CURSORES

Introducción:

• Definimos un cursor como un área de memoria que se reserva para ejecutar sentencias

SQL y para almacenar información de procesamiento de las mismas

• Con los cursores se puede acceder al área de trabajo o contexto para procesar la

información recuperada en una consulta

• Existen dos tipos de cursores:

Explícitos.- Son definidos por el programador para acceder a las filas recuperadas

en una consulta.

Implícitos.- Generados automáticamente por Oracle al ejecutar una instrucción no

asociada a un cursor por el programador. Es decir, cuando se lanza una sentencia

SQL tiene un cursor asociado

• Para una mejor comprensión, podríamos decir que utilizar cursores explícitos es como

trabajar con ficheros virtuales en memoria, donde el fichero sería el cursor y cada una

de las filas recuperadas en la consulta sería un registro.

• Cuando lanzamos una sentencia SELECT esta debe devolver una única fila , de lo

contrario obtendremos la excepción TOO\_MANY\_ROWS. Pero en muchas ocasiones

existe la necesidad de trabajar con más de una fila, para ello utilizamos los cursores

explícitos

Declaración:

• Para declarar un cursor utilizaremos el siguiente formato:

DECLARE

tipo :=

CURSOR nom\_cur [(param1 { } [{ }valor], ...)]

var %TYPE DEFAULT

IS instrucción\_select;

nom\_cur.- Nombre del cursor a declarar.

param1.- Variable usada como parámetro en la instrucción SELECT. Puede aparecer

más de un parámetro. Se debe especificar su tipo de forma obligatoria, siendo opcional

la posibilidad de darle un valor por defecto.

Instrucción\_select.- Instrucción SELECT que especifica las distintas condiciones y

procedencias que deben cumplir las filas recuperadas para el cursor. En caso de que el

cursor incorpore algún parámetro, éste aparecerá en cualquier parte de dicha

instrucción.

Ejemplo:

CURSOR c\_vecinos IS SELECT \* FROM vecinos;

CURSOR c\_libros IS

SELECT nom\_autor, titulo, editorial FROM libros;

CURSOR c\_facturas IS

SELECT \* FROM facturas WHERE unidades > 70;

CURSOR c\_alumnos (nota notas.ev1 %TYPE) IS

SELECT nom\_alu, ape1\_alu, ape2\_alu, ev1

FROM alumnos a, notas n

WHERE a.num\_alu = n.num\_alu AND ev1>=nota;

• Una vez declarado el cursor, para poder acceder a sus datos debemos abrirlo. El

formato es:

BEGIN

OPEN nom\_cur [(valor1[, ...])];

END;

nom\_cur.- Nombre del cursor a abrir.

valor1.- Valor que tomará el parámetro. Es de obligada especificación únicamente en

el caso de que, al declarar el cursor, se especificará al menos un parámetro y no se le

hubiera asignado valor por defecto.

Ejemplo: OPEN c\_vecinos;

OPEN c\_libros;

OPEN c\_facturas;

OPEN c\_alumnos(7); • OPEN asigna de forma dinámica una area de memoria para que sea utilizada por el

cursor

• Obtiene lo valores de la sentencia select

• Posiciona el puntero del cursor en la primera fila de las filas recuperadas

• Una vez abierto el cursor, podemos acceder a las filas en él almacenadas, para ello

utilizaremos la instrucción FECHT, cuyo formato es:

BEGIN

.....

FETCH nom\_cur INTO {variable1[, variable2 ...]| registro};

....

END;

/

• La sentencia FECHT lee la fila marcada por el puntero, poniendo el valor de sus

columnas en las variables que siguen a la palabra clave INTO.

• La lectura de una fila mediante FECHT hace que el puntero pase a señalar la fila

siguiente, con lo que el acceso se realizará secuencialmente en el orden en que se

recuperaron las filas.

• Debe existir una correspondencia en cuanto a número y tipo entre las columnas de la

consulta y las variables de FECHT.

• A continuación ponemos un ejemplo de un cursor al que hay que pasarle un

parámetro. No debemos olvidar que para abrir un cursor con valores diferentes para

los parámetros, debemos cerrarlo antes de abrirlo si ya ha sido utilizado con

anterioridad.

Ejemplo:

CREATE OR REPLACE PROCEDURE ver\_emple\_apell(cadena VARCHAR2)

AS

cad varchar2(70);

CURSOR c\_emple (cad varchar2)IS SELECT ename, empno FROM emp

WHERE ename LIKE cad;

vr\_emple c\_emple%ROWTYPE;

BEGIN

cad:=upper(cadena);

cad:='%'||cad||'%';

OPEN c\_emple(cad);

FETCH c\_emple INTO vr\_emple;

WHILE (c\_emple%FOUND) LOOP

DBMS\_OUTPUT.PUT\_LINE(vr\_emple.empno||'\*'||vr\_emple.ename);

FETCH c\_emple INTO vr\_emple;

END LOOP;

DBMS\_OUTPUT.PUT\_LINE('NUMERO DE EMPLEADOS: '|| c\_emple

%ROWCOUNT);

CLOSE c\_emple;

END ver\_emple\_apell;

/

Ejemplo:

DECLARE

CURSOR c\_notas IS SELECT \* FROM notas;

num NUMBER(2);

nota1 NUMBER(4,2);

nota2 NUMBER(4,2);

BEGIN

OPEN c\_notas;

FETCH c\_notas INTO num, nota1, nota2;

...

END;

/

• Una vez finalizada el acceso a las filas el cursor debe cerrarse, para ello utilizaremos la

instrucción CLOSE , cuyo formato es:

CLOSE nom\_cur;

nom\_cur.- Nombre del cursor a cerrar.

Ejemplo: CLOSE c\_notas;

• Cada cursor tiene asociados cuatro atributos predefinidos por el sistema. Cuando se

trabaja con el cursor, estos atributos devuelven información sobre el resultado de la

sentencia ejecutada sobre el cursor. Los atributos son:

%ISOPEN

Retorna TRUE si el cursor está abierto y FALSE si está cerrado.

%FOUND

Devuelve FALSE si el último FECHT ejecutado no encontró más filas y en caso de

encontrar una fila devuelve TRUE. Después de abrir el cursor y antes del primer

FECHT retorna NULL.

EJEMPLO

BEGIN

OPEN cur1;

FETCH cur1 INTO v\_nom,v\_loc;

WHILE cur1%FOUND LOOP

Dbms\_output.put\_line(v\_nom|| v\_loc);

FETCH cur1 INTO v\_nom,v\_loc;

END LOOP;

CLOSE cur1;

END;

%NOTFOUND

Devuelve FALSE si FECHT encontró una fila y TRUE en caso de no encontrarla. Es

exactamente lo contrario del atributo anterior. Después de abrir el cursor y antes del

primer FECHT retorna NULL.

EJEMPLO:

….

EXIT WHEN CUR1%notfound; -- Se suele utilizar como salida de los bucles

%ROWCOUNT

Devuelve el número de filas accedidas por FECHT hasta el momento.

• Para utilizar estos atributos nombre\_cursor%vble

• El atributo %ISOPEN es realmente útil con cursores explícitos, ya que el control de

apertura y cierre de los cursores explícitos lo tiene el programador, mientras que los

cursores implícitos los controla el servidor de Oracle

BUCLE FOR PARA TRATAR CURSORES MEDIANTE REGISTROS.

• Cuando se trabaja con cursores y registros, existe una versión del bucle FOR específica

para el tratamiento de la información. Su formato es:

FOR nom\_registro IN nom\_cursor [(valor1[,...])] LOOP

sentencias;

END LOOP;

nom\_registro.- Nombre del registro que generará automáticamente el sistema para

el control del cursor asociado.

nom\_cursor.- Nombre del cursor que el bucle controlará su tratamiento.

valor1,....- Valores de los parámetros del cursor.

• La sentencia FOR se encarga de abrir el cursor y recupera la primera fila en su

ejecución. En sucesivas ejecuciones, va recuperando cada una de las filas del cursor.

Cuando ya no hay más filas, se cierra automáticamente el cursor y se termina la

ejecución del bucle.

• Los valores de las filas del cursor son accesibles a través de los campos del registro, el

cual, lo declara el sistema automáticamente y es del tipo nom\_cursor %ROWTYPE.

• Al registro no se le puede hacer referencia fuera del bucle FOR.

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT num\_alu, ev1 FROM notas

WHERE ev1 >= nota

FOR UPDATE;

BEGIN

FOR var\_notas IN c\_notas(7) LOOP

IF var\_notas.ev1>7 AND var\_notas.ev1<10 THEN

var\_notas.ev1 := var\_notas.ev1 + 1;

UPDATE notas SET ev1 = var\_notas.ev1

WHERE CURRENT OF c\_notas;

END IF;

END LOOP;

COMMIT;

END;

/! El bucle FOR se encarga generar el cursor, de abrirlo, posicionar el puntero en

todas y cada una de las filas del cursor y de cerrarlo después de procesar la última

fila.

Bucles For para Cursores

• Procesan filas de un cursor explícito.

• Cuando usamos un bucke For de estas características no tenemos que preocuparnos

de abrir,cerrar y realizar la instrucción fetch.

• Lo único que hacemos es declararlo

FOR id\_reg IN id\_cursor LOOP

Sentencias;

END LOOP;

Id\_reg: es una variable de tipo registro donde se almacenan los datos abiertos por el

cursor. No hace falta declararlo y su ambito es unicamente el bucle

Id\_cursor: Es el identificador del cursor que previamente hemos declarado

• El bucle se termina automáticamente cuando se recupera la última fila

EJEMPLO:

DECLARE

CURSOR MICUR IS SELECT \* FROM scott.salgrade WHERE hisal>=2000;

BEGIN

FOR REG\_EMP IN MICUR LOOP

DBMS\_OUTPUT.PUT\_LINE(to\_char(micur%rowcount,’99.’)||REG\_EMP.GRADE||':'||

reg\_emp.grade);

end loop;

end;

/

%FOUND %ISOPEN %NOTFOUND %ROWCOUNT

open Antes Invalid\_cursor F Invalid\_cursor Invalid

Después NULL T NULL 0

Primer

fetch

Antes NULL T NULL 0

Después T T F 1

Ultimo

fetch

Antes T T F N

Después T T T N

close Antes T T T N

Después Invalid\_cursor F invalid invalid

Atributos en cursores implícitos

• Se abre implícitamente un cursor cuando se procesa un comando SQL que no está

asociado a un cursor explícito

• El cursor implícito se llama SQL y dispone de los 4 atributos vistos anteriormente

SQL%NOTFOUND: TRUE si el último INSERT,UPDATE,DELETE o SELECT han fallado o

no hay filas

SQL%FOUND: TRUE si el último INSERT,UPDATE,DELETE o SELECT han afectado a una

o mas filas

SQL%ROWCOUNT devuelve el número de filas afectadas en el último

INSERT,UPDATE,DELETE o SELECT

SQL%ISOPEN siempre devuelve falso ya que oracle cierra automáticamente el cursor

después de hacer la sentencia SQL asociada

Diferencias del cursor implicito con un cursor explicito

• En el caso de SELECT INTO debe devolver una y solo una fila de lo contrario se

producen errores y se levantan automáticamente las excepciones:

NO\_DATA\_FOUND: Ninguna fila

TOO\_MANY\_ROWS: Más de una fila

Se detiene la ejecución del programa y se bifurca a la sección de EXCEPTION

• Esto no es aplicable a INSERT,UPDATE,DELETE, por eso tiene sentido SQL

%ROWCOUNT

Utilización de una subconsulta en un bucle for

• Si utilizamos una subconsulta en un bucle FOR de cursor no hace

falta declararse el cursor en la zona declarativa

EJEMPLO

BEGIN

FOR REG\_EMP IN (SELECT \* FROM scott.salgrade WHERE hisal>=2000) LOOP

DBMS\_OUTPUT.PUT\_LINE(REG\_EMP.GRADE||':'|| reg\_emp.grade);

end loop;

end;

/

alias en las columnas

• Cuando utilizamos variables registro declaradas del mismo tipo que el cursor o que la

tabla, los campos tiene el mismo nombre que las columnas correpondientes.

Ejemplo:

CURSOR C1 Select dept\_no, count(\*) from emple group by dept\_no;

OPEN C1;

FETCH c1 INTO v\_reg;

• Para visualizar una columna hacemos:

V\_reg.dept\_no

• Para visualizar lo que hemos contado no podríamos por eso debemos ponerles un alias

Ejemplo:

CURSOR C1 Select dept\_no, count(\*) total from emple group by dept\_no;

OPEN C1;

FETCH c1 INTO v\_reg;

• Para visualizar el total hacemos:

V\_reg.total

EJERCICIO:

Escribir un bloque PL/SQL que visualice el ename y hiredate(fecha

de alta) de la empresa, ordenados por fecha:

a) Mediante for de cursor

b) Mediante while

SOLUCIONES:

a) DECLARE

CURSOR c\_emple IS SELECT ename,hiredate\_from emple order by 2;

V\_reg\_emp c\_emp%rowtype;

Begin

For v\_reg\_emp IN c\_emple LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_reg\_emp.ename||v\_reg\_emp.hiredate)

END LOOP;

END;

/

b) DECLARE

CURSOR c\_emple IS SELECT ename,hiredate\_from emple order by 2;

V\_reg\_emp c\_emp%rowtype;

Begin

OPEN c\_emple;

FETCH c\_emple INTOv\_reg\_emp;

WHILE c\_emple%FOUND LOOP

DBMS\_OUTPUT.PUT\_LINE(v\_reg\_emp.ename||v\_reg\_emp.hiredate)

FETCH c\_emple INTOv\_reg\_emp;

END LOOP;

Close c\_emple;

END;

/

FOR UPDATE :

• A veces podemos requerir bloquear las filas devueltas por el cursor para realizar alguna

modificación en los datos.

• Para mantener bloqueadas las filas durante el proceso se utiliza FOR UPDATE en la

sentencia SELECT asociada al cursor

DECLARE

CURSOR nombre IS SELECT \* FROM notas FOR UPDATE;

• La clausula FOR UPDATE es la última de la sentencia SELECT incluso después de

ORDER BY si lo hubiese

• Los bloqueos de las filas se realizan en la apertura del cursor, no en las recuperaciones

• No se debe lanzar una sentencia Commit antes de cerrar un cursor que lleve la

cláusula FOR UPDATE ya que en ese caso se liberan los bloqueos realizados por el

cursor

Ejemplo:

DECLARE

CURSOR c\_dept IS SELECT \* FROM emp where deptno=10 FOR UPDATE;

BEGIN

OPEN c\_dept;

FETCH c\_dept INTO v\_reg;

WHILE c\_dept %FOUND LOOP

IF v\_reg.comm IS NULL THEN

UPDATE emp SET comm=300

WHERE empno=v\_reg.empno;

END IF;

FETCH c\_dept INTO v\_reg;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

WHERE CURRENT OF nom\_cursor;

nom\_cursor.- Nombre del cursor que recuperó las filas a modificar o borrar.

• Se utiliza WHERE CURRENT OF para hacer referencia a la fila que se está utilizando

actualmente sin necesidad de hacer referencia explícitamente con ROWID o algún

identificativo de la fila

• Para utilizar esta claúsula , la sentencia select debe incluir la cláusula FOR UPDATE

• No se puede utilizar WHERE CURRENT OF cuando la select accede a más de una tabla

• Sintaxis:

WHERE CURRENT OF id\_cursor;

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT nom\_alu, ape1\_alu, ape2\_alu, ev1

FROM alumnos a, notas n

WHERE a.num\_alu = n.num\_alu

AND ev1 >= nota FOR UPDATE;

nombre VARCHAR2(9);

ape1 nombre %TYPE;

ape2 nombre %TYPE;

nota1 NUMBER(4,2);

BEGIN

OPEN c\_notas(7);

FETCH c\_notas INTO nombre, ape1, ape2, nota1;

WHILE c\_notas %FOUND LOOP

IF nota1 > 7 AND nota1 < 10 THEN

nota1 := nota1 + 1;

UPDATE notas SET ev1 = nota1

WHERE CURRENT OF c\_notas;

END IF;

FETCH c\_notas INTO nombre, ape1, ape2, nota1;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

! ERROR, ya que el cursor está formado por más de una tabla y por tanto no

puede coincidir con la tabla notas que intentamos actualizar.

• La solución al caso anterior pasa por no utilizar la opción WHERE CURRENT OF

nom\_cursor y utilizar otra condición en la cláusula WHERE.

Ejemplo:

DECLARE

CURSOR c\_notas (nota notas.ev1 %TYPE) IS

SELECT n.num\_alu,nom\_alu,ape1\_alu,ape2\_alu,ev1

FROM alumnos a, notas n

WHERE a.num\_alu=n.num\_alu AND ev1>=nota;

num NUMBER(2);

nombre VARCHAR2(9);

ape1 nombre %TYPE;

ape2 nombre %TYPE;

nota1 NUMBER(4,2);

BEGIN

OPEN c\_notas(7);

FETCH c\_notas INTO num,nombre,ape1,ape2,nota1;

WHILE c\_notas %FOUND LOOP

IF nota1 > 7 AND nota1 < 10 THEN

nota1 := nota1 + 1;

UPDATE notas SET ev1 = nota1

WHERE num\_alu = num;

END IF;

FETCH c\_notas INTO num, nombre, ape1, ape2, nota1;

END LOOP;

CLOSE c\_notas;

COMMIT;

END;

/

Ejemplo correcto:

DECLARE

CURSOR c1 IS

SELECT \*

FROM emp

WHERE deptno=10 FOR UPDATE;

BEGIN

FOR REG IN C1 LOOP

IF REG.COMM IS NULL THEN

UPDATE EMP SET COMM=300 WHERE CURRENT OF C1;

END IF;

END LOOP;

END;

/

AMBITO DE LAS VARIABLES PL/SQL

• Cuando Existen bloques anidados es muy importante

determinar el ámbito de las variables, es decir, determinar la

región del programa

• Un identificador estará visible en el bloque en el que está

declarado y en todos los subbloques anidados, dentro de él.

• Un bloque no podrá referenciar identificadores que sean

declarados en bloques anidados en él.

• Cuando un bloque encuentra un identificador en alguna de

sus sentencias, busca en su zona declarativa la declaración

de este identificador. En caso de no encontrarlo pasará a

revisar la zona declarativa de su bloque inmediatamente

exterior hasta localizarlo.

DECLARE

A NUMBER;

BEGIN

AMBITO DE A

DECLARE

B NUMBER;

BEGIN

AMBITO DE B

END;

END;

CONTROL DE ERRORES. EXCEPCIONES

• En PL/SQl se denomina excepción a cualquier identificador que nos permita manejra un

error que se produzca durante la ejecución del programa, las excepciones pueden ser

definidas internamente por el sistema o definidas por el usuario.

• El tratamiento de excepciones de PL/SQL clarifica el código del programa al separarlo

del resto del proceso, definiendo las acciones a realizar en caso de producirse algún

error.

• Cuando se produce un error, se activa una excepción y la ejecución normal del

programa se detiene y finaliza el bloque donde se produzca el error, pasándose

el control a la sección de excepciones del bloque.

• En la parte de excepciones del programa existirán distintas rutinas, cada una dedicada

a procesar una o varias excepciones.

• Cada rutina comienza con una cláusula WHEN donde se especifica la excepción o

excepciones a las que se aplicará dicha rutina y después de ejecutarse la rutina

correspondiente a un error, el proceso continúa con la siguiente sentencia

del bloque contenedor.

• Si queremos que una misma rutina trate varias excepciones, detrás de la palabra

WHEN pondremos los nombres de esas excepciones, separados por el operador

relacional OR.

Sintaxis:

EXCEPTION

WHEN exception1 [OR exception2...]THEN

Instrucciones;

WHEN exception2 [OR exception2...]THEN

Instrucciones2;

WHEN OTHERS

Instrucciones3;

• La palabra EXCEPTION es obligatoria.

• Las excepciones no pueden aparecer en asignaciones de variables ni en sentencias SQL

• Como mucho podremos tener una claúsula OTHERS y en ella se filtran todos los

errores que no tengan su correspondiente manejador

• Podríamos tener en una zona de excepciones como único manejador la clausula

OTHERS de tal modo que cualquier error en ejecución se controla dentro de esta zona.

• Un error producido en la ejecución de la sección de excepciones no puede ser tratado

en ella. En este caso, se termina la ejecución del bloque con un error que se intentará

tratar en algunos de los bloques externos.

• En la declaración de variables pueden producirse excepciones al realizar inicializaciones

incorrectas. Las excepciones producidas en la parte de declaraciones no pueden ser

tratadas en el bloque, dando lugar a la finalización de la ejecución del bloque con un

error no tratado.

Ejemplo:

DECLARE

maximo CONSTANT NUMBER(4) := 10000;

! Finalización del programa con error.

Ejemplo:

DECLARE

comunidad NUMBER(7);

BEGIN

BEGIN

SELECT pts\_comunidad INTO comunidad

FROM vecinos

WHERE apellido1 LIKE 'SALVADOR';

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

SELECT pts\_comunidad INTO comunidad

FROM vecinos

WHERE apellido1 LIKE 'CASTRILLO';

END;

IF comunidad < 7000 THEN

comunidad := comunidad + 1000;

END IF;

INSERT INTO vecinos VALUES (8, 'A', 'ROBERTO', 'LOSADA',

'22-JUN-99', comunidad, comunidad/166.386);

COMMIT;

END;

/

• Si en la parte de excepciones de un bloque no existe rutina para el tratamiento de un

determinado error, o si no existe sección de excepciones, dicho error pasa sin tratar al

bloque contenedor, que intentará tratarlo en su sección de excepciones. De esta forma,

un error no tratado va pasando de bloque en bloque hasta que es tratado o hasta que

termina el programa con error.

• Existen dos tipos de excepciones:

Excepciones predefinidas.- Son activadas por el Servidor Oracle de forma

automática cuando se produce el error correspondiente.

Excepciones definidas por el usuario.- Deben de activarse explícitamente con la

sentencia RAISE.

Excepciones predefinidas

CURSOR\_ALREADY\_OPEN.- Se intenta abrir un cursor ya abierto.

DUP\_VAL\_ON\_INDEX.- Intento de insertar o actualizar, violando la condición de

unicidad de índice.

INVALID\_CURSOR.- Operación ilegal sobre un cursor.

INVALID\_NUMBER.- En una sentencia SQL se intentó realizar una conversión de

cadena de caracteres a número y la cadena contenía caracteres no numéricos.

LOGIN\_DENIED.- Se intentó iniciar una sesión de base de datos con nombre de

usuario o password no válidos.

NO\_DATA\_FOUND.- Una sentencia SELECT ... INTO no devuelve ninguna fila.

NOT\_LOGGED\_ON.- PL/SQL realiza una llamada a ORACLE y no existe conexión.

PROGRAM\_ERROR.- Error interno de PL/SQL.

STORAGE\_ERROR.- Se produce un error de memoria.

TIMEOUT\_ON\_RESOURCE.- Se termina el tiempo de espera por un recurso.

TOO\_MANY\_ROWS.- Una sentencia SELECT ... INTO devuelve más de una fila.

VALUE\_ERROR .- Se produjo un error aritmético o de conversión, un

truncamiento o la violación de una restricción.

ZERO\_DIVIDE.- Se intentó dividir por cero.

OTHERS.- En esta rutina se tratará cualquier excepción para la que no exista

tratamiento específico.

Hay alguna más pero estas son las más utilizadas y tenemos que tener en cuenta

que no es necesario declararlas en la sección DECLARE.

Excepciones definidas por el usuario

El programador puede definir sus propias excepciones, las cuales deben ser declaradas,

llamadas y activadas. Los formatos a utilizar son:

DECLARE

...

nom\_excep\_usu EXCEPTION; ! Declaración de excepción

...

BEGIN

...

RAISE nom\_excep\_usu; ! Llamada a la excepción

...

EXCEPTION

...

WHEN nom\_excep\_usu THEN ! Activar sentencias ejecutar

sentencias;

...

END;

/

Pasos a realizar:

Se requieren tres pasos para trabajar con excepciones de usuario:

Definición: se realiza en la zona de DECLARE con el siguiente formato:

nombre\_excepción EXCEPTION

Disparar o levantar la excepción mediante la orden raise: RAISE ;

Tratar la excepción en el apartado EXCEPTION: WHEN THEN ;

Ejemplo: De la tabla VECINOS incrementar en 1000 pesetas su columna de

PTS\_COMUNIDAD y la cantidad correspondiente a la columna de valor en euros,

salvo para las filas de fecha llegada nula, que se modificará únicamente la columna

FECHA\_LLEGADA con el valor de la fecha de sistema.

DECLARE

fecha\_nula EXCEPTION;

fecha\_no\_nula EXCEPTION;

CURSOR c\_vecinos IS

SELECT \* FROM vecinos FOR UPDATE;

BEGIN

FOR i IN c\_vecinos LOOP

BEGIN

IF i.fecha\_llegada IS NULL THEN

RAISE fecha\_nula;

END IF;

RAISE fecha\_no\_nula;

EXCEPTION

WHEN fecha\_nula THEN

UPDATE vecinos SET fecha\_llegada = SYSDATE

WHERE CURRENT OF c\_vecinos;

WHEN fecha\_no\_nula THEN

i.pts\_comunidad := i.pts\_comunidad + 1000;

UPDATE vecinos SET

pts\_comunidad = i.pts\_comunidad,

e\_comunidad = i.pts\_comunidad / 166.386

WHERE CURRENT OF c\_vecinos;

END;

END LOOP;

COMMIT;

END;

/

Otras excepciones

Existen otros errores internos de Oracle que no tienen asignada una excepción, sino

un código de error y un mensaje, a los que se accede mediante funciones SQLCODE

y SQLERRM.

Cuando se produce un error de estos se trasfiere directamente el control a la

sección EXCEPTION donde se tratara el error en la clausula WHEN OTHERS de la

siguiente forma:

WHEN OTHERS THEN DBMS\_OUTPUT.PUT\_LINE('Error'||SQLCODE||SQLERRM.)

RAISE\_APPLICATION\_ERROR

En el paquete DBMS\_STANDARD se incluye un procedimiento llamado

RAISE\_APPLICATION\_ERROR que nos sirve para levantar errores y definir mensajes

de error. Su formato es el siguiente:

RAISE\_APPLICATION\_ERROR(numero\_error,mensaje\_error);

Es importante saber que el numero de error esta comprendido entre -20000 y

-20999 y el mensaje es una cadena de caracteres de hasta 512 bytes.

Este procedimiento crea una excepción que solo puede ser tratada en WHEN

OTHERS.

Ejemplo:

CREATE or REPLACE PROCEDURE subir\_horas (emple NUMBER, horas\_subir

NUMBER)

IS

horas\_actuales NUMBER;

BEGIN

Select horas into horas\_actuales from empleados where id\_empleado=emple;

if horas\_actuales is NULL then

RAISE\_APPLICATION\_ERROR(-20010,'No tiene horas');

else

update empleados set horas=horas\_actuales + horas\_subir where

id\_empleado=emple;

end if;

End subir\_horas;

**Explicación del Documento sobre Paquetes en PL/SQL**

Los **paquetes en PL/SQL** permiten **agrupar** de manera lógica **procedimientos, funciones, variables, tipos de datos y cursores** en una sola unidad dentro de la base de datos. Esto mejora la organización y modularización del código, permitiendo ocultar la implementación interna y optimizando el rendimiento.

**¿Por qué el PACKAGE BODY no tiene BEGIN?**

* **Un paquete (PACKAGE) en PL/SQL es una colección de procedimientos, funciones, cursores y variables.**
* **El PACKAGE BODY solo implementa lo que se ha declarado en el PACKAGE.**
* **Como el PACKAGE BODY no ejecuta código directamente, no necesita un BEGIN.**
* **Solo los procedimientos y funciones dentro del PACKAGE BODY necesitan un BEGIN para ejecutar su lógica.**

**Ejemplo de estructura correcta de un paquete en PL/SQL**

**1️⃣ Especificación del paquete (PACKAGE)**

**Aquí solo declaramos los procedimientos y funciones, sin definir su lógica.**

**CREATE OR REPLACE PACKAGE pk AS**

**PROCEDURE datos\_cliente(p\_cod\_cliente IN clientes.cod\_cliente%TYPE);**

**END pk;**

**/**

**2️⃣ Implementación del paquete (PACKAGE BODY)**

**Aquí escribimos el código de los procedimientos.**

**CREATE OR REPLACE PACKAGE BODY pk IS -- No necesita BEGIN aquí**

**PROCEDURE datos\_cliente(p\_cod\_cliente IN clientes.cod\_cliente%TYPE) IS**

**-- Declaración de variables y cursor**

**TYPE t\_datos\_cliente IS RECORD (**

**cod\_cliente clientes.cod\_cliente%TYPE,**

**nombres clientes.nombres%TYPE,**

**distrito clientes.distrito%TYPE,**

**telefono clientes.telefono%TYPE**

**);**

**v\_aux t\_datos\_cliente;**

**CURSOR c\_datos\_cliente IS**

**SELECT cod\_cliente, nombres, distrito, telefono**

**FROM clientes**

**WHERE cod\_cliente = p\_cod\_cliente;**

**BEGIN -- Aquí empieza la ejecución del procedimiento**

**OPEN c\_datos\_cliente;**

**FETCH c\_datos\_cliente INTO v\_aux;**

**WHILE c\_datos\_cliente%FOUND LOOP**

**dbms\_output.put\_line(lpad('-', 70, '-'));**

**dbms\_output.put\_line('DATOS EMPLEADO' );**

**dbms\_output.put\_line(lpad('-', 70, '-'));**

**dbms\_output.put\_line(v\_aux.cod\_cliente||' ' || v\_aux.nombres);**

**dbms\_output.put\_line(v\_aux.distrito||' ' || v\_aux.telefono);**

**FETCH c\_datos\_cliente INTO v\_aux;**

**END LOOP;**

**CLOSE c\_datos\_cliente;**

**END datos\_cliente; -- Fin del procedimiento**

**END pk; -- Fin del paquete**

**/**

**✔ El PACKAGE BODY no tiene BEGIN porque solo define procedimientos y funciones.  
✔ Cada procedimiento y función dentro del PACKAGE BODY sí tiene su propio BEGIN y END.  
✔ El END pk; cierra el PACKAGE BODY.**

**💡 Si el PACKAGE BODY tuviera un BEGIN, significaría que queremos ejecutar código automáticamente al compilarlo, lo cual no es el propósito de un paquete. 🚀**

**📌 Conceptos Clave**

1. **División de un Paquete**
   * **Especificación (Obligatoria)**: Define los elementos públicos (funciones, procedimientos, variables, cursores) accesibles desde fuera del paquete.
   * **Cuerpo del Paquete (Opcional)**: Implementa las funciones y procedimientos declarados en la especificación. Contiene detalles internos ocultos a los usuarios externos.
2. **Beneficios de Usar Paquetes**
   * Modulariza el código, facilitando su mantenimiento.
   * Oculta detalles de implementación, protegiendo la integridad del código.
   * Mejora el rendimiento al cargar el paquete en memoria en la primera ejecución.
   * Permite la **sobrecarga de funciones** (varias funciones con el mismo nombre pero con diferentes parámetros).
3. **Cómo Crear un Paquete**
   * Se crean en **dos pasos**:
     1. **Cabecera del paquete** (PACKAGE).
     2. **Cuerpo del paquete** (PACKAGE BODY).

**📜 Reglas de Oro**

1. **Toda función o procedimiento debe estar declarado en la cabecera si se desea usar fuera del paquete.**
2. **Las variables y cursores pueden ser públicas (cabecera) o privadas (solo en el cuerpo).**
3. **Para llamar un procedimiento o función de un paquete, se usa:**

nombre\_paquete.nombre\_procedimiento(parámetros);

1. **Un paquete se ejecuta solo una vez por sesión cuando se usa por primera vez.**
2. **La sobrecarga de funciones permite reutilizar nombres de funciones con distintos parámetros.**
3. **Se puede usar una sección de inicialización en el paquete, que se ejecuta solo una vez.**
4. **Los paquetes de Oracle ofrecen funciones útiles (ej. DBMS\_OUTPUT.PUT\_LINE).**

**📌 Ejemplo de Creación de un Paquete en PL/SQL**

-- CABECERA DEL PAQUETE

CREATE OR REPLACE PACKAGE PK1 IS

-- Declaración de un procedimiento accesible desde fuera

PROCEDURE listar\_articulos(familia IN Articulos.cArtFml%TYPE);

END PK1;

/

-- CUERPO DEL PAQUETE

CREATE OR REPLACE PACKAGE BODY PK1 IS

PROCEDURE listar\_articulos(familia IN Articulos.cArtFml%TYPE) IS

codigo Articulos.cArtCdg%TYPE;

descripcion Articulos.cArtDsc%TYPE;

-- Cursor para seleccionar los artículos de una familia específica

CURSOR cArticulos IS

SELECT cArtCdg, cArtDsc FROM Articulos WHERE cArtFml = familia;

BEGIN

OPEN cArticulos;

LOOP

FETCH cArticulos INTO codigo, descripcion;

EXIT WHEN cArticulos%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(descripcion);

END LOOP;

CLOSE cArticulos;

END listar\_articulos;

END PK1;

/

Para ejecutar el procedimiento desde fuera del paquete:

BEGIN

PK1.listar\_articulos('ELEC'); -- Lista artículos de la familia "ELEC"

END;

/

**📌 Declaración de Cursores en Paquetes**

Podemos declarar cursores dentro de los paquetes de esta forma:

CREATE OR REPLACE PACKAGE empleados AS

CURSOR a RETURN empleado%ROWTYPE;

END empleados;

/

CREATE OR REPLACE PACKAGE BODY empleados AS

CURSOR a RETURN empleado%ROWTYPE IS

SELECT \* FROM empleado WHERE salario < 10000;

END empleados;

/

**📌 Paquetes de Oracle Útiles**

Algunos paquetes predefinidos en Oracle:

* **DBMS\_OUTPUT** → Imprimir mensajes en la consola (PUT\_LINE).
* **DBMS\_SQL** → Ejecutar SQL dinámicamente.
* **STANDARD** → Contiene funciones como TO\_CHAR y ABS.
* **DBMS\_STANDARD** → Manejo de errores con RAISE\_APPLICATION\_ERROR.

Con esta guía, puedes empezar a practicar la creación y uso de **paquetes en PL/SQL** con ejemplos claros y reglas esenciales. 🚀

✅ Cursor Público (en PACKAGE) → Lo pueden usar todos los procedimientos del paquete y se puede llamar desde fuera.

✅ Cursor Privado (en PACKAGE BODY) → Solo lo pueden usar los procedimientos dentro del PACKAGE BODY.

✅ Cursor Local (dentro de un PROCEDURE) → Solo lo puede usar ese procedimiento específico.

🔹 Si necesitas que varios procedimientos compartan un cursor, usa un cursor público o privado. Si el cursor es solo para un procedimiento, decláralo dentro de él. 🎯🚀

🔍 Ejemplo para Verlo Claramente

-- CABECERA DEL PAQUETE: DECLARAMOS UN CURSOR PÚBLICO

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- Cursor público: Puede ser usado en cualquier procedimiento dentro del paquete

CURSOR c\_publico IS

SELECT id\_empleado, nombre, salario

FROM empleados;

-- Procedimiento público

PROCEDURE listar\_empleados\_publico;

PROCEDURE listar\_empleados\_privado;

END pk\_empleados;

/

-- CUERPO DEL PAQUETE: IMPLEMENTAMOS LOS PROCEDIMIENTOS

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

-- Cursor privado: Puede ser usado por cualquier procedimiento dentro del PACKAGE BODY

CURSOR c\_privado IS

SELECT id\_empleado, nombre, salario

FROM empleados WHERE salario > 3000;

-- Procedimiento que usa el cursor público

PROCEDURE listar\_empleados\_publico IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_publico;

LOOP

FETCH c\_publico INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_publico%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('PUBLICO - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_publico;

END listar\_empleados\_publico;

-- Procedimiento que usa el cursor privado

PROCEDURE listar\_empleados\_privado IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_privado;

LOOP

FETCH c\_privado INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_privado%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('PRIVADO - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_privado;

END listar\_empleados\_privado;

-- Procedimiento con un cursor local (dentro del procedimiento)

PROCEDURE listar\_empleados\_local IS

CURSOR c\_local IS

SELECT id\_empleado, nombre, salario FROM empleados WHERE salario < 2000;

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_local;

LOOP

FETCH c\_local INTO v\_id, v\_nombre, v\_salario;

EXIT WHEN c\_local%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('LOCAL - ID: ' || v\_id || ' - Nombre: ' || v\_nombre || ' - Salario: ' || v\_salario);

END LOOP;

CLOSE c\_local;

END listar\_empleados\_local;

END pk\_empleados;

/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/////////////////

Aquí cursor con return

**🎯 Explicación del RETURN en Cursores de PL/SQL**

En PL/SQL, cuando declaramos un cursor **en la cabecera del paquete**, usamos RETURN para definir **qué tipo de datos devuelve el cursor**.

**📌 Sintaxis del RETURN en Cursores**

CURSOR nombre\_cursor RETURN tipo\_de\_registro IS

SELECT columna1, columna2 FROM tabla;

✅ RETURN **define la estructura de los datos** que devolverá el cursor.  
✅ Es útil cuando queremos **devolver un conjunto de filas** desde un procedimiento.

**✍ Ejemplo Completo con RETURN en un Cursor Público**

🔹 **Un procedimiento usa un cursor público declarado en la cabecera del paquete.**  
🔹 **Otro procedimiento usa un cursor privado dentro del PACKAGE BODY.**

**🟢 Paso 1: Crear el Paquete**

-- CABECERA DEL PAQUETE (PACKAGE)

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- 1️⃣ Definimos un tipo de registro basado en la tabla empleados

TYPE t\_empleado IS RECORD (

id empleados.id\_empleado%TYPE,

nombre empleados.nombre%TYPE,

salario empleados.salario%TYPE

);

-- 2️⃣ Definimos un cursor público con RETURN usando el tipo de registro

CURSOR c\_empleados RETURN t\_empleado IS

SELECT id\_empleado, nombre, salario FROM empleados;

-- Procedimientos

PROCEDURE listar\_empleados\_publico;

PROCEDURE buscar\_empleado(p\_id IN empleados.id\_empleado%TYPE);

END pk\_empleados;

/

🔍 **Explicación**  
✅ Creamos el **tipo de registro t\_empleado**, que representa una fila de la tabla empleados.  
✅ El **cursor c\_empleados usa RETURN t\_empleado**, indicando que devuelve registros con esa estructura.  
✅ Declaramos dos procedimientos que lo usarán.

**🟠 Paso 2: Crear el Cuerpo del Paquete**

-- CUERPO DEL PAQUETE (PACKAGE BODY)

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

-- Cursor privado: Solo se usa dentro del PACKAGE BODY

CURSOR c\_empleado\_unico(p\_id empleados.id\_empleado%TYPE) IS

SELECT id\_empleado, nombre, salario

FROM empleados WHERE id\_empleado = p\_id;

-- Procedimiento que usa el cursor público

PROCEDURE listar\_empleados\_publico IS

v\_empleado t\_empleado; -- Variable del tipo de registro del cursor

BEGIN

OPEN c\_empleados;

LOOP

FETCH c\_empleados INTO v\_empleado;

EXIT WHEN c\_empleados%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_empleado.id ||

' - Nombre: ' || v\_empleado.nombre ||

' - Salario: ' || v\_empleado.salario);

END LOOP;

CLOSE c\_empleados;

END listar\_empleados\_publico;

-- Procedimiento que usa el cursor privado

PROCEDURE buscar\_empleado(p\_id IN empleados.id\_empleado%TYPE) IS

v\_id empleados.id\_empleado%TYPE;

v\_nombre empleados.nombre%TYPE;

v\_salario empleados.salario%TYPE;

BEGIN

OPEN c\_empleado\_unico(p\_id);

FETCH c\_empleado\_unico INTO v\_id, v\_nombre, v\_salario;

IF c\_empleado\_unico%FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Empleado encontrado: ' || v\_id || ' - ' || v\_nombre || ' - ' || v\_salario);

ELSE

DBMS\_OUTPUT.PUT\_LINE('No se encontró ningún empleado con el ID: ' || p\_id);

END IF;

CLOSE c\_empleado\_unico;

END buscar\_empleado;

END pk\_empleados;

/

**🚀 Ejecutar los Procedimientos**

-- Llamar al procedimiento que usa el cursor público

BEGIN

pk\_empleados.listar\_empleados\_publico;

END;

/

-- Llamar al procedimiento que usa el cursor privado pasando un ID de empleado

BEGIN

pk\_empleados.buscar\_empleado(101);

END;

/

**📜 Explicación del Uso del RETURN en el Cursor Público**

| **Parte del Código** | **Explicación** |
| --- | --- |
| TYPE t\_empleado IS RECORD (...) | Crea un tipo de registro que almacena id, nombre y salario. |
| CURSOR c\_empleados RETURN t\_empleado IS ... | Declara un cursor que devuelve registros con la estructura de t\_empleado. |
| v\_empleado t\_empleado; | Declara una variable para almacenar registros del cursor. |
| FETCH c\_empleados INTO v\_empleado; | Extrae una fila y la guarda en la variable v\_empleado. |

**🎯 Resumen Final**

✅ **El RETURN en un cursor se usa para definir la estructura de los datos que devuelve.**  
✅ **Los cursores con RETURN permiten trabajar con registros de forma más clara y estructurada.**  
✅ **El cursor público (c\_empleados) se puede usar en cualquier procedimiento del paquete.**  
✅ **El cursor privado (c\_empleado\_unico) solo se puede usar dentro del PACKAGE BODY.**

🔹 **Con esto ya puedes usar cursores públicos con RETURN de forma correcta en tus paquetes PL/SQL. 🚀**

¡Buena pregunta, Bryan! 🔥

Si quieres almacenar **muchos registros** en un paquete, necesitas usar **colecciones** en lugar de una sola variable de tipo RECORD o %ROWTYPE.

**📌 ¿Cómo Almacenar Múltiples Registros con un Cursor en un Paquete?**

Tanto RECORD como %ROWTYPE **solo almacenan un registro a la vez**.  
Si necesitas manejar **varios registros**, debes usar una **tabla PL/SQL (colección)**.

**💡 Opciones para Almacenar Múltiples Registros**

| **Método** | **Descripción** | **Ejemplo** |
| --- | --- | --- |
| **Colecciones con TABLE y RECORD** | Se usa un tipo de tabla para almacenar múltiples registros personalizados. | TYPE t\_empleado\_tab IS TABLE OF t\_empleado; |
| **Colecciones con TABLE y %ROWTYPE** | Se usa una tabla PL/SQL con %ROWTYPE para guardar filas completas. | TYPE t\_empleado\_tab IS TABLE OF empleados%ROWTYPE; |

**🔥 Ejemplo 1: Cursor Público con RETURN y Colección en la Cabecera**

CREATE OR REPLACE PACKAGE pk\_empleados AS

-- 1️⃣ Definimos un tipo de registro con solo algunas columnas

TYPE t\_empleado IS RECORD (

id empleados.id\_empleado%TYPE,

nombre empleados.nombre%TYPE

);

-- 2️⃣ Definimos una tabla (colección) de empleados

TYPE t\_empleado\_tab IS TABLE OF t\_empleado;

-- 3️⃣ Declaramos un cursor público con `RETURN`

CURSOR c\_empleados RETURN t\_empleado IS

SELECT id\_empleado, nombre FROM empleados;

-- 4️⃣ Procedimiento que llena la colección con empleados

PROCEDURE listar\_empleados (p\_lista OUT t\_empleado\_tab);

END pk\_empleados;

/

**🎯 Ejemplo 2: Implementación en el PACKAGE BODY**

CREATE OR REPLACE PACKAGE BODY pk\_empleados AS

PROCEDURE listar\_empleados (p\_lista OUT t\_empleado\_tab) IS

v\_empleado t\_empleado;

v\_lista t\_empleado\_tab := t\_empleado\_tab(); -- Inicializa la colección

i PLS\_INTEGER := 1; -- Índice para la tabla

BEGIN

OPEN c\_empleados;

LOOP

FETCH c\_empleados INTO v\_empleado;

EXIT WHEN c\_empleados%NOTFOUND;

v\_lista.EXTEND; -- Aumenta el tamaño de la colección

v\_lista(i) := v\_empleado; -- Guarda el registro

i := i + 1;

END LOOP;

CLOSE c\_empleados;

-- Devuelve la lista completa

p\_lista := v\_lista;

END listar\_empleados;

END pk\_empleados;

/

**🛠 Ejemplo 3: Llamando al Procedimiento y Mostrando Múltiples Registros**

DECLARE

v\_empleados pk\_empleados.t\_empleado\_tab; -- Variable para almacenar la lista de empleados

BEGIN

pk\_empleados.listar\_empleados(v\_empleados);

-- Recorrer la colección e imprimir cada registro

FOR i IN 1 .. v\_empleados.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_empleados(i).id || ' - Nombre: ' || v\_empleados(i).nombre);

END LOOP;

END;

/

**🚀 Resumen**

✅ **Si quieres almacenar muchos registros**, usa **colecciones** (TABLE OF RECORD).  
✅ **Puedes devolver la colección desde un procedimiento** para usarla en PL/SQL.  
✅ **Las colecciones permiten recorrer y manejar múltiples registros fácilmente.**

📌 **Cuando tienes un cursor que devuelve muchas filas, la mejor práctica es almacenarlas en una colección para poder manipularlas.** 🔥

No puedes declarar el cursor en la **cabecera** del paquete y luego **no redeclararlo en el body**. Esto se debe a que **PL/SQL requiere que el cursor tenga su implementación en el cuerpo del paquete**.

Sin embargo, sí puedes **declararlo solo en la cabecera** si lo conviertes en un **cursor explícito con retorno de un registro (RETURN RECORD)**. Pero **no puedes usar RETURN T\_AUX (colección de registros) en la cabecera del paquete** porque los cursores solo pueden devolver **una fila a la vez**.

**🔥 SOLUCIÓN: Declarar solo en la cabecera y usar RETURN RECORD**

Si quieres que el cursor **solo esté en la cabecera del paquete** y que en el body solo lo abras, entonces usa **un cursor con RETURN RECORD** en la cabecera.

**📌 CABECERA DEL PAQUETE (Declarar solo una vez el cursor)**

CREATE OR REPLACE PACKAGE pk\_facturas AS

-- 📌 1️⃣ Tipo RECORD para una fila de datos

TYPE TD\_C\_VER\_FACTURAS IS RECORD (

COD\_FACTURA FACTURAS.COD\_FACTURA%TYPE,

FECHA\_EMISION FACTURAS.FECHA\_EMISION%TYPE,

COD\_CLIENTE FACTURAS.COD\_CLIENTE%TYPE,

COD\_PRODUCTO DETALLE\_FACTURAS.COD\_PRODUCTO%TYPE,

CANTIDAD DETALLE\_FACTURAS.CANTIDAD%TYPE,

SUBTOTAL DETALLE\_FACTURAS.SUBTOTAL%TYPE

);

-- 📌 2️⃣ Cursor que devuelve una sola fila (pero se puede recorrer con WHILE)

CURSOR c\_ver\_facturas(p\_cod\_factura FACTURAS.COD\_FACTURA%TYPE)

RETURN TD\_C\_VER\_FACTURAS;

-- 📌 3️⃣ Procedimientos

PROCEDURE ver\_facturas (p\_cod\_factura IN FACTURAS.COD\_FACTURA%TYPE);

PROCEDURE ver\_facturas;

PROCEDURE ver\_facturas (p\_cod\_cliente IN FACTURAS.COD\_CLIENTE%TYPE, p\_mostrar\_detalles IN BOOLEAN);

END pk\_facturas;

/

✅ **El cursor está solo en la cabecera**  
✅ **El cursor devuelve una fila (RETURN RECORD)**

**📌 IMPLEMENTACIÓN DEL PAQUETE (Solo se usa el cursor, no se redeclara)**

CREATE OR REPLACE PACKAGE BODY pk\_facturas AS

-- 📌 1️⃣ Procedimiento para ver una factura con WHILE

PROCEDURE ver\_facturas (p\_cod\_factura IN FACTURAS.COD\_FACTURA%TYPE) IS

v\_factura TD\_C\_VER\_FACTURAS; -- 📌 Variable para almacenar la fila del cursor

BEGIN

OPEN c\_ver\_facturas(p\_cod\_factura);

FETCH c\_ver\_facturas INTO v\_factura;

WHILE c\_ver\_facturas%FOUND LOOP

dbms\_output.put\_line('Factura: ' || v\_factura.cod\_factura);

dbms\_output.put\_line('Fecha: ' || v\_factura.fecha\_emision);

dbms\_output.put\_line('Cliente: ' || v\_factura.cod\_cliente);

dbms\_output.put\_line('Producto: ' || v\_factura.cod\_producto ||

', Cantidad: ' || v\_factura.cantidad ||

', Subtotal: ' || v\_factura.subtotal);

dbms\_output.put\_line('------------------------------------------------');

FETCH c\_ver\_facturas INTO v\_factura; -- Siguiente fila

END LOOP;

CLOSE c\_ver\_facturas;

END ver\_facturas;

-- 📌 2️⃣ Procedimiento para ver todas las facturas

PROCEDURE ver\_facturas IS

CURSOR c\_todas\_facturas IS

SELECT cod\_factura FROM facturas;

v\_cod\_factura FACTURAS.COD\_FACTURA%TYPE;

BEGIN

OPEN c\_todas\_facturas;

FETCH c\_todas\_facturas INTO v\_cod\_factura;

WHILE c\_todas\_facturas%FOUND LOOP

pk\_facturas.ver\_facturas(v\_cod\_factura);

FETCH c\_todas\_facturas INTO v\_cod\_factura;

END LOOP;

CLOSE c\_todas\_facturas;

END ver\_facturas;

-- 📌 3️⃣ Procedimiento para ver facturas de un cliente

PROCEDURE ver\_facturas (p\_cod\_cliente IN FACTURAS.COD\_CLIENTE%TYPE, p\_mostrar\_detalles IN BOOLEAN) IS

CURSOR c\_facturas\_cliente IS

SELECT cod\_factura FROM facturas WHERE cod\_cliente = p\_cod\_cliente;

v\_cod\_factura FACTURAS.COD\_FACTURA%TYPE;

BEGIN

OPEN c\_facturas\_cliente;

FETCH c\_facturas\_cliente INTO v\_cod\_factura;

WHILE c\_facturas\_cliente%FOUND LOOP

IF p\_mostrar\_detalles THEN

pk\_facturas.ver\_facturas(v\_cod\_factura);

ELSE

dbms\_output.put\_line('Factura: ' || v\_cod\_factura);

END IF;

FETCH c\_facturas\_cliente INTO v\_cod\_factura;

END LOOP;

CLOSE c\_facturas\_cliente;

END ver\_facturas;

END pk\_facturas;

/

**🔥 Beneficios de esta solución**

✅ **El cursor c\_ver\_facturas solo se declara en la cabecera.**  
✅ **No necesitas redeclararlo en el body, solo lo usas con OPEN, FETCH y CLOSE.**  
✅ **Puedes recorrerlo con WHILE o LOOP sin problemas.**  
✅ **No necesitas colecciones (TABLE OF RECORD), solo un RECORD normal.**

**⛔ Lo que NO se puede hacer**

❌ **No puedes hacer RETURN T\_AUX (una colección) en la cabecera del paquete.**  
❌ **No puedes declarar un cursor sin RETURN RECORD en la cabecera y evitar redeclararlo en el body.**  
❌ **No puedes devolver una colección directamente desde un cursor en un paquete.**

**🚀 Conclusión**

Si **quieres que el cursor solo esté en la cabecera**, usa **RETURN RECORD** y **un WHILE para recorrerlo** en el body.  
Este es el método **correcto y permitido en PL/SQL**. 🎯

Aquí tienes una tabla que resume las reglas sobre el uso de cursores en un paquete PL/SQL:

| **Regla** | **¿Se puede hacer?** | **Explicación** |
| --- | --- | --- |
| **Declarar un cursor en la cabecera y no redeclararlo en el body** | ✅ Sí, pero con RETURN RECORD | Si el cursor devuelve una sola fila a la vez (RETURN RECORD), puedes declararlo en la cabecera y usarlo en el body sin redeclararlo. |
| **Declarar un cursor en la cabecera con RETURN TABLE OF RECORD** | ❌ No | Los cursores en la cabecera solo pueden devolver **una fila a la vez** (RECORD), no una colección (TABLE OF RECORD). |
| **Usar OPEN cursor\_name, FETCH, CLOSE en los procedimientos del body** | ✅ Sí | Puedes recorrer los resultados usando un WHILE con %FOUND. |
| **Recorrer los resultados de un cursor sin usar un FETCH dentro de un loop** | ❌ No | Para leer cada fila del cursor, **debes hacer FETCH manualmente** dentro de un loop (WHILE o LOOP). |
| **Hacer BULK COLLECT INTO desde un cursor declarado en la cabecera** | ❌ No | BULK COLLECT INTO solo funciona con colecciones (TABLE OF RECORD), pero los cursores en la cabecera no pueden devolver colecciones. |
| **Devolver una colección (TABLE OF RECORD) desde un cursor en un paquete** | ❌ No | Solo se pueden devolver **una fila a la vez** (RETURN RECORD). Para colecciones, hay que definirlas y usarlas dentro del body. |
| **Llamar a un procedimiento que usa un cursor sin abrirlo manualmente** | ❌ No | Si defines un cursor en la cabecera, **debes abrirlo (OPEN) y recorrerlo (FETCH) manualmente** en el body. |
| **Usar cursores implícitos en procedimientos dentro de un paquete** | ✅ Sí | Puedes hacer SELECT INTO sin declarar un cursor, pero solo si esperas **una sola fila**. |

Con esto ya sabes exactamente qué se puede hacer y qué no dentro de un paquete PL/SQL con cursores. 🚀

* **Un cursor es un conjunto de filas** que se puede procesar **una fila a la vez**.
* **No puede almacenar colecciones**. El cursor solo maneja una fila en su memoria a la vez mientras se va recorriendo.
* **Sí puede recorrer colecciones**. Puedes usar un cursor para recorrer una colección, pero el cursor en sí solo procesa una fila a la vez, no almacena todas las filas de una colección de forma simultánea.

Esto es cierto tanto si usas el cursor dentro de un **paquete**, **procedimiento** o **función**, o si lo usas directamente en un bloque anónimo. El comportamiento es el mismo: el cursor recorre las filas de la base de datos una a una, y no puede almacenar todas esas filas en su memoria a la vez.

**Ejemplo más claro:**

Un paquete consta de los siguientes elementos:

• Especificación o cabecera: contiene las

declaraciones públicas (es decir, accesibles

desde cualquier parte de la aplicación) de sus

programas, tipos, constantes, variables, cursores,

excepciones, etc.

• Cuerpo: contiene los detalles de implementación

y declaraciones privadas, es decir, accesibles

solamente desde los objetos del paquete.

La sintaxis de la cabecera es la siguiente:

create [or replace] package nombre\_paquete as

<declaraciones públicas>

<especificaciones de subprogramas>

end nombre\_paquete;

La sintaxis del cuerpo sería la siguiente:

create [or replace] package body nombre\_paquete as

<declaraciones privadas>

<código de subprogramas>

[begin

<instrucciones iniciales>]

end nombre\_paquete;

Como podéis observar la cabecera se compila

independientemente del cuerpo. Os dejo un ejemplo

de paquete para que lo veáis más claro.

/\* Cabecera \*/

create or replace package busar\_emple as

TYPE t\_reg\_emple is RECORD

(num\_empleado emple.emp\_no%TYPE,

apellido emple.apellido%TYPE,

salario emple.salario%TYPE,

departamento emple.dept\_no%TYPE);

procedure ver\_por\_numero(v\_cod

emple.emp\_no%TYPE);

procedure ver\_por\_apellido(v\_ape

emple.apellido%TYPE);

function datos (v\_cod emple.emp\_no%TYPE)

return t\_reg\_emple;

end buscar\_emple;

/\* Cuerpo \*/

create or replace package body buscar\_emple as

vg\_emple t\_reg\_emple;

procedure ver\_emple; /\* procedimiento privado\*/

procedure ver\_por\_numero (v\_cod

emple.emp\_no%TYPE)

is

begin

select emp\_no, apellido, salario, dept\_no into

vg\_emple from emple where emp\_no=v\_cod;

ver\_emple;

end ver\_por\_numero;

procedure ver\_por\_apellido (v\_ape

emple.apellido%TYPE)

is

begin

select emp\_no,apellido,salario,dept\_no into

vg\_emple from emple where apellido=v\_apellido;

ver\_emple;

end ver\_por\_apellido;

function datos (v\_cod emple.emp\_no%TYPE)

return t\_reg\_emple

is

begin

select emp\_no,apellido,salario,dept\_no into

vg\_emple from emple where emp\_no=v\_cod;

procedure ver\_emple

is

begin

DBMS\_OUTPUT.PUT\_LINE(vg\_emple.num\_emple

ado || '\*' || vg\_emple.apellido || '\*' || vg\_emple.salario ||

'\*'|| vg\_emple.departamento);

end ver\_emple;

end buscar\_emple;

Como podéis ver este paquete nos permite buscar un

empleado de tres formas distintas y visualizar sus

datos.

Utilización de los objetos definidos en los

paquetes

Podemos utilizar los objetos definidos en los

paquetes básicamente de dos maneras distintas:

• Desde el mismo paquete: esto quiere decir que

cualquier objeto puede ser utilizado dentro del

paquete por otro objeto declarado en el mismo.

Para utilizar un objeto dentro del paquete tan sólo

tendríamos que llamarlo. La llamada sería algo

así: v\_emple :=buscar\_emple.datos(v\_n\_ape);

(como veis no utilizamos el execute ya que nos

encontramos dentro del paquete).

• Desde fuera del paquete: Podemos utilizar los

objetos de un paquete siempre y cuando haya

sido declarado en la especificación del mismo.

Para llamar a un objeto o procedimiento desde

fuera del paquete utilizaríamos la siguiente

notación: execute

nombre\_paquete.nombre\_procedimiento(lista de

parametros);

Declaración de cursores en paquetes

En los paquetes también podemos introducir cursores,

para ello debemos declararlo en la cabecera del

paquete indicando su nombre, los parámetros y tipo

devuelto. Para que lo veáis más claro os dejo un

ejemplo a continuación:

CREATE or REPLACE PACKAGE empleados AS

.....

CURSOR a RETURN empleado%ROWTYPE;

...

END empleados;

CREATE or REPLACE PACKAGE BODY empleados AS

....

CURSOR a RETURN empleado%ROWTYPE

SELECT \* FROM empleado WHERE salario < 10000;

....

END empleados;

DISPARADORES DE LA BASE DE DATOS (TRIGGER).

• Un disparador o trigger es un procedimiento PL/SQL asociado

a una tabla que Oracle ejecuta automáticamente cuando se

realiza una determinada operación sobre la tabla.

• Los trigger se asemejan a procedimientos y funciones en que

son bloque pl/sql nominados con sección declarativa,

ejecutable y de control de errores

• Sin embargo, un trigger es un conjunto de acciones que se

ejecutan implícitamente cuando ocurre un evento.

• Con un disparador, por ejemplo, se puede controlar que a

determinadas tablas no se acceda en determinados días, o

bien que no se realicen determinadas operaciones por

determinados usuarios.

• El acto de ejecutar un trigger se conoce como disparar y el

evento que lo provoca generalmente son DML sobre tablas.

• Con los disparadores en general podemos:

o Implementar restricciones complejas de integridad y

seguridad

o Prevenir transacciones erróneas

o Implementar reglas administrativas complejas

o Generar automáticamente valores derivados

• No se deben definir trigger para duplicar o reemplazar la

funcionalidad ya construida dentro de la bbdd. Por ejemplo, no

definir triggers para implementar las reglas de integridad

• El uso excesivo de triggers puede dar lugar a

interdependencias complejas o podría dificultar el

mantenimiento en aplicaciones largas.

• Cuidado con los efectos recursivos y en cascada

• Son tres los eventos que hacen que se dispare un trigger:

insert, delete, update, pudiendo elegir si se dispara antes o

después de producirse uno o varios de los eventos anteriores.

• Para poder crear, modificar o borrar disparadores es necesario

tener los privilegios necesarios:

CREATE TRIGGER  Crear un disparador en el esquema propio.

CREATE ANY TRIGGER  Crear un disparador en cualquier

esquema.

ALTER ANY TRIGGER  Modificar un disparador en cualquier

esquema.

DROP ANY TRIGGER  Eliminar cualquier disparador en

cualquier esquema.

• La sintaxis para crear disparadores de la base de datos es la

siguiente:

CREATE [OR REPLACE] TRIGGER

nombre {BEFORE| AFTER }

{DELETE| INSERT|UPDATE [OF

columnas...]} [OR {DELETE|

INSERT|UPDATE [OF columnas...]} ON

tabla

[REFERENCING OLD as alias NEW as alias]

[FOR EACH {STATEMENT|ROW[WHEN

CONDICIONN]}] /\*Aqui comienza el bloque

pl/sql\*/ [DECLARE

Declaraciones; Es opcional]

BEGIN

Sentencias;

END;

• Con las opciones BEFORE o AFTER se le indica si se dispara

antes o después de producirse el evento.

• Los eventos pueden ser: la inserción de nuevas filas, la

actualización de todas o algunas de las columnas y el borrado de

todas o algunas filas. Podemos especificar más de un evento a la

vez, separándolos con operadores de relación OR. En este caso

se lanza un mismo trigger por distintos eventos.

• Se puede especificar la columna o columnas que deben

modificarse para que se lance el disparador

• El nivel del disparador puede ser:

• A nivel de orden (por defecto): Se activa una sola vez por cada

orden, independientemente del número de filas afectadas. Se

puede incluir FOR EACH STATEMENT aunque no hace falta

• A nivel de fila: Se activa un vez por cada fila afectada. Se

debe incluir FOR EACH ROW

• La cláusula WHEN sirve para especificar la restricción del trigger,

es decir, se dispara solo cuando se cumple la condición. Esta

condición tiene las siguientes restricciones:

- Solo se puede utilizar con triggers a nivel de filas

- Debe ser una condición SQL no PL/SQL

- No puede incluir una consulta a ninguna tabla

NEW y OLD

• Con ellos podemos referenciar a los valores antes y después

de la actualización a nivel de fila

• Solo se puede utilizar a nivel de fila

• Se utiliza :new.columna :old.columna

• Ejemplo: if :new.columna<:old.columna

• Cuando hacemos DELETE :new.columna es NULL

• Cuando hacemos INSERT :old.columna es NULL

• Cuando hacemos UPDATE ambas tienen valor

• Cuando utilicemos new y old con la clausuala WHEN del

trigger no hace falta :

Ejemplo: when new.salario<old.salario

• Cuando queramos poner un alias a old o new

podremos hacerlo con REFERENCING

Ejemplo: Vamos a controlar qué usuarios modifican la

información de la tabla emp (insertan, borran o actualizan) y

cuándo lo hacen. El ejemplo ejecutarlo a nivel de fila y de

orden y ver las diferencias

1º. Creamos una tabla donde almacenaremos el usuario y el

momento de utilización de la tabla y la operación realizada

sobre ella.

CREATE TABLE controla\_usuarios (

usuario VARCHAR2(17),

operacion VARCHAR2(6),

fecha DATE);

2º. Creamos los disparadores, para lo que es necesario tener

el privilegio CREATE TRIGGER, si lo creamos en nuestro

esquema, o el de CREATE ANY TRIGGER, si se crea en

cualquier otro esquema.

CREATE TRIGGER actualiza\_emple

BEFORE UPDATE ON EMP

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_vecinos

VALUES (nom\_usuario, ‘UPDATE’, sysdate);

END;

/

CREATE TRIGGER inserta\_emp

BEFORE INSERT ON emp

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_emp

VALUES (nom\_usuario, ‘INSERT’, sysdate);

END;

/

CREATE OR REPLACE TRIGGER

borra\_emp BEFORE DELETE ON emp

DECLARE

nom\_usuario VARCHAR2(30);

BEGIN

nom\_usuario := USER;

INSERT INTO controla\_emp

VALUES (nom\_usuario, 'delete',

sysdate); END;

/

Una vez creados los disparadores, cada vez que un usuario

realiza cualquier tipo de modificación del contenido de la tabla

EMP, se genera una fila en la tabla CONTROLA\_EMP, que nos

permite conocer que usuario realizó una determinada

operación y en que momento.

TIPOS DE DISPARADORES Y ORDEN DE EJECUCIÓN

• Cuando existen varios disparadores se lanzaran en el

siguiente orden:

1º . Los asociados a BEFORE ... FOR EACH STATEMENT

2º Para cada fila los BEFORE .. FOR EACH ROW

3º Se realiza INSERT, DELETE o UPDATE bloqueando las filas

hasta que la transacción se confirme

4º Se ejecutan los AFTER .. FOR EACH ROW

7º AFTER FOR EACH STATEMENT

• Cuando se dispara un trigger este forma parte de la operación

de actualización que lo lanzó. Si el trigger falla, lo hace toda la

actualización y Oracle hará un Rollback de toda la actualización

•A veces en lugar de varios trigger se hace uno solo que se

puede lanzar por múltiples eventos

Multiples eventos de disparo

• Cuando un mismo trigger es disparado por múltiples eventos

esto se hace: Before delete or update ON emp

• Para distinguir posteriormente utilizamos los siguientes

predicados:

• INSERTING: Devuelve TRUE Si el evento que disparó el

trigger fue INSERT

• DELETING: Devuelve TRUE Si el evento que disparó el

trigger fue DELETE

• UPDATING: Devuelve TRUE Si el evento que disparó el

trigger fue UPDATE

• UPDATING(nombrecolumna): Devuelve TRUE Si el evento

que disparó el trigger fue update y la columna

especificada ha sido modificada

Begin

If INSERTING then

ELSEIF DELETING then

ELSEIF UPDATING(‘SALARIO’) THEN

ENDIF

Restricciones sobre los trigger

 En el bloque del trigger debemos tener las siguientes

consideraciones:

o No podemos escribir ninguna orden de control de

transacciones(COMMIT,ROLLBACK). Tampoco podrán ser

utilizadas estas sentencias en funciones o procedimientos

que sean llamados desde el trigger

Triggers Instead of

 Son un tipo especial de trigger que solo se puede definir sobre

vistas de la bbdd

 Se activan en lugar de la orden DML funcionando de manera

invisible

 No deja especificar los of en el update, se debe hacer con codigo

updating(‘columna’);

CREATE OR REPLACE TRIGGER nombre

INSTEAD OF {DELETE| INSERT|UPDATE

} [OR {DELETE| INSERT|UPDATE }

ON nombrevista

[referencing old as alias new as alias]

[for each row]

bloque pl/sql

 Todos los trigger INSTEAD OF son triggers a nivel de fila por lo

que esta cláusula no es obligatoria, ya que está implícita

 En estos trigger no se puede usar When

Ejemplo: Crear una vista con los empleados del departamento 20.

(EMP20)

A continuación, si intentamos actualizar o borrar algo en esta vista

no da problemas. Sin embargo, cuando intentamos insertar si da

problemas, ya que no introduzco el número de departamento y es

NOT NULL en la tabla (EMP)

Solución: Crear un disparador INSTEAD OF sobre DEPT20

…….

INSTEAD OF INSERT ON emp\_20

BEGIN

Insert into emp (empno,ename,job,mgr,hiredate,sal,comm.,deptno)

v a l u e s ( : n e w . e m p n o , : n e w . e n a m e , : n e w . j o b ,

7766,sysdate,:new.sal,null,20);

End;

/

sql> Insert into emp\_20 values (7000,’PETER’, ‘CLERK’,2000);

Otros triggers en la bbdd

 Estos trigger están disponibles a partir de la versión

8i o Triggers sobre sentencias DDL

o Triggers sobre eventos del sistema

 Estos trigger se definen a nivel de bbdd o a nivel de esquema.

Los definidos a nivel de bbdd se lanzan para todos los usuarios y

los triggers a nivel de esquema se lanzan para el usuario del

esquema del trigger.

Trigger sobre sentencias DDL

CREATE OR REPLACE TRIGGER nombre {BEFORE|AFTER}

{CREATE|ALTER|DROP}[OR {CREATE|ALTER|DROP] ON

{DATABASE|nombreusuario.SCHEMA} Pl/sql block;

 Cuando no se especifica nombredeusuario de toma por defecto el

esquema del usuario que está creando el trigger

Trigger sobre eventos del sistema

Estos trigger se disparan cuando se arranca o para la base de

datos, entra o sale un usuario, cuando se crea, modifica o

elimina un objeto, etc.

En Oracle para crear este tipo de trigger tenemos que tener

privilegios de Administer database trigger.

La sintaxis de este trigger seria la siguiente:

create [or replace] trigger nombre\_trigger

{ before | after } { <lista eventos de definición> | <lista

eventos del sistema>}

on { database | schema} [when (condición)]

<cuerpo del trigger (bloque PL/SQL)>

Donde la lista de eventos de definición puede tener uno o más

eventos DDL separados por or y la lista de eventos del

sistema igualmente separados por or.

Al asociar un disparador a un evento debemos indicar el

momento en que se dispare. A continuación os dejo una tabla

de evento, momento y cuando se dispararía para dejarlo todo

mas o menos claro.

db\_event1. Eventos de la Descripción

bbdd

AFTER SERVERERROR Se dispar después de que se produzca

un error

Información sobre triggers

DESC USER\_TRIGGERS

TRIGGER\_TYPE: Puede tener Before,After o Instead of

TRIGGERING\_EVENT: DML que dispara el Trigger

REFERENCING\_NAMES: Alias para old y New

TRIGGER\_BODY: El cuerpo del trigger

Operaciones sobre triggers

DROP TRIGGER nombre

Cuando eliminamos una tabla se eliminan todos los trigger

asociados a ella

AFTER LOGON

Se dispara después de que el usuario

se conecte a la bbdd

BEFORE LOGOFF Se dispara antes de que un usuario se

desconecte

AFTER STARTUP Se disparan después del arranque de

la bbdd. Este evento solo se puede

aplicar a nivel de Database

BEFORE SHUTDOWN Se dispara antes de la parada de la

bbdd. Este evento solo se puede

aplicar a nivel de Database

ALTER TRIGGER nombre\_trigger ENABLE|DISABLE

Para activar o desactivar el trigger

ALTER TABLE nombre\_trigger ENABLE|DISABLE ALL

TRIGGERS Activa o desactiva todos los triggers de la tabla

[Diccionario de datos para triggers](https://aulavirtual32.educa.madrid.org/ies.mariadezayas.majadahonda/mod/url/view.php?id=60070)

<https://www.jpromero.com/2011/07/informacion-de-los-disparadores-oracle.html>

DCL. CREACION DE USUARIOS

Desde la versión 12c debemos usar

alter session set "\_ORACLE\_SCRIPT"=true;

cada vez que vayamos a crear usuarios.

CREATE USER statement

The CREATE USER statement allows you to create a new database user which you can

use to log in to the Oracle database.

The basic syntax of the CREATE USER statement is as follows:

CREATE USER username

IDENTIFIED BY password

[DEFAULT TABLESPACE tablespace]

[QUOTA {size | UNLIMITED} ON tablespace]

[PROFILE profile]

[PASSWORD EXPIRE]

[ACCOUNT {LOCK | UNLOCK}];Code language: SQL (Structured Query Language) (sql)

In this syntax:

CREATE USER username

Specify the name of the user to be created.

IDENTIFIED BY password

Specify a password for the local user to use to log on to the database. Note that you can

create an external or global user, which is not covered in this tutorial.

DEFAULT TABLESPACE

Specify the tablespace of the objects such as tables and views that the user will create.

If you skip this clause, the user’s objects will be stored in the database default

tablespace if available, typically it is USERS tablespace; or the SYSTEM tablespace in case

there is no database default tablespace.

QUOTA

Specify the maximum space in the tablespace that the user can use. You can have

multiple QUOTA clauses, each for a tablespace.

Use UNLIMITED if you don’t want to restrict the size of the tablespace that the user can

use.

PROFILE profile

A user profile limits the database resources or password that the user cannot exceed.

You can assign a profile to a newly created user. If you skip this clause, Oracle will

assign the DEFAULT profile to the user.

PASSWORD EXPIRE

Use the PASSWORD EXPIRE if you want to force the user to change the password for the

first time the user logs in to the database.

ACCOUNT {LOCK | UNLOCK}

Use ACCOUNT LOCK if you want to lock the user and disable access. On the other hand,

specify ACCOUNT UNLOCK to unlock user and enable access.

To execute the CREATE USER statement, you must have the CREATE USER system

privilege. Once you create the new user, the privilege domain of the user will be empty.

Therefore, if you want the user to be able to login to the database, you should grant the

CREATE SESSION system privilege to the user.

Oracle CREATE USER examples

Let’s practice with the CREATE USER statement.

1) Using Oracle CREATE USER statement to create a new local user example

This example uses the CREATE USER statement to create a new local user named john

with the password abcd1234:

CREATE USER john IDENTIFIED BY abcd1234;

Code language: SQL (Structured Query Language) (sql)

Oracle issues the following output indicating that the user john has been created

successfully.

User JOHN created.

Code language: SQL (Structured Query Language) (sql)

To find a list of users with the OPEN status, you query the information from the

dba\_users:

SELECT

username,

default\_tablespace,

profile,

authentication\_type

FROM

dba\_users

WHERE

account\_status = 'OPEN';

Code language: SQL (Structured Query Language) (sql)

As you can see from the output, the user john has a default tablespace as USERS, profile

as DEFAULT, and log in to the database using a PASSWORD.

Tambien podremos modificar y borrar usuarios de la siguiente manera

ALTER USER Antonio QUOTA UNLIMITED ON usuarios

DROP USER usuario [CASCADE]

La opción CASCADE elimina los objetos del esquema del usuario antes de eliminar al

propio usuario. Es obligatorio si el esquema contiene objetos.´

GESTION DE PRIVILEGIOS

Los privilegios son permisos que damos a los usuarios para que puedan realizar ciertas

operaciones con la base de datos. En Oracle hay más de cien posibles privilegios. Se

dividen en:

• Privilegios de sistema. Son permisos para modificar el funcionamiento de la

base de datos. Son cambios, en definitiva, que afectan a todos los usuarios y

usuarias.

• Privilegios de objeto. Son permisos que se aplican a un objeto concreto de la

base de datos.

1. PRIVILEGIOS DEL SISTEMA

Se comentan algunos de los privilegios de sistema más importantes

Privilegio Significado

CREATE SESSION Permite al usuario conectar con la base de datos

RESTRICTED

SESSION

Permite al usuario establecer sesión con la base de datos en

caso de que la base de datos esté en modo restringido

mediante la instrucción:

ALTER SYSTEM ENABLE RESTRICTED

SESSION

Sólo los usuarios con este privilegio puede conectar con la

base de datos si ésta se encuentra en este modo.

Privilegio Significado

ALTER DATABASE Permite modificar la estructura de la base de datos

ALTER SYSTEM Permite modificar los parámetros y variables del sistema

CREATE TABLE Permite crear tablas. Incluye la posibilidad de borrarlas.

GRANT ANY

OBJECT PRIVILEGE

Permite conceder privilegios sobre objetos que no son del

usuario (pertenecen a otros usuarios) a terceros usuarios.

CREATE ANY

TABLE Permite crear tablas en otros esquemas de usuario

DROP ANY TABLE Permite borrar tablas de otros usuarios

SELECT ANY TABLE Permite seleccionar datos en tablas de otros usuarios

INSERT ANY TABLE Permite añadir datos en tablas de otros usuarios

UPDATE ANY

TABLE Permite eliminar datos en tablas de otros usuarios

DELETE ANY

TABLE Permite eliminar datos en tablas de otros usuarios

En la tabla anterior se ha hecho hincapié en los privilegios referidos a las tablas, para

otros objetos el funcionamiento es similar: igual que hay CREATE TABLE, se puede

usar CREATE VIEW para las vistas o INDEX, TRIGGER, PROCEDURE,

SEQUENCE, SYNONYM, TYPE,… y de esa forma podemos conceder privilegio de

creación de otros objetos. Lo mismo con el resto de operaciones

CONCEDER y REVOCAR PRIVILEGIOS

GRANT CREATE SESSION , ALTER SESSION, CREATE VIEW, CREATE VIEW TO ANTONIO;

2. GESTION DE PRIVILEGIOS SOBRE OBJETOS

Las instrucciones vistas anteriormente otorgan o quitan permisos generales, es decir

dictan qué operaciones, en general, puede realizar un usuario.

Los privilegios de objeto marcan qué operaciones le están permitidas a un usuario

realizar sobre el objeto de otros usuarios.

Los privilegios permiten definir el modo en que otros usuarios van a tratar los

objetos de otros usuarios.

GRANT {privilegio [(listaColumnas)] [,privilegio

[(listaColumnas)] [,…]] |

ALL [PRIVILEGES]}

ON [esquema.]objeto

TO {usuario | rol | PUBLIC} [,{usuario | rol | PUBLIC} [,…]]

[WITH GRANT OPTION]

La opción ALL concede todos los privilegios posibles sobre el objeto. Se pueden

asignar varios privilegios a la vez y también varios posibles usuarios. La opción WITH

GRANT OPTION permite al usuario al que se le conceden los privilegios, que pueda,

a su vez, conceder esos mismos privilegios a otro usuario.

Ejemplo de uso de GRANT con privilegios de objeto:

GRANT UPDATE, INSERT ON jsanchez.personas TO anozal;

En la siguiente tabla se enumeran los posibles privilegios que se pueden aplicar a un

determinado objeto:

REVOKE {privilegio1 [,privilegio2] [,…]] |

ALL [PRIVILEGES]}

ON [esquema.]objeto

FROM {usuario | rol | PUBLIC} [,{usuario | rol | PUBLIC} [,…]]

[CASCADE CONSTRAINTS]

CASCADE CONSTRAINTS elimina cualquier restricción que impida el

borrado del privilegio.

Sólo puede revocar los privilegios de objeto concedidos, el usuario que concedió

dichos privilegios.

3. OTORGA R PRIVILEGIOS A UN ROL

Los roles son privilegios aglutinados sobre un mismo nombre, bajo la idea

de que ese conjunto denote un uso habitual sobre la base de datos.

Gracias a los roles se facilita la asignación de privilegios a los usuarios. Un

usuario puede tener asignados varios roles y viceversa.

Se puede crear un rol con el comando:

CREATE ROLE rol

Nos permite asignar permisos a un usuario o a un rol.