

# An Introduction to **PL/SQL**

---

Mehdi Azarmi

# Introduction

- PL/SQL is Oracle's *procedural* language extension to SQL, the non-procedural relational database language.
- Combines power and flexibility of SQL (4GL) with procedural constructs of a 3GL
- Extends SQL by adding
  - Variables and types
  - Control Structures (conditional, loops)
  - Procedures and functions
  - Exception handling
  - ...

# Block Definition

- Basic unit of PL/SQL is a block
  - Three possible sections of a block
    - Declarative section
    - Executable section
    - Exception handling
- A block performs a logical unit of work in the program
- Blocks can be nested

# Block Structure

DECLARE

```
    /* Declarative section: variables, types, and local  
subprograms. */
```

BEGIN

```
    /* Executable section: procedural and SQL statements go here.  
*/  
    /* This is the only section of the block that is required. */
```

EXCEPTION

```
    /* Exception handling section: error handling statements go  
here. */
```

END;

# Executable Section

- The only required section
- Contains constructs such as assignments, branches, loops, procedure calls, and triggers
- **SELECT, INSERT, UPDATE, DELETE** are supported
  - the SELECT statement has a special form in which a single tuple is placed in variables
- Data definition statements like **CREATE, DROP**, or ALTER are not allowed.
- PL/SQL is not case sensitive. C style comments (**/\* ... \*/**) may be used.

# Variables and Types

- Declared in the declaration section
- Variables have a specific type associated with them
- Types
  - One of the types used by SQL for database columns
  - A generic type used in PL/SQL
    - Most useful is **NUMBER** (can hold either an integer or a real number)
    - **BOOLEAN** (but not supported as a type for database columns)
  - Declared to be the same as the type of some database column
- It is essential that the variable have the same type as the relation column.
  - use the %TYPE operator

```
DECLARE
    myBeer Beers.name%TYPE;
```
  - A variable may also have a type that is a record with several fields

```
beerTuple Beers%ROWTYPE; /* (name, manufacture)*/
```

# Variables - Example

```
DECLARE
```

```
    a NUMBER := 3;
```

```
BEGIN
```

```
    a := a + 1;
```

```
END;
```

•  
`run;`                      To execute the program

- The initial value of any variable, regardless of its type, is NULL.
- This program has no effect when run, because there are no changes to the database.

# Example

```
CREATE TABLE T1(  
    e INTEGER,  
    f INTEGER  
);
```

```
DELETE FROM T1;  
INSERT INTO T1 VALUES(1, 3);  
INSERT INTO T1 VALUES(2, 4);
```

```
/* Above is plain SQL; below is the PL/SQL  
program. */
```

```
DECLARE  
    a NUMBER;  
    b NUMBER;  
BEGIN  
    SELECT e,f INTO a,b FROM T1 WHERE e>1;  
    INSERT INTO T1 VALUES(b,a);  
END;  
.  
run;
```

single tuple

cursor



# Control flow in PL/SQL

---

# IF Statement

- An IF statement looks like:

```
IF <condition>  
THEN <statement_list>  
ELSE <statement_list>  
END IF;
```

- The ELSE part is optional
- If you want a multiway branch, use:

```
IF <condition_1> THEN ...  
ELSIF <condition_2> THEN ...  
...  
ELSIF <condition_n> THEN ...  
ELSE ...  
END IF;
```

# IF - Example

```
DECLARE
    a NUMBER;
    b NUMBER;
BEGIN
    SELECT e,f INTO a,b FROM T1 WHERE e>1;
    IF b=1 THEN
        INSERT INTO T1 VALUES(b,a);
    ELSE
        INSERT INTO T1 VALUES(b+10,a+10);
    END IF;
END;

.
run;
```

# IF - Example 2

```
DECLARE
```

```
    TotalStudents NUMBER;
```

```
BEGIN
```

```
    SELECT COUNT(*)
```

```
        INTO TotalStudents
```

```
        FROM students;
```

```
...
```

# IF and UPDATE - Example

```
DECLARE
    NewMajor VARCHAR2(10) := 'CS';
    FirstName VARCHAR2(10) := 'Mehdi';
    LastName VARCHAR2(10) := 'Azarmi';
BEGIN
    UPDATE students
        SET major = NewMajor
        WHERE first_name = FirstName
        AND last_name = LastName;
    IF SQL%NOTFOUND THEN
        INSERT INTO students (ID, first_name, last_name, major)
            VALUES (student_sequence.NEXTVAL, FirstName, LastName, NewMajor);
    END IF;
END;
/
```

# Loops

- A loop allows execution of a set of statements repeatedly
- Types of loops
  - Simple loop
  - Numeric For loop
  - While loop
- Loops are created with the following:

```
LOOP
    <loop_body> /* A list of statements. */
END LOOP;
```

- At least one of the statements in <loop\_body> should be an EXIT statement of the form
- EXIT WHEN <condition>;

# LOOP - Example

```
DECLARE
    i NUMBER := 1;
BEGIN
    LOOP
        INSERT INTO T1 VALUES(i,i);
        i := i+1;
        EXIT WHEN i>100;
    END LOOP;
END;
.
run;
```

# FOR and WHILE Loops

- A **WHILE** loop can be formed with

```
WHILE <condition> LOOP
    <loop_body>
END LOOP;
```

- A simple **FOR** loop can be formed with:

```
FOR <var> IN <start>..<finish> LOOP
    <loop_body>
END LOOP;
```

- Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start> and <finish> are constants.



# FOR - Example

```
BEGIN
  FOR LoopCounter IN 1..50 LOOP
    INSERT INTO temp_table (num_col)
      VALUES (LoopCounter);
  END LOOP;
END;
/
```

# Cursors

- the SELECT statement in PL/SQL only works if the result of the query contains a **single tuple**
- If the query returns more than one tuple, or you want to manipulate a relation with more than one row, you need to use a cursor
- A **cursor** creates a named context area as a result of executing an associated SQL statement
- Permits the program to step through the multiple rows displayed by an SQL statement

# CURSOR – Example part1

```
1) DECLARE
    /* Output variables to hold the result of the query: */
2)     a T1.e%TYPE;
3)     b T1.f%TYPE;
    /* Cursor declaration: */
4)     CURSOR T1Cursor IS
5)         SELECT e, f
6)         FROM T1
7)         WHERE e < f
8)         FOR UPDATE;
9) BEGIN
10)     OPEN T1Cursor;
```

... Next page

# CURSOR– Example part2

```
11)      LOOP
          /* Retrieve each row of the result of the above query
            into PL/SQL variables: */
12)      FETCH T1Cursor INTO a, b;
          /* If there are no more rows to fetch, exit the loop: */
13)      EXIT WHEN T1Cursor%NOTFOUND;
          /* Delete the current tuple: */
14)      DELETE FROM T1 WHERE CURRENT OF T1Cursor;
          /* Insert the reverse tuple: */
15)      INSERT INTO T1 VALUES(b, a);
16)      END LOOP;
          /* Free cursor used by the query. */
17)      CLOSE T1Cursor;
18) END;
19) .
20) run;
```

# Procedure

- PROCEDURE and FUNCTIONS
- Parameters
  - **Mode** of operation:
    - IN (read-only)
    - OUT (write-only)
    - INOUT (read and write)
  - Type
- the type specifier in a parameter declaration must be **unconstrained**.
  - Example: CHAR(10) and VARCHAR(20) are illegal
  - CHAR or VARCHAR should be used instead.

# PROCEDURE - Template

```
CREATE OR REPLACE PROCEDURE PROCNAME( PARAMETERS ) AS  
<local_var_declarations>
```

```
BEGIN
```

```
    <procedure_body>
```

```
END;
```

```
.
```

```
run;
```

- The run at the end runs the statement that creates the procedure; it does not execute the procedure.
- To execute the procedure, use another PL/SQL statement, in which the procedure is invoked as an executable statement.
  - For example:

```
BEGIN addtuple1(99); END;
```

```
.
```

```
Run;
```

# PROCEDURE – Example 1

```
CREATE TABLE T2 (  
    a INTEGER,  
    b CHAR(10)  
);  
  
CREATE PROCEDURE addtuple2(  
    x IN T2.a%TYPE,  
    y IN T2.b%TYPE)  
AS  
  
BEGIN  
    INSERT INTO T2(a, b)  
    VALUES(x, y);  
END addtuple2;  
  
.  
run;
```

- Now, to add a tuple (10, 'abc') to T2:

```
BEGIN  
    addtuple2(10, 'abc');  
END;  
  
.  
run;
```

# PROCEDURE – Example 2

```
CREATE TABLE T3 (  
  a INTEGER,  
  b INTEGER  
);
```

```
CREATE PROCEDURE addtuple3(a NUMBER, b OUT NUMBER)  
AS  
BEGIN  
  b := 4;  
  INSERT INTO T3 VALUES(a, b);  
END;
```

```
.  
Run;
```

```
DECLARE  
  v NUMBER;  
BEGIN  
  addtuple3(10, v); /* second parameter should be an lvalue*/  
END;
```

```
.  
run;
```



# PROCEDURE – Final Notes

- We can also write functions instead of procedures. In a function declaration, we follow the parameter list by RETURN and the type of the return value:

- `CREATE FUNCTION <func_name>(<param_list>) RETURN <return_type> AS ...`

- In the body of the function definition, "RETURN <expression>;" exits from the function and returns the value of <expression>.

- To find out what procedures and functions you have created, use the following SQL query:

```
select object_type, object_name
from user_objects
where object_type = 'PROCEDURE' or object_type = 'FUNCTION';
```

- To drop a stored procedure/function:

```
drop procedure <procedure_name>;
drop function <function_name>;
```

# Printing

- Always use the following line (setting output buffer) at the beginning of your SQL file:

```
set serveroutput on size 32000
```

- Printing a line:

```
dbms_output.put_line(VAR1 || ' . ' || VAR2);
```

- You may declare and use a bind variable to print a local variable

```
VARIABLE x NUMBER
```

```
BEGIN
```

```
:x := 1;
```

```
END;
```

```
.
```

```
run;
```

```
PRINT :x;
```

# Debugging

- PL/SQL does not always tell you about compilation errors. Instead, it gives you a cryptic message such as:  
`"procedure created with compilation errors".`
- If you don't see what is wrong immediately, try issuing the command  
`show errors procedure <procedure_name>;`
- Alternatively, you can type, SHO ERR (short for SHOW ERRORS) to see the most recent compilation error.
- Note that the location of the error given as part of the error message is not always accurate!

# Performance of PL/SQL

- SQL results in many network trips, one for each SQL statement
- PL/SQL permits several SQL statements to be bundled into a single block
- Results in fewer calls to database
  - Less network traffic
  - faster response time

# References

- <http://infolab.stanford.edu/~ullman/fcdb/oracle/orplsql.html>
- Oracle PL/SQL Programming: Covers Versions Through Oracle Database 11g Release 2, by Steven Feuerstein and Bill Pribyl (Oct 1, 2009)