

ORACLE12c. LENGUAJE PL/SQL

Autor: CLEformación S.L

Localidad y año de impresión: Madrid, 2016

Copyright: CLEformación

Oracle Designer, Oracle Reports, PL/SQL, SQL*Plus, Oracle Enterprise Manager son marcas registradas por Oracle Corporation

Windows, Visual Basic son marcas registradas por Microsoft Corporation.

Capítulo 1. El lenguaje PL/SQL

Introducción	5
Ventajas del lenguaje PL/SQL	5
Soporte al lenguaje SQL.....	5
Soporte a la programación orientada a objetos.	5
Mejor rendimiento	5
Total portabilidad	6
Integración con SQL	6
Gran Seguridad	6
Conceptos Básicos	7
Conjunto de caracteres.....	7
Unidades Léxicas	8
Delimitadores.....	8
Identificadores	9
Literales	9
Comentarios.	10
Normas de nombrado	11
Expresiones y Comparaciones	11

Capítulo 2. Tipos de Bloques

Introducción	1
Bloque anónimo	1
Bloque nominado (almacenado).....	3
Estructura.....	3
Declaraciones	4
Cuerpo del Bloque	6

Capítulo 3. Tipos de Datos

Introducción	1
Tipos de datos predefinidos.....	1
Tipos numéricos	1
Tipos Carácter	5
Tipos LOB	7
Tipos Booleanos	8
Tipos Fecha e Interval	8
Subtipos definidos por el usuario	12
Conversión de datos.	13
Visibilidad y Ámbito.	13

Capítulo 4. Estructuras de Control

Introducción	1
Control condicional.....	1
IF-THEN-ELSE	1
CASE	3
Control de iteraciones. (Bucles)	5
LOOP.....	5
WHILE-LOOP	6
FOR-LOOP	6
NULL.....	8

Capítulo 5. Cursores

Introducción	1
Manipulación de Datos	1
Control de Transacciones.....	1
Funciones SQL	1
Pseudocolumnas SQL.....	1
Operadores SQL.....	2
Creación y manipulación de cursores	3
Declaración de un cursor.....	3
Apertura de un cursor.....	3
Lectura de un cursor.....	4
Cierre de un cursor.....	4
FOR de Cursor.	5
Atributos del Cursor	5
Cursor FOR UPDATE.....	6

Capítulo 6. Tratamiento de Errores

Introducción	1
Excepciones predefinidas	1
Ámbito de una excepción en PL/SQL.....	5
Propagación de las excepciones	5
Excepciones definidas por el usuario.....	8
Declaración de una excepción	8
Levantar excepciones con la sentencia RAISE.....	8
Asignar excepciones a errores Oracle	8
Definir mensajes de error de usuario	9
Recuperación del Código de Error y el Mensaje.....	10

Capítulo 7. Procedimientos y Funciones

Introducción	1
Ventajas de los subprogramas.	1
Procedimientos.....	1
Creación y Modificación.....	1
Ejecución	4
Eliminar un procedimiento	4
Funciones.....	5
Ejecución	6
Eliminar una Función	6
Efectos colaterales de las funciones.	6
Privilegios.....	7

Capítulo 8. Paquetes

Introducción	1
Crear o modificar un Paquete	1
Ventajas de los paquetes PL/SQL	4
Modularidad	4
Facilidad en el Diseño de la Aplicación	4
Ocultamiento de la Información	4
Funcionalidad Agregada.....	4
Mejora Ejecución	4
Sobrecarga de subprogramas en paquetes	5
Paquetes definidos por Oracle	10
STANDARD	10
DBMS_OUTPUT.....	11
UTL_FILE.....	11

Capítulo 9. Colecciones y Registros

Introducción	1
Colecciones	1
Tablas Indexadas versus Tablas Anidadas	1
Varrays versus Tablas Anidadas	2
Definición y Declaración de Colecciones	2
Inicializar Colecciones	4
Referenciar Colecciones.....	4

Asignación de Elementos a una Colección	5
Métodos para Colecciones	5
Comparación Completa de Colecciones	8
Mejorar el rendimiento mediante acciones BULK BINDING.	10
Registros	12

Capítulo 10. Disparadores

Crear un disparador	2
Disparador DML simple.	3
Funciones Booleanas	5
Pseudo Registros :NEW y :OLD.....	6
Trigger INSTEAD OF	8
Disparadores DML Compuestos	10
Restricciones de los disparadores compuestos	12
Disparadores No DML	12
Evento DDL	13
Evento de base de datos.	15

Capítulo 11. Cursores Variables

Introducción	1
Utilización de cursores variables	1
Definición y Declaración de cursores variables	2
Definición	2
Declaración	3
Control de cursores variables.....	4
Abrir un Cursor Variable.	4
Recuperar desde un Cursor Variable	6
Cerrar un Cursor Variable.....	7
Expresiones de Cursor	7
Manipulación de Expresiones de Cursor en PL/SQL	8
Uso de una Expresión de Cursor como Parámetro en Unidades PL/SQL.....	9
Restricciones de los cursores variables.....	10
Beneficios de los cursores variables	11

Capítulo 12. SQL Dinámico

Introducción	1
EXECUTE IMMEDIATE	2
Recuperación de varias filas.	3
El paquete DBMS_SQL.	4
Flujo de ejecución.....	5
Ventajas e inconvenientes de ambos métodos.....	6
Ventajas de SQL Nativo (EXECUTE IMMEDIATE).....	6
Ventajas de DBMS_SQL.	6

Anexo 0. Ejercicios de PL/SQL

Modelo de datos	1
Bloques Anónimos.....	3
Cursores.....	3
Excepciones.....	3
Funciones y Procedimientos.....	4
Paquetes.....	5
Colecciones	5
Disparadores.....	5
Cursores variables	6
SQL dinámico	6
Soluciones.....	7
Bloques Anónimos.....	7
Cursores	9
Excepciones	11
Funciones y Procedimientos	13
Paquetes.....	17
Colecciones	20
Disparadores	21
Cursores Variables	21
SQL Dinámico.....	22

El Lenguaje PL/SQL

Tabla de contenidos

Introducción	1
Ventajas del lenguaje PL/SQL	1
Soporte al lenguaje SQL.....	1
Soporte a la programación orientada a objetos.....	1
Mejor rendimiento	1
Total portabilidad	1
Integración con SQL	2
Gran Seguridad	2
Conceptos Básicos	3
Conjunto de caracteres.....	3
Unidades Léxicas.....	3
Delimitadores.....	4
Identificadores	5
Literales.....	5
Comentarios.....	6
Normas de nombrado.....	7
Expresiones y Comparaciones	7

Introducción



El lenguaje PL/SQL ofrece todas las ventajas de los lenguajes de programación como la encapsulación de datos, definición de objetos, manejo de excepciones y ocultación de información sensible. Al estar integrado en el núcleo Oracle ofrece también el acceso a la información mediante comandos SQL, portabilidad y seguridad.

Ventajas del lenguaje PL/SQL

Soporte al lenguaje SQL.

SQL se ha convertido en el lenguaje estándar de las bases de datos por su flexibilidad, potencia y facilidad de uso y aprendizaje. PL/SQL permite utilizar todas las funciones, operadores, pseudo columnas y tipos de datos de SQL.

Permite también utilizar “SQL Dinámico”, una avanzada técnica que permite flexibilizar más las aplicaciones.

Soporte a la programación orientada a objetos.

PL/SQL permite la encapsulación de operaciones y datos con lo que se pueden crear componentes que sean modulares, de fácil mantenimiento y reutilizables. También permite ocultar los detalles de los objetos y cambiarlos de tal manera que no afecten a los programas clientes.

Mejor rendimiento

PL/SQL puede enviar un bloque de comandos al servidor Oracle reduciendo de esta manera el tráfico de red. Los procedimientos almacenados son compilados una sola vez y guardados en la base de datos en forma compilada; además una vez que son invocados, éstos se guardan en la caché y son compartidos por todos los usuarios. De esta manera PL/SQL reduce el tráfico de red, los requerimientos de memoria y el tiempo de invocación.

Total portabilidad

PL/SQL es compatible con cualquier sistema operativo o plataforma donde se esté ejecutando un servidor Oracle.



Integración con SQL

PL/SQL está totalmente integrado con SQL. Soporta todos los tipos de datos de SQL así como el valor Nulo.

Los atributos %TYPE y %ROWTYPE permiten definir variables en base a las columnas de las tablas Oracle. Si variara el tipo de la columna, el procedimiento PL/SQL utilizaría la nueva definición sin tener que ser modificado.

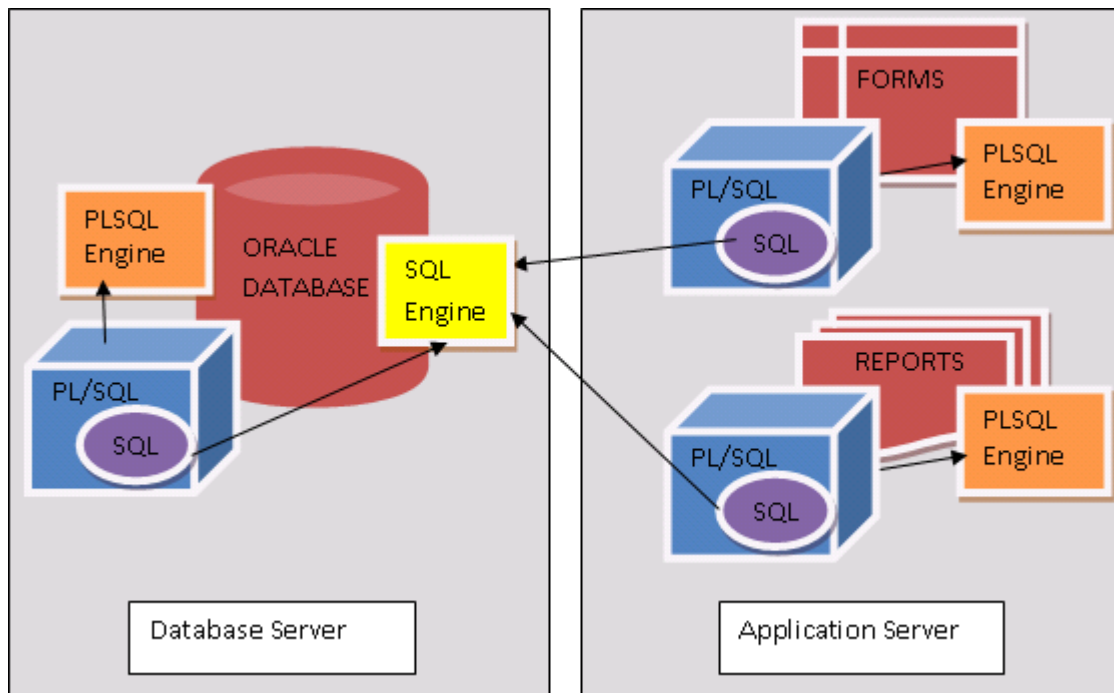
Gran Seguridad



Los procedimientos almacenados permiten dividir la aplicación entre la parte cliente y la parte servidor. De esta manera la parte cliente no puede manipular información sensible. Los disparadores dan la posibilidad de Auditoria.

Permite limitar el acceso a la información de tal manera que sólo sea manipulable a través de procedimientos. Ejemplo: Un usuario puede acceder a la información de una tabla mediante un procedimiento que acceda a ella, pero no puede acceder a la tabla mediante comandos SQL normales por carecer de privilegios sobre la tabla.

El lenguaje PL/SQL se halla también en las herramientas de Oracle con lo que ofrece las prestaciones de cálculos y procesamiento en la parte del cliente de estas herramientas reduciendo el tráfico de red.



Conceptos Básicos

Como en cualquier lenguaje de programación PL/SQL tiene también reglas de sintaxis, palabras reservadas, tipos de datos propios, puntuación, etc.

Conjunto de caracteres



Utilizar:

- Letras del alfabeto inglés
- Números
- Los símbolos `()+*/<>=!~^;::'%, "#$&_ |{}?[]`, espacio, tabulación y retorno de carro.

PL/SQL no distingue entre mayúsculas y minúsculas, excepto cadenas y caracteres literales (texto entre comillas).

Unidades Léxicas

Una línea de texto PL/SQL contiene grupos de caracteres conocidos como unidades léxicas, las cuales pueden ser clasificadas en:

- Delimitadores (símbolos simples o compuestos)
- Identificadores
- Literales
- Comentarios

El espacio separa unidades léxicas, generando un error por unir o separar con espacio estas unidades:

```
IF x > y THEN high:=x; ENDIF; /* es incorrecto, ENDIF está unido, END IF; es la forma correcta*/
```

La asignación (:=) no puede contener espacios que lo separen:

```
X : = X +1; /* es incorrecto, :=, está separada por un espacio.*/
```

Estas dos estructuras de IF, son correctas aunque una no tenga retorno de carro para finalizar cada sentencia y la otra si lo tenga.

<pre>IF x>y THEN v_max:=x;ELSE v_max:=y;END IF;</pre>	<pre>IF x>y THEN v_max:=x; ELSE v_max:=y; END IF;</pre>
--	--

Delimitadores

Los delimitadores son símbolos simples o compuestos que tienen un significado para PL/SQL. Por ejemplo se utilizan delimitadores para expresar una operación aritmética o lógica. El espacio se utiliza para dar una mayor legibilidad al programa PL/SQL.

Una lista de los delimitadores más comunes:

+	Operador de suma.
/	Operador de división.
*	Operador de multiplicación.
-	Operador de resta / negación
=	Operador de relación (Igualdad)
<	Operador de relación (Menor que)
>	Operador de relación (Mayor que)
;	Delimitador de final de sentencia.
"	Identificador de literal.
:	Identificador de Host Variable
:=	Operador de asignación.

	Operador de concatenación.
**	Operador de potencia. (X elevado a Y)
/*	Delimitador de inicio de comentario de varias líneas
*/	Delimitador de fin de comentario de varias líneas.
--	Delimitador de comentario de una sola línea.
..	Operador de rango

Identificadores

Los identificadores dan nombre a las variables, constantes, cursores, variables de cursor, subprogramas, paquetes y excepciones.

Los identificadores deben comenzar por una letra y puede ser seguida por números, letras, signos dólar (\$), guiones bajos (_) y el signo numérico (#). Cualquier otro signo provocará un error.

Se pueden utilizar indistintamente las mayúsculas y las minúsculas. Los signos dólar, guión bajo y signo numérico cuentan a la hora de diferenciar un identificador.

La longitud máxima de un identificador es de 30 caracteres. Es aconsejable que los identificadores tengan significado para una mayor legibilidad en los programas.

X	Correcto.
Total\$	Correcto.
Dep_Trabajo	Correcto.
1980Total	Incorrecto por empezar con un número.
Nombre-Jefe	Incorrecto por llevar un guión.
Si/No	Incorrecto por la barra.
IdPedido	Correcto.
idpedido	Correcto pero es igual que IdPedido.
IDPEDIDO	Correcto pero es igual a los 2 anteriores
ID_PEDIDO	Correcto y diferente a los 3 anteriores.

Existe también un conjunto de palabras llamadas palabras reservadas que no pueden ser utilizadas, como las palabras BEGIN y END, que forman parte del bloque o subprograma y son reservadas. Hay palabras reservadas en PL/SQL que no lo son pero se desaconseja su uso para evitar errores de compilación.

Literales

Un literal es un número, una cadena, un carácter o un valor Booleano que no está representado por un identificador. Es un valor constante.

- Literales numéricos: Representan números y pueden ser de dos tipos:
 - Enteros. Pueden tener signo y son representados sin punto decimal
 - Reales. Pueden tener signo y se representan con un punto decimal.

El único carácter que se puede utilizar en los literales es la E, la cual significa “potencia de”.

El rango de valores de los números es de 1E-130 hasta 10E125.

```
12          Entero
-34         Entero
67.7        Real
-927.       Real          /*Aunque su valor sea entero*/
1e28        Real
1893e130    Incorrecto por salir del rango admitido
```

- Literales de carácter y cadena de caracteres.
 - Son colecciones de 1 o más caracteres que están entre comillas simples (' '). Pueden contener todos los caracteres de PL/SQL y sí que son sensibles a las mayúsculas y minúsculas.

```
'S'
'El pedido ha sido procesado'
'Introduzca S/N'
'Entrada'
'ENTRADA'          --Este literal es diferente al superior
```

- Literales Booleanos
 - Existen tres valores, no cadenas, para los literales booleanos.

```
TRUE
FALSE
NULL
```

- Literales de tipo fecha.
 - Los literales de tipo fecha dependen de la base de datos.

```
Dia          DATE := DATE '18-04-2000';
Fecha2       TIMESTAMP := TIMESTAMP '2002-02-20 15:01:01';
Fecha3       TIMESTAMP WITH TIME ZONE := TIMESTAMP '2002-01-31 19:26:56.66
+02:00';
Fecha4       INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;
```

Comentarios.

Los comentarios son necesarios para poder documentar un programa PL/SQL. Permiten clarificar partes de código para una posible revisión posterior del código. PL/SQL ofrece 2 tipos de comentarios.

- Comentarios de línea.

Para comentar una línea o parte de una línea se utiliza dos guiones (--).

```
-- Principio del proceso

SELECT      salary --seleccionamos el salario
INTO        v_salario  --se almacena en la variable v_salario
FROM        employees  --el campo salario está en la tabla empleados
WHERE       employee_id = 100; --solo del empleado que cumpla la condición

Bono := v_salario * 0.15;      -- calculamos el 15% del salario
```

- Comentarios multilinea o de varias líneas.

Para iniciar el bloque de comentarios PL/SQL utiliza (/*) y para acabar el bloque de líneas comentadas (*).

```

/* Se obtiene el salario de un empleado */
SELECT      salary
INTO        v_salario
FROM        employees
WHERE       employee_id = 100;
/*
El bono es un 15% del salario más el coeficiente que ha sido calculado en la
función coeficiente_global.
*/
bono := (v_salario * 0.15) + coeficiente;

```

Normas de nombrado

Las siguientes convenciones se aplican a todos los objetos PL/SQL. Es decir, a las variables, constantes, nombres de cursores, programas y subprogramas. Los nombres pueden ser de 4 tipos:

Ejemplo: Para invocar al procedimiento actualiza_pedido

- Simples. Se invoca al procedimiento por su nombre.

```

Actualiza_Pedido;           -- procedimiento sin parámetros
Modificar_salario(num_emple); -- procedimiento con un parámetro

```

- Cualificado. Se antepone el nombre, en este ejemplo, del paquete que contiene este procedimiento.

```
Tramite_Pedido.Actualiza_Pedido;
```

- Remoto. Se pospone la base de datos remote detrás del procedimiento. Se accede a la base de datos remota mediante un database link y es donde se encuentra el procedimiento.

```
Actualiza_Pedido@remota;
```

- Cualificado y remoto. Es la suma de las 2 anteriores. Se referencia el paquete, el procedimiento y, mediante el indicador de acceso remoto, la base de datos remota.

```
Tramite_Pedido.Actualiza_Pedido@remota;
```

Expresiones y Comparaciones

PL/SQL evalúa el resultado de una operación examinando la expresión y el contexto de ésta. En las expresiones u operaciones pueden intervenir variables, constantes, literales y funciones.

Ejemplo:

```
Resultado := Salario * 100 / Constante15;
```

En la siguiente figura se muestra el orden de preferencia (de mayor a menor) de los diversos operadores que pueden aparecer en una expresión u operación.

Operador	Operación
**	Exponenciación (A elevado a B)

+, -	Identidad, negación
*, /	multiplicación, división
+, -, 	Suma, resta, concatenación
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparación
NOT	Negación Lógica
AND	Conjunción (A y B)
OR	Inclusión (A o B)

El orden de las operaciones se puede alterar mediante los paréntesis.

Se recomienda el uso de los paréntesis; para que de una manera explícita, quede claro lo que se quiere.

En la siguiente tabla se muestra el comportamiento de los operadores lógicos. NOT es un operador único.

x	y	X AND y	X OR y	NOT x
VERDADERO	VERDADERO	<i>VERDADERO</i>	<i>VERDADERO</i>	<i>FALSO</i>
VERDADERO	FALSO	<i>FALSO</i>	<i>VERDADERO</i>	<i>FALSO</i>
VERDADERO	NULO	<i>NULO</i>	<i>VERDADERO</i>	<i>FALSO</i>
FALSO	VERDADERO	<i>FALSO</i>	<i>VERDADERO</i>	<i>VERDADERO</i>
FALSO	FALSO	<i>FALSO</i>	<i>FALSO</i>	<i>VERDADERO</i>
FALSO	NULO	<i>FALSO</i>	<i>NULO</i>	<i>VERDADERO</i>
NULO	VERDADERO	<i>NULO</i>	<i>VERDADERO</i>	<i>NULO</i>
NULO	FALSO	<i>FALSO</i>	<i>NULO</i>	<i>NULO</i>
NULO	NULO	<i>NULO</i>	<i>NULO</i>	<i>NULO</i>

Tan pronto que PL/SQL determina el resultado de una expresión no continúa evaluando la expresión entera.

```
BEGIN
  ...
  valor1 :=10;
```

```
...  
IF (valor1 > 5) OR ( puntos = (valor6 / valor8 ) + 10)  
THEN  
...  
END IF;  
END;  
/
```

En esta expresión al cumplirse la primera parte no se seguirá evaluando las siguientes partes.

- Operador IS NULL

Devuelve TRUE (Verdadero) si la variable o expresión es NULL (NULO)

```
IF aumento IS NULL THEN ...
```

Su negación sería IS NOT NULL.

Si preguntamos por un campo o variable; si es o no nulo, utilizamos IS NULL, IS NOT NULL. No utilizar operadores de igualdad, desigualdad =, !=.

- Operador LIKE

Se utiliza para comparar un carácter, cadena o CLOB con un patrón. En caso de coincidencia devolverá el valor TRUE (VERDADERO). Existen dos caracteres comodines: (%) -porcentaje- para sustituir 0 ó más caracteres y (_) -guión bajo- para sustituir 1 solo carácter.

```
Nombre LIKE 'Naom%'
```

Puede usarse NOT LIKE.

- Operador BETWEEN

Compara si un valor dado está entre el rango más bajo y el rango más alto.

```
interes BETWEEN 2.557 and 3.234
```

Su negación NOT BETWEEN.

Equivale a un >= and <=. Ej: interes >=2.557 and interes<=3.234

- Operador IN

Devolverá TRUE si el valor está en una lista suministrada. La lista puede contener valores nulos pero serán desechados. Puede utilizarse también el operador NOT para negar los elementos de la lista.

```
valor IN (40,50,54,67,20,84)
```

Su negación NOT IN

Equivalente a un OR.

Ej:

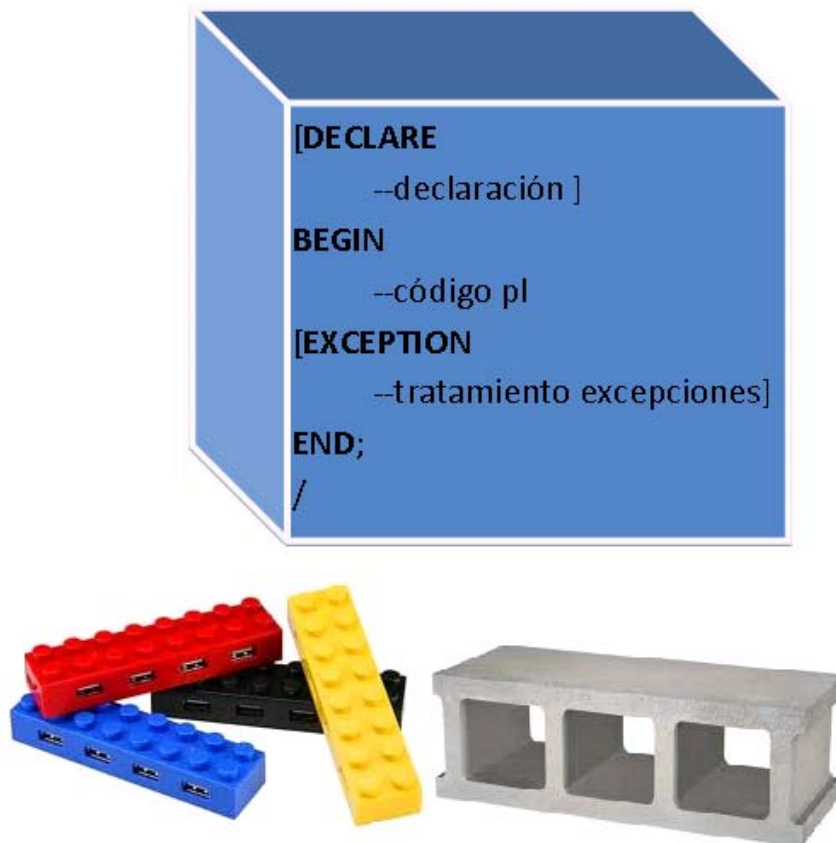
```
valor=40 or valor=50 or valor=54 or valor=67 or valor=20 or valor=84.
```


Tipos de Bloques

Tabla de contenidos

Introducción	1
Bloque anónimo	1
Bloque nominado (almacenado).....	3
Estructura.....	3
Declaraciones.....	4
Restricciones sobre declaraciones:.....	5
Variables y constantes	6
Cuerpo del Bloque	7
Estructuras de control.....	7
Excepciones.....	7

Introducción



Con PL/SQL se pueden utilizar sentencias SQL para manipular datos y estructuras de control para poder interactuar con ellos. Se pueden declarar variables y constantes, definir procedimientos y funciones. De esta manera se puede aunar todo el poder de SQL y la capacidad procedural de un lenguaje.

En PL/SQL cada variable, constante, función o parámetro tiene su tipo o *datatype*, el cual define su rango de valores, su formato y sus limitaciones. PL/SQL cuenta con una serie de formatos predefinidos y permite, también, definir tipos de datos propios del usuario.

PL/SQL es un lenguaje estructurado en bloques. Cada bloque puede contener a su vez otros sub-bloques y así sucesivamente. Cada parte de un bloque o sub-bloque resuelve, normalmente, un problema o un sub-problema. PL/SQL utiliza la táctica del “divide y vencerás” para la resolución de problemas más complejos.

Un bloque o sub-bloque relaciona lógicamente declaraciones y comandos. Las declaraciones son propias de ese bloque y dejan de existir cuando el bloque finaliza.

Bloque anónimo

Un bloque que no tiene nombre se considera anónimo. Este bloque no se guarda en la base de datos y se analizará cada vez que sea invocado.

El bloque queda en el Buffer de memoria de la aplicación en donde se esté ejecutando el programa, y por tanto se pierde al cerrar la sesión de trabajo.

Si queremos recuperarlo, lo almacenamos en un fichero de tipo texto, en nuestro sistema operativo. Pero no es un Objeto Oracle, y por tanto no sabe que existe.

Cada vez que queremos ejecutarlo lo haremos de dos formas:

- Copiamos el texto del bloque y lo pegamos en la consola, para ejecutarlo directamente.
- Lo ejecutamos como un script, dando el nombre y la dirección donde se encuentra el fichero que almacena el código.

Por ejemplo en SQL*PLUS el comando es:

```
start "direccion\nombrecompleto"
-- o también
@ "direccion\nombrecompleto"
```

Sintaxis de bloque anónimo es:

```
[DECLARE
    Declaracion de variables]
BEGIN
    Ejecucion de comandos y sentencias
[EXCEPTION
    Control de excepciones]
END;
/
```

Se construyen bloques anónimos para realizar scripts de visualización de datos, para procesar actividades que se van a ejecutar una sola vez.

Se pueden construir, para realizar un programa cuantos sub-bloques nos sean necesarios, tanto secuenciales como anidados.

```
DECLARE
    ...
BEGIN
    ...
    DECLARE
        ...
    BEGIN
        ...
    EXCEPTION
        ...
    END;
    ...
    BEGIN
        ...
        BEGIN
            ...
        END;
        ...
    EXCEPTION
        ...
    END;
EXCEPTION
    WHEN ... THEN
        ...
        DECLARE
            ...
        BEGIN
            ...
        EXCEPTION
            ...
        ...
```

```
END;  
...  
END;  
/
```

Posteriormente se analizará la visibilidad de las variables.

El carácter, / -slash-; al final del bloque PLSQL indica el fin de todo el bloque para que sea compilado.

Bloque nominado (almacenado)

Un código PL/SQL se dice que está almacenado, porque cuando lo compilas, el motor de Oracle lo guarda como un componente más de la Base de Datos, y por tanto se puede solicitar su ejecución desde cualquier otro bloque PL/SQL, o desde otros entornos, siempre y cuando se haya establecido la conexión a nuestra base de datos, se tengan los permisos oportunos y esté compilado correctamente.

Los bloques almacenados que se pueden crear son:

- Procedimientos (Procedures)
- Funciones (Functions)
- Paquetes (Packages)
- Disparadores (Triggers)

El prototipo de un bloque almacenado es:

```
CREATE TYPE-BLOQUE-ALMACENADO nombre [(declaración de parámetros)]  
IS  
    [Declaracion de variables]  
BEGIN  
    Ejecucion de comandos y sentencias  
[EXCEPTION  
    Control de excepciones]  
END;  
/
```

Posteriormente verá cada uno de ellos.

Estructura

Un bloque PL/SQL está compuesto por tres partes bien diferenciadas:

- La parte declarativa (opcional): En ella se declaran las variables, constantes, cursores, registros, etc. que se utilizarán en la parte de ejecución.
- La parte de ejecución (obligatoria): En ella se especifican todos los comandos que se realizarán para solucionar el problema o sub-problema. También alberga las estructuras de control. Es la única parte que es obligatoria.
- La parte de excepciones (opcional): En ella se tratan los errores que se hayan podido producir en la parte de ejecución.

El orden es lógico puesto que primero se declara con qué se va a trabajar; en la segunda se trabaja con ello y en la tercera se comprueba si ha habido algún error y se obra en consecuencia.

```
DECLARE
    --Parte declarativa
BEGIN
    --Parte de ejecución
EXCEPTION
    --Parte de excepciones
END;
/
```

El siguiente es un bloque anónimo que muestra un mensaje por consola. Se utiliza para ello el paquete predefinido DBMS_OUTPUT. Para poder visualizar la salida por consola hay que activar mediante la sentencia:

```
set serveroutput on
```

Si además lo ejecutamos como un script también debemos activar termout:

```
set serveroutput on
set termout on

BEGIN
    dbms_output.put_line('Primer bloque');
END;
/
```

serveroutput y termout son parámetros de sesión, su valor permanecerá durante la sesión; si iniciamos una nueva sesión tendrán los valores por defecto.

Declaraciones

La zona de declaraciones permite reservar espacio para las variables y las constantes que se utilizarán en el programa PL/SQL. Así mismo, en la zona de declaraciones se especifica el nombre y el tipo de las variables y constantes.

Las variables no inicializadas toman el valor NULL. Las variables pueden ser de tipo NOT NULL, esto significa que la variable tiene que contener un valor y por lo tanto es obligatoria en este caso inicializarla cuando se declara la misma.

Es aconsejable no hacer referencia a una variable que no haya sido inicializada previamente para evitar resultados inesperados.

```
Fecha_Pedido      DATE;
Lineas_factura     INTEGER := 1;
```

La definición de constantes se realiza mediante el uso de la cláusula CONSTANT. La inicialización de una constante se debe realizar en la zona de declaraciones puesto que si no se producirá un error de compilación. Este valor no se podrá cambiar mientras exista, se evita así que en el bloque o subprograma pueda ser modificada por error en el código del programa.

```
Dias_Interes      CONSTANT INTEGER := 360;
```

- Cláusula DEFAULT

Se puede utilizar la cláusula DEFAULT en vez de la asignación directa para inicializar una variable o una constante.

```
Salario_Minimo INTEGER := 700;
Salario_Minimo INTEGER DEFAULT 700;
```

- Cláusula NOT NULL

Al utilizar esta cláusula la variable no podrá contener valores nulos. En caso de que tuviera un valor nulo se produciría la excepción predefinida VALUE_ERROR. Se ha de asignar un valor a la variable en la zona de declaraciones al utilizar esta constante. Las cláusulas NATURALN y POSITIVEN llevan implícitas la cláusula NOT NULL.

```
Saldo REAL NOT NULL;           -- incorrecto puesto que no se inicializa.
Saldo REAL NOT NULL := 499;    -- Correcto.
```

Por optimización se recomienda evitar la declaración de variables not null. Si se desea comprobar si es nulo, implementarlo mediante programación.

- Cláusula %TYPE y %ROWTYPE.

Permite asignar a las variables el mismo tipo de datos que una columna o fila de la base de datos.

- %TYPE permite asignar el tipo de una Columna.

```
Salario employees.salary%TYPE; /* Asigna a la variable Salario el mismo tipo
de datos y precisión que la columna salary de la tabla employees.*/
```

- %ROWTYPE el de una fila completa. Permite también definir el registro de un cursor.

```
Departamento departments%ROWTYPE; /* Asigna a la variable departamento la
misma estructura que una fila de la tabla departments*/
```

Restricciones sobre declaraciones:

Cuando en una declaración se referencia una variable, ésta ha tenido que ser declarada anteriormente.

```
Departamento REAL:=Suma_Departamento/10; /* Incorrecto puesto que
Suma_Departamento todavía no ha sido declarado.*/
Suma_Departamento REAL := 20;
```

Algunos lenguajes permiten declarar una lista de variables con el mismo tipo de datos. PL/SQL no lo permite.

```
i, j, k    NUMBER(3);  -- incorrecto
i          NUMBER(3);  -- correcto
j          NUMBER(3);  -- correcto
k          NUMBER(3);  -- correcto
```

Los nombres de las columnas de una tabla en una sentencia SQL tienen preferencia en PL/SQL sobre los nombres de las variables o constantes. Esto podría llevar a situaciones de un comportamiento no deseado de una sentencia SQL.

```
DECLARE
  Dni          VARCHAR2(15)  := '23.990.934-B';
```

```

BEGIN
    UPDATE declaracion
    SET presentada = 'S'
    WHERE dni = Dni;
/* En este caso se actualizaría toda la tabla de declaraciones puesto que la
cláusula where sería cierta para cada uno de las filas.
Nuestra intención sólo era actualizar una sola fila pero los nombres de
columnas tienen preferencia sobre los nombres de variables.
*/
END;
/

```

Variables y constantes

Se declaran en la zona de declaraciones. En el ejemplo

```

DECLARE
    Discos_vendidos NUMBER(5);

```

No se puede referenciar a una variable que no haya sido declarada con anterioridad.

La manera de asignar un valor a una variable es utilizar la notación de (:=) (Dos puntos y el signo igual).

```

Artista := 'AbbA';

```

La segunda manera de asignar un valor a una variable es recuperando el valor de la base de datos. Esta asignación se realiza en la parte de ejecución. En el ejemplo:

```

BEGIN
    SELECT      first_name, salary
    INTO        v_nombre, v_salario --variables previamente declaradas
    FROM        Employees
    WHERE       employee_id = 100;
    ...

```

En PLSQL una sentencia SELECT lleva la cláusula INTO seguido de las variables donde se almacenará lo que se está seleccionando. Una SELECT-INTO solo nos devolverá datos, cuando seleccione una sola fila; en el resto de casos (cero o más de una fila) dará una excepción.

La tercera manera de asignar un valor a una variable es pasando la variable como un parámetro IN OUT o OUT a un procedimiento.

```

DECLARE
    Salario REAL(7,2);
    PROCEDURE Ajusta_salario
        (identificador      INT
        , salario            IN OUT REAL)
    IS
    BEGIN
        ...
    END;
BEGIN
    SELECT AVG(sal) INTO salario FROM emp;
    Ajusta_Salario (7788, Salario); /* Asigna un nuevo valor a salario*/

```

Las constantes son valores que no cambiarán en toda la ejecución del bloque o sub-bloque.

```

DECLARE

```

```
Disco_Platino CONSTANT REAL := 250000.00;
```

Cuerpo del Bloque

Estructuras de control

Las estructuras de control permiten manipular los datos ya sean del bloque PL/SQL como de la base de datos. Ofrecen control condicional, interactividad y control de la ejecución del programa. Los comandos PL/SQL son IF-THEN-ELSE, CASE, FOR-LOOP, WHILE-LOOP, EXIT-WHEN y GOTO. Colectivamente, estos comandos permiten resolver cualquier situación.

Excepciones

Cuando se produce un error se levanta (RAISE) una excepción y el flujo del programa pasa a la zona de excepciones.

Las excepciones pueden ser estándar o definidas por el usuario; además pueden ser provocadas por un error propio de Oracle (Ej: No hay filas en una select, se insertan valores duplicados en una clave única, etc.) o provocadas por el usuario si se cumple una determinada circunstancia. (Ej: El salario es menor que el legal, no hay stock para satisfacer un pedido, etc.)

```
DECLARE
    ...
    e_sal_bajo EXCEPTION;           -- declara una excepción
BEGIN
    ...
    IF v_salario < 2500 THEN
        RAISE e_sal_bajo;           -- levanta la excepción (raise)
    END IF;
    ...
EXCEPTION
    WHEN e_sal_bajo THEN
        ...                         -- procesa la excepción.
END;
/
```

Sobre Estructuras de control, excepciones; se tocarán en unidades posteriores.

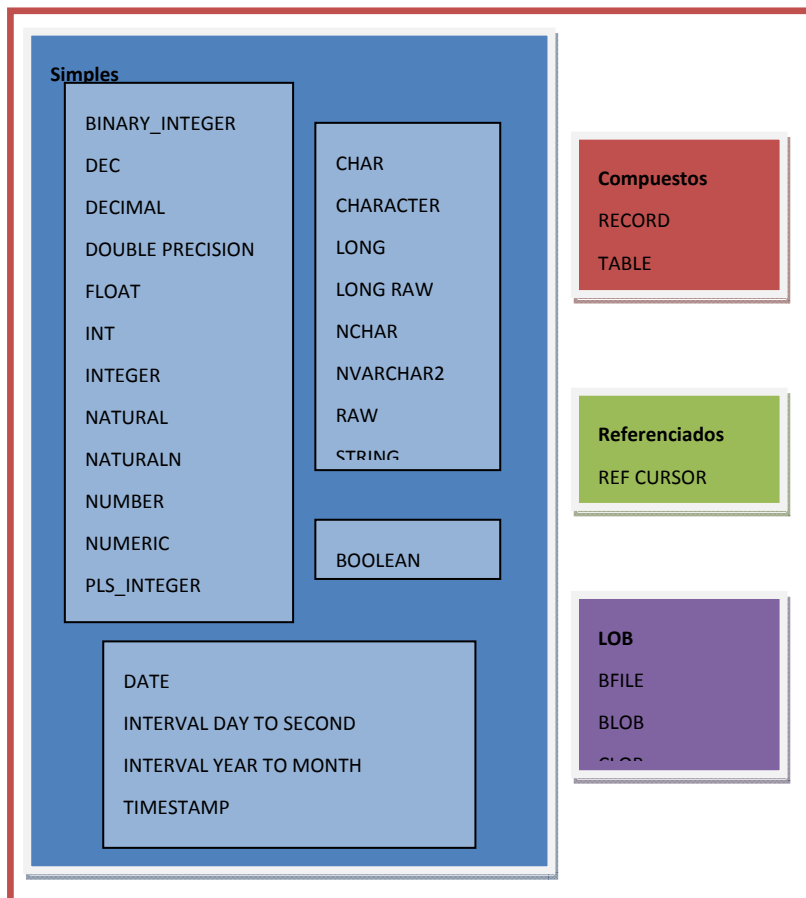
3

Tipos de Datos

Tabla de contenidos

Introducción	1
Tipos de datos predefinidos	1
Tipos numéricos.....	2
BINARY_INTEGER	2
NUMBER.....	2
PLS_INTEGER.....	3
Tipos de Datos a partir de Oracle 10g.....	3
SIMPLE_INTEGER.....	5
Tipos Carácter	5
CHAR	6
VARCHAR2.....	6
CARACTERES NACIONALES.....	6
RAW.....	7
LONG y LONG RAW	7
Tipos LOB	8
BFILE	8
BLOB	8
CLOB y NCLOB	8
Tipos Booleanos.....	9
BOOLEAN.....	9
Tipos Fecha e Interval	9
DATE	10
TIMESTAMP.....	10
TIMESTAMP WITH TIME ZONE.....	11
INTERVAL YEAR TO MONTH	11
INTERVAL DAY TO SECOND	11
Subtipos definidos por el usuario	13
Conversión de datos.....	14
Visibilidad y Ámbito.	14

Introducción



En PL/SQL cada variable, constante, función o parámetro tiene su tipo o datatype, el cual define su rango de valores, su formato y sus limitaciones.

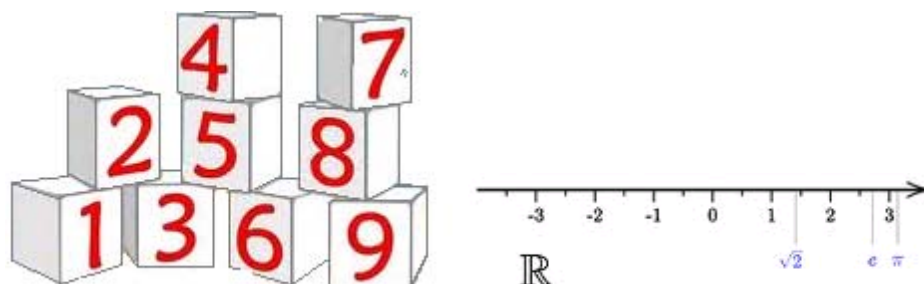
PL/SQL cuenta con una serie de formatos predefinidos y permite, también, definir tipos de datos propios del usuario.

Tipos de datos predefinidos

Un tipo de dato escalar no tiene componentes internos. Un tipo de dato compuesto tiene componentes internos que pueden ser tratados individualmente. Una referencia contiene punteros los cuales designan a otros elementos de programa. Un tipo LOB contiene unos punteros llamados localizadores LOB que especifican la localización de los objetos que están almacenados exteriormente.

Cada tipo de datos puede contener un subtipo. Los subtipos asocian limitaciones a los tipos de datos. Ejemplo: NATURALN no permite números negativos ni valores nulos.

Tipos numéricos



BINARY_INTEGER

Se utilizan para almacenar valores enteros con signo. Su rango es de -2^{31} a 2^{32} . (-2.147.483.648 a 4.294.967.296). Requieren menos espacio de almacenamiento pero las operaciones con ellos son más lentas.

Existen los siguientes subtipos:

NATURAL	Permite Nulos, 0 y mayor que 0
NATURALN	Permite 0 y mayor que 0.
POSITIVE	Permite Nulos y mayor que 0.
POSITIVEN	Sólo permite mayor que 0.
SIGNTYPE	Sólo permite -1,0 y 1.

NUMBER

Permite almacenar valores numéricos con o sin decimales. Se puede definir con precisión, cantidad de dígitos, y la escalar, cantidad de dígitos decimales. Por ejemplo: `NUMBER(6,3)` permite almacenar un número de 3 cifras y 3 decimales, 6 números en total.

El rango de los `NUMBER` es de $1E-130$.. $10E125$. El máximo valor de la precisión es de 38 dígitos.

La escala varía desde -84 hasta 127 y determina en qué posición se va a realizar el redondeo. Si no se especifica la escala se redondea al siguiente número entero.

<code>NUMBER (6,3)</code>	354,3567 pasa a 354,357
<code>NUMBER (9,-3)</code>	123456 pasa a 123000
<code>NUMBER (5)</code>	12345,687 pasa a 12346

Los subtipos de `NUMBER` son:

DEC DECIMAL NUMERIC	Declara números de coma flotante fija con una precisión de hasta 38 dígitos Decimales.
--	--

DOUBLE_PRECISION FLOAT	Declara números de coma flotante con una precisión máxima de 128 dígitos binarios equivalente a casi 38 dígitos decimales.
REAL	Declara números de coma flotante con una precisión máxima de 63 dígitos binarios equivalente a casi 18 dígitos decimales.
INTEGERS INT SMALLINT	Declara enteros con una precisión máxima de 38 dígitos decimales.

PLS_INTEGER

Se utilizan para almacenar enteros con signo. Su rango es de -2^{31} a 2^{31} .

Este tipo de datos requieren menos espacio para su almacenamiento y aprovecha la capacidad matemática de la máquina; por ello es **más aconsejable su uso** frente a otro tipo de dato siempre que sea posible.

PLS_INTEGER y BINARY_INTEGER tienen capacidades similares. Aún siendo similares en su capacidad no son compatibles. En caso de que se produjera un desbordamiento en una operación, PLS_INTEGER levantaría una excepción (error) mientras que BINARY_INTEGER asignaría el valor resultante en una variable tipo NUMBER.

Tipos de Datos a partir de Oracle 10g

Oracle Database 10g introduce dos nuevos tipos de datos numéricos de coma flotante. Los tipos de datos BINARY_FLOAT y BINARY_DOUBLE, almacenan datos numéricos de coma flotante en formato IEEE 754 de 32 bits y en formato IEEE 754 de 64 bits de doble precisión respectivamente.

En comparación con el tipo de dato Oracle NUMBER, las operaciones aritméticas con datos numéricos de coma flotante son más rápidas con estos tipos de datos. Igualmente, los valores con precisión significativa requerirán menos espacio de almacenamiento con estos tipos de datos.

Las operaciones aritméticas con los tipos de datos BINARY_FLOAT y BINARY_DOUBLE se realizan por el juego de instrucciones nativo suministrado por el proveedor de hardware. La compatibilidad con el estándar IEEE 754 asegura resultados comunes a lo largo de todas las plataformas soportadas.

Adicionalmente, los nuevos tipos de datos de coma flotante ofrecen nuevas posibilidades:

Creación de tablas con columnas de tipo BINARY_FLOAT y BINARY_DOUBLE.

Inclusión de columnas BINARY_FLOAT y BINARY_DOUBLE en cláusulas SELECT.

Creación de índices sobre columnas BINARY_FLOAT y BINARY_DOUBLE.

Las operaciones de agrupamiento están soportadas para columnas `BINARY_FLOAT` y `BINARY_DOUBLE`.

Las columnas `BINARY_FLOAT` y `BINARY_DOUBLE` pueden ser incluidas en cláusulas `order by` y `group by`.

El almacenamiento de los datos `BINARY_FLOAT` y `BINARY_DOUBLE` es independiente de la plataforma.

Hasta ahora, el tipo de datos Oracle `NUMBER` era el único tipo de dato soportado por Oracle, y todas las operaciones aritméticas se realizaban con dicho tipo de datos. Los beneficios de Oracle `NUMBER` incluyen:

Portabilidad, pues está implementado mediante software.

Representación decimal. La precisión no se pierde al convertir cadenas de texto en valores numéricos, y el redondeo se realiza en valores decimales. Muchas aplicaciones requerían este comportamiento.

Limitando el tipo de datos en la base de datos, muchos de los operadores aritméticos y funciones matemáticas no están sobrecargadas.

Sin embargo, Java y XML cobran mayor importancia en las aplicaciones de base de datos hoy en día. Estos lenguajes soportan el estándar IEEE 754 para aritmética binaria de punto flotante.

Aún más, muchas aplicaciones de base de datos requieren cálculos intensivos de coma flotante (OLAP, Data Mining, etc.), de manera que todas ellas se beneficiarán del aumento de rendimiento, reducción del espacio de almacenamiento y mayor funcionalidad de dichos tipos de datos.

`BINARY_FLOAT` y `BINARY_DOUBLE` no sustituyen al tipo de dato Oracle `Number`, sino que constituyen una alternativa que proporciona los siguientes beneficios:

`BINARY_FLOAT` y `BINARY_DOUBLE` coinciden con los tipos de datos utilizados por los clientes Java y XML de una base de datos. Ambos soportan tipos de datos equivalentes a los tipos de datos IEEE 754.

Hasta ahora, los datos numéricos eran almacenados como Oracle `Number`, y la conversión desde y hacia `BINARY_FLOAT` y `BINARY_DOUBLE` podía perder precisión o provocar un error. Esto es debido a que Oracle `Number` utiliza una representación en base 10, mientras que `BINARY_FLOAT` y `BINARY_DOUBLE` utilizan base 2.

Los cálculos aritméticos son entre 5 y 10 veces más rápidos para `BINARY_FLOAT` y `BINARY_DOUBLE` que para Oracle `Number`.

Los datos `BINARY_FLOAT` utilizan 5 bytes, mientras que `BINARY_DOUBLE` utiliza 9 bytes, incluyendo el byte de longitud de campo. Oracle `Number` utiliza de 1 a 22 bytes de longitud.

IEEE 754 proporciona toda la funcionalidad necesaria para escribir algoritmos numéricos. Muchas de esas características no son proporcionadas por Oracle `Number`.

Oracle `NUMBER` está implementado por software, lo que hace que las operaciones con dicho tipo de datos sean menos eficientes que las operaciones con valores nativos de coma flotante, implementados por hardware y soportados por la mayoría de los conjuntos de instrucciones actuales.

Las funciones aritméticas y matemáticas que utilizan los tipos de datos BINARY_FLOAT y BINARY_DOUBLE son más rápidas y utilizan menos espacio en disco.

Tanto SQL como PL/SQL ofrecen soporte completo para ambos tipos de datos. BINARY_FLOAT y BINARY_DOUBLE pueden ser utilizados en cualquier contexto donde sea posible utilizar un tipo de dato escalar.

SIMPLE_INTEGER

Oracle Database 11g ha introducido éste nuevo tipo de dato numérico entero. Éste tipo de dato es un subtipo del tipo de dato **pls_integer** de versiones anteriores. Tiene el mismo rango numérico de valores (-2147483648 .. +2147483647), pero carece de comprobación de desbordamiento (incrementando en 1 el mayor número devuelve al número más pequeño, y la inversa).

Además, este nuevo tipo de dato no permite valores nulos (NULL). Se debe de inicializar en la declaración.

PLS-00218: una variable declarada NOT NULL debe tener una asignación de inicialización

Estas dos diferencias sirven para que el funcionamiento interno de *simple_integer* sea más rápido que el de *pls_integer* utilizando sobre todo compilación nativa de PL/SQL.

Por ejemplo:

```
DECLARE
  n1 simple_integer := +2147483647;
  n2 simple_integer := -2147483648;
BEGIN
  dbms_output.put_line('n1 := ' || n1);
  dbms_output.put_line('n2 := ' || n2);
  n1:=n1+1;
  n2:=n2-1;
  dbms_output.put_line('n1+1 := ' || n1);
  dbms_output.put_line('n2-1 := ' || n2);
END;
/
```

El resultado es el siguiente:

```
n1 := 2147483647
n2 := -2147483648
n1+1 := -2147483648
n2-1 := 2147483647
```

Tipos Carácter



CHAR

Permite almacenar una cadena de longitud fija de caracteres. Como se almacena el dato internamente depende del conjunto de caracteres de la base de datos. CHAR acepta indicarle la longitud máxima de caracteres o bytes. El máximo tamaño es de 32767 bytes. Se puede especificar bytes o caracteres ya que en bases de datos con el juego de caracteres multi_byte un carácter puede ocupar más de un byte. Si no se especifica longitud se usa el valor por defecto 1.

En caso de recibir un valor mayor de 32767 bytes quedará truncado a esa longitud. Ejemplo: Un campo LONG de una columna puede llegar a 2 Gigabytes.

Si recuperamos un campo LONG en un CHAR lo máximo que podrá contener son 32767 bytes.

Se puede utilizar CHARACTER en lugar de CHAR para una mayor legibilidad.

VARI1	CHAR;	-- Un solo carácter de longitud
VARI2	CHAR(30);	-- 30 caracteres de longitud
VARI3	CHARACTER(30);	-- 30 caracteres de longitud

VARCHAR2

Se utiliza para almacenar cadenas de caracteres variables. Se ha de especificar la longitud máxima que puede tener. El límite de almacenamiento es de 32767 bytes.

PL/SQL trata los VARCHAR2 desde dos puntos de vista:

Si el campo tiene una longitud menor de 2000 bytes, PL/SQL le asignará tanta memoria como esté definido.

Si es mayor o igual a 2000 bytes, PL/SQL le asignará dinámicamente tanta memoria como sea necesaria para contener el valor actual.

Si se define un varchar2(50) y se asigna solo cuatro caracteres, la variable reservará memoria para cincuenta caracteres.

Mientras que si se define un varchar2(2000), y se asigna solo cuatro caracteres, la variable reservará memoria para cuatro caracteres.

Para cadenas de caracteres variables también se puede utilizar los tipos de datos VARCHAR o STRING.

Se pueden insertar en columnas de la base de datos tipo LONG los tipos VARCHAR2 ya que las columnas LONG permiten una longitud de 2**31. Sin embargo no se podrá recuperar una columna LONG en un campo VARCHAR2 si ésta es mayor de 32767 bytes.

CARACTERES NACIONALES

La amplitudes usadas en el conjunto de caracteres ASCII y EBCDIC son las adecuadas para el alfabeto Romano, pero también los lenguajes Asiáticos, tales como el Japonés, que contienen cientos de caracteres. Estos lenguajes requieren dos o tres bytes para representar cada carácter.

PL/SQL soporta dos conjuntos de caracteres, el llamado conjunto de caracteres de la base de datos, los cuales son usados para identificadores y código fuente, y el conjunto de caracteres nacionales, el cual es usado para los datos del lenguaje nacional. Los tipos de datos NCHAR y

NVARCHAR2 almacenan las cadenas de caracteres formadas por el conjunto de caracteres nacionales.

El máximo tamaño que pueden almacenar estos tipos de datos es de 32767 bytes.

RAW

Se utiliza para almacenar valores binarios o cadenas de caracteres. Su funcionamiento es similar a los VARCHAR2. PL/SQL no interpreta los valores almacenados en el tipo RAW. Se ha de especificar la longitud máxima que puede tener. El límite de almacenamiento es de 32767 bytes. El ancho máximo de una columna en la base de datos es de 2000 bytes. Se pueden insertar columnas RAW en LONG RAW, no está permitida la inversa.

LONG y LONG RAW

Se utiliza para almacenar cadenas de caracteres variables. Se ha de especificar la longitud máxima que puede tener. El límite de almacenamiento es de 32760 bytes.

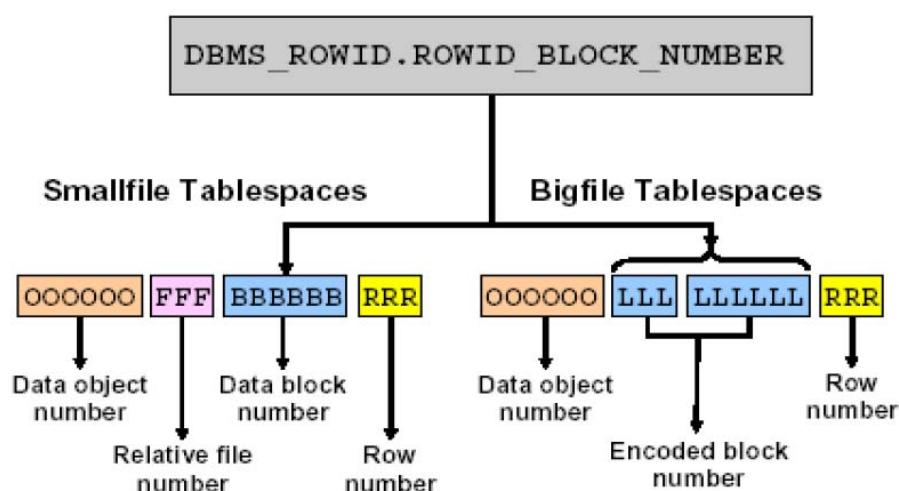
LONG RAW se utiliza para guardar valores binarios o cadenas de caracteres. El tipo LONG RAW es similar al RAW, pero no es interpretado por PL/SQL.

A partir de 9i las variables LONG y LONG RAW se pueden intercambiar con los campos LOB. Oracle recomienda migrar cualquier dato LONG a CLOB, y cualquier dato LONG RAW al tipo BLOB.

Las columnas LONG pueden almacenar texto, arreglos de caracteres o documentos cortos. Se pueden referenciar columnas LONG en sentencias UPDATE, INSERT y SELECT, pero no en expresiones, llamados a funciones o en cláusulas tales como WHERE, GROUP BY y CONNECT BY.

ROWID y UROWID.

El rowid representa unívocamente la dirección de almacenamiento de una fila dentro de la base de datos. Internamente, cualquier tabla de la base de datos tiene una pseudocolumna, la cual almacena en un valor binario el ROWID.



Existen 2 tipos de ROWID:

Rowid Físico: identifica a una fila en una tabla.

Rowid Lógico: identifica a una fila en una tabla indexada.

El tipo de datos ROWID sólo puede almacenar rowids físicos mientras que UROWID acepta rowids físicos, lógicos y rowids que no son de Oracle.

El uso de ROWID sólo se aconseja para mantener la compatibilidad. ORACLE recomienda el uso de UROWID.

Al recuperar un rowid en una variable tipo ROWID se debe utilizar la función ROWIDTOCHAR para convertirlo en una cadena de 18 caracteres. Para convertir una cadena en rowid se utiliza la función CHARTOROWID. En caso que la cadena no sea válida se levantará la excepción SYS_INVALID_ROWID.

Para manipular los tipos de datos ROWID, también se puede utilizar el paquete DBMS_ROWID.

Con los tipos de variable UROWID no es necesario utilizar las funciones ROWIDTOCHAR o CHARTOROWID siendo implícita la conversión entre tipos UROWID y CHAR.

Tipos LOB

Los tipos LOB (Large Object) permiten almacenar bloques de datos sin una estructura fija tales como gráficos, sonidos, videos o cualquier otro tipo de información. Su tamaño máximo es de 4 Gigabytes. Soportan el acceso aleatorio a los datos que se almacenan en ellos.

En los datos tipo LOB se almacenan los punteros de los objetos grandes que pueden estar almacenados dentro de la fila o fuera de la fila. CLOB, BLOB y NCLOB almacenan los objetos dentro de la base de datos mientras que BFILE apunta al objeto que está fuera de la base de datos y es dependiente del sistema operativo. PL/SQL opera con los localizadores.

Oracle convierte datos CLOBs a CHAR y VARCHAR2 y viceversa, o BLOBs a RAW y viceversa. Para manipular con tipos de datos LOB se utiliza el paquete DBMS_LOB.

BFILE

Se utiliza para almacenar objetos binarios fuera de la Base de datos. Cada BFILE utiliza un localizador que incluye el directorio y la dirección completa del fichero en el sistema operativo. No puede ser mayor que 4 Gigabytes y es de solo lectura. BFILE no participa en las transacciones, no genera REDO LOGS y no puede ser replicado.

BLOB

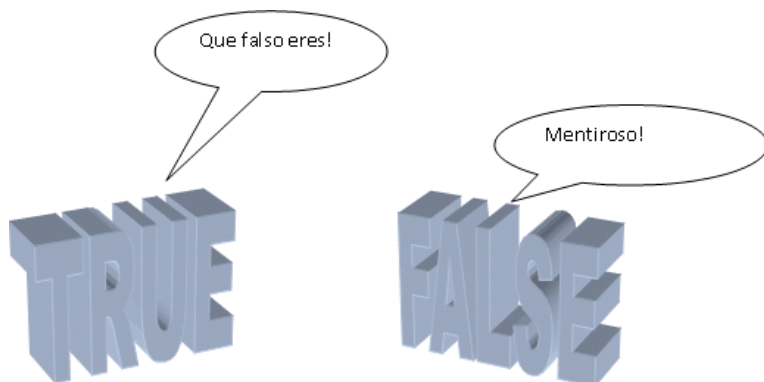
Se utiliza para almacenar objetos binarios dentro de la Base de datos. Cada BLOB utiliza un localizador que señala la posición del objeto. No puede ser mayor que 4 Gigabytes. BLOB participa plenamente en las transacciones. Los cambios hechos por el paquete DBMS_LOB pueden ser validados por commit o desechados por rollback.

CLOB y NCLOB

Se utiliza para almacenar objetos de tipo carácter grandes dentro de la Base de datos. Se soportan los tipos de carácter fijo y variable. NCLOB tiene las mismas características que CLOB salvo que los caracteres son los del conjuntos de caracteres nacionales. Cada CLOB

utiliza un localizador que señala la posición del objeto grande de caracteres. No puede ser mayor que 4 Gigabytes. CLOB participa plenamente en las transacciones. Los cambios hechos por el paquete DBMS_LOB pueden ser validados por commit o desechados por rollback.

Tipos Booleanos



BOOLEAN

Almacenan los valores lógicos TRUE, FALSE o NULL (Verdadero, falso o nulo). Solo las operaciones lógicas están permitidas en las variables BOOLEANAS.

No se pueden insertar valores TRUE o FALSE en columnas de la base de datos; así como no se pueden seleccionar columnas de la base de datos en variables BOOLEANAS.

Tipos Fecha e Interval



Permiten utilizar guardar y manipular fechas, tiempo e intervalos de tiempo.

Constan de los siguientes campos.

Nombre del campo	Valor válido para tipo DATETIME	Valor válido para INTERVAL
YEAR	-4712 al 9999 (excluyendo el año 0)	Cualquier valor distinto de 0
MONTH	01 al 12	0 al 11
DAY	01 al 31 (dependiendo del mes, el año y el calendario local)	Cualquier valor distinto de 0.
HOURL	00 a 23	00 a 23
MINUTE	00 a 59	00 a 59

SECOND	00 a 59.9(n) donde 9(n) es la precisión de la parte fraccional del segundo.	00 a 59.9(n) donde 9(n) es la precisión de la parte fraccional del segundo.
TIMEZONE_HOUR	-12 a 14 (dependiente de la zona horaria y del horario)	No aplicable
TIMEZONE_MINUTE	00 a 59	No aplicable
TIMEZONE_REGION	Se encuentra en la vista V\$TIMEZONES_NAMES	No aplicable
TIMEZONE_ABBR	Se encuentra en la vista V\$TIMEZONES_NAMES	No aplicable

Exceptuando **TIMESTAMP WITH LOCAL TIMEZONE**, todos los tipos forman parte del estándar de SQL92.

DATE

Almacena fechas fijas. La función *sysdate* devuelve un tipo **DATE**. La función *SYSDATE* retorna el día y la hora actual. PL/SQL devuelve las fechas en el formato indicado en *NLS_DATE_FORMAT*.

Se pueden realizar operaciones con las fechas añadiéndole o sustrayéndole días. PL/SQL interpreta los valores enteros como días.

```

DECLARE
    Fecha          DATE :=to_date('22/ENE/2009','dd/mm/yyyy');
    Dia_Después    DATE;
BEGIN
    Dia_Después:= Fecha + 1;           -- Dia_Después valdrá 23/01/09

```

```

DECLARE
    Dias_Trabajados    DATE;
BEGIN
    SELECT  SYSDATE - Hire_Date
    INTO   Dias_Trabajados
    FROM   Employees
    WHERE  Employee_Id = 120;
END;
/

```

TIMESTAMP

Este tipo de datos, es una extensión del **DATE**, almacena la fecha con año, mes, día, hora, minutos y segundos; y permite utilizar el parámetro de precisión para los segundos. Se puede indicar una precisión con el rango 0..9 dígitos. El valor por defecto es 6.

El formato por defecto está indicado en el parámetro de inicialización *NLS_TIMESTAMP_FORMAT*.

```

DECLARE
    Tiempo TIMESTAMP(3);
BEGIN
    Tiempo := '24-01-1967 06:48:53.275'; /* La precisión en este caso es
                                         de 3 (Milésimas de segundo)*/
    ...
END;
/

```

TIMESTAMP WITH TIME ZONE

Sus características son similares a **TIMESTAMP**. Incluye el desplazamiento de la zona horaria en horas y minutos. Es la diferencia entre el horario local y el UTC (Coordinated Universal Time) de Greenwich (Londres, Inglaterra)

El formato por defecto está indicado en el parámetro de inicialización **NLS_TIMESTAMP_TZ_FORMAT**.

```

DECLARE
    Tiempo TIMESTAMP(3)WITH TIME ZONE;
    Tiempo_Aqui TIMESTAMP(3)WITH LOCAL TIME ZONE;
BEGIN
    Tiempo := '24-01-1967 06:48:53.275 +1:00'; /* Horario español peninsular*/
    Tiempo_Aqui := '24-01-1967 06:48:53.275'; /* Exactamente igual que el
    anterior*/
    ...
END;
/

```

Es posible utilizar abreviaturas o nombres de zonas para especificar la zona horaria. Los nombres se pueden encontrar en las columnas **TIMEZONE_REGION** y **TIMEZONE_ABBR** de la vista **V\$TIMEZONE_NAMES**.

Si se declara la variable **TIMESTAMP WITH LOCAL TIME ZONE**, Oracle devuelve el valor con el desplazamiento de nuestra zona horaria.

INTERVAL YEAR TO MONTH

Se utiliza para manipular y almacenar intervalos de años y meses. Su sintaxis es:

```
INTERVAL YEAR (precisión) TO MONTH;
```

Precisión especifica el número de dígitos del valor año. Su valor varía de 0 a 4 y el valor por defecto es 2.

INTERVAL DAY TO SECOND

Se utiliza para almacenar y manipular intervalos de días, horas, minutos y segundos. Su sintaxis es:

```
INTERVAL DAY (Precisión_días) TO SECOND (Precisión_segundos);
```

Precisión_días y **Precisión_segundos** indican el número de dígitos para el campo día y el número de decimales para el campo segundo. El rango válido para ambos es de 0 a 9 y los valores por defecto son 2 y 6 respectivamente.

Operando 1	Operación	Operando 2	Tipo Resultante
------------	-----------	------------	-----------------

datetime	+	interval	Datetime
datetime	-	interval	Datetime
interval	+	datetime	Datetime
datetime	-	datetime	Interval
interval	+	interval	Interval
interval	-	interval	Interval
interval	*	numeric	Interval
numeric	*	interval	Interval
interval	/	numeric	Interval

En la tabla anterior se muestra el tipo resultante de interactuar los diferentes tipos de datos DATE y el tipo de dato resultante de la operación.

El siguiente cuadro muestra las posibles asignaciones a tipos de datos interval:

INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 días, 5 horas, 12 minutos, 10 segundos, and 222 milésimas de segundos.
INTERVAL '4 5:12' DAY TO MINUTE	4 días, 5 horas y 12 minutos.
INTERVAL '400 5' DAY(3) TO HOUR	400 días, 5 horas.
INTERVAL '400' DAY(3)	400 días.
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)	11 horas, 12 minutos, y 10.2222222 segundos.
INTERVAL '11:20' HOUR TO MINUTE	11 horas y 20 minutos.
INTERVAL '10' HOUR	10 horas.
INTERVAL '10:22' MINUTE TO SECOND	10 minutos 22 segundos
INTERVAL '10' MINUTE	10 minutos.
INTERVAL '4' DAY	4 días

INTERVAL '25' HOUR	25 horas.
INTERVAL '40' MINUTE	40 minutos.
INTERVAL '120' HOUR(3)	120 horas
INTERVAL '30.12345' SECOND(2,4)	30.1235 segundos. La fracción de segundos '12345' esta redondeada a '1235' porque la precisión es 4.

Subtipos definidos por el usuario

Los subtipos definidos por el usuario permiten definir una serie de límites a los tipos ya creados y que son propios de PL/SQL. Los subtipos se utilizan para compatibilidad con ANSI/ISO y mejorar la legibilidad de las variables indicando su uso y sus limitaciones.

Así mismo los subtipos definidos por el usuario pueden utilizar los subtipos predefinidos por PL/SQL.

Su sintaxis es la siguiente:

```
SUBTYPE nombre_subtipo IS tipo_PL/SQL [(precisión)] [NOT NULL];
```

nombre_subtipo es el nombre del subtipo definido.

tipo_PL/SQL es cualquiera de los tipos de datos soportados por PL/SQL.

precisión se aplica únicamente a aquellos tipos de datos en la que se pueda especificar una precisión.

Su puede utilizar también %TYPE y %ROWTYPE para especificar el tipo PL/SQL. El subtipo definido de esta manera tendrá todos los atributos de esa columna o fila de la base de datos pero no heredará otras características como por ejemplo NOT NULL.

```
DECLARE
  SUBTYPE fecha_ingreso IS DATE NOT NULL;
  SUBTYPE edad IS      NATURAL; -- NATURAL es un subtipo de NUMBER
  TYPE Votaciones      IS TABLE OF NUMBER(2);
  SUBTYPE Pais_Votando IS Votaciones; /* Se basa en la tabla de números
Votaciones*/
  SUBTYPE Presentador  IS Certamen.Presen%TYPE;
```

Los subtipos incrementan la fiabilidad de los datos al añadir limitaciones.

```
DECLARE
  SUBTYPE Voto IS NATURAL(2,0) /* Se define el subtipo voto como un numérico
de precisión 2 y valores positivos.*/
  Puntuacion  Voto;
BEGIN
  Puntuacion := 20;           -- Es un valor correcto
  Puntuacion := -23;         -- Generará un error VALUE_ERROR.
```

Conversión de datos.

Conversión explícita es aquella en la que se realiza mediante una función de conversión de datos. Ejemplo TO_CHAR, TO_DATE, TO_NUMBER.

Conversión implícita es aquella que realiza PL/SQL automáticamente.

	BIN_INT	BLOB	CHAR	CLOB	DATE	LONG	NUMBER	PLS_INT	RAW	UROWID	VARCHAR2
BIN_INT			X			X	X	X			X
BLOB									X		
CHAR	X			X	X	X	X	X	X	X	X
CLOB			X								X
DATE			X			X					X
LONG			X						X		X
NUMBER	X		X			X		X			X
PLS_INT	X		X			X	X				X
RAW		X	X			X					X
UROWID			X								X
VARCHAR2	X		X	X	X	X	X	X	X		

A ser posible por mejorar rendimiento, se recomienda evitar conversiones de dato. Es decir, utilizar el tipo de dato correspondiente al valor de asignación de la variable.

Visibilidad y Ámbito.

El ámbito de un identificador es la región de un bloque, subprograma, programa o paquete donde podemos actuar con este identificador. Los identificadores se definen como locales en el bloque y globales a todos sus sub-bloques. Si un identificador global es redeclarado en un sub-bloque, ambos identificadores permanecen en el ámbito.

La visibilidad de un identificador es aquella región de un bloque donde puede ser referenciado. Dentro de un sub-bloque, solo es visible el identificador local.

Si se define un identificador con el mismo nombre dentro de un sub-bloque que un identificador global; para hacer referencia al identificador global se deberá cualificar mientras que si no se cualifica será el identificador definido en el sub-bloque.

En la siguiente figura se ve un ejemplo de Visibilidad y Ámbito.

```
DECLARE
  vari REAL;
BEGIN
  ...
  DECLARE
    vari REAL;
  BEGIN
    ...
  END;
  ...
END;
/
```

Ámbito variable VARI exterior

```
DECLARE
  vari REAL;
BEGIN
  ...
  DECLARE
    vari REAL;
  BEGIN
    ...
  END;
  ...
END;
/
```

Visibilidad VARI exterior

```
DECLARE
  vari REAL;
BEGIN
  ...
  DECLARE
    vari REAL;
  BEGIN
    ...
  END;
  ...
END;
/
```

Ámbito 'vari' interior.

```

DECLARE
    vari REAL;
BEGIN
    ...
    DECLARE
        vari REAL;
    BEGIN
        ...
    END;
    ...
END;
/

```

Visibilidad de la variable 'vari' interior

Para poder utilizar la variable exterior dentro del sub-bloque, tenemos que cualificarla en el sub-bloque como en el subprograma, a continuación se muestra un ejemplo de ello.

```

<<bloque>>
DECLARE
    x PLS_INTEGER:=5;
BEGIN
    DECLARE
        x PLS_INTEGER:=3;
        y PLS_INTEGER;
    BEGIN
        y:= x + bloque.x;
        dbms_output.put_line('El resultado es: '||y);
    END;
    dbms_output.put_line('El valor de la variable X es: '||x);
END;
/

```

La salida será:

```

El resultado es: 8
El valor de la variable es: 5

```

```

<<bloque>>
DECLARE
    var PLS_INTEGER:=1;
BEGIN
    <<pro1>>
    DECLARE
        var PLS_INTEGER:=22;
    BEGIN
        <<pro2>>
        DECLARE
            var PLS_INTEGER:=333;
        BEGIN
            dbms_output.put_line(bloque.var);
            dbms_output.put_line(pro1.var);
            dbms_output.put_line(pro2.var);
        END pro2;
        dbms_output.put_line('En PRO1 var = '||var);
    END pro1;
    dbms_output.put_line('En BLOQUE var = '||var);
END;

```

/

La salida será:

1
33
1010

4

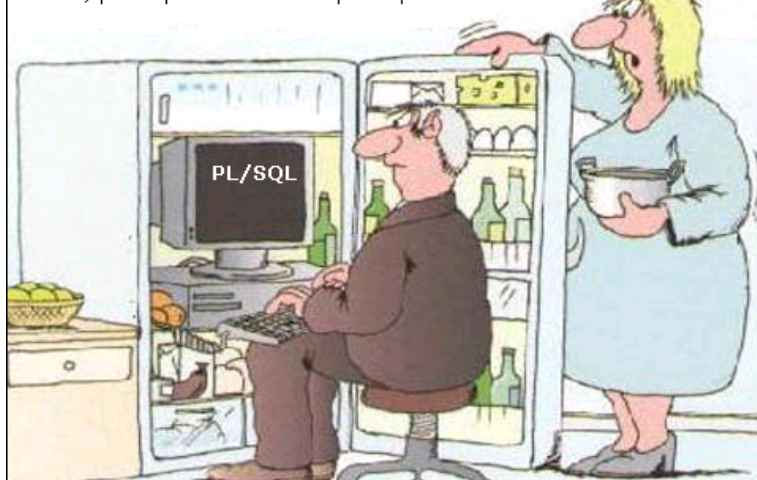
Estructuras de Control

Tabla de contenidos

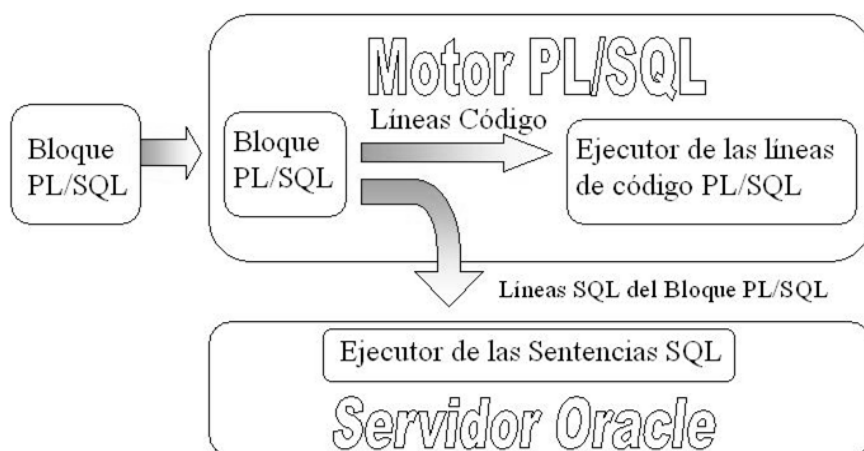
Introducción	1
Control condicional.	1
IF-THEN-ELSE.....	1
CASE	3
Cláusula WITH en PL/SQL	4
Control de iteraciones. (Bucles).....	6
LOOP	6
WHILE-LOOP	7
FOR-LOOP	8
NULL.....	9

Introducción

Estoy convencida de que necesitas refrigeración extra, pero ¿podrías hacer espacio para la comida?



Cuando se compila un bloque PL/SQL, las líneas de código procedural las ejecuta el motor de PL/SQL ya sea dentro de la herramienta en la parte del cliente o del servidor. Las sentencias SQL las ejecuta el servidor Oracle y enviará la información obtenida al motor PL/SQL.



La figura muestra como Oracle trata un bloque de PL/SQL.

Las líneas de código procedural las ejecuta el motor de PL/SQL ya sea dentro de la herramienta en la parte del cliente o del servidor. Las sentencias SQL las ejecuta el servidor Oracle y enviará la información obtenida al motor PL/SQL.

Control condicional.

Permite tomar decisiones dependiendo de la información obtenida.

IF-THEN-ELSE

El comando IF-THEN-ELSE ejecutará una serie de comandos dependiendo de la información.

IF evalúa la condición

THEN realiza la serie de comandos en caso que la condición sea cierta.

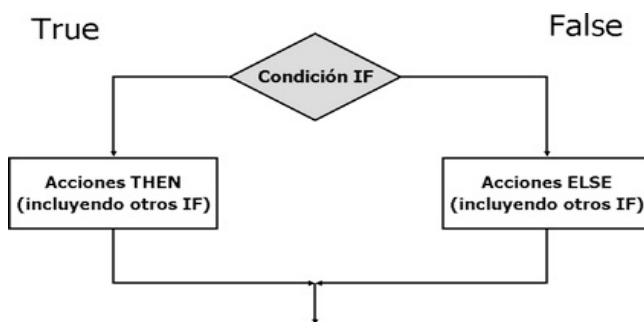
ELSIF permite evaluar las varias condiciones y en caso que se cumpla una ya no continuará evaluando el resto.

ELSE realiza la serie de comandos en caso que la condición sea falsa o nula.

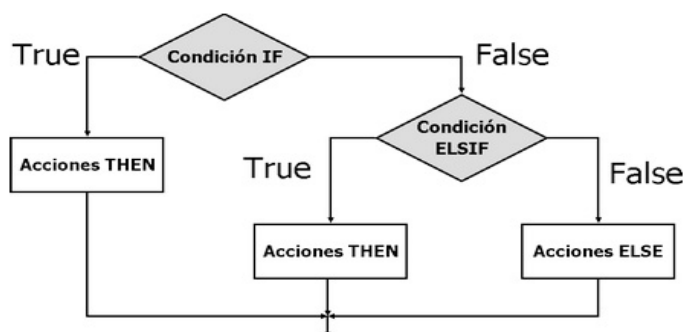
```

IF condición THEN
    instrucciones;
[ELSE
    instrucciones;]
END IF;
-----
IF condición THEN
    instrucciones;
[ELSE
    IF condicion2 THEN
        instrucciones;
    [ELSE
        instrucciones;]
    END IF;
    instrucciones;
]
END IF;
-----
IF condición THEN
    instrucciones;
ELSIF condicion2 THEN
    instrucciones;
ELSIF condicion3 THEN
    instrucciones;
[ELSE
    instrucciones;]
END IF;

```



Flujo de
IF-THEN-ELSE



Flujo de
IF-THEN-ELSIF

```
DECLARE
    v_nombre      employees.first_name%TYPE;
    v_Bono PLS_INTEGER;
    v_salario      employees.salary%TYPE;

BEGIN
    SELECT salary,first_name
    INTO    v_Salario,v_nombre
    FROM    employees
    WHERE   employee_id=154;

    IF v_Salario < 5000 THEN
        v_Bono:= 500;
    ELSIF v_Salario > 12000 THEN
        v_Bono:= 100;
    ELSE
        v_Bono:= 200;
    END IF;

    DBMS_OUTPUT.PUT_LINE('El empleado: '||v_nombre);
    DBMS_OUTPUT.PUT_LINE('El salario es: '||v_salario);
    DBMS_OUTPUT.PUT_LINE('El bono es: '||v_bono);

END;
/
```

Una sentencia Select en un bloque PLSQL se le especifica la clausula INTO. En ella se coloca en orden las variables donde se almacenará el valor de la select. Una select-into solo puede devolver una sola fila; en caso distinto saltará una excepción. Si no devuelve filas no_data_found y si devuelve más de una fila to_many_rows.

CASE

Para poder escoger entre varios valores y realizar diferentes acciones se puede utilizar la sentencia CASE.

La sentencia CASE permite también crear bloques de sentencias como si se tratara de sentencias IF anidadas.

```
CASE selector
    WHEN Expresión1 THEN secuencia_de_comandos1
    WHEN Expresión2 THEN secuencia_de_comandos2
    ...
    WHEN ExpresiónN THEN secuencia_de_comandosN
    [ELSE secuencia_de_comandosN+1]
END CASE;
```

La cláusula ELSE realiza los comandos cuando ninguna de las cláusulas WHEN se ha cumplido. No es obligatoria pero si no se utiliza y no se ha cumplido ninguna cláusula WHEN se levantará la excepción CASE_NOT_FOUND.

```
DECLARE
    V_department_id      employees.department_id%TYPE;
    V_Nombre              VARCHAR2(2000);

BEGIN
    SELECT      department_id
    INTO    v_department_id
    FROM    employees
    WHERE   employee_id=160;
```

```

v_nombre :=
    CASE V_department_id
        WHEN 20 THEN 'Marketing'
        WHEN 40 THEN 'Recursos Humanos'
        WHEN 80 THEN 'Ventas'
        ELSE 'Otro Departamento'
    END;
Dbms_output.put_line('El departamento es: '||v_nombre);
END;
/

```

También está la posibilidad de utilizar los bloques CASE con condiciones. En el caso de que se cumpla una condición el control pasará a la siguiente sentencia después del END CASE.

```

CASE
    WHEN Condición1 THEN
        secuencia_de_comandos1;
    WHEN Condición2 THEN
        secuencia_de_comandos2;
    ...
    WHEN CondiciónN THEN
        secuencia_de_comandosN;
    [ELSE secuencia_de_comandosN+1;]
END CASE;

```

```

DECLARE
    v_Salary          employees.salary%TYPE;
    v_Bono             PLS_INTEGER;
BEGIN
    SELECT            salary
    INTO              v_Salary
    FROM              employees
    WHERE             employee_id=154;

    CASE
        WHEN v_Salary< 5000 THEN
            v_Bono:= 500;
        WHEN v_Salary> 12000 THEN
            v_Bono:= 100;
        ELSE
            v_Bono:= 200;
    END CASE;
    Dbms_output.put_line('Bono: '||v_bono);
END;
/

```

Si no se cumpliera una de las condiciones y no existiera la cláusula ELSE, provocaría un error.

Por optimización y legibilidad, es recomendable implementar una sentencia CASE en vez de anidar sentencias IF o utilizar funciones tales como DECODE.

Cláusula WITH en PL/SQL

En Oracle 12c R1 se puede definir una función o procedimiento PL/SQL con la cláusula de subconsultas **WITH** y usarla en sentencias SQL ordinarias.

La sintaxis es la siguiente:

```

WITH
    [PROCEDURE nombre_procedimiento

```

```
BEGIN
...
END;]

FUNCTION nombre_función
BEGIN
...
END;
SELECT nombre_función FROM nombre_tabla;
/
```

Observaciones:

- Función WITH tiene prioridad sobre las funciones declaradas a nivel de esquema.
- Se pueden definir procedimientos que serán utilizados por la función WITH.

Ejemplos:

1. En este primer caso sencillo se declara solamente una nueva función *func_calculo*.

```
WITH
  FUNCTION func_calculo(p_entrada NUMBER) RETURN NUMBER
  IS
  BEGIN
    RETURN p_entrada*2;
  END func_calculo;
SELECT employee_id,func_calculo(employee_id),job_id FROM employees
WHERE job_id='IT_PROG';
/
```

2. Utiliza WITH con una función y un procedimiento.

```
WITH
  PROCEDURE proc_calculo(p_entrada NUMBER, p_salida OUT NUMBER)
  IS
  BEGIN
    p_salida := p_entrada+1;
  END proc_calculo;
  --
  FUNCTION func_calculo(p_entrada NUMBER) RETURN NUMBER
  IS
    v_dato NUMBER;
  BEGIN
    proc_calculo(p_entrada, v_dato);
    return v_dato*2;
  END func_calculo;
SELECT employee_id,func_calculo(employee_id),job_id FROM employees
WHERE job_id='IT_PROG';
/
```

WITH_PLSQL

Si la función WITH no es la primera declaración en una consulta, sino que se encuentra en una subconsulta entonces la consulta principal puede fallar.

```
SQL> SELECT * FROM
2  ( WITH
3      FUNCTION func_calculo(p_entrada NUMBER) RETURN NUMBER
4      IS
5      BEGIN
6          RETURN p_entrada*2;
```

```

7      END;
8      SELECT employee_id,func_calculo(employee_id),job_id
9      FROM employees
10     WHERE job_id='IT_PROG'
11 );
12 /
( WITH
*
ERROR at line 2:
ORA-32034: uso de la cláusula WITH no soportado

```

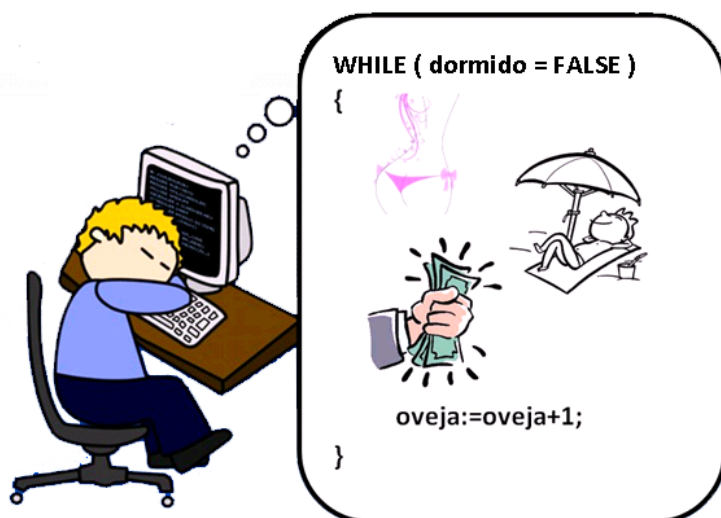
Para evitar este problema debes utilizar el *hint* **WITH_PLSQL**.

```

SQL> SELECT /*+ WITH_PLSQL */ * FROM
2  ( WITH
3      FUNCTION func_calculo(p_entrada NUMBER) RETURN NUMBER
4      IS
5      BEGIN
6          RETURN p_entrada*2;
7      END;
8      SELECT employee_id,func_calculo(employee_id),job_id
9      FROM employees
10     WHERE job_id='IT_PROG'
11 );
12 /

```

Control de iteraciones. (Bucles)



LOOP

La instrucción LOOP permite ejecutar una serie de sentencias repetidamente. Se coloca la instrucción LOOP en la primera sentencia a repetir y la instrucción END-LOOP después de la última sentencia que forma parte del bucle.

La sentencia EXIT WHEN permite salir del bucle. La condición después del WHEN es evaluada y si se cumple (TRUE) el control pasa a la sentencia inmediatamente posterior a la sentencia END LOOP.

EXIT sin ningún parámetro saldrá inmediatamente del bucle.

Si dentro de la instrucción LOOP no existen las sentencias EXIT WHEN o EXIT, se creara un bucle sin fin.

```
BEGIN
  LOOP
    [Código PL/SQL]
    EXIT WHEN condicion_corte_bucle;
    [Código PL/SQL]
  END LOOP;
END;
/
```

```
DECLARE
  Valor          PLS_INTEGER := 10;
  Suma_valores PLS_INTEGER:= 0;
BEGIN
  LOOP
    Suma_valores := Suma_valores + Valor;      -- Se suman los valores
    Valor := Valor - 1 ;
    EXIT WHEN Valor < 1;      -- Salida del bucle si el valor es 0
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('La suma de los valores es: '||Suma_valores);
END;
/
```

Permite también, al igual que los bloques PL/SQL, utilizar una etiqueta al nombre del bucle - cualquier bucle-. En caso de anidar bucles, se aconseja la utilización de etiquetas para aumentar la legibilidad del programa.

```
BEGIN
  <<bucle1>>
  LOOP
    dbms_output.put_line('b1...');
    <<bucle2>>
    LOOP
      dbms_output.put_line('b2...');
      EXIT bucle1 WHEN true;
    END LOOP;
    dbms_output.put_line('...b2');--no se ejecuta
  END LOOP;
  dbms_output.put_line('...b1');
END;
/
```

En éste ejemplo; gracias a la etiqueta podemos salir de ambos bucles, desde el bucle mas interno. Ya que sin etiqueta saldría del bucle más cercano.

WHILE-LOOP

La sentencia WHILE-LOOP asocia una condición a la repetición de las sentencias del bucle. La condición es evaluada antes de iniciar el bucle. Si se cumple la condición las sentencias son ejecutadas y la sentencia END LOOP devolverá el control al WHILE. Si no se cumple la condición el control del programa pasará a la línea siguiente al END LOOP.

```
WHILE condicion_permanencia_bucle LOOP
  Código PL/SQL
END LOOP;
```



```

DECLARE
    var      PLS_INTEGER:=0;
    suma     PLS_INTEGER:=20;
BEGIN
    WHILE var < 10 LOOP
        Suma:= suma + var;
        var:= var +1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('El resultado es: '||suma);
END;
/

```

FOR-LOOP

La sentencia FOR-LOOP especifica un rango de número enteros. Se ejecutarán tantas veces las sentencias del bucle como números enteros se especifiquen en el rango. Se puede referenciar la variable que definimos en el bucle pero no se puede asignar valores a esta variable.

```

FOR var IN rango LOOP
    Código PL/SQL
END LOOP;

```

```

DECLARE
    v_apellido      employees.last_name%TYPE;
BEGIN
    FOR numero IN 100..152 LOOP
        SELECT      last_name
        INTO    v_apellido
        FROM    Employees
        WHERE    employee_id = numero;
        dbms_output.put_line(rpad(numero,10,'-')||v_apellido);
    END LOOP;
END;
/

```

En caso de que el límite inferior y el límite superior sean iguales el bucle se ejecutará una sola vez. Siempre hay que especificar el valor inferior primero y el superior después. Si se desea que la secuencia de valores sea de mayor a menor se utilizará la cláusula REVERSE.

```

BEGIN
    FOR numero IN REVERSE 1..10 LOOP
        Dbms_output.put_line(numero);
    END LOOP;
END;
/

```

Los valores iniciales y finales pueden ser cualquier variable siempre que sea numérico. Se pueden utilizar variables que serán calculadas en el momento de la ejecución.

```

DECLARE
    v_numemple      PLS_INTEGER;
    v_departament_id employees.department_id%TYPE:=90;
BEGIN
    SELECT      count(*)
    INTO    v_numemple
    FROM    Employees
    WHERE    department_id=v_departament_id;

```

```
FOR numero IN 1..v_numemple LOOP    -- Si v_numemple=0 no ejecutará el
bucle
    Dbms_output.put_line('se ejecuta bucle for');
END LOOP;
END;
/
```

En Oracle 12g se introdujo la sentencia **CONTINUE**; marca el fin inmediato de una iteración de un bucle, y salta a la primera sentencia del bucle. El siguiente código ilustra su funcionamiento:

```
BEGIN
    FOR i IN 1..6 LOOP
        IF MOD(i,2) = 0 THEN
            CONTINUE;
        END IF;
        DBMS_OUTPUT.PUT_LINE ('i vale: ' || i);
    END LOOP;
END;
/
```

El resultado sería:

```
I vale : 1
I vale : 3
I vale : 5
```

Cada vez que el número es par, no se ejecuta la salida del literal, y salta a la siguiente iteración.

NULL

La sentencia NULL no realiza ninguna acción y pasa el control a la siguiente sentencia. Se utiliza en zonas en donde es necesario escribir código, pero no se quiere realizar ninguna acción.

```
DECLARE
    v_first_name employees.first_name%TYPE;
BEGIN
    SELECT first_name
    INTO   v_first_name
    FROM   employees
    WHERE  employee_id=10; --no existe el empleado 10
EXCEPTION
    WHEN too_many_rows THEN
        Dbms_output.put_line('demasiadas filas');
    WHEN no_data_found THEN
        NULL;           -- En este caso no se hace nada
END;
/
```

5

Cursores

Tabla de contenidos

Introducción	1
Declaración de un cursor	1
Apertura de un cursor	2
Lectura de un cursor.....	2
Cierre de un cursor.....	3
FOR de Cursor.	4
Atributos del Cursor	4
%FOUND	4
%ISOPEN	4
%NOTFOUND	4
%ROWCOUNT	4
Cursor FOR UPDATE.....	6

Introducción



Para poder seleccionar filas, PL/SQL utiliza los cursores. Los cursores estáticos pueden ser de 2 tipos:

- Implícitos: Todas las operaciones DML llevan asociadas un cursor implícito incluyendo las sentencias SELECT que devuelven una sola fila.
- Explícito: Todas las sentencias SELECT que devuelven más de una fila deben ser declaradas mediante la instrucción CURSOR y recorridos mediante un bucle FOR o usando la cláusula BULK COLLECT.

Al utilizar una SELECT-INTO nos limitamos a poder seleccionar una sola fila; al momento de necesitar seleccionar más de una fila, se utiliza cursores.

Más adelante se estudiará los cursores variables.

Declaración de un cursor

Se realiza en la zona de declaraciones, se define el nombre y la *query* asociada a él.

```
CURSOR Nombre_cursor [(parametro[, parametro]...)]  
[RETURN Tipo_valor] IS  
Sentencia_select;
```

La cláusula RETURN representa el tipo de datos que nos devolverá el cursor y PARAMETRO es un valor que se puede pasar al cursor y que hay que definir; son opcionales. Los parámetros no se pueden limitar con la precisión.

```
DECLARE  
CURSOR cur_trabajos IS  
SELECT job_id, job_title FROM Jobs;
```

El nombre de un cursor ha de ser único y no se puede utilizar variables para nombrar a los cursores. Los parámetros se pueden utilizar en la query asociada al cursor. Los valores pueden ser modificados y tendrán relevancia en el momento de la apertura del cursor, no en el momento de la declaración.

```
DECLARE
```

```
CURSOR cur_ciudades (inicio locations.location_id%TYPE
                    , final locations.location_id%TYPE)
IS
  SELECT      city
  FROM        Locations
  WHERE       location_id BETWEEN inicio AND final;
BEGIN
  ...
```

Apertura de un cursor

Al abrir un cursor se ejecuta la sentencia asociada y se identifican todas las filas que satisfagan la SELECT. En caso que exista la cláusula FOR UPDATE las filas quedarán bloqueadas.

```
...
BEGIN
  OPEN cur_trabajos; -- Apertura de un cursor
  ...
END;
/
```

Las filas son recuperadas en este momento. Para poder leerlas se utilizará la sentencia FETCH.

En el momento de la apertura del cursor se pueden pasar los parámetros para cambiar el grupo de filas seleccionadas. Si no se especifican se utilizarán los valores por defecto asignados en la declaración del cursor. Su valor se puede dar de forma posicional (implícita en el orden de la declaración) o nominal (explícita).

```
...
BEGIN
  -- Apertura de un cursor con parámetros
  OPEN cur_ciudades(inicio=>1500,final=>1800);
  ...
END;
/
```

Lectura de un cursor.

La lectura de las filas de un cursor se realiza con el comando FETCH (Buscar, traer). Cada vez que se realiza un comando FETCH se recupera una sola fila y se avanza a la siguiente fila para poder ser “traída” al programa PL/SQL.

```
FETCH nombre_del_cursor INTO registro_de_lectura;
```

Las variables que reciben la información del cursor han de ser compatibles con la definición del cursor tanto en tipo como en número. Ejemplo: Si un cursor devuelve un number y un varchar2, la instrucción FETCH ha de referenciar un number y un varchar2.

```
...
BEGIN
  ...
  LOOP
    FETCH cur_ciudades INTO v_ciudad; /*Se recupera la ciudad después de
    leer el cursor*/
```



```

        EXIT WHEN cur_ciudades%NOTFOUND; /*Cuando se acabe el cursor
saldremos del bucle*/
        DBMS_OUTPUT.PUT_LINE(v_ciudad);
    END LOOP;
    ...
END;
/

```

```

    ...
BEGIN
    ...
    LOOP
        FETCH cur_trabajos INTO v_job_id,v_job_title; /*Se recupera la
ciudad después de leer el cursor*/
        EXIT WHEN cur_trabajos%NOTFOUND; /*Cuando se acabe el cursor
saldremos del bucle*/
        DBMS_OUTPUT.PUT_LINE('Id Trabajo: '||v_job_id);
        DBMS_OUTPUT.PUT_LINE('Trabajo: '||v_job_title);
    END LOOP;    ...
END;
/

```

Para obtener otro conjunto de filas se debe cerrar el cursor y volverlo a abrir con los nuevos parámetros.

Existe la cláusula BULK COLLECT que lee TODAS las filas del cursor en la instrucción FETCH. Las variables del FETCH son tablas para que puedan recibir TODAS las filas que satisfagan la *query* asociada al cursor.

Cierre de un cursor.

El comando CLOSE cierra un cursor. Un cursor cerrado se puede volver a reabrir. Cualquier operación con un cursor cerrado levantará la excepción INVALID_CURSOR.

```

    ...
BEGIN
    ...
    CLOSE cur_ciudades;
    ...
END;
/

```

El código al completo sería:

```

DECLARE
    CURSOR cur_trabajos IS
        SELECT job_id,job_title
        FROM Jobs;
    V_job_id          jobs.job_id%TYPE;
    V_job_title       jobs.job_title%TYPE;
BEGIN
    OPEN cur_trabajos; -- Apertura de un cursor
    LOOP
        FETCH cur_trabajos INTO v_job_id,v_job_title;
        EXIT WHEN cur_trabajos%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Id Trabajo: '||v_job_id);
        DBMS_OUTPUT.PUT_LINE('Trabajo: '||v_job_title);
    END LOOP;
    CLOSE cur_trabajos;

```

```
END; /
```

FOR de Cursor.

Se puede simplificar el tratamiento de un cursor (OPEN, FETCH, CLOSE) mediante una sentencia FOR. La sentencia FOR declara implícitamente un registro %ROWTYPE del cursor; abre el cursor especificado; recorre las filas del cursor, coloca los valores en el registro y cierra el cursor cuando detecta el final del cursor.

```
DECLARE
    CURSOR cur_trabajos IS
        SELECT job_id, job_title FROM Jobs;
BEGIN
    FOR Reg IN cur_trabajos LOOP      /* Reg es del tipo cur_trabajos%ROWTYPE */
        DBMS_OUTPUT.PUT_LINE( 'ID Trabajo: ' || Reg.job_id);
        DBMS_OUTPUT.PUT_LINE( 'Trabajo: ' || Reg.job_title);
    END LOOP;      /* El cursor ha sido cerrado al finalizar el bucle */
END;
/
```

Atributos del Cursor

Los atributos del cursor permiten conocer el estado de las operaciones que se realizan con los cursores así como el estado del cursor.

%FOUND

Cuando se abre un cursor o un cursor variable, este atributo contiene el valor NULL. Cuando se ejecute el primer FETCH contendrá el valor FALSE si no se ha devuelto ninguna fila o TRUE si se ha devuelto una fila.

Si un cursor no ha sido abierto y se pregunta por este atributo se levantará la excepción INVALID_CURSOR.

%ISOPEN

Devolverá el valor TRUE si el cursor ha sido abierto o por el contrario FALSE si el cursor no ha sido abierto.

%NOTFOUND

Es el atributo opuesto a %FOUND. Será FALSE cuando se haya devuelto una fila en el FETCH o TRUE si no ha devuelto una fila. Contendrá el valor NULL si no se ha realizado ningún FETCH.

%ROWCOUNT

Cuando un cursor estático o un cursor variable, ha sido abierto; este atributo toma el valor cero. A medida que se van realizando FETCH el valor se va incrementando con el número de filas que devuelve cada FETCH.

Valores de los atributos de cursor:

%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
--------	---------	-----------	-----------

Before	OPEN	Exception	FALSE	Exception	Exception
After		NULL	TRUE	NULL	0
Before	Primer				
After	FETCH	TRUE	TRUE	FALSE	1
Before	siguientes				
After	FETCH	TRUE	TRUE	FALSE	Depende dato
Before	Ultimo				
After	FETCH	FALSE	TRUE	TRUE	Depende dato
Before	CLOSE				
After		Exception	FALSE	Exception	Exception

```

DECLARE
    CURSOR cur_emple IS
        SELECT      employee_id
        FROM        employees
        ORDER BY    hire_date ASC;
    v_employee_id employees.employee_id%TYPE;
    i              PLS_INTEGER:=1;
BEGIN
    IF cur_emple%ISOPEN THEN-- Está abierto?
        NULL;                -- No hacemos nada
    ELSE
        OPEN cur_emple;      -- Lo abrimos
    END IF;
    LOOP
        FETCH cur_emple INTO v_employee_id;    -- Leemos el cursor
        EXIT WHEN cur_emple%NOTFOUND;         --Salimos del bucle
        IF MOD(cur_emple%ROWCOUNT,3 ) = 0
        THEN
            --Llevamos un múltiplo de 3 filas leídas
            dbms_output.put_line('Grupo: '||i);
            i:=i+1;
        END IF;
        dbms_output.put_line(v_employee_id);
    END LOOP;
END;
/

```

Los mismos atributos se aplican a los cursores implícitos (Sentencias SQL que no son definidas por un cursor). Las sentencias INSERT, DELETE, UPDATE y SELECT tienen los mismos atributos, pero como nombre del cursor se les antepone la palabra SQL.

Select o DML	%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
Si no se ejecuta	NULL	FALSE	NULL	NULL
Afecta a una o varias filas	TRUE	FALSE	FALSE	Numero de filas afectadas
Otro caso	FALSE	FALSE	TRUE	

El atributo SQL%NOTFOUND no se utiliza por la sentencia SELECT INTO porque:

- Si no retorna filas es lanzada inmediatamente la excepción NO_DATA_FOUND antes de que pueda chequear el atributo.
- Una sentencia SELECT INTO que invoca a una función SQL de agregación, siempre retorna una fila (aunque sea NULL). Después de la sentencia el atributo es siempre FALSE, siendo innecesario chequear su valor.

SQL%ISOPEN siempre es FALSE porque se cierra después de ejecutar la sentencia asociada.

Si una sentencia SELECT INTO retorna múltiples filas salta la excepción TOO_MANY_ROWS; y el atributo SQL%ROWCOUNT retorna 1, no el número de filas que satisfacen la consulta. Sucede lo mismo con la clausula BULK COLLECT.

```
BEGIN
  DELETE departments
  WHERE department_id NOT IN
    (SELECT distinct department_id
     FROM employees
     WHERE department_id IS NOT NULL);
  dbms_output.put_line('Se eliminaron :'||SQL%ROWCOUNT||' departamentos.');
```

Estos atributos es para la última sentencia DML SQL ejecutada con lo que pueden inducir a error si se llaman a funciones o procedimientos inmediatamente después de una sentencia SQL. Estos procedimientos o funciones pueden ejecutar sentencias SQL y variarían los valores de los atributos que serían los producidos por las sentencias de la función, no por la propia sentencia SQL.

```
UPDATE Tabla SET Campo = Valor WHERE Condición;
Llamada_procedimiento (Parámetros);
IF SQL%NOTFOUND THEN
/* Puede que el procedimiento haya ejecutado alguna sentencia SQL y el
SQL%NOTFOUND no sea del UPDATE anterior a la llamada al procedimiento. */
```

Cursor FOR UPDATE.

```
DECLARE
  CURSOR registros IS
    SELECT employee_id,last_name, department_name
    FROM Employees e, Departments d
    WHERE e.department_id = d.depatment_id
    AND e.department_id=80
```

```
AND salary > 6000
FOR UPDATE NOWAIT;
```

FOR UPDATE es conveniente utilizarlo cuando se desee cambiar el valor de una columna o de varias columnas de una fila recuperada y se desee tener la seguridad de que no variará el valor desde su lectura hasta su modificación o eliminación.

La cláusula NO WAIT indica a Oracle que no se espera si las filas seleccionadas han sido requeridas por otro usuario. El control se devuelve al programa que puede realizar otras tareas antes de volver a intentar adquirir de nuevo el bloque de esas filas. Si no se especifica Oracle esperará hasta que las filas estén disponibles.

Al utilizar la cláusula FOR UPDATE en la SELECT que define el cursor, se indica a Oracle que bloquee todas las filas que cumplan la condición y por consiguiente todas las filas que forman parte del cursor. Se bloquean TODAS LAS FILAS hayan sido recuperadas ya con un FETCH o estén pendientes de ser recuperadas.

Las filas se liberarán cuando se realice un COMMIT o un ROLLBACK, por lo que no se puede realizar un FETCH de un cursor FOR UPDATE una vez se ha realizado un COMMIT o ROLLBACK.

En caso que en la SELECT estuvieran implicadas varias tablas se puede especificar un nombre de columna de la tabla de la cual se modificará o eliminará sus filas; con lo que sólo se bloquearan las filas de la tabla que contenga la columna o columnas señaladas. La cláusula para especificar las columnas es OF.

```
DECLARE
  CURSOR registros IS
    SELECT employee_id, last_name, department_name
    FROM Employees e, Departments d
    WHERE e.department_id = d.department_id
    AND e.department_id = 80
    AND salary > 6000
    FOR UPDATE OF salary;
/* Sólo se bloqueará la tabla EMPLOYEES puesto que es la que contiene la
columna SALARY. Sin especificar se bloquearían las filas afectadas
relacionadas, tanto de employees como de departments.*/
```

CURRENT OF permite modificar la fila del cursor que está siendo apuntada por el último FETCH.

No se admite añadir más condiciones en el where después del current of.

```
/* Tomando el cursor del ejemplo anterior */
...
BEGIN
  FOR reg IN Registros
  LOOP
    FETCH Registros INTO Reg;

    UPDATE employees
    SET salary = salary * 1.15 /* incrementamos salario un 15% */
    WHERE CURRENT OF Registros;
  END LOOP;
END;
```


6

Tratamiento de Errores

Tabla de contenidos

Introducción	1
Excepciones predefinidas	1
Ámbito de una excepción en PL/SQL	5
Propagación de las excepciones	6
Excepciones definidas por el usuario	8
Declaración de una excepción	8
Levantar excepciones con la sentencia RAISE	8
Asignar excepciones a errores Oracle	8
Definir mensajes de error de usuario	9
Recuperación del Código de Error y el Mensaje.....	10

Introducción



En PL/SQL un aviso o una condición de error, se llama excepción. Las excepciones pueden ser definidas internamente (por el sistema run-time) o las definidas por el usuario. Las excepciones definidas internamente tienen un nombre predefinido, por ejemplo ZERO_DIVIDE. Las excepciones definidas por el usuario tienen que ser declaradas con un nombre y del tipo EXCEPTION.

Cuando se produce un error se levanta una excepción (raise) y el control pasa a la zona de excepciones. Las excepciones internas son levantadas implícitamente por el servidor y las definidas por el usuario tienen que ser explícitamente levantadas mediante la instrucción RAISE.

Para tratar las excepciones se escriben rutinas separadas en la zona de excepciones. Después de la ejecución de la rutina de la excepción, el bloque que la produjo se finaliza y se devuelve el control a la siguiente sentencia. Si no está contenido por un bloque, retorna el control al ambiente donde se ha producido la invocación a la rutina.

Excepciones predefinidas

Una excepción interna es levantada (Raised) cada vez que el programa PL/SQL viola una regla o excede un límite de Oracle. Cada error de Oracle se expresa mediante un número, pero en PL/SQL las excepciones deben tratarse por un nombre.

Para poder interceptar cualquier otro error de Oracle (que no están predefinidos) una alternativa es utilizar la excepción OTHERS.

Las funciones SQLCODE y SQLERRM identifican el código y el mensaje del error producido. Se pueden asignar nombres de excepciones a códigos de error de Oracle mediante la acción PRAGMA EXCEPTION_INIT.

Nombre excepción	Error Oracle	Valor SQLCODE
------------------	--------------	---------------

Nombre excepción	Error Oracle	Valor SQLCODE
ACCESS INTO NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Nombre Excepción	Descripción de la excepción
ACCESS INTO NULL	Se intenta asignar valores un atributo de un objeto que contiene nulos.

Nombre Excepción	Descripción de la excepción
CASE_NOT_FOUND	Ninguna de la opciones WHEN de una sentencia CASE ha sido seleccionada y no existe la cláusula ELSE
COLLECTION_IS_NULL	Se intenta aplicar métodos de colección diferentes a EXIST a una tabla anidada o un Varray y ésta contiene valores nulos o no está inicializada.
CURSOR_ALREADY_OPEN	Se intenta abrir un cursor que ya está abierto.
DUP_VAL_ON_INDEX	Se intenta guardar un valor duplicado en un índice que no permite valores duplicados.
INVALID_CURSOR	Se intenta realizar una operación sobre un cursor que está cerrado.
INVALID_NUMBER	En un comando SQL la conversión de una cadena alfanumérica a un número es incorrecta ya que no representa un número válido. En PL/SQL levanta la excepción VALUE_ERROR.
LOGIN_DENIED	Se intenta conectarse a Oracle con un usuario y una contraseña incorrectos.
NO_DATA_FOUND	<p>Un SELECT INTO no devuelve filas, o se referencia a un elemento borrado en una tabla anidada o un elemento no inicializado en una tabla indexada.</p> <p>Las funciones agregadas de grupo (AVG, SUM, COUNT,etc.) siempre devuelven un nulo o un cero por lo que un comando SELECT con funciones agregadas nunca levantará esta excepción.</p> <p>Un comando FETCH puede que no devuelva filas por lo que no levantará esta excepción en el caso de que no devuelva ninguna fila.</p>
NO_DATA_FOUND	<p>Un SELECT INTO no devuelve filas, o se referencia a un elemento borrado en una tabla anidada o un elemento no inicializado en una tabla indexada.</p> <p>Las funciones agregadas de grupo (AVG, SUM, COUNT,etc.) siempre devuelven un nulo o un cero por lo que un comando SELECT con funciones agregadas nunca levantará esta excepción.</p> <p>Un comando FETCH puede que no devuelva filas por lo que no levantará esta excepción en el caso de que no devuelva ninguna fila.</p>

Nombre Excepción	Descripción de la excepción
NOT_LOGGED_ON	Se intenta realizar una llamada a una base de datos sin estar conectado a ella.
PROGRAM_ERROR	Se ha producido un error interno de PL/SQL
ROWTYPE_MISMATCH	La host variable (Ejemplo: variable de un programa 3GL) y la variable de un cursor PL/SQL no son del mismo tipo.
SELF_IS_NULL	Se intenta usar el método MEMBER a una instancia nula.
STORAGE_ERROR	Falta de recursos de memoria o está corrupta.
SUBSCRIPT_BEYOND_COUNT	Se intenta referenciar a un elemento de una tabla anidada o un varray utilizando un valor mayor que el número de elementos de la tabla.
SUBSCRIPT_OUTSIDE_LIMIT	Se intenta referenciar a un elemento de una tabla anidada o un varray utilizando un valor que está fuera del rango permitido(Ejemplo el valor -1)
SYS_INVALID_ROWID	La conversión de una cadena alfanumérica a un tipo rowid universal es incorrecta porque no representa un valor válido.
TIMEOUT_ON_RESOURCE	Se ha producido un time-out (exceso de tiempo) esperando un recurso.
VALUE_ERROR	Se ha producido un error en una operación aritmética, conversión, truncamiento o límite de precisión. Ejemplo: La columna que recibe un valor de un comando SELECT INTO es menor que el tamaño de la columna de la base de datos.
ZERO_DIVIDE	Se ha producido una división por cero. (No existe el valor infinito en Oracle)

En el siguiente ejemplo se muestra como se utilizan las excepciones predefinidas:

```

DECLARE
    v_apellido VARCHAR2(25);
BEGIN
    INSERT INTO departments(department_id,department_name)
    VALUES(departments_seq.NEXTVAL, 'I+D+i');

    SELECT      last_name
    INTO        v_apellido
    FROM        employees
    WHERE       employee_id = 280;--no existe, provoca excepción

```

```

INSERT INTO locations(location_id,city)
VALUES(locations_seq.NEXTVAL,'Madrid');

EXCEPTION
  WHEN NO_DATA_FOUND THEN      --no se ha encontrado datos
    DBMS_OUTPUT.PUT_LINE('ERROR: ' || SQLCODE || 'MENSAJE: ' || SQLERRM);
END;
/

```

En este ejemplo al dar tratamiento a la excepción, el insert realizado anteriormente en la tabla departments se ha realizado (pendiente de hacer commit o rollback); es decir, que si consulta la tabla estará la nueva fila. Mientras que si no se trata la excepción, el insert automáticamente se deshace.

Al saltar la excepción por la sentencia select, el insert en la tabla locations nunca se llega a realizar; se trate o no la excepción.

Cuando se da una excepción, salta el código que tenga a continuación y va directamente a la zona de tratamiento de excepciones a buscar el tratamiento.

Ámbito de una excepción en PL/SQL

No se puede declarar dos veces una excepción en el mismo bloque. Sin embargo, se puede declarar la misma excepción en dos bloques diferentes. Una excepción declarada en un bloque es considerada local al bloque y global a todos sus sub-bloques.

Un bloque puede referenciar solamente excepciones locales o globales, pero no puede referenciar a excepciones declaradas en sus sub-bloques.

```

<<bloque>>
DECLARE
  Error1 EXCEPTION;
  Error2 EXCEPTION;
BEGIN
  <<subbloque>>
  DECLARE
    Error1 EXCEPTION;
  BEGIN
    -- inicio del sub-bloque
    IF false THEN
      -- levantamos la excepción Error1 del sub-bloque
      RAISE Error1;
    ELSIF true THEN
      -- levantamos la excepción Error1 del bloque
      RAISE bloque.Error1;
    ELSIF true THEN
      -- levantamos la excepción Error2 del bloque
      RAISE Error2;
    END IF;
  EXCEPTION
    WHEN Error1 THEN
      -- tratamiento de la excepción Error1
      Dbms_output.put_line('Error1 sub-bloque.');
```

```
END;
/
```

En el sub-bloque se levantan tres excepciones:

- *Error1* se trata en el sub-bloque y podría propagarse -siguiente tema-; podría utilizarse la etiqueta del sub-bloque, pero implícitamente el ámbito es el del bloque local.
- *bloque.Error1* al anteponer el nombre de la etiqueta, explícitamente dejamos claro que es *Error1* del bloque.
- *Error2* su ámbito abarca todo el bloque por lo tanto dentro del sub-bloque puede ser levantada, e incluso tratada. Pero en este ejemplo es tratada en el bloque.

Realizar modificaciones para probar las posibilidades.

Propagación de las excepciones

Cuando se produce una excepción se busca en la zona de excepciones del bloque local. Si no se encuentra se propaga al bloque superior y así sucesivamente. En caso que no hubiera ningún tratamiento de esta excepción se devolvería el control al entorno como una excepción no manejada (unhandled exception).

Las excepciones no se propagan en los RPC (Remote Procedure Call) por lo que PL/SQL no puede captar una excepción levantada por un subprograma remoto.

Una excepción se puede volver a levantar una vez tratada. Si se produce una excepción y es tratada localmente en el bloque se puede volver a levantar. Se hace mediante el comando RAISE, si el RAISE no se acompaña del nombre de excepción, entonces se tratará en el bloque con el mismo nombre de tratamiento del sub-bloque y que se vuelva a tratar en el bloque donde se encuentra el sub-bloque que la ha provocado.

```
DECLARE
    Excepcion_cualquiera EXCEPTION;
BEGIN
    -- Empieza el sub-bloque1
    BEGIN
        IF TRUE THEN
            -- levantamos la excepción
            RAISE Excepcion_cualquiera;
        END IF;
    EXCEPTION
        WHEN Excepcion_cualquiera THEN
            -- Manejo del error
            Dbms_output.put_line('sub-bloque1');
    RAISE;
        /* Re-levantamos la excepción
           para pasarla al bloque superior */
    END;
    --fin del sub-bloque1
    --inicio sub-bloque2
    BEGIN
        RAISE Excepcion_cualquiera;
    EXCEPTION
        WHEN Excepcion_cualquiera THEN
            -- Manejo del error
            Dbms_output.put_line('sub-bloque1');
    RAISE;
    END;
    --fin sub-bloque2
```


EXCEPTION

```

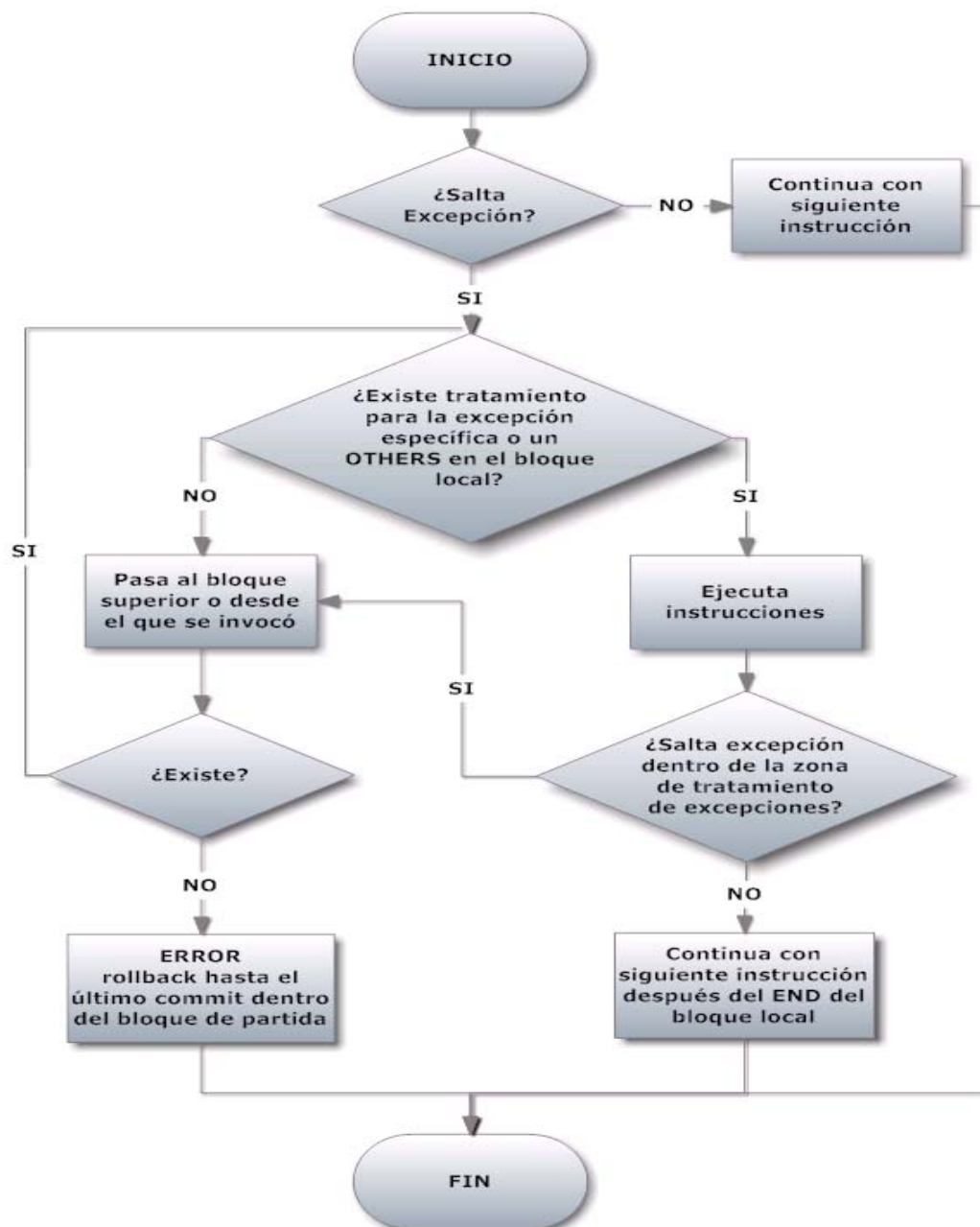
WHEN Excepcion_cualquiera THEN
    -- manejo del error de una
    -- manera diferente
    dbms_output.put_line('bloque');

```

END;

/Si se produce una excepción en la zona de declaraciones no se trata localmente si no en el bloque que engloba al sub-bloque donde se produjo la excepción. Del mismo modo si se produce en la zona de excepciones no se tratará localmente; distinto es que exista un sub-bloque en la zona de excepciones con su respectiva zona de excepciones.

Para darle un mejor entendimiento sobre el análisis de una instrucción; tiene el siguiente diagrama de flujo.



Excepciones definidas por el usuario

A diferencia de las excepciones predefinidas, las excepciones definidas por el usuario han de ser levantadas con la sentencia RAISE.

Declaración de una excepción

Se definen únicamente en la parte declarativa de un bloque PL/SQL, subprograma o paquete.

La sintaxis para declarar una excepción es:

```
DECLARE
  Nombre_excepcion EXCEPTION;
```

No se pueden declarar dos excepciones con el mismo nombre en el mismo bloque pero sí se pueden declarar dos excepciones con el mismo nombre en bloques diferentes.

Se consideran locales y no se pueden referenciar excepciones fuera del bloque.

Levantar excepciones con la sentencia RAISE

Los bloques PL/SQL y subprogramas deberían levantar las excepciones únicamente cuando un error hace imposible o indeseable la terminación de un proceso. Para levantar una excepción se utiliza el comando RAISE. Se puede utilizar en cualquier punto de la parte procedural del subprograma.

```
RAISE Nombre_excepcion;
```

También se puede levantar una excepción de una excepción predefinida de Oracle.

```
DECLARE
  Error1      EXCEPTION;
  cantidad    PLS_INTEGER := -5;
BEGIN
  IF cantidad = 50 THEN
    RAISE Error1;           -- levantamos la excepción
  END IF;

  IF cantidad < 0 THEN
    RAISE INVALID_NUMBER; -- levantamos la excepción
  END IF;
EXCEPTION
  WHEN Error1 THEN
    Dbms_output.put_line(SQLERRM);
  WHEN INVALID_NUMBER THEN
    Dbms_output.put_line(SQLERRM);
END;
/
```

Asignar excepciones a errores Oracle

Para poder tratar y capturar errores Oracle que no tienen predefinida una excepción se utiliza la excepción OTHERS o la acción PRAGMA EXCEPTION_INIT.

PRAGMA EXCEPTION_INIT indica al compilador que asocie una excepción a un código de error Oracle. Se definen únicamente en la parte declarativa de un bloque PL/SQL, subprograma o paquete.

La sintaxis para declararlo es:

```
PRAGMA EXCEPTION_INIT(nombre_excepción, Número_error_Oracle)
```

El nombre_excepción ha de haber sido declarado anteriormente en la zona declarativa.

Definir mensajes de error de usuario

El procedimiento RAISE_APPLICATION_ERROR permite levantar la excepción y definir mensajes de error del tipo ORA- por el usuario. La sintaxis es:

```
RAISE_APPLICATION_ERROR (numero_error, mensaje[, {TRUE | FALSE}]);
```

El número_error tiene el rango de -20000 a -20999 y mensaje puede tener una longitud máxima de 2048 bytes.

El tercer parámetro (TRUE|FALSE) es opcional.

TRUE el mensaje se almacena en una pila de errores.

FALSE (por defecto) el error reemplaza los errores previos.

RAISE_APPLICATION_ERROR está definido en el paquete DBMS_STANDARD por lo que se puede invocar desde cualquier programa o subprograma PL/SQL almacenado.

Cuando RAISE_APPLICATION_ERROR es invocado el subprograma acaba y devuelve el número de error y el mensaje a la aplicación.

Se puede tratar como cualquier error Oracle.

```
DECLARE
    Salary_alto  EXCEPTION;
    salario      employees.salary%TYPE;
    PRAGMA EXCEPTION_INIT(Salary_alto, -20104);
BEGIN
    SELECT      salary
    INTO        salario
    FROM        employees
    WHERE       employee_id = 120;

    IF salario > 7999 THEN
        Raise_application_error(-20104, 'El salario es Alto');
    END IF;

EXCEPTION
    WHEN Salary_alto THEN
        DBMS_OUTPUT.PUT_LINE('ERROR: ' || SQLERRM);
END;
/
```

En este ejemplo, podemos obtener el código de error de la excepción (-20104) y el mensaje de la misma ('El salario es Alto').

Con el RAISE_APPLICATION_ERROR podemos asignar diferentes mensajes en diferentes partes del bloque o procedimiento, al mismo código de error.

```
DECLARE
    v_salario      employees.salary%TYPE:=20000;
    v_comm employees.commission_pct%TYPE:=0.05;
BEGIN
    IF v_salario>10000 THEN
        raise_application_error(-20400,'El salario es muy alto');
```

```

ELSE
    INSERT INTO employees(employee_id
                        ,last_name
                        ,email
                        ,hire_date
                        ,job_id
                        ,salary)
        VALUES (employees_seq.nextval
                , 'Smith'
                , 'smith'
                , sysdate
                , 'IT_PROG'
                , v_salario);

END IF;
IF v_comm > 0.07 THEN
    raise_application_error(-20400,'La comisión es muy alta.');
```

```

ELSE
    UPDATE      employees
    SET          commission_pct=v_comm
    WHERE        department_id=30;

END IF;
END;
/
```

Recuperación del Código de Error y el Mensaje.

En la rutina que maneja la excepción se puede hacer referencia a las funciones `SQLCODE` y `SQLERRM`, las cuales devuelven el código de error y el mensaje asociado a ese código de error. Para excepciones internas el código siempre será un valor negativo (salvo `NO_DATA_FOUND` que devuelve +100).

Para excepciones definidas por el usuario `SQLCODE` devuelve +1 y `SQLERRM` devuelve "User-Defined Exception".

Si se utiliza `PRAGMA EXCEPTION_INIT` las funciones `SQLCODE` y `SQLERRM` devolverán los valores asignados en el `PRAGMA`.

Se puede pasar un valor a `SQLCODE` para que `SQLERRM` nos devuelva el mensaje asociado a ese error. Para evitar resultados inesperados se le debe suministrar valores negativos.

No se pueden utilizar `SQLCODE` y `SQLERRM` directamente en sentencias SQL. Se han de asignar los valores devueltos por estas funciones a una variable y después utilizar estas variables.

```

DECLARE
    v_error      VARCHAR2(2000);
BEGIN
    INSERT INTO locations(location_id,city)
    VALUES(3000,'Madrid');      --Viola constraint HR.LOC_ID_PK
    INSERT INTO departments(department_id,department_name)
    VALUES(80,'IT');           --Viola constraint HR.DEPT_ID_PK
EXCEPTION
    WHEN OTHERS THEN
        --almacenamos el mensaje de error
        --ya que no podemos utilizarlo
        --directamente en sentencia SQL
        v_error:=SQLERRM;
        DECLARE
            --variable auxiliar, por sintaxis.
            v_aux      PLS_INTEGER;
        BEGIN
            --buscamos en el mensaje de error
            --el nombre de la constraint
```

```
--para identificar el insert que falla
SELECT 1
INTO    v_aux
FROM    dual
WHERE   v_error LIKE '%HR.LOC_ID_PK%';
--la select devuelve una fila, continua
DBMS_OUTPUT.PUT_LINE('No se puede insertar en LOCATIONS.');
```

EXCEPTION

```
--la select no encuentra datos
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No se puede insertar en
DEPARTMENTS.');
```

END;

END;

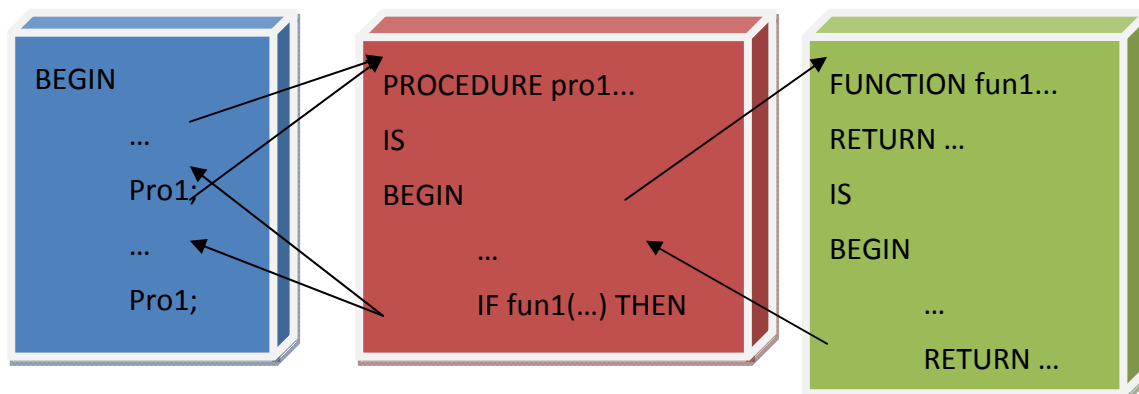
/

Procedimientos y Funciones

Tabla de contenidos

Introducción	1
Ventajas de los subprogramas.....	1
Procedimientos	1
Creación y Modificación	1
Ejecución	4
Eliminar un procedimiento	5
Funciones	5
Ejecución	6
Eliminar una Función	7
Efectos colaterales de las funciones.....	7
Privilegios.....	8
Cláusula BEQUEATH en las vistas	8

Introducción



Los subprogramas son bloques PL/SQL con un nombre, que se les pueden pasar parámetros y pueden ser invocados. Estos bloques se guardan en la base de datos una vez creados, estén o no sin errores de compilación.

PL/SQL tiene dos tipos de subprogramas: los procedimientos, que se utilizan para ejecutar una acción; y las funciones, que siempre retornan un valor.

Los subprogramas tienen:

- Las especificaciones: donde se define el tipo del subprograma (procedure o function), el nombre del mismo y los parámetros de entrada y/o salida.

Los parámetros son opcionales.

- Una parte declarativa (No se usa la cláusula DECLARE): donde se definen todos los elementos que formarán parte del subprograma: variables, constantes, cursores, tipos de datos, subprogramas, etc.
- Una parte de Ejecución: que comienza con BEGIN donde se realizan todas las acciones, sentencias de control y sentencia SQL.
- Una parte de excepciones o de control de errores (opcional), se especifican las acciones a tomar en caso que se produzca un error en la ejecución del subprograma.

Ventajas de los subprogramas.

- Modularidad: permite resolver la aplicación en módulos pequeños y específicos.
- Extensibilidad: se puede adaptar el lenguaje PL/SQL a las necesidades del usuario. Por ejemplo, crear un procedimiento que inserte una fila en una tabla.
- Los subprogramas son reusables y de fácil mantenimiento. Una vez que se ha probado un subprograma se puede utilizar en múltiples aplicaciones. Si su definición cambia sólo hay que modificar el subprograma afectado.

Procedimientos

Creación y Modificación

La sintaxis para la definición de un procedimiento es:

```

CREATE [OR REPLACE]
PROCEDURE Nombre_procedimiento [Declaración de parámetros]
  [AUTHID {DEFINER | CURRENT_USER}]
  {IS | AS}
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  [Declaraciones locales de tipos, variables, etc]
BEGIN
  Sentencias ejecutables del procedimiento
[EXCEPTION
  Excepciones definidas y las acciones de estas excepciones]
END [Nombre_procedimiento];
/

```

CREATE permite crear un procedimiento STANDALONE (No forma parte de un paquete) y guardarlo dentro de la base de datos. Si se crea un procedimiento que ya existe dará error por ello se utiliza la cláusula OR REPLACE. Si el procedimiento no existe se creará y si ya existe se reemplazará.

La cláusula AUTHID determina si un procedimiento se ejecuta con los privilegios del usuario que lo ha creado (por defecto) o si con los privilegios del usuario que lo invoca. También determina si las referencias no cualificadas las resuelve en el esquema del propietario del procedimiento o de quién lo invoca. Para especificar que se ejecute con los permisos de quién lo invoca se utiliza la cláusula CURRENT_USER.

Pragma AUTONOMOUS_TRANSACTION marca el procedimiento como autónomo. Un procedimiento autónomo permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado. Si en ejecución no se encuentra la sentencia COMMIT o ROLLBACK provocará una excepción.

No se usa la cláusula DECLARE puesto que va implícita en el IS o el AS.

No existe diferencia en utilizar el IS o el AS.

Un ejemplo sencillo de un procedimiento que incrementa el salario de los empleados un 1%, y muestra un mensaje.

```

CREATE OR REPLACE
PROCEDURE incremento1
IS
BEGIN
  UPDATE employees
  SET    salary=salary*1.1;
  Dbms_output.put_line('Se ha incrementado un 1% el salario.');
```

END;

/

El mensaje que recibirá es, que el procedimiento se ha creado correctamente o con errores de compilación. Si hay errores, para poder visualizar la pila de errores en sqlplus, se escribe SHOW ERROR.

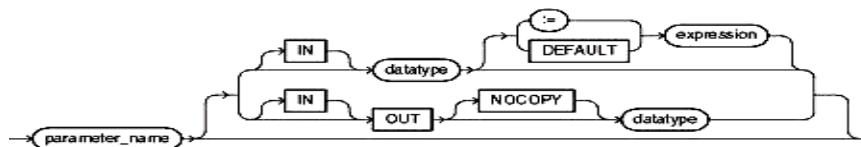
Una vez creado correctamente probamos el procedimiento llamándolo desde un bloque anónimo.

```

BEGIN
  Incremento1;
END;
/

```

Los parámetros tiene la siguiente sintaxis:



Se pueden especificar:

- **IN.** Parámetros de entrada. Suministra valores al procedimiento y el valor se trata como si fuera una constante. No se puede modificar en el valor dentro del subprograma. Se puede especificar un valor por defecto por si no es suministrado al invocar el subprograma.
- **OUT.** Parámetros de salida. Devuelve un valor al programa que invoca el subprograma. Dentro del subprograma se trata como si fuera una variable.

Se puede especificar un valor por defecto al parámetro OUT en el momento de la invocación pero este valor se pierde si no se utiliza la opción NOCOPY. En el caso que se produjera un error en el subprograma y no se tratara (Una excepción sin su tratamiento) el valor no se perdería aún no habiendo especificado NOCOPY.

Si no se especifica dentro del subprograma un valor al parámetro OUT contendrá el valor NULL (nulo).

- **IN OUT.** Parámetro de entrada y Salida. Se inicializa en el momento de invocar el subprograma y se trata como una variable dentro de él.

Los parámetros no se pueden limitar con la precisión.

```
PROCEDURE Proceso (Parametro CHAR(20)) IS ...           -- Provocaría un error.
```

Se puede definir un subtipo de dato y utilizarlo si es necesario utilizar precisión.

```
DECLARE
  SUBTYPE Cadena20 IS CHAR(20);
  PROCEDURE Proceso (Parametro Cadena20) IS           -- Correcto
```

Los parámetros se pueden suministrar de manera posicional o nombrada

```
DECLARE
  param1 INTEGER;
  param2 REAL;
  PROCEDURE proce (param1_no INTEGER, param2_no REAL) IS ...
BEGIN
  proce(param1, param2);                               -- posicional
  proce(param2_no => param2, param1_no => param1);      -- nombrada
  proce(param1_no => param1, param2_no => param2);      -- nombrada
  proce(param1, param2_no => param2);                  -- mixta
```

Un ejemplo de un procedimiento con parámetros.

```
CREATE OR REPLACE
PROCEDURE alta_depart
  (p_department_name
    departments.department_name%TYPE
  ,p_manager_id
    departments.manager_id%TYPE      DEFAULT NULL
  ,p_location_id
    departments.location_id%TYPE     DEFAULT NULL
  ,p_department_id
    departments.department_id%TYPE   DEFAULT NULL
```

```

    )
IS
    v_department_id          departments.department_id%TYPE;
BEGIN
    IF p_department_id IS NULL THEN
        v_department_id :=departments_seq.NEXTVAL;
    END IF;

    INSERT INTO departments(department_id
                           ,department_name
                           ,manager_id
                           ,location_id)
    VALUES(v_department_id
           ,p_department_name
           ,p_manager_id
           ,p_location_id);
END;
/

```

Ejemplo de prueba del PRAGMA AUTONOMOUS_TRANSACTION

```

CREATE OR REPLACE
PROCEDURE Llamador
IS
BEGIN
    UPDATE departments
    SET location_id=1700;           -- Se inicia una transacción
    --Se invoca un procedimiento autónomo
    Procedimiento_autonomo;

    ROLLBACK;                     -- Se deshará el UPDATE no el INSERT.
END Llamador;
/

CREATE OR REPLACE
PROCEDURE Procedimiento_autonomo
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO locations(location_id,city)
    VALUES (locations_seq.nextval,'Madrid');

    COMMIT; /* Sólo afectará al INSERT del procedimiento autónomo*/
END Procedimiento_autonomo;
/

```

Ejecución

Para ejecutar el procedimiento una vez que esta creado correctamente, simplemente hay que llamarlo por el nombre y sus parámetros, desde otro subprograma, paquete o bloque anónimo:

```

BEGIN
    alta_depart (p_department_name =>'Social Media'
               ,p_location_id      =>1700);
END;
/

```

Otra alternativa es escribir en sqlplus

```

EXECUTE Alta_depart
(p_department_name => 'IT'
,p_manager_id      =>    100
,p_location_id     =>    1700

```

```
,p_department_id    =>    DEFAULT
);
```

Eliminar un procedimiento

Para eliminar un procedimiento, la sintaxis es:

```
DROP PROCEDURE nombre_procedimiento;
```

Funciones

Una función es un subprograma que calcula un valor. Las funciones difieren principalmente de los procedimientos en que siempre retornan un valor mediante la instrucción RETURN.

La sintaxis para crear una función es:

```
CREATE [OR REPLACE]
FUNCTION Nombre_Función
    [(declaracion_parámetro
    [, declaracion_parámetro]...)]
RETURN Tipo_dato
    [AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
    [DETERMINISTIC]
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [Declaraciones locales de tipos, variables, etc]
BEGIN
    /*Sentencias ejecutables*/
[EXCEPTION
    --Excepciones definidas y las acciones de estas excepciones]
END [Nombre_Función];
```

CREATE permite crear una función STANDALONE (No forma parte de un paquete) y guardarla dentro de la base de datos. Si se crea una función que ya existe dará error por ello se utiliza la cláusula OR REPLACE. Si la función no existe se creará y si ya existe se reemplazará.

RETURN, en la especificación de la función, indica el tipo de dato a devolver.

La cláusula AUTHID determina si una función se ejecuta con los privilegios del usuario que la ha creado (por defecto) o sí con los privilegios del usuario que la invoca. También determina si las referencias no cualificadas las resuelve en el esquema del propietario de la función o de quién lo invoca. Para especificar que se ejecute con los permisos de quién la invoca se utiliza la cláusula CURRENT_USER.

El hint (Marca o parámetro) DETERMINISTIC ayuda al optimizador a evitar llamadas redundantes. Si una función almacenada ha sido anteriormente invocada con los mismos parámetros el optimizador puede escoger devolver el mismo valor.

La pragma AUTONOMOUS_TRANSACTION marca la función como autónoma. Una función autónoma permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado.

No se usa la cláusula DECLARE puesto que va implícita en el IS o el AS.

No existe diferencia en utilizar el IS o el AS

```
/*Media de salarios de los empleados de un departamento*/
CREATE OR REPLACE
```

```

FUNCTION Media_Salarios
    (p_department_id    departments.department_id%type)
RETURN NUMBER          -- especificamos el tipo de datos a retornar
IS
    v_media_salarios    PLS_INTEGER;
BEGIN
    SELECT AVG(salary)
    INTO   v_media_salarios
    FROM   Employees
    WHERE   department_id= p_department_id;
    RETURN v_media_salarios; -- Valor de retorno
END;
/

```

Ejecución

Las funciones son invocadas como parte de una expresión y pueden ser invocadas desde múltiples sitios.

```

BEGIN
    IF Media_Salarios(20) > 3000 THEN
        Dbms_output.put_line('mayor a 3000');
    END IF;
END;
/

```

```

SELECT first_name
        ,last_name
        ,salary
        ,salary+0.02*Media_Salarios(20) "NewSalary"
FROM employees;

```

```

DECLARE
    x      number;
BEGIN
    x:= Media_Salarios(p_department_id=>80);
    dbms_output.put_line('Media: '||x);
END;
/

```

La sentencia RETURN (No la de la parte de la especificación de la función, donde especificamos el tipo de dato que se devuelve) finaliza la ejecución de la función y devuelve el control.

Una función puede contener varias sentencias RETURN; aunque no se aconseja, puesto que denota una programación pobre.

La sentencia RETURN no tiene porque ser la última sentencia del subprograma o función.

En los procedimientos la sentencia RETURN no puede devolver ningún valor, simplemente devuelve el control al programa llamador antes de que finalice el procedimiento.

En las funciones, la sentencia debe devolver un valor. Este valor se evalúa en el momento de devolverlo, por lo que puede ser una expresión.

```

RETURN 100 + ( Valor1 ** (4/(Valor2)+2)) -- Sería correcto

```

Eliminar una Función

Para eliminar un procedimiento, la sintaxis es:

```
DROP FUNCTION nombre_funcion;
```

Efectos colaterales de las funciones.

Para que una función pueda ser llamada desde una sentencia SQL, ha de cumplir los siguientes requisitos:

Cuando se llama desde un SELECT o una sentencia INSERT, UPDATE o DELETE ejecutada paralelamente, una función no puede modificar ninguna tabla de la base de datos.

Cuando se llama desde una sentencia INSERT, UPDATE o DELETE, una función no puede consultar o modificar ninguna tabla que aparezca en la sentencia SQL.

Cuando se llama desde una sentencia SELECT, INSERT, UPDATE o DELETE, la función no puede ejecutar ningún comando transaccional (COMMIT, ROLLBACK, etc.), un comando de sesión (SET ROLE, etc.) o de control de sistema (ALTER SYSTEM). Tampoco puede realizar ningún comando DDL, (CREATE, DROP, etc.) puesto que llevan un COMMIT implícito.

Si se quebrantan estas reglas, se producirá un error en tiempo de ejecución.

Para evitar las reglas anteriores se puede utilizar la directiva de compilación PRAGMA RESTRICT_REFERENCES que indica a los paquetes las limitaciones que tienen las funciones.

Para eliminar una función, la sintaxis es:

```
DROP FUNCTION nombre_funcion;
```

Para otorgar permiso de ejecución a un usuario, la sintaxis es:

```
GRANT EXECUTE nombre_funcion TO nombre_usuario;
```

Para quitar el permiso de ejecución del procedimiento a un usuario;

```
REVOKE EXECUTE nombre_funcion FROM nombre_usuario;
```


Privilegios



El usuario debe poseer el privilegio de sistema **CREATE PROCEDURE** para poder; crear, modificar o eliminar sus procedimientos y funciones. El mismo privilegio le permite otorgar permiso de ejecución a otros usuarios sobre sus procedimientos o funciones.

Para otorgar permiso de ejecución sobre un procedimiento o función a otro usuario, la sintaxis es:

```
GRANT EXECUTE ON [esquema.]nombre_subprograma TO nombre_usuario;
```

Para quitar el permiso de ejecución del procedimiento a un usuario;

```
REVOKE EXECUTE ON [esquema.] nombre_subprograma FROM nombre_usuario;
```

Tanto el propietario como cualquier usuario que tenga el privilegio **CREATE ANY PROCEDURE** pueden otorgar permiso de ejecución sobre los procedimientos o funciones.

CREATE ANY PROCEDURE permite crear procedimientos o funciones en otros esquemas, **ALTER ANY PROCEDURE** permite modificarlos y **DROP ANY PROCEDURE** permite eliminarlos.

En las vistas **USER_SOURCE**, **ALL_SOURCE** del diccionario de datos, encontramos el código de los procedimientos, funciones, paquetes y disparadores.

Cláusula **BEQUEATH** en las vistas

Oracle Database 12c proporciona la posibilidad de indicar si una función que ha sido incluida en la definición de una vista debe ser ejecutada usando los privilegios del propietario de la vista o los privilegios del usuario que invoca a la vista.

Esta funcionalidad está soportada a través del uso de la cláusula **BEQUEATH** del comando **CREATE VIEW**.

```
CREATE [OR REPLACE] VIEW nombre_vista  
BEQUEATH { CURRENT_USER | DEFINER }  
AS consulta;
```

Opciones:

- **CURRENT_USER:** Si especifica esta cláusula, entonces las funciones que hace referencia la vista se ejecutan utilizando los derechos del usuario solicitante (el que invoca), siempre y cuando una de las siguientes condiciones se cumple:
 - El propietario de la vista tiene el privilegio de objeto INHERIT PRIVILEGES en el usuario que realiza la invocación.
 - El propietario de la vista tiene el privilegio del sistema INHERIT ANY PRIVILEGES.
- **DEFINER:** Si es especificada, las funciones que hace referencia la vista se ejecutan utilizando los derechos de la vista del creador. Esta es la opción predeterminada.

La resolución de nombres dentro de la vista se maneja utilizando el esquema del propietario de la vista.

Ejemplos:

1. Crea una función que devuelve la cantidad de registros de una tabla (*departments*) en el esquema del usuario HR:

```
CONNECT HR/oracle

CREATE OR REPLACE FUNCTION fContador RETURN NUMBER
  AUTHID CURRENT_USER
AS
  contador NUMBER;
BEGIN
  SELECT count(*) INTO contador
  FROM departments;
  RETURN contador;
END fContador;
/
```

2. Crea una vista que utiliza la función anterior:

```
CREATE OR REPLACE VIEW vContador
  AS SELECT fContador FROM dual;
```

3. Concede el privilegio de consulta sobre la vista a otro usuario.

```
GRANT SELECT ON vContador TO alumno;
```

4. Consulta la vista anterior desde ese otro usuario:

```
SQL> CONNECT alumno/curso

SQL> SELECT * FROM hr.vContador;

 FCONTADOR
-----
          27
```

5. Comprueba que el usuario no tiene acceso a la tabla *departments*. Pero como por defecto la vista tiene los permisos del creador se puede ejecutar correctamente.

```
SQL> DESCRIBE hr.departments
ERROR:
ORA-04043: el objeto hr.departments no existe
```

6. Desde el usuario HR modifica la vista para que en su ejecución tome los permisos del usuario que ejecuta el comando.

```
CONNECT HR/oracle

CREATE OR REPLACE VIEW vContador
  BEQUEATH CURRENT_USER
  AS SELECT fContador FROM dual;
```

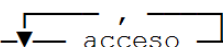
7. Consulta de nuevo la vista anterior desde ese otro usuario. Comprueba que ahora da error por los privilegios:

```
SQL> CONNECT alumno/curso

SQL> SELECT * FROM hr.vContador;
SELECT * FROM hr.vContador
      *
ERROR at line 1:
ORA-00942: la tabla o vista no existe
ORA-06512: en "HR.FCONTADOR", línea 6
```

Cláusula ACCESSIBLE BY

8. La nueva cláusula ACCESSIBLE BY en Oracle 12c permite ampliar la seguridad estándar. Ésta define una lista de objetos de base de datos (llamados descriptores de acceso) que pueden invocar a código PL/SQL.
9. Sintaxis:

→ ACCESSIBLE BY () →

Observaciones:

- Se puede utilizar dentro de las siguientes sentencias SQL:
 - CREATE FUNCTION
 - CREATE PACKAGE
 - CREATE PROCEDURE
 - CREATE TYPE
- Lista de descriptores de acceso que pueden invocar a código PL/SQL incluye objetos de base de datos como: TRIGGER, FUNCTION, PROCEDURE, PACKAGE y TYPE.
- Con esta cláusula se pueden especificar como descriptores de acceso objetos que pertenezcan a otro usuario añadiendo el nombre del esquema.

10. Ejemplos:

11. Se crea una función (*funcion_priv*) que solo puede ser invocada por un procedimiento llamado *procedimiento_pub* y por otra función (*funcion_pub*).

```
CREATE OR REPLACE FUNCTION funcion_priv (p_entrada NUMBER) RETURN NUMBER
ACCESSIBLE BY (PROCEDURE procedimiento_pub, FUNCTION funcion_pub)
IS
BEGIN
    RETURN p_entrada+1;
END;
/
```

12. Realiza una consulta utilizando una llamada a la función.

```
SQL> SELECT funcion_priv(10) FROM dual;
SELECT funcion_priv (10) FROM dual
      *
ERROR at line 1:
ORA-06552: PL/SQL: Statement ignored
ORA-06553: PLS-904: insufficient privilege to access object FUNCTION_PRIV
```

13. Crea la función que permite el acceso a la anterior función.

```
CREATE OR REPLACE FUNCTION funcion_pub (p_entrada NUMBER) RETURN NUMBER
IS
    v_retorno NUMBER;
BEGIN
    v_retorno := funcion_priv(p_entrada);
    RETURN v_retorno;
END;
/
```

14. Realiza una consulta utilizando una llamada a esta función.

```
SQL> SELECT funcion_pub(10) FROM dual;  
  
FUNCION_PUB(10)  
-----  
11
```

Privilegio INHERIT PRIVILEGES

Cuando un usuario ejecuta un procedimiento (o unidad de programa) que tiene establecidos los derechos del invocador, este código PL/SQL es ejecutado con los privilegios del usuario que invoca el programa.

Cuando el procedimiento se ejecuta, el propietario del procedimiento tiene temporalmente los privilegios de acceso del usuario que ejecuta el subprograma.

Si el propietario del procedimiento tiene menos privilegios que el usuario que realiza la ejecución, el creador del procedimiento podría utilizar para realizar las operaciones correspondientes los privilegios del usuario que lo invoca.

En versiones anteriores, el usuario que realiza la ejecución no tenía ningún control sobre quién podría tener este acceso a un procedimiento que se establece con los derechos del *invoker*.

A partir de Oracle 12c, un procedimiento *invoker's rights* sólo puede ejecutarse con los privilegios del usuario que lo ha invocado si el propietario tiene el permiso INHERIT PRIVILEGES del invocador o INHERIT ANY PRIVILEGES sobre todos los usuarios. Éste último privilegio es conveniente establecerlo solamente a usuarios de confianza.

Para conceder el privilegio se utiliza el comando GRANT con la siguiente sintaxis:

```
GRANT INHERIT PRIVILEGES ON USER usuario_invocador
  TO { propietario_subprograma; | PUBLIC };

GRANT INHERIT ANY PRIVILEGES TO propietario_subprograma;
```

Si se desea eliminar este privilegio usa el comando REVOKE:

```
REVOKE INHERIT PRIVILEGES ON usuario_invocador
  FROM { propietario_subprograma | PUBLIC };

REVOKE INHERIT ANY PRIVILEGES FROM propietario_subprograma;
```

Ejemplos:

15. El privilegio INHERIT PRIVILEGES se concede a todos los usuarios durante el proceso de creación.

```
SQL> CREATE USER alumna IDENTIFIED BY curso;

User created.

SQL> SELECT privilege, grantor, grantee FROM dba_tab_privs
  2  WHERE grantor='ALUMNA';
```

PRIVILEGE	GRANTOR	GRANTEE
INHERIT PRIVILEGES	ALUMNA	PUBLIC

16. Sin herencia el código PL/SQL establecido con los derechos del invocador fallará.

```
SQL> EXECUTE alumna.proc_lista_emp('IT_PROG')

BEGIN alumna.proc_lista_emp('IT_PROG'); END;
*
ERROR at line 1:
ORA-06598: privilegio INHERIT PRIVILEGES insuficiente
ORA-06512: en "ALUMNA.PROC_LISTA_EMP", línea 1
ORA-06512: en línea 1
```

17. Concede el privilegio INHERIT PRIVILEGES del usuario SYS (usuario que invoca al subprograma) al usuario ALUMNA (creador del procedimiento).

```
SQL> GRANT INHERIT PRIVILEGES ON USER sys TO alumna;
```

18. Comprueba mediante la vista del diccionario dba_tab_privs la concesión del privilegio anterior:

```
SQL> SELECT privilege, grantor, grantee FROM dba_tab_privs
2  WHERE grantee='ALUMNA';
```

PRIVILEGE	GRANTOR	GRANTEE
INHERIT PRIVILEGES	SYS	ALUMNA

19. La ejecución será correcta estableciendo el privilegio INHERIT PRIVILEGES del usuario correspondiente al creador del subprograma.

```
SQL> EXECUTE alumna.proc_lista_emp('IT_PROG')

select first_name from hr.employees where job_id='IT_PROG'
Alexander
Bruce
David
Valli
Diana
```

Concesión de roles a subprogramas PL/SQL

20. A partir de Oracle 12c, puedes otorgar roles individuales a los paquetes PL/SQL y a los subprogramas independientes (*standalone*). En lugar de una unidad de programa con privilegios del creador, se puede crear un código PL/SQL con los derechos del invocador y luego concederle el rol correspondiente.
21. La unidad de programa creada con los derechos del invocador se ejecuta entonces con los privilegios que tiene el usuario que lo invoca y los incluidos en el rol, pero sin ningún tipo de privilegio adicional que posea el usuario que define el programa.
22. Para realizar esto se utiliza el comando GRANT con la sintaxis:

```
GRANT nombre_rol TO {FUNCTION|PROCEDURE|PACKAGE} nombre_subprograma;
```

23. Ejemplos:

24. Se crea una unidad de programa PL/SQL. En nuestro caso una función que se ejecuta con los permisos del usuario que invoca a la función:

```
CREATE OR REPLACE FUNCTION f_calculo(p_entrada NUMBER) RETURN NUMBER
AUTHID CURRENT_USER
AS
    dato NUMBER;
BEGIN
    SELECT count(*)+p_entrada INTO dato
    FROM hr.departments;
    RETURN dato;
END f_calculo;
/
```

25. Se crea un rol y se le agrega el privilegio necesario para que el subprograma anterior puede ser ejecutado por cualquier usuario. En este caso el privilegio de consulta sobre la tabla departments del usuario HR.

```
CREATE ROLE rol_funcion;

GRANT SELECT ON hr.departments TO rol_funcion;
```

26. Se concede el rol anterior al subprograma creado.

```
GRANT rol_funcion TO FUNCTION hr.f_calculo;
```


Paquetes

Tabla de contenidos

Introducción	1
Crear o modificar un Paquete	1
Ventajas de los paquetes PL/SQL.....	4
Modularidad	4
Facilidad en el Diseño de la Aplicación	4
Ocultamiento de la Información	4
Funcionalidad Agregada	4
Mejora Ejecución	5
Sobrecarga de subprogramas en paquetes	5
Paquetes definidos por Oracle.....	6
STANDARD	6
DBMS_OUTPUT	6
UTL_FILE.....	7

Introducción



Un paquete es un objeto del esquema que agrupa lógicamente variables, constantes, tipos de datos y subprogramas PL/SQL. Los paquetes se dividen en:

- Especificación: es la zona de declaración de las variables, tipos, constantes, excepciones, cursores y subprogramas disponibles para ser usados.
- Cuerpo: zona en la que se implanta el código de los cursores y subprogramas definidos en la especificación, también puede contener otras declaraciones y otros subprogramas que no estén definidos en la especificación.

Crear o modificar un Paquete

Para crear un paquete se utiliza la sentencia `CREATE OR REPLACE PACKAGE nombre_package` y toda su definición para crear la especificación, y para crear el cuerpo se utiliza la sentencia `CREATE OR REPLACE PACKAGE BODY nombre_package` con la implementación del código:

```
CREATE [OR REPLACE] PACKAGE Nombre_paquete
  [AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [Definición_Tipo_Colección ...]
  [Definición_tipo_Registro ...]
  [Definición_Subtipos ...]
  [Declaración_Colección ...]
  [Declaración_constante ...]
  [Declaración_Excepción ...]
  [Declaración_Objeto ...]
  [Declaración_Registro ...]
  [Declaración_Variable ...]
  [Especificación_Cursor ...]
  [Especificación_Función... ]
  [Especificación_Procedimiento ...]
  [Especificación_Llamada ...]
  [PRAGMA RESTRICT_REFERENCES(Tipos_Comportamiento) ...]
END [Nombre_paquete];
/
```

Una vez creada la cabecera del paquete, posteriormente se crearía el cuerpo.

```
CREATE [OR REPLACE] PACKAGE BODY Nombre_paquete
{IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [Definición_Tipo_Colección ...]
  [Definición_tipo_Registro ...]
  [Definición_Subtipos ...]
  [Declaración_Colección ...]
```

```

[Declaración_constante ...]
[Declaracion_Excepción ...]
[Declaración_Objeto ...]
[Declaración_Registro ...]
[Declaración_Variable ...]
[Especificación_Cursor ...]
[Especificación_Función...]
[Especificación_Procedimiento ...]
[Especificación_Llamada ...]
[BEGIN
    Sentencias procedurales
[EXCEPTION
    Tratamiento de excepciones]]
END [Nombre_paquete];
/

```

La especificación contiene la parte pública del paquete la cual es visible desde otras aplicaciones. Los procedimientos tienen que ser declarados al final de la zona de especificación, excepto las PRAGMAS que hacen referencia a alguna función.

El cuerpo del paquete contiene la implementación de cada cursor y subprograma declarado en la parte de las especificaciones. Todos los que estén declarados en las especificaciones serán públicos, el resto serán privados y no podrán ser accedidos fuera del paquete.

Para concordar los procedimientos declarados en la zona de especificaciones y el cuerpo se hace una comparación carácter a carácter. La única excepción es el espacio en blanco. En caso que no coincidiera Oracle levantaría una excepción.

La invocación a un subprograma de un paquete desde otro paquete, procedimiento o función se puede realizar, siempre y cuando estos subprogramas sean públicos:

```

Nombre_paquete.nombre_procedimiento(parametros);

Variable:= Nombre_paquete.nombre_funcion(parametros);

IF Nombre_paquete.nombre_funcion(parámetros) < 10 THEN
    ...
END IF;

```

Un ejemplo sencillo de un paquete en el que se ve como se declaran los procedimientos, funciones y una variable en la especificación del paquete.

```

CREATE OR REPLACE PACKAGE nom_paquete
AS
    var_global_publica NUMBER;
    PROCEDURE prol_publico;
    PROCEDURE pro2_publico(parametro NUMBER);
    FUNCTION fun_publica RETURN NUMBER;
END nom_paquete;
/

```

Una vez creado el paquete, se continúa con la creación del cuerpo del paquete, en el cual estará la implementación de lo descrito en la especificación del paquete.

```

CREATE OR REPLACE PACKAGE BODY nom_paquete
AS
    var_global_privada NUMBER;
    -----
    FUNCTION fun_privada RETURN NUMBER
    IS
    BEGIN

```

```

        dbms_output.put_line('---fun_privada---');
        RETURN 0;
    END fun_privada;
-----
PROCEDURE pro1_publico
IS
    var_local    NUMBER;
BEGIN
    var_local:=fun_privada+1;--llamada a funcion privada
    var_global_publica:=11;
    var_global_privada:=101010;
    dbms_output.put_line('---pro1_publico---');
    dbms_output.put_line('var_global_publica: '||var_global_publica);
    dbms_output.put_line('var_global_privada: '||var_global_privada);
END pro1_publico;
-----
PROCEDURE pro2_publico(parametro NUMBER)
IS
    var_local    INTEGER;
BEGIN
    var_local:=fun_publica+3;--llamada a funcion publica
    var_global_publica:=33;
    var_global_privada:=303030;
    dbms_output.put_line('---pro2_publico---');
    dbms_output.put_line('var_global_publica: '||var_global_publica);
    dbms_output.put_line('var_global_privada: '||var_global_privada);
END pro2_publico;
-----
FUNCTION fun_publica RETURN NUMBER
IS
BEGIN
    dbms_output.put_line('---fun_publica---');
    RETURN 0;
END fun_publica;
-----
BEGIN
    var_global_publica:=0;
    var_global_privada:=1;
    dbms_output.put_line('---nom_paquete---');
    dbms_output.put_line('var_global_publica: '||var_global_publica);
    dbms_output.put_line('var_global_privada: '||var_global_privada);
END nom_paquete;
/

```

A continuación algunas llamadas a los subprogramas del paquete.

```

SQL> EXECUTE nom_paquete.pro1_publico;
---nom_paquete---
var_global_publica: 0
var_global_privada: 1
---fun_privada---
---pro1_publico---
var_global_publica: 11
var_global_privada: 101010

Procedimiento PL/SQL terminado correctamente.

SQL> EXECUTE nom_paquete.pro1_publico;
---fun_privada---
---pro1_publico---
var_global_publica: 11
var_global_privada: 101010

Procedimiento PL/SQL terminado correctamente.

SQL> EXECUTE nom_paquete.pro2_publico(1);
---fun_publica---

```

```
---pro2_publico---  
var_global_publica: 33  
var_global_privada: 303030  
  
Procedimiento PL/SQL terminado correctamente.  
  
SQL> SELECT nom_paquete.fun_publica FROM dual;  
  
FUN_PUBLICA  
-----  
0  
  
---fun_publica---
```

Algunas notas:

- La especificación de un paquete puede contener solo declaración de variables, tipos; esto no necesita de un cuerpo del paquete.
- En el cuerpo del paquete, primero se coloca lo privado, luego la implementación de lo público.
- Las funciones privadas no se pueden utilizar directamente en instrucciones SQL; se debe asignar su valor de retorno a una variable y utilizar ésta.
- El begin de un paquete es opcional y éste se ejecuta una sola vez por sesión.
- La zona de excepciones del paquete trata las excepciones originadas solo por instrucciones del begin del paquete.

Ventajas de los paquetes PL/SQL

Modularidad

Los paquetes pueden encapsular lógicamente tipos de datos y subprogramas en un modulo PL/SQL con nombre. Cada paquete es fácil de entender, y las interfaces con los paquetes son simples, claras y bien definidas, esto facilita el desarrollo de la aplicación.

Facilidad en el Diseño de la Aplicación

Cuando diseñamos una aplicación, todo lo que inicialmente se necesita es la información de la interfaz en la especificación del paquete. No se necesita definir completamente el cuerpo del paquete hasta que no se complete la definición de la aplicación.

Ocultamiento de la Información

En los paquetes, se pueden especificar tipos de datos y subprogramas para que sean públicos (visibles y accesibles) o privados (invisibles e inaccesibles). Por ejemplo, si tenemos un paquete que contiene cuatro subprogramas, tres públicos y uno privado. El paquete oculta la especificación del subprograma privado y solo implementa su código en el cuerpo del paquete.

Funcionalidad Agregada

Las variables son persistentes para la sesión. Es decir, su valor se mantiene para toda la sesión del usuario que ejecuta ese paquete. Si otro usuario invoca el paquete, las variables contendrán el valor que inicialice el paquete, no los valores modificados por otro usuario.

Mejora Ejecución

En la parte declarativa del cuerpo del paquete, opcionalmente, se puede inicializar las variables globales del cuerpo del paquete. Esta inicialización se ejecutará la primera vez que el paquete se coloque en memoria, es decir, la primera vez que un procedimiento del paquete sea invocado.

Si hay algún cambio en la implementación de una función de un paquete, Oracle no necesita recompilar las invocaciones a subprogramas porque no depende del cuerpo del paquete.

Sobrecarga de subprogramas en paquetes

PL/SQL permite dos o más subprogramas con el mismo nombre dentro del mismo paquete. Esta opción es usada cuando se necesita un subprograma igual que acepte parámetros que tienen diferentes tipos de datos.

```
CREATE OR REPLACE PACKAGE altas
AS
    procedure alta_emp
        (dni    PLS_INTEGER
        ,nombre VARCHAR2);
    procedure alta_emp
        (dni    VARCHAR2
        ,nombre VARCHAR2);
    procedure alta_emp
        (dni    VARCHAR2
        ,nombre VARCHAR2
        ,edad   PLS_INTEGER);
END altas;
/
```

```
CREATE OR REPLACE PACKAGE BODY altas
AS
    PROCEDURE alta_emp
        (dni    PLS_INTEGER
        ,nombre VARCHAR2)
    IS
    BEGIN
        dbms_output.put_line(rpad(dni,10,'*') || nombre);
    END alta_emp;
    -----
    PROCEDURE alta_emp
        (dni    VARCHAR2
        ,nombre VARCHAR2)
    IS
    BEGIN
        dbms_output.put_line(rpad(dni,13,'_') || nombre);
    END alta_emp;
    -----
    PROCEDURE alta_emp
        (dni    VARCHAR2
        ,nombre VARCHAR2
        ,edad   PLS_INTEGER)
    IS
    BEGIN
        dbms_output.put_line(rpad(dni,13,'-') || rpad(nombre,13,'-') || edad);
    END alta_emp;
    -----
END altas;
/
```

Al realizar una prueba el resultado será:

```
SQL> EXECUTE altas.alta_emp(12345,'pepe');
12345*****pepe

Procedimiento PL/SQL terminado correctamente.

SQL> EXECUTE altas.alta_emp('12345c','pepe');
12345c_____pepe

Procedimiento PL/SQL terminado correctamente.

SQL> EXECUTE altas.alta_emp('12345c','pepe',44);
12345c-----pepe-----44
```

El tercer procedimiento demuestra que además de distinto tipo de dato en los parámetros, un número distinto de parámetros también se puede aplicar.

Paquetes definidos por Oracle

Oracle cuenta con paquetes predefinidos, en ellos contiene procedimientos y/o funciones, que pueden ser utilizados por los usuarios.

Para utilizar los paquetes ofrecidos por Oracle:

```
Nombre_paquete.nombre_procedimiento_o_funcion(parametros);
```

Entre los paquetes más usados están los siguientes:

STANDARD

El paquete STANDARD define el entorno PL/SQL. El paquete declara globalmente todas las funciones, tipos, excepciones y subprogramas que PL/SQL puede utilizar. Las funciones pueden ser anuladas por definiciones propias de los paquetes. No obstante se pueden hacer referencia cualificándolas.

```
abs_diff := STANDARD.ABS(x - y);
```

DBMS_OUTPUT

El paquete muestra información desde procedimientos almacenados y triggers.

Los procedimientos son:

- PUT: Mueve el valor a un buffer. Se transforma a Varchar2 automáticamente. Los datos se guardan en el buffer hasta que no se ejecuta un NEW_LINE.
- PUT_LINE: Automáticamente ejecuta un NEW_LINE una vez que se ha introducido los valores.
- NEW_LINE: Muestra y vacía el contenido del buffer.
- GET_LINE: Recoge los valores del buffer.
- GET_LINES: Recoge una tabla de líneas.

Para poder utilizar el paquete DBMS_OUTPUT desde SQL/plus ha de activarse la opción SET SERVEROUTPUT ON.

```
BEGIN
```

```
DBMS_OUTPUT.PUT_LINE ('Esta línea aparecerá por la pantalla si se
activado');
DBMS_OUTPUT.PUT_LINE ('la opción SET SERVEROUTPUT ON en SQL/PLUS');
END;
/
```

Normalmente se utiliza PUT_LINE para depurar o seguir el control de un procedimiento PL/SQL.

UTL_FILE

Con el paquete UTL_FILE, programas PLSQL pueden leer y escribir en el sistema operativo ficheros de texto.

Los privilegios necesarios para acceder al fichero son específicos del sistema operativo.

En Oracle debe estar creado previamente el objeto directorio. Ej:

```
CREATE DIRECTORY DATOS AS 'C:\ORACLE\DATOS';
```

ALL_DIRECTORIES es una vista del diccionario de datos en la que obtenemos información sobre los directorios.

Sobre el directorio se debe otorgar permisos de lectura o escritura a los usuarios Oracle. El nombre del directorio es sensible a mayúsculas y minúsculas.

```
GRANT READ,WRITE ON DIRECTORY DATOS TO HR;
```

Un ejemplo de escritura de un archivo mediante un bloque anónimo.

```
DECLARE
archivo    UTL_FILE.FILE_TYPE;
linea      VARCHAR2(1200);
CURSOR cDept IS
    SELECT department_id,department_name
    FROM   Departments;
Reg cDept%ROWTYPE;
BEGIN
--abrir fichero para escritura
archivo := UTL_FILE.FOPEN('DATOS','fichero.txt','w');
FOR reg IN cDept LOOP
    linea := reg.department_id || ',' || reg.department_name;
    UTL_FILE.PUT_LINE(archivo,linea);
END LOOP;
--cerrar fichero
UTL_FILE.FCLOSE(archivo);
END;
/
```

Podemos describir el paquete (DESC) y ver todos los subprogramas que contiene.

Un ejemplo de lectura de un archivo del sistema operativo para luego insertar los datos en una tabla Oracle:

```
CREATE TABLE depart
(campo1    NUMBER
,campo2    VARCHAR2(30)
);
```

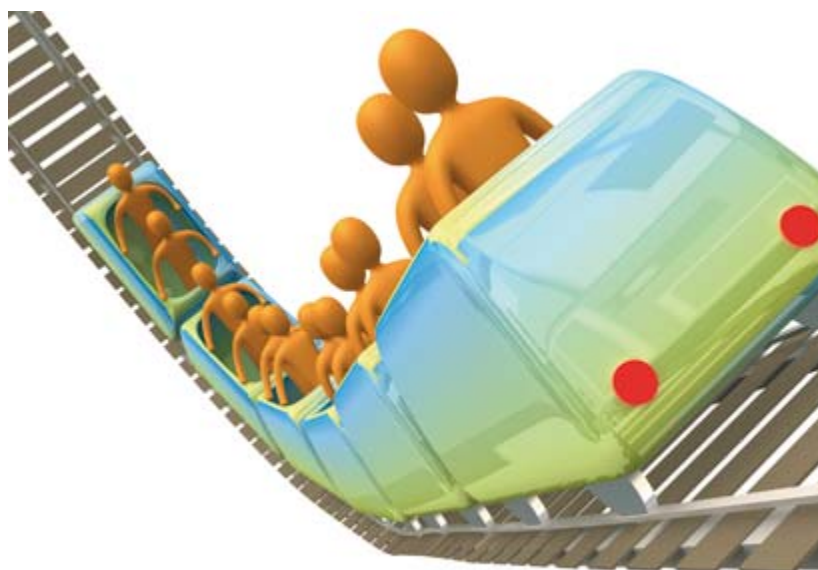
```
DECLARE
    V_Archivo          UTL_FILE.FILE_TYPE;
    V_linea            VARCHAR2(1200);
    v_campo1           Depart.CAMPO1%TYPE;
    v_campo2           Depart.CAMPO2%TYPE;
    v_delimitador      NUMBER;
BEGIN
    V_archivo := UTL_FILE.FOPEN('DATOS','fichero.txt','r');
    LOOP
        UTL_FILE.GET_LINE(v_archivo,v_linea);
        V_delimitador := instr(v_linea,',');
        v_campo1 := substr(v_linea,1,v_delimitador-1);
        v_campo2 := substr(v_linea,v_delimitador+1,length(v_linea));
        DBMS_OUTPUT.PUT_LINE(v_linea);
        DBMS_OUTPUT.PUT_LINE(v_campo1 || '*' || v_campo2);
        INSERT INTO Depart VALUES (v_campo1,v_campo2);
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Fin del archivo');
        UTL_FILE.FCLOSE(v_archivo);
    WHEN UTL_FILE.INVALID_PATH THEN
        DBMS_OUTPUT.PUT_LINE('Ruta o nombre no existe');
END;
/
```

Colecciones y Registros

Tabla de contenidos

Introducción	1
Colecciones	1
Tablas Indexadas versus Tablas Anidadas	2
Varrays versus Tablas Anidadas.....	2
Definición y Declaración de Colecciones	2
Tablas indexadas	2
Tablas anidadas.....	3
Varrays	3
Inicializar Colecciones	4
Inicializar una tabla anidada:	4
Inicializar un Varray:.....	4
Referenciar Colecciones	5
Asignación de Elementos a una Colección	5
Métodos para Colecciones	6
Comparación Completa de Colecciones	8
Algunos Ejemplos de Colecciones Multinivel	9
Mejorar el rendimiento mediante acciones BULK BINDING.	10
Registros	12

Introducción



Las colecciones y los registros son tipos compuestos que tienen componentes internos que pueden ser manipulados de forma individual, como los elementos de un array, registro, o una tabla.

Una colección es un grupo ordenado de elementos, todos del mismo tipo. Se trata de un concepto general que abarca las listas, matrices y otros tipos de datos utilizados en los algoritmos de programación clásica. Cada elemento es abordado por un subíndice único. Se puede acceder a cada elemento de una variable de tipo colección por su índice único, con la siguiente sintaxis: `nombre_variable(índice)`.

Un registro es un grupo de elementos de datos relacionados almacenados en los campos, cada uno con su propio nombre y tipo de datos. Usted puede pensar en un registro como una variable que puede contener una fila de una tabla, o algunas columnas de una fila de una tabla. Los campos corresponden a columnas de la tabla. Se puede acceder a cada campo de una variable de registro por su nombre, con la siguiente sintaxis:

`nombre_variable.nombre_campo`.

Para crear una variable de registro, bien se puede definir un Tipo de registro y luego crear una variable de ese tipo o usar `%ROWTYPE` o `%TYPE`.

Colecciones

Una colección es un grupo de elementos del mismo tipo. Cada elemento tiene un único subíndice que determina su posición en la colección. PL/SQL ofrece dos tipos de colecciones: `TABLE` y `VARRAY`; las tablas son anidadas o indexadas y tienen tamaño variable, en cambio los `VARRAY` tienen tamaño fijo.

Se pueden definir colecciones en un paquete y también se pueden pasar como parámetro.

Tablas Indexadas versus Tablas Anidadas

Las tablas indexadas y las tablas anidadas son similares. Tienen la misma estructura y sus elementos individuales son accedidos por el mismo camino. La diferencia principal es que la tabla anidada puede almacenar columnas en la base de datos y la indexada no puede.

Las tablas anidadas extienden la funcionalidad de las tablas indexadas, se pueden aplicar las sentencias SELECT, INSERT, UPDATE y DELETE sobre las tablas anidadas almacenadas en la base de datos pero no sobre las indexadas porque no se pueden almacenar en la base de datos.

Otra ventaja de las tablas anidadas es que se inicializa automáticamente a NULL, pero las indexadas no, están vacías. Por ello, se puede aplicar la condición IS NULL a las tablas anidadas pero no a las indexadas.

Varrays versus Tablas Anidadas

Las tablas anidadas difieren de los varrays en los siguientes temas:

Varrays tienen un tamaño máximo, las tablas anidadas no lo tienen.

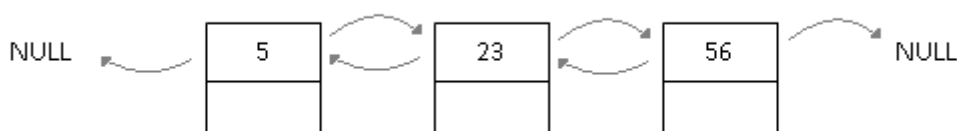
Desde una tabla anidada se puede borrar un elemento individual, desde un varray no.

Cuando se almacena en la base de datos, los varrays mantienen su orden, pero las tablas anidadas no.

Definición y Declaración de Colecciones

Para definir una colección, se debe definir el tipo de colección, TYPE TABLE o VARRAY.

Tablas indexadas



Permite localizar los elementos mediante números o cadenas aleatorias. Un valor o índice nuevo creará un nuevo elemento dentro de la tabla. Si se utiliza un valor ya existente se actualizará el valor al que apunta ese índice o valor. Es importante utilizar un valor que sea único puesto que la tabla estará indexada por ese valor.

No se pueden utilizar en sentencias INSERT o SELECT INTO. Al utilizar cadenas de caracteres para indexar la tabla, si se cambian los parámetros del grupo de caracteres durante la sesión en que la tabla indexada esté definida, se producirá un error al utilizar los métodos NEXT o PRIOR.

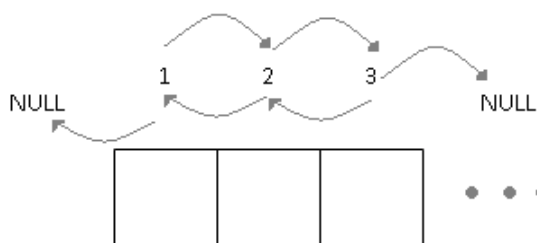
La sintaxis de definición de una tabla indexada es:

```
TYPE nombre_tipo IS TABLE OF elemento_tipo [NOT NULL]
INDEX BY [BINARY_INTEGER | PLS_INTEGER | VARCHAR2( precisión)];
```

El tipo_dato_indice puede ser numérico (BINARY_INTEGER o PLS_INTEGER) Puede ser un VARCHAR2 o cualquiera de sus subtipos VARCHAR, STRING, CHAR, etc.

Se debe especificar la longitud o precisión en los campos VARCHAR2 (Exceptuando LONG que es equivalente a VARCHAR2(32760). Los tipos RAW, LONG RAW, ROWID, CHAR, y CHARACTER no están permitidos como tipo de dato índice. Se puede utilizar tipos DATE o INTERVAL si se convierten a VARCHAR2 (Función TO_CHAR)

Tablas anidadas



Las tablas anidadas se pueden considerar como una tabla de la base de datos de una sola columna. Oracle guarda las tablas anidadas sin ningún orden en particular pero cuando se recupera la tabla anidada en PL/SQL se pueden acceder a los elementos que la forman mediante un índice.

Las tablas anidadas son de una sola dimensión pero se pueden crear tablas de varias dimensiones anidando tablas en tablas. Es decir, que cada elemento de una tabla anidada sea otra tabla anidada.

Las tablas anidadas se diferencian de las tablas normales en:

Las tablas normales tienen un límite superior mientras las tablas anidadas no tienen límite.

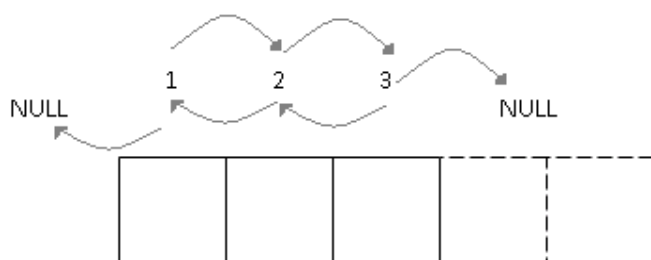
Las tablas normales tienen todos los elementos consecutivos mientras que en las tablas anidadas se pueden borrar elementos y los elementos pueden que no queden consecutivos.

La sintaxis de las tablas anidadas es:

```
TYPE nombre_tipo IS TABLE OF Tipo_elemento [NOT NULL];
```

Para las tablas anidadas declaradas en PL/SQL el Tipo_elemento puede ser cualquier tipo exceptuando REF CURSOR. Las tablas anidadas declaradas globalmente en SQL no pueden utilizarse como Tipo_elemento los tipos BINARY_INTEGER, PLS_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, POSITIVE, POSITIVEN, REF CURSOR, SIGNTYPE, STRING.

Varrays



Permiten asociar a un solo identificador una serie de elementos. Permite manipular toda la colección simultáneamente o acceder a los elementos que la forman. El número de

elementos es variable desde 0 (cuando se crea) hasta el máximo (especificado en el momento de la definición)

Su sintaxis es:

```
TYPE nombre_tipo IS {VARRAY | VARYING ARRAY} ( tamaño )
OF tipo_elemento [NOT NULL];
```

Tamaño es el número de elementos máximo que puede tener la VARRAY.

Inicializar Colecciones

Una tabla anidada o varray está vacía hasta que no se inicializa. La tabla anidada o varray propiamente tiene el valor nulo en sí misma, no sus elementos.

Para inicializar una tabla anidada o un varray se utiliza un constructor, la cual es una función definida por el sistema con el mismo nombre del tipo de la colección.

Inicializar una tabla anidada:

```
DECLARE
    TYPE Huespedes IS TABLE OF VARCHAR2(20);
    Hotel_Barcelona    Huespedes;
    Hotel_Madrid        Huespedes;
BEGIN
    Hotel_Barcelona := Huespedes ('Juan Pérez', 'Laura Gómez');
    Hotel_Madrid:= Huespedes();

END;
/
```

Al ser una tabla anidada se pueden insertar tantos elementos como se desee.

Inicializar un Varray:

```
DECLARE
    TYPE Puntuaciones IS VARRAY(10) OF NUMBER(2);
    Votos              Puntuaciones;
BEGIN
    Votos := Puntuaciones (1,2,3,4,5,6,7,8,10,12);
END;
/
```

El VARRAY no puede tener más elementos puesto que ha sido definido con 10 elementos.

Puede utilizarse NULL y se puede inicializar también en la declaración de la colección vacía, colocando el nombre de la colección sin argumentos:

```
DECLARE
    TYPE Huespedes IS TABLE OF VARCHAR2(20);
    -- inicialización a vacío
    Hotel_Barcelona    Huespedes:= Huespedes();
BEGIN
    IF Hotel_Barcelona IS NOT NULL THEN
        /* condición verdadera porque la colección es vacía pero no es nula*/
        Dbms_output.put_line('vacía pero no es nula');
    END IF;
END;
/
```

Referenciar Colecciones

Para referenciar un elemento de una colección se utiliza la siguiente sintaxis:

```
Nombre_colección(índice)
```

Índice tiene los siguientes rangos:

- Para tablas anidadas $1..2^{*31}$
- Para Varrays de 1 al límite en la definición del VARRAY
- Para tablas indexadas con una clave numérica $-2^{*31}..2^{*31}$
- Para tablas indexadas con una clave alfanumérica, el número de posibles valores alfanuméricos (delimitado por el conjunto de caracteres) y la longitud de la clave.

Asignación de Elementos a una Colección

Se puede asignar el valor de una expresión a un elemento específico en una colección usando la sintaxis:

```
Nombre_colección(suscriptor) := expresión;
```

En los siguientes casos se levantará una excepción:

- Si el índice es nulo o no es del tipo correcto se levantará la excepción `VALUE_ERROR`.
- Si el índice apunta un elemento que no está inicializado se levantará la excepción `SUBSCRIPT_BEYOND_COUNT`.
- Si la colección es nula (no se ha inicializado con ningún valor) se levantará la excepción `COLLECTION_IS_NULL`.

```
DECLARE
    TYPE ListNum IS TABLE OF INTEGER;
    nums ListNum;
BEGIN
    nums(1) := 10;          -- levanta la excepción COLLECTION_IS_NULL
    nums := ListNum (10,20,30);
    nums(1) := ASCII('A');
    nums(2) := 10 * nums(1);
    nums('B') := 15;       -- levanta la excepción VALUE_ERROR
    nums(4) := 40;         -- levanta la excepción SUBSCRIPT_BEYOND_COUNT
END;
/
```

```
DECLARE
    TYPE Cliente IS VARRAY(100) OF PLS_INTEGER;
    TYPE Persona IS VARRAY(100) OF PLS_INTEGER;
    grupo1 Cliente := Cliente (1003,1849,1034);
    grupo2 Cliente;
    grupo3 Persona;
BEGIN
    grupo2 := grupo1;
    grupo3 := grupo2;     -- error, son de distinto tipo de datos
                          --PLS-00382: el tipo de la expresión no es correcto
END;
/
```

Métodos para Colecciones

Los siguientes métodos utilizados sobre las colecciones ayudan a generalizar código, hacer uso más fácil de las colecciones y que las aplicaciones sean fácilmente mantenidas: EXIST, COUNT, LIMIT, FIRST, LAST, PRIOR, NEXT, EXTEND, TRIM Y DELETE.

Método	Tipo	Descripción
DELETE	Procedure	Elimina elementos de una tabla anidada o tabla indexada.
TRIM	Procedure	Elimina elementos desde el final de un varray o tabla anidada.
EXTEND	Procedure	Agrega elementos al final de un varray o tabla anidada.
EXISTS	Function	Retorna TRUE si y solo si existe el elemento especificado de la colección.
FIRST	Function	Retorna el primer índice de la colección.
LAST	Function	Retorna el último índice de la colección.
COUNT	Function	Retorna el número de elementos en la colección.
LIMIT	Function	Retorna el número máximo de elementos que la colección puede tener.
PRIOR	Function	Retorna el índice que precede al índice especificado.
NEXT	Function	Retorna el índice siguiente al índice especificado.

Un método de una colección es una construcción de una función o procedimiento que opera sobre colección y es invocada mediante la notación:

```
Nombre_colección.nombre_método[(parámetros)]
```

Estos métodos no pueden ser invocados desde sentencias SQL.

- El método `EXIST(n)` retorna TRUE si en la n-ésima posición existe un elemento; en otro caso FALSE.

```
IF tab.EXISTS(i) THEN
    tab(i) := nuevo_dato;
END IF;
```

- Este método `COUNT` retorna el número de elementos que contiene la colección.

```
IF empleados.COUNT = 25 THEN
    FOR i IN 1..empleados.COUNT LOOP
```

- El método **LIMIT**, Para las tablas anidadas que no tiene tamaño máximo, éste método retorna **NULL**. Para los **varray**, retorna el máximo número de elementos que puede contener, el cual se especifica en la declaración.

```
IF proyecto.LIMIT = 25 THEN

IF (proyecto.COUNT + 15) < proyecto.LIMIT THEN
```

- Los métodos **FIRST** y **LAST**, retornan el primer o último índice numérico de una colección. Si la colección está vacía, estos métodos retornan **NULL**. Si la colección tiene un solo elemento, estos métodos tendrán el mismo valor del índice.

```
IF alumnos.FIRST = alumnos.LAST THEN --un solo elemento

FOR i IN alumnos.FIRST..alumnos.LAST LOOP
```

- El método **Prior(n)** retorna el número índice que antecede al índice **n** de la colección. El método **Next(n)** retorna el número índice que sucede al índice **n** de la colección.

```
n := tab.PRIOR(tab.FIRST);    -- asigna NULL a n

i := tab.FIRST;              -- asigna el primer descriptor de la colección
WHILE i IS NOT NULL LOOP
    i := tab.NEXT(i);        /* asigna el siguiente descriptor de la colección */
END LOOP;
```

- El método **EXTEND** se utiliza para incrementar el tamaño de la colección. Este procedimiento tiene tres formas de utilizarse: **EXTEND**: que agrega un nulo al final de la colección, **EXTEND(n)**: agrega **n** nulos a la colección y **EXTEND(n,i)**: agrega **n** copias del elemento **i**-ésimo a la colección.

```
DECLARE
    TYPE lista IS TABLE OF VARCHAR2(10);
    Cursos lista;
BEGIN
    cursos := lista('DBA','Java'); --count es 2
    cursos.EXTEND;                -- agrega un lugar nulo al final de la tabla
    dbms_output.put_line(cursos.COUNT); --count es 3
    /* asigna un valor al último lugar (3)*/
    cursos(cursos.LAST):='Oracle';
    cursos.EXTEND(3);             -- agrega 3 nulos al final de la tabla
    cursos.EXTEND(2,1);           -- agrega 2 copias del elemento 1
    dbms_output.put_line(cursos.COUNT); --count es 8
END;
/
```

- El método **TRIM** tiene dos formas. **TRIM**: elimina un elemento del final de la colección. **TRIM(n)**: elimina **n** elementos desde el final de la colección.

```
DECLARE
    TYPE lista IS VARRAY(10) OF PLS_INTEGER;
    cursos lista;
BEGIN
    cursos := lista (11,22,33,44,55,66);
    dbms_output.put_line(cursos.COUNT);    --count es 6
    cursos.TRIM;                          -- count es igual a 5
    cursos.TRIM(2);                       --count es igual a 3
    dbms_output.put_line(cursos.COUNT);
END;
/
```

- El método DELETE tiene tres formas. DELETE: elimina todos los elementos de la colección. DELETE(n): elimina el n-ésimo elemento de una tabla anidada o indexada; si este elemento no existe, no hace nada. DELETE(m,n): elimina los elementos del rango m..n de una tabla anidada o indexada. En los varray, no se puede utilizar DELETE para eliminar elementos en forma individual.

```

cursos.DELETE(2);           -- elimina el elemento 2
cursos.DELETE(7,7);         -- elimina el elemento 7
cursos.DELETE(6,3);         -- no hace nada
cursos.DELETE(3,6);         -- elimina el elemento entre 3 y 6 incluidos
cursos.DELETE;              -- elimina todo los elementos

```

```

DECLARE
    TYPE Ventas_tipo    IS TABLE OF NUMBER
                        INDEX BY VARCHAR2(10);
    -- Definimos variable del tipo Ventas_tipo
    Ventas_nacionales   Ventas_tipo;
    i                   VARCHAR2(10);
BEGIN
    Ventas_nacionales('Madrid') := 200000;
    Ventas_nacionales('Barcelona') := 180000;
    Ventas_nacionales('Valencia') := 1000000;
    --contenido primer elemento (180000)
    dbms_output.put_line(ventas_nacionales(ventas_nacionales.FIRST));
    --contenido ultimo elemento (100000)
    dbms_output.put_line(ventas_nacionales(ventas_nacionales.LAST));
    --numero de elementos (3)
    dbms_output.put_line(ventas_nacionales.COUNT);
    --Modifica el elemento apuntado por Madrid
    Ventas_nacionales('Madrid') := 250000;
    --Elimina el elemento apuntado por Barcelona
    Ventas_nacionales.DELETE('Barcelona');
    --obtener el primer indice (Madrid)
    i:=ventas_nacionales.FIRST;
    --recorrer la colección
    WHILE (i IS NOT NULL) LOOP
        dbms_output.put_line(i||' '||ventas_nacionales(i));
        i:=ventas_nacionales.NEXT(i);
    END LOOP;
END;
/

```

Comparación Completa de Colecciones

Las colecciones no pueden ser comparadas por igualdad o desigualdad. el IF del ejemplo siguiente no es permitido.

```

DECLARE
    TYPE Cliente IS TABLE OF PLS_INTEGER;
    grupo1 Cliente := Cliente (11,22);
    grupo2 Cliente := Cliente (22,11);
    grupo3 Cliente := Cliente (11,2);
BEGIN
    IF grupo1 = grupo2 THEN
        dbms_output.put_line('grupo1 y grupo2 son iguales');
    END IF;
    IF grupo1 <> grupo3 THEN
        dbms_output.put_line('grupo1 y grupo3 no son iguales');
    END IF;
END;
/

```


Algunos Ejemplos de Colecciones Multinivel

Las colecciones pueden con elementos escalares o con elementos que son colecciones.

VARRAY Multinivel

```
DECLARE
    TYPE t1 IS VARRAY(10) OF PLS_INTEGER;
    TYPE nt1 IS VARRAY(10) OF t1;    -- varray multinive
    va t1 := t1(2,3,5);
        -- inicialización de varray multinivel
    nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
    i PLS_INTEGER;
    val t1;
BEGIN
    -- acceso al varray multinivel
    i := nva(2)(3);                -- i tiene valor 73
    dbms_output.put_line(i);
    nva.EXTEND;                    -- agrega un Nuevo elemento al varray nva
    nva(5) := t1(56, 32);
    nva(4) := t1(45,43,67,43345);  -- reemplaza el valor de la posición 4
    nva(4)(4) := 1;                -- reemplaza 43345 por 1
    nva(4).EXTEND;                 /* agrega un Nuevo elemento dentro del
varray de la posición 4 */
    nva(4)(5) := 89;              -- le da valor al Nuevo elemento.
END;
/
```

Tablas Multinivel

```
DECLARE
    TYPE tbl IS TABLE OF VARCHAR2(20);
    TYPE ntb1 IS TABLE OF tbl;    -- tabla de elementos varchar2
    TYPE tv1 IS VARRAY(10) OF PLS_INTEGER;
    TYPE ntb2 IS TABLE OF tv1;   -- tabla de elementos varray
    vtbl tbl := tbl('uno','dos');
    vntb1 ntb1 := ntb1(vtbl);
    vntb2 ntb2 := ntb2(tv1(3,5), tv1(8,7,3));
    f PLS_INTEGER;
    c PLS_INTEGER;
    --mostrar los datos
    procedure mostrar
    is
    begin
        --inicio mostrar informacion
        dbms_output.put_line('#####vntb1#####');
        f:=vntb1.first;
        while (f is not null) loop
            dbms_output.put_line(lpad('fila: '||f,8,chr(9)));
            c:=vntb1(f).first;
            while (c is not null) loop
                dbms_output.put(rpad(vntb1(f)(c),5));
                c:=vntb1(f).next(c);
            end loop;
            dbms_output.new_line;
            f:=vntb1.next(f);
        end loop;
        dbms_output.put_line('_____vntb2_____');
        f:=vntb2.first;
        while (f is not null) loop
            dbms_output.put_line(lpad('fila: '||f,8,chr(9)));
            c:=vntb2(f).first;
            while (c is not null) loop
                dbms_output.put(rpad(vntb2(f)(c),5));
                c:=vntb2(f).next(c);
            end loop;
            dbms_output.new_line;
            f:=vntb2.next(f);
        end loop;
    end;
```

```

        end loop;
        --fin mostrar informacion
    end mostrar;
BEGIN
    mostrar;--mostrar datos de las tablas
    vntbl.EXTEND;
    -- copia el primer elemento de vntbl
    vntbl(2) := vntbl(1);
    -- copia el primer elemento dos veces al final
    vntbl.EXTEND(2,1);
    -- se cambia el valor de al posicion f3c2
    vntbl(3)(2):='seis';
    -- elimina la primera cadena de la primera tabla lf
    vntbl.DELETE(1);
    -- elimina 2 ultimos elementos de la segunda tabla (7,3)
    vntb2(2).TRIM(2);
    mostrar;--mostrar datos de las tablas
END;
/

```

Tablas Indexadas Multinivel

```

DECLARE
    TYPE tbl IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
    /* La siguiente es una tabla indexada y sus elementos
       son otra tabla indexada */
    TYPE ntbl IS TABLE OF tbl INDEX BY PLS_INTEGER;
    TYPE val IS VARRAY(10) OF VARCHAR2(20);
    -- la siguiente es una tabla indexada y sus elementos son varray
    TYPE ntb2 IS TABLE OF val INDEX BY PLS_INTEGER;
    array1          val := val('hola', 'mundo');
    indexindex      ntbl;
    indexarray      ntb2;
    index1          tbl;
    index2          tbl; -- tabla nula
BEGIN
    index1(1) := 34;
    index1(2) := 46456;
    index1(456) := 343;
    indexindex(23) := index1;
    indexindex(45)(2) := 78;
    indexindex(35) := index2;
    indexindex(35)(2) := 78;
    indexarray(34) := val(33, 456, 656, 343);
    indexarray(44) := array1;
END;
/

```

Mejorar el rendimiento mediante acciones BULK BINDING.

El traspaso de información de las variables de un PL/SQL a SQL se llama BIND. Para cada instrucción SQL se realiza este traspaso de información. Se puede traspasar toda la información de una colección en una sola operación.

Para utilizar esta técnica en las sentencias INSERT, DELETE y UPDATE se engloban las sentencias SQL en un bloque FORALL

Para las sentencias SELECT se utiliza la cláusula BULK COLLECT anteponiéndola a la cláusula INTO.

```

DECLARE
    TYPE Tipo_Tabla_Codigos
        IS TABLE OF employees.employee_id%TYPE;
    Tabla_cod        Tipo_Tabla_codigos;
BEGIN

```

```

SELECT employee_id BULK COLLECT INTO Tabla_cod
FROM employees
WHERE salary<5000;

/* Se leen todos los registros que cumplan esa condición de golpe.
AL ser Tablas anidadas no tienen límites de elementos */
FORALL i IN Tabla_cod.FIRST .. Tabla_cod.LAST
    UPDATE employees
    SET salary=salary*1.07
    WHERE employee_id = Tabla_cod(i);
/* Se actualiza todos los registros cuyos códigos de empleado
estén en la tabla anidada Tabla_cod.
Todos los UPDATES se realizan en una sola operación. */

END;
/

```

Si en la instrucción FORALL se especifica únicamente una parte de la colección sólo esa parte será tratada mediante el BULK BINDING.

Cada instrucción SQL dentro de una cláusula FORALL lleva implícito un SAVEPOINT. Si se produce un error durante la ejecución del FORALL se realizará un ROLLBACK de todas las operaciones. Sin embargo si la excepción es tratada se produce un ROLLBACK hasta el último SAVEPOINT realizado.

Ejemplo: La Quinta vez que se ejecuta la sentencia INSERT falla, el departamento 10 ya existe.

```

DECLARE
    TYPE tipotabcod IS TABLE
        OF departments.department_id%TYPE;
    tabla_cod tipotabcod;
    TYPE tipotabnom IS TABLE
        OF departments.department_name%TYPE;
    tabla_nom tipotabnom;
BEGIN
    tabla_cod:=tipotabcod(6,7,8,9,10);
    tabla_nom:=tipotabnom('Madrid','Barcelona','Valencia','Alicante','Bilbao')
;
    COMMIT;
    FORALL Indice IN Tabla_cod.FIRST .. Tabla_cod.LAST
        INSERT INTO departments (department_id,department_name)
        VALUES (tabla_cod(indice),tabla_nom(indice));
END;
/

```

Al no tratarse el error se realizará un ROLLBACK hasta el último punto transaccional (en este caso el COMMIT)

```

DECLARE
    TYPE tipotabcod IS TABLE
        OF departments.department_id%TYPE;
    tabla_cod tipotabcod;
    TYPE tipotabnom IS TABLE
        OF departments.department_name%TYPE;
    tabla_nom tipotabnom;
BEGIN
    tabla_cod:=tipotabcod(6,7,8,9,10);
    tabla_nom:=tipotabnom('Madrid','Barcelona','Valencia','Alicante','Bilbao')
;
    COMMIT;
    FORALL Indice IN Tabla_cod.FIRST .. Tabla_cod.LAST
        INSERT INTO departments (department_id,department_name)

```

```

VALUES (tabla_cod(indice),tabla_nom(indice));
EXCEPTION
  WHEN dup_val_on_index THEN
    COMMIT;
END;
/

```

Al controlarse el error (EXCEPTION) se hará rollback la quinta sentencia SQL, con lo que las cuatro anteriores se validarán ya que hay un commit en la zona de excepciones.

Registros

Un registro es un grupo de campos; cada uno de ellos con su tipo de datos y su nombre. El atributo %ROWTYPE permite crear un registro que representa una fila o la colección de columnas de una tabla de la base de datos.

El tipo de datos RECORD crea un registro con los tipos de datos para cada campo mientras %ROWTYPE toma los tipos de datos de la base de datos.

La sintaxis para la declaración de un RECORD es:

```

TYPE nombre_tipo IS RECORD (Declaración_campo[,Declaración_campo]...);

```

Donde Declaración_campo es

```

Nombre_campo Tipo_datos_campo [[NOT NULL] {:= | DEFAULT} Expresión]

```

El único tipo_datos_campo que no se puede utilizar es REF CURSOR. Se puede utilizar %TYPE y %ROWTYPE para especificar el tipo de dato del campo.

```

DECLARE
  TYPE t_fecha IS RECORD
    (dia          NUMBER(2)
    ,mes          NUMBER(2)
    ,anyo         NUMBER(2)
    );
  TYPE cabecera IS RECORD
    (employee_id employees.employee_id%TYPE
    ,full_name    VARCHAR2(50)
    ,fecha       t_fecha
    );
  v_fecha        t_fecha;
  reg            cabecera;
BEGIN
  v_fecha.dia:=12;
  v_fecha.mes:=06;
  v_fecha.anyo:=87;

  SELECT 12,06,87 INTO v_fecha FROM dual;

  SELECT 100,'nombre apellido'
  INTO   reg.employee_id,reg.full_name
  FROM   employees
  WHERE  employee_id=100;

  reg.fecha:=v_fecha;

  reg.fecha.mes:=10;
END;
/

```

La asignación o inicialización de valores en los registros se puede hacer en el DECLARE. También se pueden añadir limitaciones.

```
DECLARE
  TYPE cabecera IS RECORD
    (employee_id employees.employee_id%TYPE NOT NULL :=100
    ,full_name      VARCHAR2(50) DEFAULT 'no name'
    ,fecha          DATE
    );
BEGIN
  NULL;
END;
/
```

Para referenciar un campo dentro de un registro se utiliza la siguiente notación:

```
Registro.campo
```

Cuando se llama a una función que devuelve un tipo RECORD (en la definición de la función se especifica que devuelve un tipo RECORD) la notación para referirse a un campo será:

```
Nombre_función (Lista_parámetros).Nombre_campo
```

```
DECLARE
  TYPE Cita IS RECORD
    (Doctor      PLS_INTEGER
    ,Consulta    PLS_INTEGER
    ,Dia         PLS_INTEGER
    ,Mes         PLS_INTEGER
    ,ANYO       PLS_INTEGER
    );
  Consulta_dada PLS_INTEGER;
  FUNCTION Busca_cita(Codigo_usuario PLS_INTEGER)
    RETURN Cita
  IS
    Cita_devuelta Cita;
  BEGIN
    Cita_devuelta.consulta:=69;
    RETURN Cita_devuelta;
  END;
BEGIN
  Consulta_dada := Busca_cita(1700).Consulta;
  /*Asignamos a consulta_dada el valor que nos ha devuelto la función
  Busca_Cita que retorna un tipo RECORD y el campo Consulta. */
  dbms_output.put_line('consulta_dada = '||consulta_dada);
END;
/
```

Las sentencias SQL: INSERT, UPDATE y DELETE tienen una clausula opcional RETURNING, ésta puede retornar la fila afectada en una variable PL/SQL RECORD.

```
DECLARE
  TYPE tipo_rec IS RECORD
    (salario      employees.salary%TYPE
    ,nombre       employees.first_name%TYPE
    );
  REG  tipo_rec;
BEGIN
  UPDATE employees
  SET salary = salary*1.03
  WHERE employee_id=150
```

```
RETURNING salary,first_name INTO REG;

dbms_output.put_line(reg.nombre||' ahora gana '||reg.salario);
END;
/
```

Los registros creados con %ROWTYPE, al tener los mismos atributos y orden, que las filas de las tablas de la base de datos son útiles a la hora de insertar y actualizar las tablas de los que han sido definidos.

```
DECLARE
    REG    departments%ROWTYPE;
BEGIN
    reg.department_id:=300;
    reg.department_name:='Marketing';
    reg.manager_id:=100;
    reg.location_id:=1700;

    INSERT INTO departments
    VALUES REG;

    reg.department_id:=300;
    reg.department_name:='IT';

    UPDATE departments
    SET ROW = REG
    WHERE department_id=300;
END;
/
```

- ROW tiene que aparecer siempre al lado izquierdo de la igualdad del SET. No se puede utilizar ROW en subconsultas.
- Si en una sentencia INSERT se utiliza una variable tipo RECORD en la cláusula VALUES, no puede aparecer ninguna otra variable.
- En una sentencia UPDATE sólo puede aparecer una cláusula SET si se utiliza ROW.

10

Disparadores

Tabla de contenidos

Crear un disparador.....	2
Disparador DML simple.	3
Funciones Booleanas	5
Pseudo Registros :NEW y :OLD	6
Trigger INSTEAD OF.....	9
Disparadores DML Compuestos.....	11
Restricciones de los disparadores compuestos.....	12
Disparadores No DML.....	13
Evento DDL.....	13
Evento de base de datos.....	15

Introducción



Los triggers (o disparadores) son procedimientos almacenados que se ejecutan o disparan en el momento que se produce un evento.

Existen disparadores para las sentencias INSERT, UPDATE y DELETE sobre tablas o vistas; y a partir de Oracle8i para eventos en la base de datos y en los esquemas, llamados disparadores de sistema (system Triggers).

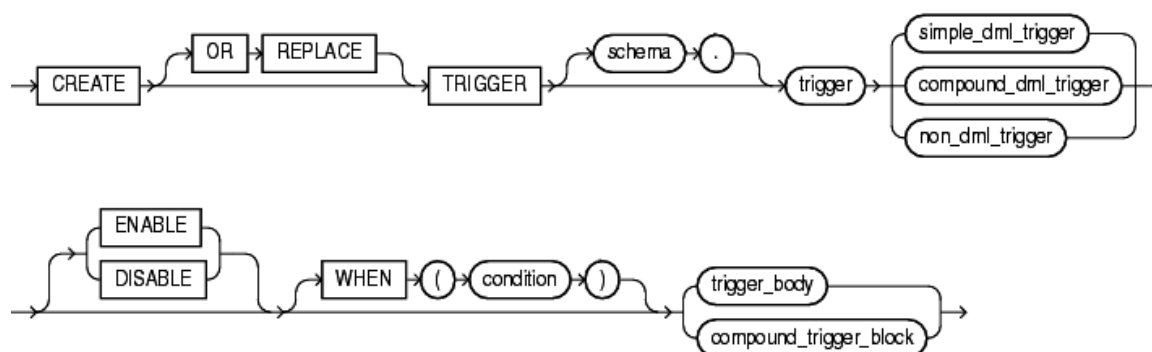
Los disparadores o triggers:

- Se utilizan para asegurar que cuando acontece una acción, se ejecutan una serie de acciones relacionadas.
- No hay que definir disparadores que ejecuten acciones que pueden ser implantadas mediante Oracle. Ejemplo: Comprobar que existe una fila en otra tabla mediante un trigger si se puede realizar esta comparación mediante una clave ajena (Foreign Key).
- Limitar el tamaño de un trigger o un disparador. En caso que la codificación de un disparador fuera excesiva es aconsejable generar un procedimiento almacenado e invocarlo desde el trigger. No puede tener un tamaño superior a 32K.
- Se utilizan para acciones globales sin importar el usuario o la aplicación que lo provoca.
- No crear disparadores recursivos. Ejemplo: Si se dispara un trigger al actualizar la tabla A, no se puede codificar que actualiza el disparador la tabla A. Existe el límite de 50 acciones recursivas.
- Utilizar juiciosamente los disparadores de DATABASE (Base de datos) ya que se disparan por cada usuario y cada vez que se produce el evento.

Crear un disparador

De una manera general podemos clasificar a los disparadores:

- Disparadores DML simples
 - Sentencia
 - Fila
- Disparadores DML compuestos
- Disparadores no DML



CREATE

Especifica la creación de un objeto.

OR REPLACE

Se especifica para reemplazar el objeto, del tipo especificado en caso de existir.

Schema

El esquema o usuario donde se creará el trigger. Si no se especifica será el propio.

trigger

El nombre del disparador. Si el trigger contiene errores de compilado se creará igualmente pero fallará en tiempo de ejecución. Si falla el disparador se inhabilita y tiene que volver a ser creado sin errores.

DISABLE

En Oracle 12g se añadió la opción de especificar la creación de un disparador en estado desactivado.

ENABLE

En Oracle12g se añadió esta opción como opuesta a la de crear el disparador en estado desactivado. Es la opción por defecto.

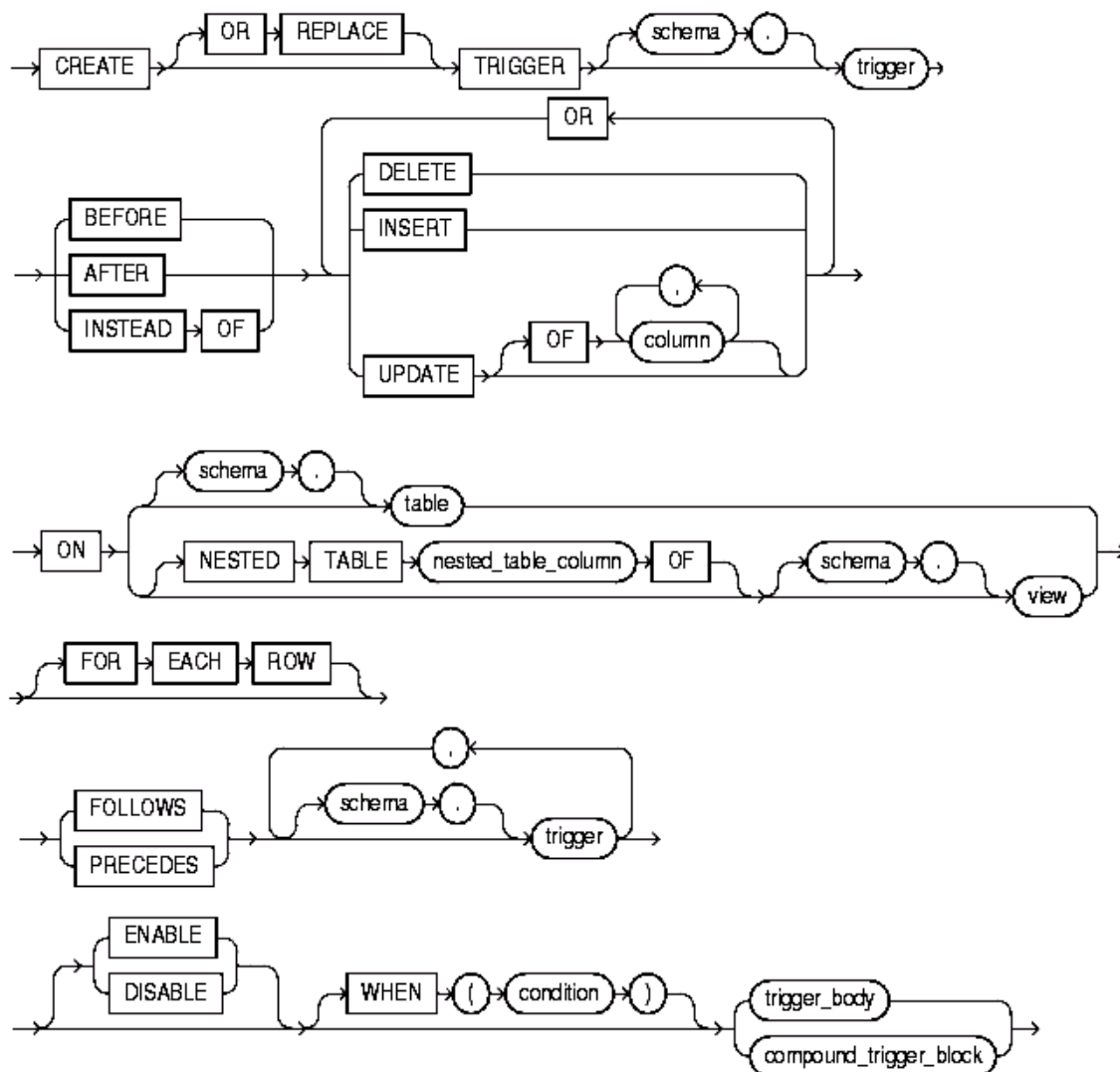
WHEN

Especifica una condición SQL que se evalúa para cada fila que afecta la sentencia disparadora. El código del trigger se ejecutará solo en caso de que la condición será TRUE.

- En un disparador simple DML se debe especificar FOR EACH ROW y no puede especificarse INSTEAD OF.
- No se puede incluir subconsultas o una expresión PLSQL (como una llamada a una función).
- No se puede especificar esta cláusula para un disparador STARTUP, SHUTDOWN, o DB_ROLE_CHANGE.
- Los registros NEW y OLD no llevan los dos puntos.

Disparador DML simple.

La sintaxis es:



BEFORE (Antes)

Se disparará antes de realizar las modificaciones. No se puede especificar BEFORE si es sobre una vista. Se puede modificar el valor :NEW, pero no el valor :OLD.

AFTER (Después)

Se disparará después de realizar las modificaciones. No se puede especificar AFTER si es sobre una vista. Se puede modificar el valor :OLD, pero no el valor :NEW.

Cuando se crea una vista materializada Oracle genera los triggers AFTER para cada INSERT, UPDATE y DELETE que modifican el log de la vista materializada.

INSTEAD OF

Oracle ejecuta el trigger en vez del evento que provoca el trigger. Se utiliza para poder actualizar vistas que no son actualizables. INSTEAD OF no se puede especificar en tablas. Se especifica únicamente en vistas. Se pueden leer los valores :NEW y :OLD, pero no se pueden modificar.

DELETE

Se disparará si se borra una fila de una tabla o de una tabla anidada.

INSERT

Se disparará si se inserta una fila de una tabla o en una tabla anidada.

UPDATE

Se disparará si se modifica una columna de una tabla o de una tabla anidada que esté especificada detrás de la cláusula OF. Si no se especifica OF, el trigger se disparará si se modifica cualquier columna.

- Se puede indicar un objeto, varray o una columna REF después de la cláusula OF pero no se podrán cambiar estos valores en el cuerpo del trigger.
- No se puede indicar un tipo LOB o una tabla anidada detrás del OF.

ON Tabla | Vista

No se puede crear triggers sobre tablas del esquema SYS.

FOR EACH ROW

Se especifica solo si es sobre una tabla. Indica que el disparador se va a ejecutar **para cada fila afectada** por la sentencia que lo ha provocado. Disparador a nivel de fila.

FOLLOWS

Se añadió en Oracle 12g para especificar el orden en el que se ejecutaran los disparadores. El disparador se ejecute *siguiendo a*.

PRECEDES

Se añadió en Oracle 12g para especificar que el disparador se ejecute *precediendo a*.

```
CREATE OR REPLACE TRIGGER tri1
  BEFORE INSERT ON departments
BEGIN
  Dbms_output.put_line('Se ha ejecutado el disparador tri1.');
```

Una vez creado el disparador tri1 de prueba, realizamos algunos *insert* para probar.

```
SQL> INSERT INTO departments(department_id,department_name)
2 VALUES(300,'Ventas');
Se ha ejecutado el disparador tri1.

1 fila creada.

SQL> INSERT INTO departments(department_id,department_name)
2 VALUES(300,'Ventas');
Se ha ejecutado el disparador tri1.
INSERT INTO departments(department_id,department_name)
*
ERROR en línea 1:
ORA-00001: restricción única (HR.DEPT_ID_PK) violada
```

Se realice el insert o no; el disparador se llega a ejecutar, ya que se trata de un disparador BEFORE.

Recordar activar serveroutput para visualizar la salida DBMS_OUTPUT

Funciones Booleanas

Estas funciones se utilizan cuando el evento de un disparador es compuesto, es decir, queremos que el disparador se active ante diferentes operaciones DML pero no queremos que haga lo mismo para cualquiera de los eventos activadores. Con lo visto hasta ahora, la única solución sería diseñar un disparador para cada una de las acciones DML.

Para diferenciar dentro del bloque PL/SQL cuál de los posibles sucesos es el que ha activado el disparador, se pueden utilizar los predicados condicionales o funciones booleanas: INSERTING, DELETING Y UPDATING.

```
CREATE OR REPLACE TRIGGER tri2
  AFTER INSERT OR DELETE OR UPDATE ON departments
BEGIN
  dbms_output.put_line('Se ha ejecutado el disparador tri2.');
```

```

                dbms_output.put_line('Actualizado el manager_id.');
```

WHEN UPDATING THEN

```

                dbms_output.put_line('Se ha actualizado.');
```

END CASE;

```

END tri2;
/
```

Al realizar unas pruebas:

```

SQL> INSERT INTO departments(department_id,department_name)
      2 VALUES(330,'Ventas');
Se ha ejecutado el disparador tri2.
Se ha insertado.

1 fila creada.

SQL> INSERT INTO departments(department_id,department_name)
      2 VALUES(330,'Ventas');
insert into departments(department_id,department_name)
*
ERROR en línea 1:
ORA-00001: restricción única (HR.DEPT_ID_PK) violada

SQL> DELETE departments WHERE location_id IS NULL;
Se ha ejecutado el disparador tri2.
Se ha eliminado.

2 filas suprimidas.

SQL> UPDATE departments SET manager_id=100 WHERE location_id=1700;
Se ha ejecutado el disparador tri2.
Actualizado el manager_id.

21 filas actualizadas.
```

Recordar que es un disparador a nivel de sentencia, se ejecuta una sola vez independientemente del número de filas afectadas.

El disparador al ser AFTER se ejecuta después de realizar la sentencia que lo provocó; si la sentencia falla, el disparador no se ejecuta.

Pseudo Registros :NEW y :OLD

Los disparadores pueden acceder a los valores anteriores y posteriores de una fila modificada.

Nótese que los registros :OLD y :NEW deben ser precedidos por los dos puntos para que puedan ser accedidas desde dentro del bloque PL/SQL.

Si la acción que se realiza es un INSERT, solo podrá ser visible el registro :NEW. Si la acción que se realiza es un UPDATE, podrán ser visibles los registros :OLD y :NEW. Si la acción que se realiza es un DELETE, solo podrá ser visible el registro :OLD.

Sentencia Disparadora	:OLD	:NEW
INSERT	NULL	Valor post INSERT
DELETE	Valor pre DELETE	NULL
UPDATE	Valor pre UPDATE	Valor post UPDATE

Cuando se utilicen en la clausula WHEN deben ir sin los dos puntos.

Para poder utilizar :NEW y :OLD el disparador tiene que ser a nivel de fila.

Las restricciones para los pseudo registros son:

- No se puede modificar los valores para :OLD (excepción ORA-04085)
- Si la sentencia disparadora es DELETE el disparador no puede cambiar los valores a los campos :NEW. (excepción ORA-04084)
- Un trigger AFTER no puede cambiar los valores a los campos a :NEW, porque la sentencia en cuestión se ejecuta antes del trigger. (excepción ORA-04084)
- No puede aparecer una operación a nivel de registro. (ejm: :NEW:=NULL)
- Un pseudo registro no puede ser un parámetro del subprograma actual. (Un campo del pseudo registro si, :NEW.campo)

```
CREATE OR REPLACE TRIGGER tri3
  BEFORE UPDATE ON departments
  FOR EACH ROW
BEGIN
  dbms_output.put_line('Se ha ejecutado el disparador tri3.');
```

```
  dbms_output.put_line('old.department_id = '||:old.department_id);
  dbms_output.put_line('old.department_name = '||:old.department_name);
  dbms_output.put_line('old.manager_id = '||:old.manager_id);
  dbms_output.put_line('old.location_id = '||:old.location_id);
  dbms_output.put_line('-----');
  dbms_output.put_line('new.department_id = '||:new.department_id);
  dbms_output.put_line('new.department_name = '||:new.department_name);
  dbms_output.put_line('new.manager_id = '||:new.manager_id);
  dbms_output.put_line('new.location_id = '||:new.location_id);
END tri3;
/
```

Si realizamos un *UPDATE* para probar.

```
SQL> update departments
  2  set manager_id=100, location_id=1700
  3  where department_id=20 or department_id=10;
Se ha ejecutado el disparador tri3.
:old.department_id = 10
:old.department_name = Administration
:old.manager_id = 200
:old.location_id = 1700
-----
:new.department_id = 10
:new.department_name = Administration
:new.manager_id = 100
:new.location_id = 1700
Se ha ejecutado el disparador tri3.
:old.department_id = 20
:old.department_name = Marketing
:old.manager_id = 201
:old.location_id = 1800
-----
:new.department_id = 20
:new.department_name = Marketing
:new.manager_id = 100
:new.location_id = 1700

2 filas actualizadas.
```

El ejemplo anterior se dispararía siempre que se modifique la tabla departments, si lo que deseamos es que se dispare cuando se modifique el campo manager_id o location_id solamente, lo tendríamos que hacer de la siguiente forma:

```
CREATE OR REPLACE TRIGGER tri3
  BEFORE UPDATE OF manager_id,location_id ON departments
  FOR EACH ROW
BEGIN
  dbms_output.put_line('*****Se ha ejecutado el disparador tri3.*****');
  dbms_output.put_line(':old.department_id = '||:old.department_id);
  dbms_output.put_line(':old.department_name = '||:old.department_name);
  dbms_output.put_line(':old.manager_id = '||:old.manager_id);
  dbms_output.put_line(':old.location_id = '||:old.location_id);
  dbms_output.put_line('-----');
  dbms_output.put_line(':new.department_id = '||:new.department_id);
  dbms_output.put_line(':new.department_name = '||:new.department_name);
  dbms_output.put_line(':new.manager_id = '||:new.manager_id);
  dbms_output.put_line(':new.location_id = '||:new.location_id);
END tri3;
/
```

El especificar campos es solo para UPDATE. Si además el disparador se especifica un INSERT o DELETE. Estas últimas sentencias insertan o eliminan filas, no campos; es decir, se ejecutará el disparador.

En el siguiente ejemplo:

- Al insertar un empleado se ha de controlar que el salario del jefe no sea inferior al del nuevo empleado.
- Solo para los nuevos empleados de los departamento 10, 20, 30, 40, 90.
- En cualquier otro caso se ha de permitir el insert.

```
CREATE OR REPLACE TRIGGER tri4
  BEFORE INSERT ON employees
  FOR EACH ROW
  WHEN (NEW.manager_id IS NOT NULL
        AND NEW.salary IS NOT NULL
        AND NEW.department_id IN (10,20,30,40,90))
  --controla que el salario del jefe de UN NUEVO empleado
  --no sea menor que el salario del empleado
DECLARE
  v_sal_jefe employees.salary%TYPE;--almacena salario del jefe
BEGIN
  --salario del empleado cuyo id de empleado es igual
  --al id del jefe del nuevo empleado.
  SELECT salary
  INTO v_sal_jefe
  FROM employees
  WHERE employee_id=:new.manager_id;
  IF v_sal_jefe<:NEW.salary THEN --salario del jefe menor que el del
empleado?
    raise_application_error(-20606
    ,'No se permite que el salario del jefe sea menor que el salario del
empleado.');
```

Aunque el disparador hubiera sido AFTER, al ser cierta la condición del IF y por lo tanto se realizara el RAISE_APPLICATION_ERROR y no tratar la excepción, el INSERT se hubiera deshecho. Pero, es de mala programación dejar que se realice ese trabajo en vano.

Al necesitar declarar variables, se delimita la sección con la palabra DECLARE en vez de IS o AS.

Trigger INSTEAD OF

- Un disparador INSTEAD OF es un disparador DML creado en una vista no editable, o en una columna de tipo tabla anidada de una vista no editable.
- La base de datos realiza las instrucciones del disparador INSTEAD OF en vez de ejecutar la instrucción DML que origino el disparo.
- No puede ser condicional.
- Es la única manera de actualizar una vista que no es de por sí actualizable.
- Siempre es a nivel de fila.
- Puede leer valores de los registros :NEW y :OLD, pero no puede modificarlos.

Ej. Supongamos para un país, tenemos una vista en la que tenemos información de los departamentos y la localidad.

```
CREATE VIEW departs AS
  SELECT
    department_name
    ,street_address
    ,postal_code
    ,city
    ,state_province
  FROM
    departments
  JOIN locations USING(location_id)
  WHERE
    country_id = 'US';
```

Se desea que realicen INSERT sobre la vista para dar de alta un departamento. Si la localidad especificada donde se da de alta el departamento no existe también hay que dar de alta la localidad.

Para ello creamos el siguiente disparador INSTEAD OF

```
CREATE OR REPLACE TRIGGER tri5
  INSTEAD OF INSERT ON departs
  --PRE: el departamento no existe
  --POS: se da de alta al departamento en la ciudad indicada
  --POS: si la localidad no existe, se da de alta la localidad
  DECLARE
    v_location_id locations.location_id%TYPE;
  BEGIN
    BEGIN
      SELECT      location_id
      INTO        v_location_id
      FROM        locations
      WHERE       city=:NEW.city
      AND         postal_code=:NEW.postal_code
      AND         street_address=:NEW.street_address;
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        --alta localidad
        v_location_id:=locations_seq.NEXTVAL;
```

```

        INSERT INTO locations
        (location_id
        ,city
        ,street_address
        ,postal_code
        ,state_province
        ,country_id)

        VALUES
        (v_location_id
        ,:NEW.city
        ,:NEW.street_address
        ,:NEW.postal_code
        ,:NEW.state_province
        ,'US');

    END;
--alta departamento
    INSERT INTO departments
    (department_id
    ,department_name
    ,manager_id
    ,location_id)

    VALUES
    (departments_seq.NEXTVAL
    ,:NEW.department_name
    ,NULL
    ,v_location_id);
END tri5;
/

```

Al realizar los siguientes INSERT sobre la vista, se ejecutará el disparador INSTEAD OF en vez del INSERT.

El primer INSERT al no existir la localidad se da de alta tanto al departamento como a la localidad. El segundo INSERT la localidad cambia en la dirección y por ello igualmente se da de alta una nueva tupla en la tabla de localidades, al igual que en la tabla de departamentos.

```

    INSERT INTO departs
    (department_name
    ,street_address
    ,postal_code
    ,city
    ,state_province
    )
    VALUES
    ('IT'
    , '10500 N De Anza Blvd'
    , '95014'
    , 'Cupertino'
    , 'California'
    );

    INSERT INTO departs
    (department_name
    ,street_address
    ,postal_code
    ,city
    ,state_province
    )
    VALUES
    ('Sports'
    , '1 Infinite Loop'
    , '95014'
    , 'Cupertino'
    , 'California'
    );

```

Disparadores DML Compuestos

Un **disparador compuesto** le permite definir las acciones que deben ocurrir en cada uno de los cuatro puntos de disparo:

- Antes de la ejecución de la sentencia.
- Antes de cada fila afectada por la ejecución de la sentencia.
- Después de cada fila afectada por la ejecución de la sentencia.
- Después de la ejecución de la sentencia.

El disparador compuesto tiene dos secciones. La primera es la sección inicial donde se declaran las variables y subprogramas. Esto básicamente es como la codificación de un trigger normal, y el código escrito en esta parte del disparador se ejecutará antes que cualquiera de los códigos definidos en las secciones opcionales.

La sección opcional es donde se crea el código para los cuatro puntos de disparo. Estos puntos deben figurar en el orden indicado en la anterior lista.

Por tanto, el formato general de este tipo de trigger es el siguiente:

```
CREATE OR REPLACE TRIGGER disparador
  FOR evento_disparador
  ON tabla|vista
  COMPOUND TRIGGER
  -- Declaraciones
  -- Subprogramas
  BEFORE STATEMENT IS
  BEGIN
    ...
  END BEFORE STATEMENT;
  --
  BEFORE EACH ROW IS
  BEGIN
    ...
  END BEFORE EACH ROW;
  --
  AFTER EACH ROW IS
  BEGIN
    ...
  END AFTER EACH ROW;
  --
  AFTER STATEMENT IS
  BEGIN
    ...
  END AFTER STATEMENT;
END disparador;
/
```

Tomando uno de los ejemplos realizados antes, que controla el salario del jefe de un nuevo empleado no sea menor que el del empleado. Si además controla las modificaciones (UPDATE) o se realice insert de varias filas; provocará un error de tabla mutante.

Una **tabla mutante**, es una tabla que está siendo modificada por una sentencia DML, o una tabla que se verá afectada por los efectos de un DELETE CASCADE debido a la integridad referencial. Las tablas mutantes sólo deben aparecer en disparadores con nivel de fila. Una tabla sobre la que se define un disparador es una tabla mutante. Es decir, para que se dé el error de tabla mutante; el disparador DML simple es a nivel de fila y en su código se consulta la tabla sobre la cual está definido.

Para darle solución al problema del ejercicio, se deberían crear dos disparadores: uno BEFORE a nivel de fila en el cual se guardarán los datos temporalmente, para ser leídos posteriormente en un disparador AFTER a nivel de sentencia en el que se consulta la tabla a la que está relacionada (no ocasionando tabla mutante) con los datos guardados por el disparador de fila realizando la comprobación de la regla de negocio a implementar.

Otra alternativa sería utilizar los disparadores compuestos. En el cual se realizaría el trabajo de ambos.

```
CREATE OR REPLACE TRIGGER tri6
FOR INSERT OR UPDATE
OF salary ON employees
WHEN (NEW.manager_id IS NOT NULL
      AND NEW.department_id IN (10,20,30,40,90)
)
COMPOUND TRIGGER
-- Declaraciones
TYPE t_tablamgr      IS TABLE OF employees.manager_id%TYPE
                        INDEX BY BINARY_INTEGER;

v_jefes t_tablamgr;
TYPE t_tablaempsal   IS TABLE OF employees.salary%TYPE
                        INDEX BY BINARY_INTEGER;

v_sal t_tablaempsal;
-- Subprogramas
BEFORE EACH ROW IS
BEGIN
    --almacenamos el jefe de cada empleado
    v_jefes(:NEW.employee_id) :=:NEW.manager_id;
    --almacenamos salario de cada empleado
    v_sal(:NEW.employee_id)   :=:NEW.salary;
END BEFORE EACH ROW;
-----
AFTER STATEMENT IS
    indice          BINARY_INTEGER;
    v_saljefe       employees.salary%TYPE;
BEGIN
    indice:=v_jefes.FIRST;--obtenemos el primer indice

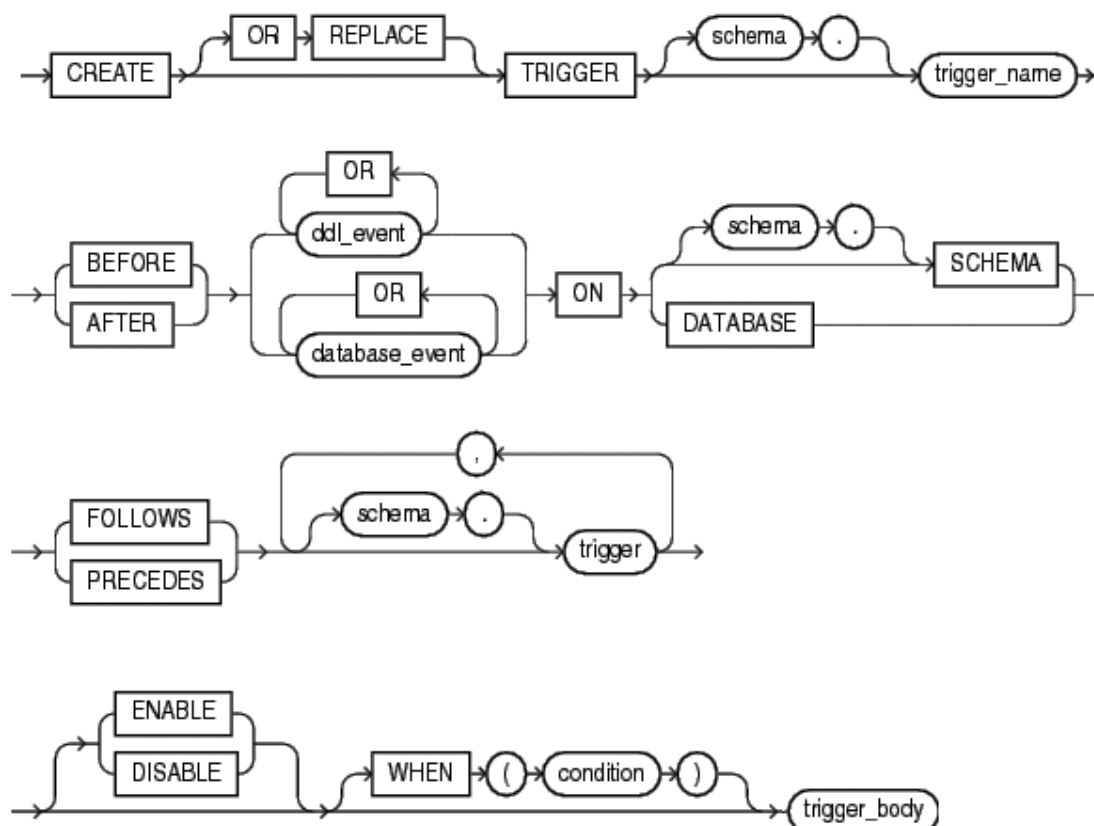
    WHILE indice IS NOT NULL LOOP
        --obtenemos el salario del jefe
        SELECT      salary
        INTO    v_saljefe
        FROM    employees
        WHERE   employee_id=v_jefes(indice);
        --si salario jefe menor salario empleado
        IF v_saljefe<v_sal(indice) THEN
            RAISE_APPLICATION_ERROR(-20005
                                   , 'No se permite ganar más que el jefe.');
```

Restricciones de los disparadores compuestos

- Los registros NEW, OLD no pueden aparecer en las secciones de sentencia ni en la parte declarativa del trigger.
- Solo la sección BEFORE EACH ROW puede cambiar los valores de NEW
- Cada sección trata sus propias excepciones, no se tratan en las otras secciones.

Disparadores No DML

La sintaxis es:



DATABASE

Se disparará cuando cualquier usuario conectado a la base de datos produzca el evento asociado al trigger.

SCHEMA

Se disparará cuando cualquier usuario conectado como schema produzca el evento asociado al trigger.

Evento DDL

Se pueden asignar a eventos del tipo `DATABASE` o `SCHEMA` (dependiendo del evento DDL) y pueden ser declarados `AFTER` o `BEFORE`. No se pueden asignar a comandos DDL que se realicen a través de un procedimiento PL/SQL.

ALTER

Se dispara cuando un comando `ALTER` modifica un objeto de la base de datos. El trigger no se disparará en caso de un `ALTER DATABASE`.

ANALYZE

Se dispara cuando se conectan o se borran las estadísticas de un objeto; o cuando se valida la estructura de un objeto.

ASSOCIATE STATISTICS

Se dispara cuando se asocia un tipo estadísticas a un objeto.

AUDIT

Se dispara cuando una sentencia SQL provoca que Oracle audite esa sentencia.

COMMENT

Se dispara cuando se añade un comentario a un objeto de la base de datos.

CREATE

Se dispara cuando se crea o añade un nuevo objeto a la base de datos. No se disparará con CREATE DATABASE o CREATE CONTROLFILE.

DIASSOCIATE STATISTICS

Se dispara cuando se desasocia un tipo estadísticas a un objeto.

DROP

Se dispara cuando se borra o elimina un objeto de la base de datos.

GRANT

Se dispara cuando un usuario da permisos de sistema, de role o de objeto a otro usuario o role.

NOAUDIT

Se dispara cuando Oracle detecta que no ha de seguir auditando sentencias SQL en un objeto.

RENAME

Se dispara cuando un comando RENAME cambia el nombre de un objeto.

REVOKE

Se dispara cuando un usuario quita permisos de sistema, de role o de objeto a otro usuario o role.

TRUNCATE

Se dispara cuando un comando TRUNCATE elimina las filas de una tabla o un cluster.

DDL

Se dispara si se produce cualquier de los eventos anteriores.

En el siguiente ejemplo se crea un disparador en el esquema HR, el disparador saltará antes de intentar realizar un DROP, e impedirá dicha acción para el usuario HR. Lo único que será capaz de borrar es el propio disparador por ser el propietario.

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON HR.SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (
    num => -20000,
    msg => 'No se permite borrar objetos.');
```

END;
/

Si por el contrario el disparador en vez de ser SCHEMA es DATABASE. El disparador impedirá realizar un DROP a cualquier usuario. El disparador puede ser creado, modificado o eliminado por un usuario con el permiso ADMINISTER DATABASE TRIGGER.

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON DATABASE
BEGIN
  RAISE_APPLICATION_ERROR (
    num => -20000,
    msg => 'No se permite borrar objetos.');
```

END;
/

Evento de base de datos.

Se pueden definir a nivel de DATABASE o de SCHEMA. Oracle crea una transacción autónoma (independientemente de la transacción que haya originado el evento)

SERVERERROR

Cuando se produce un error del servidor. Sólo AFTER. Los siguientes errores NO disparan este evento:

- ORA-01403: Datos no encontrados.
- ORA-01422: Se devuelven más filas que las esperadas.

- ORA-01423: Error producido cuando se comprobaban las filas extras en un Fetch exacto.
- ORA-01034: ORACLE no disponible
- ORA-04030: Proceso sin memoria.

LOGON

Cuando un usuario se conecta a la base de datos. Sólo AFTER

LOGOFF

Cuando un usuario se desconecta a la base de datos. Sólo BEFORE

STARTUP

Cuando se abre la base de datos. Sólo AFTER y DATABASE

SHUTDOWN

Cuando se baja y cierra la base de datos. Sólo BEFORE Y DATABASE

SUSPEND

Cuando un error del servidor deja suspendida una transacción. Sólo AFTER

En el siguiente ejemplo se tiene una tabla donde se guardará información sobre las conexiones que realicen los usuarios a la base de datos.

```
CREATE TABLE conexiones
(usuario      VARCHAR2(30)
,fecha       TIMESTAMP);
```

```
CREATE OR REPLACE TRIGGER check_user
AFTER LOGON ON DATABASE
BEGIN
    INSERT INTO conexiones
        (usuario, fecha)
    VALUES
        (USER, SYSDATE);
EXCEPTION
    WHEN OTHERS THEN
        RAISE_APPLICATION_ERROR
            (-20000
            , 'Unexpected error: ' || DBMS_UTILITY.Format_Error_Stack);
END;
```


Cursores Variables

Tabla de contenidos

Introducción	1
Utilización de cursores variables	1
Definición y Declaración de cursores variables	2
Definición	2
Declaración	3
Control de cursores variables	4
Abrir un Cursor Variable.	4
En un Procedimiento Almacenado	5
Utilizar una Variable de Recuperación.....	6
Recuperar desde un Cursor Variable	7
Cerrar un Cursor Variable.	7
Expresiones de Cursor	7
Manipulación de Expresiones de Cursor en PL/SQL.....	8
Result Sets implícitos	10
DBMS_SQL.GET_NEXT_RESULT	11
Uso de una Expresión de Cursor como Parámetro en Unidades PL/SQL.....	12
Restricciones de los cursores variables	13
Beneficios de los cursores variables.....	13

Introducción



Los cursores variables son referencias a otros cursores. Un cursor es un objeto estático y un cursor variable es un puntero a dicho cursor. Como son punteros, pueden pasar y devolver parámetros a procedimientos y funciones. Un cursor variable también puede referirse o apuntar a diferentes cursores durante su ciclo de vida.

Como un cursor, un cursor variable apunta a la fila actual en el conjunto resultante de una consulta multi-fila. Pero los cursores difieren de los cursores variables de la misma forma que las variables de las constantes. Mientras un cursor es estático, uno variable es dinámico y no tiene asociada ninguna consulta específica. Esto proporciona mayor flexibilidad.

Son como los punteros de C o Pascal, que contienen la localización en memoria de algún objeto en vez del objeto en sí. Declarando un cursor variable se crea un puntero, no un objeto.

En PL/SQL, un puntero tiene el tipo de dato REF X, REF es una abreviatura de REFERENCE y X es una clase de objeto. El tipo de dato de un cursor variable es REF CURSOR.

Para ejecutar una consulta multi-fila, Oracle abre un área de trabajo sin nombre que almacena la información procesada. Para acceder a la información, se puede utilizar un cursor explícito, que nombra el área de trabajo. O se puede utilizar un cursor variable, que apunte a esa área de trabajo.

Mientras un cursor siempre apunta a la misma área de trabajo de la consulta, un cursor variable puede referenciar a diferentes áreas de trabajo.

Utilización de cursores variables

Se pueden asignar nuevos valores a un cursor variable y pasarlos como un parámetro a los subprogramas, incluyendo subprogramas almacenados en la base de datos. Esto proporciona una vía fácil para centralizar la recuperación de los datos.

Los cursores variables están disponibles en cualquier cliente PL/SQL. Por ejemplo, se puede declarar un cursor variable en PL/SQL en entornos HOST, (en Pro*C u OCI), y pasarlos como una variable de recuperación a PL/SQL.

Las herramientas de desarrollo como Oracle*Forms u Oracle*Report tienen residente el motor PL/SQL, por lo que pueden utilizar cursores variables desde el lado cliente.

El núcleo Oracle también tiene el motor PL/SQL, y se le pueden pasar cursores variables entre la aplicación y el servidor con RPC's (Remote Procedure Calls).

Principalmente se utilizan cursores variables para pasar el resultado de una consulta entre subprogramas PL/SQL almacenados y varios clientes. Ninguno de los clientes es el propietario del conjunto resultante, simplemente comparten un puntero al área de trabajo de la consulta en la cual se encuentra almacenado el conjunto resultante.

Por ejemplo, un cliente OCI, una aplicación de Oracle*Forms y el servidor Oracle puedan referenciar la misma zona de trabajo.

Un área de trabajo de una consulta permanece mientras cualquier cursor variable apunte a él. Mientras tanto, se puede pasar el valor de un cursor variable de un lugar a otro. Por ejemplo, se puede pasar un cursor variable desde el host a un bloque PL/SQL embebido en un programa Pro*C, el área de trabajo a la que apunta el cursor variable permanece accesible después de que se complete el bloque.

Si se tiene el motor PL/SQL en el lado del cliente, las llamadas del cliente al servidor no tienen restricciones. Por ejemplo, se puede declarar un cursor variable en el lado cliente, abrir y recuperar desde él en el lado del servidor y después continuar la recuperación en el lado cliente.

Definición y Declaración de cursores variables

Para crear cursores variables, se deben de seguir dos etapas. En la primera se define un tipo REF CURSOR, entonces se declaran los cursores variables de ese tipo.

Definición

Se puede definir tipos REF CURSOR en cualquier bloque PL/SQL, subprogramas o paquetes utilizando la sintaxis siguiente:

```
TYPE nombre_tipo_ref IS REF CURSOR RETURN tipo_dato_retorno;
```

- **nombre_tipo_ref** es un especificador del tipo utilizado en las declaraciones subsiguientes de los cursores variables
- **tipo_dato_retorno** puede representar un registro o una fila en una tabla de la base de datos.

En el siguiente ejemplo, se especifica un tipo de dato retornado que representa una fila en la tabla departments de la base de datos.

```
DECLARE  
  TYPE CurTipDept IS REF CURSOR RETURN departments%ROWTYPE;
```

Los tipos REF CURSOR pueden ser restrictivos o no restrictivos. En el siguiente ejemplo se muestra una definición de un tipo REF CURSOR que especifica un tipo devuelto, pero en el caso del cursor no restrictivo no ocurre lo mismo.

```
DECLARE
  -- restrictivo
  TYPE CurTipDept IS REF CURSOR
    RETURN departments%ROWTYPE;
  -- no restrictivo
  TYPE CurTipGenerico IS REF CURSOR;
```

Los tipos de cursor REF CURSOR restrictivos provocan menos errores porque el compilador PL/SQL permite asociar a dicho cursor solo con tipos de consultas compatibles. Por otro lado los cursores no restrictivos son más flexibles porque permiten al compilador asociar dicho cursor con cualquier consulta.

Declaración

Una vez que se define el tipo REF CURSOR, se puede declarar cursores variables de ese tipo en cualquier bloque PL/SQL o subprograma. En el siguiente ejemplo, se puede declarar el cursor variable cv_dept:

```
DECLARE
  TYPE CurTipDept IS REF CURSOR
    RETURN departments%ROWTYPE;
  Cv_dept CurTipDept;           -- declaración de cursor variable
```

No se pueden declarar cursores variables en un paquete. A diferencia de las variables de paquete, los cursores variables no tienen estado persistente.

Recordar que la declaración de cursores variables crea un puntero, no un objeto. Por lo tanto, el cursor variable no puede salvarse en la base de datos.

Los cursores variables siguen las reglas de instanciación habituales. Los cursores variables locales en PL/SQL se instancian cuando se entra en un bloque o subprograma y dejan de existir cuando se sale.

En la cláusula RETURN de la definición del tipo REF CURSOR, se puede utilizar %ROWTYPE para especificar un tipo de registro que representa una fila devuelta por un cursor variable tipo restrictivo de la siguiente forma:

```
DECLARE
  TYPE CurTipTmp IS REF CURSOR
    RETURN employees%ROWTYPE;
  Cv_tmp CurTipTmp;           -- declaración de cursor variable
  TYPE CurTipEmp IS REF CURSOR
    RETURN cv_tmp%ROWTYPE;
  Cv_emp CurTipEmp;           -- declaración de cursor variable
```

También se puede utilizar el identificador %TYPE para suministrar el tipo de dato de un registro variable, de la siguiente forma:

```
DECLARE
  job_rec jobs%ROWTYPE;       -- declaración de registro variable
  TYPE CurTipJob IS REF CURSOR
    RETURN job_rec%TYPE;
  job_cv CurTipJob;           -- declaración de cursor variable
```

Se puede declarar cursores variables como parámetros formales de las funciones y procedimientos. En el siguiente ejemplo se definen el tipo de REF CURSOR CurTipEmp, y después se declara un cursor variable de ese tipo como parámetro formal del procedimiento:

```
DECLARE
  TYPE CurTipEmp IS REF CURSOR
    RETURN employees%ROWTYPE;
  PROCEDURE open_cv_emp (cv_emp IN OUT CurTipEmp)
  IS
    . . .
```

El tipo disponible SYS_REFCURSOR, que define un cursor genérico no restrictivo, es una mejora de uso, nuevo en Oracle9i. En versiones anteriores de Oracle es necesario definir un tipo de cursor variable y declarar entonces una variable de cursor de este tipo.

```
DECLARE
  -- declaración de cursor variable no restrictivo
  TYPE CurTipGenerico IS REF CURSOR;
  Cv_generico2 CurTipGenerico;
  -- a partir de Oracle 9i se puede utilizar el tipo SYS_REFCURSOR
  Cv_generico SYS_REFCURSOR;
```

Control de cursores variables

Se utilizan tres sentencias para controlar un cursor variable:

- OPEN-FOR
- FETCH
- CLOSE

Primero se utiliza OPEN para abrir el cursor variable FOR una consulta multi-fila. Entonces se realiza una recuperación (FETCH) de las filas resultantes del conjunto una a una. Cuando todas las filas se han procesado es necesario cerrar (CLOSE) el cursor variable.

Abrir un Cursor Variable.

La sentencia OPEN-FOR asocia un cursor variable con una consulta multi-fila, ejecuta la consulta, e identifica el conjunto resultante.

La sintaxis es la siguiente:

```
OPEN {cursor_variable | :nombre_cursor_variable_host }
  FOR sentencia_select;
```

Donde nombre_cursor_variable_host identifica un cursor variable declarado en un PL/SQL en un entorno host como OCI o un programa en PRO*C.

A diferencia de los cursores, los cursores variables no toman parámetros. Esto no significa que se pierda flexibilidad, porque se pueden pasar consultas completas (no sólo parámetros) al cursor variable. La consulta puede referenciar variables de recuperación y variables PL/SQL, parámetros y funciones excepto FOR UPDATE.

En el ejemplo que aparece a continuación, se abre el cursor variable cv_emp. Se pueden aplicar atributos de cursor (%FOUND, %NOTFOUND, %ISOPEN, y %ROWCOUNT) a un cursor variable.

```
IF NOT cv_emp%ISOPEN THEN
    /* Abrir el cursor variable. */
    OPEN cv_emp FOR SELECT * FROM employees;
END IF;
```

Otras sentencias OPEN-FOR pueden abrir el mismo cursor variable para diferentes consultas. Es necesario no cerrar el cursor variable antes de reabrirlo. Cuando se reabre un cursor variable para una consulta diferente, la anterior se pierde. En el caso de un cursor estático se produciría la excepción `CURSOR_ALREADY_OPEN`.

En un Procedimiento Almacenado

Normalmente, se abre un cursor variable pasándole a un procedimiento almacenado que declara un cursor variable como uno de sus parámetros formales. Por ejemplo, el siguiente procedimiento empaquetado abre un cursor variable llamado `cv_emp`.

```
CREATE PACKAGE datos_emp AS
    TYPE CurTipEmp IS REF CURSOR
        RETURN employees%ROWTYPE;
    PROCEDURE abrir_cv_emp (cv_emp IN OUT CurTipEmp);
END datos_emp;
```

```
CREATE PACKAGE BODY datos_emp AS
    PROCEDURE abrir_cv_emp (cv_emp IN OUT CurTipEmp)
    IS
    BEGIN
        OPEN cv_emp FOR SELECT * FROM employees;
    END abrir_cv_emp;
END datos_emp;
```

Cuando se declara un cursor variable como parámetro formal de un subprograma que abre el cursor variable, es necesario especificar el modo IN OUT.

Alternativamente, se puede utilizar un procedimiento standalone para abrir el cursor variable. Se define el tipo REF CURSOR en un paquete separado, y se referencia a ese tipo en el procedimiento standalone. Para la ocurrencia, si se ha creado el siguiente paquete, se puede crear procedimientos standalone que referencian los tipos definidos en el.

```
CREATE OR REPLACE PACKAGE cv_tipos AS
    TYPE CurTipGenerico IS REF CURSOR;
    TYPE CurTipEmp IS REF CURSOR
        RETURN employees%ROWTYPE;
    TYPE CurTipDept IS REF CURSOR
        RETURN departments%ROWTYPE;
END cv_tipos;
```

Para centralizar la recuperación de datos, se puede agrupar consultas de tipos compatibles en un procedimiento almacenado.

Para una mayor flexibilidad, se puede pasar un cursor variable y un selector a un procedimiento almacenado que ejecuta consultas que devuelven tipos diferentes. Ejemplo.

```
CREATE OR REPLACE PACKAGE BODY datos_emp
AS
    PROCEDURE abrir_cv_emp
        (cv_generico IN OUT          cv_tipos.CurTipGenerico
        ,elegir      IN              PLS_INTEGER)
```

```

IS
BEGIN
    CASE elegir
        WHEN 1 THEN
            OPEN cv_generico FOR
                SELECT * FROM employees;
        WHEN 2 THEN
            OPEN cv_generico FOR
                SELECT department_id
                   ,COUNT(employee_id)
                FROM employees
                GROUP BY department_id;
        WHEN 3 THEN
            OPEN cv_generico FOR
                SELECT * FROM employees
                WHERE to_number(to_char(hire_date
                                       , 'YYYY')
                               ,9999)
                           > 1999;

    END CASE;
    END abrir_cv_emp;
END datos_emp;
/

```

Utilizar una Variable de Recuperación.

Se puede declarar un cursor variable en un entorno host PL/SQL como un programa OCI o PRO*C. Para utilizar el cursor variable, se debe pasar como una variable de recuperación a PL/SQL. En el siguiente ejemplo Pro*C, se pasa un cursor variable host y un selector a un bloque PL/SQL, el cual abre el cursor variable para la consulta escogida.

```

EXEC SQL BEGIN DECLARE SECTION;
. . .
/* Declarar el cursor variable host. */
CURSOR_SQL  cv_generico;
Int         elegir;
EXEC SQL END DECLARE SECTION;
. . .

/* Inicializar el cursor variable host y el selector a un bloque PL/SQL */
EXEC SQL EXECUTE
    BEGIN
        IF :elegir = 1 THEN
            OPEN :cv_generico FOR
                SELECT * FROM employees;
        ELSIF :elegir = 2 THEN
            OPEN :cv_generico FOR
                SELECT department_id
                   ,COUNT(employee_id)
                FROM employees
                GROUP BY department_id;
        ELSIF :elegir = 3 THEN
            OPEN :cv_generico FOR
                SELECT * FROM employees
                WHERE to_number(to_char(hire_date
                                       , 'YYYY')
                               ,9999)
                           > 1999;

        END IF;
    END;
END-EXEC;

```

Los cursores variables host son compatibles con cualquier tipo de consulta devuelta.

Recuperar desde un Cursor Variable

La sentencia FETCH recupera filas una a una desde el conjunto resultante de una consulta multi-fila.

La sintaxis es:

```
FETCH {cursor_variable | :nombre_cursor_variable_host }  
INTO {nombre_variable [, nombre_variable] ..... | nombre_registro};
```

Cualquier variable en la consulta asociada se evalúa solo cuando el cursor variable se abre. Para cambiar el conjunto resultante o los valores de las variables en la consulta, se debe reabrir el cursor variable con el conjunto de variables con sus nuevos valores. Se pueden utilizar diferentes cláusulas INTO en diferentes recuperaciones con el mismo cursor variable. Cada recuperación muestra una nueva fila del mismo conjunto resultante.

PL/SQL asegura que el tipo de cursor variable devuelto es compatible con la cláusula INTO de la sentencia FETCH. Debe existir correspondencia entre los tipos del cursor variable y los campos o variables de la cláusula INTO.

Si se produce un error es en tiempo de compilación. Se produce una excepción ROWTYPE_MISMATCH, que se puede manejar.

Cuando se declara un cursor variable como parámetro formal de un subprograma que hace FETCH del cursor variable, se debe especificar el modo IN (o IN OUT).

Cerrar un Cursor Variable.

La sentencia CLOSE desactiva un cursor variable. Después el conjunto resultante está indefinido.

La sintaxis es:

```
CLOSE {cursor_variable | :nombre_cursor_variable_host };
```

Cuando se declara un cursor variable como parámetro formal de un subprograma que cierra un cursor variable, se debe especificar el modo IN (o IN OUT).

Si se trata de cerrar un cursor ya cerrado o que nunca se abrió, se dispara la excepción INVALID_CURSOR.

Expresiones de Cursor

Las expresiones de cursor, a veces conocidas como subconsultas de cursor, son un elemento del lenguaje SQL que ya estaban soportadas por versiones anteriores de Oracle en algunos entornos de ejecución, pero no por PL/SQL.

Desde Oracle9i se incorpora soporte PL/SQL para las expresiones de cursor.

Una expresión de cursor puede ser utilizada en una sentencia SELECT que abre un cursor PL/SQL y cuya manipulación se puede hacer más tarde. Puede ser usada, igualmente, como parámetro de un procedimiento o función PL/SQL, lo que es de gran relevancia en relación con las funciones de tabla.

Las nuevas características PL/SQL introducidas en Oracle9i se basan en las variables de cursor. Estas variables de cursor estaban disponibles en versiones anteriores de Oracle.

Una variable de cursor es un puntero, declarado como de tipo ref cursor, a un cursor existente. El código escrito para manipular un cursor variable puede ser reutilizado para asignaciones sucesivas a diferentes cursores existentes.

Desde Oracle9i proporciona algunas mejoras que permiten recuperaciones masivas (Bulk Collect) desde un cursor variable asignado con SQL dinámico nativo, como se puede ver en el siguiente ejemplo:

```
DECLARE
    cv_cursor SYS_REFCURSOR;

    PROCEDURE recogida_masiva (p_cursor IN SYS_REFCURSOR)
    IS
        TYPE tipo_nombre IS TABLE OF VARCHAR2(20000)
            INDEX BY BINARY_INTEGER;
        nombres TIPO_NOMBRE;
    BEGIN
        FETCH p_cursor BULK COLLECT INTO nombres;
        FOR i IN nombres.FIRST .. nombres.LAST LOOP
            DBMS_OUTPUT.PUT_LINE(nombres(i));
        END LOOP;
    END recogida_masiva;
BEGIN
    OPEN cv_cursor
    FOR 'SELECT last_name FROM employees';
    Recogida_masiva(cv_cursor);
    CLOSE cv_cursor;

    OPEN cv_cursor
    FOR 'SELECT department_name FROM departments';
    Recogida_masiva(cv_cursor);
    CLOSE cv_cursor;
END;
/
```

En versiones anteriores de Oracle, el intento de ejecutar una recuperación masiva cuando el cursor variable se asigna usando SQL dinámico nativo provoca el error ORA-01001: Invalid Cursor.

Manipulación de Expresiones de Cursor en PL/SQL

Se desea listar los nombres de departamentos, y para cada uno de ellos, listar los nombres de empleados en ese departamento. La siguiente sentencia SQL expresa los requerimientos mediante una única consulta.

```
SELECT department_name departamento,
    CURSOR (SELECT last_name apellido
        FROM employees e
        WHERE e.department_id=d.department_id
        ORDER BY apellido) empleados
FROM departments d
ORDER BY departamento;
```

Esto puede ser implementado mediante un desarrollo secuencial clásico:

```
BEGIN
    FOR departamento IN
        (SELECT department_id id, department_name nombre
```

```

        FROM departments
        ORDER BY department_name)
    LOOP
        DBMS_OUTPUT.PUT_LINE('Departamento: ' || departamento.nombre);
        FOR empleado IN
            (SELECT last_name apellido
             FROM employees
             WHERE department_id=departamento.id
             ORDER BY apellido)
        LOOP
            DBMS_OUTPUT.PUT_LINE('--Empleado: ' || empleado.apellido);
        END LOOP;
    END LOOP;
END;
/

```

Los dos ejemplos anteriores se ejecutan en entornos SQL*Plus en versiones de Oracle anteriores.

Sin embargo, si intenta asociar la sentencia SQL del primer ejemplo a un cursor dentro de un procedimiento PL/SQL, dicho procedimiento no podrá compilar en versiones de Oracle anteriores a la 9i (error PLS-00103).

```

DECLARE
    CURSOR c_departamentos IS
        SELECT department_name departamento
        ,CURSOR (SELECT first_name apellido
                 FROM employees e
                 WHERE e.department_id=d.department_id
                 ORDER BY apellido)
        FROM departments d
        ORDER BY department_name;

    v_nomdep          departments.department_name%TYPE;
    cv_empleados SYS_REFCURSOR;
    TYPE emp_tipo IS TABLE OF employees.first_name%TYPE;
    tabla_emp         emp_tipo;
    i                  BINARY_INTEGER;
BEGIN
    OPEN c_departamentos;
    LOOP
        FETCH c_departamentos INTO v_nomdep, cv_empleados;
        EXIT WHEN departamentos%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Departamento: ' || v_nomdep);
        FETCH cv_empleados BULK COLLECT INTO tabla_emp;
        i:=tabla_emp.first;
        WHILE i IS NOT NULL LOOP
            DBMS_OUTPUT.PUT_LINE('--Empleado: ' || tabla_emp(i) );
            i:=tabla_emp.NEXT(i);
        END LOOP;
    END LOOP;
    CLOSE c_departamentos;
END;
/

```

Esta última implementación contiene una única sentencia SQL, lo que la hace más optimizada que la implementación anterior con dos sentencias SQL no relacionadas.

La recuperación masiva se usa para el cursor c_empleados. Esta opción no está disponible actualmente para el cursor c_departamentos porque no es posible declarar el tipo de colección apropiada.

```

DECLARE

```



```

TYPE t_departamento IS RECORD
(Nombre_dep      departments.department_name%TYPE
,Cur_emp        SYS_REFCURSOR);
BEGIN
  NULL;
END;
/
/*Este procedimiento provoca el error "PLS-00989 La variable cursor en el
registro, object o colección no está soportada en ésta versión."*/

```

Result Sets implícitos

Esta nueva opción permite devolver cualquier *ref cursors* implícito en PL/SQL. El procedimiento **DBMS_SQL.RETURN_RESULT** es el encargado de la creación de conjuntos de resultados implícitos.

Este procedimiento devuelve el resultado de la ejecución de una sentencia de consulta a la aplicación cliente. No tiene ninguna declaración explícita de variables de salida. Su sintaxis es la siguiente:

```
DBMS_SQL.RETURN_RESULT( variable_cursor, retorno_cliente );
```

Opciones:

- *variable_cursor*: Sentencia cursor o REF CURSOR.
- *retorno_cliente*: Identifica si el resultado se retorna al cliente o no. Por defecto TRUE.

El cliente que realiza la llamada puede ser:

- Un procedimiento almacenado PL/SQL que ejecuta la instrucción recursiva utilizando el paquete DBMS_SQL.
- Un procedimiento almacenado de Java usando JDBC
- Un procedimiento almacenado de .NET utilizando ADO .NET
- Un procedimiento externo utilizando Oracle Call Interface (OCI)

Ejemplos:

1. Se tiene un procedimiento que retorna el resultado de dos consultas.

```

CREATE OR REPLACE PROCEDURE mostrar_info
IS
  v_cursor SYS_REFCURSOR;
BEGIN
  OPEN v_cursor FOR
    SELECT user FROM dual;
  DBMS_SQL.RETURN_RESULT (v_cursor);

  OPEN v_cursor FOR
    SELECT city FROM locations
    WHERE country_id IN ('UK','US') ORDER BY location_id;
  DBMS_SQL.RETURN_RESULT (v_cursor);
END mostrar_info;
/

```

2. Desde SQL*Plus se ejecuta el procedimiento almacenado anteriormente.

```
SQL> EXECUTE mostrar_info
```

```
ResultSet #1
```

```
USER
```

```
-----  
HR
```

```
ResultSet #2
```

```
CITY
```

```
-----  
Southlake  
South San Francisco  
South Brunswick  
Seattle  
London  
Oxford  
Stretford
```

DBMS_SQL.GET_NEXT_RESULT

Este procedimiento obtiene el siguiente resultado devuelto a la aplicación clientes por el procedimiento **RETURN_RESULT**. Los resultados se devuelven en mismo orden en que son retornados por este último subprograma.

Ejemplo:

3. El siguiente bloque anónimo PL/SQL llama al procedimiento anterior (*mostrar_info*) y recoge los resultados devueltos por el mismo.

```
SET SERVEROUTPUT ON

DECLARE
    v_cur      PLS_INTEGER;
    v_refcur   SYS_REFCURSOR;
    v_ret      PLS_INTEGER;
    v_col      VARCHAR2(200);
    v_cont     NUMBER := 0;
BEGIN
    v_cur := DBMS_SQL.OPEN_CURSOR(treat_as_client_for_results => TRUE);
    DBMS_SQL.PARSE( c => v_cur,
                    statement => 'BEGIN mostrar_info; END;',
                    language_flag => DBMS_SQL.NATIVE );
    v_ret := DBMS_SQL.EXECUTE(v_cur);

    LOOP
        BEGIN
            DBMS_SQL.GET_NEXT_RESULT(v_cur, v_refcur);
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                EXIT;
        END;

        v_cont := v_cont+1;
        DBMS_OUTPUT.PUT_LINE(CHR(10) || 'Resultados #' || v_cont);

        LOOP
            FETCH v_refcur INTO v_col;
            EXIT WHEN v_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(v_col);
        END LOOP;
    END LOOP;
```

```

        END LOOP;

        CLOSE v_refcur;
    END LOOP;
END;
/

```

```

Resultados #1
HR

Resultados #2
Southlake
South San Francisco
South Brunswick
Seattle
London
Oxford
Stretford

```

4.

Uso de una Expresión de Cursor como Parámetro en Unidades PL/SQL

Un cursor variable puede ser usado como un parámetro formal de un procedimiento o función PL/SQL. Una expresión de cursor define un cursor existente, y es una construcción legal en sentencias SQL (ambas cosas se cumplen en versiones previas de Oracle).

Así que podría esperarse que fuera posible invocar un procedimiento o función PL/SQL, el cual tenga un parámetro formal de tipo REF CURSOR, con una expresión de cursor como valor del parámetro.

```

MiFuncion (CURSOR (SELECT columna FROM tabla));

```

Sin embargo, esta sentencia no está permitida en versiones previas de Oracle, provocando el error ORA-22902). En Oracle9i se permite esta invocación bajo ciertas circunstancias: cuando la función (no puede ser un procedimiento) se llama desde el nivel superior de una consulta SQL.

Dada una función que puede ser llamada de la siguiente manera:

```

DECLARE
    el_cursor    SYS_REFCURSOR;
    var          PLS_INTEGER;
BEGIN
    OPEN el_cursor FOR SELECT first_name FROM employees;
    var:=MiFuncion(el_cursor);
    CLOSE el_cursor;
END;
/

```

Dicha función también puede invocarse así:

```

SELECT 'MiFuncion'  MiFuncion
FROM DUAL
WHERE MiFuncion (CURSOR (SELECT first_name FROM employees) )=0;

SELECT 'MiFuncion'  MiFuncion
FROM DUAL

```

```
ORDER BY MiFuncion (CURSOR (SELECT first_name FROM employees));
```

Esta sintaxis está permitida igualmente en la invocación de una función de tabla en la cláusula FROM de una sentencia SELECT. Sin embargo, la siguiente sintaxis no está permitida, provocando el error “PLS-00405: Subquery not allowed in this context.”

```
BEGIN
    MiFuncion (CURSOR (SELECT first_name FROM employees));
END;
/
```

Restricciones de los cursores variables

Actualmente los cursores variables están sujetos a las siguientes restricciones. Algunas de ellas se superarán en futuras versiones de PL/SQL.

- No se puede declarar cursores variables en un paquete. En la zona de declaraciones.

```
CREATE PACKAGE paquete AS
    TYPE t_empleados IS REF CURSOR
        RETURN employees%ROWTYPE;
    CV_emple t_empleados; -- no está permitido
END paquete;
/
```

- Los subprogramas remotos en otro servidor no pueden aceptar los valores de cursores variables. Por lo tanto, no se puede utilizar RPCs para pasar cursores variables de un servidor a otro.
- Si se pasa un cursor variable host (variable de recuperación) a PL/SQL, no se puede recuperar en ella en el lado del servidor a menos que también se abra en la misma llamada al servidor.
- La consulta asociada con el cursor variable en una sentencia OPEN-FOR no puede ser FOR UPDATE.
- No se pueden utilizar operadores de comparación para comprobar el cursor variable para igualdad, no igualdad o nulidad.
- No se pueden asignar nulos a un cursor variable.
- No se pueden utilizar tipos REF CURSOR para especificar tipos de columnas en una sentencia CREATE TABLE o CREATE VIEW. Las columnas de la base de datos no pueden almacenar los valores de un cursor variable.
- No se pueden especificar tipos REF CURSOR para determinar el elemento de una colección, lo que significa que esos elementos en una tabla anidada, tabla tipo índice o varray no pueden almacenar valores de cursores variables.
- Cursores y cursores variables no son interoperables. Esto significa que no se puede utilizar uno cuando se espera el otro.
- No se pueden utilizar cursores variables con SQL dinámico.

Beneficios de los cursores variables

Algunas las ventajas de utilizar cursores variables son:

- Encapsulación: las consultas se centralizan en los procedimientos almacenados que abren el cursor variable.
- Fácil Mantenimiento: Si es necesario cambiar el cursor, solo se necesitará realizar el cambio en un único lugar: el procedimiento almacenado. No es necesario cambiarlo en cada aplicación.
- Seguridad: El usuario de la aplicación es el nombre de usuario utilizado cuando la aplicación se conecta al servidor. El usuario debe tener permiso de ejecución en el procedimiento almacenado que abre dicho cursor. Pero no es necesario que el usuario tenga permiso de lectura sobre las tablas utilizadas en la consulta. Esta capacidad permite limitar los accesos a las columnas en las tablas y también el acceso a otros procedimientos almacenados.

SQL Dinámico

Tabla de contenidos

Introducción	1
EXECUTE IMMEDIATE	2
Tipos PL/SQL incluidos en SQL	3
Recuperación de varias filas.	4
El paquete DBMS_SQL.....	5
Flujo de ejecución	6
Ventajas e inconvenientes de ambos métodos.....	7
Ventajas de SQL Nativo (EXECUTE IMMEDIATE)	7
Ventajas de DBMS_SQL.	7

Introducción



En algunas ocasiones es necesario el poder escribir las sentencias SQL en momento de ejecución (Dinámico) y no en el momento en que se crea la aplicación (Estático).

Las sentencias SQL dinámicas están guardadas en cadenas que se crean en el momento de ejecución y han de contener sentencias SQL correctas. También pueden contener identificadores o variables HOST que se puedan sustituir por valores en el momento de la ejecución.

En la mayoría de SQL dinámico se utiliza la sentencia EXECUTE IMMEDIATE a excepción de las sentencias SELECT que devuelvan más de una fila, con las cuales se utiliza BULK COLLECT o OPEN-FETCH-CLOSE.

Utilizaremos SQL dinámico:

- Utilizar comandos DDL (Data Definition Language o Lenguaje de definición de datos) como CREATE; comandos DCL (Data Control Language o Lenguaje de control de datos) como GRANT; o comandos de control de sesión como ALTER SESSION. Todo este tipo de comandos no pueden ser ejecutados de una manera estática.
- Cuando no se conoce el texto de la sentencia en tiempo de compilación. Por ejemplo añadir mayor flexibilidad al poder especificar las cláusulas WHERE en momento de ejecución o realizar diferentes SELECT a diferentes esquemas dependiendo de ciertas opciones o parámetros.

- Se puede utilizar el paquete DBMS_SQL para ejecutar SQL dinámico pero EXECUTE IMMEDIATE ofrece un mejor rendimiento y más versatilidad al poder trabajar con colecciones y objetos. Además EXECUTE IMMEDIATE es más fácil de utilizar.

EXECUTE IMMEDIATE

La sentencia EXECUTE IMMEDIATE analiza, prepara y ejecuta la sentencia SQL. Permite la utilización de variables de entrada y salida. Únicamente se utilizará con las sentencias SQL que SÓLO DEVUELVAN UNA FILA.

La sintaxis es:

```
EXECUTE IMMEDIATE Comando_SQL
[INTO { variable[,variable]... | registro}]
[USING [IN | OUT | IN OUT] argumento_bind]
[, [IN | OUT | IN OUT] argumento_bind]...
[{RETURNING | RETURN} INTO argumento_bind [,argumento_bind]...];
```

- Comando_sql: la sentencia SQL que se ejecutará.
- Variable/Registro: La variable que guardará el valor de la columna o registro devuelto por el comando SQL.
- Argumento_bind.
- Un Argumento IN (Valor por defecto) será el que se pase a la sentencia SQL y sea sustituido en el momento de ejecución. No se pueden pasar nombres de objetos de esquema (Tablas, Vistas, etc.)

Si se utiliza la cláusula RETURNING únicamente los parámetros que se pueden definir son del tipo IN, los valores de salida del SQL se colocarán detrás del RETURNING INTO.

Se puede utilizar cualquier tipo de dato en los argumentos BIND a excepción de los booleanos. (TRUE, FALSE y NULL)

```
DECLARE
  Sentencia_SQL          VARCHAR2(2000);
  v_nomdep               departments.department_name%TYPE:= 'Community Manager';
  v_jobs                jobs%ROWTYPE;
  v_job_id              jobs.job_id%TYPE:= 'IT_PROG';
  plsqli_block          VARCHAR2(32767);
  v_newsal              employees.salary%TYPE;
  v_country_id          countries.country_id%TYPE:= 'NG';

BEGIN
  --DDL: se crea una tabla
  EXECUTE IMMEDIATE 'CREATE TABLE nueva (Campo1 NUMBER, campo2 NUMBER)';
  --DML: se construye y realiza un insert
  Sentencia_SQL := 'INSERT INTO departments(department_id,department_name)
                   VALUES (:1, :2)';

  EXECUTE IMMEDIATE Sentencia_SQL
  USING departments_seq.NEXTVAL, v_nomdep;
  --select de una sola fila
  Sentencia_SQL := 'SELECT * FROM Jobs WHERE job_id= :job_id';
  EXECUTE IMMEDIATE Sentencia_SQL INTO v_jobs USING v_job_id;
  --DDL: se crea un procedimiento
  EXECUTE IMMEDIATE 'CREATE PROCEDURE aumenta_sal
                   (p_employee_id IN
employees.employee_id%TYPE
                   ,p_plus
IN
BINARY_INTEGER)
                   IS
```

```

                                BEGIN
                                UPDATE employees
                                SET salary=salary+p_plus
                                WHERE employee_id=p_employee_id;
                                END aumenta_sal;';

--bloque plsql que llama a un procedimiento
plsql_block := 'BEGIN aumenta_sal(:id, :plus); END;';
EXECUTE IMMEDIATE plsql_block USING 7788, 500;
--DML: update con returning
Sentencia_SQL := 'UPDATE Employees
                    SET salary = salary*1.04
                    WHERE employee_id=:1
                    RETURNING salary INTO :2';
EXECUTE IMMEDIATE Sentencia_SQL USING 160 RETURNING INTO v_newsal;
--DML: delete
EXECUTE IMMEDIATE 'DELETE countries WHERE country_id= :country_id'
USING v_country_id;
--DDL: alter
EXECUTE IMMEDIATE q'[ALTER SESSION SET nls_date_format='dd/mm/yyyy']';
/*
se utiliza q'[ ]' para decir q lo que esta entre los corchetes
es textual, y así evitar problemas con las comillas
del formato de la fecha.
*/
END;
/

```

Tipos PL/SQL incluidos en SQL

Antes de la versión Oracle 12c R1, no se podía enlazar (unir) los tipos de datos propios de PL/SQL (por ejemplo, una matriz asociativa) con una sentencia de SQL dinámico. Ahora es posible enlazar los valores de bind variables con tipos de datos PL/SQL en los bloques anónimos, llamadas a funciones PL/SQL en consultas SQL, sentencias CALL y el operador TABLE en las consultas SQL.

Ejemplos:

1. Se crea un procedimiento almacenado.

```

CREATE OR REPLACE PROCEDURE proc_tipos (p_entrada BOOLEAN)
AUTHID DEFINER
AS
BEGIN
    IF p_entrada THEN
        DBMS_OUTPUT.PUT_LINE('ARGUMENTO TRUE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('ARGUMENTO FALSE');
    END IF;
END proc_tipos;
/

```

2. Ejecuta un bloque anónimo que usa tipos de datos PL/SQL como parte de una llamada al comando EXECUTE IMMEDIATE.

```

DECLARE
    sentencia VARCHAR2(200);
    dato BOOLEAN := TRUE;
BEGIN
    SENTENCIA := 'BEGIN PROC_TIPOS(:arg); END;';
    EXECUTE IMMEDIATE sentencia USING dato;
END;

```

/

3. En versiones anteriores a Oracle 12c la ejecución del anterior bloque mostraba el siguiente mensaje de error:

```
EXECUTE IMMEDIATE sentencia USING dato;
                                *
ERROR en línea 6:
ORA-06550: línea 6, columna 37:
PLS-00457: las expresiones deben ser de tipo SQL
ORA-06550: línea 6, columna 3:
PL/SQL: Statement ignored/
```

Recuperación de varias filas.

Para poder tratar QUERYs que devuelvan varias filas hay que generar un cursor variable con la instrucción OPEN-FOR. Leerlas con FETCH y cerrar el cursor con CLOSE.

```
OPEN {cursor_variable | :host_cursor_variable}
FOR Sentencia_sql
[USING argumento_bind[, argumento_bind]...];
```

El cursor variable ha de ser definido no restrictivo para que pueda asignarse cualquier sentencia SQL.

Para poder leer las filas devueltas se utiliza la sentencia FETCH.

```
FETCH {cursor_variable | :host_cursor_variable}
INTO { variable[,variable]... | registro};
```

Para cerrar se utiliza la sentencia CLOSE.

```
CLOSE {cursor_variable | :host_cursor_variable};
```

Ejemplo:

```
DECLARE
-- Definición de un tipo cursor no restrictivo.
TYPE tcv_generico IS REF CURSOR;
-- declara el cursor variable
cv_emple          tcv_generico;
TYPE t_idfecha IS RECORD
(id                employees.employee_id%TYPE
,fecha employees.hire_date%TYPE);
v_idfecha          t_idfecha;
BEGIN
v_idfecha.fecha:='01/01/1997';
-- abre el cursor variable
OPEN cv_emple FOR
    'SELECT employee_id,hire_date
    FROM Employees
    WHERE hire_date > :v_Fecha_in'
USING v_idfecha.fecha;
-- recorre las filas del cursor
LOOP
    FETCH cv_emple INTO v_idfecha;
    EXIT WHEN cv_emple%NOTFOUND;
    dbms_output.put_line(
        'ID: '||v_idfecha.id|
```

```

        ' HIRE_DATE: ' || v_idfecha.fecha);
    END LOOP;
    -- cerrar el cursor
    CLOSE cv_emple;    -- Cierra el cursor.
END;
/

```

El paquete DBMS_SQL.

El paquete DBMS_SQL se suministra con el sistema y permite ejecutar sentencias de SQL dinámicas tales como abrir cursores, leer los cursores, pasarle variables a los cursores, etc.

El manejo del paquete de DBMS_SQL se realiza mediante punteros a los cursores o sentencias SQL y pasando las variables como parámetros. El valor de las columnas será considerado como un parámetro del tipo OUT.

Constantes

```

v6  CONSTANT    INTEGER    := 0;
native  CONSTANT    INTEGER    := 1;
v7  CONSTANT    INTEGER    := 2;

```

Tipos

```

TYPE varchar2s IS TABLE OF VARCHAR2(256)
    INDEX BY BINARY_INTEGER;
TYPE desc_rec IS RECORD
    (col_type          BINARY_INTEGER := 0
    ,col_max_len       BINARY_INTEGER := 0
    ,col_name          VARCHAR2(32)   := ''
    ,col_name_len      BINARY_INTEGER := 0
    ,col_schema_name   VARCHAR2(32)   := ''
    ,col_schema_name_len BINARY_INTEGER := 0
    ,col_precision     BINARY_INTEGER := 0
    ,col_scale         BINARY_INTEGER := 0
    ,col_charsetid     BINARY_INTEGER := 0
    ,col_charsetform   BINARY_INTEGER := 0
    ,col_null_ok       BOOLEAN:= TRUE);
TYPE desc_tab IS TABLE OF desc_rec INDEX BY BINARY_INTEGER;

```

SQL "BULK" (de carga masiva) Tipos

```

TYPE Number_Table    IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
TYPE Varchar2_Table  IS TABLE OF VARCHAR2 (2000)
    INDEX BY BINARY_INTEGER;
TYPE Date_Table      IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
TYPE Blob_Table      IS TABLE OF BLOB
    INDEX BY BINARY_INTEGER;
TYPE Clob_Table       IS TABLE OF CLOB
    INDEX BY BINARY_INTEGER;
TYPE Bfile_Table     IS TABLE OF BFILE
    INDEX BY BINARY_INTEGER;
TYPE Urowid_Table    IS TABLE OF UROWID
    INDEX BY BINARY_INTEGER;

```

Excepciones

```

inconsistent_type    EXCEPTION;
PRAGMA EXCEPTION_INIT(inconsistent_type, -6562);

```

La excepción se levanta o produce cuando en los procedimientos COLUMN_VALUE o VARIABLE_VALUE los tipos de datos recuperados o suministrados no son iguales al tipo de datos definidos como parámetros OUT.

Flujo de ejecución

Para invocar a los procedimientos de DBMS_SQL se utilizará la notación habitual de DBMS_SQL.procedimiento (Parámetros)

- OPEN_CURSOR

```
Cursor := DBMS_SQL.OPEN_CURSOR;
```

Devuelve un identificador o puntero de un cursor. Este puntero representa la estructura de datos que Oracle genera en memoria para manejar el cursor. Este cursor es diferente a los cursores tratados en PL/SQL y en OCI. (Oracle Call Interface).

- PARSE

```
DBMS_SQL.PARSE( Cursor, 'SELECT Nombre_empleado FROM Empleados WHERE  
Cod_Empleado > :Variable',dbms_sql.native);
```

La fase de PARSE (Análisis) comprueba que la sintaxis sea correcta y se crean los planes de ejecución de la sentencia SQL. También se asigna la sentencia SQL al cursor que se ha abierto anteriormente. Si el comando que se suministra es un comando DDL se ejecuta en este momento procediendo al Commit implícito que llevan las sentencias DDL.

- BIND_VARIABLE y BIND_ARRAY

```
DBMS_SQL.BIND_VARIABLE(Cursor, ':variable', valor_variable);
```

Para poder pasar los valores a las variables definidas en la sentencia SQL dinámica se utiliza el procedimiento BIND_VARIABLE (un solo valor) o BIND_ARRAY (Una tabla de valores). En caso que se suministre una tabla de valores, en cada ejecución se tomará un valor de la tabla.

- DEFINE_COLUMN, DEFINE_COLUMN_LONG, o DEFINE_ARRAY

```
DBMS_SQL.DEFINE_COLUMN (Cursor, 1, variable);
```

Define las columnas donde se recuperarán los datos en caso que la sentencia SQL sea una SELECT. Se podría definir como el INTO de una sentencia SELECT de SQL estático.

La asignación es posicional, es decir que se define la columna según el número de orden que ocupa en la sentencia SQL.

Si la columna es del tipo LONG se debe utilizar DEFINE_COLUMN_LONG para poder recuperar el valor.

DEFINE_ARRAY permite recuperar (FETCH) múltiples líneas con una sola sentencia SELECT. Se debe especificar DEFINE_ARRAY antes de poder utilizar el procedimiento COLUMN_VALUE para recuperar las filas.

- EXECUTE

```
Filas_procesadas := dbms_sql.execute(Cursor);
```

Ejecuta el comando SQL.

- **FETCH_ROWS** y **EXECUTE_AND_FETCH**

```
Valor_sin_importancia := dbms_sql.fetch_rows(Cursor);
```

FETCH_ROWS recupera las filas que satisfagan el cursor. Cada **FETCH** recuperará un grupo de filas hasta que se llegue al final del cursor.

Es más eficiente utilizar **EXECUTE_AND_FETCH** para recuperar filas y ejecutar el cursor si únicamente se realiza un solo **FETCH**.

- **VARIABLE_VALUE**, **COLUMN_VALUE**, or **COLUMN_VALUE_LONG**

```
dbms_sql.column_value(Cursor, 1, valor_columna);
```

Cuando se utiliza una sentencia **SELECT** se debe utilizar el procedimiento **COLUMN_VALUE** para poder recuperar el valor de la columna.

Si se utiliza un procedimiento PL/SQL anónimo que devuelva valores, se utilizará **VARIABLE_VALUE** para recuperar el valor de los parámetros de salida del procedimiento PL/SQL.

Para recuperar una columna **LONG** se utilizará **COLUMN_VALUE_LONG**. En este procedimiento existe la posibilidad de indicar el desplazamiento (Offset) y la cantidad de bytes a recuperar. Una columna **LONG** puede tener hasta 2 Gb de longitud.

- **CLOSE_CURSOR**

```
DBMS_SQL.close_cursor(cursor_name);
```

Cierra el cursor.

Ventajas e inconvenientes de ambos métodos.

Ventajas de SQL Nativo (EXECUTE IMMEDIATE)

- Mayor facilidad de uso. Permite escribir la sentencia directamente o crearla de una manera sencilla con la definición de las variables y columnas.
- Mayor rendimiento a la hora de ejecutarse. A partir de Oracle 9i el motor PL/SQL ha sido diseñado para poder ejecutar SQL nativo. El rendimiento es de 1.5 a 3 veces mejor en SQL Nativo que utilizando el paquete DBMS_SQL. El uso de bind variables. Lo único que cambia de una ejecución a otra de la Sentencia SQL es el valor de las variable) acelera y mejora el rendimiento del SQL Nativo.
- Soporta tipos definidos por el usuario. Soporta todos los tipos de datos de PL/SQL así como tipos de datos definidos por el usuario, REFs y colecciones. DBMS_SQL no.
- Permite realizar **FETCH** sobre registros. DBMS_SQL no permite realizar **FETCH** sobre un tipo registro. Los tipos registro están plenamente soportados por SQL nativo.

Ventajas de DBMS_SQL.

- Está soportado en el lado del cliente. Se puede ejecutar en el lado del cliente. Cada llamada de DBMS_SQL crea un RPC (Remote Procedure Call) para satisfacer los

procedimientos de BIND_VARIABLE, EXECUTE, Etc. SQL Nativo no se puede ejecutar en el lado del cliente.

- Soporta DESCRIBE. DESCRIBE_COLUMNS permite describir las columnas de un cursor abierto. SQL Nativo no lo permite.
- Soporta comandos SQL de más de 32 Kb.
- Reutiliza los comandos SQL. Cuando se ejecuta el PARSE de un comando SQL se genera todo el plan de ejecución así como la comprobación de su sintaxis. Después se ejecuta múltiples veces cambiando únicamente los valores de las variable. SQL Nativo prepara cada vez el comando cuando se ejecuta. Aún así, la velocidad y el rendimiento de SQL Nativo supera el gasto de tiempo en su análisis.

En general es mejor y más fácil utilizar SQL Nativo que DBMS_SQL aunque dependerá de la aplicación y de los requerimientos la utilización de uno u otro.



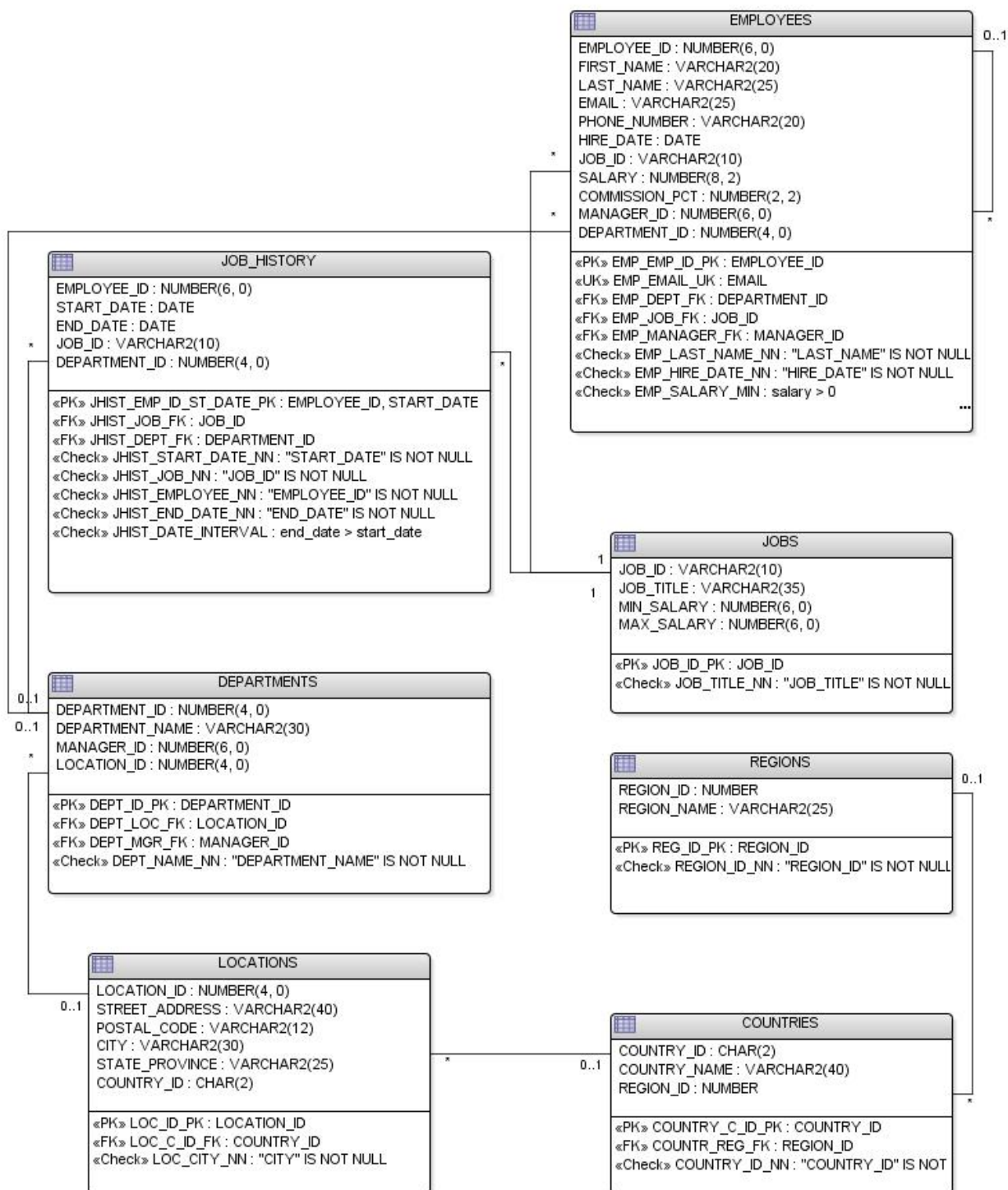
Ejercicios

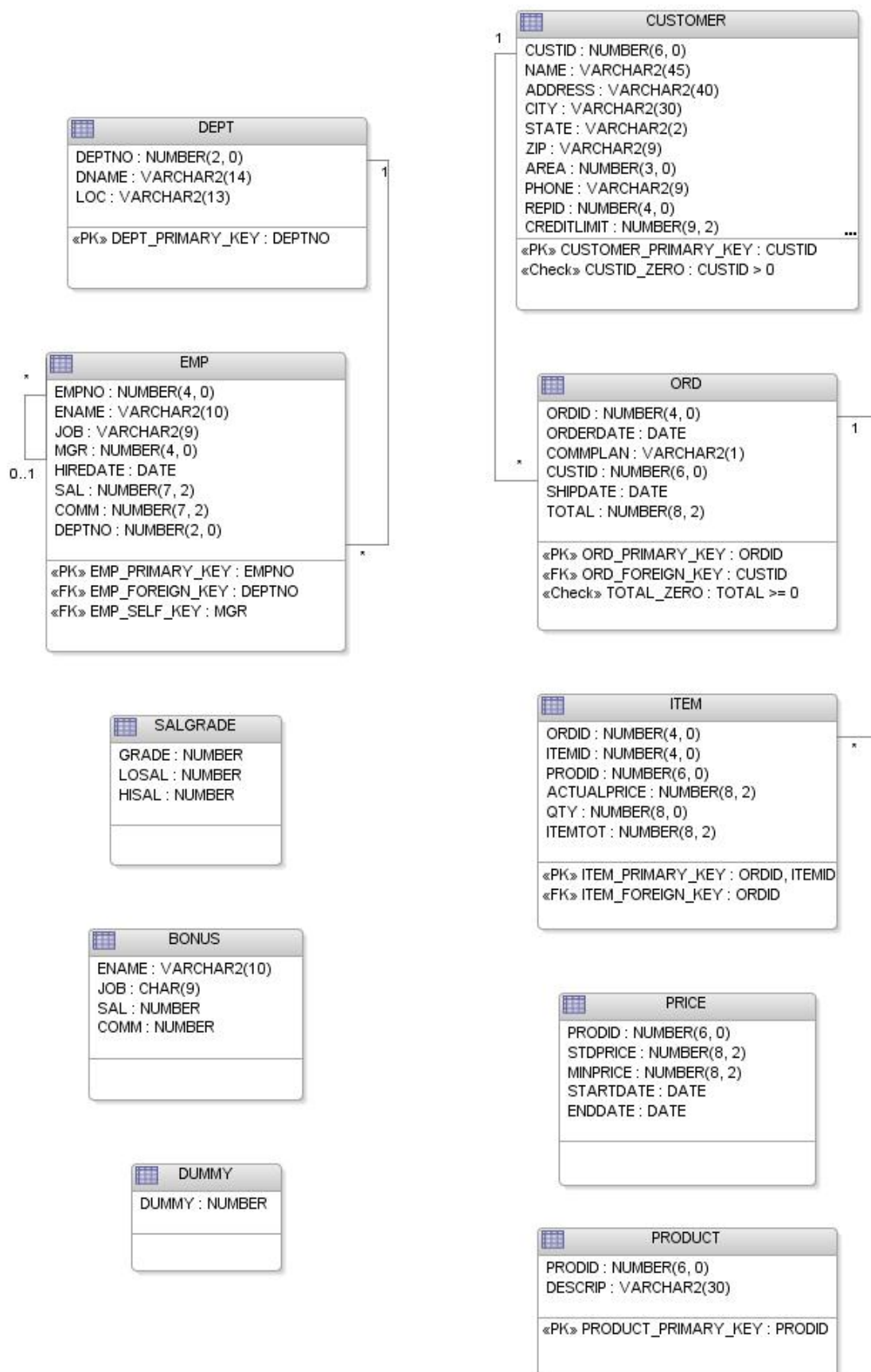
Tabla de contenidos

Modelo de datos	1
Bloques Anónimos	3
Cursores	3
Excepciones.....	3
Funciones y Procedimientos	4
Paquetes	5
Colecciones	6
Disparadores	6
Cursores variables	6
SQL dinámico	7
Soluciones	7
Bloques Anónimos	7
Ejercicio 1	7
Ejercicio 2	7
Ejercicio 3	8
Ejercicio 4	8
Ejercicio 5	8
Ejercicio 6	9
Ejercicio 7	9
Cursores	10
Ejercicio 1	10
Ejercicio 2	10
Ejercicio 3	11
Ejercicio 4	11
Excepciones.....	12
Ejercicio 1	12
Ejercicio 2	12
Ejercicio 3	13
Funciones y Procedimientos.....	14
Ejercicio 1	14
Ejercicio 2	14
Ejercicio 3	14
Ejercicio 4	15
Ejercicio 5	15
Ejercicio 6	16
Ejercicio 7	17
Paquetes	18
Ejercicio 1	18

Colecciones	20
Ejercicio 1	20
Disparadores	21
Ejercicio 1	21
Ejercicio 2	21
Ejercicio 3	22
Cursores Variables	22
Ejercicio 1	22
SQL Dinámico	23
Ejercicio 1	23
Ejercicio 2	23

Modelo de datos





Bloques Anónimos

1. Bloque anónimo que muestra mediante salida DBMS_OUTPUT la fecha y hora actual.
2. Dado un numero de empleado; mostrar nombre, apellido, salario, nombre del departamento y nombre de la ciudad.
3. Dados dos números, muestra un mensaje indicando cual es mayor o si son iguales.
4. Dado un número, muestra un mensaje diciendo si es par o impar.
5. Dado el nombre de una tabla del usuario, mostrar un mensaje indicando el nombre de su primary key en caso de que exista.
6. Dado un nombre de ciudad y de país dar de alta a la ciudad en ese país:
 - Si no existe el país, también darle de alta.
 - Código del país; dos primeras letras del nombre.
 - Código de la ciudad lo obtenemos de la secuencia existente locations_seq
7. Crear un procedimiento que indique si un número es primo o no.

Cursores

8. Nombre y el 30% del salario de los empleados.
9. Dado un nombre de país, mostrar mediante salida DBMS_OUTPUT
 - Nombre y apellido del empleado
 - Número del jefe del empleado
 - Nombre y número del departamento
 - Nombre de la ciudad
 - Nombre del trabajo
10. Listar código, nombre y apellido de empleados de una ciudad donde la primera letra de su apellido este entre dos letras –inclusive- dadas previamente como parámetros del cursor.
11. El número de empleado que tiene un salario más bajo que la media del salario más alto y más bajo del departamento donde trabaja el empleado. El literal “la diferencia de sueldo es: “

Excepciones

12. Dado un número de localidad mostrar mediante DBMS_OUTPUT:
 - Si el código no existe
 - El número de empleados y departamentos asociados.
13. Introducir el nombre de una ciudad y mostrar en pantalla mediante DBMS_OUTPUT el número de empleados que trabajan en esa ciudad y a qué país pertenece.
 - Existe y tiene empleados

“La Ciudad de XXXX está en XXXXX”

“La Ciudad XXXX tiene asignada NNN empleados.”

- No existe la ciudad
“No se ha encontrado la ciudad: XXXX “
- Que no tenga empleados trabajando en ella
- “La ciudad XXXX no tiene empleados asignados.”

14. Aumentar el sueldo de todos los empleados en un 10%.

- Si la suma de los aumentos de todos los empleados de un mismo tipo de trabajo es mayor de 5000 deshacer estos aumentos (Sólo de los empleados de ese mismo tipo de trabajo, no de todos los empleados aumentados) y mostrar el literal
- “El tipo de trabajo XXXX ha superado los 5000 en aumentos”
- Si no supera los 5000 mostrar: el número de empleados aumentado y cuanto se ha aumentado.

Funciones y Procedimientos

15. Función que calcula el bono (salario * comisión) dado un número de empleado existente.
16. Función “presupuesto” que consiste en la suma de salarios totales (salario + bono) incrementado un 5 % cada uno; de todos los empleados o para un departamento dado por parámetro. Utilizar la función “bono” previamente implementada.
17. Procedimiento “alta_job_histo” inserta en la tabla job_history la información dada por parámetro (employee_id, start_date, end_date, job_id, department_id).
18. Procedimiento “hora_laboral” que se encarga de comprobar si el momento actual está dentro de la jornada laboral (9:00 a 18:00 de lunes a viernes); en caso contrario provoca una excepción -20006 mostrando un mensaje “No se permite fuera de horario laboral”
19. Procedimiento “alta_departamento”.
 - Recibe como parámetro el nombre del departamento.
 - Opcionalmente recibirá el código de la localidad y el número del empleado que será el jefe del departamento, ambos existentes.
 - El número del departamento se obtiene de la secuencia (existente) departments_seq en caso de no obtenerlo por parámetro.
 - Si la localidad cuenta ya con 25 departamentos, dar una excepción -20107 indicando que no se permiten más departamentos en la localidad.
 - Si el empleado que será el jefe del departamento gana menos que cualquier jefe de departamento, igualarle el salario a la media de salarios de los jefes de departamento. Dar un mensaje indicando el porcentaje de subida.
20. Crear un procedimiento que nos muestre en pantalla los datos del historial de un empleado:

La fecha de inicio del trabajo realizado

La fecha final del trabajo realizado

El número de departamento

El nombre del trabajo realizado

La ciudad donde lo realizó

En caso que no tuviera Historial ese empleado deberá aparecer el literal: "No tiene historial"

21. Crear un procedimiento que nos muestre en pantalla los siguientes datos de un departamento:

El número de empleado

El nombre y apellido del empleado

La fecha en que entró en la empresa

El trabajo que realiza (nombre)

Su historial. Utilizar el procedimiento previamente creado.

En caso que no existiera el departamento se mostrará un error y finalizar el procedimiento. El número de error será el -20101 y el literal "No existe este Departamento".

Si no tiene empleados el departamento se mostrará el literal "El departamento XX no tiene empleados asignados"

Paquetes

22. Crear un paquete "Alta" con el siguientes procedimiento público.

- Empleado. Se le suministra los siguientes parámetros:
 - Nombre
 - Apellido
 - El tipo de trabajo
 - Número empleado: opcional, en caso de no proporcionarlo se obtiene de la secuencia employees_seq
 - Número de teléfono: opcional
 - El departamento asignado: opcional
 - El porcentaje de comisión: opcional

El resto de los datos se calcularán de la siguiente forma:

Dirección de correo electrónico: Una función privada devolverá la dirección que será el nombre.apellido. En el caso que hubiera otra dirección igual, la función devolverá nombre.apellido.numeroempleado.

Fecha de alta: La fecha del sistema.

Manager_id: El mismo que esté asignado al departamento

Salario: Una función pública devolverá el salario mínimo de los empleados de ese departamento. En caso que fuera más pequeño que el salario mínimo del tipo de trabajo se aplicaría el salario mínimo del tipo de trabajo.

Colecciones

23. Crear un bloque anónimo:

- Almacenar en una tabla anidada el employee_id y el manager_id; de los empleados del departamento 60.
- El salario de los empleados, cuyo employee_id tenemos en la colección; deben de tener un salario al menos del 51% del de su jefe (cuyo número de empleado lo tenemos en la colección –manager_id-).

Disparadores

24. Disparador que se ejecutará al realizar DML sobre la tabla de employees e impedirá la instrucción disparadora si esta fuera de la jornada laboral (Utilizar el procedimiento “hora_laboral” creado previamente).

25. Previamente se ha de crear una tabla que contendrá las siguientes columnas:

- Nombre de usuario: Usuario al cual permitiremos que modifique el salario de la tabla employees.
- Veces: Veces que permitimos que modifique alguna fila de la columna del salario.
- Realizadas: el número de ocasiones en que ha modificado uno de los salarios.

Un ejemplo de creación de la tabla sería:

```
create table quien_puede
(
    usuario      varchar2(30)
    , veces      NUMBER(3)
    , realizadas NUMBER(3)
);
```

El disparador ha de comprobar que el usuario puede modificar la columna salary de la tabla employees; para ello comprobará si está dado de alta en la tabla. Si no existe ha de levantar un error de aplicación número 21000.

Si puede modificar, se comprobará si no ha superado el número de veces que puede modificar. En caso que lo supera se notificará un error de aplicación 20101.

26. Crear disparador de base de datos que se ejecute en caso que se intente hacer DDL sobre el esquema e impida dicha acción.

Cursores variables

27. Crear un paquete con dos procedimientos.

- El primero ha de crear un cursor variable dependiendo de una opción:
 - Sobre la tabla employees
 - Sobre la tabla Departments
 - Sobre la tabla Locations

- El segundo ha de contar las filas de las tablas dependiendo de la opción escogida anteriormente.

SQL dinámico

28. Procedimiento que activa o desactiva todos los disparadores (de una o todas las tablas) de los que es propietario un usuario. Sus parámetros:
 - Usuario: por defecto el usuario que ejecuta.
 - Tabla: por defecto todas las tablas en las que tiene disparadores
 - Opción: por defecto desactiva disparadores
29. Crear un bloque anónimo que cree una tabla. Se le suministrará los siguientes valores.

El nombre de la nueva tabla a crear.

Una tabla con las columnas a crear. Estas columnas serán las que se deseen de la tabla employees.

Una vez creada la tabla con las columnas se ha de insertar los empleados con el salario menor a la media de los salarios. Únicamente se insertarán los datos de las columnas suministradas.

Si se indica la columna salary se ha de mostrar la suma de los salarios insertados en la nueva tabla.

-

Soluciones

Bloques Anónimos

Ejercicio 1

```
BEGIN
    dbms_output.put_line(to_char(sysdate, 'dd/mm/yyyy hh24:mi:ss'));
END;
/
```

Ejercicio 2

```
DECLARE
    --declaramos variables
    v_nombre          alumno.employees.first_name%TYPE;
    v_apellido        employees.last_name%TYPE;
    v_salario          employees.salary%TYPE;
    v_departamento   departments.department_name%TYPE;
    v_ciudad           locations.city%TYPE;
BEGIN
    --consultamos informacion
    SELECT
        first_name,
        last_name,
        salary,
        department_name,
        city
    INTO
```

```

        v_nombre,
        v_apellido,
        v_salario,
        v_departamento,
        v_ciudad
FROM
    employees
    JOIN departments
        USING(department_id)
    JOIN locations
        USING(location_id)
WHERE
    employee_id=&p_employee_id;

--mostramos informacion
DBMS_OUTPUT.PUT_LINE(
    v_nombre||' '||
    v_apellido||' '||
    v_salario||' '||
    v_departamento||' '||
    v_ciudad
    );

END;
/

```

Ejercicio 3

```

declare
    v_numero1    number:=&uno;
    v_numero2    number:=&dos;
begin
    if      v_numero1>v_numero2 then
        dbms_output.put_line(
            v_numero1||' es mayor que '||v_numero2
        );
    elsif v_numero2>v_numero1 then
        dbms_output.put_line(
            v_numero2||' es mayor que '||v_numero1
        );
    else
        dbms_output.put_line(
            v_numero2||' es igual que '||v_numero1
        );
    end if;
end;
/

```

Ejercicio 4

```

declare
    v_numero    PLS_INTEGER:=&uno;
begin
    if      mod(v_numero,2)=0 then
        dbms_output.put_line(v_numero||' es par');
    else
        dbms_output.put_line(v_numero||' es impar');
    end if;
end;
/

```

Ejercicio 5

```

DECLARE
    v_tabla          varchar2(2000):='&tabla';

```

```
        v_constraint_name          varchar2(2000);
BEGIN
    select          constraint_name
    into    v_constraint_name
    from    all_constraints
    where table_name=v_tabla
           and constraint_type='P'
           and owner=user;

    dbms_output.put_line(v_constraint_name);
END;
/
```

Ejercicio 6

```
declare
    v_city          locations.city%type:='&ciudad';
    v_contry_name    countries.country_name%type:='&pais';
    v_country_id     countries.country_id%type;
    v_aux            PLS_INTEGER;
begin
    --comprobamos si existe el pais
    select COUNT(country_id)
    into    v_aux
    from    countries
    where   country_name=v_country_name;

    if v_aux=0 then--no existe el pais
        --damos de alta
        v_country_id:=SUBSTR(country_name,1,2);
        insert into countries(country_id
                               ,country_name)
        values(v_country_id
               ,v_country_name);

    else--si existe el pais
        --consultamos id
        select country_id
        into    v_country_id
        from    countries
        where   country_name=v_country_name;
    end if;

    insert into locations(location_id
                           ,city
                           ,country_id)
    values(locations_seq.nextval
           ,v_city
           ,v_country_id)

end;
/
```

Ejercicio 7

```
DECLARE
    nume    PLS_INTEGER:=&numero;
    raiz    PLS_INTEGER;
    valor   PLS_INTEGER;
BEGIN
    raiz := TRUNC(SQRT(nume),0);
    FOR t IN REVERSE 2..raiz LOOP
        valor :=MOD (nume,t);
        IF valor = 0 THEN
            DBMS_OUTPUT.PUT_LINE ( 'El número '|| nume
                                   ||' no es primo.');
```

```

        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('El número ' || nume || ' es primo. ');
END;
/

```

Cursores

Ejercicio 1

```

declare
    --declarar el cursor
    cursor nom_cur is
        select first_name,
               salary*0.30 ali --obligado alias, sin palabra AS
        from employees;
    v_reg nom_cur%rowtype;
begin
    --abrir el cursor
    open nom_cur;
    -----
    loop
        --leer los datos
        fetch nom_cur into v_reg;
        exit when nom_cur%notfound; --salimos cuando no hay mas datos
        dbms_output.put_line(v_reg.first_name||v_reg.ali);
    end loop;
    -----
    --cerrar el cursor
    close nom_cur;
end;
/

```

Ejercicio 2

```

declare
    v_country_name      countries.country_name%type:='&pais';
    cursor cur is
        select
            first_name
            ,last_name
            ,e.manager_id      --evitar columna definida de forma ambigua
            ,department_name
            ,d.department_id    --evitar columna definida de forma ambigua
            ,city
            ,job_title
        from
            employees e
            join
            departments d
                on(e.department_id=d.department_id)
            join
            locations l
                on(d.location_id=l.location_id)
            join
            countries c
                on(l.country_id=c.country_id)
            join
            jobs j
                on(e.job_id=j.job_id)
        where
            country_name=v_country_name;
begin
    for reg in cur loop
        dbms_output.put_line(

```

```

        rpad(reg.first_name,10)||
        rpad(reg.last_name,10)||
        rpad(reg.manager_id,4)||
        rpad(reg.department_name,10)||
        rpad(reg.department_id,3)||
        rpad(reg.city,15)||
        reg.job_title
    );
end loop;
end;
/

```

Ejercicio 3

```

declare
    v_ciudad      locations.city%type:='&ciu';--ciudad
    --declaramos el cursor
    cursor cur(p_inicio CHAR,p_fin CHAR) is
        select
            employee_id,
            first_name,
            last_name
        from
            employees e
            join departments d
                on(e.department_id=d.department_id)
            join locations l
                on(l.location_id=d.location_id)
        where
            SUBSTR(last_name,1,1)
            between p_inicio and p_fin
            and city=v_ciudad
        ;

begin
    --mostramos la informacion del cursor
    DBMS_OUTPUT.PUT_LINE(RPAD(LPAD('Empleados',20,'-'),30,'-'));
    for reg in cur(p_inicio=>'&ini',p_fin=>'&fin') loop
        DBMS_OUTPUT.PUT_LINE(
            reg.employee_id||' '||
            RPAD(reg.first_name,13)||' '||
            RPAD(reg.last_name,13)
        );
    end loop;
end;
/

```

Ejercicio 4

```

DECLARE
    CURSOR mi_cursor IS
        SELECT employee_id
            ,TRUNC((SELECT (MAX(salary)+MIN(salary))/2
                        FROM employees
                        WHERE department_id=e.department_id)-salary
                ,2) diferencia
        FROM employees e
        WHERE salary<(
            SELECT (MAX(salary)+ MIN(salary))/2
            FROM employees
            WHERE department_id=e.department_id);

BEGIN
    FOR reg IN mi_cursor LOOP
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(reg.employee_id)
            ||' La diferencia es: ' || TO_CHAR(reg.diferencia) );
    END LOOP;

```



```
END;  
/
```

Excepciones

Ejercicio 1

```
declare  
    --variable en la que asignamos un codigo de localidad  
    v_loc_id      locations.location_id%type:=&lod_id;  
    v_loc_id2     pls_integer;--almacena numero de localidad  
    v_numemp      pls_integer;--almacena numero de empleados  
    v_numdep      pls_integer;--almacena numero de departamentos  
begin  
    --comprobamos si existe la  
    --localidad buscandola en la tabla  
    select      location_id  
    into  v_loc_id2  
    from  locations  
    where location_id=v_loc_id;  
  
    --existe la localidad  
    --consultamos numero de empleados  
    select COUNT(employee_id)  
    into v_numemp  
    from employees  
        join departments  
        using(department_id)  
    where location_id=v_loc_id;  
    --consultamos numero de departamentos  
    select COUNT(department_id)  
    into v_numdep  
    from departments  
    where location_id=v_loc_id;  
    --mostramos informacion  
    dbms_output.put_line('Empleados: '||v_numemp);  
    dbms_output.put_line('Departamentos: '||v_numdep);  
exception  
    when no_data_found then--no existe localidad  
        dbms_output.put_line('no existe');  
end;  
/
```

Ejercicio 2

```
declare  
    e_noemp      exception;  
    v_pais       countries.country_name%type;  
    v_numemp     PLS_INTEGER;  
    v_ciudad     locations.city%type:='&ciu';  
    v_loc_id     locations.location_id%TYPE;  
begin  
    --consultamos el id de la ciudad  
    --si no existe la ciudad no_data_found  
    select location_id  
    into v_loc_id  
    from locations  
    where city=v_ciudad;  
    --consultamos el nombre del pais  
    select country_name  
    INTO v_pais  
    from locations join countries  
        using(country_id)  
    where city=v_ciudad;  
    --consultamos cuantos empleados tiene
```

```

select count(employee_id)
into v_numemp
from employees
      join departments using(department_id)
      join locations using(location_id)
where city=v_ciudad;

if v_numemp=0 then--si no tiene empleados
    raise e_noemp;--lanzamos la excepcion
end if;
--mostramos los datos
dbms_output.put_line('La ciudad '||v_ciudad
                    ||' tiene '||v_numemp||
                    ' empleados.');
```

```

dbms_output.put_line('La ciudad '||v_ciudad
                    ||' está en '||v_pais);
```

```

exception
when no_data_found then
    dbms_output.put_line('La ciudad '||v_ciudad
                        ||' no existe.');
```

```

when e_noemp then
    dbms_output.put_line('La ciudad '||v_ciudad
                        ||' no tiene empleados.');
```

```

end;
/
```

Ejercicio 3

```

DECLARE
    CURSOR cur_trabajos IS
        SELECT distinct job_id FROM employees;
    Nume          PLS_INTEGER :=0;
    Cantidad      NUMBER :=0;
BEGIN
    FOR reg IN cur_trabajos LOOP
        DECLARE
            CURSOR cur_sal_job IS
                SELECT salary
                FROM employees
                WHERE job_id = reg.job_id
                FOR UPDATE OF salary;
            nume_local          PLS_INTEGER :=0;
            cantidad_local      NUMBER :=0;
            mas_5000            EXCEPTION;
        BEGIN
            SAVEPOINT A;
            FOR reg_local IN cur_sal_job LOOP
                cantidad_local:=cantidad_local+reg_local.salary * .10;
                nume_local := nume_local +1;
                IF cantidad_local > 5000 THEN
                    RAISE mas_5000;
                END IF;
                UPDATE employees
                SET salary=salary *1.10
                WHERE CURRENT OF cur_sal_job;
            END LOOP;
            cantidad := cantidad + cantidad_local;
            nume := nume + nume_local;
        EXCEPTION
            WHEN mas_5000 THEN
                cantidad_local :=0;
                nume_local :=0;
                DBMS_OUTPUT.PUT_LINE ( 'El tipo de trabajo '
                                    || RPAD(reg.job_id,10)
                                    || ' ha superado los 5000 de aumento.');
```

```

                ROLLBACK TO A;
```

```

        END;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Se ha aumentado '
                        || cantidad || ' a '
                        || nume || ' empleados' );
END;
/

```

Funciones y Procedimientos

Ejercicio 1

```

create or replace FUNCTION bono
(p_employee_id      in      employees.employee_id%type)
return number
is
    v_bono number;
begin
    select      salary*nvl(commission_pct,0)
    into        v_bono
    from        employees
    where       employee_id=p_employee_id;

    return      v_bono;
end;
/

```

Ejercicio 2

```

create or replace FUNCTION presupuesto
(p_department_id    in
    departments.department_id%type
    default          null)
return number
is
    v_presupuesto    number;
begin
    select SUM( (salary+bono(employee_id)) *1.05 )
    into    v_presupuesto
    from    employees
    where   p_department_id is null
           or department_id=p_department_id;

    return      v_presupuesto;
end;
/

```

Ejercicio 3

```

create or replace procedure histo_job
(p_employee_id      IN job_history.employee_id%TYPE
,p_start_date      IN job_history.start_date%TYPE
,p_end_date        IN job_history.end_date%TYPE
,p_job_id          IN job_history.job_id%TYPE
,p_department_id   IN job_history.department_id%TYPE
                  DEFAULT NULL
)
IS
BEGIN
    IF p_start_date>p_end_date THEN
        raise_application_error(-20100
                                , 'start_date no puede ser mayor a end_date. ');
    END IF;
    INSERT INTO job_history

```

```

        (employee_id
        ,start_date
        ,end_date
        ,job_id
        ,department_id
        )
VALUES
        (p_employee_id
        ,p_start_date
        ,p_end_date
        ,p_job_id
        ,p_department_id
        );
END histo_job;
/

```

Ejercicio 4

```

create or replace procedure hora_laboral
is
begin
    if      TO_CHAR(SYSDATE, 'hh24:mi')
           not between '09:00' and '18:00'
    or TRIM(TO_CHAR(sysdate, 'd'))
           in ('sábado', 'domingo')
    then
        RAISE_APPLICATION_ERROR(-20006
                                , 'No se permite fuera del horario laboral. ');
    end if;
end;
/

```

Ejercicio 5

```

create or replace procedure alta_departamento
(p_department_id      IN      departments.department_id%TYPE
                        DEFAULT NULL
,p_department_name    IN      departments.department_name%TYPE
,p_location_id        IN      locations.location_id%TYPE
                        DEFAULT NULL
,p_manager_id         IN      departments.manager_id%TYPE
                        DEFAULT NULL
)
--PRE: p_location_id y p_manager_id contienen valores existentes
IS
v_numdep              BINARY_INTEGER;
v_porcen              NUMBER;
v_minsal              employees.salary%TYPE;
v_mediasal            employees.salary%TYPE;
v_salario             employees.salary%TYPE;
v_department_id       departments.department_id%TYPE:=p_department_id;
BEGIN
    IF p_location_id IS NOT NULL THEN
        --controlamos el numero de departamentos
        --por localidad <=25
        SELECT      count(department_id)
        INTO      v_numdep
        FROM      departments
        WHERE      location_id=p_location_id;
        IF v_numdep>24 THEN
            raise_application_error(-20107,
                                'No se permite mas de 25
                                departamentos en
                                la localidad
                                '||p_location_id);

```

```

        END IF;
    END IF;
    IF p_manager_id IS NOT NULL THEN
        --minimo y media de salarios
        --de los que son jefes de departamento
        BEGIN
            SELECT          MIN(salary),avg(salary)
            INTO    v_minsal,v_mediasal
            FROM      employees
            JOIN      departments
            ON        (employees.department_id
                     =departments.department_id)
            WHERE     departments.manager_id
                     =employees.employee_id;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                --aun no existen jefes de departamento
                NULL;
        END;
        --salario del que sera nuevo jefe de departamento
        SELECT salary
        INTO    v_salario
        FROM      employees
        WHERE     employee_id=p_manager_id;
        --comparamos salarios
        IF v_salario<v_minsal THEN
            UPDATE employees
            SET     salary=v_mediasal
            WHERE   employee_id=p_manager_id;
            v_porcen:=(v_mediasal*100)/v_salario;
            dbms_output.put_line('Se ha subido el salario un '
                                ||v_porcen);
        END IF;
    END IF;
    IF p_department_id IS NULL THEN
        v_department_id:=departments_seq.nextval;
    END IF;
    --damos de alta el departamento
    INSERT INTO departments
        (department_id
        ,department_name
        ,location_id
        ,manager_id)
    VALUES
        (v_department_id
        ,p_department_name
        ,p_location_id
        ,p_manager_id
        );
END alta_departamento;
/

```

Ejercicio 6

```

CREATE OR REPLACE PROCEDURE Muestra_historial
( p_employee_id IN employees.employee_id%TYPE)
IS
    CURSOR historial IS
        SELECT start_date,end_date, job_title,city,department_id
        FROM job_history JOIN jobs USING(job_id)
        JOIN departments USING(department_id)
        JOIN locations      USING(location_id)
        WHERE employee_id = p_employee_id;
    numero          PLS_INTEGER :=0;
BEGIN

```

```

SELECT COUNT(*)
INTO    numero
FROM    job_history
WHERE   employee_id = p_employee_id;
IF numero = 0 THEN -- no tiene historial
    DBMS_OUTPUT.PUT_LINE ('El empleado ' || p_employee_id || ' no tiene
historial');
ELSE
    --tiene historial
    DBMS_OUTPUT.PUT_LINE ('Inicio    Final        Dep_id Nombre Trabajo
Ciudad');
    FOR reg IN historial LOOP
        DBMS_OUTPUT.PUT (reg.start_date);
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (reg.END_date);
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (rpad(reg.department_id,7,' '));
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (rpad(reg.job_title,30,' '));
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (reg.city);
        DBMS_OUTPUT.NEW_LINE;
    END LOOP;
END IF;
END;
/

```

Ejercicio 7

```

CREATE OR REPLACE PROCEDURE muestra_dept
(p_department_id IN employees.department_id%TYPE)
IS
    CURSOR empleados IS
        SELECT employee_id
               , first_name || ' ' || last_name Nombre
               , hire_date
               , job_title
        FROM employees JOIN jobs USING(job_id)
        WHERE department_id=p_department_id;
    numero PLS_INTEGER :=0;
BEGIN
    --existe el departamento
    BEGIN
        SELECT department_id
        INTO    numero
        FROM    departments
        WHERE   department_id = p_department_id;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE_APPLICATION_ERROR (-20101,'No existe este
departamento');
    END;
    --numero de empleados
    SELECT COUNT(*)
    INTO    numero
    FROM    employees
    WHERE   department_id = p_department_id;
    IF numero = 0 THEN
        DBMS_OUTPUT.PUT_LINE( 'El Departamento '
                               || p_department_id
                               || ' no tiene empleados asignados.'
                               );
    ELSE
        DBMS_OUTPUT.PUT_LINE ('Num.    Nombre y apellidos          Alta
Trabajo');
        FOR reg IN empleados LOOP
            DBMS_OUTPUT.PUT(reg.employee_id);

```

```

        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (rpad(reg.nombre,30,' '));
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (reg.hire_date);
        DBMS_OUTPUT.PUT ( ' ');
        DBMS_OUTPUT.PUT (reg.job_title);
        DBMS_OUTPUT.NEW_LINE;
        muestra_historial(reg.employee_id);
        DBMS_OUTPUT.PUT_LINE('-----');
---');
    END LOOP;
END IF;
END;
/

```

Paquetes

Ejercicio 1

```

CREATE OR REPLACE
PACKAGE alta
AS
    PROCEDURE empleado
        (p_first_name      IN      employees.first_name%TYPE
        ,p_last_name       IN      employees.last_name%TYPE
        ,p_job_id          IN      employees.job_id%TYPE
        ,p_employee_id     IN      employees.employee_id%TYPE
                                DEFAULT NULL
        ,p_phone_number    IN      employees.phone_number%TYPE
                                DEFAULT NULL
        ,p_department_id   IN      employees.department_id%TYPE
                                DEFAULT NULL
        ,p_commission_pct  IN      employees.commission_pct%TYPE
                                DEFAULT NULL
        );
    FUNCTION get_salario
        (p_job_id          IN      employees.job_id%TYPE
        ,p_department_id   IN      employees.department_id%TYPE
        )
        RETURN employees.salary%TYPE;
END alta;
/

```

```

CREATE OR REPLACE
PACKAGE BODY alta
AS
    --variables, funciones privadas
    -----
    FUNCTION get_email
        (p_employee_id     IN      employees.employee_id%TYPE
        ,p_first_name      IN      employees.first_name%TYPE
        ,p_last_name       IN      employees.last_name%TYPE
        )
        RETURN employees.email%TYPE
        --POST: retorna el email formado por primera letra
        --del nombre concatenado con el apellido
        --en caso de existir se le concatena el numero de empleado
    IS
        v_email            employees.email%TYPE
                           :=UPPER(SUBSTR(p_first_name,1,1)||p_last_name);
        v_employee_idemployees.employee_id%TYPE;
    BEGIN
        --consultamos empleados que tengan el email

```

```

SELECT employee_id
INTO   v_employee_id
FROM   employees
WHERE  email=v_email;

--si la select no da una excepcion, ya existe el email
RETURN v_email||p_employee_id;

EXCEPTION
    WHEN no_data_found THEN      --no hay un empleado con dicho email
        RETURN v_email;         --return solo nombre y apellido
END get_email;
-----

--funciones, procedimientos publicos
-----
PROCEDURE empleado
    (p_first_name      IN      employees.first_name%TYPE
    ,p_last_name       IN      employees.last_name%TYPE
    ,p_job_id          IN      employees.job_id%TYPE
    ,p_employee_id     IN      employees.employee_id%TYPE
                                DEFAULT NULL
    ,p_phone_number    IN      employees.phone_number%TYPE
                                DEFAULT NULL
    ,p_department_id   IN      employees.department_id%TYPE
                                DEFAULT NULL
    ,p_commission_pct  IN      employees.commission_pct%TYPE
                                DEFAULT NULL
    )
IS
    v_employee_id      employees.employee_id%TYPE:=p_employee_id;
    v_email             employees.email%TYPE;
    v_salary            employees.salary%TYPE;
BEGIN
    --si no se da employee_id se obtiene mediante la secuencia
    IF p_employee_id IS NULL THEN
        v_employee_id:=employees_seq.nextval;
    END IF;
    --una funcion privada no puede ser usada en sentencias sql
    v_email:=get_email(p_employee_id =>      v_employee_id
                      ,p_first_name=>      p_first_name
                      ,p_last_name =>      p_last_name
                      );
    CASE --solo si ambos parametros tienen valor
        WHEN p_department_id IS NOT NULL
            AND p_job_id IS NOT NULL
        THEN--se llama a la funcion get_salario
            v_salary:=get_salario
                (p_job_id          => p_job_id
                ,p_department_id  => p_department_id
                );
        ELSE
            v_salary:=NULL;
    END CASE;
    --alta empleado
    INSERT INTO employees
        (employee_id
        ,first_name
        ,last_name
        ,phone_number
        ,job_id
        ,department_id
        ,salary
        ,email
        ,hire_date
        ,manager_id
        ,commission_pct
        )

```



```

VALUES
    (v_employee_id
    ,p_first_name
    ,p_last_name
    ,p_phone_number
    ,p_job_id
    ,p_department_id
    ,v_salary
    ,v_email
    ,SYSDATE
    ,CASE
        WHEN p_department_id IS NOT NULL THEN
        (
            SELECT      manager_id
            FROM        departments
            WHERE      department_id=p_department_id
        )              --el jefe del empleado, el jefe del departamento
    END
    ,p_commission_pct
    );

END empleado;
-----
FUNCTION get_salario
    (p_job_id          IN      employees.job_id%TYPE
    ,p_department_id   IN      employees.department_id%TYPE
    )
    --PRE: p_job_id y p_department_id contienen valores existentes
    --POST: retorna el maximo de los salarios minimos
    -- de departamento o de trabajo
    -- en caso de no existir datos retorna null
    RETURN employees.salary%TYPE
IS
    v_mindep      employees.salary%TYPE;
    v_minjob      employees.salary%TYPE;
BEGIN
    --minimo salario del departamento dado
    SELECT MIN(salary)
    INTO   v_mindep
    FROM   employees
    WHERE  department_id=p_department_id;
    --minimo salario del trabajo dado
    SELECT MIN(salary)
    INTO   v_minjob
    FROM   employees
    WHERE  job_id=p_job_id;
    --se retorna el maximo de los minimos
    CASE
        WHEN v_minjob IS NULL THEN
            RETURN v_mindep;
        WHEN v_mindep IS NULL THEN
            RETURN v_minjob;
        WHEN v_mindep<v_minjob THEN
            RETURN v_minjob;
        ELSE
            RETURN v_mindep;
    END CASE;
END get_salario;
-----
END alta;
/

```

Colecciones

Ejercicio 1

```
DECLARE
```

```
type t_reg is record
    (employee_id employees.employee_id%type
    ,manager_id      employees.manager_id%type);
type t_tabla is table of t_reg;
v_emple t_tabla;
BEGIN
    select employee_id,manager_id
    bulk collect into v_emple
    from employees
    where department_id=60;

    forall i in v_emple.first..v_emple.last
        update employees
        set salary=( select salary*0.51
                    from employees
                    where employee_id=v_emple(i).manager_id)
        where employee_id=v_emple(i).employee_id
        and salary<( select salary*0.51
                    from employees
                    where employee_id=v_emple(i).manager_id);

END;
/
```

Disparadores

Ejercicio 1

```
CREATE OR REPLACE TRIGGER emp_lock
    BEFORE INSERT OR DELETE OR UPDATE ON employees
BEGIN
    hora_laboral;
END;
/
```

Ejercicio 2

```
CREATE OR REPLACE TRIGGER permiso
    BEFORE UPDATE OF salary ON employees
    FOR EACH ROW
DECLARE
    regis          quien_puede%ROWTYPE;
    demasiado     EXCEPTION;
BEGIN
    SELECT *
    INTO regis
    FROM quien_puede
    WHERE usuario = USER;
    regis.realizadas := regis.realizadas + 1;
    IF regis.realizadas > regis.veces THEN
        RAISE demasiado;
    ELSE
        UPDATE quien_puede
        SET realizadas = regis.realizadas
        WHERE usuario=USER;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RAISE_APPLICATION_ERROR (
            num=> -20100,
            msg=> 'Este usuario no puede modificar el salario.' );
    WHEN demasiado THEN
        RAISE_APPLICATION_ERROR (
            num=> -20101,
            msg=> 'Este usuario no puede modificar mas veces el salario.' );
END permiso;
```

```
/
```

Ejercicio 3

```
CREATE OR REPLACE TRIGGER anti_ddl
  BEFORE DDL ON HR.SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (
    num=> -20200,
    msg=> 'No se permite DDL.' );
END;
/
```

Cursores Variables

Ejercicio 1

```
CREATE OR REPLACE PACKAGE paql
AS
  TYPE Generico IS REF CURSOR;
  PROCEDURE Abre
    (cur_gene   OUT generico
    , opcion    IN PLS_INTEGER);
  PROCEDURE cuenta
    (cur_gene   IN OUT generico
    , opcion    IN PLS_INTEGER);
END paql;
/
```

```
CREATE OR REPLACE PACKAGE BODY paql AS
  PROCEDURE abre
    ( cur_gene   OUT generico
    , p_opcion   IN   PLS_INTEGER)
  IS
  BEGIN
    IF p_opcion = 1 THEN
      OPEN cur_gene FOR SELECT * FROM employees;
    ELSIF p_opcion = 2 THEN
      OPEN cur_gene FOR SELECT * FROM departments;
    ELSIF p_opcion = 3 THEN
      OPEN cur_gene FOR SELECT * FROM locations;
    END IF;
  END abre;

  PROCEDURE cuenta
    ( cur_gene IN OUT generico
    , p_opcion IN PLS_INTEGER )
  IS
    v_regi1   employees%ROWTYPE;
    v_regi2   departments%ROWTYPE;
    v_regi3   locations%ROWTYPE;
  BEGIN
    LOOP
      CASE p_opcion
        WHEN 1 THEN
          FETCH cur_gene INTO v_regi1;
        WHEN 2 THEN
          FETCH cur_gene INTO v_regi2;
        WHEN 3 THEN
          FETCH cur_gene INTO v_regi3;
        END CASE;
      EXIT WHEN cur_gene%NOTFOUND;
    END LOOP;
  END cuenta;
END;
```

```

        END LOOP;
        DBMS_OUTPUT.PUT_LINE ('El numero de filas es: '
                                || cur_gene%ROWCOUNT);
        CLOSE cur_gene;
    END cuenta;
END paql;
/

```

SQL Dinámico

Ejercicio 1

```

create or replace procedure ena_dis_tri(
    p_usuario    varchar2 default user,
    p_tabla      varchar2 default null,
    p_opcion     varchar2 default 'DISABLE')
is
    cursor cur is
        select table_name
        from   all_triggers
        where  table_owner=p_usuario
              and status != p_opcion||'D'      --solo los necesarios.
              and (p_tabla is null             --todas las tablas
                  or table_name=p_tabla);      -- o la tabla especificada

begin
    for reg in cur loop
        execute immediate
            'ALTER TABLE '||reg.table_name||' '||
            p_opcion||' ALL TRIGGERS';
    end loop;
end;
/

```

Ejercicio 2

```

DECLARE
    TYPE t_columnas IS TABLE OF VARCHAR2(30);
    nom_col          t_columnas :=
        t_columnas('First_Name','Last_Name','Salary');
    nom_tabla        VARCHAR2(30) := 'Nom_Tabla';
    sentencia        VARCHAR2(2000);
    sentencia2       VARCHAR2(2000);
    suma_total       NUMBER;
BEGIN
    sentencia := 'CREATE TABLE ' || nom_tabla || ' AS SELECT ';
    FOR i IN nom_col.FIRST .. nom_col.LAST LOOP
        sentencia := sentencia || nom_col(i) || ',';
        IF LOWER(nom_col(i)) = 'salary' THEN
            sentencia2 := 'SELECT SUM(salary) FROM ' || nom_tabla;
        END IF;
    END LOOP;
    sentencia := SUBSTR ( sentencia, 1, LENGTH( sentencia) -1)
        || ' FROM employees
           WHERE salary < (SELECT AVG(salary)
                           FROM employees)';

    EXECUTE IMMEDIATE sentencia;
    IF sentencia2 IS NOT NULL THEN
        EXECUTE IMMEDIATE sentencia2 INTO suma_total;
        DBMS_OUTPUT.PUT_LINE ( 'La suma de los salarios insertados es: ' ||
            suma_total );
    END IF;
END;
/

```