Introducción a PL/SQL

- **Características de PL/SQL (*Procedural Language*/SQL):**
  - Combina la potencia y flexibilidad de **SQL** con la de un lenguaje **3GL:**
    - **SQL es un Lenguaje de Cuarta Generación (4GL):** El lenguaje describe lo que debe hacerse pero no cómo hacerlo, dentro de una sintaxis relativamente simple.
    - **Lenguajes de Tercera Generación (3GL):** Tienen estructuras procedimentales para expresar cómo efectuar las operaciones.
      - **PL/SQL** permite utilizar o declarar: Variables y tipos de datos, estructuras de control (selección e iteración), procedimientos y funciones y, a partir de PL/SQL 8, también tipos de objetos y métodos.
  - Es un **lenguaje estructurado y potente**, con estructuras ideales para el trabajo con bases de datos.
  - **Integrado en el SGBD Oracle**, por lo que su ejecución es **eficiente**.
  - Permite empaquetar órdenes SQL, de forma que **minimice** la **comunicación** entre Cliente y Servidor de BD y los **accesos** a la BD.
  - PL/SQL se basa en el **Lenguaje Ada** (3GL) y tiene muchas de sus características (estructura de bloques, excepciones...)
  - Oracle y PL/SQL soportan el **estándar ANSI SQL92 (o SQL2)** a nivel básico (se pretende que sea a nivel completo en futuras versiones).

Introducción a PL/SQL

- **Caracteres:** PL/SQL utiliza los caracteres **ASCII**. PL/SQL no distingue entre mayúsculas y minúsculas, excepto en una cadena de caracteres entre comillas.
- **Palabras Reservadas:** PL/SQL tiene más de 200. Están reservadas para la definición del lenguaje y suelen ponerse siempre en mayúsculas.
- **Identificadores:** Empiezan con una letra, seguida opcionalmente de letras, números y tres caracteres especiales (**$**, _ y #). Máxima Longitud: 30.
  - **Identificadores entre comillas dobles:** Como norma general no deben usarse pues denotan un mal estilo de programación, pero son útiles para: a) Introducir otros caracteres imprimibles (incluyendo el espacio), b) Distinguir entre mayúsculas y minúsculas y c) Utilizar una palabra reservada como identificador.
    - Ejemplo: Si una tabla se llama **`EXCEPTION`**, usar **`"EXCEPTION"`** (en mayúsculas).
- **Delimitadores:** Símbolos con un significado especial, como son:
  - **Operadores Aritméticos:** **`+`**, **`-`**, **`*`**, **`/`**, **`**`** (potenciación).
  - **Operadores Relacionales:** **`=`**, **`>`**, **`<`**, **`>=`**, **`<=`**, **`<>`**, **`!=`**, **`^=`**, **`~=`**.
  - **Otros Operadores:** **`:=`** (asignación), **`||`** (concatenar cadenas), **`=>`** (asociación), **`..`** (rango), **`LIKE`**, **`BETWEEN`**, **`IN`**, **`IS NULL`** y los operadores lógicos (**`NOT`**, **`AND`** y **`OR`**).
  - **Otros:** Paréntesis para alterar la precedencia (**`(`** y **`)`**), terminador de orden (**`;`**), comentarios (**`--`**, **`/*`** y **`*/`**), espacio, tabulador, retorno de carro, delimitadores de etiquetas (**`<<`** y **`>>`**), indicador de variable de asignación (**`:`**), delimitadores de cadenas (**`'`**, comilla simple), identificador entre comillas dobles (**`"`**), selector de componente de un registro, una tabla... (**`.`**), enlace a BD (@), indicador de atributo (%) y separador de elementos (**`,`**).

Bloques PL/SQL

- **Bloque PL/SQL:** Unidad básica en **PL/SQL.**
  - Todos los programas PL/SQL están compuestos por **bloques**, que pueden anidarse.
  - Normalmente cada **bloque** realiza una unidad lógica de trabajo.
  - **Estructura de un bloque PL/SQL:**

```
DECLARE
  /* Declaración de variables, tipos, cursores y
     subprogramas locales, con VISIBILIDAD hasta END; */
BEGIN
  /* Programa: Órdenes (SQL y procedimentales) y bloques.
     Esta sección es la única obligatoria. */
EXCEPTION
  /* Manejo de excepciones (errores) */
END;
```

  - **Excepciones:** Permiten controlar los errores que se produzcan sin complicar el código del programa principal.
    - Cuando se produce un error, la ejecución del programa salta a la sección **EXCEPTION** y allí puede controlarse qué hacer dependiendo del tipo de error producido.

Bloques PL/SQL

- **Tipos de Bloques:**
  - **Anónimos (*anonymous blocks*):** Se construyen normalmente de manera dinámica para un objetivo muy concreto y se ejecutan, en general, una única vez.
  - **Nominados (*named blocks*):** Son similares a los bloques anónimos pero con una etiqueta que da nombre al bloque.
  - **Subprogramas:** Son procedimientos (*procedures*), funciones (*functions*) o grupos de ellos, llamados paquetes (*packages*).
    - Se construyen para efectuar algún tipo de operación más o menos frecuente y se almacenan para ejecutarlos cuantas veces se desee.
    - Se ejecutan con una llamada al procedimiento, función o paquete.
  - **Disparadores (*triggers*):** Son bloques nominados que se almacenan en la BD. Su ejecución está condicionada a cierta condición, como por ejemplo usar una orden concreta del DML.
- **Comentarios:** Pueden incluirse siempre que se desee.
  - **Monolínea:** Empiezan con 2 guiones **--** y terminan a final de línea.
  - **Multilínea:** Empiezan con **/*** y terminan con ***/** (como en C).

Bloques PL/SQL: Ejemplos

- **Bloque Anónimo:** Insertar 2 piezas Tornillo con distintos pesos.

```
DECLARE
  NumP NUMBER:=7; -- Número de Pieza, empezamos en 7
BEGIN
  INSERT INTO Pieza (P#,Nombre,Peso)
         VALUES (NumP,'Tornillo', 2);
  INSERT INTO Pieza (P#,Nombre,Peso)
         VALUES (NumP+1,'Tornillo', 4);
END;
```

- **Bloque Nominado:** Podemos poner una etiqueta a un bloque anónimo para facilitar referirnos a él.
  - Antes de la palabra **DECLARE** se pone el **nombre del bloque** entre ángulos dobles. **Ejemplo:** `<<Bloque_Insertar>>` `DECLARE ...`
  - Es buena costumbre, aunque es opcional, poner el nombre también después de la palabra **END**. **Ej.:** `END Bloque_Insertar;`
  - En **bloques anidados** puede ser útil nominarlos para referirse a una variable en particular de ese bloque, en caso de que existan varias con el mismo nombre: **Ejemplo:** `Bloque_Insertar.Var1`

Bloques PL/SQL: Ejemplos

- **Procedimientos:** Se usa la palabra reservada **PROCEDURE** junto con las de creación o reemplazo, sustituyendo a **DECLARE**.

```
CREATE OR REPLACE PROCEDURE Pieza7 AS
  NombrePieza7  VARCHAR2(50);
BEGIN
  SELECT NombreP INTO NombrePieza7 FROM Pieza
    WHERE P#=7;
  DBMS_OUTPUT.PUT_LINE ('Nombre de la Pieza 7: '
    ||NombrePieza7);
END;
```

Sobre SQL*Plus

- **PUT_LINE** es un procedimiento del paquete **DBMS_OUTPUT** que sirve para escribir un dato (**PUT**) seguido de un fin de línea (**NEW_LINE**). En **SQL*Plus** debe estar activada la salida por pantalla con: **SET SERVEROUTPUT ON [SIZE n]**, donde **n** Î **[2000,1000000]** es el máximo nº de bytes (**2000** pordefecto).
- **SET LINESIZE n**, establece el tamaño de una línea en **SQL*Plus**, para todas sus salidas.
- **SHOW ERR**, muestra los errores de compilación de un bloque PL/SQL.
- **EXEC <sentencia>**, ejecuta una sentencia PL/SQL. Se puede usar para ejecutar un procedimiento sin tener que crear un bloque PL/SQL.

- **Disparadores o *Triggers*:** Hay que especificar CUANDO se disparará.

```
CREATE OR REPLACE TRIGGER PesoPositivo
  -- Se activará cada vez que se inserte o actualice
  BEFORE INSERT OR UPDATE OF Peso ON Pieza
  FOR EACH ROW
BEGIN
  IF :new.Peso < 0 THEN
     RAISE_APPLICATION_ERROR(-20100, 'Peso no válido');
  END IF;
END PesoPositivo;
```

Variables y Tipos de Datos

- **Variables PL/SQL:** Se declaran con el siguiente formato:
  <Nombre> <Tipo> [[CONSTANT|NOT NULL] :=<Valor_Inicial>];
  - **Tipos de datos:** Puede ser cualquier tipo válido en una columna de la BD (vistos anteriormente) y otros tipos adicionales. El valor por defecto es NULL, excepto que se especifique un valor como <Valor_Inicial>(puedesustituirse **:=** por **DEFAULT**). Con **NOT NULL** se requiere la inicialización.
- **Tipos Escalares:**
  - **Numéricos Reales:** **NUMBER(**p,e**)** y sus subtipos totalmente equivalentes definidos por cuestiones de compatibilidad: **DEC**, **DECIMAL**, **DOUBLE PRECISION**, **INT**, **INTEGER**, **NUMERIC**, **SMALLINT** y **REAL**.
    - Se almacenan en formato decimal: Para operaciones aritméticas deben traducirse a binario.
  - **Numéricos Enteros:** **BINARY_INTEGER**, que es un entero en binario (complemento a 2) con rango ±2147483647, ideal para variables sobre las que se efectuarán operaciones (contadores...). Tienedefinidos subtipos restringidos en su rango: **NATURAL** [0,2147483647], **NATURALN** (igual que **NATURAL** pero **NOT NULL**), **POSITIVE** [1,2147483647], **POSITIVEN**, **SIGNTYPE** (–1, 0 y 1).
    - **PLS_INTEGER** es similar a **BINARY_INTEGER**, pero más rápido en las operaciones aritméticas y que genera un error si se produce un desbordamiento (ORA-1426) al asignarlo a un **NUMBER**.
  - **Carácter:** **VARCHAR2(**max_tama**)**, con max_tama<=32676 **bytes** (como columna de la BD admite 4000 → Cuidado con los errores). Si se usa un código distinto al código ASCII, el número total de caracteres puede ser menor.
    - **CHAR(**tama_fijo**)** con 1 por defecto y 32767 como máximo (como columna de la BD admite 255), *se rellena siempre con blancos*. **LONG** es una cadena de longitud variable con un máximo de 32760 (como columna de la BD admite 2GB). **NCHAR** y **NVARCHAR2** permitenalmacenar cadenas en un conjunto de caracteres Nacional distinto al propio de PL/SQL.
  - **Binarios:** **RAW(**max_tama**)**, con max_tama<=32676 **bytes. LONG RAW** admite un máximo de 32760 **bytes. ROWID** es un tipo para almacenar identificadores de fila (pseudocolumna **ROWID**).
  - **Fecha:** **DATE**, como en SQL de Oracle almacena siglo, año, mes, día, hora, min. y segs. (7 bytes).
  - **Lógico:** **BOOLEAN**, con los siguientes valores posibles: **TRUE**, **FALSE** y **NULL**.
- **TiposCompuestos:** **RECORD**, **TABLE** y **VARRAY**.
- **Tipos Referencias (punteros):** **REF CURSOR** y **REF** <TipoObjeto>.
- **Tipos LOB** (*Large OBject*)**:** **BFILE**, **LOB**, **CLOB** y **NLOB** (se usan con el paquete **DBMS_LOB**).

Variables y Tipos de Datos

- **Especificación de Tipo con <Tabla>.<Columna>%TYPE:**
  - Sirve para dar a una variable el **tipo que tenga asignado una columna** de la BD, independientemente de cómo esté definido éste.
  - También puede usarse **sobre variables** anteriormente definidas.
  - Esto hace los **programas más robustos** frente a cambios de tipo.
  - **Ejemplos:**
```
DECLARE
        NumPieza Pieza.P#%TYPE;
        PiezaX   NumPieza%TYPE;
```
  - **La restricción NOT NULL no se hereda.**
- **Subtipos con SUBTYPE: SUBTYPE <NewT> IS <Definición>;**
  - **Ejemplos:**
```
DECLARE
        SUBTYPE TNumPieza NUMBER;
        PiezaX  TNumPieza;
        PiezaY  TNumPieza(5); -- Restringe un subtipo
```
  - **Para declarar Tipos Compuestos y Referencias se usa TYPE.**
- **Conversiones de Tipo:**
  - **Explícita:** Usando funciones de conversión: **TO_CHAR**, **TO_NUMBER**...
  - **Implícita:** PL/SQL realiza conversiones automáticas si es posible, incluso entre datos de distintas familias (numéricos, caracteres, fechas...). Esto es desaconsejado para evitar problemas.

Selección: IF-THEN-ELSE

- **Expresiones Lógicas o Booleanas:** Usan una lógica trivaluada (**TRUE**, **FALSE** y **NULL**), donde **NULL** tiene el significado de "desconocido o no definido".
  - Cualquier expresión relacional con un operando nulo, devuelve **NULL**.
  - **Tablas de Verdad:**

NOT	
T	F
N	N
F	T

AND	T	N	F
T	T	N	F
N	N	N	F
F	F	F	F

OR	T	N	F
T	T	T	T
N	T	N	N
F	T	N	F

- **Estructura de Control de Selección IF-THEN-ELSE:**
  - **Formato:**

```
IF <Expresión_Booleana1> THEN
    <Secuencia_de_Órdenes1>;
[ELSIF <Expresión_Booleana2> THEN
       <Secuencia_de_Órdenes2>;]
. . .
[ELSE  <Secuencia_de_Órdenes3>;]
END IF;
```

  - Como se muestra, las cláusulas **ELSIF** y **ELSE** son opcionales y puede haber tantas cláusulas **ELSIF** como se desee.
  - Se ejecuta la **<Secuencia_de_Órdenes1>si<Expresión_Booleana1>es TRUE. Si esa expresión vale FALSE** o **NULL**, no se ejecutará y pasará a ejecutar las siguientes cláusulas.
    - Los valores **NULL** hacen que no sea equivalente intercambiar las secuencias de órdenes si se niega la **<Expresión_Booleana1>.**

Ejemplos IF-THEN-ELSE

Ejemplo 1:

```
IF A < B THEN
   C := 1;
ELSE
   C := 2;
END IF;
```

No es equivalente a lo siguiente, si las variables A o B puedentener valores NULL:

```
IF A >= B THEN
   C := 2;
ELSE
   C := 1;
END IF;
```

Lo mejor es comprobar primero si son valores NULL:

```
IF A IS NULL OR
   B IS NULL THEN
   C := 3;
ELSIF A < B THEN
   C := 1;
ELSE
   C := 2;
END IF;
```

Ejemplo 2:

```
DECLARE
  NumHabitaciones Hotel.Nhabs%TYPE;
  Comentario      VARCHAR2(40);
BEGIN
  -- Número de habitaciones del Hotel 99:
  SELECT Nhabs INTO NumHabitaciones
     FROM Hotel WHERE HotelID = 99;
  IF NumHabitaciones < 50 THEN
     Comentario:='Pequeño';
  ELSIF NumHabitaciones < 100 THEN
     Comentario:='Mediano';
  ELSE Comentario:='Grande';
  END IF;
  INSERT INTO Tabla_C (Comment) VALUES (Comentario);
END;
```

Si Nhabs es NULL se ejecuta la parte ELSE

Bucles: LOOP, WHILE *y* FOR

- **LOOP:**

```
LOOP
  <Secuencia_de_Órdenes>;
END LOOP;
```

  - Se ejecutará "infinitamente" hasta que se ejecute la orden: **EXIT [WHEN <Condición>];**
  - Son equivalentes: a) **IF <Condición> THEN EXIT; END IF;** b) **EXIT WHEN <Condición>;**

- **WHILE:**

```
WHILE <Condición> LOOP
  <Secuencia_de_Órdenes>;
END LOOP;
```

  - Se ejecuta cero o más veces mientras la condición sea cierta.
  - Puede usarse también la orden: **EXIT**.

- **FOR:**

```
FOR <Contador> IN [REVERSE] <Inf>..<Sup> LOOP
  <Secuencia_de_Órdenes>;
END LOOP;
```

  - **<Contador>** es una variable que se declara automáticamente de tipo **BINARY_INTEGER:** No hay que declararla. Si se declara una variable de igual nombre ambas serán variables distintas (como en bloques anidados).
    - La variable **<Contador>** tomará valores desde **<Inf>** hasta **<Sup>**, incrementándose automáticamente en una unidad.
    - Con **REVERSE** los valores se toman en orden inverso: Desde **<Sup>** hasta **<Inf>.**
    - **<Inf>**y **<Sup>** pueden ser constantes, variables o expresiones.

Ejemplos: LOOP, WHILE *y* FOR

- **LOOP:** Insertar 10 filas en la tabla **Tabla_Temp** con valores del 1 al 10:

```
DECLARE Cont BINARY_INTEGER:=1;
BEGIN
  LOOP
    INSERT INTO Tabla_Temp VALUES (Cont);
    Cont := Cont + 1;
    EXIT WHEN Cont>10;
  END LOOP;
END;
```

- **WHILE:**

```
DECLARE Cont BINARY_INTEGER:=1;
BEGIN
  WHILE Cont<=10 LOOP
    INSERT INTO Tabla_Temp VALUES (Cont);
    Cont := Cont + 1;
  END LOOP;
END;
```

- **FOR:**

```
BEGIN
  FOR Cont IN 1..10 LOOP
    INSERT INTO Tabla_Temp VALUES (Cont);
  END LOOP;
END;
```

  - Observe que en el bucle **FOR** no se declara la variable contador.
  - En un bucle **FOR** no puede modificarse la variable de control del bucle.

Registros PL/SQL: RECORD

- **Registros:** Son agrupaciones de datos relacionados similares a las estructuras (**`struct`**) del lenguaje C o los registros de Modula2.
  - **Es necesario definir un Tipo de Dato Registro, para declarar variables.**
  - **Formato:**

```
TYPE <Tipo_Registro> IS RECORD (
    <Campo1> <Tipo1> [[NOT NULL] :=<Expr1>],
    <Campo2> <Tipo2> [[NOT NULL] :=<Expr2>],
    . . .
    <CampoN> <TipoN> [[NOT NULL] :=<ExprN>]);
```

  - Igual que en los lenguajes de programación, para acceder a un campo se usa la **NotaciónPunto: <VariableRegistro>.<Campoi>**.
  - **AsignarRegistros:** Se permite si son del mismo tipo.
    - Si son de tipos distintos no se pueden asignar, aunque estén definidos igual. En ese caso, se pueden asignar campo a campo.
    - También se pueden asignar los campos de un **`SELECT`** en un registro compatible.
  - Declarar registros con los campos de una tabla**: <Tabla>%ROWTYPE.**

```
DECLARE
  TYPE T_HOTEL IS RECORD(
    Nhabs   Hotel.Nhabs%TYPE,
    HotelID Hotel.HotelID%TYPE);
  Hotel99 T_HOTEL;
BEGIN
  SELECT Nhabs, HotelID INTO Hotel99
    FROM Hotel WHERE ...
  ...
```

```
DECLARE
  Hotel99 Hotel%ROWTYPE;
BEGIN
  SELECT * INTO Hotel99
    FROM Hotel WHERE ...
  ...
```


Tablas PL/SQL: TABLE

- **Tablas PL/SQL:** Son agrupaciones como las matrices.
  - Es necesario definir un **Tipo de Dato Tabla,** para declarar variables.
  - **Formato:**

```
TYPE <Tipo_Tabla> IS TABLE OF <TipoBase>
  INDEX BY BINARY_INTEGER;
```

  - **Ejemplo:**

```
TYPE T_LISTA_FECHAS IS TABLE OF DATE
  INDEX BY BINARY_INTEGER;
Fechas T_LISTA_FECHAS; -- Variable Tabla
```

  - **Referenciar Elementos en una Tabla:** Se utiliza un índice entre paréntesis de tipo **`BINARY_INTEGER`** (o convertible a él).
    - **Ejemplo:** Para **insertar** la fecha actual con índice o clave **`-3`:**

```
Fechas(-3) := SYSDATE;
```

  - Una **Tabla PL/SQL** es como una tabla de la BD con un atributo o columna índice de tipo **`BINARY_INTEGER`:**
    - El **TipoBase** puede ser también un **Registro:**
      - **Formato de Acceso a un campo: <Tabla>(<Índice>).<Campo>**
    - El **LímitedelÍndice** o clave es el que impone su tipo de dato.
    - Los **DatosInsertados** en una Tabla PL/SQL no guardan ningún orden.
    - Puede **asignarse** una tabla PL/SQL a otra si son del mismo tipo.
    - Los **ValoresdelÍndice** no tienen que ser consecutivos ni empezar en ningún valor específico. Se aconseja empezar en 1.

Tablas PL/SQL: Atributos de TABLE

- **Atributos de Tablas PL/SQL:** Son valores específicos de una Tabla PL/SQL. **Formato de Acceso: <Tabla>.<Atributo>**
  - **<Tabla>.COUNT** ® Número de filas de la tabla (tipo **NUMBER**).
  - **<Tabla>.EXIST(i)** ® **TRUE** si existe una fila con índice **i** (**BOOLEAN**).
    - Sirve para evitar acceder a un elemento de la tabla que no exista.
  - **<Tabla>.FIRST** ® Primer índice de la tabla, el menor (**BINARY_INTEGER**).
  - **<Tabla>.LAST** ® Último índice de la tabla, el mayor (**BINARY_INTEGER**).
  - **<Tabla>.NEXT(i)** ® Índice que le sigue al índice **i** (**BINARY_INTEGER**).
  - **<Tabla>.PRIOR(i)** ® Índice anterior al índice **i** (**BINARY_INTEGER**).
    - **NEXT** y **PRIOR** son útiles para construir bucles que recorran la tabla entera, independientemente de los valores de índice utilizados.
  - **<Tabla>.DELETE** ® No devuelve un valor, sino que es una sentencia para borrar filas de la tabla**:**
    - **<Tabla>.DELETE** ® Borra todas las filas de la tabla PL/SQL.
    - **<Tabla>.DELETE(i)** ® Borra la fila cuyo índice es **i**.
    - **<Tabla>.DELETE(i,j)** ® Borra todas las filas de la tabla cuyos índices estén entre **i** y **j**.

Órdenes SQL en un Programa PL/SQL

- **Órdenes SQL que pueden utilizarse:**
  - **DML: SELECT**, **INSERT**, **UPDATE** y **DELETE**.
    - Permiten utilizar variables como expresiones dentro de su sintaxis.
      - Consejo: No declarar variables con el mismo nombre que las columnas.
    - No pueden utilizarse variables para los nombres de tablas y columnas (para ello usar SQL dinámico).
  - **Control de Transacciones: COMMIT**, **ROLLBACK** y **SAVEPOINT**.
- **SQL Dinámico:** Permite **crear órdenes en Tiempo de Ejecución**, de forma que esas órdenes no son compiladas en **Tiempo de Compilación**. Se necesita usar el paquete **DBMS_SQL.**
  - **Ejemplos:** Para ejecutar órdenes de **DDL**, como crear una tabla (**CREATE TABLE**), pero también para insertar valores en ella (**INSERT**), ya que en tiempo de compilación no existirá esa tabla y, por tanto, no puede compilarse la orden de inserción.
- **Órdenes DML:**
  - **INSERT, UPDATE, DELETE:** No sufren modificaciones (excepto las órdenes **UPDATE** y **DELETE** para cursores usando **CURRENT OF**).
  - **SELECT:** Su sintaxis se ve modificada para indicar donde se almacenarán los datos seleccionados (cláusula **INTO**).

Órden SELECT *en PL/SQL*

- **SELECT:** Se requiere introducir los datos seleccionados en variables.
  - **Cláusula INTO:** Se coloca justo antes de **FROM** y después de la lista de elementos seleccionados. La palabra **INTO** va seguida de**:**
    - Una **lista de tantas variables** como elementos seleccionados y con sus tipos compatibles en el mismo orden, o bien,
    - Un **registro** con sus campos también compatibles.
  - **La consulta sólo debe recuperar una fila:** Si recupera varias se producirá un error. Para consultas múltiples, se deben usar **cursores**.
  - **Ejemplo:**

```
DECLARE
  Media   NUMBER;
  Hotel99 Hotel%ROWTYPE;
BEGIN
  -- Media de habitaciones de todos los hoteles:
  SELECT AVG(Nhabs) INTO Media FROM Hotel;
  -- Todos los datos del Hotel 99:
  SELECT * INTO Hotel99
     FROM Hotel WHERE HotelID = 99;
  -- Comparar con la Media:
  IF Media > Hotel99.Nhabs THEN
     UPDATE Hotel SET Nhabs=Nhabs+1 WHERE HotelID=99;
  ...
END;
```


Cursores en PL/SQL

- **Área de Contexto de una orden SQL:** Zona de memoria con la información relacionada con la orden y con el resultado, si es una consulta.
- **Cursor:** Es un puntero al Área de Contexto.
  - **Cursor Explícito:** Se asocia directamente a una sentencia **SELECT**.
  - **Cursor Implícito:** Para el resto de órdenes SQL.
- **Cursores Explícitos:** Su procesamiento consta de 4 pasos:
  - **1. Declaración del cursor:** En la sección **DECLARE:**
    CURSOR <Nombre_Cursor> IS <Orden_SELECT>;
    - La orden **SELECT** puede utilizar variables declaradas antes.
  - **2. Apertura: OPEN <Nombre_Cursor>;**
    - Ejecuta la consulta asociada a ese cursor con el valor actual de las variables que use y pone el puntero apuntando a la primera fila.
  - **3. Extraer datos:** Es similar a la cláusula **INTO** de la orden **SELECT:**
    FETCH <Nombre_Cursor> INTO {<Variables> | <Registro>};
    - Después de cada **FETCH** el puntero avanza a la siguiente fila.
    - El **atributo <Nombre_Cursor>%FOUND** vale **TRUE** si no se ha llegado al final del cursor. El opuesto es **%NOTFOUND**.
  - **4. Cierre: CLOSE <Nombre_Cursor>;**
    - Libera la memoria de un cursor abierto. Tras esto no podrá usarse.

Atributos de los Cursores en PL/SQL

- **<Nombre_Cursor>%FOUND:** Atributo booleano que devuelve:
  - **TRUE** si la última orden **FETCH** devolvió una fila.
  - **FALSE** en caso contrario.
  - **NULL** si aún no se a efectuado ninguna extracción con **FETCH**.
  - Genera un **ERROR** si el cursor no está abierto (ORA-1001, **INVALID_CURSOR**).
- **<Nombre_Cursor>%NOTFOUND:** Atributo booleano opuesto a **%FOUND** (en los valores lógicos).
  - Suele usarse como condición de salida en un bucle.
  - Cuando toma el valor **TRUE** la anterior orden **FETCH** no realizó asignación a las variables de **INTO**, por lo que contendrán lo mismo que tenían antes (valores de la última fila recuperada).
- **<Nombre_Cursor>%ISOPEN:** Atributo booleano que devuelve **TRUE** si el cursor está abierto. En caso contrario devuelve **FALSE**.
  - Nunca genera un error.
- **<Nombre_Cursor>%ROWCOUNT:** Atributo numérico que devuelve el número de filas extraídas con **FETCH** hasta el momento.
  - Genera un **ERROR** si el cursor no está abierto.

Ejemplo de un Cursor (bucle LOOP)

- **Ejemplo:** Aplicar una "ecotasa" a los hoteles grandes (con más de 50 habitaciones). Ese impuesto será de 10 euros si el hotel tiene piscina y de 5 euros en caso contrario:

La orden **EXIT WHEN** debe situarse después del **FETCH** y antes de procesar los datos de cada fila, para evitar procesar dos veces la última fila del cursor.

¡OJO con los valores NULL.

```
DECLARE
  NumHabitaciones Hotel.Nhabs%TYPE;
  UnHotel         Hotel%ROWTYPE;
  CURSOR HotelesGrandes IS SELECT * FROM Hotel
                           WHERE Nhabs>NumHabitaciones;
BEGIN
  NumHabitaciones:=50;
  OPEN HotelesGrandes;
  -- Bucle para procesar todos los Hoteles Grandes:
  LOOP
    FETCH HotelesGrandes INTO UnHotel;
    EXIT WHEN HotelesGrandes%NOTFOUND;
    -- Procesamiento del hotel almacenado en UnHotel:
    IF UnHotel.TienePiscina = 'S' THEN
       UPDATE Hotel SET PrecioHab=PrecioHab + 10
         WHERE HotelID=UnHotel.HotelID;
    ELSE
       UPDATE Hotel SET PrecioHab=PrecioHab + 5
         WHERE HotelID=UnHotel.HotelID;
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Hoteles Procesados: '
    || HotelesGrandes%ROWCOUNT);
  CLOSE HotelesGrandes;
  COMMIT;
END;
```

Ejemplos con los bucles **WHILE** *y* **FOR**

- **Ejemplo con bucle WHILE:**

```
BEGIN
  NumHabitaciones:=50;
  OPEN HotelesGrandes;
  FETCH HotelesGrandes INTO UnHotel; -- Primera Fila (antes del bucle)
  -- Bucle para procesar todos los Hoteles Grandes:
  WHILE HotelesGrandes%FOUND LOOP
    -- Procesamiento del hotel almacenado en UnHotel:
    ...
    FETCH HotelesGrandes INTO UnHotel; -- Siguiente Fila (al final)
  END LOOP;
  CLOSE HotelesGrandes;
  COMMIT WORK;
END;
```

- **Ejemplo con bucle FOR:** Este tipo de bucle no requiere ni abrir, ni cerrar el cursor, ni usar la orden **FETCH** para recuperar las filas una a una, ni declarar la variable en la que se recuperan los datos (de tipo **<Cursor>%ROWTYPE**).

```
BEGIN
  NumHabitaciones:=50;
  -- Bucle para procesar todos los Hoteles Grandes:
  FOR UnHotel IN HotelesGrandes LOOP
    -- Procesamiento del hotel almacenado en UnHotel
    -- (a la variable UnHotel no pueden asignarsele valores):
    ...
    -- Antes de dar otra vuelta comprueba si existen más filas.
  END LOOP; -- Cierra el cursor automáticamente
  COMMIT WORK;
END;
```


Cursores Parametrizados

- **Cursores Parametrizados:** En vez de usar variables del bloque dentro del cursor (en su definición), pueden usarse parámetros (formales) que se ponen entre paréntesis tras el nombre del cursor cuando se declara.
  - Al abrir el cursor deben darse los parámetros actuales.
  - **Ejemplo:**

```
DECLARE
  UnHotel        Hotel%ROWTYPE;
  CURSOR HotelesGP (NumHabs Hotel.Nhabs%TYPE,
                    Piscina Hotel.TienePiscina%TYPE)
  IS SELECT * FROM Hotel
     WHERE Nhabs>NumHabs AND TienePiscina=Piscina;
BEGIN
  OPEN HotelesGP(50,'S');
  LOOP
    FETCH HotelesGP INTO UnHotel;
    EXIT WHEN HotelesGP%NOTFOUND;
    -- Procesamiento del hotel almacenado en UnHotel:
    ...
  END LOOP;
  CLOSE HotelesGP;
  COMMIT; -- Fin de la primera transacción
  OPEN HotelesGP(100,'N');
  LOOP
    ...
  END LOOP;
  CLOSE HotelesGP;
  COMMIT; -- Fin de la segunda transacción
END;
```

Cursores Implícitos o Cursores SQL

- **Cursores SQL o Implícitos:** Son los cursores que PL/SQL abre y cierra implícitamente: Para las órdenes SQL **INSERT**, **UPDATE**, **DELETE** y la orden **SELECT..INTO** (para recuperar una única fila).
  - En estos cursores no se puede usar **FETCH**, **OPEN** ni **CLOSE**.
  - Pueden usarse los **atributos** vistos anteriormente, usando como **nombre del cursor: SQL.**
    - El atributo **SQL%ISOPEN** siempre será **FALSE:** PL/SQL cierra automáticamente los cursores implícitos.

  - **Ej.:**

```
UPDATE Hotel SET PrecioHab=PrecioHab + 10
  WHERE HotelID = 99;
IF SQL%NOTFOUND THEN
   INSERT INTO Hotel (HotelID,PrecioHab,Nhabs,TienePiscina)
          VALUES (99, 110, 60, 'S');
END IF;
```

  - **Ej.:**

```
UPDATE Hotel SET PrecioHab=PrecioHab + 10
  WHERE Nhabs > 50 AND TienePiscina = 'S';
DBMS_OUTPUT.PUT_LINE('HotelesProcesados:'||SQL%ROWCOUNT);
IF SQL%ROWCOUNT=0 THEN
   INSERT INTO Hotel (HotelID,PrecioHab,Nhabs,TienePiscina)
          VALUES (99, 110, 60, 'S');
   DBMS_OUTPUT.PUT_LINE('Hotel99Insertado.');
END IF;
COMMIT;
```


Cursores Implícitos SELECT..INTO

- Con la **orden SELECT..INTO** pueden usarse también los atributos, pero no tiene sentido:
  - Cuando no se selecciona ninguna fila o se seleccionan más de una, PL/SQL genera una **excepción** o **error**, que transfiere la ejecución automáticamente a la sección **EXCEPTION** del bloque PL/SQL actual.
  - **Ej.:**

Este **IF** es absurdo

```
DECLARE
  Hotel99 Hotel%ROWTYPE;
BEGIN
  SELECT * INTO Hotel99 WHERE Nombre='Costanera';
  IF SQL%NOTFOUND THEN
     -- Si hay error este IF no se ejecutará nunca.
     -- Si no hay error siempre será FALSE
  END IF;
  -- Tratamiento del Hotel Costanera:
  ...
EXCEPTION
  -- Cuando no se recupera ninguna fila:
  WHEN NO_DATA_FOUND THEN
    INSERT INTO Hotel
      (HotelID,Nombre,PrecioHab,Nhabs,TienePiscina,Calif)
       VALUES (99, 'Costanera', 110, 60, 'S', 3 );
  -- Cuando se recuperan varias filas:
  WHEN TOO_MANY_ROWS THEN
    DELETE Hotel WHERE Nombre='Costanera' AND HotelID<>99;
END;
```

Cursores Explícitos SELECT FOR UPDATE

- Los cursores explícitos **evalúan la consulta** cuando se **abre** el cursor, pero **no bloquean** el acceso a las tablas involucradas.
  - **Otro usuario podrá modificarlas** mientras se procesa el cursor.
  - Puede que interese **Bloquear el acceso a esas tablas** si van a modificarse, de forma que no puedan ser modificadas por nadie hasta que termine la transacción:
- **Formato:** Se añade la cláusula **FOR UPDATE** tras la consulta**:**
  ... **<orden_SELECT> FOR UPDATE [OF <columna/s>] [NOWAIT]**
  - **<columna/s>:** Es una columna o una lista de columnas de las tablas de la consulta que se modificarán. Su especificación es opcional.
  - **NOWAIT:** Si otra sesión ha bloqueado lo que el cursor quiere bloquear, el cursor **esperará indefinidamente** hasta que sea liberado. Esta cláusula indica que el cursor no debe esperar, produciendo un error (ORA-54). En ese caso puede intentarse después, cambiar el orden de las operaciones...
  - **Ej.:**
```
CURSOR HotelesC IS -- Hoteles que empiecen por C
    SELECT * FROM Hotel WHERE Nombre LIKE 'C%' FOR UPDATE;
```
  - **Ej.:**
```
CURSOR HotelesG IS SELECT * FROM Hotel WHERE Nhabs>30
                   FOR UPDATE OF PrecioHab, Calif;
```


SELECT FOR UPDATE / WHERE CURRENT OF

- Al usar la orden **UPDATE/DELETE** es necesario especificar las tuplas a **Actualizar/Borrar** con la cláusula **WHERE.**
- Cuando se usa un **cursor SELECT FOR UPDATE** y extraemos sus filas una a una es fácil escribir esa cláusula **WHERE**, porque hay una forma de referirse a la última fila recuperada del cursor.
- **Formato:** Se sustituye la cláusula **WHERE** de una orden **UPDATE/DELETE** por: **... WHERE CURRENT OF <Cursor>**
  - **UPDATE** sólo debe modificar las columnas que se especificaron en el cursor **SELECT FOR UPDATE**. Si no se especificaron columnas, se puede modificar cualquier columna.
  - **Ej.:**
```
OPEN HotelesC; -- Aquí se producen los bloqueos
LOOP
  FETCH HotelesC INTO UnHotel;
  EXIT WHEN HotelesC%NOTFOUND;
  -- Procesamiento del hotel almacenado en UnHotel:
  IF UnHotel.Calif > 0 THEN
       UPDATE Hotel SET Calif = Calif + 1
                    WHERE CURRENT OF HolelesC;
  ELSE DELETE Hotel WHERE CURRENT OF HolelesC;
  END IF;
END LOOP;
CLOSE HotelesC; -- Cierra el cursor: NO libera bloqueos
COMMIT; -- Fin de la transacción: Libera los bloqueos
```

Bucles de Cursores y `COMMIT`

- Tras un bucle de un cursor de tipo **`SELECT FOR UPDATE`** la orden **`COMMIT`** libera los bloqueos establecidos por el cursor.
  - No puede usarse **`COMMIT`** dentro del bucle ya que esto invalidará el cursor y producirá un error en la siguiente operación con el cursor.
  - Si el cursor no ha sido declarado como de tipo **`FOR UPDATE`**, entonces no hay problema, ya que ese cursor no bloquea nada.
    - Si queremos ejecutar **`COMMIT`** dentro del bucle de un cursor tras cada modificación, el cursor no debe ser de tipo **`FOR UPDATE`**.
    - Si el cursor no es de tipo **`FOR UPDATE`**, no podemos utilizar la cláusula **`WHERE CURRENT OF`**, por lo que se empleará la clave primaria.
  - **Ej.:**

```
DECLARE
   UnEmpleado Empleados%ROWTYPE;
   CURSOR Empl_con99 IS SELECT * FROM Empleados
     WHERE Dpto IN (SELECT Dpto FROM Empleados WHERE DNI=99);
BEGIN
   OPEN Empl_con99;
   LOOP
     FETCH Empl_con99 INTO UnEmpleado;
     UPDATE Empleados SET Sueldo = Sueldo + (Sueldo*0.1)
       WHERE DNI = UnEmpleado.DNI;
     COMMIT; -- Confirma sólo la última actualización
   END LOOP;
   CLOSE Empl_con99;
END;
```


Subprogramas: *Procedimientos y Funciones*

- **Subprogramas PL/SQL:** Son bloques PL/SQL con nombre que:
  - **Se almacenan en la BD:** Los bloques anónimos (o nominados) no se almacenan.
  - **Se compilan una vez cuando se definen:** Los bloques anónimos (o nominados) se compilan cada vez que se ejecutan.
  - **Pueden ejecutarse o llamarse desde otros bloques PL/SQL.**
  - **Dos Tipos: Procedimientos (PROCEDURE) o Funciones (FUNCTION).**
  - **Borrar un Subprograma: `DROP {PROCEDURE|FUNCTION} <Nombre_S>;`**
- **Procedimientos:** Para crearlos (o reemplazarlos) hay una orden **DDL:**

```
CREATE [OR REPLACE] PROCEDURE <NombreP> [(
   <Argumento1> [IN|OUT|IN OUT] <Tipo1>,
   ...
   <ArgumentoN> [IN|OUT|IN OUT] <TipoN>)] {IS|AS}
<Bloque_del_Procedimiento_SIN_la_palabra_DECLARE>;
```

  - Los **Argumentos** son opcionales y pueden declararse de 3 modos:
    - **`IN`** (opción por defecto)**:** Argumento de **Entrada**. Es de ***sólo lectura***.
    - **`OUT`:** Argumento de **Salida**. Es de ***sólo escritura*** (sólo puede asignarsele valores). Es un paso de parámetros por variable (al terminar el procedimiento se **copia** el valor de ese argumento **en el argumento actual** que aparece en la llamada).
    - **`IN OUT`:** Argumento de **Entrada y Salida**. Es un argumento tipo **`OUT`**, pero que puede también leerse. El argumento actual también debe ser una variable.
  - La **Llamada** se efectúa como en un lenguaje de programación normal.

Procedimientos: Ejemplo

- **Ejemplo:** Programar un Procedimiento para calcular:
  - Salario medio de aquellos **n** empleados que tienen menor sueldo, siendo **n** el primer argumento.
  - Porcentaje que suponen estos **n** empleados respecto al total de empleados de la base de datos.

```
CREATE OR REPLACE PROCEDURE NMedia (P_n      IN  NUMBER,
                                    P_NMedia OUT NUMBER,
                                    P_Porcen OUT NUMBER) IS
  TotalEmpleados NUMBER;
BEGIN
  -- Calcular el porcentaje de n respecto del Total:
  SELECT COUNT(*) INTO TotalEmpleados FROM Empleados;
  P_Porcen := (100 * P_n) / TotalEmpleados;

  -- Salario Medio de los n empleados que menos ganan:
  SELECT AVG(Salario) INTO P_NMedia
    FROM (SELECT Salario FROM Empleados
          ORDER BY Salario)
    WHERE ROWNUM <= n;
END NMedia;
```

- **Observación:** Poner el nombre del subprograma tras la palabra **END** es opcional, pero muy recomendable.

Subprogramas: Argumentos Formales

- **En una Llamada con Argumentos:**
  - **Se pasa el Valor de los Argumentos Actuales (o reales) a los Formales.**
  - **Además, también se pasan los Límites o Restricciones que afecten a las variables de los Argumentos Actuales.**
- **Los Argumentos Formales no pueden tener Restricciones:**
  - **Ejemplo:** No pueden usarse como tipos **VARCHAR2(10)** o **NUMBER(5,2)**. En su lugar se usarán simplemente **VARCHAR2** o **NUMBER** respectivamente.
  - **Errores:** Pueden producirse errores si en la definición del subprograma no se tienen en cuenta las posibles restricciones de los argumentos que se utilicen en la llamada.
    - **Ejemplo:** Se produce un **error** (ORA-6502) si asignamos una cadena de longitud **7** a un argumento formal tipo **OUT** que en la llamada tenga una variable de tipo **VARCHAR2(5)**.
    - Para evitar esos errores se deben documentar, como **comentarios** de cada subprograma, todas las restricciones.
  - **Excepción: Si se usa %TYPE la restricción o limitación será la que indique ese tipo y no la del Argumento Actual.**

Funciones: La orden RETURN

- **Funciones:** Para crearlas (o reemplazarlas) hay una orden **DDL:**

```
CREATE [OR REPLACE] FUNCTION <NombreF> [(
   <Argumento1> [IN|OUT|IN OUT] <Tipo1>,
   ...
   <ArgumentoN> [IN|OUT|IN OUT] <TipoN>)]
   RETURN <TipoRetorno> {IS|AS}
<Bloque_de_la_Función_SIN_la_palabra_DECLARE>;
```

  - Los **Argumentos** son opcionales (también pueden ser de los 3 modos).
  - La **Llamada a una Función:**
    - Se efectúa como en un lenguaje de programación normal.
    - No es una orden PL/SQL como lo es la llamada a Procedimiento.
    - Es una **expresión** que toma el valor que se especifique en el bloque como valor de retorno con:

      RETURN <Expresión_a_Devolver>;

      - La expresión tras **RETURN** debe ser del tipo de retorno especificado.
      - La sentencia **RETURN** es obligatoria. Pueden existir varias pero sólo se ejecutará una de ellas, pues la primera en ejecutarse devuelve el control al entorno que hizo la llamada.
      - La sentencia **RETURN** puede usarse **en un Procedimiento,** sin expresión**,** y sirve sólo para devolver el control a donde se llamó.

Funciones: Ejemplo

- **Ejemplo:** Programar una Función que tenga como argumento el **DNI** de un Empleado. La función **devolverá:**
  - La diferencia entre su Salario y el Salario más grande de entre todos los empleados de su departamento.
  - La función devolverá **-1** si no existe ese empleado.

```
CREATE OR REPLACE FUNCTION DifSalario
  (P_DNI_Empleado IN NUMBER) RETURN NUMBER IS
  SalarioDNI Empleados.Salario%TYPE;
  SalarioMAX Empleados.Salario%TYPE;
  DptoDNI    Empleados.Dpto%TYPE;
BEGIN
  -- Hallar el Salario y Dpto del Empleado en cuestión:
  SELECT Salario, Dpto INTO SalarioDNI, DptoDNI
    FROM Empleados WHERE DNI = P_DNI_Empleado;

  -- Hallar el Salario mayor de entre todos sus colegas:
  SELECT MAX(Salario) INTO SalarioMAX FROM Empleados
    WHERE Dpto = DptoDNI;
  RETURN SalarioMAX - SalarioDNI;
EXCEPTION
  WHEN NO_DATA_FOUND THEN RETURN -1;
END DifSalario;
```

No puede producirse la excepción **TOO_MANY_ROWS** si DNI es llave primaria.

Subprogramas: Observaciones

- **Subprogramas Locales:** Pueden definirse subprogramas al final de la sección declarativa de un bloque PL/SQL (con visibilidad sólo en ese bloque).
- **Diccionario de Datos:** En diversas vistas del diccionario de datos se almacena información sobre los subprogramas:
  - **Vista `USER_OBJECTS`:** Información de todos los objetos pertenecientes al usuario (fecha de creación, última modificación, estado válido o no...).
  - **Vista `USER_SOURCE`:** Código fuente original de cada subprograma. Ejemplo: Para ver el código de un subprograma llamado **`NMedia`**:
    ```
    SELECT text FROM USER_SOURCE WHERE name='NMedia' ORDER BY line;
    ```
  - **Vista `USER_ERRORS`:** Información sobre los errores de compilación.
- **Dependencias:** Un subprograma **"depende directamente"** de todos los objetos que utilice (tablas, otros subprogramas...) y **"depende indirectamente"** de los objetos que utilicen los subprogramas que a su vez necesite utilizar.
  - Una **Orden DDL** sobre un objeto, **puedeinvalidar** todos los objetos que **dependan** de él (directa o indirectamente) y podrán necesitar ser recompilados.
- **Privilegios:** Para que un usuario pueda ejecutar un subprograma ha de concedérsele permiso **`EXECUTE`** sobre ese objeto:
  ```
  GRANT EXECUTE ON <NombreSubprograma> TO <Usuario>;
  ```
  - **Paracompilarse**, un subprograma necesita permisos sobre todos los objetos que utilice. Si no son suyos debe tener concedido el permiso de forma explícita, no a través de un **rol**, ya que un usuario puede desactivar un rol (`SET ROLE`) en una sesión.
  - **Un subprograma se ejecuta como si lo ejecutara su propietario**, por lo que por ejemplo, el subprograma no usará las tablas del usuario que lo ejecute aunque se llamen igual que las tablas del propietario del subprograma.

Paquetes PL/SQL

- **Paquete: Permite almacenar juntos Objetos Relacionados:**
  - **Esos objetos pueden ser de distintos tipos.**
  - **En síntesis, un Paquete es una *sección declarativa con nombre*:**
    - **Cualquier objeto que pueda incluirse en una sección declarativa podrá incluirse en un paquete: Procedimientos, Funciones, Cursores, Excepciones, Tipos (Registros, Tablas...) y Variables.**
    - **Esos objetos pueden usarse en otros bloques PL/SQL y se comportan como *variables globales*.**
  - **Un Paquete tiene Dos partes: Especificación** (o cabecera) **y Cuerpo.**
- **Especificación de un Paquete:**

```
CREATE [OR REPLACE] PACKAGE <NombreP> {IS|AS}
   {<Espec_Procedure> | <Espec_Function> | <Decl_Cursor> |
    <Decl_Exception> | <Def_Tipo> | <Decl_Variable>}
END <NombreP>;
```

  - La sintaxis de los elementos de un paquete sigue las mismas reglas y pueden incluirse todos los objetos que se desee.
  - **Procedimientos y Funciones:** Se incluyen sólo sus declaraciones formales: Sólo la cabecera de cada subprograma acabada en **`;`**

Paquetes PL/SQL

- **Cuerpo de un Paquete:** Contiene el código de los subprogramas.

```
CREATE [OR REPLACE] PACKAGE BODY <NombreP> {IS|AS}
   {<Def_Procedure> | <Def_Function>}
END <NombreP>;
```

  - **Compilación del Cuerpo: NO** puede compilarse si previamente no se ha compilado la **Especificación** del paquete correspondiente.
  - **El Cuerpo es Opcional:** Sólo debe incluirse *obligatoriamente* si la **Especificación** contiene Procedimientos y/o Funciones.
- **Todos los Objetos de la Especificación del Paquete:**
  - Pueden **usarse** en cualquier sitio del **Cuerpo** sin redeclararlos.
  - Son visibles fuera del **Paquete**, por lo que pueden usarse como objetos **globales**.
  - Para **usarlos** se debe **Cualificar cada Objeto** con el nombre del Paquete al que pertenece**: <NombrePaquete>.<NombreObjeto>**
- **Sobrecarga de los Subprogramas de un Paquete:**
  - Pueden definirse **Subprogramas con el mismo Nombre** si tienen **distinto número de argumentos, o éstos son de Tipos de distinta Familia** (no vale **CHAR** y **VARCHAR2**)**.**
    - No vale diferenciar sólo por el modo (**IN OUT**) o el tipo del valor devuelto.
  - Permite utilizar la misma operación a variables de distinto tipo.

Paquetes PL/SQL: Ejemplo

- Dos procedimientos para **Cambiar de Departamento** a un empleado facilitando el nuevo Departamento y su DNI o su Nombre**:**

Especificación

```
CREATE OR REPLACE PACKAGE EmpleadosPKG AS
  -- Cambia el empleado DNI_E al Departamento Dpto_E
  PROCEDURE CambiaDpto (P_DNI_E  IN Empleados.DNI%TYPE,
                        P_Dpto_E IN Empleados.Dpto%TYPE);
  -- Cambia el empleado Nombre_E al Departamento Dpto_E
  PROCEDURE CambiaDpto (P_Nombre_E IN Empleados.Nombre%TYPE,
                        P_Dpto_E   IN Empleados.Dpto%TYPE);
END EmpleadosPKG;
```

Cuerpo

```
CREATE OR REPLACE PACKAGE BODY EmpleadosPKG AS
  PROCEDURE CambiaDpto (P_DNI_E  IN Empleados.DNI%TYPE,
                        P_Dpto_E IN Empleados.Dpto%TYPE) IS
  BEGIN
    UPDATE Empleados SET Dpto=P_Dpto_E WHERE DNI=P_DNI_E;
  END CambiaDpto;

  PROCEDURE CambiaDpto (P_Nombre_E IN Empleados.Nombre%TYPE,
                        P_Dpto_E   IN Empleados.Dpto%TYPE);
  BEGIN
    UPDATE Empleados SET Dpto=P_Dpto_E
      WHERE Nombre=P_Nombre_E;
  END CambiaDpto;
END EmpleadosPKG;
```

OJO: Pueden ser actualizadas varias filas.

Disparadores: TRIGGER

- Es un bloque PL/SQL que **se Ejecuta de forma Implícita** cuando se ejecuta cierta **Operación DML: INSERT, DELETE o UPDATE.**
  - Contrariamente, los procedimientos y las funciones se ejecutan haciendo una llamada **Explícita** a ellos.
  - **Un Disparador No admite Argumentos.**
- **Utilidad:** Sus aplicaciones son inmensas, como por ejemplo:
  - **Mantenimiento de Restricciones de Integridad complejas.**
    - Ej: Restricciones de Estado (como que el sueldo sólo puede aumentar).
  - **Auditoría de una Tabla,** registrando los cambios efectuados y la identidad del que los llevó a cabo.
  - **Lanzar cualquier acción** cuando una tabla es modificada.
- **Formato:**

```
CREATE [OR REPLACE] TRIGGER <NombreT>
   {BEFORE|AFTER} <Suceso_Disparo> ON <Tabla>
   [FOR EACH ROW [WHEN <Condición_Disparo>]]
<Bloque_del_TRIGGER>;
```

  - **<Suceso_Disparo>** es la operación **DML** que efectuada sobre **<Tabla>** disparará el *trigger*.
    - Puede haber varios sucesos separados por la palabra **OR.**

Elementos de un TRIGGER

- **Nombre:** Se recomienda que identifique su función y la tabla sobre la que se define.
- **Tipos de Disparadores:** Hay **12 Tipos** básicos según:
  - **Orden, Suceso u Operación DML: INSERT, DELETE o UPDATE.**
    - Si la orden **UPDATE** lleva una lista de atributos el Disparador sólo se ejecutará si se actualiza alguno de ellos: **UPDATE OF <Lista_Atributos>**
  - **Temporización:** Puede ser **BEFORE (anterior) o AFTER (posterior).**
    - Define si el disparador se activa **antes o después** de que se ejecute la operación **DML** causante del disparo.
  - **Nivel:** Puede ser **a Nivel de Orden** o **a Nivel de Fila (FOR EACH ROW).**
    - **Nivel de Orden** (*statement trigger*)**:** Se activan sólo una vez, antes o después de la **Orden** u operación **DML**.
    - **Nivel de Fila** (*row trigger*)**:** Se activan una vez por cada **Fila** afectada por la operación **DML** (una misma operación **DML** puede afectar a varias filas).
- **Ejemplo:** Guardar en una tabla de control la fecha y el usuario que modificó la tabla `Empleados`: (NOTA: Este trigger es **AFTER** y a nivel de orden)

```
CREATE OR REPLACE TRIGGER Control_Empleados
  AFTER INSERT OR DELETE OR UPDATE ON Empleados
BEGIN
  INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha)
    VALUES ('Empleados', USER, SYSDATE);
END Control_Empleados;
```

Orden en la Ejecución de Disparadores

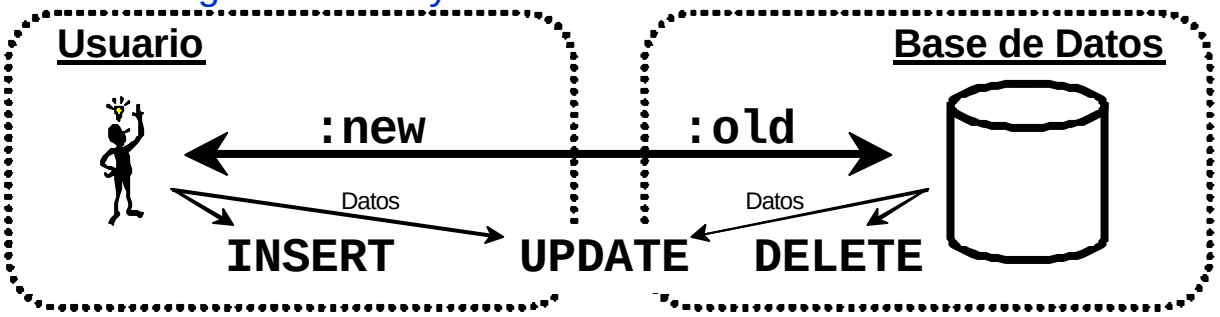
- Una tabla puede tener distintos Tipos de **Disparadores** asociados a una misma **orden DML.**
- En tal caso, el **Algoritmo de Ejecución** es:
  - **1. Ejecutar, si existe, el disparador tipo `BEFORE` a nivel de orden.**
  - **2. Para cada fila a la que afecte la orden:** (esto es como un bucle, para cada fila)
    - **a) Ejecutar, si existe, el disparador `BEFORE` a nivel de fila,** sólo si dicha fila cumple la condición de la cláusula **`WHEN`** (si existe).
    - **b) Ejecutar la propia orden.**
    - **c) Ejecutar, si existe, el disparador `AFTER` a nivel de fila,** sólo si dicha fila cumple la condición de la cláusula **`WHEN`** (si existe).
  - **3. Ejecutar, si existe, el disparador tipo `AFTER` a nivel de orden.**

Disparadores de Fila: `:old` *y* `:new`

- **Disparadores a Nivel de Fila:**
  - Se ejecutan **una vez por cada Fila** procesada por la orden **DML** disparadora.
  - Para acceder a la **Fila Procesada** en ese momento, se usan dos **Pseudo-Registros** de tipo **<TablaDisparo>%ROWTYPE: `:old` y `:new`**

Orden DML	`:old`	`:new`
`INSERT`	**No Definido: NULL** ∀ campo.	**ValoresNuevos** a Insertar.
`UPDATE`	**ValoresOriginales** (antes de la orden).	**Valores Actualizados** (después).
`DELETE`	**ValoresOriginales** (antes de la orden).	**No Definido: NULL** ∀ campo.

  - **Esquema** para la comprensión del significado y utilidad de los Pseudo-Registros `:old` y `:new`



Disparadores a Nivel de Fila: Ejemplo

– **Ejemplo:** Programar un Disparador que calcule el campo código de Pieza (P#) cada vez que se inserte una nueva pieza:

¡OJO: No rellena huecos, si existen.

```
CREATE OR REPLACE TRIGGER NuevaPieza
  BEFORE INSERT ON Pieza FOR EACH ROW
BEGIN
  -- Establecer el nuevo número de pieza:
  SELECT MAX(P#)+1 INTO :new.P# FROM Pieza;
  IF :new.P# IS NULL THEN
     :new.P# := 1;
  END IF;
END NuevaPieza;
```

- Cuando se ejecute la orden **DML** se emplean los valores del **Pseudo-Registro `:new`**. Una orden válida sería la siguiente, sin que produzca un error por no tener la llave: `INSERT INTO Pieza (Nombre, Peso) VALUES ('Alcayata', 0.5);`
  - En este caso, si se suministra un valor para la llave primaria, tampoco producirá un error, pero ese valor será ignorado y sustituido por el nuevo valor calculado por el disparador.

– **Modificar Pseudo-Registros:**
- No puede modificarse el **Pseudo-Registro `:new`** en un disparador **AFTER** a nivel de fila.
- El **Pseudo-Registro `:old`** nunca se modificará: Sólo se leerá.

Disparadores a Nivel de Fila: WHEN

- **Formato: ... FOR EACH ROW WHEN <Condición_Disparo>**
  – Sólo es válida en Disparadores **a Nivel de Fila** y siempre es **opcional**.
  – Si existe, el cuerpo del disparador se ejecutará **sólo para las filas que cumplan la Condición de Disparo** especificada.
  – En la **Condición de Disparo** pueden usarse los Pseudo-Registros **`:old`** y **`:new`**, pero en ese caso **no se escribirán los dos puntos** (`:`), los cuales son obligatorios en el cuerpo del disparador.
  – **Ejemplo:** Deseamos que los precios grandes no tengan más de 1 decimal. Si tiene 2 ó más decimales, redondearemos ese precio:

```
-- Si el Precio>200, redondearlos a un decimal:
CREATE OR REPLACE TRIGGER Redondea_Precios_Grandes
  BEFORE INSERT OR UPDATE OF Precio ON Pieza
  FOR EACH ROW WHEN (new.Precio > 200)
BEGIN
  :new.Precio := ROUND(:new.Precio,1);
END Redondea_Precios_Grandes;
```

- Se puede escribir ese disparador sin la cláusula **`WHEN`**, usando un **`IF`**:

```
... BEGIN
      IF :new.Precio > 200 THEN
         :new.Precio := ROUND(:new.Precio,1);
...   END IF;
```

Predicados INSERTING, DELETING, *y* UPDATING

- En los disparadores que se ejecutan cuando ocurren **diversas Operaciones DML (INSERT, DELETE o UPDATE),** pueden usarse 3 **Predicados Booleanos** para conocer la operación disparadora**:**
  - **INSERTING** Vale **TRUE** si la orden de disparo es **INSERT**.
  - **DELETING** Vale **TRUE** si la orden de disparo es **DELETE**.
  - **UPDATING** Vale **TRUE** si la orden de disparo es **UPDATE**.
- **Ejemplo:** Guardar en una tabla de control la fecha, el usuario que modificó la tabla Empleados y el **Tipo de Operación** con la que modificó la tabla. Usando los Pseudo-Registros **:old** y **:new** también se pueden registrar los valores antiguos y los nuevos (si procede)**:**

```
CREATE OR REPLACE TRIGGER Control_Empleados
  AFTER INSERT OR DELETE OR UPDATE ON Empleados
BEGIN
  IF INSERTING THEN
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'INSERT');
  ELSIF DELETING THEN
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'DELETE');
  ELSE
     INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha,Oper)
       VALUES ('Empleados', USER, SYSDATE,'UPDATE');
END Control_Empleados;
```


Disparadores de Sustitución: INSTEAD OF

- **Disparadores de Sustitución:** Disparador que se ejecuta en lugar de la orden DML (ni antes ni después, sino sustituyéndola).
  - Cuando se intenta **modificar una vista** esta modificación puede no ser posible debido al formato de esa vista.
  - **Características:**
    - **Sólo pueden definirse sobre Vistas.**
    - **Se activan en lugar de la Orden DML** que provoca el disparo, o sea, la orden disparadora no se ejecutará nunca.
    - **Deben tener Nivel de Filas.**
    - Se declaran usando **INSTEAD OF** en vez de **BEFORE/AFTER**.
- **Ejemplo:** Supongamos la siguiente vista**:**

```
CREATE VIEW Totales_por_Suministrador AS
   SELECT S#, MAX(Precio) Mayor, MIN(Precio) Menor
   FROM Suministros SP, Pieza P
   WHERE SP.P# = P.P#  GROUP BY S#;
```

  - Disparador que borre un suministrador si se borra una tupla sobre esa vista:

```
CREATE OR REPLACE TRIGGER Borrar_en_Totales_por_S#
   INSTEAD OF DELETE ON Totales_por_Suministrador
   FOR EACH ROW
BEGIN
   DELETE FROM Suministrador WHERE S# = :old.S#;
END Borrar_en_Totales_por_S#;
```

Disparadores: Observaciones

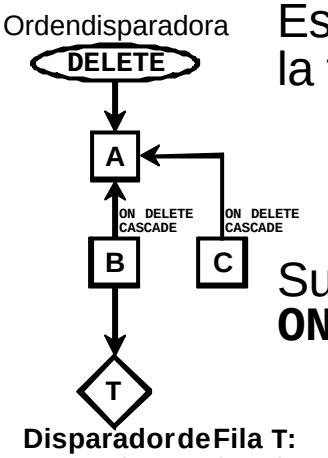
- Un *trigger* puede contener cualquier orden legal en un bloque PL/SQL, con las siguientes **Restricciones:**
  - **No puede emitir órdenes de Control de Transacciones: `COMMIT`, `ROLLBACK` o `SAVEPOINT`.**
    - El disparador se activa como parte de la orden que provocó el disparo y, por tanto, forma parte de la misma transacción. Cuando esa transacción es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.
  - Cualquier **Subprograma** llamado por el disparador **tampoco puede emitir órdenes de control de transacciones.**
  - No puede ni declarar variables ni hacer referencia a columnas de tipo **`LONG`** o **`LONG RAW`** de la tabla sobre la que se define el disparador.
  - **Un disparador puede tener Restringido el acceso a ciertas tablas.**
    - Dependiendo del tipo de disparador y de las restricciones que afecten a las tablas, dichas tablas pueden ser **mutantes**.
- **Diccionario de Datos:** Todos los datos de un **`TRIGGER`** están almacenados en la vista **`USER_TRIGGERS`**, en columnas como **`OWNER`**, **`TRIGGER_NAME`**, **`TRIGGER_TYPE`**, **`TABLE_NAME`**, **`TRIGGER_BODY`**...
- **Borrar un Disparador: `DROP TRIGGER <NombreT>;`**
- **Habilitar/Deshabilitar un Disparador,** sin necesidad de borrarlo:
  `ALTER TRIGGER <NombreT> {ENABLE | DISABLE};`
  - Esto no puede hacerse con los subprogramas.

Disparadores: Tablas Mutantes

- **Tabla de Restricción o Tabla Padre** (*constraining table, Parent*)**:** Tabla a la que referencia una llave externa en una **Tabla Hijo** (*Child*), por una Restricción de Integridad Referencial.
- **Tabla Mutante** (*mutating table*)**:** Una tabla es mutante**:**
  - **1. Si está modificándose "actualmente" por una orden DML (`INSERT`, `DELETE` o `UPDATE`).**
  - **2. Si está siendo leída por Oracle para forzar el cumplimiento de una restricción de integridad referencial.**
  - **3. Si está siendo actualizada por Oracle para cumplir una restricción con `ON DELETE CASCADE`.**
    - **Ejemplo:** Si la cláusula **`ON DELETE CASCADE`** aparece en la tabla **Suministros**, borrar una **Pieza** implica borrar todos sus **Suministros.**
      - **Pieza** (y Suministrador) es una Tabla de Restricción de **Suministros**.
      - Al borrar una pieza ambas tablas serán mutantes, si es necesario borrar suministros. Si se borra una pieza sin suministros sólo será mutante la tabla **Pieza**.
- **Un Disparador de Fila se define sobre una Tabla Mutante** (casi siempre)**.**
  - En cambio, un **Disparador de Orden NO** (casi nunca)**:** El disparador de Orden se ejecuta antes o después de la orden, pero no a la vez.

Disparadores: Tablas Mutantes

- **Las Órdenes SQL en el cuerpo de un disparador tienen las siguientes 2 Restricciones por las que NO PUEDEN:**
  - **1. Leer o Modificar ninguna Tabla Mutante de la orden que provoca el disparo.**



Disparador de Fila T: No puede acceder a las tablas **A**, **B** y **C**, con esa orden disparadora.

Esto incluye lógicamente la tabla de dicha orden y la tabla del disparador, que pueden ser distintas.

Ej.: Borramos una tupla de una tabla **A**. Esto implica borrar tuplas de una tabla **B** (que tiene una restricción **`ON DELETE CASCADE`** sobre **A**). Si este segundo borrado sobre **B** dispara algún disparador **T**, entonces, ese disparador no podrá ni leer ni modificar las tablas **A** y **B** pues son **mutantes**. Si lo hace se producirá un error.

Sus tablas de restricción afectadas por la cláusula **`ON DELETE CASCADE`**, también son mutantes.

Ej.: Si borrar en **A** implica borrar en una tercera tabla **C**, el disparador sobre **B** no podrá tampoco acceder a **C**, si ese disparador se activa por borrar sobre **A** (aunque en **C** no se borre nada). Sí podrá acceder a **C** si se activa por borrar directamente sobre **B**. También podría acceder a **C** si ésta no tuviera el **`ON DELETE CASCADE`** pero hay que tener en cuenta que NO se podrán borrar valores en **A**, si están siendo referenciados en **C** (ORA-2292).

  - **2. Modificar las columnas de clave primaria, única o externa de una Tabla de Restricción a la que la tabla del disparador hacen referencia:** Sí pueden modificarse las otras columnas.

¿Qué Tablas son Mutantes?

- Al escribir disparadores es importante responder a **Dos Preguntas:**
  - **¿Qué Tablas son Mutantes?**
  - **¿Cuando esas Tablas Son mutantes?**
- **¿Qué Tablas son Mutantes?** → Para responder a esa pregunta es necesario conocer el tipo de orden **DML** que se está ejecutando.
  - **1.** Son mutantes las tablas afectadas por una operación **`INSERT`**, **`DELETE`** o **`UPDATE`**.
  - **2.** Si una **tabla Hijo** (p.e. `Empleados`) tiene un atributo llave externa (`Dpto`) a otra **tabla Padre** (`Departamentos`), **ambas** tablas serán mutantes si:
    - **Insertamos** (**`INSERT`**) en la tabla **Hijo:** Comprobar valores en la tabla padre.
    - **Borramos** (**`DELETE`**) de la tabla **Padre:** Impedir que tuplas hijas se queden sin padre.
    - **Actualizamos** (**`UPDATE`**) la tabla **Padre** o la tabla **Hijo:** Las 2 operaciones anteriores.
  - **3.** Si existe la restricción **`ON DELETE CASCADE`**, esta implica que si se borra de la **tabla Padre**, se borrarán las tuplas relacionadas en la **tabla Hijo** y, a su vez, pueden borrarse tuplas de tablas hijos de esa **tabla Hijo**, y así sucesivamente. En ese caso todas esas tablas serán mutantes.
    - En disparadores activados por un **`DELETE`**, es importante tener en cuenta si pueden ser activados por un borrado en cascada y, en ese caso no es posible acceder a todas esas tablas mutantes.

¿Cuándo son las Tablas Mutantes?

- **Las 2 Restricciones anteriores en las órdenes SQL de un Disparador se aplican a:**
  - **Todos los Disparadores con Nivel de Fila.**
    - **Excepción:** Cuando la orden disparadora es un **INSERT** que afecta a una única fila, entonces esa tabla disparadora no es considerada como mutante en el **Disparador de Fila-Anterior**.
      - Con las órdenes del tipo **INSERT INTO Tabla SELECT...** la tabla del disparador será mutante en ambos tipos de disparadores de Fila si se insertan varias filas.
      - Este es el único caso en el que un disparador de Fila puede leer o modificar la tabla del disparador.
  - **Los Disparadores a Nivel de Orden cuando la orden de disparo se activa como resultado de una operación ON DELETE CASCADE (al borrar tuplas en la tabla Padre).**
- **Los ERRORES por Tablas Mutantes se detectan y se generan en Tiempo de Ejecución y no de Compilación** (ORA-4091)**.**

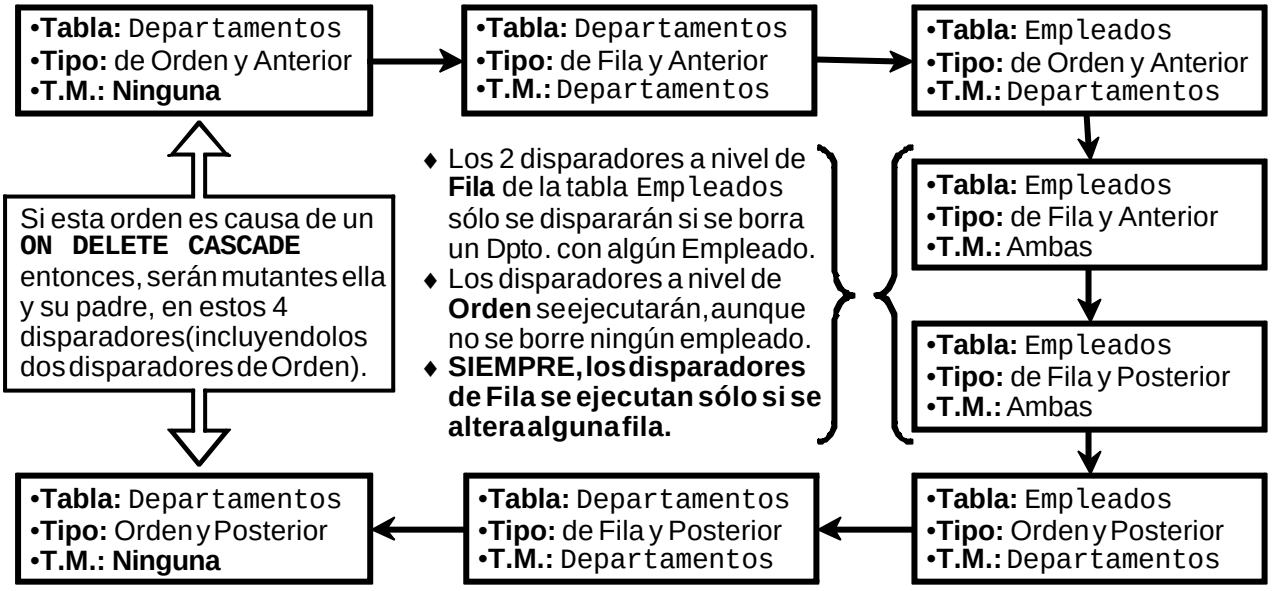
Tablas Mutantes: Disparador de Ejemplo

- **Disparador** que modifique el número de empleados de un departamento (columna `Departamentos.Num_Emp`) cada vez que sea necesario.
  - Ese número cambia al **INSERTAR** o **BORRAR** uno o más empleados, y al **MODIFICAR** la columna `Dpto` de la tabla `Empleados`, para uno o varios empleados.
  - La tabla `Departamentos` es una tabla de restricción de la tabla `Empleados`, pero el Disparador es correcto, porque modifica `Num_Emp`, que no es la llave primaria.
  - Este disparador no puede consultar la tabla `Empleados`, ya que esa tabla es mutante: `SELECT COUNT(*) INTO T FROM Empleados WHERE Dpto = :new.Dpto;`

```
CREATE OR REPLACE TRIGGER Cuenta_Empleados
  BEFORE DELETE OR INSERT OR UPDATE OF Dpto ON Empleados
  FOR EACH ROW
BEGIN
  IF INSERTING THEN
     UPDATE Departamentos SET Num_Emp = Num_Emp+1
       WHERE NumDpto=:new.Dpto;
  ELSIF UPDATING THEN
     UPDATE Departamentos SET Num_Emp = Num_Emp+1
       WHERE NumDpto=:new.Dpto;
     UPDATE Departamentos SET Num_Emp = Num_Emp-1
       WHERE NumDpto=:old.Dpto;
  ELSE
     UPDATE Departamentos SET Num_Emp = Num_Emp-1
       WHERE NumDpto=:old.Dpto;
  END IF;
END;
```

Tablas Mutantes: Ejemplo Esquemático

- Sea una **tabla Padre** `Departamentos`, y una **Hijo** `Empleados` cuyo atributo llave externa es `Dpto` con la restricción **`ON DELETE CASCADE`** que fuerza a borrar todos los empleados de un departamento si su departamento es borrado.
- Supongamos que para la orden **DELETE** están implementados los 4 tipos de disparadores posibles (de fila y de órden, anterior y posterior) para las dos tablas.
- Si se ejecuta una orden **DELETE** sobre la **tabla Padre** `Departamentos`, se ejecutarán los siguientes disparadores, con las tablas mutantes indicadas, en este orden:



Tablas Mutantes: Esquema Padre/Hijo

Órdenes sobre `Departamentos` (Tabla Padre):

DELETE:

Los 4 disparadores de la tabla Hijo `Empleados` sólo se dispararán si se borra un Departamento que tenga algún Empleado.

Si esta orden sobre el Padre es causa de un **`ON DELETE CASCADE`** sobre una tabla **X** (padre del padre) entonces, serán mutantes ella y su padre **X**, en sus 4 disparadores. Ambas tablas también serán mutantes en los 4 disparadores de la tabla Hijo `Empleados`.

INSERT: Su tabla Hijo no se ve afectada: No se disparan sus disparadores.

UPDATE: Su tabla Hijo no se ve afectada, porque sólo se permiten actualizar valores no referenciados en sus tablas Hijos (ORA-2292).

Órdenes sobre `Empleados` (Tabla Hijo): No se dispara ningún disparador del Padre.

DELETE: No afecta a la tabla Padre y sólo es mutante la Hijo.

INSERT:

¨ **Insertar una fila:**

Las tabla Hijo **no** es mutante en el disparador de **Fila-Anterior** (a pesar de ser la tabla del disparador) y **sí** lo es en el de **Fila-Posterior**.

La tabla Padre es mutante sólo si se intenta modificar el atributo al que hace referencia la tabla Hijo y sólo en el disparador de **Fila-Posterior**, ya que durante la ejecución de disparador de Fila-Anterior aún no está mutando ninguna tabla.

¨ **Insertar varias filas:** Las tablas Padre e Hijo son mutantes en los dos disparadores de **Fila**, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

UPDATE: Las tablas Padre e Hijo son mutantes en los dos disparadores de **Fila**, pero la tabla Padre sólo si se modifica el atributo al que hace referencia la tabla Hijo.

Tablas Mutantes: Solución

- El disparador anterior `Cuenta_Empleados` tiene dos inconvenientes:
  - Modifica el campo `Num_Emp` aunque este no tenga el valor correcto.
  - Si se modifica directamente ese campo, quedará incorrecto siempre.
- **Solución al Problema de la Tabla Mutante:**
  - Las tablas mutantes surgen básicamente en los disparadores a nivel de Fila. Como en estos no puede accederse a las tablas mutantes, la **Solución es Crear Dos Disparadores:**
    - **A Nivel de Fila:** En este guardamos los valores importantes en la operación, pero no accedemos a tablas mutantes.
      - Estos valores pueden guardarse en:
        - » **Tablas de la BD** especialmente creadas para esta operación.
        - » **Variables o tablas PL/SQL de un paquete:** Como cada sesión obtiene su propia instancia de estas variables no tendremos que preocuparnos de si hay actualizaciones simultáneas en distintas sesiones.
    - **A Nivel de Orden Posterior (AFTER):** Utiliza los valores guardados en el disparador a Nivel de Fila para acceder a las tablas que ya no son mutantes.

Tablas Mutantes: Solución

```
CREATE OR REPLACE PACKAGE Empleados_Dpto AS
  TYPE T_Dptos IS TABLE OF Empleados.Dpto%TYPE
    INDEX BY BINARY_INTEGER;
  Tabla_Dptos T_Dptos;
END Empleados_Dpto;
```

```
CREATE OR REPLACE TRIGGER Fila_Cuenta_Empleados
  AFTER DELETE OR INSERT OR UPDATE OF Dpto ON Empleados FOR EACH ROW
DECLARE Indice BINARY_INTEGER;
BEGIN Indice := Empleados_Dpto.Tabla_Dptos.COUNT + 1;
      IF INSERTING   THEN
           Empleados_Dpto.Tabla_Dptos(Indice)  := :new.Dpto;
      ELSIF UPDATING THEN
           Empleados_Dpto.Tabla_Dptos(Indice)  := :new.Dpto;
           Empleados_Dpto.Tabla_Dptos(Indice+1):= :old.Dpto;
      ELSE Empleados_Dpto.Tabla_Dptos(Indice)  := :old.Dpto;
      END IF;
END Fila_Cuenta_Empleados;
```

```
CREATE OR REPLACE TRIGGER Orden_Cuenta_Empleados
  AFTER DELETE OR INSERT OR UPDATE OF Dpto ON Empleados
DECLARE Indice BINARY_INTEGER;
        Total   Departamentos.Num_Emp%TYPE;
        Departamento  Departamentos.NumDpto%TYPE;
BEGIN
  FOR Indice IN 1..Empleados_Dpto.Tabla_Dptos.COUNT LOOP
     Departamento := Empleados_Dpto.Tabla_Dptos(Indice);
     SELECT COUNT(*) INTO Total FROM Empleados WHERE Dpto    = Departamento;
     UPDATE Departamentos SET Num_Emp = Total  WHERE NumDpto = Departamento;
  END LOOP;
  Empleados_Dpto.Tabla_Dptos.DELETE;
END Orden_Cuenta_Empleados;
```

Tablas Mutantes: Observaciones

- **Variables Contenidas en un Paquete:**
  - Son Variables Globales de la Sesión.
  - Son visibles para todos los programas de cada sesión.
- **En el Disparador a Nivel de Orden Posterior:**
  - Es necesario **borrar la tabla PL/SQL**, para que la siguiente orden empiece a introducir valores en esa tabla a partir de la posición 1.
    - También podría declararse otra variable en el paquete que mantuviera el número de elementos de la tabla, poniéndola a cero al final. Esto evita tener que usar los atributos de la tabla PL/SQL (**`COUNT`** y **`DELETE`**).
  - Hay que tener en cuenta que se ejecutará **después** de la orden, o sea, que los datos ya estarán actualizados.
  - Si este disparador produce un **error** la orden se deshará (*rollback*). Ese error puede ser:
    - Producido por un error en el disparador o por una operación no válida del disparador.
    - Generado por el disparador de forma explícita porque se haya detectado que la orden disparadora no es admisible (por cualquier razón).
      - Esto se hace con el procedimiento **`RAISE_APPLICATION_ERROR`**.
- **Por supuesto, cualquier orden SQL de cualquier bloque PL/SQL debe Respetar siempre todas las Restricciones de Integridad.**

Tratamiento de Errores: Excepciones

- Todo **buen programa** debe ser capaz de **gestionar TODOS los posibles ERRORES** que puedan surgir durante su ejecución.
  - **PL/SQL** permite hacer una gestión de **errores** basada en **excepciones**, mecanismo heredado del lenguaje Ada.
  - Cuando se produce un **error**, se *genera* una **excepción** y el control del programa pasa al gestor de excepciones (sección **`EXCEPTION`**).
    - Esto permite tratar los errores independientemente, sin oscurecer la lógica del programa y hace que sea más fácil tratar todos los errores.
    - En los lenguajes sin excepciones (como C) es necesario controlarlos explícitamente poniendo condiciones de selección (`if`) antes o después de todas las operaciones que puedan producir errores.
    - **Sección `EXCEPTION`: `WHEN <Excepción> THEN <Gestor>`**
      - Pueden incluirse cuantas cláusulas **`WHEN`** se desee y enlazarlas con **`OR`**.
- **Tipos de Excepciones:** Dos tipos
  - **Predefinidas:** Son las excepciones más comunes en Oracle o SQL y para usarse no es necesario definirlas ni declararlas.
  - **Definidas por el Usuario:** Permiten definir cualquier tipo de error, como por ejemplo un error relacionado con ciertos datos no válidos. Se declaran en la **sección declarativa**:
    - **Ejemplo:**

```
DECLARE
  MuchosEmpleados EXCEPTION;
```

Excepciones Predefinidas

- Hay **20 Excepciones Predefinidas** que controlan errores particulares (excepto **OTHERS** que controla cualquier tipo de error). Algunas son**:**
  - **INVALID_CURSOR:** Se genera al intentar efectuar una operación ilegal sobre un cursor, como cerrar o intentar extraer datos de un cursor no abierto.
  - **CURSOR_ALREADY_OPEN:** Surge al intentar abrir un cursor ya abierto.
  - **NO_DATA_FOUND:** Cuando una orden **SELECT..INTO** no devuelve ninguna fila o cuando se intenta referenciar un elemento de una tabla PL/SQL al que no se le ha asignado ningún valor previamente.
  - **TOO_MANY_ROWS:** Si una orden **SELECT..INTO** devuelve más de una fila.
  - **INVALID_NUMBER:** Si falla la conversión de cierto valor a un tipo **NUMBER** o cuando usamos un dato no numérico en lugar de un dato numérico.
  - **VALUE_ERROR:** Se genera cada vez que se produce un error aritmético, de conversión, de truncamiento o de restricciones en una orden procedimental (si es una orden SQL se produce la excepción **INVALID_NUMBER**). Ej.: Si asignamos una cadena o número de mayor longitud que el tipo de la variable receptora.
  - **STORAGE_ERROR y PROGRAM_ERROR:** Son errores internos que no deberían producirse. Ocurren respectivamente si PL/SQL se queda sin memoria o por un fallo en el motor PL/SQL de Oracle y debería avisarse del error al departamento de soporte técnico de Oracle.
  - **DUP_VAL_ON_INDEX:** Es el error ORA-1, que se genera cuando se intenta insertar una fila en una tabla con un atributo **UNIQUE** y el valor de ese campo en la fila que se intenta insertar ya existe.
  - **ZERO_DIVIDE:** Surge al intentar dividir por cero.

Generación de Excepciones

- Cuando se **Genera una Excepción** el control pasa a la sección **EXCEPTION** del bloque PL/SQL y resulta imposible devolver el control a la sección principal del bloque.
  - Las **Excepciones Predefinidas** se generan automáticamente cuando ocurre el error Oracle asociado a cada una.
  - Las **Excepciones Definidas por el Usuario** hay que generarlas explícitamente mediante la orden**:** **RAISE <Excepción>;**
- **Ej.:**

```
DECLARE
   Demasiados_Empleados EXCEPTION;
   Num_Empleados        NUMBER(3);
   Depart               Empleados.Dpto%TYPE;
BEGIN
   SELECT Dpto INTO Depart FROM Empleados WHERE DNI=99;
   SELECT COUNT(*) INTO Num_Empleados FROM Empleados
          WHERE Dpto = Depart;
   IF Num_Empleados > 20 THEN
      RAISE Demasiados_Empleados; -- Generar Excepción
   END IF;
EXCEPTION
   WHEN Demasiados_Empleados THEN
    INSERT INTO Tabla_Avisos(msg) VALUES ('Hay ' ||
      Num_Empleados||' empleados en el Dpto. '||Depart);
   WHEN NO_DATA_FOUND THEN
    INSERT INTO Tabla_Avisos(msg) VALUES
    ('No existe el empleado con DNI 99');
END;
```

El Gestor de Excepciones OTHERS

- **El Gestor OTHERS se ejecuta (si existe) para todas las excepciones que se produzcan:**
  - Debe ser siempre **el último gestor de excepciones**, ya que tras él no se ejecutará ninguno.
  - Es una buena práctica definir el gestor **OTHERS en el bloque más externo**, para asegurarse de que ningún error queda sin detectar.
  - **Funciones para Determinar el Error** (no pueden usarse en órdenes SQL)**:**
    - **SQLCODE:** Devuelve el **código del error** actual.
    - **SQLERRM:** Devuelve el **mensaje de error** actual (máxima longitud: 512 caracteres). Admite opcionalmente un código como argumento.
- **Ej.:**

```
DECLARE ...
  Codigo_Error NUMBER;
  Msg_Error    VARCHAR2(200);
BEGIN
  ...
EXCEPTION
  WHEN ... THEN
    ...
  WHEN OTHERS THEN
    Codigo_Error := SQLCODE;
    Msg_Error := SUBSTR(SQLERRM,1,200);
    INSERT INTO Tabla_Avisos (Cod_Error, Msg) VALUES
      (Codigo_Error, Msg_Error);
END;
```

- Si la asignación es superior al espacio reservado para la variable `Msg_Error` se generará la excepción **VALUE_ERROR**.
- También hay que asegurarse que el valor insertado en `Tabla_Avisos` no excede de la longitud máxima.

RAISE_APPLICATION_ERROR

- **RAISE_APPLICATION_ERROR permite que un programa PL/SQL pueda Generar Errores tal y como lo hace Oracle:**
  - Cuando se produce un error no tratado en la sección **EXCEPTION**, el error pasa fuera del bloque, al entorno que realizó la llamada.
  - Con **RAISE_APPLICATION_ERROR** se pueden generar errores similares con el mensaje que se quiera y, como ocurre con los errores generados por Oracle, **el programa genera una EXCEPCIÓN:**
    - La excepción puede tratarse en la sección **EXCEPTION** del bloque PL/SQL que la genera o del bloque que efectúe su llamada, usando el manejador **OTHERS** y **SQLCODE**/**SQLERRM**.
  - **Formato: RAISE_APPLICATION_ERROR(<NE>, <ME>, [<PE>]);**
    - **<NE>: Número del Error**, comprendido entre –20.000 y –20.999.
    - **<ME>: Mensaje del Error**, de longitud máxima 512 caracteres.
    - **<PE>: Preservar Errores** es un valor lógico opcional. Indica si el error se introduce a la lista de errores ya generados (**TRUE**) o sustituye la lista actual con el nuevo error (**FALSE**, valor predeterminado).
  - Permite generar errores con mensajes más significativos que los que generaría Oracle: Puede utilizarse en la sección **EXCEPTION**.
  - Hace que requieran el mismo tratamiento los errores definidos por el usuario y los errores predefinidos.

RAISE_APPLICATION_ERROR

- **Ejemplo:** Procedimiento para cambiar de Dpto. a un empleado dando su DNI y su Dpto. de destino. Si en el nuevo Dpto. hay más de 20 empleados o no existe dicho empleado, generar sendos errores.

```
CREATE OR REPLACE PROCEDURE CambiaDpto(
    P_DNI  Empleados.DNI%TYPE,
    P_Dpto Empleados.Dpto%TYPE) AS
  Num_Empleados NUMBER(3);
BEGIN
  SELECT COUNT(*) INTO Num_Empleados FROM Empleados
    WHERE Dpto = P_Dpto;
  IF Num_Empleados+1 > 20 THEN
     RAISE_APPLICATION_ERROR(-20000, 'Dpto. ' || P_Dpto
       || ' totalmente ocupado');
  ELSE
    UPDATE Empleados SET Dpto=P_Dpto
      WHERE DNI = P_DNI;
    IF SQL%NOTFOUND THEN
      RAISE_APPLICATION_ERROR(-20001,
        'No existe Empleado con DNI = ' || P_DNI);
    END IF;
  END IF;
END CambiaDpto;
```


Propagación de Excepciones

- **Anidamiento de Bloques:**
  - Un bloque que contiene a otro se dice que es de **nivel superior**.
  - Un bloque contiene a otro porque esté definido uno dentro de otro o porque uno efectúe una llamada a un procedimiento o función.
- **Propagación de Excepciones:**
  - **Cuando se Genera una Excepción en la Sección Ejecutable:**
    - **1.** Si el bloque actual **tiene un gestor** para la excepción, pasa el control a ese gestor y, al terminar, pasa el control al bloque superior (si existe).
    - **2.** Si el bloque actual **NO tiene un gestor** para la excepción, **propaga** la excepción al bloque de nivel superior y se aplica el paso **1.**
  - **Cuando se Genera una Excepción en la Sección Declarativa:**
    - Pasa el control directamente al bloque de nivel superior, ya que el bloque que genera la excepción no ha comenzado a ejecutarse.
    - Ej. que genera la excepción **VALUE_ERROR:** `Aux NUMBER(2):='abc';`
  - **Cuando se Genera una Excepción en la Sección de Excepciones:**
    - Pasa el control directamente al bloque de nivel superior, ya que sólo puede haber una excepción activa en cada momento.
    - Dentro de un Gestor de Excepciones puede usarse **RAISE** sin argumentos para generar de nuevo la misma excepción y pasar la misma excepción al bloque de nivel superior.
- Si finalmente la excepción no es tratada se propaga fuera del entorno que realizó la llamada y éste debería dárle algún tratamiento (usualmente mostrar el mensaje de error al usuario).

Localización de Errores

- Cuando en un mismo bloque PL/SQL puede generarse una misma excepción en distintas órdenes, ¿Cómo detectar cual ha sido?
  - **Ejemplo:**

```
BEGIN
    SELECT ...
    SELECT ...
    ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    -- ¿Qué orden SELECT generó el error?
END;
```

  - **DosSoluciones:** Usar una variable o poner cada orden en su propio bloque.

```
DECLARE
  Contador NUMBER(1) := 1;
BEGIN
  SELECT ...
  Contador:=2;
  SELECT ...
...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    IF Contador=1 THEN
      ...
    ELSE
      ...
    END IF;
END;
```

```
BEGIN
  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;
  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;
  ...
END;
```


Métodos de Prueba y Depuración

- **Prueba:** Cualquier programa (incluidos los bloques PL/SQL) debe **Probarse** exhaustivamente para asegurar su correcto funcionamiento.
  - Las **Pruebas** se hacen en un entorno especial de **Desarrollo** y no de **Producción**, que debe ser suficientemente simple.
- **Depuración:** Si existen **Errores:**
  - **Localizar** dónde está el error y **Definir** qué está mal y porqué está fallando.
  - **Dividir** el programa en partes y **Probar** cada una de ellas, empezando por casos de prueba sencillos.
- **Técnicas de Depuración:**
  - **Insertar** los valores intermedios de las variables en una tabla temporal, que se consultará cuando el programa termine.
    - Existe un paquete llamado **DEBUG** que facilita esta tarea. Tiene dos procedimientos:
      - **DEBUG.Debug(A,B)** → **Inserta** en la tabla **Debug_Table** esos dos valores de tipo **VARCHAR2** en una fila (usualmente **A** es el nombre de una variable y/o algún comentario sobre el lugar donde se utiliza y **B** es el valor de esa variable).
      - **DEBUG.Reset** → Inicializa la depuración (borra las filas de **Debug_Table**).
  - **Visualizar** los valores intermedios de las variables, conforme el programa se va ejecutando: Paquete **DBMS_OUTPUT** y sus procedimientos **PUT**, **PUT_LINE**...
    - Esos procedimientos insertan sus argumentos en un buffer interno.
    - En SQL*Plus puede activarse para mostrar el buffer: **set serveroutput on [size n]**, donde **n** Î **[2000,1000000]** es el número de bytes máximo (**2000** por defecto).
    - PL/SQL es un lenguaje para gestionar BD, por lo que la E/S es limitada.
  - **Depuradores PL/SQL:** Existen varios programas *debuggers* para rastrear un programa PL/SQL, como son **OracleProcedureBuilder** y **SQL-Station**.