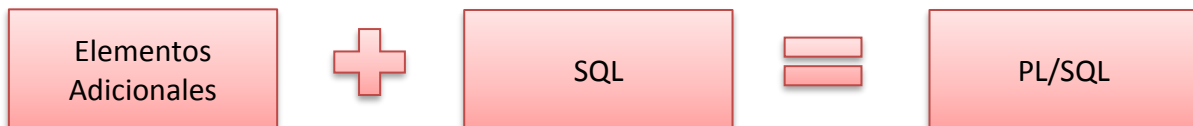


TEMA 10.
INTRODUCCIÓN A LA PROGRAMACIÓN PL/SQL

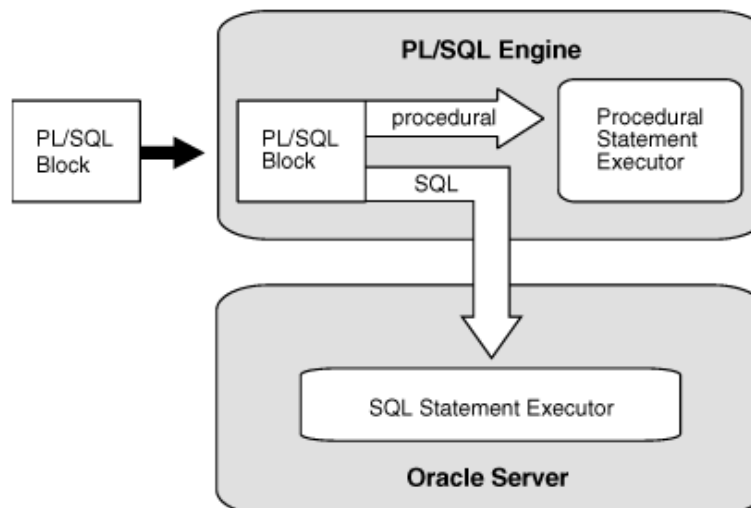
10.1. LENGUAJE DE PROGRAMACIÓN PL/SQL

- El lenguaje que se emplea para programar varía de un Sistema Manejador de Bases de Datos Relacional (RDBMS) a otro. No existe un estándar como en el caso del SQL.
- El lenguaje que utiliza Oracle se llama PL/SQL (*Procedural Language extension to SQL*) y es un lenguaje de programación que se usa para acceder y trabajar con bases de datos en Oracle desde distintos entornos.
- PL/SQL permite escribir código que se ejecuta en la base de datos justo donde se encuentran los datos, permitiendo un alto rendimiento en comparación con otros lenguajes que se ejecutan fuera de la base de datos.



- Principales elementos adicionales:
 - Variables, constantes, tipos de datos.
 - Estructuras de control: LOOPS, IF, ELSE, etc.
 - Unidades o piezas de código PL/SQL reutilizables (escritas 1 vez, ejecutadas N veces)
 - Características como encapsulamiento de datos, manejo de excepciones, ocultamiento de información, programación orientada a objetos, etc.
- Cada programa PL/SQL puede contener uno o más bloques de código que pueden estar anidados.

10.2. ARQUITECTURA PL/SQL



- Código PL/SQL es ejecutado en un “PL/SQL engine” a través de un “executor”.
- Este “engine” es implementado a través de una “Máquina virtual” encargado de procesar las instrucciones del código PL/SQL.
- Sentencias SQL son enviadas al servidor Oracle para ser ejecutadas por su propio “executor”.
- El “executor” PL/SQL espera el resultado de una sentencia SQL antes de continuar con la ejecución de las demás instrucciones PL/SQL.

10.2.1. Ventajas de la arquitectura

- Integración de estructuras procedurales con SQL
- Desempeño.
- En cuanto a programación:
 - Desarrollo modularizado.
 - Manejo adecuado de errores.
 - Portabilidad.
 - Integración con herramientas Oracle.

10.3. ELEMENTOS DE UN PROGRAMA PL/SQL

```
--sección declarativa (opcional)
[declare]

--ordenes sql y pl/sql (obligatorio)
begin
  <instrucciones pl/sql>

--excepciones (opcional)
[exception]

--fin de bloque (obligatorio)
end;
/
```

- En la sintaxis anterior se muestran 3 bloques: `declare`, `begin` y `exception`. Cada bloque contiene instrucciones.
- Observar que `end` termina con “;” que indica el fin del bloque `begin`.
- El carácter “/” provoca que el bloque de código sea ejecutado.
- Todas las sentencias que se encuentran dentro de un bloque, declaraciones y las instrucciones que indican el fin de un bloque deben terminar con “;”
- El siguiente ejemplo muestra el programa PL/SQL más simple que puede escribirse:


```
begin
  null;
end;
/
```
- Dentro del cuerpo del bloque (posterior a la instrucción `begin`) debe existir al menos una instrucción.

10.3.1. Tipos de bloques PL/SQL

- Bloques anónimos
- Procedimientos
- Funciones.

<pre>[DECLARE] BEGIN --statements [EXCEPTION] END;</pre>	<pre>PROCEDURE name IS BEGIN --statements [EXCEPTION] END;</pre>	<pre>FUNCTION name RETURN datatype IS BEGIN --statements RETURN value; [EXCEPTION] END;</pre>
-----------------------------------------------------------------	-------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------

10.3.1.1. Bloques anónimos

- No contienen un nombre.
- No son almacenados en la BD.
- Al no ser almacenados, se compilan y se ejecutan cada vez que se requieran durante la ejecución de alguna aplicación.

10.3.1.2. Procedimientos.

- Son objetos con nombre almacenados en la BD.
- No requieren ser re- compilados cada vez que se deseen ejecutar.

10.3.1.3. Funciones.

- Similares a un procedimiento con la diferencia que una función regresa un valor de un determinado tipo de dato.

¿Qué se puede construir con bloques PL/SQL?

- Bloques anónimos
 - Embebidos en una aplicación
- Procedimientos almacenados.
 - Almacenados en la BD, se invocan de manera repetida por nombre, aceptan parámetros.
- Paquetes PL/SQL
 - Modulos que agrupan a un conjunto de procedimientos y/o funciones.
- Triggers
 - Asociados a una tabla y se ejecutan al ocurrir un evento.

- Objetos personalizados
 - Definidos por el usuario y almacenados en la BD.

10.3.2. *Hola Mundo con PL/SQL*

Crear un bloque PL/SQL anónimo que imprima el mensaje “Hola mundo!” y la fecha actual del sistema. Ejecutarlo en diferentes clientes (SQL *Plus y Sql Developer).

```
set serveroutput on
begin
    dbms_output.put_line('Hola mundo!, fecha actual: '|| sysdate);
end;
/
```

SERVEROUTPUT es una variable de ambiente de SQL *Plus que habilita la salida de mensajes a consola a través del uso de un buffer. De no activarse, los mensajes no se imprimirán en la consola.

10.4. VARIABLES, ASIGNACIONES Y OPERADORES.

- Los tipos de datos que se emplean para declarar variables en un programa PL/SQL están integrados por los siguientes grupos:
 - Todos los tipos de datos SQL vistos en temas anteriores.
 - Tipos de datos particulares o específicos de PL/SQL
- El alcance de una variable es local. Esto significa que una variable será visible únicamente en el bloque PL/SQL donde fue declarada.

Sintaxis:

```
<identifier> [constant] <datatype> [not null] [:= | default <expression>];
```

Ejemplo:

```
set serveroutput on
declare
    v_uno number;
    v_dos number := 2;
    v_tres number not null := 3;
    v_cuatro number default 4;
    v_str varchar2(50);
begin
    v_uno := 1;
    v_str := v_uno||','||v_dos||','||v_tres||','||v_cuatro;
    dbms_output.put_line(v_str);
end;
/
```

- Observar el operador := empleado para asignar un valor inicial a las variables.
- Es posible emplear not null para garantizar que la variable no puede tener valores nulos (requiere asignar valor inicial).
- De forma equivalente se puede emplear default para inicializar una variable

10.4.1. *Otras formas de declarar variables:*

El atributo %type permite hacer referencia a una columna de una tabla o una variable que se haya definido anteriormente. Su sintaxis es:

```
nombreVariable  tabla.columna%type;
```

Ejemplos :

```
v_nombre empleado.nombre%type;
v_balance number;
```

Es posible hacer referencia a un renglón de una tabla o cursor, con el fin de crear variables que tengan que permitan acceder a todos los atributos o columnas de dicho renglón. para ello utilice emplea %rowtype. Su sintaxis es:

```
nombrevariable {tabla | cursor}%rowtype;
```

Ejemplo:

Crear un Script PL/SQL que muestre en consola los datos del plan de estudios con id =1

```
declare
  v_id plan_estudios.plan_estudios_id%type;
  v_clave plan_estudios.clave%type;
  v_fecha plan_estudios.fecha_inicio%type;

begin
  select plan_estudios_id,clave,fecha_inicio
  into v_id,v_clave,v_fecha
  from plan_estudios
  where plan_estudios_id=1;

  dbms_output.put_line('id:      '||v_id);
  dbms_output.put_line('clave:  '||v_clave);
  dbms_output.put_line('fecha:  '||v_fecha);
end;
/
```

- Observar el tipo de dato asignado a cada variable. Se hace corresponder con el tipo de dato asignado en la tabla. Se recomienda esta técnica de ser posible para evitar conflicto de tipos de dato.
- Observar el uso de la instrucción **into**. En PL/SQL el resultado de una instrucción **select** **debe** ser asignado a una variable, o de lo contrario el programa no compilará. En este caso, se obtienen 3 columnas, se emplean las variables v_id, v_clave y v_fecha para almacenar los valores correspondientes.
- Observar que esta sentencia SQL solo regresa un registro. Esto es un requisito ya que las variables antes mencionadas solo pueden referenciar un solo valor a la vez.
- Otra restricción es que la sentencia no debe regresar 0 registros. Esto se debe a que las variables deben ser inicializadas.

- Para situaciones en las que la sentencia `select` obtiene 0 registros o más de un registro, la técnica cambia. Se hace uso de un **cursor** en lugar de la instrucción `into`. Este concepto se revisa más adelante.

Convenciones para el nombrado de variables:

Estructura PL/SQL	Convención
Variable	v_variable_name
Constante	c_constant_name
Parámetro en un subprograma	p_param_name
Cursor	cur_cursor_name
Excepción	e_exception_type

10.4.2. Operadores.

Los operadores que emplea un programa PL/SQL comprende a los operadores clásicos que se encuentran en la mayoría de los lenguajes: operadores aritméticos, operadores lógicos, etc. Existen algunas diferencias particulares que se muestran a continuación:

- El operador `=` se emplea para hacer comparaciones (no se emplea el operador `==`)
- El operador de negación puede ser cualquiera de los siguientes: `!=`, `<>`, `~=`, `^=`
- El operador para concatenar cadenas es `||`

10.4.3. Estructuras de control

Sentencia IF-THEN

```
if condición then
    secuencia_de_instrucciones
end if;
```

Sentencia IF-THEN-ELSE

```
if condición then
    secuencia_de_instrucciones1
else
    secuencia_de_instrucciones2
end if;
```

Sentencia IF-THEN-ELSIF

```
if condición then
    secuencia_de_instrucciones1
elsif condición2 then
    secuencia_de_instrucciones2
else
    secuencia_de_instrucciones3
end if;
```

Sentencia CASE

Al igual que IF, la sentencia CASE selecciona una secuencia de sentencias a ejecutar. Existen 2 estilos:

- La instrucción case contiene un valor escalar (simple case)
- La sentencia case no contiene expresión o valor, se emplean las expresiones booleanas definidas en la instrucción when.

Sintaxis:

```
case [selector]
  when expresión1 then
    secuencia_de_instrucciones1;
  when expresión2 then
    secuencia_de_instrucciones2;
    ...
  when expresiónn then
    secuencia_de_instruccionesn;

  [else secuencia_de_instrucciones n+1];
end case;
```

La condición else es opcional y sólo se aplica si ninguna de las condiciones when anteriores se ejecuta.

Ejemplo:

- Estilo 1: Uso de un escalar.

```
declare
  v_suerte number := 3;
begin

  case v_suerte
    when 1 then
      dbms_output.put_line('El numero de la suerte es UNO');
    when 2 then
      dbms_output.put_line('El numero de la suerte es DOS');
    when 3 then
      dbms_output.put_line('El numero de la suerte es TRES');
    else
      dbms_output.put_line('Numero de la suerte invalido');
  end case;
end;
/
```

- Estilo 2: Uso de una expresión booleana en la instrucción when.

```

declare
  v_suerte number := 3;
begin
  case
    when v_suerte = 1 then
      dbms_output.put_line('El numero de la suerte es UNO');
    when v_suerte = 2 then
      dbms_output.put_line('El numero de la suerte es DOS');
    when v_suerte = 3 then
      dbms_output.put_line('El numero de la suerte es TRES');
    else
      dbms_output.put_line('Numero de la suerte invalido');
    end case;
end;
/

```

10.4.4. Estructuras de Iteración.

PL/SQL soporta 3 estructuras de iteración:

- For Loops
- Simple Loops
- While Loops

10.4.4.1. For Loops

- Generalmente un For loop se emplea para recorrer cursores, pero también se pueden emplear para otros propósitos empleado un For numérico (iteración por rango de valores).

Sintaxis For numérico:

```

for i in starting_number .. ending_number loop
  <statements>
end loop;

```

Ejemplo:

```

begin
  for v_index in 1 .. 10 loop
    dbms_output.put_line('Iterando : '||v_index);
    if v_index >= 5 then
      dbms_output.put_line('terminando prematuramente');
      exit;
    end if;
    if v_index = 2 then
      dbms_output.put_line('El siguiente numero sera el 3');
      continue;
    end if;
  end loop;
end;
/

```

- Observar el uso de `exit` y `continue`. Corresponden con las instrucciones `break` y `continue` que comúnmente se emplean en otros lenguajes de programación.

El uso de Simple Loops y su uso con cursores se explica más adelante.

10.4.4.2. While Loops.

Sintaxis:

```
while <condition> loop
    statement 1;
    statement 2;
    ...
end loop;
```

Como se puede observar, su estructura es muy similar al clásico `while` encontrado en otros lenguajes. Puede ser empleado con cursores, pero en la práctica se prefiere el uso de un `for loop`.

Ejemplo:

```
declare
    v_index number := 1;
begin
    while v_index <= 10 loop
        dbms_output.put_line(v_index);
        v_index := v_index+1;
    end loop;
end;
/
```

10.5. CURSORES

Hasta este momento, en todos los ejemplos PL/SQL la instrucción `select` no devuelve más de un registro. Si deseamos obtener más de un valor se requiere hacer uso de un **cursor** explícito para extraer individualmente cada fila.

Un cursor es un área de memoria privada utilizada para realizar operaciones con los registros devueltos tras ejecutar una sentencia `select`. Existen dos tipos:

- Implícitos
 - Creados y administrados internamente por el manejador para procesar sentencias SQL
- Explícitos
 - Declarados explícitamente por el programador.

Su sintaxis es la siguiente:

```
cursor <cursor_name> is <select_statement>
```

El cursor explícito se abre mediante la siguiente sintaxis:

```
open <cursor_name>;
```

La extracción de los datos y su almacenamiento en variables PL/SQL se realiza utilizando la siguiente sintaxis:

```
fetch <cursor_name> into listavariables;
fetch <cursor_name> into registro1/sql;
```

Importante: ¡Al terminar de usar un cursor se debe cerrar!

```
close <cursor_name>;
```

Un cursor define 6 atributos que se emplean para controlar el acceso a los datos, en especial los siguientes 4 atributos: `%found` y `%notfound` para controlar si la última orden `fetch` devolvió o no una fila, `%isopen` para saber si el cursor está abierto; y `%rowcount` que devuelve el número de filas extraídas por el cursor.

Ejemplo:

Construir un script que imprima el nombre, apellidos, nombre de la asignatura y el número de cursos que imparte cada profesor de la universidad.

```
declare
  --declaración del cursor
  cursor cur_datos_profesor is
    select p.nombre,p.apellido_paterno, p.apellido_materno,
           a.nombre, count(*) cursos
    from profesor p, curso c, asignatura a
   where p.profesor_id=c.profesor_id
   and c.asignatura_id = a.asignatura_id
   group by p.nombre,p.apellido_paterno,p.apellido_materno,a.nombre;

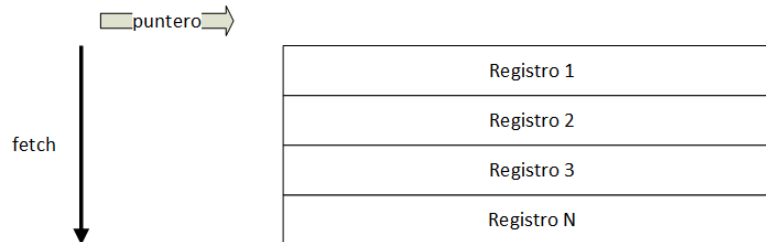
  --declaración de variables
  v_nombre profesor.nombre%type;
  v_ap_pat profesor.apellido_paterno%type;
  v_ap_mat profesor.apellido_materno%type;
  v_asignatura asignatura.nombre%type;
  v_num_cursos number;

begin
  open cur_datos_profesor;
  dbms_output.put_line('resultados obtenidos');
  dbms_output.put_line(
    'nombre apellido paterno apellido materno asignatura #cursos');
  loop
    fetch cur_datos_profesor into
      v_nombre,v_ap_pat,v_ap_mat,v_asignatura,v_num_cursos;
    exit when cur_datos_profesor%notfound;
    dbms_output.put_line(
      v_nombre||' ' , '||v_ap_pat||' ' , '||v_ap_mat
      ||' ' , '||v_asignatura||' ' , '||v_num_cursos);
  end loop;
  --Importante: Cerrar el cursor al terminar para liberar recursos.
  close cur_datos_profesor;
end;
/
```

- Observar la declaración del cursor, ya no se hace uso de la instrucción `into`.
- Antes de acceder a los datos del cursor, este se debe abrir.
- Para iterar un cursor se hace uso de un Simple Loop.
- Al terminar el ciclo, el cursor debe cerrarse.

10.5.1. Simple Loop para cursores.

- Un Simple loop requiere que el programador controle el punto de inicio y el punto en el que el loop debe terminar. De no realizarse de forma correcta, se pueden provocar loop infinitos entre otros errores.
- En el ejemplo anterior, observar que únicamente se especifica la instrucción `loop` sin especificar alguna otra configuración para controlar su inicio y su fin. Esto implica un ciclo infinito, que debe ser roto en algún momento, al cumplirse cierta condición.
- La instrucción `fetch` provoca que el puntero asociado al cursor se mueva al siguiente renglón obtenido. Un cursor puede verse como una lista de renglones y un puntero que se va moviendo de arriba hacia abajo para recorrerlo:



- Observar que la siguiente línea del código es la instrucción `exit`. Si el puntero ha llegado a su fin, el valor del atributo `%notfound` será `true`, y por lo tanto el ciclo termina. Es importante validar esta condición antes de acceder a los valores de cada registro. De esta forma, la instrucción `exit` permite controlar el momento en el que el ciclo debe terminar.

10.5.2. For loops para cursores.

- Una alternativa para simplificar el manejo de cursores es mediante el uso de un For loop. El siguiente código es equivalente al anterior con diversas mejoras:

```

set serveroutput on
declare
    --declaración del cursor
    cursor cur_datos_profesor is
    select p.nombre as nombre_profesor, p.apellido_paterno, p.apellido_materno,
        a.nombre as nombre_asignatura, count(*) cursos
    from profesor p, curso c, asignatura a
    where p.profesor_id=c.profesor_id
    and c.asignatura_id = a.asignatura_id
    group by p.nombre, p.apellido_paterno, p.apellido_materno, a.nombre;

begin
    dbms_output.put_line('resultados obtenidos');
    dbms_output.put_line(
        'nombre apellido paterno apellido materno asignatura #cursos');
    for p in cur_datos_profesor loop
        dbms_output.put_line(
            p.nombre_profesor||' ' , '||p.apellido_paterno
            ||' ' , '||p.apellido_materno
            ||' ' , '||p.nombre_asignatura||' ' , '||p.cursos);
    end loop;
end;
/

```

- Observar que ya no fue necesario declarar variables.
- Observar el uso de un For Loop. La variable 'p' representa a cada uno de los renglones que obtiene el cursor. Por lo tanto, para acceder a los valores de las columnas se puede emplear la sintaxis `p.<nombre_columna>` como se muestra en el código. Esto permite eliminar la necesidad de declarar variables para cada columna y el uso de la instrucción `fetch`. El For loop se encarga de ello.
- Sin embargo, se debe tener cuidado con los nombres de las columnas. En este ejemplo existen 2 columnas llamadas 'nombre' que corresponden al nombre del profesor y al nombre de la asignatura. En este caso se debe especificar un alias a cada columna para evitar ambigüedades (observar los alias marcados en negritas).
- Finalmente, observar que ya no fue necesario abrir y cerrar el cursor. Este proceso también es manejado por el For Loop.

10.6. TRIGGERS

- Un trigger (disparador) es un programa PL/SQL que se almacena en la base de datos. Su principal diferencia con respecto a otros programas como procedimientos o funciones, es que un trigger no puede ser invocado por el programador o usuario, más bien, los triggers se ejecutan cuando ocurre un evento en la base de datos.
- El usuario que desee crear un trigger deberá contar con el privilegio `create trigger`.

10.6.1. Tipos de triggers

Oracle soporta 5 tipos de triggers.

10.6.1.1. DDL Triggers

Es posible ejecutar un trigger al crear, cambiar o eliminar objetos en un esquema. Por ejemplo, tablas, etc. Pueden ser útiles para monitorear o controlar las sentencias DDL que se ejecutan en la base de datos.

10.6.1.2. DML Triggers.

Representan el tipo de trigger más común y usado. El trigger se ejecuta cuando se realiza una operación `insert`, `update` o `delete` sobre un dato que pertenece a una tabla. El trigger se asocia a la tabla (no a los datos). En esta categoría los triggers pueden ser útiles para:

- Auditoría (monitorear y controlar quién y cuándo se modifica un dato).
- Respaldo de datos antes que estos sean modificados o eliminados.
- Asignación automática de valores para llaves primarias.
- Implementar reglas de negocio o restricciones complejas que deben ser validadas al ocurrir o aplicar un cambio a los datos.

Un DML trigger puede ser de 2 tipos:

- *Simple DML Trigger:*

El trigger se dispara cuando ocurre alguno de los siguientes eventos:

- Antes de ejecutar la sentencia DML que provoca la activación del trigger, conocido como ***before statement trigger*** o *statement-level before trigger*
- Después de ejecutar la sentencia DML que provoca la activación del trigger, conocido como ***after statement trigger*** o *statement-level after trigger*.
- Antes de aplicar la operación DML a cada uno de los registros afectados por la sentencia que provoca la activación del trigger, conocido como ***before each row trigger*** o *row-level before trigger*. Notar que el trigger se dispara por cada registro afectado.
- Después de aplicar la operación DML a cada uno de los registros afectados por la sentencia que provoca la activación del trigger, conocido como ***after each row trigger*** o *row-level after trigger*.

- ***Compound DML Trigger.***

A diferencia de un simple DML trigger, en este caso el trigger puede dispararse cuando ocurra uno, más, o inclusive en todos los eventos mencionados anteriormente. Algunos beneficios:

- Permite realizar acciones distintas en diferentes eventos.
- Permite compartir datos. Por ejemplo, las variables que hayan sido inicializadas en un evento pueden ser leídas cuando el trigger se ejecute en un evento posterior.

10.6.1.3. Instead-of triggers.

Permiten ‘redireccionar’ una operación `insert`, `update`, o `delete`. Es decir, cuando una sentencia DML se ejecuta, el trigger la intercepta y su código será el encargado de aplicar la operación o en general, lo que se haya programado.

- Este tipo de triggers se emplean, por ejemplo, en sentencias DML que actúan sobre vistas que no permiten operaciones DML (non-updatable views). En este caso, el trigger es el encargado de implementar la operación DML.
- Es importante mencionar que en un DML trigger no redirecciona la operación como en un `instead-of trigger`. Es decir, se ejecuta tanto el trigger DML como la operación DML. La única forma de evitar que se ejecute la operación es lanzando una excepción en el trigger.
- Otro uso de los `instead-of triggers` es la implementación de transparencia de operaciones DML en bases de datos distribuidas.

10.6.1.4. System or database event triggers.

- Ocurren al producirse un evento a nivel del sistema (DBMS), por ejemplo: cuando un usuario hace ‘login’ o ‘logout’, etc.

Para efectos del curso, únicamente se consideran los DML triggers los cuales se estudian a detalle en la siguiente sección.

10.6.2. DML triggers.

Adicional a las características mencionadas anteriormente:

- Un DML trigger siempre es asociado a una tabla.
- Cada tabla puede tener asociados varios triggers.
- Si existe una transacción en curso, el trigger también participará en ella.
- Para el caso de los **row-level** triggers, es posible acceder a los valores del registro afectado.







Sintaxis:

```
create [or replace] trigger <trigger_name>
{before | after}
{delete | insert | update} of <column_name1,...> on <table_name>
[for each row]
[declare]
  [<nombre_variable> <tipo_de_dato>[:= <valor_inicial>] ]
begin
  <instrucciones pl-sql>
end;
/
```

- Las instrucciones `before` y `after` definen el momento en el que se ejecuta el trigger.
 - `before`: El trigger se ejecuta antes de aplicar el cambio.
 - `after`: EL trigger se ejecuta posterior a aplicar el cambio.
- Si se omite `for each row`, el trigger se ejecuta una sola vez (statement level) y es el comportamiento por default. En caso contrario, se ejecuta una vez por cada registro creado, modificado o eliminado (row level).
- Dentro de los bloques `begin` y `end`, se escribe el código PL-SQL a ejecutar.

Ejemplo 1:

Suponer la siguiente tabla de artículos de una tienda.

ARTICULO		
 ARTICULO_ID	NUMBER(10,0)	NOT NULL
 CODIGO	VARCHAR2(6)	NOT NULL
 DESCRIPCION	VARCHAR2(500)	NOT NULL
 EXISTENCIAS	NUMBER(5,0)	NOT NULL
 EXISTENCIA_MINIMA	NUMBER(5,0)	NOT NULL
 REQUIERE_SURTIR	NUMBER(1,0)	NOT NULL

- Cuando el campo `existencias` se modifica, significa que un artículo se ha vendido. Se desea que, al momento de actualizar este campo, se revise el valor del campo `existencia_minima`. Si su valor es menor al valor del campo `existencias`, significa que el artículo se está agotando, por lo cual se debe solicitar la compra o el surtido de más artículos. Para ello el valor del campo `requiere_surtir` deberá estar en `true` (1).

Generar un Trigger que se encargue de actualizar el campo `existencia_minima` de forma automática.

```
create or replace trigger trg_articulo_existencia
after insert or update of existencias on articulo
begin
    update articulo
    set requiere_surtir = 1
    where existencias < existencia_minima;
end;
/
show errors
```

- En el código se observa que el trigger se va a activar al aplicar una operación insert o update sobre el campo existencias de la tabla articulo.
- Observar que se trata de un trigger tipo “statement level” ya que no se especificó for each row. Esto significa que el trigger se ejecuta una sola vez a pesar de que la instrucción update que provoque su activación modifique más de un registro.
- En el cuerpo del trigger, se actualiza el atributo requiere_surtir cuando el número de existencias sea demasiado bajo.
- Observar la instrucción show errors. No forma parte de la definición del trigger, pero permite mostrar los errores en consola en caso de existir problemas de compilación.

Ejemplo 2:

Suponer que la tabla contiene los siguientes datos.

ARTICULO_ID	CODIGO	DESCRIPCION	EXISTENCIAS	EXISTENCIA_MINIMA	REQUIERE_SURTIR
1	A9D28	JABON DE BAÑO	6	5	0

¿Qué pasa si se modifica el campo existencia_minima al valor 7?

- En este caso, el campo requiere_surtir debería actualizarse a 1 ya que existen solo 6 artículos y se requieren como mínimo 7
- El trigger anterior no resuelve este escenario ya que este solo se dispara cuando se aplica una operación sobre la columna existencias. El siguiente código muestra una solución

```
create or replace trigger trg_articulo_existencia
after insert or update of existencias, existencia_minima on articulo
begin
    update articulo
    set requiere_surtir = 1
    where existencias < existencia_minima;
end;
```

- En la definición del trigger se puede configurar más de una columna. Esto permite que el atributo requiere_surtir se actualice correctamente al modificar cualquiera de los 2 atributos.

Ejemplo 3:

Suponer que la tabla contiene los siguientes datos:

ARTICULO_ID	CODIGO	DESCRIPCION	EXISTENCIAS	EXISTENCIA_MINIMA	REQUIERE_SURTIR
1	A9D28	JABON DE BAÑO	2	5	1

¿Qué pasa si se actualiza el valor del campo `existencias` A 20?

¿Qué pasa si la tabla tiene un número grande registros? 200,000, etc.

- En este caso, el campo `requiere_surtir` tendría que actualizarse a 0 ya que existen suficientes artículos en la tienda. El trigger actual no actualiza a este valor. Una solución es agregar una sentencia `update` para este escenario:

```
update articulo
set requiere_surtir = 1
where existencias < existencia_minima;
```

```
update articulo
set requiere_surtir = 0
where existencias >= existencia_minima;
```

- Sin embargo, si la tabla tiene una gran cantidad de registros, la solución propuesta se vuelve ineficiente. El manejador tendrá que comparar todos los valores del campo `existencias` 2 veces para poder actualizar el campo `requiere_surtir`. Un índice podría ayudar, sin embargo, sería más adecuado hacer uso de un trigger tipo “Row level” para actualizar solo a aquellos registros que así lo requieran:

```
create or replace trigger trg_articulo_existencia
before insert or update of existencias, existencia_minima on articulo
for each row
begin
    if :new.existencias < :new.existencia_minima then
        :new.requiere_surtir := 1;
    else
        :new.requiere_surtir := 0;
    end if;
end;
```

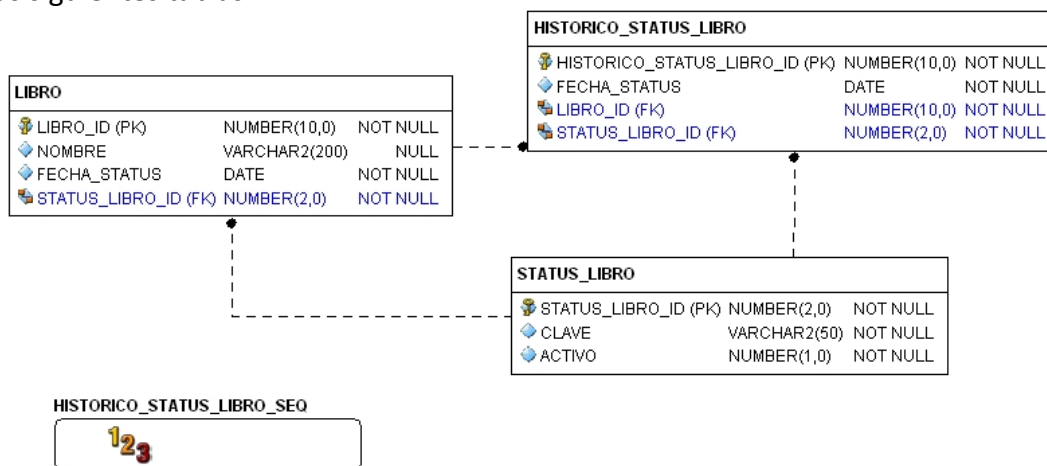
- Observar que se emplea la instrucción `for each row` (row level trigger). Esto permite al trigger procesar únicamente a los registros que sufrieron cambios. Las 2 sentencias `update` mostradas anteriormente ya no serán necesarias, lo que permite mejorar el desempeño.
- Observar que se hace uso de 2 nuevas variables `:new` y `:old` llamadas **pseudorecords**.
- Existe un tercer pseudorecord llamado `:parent` el cual se emplea únicamente en tablas anidadas (nested).
- A nivel general, estas 2 variables hacen referencia a el registro que se está actualizando, insertando o eliminando. Sus valores dependen de la operación DML y también dependen del momento en el que el trigger se dispara: `after` o `before`. Esto se puede ilustrar en la siguiente tabla:

Variable	Before Insert	Before update	Before Delete
:new	Contiene los valores del nuevo registro que se va a insertar.	Contiene los nuevos valores del registro que se va a actualizar	NULL. Para una operación delete no existe un nuevo registro.
:old	NULL. En una operación Insert no existe registro anterior.	Contiene los valores viejos o anteriores del registro que se va a actualizar	Contiene los valores del registro que se va a eliminar.
Variable	After Insert	After update	After Delete
:new	Contiene los valores del registro que se acaba de insertar.	Contiene los valores nuevos del registro que se acaba de actualizar.	NULL. Para una operación delete no existe un nuevo registro.
:old	NULL. En una operación Insert no existe registro anterior.	Contiene los valores viejos o anteriores del registro que se acaba de actualizar.	Contiene los valores del registro que se acaba de eliminar.

- En resumen:
 - :old solo tiene valores para update y delete.
 - :new solo tiene valores para insert y update.
- En memoria el manejador almacena tanto el valor nuevo (:new) como el valor anterior(:old) del registro en turno. Dentro del código del trigger es posible hacer referencia a estos valores.
- Finalmente, observar que no se requiere lanzar una operación de update para modificar el nuevo valor del campo `requiere_surtir`, basta con asignarlo empleando :new
`:new.requiere_surtir := 1;`

Ejemplo 3:

Suponer las siguientes tablas:



- Generar un trigger, de tal forma que al crear o al modificar el status de un libro, se genere un registro en su histórico.

```
create or replace trigger hist_status_trigger
after insert or update of status_libro_id on libro
for each row
declare
v_status_id number(2,0);
v_fecha_status date;
v_hist_id number(10,0);
v_libro_id number(10,0);
begin
    -- obtiene el consecutivo de la secuencia
    select historico_status_libro_seq.nextval into v_hist_id from dual;

    --asigna valores a las variables con el nuevo status y fecha
    v_status_id := :new.status_libro_id;
    v_fecha_status := :new.fecha_status;
    v_libro_id := :new.libro_id;

    dbms_output.put_line('status anterior: '|| :old.status_libro_id);
    dbms_output.put_line('status nuevo: '|| :new.status_libro_id);

    dbms_output.put_line('insertando en historico, libro_id: '
        || v_libro_id ||', status_id: ' || v_status_id
        ||', fecha: '|| v_fecha_status||', hist_id: '||v_hist_id);

    -- inserta en el histórico

    insert into historico_status_libro
    (historico_status_libro_id,status_libro_id,fecha_status,libro_id)
    values(v_hist_id,v_status_id,v_fecha_status,v_libro_id);
end;
```

10.6.2.1. Predicados condicionales.

Permiten identificar en el código del trigger el evento particular que provocó su activación. Por ejemplo, determinar si el evento fue una operación insert, update, delete, e inclusive saber qué columna fue actualizada en el caso de una operación update.

Para identificar el evento se emplean los siguientes predicados condicionales:

- Inserting: El trigger fue activado por una operación de inserción.
- Updating: El trigger fue activado por una operación de actualización.
- Updating('<nombre_columna>') El trigger fue activado al actualizar una determinada columna.
- Deleting: El trigger fue activado por una operación de eliminación.

Ejemplo:

```

create or replace trigger t
before
  insert or
  update of nombre, status_libro_id or
  delete
on libro
begin
  case
    when inserting then
      dbms_output.put_line('insertando libro');
    when updating('nombre') then
      dbms_output.put_line('Actualizando el nombre');
    when updating('status_libro_id') then
      dbms_output.put_line('Actualizando status');
    when deleting then
      dbms_output.put_line('Eliminando');
  end case;
end;

```

10.6.2.2. DML Compound triggers.

Como se mencionó anteriormente este tipo de trigger permite obtener información o realizar acciones en 4 puntos diferentes: antes o después de ejecutar la sentencia DML, antes o después de aplicar la operación DML a cada registro afectado.

En este tipo de triggers existe un bloque de declaración que es común a todos los eventos.

Esta funcionalidad permite realizar validaciones como auditoría, verificar, guardar reemplazar valores antes que estos sean modificados. Existen 2 principales escenarios de uso:

- Para mejorar desempeño: Registros se acumulan en una tabla y después se hacen operaciones en por “bonches” o “conjuntos”.
- Para evitar un error clásico que puede ocurrir al trabajar con triggers: error de tablas mutantes (ORA-04091).

Sintaxis general de un compound trigger:

```

create [or replace] trigger <trigger_name>
for {insert | update | update of <column_name>[,<column_name>..] | delete}
on <table_name>
compound trigger

--declarative, common section

[before statement is
  [declaration_statement;]
begin
  begin_statement;
end before statement;
]

[before each row is
  [declaration_statement;]
begin
  begin_statement;
end before each row;
]

```

```

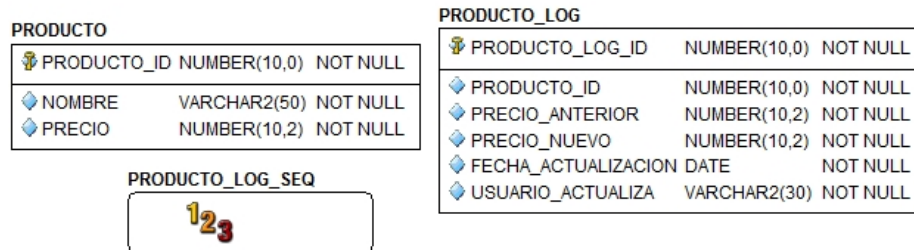
[after each row is
[declaration_statement;]
begin
    begin_statement;
end after each row;
]

[after statement is
[declaration_statement;]
begin
    begin_statement;
end after statement;
]
end [<trigger_name>];
/

```

Ejemplo:

Considerar el siguiente modelo relacional que contiene el catálogo de productos de una tienda de autoservicio.



- El gerente de la tienda desea controlar los cambios de precio de los productos. Para ello, se decide implementar un trigger que se encargue de guardar un “log” o registro por cada cambio que se detecte en la tabla `producto`.
- Por cada precio modificado se deberá insertar un nuevo registro en la tabla `producto_log` haciendo uso de la secuencia. Entre otras cosas, observar que se debe guardar el precio anterior y el precio nuevo, así como el usuario que realizó el cambio y la fecha de cambio.
- Se tiene un requerimiento adicional: El cambio de precio puede aplicarse a un número importante de registros. Por ejemplo, la siguiente sentencia SQL modifica incrementa el precio de todos los vinos en un 10%:

```

update producto
set precio =precio*1.1
where lower(nombre) like 'wine%';

```

- Esto implica que se actualice una cantidad importante de registros en el catálogo. Un DML trigger simple puede presentar algunos inconvenientes de desempeño ya que por cada registro se genera una sentencia `insert` en `producto_log`.
- Para agilizar este proceso, se decide emplear un compound trigger. La idea principal es que se hagan inserciones por bloque o bonche de registros lo que permitiría reducir los tiempos de procesamiento. El código se muestra a continuación.

```

create or replace trigger producto_precio_trigger
for update of precio on producto
compound trigger
--declaraciones globales y comunes
--declara un objeto type para guardar los valores
-- que se van a insertar en producto_log
type prod_actualizado_type is record (
    producto_log_id producto_log.producto_log_id%type,
    producto_id producto_log.producto_id%type,
    precio_anterior producto_log.precio_anterior%type,
    precio_nuevo producto_log.precio_nuevo%type,
    fecha_actualizacion producto_log.fecha_actualizacion%type,
    usuario_actualiza producto_log.usuario_actualiza%type
);

--Crea un objeto tipo collection para almacenar los productos
type precio_list_type is table of prod_actualizado_type;

--Crea una colección y la inicializa.
precio_list precio_list_type := precio_list_type();

--inicia la sección before each row
before each row is
    --declara variables que solo se usan en este bloque
    v_usuario varchar2(30) := sys_context('USERENV','SESSION_USER');
    v_fecha date := sysdate;
    v_index number;

begin
    --asigna espacio a la colección
    precio_list.extend;
    --obtiene el índice siguiente para guardar el objeto modificado
    v_index := precio_list.last;
    --guarda el nuevo registro cuyo precio ha cambiado.
    precio_list(v_index).producto_log_id := producto_log_seq.nextval;
    precio_list(v_index).producto_id := :new.producto_id;
    precio_list(v_index).precio_anterior := :old.precio;
    precio_list(v_index).precio_nuevo := :new.precio;
    precio_list(v_index).fecha_actualizacion := v_fecha;
    precio_list(v_index).usuario_actualiza := v_usuario;
end before each row;
--inicia after statement
--aquí se hacen las inserciones de forma eficiente
after statement is
begin
    forall i in precio_list.first .. precio_list.last
        insert into producto_log(producto_log_id,producto_id,
            precio_anterior,precio_nuevo,fecha_actualizacion,
            usuario_actualiza)
        values(precio_list(i).producto_log_id, precio_list(i).producto_id,
            precio_list(i).precio_anterior,precio_list(i).precio_nuevo,
            precio_list(i).fecha_actualizacion, precio_list(i).usuario_actualiza
        );
    end after statement;
end;
/
show errors;

```

- Observar que el trigger atiende 2 eventos:
 - Before each row (cada vez que se hace la actualización al precio de un producto)
 - After statement (posterior a que se termina de ejecutar la instrucción update).
- Observar que, en la sección de declaraciones globales, se agrega un código que contiene la declaración de una colección de objetos llamada `precio_list`. Para mayores detalles en cuanto al manejo y creación de colecciones, revisar la sección “Colecciones” en este mismo documento.
- El objetivo de esta lista es guardar todos los datos que se insertarán en `producto_log`. En el bloque `before each row` se puede observar la forma en la que se van agregando elementos a la lista en lugar de insertarlos directamente a la tabla `precio_log`.
- Esta estrategia permite reducir y mejorar el desempeño ya que todos los datos se guardan en memoria dentro de la colección.
- Una vez que se terminó de ejecutar el trigger en el bloque `before each row`, se ejecuta el bloque `after statement`. Observar que en este bloque se realiza la inserción masiva de registros que contiene la colección `precio_list`.
- Observar el uso de la instrucción `forall` encargada de iterar la colección y realizar las inserciones. Esta técnica permite mejorar el desempeño ya que se realizan en un solo bloque.
- En situaciones donde se tenga una gran cantidad de registros, para evitar el uso de grandes cantidades de memoria, se puede implementar una técnica de inserciones parciales o por partes. Por ejemplo, se puede establecer que cada 1000 registros se vacie la colección para evitar problemas de memoria.

Ejemplo:

A continuación, se presenta el código del trigger anterior, pero ahora con la siguiente variante:

- La colección de datos deberá vaciarse cuando tenga como máximo 10 registros.
- El código que se modificó/agregó se muestra resaltado.

```
create or replace trigger producto_precio_trigger
for update of precio on producto
compound trigger
--declaraciones globales y comunes

--declara un objeto type para guardar los valores
-- que se van a insertar en producto_log
type prod_actualizado_type is record (
  producto_log_id producto_log.producto_log_id%type,
  producto_id producto_log.producto_id%type,
  precio_anterior producto_log.precio_anterior%type,
  precio_nuevo producto_log.precio_nuevo%type,
  fecha_actualizacion producto_log.fecha_actualizacion%type,
  usuario_actualiza producto_log.usuario_actualiza%type
);

--Crea un objeto tipo collection par almacenar los productos
type precio_list_type is table of prod_actualizado_type;

--Crea la colección y la inicializa.
precio_list precio_list_type := precio_list_type();
```

```

--umbral empleado para vaciar la colección
v_umbral number := 10;

--se crea un procedimiento para vaciar la colección si alcanza
-- cierto número de elementos.
procedure vacia_coleccion is
    v_num_rows integer;
begin
    v_num_rows := precio_list.count();
    forall i in 1 .. v_num_rows
        insert into producto_log(producto_log_id, producto_id,
            precio_anterior, precio_nuevo, fecha_actualizacion,
            usuario_actualiza)
        values(precio_list(i).producto_log_id, precio_list(i).producto_id,
            precio_list(i).precio_anterior, precio_list(i).precio_nuevo,
            precio_list(i).fecha_actualizacion, precio_list(i).usuario_actualiza
        );

    dbms_output.put_line('Se han vaciado ' || v_num_rows || ' registros');
    precio_list.delete();
end vacia_coleccion;

--inicia la sección before each row
before each row is
    --declara variables que solo se usan en este bloque
    v_usuario varchar2(30) := sys_context('USERENV', 'SESSION_USER');
    v_fecha date := sysdate;
    v_index number;
    v_umbral number := 10;
begin
    --asigna espacio a la colección
    precio_list.extend;
    --obtiene el índice siguiente para guardar el objeto modificado
    v_index := precio_list.last;
    --guarda el nuevo registro cuyo precio ha cambiado.
    precio_list(v_index).producto_log_id := producto_log_seq.nextval;
    precio_list(v_index).producto_id := :new.producto_id;
    precio_list(v_index).precio_anterior := :old.precio;
    precio_list(v_index).precio_nuevo := :new.precio;
    precio_list(v_index).fecha_actualizacion := v_fecha;
    precio_list(v_index).usuario_actualiza := v_usuario;

    -- si la lista tiene más elementos de los establecidos por el umbral,
    -- se vacia.
    if v_index >= v_umbral then
        vacia_coleccion();
    end if;

end before each row;
--inicia after statement
--aquí se hacen las inserciones de forma eficiente
after statement is
begin
    vacia_coleccion();
end after statement;
end;
/
show errors;

```

- Observar la creación de la variable `v_umbral` que se emplea para revisar el tamaño de la colección. En este ejemplo, si el número de elementos en la colección alcanza los 10 elementos, la colección será vaciada.
- Observar que en esta sección de declaraciones globales se ha creado un procedimiento `vacia_coleccion` (notar que se omite la instrucción `create` ya que forma parte únicamente del código del trigger).
- Como se puede ver, en esta sección se puede crear cualquier tipo de objeto o inicializar cualquier tipo de variable que sea requerida para poder ejecutar el código del trigger en los eventos de interés.
- Notar que en el código del procedimiento se hace la inserción de los registros que contiene la colección y se vacía empleando el método `delete`.
- Observar ahora que en el bloque `before each row` se verifica si la colección tiene más de los registros permitidos, de ser así se invoca al procedimiento para que se haga el vaciado de la colección.
- Finalmente, en el bloque `after statement` se hace un último vaciado para asegurar que todos los registros han sido insertados en `producto_log`.

Un ejemplo de la salida al ejecutar la sentencia `update` antes mencionada es:

```
SQL> update producto
set precio =precio*1.1
where lower(nombre) like 'wine%';
```

```
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 10 registros
Se han vaciado 7 registros
```

```
87 rows updated.
```

10.6.3. Restricciones de los DML triggers

Existen algunos riesgos que pueden presentarse al trabajar con triggers.

- Un trigger puede contener en su definición código SQL, por ejemplo, sentencias DML que a su vez provoquen la ejecución de otro trigger. Dicho trigger puede contener a su vez código que provoque la ejecución de otro trigger y así sucesivamente, formando una cascada de llamadas a triggers. Estas llamadas subsecuentes son válidas, pero tienen un límite. En el caso de Oracle, el límite se establece a 32 llamadas en cascada. Posterior a este valor, se lanza excepción.
- En Oracle, el tamaño del código que representa a un trigger debe ser menor a 32,720 bytes. Si este tamaño no es suficiente, el código del trigger se debe dividir para poder ser extraído haciendo uso de funciones o procedimientos. La razón de este valor es que el código de un trigger se almacena en campos tipo Long.

10.6.3.1. Tablas mutantes.

- Este concepto representa a un error que comúnmente ocurre, en especial cuando se está en el proceso de aprendizaje con triggers nivel registro (row level).
- Este error se refiere a que una tabla no puede ser modificada o consultada dentro del código del trigger mientras esta se está actualizando (la tabla está mutando). Para entender este concepto considerar el siguiente escenario:

Si un trigger se ejecuta debido a un cambio realizado a los datos de una tabla, dicho trigger NO puede ver el estado final del cambio sino hasta el final del código del trigger. Por lo tanto, el trigger no debe generar sentencias que consulten o modifiquen datos hacia la misma tabla que lo disparó.



En los ejemplos anteriores de triggers tipo 'statement level' se aplicaron cambios sobre la misma tabla. Esto no representa inconveniente alguno debido a que este problema de tablas mutantes ocurre únicamente para triggers tipo 'row level'. ¿Por qué razón?

Un trigger tipo 'row level' se ejecuta N veces. En cada iteración, el estado de la tabla puede cambiar, y por lo tanto sería inconsistente consultar su estado mientras se ejecutan todas las iteraciones. Por lo contrario, un trigger 'statement level' solo se ejecuta una vez, su estado se conoce justo al ejecutar el trigger por lo que resulta seguro aplicar o consultar el estado actual de los datos de la tabla.

En un trigger tipo 'row level' Es posible acceder a los valores nuevos y viejos empleando :new y :old, pero NO se pueden lanzar consultas o actualizaciones sobre la misma tabla.

Ejemplo:

Suponer la siguiente tabla y los siguientes datos:

LIBRO		
	LIBRO_ID	NUMBER(10,0) NOT NULL
	NOMBRE	VARCHAR2(50) NOT NULL

```
insert into libro(libro_id,nombre) values (1,'L1');
insert into libro(libro_id,nombre) values (2,'L2');
insert into libro(libro_id,nombre) values (3,'L3');
```

Suponer que se define el siguiente trigger

```
create or replace trigger trg_libro
after delete on libro
for each row
declare
  v_count number;
begin
  select count(*) into v_count
  from libro;
  dbms_output.put_line('Numero de registros en la tabla: '||v_count);
end;
/
```

Considerar que se intenta ejecutar la siguiente sentencia:

```
delete from libro where libro_id in(1,2);
```

Ocurre el siguiente error:

```
ERROR at line 1:  
ORA-04091: la tabla JORGE.LIBRO esta mutando, puede que el  
disparador/la  
funcion no puedan verla  
ORA-06512: en "JORGE.TRG_LIBRO", linea 4  
ORA-04088: error durante la ejecucion del disparador  
'JORGE.TRG_LIBRO'
```

- Observar que el trigger intenta obtener el número de registros de la misma tabla. Esta instrucción es invalida ya que no se conoce aún cuantos registros tendrá la tabla hasta que el trigger se ejecute por cada registro, en este caso con `libro_id` 1 y 2.

Resumen:

Evitar la ejecución de sentencias insert, update, delete dentro del código del trigger que apunten a una misma tabla.

Para resolver el problema, replantear la estrategia, o hacer uso de las variables `:new` y `:old`. Otra forma de resolver este problema es con el uso de un *compound DML trigger*.

10.7. PROCEDIMIENTOS ALMACENADOS

En secciones anteriores se revisó que los bloques anónimos que se compilan cada vez que son ejecutados y no se almacenan en la base de datos. Si se desea que estos bloques sean guardados en la base de datos, se requiere hacer uso de un procedimiento almacenado (stored procedure).

Un procedimiento almacenado es un programa PL/SQL que es compilado y almacenado en el servidor. Una vez realizado esto, no se requiere volver a escribir todas las instrucciones sino únicamente hacer referencia al procedimiento. Esto mejora el rendimiento del servidor, ya que el programa se compila una sola vez y se ejecuta N veces.

A diferencia de los bloques anónimos, un procedimiento requiere de un nombre. Una función puede regresar un valor, un procedimiento no regresa valores. Tanto las funciones como los procedimientos almacenados pueden ser invocados por el usuario.

Sintaxis:

```

create [or replace] procedure [schema.]<procedure_name>
[
    (argument [{ in | out | in out }][nocopy] datatype [default expr]
    [,argument [{ in | out | in out }][nocopy] datatype [default expr]
    ]...
)
]
[ invoker_rights_clause ]
{ is | as }
{ pl/sql_subprogram_body | call_spec } ;

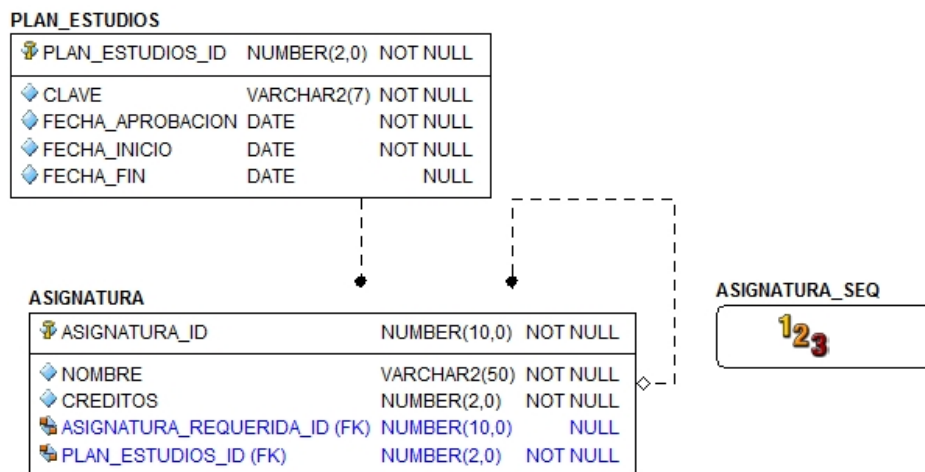
```

- El usuario que desee crear un procedimiento deberá contar con el privilegio `create procedure`.

Ejemplo:

Considerando el siguiente modelo relacional, crear un procedimiento almacenado llamado `creaAsignatura` que será empleado para registrar una nueva asignatura. Las especificaciones del procedimiento son:

- Recibirá como valores de entrada: nombre de la asignatura, créditos, clave del plan de estudios, nombre de la asignatura requerida en caso que la materia esté seriada con otra.
- Adicional a los parámetros anteriores, el procedimiento recibirá un parámetro de salida. El procedimiento deberá inicializar este parámetro con el identificador de la asignatura asignado empleando la secuencia.
- Notar que el procedimiento recibe la clave del plan de estudios, por lo que deberá obtener el identificador correspondiente para ser asignado en el campo `plan_estudios_id`.
- De forma similar, el procedimiento recibe de forma opcional el nombre de la asignatura requerida. El procedimiento deberá obtener el identificador correspondiente.



El código del procedimiento se muestra a continuación.

```

create or replace procedure creaAsignatura (
    v_asignatura_id out number ,v_nombre in varchar2,
    v_creditos in number,v_clave_plan in varchar2,
    v_nombre_asignatura_requerida in varchar2 default null) is

    --declaracion de variables

```

```

v_asignatura_req_id asignatura.asignatura_requerida_id%type;
v_plan_id plan_estudios.plan_estudios_id%type;

begin

--generando el siguiente id de asignatura
select asignatura_seq.nextval into v_asignatura_id
from dual;

--obtiene el id del plan
select plan_estudios_id into v_plan_id
from plan_estudios
where clave =v_clave_plan;

--en caso de haber asignatura requerida, obtiene el id
if v_nombre_asignatura_requerida is not null then
  select asignatura_id into v_asignatura_req_id
  from asignatura
  where lower(nombre) = lower(v_nombre_asignatura_requerida);
else
  v_asignatura_req_id := null;
end if;
--insertando a la nueva asignatura
insert into asignatura(asignatura_id,nombre,creditos, plan_estudios_id,
  asignatura_requerida_id)
values(v_asignatura_id,v_nombre,v_creditos,v_plan_id,v_asignatura_req_id);
end;
/
show errors

```

- En este ejemplo la principal finalidad de construir el procedimiento es ocultar al usuario los detalles de las consultas SQL requeridas para crear una asignatura. El usuario final únicamente proporciona los datos y el procedimiento se encarga de aplicar las sentencias SQL necesarias para registrar una nueva asignatura.
- En general, un procedimiento permite implementar y ocultar cierta lógica de código que puede ser reutilizable.
- Observar que el procedimiento recibe 5 parámetros. Para cada uno de ellos se especifica su tipo:
 - IN: Es un parámetro de entrada. Se emplea únicamente para leer su valor dentro del cuerpo del procedimiento.
 - OUT: Es un parámetro de salida. Este tipo de parámetro permite implementar una especie de valor o valores de retorno. El procedimiento podrá escribir o modificar el valor de este tipo de parámetros para que puedan ser leídos fuera del procedimiento, posterior a su ejecución. Este tipo de parámetros funcionan similar a la estrategia de paso de valores por referencia.
 - IN OUT: Es un parámetro que combina los 2 tipos anteriores. Puede ser tanto de entrada como de salida.
- Observar que en la declaración de la lista de parámetros no se especifica la precisión o el tamaño del tipo de dato. Solo se especifica su tipo.
- Observar la palabra `default` empleada en el parámetro `v_nombre_asignatura_requerida`. Este parámetro es considerado opcional ya que, si el usuario no lo especifica, se empleará el valor por defecto, que en este caso es `null`.

- Observar la declaración de 2 variables adicionales para almacenar los identificadores del plan de estudios y de la asignatura requerida. En un procedimiento almacenado no se requiere especificar el bloque `declare`.
- Finalmente, observar la lógica del procedimiento para obtener los valores requeridos para crear la nueva asignatura.

10.7.1.1. Ejecución de un procedimiento almacenado.

En SQL *Plus se puede invocar un procedimiento empleando la siguiente sintaxis:

```
exec nombreProcedimiento(<param1>,<param2>,...,<paramN>)
```

Ejemplo:

```
set serveroutput on
variable v_id number
exec creaAsignatura(:v_id,'Electronica',12,'PL-2009');
exec dbms_output.put_line(:v_id);
```

- Observar que la sintaxis para declarar variables y para hacer referencia a ella cambia ligeramente. La razón es que el código anterior se está ejecutando directamente el SQL *Plus, el código no se está ejecutando dentro de un programa PL/SQL.
- Para declarar variables directamente en SQL *Plus se emplea la palabra reservada `variable`.
- Para hacer referencia a dicha variable se emplea la sintaxis `:<nombre_variable>`. En este caso `:v_id`
- Notar que se usa `exec` para invocar a cualquier procedimiento o función directamente desde SQL *Plus.

De forma adicional, un procedimiento se puede invocar desde otro programa PL/SQL: trigger, procedimiento o un bloque anónimo. El siguiente código muestra un bloque anónimo que hace uso del procedimiento anterior.

```
set serveroutput on
declare
    v_id number;

begin

    creaAsignatura(v_id,'Electronica',12,'PL-2009');
    dbms_output.put_line('La asignatura Electronica creada con id: '||v_id);

    creaAsignatura(v_id,'Electronica Digital',12,'PL-2009','Electronica');
    dbms_output.put_line(
        'La asignatura Electronica Digital creada con id: '||v_id);
end;
/
```

- `exec` se omite si el procedimiento se invoca desde un programa PL/SQL
- Observar que en la primera llamada no se especifica el parámetro opcional que corresponde con el nombre de la materia antecedente o requerida.

- Existe una observación más al código anterior. El código no es *transaccional*. Las operaciones `insert` que generó el programa al invocar 2 veces al procedimiento nunca fueron confirmadas. Lo correcto es escribir la instrucción `commit` al final de la ejecución del bloque anónimo. Como buena práctica, instrucciones como `commit`, `rollback`, `savepoint` no deben incluirse en el cuerpo del procedimiento. El control transaccional se debe controlar de forma externa:

```
declare
    v_id number;

begin
    creaAsignatura(v_id, 'Electronica', 12, 'PL-2009');
    dbms_output.put_line('La asignatura Electronica creada con id: ' || v_id);
    creaAsignatura(v_id, 'Electronica Digital', 12, 'PL-2009', 'Electronica');
    dbms_output.put_line(
        'La asignatura Electronica Digital creada con id: ' || v_id);
    commit;

    exception
        when others then
            dbms_output.put_line('Error al crear asignaturas');
            rollback;
            raise;
end;
/
```

- Observar que al final del bloque anónimo se hace `commit` para confirmar la inserción de las 2 nuevas asignaturas.
- El bloque `exception` permite manejar cualquier error que ocurra al intentar ejecutar el procedimiento. De ocurrir, se hace `rollback` para garantizar el estado consistente de la base de datos y se re-lanza la excepción. Este es el comportamiento correcto y adecuado desde el punto de vista transaccional. Notar que no se especifica `end` para terminar el bloque de excepción ya que este siempre debe aparecer al final del bloque `begin`.
- El manejo de errores se revisará a detalle más adelante.

10.8. FUNCIONES

- La principal diferencia de una función con respecto a un procedimiento es que una función si puede regresar un valor.
- Lo anterior implica que una función puede ser invocada desde una sentencia SQL mientras que un procedimiento almacenado no.

```
select functionName(<param1>,<param2>,...,<paramN>) from dual
```

- Las funciones también pueden ser empleados como operando derecho en una expresión.

```
v_valor = functionName(<param1>,<param2>,...,<paramN>)
```

Sintaxis:

```

create [or replace] function [schema.]<function_name>
[
    (argument [{ in | out | in out }][nocopy] datatype [default expr]
    [,argument [{ in | out | in out }][nocopy] datatype [default expr]
    ]...
)
]
return {datatype}
[ authid [definer | current_user]]
[ deterministic | parallel_enabled ]
[ pipelined ]
[ result cache [relies on <table_name>]]
is
{ pl/sql_subprogram_body | call_spec } ;

```

- El usuario que desee crear una función deberá contar con el privilegio `create procedure` (el mismo empleado para los procedimientos).

Ejemplo:

- Crear una función que realice la unión de hasta 5 cadenas empleando un carácter de unión. Por ejemplo, para las cadenas "A", "B", "C", "D", "E", generar una cadena empleando el carácter "#" como carácter de unión. La función deberá regresar la cadena "A#B#C#D#E".
- Como mínimo la función deberá recibir las primeras 2 cadenas y el carácter de unión. El código es el siguiente:

```

create or replace function joinStrings(
    join_string varchar2,
    str_1 varchar2,
    str_2 varchar2,
    str_3 varchar2 default null,
    str_4 varchar2 default null,
    str_5 varchar2 default null
) return varchar2 is

    v_str varchar2(4000);
begin
    v_str := str_1||join_string||str_2;
    if str_3 is not null then
        v_str := v_str||join_string||str_3;
    end if;
    if str_4 is not null then
        v_str := v_str||join_string||str_4;
    end if;
    if str_5 is not null then
        v_str := v_str||join_string||str_5;
    end if;
    return v_str;
end;
/
show errors

```

- Observar el primer parámetro corresponde con el carácter de unión. A partir del tercer parámetro se agrega la opción default ya que como mínimo solo se requieren 2 cadenas.

10.8.1.1. Ejecución de funciones.

Existen 2 principales técnicas para ejecutar una función.

- A. Empleando parámetros con *notación posicional*. Esto significa que los parámetros se deben proporcionar en el orden en el que los solicita la función. Si existen parámetros opcionales, se debe especificar un valor por default, o null.

```
select joinStrings('#','A','B',null,null,'E') as "join_string"
from dual;
```

- Observar que se hace uso de la instrucción `select` para poder invocar a la función. De forma similar a otras funciones existentes.
- En este ejemplo se requiere indicar que la tercer y cuarta cadena no tienen valor para poder especificar el quinto parámetro.
- Como resultado se obtiene:

```
join_string
-----
A#B#E
```

- B. Empleando parámetros con *notación por nombre*. Esto significa que los parámetros pueden ser especificados en un orden diferente, pero para ser asignados correctamente, se debe especificar el nombre del parámetro al momento de invocar. En Oracle se emplea el operador `=>` para asociar el nombre del parámetro con su valor.

```
select joinStrings(
  str_1 => 'A',
  str_5 => 'E',
  str_2 => 'B',
  join_string => '#') as "join_string"
from dual;
```

- Observar que los parámetros se especifican en orden diferente.
- Para los casos en donde existen parámetros opcionales, esta técnica suele ser más adecuada ya que no se requiere especificarlos como lo fue el caso anterior.
- Esta notación también puede ser aplicada para invocar procedimientos.

Finalmente, de este ejemplo se pueden mencionar algunas observaciones:

- La función tiene un número finito de parámetros. ¿Qué sucede si se desea unir más de 5 cadenas? Una manera de solucionarlo es mediante el uso de colecciones, el cual se revisará más adelante.
- Observar que la cadena resultante se ha declarado como una cadena con longitud máxima de 4000 caracteres. ¿Qué sucede si al aplicar la unión se excede este valor? Una manera de resolver es emplear un objeto LOB (Large Object), por ejemplo, un tipo de dato CLOB. El manejo de objetos LOB se verá más adelante.

10.8.1.2. Cláusula pipelined

Esta cláusula se emplea para mejorar el desempeño de funciones cuando la función regresa una colección como valor de retorno. En Oracle existen colecciones llamadas varrays o nested tables como se verá más adelante.

Ejemplo:

Crear una función que regrese una colección de números enteros, emplear la cláusula pipelined.

```
create or replace type numeros_varray
as varray(10) of number;
/

create or replace function get_numeros
return numeros_varray
pipelined is

--declaracion de la colección
mis_numeros numeros_varray;
begin
mis_numeros := numeros_varray(0,1,2,3,4,5,6,7,8,9);
for i in 1 .. mis_numeros.last loop
    pipe row(mis_numeros(i));
end loop;
return;
end;
/
show errors

--muestra los resultados
col numeros format a50
select get_numeros() as numeros from dual;
```

- Observar la creación del tipo de dato `numeros_varray` para inicializar una colección de 10 elementos.
- Para poder crear el tipo de dato `numeros_varray`, el usuario debe tener el privilegio `create type`
- Observar el uso de `pipelined` en la definición de la función, el tipo de dato a regresar es una colección de 10 elementos de tipo `numeros_varray`.
- La instrucción `pipe` se encarga de regresar todos los valores de la colección de forma más eficiente.

Los detalles para implementar colecciones se revisarán más adelante.

10.8.1.3. Cláusula result_cache

La cláusula `result_cache` se emplea para guardar en memoria el resultado de una función, así como los valores de los parámetros. Si la función se vuelve a invocar exactamente con los mismos valores de los parámetros, la función no se invoca y se regresa el resultado almacenado en memoria (SGA). Este caché de resultados está disponible para todas las sesiones.

El principal objetivo de esta funcionalidad es mejorar el desempeño al evitar la ejecución de la función repetidamente con los mismos parámetros.

Ejemplo:

```
create or replace function cache_number(v_id number)
  return number
  result_cache relies_on(estudiante) is
  v_resultado number;
begin
  dbms_output.put_line('invocando a la funcion');
  v_resultado := v_id * 2;
  return v_resultado;
end;
/
```

```
set serveroutput on
select cache_number(1) from dual ;
select cache_number(1) from dual ;
select cache_number(2) from dual ;
select cache_number(2) from dual ;
select cache_number(3) from dual ;
```

- Observar que adicional a la cláusula `result_cache` es posible especificar de forma opcional la cláusula `relies_on`. Esta cláusula se emplea para escribir una lista de tablas. Si ocurre un cambio en los datos de dichas tablas, el caché se invalida. En el ejemplo se monitorea la tabla `estudiante`. Esta opción puede ser útil siempre cuando el resultado de la función dependa de los datos que contenga una tabla.
- Para probar esta funcionalidad, observar la salida al invocar la función. El mensaje “invocando a la función” solo aparece en consola al cambiar el valor del parámetro.

10.8.1.4. Algunos aspectos importantes de las funciones.

- Las funciones no pueden incluir código DML. Esto se debe a que si la función se llama desde una sentencia `select` como fue en el ejemplo anterior, dicha sentencia no puede contener a su vez una sentencia DML. El error que se obtiene al intentar esto es ORA-14551 “Can’t have a DML operation inside a query”.

Ejemplo:

```
select crea_asignatura_fx('hidraulica',10,'PL-2009') as id from dual
*
ERROR at line 1:
ORA-14551: no se puede realizar una operacion DML dentro de una consulta
ORA-06512: en "CONTROL_ESCOLAR.CREA_ASIGNATURA_FX", linea 35
ORA-06512: en linea 1
```

- Otra razón que puede justificar lo anterior es por cuestiones de diseño. Generalmente una función debe ser totalmente independiente al exterior. Es decir, debe trabajar únicamente con los parámetros de entrada para producir un valor de retorno. No debe alterar el estado de la base de datos.

- Históricamente los procedimientos almacenados son los encargados para realizar estas tareas.

Existe una forma de crear una función con operaciones DML. Para ello, es necesario especificar que las operaciones DML que realizará la función se harán en un contexto y transacción totalmente independiente a la transacción en la que fue invocada. Si esta situación es válida, entonces es posible agregar código DML. Para indicar esta condición se emplea la instrucción `pragma autonomous_transaction`

El siguiente ejemplo muestra la implementación del procedimiento anterior que se encarga de crear una asignatura, pero ahora empleando una función.

```
create or replace function crea_asignatura_fx (
  v_nombre varchar2,
  v_creditos number,
  v_clave_plan varchar2,
  v_nombre_asignatura_requerida varchar2 default null)
return number is

  --la función ejecuta el siguiente código en su propia transacción
  pragma autonomous_transaction;

  --declaracion de variables
  v_asignatura_req_id asignatura.asignatura_requerida_id%type;
  v_plan_id plan_estudios.plan_estudios_id%type;
  v_asignatura_id asignatura.asignatura_id%type := 0;

begin

  --generando el siguiente id de asignatura
  select asignatura_seq.nextval into v_asignatura_id
  from dual;

  --obtiene el id del plan
  select plan_estudios_id into v_plan_id
  from plan_estudios
  where clave =v_clave_plan;

  --en caso de haber asignatura requerida, obtiene el id
  if v_nombre_asignatura_requerida is not null then
    select asignatura_id into v_asignatura_req_id
    from asignatura
    where lower(nombre) = lower(v_nombre_asignatura_requerida);
  else
    v_asignatura_req_id := null;
  end if;
  --insertando a la nueva asignatura

  insert into asignatura(asignatura_id,nombre,creditos, plan_estudios_id,
    asignatura_requerida_id)
  values(v_asignatura_id,v_nombre,v_creditos,
    v_plan_id,v_asignatura_req_id);

  --la función debe terminar su propia transacción
  commit;
```

```

exception
when others then
    dbms_output.put_line('Error al crear la asignatura');
    --la función termina su propia transacción con rollback
    rollback;

    --Regresa el id asignado, 0 si ocurre un error
    return v_asignatura_id;
end;
/
show errors

```

- Observar el uso de pragma `autonomous_transaction` empleado para delimitar una transacción autónoma que será ejecutada únicamente en el contexto de la función. Una vez que la función termine su ejecución, esta transacción debe concluirse, ya sea a través de un `commit`, o un `rollback` en caso de encontrar algún error como se muestra al final.
- Observar el siguiente código empleado para crear una nueva asignatura:

```

sql> select crea_asignatura_fx('hidraulica',10,'PL-2009') as id from dual;
sql> rollback;
sql> select nombre from asignatura where nombre='hidraulica';

ID
-----
83

```

- Notar que a pesar de especificar `rollback`, el nuevo registro sigue existiendo. Esto debido a que la función realiza la inserción en su propia transacción.

10.9. COLECCIONES.

Las colecciones son estructuras de datos que permiten manejar una gran cantidad de datos en memoria dentro de un programa PL/SQL.

Oracle soporta 3 tipos de colecciones:

- *Varrays*. Similares a un arreglo de longitud fija asignada al momento de su creación, comúnmente usados en otros lenguajes de programación. Este tipo de colección debe ser empleado cuando su tamaño es conocido y estático.
- *Nested tables*. Similar a un objeto `List` de otros lenguajes como Java en donde su tamaño puede aumentar de forma dinámica.
- *Associative arrays*. De forma similar, su tamaño puede crecer de forma dinámica, su estructura es similar a un objeto `Map` o `Set` de otros lenguajes de programación como Java.

Es posible simular colecciones multidimensionales empleando objetos personalizados (objetos definidos por el usuario).

10.9.1. Varrays

- Son estructuras unidimensionales. Cada elemento es referenciado a través de un índice numérico iniciando en 1. En la mayoría de los lenguajes de programación este índice inicia en 0.

- Varrays pueden ser empleados en programas PL/SQL y como tipos de datos para las columnas de una tabla o para la creación de objetos personalizados.

Sintaxis:

- Como primer paso se debe crear un tipo de dato para poder hacer referencia al arreglo.

```
type <type_name> is {varray | varying varray} (size_limit)
of <element_type> [ not null]
```

- Una vez creado el tipo de dato, se declara una variable cuyo tipo de dato corresponda al creado en el punto anterior

```
<type_name> <nombre_variable>
```

Ejemplo:

- Crear un programa PL/SQL que defina un arreglo de 10 elementos con valores 10, 20,...,100 y los imprima en consola. Inicializar el arreglo con valores nulos y posteriormente asignar su valor correspondiente.

```
set serveroutput on
declare
  --declara el tipo de dato int_varray
  type int_varray is varray(10) of number;

  --declara la variable tipo int_varray
  mi_arreglo int_varray;
begin
  --inicializa el arreglo con nulos
  mi_arreglo :=
    int_varray(null,null,null,null,null,null,null,null,null,null);

  --asigna un valor a cada elemento y lo imprime
  for i in 1..mi_arreglo.limit loop
    mi_arreglo(i) := i*10;
    dbms_output.put_line('mi_arreglo('||i||') ='||mi_arreglo(i));
  end loop;
end;
/
```

- Observar las 2 instrucciones en el bloque declare. El primero crea un tipo de dato llamado `int_varray`. Por convención se emplea el tipo de dato del arreglo, seguido de “_” y la palabra `varray`.
- Empleando este tipo de dato se declara una variable llamada `mi_arreglo`. Su tipo de dato es `int_array`.
- Observar la instrucción para inicializar el arreglo con valores `null`. Un arreglo siempre debe ser inicializado, ya sea con valores no nulos, o con valores nulos.
- Existe una forma alternativa para inicializar cada elemento del arreglo sin tener que especificar la lista de valores desde el inicio:

```

set serveroutput on
declare
    --declara el tipo de dato int_varray
    type int_varray is varray(10) of number;

    --declara la variable tipo int_varray
    mi_arreglo int_varray;
begin
    --inicializa el arreglo pero sin elementos
    mi_arreglo := int_varray();

    --asigna un valor a cada elemento y lo imprime
    for i in 1..mi_arreglo.limit loop
        mi_arreglo.extend;
        mi_arreglo(i) := i*10;
        dbms_output.put_line('mi_arreglo('||i||') ='||mi_arreglo(i));
    end loop;
end;
/

```

- Observar las 2 líneas marcadas en negritas. En la primera se inicializa un arreglo, pero sin elementos. La instrucción `mi_arreglo.extend` permite reservar memoria para el siguiente elemento del arreglo. Posterior a ello ya se puede asignar un valor.

10.9.1.1. Uso de varrays en Tablas.

Suponer que se desea almacenar la siguiente información:

- Nombre del estudiante, apellido paterno, apellido materno y una lista de números telefónicos.

Para el caso de los números telefónicos se tiene un atributo multivalor. En el curso se revisaron las posibles técnicas para implementar este requerimiento. Una de ellas es la creación de una tabla de números telefónicos. Esta técnica funciona, pero implica ligar 2 tablas para poder recuperar los datos.

El uso de varrays permite asociar una colección como tipo de dato de una columna. Lo anterior permitirá crear una tabla llamada estudiante que contenga una columna llamada teléfonos y cuyo tipo de dato sea `varchar2_array` que permita almacenar los teléfonos de los estudiantes.

```

create or replace type telefono_varray
as varray(10) of varchar2(20);
/

create table estudiante(
    estudiante_id number(10,0),
    nombre varchar2(50),
    apellido_paterno varchar2(50),
    apellido_materno varchar2(50),
    telefonos telefono_varray
);

insert into estudiante(estudiante_id,nombre,
    apellido_paterno,apellido_materno,telefonos)
values(1, 'Juan', 'Martinez', 'Paz',
    telefono_varray('556598394', '0334902945', '55453234521'));

```

```
select estudiante_id, telefonos from estudiante
```

```
ESTUDIANTE_ID TELEFONOS
```

```
-----
1 TELEFONO_VARRAY ('556598394', '0334902945', '55453234521')
```

- En la primera instrucción se crea un tipo de dato llamado `telefono_varray`. Observar que se usa la instrucción `create or replace`. A diferencia del programa anterior, esta instrucción permite crear un tipo de dato `telefono_varray` como un objeto independiente que puede ser reutilizado en el esquema que fue creado.
- Lo anterior implica que el usuario debe contar con el privilegio `create type` para poder crear el tipo de dato.
- Observar que la instrucción `create or replace type` requiere el uso de `"/"` ya que la definición de un tipo de dato puede contener bloques PL/SQL (`begin, end`).
- Observar la línea marcada en negritas en la definición de la tabla. El tipo de dato del campo `telefonos` es `telefono_varray`. Esto permitirá almacenar la lista de teléfonos.
- Observar la sintaxis para insertar un registro con un arreglo de teléfonos marcada en negritas. Se especifica la lista de teléfonos separados por `'`, en este caso hasta 10 elementos.
- Finalmente, observar la forma en la que se presentan los resultados. La lista de teléfonos se presenta como un `varray`, aunque en este formato no es muy utilizable.

10.9.2. Nested Tables

- Una de las principales diferencias con un `Varray` es la posibilidad de crecimiento dinámico de los elementos de la colección.
- La sintaxis es similar a la vista anteriormente a excepción de especificar el tipo de colección:

```
type <type_name> is table of <element_type> [ not null]
```

- Observar que en este caso no se especifica el tamaño o límite de la colección.
- Las restricciones y reglas para inicializar este tipo de colección son similares a las vistas para las colecciones tipo `Varray`.

10.9.2.1. Uso de nested tables en tablas.

El siguiente código muestra el ejemplo anterior pero ahora haciendo uso de nested tables.

```
create or replace type telefono_table
as table of varchar2(20);
/

create table estudiante(
  estudiante_id number(10,0),
  nombre varchar2(50),
  apellido_paterno varchar2(50),
  apellido_materno varchar2(50),
  telefonos telefono_table
) nested table telefonos store as estudiante_telefonos;
```

- Observar la definición del tipo de dato llamado `telefono_table`, ya no es necesario especificar un tamaño o límite. Por convención se emplea el sufijo `_table`.
- Observar en la definición de la tabla, el tipo de dato de la columna `telefonos`.
- Al final de la definición de la tabla, observar la instrucción empleada para indicar que la columna teléfonos será implementada como un `nested table` (tabla anidada de la tabla estudiante) y que esta se guardará con el nombre `estudiante_telefonos` (el nombre con el que se guarda es cualquier cadena).

La sintaxis empleada para la inserción de registros es idéntica a la empleada con `varrays`, solo se modifica el tipo de dato.

```
insert into estudiante(estudiante_id,nombre,
  apellido_paterno,apellido_materno,telefonos)
values(1,'Juan','Martinez','Paz',
  telefono_table('556598394','0334902945','55453234521'));
```

Al seleccionar los datos de la tabla se muestran de la siguiente forma:

```
select estudiante_id, telefonos from estudiante;
```

```
ESTUDIANTE_ID TELEFONOS
```

```
-----
1 TELEFONO_TABLE('556598394', '0334902945', '55453234521')
```

Finalmente, para extraer cada uno de los elementos del `nested table`, se emplea la siguiente sintaxis. Observar que la consulta trata a la `nested table` como si se tratara de 2 tablas independientes empleando los alias `e` para estudiante y `t` para la `nested table`.

```
select estudiante_id,column_value
from estudiante e, table(e.telefonos) t;
```

```
ESTUDIANTE_ID COLUMN_VALUE
```

```
-----
1 556598394
1 0334902945
1 55453234521
```

10.9.2.2. *Nestes tables empleando Objetos personalizados.*

- En el ejemplo anterior se generó una colección de cadenas, cada una de ellas representa un número telefónico.
- ¿Qué sucede si se desea agregar una colección que defina sus propios atributos?, Por ejemplo, suponer que se desea guardar una lista de teléfonos, pero cada teléfono está formado por los siguientes atributos:
 - Número de teléfono (cadena de caracteres)
 - Tipo de teléfono (cadena que indica el tipo de teléfono: casa, móvil, etc.)
 - Activo (bandera que indica si el teléfono es vigente o no).
- Para resolver este requerimiento, se puede crear un Tipo de dato personalizado y después asociarlo a la colección:


```

create type telefono_t as object (
    numero varchar2(20),
    tipo   varchar2(20),
    activo number(1,0)
);
/

create or replace type telefono_table
as table of telefono_t;
/

```

- Observar el tipo de dato de la nested table ahora es telefono_t.
- La definición de la tabla empleado no cambia:

```

create table estudiante(
    estudiante_id number(10,0),
    nombre varchar2(50),
    apellido_paterno varchar2(50),
    apellido_materno varchar2(50),
    telefonos telefono_table
) nested table telefonos store as estudiante_telefonos;

```

- Inserción de datos: Observar los objetos tipo telefono_t asociados a la nested table:

```

insert into estudiante(estudiante_id,nombre,
    apellido_paterno,apellido_materno,telefonos)
values(1,'Juan','Martinez','Paz',
    telefono_table(
        telefono_t('556598394','casa',1),
        telefono_t('0334902945','depto',1),
        telefono_t('55453234521','celular',0)
    )
);

```

- Consultas de datos.

```
select estudiante_id, telefonos from estudiante;
```

```
ESTUDIANTE_ID TELEFONOS(NUMERO, TIPO, ACTIVO)
```

```

-----
1 TELEFONO_TABLE(TELEFONO_T('556598394', 'casa', 1), TELEFONO_T('0334902945',
'depto', 1), TELEFONO_T('55453234521', 'celular', 0))

```

- Consultar el detalle de la colección:

```

select estudiante_id,t.*
from estudiante e, table(e.telefonos) t;

```

ESTUDIANTE_ID	NUMERO	TIPO	ACTIVO
1	556598394	casa	1
1	0334902945	depto	1
1	55453234521	celular	0

10.9.2.3. API para colecciones.

Existe un API llamada Collection API. La finalidad de esta API es facilitar el acceso a colecciones. Define una serie de métodos (funciones o procedimientos) que permiten facilitar la interacción con colecciones. Algunos ejemplos de estos métodos son: `count`, `delete`, `exists`, `extend`, `first`, `last`, `limit`, `next(n)`, `prior(n)`, `trim`. Para mayores detalles, revisar la documentación oficial de Oracle asociada con el tema de colecciones.

10.10. MANEJO DE ERRORES

- Similar a algunos lenguajes de programación como Java, errores en PL/SQL son manejados empleando excepciones.
- En un programa PL/SQL, las excepciones se manejan en un bloque especial llamado `exception`.
- Existen 2 tipos de errores:
 - Errores de compilación
 - Errores que ocurren en tiempo de ejecución.
- Errores que ocurren en tiempo de ejecución pueden ser manejados en un bloque de excepción, o dejar que se propaguen hasta nivel de la sesión.
- Los bloques de excepción deben incluirse justo antes de la instrucción `end`; correspondiente al bloque `begin`.
- La sintaxis de un bloque es la siguiente:

```
exception
when {<predefined_exception> | <user_defined_exception> | others} then
--exception handling statements;
[ return | exit ]
. . .
```

- Cada bloque `exception` contiene al menos una instrucción `when` empleada para indicar el tipo de excepción que puede ser manejada:
 - Excepciones predefinidas: Oracle contiene diversas excepciones definidas en el paquete `sys.standard`. Cada excepción tiene un nombre único y un código).
 - Excepciones definidas por el usuario: El usuario puede definir sus propias excepciones
 - Others: Significa que el bloque `when` puede manejar cualquier excepción que no se ha manejado hasta este punto.
- Algo importante a destacar es la ausencia de la instrucción `end`. No se requiere cerrar el bloque `exception` ya que aparece al final del bloque `begin`, y tampoco se requiere cerrar cada bloque `when`.

Ejemplo:

```
set serveroutput on
declare
  a varchar2(1);
  b varchar2(2) := 'ab';
begin
  a := b;
exception
  when value_error then
    dbms_output.put_line('Asignacion incorrecta');
end;
/
```

- En el ejemplo anterior, se genera una excepción al intentar asignar el valor de 'a' con el valor de 'b' debido a la longitud máxima de 'a'. Oracle genera una excepción llamada `value_error` cuando el valor a asignar a una variable es incorrecto.
- En el bloque `exception` se maneja este tipo de excepción. La única acción es imprimir un mensaje indicando el problema. Sin embargo, el mensaje no es tan claro y no muestra detalles. Más adelante se revisará la solución a esta mala práctica.
- Si se desea agregar más instrucciones después de un bloque `exception`, se debe crear un bloque `begin` anidado:

```

set serveroutput on
declare
  a number;
  v_codigo number;
  v_mensaje varchar2(64);
begin
  declare
    b varchar2(2);
  begin
    select 1
    into b
    from dual
    where 1 = 2;
  exception
    when value_error then
      dbms_output.put_line('Asignacion incorrecta, '
        || 'no se puede asignar b con un valor numerico');
  end; -- ojo, este end corresponde a begin anidado

  dbms_output.put_line('continuando con la ejecucion');

exception
  when others then
    dbms_output.put_line('Manejando excepcion');
    v_codigo := sqlcode;
    v_mensaje := sqlerrm;
    dbms_output.put_line('Codigo: ' || v_codigo);
    dbms_output.put_line('Mensaje: ' || v_mensaje);
end; -- este end corresponde a begin externo
/

```

En este ejemplo existen 2 causas de excepción:

- La consulta `select` intenta asignar el valor 1 a la variable `b`, pero la variable `b` fue declarada de tipo `varchar2`. Esto genera la excepción `value_error`
- La consulta `select` regresará 0 registros ya que la condición `1 = 2` siempre resultará en un valor `false`. Esto genera una excepción con nombre `no_data_found` ya que la cláusula `into` requiere que la sentencia `select` regrese un solo valor.
- Observar que, al terminar el bloque interno, la ejecución puede continuar

La salida obtenida es:

```

Manejando excepcion
Codigo: 100
Mensaje: ORA-01403: No se ha encontrado ningun dato

PL/SQL procedure successfully completed.

```

- Al ejecutar el ejemplo, la primera excepción que se genera es `no_data_found` ya que para poder asignar el valor a la variable `b` se debió haber obtenido exactamente 1 registro.
- Esta excepción no es manejada por el bloque interno ya que el bloque `exception` solo maneja excepciones del tipo `value_error`.
- La excepción se propaga al bloque exterior. El bloque `exception` exterior maneja cualquier tipo de excepción al incluir la palabra `others`.
- Observar el uso de 2 funciones empleadas para mostrar el detalle de la excepción:
 - `sqlcode`: Obtiene el valor numérico asociado a la excepción.
 - `sqlerrm`: Muestra el nombre de la excepción, mensaje del error y el código de error de la excepción.

Ejemplo:

Observar la salida del siguiente programa.

```

set serveroutput on
declare
  a varchar2(1) := '&1';
begin
  dbms_output.put_line(
    'Valor de la variable de sustitucion: ' || a);

exception
  when others then
    dbms_output.put_line('Manejando excepcion generica');
end;
/

```

Al ejecutar el programa se obtiene lo siguiente:

```

Enter value for 1: hola
old 2: a varchar2(1) := '&1';
new 2: a varchar2(1) := 'hola';
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: error : character string buffer too small numerico o de
valor
ORA-06512: en linea 2

```

- Recordando, la sintaxis `&1` en Oracle corresponde al concepto de variable de sustitución. Esto significa que el programa solicitará un valor al usuario para ser asignado a la variable 'a'.
- El usuario captura el valor 'hola'.
- Se produce una excepción ya que la variable 'a' fue declarada con una longitud de 1 carácter, y la palabra hola es de 4.

- Observar que el mensaje que se encuentra en el bloque `exception` no aparece en la salida del programa. Esto significa que la excepción **no** fue manejada en el bloque a pesar de usar `others`. ¿por qué razón?
- Las excepciones que ocurren en el bloque `declare` no pueden ser manejadas por el bloque local. Si se desea manejar una excepción en este bloque, es necesario agregar un bloque exterior:

```

set serveroutput on

begin

    declare
        a varchar2(1) := '&1';
    begin
        dbms_output.put_line(
            'Valor de la variable de sistutucion: ' || a);

    exception
        when others then
            dbms_output.put_line('Manejando excepcion generica');
    end;

exception
    when others then
        dbms_output.put_line('Este bloque si se ejecuta!');
    end;
/

```

10.10.1. Excepciones pre-definidas.

La siguiente tabla muestra los códigos y los nombres de las excepciones predefinidas

Nombre	Código
<code>access_into_null</code>	-6530
<code>case_not_found</code>	-6592
<code>collection_is_null</code>	-6531
<code>cursor_already_open</code>	-6511
<code>dup_val_on_index</code>	-1
<code>invalid_cursor</code>	-1001
<code>invalid_number</code>	-1722
<code>login_denied</code>	-1017
<code>no_data_found</code>	+100
<code>no_data_needed</code>	-6548
<code>not_logged_on</code>	-1012
<code>program_error</code>	-6501
<code>rowtype_mismatch</code>	-6504
<code>self_is_null</code>	-30625
<code>storage_error</code>	-6500
<code>subscript_beyond_count</code>	-6533
<code>subscript_outside_limit</code>	-6532
<code>sys_invalid_rowid</code>	-1410
<code>timeout_on_resource</code>	-51
<code>too_many_rows</code>	-1422
<code>value_error</code>	-6502
<code>zero_divide</code>	-1476

10.10.2. Excepciones definidas por el usuario.

Existen 2 técnicas para crear excepciones definidas por el usuario:

- Declarando una variable de tipo `exception`. Representa la forma más simple, pero no permite asignar un código ni un mensaje.
- Construir una excepción dinámica. Permite especificar código y mensaje. Los códigos a emplear deben estar en el rango negativo [-20,999, -20,000]

Ejemplo:

```
set serveroutput on
declare
  e exception;
begin
  dbms_output.put_line('Iniciando ejecucion');
  raise e;
  dbms_output.put_line('Esta linea ya no se ejecuta');
exception
  when others then
    if sqlcode = 1 then
      dbms_output.put_line('Error generado: ' || sqlerrm);
    end if;
end;
/
```

- El ejemplo anterior representa la forma más simple para crear una excepción.
- La instrucción `raise` se emplea para lanzarla.
- Observar que la última instrucción ya no se ejecuta ya que la excepción ha sido lanzada en la línea anterior.
- La única forma de manejar la excepción es empleando `others`. Por default el código de error de las excepciones creadas por el usuario es 1.

10.10.2.1. Creación de excepciones dinámicas definidas por el usuario.

- Permiten lanzar excepciones asignándoles un código de error y un nombre.

Sintaxis:

```
raise_application_error(<error_code>,<error_message>[, keep_error])
```

- El parámetro `keep_error` es una bandera que indica si el error producido por la esta excepción será agregado a un stack de errores existente. El valor por default es `false`.

Ejemplo:

```

set serveroutput on

begin
    raise_application_error(-20001, 'Creando y lanzando excepcion');
exception
    when others then
        dbms_output.put_line('Manejando excepcion: ' || sqlerrm);

end;
/

```

- El ejemplo crea una excepción con un código y mensaje.
- El único detalle que existe es que la excepción no se puede manejar empleando su nombre ya que no fue creada empleando un nombre. La única opción es emplear `others`.
- El código asignado debe estar en el rango de valores antes asignado.

El siguiente programa muestra la forma en la que se crea y maneja una excepción empleando su nombre:

```

set serveroutput on
declare
    mi_excepcion exception;
    pragma exception_init(mi_excepcion, -20001);
begin

    raise_application_error(-20001, 'Lanzando excepcion con nombre
mi_excepcion');
exception
    when mi_excepcion then
        dbms_output.put_line('Manejando excepcion: ' || sqlerrm);
        dbms_output.put_line('Re-lanzando error');
        raise;
end;
/

```

- La instrucción `pragma` es una directiva (o 'hint') que se le especifica al compilador para realizar cierto procesamiento o modificar el comportamiento de un programa PL/SQL. El procesamiento que se realiza con esta instrucción se ejecuta en tiempo de compilación y no en tiempo de ejecución. Este procesamiento es pasado o proporcionado al compilador para poder procesar de forma adecuada al programa PL/SQL.
- En este caso, la instrucción `pragma exception_init` permite inicializar a una excepción con nombre `mi_excepcion` y código 20001. Observar que la instrucción aparece en el bloque `declare`.
- Observar que el nombre de la excepción corresponde con el nombre de la variable.
- Observar que ahora es posible hacer referencia a la excepción empleando su nombre en el bloque `exception`.
- Finalmente, notar que se emplea la instrucción `raise` sin argumentos dentro del bloque `exception` para re-lanzar la excepción. Dependiendo la lógica del programa, pueden existir situaciones en las que se deba manejar la excepción para realizar algún procesamiento y después re-lanzarla para terminar el programa o para notificar su existencia a un bloque externo o propagarla hasta nivel de la sesión.

10.10.3. Stack Traces

En aplicaciones reales generalmente las excepciones se re-lanzan empleando la instrucción `raise` para que estas sean propagadas y manejadas en un bloque superior o inclusive se dejan propagar hasta nivel de sesión.

Esta técnica provoca la formación de una especie de cadena o 'traza' que indica el origen de la excepción, así como las líneas de código donde la excepción fue relanzada. Este concepto existe también en otros lenguajes como son los StackTraces de las excepciones en el lenguaje Java.

Considerar el siguiente ejemplo:

```
begin
  dbms_output.put_line('Bloque nivel 1');

begin
  dbms_output.put_line('Bloque nivel 2');

begin
  dbms_output.put_line('Bloque nivel 3');
  dbms_output.put_line('provocando exception' || (3/0) );
exception
  when others then
    raise;
end;
exception
  when others then
    raise;
end;
exception
  when others then
    raise;
end;
/
```

- En este código existen 3 bloques anidados, y en el tercer nivel se está provocando una excepción ya que se intenta realizar una división con cero.
- Cada uno de los bloques contiene un bloque `exception` en donde únicamente la excepción es relanzada.
- La salida de este programa es la siguiente:

```
Bloque nivel 1
Bloque nivel 2
Bloque nivel 3
begin
  *
  ERROR at line 1:
  ORA-01476: el divisor es igual a cero
  ORA-06512: en línea 20
```

- Observar que el error únicamente se reporta en la línea 20 la cual se ha resaltado en negritas. En realidad, esta línea solo relanzó la excepción, pero no se indica la línea que originó el problema (línea 9).

Para resolver este inconveniente, existen algunas funciones de utilidad que permiten formatear y presentar la historia completa de un stacktrace. Existen 3 funciones:

- `dbms_utility.format_call_stack`
- `dbms_utility.format_error_backtrace`
- `dbms_utility.format_error_stack`

Para cuestiones de desarrollo y debugueo `dbms_utility.format_error_backtrace` es la que proporciona el mayor detalle. Se recomienda agregar la instrucción en cada bloque de excepción:

```
set serveroutput on
begin
  dbms_output.put_line('Bloque nivel 1');

  begin
    dbms_output.put_line('Bloque nivel 2');

    begin
      dbms_output.put_line('Bloque nivel 3');
      dbms_output.put_line('provocando exception' || (3/0) );
    exception
      when others then
        dbms_output.put_line(dbms_utility.format_error_backtrace);
        raise;
      end;
    exception
      when others then
        dbms_output.put_line(dbms_utility.format_error_backtrace);
        raise;
      end;
    exception
      when others then
        dbms_output.put_line(dbms_utility.format_error_backtrace);
        raise;
      end;
  end;
/
```

La salida ahora es:

```
Bloque nivel 1
Bloque nivel 2
Bloque nivel 3
ORA-06512: en linea 9

ORA-06512: en linea 13

ORA-06512: en linea 18
```

```
begin
  *
  ERROR at line 1:
  ORA-01476: el divisor es igual a cero
ORA-06512: en linea 23
```

- Observar las 4 líneas resaltadas que muestran el stacktrace completo.
 - En la línea 9 se origina el problema.
 - En la línea 13 la excepción es relanzada en el bloque 3
 - En la línea 18 la excepción es relanzada en el bloque 2
 - Finalmente, en la línea 23 la excepción es relanzada en el bloque 1

10.11. LARGE OBJECTS (LOBs)

LOBs son estructuras empleadas para almacenar objetos como texto, archivos binarios (imágenes, videos, documentos), etc. A partir de Oracle 11g, se hace uso de un nuevo concepto llamado **Secure files** que permite incrementar considerablemente el desempeño para realizar el manejo de este tipo de objetos.

Se emplean los tipos de datos `clob`, `nclob` y `blob` para almacenar LOBs en la base de datos, y `bfile` para almacenar objetos fuera de ella. `Bfile` almacena una especie de 'puntero' que permite ubicar al archivo fuera de la BD.

10.11.1. Objetos CLOB

- Se emplean los tipos de datos `clob` y `nclob` (n = national, permite almacenar texto empleando un juego de caracteres diferente al configurado en la BD: UNICODE). Para efectos del curso, solo se revisa el tipo de dato `clob`.
- Este tipo de dato permite almacenar hasta 128 terabytes de texto en un solo registro, por ejemplo, un capítulo completo de un libro, un documento xml, etc.
- Las columnas declaradas con tipo de dato `clob` típicamente se almacenan de forma separada con respecto al resto del registro. En su lugar se almacena un 'descriptor' o 'locator' el cual puede visualizarse como un puntero que lleva al lugar físico donde se encuentra el texto.
- Existen diversas formas de inicializar un objeto `clob`:

```
v_texto1 clob; --declara un objeto clob nulo
v_texto2 clob := empty_clob(); --declara e inicializa un objeto clob vacio.
v_texto3 clob := 'texto'; --inicializa un objeto clob con una cadena.
```

10.11.2. Lectura de archivos y almacenamiento en la Base de Datos.

El paquete `dbms_lob` contiene todas las funciones y herramientas necesarias para cargar archivos de gran tamaño para ser cargados en la base de datos. Para realizar este proceso se requieren de diversos pasos:

- Crear un objeto `directory`. Este tipo de objeto se almacena en la base de datos y puede visualizarse como un directorio virtual que contiene un nombre y apunta a un directorio físico en el sistema de archivos.

```
create or replace directory data_dir as '/tmp/data';
```

- Esta instrucción debe ser ejecutada por el usuario SYS ya que es el único que puede crear objetos tipo `directory`.
 - En este ejemplo, el nombre del directorio `data_dir` mapea al directorio real `/tmp/data`
- Para que un usuario puede hacer uso del directorio anterior, este debe contar con el privilegio `read on directory`.

```
grant read on directory data_dir to <usuario>
```

- El siguiente paso es crear un programa PL/SQL encargado de leer el contenido de un archivo de texto y almacenarlo en la base de datos.

Ejemplo:

Suponer que existe la siguiente tabla y secuencia.

LIBRO_SEQ	
123	

LIBRO			
LIBRO_ID	NUMBER(10,0)	NOT NULL	
TITULO	VARCHAR2(50)	NOT NULL	
TEXTO	CLOB	NOT NULL	

Crear un procedimiento encargado de registrar un nuevo libro. El procedimiento recibirá los siguientes parámetros:

- p_articulo_id: Parámetro de salida en el que el procedimiento actualizará su valor con el identificador asignado empleando la secuencia.
- p_titulo: el título del libro
- p_nombre_archivo: El nombre del archivo donde se encuentra el texto del libro. Su contenido deberá ser insertado en la columna texto cuyo tipo de dato es CLOB.
- Suponer que los archivos de texto se encuentran en el directorio /tmp/data_dir

Solución:

- Como primer paso es necesario la existencia del objeto `directory` en la base de datos. Esta instrucción deberá ser ejecutada por el usuario SYS.

```
create or replace directory data_dir as '/tmp/data_dir';
```

- No olvidar otorgar el privilegio correspondiente para que el usuario que genere el procedimiento pueda leer archivos del directorio.

```
grant read on directory data_dir to jorge;
```

- Las siguientes instrucciones se emplean para crear un directorio y un archivo de prueba. Observar que en SQL *Plus se pueden ejecutar instrucciones del sistema operativo ante poniendo el carácter `!` antes del comando.

```
!mkdir -p /tmp/data_dir
!echo 'Este es un texto de prueba ' > /tmp/data_dir/prueba.txt
```

- Estas 2 instrucciones no deben ser ejecutadas con el usuario SYS.

El archivo `prueba.txt` será empleado para verificar el correcto funcionamiento del procedimiento.

El código del procedimiento se muestra a continuación.

```
create or replace procedure crea_libro(p_libro_id out number,
  p_titulo in varchar2, p_nombre_archivo in varchar2) is

  v_bfile bfile;
  v_src_offset number := 1;
  v_dest_offset number:= 1;
  v_dest_clob clob;
  v_src_length number;
  v_dest_length number;
  v_lang_context number := dbms_lob.default_lang_ctx;
  v_warning number;

begin

  v_bfile := bfilename('DATA_DIR',p_nombre_archivo);
  if dbms_lob.fileexists(v_bfile) = 1 and not
    dbms_lob.isopen(v_bfile) = 1 then
    dbms_lob.open(v_bfile,dbms_lob.lob_readonly);
  else
    raise_application_error(-20001,'El archivo '
      ||p_nombre_archivo
      ||' no existe en el directorio DATA_DIR'
      ||' o el archivo esta abierto');
  end if;

  select libro_seq.nextval into p_libro_id
  from dual;

  insert into libro(libro_id,titulo,texto)
  values(p_libro_id,p_titulo,empty_clob());

  select texto into v_dest_clob
  from libro
  where libro_id = p_libro_id;

  dbms_lob.loadclobfromfile(
    dest_lob => v_dest_clob,
    src_bfile => v_bfile,
    amount => dbms_lob.getlength(v_bfile),
    dest_offset => v_dest_offset,
    src_offset => v_src_offset,
    bfile_csid => dbms_lob.default_csid,
    lang_context => v_lang_context,
    warning => v_warning
  );
  dbms_lob.close(v_bfile);

  v_src_length := dbms_lob.getlength(v_bfile);
  v_dest_length := dbms_lob.getlength(v_dest_clob);

  if v_src_length = v_dest_length then
    dbms_output.put_line('Escritura correcta, bytes escritos: '
      || v_src_length);
  else
    raise_application_error(-20002,'Error al escribir datos.\n'
      ||' Se esperaba escribir '||v_src_length
      ||' Pero solo se escribio '||v_dest_length);
  end if;
end;
/
show errors
```

- Posterior a la declaración de las variables, observar la creación de un objeto `bfile` que representa al archivo `prueba.txt` ubicado en el directorio representado por el objeto `DATA_DIR`.
- Notar que el nombre del directorio '`DATA_DIR`' está en mayúsculas. Esto se debe a que los nombres de los objetos se guardan en el diccionario de datos en mayúsculas por default.
- El objeto `bfile` se emplea en todo el procedimiento para representar al archivo origen `prueba.txt`
- Observar las validaciones que siguen. El archivo se abre en modo lectura para poder ser leído, se lanza excepción en caso de no poder realizar esta acción.
- Observar que en la instrucción `insert`. Como primer paso, se inserta un objeto `clob` vacío. Recordando lo mencionado anteriormente, al realizar la instrucción `insert`, en la columna `texto` se guardará un puntero (locator) que indica la ubicación física de los datos, en este caso, del texto del libro.
- Para poder hacer referencia a este puntero se realiza la instrucción `select` empleando la variable `v_dest_clob`.
- La referencia o locator obtenido en el punto anterior es empleado por el procedimiento `loadclobfromfile` para escribir el contenido del archivo.
- Básicamente este procedimiento lee el contenido del archivo especificado por el objeto `bfile` (`v_bfile`) y escribe su contenido en la ubicación indicada por el objeto `clob` destino `v_dest_clob`. Notar que se emplean parámetros nombrados para invocar el procedimiento.
- La descripción de los parámetros se muestra a continuación:

```

dbms_lob.loadclobfromfile (
  dest_lob      in out nocopy  nocopy clob character set any_cs,
  src_bfile     in             bfile,
  amount        in             integer,
  dest_offset   in out         integer,
  src_offset    in out         integer,
  bfile_csid    in             number,
  lang_context  in out         integer,
  warning       out            integer);

```

Nombre del parámetro	Descripción
<code>Dest_lob</code>	Locator destino (Objeto Clob/NClob) donde se realizará la escritura. Recordar que el objeto CLOB tiene asociado a un puntero (locator) que indica la ubicación física donde se hará la escritura. Este puntero es lo que realmente se almacena en la tabla.
<code>src_bfile</code>	Objeto Bfile que representa al archivo a partir del cual se realizará la lectura de los datos.
<code>amount</code>	Cantidad de bytes a leer. Se puede emplear <code>dbms_lob.lobmaxsize</code> para leer todo el contenido del archivo.
<code>src_offset</code>	Parámetro tanto de salida como de entrada que se especifica el número de índice donde iniciará la lectura. Por ejemplo, si se desea escribir a partir del segundo carácter de la palabra Hola, su valor será 2. Por lo general su valor es 1, que indica una lectura desde el primer carácter. Observar que se hace uso de una variable en lugar de pasarle el valor directo ya que se requiere de una variable, para que el procedimiento pueda escribir en ella (out)

Nombre del parámetro	Descripción
dest_offset	Similar al anterior, pero en teste caso indica el índice donde se inicia la escritura. Típicamente su valor es 1 lo que indica que la escritura comenzará desde la primera posición.
bfile_csid	Juego de caracteres del archivo origen. dbms_lob.default_csid Emplea el default de la base de datos.
warning	Empleado para indicar posibles mensajes de advertencia que pudieran ocurrir durante la carga.

Finalmente, el siguiente código invoca al procedimiento anterior.

```
variable v_libro_id number
exec crea_libro(:v_libro_id, 'Mi primer libro', 'prueba.txt');
```

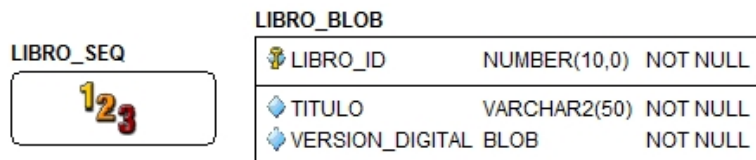
10.11.3. Objetos BLOB

La forma de trabajar con objetos BLOB es similar a la técnica anterior.

```
v_bin1 blob; --declara un objeto blob nulo
v_bin2 blob := empty_blob(); --declara e inicializa un objeto blob vacío.
v_bin3 blob := '43'||'41'||'52'; --inicializa un objeto BLOB con una cadena
--en formato hexadecimal que representa a
--la palabra CAR.
```

En el siguiente ejemplo se muestra una variante del ejemplo anterior.

Suponer que el contenido del libro ahora se guarda en un documento PDF (archivo binario) en el campo version_digital.



La variante del procedimiento anterior para guardar objetos BLOB en la tabla anterior se muestra a continuación. Considerar que existe un archivo llamado prueba.pdf en el directorio configurado.

Observar las diferencias del procedimiento resaltadas en negritas.

```

create or replace procedure crea_libro_blob(p_libro_id out number,
p_titulo in varchar2, p_nombre_archivo in varchar2) is

v_bfile bfile;
v_src_offset number := 1;
v_dest_offset number:= 1;
v_dest_blob blob;
v_src_length number;
v_dest_length number;

begin

    v_bfile := bfilename('DATA_DIR',p_nombre_archivo);
    if dbms_lob.fileexists(v_bfile) = 1 and not
        dbms_lob.isopen(v_bfile) = 1 then
        dbms_lob.open(v_bfile,dbms_lob.lob_readonly);
    else
        raise_application_error(-20001,'El archivo '
        ||p_nombre_archivo
        ||' no existe en el directorio DATA_DIR'
        ||' o el archivo esta abierto');
    end if;

    select libro_seq.nextval into p_libro_id
    from dual;

    insert into libro_blob(libro_id,titulo,version_digital)
    values(p_libro_id,p_titulo,empty_blob());

    select version_digital into v_dest_blob
    from libro_blob
    where libro_id = p_libro_id;

    dbms_lob.loadblobfromfile(
        dest_lob => v_dest_blob,
        src_bfile => v_bfile,
        amount => dbms_lob.getlength(v_bfile),
        dest_offset => v_dest_offset,
        src_offset => v_src_offset);
    dbms_lob.close(v_bfile);

    v_src_length := dbms_lob.getlength(v_bfile);
    v_dest_length := dbms_lob.getlength(v_dest_blob);

    if v_src_length = v_dest_length then
        dbms_output.put_line('Escritura correcta, bytes escritos: '
        || v_src_length);
    else
        raise_application_error(-20002,'Error al escribir datos.\n'
        ||' Se esperaba escribir '||v_src_length
        ||' Pero solo se escribio '||v_dest_length);
    end if;
end;
/
show errors

```

- Observar que ahora se emplea el procedimiento `loadblobfromfile`. El número de parámetros es menor al procedimiento `loadclobfromfile`.

De forma similar al ejemplo anterior, para invocar al procedimiento desde SQL *Plus:

```
variable v_id number
exec crea_libro_blob(:v_id, 'Mi primer libro', 'test.pdf');
```

10.12. SQL DINÁMICO.

- SQL Dinámico es una funcionalidad que ofrece la programación PL/SQL la cual permite escribir y formar código SQL dentro de un programa y ser ejecutado.
- Visto de otra forma, en un programa se puede definir una variable, por ejemplo, de tipo `varchar2` que contenga código SQL y este pueda ser ejecutado como tal.
- Esta funcionalidad se emplea generalmente para construir sentencias SQL de forma dinámica con base a ciertas condiciones que ocurran durante la ejecución del programa PL/SQL.
- En Oracle a esta funcionalidad se le conoce como **Native Dynamic SQL** (NDS).
- El código que se genera de forma dinámica generalmente es código DDL o código DML.

10.12.1. Sentencias DDL dinámicas.

- Suponer que se desea escribir un programa PL/SQL que elimine una secuencia llamada `my_sequence`. Una solución inicial podría ser:

```
set serveroutput on
begin
  dbms_output.put_line('eliminando secuencia');
  drop sequence my_sequence;
end;
/
```

- El programa anterior no compila.
- La principal razón es que en un programa PL/SQL no es posible ejecutar sentencia DDL como son `create`, `drop`, etc.
- Suponiendo que esta restricción no existiera, habría una razón más: Si la secuencia no existiera, el programa no compilaría ya que en tiempo de compilación se verificaría que el objeto existiera, y posterior a su eliminación, el programa se volvería inválido ya que la secuencia fue eliminada.

La forma correcta de implementar este requerimiento es:

- Verificar si la secuencia existe.
- En caso de existir, generar una cadena que contenga la sentencia DDL y ejecutarla de forma dinámica. El programa compilará sin problemas ya que las cadenas que contienen código SQL no se validan en tiempo de compilación. Esto se realiza hasta que el programa se ejecuta.
- Código DDL se puede ejecutar, pero únicamente como SQL dinámico.

Ejemplo:


```

set serveroutput on
declare
  v_sql varchar2(4000);
  v_exists number(1);
begin

  dbms_output.put_line('Verificando si la secuencia existe');
  select count(*) into v_exists
  from user_sequences
  where sequence_name = 'MY_SEQUENCE';
  if v_exists = 1 then
    dbms_output.put_line('eliminando secuencia');
    v_sql := 'drop sequence my_sequence';
    --ejecuta la sentencia ddl de forma dinámica;
    execute immediate v_sql;
  else
    dbms_output.put_line('La secuencia my_sequence no existe');
  end if;
end;
/

```

- Observar que se usa la función `count` para validar si la secuencia existe. La razón de su uso es que se obtendrá un valor 0 cuando la secuencia no exista y se obtendrá 1 cuando la secuencia exista (1 registro encontrado).
- Lo anterior permite que siempre se asigne un valor a la variable `v_exists` (0 ó 1) y evitar errores asociados con la cláusula `into`: Recordar que `into` requiere siempre de un valor (revisar la sección de cursores). Por ejemplo, la siguiente instrucción provocaría una excepción cuando la secuencia no exista:

```

select 1 into v_exists
from user_sequences
where sequence_name = 'MY_SEQUENCE';

```

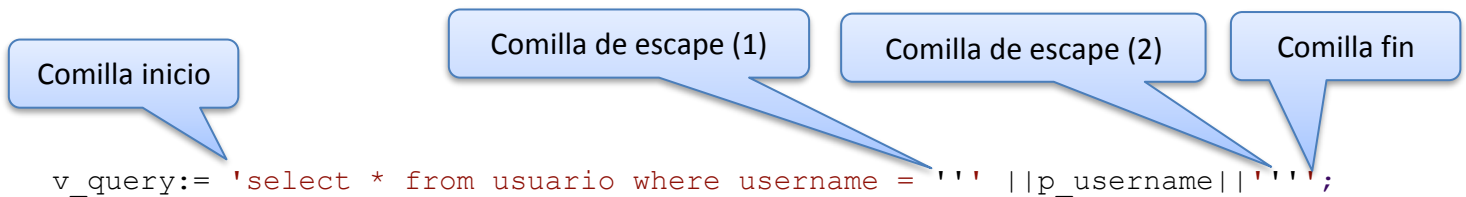
- Si la secuencia no existiera, la sentencia `select` regresaría 0 registros, y por lo tanto no podría asignarse un valor a `v_exists` y se produce excepción.
- Continuando con la explicación del programa, observar el uso de la instrucción `execute`. En su forma más básica, recibe una cadena la cual representa el código DDL a ejecutar. Como se comentó anteriormente, hasta que el programa ejecuta esta instrucción, la cadena es parseada y validada. Si todo es correcto, la sentencia DDL se ejecuta.

10.12.2. Sentencias Dinámicas con parámetros, ataques por inyección de SQL.

- Un uso común de sentencias SQL dinámicas es la posibilidad de crear sentencias empleando los valores proporcionados por los parámetros de un programa PL/SQL, por ejemplo, valores proporcionados por el usuario.

Ejemplo:

Suponer que se tiene la siguiente sentencia dinámica que obtiene los datos de un usuario con base a su username. El id es proporcionado desde una interfaz gráfica.



De la sentencia anterior se pueden comentar varias situaciones:

- Suponer que el programa PL/SQL recibe un parámetro `p_username` que contiene en nombre del usuario a buscar y su valor es `admin`.
- La sentencia SQL que debe formarse será:



- Observar que se requiere escribir comillas simples adicionales. En la sentencia anterior las comillas que han sido aumentadas en tamaño representan a las comillas simples que requiere toda sentencia SQL para indicarle al manejador que se trata de una cadena de caracteres.
- Debido a que la sentencia SQL es en sí una cadena, estas comillas deben ser escapadas por otra comilla para evitar confusión con la comilla de inicio y fin.
- Lo anterior implica un cuidado especial para evitar problemas con el uso de las comillas. Emplear este estilo de concatenación de parámetros se considera mala práctica.

Suponer ahora que el usuario no proporciona su `username`, tal vez proporciona su número de empleado que es un dato que no requiere comillas:

```
v_query := 'select * from usuario where num_empleado = ' || p_num_empleado;
```

- Un usuario malicioso pudiera escribir como numero de empleado la siguiente cadena:

```
15; drop table usuario;
```

- La sentencia SQL dinámica será:

```
v_query := 'select * from usuario where username = 15; drop table usuario;';
```

- La sentencia SQL anterior es válida. En realidad, son 2 sentencias SQL. La primera obtiene al empleado 15 y la otra provoca que *¡La tabla sea eliminada!*

Existen muchas otras variantes que pueden generar problemas y errores graves en una aplicación. En la siguiente liga se muestran algunos ejemplos: https://www.w3schools.com/sql/sql_injection.asp

- A este tipo de problemas se les conoce como **Ataque por inyección de SQL**.

- Cualquier desarrollo de aplicaciones que implique la construcción de sentencias SQL dinámicas, no solo en PL/SQL, si no en cualquier lenguaje de programación debe tener en cuenta esta situación y programar de forma correcta.
- La siguiente sección muestra la forma en la que un programa PL/SQL puede ser inmune a este tipo de ataques de seguridad.

10.12.3. Paquete `dbms_assert`

El paquete `dbms_assert` define diversas funciones que ayudan con el problema de inyección de SQL. Las funciones más comunes se muestran a continuación:

- `enquote_literal`: Recibe una cadena y le agrega una comilla al inicio y al fin.
- `simple_sql_name`: Recibe una cadena y verifica que la cadena cumpla con las reglas lexicográficas para ser considerada como un nombre de un objeto SQL válido. Algunas de las reglas que debe cumplir la cadena son:
 - Debe iniciar con un carácter alfanumérico.
 - Puede contener `_`, `#`, `$` a partir del segundo carácter.
 - La cadena puede contener la comilla invertida ``` empleada en algunos manejadores.

Estas reglas permiten filtrar código malicioso. Por ejemplo, `drop database mybd;` no cumple con las reglas anteriores, por lo que la función generará una excepción.

- `schema_name`: Recibe una cadena y verifica que su valor corresponda con un nombre válido que pueda ser asignado a un esquema dentro de la base de datos.

10.12.4. Parámetros (placeholders).

- Se refiere a la capacidad de 'parametrizar' una sentencia SQL empleando la notación `:<nombre_parametro>`.
- Estos parámetros pueden aparecer en una sentencia para ser sustituidos por algún valor, por ejemplo, en la cláusula `values` de una sentencia `insert`, para comparar valores en la cláusula `where`, etc.
- A estos parámetros se les conoce también como placeholders o bind variables.

Ejemplos:

```
select * from usuario where username = :ph_username;
insert into empleado(empleado_id,nombre) values (:ph_id, :ph_nombre);
update empleado set nombre = :ph_nombre where id = :ph_id;
```

- Observar el uso de parámetros o placeholders marcados en negritas. No se requiere hacer uso de comillas, y aunque su valor sea un código malicioso, este nunca se ejecutará como una sentencia SQL ya que se emplean exclusivamente como valores de alguna columna o condición.
- Notar que este tipo de parámetros **no pueden** ser empleados para ser sustituidos en nombres de columnas o tablas ni para sustituir alguna palabra reservada como `select`, `update`, `into`, etc.
- En los ejemplos anteriores se emplea el prefijo `ph_` para hacer énfasis de que se trata de un placeholder, empleado solo por orden y claridad del código. Se recomienda emplear esta convención.

10.12.4.1. Sustitución de parámetros de entrada.

- Los ejemplos anteriores representan parámetros de entrada cuyos valores deben ser proporcionados para que la consulta pueda ser ejecutada.
- Para sustituir los valores de cada parámetro se emplea la cláusula `using` en conjunto con la instrucción `execute immediate`.

Ejemplo:

```
v_query :=
  'insert into empleado(empleado_id,nombre,ap_paterno) '
  || ' values(:ph_id,:ph_nombre, :ph_ap_paterno) ';

execute immediate v_query using v_id,v_nombre,v_ap_pat;
```

- Observar que la sentencia dinámica contiene 3 placeholders.
- La instrucción `using` indica la lista de variables cuyos valores se emplearán para sustituir a cada placeholder en el orden de izquierda a derecha. Estas variables pueden ser parámetros de algún procedimiento, o generadas dentro del mismo programa PL/SQL.
- Como se puede observar, no existe ninguna correspondencia entre los nombres de los placeholders y los nombres de las variables empleadas para sustituir. Se relacionan únicamente por orden de aparición.
- Los placeholders heredan el tipo de dato de los parámetros especificados en `using`. Por esta razón los placeholders no requieren hacer uso de comillas.

10.12.4.2. Manejo de parámetros de salida.

- Existen consultas dinámicas que pueden regresar datos. Puede ser un solo dato o un conjunto de valores.
- La forma en la que se maneja este escenario es haciendo uso de parámetros de salida, empleando la misma sintaxis que un parámetro de entrada, pero ahora se emplea la cláusula `returning` o la cláusula `into`.

Ejemplo 1:

```
set serveroutput on

declare
  v_sql varchar2(4000);
  v_id number := 1;
  v_nombre empleado.nombre%type;
  v_puesto_id empleado.puesto_id%type;

begin

  v_sql := 'select nombre,puesto_id '
  || ' into :ph_nombre,:ph_puesto_id '
  || ' from empleado where empleado_id = :ph_id';

  execute immediate v_sql into v_nombre,v_puesto_id using v_id;
  dbms_output.put_line('Nombre del empleado: '||v_nombre);
  dbms_output.put_line('Puesto id del empleado: '||v_puesto_id);
end;
/
```

- Para sentencias `select` que regresan a lo más un registro, se emplea la cláusula `into` (similar a lo visto en cursores).
- En este ejemplo se tiene un parámetro de entrada `:ph_id` y 2 parámetros de salida `:ph_nombre` y `:ph_puestp_id`. Estos 2 parámetros son de salida ya que en ellos se asignará el valor del nombre y del puesto del empleado con `id = 1`
- Observar que en la instrucción `execute immediate` también se hace uso de `into` para asociar a los 2 parámetros de salida con las variables `v_nombre` y `v_puesto_id` respectivamente. Esta técnica es válida cuando se obtiene a lo más un registro.
- Similar al ejemplo anterior se emplea `using` para asociar al parámetro de entrada.

Ejemplo 2:

El siguiente ejemplo muestra la manera de procesar una consulta `select` dinámica que regresa más de un registro haciendo uso de un cursor.

```
set serveroutput on

declare
  v_sql varchar2(4000);
  v_id number := 1;
  v_empleado empleado%rowtype;
  d_cursor sys_refcursor;

begin

  v_sql := ' select empleado_id,nombre,puesto_id'
  || ' from empleado where empleado_id >= :ph_id';

  --se abre un cursor dinámico, se recorre y se cierra
  open d_cursor for v_sql using v_id;
  loop
    fetch d_cursor into v_empleado;
    exit when d_cursor%notfound;

    dbms_output.put_line('id del empleado: '||v_empleado.empleado_id);
    dbms_output.put_line('Nombre del empleado: '||v_empleado.nombre);
    dbms_output.put_line('Puesto id del empleado: '||v_empleado.puesto_id);

  end loop;
  close d_cursor;
end;
/
```

- Observar el tipo de dato de la variable `d_cursor sys_refcursor`. Se emplea este tipo de dato para poder hacer referencia al resultado que generará la consulta dinámica.
- Observar el tipo de dato de la variable `v_empleado` que representa a un renglón o registro de la tabla empleado.
- Observar el uso de la instrucción `open` en conjunto con la instrucción `for` y `using` empleadas para ejecutar la consulta dinámica y generar el cursor dinámico. Esas instrucciones realizan la función de `execute immediate` y generan un cursor dinámico.

- La cláusula `using` tiene la misma función que los ejemplos anteriores, empujada para sustituir los placeholders de entrada.

Ejemplo 3:

Recordando el ejemplo de manejo de objetos CLOB en el que se registra un libro con su texto, realizar un programa PL/SQL que modifique el valor del campo `texto` con la siguiente cadena 'Texto modificado'. El cambio debe aplicarse al libro con `id = 1`.



```

set serveroutput on
declare
  v_sql varchar2(4000);
  v_texto clob;
  v_nuevo_texto varchar2(20) := 'Texto modificado';
begin
  v_sql := 'update libro set texto = empty_clob()'
    || ' where libro_id = :ph_libro_id'
    || ' returning texto into :ph_texto';

  execute immediate v_sql using 1 returning into v_texto;
  dbms_lob.writeappend(v_texto, length(v_nuevo_texto), v_nuevo_texto);
end;
/

```

- En este ejemplo se ilustra el uso de `returning`. Sentencias `insert`, `update` y `delete` pueden hacer uso de esta instrucción para regresar o hacer referencia a alguna columna.
- En este ejemplo, observar que el campo `texto` es actualizado a una cadena vacía empleando la función `empty_clob()`.
- El placeholder `:ph_texto` se emplea para hacer referencia al campo `texto`. Es un placeholder de salida ya que su valor será asignado a una variable. Recordando de la sección anterior, hacer referencia a una columna CLOB, significa hacer referencia al puntero (locator) que indica el lugar físico donde se almacena la cadena.
- Observar que se usa `returning` en la instrucción `execute immediate`. En este caso, el valor del placeholder de salida `:ph_texto` será asignado a la variable `v_texto` la cual es de tipo CLOB.
- Una vez que se tiene la referencia del objeto CLOB en la variable `v_texto`, se emplea la función `writeappend` para escribir la nueva cadena. Observar que el primer parámetro de esta función es un objeto de tipo `clob` en el que se escribirá el texto. El segundo parámetro indica la cantidad de caracteres a escribir y el tercer parámetro corresponde con la cadena a escribir.
- Notar que el valor de la instrucción `using` es 1. Este valor será sustituido en el placeholder de entrada `:ph_libro_id` de forma similar a los ejemplos anteriores.

En este ejemplo se emplea SQL dinámico para poder hacer uso de la instrucción `returning` y así obtener la referencia al puntero (locator) del campo CLOB requerida para poder actualizar su valor.

Ejemplo 4

Este ejemplo pone en práctica los conceptos vistos en esta sección así como los de la sección anterior aplicado a un escenario real.

Crear un procedimiento almacenado llamado `p_guarda_objeto_blob`. Su función principal es leer el contenido de una columna BLOB que pertenece a una tabla y guardar su contenido en un archivo dentro de un directorio especificado. El procedimiento puede emplearse, por ejemplo, para exportar todas las fotos almacenadas en una tabla hacia un directorio del servidor. El procedimiento podrá ser empleado para exportar datos de cualquier tabla. Este procedimiento deberá recibir los siguientes parámetros:

- Parámetro de entrada que indica el nombre del objeto `directory` en el que se debe escribir el contenido del archivo.
- Parámetro de entrada que indique el nombre del archivo.
- Parámetro de entrada que indica el nombre de la tabla donde se encuentra la columna BLOB.
- Parámetro de entrada que indica el nombre de la columna BLOB.
- Parámetro de entrada que indica el nombre de la columna que será empleada como llave primaria para localizar al registro en cuestión. Por simplicidad se asume que la tabla tiene una llave primaria numérica de un solo campo.
- Parámetro de entrada numérico que indica el valor de la llave primaria.
- Por ejemplo, exportar el objeto `blob` la columna `foto`, que pertenece a la tabla `empleado` que tiene como llave primaria al campo `foto_id` con valor 1. El objeto será almacenado en el directorio al que apunta el objeto `directory` creado en la base de datos llamado `export_data`. Guardar la foto con el nombre `foto.jpg`
- Parámetro numérico de salida que indique el número de bytes que se escribieron, o -1 en caso de no haber realizado la exportación debido a la ocurrencia de algún error.

El procedimiento deberá ser inmune a ataques de inyección de SQL.

Solución:

- Se asume la existencia de la tabla `libro_blob` que contiene un registro con `id = 1`. El contenido del campo `version_digital` corresponde a un archivo pdf.
- El usuario que ejecuta el procedimiento tiene privilegios de lectura y **escritura** sobre el objeto `directory` llamado `DIR_DATA`. Es decir, las siguientes líneas deben ser ejecutadas empleando al usuario SYS:

```
create or replace directory data_dir as '/tmp/data_dir';
grant read,write on directory data_dir to <usuario>;
```

El código del procedimiento almacenado se muestra a continuación.

```
set serveroutput on
create or replace procedure guarda_objeto_blob(
  v_nombre_directorio in varchar2,
  v_nombre_archivo     in varchar2,
  v_nombre_tabla       in out varchar2,
  v_nombre_col_blob    in out varchar2,
  v_nombre_col_pk      in out varchar2,
  v_valor_pk           in number,
  v_longitud           out number
) is
```

```
v_sql varchar2(2000);
v_blob blob;

v_file utl_file.FILE_TYPE;
v_buffer_size number :=32767;
v_buffer RAW(32767);
v_position number := 1;

begin
    --verifica que los nombres de las tablas y columnas sean
    --cadenas validas. Ayuda con la inyección de SQL.
    v_nombre_tabla := dbms_assert.simple_sql_name(v_nombre_tabla);
    v_nombre_col_blob := dbms_assert.simple_sql_name(v_nombre_col_blob);
    v_nombre_col_pk := dbms_assert.simple_sql_name(v_nombre_col_pk);

    v_sql := 'select '
        || v_nombre_col_blob
        || ' into :ph_blob'
        || ' from '
        || v_nombre_tabla
        || ' where '
        || v_nombre_col_pk
        || ' = :ph_pk';

    --ejecuta la consulta dinamica
    execute immediate v_sql into v_blob using v_valor_pk;

    --abre el archivo para escribir
    v_file := utl_file.fopen(upper(v_nombre_directorio),
        v_nombre_archivo,'wb',v_buffer_size);

    v_longitud := dbms_lob.getlength(v_blob);

    --lee el archivos por partes hasta completar
    while v_position < v_longitud loop
        dbms_lob.read(v_blob,v_buffer_size,v_position,v_buffer);
        utl_file.put_raw(v_file,v_buffer,true);
        v_position := v_position+v_buffer_size;
    end loop;
    utl_file.fclose(v_file);

    -- cierra el archivo en caso de error y relanza la excepción.
    exception
    when others then
        --cerrar v_file en caso de error.
        if utl_file.is_open(v_file) then
            utl_file.fclose(v_file);
        end if;
        --muestra detalle del error
        dbms_output.put_line(dbms_utility.format_error_backtrace);
        --relanza la excepcion para que sea manejada por el
        --programa que invoque a este procedimiento.
        v_longitud := -1;
        raise;
end;
/
show errors
```


- Observar el tipo de dato de la variable `v_file`, empleada para hacer referencia al archivo en el que se realizará la escritura.
- Las variables `v_buffer`, `v_buffer_size` y `v_position` se emplean para almacenar el contenido de la columna BLOB en un buffer para posteriormente escribir al archivo.
- Observar el uso de la función `simple_sql_name`, ayuda con la detección de inyección de SQL para los atributos que corresponden con nombres de tablas o de columnas. Observar que se reasignan sus valores con la cadena que regresa la función. Por esta razón es que estos 3 parámetros están declarados como parámetros de entrada y de salida (`in out`).
- Observar la construcción de la sentencia SQL dinámica. Al emplear la función `simple_sql_name` es seguro concatenar su valor. Notar que se hace uso de un placeholder de salida `:ph_blob` y otro más de entrada `:ph_pk`. El primero se emplea para obtener la referencia de la columna blob cuyo contenido va a ser leído, y el segundo para ser sustituido en la condición `where` como valor de la llave primaria y así poder localizar al registro de interés.
- Ambos placeholders son asociados a las variables `v_blob` y `v_valor_pk` en la instrucción `execute immediate`.
- Las siguientes líneas realizan la apertura del archivo para escribir el contenido de la columna BLOB que es representada por la variable `v_blob`.
- La función `dbms_lob.read` se encarga de leer el contenido del blob. Observar que la lectura se hace por partes empleando un buffer con una capacidad finita de 32767 bytes. Esta técnica permite leer una cantidad controlada de bytes evitando que el programa sature la memoria para columnas BLOB muy grandes. En el ciclo `while` se realizan N lecturas, tantas como sean necesarias. La variable `v_buffer_size` es un parámetro `in out` que indica la cantidad de bytes máxima a leer, y una vez que se ha leído se actualiza su valor para reflejar el número real de bytes leídos.
- Finalmente, se realiza el manejo de posibles errores.

El siguiente programa anónimo muestra el uso del procedimiento anterior.

```

set serveroutput on
declare
  v_longitud number;
  v_nombre_tabla varchar2(30) := 'libro_blob';
  v_nombre_col_blob varchar2(30) := 'version_digital';
  v_nombre_col_pk varchar2(30) := 'libro_id';
  v_nombre_directorio varchar2(30) := 'DATA_DIR';
  v_valor_pk numeric(10,0) := 1;
  v_nombre_archivo varchar2(100) := 'test.pdf';
begin
  dbms_output.put_line('exportando objeto blob');
  guarda_objeto_blob(v_nombre_directorio,v_nombre_archivo,
    v_nombre_tabla,v_nombre_col_blob,v_nombre_col_pk,
    v_valor_pk,v_longitud);
  dbms_output.put_line('Listo, bytes escritos: '|| v_longitud);
  --hace permanente el cambio
  commit;
exception
  when others then
    dbms_output.put_line('Error al exportar, se hara rollback');
    dbms_output.put_line(dbms_utility.format_error_backtrace);
    rollback;
    raise;
end;
/

```

Ejemplo 5

Realizar un último ejemplo empleando la misma estrategia, pero ahora el procedimiento realizará la operación contraria: leer un archivo binario y almacenar su contenido en una columna de un registro existente. Este ejemplo se realizó en secciones anteriores, pero en este caso, los nombres de las columnas y el de la tabla se parametrizan para hacerlo dinámico.

Crear un procedimiento llamado `carga_blob_en_bd` que reciba los siguientes parámetros:

- Parámetro de entrada que indica el nombre del objeto `directory` en el que encuentra el archivo binario a leer.
- Parámetro de entrada que indique el nombre del archivo.
- Parámetro de entrada que indica el nombre de la tabla donde se encuentra la columna BLOB.
- Parámetro de entrada que indica el nombre de la columna BLOB.
- Parámetro de entrada que indica el nombre de la columna que será empleada como llave primaria para localizar al registro en cuestión. Por simplicidad se asume que la tabla tiene una llave primaria numérica de un solo campo.
- Parámetro de entrada numérico que indica el valor de la llave primaria.
- Parámetro numérico de salida que indique el número de bytes que se escribieron, o -1 en caso de no haber realizado la exportación debido a la ocurrencia de algún error.

Solución:

```
set serveroutput on
```

```
create or replace procedure carga_blob_en_bd (
  v_nombre_directorio in varchar2,
  v_nombre_archivo     in varchar2,
  v_nombre_tabla       in out varchar2,
  v_nombre_col_blob    in out varchar2,
  v_nombre_col_pk      in out varchar2,
  v_valor_pk           in number,
  v_longitud            out number
) is
  v_sql varchar2(2000);
  v_bfile bfile;
  v_src_offset number := 1;
  v_dest_offset number:= 1;
  v_dest_blob blob;
  v_src_length number;
  v_dest_length number;

begin
  --verifica que los nombres de las tablas y columnas sean
  --cadenas validas. Ayuda con la inyección de SQL.
  v_nombre_tabla := dbms_assert.simple_sql_name(v_nombre_tabla);
  v_nombre_col_blob := dbms_assert.simple_sql_name(v_nombre_col_blob);
  v_nombre_col_pk := dbms_assert.simple_sql_name(v_nombre_col_pk);
```

```

v_bfile := bfilename(upper(v_nombre_directorio),v_nombre_archivo);

if dbms_lob.fileexists(v_bfile) = 1 and not
  dbms_lob.isopen(v_bfile) = 1 then
  dbms_lob.open(v_bfile,dbms_lob.lob_readonly);
else
  raise_application_error(-20001,'El archivo '
    ||v_nombre_archivo
    ||' no existe en el directorio '
    ||v_nombre_directorio
    ||' o el archivo esta abierto');
end if;

--sentencia dinamica.

v_sql := 'update '
  ||v_nombre_tabla
  ||' set '
  ||v_nombre_col_blob
  ||' = empty_blob()'
  ||' where '
  ||v_nombre_col_pk
  ||' = :ph_pk'
  ||' returning '
  ||v_nombre_col_blob
  ||' into :ph_blob';

--ejecuta la consulta dinamica
execute immediate v_sql using v_valor_pk returning into v_dest_blob;

--escribe el contenido del archivo en el objeto blob: v_dest_blob
dbms_lob.loadblobfromfile(
  dest_lob    => v_dest_blob,
  src_bfile   => v_bfile,
  amount      => dbms_lob.getlength(v_bfile),
  dest_offset => v_dest_offset,
  src_offset  => v_src_offset);
dbms_lob.close(v_bfile);

v_src_length := dbms_lob.getlength(v_bfile);
v_dest_length := dbms_lob.getlength(v_dest_blob);

if v_src_length = v_dest_length then
  dbms_output.put_line('Escritura correcta, bytes escritos: '
    || v_src_length);
  v_longitud := v_src_length;
else
  raise_application_error(-20002,'Error al escribir datos.\n'
    ||' Se esperaba escribir '
    ||v_src_length
    ||' Pero solo se escribio '
    ||v_dest_length);
end if;

```

```

exception
  when others then
    v_longitud := -1;
    dbms_output.put_line( dbms_utility.format_error_backtrace );
    raise;
end;
/
show errors

```

- Observar el uso de `returning` en la sentencia SQL dinámica. Se emplea una sentencia `update` para actualizar el valor de la columna `blob` a un `blob` vacío. Este `update` permite obtener la referencia (locator) de la columna `blob` a través de la instrucción `returning`. Esta referencia servirá para actualizar su valor empleando la función `loadblobfromfile`.
- Observar que se tiene un placeholder de salida `:ph_blob` empleado para obtener la referencia al objeto `blob`.
- En la instrucción `execute immediate` se asocia el placeholder `:ph_blob` a la variable `v_dest_blob` haciendo uso nuevamente de `returning`.

Finalmente, el programa empleado para invocar al procedimiento es muy similar al anterior:

```

set serveroutput on
declare
  v_longitud number;
  v_nombre_tabla varchar2(30) := 'libro_blob';
  v_nombre_col_blob varchar2(30) := 'version_digital';
  v_nombre_col_pk varchar2(30) := 'libro_id';
  v_nombre_directorio varchar2(30) := 'DATA_DIR';
  v_valor_pk numeric(10,0) := 61;
  v_nombre_archivo varchar2(100) := 'test.pdf';

begin

  dbms_output.put_line('Cargando BLOB en la BD');

  carga_blob_en_bd(v_nombre_directorio,v_nombre_archivo,
                  v_nombre_tabla,v_nombre_col_blob,v_nombre_col_pk,
                  v_valor_pk,v_longitud);

  dbms_output.put_line('Listo, bytes escritos: ' || v_longitud);
  --hace permanente el cambio
  commit;
  exception
  when others then
    dbms_output.put_line('Error al realizar la carga, se hara rollback');
    dbms_output.put_line( dbms_utility.format_error_backtrace );
    rollback;
    raise;
end;
/

```