WIKIPEDIA

PL/SQL

PL/SQL (**Procedural Language for SQL)** is Oracle Corporation's procedural extension for SQL and the Oracle relational database. PL/SQL is available in Oracle Database (since version 6 - stored PL/SQL procedures/functions/packages/triggers since version 7), Times Ten in-memory database (since version 11.2.1), and IBM Db2 (since version 9.7).[1] Oracle Corporation usually extends PL/SQL functionality with each successive release of the Oracle Database.

PL/SQL includes procedural language elements such as conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variables of those types, and triggers. It can handle exceptions (run-time errors). Arrays are supported involving the use of PL/SQL collections. Implementations from version 8 of Oracle Database onwards have included features associated with object-orientation. One can create PL/SQL units such as procedures, functions, packages, types, and triggers, which are stored in the database for reuse by applications that use any of the Oracle Database programmatic interfaces.

Historically, the first public version of PL/SQL definition[2] was in 1995, and the Oracle's inception year ~1992. It implements the ISO SQL/PSM standard.[3]

Contents

PL/SQL program unit

The main feature of SQL (non-procedural) is also a drawback of SQL: one cannot use control statements (decision-making or iterative control) if only SQL is to be used. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features like other procedural programming languages. A PL/SQL program unit is one of the following: PL/SQL anonymous block, procedure, function, package specification, package body, trigger, type specification, type body, library. Program units are the PL/SQL source code that is compiled, developed and ultimately executed on the database.[4]

PL/SQL anonymous block

The basic unit of a PL/SQL source program is the block, which groups together related declarations and statements. A PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END. These keywords divide the block into a declarative part, an executable part, and an exception-handling part. The declaration section is optional and may be used to define and initialize constants and variables. If a variable is not initialized then it defaults to NULL value. The optional exception-handling part is used to handle run time errors. Only the executable part is required. A block can have a label.[5]

For example:

```
<<label>>   -- this is optional
DECLARE
-- this section is optional
  number1 NUMBER(2);
  number2 number1%TYPE := 17;             -- value default
  text1   VARCHAR2(12) := 'Hello world';
  text2   DATE         := SYSDATE;        -- current date and time
BEGIN
-- this section is mandatory, must contain at least one executable statement
  SELECT street_number
    INTO number1
    FROM address
    WHERE name = 'INU';
EXCEPTION
-- this section is optional
   WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('Error Code is ' || TO_CHAR(sqlcode));
     DBMS_OUTPUT.PUT_LINE('Error Message is ' || sqlerrm);
END;
```

The symbol `:=` functions as an assignment operator to store a value in a variable.

Blocks can be nested – i.e., because a block is an executable statement, it can appear in another block wherever an executable statement is allowed. A block can be submitted to an interactive tool (such as SQL*Plus) or embedded within an Oracle Precompiler or OCI program. The interactive tool or program runs the block once. The block is not stored in the database, and for that reason, it is called an anonymous block (even if it has a label).

Function

The purpose of a PL/SQL function is generally used to compute and return a single value. This returned value may be a single scalar value (such as a number, date or character string) or a single collection (such as a nested table or array). User-defined functions supplement the built-in functions provided by Oracle Corporation.[6]

The PL/SQL function has the form:

```
CREATE OR REPLACE FUNCTION <function_name> [(input/output variable declarations)] RETURN return_type
[AUTHID <CURRENT_USER | DEFINER>] <IS|AS>   -- heading part
amount number;   -- declaration block
BEGIN   -- executable part
    <PL/SQL block with return statement>
        RETURN <return_value>;
[Exception
    none]
        RETURN <return_value>;
END;
```

Pipe-lined table functions return collections[7] and take the form:

```
CREATE OR REPLACE FUNCTION <function_name> [(input/output variable declarations)] RETURN return_type
[AUTHID <CURRENT_USER | DEFINER>] [<AGGREGATE | PIPELINED>] <IS|USING>
    [declaration block]
BEGIN
    <PL/SQL block with return statement>
        PIPE ROW <return type>;
        RETURN;
[Exception
    exception block]
        PIPE ROW <return type>;
        RETURN;
END;
```

A function should only use the default IN type of parameter. The only out value from the function should be the value it returns.

Procedure

Procedures resemble functions in that they are named program units that can be invoked repeatedly. The primary difference is that **functions can be used in a SQL statement whereas procedures cannot**. Another difference is that the procedure can return multiple values whereas a function should only return a single value.[8]

The procedure begins with a mandatory heading part to hold the procedure name and optionally the procedure parameter list. Next come the declarative, executable and exception-handling parts, as in the PL/SQL Anonymous Block. A simple procedure might look like this:

```
CREATE PROCEDURE create_email_address ( -- Procedure heading part begins
    name1 VARCHAR2,
    name2 VARCHAR2,
    company VARCHAR2,
    email OUT VARCHAR2
) -- Procedure heading part ends
AS
-- Declarative part begins (optional)
error_message VARCHAR2(30) := 'Email address is too long.';
BEGIN -- Executable part begins (mandatory)
    email := name1 || '.' || name2 || '@' || company;
EXCEPTION -- Exception-handling part begins (optional)
WHEN VALUE_ERROR THEN
```

```
    DBMS_OUTPUT.PUT_LINE(error_message);
END create_email_address;
```

The example above shows a standalone procedure - this type of procedure is created and stored in a database schema using the CREATE PROCEDURE statement. A procedure may also be created in a PL/SQL package - this is called a Package Procedure. A procedure created in a PL/SQL anonymous block is called a nested procedure. The standalone or package procedures, stored in the database, are referred to as "stored procedures".

Procedures can have three types of parameters: IN, OUT and IN OUT.

1. An IN parameter is used as input only. An IN parameter is passed by reference, though it can be changed by the inactive program.
2. An OUT parameter is initially NULL. The program assigns the parameter value and that value is returned to the calling program.
3. An IN OUT parameter may or may not have an initial value. That initial value may or may not be modified by the called program. Any changes made to the parameter are returned to the calling program by default by copying but - with the NO-COPY hint - may be passed by reference.

PL/SQL also supports external procedures via the Oracle database's standard `ext-proc` process. [9]

Package

Packages are groups of conceptually linked functions, procedures, variables, PL/SQL table and record TYPE statements, constants, cursors, etc. The use of packages promotes re-use of code. Packages are composed of the package specification and an optional package body. The specification is the interface to the application; it declares the types, variables, constants, exceptions, cursors, and subprograms available. The body fully defines cursors and subprograms, and so implements the specification. Two advantages of packages are:[10]

1. Modular approach, encapsulation/hiding of business logic, security, performance improvement, re-usability. They support object-oriented programming features like function overloading and encapsulation.
2. Using package variables one can declare session level (scoped) variables since variables declared in the package specification have a session scope.

Trigger

A database trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs. It is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is enabled, the database automatically invokes it—that is, the trigger fires—whenever its triggering event occurs. While a trigger is disabled, it does not fire.

You create a trigger with the CREATE TRIGGER statement. You specify the triggering event in terms of triggering statements, and the item they act on. The trigger is said to be created on or defined on the item—which is either a table, a view, a schema, or the database. You also specify the timing point, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects.

If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a DML trigger. If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a system trigger.

An INSTEAD OF trigger is either: A DML trigger created on a view or a system trigger defined on a CREATE statement. The database fires the INSTEAD OF trigger instead of running the triggering statement.

Purpose of triggers

Triggers can be written for the following purposes:

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Data types

The major datatypes in PL/SQL include NUMBER, CHAR, VARCHAR2, DATE and TIMESTAMP.

Numeric variables

```
variable_name number([P, S]) := 0;
```

To define a numeric variable, the programmer appends the variable type **NUMBER** to the name definition. To specify the (optional) precision (P) and the (optional) scale (S), one can further append these in round brackets, separated by a comma. ("Precision" in this context refers to the number of digits the variable can hold, and "scale" refers to the number of digits that can follow the decimal point.)

A selection of other data-types for numeric variables would include: binary_float, binary_double, dec, decimal, double precision, float, integer, int, numeric, real, small-int, binary_integer.

Character variables

```
variable_name varchar2(20) := 'Text';

-- e.g.:
address varchar2(20) := 'lake view road';
```

To define a character variable, the programmer normally appends the variable type VARCHAR2 to the name definition. There follows in brackets the maximum number of characters the variable can store.

Other datatypes for character variables include: varchar, char, long, raw, long raw, nchar, nchar2, clob, blob, and bfile.

Date variables

```
variable_name date := to_date('01-01-2005 14:20:23', 'DD-MM-YYYY hh24:mi:ss');
```

Date variables can contain date and time. The time may be left out, but there is no way to define a variable that only contains the time. There is no DATETIME type. And there is a TIME type. But there is no TIMESTAMP type that can contain fine-grained timestamp up to millisecond or nanosecond. The `TO_DATE` function can be used to convert strings to date values. The function converts the first quoted string into a date, using as a definition the second quoted string, for example:

```
 to_date('31-12-2004', 'dd-mm-yyyy')
```

or

```
 to_date ('31-Dec-2004', 'dd-mon-yyyy', 'NLS_DATE_LANGUAGE = American')
```

To convert the dates to strings one uses the function `TO_CHAR (date_string, format_string)`.

PL/SQL also supports the use of ANSI date and interval literals.[11] The following clause gives an 18-month range:

```
WHERE dateField BETWEEN DATE '2004-12-30' - INTERVAL '1-6' YEAR TO MONTH
    AND DATE '2004-12-30'
```

Exceptions

Exceptions—errors during code execution—are of two types: user-defined and predefined.

User-defined exceptions are always raised explicitly by the programmers, using the `RAISE` or `RAISE_APPLICATION_ERROR` commands, in any situation where they determine it is impossible for normal execution to continue. The `RAISE` command has the syntax:

```
RAISE <exception name>;
```

Oracle Corporation has *predefined* several exceptions like `NO_DATA_FOUND`, `TOO_MANY_ROWS`, *etc.* Each exception has an SQL error number and SQL error message associated with it. Programmers can access these by using the `SQLCODE` and `SQLERRM` functions.

Datatypes for specific columns

```
Variable_name Table_name.Column_name%type;
```

This syntax defines a variable of the type of the referenced column on the referenced tables.

Programmers specify user-defined datatypes with the syntax:

```
type data_type is record (field_1 type_1 := xyz, field_2 type_2 := xyz, ..., field_n type_n := xyz);
```

For example:

```
declare
    type t_address is record (
        name address.name%type,
        street address.street%type,
        street_number address.street_number%type,
        postcode address.postcode%type);
    v_address t_address;
begin
    select name, street, street_number, postcode into v_address from address where rownum = 1;
end;
```

This sample program defines its own datatype, called *t_address*, which contains the fields *name*, *street*, *street_number* and *postcode*.

So according to the example, we are able to copy the data from the database to the fields in the program.

Using this datatype the programmer has defined a variable called *v_address* and loaded it with data from the ADDRESS table.

Programmers can address individual attributes in such a structure by means of the dot-notation, thus:

```
v_address.street := 'High Street';
```

Conditional statements

The following code segment shows the IF-THEN-ELSIF-ELSE construct. The ELSIF and ELSE parts are optional so it is possible to create simpler IF-THEN or, IF-THEN-ELSE constructs.

```
IF x = 1 THEN
   sequence_of_statements_1;
ELSIF x = 2 THEN
   sequence_of_statements_2;
ELSIF x = 3 THEN
   sequence_of_statements_3;
ELSIF x = 4 THEN
   sequence_of_statements_4;
ELSIF x = 5 THEN
   sequence_of_statements_5;
ELSE
   sequence_of_statements_N;
END IF;
```

The CASE statement simplifies some large IF-THEN-ELSIF-ELSE structures.

```
CASE
   WHEN x = 1 THEN sequence_of_statements_1;
   WHEN x = 2 THEN sequence_of_statements_2;
   WHEN x = 3 THEN sequence_of_statements_3;
   WHEN x = 4 THEN sequence_of_statements_4;
   WHEN x = 5 THEN sequence_of_statements_5;
   ELSE sequence_of_statements_N;
END CASE;
```

CASE statement can be used with predefined selector:

```
CASE x
   WHEN 1 THEN sequence_of_statements_1;
   WHEN 2 THEN sequence_of_statements_2;
   WHEN 3 THEN sequence_of_statements_3;
   WHEN 4 THEN sequence_of_statements_4;
   WHEN 5 THEN sequence_of_statements_5;
```

```
    ELSE sequence_of_statements_N;
  END CASE;
```

Array handling

PL/SQL refers to arrays as "collections". The language offers three types of collections:

1. Associative arrays (Index-by tables)
2. Nested tables
3. Varrays (variable-size arrays)

Programmers must specify an upper limit for varrays, but need not for index-by tables or for nested tables. The language includes several collection methods used to manipulate collection elements: for example FIRST, LAST, NEXT, PRIOR, EXTEND, TRIM, DELETE, etc. Index-by tables can be used to simulate associative arrays, as in this example of a memo function for Ackermann's function in PL/SQL.

Associative arrays (index-by tables)

With index-by tables, the array can be indexed by numbers or strings. It parallels a Java *map*, which comprises key-value pairs. There is only one dimension and it is unbounded.

Nested tables

With nested tables the programmer needs to understand what is nested. Here, a new type is created that may be composed of a number of components. That type can then be used to make a column in a table, and nested within that column are those components.

Varrays (variable-size arrays)

With Varrays you need to understand that the word "variable" in the phrase "variable-size arrays" doesn't apply to the size of the array in the way you might think that it would. The size the array is declared with is in fact fixed. The number of elements in the array is variable up to the declared size. Arguably then, variable-sized arrays aren't that variable in size.

Cursors

A cursor is a pointer to a private SQL area that stores information coming from a SELECT or data manipulation language (DML) statement (INSERT, UPDATE, DELETE, or MERGE). A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.[12]

A cursor can be explicit or implicit. In a FOR loop, an explicit cursor shall be used if the query will be reused, otherwise an implicit cursor is preferred. If using a cursor inside a loop, use a FETCH is recommended when needing to bulk collect or when needing dynamic SQL.

Looping

As a procedural language by definition, PL/SQL provides several iteration constructs, including basic LOOP statements, WHILE loops, FOR loops, and Cursor FOR loops. Since Oracle 7.3 the REF CURSOR type was introduced to allow recordsets to be returned from stored procedures and functions. Oracle 9i introduced the predefined SYS_REFCURSOR type, meaning we no longer have to define our own REF CURSOR types.

LOOP statements

```
<<parent_loop>>
LOOP
    statements

    <<child_loop>>
    loop
        statements
        exit parent_loop when <condition>; -- Terminates both loops
        exit when <condition>; -- Returns control to parent_loop
    end loop child_loop;
        if <condition> then
           continue; -- continue to next iteration
        end if;

    exit when <condition>;
END LOOP parent_loop;
```

[13]

Loops can be terminated by using the `EXIT` keyword, or by raising an exception.

FOR loops

```
DECLARE
    var NUMBER;
BEGIN
    /* N.B. for loop variables in PL/SQL are new declarations, with scope only inside the loop */
    FOR var IN 0 .. 10 LOOP
        DBMS_OUTPUT.PUT_LINE(var);
    END LOOP;

    IF var IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('var is null');
    ELSE
        DBMS_OUTPUT.PUT_LINE('var is not null');
    END IF;
END;
```

Output:

```
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
 var is null
```

Cursor FOR loops

```
FOR RecordIndex IN (SELECT person_code FROM people_table)
LOOP
  DBMS_OUTPUT.PUT_LINE(RecordIndex.person_code);
END LOOP;
```

Cursor-for loops automatically open a cursor, read in their data and close the cursor again.

As an alternative, the PL/SQL programmer can pre-define the cursor's SELECT-statement in advance to (for example) allow re-use or make the code more understandable (especially useful in the case of long or complex queries).

```
DECLARE
  CURSOR cursor_person IS
    SELECT person_code FROM people_table;
BEGIN
  FOR RecordIndex IN cursor_person
  LOOP
    DBMS_OUTPUT.PUT_LINE(recordIndex.person_code);
  END LOOP;
END;
```

The concept of the person_code within the FOR-loop gets expressed with dot-notation ("."):

```
RecordIndex.person_code
```

Dynamic SQL

While programmers can readily embed Data Manipulation Language (DML) statements directly into PL/SQL code using straightforward SQL statements, Data Definition Language (DDL) requires more complex "Dynamic SQL" statements in the PL/SQL code. However, DML statements underpin the majority of PL/SQL code in typical software applications.

In the case of PL/SQL dynamic SQL, early versions of the Oracle Database required the use of a complicated Oracle `DBMS_SQL` package library. More recent versions have however introduced a simpler "Native Dynamic SQL", along with an associated `EXECUTE IMMEDIATE` syntax.

Similar languages

PL/SQL works analogously to the embedded procedural languages associated with other relational databases. For example, Sybase ASE and Microsoft SQL Server have Transact-SQL, PostgreSQL has PL/pgSQL (which emulates PL/SQL to an extent), MariaDB includes a PL/SQL compatibility parser,[14] and IBM Db2 includes SQL Procedural Language,[15] which conforms to the ISO SQL's SQL/PSM standard.

The designers of PL/SQL modeled its syntax on that of Ada. Both Ada and PL/SQL have Pascal as a common ancestor, and so PL/SQL also resembles Pascal in most aspects. However, the structure of a PL/SQL package does not resemble the basic Object Pascal program structure as implemented by a Borland Delphi or Free Pascal unit. Programmers can define public and private global data-types, constants, and static variables in a PL/SQL package.[16]

PL/SQL also allows for the definition of classes and instantiating these as objects in PL/SQL code. This resembles usage in object-oriented programming languages like Object Pascal, C++ and Java. PL/SQL refers to a class as an "Abstract Data Type" (ADT) or "User Defined Type" (UDT), and defines it as an Oracle SQL data-type as opposed to a PL/SQL user-defined type, allowing its use in both the Oracle SQL Engine and the Oracle PL/SQL engine. The constructor and methods of an Abstract Data Type are written in PL/SQL. The resulting Abstract Data Type can operate as an object class in PL/SQL. Such objects can also persist as column values in Oracle database tables.