

PLSQL

Álvaro González Sotillo

3 de abril de 2018

Índice

1. Introducción	1
2. Bloques anónimos	2
3. Variables	2
4. Control de flujo	3
5. Sentencias SQL en PLSQL	4
6. Funciones y procedimientos	5
7. Control de errores	7
8. Disparadores (<i>triggers</i>)	8
9. Referencias	11

1. Introducción

- PLSQL es un lenguaje de programación ejecutado en los servidores Oracle
 - Con acceso a todas las sentencias SQL
 - Incluye variables, funciones, control de flujo...
- PLSQL es un lenguaje imperativo
 - Completa a SQL, que es un lenguaje declarativo

1.1. Palabras reservadas

- Vista **V\$RESERVED_WORDS**
- Definen estructuras de programa
- No pueden ser usados como identificadores

1.2. Identificadores

- Nombres definidos por el programador
 - No puede ser una palabra reservada
 - Constante, variable, excepción, paquete, función, procedimiento, tabla, cursor...
 - Hasta 30 caracteres
 - Comienza por una letra.
 - Puede contener \$, #, pero no puede contener operadores + % = / *

2. Bloques anónimos

```
select * from pepe where nombre='a';  
  
SET SERVEROUTPUT ON;  
begin  
    dbms_output.put_line('Hola');  
END;  
/
```

Listing 1: Bloque anónimo

3. Variables

- Valores referenciados por un identificador
- Deben declararse al principio de los bloques

```
SET SERVEROUTPUT ON;  
  
DECLARE  
    msg varchar(255);  
BEGIN  
    msg := 'Hola';  
    dbms_output.put_line(msg);  
END;  
/
```

3.1. Tipos de variable

- Se pueden utilizar todos los tipos SQL
 - char, varchar
 - number, integer, float
 - date, timestamp
 - blob, clob
- Tipos propios de PLSQL
 - bool
 - pls_integer

3.2. Tipos referidos

- %type : Tipo de un campo de una tabla
- %rowtype : Tipo compuesto, referido a una fila de una tabla

```
create table cliente( id integer, nombre varchar(255) );

DECLARE
    filacliente cliente%rowtype;
BEGIN
    filacliente.id := 1;
    filacliente.nombre := 'María';
    insert into cliente values filacliente;
END;
/
```

4. Control de flujo

4.1. Condicional

```
DECLARE
    numero integer := 1;
BEGIN
    if( numero < 0 ) then
        dbms_output.put_line( "Menor que cero");
    elsif( numero > 0 ) then
        dbms_output.put_line( "Mayor que cero");
    else
        dbms_output.put_line( "Igual que cero");
    end if;
END;
/
```

4.2. Condicional múltiple (I)

```
case
    when vsalario<0 then
        dbms_output.put_line('Incorrecto');
    when vsalario=0 THEN
        dbms_output.put_line('Gratis!');
    when vsalario<10000 then
        dbms_output.put_line('Salado!');
    when vsalario<90000 then
        dbms_output.put_line('Mas o menos');
    else
        dbms_output.put_line('Correcto');
end case;
```

4.3. Condicional múltiple (II)

```
case v_job_grade
    when 1 THEN
        dbms_output.put_line('Jefe!');
    when 2 then
        dbms_output.put_line('Jefecito');
    when 3 then
        dbms_output.put_line('Empleado regular');
    ELSE
        dbms_output.put_line('CEO');
end case;
```

4.4. Bucle loop

```
LOOP
  -- Instrucciones
  IF (expresion) THEN
    -- Instrucciones
  EXIT;
END IF;
END LOOP;
```

4.5. Bucle while

```
WHILE (expresion) LOOP
  -- Instrucciones
END LOOP;
```

4.6. Bucle for

```
DECLARE
  c PLS_INTEGER DEFAULT 0;
BEGIN
  FOR c IN REVERSE 1..10 LOOP
    dbms_output.put_line ('Contador = '||c);
  END LOOP;
END;
```

5. Sentencias SQL en PLSQL

5.1. Variables en select

```
create table empleados( empno number(20), salario number(8,2), nombre varchar(255));
DECLARE
  vsalario NUMBER;
BEGIN
  SELECT salario INTO vsalario FROM empleados WHERE empno=7369;
  dbms_output.put_line('El empleado numero 7369 tiene un vsalario de '||vsalario||' ');
end;
/
```

5.2. Variables en insert, update, delete

- Se utilizan como un valor inmediato

```
declare
  vempno number;
begin
  vempno := 100;
  insert into empleados(empno, salario, nombre)
    values( vempno, 1000, 'Juan');
  update empleados
    set salario = salario + 100
    where empno = vempno;
  delete from empleados where empno = vempno;
end;
/
```

5.3. Recorrer consultas

```
DECLARE
  c empleados %ROWTYPE;
  salariototal number;
  numeroempleados number;
  mediasalario number;
begin
  numeroempleados := 0;
  for c in (select * from empleados) loop
    dbms_output.put_line(c.nombre);
    numeroempleados := numeroempleados + 1;
    salariototal := salariototal + c.salario;
  end loop;
  mediasalario := salariototal / numeroempleados;
end;
/
```

6. Funciones y procedimientos

- Son bloques de código identificados con un nombre
- Pueden invocarse desde otros bloques de código
- En la invocación, se utilizan parámetros
 - De entrada
 - De salida

6.1. Funciones

- Las funciones devuelven **siempre** un valor
- Pueden recibir parámetros
- Por convenio:
 - El resultado de una función solo depende de sus parámetros
 - Una función no cambia la base de datos

```
CREATE OR REPLACE FUNCTION es_par(numero IN number)
RETURN boolean
IS
  resto number;
BEGIN
  resto := mod(numero,2);
  if( resto = 0 ) then
    return true;
  else
    return false;
  end if;
END;
/
```

6.1.1. Funciones en SQL

- Una función puede utilizarse en SQL

```
select empno, es_par(empno) from empleados;
```

6.1.2. Funciones predefinidas

replace	sysdate	lpad	instr
substr	nvl	trim	trunc
upper	to_date	mod	length
lower	to_char	decode	
rpadd	to_number		

6.2. Procedimientos

- Los procedimientos no devuelven un valor
 - Pero pueden tener parámetros out

```
CREATE OR REPLACE PROCEDURE aumenta_salario(vempno IN number)
IS
BEGIN
    update empleados
    set salario=salario+100
    where empno = vempno;
END;
/
```

6.3. Parámetros in

- Es el tipo de parámetros por defecto
- Un parámetro in se pasa *por valor*
- Se copia el valor introducido en el parámetro
- Un cambio del parámetro no afecta al bloque llamante

```
create or replace procedure suma_uno(n in numeric) is
begin
    n := n +1;
end;
/

declare
    numero numeric(10,0);
begin
    numero := 3;
    sumauno(numero);
    dbms_output.put_line(numero);
end;
/
```

6.4. Parámetros out

- Un parámetro out se pasa *por referencia*
- Un cambio del parámetro afecta al bloque llamante

```

create or replace procedure suma_uno(n in out numeric) is
begin
  n := n + 1;
end;
/

declare
  numero numeric(10,0);
begin
  numero := 3;
  sumauno(numero);
  dbms_output.put_line(numero);
end;
/

```

7. Control de errores

- Si se produce un error, se lanza una **excepción**
 - Se interrumpe el flujo de programa
 - Hasta que se **atrapa**
 - Puede atraparse en cada bloque/funcion/procedimiento

```

DECLARE
  -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;

```

7.1. Sección *exception*

- Se especifican varios tipos de excepción que se esperan

```

DECLARE
  -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
  WHEN ZERO_DIVIDE THEN
    -- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE
  WHEN OTHERS THEN
    -- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
    -- en los bloques anteriores
END;

```

7.2. Excepciones predefinidas

- Estas son algunas (hay muchas)

NO_DATA_FOUND	TOO_MANY_ROWS	ACCESS_INTO_NULL
INVALID_NUMBER	NO_DATA_FOUND	VALUE_ERROR
ROWTYPE_MISMATCH	ZERO_DIVIDE	

7.3. SQLCODE y SQLERRM

- Funciones predefinidas
- SQLCODE: Número de error (independiente del idioma)
- SQLERRM:
 - Sin parámetros: Mensaje de error en el idioma de la base de datos
 - Con un parámetro: mensaje de ese sqlcode

```
DECLARE
    result NUMBER;
BEGIN
    SELECT 1/0 INTO result FROM DUAL;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.put_line('Error: ' || TO_CHAR(SQLCODE));
        DBMS_OUTPUT.put_line(SQLERRM);
END;
```

7.4. Excepciones de usuario

- En ocasiones queremos enviar un mensaje de error personalizado
- Están disponibles los números de error entre 20001 y 20999
- Se pueden atrapar con `when others` y comprobarse con `SQLCODE`

```
DECLARE
    n number;
BEGIN
    SELECT count(*) into n from empleados
    if( n < 10 ) then
        RAISE_APPLICATION_ERROR(-20001,'La empresa necesita al menos 10 empleados');
    end if;
EXCEPTION
    WHEN OTHERS THEN
        if( sqlcode = -20001 ) then
            dbms_output.put_line('Pocos empleados');
        end if;
END;
```

8. Disparadores (*triggers*)

- Las funciones y procedimientos se invocan desde *fuera* de la base de datos
- Los disparadores los lanza la propia base de datos en respuesta a eventos
- Cada tabla tiene sus propios eventos
- Los disparadores se pueden lanzar *antes* o *después* del evento
- Los disparadores se pueden lanzar una vez por cada fila afectada, o una vez para toda la sentencia SQL

```

CREATE or replace TRIGGER personal_minimo
BEFORE DELETE ON empleados
declare
    n number;
begin
    SELECT count(*) into n from empleados;
    if( n < 10 ) then
        RAISE_APPLICATION_ERROR(-20001,'La empresa necesita al menos 10 empleados');
    end if;
end;
/

```

8.1. Eventos

Evento DML

delete	Borrado de una fila
insert	Insertción de una fila
update	Modificación de fila
update of	Modificación de un campo de una fila

Evento DDL/sistema

ALTER	Modificación de objetos
ANALYSE	
ASSOCIATE STATISTICS	
AUDIT	
COMMENT	Creación de objetos
CREATE	
DDL	
DISASSOCIATE STATISTICS	
DROP	Borrado de Objetos
GRANT	
LOGON	
LOGOFF	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	
SERVERERROR	
STARTUP	
SHUTDOWN	
SUSPEND	

- Se pueden combinar para un mismo *trigger*
 - Las funciones INSERTING, UPDATING y DELETING sirven para diferenciar por qué se ha lanzado

```

CREATE or replace TRIGGER ejemplo_or
BEFORE DELETE OR UPDATE OR INSERT ON empleados
begin
    case

```

```

when inserting THEN
    dbms_output.put_line('Insertando empleados');
when updating then
    dbms_output.put_line('Actualizando empleados');
when deleting then
    dbms_output.put_line('Borrando empleados');
else
    dbms_output.put_line('Inesperado');
end case;
end;
/

```

8.2. for each row

- Por defecto, un *trigger* se lanza una vez por cada sentencia SQL que provoque cambios
- Si se especifica `for each row`, se lanza una vez por cada fila cambiada

8.3. Momentos del evento

- Se puede lanzar
 - before
 - after
 - instead of: No se ejecuta el SQL, sino otro alternativo. Útil para vistas modificables.
- Las variables `:old` y `:new` existen en los *triggers* tipo `for each row`
 - `:old`: Variable tipo `%rowtype` con los datos antiguos de la fila
 - `:new`: Datos nuevos de la fila

Momento	Evento	:old	:new
before	delete	Lectura	
before	insert		Lectura/escritura
before	update	Lectura	Lectura/escritura
after	delete	Lectura	
after	insert		Lectura
after	update	Lectura	Lectura

8.4. Ejemplo típico: auditoría

```

alter table empleados add (
    createdby varchar(255),
    createddate timestamp,
    modifiedby varchar(255),
    modifieddate timestamp
);

```

```

create or replace trigger auditoria_creacion_empleados
before insert
on empleados
for each row
begin
    :new.createdby := username;

```

```
:new.createddate := systimestamp;  
end;  
/
```

8.5. Sintaxis (casi) completa

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>  
  {BEFORE|AFTER}  
  {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]  
  [OR {DELETE|INSERT|UPDATE [OF col1, col2, ..., colN]...}]  
  ON <nombre_tabla>  
  [FOR EACH ROW [WHEN (<condicion>)]]  
DECLARE  
  -- variables locales  
BEGIN  
  -- Sentencias  
[EXCEPTION]  
  -- Sentencias control de excepcion  
END;
```

9. Referencias

- Formatos:
 - [Transparencias](#)
 - [PDF](#)
- Creado con:
 - [Emacs](#)
 - [org-reveal](#)
 - [Latex](#)