

# 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: Data Definition Language** zur Definition der Schemata

- Tabellen
- Sichten
- Indexe
- Integritätsbedingungen

**DML: Data Manipulation Language** 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.

<b>TABLE_NAME</b>
<b>COMMENTS</b>
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(40)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(40)
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=cltY. (Ausnahmen siehe Folie 78)
- Innerhalb von Quotes ist SQL nicht case-insensitive, d.h. City='Berlin' ≠ City='berlin'.
- String-Konstanten in der WHERE-Klausel werden in einfache Anführungszeichen eingeschlossen, nicht in doppelte. (doppelte Anführungszeichen machen etwas anderes, siehe Folie 78)
- Jeder Befehl wird mit einem Strichpunkt ";" abgeschlossen.
- Kommentarzeilen werden in /\* ... \*/ 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;
```

<u>Name</u>	<u>COUNTRY</u>
Tokyo	J
Stockholm	S
Warsaw	PL
Cochabamba	BOL
Hamburg	D
Berlin	D
..	..

**DISTINCT**

```
SELECT * FROM Island;
```

<u>Name</u>	<u>Islands</u>	<u>Area</u>	...
:	:	:	:
Jersey	Channel Islands	117	...
Mull	Inner Hebrides	910	...
Montserrat	Lesser Antilles	102	...
Grenada	Lesser Antilles	344	...
:	:	:	:

```
SELECT Islands
FROM Island;
```

<u>Islands</u>
:
Channel Islands
Inner Hebrides
Lesser Antilles
Lesser Antilles
:

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

<u>Islands</u>
:
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  $op \in \{=, <, >, \leq, \geq\}$ ,
- <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<sub>1</sub>', ..., 'val<sub>n</sub>') 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**

**BEDINGUNGEN AN GRUPPIERUNGEN**

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

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

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

- 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.

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*.

**EQUIJOIN**

**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
⋮	⋮

**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> ∈ {=, <, >, <=, >=} 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**

(mehrsplätige (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

- insbesondere 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);
```

<b>sum(pop_biggest)</b>
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
```

**WITH: AD-HOC VIEWS ALS BENANNTE SUBQUERIES**

- "subquery factoring"
- Subqueries separat entwickeln und schreiben
- mehrfach verwendbar

```
WITH <name1> AS (<subquery1>),
     ...
     <namen> AS (<subqueryn>)
<select-query>
```

- <name<sub>1</sub>>, ..., <name<sub>n</sub>> in <select-query> 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!

## 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-Gleichheitsbedingung):
 

```
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];
```

**SYNTACTIC SUGAR: JOIN**

**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
```

mit `<joined-tables-spec>`:

- bisher: `SELECT ... FROM ... WHERE <(join-)conditions>`
- abkürzend:
 

```
SELECT ... FROM <joined-tables-spec>
WHERE <conditions>
```

- 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:  
 $R \bowtie [Bedingung]R \dots \bowtie [Bedingung]R \bowtie [Bedingung]R$
- 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<sub>1</sub> ⋈ [R<sub>1</sub>.name = R<sub>2</sub>.river] River R<sub>2</sub>

- 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).

**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-Vorlesungsabschnitt 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.