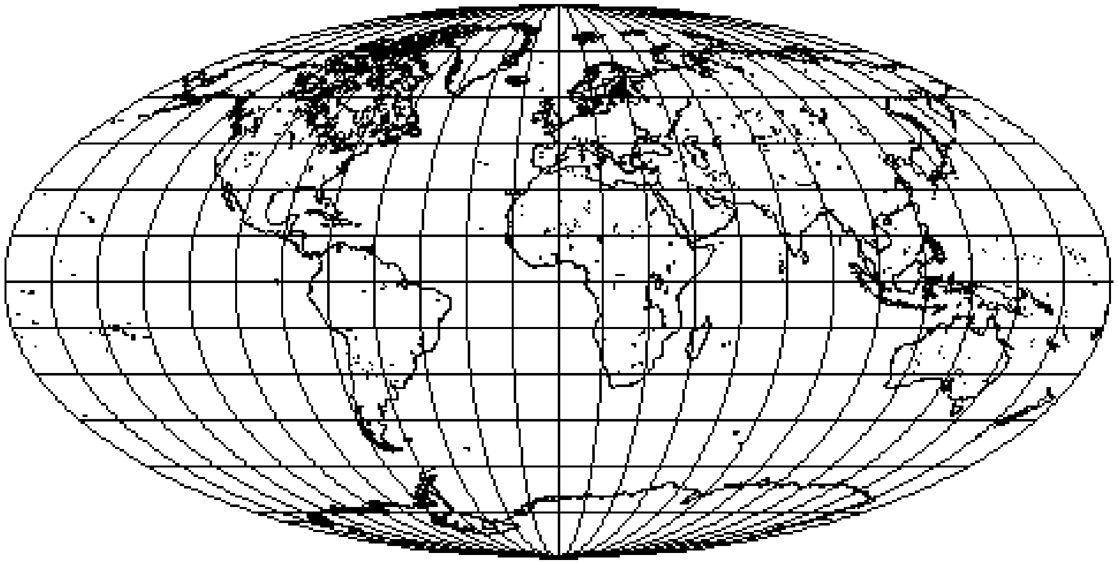
Praktikum: Datenbankprogrammierung in SQL/ORACLE



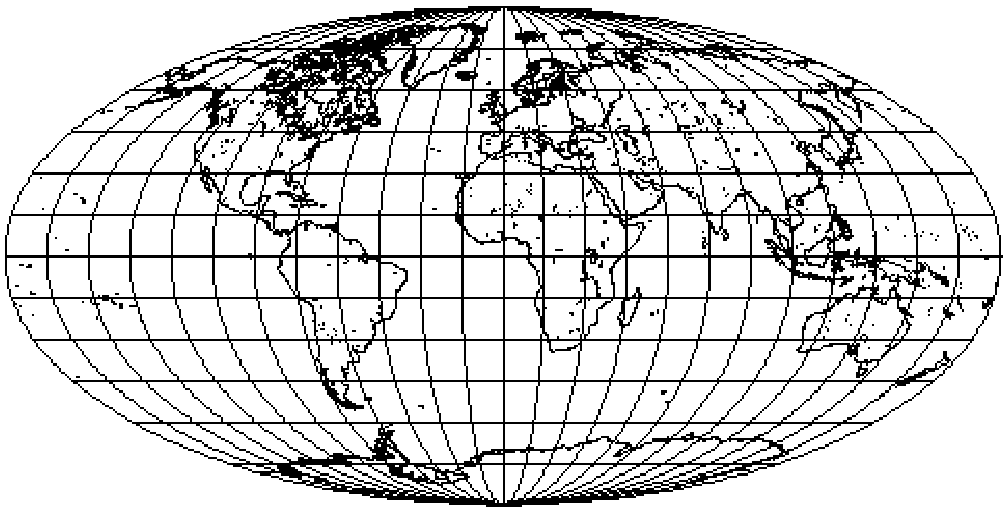
Prof. Dr. Wolfgang May
Universität Göttingen

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
  - Wirtschaft
  - Bevölkerung
  - Sprachen
  - Religionen
  - Ethn. Gruppen
- Länder
- Landesteile
- Städte
- Organisationen

- Flüsse
- Seen
- Meere
- Inseln
- Berge
- Wüsten

- CIA World Factbook
- "Global Statistics": Länder, Landesteile, Städte
- 1987: 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

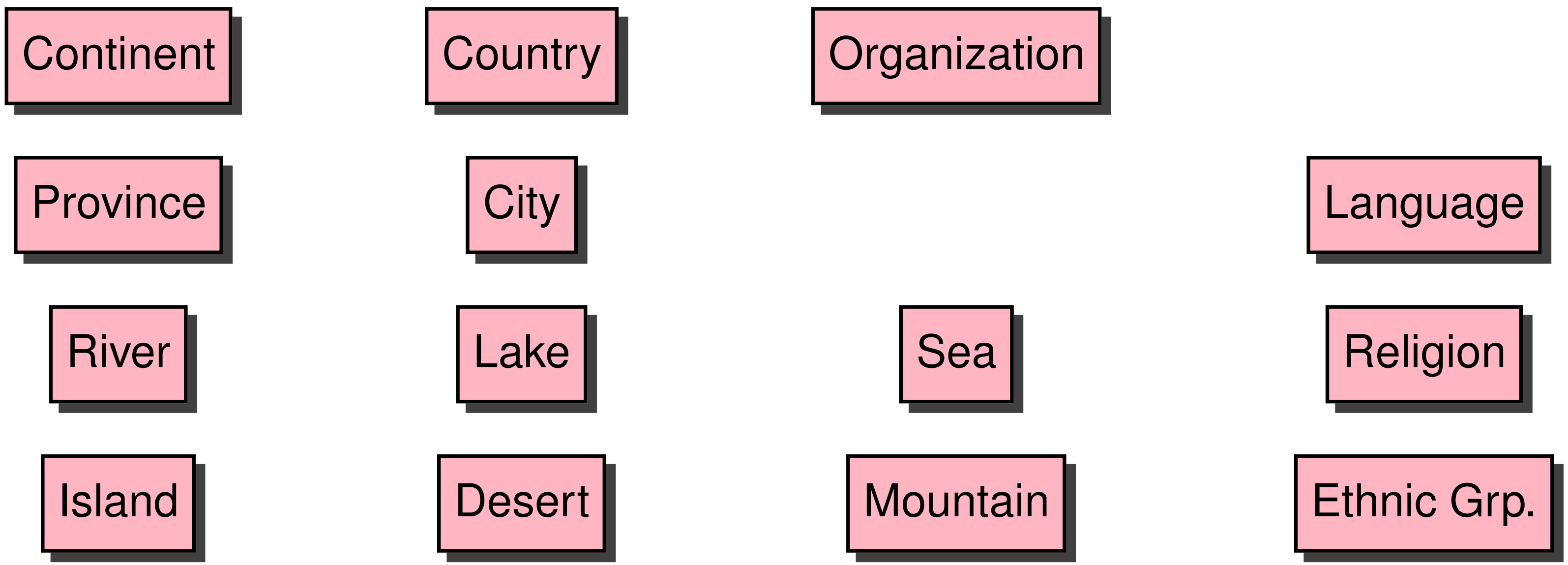
Teil III: Erweiterungen
Prozedurale Konzepte, OO, Einbettung

Kapitel 1
Semantische Modellierung

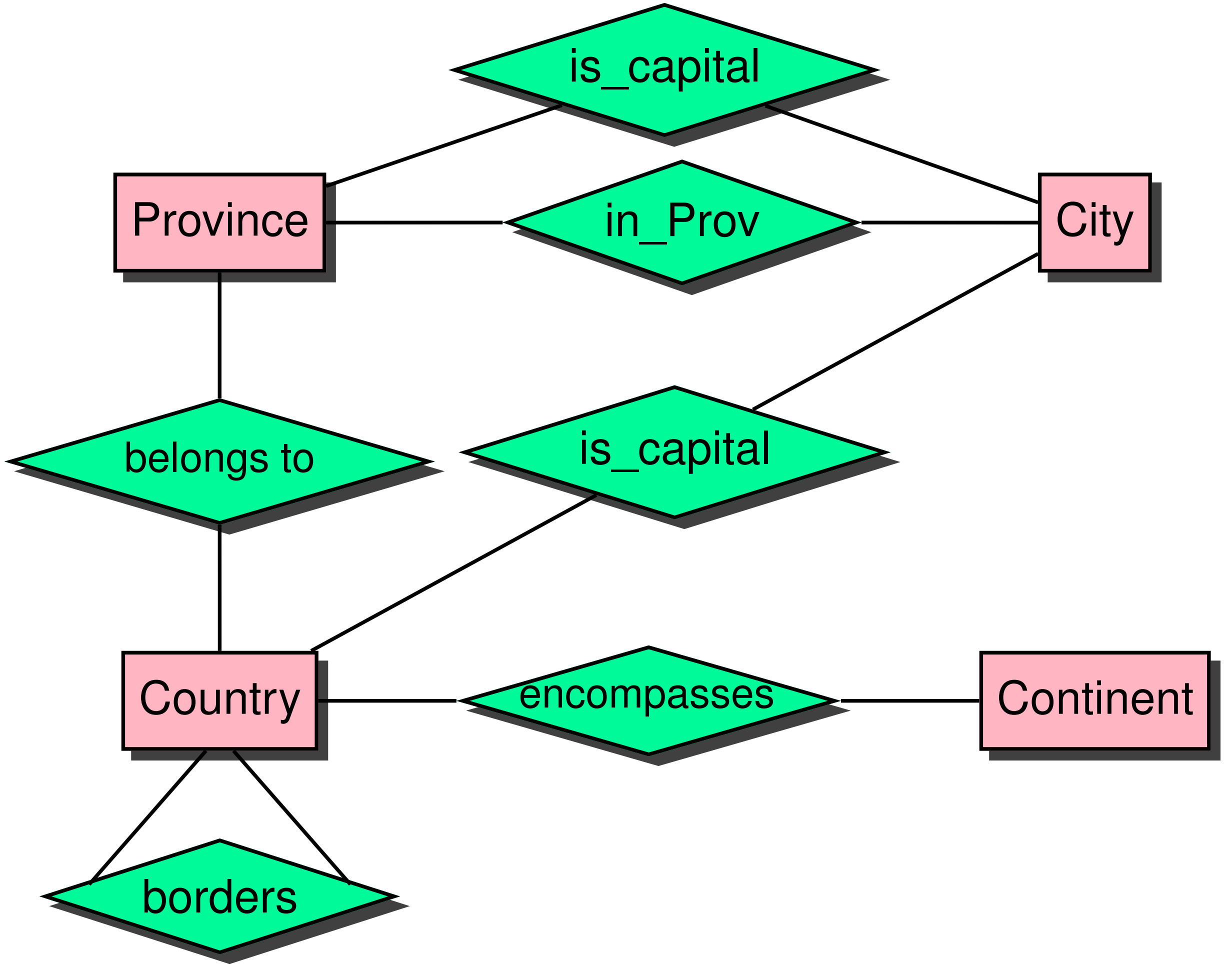
Entity-Relationship-Modell (Chen, 1976)

Strukturierungskonzepte zur Beschreibung eines Schemas im ERM:

- Entitäts– (entity) Typen ($\equiv$ Objekttypen) und
- Beziehungs– (relationship) Typen



Entities und Beziehungen



ENTITIES

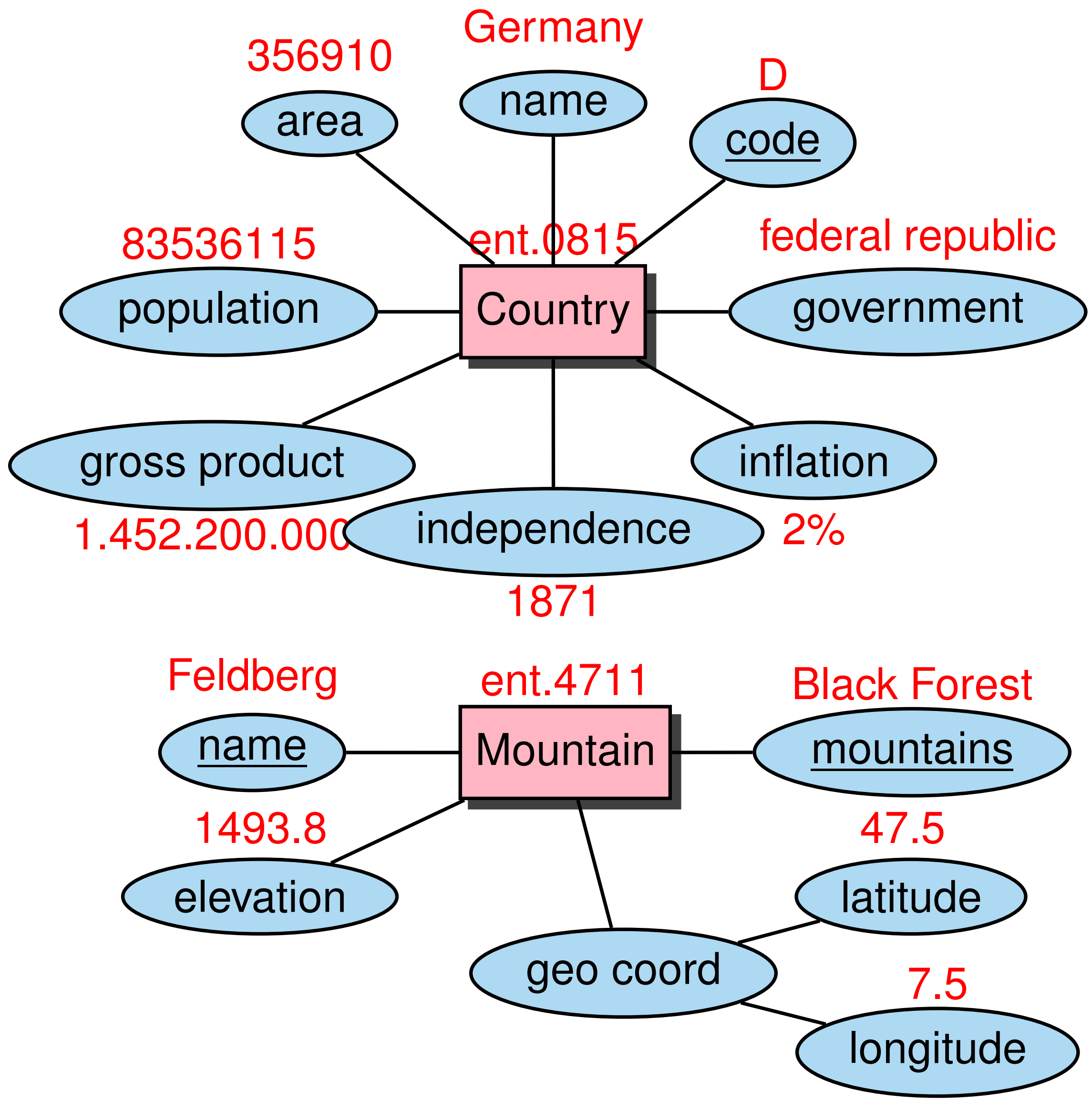
Entitätstyp ist durch ein Paar $(E, \{A_1, \ldots, A_n\})$ gegeben, wobei E der Name und $\{A_1, \ldots, A_n\},\ n \geq 0$, die Menge der Attribute des Typs ist.

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:


356910
Germany
D
area
name
code
83536115
ent.0815
federal republic
population
Country
government
gross product
inflation
1.452.200.000
independence
2%
1871
Feldberg
ent.4711
Black Forest
name
Mountain
mountains
1493.8
47.5
elevation
latitude
geo coord
7.5
longitude

BEZIEHUNGEN

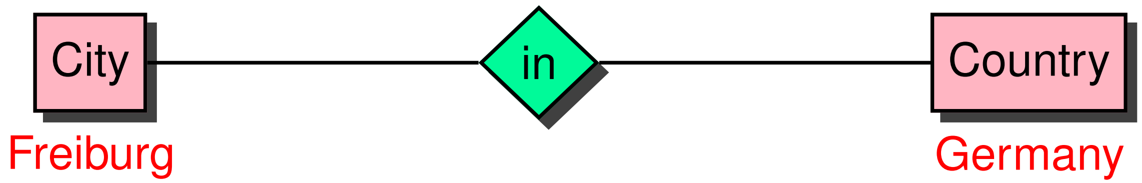
Beziehungstyp: Menge gleichartiger Beziehungen zwischen Entitäten; ein Beziehungstyp ist durch ein Tripel $(B, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_n\})$ gegeben, wobei B der Name, $\{RO_1, \ldots, RO_k\},\ k \geq 2$, die Menge der sog. Rollen, $\{E_1, \ldots, E_k\}$ die den Rollen zugeordnete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

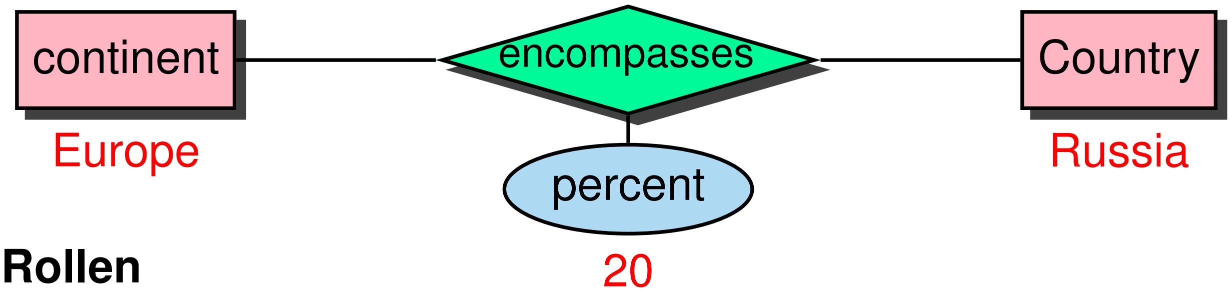
Attribut: Relevante Eigenschaft der Beziehungen eines Typs.

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.

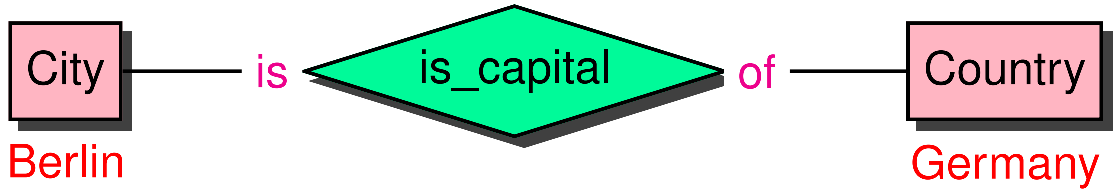
BEZIEHUNGEN



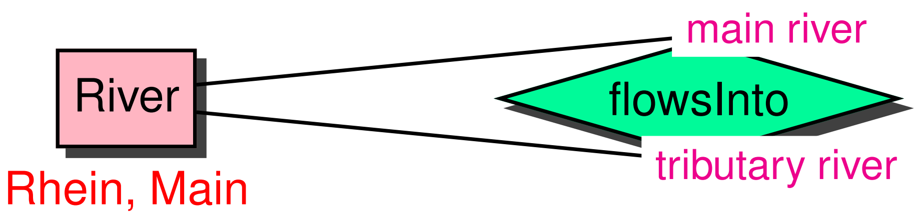
attributierte Beziehung



Beziehung mit Rollen



rekursive Beziehung (nicht symmetrisch, mit Rollen)



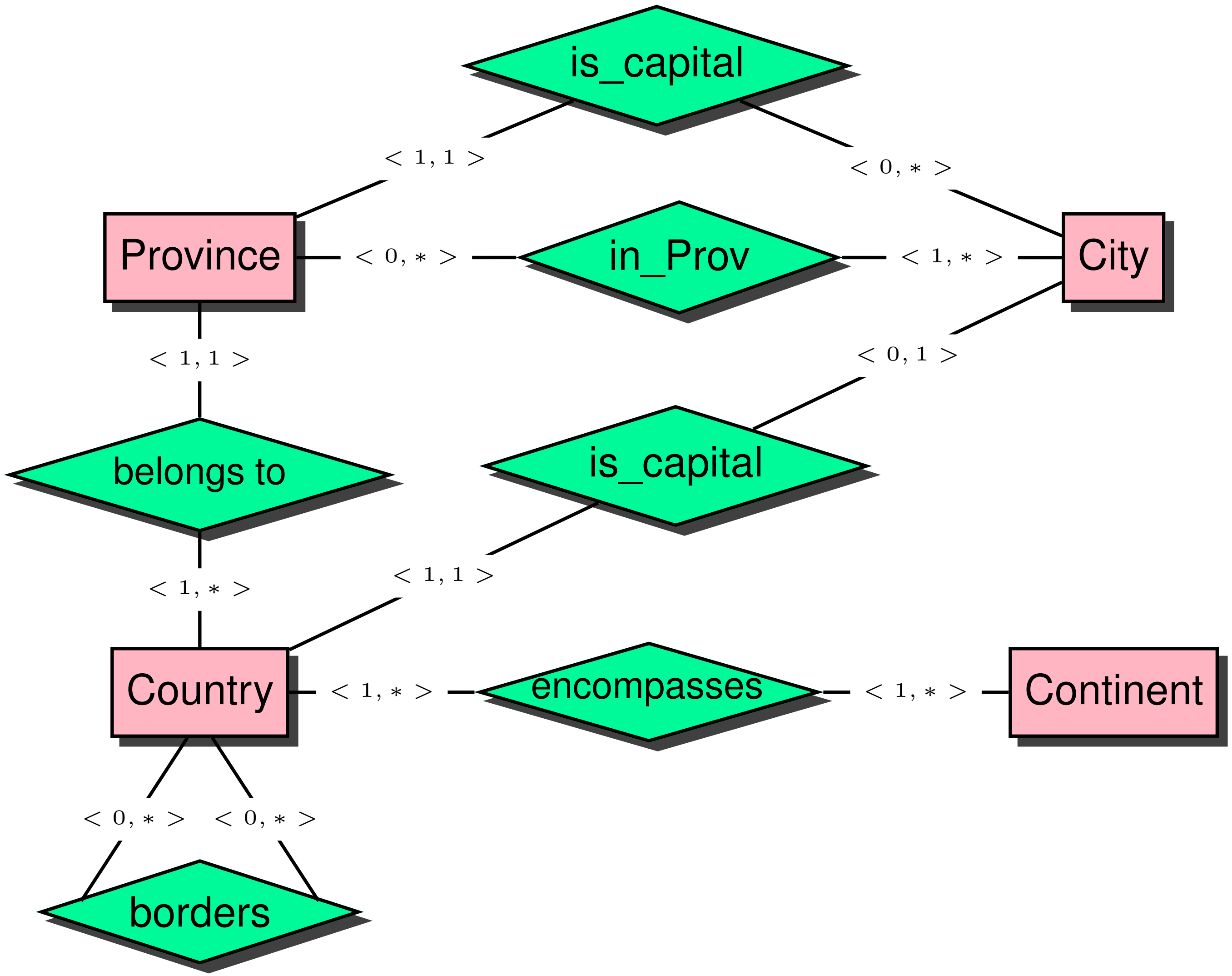
BEZIEHUNGSKOMPLEXITÄTEN

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.

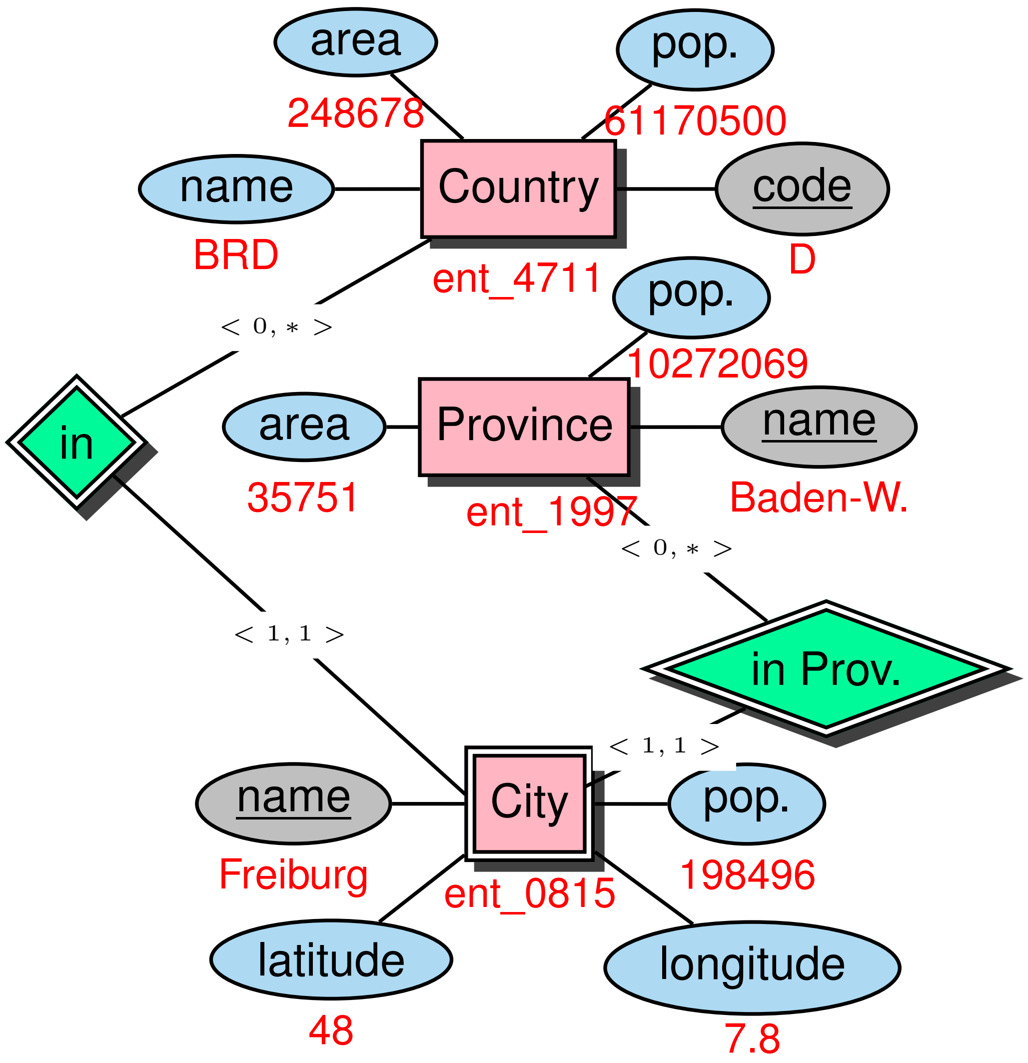
BEZIEHUNGEN



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**).

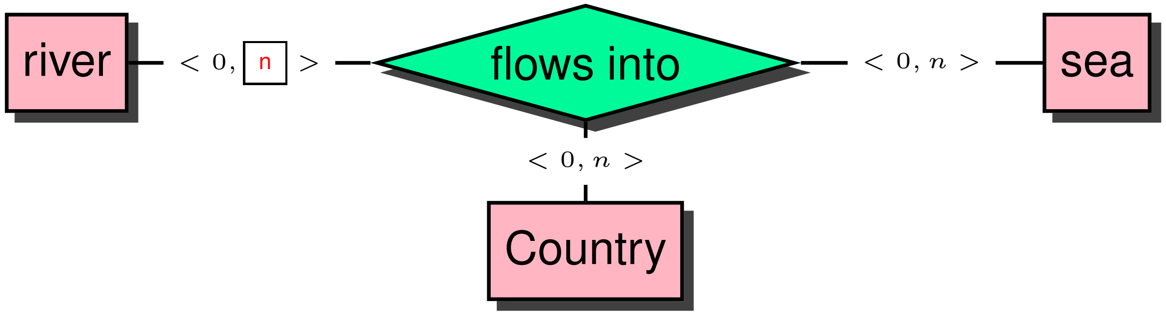


Es gibt z.B. noch ein Freiburg/CH und Freiburg/Elbe, Niedersachsen

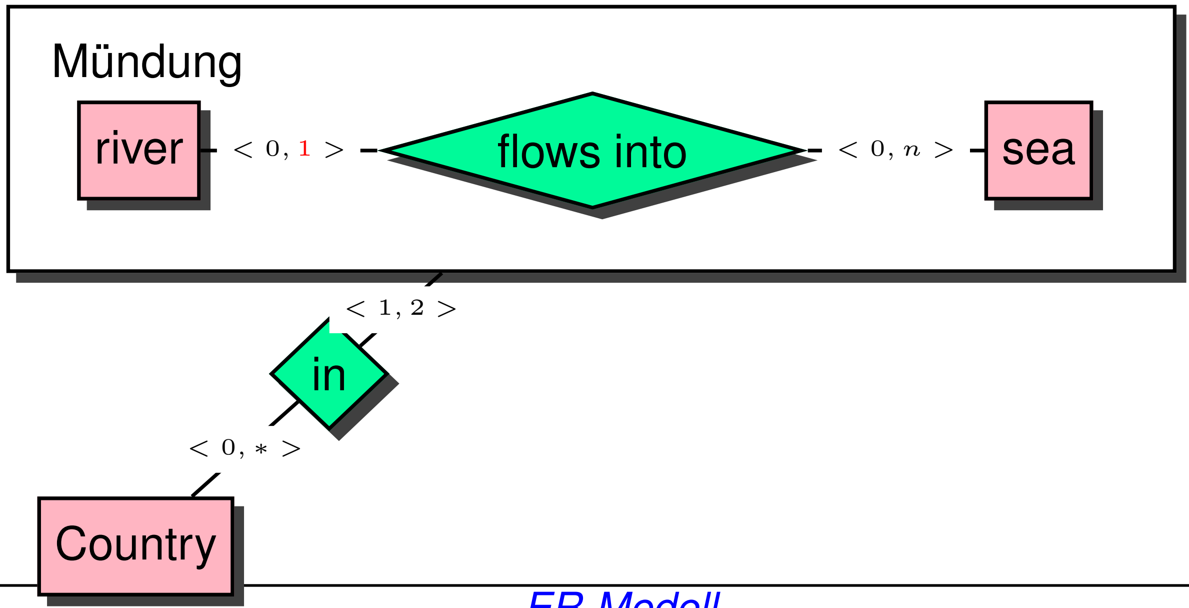
AGGREGATION

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

Die “einfache” Modellierung als dreistellige Beziehung ist wegen den Kardinalitäten nicht adäquat:

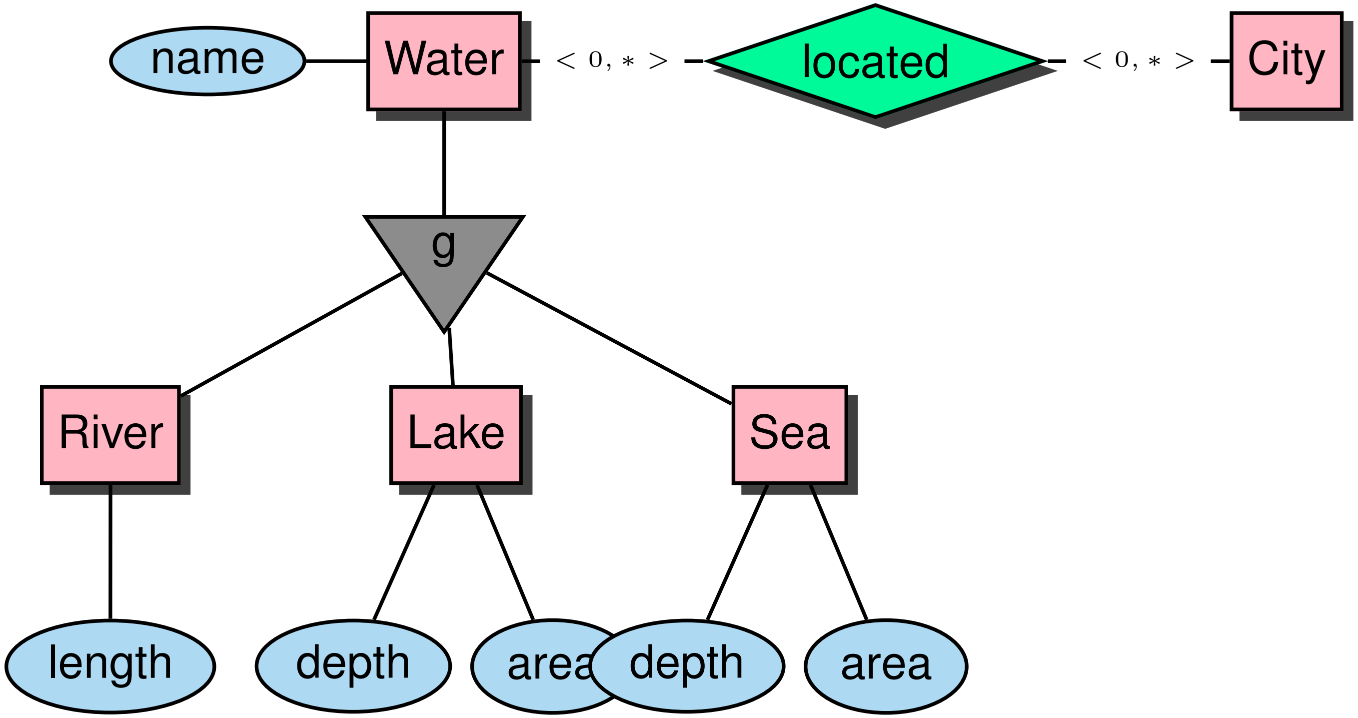


Sinnvoll, einen (reifizierten) *Aggregattyp Mündung* einzuführen:



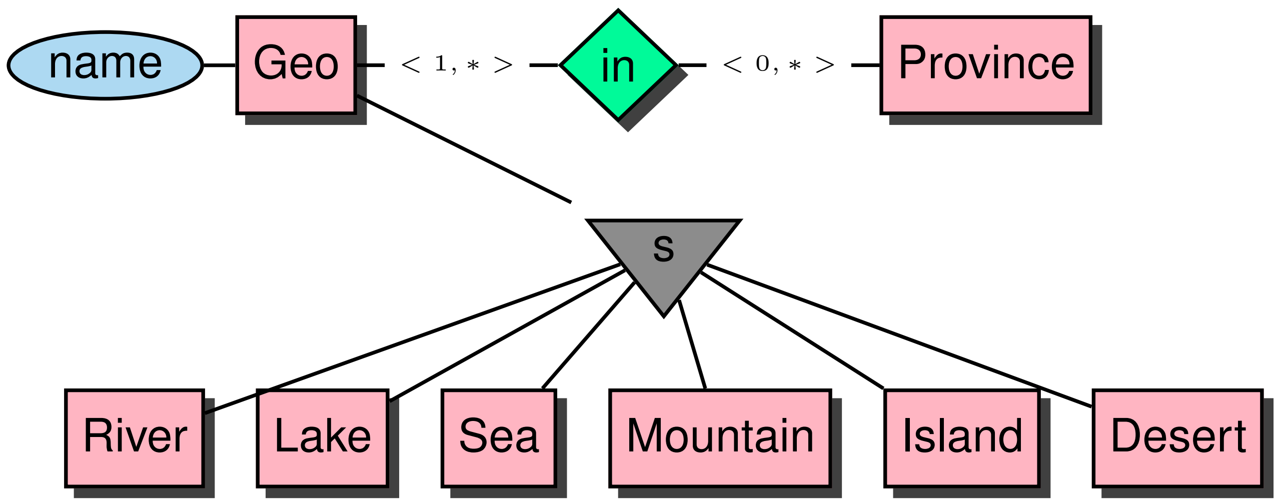
GENERALISIERUNG/SPEZIALISIERUNG

- 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.

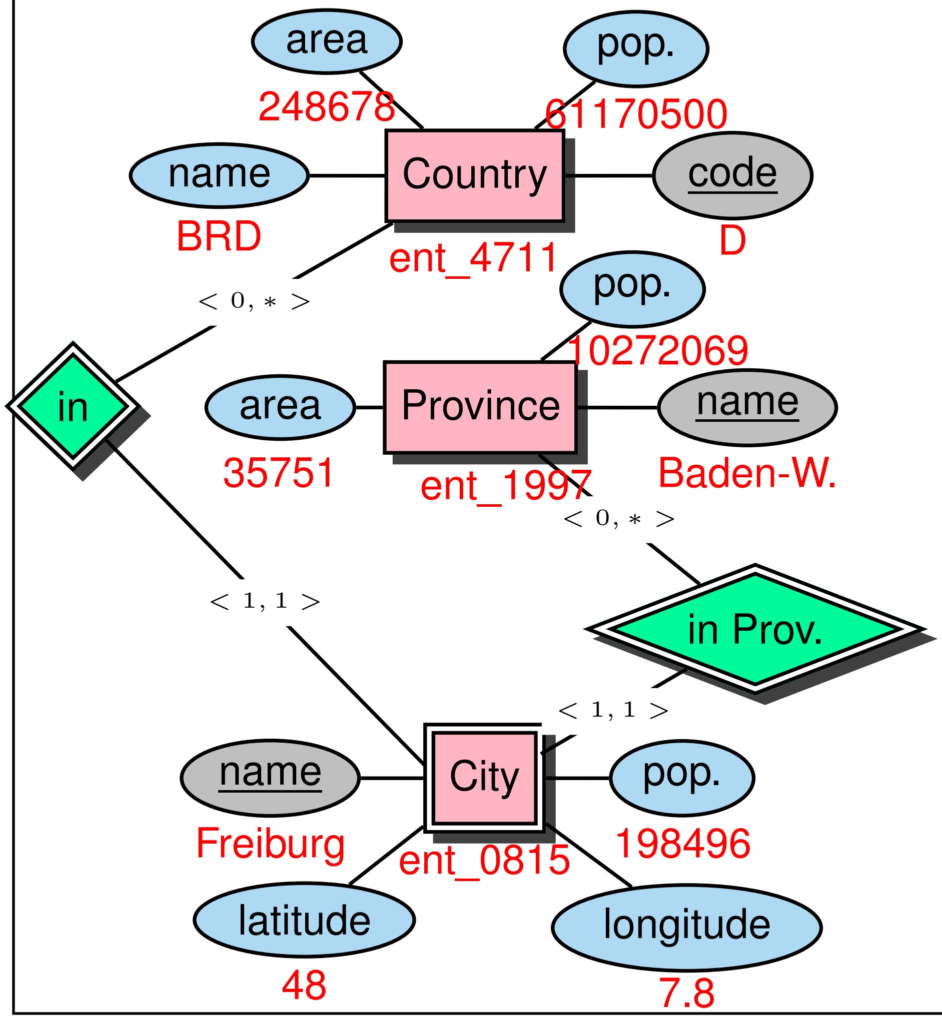
ENTITÄTSTYPEN

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

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

Schwache Entitätstypen

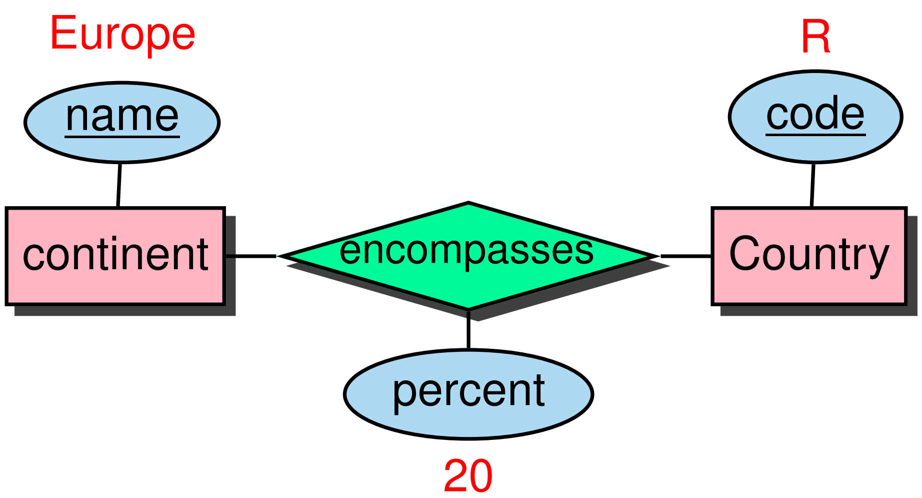
Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.



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

BEZIEHUNGSTYPEN

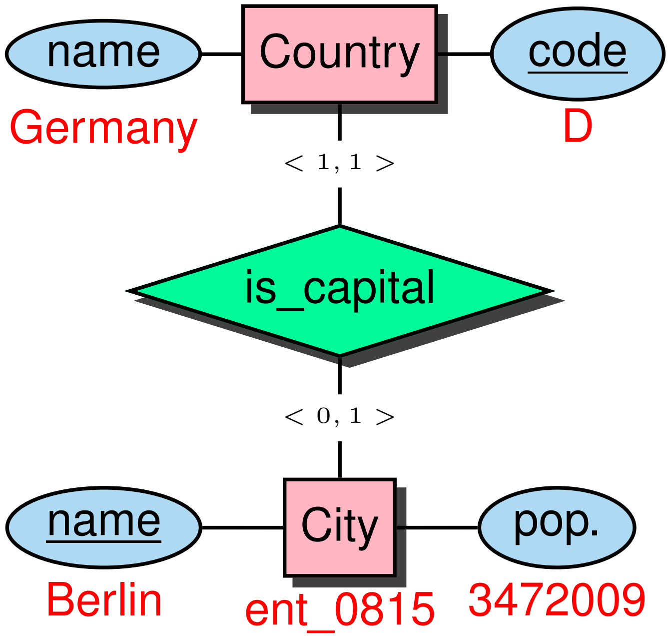
$(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
$B(E_1\_K_{11}, \ldots, E_1\_K_{1p_1}, \ldots, E_k\_K_{k1}, \ldots, E_k\_K_{kp_k},\ A_1, \ldots, A_m)$,
wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
(man darf aber umbenennen, z.B.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



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

BEZIEHUNGSTYPEN

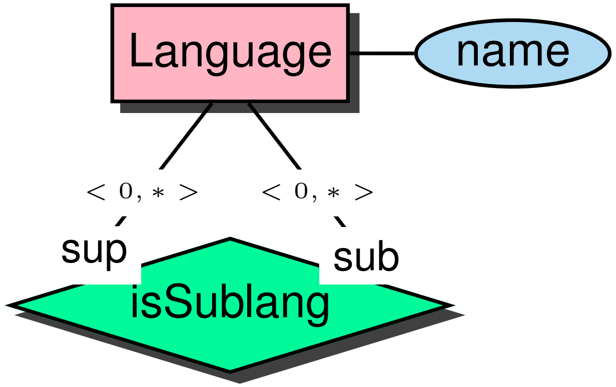
Für zweistellige Beziehungstypen können im Falle einer (0,1)- oder (1,1)-Beziehungskomplexität das Relationsschema des Beziehungstyps und das Schema des Entitätstyps zusammengefasst werden:



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	
..	..	..	..	..	

BEZIEHUNGSTYPEN

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



sublanguage	
sublang	**suplang**
English	Germanic
German	Germanic
Alsatian	German
Letzeburgisch	German
Yiddish	German
Bahamian Creole	English
Jamaican Patois	English
Pitkern	English
Norfuk	English
:	:

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
⋮	

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.

Data Dictionary: All tables

```
SELECT table_name FROM tabs ORDER BY 1;
```

Table_name
AIRPORT
BORDERS
CITY
CITYLOCALNAME
CITYPOPS
CONTINENT
COUNTRY
COUNTRYLOCALNAME
COUNTRYPOPS
DESERT
ECONOMY
ENCOMPASSES
ETHNIC_GROUP
GEO_DESERT
GEO_ESTUARY
GEO_ISLAND
GEO_LAKE
GEO_MOUNTAIN
GEO_RIVER
GEO_SEA
GEO_SOURCE
ISLAND
ISLANDIN
IS_MEMBER
LAKE
LAKEONISLAND
LANGUAGE
LOCATED
LOCATEDON
MERGES_WITH
MOUNTAIN
MOUNTAINONISLAND
ORGANIZATION
POLITICS
POPULATION
PROVINCE
PROVINCELOCALNAME
PROVPOPS
RELIGION
RIVER
RIVERONISLAND
RIVERTHROUGH
SEA
SUBLANGUAGE

44 Zeilen wurden ausgewählt.

Data Dictionary: Schema of a Table

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

```
DESC City;
```

Name	NULL?	Typ
NAME	NOT NULL	VARCHAR2(50)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(50)
POPULATION		NUMBER
LATITUDE		NUMBER
LONGITUDE		NUMBER
ELEVATION		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.	Lat.	Long.
⋮	⋮	⋮	⋮	⋮	⋮
Vienna	A	Vienna	1583000	48.2	16.37
Innsbruck	A	Tyrol	118000	47.17	11.22
Stuttgart	D	Baden-W.	588482	48.7	9.1
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 84)
- 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 84)
- 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>.

Selektion (WHERE) und Projektion (SELECT): Beispiele

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

Selektion (WHERE) und Projektion (SELECT): Beispiele

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 Anzahl 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
⋮	⋮

GESCHACHTELTE AGGREGATIONSOPERATOREN

Welche Organisation hat die meisten Mitglieder?

- erst gruppieren:

```
SELECT organization, COUNT(*)
FROM ismember
GROUP BY organization;
```

Organization	COUNT(*)
UN	193
BeNeLux	3
EU	32
UPU	226
⋮	⋮

- nur die zweite Spalte nehmen, MAX bilden:

```
SELECT MAX(COUNT(*))
FROM ismember
GROUP BY organization;
```

Oracle: 226

Postgres: ERROR: aggregate function calls cannot be nested

Geschachtelte Aggregationsoperatoren

- Oracle:

```
SELECT organization, COUNT(*)
FROM ismember
GROUP BY organization
HAVING COUNT(*) = ( SELECT max(count(*))
                    FROM ismember
                   GROUP by organization );
```

Organization	COUNT(*)
UPU	226

- Postgres: mit “> ALL” oder Subquery in der FROM-Klausel:

```
SELECT organization, COUNT(*)
FROM ismember
GROUP BY organization
HAVING COUNT(*) >=
   ALL ( SELECT count(*)
         FROM ismember
         GROUP by organization );
```

```
SELECT organization, COUNT(*)
FROM ismember
GROUP BY organization
HAVING COUNT(*) =
   ( SELECT max(num)
     FROM ( SELECT count(*) as num
            FROM ismember
            GROUP by organization ) bla);
```

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.

Join-Anfragen

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
⋮	⋮	⋮

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>[ CROSS JOIN <table_3>... ]
WHERE ...
```

- natürliches Join (über alle gemeinsamen Spaltennamen):

```
SELECT ...
FROM <table_1> NATURAL JOIN <table_2>[ NATURAL JOIN <table_3>... ]
WHERE ...
```

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

```
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.
- kann links-vor-rechts-verkettet werden:
  SELECT ...
  FROM ‹table_1›
  [INNER] JOIN ‹table_2› ON ‹conditions_12 ›
  : [INNER] JOIN ‹table_i›ON ‹conditions_12..i-1›
  WHERE ‹more conditions›

Nicht nur Syntactic Sugar: Outer Join

- 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.

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 SUBQUERIES

- 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');
```

UNKORRELIERTE SUBQUERIES

... 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 sind die Namen derjenigen 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) ;
```

Eine äquivalente Anfrage ohne Subquery müsste 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 (neben dem Mengenoperatoren UNION, INTERSECT und MINUS/EXCEPT) die einzige Form, in der Subqueries in der relationalen Algebra verwendet werden können – als Teilterme.

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(...)
5373712954

WITH: Ad-hoc Views als benannte Subqueries

- “subquery factoring”
- Subqueries separat entwickeln und schreiben
- mehrfach verwendbar

WITH <$\texttt{name}_1$> AS (<$\texttt{subquery}_1$>), ..., <$\texttt{name}_n$> AS (<$\texttt{subquery}_n$>)
<select-query>

- <$\texttt{name}_1$>, ..., <$\texttt{name}_n$> in <select-query> als Tabellennamen verwendbar.

Beispiel

```
WITH europcountries AS
     (SELECT * FROM country
      WHERE code IN
        (SELECT country FROM encompasses
         WHERE continent='Europe')),
     tokiopop AS
       (SELECT population FROM city WHERE name='Tokyo')
SELECT name
FROM europcountries
WHERE population > (SELECT population FROM tokiopop);
```

`tokiopop` ist eine einspaltige, einelementige Tabelle:
`... WHERE population > tokiopop`
ist nicht erlaubt!

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
```

Funktionales CASE WHEN ... THEN ... END-Konstrukt

- Fallunterscheidung mit *funktionaler* Semantik:
  nicht prozedurales "if ... then do something", sondern
- "result = (if $cond$ then $expr_1$ else $expr_2$)"

```
select country, name,
       case when population > 1000000 then 'big'
            when population < 100000 then 'small'
            else 'medium' end
       as size
from city
```

Name	Country	Size
Frankfurt	D	medium
Berlin	D	big
Göttingen	D	medium
Andorra	AND	small
:	:	

CASE WHEN ... for Aggregations

- very useful for statistics that should distinguish between cases

Example

```
select country,
  count(case when population >= 1000000 then 1 end) as big,
  count(case when population between 100000 and 999999
                 then 1 end) as medium,
  count(case when population < 100000 then 1 end) as small
from city
group by country
```

Country	**big**	**medium**	**small**
D	7	72	9
CH	0	6	4
:	:	:	:

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 (“root”) auswählt,
- `<recurse-condition>` spezifiziert die Join-Bedingung zwischen Eltern- und Kindtupel, `PRIOR <columnnname>`, 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 $R_1 \bowtie[R_1.\text{name} = R_2.\text{river}]$ River R_2

Beispiel: Alle Flüsse, die in den Zaire fliessen:

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

Level	Name	Length
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).

Oracle: weitere Funktionalität zu CONNECT BY

```
SELECT level, name, length
FROM river
START WITH sea is not null -- rivers flowing into seas
CONNECT BY PRIOR name = river;
```

Aber wer gehört zu wem? – Zugriff über SELECT:

- connect_by_root ‹columnname›: Operator um auf Spalten des Start-Tupels zuzugreifen,
- connect_by_isleaf: true/false wenn das erreichte Tupel ein Blatt (“Ende”) ist,
- sys_connect_by_path(‹columnname›,‹char›): Pfad als String ausgeben.

```
SELECT level, name AS Fluss1, length,
  connect_by_root name AS Fluss2,
  connect_by_isleaf AS IstQuellfluss,
  connect_by_root sea || sys_connect_by_path(name,'<-') AS Pfad
FROM river
START WITH sea IS NOT NULL
CONNECT BY PRIOR name = river;
```

Level	Fluss1	Länge	Fluss2	QF	Pfad
3	Leine	281	Weser	1	North Sea←Weser←Aller ←Leine
:	:	:	:	:	

INTERNE AUSWERTUNG UND OPTIMIERUNG

... macht das Datenbanksystem automatisch: algebraische Äquivalenzumformungen, Erstellung und Benutzung von Indexen und Statistiken (Wertverteilungen etc.).

Auswertungsplan

- Abfolge interner algebraischer Operatoren (auf einer niedrigeren Ebene als die relationale Algebra; vgl. DB-Vorlesungsabschitt zu Join-Algorithmen):
  - table full scan
  - table index lookup (`select * from country where code='D'`)
  - full join
  - hash join
  - merge join
  - index-based join
  - etc.

AUSWERTUNGSPLAN ANSCHAUEN

- SQL Developer: Anfrage angeben und auf das 3. oder 4. Icon (Autotrace, Explain Plan) klicken. Stellt das Ergebnis in Tabellenform mit Schritten und erwarteten Kosten dar.
- sqlplus: `SET AUTOTRACE ON`.
  Danach wird nach jedem Anfrageergebnis der Auswertungsplan angegeben.
- sqlplus: explain plan for select ... from ... where ...
  schreibt den Auswertungsplan in eine interne Tabelle:

```
select substr (lpad(' ', level-1) || operation ||
     ' (' || options || ')',1,30 ) as "Operation",
  object_name as "Object",
  cost, bytes, cardinality as "Rows", time
from plan_table
start with id = 0
connect by prior id=parent_id;
```

[Filename: PLSQL/explainplan.sql]

- vor dem nächsten `EXPLAIN PLAN` sollte man
  `DELETE FROM PLAN_TABLE`
  machen.