Database and Information Systems

Michal Krátký & Radim Bača

Department of Computer Science
Faculty of Electrical Engineering and Computer Science
VŠB – Technical University of Ostrava

2020/2021

Content

Procedures

PL/SQL allows us to create several types of procedures. The difference is mainly in the way how they are started and whether they are stored in a database:

- Anonymous procedures.
- Named procedures.
- Named functions.

Anonymous procedures

- Anonymous procedures are not stored in a database and can not be invoked in another procedure.
- These procedures can be stored in the file or directly written on a command line and run from the client (for example Oracle SQL Developer).
- They are not precompiled and they can be therefore slower compared to named procedures.
- An anonymous procedure is simply an PL/SQL block.

Example 1

Let us have an anonymous procedure inserting an email into the table `Email` with one attribute `email` of the type `VARCHAR2(30)`.

```
DECLARE
   v_name VARCHAR2(30) := 'michal.kratky@vsb.cz';
BEGIN
   INSERT INTO Email VALUES (v_name);
END;
```

Example 2

Let us have an anonymous procedure inserting an email into the table `Email` using the query.

```
DECLARE
   v_login CHAR(6) := 'bon007';
BEGIN
   INSERT INTO Email
          SELECT email FROM Student
            WHERE login = v_login;
END;
```

Named procedures

- Named procedure contains a header with the name and parameters.
- We can invoke such a procedure in other procedures or run using the `EXECUTE` command.
- They are precompiled and stored in the database.

Structure of the named procedure 1/4

```
CREATE [OR REPLACE] PROCEDURE procedure_name
        [(parameter_name [mod] data_type, ... )]
IS|AS
        definition of local variables
BEGIN
        procedure body
END [procedure_name]
```

- `parameter_name` is the name of a parameter which is usually started by the `p_` prefix: we can simply distinguish them from local variables with the `v_` prefix.

Structure of the named procedure 2/4

```
CREATE [OR REPLACE] PROCEDURE procedure_name
        [(parameter_name [mod] data_type , ... )]
IS|AS
        definition of local variables
BEGIN
        procedure body
END [procedure_name]
```

- `mod` is a parameter mode: `IN` (input variable), `OUT` (output variable), or `IN OUT` (input output variable).

Structure of the named procedure 3/4

```
CREATE [OR REPLACE] PROCEDURE procedure_name
        [(parameter_name [mod] data_type, ... )]
IS|AS
        definition of local variables
BEGIN
        procedure body
END [procedure_name]
```

- `data_type` is a valid variable data type. `VARCHAR2`, or `NUMBER` parameters does not have the parenthesis with their length.

Structure of the named procedure 4/4

```
CREATE [OR REPLACE] PROCEDURE procedure_name
        [(parameter_name [mod] data_type, ... )]
IS|AS
        definition of local variables
BEGIN
        procedure body
END [procedure_name]
```

- Input variables in PL/SQL are implicitly passed by a reference, but it is not possible to change their values. Therefore we need to use `OUT` or `IN OUT` modes.

Example 1/2

This named procedure inserts an email of a student whose login we pass as a parameter.

```
CREATE OR REPLACE PROCEDURE
   InsertEmail(p_login VARCHAR2)
AS
   v_email VARCHAR2(60);
BEGIN
   SELECT email INTO v_email
          FROM Student WHERE login=p_login;
   INSERT INTO Email VALUES(v_email);
END;
```

Example 2/2

After a successful compilation (e.g. in Oracle SQL Developer) we can run the named procedure using the `EXECUTE` (or `EXEC`) command.

```
EXECUTE InsertEmail('jan440');
```

Compiling of an procedure

- A named procedure in a client environment (such as Oracle SQL developer) is created running code starting with `CREATE` and ending with `END;`
- The procedure is compiled and stored in the database.
  - There can be bugs which are reported in the Oracle SQL developer's log window.
    - After the issues are solved we have to compile the procedure again.
  - When the procedure is successfully compiled we can run it using `EXECUTE`.

Functions

- Functions (better say named functions) are very similar to named procedures.
- They define the return type and they have to return a value of the return type.

The structure of a function:

```
CREATE [OR REPLACE] FUNCTION function_name
        [(parameter_name [mod] data_type, ... )]
RETURN return_data_type
IS|AS
        definition of local variables
BEGIN
        procedure body
END [procedure_name]
```

Example 1/2

This named function returns an email of a student whose login we pass as a parameter.

```
CREATE OR REPLACE FUNCTION
GetStudentEmail( p_login IN Student.login%TYPE)
RETURN Student.email%TYPE AS
   v_email Student.email%TYPE;
BEGIN
   SELECT email INTO v_email FROM Student
     WHERE login = p_login;
   RETURN v_email;
END GetStudentEmail;
```

Example 2/2

We can run the named function using the `EXECUTE` (or `EXEC`) command.

```
SET SERVEROUTPUT ON;
EXECUTE DBMS_OUTPUT.PUT_LINE(
    GetStudentEmail('sob28'));
```

The first command switch on the standard output and the second command calls the function and print out the result.

Output parameters, Example 1/2

Or we can use an output parameter of a procedure.

```
CREATE OR REPLACE PROCEDURE GetStudentEmail(
   p_login IN Student.login%TYPE,
   p_email OUT Student.email%TYPE)
AS
BEGIN
  SELECT email INTO p_email FROM Student
    WHERE login = p_login;
END GetStudentEmail;
```

Output parameters, Example 2/2

And invoke it in the following way:

```
DECLARE
    v_email  Student.email%TYPE;
BEGIN
    GetStudentEmail('kra22', v_email);
    DBMS_OUTPUT.PUT_LINE(v_email);
END;
```

PL/SQL procedure and function invocation

- Using anonymous block:

```
BEGIN
 InsertEmail('jan440');
END;
```

- Using SQL:

```
SELECT InsertEmail('jan440') FROM DUAL;
```

- In Oracle SQL Developer or SQL*Plus using `EXECUTE`:

```
EXEC InsertEmail('jan440');
```

Programming constructs

- We can use several programming constructs such as conditions or cycles.
- Their syntax is very close to the similar constructs in other programming languages.

Branching

The condition has the following syntax:

```
IF condition1 THEN
   command
[ELSIF condition2 THEN command]
[ELSE commands]
END IF;
```

Cycles 1/3

Basically, there are **three types of cycles** available:

- **The first type of cycle** is ended using the `EXIT` keyword.
  - The termination condition can be written down as the `EXIT WHEN` condition.
  - In this way, we can write a cycle with a condition at the end.

```
LOOP
   cycle commands
   [EXIT; | EXIT WHEN condition;]
END LOOP;
```

Cycles 1/3 – Cycle with a condition at the end

```
DECLARE
   v_i int := 0;
BEGIN
   LOOP
      DBMS_OUTPUT.PUT_LINE('v_i: ' || v_i);
      EXIT WHEN v_i >= 5;
      v_i := v_i + 1;
   END LOOP;
END;
```

Output:

```
v_i: 0
v_i: 1
v_i: 2
v_i: 3
v_i: 4
v_i: 5
```

Cycles 2/3

The second type is a cycle with a condition at the beginning:

```
WHILE condition LOOP
   cycle commands
END LOOP;
```

Cycles 2/3 – Condition at the beginning

```
DECLARE
   v_i int := 0;
BEGIN
   WHILE v_i < 6 LOOP
      DBMS_OUTPUT.PUT_LINE('v_i: ' || v_i);
      v_i := v_i + 1;
   END LOOP;
END;
```

Output:

```
v_i: 0
v_i: 1
v_i: 2
v_i: 3
v_i: 4
v_i: 5
```

Cycles 3/3

The last type of cycle is the `FOR` cycle, where the number of iterations is known in advance. The variable `value1` represents the beginning value of the variable `variable_name` and the `value2` is the end value.

```
FOR variable_name IN [REVERSE] value1..value2 LOOP
   cycle commands
END LOOP;
```

Cycles 3/3 – Cycle FOR, Example 1

```
DECLARE
   v_i int;
BEGIN
   FOR v_i IN 0..5 LOOP
      DBMS_OUTPUT.PUT(v_i);
      IF v_i <> 5 THEN
         DBMS_OUTPUT.PUT(', ');
      END IF;
   END LOOP;
   DBMS_OUTPUT.NEW_LINE();
END;
```

Output:

```
0, 1, 2, 3, 4, 5
```

The FOR is also used for cursors where it iterates through the result of a query.

Cursors

Cursors are auxiliary variables created after processing of an SQL command, which allows us to iterate in the result. There are two types of cursors:

- *Implicit cursor* – created automatically after the INSERT, DELETE or UPDATE command.
- *Explicit cursor* – defined together with the local variables. Such a cursor is usually bounded with a SELECT command which returns more than one row.

Explicit cursor

- Definition of the explicit cursor has the following syntax:
  `CURSOR cursor_name IS select_command;`
- `select_command` returns the set of records.
- We can iterate the records and work with values.
- In one point in time, the cursor references to just one record.

Work with explicit cursor

The work with the explicit cursor is done using the following commands:

- `OPEN cursor_name` – it opens the cursor. It processes the SQL query and set the cursor on the first record.
- `FETCH cursor_name INTO record_variable` – it reads the record on an actual cursor position into the `record_variable` and moves on the next record.
- `CLOSE cursor_name` - close the cursor.

Example

We read the surname of all records in the table `Student`.

```
DECLARE
  CURSOR c_surname IS SELECT * FROM Student;
  v_record Student%ROWTYPE;
  v_tmp INTEGER := 0;
BEGIN
  OPEN c_surname;
  LOOP
    FETCH c_surname INTO v_record;
    EXIT WHEN c_surname%NOTFOUND;
    v_tmp := c_surname%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(v_tmp || v_record.surname);
  END LOOP;
  CLOSE c_surname;
END;
```

- `%NOTFOUND` – it returns true when there is no record in the cursor.
- `%ROWCOUNT` – it returns the number of records retrieved using `FETCH` so far.

Explicit cursor with FOR cycle

- Using `OPEN` and `CLOSE` for a cursor is often problematical when we forget to close the cursor.
- Using the `FOR` cycle is more simple.
- We do not have to open and close the cursor. It is done automatically.

Example, Variant 1

We read the surname of all records in the table `Student`.

```
DECLARE
   CURSOR c_surname IS SELECT surname FROM Student;
   v_surname Student.surname%TYPE;
   v_tmp NUMBER := 0;
BEGIN
   FOR one_surname IN c_surname LOOP
      v_tmp:= c_surname%ROWCOUNT;
      v_surname := one_surname.surname;
      DBMS_OUTPUT.PUT_LINE(v_tmp ||' '|| v_surname);
   END LOOP;
END;
```

Example, Variant 2

There is a variant where we do not declare the `CURSOR` variable.

```
DECLARE
   v_surname Student.surname%TYPE;
   v_tmp NUMBER := 0;
BEGIN
   FOR one_surname IN (SELECT surname FROM Student)
   LOOP
      v_tmp := v_tmp + 1;
      v_surname := one_surname.surname;
      DBMS_OUTPUT.PUT_LINE(v_tmp || ' ' || v_surname);
   END LOOP;
END;
```

OPEN/FETCH/CLOSE vs FOR

- In previous versions of Oracle the FOR cycle was not recommended when we work with bigger amount of records.
- The `FETCH BULK INTO` was recommended instead.
- Since version Oracle 10g, records are buffered per 100 records; it is very fast in both variants.
- Therefore the FOR cycle is recommended since it has more simple syntax.

References

- Oracle books:
  `https://docs.oracle.com/en/database/oracle/oracle-database/18/books.html`:
  - PL/SQL Language Reference
  - PL/SQL Packages and Types Reference