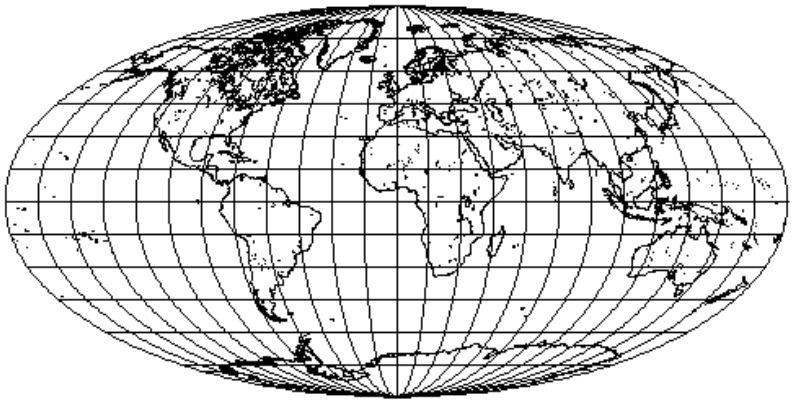
Praktikum: Datenbankprogrammierung in SQL/ORACLE



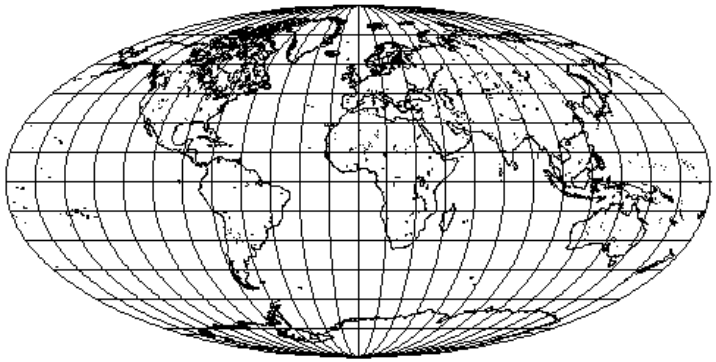
Prof. Dr. Wolfgang May
Universität Göttingen

Mit Beiträgen von Erik Behrends, Rainer Himmeröder, Marco Koch, Heiko Oberdiek.

INHALT: SQL-3 STANDARD/ORACLE

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

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
- Grundidee und Teile der TERRA-Datenbasis des Instituts für Programmstrukturen und, Datenorganisation der Universität Karlsruhe,
- . . . einige weitere WWW-Seiten,
- Datenintegration mit *FLORID* in Freiburg/1998.
- Ergänzungen in 2009.

TEIL I: Grundlagen

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

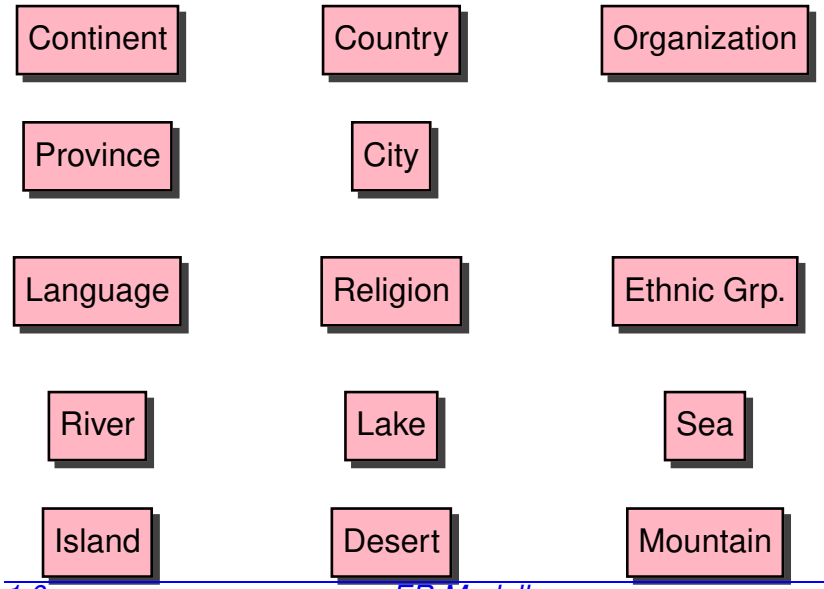
Teil III: Erweiterungen
Prozedurale Konzepte, OO, Einbettung

Kapitel 1
Semantische Modellierung

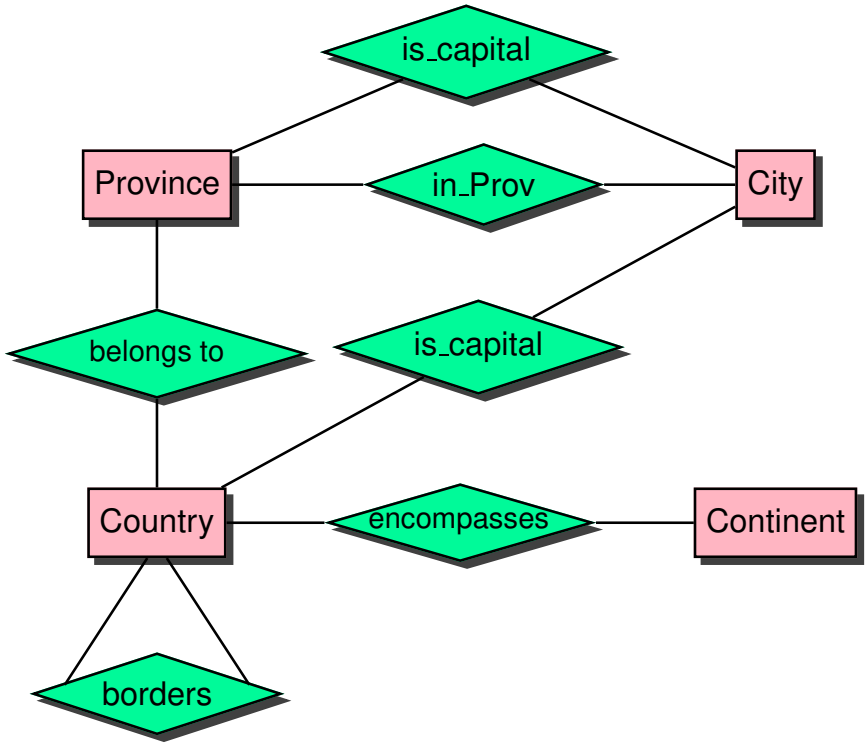
ENTITY-RELATIONSHIP-MODELL (CHEN, 1976)

Strukturierungskonzepte zur Beschreibung eines Schemas im ERM:

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



ENTITIES UND BEZIEHUNGEN



ENTITIES

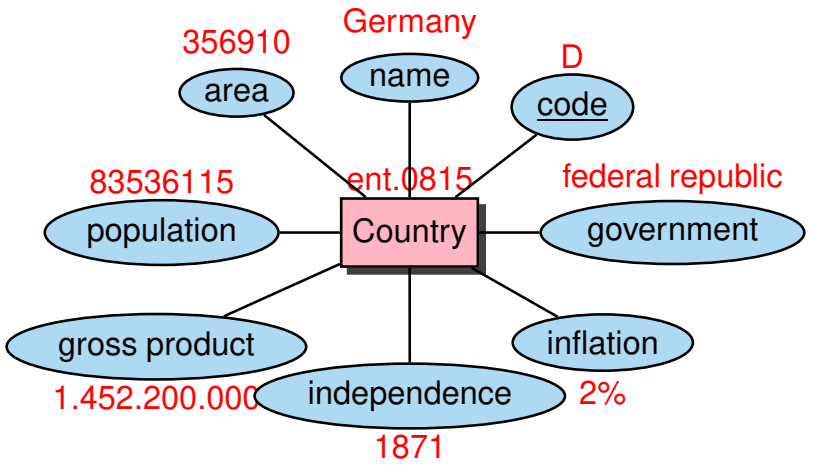
Entitätstyp ist durch ein Paar $(E, \{A_1, \ldots, A_n\})$ gegeben, wobei E der Name und $\{A_1, \ldots, A_n\},\ n \geq 0$, die Menge der Attribute des Typs ist.

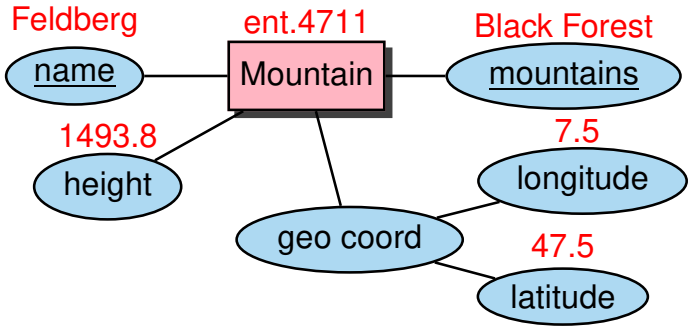
Attribut: Relevante Eigenschaft der Entitäten eines Typs. Jedes Attribut kann *Werte* aus einem bestimmten *Wertebereich (domain)* annehmen.

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

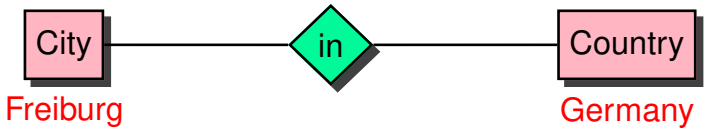
Beziehungstyp: Menge gleichartiger Beziehungen zwischen Entitäten; ein Beziehungstyp ist durch ein Tripel $(B, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_n\})$ gegeben, wobei B der Name, $\{RO_1, \ldots, RO_k\},\ k \geq 2$, die Menge der sog. Rollen, $\{E_1, \ldots, E_k\}$ die den Rollen zugeordnete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

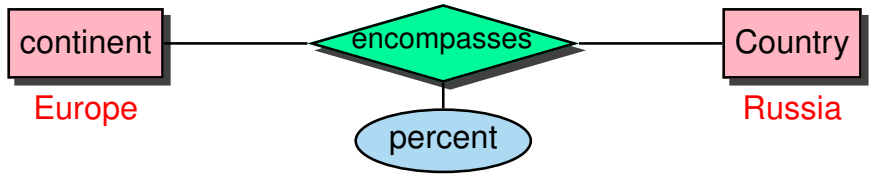
Attribut: Relevante Eigenschaft der Beziehungen eines Typs.

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.

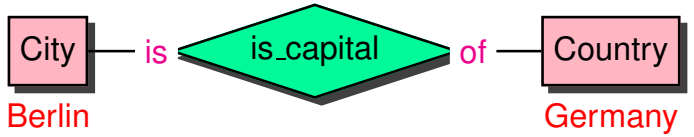
BEZIEHUNGEN



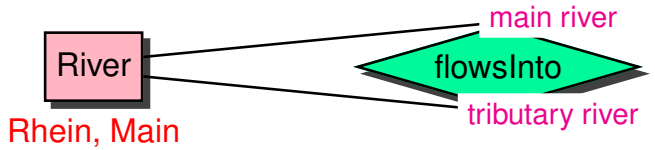
attributierte Beziehung



Beziehung mit Rollen



rekursive Beziehung



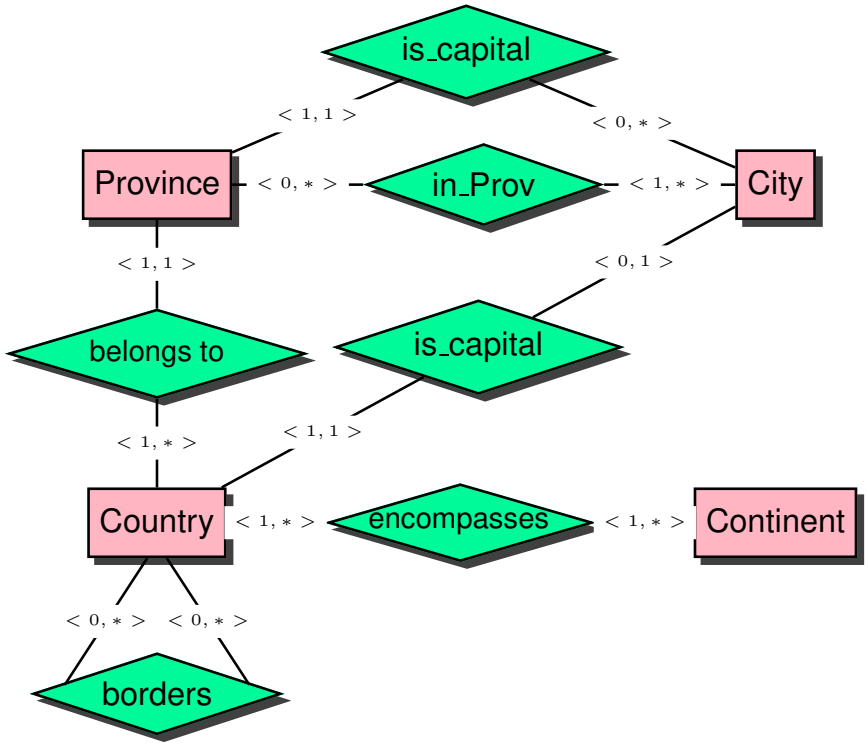
BEZIEHUNGSKOMPLEXITÄTEN

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

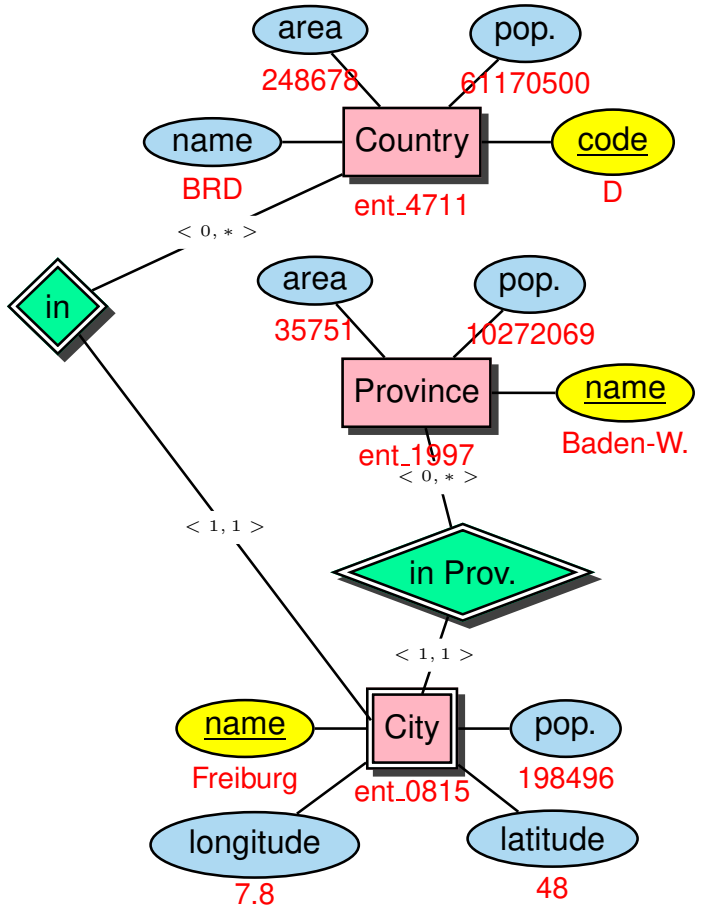


SCHWACHE ENTITÄTSTYPEN

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**).

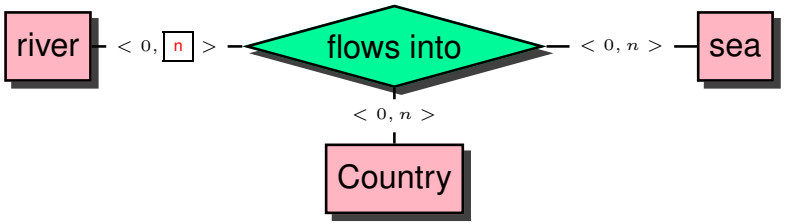
SCHWACHE ENTITÄTSTYPEN



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

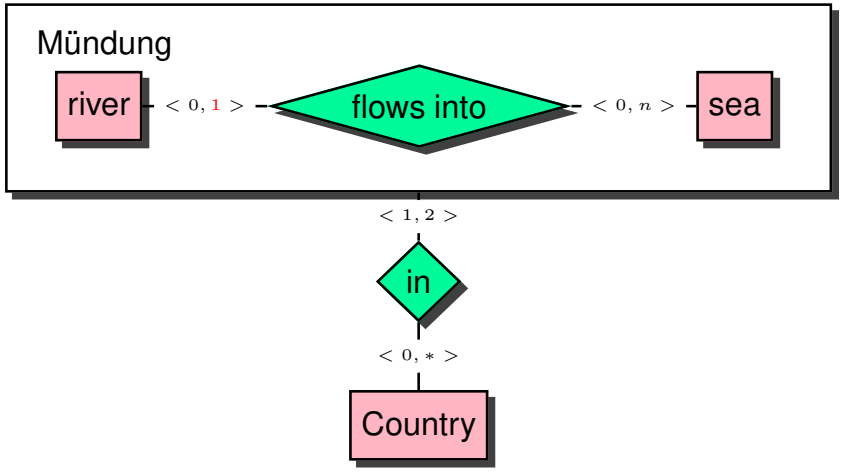
MEHRSTELLIGE BEZIEHUNGEN

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



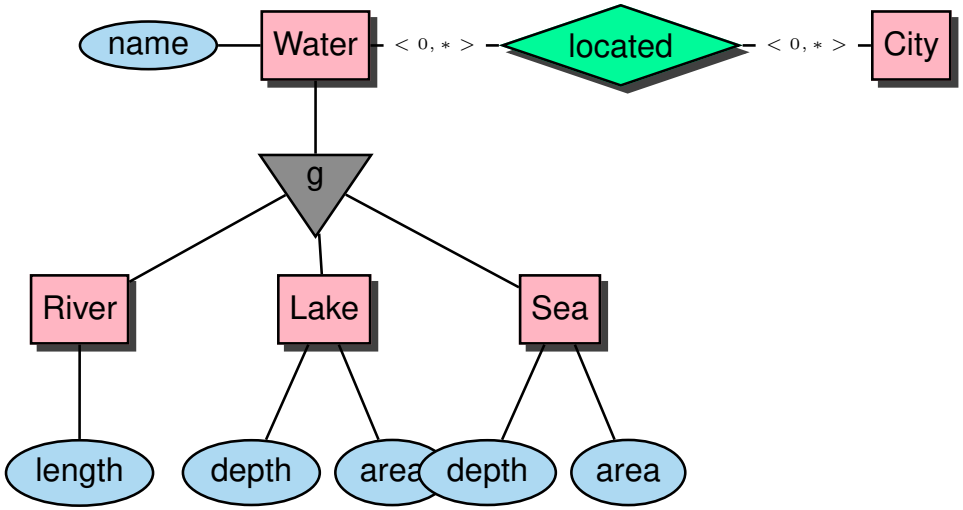
AGGREGATION

Sinnvoll, einen *Aggregattyp Mündung* einzuführen:



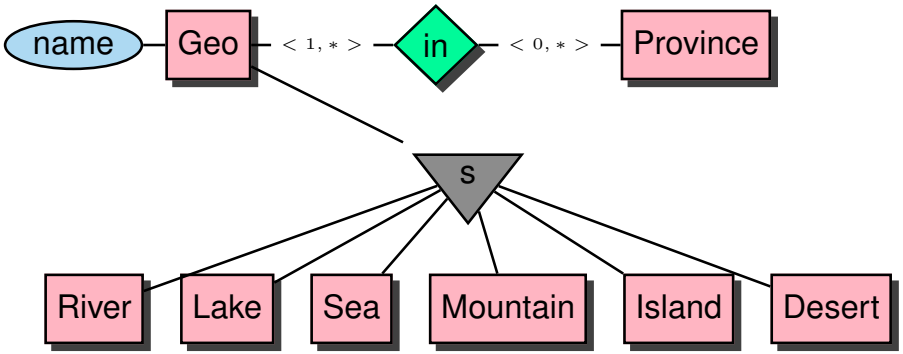
GENERALISIERUNG/SPEZIALISIERUNG

- 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:



Kapitel 2
Das Relationale Modell

- nur ein einziges Strukturierungskonzept *Relation* für Entitytypen *und* Beziehungstypen,
- Relationenmodell von Codd (1970): mathematisch fundierte Grundlage: Mengentheorie

DAS RELATIONALE MODELL

- 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: VARCHAR2(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.

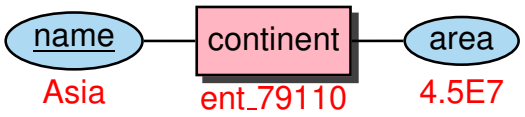
ABBILDUNG ERM IN RM

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

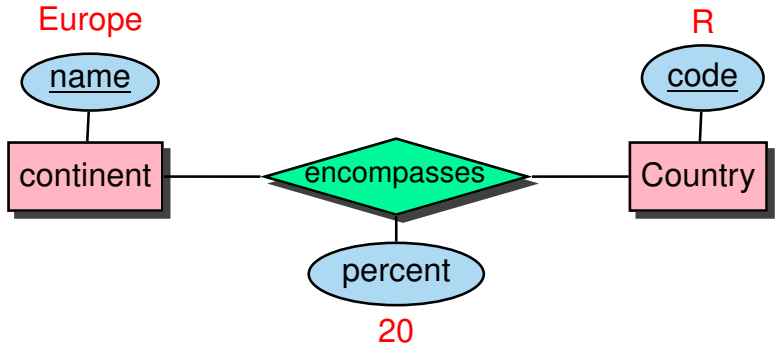
$(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$



Continent	
Name	**Area**
VARCHAR2(20)	NUMBER
Europe	9562489.6
Africa	3.02547e+07
Asia	4.50953e+07
America	3.9872e+07
Australia	8503474.56

BEZIEHUNGSTYPEN

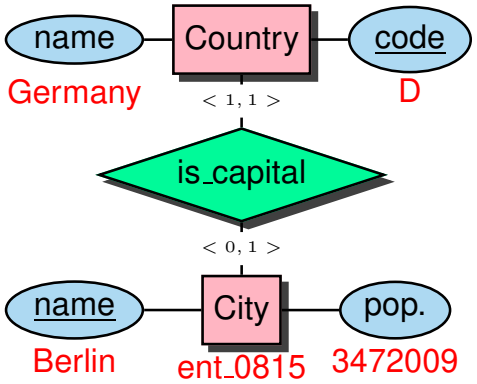
$(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
$B(E_1_K_{11}, \ldots, E_1_K_{1p_1}, \ldots,$
$E_k_K_{k1}, \ldots, E_k_K_{kp_k}, A_1, \ldots, A_m),$
wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
(man darf aber umbenennen, z.B. *Country* für *Country.Code*)



encompasses		
Country	**Continent**	**Percent**
VARCHAR2(4)	VARCHAR2(20)	NUMBER
R	Europe	20
R	Asia	80
D	Europe	100
...	...	...

BEZIEHUNGSTYPEN

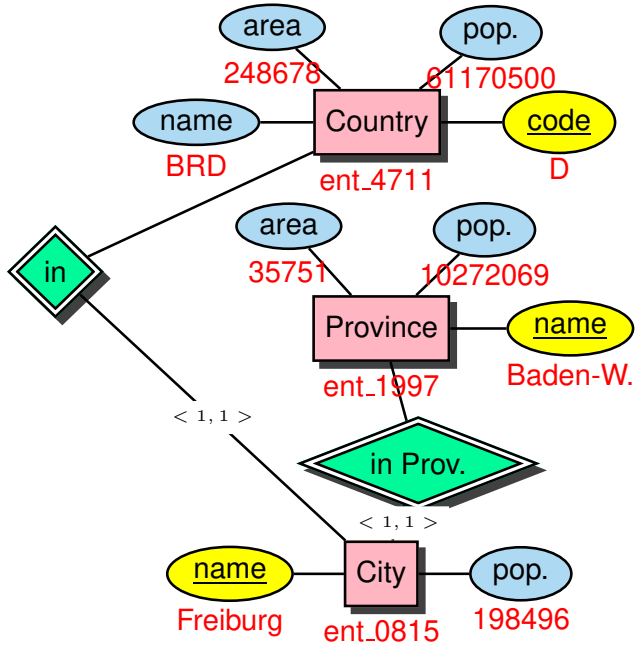
Für zweistellige Beziehungstypen können im Falle einer (1,1)-Beziehungskomplexität das Relationsschema des Beziehungstyps und das Schema des Entitätstyps zusammengefasst werden:



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

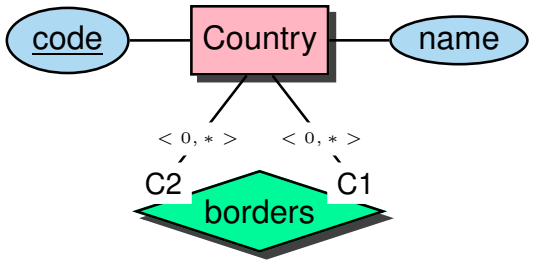
Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.



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

BEZIEHUNGSTYPEN

Falls B_{ER} Rollenbezeichnungen enthält, so werden diese als Name der entsprechenden (Fremdschlüssel)attribute gewählt:



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

Kapitel 3
SQL = Structured Query Language

- Standard-Anfragesprache
- Standardisierung:
  SQL-89, SQL-92 (SQL2), SQL:1999 (SQL3), SQL:2003
- SQL2 in 3 Stufen eingeführt (entry, intermediate und full level).
- SQL3: Objektorientierung
- SQL:2003: XML
- deskriptive Anfragesprache
- Ergebnisse immer Mengen von Tupeln (Relationen)
- Implementierungen: ORACLE (im Praktikum), IBM DB2, Microsoft SQL Server, PostgreSQL, MySQL, etc.

AUFBAU

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: Enthält *Metadaten* über die Datenbank. (in Tabellen; Anfragen daran werden auch mit der DML gestellt)

... inzwischen gehen SQL-Systeme weit über diese Dinge hinaus.

3.1 Data Dictionary

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

⇒ Wenn man sich in eine unbekannte Datenbank einarbeiten soll, oder zusätzlich zur Doku weitere Informationen benötigt, wird man hier fündig.

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`, `TABS` für `USER_TABLES`.

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

Jede der Tabellen besitzt mehrere Spalten, die spezifische Informationen über die jeweiligen Objekte enthalten.


```
SELECT table_name FROM tabs;
```

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

33 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(35)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(35)
POPULATION		NUMBER
LONGITUDE		NUMBER
LATITUDE		NUMBER

3.2 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.

ALLGEMEINE SYNTAKTISCHE HINWEISE

- SQL ist case-insensitive, d.h. CITY=city=City=cItY.
  (Ausnahmen siehe Folie 75)
- Innerhalb von Quotes ist SQL nicht case-insensitive, d.h. City='Berlin' $\neq$ City='berlin'.
- String-Konstanten in der WHERE-Klausel werden in einfache Anführungszeichen eingeschlossen, nicht in doppelte.
  (doppelte Anführungszeichen machen etwas anderes, siehe Folie 75)
- Jeder Befehl wird mit einem Strichpunkt ";" abgeschlossen.
- Kommentarzeilen werden in $/* \ldots */$ eingeschlossen, oder mit `--` oder `rem` eingeleitet.

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
..	..

DISTINCT

```
SELECT * FROM Island;
```

Name	**Islands**	**Area**	...
⋮	⋮	⋮	⋮
Jersey	Channel Islands	117	...
Mull	Inner Hebrides	910	...
Montserrat	Lesser Antilles	102	...
Grenada	Lesser Antilles	344	...
⋮	⋮	⋮	⋮

```
SELECT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Lesser Antilles
Lesser Antilles
⋮

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

Islands
⋮
Channel Islands
Inner Hebrides
Lesser 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`, ...

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 genau ein beliebiges Zeichen repräsentieren und Prozentzeichen null bis beliebig viele Zeichen darstellen,
- <attribute> `IN` <value-list>, wobei <value-list> entweder von der Form ('val$_1$',...,'val$_n$') 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
⋮	⋮	⋮

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 Städte.

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

Count(*)
3064

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

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

Count(DISTINCT(Country))
68

AGGREGATFUNKTIONEN

Beispiel: Ermittle die Gesamtsumme aller Einwohner von Städten Österreichs sowie die 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 als 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
⋮

3.3 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	Population
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;
```

Continent	Abbreviation
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
⋮	⋮	⋮

3.4 Subqueries

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

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

```
SELECT <attr-list>
FROM <table>
WHERE <attribute> IN <subquery>;
```

- <subquery> ist eine SELECT-Anfrage (*Subquery*),
- für <op> $\in \{=, <, >, <=, >=\}$ muss <subquery> eine einspaltige Ergebnisrelation liefern, mit deren Werten der Wert von <attribute> verglichen wird.
- für IN <subquery> sind auch mehrspaltige Ergebnisrelationen erlaubt.
- 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*.

... mit einem einzelnen Wert als Ergebnis der Subquery:

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

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

... mit einem mehrzeiligen Ergebnis der Subquery und `IN`:
(meistens werden Mengen von (Fremd)Schlüsseln berechnet)

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
Malta
Seychelles

UNKORRELIERTE SUBQUERY MIT MEHRSPALTIGEM IN

(mehrspaltige (Fremd)Schlüssel)

Beispiel: Alle Städte, von denen bekannt ist, dass sie an einem Gewässer 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
⋮	⋮	⋮

Alternative:

```
... WHERE Area < (SELECT min(area) FROM ...)
```


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
France
Spain
⋮

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;
```

Hinweis: Diese Äquivalenzumformung ist so nur für nicht-negiertes EXISTS möglich.

SUBQUERIES MIT NOT EXISTS

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

```
SELECT Name
FROM Country
WHERE NOT EXISTS
      ( SELECT *
        FROM City
        WHERE Population > 1000000
        AND City.Country = Country.Code) ;
```

Äquivalent ohne Subquery muss mit `MINUS` und einem der obigen gebildet werden

(vgl. Umformungen in relationale Algebra)

SUBQUERIES IN DER FROM-ZEILE

Eine Subquery kann überall auftreten, wo eine Relation/Tabelle stehen kann.

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

Tabellen oder Werte, die auf unterschiedliche Weise zusammengestellt oder berechnet werden, können in Beziehung zueinander gestellt werden.

Hinweis: dies ist die einzige Art, wie Subqueries in der relationalen Algebra existieren.

SUBQUERIES IN DER FROM-ZEILE

- Aliase für die Zwischenergebnis-Tabellen

Beispiel: Gesucht sind alle Paare (Land,Organisation), so dass das Land mehr als 50 Millionen Einwohner hat und in einer Organisation mit mindestens 20 Mitgliedern Mitglied ist.

```
SELECT c.name, org.organization
FROM
  (SELECT Name, Code
   FROM Country
   WHERE Population > 50000000) c,
  isMember,
  (SELECT organization
   FROM isMember
   GROUP BY organization
   HAVING count(*) > 20) org
WHERE c.code = isMember.country
  AND isMember.organization = org.organization;
```


SUBQUERIES IN DER FROM-ZEILE

- 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)
274439623

SUBQUERIES IN DER FROM-ZEILE

- Berechnung von einzelnen Zwischenergebnissen zur Weiterverwendung

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

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

population	urban_residents	relativ
5761875727	1120188570	.194413872

SUBQUERIES IN DER SELECT-ZEILE

... eine Subquery, die einen einzelnen Wert ergibt, kann auch statt einer Konstanten in der SELECT-Zeile stehen:

(die einelementige Dummy-Tabelle “dual” kann man immer nehmen, wenn man eigentlich keine FROM-Zeile benötigen würde)

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

```
SELECT  (SELECT SUM(Population) FROM Country) -
        (SELECT SUM(Population) FROM City)
FROM dual
```

SELECT(...)-SELECT(...)
4641687157

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, isMember
WHERE Organization = 'EU'
AND isMember.Country = Country.Code
AND isMember.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

NULLWERTE

- Wert ist nicht vorhanden, nicht bekannt, nicht definiert,
- Tatsächliche Bedeutung ist anwendungsabhängig,
- Abfrage: WHERE ... IS [NOT] NULL

```
SELECT * FROM City WHERE population IS NULL;
```

- Nullwerte erfüllen keine (Vergleichs)bedingungen (insbesondere auch keine Join-Gleicheitsbedingung):

```
SELECT c1.name, c2.name, c1.population
FROM City c1, City c2
WHERE c1.population = c2.population
  AND c1.name <> c2.name ORDER BY 3;
```

- Nullwerte werden bei ORDER BY als größte Werte angesehen. Mit NULLS LAST|FIRST kann man dies (passend zu ASC|DESC) beeinflussen:

```
SELECT name, population FROM city
ORDER BY population [ASC|DESC] [NULLS LAST|FIRST];
```

Nullwerte (Cont'd)

- Nullwerte werden in Aggregationsoperatoren (SUM, COUNT, ...) ignoriert:

```
SELECT AVG(population) FROM city
WHERE province='Hawaii';
```

- Sonstige Operationen mit NULL ergeben NULL:

```
SELECT 1 + NULL FROM DUAL  => NULL
```

- Mit der Funktion nvl(attr, wert) kann vorgegeben werden, mit was anstelle von NULL gerechnet werden soll:

```
SELECT AVG(nvl(population,0)) FROM city
WHERE province='Hawaii';

SELECT 1 + nvl(NULL,2) FROM DUAL  =>  3
```


SYNTACTIC SUGAR: JOIN

- bisher: SELECT ... FROM ... WHERE <(join-)conditions>
- abkürzend:
  SELECT ... FROM <joined-tables-spec>
  WHERE <conditions>

mit <joined-tables-spec>:

- kartesisches Produkt:

```
SELECT ...
FROM <table_1> CROSS JOIN <table_2>
WHERE ...
```

- natürliches Join (über alle gemeinsamen Spaltennamen):

```
SELECT ...
FROM <table_1> NATURAL JOIN <table_2>
WHERE ...
```

Beispiel: Alle Paare (Fluss, See), die in derselben Provinz liegen:

```
SELECT country, province, river, lake
FROM geo_river NATURAL JOIN geo_lake;
```

geht auch mit mehr als zwei Relationen:

```
SELECT country, province, river, lake, sea
FROM geo_river NATURAL JOIN geo_lake
               NATURAL JOIN geo_sea;
```

Syntactic Sugar: Join (Cont'd)

- inneres Join mit Angabe der Join-Bedingungen:
  SELECT ...
  FROM <table_1> [INNER] JOIN <table_2>
  ON <conditions>
  WHERE <more conditions>

```
SELECT code, y.name
FROM country x JOIN city y
  ON x.capital=y.name AND x.code=y.country AND
     y.province = y.province AND
     x.population < 4 * y.population;
```

  kein wesentlicher Vorteil gegenüber SFW.
  Mehr als zwei Relationen sind hier nicht erlaubt, z.B.
  ... FROM country x JOIN city y JOIN organization z ...

- äußeres Join:
  SELECT ...
  FROM <table_1>
  [LEFT | RIGHT | FULL] OUTER JOIN <table_2>
  ON <conditions>
  WHERE <more conditions>

```
SELECT r.name, l.name
FROM  river r FULL OUTER JOIN lake l
  ON r.lake = l.name;
```

  deutlich kürzer und klarer als SFW mit UNION um das Outer Join zu umschreiben.

REKURSIVE ANFRAGEN: CONNECT BY

- Rekursion/Iteration in der relationalen Algebra nicht möglich
- für transitive Hülle und Durchlaufen von Eltern-Kind-Relationen benötigt

SQL: CONNECT BY

- mehrfaches Join einer Relation mit sich selbst:
  $R \bowtie [\textit{Bedingung}] R \ldots \bowtie [\textit{Bedingung}] R \bowtie [\textit{Bedingung}] R$
- z.B. für R = borders oder R = river[name,river]

```
SELECT ...
FROM <relation>
START WITH <initial-condition>
CONNECT BY [ NOCYCLE ] <recurse-condition>
```

- <relation> kann eine Tabelle, ein View, oder eine Subquery sein,
- <initial-condition> ist eine Bedingung, die das oder die Anfangstupel auswählt,
- <recurse-condition> spezifiziert die Join-Bedingung zwischen Eltern- und Kindtupel, PRIOR, um Bezug zum "Elterntupel" zu nehmen,
- LEVEL: Pseudospalte, die für jedes Tupel die Rekursionsebene angibt

CONNECT BY: BEISPIEL

Transitive Hülle von River mit der Vorschrift:

River R_1 $\bowtie$[R_1.name = R_2.river] River R_2

- Alle Flüsse, die in den Zaire fliessen:

```
SELECT level, name AS Flussname, length
FROM river
START WITH name = 'Zaire'
CONNECT BY   PRIOR name = river;
```

Level	Name	Länge
1	Zaire	4374
:	:	:
2	Kwa	100
3	Cuango	1100
:	:	:
3	Fimi	200
4	Lukenie	900
:	:	:

Das Ergebnis ist eine Relation, die man natürlich auch wieder als Subquery irgendwo einsetzen kann.

Hinweis: hier fehlen Flüsse, die über einen See in den Zaire fliessen (Aufgabe).

Kapitel 4
Schema-Definition

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

ERZEUGEN VON TABELLEN

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

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

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

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

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

weitere Datentypen findet man im Manual.

Andere DBMS verwenden in der Regel andere Namen für dieselben oder ähnliche Datentypen!

TABELLENDEFINITION

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

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

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

Randbemerkung: case-sensitive Spaltennamen

Falls man case-sensitive Spaltennamen benötigt, kann man dies mit doppelten Anführungszeichen erreichen:

```
CREATE TABLE "Bla"
 ("a" NUMBER,
  "A" NUMBER);
desc "Bla";
insert into "Bla" values(1,2);
select "a" from "Bla";    ->  1
select "A" from "Bla";    ->  2
select a from "Bla";      ->  2(!)
```

TABELLENDEFINITION: CONSTRAINTS

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

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

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

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

TABELLENDEFINITION: DEFAULT-WERTE

```
DEFAULT <value>
```

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

```
CREATE TABLE isMember
      ( Country       VARCHAR2(4),
        Organization  VARCHAR2(12),
        Type          VARCHAR2(35)
                      DEFAULT 'member')

INSERT INTO isMember VALUES
   ('CH', 'EU', 'membership applicant');
INSERT INTO isMember (Land, Organization)
   VALUES ('R', 'EU');
```

Country	Organization	Type
CH	EU	membership applicant
R	EU	member
⋮	⋮	⋮

TABELLENDEFINITION: CONSTRAINTS

Zwei Arten von Bedingungen:

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

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

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


TABELLENDEFINITION: BEDINGUNGEN (ÜBERBLICK)

Syntax:

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

Schlüsselwörter in `<bedingung>`:

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

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.

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(35) NOT NULL UNIQUE,
    Code          VARCHAR2(4) PRIMARY KEY);
```

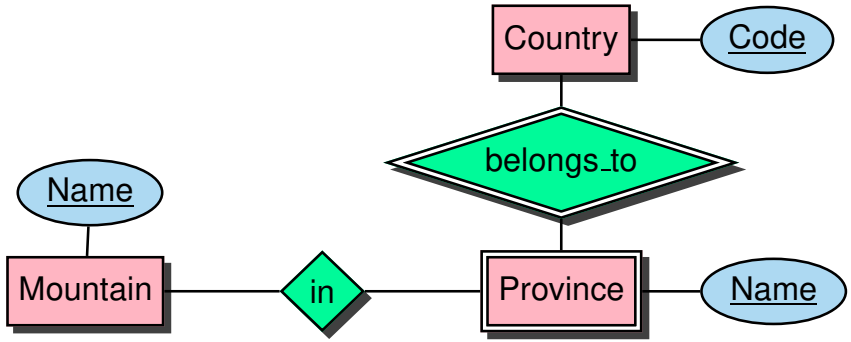

TABELLENDEFINITION: FOREIGN KEY ...REFERENCES

- FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE|ON DELETE SET NULL]: gibt an, dass das Attributtupel <column-list> der Tabelle ein Fremdschlüssel ist und das Attributtupel <column-list2> der Tabelle <table> referenziert.
- Das referenzierte Attributtupel <table>(<column-list2>) muss ein *Candidate Key* von <table> sein.
- Eine REFERENCES-Bedingung wird durch NULL-Werte nicht verletzt.
- ON DELETE CASCADE|ON DELETE SET NULL: Referentielle Aktion (später).

```
CREATE TABLE isMember
  (Country       VARCHAR2(4)
        REFERENCES Country(Code),
   Organization  VARCHAR2(12)
        REFERENCES Organization(Abbreviation),
   Type          VARCHAR2(35) DEFAULT 'member');
```

TABELLENDEFINITION: FREMDSCHLÜSSEL

Ein Berg liegt in einer Provinz eines Landes:




```
CREATE TABLE geo_Mountain
  ( Mountain VARCHAR2(35)
                REFERENCES Mountain(Name),
    Country VARCHAR2(4) ,
    Province VARCHAR2(35) ,
    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(35),      -- + <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: Ein Benutzer benötigt häufig die Information, welche Stadt in welchem Land liegt, ist jedoch weder an Landeskürzeln noch Einwohnerzahlen interessiert.

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

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.
- Eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, kann mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden.
- Mit `DROP TABLE <table> CASCADE CONSTRAINTS` wird eine Tabelle mit allen auf sie zeigenden referentiellen Integritätsbedingungen gelöscht und die referenzierenden Tupel werden entfernt.

ÄNDERN VON TABELLEN UND VIEWS

später.

Kapitel 5
Einfügen und Ändern von Daten

- Einfügen (in existierende Tabellen):
  - Tupel (als Konstanten)
  - Mengen (Ergebnisse von Anfragen)
- Ändern: Einfache Erweiterung des SELECT-FROM-WHERE-Statements.

5.1 Einfügen von Daten

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

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

- Ergebnis einer Anfrage einfügen:

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

- Rest wird ggf. mit Nullwerten aufgefüllt.

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

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

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

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

Es geht auch noch kompakter (implizite Tabellendefinition):

```
CREATE TABLE Metropolis AS
SELECT Name, Country, Population
FROM City WHERE Population > 1000000;
```

5.2 Löschen von Daten

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

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

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

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

```
DELETE FROM City;
```

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

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


5.3 Ändern von Tupeln

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

Beispiel:

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

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

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

5.4 Referentielle Integrität – A First Look

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`-Klausel enthält, erstellt wird, muss `<table>` bereits definiert und `<column-list>` dort ein *Candidate Key* sein.
- Beim Einfügen von Daten müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.
- Beim Löschen oder Verändern eines referenzierten Tupels muss die referentielle Integrität erhalten bleiben.
- Eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, wird mit `DROP TABLE <table> CASCADE CONSTRAINTS` gelöscht.

5.5 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.

Kapitel 6
Spezialisierte Datentypen

- (einfache) Built-In-Typen: Zeitangaben
- Möglichkeit, zusammengesetzte Datentypen selber zu definieren
  (z.B. Geo-Koordinaten aus Länge, Breite) [seit Oracle 8i/1997]
- Verlassen der 1. Normalform: Mengenwertige Einträge – Geschachtelte Tabellen [seit Oracle 8i/8.1.5/1997]
- selbstdefinierte Objekttypen (Siehe Folie 219)
  - Objekte an Stelle von Tupeln und Attributwerten
  - mit Objektmethoden
  - basierend auf PL-SQL [seit Oracle 8.0/1997/1998]
  - mit Java-Methoden [seit Oracle 8i/8.1.5/1999]
  - Objekttypen basierend auf Java-Klassen, Vererbung [seit Oracle 9i/2001]
- Built-In-Typen mit festem Verhalten
  - XMLType (siehe Folie 354) [seit Oracle 9i-2/2002]
  - Ergänzungen durch “DataBlades”, “Extensions” (Spatial Data (seit Oracle 8i/8.1.5) etc.)

6.1 Datums- und Zeitangaben

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

- 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';
```

Country	Independence
MC	01 01 1419
NL	01 01 1579
E	01 01 1492
THA	01 01 1238

Verwendung von Zeitangaben

- SYSDATE liefert das aktuelle Datum.

```
ALTER SESSION SET NLS_DATE_FORMAT = "hh:mi:ss";
SELECT SYSDATE FROM DUAL;
```

SYSDATE
10:50:43

- Funktion

```
EXTRACT (
      { YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
    | { TIMEZONE_HOUR | TIMEZONE_MINUTE }
    | { TIMEZONE_REGION | TIMEZONE_ABBR }
FROM { datevalue | intervalvalue } )
```

Beispiel: Alle Länder, die zwischen 1988 und 1992 gegründet wurden:

```
SELECT Country, EXTRACT(MONTH FROM Independence),
       EXTRACT(YEAR FROM Independence)
FROM Politics
WHERE EXTRACT(YEAR FROM Independence)
      BETWEEN 1988 AND 1992;
```

Country	EXTR...	EXTR...
MK	9	1991
SLO	6	1991
:	:	:

Rechnen mit Datumswerten

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

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

```
SELECT MONTHS_BETWEEN(LAST_DAY(D1), LAST_DAY(D2))
FROM (SELECT independence as D1 FROM politics
        WHERE country='R'),
     (SELECT independence as D2 FROM politics
        WHERE country='UA');
```

MONTHS_BETWEEN(...)
-4

Formattoleranz

- NLS_date_format ist verbindlich für das Ausgabeformat
- für das Eingabeformat wendet Oracle zusaetzlich Heuristiken an:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';
-- die folgenden beiden werden erkannt:
SELECT to_char(to_date('24.12.2002')) FROM dual;
SELECT to_char(to_date('24 JUN 2002')) FROM dual;
-- das wird nicht erkannt:
SELECT to_char(to_date('JUN 24 2002')) FROM dual;
-- ORA-01858: a non-numeric character was found
-- where a numeric was expected

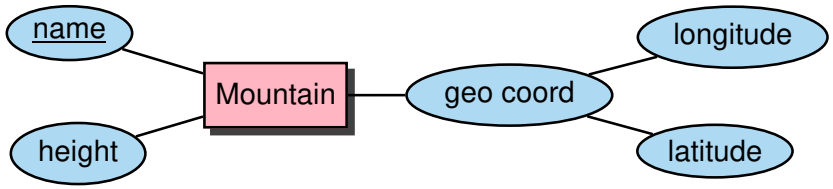
ALTER SESSION SET NLS_DATE_FORMAT = 'MON DD YYYY';
SELECT to_char(to_date('JUN 24 2002')) FROM dual;
```

Explizite Formatvorgabe im Einzelfall

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';
SELECT to_char(to_date('JUN 24 2002','MON DD YYYY'))
FROM dual;
-- 24 06 2002
SELECT to_char(to_date('JUN 24 2002','MON DD YYYY'),
               'MM/DD-YYYY')
FROM dual;
-- 06/24-2002
```


6.2 Zusammengesetzte Datentypen

- "First Normal Form": nur atomare Werte
- Erweiterung I: Strukturierte Werte



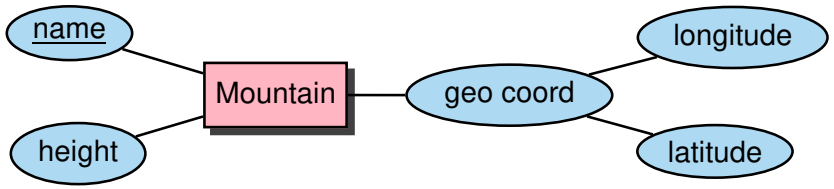
Neue Klasse von Schemaobjekten: `CREATE TYPE`

- `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*).

ZUSAMMENGESETZTE DATENTYPEN

Geographische Koordinaten:




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

CREATE TABLE Mountain
      ( Name          VARCHAR2(35),
        Height        NUMBER,
        Coordinates   GeoCoord);
```

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

```
INSERT INTO Mountain
  VALUES ('Feldberg', 1493, GeoCoord(7.5,47.5));

SELECT * FROM Mountain;
```

Name	Height	Coordinates(Longitude, Latitude)
Feldberg	1493	GeoCoord(7.5,47.5)

ZUSAMMENGESETZTE DATENTYPEN

Zugriff auf einzelne Komponenten von komplexen Attributen in der bei Records üblichen *dot*-Notation.
Hierbei muss der Pfad mit dem Alias einer Relation beginnen (Eindeutigkeit!):

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

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	7.5	47.5

Constraints in zusammengesetzten Datentypen:

```
CREATE TABLE Mountain
  (Name          VARCHAR2(35),
  Height         NUMBER,
  Coordinates    GeoCoord,
CHECK ((Coordinates.Longitude > -180) AND
  (Coordinates.Longitude <= 180)),
CHECK ((Coordinates.Latitude >= -90) AND
  (Coordinates.Latitude <= 90)));
```

6.3 Collections

- “First Normal Form”: nur atomare Werte
- Erweiterung II: Collections:
  Wert eines Attributs ist eine Menge

NestedPolitics			
Country	**Independence**	**Dep.**	**Memberships**
D	18-JAN-1871	NULL	EU, NATO, OECD, ...
GBJ	NULL	GB	$\emptyset$
⋮	⋮	⋮	⋮

- Collection kann durch Aggregation aus einem GROUP-BY gebildet werden:

```
SELECT country, collect(organization)
FROM isMember
GROUP BY country;
```

- Ergebnis z.B. `SYSTPkEqWcRtkgT/gQEyGzFEpmA==('EU', 'NATO', 'OECD', ...)`
- erzeugt ad-hoc einen systemeigenen Typ “`SYSTP...`”, der die Collection aufnimmt.

Tabellen mit Collections erzeugen

Verwendet eine einfache Form des etwas komplexeren Konzeptes “Nested Tables” (siehe Folie 110 ff.)

```
CREATE [OR REPLACE] TYPE <collection_type> AS
  TABLE OF <basic_type>;
/
CREATE TABLE <table_name>
  (... ,
   <collection-attr> <collection_type> ,
   ... )
NESTED TABLE <collection-attr> STORE AS <name >;
```

TABLE-Typ `MON_ORGLIST` definieren:

```
CREATE OR REPLACE
   TYPE MON_ORGLIST AS TABLE OF VARCHAR2(12);
/
CREATE TABLE NestedPolitics
( country VARCHAR2(4) PRIMARY KEY,
  independence DATE,
  dependent VARCHAR2(4), -- REFERENCES Country(Code)
  memberships MON_ORGLIST)
NESTED TABLE memberships STORE AS o_list;
```

Tabellen mit Collections füllen (1)

- explizit unter Verwendung der entsprechenden Konstruktormethode:

```
INSERT INTO NestedPolitics
VALUES('BAV', '01-APR-2010',
       NULL, MON_ORGLIST('EU','OECD'));
INSERT INTO NestedPolitics
VALUES('SYLT', NULL, 'D', MON_ORGLIST());
```

- eine leere Tabelle ist etwas anderes als NULL.
- $\Rightarrow$ damit wird es schwieriger, herauszufinden welche Länder nirgends Mitglied sind!
- man kann keine Bedingungen für die in einer Collection erlaubten Werte formulieren (insb. keine REFERENCES).

Tabellen mit Collections füllen (2)

- collect(...) erzeugt eine Instanz eines ad-hoc-Typs, der Zeichenketten (oder Zahlen oder DATE) enthält,
- man muss (leider) explizit mitteilen, dass diese in den Zieltyp (hier MON_ORGLIST) gecastet werden muss:
  CAST(<instanz-eines-typs> AS <kompatibler typ>)

```
INSERT INTO NestedPolitics
( SELECT p.country, p.independence, p.dependent,
         CAST(collect(i.organization) AS MON_ORGLIST)
  FROM Politics p LEFT OUTER JOIN isMember i
    ON p.country = i.country
  GROUP BY p.country, p.independence, p.dependent);
```

```
SELECT country, memberships
FROM NestedPolitics
WHERE country = 'D';
```

Island	Seas
Cuba	MON_ORGLIST('EU', 'NATO', 'OECD', ...)

- Solche Instanzen können mit "=" verglichen werden

```
SELECT a.country, b.country, a.memberships
FROM NestedPolitics a, NestedPolitics b
WHERE a.country < b.country
  AND a.memberships = b.memberships;
```

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

Tabellen mit Collections anfragen

Mit [THE|TABLE] (<collection-wertiger Wert>) kann man die Collection wie eine Tabelle verwenden.
(THE ist die schon länger gebräuchliche Syntax)

```
SELECT * FROM TABLE(SELECT memberships
                    FROM NestedPolitics
                    WHERE country = 'D');
```

COLUMN_VALUE
EU
NATO
OECD

- Test: mit Konstanten ist nur TABLE, nicht THE erlaubt:

  ```
  SELECT * FROM TABLE(MON_ORGLIST('EU','NATO'));
  ```

- eine Spalte, die nur den Namen COLUMN_VALUE hat,
- oft als `SELECT column_value as <alias>`.
- Hinweis:

  ```
  SELECT * FROM TABLE(SELECT memberships
                      FROM NestedPolitics);
  ```

  ist nicht zulässig, da es ja mehrere Tabellen wären:
  ⇒ `single-row subquery returns more than one row`

Tabellen mit Collections anfragen

Mit TABLE(<attrname>) kann auch innerhalb eines Tupels ein collection-wertiges Attribut als Tabelle zugreifbar gemacht werden:
(hier ist THE nicht erlaubt)

- in Subqueries:

  ```
  SELECT country
  FROM NestedPolitics
  WHERE EXISTS (SELECT *
                FROM TABLE(memberships)
                WHERE column_value = 'NATO');
  ```

- oder auch als *korreliertes Join* in der FROM-Zeile:
  jede umgebende Zeile mit *ihrer* geschachtelten Tabelle joinen und ausmultiplizieren:

  ```
  SELECT country, m.*
         -- oder  m.column_value as membership
  FROM NestedPolitics, TABLE(memberships) m;
  ```

Country	**COLUMN_VALUE** (bzw. **membership**)
D	EU
D	NATO
D	OECD
:	:

Vergleich mit 1:n- bzw. m:n-Beziehungen als separate Tabelle

- Man sieht relativ einfach, dass die nested table o_list ähnlich der bestehenden “flachen” Tabelle isMember gespeichert ist, und dass

```
SELECT p.country, p.independence, im.organization
FROM Politics p, isMember im
WHERE p.country = im.country;

SELECT p.country, p.independence, im.organization
FROM Politics p,
  -- korreliertes Join, waere z.B. in OQL zulaessig
  (SELECT * FROM isMember where country = p.country) im
```

  äquivalent ist.

- Anmerkung:
  korreliertes Join: i-te Relation in Abhängigkeit von $i-1$ter berechnen
  - in SQL nicht erlaubt
  - in Sprachen zu Datenmodellen, die Referenzen/Objektwertige Attribute, mengen-/mehrwertige Attribute oder baumartige Hierarchien besitzen, üblicherweise erlaubt (OQL, XML/XQuery; Forschungs-Sprachen aus 1995-2000: OEM, F-Logic)
  - daher auch für SQL mit Collections naheliegend.

Collection im Ganzen kopieren

```
UPDATE NestedPolitics
SET memberships = (SELECT memberships
                   FROM NestedPolitics
                   WHERE country = 'D')
WHERE country='BAV';
  -- optional THE (SELECT ...)
```

Einfügen, Ändern und Löschen mit THE

- Man kann immer nur eine Collection gleichzeitig anfassen, und muss diese mit einer SELECT-Anfrage auswählen (also nicht 'XXX' in alle Mitgliedschaftslisten einfügen, oder überall 'EU' durch 'EWG' ersetzen)

```
INSERT INTO THE (SELECT memberships
                   FROM NestedPolitics
                   WHERE country = 'D')
  VALUES('XXX');
DELETE FROM THE (SELECT memberships
                  FROM NestedPolitics
                  WHERE country = 'D')
  WHERE column_value = 'XXX';
UPDATE THE (SELECT memberships
            FROM NestedPolitics
            WHERE country = 'D')
  SET column_value = 'XXX'
  WHERE column_value = 'EU';
```

6.4 Geschachtelte Tabellen

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

- Tabellenwertige Attribute
- Generischer Typ `TABLE OF <inner_type>`

⇒ Generische Syntax

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 >;
```

Beispiel

```
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 Lang_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 Lang_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

FÜLLEN VON GESCHACHTELTEN TABELLEN

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

collect() über mehrspaltige Tupel nicht erlaubt

```
-- nicht erlaubt:
INSERT INTO NLanguage
 (SELECT country, collect(name,percentage)
  FROM language
  GROUP BY country)
-- PLS-306: wrong number or types of arguments in
--  call to 'SYS_NT_COLLECT'
```

... also anders: Tupelmenge als Tabelle casten

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

```
INSERT INTO NLanguage -- zulässig, 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) !!

Füllen von Geschachtelten Tabellen

... also erst Tupel erzeugen und dann die geschachtelten Tabellen hinzufügen:

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

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


ARBEITEN MIT GESCHACHTELTE TABELLEN

Mit THE und TABLE wie für Collections beschrieben:

- Kopieren ganzer eingebetteter Tabellen mit

```
INSERT INTO ... VALUES(..., THE(SELECT ...),...);
INSERT INTO ... (SELECT ..., THE (SELECT ...)...);
INSERT INTO THE (...) ...;
DELETE FROM THE (   ) ...;
UPDATE THE (...) ...;
```

- TABLE(<attr>) in Unterabfrage:

```
SELECT Country
FROM NLanguage
WHERE 'German' IN (SELECT name
                   FROM TABLE (Languages));
```

- TABLE(<attr>) als korreliertes Join:

```
SELECT Country, nll.*
FROM NLanguage nl, TABLE(nl.Languages) nll;
```

KOMPLEXE DATENTYPEN

```
SELECT * FROM USER_TYPES
```

Type_name	Type_oid	Typecode	Attrs	Meths
GeoCoord	_	Object	2	0
Language_T	_	Object	2	0
Mon_Orglist	_	Collection	0	0
Languages_List	_	Collection	0	0

Löschen: `DROP TYPE [FORCE]`
Mit `FORCE` kann ein Typ gelöscht werden, dessen Definition von anderen Typen noch gebraucht wird.

Szenario von oben:
`DROP TYPE 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
```


Kapitel 7
TEIL II: Dies und Das

Teil I: Grundlagen

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

Teil II: Weiteres zum “normalen” SQL

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

Teil III: Erweiterungen
Prozedurale Konzepte, OO, Einbettung

7.1 Ändern von Schemaobjekten

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

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

⋮

Ändern von Tabellen

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

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

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

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

- Syntaxvielfalt nützlich wenn die Statements automatisch generiert werden.

Hinzufügen von Spalten zu einer Tabelle

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

Neue Spalten werden mit `NULL`-Werten aufgefüllt.

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

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

Entfernen von Spalten

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


Hinzufügen von Tabellenbedingungen

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

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

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER CHECK (Unemployment >= 0),
       CHECK (Industry + Service + Agriculture <= 102));
```

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

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

SPALTENDEFINITIONEN EINER TABELLE ÄNDERN

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

```
ALTER TABLE Country MODIFY (Capital NOT NULL);
ALTER TABLE encompasses
  ADD (PRIMARY KEY (Country,Continent));
ALTER TABLE Desert
  MODIFY (Area CONSTRAINT DesertArea CHECK (Area > 10));
ALTER TABLE isMember
  MODIFY (type VARCHAR2(10)))   -- change maximal length;
```

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

INTEGRITÄTSBEDINGUNGEN (DE)AKTIVIEREN

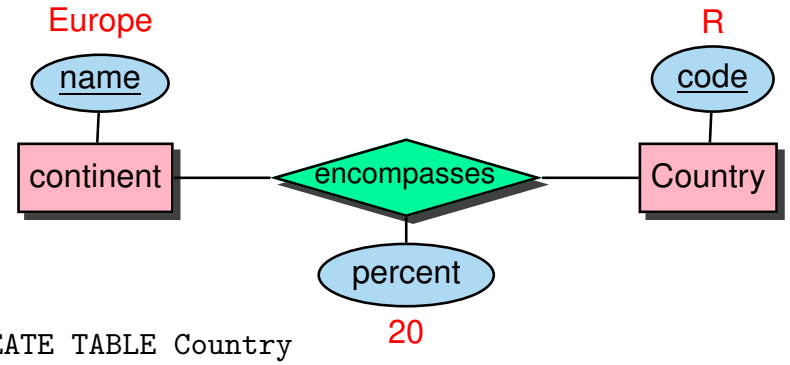
- (Integritäts)bedingungen an eine Tabelle
  - entfernen,
  - zeitweise außer Kraft setzen,
  - wieder aktivieren.

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

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

7.2 Referentielle Integrität

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




```
CREATE TABLE Country
 (Name     VARCHAR2(35),
  Code     VARCHAR2(4) PRIMARY KEY,
  ...);

CREATE TABLE Continent
 (Name     VARCHAR2(20) PRIMARY KEY,
  Area     NUMBER(2));

CREATE TABLE encompasses
 (Continent  VARCHAR2(20) REFERENCES Continent(Name),
  Country    VARCHAR2(4) REFERENCES Country(Code),
  Percentage NUMBER);
```


REFERENTIELLE INTEGRITÄT

Country

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

City

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

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

- (<attr-list'>) muss Candidate Key der referenzierten Tabelle sein (NOT NULL UNIQUE).

REFERENTIELLE INTEGRITÄT

- als Spaltenbedingung:

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

```
CREATE TABLE City
(...
Country VARCHAR2(4)
        CONSTRAINT CityRefsCountry
        REFERENCES Country(Code) );
```

- als Tabellenbedingung:

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

```
CREATE TABLE Country
(...
CONSTRAINT CapitalRefsCity
  FOREIGN KEY (Capital,Code,Province)
  REFERENCES City(Name,Country,Province) );
```


REFERENTIELLE AKTIONEN

- Bei Veränderungen am Inhalt einer Tabelle sollen automatisch Aktionen ausgeführt werden, um die referentielle Integrität der Datenbasis zu erhalten.
- Ist dies nicht möglich, so werden die gewünschten Operationen nicht ausgeführt, bzw. zurückgesetzt.

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

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

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

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

   Anderenfalls ist es unkritisch:

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

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

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

REFERENTIELLE AKTIONEN IM SQL-2-STANDARD

`NO ACTION:`

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

```
DELETE FROM River;
```

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

`RESTRICT:`

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

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

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

`CASCADE:`

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

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

ändert überall:

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

REFERENTIELLE AKTIONEN IM SQL-2-STANDARD

`SET DEFAULT:`

Die Operation wird ausgeführt und bei den referenzierenden Tupeln wird der entsprechende Fremdschlüsselwert auf die für die entsprechende Spalten festgelegten `DEFAULT`-Werte gesetzt (dafür muss dann wiederum ein entsprechendes Tupel in der referenzierten Relation existieren). Falls kein `DEFAULT`-Wert definiert wurde, entspricht das Verhalten `SET NULL` (s.u.).

`SET NULL:`

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

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

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

located(Bremerhaven,Nds.,D,`NULL`,`NULL`,North Sea)

Referentielle Aktionen im SQL-2-Standard

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

`<columnConstraint>` (für einzelne Spalten)

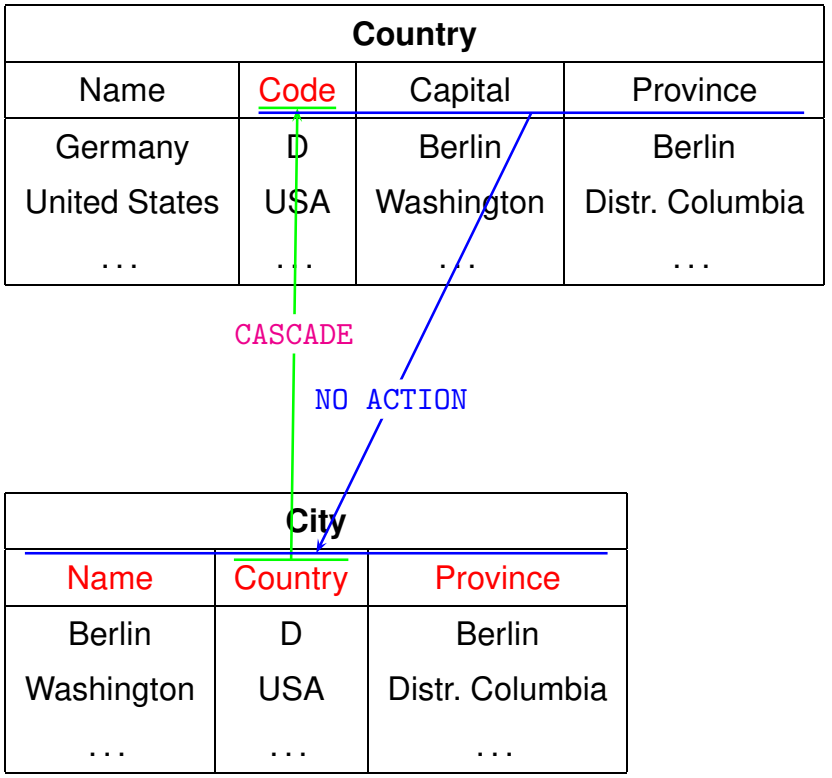
```
<col> <datatype>
  CONSTRAINT <name>
  REFERENCES <table'> (<attr'>)
  [ ON DELETE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
  [ ON UPDATE {NO ACTION | RESTRICT | CASCADE |
              SET DEFAULT | SET NULL } ]
```

oder `<tableConstraint>` (für mehrere Spalten)

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

angegeben.

Referentielle Aktionen




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

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

REFERENTIELLE AKTIONEN IN ORACLE:

- ORACLE 9-11: nur `ON DELETE/UPDATE NO ACTION`, `ON DELETE CASCADE` und (seit Oracle 8.1.5) `ON DELETE SET NULL` implementiert.
- Wird `ON ...` nicht angegeben, wird `NO ACTION` als Default verwendet.
- `ON UPDATE CASCADE` fehlt, was beim Durchführen von Updates ziemlich lästig ist.
- Hat aber so seine Gründe ...

Syntax als `<columnConstraint>`:

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

Syntax als `<tableConstraint>`:

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


REFERENTIELLE AKTIONEN: UPDATE OHNE CASCADE

Beispiel: Umbenennung eines Landes:

```
CREATE TABLE Country
  ( Name  VARCHAR2(35) NOT NULL UNIQUE,
    Code  VARCHAR2(4) PRIMARY KEY);
```

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

```
CREATE TABLE Province
  ( Name    VARCHAR2(35)
    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 ... ;
```

Referentielle Aktionen: Zyklische Referenzen

Country

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

Province

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

City

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

Referentielle Aktionen: Problematik

Country

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

CASCADE

SET NULL

Province

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

CASCADE

City

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

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

REFERENTIELLE AKTIONEN

Im allgemeinen Fall:

- Schon eine einzelne Operation bringt in Verbindung mit `ON DELETE/UPDATE SET NULL/SET DEFAULT` und `ON UPDATE CASCADE` Mehrdeutigkeiten, Widersprüche etc.
- Aufgrund von SQL-Triggern induziert ein User-Update häufig mehrere Datenbank-Updates,
- nichttriviale Entscheidung, welche Updates getriggert werden sollen,
- im Fall von Inkonsistenzen Analyse der Ursache sowie maximal zulässiger Teilmengen der User-Requests,
- Stabile Modelle, exponentieller Aufwand.

... siehe dbis-Webseiten

7.3 View Updates

Views

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

VIEW UPDATES

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

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

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

VIEW UPDATES

- abgeleitete Werte können nicht verändert werden:

Beispiel:

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

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

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

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

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

VIEW UPDATES

Beispiel:

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

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

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

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

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

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

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

VIEW UPDATES

Beispiel:

- *Country* darf nicht verändert werden:

City	Country
Berlin	Germany
Freiburg	Germany

Umsetzung auf Basistabelle wäre nicht eindeutig:

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

Nur in *City* werden die Tupel gelöscht:

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


VIEW UPDATES

- ORACLE: Zulässigkeitsentscheidung durch Heuristiken
- basieren nur auf Schemainformation,
- nicht auf *aktuellem* Datenbankzustand !
- Schlüsseleigenschaften wichtig: Schlüssel einer Basistabelle müssen im View erhalten bleiben.
- Schlüssel einer Basistabelle = Schlüssel des Views: Abbildung möglich.
- Schlüssel einer Basistabelle $\supseteq$ ein Schlüssel des Views: Umsetzung möglich.
  (bei $\subsetneq$ sind eventuell mehrere Tupel der Basistabelle betroffen).
- Schlüssel einer Basistabelle überdeckt keinen Schlüssel des Views: i.a. keine Umsetzung möglich (siehe Aufgaben).
- die Heuristik ist nicht immer so ganz korrekt (siehe Aufgaben).

VIEW UPDATES

Beispiel:

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

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

Country	Name	Population
D	Nordrhein-Westfalen	17816079

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

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

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

Country	Name	Population
D	Bayern	11921944

VIEW UPDATES

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

Beispiel

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

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

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

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

MATERIALIZED VIEWS

- Views werden bei jeder Anfrage neu berechnet.

+ repräsentieren immer den aktuellen Datenbankzustand.

- zeitaufwendig, ineffizient bei wenig veränderlichen Daten

⇒ *Materialized Views*

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

7.4 Zugriffsrechte

BENUTZERIDENTIFIKATION

- Benutzername
- Password
- `sqlplus /`: Identifizierung durch UNIX-Account

ZUGRIFFSRECHTE INNERHALB ORACLE

- Zugriffsrechte an ORACLE-Account gekoppelt
- initial vom DBA vergeben

SCHEMAKONZEPT

- Jedem Benutzer ist sein *Database Schema* zugeordnet, in dem "seine" Objekte liegen.
- Bezeichnung der Tabellen *global* durch
  `<username>.<table>`
  (z.B. `dbis.City`),
- im eigenen Schema nur durch `<table>`.

SYSTEMPRIVILEGIEN

- berechtigen zu Schemaoperationen
- `CREATE [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte erzeugen,
- `ALTER [ANY] TABLE/TYPE/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte verändern,
- `DROP [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte löschen.
- `SELECT/INSERT/UPDATE/DELETE [ANY] TABLE`: Benutzer darf in Tabellen Tupel lesen/erzeugen/verändern/entfernen.
- `ANY`: Operation in *jedem* Schema erlaubt,
- ohne `ANY`: Operation nur im eigenen Schema erlaubt

Praktikum:

- `CREATE SESSION, ALTER SESSION, CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE PROCEDURE...`
- Zugriffe und Veränderungen an den eigenen Tabellen nicht explizit aufgeführt (`SELECT TABLE`).

SYSTEMPRIVILEGIEN

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

- `PUBLIC`: jeder erhält das Recht.
- `ADMIN OPTION`: Empfänger darf dieses Recht weiter vergeben.

Rechte entziehen:

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

nur wenn man dieses Recht selbst vergeben hat (im Fall von `ADMIN OPTION` kaskadierend).

Beispiele:

- `GRANT CREATE ANY INDEX, DROP ANY INDEX TO opti-person WITH ADMIN OPTION;`
  erlaubt `opti-person`, überall Indexe zu erzeugen und zu löschen,
- `GRANT DROP ANY TABLE TO destroyer;`
  `GRANT SELECT ANY TABLE TO supervisor;`
- `REVOKE CREATE TABLE FROM mueller;`

Informationen über Zugriffsrechte im Data Dictionary:

```
SELECT * FROM SESSION_PRIVS;
```

OBJEKTPRIVILEGIEN

berechtigen dazu, Operationen auf existierenden Objekten auszuführen.

- Eigentümer eines Datenbankobjektes
- Niemand sonst darf mit einem solchen Objekt arbeiten, außer
- Eigentümer (oder DBA) erteilt explizit entsprechende Rechte:

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

- `<object>`: TABLE, VIEW, PROCEDURE/FUNCTION, TYPE,
- Tabellen und Views: Genauere Einschränkung für INSERT, REFERENCES und UPDATE durch `<column-list>`,
- `<privilege-list>`: DELETE, INSERT, SELECT, UPDATE für Tabellen und Views,
  INDEX, ALTER und REFERENCES für Tabellen,
  EXECUTE für Prozeduren, Funktionen und TYPEn.
- ALL: alle Privilegien die man an dem beschriebenen Objekt (ggf. auf der beschriebenen Spalte) hat.
- GRANT OPTION: Der Empfänger darf das Recht weitergeben.

OBJEKTPRIVILEGIEN

Rechte entziehen:

```
REVOKE <privilege-list> | ALL
ON <object>
FROM <user-list> | PUBLIC
[CASCADE CONSTRAINTS];
```

- CASCADE CONSTRAINTS (bei REFERENCES): alle referentiellen Integritätsbedingungen, die auf einem entzogenen REFERENCES-Privileg beruhen, fallen weg.
- Berechtigung von mehreren Benutzern erhalten: Fällt mit dem letzten REVOKE weg.
- im Fall von GRANT OPTION kaskadierend.

Überblick über vergebene/erhaltene Rechte:

```
SELECT * FROM USER_TAB_PRIVS;
```

- Rechte, die man für eigene Tabellen vergeben hat,
- Rechte, die man für fremde Tabellen bekommen hat

```
SELECT * FROM USER_COL_PRIVS;
SELECT * FROM USER_TAB/COL_PRIVS_MADE/RECD;
```

Stichwort: Rollenkonzept

SYNONYME

Schemaobjekt unter einem anderen Namen als ursprünglich abgespeichert ansprechen:

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

- Ohne `PUBLIC`: Synonym ist nur für den Benutzer definiert.
- `PUBLIC` ist das Synonym systemweit verwendbar. Geht nur mit `CREATE ANY SYNONYM`-Privileg.

Beispiel: Benutzer will oft die Relation “City”, aus dem Schema “dbis” verwenden.

- `SELECT * FROM dbis.City;`
- ```
  CREATE SYNONYM DCity
    FOR dbis.City;

  SELECT * FROM DCity;
  ```

Synonyme löschen: `DROP SYNONYM <synonym>;`

ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS

- `GRANT SELECT` kann nicht auf Spalten eingeschränkt werden.
- Stattdessen: Views verwenden.

```
GRANT SELECT [<column-list>]   -- nicht erlaubt
ON <table>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

kann ersetzt werden durch

```
CREATE VIEW <view> AS
  SELECT <column-list>
  FROM <table>;

GRANT SELECT
ON <view>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```
```

ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS: BEISPIEL

pol ist Besitzer der Relation *Country*, will *Country* ohne Hauptstadt und deren Lage für *geo* les- und schreibbar machen.

View mit Lese- und Schreibrecht für *geo*:

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

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

- Referenzen auf Views müssen separat erlaubt werden:

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


7.5 Anpassung der Datenbank an Sprache, Zeichensatz etc.

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

ANPASSUNGS-PARAMETER

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

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

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

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

7.6 Optimierung der Datenbank

- möglichst wenige Hintergrundspeicherzugriffe
- Daten soweit wie möglich im Hauptspeicher halten

Datenspeicherung:

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

Anfrageoptimierung:

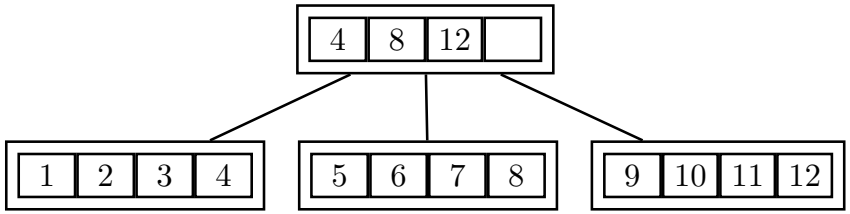
- Datenmengen klein halten
- frühzeitig selektieren
- Systeminterne Optimierung

Algorithmische Optimierung !

ZUGRIFFSPFADE: INDEXE

Zugriff über indizierte Spalte(n) erheblich effizienter.

- Baumstruktur; ORACLE: B*-Mehrweg-Baum,
- B*-Baum: Knoten enthalten *nur* Weg-Information, Verzweigungsgrad hoch, Höhe des Baumes klein.



- Suche durch Schlüsselvergleich: logarithmischer Aufwand.
- Schneller Zugriff (logarithmisch) versus hoher Reorganisationsaufwand ($\rightarrow$ Algorithmentechnik),
- bei sehr vielen Indexen auf einer Tabelle kann es beim Einfügen, Ändern und Löschen von Sätzen zu Performance-Verlusten kommen,
- logisch und physikalisch unabhängig von den Daten der zugrundeliegenden Tabelle,
- keine Auswirkung auf die *Formulierung* einer SQL-Anweisung, nur auf die *interne* Auswertung,
- mehrere Indexe für eine Tabelle möglich.

ZUGRIFFSPFADE: INDEXE

Zugriff über indizierte Spalte(n) erheblich effizienter:

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

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

- Ausgabe alphabetisch nach Code geordnet:
  Auf Schlüsselattribut ist automatisch ein Index angelegt und wird verwendet.

```
SELECT Name, Population
FROM Country
WHERE Population > 50000000;
```

- Ausgabe nicht sinnvoll geordnet:
  kein Index vorhanden, linearer Durchlauf (“Scan”).

```
CREATE INDEX CountryPopIndex ON Country (Population);
SELECT Name, Population
FROM Country
WHERE Population > 50000000;
```

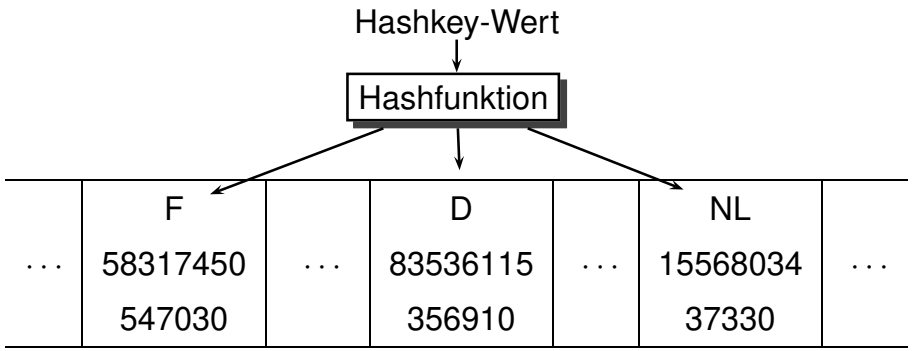
- Ausgabe jetzt nach Population geordnet.
  (Zugriff auf 50000000 über Baum, Rest linear ausgeben)

```
DROP INDEX CountryPopIndex;
```

HASHING

Aufgrund der Werte einer/mehrerer Spalten (*Hashkey*) wird durch eine *Hashfunktion* berechnet, wo das/die entsprechende(n) Tupel zu finden sind.

- Zugriff in *konstanter* Zeit,
- keine Ordnung.
- gezielter Zugriff auf die Daten über ein bestimmtes Land
  Hashkey: Country.Code



In ORACLE ist Hashing nur für *Cluster* implementiert.

CLUSTER

- Zusammenfassung einer Gruppe von Tabellen, die alle eine oder mehrere gemeinsame Spalten (Clusterschlüssel) besitzen, oder
- Gruppierung einer Tabelle nach dem Wert einer bestimmten Spalte (Clusterschlüssel);
- bei einem Hintergrundspeicherzugriff werden semantisch zusammengehörende Daten in den Hauptspeicher geladen.

Vorteile eines Clusters:

- geringere Anzahl an Plattenzugriffen und schnellere Zugriffsgeschwindigkeit
- geringerer Speicherbedarf, da jeder Clusterschlüsselwert nur einmal abgespeichert wird

Nachteile:

- ineffizient bei häufigen Updates der Clusterschlüsselwerte, da dies eine physikalische Reorganisation bewirkt
- schlechtere Performance beim Einfügen in Cluster-Tabellen

CLUSTERING

Sea und geo_Sea mit Clusterschlüssel `Sea.Name`:

Cl_Sea		
Mediterranean Sea	Depth	
	5121	
	Province	Country
	Catalonia	E
	Valencia	E
	Murcia	E
	Andalusia	E
	Languedoc-R.	F
	Provence	F
	⋮	⋮
Baltic Sea	Depth	
	459	
	Province	Country
	Schleswig-H.	D
	Mecklenb.-Vorp.	D
	Szczecin	PL
	⋮	⋮

CLUSTERING

City nach (*Province,Country*):

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

ERZEUGEN EINES CLUSTERS IN ORACLE

Cluster erzeugen und Clusterschlüssel angeben:

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

```
CREATE CLUSTER Cl_Sea (SeaName VARCHAR2(35));
```

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

```
CREATE TABLE Cgeo_Sea
  (Province  VARCHAR2(35),
   Country   VARCHAR2(4),
   Sea       VARCHAR2(35))
  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;
```

TEIL III: Erweiterungen

Teil I: Grundlagen

Teil II: Diverses

Teil III: Prozedurale Konzepte, OO, Einbettung

- PL/SQL: Prozeduren, Funktionen, Trigger
- Objektorientierung
- SQL und Java
- SQL und XML

SITUATION

- 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 auch Java (JDBC/SQLJ),
- 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.

Kapitel 8
Prozedurale Erweiterungen: PL/SQL

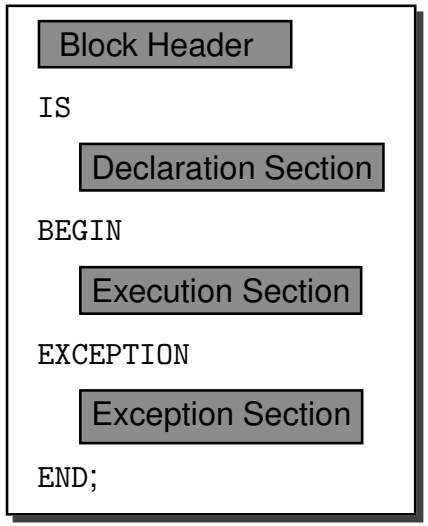
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- “Stored Procedures/Functions” innerhalb der DB
- direkter Zugriff auf Datenbankinhalt
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern

Weitere Nutzung

- Programmierung von Objektmethoden (seit Oracle 8/1997)

8.1 Prozeduren, Funktionen und Kontrollstrukturen in PL/SQL

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.

EINFACHE, ANONYME BLÖCKE

- nur Declaration und Execution Section
- werden direkt ausgeführt
- `DECLARE ... BEGIN ... END;`
  `/`

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

(Beispiel → nächste Folie)

AUSGABE-GENERIERUNG

- verwendet das DBMS_Output Package
- einmalig SET SERVEROUTPUT ON
  (z.B., beim Starten von sqlplus)
- innerhalb von PL/SQL-Blocks:

  ```
  dbms_output.put_line('bla');
  ```

- Bei Prozeduren etc.: Ausgabe erscheint erst *nach* kompletter Ausführung der Prozedur etc.

```
set serveroutput on;
DECLARE
bla NUMBER;
BEGIN
 bla := 42;
 dbms_output.put_line(bla);
END;
/
```

[Filename: PLSQL/output.sql]

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>;
/
```

- `datatype` darf dabei nur ein atomarer SQL-Datentyp sein. Es können damit also keine Tabellen zurückgegeben werden.
- 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 (siehe Oracle-Dokumentation *PL/SQL User's Guide and Reference*).

PROZEDUREN UND FUNKTIONEN

- Im Falle von "... created with compilation errors":

  ```
  SHOW ERRORS;
  ```

  ausgeben lassen.
- Prozeduren und Funktionen können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.
- Aufruf von Prozeduren im PL/SQL-Skript:
  `<procedure> (arg1,...,argn);`
  (wenn ein formaler Parameter als OUT oder IN OUT angegeben ist, muss das Argument eine Variable sein)
- Aufruf 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 in sqlplus 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;
/
```

[Filename: PLSQL/insertcountry.sql]

```
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;
/
```

[Filename: PLSQL/density.sql]

```
SELECT Density('D')
FROM dual;
```


PL/SQL-VARIABLEN UND DATENTYPEN.

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

```
DECLARE
<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.

```
DECLARE
anzahl NUMBER DEFAULT 0;
name VARCHAR2(35);
```

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 ...
```

```
DECLARE
cname country.name%TYPE;
BEGIN
 SELECT name
 INTO cname
 FROM country
 WHERE code='D';
 dbms_output.put_line(cname);
END;
/
```

[Filename: PLSQL/simple.sql]

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(35),
   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 ... ;
  ```

```
DECLARE
c continent%ROWTYPE;
BEGIN
 SELECT *
 INTO c
 FROM continent
 WHERE name='Europe';
 dbms_output.put_line(c.name || ' : ' || c.area);
END;
/
```

[Filename: PLSQL/simple2.sql]

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 (Forts.)

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(i)` (fkt): `TRUE` falls Zeile `i` der Tabelle nicht leer.
- `DELETE` (proc): Löscht alle Zeilen einer Tabelle.
- `DELETE(i)`: Löscht Zeile `i` 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 tmprowid ROWID;
BEGIN
   ⋮
   INSERT INTO Politics (Country,Independence)
     VALUES (Code,SYSDATE)
     RETURNING ROWID
     INTO tmprowid;
   ⋮
END;
```


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.

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;
```


8.2 Cursore/Iteratoren zur Verarbeitung von Ergebnismengen

- Datenbankanfragen: mengenorientiert
- Programmiersprache: variablenbasiert

Design Patterns: Kollektionen und Iteratoren

(vgl. Informatik I)

- Kollektion: Sammlung von Items (Liste, Baum, Heap, Menge)
- Iterator: Hilfsklasse zum Durchlaufen/Aufzählen aller Items
- Methoden:
  - Erzeugen/Initialisieren des Iterators,
  - Weiterschalten, Test, ob noch weitere Elemente vorhanden sind,
  - Zugriff auf ein Element,
  - (Schliessen des Iterators)

... Iteratoren werden im Weiteren immer wieder verwendet.

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:

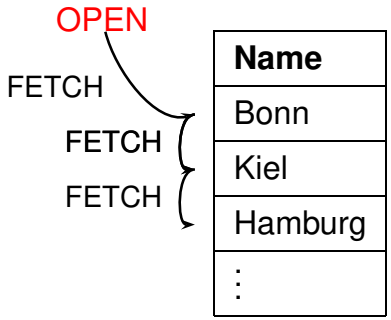
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```


Cursore: Grundprinzip

- `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 der Cursor initialisiert:

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



Cursore: Verwendung

- `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 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.

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

[Filename: PLSQL/cursor1.sql]

Cursore: Attribute

Kontrolle über die Verarbeitung eines Cursors:

- `<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.

Cursore: Attribute

```
CREATE OR REPLACE PROCEDURE first_city
     (the_country country.code%TYPE)
IS BEGIN
 DECLARE CURSOR cities_in
 (crs_country Country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
 city_in cities_in%ROWTYPE;
 BEGIN
   OPEN cities_in (the_country);
   FETCH cities_in INTO city_in;
   IF cities_in%FOUND
   THEN DBMS_OUTPUT.PUT_LINE(city_in.name);
   ELSE DBMS_OUTPUT.PUT_LINE('Nothing found!');
   END IF;
   CLOSE cities_in;
 END;
END;
/
```

[Filename: PLSQL/cursor-attrs.sql]

```
execute first_city('D');
execute first_city('X');
```

- Aufgabe: Programmieren Sie eine explizite WHILE-Schleife, die alle Städte eines Landes ausgibt.

Cursore: Hinweis

nicht möglich:

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

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

CURSOR FOR LOOP

Spezielle Schleife zur Iteration über den Inhalt eines Cursors:

```
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

Beispiel: Für jede Stadt in dem gegebenen Land soll der Name ausgegeben werden:

```
CREATE OR REPLACE PROCEDURE list_cities
     (the_country country.code%TYPE)
IS
BEGIN
  DECLARE CURSOR cities_in
     (crs_country country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
  BEGIN
     FOR the_city IN cities_in(the_country)
     LOOP
          dbms_output.put_line(the_city.name);
     END LOOP;
  END;
END;
/
```

[Filename: PLSQL/cursor-loop1.sql]

```
execute list_cities('D');
```

EINGEBETTETER CURSOR FOR LOOP

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

```
CREATE OR REPLACE PROCEDURE list_big_cities
(the_country country.code%TYPE)
IS
BEGIN
  FOR the_city IN
   (  SELECT Name
      FROM City
      WHERE Country = the_country
      AND Population > 1000000 )
  LOOP
   dbms_output.put_line(the_city.Name);
  END LOOP;
END;
/
```

[Filename: PLSQL/cursor-loop2.sql]

```
execute list_big_cities('D');
```


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).

DDL-Kommandos in PL/SQL

DDL-Statements werden in PL/SQL nicht direkt unterstützt:

- `EXECUTE IMMEDIATE <string>`
  <string> kann dabei eine Konstante sein, oder kann dynamisch zusammengesetzt werden

```
BEGIN
 execute immediate 'drop table continent';
END;
```

```
CREATE OR REPLACE PROCEDURE clean
IS
 BEGIN
  FOR tn IN
   ( SELECT table_name FROM all_tables
     WHERE table_name LIKE 'TMP_%')
  LOOP
   execute immediate 'DROP TABLE ' || tn.table_name;
  END LOOP;
END;
/
```

[Filename: PLSQL/clean.sql]

- Entsprechende Privilegien muss man direkt (GRANT ... TO <user>), und nicht nur über eine Rolle bekommen haben.

Dynamic SQL mit EXECUTE IMMEDIATE

- String zusammenbauen wie eben, oder
- Platzhalter für Werte, die dann bei Ausführung eingesetzt werden

```
DECLARE country VARCHAR2(4) := 'CDN';
        org VARCHAR2(10) := 'EU';
BEGIN
 execute immediate
     'insert into isMember VALUES (:1, :2, :3)'
   using country, org, 'candidate';
END;
/
```

- Wert in eine PL/SQL-Variable einlesen: INTO

```
CREATE OR REPLACE PROCEDURE sp (cname City.name%TYPE)
IS BEGIN declare cty city%ROWTYPE;
 BEGIN
 execute immediate 'select * from city where name= :1'
   into cty
   using cname;
 dbms_output.put_line(cty.population);
 END; END;
/
execute sp('Berlin');
```

[Filename: PLSQL/dynamicselect.sql]

8.3 Zugriffsrechte auf PL/SQL-Datenbankobjekte

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.

8.4 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: Beispiel

```
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;
```


(RE)AKTIVES VERHALTEN

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

8.5 Trigger

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 Oracle bisher 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`. Schreiben in `NEW`-Werte nur mit `BEFORE`-Trigger.
- `FOR EACH ROW`: Row-Trigger, sonst Statement-Trigger.
- `WHEN (<condition>)`: zusätzliche Bedingung; hier werden `OLD` und `NEW` verwendet; Subqueries an die Datenbank sind nicht erlaubt.
- Referenzieren der Variablen im PL/SQL-Teil als `:OLD` und `:NEW`.

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;
/
```

[Filename: PLSQL/changecode.sql]

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

SELECT * FROM Province WHERE Country='UK';
```

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
WHEN (:NEW.population > 2)
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code,SYSDATE);
END;
/
```

[Filename: PLSQL/newcountry.sql]

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

SELECT * FROM Politics WHERE country='LU';
```


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

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;
```

[Filename: PLSQL/allcountry-view.sql]

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

[Filename: PLSQL/insert-allcountry.sql]

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;
/
```

[Filename: PLSQL/instead-of.sql]

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

FEHLERBEHANDLUNG DURCH EXCEPTIONS IN PL/SQL

- 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;
/
```

[Filename: PLSQL/trigger-nachmittag.sql]

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)

8.6 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.

Kapitel 9
Objekt-Relationale Datenbanksysteme

Integration von relationalen Konzepten und Objektorientierung:

- Komplexe Datentypen: Erweiterung des Domain-Konzepts von SQL-2 (vgl. `DATE`, Geo-Koordinaten)
- Abstrakte Datentypen ("Objekttypen"):
  - Unterscheidung zwischen dem *Zustand* und *Verhalten* eines *Objektes* (Kapselung interner Funktionalität).
  - Im Gegensatz zu einem *Tupel* besitzt ein Objekt
    - Attribute (beschreiben seinen Zustand),
    - Methoden – Abfragen und Ändern des Zustandes: *Prozeduren* und *Funktionen* (Oracle 8: PL/SQL, Oracle 8i: auch Java, siehe Folie 300)
    - `MAP/ORDER`-Funktionen: Ordnung auf Objekttyp
- Spezielle Ausprägungen:
  - Geschachtelte Tabellen als Attributwerte,
  - Erweiternde Datentypen (Spatial etc.),
  - Built-In XMLType zur Verarbeitung von XML-Daten (siehe Folie 354).

STUFEN DER OBJEKTORIENTIERUNG

"Konservative" objektrelationale Erweiterungen (seit Oracle 8)

(siehe Folie 221)

- Objekte als "Werte" von Attributen:
  Spalten einer Tupeltabelle können *objektwertig* sein.
- Objekte anstelle von Tupeln:
  Tabellen von Tupeln vs. *Object Tables* aus Objekten
- Typ definiert gemeinsame Signatur seiner Instanzen (Objekte)
- bereits behandelt: Komplexe Attributtypen. Besitzen nur *Wertattribute*, keine Methoden.

Objektorientierte Datenbanken

(siehe Folie 246)

- Beziehungen nicht mehr über Schlüssel/Fremdschlüssel sondern über Referenzen
  $\Rightarrow$ Navigation anstatt Joins
- seit ORACLE 9i: Subtypen und Vererbung, Objekttypen aus Java-Klassen.

9.1 Objektrelationale Konzepte

Alles funktioniert (fast) genauso wie bisher:

- Spalten einer Tupeltabelle können *objektwertig* sein (vgl. Geo-Koordinaten)
- Tabellen von Tupeln vs. *Object Tables* aus Objekten

  `INSERT INTO <table>`
  `VALUES(<object-constructor>(attr`$_1$`, ..., attr`$_n$`))`

  anstatt

  `INSERT INTO <table>`
  `VALUES(attr`$_1$`, ..., attr`$_n$`)`
- Zugriff auf Attribute wie bisher mit `tablename.attr`,
- zusätzlich Aufruf von Methoden mit `tablename.meth(...)`.

9.1.1 Definition von Objekttypen

Typdeklaration

- Attribute,
- *Signaturen* der Methoden,

Typ-Implementierung

- **Type Body:** Implementierung der Methoden in PL/SQL
- seit Oracle 8i auch in PL/SQL+Java (siehe Folien 300 und 306)

OBJEKTTYPDEKLARATION

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   <attr> <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>]
  );
/
```

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

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
  );
/
```

[Filename: ObjRel/geocoord-type.sql]

- wenn der Objekttyp bereits existiert (wie in diesem Fall):

```
ALTER TYPE GeoCoord
 ADD MEMBER FUNCTION Distance (other IN GeoCoord)
     RETURN NUMBER,
 ADD MAP MEMBER FUNCTION Distance_Greenwich
     RETURN NUMBER
 CASCADE ;
```

[Filename: ObjRel/geocoord-type-ext.sql]

- CASCADE: “forwards” definition to tables where it is used.

TYPE BODY

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

Funktionen: dürfen den Datenbankzustand nicht verändern,

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

$\Rightarrow$ verwenden nur den Zustand der beteiligten Objekte.

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;
/
```


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;
/
```

[Filename: ObjRel/geocoord-body.sql]

Erzeugung von Objekten

- Konstruktormethode:

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

  Also kein `NEW`, sondern nur einfach

```
GeoCoord(8,48)
CityType('Berlin', 'Berlin', 'D',
         3472009, GeoCoord(13.3,52,45))
```

Methodenaufruf

- Funktionen: in Anfragen oder in PL/SQL-Programmen
- Prozeduren: in PL/SQL-Programmen
- Syntax:

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

Beispiel

Wie gross ist der Abstand zwischen zwei Längengraden auf der Höhe von Berlin, bzw. am Äquator?

```
SELECT geoCoord(-30,52.45).Distance(geoCoord(-31,52.45))
FROM DUAL;
SELECT geoCoord(-30,0).Distance(geoCoord(-31,0))
FROM DUAL;
```


9.1.2 Zugriffscharakteristik (Veraltet – bis einschl. Version 8i notwendig)

- Explizite Angabe der Read/Write-Zugriffscharakteristik der Methoden in PRAGMA RESTRICT REFERENCES-Klauseln
- seit 9i: wird zur Compile-Time intern gemacht (überprüft wurde es sowieso schon ...)

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (<attr> <datatype>,
   <attr> <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>]
  );
/
```

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: es muss *zugesichert* werden, 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)
```

$\Rightarrow$ verwendet nur den Zustand der beteiligten Objekte.

BEISPIEL: GEO-KOORDINATEN MIT PRAGMA-KLAUSEL

```
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)
  );
/
```

9.1.3 Verwendung von Objekttypen

- Als Werte von Attributen: “Spaltenobjekte” (vgl. Geo-Koordinaten)
- in *Objekttabellen*: `TABLE OF <objecttype>` “Zeilenobjekte” vollwertige Objekte

SPALTENOBJEKTE

- Attribut eines Tupels oder eines Objekts ist objektwertig,

```
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)));
```

[Filename: ObjRel/mountain-table.sql]

- Constraints werden wie immer bei der Tabellendefinition angegeben.

```
INSERT INTO Mountain
   VALUES ('Feldberg', 1493, GeoCoord(7.5, 48.5));
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*,
- ihre Attribute verhalten sich genauso wie die Attribute von Tupeltabellen,
- zusätzlich kann man Methoden aufrufen,
- referentielle Integritätsbedingungen zwischen bestehenden relationalen Tabellen und solchen Objekttabellen wie üblich formulierbar,
- (erhalten eine eindeutige OID und sind damit referenzierbar)

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

mit `<constraint-list>` wie bisher:

- 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(35),
   Country VARCHAR2(4),
   Population NUMBER,
   Coordinates GeoCoord,
   MEMBER FUNCTION Distance (other IN City_Type)
      RETURN NUMBER,
   MEMBER FUNCTION NoOfOrganizations
      RETURN NUMBER);
/
```

[Filename: ObjRel/citytype.sql]

ZEILENOBJEKTE

Beispiel: *City_Type*

```
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;
  MEMBER FUNCTION NoOfOrganizations RETURN NUMBER
  IS
   n NUMBER;
  BEGIN
   SELECT count(*) INTO n
   FROM Organization o
   WHERE o.city = SELF.name
     AND o.province = SELF.province
     AND o.country = SELF.country;
   RETURN n;
  END;
END;
/
```

[Filename: ObjRel/citytypebody.sql]

OBJEKTTABELLEN: ZEILENOBJEKTE

- der (ggf. mehrspaltige) Primärschlüssel wird als Tabellenbedingung angegeben,
- Die Fremdschlüsselbedingung auf die relationale Tabelle *Country* wird ebenfalls als Tabellenbedingung angegeben:

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

[Filename (beides zusammen): ObjRel/citytable.sql]

VERWENDUNG VON OBJEKTTABELLEN

Auslesen und Ändern von Attributen wie bekannt

- Auslesen:

```
SELECT Name FROM ORCity;
SELECT * FROM ORCity;
```

- Ändern:

```
UPDATE ORCity cty
SET coordinates = NULL
WHERE cty.coordinates.longitude IS NULL;
```

Methodenaufrufe wie vermutet

```
SELECT Name, c.NoOfOrganizations() FROM ORCity c
WHERE c.NoOfOrganizations() > 0;
```


VERWENDUNG VON OBJEKTTABELLEN

... Auslesen von Objekten als Objekte:

Das folgende geht so *nicht*:

```
SELECT cty1.Distance(cty2)
FROM ORCity cty1, ORCity cty2
WHERE cty1.Name='Berlin' AND cty2.Name='Stuttgart';
```

Die VALUE()-Funktion

```
VALUE (<var>)
```

selektiert ein Objekt als Objekt:

```
SELECT VALUE(cty)
FROM ORCity 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 ORCity cty1, ORCity cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

```
SELECT cty1.Name, cty2.Name,
   cty1.Distance(VALUE(cty2))
FROM ORCity cty1, ORCity 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 ... ;
```


9.1.4 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: (Betragsfunktion)

- 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: (vgl. Methode compareTo($other$) des “Comparable” Interfaces in Java)

- 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 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,
       cty.coordinates.Distance_Greenwich()
FROM ORCity cty
WHERE NOT coordinates IS NULL
ORDER BY coordinates;
```

[Filename: ObjRel/orderby.sql]

ORDER-METHODEN

- Vergleich von SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird.
- Ergebnis: NUMBER
  - $x < 0$ falls SELF < Parameter,
  - 0 (Gleichheit), oder
  - $x > 0$ falls 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).

9.1.5 Objektrelationale Modellierung: Zusammenfassung

- Objekte anstatt Tupel oder Attributwerte
- Anfragen praktisch unverändert gegenüber rein relationaler DB (insb. Beziehungen weiterhin über Schlüssel/Fremdschlüssel und Join-basierte Anfragen)
- zusätzlich Methoden, Ordnungsmethoden.

BEISPIEL/AUFGABE

(wird auf Folie 256 analog ausprogrammiert)

- City, Country, Organization als Objekttypen und -tabellen
- Komfortablere Methoden: Mitgliedschaften werden über Methoden eingetragen und abgefragt (ohne Berücksichtigung der Arten der Mitgliedschaft):

  organization.isMember($carcode$)
  country.isMemberIn($org\text{-}abbrev$)
  organization.makeMember($carcode$)
  country.makeMemberIn($org\text{-}abbrev$)

  Interne Implementierung z.B. über die bekannte Tabelle `isMember`.

  Hinweis: Boolesche Anfragen der Art "Ist x Mitglied in y" sind damit möglich. Es ist jedoch keine Methode "alle Mitglieder von y möglich – diese müßte eine Relation bzw. Menge zurückgeben.
- ... man kann aber diese Implementierung dann auch beliebig ändern.

9.2 Objektorientierte Modellierung

... soweit dienten die Datentypen im wesentlichen zur Bereitstellung von spezialisiertem Verhalten:

- Built-in: DATE
- zusammengesetzt: Geo-Koordinaten
- Geschachtelte Tabellen (parametrisierter Datentyp)
- benutzerdefinierte Objekttypen
- Grundlage für Datentypen wie XMLType etc.

Objektorientierte Modellierung

Geht über die Nutzung als “Datentypen” hinaus ...

- ... zu Modellierungsaspekten:
- Spezialisierung: Klassenhierarchie; Subtypen als Spezialisierung allgemeiner Typen.
- Objekt-Identität und Referenzen auf Objekte als Werte von Attributen zum Ausdrücken von Beziehungen,
- Objekte: *Wertattribute* und *Referenzattribute*.
- Anfragen durch Navigation etc. ($\Rightarrow$ unsymmetrisch)

OBJEKTREFERENZEN

- Weiterer Datentyp für Attribute: Referenzen auf Objekte

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

- *Objekttyp* als Ziel der Referenz.
- nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – können referenziert werden.
- problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen.
- `PRIMARY KEY`s dürfen keine `REF`-Attribute umfassen.
- 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 ... ;
  ```

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,
   hasHqIn REF City_Type,
   MEMBER FUNCTION isMember (the_country IN VARCHAR2)
   -- EU.isMember('SLO') = 'membership applicant'
      RETURN VARCHAR2,
   MEMBER FUNCTION people RETURN NUMBER,
   MEMBER FUNCTION numberOfMembers RETURN NUMBER,
   MEMBER PROCEDURE addMember
      (the_country IN VARCHAR2, the_type IN VARCHAR2));
/
```

[Filename: Obj/org-type.sql]

Beispiel: Objekttyp *Organization*

Tabellendefinition:

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

- Type Body noch nicht definiert.
  Weiter erstmal nur mit einem Objekt als Beispiel:

Einfügen unter Verwendung des Objektkonstruktors:

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

Setzen des Referenzattributes *hasHqIn*:

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

[Filename (alles zusammen): Obj/org-table.sql]

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, hasHqIn
FROM Organization_ObjTab;
```

Name	Abbrev	hasHqIn
European Community	EU	<oid>

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

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

Abbrev	hasHqIn
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.hasHqIn.name
FROM Organization_ObjTab org;
```

Abbrev	hasHqIn.Name
EU	Brussels

Die Funktionen VALUE, REF, DEREF

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.

ZYKLISCHE REFERENZEN

Die Modellierung als Objektgraph (d.h., Beziehungen nicht durch Tabellen, sondern als Objektreferenzen) führt oft zu Zyklen:

- `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.

UNVOLLSTÄNDIGE DATENTYPEN

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

```
CREATE OR REPLACE TYPE City_type;
/
```

- Die Nutzung in Referenzen ist damit erlaubt:

```
CREATE TYPE city_list AS TABLE OF REF City_type;
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(35),
   Code VARCHAR2(4),
   Capital REF City_Type);
/
```

- Die *direkte Nutzung* wäre erst erlaubt, wenn City_type komplett ist:

```
CREATE TYPE city_list_2 AS TABLE OF City_type;
/ -- waere eine Tabelle von City-Objekten
CREATE OR REPLACE TYPE Country_Type_2 AS OBJECT
  (Name VARCHAR2(35),
   Code VARCHAR2(4),
   Capital City_Type);
/ -- Capital waere ein Spaltenobjekt
```

Zyklische Referenzen: Beispiel

```
CREATE OR REPLACE TYPE City_Type
/

CREATE OR REPLACE TYPE Country_Type AS OBJECT
  (Name VARCHAR2(35),
   Code VARCHAR2(4),
   Capital REF City_Type,
   Area NUMBER,
   Population NUMBER);
/

CREATE OR REPLACE TYPE Province_Type AS OBJECT
  (Name VARCHAR2(35),
   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);
/
```


Objektorientierung: Modellierungsaspekte

- Beziehungen durch Referenzattribute,
- Anfragen per Navigation (anstatt Join),
- können nur in einer Richtung verfolgt werden,
- erfordert also doppelte Speicherung,
- müssen auf beiden Seiten separat konsistent gehalten werden.

Beispiel/Aufgabe

- City, Country, Organization als Objektgraph
- Beziehungen immer über Methoden behandeln:

  organization.isMember($carcode$)
  country.isMemberIn($org\text{-}abbrev$)
  organization.makeMember($carcode$)
  country.makeMemberIn($org\text{-}abbrev$)

- Interne Implementierung von z.B. Mitgliedschaften wie oben als Collection von Referenzen, oder über die bekannte Tabelle `isMember`.

9.3 Methoden: Funktionen und Prozeduren

`TYPE BODY` enthält die Implementierungen der Methoden in PL/SQL

Anpassung von PL/SQL an Objektrelationale Features

- 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.
- Weitere Erweiterung: Java (siehe Folie 300).

Member-Methods: Beispiel

```
CREATE OR REPLACE TYPE BODY Organization_Type IS

MEMBER FUNCTION isMember (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;                                  (bitte umblättern)
```

Member-Methods: Beispiel (Forts.)

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

MEMBER PROCEDURE addMember
   (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  IF NOT SELF.isMember(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;
/
```

[Filename: Obj/orgs-type-body.sql]

- `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.isMember('D')
  FROM Organization_ObjTab org
  WHERE NOT org.isMember('D') = 'no';
```

(noch ist die Tabelle aber nicht sinnvoll gefüllt ...)

Prozeduren

- `MEMBER PROCEDURES` können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden.

⇒ freie Prozeduren in PL/SQL, um `MEMBER PROCEDURES` aufzurufen

Beispiel: Freie Prozedur

```
CREATE OR REPLACE PROCEDURE makeMember
  (the_org IN VARCHAR2, the_country IN VARCHAR2,
   the_type IN VARCHAR2) IS
  n NUMBER;
  x 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 x
   FROM Organization_ObjTab org
   WHERE Abbrev = the_org;
  IF x.isMember(the_country)='no' THEN
   x.addMember(the_country, the_type);
  END IF;
END;
/
```

[Filename: Obj/makemember.sql]

```
EXECUTE makeMember('EU', 'USA', 'special member');
EXECUTE makeMember('XX', 'USA', 'member');
```


Beispiel: Füllen der Objekttabelle

Übertragung des Datenbestandes aus den relationalen Tabellen *Organization* und *isMember* 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 isMember)
 LOOP makeMember(the_membership.organization,
                 the_membership.country,
                 the_membership.type);
 END LOOP;
END;
/
EXECUTE Insert_All_Members;
UPDATE Organization_ObjTab org
SET hasHqIn =
  (SELECT REF(cty)
   FROM ORCity cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
   AND cty.Name = old.City
   AND cty.Province = old.Province
   AND cty.Country = old.Country);
```

[Filename: Obj/fill-organizations.sql]

Beispiel: Nutzung freier Methoden

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

[Filename: Obj/is-member.sql]

```
SELECT isMemberIn('EU', 'HR')
FROM DUAL;
```

isMemberIn('EU', 'HR')
applicant

Es ist (zumindest bis ORACLE 11) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

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

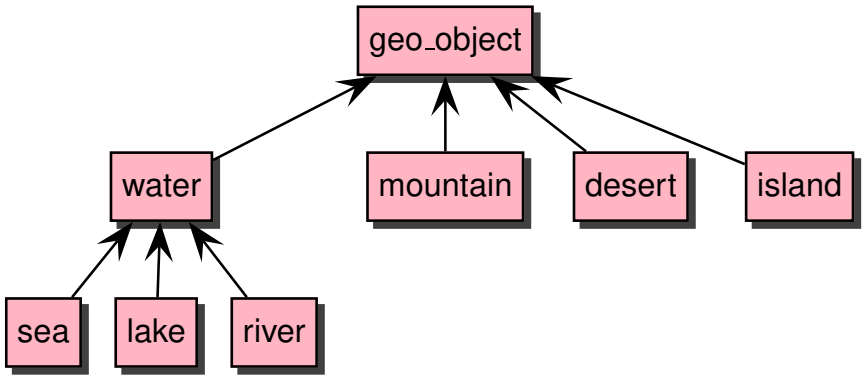

MODELLIERUNG VS. IMPLEMENTIERUNG

- Das Beispiel illustriert Objektmethoden und ihre Anbindung durch freie Methoden am *objektorientierten* Szenario:
  - Headquarter als Referenz,
  - Mitglieder als geschachtelte Tabelle,
- man kann dieselben Methoden auch mit einer *objektrelationalen* Tabelle OROrganization, und Ablegen der Mitgliedschaften in der Relation `isMember` implementieren (Aufgabe).

$\Rightarrow$ für den Benutzer bleiben die freien Methoden dieselben.

9.4 Klassenhierarchie und Vererbung

(Abschnitt ist optional)



KLASSENHIERARCHIE UND VERERBUNG

- seit ORACLE 9i
- Typhierarchie:
- Subtyp: abgeleitet von einem Elterntyp
- Vererbung: Verbindung zwischen Subtypen und ihren Obertypen in einer Typhierarchie
- Subtypen: “Spezialisierung”
  - hinzufügen neuer Attribute und Methoden
  - overriding (Überschreiben) von geerbten Methoden
- Spezielle Eigenschaften von Klassen:
  abstrakte (NOT INSTANTIABLE) und finale (FINAL) Klassen
- abstrakte Klassen können abstrakte Methoden (NOT INSTANTIABLE) haben
- Klassen können finale Methoden haben:
  diese können in Subklassen nicht überschrieben werden

ERZEUGEN VON SUBTYPEN

Abstrakte Klasse für geographische Objekte
(die alle einen Namen besitzen):

```
CREATE OR REPLACE TYPE geo_object_type AS OBJECT (
   name VARCHAR2(35),
   MEMBER FUNCTION get_name RETURN VARCHAR2,
   NOT INSTANTIABLE
          MEMBER FUNCTION set_name RETURN VARCHAR2
  )
  NOT INSTANTIABLE -- DEFAULT: INSTANTIABLE
  NOT FINAL;       -- DEFAULT: FINAL
/

CREATE OR REPLACE TYPE BODY geo_object_type IS
  MEMBER FUNCTION get_name RETURN VARCHAR2
  IS BEGIN RETURN name; END;
  -- no implementation for set_name
  -- (to be class-specific)
  END;
/
```


ERZEUGEN VON SUBTYPEN

```
CREATE OR REPLACE TYPE water_type
  UNDER geo_object_type (
  MEMBER FUNCTION bla RETURN NUMBER
  -- empty derivation not allowed in current version
  )
  NOT FINAL
  NOT INSTANTIABLE;
/
```

- + Angabe eines TYPE BODY, der `bla` implementiert.

ERZEUGEN VON SUBTYPEN

- finale Klassen für Meere, Seen und Flüsse etc.
- müssen alle bis jetzt nicht implementierten Methoden anbieten
- erfordert OVERRIDING

```
CREATE OR REPLACE TYPE sea_type
  UNDER water_type (
  depth NUMBER,
  OVERRIDING
      MEMBER FUNCTION set_name RETURN VARCHAR2,
  [OVERRIDING -- optional
      MEMBER FUNCTION bla RETURN NUMBER]
  )
  INSTANTIABLE;
/
```

- + Angabe eines TYPE BODY, der `set_name` implementiert.
- optional kann man auch `bla` überschreiben.
- analog für Meere, Flüsse, Berge, Inseln und Wüsten.

TABELLEN ÜBER ALLGEMEINEN KLASSEN

- eine Tabelle für alle geographischen Objekte

```
CREATE TABLE geo_obj OF geo_object_type;
INSERT INTO geo_obj
  SELECT sea_type(name, depth) FROM sea;
INSERT INTO geo_obj
  SELECT lake_type(name, area) FROM lake;
INSERT INTO geo_obj
  SELECT river_type(name, NULL, NULL, NULL, length)
  FROM river;
INSERT INTO geo_obj
  SELECT mountain_type(name, height, coordinates)
  FROM mountain;
INSERT INTO geo_obj
  SELECT desert_type(name, area) FROM desert;
INSERT INTO geo_obj
  SELECT island_type(name, islands, area, coordinates)
  FROM island;
```

ANFRAGEN AN TABELLEN ÜBER ALLGEMEINEN KLASSEN

- die Tabelle `geo_obj` ist eine Kollektion von Objekten der Klasse `geo_obj_type` (abstrakt)
- enthält Instanzen der finalen Subklassen, z.B. Flüsse und Berge.
- Substituierbarkeit:
  "Ein Objekt eines Typs t kann überall auftreten, wo ein Objekt eines Obertyps von t erwartet wird"
  - Zeilenobjekte in Objekttabellen
  - Spaltenobjekte (objektwertige Attribute)
  - Referenzattribute
  - Argumente und Rückgabewerten von Methoden
- select name from geo_obj;
  da alle geo_objects einen Namen haben.

ANFRAGEN AN KLASSENSPEZIFISCHE EIGENSCHAFTEN

- Kollektion von Instanzen einer abstrakten Klasse
- Auswahl der Objekte einer speziellen Subklasse
- Verwendung von klassenspezifischen Eigenschaften
- ähnlich wie in C++/Java: Typumwandlungen

SPEZIELLSTE KLASSENZUGEHÖRIGKEIT

- `SYS_TYPEID(<object>)`
  ergibt die ID der speziellsten Klasse, zu der ein Objekt gehört
- herausfinden des Klassennamens in `all_types`

```
SELECT type_name, typeid, supertype_name
FROM all_types
WHERE typeid = (SELECT SYS_TYPEID(value(x))
                FROM geo_obj x
                WHERE name='Llullaillaco');
```

type_name	typeid	supertype_name
mountain	08	geo_object

TYPTESTS

- <object> IS OF(<type>)
  testet ob <object> vom Typ <type> ist.
- normalerweise testet man Zugehörigkeit zu einem Subtyp des für die Tabelle bekannten Typs.
- Ausgeben aller Namen von Bergen:

```
SELECT x.name
FROM geo_obj x
WHERE value(x) IS OF (mountain_type);
```

- wie bekommt man die Namen und die Höhe?

```
SELECT x.name, x.height
```

ist nicht erlaubt
(geo_objects haben keine Höhe!)

TYPUMWANDLUNGEN

- TREAT (<object> AS <type>)
  behandelt <object> als eine Instanz des Typs <type>
- falls möglich
- sonst: NULL

```
SELECT x.name,
       (TREAT (value(x) AS mountain_type)).height
FROM geo_obj x
WHERE value(x) IS OF (mountain_type);
```

9.5 Diverses zu Objekttypen

Änderungen an Objekttypen

Benutzerdefinierte Typen können mit `ALTER TYPE` verändert werden:

- Hinzunehmen und Löschen von Attributen
- Hinzunehmen und Löschen von Methoden
- Modifikation eines numerischen Attributs (Länge, Präzision)
- `VARCHAR` kann verlängert werden
- Ändern der `FINAL`- und `INSTANTIABLE`-Eigenschaften

```
ALTER TYPE <type>
  ADD ATTRIBUTE (<name> <datatype>),
  DROP ATTRIBUTE <name>,
  MODIFY ATTRIBUTE (<name> <datatype>),
  ADD MEMBER FUNCTION/PROCEDURE <method-spec>
      -- requires new CREATE TYPE BODY!
  DROP MEMBER FUNCTION/PROCEDURE <method-spec>
      <options>
```


Änderung von Typdefinitionen: Abhängigkeiten

Objekttypen-Definitionen und Referenzattribute erzeugen einen Graphen, der dem von Fremdschlüsseldefinitionen erzeugten ähnlich ist.

- Abhängige Schemaobjekte, die einen Typ referenzieren sind z.B.:
  - Tabellen
  - Typen, insb. Subtypen
  - PL/SQL: Prozeduren, Funktionen, Trigger
  - Views, Objekt-Views
- Veränderungen: `ALTER TYPE`
- Propagieren von Änderungen: `CASCADE`
- Compilierbare abhängige Datenbankobjekte (PL/SQL, Sichten, ...): `INVALIDATE`
  werden als *invalid* markiert und bei der nächten Benutzung neu compiliert.
- Tabellen: neue Attribute werden mit `NULL`werten initialisiert.

Die Datenbank muss nach Typveränderungen revalidiert werden
(siehe Handbücher).

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);
```

- Funktions-basierte Indexe:

```
CREATE INDEX name ON
    Organization_Obj_Tab (numberOfMembers);
```

  arbeiten mit vorberechneten Werten.

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.

REFERENTIELLE INTEGRITÄT

- Vgl. `FOREIGN KEY ... REFERENCES ... ON DELETE/UPDATE CASCADE`
- Veränderungen an Objekten:
  OID bleibt unverändert
  $\rightarrow$ 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;
  ```


9.6 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:
Die relationale Repräsentation ist oft effizienter:

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

$\Rightarrow$ Definition eines relationalen Basisschemas mit Object-Views.

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

$\Rightarrow$ Veränderungen durch Neudefinition 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.

- enthalten Zeilenobjekte, d. h. hier werden neue Objekte *definiert*.
- 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.

Syntax

- 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(35),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/
```

Sinnvollerweise würde man hier gleich auch noch Methoden definieren.

```
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,
   hasHqIn REF City_Type,
   numberOfPeople NUMBER);
/

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

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

Beide angegebenen Attribute sind auch einzeln nicht erlaubt.

9.7 Objektorientierung und Data Dictionary

- TABS = USER_TABLES(table_name, ...): alle *relationalen* Tabellen die dem Benutzer gehören
- USER_OBJECT_TABLES(table_name, table_type, nested...): alle Objekttabellen die dem Benutzer gehören. `table_type` gibt den Objekttyp an, aus dessen Instanzen die Tabelle besteht. Hierzu gehören auch Nested Tables!
- USER_ALL_TABLES(table_name, ...): alle Tabellen die dem Benutzer gehören.
- USER_OBJECTS(object_name, object_type, status): alle Datenbankobjekte, die dem Benutzer gehören.
- Objekttabellen, deren Type mit DROP TYPE ... FORCE gelöscht wurde, stehen nicht mehr in USER_OBJECT_TABLES. Sie existieren aber noch (neues CREATE TABLE ergibt eine Fehlermeldung). (Bsp. nächste Folie)
- USER_TYPES(type_name): alle Objekttypen, die dem Benutzer gehören.

Löschen von Datenbankobjekten

```
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
SELECT table_name, table_type FROM user_object_tables;
-- name: blatab, type: bla
DROP TYPE bla FORCE;
SELECT table_name, table_type FROM user_object_tables;
-- nichts
CREATE TYPE bla AS OBJECT (x NUMBER);
/
CREATE TABLE blatab OF bla;
-- ORA-00955: name is already used by an existing object
SELECT * FROM user_objects WHERE object_name='BLATAB';
-- blatab - invalid
SELECT object_name, object_type, status from user_objects;
SELECT object_name, object_type, status from user_objects
 WHERE object_type = 'TABLE';
```

Problem: alle Tabellen löschen (Skript drop-all-tables)

- Verwendung von user_tables ignoriert invalide Tabellen.
- Nested Tables können nicht mit DROP TABLE gelöscht werden, sondern nur mit der Tabelle zu der sie gehören.
- Verwendung von user_objects where object_type = 'TABLE' bricht ab, wenn es eine (möglicherweise invalide, also in user_object_tables auch nicht mehr als nested gelistete) Nested Table löschen soll.

9.8 Fazit

- *Objektrelationale Tabellen* (Folie 221):
  Kompatibilität mit den grundlegenden Konzepten von SQL. U.a. Fremdschlüsselbedingungen von objektrelationalen Tabellen zu relationalen Tabellen.
- *Objektorientiertes Modell* (Folie 246):
  ... etwas kompliziert zu handhaben.
- *Object/Objekt-Relationale Views* (Folie 279):
  erlauben ein objektorientiertes externes Schema. Benutzer-Interaktionen werden durch Methoden und `INSTEAD OF`-Trigger auf das interne Schema umgesetzt. Implementierung auf relationaler Basis.
- *Objekttypen-Konzept* als Basis für (vordefinierte, in Java implementierte Klassen als) Datentypen zur Behandlung von nicht-atomaren Werten (XML (siehe Folie 354), Multimedia etc.).

Kapitel 10
Embedded SQL

KOPPLUNGSARTEN ZWISCHEN DATENBANK- UND PROGRAMMIERSPRACHEN

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

10.1 Embedded SQL: Grundprinzipien

... realisiert für C, Pascal, C++, Java (als SQLJ, siehe Folie 342) und weitere.

Impedance Mismatch bei der SQL-Einbettung

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

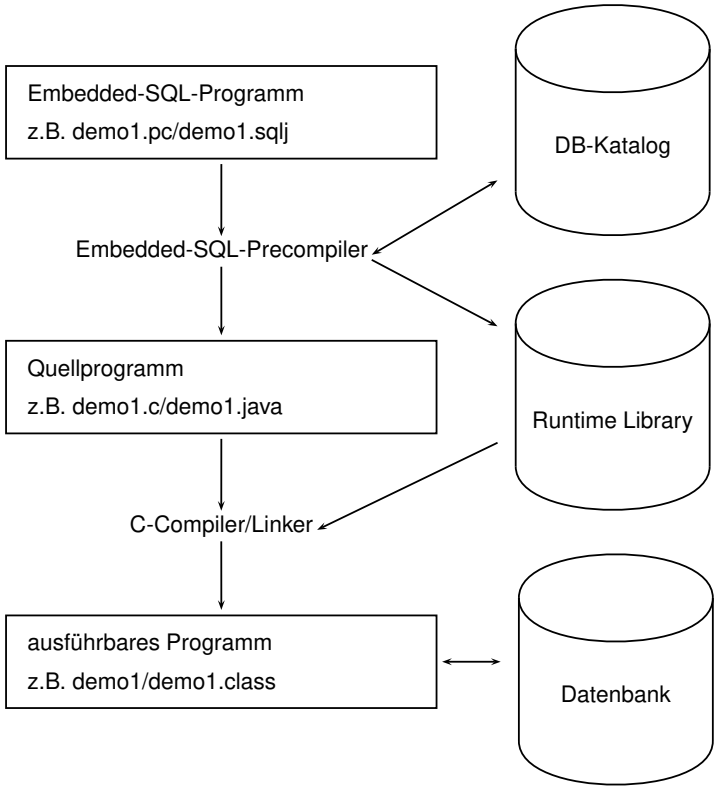
Realisierte Lösung

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

Auswirkungen auf die Hostsprache

- Struktur der Hostsprache bleibt unverändert,
- Spezielle Anweisungen zum Verbindungsaufbau,
- Jede SQL-Anweisung kann eingebettet werden,
- Verwendung von “Hostvariablen” (der umgebenden Programmiersprache) in SQL-Statements,
- SQL-Anweisungen wird `EXEC SQL` (oder sonstwas) vorangestellt.

ENTWICKLUNG EINER EMBEDDED SQL-APPLIKATION



- SQLJ (siehe Folie 342): Zwischenschritt bei der Compilierung muss nicht separat ausgeführt werden.

10.2 Embedded SQL in C [Legacy]

Hinweis: dieser Abschnitt kann ausgelassen und durch SQLJ (Folie 342) ersetzt werden. Er ist nur noch für die Arbeit mit Legacy-Datenbanken relevant, die diese Technologie verwenden.

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 und Ausnahmefällen

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[35];     /* 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 damit häufig auf einen Cursor verzichtet werden kann.
- verringert zudem Kommunikationsaufwand zwischen Client und Server.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[35][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.

PL/SQL IN EMBEDDED-ANWEISUNGEN

- 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;
```

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: `EXEC SQL SAVEPOINT <name>`

MECHANISMEN FÜR AUSNAHMEBEHANDLUNG

SQLCA (SQL Communications Area)

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.

Kapitel 11
Java und Datenbanken

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

Mehrere der bisher behandelten Aspekte können mit Java gekoppelt werden:

- Prozeduren und Funktionen, Member Methods: Java Stored Procedures (Folie 300),
- Objekttypen: Java Object Types (Folie 306)
  (so kann man beliebige Datenstrukturen implementieren und anbieten $\rightarrow$ XML),
- Low-Level-Infrastruktur für Datenbankzugriff aus Java: JDBC (Folie 310),
- Embedded SQL (intern basierend auf JDBC): SQLJ (Folie 342).

11.1 Java in Stored Procedures und Member Methods

- Oracle hat (seit 8i) eine eigene, integrierte JVM
  - keine GUI-Funktionalität
  - Java-Entwicklung außerhalb des DB-Servers
  - keine `main()`-Methode in Klassen, nur statische Methoden (= Klassenmethoden)
  - ab 9i Nutzung von Klassen als Objekttypen
  - kein Multithreading
  - DB-Zugriff über JDBC/SQLJ, dabei wird der serverseitige JDBC-Treiber verwendet (siehe Folien 310 und 342).
- Quelldateien (`.java`), Klassendateien (`.class`) oder Archive (`.jar`) können eingelesen werden.
- Shell: `loadjava`, `dropjava`
- DB: CREATE JAVA SOURCE, DROP JAVA SOURCE
- Einbettung in Prozedur/Funktion (*Wrapper*, *call spec*) (void-Methoden als Prozeduren, non-void als Funktionen)

LADEN VON JAVA-CODE PER SHELL

Außerhalb der DB wird eine Klasse geschrieben:

```
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!";  // return value
 } }
```

[Filename: Java/Greet.java]

- Einlesen in die Datenbank mit `loadjava`.
  Empfehlenswert ist hier ein Alias:

  `alias loadjava='loadjava -u` *uname*/*passwd*`'`

  dann braucht das Passwort nicht angegeben zu werden:

  ```
  dbis@s042> loadjava -r Greet.java
  ```

- `-r`: wird sofort compiliert und Referenzen aufgelöst (sonst: on-demand zur Laufzeit)
- Einlesen von .class-Dateien (ohne `-r`):

  ```
  dbis@s042> loadjava Greet.class
  ```

  Klappt nur, wenn bei der Datenbank dieselbe Java-Version, wie auf dem Rechner wo man es compiliert hat, läuft.
- analog mit .jar (das Sourcen und/oder class-Files enthält)
- Löschen einer Java-Klasse: analog mit `dropjava Greet.java`

Erzeugen von Java-Klassen in SQL

Klasse ausserhalb der DB entwickeln und dann in der DB generieren:

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
// here also imports are allowed
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java"); // Java output
    return "Hello " + name + "!";  // return value
  }
};
/
```

[Filename: Java/Greet-Create.sql]

- Wichtig: in doppelte Anführungszeichen (Version 11g)
- dann mit dem auf der nächsten Folie beschriebenen Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- Löschen mit

  ```
  DROP JAVA SOURCE "Hello";
  ```

- Analog: Klassen als Binaries laden:

  ```
  CREATE OR REPLACE JAVA CLASS USING
        BFILE(directory_object, filename);
  CREATE OR REPLACE JAVA CLASS USING
        {BLOB|CLOB|BFILE} subquery;
  ```


Einbinden des Java-Codes in PL/SQL-Funktion/Prozedur

Innerhalb der Datenbank:

- Funktion als Wrapper (*call spec*):

```
CREATE OR REPLACE FUNCTION greet (person IN VARCHAR2)
  RETURN VARCHAR2 AS
LANGUAGE JAVA
NAME 'Greet.sayHello (java.lang.String)
      return java.lang.String';
/
```

[Filename: Java/Greet.sql]

(Bei void-Methoden: Prozedur als Wrapper)

- Aufruf: `SELECT greet('Jim') FROM DUAL;`

GREET('JIM')
Hello Jim!

- Um die Java-Ausgabe auch zu bekommen, muss man sowohl das Java-Output-Buffering als auch den SQL-Output aktivieren:

  ```
  CALL dbms_java.set_output(2000);
  SET SERVEROUTPUT ON;
  ```

  Beispiel: `SELECT greet(name) FROM COUNTRY;`

Syntax des Prozedur/Funktions-Wrappers

```
CREATE [OR REPLACE]
  { PROCEDURE <proc_name>[(<parameter-list>)]
   | FUNCTION <func_name>[(<parameter-list>)]
      RETURN sql_type}
  {IS | AS} LANGUAGE JAVA
  NAME '<java_method_name>[(<java-parameter-list>)]
   [return <java_type_fullname>]';
  /
```

- Bei void-Methoden: Prozeduren,
- Bei non-void-Methoden: Funktionen,
- Die <parameter-list> muss der <java-parameter-list> entsprechen:
  - gleiche Länge,
  - sich entsprechende Parameter-Typen;
    Parameter-Typ-Mapping: siehe Abschnitt über JDBC
- Achtung: In der Namensspezifikation muss return klein geschrieben werden,
- Aufruf des Wrappers eingebettet aus SQL und PL/SQL in Anfragen, DML-Operationen, Prozeduren, Triggern, ...

Soweit ist noch kein Datenbank-Zugriff aus den Methoden möglich. Dies wird durch JDBC ermöglicht (siehe Folie 310).

Nullwerte an Java übergeben

- wenn das Argument NULL ist, ist es in Java `null`,
- `return null` wird als SQL NULL-Wert interpretiert.

```
CREATE OR REPLACE JAVA SOURCE NAMED "Hello"
AS
public class Greet
{ public static String sayHello (String name)
  { System.out.println("This is Java");
    if (name != null)
       return "Hello " + name + "!";
    else return null;
  }
};
/
```

[Filename: Java/Greet-Null-Create.sql]

- wie vorher per Wrapper eine PL/SQL-Prozedur daraus erzeugen,
- `SELECT greet(NULL) FROM DUAL`
- Anmerkung: in Oracle problemlos, in DB2 muss CREATE PROCEDURE mit GENERAL WITH NULLS bzw. SIMPLE WITH NULLS spezifiziert werden (→ Doku)

11.2 Java-Objekttypen

Man kann Java-Klassen als SQL-Typen registrieren. Die Java-Klasse muss das Interface `java.sql.SQLData` implementieren.
Methoden:

- `public String getSQLTypeName()`
  liefert den entsprechenden SQL-Datentyp zurück
- `public void readSQL(SQLInput stream, String typeName) throws SQLException`
  liest Daten aus der Datenbank und initialisiert das Java-Objekt
- `public void writeSQL(SQLOutput stream)`
  bildet das Java-Objekt auf die Datenbank ab.
  (vgl. Marshalling/Unmarshalling zwischen XML und Java in JAXB)

Diese drei Methoden werden nachher nicht vom Benutzer, sondern intern bei der Umsetzung aufgerufen.

BEISPIEL: JAVA-KLASSE `GeoCoordJ`

```
import java.sql.*;

public class GeoCoordJ implements java.sql.SQLData {
 private double longitude, latitude;

  public String getSQLTypeName() {
    return "SCOTT.GEOCOORD";
 }

 public void readSQL(SQLInput stream, String typeName)
   throws SQLException {
     longitude = stream.readDouble();
     latitude = stream.readDouble();
 }

 public void writeSQL(SQLOutput stream)
   throws SQLException {
     stream.writeDouble(longitude);
     stream.writeDouble(latitude);
 } //... to be continued
```

- `SCOTT.GEOCOORD`: Name des SQL-Typs in Oracle
- Felder lesen/setzen in der Reihenfolge der SQL-Definition
- Lese-/Schreibmethoden: `stream.read/write<type>`

BEISPIEL CONT.: GeoCoordJ

```
//... continued
  public double distance(GeoCoordJ other) {
    return
      6370 *
      Math.acos(
         Math.cos(this.latitude/180*3.14) *
         Math.cos(other.latitude/180*3.14) *
         Math.cos(
          (this.longitude - other.longitude)
             /180*3.14
         ) +
         Math.sin(this.latitude/180*3.14) *
         Math.sin(other.latitude/180*3.14)
      );
  }
}
```

[Filename: Java/GeoCoordJ.java]

```
dbis@s042> loadjava -r GeoCoordJ.java
```


SQL-WRAPPER-TYPE

```
CREATE OR REPLACE TYPE geocoord
AS OBJECT EXTERNAL
NAME 'GeoCoordJ'
LANGUAGE JAVA USING SQLData(
longitude number external name 'longitude',
latitude number external name 'latitude',
MEMBER FUNCTION distance (other IN GeoCoord)
  RETURN NUMBER EXTERNAL
  NAME 'distance (GeoCoordJ) return double');
/
CREATE TABLE geoTable OF geocoord;
INSERT INTO geoTable VALUES (geocoord(10,20));
INSERT INTO geoTable VALUES (geocoord(20,30));

SET SERVEROUTPUT ON
CALL dbms_java.set_output(2000);

SELECT g.distance(geocoord(0,51)) FROM geoTable g;
```

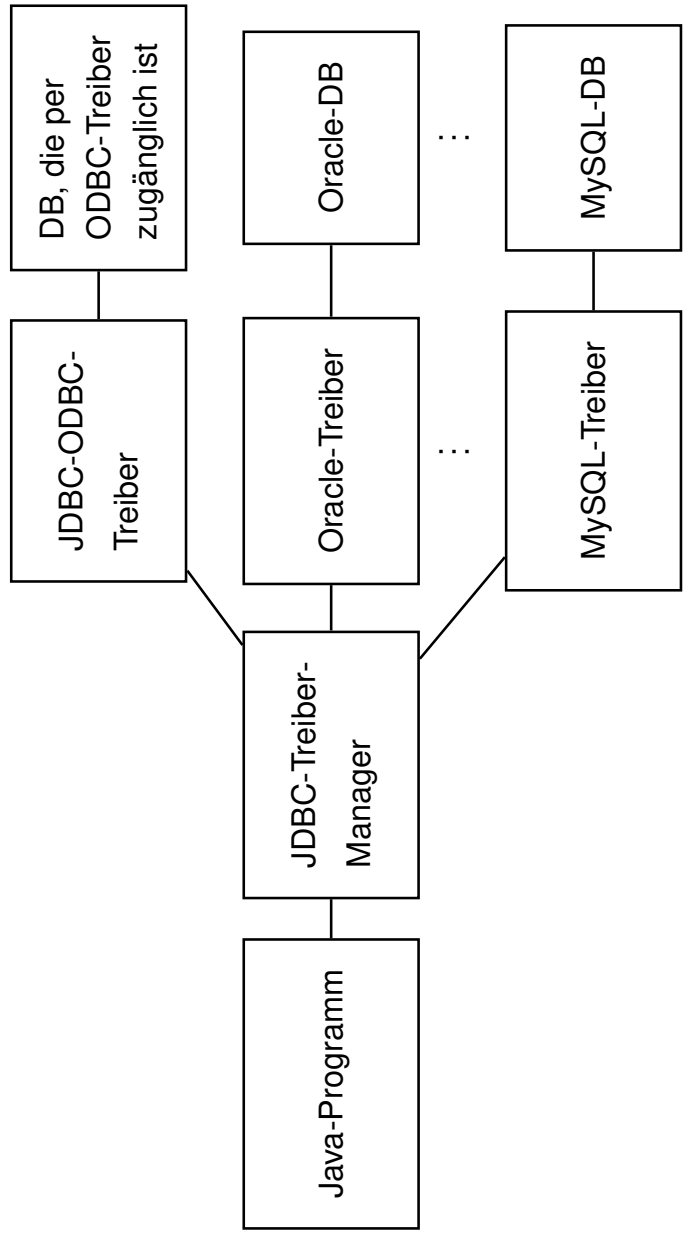
[Filename: Java/GeoCoordJ.sql]

```
  G.DISTANCE(GEOCOORD(0,51))
--------------------------
                  5530.99686
                  4708.68206
```

11.3 JDBC (*Java Database Connectivity*): API für Low-Level-Datenbankzugriff

- Interface für den (entfernten) Datenbankzugriff von Java-Programmen aus,
- Teil des SDK (`java.sql.*`),
- Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden,
- setzt die Idee von ODBC (Open DataBase Connectivity; ein 1992 entwickelter Standard zum Zugriff auf Datenbanken aus Programmiersprachen) auf Java um,
- gemeinsame Grundlage ist der X/Open SQL CLI (Call Level Interface) Standard.

JDBC-Architektur



JDBC-ARCHITEKTUR

- Kern: Treiber-Manager (`java.sql.DriverManager`)
- darunter: Treiber für einzelne DBMS'e

JDBC-API

- flexibel:
  - Applikation kann unabhängig vom darunterliegenden DBMS programmiert werden
- "low-level":
  - Statements werden durch Strings übertragen
  - im Gegensatz zu SQLJ (später) keine Verwendung von Programmvariablen in den SQL-Befehlen (d.h. Werte müssen explizit eingesetzt werden)

Darauf aufbauend:

- Embedded SQL für Java (SQLJ)
- 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`, `PreparedStatement` und `CallableStatement`)
- Verarbeitung der Ergebnismenge (`ResultSet`)

JDBC-Treiber-Manager

`java.sql.DriverManager`

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

- Es wird nur ein `DriverManager` benötigt.

$\Rightarrow$ 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.

Verbindungsaufbau

- `DriverManager` erzeugt offene Verbindungs-Instanz:

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

  oder

  ```
  DriverManager.getConnection(<jdbc-url>, <props>);
  ```

  (Login-Daten in externer Datei, `java.util.Properties`).

- Datenbank wird eindeutig durch JDBC-URL bezeichnet

JDBC-URL:

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

Bei uns:

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

```
String url =
  'jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis';
```

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 Parametern
- `CallableStatement`: Aufruf von gespeicherten Prozeduren

KLASSE "STATEMENT"

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

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

- `ResultSet <statement>.executeQuery(<string>)`: *SQL-Anfragen* an die Datenbank. Dabei wird eine Ergebnismenge zurückgegeben.
- `int <statement>.executeUpdate(<string>)`: SQL-Statements, die eine Veränderung an der Datenbasis vornehmen (einschliesslich CREATE PROCEDURE etc). Der Rückgabewert gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren.
- `boolean <statement>.execute(<string>)` – Sonstiges:
  - Generierung und Aufrufe von Prozeduren/Funktionen (siehe CallableStatements),
  - Statement dynamisch als String erzeugt, und man weiß nicht, ob es eine Query oder ein Update ist,
  - "true" wenn das (erste) Ergebnis ein ResultSet ist; "false" sonst (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" (Iterator-Pattern):

```
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 (bzw. am Anfang auf das erste) 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,

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

  `<getString>` funktioniert immer (*type casting*).

- `<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.

Beispiel-Code

```
import java.sql.*;
class jdbcCities {
public static void main (String args [])
      throws SQLException {
  // ORACLE-Treiber laden
  DriverManager.registerDriver
      (new oracle.jdbc.driver.OracleDriver());
  // Verbindung zur Datenbank herstellen
  String url =
   "jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis";
  Connection conn =
   DriverManager.getConnection(url,"scott","tiger");
  // 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");
   }
  conn.close();
  }}
```

[Filename: Java/jdbcCities.java]

BEHANDLUNG VON 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 in `java.sql.Types`
`java.lang.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

Im Fall von allgemeinen Anfragen weiß man oft nicht, wieviele Spalten eine Ergebnismenge hat, wie sie heißen, und welche Typen sie haben.

Instanz der Klasse `ResultSetMetaData` enthält Metadaten über das vorliegende ResultSet:

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

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

- `int getColumnCount()`:
  Spaltenanzahl der Ergebnismenge
- `String getColumnLabel(int)`:
  Attributname der Spalte <int>
- `String getTableName(int)`:
  Tabellenname 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(int 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.

Beispiel: Auslesen einer beliebigen Tabelle

```
import java.sql.*;
class jdbcSelect {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");
 Statement stmt = conn.createStatement();
 ResultSet rset =
   stmt.executeQuery("SELECT * FROM " + args[0]);
 ResultSetMetaData rsetmetadata = rset.getMetaData();
 int numCols = rsetmetadata.getColumnCount();
 while (rset.next()) {
  for(int i=1; i<=numCols; i++) {
     String value = rset.getString(i);
     if (rset.wasNull()) System.out.print("null");
     else System.out.print(value);
     System.out.print("  ");
  }
  System.out.println();
 }
 rset.close(); stmt.close(); conn.close();
}}
```

[Filename: Java/jdbcSelect.java]

```
dbis@c42> java jdbcSelect City
```


Auslesen von Objekten

```
dbis@c42> java jdbcSelect Mountain
```

- Für Instanzen von Objekttypen wird immer “null” ausgegeben.
- Objekte mit getObject auslesen
- Objekte sind vom Typ oracle.sql.STRUCT
  → testen und ggf. casten
- Objekttyp-Name: String name = x.getSQLTypeName()
- attribute: Object[] attrs = x.getAttributes()
  enthält dann Strings, Zahlwerte, oder wieder Objekte

Beispiel: Auslesen von Objekten

[Filename: Java/jdbcSelectObj.java]

```
import java.sql.*;
class jdbcSelectObj {
  public static void main (String args []) throws SQLException {
   Connection conn = getConn();
   Statement stmt = conn.createStatement();
   ResultSet rset =
     stmt.executeQuery("SELECT * FROM " + args[0]);
   ResultSetMetaData rsetmetadata = rset.getMetaData();
   int numCols = rsetmetadata.getColumnCount();
   while (rset.next()) {
    for(int i=1; i<=numCols; i++) {
     Object value = rset.getObject(i);
     if (rset.wasNull()) System.out.print("null  ");
     else if (value instanceof oracle.sql.STRUCT)
     { System.out.print("(" +
         ((oracle.sql.STRUCT)value).getSQLTypeName() + ": ");
       Object[] attributes =
         ((oracle.sql.STRUCT)value).getAttributes();
       // attributes
       for (int j = 0; j < attributes.length; j++)
            System.out.print(attributes[j] + "  ");
       System.out.print(")");
     }
     else System.out.print(value + "  "); }
    System.out.println();  }
   rset.close(); stmt.close(); conn.close();  }
 private static Connection getConn() throws SQLException
 {
   DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
   String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
   return DriverManager.getConnection(url,"scott","tiger");
```


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 giveCountryPop =
  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.

```
giveCountryPop.setString(1,"D");
ResultSet rset = giveCountryPop.executeQuery();
if (rset.next()) System.out.print(rset.getInt(1));
giveCountryPop.setString(1,"CH");
ResultSet rset = giveCountryPop.executeQuery();
if (rset.next()) System.out.print(rset.getInt(1));
```


PreparedStatement (Cont’d)

- Nullwerte werden gesetzt durch

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

  `<sqlType>` bezeichnet den JDBC-Typ dieser Spalte.
- nicht sinnvoll in Anfragen (Abfrage nicht mit “= NULL” sondern mit “IS NULL”), sondern z.B. Bei INSERT-Statements oder Prozeduraufrufen etc.

Beispiel: PreparedStatement

```
import java.sql.*;
class jdbcCountryPop {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 PreparedStatement giveCountryPop =
   conn.prepareStatement(
      "SELECT Population FROM Country WHERE Code = ?");
 giveCountryPop.setString(1,args[0]);
 ResultSet rset = giveCountryPop.executeQuery();
 if(rset.next()) {
  int pop = rset.getInt(1);
  if (rset.wasNull()) System.out.print("null");
  else System.out.print(pop);
  }
 else System.out.print("Kein zulaessiger Landescode");
 System.out.println();
 conn.close();
}}
```

[Filename: Java/jdbcCountryPop.java]

```
dbis@c42> java jdbcCountryPop D
dbis@c42> java jdbcCountryPop X
```


Erzeugen von Funktionen, Prozeduren etc.

- 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);
```

Callable Statements: Gespeicherte Prozeduren

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

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.NUMERIC);
```


CALLABLE STATEMENTS MIT PARAMETERN

- Vorbereitung (s.o.)

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

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

```
cstmt.setString(2,"Gottingen");
cstmt.setString(3,"Berlin");
```

- Aufruf mit

```
cstmt.execute();
```

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

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

Beispiel: CallableStatement

```
import java.sql.*;
class jdbcCallProc {
  public static void main (String args [])
   throws SQLException {
 DriverManager.registerDriver(new
         oracle.jdbc.driver.OracleDriver());
 String url = "jdbc:oracle:thin:/*hier korrekt fortsetzen
 Connection conn =
    DriverManager.getConnection(url,"scott","tiger");

 CallableStatement call =
      conn.prepareCall("{? = call greet(?)}");
 call.registerOutParameter(1,java.sql.Types.VARCHAR);
 call.setString(2,args[0]);
 call.execute();
 String answer = call.getString(1);
 System.out.println(answer);
 conn.close();
}}
```

[Filename: Java/jdbcCallProc.java]

Wenn die Funktion “Greet” (vgl. Folie 303) für den User scott/tiger verfügbar ist:

```
dbis@c42> java jdbcCallProc Joe
```


ABFRAGEN VON ERGEBNISSEN BEI EXECUTE()

Häufig: `<string>` dynamisch generiert.

- erste Annahme: liefert nur ein Ergebnis.
- `boolean <statement>.execute(<string>)`,
  `boolean <prepared-statement>.execute()`,
  `boolean <callable-statement>.execute()`
- “true” wenn das (erste) Ergebnis ein ResultSet ist; “false” sonst.
- `ResultSet getResultSet()`: Falls das (erste) Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis ein Update-Zähler ist: `null` zurückgeben.
- `int getUpdateCount()`: Falls das (erste) Ergebnis ein Update-Zähler ist, wird dieser ($n \geq 0$) zurückgegeben; falls kein Ergebnis mehr vorhanden, oder das (erste) Ergebnis eine Ergebnismenge ist, wird -1 zurückgegeben.

FOLGE VON ERGEBNISSEN VERARBEITEN

- SQL-Statements, die mehrere Ergebnisse nacheinander zurückliefern
  Hinweis: Statement wie oben darf keinen Strichpunkt enthalten, darf also keine Sequenz sein.
  $\Rightarrow$ dieser Fall tritt also nur seltenst ein ...
  [Rest der Folie ist optional]
- `getMoreResults()`: schaltet zum nächsten Ergebnis. `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 (2)

```
stmt.execute(StatementWithUnknownResultSequence);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) { // update, n Tupel geaendert
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue; }
    if (rowCount == 0) { // update, aber nichts geaendert
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue; }
    // sonst: query
    ResultSet rs = stmt.getResultSet();
    if (rs != null) {
       .....  // verarbeite Metadaten
       while (rs.next())
          { ....}  // verarbeite Ergebnismenge
       stmt.getMoreResults();
       continue;
    }
    break;
}
```

FOLGE VON STATEMENTS VERARBEITEN

- man kann natürlich jedes einzeln abschicken ...

executeBatch()

- `<statement>.addBatch(<string>)`: Statement (keine Query) zum Batch dazunehmen,
- ... beliebig oft ... und dann
- `int[] <statement>.executeBatch()`: alle Statements ausführen; ergibt ein Array mit updateCount-Werten,
- `clearBatch()`

TRANSAKTIONSSTEUERUNG

Sei `con` eine Instanz der Klasse `Connection`.
Per Default ist für eine `Connection` der Auto-Commit-Modus gesetzt:

- implizites Commit nach jeder ausgeführten Anweisung (Transaktion besteht also nur aus einem Statement)
- `con.setAutoCommit(false)` schaltet den Auto-Commit-Modus aus und man muss explizite Commits ausführen.

Dann hat man die folgenden Methoden:

- `con.setSavepoint(String name)` (setzt Sicherungspunkt)
- `con.commit()` (macht Änderungen persistent)
- `con.rollback([<savepoint>])` (nimmt alle Änderungen [bis zu `<savepoint>`] zurück.

11.4 JDBC in Java Stored Procedures

In Stored Procedures verwendet man ebenfalls JDBC-Technologie mit dem serverseitigen JDBC-Treiber von Oracle (`jdbc:default:connection:`). User/Password werden natürlich nicht angegeben, da es bereits in der DB abläuft:

```
import java.sql.*;
public class getCountryData{
 public static void getPop (String code)
       throws SQLException {
  String sql =
    "SELECT name,population FROM country WHERE code = ?";
  try {
    Connection conn = DriverManager.getConnection
                           ("jdbc:default:connection:");
    PreparedStatement pstmt = conn.prepareStatement(sql);
    pstmt.setString(1, code);
    ResultSet rset = pstmt.executeQuery();
    if (rset.next());
       System.out.println(rset.getString(2));
    rset.close();
    pstmt.close();
  }
  catch (SQLException e) {
    System.err.println(e.getMessage());
  }}}
```

[Filename: Java/getCountryData.java]

JAVA-KLASSE IN PL/SQL-PROZEDUR EINBINDEN

Laden in die Datenbank:

```
loadjava -u user/passwd -r getCountryData.java
```

Definition und Ausführung des Wrappers in der DB:

```
CREATE PROCEDURE getPopulation (code IN VARCHAR2) IS
  LANGUAGE JAVA
  NAME 'getCountryData.getPop(java.lang.String)';
/
```

[Filename: Java/getCountryData.sql]

... Output aktivieren:

```
SET SERVEROUTPUT ON;
CALL dbms_java.set_output(2000);

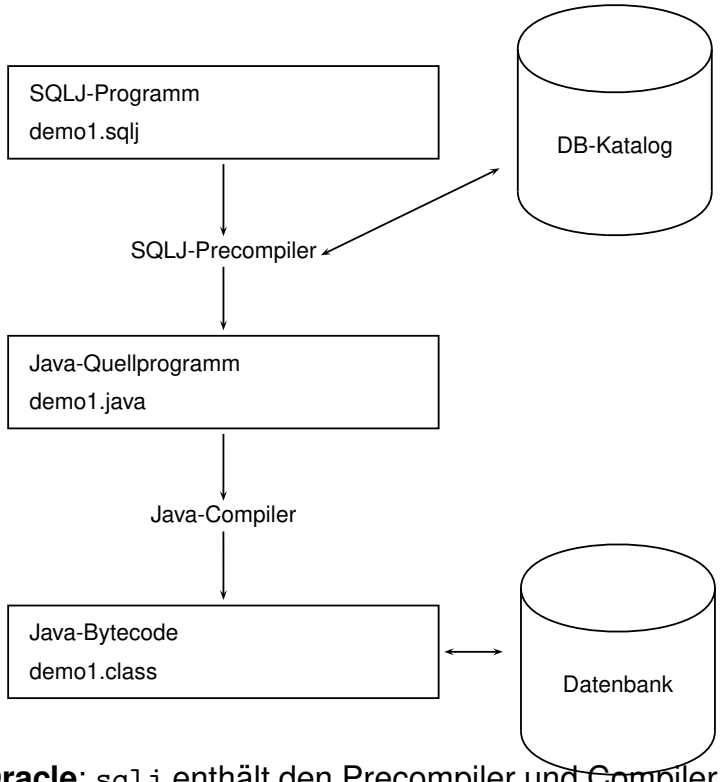
EXEC getPopulation('D');
83536115
```

11.5 SQLJ

Realisierung des “Embedded SQL”-Konzeptes für Java:

- Standardisierte Java-DB Schnittstelle, verwendet JDBC
- ANSI-Standard als Teil des SQL-Standards, beteiligt waren u.a. Oracle, Sun, IBM, Microsoft (ANSI x.3.135.10-1998)
- Besteht aus drei Teilen:
  - Part 0: Embedded SQL in Java.
  - Part 1: SQL routines using Java
    (siehe Abschnitt “Java in Stored Procedures”).
  - Part 2: SQL types using Java
    (Java-Klassen als SQL Datentypen (u.a. $\rightarrow$ XMLType)).
- SQLJ Part 0: SQL-in-Java
- SQLJ Part 1 & 2: Java-in-SQL
- Eingebettete SQLJ-Aufrufe werden als pures Java übersetzt und werden auf JDBC-Aufrufe abgebildet.
- Hier: Beschränkung auf SQLJ Part 0, Part 1 als Java Stored Procedures

ENTWICKLUNG EINER SQLJ-APPLIKATION



- **Oracle**: `sqlj` enthält den Precompiler und Compiler.
  Der Aufruf von `sqlj demo1.sqlj` erzeugt `demo1.java` und `demo1.class`.
- die Quelldatei muss die Endung `.sqlj` haben.
- Wenn man `demo1.java` anschaut, findet man die Umsetzung via JDBC.

ANWEISUNGEN IN SQLJ

- Anfragen:

```
#sql anIterator
  = {SELECT name, population FROM country};
```

  wobei anIterator ein (auch per SQLJ) geeignet definierter Iterator ist.

- DML und DDL:

```
#sql{<statement>};
```

- Prozeduraufrufe:

```
#sql{CALL <proc_name>[(<parameter-list>)]};
```

- Funktionsaufrufe:

```
#sql <variable>=
  {VALUES(<func_name>[(<parameter-list>)])};
```

- Aufruf unbenannter Blöcke:

```
#sql {BEGIN ... END};
```


VERBINDUNGSAUFBAU ZU ORACLE

Ausführliche Variante

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
//--------------
import sqlj.runtime.*;
import sqlj.runtime.ref.DefaultContext;
  :
String url =
  "jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis";
String user = "...";
String passwd = "...";
DriverManager.registerDriver
  (new oracle.jdbc.driver.OracleDriver());
Connection con =
  DriverManager.getConnection(url,user,passwd);
DefaultContext ctx = new DefaultContext(con);
DefaultContext.setDefaultContext(ctx);
Oracle.connect(url, user, passwd);
//--------------
```

VERBINDUNGSAUFBAU ZU ORACLE

Kompaktere Variante

- `connect.properties` ist eine Datei (bzw. Datei im .jar-Archiv), die folgendermassen aussieht:

```
#connect.properties:
sqlj.url=jdbc:oracle:thin:@xxx.xxx.xxx.xxx:1521:dbis
sqlj.user=<user>
sqlj.password=<passwd>
```

[Filename: Java/connect.properties – muss jeder selber schreiben]

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;
  :
Oracle.connect(<JavaClass>.class, "connect.properties");
  :
```

- `<JavaClass>.class` ist eine Klasse, die im Dateisystem/jar-Archiv im selben Verzeichnis wie `connect.properties` liegt (der Name dieser Klasse dient nur dazu, `connect.properties` zu finden!).

HOSTVARIABLEN

- Verwendung von Variablen einer Host-Sprache (hier Java) in SQL-Statements
- Dient dem Datenaustausch zwischen Datenbank und Anwendungsprogramm
- in SQLJ-Statements wird Hostvariablen ein Doppelpunkt (":") vorangestellt
- Datentypen der Datenbank- und Programmiersprache müssen kompatibel sein (siehe JDBC)

In Host-Variablen schreiben:

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

Aus Host-Variablen lesen:

```
int population = 75000000;
#sql{UPDATE country SET population = :population
      WHERE code='D'};
```

ITERATOREN

- Allgemein: Design-Pattern, sequenzieller Zugriff auf alle Objekte, die in einem Container enthalten sind
- Hier: Iteratoren bilden das Cursor-Konzept auf SQLJ ab.
- Iteratoren mit benannten Spalten:
  - Spaltenzugriff über Spaltennamen
  - Weiterschaltung mit next()
- Positionsiteratoren:
  - Spaltenzugriff über Position
  - Weiterschaltung mit FETCH ... INTO

ITERATOREN MIT BENANNTEN SPALTEN

Hierbei erhalten die Attribute des Iterators Namen (“Schema”):

```
import java.sql.*;
import oracle.sqlj.runtime.Oracle;

class sqljIteratorExample {
 public static void main (String args []){
  // Datenbank-Verbindung aufbauen
  try {
   Oracle.connect(sqljIteratorExample.class, "connect.properties"
   // Deklaration des benannten Iterators mit Spaltennamen
   #sql iterator CountryIter(String name, int population);
   // Iteratorobjekt wird definiert
   CountryIter cIter;
   // Initialisieren des Iterators mit der SQL-Anweisung
   #sql cIter = {SELECT name, population FROM country};
   // Abarbeitung der Ergebnismenge durch Iteration
   while (cIter.next()) {
      System.out.println(cIter.name() + " has " +
                cIter.population() + " inhabitants."); }
   // Schliessen des Iterators
   cIter.close();
  }
  catch (SQLException e) {
    System.err.println(e.getMessage()); }
 }}
```

[Filename: Java/sqljIteratorExample.sqlj]

POSITIONSITERATOREN

```
// Verbindungsaufbau, Deklaration Positionsiterator
Oracle.connect(Countries.class, "connect.properties");
#sql iterator CountryPosIterator(String, int);
// Hilfsvariablen
String name = "";
int pop = 0;

// Iteratorobjekt wird definiert
CountryPosIterator cIter;
// Initialisieren des Iterators mit der SQL-Anweisung
#sql cIter = {SELECT name, population FROM country};

// Abarbeitung der Ergebnismenge durch Iteration
while (true) {
   //hole naechsten Datensatz
   #sql{FETCH :cIter INTO :name,:pop};
   //Ende des Iterators erreicht?
   if(cIter.endFetch()) break;
   System.out.println(name + " has " +
                      pop + " inhabitants.");
}
// Schliessen des Iterators
cIter.close();
```


VERGLEICH: JDBC UND SQLJ

JDBC

- Call-Level-Schnittstelle
- Dynamisches SQL
- Fehlererkennung erst zur Laufzeit
- Hohe Flexibilität

```
int countries;
Statement stmt = con.createStatement();
String query = "SELECT COUNT(*) FROM country";
ResultSet rset = stmt.executeQuery(query);
rset.next();
countries = rset.getInt(1);
```

SQLJ

- Embedded SQL
- Statisches SQL
- Fehlererkennung bereits zur Übersetzungszeit
- Kompakte Syntax

```
int countries;
#sql{SELECT COUNT(*) INTO :countries FROM country};
```

11.6 Weitere SQL/Oracle-Werkzeuge

- seit ORACLE8i (1999; i= internet)
  Mit eingebauter Java Virtual Machine, Zugriff auf das Filesystem,
  Oracle-Web Server/Internet Application Server (seit 9i): HTML-Seiten werden abhängig vom Datenbankinhalt erstellt.
- mit den Paketen IAS, Internet File System Server wachsen Datenbank und Betriebssystem zunehmend zusammen.
- seit ORACLE9i: Integration aus der XML-Welt (*XMLType*): XPath, XSLT, DOM, XML Schema.
  ... siehe weitere Folien.
- ORACLE10g: grid computing
  Oracle Rules Manager für Aktive Ereignis-basierte Regeln

ENTWICKLUNGSLINIE ORACLE

- 1977: Gründung durch Larry Ellison
- 1979: erstes Produkt
- 1992: Oracle 7
- letzte 7er: 7.3.4: erste SQLJ/JDBC-Version (SQLJ bis 8.1.5 über OTN)
- 1997/1998: Oracle 8 (bis 8.0.4): Objekttypen, Nested Tables
- 3.1999: Oracle 8i/8.1.5 (i = Internet); JVM, Java Stored Procedures & Member Methods, SQLJ
- 2.2001: Oracle 8.1.6: ein bisschen XML-Support (als Java-Tools)
- 6.2001: Oracle 9i: Java-Klassen als Object Types, Vererbung
- 5.2002: 9i-R2/9.2.0: verbesserter XML-Support (XMLType)
  `http://www.oracle.com/technology/tech/xml/xmldb/9.2.0.2.0/NewFeatures.pdf`
- 2003/2004: Oracle 10g (g = Grid)

u.a. von
`http://www.gulp.de/kb/mk/chanpos/oraclereleases.html`
(nicht auswendiglernen)

Kapitel 12
SQL und XML

12.1 XML: “Extensible Markup Language”

... mehr als nur “Language”: Datenmodell, viele Sprachen

- Instantiierung von SGML (vgl. HTML)
- Semantische Tags

⇒ wer HTML kennt, weiß, wie XML “aussieht”.

- Baumstruktur
- Elemente (Name, Attribute und Inhalt)
- rekursiver Aufbau

⇒ abstrakter Datentyp mit Konstruktoren und Operationen.

- Navigation im Baum
- vgl. Pfadausdrücke in Java, OQL, SQL
  (z.B. x.coordinates.longitude)
- Pfadausdrücke in Unix:
  (z.B. /home/may/teaching/dbp/folien.tex)

⇒ Adressierungssprache “XPath”

XML: Beispiel

```
<country id="D" capital="cty-Germany-Stuttgart">
    <name>Germany</name>
    <total_area>356910</total_area>
    <population>83536115</population>
    <encompassed continent="europe">100</encompassed>
    <ethnicgroup name="German">95.1</ethnicgroup>
    <ethnicgroup name="Italians">0.7</ethnicgroup>
    <religion name="Roman Catholic">37</religion>
    <religion name="Protestant">45</religion>
    <language name="German">100</language>
    <border country="F">451</border>
    <border country="A">784</border>
    <border country="CZ">646</border>
    :
```

```
  <province id="prov-Germany-Baden-Wuerttemberg">
    <name>Baden Wuerttemberg</name>
    <area>35742</area>
    <population>10272069</population>
    <city is_state_cap="yes" id="cty-Germany-Stuttgart">
      <name>Stuttgart</name>
      <longitude>9.1</longitude>
      <latitude>48.7</latitude>
      <population year="95">588482</population>
    </city>
    <city id="cty-Germany-Mannheim">
      <name>Mannheim</name>
      :
    </city>
    :
  </province>
  <province id="prov-Germany-Berlin">
    <name>Berlin</name>
    <area>889</area>
    <population>3472009</population>
    <city is_country_cap="yes" is_state_cap="yes"
          id="cty-Germany-Berlin">
      <name>Berlin</name>
      <longitude>13.3</longitude>
      <latitude>52.45</latitude>
      <population year="95">3472009</population>
    </city>
  </province>
  :
</country>
```


XML

(Siehe Vorlesung “Semistrukturierte Daten und XML”)

- Verwendung:
  - Dokumente
  - Datenaustausch
  - Datenspeicherung
- sehr flexibles “Datenmodell”: DOM-API
  rekursiv definierte Baumstruktur aus
  - Elementen,
  - Attributen und
  - Textknoten.
- Schema: DTD (Document Type Description), XML Schema
- Erweiterungen: XPath, XPointer, XLink
- Anfragesprache: XQuery
- Transformationssprache: XSL/XSLT
- Als Repräsentationssprache für viele Anwendungen genutzt

12.2 Der SQL/XML bzw. SQLX Standard – Kombination relationaler Daten und XML

- Abbildung von relationalen Daten nach XML
- Speicherung von XML-Daten in RDBMS
- Entwurf eines ISO-Standards seit 2003: `www.sqlx.org`
- SQL-Objektdatentyp “XMLType”
  - mit entsprechenden Konstruktoren für XML-Strukturen,
  - und Zugriffsmethoden (basierend auf den Standards der XML-Welt),
  - benutzbar von SQL und innerhalb von PL/SQL.
- zum Teil noch unvollständig und überraschend ...
- Folien: siehe Vorlesung “Semistructured Data and XML”