellt wird, muß `<table>` bereits definiert und `<column-list>` dort als `PRIMARY KEY` deklariert sein.
- Beim Einfügen von Daten müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.
- Beim Löschen eines referenzierten Tupels muß die referentielle Integrität erhalten bleiben.
- Tabellen bzw. Views werden mit `DROP TABLE` bzw. `DROP VIEW` gelöscht.
- Es ist nicht möglich, eine Tabelle zu löschen, die referenzierte Tupel enthält.
- eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, wird mit `DROP TABLE <table> CASCADE CONSTRAINTS` gelöscht.
- Geschachtelte Tabellen unterstützen Referentielle Integrität nicht.

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>;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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

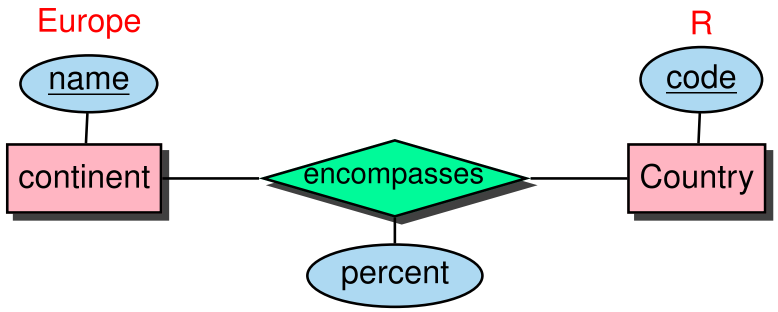
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

```
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';`

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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 NULLs 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)

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Referentielle Aktionen im SQL-2-Standard

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

<columnConstraint> (für einzelne Spalten)

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

oder <tableConstraint> (für mehrere Spalten)

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

angegeben.

Referentielle Aktionen

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

CASCADE

NO ACTION

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

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

Referentielle Aktionen in ORACLE:

- ORACLE 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

Country

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

Province

Name	Country	Capital
Berlin	D	Berlin
Distr.Col.	US	Washington
...	...	...

City

Name	Country	Province
Berlin	D	B
Washington	USA	Distr.Col.
...	...	...

Referentielle Aktionen: Problematik `ON UPDATE`

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

CASCADE

SET NULL

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

CASCADE

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

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


Referentielle Aktionen

Im allgemeinen Fall:

- Schon 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
SET Country = 'Poland'
WHERE Name = 'Berlin';
```

```
UPDATE CityCountry
SET Country = 'Deutschland'
WHERE Country = 'Germany';
```

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

```
DELETE FROM CityCountry
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

⇒ *Materialized Views*

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


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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 ];
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

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:

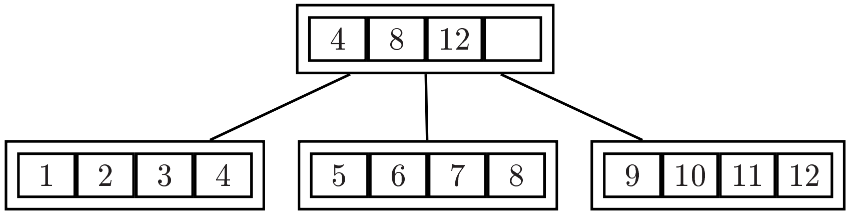
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

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)

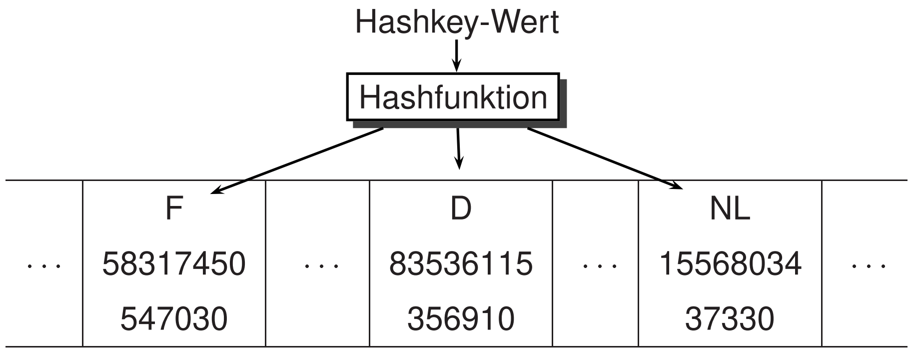
CREATE INDEX PLZIndex ON PLZ (Country,PLZ);

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	...
⋮	⋮	⋮	⋮	⋮

Praktikum: Datenbankprogrammierung in SQL/ORACLE

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;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

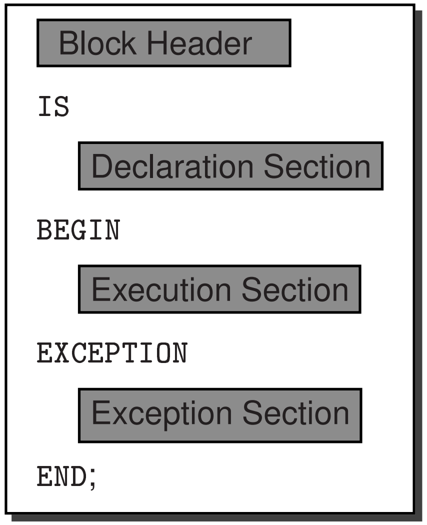
Prozedurale Erweiterungen: PL/SQL

- keine prozeduralen Konzepte in SQL (Schleifen, Verzweigungen, Variablendeklarationen)
- viele Aufgaben nur umständlich über Zwischentabellen oder überhaupt nicht in SQL zu realisieren.
  - Transitive Hülle.
- Programme repräsentieren anwendungsspezifisches Wissen, das nicht in der Datenbank enthalten ist.

Erweiterungen

- Einbettung von SQL in prozedurale Wirtssprachen (*embedded SQL*); meistens Pascal, C, C++, oder neuerdings auch Java (JDBC),
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern.
- benötigt für Objektmethoden.

Blockstruktur von PL/SQL



- Block Header: Art des Objekts (Funktion, Prozedur oder *anonym* (innerhalb eines anderen Blocks)), und Parameterdeklarationen.
- Declaration Section: Deklarationen der in dem Block verwendeten Variablen,
- Execution Section: Befehlssequenz des Blocks,
- Exception Section: Reaktionen auf eventuell auftretende Fehlermeldungen.

Prozeduren

```
CREATE [OR REPLACE] PROCEDURE <proc_name>
  [(<parameter-list>)]
  IS <pl/sql-body>;
/
```

- OR REPLACE: existierende Prozedurdefinition wird überschrieben.
- (<parameter-list>): Deklaration der formalen Parameter:

```
  (<variable> [IN|OUT|IN OUT] <datatype>,
    ⋮
   <variable> [IN|OUT|IN OUT] <datatype>)
```

- IN, OUT, IN OUT: geben an, wie die Prozedur/Funktion auf den Parameter zugreifen kann (Lesen, Schreiben, beides).
- Default: IN.
- Bei OUT und IN OUT muß beim Aufruf eine Variable angegeben sein, bei IN ist auch eine Konstante erlaubt.
- <datatype>: alle von PL/SQL unterstützten Datentypen; *ohne* Längenangabe (VARCHAR2 anstelle VARCHAR2(20)).
- <pl/sql-body> enthält die Definition der Prozedur in PL/SQL.

Funktionen

Analog, zusätzlich wird der Datentyp des Ergebnisses angegeben:

```
CREATE [OR REPLACE] FUNCTION <funct_name>
  [(<parameter-list>)]
  RETURN <datatype>
  IS <pl/sql body>;
/
```

- PL/SQL-Funktionen werden mit

  ```
  RETURN <ausdruck>;
  ```

  verlassen. Jede Funktion muß mindestens ein `RETURN`-Statement im `<body>` enthalten.
- Eine Funktion darf keine Seiteneffekte auf die Datenbasis haben.

Wichtig: nach dem Semikolon noch ein Vorwärtsslash (“/”), um die Deklaration auszuführen!!!

Im Falle von “... created with compilation errors”:

```
SHOW ERRORS;
```

ausgeben lassen.

Prozeduren und Funktionen können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Prozeduren und Funktionen

- Aufruf von Prozeduren im PL/SQL-Skript:
  `<procedure> (arg1,...,argn);`
  (wenn ein formaler Parameter als OUT oder INOUT angegeben ist, muss das Argument eine Variable sein)
- Aufruf in von Prozeduren in SQLPlus:
  `execute <procedure> (arg1,...,argn);`
- Verwendung von Funktionen in PL/SQL:
  `... <function> (arg1,...,argn) ...`
  wie in anderen Programmiersprachen.

Die system-eigene Tabelle `DUAL` wird verwendet um das Ergebnis freier Funktionen ausgeben zu lassen.

```
SELECT <function> (arg1,...,argn)
FROM DUAL;
```

Beispiel: Prozedur

- Einfache Prozedur: PL/SQL-Body enthält nur SQL-Befehle

Informationen über Länder sind über mehrere Relationen verteilt.

```
CREATE OR REPLACE PROCEDURE InsertCountry
 (name VARCHAR2, code VARCHAR2, area NUMBER, pop NUMBER,
  gdp NUMBER, inflation NUMBER, pop_growth NUMBER)
IS
BEGIN
  INSERT INTO Country (Name,Code,Area,Population)
         VALUES (name,code,area,pop);
  INSERT INTO Economy (Country,GDP,Inflation)
         VALUES (code,gdp,inflation);
  INSERT INTO Population (Country,Population_Growth)
         VALUES (code,pop_growth);
END;
/
EXECUTE InsertCountry
   ('Lummerland', 'LU', 1, 4, 50, 0.5, 0.25);
```


Beispiel: Funktion

- Einfache Funktion: Einwohnerdichte eines Landes

```
CREATE OR REPLACE FUNCTION Density
 (arg VARCHAR2)
RETURN number
IS
  temp number;
BEGIN
  SELECT Population/Area
    INTO temp
    FROM Country
    WHERE code = arg;
  RETURN temp;
END;
/
SELECT Density('D')
FROM dual;
```

PL/SQL-Variablen und Datentypen.

Deklaration der PL/SQL-Variablen in der Declaration Section:

```
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
⋮
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
```

Einfache Datentypen:

BOOLEAN: TRUE, FALSE, NULL,

BINARY_INTEGER, PLS_INTEGER: Ganzzahlen mit Vorzeichen.

NATURAL, INT, SMALLINT, REAL, ...: Numerische Datentypen.

```
anzahl NUMBER DEFAULT 0;
name VARCHAR2(30);
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

anchored Typdeklaration

Angabe einer PL/SQL-Variablen, oder Tabellenspalte (!) deren Typ man übernehmen will:

```
<variable> <variable'>%TYPE
  [NOT NULL] [DEFAULT <value>];
oder
<variable> <table>.<col>%TYPE
  [NOT NULL] [DEFAULT <value>];
```

- cityname City.Name%TYPE
- %TYPE wird zur Compile-Time bestimmt.

Zuweisung an Variablen

- “klassisch” innerhalb des Programms:

  ```
  a := b;
  ```

- Zuweisung des (einspaltigen und einzeiligen!) Ergebnisses einer Datenbankanfrage an eine PL/SQL-Variable:

  ```
  SELECT ...
  INTO <PL/SQL-Variable>
  FROM ...
  ```

Beispiel:

```
the_name country.name%TYPE
         ⋮
SELECT name
INTO the_name
FROM country
WHERE name='Germany';
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

PL/SQL-Datentypen: Records

Ein `RECORD` enthält mehrere Felder, entspricht einem Tupel in der Datenbasis:

```
TYPE city_type IS RECORD
  (Name City.Name%TYPE,
   Country VARCHAR2(4),
   Province VARCHAR2(32),
   Population NUMBER,
   Longitude NUMBER,
   Latitude NUMBER);

the_city city_type;
```

anchored Typdeklaration für Records

Records mit Tabellenzeilen-Typ deklarieren: `%ROWTYPE`:

```
<variable> <table-name>%ROWTYPE;
```

Äquivalent zu oben:

```
the_city city%ROWTYPE;
```

Zuweisung an Records

- Aggregierte Zuweisung: zwei Variablen desselben Record-Typs:

```
<variable> := <variable'>;
```

- Feldzuweisung: ein Feld wird einzeln zugewiesen:

```
<record.feld> := <variable>|<value>;
```

- `SELECT INTO`: Ergebnis einer Anfrage, die *nur ein einziges Tupel* liefert:

```
SELECT ...
INTO <record-variable>
FROM ... ;
```

```
the_country country%ROWTYPE
        ⋮
SELECT *
INTO the_country
FROM country
WHERE name='Germany';
```

Vergleich von Records:
Beim Vergleich von Records muß jedes Feld einzeln verglichen werden.

PL/SQL-Datentypen: PL/SQL Tables

Array-artige Struktur, *eine* Spalte mit beliebigem Datentyp (also auch `RECORD`), normalerweise mit `BINARY_INTEGER` indiziert.

```
TYPE <type> IS TABLE OF <datatype>
  [INDEX BY BINARY_INTEGER];
```

```
<var> <type>;
```

```
plz_table_type IS TABLE OF City.Name%TYPE
  INDEX BY BINARY_INTEGER;
```

```
plz_table plz_table_type;
```

- Adressierung: `<var>(1)`

```
plz_table(79110):= Freiburg;
plz_table(33334):= Kassel;
```

- *sparse*: nur die Zeilen gespeichert, die Werte enthalten.

Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

PL/SQL-Datentypen: PL/SQL Tables

Zusätzlich *built-in*-Funktionen und -Prozeduren:

```
<variable> := <pl/sql-table-name>.<built-in-function>;
```

oder

```
<pl/sql-table-name>.<built-in-procedure>;
```

- `COUNT` (fkt): Anzahl der belegten Zeilen.
  plz_table.count = 2
- `EXISTS` (fkt): `TRUE` falls Tabelle nicht leer.
- `DELETE` (proc): Löscht alle Zeilen einer Tabelle.
- `FIRST/LAST` (fkt): niedrigster/höchster belegter Indexwert.
  plz_table.first = 33334
- `NEXT/PRIOR(n)` (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen belegten Indexwert.
  plz_table.next(33334) = 79110

SQL-Statements in PL/SQL

- DML-Kommandos `INSERT`, `UPDATE`, `DELETE` sowie `SELECT INTO`-Statements.
- Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten.
- Befehle, die *nur ein einziges Tupel betreffen* können mit `RETURNING` Werte an PL/SQL-Variablen zurückgeben:

```
UPDATE ... SET ... WHERE ...
RETURNING <expr-list>
INTO <variable-list>;
```

Z.B. Row-ID des betroffenen Tupels zurückgeben:

```
DECLARE rowid ROWID;
BEGIN
   ⋮
   INSERT INTO Politics (Country,Independence)
     VALUES (Code,SYSDATE)
     RETURNING ROWID
     INTO rowid;
   ⋮
END;
```

- DDL-Statements in PL/SQL nicht direkt unterstützt: DBMS_SQL-Package.

Kontrollstrukturen

- `IF THEN - [ELSIF THEN] - [ELSE] - END IF`,
- verschiedene Schleifen:
- Simple `LOOP`: `LOOP ... END LOOP;`
- `WHILE LOOP`:
  `WHILE <bedingung> LOOP ... END LOOP;`
- Numeric `FOR LOOP`:
  ```
  FOR <loop_index> IN
    [REVERSE] <Anfang> .. <Ende>
   LOOP ... END LOOP;
  ```
  Die Variable `<loop_index>` wird dabei *automatisch* als `INTEGER` deklariert.
- `EXIT [WHEN <bedingung>]`: `LOOP` verlassen.
- den allseits beliebten `GOTO`-Befehl mit Labels:
  `<<label_i>> ... GOTO label_j;`
- `NULL`-Werte verzweigen immer in den `ELSE`-Zweig.
- `GOTO`: nicht von außen in ein `IF`-Konstrukt, einen `LOOP`, oder einen lokalen Block hineinspringen, nicht von einem `IF`-Zweig in einen anderen springen.
- hinter einem Label muß immer mindestens ein ausführbares Statement stehen;
- `NULL` Statement.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Geschachtelte Blöcke

Innerhalb der *Execution Section* werden *anonyme Blöcke* zur Strukturierung verwendet. Hier wird die *Declaration Section* mit `DECLARE` eingeleitet (es gibt keinen Block Header):

```
BEGIN
  -- Befehle des äußeren Blocks --
  DECLARE
   -- Deklarationen des inneren Blocks
  BEGIN
   -- Befehle des inneren Blocks
  END;
  -- Befehle des äußeren Blocks --
END;
```

Cursorbasierter Datenbankzugriff

Zeilenweiser Zugriff auf eine Relation aus einem PL/SQL-Programm.

Cursordeklaration in der *Declaration Section*:

```
CURSOR <cursor-name> [(<parameter-list>)]
IS
  <select-statement>;
```

- `(<parameter-list>)`: Parameter-Liste,
- nur `IN` als Übergaberichtung erlaubt.
- Zwischen `SELECT` und `FROM` auch PL/SQL-Variablen und PL/SQL-Funktionen. PL/SQL-Variablen können ebenfalls in den `WHERE`-, `GROUP`- und `HAVING`-Klauseln verwendet werden.

Beispiel

Alle Städte in dem in der Variablen `the_country` angegebenen Land:

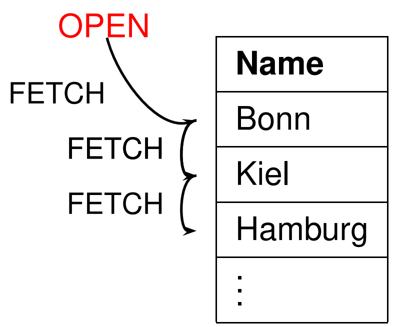
```
DECLARE CURSOR cities_in
   (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```


Cursore

- `OPEN <cursor-name>[(<argument-list>)];`

  Erzeugt mit dem gegebenen `SELECT`-Statement eine *virtuelle Tabelle* mit einem "Fenster", das über einem Tupel stehen kann und schrittweise vorwärts bewegt wird. Mit `OPEN` wird die Anfrage ausgeführt und der Cursor initialisiert:

  ```
  OPEN cities_in ('D');
  ```



Cursore

- `FETCH <cursor-name> INTO <record-variable>;` oder
  `FETCH <cursor-name> INTO <variable-list>;`
  bewegt den Cursor auf die nächste Zeile des Ergebnisses der Anfrage und kopiert diese in die angegebene Record-Variable oder Variablenliste.

  Diese wird kann z.B. mit `<cursor-name>%ROWTYPE` mit dem Record-Typ des Cursors definiert werden:

  ```
  <variable> <cursor-name>%ROWTYPE;
  ```
- `CLOSE <cursor-name>;` schließt einen Cursor.

Beispiel

```
DECLARE CURSOR cities_in
   (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
city_in cities_in%ROWTYPE;
BEGIN
   OPEN cities_in ('D');
   FETCH cities_in INTO city_in;
   CLOSE cities_in;
END;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Cursore

nicht möglich:

```
OPEN cities_in ('D');
OPEN cities_in ('CH');
FETCH cities_in INTO <variable>;
```

- *ein* parametrisierter Cursor,
- *nicht* eine Familie von Cursoren!

Cursore: Attribute

- `<cursor-name>%ISOPEN`: Cursor offen?
- `<cursor-name>%FOUND`: Solange ein Cursor bei der letzten `FETCH`-Operation ein neues Tupel gefunden hat, ist `<cursor-name>%FOUND = TRUE`.
- `<cursor-name>%NOTFOUND`: `TRUE` wenn man alle Zeilen eines Cursors ge`FETCH`t hat.
- `<cursor-name>%ROWCOUNT`: Anzahl der von einem Cursor bereits gelesenen Tupel.
- nicht innerhalb eines SQL-Ausdrucks.

Cursor FOR LOOP

```
FOR <record_index> IN <cursor-name>
LOOP ... END LOOP;
```

- `<record_index>` wird dabei *automatisch* als Variable vom Typ `<cursor-name>%ROWTYPE` deklariert,
- `<record_index>` *immer* von einem Record-Type – ggf. einspaltig.
- Es wird automatisch ein `OPEN` ausgeführt,
- bei jeder Ausführung des Schleifenkörpers wird *automatisch* ein `FETCH` ausgeführt,
- → Schleifenkörper enthält i.a. *keinen* `FETCH`-Befehl,
- am Ende wird automatisch ein `CLOSE` ausgeführt,
- Spalten müssen explizit adressiert werden.

Cursor FOR LOOP

Beispiel: Für jede Stadt in dem gegebenen Land soll eine Prozedur “request_Info” aufgerufen werden:

```
DECLARE CURSOR cities_in
   (the_country country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country = the_country;
BEGIN
   the_country:='D'; % oder sonstwie setzen
   FOR the_city IN cities_in(the_country)
   LOOP
      request_Info(the_city.name);
   END LOOP;
END;
```

Cursor FOR LOOP

- SELECT-Anfrage kann auch direkt in die FOR-Klausel geschrieben werden.

```
CREATE TABLE big_cities
(name VARCHAR2(25));

BEGIN
  FOR the_city IN
   SELECT Name
   FROM City
   WHERE Country = the_country
   AND Population > 1000000
  LOOP
   INSERT INTO big_cities
      VALUES (the_city.Name);
  END LOOP;
END;
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Schreibzugriff via Cursor

Mit WHERE CURRENT OF <cursor-name> kann man auf das zuletzt von dem genannten Cursor geFETCHte Tupel zugreifen:

```
UPDATE <table-name>
SET <set_clause>
WHERE CURRENT OF <cursor_name>;

DELETE FROM <table-name>
WHERE CURRENT OF <cursor_name>;
```

- Dabei bestimmt die Positionierung des Cursors bezüglich der Basistabellen den Ort der Änderung (im Gegensatz zu View Updates).

Zugriffsrechte

Benutzung von Funktionen/Prozeduren:

- Benutzungsrechte vergeben:
  `GRANT EXECUTE ON <procedure/function> TO <user>;`
- Prozeduren und Funktionen werden jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt.
- nach
  `GRANT EXECUTE ON <procedure/function> TO <user>;`
  kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat.
- Möglichkeit, Zugriffsberechtigungen strenger zu formulieren als mit `GRANT ... ON <table> TO ...`:
  Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

Geschachtelte Tabellen unter PL/SQL

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

Nutzung geschachtelter Tabellen in ORACLE nicht ganz unproblematisch:
“Bestimme alle Länder, in denen Deutsch gesprochen wird, sowie den Anteil der deutschen Sprache in dem Land”

Eine solche Anfrage muß für *jedes* Tupel in *Nested_Languages* die innere Tabelle untersuchen.

- `SELECT THE` kann jeweils nur ein Objekt zurückgeben,
- keine Korrelation mit umgebenden Tupeln möglich.
- Verwendung einer (Cursor-)Schleife.

Geschachtelte Tabellen unter PL/SQL

```
CREATE TABLE tempCountries
 (Land    VARCHAR2(4),
  Sprache VARCHAR2(20),
  Anteil  NUMBER);

CREATE OR REPLACE PROCEDURE Search_Countries
      (the_Language IN VARCHAR2)
IS CURSOR countries IS
   SELECT Code
   FROM Country;
BEGIN
   DELETE FROM tempCountries;
   FOR the_country IN countries
   LOOP
      INSERT INTO tempCountries
      SELECT the_country.code,Name,Percentage
      FROM THE(SELECT Languages
               FROM Nested_Language
               WHERE Country = the_country.Code)
      WHERE Name = the_Language;
   END LOOP;
END;
/

EXECUTE Search_Countries('German');
SELECT * FROM tempCountries;
```


- Bis jetzt: Funktionen und Prozeduren werden durch den Benutzer explizit aufgerufen.
- Trigger: Ausführung wird durch das Eintreten eines Ereignisses in der Datenbank angestoßen.

Einschub: Integritätsbedingungen

- Spalten- und Tabellenbedingungen
- Wertebereichsbedingungen (*domain constraints*),
- Verbot von Nullwerten,
- Uniqueness und Primärschlüssel-Bedingungen,
- `CHECK`-Bedingungen.

! Alles nur als Bedingungen an *eine* Zeile innerhalb *einer* Tabelle formulierbar.

Assertions

- Bedingungen, die den gesamten DB-Zustand betreffen.

  `CREATE ASSERTION <name> CHECK (<bedingung>)`

- Diese werden allerdings von ORACLE8 nicht unterstützt.

⇒ Also muß man sich etwas anderes überlegen.

Trigger

- spezielle Form von PL/SQL-Prozeduren,
- werden beim Eintreten eines bestimmten Ereignisses ausgeführt.
- Spezialfall aktiver Regeln nach dem **E**vent-**C**ondition-**A**ction-Paradigma.
- einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet.
- Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst (Event).
- Ausführung von Bedingungen an den Datenbankzustand abhängig (Condition).
- Action:
- *vor* oder *nach* der Ausführung der entsprechenden aktivierenden Anweisung ausgeführt.
- einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) ausgeführt.
- Trigger-Aktion kann auf den alten und neuen Wert des gerade behandelten Tupels zugreifen.

Trigger

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  BEFORE | AFTER
  {INSERT | DELETE | UPDATE} [OF <column-list>]
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ⋮
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ON <table>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH ROW]
  [WHEN (<condition>)]
  <pl/sql-block>;
```

- `BEFORE`, `AFTER`: Trigger wird vor/nach der auslösenden Operation ausgeführt.
- `OF <column>` (nur für `UPDATE`) schränkt Aktivierung auf angegebene Spalte ein.
- Zugriff auf Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion mittels `:OLD` bzw. `:NEW`. (Aliasing durch `REFERENCING OLD AS ... NEW AS ...`).

  Schreiben in `:NEW`-Werte nur mit `BEFORE`-Trigger.
- `FOR EACH ROW`: Row-Trigger, sonst Statement-Trigger.
- `WHEN (<condition>)`: zusätzliche Bedingung; `OLD` und `NEW` sind in `<condition>` erlaubt.

Trigger: Beispiel

Wenn ein Landes-Code geändert wird, pflanzt sich diese Änderung auf die Relation *Province* fort:

```
CREATE OR REPLACE TRIGGER change_Code
BEFORE UPDATE OF Code ON Country
FOR EACH ROW
BEGIN
  UPDATE Province
  SET Country = :NEW.Code
  WHERE Country = :OLD.Code;
END;
/

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

Trigger: Beispiel

Wenn ein Land neu angelegt wird, wird ein Eintrag in *Politics* mit dem aktuellen Jahr erzeugt:

```
CREATE TRIGGER new_Country
AFTER INSERT ON Country
FOR EACH ROW
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code,SYSDATE);
END;
/

INSERT INTO Country (Name,Code)
  VALUES ('Lummerland', 'LU');

SELECT * FROM Politics;
```


Trigger: Mutating Tables

- Zeilenorientierte Trigger: immer direkt vor/nach der Veränderung einer Zeile aufgerufen
- jede Ausführung des Triggers sieht einen anderen Datenbestand der Tabelle, auf der er definiert ist, sowie der Tabellen, die er evtl. ändert
- $\leadsto$ Ergebnis *abhängig von der Reihenfolge* der veränderten Tupel

ORACLE: Betroffene Tabellen werden während der gesamten Aktion als *mutating* gekennzeichnet, können nicht von Triggern gelesen oder geschrieben werden.

Nachteil: Oft ein zu strenges Kriterium.

- Trigger soll auf Tabelle zugreifen auf der er selber definiert ist.
  - Nur das auslösende Tupel soll von dem Trigger gelesen/geschrieben werden: Verwendung eines `BEFORE`-Triggers und der `:NEW`- und `:OLD`-Variablen
  - Es sollen neben dem auslösenden Tupel auch weitere Tupel verwendet werden: Verwendung eines Statement-orientierten Triggers
- Trigger soll auf andere Tabellen zugreifen: Verwendung von Statement-Triggern und ggf. Hilfstabellen.

INSTEAD OF-Trigger

- *View Updates*: Updates müssen auf Basistabellen umgesetzt werden.
- View-Update-Mechanismen eingeschränkt.
- INSTEAD OF-Trigger: Änderung an einem View wird durch andere SQL-Anweisungen ersetzt.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  INSTEAD OF
  {INSERT | DELETE | UPDATE} ON <view>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH STATEMENT]
  <pl/sql-block>;
```

- Keine Einschränkung auf bestimmte Spalten möglich
- Keine WHEN-Klausel
- Default: FOR EACH ROW

Praktikum: Datenbankprogrammierung in SQL/ORACLE

View Updates und INSTEAD OF-Trigger

```
CREATE OR REPLACE VIEW AllCountry AS
SELECT Name, Code, Population, Area,
       GDP, Population/Area AS Density,
       Inflation, population_growth,
       infant_mortality
FROM Country, Economy, Population
WHERE Country.Code = Economy.Country
  AND Country.Code = Population.Country;

INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP,
  Inflation, population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);
```

Fehlermeldung: Über ein Join-View kann nur *eine* Basistabelle modifiziert werden.

View Updates und `INSTEAD OF`-Trigger

```
CREATE OR REPLACE TRIGGER InsAllCountry
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
  INSERT INTO
    Country (Name,Code,Population,Area)
  VALUES (:NEW.Name, :NEW.Code,
          :NEW.Population, :NEW.Area);
  INSERT INTO Economy (Country,Inflation)
  VALUES (:NEW.Code, :NEW.Inflation);
  INSERT INTO Population
      (Country, Population_Growth,infant_mortality)
  VALUES (:NEW.Code, :NEW.Population_Growth,
          :NEW.infant_mortality);
END;
/
```

- aktualisiert *Country*, *Economy* und *Population*.
- Trigger *New_Country* (`AFTER INSERT ON COUNTRY`) aktualisiert zusätzlich *Politics*.

Fehlerbehandlung

- Declaration Section: Deklaration (der Namen) benutzerdefinierter Exceptions.

  ```
  DECLARE <exception> EXCEPTION;
  ```

- Exception Section: Definition der beim Auftreten einer Exception auszuführenden Aktionen.

  ```
  WHEN <exception>
     THEN <PL/SQL-Statement>;
  WHEN OTHERS THEN <PL/SQL-Statement>;
  ```

- Exceptions können dann an beliebigen Stellen des PL/SQL-Blocks durch `RAISE` ausgelöst werden.

  ```
  IF <condition>
    THEN RAISE <exception>;
  ```

Ablauf

- auslösen einer Exception
- entsprechende Aktion der `WHEN`-Klausel ausführen
- innersten Block verlassen (oft Anwendung von anonymen Blöcken sinnvoll)

Trigger/Fehlerbehandlung: Beispiel

Nachmittags dürfen keine Städte gelöscht werden:

```
CREATE OR REPLACE TRIGGER nachm_nicht_loeschen
BEFORE DELETE ON City
BEGIN
  IF TO_CHAR(SYSDATE,'HH24:MI')
   BETWEEN '12:00' AND '18:00'
  THEN RAISE_APPLICATION_ERROR
   (-20101,'Unerlaubte Aktion');
  END IF;
END;
/
```


Beispiel

```
CREATE OR REPLACE TRIGGER bla
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 IF user='may'
  THEN NULL;
 END IF;
...
END;
/

INSERT INTO AllCountry
 (Name, Code, Population, Area, GDP, Inflation,
  population_growth, infant_mortality)
 VALUES ('Lummerland','LU',4,1,0.5,0,25,0);

1 Zeile wurde erstellt.

SQL> select * from allcountry where Code='LU';

Es wurden keine Zeilen ausgewaehlt
```

(aus A. Christiansen, M. Höding, C. Rautenstrauch und G. Saake, ORACLE 8 effizient einsetzen, Addison-Wesley, 1998)

Weitere PL/SQL-Features

- *Packages*: Möglichkeit, Daten und Programme zu kapseln;
- `FOR UPDATE`-Option bei Cursordeklarationen;
- *Cursorvariablen*;
- *Exception Handlers*;
- *benannte* Parameterübergabe;
- PL-SQL Built-in Funktionen: Parsing, String-Operationen, Datums-Operationen, Numerische Funktionen;
- Built-in Packages.

- Definition komplexer Transaktionen,
- Verwendung von `SAVEPOINT`s für Transaktionen,

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Objekt-Relationale Datenbanksysteme

Integration von relationalen Konzepten und Objektorientierung:

- Komplexe Datentypen: Erweiterung des Domain-Konzepts von SQL-2
- Abstrakte Datentypen ("Objekttypen"): Objekt-Identität und Kapselung interner Funktionalität.
- Spezialisierung: Klassenhierarchie; Subtypen als Spezialisierung allgemeiner Typen.
- Tabellen als Subtabellen von Tabellen.
- Funktionen als Bestandteile eines ADT's oder von Tabellen, oder freie Funktionen.
- Methoden- und Funktionsaufrufe im Rahmen von `SELECT`-Ausdrücken.

Objektorientierung

- Unterscheidung zwischen dem *Zustand* und *Verhalten* eines *Objektes*.
- in ORACLE 8: Tabellen von Tupeln vs. *Object Tables* aus Objekten
- Im Gegensatz zu einem *Tupel* besitzt ein Objekt *Attribute* (beschreiben seinen Zustand) und *Methoden* (Abfragen und Ändern des Zustandes).
- Typ definiert gemeinsame Signatur seiner Instanzen (Objekte)
- bereits behandelt: Komplexe Attributtypen. Besitzen nur *Wertattribute*, keine Methoden.
- Methoden: *Prozeduren* und *Funktionen*
- MAP/ORDER-Funktion: Ordnung auf Objekttyp
- Spalten einer Tupeltabelle können *objektwertig* oder *referenzwertig* sein.
- Objekte: *Wertattribute* und *Referenzattribute*.
- ORACLE8: keine Subtypen und Vererbung

Typdeklaration: Attribute, *Signaturen* der Methoden, READ/WRITE Zugriffscharakteristik.

Type Body: Implementierung der Methoden in PL/SQL.

Objekttypdeklaration

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   ⋮
   <attr> REF <object-datatype>,
   ⋮
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>,
   ⋮
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
   ⋮
   [ MAP MEMBER FUNCTION <func-name>
         RETURN <datatype>, |
     ORDER MEMBER FUNCTION <func-name>(<var> <type>)
         RETURN <datatype>,]
   [ <pragma-declaration-list>]
   );
```

- <parameter-list> wie in PL/SQL,
- ähnlich CREATE TABLE, aber *keine* Integritätsbedingungen (erst bei der (Objekt)tabellen-Definition)

`PRAGMA`-Klauseln:

Read/Write-Zugriffscharakteristik

`<pragma-declaration-list>`:

für jede Methode eine `PRAGMA`-Klausel

```
PRAGMA RESTRICT_REFERENCES
  (<method_name>,<feature-list>);
```

`<feature-list>`:

`WNDS`	Writes no database state,
`WNPS`	Writes no package state,
`RNDS`	Reads no database state,
`RNPS`	Reads no package state.

Funktionen: werden nur ausgeführt, wenn *zugesichert* ist, daß sie den Datenbankzustand nicht verändern:

```
PRAGMA RESTRICT_REFERENCES
  (<function_name>,WNPS,WNDS);
```

`MAP`/`ORDER`**-Funktionen:** kein Datenbankzugriff erlaubt

```
PRAGMA RESTRICT_REFERENCES
  (<function-name>,WNDS,WNPS,RNPS,RNDS)
```

⇒ verwendet nur den Zustand der beteiligten Objekte.

Beispiel: Geo-Koordinaten

- Methode *Distance*(geo-coord-Wert)
- `MAP`-Methode: Entfernung von Greenwich.

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude NUMBER,
   MEMBER FUNCTION
      Distance (other IN GeoCoord)
      RETURN NUMBER,
   MAP MEMBER FUNCTION
      Distance_Greenwich RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance,WNPS,WNDS,RNPS,RNDS),
   PRAGMA RESTRICT_REFERENCES
      (Distance_Greenwich,WNPS,WNDS,RNPS,RNDS)
  );
/
```

Type Body

- Implementierung der Objektmethoden,
- muß der der bei `CREATE TYPE` vorgegeben Signatur entsprechen,
- für *alle* deklarierten Methoden muß Implementierung angegeben werden.
- Methode `SELF`, um auf die Attribute des Host-Objektes zuzugreifen.

Type Body

```
CREATE [OR REPLACE] TYPE BODY <type>
AS
   MEMBER FUNCTION <func-name> [(<parameter-list>)]
      RETURN <datatype>
   IS
      [<var-decl-list>;]
      BEGIN <PL/SQL-code> END;
   ⋮
   MEMBER PROCEDURE <proc-name> [(<parameter-list>)]
   IS
      [<var-decl-list>;]
      BEGIN <PL/SQL-code> END;
   ⋮
   [MAP MEMBER FUNCTION <func-name>
        RETURN <datatype> |
    ORDER MEMBER FUNCTION <func-name>(<var> <type>)
        RETURN <datatype>
    IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;]
END;
/
```

Erzeugung von Objekten

- Konstruktormethode:

  `<type>(<arg_1>, ..., <arg_n>)`

Methodenaufruf

(aus einem PL-SQL Programm)

`<object>.<method-name>(<argument-list>)`

mit `SELF` für <object> kann ein Objekt seine eigenen Methoden aufrufen.

Beispiel: Geo-Koordinaten

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS

MEMBER FUNCTION Distance (other IN GeoCoord)
  RETURN NUMBER
  IS
  BEGIN
   RETURN 6370 * ACOS(cos(SELF.latitude/180*3.14)
              * cos(other.latitude/180*3.14)
              * cos((SELF.longitude-
                     other.longitude)/180*3.14)
              + sin(SELF.latitude/180*3.14)
              * sin(other.latitude/180*3.14));
  END;

MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;

END;
/
```

Spaltenobjekte

- Attribut eines Tupels (oder eines Objekts) ist objektwertig,
- *keine* OID, also *nicht* referenzierbar.

Beispiel: Geo-Koordinaten

```
CREATE TABLE Mountain
  (Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
   Height NUMBER CONSTRAINT MountainHeight
      CHECK (Height > = 0),
   Coordinates GeoCoord CONSTRAINT MountainCoord
      CHECK ((Coordinates.Longitude >= -180) AND
             (Coordinates.Longitude <= 180) AND
             (Coordinates.Latitude >= -90) AND
             (Coordinates.Latitude <= 90)));
```

- Constraints werden wie immer bei der Tabellendefinition angegeben.

```
INSERT INTO Mountain
   VALUES ('Feldberg', 1493, GeoCoord(8,48));
SELECT Name, mt.coordinates.distance(geocoord(0,90))
FROM Mountain mt;
```

- Tupelvariable *mt* um den Zugriffspfad zu *coordinates.distance* eindeutig zu machen.

Zeilenobjekte

- Elemente von *Objekttabellen*,
- erhalten eine eindeutige OID und sind damit referenzierbar.
- OID entspricht dem *Primärschlüssel* und wird mit den (weiteren) Integritätsbedingungen bei der Tabellendefinition angegeben.
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.

```
CREATE TABLE <name> OF <object-datatype>
  [(<constraint-list>)];
```

`<constraint-list>`:

- attributbezogene Bedingungen entsprechen den Spaltenbedingungen:

```
<attr-name> [DEFAULT <value>]
   [<colConstraint> ... <colConstraint>]
```

- Tabellenbedingungen: Syntax wie bei Tupeltabellen.

Zeilenobjekte

Beispiel: *City_Type*

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province VARCHAR2(32),
   Country VARCHAR2(4),
   Population NUMBER,
   Coordinates GeoCoord,
   MEMBER FUNCTION Distance (other IN City_Type)
      RETURN NUMBER,
   PRAGMA RESTRICT_REFERENCES
      (Distance,WNPS,WNDS,RNPS,RNDS));
/
```

```
CREATE OR REPLACE TYPE BODY City_Type
AS
  MEMBER FUNCTION Distance (other IN City_Type)
  RETURN NUMBER
  IS
  BEGIN
   RETURN self.coordinates.distance(other.coordinates);
  END;
END;
/
```


Objekttabellen: Zeilenobjekte

- der (mehrspaltige) Primärschlüssel wird als Tabellenbedingung angegeben,
- Primärschlüssel darf keine Referenzattribute umfassen,
- Die Fremdschlüsselbedingung auf die relationale Tabelle *Country* wird ebenfalls als Tabellenbedingung angegeben:

```
CREATE TABLE City_ObjTab OF City_Type
  (PRIMARY KEY (Name, Province, Country),
   FOREIGN KEY (Country) REFERENCES Country(Code));
```

- Objekte werden unter Verwendung des Objektkonstruktors `<object-datatype>` in Objekttabellen eingefügt.

```
INSERT INTO City_ObjTab
SELECT City_Type
  (Name,Province,Country,Population,
   GeoCoord(Longitude,Latitude))
FROM City
WHERE Country = 'D'
  AND NOT Longitude IS NULL;
```

Verwendung von Objekten

- Zeilenobjekt *als Ganzes* selektieren,

  `VALUE (<var>)`

  in Kombination mit Aliasing

  `FROM <tabelle> <var>`

- z.B. Vergleich oder in einer `ORDER BY`-Klausel,

Beispiel

```
SELECT VALUE(cty)
FROM City_ObjTab cty;
```

VALUE(Cty)(Name, Province, Country, Population, Coordinates(Longitude, Latitude))
City_Type('Berlin', 'Berlin', 'D', 3472009, GeoCoord(13, 52))
City_Type('Bonn', 'Nordrh.-Westf., 'D', 293072, GeoCoord(8, 50))
City_Type('Stuttgart', 'Baden-Wuertt., 'D', 588482, GeoCoord(9, 49))
⋮

Verwendung von Objekten: VALUE

- Objekte auf Gleichheit testen
- Objekt als Argument einer Methode

```
SELECT cty1.Name, cty2.Name,
   cty1.coordinates.Distance(cty2.coordinates)
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

```
SELECT cty1.Name, cty2.Name,
   cty1.Distance(VALUE(cty2))
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

- Zuweisung eines Objektes mit einem `SELECT INTO`-Statement an eine PL/SQL-Variable

```
SELECT VALUE(<var>) INTO <PL/SQL-Variable>
FROM <tabelle> <var>
WHERE ... ;
```

Objektreferenzen

- Weiterer Datentyp für Attribute: Referenzen auf Objekte

  `<ref-attr> REF <object-datatype>`

- `PRIMARY KEY`s dürfen keine `REF`-Attribute umfassen.
- *Objekttyp* als Ziel der Referenz
- nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – können referenziert werden.
- Objekttyp kann in verschiedenen Tabellen vorkommen
- Einschränkung auf eine bestimmte Tabelle bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit `SCOPE`:
  - als Spaltenconstraint (nur bei Tupeltabellen):

    ```
    <ref-attr> REF <object-datatype>
      SCOPE IS <object-table>
    ```

  - als Tabellenconstraint:

    ```
    SCOPE FOR (<ref-attr>) IS <object-table>
    ```

- Erzeugen einer Referenz (Selektieren einer OID):

  ```
  SELECT ..., REF(<var>), ...
  FROM <objekt-tabelle> <var>
  WHERE ... ;
  ```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Beispiel: Objekttyp *Organization*

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(30));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/

CREATE OR REPLACE TYPE Organization_Type AS OBJECT
  (Name VARCHAR2(80),
   Abbrev VARCHAR2(12),
   Members Member_List_Type,
   Established DATE,
   has_hq_in REF City_Type,
   MEMBER FUNCTION is_member (the_country IN VARCHAR2)
  -- EU.is_member('SL') = 'membership applicant'
      RETURN VARCHAR2,
   MEMBER FUNCTION people RETURN NUMBER,
   MEMBER FUNCTION number_of_members RETURN NUMBER,
   MEMBER PROCEDURE add_member
      (the_country IN VARCHAR2, the_type IN VARCHAR2),
   PRAGMA RESTRICT_REFERENCES (is_member,WNPS,WNDS),
   PRAGMA RESTRICT_REFERENCES (people,WNDS,WNPS));
   PRAGMA RESTRICT_REFERENCES (number_of_members,WNDS,WNPS))
/
```

Beispiel: Objekttyp *Organization*

Tabellendefinition:

```
CREATE TABLE Organization_ObjTab OF Organization_Type
  (Abbrev PRIMARY KEY,
   SCOPE FOR (has_hq_in) IS City_ObjTab)
   NESTED TABLE Members STORE AS Members_nested;
```

Einfügen unter Verwendung des Objektkonstruktors:

```
INSERT INTO Organization_ObjTab VALUES
  (Organization_Type('European Community','EU',
                     Member_List_Type(),NULL,NULL));
```

Referenzattribut *has_hq_in*:

```
UPDATE Organization_ObjTab
SET has_hq_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty
   WHERE Name = 'Brussels'
     AND Province = 'Brabant'
     AND Country = 'B')
WHERE Abbrev = 'EU';
```


Selektion von Objektattributen

- Wertattribute

```
SELECT Name, Abbrev, Members
FROM Organization_ObjTab;
```

Name	Abbrev	Members
European Community	EU	Member_List_Type(...)

- Referenzattribute:

  `SELECT <ref-attr-name>`

  liefert OID:

```
SELECT Name, Abbrev, has_hq_in
FROM Organization_ObjTab;
```

Name	Abbrev	has_hq_in
European Community	EU	<oid>

- `DEREF(<oid>)` liefert das zugehörige Objekt:

```
SELECT Abbrev, DEREF(has_hq_in)
FROM Organization_ObjTab;
```

Abbrev	has_hq_in
EU	City_Type('Brussels', 'Brabant', 'B', 951580, GeoCoord(4, 51))

Verwendung von Referenzattributen

- Attribute und Methoden eines referenzierten Objekts werden durch *Pfadausdrücke* der Form

  `SELECT <ref-attr-name>.<attr-name>`

  adressiert ("*navigierender Zugriff*").

- Aliasing mit einer Variablen um den Pfadausdruck eindeutig zu machen.

```
SELECT Abbrev, org.has_hq_in.name
FROM Organization_ObjTab org;
```

Abbrev	has_hq_in.Name
EU	Brussels

Mit `REF` und `DEREF` läßt sich `VALUE` ersetzen:

```
SELECT VALUE(cty) FROM City_ObjTab cty;
```

und

```
SELECT DEREF(REF(cty)) FROM City_ObjTab cty;
```

sind äquivalent.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Zyklische Referenzen

- `City_Type: country REF Country_Type`
- `Country_Type: capital REF City_Type`
- Deklaration jedes Datentypen benötigt bereits die Definition des anderen
- Definition von *unvollständigen* Typen "Forward-Deklaration"

  ```
  CREATE TYPE <name>;
  /
  ```

- wird später durch eine komplette Typdeklaration ergänzt

Zyklische Referenzen: Beispiel

```
CREATE OR REPLACE TYPE City_Type
/

CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/

CREATE OR REPLACE TYPE Province_Type AS OBJECT
  (Name VARCHAR2(32),
   Country REF Country_Type,
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/

CREATE OR REPLACE TYPE City_Type AS OBJECT
  (Name VARCHAR2(35),
   Province REF Province_Type,
   Country REF Country_Type,
   Population NUMBER,
   Coordinates GeoCoord);
/
```


Unvollständige Datentypen

Unvollständige Datentypen können nur zur Definition von *Referenzen* auf sie benutzt werden, nicht zur Definition von Spalten oder in geschachtelten Tabellen:

```
CREATE TYPE City_type;
/
```

```
erlaubt:
CREATE TYPE city_list AS TABLE OF REF City_type;
/

CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital REF City_Type);
/
```

```
erst erlaubt, wenn city_type komplett ist:
CREATE TYPE city_list AS TABLE OF City_type;
/

CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(32),
   Code VARCHAR2(4),
   Capital City_Type);
/
```

Referentielle Integrität

- Vgl. `FOREIGN KEY ... REFERENCES ... ON DELETE/UPDATE CASCADE`
- Veränderungen an Objekten:
  OID bleibt unverändert
  → referentielle Integrität bleibt gewahrt.
- Löschen von Objekten:
  dangling references möglich.

  Überprüfung durch

  ```
  WHERE <ref-attribute> IS DANGLING
  ```

  Verwendung z.B. in einem `AFTER`-Trigger:

  ```
  UPDATE <table>
    SET <attr> = NULL
    WHERE <attr> IS DANGLING;
  ```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Methoden: Funktionen und Prozeduren

- `TYPE BODY` enthält die Implementierungen der Methoden in PL/SQL
- PL/SQL an geschachtelte Tabellen und objektorientierte Features angepaßt.
- PL/SQL unterstützt keine Navigation entlang Pfadausdrücken (in SQL ist es erlaubt).
- Jede `MEMBER METHOD` besitzt einen *impliziten* Parameter `SELF`, der das jeweilige Host-Objekt referenziert.
- Tabellenwertige Attribute können innerhalb PL/SQL wie PL/SQL-Tabellen behandelt werden:

  Built-in Methoden für Collections (PL/SQL-Tabellen) können auch auf tabellenwertige Attribute angewendet werden:

  `<attr-name>.COUNT`: Anzahl der in der geschachtelten Tabelle enthaltenen Tupel

  Verwendung in in PL/SQL eingebetteten SQL-Statements – z.B. `SELECT <attr>.COUNT` – nicht erlaubt.
- zukünftige Erweiterung: Java

```
CREATE OR REPLACE TYPE BODY Organization_Type IS

MEMBER FUNCTION is_member (the_country IN VARCHAR2)
  RETURN VARCHAR2
IS
BEGIN
  IF SELF.Members IS NULL OR SELF.Members.COUNT = 0
   THEN RETURN 'no'; END IF;
  FOR i in 1 .. Members.COUNT
  LOOP
   IF the_country = Members(i).country
      THEN RETURN Members(i).type; END IF;
  END LOOP;
  RETURN 'no';
END;

MEMBER FUNCTION people RETURN NUMBER IS
p NUMBER;
BEGIN
  SELECT SUM(population) INTO p
  FROM Country ctry
  WHERE ctry.Code IN
  (SELECT Country
   FROM THE (SELECT Members
               FROM Organization_ObjTab org
               WHERE org.Abbrev = SELF.Abbrev));
  RETURN p;
END;
```



```
MEMBER FUNCTION number_of_members RETURN NUMBER
IS
BEGIN
  IF SELF.Members IS NULL THEN RETURN 0; END IF;
  RETURN Members.COUNT;
END;

MEMBER PROCEDURE add_member
   (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  IF NOT SELF.is_member(the_country) = 'no'
   THEN RETURN; END IF;
  IF SELF.Members IS NULL THEN
   UPDATE Organization_ObjTab
   SET Members = Member_List_Type()
   WHERE Abbrev = SELF.Abbrev;
  END IF;
  INSERT INTO
  THE (SELECT Members
   FROM Organization_ObjTab org
   WHERE org.Abbrev = SELF.Abbrev)
  VALUES (the_country,the_type);
END;
END;
/
```

- `FROM THE(SELECT ...)` kann nicht durch `FROM SELF.Members` ersetzt werden (PL/SQL vs. SQL).

Methodenaufrufe: Funktionen

- `MEMBER FUNCTIONS` können in SQL und PL/SQL durch `<object>.<function>(<argument-list>)` selektiert werden.
- parameterlose Funktionen: `<object>.<function>()`
- aus SQL: `<object>` ist durch einen Pfadausdruck mit Alias gegeben.

```
SELECT Name, org.is_member('D')
  FROM Organization_ObjTab org
  WHERE NOT org.is_member('D') = 'no';
```

- `MEMBER PROCEDURES` können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden.
- freie Prozeduren in PL/SQL, um `MEMBER PROCEDURES` aufzurufen

Methodenaufrufe: Prozeduren

```
CREATE OR REPLACE PROCEDURE make_member
  (the_org IN VARCHAR2, the_country IN VARCHAR2,
   the_type IN VARCHAR2) IS
  n NUMBER;
  c Organization_Type;
BEGIN
  SELECT COUNT(*) INTO n
   FROM Organization_ObjTab
   WHERE Abbrev = the_org;
  IF n = 0
  THEN INSERT INTO Organization_ObjTab
   VALUES(Organization_Type(NULL,
      the_org,Member_List_Type(),NULL,NULL));
  END IF;
  SELECT VALUE(org) INTO c
   FROM Organization_ObjTab org
   WHERE Abbrev = the_org;
  IF c.is_member(the_country)='no' THEN
   c.add_member(the_country,the_type);
  END IF;
END;
/
EXECUTE make_member('EU','USA','special member');
EXECUTE make_member('XX','USA','member');
```

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *is_member* in die Objekttabelle *Organization_ObjTab*:

```
INSERT INTO Organization_ObjTab
 (SELECT Organization_Type
     (Name,Abbreviation,NULL,Established,NULL)
  FROM Organization);
```

```
CREATE OR REPLACE PROCEDURE Insert_All_Members IS
BEGIN
 FOR the_membership IN
     (SELECT * FROM is_member)
 LOOP make_member(the_membership.organization,
                  the_membership.country,
                  the_membership.type);
 END LOOP;
END;
/
EXECUTE Insert_All_Members;
```

```
UPDATE Organization_ObjTab org
SET has_hq_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
   AND cty.Name = old.City
   AND cty.Province = old.Province
   AND cty.Country = old.Country);
```


Verwendung von Objekten

```
CREATE OR REPLACE FUNCTION is_member_in
     (the_org IN VARCHAR2, the_country IN VARCHAR2)
 RETURN is_member.Type%TYPE IS
 c is_member.Type%TYPE;
BEGIN
  SELECT org.is_member(the_country) INTO c
  FROM Organization_ObjTab org
  WHERE Abbrev=the_org;
  RETURN c;
END;
/
```

Die system-eigene Tabelle `DUAL` wird verwendet um das Ergebnis freier Funktionen ausgeben zu lassen.

```
SELECT is_member_in('EU','SL')
FROM DUAL;
```

is_member_in('EU','SL')
applicant

Es ist (zumindest in ORACLE 8.0) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

```
UPDATE Organization_ObjTab org
SET org.has_hq_in.Name = 'UNO City'    -- NICHT ERLAUBT
WHERE org.Abbrev = 'UN';
```

ORDER- und MAP-Methoden

- Objekttypen besitzen im Gegensatz zu den Datentypen NUMBER und VARCHAR keine inhärente Ordnung.
- Ordnung auf Objekten eines Typs kann über dessen funktionale Methoden definiert werden.
- ORACLE 8: für jeden Objekttyp eine MAP FUNCTION oder ORDER FUNCTION.

MAP-Funktion:

- keine Parameter,
- bildet jedes Objekt auf eine Zahl ab.
- Lineare Ordnung auf dem Objekttyp, "Betragsfunktion"
- sowohl für Vergleiche <, > und BETWEEN, als auch für ORDER BY verwendbar.

ORDER-Funktion:

- besitzt *ein* Argument desselben Objekttyps das mit dem Hostobjekt verglichen wird.
- Damit sind ORDER-Funktionen für Vergleiche <, > geeignet, im allgemeinen aber nicht unbedingt für Sortierung.

- MAP- und ORDER-Funktionen erfordern PRAGMA RESTRICT_REFERENCES (<name>,WNDS,WNPS,RNPS,RNDS), d.h. sie dürfen *keinen Datenbankzugriff* enthalten.

MAP-Methoden: Beispiel

MAP-Methode auf *GeoCoord*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
...
MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;
END;
/
```

```
SELECT Name, cty.coordinates.longitude,
          cty.coordinates.latitude
FROM City_ObjTab cty
WHERE NOT coordinates IS NULL
ORDER BY coordinates;
```

`MAP`-Methoden

Viele Operationen sind in `MAP`-Methoden nicht erlaubt:

- keine Anfragen an die Datenbank:

  In *Organization_Type* kann *People* nicht als `MAP`-Methode verwendet werden.

- keine built-in Methoden von geschachtelten Tabellen:

  In *Organization_Type* kann *number_of_members* ebenfalls nicht als `MAP`-Methode verwendet werden.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

`ORDER`-Methoden

- Vergleich von `SELF` mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird.
- Ergebnis -1 (`SELF` < Parameter), 0 (Gleichheit) oder 1 (`SELF` > Parameter)
- Wird `ORDER BY` angegeben, werden die Ausgabeobjekte paarweise verglichen und entsprechend der `ORDER`-Methode geordnet.
- Ein Beispiel hierfür ist etwa die Erstellung der Fußball-Bundesligatabelle: Ein Verein wird vor einem anderen plaziert, wenn er mehr Punkte hat. Bei Punktgleichheit entscheidet die Tordifferenz. Ist auch diese dieselbe, so entscheidet die Anzahl der geschossenen Tore (vgl. Aufgabe).

Indexe auf Objektattributen

Indexe können auch auf Objektattributen erstellt werden:

```
CREATE INDEX <name>
  ON <object-table-name>.<attr>[.<attr>]*;
```

- Indexe können *nicht* über komplexen Attributen erstellt werden:

```
-- nicht erlaubt:
CREATE INDEX city_index
   ON City_ObjTab(coordinates);
```

- Indexe können über elementare Teilattribute eines komplexen Attributes erstellt werden:

```
CREATE INDEX city_index
   ON City_ObjTab(coordinates.Longitude,
                  coordinates.Latitude);
```

Zugriffsrechte auf Objekte

Recht an Objekttypen:

```
GRANT EXECUTE ON <Object-datatype> TO ...
```

- bei der Benutzung eines Datentyps stehen vor allem die Methoden (u.a. die entsprechende Konstruktormethode) im Vordergrund.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Änderungen von Objekttypen

- Verwendung von Objekttypen und Referenzattributen erzeugt ein ähnliches Netz von Referenzen wie bei der Definition von referentiellen Integritätsbedingungen.
- Änderungen an Objekttypen in ORACLE 8.0 sehr restriktiv: `CREATE OR REPLACE TYPE` und `ALTER TYPE` sind (zumindest in ORACLE 8.0) nicht erlaubt, wenn der Objekttyp irgendwo verwendet wird.

! nicht möglich, einem irgendwo verwendeten Objekttyp ein Attribut (oder auch nur eine Methode!) hinzuzufügen.

*In conclusion, carefully plan the object types for your database so that you get things right the first time. Then keep your fingers crossed and hope that things do not change once you have everything up and running (*ORACLE *8: Architecture).*

Ein erstes Fazit

- Daten*haltung* in einem objektorientierten Schema bereits bei kleinen Änderungen sehr problematisch.
- anwendungsorientierte (nicht-relationale) Repräsentation durch Methoden sowie freie Prozeduren und Funktionen.
- Integration von anwendungsspezifischer Funktionalität durch Objektmethoden vereinfacht.

⇒ Datenhaltung: relational
Nutzung: objektorientiert.

Object-Views

- maßgeschneiderte Object-Views mit sehr weitgehender Funktionalität

Legacy-Datenbanken: Integration bestehender Datenbanken in ein "modernes" objektorientiertes Modell:

Objekt-Views über relationale Ebene legen:
"Objekt-Abstraktionen"

Effizienz + Benutzerfreundlichkeit:
relationale Repräsentation oft effizienter:

- Geschachtelte Tabellen intern als separate Tabellen gespeichert.
- $n:m$-Beziehungen: gegenseitige geschachtelte Tabellen notwendig.

⇒ Definition eines relationalen Basisschemas mit Object-Views.

Einfache Modifizierbarkeit: `CREATE OR REPLACE TYPE` und `ALTER TYPE` nur sehr eingeschränkt

⇒ Veränderungen durch Definition geeigneter Object-Views abfangen.

Häufige Empfehlung: Object Views mit geschachtelten Tabellen, Referenzen etc. auf Basis eines relationalen Grundschemas verwenden.

Object-Views

Benutzer führt seine Änderungen auf dem durch die Objektviews gegebenen externen Schema durch.

- Abbildung direkter Änderungen (`INSERT`, `UPDATE` und `DELETE`) durch `INSTEAD OF`-Trigger auf das darunterliegende Schema.
- Benutzer darf erst gar keine solchen Statements an das View stellen. Entsprechende Funktionalität durch Methoden der Objekttypen, die die Änderungen direkt auf den zugrundeliegenden Basistabellen ausführen.

Objektrelationale Views

- Tupel-Views ohne Methoden:

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
  <select-clause>;
```

- `SELECT`-clause: zusätzlich Konstruktormethoden für Objekte und geschachtelte Tabellen.
- Für die Erzeugung geschachtelter Tabellen für Object Views werden die Befehle `CAST` und `MULTISET` verwendet.

Beispiel

```
CREATE TYPE River_List_Entry AS OBJECT
 (name VARCHAR2(20),
  length NUMBER);
/
CREATE TYPE River_List AS
 TABLE OF River_List_Entry;
/
CREATE OR REPLACE VIEW River_V
  (Name, Length, Tributary_Rivers)
AS SELECT
   Name, Length,
   CAST(MULTISET(SELECT Name,Length FROM River
                 WHERE River = A.Name) AS River_List)
   FROM River A;
```

Objektviews

- enthalten Zeilenobjekte, d.h. hier werden neue Objekte *definiert*.
- durch `WITH OBJECT OID <attr-list>` wird angegeben, wie die Objekt-ID berechnet wird werden soll.
- Verwendung von `CAST` und `MULTISET`.

```
CREATE [OR REPLACE] VIEW <name> OF <type>
  WITH OBJECT OID (<attr-list>)
  AS <select-statement>;
```

- in `<select-statement>` wird *kein* Objektkonstruktor verwendet!

Object Views: *Country*

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
 (Name       VARCHAR2(32),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/
```

```
CREATE OR REPLACE VIEW Country_ObjV OF Country_Type
WITH OBJECT OID (Code)
AS
SELECT Country.Name, Country.Code, REF(cty),
       Area, Country.Population
FROM Country, City_ObjTab cty
WHERE cty.Name = Country.Capital
  AND cty.Province = Country.Province
  AND cty.Country = Country.Code;
```

```
SELECT Name, Code, c.capital.name, Area, Population
FROM Country_ObjV c;
```

Object Views: Was nicht geht

- Object View darf keine geschachtelte Tabelle und
- kein Ergebnis einer funktionalen Methode einer zugrundeliegenden Tabelle enthalten.

Object View auf Basis von *Organization_ObjTab*:

```
CREATE OR REPLACE TYPE Organization_Ext_Type AS OBJECT
  (Name VARCHAR2(80),
   Abbrev VARCHAR2(12),
   Members Member_List_Type,
   established DATE,
   has_hq_in REF City_Type,
   number_of_people NUMBER);
/
```

```
CREATE OR REPLACE VIEW Organization_ObjV
  OF Organization_Ext_Type
AS
SELECT Name, Abbrev, Members, established,
       has_hq_in, org.people()
  FROM Organization_ObjTab org;
```

```
FEHLER in Zeile 3:
ORA-00932: nicht übereinstimmende Datentypen
```

Beide angegebenen Attribute sind auch einzeln nicht erlaubt.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Fazit

+ Kompatibilität mit den grundlegenden Konzepten von ORACLE 7.
  U.a. Fremdschlüsselbedingungen von Objekttabellen zu relationalen Tabellen.
+ *Object Views* erlauben ein objektorientiertes externes Schema. Benutzer-Interaktionen werden durch Methoden und `INSTEAD OF`-Trigger auf das interne Schema umgesetzt.
– Flexibilität/Ausgereiftheit:
  Typen können nicht verändert/erweitert werden.
  (inkrementelle!) Anpassungen des Schemas nicht möglich.

Embedded SQL, JDBC

Kopplungsarten zwischen Datenbank- und Programmiersprachen

- Erweiterung der Datenbanksprache um Programmierkonstrukte (z.B. PL/SQL)
- Erweiterung von Programmiersprachen um Datenbankkonstrukte: Persistente Programmiersprachen, Datenbank-Programmiersprachen
- Datenbankzugriff aus einer Programmiersprache
- Einbettung der Datenbanksprache in eine Programmiersprache: “Embedded SQL”

Embedded SQL

- C, Pascal, C++

Impedance Mismatch bei der SQL-Einbettung

- Typsysteme passen nicht zusammen
- Unterschiedliche Paradigmen: Mengenorientiert vs. einzelne Variablen

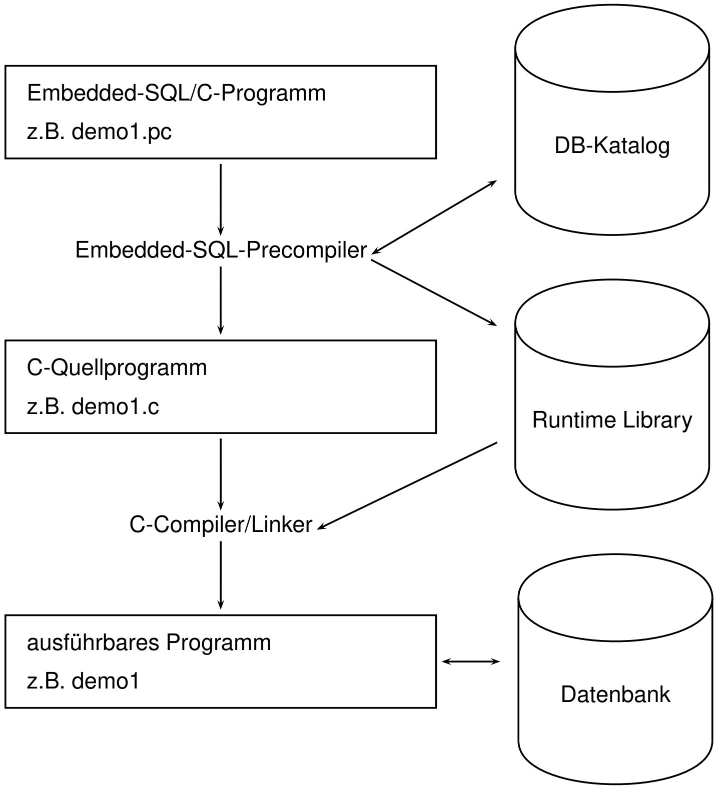
Realisierte Lösung

- Abbildung von Tupeln bzw. Attributen auf Datentypen der Hostsprache
- Iterative Verarbeitung der Ergebnismenge mittels Cursor

Auswirkungen auf die Hostsprache

- Struktur der Hostsprache bleibt unverändert
- Jede SQL-Anweisung kann eingebettet werden
- SQL-Anweisungen wird `EXEC SQL` vorangestellt
- Wie kommunizieren Anwendungsprogramm und Datenbank ?

Entwicklung einer Embedded SQL-Applikation



Verbindungsaufbau

Embedded-Anwendung: Verbindung zu einer Datenbank muß explizit hergestellt werden.

```
EXEC SQL CONNECT :username IDENTIFIED BY :passwd;
```

- `username` und `passwd` Hostvariablen vom Typ `CHAR` bzw. `VARCHAR`..
- Strings sind hier nicht erlaubt!

Äquivalent:

```
EXEC SQL CONNECT :uid;
```

wobei `uid` ein String der Form `"name/passwd"` ist.

Hostvariablen

- Kommunikation zwischen Datenbank und Anwendungsprogramm
- Output-Variablen übertragen Werte von der Datenbank zum Anwendungsprogramm
- Input-Variablen übertragen Werte vom Anwendungsprogramm zur Datenbank.
- jeder Hostvariablen zugeordnet: Indikatorvariable zur Verarbeitung von NULL-Werten.
- werden in der *Declare Section* deklariert:

```
EXEC SQL BEGIN DECLARE SECTION;
 int population;        /* host variable */
 short population\_ind;   /* indicator variable */
EXEC SQL END DECLARE SECTION;
```

- in SQL-Statements wird Hostvariablen und Indikatorvariablen ein Doppelpunkt (“:”) vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein

Indikatorvariablen

Verarbeitung von Nullwerten

Indikatorvariablen für Output-Variablen:

- -1 : der Attributwert ist `NULL`, der Wert der Hostvariablen ist somit undefiniert.
- 0 : die Hostvariable enthält einen gültigen Attributwert.
- $>$0 : die Hostvariable enthält nur einen Teil des Spaltenwertes. Die Indikatorvariable gibt die ursprüngliche Länge des Spaltenwertes an.
- -2 : die Hostvariable enthält einen Teil des Spaltenwertes wobei dessen ursprüngliche Länge nicht bekannt ist.

Indikatorvariablen für Input-Variablen:

- -1 : unabhängig vom Wert der Hostvariable wird `NULL` in die betreffende Spalte eingefügt.
- $>=$0 : der Wert der Hostvariable wird in die Spalte eingefügt.

Cursore

- Analog zu PL/SQL
- notwendig zur Verarbeitung einer Ergebnismenge, die mehr als ein Tupel enthält

Cursor-Operationen

- `DECLARE <cursor-name> CURSOR FOR <sql statement>`
- `OPEN <cursor-name>`
- `FETCH <cursor-name> INTO <varlist>`
- `CLOSE <cursor-name>`

Fehlersituationen

- der Cursor wurde nicht geöffnet bzw. nicht deklariert
- es wurden keine (weiteren) Daten gefunden
- der Cursor wurde geschlossen, aber noch nicht wieder geöffnet

`Current of`-**Klausel** analog zu PL/SQL

Beispiel

```
int main() {
   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* output host var */
      int cityEinw;         /* output host var */
      char* landID = "D";    /* input host var */
      short ind1, ind2;      /* indicator var */
      char* uid = "/";
   EXEC SQL END DECLARE SECTION;
   /* Verbindung zur Datenbank herstellen */
   EXEC SQL CONNECT :uid;
   /* Cursor deklarieren */
   EXEC SQL DECLARE StadtCursor CURSOR FOR
      SELECT Name, Einwohner
      FROM Stadt
      WHERE Code = :landID;
   EXEC SQL OPEN StadtCursor;  /* Cursor oeffnen */
   printf("Stadt                  Einwohner\n");
   while (1)
   {EXEC SQL FETCH StadtCursor INTO :cityName:ind1 ,
                         :cityEinw INDICATOR :ind2;
    if(ind1 != -1 && ind2 != -1)
    {  /* keine NULLwerte ausgeben */
       printf("%s     %d \n", cityName, cityEinw);
    }};
   EXEC SQL CLOSE StadtCursor; }
```

Hostarrays

- sinnvoll, wenn die Größe der Antwortmenge bekannt ist oder nur ein bestimmter Teil interessiert.
- vereinfacht Programmierung, da häufig auf Cursor verzichtet werden kann.
- verringert zudem Kommunikationsaufwand zwischen Client und Server.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* host array */
   int cityPop[20];          /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
         INTO :cityName, :cityPop
         FROM City
         WHERE Code = 'D';
```

holt 20 Tupel in die beiden Hostarrays.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

PL/SQL

- Oracle Pro*C/C++ Precompiler unterstützt PL/SQL-Blöcke.
- PL/SQL-Block kann anstelle einer SQL-Anweisung verwendet werden.
- PL/SQL-Block verringt Kommunikationsaufwand zwischen Client und. Server
- Übergabe in einem Rahmen:

```
EXEC SQL EXECUTE
DECLARE
  ...
BEGIN
  ...
END;
END-EXEC;
```

Statisches vs. Dynamisches SQL

SQL-Anweisungen können durch Stringoperationen zusammengestellt werden. Zur Übergabe an die Datenbank dienen unterschiedliche Befehle, abhängig von den in der Anweisung auftretenden Variablen.

Transaktionen

- Anwendungsprogramm wird als geschlossene Transaktion behandelt, falls es nicht durch `COMMIT`- oder `ROLLBACK`-Anweisungen unterteilt ist
- In Oracle wird nach Beendigung des Programms automatisch ein `COMMIT` ausgeführt
- DDL-Anweisungen generieren vor und nach ihrer Ausführung implizit ein `COMMIT`
- Verbindung zur Datenbank durch
  `EXEC SQL COMMIT RELEASE;` oder
  `EXEC SQL ROLLBACK RELEASE;`
  beenden.

Savepoints

- Transaktion läßt sich durch Savepoints unterteilen.
- Syntax : `EXEC SQL SAVEPOINT <name>`
- Ein `ROLLBACK` zu einem Savepoint löscht alle danach gesetzten Savepoints.

Mechanismen für Ausnahmebehandlung

- SQL Communications Area (SQLCA)
- `WHENEVER`-Statement

SQLCA

enthält Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung

```
struct sqlca {
   char    sqlcaid[8];
   long    sqlcabc;
   long    sqlcode;
   struct { unsigned short sqlerrml;
            char sqlerrmc[70];
   } sqlerrm;
   char    sqlerrp[8];
   long    sqlerrd[6];
   char    sqlwarn[8];
   char    sqlext[8];
};
```

Interpretation der Komponente `sqlcode`:

- 0: die Verarbeitung einer Anweisung erfolgte ohne Probleme.
- $>$0: die Verarbeitung ist zwar erfolgt, dabei ist jedoch eine Warnung aufgetreten.
- $<$0: es trat ein ernsthafter Fehler auf und die Anweisung konnte nicht ausgeführt werden.

WHENEVER-Statement

spezifiziert Aktionen die im Fehlerfall automatisch vom DBS ausgeführt werden sollen.

```
EXEC SQL WHENEVER <condition> <action>;
```

<condition>

- SQLWARNING : die letzte Anweisung verursachte eine “no data found” verschiedene Warnung (siehe auch sqlwarn). Dies entspricht sqlcode $>$ 0 aber ungleich 1403.
- SQLERROR : die letzte Anweisung verursachte einen (ernsthaften) Fehler. Dies entspricht sqlcode $<$ 0.
- NOT FOUND : SELECT INTO bzw. FETCH liefern keine Antworttupel zurück. Dies entspricht sqlcode 1403.

<action>

- CONTINUE : das Programm fährt mit der nächsten Anweisung fort.
- DO flq proc_name> : Aufruf einer Prozedur (Fehlerroutine); DO break zum Abbruch einer Schleife.
- GOTO <label> : Sprung zu dem angegebenen Label.
- STOP: das Programm wird ohne commit beendet (exit()), stattdessen wird ein rollback ausgeführt.

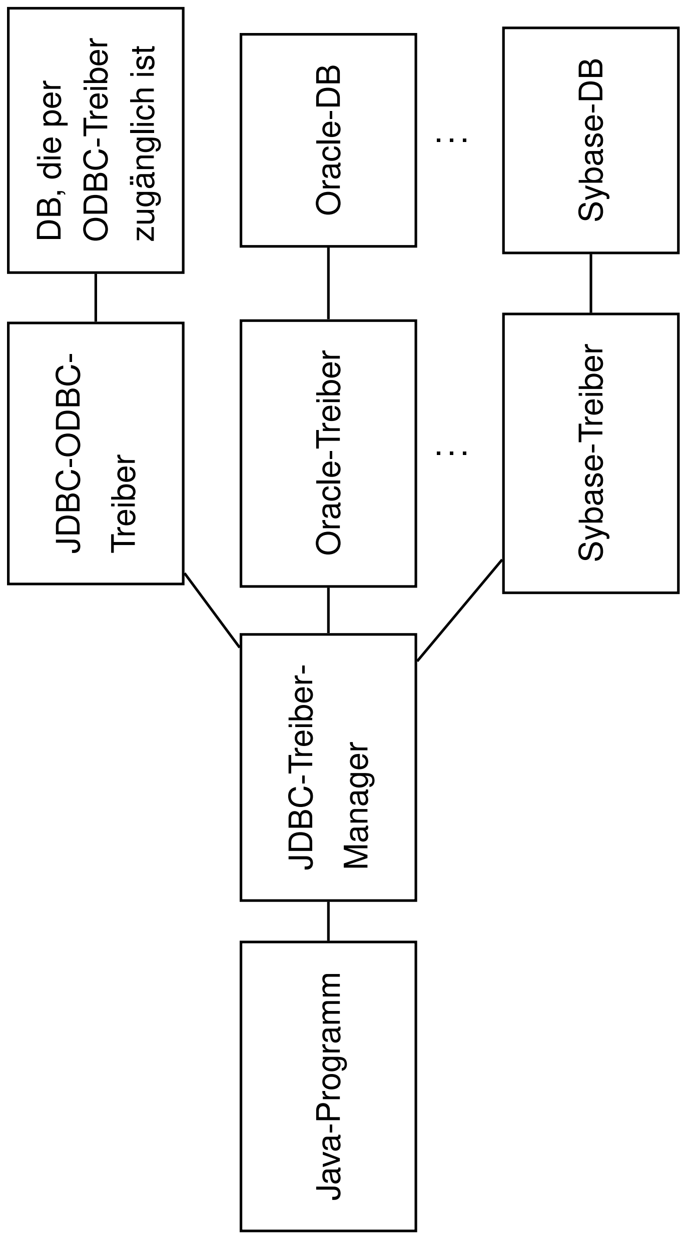
Java und Datenbanken

- Java: plattformunabhängig
- überall, wo *Java Virtual Machine* läuft, können Java-Programme ablaufen.
- API’s: Application Programming Interfaces; Sammlungen von Klassen und Schnittstellen, die eine bestimmte Funktionalität bereitstellen.

JDBC: Datenbankzugriff-API

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus
- Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
- setzt die Idee von ODBC auf Java um
- gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard

JDBC-Architektur



JDBC-Architektur

- Kern: Treiber-Manager
- darunter: Treiber für einzelne DBMS'e

Arten von Treibern:

- Angestrebt:
  - DBMS-Client-Server-Netzwerk-Protokoll mit *pure Java*-Treiber: JDBC-Aufrufe werden in das DBMS-Netzwerkprotokoll umgesetzt. JDBC-Client ruft direkt den DBMS-Server auf.
  - JDBC-Netz mit *pure Java*-Treiber: JDBC-Aufrufe werden in das JDBC-Netzwerkprotokoll übersetzt, und dann auf einem Server in ein bestimmtes DBMS-Protokoll übersetzt.
- als Übergangslösungen:
  - JDBC-ODBC-Bridge und ODBC-Treiber: Über die JDBC-ODBC-Bridge werden die ODBC-Treiber verwendet.
  - Natives API: Die JDBC-Aufrufe werden in Aufrufe von Client-APIs der entsprechenden Datenbankhersteller übersetzt.

JDBC-API

- flexibel:
  - Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
  - de facto: Portabilität nur im SQL-2 Standard gesichert (stored procedures, objektrelationale Features)
- "low-level":
  - Statements werden durch Strings übertragen
  - im Gegensatz zu Embedded SQL keine Verwendung von Programmvariablen in den SQL-Befehlen

Darauf aufbauend in Entwicklung:

- Embedded SQL für Java
- direkte Darstellung von Tabellen und Tupeln in Form von Java-Klassen

JDBC-Funktionalität

- Aufbau einer Verbindung zur Datenbank (`DriverManager`, `Connection`)
- Versenden von SQL-Anweisungen an die Datenbank (`Statement` und Subklassen)
- Verarbeitung der Ergebnismenge (`ResultSet`)

JDBC-Treiber-Manager

DriverManager

- verwaltet (registriert) Treiber
- wählt bei Verbindungswunsch den passenden Treiber aus und stellt Verbindung zur Datenbank her.

- Es wird nur ein DriverManager benötigt.

⇒ Klasse DriverManager:
  - nur static Methoden (operieren auf Klasse)
  - Konstruktor ist private (keine Instanzen erzeugen)

Benötigte Treiber müssen angemeldet werden:

```
DriverManager.registerDriver(driver*)
```

Im Praktikum für den Oracle-Treiber:

```
DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());
```

erzeugt eine neue Oracle-Treiber-Instanz und “gibt” sie dem DriverManager.

Praktikum: Datenbankprogrammierung in SQL/ORACLE

Verbindungsaufbau

- Aufruf an den DriverManager:

```
Connection <name> =
  DriverManager.getConnection
   (<jdbc-url>, <user-id>, <passwd>);
```

- Datenbank wird eindeutig durch JDBC-URL bezeichnet

JDBC-URL:

- jdbc:<subprotocol>:<subname>
- <subprotocol> bezeichnet Treiber und Zugriffsmechanismus
- <subname> bezeichnet Datenbank

Bei uns:

```
jdbc:oracle:<driver-name>:
  @<IP-Address DB Server>:<Port>:<SID>
```

```
String url =
  'jdbc:oracle:thin:@132.230.150.11:1521:dev';
Connection conn =
  DriverManager.getConnection(url,'jdbc_1','jdbc_1');
```

liefert eine offene Verbindungs-Instanz conn zurück.

Verbindung beenden: `conn.close();`

Versenden von SQL-Anweisungen

`Statement`-Objekte

- werden durch Aufruf von Methoden einer bestehenden Verbindung `<connection>` erzeugt.
- `Statement`: einfache SQL-Anweisungen ohne Parameter
- `PreparedStatement`: Vorcompilierte Anfragen, Anfragen mit Parameter
- `CallableStatement`: Aufruf von gespeicherten Prozeduren

Klasse “Statement”

```
Statement <name> = <connection>.createStatement();
```

Sei `<string>` ein SQL-Statement *ohne Semikolon*.

- `ResultSet <statement>.executeQuery(<string>)`:
  Anfragen an die Datenbank. Dabei wird eine Ergebnismenge zurückgegeben.
- `int <statement>.executeUpdate(<string>)`:
  SQL-Statements, die eine Veränderung an der Datenbasis vornehmen. Der Rückgabewert gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren.
- `<statement>.execute(<string>)`: Statement gibt mehr als eine Ergebnismenge zurück (Folge von Anweisungen). Ergebnismengen etc. sind ueber das Statement-Objekt abfragbar (siehe später).

Ein `Statement`-Objekt kann beliebig oft wiederverwendet werden, um SQL-Anweisungen zu übermitteln.

Mit der Methode `close()` kann ein Statement-Objekt geschlossen werden.

Behandlung von Ergebnismengen

Klasse “ResultSet”:

```
ResultSet <name> = <statement>.executeQuery(<string>);
```

- virtuelle Tabelle, auf die von der “Hostsprache” – hier also Java – zugegriffen werden kann.
- ResultSet-Objekt unterhält einen Cursor, der mit

  ```
  <result-set>.next();
  ```

  auf das nächste Tupel gesetzt wird.
- <result-set>.next()
  liefert den Wert false wenn alle Tupel gelesen wurden.

```
ResultSet countries =
  stmt.executeQuery(``SELECT Name, Code, Population
                     FROM Country'');
```

Name	**code**	**Population**
Germany	D	83536115
Sweden	S	8900954
Canada	CDN	28820671
Poland	PL	38642565
Bolivia	BOL	7165257
..	..	..

Behandlung von Ergebnismengen

- Zugriff auf die einzelnen Spalten des Tupels unter dem Cursor mit

  ```
  <result-set>.get<type>(<attribute>)
  ```

- <type> ist dabei ein Java-Datentyp,

Java-Typ	get-Methode
INTEGER	getInt
REAL, FLOAT	getFloat
BIT	getBoolean
CHAR, VARCHAR	getString
DATE	getDate
TIME	getTime

  <getString> funktioniert immer.

- <attribute> kann entweder durch Attributnamen, oder durch die Spaltennummer gegeben sein.

  ```
  countries.getString(``Code'');
  countries.getInt(``Population'');
  countries.getInt(3);
  ```

- Bei get<type> werden die Daten des Ergebnistupels (SQL-Datentypen) in Java-Typen konvertiert.

Behandlung von Ergebnismengen

```
class Hello {
public static void main (String args []) throws SQLException {
  // ORACLE-Treiber laden
  DriverManager.registerDriver(new oracle.jdbc.dnlddriver.OracleDriver());
  // Verbindung zur Datenbank herstellen
  String url = "jdbc:oracle:dnldthin:@132.230.150.161:1521:test";
  Connection conn = DriverManager.getConnection(url,:username,:passwd);
  // Anfrage an die Datenbank
  Statement stmt = conn.createStatement();
  ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City");
  while (rset.next ()) { // Verarbeitung der Ergebnismenge
   String s = rset.getString(1);
   int i = rset.getInt("Population");
   System.out.println (s + " " + i "\ n");
   }
  }
}
```


Arbeiten mit Ergebnismengen

JDBC-Datentypen

- JDBC steht zwischen Java (Objekttypen) und SQL (Typen mit unterschiedlichen Namen).
- `java.sql.types` definiert *generische* SQL-Typen, mit denen JDBC arbeitet:

Java-Typ	JDBC-SQL-Typ
String	CHAR, VARCHAR
java.math.BigDecimal	NUMBER, NUMERIC, DECIMAL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	FLOAT, DOUBLE
java.sql.Date	DATE (Tag, Monat, Jahr)
java.sql.Time	TIME (Stunde, Minute, Sekunde)

Diese werden auch verwendet, um Meta-Daten zu verarbeiten.

Behandlung von Ergebnismengen

Informationen über die Spalten der Ergebnismenge:

```
ResultSetMetaData <name> = <result-set>.getMetaData();
```

erzeugt ein `ResultSetMetaData`-Objekt, das Informationen über die Ergebnismenge enthält:

Methode	Beschreibung
int getColumnCount()	Spaltenanzahl der Ergebnismenge
String getColumnLabel(int)	Attributname der Spalte <int>
String getTableName(int)	Tabellenname der Spalte <int>
String getSchemaName(int)	Schemaname der Spalte <int>
int getColumnType(int)	JDBC-Typ der Spalte <int>
String getColumnTypeName(int)	Unterliegender DBMS-Typ der Spalte <int>

Behandlung von Ergebnismengen

- keine `NULL`-Werte in Java:

  `<resultSet>.wasNULL()`

  testet, ob der zuletzt gelesene Spaltenwert `NULL` war.

Beispiel: Ausgabe der aktuellen Zeile eines ResultSets

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
for(i=1; i<=numCols; i++) {
String returnValue = rset.getString(i);
if (rset.wasNull())
    System.out.println ("null");
else
    System.out.println (returnValue);
}
```

- Mit der Methode `close()` kann ein ResultSet-Objekt explizit geschlossen werden.

Prepared Statements

```
PreparedStatement <name> =
  <connection>.prepareStatement(<string>);
```

- SQL-Anweisung `<string>` wird vorcompiliert.
- damit ist die Anweisung fest im Objektzustand enthalten
- effizienter als `Statement`, wenn ein SQL-Statement häufig ausgeführt werden soll.
- Abhängig von `<string>` ist nur eine der (parameterlosen!) Methoden
  - `<prepared-statement>.executeQuery()`,
  - `<prepared-statement>.executeUpdate()` oder
  - `<prepared-statement>.execute()`

  anwendbar.

Prepared Statements: Parameter

- Eingabeparameter werden durch “?” repräsentiert

```
PreparedStatement pstmt =
  conn.prepareStatement("SELECT Population
                         FROM Country
                         WHERE Code = ?");
```

- “?”-Parameter werden mit

  `<prepared-statement>.set<type>(<pos>,<value>);`

  gesetzt, bevor ein `PreparedStatement` ausgeführt wird.
- `<type>`: Java-Datentyp,
- `<pos>`: Position des zu setzenden Parameters,
- `<value>`: Wert.

```
pstmt.setString(1,"D");
ResultSet rset = pstmt.ExecuteQuery();
...
pstmt.setString(1,"CH");
ResultSet rset = pstmt.ExecuteQuery();
...
```

- Nullwerte werden durch

  `setNULL(<pos>,<type>);`

  gesetzt, wobei `<type>` den JDBC-Typ dieser Spalte bezeichnet:

  `pstmt.setNULL(1,Types.String);`

Callable Statements: Gespeicherte Prozeduren

- Erzeugen von Prozeduren und Funktionen mit

  `<statement>.executeUpdate(<string>);`

  (`<string>` von der Form `CREATE PROCEDURE ...`)

```
s = 'CREATE PROCEDURE bla() IS BEGIN ... END';
stmt.executeUpdate(s);
```

- der *Aufruf der Prozedur* wird als `CallableStatement`-Objekt erzeugt:
- Aufrufsyntax von Prozeduren bei den verschiedenen Datenbanksystemen unterschiedlich

⇒ JDBC verwendet eine *generische* Syntax per Escape-Sequenz (Umsetzung dann durch Treiber)

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>}");
cstmt = conn.prepareCall("{call bla()}");
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Callable Statements mit Parametern

```
s = 'CREATE FUNCTION
            distance(city1 IN Name, city2 IN Name)
      RETURN NUMBER IS BEGIN ... END';
stmt.executeUpdate(s);
```

- Parameter:

```
CallableStatement <name> =
  <connection>.prepareCall("{call <procedure>(?,...,?)}");
```

- Rückgabewert bei Funktionen:

```
CallableStatement <name> =
  <connection>.prepareCall
      ("{? = call <procedure>(?,...,?)}");
cstmt = conn.prepareCall("{? = call distance(?,?)}");
```

- Für `OUT`-Parameter sowie den Rückgabewert muß zuerst der JDBC-Datentyp der Parameter mit

  ```
  <callable-statement>.registerOutParameter
        (<pos>,java.sql.Types.<type>);
  ```

  registriert werden.

  ```
  cstmt.registerOutParameter(1,java.sql.types.number);
  ```

Callable Statements mit Parametern

- Vorbereitung (s.o.)

```
cstmt = conn.prepareCall("{? = call distance(?,?)}");
cstmt.registerOutParameter(1,java.sql.types.number);
```

- `IN`-Parameter werden über `set<type>` gesetzt.

```
cstmt.setString(2,'Freiburg');
cstmt.setString(3,'Berlin');
```

- Aufruf mit

```
ResultSet <name> =
  <callable-statement>.executeQuery();
```

oder

```
<callable-statement>.executeUpdate();
```

oder

```
<callable-statement>.execute();
```

```
Im Beispiel:   cstmt.execute();
```

- Lesen des `OUT`-Parameters mit `get<type>`:

```
int distance = cstmt.getInt(1);
```


Praktikum: Datenbankprogrammierung in SQL/ORACLE

Sequentielle Ausführung

- SQL-Statements, die mehrere Ergebnismengen zurückliefern:
- `<statement>.execute(<string>)`, `<prepared-statement>.execute()`, `<callable-statement>.execute()`
- Häufig: `<string>` dynamisch generiert
- `getResultSet()` bzw. `getUpdateCount()`: nächsten Rückgabewert bzw. Update-Zähler abrufen.
- `getMoreResults()` und dann wieder `getResultSet()` bzw. `getUpdateCount()`: nächstes Ergebnis abrufen.

Sequentielle Ausführung

- `getResultSet()`: Falls nächstes Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis ein Update-Zähler ist: `null` zurückgeben.
- `getUpdateCount()`: Falls nächste Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein nächstes Ergebnis mehr vorhanden, oder nächstes Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.
- `getMoreResults()`: `true`, wenn das nächste Ergebnis eine Ergebnismenge ist, `false`, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse.
- alle Ergebnisse verarbeitet:

```
((<stmt>.getResultSet() == null) &&
 (<stmt>.getUpdateCount() == -1))
```

bzw.

```
((<stmt>.getMoreResults() == false) &&
 (<stmt>.getUpdateCount() == -1))
```


Folge von Ergebnissen verarbeiten

```
stmt.execute(queryStringWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) {
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue;
    }
    if (rowCount == 0) {
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue;
    }
    ResultSet rs = stmt.getResultSet();
    if (rs != null) {
       .....  // verarbeite Metadaten
       while (rs.next())
          { ....}  // verarbeite Ergebnismenge
       stmt.getMoreResults();
       continue;
    }
    break;
}
```

Weitere SQL/Oracle-Werkzeuge

- Dynamic SQL: SQL-Statements können *in PL-SQL* bei Laufzeit als Strings zusammengesetzt und an die Datenbank übermittelt werden.
- ORACLE8i: Mit eingebauter Java Virtual Machine, Zugriff auf das Filesystem,
  i= internet: XML-Schnittstelle, Web-Application-Server etc.
- ORACLE-Web Server/Internet Application Server (9i): HTML-Seiten werden mit abhängig vom Datenbankinhalt erstellt.
- mit den neuesten Paketen (IAS, Internet File System Server) verwischt der Unterschied zwischen Datenbank und Betriebssystem.

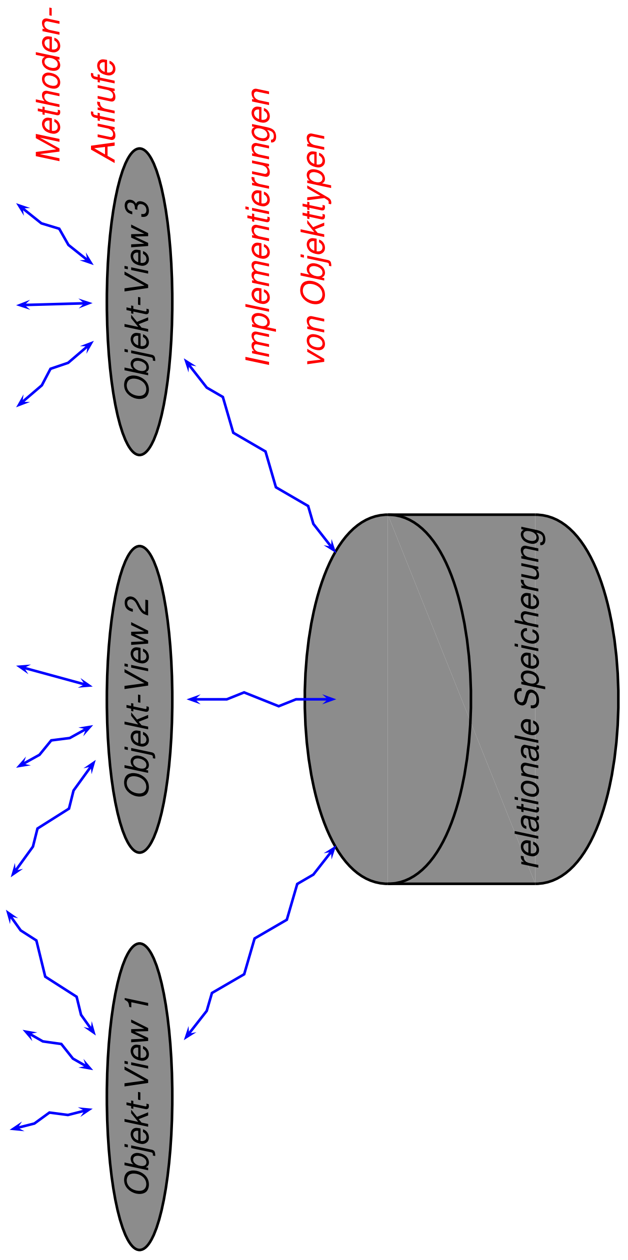
Praktikum: Datenbankprogrammierung in SQL/ORACLE

JDBC 268

ORACLE8 ?

+ *Komplexe Datentypen*
+ *Objekte: Objektmethoden, Objektreferenzen, Pfadausdrücke ⇒ sehr benutzerfreundliches Interface möglich*
  (vgl. add_member, is_member*)*
– *Nested Tables:*
  - *Speicherung: als separate Tabellen (`STORE AS ...`)*
  - *DML: umständlich `SELECT FROM THE`, `TABLE ...`, `CAST MULTISET`*
  - *Handhabung: Anfrage darf nur eine Nested Table betrachten ⇒ Cursor notwendig*
  - *Eigentlich keine Vorteile ??*
– *Veränderungen an Objekttypen nicht möglich ⇒ Objekttypen zur* Speicherung *wenig geeignet.*
- I think this is the power of the system. Object Views.

Datenbank-Architektur



- *Änderungen an der relationalen Speicherung: leicht möglich. Objekttyp-Implementierungen können angepaßt werden, ohne das Benutzerinterface zu verändern.*
- *Änderungen an den Objekttypen: unabhängig von Speicherung (Views), man kann also die Objekttypen löschen und neu erstellen, ohne die Daten zu verlieren. Interface inkrementell ändern.*
- *Hinzufügen von Funktionalität: Geeignete Objekttypen/Methoden/Views neu definieren.*