# 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

| | | %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 |

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;
/
```

## 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;
/
```

# 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;
```

# Oracle CREATE SEQUENCE

**Summary**: in this tutorial, you will learn how to use the Oracle CREATE SEQUENCE statement to create a new sequence in Oracle.

## Introduction to Oracle CREATE SEQUENCE statement

The CREATE SEQUENCE statement allows you to create a new sequence in the database.

Here is the basic syntax of the CREATE SEQUENCE statement:

```
CREATE SEQUENCE schema_name.sequence_name
[INCREMENT BY interval]
[START WITH first_number]
[MAXVALUE max_value | NOMAXVALUE]
[MINVALUE min_value | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE cache_size | NOCACHE]
[ORDER | NOORDER];
```

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

### CREATE SEQUENCE

Specify the name of the sequence after the CREATE SEQUENCE keywords. If you want to create a sequence in a specific schema, you can specify the schema name in along with the sequence name.

### INCREMENT BY

Specify the interval between sequence numbers after the INCREMENT BY keyword.

The interval can have less than 28 digits. It also must be less than MAXVALUE - MINVALUE.

If the interval is positive, the sequence is ascending e.g., 1,2,3,...

If the interval is negative, the sequence is descending e.g., -1, -2, -3 ...

The default value of interval is 1.

### START WITH

Specify the first number in the sequence.

The default value of the first number is the minimum value of the sequence for an ascending sequence and maximum value of the sequence for a descending sequence.

## MAXVALUE

Specify the maximum value of the sequence.

The max_value must be equal to or greater than first_number specify after the START WITH keywords.

## NOMAXVALUE

Use NOMAXVALUE to denote a maximum value of 10^27 for an ascending sequence or -1 for a descending sequence. Oracle uses this option as the default.

## MINVALUE

Specify the minimum value of the sequence.

The min_value must be less than or equal to the first_number and must be less than max_value.

## NOMINVALUE

Use NOMINVALUE to indicate a minimum value of 1 for an ascending sequence or -10^26 for a descending sequence. This is the default.

## CYCLE

Use CYCLE to allow the sequence to generate value after it reaches the limit, min value for a descending sequence and max value for an ascending sequence.

When an ascending sequence reaches its maximum value, it generates the minimum value.

On the other hand, when a descending sequence reaches its minimum value, it generates the maximum value.

## NOCYCLE

Use NOCYCLE if you want the sequence to stop generating the next value when it reaches its limit. This is the default.

### CACHE

Specify the number of sequence values that Oracle will preallocate and keep in the memory for faster access.

The minimum of the cache size is 2. The maximum value of the cache size is based on this formula:

```
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)
```
Code language: SQL (Structured Query Language) (sql)

In case of a system failure event, you will lose all cached sequence values that have not been used in committed SQL statements.

### ORDER

Use ORDER to ensure that Oracle will generate the sequence numbers in order of request.

This option is useful if you are using Oracle Real Application Clusters. When you are using exclusive mode, then Oracle will always generate sequence numbers in order.

### NOORDER

Use NOORDER if you do not want to ensure Oracle to generate sequence numbers in order of request. This option is the default.

## Oracle CREATE SEQUENCE statement examples

Let's take some example of using sequences.

### 1) Basic Oracle Sequence example

The following statement creates an ascending sequence called id_seq, starting from 10, incrementing by 10, minimum value 10, maximum value 100. The sequence returns 10 once it reaches 100 because of the CYCLE option.

```
CREATE SEQUENCE id_seq
  INCREMENT BY 10
  START WITH 10
  MINVALUE 10
  MAXVALUE 100
  CYCLE
  CACHE 2;
```
Code language: SQL (Structured Query Language) (sql)

To get the next value of the sequence, you use the NEXTVAL pseudo-column:

```
SELECT
  id_seq.NEXTVAL
FROM
  dual;
```
Code language: SQL (Structured Query Language) (sql)

Here is the output:

```
  NEXTVAL
---------
       10
```
Code language: SQL (Structured Query Language) (sql)

To get the current value of the sequence, you use the CURRVAL pseudo-column:

```
SELECT
  id_seq.CURRVAL
FROM
  dual;
```
Code language: SQL (Structured Query Language) (sql)

The current value is 10:

```
  CURRVAL
---------
       10
```
Code language: SQL (Structured Query Language) (sql)

This SELECT statement uses the id_seq.NEXTVAL value repeatedly:

```
SELECT
  id_seq.NEXTVAL
FROM
  dual
CONNECT BY level <= 9;
```
Code language: SQL (Structured Query Language) (sql)

Here is the output:

```
  NEXTVAL
---------
       20
       30
       40
       50
       60
       70
       80
       90
      100

9 rows selected
```
Code language: SQL (Structured Query Language) (sql)

Because we set the CYCLE option for the id_seq sequence, the next value of the id_seq will be 10:

```
SELECT id_seq.NEXTVAL FROM dual;
```
Code language: SQL (Structured Query Language) (sql)

And here is the output:

```
  NEXTVAL
---------
       10
```
Code language: SQL (Structured Query Language) (sql)

## 2) Using a sequence in a table column example

Prior Oracle 12c, you can associate a sequence indirectly with a table column only at the insert time.

See the following example.

First, create a new table called tasks:

```
CREATE TABLE tasks(
  id NUMBER PRIMARY KEY,
  title VARCHAR2(255) NOT NULL
);
```
Code language: SQL (Structured Query Language) (sql)

Second, create a sequence for the id column of the tasks table:

```
CREATE SEQUENCE task_id_seq;
```
Code language: SQL (Structured Query Language) (sql)

Third, insert data into the tasks table:

```
INSERT INTO tasks(id, title)
VALUES(task_id_seq.NEXTVAL, 'Create Sequence in Oracle');

INSERT INTO tasks(id, title)
VALUES(task_id_seq.NEXTVAL, 'Examine Sequence Values');
```
Code language: SQL (Structured Query Language) (sql)

Finally, query data from the tasks table:

```
SELECT
  id, title
FROM
  tasks;
```
Code language: SQL (Structured Query Language) (sql)

| ID | TITLE |
|---|---|
| 1 | Create Sequence in Oracle |
| 2 | Examine Sequence Values |

In this example, the tasks table has no direct association with the task_id_seq sequence.

## 3) Using the sequence via the identity column example

From Oracle 12c, you can associate a sequence with a table column via the identity column.

First, drop the tasks table:

```
DROP TABLE tasks;
Code language: SQL (Structured Query Language) (sql)
```

Second, recreate the tasks table using the identity column for the id column:

```
CREATE TABLE tasks(
   id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
   title VARCHAR2(255) NOT NULL
);
Code language: SQL (Structured Query Language) (sql)
```

Behind the scenes, Oracle creates a sequence that associates with the id column of the tasks table.

Because Oracle generated the sequence automatically for the id column, in your Oracle instance, the name of the sequence may be different.

| | COLUMN_NAME | DATA_TYPE | NULLABLE | DATA_DEFAULT | COLUMN_ID | COMMENTS |
|---|---|---|---|---|---|---|
| 1 | ID | NUMBER | No | "OT"."ISEQ$$_74366".nextval | 1 | (null) |
| 2 | TITLE | VARCHAR2(255 BYTE) | No | (null) | 2 | (null) |

Oracle uses the sys.idnseq$ to store the link between the table and the sequence.

This query returns the association of the tasks table and ISEQ$$_74366 sequence:

```
SELECT
   a.name AS table_name,
   b.name AS sequence_name
FROM
   sys.idnseq$ c
   JOIN obj$ a ON c.obj# = a.obj#
   JOIN obj$ b ON c.seqobj# = b.obj#
WHERE
   a.name = 'TASKS';
Code language: SQL (Structured Query Language) (sql)
```

Third, insert some rows into the tasks table:

```
INSERT INTO tasks(title)
VALUES('Learn Oracle identity column in 12c');
```

```
INSERT INTO tasks(title)
VALUES('Verify contents of the tasks table');
```
Code language: SQL (Structured Query Language) (sql)

Finally, query data from the tasks table:

```
SELECT
   id, title
FROM
   tasks;
```
Code language: SQL (Structured Query Language) (sql)

| ID | TITLE |
|---|---|
| 1 | Learn Oracle identity column in 12c |
| 2 | Verify contents of the tasks table |

In this tutorial, you have learned how to use the Oracle CREATE SEQUENCE statement to create a new sequence in the database.

# SQL EN PL/SQL

- Pl/SQL soporta el lenguaje de consulta SQL
    - (DQl: sentencia select)
    - DML: Insert, Update y Delete
    - Sentencias para el control de transacciones: Commit, Rollback y Savepoint

- Hay que tener en cuenta que un bloque PL/SQL no conforma una transacción por sí solo, tendremos que controlar nosotros este aspecto con las sentencias SQL correspondientes
- PL/SQL no soporta el lenguaje de definición de datos (DDl) Create table, Alter table o Drop table
- Tampoco el lenguaje de control de datos DCL . Grant y Revoke

## SENTENCIA SELECT

- La sentencia Select en un bloque Pl/SQL difiere levemente de una sentencia select en SQL
- Debe incluir una cláusula nueva que es la cláusula INTO para recoger los valores devueltos por la consulta
- Estos valores deben almacenarse en variables declaradas en el bloque Pl/SQL

```
SELECT lista_select INTO
 { VARIABLE[,VARIABLE...] | registro}
FROM tabla
[WHERE condiciones];
```

- Es muy importante saber que una sentencia select en un bloque Pl/sql debe devolver como resultado una única fila de lo contrario generaría un error.

Ejemplo: Guardar en las variables v_id y v_nombre el nombre y el código de departamento del departamento 10

```
Declare
V_id dept.deptno%TYPE;
V_nombre dept.dname%TYPE;
Begin
Select deptno,dname into v_id, v_nombre from dept
Where deptno =10;
End;
/
```

- La claúsula INTO debe contener tantas variables como columnas seleccionemos en la Select

Ejemplo2:

```
'
variable gmax number;
begin
select max(sal) into :gmax from scott.emp where deptno= & g_dept;
end;
/
print gmax
```

- Como vemos hemos utilizado variables de intercambio (g_dept). Estas variables son propias de SQL*PLUS y no hace falta declararlas.

- Para utilizarlas en un código Pl/SQl pondremos &g_dept

- Estas variables se pierden en caso de producirse un error, por tanto conviene salvarlas

  declare

  var1_num1:= & g_dept;

  Si fuese alfanumerica sería:

  var1_car1:= '& g_dept';

- Las variables de SQl de enlace (gmax) se deben utilizar dentro del código con :gmax

## SENTENCIA INSERT

- Vamos a buscar el máximo código de departamento. Lo incrementamos en 1 e insertamos el departamento EDUCACIÓN

```
declare
v_max SCOTT.dept.deptno%TYPE;
begin
select max(SCOTT.DEPT.DEPTNO) into V_MAX from scott.DEPT;
V_MAX:=V_MAX +1;
INSERT INTO SCOTT.DEPT(DEPTNO,DNAME) VALUES (V_MAX,'EDUCACION');
end;
/
```

- A diferencia de la sentencia select, podemos insertar más de una fila a la vez sin que se produzca ningún error.

## SENTENCIA UPDATE

- Para modificar datos en la base de datos

Ejemplo: Introducir un código de un empleado y a ese empleado incrementarle su salario en un 10%

```
Declare
V_incremento number(2) := 10;
Begin
Update scott.emp set scott.emp.sal= scott.emp.sal + (scott.emp.sal*v_incremento/ 100) where scott.emp.empno= &v_emple;
End;
/
```

- Al igual que Insert podemos actualizar más de una fila cada vez

## SENTENCIA DELETE

- La incorpora Pl/SQL para sus bloques

Ejemplo: eliminar el departamento que deseemos

```
Begin
Delete from emp
Where deptno=& v_deptno;
End;
/
```

- También se puede eliminar más de una fila cada vez

sql en pl/sql

# VARIABLES NO PL/SQL. VARIABLES DE ENLACE (HOST) Y VARIABLES DE SUSTITUCIÓN

- En un bloque Pl/SQL podemos hacer referencia a variables que no han sido declaradas en el propio bloque
- Se llaman variables de enlace (Bind) o variables de Host
- Con estas variables podemos transferir valores de ejecución a bloques Pl/SQL
- Para declarar una variables de este tipo lo podemos hacer:
    - Desde el entorno SQL* PLUS
    - Antes de codificar el bloque Pl/SQL en un fichero

- Pondremos:

  VARIABLE identificador tipo
- Las variables de tipo numérico no necesitan que se especifique la longitud pero las alfanuméricas si.

```
Variable g_dias NUMBER
Begin
        :g_dias:=7;
End;
/
```

- Una vez declarada podemos utilizarla dentro de un bloque PL/SQL excepto dentro de los bloques procedimientos y funciones
- Para hacer referencia a una de estas variables dentro del bloque hay que preceder la variable con : para indicar que no es una variable declarada en el bloque
- Todas las sentencias Pl/SQL finalizan con ;
- Después del bloque hay que poner / para que se ejecute el bloque
- El valor de una variable declarada en SQL*PLUS se mantiene cuando finaliza la ejecución del bloque. De esta forma podríamos visualizar el valor de esa variable fuera del bloque. Para ello utilizo PRINT (comando SQL*PLUS)

```
PRINT g_dias
```

- Además también podemos utilizar variables de sustitución de tipo & simple y && utilizando los comandos SQL*PLUS DEFINE y ACCEPT (en lugar de VARIABLE)

_____

```
DECLARE
        V_numero NUMBER(7) ;
BEGIN
        V_numero :=& n_dias ;
        Dbms_output.put_line(V_numero) ;
END ;
/
```