

## TEIL III: Erweiterungen

Teil I: Grundlagen

Teil II: Diverses

Teil III: Prozedurale Konzepte, OO, Einbettung

- PL/SQL: Prozeduren, Funktionen, Trigger
- Objektorientierung
- SQL und Java
- SQL und XML

### SITUATION

- keine prozeduralen Konzepte in SQL (Schleifen, Verzweigungen, Variablendeklarationen)
- viele Aufgaben nur umständlich über Zwischentabellen oder überhaupt nicht in SQL zu realisieren.
  - Transitive Hülle.
- Programme repräsentieren anwendungsspezifisches Wissen, das nicht in der Datenbank enthalten ist.

### ERWEITERUNGEN

- Einbettung von SQL in prozedurale Wirtssprachen (*embedded SQL*); meistens Pascal, C, C++, oder auch Java (JDBC/SQLJ),
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL (Procedural language extensions to SQL)*.
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern.
- benötigt für Objektmethoden.

## Kapitel 8

# Prozedurale Erweiterungen: PL/SQL

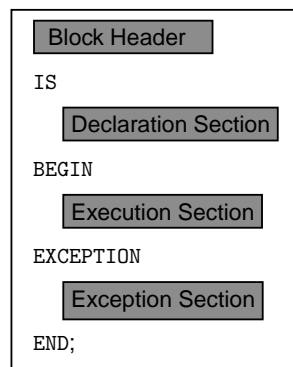
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL (Procedural language extensions to SQL)*.
- “Stored Procedures/Functions” innerhalb der DB
- direkter Zugriff auf Datenbankinhalt
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank; Nutzung in Prozeduren, Funktionen und Triggern

### Weitere Nutzung

- Programmierung von Objektmethoden (seit Oracle 8/1997)

## 8.1 Prozeduren, Funktionen und Kontrollstrukturen in PL/SQL

### Blockstruktur von PL/SQL



- Block Header: Art des Objekts (Funktion, Prozedur oder *anonym* (innerhalb eines anderen Blocks)), und Parameterdeklarationen.
- Declaration Section: Deklarationen der in dem Block verwendeten Variablen,
- Execution Section: Befehlssequenz des Blocks,
- Exception Section: Reaktionen auf eventuell auftretende Fehlermeldungen.

## EINFACHE, ANONYME BLÖCKE

- nur Declaration und Execution Section
- werden direkt ausgeführt
- DECLARE ... BEGIN ... END;  
/

Wichtig: nach dem Semikolon noch ein Vorwärtsslash ("/") in einer separaten Zeile, um die Deklaration auszuführen!!!

(Beispiel → nächste Folie)

## AUSGABE-GENERIERUNG

- verwendet das DBMS\_Output Package
- einmalig **SET SERVEROUTPUT ON**  
(z.B., beim Starten von sqlplus)
- innerhalb von PL/SQL-Blocks:  
`dbms_output.put_line('bla');`
- Bei Prozeduren etc.: Ausgabe erscheint erst *nach* kompletter Ausführung der Prozedur etc.

```
set serveroutput on;
DECLARE
bla NUMBER;
BEGIN
  bla := 42;
  dbms_output.put_line(bla);
END;
/
```

[Filename: PLSQL/output.sql]

## PROZEDUREN

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

- OR REPLACE: existierende Prozedurdefinition wird überschrieben.
- (*<parameter-list>*): Deklaration der formalen Parameter:
 

```
(<variable> [IN|OUT|IN OUT] <datatype>,  
  ⋮  
  <variable> [IN|OUT|IN OUT] <datatype>)
```
- IN, OUT, IN OUT: geben an, wie die Prozedur/Funktion auf den Parameter zugreifen kann (Lesen, Schreiben, beides).
- Default: IN.
- Bei OUT und IN OUT muss beim Aufruf eine Variable angegeben sein, bei IN ist auch eine Konstante erlaubt.
- *<datatype>*: alle von PL/SQL unterstützten Datentypen; *ohne* Längenangabe (VARCHAR2 anstelle VARCHAR2(20)).
- *<pl/sql-body>* enthält die Definition der Prozedur in PL/SQL.

## FUNKTIONEN

Analog, zusätzlich wird der Datentyp des Ergebnisses angegeben:

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

- *datatype* darf dabei nur ein atomarer SQL-Datentyp sein. Es können damit also keine Tabellen zurückgegeben werden.
- PL/SQL-Funktionen werden mit
 

```
RETURN <ausdruck>;
```

 verlassen. Jede Funktion muss mindestens ein RETURN-Statement im *<body>* enthalten.
- Eine Funktion darf keine Seiteneffekte auf die Datenbasis haben (siehe Oracle-Dokumentation *PL/SQL User's Guide and Reference*).

## PROZEDUREN UND FUNKTIONEN

- Im Falle von "... created with compilation errors":  
`SHOW ERRORS;`  
 ausgeben lassen.
- Prozeduren und Funktionen können mit `DROP PROCEDURE/FUNCTION <name>` gelöscht werden.
- Aufruf von Prozeduren im PL/SQL-Skript:  
`<procedure> (arg1,...,argn);`  
 (wenn ein formaler Parameter als OUT oder IN OUT angegeben ist, muss das Argument eine Variable sein)
- Aufruf von Prozeduren in SQLPlus:  
`execute <procedure> (arg1,...,argn);`
- Verwendung von Funktionen in PL/SQL:  
`... <function> (arg1,...,argn) ...`  
 wie in anderen Programmiersprachen.
- Die system-eigene Tabelle `DUAL` wird verwendet um das Ergebnis freier Funktionen in sqlplus ausgeben zu lassen:  
`SELECT <function> (arg1,...,argn)`  
`FROM DUAL;`

## BEISPIEL: PROZEDUR

- Einfache Prozedur: PL/SQL-Body enthält nur SQL-Befehle

Informationen über Länder sind über mehrere Relationen verteilt.

```
CREATE OR REPLACE 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;
/
```

[Filename: PLSQL/insertcountry.sql]

```
EXECUTE InsertCountry
('Lummerland', 'LU', 1, 4, 50, 0.5, 0.25);
```

## BEISPIEL: FUNKTION

- Einfache Funktion: Einwohnerdichte eines Landes

```
CREATE OR REPLACE FUNCTION Density (arg VARCHAR2)
RETURN number
IS
    temp number;
BEGIN
    SELECT Population/Area
    INTO temp
    FROM Country
    WHERE code = arg;
    RETURN temp;
END;
/
```

[Filename: PLSQL/density.sql]

```
SELECT Density('D')
FROM dual;
```

## PL/SQL-VARIABLEN UND DATENTYPEN.

Deklaration der PL/SQL-Variablen in der Declaration Section:

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

### Einfache Datentypen:

BOOLEAN: TRUE, FALSE, NULL,

BINARY\_INTEGER, PLS\_INTEGER: Ganzzahlen mit Vorzeichen.

NATURAL, INT, SMALLINT, REAL, ...: Numerische Datentypen.

```
DECLARE
anzahl NUMBER DEFAULT 0;
name VARCHAR2(30);
```

### anchored TYPDEKLARATION

Angabe einer PL/SQL-Variablen, oder Tabellenspalte (!) deren Typ man übernehmen will:

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

oder

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

- cityname City.Name%TYPE
- %TYPE wird zur Compile-Time bestimmt.

### ZUWEISUNG AN VARIABLEN

- "klassisch" innerhalb des Programms:

```
a := b;
```

- Zuweisung des (einspaltigen und einzeiligen!) Ergebnisses einer Datenbankabfrage an eine PL/SQL-Variablen:

```
SELECT ...
  INTO <PL/SQL-Variablen>
FROM ...
```

```
DECLARE
cname country.name%TYPE;
BEGIN
  SELECT name
  INTO cname
  FROM country
  WHERE code='D';
  dbms_output.put_line(cname);
END;
/
```

[Filename: PLSQL/simple.sql]

## PL/SQL-DATENTYPEN: RECORDS

Ein RECORD enthält mehrere Felder, entspricht 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;
```

### anchored Typdeklaration für Records

Records mit Tabellenzeilen-Typ deklarieren: %ROWTYPE:

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

Äquivalent zu oben:

```
the_city city%ROWTYPE;
```

### Zuweisung an Records

- Aggregierte Zuweisung: zwei Variablen desselben Record-Typs:  

```
<variable> := <variable'>;
```
- Feldzuweisung: ein Feld wird einzeln zugewiesen:  

```
<record.feld> := <variable>|<value>;
```
- SELECT INTO: Ergebnis einer Anfrage, die *nur ein einziges Tupel* liefert:

```
SELECT ...
  INTO <record-variable>
FROM ... ;
```

```
DECLARE
c continent%ROWTYPE;
BEGIN
  SELECT *
  INTO c
  FROM continent
  WHERE name='Europe';
  dbms_output.put_line(c.name || ' : ' || c.area);
END;
/
```

[Filename: PLSQL/simple2.sql]

### Vergleich von Records

Beim Vergleich von Records muss jedes Feld einzeln verglichen werden.

## PL/SQL-DATENTYPEN: PL/SQL TABLES

Array-artige Struktur, *eine* Spalte mit beliebigem Datentyp (also auch RECORD), normalerweise mit BINARY\_INTEGER indiziert.

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

<var> <type>;

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

plz_table plz_table_type;
```

- Adressierung: <var>(1)
 

```
plz_table(79110) := Freiburg;
plz_table(33334) := Kassel;
```
- *sparse*: nur die Zeilen gespeichert, die Werte enthalten.

Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

### PL/SQL-Datentypen: PL/SQL Tables (Forts.)

Zusätzlich *built-in*-Funktionen und -Prozeduren:

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

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

- COUNT (fkt): Anzahl der belegten Zeilen.
 

```
plz_table.count = 2
```
- EXISTS(i) (fkt): TRUE falls Zeile i der Tabelle nicht leer.
- DELETE (proc): Löscht alle Zeilen einer Tabelle.
- DELETE(i): Löscht Zeile i einer Tabelle.
- FIRST/LAST (fkt): niedrigster/höchster belegter Indexwert.
 

```
plz_table.first = 33334
```
- NEXT/PRIOR(n) (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen belegten Indexwert.
 

```
plz_table.next(33334) = 79110
```

## SQL-STATEMENTS IN PL/SQL

- DML-Kommandos INSERT, UPDATE, DELETE sowie **SELECT INTO**-Statements.
- Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten.
- Befehle, die *nur ein einziges Tupel betreffen*, können mit RETURNING Werte an PL/SQL-Variablen zurückgeben:  

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

Z.B. Row-ID des betroffenen Tupels zurückgeben:

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

## 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.
- EXIT [WHEN <bedingung>]: LOOP verlassen.
- den allseits beliebten GOTO-Befehl mit Labels:  

```
<<label_i>> ... GOTO label_j;
```
- NULL-Werte verzweigen immer in den ELSE-Zweig.
- GOTO: nicht von außen in ein IF-Konstrukt, einen LOOP, oder einen lokalen Block hineinspringen, nicht von einem IF-Zweig in einen anderen springen.
- hinter einem Label muss immer mindestens ein ausführbares Statement stehen;
- NULL Statement.

## GESCHACHELTE BLÖCKE

Innerhalb der *Execution Section* werden *anonyme Blöcke* zur Strukturierung verwendet. Hier wird die *Declaration Section* mit DECLARE eingeleitet (es gibt keinen Block Header):

```
BEGIN
  -- Befehle des äußeren Blocks --
  DECLARE
    -- Deklarationen des inneren Blocks
  BEGIN
    -- Befehle des inneren Blocks
  END;
  -- Befehle des äußeren Blocks --
END;
```

## 8.2 Cursore/Iteratoren zur Verarbeitung von Ergebnismengen

- Datenbankanfragen: mengenorientiert
- Programmiersprache: variablenbasiert

### Design Patterns: Kollektionen und Iteratoren

(vgl. Informatik I)

- Kollektion: Sammlung von Items (Liste, Baum, Heap, Menge)
- Iterator: Hilfsklasse zum Durchlaufen/Aufzählen aller Items
- Methoden:
  - Erzeugen/Initialisieren des Iterators,
  - Weiterschalten, Test, ob noch weitere Elemente vorhanden sind,
  - Zugriff auf ein Element,
  - (Schliessen des Iterators)

... Iteratoren werden im Weiteren immer wieder verwendet.

**CURSORBASIERTER DATENBANKZUGRIFF**

Zeilenweiser Zugriff auf eine Relation aus einem PL/SQL-Programm.

Cursordeklaration in der *Declaration Section*:

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

- (<parameter-list>): Parameter-Liste,
- nur IN als Übergaberichtung erlaubt.
- Zwischen SELECT und FROM auch PL/SQL-Variablen und PL/SQL-Funktionen. PL/SQL-Variablen können ebenfalls in den WHERE-, GROUP- und HAVING-Klauseln verwendet werden.

**Beispiel:**

Alle Städte in dem in der Variablen the\_country angegebenen Land:

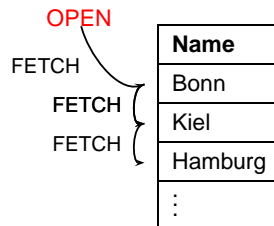
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
FROM City
WHERE Country=the_country;
```

**Cursore: Grundprinzip**

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

Erzeugt mit dem gegebenen SELECT-Statement eine *virtuelle Tabelle* mit einem "Fenster", das über einem Tupel stehen kann und schrittweise vorwärts bewegt wird. Mit OPEN wird die Anfrage ausgeführt und der Cursor initialisiert:

```
OPEN cities_in ('D');
```



### Cursore: Verwendung

- `FETCH <cursor-name> INTO <record-variable>;` 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.  
Diese kann z.B. mit `<cursor-name>%ROWTYPE` mit dem Record-Typ des Cursors definiert werden:  
`<variable> <cursor-name>%ROWTYPE;`
- `CLOSE <cursor-name>;` schließt einen Cursor.

```
DECLARE CURSOR cities_in
(crs_country Country.Code%TYPE)
IS SELECT Name
FROM City
WHERE Country = crs_country;
city_in cities_in%ROWTYPE;
BEGIN
OPEN cities_in ('D');
FETCH cities_in INTO city_in;
dbms_output.put_line(city_in.Name);
FETCH cities_in INTO city_in;
dbms_output.put_line(city_in.Name);
CLOSE cities_in;
END;
/
```

[Filename: PLSQL/cursor1.sql]

### Cursore: Attribute

Kontrolle über die Verarbeitung eines Cursors:

- `<cursor-name>%ISOPEN`: Cursor offen?
- `<cursor-name>%FOUND`: Solange ein Cursor bei der letzten `FETCH`-Operation ein neues Tupel gefunden hat, ist `<cursor-name>%FOUND = TRUE`.
- `<cursor-name>%NOTFOUND`: `TRUE` wenn man alle Zeilen eines Cursors `GEFETCHT` hat.
- `<cursor-name>%ROWCOUNT`: Anzahl der von einem Cursor bereits gelesenen Tupel.
- nicht innerhalb eines SQL-Ausdrucks.

Cursore: Attribute

```
CREATE OR REPLACE PROCEDURE first_city
    (the_country country.code%TYPE)
IS BEGIN
    DECLARE CURSOR cities_in
    (crs_country Country.Code%TYPE)
    IS SELECT Name
    FROM City
    WHERE Country = crs_country;
    city_in cities_in%ROWTYPE;
    BEGIN
    OPEN cities_in (the_country);
    FETCH cities_in INTO city_in;
    IF cities_in%FOUND
    THEN DBMS_OUTPUT.PUT_LINE(city_in.name);
    ELSE DBMS_OUTPUT.PUT_LINE('Nothing found!');
    END IF;
    CLOSE cities_in;
    END;
END;
/
```

[Filename: PLSQL/cursor-attribs.sql]

```
execute first_city('D');
execute first_city('X');
```

- Aufgabe: Programmieren Sie eine explizite WHILE-Schleife, die alle Städte eines Landes ausgibt.

Cursore: Hinweis

nicht möglich:

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

- ein parametrisierter Cursor,
- nicht eine Familie von Cursorsen!

## CURSOR FOR LOOP

Spezielle Schleife zur Iteration über den Inhalt eines Cursors:

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

- <record\_index> wird dabei *automatisch* als Variable vom Typ <cursor-name>%ROWTYPE deklariert,
- <record\_index> *immer* von einem Record-Type – ggf. einspaltig.
- Es wird automatisch ein OPEN ausgeführt,
- bei jeder Ausführung des Schleifenkörpers wird *automatisch* ein FETCH ausgeführt,
- → Schleifenkörper enthält i.a. *keinen* FETCH-Befehl,
- am Ende wird automatisch ein CLOSE ausgeführt,
- Spalten müssen explizit adressiert werden.

### Cursor FOR LOOP: Beispiel

Beispiel: Für jede Stadt in dem gegebenen Land soll der Name ausgegeben werden:

```
CREATE OR REPLACE PROCEDURE list_cities
  (the_country country.code%TYPE)
IS
BEGIN
  DECLARE CURSOR cities_in
    (crs_country country.Code%TYPE)
  IS SELECT Name
     FROM City
     WHERE Country = crs_country;
  BEGIN
    FOR the_city IN cities_in(the_country)
    LOOP
      dbms_output.put_line(the_city.name);
    END LOOP;
  END;
END;
/
```

[Filename: PLSQL/cursor-loop1.sql]

```
execute list_cities('D');
```

## EINGEBETTETER CURSOR FOR LOOP

- SELECT-Anfrage kann auch direkt in die FOR-Klausel geschrieben werden.

```
CREATE OR REPLACE PROCEDURE list_big_cities
(the_country country.code%TYPE)
IS
BEGIN
  FOR the_city IN
  ( SELECT Name
    FROM City
    WHERE Country = the_country
      AND Population > 1000000 )
  LOOP
    dbms_output.put_line(the_city.Name);
  END LOOP;
END;
/
```

[Filename: PLSQL/cursor-loop2.sql]

```
execute list_big_cities('D');
```

## SCHREIBZUGRIFF VIA CURSOR

Mit WHERE CURRENT OF <cursor-name> kann man auf das zuletzt von dem genannten Cursor gefETCHte Tupel zugreifen:

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

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

- Dabei bestimmt die Positionierung des Cursors bezüglich der Basistabellen den Ort der Änderung (im Gegensatz zu View Updates).

**DDL-KOMMANDOS IN PL/SQL**

DDL-Statements werden in PL/SQL nicht direkt unterstützt:

- EXECUTE IMMEDIATE <string>  
<string> kann dabei eine Konstante sein, oder kann dynamisch zusammengesetzt werden
- ```

BEGIN
  execute immediate 'drop table continent';
END;
/

CREATE OR REPLACE PROCEDURE clean
IS
BEGIN
  FOR tn IN
    (SELECT Table_name FROM all_tables
     WHERE table_name LIKE 'TMP_%')
  LOOP
    execute immediate 'DROP TABLE ' || tn.table_name;
  END LOOP;
END;
/

```
- Entsprechende Privilegien muss man direkt (GRANT ... TO <user>), und nicht nur über eine Rolle bekommen haben.

**DYNAMIC SQL MIT EXECUTE IMMEDIATE**

- Werte für Variablen einsetzen: USING :n  
(ebenso bei Aufruf von Prozeduren/Funktionen)

```

DECLARE country VARCHAR2(4) := 'CDN';
        org VARCHAR2(10) := 'EU';
BEGIN
  execute immediate
    'insert into is_member VALUES (:1, :2, :3)'
    using country, org, 'candidate';
END;
/

```

- Wert in eine PL/SQL-Variable einlesen: INTO

```

CREATE OR REPLACE PROCEDURE sp (name City.name%TYPE)
IS BEGIN declare cty city%ROWTYPE;
BEGIN
  execute immediate 'select * from city where name= :1'
    into cty
    using name;
  dbms_output.put_line(cty.population);
END;
END;
/
execute sp('Berlin');

```

### 8.3 Zugriffsrechte auf PL/SQL-Datenbankobjekte

Benutzung von Funktionen/Prozeduren:

- Benutzungsrechte vergeben:  
`GRANT EXECUTE ON <procedure/function> TO <user>;`
- Prozeduren und Funktionen werden jeweils mit den Zugriffsrechten des *Besitzers* ausgeführt.
- nach  
`GRANT EXECUTE ON <procedure/function> TO <user>;`  
 kann dieser User die Prozedur/Funktion auch dann aufrufen, wenn er kein Zugriffsrecht auf die dabei benutzten Tabellen hat.
- Möglichkeit, Zugriffsberechtigungen strenger zu formulieren als mit `GRANT ... ON <table> TO ...`:  
 Zugriff nur in einem ganz speziellen, durch die Prozedur oder Funktion gegebenen Kontext.

### 8.4 Geschachtelte Tabellen unter PL/SQL

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

Nutzung geschachtelter Tabellen in ORACLE nicht ganz unproblematisch:

*“Bestimme alle Länder, in denen Deutsch gesprochen wird, sowie den Anteil der deutschen Sprache in dem Land”*

Eine solche Anfrage muss für *jedes* Tupel in *Nested\_Languages* die innere Tabelle untersuchen.

- `SELECT THE` kann jeweils nur ein Objekt zurückgeben,
- keine Korrelation mit umgebenden Tupeln möglich.
- Verwendung einer (Cursor-)Schleife.

Geschachtelte Tabellen unter PL/SQL: Beispiel

```
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 Nested_Language
WHERE Country = the_country.Code)
WHERE Name = the_Language;
END LOOP;
END;
/
EXECUTE Search_Countries('German');
SELECT * FROM tempCountries;
```

**(RE)AKTIVES VERHALTEN**

- Bis jetzt: Funktionen und Prozeduren werden durch den Benutzer explizit aufgerufen.
- Trigger: Ausführung wird durch das Eintreten eines Ereignisses in der Datenbank angestoßen.

## 8.5 Trigger

### EINSCHUB: INTEGRITÄTSBEDINGUNGEN

- Spalten- und Tabellenbedingungen
- Wertebereichsbedingungen (*domain constraints*),
- Verbot von Nullwerten,
- Uniqueness und Primärschlüssel-Bedingungen,
- CHECK-Bedingungen.

! Alles nur als Bedingungen an *eine* Zeile innerhalb *einer* Tabelle formulierbar.

### ASSERTIONS

- Bedingungen, die den gesamten DB-Zustand betreffen.  
`CREATE ASSERTION <name> CHECK (<bedingung>)`
- Diese werden allerdings von ORACLE bisher nicht unterstützt.

⇒ Also muss man sich etwas anderes überlegen.

### TRIGGER

- spezielle Form von PL/SQL-Prozeduren,
- werden beim Eintreten eines bestimmten Ereignisses ausgeführt.
- Spezialfall aktiver Regeln nach dem **Event-Condition-Action**-Paradigma.
- einer Tabelle (oft auch noch einer bestimmten Spalte) zugeordnet.
- Bearbeitung wird durch das Eintreten eines Ereignisses (Einfügen, Ändern oder Löschen von Zeilen der Tabelle) ausgelöst (**Event**).
- Ausführung von Bedingungen an den Datenbankzustand abhängig (**Condition**).
- **Action**:
  - *vor* oder *nach* der Ausführung der entsprechenden aktivierenden Anweisung ausgeführt.
  - einmal pro auslösender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) ausgeführt.
  - Trigger-Aktion kann auf den alten und neuen Wert des gerade behandelten Tupels zugreifen.

**TRIGGER**

```
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 (<condition>)]
  <pl/sql-block>;
```

- BEFORE, AFTER: Trigger wird vor/nach der auslösenden Operation ausgeführt.
- OF <column> (nur für UPDATE) schränkt Aktivierung auf angegebene Spalte ein.
- Zugriff auf Zeileninhalte vor und nach der Ausführung der aktivierenden Aktion mittels OLD bzw. NEW. Schreiben in NEW-Werte nur mit BEFORE-Trigger.
- FOR EACH ROW: Row-Trigger, sonst Statement-Trigger.
- WHEN (<condition>): zusätzliche Bedingung; hier werden OLD und NEW verwendet; Subqueries an die Datenbank sind nicht erlaubt.
- Referenzieren der Variablen im PL/SQL-Teil als :OLD und :NEW.

**TRIGGER: BEISPIEL**

Wenn ein Landes-Code geändert wird, pflanzt sich diese Änderung auf die Relation *Province* fort:

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

[Filename: PLSQL/changeCode.sql]

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

```
SELECT * FROM Province WHERE Country='UK';
```

### TRIGGER: BEISPIEL

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;
/
```

[Filename: PLSQL/newcountry.sql]

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

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

### TRIGGER: MUTATING TABLES

- Zeilenorientierte Trigger: immer direkt vor/nach der Veränderung einer Zeile aufgerufen
- jede Ausführung des Triggers sieht einen anderen Datenbestand der Tabelle, auf der er definiert ist, sowie der Tabellen, die er evtl. ändert
- ~> Ergebnis *abhängig von der Reihenfolge* der veränderten Tupel

ORACLE: Betroffene Tabellen werden während der gesamten Aktion als *mutating* gekennzeichnet, können nicht von Triggern gelesen oder geschrieben werden.

Nachteil: Oft ein zu strenges Kriterium.

- Trigger soll auf Tabelle zugreifen auf der er selber definiert ist.
  - Nur das auslösende Tupel soll von dem Trigger gelesen/geschrieben werden: Verwendung eines BEFORE-Triggers und der :NEW- und :OLD-Variablen
  - Es sollen neben dem auslösenden Tupel auch weitere Tupel verwendet werden: Verwendung eines Statement-orientierten Triggers
- Trigger soll auf andere Tabellen zugreifen: Verwendung von Statement-Triggern und ggf. Hilfstabellen.

## INSTEAD OF-TRIGGER

- *View Updates*: Updates müssen auf Basistabellen umgesetzt werden.
- View-Update-Mechanismen eingeschränkt.
- INSTEAD OF-Trigger: Änderung an einem View wird durch andere SQL-Anweisungen ersetzt.

```
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>;
```

- Keine Einschränkung auf bestimmte Spalten möglich
- Keine WHEN-Klausel
- Default: FOR EACH ROW

## VIEW UPDATES UND INSTEAD OF-TRIGGER

```
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;
```

[Filename: PLSQL/allcountry-view.sql]

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

[Filename: PLSQL/insert-allcountry.sql]

Fehlermeldung: Über ein Join-View kann nur *eine* Basistabelle modifiziert werden.

## VIEW UPDATES UND INSTEAD OF-TRIGGER

```
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;
/
```

[Filename: PLSQL/instead-of.sql]

- aktualisiert *Country*, *Economy* und *Population*.
- Trigger *New\_Country* (AFTER INSERT ON COUNTRY) aktualisiert zusätzlich *Politics*.

## FEHLERBEHANDLUNG

- Declaration Section: Deklaration (der Namen) benutzerdefinierter Exceptions.  

```
DECLARE <exception> EXCEPTION;
```
- Exception Section: Definition der beim Auftreten einer Exception auszuführenden Aktionen.  

```
WHEN <exception>
    THEN <PL/SQL-Statement>;
WHEN OTHERS THEN <PL/SQL-Statement>;
```
- Exceptions können dann an beliebigen Stellen des PL/SQL-Blocks durch RAISE ausgelöst werden.  

```
IF <condition>
    THEN RAISE <exception>;
```

## ABLAUF

- auslösen einer Exception
- entsprechende Aktion der WHEN-Klausel ausführen
- innersten Block verlassen (oft Anwendung von anonymen Blöcken sinnvoll)

**TRIGGER/FEHLERBEHANDLUNG: BEISPIEL**

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;
/
```

[Filename: PLSQL/trigger-nachmittag.sql]

**BEISPIEL**

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

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

1 Zeile wurde erstellt.

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

Es wurden keine Zeilen ausgewaehlt.

(aus A. Christiansen, M. Höding, C. Rautenstrauch und G. Saake, ORACLE 8 effizient einsetzen, Addison-Wesley, 1998)

## 8.6 Weitere PL/SQL-Features

- *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.
  
- Definition komplexer Transaktionen,
- Verwendung von SAVEPOINTS für Transaktionen.