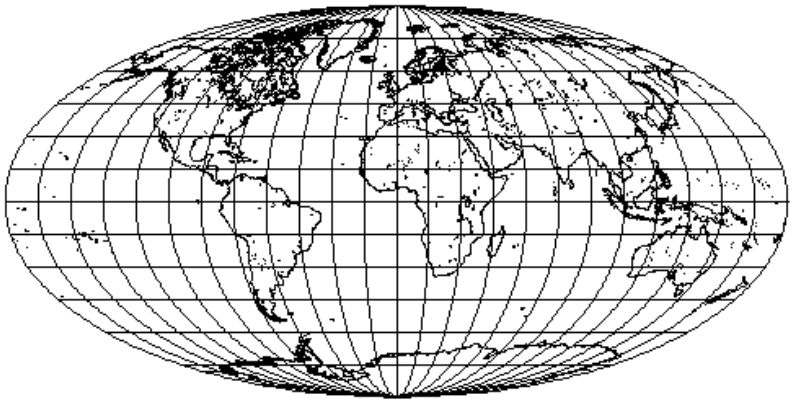
# Praktikum: Datenbankprogrammierung in SQL/ORACLE



Prof. Dr. Wolfgang May
Universität Göttingen

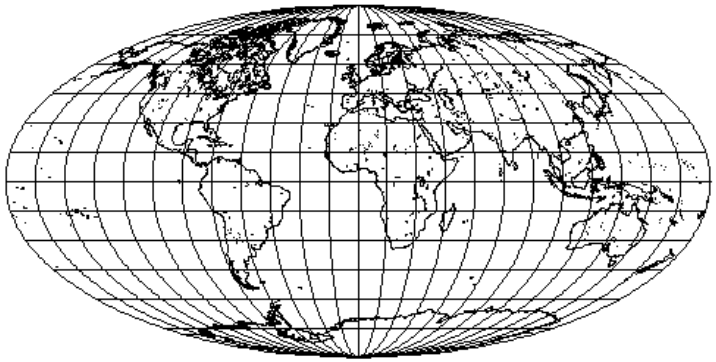
Mit Beiträgen von Erik Behrends, Rainer Himmeröder, Marco Koch, Heiko Oberdiek.



## 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.





# 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

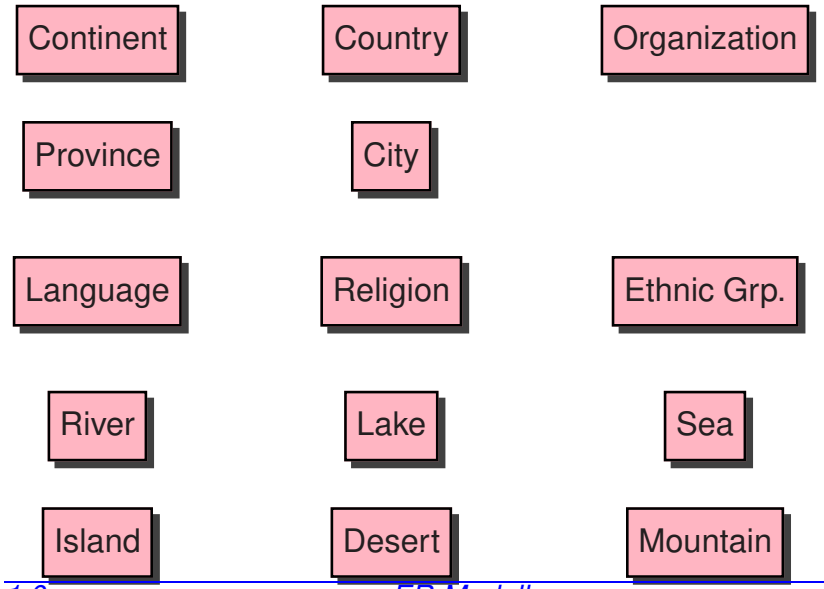
Teil III: Erweiterungen
Prozedurale Konzepte, OO, Einbettung

# Kapitel 1
# Semantische Modellierung

## ENTITY-RELATIONSHIP-MODELL (CHEN, 1976)

Strukturierungskonzepte zur Beschreibung eines Schemas im ERM:

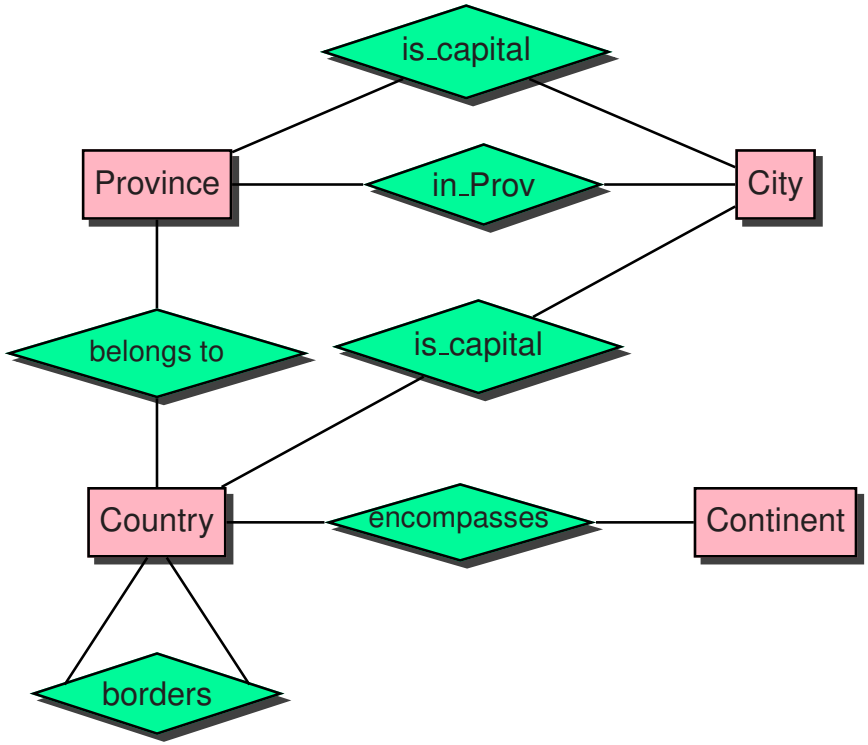
- Entitäts– (entity) Typen (≡ Objekttypen) und
- Beziehungs– (relationship) Typen








## 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.

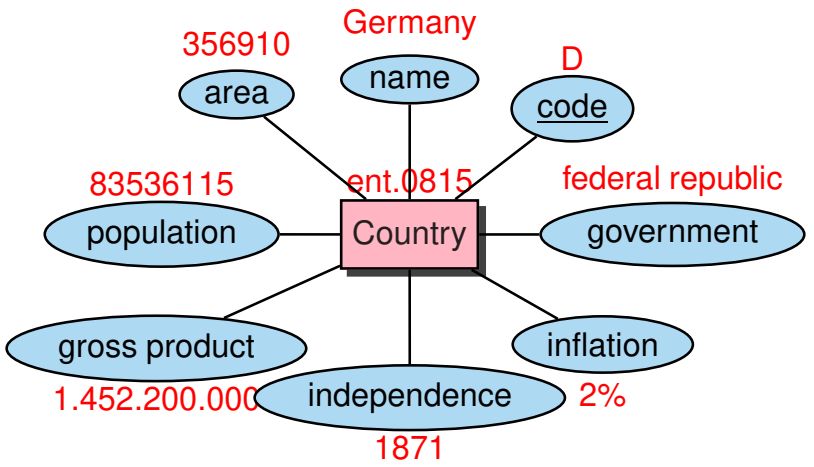
**Entität:** besitzt zu jedem Attribut ihres Entitätstyps $E$ einen Wert.

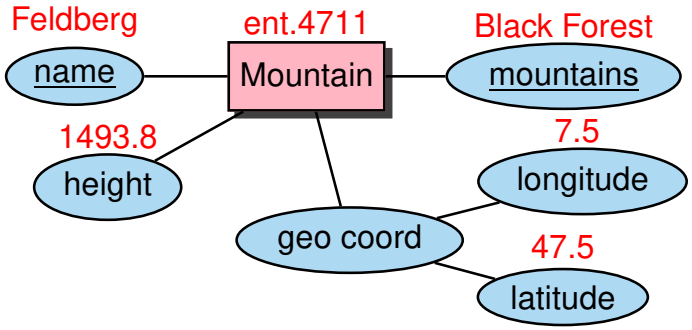
**Schlüsselattribute:** Ein Schlüssel ist eine Menge von Attributen eines Entitätstyps, deren Werte zusammen eine eindeutige Identifizierung der Entitäten eines Zustands gewährleisten soll (siehe auch *Schlüsselkandidaten, Primärschlüssel*).





## 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.

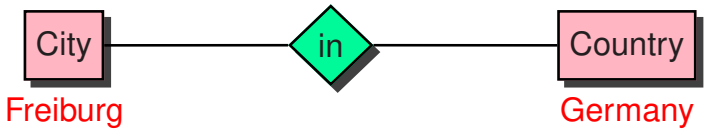
**Attribut:** Relevante Eigenschaft der Beziehungen eines Typs.

**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.

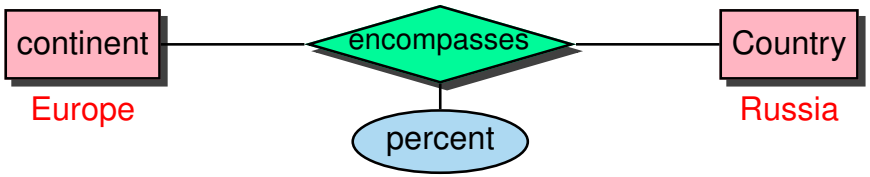




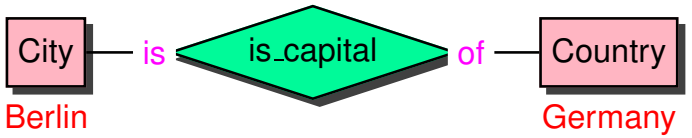
## BEZIEHUNGEN



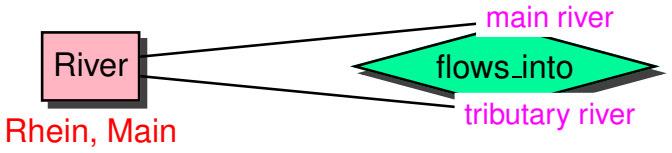

**attributierte Beziehung**




**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.

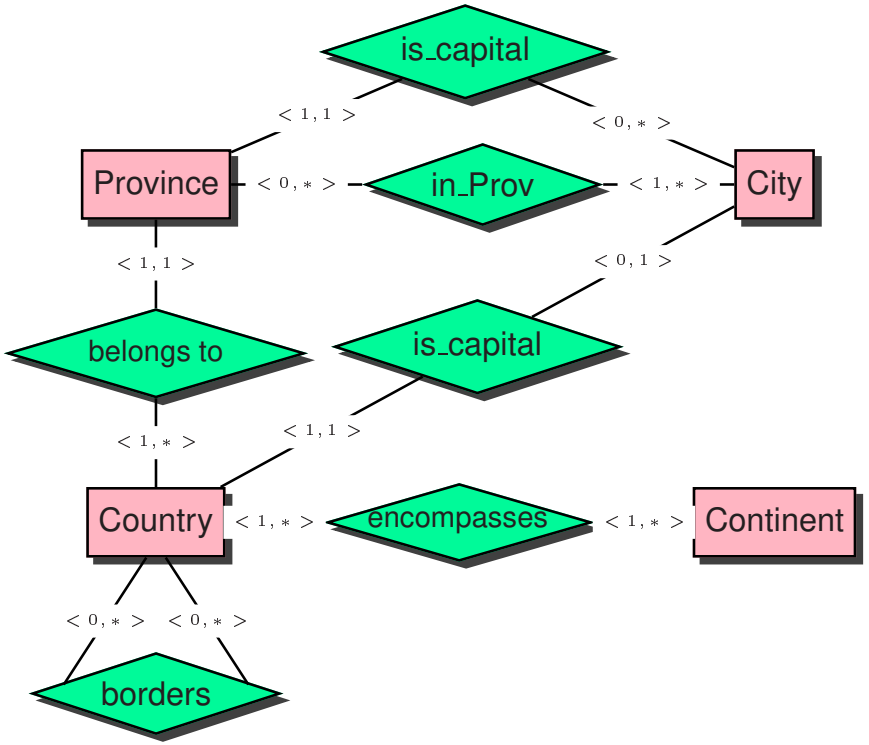
Ein **Komplexitätsgrad** eines Beziehungstyps $B$ bzgl. einer seiner Rollen $RO$ ist ein Ausdruck der Form $(min, max)$.

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

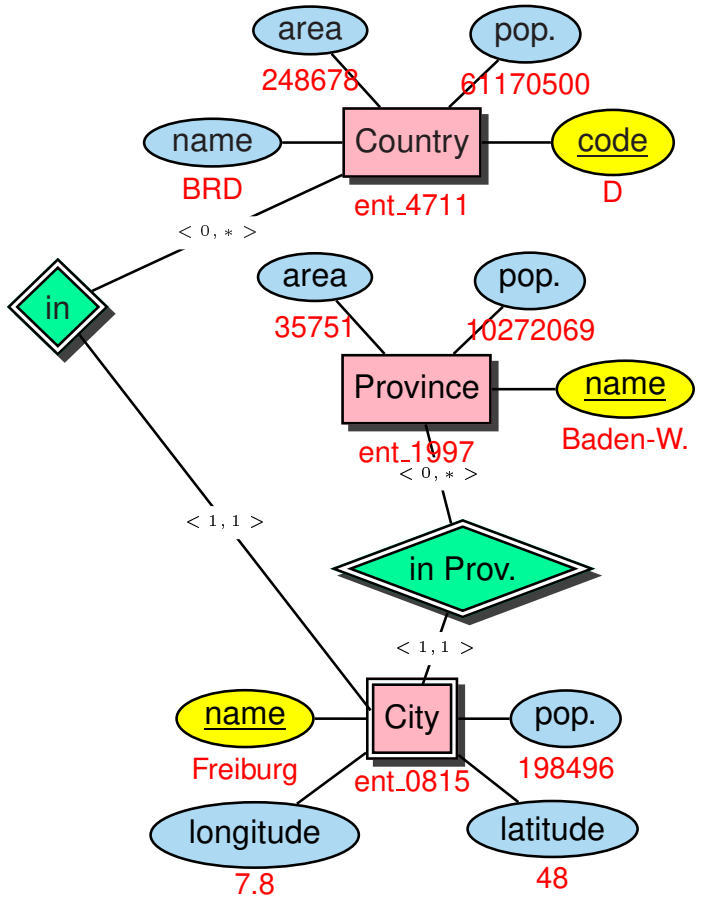
Ein schwacher Entitätstyp ist ein Entitätstyp ohne Schlüssel.

- 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**).





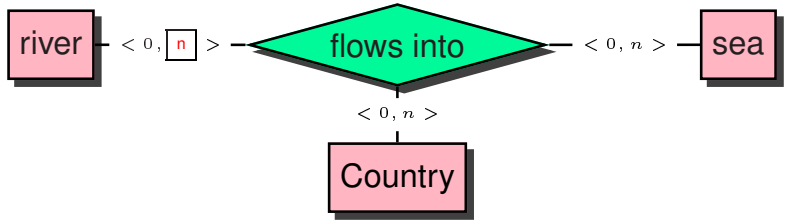
## SCHWACHE ENTITÄTSTYPEN




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

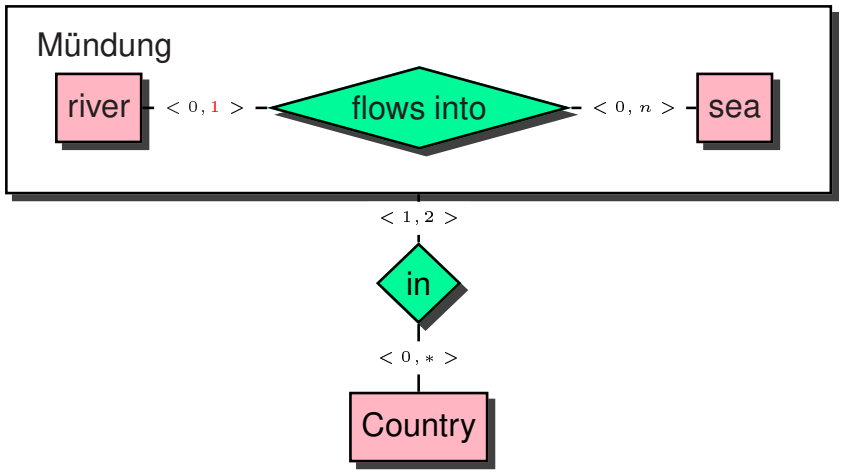
## MEHRSTELLIGE BEZIEHUNGEN

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:

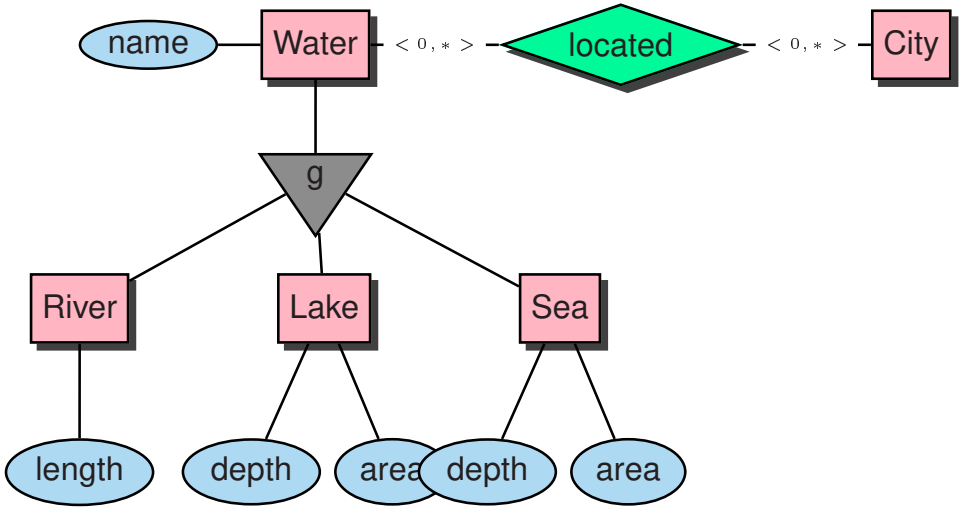






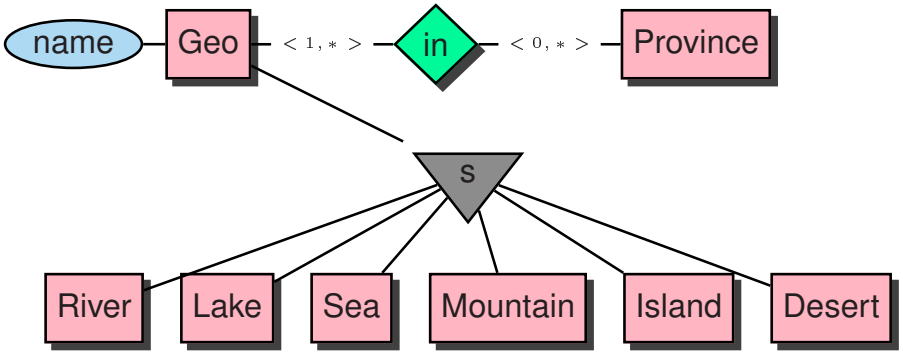

## GENERALISIERUNG/SPEZIALISIERUNG

- 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: VARCHAR(25), Area: NUMBER
- Die Elemente einer Relation werden als *Tupel* bezeichnet.
  (Asia,4.5E7)

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

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





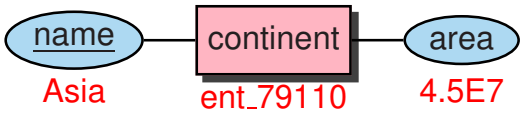
## ABBILDUNG ERM IN RM

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

1. Entitätstypen: $(E_{ER}, \{A_1, \ldots, A_n\}) \longrightarrow E(A_1, \ldots, A_n)$,
2. Beziehungstypen:
   $(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
   $B(E_1_K_{11}, \ldots, E_1_K_{1p_1}, \ldots,$
   $E_k_K_{k1}, \ldots, E_k_K_{kp_k}, A_1, \ldots, A_m)$ ,
   wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
   Falls $B_{ER}$ Rollenbezeichnungen enthält, so wird durch die Hinzunahme der Rollenbezeichnung die Eindeutigkeit der Schlüsselattribute im jeweiligen Beziehungstyp erreicht.
   Für $k = 2$ können im Falle einer (1,1)-Beziehungskomplexität das Relationsschema des Beziehungstyps und das Schema des Entitätstyps zusammengefasst werden.
3. Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.
4. Aggregattypen können unberücksichtigt bleiben, sofern der betreffende Beziehungstyp berücksichtigt wurde.

# ENTITÄTSTYPEN

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



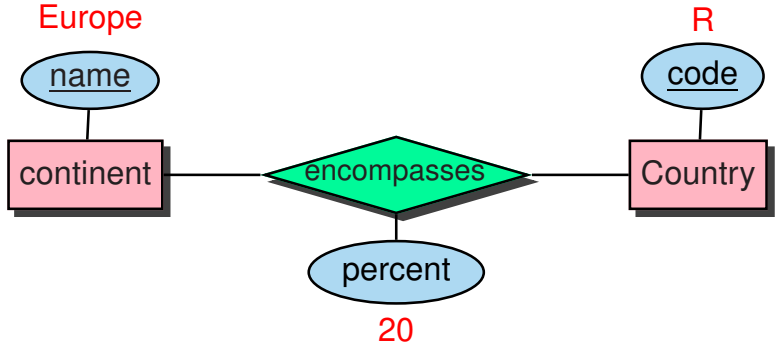

| Continent | |
|---|---|
| **Name** | **Area** |
| VARCHAR(20) | NUMBER |
| Europe | 9562489.6 |
| Africa | 3.02547e+07 |
| Asia | 4.50953e+07 |
| America | 3.9872e+07 |
| Australia | 8503474.56 |





# BEZIEHUNGSTYPEN

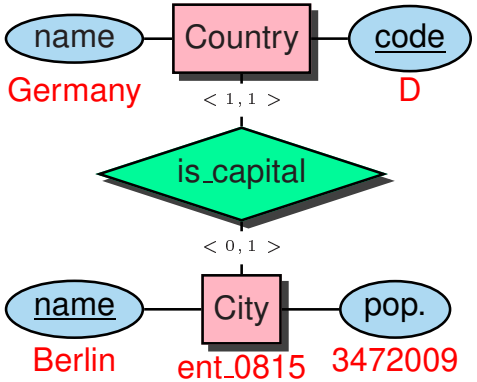
$(B_{ER}, \{RO_1 : E_1, \ldots, RO_k : E_k\}, \{A_1, \ldots, A_m\}) \longrightarrow$
$B(E_1_K_{11}, \ldots, E_1_K_{1p_1}, \ldots,$
$E_k_K_{k1}, \ldots, E_k_K_{kp_k}, A_1, \ldots, A_m),$
wobei $\{K_{i1}, \ldots, K_{ip_i}\}$ Primärschlüssel von $E_i, 1 \leq i \leq k$.
(man darf aber umbenennen, z.B. *Country*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




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

# BEZIEHUNGSTYPEN

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



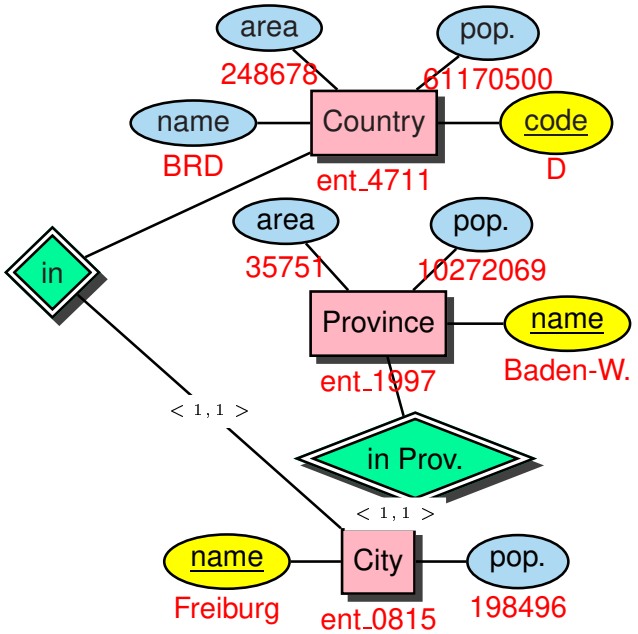

| Country | | | | | |
|---|---|---|---|---|---|
| **Name** | **code** | **Population** | **Capital** | **Province** | **...** |
| Germany | D | 83536115 | Berlin | Berlin | |
| Sweden | S | 8900954 | Stockholm | Stockholm | |
| Canada | CDN | 28820671 | Ottawa | Quebec | |
| Poland | PL | 38642565 |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





# SCHWACHE ENTITÄTSTYPEN

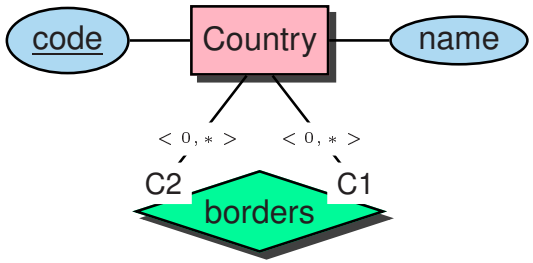
Für einen schwachen Entitätstyp müssen die Schlüsselattribute des identifizierenden Entitätstyps hinzugenommen werden.




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

## BEZIEHUNGSTYPEN

Falls $B_{ER}$ Rollenbezeichnungen enthält, so 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-2 (1992), SQL-3 (1996)
- SQL-2 in 3 Stufen, entry, intermediate und full level eingeführt.
- SQL-3: Objektorientierung
- deskriptive Anfragesprache
- Ergebnisse immer Mengen von Tupeln (Relationen)
- Implementierungen: ORACLE (im Praktikum), IBM DB2, MS SQL Server 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<br>Arguments in objects accessible to the user |
| ALL_CATALOG<br>All tables, views, synonyms, sequences accessible to the user |
| ALL_CLUSTERS<br>Description of clusters accessible to the user |
| ALL_CLUSTER_HASH_EXPRESSIONS<br>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 | LAKE |
| CONTINENT | LANGUAGE |
| COUNTRY | LOCATED |
| DESERT | IS_MEMBER |
| ECONOMY | MERGES_WITH |
| ENCOMPASSES | MOUNTAIN |
| ETHNIC_GROUP | ORGANIZATION |
| GEO_DESERT | POLITICS |
| GEO_ISLAND | POPULATION |
| GEO_LAKE | PROVINCE |
| GEO_MOUNTAIN | RELIGION |
| GEO_RIVER | RIVER |
| GEO_SEA | SEA |

28 Zeilen wurden ausgewählt.

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

```
DESC City;
```

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





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





## 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 | NULL | ... |
| Mull | Inner Hebrides | 910 | ... |
| Montserrat | Antilles | 106 | ... |
| Grenada | Antilles | NULL | ... |
| ⋮ | ⋮ | ⋮ | ⋮ |

```
SELECT Islands
FROM Island;
```

| Islands |
|---|
| ⋮ |
| Channel Islands |
| Inner Hebrides |
| Antilles |
| Antilles |
| ⋮ |

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

| Islands |
|---|
| ⋮ |
| Channel Islands |
| Inner Hebrides |
| Antilles |
| ⋮ |





# DUPLIKATELIMINIERUNG

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

## 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(*) |
|---|
| 3114 |

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

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

| Count(DISTINCT(Country)) |
|---|
| 68 |

## 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 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 |
| 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 and Montenegro |
| 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 Migliedern Mitglied ist.

```
SELECT c.name, org.organization
FROM
  (SELECT Name, Code
   FROM Country
   WHERE Population > 50000000) c,
  is_member,
  (SELECT organization
   FROM is_member
   GROUP BY organization
   HAVING count(*) > 20) org
WHERE c.code = is_member.country
  AND is_member.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) |
|---|
| 273837106 |

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

| absolut | relativ |
|---|---|
| 4620065771 | 0.196734334 |





## 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(...) |
|---|
| 4620065771 |

## BEISPIELANFRAGE

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

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

| Name |
|---|
| Austria |
| Denmark |
| Germany |
| Ireland |
| Italy |
| Netherlands |
| Spain |
| United Kingdom |