Datenbankpraktikum SQL

Prof. Dr. Wolfgang May
may@informatik.uni-goettingen.de
Institut für Informatik
Universität Göttingen

Göttingen, März 2005

Vorwort

Im Rahmen des SQL-Praktikums sollen die wichtigsten Elemente des SQL-Standards an einer bestehenden relationalen ORACLE-Datenbank angewendet werden. Grundlagen für das Datenbankpraktikum sind

- Kenntnis der ER-Modellierung,
- Kenntnis des relationalen Datenbankmodells,

wie sie parallel in der Vorlesung "Datenbanken" erworben werden können.

Das relationale Datenbankmodell wurde 1970 von Codd [Cod70] vorgestellt. Bis heute bildet es die Grundlage für die meisten kommerziellen Datenbanksysteme. Die ER-Modellierung ist ein weit verbreitetes Hilfsmittel zum konzeptuellen Entwurf (relationaler) Datenbanken. Ihrer Bedeutung entsprechend sind diese Themen in vielen Lehrbüchern (etwa [Dat90], [KS91], [Vos94] und [UW97]) ausführlich dargestellt. In der Vorlesung "Datenbanken" werden diese Themen ebenfalls umfassend behandelt.

Eine Datenbank besteht (für den Benutzer) aus einem *Datenbankschema* und einer *Datenbasis* (Datenbankinhalt). Das Datenbankschema wird mit Hilfe der *Data Definition Language (DDL)* definiert und verändert. Informationen über das Schema werden im *Data Dictionary*, einem Teil des *Datenbank-Management-System (DBMS)*, gehalten. Die *Datenbasis* (Datenbankinhalt) wird in den Relationen der Datenbank gespeichert und über die *Data Manipulation Language (DML)* verändert.

Die Kurzform **SQL** steht für **S**tructured **Q**uery **L**anguage, eine international standardisierte Sprache zur Definition und Manipulation relationaler (und inzwischen auch objektrelationaler) Datenbanksysteme. Die bekanntesten Datenbanksysteme, die SQL implementieren sind ORACLE, Informix, INGRES, SYBASE, MySQL, MS SQL Server, etc ...

1986 wurde der erste Standard unter dem Namen SQL1 bzw. SQL86 verabschiedet. Der momentan gültige Standard ist SQL2 [ISO92] (auch als SQL92 bezeichnet). Dieser Standard soll in drei Schritten (entry, intermediate und full level) in existierende Produkte eingeführt werden. Während SQL2 noch nicht vollständig umgesetzt ist, ist bereits die nächste Fassung des SQL-Standards - SQL3 - vorgeschlagen [ISO94]. Wesentliche Neuerung gegenüber den bisherigen Standards sollen Erweiterungen in Richtung Objektorientierung und Rekursion sein. Der Bedeutung der Sprache gemäß steht auch hier ein breites Spektrum an Literatur zur Verfügung, u. a. [MU97] und [DD94]. ORACLE 8 ist aktuell u. a. in [HV98] beschrieben.

In dem Praktikum wird jeweils die aktuelle Version von ORACLE verwendet. Die Version ORACLE 7 war ein rein relationales Datenbanksystem, etwa dem SQL-2 Standard entsprechend. Die Spracherweiterung PL/SQL bietet zusätzliche prozedurale Konstrukte. In ORACLE 8 (seit 1997) stehen zusätzlich geschachtelte Tabellen und *objektrelationale* Features zur Verfügung. Seit ORACLE 9 (2001) wurden umfangreiche Funktionalitätspakete (z. B. IAS; *Internet Application Server*) integriert, die weit über die eigentliche Datenbankfunktionalität hinausgehen. Sie werden in diesem grundlegenden Praktikum nicht behandelt.

In diesem Skript wird beschrieben, wie man

- ein Datenbankschema erstellt,
- Daten aus einer Datenbank ausliest,
- den Inhalt einer Datenbasis ändert,
- unterschiedliche Sichten auf dieselben Daten definiert,
- Zugriffsrechte auf eine Datenbank erteilt,
- objektrelationale Konzepte integriert, sowie
- SQL-Anfragen in eine höhere Programmiersprache einbettet.

Im Praktikum ist die Reihenfolge, in der die Befehle eingeführt werden, verändert: Da von einer gegebenen Datenbank ausgegangen wird, werden zuerst Anfragebefehle behandelt, um möglichst schnell praktisch arbeiten zu können. Danach wird die Erstellung des Datenbankschemas sowie die Erteilung von Zugriffsrechten behandelt. Aufbauend auf diesen grundlegenden Kenntnissen werden weitere Aspekte der Anfragebearbeitung, Modifikationen an Datenbankinhalt und -schema, sowie algorithmische Konzepte rund um SQL behandelt. Die mit ORACLE 8 neu hinzugekommenen Konzepte zur Objektorientierung werden an geeigneten Stellen separat vorgestellt. Weiterhin wird die Einbindung in C++ und Java beschrieben.

Das Praktikum wird mit der geographischen Datenbasis MONDIAL durchgeführt (siehe Anhang A). Die Idee des Praktikums sowie die Datenbasis MONDIAL basiert auf der Datenbasis TERRA[1] und dem SQL-Versuch des Datenbankpraktikums[2] am *Institut für Programmstrukturen und Datenorganisation* der *Universität Karlsruhe*. Beides ist in [DR90] umfassend beschrieben.

Am *Institut für Informatik* der *Universität Freiburg* wurde das Praktikum erstmals im Wintersemester 97/98 unter ORACLE 7 durchgeführt. Im Zuge der Erweiterung auf ORACLE 8 (Objektrelationale Erweiterungen) zum Wintersemester 98/99 wurde auch die Datenbasis mit Hilfe von im WWW bereitgestellten Daten erneuert (Stand 1997) und erweitert. Seitdem wird das SQL-Praktikum regelmäßig an der *Universität Freiburg* durchgeführt. Seit dem Wintersemester 2001/02 ist es in reduzierter Form fester Bestandteil der Vorlesung *Introduction to Databases and SQL* im Master-Studiengang in *Applied Computer Sciences*. Im Winter 2002/03 wurde die Beschreibung neuer Funktionalität von ORACLE 9 (Vererbungshierarchie) integriert. Ebenfalls im Winter 2002/03 wurde das Material erstmals in der Vorlesung *Einführung in Datenbanken und SQL* im Bachelor-Studiengang *Angewandte Informatik* an der *Universität Göttingen* verwendet.

[1] TERRA unterliegt dem Copyright des Instituts für Programmstrukturen und Datenorganisation der Universität Karlsruhe.

[2] in welchem der Autor im Sommersemester 1992 seine ersten Schritte mit ORACLE gemacht hat.

Inhaltsverzeichnis

Teil I

Grundlegende Konzepte

1 DATENBANKANFRAGEN IN SQL

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

`SELECT`	Attribut(e)
`FROM`	Relation(en)
`WHERE`	Bedingung(en)

Das Ergebnis einer Anfrage ist immer eine Menge von Tupeln. Beim Programmieren in SQL sind die folgenden Eigenheiten zu berücksichtigen:

- 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 /∗ . . . ∗/ eingeschlossen (auch über mehrere Zeilen), oder mit `--` oder `rem` eingeleitet (kommentiert den Rest der Zeile aus).

1.1 Auswahl von Spalten einer Relation (Projektion)

Mit Hilfe von Projektionen werden die Spalten einer Relation bestimmt, die ausgegeben werden sollen:

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

An die bereits beschriebene Relation `City` kann man etwa folgende Anfrage stellen:

Beispiel:

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

Name	Country	Population
Tokyo	J	7843000
Stockholm	S	711119
Cordoba	E	315948
Cordoba	MEX	130695
Cordoba	RA	1207813
..	..	..

Wird als `<attr-list>` ein ∗ angegeben, so werden sämtliche Attribute der Relation ausgegeben. Die einfachste Form einer Anfrage ist also z. B.

```
SELECT * FROM City;
```

Name	Country	Province	Population	Longitude	Latitude
⋮	⋮	⋮	⋮	⋮	⋮
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.

Bei dieser Anfrage stellt man fest, dass für viele Städte unter *Province* der Name des Landes eingetragen ist – dies ist immer dann der Fall, wenn in den Datenquellen keine spezielleren Informationen gefunden werden konnten.

Durch das Schlüsselwort `DISTINCT` wird sichergestellt, dass das Ergebnis keine Duplikate enthält (im Standard ist `DISTINCT <attr-list>` erlaubt, was jedoch in ORACLE nicht zugelassen ist.) :

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

Will man z. B. feststellen, welche Inselgruppen in der Datenbasis enthalten sind, kann man dies folgendermaßen erreichen:

Beispiel:

```
SELECT Islands
FROM Island;
```

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
Antilles
⋮

Beispiel:

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

Islands
⋮
Channel Islands
Inner Hebrides
Antilles
⋮

Die Duplikateliminierung erfolgt nur bei den Mengenoperatoren `UNION`, `INTERSECT`, ... (vgl. Abschnitt 1.5) automatisch. Ansonsten muss sie explizit durch die `DISTINCT`-Klausel gefordert werden. Einige Gründe für dieses Vorgehen sind folgende:

- Duplikateliminierung ist "teuer " (Sortieren $(O(n \cdot \log n))$ + Eliminieren)
- Nutzer will die Duplikate sehen
- Anwendung von Aggregatfunktionen (vgl. Abschnitt 1.4) auf Relationen mit Duplikaten

1.2 Auswahl von Zeilen einer Relation (Selektion)

Mit Hilfe von Selektionen werden Tupel ausgewählt, die bestimmte Bedingungen erfüllen.

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

```
WHERE <predicate>;
```

<predicate> kann dabei die folgenden Formen annehmen:

- <attribute> <rel> <expr> mit rel $\in \{=, <, >, <=, >=\}$, oder <expr> BETWEEN <expr> AND <expr>, wobei <expr> ein mathematischer Ausdruck über Attributen und Konstanten ist,
- <attribute> <IS NULL,
- <attribute> [NOT] LIKE <string>, wobei underscores im String ein Zeichen repräsentieren und Prozentzeichen null bis beliebig viele Zeichen darstellen,
- <attribute-list> IN <value-list>, wobei <value-list> entweder von der Form (<val1>, ..., <valn>) ist, oder durch eine Subquery (vgl. Abschnitt 1.6.2) 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	Einwohner
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, Einwohner
FROM City
WHERE Country LIKE '%J_%';
```

Name	Country	Population
Kingston	JA	101000
Beijing	TJ	7000000
Amman	JOR	777500
Suva	FJI	69481
⋮	⋮	⋮

Durch die Forderung, dass nach dem J noch ein weiteres Zeichen folgen muss, führt dazu, dass die japanischen Städte nicht aufgeführt werden.

Nullwerte können in allen Spalten auftreten, in denen sie nicht durch NOT NULL oder PRIMARY KEY (vgl. Abschnitt 2) verboten sind. Einträge können mit IS [NOT] NULL auf Nullwerte getestet werden:

Beispiel:

```
SELECT Name, Country
FROM City
WHERE Population IS NULL;
```

Es werden 469 Städte ausgegeben, deren Einwohnerzahl nicht bekannt ist.

1.3 Sortieren von Zeilen

Soll die Ausgabe einer SELECT-Anfrage nach bestimmten Kriterien geordnet sein, so kann dies durch die ORDER BY-Klausel erreicht werden:

```
SELECT <attr-list>
FROM <table>
WHERE <predicate>
ORDER BY <attribute-list> [DESC];
```

Wird die Option [DESC] verwendet, wird in absteigender Reihenfolge nach den in <attr-list> angegebenen Attributen sortiert, sonst in aufsteigender Reihenfolge.
Statt der Attributnamen kann die <attribute-list> auch die Position des betreffenden Attributs in der <attribute-list> enthalten. Dies ist besonders dann sinnvoll, wenn nach einem in der Anfrage berechneten Attribut sortiert werden soll, das keinen Namen hat (s. Beispiel). Für solche *virtuellen Spalten* ist es sinnvoll, einen *Alias* zu definieren, der in der Ausgabe als Spaltenname verwendet wird.

Beispiel: Gesucht seien alle Millionenstädte sortiert nach ihrer Einwohnerzahl, wobei die Ausgabe in Mio. Einwohner erfolgen soll.

```
SELECT Name, Country,
  Population/1000000 AS Mio_Inh
FROM City
WHERE Pop > 1000000
ORDER BY 3 DESC;
```

Name	**Country**	**Mio_Inh**
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
⋮	⋮	⋮

1.4 Aggregatfunktionen und Gruppierung

SQL bietet zusätzlich die Möglichkeit, statistische Informationen über eine Menge von Tupeln abzufragen. Dazu stehen die folgenden *Aggregatfunktionen* zur Verfügung:

- COUNT (*) oder COUNT ([DISTINCT]<attribute>) zum Zählen der Häufigkeit des Auftretens,
- MAX (<attribute>) zur Bestimmung des Maximums,

- `MIN (<attribute>)` analog zur Bestimmung des Minimums,
- `SUM ([DISTINCT] <attribute>)` zum Aufsummieren aller Werte und
- `AVG ([DISTINCT] <attribute>)` zur Durchschnittsbildung.

Im einfachsten Fall werden diese Aggregatfunktionen auf eine ganze Relation angewandt.

Beispiel: Gesucht sei die Zahl der 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

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

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

SUM(Population)	MAX(Population)
2434525	1583000

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

Gruppierung. Komplexere Anfragen können formuliert werden, wenn die Gruppierungsfunktion `GROUP BY` verwendet wird. `GROUP BY` berechnet *eine Zeile*, die Daten enthalten kann, die mit Hilfe der Aggregatfunktionen über mehrere Zeilen berechnet werden.

Eine Anweisung

```
SELECT <expr-list>
FROM <table-list>
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,
- *Aggregatfunktionen*, die dann über alle Tupel in der entsprechenden Gruppe gebildet werden,

enthalten (nicht alle Attribute aus `<attr-list>` müssen auch `SELECT`ed werden!).
Die `WHERE`-Klausel `<predicate>` enthält dabei nur Attribute der Relationen in `<table-list>` (also *keine* Aggregatfunktionen).

Der `GROUP BY`-Operator unterteilt die in der `FROM`-Klausel angegebene Tabelle in Gruppen, so dass alle Tupel innerhalb einer Gruppe den gleichen Wert für die `GROUP BY`-Attribute haben. Dabei werden alle Tupel, die die `WHERE`-Klausel nicht erfüllen, eliminiert. Danach werden *innerhalb* jeder solchen Gruppe die Aggregatfunktionen berechnet.

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
⋮	⋮

Wie bereits erwähnt, darf die `WHERE`-Klausel `<predicate>` dabei nur Attribute von `<table-list>` erhalten, d.h. dort ist es nicht möglich, Bedingungen an die Gruppenfunktionen auszudrücken.

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

```
SELECT <expr-list>
FROM <table-list>
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.

Beispiel: Gesucht ist für jedes Land die Gesamtzahl der Einwohner, die in 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
⋮	⋮

1.5 Mengenoperationen

Mehrere SQL-Anfragen können über Mengenoperatoren verbunden werden:

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

Die folgenden Operatoren stehen zur Verfügung.

- `UNION [ALL]`

- MINUS [ALL]
- INTERSECT [ALL]

Als Default werden dabei Duplikate automatisch eliminiert. Dies kann durch ALL verhindert werden.

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
⋮

… allerdings ist "Armenia" nicht die Hauptstadt von Armenien, sondern liegt in Kolumbien.

1.6 Anfragen, die mehrere Relationen umfassen

In vielen Fällen interessieren Informationen, die über zwei oder mehr Relationen verteilt sind.

1.6.1 Join-Anfragen

Eine Möglichkeit, solche Anfragen zu stellen, sind *Join-Anfragen*. Dabei werden in der FROM-Zeile mehrere Relationen aufgeführt. Prinzipiell kann man sich einen Join als kartesisches Produkt der beteiligten Relationen vorstellen (Theorie: siehe Vorlesung). Dabei erhält man eine Relation, deren Attributmenge die *disjunkte* Vereinigung der Attributmengen der beteiligten Relationen ist.

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

<predicate> kann dabei Attribute aus allen beteiligten Relationen beinhalten, insbesondere ist ein Vergleich von Attributen aus mehreren Relationen möglich: Treten in verschiedenen Relationen Attribute gleichen Namens auf, so werden diese durch (Relation.Attribut) qualifiziert.

Beispiel: Gesucht seien die Länder, in denen es Städte namens Victoria gibt:

```
SELECT Country.Name
FROM City, Country
WHERE City.Name = 'Victoria'
AND City.Country = Country.Code;
```

Country.Name
Canada
Seychelles

Einen Spezialfall dieser Art Anfragen zu stellen, bilden Anfragen, die die Verbindung einer Relation mit sich selbst benötigen. In diesem Fall müssen *Aliase* vergeben werden, um die verschiedenen Relationsausprägungen voneinander zu unterscheiden:

Beispiel: Gesucht seien beispielsweise alle Städte, deren Namen in der Datenbasis mehrfach vorkommen zusammen mit den Ländern, in denen sie jeweils liegen.

```
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
⋮	⋮	⋮

1.6.2 Subqueries

Subqueries sind eine andere Möglichkeit zur Formulierung von Anfragen, die mehrere Relationen umfassen. Dabei werden mehrere `SELECT`-Anfragen ineinander geschachtelt. Meistens stehen Subqueries dabei in der `WHERE`-Zeile.

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

wobei `<subquery>` eine `SELECT`-Anfrage (*Subquery*) ist und

- falls `<subquery>` nur einen einzigen Wert liefert, ist `rel` eine der Vergleichsrelationen $\{=,<,>,<=,>=\}$,
- falls `<subquery>` mehrere Werte/Tupel liefert, ist `rel` entweder `IN` oder von der Form Φ `ANY` oder Φ `ALL` wobei Φ eine der o.g. Vergleichsrelationen ist.

Bei den Vergleichsrelationen muss die Subquery ein einspaltiges Ergebnis liefern, bei `IN` sind im SQL-Standard und in ORACLE seit Version 8 auch mehrere Spalten erlaubt:

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

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

Name	Country	Province
Ajaccio	F	Corse
Karlstad	S	Värmland
San Diego	USA	California
..	..	..

Beispiel: Auch damit lassen sich alle Länder bestimmen, in denen es eine Stadt namens Victoria gibt:

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

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
..	..	..

Dennoch können `ANY` und `ALL` meistens durch effizientere Anfragen unter Verwendung der Aggregatfunktionen `MIN` und `MAX` ersetzt werden:

```
< ALL (SELECT <attr>) ≡ < (SELECT MIN(<attr>)) ,
< ANY (SELECT <attr>) ≡ < (SELECT MAX(<attr>)) ,
```

analog für die anderen Vergleichsrelationen.

Beispiel: Das obige Beispiel kann z. B. effizienter formuliert werden:

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

Man unterscheidet unkorrelierte Subqueries und korrelierte Subqueries: Eine Subquery ist *unkorreliert*, wenn sie unabhängig von den Werten des in der umgebenden Anfrage verarbeiteten Tupels ist. Solche Subqueries dienen – wie in den obigen Beispielen – dazu, eine Hilfsrelation oder ein Zwischenergebnis zu bestimmen, das für die übergeordnete Anfrage benötigt wird. In diesem Fall wird die Subquery vor der umgebenden Anfrage *einmal* ausgewertet, und das Ergebnis wird bei der Auswertung der `WHERE-Klausel` der äußeren Anfrage verwendet. Durch diese streng sequenzielle Auswertung ist eine Qualifizierung mehrfach vorkommender Attribute *nicht erforderlich* (jedes Attribut ist eindeutig der momentan ausgewerteten `FROM`-Klausel zugeordnet).

Eine Subquery ist *korreliert*, wenn in sie von Attributwerten des gerade von der umgebenden Anfrage verarbeiteten Tupels abhängig ist. In diesem Fall wird die Subquery *für jedes Tupel der umgebenden Anfrage* einmal ausgewertet. Damit ist ist eine 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
⋮	⋮

1.6.3 Subqueries mit EXISTS

Das Schlüsselwort `EXISTS` bzw. `NOT EXISTS` bildet den Existenzquantor nach. Subqueries mit `EXISTS` sind i.a. korreliert um eine Beziehung zu den Werten der äußeren Anfrage herzustellen.

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

Beispiel: Gesucht seien diejenigen Länder, für die Städte mit mehr als 1 Mio. Einwohnern in der Datenbasis abgespeichert sind.

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

Name
Serbia and Montenegro
France
Spain
Austria
Czech Republic
⋮

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;
```

Die Subquery mit `EXISTS` ist allerdings in jedem Fall korreliert, also evtl. deutlich ineffizienter als die Anfrage mit `IN`.

1.6.4 Subqueries in der FROM-Zeile

Zusätzlich zu den bisher gezeigten Anfragen, wo die Subqueries immer in der `WHERE`-Klausel verwendet wurden, sind [in einigen Implementierungen; im SQL-Standard ist es nicht vorgesehen] auch Subqueries in der `FROM`-Zeile erlaubt. In der `FROM`-Zeile werden Relationen angegeben, deren Inhalt verwendet werden soll:

```
SELECT   Attribut(e)
FROM     Relation(en)
WHERE    Bedingung(en)
```

Diese können anstelle von Relationsnamen auch durch SELECT-Statements gegeben sein. Dies ist insbesondere sinnvoll, wenn Werte, die auf unterschiedliche Weise aus einer oder mehreren Tabellen gewonnen werden, zueinander in Beziehung gestellt werden oder weiterverarbeitet werden sollen:

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

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

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

Population-Urban_Residents
4620065771

Dies ist 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);
```

sum(pop_biggest)
273837106

1.7 Data Dictionary

Datenbanksysteme enthalten zusätzlich zu der eigentlichen Datenbank Tabellen, die *Metadaten*, d.h. Daten über die Datenbank enthalten. Diese werden als *Data Dictionary* bezeichnet. Im folgenden werden einige Tabellen des Data Dictionaries unter ORACLE beschrieben:

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

TABLE_NAME	COMMENTS
ALL_ARGUMENTS	Arguments in object accessible to the user
ALL_CATALOG	All tables, views, synonyms, sequences accessible to the user
ALL_CLUSTERS	Description of clusters accessible to the user
⋮	⋮

Von Interesse sind evtl. die folgenden Data-Dictionary-Tabellen:

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

`ALL_CATALOG`: Enthält alle Tabellen, Views und Synonyme, die einem Benutzer zugänglich sind, u. a. auch einen Eintrag `<username> COUNTRY TABLE`.

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 `USER_CATALOG`, `USER_TABLES` etc., meistens existieren für `USER_...` auch Abkürzungen, etwa `OBJ` für `USER_OBJECTS`.

ALL_USERS: Enthält Informationen über alle Benutzer der Datenbank.

Außerdem kann man Informationen über die Definition der einzelnen Tabellen und Views mit dem Befehl `DESCRIBE <table>` oder kurz `DESC <table>` abfragen. Dies ist insbesondere sinnvoll, wenn der Name einer Spalte länger ist, als der Datentyp ihrer Einträge (Im Output kürzt SQL den Namen auf die Spaltenlänge). So erscheint etwa die Ausgabe der Tabelle *City* folgendermaßen:

```
NAME                      COUN          PROVINCE POPULATION LONGITUDE LATITUDE
------------------------- ---- ----------------- ---------- --------- --------
Stuttgart                 D    Baden-Wuerttemberg     588482       9,1     48,7
```

Die zweite Spalte heißt jedoch `COUNTRY`, wie eine kurze Nachfrage beim Data Dictionary bestätigt:

`DESC City;`

Name	**NULL?**	**Typ**
NAME	NOT NULL	VARCHAR2(35)
COUNTRY	NOT NULL	VARCHAR2(4)
PROVINCE	NOT NULL	VARCHAR2(32)
POPULATION		NUMBER
LONGITUDE		NUMBER
LATITUDE		NUMBER