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

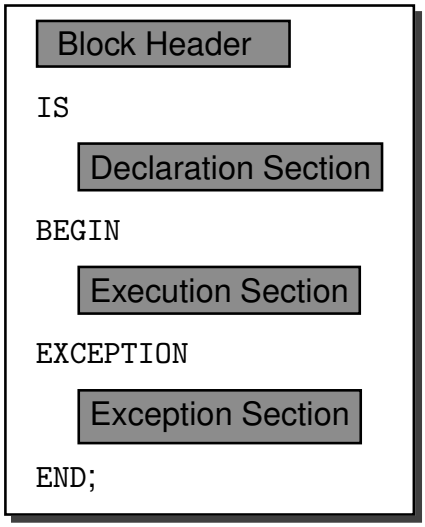
- Erweiterung von SQL um prozedurale Elemente *innerhalb* der SQL-Umgebung, *PL/SQL* (*Procedural language extensions to SQL*).
- “Stored Procedures/Functions” innerhalb der DB
- direkter Zugriff auf Datenbankinhalt
- Vorteile von PL/SQL: Bessere Integration der prozeduralen Elemente in die Datenbank;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

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(50);
```

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.

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(50),
   Population NUMBER,
   Latitude NUMBER,
   Longitude 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 VARIABLEN

- "klassisch" innerhalb des Programms:

```
a := b;
```

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

```
SELECT ...
INTO <PL/SQL-Variable>
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]

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]

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.

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 werden mit `DROP PROCEDURE/FUNCTION <name>` gelöscht.
- 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;
```

Prozedur mit IN- und OUT-Parametern

- eine Funktion kann nur einen *einzelnen* Wert zurückgeben, und darf den Datenbankzustand nicht verändern.

```
CREATE OR REPLACE PROCEDURE ChangePop (cc VARCHAR2, pop IN OUT NUMBER, dens OUT NUMBER)
IS
  temp number;
  BEGIN
  SELECT population INTO temp FROM country WHERE code = cc;
  UPDATE country SET population = pop WHERE code = cc;
  SELECT population/area INTO dens FROM country WHERE code = cc;
  pop := temp;
END;
/
DECLARE pop NUMBER; dens NUMBER;
BEGIN
  pop := 80000000;
  ChangePop('D', pop, dens);
  dbms_output.put_line(pop);     -- 82521653 the before pop
  dbms_output.put_line(dens);    -- 226.94   new density
END;
/
```

[Filename: PLSQL/procinout.sql]

SQL-Statements in PL/SQL

- DML-Kommandos `INSERT`, `UPDATE`, `DELETE` sowie `SELECT INTO`-Statements.
- Diese SQL-Anweisungen dürfen auch PL/SQL-Variablen enthalten.
- DDL-Statements sind in PL/QL nicht (direkt) erlaubt! (siehe Folie 225)
- 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 berüchtigten `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.

GESCHACHTELTE 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:

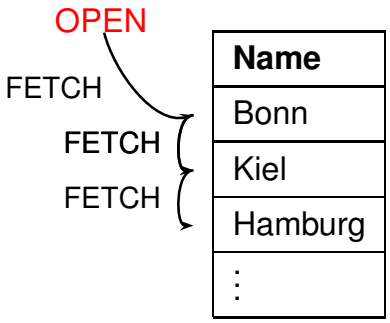
```
DECLARE
CURSOR cities_in (the_country Country.Code%TYPE)
IS SELECT Name
   FROM City
   WHERE Country=the_country;
```

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 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 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 ge`FETCH`t 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-attrs.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 Cursoren!

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).

PL/SQL-Datentypen: PL/SQL Tables

Array-artige Struktur, *eine* Spalte mit beliebigem Datentyp (also auch `RECORD`), normalerweise mit `BINARY_INTEGER` indiziert.

Typ-Definition: `TYPE <tabtype> IS TABLE OF <datatype> [INDEX BY BINARY_INTEGER];`
Var-Deklaration: `<tabname> <tabtype>;`

Built-in-Funktionen und -Prozeduren:

```
<variable> := <tabname>.<built-in-function>;
```
oder
```
<tabname>.<built-in-procedure>;
```

- `COUNT` (fkt): Anzahl der belegten Zeilen.
- `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.
  (ist null falls Tabelle leer ist!)
- `NEXT/PRIOR(n)` (fkt): Gibt ausgehend von n den nächsthöheren/nächstniedrigen belegten Indexwert.

PL/SQL Tables als einfache Collections

- dann (implizit) indiziert mit 1..n (falls nicht leer)

```
SELECT ...
BULK COLLECT INTO <tabname>
FROM ...
WHERE ...
```

- analog
  `TABLE OF <table>.<attr>%TYPE`
  und dann Zugriff nur mit `tab(i)`.
- bei BULK COLLECT wird der vorherige Inhalt der Tabelle überschrieben.

```
DECLARE
 TYPE tabtype IS TABLE OF city%ROWTYPE;
 tab tabtype;
BEGIN
 SELECT *
  BULK COLLECT INTO tab
  FROM city
  WHERE country = 'D';
 IF tab.COUNT > 0 THEN
   FOR i IN tab.FIRST .. tab.LAST LOOP
     dbms_output.put_line(tab(i).name);
   END LOOP;
 END IF;
END;
/
```

[Filename: PLSQL/table1.sql]

PL/SQL Tables als indizierte Collections

```
TYPE <tabtype> IS TABLE OF <datatype>
  INDEX BY BINARY_INTEGER;
<tabname> <tabtype>;
```

- Adressierung: `<tabname>`(n)
- *sparse*: nur die Zeilen gespeichert, die Werte enthalten.
- Dann springen mit `WHILE` und `<tabname>.next`:

```
DECLARE
 TYPE plz_table_type IS TABLE OF City.Name%TYPE
      INDEX BY BINARY_INTEGER;
 plztab plz_table_type;
 i NUMBER;
BEGIN
  plztab(37077):= 'Goettingen';
  plztab(79110):= 'Freiburg';
  plztab(33334):= 'Kassel';
  i := plztab.first;                -- 33334
 WHILE NOT i IS NULL LOOP
  dbms_output.put_line(i || ' ' || plztab(i));
  i := plztab.next(i);
 END LOOP;
END;
/
```

[Filename: PLSQL/table2.sql]

PL/SQL Tables

- Tabellen können auch als Ganzes zugewiesen werden

```
andere_table := plz_table;
```

- Unterschied BULK COLLECT zu Cursor:
  - Cursor wird on-demand iteratorbasiert ausgewertet, kann abgebrochen werden,
  - BULK COLLECT wertet komplett aus und legt das Ergebnis in PL/SQL-Tabelle ab.
    - BULK COLLECT ist daher ineffizienter, gibt aber die DB sofort wieder frei (falls eine andere Transaktion schreibend zugreifen möchte).

PL/SQL Tables als Rückgabewert einer Funktion

```
CREATE OR REPLACE TYPE membership_type AS OBJECT (
       country VARCHAR2(4),
       type VARCHAR2(60) );
/
CREATE OR REPLACE TYPE memberships_type AS TABLE OF membership_type;
/
CREATE OR REPLACE FUNCTION members_of(org VARCHAR2)
RETURN memberships_type
IS
  toreturn memberships_type;
BEGIN
  SELECT membership_type(country, type)
    BULK COLLECT INTO toreturn
    FROM ismember
    WHERE organization = org;
  RETURN toreturn;
END;
/
SELECT country, type
FROM TABLE(SELECT members_of('EU') FROM DUAL);
```

[Filename: PLSQL/return_table.sql]

8.3 Dynamic SQL

- `EXECUTE IMMEDIATE <string>`
- ‹string› kann dabei eine Konstante sein,
  - auf diese Weise kann man auch DDL-Statements in PL/SQL aufrufen:

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

- oder kann dynamisch zusammengesetzt werden (siehe nächste Folie)
- und Platzhalter für *Werte*, die dann als *Konstanten* des entsprechenden Datentyps bei Ausführung eingesetzt werden, enthalten.
- Escapen der single-quotes in dem auszuführenden String: `'... where name=''foo'''`.

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


Dynamic SQL: Kommandos als String erstellen und ausführen

- Hiermit kann man auch Tabellennamen, Spaltennamen und beliebige Dinge in das DDL-Statement oder auch in eine Anfrage einfügen.

```
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;
/
```

[Filename: PLSQL/clean.sql]

Dynamic SQL: Anfragen mit einzeiligem Ergebnis

- Wert in eine PL/SQL-Variable einlesen: INTO ‹pl/sql-variable›
  (bei einzeiligem Ergebnis)

```
CREATE OR REPLACE PROCEDURE getcountrypop (ccode country.code%TYPE)
IS
  c country%ROWTYPE;
  BEGIN
  execute immediate 'select * from country where code= :1'
    into c
    using ccode;
  dbms_output.put_line(c.population);
  END;
/
execute getcountrypop('CH');
```

[Filename: PLSQL/dynamicselect.sql]

Dynamic SQL: Anfragen mit mehrzeiligen Ergebnissen

- mit einem PL/SQL-Cursor (dann geht es ohne “execute immediate”):

```
CREATE OR REPLACE PROCEDURE getcitiesCursor (ccode country.code%TYPE)
IS
  TYPE CurRefType  IS REF CURSOR;
  cv CurRefType;
  c city%ROWTYPE;
  BEGIN
    OPEN cv FOR
      'select * from city where country = :1'
      using ccode;
    LOOP
      FETCH cv into c;
      dbms_output.put_line(c.name);
      EXIT WHEN cv%NOTFOUND;
    END LOOP;
  END;
/
execute getcitiesCursor('AUS');
```

[Filename: PLSQL/dynamicselectCursor.sql]

Dynamic SQL: Anfragen mit mehrzeiligen Ergebnissen

- BULK COLLECT INTO ‹pl/sql table›

```
CREATE OR REPLACE PROCEDURE getcitiesBulk (ccode country.code%TYPE)
IS
  TYPE ctynametabtype IS TABLE OF City.Name%TYPE
       INDEX BY BINARY_INTEGER;
  cities ctynametabtype;
  i NUMBER;
  BEGIN
    execute immediate 'select name from city where country = :1'
      bulk collect into cities
      using ccode;
    i := cities.first;
    WHILE NOT i IS NULL LOOP
      dbms_output.put_line(i || ' ' || cities(i));
      i := cities.next(i);
    END LOOP;
  END;
/
execute getcitiesBulk('AUS');
```

[Filename: PLSQL/dynamicselectBulk.sql]

8.4 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.
- Entsprechende Privilegien muss man direkt (GRANT ... TO ‹user›), und nicht nur über eine Rolle bekommen haben.

8.5 Geschachtelte Tabellen unter PL/SQL

Nested_Spoken		
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_Spoken* 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_Spoken
               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.

8.6 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 **E**vent-**C**ondition-**A**ction-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>]]*
  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
- $\leadsto$ 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 durch Exceptions in PL/SQL

- 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 SYSDATE
     BETWEEN to_date('12:00', 'HH24:MI')
         AND to_date('18:00', 'HH24:MI')
  THEN RAISE_APPLICATION_ERROR
         (-20101,'Unerlaubte Aktion');
  END IF;
END;
/
```

[Filename: PLSQL/trigger-nachmittag.sql]

Beispiel

```
CREATE OR REPLACE TRIGGER dummytrigger
INSTEAD OF INSERT ON AllCountry
FOR EACH ROW
BEGIN
 -- username must be in capitals!!!
 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)

8.7 Zeitgesteuerte Jobs

(eigentlich nicht direkt zu PL/SQL gehörend)

- Implementierung zeitabhängiger Trigger,
- bei jedem Produkt anders.
- Beschreibung hier für Oracle (Stand 12c, 2014)

Jobs und Schedules in Oracle

- Man benötigt die Systemprivilegien CREATE JOB, MANAGE SCHEDULER, ggf. auch CREATE EXTERNAL JOB
- die Konfiguration benutzt eine objektorientierte Syntax (ist auch auf Basis der objektrelationalen Interna umgesetzt)
- interne (PL/SQL) und externe (Aufruf eines Programmes via Pfad) Jobs möglich
- komplexe Spezifikationen von Aufrufzeiten via Schedules möglich.

Einmalige zeitgesteuerte Jobs: Beispiel

- der folgende Job trägt nach einer Minute die (dann) aktuelle Zeit in die Tabelle “jobtest” ein, danach wird der Job gelöscht.
- Syntax in SQLplus:

  `execute DBMS_SCHEDULER.CREATE_JOB (<job-spezifikation>);`

  ... dabei werden aber keine Zeilenumbrüche akzeptiert.
- also dasselbe (didaktisch) besser in einen PL/SQL-begin/end-Block packen:

```
create table jobtest (x DATE);
begin
  DBMS_SCHEDULER.CREATE_JOB
   (job_name => 'job1',
    job_type => 'PLSQL_BLOCK',
    job_action => 'begin insert into jobtest
                        values (SYSDATE); end;',
    start_date => SYSDATE+1/1440,
    enabled => TRUE);
end;
/
```

[Filename: PLSQL/simple-job.sql]

Job-Attribute und Aufruf via Scheduler

- enabled: TRUE aktiviert sofort, FALSE hält einen Job deaktiviert

  ```
  execute DBMS_SCHEDULER.ENABLE('job1');
  execute DBMS_SCHEDULER.DISABLE('job1');
  ```

  manuell aufrufen bzw löschen:

  ```
  execute DBMS_SCHEDULER.RUN_JOB('job1');
  execute DBMS_SCHEDULER.DROP_JOB('job1');
  ```

  Fehlermeldungen erhält man nur bei manuellem Aufruf!
- start_date, end_date: ggf. Anfang und Ende.
- auto_drop: default TRUE; FALSE sorgt für Wiederholung.
- repeat_interval: basierend auf “Schedules”; Details siehe Dokumentation, z.B.
  - FREQ = YEARLY|MONTHLY|...|SECONDLY; Basisangabe, wie oft,
  - INTERVAL = 1..99 jedes, jedes zweite, jedes 99. von FREQ,
  - BYMONTH =...; BYDAY=...; Spezifikation des “wann” innerhalb des “wie oft”,

  `repeat_interval => 'FREQ = WEEKLY; INTERVAL = 2;` jede zweite Woche
  `BYDAY = MON, THU; BYHOUR = 15, 16; BYMINUTE = 00'` Montags+Donnerstags, 15+16 Uhr
- job_type: ’PL/SQL_BLOCK’, ’STORED_PROCEDURE’, ’EXECUTABLE’ (externer Job)

Wiederholende zeitgesteuerte Jobs: Beispiel

- der folgende Job erhöht alle zwei Minuten zu jeweils 5 angegebenen Sekundenzeitpunkten die Bevölkerung um 1:

```
begin
 DBMS_SCHEDULER.DROP_JOB('job2');
 DBMS_SCHEDULER.CREATE_JOB
  (job_name => 'job2',
   job_type => 'PLSQL_BLOCK',
   job_action => 'begin
                  update country set population = population + 1 where code=''CN'';
                  end;',
   auto_drop => FALSE,
   repeat_interval => 'FREQ = MINUTELY; INTERVAL = 2;
     BYSECOND = 5, 18, 31, 45, 51',
   start_date => SYSDATE+1/2880,  -- after 30 secs
   end_date => SYSDATE+11/1440, -- after 11 minutes
   enabled => TRUE);
end;
/
```

[Filename: PLSQL/repeating-job.sql]

- ... in der 11. Minute nur noch 2x. Zusammen also 5x5 (0,2,4,6,8 min) + 1x2 = 27x.

Externe Jobs

- siehe Dokumentation.
- benötigt das Recht CREATE EXTERNAL JOB,
- werden auf dem Rechner ausgeführt, auf dem Oracle läuft,
- Oracle führt einen Benutzerwechsel durch (konfigurierbar, default: nobody/nogroup),
- job_action enthält nur Pfad des auszuführenden Programms, Parameter müssen separat übergeben werden,
- Fehlermeldungen erhält man nur bei manuellem Aufruf mit

```
execute DBMS_SCHEDULER.RUN_JOB('job1');
```

Informationen über bestehende Jobs

```
SELECT job_name, job_action
FROM user_scheduler_jobs;
```

8.8 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 `SAVEPOINT`s für Transaktionen.

8.9 Scripting: SQLplus und PL/SQL

- SQLPlus ist Oracle's *Client* für SQL,
  - Interface für Queries,
  - Ausführen von Skripten aus dem Filesystem des *Benutzers*. (z.B. create.sql)
- PL/SQL ist die *server-interne* Skriptsprache.
  - *innerhalb SQL aufrufbare Prozeduren/Funktionen/Trigger*
  - anonyme BEGIN ... END-Blöcke
- Man kann beide (ein bisschen) kombinieren. PL/SQL wird dann client-seitig ausgeführt.

PL/SQL in SQLPlus-Skripten

- PL/SQL `begin ... end;`-Blöcke können in SQLPlus-Skripten enthalten sein, und werden auf dem *Client* ausgewertet.
  Um ausgeführt zu werden, müssen sie mit "/" abgeschlossen werden.

Aufruf von Skripten (im *lokalen Filesystem*) aus PL/SQL in SQLPlus-Skripten

- Mit `@` (wie in SQLplus; wobei `start` hier nicht erlaubt ist!),
- Das aufgerufene Skript muss dann PL/SQL-konform sein
  - einige SQL-Kommandos, die in SQLplus direkt zulässig sind, jedoch nicht in PL/SQL, müssen in
    `execute immediate '...';`
    eingeschlossen sein.

Beispiel: Das create.sql-Skript

```
spool create.log
-- note: the file is located in the directory where the
-- *files* are located,
-- i.e. /afs/informatik.uni-goettingen.de/group/dbis/public/Mondial/create.log
start mondial-drop-tables;
start mondial-schema;
start mondial-inputs;
set serveroutput on;
 -- set read-all for DBIS user
BEGIN
 IF user = 'DBIS'  -- capitalized!
   THEN
    dbms_output.put_line('dbis: set grants');
    @mondial-grants.plsql;
 ELSE dbms_output.put_line('not dbis, no grants');
 END IF;
END ;
/
spool off
prompt Logfile 'create.log' created ...
```

[Filename: SQLPlus/create.sql]

- `$ORACLE_PATH` muss das Directory enthalten, in dem die o.g. Dateien liegen
- mondial-grants.plsql ist ein PL/SQL-Fragment

Beispiel (cont'd)

- mondial-grants.sql ist ein SQLplus-Skript, das einfach nur das PL/SQL-Skript aufruft:

```
BEGIN
  @mondial-grants.plsql;
END ;
/
```

[Filename: SQLPlus/mondial-grants.sql]

- mondial-grants.plsql ist ein PL/SQL-Skript, das von anderen Skripten (innerhalb eines PL/SQL-begin-end-Blocks) aufgerufen werden kann.
  (es enthält *kein* "/" am Ende!)
  (in einem reinen SQLPlus-Skript könnte "GRANT ..." alleine stehen)

[Filename: SQLPlus/mondial-grants.plsql]

```
execute immediate 'GRANT EXECUTE ON geocoord TO student';

execute immediate 'GRANT SELECT ON Country TO student';
execute immediate 'GRANT SELECT ON City TO student';
execute immediate 'GRANT SELECT ON Province TO student';
execute immediate 'GRANT SELECT ON Economy TO student';
execute immediate 'GRANT SELECT ON Population TO student';
execute immediate 'GRANT SELECT ON Politics TO student';
```