PL/SQL

Sangeeta Bhandari
PG Department of Computer Sc. & IT

PL/SQL : INTRODUCTION

- PL/SQL is Oracle's *procedural* language extension to SQL, the non-procedural relational database language.

- With PL/SQL, you can use SQL statements to manipulate ORACLE data and the *flow* of control statements to process the data. Moreover, you can declare constants and variables, define subprograms (procedures and functions), and trap runtime errors. Thus, PL/SQL combines the data manipulating power of SQL with the data processing power of procedural languages.

PL/SQL : INTRODUCTION

- While PL/SQL is just like any other programming language, it has syntax and rules that determine how programming statements work together. It is important for you to realize that PL/SQL is not a stand-alone programming language.

- PL/SQL is a part of the Oracle RDBMS

PL/SQL : INTRODUCTION

- Procedural extension allowing for modularity, variable declaration, loops and logical constructs.
- Allows for advanced error handling
- Communicates natively with other oracle database objects.
- Managed centrally within the Oracle database.

PL/SQL : INTRODUCTION

- SQL is not functionally complete
  - Lacks full facilities of a programming language
- So top up functionality by embedding SQL in a procedural language
- PL/SQL techniques are specific to Oracle
  - but procedures and functions can be ported to other systems

Why use PL/SQL

- Manage business rules – through *middle layer* application logic.
- Generate code for triggers
- Generate code for interface
- Enable database-centric client/server applications
- Permits all operations of standard programming languages e.g.
  - Conditions IF-THEN-ELSE-END IF;
  - Jumps GOTO

Why use PL/SQL

- Provides loops for controlling iteration
  - LOOP-EXIT; WHEN-END LOOP; FOR-END LOOP; WHILE-END LOOP
- Allows extraction of data into variables and its subsequent manipulation

PL/SQL BLOCKS

- PL/SQL blocks can be divided into two groups:
  1. Named and
  2. Anonymous.
- Named blocks are used when creating subroutines. These subroutines are procedures, functions, and packages.
- The subroutines can be stored in the database and referenced by their names later on.
- In addition, subroutines can be defined within the anonymous PL/SQL block.
- Anonymous PL/SQL blocks do not have names. As a result, they cannot be stored in the database and referenced later.

PL/SQL BLOCK STRUCTURE

- PL/SQL blocks contain three sections
  1. Declare section
  2. Executable section and
  3. Exception-handling section.

- The executable section is the only mandatory section of the block.
- Both the declaration and exception-handling sections are optional.

PL/SQL BLOCK STRUCTURE

- Declaration section (optional)
  - Any needed variables declared here
- Executable or begin section
  - Program code such as statements to retrieve or manipulate data in a table
- Exception section (optional)
  - Error traps can catch situations which might ordinarily crash the program

DECLARATION SECTION

- The *declaration section* is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants, cursors and so on.
- Example
  - DECLARE
  - v_first_name VARCHAR2(35) ;
  - v_last_name VARCHAR2(35) ;
  - v_counter NUMBER := 0 ;

EXECUTABLE SECTION

- The executable section is the next section of the PL/SQL block.
- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
    SELECT first_name, last_name
            INTO v_first_name, v_last_name
            FROM student
            WHERE student_id = 123 ;
    DBMS_OUTPUT.PUT_LINE
    ('Student name :' || v_first_name ||' '||
v_last_name);
END;
```

EXCEPTION-HANDLING SECTION

- The *exception-handling section* is the last section of the PL/SQL block.
- This section contains statements that are executed when a runtime error occurs within a block.

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  DBMS_OUTPUT.PUT_LINE
            (' There is no student with student id 123 ');
END;
```

PL/SQL EXAMPLE

```
DECLARE
 v_first_name VARCHAR2(35);
 v_last_name VARCHAR2(35);
BEGIN
 SELECT first_name, last_name
 INTO v_first_name, v_last_name
 FROM student
 WHERE student_id = 123;
 DBMS_OUTPUT.PUT_LINE
 ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
 WHEN NO_DATA_FOUND THEN
                        DBMS_OUTPUT.PUT_LINE
                                ('There is no student with student id
123');
END;
```

EXECUTING PL/SQL

- PL/SQL can be executed directly in SQL*Plus. A PL/SQL program is normally saved with an .sql extension. To execute an anonymous PL/SQL program, simply type the following command at the SQL prompt:
- SQL> @DisplayAge

GENERATING OUTPUT

- Like other programming languages, PL/SQL provides a procedure (i.e. PUT_LINE) to allow the user to display the output on the screen. For a user to able to view a result on the screen, two steps are required.
- First, before executing any PL/SQL program, type the following command at the SQL prompt (Note: you need to type in this command only once for every SQL*PLUS session):
- SQL> SET SERVEROUTPUT ON;
- or put the command at the beginning of the program, right before the declaration section.

EXAMPLE

```
DECLARE
v_student_id NUMBER := &sv_student_id;
v_first_name VARCHAR2(35);
v_last_name VARCHAR2(35);
BEGIN
SELECT first_name, last_name
INTO v_first_name, v_last_name
FROM student
WHERE student_id = v_student_id;
DBMS_OUTPUT.PUT_LINE
                        ('Student name: '||v_first_name||'
'||v_last_name);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('There is no such student');
END;
```

EXAMPLE

- When this example is executed, the user is asked to provide a value for the student ID.
- The example shown above uses a single ampersand for the substitution variable.
- When a single ampersand is used throughout the PL/SQL block, the user is asked to provide a value for each occurrence of the substitution variable.

PL/SQL Variables

- Variables are local to the code block
- Names can be up to 30 characters long and must begin with a character
- Declaration is like that in a table
  - Name then data type the semi-colon
  - Can be initialized using := operator in the declaration
  - Can be changed with := in the begin section
  - Can use constraints
- Variables can be composite or collection types
  - Multiple values of different or same type

Common PL/SQL Data Types

- CHAR (max_length)
- VARCHAR2 (max_length)
- NUMBER (precision, scale)
- BINARY_INTEGER – more efficient than number
- RAW (max_length)
- DATE
- BOOLEAN (true, false, null)
- Also LONG, LONG RAW and LOB types but the capacity is usually less in PL/SQL than SQL

PL/SQL Variable Constraints

- NOT NULL
  - Can not be empty
- CONSTANT
  - Can not be changed

Conditional Structures

- IF-THEN
- IF-THEN-ELSE
- IF-THEN-ELSIF
  - An alternative to nested IF-THEN_ELSE

Modules in PL/SQL

There are 4 types of modules in PL/SQL

- **Procedures** – series of statements may or may not return a value
- **Functions** – series of statements must return a single value
- **Triggers** – series of PL/SQL statements (actions) executing after an event has triggered a condition (ECA)
- **Packages** – collection of procedures and function that has 2 parts:
  - a listing and a body.

Stored Procedures

```
CREATE PROCEDURE ProcedureName(parameter1 datatype, parameter2 datatype, ...) AS
   Additional declarations of local variables
BEGIN
   executable section
EXCEPTION
   Optional exception section
END;
```

Stored Procedures

- The first line is called the Procedure Specification
- The remainder is the Procedure Body
- A procedure is compiled and loaded in the database as an object
- Procedures can have parameters passed to them

Stored Procedures

- Run a procedure with the PL/SQL EXECUTE command
- Parameters are enclosed in parentheses

Using SQL in procedures

- Select values into PL/SQL variables
  - using INTO
- **Record.element** notation will address components of tuples *(dot notation)*
- **%rowtype** allows full rows to be selected into one variable
- V_employee employee%rowtype

Empid	enam e	doj	addr	sal	job	deptn o	phno

Example (Anonymous Block of Code)

```
Declare
   v_employee    employee%rowtype;
Begin
   select *
   into v_employee
   from employee
     where empid = 65284;

   update employee
   set salary = v_employee.salary + 1000
      where empid = v_employee.empid;
End;
```

Stored Functions

- Like a procedure except they return a single value

Triggers

- Associated with a particular table
- Automatically executed when a particular event occurs
  - Insert
  - Update
  - Delete
  - Others

Triggers vs. Procedures

- Procedures are explicitly executed by a user or application
- Triggers are implicitly executed (fired) when the triggering event occurs
- Triggers should not be used as a lazy way to invoke a procedure as they are fired every time the event occurs

Triggers

```
CREATE TRIGGER TriggerName
BEFORE [AFTER] event[s] ON TableName
[FOR EACH ROW]
DECLARE
   Declaration of any local variables
BEGIN
   Statements in Executable section
EXCEPTION
   Statements in optional Exception section
END;
/
```

Triggers

- The trigger specification names the trigger and indicates when it will fire
- The trigger body contains the PL/SQL code to accomplish whatever task(s) need to be performed

Triggers

```
CREATE TRIGGER TriggerName
BEFORE [AFTER] event[s] ON TableName
[FOR EACH ROW]
DECLARE
   Declaration of any local variables
BEGIN
   Statements in Executable section
EXCEPTION
   Statements in optional Exception section
END;
/
```

Triggers Timing

- A triggers timing has to be specified first
  - Before (most common)
    - Trigger should be fired before the operation
      - i.e. before an insert
  - After
    - Trigger should be fired after the operation
      - i.e. after a delete is performed

Trigger Events

- Three types of events are available
  - DML events
  - DDL events
  - Database events

DML Events

- Changes to data in a table
  - Insert
  - Update
  - Delete

DDL Events

- Changes to the definition of objects
  - Tables
  - Indexes
  - Procedures
  - Functions
  - Others
    - Include CREATE, ALTER and DROP statements on these objects

Database Events

- Server Errors
- Users Log On or Off
- Database Started or Stopped

Trigger DML Events

- Can specify one or more events in the specification
  - i.e. INSERT OR UPDATE OR DELETE
- Can specify one or more columns to be associated with a type of event
  - i.e. BEFORE UPDATE OF SID OR SNAME

Trigger Level

- Two levels for Triggers
  - Row-level trigger
    - Requires FOR EACH ROW clause
      - If operation affects multiple rows, trigger fires once for each row affected
  - Statement-level trigger
  - DML triggers should be row-level
  - DDL and Database triggers should not be row-level

Event Examples

```
Example 1:
CREATE TRIGGER NameChange
BEFORE UPDATE OF STUDENT_FIRST_NAME, STUDENT_LAST_NAME ON STUDENT
FOR EACH ROW

Example 2:

CREATE TRIGGER AlterStudent
AFTER INSERT OR UPDATE OR DELETE ON STUDENT
FOR EACH ROW

Example 3:

CREATE TRIGGER ErrorLog
AFTER SERVERERROR ON DATABASE

Example 4:

CREATE Trigger TrackChanges
AFTER CREATE ON SCHEMA
```

Cursors

- Cursors Hold Result of an SQL Statement
- Two Types of Cursors in PL/SQL
  - Implicit – Automatically Created When a Query or Manipulation is for a Single Row
  - Explicit – Must Be Declared by the User
    - Creates a Unit of Storage Called a Result Set

Cursors

- Declaring an Explicit Cursor

  CURSOR CursorName IS SelectStatement;
- Opening an Explicit Cursor

  OPEN CursorName;
- Accessing Rows from an Explicit Cursor

  FETCH CursorName INTO RowVariables;

Cursors

- Declaring Variables of the Proper Type with %TYPE

  VarName TableName.FieldName%TYPE;
- Declaring Variables to Hold An Entire Row

  VarName CursorName%ROWTYPE;
- Releasing the Storage Area Used by an Explicit Cursor

  CLOSE CursorName;

Iterative Structures

- LOOP … EXIT … END LOOP
  - EXIT with an If Avoids Infinite Loop
- LOOP … EXIT WHEN … END LOOP
  - Do Not Need An If to Control EXIT
- WHILE … LOOP … END LOOP
  - Eliminates Need for EXIT
- FOR … IN … END LOOP
  - Eliminates Need for Initialization of Counter

Cursor Control With Loops

- Need a Way to Fetch Repetitively
- Need a Way to Determine How Many Rows to Process With a Cursor
  - Cursor Attributes
    - **CursorName%ROWCOUNT – Number of Rows in a Result Set**
    - **CursorName%FOUND – True if a Fetch Returns a Row**
    - **CursorName%NOTFOUND – True if Fetch Goes Past Last Row**

Cursor For Loop

- Processing an Entire Result Set Common
- Special Form of FOR … IN to Manage Cursors
- No Need for Separate OPEN, FETCH and CLOSE statements
- Requires %ROWTYPE Variable

Example

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' ||
 c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

OutPut

1 Ramesh Allahabad
2 Suresh Kanpur
3 Mahesh Ghaziabad
4 Chandan Noida
5 Alex Paris
6 Sunita Delhi
PL/SQL procedure successfully completed.

THANKS