PL/SQL

6.1 Allgemeines

In Kapitel 4 haben Sie die Grundlagen der Abfragesprache SQL kennengelernt. In Kapitel 5 sind diese Grundlagen um Elemente wie beispielsweise Sequenzen, Trigger und Stored Procedures erweitert worden. Trotz der recht einfach gehaltenen Beispiele sollte die Flexibilität, die diese Elemente bieten, klar geworden sein. Mit den im letzten Kapitel erläuterten Triggern und Stored Procedures haben Sie die ersten Ansätze von *PL/SQL* kennengelernt. PL/SQL steht für *procedural structured query language* und stellt eine Erweiterung der standardisierten SQL-Syntax dar. Oracle hat die einfache Abfragesprache SQL um prozedurale Elemente und Kontrollstrukturen erweitert, so dass man PL/SQL (fast) als Programmiersprache bezeichnen kann. PL/SQL ist verfügbar ab der Version 6.x der Datenbank von Oracle. Andere professionelle Datenbanken verfügen in der Regel ebenfalls über eine Script-Sprache. Beim Microsoft SQL Server nennt sie sich beispielsweise *Transact SQL*; bei INFORMIX heißt sie SPL für *stored procedure language*.

Die Beispiele für Funktionen und Stored Procedures im Kapitel 5 waren deshalb so einfach gehalten, weil innerhalb des Funktionsrumpfes lediglich SQL-Anweisungen eingebettet waren. Was völlig fehlte, waren Kontrollstrukturen, mit denen man einen echten Programmfluss abbilden kann. Genau diese Elemente sind unter PL/SQL zusammengefasst und sollen in diesem Kapitel näher durchleuchtet werden. Auch für die in diesem Kapitel erläuterten Funktionalitäten und Anweisungen gilt, dass immer ein explizites COMMIT abgesetzt werden muss, bevor die Änderungen an den Daten für alle Anwender sichtbar werden. Aus Gründen der Vereinfachung wird dieses COMMIT nicht bei jedem Listing mit abgedruckt. Oftmals ist es vielleicht auch nicht erwünscht, weil das Programm noch nicht so wie geplant funktioniert. Über ein einfaches ROLLBACK können dann alle Datenänderungen durch ein PL/SQL-Programm wieder rückgängig gemacht werden.

Der ein oder andere Leser mag jetzt vielleicht denken „Nein, nicht schon wieder eine neue Sprache, ich habe gerade erst Turbo Pascal für Windows gelernt". Dieser Einwand ist sicherlich berechtigt. Es ist jedoch zu bedenken, dass PL/SQL gegenüber einer herkömmlichen Programmiersprache wesentliche Vorteile bietet. PL/SQL versteht sich selbst eher als Erweiterung von SQL denn als eigenständige Programmiersprache. So verwendet Oracle selbst PL/SQL-Programme, um die Syntax von SQL zu erweitern. Die Funktion TO_CHAR oder der Operator LIKE sind auf diese Weise implementiert.

Die Vorteile von PL/SQL sind im einzelnen:

- Systemunabhängigkeit, Interpreter und Compiler arbeiten direkt auf der Datenbank, die ebenfalls vom Betriebssystem (Windows NT, Novell, Unix, etc.) unabhängig ist.
- Performance-Vorteile, weil die compilierten Anweisungen direkt auf dem Server ausgeführt werden.
- Geringere Netzbelastung, weil das Datenbankfrontend wesentlich weniger Informationen über das Netz übertragen muss.
- Performance-Gewinne auf der Client-Seite, weil compilierte PL/SQL-Programme lediglich Prozessorzeit des Servers beanspruchen; der Client stößt den Prozess auf dem Server nur an.
- Unabhängigkeit vom Betriebssystem des Clients. PL/SQL-Programme werden mit einem einfachem Texteditor erstellt und über SQL*Plus an die Datenbank gesendet. (Siehe auch die Script-Verarbeitung von SQL*Plus in Kapitel Abschnitt 3.1.1)

PL/SQL ist zusätzlich in diverse Datenbanktools von Oracle wie beispielsweise *Oracle Forms* oder *Oracle Reports* integriert. Beide Programme gehören allerdings nicht zum Lieferumfang des Oracle Servers, sondern müssen zusätzlich erworben werden. Hieran sehen Sie allerdings, dass es durchaus lohnenswert ist, sich ein wenig mit PL/SQL zu beschäftigen. Sollten Sie einmal mit den Tools Oracle Forms und Oracle Reports konfrontiert werden, wird der Einstieg wesentlich leichter fallen. Allerdings ist auch anzumerken, dass Sie mit PL/SQL in Verbindung mit anderen Datenbanken nicht viel anfangen können. Bezüglich der Script-Sprachen „kocht jeder Hersteller sein eigenes Süppchen"; die jeweiligen SQL-Erweiterungen sind zu nichts außer zu sich selbst kompatibel.

Geschrieben werden PL/SQL-Programme genau so, wie Sie in den Kapiteln 3 bis 5 SQL-Scripte erzeugt haben. Sie starten einen beliebigen Texteditor, tragen dort die entsprechenden Zeilen ein und starten dieses Script aus SQL*Plus heraus. Der Compiler der Datenbank parst den ASCII-Text und legt das Programm als Objektcode in der Datenbank ab. Danach können Sie die definierten Prozeduren oder allgemein die Unterroutinen aus Anwendungen heraus aufrufen. Bei einer solchen Anwendung kann es sich um SQL*Plus o.ä. oder um ein Datenbankfrontend handeln, welches für eine bestimmte Anwendung geschrieben wurde.

Dieses Kapitel hat die Aufgabe, die wesentlichen Sprachelemente und Syntaxstrukturen von PL/SQL anhand ausgewählter Beispiele zu erläutern. Nehmen Sie diese Beispiele als Grundlage für die eigenen Programmierversuche in PL/SQL.

Hinweis:
Die Portabilität von PL/SQL hat aber auch einen großen Nachteil, denn der Quelltext ist nicht vor dem Zugriff durch Dritte geschützt. Zu diesem Zweck liefert Oracle mit dem Wrapper WRAP80.EXE einen Compiler aus, der vorhandene Scripte zu Objektcode compiliert. Dieser Objektcode (Endung *.PLB) kann danach verteilt und in die Datenbank eingespielt werden, ohne gleich den Quelltext preisgeben zu müssen. Es handelt sich dabei um ein Kommandozeilentool, welches im Verzeichnis %ORAHOME%/BIN zu finden ist.

6.2 Kommentare

Jede Programmiersprache besitzt die Möglichkeit, im Quelltext Kommentare einzufügen. Diese Anmerkungen tragen nichts zur Funktionalität der Routine bei; sie erhöhen lediglich die Lesbarkeit des Quelltextes. Meine Erfahrungen haben gezeigt, dass Entwickler von dieser Möglichkeit leider jedoch viel zu wenig Gebrauch machen, weil es eben Zeit kostet und man kurzfristig keinen rechenbaren Vorteil von der Kommentierung eines Quelltextes erhält. Mittel- und langfristig ist die Kommentierung von Quelltexten jedoch unabdingbar. Jeder Entwickler hat einen eigenen Programmierstil, der nur marginal durch Richtlinien für die Quellcodegestaltung eingeschränkt wird bzw. eingeschränkt werden kann. Ist man als Entwickler dann gezwungen, den Quelltext eines anderen durchzuarbeiten und womöglich noch zu verändern und Anpassungen vorzunehmen, kann auf Kommentare nicht verzichtet werden. Pflege und Wartbarkeit eines Programmes werden dadurch wesentlich erhöht. Im optimalen Fall sollte eine Zeile Quelltext durch eine Zeile Kommentar erläutert werden.

Innerhalb von PL/SQL-Scripts kennzeichnet man Kommentare auf zweierlei Weise:

- Der Kommentar wird in „/*“ und „*/“ gekapselt. Diese Art der Kommentierung lehnt sich an die Programmiersprache C an.
- Einzeilige Kommentare werden durch zwei „--“-Zeichen eingeleitet. Der Kommentar endet dabei mit dem Zeilenende.

Das folgende Listing zeigt beide Methoden

```
DROP TABLE STATIST;                        /*Tabelle löschen*/

/*um sie hier wieder neu zu erzeugen */

CREATE TABLE STATIST(
 DATUM DATE,
 ANZ_KUND NUMBER);

CREATE OR REPLACE TRIGGER test AFTER       --Ersetzen oder Erzeugen
INSERT OR DELETE ON KUNDEN

DECLARE                                    --Variablendeklaration
   anzahl_kunden Number;

BEGIN                                      --Funktionsrumpf

/*Anzahl der Kunden
  selektieren */

   SELECT COUNT(Kundennr)
   INTO Anzahl_Kunden
   FROM KUNDEN;

   INSERT INTO statist
   (DATUM, ANZ_KUND)
   VALUES
   (sysdate, Anzahl_Kunden);
END;
/
```


6.3 Datentypen, Variablen und Konstanten

6.3.1 Allgemeines

Jede Anwendung basiert darauf, dass man bestimmte Variablen im Programm definiert. Diese werden dann zur Laufzeit des Programmes mit Werten gefüllt, auf denen verschiedene Operationen durchgeführt und dann zur Ausgabe gebracht werden. Auswahl und Anzahl der Variablen und Datentypen beeinflusst in ganz erheblichem Maße die Performance einer Anwendung. Bei Datenbankanwendungen in PL/SQL kommt dieser *richtigen* Auswahl an Variablen eine noch größere Bedeutung zu, weil die Anwendungen in einer Mehrbenutzerumgebung ausgeführt und genutzt werden. Unzureichende Performance eines Programmes kann schnell zu erheblichen Kosten führen, wenn beispielsweise 100 Benutzer fünf Minuten auf eine Antwort der Datenbank warten müssen, nur weil die Anwendung schlecht programmiert ist.

Bei der Programmierung der Funktion *Write_Statistic* aus dem Kapitel 5 haben Sie schon eine lokale Variable für die Funktion verwendet. Ähnlich der Syntax von Turbo Pascal findet eine Deklaration solcher lokaler Variablen vor dem ersten BEGIN innerhalb der Funktion statt:

```
CREATE OR REPLACE FUNCTION Write_Statistic
RETURN NUMBER
IS

  anzahl_kunden NUMBER;

BEGIN
  SELECT COUNT(kundennr)
  INTO   anzahl_kunden
  FROM   kunden;
  RETURN anzahl_kunden;
END;
/
```

Hinweis:
Jede Variable, die innerhalb einer Prozedur, eines Paketes oder allgemein innerhalb eines PL/SQL-Programmes initialisiert wird, erhält den Wert NULL.

6.3.2 Einfache Datentypen

Erlaubte Datentypen innerhalb von PL/SQL sind hier alle Basistypen, die Sie auch zum Erzeugen von Tabellenstrukturen verwenden dürfen. Eine genaue Aufstellung hierüber finden Sie im Anhang, Abschnitt Datentypen. Zusätzlich zu diesen Typen kennt PL/SQL noch eine Reihe weiterer Typen, die in der Tabelle 6-1 aufgelistet sind.

Typ	Beschreibung
BOOLEAN	Speichert Wahrheitswerte, zulässige Werte sind TRUE, FALSE oder NULL.
BINARY_INTEGER	Speichert Integer-Werte im Intervall von -2147483647 bis 2147483647.
NATURAL	Speichert Integer-Werte im Intervall von 0 bis 2147483647.
POSITIVE	Speichert Integer-Werte im Intervall von 1 bis 2147483647.

Tabelle 6-1: *Datentypen von PL/SQL*

Hinweis:
Ab Oracle 7.3 sollten Sie innerhalb von PL/SQL den Datentyp PLS_INTEGER verwenden, wenn Sie mit Ganzzahlen arbeiten. Dieser Datentyp nutzt die interne Integer-Arithmetik, die auf der untersten Ebene, also in Maschinensprache, definiert ist. Hierdurch erzielt man einen zusätzlichen Performance-Gewinn innerhalb einer PL/SQL-Anwendung.

Bei einigen Anwendungen kann es vorkommen, dass Sie eine Variable benötigen, die ein bestimmtes Feld einer bestimmten Tabelle darstellen bzw. beinhalten soll. Zur Entwicklungszeit wissen Sie aber noch nicht konkret, um welchen Datentyp es sich genau handelt. Denkbar wäre auch, dass Sie den Typ zwar kennen, aber nicht mit Sicherheit ausschließen können, dass sich dieser Typ noch einmal ändert. Für diesen Anwendungsfall kennt Oracle den Typ *%Type*. Eine Variablendefinition dieses Typs benötigt zusätzlich die Referenz auf die Tabellenspalte:

```
Variablenname Tabellenname.Spaltenname%TYPE;
```

Variablenname identifiziert die Variable, *Tabellenname.Spaltenname* die Spalte der Tabelle, z.B.

```
DECLARE
   nummer kunden.kundennr%TYPE;
```

Der Vorteil dieser Art der Typ-Definition liegt in ihrer Flexibilität. Man hat damit die Daten und die Routinen, die auf den Daten arbeiten, gegeneinander gekapselt.

Hinweis:
Die direkte Zuordnung einer Variablen zu dem Datentyp einer Tabelle hat noch einen weiteren großen Vorteil. Bei der Programmierung mit PL/SQL müssen die Variablen nicht nur mit dem Datentyp der korrespondierenden Tabellenspalte sondern auch mit der Länge des Typs übereinstimmen. Durch die Verwendung der Referenzvariablen sind beide Bedingungen automatisch erfüllt.

6.3.3 Zusammengesetzte Datentypen

Neben der Referenzierung einer Tabellenspalte kennt Oracle weiterhin die Möglichkeit, eine komplette Zeile einer Tabelle als Variablentyp zu referenzieren. Hier handelt es sich nicht um eine einfache Variable, sondern um eine Daten-Struktur, die genau so viele Elemente beinhaltet bzw. aufnehmen kann, wie Spalten in der Tabelle existieren. Deklariert wird eine solche Variable über *%ROWTYPE:*

```
Variablenname Tabellenname%ROWTYPE;
```

Dazu folgendes Beispiel:

```
DECLARE
   datensatz_kunde KUNDEN%ROWTYPE;
```

So ist sichergestellt, dass die Variable *Datensatz_Kunde* immer alle Elemente beinhaltet, die sich auch in einer Zeile der Kundentabelle befinden. Über die folgende SELECT-Anweisung wird die Datenstruktur mit Werten gefüllt:

```
SELECT * FROM kunden
INTO     datensatz_kunde
WHERE    kundennr = 100;
```

Auf die einzelnen Elemente der Datenstruktur wird über den Punktoperator zugegriffen, so wie Sie es von anderen Programmiersprachen wie beispielsweise Object Pascal oder C her kennen:

```
Datensatz_Kunde.Kundennr;
```

Bei diesem Datentyp handelt es sich um eine zusammengesetzte Struktur, um einen *Record*. Die Struktur dieses Records wird implizit durch die Tabellenstruktur vorgegeben, die referenziert wird. Unter PL/SQL besteht auch die Möglichkeit, benutzerdefinierte Datenstrukturen, Records, zu erzeugen. Solche Strukturen können dann ebenfalls als Basis für Variablendeklarationen dienen. Die Syntax zum Erzeugen eines solchen Records lautet wie folgt:

```
TYPE Record_Name IS RECORD(
      Element_1      Datentyp [NOT NULL] [Startwert],
      Element_2      Datentyp [NOT NULL] [Startwert],
      Element_3      Datentyp [NOT NULL] [Startwert],
      ... );
```

Record_Name identifiziert die Datenstruktur. Die Bezeichnungen *Element_1* bis *Element_n* beschreiben die einzelnen Strukturelemente, wobei jedes dieser Elemente von einem bestimmten Datentyp ist. Optional kann angegeben werden, ob ein Element NULL-Werte enthalten darf und es mit einem Startwert initialisiert werden soll. Als Datentypen sind wieder alle Basis-Datentypen (vgl. Internet) sowie die PL/SQL-spezifischen zugelassen.

Das folgende Script finden Sie auf der beiliegenden CD unter dem Dateinamen RECORD.SQL. Es zeigt die Definition eines Records auf Basis einer Tabelle. Ein weiteres Strukturelement speichert eine Information darüber ab, ob der Kunde einen Fax-Anschluss besitzt:

```
DECLARE

TYPE kundenrecord IS RECORD (
     Name kunden.name%TYPE,
     fax BOOLEAN);

test kundenrecord;

BEGIN
   SELECT name INTO test.name
   FROM   kunden
   WHERE  kundennr = 100;
END;
/
```

Analog können Sie gesamte Tabellenstrukturen über den Datentyp *%ROWTYPE* referenzieren. Hierdurch gestaltet es sich noch einfacher, Records zu erzeugen, die die gesamte Struktur einer bestimmten Tabelle beinhalten und zusätzlich weitere Informationen in neue Strukturelemente aufnehmen können.

6.3.4 Variablen initialisieren

PL/SQL bietet die Möglichkeit, den Variablen bei ihrer Deklaration direkt Werte zuzuweisen. Das Script INIT.SQL zeigt im folgenden einige Beispiele für eine solche Initialisierung. Auch diese Datei finden Sie im Verzeichnis SCRIPTS im Internet:

```
DECLARE

i        INTEGER := 1;
cDummy   VARCHAR2(50) := 'TESTSTRING';

TYPE kundenrecord IS record (
     nummer kunden.kundennr%TYPE := '999',
     name   kunden.name%type     := cDummy,
```



```
     fax    BOOLEAN);

test kundenrecord;

BEGIN
   SELECT name INTO test.name
   FROM   kunden
   WHERE  kundennr = 100;
END;
/
```

Hinweis:
PL/SQL kennt noch eine Reihe weiterer Datentypen, auf die an dieser Stelle allerdings nicht näher eingegangen werden soll. Der wohl wichtigste Typ überhaupt in PL/SQL ist der Typ *CURSOR*. Im Abschnitt 6.6 werden wir uns intensiver mit diesem Typ beschäftigen.

6.4 Operatoren

6.4.1 Relationale Operatoren

Tabelle 6-2 listet die relationalen Operatoren auf, die im Zusammenhang mit der Programmierung in PL/SQL, gerade bei IF-Bedingungen, häufig verwendet werden.

Operator	Beschreibung
<>	ungleich
^=	ungleich
!=	ungleich
>	größer als
<	kleiner als
>=	größer gleich
<=	kleiner gleich

Tabelle 6-2:
Relationale Operatoren

6.4.2 Arithmetische Operatoren

Für Berechnungen im Zusammenhang mit numerischen Variablen und dem Datumstyp sind folgende Operatoren wichtig:

Operator	Beschreibung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
**	Exponentiation

Tabelle 6-3: *Arithmetische Operatoren*

6.4.3 Logische Operatoren

Als logische Operatoren kennt Oracle AND, OR und NOT. Die folgenden Tabellen geben Aufschluss darüber, wie die Wahrheitswerte und Booleschen Ausdrücke miteinander verknüpft werden:

NOT	TRUE	FALSE	NULL
FALSE	TRUE	NULL	

Tabelle 6-4: NOT-*Verknüpfungen*

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Tabelle 6-5: AND-*Verknüpfungen*

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Tabelle 6-6: OR-*Verknüpfungen*

6.5 Kontrollstrukturen

6.5.1 IF-Konstrukt

PL/SQL kennt mit dem IF-Konstrukt eine Möglichkeit, in verschiedene Programmteile zu verzweigen. Die Syntax ist ähnlich der Programmiersprache C. Allgemein lautet sie:

```
IF Bedingung_1 THEN
   Anweisung_1;
   Anweisung_2;
   ...
END IF;
```

Bedingung_1 liefert einen Wahrheitswert, also TRUE oder FALSE. In der Regel findet hier ein Vergleich über die relationalen Operatoren statt. Bei TRUE werden alle Anweisungen ausgeführt, die zwischen THEN und END IF stehen. Bei diesen Anweisungen kann es sich um einfache SQL-Anweisungen oder um Funktions- oder Prozeduraufrufe handeln.

Optional kann ein ELSE-Zweig eingefügt werden:

```
IF Bedingung_1 THEN
   Anweisung_1;
   Anweisung_2;
ELSE
   Anweisung_3;
   Anweisung_4;
END IF;
```

Sobald die *Bedingung_1* TRUE zurückliefert, werden *Anweisung_1* und *Anweisung_2* ausgeführt. Liefert die Bedingung hingegen FALSE zurück, so werden die Anweisungen *Anweisung_3* und *Anweisung_4* ausgeführt. Es kann jedoch nur ein ELSE-Zweig in die Abfragekonstruktion eingebracht werden. Soll das Programm in mehr als zwei Programmpfade verzweigen, so sind zusätzliche ELSIF-Elemente einzuführen:

```
IF Bedingung_1 THEN
   Block_1;
ELSIF Bedingung_2 THEN
   Block_2;
ELSE
   Block_3;
END IF;
```

Wenn *Bedingung_1* TRUE liefert, wird der Teil *Block_1* ausgeführt. Wenn diese Bedingung allerdings FALSE zurückliefert, gelangt das Programm in den ELSIF-Teil und prüft *Bedingung_2*. Ist diese TRUE, so werden die Anweisungen in *Block_2* ausgeführt. Erst

wenn diese Bedingung ebenfalls FALSE zurückliefert, wird *Block_3* aus dem ELSE-Teil ausgeführt. ELSIF-Teile sind im Prinzip in unbegrenzterAnzahl erlaubt. Es ist zu beachten, dass der ELSE-Teil sich immer auf die erste IF-Abfrage bezieht, im obigen Fall also auf *Bedingung_1*.

Dadurch werden Probleme verhindert, die Ihnen vielleicht bei der Programmierung in C aufgefallen sind. Das folgende Beispiel eines einfachen C-Programmes zeigt dies:

```
int a,b,c:
a=5;
b=7;
c=9;

if (a<b)
  if (a<c)
    cout << "a ist die kleinste Zahl";
else cout << "a ist nicht kleiner als b";
```

Anhand der Schreibweise ist leicht zu erkennen, dass der else-Teil sich auf die erste if-Anweisung bezieht. Dies kann der C-Compiler allerdings nicht so ohne weiteres wissen. Er würde ein Programm erzeugen, das folgender Logik entspräche:

```
int a,b,c:
a=5;
b=7;
c=9;

if (a<b)
  if (a<c)
    cout << "a ist die kleinste Zahl";
  else cout << "a ist nicht kleiner als b";
```

Das folgende Beispiel zeigt die Verwendung einer IF-Bedingung innerhalb einer IF-Bedingung:

```
IF Bedingung_1 THEN
   IF Bedingung_2 THEN
      Block_1;
   ELSE Bedingung_3 THEN
      Block_2;
   ENDIF;
ELSE
   Block_3;
END IF;
```

Sie sehen, auch solche Konstruktionen sind möglich. Wichtig ist, dass jedes IF-Konstrukt mit ENDIF in sich abgeschlossen wird. Bei diesem Beispiel bezieht sich die erste ELSE-Anweisung auf das zweite IF-Konstrukt. *Block_2* wird also nur dann ausgeführt, wenn *Bedingung_1* TRUE und *Bedingung_2* FALSE liefert.

6.5.2 LOOP-Konstrukt

Über das LOOP-Konstrukt werden in PL/SQL Schleifen programmiert. Man unterscheidet dabei grundsätzlich drei verschiedene Schleifentypen:

- FOR-LOOP
- WHILE-LOOP
- LOOP-EXIT

6.5.2.1 FOR-LOOP

Eine Konstruktion über FOR-LOOP wird immer dann verwendet, wenn man schon zur Entwicklungszeit genau weiß, wie oft die Schleife durchlaufen werden soll. In der Regel verwendet man dabei einen numerischen Schleifenzähler. Es ist auch möglich, FOR-LOOP-Konstruktionen in Verbindung mit Cursorn, die im Abschnitt 6.6 erläutert werden, zu verwenden. Die allgemeine Syntax lautet:

```
FOR Zähler IN [REVERSE]              Untergrenze..Obergrenze LOOP
   Anweisungsblock;
END LOOP;
```

Zähler identifiziert dabei die Laufvariable, *Untergrenze* die kleinere Intervallgrenze, *Obergrenze* die größere Intervallgrenze. Das folgende Beispiel finden Sie im Internet unter dem Dateinamen FORLOOP.SQL:

```
DECLARE
  i INTEGER;

BEGIN

FOR i IN 500..599 LOOP
   INSERT INTO kunden
   (kundennr, anlage)
   VALUES
   (i, SYSDATE);
END LOOP;

END;
/
```

Optional kann noch das Schlüsselwort REVERSE eingefügt werden. Hierdurch wird der Zähler dekrementiert, während er ohne die Angabe von REVERSE per Voreinstellung inkrementiert wird. Das folgende Listing zeigt die Verwendung von REVERSE. Hier ist zu

beachten, dass die untere Intervallgrenze immer vor der oberen Intervallgrenze angegeben werden muss:

```
DECLARE
   i INTEGER;

BEGIN

FOR i IN REVERSE 500..599 LOOP
   INSERT INTO kunden (kundennr, anlage)
   VALUES             (i, SYSDATE);
END LOOP;

END;
/
```

Wie Abbildung 6-1 zeigt, sind über die obige Anweisung 99 Sätze in der Kundentabelle eingefügt worden. Über solche Schleifen haben Sie also eine Möglichkeit, schnell eine Testdatenbank mit einer großen Anzahl an Datensätzen aufzubauen. Gerade im Hinblick auf einen Performance-Test kann dies manchmal sehr hilfreich sein.

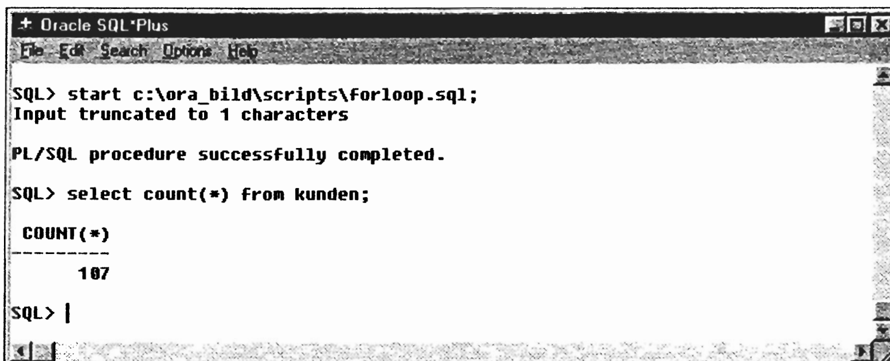

Abbildung 6-1: *Datensätze in einer Schleife erzeugen*

6.5.2.2 WHILE-LOOP

Bei der `WHILE-LOOP`-Konstruktion spricht man von einer abweisenden Schleife. Sie wird immer dann benutzt, wenn man zur Entwicklungszeit noch nicht genau weiß, wie diese Schleife durchlaufen werden soll. Vor Eintritt in den Schleifenrumpf muss also eine Bedin-

gung überprüft werden. Im ungünstigsten Fall liefert diese Bedingung `FALSE` zurück, so dass die Schleife keinmal durchlaufen wird. Die allgemeine Syntax lautet:

WHILE *Bedingung* LOOP
 Anweisungsblock
END LOOP

Bei dem folgenden Beispiel handelt es sich um das Script **WHILE.SQL** aus dem Netz. Über eine `WHILE`-Schleife werden hier die Datensätze aus der Kundentabelle gelöscht, die zuvor über die `FOR-LOOP`-Schleife eingefügt wurden. Dies ist zwar nicht die sauberste Art der Programmierung. Als Beispiel für die `WHILE`-Schleife ist das folgende Listing jedoch sehr dienlich:

```
DECLARE
   i     INTEGER;
   bignr INTEGER;
   to_do BOOLEAN;

BEGIN

to_do := true;
i := 500;

WHILE to_do LOOP
   SELECT MAX(kundennr) INTO bignr FROM KUNDEN;

   IF bignr > 499 THEN
      DELETE FROM kunden
      WHERE kundennr = bignr;
   ELSE
      to_do := FALSE;
   END IF;

END LOOP;

END;
/
```

Die Abbildung 6-2 zeigt, dass sich die Anzahl Datensätze in der Kundentabelle wieder auf sieben reduziert hat.

```
Oracle SQL*Plus
File Edit Search Options Help

SQL> start c:\ora_bild\scripts\while.sql;
Input truncated to 1 characters

PL/SQL procedure successfully completed.

SQL> select count(*) from kunden;

 COUNT(*)
---------
        7

SQL>
```

Abbildung 6-2: *Datensätze löschen*

6.5.2.3 LOOP-EXIT

Bei der `LOOP-EXIT`-Konstruktion handelt es sich im Prinzip um das Pendant zur abweisenden `WHILE`-Schleife. `LOOP-EXIT` sollte immer dann zum Einsatz kommen, wenn man die genaue Anzahl an Schleifendurchläufen nicht kennt, die Schleife aber in jedem Fall mindestens einmal durchlaufen wird. Dieser Typ kann auch elegant genutzt werden, um Endlosschleifen zu programmieren. Die allgemeine Syntax lautet:

```
LOOP
  Anweisungsblock
  [EXIT WHEN Bedingung]
END LOOP
```

Wie Sie sehen, handelt es sich bei der `EXIT`-Anweisung um eine optionale Erweiterung dieses Schleifentyps. Ohne eine solche Abbruchbedingung erhält man genau die eben erwähnte Endlosschleife. Mit der `EXIT`-Anweisung wird die Schleife dann beendet, wenn *Bedingung* `TRUE` zurückliefert.

Das folgende Beispiel zeigt das Script EXIT.SQL:

```
DECLARE
   i INTEGER;

BEGIN

i := 500;
```



```
LOOP
   INSERT INTO kunden (kundennr, anlage)
   VALUES (i, sysdate);

   i := i + 1;
   EXIT WHEN i = 600;
END LOOP;

END;
/
```

Auch wenn es vielleicht nicht besonders kreativ ist, jetzt wieder 99 Datensätze einzufügen, so zeigt dieses Listing doch deutlich die Unterschiede zu dem `FOR-LOOP`-Konstrukt.

6.5.3 GOTO-Anweisung

Die Erwähnung der `GOTO`-Anweisung, die in PL/SQL implementiert ist, erscheint problematisch. `GOTO` springt an eine andere Stelle im Programm, die durch ein *Label* gekennzeichnet ist. Mit dieser Anweisung ist dem Erstellen von Spaghetti-Code innerhalb von PL/SQL Tür und Tor geöffnet. Ich bin der Überzeugung, dass jedes `GOTO` innerhalb eines Programmes durch eine sauber strukturierte, prozedurorientierte Programmierung vermieden werden kann. Leider bieten auch Sprachen wie C diese unglückselige Anweisung. Der Vollständigkeit halber sei sie hier jedoch erwähnt.

Innerhalb eines Programmes muss zunächst ein Label, das als Sprungziel fungiert, definiert sein. Die Kennzeichnung eines solchen erfolgt folgendermaßen:

<<Label>

Das folgende Listing zeigt die Verwendung des `GOTO`-Befehls:

```
DECLARE
   i INTEGER;

BEGIN

<<anfang>>

i := 500;

LOOP

   i := i - 1;

   IF i < 100 THEN
      GOTO ende
```

```
   END IF;
END LOOP;


<<ende>
GOTO ANFANG
END;
/
```

6.5.4 NULL-Anweisung

Es kann bei bestimmten Problemstellungen vorkommen, dass Sie die Datenbank anweisen möchten *nichts* zu tun. Im Zusammenhang mit dem *Exception Handling* wird diese Funktion noch an Bedeutung gewinnen, z.B.:

```
IF i = 0 THEN
   i := i + 1;
ELSE
   NULL;
END IF;
```

Der folgende Funktionsblock zeigt, dass die Variable *i* um 1 inkrementiert werden soll, falls sie den Wert 0 besitzt. Ansonsten verzweigt die if-Bedingung in den else-Teil, in dem dann *nichts* weiter passiert. Dieses NULL hinter dem ELSE ist nicht mit dem Typ NULL zu verwechseln, den man beispielsweise als Feldinhalt findet. Hierbei handelt es sich um eine Anweisung, die einfach besagt, nichts zu tun.

6.6 Cursor

6.6.1 Allgemeines zu Cursorn

Im Abschnitt 6.5.2 haben Sie die verschiedenen Schleifen-Typen von PL/SQL kennengelernt. Im Abschnitt über die WHILE-LOOP-Konstruktion habe ich dazu folgendes Beispiel programmiert:

```
DECLARE
  i      INTEGER;
  bignr  INTEGER;
  to_do  BOOLEAN;

BEGIN

to_do := true;
i := 500;

WHILE to_do LOOP
   SELECT MAX(kundennr) INTO bignr FROM KUNDEN;

   IF bignr > 499 THEN
      DELETE FROM kunden
      WHERE kundennr = bignr;
   ELSE
      to_do := false;
   END IF;

END LOOP;

END;
/
```

Aus programmiertechnischer Sicht ist diese Aufgabenstellung, nämlich das Löschen aller Datensätze, die eine Kundennummer > 499 besitzen, denkbar schlecht gelöst. Innerhalb einer WHILE-Schleife wird jeweils die größte Kundennummer über eine SELECT-Anweisung geholt. Danach erfolgt eine Prüfung, ob diese Kundennummer > 499 ist. Falls ja, wird der entsprechende Satz gelöscht; bei nein wird eine Boolesche Variable gesetzt, damit die Schleife danach beendet werden kann. Sollte sich in der Kundentabelle eine große Anzahl an Sätzen befinden, wird das obige Programm ein großes Stück aus dem Kuchen der zur Verfügung stehenden Prozessorzeit herausschneiden, denn pro Datensatz wird eine SELECT- und eine DELETE-Anweisung an die Datenbank abgesetzt.

Hinweis:
Der natürlich schnellste Weg zur Lösung einer solchen Aufgabenstellung, besteht natürlich darin, genau eine Anweisung abzusetzen:

```
DELETE FROM kunden
WHERE  kundennr > 499;
```

Allerdings erzielen wir mit einer solchen Vorgehensweise keinen Lerneffekt. Schließlich soll es in diesem Kapitel doch um die Arbeit mit Datenbank-Cursorn und um die Programmierung mit PL/SQL gehen.

Cursor werden dann benötigt, wenn zu bearbeitende Daten über ein SELECT-Statement gelesen werden, dieses aber mehr als einen Datensatz zurückliefert. Bei Öffnen des Cursors wird das SELECT-Statement komplett ausgeführt, die Ergebnisse aber nicht an das Client-Programm übermittelt. Statt dessen kann der Client mit der FETCH-Anweisung jeden Eintrag aus dieser Ergebnisliste einzeln sequentiell lesen und verarbeiten. Die komplette Ergebnismengeliste bleibt solange in der DB-Engine gespeichert, bis sie mit CLOSE CURSOR gelöscht wird. Dies illustriert das folgende Beispiel:

```
DECLARE
  nummer      kunden.kundennr%TYPE;
  nachname    kunden.name%TYPE;

BEGIN
  SELECT kundennr, name
  INTO   nummer, nachname
  FROM   kunden;
END;
/
```

Das entsprechende Script finden Sie im Netz unter dem Dateinamen CURSOR1.SQL. In diesem Script werden zwei Variablen deklariert, die mittels einer nachfolgenden SELECT-Anweisung gefüllt werden sollen. Bei dem Versuch, dieses Script in SQL*Plus zu starten, erhält man folgende Fehlermeldung:

```
SQL> start c:\scripts\cursor1.sql;
declare
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5


SQL>
```

Da innerhalb der SQL-Anweisung keine genauere Einschränkung der zu selektierenden Daten definiert ist, umfasst die Ergebnismenge den kompletten Inhalt der Tabelle *Kunden*. Für solche Anwendungszwecke kennt Oracle *Cursor*-Objekte. Ein solcher Cursor wird definiert, um die Ergebnismenge einer SELECT-Anweisung aufzunehmen. Zu einem späteren Zeitpunkt im Programmablauf werden Operationen auf dieser Ergebnismenge ausgeführt. Nach Initialisierung eines Cursors kann innerhalb eines PL/SQL-Programmes auf die Ergebnismenge des Cursors zurückgegriffen werden. Die Abbildung 6-3 zeigt schematisch Definition und Verwendung eines Cursors.

Cursordefinition:

```
declare

cursor Dummy is
       select * from kunden
       where kundennr > 101
       and kundennr < 105;
```

Ergebnismenge:

```
KUNDENNR NAME       VORNAME  STRASSE          PLZ   ANLAGE
     100 Meier      Michael  Rosenstr. 8      66666 01-MAY-97
     101 Müller     Sabine   Liebigstr. 8     66666 03-MAY-97
     102 Behring    Thomas   Im Weingarten 1  12345 02-JAN-97
     103 Zimmermann Petra    Hauptstr. 3      54321 02-JAN-97
     104 Schröder   Martin   Landstr. 1       99999 01-APR-97
     105 Oppermann  Monika   Fasanenweg 2     32100 16-JUN-97
```

Abbildung 6-3: *Cursor mit entsprechender Ergebnismenge*

Die Ergebnismenge der Cursordefinition aus Abbildung 6-3 finden Sie im unteren Bildabschnitt in der eingegrenzten Fläche wieder. Falls Sie schon einmal mit Borland Delphi oder Borland C++-Builder gearbeitet haben, werden Sie viele Parallelen zwischen den Cursorn von Oracle und der Komponente *TQuery* der Entwicklungssysteme erkennen. Auch hier kann man nach Ausführung der SQL-Anweisung des TQuery-Objektes auf die Ergebnismenge zurückgreifen. (Abhängig vom verwendeten Datenbank-Treiber ist es sogar möglich, Änderungen in der Ergebnismenge vorzunehmen, die sich *live* auf den tatsächlichen Datenbestand auswirken!)

6.6.2 Cursor erzeugen

Die Arbeit mit Cursorn innerhalb eines PL/SQL-Programmes setzt zunächst voraus, dass ein solcher Cursor deklariert wurde. Bezüglich der Deklaration lässt er sich mit einfachen Variablentypen vergleichen, denn ein Cursor kann innerhalb eines PL/SQL-Blockes oder innerhalb einer Stored Procedure deklariert werden, eben genau so wie Variablen auch. Eine Cursor-Deklaration muss folgender Syntax entsprechen:

```
DECLARE

CURSOR Cursorname
Parameterliste
IS
SELECT-Anweisung
```

Cursorname identifiziert dabei den Namen des Cursors. Für die Bezeichnungen von Cursorn gelten die gleichen Regeln der Nomenklatur wie für alle anderen Objekte der Datenbank auch. Optional kann einem Cursor eine beliebige Anzahl von Parametern mitgegeben werden. Diese Parameterliste ist mit der einer Stored Procedure vergleichbar. Es können mehrere Parameter übergeben werden, wobei sie jeweils durch Komma getrennt werden. Jeder Parameter kann dabei noch mit einem Wert vorbelegt werden. Dieser Wert stellt dann die Standardeinstellung dar. Die Parameterliste eines Cursors dient dazu, die *SELECT-Anweisung* genauer zu spezifizieren bzw. die Selektionsbedingungen der SQL-Anweisung zur Laufzeit des PL/SQL-Programmes zu modifizieren.

6.6.3 Cursor öffnen und schließen

Nach erfolgter Deklaration eines Cursors muss dieser *geöffnet* werden. Unter dem *Öffnen eines Cursors* versteht man die Ausführung der SELECT-Anweisung. Vor dieser Ausführung findet zunächst noch die Allozierung von Hauptspeicher statt, in dem dann die Ergebnismenge abgelegt werden kann. Das Öffnen eines Cursors findet über den OPEN-Befehl statt:

```
SQL> DECLARE
  2
  2  i INTEGER;
  3
  3  CURSOR Delete_Garbage is
  4         SELECT kundennr FROM kunden
  5         WHERE kundennr > 499;
  6
  6  BEGIN
  7
  7    OPEN Delete_Garbage;
  8
  8  END;
  9  /

PL/SQL procedure successfully completed.

SQL>
```


Das folgende Beispiel zeigt die Verwendung eines Cursors innerhalb eines PL/SQL-Blockes. Das Programm hat die Aufgabe, die Kundentabelle von den (überflüssigen) Datensätzen zu befreien, die bei der Arbeit mit Schleifen im Abschnitt 6.5 als Beispieldaten entstanden sind. Es stellt ferner eine Verbesserung des Listings dar, mit dem dieses Kapitel eingeleitet wurde. Sie finden es im Internet unter dem Dateinamen CURSOR2.SQL.

```
DECLARE

i INTEGER;

CURSOR delete_garbage is
       SELECT kundennr FROM kunden
       WHERE  kundennr > 499;

BEGIN

  OPEN delete_garbage;

  LOOP
    FETCH delete_garbage
    INTO  i;
    EXIT WHEN delete_darbage%NOTFOUND;

    DELETE from kunden
    WHERE  kundennr = i;

  END LOOP;

  CLOSE delete_garbage;

END;
/
```

Innerhalb eines Cursors ist ein Datensatzzeiger definiert, der auf den aktuellen Datensatz der Ergebnismenge zeigt. Direkt hinter einer `OPEN`-Anweisung eines Cursors befindet sich dieser Zeiger auf dem ersten Satz. Zugriff auf den durch den Datensatzzeiger definierten Satz erhält man über die `FETCH`-Anweisung. Die Cursor-Bezeichnung folgt ihr. Ähnlich dem `SELECT INTO` gibt es hier ein `FETCH INTO`. Hinter dem `INTO`-Schlüsselwort stehen die Variablen, in denen Werte aus der Ergebnismenge eingelesen werden sollen. Jede `FETCH`-Anweisung setzt den Datensatzzeiger um einen Satz weiter nach unten. Ist das Ende der Ergebnismenge erreicht, wird etwas ähnliches wie „End of File" zurückgeliefert. Innerhalb von Oracle ist dieser Parameter als „`%NOTFOUND`" definiert. Vorangestellt wird hier jeweils der Cursorname. Über diese Anweisung wird die Schleife des obigen Listings verlassen. Abbildung 6-4 verdeutlicht noch einmal die Arbeit der `FETCH`-Anweisung.

Hinweis:
Wenn versucht wird, einen bereits geöffneten Cursor nochmals mittels OPEN zu öffnen, wird eine Exception vom Typ CURSOR_ALREADY_OPEN erzeugt. Bei komplizierteren Programmen kann eine Ausnahmebehandlung dieses Ereignisses manchmal recht nützlich sein.

	KUNDENNR	NAME	VORNAME	STRAßE	PLZ	ANLAGE
	100	Meier	Michael	Rosenstr. 8	66666	01-MAY-97
FETCH	101	Müller	Sabine	Liebigstr. 8	66666	03-MAY-97
FETCH	102	Behring	Thomas	Im Weingarten 1	12345	02-JAN-97
FETCH	103	Zimmermann	Petra	Hauptstr. 3	54321	02-JAN-97
FETCH	104	Schröder	Martin	Landstr. 1	99999	01-APR-97
FETCH	105	Oppermann	Monika	Fasanenweg 2	32100	16-JUN-97
FETCH	%NOTFOUND					

Abbildung 6-4: *FETCH-Anweisung und %NOTFOUND-Meldung*

Die Abbildung 6-4 zeigt, dass die Eigenschaft %NOTFOUND erst dann TRUE liefert, wenn die FETCH-Anweisung keine Daten mehr erhalten kann. Der Datensatzzeiger muss also über ein letztes FETCH-Statement *hinter* dem letzten Satz der Ergebnismenge positioniert werden.

Das folgende Listing zeigt die Verwendung von Cursorn, denen zur Laufzeit Parameter übergeben werden. Im Internet finden Sie dieses Beispiel unter dem Dateinamen CURSOR3.SQL:

```
DECLARE

i INTEGER;

CURSOR delete_garbage (grenze number default 499) IS
       SELECT kundennr FROM kunden
       WHERE  kundennr > grenze;

BEGIN
  OPEN delete_garbage(300);
  LOOP
     FETCH delete_garbage
```



```
     INTO  i;
     EXIT  WHEN delete_garbage%NOTFOUND;

     DELETE FROM kunden
     WHERE kundennr = i;

  END LOOP;
  CLOSE delete_garbage;
END;
/
```

Im Zuge der Cursoröffnung wird diesem ein Parameter mitgegeben, der die SQL-Anweisung innerhalb des Cursors modifiziert. Versuchen Sie einmal als Übung das obige Listing dahingehend zu modifizieren, dass dem Cursor sowohl eine Ober- als auch eine Untergrenze übergeben wird.

Über das CLOSE-Statement wird ein solcher Cursor wieder geschlossen. Nach Verwendung eines Cursors innerhalb eines PL/SQL-Programmes sollte dieser schnellstmöglich wieder geschlossen werden, damit auch die von ihm allozierten Ressourcen auf dem Server wieder freigegeben werden. Es gibt aber noch einen weiteren Grund, warum man Cursor nach deren Verwendung sofort wieder schließen sollte. Bei einigen Anwendungen kommt es häufiger vor, dass ein Cursor mehrmals im Programm verwendet wird bzw. mehrmals auf ihn zugegriffen wird. Wie oben gesehen, werden einem Cursor Parameter innerhalb der OPEN-Anweisung übergeben. Unterschiedliche Parameter erfordern deshalb ein Neuöffnen des Cursors. Dies funktioniert aber nur, wenn Sie diesen zuvor geschlossen haben. (Gleiches gilt übrigens auch für die TQuery-Komponente aus der Borland-Entwicklerschmiede)

Über den DEFAULT-Zusatz innerhalb der Deklaration wird der Parameter mit einem Startwert initialisiert. Sie werden sich sicherlich fragen, wozu denn eine Initialisierung des Parameters notwendig ist, wenn dieser doch bei Aufruf des OPEN-Befehls ohnehin überschrieben wird. Die Antwort ist einfach. Durch Vorbelegung eines Parameters definieren Sie diesen als optionalen Parameter, d.h. er könnte in der OPEN-Anweisung weggelassen werden. Anhand der Parameter-Übergabe erkennt man gut die nahe Verwandtschaft von PL/SQL und C.

6.6.4 Eigenschaften von Cursorn

Über die Eigenschaft %NOTFOUND haben wir in dem obigen Listing das Verlassen der Schleife realisiert. Neben dieser gibt es noch eine Reihe weiterer Eigenschaften von Cursorn. Die Eigenschaften werden abgefragt, indem man den Cursornamen vor die Eigen-

schaft stellt (Cursorname%Eigenschaft). Tabelle 6-6 listet die Eigenschaften von Cursorn auf.

Eigenschaft	Funktion
%NOTFOUND	Liefert einen Wahrheitswert darüber, ob sich der Datensatz am Ende der Ergebnismenge befindet.
%FOUND	Liefert einen Wahrheitswert darüber, ob sich der Datensatz am Ende der Ergebnismenge befindet. Er entspricht NOT %NOTFOUND.
%ROWCOUNT	Liefert die Anzahl der Datensätze, die sich innerhalb der Ergebnismenge eines Cursors befinden, nachdem dieser geöffnet wurde. Hierüber wird auch die Verwendung von FOR-LOOP-Konstruktionen möglich.
%ISOPEN	Liefert einen Wahrheitswert darüber, ob ein Cursor geöffnet ist. Im Hinblick auf die Verwendung von mehreren Cursorn innerhalb eines PL/SQL-Blockes ist diese Eigenschaft wichtig.

Tabelle 6-6: *Eigenschaften von Cursorn*

Hinweis:
Im Verzeichnis PLSQL80 der Server-Installation finden Sie eine Reihe von Beispielscripten. Deren genaue Analyse verhilft Ihnen zu einem tieferen Verständnis dieser Scriptsprache.

6.6.5 Cursor-Referenzen

Im Zusammenhang mit der Definition von Cursorn kommt noch den *Cursor-Referenzen* besondere Bedeutung zu. Es handelt sich hierbei um einen Datentyp, der einen Zeiger auf eine Ergebnismenge aufnehmen kann; implementiert ist dieser Typ ab der Version 8 der Datenbankengine. Dieser Zeiger zeigt auf die Ergebnismenge eines geöffneten Cursors. Interessant sind diese Cursor insbesondere dann, wenn von Anwendungsprogrammen bzw. Datenbankfrontends auf die Ergebnismengen zurückgegriffen werden soll. Das folgende Script zeigt die Definition einer solchen Referenz. Im Kapitel 10 wird diese Referenz in einem Delphi-Programm dann weiter verarbeitet.

```
CREATE OR REPLACE PACKAGE cur_ref IS

TYPE kunden_ref IS REF CURSOR RETURN kunden%ROWTYPE;

END cur_ref;
/
```


Damit ist der neue Datentyp *kunden_ref* erzeugt worden. Die folgende Prozedur nutzt diese Referenz zur Rückgabe einer Ergebnismenge:

```
CREATE OR REPLACE PROCEDURE get_kunden(cur IN OUT cur_ref.kunden_ref) is
BEGIN
   OPEN cur FOR SELECT * FROM kunden;
END;
 /
```

Hinweis:
Die obige Deklaration der Cursor-Referenz zeigt, dass zum Zeitpunkt der Typ-Deklaration der Typ des Rückgabewertes (kunden%ROWTYPE) schon bekannt ist. Obwohl PL/SQL auch ungebundene Cursor-Referenzen, bei deren Deklaration also der Rückgabetyp noch nicht feststeht, kennt, rate ich von deren Verwendung ab. Gerade bei größeren PL/SQL-Programmen ist außer dem Autor selbst kaum jemand in der Lage den Quelltext vernünftig interpretieren zu können, sofern ungebundene Cursor-Referenzen verwendet werden.

6.6.6 CURSOR-FOR-Loops

Im Zusammenhang mit Cursorn kennt PL/SQL noch eine besondere Variante der FOR-Schleife. Bislang wurde ein Cursor immer definiert und geöffnet; auf die Datenmenge wurde dann mittels einer LOOP-END LOOP-Schleife zugegriffen, die eine EXIT-Abfrage für den Abbruch beinhaltete. Mit Hilfe einer FOR-Schleife kann man genau diese Aufgabe wesentlich eleganter lösen, wie das folgende Script zeigt. Im Internet finden Sie dieses Script unter dem Dateinamen CUR_FOR.SQL:

```
DECLARE

CURSOR liste IS
       SELECT kundennr, name FROM kunden;

BEGIN
  dbms_output.enable;

  FOR kundensatz IN liste
  LOOP
      dbms_output.put(kundensatz.kundennr);
      dbms_output.put_line(kundensatz.name);
  END LOOP;

END;
/
```

Der gesamte Administrationsprozess des Cursors findet hier innerhalb des Schleifenkopfs, der fett gedruckten Zeile, statt. Sofern der Cursor noch nicht geöffnet ist, wird dieser geöffnet. Falls dies schon der Fall ist, wird bei jedem Schleifendurchlauf der Datensatzzeiger um 1 nach unten bewegt. Implizit findet bei jedem Fetch automatisch eine Überprüfung statt, ob die Variable %NOTFOUND gesetzt ist. Falls dies der Fall sein sollte, wird die Bearbeitung des Programms nach dem END LOOP fortgesetzt und der Cursor wird wieder geschlossen. Außerhalb der Schleife kann man also nicht mehr auf die Ergebnismenge des Cursors zurückgreifen.

Besondere Beachtung verdient die Variable *kundensatz*. Sie muss nicht im DECLARE-Teil der Anwendung deklariert werden, sondern wird zur Laufzeit des Programms, genauer bei Eintritt in die Schleife, erzeugt. In ihr ist die gesamte Ergebnismenge des Cursors abgelegt. Benötigte Ressourcen hierfür werden ebenfalls erst zur Laufzeit alloziert. Bei der Variablen *kundensatz* handelt es sich nur um eine einfache Laufvariable für die Schleife, es stellt vielmehr einen Record dar.

6.6.7 **Ein Beispielscript** (Datensätze durchnummerieren)

Die Verwendung von Cursorn eignet sich hervorragend für die Manipulation großer Datenmengen innerhalb einer Datenbank. Dabei kann ein solcher Cursor mit dem Zusatz FOR UPDATE definiert werden. Dadurch wird die selektierte Datenmenge gesperrt; und es ist sichergestellt, dass kein anderer Benutzer genau diese Informationen während dessen ändern kann. Der folgende Cursor reserviert eine Datenmenge zum updaten:

```
CURSOR auto_number IS
    SELECT empno FROM emp
    FOR UPDATE;
```

In diesem Beispiel ist die zu ändernde Tabellenspalte eindeutig durch die Selektion spezifiziert. Die folgende Variante ist aber auch syntaktisch korrekt:

```
CURSOR auto_number IS
    SELECT * FROM emp
    FOR UPDATE OF empno;
```

Das folgende Script zeigt ein sehr nützliches Beispiel für die Programmierung der Datenbank mit Hilfe von PL/SQL. Ich habe schon bei verschiedenen Aufgaben an Datenbanken eine Tabelle dahingehend verändern müssen, dass die einzelnen Sätzen einer bestimmten Tabelle fortlaufend durchzunumerieren waren. Beim dritten Mal habe ich endlich entschieden, das Script zu speichern. Ich hoffe, dass dieses Beispiel auch Ihnen ein wenig helfen mag. Als Beispiel habe ich die Tabelle *EMP* der Beispieldatenbank von Oracle gewählt. Zugriff auf diese Tabelle erhalten Sie über die Anmeldung als *Scott*. Das Passwort ist *Tiger*. Eindeutiger Schlüssel innerhalb dieser Tabelle ist das Feld *empno*. Zunächst wird die Tabelle um eine weitere Spalte erweitert, die dann im nächsten Schritt mit einer fortlaufenden Nummer gefüllt wird. Im Internet finden Sie das Script unter dem Dateinamen (AUTONR.SQL):


```
ALTER TABLE emp ADD ident_nr number(5);

DECLARE
  i      INTEGER;
  nummer INTEGER;

  CURSOR auto_number IS
    SELECT empno FROM emp
    FOR UPDATE;

BEGIN
  OPEN   auto_number;
  nummer := 1;
  LOOP
     FETCH auto_number
     INTO  i;

     EXIT WHEN auto_number%NOTFOUND;

     UPDATE emp
     SET    ident_nr = nummer
     WHERE  empno = i;

     nummer := nummer + 1;
   END LOOP;
END;
/
```

Vielleicht fragen Sie sich jetzt, was bei einer solchen Anforderung, wie oben beschrieben zu tun ist, wenn kein eindeutiger Schlüssel (in diesem Fall die Spalte *empno*) vorhanden ist. Hier kommt uns die der Schlüssel *rowid* zu Hilfe. Es ist durchaus möglich, eine Tabellenstruktur zu erzeugen, die keinen eindeutigen Schlüssel, also keinen Primärschlüssel aufweist. Dennoch kann Oracle die einzelnen Zeilen einer Spalte eindeutig über die *rowid* identifizieren. *rowid* ist eine eindeutige Nummerierung bzw. Adressierung eines Satzes innerhalb der Speicherstruktur. Das folgende Listing (AUTONR2.SQL) zeigt die Verwendung dieser *rowid*:

```
ALTER TABLE emp ADD ident_nr NUMBER(5);

DECLARE
  i      INTEGER;
  nummer INTEGER;

  CURSOR auto_number is
    SELECT rowid FROM emp
    FOR UPDATE;
```

```
BEGIN
  OPEN auto_number;
  nummer := 1;
  LOOP
     FETCH auto_number
     INTO i;

     EXIT WHEN auto_number%NOTFOUND;

     UPDATE emp
     SET    ident_nr = nummer
     WHERE  rowid = i;

     nummer := nummer + 1;
   END LOOP;
END;
/
```

Hinweis:
Jeder Datensatz jeder Tabelle innerhalb einer Oracle-Datenbank besitzt eine eindeutige Rowid. Hierüber findet das DBMS den physikalischen Datensatz. Sie können sich diese Information als eine zusätzliche Tabellenspalte vorstellen. Allerdings ist es nicht möglich (und auch nicht sinnvoll) dieses Feld direkt zu beschreiben. Jegliche Versuche dieser Art gehören hart bestraft. Die folgende Anweisung zeigt eine einfache Selektion auf eine solche Rowid:

```
SQL> SELECT rowid, deptno FROM dept;

ROWID                 DEPTNO
------------------ ---------
00000035.0000.0002        10
00000035.0001.0002        20
00000035.0002.0002        30
00000035.0003.0002        40

SQL>
```

Neben der Abfrage der Rowid können Sie diese außerdem in `WHERE`-Bedingungen verwenden. Der Vorteil hierbei ist, dass das DBMS direkt auf den physikalischen Satz in der Tabelle zugreifen kann. Unter Umständen kann die Verwendung von Rowids die Datenbank in erheblichem Maße beschleunigen. Bei Verwendung solcher interner Tabellenspalten kann man als Anwendungsentwickler nicht sicher sein, dass solche Anwendungen auch bei späteren Versionen der Datenbank immer noch lauffähig sind.

Hinweis:
Bei der Pseudospalte RowId handelt es sich um eine Oracle-interne Datenstruktur. Die Programmierung unter Zuhilfenahme dieses Feldes kann unter Umständen die Performance erheblich verbessern. Allerdings kann man als Anwendungsentwickler nie sicher sein, dass Oracle diese Struktur nicht in zukünftigen Versionen ändert, wie in der Version 8 eben geschehen. Um dieses Problem ein wenig zu entschärfen, liefert Oracle das Paket DBMS_ROWID mit aus. Hierin sind Prozeduren und Funktionen definiert, die auf der RowId operieren.

Neben dieser, vergleichsweise komplizierten Möglichkeit der automatischen Nummerierung, möchte ich Ihnen jetzt noch eine einfachere Version vorstellen. Der Trick dieses Zweizeilers basiert auf der Funktion von Sequenzen, die Sie im Kapitel 5 kennengelernt haben. Zunächst die Anweisungen selbst:

```
SQL> CREATE SEQUENCE dummy;

Sequence created.

SQL> UPDATE kunden
  2  SET    kundennr = dummy.nextval;

7 rows updated.
```

Im ersten Schritt wurde eine einfache Sequenz mit Standardparameter generiert, d.h. sowohl der Startwert als auch die Schrittweite ist '1'. Mit der `UPDATE`-Anweisung danach werden die Kundennummern neu nummeriert, wie die folgende Anweisung zeigt:

```
SQL> SELECT kundennr, name FROM kunden;

 KUNDENNR NAME
--------- ------------
        1 Meier
        2 Müller
        3 Behring
        4 Zimmermann
        5 Schröder
        6 Oppermann
        7 Testkandidat

7 rows selected.
```