INTRODUCTION TO PL/SQL

Introduction to PL/SQL

- Procedural Language extension for SQL
- Oracle Proprietary
- 3GL Capabilities
- Integration of SQL
- Portable within Oracle data bases
- Callable from any client

- PL/SQL is a procedural language designed specifically to embrace SQL statements within its syntax.
- PL/SQL program units are compiled by the Oracle Database server and stored inside the database.
- And at run-time, both PL/SQL and SQL run within the same server process, bringing optimal efficiency.
- PL/SQL automatically inherits the robustness, security, and portability of the Oracle Database.
- An application that uses Oracle Database is worthless unless only correct and complete data is persisted.
- The time-honored way to ensure this is to expose the database only via an interface that hides the implementation details -- the tables and the SQL statements that operate on these.
- This approach is generally called the thick database paradigm, because PL/SQL subprograms inside the database issue the SQL statements from code that implements the surrounding business logic; and because the data can be changed and viewed only through a PL/SQL interface.

Why use PL/SQL?

- The implementation details are the tables and the SQL statements that manipulate them. These are hidden behind a PL/SQL interface.
- This is the Thick Database paradigm: select, insert, update, delete, merge, commit, and rollback are issued only from database PL/SQL.
- Developers and end-users of applications built this way are happy with their correctness, maintainability, security, and performance.
- But when developers follow the NoPlsql paradigm, their applications have problems in each of these areas and end-users suffer.

Structure of PL/SQL

- PL/SQL is Block Structured

  A block is the basic unit from which all PL/SQL programs are built. A block can be named (functions and procedures) or anonymous.

- Sections of block

  1- Header Section

  2- Declaration Section

  3- Executable Section

  4- Exception Section

Structure of PL/SQL

HEADER

Type and Name of block

DECLARE

Variables; Constants; Cursors;

BEGIN

PL/SQL and SQL Statements

EXCEPTION

Exception handlers

END;

- PL/SQL block is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END, which break up the block into three sections:
- Declarative: statements that declare variables, constants, and other code elements, which can then be used within that block
- Executable: statements that are run when the block is executed
- Exception handling: a specially structured section you can use to “catch,” or trap, any exceptions that are raised when the executable section runs
- Only the executable section is required. You don’t have to declare anything in a block, and you don’t have to trap exceptions raised in that block.
- A block itself is an executable statement, so you can nest blocks within other blocks.

```
DECLARE
        a number;
        text1 varchar2(20);
        text2 varchar2(20) := “HI”;
BEGIN
      --------- --------- ----------
END;
```

Important Data Types in PL/SQL include NUMBER, INTEGER, CHAR, VARCHAR2, DATE etc

to_date(‘02-05-2007','dd-mm-yyyy') { Converts String to Date}

The classic “Hello World!” block contains an executable section that calls the DBMS_OUTPUT.PUT_LINE procedure to display text on the screen:
BEGIN **DBMS_OUTPUT.put_line** ('Hello World!'); END;

Next, block declares a variable of type VARCHAR2 (string) with a maximum length of 100 bytes to hold the string ‘Hello World!’. DBMS_OUTPUT.PUT_LINE then accepts the variable, rather than the literal string, for display:
DECLARE l_message VARCHAR2 (100) := 'Hello World!'; BEGIN DBMS_OUTPUT.put_line (l_message); END;

Next example block adds an exception section that traps any exception (WHEN OTHERS) that might be raised and displays the error message, which is returned by the SQL function (provided by Oracle).
DECLARE l_message VARCHAR2 (100) := 'Hello World!'; BEGIN DBMS_OUTPUT.put_line (l_message); EXCEPTION WHEN OTHERS THEN DBMS_OUTPUT.put_line (SQL); END;

The following example block demonstrates the PL/SQL ability to nest blocks within blocks as well as the use of the *concatenation* operator (||) to join together multiple strings.

```
DECLARE l_message VARCHAR2 (100) := 'Hello';
BEGIN DECLARE l_message2 VARCHAR2 (100) := l_message || ' World!';
BEGIN DBMS_OUTPUT.put_line (l_message2); END; EXCEPTION WHEN
OTHERS THEN DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

There are many different tools for executing PL/SQL code. The most basic is SQL*Plus, a command-line interface for executing SQL statements as well as PL/SQL blocks.

Whereas some developers continue to rely solely on SQL*Plus, most use an integrated development environment (IDE). Among the most popular of these IDEs are:

Oracle SQL Developer, from Oracle
Toad and SQL Navigator, from Quest Software
PL/SQL Developer, from Allround Automations

Structure of PL/SQL

- Data Types 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 table

PL/SQL Control Structure

- PL/SQL has a number of control structures which includes:
- Conditional controls
- Iterative or loop controls.
- Exception or error controls

- It is these controls, used singly or together, that allow the PL/SQL developer to direct the flow of execution through the program.

PL/SQL Control Structure

- **Conditional Controls**

  IF....THEN....END IF;

  IF....THEN...ELSE....END IF;

  IF....THEN...ELSIF....THEN....ELSE....END IF;

PL/SQL Control Structure

- LOOP
  ...SQL Statements...
  EXIT;
  END LOOP;

- WHILE loops
- WHILE condition LOOP
  ...SQL Statements...
  END LOOP;

- FOR loops
- FOR <variable(numeric)> IN [REVERSE] <lowerbound>..<upperbound> LOOP .... END LOOP;

PL/SQL Control Structure

- **Cursor**

```
DECLARE
        name varchar2(20);
       Cursor c1 is
         select t.name
          from table t
          where date is not null;
BEGIN
    OPEN c1;
     LOOP
         FETCH c1 into name;
         exit when c1%NOTFOUND;
    END LOOP;
   CLOSE c1;
END;
```

Debuging

- show error
- DBMS_OUTPUT.PUT_LINE(‘ .. ‘);

Execution

- How to execute a function in PL/SQL?

  Var issue_flag number;

  exec :issue_flag:=fun_name(arg1,arg2,. . . .);

  PRINT :issue_flag;

- How to execute a procedure in PL/SQL?

  Exec procedure_name(arg1,arg2,. . . .);

PL/SQL

- Originally modeled after ADA
  - Created for Dept. of Defense
- Allows expanded functionality of database applications
- Continues to improve with each new database release

PL/SQL

- Features
  - Tight integration with SQL
    - Supports data types, functions, pseudo-columns, etc.
  - Increased performance
    - A block of statements sent as a single statement
  - Increased productivity
    - Same techniques can be used with most Oracle products
  - Portability
    - Works on any Oracle platform
  - Tighter security
    - Users may access database objects without granted privileges

PL/SQL Programs

- 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

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

PL/SQL Variables Examples

Age number;

Last char (10);

DVal Date := Sysdate;

SID number not null;

Adjust constant number := 1;

CanLoop boolean := true

Conditional Structures

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

IF-THEN Structure

```
IF [T/F condition] THEN
   statements to perform when condition is true;
END IF;
```

IF-THEN-ELSE Structure

```
IF [T/F condition] THEN
   statements to perform when condition is true;
ELSE
   statements to perform when condition is false;
END IF;
```

IF-THEN-ELSIF Structure

```
IF [first T/F condition] THEN
   statements to perform when first condition is true;
ELSIF [second T/F condition] THEN
   statements to perform when second condition is true;
ELSIF [third T/F condition] THEN
   statements to perform when third condition is true;
ELSE
   statements to perform when all conditions are false;
END IF;
```

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

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

Table Name

- The next item in the trigger is the name of the table to be affected

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
```

Triggers

- Conditions Available So Multiple Operations Can Be Dealt With In Same Trigger
  - Inserting, Updating, Deleting
- Column Prefixes Allow Identification Of Value Changes
  - New, Old

Triggers Exceptions

- EXCEPTION Data Type Allows Custom Exceptions
- RAISE Allows An Exception To Be Manually Occur
- RAISE_APPLICATION_ERROR Allows Termination Using A Custom Error Message
  - Must Be Between -20000 and -20999
  - Message Can Be Up to 512 Bytes

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

Result Set

MIS380	**DATABASE DESIGN**	**4**
MIS202	**INFORMATION SYSTEMS**	**3**
<Cursor		
MIS485	**MANAGING TECHNOLOGY**	**4**
MIS480	**ADVANCED DATABASE**	**4**

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