Teil III

Prozedurale Konzepte in Oracle: PL/SQL

SQL bietet keinerlei prozedurale Konzepte wie z. B. Schleifen, Verzweigungen oder Variablendeklarationen. Entsprechend sind viele Aufgaben nur umständlich über Zwischentabellen oder evtl. überhaupt nicht in SQL zu realisieren. Anwendungsprogramme repräsentieren häufig zusätzliches anwendungsspezifisches Wissen, das nicht in der Datenbank enthalten ist.
Deshalb gibt es einige Erweiterungen und Umgebungen für SQL, die prozedurale Konzepte anbieten. Zum einen ist dies die Einbettung von SQL in prozedurale Hostsprachen (*embedded SQL*); z. B. Pascal, C, C++, oder neuerdings auch Java (JDBC), zum anderen eine spezielle Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, genannt *PL/SQL*. Eine Erweiterung hat hierbei den Vorteil, dass man diese prozeduralen Elemente enger an die Datenbank anbinden und z. B. in Prozeduren und Triggern nutzen kann.
In Oracle 7 bestand eine klare Trennung zwischen dem "eigentlichen", deklarativen SQL und der *prozeduralen Erweiterung* PL/SQL. Dennoch war PL/SQL bereits ein integrierter Teil des Oracle-Systems (im Gegensatz zu *embedded SQL* oder *JDBC*).
Mit der Erweiterung zu Objektorientierung und *Methoden* in Oracle 8 verschwimmt diese Trennung: bereits innerhalb des deklarativen Teils von SQL werden im Zuge von Methoden Konzepte von PL/SQL verwendet, um diese Methoden zu implementieren.

14 PROZEDURALE ERWEITERUNGEN: PL/SQL

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

14.1 PL/SQL-Blöcke

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

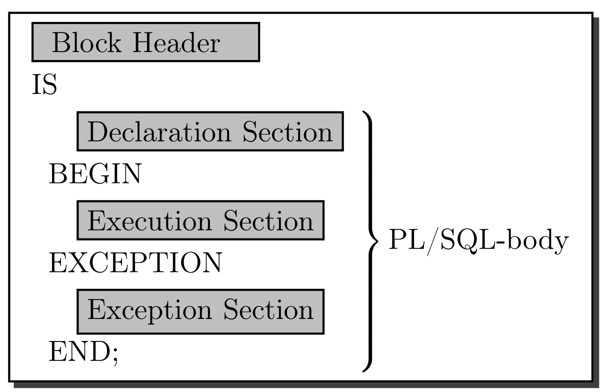

Abbildung 14.1: Struktur eines PL/SQL-Blocks

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

Prozeduren. Für Prozeduren sieht die Deklaration folgendermaßen aus:

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

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

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

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

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

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

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

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

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

Beispiel 12 (Funktion) Die folgende Funktion berechnet die Distanz zwischen zwei Paaren (*Länge,Breite*) von geographischen Koordinaten (Funktionsaufruf siehe Beispiel
Eine Funktion darf keine Seiteneffekte auf die Datenbasis haben (ansonsten erzeugt ORACLE eine Fehlermeldung).

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

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

```
  - Befehle des äußeren Blocks -
END;       /* äußerer Block */
```

Trigger werden in Abschnitt

Deklaration ausführen. Wenn man eine Deklaration ausführt, muss nach dem Semikolon noch ein Vorwärtsslash ("/") folgen, um die Deklaration zu verarbeiten!
Falls sich SQL*Plus mit einem "... created with compilation errors" zurückmeldet, kann man sich die Fehlermeldung mit

```
SHOW ERRORS;
```

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

Prozeduren und Funktionen aufrufen. Prozeduren werden innerhalb von PL/SQL-Blöcken durch

```
<procedure> (arg1,...,argn);
```

aufgerufen. In SQLPlus werden Prozeduren mit

```
execute <procedure> (arg1,...,argn);
```

aufgerufen. Funktionsaufrufe haben die übliche Syntax

```
... <function> (arg1,...,argn) ...
```

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

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

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

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

SELECT distance(7.8, 48, Longitude, Latitude)
FROM City
WHERE Name = 'Stockholm';
```

Zugriffsrechte. Anderen Benutzern kann man durch `GRANT EXECUTE ON <procedure/function> TO <user>` die Benutzung von Prozeduren und Funktionen erlauben. Dabei muss man berücksichtigen, dass Prozeduren und Funktionen jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt werden. Vergibt man also `GRANT EXECUTE ON <procedure/function> TO <user>`, so kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat. Andererseits hat man damit die

Möglichkeit, Zugriffsberechtigungen implizit strenger zu formulieren als mit `GRANT ... ON <table> TO ...`: Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

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

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

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

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

14.2 PL/SQL-Deklarationen

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

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

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

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

```
<variable> <variable'>%TYPE [NOT NULL] [DEFAULT <value>];
```

oder

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

Im ersten Fall bekommt `<variable>` den gleichen Datentyp wie die Variable `<variable'>` im zweiten Fall den Datentyp der Spalte `<col>` in der Tabelle

`<table>`. Der Typ einer anchored-Deklaration wird zur Compile-Time bestimmt, d.h. wenn sich der Datentyp der verwendeten Datenbank ändert, muss das PL/SQL-Programm neu übersetzt werden.

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

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

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

```
isCapital: BOOLEAN;
```

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

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

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

```
SELECT Name, rowid FROM City;
```

bekommt man diese Spalte auch angezeigt.

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

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

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

`RECORD`: Ein Record enthält mehrere Felder und entspricht damit einem Tupel in der Datenbasis:

```
TYPE city_type IS RECORD
  (Name City.Name%TYPE,
   Country VARCHAR2(4),
   Province VARCHAR2(32),
   Population NUMBER,
   Longitude NUMBER,
   Latitude NUMBER);

the_city city_type;
```

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

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

[1] `ROWID` gehört nicht zum SQL Standard

Man hätte also in dem obigen Beispiel auch einfach

```
the_city City%ROWTYPE;
```

deklarieren können.
Zur Zuweisung an Records gibt es verschiedene Möglichkeiten:

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

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

Feldzuweisung: Ein Feld eines Records wird einzeln zugewiesen:

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

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

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

Beim Vergleich von Records muss jedes Feld einzeln verglichen werden.

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

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

<var> <type>;
```

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

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

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

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

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

```
andere_table := plz_table;
```

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

```
<variable> := <pl/sql-table-name>.<built-in-function>;
```

oder

```
<pl/sql-table-name>.<built-in-procedure>;
```

aufgerufen. Die folgenden built-in-Funktionen bzw. Prozeduren sind implementiert (PL/SQL Version 2.3):

`COUNT` (fkt): Gibt die Anzahl der belegten Zeilen aus.

`EXISTS` (fkt): `TRUE` falls Tabelle nicht leer ist.

`DELETE` (proc): Löscht alle Zeilen einer Tabelle.

`FIRST/LAST` (fkt): Gibt den niedrigsten/höchsten Indexwert, für den der Eintrag in der Tabelle definiert ist.

`NEXT/PRIOR(n)` (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen Indexwert, für den der Eintrag in der Tabelle definiert ist.

Beispiel 14

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

□

14.3 SQL-Statements in PL/SQL

In der Execution Section eines PL/SQL-Blocks können an beliebigen Stellen DML-Kommandos, also `INSERT`, `UPDATE`, `DELETE` sowie `SELECT INTO`-Statements stehen. Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten. Zusätzlich zu der bekannten SQL-Syntax ist es möglich, bei `INSERT`, `UPDATE` und `DELETE`-Befehlen, die *nur ein einziges Tupel betreffen* mit Hilfe der `RETURNING`-Klausel Werte an PL/SQL-Variablen zurückzugeben.

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

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

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

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

14.4 Kontrollstrukturen

PL/SQL enthält die folgenden Kontrollstrukturen:

IF THEN - [ELSIF THEN] - [ELSE] - END IF,

verschiedene Schleifen:

Simple LOOP:

```
LOOP ... END LOOP;
```

WHILE LOOP:

```
   WHILE <bedingung> LOOP ... END LOOP;
```

Numeric FOR LOOP:

```
   FOR <loop_index> IN [REVERSE] <Anfang> .. <Ende>
   LOOP ... END LOOP;
```

Die Variable <loop_index> wird dabei *automatisch* als INTEGER deklariert.

Mit EXIT [WHEN <bedingung>]; kann man einen LOOP jederzeit verlassen.

den allseits beliebten GOTO-Befehl mit Labels:

```
   <<label_i>> ... GOTO label_j;
```

Zu bemerken ist, dass NULL-Werte (außer bei IS NULL-Abfrage) immer in den ELSE-Zweig verzweigen, da ein Nullwert keine Bedingung erfüllt.
Für GOTO gibt es einige Einschränkungen: Man kann nicht von außen in ein IF-Konstrukt, einen LOOP, oder einen lokalen Block hineinspringen, ebenfalls nicht von einem IF-Zweig in einen anderen; d.h. Labels sind lokal zu dem jeweiligen Block.
Da hinter einem Label immer mindestens ein ausführbares Statement stehen muss, und einem ein solches vielleicht nicht immer sinnvoll einfällt, gibt es das NULL Statement, das *nichts* tut.

Bildschirmausgaben in PL/SQL. Bildschirmausgaben (z.B. als Testausgaben) können unter Verwendung des DBMS_Output-Packages gemacht werden. Dazu muss man nur zuerst

```
SET SERVEROUTPUT ON
```

setzen (einmal, z. B. beim Aktivieren von sqlplus) und kann dann innerhalb des PL/SQL-Bodies den Befehl

```
dbms_output.put_line('bla');
```

verwenden. Diese Ausgaben erscheinen aber erst *nach* Ausführung der Prozedur/des Triggers am Bildschirm!

14.5 Cursorbasierter Datenbankzugriff.

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

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

(<parameter-list>) gibt auch hier eine Parameter-Liste an. Dabei ist als Zugriffsart nur IN zugelassen. Da IN sowieso der Default-Wert ist, kann man diese Angabe auch gleich weglassen. Zwischen SELECT und FROM dürfen damit nicht nur die in SQL zugelassenen Ausdrücke stehen, sondern zusätzlich PL/SQL-Variablen und PL/SQL-Funktionen. PL/SQL-Variablen können ebenfalls in den WHERE-, GROUP- und HAVING-Klauseln verwendet werden.
Einen *Cursor* hat man sich als eine (geordnete !) Tabelle mit einem “Fenster”, das über einem Tupel stehen kann und schrittweise vorwärts bewegt wird, vorzustellen.

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

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

Im Umgang mit einem Cursor stehen drei Operationen zur Verfügung:

```
OPEN <cursor-name>[(<argument-list>)];
```

öffnet einen Cursor. Dabei wird das entsprechende SELECT-Statement abgearbeitet und eine *virtuelle Tabelle* erstellt. Die angegebenen Argumente werden für die IN-Parameter des Cursors eingesetzt. Der Cursor wird *vor* die erste Zeile dieser Tabelle gesetzt. Damit gibt es in dieser Situation keine *aktuelle Zeile*.

```
FETCH <cursor-name> INTO <record>;
```

oder

```
FETCH <cursor-name> INTO <variable-list>;
```

bewegt den Cursor auf die nächste Zeile des Ergebnisses der Anfrage und kopiert diese in die angegebene Record-Variable oder Variablenliste.
Da ein Cursor eine *virtuelle Tabelle* ist, ist der entsprechende Datentyp durch die Tabellendeklarationen und das SELECT-Statement vorgegeben. Auch hier kann man mit der <cursor-name>%ROWTYPE-Deklaration eine Record-Variable mit dem entsprechenden Typ deklarieren:

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

```
CLOSE <cursor-name>;
```

schließt einen Cursor.

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

```
city_in cities_in%ROWTYPE;
```

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

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

```
  CLOSE cities_in;
END;
```

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

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

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

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

`<cursor-name>%ISOPEN`: gibt an, ob ein Cursor geöffnet ist.

`<cursor-name>%FOUND`: Solange ein Cursor bei der letzten `FETCH`-Operation ein neues Tupel gefunden hat, ist `<cursor-name>%FOUND = TRUE`.

`<cursor-name>%NOTFOUND`: hat man alle Zeilen eines Cursors ge`FETCH`t, nimmt dieses Attribut den Wert `TRUE` an.

`<cursor-name>%ROWCOUNT`: gibt zu jedem Zeitpunkt an, wieviele Records von einem Cursor bereits gelesen wurden.

Diese Cursorattribute dürfen nicht innerhalb eines SQL-Ausdrucks verwendet werden – sie werden für die Auswertung in Schleifenbedingungen eingesetzt.

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

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

Die Variable `<record_index>` wird dabei *automatisch* mit dem entsprechenden `%ROWTYPE` deklariert.
Bei jeder Ausführung des Schleifenkörpers wird dabei *automatisch* ein `FETCH` in die Schleifenvariable ausgeführt (daher wird der Schleifenkörper i.a. *keinen* `FETCH`-Befehl enthalten!) Außerdem spart man die Deklaration der Variablen eines `RECORD`-Typs sowie die `OPEN`- und `CLOSE`-Statements.

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

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

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

```
      request_Info(the_city);
    END LOOP;
 END;
```

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

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

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

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

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

Die folgenden Kommandos ändern/löschen jeweils das zuletzt von dem genannten Cursor ge`FETCH`te Tupel:

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

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

14.6 Nutzung Geschachtelter Tabellen unter PL/SQL

Geschachtelte Tabellen (vgl. Abschnitt

In Abschnitt

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

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

Beispiel 15 In Anschnitt

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

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

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

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

Die bisher beschriebenen Funktionen und Prozeduren werden durch den Benutzer explizit aufgerufen. Daneben gibt es noch *Trigger* (vgl. Abschnitt

14.7 Trigger

Trigger sind eine spezielle Form von PL/SQL-Prozeduren, die beim Eintreten eines bestimmten Ereignisses vom System automatisch ausgeführt werden. Damit sind sie ein Spezialfall aktiver Regeln nach dem **E**vent-**C**ondition-**A**ction-Paradigma.
In ORACLE 8 gibt es verschiedene Arten von Triggern: `BEFORE`- und `AFTER`-Trigger sind bereits aus ORACLE 7 bekannt, während `INSTEAD OF`-Trigger erst in Version 8 eingeführt wurden.

14.7.1 BEFORE- und AFTER-Trigger

Ein BEFORE- oder AFTER-Trigger ist einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet. Seine Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst. Die Ausführung eines Triggers kann zusätzlich von Bedingungen an den Datenbankzustand abhängig gemacht werden. Weiterhin unterscheidet man, ob ein Trigger *vor* oder *nach* der Ausführung der entsprechenden aktivierenden Anweisung in der Datenbank ausgeführt wird. Ein Trigger kann einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) seiner Tabelle ausgeführt werden.
Die Definition eines Triggers besteht aus einem Kopf, der die oben beschriebenen Eigenschaften enthält, sowie einem in PL/SQL geschriebenen Rumpf. Da Trigger im Zusammenhang mit Veränderungen an (Zeilen) der Datenbasis verwendet werden, gibt es die Möglichkeit im Rumpf den alten und neuen Wert des gerade behandelten Tupels zu verwenden.

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

Mit BEFORE und AFTER wird angegeben, ob der Trigger vor oder nach Ausführung der auslösenden Operation (einschließlich aller referentiellen Aktionen) ausgeführt wird.

Die Angabe von OF <column> ist nur für UPDATE erlaubt. Wird OF <column> nicht angegeben, so wird der Trigger aktiviert wenn irgendeine Spalte eines Tupels verändert wird.

Mittels der *Transitions-Variablen* OLD bzw. NEW kann auf die Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion zugegriffen werden. Mit REFERENCING OLD AS ... NEW AS ... kann man sie auch anders nennen. Schreibzugriff auf NEW-Werte ist nur mit BEFORE-Triggern erlaubt.

FOR EACH ROW definiert einen Trigger als Row-Trigger. Fehlt diese Zeile, wird der Trigger als Statement-Trigger definiert.

Mit WHEN kann die Ausführung eines Triggers weiter eingeschränkt werden. Hier können Transitionsvariablen OLD und NEW, die die Werte des betroffenen Tupels enthalten, verwendet werden. Subqueries an die Datenbank sind nicht erlaubt.

Der <pl/sql-block> enthält den auszuführenden Programmteil.

Referenzieren der Variablen im PL/SQL-Teil als :OLD und :NEW (dies sind “bind-variables”, die PL/SQL von außen zugeführt werden, daher der zusätzliche “:”).

Der `<pl/sql-block>` darf keine Befehle zur Transaktionskontrolle enthalten.

Ist ein Trigger für verschiedene Ereignisse definiert, kann das auslösende Ereignis im Rumpf durch `IF INSERTING THEN`, `IF UPDATING OF <column-list> THEN` usw. abgefragt werden.

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

```
CREATE OR REPLACE TRIGGER change_Code
BEFORE UPDATE OF Code ON Country
FOR EACH ROW
BEGIN
  UPDATE Province
  SET Country = :NEW.Code
  WHERE Country = :OLD.Code;
END;
/

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

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

```
CREATE TRIGGER new_Country
AFTER INSERT ON Country
FOR EACH ROW
WHEN (NEW.population > 2)
BEGIN
   INSERT INTO Politics (Country,Independence)
   VALUES (:NEW.Code, SYSDATE);
END;
/

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

SELECT * FROM Politics WHERE country='LU';
```

Mutating Tables. Zeilenorientierte Trigger werden immer direkt vor/nach der Veränderung einer Zeile aufgerufen. Damit “sieht” jede Ausführung eines solchen Triggers einen anderen Datenbestand der Tabelle, auf der er definiert

ist, sowie der Tabellen, die er evtl. ändert. Um hier zu garantieren, dass das Ergebnis *unabhängig von der Reihenfolge* der veränderten Tupel ist, werden die entsprechenden Tabellen während der gesamten Aktion als *mutating* gekennzeichnet (ebenso Tabellen, die durch ein `ON DELETE CASCADE` von den Änderungen an einer solchen Tabelle betroffen sein können. Tabellen, die als *mutating* markiert sind, können nicht von Triggern gelesen oder geschrieben werden. Analoge Einschränkungen gelten auch für Tabellen, die über Fremdschlüssel ohne `CASCADE` verbunden sind (*constraining*).
Dieses *mutating table syndrom* ist bei der Triggerprogrammierung ausgesprochen störend, und kann in einigen Fällen nur durch umständliche Tricks gelöst werden. Man kann folgende Fälle unterscheiden:

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

Beispiel 18 (Mutating Table Syndrom: Lösung mit Hilfstabelle) Bei der Umbenennung eines Flusses muss für alle Flüsse, die in den umbenannten Fluss fließen, der Name des Zielflusses ebenfalls geändert werden (Definition der Tabelle *River* siehe Seite
Als Lösung wird eine Hilfstabelle angelegt, die alle Umbenennungen protokolliert (und durch einen Zeilentrigger unterhalten wird). Wenn alle Flüsse umbenannt sind, wird mit einem Statement-`AFTER`-Trigger die Hilfstabelle ausgewertet, um die Zielflüsse anzupassen:

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

CREATE OR REPLACE TRIGGER upd_river_name -- Row-Trigger
BEFORE UPDATE OF Name ON River
FOR EACH ROW
BEGIN
  INSERT INTO aux_rename_river
  VALUES(:OLD.Name,:NEW.Name);
END;
/
CREATE OR REPLACE TRIGGER upd_river_names -- Statement-Trigger
```

```
AFTER UPDATE OF Name ON River
BEGIN
  UPDATE River
  SET River = (SELECT new
               FROM aux_rename_river
               WHERE old = River)
  WHERE River IN (SELECT old
               FROM aux_rename_river);
  DELETE FROM aux_rename_river;
END;
/
```

14.7.2 INSTEAD OF-Trigger

INSTEAD OF-Trigger dienen dazu, *View Updates* (vgl. Abschnitt

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

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

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

SELECT * from user_updatable_columns
WHERE table_name = 'ALLCOUNTRY';

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

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

```
CREATE OR REPLACE TRIGGER InsAllCountry
INSTEAD OF INSERT ON AllCountry
```

```
FOR EACH ROW
BEGIN
   INSERT INTO Country (Name,Code,Population,Area)
   VALUES (:NEW.Name, :NEW.Code, :NEW.Population, :NEW.Area);
   INSERT INTO Economy (Country,Inflation)
   VALUES (:NEW.Code, :NEW.Inflation);
   INSERT INTO Population (Country, Population_Growth,infant_mortality)
   VALUES (:NEW.Code, :NEW.Population_Growth, :NEW.infant_mortality);
END;
/
```

Zusammen mit dem Trigger *New_ Country*, der das Gründungsdatum automatisch einträgt, werden somit die Relationen *Country*, *Economy*, *Population* und *Politics* aktualisiert.
Für *Object Views* (vgl. Abschnitt

Zugriffsrechte. Bei der Verwendung von Triggern muss man berücksichtigen, dass Trigger genauso wie Prozeduren und Funktionen mit den Zugriffsrechten des *Besitzers* ausgeführt werden. Da man Zugriffsrechte an Triggern nicht explizit vergeben kann, werden sie *automatisch* von jedem benutzt, der entsprechende Veränderungen an der Trigger-Tabelle vornimmt.
Das folgende Beispiel ist sinngemäß aus [CHRS98] entnommen:

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

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

1 Zeile wurde erstellt.

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

Es wurden keine Zeilen ausgewaehlt
```

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

14.8 Ausnahmebehandlung von Fehlersituationen

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

```
DECLARE
<exception> EXCEPTION;
   ⋮
BEGIN
   ⋮
   IF ...
   THEN RAISE < exception>;
   ⋮
EXCEPTION
   WHEN <exception1>
   THEN <PL/SQL-Statement>;
   ⋮
   WHEN <exceptionn>
   THEN <PL/SQL-Statement>;
   [WHEN OTHERS THEN <PL/SQL-Statement>;] END;
```

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

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

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

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

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

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

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