Lenguaje SQL: procedimental PL/SQL

BASES DE DATOS

Profesor: Héctor Gómez Gauchía

Materiales: Héctor Gómez Gauchía, Mercedes García Merayo

PL/SQL

- Los sistemas gestores de bases de datos incorporan utilidades que amplían el lenguaje **SQL con elementos de la programación estructurada.**
- La razón es que hay diversas acciones en la base de datos para las que SQL no es suficiente.
- **PL/SQL** es el lenguaje procedimental implementado por el precompilador de Oracle.
- El código PL/SQL puede almacenarse en la propia base de datos o en archivos externos.

Conceptos Básicos

- **Bloque PL/SQL**

  Fragmento de código que puede ser interpretado por Oracle.
- **Procedimiento**

  Programa PL/SQL almacenado en la base de datos que puede ser ejecutado invocándolo con su nombre.
- **Función**

  Programa PL/SQL que a partir de unos datos de entrada obtiene un resultado. Una función puede ser utilizada desde cualquier otro programa PL/SQL e incluso desde una instrucción SQL.
- **Trigger** (disparador)

  Programa PL/SQL que se ejecuta automáticamente cuando se produce un determinado suceso en un objeto de la base de datos.

Bloque

- **Declaraciones (DECLARE)**

  Define e inicializa las variables, constantes, excepciones de usuario y cursores utilizados en el bloque.
- **Comandos ejecutables (BEGIN)**

  Sentencias para manipular la base de datos y los datos del programa.
- **Tratamiento de excepciones (EXCEPTION)**

  Para indicar las acciones a realizar en caso de error.
- **Final del bloque**

  La palabra **END** da fin al bloque.

Bloque

```
[DECLARE
declaraciones ]
BEGIN
instrucciones ejecutables
[EXCEPTION
instrucciones de manejo de errores ]
END
```

Bloque

- Comentarios pueden ser de dos tipos:
  - **Comentarios de varias líneas.**

    Comienzan con /* y terminan con */ .
  - **Comentarios de línea simple.**

    Utilizan los signos -- (doble guión). El texto a la derecha de los guiones se considera comentario (el de la izquierda no) .

```
DECLARE
    v NUMBER := 17;
BEGIN
/* Este es un comentario que
ocupa varias líneas */
v:=v*2; -- este sólo ocupa esta línea
END;
```

Declaración de variables

- Las variables se declaran en el apartado **DECLARE del bloque.**

```
DECLARE
identificador [CONSTANT] tipoDatos
  [:=valorIni];
[siguienteVariable…]
```

```
DECLARE
pi CONSTANT NUMBER(9,7):=3.1415927;
radio NUMBER(5);
area NUMBER(14,2) := 23.12;
```

Declaración de variables

- El operador := sirve para asignar valores a una variable. Si no se inicializa la variable, esta contendrá el valor NULL.
- La palabra **CONSTANT** indica que la variable no puede ser modificada.
- Los identificadores de Oracle deben empezar por letra y continuar con letras, números o guiones bajos (_), el signo de dólar ($) y la almohadilla (#).

No se pueden declarar varias variables en la misma instrucción.

Declaración de variables

- Las variables PL/SQL pueden pertenecer a uno de los siguientes tipos de datos:

CHAR(n)	Texto de anchura fija
VARCHAR2(n)	Texto de anchura variable
NUMBER[(p[,s])]	Número. Opcionalmente puede indicar el tamaño del número (p) y el número de decimales (s)
DATE	Almacena fechas
INTEGER	Enteros de -32768 a 32767
BOOLEAN	Permite almacenar los valores TRUE (verdadero) y FALSE (falso)

Declaración de variables

- **expresión %TYPE**

  Se utiliza para dar a una variable el mismo tipo de otra variable o el tipo de una columna de una tabla de la base de datos.

  ```
  identificador variable tabla.columna%TYPE;
  ```

  ```
  nom personas.nombre%TYPE;
  precio NUMBER(9,2);
  precio_iva precio%TYPE;
  ```

- Hay que tener en cuenta que las variables declaradas en un bloque concreto, son eliminadas cuando éste acaba.

Paquetes estándar

- Oracle incorpora una serie de paquetes para ser utilizados dentro del código PL/SQL.
  - El paquete DBMS_RANDOM contiene funciones para utilizar número aleatorios. La más útil es RANDOM que devuelve un número entero (positivo o negativo) aleatorio (y muy grande).

```
Entre 1 y 10:
MOD(ABS(DBMS_RANDOM.RANDOM),10)+1
Entre 30 y 50:
MOD(ABS(DBMS_RANDOM.RANDOM),21)+30
```

Salida por pantalla

- El paquete DBMS_OUTPUT sirve para utilizar funciones y procedimientos de escritura como PUT_LINE o NEW_LINE().

```
DECLARE
a NUMBER := 17;
BEGIN
      DBMS_OUTPUT.PUT_LINE(a);
END;
```

Instrucción SELECT INTO

```
SELECT listaDeCampos
INTO listaDeVariables
FROM tabla
[JOIN ...]
[WHERE condición]
```

- PL/SQL admite el uso de un SELECT que permite almacenar valores en variables.

- La cláusula `INTO` es obligatoria en PL/SQL y además la expresión `SELECT` ***sólo puede devolver una única fila; de otro modo, ocurre un error.***

Instrucción SELECT INTO

```
DECLARE
   v_salario NUMBER(9,2);
   v_nombre VARCHAR2(50);

BEGIN
   SELECT salario,nombre INTO v_salario, v_nombre
   FROM empleados WHERE dni='12344';
   DBSM_OUTPUT.PUT_LINE
   ('El incremento será de ' || v_salario*0.2 || 'euros');
END;
```

Instrucciones DML

- Se permiten las instrucciones **INSERT, UPDATE y DELETE** con la ventaja de que en PL/SQL pueden utilizar variables.

- Es posible insertar los datos recuperados mediante una consulta **SELECT**

```
INSERT INTO Prestamo
SELECT * FROM Nuevos_Prestamos
```

Instrucciones DML y de transacción

- Es posible eliminar/actualizar mediante consultas anidadas

```
DELETE
FROM Clientes WHERE Clientes.NumPrestamo NOT IN
(SELECT NumPrestamo FROM Prestamo)

UPDATE Prestamo
SET sucursal= 'Centro'
WHERE sucursal IN
                (SELECT sucursal
                FROM Sucursales_Cerradas)
```

Instrucciones de control de flujo

- **Instrucción IF**

```
IF condición THEN
instrucciones
END IF;
```

- **Instrucción IF-THEN-ELSE**

```
IF condición THEN
 instrucciones
ELSE
 instrucciones
END IF;
```

Instrucciones de control de flujo

- Instrucción IF-THEN-ELSEIF

```
IF condición1 THEN
 instrucciones1
ELSIF condición2 THEN
 instrucciones2
[ELSIF.... ]
[ELSE
 instruccionesElse]
END IF;
```

Instrucciones de control de flujo

- Instrucción CASE

```
CASE selector
WHEN expresion1 THEN resultado1
WHEN expresion2 THEN resultado2
...
[ELSE resultadoElse]
END;
```

Instrucciones de control de flujo

```
texto:= CASE actitud
          WHEN 'A' THEN 'Muy buena'
          WHEN 'B' THEN 'Buena'
          WHEN 'C' THEN 'Normal'
          WHEN 'D' THEN 'Mala'
           ELSE 'Desconocida'
           END;
aprobado:= CASE
          WHEN actitud='A' AND nota>=4 THEN TRUE
          WHEN nota>=5 AND (actitud='B' OR actitud='C') THEN
             TRUE
          WHEN nota>=7 THEN TRUE
          ELSE FALSE
           END;
```

Bucles

- **LOOP**

```
LOOP
instrucciones
...
EXIT [WHEN condición]
END LOOP;
```

- **WHILE**

```
WHILE condición LOOP
instrucciones
END LOOP;
```

Bucles

- **FOR**

```
FOR contador IN [REVERSE]
  valorBajo..valorAlto
instrucciones
END LOOP;
```

- La variable contador no tiene que estar declarada en el `DECLARE`, es declarada automáticamente en el propio `FOR` y se elimina cuando este finaliza.

Cursores

- Los cursores representan consultas **SELECT** que devuelven más de un resultado y que permiten el acceso a cada fila de dicha consulta.

- El cursor tiene un puntero señalando a una de las filas del **SELECT**.

- Se puede recorrer el cursor haciendo que el puntero se mueva por las filas.

Cursores

- **Declaración del cursor**

- **Apertura del cursor**

  Tras abrir el cursor, el puntero del cursor señalará a la primera fila (si la hay) .

- **Procesamiento del cursor**

  La instrucción FETCH permite recorrer el cursor registro a registro hasta que el puntero llegue al final.

- **Cierre del cursor**

Declaración de Cursores

- Se declaran en el apartado **DECLARE**

```
CURSOR nombre IS sentenciaSELECT;
```

```
CURSOR cursorProvincias IS
SELECT p.nombre, SUM(poblacion) AS tpoblacion
FROM localidades l  JOIN provincias p USING (n_provincia)
GROUP BY p.nombre;
```

Nombre	N_provincia	poblacion
Majadahonda	01	125000
Toledo	02	275000
Getafe	01	285000
Illescas	02	32000
Santiago	03	115000

N_provincia	Nombre
01	Madrid
02	Toledo
03	A Coruña

→

Nombre	tpoblacion
Madrid	410000
Toledo	307000
A Coruña	115000

Cursor

Apertura de Cursores

```
OPEN nombre_cursor;
```

- Reserva memoria suficiente para el cursor
- Ejecuta la sentencia **SELECT** a la que se refiere el cursor
- Coloca el puntero de recorrido de registros en la primera fila
- Si la sentencia **SELECT** del cursor no devuelve registros, Oracle no devolverá una excepción, hasta intentar leer no sabremos si hay resultados o no.

Instrucción FETCH

```
FETCH cursor INTO listaDeVariables;
```

- Esta instrucción almacena el contenido de la fila a la que apunta actualmente el puntero en la lista de variables indicada.
- La lista de variables debe tener el mismo tipo y número que las columnas representadas en él.
- Tras esta instrucción el puntero de registros avanza a la siguiente fila (si la hay).

```
FETCH cursorProvincias INTO (v_nombre, v_poblacion);
```

Instrucción FETCH

- Una instrucción FETCH lee una sola fila y su contenido lo almacena en variables. Se usa siempre dentro de bucles a fin de poder leer todas las filas del cursor

```
LOOP
FETCH cursorProvincias INTO (v_nombre,
  v_poblacion);
EXIT WHEN...
instrucciones--proceso de los datos del
  cursor
END LOOP;
```

Cierre del cursor

```
CLOSE cursor;
```

- Al cerrar el cursor se libera la memoria que ocupa y se impide su procesamiento.
- Tras cerrar el cursor se podría abrir de nuevo.

Atributos de los cursores

- `%ISOPEN`

  Devuelve verdadero si el cursor ya está abierto.

- `%NOTFOUND`

  Devuelve verdadero si la última instrucción `FETCH` no devolvió ningún valor.

- `%FOUND`

  Opuesto al anterior, devuelve verdadero si el último `FETCH` devolvió una fila.

- `%ROWCOUNT`

  Indica el número de filas que se han recorrido en el cursor. Inicialmente vale cero.

Atributos de los cursores

```
DECLARE
    CURSOR cursorProvincias IS
     SELECT p.nombre, SUM(poblacion) AS tpoblacion
     FROM LOCALIDADES l JOIN PROVINCIAS p USING (n_provincia)
     GROUP BY p.nombre;
    v_nombre PROVINCIAS.nombre%TYPE;
    v_poblacion LOCALIDADES.poblacion%TYPE;

BEGIN
    OPEN cursorProvincias;
    LOOP
        FETCH cursorProvincias INTO v_nombre, v_poblacion;
        EXIT WHEN cursorProvincias%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_nombre || ',' || v_poblacion);
    END LOOP;
    CLOSE cursorProvincias;
END;
```

Registros

- Tipo de datos que se compone de datos más simples.
- Cada fila de una tabla o vista se puede interpretar como un registro.

```
TYPE nombreTipoRegistro IS RECORD (
campo1 tipoCampo1 [:= valorInicial],
campo2 tipoCampo2 [:= valorInicial],
...
campoN tipoCampoN [:= valorInicial]);

nombreVariableDeRegistro nombreTipoRegistro;
```

Registros

- `%ROWTYPE`

  Al declarar registros, se puede utilizar el modificador `%ROWTYPE` que sirve para asignar a un registro la estructura de una tabla *o cursor*.

```
v_registro nombreTabla%ROWTYPE;
```

- `v_registro` es un registro que constará de los atributos y tipos que las columnas de la tabla `nombreTabla`.

Registros

```
DECLARE
    CURSOR cursorProvincias IS
      SELECT p.nombre, SUM(poblacion) AS tpoblacion
      FROM LOCALIDADES l  JOIN PROVINCIAS p USING (n_provincia)
      GROUP BY p.nombre;
      rProvincias cursorProvincias%ROWTYPE;

BEGIN
   OPEN cursorProvincias;
   LOOP
     FETCH cursorProvincias INTO rProvincias;
     EXIT WHEN cursorProvincias%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE
        (rProvincias.nombre || ',' ||    rProvincias.tpoblacion);
     END LOOP;
    CLOSE cursorProvincias;
END;
```

Recorrido de cursores

- La forma más habitual de recorrer todas las filas de un cursor es un bucle **FOR** que se encarga de
  - Abrir un cursor antes de empezar el bucle.
  - Recorrer todas las filas del cursor y almacenar el contenido de cada fila en una variable de registro.
  - La variable de registro utilizada en el bucle **FOR** se crea al inicio del bucle y se elimina cuando éste finaliza.
  - Cerrar el cursor cuando finaliza el **FOR**.

Recorrido de los cursores

```
DECLARE
    CURSOR cursorProvincias IS
      SELECT p.nombre, SUM(poblacion) AS tpoblacion
      FROM LOCALIDADES l JOIN PROVINCIAS p USING (n_provincia)
      GROUP BY p.nombre;

BEGIN
       FOR rProvincias IN cursorProvincias LOOP
          DBMS_OUTPUT.PUT_LINE
             (rProvincias.nombre || ',' || rProvincias .tpoblacion);
       END LOOP;
END;
```

Actualización con cursores

- Se pueden realizar actualizaciones de registros sobre el cursor que se está recorriendo.
- Se deben bloquear los registros del cursor a fin de detener otros procesos que también desearan modificar los datos.

```
CURSOR ...
SELECT...
FOR UPDATE [OF campo] [NOWAIT]
```

- `NOWAIT` para que el programa no se quede esperando en caso de que la tabla esté bloqueada por otro usuario.

Actualización con cursores

```
DECLARE
CURSOR c_emp IS
SELECT id_emp, nombre, n_departamento, salario
FROM empleados, departamentos
WHERE empleados.id_dep=departamentos.id_dep
AND empleados.id_dep=80
FOR UPDATE OF salario NOWAIT;
BEGIN
FOR r_emp IN c_emp LOOP
IF r_emp.salario<1500 THEN
UPDATE empleados SET salario = salario *1.30
WHERE CURRENT OF c_emp;
END LOOP;
END;
```

Procedimientos

- Los procedimientos son compilados y almacenados en la base de datos.
- Gracias a ellos se consigue una reutilización eficiente del código.

```
CREATE [OR REPLACE] PROCEDURE nombreProcedimiento
[(parámetro1 [modo] tipoDatos
[,parámetro2 [modo] tipoDatos [,...]])]
IS
secciónDeDeclaraciones
BEGIN
instrucciones
[EXCEPTION
controlDeExcepciones]
END;
```

Procedimientos

- Al declarar cada parámetro se indica el tipo de los mismos, pero no su tamaño; es decir sería `VARCHAR y no VARCHAR(50).`
- La opción modo permite elegir si el parámetro es de tipo `IN, OUT o IN OUT.`
- No se utiliza la palabra `DECLARE` para indicar el inicio de las declaraciones. La sección de declaraciones figura tras las palabras `IS`.

Parámetros

- **Parámetros IN.**

  El procedimiento recibe una copia del valor o variable que se utiliza como parámetro al llamar al procedimiento.

- **Parámetros OUT.**

  Sólo pueden ser variables y no pueden tener un valor por defecto. Son variables sin declarar que se envían al procedimiento de modo que si en el procedimiento cambian su valor, ese valor permanece en ellas cuando el procedimiento termina.

- **Parámetros IN OUT.**

  Son una mezcla de los dos anteriores. Se trata de variables declaradas anteriormente cuyo valor puede ser utilizado por el procedimiento que, además, puede almacenar un valor en ellas. No se las puede asignar un valor por defecto.

- Si no se indica modo alguno, se usa **IN**.

Procedimientos

```
CREATE OR REPLACE PROCEDURE consultarEmpresa
(v_Nombre VARCHAR2, v_CIF OUT VARCHAR2, v_dir OUT VARCHAR2)
IS
BEGIN
    SELECT cif, direccion INTO v_CIF, v_dir
    FROM EMPRESAS
    WHERE nombre like '%'||v_nombre||'%';
   EXCEPTION
      WHEN NO_DATA_FOUND  THEN
        DBMS_OUTPUT.PUT_LINE('No se encontraron datos');
      WHEN TOO_MANY_ROWS  THEN
         DBMS_OUTPUT.PUT_LINE ('Hay más de una fila con esos datos');
END;
```

Procedimientos

Los procedimientos no pueden leer los valores que posean las variables **OUT**, solo escribir en ellas. Si se necesitan ambas cosas es cuando hay que declararlas con **IN OUT.**

La llamada al procedimiento anterior podría ser:

```
DECLARE
    v_c VARCHAR2(50);
    v_d VARCHAR2(50);
BEGIN
consultarEmpresa('Hernández',v_c,v_d);
DBMS_OUTPUT.PUT_LINE(v_c);
DBMS_OUTPUT.PUT_LINE(v_d);
```

Funciones

- Las funciones son un tipo especial de procedimiento que *devuelven un valor.*

```
CREATE [OR REPLACE] FUNCTION nombreFunción
[(parámetro1 [modo] tipoDatos
[,parámetro2 [modo] tipoDatos [,...]])]
RETURN tipoDeDatos
IS
secciónDeDeclaraciones
BEGIN
instrucciones
[EXCEPTION
controlDeExcepciones]
END;
```

Funciones

```
CREATE OR REPLACE FUNCTION cuadrado
(x NUMBER)
RETURN NUMBER
IS
BEGIN
  RETURN x*x;
END;
```

En PL/SQL la recursividad está permitida.

```
CREATE FUNCTION Factorial
(n NUMBER)
IS
BEGIN
  IF (n<=1) THEN
    RETURN 1
  ELSE
    RETURN n * Factorial(n-1);
  END IF;
END;
```

Excepciones

- Una excepción es un evento que causa que la ejecución de un programa finalice.
- Las excepciones se deben a:
  - Un error detectado por Oracle.
  - Provocadas por el desarrollador en el programa
- Las excepciones se pueden capturar a fin de que el programa controle la finalización.
- La captura se realiza utilizando el bloque **`EXCEPTION`** que es el bloque que está justo antes del **`END`** del bloque.

Excepciones

```
DECLARE
sección de declaraciones
BEGIN
instrucciones
EXCEPTION
  WHEN excepción1 [OR excepción2 ...] THEN
       instrucciones
  [WHEN excepción3 [OR...] THEN
  instrucciones]
  [WHEN OTHERS THEN
  instrucciones]
END;
```

Excepciones

- Categorías de excepciones
  - Definidas internamente: El Sistema las lanza automáticamente. Tienen asociado un código de error, pero no un nombre, a menos que sea definido.

```
PRAGMA EXCEPTION_INIT(exception_name, error_code)
```

```
PRAGMA EXCEPTION_INIT(out_memory, -27102)
```

Excepciones

- Categorías de excepciones
  - Predefinidas: Es una excepción interna con un nombre asociado.
  - Son lanzadas automáticamente y se pueden capturar mediante el nombre asociado.

Excepciones predefinidas

CASE_NOT_FOUND	ORA-06592	Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE
CURSOR_ALREADY_OPEN	ORA-06511	Se intenta abrir un cursor que ya se había abierto
DUP_VAL_ON_INDEX	ORA-00001	Se intentó añadir una fila que provoca que un índice único repita valores
INVALID_CURSOR	ORA-01001	Se realizó una operación ilegal sobre un cursor
INVALID_NUMBER	ORA-01722	Falla la conversión de carácter a número
NO_DATA_FOUND	ORA-01403	El SELECT de fila única no devolvió valores
ROWTYPE_MISMATCH	ORA-06504	Hay incompatibilidad de tipos entre el cursor y las variables a las que se intentan asignar sus valores
TOO_MANY_ROWS	ORA-01422	El SELECT de fila única devuelve más de una fila
VALUE_ERROR	ORA-06502	Hay un error aritmético, de conversión, de redondeo o de tamaño en una operación
ZERO_DIVIDE	ORA-01476	Se intenta dividir entre el número cero.

Excepciones

- Categorías de excepciones
  - Definidas por el usuario: Puedes declarer tus propias excepciones asociadas a comportamientos que pueden ser incorrectos desde un punto de vista de la lógica del negocio

```
nombreExcepcion EXCEPTION;
```

```
Fondos_Insuficientes EXCEPTION;
```

  - Deben ser lanzados explícitamente en el código.

```
RAISE nombreExcepcion;
```

```
RAISE Fondos_Insuficientes;
```

Excepciones

```
DECLARE v_ratio NUMBER(3,1);

BEGIN
        SELECT precio/ganancia INTO v_ratio FROM stocks
        WHERE id= 'XYZ';

        INSERT INTO stats (id, ratio) VALUES ('XYZ', v_ratio);
        COMMIT;

EXCEPTION

     WHEN ZERO_DIVIDE THEN
        INSERT INTO stats (id, ratio) VALUES ('XYZ', NULL);
        COMMIT;
     WHEN OTHERS THEN
         ROLLBACK;
END;
```

Excepciones

```
CREATE OR REPLACE PROCEDURE consultarExistencias
(v_unidades NUMBER, v_codigo NUMBER)
IS
    DECLARE
    unidades NUMBER;
    STOCK_INSUF EXCEPTION;

    BEGIN
     SELECT stock  INTO unidades FROM PIEZAS
         WHERE CODIGO = v_codigo;
    IF unidades < v_unidades THEN
      RAISE STOCK_INSUF;
    END IF;

    EXCEPTION
   WHEN STOCK_INSUF THEN
    dbms_output.put_line('No hay suficientes unidades');
    END;
```

TRIGGERS

- Un trigger es código (PL/SQL) que se ejecuta automáticamente cuando se realiza una determinada acción sobre la base de datos.
- Tipos de triggers
  - **Triggers de tabla.** Se disparan cuando ocurre una acción DML sobre una tabla.
  - **Triggers de vista.** Se lanzan cuando ocurre una acción DML sobre una vista.
  - **Triggers de sistema.** Se disparan cuando se produce un evento sobre la base de datos (conexión de un usuario, borrado de un objeto,…)

Elementos Básicos

- El evento que da lugar a la ejecución del trigger: **INSERT, UPDATE o DELETE.**
- Instante en el que se lanza el trigger en relación a dicho evento: **BEFORE** (antes), **AFTER** (después) o **INSTEAD OF** (en lugar de).
- Las veces que el trigger se ejecuta: **instrucción** o **fila**.
- El código que ejecuta dicho trigger.

Conceptos Básicos

- **BEFORE.** El código del trigger se ejecuta antes de ejecutar la instrucción DML que causó el lanzamiento del trigger.
- **AFTER.** El código del trigger se ejecuta después de haber ejecutado la instrucción DML que causó el lanzamiento del trigger.
- **INSTEAD OF.** El trigger sustituye a la operación DML. Se utiliza para vistas que no admiten instrucciones DML.
- **De instrucción.** El cuerpo del trigger se ejecuta una sola vez por cada evento que lance el trigger. Opción por defecto.
- **De fila.** El código se ejecuta una vez por cada fila afectada por el evento.

Triggers de instrucción

```
CREATE [OR REPLACE] TRIGGER
  nombreDeTrigger
cláusulaDeTiempo evento1 [OR
  evento2[,...]]
ON tabla
[DECLARE
declaraciones
]
BEGIN
cuerpo
[EXCEPTION captura de excepciones]
END;
```

Triggers de instrucción

- La cláusula de tiempo es una de estas palabras: **BEFORE** o **AFTER**.
- **Evento**

```
{INSERT|UPDATE
   [OF columna1[,columna2,…]]|DELETE}
```

- En el caso de la instrucción `UPDATE`, el apartado `OF` hace que el trigger se ejecute sólo cuando se modifique la columna(s) indicada(s).
- En la sintaxis del trigger, el apartado `OR` permite asociar más de un evento al trigger.

Triggers de instrucción

```
CREATE OR REPLACE TRIGGER Log_emp_salarios
AFTER UPDATE ON Empleados

BEGIN
INSERT INTO Emp_log (Log_date,Accion)
VALUES (SYSDATE, 'Empleados cambio salarios');

END;
```

Triggers de fila

```
CREATE [OR REPLACE] TRIGGER nombreDeTrigger
cláusulaDeTiempo evento1 [OR evento2[,...]]
ON tabla
[REFERENCING {OLD AS nombreViejo | NEW AS
  nombreNuevo}]
FOR EACH ROW
[WHEN condición]
[declaraciones]
Cuerpo
```

Triggers de fila

- **FOR EACH ROW** hace que el trigger se ejecute por cada fila afectada en la tabla por la instrucción DML.
- **WHEN** permite colocar una condición que deben cumplir los registros para que el trigger se ejecute.
- **REFERENCING** permite indicar un nombre para los valores antiguos y otro para los nuevos.
- Cuando se ejecutan instrucciones **UPDATE**, se modifican valores antiguos (**OLD**) por valores nuevos (**NEW**).
- En el apartado de instrucciones del trigger hay que anteponer ":" a las palabra **NEW** y **OLD**

Triggers de fila

```
CREATE TABLE PIEZAS (
tipo VARCHAR2(2),
modelo NUMBER(2),
precio_venta NUMBER(11,4) not null default 0,
PRIMARY KEY (TIPO,MODELO));

                                CREATE TABLE PIEZAS_AUDIT(
                                precio_viejo NUMBER(11,4),
                                precio_nuevo NUMBER(11,4),
                                tipo VARCHAR2(2),
                                modelo NUMBER(2),
                                fecha DATE
                                PRIMARY KEY (TIPO,MODELO,FECHA),
                                CONSTRAINT fk_pieza FOREIGN KEY (TIPO,MODELO) REFERENCES PIEZAS);

CREATE OR REPLACE TRIGGER crear_audit_piezas
 BEFORE UPDATE OF precio_venta
 ON PIEZAS
 FOR EACH ROW
  WHEN (OLD.precio_venta<NEW.precio_venta)
 BEGIN
  INSERT INTO PIEZAS_AUDIT VALUES (:OLD.precio_venta, :NEW.precio_vent, :OLD.tipo, :OLD.modelo, SYSDATE);
END;
```

IF INSERTING, IF UPDATING, IF DELETING

- Se utilizan para determinar la instrucción DML que se estaba realizando cuando se lanzó el trigger.
- Se utiliza en triggers que se lanzan para varias operaciones.

```
CREATE OR REPLACE TRIGGER trigger1
BEFORE INSERT OR DELETE OR UPDATE [OF campo1] ON tabla
FOR EACH ROW
BEGIN
IF DELETING THEN
instrucciones que se ejecutan si el trigger saltó por borrar filas
ELSIF INSERTING THEN
instrucciones que se ejecutan si el trigger saltó por insertar filas
ELSE
instrucciones que se ejecutan si el trigger saltó por modificar filas
END IF
END;
```

Instead Of

- `INSTEAD OF` triggers permiten modificar datos a través de vistas que no son directamente modificables mediante `UPDATE`, `INSERT`, and `DELETE`.
- `INSTEAD OF` solo se puede aplicar a triggers definidos sobre vistas.
- `BEFORE y AFTER` no son aplicables.
- `INSTEAD OF` solo puede ser de tipo fila.

```
CREATE OR REPLACE VIEW pedido_info AS
SELECT c.clienteId, c.apellido, c. nombre, o.orderId, o.fecha, o.estado
FROM clientes c, pedidos o
WHERE c.clienteId = o.clienteId;
```

Instead Of

```
CREATE OR REPLACE TRIGGER pedido_insert
INSTEAD OF INSERT ON pedido_info

BEGIN

INSERT INTO clientes(clienteId, apellido, nombre) VALUES (
:new.clienteId, :new.apellido, :new.nombre);

INSERT INTO pedidos (pdidoId, fecha, clienteId) VALUES ( :new.pedidoId,
:new.fecha, :new.clienteId);

END;
```

Orden de ejecución

- Sobre una misma tabla puede haber varios triggers. El orden de ejecución sería:

1. Disparadores de tipo BEFORE de tipo instrucción
2. Disparadores de tipo BEFORE por cada fila
3. Se ejecuta la propia orden que desencadenó al trigger.
4. Disparadores de tipo AFTER con nivel de fila.
5. Disparadores de tipo AFTER con nivel de instrucción.

Triggers

- `DROP TRIGGER nombreTrigger;`

  Elimina un trigger.
- `ALTER TRIGGER nombreTrigger DISABLE;`

  Desactiva un trigger.
- `ALTER TRIGGER nombreTrigger ENABLE;`

  Activa un trigger.
- `ALTER TABLE nombreTabla {DISABLE|ENABLE} ALL TRIGGERS;`

  Desactiva o activa todos los triggers de una tabla.