Datenbanksysteme (LU 1.0)

Kurzeinführung in SQL und PL/SQL

Nicole Hartmann
Gerhard Krizovsky
Heinrich Moser
Gerald Pfeifer
Katrin Seyr

15. Oktober 2003

Inhaltsverzeichnis

Kapitel 1

Interaktives SQL

Dieses Kapitel bietet eine kurze Einführung in interaktives SQL, erklärt die Benützung von Command-Files und stellt die wichtigsten SQL Befehle vor.

1.1 Allgemeines

1.1.1 Anmerkungen zur Schreibweise

CREATE TABLE Kommandos werden in Großbuchstaben geschrieben.

{**UNION**|**MINUS**|**INTERSECT**} Geschwungene Klammern kennzeichnen alternative Angaben, wovon genau eine auftreten muss. Die verschiedenen Alternativen sind durch „|“getrennt.

[NOT|**AND**|**OR]** Eckige Klammern kennzeichnen optionale Angaben, mehrere Optionen sind auch hier durch „|“ getrennt.

POWER(n,m) Runde Klammern sind Teil des Kommandos.

name, name, . . . Punkte folgen Parameterlisten und zeigen an, dass vorgehende Angaben beliebig wiederholt werden können.

order|**noorder** Wird nichts angegeben, so wird der unterstrichene Wert (= Defaultwert) angenommen.

1.1.2 Online-Dokumentation

Die Beschreibung der SQL-Syntax in diesem Skriptum beschränkt sich auf eine Übersicht über die wichtigsten Befehle. Details und weitereführende Informationen finden Sie in der Oracle Online-Dokumentation (siehe http://minteka.dbai.tuwien.ac.at:8000/doc/).
Die HTML-Version der SQL-Referenz befindet sich auf

`http://minteka.dbai.tuwien.ac.at:8000/doc/server.817/a85397/toc.htm`

1.1.3 Probleme und Fehlermeldungen

Probleme oder Fehlermeldungen, die bei der Verwendung von SQL auftreten, werden in der FAQ-Sektion der Übungshomepage gesammelt und erklärt. Siehe dazu http://www.dbai.tuwien.ac.at/education/dbue/

1.2 SQL*PLUS

SQL*PLUS ermöglicht die interaktive Eingabe von SQL-Statements und bietet außerdem noch Möglichkeiten, Abfrageergebnisse zu formatieren, Optionen zu setzen und SQL-Statements zu editieren beziehungsweise zu speichern.

1.2.1 Aufruf und Beenden von SQL*PLUS

SQL*PLUS wird von der Systemumgebung folgendermaßen aufgerufen:

```
sqlplus /
```

Nach erfolgreichem Login meldet sich SQL*PLUS mit dem Prompt

```
SQL>
```

Nun können folgende Befehle eingegeben werden:

- SQL-Befehle, um auf die Informationen in der Datenbank zuzugreifen. SQL-Befehle werden mit einem „;“ (Semikolon) abgeschlossen. Der betreffende Befehl wird dann umgehend ausgeführt.
- SQL*PLUS-Befehle, die das Arbeiten mit SQL vereinfachen (siehe 1.2.2).

Um SQL*PLUS zu verlassen, verwendet man das EXIT Kommando:

```
SQL> EXIT
```

1.2.2 SQL*PLUS-Befehle

Die wichtigsten Befehle sind:

/: bewirkt die Ausführung des Inhalts im Eingabepuffer. In diesem Puffer befindet sich immer das letzte SQL-Kommando. Mittels dieses Befehles kann dasselbe SQL-Kommando mehrmals ausgeführt werden.

EDIT *filename* ruft einen Editor mit dem File *filename* auf. Der Inhalt von *filename* kann nun editiert werden. Standardmäßig wird der **vi**-Editor verwendet. Wollen Sie lieber mit **emacs** oder **pico** arbeiten, so können Sie die Standardeinstellung ändern, indem Sie in Ihrem Homedirectory ein File namens *login.sql* anlegen und in dieses File die Zeile

DEFINE_EDITOR = emacs

einfügen. Damit wird automatisch beim Start von SQL*PLUS der **emacs**-Editor als Standardeditor gesetzt.

HELP *command* ruft die Online-Hilfe für den Befehl *command* auf.

HOST bzw. **!** *command*. Mit diesem Befehl können Shell-Kommandos ausgeführt werden, ohne dass SQL*PLUS verlassen werden muss. Z.B. zeigen

```
host ls
!ls
```

den Inhalt des aktuellen Directory an.

SET Über **SET** können verschiedene Optionen zur Darstellung der Abfrageergebnisse gesetzt werden.

Der Befehl **SET LINESIZE 200** setzt beispielsweise die Ausgabebreite auf 200 und sorgt damit dafür, dass bei Abfragen mit vielen Feldern weniger Zeilenumbrüche durchgeführt werden.

Der Befehl **SET PAGESIZE 100** sorgt dafür, dass erst nach 100 angezeigten Zeilen (Tupel) die Spaltenüberschriften (Attributnamen) erneut angedruckt werden.

SPOOL *filename* speichert die Ergebnisse von Queries in dem File *filename*. Dieser Befehl ist solange aktiv, bis das Spooling mit **SPOOL OFF** wieder unterdrückt wird. Solange das Spooling aktiv ist, kann das Ergebnisfile nicht ausgelesen werden.

START *filename* führt den Inhalt des Commandfiles *filename* aus.

Wichtig: SQL*PLUS-Befehle benötigen keine Trennzeichen!

1.2.3 Das Command-File

In einem Command-File können mehrere SQL*PLUS- und SQL-Befehle zusammengefasst und dann mit dem SQL*PLUS-Befehl **START** auf einmal ausgeführt werden. Dabei ist folgendes zu berücksichtigen:

1. Das Command-File muss die Extension *.sql* haben.
2. Die einzelnen Befehle müssen mittels eines „;“ (Semikolon) am Zeilenende abgeschlossen werden.

   Beispiel

   ```
   SELECT * FROM table_1; /* Query 1 */
   ⋮
   SELECT * FROM table_n; /* Query n */
   ```

3. Um die Ergebnisse der Queries im Commandfile in eine Datei umzuleiten, empfiehlt es sich, den SQL*PLUS-Befehl **SPOOL** zu verwenden.

   Beispiel

   ```
   SPOOL Ergebnis_file /* Kein ; nach SQL*PLUS-Befehlen!!!
   */
   SELECT * FROM table_1; /* Query 1 */
   ⋮
   SELECT * FROM table_n; /* Query n */
   SPOOL OFF
   ```

   Nachdem das Command-File mit **START** ausgeführt worden ist, können die Ergebnisse mittels

   EDIT *Ergebnis_file*

   in den Editor geladen werden. Achtung: Evtl. wurde an das Ergebnisfile automatisch die Erweiterung **.lst** angehängt.

4. Befinden sich im Commandfile PL/SQL-Blöcke, so muss nach dem `END;` eine einzelne Zeile mit einem `/` stehen, damit der PL/SQL-Modus beendet und der Block ausgeführt wird.

   ```
   DECLARE
   ⋮
   BEGIN
   ⋮
   END;
   /
   ```

1.3 SQL

SQL (Structured Query Language) wurde von ANSI (American National Standards Institute) als Sprachenschnittstelle für relationale Datenbanksysteme genormt und wird im relationalen Datenbankmanagementsystem ORACLE als Datendefinitions- (DDL-Data Definition Language) und Datenmanipulationssprache (DML-Data Manipulation Language) verwendet.
Es wird vorausgesetzt, dass Sie das **SQL**-Kapitel im Vorlesungsskriptum gelesen und verstanden haben. Hier im Übungsskriptum wird auf Eigenheiten und die spezielle Syntax von Oracle eingegangen.

1.3.1 Kurze Einführung

Terminologie

Die Terminologie des Relationenmodells unterscheidet sich von der Terminologie, die in SQL-Befehlen bzw. allgemein in Datenbankmanagementsystemen verwendet wird.

Tabelle Eine Tabelle (*table*) entspricht einer *Relation* im Relationenmodell. Mit den Befehlen **CREATE TABLE**, **ALTER TABLE** und **DROP TABLE** erzeugen, ändern und entfernen Sie Tabellen. In Anhang A sehen Sie, wie mit der **CREATE TABLE**-Anweisung das Relationenschema einer Tabelle festgelegt wird.

Spalte Eine Spalte (*column*, auch *Feld* genannt) entspricht einem *Attribut* im Relationenmodell.

Zeile Eine Zeile (*row*, auch *Datensatz* oder *Record* genannt) entspricht einem *Tupel* im Relationenmodell.

Sicht Eine Sicht (*view*, auch *Abfrage* genannt) entspricht einer *virtuellen Relation*.

Wenn Sie die folgenden Beispiele praktisch durchführen, wird Ihnen die Bedeutung der Begriffe schnell klar werden.

Beispiele mit sqlplus

Die folgenden Beispiele können Sie direkt in sqlplus ausführen, um ein Gefühl für das Arbeiten mit Oracle zu erhalten.
Starten Sie **sqlplus**.

```
uXXXXXXX@minteka:~ > sqlplus /

SQL*Plus: Release 8.1.7.0.0 - Production on <aktuelles Datum>

(c) Copyright 2000 Oracle Corporation.  All rights reserved.


Connected to:
Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
JServer Release 8.1.7.0.0 - Production

SQL>
```

Beginnen wir nun damit, eine Tabelle (*Relation*) und ihre Spalten (*Attribute*) zu erstellen.

```
SQL> CREATE TABLE dept (
  2    deptno NUMBER(2) PRIMARY KEY,
  3    dname  VARCHAR2(14),
  4    loc    VARCHAR2(13)
  5  );

Table created.
```

Die Tabelle *dept* wird mit den Spalten *deptno*, *dname* und *loc* angelegt. Die Tabelle könnte z.B. folgende vier Zeilen enthalten:

```
SQL> INSERT INTO dept VALUES (10, 'ACCOUNTING', 'NEW YORK');

1 row created.

SQL> INSERT INTO dept VALUES (20, 'RESEARCH', 'DALLAS');

1 row created.

SQL> INSERT INTO dept VALUES (30, 'SALES', 'CHICAGO');

1 row created.

SQL> INSERT INTO dept VALUES (40, 'OPERATIONS', 'BOSTON');

1 row created.

SQL> SELECT * FROM dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON
```

Nun können wir eine Sicht auf diese Tabelle erstellen:

```
SQL> CREATE VIEW AbteilungenNordOsten (Abteilungsnummer, Abteilungsname) AS
  2    SELECT DEPTNO, DNAME
  3      FROM dept
  4     WHERE LOC IN ('NEW YORK', 'BOSTON');

View created.

SQL> SELECT * FROM AbteilungenNordOsten;

ABTEILUNGSNUMMER ABTEILUNGSNAME
---------------- --------------
              10 ACCOUNTING
              40 OPERATIONS
```

Natürlich kann eine Sicht auch weit komplexere SELECTs, z.B. solche mit mehreren Tabellen enthalten. Es ist zu beachten, dass die Sicht selbst keine Daten enthält. Jedesmal, wenn Daten der Sicht abgefragt werden, werden diese von ORACLE erneut aus den zugrunde liegenden Tabellen ermittelt.

```
SQL> INSERT INTO dept VALUES (50, 'DEVELOPMENT', 'NEW YORK');

1 row created.

SQL> INSERT INTO dept VALUES (60, 'SUPPORT', 'CHICAGO');
```

```
1 row created.

SQL> SELECT * FROM AbteilungenNordOsten;

ABTEILUNGSNUMMER ABTEILUNGSNAME
---------------- --------------
              10 ACCOUNTING
              40 OPERATIONS
              50 DEVELOPMENT
```

Folgendes Beispiel demonstriert die Verwendung von Aggregatfunktionen und der **UNION**-Klausel:

```
SQL> SELECT 'Niedrigste Abteilungsnummer' Bezeichnung,
  2         MIN(deptno) Wert
  3    FROM dept
  4  UNION
  5  SELECT 'Höchste Abteilungsnummer' Bezeichnung,
  6         MAX(deptno) Wert
  7    FROM dept;

BEZEICHNUNG                       WERT
--------------------------- ----------
Höchste Abteilungsnummer            60
Niedrigste Abteilungsnummer         10
```

Für die folgenden Beispiele verwenden wir die von Oracle mitgelieferten Beispieltabellen. Dazu löschen wir zuerst unsere vorhin erstellte Sicht und Tabelle und rufen dann das Beispielskript auf (beachten Sie, dass **START** ein sqlplus-Befehl ist und daher nicht mit einem Semikolon (;) abgeschlossen werden muss).

```
SQL> DROP VIEW AbteilungenNordOsten;

View dropped.

SQL> DROP TABLE dept;

Table dropped.

SQL> START /public/demobld.sql
Building demonstration tables.  Please wait.
Demonstration table build is complete.
Disconnected from Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
JServer Release 8.1.7.0.0 - Production
```

Nach der Erstellung der Demonstrationstabellen muss sqlplus durch Eingabe von **sqlplus /** wieder neu gestartet werden.
Sehen wir uns nun etwas komplexere Abfragen an: Joins werden in Oracle nicht, wie Sie es vielleicht von anderen DBMS gewohnt sind, in der **FROM**-Klausel mit Schlüsselwörtern wie **INNER**

JOIN, **LEFT OUTER JOIN** usw. sondern über Bedingungen in der **WHERE**-Klausel erzeugt. D.h. im Grunde wird immer ein kartesisches Produkt erstellt, das dann über die **WHERE**-Bedingungen eingeschränkt wird.

```
SQL> /* Beispiel für einen einfachen INNER JOIN */
SQL> SELECT emp.ENAME, dept.DNAME
  2    FROM emp, dept
  3   WHERE emp.DEPTNO = dept.DEPTNO;

ENAME      DNAME
---------- --------------
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER     ACCOUNTING
SMITH      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH
SCOTT      RESEARCH
JONES      RESEARCH
ALLEN      SALES
BLAKE      SALES
MARTIN     SALES

ENAME      DNAME
---------- --------------
JAMES      SALES
TURNER     SALES
WARD       SALES

14 rows selected.
```

Bei einem **OUTER JOIN** wird einfach in der **WHERE**-Klausel neben die Spalte, die **NULL** sein darf, ein **(+)** gesetzt. Die folgende Abfrage selektiert beispielsweise *alle* Abteilungen (departments), auch wenn kein Mitarbeiter (employee) zu dieser Abteilung existiert. Wie Sie sehen, wird bei diesem Beispiel im Gegensatz zur vorherigen Abfrage auch die Abteilung „Operations“ ausgegeben.

```
SQL> /* Beispiel für einen OUTER JOIN */
SQL> SELECT emp.ENAME, dept.DNAME
  2    FROM emp, dept
  3   WHERE emp.DEPTNO (+) = dept.DEPTNO;

ENAME      DNAME
---------- --------------
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER     ACCOUNTING
SMITH      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH
```

```
SCOTT      RESEARCH
JONES      RESEARCH
ALLEN      SALES
BLAKE      SALES
MARTIN     SALES

ENAME      DNAME
---------- --------------
JAMES      SALES
TURNER     SALES
WARD       SALES
           OPERATIONS

15 rows selected.
```

Eine weitere nützliche Technik besteht darin, eine Sicht nicht explizit mit **CREATE VIEW** zu erzeugen sondern einfach als Unterabfrage (*subquery)* direkt in die **FROM**-Klausel zu schreiben. Die folgende Abfrage gibt an, wieviel Mitarbeiter pro Abteilung mehr bzw. weniger als 2000 verdienen.

```
SQL> SELECT view1.deptno, view1.ueber_2000, view2.unter_2000
  2    FROM (SELECT deptno, COUNT(*) ueber_2000
  3            FROM emp
  4           WHERE sal >= 2000
  5           GROUP BY deptno) view1,
  6          (SELECT deptno, COUNT(*) unter_2000
  7            FROM emp
  8           WHERE sal < 2000
  9           GROUP BY deptno) view2
 10   WHERE view1.deptno = view2.deptno;

    DEPTNO UEBER_2000 UNTER_2000
---------- ---------- ----------
        10          2          1
        20          3          2
        30          1          5
```

Unterabfragen können auch in **WHERE**-Klauseln (z.B. **WHERE ... = (SELECT ...)**) und in **SET**-Klauseln von **UPDATE**-Anweisungen (z.B. **SET ... = (SELECT ...)**) verwendet werden.
Nun können Sie die Beispieltabellen wieder entfernen.

```
SQL> START /public/demodrop.sql
Dropping demonstration tables.  Please wait.
Demonstration table drop is complete.
Disconnected from Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
JServer Release 8.1.7.0.0 - Production
```

1.3.2 SQL als Datendefinitionssprache

Mit SQL können Relationen, Indexe, Views (virtuelle Relationen) und Cluster definiert werden. Die SQL-Statements, die Datenbankobjekte erzeugen oder löschen, nennt man DDL-Statements.

Tabellen (Relationen)

Unter SQL besteht eine Datenbank aus einer Menge von Relationenschemata. Ein Relationenschema besteht aus einem Namen, einer Menge von Attributen und deren Wertebereichen. Mit **CREATE TABLE** (siehe 1.4.10) wird ein Relationenschema definiert sowie eine leere Relation zu diesem Schema angelegt.

Views (Benutzersichten)

Relationen werden physisch abgespeichert. Da möglicherweise nicht alle Benutzer einer Datenbank die gleiche Sicht auf den in der Datenbank dargestellten Realitätsausschnitt haben sollen, können mit SQL virtuelle Relationen, sogenannte Views, definiert werden. SQL speichert von Views nur die Definition – die Tupel werden aus den Relationen berechnet. Views werden mit **CREATE VIEW** (siehe 1.4.11) angelegt.
Benutzersichten werden verwendet:

- um für verschiedene Benutzer verschiedene Datensichten zu definieren,
- zur Vereinfachung komplexer Abfragen,
- um die Tupel mehrerer Relationen zu einer virtuellen Relation zusammenzufassen.

Werden in einer View Updates durchgeführt, so wirken sich diese auch auf die darunterliegenden Relationen aus. Allerdings dürfen in ORACLE Updates nur in Views durchgeführt werden, die sich ausschließlich auf eine Relation beziehen. Andernfalls dürfen nur Queries ausgeführt werden.

Indexe und Cluster

Neben den Datenbankobjekten Relationen und Views unterstützt ORACLE noch Cluster und Indexe. Indexe werden verwendet, um eine schnellere Suche nach bestimmten Spalten einer Tabelle (d.h. Attributen einer Relation) zu ermöglichen. Mit dem Befehl **CREATE INDEX** wird ein Index auf bestimmte Spalten einer Tabelle (oder auf einen Cluster) erzeugt. Ein Index ist eine geordnete Liste aller Einträge der Spalten dieser Tabelle. Dadurch wird die Suche nach diesen Einträgen beschleunigt. Einfüge-, Lösch- und Änderoperationen werden jedoch langsamer, da der Index aktualisiert werden muss.
Beispiel: Wenn man in der Tabelle *emp* (siehe Anhang A) nach einem Mitarbeiter mit einem bestimmten Namen (*ename*) sucht, so wird die gesamte Tabelle von oben nach unten durchsucht, bis der Mitarbeiter gefunden wurde. D.h. die Abfrage

> **SELECT** ... **FROM** emp
> **WHERE** ename = 'SCOTT'

kann – bei vielen Datensätzen – lange dauern. Sobald aber mit

> **CREATE INDEX** idx_ename **ON** emp (ename)

ein Index (d.h. eine nach *ename* sortierte Liste) angelegt wurde, wird – bei der gleichen Abfrage wie oben – automatisch eine binäre Suche, die natürlich viel schneller ist, durchgeführt. Das geschieht, ohne dass der Datenbank-Benutzer etwas an seinem **SELECT**-Statement ändern muss.

Cluster werden eingesetzt, um die Daten einer oder mehrerer Tabellen so zu strukturieren, dass sie physikalisch nahe beieinander gespeichert werden. Dies kann vor allem bei häufigen Joins zu Geschwindigkeitsvorteilen führen.
Praktisch sieht das so aus, dass Sie zuerst mit **CREATE CLUSTER** einen Cluster anlegen und dann mit **CREATE TABLE** Tabellen in diesen Cluster setzen. Zum Schluss legen Sie mit **CREATE INDEX** einen Index auf den Cluster. Ein Beispiel finden Sie in Kapitel 1.4.7.
Sowohl Cluster als auch Indexe sind für den Benutzer transparent, man spricht in diesem Zusammenhang vom Konzept der physischen Datenunabhängigkeit.

Details über diese beiden Konzepte finden Sie in „Oracle8i Concepts“, Kapitel 10 „Schema Objects“[1]. Die Kenntnis des entsprechenden Kapitels im Vorlesungsskriptum, „Physisches Datenbankdesign“, wird vorausgesetzt.

Datentypen

Datentypen definieren den Wertebereich eines Attributes und die darauf anwendbaren Operationen (z.B. Addition, Verkettung, ...). ORACLE unterstützt folgende Datentypen:

Vordefinierte Datentypen

VARCHAR2(*n*) steht für einen String mit variabler Länge, die kleiner oder gleich *n* ist. *n* muss im Bereich von 1 bis 4000 liegen. Ist der String leer, wird automatisch ein Leerzeichen eingefügt. Will man überprüfen, ob ein String leer ist, muss der **IS NULL** Operator (siehe 1.3.3) verwendet werden.

CHAR(*n*) steht für einen String mit der fixen Länge n. Besteht ein String aus weniger als *n* Zeichen werden die restlichen Zeichen mit Leerzeichen aufgefüllt. Die maximale Größe beträgt 2000 Zeichen. Um zu überprüfen, ob ein String leer ist, muss der **IS NULL** Operator (siehe 1.3.3) verwendet werden.

LONG(*n*) In einer Variable dieses Typs kann ein String mit bis zu $2^{31} - 1$ Zeichen gespeichert werden. In einer Relation darf nur ein Attribut dieses Typs vorkommen.

NUMBER steht für Fließkommazahlen mit maximal 38 Stellen (sowohl Vor- als auch Nachkommastellen) im Intervall zwischen 10^{-129} und $9.99 * 10^{125}$.

NUMBER(*p,s*) steht für Fließkommazahlen mit *p* Vorkomma- und *s* Nachkommastellen im Intervall zwischen 10^{-129} und $9.99 * 10^{125}$. Die dezimale Genauigkeit *p* liegt zwischen 1 und 38, *s* liegt zwischen –84 und 127. Ist *s* negativ wird die Zahl links vom Komma um *s* Stellen gerundet. Ist

[1] `http://minteka.dbai.tuwien.ac.at:8000/doc/server.817/a76965/c08schem.htm`

s zum Beispiel -2 wird auf Hunderterstellen gerundet (d.h. 765483,678 wird als 765500 in der Datenbank gespeichert).

NUMBER(*p*) steht für Fließkommazahlen mit *p* Vorkomma– und keinen Nachkommastellen im Intervall zwischen 10^{-129} und $9.99 * 10^{125}$. Die dezimale Genauigkeit *p* liegt zwischen 1 und 38.

FLOAT Dieser Datentyp entspricht dem ANSI Standard für Fließkommazahlen.
FLOAT und FLOAT(*) sind äquivalent und haben eine Genauigkeit von 126 Binärstellen.
Mit **FLOAT**(*binary_precision*) kann man die maximale Genauigkeit in Binärstellen angeben. *binary_precision* kann zwischen 1 und 126 liegen.

INTEGER steht für ganze Zahlen, die mit oder ohne Vorzeichen angegeben werden können. Wird kein Vorzeichen angegeben ist die Zahl automatisch positiv.

DATE wird verwendet, um Datums– und Zeitinformation zu speichern. Ein Datum hat normalerweise das Format DD-MMM-YYYY wie z.B. in 13-NOV-2002, eine Zeitangabe hat das Format HH:MI:SS a.m. (oder p.m.). Will man eine Zeitangabe eingeben oder vom Standarddatumformat abweichen, so muss man die **TO_DATE**-Funktion mit einer Formatmaske benutzen (siehe 1.3.4 oder auch das Online Reference Manual).

Man beachte, dass der Leerstring und NULL von ORACLE als gleich behandelt werden, d.h. ein VARCHAR2, der kein NULL erlaubt, kann auch keinen Leerstring enthalten. (Laut ORACLE-Doku kann sich das jedoch in zukünftigen Versionen ändern.)
Weiters existieren noch Datentypen zum speichern (großer) binärer Daten oder Texte, deren Aufzählung aber den Rahmen dieses Skriptums sprengen würde.

1.3.3 Operatoren

Arithmetische Operatoren

+, - Vorzeichen für eine positive oder negative Zahl, sowie Additions- als auch Subtraktionszeichen.

***, /** Operatoren für Multiplikation und Division. Es gilt Punkt- vor Strichrechnung.

Vergleichsoperatoren

= Gleichheitsoperator

!= Ungleichheitoperator

$<$**,** $>$ Größer- Kleiner als

$<=$**,** $>=$ Größer- Kleinergleich

IN wählt alle Tupel aus, die in einer bestimmten Menge vorkommen.

Beispiel:
SELECT * FROM emp
WHERE job **IN** ('CLERK', 'ANALYST')

Es werden alle Tupel ausgewählt, die als Wert des Attributes Job 'CLERK' oder 'ANALYST' haben.

NOT IN wählt die Komplementärmenge zu **IN** aus.

ANY vergleicht einen Wert mit jedem anderen Wert einer Query oder Liste, wobei immer ein Vergleichsoperator (=, !=, <, >, <=, >=) vorangestellt sein muss. Der Ausdruck ist **TRUE**, wenn die Bedingung mit *mindestens einem* (**ANY**) Wert aus der Liste erfüllt ist.

Beispiel:
SELECT * FROM emp
WHERE sal >= **ANY**
(**SELECT** sal **FROM** emp, dept
WHERE emp.deptno = dept.deptno **AND** dname = 'RESEARCH');

Ein Mitarbeiter wird selektiert, wenn *mindestens ein* (**ANY**) Mitarbeiter der Research-Abteilung existiert, der genausoviel oder weniger verdient.

= ANY verhält sich genauso wie **IN**.

ALL vergleicht einen Wert mit jedem anderen Wert einer Query oder einer Liste, wobei immer ein Vergleichsoperator (=, !=, <, >, <=, >=) vorangestellt sein muss. Der Ausdruck ist **TRUE**, wenn die Bedingung mit *allen* (**ALL**) Werten aus der Liste erfüllt ist.

Beispiel:
SELECT * FROM emp
WHERE sal >= **ANY**
(**SELECT** sal **FROM** emp, dept
WHERE emp.deptno = dept.deptno **AND** dname = 'RESEARCH');

Ein Mitarbeiter wird selektiert, wenn *alle* (**ALL**) Mitarbeiter der Research-Abteilung weniger oder genausoviel wie er verdienen.

!= ALL verhält sich genauso wie **NOT IN**.

[NOT] BETWEEN überprüft, ob ein Wert zwischen zwei anderen Werten liegt (bzw. nicht liegt).

Beispiel:
SELECT * FROM emp
WHERE sal

BETWEEN 2000 **AND** 3000;

[NOT] EXISTS überprüft, ob ein Wert in einer Subquery (nicht) enthalten ist.

Beispiel:
SELECT deptno
FROM dept
WHERE EXISTS
(**SELECT** * **FROM** emp
WHERE dept.deptno = emp.deptno);

Es werden alle Abteilungen ausgewählt, die mindestens einen Mitarbeiter haben.

LIKE Der LIKE Operator ist ziemlich mächtig und erlaubt die Angabe von Vergleichsmustern. Das Prozentzeichen (**%**) wird als Substitut (wild card) benutzt, um beliebige Zeichenfolgen (also auch die leere Zeichenfolge) zu repräsentieren. Der Unterstrich (_) steht für genau ein beliebiges Zeichen.

Beispiel:
SELECT * FROM emp
WHERE job **LIKE** '%r';

Liefert alle Tupel, deren Job mit „r" endet (z.B. Maler, Dachdecker, oder auch nur „r")

IS [NOT] NULL überprüft, ob einem Attribut einen Wert zugewiesen wurde.

Beispiel:
SELECT ename, deptno
FROM emp
WHERE comm **IS NULL**;

Liefert alle Tupel deren Attribut *comm* leer ist. Ein Leerstring wird in Oracle auch als **NULL** behandelt.

Logische Operatoren

NOT liefert **TRUE**, wenn die darauffolgende Bedingung falsch ist

Beispiel:
SELECT * FROM emp
WHERE NOT (job **IS NULL**);

Liefert alle Mitarbeiter, die einen Job haben.

AND liefert **TRUE**, wenn beide Teile einer Bedingung erfüllt sind.

Beispiel:
SELECT * **FROM** emp
WHERE job = 'CLERK'
AND deptno = 10;

Liefert alle Mitarbeiter mit dem Job 'Clerk' und der Abteilungsnummer 10.

OR liefert **TRUE**, wenn einer der beiden Teile einer Bedingung erfüllt ist

Beispiel:
SELECT * **FROM** emp
WHERE job = 'CLERK'
OR deptno = 10;

Liefert sämtliche Tupel zurück, für die gilt, dass der Job = 'CLERK' oder dass die deptno = 10 ist; liefert auch alle Tupel, für die beide Bedingungen gelten.

Set Operatoren

Setoperatoren sind Operatoren, die die Abfrageergebnisse zweier Queries in einer Ergebnistabelle zusammenfassen. Beide Queries müssen dieselbe Anzahl an Attributen in ihrer Ergebnisrelation haben.

UNION erzeugt die Vereinigung mehrerer Abfragen.

Beispiel:
SEL¯ECT Betrag, 'Aufwand'
FROM Kosten
WHERE Monat $<=$ 6 **AND** Jahr $=$ 1993
UNION
SELECT Höhe, 'Ertrag'
FROM Einnahmen
WHERE Monat $<=$ 6 **AND** Jahr $=$ 1993;

Liefert als Ergebnis eine Liste von Aufwendungen und Erträgen. Die zweite Spalte – der Name wird von ORACLE vergeben – zeigt an, aus welcher Relation der Wert stammt.

Betrag	'Aufwand'
3.4	Ertrag
4.6	Aufwand
12.5	Aufwand
17.4	Ertrag

INTERSECT erzeugt den Durchschnitt mehrerer Abfragen.

MINUS erzeugt die Differenzmenge zweier Abfragen.

Sonstige Operatoren

(+) Operator um einen Outer Join zu erzeugen, d.h. dass das Attribut, bei dem dieser Operator hinzugefügt wird, auch NULL-Werte annnehmen darf.

Beispiel:
SELECT ename, job, dept.deptno
FROM emp, dept
WHERE emp.deptno (+) = dept.deptno

Ergebnistabelle:

ENAME	JOB	DEPTNO
CLARK	MANAGER	10
KING	PRESIDENT	10
SMITH	CLERK	10
ADAMS	CLERK	20
FORD	CLERK	20
SCOTT	ANALYST	20
JONES	ANALYST	20
ALLEN	MANAGER	20
BLAKE	SALESMAN	30
MARTIN	MANAGER	30
JAMES	CLERK	30
TURNER	SALESMAN	30
WARD	SALESMAN	30
		40

Man beachte das letzte Tupel (deptno = 40) ! Obwohl es in der Tabelle *emp* keine Einträge zu der Abteilung „40“ gibt wird dieses Tupel trotzdem ausgegeben.

Operator || führt eine Verkettung von Strings durch.

1.3.4 Funktionen in SQL

Skalare Funktionen

Skalare Funktionen liefern für jede Zeile einer Abfrage eine Zeile in der Ergebnistabelle zurück. Skalare Funktionen können in **SELECT**-Statements (siehe 1.4.21) und **WHERE** Bedingungen verwendet werden.

LENGTH liefert die Länge eines Strings.

Beispiel: SELECT LENGTH('CANDIDE') **FROM** DUAL[2];

NVL wandelt einen Wert, wenn er NULL ist, in einen anderen Wert um.

Beispiel:
SELECT Name, NVL(Telefonnummer, 'kein Telefon') **FROM** Person;

Selektiert alle Personen mit Telefonnummer. Wenn eine Person keine Telefonnummer hat, wird stattdessen 'kein Telefon' ausgegeben.

TO_CHAR wandelt eine Zahl oder ein Datum in einen String um.

Der erste Parameter ist der zu konvertierende Wert, der zweite Parameter eine optionale Formatierungszeichenfolge (siehe TO_DATE und TO_NUMBER).

TO_DATE wandelt einen String in ein Datum um.

Der erste Parameter stellt den Wert, der zweite Parameter das Format des Wertes dar. Es empfiehlt sich, bei dieser Funktion das ISO-Format für Datums- (`YYYY-MM-DD`) und Zeitangaben (`YYYY-MM-DD HH24:MI:SS`) zu verwenden. (Man beachte, dass Minuten mit MI und nicht mit MM abgekürzt werden!) Damit vermeidet man alle Probleme, die mit Datumsfeldern auftreten können (englische Monatsnamen, 12-Stunden-System, zweistellige Jahreszahlen, amerikanische Datumsschreibweise mit Monat vor Tag usw.)

Beispiel:
INSERT INTO BIRTHDAYS (BNAME, BDAY) **VALUES**
('ANNIE', TO_DATE('2002-11-13 10:56:00', 'YYYY-MM-DD HH24:MI:SS'));

Eine genaue Beschreibung der möglichen Formatierungszeichenfolgen finden Sie unter `http://minteka.dbai.tuwien.ac.at:8000/doc/server.817/a85397/sql_elem.htm#35921`

TO_NUMBER wandelt einen String in eine Zahl um.

Auch hier kann als zweiter Parameter optional das Format des Strings angegeben werden. Eine genaue Beschreibung der möglichen Formatierungszeichenfolgen finden Sie unter `http://minteka.dbai.tuwien.ac.at:8000/doc/server.817/a85397/sql_elem.htm#34597`

USER gibt den aktuellen ORACLE-User aus.

Beispiel: SELECT USER **FROM** DUAL;

[2] **DUAL** ist eine Hilfsrelation von Oracle, die immer genau einen Datensatz enthält. Sie kann verwendet werden, um mit SELECT Daten zu selektieren, die nicht aus einer Relation stammen sondern berechnet werden.

Aggregatfunktionen

Aggregatfunktionen liefern eine Ergebniszeile für eine oder mehrere Abfragezeilen zurück. Die meisten Aggregatfunktionen akzeptieren folgenden Optionen:

DISTINCT liefert Tupel, die genau gleich sind nur einmal zurück.

ALL liefert alle Tupel, die eine Abfrage ergibt, auch wenn diese öfters vorkommen. Dies ist die Defaulteinstellung.

Aggregatfunktionen können nur im **SELECT**-Statement (siehe 1.4.21) oder in der **HAVING** Klausel (siehe 1.4.21) einer Anweisung vergewendet werden. Die Syntax ist bei allen hier vorgestellten Aggregatfunktionen gleich.

Die Aggregatfunktionen selbst sind:

COUNT gibt die Anzahl der Tupel an, die nicht NULL sind. (Eine Ausnahme bildet **COUNT(*)**.)

Beispiel:
SELECT COUNT (**DISTINCT** job) AnzahlJobs
FROM emp;

Liefert die Anzahl aller Jobs, die in der Ergebnistabelle vorkommen, wobei Jobs, die öfters vorkommen, nur einmal aufgezählt werden.

AVG berechnet den Durchschnitt einer Spalte aus Zahlen.

MAX liefert den größten Wert einer Spalte.

MIN liefert den kleinsten Wert einer Spalte.

SUM liefert die Gesamtsumme einer Spalte aus Zahlen.

1.3.5 SQL als Datenmanipulationssprache

Nachdem man mit **CREATE TABLE** (siehe 1.4.10) eine Relation erzeugt hat, kann man Daten interaktiv mit **INSERT** (siehe 1.4.17) in die Relation einlesen. Mit dem **SELECT**-Statement (siehe 1.4.21) können bestehende Daten abgefragt werden, zum Ändern und Löschen dienen die Befehle **UPDATE** (siehe 1.4.22) und **DELETE** (siehe 1.4.12).

1.3.6 Transaktionsmanagement

Die folgende Beschreibung gibt eine kurze Einführung in das Transaktionsmanagement von ORACLE. Für genauere Informationen lesen Sie bitte die entsprechenden Abschnitte in „Oracle8i Concepts“, Kapitel 16: „Transaction Management“[3].

[3] `http://minteka.dbai.tuwien.ac.at:8000/doc/server.817/a76965/c15trans.htm`

Das Transaktionsmanagement eines Datenbankmanagementsystemes auf Multiusersystemen regelt den parallelen Zugriff mehrerer Benutzer auf die Datenbank. Eine Transaktion besteht meist aus mehreren SQL–Statements, kann aber auch nur aus einem Statement bestehen. Mit dem ersten ausführbaren SQL-Statement nach einem **COMMIT**, **ROLLBACK** oder einer Anbindung an die Datenbank wird eine Transaktion begonnen. Von nun an gehören alle SQL-Statements zu dieser Transaktion, bis sie beendet wird. Eine Transaktion kann beendet werden durch:

- ein **COMMIT**-Statement (siehe 1.4.4). Damit werden alle Änderungen, die seit dem Beginn dieser Transaktion durchgeführt wurden, permanent in die Datenbank eingetragen. Mit dem nächsten ausführbaren SQL-Statement beginnt eine neue Transaktion.

- ein **ROLLBACK**-Statement (siehe 1.4.19). Damit werden alle Änderungen, die seit dem Beginn dieser Transaktion durchgeführt wurden, rückgängig gemacht.

- ein DDL (= Data Definition Language)-Statement (z.B. **CREATE**, **DROP**, **RENAME**). Bevor ein DDL-Statement ausgeführt wird, wird eine allenfalls begonnene Transaktion automatisch beendet, und es werden alle bisherigen Änderungen in die Datenbank eingetragen. DDL-Statements sind immer Transaktionen, die nur aus diesem einen Befehl bestehen. Wurde ein DDL-Statement erfolgreich beendet, so beginnt mit dem nächsten Statement wieder eine neue Transaktion.

- das reguläre Beenden von SQL*PLUS. Damit werden alle Änderungen, die seit dem Beginn der aktuellen Transaktion durchgeführt wurden, permanent in die Datenbank eingetragen.

- ein abnormales Ende von SQL*PLUS. Damit werden alle Änderungen, die seit dem Beginn der aktuellen Transaktion durchgeführt wurden, rückgängig gemacht.

Eine Transaktion kann auch bis zu einem Synchronisationspunkt (mit dem **SAVEPOINT**-Statement deklariert, siehe 1.4.20) zurückgesetzt werden (**ROLLBACK TO**-Statement). Dadurch wird die Wirkung all jener Statements aufgehoben, die zwischen dem **SAVEPOINT**– und dem **ROLLBACK TO**-Statement ausgeführt wurden. Die Transaktion wird dadurch nicht beendet.

1.4 Syntax der SQL-Statements

In der folgenden Beschreibung sind nur die SQL-Befehle berücksichtigt, die bei der interaktiven Verwendung von SQL im Rahmen der Übung von Bedeutung sind.

1.4.1 /* . . . */

Funktion: Fügt einen Kommentar ein.

Syntax: **/*** *text* ***/**

Beschreibung: Kommentare können innerhalb jedes SQL-Statements vorkommen und sich über mehrere Zeilen erstrecken. Dabei ist darauf zu achten, dass nach (vor) dem „/*“ („*/“) ein Leerzeichen gesetzt wird, da ansonsten die letzte Abfrage erneut ausgeführt wird.

Beispiel:

SELECT ename, sal + comm comp, job, loc
/* Select all employees whose compensation is
greater than that of Jones. */
FROM emp, dept /* dept is used to get the department name. */
WHERE emp.deptno = dept.deptno
AND sal + comm > /* Subquery */
(**SELECT** sal + comm /* comp = sal + comm */
FROM emp /* Jones's */
WHERE ename = 'Jones'); /* compensation */

1.4.2 ALTER TABLE

Funktion: Ändert eine Relation.

Syntax:

ALTER TABLE *table*
ADD (*table_constraint* [, *table_constraint*] ...)

bzw.

ALTER TABLE *table*
ADD (*column datatype* [**DEFAULT** *expr*] [*column_constraints*]
[, *column datatype* [**DEFAULT** *expr*] [*column_constraints*]] ...)

bzw.

ALTER TABLE *table*
MODIFY (*column datatype* [**DEFAULT** *expr*] [*column_constraints*]
[, *column datatype* [**DEFAULT** *expr*] [*column_constraints*]] ...)

bzw.

ALTER TABLE *table*
DROP [*column_constraint* | *table_constraint*]

Schlüsselwörter und Parameter:

table ist der Name der Relation.

column ist der Name eines der Attribute der Relation.

datatype ist der Datentyp des Attributes (siehe 1.3.2).

column_constraint definiert eine Wertebereichseinschränkung für ein Attribut (siehe 1.4.6).

table_constraint definiert eine Wertebereichseinschränkung für eines oder mehrere Attribute (siehe 1.4.6).

DEFAULT legt den Wert des Attributes fest, wenn dieser bei einem **INSERT** nicht angegeben wird.

Beschreibung: Über diese Anweisungen können Attribute hinzugefügt und geändert werden; Constraints können hinzugefügt und entfernt werden. Weitere Möglichkeiten der **ALTER TABLE**-Anweisung finden Sie in der SQL-Referenz der Oracle Online Hilfe.

Beispiele:

ALTER TABLE dept **ADD** UNIQUE (dname);

ALTER TABLE emp
ADD (thriftplan NUMBER(7,2), loancode CHAR(1) NOT NULL);

ALTER TABLE emp
MODIFY (thriftplan NUMBER(9,2));

ALTER TABLE dept
DROP UNIQUE (dname);

1.4.3 Expression (expr, expr-list)

Funktion: Repräsentiert einen String, eine Zahl, ein Datum oder einen Ausdruck.

Syntax:

Form I: Ein Attribut, eine Konstante oder ein spezieller Wert.

[table.] column
text
number
NULL

Beispiel:

emp.ename
'this is a text string'
10
NULL

Form II: Eine Hostvariable in Embedded SQL-Statements.

:variable

Form III: Eine Funktion.

function_name ([**DISTINCT**| **ALL**] expr [,expr]. . .)

Beispiel:
AVG(value)
LENGTH('BLAKE')

Beschreibung: Die für die Übung relevanten (Gruppierungs)Funktionen sind:

COUNT([**DISTINCT** | **ALL**] *expr*) . . . Anzahl der Tupel in der Gruppe
LENGTH[*expr*) . . . liefert die Länge eines Strings.
SUM(*expr*) . . . Der Wert von *expr* wird für alle Tupel summiert.
AVG(*expr*) . . . Durchschnitt (siehe 1.3.4)
MAX(*expr*) . . . Maximum (siehe 1.3.4)
MIN(*expr*) . . . Minimum (siehe 1.3.4)

Form IV: Eine Kombination anderer Ausdrücke (in aufsteigender Präzedenzordnung).

(expr)
+expr, −expr
expr ∗ expr, expr / expr
expr + expr, expr − expr, expr || expr

Beispiel:
('Clark'||'Smith')
LENGTH('Moose')∗57
SQRT(144)+72

Beschreibung: Der Operator || führt eine Verkettung von Strings durch.

Form V: (expr-list) Eine geklammerte Liste von Ausdrücken.

(expr [,expr]. . .)

Beispiel:
(10, 20, 40)
('Scott', 'Blake', 'Taylor')
(**LENGTH**('Moose')*57, −SQRT(144)+72, 69)

Beschreibung Expressions werden in

- der **SELECT**-Liste von **SELECT**-Statements (siehe 1.4.21),
- in Conditions (siehe 1.4.5) der **WHERE**– und **HAVING**-Klauseln,
- in **ORDER BY**-Klauseln (siehe 1.4.21),
- in der **VALUES**-Klausel des **INSERT**-Statements (siehe 1.4.17),
- in der **SET**-Klausel des **UPDATE**-Statements (siehe 1.4.22) verwendet.

1.4.4 COMMIT

Funktion: Übernimmt alle Änderungen der aktuellen Transaktion ”permanent“ in die Datenbank, löscht alle Synchronisationspunkte, beendet die Transaktion und gibt alle Locks der Transaktion frei.

Syntax: COMMIT [WORK]

Schlüsselwörter und Parameter:

WORK ist optional und dient der ANSI Kompatibilität.

Beschreibung: Es empfiehlt sich in der Praxis, Transaktionen mit Hilfe von **COMMIT** explizit zu beenden. Bei einem unvorhergesehenen Ausstieg aus SQL*PLUS wird die letzte nicht mit **COMMIT** bestätigte Transaktion mit Hilfe von **ROLLBACK** (siehe 1.4.19) zurückgesetzt.
Bei normalem Beenden von SQL*PLUS wird die letzte Transaktion mit **COMMIT** bestätigt. Erst nach einem **COMMIT** werden die Datenbankänderungen auch für andere Benutzer sichtbar. Das Schlüsselwort **WORK** ist optional und ändert nichts an der Funktionalität, es dient lediglich der Gewährleistung der ANSI Kompatibilität.

Beispiel:
INSERT INTO dept **VALUES** (’50’, ’Marketing’, ’Tampa’)
COMMIT WORK

1.4.5 Condition

Funktion: Ist ein Element in **SELECT**– (siehe 1.4.21), **INSERT**– (siehe 1.4.17), **UPDATE**– (siehe 1.4.22) und **DELETE**-Statements (siehe 1.4.12), das entweder **TRUE** oder **FALSE** ergibt.

Syntax:

Form I: Ein Vergleich mit einem Ausdruck oder einem Query-Ergebnis.

<expr> <comparison operator> <expr>
<expr> <comparison operator> (<subquery>)
<expr-list> <equal-or-not> (<subquery>)

Form II: Vergleich mit einem beliebigen oder allen Elementen einer Liste oder Query.

<expr> <comparison operator> { **ANY** | **ALL** } <expr-list>
<expr> <comparison operator> { **ANY** | **ALL** } (<subquery>)
<expr-list> <equal-or-not> { **ANY** | **ALL** } (<expr-list> [, <expr-list>] . . .)
<expr-list> <equal-or-not> { **ANY** | **ALL** } (<subquery> [,<subquery>] . . .)

Form III: Testen des Vorkommens in einer Liste oder Subquery.

<expr> **[NOT] IN** <expr-list>
<expr> **[NOT] IN** (<subquery>)
<expr-list> **[NOT] IN** (<expr-list> [,<expr-list>] . . .)
<expr-list> **[NOT] IN** (<subquery> [,<subquery>] . . .)

Form IV: Bereichsüberprüfung.

<expr> **[NOT] BETWEEN** <expr> AND <expr>

Form V: Testen auf NULL-Wert.

<expr> **IS [NOT] NULL**

Form VI: Testen auf Vorkommen von Tupel in einer Subquery.

EXISTS (<subquery>)

Form VII: Pattern-Matching

<string1> **[NOT] LIKE** <string2>

Form VIII: Eine Kombination von anderen Bedingungen (aufgelistet in Präzedenzordnung).

(<condition>)
NOT <condition>
<condition> { **AND** | **OR** } <condition>

Beispiel:

ename = 'Smith'
emp.deptno = dept.deptno
hiredate > '01-Jan-88'
job **IN** ('President', 'Clerk', 'Analyst')
sal **BETWEEN** 500 **AND** 1000
comm **IS NULL AND** sal = 2000

1.4.6 Constraint clause

Funktion: Schränkt den gültigen Wertebereich für ein Attribut bzw. für eine Gruppe von Attributen ein. Wird im **CREATE TABLE**-Statement (siehe 1.4.10) verwendet.

Syntax Table Constraint:

[**CONSTRAINT** *constraint*]
{ {**UNIQUE** | **PRIMARY KEY**} (*column* [,*column*] . . .)
| **FOREIGN KEY** (*column* [,*column*] . . .)
REFERENCES *table* [(*column* [,*column*] . . .)] **[ON DELETE CASCADE]**
| **CHECK** (*condition*) }

Syntax Column Constraint:

[**CONSTRAINT** *constraint*]
{ **[NOT] NULL**
| { **UNIQUE** | **PRIMARY KEY** }
| **REFERENCES** *table* [(*column*)] **[ON DELETE CASCADE]**
| **CHECK** (*condition*) }

Schlüsselwörter und Parameter:

column spezifiziert den Namen des Attributes, auf das sich die Einschränkung bezieht.

constraint gibt den Namen der Einschränkung an.

NULL|NOT NULL gibt an, ob ein Attribut Nullwerte haben darf oder nicht.

UNIQUE erzwingt eindeutige Attributwerte.

PRIMARY KEY identifiziert ein Attribut als Primärschlüssel.

FOREIGN KEY (*column* . . .) REFERENCES *table* (*column* . . .) [ON DELETE CASCADE] definiert ein Attribut als Fremdschlüssel aus der Relation *table*. Defaultmäßig werden die Attribute des Primärschlüssels der Relation *table* angenommen. Die Klausel ON DELETE CASCADE dient zur Gewährleistung der referentiellen Integrität. Wird in der referenzierten Tabelle ein Tupel gelöscht, werden die referenzierenden Tupel ebenfalls gelöscht.

CHECK *condition* gibt eine Bedingung (siehe 1.4.5) an, die ein Attribut erfüllen muß, damit ein Tupel in die Relation aufgenommen wird.

Beschreibung: Es gibt zwei Arten von Constraints, Table–Constraints und Column–Constraints. Sie unterscheiden sich darin, dass ein Column–Constraint sich nur auf ein Attribut bezieht, während ein Table-Constraint sich auf ein oder mehrere Attribute bezieht. Das Benennen von Constraints ist optional, ermöglicht es aber, den Constraint mit **ALTER TABLE** nachträglich zu entfernen. Constraints werden mit Hilfe des **CREATE TABLE**-Statements (siehe 1.4.10) bzw. des **ALTER TABLE**-Statements (siehe 1.4.2) erstellt.

Ein **NULL**-Constraint darf nur in einem Column Constraint verwendet werden. **NULL** bedeutet, dass ein Attribut einen Nullwert für jedes beliebige Tupel einer Relation beinhalten darf. **NOT NULL** besagt wiederum, dass jedes Tupel der Relation für das Attribut einen von **NULL** unterschiedlichen Wert haben muss. Defaultmäßig ist jedes Attribut auf **NULL** gesetzt.
Ein **UNIQUE**-Constraint erzwingt, dass jedes Tupel in der Relation einen eindeutigen Wert für das Attribut hat, das Attribut **NOT NULL** deklariert wird und kein PRIMARY KEY sein darf. Ein **PRIMARY KEY**-Constraint wird verwendet um ein Tupel einer Relation eindeutig zu identifizieren. Diese Einschränkung ist eine Obermenge des **UNIQUE**-Constraints und erzwingt daher implizit Attribute, die **NOT NULL** deklariert sind. Das Attribut darf nicht als **UNIQUE** deklariert werden. Besteht der Primärschlüssel aus mehr als einem Attribut, so muß für die Definition ein Table–Constraint verwendet werden.
Ein **FOREIGN KEY/REFERENCES**-Constraint gibt an, aus welcher Relation ein Fremdschlüssel stammt.
Ein **CHECK**-Constraint dient dazu, nur solche Tupel aufzunehmen, die die **condition** (siehe 1.4.5) erfüllen. In einem Column–Constraint darf sich diese Einschränkung nur auf das betreffende Attribut beziehen, in einem Table–Constraint können mehrere Attribute in der Bedingung vorkommen.

Beispiel:

CREATE TABLE emp
(empno **NUMBER(5) CONSTRAINT** unq_constraint **UNIQUE**,
ename **CHAR(9) CONSTRAINT** nn_constraint **NOT NULL**,
deptno **NUMBER CONSTRAINT** fk_deptno **REFERENCES** dept (deptno));

CREATE TABLE ship_container
(ship_no **NUMBER NOT NULL**,
container_no **NUMBER NOT NULL**,
PRIMARY KEY (ship_no, container_no));

CREATE TABLE dept
(deptno **NUMBER CHECK** (deptno BETWEEN 10 AND 99),
dname **CHAR(9) CHECK** (dname = UPPER(dname)),
loc **CHAR(10) CHECK** (loc IN ('Dallas', 'Boston', 'New York')));

1.4.7 CREATE CLUSTER

Funktion: Definiert einen Cluster, der eine oder mehrere Relationen enthalten kann.

Syntax:

CREATE CLUSTER *cluster* (*column datatype* [, *column datatype*] ...)

Schlüsselwörter und Parameter:

cluster ist der Name des Clusters.

column ist der Name eines der Attribute, die den Clusterkey bilden (siehe 1.3.2).

datatype ist das Format des Attributs (siehe 1.3.2).

Die Tupel der am Cluster beteiligten Relationen werden physikalisch zusammenliegend gespeichert Bei Joins zwischen diesen Relationen wird die Zugriffszeit beschleunigt. Jeder Clusterkeywert wird nur einmal abgespeichert. Generell sollten Relationen zu Clustern zusammengefasst werden, wenn sie häufig mit einem Join verknüpft werden. Allerdings wird durch einen Cluster auch die Geschwindigkeit bei einem kompletten Suchlauf durch eine der Relationen des Clusters reduziert.
Die Relationen werden einem Cluster mit dem **CREATE TABLE**-Statement (siehe 1.4.10) zugeordnet. Dabei müssen die Attribute, die den Clusterkey bilden, mit denen der jeweiligen Relationen in Datentyp und Größe übereinstimmen. Es dürfen maximal 16 Tabellen auf diese Weise einem Cluster hinzugefügt werden. Bevor auf Relationen eines Clusters DML-Statements angewendet werden, muss ein Index auf dem Cluster definiert werden (siehe 1.4.8).

Beispiel

```
/* Definition des Clusters */
CREATE CLUSTER personnel (department_number NUMBER);
/* Einfügen der Relationen emp und dept in den Cluster */
CREATE TABLE emp
      (empno NUMBER,
       deptno NUMBER,
       ...)
      CLUSTER personnel (deptno);

CREATE TABLE dept
      (deptno NUMBER,
       dname CHAR(9),
       ...)
      CLUSTER Personnel (deptno);
/* Clusterindex für Ausführung von DML-Statements */
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

1.4.8 CREATE INDEX

Funktion: Erzeugt einen Index für eine existierende Relation oder einen existierenden Cluster.

Syntax:

CREATE [UNIQUE] INDEX *index* **ON**
{*table* (*column* [**ASC** | **DESC**] [, *column*[**ASC**|**DESC**]] ...)
| **CLUSTER** *cluster*}

Schlüsselwörter und Parameter:

UNIQUE stellt die Eindeutigkeit der Werte in den Indexspalten sicher.

index ist der Name des Index.

table ist der Name einer existierenden Relation, deren Attribute indiziert werden.

column ist der Name eines der Attribute, für die der Index kreiert wird.

cluster ist der Name eines existierenden Clusters, für den der Index erzeugt wird.

Beschreibung: Indexe können dann nützlich sein, wenn

- Relationen in der Reihenfolge des Indexattributs abgearbeitet werden,
- nach Tupeln mit spezifizierten Indexattributwerten gesucht wird.

Allerdings kann es zu Nachteilen beim Einfügen, Löschen und Ändern von Attributwerten kommen, da sowohl die Relations- als auch die Indexdaten verändert werden.
Es dürfen nicht mehr als 16 Attribute im Index vorkommen. Auf einer Relation können jedoch mehrere Indexe definiert werden.
Die Erstellung eines Clusterindex ist notwendig, um auf die Relationen des Clusters DML-Statements anwenden zu können. Dabei müssen keine Indexattribute angegeben werden, da der Index automatisch auf allen zum Clusterkey gehörenden Attributen angelegt wird.

Beispiel

CREATE UNIQUE INDEX idx_employee **ON** emp(empno);

1.4.9 CREATE SEQUENCE

Funktion: Generiert automatisch eindeutige Schlüssel für die Daten einer Relation.

Syntax:

CREATE SEQUENCE *sequence*
function

Schlüsselwörter und Parameter:

sequence ist der Name der Sequenz.

function ist eine der folgenden Funktionen:

- **INCREMENT BY** *xyz* Das Intervall zwischen den Sequenzwerten beträgt xyz.
- **START WITH** *xyz* definiert die Zahl, bei der die Sequenz beginnt.
- **MAXVALUE** *a* spezifiziert *a* als größten Wert, den die Sequenz annehmen kann.
- **MINVALUE** *a* spezifiziert *a* als kleinsten Wert, den die Sequenz annehmen kann.

Zugriff auf Sequenzwerte:

CURRVAL liefert den aktuellen Wert der Sequenz.

NEXTVAL inkrementiert die Sequenz und liefert den neuen Wert.

Beispiel

CREATE SEQUENCE eseq
INCREMENT BY 10;

SELECT eseq.**NEXTVAL FROM** DUAL;

Der erste Zugriff mit eseq.**NEXTVAL** liefert den Wert 1, der zweite 11, der dritte 21,.....

1.4.10 CREATE TABLE

Funktion: Erzeugt eine neue Relation.

Syntax:

CREATE TABLE *table*
({ *column datatype* [**DEFAULT** *expr*] [*column_constraint*] | *table_constraint* }
[,{ *column datatype* [**DEFAULT** expr] [*column_constraint*] | *table_constraint* }] . . .)
[**CLUSTER** *cluster* (*c_column* [*, c_column*] . . .)]
[**AS** *subquery*]

Schlüsselwörter und Parameter:

table ist der Name der Relation.

column ist der Name eines der Attribute der Relation.

datatype ist der Datentyp des Attributes (siehe 1.3.2).

column_constraint definiert eine Wertebereichseinschränkung für ein Attribut (siehe 1.4.6).

table_constraint definiert eine Wertebereichseinschränkung für eines oder mehrere Attribute (siehe 1.4.6).

cluster ist der Name eines existierenden Clusters, zu dem die Relation gehört. Der jeweilige Cluster muss immer vor den Relationen, aus denen er besteht, definiert werden.

c_column ist der Name eines Attributes, das Teil des Clusterkeys jenes Clusters ist, zu dem die Relation gehört. Sämtliche *c_columns* müssen auch unter den *columns* sein.

subquery ist ein **SELECT**-Statement, welches Tupel aus anderen Relation holt und in eine neue Relation importiert (siehe 1.4.21). Bei der Verwendung dieser Option genügt es, nur die entsprechenden Attributnamen anzugeben, die Datentypen werden dann aus der Query abgeleitet.

DEFAULT legt den Wert des Attributes fest, wenn dieser bei einem **INSERT** nicht angegeben wird.

Beschreibung: Eine Relation kann bis zu 1000 Attribute besitzen. Relationen werden ohne Tupel erstellt, außer sie werden mit Hilfe einer Query generiert. Tupel können mit Hilfe des **INSERT**-Statements (siehe 1.4.17) in eine Relation eingefügt werden.

Beipiel:
```
CREATE TABLE emp
    (empno NUMBER,
    ename  CHAR(10),
    job    CHAR(9),
    mgr    NUMBER,
    hiredate DATE,
    sal    NUMBER(10,2),
    comm   NUMBER(9,0),
    deptno NUMBER(2)),
    CLUSTER personnel(deptno);
```

oder

```
CREATE TABLE bigbucks
    AS SELECT empno
    FROM emp
    WHERE sal >
        (SELECT AVG(sal)
        FROM emp);
```

1.4.11 CREATE VIEW

Funktion: Definiert eine logische Relation, die auf anderen Relationen basiert.

Syntax: **CREATE [OR REPLACE] VIEW** *view* [(*alias* [, *alias*] . . .)] **AS** *subquery*

Schlüsselwörter und Parameter:

OR REPLACE dient dazu, die Definition einer existierenden View zu ändern, ohne sie vorher zu löschen und wieder zu erstellen.

view ist der Name der neuen virtuellen Relation.

alias sind die neuen Namen, die die Attribute aus den der View zugrunde liegenden Relationen bekommen. Wird kein *alias* angegeben, so heißt ein Attribut genauso wie in der zugrundeliegenden Relation.

subquery ist ein **SELECT**-Statement, das die Daten der Viewrelation aus den zugrundeliegenden Relationen erzeugt.

Beschreibung: Eine View ist eine logische Sichtweise einer oder meherer Relationen. In ihr werden keine Daten gespeichert, lediglich ihre Definition. Siehe auch 1.3.2.

Beispiel:
```
CREATE VIEW empdept(name, salary, dname)
    AS SELECT emp.ename, emp.sal, dept.dname
    FROM emp, dept
    WHERE emp.deptno = dept.deptno;
```

1.4.12 DELETE

Funktion: Löscht Tupel aus einer Relation.

Syntax: **DELETE [FROM]** *table* [**WHERE** *condition*]

Schlüsselwörter und Parameter:

table ist der Name der Relation oder View, aus welcher Tupel gelöscht werden sollen.

condition ist die Bedingung (siehe 1.4.5), die für die zu löschenden Tupel erfüllt sein muss.

Beschreibung: Dieses Statement löscht alle jene Tupel aus der Relation *table*, welche die **WHERE**-Klausel erfüllen. Werden Tupel aus einer View gelöscht, so werden sie auch aus der zugrunde liegenden Relation gelöscht. Wird die **WHERE**-Klausel weggelassen, so werden alle Tupel der Relation gelöscht.

Beispiel:

DELETE FROM emp
WHERE job = 'SALESMAN' **AND** comm < 100;

1.4.13 DROP CLUSTER

Funktion: Löscht einen Cluster aus der Datenbank.

Syntax: **DROP CLUSTER** *cluster* [**INCLUDING TABLES**];

Schlüsselwörter und Parameter:

cluster ist der Name des Clusters, der gelöscht werden soll.

INCLUDING TABLES bewirkt, dass auch alle Relationen, die zu dem Cluster gehören, gelöscht werden. Wird **INCLUDING TABLES** nicht angegeben, so müssen sämtliche Relationen, die zu diesem Cluster gehören, vorher mit dem **DROP TABLE**-Statement (siehe 1.4.15) gelöscht werden.

Beschreibung: **DROP CLUSTER** löscht auch den zugehörigen Clusterindex.

Beispiel: **DROP CLUSTER** personnel **INCLUDING TABLES**;

1.4.14 DROP INDEX

Funktion: Löscht einen Index aus der Datenbank.

Syntax: **DROP INDEX** *index*

Schlüsselwörter und Parameter:

index ist der Name des zu löschenden Index.

Beschreibung: Wird ein Index gelöscht, so hat das keinen Einfluss auf die zugehörige Relation. Löscht man einen Clusterindex, so kann man solange nicht auf die Daten des Clusters zugreifen, bis ein neuer Clusterindex definiert wurde.

Beispiel: **DROP INDEX** idx_employee;

1.4.15 DROP TABLE

Funktion: Löscht eine Relation und sämtliche darin enthaltenen Daten aus der Datenbank.

Syntax: **DROP TABLE** *table*

Schlüsselwörter und Parameter:

table ist der Name der zu löschenden Relation.

Beschreibung: Wird eine Relation gelöscht, so werden automatisch auch alle zugehörigen Indexe gelöscht. Views, die auf dieser Relation basieren, existieren weiterhin, werden allerdings ungültig.

Beispiel: **DROP TABLE** emp;

1.4.16 DROP VIEW

Funktion: Löscht eine View aus der Datenbank.

Syntax: **DROP VIEW** *view*

Schlüsselwörter und Parameter:

view ist der Name der zu löschenden View.

Beschreibung: Views, die auf dieser View basieren, existieren weiterhin, werden allerdings ungültig.

Beispiel: **DROP VIEW** empdept;

1.4.17 INSERT

Funktion: Fügt neue Tupel in die angegebene Relation oder die der angegebenen View zugrunde liegenden Relation ein.

Syntax: **INSERT INTO** *table* [(*column* [,*column*] . . .)]
{**VALUES** (*value* [, *value*] . . .) | *subquery*}

Schlüsselwörter und Parameter:

table ist der Name einer existierenden Relation oder View, wo Tupel eingefügt werden sollen.

column ist ein Attribut der Relation oder View, in die eingefügt werden soll.

value ist ein Wert, der in das seiner Position entsprechende Attribut eingefügt wird. Für *value* kann jeder beliebige Ausdruck (siehe 1.4.3) verwendet werden. Jeder *value*, der ungleich **NULL** ist, muss denselben Datentyp wie das zugehörige Attribut haben.

subquery ist ein **SELECT**-Statement (siehe 1.4.21), das soviele Werte liefern muss, wie Attribute aufgelistet sind.

Beschreibung: Bei der **VALUES**-Variante des **INSERT**-Statements wird ein einziges Tupel eingefügt. Die i-te Position der **VALUES**-Liste entspricht dabei dem i-ten Attribut.
Bei der *query*-Variante werden all jene Tupel eingefügt, die die Query liefert. Auch hier entspricht das i-te Attribut in der **SELECT**-Liste der Query dem i-ten Attribut in der Attributliste. Kommt ein Attribut in der Attribut-Liste nicht vor, so wird an der betreffenden Stelle in der Relation ein NULL-Wert eingetragen, daher darf ein solches Attribut nicht **NOT NULL** deklariert sein. Sollte keine Attributliste angegeben sein, so müssen für alle Attribute der Relation Werte bereitgestellt werden.

Beispiel:

INSERT INTO emp(empno, ename, job, sal, comm, deptno)
VALUES(7890, 'Jinks', 'Clerk', 1.2E3, NULL, 40);

INSERT INTO dept
VALUES(50, 'Production', 'San Francisco');

INSERT INTO bonus
SELECT ename, job, sal, comm
FROM emp
WHERE comm $<$ 25 $*$ sal;

1.4.18 RENAME

Funktion: Benennt eine existierende Relation oder View um.

Syntax: **RENAME** *old* **TO** *new*

Schlüsselwörter und Parameter:

old ist der aktuelle Name der Relation oder View.

new ist der neue Name der Relation oder View.

Beschreibung: Benennt eine Relation oder eine View von *old* in *new* um. *new* darf noch nicht existieren. Mit **RENAME** können keine Attribute umbenannt werden. Möchte man diese umbenennen, so kann man das, durch das Erstellen einer neuen Relation aus einer alten, mit dem **CREATE TABLE ... AS SELECT**-Statement (siehe 1.4.10) erreichen.

Beispiel: **RENAME** dept **TO** emp_dept;

1.4.19 ROLLBACK

Funktion: Beendet eine Transaktion. Die Änderungen, die seit dem letzten **COMMIT** (siehe 1.4.4) oder DDL-Statement durchgeführt wurden, gehen verloren. Die Datenbank ist wieder im selben Zustand wie am Beginn der abgebrochenen Transaktion.

Syntax: **ROLLBACK** [**WORK**] [**TO** [**SAVEPOINT**] *savepoint*]

Schlüsselwörter und Parameter:

WORK ist optional und dient der ANSI Kompatibilität.

SAVEPOINT ist optional und bewirkt ein Rückgängigmachen aller Befehle, die dem Synchronisationspunkt *savepoint* folgen. Die Transaktion wird dabei nicht beendet.

savepoint ist ein Synchronisationspunkt, der während der aktuellen Transaktion erzeugt worden ist.

Beschreibung: Ein **ROLLBACK** ohne die **SAVEPOINT**-Klausel beendet die Transaktion, macht alle Änderungen der laufenden Transaktion rückgängig, löscht alle Synchronisationspunkte dieser Transaktion und gibt ihre Locks frei.
Ein **ROLLBACK** mit der **SAVEPOINT**-Klausel macht nur einen Teil der Transaktion rückgängig (bis zum Synchronisationspunkt), löscht alle Synchronisationspunkte, die dem angegebenen folgen, und gibt alle Locks auf Relationen und Tupel frei, die seit dem Synchronisationspunkt vorgenommen worden sind.

Beispiel:

```
COMMIT; /* Die vorige Transaktion ist beendet, */
            /* mit dem nächsten Statement beginnt eine neue Transaktion. */
INSERT INTO emp(name, sal, bdate)
    VALUES('Jones, Bill', 10000, 1944);
SAVEPOINT s1;
INSERT INTO emp(name, sal, bdate)
    VALUES('Smith, Stan', 5000, 1950);
SAVEPOINT s2;
ROLLBACK WORK TO SAVEPOINT s1;
/* Datenbank wurde auf den Zustand von s1 zurückgesetzt */
INSERT INTO emp(name, sal, bdate)
    VALUES('Smith, Stan', 5000, 1948);
ROLLBACK WORK TO SAVEPOINT s1;
/* Datenbank wurde auf den Zustand von s1 zurückgesetzt */
COMMIT;
/* Es wurde nur das allererste INSERT durchgeführt. */
```

1.4.20 SAVEPOINT

Funktion: Deklariert einen Synchronisationspunkt innerhalb einer Transaktion.

Syntax: **SAVEPOINT** *savepoint*

Schlüsselwörter und Parameter:

savepoint ist der Name des Synchronisationspunktes.

Beschreibung: Der Benutzer kann innerhalb einer Transaktion mehrere (defaultmäßig maximal 5) Synchronisationspunkte deklarieren. Die Namen der Synchronisationspunkte einer Transaktion müssen eindeutig sein. Wird ein bereits bestehender Name verwendet, so wird der frühere Synchronisationspunkt dadurch gelöscht. Alle Synchronisationspunkte verlieren bei der Beendigung einer Transaktion ihre Gültigkeit.

Beispiel: Siehe 1.4.19.

1.4.21 SELECT

Funktion: Dient zur Abfrage einer oder mehrerer Relationen und kann als eigenes Statement oder auch als Query oder Subquery in anderen Statements verwendet werden.

Syntax:

SELECT [**ALL**|**DISTINCT**] {* | *table.** | *expr* [*c_alias*]}
[, {*table.**|*expr* [*c_alias*]}] ...
FROM {*table* | (*subquery*)} [*t_alias*] [, {*table* | (*subquery*)} [*t_alias*]] ...
[**WHERE** *condition*]
[**GROUP BY** *expr* [, *expr*] ... [**HAVING** *condition*]]
[{**UNION**|**INTERSECT**|**MINUS**|} **SELECT** ...]
[**ORDER BY**{*expr*|*position*} [**ASC**|**DESC**]
[, {*expr*|*position*} [**ASC**|**DESC**]]...]
[**FOR UPDATE OF** *column* [, *column*] ... [**NOWAIT**]]

Beschreibung: Durch ein **SELECT**-Statement wird ein sogenannter SPJ-Ausdruck spezifiziert. SPJ steht für Selection, Projection und Join. Bei der Abarbeitung eines **SELECT**-Statements wird zuerst das kartesische Produkt aller in der **FROM**-Klausel genannten Relationen gebildet. Daraus werden alle Tupel gestrichen, die die **WHERE**-Klausel nicht erfüllen. Die verbleibenden Tupel werden nach der **GROUP BY**-Klausel (siehe 1.4.21) in Gruppen zusammengestellt, von denen jene gestrichen werden, die die **HAVING**-Klausel nicht erfüllen. Dann werden die Ausdrücke in der **SELECT**-Klausel ausgewertet und – sollte **DISTINCT** verlangt worden sein – doppelte Tupel aus der Ergebnistabelle gestrichen. Zum Schluß werden die Tupel der Ergebnistabelle wie in der **ORDER BY**-Klausel angegeben geordnet.

Schlüsselwörter und Parameter:

ALL gibt an, dass alle Ergebnistupel angezeigt werden sollen. **ALL** ist der Defaultwert.

DISTINCT gibt an, dass in der Ergebnistabelle kein Tupel doppelt vorkommen darf. Gleiche Tupel werden aus der Ergebnistabelle gestrichen. Zwei Tupel sind genau dann gleich, wenn ihre Werte für jedes Attribut übereinstimmen.

***** Alle Attribute des Joins sollen angezeigt werden. Diese Variante darf nicht gemeinsam mit **UNION**, **INTERSECT** oder **MINUS** verwendet werden.

table.* Alle Attribute der Relation *table* sollen angezeigt werden.

expr ist ein arithmetischer Ausdruck, der aus Konstanten, Attributen, SQL-Funktionen und SQL-Operatoren gebildet wird (siehe 1.4.3).

c_alias Wird für ein Attribut ein Aliasname angegeben, so wird dieser für die Überschrift in der Ergebnistabelle verwendet. Der interne Name des Attributes wird aber nicht verändert. Das Attribut kann also nicht unter diesem Namen angesprochen werden.

table ist eine Relation oder eine View, aus der Informationen abgefragt werden.

subquery ist eine Unter-SELECT-Anweisung der Form **SELECT ... FROM ...** usw. Die Daten dieses Unter-SELECTS stehen dann wie eine View zur Verfügung.

t_alias ist ein Aliasname, mit dem die Relation innerhalb des aktuellen **SELECT**-Statements angesprochen werden kann. Wird ein Aliasname angegeben, so müssen sich alle Referenzen innerhalb der Query auf diesen Namen beziehen.

condition ist ein boolscher Ausdruck gebildet aus Konstanten, Attributnamen, arithmetischen und logischen Operatoren, Aggregatfunktionen und Vergleichsprädikaten (siehe 1.4.5).

UNION (siehe 1.3.3) erzeugt die Vereinigung mehrerer Abfragen.

INTERSECT (siehe 1.3.3) erzeugt den Durchschnitt mehrerer Abfragen.

MINUS (siehe 1.3.3) erzeugt die Differenzmenge zweier Abfragen.

position In der **ORDER BY**-Klausel kann man anstelle des Attributnamens auch die Position des jeweiligen Attributes innerhalb der **SELECT**-Liste angeben.

ASC|DESC gibt an, ob aufsteigend oder absteigend sortiert werden soll. Der Defaultwert ist **ASC**.

column ist ein Attribut, das zu einer Relation, die in der **FROM**-Klausel aufgelistet ist, gehört.

FOR UPDATE OF Ein **SELECT ... FOR UPDATE**-Statement geht gewöhnlich einem oder mehereren **UPDATE ... WHERE**-Statements (siehe 1.4.22) voraus. Wenn man die **FOR UPDATE**-Variante verwendet, belegt ORACLE die selektierten Tupel der Relationen mit einem Lock. Wenn ein Lock auf einem Tupel gesetzt worden ist, können andere Benutzer keinen Lock beziehungsweise kein Update durchführen. Die Locks sind gültig, solange die aktuelle Transaktion nicht beendet oder zu einem Synchronisationspunkt vor dem **SELECT**-Statement zurückgesetzt wird. **FOR UPDATE OF** signalisiert, dass ein Verändern der selektierten Tupel beabsichtigt ist, dies ist jedoch nicht notwendig. Unter Verändern ist die Anwendung eines **UPDATE**–, **INSERT**– oder **DELETE**-Statements gemeint. Diese Option kann nicht mit

- **DISTINCT** oder **GROUP BY**
- **UNION**–, **INTERSECT**– oder **MINUS**-Operatoren
- Gruppierungsfunktionen wie **COUNT** oder **MAX**

verwendet werden.

NOWAIT Falls es nicht möglich ist, die selektierten Tupel zu sperren, z.B. weil schon jemand anders einen Lock auf diese Tupel hat, so wird bei **NOWAIT** das **SELECT**-Statement abgebrochen. Andernfalls wird gewartet, bis alle zu sperrenden Tupel frei sind.

GROUP BY

Bei Verwendung der **GROUP BY**-Klausel werden die Tupel entsprechend der Werte der Attribute oder Ausdrücke, die in der Klausel angegeben sind, gruppiert. Jeder Ausdruck in der **SELECT**-Klausel muss dann entweder gruppiert werden oder dem Ausdruck muss eine Aggregatfunktion vorangestellt sein.
GROUP BY-Ausdrücke können jedes beliebige Attribut der Relationen der **FROM**-Klausel verwenden, egal ob sie in der **SELECT**-Liste aufscheinen oder nicht. Fehlt **GROUP BY**, so wird das gesamte Ergebnis als eine Gruppe aufgefasst.

Beispiel:
Die zwei Beispiele sollen verdeutlichen, was man bei Anwendung einer Gruppierung selektieren darf.

```
/* Richtig */
SELECT deptno, min(sal), max(sal)
    FROM emp
    GROUP BY deptno;
```

oder

```
/* Falsch, da mgr nicht in GROUP BY-Klausel */
SELECT mgr, deptno, avg(sal)
    FROM emp
    GROUP BY deptno;
```

Ergebnistabelle:

DEPTNO	MIN(sal)	MAX(sal)
10	1300	5000
20	800	3000
30	950	2850

Mit Hilfe der **HAVING**-Klausel können Tupel gestrichen werden, die die gewünschten Bedingungen nicht erfüllen.

Beispiel:

SELECT deptno, MIN(sal), MAX(sal) **FROM** emp
WHERE job = 'CLERK'
GROUP BY deptno
HAVING MIN(sal) < 1000;

Ergebnistabelle:

DEPTNO	MIN(sal)	MAX(sal)
20	800	1100
30	950	950

1.4.22 UPDATE

Funktion: Verändert die Daten in der angegebenen Relation.

Syntax:

UPDATE *table*
SET *column* = { *expr* | (*subquery*) }
[, *column* = { *expr* | (*subquery*) }] ...
[**WHERE** *condition*]

oder

UPDATE *table*
SET (*column* [, *column*] ...) = (*subquery*)
[, (*column* [, *column*] ...) = (*subquery*)] ...
[**WHERE** *condition*]

Schlüsselwörter und Parameter:

table ist der Name einer existierenden Relation, in der Tupel geändert werden sollen.

column ist der Name eines Attributs, das geändert werden soll.

expr ist ein Ausdruck (siehe 1.4.3), der den Wert liefert, den das Attribut bekommen soll.

subquery ist ein **SELECT**-Statement (siehe 1.4.21) und liefert die neuen Werte der Attribute. Dabei müssen die einzelnen Attributnamen in der Attributliste und in der **SELECT**-Liste der Query übereinstimmen.

condition Bei allen Tupeln der Relation, die die Bedingung (siehe 1.4.5) erfüllen, werden die Attributwerte erneuert.

Beschreibung Die **SET**-Klausel bestimmt, welche Attribute verändert und welche neuen Werte in ihnen gespeichert werden sollen. Die **WHERE**-Klausel legt fest, welche Tupel der Relation von der Änderung betroffen sind. Wird keine **WHERE**-Klausel angegeben, so beziehen sich die Änderungen auf die gesamte Relation.
Bei der Verwendung einer Subquery muss für jedes Tupel, das verändert wird, genau ein Tupel zurückgeliefert werden. Jeder Wert des Ergebnisses der Subquery wird den entsprechenden Attributen der Attributliste zugeordnet. Liefert die Subquery keine Tupel, so wird der Wert auf **NULL** gesetzt. Die Subquery kann auch auf die zu verändernde Relation angewendet werden.

Beispiel:

```
UPDATE emp
    SET job = 'Manager',
        sal = sal + 1000,
        deptno = 20,
    WHERE ename = 'Jones';
```

oder

```
UPDATE emp a
    SET deptno =
        (SELECT deptno
           FROM dept
           WHERE loc = 'Boston' ),
        (sal, comm) =
        (SELECT 1.1*AVG(sal), 1.5*AVG(comm)
           FROM emp b
           WHERE a.deptno = b.deptno)
    WHERE deptno IN
        (SELECT deptno
           FROM dept
           WHERE loc = 'Dallas' OR
                       loc = 'Detroit');
```

Kapitel 2

PL/SQL

2.1 Was ist PL/SQL

Mit Hilfe von PL/SQL können SQL Statements verwendet werden, um Daten in einer Datenbank zu manipulieren. Weiters können Prozeduren, Funktionen, Variablen und Konstanten definiert werden. Es ist auch möglich Laufzeitfehler abzufangen. PL/SQL kombiniert die Fähigkeit der Datenmanipulation mit SQL mit den Fähigkeiten einer prozeduralen Programmiersprache.
Eine gute Möglichkeit, um mit PL/SQL vertraut zu werden, ist anhand eines Beispielprogrammes. Das folgende Programm beschreibt die Bestellung von Tennisschlägern. Zuerst wird eine Variable des Typs NUMBER deklariert, die die Anzahl der lagernden Tennisschläger speichert. Anschließend wird abgefragt, wie viele Tennisschläger in der Datenbank (Tabelle *inventory*) vorhanden sind. Ist die Anzahl dieser größer 0 wird die Tabelle inventory aktualisiert (die Anzahl der vorhandenen Tennisschläger wird um 1 reduziert) und ein Eintrag in der Tabelle *purchase_record* getätigt. Ansonsten erfolgt ein Eintrag in die Tabelle *purchase_record*, die angibt, dass die Tennisschläger ausverkauft sind.

```
DECLARE
    qty_on_hand NUMBER(5);
BEGIN
    SELECT quantity
    INTO qty_on_hand
    FROM inventory
    WHERE product = 'TENNIS RACKET'
    FOR UPDATE OF quantity;

    IF qty_on_hand > 0 THEN /* check quantity */
        UPDATE inventory
        SET quantity = quantity - 1
        WHERE product = 'TENNIS RACKET';
        INSERT INTO purchase_record VALUES
```

```
        ('Tennis racket purchased', SYSDATE);
    ELSE
        INSERT INTO purchase_record VALUES
        ('Out of Tennis rackets', SYSDATE);
    END IF;

    COMMIT;
END;
```

2.2 Blockstruktur von PL/SQL

PL/SQL ist eine blockstrukturierte Sprache. Die grundlegenden Einheiten (Prozeduren, Funktionen) bilden logische Blöcke in PL/SQL, die wiederum eine beliebige Anzahl von Unterblöcken enthalten können. Im Normalfall ist ein Block für die Lösung eines konkreten Problems oder Unterproblems zuständig.
Ein PL/SQL Block besteht aus 3 Teilen, wobei der Deklarations- und der Ausnahmeteil nicht zwingend vorhanden sein müssen.

```
DECLARE
    – Deklarationen
BEGIN
    – Programm
EXCEPTION
    – Ausnahmebehandlung
END;
```

Variablen und Konstanten gelten nur in jenem Block, in dem sie deklariert wurden und verlieren ihre Gültigkeit sobald der Block beendet ist.
Unterblöcke können nur im ausführbaren Teil (Teil der zwischen **BEGIN** und **EXCEPTION**) sowie im Exception Teil deklariert werden. Weiters können Prozeduren und Funktionen (siehe 2.7) im Deklarationsteil eines Blockes definiert werden. Diese können dann aber nur in dem Block aufgerufen werden, in dem sie deklariert wurden.

2.2.1 Kommentare in PL/SQL

In PL/SQL werden - ähnlich SQL - Kommentare durch „/*“ eingeleitet und mit „*/“ beendet (siehe 1.4.1).

2.3 Deklaration und Benutzung von Variablen und Konstanten

PL/SQL erlaubt es, Variablen und Konstanten zu definieren, die sowohl in SQL-Statements als auch in prozeduralen Statements verwendet werden können. Vor der Verwendung von Variablen und Konstanten müssen diese allerdings im Deklarationsteil definiert werden.

2.3.1 Datentypen

Datentypen definieren den Wertebereich von Variablen und Konstanten sowie die darauf anwendbaren Operationen (z.B. Addition, Verkettung, ...). PL/SQL unterstützt sämtliche SQL-Datentypen (siehe 1.3.2), sowie einige zusätzliche Datentypen, von denen die wichtigsten hier erläutert werden:

PLS_INTEGER speichert ganze Zahlen mit Vorzeichen. Der Wertebereich liegt zwischen -2147483647 und 214748364. Variablen vom Typ **PLS_INTEGER** benötigen weniger Speicherplatz als **NUMBER** Variablen. Des weiteren benötigen arithmetische Operationen mit Variablen vom Typ **PLS_INTEGER** weniger Laufzeit als mit Variablen vom Typ **NUMBER** . Wird eine Variable dieses Typs in die Datenbank eingelesen so wird sie automatisch in einen SQL-Datentyp konvertiert.

BOOLEAN Der Datentyp **BOOLEAN** wird benutzt, um die logischen Werte TRUE, FALSE und NULL zu speichern. Es können keine Variablen vom Typ **BOOLEAN** in die Datenbank eingetragen werden und auch keine Variablen aus der Datenbank in Variablen dieses Typs eingelesen werden.

Neben den hier beschriebenen Datentypen gibt es zusätzlich auch noch benutzerdefinierte Datentypen (2.3.4) und Attribute für Datentypen (2.3.5).

2.3.2 Deklaration von Variablen und Konstanten

PL/SQL-Variablen und -Konstanten können jeden SQL-Datentyp (siehe 1.3.2), sowie jeden PL/SQL-Datentyp annehmen. Diese werden im Deklarationsteil definiert. Variablen sind standardmäßig mit dem Wert **NULL** initialisiert. Konstantendeklarationen werden mit dem Schüsselwort **CONSTANT** eingeleitet. Konstanten muss bei ihrer Deklaration ein Wert zugewiesen werden.
Will man eine Konstante deklarieren wird einfach das Schlüsselwort **CONSTANT** vor dem Datentyp eingefügt und dieser Konstante sofort ein Wert zugewiesen.

Beispiel:

```
DECLARE                                          /* Deklarieren von Variablen und Konstanten: */
    emp_name     CHAR;                           /* Variable des Typ CHAR */
    part_no      NUMBER(4);                      /* Variable des Typ NUMBER */
    in_stock     BOOLEAN;                        /* Variable des Typ BOOLEAN */
    credit_limit CONSTANT REAL := 5000.00;       /* Konstante */
```

```
BEGIN
    /* statements */
END;
```

2.3.3 Wertzuweisung zu PL/SQL-Variablen

Es gibt zwei Möglichkeiten, um einer Variable einen Wert zuzuzweisen:

- Verwendung des Zuweisungsoperators :=
- Zuweisung aus Werten der Datenbank

  Die Zuweisung aus Werten der Datenbank erfolgt über ein gewöhnliches SQL-Statement in das zwischen die **SELECT** und die **FROM**-Klausel der Zusatz „**INTO** *Variable1* [,*Variable2*] . . .“ gesetzt wird.

  Wenn die **SELECT**-Anweisung keine Zeile zurückgibt, tritt der Fehler $+100$ *ORA-01403: no data found* auf. Bei mehr als einer Zeile wird der Fehler -1422 *ORA-01422: exact fetch returns more than requested number of rows* ausgelöst.

Beispiel:

```
DECLARE
    emp_name    CHAR;
    part_no     NUMBER(4);
    in_stock    BOOLEAN;
    bonus       NUMBER(4);
BEGIN
    part_no := 3;
    emp_name := 'Bubu';
    part_no := part_no * 4;
    in_stock := FALSE;
    SELECT sal * 0.10
      INTO bonus
      FROM emp
      WHERE empname = emp_name;
END;
```

2.3.4 Benutzerdefinierte Unterdatentypen

Jeder PL/SQL-Datentyp spezifiziert eine Anzahl von Operationen und Werten. In PL/SQL ist es möglich, eigene Unterdatentypen zu definieren. Unterdatentypen besitzen die selben Operationen wie

ihr Basistyp. Sie dürfen Basistypen jedoch nicht einschränken (etwa in der Länge). Dazu folgendes Beispiel:

```
DECLARE
    SUBTYPE EmpDate IS DATE;                 /* basiert auf Datentyp DATE */
    SUBTYPE id_num  IS emp.empno%TYPE;       /* basiert auf Datentyp der Spalte emp.empno,
                                                %TYPE wird im Kapitel 2.3.5 beschrieben */
    SUBTYPE Delimiter IS CHAR(1);            /* ILLEGAL !!!! muss CHAR sein */

    /* Workaround, um eingeschränkte Unterdatentypen zu erzeugen: */
    temp VARCHAR2(15);
    SUBTYPE word    IS temp%TYPE;            /* Maximale Länge von word ist 15. */
BEGIN
END;
```

2.3.5 Attribute von Datentypen

Attribute sind Eigenschaften einer Variable, die es erlauben Datentypen und Strukturen zu referenzieren, ohne deren Definition zu wiederholen. Das % Zeichen kennzeichnet dabei den Attributindikator.

%TYPE liefert den Datentyp einer Variablen oder Datenbankspalte. Das ist nützlich, wenn Variablen deklariert werden, denen ein Wert aus der Datenbank zugewiesen wird. Im folgenden Beipiel wird die Variable *my_title* deklariert, die den gleichen Datentyp wie die Spalte *title* der Tabelle *books* erhalten soll.

Beispiel:

```
DECLARE
    my_title books.title%TYPE;
BEGIN
END;
```

%ROWTYPE speichert eine Zeile einer Tabelle. Die Elemente der mit **%ROWTYPE** deklarierten Variable entsprechen den Attributen der zugrunde liegenden Tabelle.

Beispiel:

```
DECLARE
    dept_rec dept%ROWTYPE;
BEGIN
    my_deptno := dept_rec.deptno;
END;
```

2.3.6 Konvertierung von Datentypen

Manchmal ist es nötig, einen Wert von einem Datentyp in einen anderen umzuwandeln. PL/SQL unterstützt sowohl explizite als auch implizite Konvertierung.

Explizite Konvertierung

Explizite Konvertierung wurde bereits im Kapitel 1.3.4 behandelt.

Implizite Konvertierung

PL/SQL kann Datentypen implizit konvertieren, wenn es nötig ist. Im folgenden Beispiel sind zwei **CHAR**-Variablen definiert, die zwei Zahlenwerte beinhalten. Die dritte Variable ist eine Variable vom Typ **NUMBER**, die die Differenz der in den obigen beiden **CHAR** Variablen gespeicherten Zahlen aufnehmen soll.

Beispiel:

```
DECLARE
    start_time      CHAR(5);
    finish_time     CHAR(5);
    elapsed_time    NUMBER;
BEGIN
    /* Systemzeit in Sekunden aus Datenbank holen */
    SELECT TO_CHAR(SYSDATE,'SSSS')
      INTO start_time
      FROM sys.dual;
    ...
    /* Systemzeit in Sekunden noch einmal aus Datenbank holen. */
    SELECT TO_CHAR(SYSDATE,'SSSS')
      INTO finish_time
      FROM sys.dual ;
    /* Differenz berechnen */
    elapsed_time := finish_time - start_time;
END;
```

Es liegt in der Verantwortung des Programmierers, dass die Werte einer Variablen konvertierbar sind, ansonsten kommt es zu einem Laufzeitfehler, deswegen wird empfohlen, explizite Konvertierung zu verwenden.

2.4 Kontrollstrukturen

In den folgenden Punkten wird die Syntax der PL/SQL Kontrollstrukturen vorgestellt:

2.4.1 IF-Statement

IF condition **THEN**
    sequence_of_statements1;
ELSIF condition **THEN**
    sequence_of_statements2;
ELSE
    sequence_of_statements3;
END IF;

Das **ELSIF** kann beliebig oft vorkommen, der **ELSE**-Zweig allerdings höchstens einmal.

2.4.2 LOOP Statement

Die einfachste Form eines **LOOP**-Statements ist die unendliche Schleife. Die **LOOP**-Schleife wird im im Allgemeinen mit einer **EXIT**-Anweisung verlassen (siehe 2.4.5).

LOOP
    sequence_of _statements;
END LOOP;

2.4.3 WHILE-Schleife

Die **WHILE**-Schleife wird ausgeführt, solange *condition* zutrifft. Ist *condition* schon vor dem Eintreten in die Schleife falsch, so wird die Schleife nie ausgeführt.

WHILE condition **LOOP**
    sequence_of_statements;
END LOOP;

2.4.4 FOR-Schleife

Die **FOR**-Schleife wird verwendet, wenn schon vor Ausführung der Schleife die Anzahl der Durchläufe bekannt ist. *counter* wird auf *lower_bound* gesetzt und nach jedem Durchlauf um 1 erhöht. Nachdem die Schleife mit $counter = higher\_bound$ durchgeführt wurde, wird sie beendet.

FOR counter **IN [REVERSE]** lower_bound..higher_bound **LOOP**
    sequence_of_statements;
END LOOP;

Die Variable *counter* wird automatisch deklariert und muss daher nicht im **DECLARATION**-Teil des PL/SQL-Programms angegeben werden. Nach Ende der Schleife ist die Variable nicht mehr verfügbar.
Das Schlüsselwort **REVERSE** führt dazu, dass von *higher_bound* zu *lower_bound* hinuntergezählt wird. Auch die **FOR**-Schleife kann – genauso wie die **WHILE**- und **LOOP**-Schleife – mit **EXIT** vorzeitig abgebrochen werden.

2.4.5 EXIT-Statement

Das **EXIT** Statement zwingt zum Verlassen einer Schleife und darf daher nur in Scheifen verwendet werden.

Beispiel:

```
LOOP
    ...
    IF credit_rating < 3 THEN
        ...
        EXIT /* Schleife sofort verlassen */
    END IF;
END LOOP;
```

2.4.6 EXIT-WHEN Statement

Das **EXIT-WHEN** Statement erlaubt es, eine Schleife bei einer bestimmten Bedingung zu verlassen.

Beispiel:

```
LOOP
    ...
    EXIT WHEN x < 3; /* verlasse Schleife wenn Bedingung erfüllt */
    ...
END LOOP;
```

2.4.7 LOOP Labels

In PL/SQL können Schleifen benannt werden. Sogenannte Labels müssen direkt vor einem **LOOP** Statement stehen und in doppelten spitzen Klammern eingeschlossen sein.

Beispiel:

```
<<outer>>/* Label*/
LOOP
    ...
        LOOP
            ...
            EXIT outer WHEN ... /* verlasse beide Schleifen */
        END LOOP;
    ...
END LOOP outer;
```

2.5 Operatoren

siehe 1.3.3

2.6 Cursor

SQL ist eine mengenorientierte Sprache und gibt als solche immer eine Menge von Tupeln als Resultat einer Abfrage zurück. Wenn mit einer prozeduralen Sprache darauf zugegriffen werden soll, muss ein Tupel nach dem anderen abgearbeitet werden. Dazu werden Cursor verwendet. Ein Cursor ist ein Zeiger, der auf ein Ergebnistupel nach dem anderen zeigen kann. Bei jedem SQL-Statement, das Daten manipuliert, wird implizit ein Cursor deklariert, der auf das Ergebnistupel zeigt. Für Abfragen, die mehr als ein Ergebnistupel beinhalten, muss explizit ein Cursor deklariert werden, um auf die einzelnen Tupel der Abfrage zugreifen zu können. Ein Cursor wird immer im Deklarationsteil deklariert.

Beispiel:

```
DECLARE
   CURSOR c1 IS
     SELECT  empno, ename, job
     FROM emp
     WHERE deptno = 20;
BEGIN
END;
```

Weiters können Parameter für einen Cursor definiert werden. Diesen Parametern können dann beim öffnen des Cursors Werte zugewiesen werden. Erfolgt keine Zuweisung so erhalten die Parameter ihre Standardwerte.

Beispiel:

```
DECLARE
   CURSOR c1 (low INTEGER DEFAULT 0, high INTEGER)
     IS SELECT  empno, ename, job
     FROM emp
     WHERE deptno >= low AND deptno <= high;
BEGIN
   OPEN c1(20, 35);
END;
```

2.6.1 Cursoroperationen

Folgende Operationen können mit einem Cursor ausgeführt werden:

OPEN Das **SELECT**-Statement aus der entsprechenden Cursordeklaration wird ausgeführt, und der Cursor wird vor dem ersten Tupel des Ergebnisses positioniert. Wird ein Cursor mit dem Befehl **FOR UPDATE** deklariert, dann sperrt das **OPEN** Statement die entsprechenden Zeilen (siehe auch 2.9).

Beispiel:

```
DECLARE
      emp_name VARCHAR2;
      CURSOR c1 IS
      SELECT  empno, ename, job
         FROM emp
         WHERE deptno = 20;
      CURSOR c2 (name VARCHAR2, salary NUMBER) IS
      SELECT  empno, ename, job
         FROM emp
         WHERE ename = name AND sal = salary;
BEGIN
      emp_name := 'Bubu';
      OPEN c1;
      OPEN c2 (emp_name, 3000);
END;
```

FETCH cursor_name INTO <**Variablen**> Das aktuelle Tupel wird in die PL/SQL-Variablen übertragen und der Cursor auf das nächste Tupel gerichtet. Falls der Cursor bereits auf das letzte Tupel gezeigt hat, kommt es zu **keiner** Fehlermeldung. Um derartige Fehler zu erkennen, muss mit den Attributen %FOUND und %NOTFOUND gearbeitet werden (siehe 2.6.4).

Beispiel:

```
DECLARE
    emp_name    VARCHAR2;
    emp_no      NUMBER;
    emp_job     VARCHAR2;
    CURSOR c1 IS
        SELECT  empno, ename, job
        FROM emp
WHERE deptno = 20;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO emp_no, emp_name, emp_job;
        EXIT WHEN c1%NOTFOUND;
        ...
    END LOOP;
END;
```

CLOSE cursor_name schließt den Cursor. Die Cursorposition ist danach undefiniert. Ein nachfolgendes **FETCH** auf diesen Cursor löst eine Exception aus (siehe 2.8).

2.6.2 Cursor FOR Loops

In den meisten Fällen, in denen ein Cursor benötigt wird, kann eine **Cursor FOR loop** anstatt der Befehle **OPEN, FETCH** und **CLOSE** verwendet werden. Eine **Cursor FOR loop** deklariert implizit den Schleifenindex als **%ROWTYPE**-Record (siehe 2.3.5), öffnet einen Cursor, speichert die Werte der Abfrage zeilenweise im Record und schliesst den Cursor nachdem das letzte Element ausgelesen wurde oder die Schleife aus einem anderen Grund verlassen wurde (z.B. Exception ausgelöst). Nach Verlassen der **Cursor FOR loop** wird die Schleifenvariable wieder gelöscht (siehe B.1).
Beispiel:

```
DECLARE
    CURSOR dept_aendern (dept_no INTEGER DEFAULT 0) IS
    SELECT *
        FROM emp
        WHERE deptno = dept_no
        FOR UPDATE OF deptno;
...
BEGIN
...
/* Die Mitarbeiter werden in das neue Department verlegt */
    FOR deptaendernrec IN dept_aendern(dept_no) LOOP
```

```
        UPDATE emp SET deptno = restr_deptno
        WHERE CURRENT OF dept_aendern;
    END LOOP;
...
END;
```

2.6.3 Cursorvariablen

Cursorvariablen haben die gleiche Funktionalität wie Cursor, sind aber nicht an eine spezielle Abfrage gebunden. Cursorvariablen sind echte PL/SQL-Variablen (Cursor sind eher mit Konstanten zu vergleichen) und können an Funktionen und Prozeduren als Parameter übergeben werden.

Beispiel:

```
DECLARE
    TYPE GenericCurTyp IS REF CURSOR; /* Allgemeiner Cursor */
    TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE; /* Spezieller Cursor */

    /* generic_cv ist eine Cursorvariable */
    PROCEDURE open_cv (generic_cv IN OUT GenericCurTyp, choice IN NUMBER) IS
    BEGIN
        /* Öffnen und Zuweisen von Cursorvariablen */
        IF choice = 1 THEN
            OPEN generic_cv FOR SELECT * FROM emp;
        ELSIF choice = 2 THEN
            OPEN generic_cv FOR SELECT * FROM dept;
        END IF;
    END;
BEGIN
...
END;
```

2.6.4 Cursorattribute

Jeder Cursor bzw. Cursorvariable besitzt vier Attribute. Diese Attribute enthalten Informationen zu dem momentanen Status des Cursors und können nur in prozeduralen Statements verwendet werden. Syntaktisch sieht die Verwendung von Cursorattributen folgendermaßen aus:

```
cursor_name%FOUND;
cursor_name%NOTFOUND;
```

%FOUND Nach dem Öffnen eines Cursors und vor dem ersten Ausführen des **FETCH**-Befehls hat das Attribut **%FOUND** den Wert **NULL**. Danach hat es den Wert **TRUE**, bis der **FETCH** Befehl keine Zeile mehr returniert, worauf es den Wert **FALSE** annimmt. Falls der Cursor noch nicht geöffnet ist wird eine Exception ausgelöst.

%ISOPEN ist **TRUE**, wenn der Cursor geöffnet ist, ansonsten **FALSE**

%NOTFOUND logisches Gegenteil von **%FOUND**

%ROWCOUNT enthält die Anzahl der Zeilen, die mit dem **FETCH**-Befehl ausgelesen wurden. Ist ein Cursor noch nicht geöffnet, wird eine Exception ausgelöst.

2.6.5 CURRENT OF Klausel

Die **CURRENT OF** Klausel wird in **UPDATE** und **DELETE** Statements verwendet um auf das Tupel zuzugreifen, auf welches der Cursor momentan zeigt.

Beispiel:

```
DECLARE
/* Deklarieren des Cursors */
      CURSOR c1
        IS SELECT *
        FROM emp, dept
        WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
        FOR UPDATE OF sal;
      ...
BEGIN
      OPEN c1;
      LOOP
        FETCH c1 INTO ...
        ...
        /* Zugriff auf das Tupel, auf das der Cursor momentan zeigt.
        UPDATE emp SET sal = new_sal WHERE CURRENT OF c1;
      END LOOP;
END;
```

2.7 Unterprogramme

Unterprogramme sind benannte PL/SQL Blöcke, die aufgerufen werden können und denen Parameter übergeben werden können. In PL/SQL gibt es zwei Typen von Unterprogrammen: Prozeduren und Funktionen. Die Deklaration erfolgt im Deklarationsteil des Blocks/Unterprogrammes, in dem die

Prozedur/Funktion aufgerufen wird, und zwar immer nach Deklaration von Variablen, Konstanten und Cursorn.

2.7.1 Prozeduren

Eine Prozedur ist ein Unterprogramm, das bestimmte Aktionen ausführt. Einer Prozedur können Parameter übergeben werden, allerdings dürfen diese keine Größenbeschränkungen haben (z.B. x **NUMBER**(3) erzeugt einen Fehler).

PROCEDURE name [(parameter [, parameter] . . .)] **IS**
    Lokale Deklarationen
BEGIN
    Ausführbare Statements
EXCEPTION
    Ausnahmeteil
END

2.7.2 Funktionen

Eine Funktion ist ein Unterprogramm, welches einen Wert berechnet und diesen an das aufrufende Programm zurückliefert. Einer Funktion können Parameter übergeben werden, allerdings dürfen auch hier die Datentypen der Parameter keinen Größenbeschränkungen unterliegen.

FUNCTION name [(parameter [, parameter] . . .)] **RETURN** datatype **IS**
    Lokale Deklarationen
BEGIN
    Ausführbare Statements
    RETURN returnvariable; /* beendet die Funktion, liefert einen Wert zurück */
EXCEPTION
    Ausnahmeteil
END

2.7.3 Arten von Parametern

IN Mit Hilfe von **IN**-Parametern werden Werte an ein Unterprogramm übergeben. Innerhalb des Unterprogrammes verhält sich dieser Parameter wie eine Konstante. **IN**-Paramter können mit Defaultwerten initialisiert werden (z.B. „acct_id **IN INTEGER DEFAULT** 4711“).

Das folgende Beispiel erzeugt einen Syntaxfehler, da versucht wird, einem **IN**-Parameter einen Wert zuzuweisen:

```
PROCEDURE debit_account (acct_id IN INTEGER, amount IN REAL) IS
    minimum CONSTANT REAL := 10.0;
    service CONSTANT REAL := 0.50;
BEGIN
    ...
    IF amount < minimum THEN
        amount := amount + service; /* Syntax ! */
    END IF
END
```

OUT **OUT**-Parameter liefern einen Wert an das aufrufende Programm zurück. Innerhalb eines Unterprogrammes verhalten sie sich wie nicht initialisierte Variablen, deswegen können ihre Werte keiner anderen Variable und auch nicht sich selbst zugewiesen werden.

Das folgende Beispiel erzeugt einen Syntaxfehler, da versucht wird, auf den Inhalt eines **OUT**-Parameters zuzugreifen:

```
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
    hire_date DATE;
BEGIN
    SELECT sal * 0.10, hiredate
      INTO bonus, hire_date
      FROM emp
      WHERE empno = emp_id ;
    IF MONTHS_BETWEEN (SYSDATE, hire_date ) > 60 THEN
      bonus := bonus + 500; /* causes syntax error ! */
    END IF
END
```

IN OUT **IN OUT**-Parameter übergeben dem Unterprogramm Werte und liefern neue Werte an das aufrufende Programm zurück. Innerhalb des Unterprogrammes verhalten sich **IN OUT**-Parameter wie initialisierte Variablen, d.h. **IN OUT**-Parameter können wie normale Variablen behandelt werden. Daher können sie anderen Variablen zugewiesen werden als auch Werte erhalten.

2.8 Fehlerbehandlung

Falls in einem PL/SQL-Block ein Fehler auftritt, wird eine Exception ausgelöst. Die normale Ausführung des Blockes wird gestoppt und der **Exception**-Teil des Blockes wird ausgeführt (siehe 2.2).
Vordefinierte Exceptions werden automatisch vom Laufzeitsystem ausgelöst. (z.B. **ZERO_DIVIDE** bei einer Division durch Null). Benutzerdefinierte Exceptions werden im Deklarationsteil eines

Blockes definiert und müssen mit dem Schlüsselwort **RAISE** ausgelöst werden. Im ausführbaren Teil wird uberprüft, ob die Bedingungen zur Auslösung der Exception erfüllt sind. Ist dies der Fall, wird die Exception ausgelöst und der Code im Exceptionteil abgearbeitet.

2.8.1 Vordefinierte Exceptions

Eine interne Exception wird immer dann ausgelöst, wenn im PL/SQL-Programm ein ORACLE-Fehler auftritt, oder Systemgrenzen überschritten werden. Jeder ORACLE-Fehler hat eine Nummer. Jedoch müssen Exceptions über ihren Namen angesprochen werden. Deswegen definiert PL/SQL einige ORACLE-Fehler als Exceptions.
Will man andere ORACLE-Fehler abfangen, so muss man den **OTHER** Handler verwenden, der alle Exceptions abfängt, die im Code nicht namentlich erwähnt werden.
Eine Liste der vordefinierten Exceptions befindet sich auf `http://minteka.dbai.tuwien.ac.at:8000/doc/appdev.817/a77069/06_errs.htm#784`.

```
DECLARE
    ...
BEGIN
    SELECT price / earnings
        INTO pe_ratio
        FROM stocks
        WHERE symbol = 'XYZ'; /* könnte einen Divison durch Null Fehler erzeugen */
    INSERT INTO stats (symbol, ratio)
        VALUES ('XYZ', pe_ratio);
    COMMIT;
EXCEPTION
    WHEN ZERO_DIVIDE THEN /* handles 'division-by-zero' error */
        INSERT INTO stats (symbol, ratio)
            VALUES ('XYZ', NULL);
        COMMIT;
    ...
    WHEN OTHERS THEN /* handles all other errors */
        ROLLBACK;
END;
```

2.8.2 Benutzerdefinierte Exceptions

In PL/SQL ist es möglich, eigene Exceptions zu definieren. Diese müssen im Deklarationsteil eines Blockes/Unterprogrammes mit dem Datentyp **EXCEPTION** deklariert und mit dem Befehl **RAISE** explizit ausgelöst werden. Selbstdefinierte Exceptions sind nur in dem Block (sowie in allen Unterblöcken des Blockes) gültig, in dem sie deklariert wurden.

```
DECLARE
    past_due EXCEPTION;
BEGIN
    ...
    IF ... THEN
        RAISE past_due;
    END IF;
    ...
END;
EXCEPTION
    WHEN past_due THEN
        ...
END;
```

2.8.3 EXCEPTION_INIT

Mit Hilfe der Klausel **EXCEPTION_INIT** kann man jeder selbstdefinierten Exception und jedem ORACLE-Fehler einen Namen und eine ORACLE-Fehlernummer zuweisen. Dadurch ist es möglich, interne Fehler mit Namen anzusprechen und einzeln abzufragen.

```
DECLARE
    deadlock_detected EXCEPTION;
    PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
    ...
EXCEPTION
    WHEN deadlock_detected THEN
        /* Fehlerbehandlung */
    ...
END;
```

2.8.4 SQLCODE - SQLERRM

Mit Hilfe der Funktionen **SQLCODE** und **SQLERRM** kann ausgeben werden, welcher Fehler aufgetreten ist, wobei **SQLCODE** die ORACLE Fehlernummer beinhaltet und **SQLERRM** die dazugehörige Fehlermeldung. Bei benutzerdefinierten Exceptions ist die Fehlernummer immer +1, die Fehlermeldung lautet 'User-Defined Exception', ausser es wurde die Klausel **EXCEPTION_INIT** (siehe 2.8.3) für diese Exception verwendet.
Wurde keine Exception ausgelöst, returniert **SQLCODE** den Wert 0 und **SQLERRM** folgende Meldung:

```
ORA-0000: normal, succesful completion
```

```
DECLARE
    err_num NUMBER;
    err_msg VARCHAR2(100);
BEGIN
    ...
EXCEPTION
    ...
    WHEN OTHERS THEN
        err_num := SQLCODE;
        err_msg := SUBSTR(SQLERRM, 1, 100);
        INSERT INTO errors VALUES (err_num, err_msg)
/* In diese selbst angelegte Tabelle werden Fehlernummer und Text eingetragen. */
    ...
END;
```

Weiters besteht die Möglichkeit der Funktion **SQLERRM** eine Fehlernummer als Parameter zu übergeben worauf diese mit der passenden Fehlermeldung antwortet.

```
DECLARE
    err_msg VARCHAR2(100);
BEGIN
    /* Hole alle Fehlermeldungen */
    FOR err_num IN 1..9999 LOOP
        err_msg := SQLERRM(-err_num); /* negative ORACLE Fehlernummer */
        INSERT INTO my_errors VALUES (err_num, err_msg);
    END LOOP;
END;
```

2.9 Transaktionen

Im Prinzip werden Transaktionen in PL/SQL analog zu interaktivem SQL (siehe 1.3.6) behandelt.

Um in PL/SQL Programmen transaktionale Sicherheit zu gewährleisten, stellt ORACLE folgende Mechanismen zur Verfügung:

- Read-consistent Views stellen sicher, dass nach Beginn einer Query das Endergebnis dieser Query nicht durch **UPDATE, DELETE** oder **INSERT** Statements anderer Transaktionen beeinflusst wird. *Wird eine Query erneut ausgeführt, so kann sich das Ergebnis von der vorhergehenden Ausführung unterscheiden (auch innerhalb einer Transaktion).*

- Über Locks wird sichergestellt, dass Tupel durch andere Transaktionen zwischenzeitlich nicht verändert werden können. Locks werden beispielsweise durch die **FOR UPDATE** Klausel (siehe 2.6.1) implizit gesetzt.

Sollten Werte mit einem **SELECT**- oder **FETCH**-Statement aus der Datenbank gelesen werden und in einem späteren **UPDATE**- oder **DELETE**-Statement verwendet werden, ist nicht sichergestellt, dass diese Werte nicht zwischenzeitlich durch andere Transaktionen (nach deren **COMMIT**) verändert wurden.
Beachten Sie auch, dass ein Fehler während der Ausführung eines PL/SQL-Statements die Transaktion nicht automatisch abbricht. Der Fehler wird registriert, der Befehl rückgängig gemacht, und dann im Programmcode fortgefahren. Es ist also die Aufgabe des Programmierers, die Transaktion abzubrechen, wenn ein Fehler aufgetreten ist.
Weiters ist zu beachten, dass bei einem **ROLLBACK** zwar die Datenbank, nicht aber der Programmcounter und die PL/SQL-Variablen auf den Zustand zu Beginn der Transaktion bzw. des jeweiligen Synchronisationspunktes zurückgesetzt werden. Der Programmierer ist dafür verantwortlich, dass das Programm nach einem **ROLLBACK** an der richtigen Stelle weiterarbeitet (siehe auch Kapitel B.1 Musterbeispiel).

2.9.1 RETURNING Klausel

Oft ist es nützlich, Informationen über die Zeile zu erhalten, die bei einer SQL Operation verändert wurde. **INSERT-**, **UPDATE-** und **DELETE**-Statements können eine **RETURNING** Klausel beinhalten, die die Spaltenwerte der veränderten Zeile in PL/SQL Variablen schreibt, wobei bei **INSERT** und **UPDATE** die Spaltenwerte *nach* der ausgeführen Operation ausgegeben werden, bei **DELETE** *vor* der ausgeführten Operation.

```
PROCEDURE update_salary (emp_id NUMBER) IS
    name VARCHAR2(15);
    new_sal NUMBER;
BEGIN
    UPDATE emp SET sal = sal * 1.1
        WHERE empno = emp_id
        RETURNING ename, sal INTO name, new_sal;
END;
```

Wenn keine Zeile von der Änderung betroffen ist, bleibt der Inhalt der **RETURNING**-Variablen unverändert. Sind mehrere Zeilen betroffen, so tritt Fehler -1422 *ORA-01422: exact fetch returns more than requested number of rows* auf.

2.10 PL/SQL-Packages

In PL/SQL ist es möglich, logisch zusammengehörende Variablen, Cursor und Unterprogramme in einem Package zusammenzufassen.

Packages bestehen aus zwei Teilen, einer Spezifikation und einem Body. Die Spezifikation ist die Schnittstelle zu anderen Applikationen; dort werden Variablen, Cursor, Exceptions und Unterprogramme deklariert, die dann im Package Body implementiert werden. Packages können kompiliert und in der Datenbank gespeichert werden, wodurch andere Applikationen darauf zugreifen können.

```
/* Spezifikation */
CREATE PACKAGE emp_actions AS
        PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);
        PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

/* Package Body */
CREATE PACKAGE BODY emp_actions AS
        PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
        BEGIN
            INSERT INTO emp VALUES (empno, ename, ...);
        END hire_employee;
        PROCEDURE fire_employee (emp_id NUMBER) IS
        BEGIN
            DELETE FROM emp WHERE empno = emp_id;
        END fire_employee;
END emp_actions;
```

In ORACLE gibt es bereits vordefinierte Packages, von denen für die Übung nur das folgende von Bedeutung ist:

2.10.1 DBMS_OUTPUT

Dieses Package ermöglicht es, Output von PL/SQL Blocks und Unterprogrammen anzuzeigen. Die Prozedure *put_line* speichert Outputinformationen in einem Puffer. Wird die SQLPLUS-Umgebungsvariable **SERVEROUTPUT ON** (Befehl: **SET SERVEROUTPUT ON** im SQLPLUS-Prompt) gesetzt, dann wird der Output am Bildschirm angezeigt.

```
PROCEDURE calc_payroll (payroll IN OUT) IS
      CURSOR c1 IS
          SELECT sal, comm
          FROM emp;
BEGIN
      payroll := 0;
      FOR c1rec IN c1 LOOP
          payroll := payroll + c1rec.sal + c1rec.comm;
      END LOOP;
```

dbms_output.put_line('payroll: ' || **TO_CHAR**(payroll));
END calc_payroll;

2.11 PL/SQL und SQL*Plus

Folgendes muss beim Arbeiten mit SQL*Plus beachtet werden:

- Durch die Eingabe eines PL/SQL-Blocks (beginnend mit **DECLARE** oder **CREATE PROCEDURE** bzw. **CREATE FUNCTION**) wechselt SQL*Plus in den PL/SQL-Modus. Es genügt dann nicht mehr, mit „**END;**“ abzuschließen. Um die Eingabe eines PL/SQL-Blocks zu beenden, muss nach dem „**END;**“ eine Zeile mit nur einem Schrägstrich „ **/** “ folgen. Dies gilt auch für PL/SQL-Blöcke, die über ein Commandfile (siehe 1.2.3) ausgeführt werden.

2.12 Stored Procedures

PL/SQL-Programme können als „Stored Procedures“ erstellt werden. In diesem Fall werden sie nicht direkt ausgeführt sondern in der Datenbank gespeichert.

2.12.1 Erstellen von Stored Procedures

Eine Stored Procedure wird mit den Befehlen **CREATE PROCEDURE** und **CREATE FUNCTION** erstellt.

CREATE PROCEDURE name [(parameter [, parameter]...)] **IS**
    Lokale Deklarationen
BEGIN
    Ausführbare Statements
EXCEPTION
    Ausnahmeteil
END;

CREATE FUNCTION name [(parameter [, parameter]...)] **RETURN** datatype **IS**
    Lokale Deklarationen
BEGIN
    Ausführbare Statements
    RETURN returnvariable; /* beendet die Funktion, liefert einen Wert zurück */
EXCEPTION
    Ausnahmeteil
END;

Falls beim Anlegen ein Fehler auftritt, so kann mit **SHOW ERRORS** eine Fehlerbeschreibung abgerufen werden.

2.12.2 Entfernen von Stored Procedures

Stored Procedures werden mit den Befehlen **DROP PROCEDURE** und **DROP FUNCTION** entfernt.

DROP [**PROCEDURE** | **FUNCTION**] name

2.12.3 Aufrufen von Stored Procedures

Mit „**EXECUTE** name[(parameter [, parameter] ...)]“ wird eine Procedure unter SQL*Plus aufgerufen. Der Rückgabewert einer Function kann mit „**SELECT** name[(parameter [, parameter] ...)] **FROM** DUAL“ ermittelt werden.

2.13 Nähere Informationen

In diesem Kapitel wurden nur Grundzüge und Teile des Leitungsumfangs von PL/SQL vorgestellt. Nähere Informationen zu PL/SQL können in der ORACLE Online Doku unter http://minteka.dbai.tuwien.ac.at:8000/doc/appdev.817/a77069/toc.htm nachgelesen werden.

Anhang A

Beispielrelationen

A.1 Tabellenstruktur

```
CREATE TABLE dept
(
        deptno      NUMBER(2) PRIMARY KEY,
        dname       VARCHAR2(14),
        loc         VARCHAR2(13)
);

CREATE TABLE emp
(
        empno       NUMBER(4) PRIMARY KEY,
        ename       VARCHAR2(10),
        job         VARCHAR2(9),
        mgr         NUMBER(4) REFERENCES emp(empno),
        hiredate    DATE,
        sal         NUMBER(7,2),
        comm        NUMBER(7,2),
        deptno      NUMBER(2) REFERENCES dept(deptno) NOT NULL
);

CREATE SEQUENCE     seq_deptno
        START WITH  10
        INCREMENT BY 10;
```

Die Syntax von **CREATE SEQUENCE** wird in 1.4.9 erläutert.

A.2 Daten

Relation: dept

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

Der Schlüssel DEPTNO wird über die **SEQUENCE** *seq_deptno* generiert!

Relation: emp

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30
7782	CLARK	MANAGER	7839	09-JUN-81	2450		10
7788	SCOTT	ANALYST	7566	19-APR-87	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500		30
7876	ADAMS	CLERK	7788	23-MAY-87	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		30
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		10

A.3 Physische Datenorganisation

A.3.1 Beispiel Taxiunternehmen

Für ein Taxi-Unternehmen ist eine Datenbank zu entwerfen und zu implementieren. Daten werden über Taxifahrer, Taxis und Taxistandplätze erhoben.
Folgende Informationen über die Abhängigkeit von Daten wurden ermittelt:

- Im Unternehmen gibt es Taxis bestimmter Autotypen (*autotyp*). Jeder Autotyp hat einen bestimmten Kilometerstand (*kmstand*), ab dem er am besten aus dem Verkehr gezogen und verkauft werden sollte. Jedes einzelne Taxi hat eine Identifikationsnummer (*taxinr*). Zusätzlich sind zu jedem Taxi seine Anschaffungskosten (*ankosten*) und seine Betriebskosten pro Kilometer (*kmkosten*) bekannt.

- Jeder Angestellte im Taxiunternehmen hat einen Namen (*name*), eine Wohnadresse (*adresse*) und eine eindeutige Sozialversicherungsnummer (*svnr*). Innerhalb des Unternehmens wird ein Angestellter durch seine Sozialversicherungsnummer identifiziert. Weiters ist für jeden Angestellten seine Wochenarbeitszeit in Stunden (*wochenzeit*) und seine Lohngruppe (*lohngr*) bekannt. Jeder Lohngruppe entspricht ein bestimmter Stundenlohn (*lohn*).

- Die im Unternehmen beschäftigten Taxifahrer arbeiten an bestimmten Wochentagen (*wochentag*) zu bestimmten Zeiten (*von*, *bis*) im Schichtdienst. Während einer Schicht sind sie einem bestimmten Taxistandplatz zugeteilt.

- Taxifahrer haben Fahrerfahrung auf keinem, einem Autotyp oder mehreren Autotypen.

- Taxifahrer fahren jeweils mit einem Taxi. Jede Woche wird für jeden Taxifahrer die Kilometerleistung (*kml*) erhoben, das heißt die Anzahl der Kilometer, die in einer Woche gefahren wurden. Es wird jeweils nur die letzte Kilometerleistung pro Taxifahrer gespeichert.

- Jeder Bezirk (*bezirk*) der Stadt hat eine bestimmte Anzahl (*standanz*) von Taxistandplätzen. Jeder Taxistandplatz hat einen eindeutigen Namen (*standname*), kann eine gewisse Anzahl (*kapazitaet*) von wartenden Taxis aufnehmen und hat eine bestimmte Telefonnummer (*standtelnr*), unter der der Standplatz erreichbar ist.

Weiters sind bereits einige Abfragen bekannt, die regelmäßig durchgeführt werden sollen.

- Einmal im Jahr wird eine Liste benötigt, die für jeden Bezirk den Standplatz mit der größten Kapazität enthält. Die Liste soll den Bezirk, den Namen des Taxistandes und dessen Kapazität enthalten und nach Bezirken sortiert sein.

- Wöchentlich werden Gehaltslisten erstellt. Der Wochenlohn eines Angestellen errechnet sich nach folgender Formel:

Wochenlohn = (Wochenarbeitszeit *
Stundenlohn der entsprechenden Lohngruppe) +
Prämie

Prämie = Kilometerleistung * 0.9

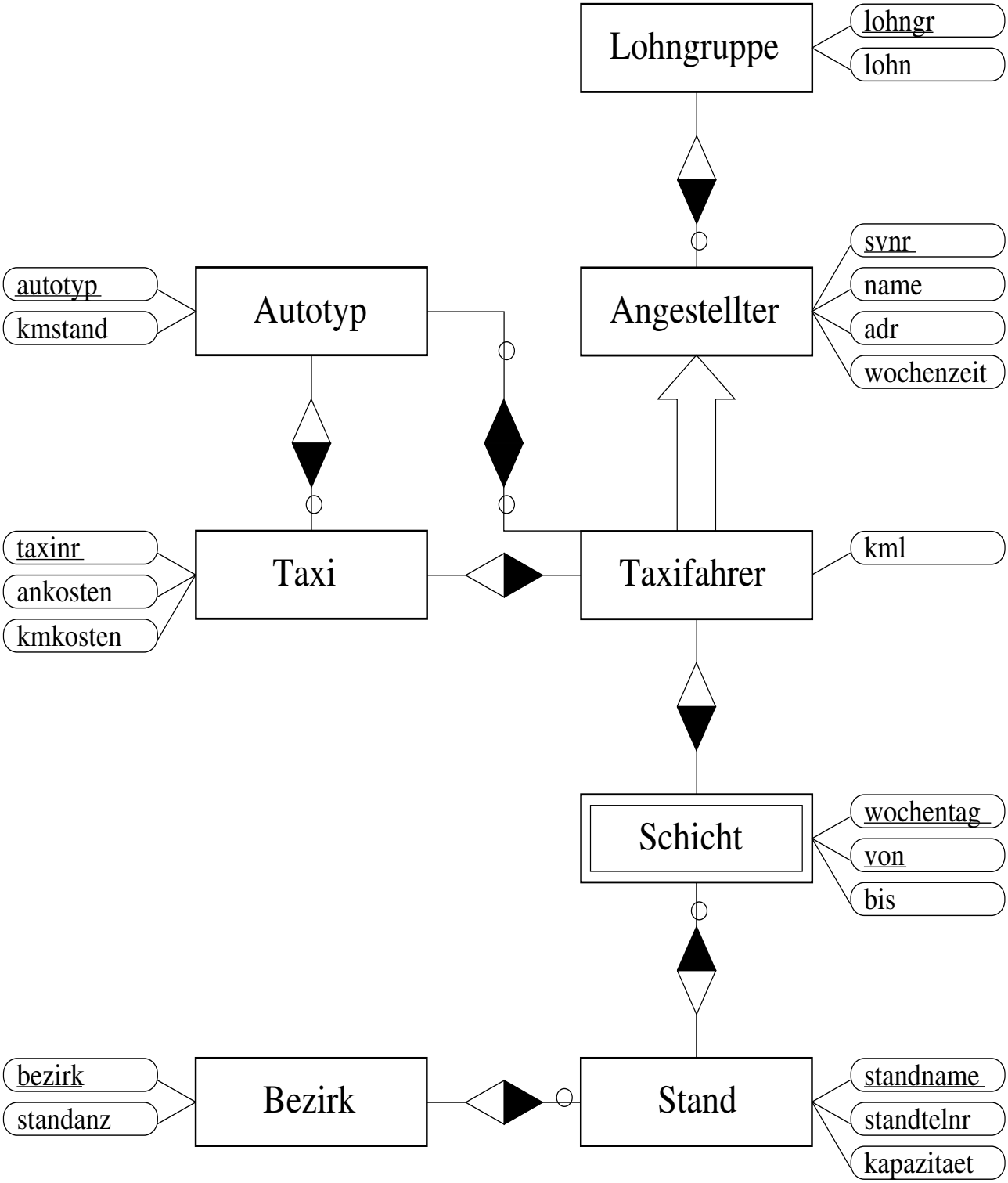
Die Liste soll Sozialversicherungsnummer, Namen und Wochenlohn jedes Angestellten enthalten und nach dem Namen sortiert sein.

- Pro Wochentag wird die Anzahl jener Taxifahrer gesucht, die mit einem Auto fahren, für dessen Typ sie keine Ausbildung haben. Diese Aufstellung wird viermal im Jahr benötigt.
- Einmal im Monat wird eine Aufstellung benötigt, die Information darüber enthält, wieviele Stunden die einzelnen Autotypen pro Woche gefahren werden. Diese Liste soll nach Autotypen sortiert werden.
- Die Leistung eines Taxifahrer wird in diesem Unternehmen nach dem Quotient *kilometerleistung/wochenzeit* beurteilt. Wöchentlich werden Fahrer gesucht, bei denen dieser Quotient 10% über dem Durchschnitt liegt. Diese Liste soll die Sozialversicherungsnummer, den Namen, die Wochenarbeitszeit und die Kilometerleistung solcher Fahrer enthalten und nach den Namen sortiert sein.
- Es werden Fahrer gesucht, bei denen der Quotient *kilometerleistung/wochenzeit* mehr als 10% unter dem Durchschnitt liegt. Diese Liste soll die Sozialversicherungsnummer, den Namen, die Wochenarbeitszeit und die Kilometerleistung solcher Fahrer enthalten und nach den Namen sortiert sein.
- Oft möchte man herausfinden, welcher Fahrer mit welchem Auto zu einer bestimmten Zeit unterwegs war. Zum Beispiel möchte man wissen, welche Fahrer (*name*) mit welchem Auto (*taxinr* und *autotyp*) Montag um 13.00 Dienst hatten.

A.3.2 Das EER-Diagramm

A.3.3 Das Relationenmodell

lohngruppe	(lohngr, lohn)
angestellter	(svnr, name, adresse, wochenzeit, lohngr)
taxifahrer	(svnr, taxinr, kml)
autotyp	(autotyp, kmstand)
taxi	(taxinr, autotyp, ankosten, kmkosten)
erfahrung	(svnr, autotyp)
schicht	(svnr, wochentag, von, bis, standname)
bezirk	(bezirk, standanz)
stand	(standname, standtelnr, kapazitaet, bezirk)


Lohngruppe
lohngr
lohn
Angestellter
svnr
name
adr
wochenzeit
Autotyp
autotyp
kmstand
Taxi
taxinr
ankosten
kmkosten
Taxifahrer
kml
Schicht
wochentag
von
bis
Bezirk
bezirk
standanz
Stand
standname
standtelnr
kapazitaet

A.3.4 Implementierung

Im folgenden wird exemplarisch aufgezeigt, wie man bei der Implementierung des Modells in ORACLE vorgehen soll. Es wird vorausgesetzt, dass Sie die entsprechenden Teile im Online Reference Manual bereits gelesen haben.
Für das Beispiel ist die Bildung des folgenden Clusters mit Hilfe des **CREATE CLUSTER**-Statements (siehe 1.4.7) aufgrund der Häufigkeit der benötigten Joins in den Queries sinnvoll:

```
CREATE CLUSTER ang_tf_schicht (svnr INTEGER);
...
```

Damit wird den oft vorkommenden Joins zwischen den Relationen *angestellter*, *taxifahrer* und *schicht* Rechnung getragen, die alle über das Attribut *svnr* erfolgen.
Natürlich würden sich noch weitere Clusterbildungen anbieten, so z.B. ein Cluster mit dem Attribut *taxinr* zwischen den Relationen *Taxi* und *taxifahrer* oder ein Cluster mit dem Attribut *lohngr* zwischen den Relationen *lohngruppe* und *angestellter*. Diese führen aber zu Konflikten mit dem oben definierten Cluster, da eine Relation natürlich immer nur zu einem einzigen Cluster gehören kann. Deshalb können die genannnten Cluster nicht gemeinsam definiert werden. Solche Zielkonflikte sind häufig bei *Real-World*-Beispielen anzutreffen und sind erst nach Einholung weiterer Information aufzulösen.
Nach der Definition der Cluster kann man nun die Relationen des Modells mit dem **CREATE TABLE**-Statement (siehe 1.4.10) implementieren. Dabei sind vor allem auch die Constraints zu beachten. Sie geben Auskunft über Wertebereichseinschränkungen, Primärschlüssel, Fremdschlüssel und NULL-Werte (siehe 1.4.6). So sieht die Definition der Relation *taxifahrer* beispielsweise wie folgt aus:

```
CREATE TABLE taxifahrer (
svnr            INTEGER NOT NULL REFERENCES angestellter (svnr),
taxinr          INTEGER NOT NULL,
kml             INTEGER,
PRIMARY KEY (svnr),
FOREIGN KEY (taxinr) REFERENCES taxi (taxinr))
CLUSTER ang_tf_schicht (svnr);
...
```

Nun müssen noch Indizes definiert werden. Die Definition eines Cluster-Index ist obligatorisch. Für die Definition wird das **CREATE INDEX**-Statement (siehe 1.4.8) verwendet.

```
CREATE INDEX ang_tf_schicht_ind ON CLUSTER ang_tf_schicht;
CREATE INDEX taxi_ind ON taxi (taxinr);
...
```

Anschließend können die Relationen mit Daten gefüllt werden. Dazu verwendet man das **INSERT**-Statement (siehe 1.4.17).

```
INSERT INTO angestellter (1, 'Hans Muster', 'Blumenweg', 40, 'c');
...
```

Jetzt sind alle Voraussetzungen vorhanden, um Queries auf das implementierte Modell anzuwenden. Die Ergebnisse können mit einem **SPOOL**-Befehl (siehe 1.2) in eine Datei umgeleitet werden.

```
SPOOL ergebnis
CREATE VIEW query1
      AS SELECT ...;

SELECT * FROM query1;

⋮
SPOOL OFF
```

Zum Schluss darf man nicht vergessen, alle erzeugten Datenbankobjekte wieder zu löschen.

```
/* Views löschen */
DROP VIEW query1;

...
/* Indizes löschen */
DROP INDEX taxi_ind;

...
/* Relationen löschen */
DROP TABLE taxifahrer;

...
/* Cluster löschen */
DROP CLUSTER ang_tf_schicht;

...
```

Anhang B

Musterbeispiele und -lösungen

B.1 Ein PL/SQL-Beispiel

Die Anwendung verwendet die in Anhang A definierten Relationen

B.1.1 Aufgabenstellung

Restrukturierung und Einsparungsmaßnahmen:

1. Alle Abteilungen (Relation dept) sollen nach Boston verlegt werden.
2. Für alle Angestellten (Relation emp) soll deren Kommission (Attribut comm) um 10% gekürzt werden.
3. Alle Abteilungen mit weniger als 4 Mitarbeitern sollen in einer neue Abteilung mit dem Namen „RESTRUCT“ zusammengefasst werden. Die Mitarbeiter der Abteilung werden dazu in die Abteilung „RESTRUCT“ transferiert, ihre alte Abteilung wird aufgelöst.

B.1.2 Lösung

```
DECLARE

employees INTEGER;
d_deptno INTEGER := 50;
new_dept CONSTANT VARCHAR2(15) := 'RESTRUCT';
new_loc CONSTANT VARCHAR2(10) := 'BOSTON';

CURSOR alle_boston IS
SELECT *
FROM dept
```

```
FOR UPDATE OF loc;

CURSOR alle_emp IS
SELECT comm
FROM emp
WHERE comm IS NOT NULL
FOR UPDATE OF comm;

CURSOR anz_emp IS
SELECT DISTINCT deptno
FROM emp e1
WHERE 4 > (SELECT COUNT(*)
   FROM emp e2
   WHERE e2.deptno = e1.deptno);


/* Hier erfolgt die Verlegung der Mitarbeiter in das Department
   RESTRUCT und die überflüssigen Departments werden gelöscht */
PROCEDURE processEmployees (dept_no IN INTEGER,
restr_deptno IN INTEGER) IS

CURSOR dept_aendern (dept_no INTEGER DEFAULT 0) IS
SELECT * FROM emp
WHERE deptno = dept_no
FOR UPDATE OF deptno;

CURSOR dept_loeschen (dept_no INTEGER DEFAULT 0) IS
SELECT * FROM dept
WHERE deptno = dept_no;

BEGIN
/* Die Mitarbeiter werden in das neue Department verlegt */
FOR deptaendernrec IN dept_aendern(dept_no) LOOP
UPDATE emp SET deptno = restr_deptno
WHERE CURRENT OF dept_aendern;
END LOOP;

/* Die nun nicht mehr gebrauchten Departments werden gelöscht */
FOR deptloeschenrec IN dept_loeschen(dept_no) LOOP
DELETE FROM dept
WHERE deptno = dept_no;
END LOOP;
```

```
END processEmployees;


BEGIN

/* Aufgabe 1: Alle Departements nach Boston verlegen */
FOR bostonrec IN alle_boston LOOP
UPDATE dept SET loc = new_loc
WHERE CURRENT OF alle_boston;
END LOOP;

/* Aufgabe 2: Alle Kommisionen werden um 10% gekürzt */
FOR emprec IN alle_emp LOOP
UPDATE emp SET comm = emprec.comm * 0.9
WHERE CURRENT OF alle_emp;
END LOOP;

/* Aufgabe 3:  Alle Abteilungen mit weniger als 4 Mitarbeiter werden
   in Abteilung 'RESTRUCT' verlegt */

/* Es wird ein Unterblock angelegt, da das SELECT Statement eine
   Exception auslöst, wenn das Department 'RESTRUCT' nicht
   existiert. Nach einer Exception ist es nämlich nicht möglich
   an die Stelle zurückzukehren, in der die Exception ausgelöst
   wurde, sondern es wird einfach im Code fortgefahren */

BEGIN
/* Falls das Department 'RESTRUCT' noch nicht existiert wird es
               jetzt angelegt */

SELECT deptno INTO d_deptno
FROM dept
WHERE dname = new_dept;

EXCEPTION
/* Dieser Fehler wird ausgelöst, wenn ein SELECT Statement keine
   Tupel zurückliefert */
WHEN NO_DATA_FOUND THEN
INSERT INTO dept VALUES (d_deptno, new_dept, new_loc);
END;

/* Nun werden alle Departments ausgewählt, die weniger als 4
```

```
   Mitarbeiter haben */

FOR anzemprec IN anz_emp LOOP
processEmployees(anzemprec.deptno, d_deptno);
/* Ausgabe der Departmentnummer */
dbms_output.put_line('DEPARTMENT: '  || anzemprec.deptno);
END LOOP;

COMMIT;
EXCEPTION
/* Wenn ein Fehler auftritt wird er hier ausgegeben, und es
   erfolgt ein Rollback */
WHEN OTHERS THEN dbms_output.put_line('Fehler: ' || SQLERRM);
  ROLLBACK;

END;
/
```

Anhang C

EER-Design

T.J. Teorey, D. Yan and J. P. Fry:

„A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model“, ACM Computing Surveys, Vol. 18, No. 2, pp. 197-222, 1986.