Datenbank-Programmierung

Kapitel 8: Stored Procedures und Trigger

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Sommersemester 2018/19

http://www.informatik.uni-halle.de/~brass/dbp19/

Lernziele

Nach diesem Kapitel sollten Sie Folgendes können:

- You should know some advantages of using stored procedures.
- You should have at least some basic impression of the Oracle PL/SQL syntax.
- You should be able to explain what triggers are.
- You should know some applications of triggers.
- You should have some basic impression of the trigger syntax in Oracle.

Inhalt

Introduction (1)

- PL/SQL is an imperative programming language with variables and the usual control structures.
- PL/SQL has an especially good interface to SQL, e.g. the same type system as the Oracle database.
- Procedures written in PL/SQL can be stored and executed on the database server.
- PL/SQL is not meant for creating user interfaces.

  However, it is used in other Oracle user interface products. All input/output must be done with parameters or database tables. E.g., if a web page is created, it is first stored in the DB. It is a "server-side" programming language.

Introduction (2)

Advantages:

- Instead of sending many SQL commands to the server over the network, you send only a single call to a procedure stored on the server.

  So the network load is reduced as the data intensive computation is done on the DB server and only the final result is sent to the client where it is displayed.

- SQL statements in PL/SQL procedures are stored in preprocessed form on the server.

- PL/SQL procedures with their dependencies are recorded in the data dictionary.

Introduction (3)

Natural Development:

- DBMS were introduced to put the data of a company under central control:
  - to avoid redundancies and inconsistencies,
  - to share the data,
  - to simplify the development of new applications,
  - to simplify administration (data stored in a single place).
- Future DBMS should manage also procedures or entire programs, not only data.

Introduction (4)

A Subprogram-Library is not the Same:

- Application programs and their dependencies (used tables) are not in data dictionary.

  If the DB schema is changed, problems are only discovered when the corresponding application program is executed.

- Such procedures exist only outside the DBMS, they cannot be used in queries and triggers.
- Application programs run on the client.
- Different client architectures $\rightarrow$ different versions.
- Changing procedures might require relinking.

Inhalt

Example (1)

```
(1)  CREATE PROCEDURE
(2)     Withdraw(acc number, amt number)
(3)  IS
(4)     b Account.Balance%TYPE;
(5)     -- b is a variable with the same type
(6)     -- as column Balance of table Account
(7)  BEGIN
(8)     SELECT Balance INTO b FROM Account
(9)        WHERE No=acc FOR UPDATE OF Balance;
(10)    IF b < amt THEN
(11)       INSERT INTO Rejected VALUES
(12)          (SYSDATE, acc, amt);
(13)    ...
```

Example (2)

```
(13)      ELSE -- Current Balance is sufficient.
(14)         b := b - amt;
(15)         UPDATE Account SET Balance = b
(16)            WHERE No = acc;
(17)         INSERT INTO Done
(18)            VALUES (SYSDATE, acc, amt);
(19)      END IF;
(20)      COMMIT;
(21)   END;
```

Lexical Syntax

- The lexical syntax of PL/SQL (defining tokens / word symbols) is intentionally very similar to SQL.
- PL/SQL is case-insensitive, except in string literals.

  And in delimited/quoted identifiers.
- Comments can be written "`--` ... ⟨`Line End`⟩" or "`/*` ... `*/`".
- Identifiers can be up to 30 characters long.
- PL/SQL has 218 reserved words.

  This includes SQL reserved words, plus about 100 new ones.

Data Types (1)

- Oracle SQL data types are also PL/SQL types.

  A `VARCHAR2`-variable can be declared with a length of up to 32767 characters in PL/SQL, whereas a database column of `VARCHAR2`-type can have only 4000 bytes.

- PL/SQL has a type `BOOLEAN` with values `TRUE`, `FALSE`.

  It can be used for variables, but not stored in the DB.

- PL/SQL also has a type `BINARY_INTEGER`.

  `BINARY_INTEGER` has values $-(2^{31}-1)$ to $+2^{31}-1$. It has e.g. the subtypes `NATURAL` for non-negative numbers and `POSITIVE` for positive numbers. `PLS_INTEGER` is the same as `BINARY_INTEGER`, except that an overflow creates an exception instead of an element of the larger `NUMBER` type.

Data Types (2)

- Procedure parameters can have only unconstrained data types, e.g. "`NUMBER`", but not "`NUMBER(4)`".
- PL/SQL has also record types, so you can store an entire database row into a single variable.
- PL/SQL also has collection types
  - `VARRAY` (array of varying size) and
  - `TABLE` (something between a set and an array).
- PL/SQL allows you to define your own data types.

Data Types (3)

- Instead of explicitly specifying a type, one can also use the following special constructs:
  - `TABLE.COLUMN%TYPE`: the type of a DB column,
  - `TABLE%ROWTYPE`: the record-type corresponding to a DB row, and
  - `VARIABLE%TYPE`: the type of another variable.

Declarations

- The basic declaration syntax is "⟨Variable⟩ ⟨Type⟩;":

```
current_balance NUMBER(8,2);
```

- You can specify an intial value, else NULL is used:

```
emp_count PLS_INTEGER := 0;
```

- You can forbid the assignment of null values:

```
zip_code CHAR(8) NOT NULL := 'PA 15260';
```

- You can forbid any future assignments:

```
credit_limit CONSTANT NUMBER := 100.00;
```

Expressions and Assignments

- Boolean expressions are similar to `WHERE`-conditions, e.g. you can use `=`, `<>`, `<`, `>`, `<=`, `>=`, `LIKE`, `BETWEEN`, `IS NULL`, `AND`, `OR`, `NOT`.
- For other data types, expressions are also very similar to SQL (what you could write in the `SELECT`-list).

  E.g. for numbers you can use +, −, ∗, /, for strings || (concatenation), and for date values −, + (number of days). In addition, the data type functions of Oracle SQL are also available in PL/SQL (`ABS`, ...).
- An assignment has the form

  ⟨Variable⟩ := ⟨Expression⟩;

Procedure Calls (1)

- As usual, a procedure call has the form

  ⟨Procedure⟩(⟨Parameter⟩,..., ⟨Parameter⟩);

  Procedures without parameters are called without "()": ⟨Procedure⟩;

- Procedures contained in packages (library modules) must be called in the form

  ⟨Package⟩.⟨Procedure⟩(...);

- E.g. in order to print something, one needs to call a procedure of Oracle's DBMS_OUTPUT package:

```
DBMS_OUTPUT.PUT_LINE('Wrong Employee name');
```

Procedure Calls (2)

- PL/SQL allows to define default values for the parameters:

```
(1)  CREATE PROCEDURE
(2)     Withdraw(acc number,
(3)                 amt number DEFAULT 60.00)
(4)  IS ...
```

- In this case, you do not have to specify values for these parameters, e.g. `Withdraw(123456)` becomes legal.

  If no default value was specified, a parameter value must be given in the call. If the declared default value is "`NULL`", the procedure can detect whether a value was actually specified.

Procedure Calls (3)

- Oracle uses default values e.g. in the `HTP` package for creating HTML output.
- E.g. the procedure `HTP.BR` is declared with two parameters, `cclear` and `cattributes`, and will print

```
<BR CLEAR="cclear" cattributes>
```

- One can also call it "`HTP.BR;`", then the parameters default to null, and only "`<BR>`" will be printed.
- PL/SQL allows besides the usual "positional" notation for parameter values also a named notation:

```
Withdraw(amt => 100.00, acc => 123456);
```

Procedure Calls (4)

- PL/SQL has not only input parameters, but also output parameters and input/output parameters:

```
(1)  CREATE PROCEDURE
(2)     Withdraw(acc IN NUMBER, amt IN NUMBER,
(3)                 ok OUT BOOLEAN)
```

- `IN` is the default, so you do not have to specify this. There is also a third mode, "`IN OUT`".
- For `IN` parameters, any expression can be used in the call, `OUT` and `IN OUT` parameters need a variable.
- Inside the procedure, `IN` parameters are write protected, and `OUT` parameters are read protected.

Conditional Statements

- An IF-statement in PL/SQL is written as follows:

```
IF ⟨Condition⟩ THEN
     ⟨Sequence of Statements⟩
ELSIF ⟨Condition⟩ THEN
     ⟨Sequence of Statements⟩
  ⋮
ELSE
     ⟨Sequence of Statements⟩
END IF;
```

- The ELSIF- and ELSE-parts are optional.

Loop Statements (1)

- The `WHILE`-statement executes a sequence of statements (the loop body) iteratively as long as a condition is true:

```
WHILE ⟨Condition⟩ LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

- E.g., we can compute the factorial $1 * 2 * 3 * \cdots * 20$:

```
factorial := 1;
i := 1;
WHILE i <= 20 LOOP
    factorial := factorial * i;
    i := i + 1;
END LOOP;
```

Loop Statements (2)

- The FOR-statement executes a sequence of statements (the loop body) once for each possible value of a control variable.

```
FOR ⟨Variable⟩ IN [REVERSE] ⟨Lower⟩..⟨Upper⟩ LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

There is also a FOR-loop for working with cursors (see below).

- E.g., we can compute the factorial $1 * 2 * 3 * \cdots * 20$:

```
factorial := 1;
FOR i IN 1..20 LOOP
    factorial := factorial * i;
END LOOP;
```

Loop Statements (3)

- The LOOP-statement executes a sequence of statements (the loop body) repeatedly until an EXIT statement is executed.

```
LOOP
    ⟨Sequence of Statements⟩
END LOOP;
```

- "EXIT;" ends the innermost enclosing loop.
- Instead of

  `IF ⟨Condition⟩ THEN EXIT; END IF;`

  you can also write

  `EXIT WHEN ⟨Condition⟩;`

Loop Statements (4)

- Computation of the factorial with LOOP statement:

```
factorial := 1;
i := 1;
LOOP
    factorial := factorial * i;
    i := i + 1;
    EXIT WHEN i > 20;
END LOOP;
```

Of course, the FOR-Loop is the right way to compute the factorial, since the needed number of iterations is known beforehand.

- You can label loops "<<Name>> LOOP ..." and then use "EXIT Name" to finish an outer loop.

SQL Statements (1)

- Every SQL command except DDL commands is also a valid PL/SQL statement.

  I.e. `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `COMMIT`, etc., but not `CREATE TABLE`.

- You can use PL/SQL variables in SQL commands whereever a constant would be allowed in SQL.

- Beware of name conflicts! PL/SQL variables are not specially marked in SQL statements.

  This is a difference to host variables in Embedded SQL, which are marked with ":". If a variable has the same name as a DB table, PL/SQL assumes that the variable is meant. A name conflict between a variable and a column is resolved in favor of the column.

SQL Statements (2)

- Example (Give a raise to a specific employee):

```
(1)  CREATE PROCEDURE
(2)     raise_sal(no NUMBER, pct NUMBER) IS
(3)  BEGIN
(4)     UPDATE EMP
(5)        SET Sal := Sal * (1 + pct/100)
(6)        WHERE EmpNo = no;
(7)     INSERT INTO Sal_Development
(8)        SELECT SYSDATE, no, Sal
(9)        FROM Emp WHERE EmpNo = no;
(10) END;
```

SQL Statements (3)

Single Row SELECT:

- SELECT statements returning a single row can be used to store values from the DB in variables:

```
SELECT Sal INTO s FROM Emp WHERE EmpNo = no;
```

- This is like an assignment to the variable "s".
- Of course, you can use SELECT INTO also with multiple columns and corresponding variables:

```
SELECT Sal, Comm INTO s, c FROM ...
```

- An exception is generated if the SELECT statement should return no row or more than one row.

Cursors (1)

- If a `SELECT` statement can return more than one row, you need a cursor (or a special kind of `FOR`-Loop).
- The cursor must be declared in the declaration section (the part before the `BEGIN`).

  You specify an SQL query when you declare the cursor.
- Then you `OPEN` the cursor (executes query).
- Next, you use a `FETCH`-statement in a loop to get the column values for each output row.
- Finally, you `CLOSE` the cursor.

Cursors (2)

- Example: Create a web page containg the names of all employees (using Oracle's HTP package).

```
(1)  CREATE PROCEDURE printEmployees IS
(2)     name Emp.EName%TYPE;
(3)     CURSOR c IS SELECT EName FROM Emp;
(4)  BEGIN
(5)     htp.htmlOpen;
(6)     htp.headOpen;
(7)     htp.title('Employee List');
(8)     htp.headClose;
(9)     htp.bodyOpen;
(10)    htp.header(1, 'Employees'); -- <H1>
```

Cursors (3)

```
(11)      htp.ulistOpen;
(12)      OPEN c;
(13)      LOOP
(14)          FETCH c INTO name;
(15)          EXIT WHEN c%NOTFOUND;
(16)          htp.listItem(name);
(17)      END LOOP;
(18)      CLOSE c;
(19)      htp.ulistClose;
(20)      htp.print('Generated: ' || SYSDATE);
(21)      htp.bodyClose;
(22)      htp.htmlClose;
(23)  END;
```

Cursors (4)

- Alternative (with `WHILE`-loop):

```
(12)  OPEN c;
(13)  FETCH c INTO name;
(14)  WHILE c%FOUND LOOP
(15)     htp.listItem(name);
(16)     FETCH c INTO name;
(17)  END LOOP;
(18)  CLOSE c;
```

Cursors (5)

- Alternative (with `FOR`-loop):

```
(12)  FOR name IN c LOOP
(13)     htp.listItem(name);
(14)  END LOOP;
```

- This also does `OPEN` and `CLOSE` automatically.
- Alternative (without explicit cursor):

```
(12)  FOR name IN (SELECT Ename FROM Emp)
(13)  LOOP
(14)     htp.listItem(name);
(15)  END LOOP;
```

Cursors (6)

- For updates in loops, one can refer to "`CURRENT OF ⟨Cursor⟩`" in `WHERE`-conditions.
- This requires a "`SELECT ... FOR UPDATE`" query.
- Example: Suppose we want to assign a unique number to every instructor. We have used `ALTER TABLE` to add a column `No` to the table `Instructor`, but it first contains null values.
- The following program also uses PL/SQL's record-types (which allow to store entire rows in a single variable).

Cursors (7)

```
(1)  CREATE PROCEDURE Number_Instructors IS
(2)     inst Instructor%ROWTYPE;
(3)     i NUMBER(4) := 1;
(4)     CURSOR c IS SELECT * FROM Instructor
(5)                    FOR UPDATE;
(6)  BEGIN
(7)     FOR inst IN c LOOP
(8)        UPDATE Instructor SET no = i
(9)           WHERE CURRENT OF c;
(10)       i := i + 1;
(11)    END LOOP;
(12) END;
```

Cursors (8)

- A cursor can have parameters, e.g.

```
(3)  CURSOR c(dno NUMBER) IS
(4)      SELECT EName FROM Emp
(5)      WHERE DeptNo = dno;
```

- Values for these parameters are specified in the OPEN-statement (when the query is executed):

```
(12)  OPEN c(20);
```

- The FOR-loop needs no explicit OPEN, so it also allows to specify the parameter value:

```
(12)  FOR name IN c(20) LOOP ... END LOOP;
```

Error Handling (1)

- Exceptions are run-time errors, like division by zero.
- The processing of SQL statements can also cause exceptions, e.g.

  ```
  SELECT Sal INTO s FROM Emp WHERE EmpNo = no;
  ```

  must return exactly one row.
- If there is no employee with the given number, Oracle will raise the exception "`NO_DATA_FOUND`".
- If the query would return more than one row, the exception "`TOO_MANY_ROWS`" will be raised.

Error Handling (2)

- Normally, exceptions will terminate your program.
- PL/SQL allows you to react on such errors by writing exception handlers, i.e. sequences of statements which are executed if such an error happens.

  Exception handlers are defined at the end of the block (with the keyword "`EXCEPTION`" before the final "`END`").

- You can write handlers for specific exceptions

  `WHEN` e_1 `OR ... OR` e_n `THEN ...`

  or write a default handler ("`WHEN OTHERS`").

Error Handling (3)

- After the exception is handled, control returns to the enclosing block.

  E.g. a procedure returns normally. Statements after the one causing the error are not executed.

- If an exception is not handled, it is propagated to the enclosing block. It may be handled there.

- You can use the procedure

  ```
  raise_application_error(error_number,
                          message);
  ```

  to create a normal Oracle error message.

  The error number must be between -20000 and -20999. Of course, this procedure can not only be used in error handlers.

Error Handling (4)

```
(1)  CREATE PROCEDURE printSalary(no NUMBER);
(2)      s NUMBER(6,2);
(3)  BEGIN
(4)      SELECT Sal INTO s WHERE EmpNo = no;
(5)      DBMS_OUTPUT.PUT_LINE('Sal = ' || s);
(6)  EXCEPTION
(7)      WHEN NO_DATA_FOUND THEN
(8)          DBMS_OUTPUT.PUT_LINE('Wrong EmpNo');
(9)  END;
```

Error Handling (5)

- You can declare your own exceptions: You define them like a variable with type "`EXCEPTION`".
- Then you use the statement "`RAISE` ⟨`Exception`⟩" to cause this exception.
- This allows, e.g., to have non-local jumps if you have many nested procedures and want to get back to some outer level.

Anonymous PL/SQL Blocks

- "Anonymous PL/SQL blocks" are pieces of code which are not part of a procedure.

```
(1)  DECLARE -- Instead of procedure head
(2)      i number;
(3)      facorial number;
(4)  BEGIN
(5)      ... -- Computation of 20!, see above
(6)      DBMS_OUTPUT.PUT_LINE(factorial);
(7)  END;
```

- You can directly enter such blocks into SQL*Plus or use them inside Embedded SQL (see below).

Functions (1)

- Functions are similar to procedures, but return a value:

```
 (1)  CREATE FUNCTION
 (2)     factorial(n INTEGER) RETURN NUMBER
 (3)  IS
 (4)     i INTEGER;
 (5)     f NUMBER := 1;
 (6)  BEGIN
 (7)     FOR i IN 1..n LOOP f := f * i;
 (8)                       END LOOP;
 (9)     RETURN f;
(10)  END;
```

Functions (2)

- Functions can be used in PL/SQL expressions, e.g.

```
x := factorial(20) / 1000;
```

- Functions can also be used in SQL queries (!):

```
SELECT n, factorial(n)
FROM   test_inputs
```

- Functions are not allowed to have side effects.
- Functions must execute a `RETURN` statement, or the exception "`PROGRAM_ERROR`" is raised.
- `RETURN;` (without value) can be used in procedures to transfer the control back to the caller.

Packages (1)

- PL/SQL has a module mechanism.
- You can define "Packages" which can contain declarations of types, procedures, functions, global variables, constants, cursors, and exceptions.
- The interface (public part) and the implementation (private part) are defined separately.
- Global variables persist for the duration of a session.

  Across transactions. But remote calls are restricted.

- When you refer to a packaged procedure etc. from outside a package, you must write "⟨Package⟩.⟨Name⟩".

Packages (2)

```
(1)  CREATE PACKAGE Com_Fun AS
(2)     FUNCTION fac(n INTEGER) RETURN NUMBER;
(3)     FUNCTION bino(n INTEGER) RETURN NUMBER;
(4)  END Com_Fun;
(5)  CREATE PACKAGE BODY Com_Fun AS
(6)     FUNCTION fac(n INTEGER) RETURN NUMBER IS
(7)        i INTEGER; f NUMBER;
(8)     BEGIN
(9)        ...
(10)    END;
(11)    ...
(12) END Com_Fun;
```

Using PL/SQL (1)

Defining Stored Procedures in SQL*Plus:

- Write the procedure definition into a file ⟨`File`⟩`.sql`, and then execute the file with `@`⟨`File`⟩.

  There must be a "/" on a line by itself after the procedure definition. After this, more procedure definitions or other commands can follow.

- If there are compilation errors, use the SQL*Plus command "`SHOW ERRORS`" to see the error messages.

  If you write `CREATE OR REPLACE PROCEDURE ...`, you can easily re-execute the source file after changing it.

- You delete a stored procedure with

  `DROP PROCEDURE` ⟨`Name`⟩`;`

Using PL/SQL (2)

Executing PL/SQL in SQL*Plus:

- You call a procedure in SQL*Plus with

  `EXECUTE ⟨Procedure⟩(⟨Parameters⟩);`

  E.g.: `EXECUTE withdraw(123456, 100.00);`

  Any statement can be `EXECUTE`d, not only procedure calls.

- The `DBMS_OUTPUT` procedures write their output into a DB table. If you want that SQL*Plus displays the output, you have to "`set serveroutput on`".

- You can directly enter anonymous PL/SQL Blocks.

- PL/SQL functions can be used in SQL queries.

Using PL/SQL (3)

Bind Variables in SQL*Plus:

- In order to call a procedure with `OUT` parameters from SQL*Plus, first declare "bind variables":

```
VARIABLE ⟨Name⟩ ⟨Type⟩
```

- Then you can use this variable, marked with "`:`", in the procedure call:

```
EXECUTE p('Input', :x);
```

- To display the current value of a bind variable, use

```
PRINT x
```

"`set autoprint on`" automatically prints bind variables.

Using PL/SQL (4)

Calling PL/SQL from Pro*C:

- PL/SQL can be used in Embedded SQL programs:

```
(1)  EXEC SQL EXECUTE
(2)     BEGIN raise_sal(:emp_no, 5); END;
(3)  END-EXEC;
```

- You can also use an anonymous PL/SQL block inside C for computations which can be done simpler in PL/SQL (but C with standard SQL is portable).

- You can use host variables (C variables) in PL/SQL whereever a PL/SQL variable would be allowed.

  As usual, host variables must be marked with a colon.

Using PL/SQL (5)

Data Dictionary Tables:

- `USER_SOURCE(NAME, TYPE, LINE, TEXT)` contains the source code of your procedures.

  `TYPE` can be, e.g., `FUNCTION`, `PACKAGE`, `PACKAGE BODY`, `PROCEDURE`, `TYPE`.

- `USER_ARGUMENTS` contains information about the arguments (parameters) of your procedures.

  Columns are, among others: `OBJECT_NAME` (Name of Procedure or Function), `PACKAGE_NAME`, `ARGUMENT_NAME`, `POSITION`, `DATA_TYPE`, `IN_OUT`.

- Procedures are registered in `OBJ` (= `USER_OBJECTS`).

  They have `OBJECT_TYPE='PROCEDURE'`. One can find for instance the creation date in this table. `USER_DEPENDENCIES` is also useful.

Inhalt

Triggers (1)

- Triggers are ECA (Event, Condition, Action) rules:
  - If a triggering event occurs,

    E.g. an insertion on a certain table.
  - and when a condition is satisfied,

    E.g. the inserted value is large.
  - an action is done.

    E.g. the tuple is inserted also in another table regularly checked by the department head.
- Thus, triggers are basically procedures that are implicitly called by the usual database commands.

  This is a kind of "hook" for own code to extend the DB.

Triggers (2)

- Typical applications of triggers are:
  - Enforcement of complex integrity constraints.

    Of course, if a constraint can be specified declaratively in the `CREATE TABLE` statement, that is much better.
  - Propagation of updates to redundant data structures (e.g. materialized views, derived columns).
  - Enforcement of complex authorizations.
  - Enforcement of complex business rules.
- Oracle also has "`INSTEAD OF`" trigger to define what should happen if a user tries to updates a view that would normally not be updatable.

Triggers in Oracle (1)

- Oracle distinguishes between
  - "Row Triggers" which are called once for each row affected by the triggering statement, and
  - "Statement Triggers" which are called only once when the triggering statement is executed.
- You can access the affected rows including the old and new attribute values only in row triggers.

  So statement triggers can for instance log that a certain type of action has taken place, but they cannot log the concrete data. Of course, a statement trigger could scan the entire table affected by the update (e.g. for integrity violations), but this is usually inefficient.

Triggers in Oracle (2)

- The triggering event is classically an `INSERT`, `UPDATE` or `DELETE` on a specific table.

  For the update, one can select specific attributes.

- Since Oracle 8i, it is also possible to react on system events (`STARTUP`, `SHUTDOWN`, `SERVERERROR`) client events (`LOGON`, `LOGOFF`), and DDL commands (`CREATE`, `ALTER`, `DROP`).

  There are "event attribute functions" to access more data for these events.

Triggers in Oracle (3)

- Oracle allows to define wether the trigger is called `BEFORE` or `AFTER` the triggering statement.

  `BEFORE` row trigger can still change the inserted/new values (e.g. set them to the allowed maximum).

- The condition part of a trigger in Oracle is quite restricted.

  You can use it only in "row triggers" and refer only to the current tuple in the condition.

- However, the action part is any PL/SQL block, so you can test arbitrary conditions there.

Example (1)

- Suppose we have a table

Inventory				
ItemNo	ItemName	Stock	MinStock	Reordered
⋮	⋮	⋮	⋮	⋮

- Our application programs change this table by

```
UPDATE Inventory SET Stock = Stock - :taken
WHERE  ItemNo = :no
```

- If the `Stock` becomes smaller than `MinStock`, we want to automatically reorder the part.

Example (2)

```
(1)  CREATE TRIGGER Reorder
(2)  AFTER UPDATE OF Stock ON Inventory
(3)  FOR EACH ROW
(4)  WHEN (new.Stock < new.MinStock
(5)        AND new.Reordered IS NULL)
(6)  BEGIN
(7)     INSERT INTO Order
(8)        VALUES (SYSDATE, :new.ItemNo);
(9)     UPDATE Inventory SET Reordered = SYSDATE
(10)       WHERE ItemNo = :new.ItemNo;
(11) END;
```

Syntax (1)

Event:

- The triggering event is specified by

```
(2)  AFTER UPDATE OF Stock ON Inventory
(3)  FOR EACH ROW
```

- So this trigger will be executed after a command

```
UPDATE Inventory SET Stock = ...
```

- It is executed once for each changed row.
- Triggers can fire on different events (but each trigger is only for one table):

```
(2)  AFTER UPDATE OF Stock, MinStock OR INSERT
(3)     ON Inventory
```

Syntax (2)

Condition:

- When the triggering event occurs, the condition is checked:

```
(4)  WHEN (new.Stock < new.MinStock
(5)        AND new.Reordered IS NULL)
```

- The condition is any `WHERE` condition without subqueries.
- The tuple variable "new" refers to the tuple after the update or insertion, "old" is the version before the update/deletion.

Syntax (3)

Condition, continued:

- If there is a name conflict with the tuple variables `new` and `old`, one can use

  ```
  REFERENCING new AS myNew
  ```

  This must be written before "`FOR EACH ROW`".

- The condition can alternatively be tested in the action (see example on next slide).
- This is probably slightly less efficient, but allows more general conditions to be tested.
- "`WHEN`" conditions can only be used in row triggers.

Syntax (4)

```
(1)  CREATE TRIGGER Reorder
(2)  AFTER UPDATE OF Stock ON Inventory
(3)  FOR EACH ROW
(4)  BEGIN
(5)      IF :new.Stock < :new.MinStock
(6)          AND :new.Reordered IS NULL THEN
(7)          INSERT INTO Order
(8)              VALUES (SYSDATE, :new.ItemNo);
(9)          UPDATE Inventory
(10)             SET Reordered = SYSDATE
(11)             WHERE ItemNo = :new.ItemNo;
(12)     END IF;
(13) END;
```

Syntax (5)

Action:

- The action part is any PL/SQL block.
- A PL/SQL block has the general form

```
(1)  DECLARE
(2)      ... -- Declarations
(3)  BEGIN
(4)      ... -- PL/SQL Statements
(5)  EXCEPTION
(6)      ... -- Exception handlers
(7)  END;
```

- The `DECLARE` and `EXCEPTION` parts are optional.

Syntax (6)

Action, continued:

- `COMMIT` or `ROLLBACK` is not allowed in the PL/SQL block (trigger action).
- However, one can call the procedure

  `raise_application_error(error_number, message)`

  to abort the SQL command which fired the trigger.

  One can select any error number between -20000 and -20999.

- The conditions (predicates)

  `INSERTING`, `DELETING`, `UPDATING`, `UPDATING(`A`)`

  can be used to react on the type of update.

Syntax (7)

Action, continued:

- In row triggers, you can use the variables `:new` and `:old`, which contain the tuple before and after the command.

  You cannot access the affected tuples in statement triggers. Note that in the `WHEN` condition, `old` and `new` are written without ":".

- In row triggers, you are not allowed to access other tuples (besides the changed one) from the updated table.

  Such triggers are executed in the middle of changing this table, so the state of the table is undefined.

Second Example

```
 (1)  CREATE TRIGGER Course_Change_Restriction
 (2)  -- Changes are allowed only on working days.
 (3)  BEFORE INSERT OR UPDATE OR DELETE ON Course
 (4)  DECLARE weekday VARCHAR(3);
 (5)  BEGIN
 (6)     weekday := TO_CHAR(SYSDATE, 'DY');
 (7)     IF weekday='SAT' OR weekday='SUN' THEN
 (8)        raise_application_error(-20000,
 (9)           'No changes allowed today.');
(10)     END IF;
(11)  END;
```

Termination and Confluence

- If the action part of a trigger does database changes, this might fire other triggers.
- It is possible that the trigger execution does not terminate.
- Also, if more than one trigger has to be executed, the system chooses an arbitrary sequence.

  In Oracle, the statement before trigger is fired first. Then for each affected row, a row before trigger is fired followed by the row after trigger. Then the statement `AFTER` trigger is fired.

- A set of triggers is confluent if the final database state does not depend on this execution sequence.

PostgreSQL Beispiel (1)

- Quelle: [https://dba.stackexchange.com/questions/196072/accessing-the-old-new-table-value-in-a-trigger-function-in-plain-sql]
- Tabellen:

```
CREATE TABLE IF NOT EXISTS orders(
                ID INT NOT NULL PRIMARY KEY,
                X INT);
CREATE TABLE IF NOT EXISTS order_items(
                ID INT NOT NULL PRIMARY KEY,
                ORDER_ID INT,
                Y INT);
```

- Daten:

```
INSERT INTO orders VALUES(1, 0);
INSERT INTO order_items VALUES(10, 1, 20);
INSERT INTO order_items VALUES(11, 1, 30);
```

PostgreSQL Beispiel (2)

- Trigger in PostgreSQL können nicht direkt Programmcode enthalten, sondern müssen eine "Trigger Function" aufrufen.

  [https://www.postgresql.org/docs/9.6/plpgsql-trigger.html]

- Trigger Functions sind dadurch gekennzeichnet, dass Sie den Ergebnistyp "`trigger`" und keine Parameter haben.
- In einer Trigger Function sind eine ganze Reihe lokaler Variablen automatisch deklariert, über diese kann man die Daten des Triggers abfragen, z.B.:
  - `OLD`: Tupel vor der Änderung (bei Row Triggern)
  - `NEW`: Tupel nach der Änderung
  - `TG_OP`: Operation, z.B. "`INSERT`".

PostgreSQL Beispiel (3)

- Beispiel für "Trigger Function" (Aktion):

```
CREATE OR REPLACE FUNCTION func_order_items()
RETURNS trigger AS
$$
BEGIN
  IF (TG_OP = 'UPDATE') THEN
    UPDATE orders
    SET X = (SELECT SUM(Y) FROM order_items
             WHERE order_id = OLD.order_id)
    WHERE ID = OLD.order_id;
```

PostgreSQL Beispiel (4)

- Beispiel für “Trigger Function”, Forts.:

```
    ELSIF (TG_OP = 'INSERT') THEN
      UPDATE orders
      SET X = (SELECT SUM(Y) FROM order_items
               WHERE order_id = NEW.order_id)
      WHERE ID = NEW.order_id;
    END IF;
    RETURN NULL;
END
$$
LANGUAGE PLPGSQL;
```

PostgreSQL Beispiel (5)

- Trigger:

```
CREATE TRIGGER trigger_order_items
AFTER INSERT OR UPDATE
ON order_items
FOR EACH ROW EXECUTE PROCEDURE func_order_items();
```

- Beispiel-Updates:

```
UPDATE order_items SET Y = 200 WHERE ID = 10;
INSERT INTO order_items VALUES (12, 1, 200);
```

Literatur/Quellen

- Elmasri/Navathe: Fundamentals of Database Systems, 3rd Edition, 1999. Section 10.5, "Programming Oracle Applications"
- R. Sunderraman: Oracle Programming — A Primer, Addison-Wesley, 1999. Chapter 4, "PL/SQL"
- Michael Gertz: Oracle/SQL Tutorial, 1999. `http://www.db.cs.ucdavis.edu/teaching/sqltutorial/`
- Oracle8 Application Developer's Guide, Oracle Corporation, 1997, Part No. A58241-01.
- PL/SQL User's Guide and Reference, Oracle Corporation, 1997, Part No. A58236-01.