PL/SQL - Programmierung von Programmeinheiten

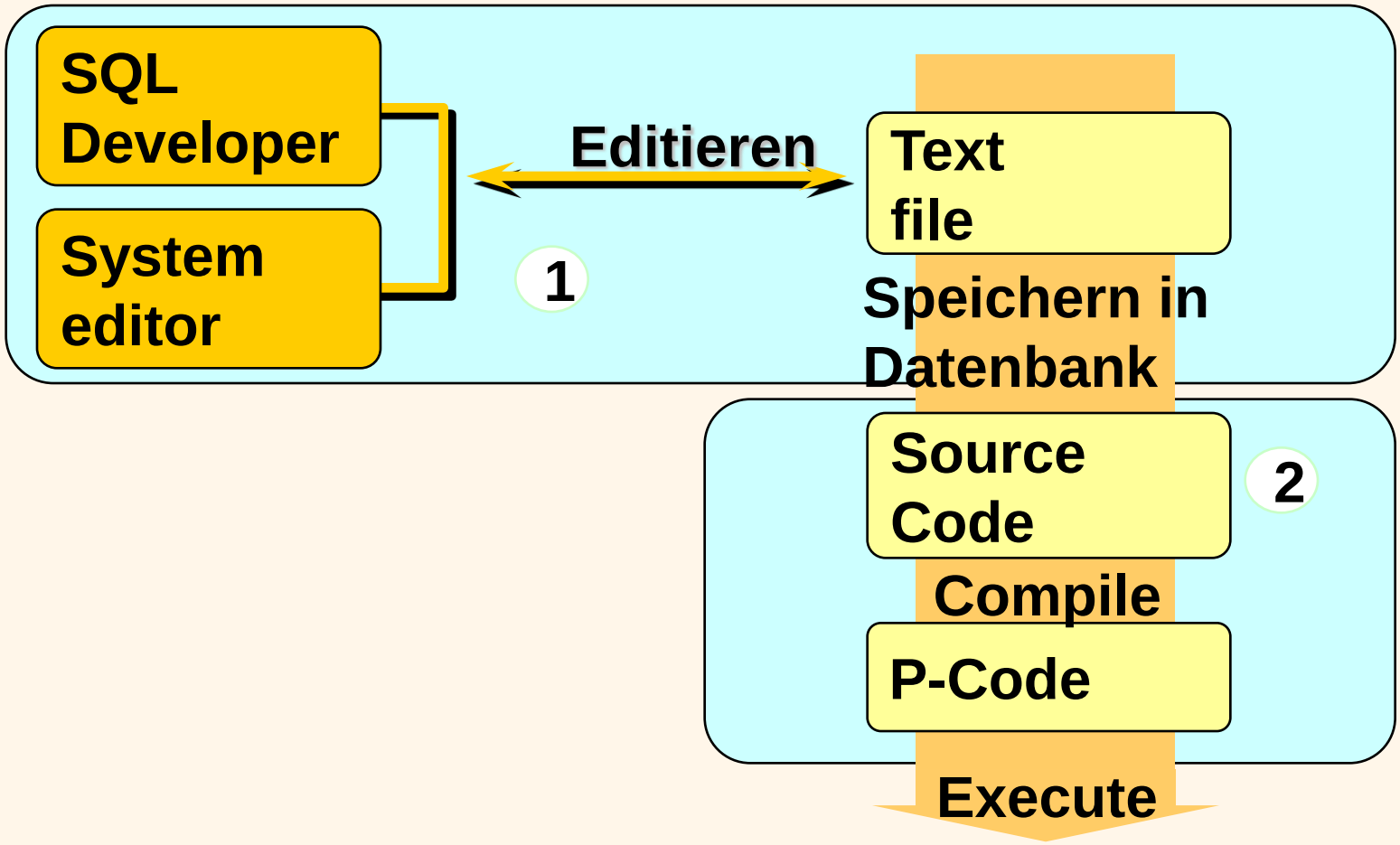
Gespeicherte Prozeduren

- Eine Prozedur ist ein benannter PL/SQL Block, der eine **Aktion** ausführt.
- **Parameter** zum Übermitteln der Daten von der Aufrufumgebung zur Prozedur.
- Prozeduren können von jedem Tool oder Sprache **aufgerufen** werden, die PL/SQL unterstützen.
- Prozeduren können als **Bausteine** für eine Applikation dienen.
- Eine Prozedur kann in der Datenbank gespeichert sein als **Datenbankobjekt** zur wiederholten Ausführung.
- Eine Prozedur kann als **eigenständige Transaktion** laufen.
- Eine Prozedur wird wie ein **PL/SQL-Befehl** aufgerufen.

Syntax
zum Erzeugen einer Prozedur

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  (argument1 [mode1] datatype1,
   argument2 [mode2] datatype2,
   . . .
IS [AS]
PL/SQL Block;
```

Entwicklung von Prozeduren/Funktionen



Parameter-Modi

IN	OUT	IN OUT
Default	Muss bekannt sein	Muss bekannt sein
Wert wird zum Subprogram übermittelt	Rückgabe an Aufrufumgebung	Wert zum Subpro-gramm übermittelt; Rückgabe an Aufrufumgebung
Formaler Parameter wie Konstante	Uninitialisierte Variable	Initialisierte Variable
Aktueller Parameter: Literal, Ausdruck, Kon-stante, init.Variable	Muss Variable sein	Muss Variable sein

IN-Parameter: Beispiel


```
SQL> CREATE OR REPLACE PROCEDURE raise_salary
  2   (v_id in emp.empno%TYPE)
  3   IS
  4   BEGIN
  5       UPDATE emp
  6       SET    sal = sal * 1.10
  7       WHERE  empno = v_id;
  8   END raise_salary;
  9   /
Procedure created.

SQL> EXECUTE raise_salary (7369)
PL/SQL procedure successfully completed.
```

OUT-Parameter: Beispiel

```
SQL> CREATE OR REPLACE PROCEDURE query_emp
  1   (v_id      IN   emp.empno%TYPE,
  2    v_name    OUT  emp.ename%TYPE,
  3    v_salary OUT   emp.sal%TYPE,
  4    v_comm    OUT  emp.comm%TYPE)
  5  IS
  6  BEGIN
  7    SELECT   ename, sal, comm
  8    INTO     v_name, v_salary, v_comm
  9    FROM     emp
 10    WHERE    empno = v_id;
 11  END query_emp;
 12  /
```

OUT-Parameter und SQL*Plus

```
SQL> START emp_query.sql
Procedure created.
```

```
SQL> VARIABLE g_namevarchar2(15)
SQL> VARIABLE g_salary     number
SQL> VARIABLE g_commnumber
```

```
SQL> EXECUTE query_emp (7654, :g_name, :g_salary,
  2   :g_comm)
PL/SQL procedure successfully completed.
```

```
SQL> PRINT g_name
G_NAME
---------------
MARTIN
```

Parameter Passing

- Position
- Namen
- Kombination aus beiden

Beispiel-Prozedur

```
SQL> CREATE OR REPLACE PROCEDURE add_dept
  1   (v_name  IN dept.dname%TYPE  DEFAULT 'unknown',
  2    v_loc   IN dept.loc%TYPE    DEFAULT 'unknown')
  3  IS
  4  BEGIN
  5     INSERT INTO dept
  6     VALUES (dept_deptno.NEXTVAL, v_name, v_loc);
  7  END add_dept;
  8  /
```

Parameter Passing (Beispiele)

```
SQL> begin
  2  add_dept;
  3  add_dept ( 'TRAINING', 'NEW YORK');
  4  add_dept ( v_loc => 'DALLAS', v_name =>
                'EDUCATION') ;
  5  add_dept ( v_loc => 'BOSTON') ;
  6  end;
  7  /
PL/SQL procedure successfully completed.
```

```
SQL>     SELECT * FROM dept;

DEPTNO   DNAME              LOC
------   --------------     -------------
   ...   ...                ...
    41   unknown            unknown
    42   TRAINING           NEW YORK
    43   EDUCATION          DALLAS
    44   unknown            BOSTON
```

Löschen von Prozeduren

Syntax

```
DROP PROCEDURE procedure_name
```

Beispiel

```
SQL> DROP PROCEDURE raise_salary;
Procedure dropped.
```

Interaktives Löschen auch mit Hilfe von Tools,
z.B. SQL Developer

Gespeicherte Funktionen

- Eine Funktion ist ein benannter PL/SQL Block, der einen **Wert** zurückliefert.
- Eine Funktion kann in der Datenbank als **Datenbankobjekt** für wiederholte Ausführung gespeichert werden.
- Eine Funktion kann als Teil eines **Ausdrucks** aufgerufen werden.

Syntax zum Erzeugen einer Funktion

```
CREATE [OR REPLACE] FUNCTION function_name
  (argument1 [mode1] datatype1,
   argument2 [mode2] datatype2,
   . . .
RETURN datatype
IS|AS
PL/SQL Block;
```

Erzeugen neuer Funktionen (Beispiel)

SQL*Plus

```
SQL> CREATE OR REPLACE FUNCTION get_sal
  2    (v_id  IN   emp.empno%TYPE)
  3  RETURN NUMBER
  4  IS
  5    v_salary     emp.sal%TYPE :=0;
  6  BEGIN
  7    SELECT sal
  8    INTO   v_salary
  9    FROM   emp
 10    WHERE  empno = v_id;
 11    RETURN (v_salary);
 12 END get_sal;
 13 /
```

Ausführen von Funktionen

- Aufrufen einer Funktion als Teil eines PL/SQL Ausdrucks
- Erzeugen einer Host-Variable, um den Rückgabewert aufzunehmen.
- Ausführen der Funktion. Host-Variable wird belegt mit dem RETURN Wert.

```
SQL> VARIABLE g_salary number
```

```
SQL> EXECUTE :g_salary := get_sal(7934)
PL/SQL procedure successfully completed.
```

Wo können benutzerdefinierter Funktionen aufgerufen werden?

- Select-Liste in einem SELECT Befehl
- Bedingung in einer WHERE oder HAVING Klausel
- CONNECT BY, START WITH, ORDER BY und GROUP BY Klausel
- VALUES Klausel eines INSERT Befehls
- SET Klausel eines UPDATE Befehls

Restriktionen für benutzerdefinierte Funktionen

- Muss als gespeicherte Funktion vorliegen.
- Nur IN Parameter erlaubt.
- Erlaubte Datentypen: CHAR, DATE oder NUMBER - keine PL/SQL Typen wie BOOLEAN, RECORD, oder TABLE.
- Rückgabe-Typ muss ein interner Typ des Oracle Server sein.
- INSERT, UPDATE oder DELETE verboten.
- Keine Aufrufe von Subprogrammen, die diese Restriktion verletzen

Löschen von Funktionen

Syntax

```
DROP FUNCTION function_name
```

Beispiel

```
SQL> DROP FUNCTION get_salary;
Function dropped.
```

Interaktives Löschen auch mit Hilfe von Tools,
z.B. SQL Developer

Vergleich von Prozeduren und Funktionen

Prozedur	Funktion
Ausführen als PL/SQL Befehl	Aufruf als Teil eines Ausdrucks
Kein RETURN Datentyp	RETURN Datentyp
Kann einen oder mehrere Werte zurückgeben	Muss einen Wert zurückliefern
Schachtelung: Aufruf einer Prozedur innerhalb einer Prozedur	Schachtelung: Argument der Funktion kann selbst das Ergebnis einer Funktion sein

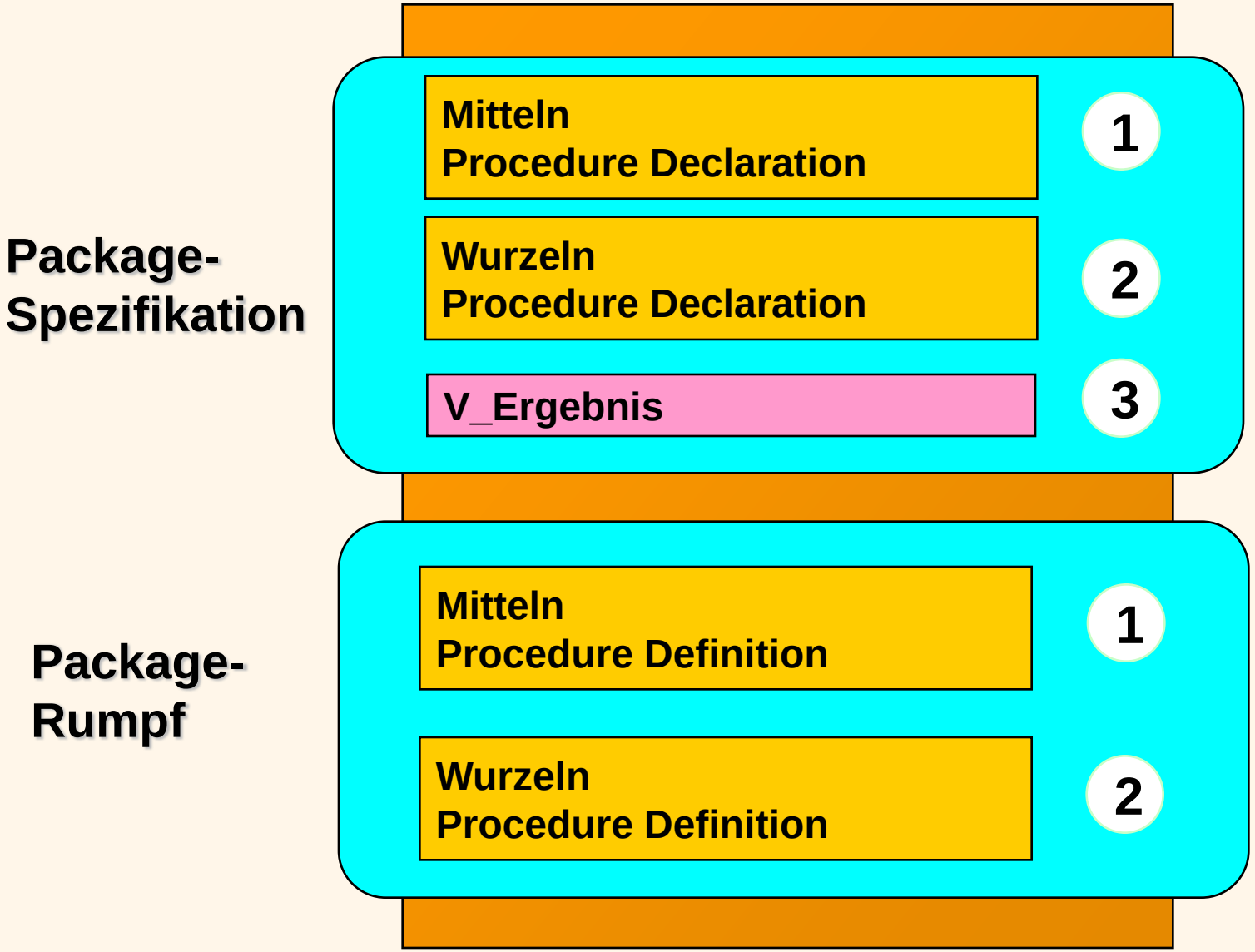
Packages: Überblick

- Gruppierung logisch zusammenhängender PL/SQL-Typen, Konstrukte (Cursor, Exeptions, Variable) und Subprogramme
- Besteht aus zwei Teilen:
  - Spezifikation
  - Rumpf
- Kann nicht aufgerufen, parametrisiert oder geschachtelt werden
- Erlaubt Oracle, gleichzeitig mehrere Objekte in den Speicher einzulesen
- Vordefinierte Packages für Standardaufgaben, insbesondere für I/O, zum Beispiel:
  - DBMS_OUTPUT
  - HTTP / HTF
  - UTL_FILE

Vorteile von Packages

- Information Hiding und Kapselung
  - Objekte eines Packages benutzbar, ohne interne Implementierung zu kennen
- Top-Down Design
  - Kopf kann vor dem Rumpf implementiert werden, d.h. Nutzung der deklarierten Objekte schon möglich
- Objektpersistenz
  - Objekte, die im Kopf deklariert werden, bleiben für die Dauer der Sitzung erhalten
- Bessere Performance
  - Package wird als ganzes in den Hauptspeicher geladen, spart weitere I/Os
- Overloading
  - Mehrere Prozeduren und Funktionen mit gleichem Namen im Package, jeweils mit unterschiedlichen Parametern (Anzahl, Typ)

Beispiel-Package



Beispiel Package-Spezifikation

```
SQL> CREATE OR REPLACE PACKAGE Mathe_Pckg AS
  2    PROCEDURE Mitteln (x IN NUMBER, y IN NUMBER)
  3    PROCEDURE Wurzeln (x IN NUMBER, y IN NUMBER)
  4    V_Ergebnis NUMBER;
  5  END Mathe_Pckg;
  6  /
```

Beispiel Package-Rumpf

```
SQL> CREATE OR REPLACE PACKAGE BODY Mathe_Pckg AS
  2    PROCEDURE Mitteln (x IN NUMBER, y IN NUMBER)IS
  3    BEGIN
  4      V_Ergebnis := (x+y)/2;
  5      DBMS_OUTPUT.PUT_LINE('Ergebnis = '|| V_Ergebnis);
  7    END Mitteln;
  8    PROCEDURE Wurzeln (x  IN NUMBER, y IN NUMBER)IS
  9    BEGIN
 10      V_Ergebnis := SQRT (x*y);
 11      DBMS_OUTPUT.PUT_LINE ('Ergebnis = '|| V_Ergebnis);
 12    END Wurzeln;
 13  BEGIN
 14    V_Ergebnis := 0;
 15  END Mathe_Pckg;
 16  /
```

Aufruf von Package-Konstrukten

Beispiel1: Aufruf einer Package-Prozedur aus SQL*Plus

```
SQL>   EXECUTE Mathe_Pckg.Mitteln(7,9);
```

Beispiel 2: Zugriff auf Variablen eines Package

```
SQL> IF Mathe_Pckg.V_Ergebnis = 10 THEN ... END IF;
```

Beispiel3: Aufruf einer Package-Prozedur in einem anderen Schema

```
SQL> EXECUTE Benutzer.Mathe_Pckg.Wurzeln(2,8);
```

Löschen von Packages

Löschen der Package-Spezifikation und des Rumpfes

```
DROP PACKAGE package_name
```

Löschen nur des Package-Rumpfes

```
DROP PACKAGE BODY package_name
```

Erzeugen von Triggern

- Trigger-Zeitpunkt: BEFORE oder AFTER
- Auslösendes Event: INSERT oder UPDATE oder DELETE (DML-Event)
- Tabellen-Name: ON Tabelle
- Trigger-Type: Zeilen- oder befehlsorientiert
- WHEN-Klausel: Einschränkende Bedingung
- Trigger-Rumpf: DECLARE
  BEGIN
  END;

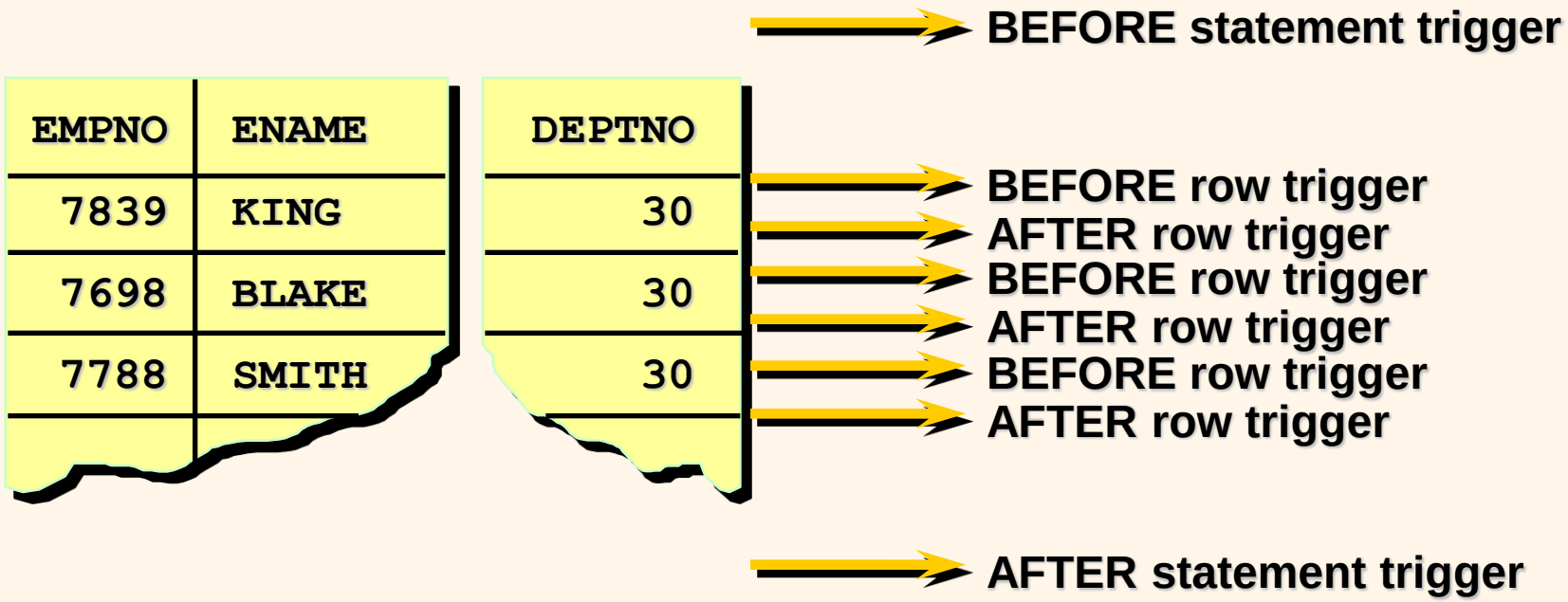
Ergänzung zu Events:

- weitere auslösende Ereignisse: DDL-Events (Schema-Änderungen), Database Events (z.B. Server Error)
- INSTEAD OF: auslösendes Ereignis (auf einer View) wird nicht ausgeführt – stattdessen Aktionen des Trigger-Rumpfes – Behandlung des View-Update-Problems

Besonderheiten und Einschränkungen

- maximale **Größe**: 32 K
- Erlaubte **Befehle** im Trigger-Rumpf:
  - DML-Befehle
  - SELECT, nur als SELECT INTO oder als Bestandteil eines Cursors
  - keine DDL-Befehle erlaubt
  - Keine Transaktionsbefehle (COMMIT, ROLLBACK) erlaubt
- Restriktionen auf **Mutating Tables**
  - Mutating Table = Tabelle in Veränderung - durch DML-Befehl (oder CASCADE-Option)
  - Trigger erhält keinen Zugriff auf mutating Tables (inkonsistente Daten nicht sichtbar)
  - bei versuchtem Zugriff Laufzeitfehler
- **Abhängigkeiten** zu anderen DB-Objekten beachten
  - aufgerufene Prozeduren / Funktionen
- **Keine Modifikation** möglich, nur DROP und CREATE
- **ALTER TRIGGER**
  - zum Ein- und Ausschalten (ENABLE / DISABLE)
  - zum Neuübersetzen (RECOMPILE)

Auslösen von Triggern



Befehlsorientierter Trigger (Syntax)

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
PL/SQL block;
```

Befehlsorientierter BEFORE-Trigger Beispiel

```
SQL> CREATE OR REPLACE TRIGGER secure_emp
  2  BEFORE INSERT ON emp
  3  BEGIN
  4   IF (TO_CHAR (sysdate,'DY') IN ('SAT','SUN'))
  5     OR (TO_CHAR(sysdate,'HH24')NOT BETWEEN
  6    '08' AND '18'
  7     THEN RAISE_APPLICATION_ERROR (-20500,
  8    'You may only insert into EMP during normal
  9     hours.');
 10   END IF;
 11  END;
 12  /
```

Zeilenorientierter Trigger (Syntax)

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
[WHEN condition]
PL/SQL block;
```

Zeilenorientierter Trigger (Beispiel)

Verwendung von old und new-Präfixen beachten!

```
SQL>CREATE OR REPLACE TRIGGER audit_emp_values
  2 AFTER DELETE OR INSERT OR UPDATE ON emp
  3 FOR EACH ROW
  4 BEGIN
  5   INSERT INTO audit_emp_values (user_name,
  6    timestamp, id, old_last_name, new_last_name,
  7    old_title, new_title, old_salary, new_salary)
  8   VALUES (USER, SYSDATE, :old.empno, :old.ename,
  9    :new.ename, :old.job, :new.job,
 10    :old.sal, :new.sal);
 11 END;
 12 /
```

Ausführung von Triggern

1. Ausführung aller befehlsorientierten BEFORE Trigger
2. Für jede betroffene Zeile
   a. Ausführung aller zeilenorientierten BEFORE Trigger
   b. Ausführung des DML-Befehls und der Integritätsprüfungen
   c. Ausführung aller zeilenorientierten AFTER Trigger
3. Ausführung der verzögerten Integritätsprüfungen
4. Ausführung aller befehlsorientierten AFTER Trigger

Anwendungen für Trigger

Anwendungsgebiete:

- Security (Zugriffskontrolle)
- Auditing (Protokollierung von Zugriffen)
- Datenintegrität (komplexe Bedingungen, s. Vorlesung zum Thema)
- Replikation von Tabellen
- Abgeleitete Daten
- Aufzeichnen von Ereignissen (benutzerdefinierte Logs)

Entwurfsempfehlungen:

- Garantiere die Verknüpfung zweier Operationen auf der Datenbank
- Keine Definition von Triggern für Features, die die DB sowieso beherrscht, z.B. deklarative Integritätsbedingungen
- Begrenze die Größe des Triggers, max. 60 Programmzeilen
- Nutze Trigger für zentrale Aufgaben, unabhängig vom Client
- Vermeide rekursive Trigger