Datenbankpraktikum SQL

Wolfgang May
may@informatik.uni-freiburg.de
Institut für Informatik
Universität Freiburg

Freiburg, Oktober 2001

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 objekt-relationaler) 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 zum Wintersemester 98/99 wurde auch die Datenbasis erweitert.

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

[2] in welchem der Autor 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 Unterstriche 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>)` $\equiv$ `< (SELECT MIN(<attr>))` ,
`< ANY (SELECT <attr>)` $\equiv$ `< (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 `ROM`-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

2 SCHEMA-DEFINITION

Das *Datenbankschema* umfasst alle Informationen *über* die Datenbank (d.h., alles *außer* den eigentlichen Daten) und wird mit Hilfe der *Data Definition Language (DDL)* manipuliert.

Die DDL-Kommandos in SQL umfassen die Befehle `CREATE` / `ALTER` / `DROP` für das Anlegen, Ändern und Entfernen von Schemaobjekten. Relationen können als *Tabellen* (vgl. Abschnitt 2.1) *extensional* gespeichert werden, oder als *Views (Sichten)* (vgl. Abschnitt 2.2) *intensional* definiert sein.

Weiterhin enthält das Datenbankschema verschiedene Arten von Bedingungen, u.a. Wertebereichbedingungen einzelner Attribute, Bedingungen innerhalb einer Tabelle (intrarelational), Schlüsselbedingungen, sowie Bedingungen die mehrere Tabellen betreffen (interrelational).

Ferner werden in Abschnitt 13 die Konzepte `INDEX` und `CLUSTER` zur optimierten Speicherung von Tabellen, sowie in Abschnitt 14 `PROZEDUREN`, `FUNKTIONEN` und `TRIGGER` eingeführt. Zur Schemadefinition gehört außerdem noch die Vergabe von Zugriffsrechten durch `GRANT`-Befehle (vgl. Abschnitt 10) sowie die Definition von `SYNONYM`en (ebenfalls Abschnitt 10).

2.1 Definition von Tabellen

In der einfachsten Form besteht eine Tabellendefinition nur aus der Angabe der Attribute sowie ihres Wertebereichs:

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

Die Bezeichnungen der erlaubten Datentypen können dabei abhängig von dem verwendeten System unterschiedlich sein. Oracle erlaubt unter anderem

`CHAR`(n)**:** Zeichenkette fester Länge n. Kürzere Zeichenketten werden ggf. aufgefüllt.

`VARCHAR`(n)**:** Zeichenkette variabler Länge $\leq n$. Jede Zeichenkette benötigt nur soviel Platz, wie sie lang ist. Auf `CHAR` und `VARCHAR` ist der Operator || zur Konkatenation von Strings definiert. In Oracle ist `VARCHAR2` eine verbesserte Implementierung des Datentyps `VARCHAR` früherer Versionen.

`NUMBER`**:** Zahlen mit Betrag $1.0 \cdot 10^{-30} \leq x < 1.0 \cdot 10^{125}$ und 38-stelliger Genauigkeit. Auf `NUMBER` sind die üblichen Operatoren $+$, $-$, $*$ und / sowie die Vergleiche $=$, $>$, $>=$, $<=$ und $<$ erlaubt. Außerdem gibt es `BETWEEN x AND y`. Ungleichheit wird je nach System (auch innerhalb Oracle noch plattformabhängig) mit $!=$, $\wedge =$, $\neg =$ oder $<>$ beschrieben.

`LONG`**:** Zeichenketten von bis zu 2GB Länge.

`DATE`**:** Datum und Zeiten: Jahrhundert – Jahr – Monat – Tag – Stunde – Minute – Sekunde. U.a. wird auch Arithmetik für solche Daten angeboten (vgl. Abschnitt 4).

weitere Datentypen findet man im ORACLE SQL Reference Manual.

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

```
CREATE TABLE City
      ( Name          VARCHAR2(35),
        Country       VARCHAR2(4),
        Province      VARCHAR2(32),
        population    NUMBER,
        Longitude     NUMBER,
        Latitude      NUMBER );
```

Darüberhinaus können mit der Tabellendefinition noch Eigenschaften und Bedingungen an die jeweiligen Attributwerte formuliert werden. Dabei ist weiterhin zu unterscheiden, ob eine Bedingung nur ein einzelnes oder mehrere Attribute betrifft. Erstere sind unter anderem Wertebereichseinschränkungen, die Angabe von Default-Werten, oder die Forderung, dass ein Wert angegeben werden muss. Bedingungen an die Tabelle als Ganzes ist z.B. die Angabe von Schlüsselbedingungen.

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

(Die in `[...]` aufgeführten Teile der Spezifikation sind optional)

- Als `DEFAULT <value>` kann dabei ein beliebiger Wert des erlaubten Wertebereichs spezifiziert werden, der eingesetzt wird, falls vom Benutzer an dieser Stelle kein Wert angegeben wird.

  Beispiel 1 (`DEFAULT`-Wert) Ein Mitgliedsland einer Organisation wird als volles Mitglied angenommen, wenn nichts anderes bekannt ist:

  ```
  CREATE TABLE is_member
        ( Country       VARCHAR2(4),
          Organization  VARCHAR2(12),
          Type          VARCHAR2(30)
                        DEFAULT 'member')

  INSERT INTO is_member VALUES
     ('CZ', 'EU', 'membership applicant');
  INSERT INTO is_member (Land,Organization)
     VALUES ('D', 'EU');
  ```

Country	Organization	Type
CZ	EU	membership applicant
D	EU	member
⋮	⋮	⋮

  □

- `<colConstraint>` ist eine Bedingung, die nur *eine* Spalte betrifft, während `<tableConstraint>` mehrere Spalten betreffen kann. Jedes `<colConstraint>` bzw. `<tableConstraint>` ist von der Form

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

  Dabei ist der Teil `CONSTRAINT <name>` optional, fehlt er, so wird dem Constraint ein interner Name zugeordnet. Im allgemeinen ist es sinnvoll, `NULL`-, `UNIQUE`-, `CHECK`- und `REFERENCES`-Constraints einen Namen zu geben, um sie ggf. ändern oder löschen zu können (`PRIMARY KEY` kann man ohne Namensnennung löschen, vgl. Abschnitt 7).

Beide Arten von Constraints verwenden in `<bedingung>` prinzipiell dieselben Schlüsselwörter:

1. `[NOT] NULL`: Gibt an, ob die entsprechende Spalte Nullwerte enthalten darf (nur als `<colConstraint>`).
2. `UNIQUE (<column-list>)`: Fordert, dass jeder Wert nur einmal auftreten darf.
3. `PRIMARY KEY (<column-list>)`: Bestimmt diese Spalte zum Primärschlüssel der Relation.
   Damit entspricht `PRIMARY KEY` der Kombination aus `UNIQUE` und `NOT NULL`. (`UNIQUE` wird von `NULL`-Werten *nicht* unbedingt verletzt, während `PRIMARY KEY` `NULL`-Werte *verbietet*.)
   Da auf jeder Relation nur ein `PRIMARY KEY` definiert werden darf, wird die Schlüsseleigenschaft von Candidate Keys üblicherweise über `NOT NULL` und `UNIQUE` definiert.
   In ORACLE wird bei einer `PRIMARY KEY`-Deklaration automatisch ein Index über die beteiligten Attribute angelegt.
4. `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [ON DELETE CASCADE]`: gibt an, dass ein Attributtupel Fremdschlüssel (d.h. Schlüssel in einer (nicht unbedingt anderen!) Relation) ist; vgl. Abschnitt 8 (Referentielle Integrität).
   Das referenzierte Attributtupel `<table>(<column-list2>)` muss in der Definition von `<table>` als `PRIMARY KEY` deklariert sein.
   Eine `REFERENCES`-Bedingung wird durch `NULL`-Werte nicht verletzt; sogar dann nicht, wenn der Rest absolut sinnlos ist:

   Beispiel 2 In der Tabelle City(Name,Country,Province) müssen die Einträge (Country,Province) einen entsprechenden Eintrag in *Province* referenzieren. Dennoch kann man ('Brussels','XX',NULL) in die Relation *City* einfügen, obwohl 'XX' kein gültiges Land ist. □

   Der optionale Zusatz `ON DELETE CASCADE` wird in Abschnitt 8 (Referentielle Integrität) behandelt.
5. `CHECK (<condition>)`: Keine Zeile darf `<condition>` verletzen. `NULL`-Werte ergeben dabei ggf. ein *unknown*, also *keine Bedingungsverletzung*.

Bei einem `<colConstraint>` ist die Spalte implizit bekannt, somit fällt der `(<column-list>)` Teil weg.

Beispiel 3 (`PRIMARY` und `UNIQUE`) Für die Tabelle *Country* wird das Kürzel *Code* als `PRIMARY KEY` definiert, und die Eindeutigkeit des Namens durch `NOT NULL` (column constraint) und `UNIQUE (Name)` (table constraint) als Candidate Key garantiert:

```
CREATE TABLE Country
  ( Name         VARCHAR2(32) NOT NULL UNIQUE,
    Code         VARCHAR2(4) PRIMARY KEY);
```

Code als `PRIMARY KEY` wird in den verschiedenen Relationen referenziert, d.h. ist dort *Fremdschlüssel*:

```
CREATE TABLE is_member
```

```
( Country       VARCHAR2(4) CONSTRAINT MemberRefsCountry
    REFERENCES Country(Code),
  Organization  VARCHAR2(12) CONSTRAINT MemberRefsOrg
    REFERENCES Organization(Abbreviation),
  Type          VARCHAR2(30) DEFAULT 'member');
```
□

Beispiel 4 (`CHECK`-Bedingungen) `CHECK`-Bedingungen werden verwendet um Werte von Attributen einzuschränken; ihre Syntax ist für column constraints dieselbe wie für table constraints. Die vollständige Definition der Relation *City* mit Bedingungen und Schlüsseldeklaration lautet wie folgt:

```
CREATE TABLE City
  ( Name        VARCHAR2(35),
    Country VARCHAR2(4) References Country(Code),
    Province VARCHAR2(32),
    Population NUMBER CONSTRAINT CityPop CHECK (Population >= 0),
    Longitude NUMBER CONSTRAINT CityLong
              CHECK ((Longitude >= -180) AND (Longitude <= 180)) ,
    Latitude NUMBER CONSTRAINT CityLat
              CHECK ((Latitude >= -90) AND (Latitude <= 90)) ,
    CONSTRAINT CityKey PRIMARY KEY (Name, Country, Province),
    FOREIGN KEY (Country,Province)
              REFERENCES Province (Country,Name));
```

Die Bedingungen `CityPop`, `CityLong` und `CityLat` definieren Wertebereichseinschränkungen, `CityKey` deklariert `(Name, Country, Province)` als Schlüssel. Dies verlangt a) dass der jeweilige Wert nur einmal auftreten darf und b) ein Wert angegeben werden muss. □

Im Zusammenhang mit `FOREIGN KEY` bzw. `REFERENCES`-Deklarationen ist folgendes zu beachten:

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`–Klausel enthält, erstellt wird, muss `<table>` bereits definiert und `<column-list>` dort als `PRIMARY KEY` deklariert sein.
- Beim Einfügen von Daten müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.
- Zu diesem Zweck können Constraints zeitweise deaktiviert werden (vgl. Abschnitt 7).

2.2 Definition von Views

Mit *Views (Sichten)* können die in einer Datenbank gespeicherten Daten einem Benutzer in einer von der tatsächlichen Speicherung verschiedenen Form dargestellt werden. Dies ist z.B. wünschenswert, wenn verschiedenen Benutzern verschiedene Ausschnitte aus der Datenbasis oder verschieden aufbereitete Daten zur Verfügung gestellt werden sollen. Bei Anfragen kann auf Sichten genauso wie auf Tabellen zugegriffen werden. Lediglich Änderungsoperationen sind nur in eingeschränktem Umfang möglich. Views werden nicht zum Zeitpunkt ihrer Definition berechnet, sondern jedesmal, wenn auf sie zugegriffen wird. Sie spiegeln also stets den aktuellen Zustand der ihnen zugrundeliegenden Relationen wieder.

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
<select-clause>;
```

Durch die (optionale) Angabe von `OR REPLACE` wird eine View, falls es bereits definiert ist, einfach durch die neue Definition ersetzt.

Beispiel: Angenommen, ein Benutzer benötigt häufig die Information, welche Stadt in welchem Land liegt, sei jedoch weder an Landeskürzeln noch Einwohnerzahlen interessiert. Dann ist es sinnvoll, eine entsprechende Sicht zu definieren.

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

Wenn der Benutzer nun nach allen Städten in Kamerun sucht, so kann er die folgende Anfrage stellen:

Views können außerdem zur Zugriffskontrolle eingesetzt werden (vgl. Abschnitt 10).

2.3 Löschen von Tabellen und Views

Tabellen bzw. Views werden mit `DROP TABLE` bzw. `DROP VIEW` gelöscht:

```
DROP TABLE <table-name> [CASCADE CONSTRAINTS];
DROP VIEW <view-name>;
```

- Tabellen müssen nicht leer sein, wenn sie gelöscht werden sollen.
- eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, kann mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden.
- Mit `DROP TABLE <table> CASCADE CONSTRAINTS` wird eine Tabelle mit allen auf sie zeigenden referentiellen Integritätsbedingungen gelöscht.

Veränderungen an der Struktur von Tabellen/Views (`ALTER`) werden in Abschnitt 7 behandelt.

3 EINFÜGEN, LÖSCHEN UND ÄNDERN VON DATEN

3.1 Einfügen von Daten

Das Einfügen von Daten erfolgt mittels der `INSERT`-Klausel. Von dieser Klausel gibt es im wesentlichen zwei Ausprägungen: Daten können von Hand einzeln eingefügt werden, oder das Ergebnis einer Anfrage kann in eine Tabelle übernommen werden:

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

oder

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

Fehlt die optionale Angabe `<column-list>`, so werden die Attributwerte in der durch die Tabellendefinition gegebenen Reihenfolge eingesetzt. Enthält eine `<value-list>` weniger Werte als in der Tabellendefinition bzw. `<column-list>` angegeben, so wird der Rest mit Nullwerten aufgefüllt.

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

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

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

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

Beim Einfügen von Daten in eine Tabelle, für die Fremdschlüsselbedingungen definiert sind müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.

Bemerkung: Es ist allerdings sinnvoller, die oben genannte Tabelle `Metropolis` als View anzulegen:

```
CREATE VIEW Metropolis (Name, Country, Population) AS
SELECT Name, Country, Population
FROM City
WHERE Population > 1000000;
```

Der Unterschied liegt darin, dass ein View *Metropolis zu jeder Zeit* aktuell (bezüglich der Tabelle *City*) ist, während die Lösung mit der festen Tabelle jeweils die Situation zu dem Zeitpunkt beschreibt, wo

das `INSERT ...`-Statement ausgeführt wurde.

3.2 Löschen von Daten

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

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

Mit einer leeren `WHERE`-Bedingung kann man z.B. alle Tupel einer Tabelle löschen (die Tabelle bleibt bestehen, sie kann mit `DROP TABLE` entfernt werden):

```
DELETE FROM City;
```

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

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

Beim Löschen eines referenzierten Tupels muss die referentielle Integrität erhalten bleiben (vgl. Abschnitt 8).

3.3 Ändern von Daten

Die Änderung einzelner Tupel erfolgt mittels der `UPDATE`-Operation.

Syntax:

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

Beispiel: Sankt-Peterburg wird in Leningrad umbenannt, außerdem steigt die Einwohnerzahl um 1000:

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

Oft ist `<subquery>` eine *korrelierte Subquery*, die abhängig von den Werten des zu verändernden Tupels die neuen Werte der zu verändernden Attribute berechnet:

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

```
UPDATE Country
```

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

4 ZEIT- UND DATUMSANGABEN

Für jeden Wert vom Typ `DATE` werden Jahrhundert, Jahr, Monat, Tag, Stunde, Minute und Sekunde gespeichert. Das Default-Format wird mit dem Parameter `NLS_DATE_FORMAT` gesetzt, als Default ist `'DD-MON-YY'` eingestellt, d.h. man gibt ein Datum z.B. als `'20-Okt-97'` an. Für die fehlenden Komponenten setzt ORACLE dabei Default-Werte ein.

- Gibt man ein Datum ohne Zeitkomponente an, wird dafür als Default 12:00:00 a.m. eingesetzt.
- Gibt man für ein Datum nur eine Zeitkomponente an, so wird der erste Tag des aktuellen Monats des aktuellen Jahres eingesetzt.
- partielle Zeit-/Datumsangaben werden nach diesem Muster aufgefüllt.

Die zulässigen Formate für Datums- und Zeitangaben findet man im ORACLE SQL Reference Guide.

MONDIAL enthält die Gründungsdaten der Länder in der Relation *politics*. Um diese Daten eingeben zu können muss u.a. die Angabe des Jahres auf 4 Ziffern erweitert werden:

```
CREATE TABLE Politics
  ( Country VARCHAR2(4),
    Independence DATE,
    Government VARCHAR2(120));
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';

INSERT INTO politics VALUES ('AL','28 11 1912','emerging democracy');
INSERT INTO politics VALUES ('A','12 11 1918','federal republic');
INSERT INTO politics VALUES ('B','04 10 1830','constitutional monarchy');
INSERT INTO politics VALUES ('D','18 01 1871','federal republic');
```

Beispiel 5 Alle Länder, die zwischen 1200 und 1600 gegründet wurden lassen sich damit folgendermaßen bestimmen:

```
SELECT Country, Independence FROM politics
    WHERE Independence BETWEEN '01 01 1200' AND '31 12 1599';
```

□

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

Eine Konversion von `CHAR` und `DATE` ist mit `TO_DATE` und `TO_CHAR` möglich. Im Zusammenspiel mit `NLS_DATE_FORMAT` kann man damit die einzelnen Komponenten von Datumsangaben extrahieren:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM';
```

```
SELECT Country, TO_DATE(TO_CHAR(Independence,'DD MM'),'DD MM') FROM Politics;
```

Country	TO_DA
RSM	01 01
YU	11 04
SK	01 01
SLO	25 06
E	01 01

Oracle bietet einige Funktionen um mit dem Datentyp `DATE` zu arbeiten:

- `SYSDATE` liefert das aktuelle Datum.
- Addition und Subtraktion von Absolutwerten auf `DATE` ist erlaubt, Zahlen werden als Tage interpretiert: `SYSDATE` $+ 1$ ist morgen, `SYSDATE` $+ (10/1440)$ ist "in zehn Minuten".
- `ADD_MONTHS`(d, n) addiert n Monate zu einem Datum d. Ist d der letzte Tag eines Monats, so ist das Ergebnis der letzte Tag des resultierenden Monats.[1]
- `LAST_DAY`(d) ergibt den letzten Tag des in d angegebenen Monats.
- `MONTHS_BETWEEN`(d_1, d_2) gibt an, wieviele Monate zwischen den angegebenen Daten liegen (Dezimalwert, ein Monat wird mit 31 Tagen angenommen).
- weitere Funktionen siehe Oracle SQL Reference Guide.

[1] Interessant: `ADD_MONTHS`(30-JAN-97,1).

5 OBJEKTTYPEN ALS KOMPLEXE ATTRIBUTE UND GESCHACHTELTE TABELLEN

Neben den bereits genannten Standard-Datentypen können mit Hilfe von ORACLE's objektorientierter Funktionalität komplexe Attribute, geschachtelte Tabellen (oft auch als *Collections* bezeichnet) sowie "richtige" Objekttypen definiert werden. "Richtige" Objekttypen soll hier bedeuten, dass ein Objekt Methoden bereitstellt, mit dem seine Daten manipuliert und abgefragt werden können. Dazu sind prozedurale Konzepte notwendig, die erst in Abschnitt 14 vorgestellt werden – entsprechend werden "richtige" Objekttypen erst in Abschnitt IV behandelt.

Bereits ohne Nutzung prozeduraler Konzepte erlaubt der "strukturelle" objektorientierte Ansatz die Erweiterung der deklarativen Konzepte um komplexe Attribute und geschachtelte Tabellen. Erstere sind bereits aus klassischen nicht-objektorientierten Programmiersprachen als *Records* bekannt, letztere stellen sich als "syntactic sugar" für Joins, Subqueries und Gruppierung heraus.

Beide Konzepte werden in ORACLE durch die Einführung von Schemaobjekten des Typs `TYPE` implementiert. `TYPE`n bestehen aus zwei Teilen:

- einer *Spezifikation*, die die Signatur des Typs angibt und mit `CREATE TYPE` erstellt wird. Dabei werden die Attribute (wie von `CREATE TABLE` bekannt) sowie die Signatur der Methoden definiert.
- ggf. einem *Body*, der die Implementierung der Methoden (in PL/SQL formuliert; vgl. Abschnitt 14) enthält und mit `CREATE TYPE BODY` erstellt wird. Der Body muss nur angegeben werden, falls für ein Objekt Methoden definiert werden sollen.

Komplexe Attribute und geschachtelte Tabellen sind Typen ohne Methoden und ohne Body:

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (  <attr> <datatype>,
     ⋮
     <attr> <datatype>);
/
 oder
CREATE [OR REPLACE] TYPE <type> AS TABLE OF <type'>;
/
```

Hierbei entsprechen die Attribute `<attr>` den Spalten `<col>` bei der Definition von Tabellen – man sieht, es ist eigentlich in erster Linie einmal eine Frage der Terminologie. Unter ORACLE ist es wichtig, die Eingabe zusätzlich zu den ";" noch durch "/" abzuschließen.

5.1 Komplexe Attribute

Ein komplexes Attribut ist ein Attribut, das aus mehreren Teilen besteht, z.B. "geographische Koordinaten" als Paar (Länge, Breite). Die Definition eines komplexen Attributs geschieht analog zu der Definition einer Tabelle.

Beispiel 6 (Komplexe Attribute) Geographische Koordinaten werden wie folgt als komplexes Attribut modelliert:

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

Komplexe Attribute werden dann genauso wie andere Attribute in Tabellendefinitionen verwendet:

```
CREATE TABLE Mountain
  (Name        VARCHAR2(20),
   Height      NUMBER,
   Coordinates GeoCoord);
```

Durch `CREATE TYPE <type> AS OBJECT (...)` mit n Attributen wird automatisch eine n-stellige *Konstruktormethode* mit Namen `<type>` definiert, mit der Objekte dieses Typs generiert werden können:

```
INSERT INTO Mountain VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

Entsprechend ist auch die Ausgabe:

Beispiel:

```
SELECT * FROM Mountain;
```

Name	Height	Coordinates(Longitude, Latitude)
Feldberg	1493	GeoCoord(8,48)

Die einzelnen Komponenten von komplexen Attributen werden in Anfragen mit der aus anderen Programmiersprachen bereits bekannten *dot*-Notation adressiert:

Beispiel:

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

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	8	48

Da komplexe Attribute prinzipiell *Objekte* sind (vgl. Abschnitt 15), können sie (zumindest in ORACLE 8.0.3 und 8.0.4) nur unter Verwendung eines Tabellen-Alias angesprochen werden. Die obige Anfrage muss umformuliert werden zu

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

(Bemerkung: "normale" Qualifizierung als `Mountain.Coordinates.Longitude` genügt nicht.)

5.2 Geschachtelte Tabellen

Manchmal ist es sinnvoll, in einem Relation ein Attribut zu definieren, das nicht höchstens einen Wert annehmen kann, sondern eine *Menge von Werten*:

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

Mit geschachtelten Tabellen lassen sich Zusammenhänge modellieren, die sonst in verschiedenen Tabellen abgelegt werden in Anfragen per Join ausgewertet werden müssten. Eine solche Lösung bietet sich für mengenwertige Attribute oder 1:n-Beziehungen an, oft sogar für m:n-Beziehungen. Dazu wird zuerst der Typ der Tupel der inneren Tabelle als Objekt definiert, dann der Tabellentyp der inneren Tabelle auf Basis des eben definierten Tupeltyps. Dieser kann dann bei der Definition der äußeren Tabelle verwendet werden. Dabei wird angegeben, dass es sich um eine geschachtelte Tabelle handelt und diese bekommt einen Namen zugewiesen:

```
CREATE [OR REPLACE] TYPE <inner_type> AS OBJECT (...);
CREATE [OR REPLACE] TYPE <inner_table_type> AS
  TABLE OF <inner_type>;
/
CREATE TABLE <table_name>
  (  ... ,
     <table-attr> <inner_table_type> ,
     ... )
NESTED TABLE <table-attr> STORE AS <name >;
```

In der TERRA^{++}-Datenbank sind z.B. die in den einzelnen Ländern gesprochenen Sprachen durch geschachtelte Tabellen modelliert:

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

ORACLE behandelt geschachtelte Tabellen ähnlich wie Cluster (vgl. Abschnitt 13): Es werden zwei *logische* Tabellen angelegt, die in demselben physikalischen Datensegment abgelegt werden.

Geschachtelte Tabellen unterstützen auch keine (referentiellen) Integritätsbedingungen, die den Inhalt der inneren Tabelle mit dem umgebeneden Tupel oder einer anderen Tabelle verknüpfen.

Analog zu komplexen Attributen ist auch für geschachtelte Tabellen eine Konstruktormethode desselben Namens definiert, die als Argument eine *Liste* von Objekten des Type der Listenelemente (natürlich wiederum mit der entsprechenden Konstruktormethode des Listenelement-Typs erzeugt) enthält. Diese wird dazu verwendet, `VALUES` in eine solche Tabelle einzufügen (Zu `INSERT INTO <table> SELECT ...` siehe weiter unten):

```
INSERT INTO NLanguage VALUES ('SK',
  Languages_list
  ( Language_T('Slovak',95),
    Language_T('Hungarian',5)));
```

Die übliche Anfragesyntax liefert geschachtelte Tabellen jeweils als ganzes zurück – d.h. als Liste von Objekten unter Verwendung der entsprechenden Objektkonstruktoren:

Beispiel:

```
SELECT *
FROM NLanguage
WHERE Country='CH';
```

Country	Languages(Name, Percentage)
CH	Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Beispiel:

```
SELECT Languages
FROM NLanguage
WHERE Country='CH';
```

Languages(Name, Percentage)
Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Um *innerhalb* geschachtelter Tabellen arbeiten zu können werden diese mit dem Schlüsselwort `THE` angesprochen. `THE (SELECT ...)` teilt ORACLE mit, dass das Ergebnis einer Anfrage eine Tabelle ist:

```
SELECT ...
FROM THE (<select-statement>)
WHERE ... ;

INSERT INTO THE (<select-statement>)
VALUES ... / SELECT ... ;

DELETE FROM THE (<select-statement>)
WHERE ... ;
```

wobei `<select-statement>` eine *Tabelle* selektieren muss.

Beispiel:

```
SELECT Name, Percentage
FROM THE  (SELECT Languages
           FROM NLanguage
           WHERE Country='CH');
```

Name	Percentage
German	65
French	18
Italian	12
Romansch	1

```
INSERT INTO THE (SELECT Languages FROM NLanguage where Country= A.Country)
   VALUES ('Hungarian',2);
```

Wenn man das Ergebnis einer Subquery, die eine Menge von Tupel liefert, als geschachtelte Tabelle in eine Tabelle einfügen will, muss man diese mit Hilfe der Befehle `CAST` und `MULTISET` strukturieren:

```
INSERT INTO <table>
VALUES (..., CAST(MULTISET(SELECT ...) AS <nested-table-type>),...);
```

wobei `<nested-table-type>` der Typ der geschachtelten Tabelle an der entsprechenden Stelle ist:

Dennoch ist `CAST/MULTISET` zum Füllen von geschachtelten Tabellen nicht so geeignet wie man auf den ersten Blick vermuten könnte:

Beispiel 7 (`CAST` und `MULTISET`) Mit

```
INSERT INTO NLanguage - zulässige, aber falsche Anfrage !!!!
 (SELECT Country,
         CAST(MULTISET(SELECT Name, Percentage
                       FROM Language
                       WHERE Country = A.Country)
              AS Languages_List)
 FROM Language A);
```

wird jedes Tupel (Land, Sprachenliste) x-mal eingefügt, wobei x die Anzahl der für das Land gespeicherten Sprachen ist. □

Daher ist es notwendig, einzeln vorzugehen:

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

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

Liefert eine Subquery bereits eine Tabelle, kann diese ohne casting direkt (unter Verwendung von `THE`) eingefügt werden:

```
INSERT INTO <table>
VALUES (. . . , THE(SELECT <attr>
                    FROM <table'>
                    WHERE . . .));
```

wobei das Attribut <attr> von <table'> eine geschachtelte Tabelle geeigneten Typs ist.

```
INSERT INTO NLanguage VALUES
 ('CHXX', THE (SELECT Languages from NLanguage
                WHERE Country='CH'));
```

Einige dieser Features scheinen in der derzeitigen Version noch nicht voll ausgereift zu sein. So darf eine Unterabfrage nur *eine einzige* geschachtelte Tabelle zurückgeben. Damit ist es also nicht möglich in Abhängigkeit von dem Tupel der äußeren Tabelle die jeweils passende innere Tabelle auszuwählen, bzw. nebeneinander Daten aus der äußeren und der inneren Tabelle auszugeben:

Beispiel 8 (Unzulässige Anfragen)

- Es sollen alle Länder ausgegeben werden, in denen Deutsch gesprochen wird:
  ```
  SELECT Country - UNZULÄSSIG !!!! FROM NLanguage A,
  THE SELECT Languages FROM NLanguage B
  WHERE B.Country=A.Country) WHERE Name='German';
  ```
- Es sollen alle Paare (L, S) ausgegeben werden, so dass die Sprache S im Land L gesprochen wird:
  ```
  SELECT Country, Name, Percentage - UNZULÄSSIG !!!!
  FROM NLanguage A,
       THE ( SELECT Languages
             FROM NLanguage B
             WHERE B.Country=A.Country);
  ```
  □

Die zweite der oben aufgeführten Anfragen ist zulässig, wenn man nur ein Land selektiert – d.h., nur eine geschachtelte Tabelle verwendet wird:

```
SELECT Country, Name, Percentage
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE A.Country = 'CH';
```

Die erste der oben aufgeführten Anfragen kann mit Hilfe einer Subquery und des `TABLE (<attr>)`-Konstrukts (wobei <attr> ein tabellenwertiges Attribut sein muss) gelöst werden. Dieses erlaubt, die innere Tabelle in einer Subquery zu verwenden:

Beispiel:

```
SELECT Country
FROM NLanguage
WHERE EXISTS
  (SELECT *
   FROM TABLE (Languages)
   WHERE Name='German');
```

Country
A
B
CH
D
NAM

wobei hier `(Languages)` immer die innere Tabelle *NLanguage.Languages* des aktuellen Tupels adressiert.

Da `TABLE (<attr>)` jedoch nicht in der `FROM`-Zeile des äußeren `SELECT`-Statements vorkommen kann, kann man auch keine Attribute der inneren Tabelle für das äußere `SELECT`-Statement angeben. Entspre-

chend ist es nicht möglich, in der obigen Ausgabe auch den prozentualen Anteil in den verschiedenen Ländern auszugeben.

Eine etwas bessere Formulierung liefert der `CURSOR`-Operator (verwandt mit dem PL/SQL-Cursorkonzept):

Beispiel:

```
SELECT Country,
       cursor (SELECT *
               FROM TABLE (Languages))
FROM NLanguage;
```

Country	**CURSOR(SELECT...)**
CH	CURSOR STATEMENT : 2
NAME	**PERCENTAGE**
French	18
German	65
Italian	12
Romansch	1

Dieses Problem, lässt sich erst mit PL/SQL (vgl. Abschnitt 14) und *Cursoren* (oder evtl. mit Version Oracle 8.1) lösen, ebenso die zweite oben beschriebene Aufgabe.

Hat man eine Tabelle *All_Languages*, die alle Sprachen enthält, so ist immerhin die folgende korrelierte Subquery möglich:

```
SELECT Country, Name
FROM NLanguage, All_Languages
WHERE Name IN
  (SELECT Name
   FROM TABLE (Languages));
```

Fazit: Man sollte den Wertebereich von geschachtelten Tabellen in *einer* Tabelle zugreifbar haben.

Die so definierten Objekttypen kann man sich mit `SELECT * FROM USER_TYPES` ausgeben lassen:

Type_name	**Type_oid**	**Typecode**	**Attributes**	**Methods**	**Pre**	**Inc**
GeoCoord	_	Object	2	0	NO	NO
Language_T	_	Object	2	0	NO	NO
Languages_List	_	Collection	0	0	NO	NO

Typen können mit `DROP TYPE` gelöscht werden. Hat man *abhängige Typen*, d.h. die Definition von Type B nutzt die Definition von Type A, so muss man diese Abhängigkeit übergehen:

In dem obigen Szenario führt `DROP TYPE Language_T` zu der Fehlermeldung *"Typ mit abhängigen Typen oder Tabellen kann nicht gelöscht oder ersetzt werden"* – da der Typ `Languages_List` davon abhängig ist. `DROP TYPE Language_T FORCE` führt dazu, dass der Typ `Language_T` gelöscht wird, allerdings leidet darunter der abhängige Typ `Languages_List` auch etwas:

```
SQL> desc Languages_List;
FEHLER:
ORA-24372: Ungültiges Objekt für Beschreibung
```

6 TRANSAKTIONEN IN Oracle

Beginn einer Transaktion

```
SET TRANSACTION READ [ONLY | WRITE];
```

Sicherungspunkte setzen

Für eine längere Transaktion können zwischendurch Sicherungspunkte gesetzt werden: Falls die Transaktion in einem späteren Stadium scheitert, kann man auf den Sicherungspunkt zurücksetzen, und von dort aus versuchen, die Transaktion anders zu einem glücklichen Ende zu bringen:

```
SAVEPOINT <savepoint>;
```

Ende einer Transaktion

- `COMMIT`-Anweisung, macht alle Änderungen persistent,
- `ROLLBACK [TO <savepoint>]` nimmt alle Änderungen [bis zu `<savepoint>]` zurück,
- DDL-Anweisung (z.B. `CREATE, DROP, RENAME, ALTER`),
- Benutzer meldet sich von Oracle ab,
- Abbruch eines Benutzerprozesses.

7 ÄNDERN DES DATENBANKSCHEMAS

Mit Hilfe der `ALTER`-Anweisung kann (unter anderem) das Datenbankschema verändert werden. Für alle Datenbankobjekte, die mit einem `CREATE`-Befehl erzeugt werden, gibt es den analogen `DROP`-Befehl um sie zu löschen, und den entsprechenden `ALTER`-Befehl zum Verändern des Objektes.

Mit `ALTER TABLE` können Spalten und Bedingungen hinzugefügt werden, bestehende Spaltendeklarationen verändert werden, und Bedingungen gelöscht, zeitweise außer Kraft gesetzt und wieder aktiviert werden:

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ⋮
  DROP <drop-clause>
  DISABLE <disable-clause>
  ⋮
  DISABLE <disable-clause>
  ENABLE <enable-clause>
  ⋮
  ENABLE <enable-clause>;
```

Die einzelnen Kommandos werden im folgenden behandelt:

Mit `ADD` können zu einer Tabelle Spalten und Tabellenbedingungen hinzugefügt werden:

```
ALTER TABLE <table>
  ADD (<col> <datatype> [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
       ⋮
       <col> <datatype> [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
       <tableConstraint>,
       ⋮
       <tableConstraint>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ... ;
```

Beispiel: Die Relation *economy* wird um eine Spalte *unemployment* erweitert. Außerdem wird zugesichert, dass die Summe der Anteile von Industrie, Dienstleistung und Landwirtschaft am Bruttosozialprodukt maximal 100% ist:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER,
       CHECK (Industry + Services + Agriculture <= 100));
```

Beim Einfügen einer neuen Spalte wird diese mit NULL-Werten gefüllt. Damit kann eine solche Spalte nicht als NOT NULL deklariert werden (das kann man aber später per ALTER TABLE ADD (CONSTRAINT ...) noch nachholen).

Mit ALTER TABLE ... MODIFY lassen sich Spaltendefinitionen verändern:

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<col> [<datatype>] [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
          ⋮
          <col> [<datatype>] [DEFAULT <value>] [<colConstraint> ... <colConstraint>])
  DROP <drop-clause>
  ... ;
```

Hier sind allerdings als <colConstraint> nur NULL und NOT NULL erlaubt. Alle anderen Bedingungen müssen mit ALTER TABLE ... ADD (<tableConstraint>) hinzugefügt werden.

Beispiele:

```
ALTER TABLE Country MODIFY (Capital NOT NULL);
ALTER TABLE encompasses ADD (PRIMARY KEY (Country,Continent));
ALTER TABLE Desert ADD (CONSTRAINT DesertArea CHECK (Area > 10));
```

Wird bei ADD oder MODIFY eine Bedingung formuliert, die der aktuelle Datenbankzustand nicht erfüllt, liefert ORACLE eine Fehlermeldung. Im obigen Beispiel wird die dritte Zeile akzeptiert, obwohl Null-Werte (die per Definition keine Bedingung verletzen können) auftreten.

Mit ALTER ... DROP/DISABLE/ENABLE können Bedingungen an eine Tabelle entfernt oder zeitweise außer Kraft gesetzt und wieder aktiviert werden:

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

Mit ENABLE/DISABLE ALL TRIGGERS werden alle zu einer Tabelle definierten Trigger (siehe Abschnitt 14.7) aktiviert/deaktiviert.

Eine PRIMARY KEY-Bedingung kann nicht gelöscht/disabled werden solange dieser Schlüssel durch einen Fremdschlüssel in einer REFERENCES-Deklaration referenziert wird. Wird CASCADE angegeben, werden eventuelle Fremdschlüssel-Bedingungen ebenfalls gelöscht/disabled. Beim enable'n muss man allerdings kaskadierend disable'te Constraints manuell einzeln wieder aktivieren werden.

In dem Abschnitt über referentielle Integrität wird gezeigt, dass die Definition von referentiellen Integritätsbedingungen das Verändern von Tupeln erheblich behindert. Deshalb müssen in solchen Fällen

vor dem Update die entsprechenden Integritätsbedingungen deaktiviert und nachher wieder aktiviert werden.

`ALTER SESSION` wurde bereits in Abschnitt 4 (Datumsangaben) verwendet, um die Parameter einer ORACLE-Sitzung zu verändern.

Teil II

Erweiterte Konzepte innerhalb SQL

8 REFERENTIELLE INTEGRITÄT

Referentielle Integritätsbedingungen entstehen aus dem Zusammenhang zwischen Primär- und Fremdschlüsseln. Eine referentielle Integritätsbedingung

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

definiert eine Inklusionsabhängigkeit: Zu jedem (Fremdschlüssel)wert der Attribute `(<attr-list>)` der *referenzierenden* Tabelle (*C- (Child-) Table*) `<table>` muss ein entsprechender Schlüsselwert von `(<attr-list>)` in der *referenzierten* Tabelle `<table>` (*P- (Parent-) Table*) existieren.

Dabei muss `(<attr-list'>)` ein Candidate Key der referenzierten Tabelle sein.

Referentielle Integritätsbedingungen treten immer auf, wenn bei der Umsetzung vom ER-Modell zum relationalen Modell Schlüsselattribute der beteiligten Entities in Beziehungstypen eingehen:

```
CREATE TABLE Country
 (Name  VARCHAR2(32),
  Code  VARCHAR2(4) PRIMARY KEY,
  ...);

CREATE TABLE Continent
 (Name     VARCHAR2(10) PRIMARY KEY,
  Area     NUMBER(2));

CREATE TABLE encompasses
 (Continent    VARCHAR2(10) REFERENCES Continent(Name),
  Country      VARCHAR2(4) REFERENCES Country(Code),
  Percentage   NUMBER);
```

Aufgabe **referentieller Aktionen** bzgl. einer C-Tabelle ist, bei Veränderungen am Inhalt der P-Tabelle Aktionen auf der C-Tabelle auszuführen, um die referentielle Integrität der Datenbasis zu erhalten. Ist dies nicht möglich, so werden die gewünschten `DELETE/UPDATE`-Operationen nicht ausgeführt, bzw. zurückgesetzt.

8.1 Referentielle Aktionen im SQL-2 Standard

Nach dem *SQL-2-Standard* werden referentielle Integritätsbedingungen werden bei `CREATE TABLE` und `ALTER TABLE` als `<columnConstraint>` (für einzelne Spalten)

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

oder <tableConstraint> (für mehrere Spalten) angegeben:

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

Die Klauseln ON DELETE und ON UPDATE geben an, welche referentiellen Aktionen bei einem DELETE bzw. UPDATE auf die referenzierte Tabelle ausgeführt werden sollen, um die referentielle Integrität der Datenbasis zu gewährleisten.

1. Ein INSERT bzgl. der referenzierten Tabelle oder ein DELETE bzgl. der referenzierenden Tabelle ist immer unkritisch.
   `INSERT INTO Country VALUES ('Lummerland','LU',...);`
   `DELETE FROM is_member ('D','EU');`
2. Ein INSERT oder UPDATE bzgl. der referenzierenden Tabelle, das einen Fremdschlüsselwert erzeugt, zu dem kein Schlüssel in der referenzierten Tabelle existiert, ist immer unzulässig:
   `INSERT INTO City VALUES ('Karl-Marx-Stadt','DDR',...);`
   anderenfalls ist es unkritisch:
   `UPDATE City SET Country='A' WHERE Name='Munich';`
3. Notwendig sind damit nur referentielle Aktionen für DELETE und UPDATE bzgl. der referenzierten Tabelle:
   `UPDATE Country SET Code='UK' WHERE Code='GB';` oder
   `DELETE FROM Country WHERE Code='I';`

NO ACTION:
Die DELETE/UPDATE-Operation auf der P-Tabelle wird zunächst ausgeführt; *Nach der Operation* wird überprüft, ob "dangling references" in der C-Tabelle entstanden sind. Falls ja, war die Operation verboten und wird zurückgenommen.

```
CREATE TABLE River
(Name VARCHAR2(20) CONSTRAINT RiverKey PRIMARY KEY,
 River VARCHAR2(20) REFERENCES River(Name),
 Lake VARCHAR2(20) REFERENCES Lake(Name),
 Sea VARCHAR2(25) REFERENCES Sea(Name),
 Length NUMBER);

DELETE FROM River;
```

Unter der Annahme, dass es keine weiteren Referenzen auf *River* gäbe (was in MONDIAL nicht der Fall ist, da *located River* referenziert), würden dabei alle Flüsse gelöscht. Referenzen durch Nebenflüsse, die ebenfalls gelöscht werden, sind dabei kein Hindernis (da der Datenbankzustand *nach* der kompletten Operation betrachtet wird).

RESTRICT:
Die DELETE/UPDATE-Operation auf der P-Tabelle wird nur dann ausgeführt, wenn keine "dangling references" in der C-Tabelle entstehen können:

```
DELETE FROM Organization;
```

wird abgebrochen, falls es eine Organisation gibt, die Mitglieder hat (Referenz von *is_member* auf *Organization*).

CASCADE:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird ausgeführt. Die referenzierenden Tupel der C-Tabelle werden ebenfalls mittels DELETE entfernt, bzw. mittels UPDATE geändert. Ist die C-Tabelle selbst P-Tabelle bzgl. einer anderen Bedingung, so wird das DELETE/UPDATE bzgl. der dort festgelegten Lösch/Änderungs-Regel weiter behandelt:

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

wird am sinnvollsten dadurch ausgeführt, dass die Ersetzung für referenzierende Tupel ebenfalls durchgeführt wird:

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

SET DEFAULT:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird ausgeführt und bei den referenzierenden Tupeln der C-Tabelle wird der entsprechende Fremdschlüsselwert auf die für die betroffenen Spalten festgelegten DEFAULT-Werte gesetzt. Dafür muss dann wiederum ein entsprechendes Tupel in der referenzierten Relation existieren. Beispiel: Eine Firmendatenbank, in denen jedes Projekt einem Mitarbeiter zugeordnet ist. Fällt dieser aus, werden seine Projekte zur Chefsache gemacht.

SET NULL:

Die DELETE/UPDATE-Operation auf der P-Tabelle wird ausgeführt. In der C-Tabelle wird der entsprechende Fremdschlüsselwert durch NULL ersetzt. Voraussetzung ist hier, dass NULLs zulässig sind.

Beispiel: Die Relation *located* gibt an, an welchem Fluss/See/Meer eine Stadt liegt, z.B.

located(Bremerhaven,Nds.,D,Weser,NULL,Nordsee)

Wird nun mit `DELETE * FROM River WHERE Name='Weser';`
das Tupel 'Weser' in der Relation River gelöscht, sollte die Information, dass Bremerhaven an der Nordsee liegt, erhalten bleiben:

located(Bremerhaven,Nds.,D,NULL,NULL,Nordsee)

8.2 Referentielle Aktionen in Oracle

In Oracle 9 sind nur ON DELETE/UPDATE NO ACTION, ON DELETE CASCADE und ON DELETE SET NULL implementiert :-(. Als Default wird NO ACTION angenommen; damit ist nur optional anzugeben, falls ON DELETE CASCADE oder ON DELETE SET NULL verwendet werden soll:

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

für <columnConstraint> bzw.

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

für <tableConstraint> (für mehrere Spalten).

Insbesondere die Tatsache, dass ON UPDATE CASCADE fehlt, ist beim Durchführen von Updates ziemlich lästig:

Beispiel 9 (Umbenennung eines Landes) Für die Tabelle *Country* ist das Kürzel *Code* als PRIMARY KEY definiert. *Code* wird u.a. in *Province* referenziert, d.h. ist dort *Fremdschlüssel*:

```
CREATE TABLE Country
  ( Name  VARCHAR2(32) NOT NULL UNIQUE,
    Code  VARCHAR2(4) CONSTRAINT CountryKey PRIMARY KEY);

CREATE TABLE Province
  ( Name    VARCHAR2(32)
    Country VARCHAR2(4) CONSTRAINT ProvRefsCountry
                    REFERENCES Country(Code));
```

Die beiden Tabellen enthalten unter anderem die Tupel ('United Kingdom','GB') und ('Yorkshire','GB'). Nun soll das Landeskürzel von 'GB' nach 'UK' geändert werden.

- `UPDATE Country SET Code='UK' WHERE Code='GB'` führt zu einer "dangling reference" des Tupels ('Yorkshire','GB').
- will man zuerst `UPDATE Province SET Code='UK' WHERE Code='GB'` ändern, gibt es kein zu referenzierendes Tupel für ('Yorkshire','UK').

Damit muss man zuerst die referentielle Integritätsbedingung außer Kraft setzen, dann die Updates vornehmen, und danach die referentielle Integritätsbedingung wieder aktivieren:

```
ALTER TABLE Province DISABLE CONSTRAINT ProvRefsCountry;
UPDATE Country SET Code='UK' WHERE Code='GB';
UPDATE Province SET Country='UK' WHERE Country='GB';
ALTER TABLE Province ENABLE CONSTRAINT ProvRefsCountry;
```

□

Man kann ein Constraint auch bei der Tabellendefinition mitdefinieren, und sofort disablen:

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

In dem oben beschriebenen Fall stellt "nur" das Ändern von Daten ein Problem dar – das mit `ON UPDATE CASCADE` gelöst werden könnte. Ein analoges Problem ergibt sich aus gegenseitigen und zyklischen Referenzen zwischen verschiedenen Tabellen:

```
CREATE TABLE Country
   ( Name VARCHAR2(32),
```

```
      Code VARCHAR2(4) PRIMARY KEY,
      Capital VARCHAR2(35),
      Province VARCHAR2(32),
      ⋮
      CONSTRAINT CountryCapRefsCity
         FOREIGN KEY (Capital,Code,Province)
         REFERENCES City(Name,Country,Province));

CREATE TABLE Province
   ( Name VARCHAR2(32),
     Country VARCHAR2(4),
     Capital VARCHAR2(35),
     ⋮
     PRIMARY KEY (Name, Country),
     CONSTRAINT ProvRefsCountry
        FOREIGN KEY (Country)
        REFERENCES Country(Name)),
     CONSTRAINT ProvCapRefsCity
        FOREIGN KEY (Capital,Country,Name)
        REFERENCES City(Name,Country,Prov));

CREATE TABLE City
   ( Name VARCHAR2(35),
     Country VARCHAR2(4),
     Province VARCHAR2(32),
     ⋮
     PRIMARY KEY (Name, Country, Province),
     CONSTRAINT CityRefsProv
        FOREIGN KEY (Country, Province)
        REFERENCES Province(Country, Name));
```

Ein Einfügen von Daten in diese Tabellen ist so nicht möglich, da in der jeweils anderen Tabelle noch nichts existiert. In diesem Fall wird man z.B. `CityRefsProv` bei der Tabellendefinition disablen, dann die Relationen *City*, *Province* und *Country* "von unten her" füllen, und danach das Constraint durch

```
ALTER TABLE Country
  ENABLE CONSTRAINT CityRefsProv;
```

aktivieren.

Wie bereits in Abschnitt 2.3 gesagt, können Tabellen, auf die noch eine referentielle Integritätsbedingung zeigt, mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden. Mit

```
DROP TABLE <table> CASCADE CONSTRAINTS;
```

wird eine Tabelle mit allen auf sie zeigenden referentielle Integritätsbedingungen gelöscht.

9 VIEWS – TEIL 2

9.1 View Updates

Views werden häufig (in Kombination mit der Vergabe von Zugriffsrechten, siehe Abschnitt10) dazu benutzt, den realen Datenbestand für Benutzer in einer veränderten Form darzustellen. Damit ein Benutzer in seiner Sicht(weise) Updates ausführen kann, müssen diese auf die Basisrelationen abgebildet werden. ORACLE verwendet Heuristiken, um aufgrund des Schemas festzustellen, ob eine solche Abbildung eindeutig möglich ist.

Über die Tabelle `USER_UPDATABLE_COLUMNS` des Data Dictionary kann der Benutzer abfragen, welche (View-)Spalten updatable sind (in dieser Abfrage ist es wichtig, den Tabellennamen in Großbuchstaben anzugeben!). In dem folgenden View können alle Spalten außer *Density* verändert werden. Da *Density* ein abgeleiteter Wert ist, kann diese Spalte natürlich nicht direkt verändert werden.

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

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

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

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

Name	Code	Area	population	Density
Lummerland	LU	1	4	4

Da das View bei der Ausgabe aus der (durch das `INSERT` veränderten) aktuellen Basistabelle *Country* neu berechnet wird, enthält es auch den Wert für *Density*.

Werte, die durch Aggregatfunktionen berechnet wurden, sind in Views ebenfalls nicht veränderbar. Diese Fälle sind relativ einfach damit zu begründen sind, dass berechnete Werte nicht geändert werden können.

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

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

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

Über diese Sicht können also Städte(namen) verändert werden. das Einfügen von Tupeln ist nicht möglich, da das Attribut *Country* nicht verändert/inserted werden darf, andererseits *Country* aber als Teil des Schlüssels einer neu eingefügten Stadt angegeben werden müsste.

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

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

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

Aber entgegen der oben angegebenen Werte können Daten in *CityCountry* gelöscht werden:

```
SQL> delete from CityCountry where country='Austria';
9 Zeilen wurden gelöscht.
```

Dieser Befehl löscht die betroffenen Städte aus *City*, löscht jedoch *Austria* nicht aus *Country*.

Ein spezielles Problem stellen Join-Views dar, bei denen mehrere Basistabellen verknüpft werden. Generell erlaubt ORACLE 8 nicht, dass ein View Update *mehrere* Basistabellen gleichzeitig verändert. Außerdem kommt es auch häufig vor, dass Werte zwar unverändert aus Basistabellen übernommen werden, und es trotzdem nicht möglich ist, eine eindeutige Abbildung der Änderungen auf die Basistabelle zu garantieren. Die ORACLE-Heuristiken basieren *nur* auf Schema-Informationen, betrachten also nicht, ob in der *gegebenen Datenbankinstanz* eine eindeutige Umsetzung möglich ist. Dabei spielen Schlüsseleigenschaften eine wichtige Rolle:

- Ist der Schlüssel einer Basistabelle auch gleichzeitig Schlüssel des Views, ist eine eindeutige Umsetzung möglich.
- umfasst der Schlüssel einer Basistabelle einen Schlüssel des Views, ist eine Umsetzung möglich (wobei eine Veränderung/Löschung an einem Tupel des Views möglicherweise mehrere Tupel der Basistabelle beeinflusst):

  ```
  CREATE OR REPLACE VIEW temp AS
  SELECT country, name, population
  ```

```
FROM Province A
WHERE population = (SELECT MAX(population)
                    FROM Province B
                    WHERE A.Country = B.Country);
SELECT * FROM temp WHERE Country = 'D';
```

Country	Name	Population
D	Nordrhein-Westfalen	17816079

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

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

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

Country	Name	Population
D	Bayern	11921944

- Ein Einfügen in das folgende View ist nicht möglich, da der Schlüssel der Basisrelation (*Province.Name* und *Province,Country* nicht komplett in den Attributen des Views enthalten ist.

```
CREATE OR REPLACE VIEW CountryProvPop AS
SELECT country, population
FROM Province A;
```

- Umfasst der Schlüssel einer Basistabelle keinen Schlüssel des Views komplett, ist keine eindeutige Umsetzung mehr möglich (siehe Aufgaben).

Bei Join-Views hat man allgemein das Problem, das man meistens einen Equi-Join über Schlüsselattribute bildet, wobei nur eines der Attribute dann in dem View auftritt – das andere (das zwar mit dem anderen übereinstimmt, aber eben formal nicht dasselbe Attribut ist wird dann nicht auf die darunterliegende Basistabelle umgesetzt.

In dem obigen Beispiel wurde ein Tupel eines Views so modifiziert, dass es aus dem Wertebereich des Views hinausfiel. Da Views häufig verwendet werden, um den "Aktionsradius" eines Benutzers einzuschränken, ist dies in vielen Kontexten unerwünscht und kann durch `WITH CHECK OPTION` verhindert werden:

Beispiel 10 Ein Benutzer soll *nur* mit US-amerikanischen Städte arbeiten.

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

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

liefert die Fehlermeldung

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

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

9.2 Materialized Views; View Maintenance

Views werden bei jeder Anfrage neu berechnet. Dies hat den Vorteil, dass sie immer den aktuellen Datenbankzustand repräsentieren. Bei der Verwendung großer Views über teilweise nur selten veränderten Daten – wie sie im realen Leben häufig vorkommen – ist eine ständige komplette Neuberechnung jedoch ineffizient. Zu diesem Zweck können *Materialized Views* eingesetzt werden, die bei jeder Datenänderung automatisch aktualisiert werden (dies kann u.a. durch Trigger (vgl. Abschnitt 14 geschehen). Materialized Views werden im Praktikum nicht behandelt. Die Probleme, die sich aus der Aktualisierung ergeben werden unter dem Stichwort *View Maintenance* zusammengefasst.

Sowohl *View Updates* als auch *View Maintenance* sind aktuelle Forschungsthemen (Theorie und Implementierung).

10 ZUGRIFFSRECHTE

Jeder Benutzer weist sich gegenüber ORACLE durch seinen Benutzernamen und sein Passwort aus.[1] Dem Benutzernamen werden vom Datenbankadministrator (DBA) Zugriffsrechte erteilt. Der DBA hat somit eine Schlüsselfunktion in der Verwaltung der Datenbank. In der Praxis fallen u.a. folgende Tätigkeiten in seinen Aufgabenbereich: Einrichten der Datenbank, Verwaltung der Benutzerrechte, Performance-Tuning der Datenbank, Durchführung von Datensicherungsmaßnahmen und "Beaufsichtigung" der Benutzer (Auditing).

Bei der Vergabe von Zugriffsrechten an die Benutzer wird meist nach der Regel verfahren, das jeder Benutzer so wenige Rechte wie möglich bekommt - damit er möglichst wenig Schaden in der Datenbank anrichten kann - und so viele Rechte wie nötig um seine Anwendungen durchführen zu können.

Schemakonzept. Die Organisation von Tabellen in ORACLE basiert auf dem *Schemakonzept*: Jedem Benutzer ist sein *Database Schema* zugeordnet, in dem er sich defaultmäßig aufhält und in dem er Objekte erzeugt und Anfragen an sie stellt. Die Bezeichnung der Tabellen geschieht somit global durch `<username>.<table>`, wird bei einer Anfrage durch einen Benutzer das Schema nicht angegeben, wird automatisch das entsprechende Objekt aus dem eigenen Schema angesprochen.

Die Zugriffsrechte teilen sich in zwei Gruppen: "Systemprivilegien", die z.B. zu Schemaoperationen berechtigen, sowie "Objekt"rechte, d.h., was man dann mit den einzelnen Objekten machen kann.

Systemprivilegien. In ORACLE existieren über 80 verschiedene Systemberechtigungen. Jede Systemberechtigung erlaubt einem Benutzer die Ausführung einer bestimmten Datenbankoperation oder Klasse von Datenbankoperationen. Auch bestimmte Schema-Objekte, wie z.B. Cluster, Indices und Trigger werden nur durch Systemberechtigungen gesteuert. Da Systemberechtigungen oftmals weitreichende Konsequenzen haben, darf sie in der Regel nur der DBA vergeben.

- `CREATE [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte erzeugen,
- `ALTER [ANY] TABLE/TYPE/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte verändern,
- `DROP [ANY] TABLE/TYPE/VIEW/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte löschen.
- `SELECT/INSERT/UPDATE/DELETE [ANY] TABLE`: Benutzer darf in Tabellen Tupel lesen/erzeugen/verändern/entfernen.

`ANY` berechtigt dazu die entsprechende Operation in *jedem* Schema auszuführen, fehlt `ANY`, so bleibt die Ausführung auf das eigene Schema beschränkt. Die Teilnehmer des Praktikums sollten etwa mit den Privilegien `CREATE SESSION`, `ALTER SESSION`, `CREATE TABLE`, `CREATE TYPE`, `CREATE CLUSTER`, `CREATE SYNONYM`, `CREATE VIEW` ausgestattet sein. Dabei sind die Zugriffe und Veränderungen an den eigenen Tabellen nicht explizit aufgeführt (würde logischerweise `SELECT TABLE` heißen).

[1] Im Rahmen des Praktikums wird durch den Aufruf von `sqlplus /` der UNIX-Account zur Autorisierung verwendet.

Will man auf eine Tabelle zugreifen, deren Zugriff man nicht gestattet ist, etwa `SELECT * FROM Kohl.Private`, bekommt man nur die Meldung `ORA-00942: Tabelle oder View nicht vorhanden`.

Systemprivilegien werden mit Hilfe der `GRANT`-Anweisung erteilt:

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

`<privilege-list>` ist dabei eine Aufzählung der administrativen Privilegien, die man jemandem erteilen will. Mit der `<user-list>` gibt man an, wem diese Privilegien erteilt werden; mit `PUBLIC` erhält jeder das Recht. Wird ein Systemprivileg zusätzlich mit `ADMIN OPTION` vergeben, so kann der Empfänger es auch weiter vergeben.

Informationen bzgl. der Zugriffsrechte werden in Tabellen des Data Dictionary gespeichert.[2] Mit `SELECT * FROM SESSION_PRIVS` erfährt man, welche generellen Rechte man besitzt.

Objektprivilegien. Jedes Datenbankobjekt hat einen Eigentümer, dies ist meistens derjenige der es angelegt hat (wozu er allerdings auch wieder das Recht haben muss). Prinzipiell darf niemand sonst mit einem solchen Objekt arbeiten, außer der Eigentümer oder der DBA erteilt ihm explizit entsprechende Rechte:

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

Hierbei wird das betreffende Objekt durch `<object>` beschrieben und kann von der Art `TABLE`, `VIEW`, `PROCEDURE/FUNCTION/PACKAGE` oder `TYPE` sein. `<privilege-list>` ist eine Aufzählung der Privilegien, die man erteilen will. Mögliche Privilegien sind `CREATE`, `ALTER` und `DROP` für die verschiedenen Arten von Schemaobjekten, `DELETE`, `INSERT`, `SELECT` und `UPDATE` für Tabellen[3] und Views, außerdem `INDEX`, und `REFERENCES` für Tabellen. Für `INSERT`, `REFERENCES` und `UPDATE` kann optional angegeben werden, für welche Spalten das Recht vergeben wird (bei `DELETE` ist das natürlich nicht erlaubt und resultiert auch in einer Fehlermeldung[4]. Für Prozeduren/Funktionen/Packages und Typen kann man `EXECUTE` vergeben. Mit `ALL` gibt man alle Privilegien die man an dem beschriebenen Objekt hat, weiter.

- `SELECT`, `UPDATE`, `INSERT`, `DELETE` klar, die entsprechenden Zugriffs- und Änderungsoperationen,
- `REFERENCES`: das Recht, Spalten einer Tabelle als Fremdschlüssel in einer `CREATE TABLE`-Anweisung zu deklarieren und referentielle Aktionen anzugeben,
- `INDEX` das Recht, Indexe auf dieser Tabelle zu erstellen.
- `EXECUTE` das Recht, eine Prozedur, Funktion oder einen Trigger auszuführen, oder einen Typ zu benutzen.

Mit der `<user-list>` gibt man an, wem diese Privilegien erteilt werden; mit `PUBLIC` erhält jeder das Recht. Wird ein Recht mit `GRANT OPTION` vergeben, kann derjenige, der es erhält, es auch weiter vergeben.

Rechte entziehen. Gelegentlich ist es erforderlich, jemandem ein Recht (oder nur die `GRANT OPTION`) zu entziehen; dies kann man natürlich nur machen, wenn man dieses Recht selbst verge-

[2] Zur Erinnerung: alle Tabellen, die im Data Dictionary enthalten sind, können mit `SELECT * FROM DICTIONARY` erfragt werden.

[3] dabei kann man mit `UPDATE`-Recht weder löschen noch einfügen.

[4] für ORACLE 8.0.3 getestet.

ben hat:

Administrative Rechte:

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

Objekt-Rechte:

```
REVOKE <privileg-list> | ALL
ON TABLE <table>
FROM <user-list> | PUBLIC [CASCADE CONSTRAINTS];
```

`CASCADE CONSTRAINTS` sorgt dafür, dass alle referentiellen Integritätsbedingungen, die auf einem entzogenen `REFERENCES`-Privileg beruhen, wegfallen.

- Hat ein Benutzer eine Berechtigung von mehreren Benutzern erhalten, behält er sie, bis sie ihm jeder einzelne entzogen hat.
- Hat ein Benutzer eine Berechtigung mit `ADMIN OPTION` oder `GRANT OPTION` weitergegeben und bekommt sie entzogen, so wird sie automatisch auch allen Benutzern entzogen, an die er sie weitergegeben hat.

Mit

```
SELECT * FROM USER_TAB_PRIVS;
```

kann man sich die Informationen über Tabellen ausgegeben lassen, die einem gehören und auf die man irgendwelche Rechte vergeben oder die jemandem anderem gehören und man Rechte dafür bekommen hat. Hat man Rechte nur für bestimmte Spalten einer Tabelle vergeben oder erhalten, erfährt man dies mit

```
SELECT * FROM USER_COL_PRIVS;
```

Die Tabellen `USER_TAB/COL_PRIVS_MADE` und `USER_TAB/COL_PRIVS_RECD` enthalten im einzelnen die vergebenen oder erhaltenen Rechte für Tabellen bzw. Spalten.

Rollenkonzept. Neben der direkten Vergabe von Privilegien an einzelne Beutzer kann die Vergabe von Privilegien in Oracle auch durch *Rollen* geregelt werden. Benutzer die über die gleichen Rechte verfügen sollen werden dann einer bestimmten Rolle zugeordnet. Das Rollenkonzept vereinfacht somit auch die Rechtevergabe für den DBA, da die Rechte nur einmal der Rolle zugeordnet werden, anstatt dies für jeden einzelnen Benutzer zu wiederholen. Rollen können hierarchisch strukturiert sein. Dabei muss darauf geachtet werden, dass keine Zyklen bei der Zuweisung entstehen. Die Zuordnung von Benutzern zu einer Rolle erhält man (falls man die DBA-Tabellen lesen darf) über folgende Anfrage

```
SELECT * FROM DBA_ROLE_PRIVS;
```

11 SYNONYME

Synonyme können verwendet werden, um ein Schemaobjekt unter einem anderen Namen als ursprünglich abgespeichert anzusprechen. Dies ist insbesondere sinnvoll, wenn man eine Relation, die in einem anderen Schema liegt, nur unter ihrem Namen ansprechen will.

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

Ohne den Zusatz `PUBLIC` ist das Synonym nur für den Benutzer definiert, der es definiert hat. Mit `PUBLIC` ist das Synonym systemweit verwendbar. Das dazu notwendige `CREATE ANY SYNONYM`-Privileg haben i.a. nur die SysAdmins.

Beispiel: Wenn ein Benutzer keine eigene Relation "City" besitzt, sondern stattdessen immer die Relation "City", aus dem Schema "dbis" verwendet, müsste er diese immer mit `SELECT * FROM dbis.City` ansprechen. Definiert man

```
CREATE SYNONYM City
  FOR dbis.City;
```

so kann man sie auch über `SELECT * FROM City` erreichen.

Man kann übrigens ein Synonym auch für eine Tabelle definieren, die (noch) nicht existiert und/oder auf die man noch kein Zugriffsrecht hat. Wird die Tabelle dann definiert und erhält man das Zugriffsrecht, tritt das Synonym in Kraft.

Synonyme werden mit `DROP SYNONYM <synonym>` gelöscht.

12 ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS

Bei der Aufstellung der Objektprivilegien im vorhergehenden Abschnitt lässt sich feststellen, dass bei `GRANT SELECT` der Zugriff nicht auf Spalten eingeschränkt werden kann. Dies kann allerdings einfach über Views (vgl. Abschnitt 2.2) geschehen:

Der *nicht existierende* Befehl

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

kann ersetzt werden durch

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

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

Beispiel: Der Benutzer `pol` ist Besitzer der Relation "Country". Er entscheidet sich, seinem Kollegen `geo` die Daten über die Hauptstadt und ihre Lage nicht zur Verfügung zu stellen. Alle restlichen Daten sollen von `geo` aber gelesen *und* geschrieben werden können.

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

GRANT SELECT ON pubCountry TO geo;
```

Da das View über Basis *genau einer* Basisrelation definiert ist und der von Basisrelation und View übereinstimmt, sind Updates und Einfügungen erlaubt.

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

Zusätzlich wird `geo` sich ein Synonym definieren:

```
CREATE SYNONYM Country FOR pol.pubCountry;
```

Ärgerlich ist in diesem Zusammenhang nur, dass man keine Referenzen auf Views legen kann (diese haben keinen Primary Key): Um Country.Code zu referenzieren, muss `geo` trotzdem die *Basistabelle* `pol.pubCountry` verwenden:

```
pol : GRANT REFERENCES Code ON Country TO geo;
```

```
geo : ... REFERENCES pol.Country(Code);
```

13 OPTIMIERUNG DER DATENBANK

Dieser Abschnitt behandelt die in ORACLE vorhandenen Konstrukte, um Speicherung und Zugriff einer Datenbank zu optimieren. Diese Konstrukte dienen dazu, (a) effizienter bestimmte Einträge zu finden (Indexe) und (b) physikalisch schneller zugreifen zu können (Cluster). (a) ist dabei ein algorithmisches Problem, betreffend die Suche nach Werten; (b) soll die Zugriffe auf den Hintergrundspeicher minimieren.

Diese Strukturen haben keine Auswirkung auf die Formulierung der restlichen SQL-Anweisungen!

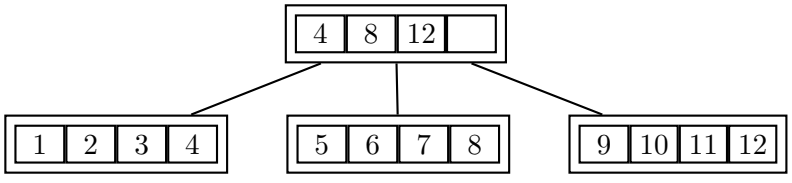
13.1 Indexe

Indexe unterstützen den Zugriff auf eine Tabelle über einen gegebenen Spaltenwert:

```
CREATE TABLE PLZ
  (City    VARCHAR2(35)
   Country VARCHAR2(4)
   ProvinceVARCHAR2(32)
   PLZ     NUMBER)
```

- Indexe in ORACLE als B-Baum organisiert,
- logisch und physikalisch unabhängig von den Daten der zugrundeliegenden Tabelle,
- keine Auswirkung auf die Formulierung einer SQL-Anweisung,
- mehrere Indexe für eine Tabelle möglich,
- bei sehr vielen Indexen auf einer Tabelle kann es beim Einfügen, Ändern und Löschen von Sätzen zu Performance-Verlusten kommen.

B-Baum:



- Logarithmische Höhe des Baumes,
- B-Baum: Knoten enthalten auch die Werte der Tupel,
- B*-Baum: Knoten enthalten *nur* die Weg-Information, daher ist der Verzweigungsgrad sehr hoch, und die Höhe des Baumes auch absolut gesehen gering.
- Schneller Zugriff (logarithmisch) versus hoher Reorganisationsaufwand (→ Algorithmentechnik-Vorlesung).

Indexarten:

- **eindeutige Indexe**
  - keine gleichen Werte in der Spalte des Indexe erlaubt,
  - bei einer `PRIMARY KEY`-Deklaration erzeugt ORACLE automatisch einen Index über diese Spalte(n).
- **mehrdeutige Indexe**
  - gleiche Werte sind erlaubt.
- **zusammengesetzte Indexe**
  - Index besteht aus mehreren Spalten (Attributen) einer Tabelle
  - sinnvoll, wenn häufig `WHERE`-Klauseln über mehrere Spalten formuliert werden (z.B. (L_ID,LT_ID)).

```
CREATE [UNIQUE] INDEX <name> ON <table>(<column-list>);
```

Im obigen Beispiel wäre ein Index über L_ID, PLZ sinnvoll:

```
CREATE INDEX PLZIndex ON PLZ (Country,PLZ);
SELECT *
  FROM plz
  WHERE plz = 79110 AND Country = 'D';
```

Indexe werden mit `DROP INDEX` gelöscht.

Wichtig: Indexe sind physikalische Strukturen, die in der Datenbank gespeichert sind, während Schlüssel ein rein logisches Konzept sind. Damit gehören Indexe *nicht* zur Modellierung (ER- und relationale Modellierung), sondern werden erst betrachtet, wenn das Datenbankschema entworfen wird.

13.2 Bitmap-Indexe

Die Motivation der oben genannten Indexe (insbesondere als `UNIQUE INDEX`) liegt in erster Linie darin, den Zugriff zu einzelnen Elementen zu verbessern indem nicht mehr die gesamte Relation linear durchsucht werden muss. Erst auf den zweiten Blick wird erkennbar, dass damit auch alle Werte mit demselben Indexwert effizient zugegriffen werden können.

Bei `BITMAP`-Indexen liegt das Hauptinteresse darauf, zu bestimmten Attributen, die nur wenige verschiedene Wert annehmen, *alle* Tupel mit einem bestimmten Attributwert optimal selektieren zu können: Für die entsprechenden Attribute wird eine Bitmap angelegt, die für jeden vorkommenden Wert ein Bit enthält. Dieses wird für ein Tupel genau dann gesetzt, wenn das Tupel den entsprechenden Attributwert enthält:

```
CREATE BITMAP INDEX <name> ON <table>(<column-list>);
```

Beispiel 11 (Bitmap-Index) Die Relation *encompasses* kann vorteilhaft mit einem Bitmap-Index über Kontinent erweitert werden:

```
CREATE BITMAP INDEX continents_index ON encompasses(continent);
```

Country	**Perc.**
D	100
CH	100
USA	100
R	20
R	80
ZA	100
NZ	100

EUR	AS	AM	AF	AUS
×				
×				
		×		
×				
	×			
			×	
				×

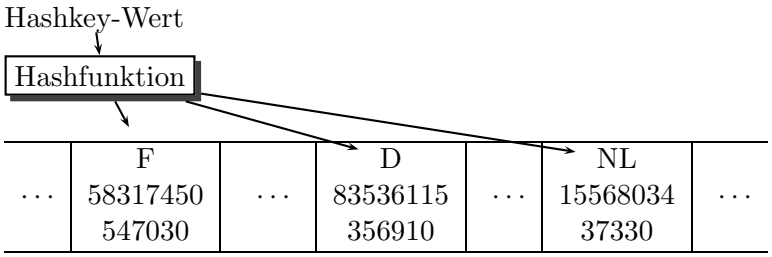
□

Bitmap-Indexe haben insbesondere dann Performanz-Vorteile, wenn eine `WHERE`-Klausel mehrere solche Spalten auswertet und damit auf eine Verknüpfung auf logischer (Bit)-Ebene auf diesen Indexen reduziert werden kann. Bitmap-Indexe sind damit maßgeschneidert für Data-Warehousing-Anwendungen.

13.3 Hashing

Hashing unterstützt Zugriff auf eine Tabelle über bestimmte Attributwerte (eben die, die als *Hashkey* definiert sind). Dabei wird aufgrund dieser Attributwerte *in konstanter Zeit* berechnet, wo die entsprechenden Tupel zu finden sind: Eine Hash-Funktion bildet jeden Wert des angegebenen Hashkeys auf eine Zahl innerhalb eines festgelegten Hash-Bereiches ab. Die Vorgehensweise ist damit ähnlich zu einem Index.

Z.B. wäre ein Hash-Zugriff sinnvoll, wenn um auf die Daten über ein bestimmtes Land gezielt zugreifen zu können: Hashkey ist *Country.Code*.



Vorteile der Hash-Methode:

- falls Anzahl der Hash-Werte ausreichend groß, genügt ein Blockzugriff, um einen Datensatz zu finden.
- kein Zugriff auf Indexblöcke erforderlich.

In Oracle 8 ist Hashing nur für Cluster implementiert.

13.4 Cluster

Cluster unterstützen Zugriff auf mehrere Tabellen mit gemeinsamen Spalten durch physikalisches Zusammenfassen der Tabellen nach diesen Spalten, um jeweils bei einem Hintergrundspeicherzugriff semantisch zusammengehörende Daten in den Hauptspeicher zu bringen.

- Zusammenfassung einer Gruppe von Tabellen, die alle eine oder mehrere gemeinsame Spalten (Clusterschlüssel) besitzen,

- sinnvoll bei referentiellen Integritätsbedingungen (Fremdschlüsselbeziehungen) oder bei häufigen `JOIN`-Anweisungen über dieselbe Spalte(n),
- auch zu einer einzelnen Tabelle kann ein Cluster erzeugt werden, z.B. um Städte nach Landesteilen geordnet abzuspeichern.

Vorteile eines Clusters:

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

Nachteile:

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

Wird ein Cluster erstellt, muss zuerst dessen Cluster-Schlüssel angegeben werden, um die in diesem Cluster gespeicherten Tabellen geeignet zu organisieren.

Erzeugen eines Clusters:

1. Cluster und Clusterschlüssel definieren,
2. Tabellen erzeugen und mit der Angabe der Spalten für den Clusterschlüssel in den Cluster einfügen,
3. Clusterindex über den Clusterschlüssel-Spalte(n) definieren. Dies muss *vor* dem ersten DML-Kommando geschehen.

In Mondial gibt es einige Relationen (*Mountain*, *Lake*, ...), in denen zusammengehörende Daten über auf zwei Relationen verteilt sind: Z.B. enthält *Sea* die topographischen Informationen über ein Meer, während *geo_Sea* dessen Lage in Bezug auf geopolitische Begriffe bezeichnet. Diese könnte man clustern. Ebenso ist es sinnvoll, die Relation *City* nach (*Country, Province*) zu clustern. Beide Cluster sind in Abb. 13.1 und 13.2 gezeigt.

Diese Speicherstruktur erreicht man folgendermaßen:

Zuerst wird ein Cluster erzeugt, und angegeben, welche Spalten (und ihre Datentypen) den Clusterschlüssel bilden sollen:

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

Als Default wird ein *indexed Cluster* erstellt, d.h. die Zeilen werden entsprechend ihren Clusterschlüsselwerten indiziert und geclustert.

Gibt man `HASH` und eine Hashfunktion an wird aus den Zeilen eines Tupels dessen Hashwert (mit `<funktion>` falls angegeben, sonst intern) berechnet, der dann modulo `<integer>` genommen wird. Nach diesem Wert wird dann geclustert (wobei vollkommen unzusammenhängende Zeilen zusammen geclustert werden können).

Die Tabellen werden durch ein weiteren Befehl in der `CREATE TABLE`-Definition einem Cluster zugeordnet. Dabei muss die Zuordnung der Spalten zum Clusterschlüssel angegeben werden:

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

Sea		
Mediterranean Sea	**Depth**	
	5121	
	Province	**Country**
	Catalonia	E
	Balearic Isl.	E
	Valencia	E
	Murcia	E
	Andalusia	E
	Corse	F
	Languedoc-R.	F
	Provence	F
	⋮	⋮
Baltic Sea	**Depth**	
	459	
	Province	**Country**
	Schleswig-H.	D
	Mecklenb.-Vorp.	D
	Szczecin	PL
	Koszalin	PL
	Gdansk	PL
	Olsztyn	PL
	⋮	⋮

Abbildung 13.1: Geclusterte Tabellen

```
 <col> <datatype>)
CLUSTER <cluster>(<column-list>);
```

Zum Schluss muss noch der Clusterschlüsselindex erzeugt werden:

```
CREATE INDEX <name> ON CLUSTER <cluster>;
```

Mit den folgenden Befehlen kann man den oben beschriebenen Cluster *Cl_Sea* generieren:

```
CREATE CLUSTER Cl_Sea (Sea VARCHAR2(25));

CREATE TABLE CSea
  (Name    VARCHAR2(25) PRIMARY KEY,
   Depth   NUMBER)
  CLUSTER Cl_Sea (Name);

CREATE TABLE Cgeo_Sea
  (ProvinceVARCHAR2(35),
   Country VARCHAR2(4),
   Sea     VARCHAR2(25))
  CLUSTER Cl_Sea (Sea);
```

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

Abbildung 13.2: Geclusterte Tabellen

```
CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```

Alle drei Methoden erfordern relativ hohen Aufwand, wenn sie reorganisiert werden müssen (Überlaufen von Indexknoten, Cluster-Bereichen, Hash-Speicherbereichen oder des Wertebereichs der Hash-Funktion).

Teil III

Prozedurale Konzepte in Oracle: PL/SQL

SQL bietet keinerlei prozedurale Konzepte wie z.B. Schleifen, Verzweigungen oder Variablendeklarationen. Entsprechend sind viele Aufgaben nur umständlich über Zwischentabellen oder evtl. überhaupt nicht in SQL zu realisieren. Anwendungsprogramme repräsentieren häufig zusätzliches anwendungsspezifisches Wissen, das nicht in der Datenbank enthalten ist.

Deshalb gibt es einige Erweiterungen und Umgebungen für SQL, die prozedurale Konzepte anbieten. Zum einen ist dies die Einbettung von SQL in prozedurale Hostsprachen (*embedded SQL*); z.B. Pascal, C, C++, oder neuerdings auch Java (JDBC), zum anderen eine spezielle Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, genannt *PL/SQL*. Eine Erweiterung hat hierbei den Vorteil, dass man diese prozeduralen Elemente enger an die Datenbank anbinden und z.B. in Prozeduren und Triggern nutzen kann.

In Oracle 7 bestand eine klare Trennung zwischen dem "eigentlichen", deklarativen SQL und der *prozeduralen Erweiterung* PL/SQL. Dennoch war PL/SQL bereits ein integrierter Teil des Oracle-Systems (im Gegensatz zu *embedded SQL* oder *JDBC*).

Mit der Erweiterung zu Objektorientierung und *Methoden* in Oracle 8 verschwimmt diese Trennung: bereits innerhalb des deklarativen Teils von SQL werden im Zuge von Methoden Konzepte von PL/SQL verwendet, um diese Methoden zu implementieren.

14 PROZEDURALE ERWEITERUNGEN: PL/SQL

Der Name PL/SQL steht für *Procedural language extensions to SQL*. Bis ORACLE 7 wurde PL/SQL – in erster Linie – dazu verwendet, *Prozeduren* und *Funktionen* zu schreiben. Diese werden wiederum entweder als Prozeduren/Funktionen vom Benutzer eingesetzt oder automatisch als *Trigger* als Reaktion auf bestimmte Ereignisse in der Datenbank aufgerufen. Mit ORACLE 8 werden auch die Methoden der Objekttypen in PL/SQL implementiert.

14.1 PL/SQL-Blöcke

Die allgemeine Struktur eines PL/SQL-*Blocks* ist in Abb. 14.1 dargestellt.

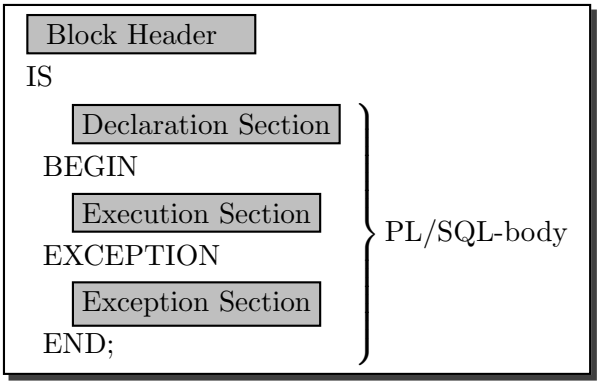


Abbildung 14.1: Struktur eines PL/SQL-Blocks

Der *Block Header* gibt die Art des zu definierenden Schemaobjekts (Funktion, Prozedur, Trigger oder *anonym* (innerhalb eines anderen Blocks)) sowie die Aufrufparameter an. Der *PL/SQL-Body* enthält die Definition des Prozedur- bzw. Funktionsrumpfes in PL/SQL. Dabei sind rekursive Prozeduren und Funktionen erlaubt. Im einzelnen enthält die *Declaration Section* die Deklarationen der in dem Block verwendeten Variablen, die *Execution Section* enthält die Befehlssequenz des Blocks, und in der *Exception Section* werden Reaktionen auf eventuell auftretende Fehlermeldungen angegeben.

Prozeduren. Für Prozeduren sieht die Deklaration folgendermaßen aus:

```
CREATE [OR REPLACE] PROCEDURE <proc_name>
  [(<parameter-list>)]
  IS <pl/sql-body>;
```

Wird `OR REPLACE` angegeben, so wird eine eventuell bereits existierende Prozedurdefinition überschrieben. Die Deklaration der formalen Parameter in `(<parameter-list>)` ist dabei von der Form

```
(<variable> [IN|OUT|IN OUT] <datatype>,
 ⋮
 <variable> [IN|OUT|IN OUT] <datatype>)
```

wobei `<variable>` Variablennamen mit den bei `<datatype>` angegebenen Datentypen sind, und `IN`, `OUT` und `IN OUT` angeben, wie die Prozedur/Funktion auf den Parameter zugreifen kann (Lesen, Schreiben, beides). Dies entspricht dem Begriff des Wert- bzw. Variablenparameters:

- `IN`: Beim Aufruf der Prozedur muss dieses Argument einen Wert besitzen. Falls keine Angabe erfolgt, wird als Default `IN` gesetzt.
- `OUT`: Dieses Argument wird durch die Prozedur gesetzt.
- `IN OUT`: Beim Aufruf der Prozedur muss dieses Argument einen Wert besitzen und die Prozedur gibt einen Wert an ihre Umgebung zurück.
- Bei `OUT` und `IN OUT` muss beim Aufruf eine Variable angegeben sein, bei `IN` ist auch eine Konstante erlaubt.

Als Parameter sind alle von PL/SQL unterstützten Datentypen erlaubt. Dabei werden diese *ohne* Längenangabe spezifiziert, also `VARCHAR2` anstelle `VARCHAR2(20)`.

Funktionen. Funktionen werden analog definiert, zusätzlich wird der Datentyp des Ergebnisses angegeben:

```
CREATE [OR REPLACE] FUNCTION <funct_name>
  [(<parameter-list>)]
  RETURN <datatype>
  IS <pl/sql body>;
```

PL/SQL-Funktionen werden mit `RETURN <ausdruck>` verlassen, wobei `<ausdruck>` ein Ausdruck über PL/SQL-Variablen und -Funktionen ist. Jede Funktion muss mindestens ein `RETURN`-Statement enthalten.

Beispiel 12 (Funktion) Die folgende Funktion berechnet die Distanz zwischen zwei Paaren (*Länge,Breite*) von geographischen Koordinaten (Funktionsaufruf siehe Beispiel 13):

```
CREATE OR REPLACE FUNCTION distance
       (l1 NUMBER, b1 NUMBER, l2 NUMBER, b2 NUMBER)
RETURN NUMBER
IS
BEGIN
 RETURN 6370*ACOS(cos(b1/180*3.14)*cos(b2/180*3.14)*cos((l1-l2)/180*3.14)
              + sin(b1/180*3.14)*sin(b2/180*3.14));
END;
/
```

Eine Funktion darf keine Seiteneffekte auf die Datenbasis haben (ansonsten erzeugt Oracle eine Fehlermeldung).

Anonyme Blöcke. Innerhalb der *Execution Section* können geschachtelte Blöcke auftreten. Dabei werden *anonyme Blöcke* verwendet. Da diese keinen Header besitzen, wird die *Declaration Section* mit `DECLARE` eingeleitet:

```
BEGIN      /* äußerer Block */
  - Befehle des äußeren Blocks -
  DECLARE    /* innerer Block */
     - Deklarationen des inneren Blocks
  BEGIN      /* innerer Block */
     - Befehle des inneren Blocks
  END;       /* innerer Block */
  - Befehle des äußeren Blocks -
END;       /* äußerer Block */
```

Trigger werden in Abschnitt 14.7 detailliert behandelt.

Deklaration ausführen. Wenn man eine Deklaration ausführt, muss nach dem Semikolon noch ein Vorwärtsslash ("/") folgen, um die Deklaration zu verarbeiten!

Falls sich SQL*Plus mit einem "... created with compilation errors" zurückmeldet, kann man sich die Fehlermeldung mit

```
SHOW ERRORS;
```

ausgeben lassen.

Prozeduren und Funktionen können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.

Prozeduren und Funktionen aufrufen. Prozeduren werden innerhalb von PL/SQL-Blöcken durch
`<procedure> (arg1,...,argn);`
aufgerufen. In SQLPlus werden Prozeduren mit
`execute <procedure> (arg1,...,argn);`
aufgerufen. Funktionsaufrufe haben die übliche Syntax
`... <function> (arg1,...,argn) ...`
wie in anderen Programmiersprachen.

Von SQLPlus können Funktionen nicht direkt, sondern nur aus `SELECT`-Statements aufgerufen werden. Da eine solche Pseudo-Anfrage genau einen Wert zurückgeben soll, muss eine entsprechende Relation in der `FROM`-Zeile angegeben werden. Oracle bietet für diesen Fall eine einspaltige Tabelle `DUAL`, die genau ein Tupel enthält:

```
SELECT <function-name>(<argument-list>)
FROM DUAL;
```

Beispiel 13 Die folgende Anfrage ergibt die Entfernung von Freiburg zum Nordpol, bzw. nach Stockholm (Definition der Funktion `distance` siehe Beispiel 12):

```
SELECT distance(7.8, 48, 0, 90)
FROM DUAL;

SELECT distance(7.8, 48, Longitude, Latitude)
FROM City
```

```
WHERE Name = 'Stockholm';
```

Zugriffsrechte. Anderen Benutzern kann man durch `GRANT EXECUTE ON <procedure/function> TO <user>` die Benutzung von Prozeduren und Funktionen erlauben. Dabei muss man berücksichtigen, dass Prozeduren und Funktionen jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt werden. Vergibt man also `GRANT EXECUTE ON <procedure/function> TO <user>`, so kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat. Andererseits hat man damit die Möglichkeit, Zugriffsberechtigungen implizit strenger zu formulieren als mit `GRANT ... ON <table> TO ...`: Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

Beispiel. Die Informationen über Länder sind in MONDIAL über mehrere Relationen verteilt. Die folgende Prozedur *InsertCountry* verteilt die eingegebenen Werte für Name, Code, Area, Population, GDP, Inflation und Population Growth auf die entsprechenden Relationen:

```
CREATE PROCEDURE InsertCountry
 (name VARCHAR2, code VARCHAR2, area NUMBER, pop NUMBER,
  gdp NUMBER, inflation NUMBER, pop_growth NUMBER)
IS
BEGIN
  INSERT INTO Country (Name,Code,Area,Population)
         VALUES (name,code,area,pop);
  INSERT INTO Economy (Country,GDP,Inflation)
         VALUES (code,gdp,inflation);
  INSERT INTO Population (Country,Population_Growth)
         VALUES (code,pop_growth);
END;
/

EXECUTE InsertCountry ('Lummerland', 'LU', 1, 4, 50, 0.5, 0);
```

Meistens ist eine Prozedur jedoch nicht so einfach wie in dem obigen Beispiel, wo nur einige SQL-Statements sequenziell zusammengefasst wurden. Zur Definition von komplizierteren Funktionen und Prozeduren werden wie in prozeduralen Programmiersprachen üblich Variablen und Programmkonstrukte verwendet.

14.2 PL/SQL-Deklarationen

In der *Declaration Section* werden die in dem entsprechenden Block verwendeten PL/SQL-Datentypen, PL/SQL-Variablen und *Cursors* deklariert.

PL/SQL-Variablen. PL/SQL-Variablen mit werden durch Angabe ihres Datentyps und einer optionalen Angabe eines Default-Wertes deklariert:

```
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
⋮
<variable> <datatype> [NOT NULL] [DEFAULT <value>];
```

Anstatt `<datatype>` direkt explizit anzugeben, kann man eine *anchored* Typdeklaration machen, indem man angibt, mit welcher Variablen der Typ übereinstimmen soll:

```
<variable> <variable'>%TYPE [NOT NULL] [DEFAULT <value>];
oder
<variable> <table>.<col>%TYPE [NOT NULL] [DEFAULT <value>];
```

Im ersten Fall bekommt `<variable>` den gleichen Datentyp wie die Variable `<variable'>` im zweiten Fall den Datentyp der Spalte `<col>` in der Tabelle `<table>`. Der Typ einer anchored-Deklaration wird zur Compile-Time bestimmt, d.h. wenn sich der Datentyp der verwendeten Datenbank ändert, muss das PL/SQL-Programm neu übersetzt werden.

Zuweisung an Variablen. Die Zuweisung von Werten zu Variablen geschieht in der bekannten Syntax `<variable> := <value>`.

PL/SQL-Datentypen. Zusätzlich zu den aus SQL bekannten Datentypen gibt es in PL/SQL weitere eingebaute Datentypen:

`BOOLEAN`: kann die Werte `TRUE`, `FALSE` und `NULL` annehmen:
```
is_capital: BOOLEAN;
```

`BINARY_INTEGER`, `PLS_INTEGER`: Ganzzahlen mit Vorzeichen. Dieser Datentyp rechnet auf der "klassischen" Repräsentation von Zahlen in Bytefolgen, d.h. Verknüpfungen finden ohne teure Konvertierung statt. `PLS_INTEGER` ist neu und besser.

`NATURAL`, `INT`, `SMALLINT`, `REAL`, `...`: Diverse numerische Datentypen.

`ROWID`: Jede Tabelle enthält eine *Pseudospalte*, die in jeder Zeile einen 6-Byte-Binärwert (vgl. Abschnitt über Speicherplatzberechnung) enthält, der die Zeile eindeutig in der Datenbank identifiziert (enthält den Block, die Reihe in diesem Block, sowie die Datei der Datenbank, wo man das Tupel findet)[1]. Der Zugriff über die `ROWID` ist mit Abstand die schnellste Methode, um auf ein Tupel zuzugreifen. Mit
```
SELECT Name, rowid FROM City;
```
bekommt man diese Spalte auch angezeigt.

Als Besonderheit können in PL/SQL *komplexe Datentypen* durch Deklarationen der Form

```
TYPE <name> IS <konstrukt>;
```

deklariert werden. Dabei kann `<konstrukt>` ein *Record* oder eine *PL/SQL-Table* sein:

`RECORD`: Ein Record enthält mehrere Felder und entspricht damit einem Tupel in der Datenbasis:
```
TYPE city_type IS RECORD
  (Name City.Name%TYPE,
   Country VARCHAR2(4),
   Province VARCHAR2(32),
   Population NUMBER,
   Longitude NUMBER,
   Latitude NUMBER);

the_city city_type;
```
definiert eine Variable `the_city` mit den angegebenen Feldern.

[1] `ROWID` gehört nicht zum SQL Standard

Häufig werden Records benötigt, deren Typ mit dem Typ einer Zeile einer bestimmten Tabelle übereinstimmt. Solche Deklarationen können analog zu anchored-Deklarationen als `%ROWTYPE`-Deklaration angegeben werden:

```
<variable> <table-name>%ROWTYPE;
```

Man hätte also in dem obigen Beispiel auch einfach

```
the_city City%ROWTYPE;
```

deklarieren können.

Zur Zuweisung an Records gibt es verschiedene Möglichkeiten:

- Aggregierte Zuweisung: Hat man zwei Variablen desselben Record-Typs, kann man sie als Ganzes zuweisen:

  ```
  <variable> := <variable'>;
  ```

- Feldzuweisung: Ein Feld eines Records wird einzeln zugewiesen:

  ```
  <record.feld> := <variable>|<value>;
  ```

- `SELECT INTO`: Mit der `SELECT INTO`-Anweisung kann das Ergebnis einer Anfrage, die nur *eine einzige Zeile* als Ergebnis haben liefert, direkt an eine geeignete Record-Variable übergeben werden:

  ```
  SELECT <column-list>
  INTO <record-variable>
  FROM ...
  WHERE ... ;
  ```

Beim Vergleich von Records muss jedes Feld einzeln verglichen werden.

PL/SQL `TABLE`: ist eine array-artige Struktur, die aus *einer* Spalte mit einem beliebigen Datentyp (also auch `RECORD`) mit einem optionalen Index vom Typ `BINARY_INTEGER` besteht:

```
TYPE <type> IS TABLE OF
  <datatype>
  [INDEX BY BINARY_INTEGER];

<var> <type>;
```

Die Tabelleneinträge werden dann wie üblich mit `<var>(1)` etc. angesprochen. Im Gegensatz zu Arrays in den bekannten Programmiersprachen muss die maximale Anzahl von Einträgen in PL/SQL Tables *nicht* im Voraus angegeben werden. Es wird einfach immer soviel Platz belegt, wie die vorhandenen Einträge benötigen. Tabellenwertige Variablen können nicht innerhalb von SQL-Anweisungen verwendet werden.

PL/SQL-Tabellen sind *sparse* (im Gegensatz zu *dense*); d.h. es werden nur diejenigen Zeilen gespeichert, die Werte enthalten. Damit ist es ohne weiteres möglich, in einer Tabelle nur einzelne, voneinander "weit entfernte" Zeilen zu definieren:

```
plz_table_type IS TABLE OF City.Name%TYPE
  INDEX BY BINARY_INTEGER;

plz_table plz_table_type;
plz_table(79110):= Freiburg;
plz_table(33334):= Kassel;
```

Nach dieser Zuweisung enthalten nur die Zeilen 33334 und 79110 Werte. Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

Zusätzlich bieten PL/SQL-Tabellen *built-in*-Funktionen und -Prozeduren. Diese werden mit

```
<variable> := <pl/sql-table-name>.<built-in-function>;
```
oder
```
<pl/sql-table-name>.<built-in-procedure>;
```
aufgerufen. Die folgenden built-in-Funktionen bzw. Prozeduren sind implementiert (PL/SQL Version 2.3):

- `COUNT` (fkt): Gibt die Anzahl der belegten Zeilen aus.
- `EXISTS` (fkt): `TRUE` falls Tabelle nicht leer ist.
- `DELETE` (proc): Löscht alle Zeilen einer Tabelle.
- `FIRST/LAST` (fkt): Gibt den niedrigsten/höchsten Indexwert, für den der Eintrag in der Tabelle definiert ist.
- `NEXT/PRIOR(n)` (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen Indexwert, für den der Eintrag in der Tabelle definiert ist.

Beispiel 14

```
plz_table.count = 2
plz_table.first = 33334
plz_table.next(33334) = 79110
```
□

14.3 SQL-Statements in PL/SQL

In der Execution Section eines PL/SQL-Blocks können an beliebigen Stellen DML-Kommandos, also `INSERT`, `UPDATE`, `DELETE` sowie `SELECT INTO`-Statements stehen. Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten.

Zusätzlich zu der bekannten SQL-Syntax ist es möglich, bei `INSERT`, `UPDATE` und `DELETE`-Befehlen, die *nur ein einziges Tupel betreffen* mit Hilfe der `RETURNING`-Klausel Werte an PL/SQL-Variablen zurückzugeben.

```
UPDATE ... SET ... WHERE ...
RETURNING <expr-list>
INTO <variable-list>;
```

Dies wird insbesondere verwendet, um die Row-ID eines gelöschten/geänderten/eingefügten Tupels zurückgeben zu lassen:

```
DECLARE rowid ROWID;
   ⋮
BEGIN
   ⋮
   INSERT INTO Politics (Country,Independence)
     VALUES (Code, SYSDATE)
     RETURNING ROWID
     INTO rowid;
   ⋮
END;
```

DDL-Statements werden in PL/SQL nicht direkt unterstützt. Will man DDL-Statements aus PL/SQL abgeben, so muss man das DBMS_SQL-Package benutzen. Dieses wird im Praktikum nicht behandelt.

14.4 Kontrollstrukturen

PL/SQL enthält die folgenden Kontrollstrukturen:

- `IF THEN - [ELSIF THEN] - [ELSE] - END IF`,
- verschiedene Schleifen:
- Simple `LOOP`:
  `LOOP ... END LOOP;`
- `WHILE LOOP`:
  `WHILE <bedingung> LOOP ... END LOOP;`
- Numeric `FOR LOOP`:
  `FOR <loop_index> IN [REVERSE] <Anfang> .. <Ende>`
  `LOOP ... END LOOP;`
  Die Variable `<loop_index>` wird dabei *automatisch* als `INTEGER` deklariert.
- Mit `EXIT [WHEN <bedingung>];` kann man einen `LOOP` jederzeit verlassen.
- den allseits beliebten `GOTO`-Befehl mit Labels:
  `<<label_i>> ... GOTO label_j;`

Zu bemerken ist, dass `NULL`-Werte (außer bei `IS NULL`-Abfrage) immer in den `ELSE`-Zweig verzweigen, da ein Nullwert keine Bedingung erfüllt.

Für `GOTO` gibt es einige Einschränkungen: Man kann nicht von außen in ein `IF`-Konstrukt, einen `LOOP`, oder einen lokalen Block hineinspringen, ebenfalls nicht von einem `IF`-Zweig in einen anderen; d.h. Labels sind lokal zu dem jeweiligen Block.

Da hinter einem Label immer mindestens ein ausführbares Statement stehen muss, und einem ein solches vielleicht nicht immer sinnvoll einfällt, gibt es das `NULL` Statement, das *nichts* tut.

14.5 Cursorbasierter Datenbankzugriff.

Cursors ermöglichen es, in einem PL/SQL-Programm auf Relationen *zeilenweise* zuzugreifen. Cursors werden ebenfalls in der *Declaration Section* definiert. Die Definition erfolgt dabei analog zu den `TYPE <name> IS ...`-Definitionen:

```
CURSOR <cursor-name> [(<parameter-list>)]
IS
  <select-statement>;
```

`(<parameter-list>)` gibt auch hier eine Parameter-Liste an. Dabei ist als Zugriffsart nur `IN` zugelassen. Da `IN` sowieso der Default-Wert ist, kann man diese Angabe auch gleich weglassen. Zwischen `SELECT` und `FROM` dürfen damit nicht nur die in SQL zugelassenen Ausdrücke stehen, sondern zusätzlich PL/SQL-Variablen und PL/SQL-Funktionen. PL/SQL-Variablen können ebenfalls in den `WHERE`-, `GROUP`- und `HAVING`-Klauseln verwendet werden.

Einen *Cursor* hat man sich als eine (geordnete !) Tabelle mit einem "Fenster", das über einem Tupel stehen kann und schrittweise vorwärts bewegt wird, vorzustellen.

Beispiel: Der folgende Cursor enthält alle Städte zu dem in der Variablen `the_country` angegebenen Land:

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

Im Umgang mit einem Cursor stehen drei Operationen zur Verfügung:

- `OPEN <cursor-name>[(<argument-list>)];`
  öffnet einen Cursor. Dabei wird das entsprechende `SELECT`-Statement abgearbeitet und eine *virtuelle Tabelle* erstellt. Die angegebenen Argumente werden für die `IN`-Parameter des Cursors eingesetzt. Der Cursor wird *vor* die erste Zeile dieser Tabelle gesetzt. Damit gibt es in dieser Situation keine *aktuelle Zeile*.
- `FETCH <cursor-name> INTO <record>;` oder
  `FETCH <cursor-name> INTO <variable-list>;`
  bewegt den Cursor auf die nächste Zeile des Ergebnisses der Anfrage und kopiert diese in die angegebene Record-Variable oder Variablenliste.
  Da ein Cursor eine *virtuelle Tabelle* ist, ist der entsprechende Datentyp durch die Tabellendeklarationen und das `SELECT`-Statement vorgegeben. Auch hier kann man mit der `<cursor-name>%ROWTYPE`-Deklaration eine Record-Variable mit dem entsprechenden Typ deklarieren:
  ```
  <variable> <cursor-name>%ROWTYPE;
  ```
- `CLOSE <cursor-name>;` schließt einen Cursor.

Beispiel: Für den oben deklarierten Cursor `cities_in` kann man eine geeignete Variable wie folgt deklarieren:

```
city_in cities_in%ROWTYPE;
```

Die einzelnen Spalten einer solchen `RECORD`-Variable spricht man dann mit den Spaltennamen, die ihnen das `SELECT`-Statement des Cursors zuordnet, an; z.B. `city_in.Name`. Hat man in dem `SELECT`-Statement des Cursors Spaltenaliase vergeben, werden diese als Feldbezeichner des so definierten Records verwendet.

Eine Befehlssequenz, die diesen Cursor und die Variable verwendet, sähe dann etwa so aus:

```
BEGIN
  OPEN cities_in ('D');
  FETCH cities_in INTO city_in;
  CLOSE cities_in;
END;
```

Wie man an den folgenden Befehlen sieht, ist es *nicht* möglich, einen parametrisierten Cursor für zwei verschiedene Werte offen zu haben:

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

Der `FETCH`-Befehl weiß nicht, aus welcher "Cursorinstanz" er den Wert holen sollte. D.h. man hat wirklich *einen* parametrisierten Cursor, *nicht* eine Familie von Cursoren!

Cursorattribute. Normalerweise wird der Inhalt eines Cursors in einer Schleife verarbeitet (im obigen Beispiel wird *nur eine* Stadt aus *city_in* verarbeitet). Um eine solche Schleife zu formulieren, kann man *Cursorattribute* verwenden:

- `<cursor-name>%ISOPEN`: gibt an, ob ein Cursor geöffnet ist.
- `<cursor-name>%FOUND`: Solange ein Cursor bei der letzten `FETCH`-Operation ein neues Tupel gefunden hat, ist `<cursor-name>%FOUND = TRUE`.
- `<cursor-name>%NOTFOUND`: hat man alle Zeilen eines Cursors ge`FETCH`t, nimmt dieses Attribut den Wert `TRUE` an.
- `<cursor-name>%ROWCOUNT`: gibt zu jedem Zeitpunkt an, wieviele Records von einem Cursor bereits gelesen wurden.
- Diese Cursorattribute dürfen nicht innerhalb eines SQL-Ausdrucks verwendet werden – sie werden für die Auswertung in Schleifenbedingungen eingesetzt.

Cursor FOR-LOOP. Speziell für den Umgang mit Cursoren gibt es den *Cursor `FOR LOOP`*:

```
FOR <record_index> IN <cursor-name> | <select-statement>
 LOOP ... END LOOP;
```

Die Variable `<record_index>` wird dabei *automatisch* mit dem entsprechenden `%ROWTYPE` deklariert.

Bei jeder Ausführung des Schleifenkörpers wird dabei *automatisch* ein `FETCH` in die Schleifenvariable ausgeführt (daher wird der Schleifenkörper i.a. *keinen* `FETCH`-Befehl enthalten!) Außerdem spart man die Deklaration der Variablen eines `RECORD`-Typs sowie die `OPEN`- und `CLOSE`-Statements.

Beispiel: Für jede Stadt in dem gegebenen Land soll eine Prozedur “request_Info” aufgerufen werden:

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

BEGIN
   the_country:='D';   % oder sonstwie setzen
   FOR the_city IN cities_in(the_country)
   LOOP
     request_Info(the_city);
   END LOOP;
END;
```

Man kann die `SELECT`-Anfrage dabei auch direkt in der `FOR`-Klausel schreiben. Es ist zu beachten, dass ein Cursor *immer* ein Record-Type ist – ggf. ein einspaltiges. Um eine Spalte eines Cursors einzeln zu bekommen, muss man sie explizit adressieren:

```
CREATE TABLE big_cities
(name VARCHAR2(25));
BEGIN
   FOR the_city IN
     SELECT Name
     FROM City
     WHERE Country = the_country
     AND Population > 1000000
```

```
    LOOP
      INSERT INTO big_cities VALUES (the_city.Name);
    END LOOP;
  END;
```

Wenn ein SELECT-Statement nur *eine einzige* Zeile liefert, benötigt man keinen expliziten Cursor, um sie zu verarbeiten, sondern kann das SELECT-Statement als *impliziten* Cursor betrachten und direkt INTO ein Record SELECTen:

```
SELECT <column-list>
INTO <record>
FROM <table-list>;
```

Mit den bis jetzt eingeführten Befehlen kann man das Ergebnis einer Anfrage zeilenweise holen und weiterverarbeiten. Bis jetzt ist es jedoch nicht möglich, eine Tabelle zeilenweise zu verändern. Dazu dient die WHERE CURRENT OF-Klausel:

Die folgenden Kommandos ändern/löschen jeweils das zuletzt von dem genannten Cursor geFETCHte Tupel:

```
UPDATE <table-name>
SET <set_clause>
WHERE CURRENT OF <cursor_name>;

DELETE FROM <table-name>
WHERE CURRENT OF <cursor_name>;
```

14.6 Nutzung Geschachtelter Tabellen unter PL/SQL

Geschachtelte Tabellen (vgl. Abschnitt 5.2) können erst unter Verwendung von PL/SQL vollständig genutzt werden.

In Abschnitt 5.2 wurde gezeigt, dass die Nutzung geschachtelter Tabellen in ORACLE nicht ganz unproblematisch ist, da eine SELECT THE jeweils nur ein Objekt zurückgeben kann (keine Korrelation mit umgebenden Tupeln möglich). Durch Verwendung einer (Cursor-)Schleife kann dieses Problem gelöst werden.

Zusätzlich können die oben für PL/SQL-Tables beschriebenen Methoden COUNT, EXISTS, DELETE, FIRST/LAST und NEXT/PRIOR(n) *aus PL/SQL* (in SQL-Statements innerhalb SQL *nicht* erlaubt) auch auf geschachtelte Tabellen angewendet werden.

Dieser Abschnitt liefert gleichzeitig ein Beispiel, wie eine komplette Prozedur aussieht.

Beispiel 15 In Anschnitt 5.2 wurde eine geschachtelte Tabelle zur Speicherung der in einem Land gesprochenen Sprachen angelegt. Mit der folgenden Prozedur lassen sich auf Basis der geschachtelten Tabelle alle Länder, in denen eine bestimmte Sprache gesprochen wird, auslesen:

```
DROP TABLE tempCountries;
CREATE TABLE tempCountries
 (Land    VARCHAR2(4),
  Sprache VARCHAR2(20),
  Anteil  NUMBER);
```

```
CREATE OR REPLACE PROCEDURE Search_Countries
(the_Language IN VARCHAR2)
IS
  CURSOR countries IS
    SELECT Code
    FROM Country;
BEGIN
  DELETE FROM tempCountries;
  FOR the_country IN countries
  LOOP
    INSERT INTO tempCountries
    SELECT the_country.code,Name,Percentage
    FROM THE(SELECT Languages
             FROM NLanguage
             WHERE Country = the_country.Code)
    WHERE Name = the_Language;
  END LOOP;
END;
/

EXECUTE Search_Countries('English');
SELECT * FROM tempCountries;
```

An dieser Stelle ist zu beachten, dass der Inhalt des einspaltigen Cursors durch `the_country.Code` angesprochen werden muss – `SELECT the_country,Name,Percentage` erzeugt eine Fehlermeldung (häufiger Fehler)! □

Die bisher beschriebenen Funktionen und Prozeduren werden durch den Benutzer explizit aufgerufen. Daneben gibt es noch *Trigger* (vgl. Abschnitt 14.7), deren Ausführung durch das Eintreten eines Ereignisses in der Datenbank angestoßen wird.

14.7 Trigger

Trigger sind eine spezielle Form von PL/SQL-Prozeduren, die beim Eintreten eines bestimmten Ereignisses vom System automatisch ausgeführt werden. Damit sind sie ein Spezialfall aktiver Regeln nach dem **E**vent-**C**ondition-**A**ction-Paradigma.

In Oracle 8 gibt es verschiedene Arten von Triggern: `BEFORE`- und `AFTER`-Trigger sind bereits aus Oracle 7 bekannt, während `INSTEAD OF`-Trigger erst in Version 8 eingeführt wurden.

14.7.1 `BEFORE`- und `AFTER`-Trigger

Ein `BEFORE`- oder `AFTER`-Trigger ist einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet. Seine Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst. Die Ausführung eines Triggers kann zusätzlich von Bedingungen an den Datenbankzustand abhängig gemacht werden. Weiterhin unterscheidet man, ob ein Trigger *vor* oder *nach* der Ausführung der entsprechenden aktivierenden Anweisung in der Datenbank ausgeführt

wird. Ein Trigger kann einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) seiner Tabelle ausgeführt werden.

Die Definition eines Triggers besteht aus einem Kopf, der die oben beschriebenen Eigenschaften enthält, sowie einem in PL/SQL geschriebenen Rumpf.

Da Trigger im Zusammenhang mit Veränderungen an (Zeilen) der Datenbasis verwendet werden, gibt es die Möglichkeit im Rumpf den alten und neuen Wert des gerade behandelten Tupels zu verwenden.

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  BEFORE | AFTER
  {INSERT | DELETE | UPDATE} [OF <column-list>]
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ⋮
  [ OR {INSERT | DELETE | UPDATE} [OF <column-list>]]
  ON <table>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH ROW]
  [WHEN (<trigger-condition>)]
  <pl/sql-block>;
```

- Mit `BEFORE` und `AFTER` wird angegeben, ob der Trigger vor oder nach Ausführung der auslösenden Operation (einschließlich aller referentiellen Aktionen) ausgeführt wird.
- Die Angabe von `OF <column>` ist nur für `UPDATE` erlaubt. Wird `OF <column>` nicht angegeben, so wird der Trigger aktiviert wenn irgendeine Spalte eines Tupels verändert wird.
- Mittels der in `REFERENCING OLD AS ... NEW AS ...` angegebenen *Transitions-Variablen* kann auf die Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion zugegriffen werden. Als Default erreicht man diese Werte unter `:OLD` bzw. `:NEW`.
  Schreibzugriff auf `:NEW`-Werte ist nur mit `BEFORE`-Triggern erlaubt.
- `FOR EACH ROW` definiert einen Trigger als Row-Trigger. Fehlt diese Zeile, wird der Trigger als Statement-Trigger definiert.
- Mit `WHEN` kann die Ausführung eines Triggers weiter eingeschränkt werden. Insbesondere können die Transitionsvariablen `OLD` und `NEW` in der `WHEN`-Bedingung verwendet werden.
- Der `<pl/sql-block>` eines Triggers darf keine Befehle zur Transaktionskontrolle enthalten.
- Ist ein Trigger für verschiedene Ereignisse definiert, kann das auslösende Ereignis im Rumpf durch `IF INSERTING THEN`, `IF UPDATING OF <column-list> THEN` usw. abgefragt werden.

Die Verwendung von Triggern gemeinsam mit referentiellen Integritätsbedingungen ist problematisch. Dazu kommt, dass dabei häufig unklare Fehlermeldungen erzeugt werden. Daher ist es naheliegend, wenn man schon referentielle Integritätsbedingungen betrachtet, diese komplett durch Trigger zu überwachen und einzuhalten. Dabei lässt sich auch `ON UPDATE CASCADE` mit Triggern verwirklichen:

Beispiel 16 (Umbenennung eines Landes, vgl. Bsp. 9) Wenn ein Landes-Code geändert wird, pflanzt sich diese Änderung auf die Relation *Province* fort:

```
CREATE OR REPLACE TRIGGER change_Code
BEFORE UPDATE OF Code ON Country
FOR EACH ROW
BEGIN
  UPDATE Province
```

```
  SET Country = :NEW.Code
  WHERE Country = :OLD.Code;
END;
/

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

Beispiel 17 (Gründung eines Landes) Wenn ein Land neu angelegt wird, wird ein Eintrag in *Politics* mit dem aktuellen Jahr erzeugt:

```
CREATE TRIGGER new_Country
AFTER INSERT ON Country
FOR EACH ROW
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code, SYSDATE);
END;
/

INSERT INTO Country (Name,Code) VALUES ('Lummerland', 'LU');
SELECT * FROM Politics;
```

Mutating Tables. Zeilenorientierte Trigger werden immer direkt vor/nach der Veränderung einer Zeile aufgerufen. Damit "sieht" jede Ausführung eines solchen Triggers einen anderen Datenbestand der Tabelle, auf der er definiert ist, sowie der Tabellen, die er evtl. ändert. Um hier zu garantieren, dass das Ergebnis *unabhängig von der Reihenfolge* der veränderten Tupel ist, werden die entsprechenden Tabellen während der gesamten Aktion als *mutating* gekennzeichnet (ebenso Tabellen, die durch ein `ON DELETE CASCADE` von den Änderungen an einer solchen Tabelle betroffen sein können. Tabellen, die als *mutating* markiert sind, können nicht von Triggern gelesen oder geschrieben werden. Analoge Einschränkungen gelten auch für Tabellen, die über Fremdschlüssel ohne `CASCADE` verbunden sind (*constraining*).

Dieses *mutating table syndrom* ist bei der Triggerprogrammierung ausgesprochen störend, und kann in einigen Fällen nur durch umständliche Tricks gelöst werden. Man kann folgende Fälle unterscheiden:

- Trigger soll auf diejenige Tabelle zugreifen auf der er selber definiert ist.
  - Nur das auslösende Tupel soll von dem Trigger gelesen/geschrieben werden: In diesem Fall ist in dem Trigger kein Datenbankzugriff notwendig, sondern es sollte ein `BEFORE`-Trigger und die `:NEW`- und `:OLD`-Variablen verwendet werden: Durch Setzen des `:NEW`-Wertes *vor* dem Datenbankzugriff werden die `:NEW`-Werte bei dem anschließenden Datenbankzugriff in die Datenbank geschrieben.
  - Es sollen neben dem auslösenden Tupel auch weitere Tupel gelesen werden: Dann muss ein Statement-orientierter Trigger verwendet werden (hier sollte auch klar sein, dass bei der Verwendung eines zeilenorientierter Triggers jedes Mal ein anderer Zustand derselben Tabelle gelesen/geschrieben würde!).
- Trigger soll auf andere Tabellen zugreifen, die als *mutating* gekennzeichnet sind. In diesem Fall muss ein Statement-Trigger verwendet werden. Gegebenenfalls müssen Hilfstabellen verwendet werden.

Beispiel 18 (Mutating Table Syndrom: Lösung mit Hilfstabelle) Bei der Umbenennung eines Flusses muss für alle Flüsse, die in den umbenannten Fluss fließen, der Name des Zielflusses ebenfalls geändert werden (Definition der Tabelle *River* siehe Seite 44). Da dies für jede veränderte Zeile durchgeführt werden muss, ist ein Zeilentrigger notwendig. Der naheliegende Ansatz mit

```
CREATE OR REPLACE TRIGGER upd_river_name -- NICHT ERLAUBT !!!
AFTER UPDATE OF Name ON River
FOR EACH ROW       -- Row-Trigger
BEGIN
  UPDATE River
  SET River = :NEW.Name
  WHERE River = :OLD.Name;
END;
/
UPDATE RIVER
SET Name = 'Congo'
WHERE Name = 'Zaire';
```

schlägt fehl: Dieser Trigger greift auf andere Tupel der Tabelle *River* zu.

Als Lösung wird eine Hilfstabelle angelegt, die alle Umbenennungen protokolliert (und durch einen Zeilentrigger unterhalten wird). Wenn alle Flüsse umbenannt sind, wird mit einem Statement-`AFTER`-Trigger die Hilfstabelle ausgewertet, um die Zielflüsse anzupassen:

```
CREATE TABLE aux_rename_river  -- Hilfstabelle
(old VARCHAR2(20),
 new VARCHAR2(20));

CREATE OR REPLACE TRIGGER upd_river_name -- Row-Trigger
BEFORE UPDATE OF Name ON River
FOR EACH ROW
BEGIN
  INSERT INTO aux_rename_river
  VALUES(:OLD.Name,:NEW.Name);
END;
/
CREATE OR REPLACE TRIGGER upd_river_names -- Statement-Trigger
AFTER UPDATE OF Name ON River
BEGIN
  UPDATE River
  SET River = (SELECT new
               FROM aux_rename_river
               WHERE old = River)
  WHERE River IN (SELECT old
               FROM aux_rename_river);
  DELETE FROM aux_rename_river;
END;
/
```

14.7.2 INSTEAD OF-Trigger

INSTEAD OF-Trigger dienen dazu, *View Updates* (vgl. Abschnitt 9.1) umzusetzen. Damit sind INSTEAD OF-Trigger immer zu Views zugeordnet:

```
CREATE [OR REPLACE] TRIGGER <trigger-name>
  INSTEAD OF
  {INSERT | DELETE | UPDATE} ON <view>
  [REFERENCING OLD AS <name> NEW AS <name>]
  [FOR EACH STATEMENT]
  <pl/sql-block>;
```

Die Bearbeitung wird ebenfalls durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen des Views) ausgelöst, wobei allerdings *nicht* nach einzelnen Spalten aufgelöst werden kann. Das Auslösen eines Triggers kann hier nicht zusätzlich von Bedingungen an den Datenbankzustand abhängig gemacht werden. Im Gegensatz zu BEFORE- und AFTER-Triggern ist für INSTEAD OF-Trigger FOR EACH ROW als Default gesetzt.

```
CREATE OR REPLACE VIEW AllCountry AS
SELECT Name, Code, Population, Area, GDP, Population/Area AS Density,
 Inflation, population_growth, infant_mortality
FROM Country, Economy, Population
WHERE Country.Code = Economy.Country
  AND Country.Code = Population.Country;

SELECT * from user_updatable_columns
WHERE table_name = 'ALLCOUNTRY';

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

ergibt eine Fehlermeldung, dass über ein Join-View nur *eine* Basistabelle modifiziert werden kann. Der folgende INSTEAD OF-Trigger verteilt das Update geeignet auf die Relationen:

```
CREATE OR REPLACE TRIGGER InsAllCountry
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
   INSERT INTO Country (Name,Code,Population,Area)
   VALUES (:NEW.Name, :NEW.Code, :NEW.Population, :NEW.Area);
   INSERT INTO Economy (Country,Inflation)
   VALUES (:NEW.Code, :NEW.Inflation);
   INSERT INTO Population (Country, Population_Growth,infant_mortality)
   VALUES (:NEW.Code, :NEW.Population_Growth, :NEW.infant_mortality);
END;
/
```

Zusammen mit dem Trigger *New_Country*, der das Gründungsdatum automatisch einträgt, werden somit die Relationen *Country*, *Economy*, *Population* und *Politics* aktualisiert.

Für *Object Views* (vgl. Abschnitt 16) ist es jedoch sinnvoller, den Benutzer erst gar keine Updates an das View stellen zu lassen, sondern die entsprechende Funktionalität durch Methoden zur Verfügung zu stellen, die die Änderungen direkt auf den zugrundeliegenden Basistabellen ausführen.

Zugriffsrechte. Bei der Verwendung von Triggern muss man berücksichtigen, dass Trigger genauso wie Prozeduren und Funktionen mit den Zugriffsrechten des *Besitzers* ausgeführt werden. Da man Zugriffsrechte an Triggern nicht explizit vergeben kann, werden sie *automatisch* von jedem benutzt, der entsprechende Veränderungen an der Trigger-Tabelle vornimmt.

Das folgende Beispiel ist sinngemäß aus [CHRS98] entnommen:

```
CREATE OR REPLACE TRIGGER bla
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 IF user='may'
  THEN NULL;
 END IF;
...
END;
/

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

1 Zeile wurde erstellt.

SQL> select * from allcountry where Code='LU';

Es wurden keine Zeilen ausgewaehlt
```

Das ORACLE-Echo gibt beim Einsatz von Triggern *nicht* die tatsächlich ausgeführten Aktionen an.

14.8 Ausnahmebehandlung von Fehlersituationen

Die *Exception Section* eines PL/SQL-Blocks gibt an, wie ggf. auf Ausnahme- und Fehlersituationen reagiert werden soll. Benutzerdefinierte Exceptions werden im Deklarationsteil durch ‹exception› EXCEPTION deklariert. Die beim Auftreten einer Exception auszuführenden Aktionen werden in der *Exception Section* definiert. Exceptions können dann an beliebigen Stellen des PL/SQL-Blocks durch `RAISE` ausgelöst werden.

```
DECLARE
<exception> EXCEPTION;
   ⋮
```

```
BEGIN
   ⋮
   IF ...
   THEN RAISE < exception>;
   ⋮
EXCEPTION
   WHEN <exception_1>
   THEN <PL/SQL-Statement>;
   ⋮
   WHEN <exception_n>
   THEN <PL/SQL-Statement>;
   [WHEN OTHERS THEN <PL/SQL-Statement>;] END;
```

Wird eine Exception ausgelöst, so wird die in der entsprechenden `WHEN`-Klausel aufgeführte Aktion ausgeführt und der innerste Block verlassen (hier ist oft die Anwendung von anonymen Blöcken sinnvoll). Ist für eine Fehlermeldung keine Aktion in der Exception-Section angegeben, stattdessen aber eine Aktion unter `WHEN OTHERS` angegeben wird diese ausgeführt.

Zum Auslösen vordefinierter Fehlermeldungen dient weiterhin `RAISE_APPLICATION_ERROR`:

Beispiel 19 (Zeitabhängige Bedingung) Nachmittags dürfen keine Städte gelöscht werden:

```
CREATE OR REPLACE TRIGGER nachm_nicht_loeschen
BEFORE DELETE ON City
BEGIN
  IF TO_CHAR(SYSDATE,'HH24:MI') BETWEEN '12:00' AND '18:00'
  THEN RAISE_APPLICATION_ERROR(-20101,'Unerlaubte Aktion');
  END IF;
END;
/
```

Dieses Beispiel verwendet einen Statement-Trigger, und zeigt, wie Fehlermeldungen programmiert werden können. □

Was es sonst noch gibt. In der Kurseinheit über PL/SQL wurde ein Überblick über PL/SQL gegeben. Dabei wurden einige Features nicht behandelt:

- *Packages*: Möglichkeit, Daten und Programme zu kapseln;
- `FOR UPDATE`-Option bei Cursordeklarationen;
- *Cursorvariablen*;
- *Exception Handlers*;
- *benannte* Parameterübergabe;
- PL-SQL Built-in Funktionen: Parsing, String-Operationen, Datums-Operationen, Numerische Funktionen;
- Built-in Packages, die einem Anwender das Leben mit einem DBMS erheblich erleichtern.

Unter PL/SQL wird dann auch die Verwendung von `SAVEPOINT`s für Transaktionen interessant, wenn man wirklich komplexe Transaktionen definieren kann.

Teil IV

Objektorientierung in Oracle 8

15 OBJEKT-RELATIONALE ERWEITERUNGEN

Objekt-Relationale Datenbanksysteme – u.a. ORACLE 8 – vereinen relationale Konzepte und Objektorientierung. Dabei werden die folgenden Konzepte angeboten [Vos94]:

- Komplexe und Abstrakte Datentypen. Dabei werden *Value Types* und *Object Types* unterschieden. Erstere erweitern nur das Domain-Konzept von SQL2, letztere unterstützen Objekt-Identität und Kapselung interner Funktionalität.
- Spezialisierung: ein Typ kann als Subtyp eines oder mehrerer anderer Typen vereinbart werden. Dabei hat jeder Subtyp *genau* einen (direkten oder indirekten) maximalen Supertyp (der keinen weiteren Supertyp besitzt).
- Tabellenverbände als Form der Spezialisierung von Relationen: Eine Tabelle kann als Subtabelle einer oder mehrerer anderer Tabellen definiert werden.
- Funktionen als Bestandteile eines ADT's oder von Tabellen, oder freie Funktionen.
- Methoden- und Funktionsaufrufe im Rahmen von `SELECT`-Ausdrücken.

15.1 Objekte

Objektorientierung bedeutet, dass zwischen dem *Zustand* und dem *Verhalten* eines *Objektes* unterschieden wird. Objektorientierung in ORACLE 8 bedeutet, dass es neben Tabellen, deren Inhalt aus Tupels besteht, auch *Object Tables* gibt, deren Inhalt aus Objekten besteht. Ebenso können einzelne Spalten einer Tupeltabelle *objektwertig* sein. Im Gegensatz zu einem *Tupel* besitzt ein Objekt *Attribute*, die den inneren Zustand des Objektes beschreiben, sowie *Methoden*, mit denen der Zustand abgefragt und manipuliert werden kann. Einfache Objekttypen wurden bereits in Abschnitt 5.1 als *komplexe Attribute* behandelt: Komplexe Attribute sind Objekttypen, die nur *Wertattribute* und keine Methoden besitzen. Objekte können neben *Wertattributen* auch *Referenzattribute* besitzen: In diesem Fall enthält das Attribut eine Referenz auf ein anderes Objekt. Referenzattribute werden durch `REF <object-datatype>` gekennzeichnet.

Grob gesehen entsprechen Objekttabellen den Klassen in objektorientierten Programmiersprachen während Tupel den Instanzen entsprechen. Im Gegensatz zu der üblichen Vorstellung von Objektorientierung bietet ORACLE 8 keine Vererbung zwischen Klassen, Subklassen und Instanzen. Damit beschränkt sich die Klassen-/Tabellenzugehörigkeit darauf, gemeinsame Methoden aller zugehörigen Objekte zu definieren. Als Methoden stehen *Prozeduren* und *Funktionen* zur Verfügung, wobei eine Funktion durch `MAP` oder `ORDER` ausgezeichnet werden kann um eine Ordnung auf den Objekten einer Klasse zu definieren (anderenfalls sind nur Tests auf Gleichheit erlaubt). Die *Signaturen* der Methoden sowie ihre `READ/WRITE` Zugriffscharakteristik werden in der `TYPE`-Deklaration angegeben, die Implementierung der Methoden ist als PL/SQL-Programm im `TYPE BODY` gegeben.

Die allgemeine Form einer Objekttypdeklaration sieht damit – grob – so aus:

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (  <attr> <datatype>,
     ⋮
     <attr> REF <object-datatype>,
     ⋮
     MEMBER FUNCTION <func-name> [(<parameter-list>)] RETURN <datatype>,
     ⋮
     MEMBER PROCEDURE <proc-name> [(<parameter-list>)],
     ⋮
     [ MAP MEMBER FUNCTION <func-name> RETURN <datatype>, |
       ORDER MEMBER FUNCTION <func-name>(<var> <type>) RETURN <datatype>,]
     [ <pragma-declaration-list> ]
     );
```

Dabei muss `<object-datatype>` ebenfalls durch einen Befehl der Form `CREATE TYPE <object-datatype> AS OBJECT ...` als Objekttyp deklariert sein. `<parameter-list>` hat dieselbe Syntax wie bereits in Abschnitt 14.1 für Prozeduren und Funktionen angegeben. Der erste Teil der Definition, in dem die Attribute festlegt werden ist der Definition von Tabellen durch `CREATE TABLE` (vgl. Abschnitt 2.1) sehr ähnlich – es ist jedoch zu beachten, dass bei `CREATE TYPE` *keine* Integritätsbedingungen angegeben werden können (diese werden erst bei der (Objekt)tabellen-Definition angegeben).

`PRAGMA`-Klauseln. In `<pragma-declaration-list>` kann für jede Methode eine `PRAGMA`-Klausel der Form

```
PRAGMA RESTRICT_REFERENCES (<method_name>,<feature-list>));
```

angegeben werden, die spezifiziert, ob eine Prozedur/Funktion schreibend/lesend auf die Datenbank zugreifen darf. `<feature-list>` ist eine Komma-Liste mit den möglichen Einträgen

`WNDS`	Writes no database state,
`WNPS`	Writes no package state,
`RNDS`	Reads no database state,
`RNPS`	Reads no package state.

Diese Angabe ist insbesondere für Funktionen wichtig, da Oracle diese nur ausführt, wenn *zugesichert* ist, dass sie den Datenbankzustand nicht verändern. Daher muss bei Funktionen zumindest

```
PRAGMA RESTRICT_REFERENCES (<function_name>,WNPS,WNDS);
```

gesetzt werden.

`MAP`- und `ORDER`-Funktionen erfordern

```
PRAGMA RESTRICT_REFERENCES (<function-name>,WNDS,WNPS,RNPS,RNDS)
```

d.h. sie dürfen *keinen Datenbankzugriff* enthalten (was sich als sehr strenge Einschränkung herausstellt).

Beispiel 20 (Objekttyp *GeoCoord*) Der komplexe Attributtyp *GeoCoord* (vgl. Beispiel 6) kann wie folgt um eine Methode *Distance* erweitert werden, die die Entfernung zu einem gegebenen zweiten *GeoCoord*-Wert berechnet. Außerdem wird die Entfernung von Greenwich als `MAP`-Methode deklariert:

```
CREATE OR REPLACE TYPE GeoCoord AS OBJECT
```

```
(Longitude NUMBER,
 Latitude NUMBER,
 MEMBER FUNCTION Distance (other IN GeoCoord) RETURN NUMBER,
 MAP MEMBER FUNCTION Distance_Greenwich RETURN NUMBER,
 PRAGMA RESTRICT_REFERENCES (Distance,WNPS,WNDS,RNPS,RNDS));
 PRAGMA RESTRICT_REFERENCES (Distance_Greenwich,WNPS,WNDS,RNPS,RNDS));
/
```

Der `TYPE BODY` enthält die Implementierung der Objektmethoden in PL/SQL. Für jedes Objekt ist automatisch die Methode `SELF` definiert, mit der das Objekt selber als Host-Objekt einer Methode angesprochen werden kann. `SELF` wird in Methodendefinitionen verwendet, um auf die Attribute des Host-Objektes zuzugreifen.

Die Definition des `TYPE BODY` muss der bei `CREATE TYPE` vorgegeben Signatur desselben Typs entsprechen. Insbesondere muss für *alle* deklarierten Methoden eine Implementierung angegeben werden.

```
CREATE [OR REPLACE] TYPE BODY <type>
AS
     MEMBER FUNCTION <func-name> [(<parameter-list>)] RETURN <datatype>
     IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;
     ⋮
     MEMBER PROCEDURE <proc-name> [(<parameter-list>)]
     IS
        [<var-decl-list>;]
        BEGIN <PL/SQL-code> END;
     ⋮
     [ MAP MEMBER FUNCTION <func-name> RETURN <datatype> |
       ORDER MEMBER FUNCTION <func-name>(<var> <type>) RETURN <datatype>
      IS
       [<var-decl-list>;]
       BEGIN <PL/SQL-code> END;]
END;
/
```

Beispiel 21 (Objekttyp *GeoCoord* (Forts.)) Der `TYPE BODY` zu *GeoCoord* enthält nun die Implementierung der Methoden *Distance* und *Distance_Greenwich*:

```
CREATE OR REPLACE TYPE BODY GeoCoord
AS
MEMBER FUNCTION Distance (other IN GeoCoord)
  RETURN NUMBER
  IS
  BEGIN
   RETURN 6370*ACOS(cos(SELF.latitude/180*3.14)*cos(other.latitude/180*3.14)
              *cos((SELF.longitude-other.longitude)/180*3.14)
              + sin(SELF.latitude/180*3.14)*sin(other.latitude/180*3.14));
  END;
```

```
MAP MEMBER FUNCTION Distance_Greenwich
  RETURN NUMBER
  IS
  BEGIN
   RETURN SELF.Distance(GeoCoord(0,51));
  END;
END;
/
```

Methoden eines Objektes werden mit

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

aufgerufen.

Zeilen- und Spaltenobjekte. Im weiteren unterscheidet man *Zeilenobjekte* und *Spaltenobjekte*: Zeilenobjekte sind Elemente von *Objekttabellen*. Spaltenobjekte erhält man, wenn ein Attribut eines Tupels (oder eines Objekts) objektwertig ist (vgl. Abschnitt 5.1). Während Zeilenobjekte eine eindeutige OID erhalten (siehe Abschnitt 15.3) über die sie *referenzierbar* sind, haben Spaltenobjekte *keine* OID, sind also *nicht* referenzierbar. Bei der Tabellendefinition werden wie üblich auch Integritätsbedingungen angegeben.

Wie bereits in Abschnitt 5.1 werden Zeilen- und Spaltenobjekte mit Hilfe der entsprechenden *Konstruktormethode* erzeugt. Zu beachten ist hierbei, dass die für einen n-stelligen Konstruktor auch n Argumente angegeben werden. Wenn die Werte zum Zeitpunkt der Objekterzeugung noch nicht bekannt sind, müssen `NULL`werte angegeben werden.

Bei `SELECT`-Statements müssen immer Tupel- bzw. Objektvariablen durch Aliasing verwendet werden, wenn ein *Zugriffspfad* `<var>.<method>*[.<attr>]` angegeben wird – also wenn Attribute oder Methoden von objektwertigen Attributen selektiert werden.

15.2 Spaltenobjekte

Spaltenobjekte erhält man, indem man ein Attribut einer Tabelle (Tupel- oder Objekttabelle) objektwertig definiert (auch komplexe Attribute wie in Abschnitt 5.1 definieren Spaltenobjekte).

Beispiel 22 (Spaltenobjekte) Der in Beispiel 20 erzeugte komplexe Attributtyp *GeoCoord* wird wie bereits in Beispiel 6 in der Definition der Tabelle *Mountain* verwendet, um *Spaltenobjekte* zu erzeugen:

```
CREATE TABLE Mountain
(Name VARCHAR2(20) CONSTRAINT MountainKey PRIMARY KEY,
 Height NUMBER CONSTRAINT MountainHeight
     CHECK (Height > = 0),
 Coordinates GeoCoord CONSTRAINT MountainCoord
     CHECK ((Coordinates.Longitude >= -180) AND
            (Coordinates.Longitude <= 180) AND
            (Coordinates.Latitude >= -90) AND
            (Coordinates.Latitude <= 90)));
```

Spaltenobjekte werden nun mit Hilfe der entsprechenden Konstruktormethode erzeugt:

```
INSERT INTO Mountain VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

Die Entfernung eines Berges zum Nordpol erhält man dann mit

```
SELECT Name, mt.coordinates.Distance(GeoCoord(0,90))
FROM Mountain mt;
```

unter Verwendung der Tupelvariablen `mt` um den Zugriffspfad zu *coordinates.Distance* eindeutig zu machen. □

Enthält ein in einer objektwertigen Spalte `<attr>` verwendeter Objekttyp `<object-type>` eine tabellenwertiges Attribut `<tab-attr>`, so muss bei der `NESTED TABLE ... STORE AS`-Klausel im `CREATE TABLE`-Statement ebenfalls ein Pfadausdruck verwendet werden:

```
CREATE TABLE ...
  (  ...
     <attr> <object-type>
     ...)
  NESTED TABLE <attr>.<tab-attr> STORE AS <name>;
```

15.3 Zeilenobjekte

Objekttabellen enthalten im Gegensatz zu den "klassischen" relationalen Tupeltabellen keine Tupel, sondern Objekte. Innerhalb jeder Objekttabelle besitzt jedes Objekt eine eindeutige *OID (Objekt-ID)*, die dem *Primärschlüssel* im relationalen Modell entspricht. Dieser wird mit den (weiteren) Integritätsbedingungen bei der Tabellendefinition angegeben. Erwähnenswert ist hierbei die problemlose Integration referentieller Integritätsbedingungen von Objekttabellen zu bestehenden relationalen Tabellen durch die Möglichkeit, Fremdschlüsselbedingungen in der gewohnten Syntax anzugeben.

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

`<constraint-list>` ist dabei eine Liste von attributbezogenen Bedingungen sowie Tabellenbedingungen – letztere haben dieselbe Syntax wie bei Tupeltabellen. Attributbedingungen entsprechen den Spaltenbedingungen `<column-constraint>` bei Tupeltabellen und sind von der Form

```
  <attr-name> [DEFAULT <value>]
     [<colConstraint> ... <colConstraint>]
```

Beispiel 23 (Objekt-Tabelle *City*) Als Beispiel wird eine Objekttabelle definiert, die alle Städte enthält, und wie oben den Datentyp *GeoCoord* verwendet. Der Objekttyp *City_Type* wird analog zu der Tupeltabelle *City* definiert:

```
CREATE OR REPLACE TYPE City_Type AS OBJECT
 (Name VARCHAR2(35),
  Province VARCHAR2(32),
  Country VARCHAR2(4),
  Population NUMBER,
  Coordinates GeoCoord,
 MEMBER FUNCTION Distance (other IN City_Type) RETURN NUMBER,
```

```
 PRAGMA RESTRICT_REFERENCES (Distance,WNPS,WNDS,RNPS,RNDS));
/
```

Die Methode *Distance(city)* dient dazu, die Entfernung zu einer anderen Stadt zu berechnen. Sie basiert auf der Methode *Distance(geo_coord)* des Datentyps *Geo_Coord*. Auch hier wird wieder `SELF` verwendet, um auf die Attribute des Host-Objekts zuzugreifen:

```
CREATE OR REPLACE TYPE BODY City_Type
AS
 MEMBER FUNCTION Distance (other IN City_Type)
 RETURN NUMBER
 IS
 BEGIN
   RETURN SELF.coordinates.Distance(other.coordinates);
 END;
END;
/
```

Die Objekttabelle wird nun entsprechend definiert, wobei der (mehrspaltige) Primärschlüssel als Tabellenbedingung angegeben werden kann. Der Primärschlüssel darf keine `REF`-Attribute enthalten (vgl. Abschnitt 15.4).[1] Die Fremdschlüsselbedingung für *Country* wird ebenfalls als Tabellenbedingung angegeben:

```
CREATE TABLE City_ObjTab OF City_Type
 (PRIMARY KEY (Name, Province, Country),
  FOREIGN KEY (Country) REFERENCES Country(Code));
```

Objekte werden unter Verwendung des Objektkonstruktors `<object-datatype>` in Objekttabellen eingefügt.

Beispiel 24 (Objekt-Tabelle *City* (Forts.)) Die Objekttabelle *City_ObjTab* wird wie folgt aus der Tupeltabelle *City* (z.B. mit allen deutschen Städten, deren geographische Koordinaten bekannt sind) gefüllt:

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

Will man ein selektiertes Zeilenobjekt *als Ganzes* ansprechen (etwa für einen Vergleich oder in einer `ORDER BY`-Klausel), so muss man `VALUE (<var>)` verwenden:

[1] Dies ist nicht so richtig gut, da *schwache Entitätstypen* eine Schlüsselattribut von einem anderen Entitätstyp importieren. Diese Beziehung wird i.a. durch Referenzen modelliert.

so, if you have a scoped ref – you can in fact index uniquely the columns. For some unknown reason to me – they will not let you create a primary key (simply a `NOT NULL` constraint plus a unique index), nor a `UNIQUE` constraint (simply a unique index). I cannot figure out why not. But anyway, except for losing the fact that name, a_ref, b_ref is a primary key in the data dictionary – the above lets you accomplish what you want. You cannot index an unscoped ref (why? good question – i don't know) and thats documented in the server concepts manual (page 11-8) □

Beispiel 25 (Objekt-Tabelle *City* (Forts.)) Gibt man `VALUE (<var>)` aus, erhält man die bekannte Darstellung mit Hilfe des Objektkonstruktors:

```
SELECT VALUE(cty)
FROM City_ObjTab cty;
```

VALUE(Cty)(Name, Province, Country, Population, Coordinates(Longitude, Latitude))
City_Type('Berlin', 'Berlin', 'D', 3472009, GeoCoord(13, 52))
City_Type('Bonn', 'Nordrhein-Westfalen', 'D', 293072, GeoCoord(8, 50))
City_Type('Stuttgart', 'Baden-Wuerttemberg', 'D', 588482, GeoCoord(9, 49))
⋮

□

`VALUE` kann ebenfalls verwendet werden, um in der obigen Anfrage zwei Objekte auf Gleichheit zu testen, oder um ein Objekt als Argument an eine Methode zu geben. Im Beispiel kann man die Distanz zwischen je zwei Städten entweder über die Methode *Distance* des Attributs *coordinates* oder direkt über die Methode *Distance* des Objekttyps *City* berechnen. Wieder werden Objektvariablen `cty1` und `cty2` verwendet, um die Zugriffspfade eindeutig zu machen:

```
SELECT cty1.Name, cty2.Name, cty1.coordinates.Distance(cty2.coordinates)
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);

SELECT cty1.Name, cty2.Name, cty1.Distance(VALUE(cty2))
FROM City_ObjTab cty1, City_ObjTab cty2
WHERE NOT VALUE(cty1) = VALUE(cty2);
```

Wird ein Objekt mit einem `SELECT INTO`-Statement einer PL/SQL-Variablen zugewiesen, wird ebenfalls `VALUE` verwendet.

```
SELECT VALUE(<var>) INTO <PL/SQL-Variable>
FROM <tabelle> <var>
WHERE ... ;
```

15.4 Objektreferenzen

Als Datentyp für Attribute sind neben den bereits bekannten skalaren Typen, Collection-Typen und Objekttypen auch *Referenzen* auf Objekte möglich. Eine entsprechende Attributdeklaration ist dann von der Form

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

Dabei wird ein *Objekttyp* als Ziel der Referenz angegeben. Wenn ein referenzierter Objekttyp in verschiedenen Tabellen vorkommt, wird damit das Ziel der Referenz hier nicht auf eine bestimmte Tabelle beschränkt. Einschränkungen dieser Art werden bei der Deklaration der entsprechenden Tabelle als Spalten- oder Tabellenconstraints mit `SCOPE` angegeben:

- als Spaltenconstraint (nur bei Tupeltabellen):
  `<ref-attr> REF <object-datatype> SCOPE IS <object-table>`
- als Tabellenconstraint:
  `SCOPE FOR (<ref-attr>) IS <object-table>`

Hierbei ist zu berücksichtigen, dass nur Objekte, die eine OID besitzen – also Zeilenobjekte einer Objekttabelle – referenziert werden können. Die OID eines zu referenzierenden Objektes wird folgendermaßen selektiert:

```
SELECT ..., REF (<var>), ...
FROM <tabelle> <var>
WHERE ... ;
```

Beispiel 26 (Objekttyp *Organization*) Politische Organisationen können als Objekte modelliert werden: Eine Organisation besitzt einen Namen, eine Abkürzung, einen Sitz in einer bestimmten Stadt (die in Beispiel 23 ebenfalls als Objekt modelliert wird und damit als *Referenzattribut* gespeichert werden kann) und hat eine Liste von Mitgliedern (hier bezeichnet durch die jeweiligen Landeskürzel).

```
CREATE TYPE Member_Type AS OBJECT
 (Country VARCHAR2(4),
  Type VARCHAR2(30));
/
CREATE TYPE Member_List_Type AS
 TABLE OF Member_Type;
/
CREATE OR REPLACE TYPE Organization_Type AS OBJECT
 (Name VARCHAR2(80),
  Abbrev VARCHAR2(12),
  Members Member_List_Type,
  Established DATE,
  seated_in REF City_Type,
  MEMBER FUNCTION is_member (the_country IN VARCHAR2) RETURN VARCHAR2,
  MEMBER FUNCTION people RETURN NUMBER,
  MEMBER FUNCTION number_of_members RETURN NUMBER,
  MEMBER PROCEDURE add_member
       (the_country IN VARCHAR2, the_type IN VARCHAR2),
  PRAGMA RESTRICT_REFERENCES (is_member,WNPS,WNDS),
  PRAGMA RESTRICT_REFERENCES (people,WNDS,WNPS);
  PRAGMA RESTRICT_REFERENCES (number_of_members,WNDS,WNPS));
/
```

Die entsprechende Tabellendefinition sieht dann folgendermaßen aus:

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

Das Einfügen in Objekttabellen geschieht nun wie üblich unter Verwendung des Objektkonstruktors:

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

Die geschachtelte Tabelle *Members* wird ebenfalls wie bereits bekannt mit Werten gefüllt:

```
INSERT INTO
```

```
THE(SELECT Members
    FROM Organization_ObjTab
    WHERE Abbrev='EU')
(SELECT Country, Type
 FROM is_member
 WHERE Organization = 'EU');
```

Will man eine *Referenz* (also eine OID) auf ein Objekt, das durch eine Objektvariable beschrieben wird, verarbeiten, muss man `SELECT REF(<var>)` angeben (Man beachte, dass `<var>` selber also weder das Objekt, noch eine Referenz ist!). `SELECT REF(VALUE(<var>))` ergibt keine OID, sondern eine Fehlermeldung.

Beispiel 27 (Objekttyp *Organization* (Forts.)) Um das Referenzattribut *seated_in* auf eine Stadt zeigen zu lassen, muss die entsprechende Referenz in *City_ObjTab* gesucht werden:

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

Die Selektion von Wertattributen erfolgt ebenfalls wie bei gewöhnlichen relationalen Tabellen – Aliasing muss nur zur Selektion von Methoden oder objektwertigen Attributen verwendet werden:

```
SELECT Name, Abbrev, Members, seated_in
FROM Organization_ObjTab;
```

gibt *Members* in der bekannten Darstellung mit Hilfe des Objektkonstruktors aus:

Name	**Abbrev**	**Members**
European Community	EU	Member_List_Type(...)

Bei Referenzattributen bekommt man auf ein gewöhnliches `SELECT <ref-attr-name>` (erwartungsgemäß) eine Objekt-ID geliefert (die zum internen Gebrauch innerhalb von SQL oft benötigt wird, jedoch in der Benutzerausgabe nicht erwünscht ist):

```
SELECT Name, Abbrev, seated_in
FROM Organization_ObjTab;
```

Name	**Abbrev**	**seated_in**
European Community	EU	`<oid>`

Hat man eine solche Referenz gegeben, erhält man mit `DEREF(<oid>)` das zugehörige Objekt:

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

Abbrev	**seated_in**
EU	City_Type('Brussels', 'Brabant', 'B', 951580, GeoCoord(4, 51))

Die einzelnen Attribute eines referenzierten Objekts kann man durch *Pfadausdrücke* der Form `SELECT <ref-attr-name>.<attr-name>` adressieren ("*navigierender Zugriff*"). Dabei muss Aliasing mit einer Variablen verwendet werden um den Pfadausdruck eindeutig zu machen:

```
SELECT Abbrev, org.seated_in.name
FROM Organization_ObjTab org;
```

Abbrev	seated_in.Name
EU	Brussels

`DEREF` wird insbesondere in PL/SQL-Routinen (z.B. `SELECT DEREF(...) INTO ...`) und bei der Definition von Objekt-Views(vgl. Abschnitt 16) verwendet.

Bemerkung 1 Mit `REF` und `DEREF` lässt sich `VALUE` ersetzen:

```
SELECT VALUE(cty) FROM City_ObjTab cty;
```

und

```
SELECT DEREF(REF(cty)) FROM City_ObjTab cty;
```

sind äquivalent. □

Zyklische Referenzen. Häufig kommen Objekttypen vor, die sich gegenseitig referenzieren sollen: Eine Stadt liegt in einem Land, andererseits ist die Hauptstadt eines Landes wieder eine Stadt. In diesem Fall benötigt die Deklaration jedes Datentypen bereits die Definition des anderen. Zu diesem Zweck erlaubt ORACLE die Definition von *unvollständigen* Typen (auch als *Forward*-Deklaration bekannt) durch

```
CREATE TYPE <name>;
/
```

Eine solche unvollständige Deklaration kann in `REF`-Deklarationen verwendet werden. Sie wird später durch eine komplette Typdeklaration ersetzt.

Beim Füllen dieser Tabellen muss man ebenfalls schrittweise vorgehen: Da jedes Tupel einer Tabelle eine Referenz auf ein Tupel einer anderen Tabelle enthält, muss man auch die Objekte zuerst "unvollständig" erzeugen, d.h., Objektfragmente anlegen, die bereits *eindeutig identifizierbar sind*, aber *keine Referenzen auf noch nicht angelegte Objekte* enthalten. Danach kann man mit `UPDATE` schrittweise die Tupel um Referenzen ergänzen.

Beispiel 28 (Zyklische Referenzen) Zwischen Städten, Provinzen, und Ländern bestehen zyklische Referenzen.

```
CREATE OR REPLACE TYPE City_Type
/
CREATE OR REPLACE TYPE Country_Type AS OBJECT
(Name VARCHAR2(32),
 Code VARCHAR2(4),
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
CREATE OR REPLACE TYPE Province_Type AS OBJECT
(Name VARCHAR2(32),
```

```
 Country REF Country_Type,
 Capital REF City_Type,
 Area NUMBER,
 Population NUMBER);
/
CREATE OR REPLACE TYPE City_Type AS OBJECT
 (Name VARCHAR2(35),
  Province REF Province_Type,
  Country REF Country_Type,
  Population NUMBER,
  Coordinates GeoCoord);
/
```

Beim der Tabellen mit Daten muss ebenfalls die Reihenfolge beachtet werden. Sind die zu referenzierenden Objekte beim Erzeugen des referenzierenden Objektes noch nicht bekannt, muss ein *NULL*wert anstelle der Referenz bei der Erzeugung angegeben werden, der später dann durch die Referenz ersetzt wird.

Wenn Objekttypen, die sich gegenseitig zyklisch referenzieren, gelöscht werden, muss man zumindest einmal ein Objekttyp löschen, der noch referenziert wird. Dazu muss

```
DROP TYPE <type> FORCE;
```

verwendet werden (vgl. Seite 5.2).

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

Referentielle Integrität. Die Verwendung von OID's garantiert, dass auch bei Änderung der Schlüsselattribute eines Objekts die referentielle Integrität gewahrt bleibt – im Gegensatz zu dem relationalen Konzept mit Fremdschlüsseln, bei dem explizit integritätserhaltende Maßnahmen durch `ON UPDATE CASCADE/RESTRICT` ergriffen werden müssen.

Durch Löschen von Objekten können dennoch *dangling references* entstehen; der Fall `ON DELETE CASCADE` wird also nicht direkt abgedeckt. Dafür lassen sich *entstandene* dangling references durch

```
WHERE <ref-attribute> IS DANGLING
```

überprüfen (vgl. `WHERE ... IS NULL`). Dies kann z.B. in einem `AFTER`-Trigger der Form

```
UPDATE <table>
  SET <attr> = NULL
  WHERE <attr> IS DANGLING;
```

genutzt werden.

15.5 Methoden: Funktionen und Prozeduren

Der *Type Body* enthält die Implementierungen der Methoden in PL/SQL (als zukünftige Erweiterung ist die direkte Einbindung von Java vorgesehen). Dazu ist PL/SQL ebenfalls an einigen Stellen an

geschachtelte Tabellen und objektorientierte Features angepasst worden. Allerdings wird Navigation mit Pfadausdrücken in PL/SQL *nicht* unterstützt.

Beispiel 29 (Pfadausdrücke) Für jede Organisation soll die Bevölkerungszahl des Landes, in der ihr Sitz liegt, ausgegeben werden.

In SQL ist das einfach:

```
SELECT org.seated_in.country.population
FROM Organization_ObjTab org;
```

Ein entsprechende Zuweisung in PL/SQL

```
-- the_org enth"alt ein Objekt vom Typ Organization_Type
pop := the_org.seated_in.country.population;
```

ist nicht erlaubt: Beim Navigieren mit Pfadausdrücken werden effektiv Datenbankzugriffe ausgeführt. Diese können in PL/SQL nur in eingebetteten SQL-Ausdrücken vorgenommen werden. Solche Operationen müssen durch entsprechende Datenbankzugriffe ersetzt werden:

```
-- the_org enth"alt ein Objekt vom Typ Organization_Type
SELECT org.seated_in.country.population
INTO pop
FROM Organization_ObjTab org
WHERE org.abbrev = the_org.abbrev;
```

Jede `MEMBER METHOD` besitzt einen *impliziten* Parameter `SELF` (wie in objektorientierten Programmiersprachen üblich), der bei einem Methodenaufruf das jeweilige Host-Objekt referenziert: ruft man z.B. die Methode `is_member` des Objektes "NATO" auf, so ergibt `SELF` die das Objekt "NATO".

Weiterhin können die bereits in Abschnitt 14.6 beschriebenen Methoden, die ursprünglich für *Collections* (d.h. damals nur PL/SQL-Tabellen) definiert sind, aus PL/SQL auch auf geschachtelte Tabellen angewendet werden. Für ein tabellenwertiges Attribut `<attr-name>` gibt z.B. die Methode `<attr-name>.COUNT` die Anzahl der in der geschachtelten Tabelle enthaltenen Tupel an. Damit wird die geschachtelte Tabelle wie ein Array behandelt[2]. Diese Methoden sind allerdings nicht innerhalb in PL/SQL eingebetteter SQL-Statements – z.B. `SELECT <attr>.COUNT` – erlaubt.

Beispiel 30 (Objekttyp *Organization*: Type Body) Der `TYPE BODY` von *Organization_Type* sieht damit wie folgt aus:

```
CREATE OR REPLACE TYPE BODY Organization_Type IS
MEMBER FUNCTION is_member (the_country IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
  IF SELF.Members IS NULL OR SELF.Members.COUNT = 0 THEN RETURN 'no'; END IF;
  FOR i in  1 .. Members.COUNT
  LOOP
    IF the_country = Members(i).country
    THEN RETURN Members(i).type;
    END IF;
  END LOOP;
```

[2] nicht elegant, aber praktisch.

```
  RETURN 'no';
END;
MEMBER FUNCTION people RETURN NUMBER IS
p NUMBER;
BEGIN
  SELECT SUM(population) INTO p
  FROM Country ctry
  WHERE ctry.Code IN
    (SELECT Country
     FROM THE (SELECT Members
               FROM Organization_ObjTab org
               WHERE org.Abbrev = SELF.Abbrev));
  RETURN p;
END;
MEMBER FUNCTION number_of_members RETURN NUMBER
IS
BEGIN
  IF SELF.Members IS NULL THEN RETURN NULL; END IF;
  RETURN Members.COUNT;
END;
MEMBER PROCEDURE add_member
       (the_country IN VARCHAR2, the_type IN VARCHAR2) IS
BEGIN
  IF NOT SELF.is\_member(the\_country) = 'no' THEN RETURN;
     END IF; \\
  IF SELF.Members IS NULL THEN
     UPDATE Organization_ObjTab
     SET Members = Member_List_Type()
     WHERE Abbrev = SELF.Abbrev;
  END IF;
  INSERT INTO
  THE (SELECT Members
            FROM Organization_ObjTab org
            WHERE org.Abbrev = SELF.Abbrev)
  VALUES (the_country,the_type);
END;
END;
/
```

In *is_member* wäre die naheliegendste Idee gewesen, einen Cursor über `(THE) SELF.Members` zu definieren. Dies ist aber zumindest in Oracle 8.0 nicht möglich (wo würde syntaktisch das `THE` hingehören?). Stattdessen wird eine `FOR`-Schleife mit `Members.COUNT` verwendet. Zu beachten ist weiterhin, dass der Fall "*Members* ist `NULL`" bei *is_member* und *add_member* separat betrachtet werden muss.

In *people* und *add_member* würde man gerne das innere umständliche `FROM THE(SELECT ...)` durch `FROM SELF.Members` oder etwas ähnliches ersetzen. Dies ist (zumindest in Oracle 8.0) nicht möglich: Mit `SELF.Members` befindet man sich in PL/SQL, d.h. man müsste PL/SQL-Konstrukte verwenden um diese Tabelle zu bearbeiten – `SELECT` ist jedoch ein SQL-Konstrukt, das sich an Datenbank-Tabellen richtet. Hier fällt der immer noch bestehende Unterschied zwischen Datenbank und PL/SQL-

Umgebung störend auf[3]. □

Die Objekt-`MEMBER FUNCTIONS` können dann in SQL und PL/SQL wie Attribute durch `<object>.<function>(<ar` selektiert werden (im Falle einer parameterlosen Funktion muss `<object>.<function>()` angegeben werden). Bei einem Aufruf aus SQL ist `<object>` durch einen Pfadausdruck gegeben, wobei Aliasing verwendet werden muss um diesen eindeutig zu machen.

Beispiel 31 (Objekttyp *Organization*: Nutzung von `MEMBER FUNCTIONs`) Die folgende Anfrage gibt alle politischen Organisationen sowie die Art der Mitgliedschaft aus, in denen Deutschland Mitglied ist:

```
SELECT Name, org.is_member('D')
  FROM Organization_ObjTab org
  WHERE NOT org.is_member('D') = 'no';
```

Objekt-`MEMBER PROCEDURES` können nur aus PL/SQL mit `<objekt>.<procedure>(<argument-list>)` aufgerufen werden. Dazu werden freie Prozeduren (d.h., nicht als Objektmethoden an einen Typ/Objekt gebunden) in PL/SQL geschrieben, die es ermöglichen, auf die entsprechenden Objekttabellen zuzugreifen.

Beispiel 32 (Objekttyp *Organization*: Verwendung) Die folgende freie Prozedur erlaubt das Einfügen eines Mitglieds in ein politische Organisation. Dabei wird der Aufruf auf die Objektmethode *add_member* der ausgesuchten Organisation abgebildet. Ist die angegebene Organisation noch nicht vorhanden, so wird ein entsprechendes Objekt erzeugt:

```
CREATE OR REPLACE PROCEDURE make_member
  (the_org IN VARCHAR2, the_country IN VARCHAR2, the_type IN VARCHAR2) IS
  n NUMBER;
  c Organization_Type;
BEGIN
  SELECT COUNT(*) INTO n
    FROM Organization_ObjTab
    WHERE Abbrev = the_org;
  IF n = 0
  THEN
    INSERT INTO Organization_ObjTab
      VALUES(Organization_Type(NULL,the_org,Member_List_Type(),NULL,NULL));
  END IF;
  SELECT VALUE(org) INTO c
    FROM Organization_ObjTab org
    WHERE Abbrev = the_org;
  IF c.is_member(the_country)='no' THEN
     c.add_member(the_country,the_type);
  END IF;
END;
/
```

Mit den bisher in diesem Beispiel angegebenen `INSERT`-Statements wurde die EU in die neue Tabelle eingefügt. Mit den folgenden Befehlen werden die USA als *special member* in die EU eingetragen,

[3] dieser Unterschied ist aber bei embedded SQL in C noch viel störender.

außerdem wird eine noch nicht existierende Organisation erzeugt, deren einziges Mitglied die USA sind:

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

Damit kann der gesamte Datenbestand aus den relationalen Tabellen *Organization* und *is_member* in die Objekttabelle *Organization_ObjTab* übertragen werden:

```
INSERT INTO Organization_ObjTab
 (SELECT Organization_Type(Name,Abbreviation,NULL,Established,NULL)
  FROM Organization);

CREATE OR REPLACE PROCEDURE Insert_All_Members IS
BEGIN
 FOR the_membership IN
     (SELECT * FROM is_member)
 LOOP
     make_member(the_membership.organization,
                 the_membership.country,
                 the_membership.type);
 END LOOP;
END;
/

EXECUTE Insert_All_Members;

UPDATE Organization_ObjTab org
SET seated_in =
  (SELECT REF(cty)
   FROM City_ObjTab cty, Organization old
   WHERE org.Abbrev = old.Abbreviation
     AND cty.Name = old.City
     AND cty.Province = old.Province
     AND cty.Country = old.Country);
```

Als weiteres kann man eine freie Funktion *is_member_in* definieren, die testet, ob ein Land Mitglied in einer Organisation ist. Diese Prozedur verwendet natürlich die Objektmethode *is_member* der entsprechenden Organisation:

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

```
/
```

Beispiel: Die system-eigene Tabelle `DUAL` wird verwendet um das Ergebnis freier Funktionen ausgeben zu lassen.

```
SELECT is_member_in('EU','SL')
FROM DUAL;
```

is_member_in('EU','SL')
applicant

Bemerkung 2 Es ist (zumindest in ORACLE 8.0.x) nicht möglich, durch Navigation mit Pfadausdrücken Tabelleninhalte zu verändern:

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

15.6 ORDER- und MAP-Methoden

Objekttypen besitzen im Gegensatz zu den Datentypen `NUMBER` und `VARCHAR` keine inhärente Ordnung. Um eine Ordnung auf Objekten eines Typs zu definieren, können dessen funktionale Methoden verwendet werden. In ORACLE8 kann dazu für jeden Objekttyp eine Funktion als `MAP FUNCTION` oder `ORDER FUNCTION` ausgezeichnet werden.

- Eine `MAP`-Funktion besitzt keine Parameter und bildet jedes Objekt auf eine Zahl ab. Damit wird eine lineare Ordnung auf dem Objekttyp definiert, die sowohl für Vergleiche `<`, `>` und `BETWEEN`, als auch für `ORDER BY` verwendet werden kann.
  Einer `MAP`-Funktion entspricht dem mathematischen Konzept einer *Betragsfunktion*.
- Eine `ORDER`-Funktion besitzt *ein* Argument desselben Datentyps das ein reiner `IN`-Parameter ist (deshalb die Deklaration `<func-name>(<var> <type>)`) und mit dem Hostobjekt verglichen wird.
  Eine `ORDER`-Funktionen entspricht einem *direkten Vergleich* zweier Werte/Objekte. Damit sind `ORDER`-Funktionen für Vergleiche `<`, `>` geeignet, im allgemeinen aber nicht unbedingt für Sortierung.
- `MAP`- und `ORDER`-Funktionen erfordern `PRAGMA RESTRICT_REFERENCES (<name>,WNDS,WNPS,RNPS,RNDS)`, d.h. sie dürfen *keinen Datenbankzugriff* enthalten.

MAP-Methoden.

Beispiel 33 (Objekttyp *GeoCoord*: MAP-Methode) Der Objekttyp *City_Type* verwendet den Objekttyp *GeoCoord* zur Speicherung geographischer Koordinaten. Auf *GeoCoord* ist eine `MAP`-Methode definiert, die nun verwendet werden kann, um Städte nach ihrer Entfernung von Greenwich zu ordnen:

```
SELECT Name, cty.coordinates.longitude, cty.coordinates.latitude
FROM City_ObjTab cty
WHERE NOT coordinates IS NULL
ORDER BY coordinates;
```

Andererseits sind viele Operationen in `MAP`-Methoden nicht erlaubt: Wie bereits beschrieben, dürfen diese keine Anfragen an die Datenbank enthalten. Weiterhin scheinen – zumindest in ORACLE8 – keine built-in Methoden von geschachtelten Tabellen verwendbar zu sein (siehe folgendes Beispiel):

Beispiel 34 (Objekttyp *Organization*: MAP-Methode)

- In *Organization_Type* is es nicht möglich, die Methode *People* als MAP-Methode zu verwenden, da diese eine Datenbankanfrage enthält.
- Die Methode *number_of_members* kann – zumindest in Oracle 8.0.x – ebenfalls nicht als MAP-Methode verwendet werden: Wird die Deklaration von *Organization_Type* in `MAP MEMBER FUNCTION number_of_members RETURN NUMBER` geändert, so ergibt

  ```
  SELECT * FROM Organization_ObjTab org ORDER BY VALUE(org);
  ```

  eine Fehlermeldung, dass der Zugriff auf geschachtelte Tabellen hier nicht erlaubt sei. □

ORDER-Methoden. ORDER-Methoden *vergleichen* jeweils SELF mit einem anderen Objekt desselben Typs, das formal als Parameter angegeben wird. Als Ergebnis muss -1 (SELF < Parameter), 0 (Gleichheit) oder 1 (SELF > Parameter) zurückgegeben werden. Damit lassen sich kompliziertere (Halb-)Ordnungen definieren.

Wird bei einer Anfrage ein ORDER BY angegeben, werden die Ausgabeobjekte paarweise verglichen und entsprechend der ORDER-Methode geordnet.[4].

Ein Beispiel hierfür ist etwa die Erstellung der Fußball-Bundesligatabelle: Ein Verein wird vor einem anderen platziert, wenn er mehr Punkte hat. Bei Punktgleichheit entscheidet die Tordifferenz. Ist auch diese dieselbe, so entscheidet die Anzahl der geschossenen Tore (vgl. Aufgabe).

Verwendung. ORDER/MAP-Methoden definieren eine Ordnung auf Objekten eines Typs, und können damit auf zwei Arten verwendet werden:

- Ordnen nach dem Attributwert einer objektwertigen Spalte:

  ```
  CREATE TYPE <type> AS OBJECT
    (  ...
       MAP/ORDER MEMBER METHOD <method> ...);

  CREATE TABLE <table>
    (  ...,  <attr> <type>,
       ...);

  SELECT * FROM <table>
  ORDER BY <attr>;
  ```

- Die Elemente einer Objekttabelle sollen nach ihrem *eigenen* Objektwert geordnet werden:

  ```
  CREATE TYPE <type> AS OBJECT
    (  ...
       MAP/ORDER MEMBER METHOD <method> ...);

  CREATE TABLE <table> OF <type>;

  SELECT * FROM <table> <alias>
  ORDER BY value(<alias>);
  ```

[4] Da muss man mal eine nicht-zyklenfreie Relation ausprobieren!

15.7 Objekttypen: Indexe, Zugriffsrechte und Änderungen

Indexe auf Objektattributen. Indexe (vgl. Abschnitt 13) können auch auf Objektattributen erstellt werden:

```
CREATE INDEX <name>
  ON <object-table-name>.<attr>[.<attr>]*;
```

Hierbei trägt die eckige Klammer der Tatsache Rechnung, dass evtl. ein Index über ein Teilattribut eines komplexen Attributes erstellt werden soll. Indexe können *nicht* über komplexen Attributen (also Attributen eines selbstdefinierten Typs) erstellt werden: Es ist z.B. nicht möglich, in Beispiel 6 einen Index über das Attribut *coordinates*, bestehend aus *longitude* und *latitude* zu erstellen:[5]

```
CREATE INDEX city_index ON City_ObjTab(coordinates);          Nicht erlaubt
CREATE INDEX city_index ON City_ObjTab(coordinates.Longitude,coordinates.Latitude);
  --     erlaubt
```

Indexe über Referenzattributen sind (zumindest in Oracle 8.0.3) nur erlaubt, wenn für das Referenzattribut ein `SCOPE IS`-Constraint angegeben ist.

Zugriffsrechte auf Objekte. Rechte an Objekttypen werden durch `GRANT EXECUTE ON <Object-datatype> TO ...` vergeben – die zugrundeliegende Idee ist hier, dass bei der Benutzung eines Datentyps vor allem die Methoden (u.a. die entsprechende Konstruktormethode) im Vordergrund stehen.

Änderungen von Objekttypen. Durch die Verwendung von Objekttypen und Referenzattributen entsteht ein ähnliches Netz von Referenzen wie bei der Definition von `FOREIGN KEY`-Definitionen. Für `FOREIGN KEY`-Definitionen wurde bereits gezeigt, dass Änderungen am Datenbestand durch referentielle Integritätsbedingungen ziemlich kompliziert werden können – Bedingungen müssen aus- und wieder eingeschaltet werden. Durch Benutzung von Objekttypen in anderen Objekttypen und Tabellen entsteht ein ähnliches Netz von Abhängigkeiten. In dieser Situation werden Änderungen an Objekttypen in Oracle 8.0.x sehr restriktiv behandelt.

Insbesondere ist es meistens nicht möglich, einem irgendwo verwendeten Objekttyp ein Attribut (oder auch nur eine Methode!) hinzuzufügen: `CREATE OR REPLACE TYPE` und `ALTER TYPE` sind (zumindest in Oracle 8.0.x) nicht erlaubt, wenn der Objekttyp irgendwo verwendet wird.

> *In conclusion, carefully plan the object types for your database so that you get things right the first time. Then keep your fingers crossed and hope that things do not change once you have everything up and running (*Oracle*8: Architecture).*

Ein erstes Fazit

- Daten*haltung* in einem objektorientierten Schema kann bereits bei kleinen Änderungen sehr problematisch werden.
- Methoden sowie freie Prozeduren und Funktionen ermöglichen eine anwendungsorientierte (nichtrelationale) Repräsentation der Daten.
- Die Integration von anwendungsspezifischer Funktionalität in die Datenbank wird durch Objektmethoden vereinfacht.

⇒ Datenhaltung in einem relationalen Basisschema, Nutzung unter einem objektorientierten Schema.

[5] Fehlermeldung in solchen Fällen: *Index für Spalte mit Datentyp ADT kann nicht erstellt werden*; ADT bedeutet *Abstrakter Datentyp*.

16 OBJECT-VIEWS IN Oracle8

Object Views dienen genauso wie relationale Views dazu, eine andere Darstellung der in der Datenbank enthaltenen Daten zu ermöglichen. Durch die Definition von Objektdatentypen, die evtl. nur in den Views verwendet werden, ist es möglich, den Benutzern maßgeschneiderte Object-Views mit sehr weitgehender Funktionalität anzubieten.

Object Views haben verschiedene Anwendungen:

Legacy-Datenbanken Häufig müssen bestehende Datenbanken in ein "modernes" objektorientiertes Modell eingebunden werden. In diesem Fall kann man über die relationale Ebene eine Ebene von *Objekt-Views* legen, die nach außen hin ein objektorientiertes Datenbanksystem erscheinen lassen. In diesem Fall wird auch von *Objekt-Abstraktionen* gesprochen.

Effizienz + Benutzerfreundlichkeit: Selbst bei einem Neuentwurf zeigt sich an vielen Stellen, dass eine relationale Repräsentation am effizientesten ist:

- Geschachtelte Tabellen werden intern als separate Tabellen gespeichert.
- $n : m$-Beziehungen könnten "objektorientiert" durch gegenseitige geschachtelte Tabellen modelliert werden, was aber sehr ineffizient ist.

Auch hier ist es sinnvoll, ein – zumindest teilweise – relationales Basisschema zu definieren, und darauf Object-Views zu erstellen.

Einfache Modifizierbarkeit: Da `CREATE OR REPLACE TYPE` und `ALTER TYPE` nur sehr eingeschränkt verwendbar sind, kann ein objektorientiertes Datenbankschema nur mit sehr hohem Aufwand verändert werden. Auch hier kann es sinnvoll sein, solche Veränderungen durch die Definition geeigneter Object-Views abzufangen.

In der ORACLE-Community wird daher häufig empfohlen, Object Views mit geschachtelten Tabellen, Referenzen etc. *auf Basis eines relationalen Grundschemas* zu verwenden. In diesem Fall können das Grundschema und das externe Schema unabhängig voneinander (durch jeweilige Anpassung der Methodenimplementierung) geändert werden.

In jedem Fall führt der Benutzer seine Änderungen auf dem durch die Objektviews gegebenen externen Schema durch. Direkte Änderungen durch `INSERT`, `UPDATE` und `DELETE`, werden durch `INSTEAD OF`-Trigger (vgl. Abschnitt 14.7.2) auf das darunterliegende Schema umgesetzt. Viel besser ist es jedoch, den Benutzer erst gar keine solchen Statements an das View stellen zu lassen, sondern die entsprechende Funktionalität durch Methoden der Objekttypen zur Verfügung zu stellen, die die Änderungen direkt auf den zugrundeliegenden Basistabellen ausführen.

16.1 Anlegen von Objektviews

Objektrelationale Views. Relativ einfache objektrelationale Views auf Basis von relationalen oder objektorientierten Basistabellen lassen sich auch ohne Objekttypen definieren (wobei man natürlich dann keine Methoden hat). Die Syntax ist prinzipiell dieselbe wie für relationale Views:

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
  <select-clause>;
```

Dabei sind in der SELECT-clause jetzt zusätzlich Konstruktormethoden für Objekte und geschachtelte Tabellen erlaubt. Für die Erzeugung geschachtelter Tabellen für Object Views werden die Befehle CAST und MULTISET (vgl. Abschnitt 5.2/Beispiel 7) verwendet.

Beispiel 35 (Objektrelationales View über Relationalem Schema) Um für jedem Fluss auch direkt auf seinen Nebenflüsse zugreifen zu können, wird ein entsprechendes View definiert:

```
CREATE TYPE River_List_Entry AS OBJECT
 (name VARCHAR2(20),
  length NUMBER);
/
CREATE TYPE River_List AS
 TABLE OF River_List_Entry;
/

CREATE OR REPLACE VIEW River_V
  (Name, Length, Tributary_Rivers)
AS SELECT
   Name, Length,
   CAST(MULTISET(SELECT Name,Length FROM River
                 WHERE River = A.Name) AS River_List)
   FROM River A;

SELECT * FROM River_V;
```

Objektviews. Objektviews enthalten wie Objekttabellen Zeilenobjekte, d.h. hier werden neue Objekte *definiert*. Bei der Definition von Objektviews wird durch WITH OBJECT OID <attr-list> angegeben, aus welchen Attributen die Objekt-ID berechnet wird. Auch hier werden CAST und MULTISET verwendet um Views mit geschachtelten Tabellen zu definieren.

```
CREATE [OR REPLACE] VIEW <name> OF <type>
  WITH OBJECT OID (<attr-list>)
  AS <select-statement>;
```

Hierbei ist zu beachten, dass in <select-statement> *kein* Objektkonstruktor verwendet wird:

Beispiel 36 (Object Views: *Country*)

```
CREATE OR REPLACE TYPE Country_Type AS OBJECT
 (Name       VARCHAR2(32),
  Code       VARCHAR2(4),
  Capital    REF City_Type,
  Area       NUMBER,
  Population NUMBER);
/

CREATE OR REPLACE VIEW Country_ObjV OF Country_Type
```

```
WITH OBJECT OID (Code)
AS
SELECT Country.Name, Country.Code, REF(cty), Area, Country.Population
FROM Country, City_ObjTab cty
WHERE cty.Name = Country.Capital
  AND cty.Province = Country.Province
  AND cty.Country = Country.Code;

SELECT Name, Code, c.capital.name, Area, Population
FROM Country_ObjV c;
```

Beispiel 37 (Object Views: Was nicht geht) Leider scheint das Konzept nicht absolut ausgereift zu sein: Will man ein Object View auf Basis von *Organization_ObjTab* definieren, stellt man fest, dass dieses offensichtlich weder die geschachtelte Tabelle noch das Ergebnis einer funktionalen Methode der zugrundeliegenden Tabellen enthalten darf:

```
CREATE OR REPLACE TYPE Organization_Ext_Type AS OBJECT
 (Name VARCHAR2(80),
  Abbrev VARCHAR2(12),
  Members Member_List_Type,  -- nicht erlaubt
  established DATE,
  seated_in REF City_Type,
  number_of_people NUMBER);  -- nicht erlaubt
/
CREATE OR REPLACE VIEW Organization_ObjV OF Organization_Ext_Type
AS
SELECT Name, Abbrev, Members, established, seated_in, org.people()
 FROM Organization_ObjTab org;

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

Beide angegebenen Attribute sind auch einzeln nicht erlaubt. □

16.2 Fazit

+ Kompatibilität mit den grundlegenden Konzepten von Oracle 7. U.a. können Fremdschlüsselbedingungen von Objekttabellen zu relationalen Tabellen definiert werden.
+ Durch *Object Views* kann basierend auf einem Grundschema ein Zweitschema definiert werden, mit dem sich eine Datenbank den Benutzern präsentiert. Durch entsprechende Methoden und `INSTEAD OF`-Trigger können Benutzer-Interaktionen auf das Grundschema umgesetzt werden.
– Flexibilität/Ausgereiftheit: Typen können nicht mehr verändert/erweitert werden, wenn sie in irgendeiner Form benutzt werden. Damit sind (inkrementelle!) Anpassungen des Schemas nicht möglich.

Teil V

Kombination von SQL mit anderen Programmiersprachen

17 EINBETTUNG VON SQL IN HÖHERE PROGRAMMIERSPRACHEN

Neben der Erweiterung von SQL um prozedurale Konzepte (siehe Abschnitt 14) kann SQL auch umgekehrt in eine bestehende prozedurale höhere Programmiersprache wie C, C++, Ada, FORTRAN, PASCAL, COBOL, etc. eingebunden werden. Diese Kopplungsart von Programmier- und Datenbanksprache bezeichnet man als *Embedded SQL*. Die (höhere) Programmiersprache wird dabei als *Hostsprache* bezeichnet. Eine weitere Kopplungsart stellt die Erweiterung der Programmiersprache um Datenbankkonstrukte dar. Dieser Ansatz wird bei persistenten Programmiersprachen (z.B. ObjectStore, POET) und Datenbank-Programmiersprachen (z.B. Pascal/R) verfolgt.

Bei der Einbettung von SQL in eine bestehende Programmiersprache – etwa C – werden SQL-Anweisungen durch Schlüsselwörter, z.B. `EXEC SQL`, eingeleitet und prinzipiell wie Anweisungen der Programmiersprache behandelt.

Dabei ergibt sich aufgrund der unterschiedlichen Typsysteme und Paradigmen ein *impedance mismatch* (Missverhältnis): Den Tupelmengen in SQL steht in der Regel kein vergleichbarer Datentyp in der Programmiersprache gegenüber. Des weiteren sind Programmiersprachen in der Regel imperativ, während SQL deklarativ ist. Dieser Konflikt wird in der Praxis dadurch gelöst, dass die Tupel bzw. Attribute auf die Datentypen der Hostsprache abgebildet werden, und Tupelmengen iterativ durch *Cursore* verarbeitet werden.

Bei der Kombination von SQL mit objektorientierten Programmiersprachen ist der konzeptuelle *impedance mismatch* noch größer. Aus dieser Situation entstanden die Konzepte von ODBC (*Open Database Connectivity*) in der Microsoft-Welt, sowie JDBC (*Java Database Connectivity*, vgl. Abschnitt 18), die prinzipiell eine Weiterentwicklung der Embedded-Idee darstellen.

17.1 Embedded-SQL in C

In diesem Abschnitt wird die Einbettung von SQL-Anweisungen in C beschrieben.[1]

Bei der Einbettung von SQL in C – werden SQL-Anweisungen durch das Schlüsselwort `EXEC SQL` eingeleitet und prinzipiell wie Anweisungen der Programmiersprache behandelt.

Ein solches C-Programm `<name>.pc` mit Embedded-SQL-Statements wird mit Hilfe eines Precompilers in ein C-Programm `<name>.c` transformiert, wobei die `EXEC SQL`-Statements in Aufrufe einer runtime library übersetzt werden. Das C-Programm `<name>.c` wird dann mit dem üblichen C-Compiler weiter übersetzt (siehe Abbildung 17.1).

Die dabei verwendeten SQL-Statements können entweder fest vorgegeben (statisch), oder im Zuge des

[1] In diesem Abschnitt wird dazu der ORACLE Pro* C/C++ Precompiler Version 2.1 verwendet. Die beschriebene Vorgehensweise zur Einbettung von SQL-Anweisungen lässt sich auch auf andere Programmiersprachen übertragen.

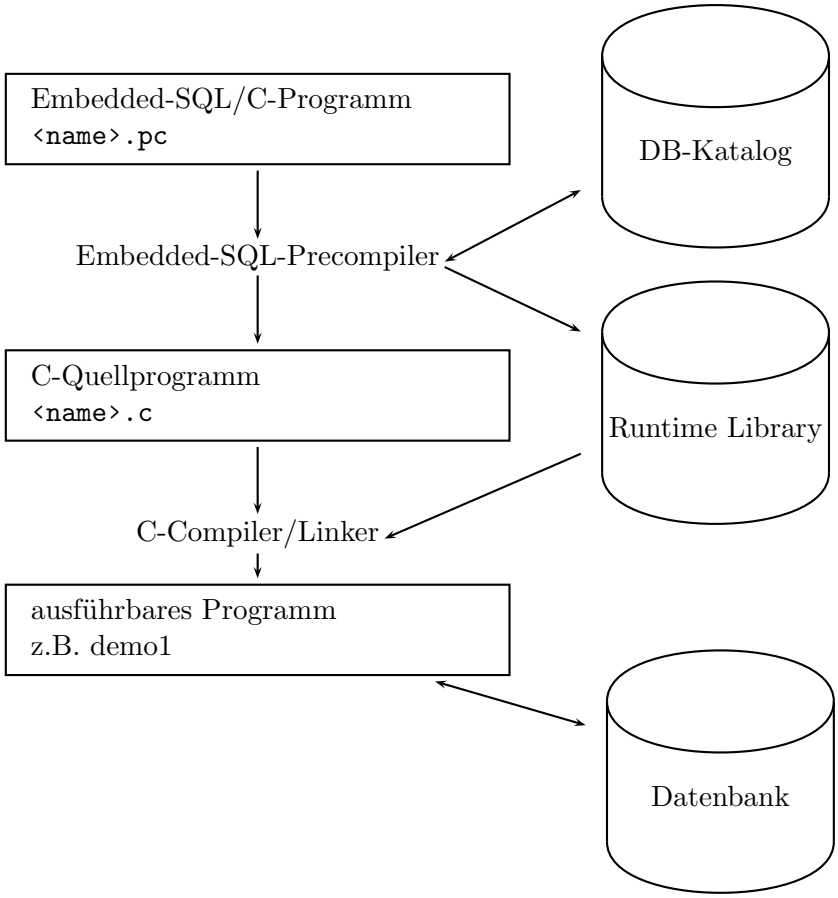


Abbildung 17.1: Entwicklungsschritte von Embedded-SQL Anwendungen

Programmablaufs als Strings zusammengesetzt werden (dynamische SQL-Anweisungen). Hier stehen statische SQL-Anweisungen im Vordergrund.

Verbindung zur Datenbank. Im Gegensatz zu SQL*plus und PL/SQL, wo die Verbindung zur Datenbank automatisch gegeben ist, muss in einer Embedded-Anwendung zuerst eine Verbindung zu einer Datenbank hergestellt werden. Die geschieht mit dem `CONNECT`-Statement:

```
EXEC SQL CONNECT :username IDENTIFIED BY :passwd;
```

wobei `username` und `passwd` Hostvariablen (siehe nächsten Abschnitt) vom Typ `CHAR` bzw. `VARCHAR` sind. Die Verwendung von Hostvariablen für das Einloggen kann *nicht* durch direkte Verwendung von Strings in der `CONNECT` Anweisung umgangen werden, eine Anweisung der Art

```
EXEC SQL CONNECT ‘‘dummy’’ IDENTIFIED BY ‘‘dummypasswd’’; /* nicht erlaubt */
```

schlägt daher fehl.

Alternativ kann das Einloggen auch in der Form

```
EXEC SQL CONNECT :uid;
```

erfolgen, wobei `uid` dann die Benutzerkennung, einen slash (“`/`”) und das Passwort enthält, etwa `"dummy / dummypasswd"`. Wenn der Betriebssystemaccount mit dem Datenbankaccount überein-

stimmt (wie im Praktikum), kann dabei "/" statt dem Benutzernamen verwendet werden.

Hostvariablen. Die Kommunikation zwischen der Datenbank und dem Anwendungsprogramm findet über *Hostvariablen*, d.h. Variablen der umgebenden Programmiersprache statt. Diese können sowohl in den Anweisungen der Hostsprache als auch in den embedded-SQL-Anweisungen stehen. Innerhalb eines SQL-Statements wird einer Hostvariable immer ein Doppelpunkt (":") vorangestellt (nicht jedoch in der Hostsprache), um sie von Datenbankobjekten abzuheben. Dabei dürfen Hostvariablen in statischen SQL-Anweisungen nicht an Stelle von Datenbankobjekten (Tabellen, Views, etc.) verwendet werden. Hostvariablen, die in der `INTO`-Klausel eines `SELECT`- oder `FETCH`-Statements auftreten, sind *Output*-Variablen (bzgl. des SQL-Statements). Alle anderen Hostvariablen in einem SQL-Statement sind hingegen *Input*-Variablen. Input-Variablen müssen initialisiert werden, bevor sie in einem SQL-Statement verwendet werden.

Hostvariablen können auch in einem `struct` zusammengefasst werden. Die einzelnen Komponenten des `struct` werden dann als Hostvariable betrachtet. Structs dürfen jedoch nicht geschachtelt werden. Bei der Verwendung einer Hostvariable ist darauf zu achten, dass sie zu dem Datentyp des Attributs kompatibel ist, auf das im SQL-Statement Bezug genommen wird. Die Beziehung zwischen C- und Oracle-Datentypen ist in Abbildung 17.2 beschrieben.

Oracle stellt für die Embedded-Programmierung in C einen C-Datentyp `VARCHAR` bereit, der an den internen Datentyp `VARCHAR2` angelehnt ist.

Indikatorvariablen. Jeder Hostvariablen kann eine Indikatorvariable zugeordnet werden, die den aktuellen Status der Hostvariablen anzeigt. Indikatorvariablen sind insbesondere für die Verarbeitung von Nullwerten von Bedeutung (hier äußert sich der Impedance Mismatch, dass herkömmliche Programmiersprachen keine Nullwerte kennen). Sie sind als `short` zu deklarieren; ihre Werte werden wie folgt interpretiert:

Indikatorvariablen für Output-Variablen:

- -1 : der Attributwert ist `NULL`, der Wert der Hostvariablen ist somit undefiniert.
- 0 : die Hostvariable enthält einen gültigen Attributwert.
- >0 : die Hostvariable enthält nur einen Teil des Spaltenwertes. Die Indikatorvariable gibt die ursprüngliche Länge des Spaltenwertes an.
- -2 : die Hostvariable enthält einen Teil des Spaltenwertes wobei dessen ursprüngliche Länge nicht bekannt ist.

Indikatorvariablen für Input-Variablen:

- -1 : unabhängig vom Wert der Hostvariable wird `NULL` in die betreffende Spalte eingefügt.
- $\geq$0 : der Wert der Hostvariable wird in die Spalte eingefügt.

Innerhalb eines SQL-Statements (nicht jedoch in der Hostsprache) wird der Indikatorvariable ein Doppelpunkt (":") vorangestellt, dabei muss sie direkt auf die entsprechende Hostvariable folgen. Zur Verdeutlichung kann zwischen den beiden Variablen auch das Schlüsselwort `INDICATOR` stehen (siehe demo1.pc). Wird für Hostvariablen ein `struct` verwendet, so lässt sich diesem auch ein entsprechender `struct` für Indikatorvariablen zuordnen.

Declare-Section. Host- und Indikatorvariablen werdem in der *Declare-Section* deklariert:

```
EXEC SQL BEGIN DECLARE SECTION;
   int population;        /* host variable */
```

C-Datentyp	ORACLE-Datentyp
char char[n] VARCHAR[n] int short long float double	VARCHAR2(n) CHAR(X)
int short long float double char char[n] VARCHAR[n]	NUMBER NUMBER(P,S)
char[n] VARCHAR[n]	DATE
char[n] VARCHAR[n]	LONG
unsigned char VARCHAR[n]	RAW(X)
unsigned char[n] VARCHAR[n]	LONG RAW

Abbildung 17.2: Kompatibilität zwischen C- und ORACLE-Datentypen

```
    short population\_ind;   /* indicator variable */
EXEC SQL END DECLARE SECTION;
```

Deklarationen werden nicht wie SQL-Anweisungen in einen Aufruf der runtime library übersetzt. Je nach Version bzw. Modus des Precompilers können Host- bzw. Indikatorvariablen auch als "gewöhnliche" C-Variablen ohne Declare-Section deklariert werden.

Cursore. Wenn die Anwortmenge zu einer SELECT-Anfrage mehrere Tupel enthält, die in einem Anwendungsprogramm verarbeitet werden sollen, geschieht dies über einen Cursor (oder alternativ ein Hostarray). Die Cursoroperationen sind im Prinzip dieselben wie in PL/SQL, sie sind allerdings hier keine "vollwertigen" Befehle, sondern müssen ebenfalls durch EXEC SQL eingeleitet werden.

Mit der Deklaration

```
EXEC SQL DECLARE <cursor-name> CURSOR FOR <select-statement>;
```

wird ein Cursor benannt und einer Anfrage zugeordnet.

Für den Umgang mit Cursoren stehen auch hier drei Operationen zur Verfügung:

- `OPEN <cursor-name>;`
  öffnet einen Cursor, indem die entsprechende Anfrage ausgeführt und der Cursor vor das erste

Ergebnistupel gesetzt wird.

- `FETCH <cursor-name> INTO <varlist>;`
  setzt den Cursor, solange noch Daten vorhanden sind, auf das nächste Tupel. `FETCH` erzeugt in den folgenden Fällen einen Fehler:
  - der Cursor wurde nicht geöffnet bzw. nicht deklariert.
  - es wurden keine (weiteren) Daten gefunden.
  - der Cursor wurde geschlossen, aber noch nicht wieder geöffnet.
- `CLOSE <cursor-name>;`
  schließt den Cursor.

Für Anfragen, die nur ein einziges Tupel liefern, kann dieses direkt mit einem `SELECT ... INTO :<hostvar>`-Statement einer Hostvariablen zugewiesen werden. In diesem Fall wird impliziert ein Cursor deklariert.

Hostarrays. Mittels *Hostarrays* lässt sich sowohl die Programmierung vereinfachen, als auch der Kommunikationsaufwand zwischen Client und Server reduzieren: Mit einem Hostarray können mit einem einzigen `FETCH`-Zugriff mehrere Tupel aus der Datenbank in die Programmumgebung übertragen werden. Einem Hostarray lässt sich ein entsprechendes Indikatorarray zuordnen. Hostarrays können insbesondere dann verwendet werden, wenn die Größe der Antwortmenge im voraus bekannt ist oder nur ein bestimmter Teil der Antwortmenge interessiert, etwa die ersten 10 Antworten.

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* host array */
   int cityPop[20];         /* host array */
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT Name, Population
         INTO :cityName, :cityPop
         FROM City
         WHERE Code = 'D';
```

Hier wird angenommen, dass die Zahl der deutschen Städte in der Datenbank gleich oder kleiner 20 ist. Somit ist die Verwendung eines Cursors hier nicht nötig. Ist die Zahl der Antworttupel kleiner als die Größe des Hostarrays, so kann die genaue Zahl der übertragenen Tupel über `sqlerrd[2]` in Erfahrung gebracht werden. Reicht die Größe des Array nicht aus um die gesamte Antwortmenge zu fassen, werden nur die ersten 20 Tupel in das Hostarray übertragen. Durch Verwendung eines Cursors ist es jedoch möglich, diesen Schritt so oft zu wiederholen, bis alle Tupel verarbeitet wurden:

```
EXEC SQL BEGIN DECLARE SECTION;
   char cityName[25][20];    /* output host variable */
   int cityPop[20];         /* output host variable */
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE CityCursor CURSOR FOR
   SELECT Name, Population
   FROM City
   WHERE Code = 'D';
```

```
/* demo1.pc */

#include <stdio.h>
#include <sqlca.h>

void sql_error() {
   printf("%s \n",sqlca.sqlerrm.sqlerrmc);
   /* in abhaengigkeit von sqlca.sqlcode weitere Anweisungen
      evtl. exit(1) */
}

int main() {

   EXEC SQL BEGIN DECLARE SECTION;
      char cityName[25];    /* Output Host-variable */
      int cityPop;         /* Output Host-variable */
      char* the_code = "D";    /* Input Host-variable */
      short ind1, ind2;      /* Indikator-variablen */
      char* uid = "/";       /* Anmelden ueber Benutzeraccount */
   EXEC SQL END DECLARE SECTION;
   EXEC SQL WHENEVER SQLERROR DO sql_error();
   /* Verbindung zur Datenbank herstellen */
   EXEC SQL CONNECT :uid;

   /* Cursor deklarieren */
   EXEC SQL DECLARE CityCursor CURSOR FOR
      SELECT Name, Population
      FROM City
      WHERE Code = :the_code;
   EXEC SQL OPEN CityCursor;  /* Cursor oeffnen */

   printf("Stadt                          Einwohner\n");
   printf("---------------------------------------\n");

   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1) {
      EXEC SQL FETCH CityCursor INTO :cityName:ind1 ,
                                     :cityPop INDICATOR :ind2;
      if(ind1 != -1 && ind2 != -1)
      {  /* keine Nullwerte ausgeben */
         printf("%s     %d \n", cityName, cityPop);
      }
   };
   EXEC SQL CLOSE CityCursor;
   EXEC SQL ROLLBACK RELEASE;
}
```

Abbildung 17.3: Beispielprogramm

```
EXEC SQL OPEN CityCursor;

EXEC SQL WHENEVER NOT FOUND DO break;
for(;;)
{
   EXEC SQL FETCH CityCursor INTO :cityName, :cityPop;
   /* Verarbeitung der Tupel */
};
```

In jedem Schleifendurchlauf werden hier 20 Tupel in die Arrays übertragen, außer beim letzten Durchlauf, bei dem es ggf. weniger sein können.

`CURRENT OF`-Klausel. Mit der `CURRENT OF <cursor-name>`-Klausel kann in einem `DELETE`- oder `UPDATE`-Statement Bezug auf das Tupel genommen werden, auf das mit dem vorhergehenden `FETCH` zugegriffen wurde.

```
EXEC SQL DECLARE CityCursor CURSOR FOR
     SELECT Name, Population FROM City;
...
EXEC SQL OPEN CityCursor;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;) {
    EXEC SQL FETCH CityCursor INTO :cityName, :cityPop;
    ...
    EXEC SQL UPDATE City SET Population = :newPop;
    WHERE CURRENT OF CityCursor;
}
```

PL/SQL. Der Oracle Pro*C/C++ Precompiler unterstützt die Verwendung von PL/SQL-Blöcken (siehe Abschnitt 14) in einem C-Programm. Ein PL/SQL-Block kann grundsätzlich überall dort verwendet werden, wo auch "gewöhnliche" SQL-Anweisungen erlaubt sind. Da PL/SQL eigene Konstrukte für die Verarbeitung von Nullwerten besitzt, braucht man dafür nicht auf Indikatorvariablen zurückzugreifen. Hostvariablen können in der bekannten Weise verwendet werden.

Neben dem Vorteil, dass PL/SQL direkt in Oracle integriert ist, bietet es im Zusammenhang mit der Embedded-Programmierung auch einen Laufzeitvorteil gegenüber "gewöhnlichen" SQL-Anweisungen. SQL-Statements werden einzeln an den Oracle Server gesendet und dort verarbeitet. SQL-Anweisungen die in einem PL/SQL-Block zusammengefasst sind, werden hingegen als Ganzes an den Server geschickt (und so wie eine einzige Anweisung behandelt). Somit kann der Kommunikationsaufwand zwischen Client und Server gesenkt werden. Eingebettete PL/SQL-Blöcke werden in einem Rahmen an die Datenbank übergeben:

```
EXEC SQL EXECUTE
DECLARE
  ...
BEGIN
  ...
END;
END-EXEC;
```

Transaktionen. Falls ein Anwendungsprogramm nicht durch `COMMIT`- oder `ROLLBACK`-Anweisungen unterteilt ist, wird es als eine geschlossene Transaktion behandelt. In ORACLE wird nach Beendigung des Programms automatisch ein `ROLLBACK` ausgeführt. Falls Änderungen nicht durch ein explizites oder implizites `COMMIT` dauerhaft wurden, gehen diese verloren. DDL-Anweisungen generieren vor und nach ihrer Ausführung implizit ein `COMMIT`. Die Verbindung zur Datenbank sollte "ordentlich" durch

```
EXEC SQL COMMIT RELEASE;    oder
EXEC SQL ROLLBACK RELEASE;
```

beendet werden. Ohne `RELEASE` können nach Beendigung des Anwendungsprogramms noch solange Ressourcen (wie Locks und Cursor) in der Datenbank belegt sein, bis das Datenbanksystem von sich aus erkennt, dass die Verbindung beendet wurde.

Durch Savepoints lassen sich umfangreichere Transaktionen in einzelne Teile gliedern. Mit

```
EXEC SQL SAVEPOINT <name>;
```

wird ein Savepoint gesetzt. Um einen Teil einer Transaktion rückgängig zu machen, verwendet man ein `ROLLBACK` bis zum betreffenden Savepoint, hier

```
EXEC SQL ROLLBACK TO SAVEPOINT <name>;
```

Zusätzlich werden dabei alle Savepoints und Locks nach dem spezifizierten Savepoint gelöscht bzw. freigegeben. Dagegen hat ein `ROLLBACK` ohne Bezug zu einem Savepoint folgende Auswirkungen:

- alle Änderungen der Transaktion werden rückgängig gemacht,
- alle Savepoints werden gelöscht,
- die Transaktion wird beendet,
- alle expliziten Cursore werden gelöscht.

Da ein `ROLLBACK` Teil einer Ausnahmebehandlung sein kann, ist dessen Einsatz auch in Fehlerroutinen sinnvoll.

Fehlerbehandlung. Ein Anwendungsprogramm sollte so konzipiert sein, dass auf "vorhersehbare" Fehlersituationen, die durch ein SQL-Statement entstehen können, noch sinnvoll reagiert werden kann, etwa durch Ausgabe von Fehlermeldungen oder Abbruch einer Schleife. Dazu stehen zwei Mechanismen zur Verfügung: die *SQL Communications Area* (SQLCA) und das `WHENEVER`-Statement. SQLCA ist eine Datenstruktur, die Statusinformationen bzgl. der zuletzt ausgeführten SQL-Anweisung enthält:

```
struct sqlca {
   char   sqlcaid[8];   /* enthaelt den string "SQLCA" */
   long   sqlcabc;      /* die laenge der struktur in bytes */
   long   sqlcode;      /* enthaelt fehlernummer */
   struct {
          unsigned short sqlerrml;  /* laenge des meldungstextes */
          char sqlerrmc[70];       /* meldungstext */
   } sqlerrm;
   char   sqlerrp[8];   /* zur zeit nicht belegt */
   long   sqlerrd[6];   /* statuscodes */
   char   sqlwarn[8];   /* flags bzgl. warnungen */
   char   sqlext[8];    /* zur zeit nicht belegt */
};
```

Von besonderem Interesse in dieser Datenstruktur ist die Komponente `sqlcode`. Dort sind drei Fälle zu unterscheiden:

- 0: die Verarbeitung einer Anweisung erfolgte ohne Probleme.
- >0: die Verarbeitung ist zwar erfolgt, dabei ist jedoch eine Warnung aufgetreten, z.B. "no data found".
- <0: es trat ein ernsthafter Fehler auf und die Anweisung konnte nicht ausgeführt werden. In diesem Fall sollte eine entsprechende Fehlerroutine abgearbeitet werden.

Im Array `sqlerrd` sind in der aktuellen Version die dritte Komponente (Anzahl der erfolgreich verarbeiteten Sätze) und die fünfte Komponente (Offset in der SQL-Anweisung, in der ein Übersetzungsfehler aufgetreten ist) belegt. Die weiteren Komponenten sind für zukünftigen Gebrauch reserviert. Die SQLCA wird durch die Headerdatei `sqlca.h` deklariert (siehe demo1.pc).

Mit dem `WHENEVER`-Statement können Aktionen spezifiziert werden, die im Fehlerfall automatisch ausgeführt werden sollen. Die Syntax des `WHENEVER`-Statements lautet:

```
EXEC SQL WHENEVER <condition> <action>;
```

Als `<condition>` sind erlaubt:

- `SQLWARNING` : die letzte Anweisung verursachte eine Warnung (siehe auch `sqlwarn`), die Warnung ist verschieden zu "no data found". Dies entspricht `sqlcode` > 0, aber ungleich 1403.
- `SQLERROR` : die letzte Anweisung verursachte einen (ernsthaften) Fehler (siehe oben). Dies entspricht `sqlcode` < 0.
- `NOT FOUND` : `SELECT INTO` bzw `FETCH` liefern keine Antworttupel zurück. Dies entspricht `sqlcode` 1403.

Als `<action>` kommen in Frage:

- `CONTINUE` : das Programm fährt mit der nächsten Anweisung fort.
- `DO <proc_name>` : Aufruf einer Prozedur (Fehlerroutine); `DO break` zum Abbruch einer Schleife.
- `GOTO <label>` : Sprung zu dem angegebenen Label.
- `STOP` : das Programm wird ohne `commit` beendet (exit()), stattdessen wird ein `rollback` ausgeführt.

Der Geltungsbereich einer `WHENEVER`-Anweisung erstreckt sich bis zur nächsten `WHENEVER`-Anweisung, die bzgl. der `<condition>` übereinstimmt.

Es wird empfohlen, nach jeder relevanten SQL-Anweisung den Status abzufragen, um gegebenenfalls eine Fehlerbehandlung einzuleiten. Der Precompiler erzeugt automatisch nach jeder SQL-Anweisung eine entsprechende Abfrage gemäß dem aktuellen `WHENEVER`-Statement.[2]

[2] Man betrachte den C-Code, den der Precompiler aus den `WHENEVER`-Statements generiert.

18 JDBC

Die Programmiersprache Java zeichnet sich insbesondere durch ihre Plattformunabhängigkeit aus – auf jedem Computer, auf dem eine *Virtual Java Machine* läuft, können Java-Programme ablaufen. Um Datenbankzugriffe in Java zu standardisieren (und damit die Entwicklung verschiedener, zueinander inkompatibler Dialekte zu verhindern) wurde das JDBC-API[1] (API = Application Programming Interface, eine Sammlung von Klassen und Schnittstellen, die eine bestimmte Funktionalität bereitstellen) entwickelt. JDBC zielt weiterhin darauf ab, eine Anwendung *unabhängig von dem darunterliegenden Datenbanksystem* programmieren zu können. Um dies gewährleisten zu können (JDBC-intern müssen die Statements schließlich über einen Treiber doch wieder in Statements der jeweiligen SQL-Version umgesetzt werden), müssen die verwendeten SQL-Anweisungen dem SQL92 Entry Level entsprechen – ansonsten riskiert man, nur bestimmte Datenbanken ansprechen zu können.

JDBC gilt als "low-level"-Interface, da SQL-Anweisungen als Zeichenketten direkt ausgeführt werden. Auf Basis des JDBC-API lassen sich dann "higher-level"-Anwendungen erstellen. Aufsetzend auf JDBC sind zwei Arten von "higher-level"-APIs in der Entwicklung:

- Embedded SQL (siehe Abschnitt 17) für Java. Im Gegensatz zu JDBC können in Embedded SQL Java-Programm-Variablen in SQL-Statements verwendet werden um Werte mit der Datenbank auszutauschen. Der Embedded-SQL-Präprozessor übersetzt diese Statements in Java-Statements und JDBC-Aufrufe.
- Direkte Darstellung von Tabellen und Tupeln in Form von Java-Klassen. In dieser objekt-relationalen Darstellung repräsentiert ein Tupel eine Instanz einer Klasse und die Tabellenspalten entsprechen den Attributen des Objektes. Für den Programmierer sind nur die Java-Objekte sichtbar, die entsprechenden SQL-Operationen laufen dagegen im Hintergrund ab.

Ein ähnliches Ziel wie JDBC verfolgt bereits Microsofts ODBC (Open DataBase Connectivity), das eine C-Schnittstelle anbietet, mit der auf fast alle gebräuchlichen Datenbanksysteme zugegriffen werden kann. Derzeit (1999) ist ODBC das am häufigsten verwendete Interface für den Zugriff auf entfernte Datenbanken aus einem Anwendungsprogramm. ODBC kann von Java aus über eine sogenannte JDBC-ODBC-Bridge verwendet werden. Gegenüber ODBC soll JDBC leichter zu verstehen und anzuwenden sein, außerdem werden die bekannten Vorteile von Java bezüglich Sicherheit und Robustheit genutzt. Prinzipiell kann man sich JDBC als ein ODBC vorstellen, das in eine objekt-orientierte Sichtweise übersetzt worden ist. Wie ODBC basiert auch JDBC auf dem X/Open SQL CLI (Call Level Interface) Standard.

18.1 Architektur

Der Zugriff mit JDBC auf die Datenbank erfolgt über (zu JDBC gehörende) Treiber, die mit den spezifischen DBMS, die angesprochen werden sollen, kommunizieren können. Die SQL-Anweisungen

[1] das Akronym für Java Database Connectivity ist zugleich ein eingetragenes Warenzeichen von JavaSoft.

werden in den entsprechenden SQL-Dialekt übersetzt und an die Datenbank gesendet, und die Ergebnisse werden zurückgegeben. Um dieses zu ermöglichen, sind

1. geeignete Treiber für das darunterliegende Datenbanksystem, sowie
2. geeignete SQL-Befehle in dem SQL-Dialekt des darunterliegenden Datenbanksystems

erforderlich.

(1) ist hierbei ein reines Programmierproblem, das bei entsprechendem Bedarf und finanziellem Aufwand zu lösen ist, während (b) von den Datenbankherstellern gelöst werden muss.

18.1.1 Treiber

Der Kern von JDBC ist ein *Treibermanager*, der die Java-Anwendungen mit einem geeigneten JDBC-Treiber verbindet, der den Zugriff auf ein Datenbanksystem ermöglicht. Für jedes verwendete Datenbanksystem muss ein solcher JDBC-Treiber installiert sein. Diese JDBC-Treiber können auf unterschiedliche Arten verwirklicht sein:

- Herstellerabhängiges DBMS-Netzwerk-Protokoll mit *pure Java*-Treiber: Der Treiber wandelt JDBC-Aufrufe in das vom DBMS verwendete Netzwerkprotokoll um. Damit kann der DBMS-Server direkt von (JDBC-)Client-Rechner aufgerufen werden. Diese Treiber werden von den DBMS-Herstellern passend zu dem DBMS-Netzwerkprotokoll entwickelt.
- JDBC-Netz mit *pure Java*-Treiber: JDBC-Aufrufe werden in ein von dem/den DBMS unabhängiges Netzprotokoll übersetzt, das dann auf einem Server in ein bestimmtes DBMS-Protokoll übersetzt wird. Über diese Netzserver-Middleware können Java-Clients mit vielen verschiedenen Datenbanken verbunden werden. Es ist somit die flexibelste Lösung – die allerdings auf geeignete Produkte der Hersteller angewiesen ist.

Diese Treiber sind natürlich der bevorzugte Weg, um aus JDBC auf Datenbanken zuzugreifen (u.a., da auch hier die Installation Java-typisch automatisch aus einem Applet erfolgen kann, das vor Benutzung den (pure-Java!)-Treiber lädt). Bis alle entsprechenden Treiber auf dem Markt sind, finden jedoch einige Übergangslösungen Anwendung, die auf bereits vorhandenen (Non-Java)-Treibern basieren:

- JDBC-ODBC-Bridge und ODBC-Treiber: Über die JDBC-ODBC-Bridge werden die ODBC-Treiber verwendet. Dazu muss im allgemeinen der ODBC-Binärcode und häufig auch der Datenbankcode der Clients auf jeder Client-Maschine geladen sein – jede Maschine muss damit ein vollwertiger Client sein.
- Natives API und teilweise in Java geschriebene Treiber: Die JDBC-Aufrufe werden in Aufrufe von Client-APIs der entsprechenden Datenbankhersteller übersetzt. Auch hier wird der Binärcode auf jeder Client-Maschine geladen.

Welche JDBC-Treiber verfügbar sind, kann man aktuell auf `http://www.javasoft.com/products/jdbc` erfahren. Mittlerweile sollten für alle gängigen Datenbanksysteme geeignete pure-Java-Treiber vorhanden sein. Die Tauglichkeit von JDBC-Treibern wird durch eine JDBC-Treiber-Testsuite überprüft und durch die Bezeichnung *JDBC Compliant* bestätigt. Solche Treiber unterstützen zumindest ANSI SQL-2 Entry Level (der SQL-92 Standard).

Die Architektur von JDBC ist in Abbildung 18.1 gezeigt.

Der im Praktikum für Oracle verwendete Treiber gehört zu der zuerst genannten Art: Der Treiber ruft *direkt* den Oracle-Server auf. Damit sind prinzipiell alle SQL-Befehle von Oracle möglich, man unterliegt also nicht der Einschränkung auf den SQL-2 Standard. Andererseits können Programme,

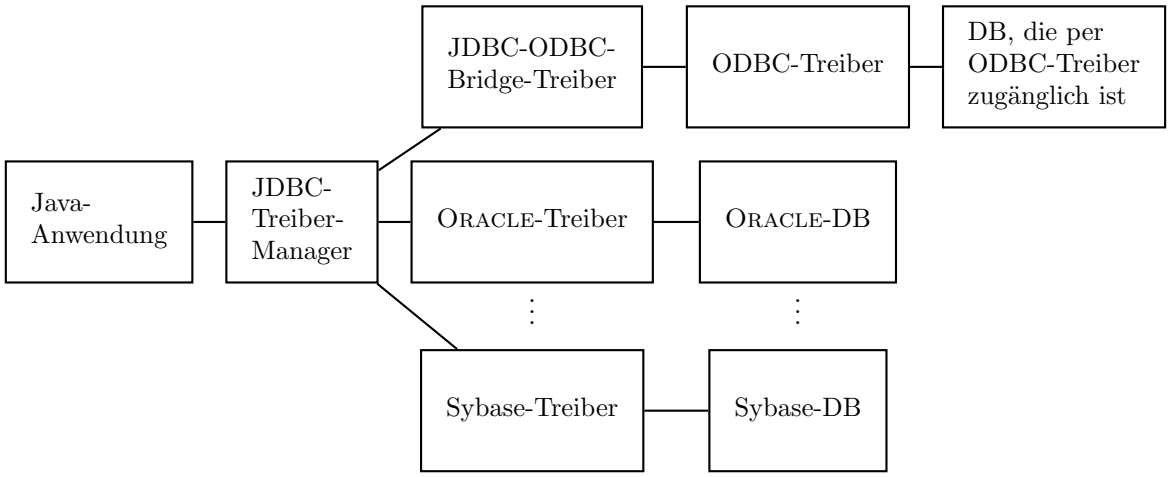


Abbildung 18.1: JDBC-Architektur

die dies ausnutzen (z.B. Erzeugung von *stored Procedures* oder objektrelationalen Features) i.a. *nicht* mit anderen Datenbanksystemen eingesetzt werden.

18.2 JDBC-Befehle

Die Funktionalität von JDBC lässt sich in drei Bereiche untergliedern:

- Aufbau einer Verbindung zur Datenbank.
- Versenden von SQL-Anweisungen an die Datenbank.
- Verarbeitung der Ergebnismenge.

Für den Benutzer wird diese Funktionalität im wesentlichen durch die im folgenden beschriebenen Klassen `Connection, Statement` (und davon abgeleitete Klassen) und `ResultSet` repräsentiert.

18.2.1 Verbindungsaufbau

Die Verwaltung von Treibern und der Aufbau von Verbindungen wird über die Klasse `DriverManager` abgewickelt. Alle Methoden von `Driver Manager` sind als *static* deklariert – d.h. operieren auf der Klasse, nicht auf irgendwelchen Instanzen. Der Konstruktor für `Driver Manager` ist sogar als `private` deklariert, womit der Benutzer keine Instanzen der Klasse erzeugen kann.

Bevor man eine Verbindung zu einer Datenbank herstellen kann, muss der entsprechende Treiber geladen werden – was hier über den Aufruf der Methode `registerDriver` der Klasse `DriverManager` geschieht:

```
DriverManager.registerDriver(<driver-name>);
```

Welcher Klassenname `<driver-name>` eingesetzt werden muss, steht in dem Handbuch des entsprechenden Treibers, im Praktikum ist dies

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

Der Verbindungsaufbau zu der lokalen Datenbank wird über die Methode

```
DriverManager.getConnection(<jdbc-url>, <user-id>, <passwd>)
```

der Klasse DriverManager vorgenommen. Analog zu den URLs (*Uniform Resource Locator*) des Internet bezeichnet eine JDBC-URL eindeutig eine Datenbank. Eine JDBC-URL besteht wiederum aus drei Teilen:

```
<protocol>:<subprotocol>:<subname>
```

Innerhalb einer JDBC-URL wird immer das `jdbc`-Protokoll verwendet. Dabei bezeichnet `<subprotocol>` den Treiber oder den Datenbank-Zugriffsmechanismus, den ein Treiber unterstützt (siehe Dokumentation des Treibers). Die Datenbank wird durch `<subname>` identifiziert. Der Aufbau des `<subname>` erfolgt in Abhängigkeit des `<subprotocol>`. Für den im Praktikum verwendeten Treiber lautet die Syntax:

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

wobei `<driver-name>` der entsprechende durch `DriverManager.registerDriver(<driver-name>)` registrierte Treibername ist. Wird die User-Id und das Passwort innerhalb der JDBC-URL angegeben, so kann auf die Angabe dieser Argumente innerhalb von `getConnection` verzichtet werden.

Durch den Aufruf von

```
DriverManager.getConnection(<jdbc-url>, <user-id>, <passwd>)
```

wird für jeden registrierten Treiber `<driver>` die Methode `<driver>.connect(<jdbc-url>)` aufgerufen. Derjenige Treiber, der als erster die gewünschte Verbindung herstellen kann, wird genommen. Der Aufruf liefert eine offene Verbindung zurück, die natürlich einem Bezeichner zugewiesen werden muss. Im Beispiel wird die Verbindung `conn` mit

```
String url = ‘‘jdbc:oracle:thin:jdbc_1/jdbc_1@132.230.150.11:1521:dev’’;
Connection conn = DriverManager.getConnection(url,‘‘dummy’’,‘‘passwd’’);
```

aufgebaut. Mit der Verbindung an sich kann man eigentlich noch nicht so richtig etwas anfangen. Insbesondere ist es nicht – wie man vielleicht erwartet hätte – möglich, über die Verbindung direkt SQL-Statements an die Datenbank zu übergeben.

Mit der Methode `close` kann ein Connection-Objekt geschlossen werden.

18.2.2 Versenden von SQL-Anweisungen

SQL-Anweisungen werden über `Statement`-Objekte an die Datenbank gesendet. `Statement`-Objekte werden durch Aufruf der Methode `createStatement` (und verwandter Methoden) einer bestehenden Verbindung `<connection>` erzeugt.

Zur Zeit existieren drei verschiedene Klassen von SQL-Anweisungen:

- `Statement`: Instanzen dieser Klasse werden per `<connection>.createStatement()` erzeugt. Mit `Statement` können nur einfache SQL-Anweisungen ohne Parameter verarbeitet werden.
- `PreparedStatement`: Instanzen dieser Klasse werden per `<connection>.prepareStatement(<string>)` erzeugt. Die Klasse `PreparedStatement` ist von der Klasse `Statement` abgeleitet. Neben der Vorcompilierung von Anfragen erlaubt sie auch die Verwendung von Anfragen mit Parametern (vgl. Abschnitt 18.2.4).
- `CallableStatement`: Instanzen dieser Klasse werden per `<connection>.prepareCall` erzeugt. Die Klasse `CallableStatement` ist wiederum von `PreparedStatement` abgeleitet; hiermit können in der

Datenbank gespeicherte Prozeduren aufgerufen werden (vgl. Abschnitt 18.2.5).

Die erzeugte Statement-Instanz wird einem Bezeichner zugewiesen, um später wieder verwendet werden zu können:

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

Die damit erzeugte Instanz `<name>` der Klasse `Statement` kann nun verwendet werden, um SQL-Befehle an die Datenbank zu übermitteln. Dies geschieht abhängig von der Art des SQL-Statements über verschiedene Methoden. Im folgenden bezeichnet `<string>` ein SQL-Statement *ohne Semikolon*.

- `<statement>.executeQuery(<string>)`: Mit `executeQuery` werden *Anfragen* an die Datenbank übergeben. Dabei wird eine Ergebnismenge an eine geeignete Instanz der Klasse `ResultSet` zurückgegeben (siehe Abschnitt 18.2.3).
- `<statement>.executeUpdate(<string>)`: Als Update werden alle SQL-Statements bezeichnet, die eine Veränderung an der Datenbasis vornehmen, insbesondere alle DDL-Anweisungen (`CREATE TABLE`, etc.), außerdem `INSERT`-, `UPDATE`- und `DELETE`-Statements. Der Rückgabewert von `executeUpdate` gibt an, wieviele Tupel von der SQL-Anweisung betroffen waren. Dieser Wert ist 0, wenn `INSERT`, `UPDATE` oder `DELETE` kein Tupel betreffen oder eine DDL-Anweisung ausgeführt wurde. Abhängig davon, ob man an dem Rückgabewert interessiert ist, kann man `executeUpdate` entweder als im Stil einer Prozedur oder einer Funktion aufrufen:
  ```
  <statement>.executeUpdate(<string>);
  int n = <statement>.executeUpdate(<string>);
  ```
- `<statement>.execute(<string>)`: Die Methode `execute` wird verwendet, wenn ein Statement mehr als eine Ergebnismenge zurückliefert. Dies ist z.B. der Fall, wenn verschiedene gespeicherte Prozeduren und SQL-Befehle nacheinander ausgeführt werden (vgl. Abschnitt 18.2.6).

Ein einmal erzeugtes `Statement`-Objekt kann beliebig oft wiederverwendet werden, um SQL-Anweisungen zu übermitteln. Da dabei mit der Übermittlung des nächsten Statements der Objektzustand verändert wird, sind die Übertragungsdaten des vorhergehenden Statements, z.B. Warnungen, danach nicht mehr zugreifbar.

Mit der Methode `close` kann ein Statement-Objekt geschlossen werden.

18.2.3 Behandlung von Ergebnismengen

Die Ergebnismenge des SQL-Statements wird durch eine Instanz der Klasse `ResultSet` repräsentiert:

```
ResultSet <name> = <statement>.executeQuery(<string>);
```

prinzipiell ist es (wieder einmal) eine virtuelle Tabelle, auf die von der "Hostsprache" – hier also Java – zugegriffen werden kann. Zu diesem Zweck unterhält ein `ResultSet`-Objekt einen Cursor, der durch die Methode `<result-set>.next` jeweils auf das nächste Tupel gesetzt wird. Der Cursor bleibt solange gültig wie die entsprechenden `ResultSet` und `Statement` Objekte existieren. Sind alle Elemente eines ResultSets gelesen, liefert `<result-set>.next` den Wert `false` zurück.

Auf die einzelnen Spalten des jeweils unter dem Cursor befindlichen Ergebnistupels wird über die Methoden `<result-set>.get<type>(<attribute>)` zugegriffen. `<type>` ist dabei ein Java-Datentyp, `<attribute>` kann entweder über den Attributnamen, oder durch die Spaltennummer (beginnend bei 1) gegeben sein. JDBC steht zwischen Java (mit den bekannten Objekttypen) und den verschiedenen SQL-Dialekten (die prinzipiell immer dieselben Datentypen, aber unter unterschiedlichen Namen verwenden). In `java.sql.types` werden *generische* SQL-Typen definiert, mit denen JDBC arbeitet.

Bei dem Aufruf von `get<type>` werden die Daten des Ergebnistupels (die als SQL-Datentypen vorliegen) in Java-Typen konvertiert wie in Abbildung 18.2 angegeben.

Java-Typ	JDBC/SQL-Typ mit Codenummer
`String`	`CHAR (1), VARCHAR (12), LONGVARCHAR (-1)`
`java.math.BigDecimal`	`NUMERIC (2), DECIMAL (3)`
`boolean`	`BIT (-7)`
`byte`	`TINYINT (-6)`
`short`	`SMALLINT (5)`
`int`	`INTEGER (4)`
`long`	`BIGINT (-5)`
`float`	`REAL (7), FLOAT (6)`
`double`	`DOUBLE (8)`
`java.sql.Date`	`DATE (91)` (für ORACLE-Typ `DATE`: Tag, Monat, Jahr)
`java.sql.Time`	`TIME (92)` (für ORACLE-Typ `DATE`: Stunde, Minute, Sekunde)
`java.sql.Timestamp`	`TIMESTAMP (93)`

Abbildung 18.2: Abbildung von Java- auf SQL-Datentypen

Die empfohlene Zuordnung der wichtigsten SQL-Typen zu `get<type>`-Methoden ist in Abbildung 18.3 angegeben (für weitere Methoden, siehe Literatur). Wenn man nicht weiß, von welchem Datentyp eine Spalte ist, kann man immer `<getString>` verwenden.

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

Abbildung 18.3: `get<type>`-Methoden zu JDBC/SQL-Datentypen

Beispiel 38 Ein einfaches Beispiel, das für alle Städte den Namen und die Einwohnerzahl ausgibt, sieht dann etwa so aus:

```
import jdbc.sql.*;

class Hello {
  public static void main (String args []) throws SQLException {
    // \Oracle-Treiber laden
    DriverManager.registerDriver(new oracle.jdbc.dnlddriver.OracleDriver());
    // Verbindung zur Datenbank herstellen
    String url = "jdbc:oracle:dnldthin:@132.230.150.161:1521:test";
    Connection conn = DriverManager.getConnection(url,"dummy","passwd");
    // Anfrage an die Datenbank
    Statement stmt = conn.createStatement ();
```

```
    ResultSet rset = stmt.executeQuery("SELECT Name, Population FROM City);
    while (rset.next ()) {   // Verarbeitung der Ergebnismenge
        String s = rset.getString(1);
        int i = rset.getInt("Population");
        System.out.println(s + "  " + i "\n");
    }
  }
}
```

An dieser Stelle ist anzumerken, dass `System.out.println(...)` die Ausgabe in das xterm sendet. Für die Anwendung als Applet muss `output.appendText(...)` verwendet werden (siehe Aufgaben);□

Informationen über die Spalten der Ergebnismenge können mit `<result-set>.getMetaData` in Erfahrung gebracht werden, womit ein `ResultSetMetaData`-Objekt erzeugt werden kann, das alle Informationen über die Ergebnismenge enthält:

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

Die wichtigsten Methoden von `ResultSetMetaData` sind in Abschnitt 18.4 angegeben.

Da die Java-Datentypen keine `NULL`-Werte vorsehen, kann mit der Methode `<resultSet>.wasNULL()` getestet werden, ob der zuletzt gelesene Spaltenwert `NULL` war.

Beispiel 39 Das folgende Codefragment gibt die aktuelle Zeile eines ResultSets – `rset` einschließlich Nullwerten – aus: `ResultSet.getColumnCount` gibt die Anzahl der Spalten, die dann mit `ResultSet.getString` eingelesen werden.

```
ResultSetMetaData rsetmetadata = rset.getMetaData();
int numCols = rsetmetadata.getColumnCount();
for(i=1; i<=numCols; i++) {
    String returnValue = rset.getString(i);
    if (rset.wasNull())
        System.out.println ("null");
    else
       System.out.println (returnValue);
}
```

Mit der Methode `close` kann ein ResultSet-Objekt explizit geschlossen werden.

18.2.4 Prepared Statements

Im Gegensatz zu der Klasse `Statement` wird bei `PreparedStatement` die SQL-Anweisung bei der Erzeugung des Statement-Objektes durch

```
PreparedStatement <name> = <connection>.prepareStatement(<string>);
```

vorcompiliert. Im Gegensatz zu einem `Statement` – bei dem dem bei `execute...` immer das auszuführende Statement als `<string>` angegeben wird, ist bei `PreparedStatement` der SQL-Befehl bereits in dem Objektzustand fest enthalten. Damit ist die Verwendung eines `PreparedStatement` effizienter als `Statement`, wenn ein SQL-Statement häufig ausgeführt werden soll.

Abhängig von der Art des SQL-Befehls ist für ein `PreparedStatement` auch nur eine der (hier parameterlosen!) Methoden `<prepared-statement>.executeQuery()`, `<prepared-statement>.executeUpdate()` oder `<prepared-statement>.execute()` anwendbar, um es auszuführen.

Zusätzlich kann der SQL-Befehl `<string>` bei der Erzeugung eines `PreparedStatement`s Eingabeparameter, bezeichnet durch "?", enthalten:

```
PreparedStatement pstmt = con.prepareStatement("SELECT Population FROM
                                                Country WHERE Code = ?");
```

Die Werte für die "?"-Parameter werden mit Hilfe der 2-stelligen Methoden

```
<prepared-statement>.set<type>(<pos>,<value>);
```

gesetzt, bevor ein `PreparedStatement` ausgeführt wird. `<type>` gibt dabei den Java-Datentyp an, `<pos>` gibt die Position des zu setzenden Parameters an, `<value>` den zu setzenden Wert.

Beispiel 40 Mit dem folgenden Codefragment können zuerst alle Städte in Deutschland und danach alle Städte in der Schweiz bearbeitet werden:

```
PreparedStatement pstmt = con.prepareStatement("SELECT Population FROM
                                                Country WHERE Code = ?");
pstmt.setString(1,"D");
ResultSet rset = pstmt.ExecuteQuery();
...
pstmt.setString(1,"CH");
ResultSet rset = pstmt.ExecuteQuery();
...
```

`PreparedStatements` mit Parametern sind insbesondere praktisch, um in Schleifen verwendet zu werden. □

Analog zu `<result-set>.get<type>` bildet Java den Java-Datentyp `<type>` auf einen SQL-Datentyp ab.

Nullwerte können über die Methode `setNULL(<pos>,<type>)` als Parameter angegeben werden, wobei `<type>` den SQL-Typ dieser Spalte bezeichnet:

```
pstmt.setNULL(1,Types.String);
```

18.2.5 Callable Statements: Gespeicherte Prozeduren

Prozeduren und Funktionen werden in der üblichen Syntax (`CREATE PROCEDURE` bzw. `CREATE FUNCTION`) durch

```
<statement>.executeUpdate(<string>);
```

als Datenbankobjekte erzeugt. Dann wird der *Aufruf der Prozedur* als `CallableStatement`-Objekt erzeugt. Da die Aufrufsyntax der verschiedenen Datenbanksysteme unterschiedlich ist, wird in JDBC eine generische Syntax per Escape-Sequenz verwendet. Mit dem Befehl

```
CallableStatement <name> = <connection>.prepareCall(‘‘{call <procedure>}’’);
```

wird ein `callableStatement`-Objekt erzeugt. Je nachdem, ob die Prozedur eine Anfrage (die ein *ResultSet* zurückgibt), eine Veränderung an der Datenbank, oder mehrere aufeinanderfolgende SQL-Statements, die unterschiedliche Ergebnisse zurückgeben, ausführt, wird die Prozedur mit

```
ResultSet <name> = <callable-statement>.executeQuery();
<callable-statement>.executeUpdate();
oder
<callable-statement>.execute();
```

aufgerufen. Da `CallableStatement` eine Subklasse von `PreparedStatement` ist, können auch hier Parameter – hier auch als `OUT`-Parameter der Prozedur – verwendet werden:

```
CallableStatement <name> = <connection>.prepareCall(``{call <procedure>(?,...,?)}'');
```

Gibt die Prozedur einen Rückgabewert zurück [testen, ob das mit Oracle-PL/SQL-Funktionen klappt], ist die Syntax folgendermaßen:

```
CallableStatement <name> =
  <connection>.prepareCall(``{? = call <procedure>(?,...,?)}'');
```

Ob die einzelnen "?"-Parameter `IN`, `OUT`, oder `INOUT`-Parameter sind, hängt von der Prozedurdefinition ab und wird von JDBC anhand der Metadaten der Datenbank selbständig analysiert.

`IN`-Parameter werden wie bei `PreparedStatement` über `set<type>` gesetzt. Für `OUT`-Parameter muss vor der Verwendung zuerst der JDBC-Datentyp der Parameter mit

```
<callable-statement>.registerOutParameter(<pos>,java.sql.Types.<type>);
```

registriert werden (die Prozedurdefinition enthält ja nur den SQL-Datentyp). Um den `OUT`-Parameter dann schlussendlich nach Ausführung der Prozedur lesen zu können, wird die entsprechende `get<type>`-Methode verwendet:

```
<java-type> <var> = <callable-statement>.get<type>(<pos>);
```

Für `INOUT`-Parameter muss ebenfalls `registerOutParameter` aufgerufen werden. Sie werden entsprechend mit `set<type>` gesetzt und mit `get<type>` gelesen.

Liefert der Aufruf eines `CallableStatement`-Objekts mehrere Result Sets zurück, sollte man aus Portabilitätsgründen alle diese Ergebnisse lesen, bevor `OUT`-Parameter gelesen werden.

18.2.6 Sequenzielle Ausführung

Mit `<statement>.execute(<string>)`, `<prepared-statement>.execute()` und `<callable-statement>.execut` können SQL-Statements, die mehrere Ergebnismengen zurückliefern, an eine Datenbank übermittelt werden. Dies kommt speziell dann vor, wenn ein SQL-Statement in Java dynamisch als String generiert und dann komplett an die Datenbank gesendet wird (Verwendung der Klasse `Statement`) und man nicht weiß, wieviele Ergebnismengen im speziellen Fall zurückgeliefert werden.

Nach Ausführung von `execute` kann mit `getResultSet` bzw. `getUpdateCount` die erste Ergebnismenge bzw. der erste Update-Zähler entgegengenommen werden. Um jede weitere Ergebnismengen zu holen, muss `getMoreResults` und dann wieder `getResultSet` bzw. `getUpdateCount` aufgerufen werden. Wenn man nicht weiß, welcher Art das erste/nächste Ergebnis ist, muss man es ausprobieren:

- Aufruf von `getResultSet`: Falls das nächste Ergebnis eine Ergebnismenge ist, wird diese zurückgegeben. Falls entweder kein nächstes Ergebnis mehr vorhanden ist, oder das nächste Ergebnis keine

Ergebnismenge sondern ein Update-Zähler ist, so wird `null` zurückgegeben.

- Aufruf von `getUpdateCount`: Falls da nächste Ergebnis ein Update-Zähler ist, wird dieser (eine Zahl $n \geq 0$) zurückgegeben; falls das nächste Ergebnis eine Ergebnismenge ist, oder keine weiteren Ergebnisse mehr vorliegen, wird -1 zurückgegeben.
- Aufruf von `getMoreResults`: `true`, wenn das nächste Ergebnis eine Ergebnismenge ist,`false`, wenn es ein Update-Zähler ist, oder keine weiteren Ergebnisse mehr abzuholen sind.
- es sind also alle Ergebnisse verarbeitet, wenn

  `((<stmt>.getResultSet() == null) && (<stmt>.getUpdateCount() == -1))`

  bzw.

  `((<stmt>.getMoreResults() == false) && (<stmt>.getUpdateCount() == -1))`

  ist.

Dabei kann man mit `getResultSet`, `getUpdateCount` und `getMoreResults` bestimmen, ob ein Ergebnis ein `ResultSet`-Objekt oder ein Update-Zähler ist und eine Ergebnismenge nach der anderen ansprechen.

Beispiel 41 (Sequenzielle Verarbeitung von Ergebnismengen) Das folgende Codesegment (aus [HCF98] geht eine Folge von Ergebnisse (Ergebnismengen und Update-Zähler in beliebiger Reihenfolge) durch:

```
stmt.execute(queryStringWithUnknownResults);
while (true) {
    int rowCount = stmt.getUpdateCount();
    if (rowCount > 0) {
       System.out.println("Rows changed = " + count);
       stmt.getMoreResults();
       continue;
    }
    if (rowCount == 0) {
       System.out.println("No rows changed");
       stmt.getMoreResults();
       continue;
    }
    ResultSet rs = stmt.getResultSet;
    if (rs != null) {
       .....  // benutze Metadaten ueber die Ergebnismenge
       while (rs.next()) {
           ....  // verarbeite Ergebnismenge
           stmt.getMoreResults();
           continue;
    }
    break;
}
```

18.3 Transaktionen in JDBC

nach der Erzeugung befinden sich Connection-Objekte im *Auto-Commit-Modus*, d.h. alle SQL-Anweisungen werden automatisch als individuelle Transaktionen freigegeben. Der Auto-Commit-Modus kann durch die Methode

```
<connection>.setAutoCommit(false);
```

aufgehoben werden. Danach können Transaktionen durch

```
<connection>.commit()       oder <connection>.rollback()
```

explizit freigegeben oder rückgängig gemacht werden.

18.4 Schnittstellen der JDBC-Klassen

Dieser Abschnitt gibt eine Auflistung relevanter Schnittstellen einzelner JDBC Klassen. Die getroffene Einschränkung erklärt sich daraus, dass für Standardanwendungen nur ein (kleiner) Teil der Methoden benötigt wird.[2]

Die Schnittstellen der JDBC-Klassen Numeric, DatabaseMetaData, Date, DataTruncation, DriverPropertyInfo, CallableStatement, Timestamp, SQLException, SQLWarning, NULLData, Time, Date und Types werden hier deshalb nicht dargestellt.

Driver:

Methode	Beschreibung
boolean acceptsURL(String)	`true` falls Treiber mit angegebener JDBC-URL eine Verbindung zur Datenbank herstellen kann
Connection connect(String,Properties)	Verbindungsaufbau zur Datenbank
boolean jdbcCompliant()	ist es oder ist es nicht?

DriverManager:

Methode	Beschreibung
void registerDriver(Driver)	Treiber wird dem DriverManager bekannt gemacht
Connection getConnection(String)	Verbindungsaufbau
Connection getConnection(String, Properties)	Verbindungsaufbau
Connection getConnection(String, String, String)	Verbindungsaufbau
Driver getDriver(String)	Versucht, einen Treiber zu finden, der die durch String gegebene URL versteht
Enumeration getDrivers()	Liste der verfügbaren JDBC-Treiber
void deregisterDriver(Driver)	
DriverManager DriverManager()	Konstruktor (`private`)

Connection:

[2] Die Klasse `DatabaseMetaData` verfügt über mehr als 100 Schnittstellen.

Methode	Beschreibung
Statement createStatement()	s.o.
CallableSatement prepareCall(String)	s.o.
PreparedStatement prepareStatement(String)	s.o.
DatabaseMetaData getMetaData()	eine Instanz der Klasse DatabaseMetaData enthält Informationen über die Verbindung
void commit()	alle Änderungen seit dem letzten Commit bzw. Rollback werden dauerhaft, alle Locks werden freigegeben
void rollback()	alle Änderungen seit dem letzten Commit bzw. Rollback werden verworfen, alle Locks werden freigegeben
void clearWarnings()	alle Warnings der Verbindung werden gelöscht
SQLWarning getWarnings()	erste Warnung die im Zusammenhang mit der Verbindung aufgetreten ist
void setAutoCommit(boolean)	auto-commit Status wird gesetzt
void setReadOnly(boolean)	read-only Modus wird spezifiziert
boolean isReadOnly()	
void close()	die Verbindung zwischen der Instanz und der Datenbank wird beendet
boolean isClosed()	

Statement:

Methode	Beschreibung
ResultSet executeQuery(String)	s.o.
int executeUpdate(String)	s.o.
boolean execute(String)	s.o
ResultSet getResultSet()	Ergebnismenge zum ausgeführten SQL Statement
int getUpdateCount()	Ergebnis des letzten Update
void clearWarnings()	alle Warnings der Statement-Instanz werden gelöscht
SQLWarning getWarnings()	erste Warnung im Zusammenhang mit dem Statement

PreparedStatement: Subklasse von Statement

Methode	Beschreibung
ResultSet executeQuery()	s.o.
int executeUpdate()	s.o.
boolean execute()	s.o.
void set‹type›(int, ‹type›)	entsprechend dem Parameterindex wird ein Argument vom Datentyp ‹type› gesetzt

ResultSet:

Methode	Beschreibung
‹type› get‹type›(int)	gibt eine Instanz vom Typ ‹type› entsprechend dem Spaltenindex zurück
‹type› get‹type›(String)	gibt eine Instanz vom Typ ‹type› entsprechend dem Attributnamen zurück
ResultSetMetaData getMetaData()	eine Instanz der Klasse ResultSetMetaData enthält Informationen über Anzahl, Typ und weitere Eigenschaften der Spalten der Ergebnismenge
int findColumn(String)	Spaltenindex zum Attributnamen
boolean next()	es wird auf das nächste Ergebnistupel zugegriffen, Rückgabewert ist `false` wenn keine weiteren Ergebnistupel existieren
boolean wasNull()	`true` falls der zu letzt gelesene Wert `NULL` war, sonst false
void close()	Verbindungsabbau zwischen der ResultSet Instanz und Datenbank

ResultSetMetaData:

Methode	Beschreibung
int getColumnCount()	Spaltenanzahl der Ergebnismenge
String getColumnLabel(int)	Attributname der Spalte ‹int›
String getTableName(int)	Tabellenname der Spalte ‹int›
String getSchemaName(int)	Schemaname der Spalte ‹int›
int getColumnType(int)	JDBC-Typ der Spalte ‹int› (als Integer-Wert, siehe Abbildung 18.2)
String getColumnTypeName(int)	Unterliegender DBMS-Typ der Spalte ‹int›

A DIE Mondial-DATENBANK

Die MONDIAL Datenbasis [1] enthält geographische Informationen über Länder, Provinzen, Städte, Organisationen, sowie Berge, Gewässer, Inseln und Wüsten.

Zur Erstellung der Datenbasis wurden Informationen aus den folgenden Quellen verwendet:

- CIA World Factbook (http://www.odci.gov/cia/publications/factbook/index.html) in der Version von 1996 (Informationen über Länder und politische Organisationen),
- "Global Statistics", eine Zusammenstellung über Länder, Städte und Landesteile (http://www.stats.demon.nl/) (courtesy of Johan van der Heijden).
- einige geographische Koordinaten von `http://www.bcca.org/misc/qiblih/latlong.html`,
- sowie geographische Daten aus der TERRA-Datenbasis.
- Update 2005.

Die Datengewinnung aus dem WWW sowie die Datenintegration erfolgte mit Hilfe des F-Logic-Systems FLORID[2] des Lehrstuhls für Datenbanken und Informationssysteme der Universität Freiburg.

[1] http://www.informatik.uni-freiburg.de/~may/Mondial/
[2] http://www.informatik.uni-freiburg.de/~dbis/florid/

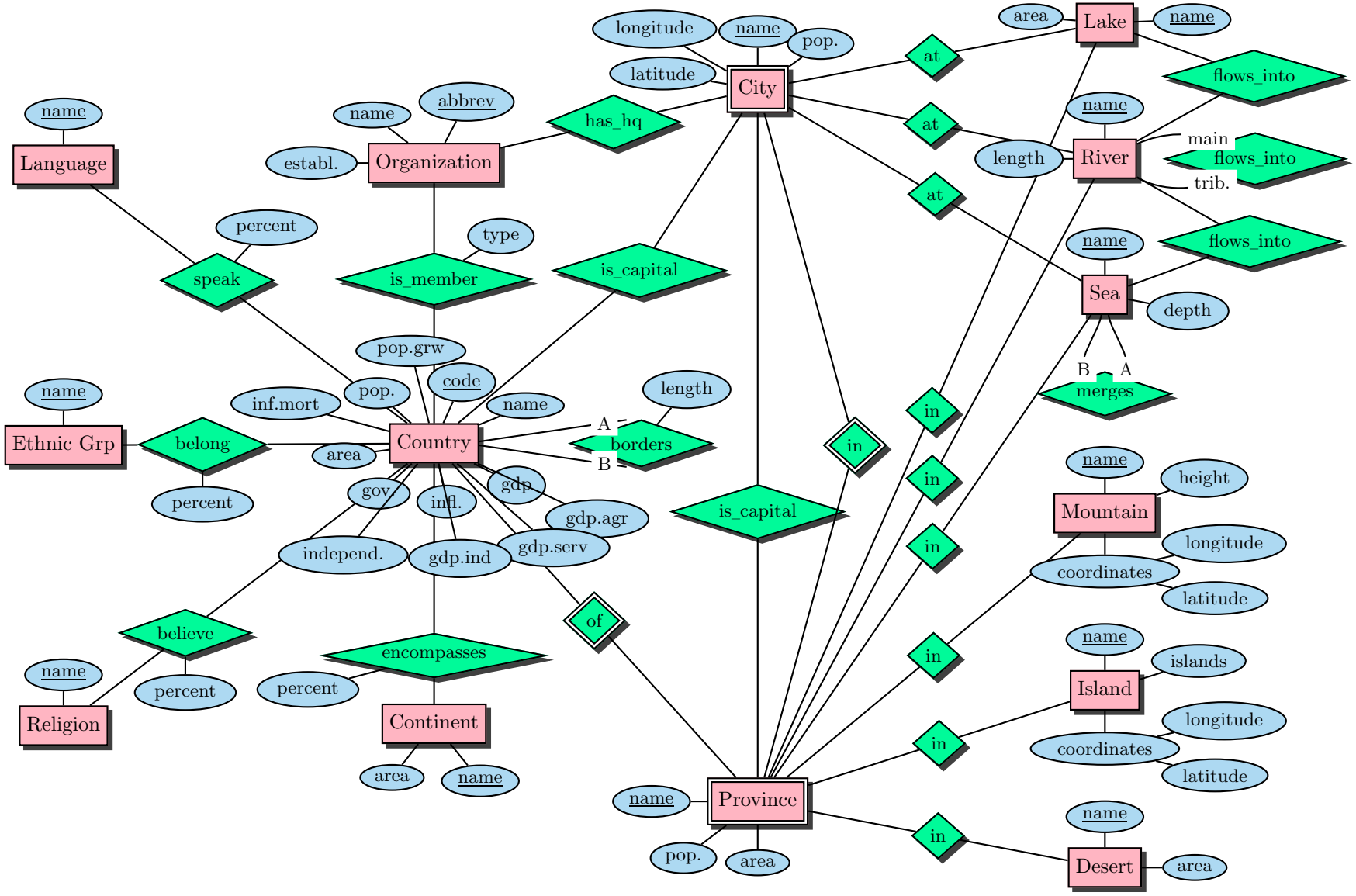


Abbildung A.1: ER-Diagramm der MONDIAL-Datenbasis

Das relationale Schema der Datenbank enthält die folgenden Relationen und Attribute. Abbildung A.2 zeigt die referentiellen Abhängigkeiten innerhalb des Schemas.

Types

GeoCoord: geographic position.
- **Latitude**: geographic latitude
- **Longitude**: geographic longitude

Tables

Country: the countries of the world with some data.
- **Name**: the country name
- **Code**: the internet country code (two letters)
- **Capital**: the name of the capital
- **Province**: the province where the capital belongs to
- **Population**: the population number
- **Area**: the total area

Province: information about administrative divisions.
- **Name**: the name of the administrative division
- **Country**: the country code where it belongs to
- **Population**: the population of the province
- **Area**: the total area of the province
- **Capital**: the name of the capital
- **CapProv**: the name of the province where the captital belongs to

⇒ Note that *capprov* is not necessarily equal to *name*. For example, the municipality of *Bogota (Columbia)* is a province of its own, and *Bogota* is the capital of the surrounding province *Cudinamarca*.

City: information about cities.
- **Name**: the name of the city
- **Country**: the country code where it belongs to
- **Province**: the name of the province where it belongs to
- **Population**: population of the city
- **Latitude**: geographic latitude
- **Longitude**: geographic longitude

Continent: information about continents.
- **Name**: name of the continent
- **Area**: total area of the continent

encompasses: information to which continents a country belongs.
- **Country**: the country code
- **Continent**: the continent name
- **Percentage**: percentage, how much of the area of a country belongs to the continent

borders: informations about neighboring countries.
- **Country1**: a country code
- **Country2**: a country code
- **Length**: length of the border between country1 and country2

⇒ Note that in this relation, for every pair of neighboring countries (A, B), only one tuple is given – thus, the relation is *not* symmetric.

Organization: information about political and economical organizations.
- **Abbreviation**: the abbreviation of the organization
- **Name**: the full name of the organization
- **Established**: date of establishment
- **City**: the city where it is seated
- **Province**: the province of its seat
- **Country**: the country code of its seat

is_member: memberships in political and economical organizations.
- **Organization**: the abbreviation of the organization
- **Country**: the code of the member country
- **Type**: the type of membership

Economy: economical information about the countries.
- **Country**: the country code
- **GDP**: gross domestic product (in million dollar)
- **Agriculture**: percentage of agricultural sector of the GDP
- **Industry**: percentage of industrial sector of the GDP
- **Services**: percentage of service sector of the GDP
- **Inflation**: inflation rate (percentage, per annum)

Population: information about the population of the countries.
- **Country**: the country code
- **Population_Growth**: population growth rate (percentage, per annum)
- **Infant_Mortality**: infant mortality (per thousand)

Politics: political information about the countries.
- **Country**: the country code
- **Independence**: date of independence
- **Government**: type of government

Language: information about the languages spoken in a country.
- **Country**: the country code
- **Name**: name of the language
- **Percentage**: percentage of the language in this country

Religion: information about the religions in a country.
- **Country**: the country code
- **Name**: name of the religion
- **Percentage**: percentage of the religion in this country

Ethnic_Group: information about the ethnic groups in a country.
- **Country**: the country code
- **Name**: name of the ethnic group
- **Percentage**: percentage of the ethnic group in this country

located: information about cities located at rivers, lakes, and seas.
- **City**: the name of the city
- **Province**: the province where the city belongs to
- **Country**: the country code where the city belongs to
- **River**: the river where it is located at
- **Lake**: the lake where it is located at
- **Sea**: the sea where it is located at

⇒ Note that for a given city, there can be several lakes/seas/rivers where it is located at.

River: information about rivers.
- **Name**: the name of the river
- **River**: the river where it flows to
- **Lake**: the lake where it flows to
- **Sea**: the sea where it flows to
- **Length**: the length of the river

Mountain: information about mountains.
- **Name**: the name of the mountain
- **Height**: the height of the mountain
- **Coordinates**: its geographical coordinates as (longitude, latitude)

Lake: information about lakes.
- **Name**: the name of the lake
- **Area**: the total area of the lake

Sea: information about seas.
- **Name**: the name of the sea
- **Depth**: the maximal depth of the sea

Island: information about islands.
- **Name**: the name of the island
- **Islands**: the group of the islands where it belongs to
- **Area**: the total area of the island
- **Coordinates**: its geographical coordinates as (longitude, latitude)

Desert: information about deserts.
- **Name**: the name of the desert
- **Area**: the total area of the desert

geo_river: geographical information about rivers.
- **River**: the name of the river
- **Country**: the country code where it is located
- **Province**: the province of this country

geo_mountain: geographical information about moutains.
- **Mountain**: the name of the mountain
- **Country**: the country code where it is located
- **Province**: the province of this country

geo_lake: geographical information about lakes.
- **Lake**: the name of the lake
- **Country**: the country code where it is located
- **Province**: the province of this country

geo_sea: geographical information about seas.
- **Sea**: the name of the sea
- **Country**: the country code where it is located
- **Province**: the province of this country

geo_island: geographical information about islands.
- **Island**: the name of the island
- **Country**: the country code where it is located
- **Province**: the province of this country

geo_desert: geographical information about deserts.
- **Desert**: the name of the desert
- **Country**: the country code where it is located
- **Province**: the province of this country

merges_with: information about neighboring seas.
- **Sea1**: a sea
- **Sea2**: a sea

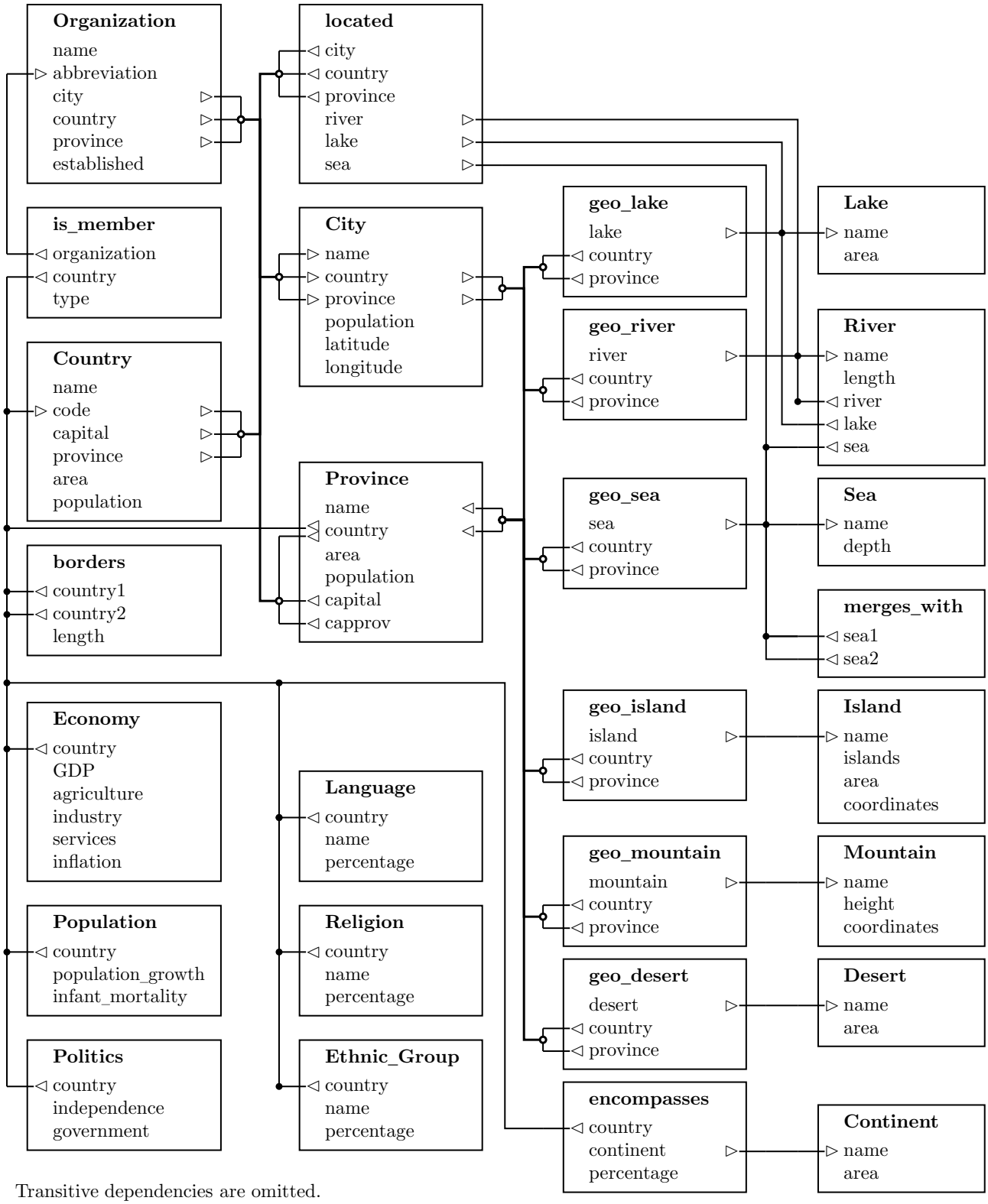


Abbildung A.2: Referential Dependencies of the MONDIAL Database

Das Datenbankschema (ohne Fremdschlüsselbeziehungen) wird unter SQL/Oracle durch das folgende Skript generiert:

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




CREATE TABLE Country (
    Name VARCHAR2(40)
        CONSTRAINT Country_Name_NotNull NOT NULL
        CONSTRAINT Country_Name_Unique UNIQUE,
    Code CHAR(2)
        CONSTRAINT Country_Key PRIMARY KEY,
    Capital VARCHAR2(40),
    Province VARCHAR2(40),
    Population NUMBER
        CONSTRAINT Country_Population_Check CHECK (
             Population >= 0
        ),
    Area NUMBER
        CONSTRAINT Country_Area_Check CHECK (
             Area >= 0
        )
);




CREATE TABLE Province (
    Name VARCHAR2(40),
    Country CHAR(2),
    Population NUMBER
        CONSTRAINT Province_Population_Check CHECK (
             Population >= 0
        ),
    Area NUMBER
        CONSTRAINT Province_Area_Check CHECK (
             Area >= 0
        ),
    Capital VARCHAR2(40),
    CapProv VARCHAR2(40),
    CONSTRAINT Province_Key PRIMARY KEY (Country, Name)
);
```

```
CREATE TABLE City (
    Name VARCHAR2(40),
    Country CHAR(2),
    Province VARCHAR2(40),
    Population NUMBER
        CONSTRAINT City_Population_Check CHECK (
             Population >= 0
        ),
    Latitude NUMBER
        CONSTRAINT City_Latitude_Check CHECK (
             (Latitude >= -90) AND (Latitude <= 90)
        ),
    Longitude NUMBER
        CONSTRAINT City_Longitude_Check CHECK (
             (Longitude >= -180) AND (Longitude <= 180)
        ),
    CONSTRAINT City_Key PRIMARY KEY (Country, Province, Name)
);
```

```
CREATE TABLE Continent (
    Name VARCHAR2(20)
        CONSTRAINT Continent_Key PRIMARY KEY,
    Area NUMBER
        CONSTRAINT Continent_Area_Check CHECK (
             Area >= 0
        )
);
```

```
CREATE TABLE encompasses (
    Country CHAR(2),
    Continent VARCHAR2(20),
    Percentage NUMBER
        CONSTRAINT encompasses_Percentage_Check CHECK (
             (Percentage > 0) AND (Percentage <= 100)
        ),
    CONSTRAINT encompasses_Key PRIMARY KEY (Continent, Country)
);
```

```
CREATE TABLE borders (
    Country1 CHAR(2),
    Country2 CHAR(2),
    Length NUMBER
        CONSTRAINT borders_Length_Check CHECK (
             Length > 0
        ),
    CONSTRAINT borders_Key PRIMARY KEY (Country1, Country2)
);



CREATE TABLE Organization (
    Abbreviation VARCHAR2(15)
        CONSTRAINT Organization_Key PRIMARY KEY,
    Name VARCHAR2(100)
        CONSTRAINT Organization_Name_NotNull NOT NULL
        CONSTRAINT Organization_Name_Unique UNIQUE,
    Established DATE,
    City VARCHAR2(40),
    Province VARCHAR2(40),
    Country CHAR(2)
);



CREATE TABLE is_member (
    Organization VARCHAR2(15),
    Country CHAR(2),
    Type VARCHAR2(30),
    CONSTRAINT is_member_Key PRIMARY KEY (Country, Organization)
);



CREATE TABLE Economy (
    Country CHAR(2)
        CONSTRAINT Economy_Key PRIMARY KEY,
    GDP NUMBER
        CONSTRAINT Economy_GDP_Check CHECK (
             GDP >= 0
        ),
    Agriculture NUMBER,
    Industry NUMBER,
    Services NUMBER,
    Inflation NUMBER
);
```

```
CREATE TABLE Population (
    Country CHAR(2)
        CONSTRAINT Population_Key PRIMARY KEY,
    Population_Growth NUMBER,
    Infant_Mortality NUMBER
);


CREATE TABLE Politics (
    Country CHAR(2)
        CONSTRAINT Politics_Key PRIMARY KEY,
    Independence DATE,
    Government VARCHAR2(120)
);


CREATE TABLE Language (
    Country CHAR(2),
    Name VARCHAR2(50),
    Percentage NUMBER
        CONSTRAINT Language_Percentage_Check CHECK (
             (Percentage > 0) AND (Percentage <= 100)
        ),
    CONSTRAINT Language_Key PRIMARY KEY (Country, Name)
);


CREATE TABLE Religion (
    Country CHAR(2),
    Name VARCHAR2(50),
    Percentage NUMBER
        CONSTRAINT Religion_Percentage_Check CHECK (
             (Percentage > 0) AND (Percentage <= 100)
        ),
    CONSTRAINT Religion_Key PRIMARY KEY (Country, Name)
);


CREATE TABLE Ethnic_Group (
    Country CHAR(2),
    Name VARCHAR2(50),
    Percentage NUMBER
        CONSTRAINT Ethnic_Group_Percentage_Check CHECK (
             (Percentage > 0) AND (Percentage <= 100)
        ),
    CONSTRAINT Ethnic_Group_Key PRIMARY KEY (Country, Name)
);
```

```
CREATE TABLE located (
    City VARCHAR2(40)
        CONSTRAINT located_City_NotNull NOT NULL,
    Province VARCHAR2(40)
        CONSTRAINT located_Province_NotNull NOT NULL,
    Country CHAR(2)
        CONSTRAINT located_Country_NotNull NOT NULL,
    River VARCHAR2(30),
    Lake VARCHAR2(30),
    Sea VARCHAR2(30)
);
```

```
CREATE TABLE River (
    Name VARCHAR2(30)
        CONSTRAINT River_Key PRIMARY KEY,
    River VARCHAR2(30),
    Lake VARCHAR2(30),
    Sea VARCHAR2(30),
    Length NUMBER
        CONSTRAINT River_Length_Check CHECK (
             Length >= 0
        )
);
```

```
CREATE TABLE Mountain (
    Name VARCHAR2(30)
        CONSTRAINT Mountain_Key PRIMARY KEY,
    Height NUMBER
        CONSTRAINT Mountain_Height_Check CHECK (
             Height >= 0
        ),
    Coordinates GeoCoord
        CONSTRAINT Mountain_Coordinates_Check CHECK (
             (Coordinates.Longitude >= -180) AND
             (Coordinates.Longitude <= 180) AND
             (Coordinates.Latitude >= -90) AND
             (Coordinates.Latitude <= 90)
        )
);
```

```
CREATE TABLE Lake (
    Name VARCHAR2(30)
        CONSTRAINT Lake_Key PRIMARY KEY,
    Area NUMBER
        CONSTRAINT Lake_Area_Check CHECK (
             Area >= 0
        )
);


CREATE TABLE Sea (
    Name VARCHAR2(30)
        CONSTRAINT Sea_Key PRIMARY KEY,
    Depth NUMBER
        CONSTRAINT Sea_Depth_Check CHECK (
             Depth >= 0
        )
);


CREATE TABLE Island (
    Name VARCHAR2(30)
        CONSTRAINT Island_Key PRIMARY KEY,
    Islands VARCHAR2(30),
    Area NUMBER
        CONSTRAINT Island_Area_Check CHECK (
             Area >= 0
        ),
    Coordinates GeoCoord
        CONSTRAINT Island_Coordinates_Check CHECK (
             (Coordinates.Longitude >= -180) AND
             (Coordinates.Longitude <= 180) AND
             (Coordinates.Latitude >= -90) AND
             (Coordinates.Latitude <= 90)
        )
);


CREATE TABLE Desert (
    Name VARCHAR2(30)
        CONSTRAINT Desert_Key PRIMARY KEY,
    Area NUMBER
        CONSTRAINT Desert_Area_Check CHECK (
             Area >= 0
        )
);
```

```
CREATE TABLE geo_river (
    River VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_river_Key PRIMARY KEY (Country, Province, River)
);


CREATE TABLE geo_mountain (
    Mountain VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_mountain_Key PRIMARY KEY (Country, Province, Mountain)
);


CREATE TABLE geo_lake (
    Lake VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_lake_Key PRIMARY KEY (Country, Province, Lake)
);


CREATE TABLE geo_sea (
    Sea VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_sea_Key PRIMARY KEY (Country, Province, Sea)
);


CREATE TABLE geo_island (
    Island VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_island_Key PRIMARY KEY (Country, Province, Island)
);


CREATE TABLE geo_desert (
    Desert VARCHAR2(30),
    Country CHAR(2),
    Province VARCHAR2(40),
    CONSTRAINT geo_desert_Key PRIMARY KEY (Country, Province, Desert)
);
```

```
CREATE TABLE merges_with (
    Sea1 VARCHAR2(30),
    Sea2 VARCHAR2(30),
    CONSTRAINT merges_with_Key PRIMARY KEY (Sea1, Sea2)
);
```

Literaturverzeichnis

[CHRS98] A. Christiansen, M. Höding, C. Rautenstrauch, and G. Saake. *Oracle8 effizient einsetzen.* Addison-Wesley, 1998. 87

[Cod70] E. Codd. A Relational Model For Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, 1970. iii

[Dat90] C. J. Date. *An Introduction to Database Systems Volume 1.* Addison-Wesley, 1990. iii

[DD94] C. Date and H. Darwen. *A Guide to the SQL standard: A User's Guide to the Standard Relational Language SQL.* Addison-Wesley, 1994. iii

[DR90] M. Dürr and K. Radermacher. *Einsatz von Datenbanksystemen.* Springer Verlag, 1990. iv

[HCF98] G. Hamilton, R. Cattell, and M. Fisher. *JDBC – Datenbankzugriff mit Java.* Addison-Wesley, 1998. 134

[HV98] U. Hohenstein and V.Pleßer. *Oracle 8.* dpunkt Verlag, 1998. iii

[ISO92] ISO/IEC JTC1/SC21. Information Technology – Database Languages – SQL2, July 1992. ANSI, 1430 Broadway, New York, NY 10018. iii

[ISO94] ISO/IEC JTC1/SC21/WG3. ISO/ANSI working draft Database Languages – SQL3, August 1994. J. Melton (Ed.), ANSI, 1430 Broadway, New York, NY 10018. iii

[KS91] H. F. Korth and A. Silberschatz. *Database System Concepts.* McGraw-Hill, 1991. iii

[MU97] G. Matthiessen and M. Unterstein. *Relationale Datenbanken und SQL: Konzepte der Entwicklung und Anwendung.* Addison-Wesley, 1997. iii

[UW97] J. Ullman and J. Widom. *A First Course in Database Systems.* Prentice Hall, 1997. iii

[Vos94] G. Vossen. *Datenmodelle, Datenbanksprachen und Datenbankmanagement-Systeme.* Addison-Wesley, 1994. iii, 91