Introducción.

El trabajo realizado hasta ahora con la base de datos se ha hecho de forma interactiva: el usuario introducía una orden (en SQL) y Oracle proporcionaba una respuesta. Esta forma de trabajar no resulta operativa en un entorno de producción, porque todos los usuarios no conocen ni utilizan SQL, y además suelen producirse frecuentes errores.

Para superar estas limitaciones, Oracle incorpora un gestor PL/SQL en el servidor de la BD y en algunas de sus herramientas (Forms, Reports, Graphics, etc.). Este lenguaje incorpora todas las características propias de los lenguajes de tercera generación: manejo de variables, estructura modular (procedimientos y funciones), estructuras de control (bifurcaciones, bucles y demás estructuras), control de excepciones, y una total integración en el entorno Oracle.

Los programas creados con PL/SQL se pueden almacenar en la base de datos como un objeto más de ésta; así se facilita a todos los usuarios autorizados el acceso a estos programas, y en consecuencia, la distribución, instalación y mantenimiento de software. Además, los programas se ejecutan en el servidor, suponiendo un significativo ahorro de recursos en los clientes y de disminución del tráfico de red.

El uso del lenguaje PL/SQL es también imprescindible para construir disparadores de bases de datos, que permiten implementar reglas complejas de negocio y auditoria en la BD.

PL/SQL soporta todos los comandos de consulta y manipulación de datos, aportando sobre SQL las estructuras de control y otros elementos propios de los lenguajes procedimentales de tercera generación. Su unidad de trabajo es el bloque, formado por un conjunto de declaraciones, instrucciones y mecanismos de gestión de errores y excepciones.

Transacciones.

Oracle es un sistema de base de datos puramente transaccional, de tal forma, que la instrucción BEGIN TRANSACTION no existe.

Una transacción es un conjunto de sentencias SQL que se ejecutan en una base de datos como una única operación, confirmándose o deshaciéndose todo el conjunto de sentencias SQL. La transacción puede quedar finalizada (con las sentencias apropiadas) o implícitamente (terminando la sesión).

Durante la transacción, todas las modificaciones que hagamos sobre base de datos, no son definitivas, más concretamente, se realizan sobre un tablespace especial que se denomina tablespace de ROLLBACK, o RBS (RollBack Segment). Este tablespace tiene reservado un espacio para cada sesión activa en el servidor, y es en ese espacio donde se almacenan todas las modificaciones de cada transacción. Una vez que la transacción se ha finalizado, las modificaciones temporales almacenadas en el RBS, se vuelcan al tablespace original, donde está almacenada nuestra tabla. Esto permite que ciertas modificaciones que se realizan en varias sentencias, se puedan validar todas a la vez, o rechazar todas a la vez.

Dentro de una transacción se pueden crear los llamados “punto de control” mediante la sentencia:

```
SAVEPOINT Nombre_punto_control
```

Las sentencias de finalización de transacción son:

- **COMMIT**: la transacción termina correctamente, se vuelcan los datos al tablespace original y se vacía el RBS.

- **ROLLBACK**: se rechaza la transacción y el vacía el RBS. Cualquier cambio realizado desde que se inició la transacción se deshace, quedando la base de datos en el mismo estado que antes de iniciarse la transacción.

A la hora de hacer un ROLLBACK o un COMMIT se podrá hacer hasta cierto punto con la sintaxis:

```
COMMIT TO punto_control;
ROLLBACK TO punto_control;
```

Cuando tenemos abierta una sesión (WorkSheet), los cambios que realizamos no son visibles a otra sesión hasta que no hagamos un COMMIT. Este se puede realizar de forma manual, ejecutando el comando COMMIT; o bien, de forma automática, cuando cerramos la sesión.

En una transacción los datos modificados no son visibles por el resto de usuarios hasta que se confirme la transacción.

Si alguna de las tablas afectadas por la transacción tiene triggers, las operaciones que realiza el trigger están dentro del ámbito de la transacción, y son confirmadas o deshechas conjuntamente con la transacción. Durante la ejecución de una transacción, una segunda transacción no podrá ver los cambios realizados por la primera transacción hasta que esta se confirme.

El siguiente ejemplo muestra una supuesta transacción bancaria:

```
DECLARE
        importe NUMBER; ctaOrigen
        VARCHAR2(23); ctaDestino
        VARCHAR2(23);
BEGIN
        importe := 100;
        ctaOrigen := '2530 10 2000 1234567890';
        ctaDestino := '2532 10 2010 0987654321';

        UPDATE CUENTAS SET SALDO = SALDO – importe WHERE CUENTA = ctaOrigen; UPDATE
        CUENTAS SET SALDO = SALDO + importe WHERE CUENTA = ctaDestino;
        INSERT INTO MOVIMIENTOS VALUES
                (ctaOrigen, ctaDestino, importe*(-1), SYSDATE);
        INSERT INTO MOVIMIENTOS VALUES
                (ctaDestino,ctaOrigen, importe, SYSDATE);
        COMMIT;
EXCEPTION
        WHEN OTHERS THEN
        dbms_output.put_line('Error en la transaccion:'||SQLERRM);
        dbms_output.put_line('Se deshacen las modificaciones);
        ROLLBACK;
END;
```

Programación PL/SQL.

SQL es un lenguaje de consulta para los sistemas de bases de datos relaciónales, pero que no posee la potencia de los lenguajes de programación.

PL/SQL (Procedural Language/Structured Query Language) apareció por primera vez en la versión 6 de Oracle (1988) y amplia SQL con los elementos característicos de los lenguajes de programación: variables, sentencias de control de flujo, bucles...

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona **Oracle** para extender el SQL estándar con otro tipo de instrucciones.

Para poder trabajar necesitaremos tener los siguientes elementos:

- Una instancia de ORACLE o superior funcionando correctamente.
- Herramientas cliente de ORACLE, (WorkSheet o SQL*Plus) para poder ejecutar los ejemplos.
- Haber configurado correctamente una conexión a ORACLE.

Con PL/SQL vamos a poder programar las unidades de programa de la base de datos ORACLE, están son:

- Procedimientos almacenados
- Funciones
- Triger
- Scripts

Pero además PL/SQL nos permite realizar programas sobre las siguientes herramientas de ORACLE:

- Oracle Forms
- Oracle Reports
- Oracle Graphics
- Oracle Aplication Server

Fundamentos de PL/SQL

Para programar en PL/SQL es necesario conocer sus fundamentos.

Como introducción vamos a ver algunos elementos y conceptos básicos del lenguaje.

- PL/SQL **no** es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas como otros lenguajes de programación como C o Java. **Sin embargo debemos recordar que ORACLE es CASE-SENSITIVE en la búsqueda de texto.**
- Una línea en PL/SQL contiene grupos de caracteres conocidos como UNIDADES LEXICAS, que pueden ser clasificadas como:

  DELIMITADOR: Es un símbolo simple o compuesto que tiene una función especial en PL/SQL. Estos pueden ser:

  - Operadores Aritméticos
  - Operadores Lógicos

- Operadores Relaciónales

IDENTIFICADOR: Se emplean para dar nombre a los objetos PL/SQL, tales como variables, cursores, tipos y subprogramas.

Los identificadores constan de una **letra**, seguida por una secuencia opcional de caracteres, que pueden incluir **letras**, **números**, signos de dólar (**$**), caracteres de **subrayado** y símbolos de almohadilla (**#**). Los demás caracteres no pueden emplearse. La longitud **máxima** de un identificador es de **30 caracteres** y todos los caracteres son significativos.

LITERAL: Es un valor de tipo numérico, carácter, cadena o lógico no representado por un identificador (es un valor explícito).

COMENTARIO: Es una aclaración que el programador incluye en el código. Son soportados 2 estilos de comentarios, el de línea simple y de multilínea, para lo cual son empleados ciertos caracters especiales como son:

```
-- Linea simple
/*
Conjunto de Líneas
*/
```

- Cuando se escribe código en PL/SQL, este puede estar agrupado en unidades denominadas "conjunto de instrucciones". Un conjunto de instrucciones puede contener otros subconjuntos y así sucesivamente.

Un conjunto de instrucciones queda delimitado por las palabras reservadas BEGIN y END.

```
BEGIN
        Sentencias . . .
        Sentencias . . .
        BEGIN
                Sentencias . . .
                Sentencias . . .
                Sentencias . . .
        END;
        Sentencias . . .
        Sentencias . . .
END;
```

Bloques PL/SQL

Los bloques PL/SQL son de varios tipos:

- **Anónimos (Anonymous blocks**). Se construyen de forma dinámica y se ejecutan una sola vez.

- **Con nombre (Named blocks**). Son bloques con nombre, que al igual que el anterior se construyen, generalmente, de forma dinámica y se ejecutan una sola vez.

- **Subprogramas**. Procedimientos, paquetes o funciones almacenados en la BD. No suelen cambiar después de su construcción y se ejecutan múltiples veces mediante una llamada call.

- **Disparadores(Triggers)**. Son bloques con nombre que también se almacenan en la BD. Tampoco suelen cambiar después de su construcción y se ejecutan varias veces. Se ejecutan *de forma automática ante algún suceso de disparo*, que será una orden del lenguaje de

manipulación de datos (INSERT, UPDATE o DELETE) que se ejecuta sobre una tabla de la BD.

Los bloques PL/SQL presentan una estructura específica compuesta de tres partes bien diferenciadas:

- La **sección declarativa** en donde se declaran todas las constantes y variables que se van a utilizar en la ejecución del bloque. Esta sección es opcional.
- La **sección de ejecución** que incluye las instrucciones a ejecutar en el bloque PL/SQL. Estas instrucciones pueden ser tanto de tipo DML como DDL, así como ordenes procedimentales. Esta es la única sección que es obligatoria.
- La **sección de excepciones** en donde se definen los manejadores de errores que soportará el bloque PL/SQL. Esta sección es opcional y no se ejecutará a menos que aparezca un error.

Cada una de las partes anteriores se delimita por una palabra reservada, de modo que un bloque PL/SQL se puede representar como sigue:

```
[DECLARE
        Declaración de variables] /*Parte declarativa*/
BEGIN
        Sentencias SQL y PL/SQL /*Parte de ejecucion*/
[EXCEPTION
        Manejadores de excepciones] /*Parte de excepciones*/
END;
```

Un bloque anónimo es aquel bloque que no tiene asignado un nombre.

```
SET SERVEROUTPUT ON;

DECLARE
 A VARCHAR(10) := '';

BEGIN
  SELECT TO_CHAR(SYSDATE) INTO A FROM DUAL;
  DBMS_OUTPUT.PUT_LINE('LA FECHA ACTUAL ES : ' || A);

EXCEPTION
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('HOLA');
END;
```

Para que la salida pueda verse al ejecutar el programa tiene que estar activa la siguiente variable:

```
SET SEVEROUTPUT ON;
```

Para mostrar el contenido de una expresión se debe utilizar la sentencia:

```
DBMS_OUTPUT.PUT_LINE (cadena_caracteres)
```

Declaración de variables

Las variables deben declararse dentro de la sección DECLARE y deben seguir la siguiente sintaxis:

```
Nombre_variable [CONSTANT] TIPO [NOT NULL] [:= inicialización];
```

Cualquier variable que se declare y no se inicialice tiene por defecto el valor NULL. Los tipos posibles son todos aquellos válidos para SQL añadiendo algunos propios de PL/SQL. Para más información sobre los tipos propios de PL/SQL consultar el ***PL/SQL User's Guide and Referente.*** Podemos hacer que una variable nunca tome valores nulos utilizando la cláusula NOT NULL, en este caso, hay que inicializar la variable.

La declaración de una constante es similar a la declaración de una variable, añadiendo la palabra CONSTANT y asignándole a continuación un valor a la constante.

Ejemplos:

```
Interes NUMBER(5,3);
Descripcion VARCHAR2(50) := 'inicial';
Fecha_max DATE;
Contabilizado BOOLEAN := TRUE;
PI CONSTANT REAL := 3.14159
```

Otra forma de asignarle un valor a una variable es mediante la clausula INTO de la sentencia SELECT:

```
SELECT COUNT(*) INTO xNumFac FROM FACTURAS;
```

Atributos %TYPE y **%ROWTYPE**.

Se puede declarar el tipo de una variable tomándolo de otro identificador, usando el atributo **%TYPE** y se puede declarar el tipo de una variable también cuando es un tipo estructurado con el atributo **%ROWTYPE**. Esto es particularmente útil cuando una variable va a tomar valores de una columna de una tabla. Declarar variables con el atributo %TYPE tiene dos ventajas. Primero, no necesitamos conocer el tipo exacto de la columna de la tabla. Segundo, si cambiamos la definición y/o tipo de la columna de la tabla, el tipo de la variable cambia automáticamente en tiempo de ejecución.

En la declaración: si tenemos una variable " y " por ejemplo, y está declarada de tipo char podemos declarar otra variable "j" de la siguiente forma:

```
J y%type;
```

Lo mismo ocurriría para declarar una estructura que ya esta declarada como por ejemplo una tabla que ya tenemos declarada. Ejemplo:

```
J employee%rowtype  J tendría la misma estructura que la tabla employee.
```

En este caso para acceder a cada campo que tuviera el tabla employee mediante la variable J tendríamos que usar la estructura variable.nombre_campo.

Un bloque tiene acceso a los objetos identificados dentro de su esquema. Solo podremos acceder a los objetos del usuario donde estemos conectados y a los que ese usuario pueda acceder porque le hayan otorgado permisos.

Estructuras básicas de control

Como PL/SQL es un lenguaje 3GL, cuenta con las estructuras típicas de control de flujo: bifurcaciones condicionales y bucles:

Bifurcaciones condicionales:

a. IF.

La sintaxis básica es:

```
IF condición THEN
        --Se ejecuta si se cumple
        condición Bloque de instrucciones;
[ELSIF condición THEN
        --Se ejecuta si no se cumple
        condición Bloque de instrucciones;]
. . .
[ELSE
        --Se ejecuta si no se cumple
        condición Bloque de instrucciones;]
END IF;
```

Como en cualquier lenguaje de programación, condición es cualquier expresión que de cómo resultado un valor booleano. Hay que saber que las estructuras IF se pueden anidar unas dentro de otras.

a. IF –THEN

Se evalúa la condición y si resulta **verdadera**, se ejecutan uno o más líneas de código de programa. En el caso de que la condición resulte **falsa o nula**, NO se realiza NINGUNA acción.

```
IF fecha_nac < '1-01-1970' THEN  --No termina con un ;
    Salario := salario *1.15;    --aumento de salario en un 15%
END IF;
```

Se pueden anidar varias instrucciones:

```
IF fecha_nac < '1-01-1970' THEN
    IF apellido ='Martínez' THEN   --IF ANIDADO
    salario:= salario *1.15;       --aumento de salario en un 15%
    END IF;                        --END IF OBLIGATORIO
END IF;
```

b. IF –THEN –ELSE

Se evalúa la condición y si resulta **verdadera**, se ejecutan uno o más líneas de código de programa. En el caso de que la condición resulte **falsa**, se ejecutan las instrucciones que siguen a la instrucción ELSE. Sólo se permite una instrucción ELSE en cada instrucción IF.

```
IF fecha_nac <'1-01-1970' THEN   --No termina con un ;
    salario:= salario *1.15;     --aumento de salario en un 15%
ELSE                                    --No termina con un ;
    salario:= salario* 1.05;     --aumento de salario en un 5%
END IF;
```

c. IF –THEN –ELSIF

Se evalúa la condición y si resulta **verdadera**, se ejecutan uno o más líneas de código de programa. En el caso de que la condición resulte ser **falsa**, se evalúa la condición especificada en el ELSIF.

```
IF condition1 THEN
   statement1;
ELSE
   IF condition2 THEN
      statement2;
   ELSE
      IF condition3 THEN
         statement3;
      END IF;
   END IF;
END IF;
```

```
IF condition1 THEN
   statement1;
ELSIF condition2 THEN
   statement2;
ELSIF condition3 THEN
   statement3;
END IF;
```

```
IF apellido =„Pérez“ THEN
    salario:= salario *1.10; --aumento de salario en un
10% ELSIF apellido =‘Martínez’ THEN
    salario:= salario *1.15; --aumento de salario en un
15% ELSIF apellido=‘Alvarez’ THEN
    salario:= salario *1.20;   --aumento de salario en un 20%
ELSE
    salario:= salario* 1.05;   --aumento de salario en un 5%
END IF;                                --Sólo se necesita un único END IF
```

b. CASE

La instrucción CASE puede evaluar múltiples expresiones y devolver para cada una de ellas un valor/bloque de instrucciones. El resultado de cada WHEN puede ser un valor o una sentencia; en el primer caso el resultado de una sentencia CASE se puede guardar en una variable.

Su sintaxis es:

```
CASE variable
    WHEN expresión1 THEN valor1/bloque de instrucciones
    WHEN expresión2 THEN valor2/bloque de instrucciones
    WHEN expresión3 THEN valor3/bloque de instrucciones
    WHEN expresión4 THEN valor4/bloque de instrucciones
    ELSE valor5/bloque de instrucciones
END
```

Ejemplos:

```
CREATE TABLE C2
(
 Nombre     VARCHAR2(20 ),
 EC         VARCHAR2(1)
);
COMMIT;
INSERT INTO C2 ( NOMBRE, EC ) VALUES ('Juan ', 'S');
INSERT INTO C2 ( NOMBRE, EC ) VALUES ('Maria', 'C');
INSERT INTO C2 ( NOMBRE, EC ) VALUES ('Ana', 'D');
INSERT INTO C2 ( NOMBRE, EC ) VALUES ('Luis', 'S');
INSERT INTO C2 ( NOMBRE, EC ) VALUES ('Pepe', NULL);
COMMIT;
```

```
SELECT Nombre, CASE EC
                    WHEN 'C' THEN 'Casado/a'
                    WHEN 'S' THEN 'Soltero/a'
                    WHEN 'D' THEN 'Divorciado/a'
                    ELSE 'Otros'
               END
               AS "Estado Civil"
FROM C2;
```

Otra sintaxis es:

```
CASE
    WHEN condición1 THEN expresión1/bloque de instrucciones
    WHEN condición2 THEN expresión2/bloque de instrucciones
    WHEN condición3 THEN expresión3/bloque de instrucciones
    WHEN condición4 THEN expresión4/bloque de instrucciones
    ELSE expresión5/bloque de instrucciones
END
```

En esta sintaxis después del CASE no aparece ninguna variable y cada WHEN tiene su propia condición a evaluar.

Bucles

```
LOOP
        sentencias
END LOOP;
```

Las sentencias de dentro del bucle se ejecutarán durante un número indefinido de vueltas, hasta que aparezca la instrucción EXIT; que finalizará el bucle. Este tipo de bucle se denomina bucle incondicional.

```
LOOP
        Sentencias
        IF (expresion) THEN
                Sentencias
                EXIT;
        END IF;
END LOOP;
```

Otra opción es incluir la estructura EXIT WHEN condición, se terminará el bucle cuando la condición se cumpla:

```
LOOP
        Sentencias
EXIT WHEN condición;
        Sentencias
END LOOP;
```

Ejemplo

```
DECLARE -- declare and assign values to
  variables total NUMBER(9) := 0;
  counter NUMBER(6) := 0;
BEGIN
  LOOP
```

```
  counter := counter + 1; -- increment counter variable
  total := total + counter * counter; -- compute total
  -- exit loop when condition is true
  EXIT WHEN total > 25000; -- LOOP until condition is met
 END LOOP;
 DBMS_OUTPUT.PUT_LINE('Counter: ' || TO_CHAR(counter)
            || ' Total: ' || TO_CHAR(total));
END;
```

Un tipo de bucle más común son los bucles condicionales:

```
WHILE condicion LOOP
        Sentencias
END LOOP;
```

Ejemplo

```
DECLARE
   i            NUMBER := 1;
   i_cubed   NUMBER;
BEGIN
  WHILE i <= 10 LOOP
   i_cubed := i**3;
   DBMS_OUTPUT.PUT_LINE('Number: ' || TO_CHAR(i) ||
              ' Cube: ' || TO_CHAR(i_cubed));
   i := i + 1;
 END
LOOP; END;
```

En los bucles WHILE también se pueden utilizar las órdenes EXIT o EXIT WHEN para salirnos sin esperar a que la condición devuelva un valor falso.

Y por último el bucle FOR:

```
FOR contador IN [REVERSE] limite_inferior..limite_superior
        LOOP sentencias
END LOOP;
```

Contador deberá ser una variable de tipo numérico que sea capaz de contener los valores comprendidos entre limite_inferior y limite_superior, los cuales deberán ser expresiones numéricas, ya sean constantes (1,10…) o funciones (ROUND(max,0), ASCII('A')…) .

Si la variable contador no está definida, PL/SQL definirá una variable de tipo INTEGER al iniciar el bucle, y la liberará al finalizar el bucle.

```
SET SERVEROUTPUT ON;
BEGIN
  FOR loop_counter IN 1..10 LOOP
   DBMS_OUTPUT.PUT_LINE('Number: ' || TO_CHAR(loop_counter)
        || ' Square: ' || TO_CHAR(loop_counter**2));
 END
LOOP; END;
```

En el caso de especificar REVERSE el bucle se recorre en sentido inverso.

Excepciones.

Anteriormente dijimos que un bloque de código puede contener una sección denominada EXCEPTION. Esta sección es la encargada de recoger todas las anomalías que se puedan producir dentro del bloque de código.

Una excepción es una situación especial dentro de la ejecución de un programa, que puede ser capturada para asignar un nuevo comportamiento. Una excepción puede ser un error de ejecución (una división entre 0) o cualquier otro tipo de suceso.

Las excepciones deben ser declaradas dentro de la sección DECLARE, como si de una variable se tratase:

```
DECLARE
e_sin_alumnos EXCEPTION;
```

Una vez que la excepción está definida, ésta debe ser lanzada, ya sea automáticamente por Oracle (cuando se produce un error controlado por Oracle), o lanzada manualmente por el usuario a través de la instrucción **RAISE <excepción>.**

La sintaxis del manejador de excepciones es:

```
EXCEPTION
        WHEN nb_excepcion_1
     THEN instrucciones excep1;
        WHEN nb_excepcion_2
     THEN instrucciones excep2;
    ...
    WHEN nb_excepcion_n THEN
            instrucciones excepn;
    [WHEN OTHERS THEN
            instrucciones;]
```

Ejemplo:

```
DECLARE
   VALOR_NEGATIVO
   EXCEPTION; valor NUMBER;
BEGIN
   valor := -1;
   IF valor < 0 THEN
            RAISE VALOR_NEGATIVO;
   END IF;

EXCEPTION
WHEN VALOR_NEGATIVO THEN
            dbms_output.put_line('El valor no puede ser negativo');
END;
```

Cuando se produce un error, se ejecuta el bloque EXCEPTION. Si existe un bloque de excepción apropiado para el tipo de error producido se ejecuta dicho bloque. Si este último no existe, se ejecutará el bloque de excepción WHEN OTHERS THEN (en el caso de haberlo definido; este bloque debe ser el último manejador de excepciones). Una vez finalizada la ejecución del bloque de EXCEPTION no se continúa ejecutando el bloque anterior.

En ocasiones queremos enviar un mensaje de error personalizado al producirse una excepción PL/SQL. Para ello es necesario utilizar la instruccion **RAISE_APPLICATION_ERROR**;

```
RAISE_APPLICATION_ERROR(<error_num>,<mensaje>);
```

donde:

- **error_num** es un entero negativo comprendido entre -20001 y -20999
- **mensaje** la descripción del error

```
DECLARE
 v_div NUMBER;
BEGIN
   SELECT 1/0 INTO v_div FROM DUAL;
EXCEPTION
 WHEN OTHERS THEN
 RAISE_APPLICATION_ERROR(-20001,'No se puede dividir por cero');
END;
```

Dentro del bloque de excepciones conviene recordar la existencia de la excepción **OTHERS**, que simboliza cualquier condición de excepción que no ha sido declarada. Se utiliza comúnmente para controlar cualquier tipo de error que no ha sido previsto. En ese caso, es común observar la sentencia **ROLLBACK** en el grupo de sentencias de la excepción o alguna de las funciones **SQLCODE** – **SQLERRM**.

SQLCODE devuelve el número del error de Oracle y un 0 (cero) en caso de éxito al ejecutarse una sentencia SQL.

SQLERRM devuelve el la descripción del correspondiente mensaje de error. También es posible entregarle a la función **SQLERRM** un número negativo que represente un error de Oracle y ésta devolverá el mensaje asociado.

Estas funciones no pueden ser utilizadas directamente en una sentencia SQL, pero sí se puede asignar su valor a alguna variable de programa y luego usar esta última en alguna sentencia.

```
SET SERVEROUTPUT ON;
DECLARE
 err_num NUMBER;
 err_msg VARCHAR2(255);
 result NUMBER;
 msg VARCHAR2(255);
BEGIN
 msg := SQLERRM(-1403);
 DBMS_OUTPUT.put_line(MSG);
 SELECT 1/0 INTO result FROM DUAL;
EXCEPTION
WHEN OTHERS THEN
 err_num := SQLCODE;
 err_msg := SQLERRM;
 DBMS_OUTPUT.put_line('Error:'||TO_CHAR(err_num));
 DBMS_OUTPUT.put_line(err_msg);
END;
```

```
DECLARE
   e_sinreg EXCEPTION; a
   number(10) := 25; b
   number(10) := 0;
   c number(10);

BEGIN
  Select count(*) INTO a FROM
  Articulos; If a < 10 THEN
     RAISE e_sinreg;
  END IF;
  c := a / b;
  DBMS_OUTPUT.PUT_LINE(' Esto nunca llegará a mostrarse. ');

EXCEPTION
  WHEN ZERO_DIVIDE THEN DBMS_OUTPUT.PUT_LINE('No se puede dividir por 0');
  WHEN e_sinreg THEN DBMS_OUTPUT.PUT_LINE('Hay menos de 10 articulos.');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE('Se ha producido otra excepción.');
END;
```

Las líneas de código debajo del manejador específico se ejecutarán cuando esa excepción se produzca.

Algunas excepciones se lanzarán automáticamente cuando se produzcan ciertos tipos de errores en la ejecución del bloque de código. Cada excepción automática tiene asociado un código de error ORA-XXXX el cual si se produce, hará que se lance la excepción correspondiente.

A continuación se muestra una lista de las excepciones automáticas predefinidas por Oracle:

Excepción	Error Oracle
ACCESS_INTO_NULL	ORA-06530
COLLECTION_IS_NULL	ORA-06531
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01722
LOGIN_DENIED	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012
PROGRAM_ERROR	ORA-06501
ROWTYPE_MISMATCH	ORA-06504
STORAGE_ERROR	ORA-06500
SUBSCRIPT_BEYOND_COUNT	ORA-06533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532
TIMEOUT_ON_RESOURCE	ORA-00051
TOO_MANY_ROWS	ORA-01422
VALUE_ERROR	ORA-06502
ZERO_DIVIDE	ORA-01476

CURSORES.

Un cursor es el nombre para un área memoria privada que contiene información procedente de la ejecución de una sentencia SELECT. Cada cursor tiene unos atributos que nos devuelven información útil sobre el estado del cursor en la ejecución de la sentencia SQL. Cuando un cursor está abierto y los datos referenciados por la consulta SELECT cambian, estos cambios no son recogidos por el cursor.

PL/SQL crea implícitamente un cursor para todas las sentencias SQL de manipulación de datos sobre un conjunto de filas, incluyendo aquellas que solo devuelven una sola fila.

En PL/SQL no se pueden utilizar sentencias SELECT de sintaxis básica (SELECT <lista> FROM <tabla(s)>). PL/SQL utiliza cursores para gestionar las instrucciones **SELECT**. Un cursor es un conjunto de registros devuelto por una instrucción SQL.

Podemos distinguir dos tipos de cursores:

- **Cursores implícitos**. Este tipo de cursores se utiliza para operaciones **SELECT INTO**. Se usan cuando la consulta devuelve un único registro.
- **Cursores explícitos**. Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. Ocasionalmente también se utilizan en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Un cursor se define como cualquier otra variable de PL/SQL y debe nombrarse de acuerdo a los mismos convenios que cualquier otra variable.

Los cursores **implícitos** se utilizan para realizar consultas **SELECT** que devuelven un único registro. Deben tenerse en cuenta los siguientes puntos cuando se utilizan cursores implícitos:

- Los cursores implícitos no deben ser declarados
- Con cada cursor implícito debe existir la palabra clave **INTO**.
- Las variables que reciben los datos devueltos por el cursor tienen que contener el mismo tipo de dato que las columnas de la tabla.
- Los cursores implícitos solo pueden devolver una única fila. En caso de que se devuelva más de una fila (o ninguna fila) se producirá una excepción. Las más comunes son :

Excepción	
NO_DATA_FOUND	Se produce cuando una sentencia SELECT intenta recuperar datos pero ninguna fila satisface sus condiciones. Es decir, cuando *"no hay datos"*
TOO_MANY_ROWS	Dado que cada cursor implícito sólo es capaz de recuperar una fila , esta excepción detecta la existencia de más de una fila.

```
SET SERVEROUTPUT
ON; declare
      vdescripcion
VARCHAR2(50); begin
   SELECT DESCRIPCION INTO vdescripcion from PAISES WHERE CO_PAIS
   = 'ESP'; dbms_output.put_line('La lectura del cursor es: ' || vdescripcion);
end;
```

Para procesar instrucciones SELECT que devuelvan más de una fila, son necesarios cursores **explícitos** combinados con una estructura de bloque. A partir de ahora, cuando hagamos referencia a **cursores** nos referiremos a cursores explícitos.

Para trabajar con un cursor hay que realizar los siguientes pasos:

1.- Declarar el cursor
2.- Abrir el cursor en el servidor
3.- Recuperar cada una de sus filas (bucle)
4.- Cerrar el cursor

1. Declarar el cursor

Al igual que cualquier otra variable, el cursor se declara en la sección DECLARE. Se define el nombre que tendrá el cursor y qué consulta SELECT ejecutará. No es más que una declaración. La sintaxis básica es:

```
CURSOR nombre_cursor IS instrucción_SELECT
CURSOR nombre_cursor(param1 tipo1, ..., paramN tipoN) IS instrucción_SELECT
```

Una vez que el cursor está declarado ya podrá ser utilizado dentro del bloque de código.

Antes de utilizar un cursor se debe abrir. En ese momento se ejecuta la sentencia SELECT asociada y se almacena el resultado en el área de contexto (estructura interna de memoria que maneja el cursor). Un puntero señala a la primera fila

2. Abrir el cursor

Al abrir el cursor se ejecuta la sentencia SELECT asociada y cuyo resultado se guarda en el servidor en un área de memoria interna (tablas temporales) de las cuales se va retornando cada una de las filas según se va pidiendo desde el cliente. Al abrir un cursor, un puntero señalará al primer registro.

La sintaxis de apertura de un cursor es:

```
OPEN nombre_cursor;
OPEN nombre_cursor(valor1, valor2, ..., valorN);
```

Una vez que el cursor está abierto, se podrá empezar a pedir los resultados al servidor.

3. Recuperar cada una de sus filas.

Una vez que el cursor está abierto en el servidor se podrá hacer la petición de recuperación de fila. En cada recuperación solo se accederá a **una única** fila. La sintaxis de recuperación de fila de un cursor es:

```
FETCH nombre_cursor INTO variables;
```

Podremos recuperar filas mientras la consulta SELECT tenga filas pendientes de recuperar. Para saber cuándo no hay más filas podemos consultar los siguientes atributos de un cursor:

Nombre de atributo	Retorna	Descripción
Nombre_cursor%FOUND	BOOLEAN	Retorna si la última fila recuperada fue válida
Nombre_cursor%ISOPEN	BOOLEAN	Retorna si el cursor está abierto
Nombre_cursor%NOTFOUND	BOOLEAN	Retorna si la última fila fue inválida
Nombre_cursor%ROWCOUNT	NUMBER	Retorna el número de filas recuperadas

Al recuperar un registro, la información recuperada se guarda en una o varias variables. Si sólo se hace referencia a una variable, ésta se puede declarar con *%ROWTYPE*. Si se utiliza una lista de variables, cada variable debe coincidir en tipo y orden con cada una de las columnas de la sentencia SELECT.

Así lo acción más típica es recuperar filas mientras queden alguna por recuperar en el servidor. Esto lo podremos hacer a través de los siguientes bloques:

```
OPEN nombre_cursor;
LOOP
        FETCH nombre_cursor INTO variables;
        EXIT WHEN nombre_cursor%NOTFOUND;
        <procesar cada una de las filas>
END LOOP;
```

```
OPEN nombre_cursor;
FETCH nombre_cursor INTO lista_variables;
WHILE nombre_cursor%FOUND
LOOP
    /* Procesamiento de los registros recuperados
    */ FETCH nombre_cursor INTO lista_variables;
END LOOP;
CLOSE nombre_cursor;
```

```
FOR variable IN nombre_cursor LOOP
  /* Procesamiento de los registros recuperados */
END LOOP;
```

4. Cerrar el cursor

Una vez que se han recuperado todas las filas del cursor, hay que cerrarlo para que se liberen de la memoria del servidor los objetos temporales creados. Si no cerrásemos el cursor, la tabla temporal quedaría en el servidor almacenada con el nombre dado al cursor y la siguiente vez ejecutásemos ese bloque de código, nos daría la excepción CURSOR_ALREADY_OPEN (cursor ya abierto) cuando intentásemos abrir el cursor. Para cerrar el cursor se utiliza la siguiente sintaxis:

```
CLOSE nombre_cursor;
```

Cuando trabajamos con cursores debemos considerar:

- Cuando un cursor está cerrado, no se puede leer.
- Cuando leemos un cursor debemos comprobar el resultado de la lectura utilizando los atributos de los cursores.
- Cuando se cierra el cursor, es ilegal tratar de usarlo.
- El nombre del cursor es un identificador, no una variable. Se utiliza para identificar la consulta, por eso no se puede utilizar en expresiones.

Atributos en cursores implícitos

Los cursores implícitos no se pueden manipular por el usuario, pero Oracle sí permite el uso de sus atributos. Las sentencia a través de las que podemos obtener información de estos atributos son: SELECT...INTO, INSERT, UPDATE, DELETE.

En este caso, se debe anteponer al nombre del atributo el prefijo SQL, en lugar del nombre del cursor.

- **SQL%NOTFOUND** devuelve TRUE cuando la última sentencia SELECT no recuperó ninguna fila, o cuando INSERT, DELETE o UPDATE no afectan a ninguna fila
- **SQL%FOUND** devuelve TRUE cuando la última sentencia SELECT devuelve alguna fila, o cuando INSERT, DELETE o UPDATE afectan a alguna fila
- **SQL%ROWCOUNT** devuelve el número de filas afectadas por INSERT, DELETE o UPDATE o las filas devueltas por una sentencia SELECT
- **SQL%ISOPEN** siempre devuelve FALSE, porque Oracle cierra automáticamente el cursor implícito cuando termina la ejecución de la sentencia SELECT

Ejemplos:

```
DECLARE
 CURSOR cpaises IS
        SELECT CO_PAIS, DESCRIPCION, CONTINENTE
 FROM PAISES; co_pais VARCHAR2(3);
 descripcion VARCHAR2(50);
 continente VARCHAR2(25);
BEGIN
 OPEN cpaises;
 FETCH cpaises INTO co_pais,descripcion,continente;
 DBMS_OUTPUT.PUT_LINE(continente);
 CLOSE cpaises;
END;

DECLARE
 CURSOR cpaises IS
   SELECT CO_PAIS, DESCRIPCION, CONTINENTE
 FROM PAISES; registro cpaises%ROWTYPE;
BEGIN
 OPEN cpaises;
 FETCH cpaises INTO registro;
 DBMS_OUTPUT.PUT_LINE(continente);
 CLOSE cpaises;
END;

DECLARE
  r ARTICULOS%ROWTYPE;
BEGIN
 FOR r IN ( SELECT * FROM ARTICULOS ) LOOP
    DBMS_OUTPUT.PUT_LINE(r.cArtDsc);
 END LOOP;
END;

BEGIN
        UPDATE ARTICULOS
        SET cArtDsc = `Pantalla LCD'
        WHERE cCodArt = 'LCD';
        IF SQL%NOTFOUND THEN -- Otra opción : SQL%ROWCOUNT = 0
                INSERT INTO ARTICULOS (cCodArt,cDesArt)
                VALUES ('LCD','Pantalla LCD');
        END IF;
END;
```

Cursores Parametrizados

Los cursores son aquellos que permiten utilizar la orden OPEN para pasarle al cursor el valor de uno o varios de sus parámetros.

```
DECLARE
   CURSOR cArt (cFml Articulos.cArtFml%TYPE)
   IS SELECT cArtCdg,cArtDsc FROM Articulos WHERE cArtFml = cFml;
   xCod        Articulos.cArtCdg%TYPE;
      xDes     Articulos.cArtDsc%TYPE;
BEGIN
      OPEN cArt('F1');
      LOOP
          FETCH cArt INTO xCod,xDes;
        EXIT WHEN cArt%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (xDes);
      END LOOP;
      CLOSE cArt;
END;
```

Cursores de actualización

Los cursores de actualización se declaran igual que los cursores explícitos, añadiendo **FOR UPDATE** al final de la sentencia **SELECT**.

```
CURSOR nombre_cursor IS instrucción_SELECT FOR UPDATE
```

Para actualizar los datos del cursor hay que ejecutar una sentencia **UPDATE** especificando la cláusula **WHERE CURRENT OF** <*cursor_name*>.

```
UPDATE <nombre_tabla> SET <campo_1> = <valor_1>[,<campo_2> = <valor_2>]
WHERE CURRENT OF <cursor_name>
```

Cuando trabajamos con cursores de actualización debemos tener en cuenta que la sentencia UPDATE genera bloqueos en la base de datos (transacciones, disparadores,etc).

```
DECLARE
   CURSOR cpaises IS
   select CO_PAIS, DESCRIPCION, CONTINENTE from paises
   FOR UPDATE;
   co_pais VARCHAR2(3);
   descripcion VARCHAR2(50);
   continente VARCHAR2(25);
BEGIN
   OPEN cpaises;
   FETCH cpaises INTO co_pais,descripcion,continente;
   WHILE cpaises%found
   LOOP
     UPDATE PAISES SET CONTINENTE = CONTINENTE || '.'
      WHERE CURRENT OF cpaises;
      FETCH cpaises INTO co_pais,descripcion,continente;
   END LOOP;
   CLOSE cpaises;
   COMMIT;
END;
```

FUNCIONES, PROCEDIMIENTOS Y PAQUETES.

Una vez que tenemos escrito un bloque de código, podemos guardarlo en un fichero .SQL para su posterior uso, o bien guardarlo en base de datos para que pueda ser ejecutado por cualquier aplicación. El segundo caso se realiza mediante procedimientos almacenados (Stored Procedure).

A la hora de guardar un bloque de código hay que tener en cuenta ciertas normas:

- Palabra reservada DECLARE desaparece.
- Podremos crear procedimientos y funciones. Los procedimientos no podrán retornar ningún valor sobre su nombre, mientras que las funciones deben retornar un valor de un tipo de dato básico.

Un procedimiento [almacenado] es un subprograma que ejecuta una acción específica y que no devuelve ningún valor por si mismo, como sucede con las funciones. Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código. Para crear un procedimiento (stored procedure: procedimiento almacenado) usaremos la siguiente sintaxis:

```
CREATE {OR REPLACE} PROCEDURE nombre_proc( param1 [IN | OUT | IN OUT] tipo,... )
IS
        -- Declaración de variables locales
        BEGIN
                -- Instrucciones de ejecución
        [EXCEPTION]
                -- Instrucciones de excepción
        END;
```

Tras crear el procedimiento, éste se compila y luego se almacena en la BD de forma compilada. Este procedimiento luego puede ser invocado desde cualquier bloque PL/SQL.

El uso de OR REPLACE permite sobrescribir un procedimiento existente. Si se omite, y el procedimiento existe, se producirá, un error. Debemos especificar el tipo de datos de cada parámetro.

Al especificar el tipo de dato del parámetro no debemos especificar la longitud del tipo, aunque si puede ser utilizando el operador %TYPE.

Los parámetros pueden ser de entrada (**IN**), de salida (**OUT**) o de entrada salida (**IN OUT**). El valor por defecto es **IN**, y se toma ese valor en caso de que no especifiquemos nada.

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta NUMBER, new_saldo NUMBER)
IS
 -- Declaracion de variables locales
BEGIN
 UPDATE SALDOS_CUENTAS
        SET SALDO = new_saldo,
      FX_ACTUALIZACION = SYSDATE
 WHERE CO_CUENTA = cuenta;
END Actualiza_Saldo;
```

También podemos asignar un valor por defecto a los parámetros, utilizando la cláusula **DEFAULT** o el operador de asignación (:=) .

```
CREATE OR REPLACE
        PROCEDURE Actualiza_Saldo(cuenta NUMBER, new_saldo NUMBER DEFAULT 10)
```

Una vez creado y compilado el procedimiento almacenado podemos ejecutarlo. Existen dos formas de pasar argumentos a un procedimiento almacenado a la hora de ejecutarlo. Estas son:

- **Notación posicional**: Se pasan los valores de los parámetros en el mismo orden en que el procedure los define.

```
BEGIN
        Actualiza_Saldo(200501,2500);
    COMMIT;
END;
```

- **Notación nominal**: Se pasan los valores en cualquier orden nombrando explícitamente el parámetro y su valor separados por el símbolo => .

```
        BEGIN
                Actualiza_Saldo(cuenta => 200501,new_saldo => 2500);
                COMMIT;
        END;
```

```
CREATE OR REPLACE PROCEDURE today_is AS
BEGIN
 DBMS_OUTPUT.PUT_LINE( 'Hoy es ' || TO_CHAR(SYSDATE, '
DD/MM/YYYY') ); END today_is;
/
-- para ejecutarlo
SET SERVEROUTPUT ON;
BEGIN
 today_is(); -- the parentheses are optional
here END;

CREATE OR REPLACE PROCEDURE today2_is ( fecha DATE ) AS
BEGIN
 DBMS_OUTPUT.PUT_LINE( 'Hoy es ' || TO_CHAR(fecha, '
DD/MM/YYYY') ); END;
-- para ejecutarlo
SET SERVEROUTPUT ON;
BEGIN
 today2_is(to_date('01/02/2008')); -- the parentheses are optional
here END;

-- para ejecutarlo
SET SERVEROUTPUT ON;
BEGIN
 today2_is(fecha => to_date('01/02/2008')); -- the parentheses are
optional here END;
```

Para crear una función usaremos la siguiente sintaxis:

```
CREATE {OR REPLACE} FUNCTION nombre_func(param1
tipo,param2 tipo,... ) RETURN tipo_dato IS
        -- Declaración de variables locales
        BEGIN
                -- Instrucciones de ejecución
        [EXCEPTION]
                -- Instrucciones de excepción
END;
```

Packages.

Además de brindarnos múltiples elementos que nos permiten desarrollar una aplicación robusta, Oracle nos ofrece la posibilidad de programar en forma modular, clara y eficiente. En este apartado veremos cómo embeber procedimientos, funciones, definiciones de tipos de datos y declaraciones de variables en una misma estructura que los agrupe y relacione lógicamente. Esta estructura se denomina *Package* (Paquete) y su uso nos permite no sólo mejorar la calidad de diseño de nuestras aplicaciones sino también optimizar el desempeño de las mismas.

Un Paquete es un objeto PL/Sql que agrupa lógicamente otros objetos PL/Sql relacionados entre sí, encapsulándolos y convirtiéndolos en una *unidad* dentro de la base de datos.

Los Paquetes están divididos en 2 partes: especificación (obligatoria) y cuerpo (no obligatoria). La especificación o encabezado es la interfaz entre el Paquete y las aplicaciones que lo utilizan y es allí donde se *declaran* los tipos, variables, constantes, excepciones, cursores, procedimientos y funciones que podrán ser invocados desde fuera del paquete.

En el cuerpo del paquete se implementa la especificación del mismo. El cuerpo contiene los detalles de implementación y declaraciones privadas, manteniéndose todo esto oculto a las aplicaciones externas, siguiendo el conocido concepto de *"caja negra"*. Sólo las declaraciones hechas en la especificación del paquete son visibles y accesibles desde fuera del paquete (por otras aplicaciones o procedimientos almacenados) quedando los detalles de implementación del cuerpo del paquete totalmente ocultos e inaccesibles para el exterior.

Para acceder a los elementos declarados en un paquete basta con anteceder el nombre del objeto referenciado con el nombre del paquete donde está declarado y un punto, de esta manera: Paquete.Objeto donde *Objeto* puede ser un tipo, una variable, un cursor, un procedimiento o una función declarados dentro del paquete.

Para referenciar objetos desde adentro del mismo paquete donde han sido declarados no es necesario especificar el nombre del paquete y pueden (deberían) ser referenciados directamente por su nombre. Finalmente y siguiendo a la parte declarativa del cuerpo de un paquete puede, opcionalmente, incluirse la sección de inicialización del paquete. En esta sección pueden, por ejemplo, inicializarse variables que utiliza el paquete. La sección de inicialización se ejecuta sólo la primera vez que una aplicación referencia a un paquete, esto es, se ejecuta sólo una vez por sesión.

Como hemos dicho anteriormente, la creación de un paquete pasa por dos fases:

- Crear la cabecera del paquete donde se definen que procedimientos, funciones, variables, cursores, etc. Disponibles para su uso posterior fuera del paquete. En esta parte solo se declaran los objetos, no se implementa el código.
- Crear el cuerpo del paquete, donde se definen los bloques de código de las funciones y procedimientos definidos en la cabecera del paquete.

Para crear la cabecera del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE
nombre_de_paquete IS < Declaraciones >
END;
```

Para crear el cuerpo del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE BODY
nombre_paquete IS < Bloques de código>
END;
```

Hay que tener en cuenta que toda declaración de función o procedimiento debe estar dentro del cuerpo del paquete, y que todo bloque de código contenido dentro del cuerpo debe estar declarado dentro de la cabecera de paquete.

Cuando se quiera acceder a las funciones, procedimientos y variables de un paquete se debe anteponer el nombre de este:

Nombre_paquete.función(x)

Nombre_paquete.procedimiento(x)

Nombre_paquete.variable

```
CREATE OR REPLACE PACKAGE PK1 IS
  PROCEDURE xLis (xfamilia IN Articulos.cArtFml%TYPE);
END;

CREATE OR REPLACE PACKAGE BODY PK1 IS
        procedure xLis (xfamilia Articulos.cArtCdg%TYPE)
        IS
                xfam Articulos.cArtFml%type;
                xCod Articulos.cArtCdg%TYPE;
                xDes Articulos.cArtDsc%TYPE;
                CURSOR cArticulos IS SELECT cArtCdg,cArtDsc
                        FROM Articulos WHERE cArtFml = xfam;
        BEGIN
                xfam := xfamilia;
                OPEN cArticulos;
                LOOP
                        FETCH cArticulos INTO xCod,xDes;
                        EXIT WHEN cArticulos%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE (xDes);
                END LOOP;
                CLOSE cArticulos;
        END;
END;
```

Ventajas del uso de Paquetes.

Dentro de las ventajas que ofrece el uso de paquetes podemos citar que:

- **Permite modularizar el diseño de nuestra aplicación**

El uso de Paquetes permite encapsular elementos relacionados entre sí (tipos, variables, procedimientos, funciones) en un único módulo PL/Sql que llevará un nombre que identifique la funcionalidad del conjunto.

- **Otorga flexibilidad al momento de diseñar la aplicación**

En el momento de diseñar una aplicación todo lo que necesitaremos inicialmente es la información de interfaz en la especificación del paquete. Puede codificarse y compilarse la especificación sin su cuerpo para posibilitar que otros sub-programas que referencian a estos elementos declarados puedan compilarse sin errores. De esta manera podremos armar un "prototipo" de nuestro sistema antes de codificar el detalle del mismo.

- **Permite ocultar los detalles de implementación**

Pueden especificarse cuáles tipos, variables y sub-programas dentro del paquete son públicos (visibles y accesibles por otras aplicaciones y sub-programas fuera del paquete) o privados (ocultos e inaccesibles fuera del paquete). Por ejemplo, dentro del paquete pueden existir procedimientos y funciones que serán invocados por otros programas, así como también otras rutinas de uso interno del paquete que no tendrán posibilidad de ser accedidas fuera del mismo. Esto asegura que cualquier cambio en la definición de estas rutinas internas afectará sólo al paquete donde se encuentran, simplificando el mantenimiento y protegiendo la integridad del conjunto.

- **Agrega mayor funcionalidad a nuestro desarrollo**

Las definiciones públicas de tipos, variables y cursores hechas en la especificación de un paquete persisten a lo largo de una sesión. Por lo tanto pueden ser compartidas por todos los sub-programas y/o paquetes que se ejecutan en ese entorno durante esa sesión. Por ejemplo, puede utilizarse esta técnica (en dónde sólo se define una especificación de paquete y no un cuerpo) para mantener tipos y variables globales a todo el sistema.

- **Introduce mejoras al rendimiento**

En relación a su ejecución, cuando un procedimiento o función que está definido dentro de un paquete es llamado por primera vez, todo el paquete es ingresado a memoria. Por lo tanto, posteriores llamadas al mismo u otros sub-programas dentro de ese paquete realizarán un acceso a memoria en lugar de a disco. Esto no sucede con procedimientos y funciones estándares.

- **Permite la "Sobrecarga de funciones" (*Overloading)*.**

PL/SQL nos permite que varios procedimientos o funciones almacenadas, declaradas dentro de un mismo paquete, tengan el mismo nombre. Esto nos es muy útil cuando necesitamos que los sub-programas puedan aceptar parámetros que contengan diferentes tipos de datos en diferentes instancias. En este caso Oracle ejecutará la "versión" de la función o procedimiento cuyo encabezado se corresponda con la lista de parámetros recibidos.

Disparadores o TRIGGERS.

Los disparadores (o triggers) son bloques de código PL/SQL asociados a una tabla y que se ejecutan **automáticamente** como reacción a una operación DML específica (INSERT, UPDATE o DELETE) sobre dicha tabla.

En definitiva, los disparadores son eventos a nivel de tabla que se ejecutan automáticamente cuando se realizan ciertas operaciones sobre la tabla.

Para crear un disparador utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} TRIGGER
        nombre_disp [BEFORE|AFTER]
        [DELETE|INSERT|UPDATE {OF columnas}] [ OR [DELETE|INSERT|UPDATE {OF
         columnas}]...] ON tabla
        [FOR EACH ROW [WHEN condicion
disparo]] [DECLARE]
        -- Declaración de variables locales
BEGIN
        -- Instrucciones de ejecución
[EXCEPTION]
        -- Instrucciones de excepción
END;
```

El uso de OR REPLACE permite sobrescribir un trigger existente. Si se omite, y el trigger existe, se producirá, un error.

En principio, dentro del cuerpo de programa del trigger podemos hacer uso de cualquier orden de consulta o manipulación de la BD, y llamadas a funciones o procedimientos siempre que:

- No se utilicen comandos DDL
- No se acceda a las tablas que están siendo modificadas con DELETE, INSERT o UPDATE en la misma sesión
- No se violen las reglas de integridad, es decir no se pueden modificar llaves primarias, ni actualizar llaves externas
- No se utilicen sentencias de control de transacciones (Commit, Rollback o Savepoint)
- No se llamen a procedimientos que trasgredan la restricción anterior
- No se llame a procedimientos que utilicen sentencias de control de transacciones

Predicados condicionales.

Cuando se crea un trigger para más de una operación DML, se puede utilizar un predicado condicional en las sentencias que componen el trigger que indique que tipo de operación o sentencia ha disparado el trigger. Estos predicados condicionales son los siguientes:

Inserting	Retorna **true** cuando el trigger ha sido disparado por un **INSERT**
Deleting	Retorna **true** cuando el trigger ha sido disparado por un **DELETE**
Updating	Retorna **true** cuando el trigger ha sido disparado por un **UPDATE**
Updating (columna)	Retorna **true** cuando el trigger ha sido disparado por un **UPDATE** y la **columna** ha sido modificada.

```
CREATE TRIGGER audit_trigger BEFORE INSERT OR DELETE OR UPDATE
 ON classified_table FOR EACH ROW
 BEGIN
  IF INSERTING THEN
    INSERT INTO audit_table
      VALUES (USER || ' is inserting' ||
            ' new key: ' || :new.key);
  ELSIF DELETING THEN
    INSERT INTO audit_table
      VALUES (USER || ' is deleting' ||
            ' old key: ' || :old.key);
  ELSIF UPDATING('FORMULA') THEN
    INSERT INTO audit_table
      VALUES (USER || ' is updating' ||
            ' old formula: ' || :old.formula ||
            ' new formula: ' || :new.formula);
  ELSIF UPDATING THEN
    INSERT INTO audit_table
      VALUES (USER || ' is updating' ||
            ' old key: ' || :old.key ||
            ' new key: ' || :new.key);
  END IF;
 END;
```

Tipos de triggers.

Los triggers pueden definirse para las operaciones INSERT, DELETE o Update, y pueden dispararse antes o después de la operación. Finalmente, el nivel de los disparadores puede ser la fila o registro o la orden.

El modificador BEFORE o AFTER indica que el trigger se ejecutará antes o después de ejecutarse la sentencia SQL definida por DELETE, INSERT o UPDATE. Si incluimos el modificador OF el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

El alcance de los disparadores puede ser la fila o de orden. El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre cada fila de la tabla. Si se acompaña del modificador WHEN, se establece una restricción; el trigger solo actuará, sobre las filas que satisfagan la restricción.

Tipos de disparadores.

Categoria	**Valores**	**Comentarios**
Orden	INSERT, DELETE, UPDATE	Define que tipo de operación DML provoca la activación del trigger
Temporalización	BEFORE o AFTER	Define si el disparador se activa antes o después de que se ejecute la operación DML
Nivel	Fila u Orden	Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo. Los Triggers con nivel de orden se activan sólo una vez, antes o después de la orden. Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador.

Orden de ejecución de los triggers

Una misma tabla puede tener varios triggers asociados. En tal caso es necesario conocer el orden en el que se van a ejecutar.

Los disparadores se activan al ejecutarse la sentencia SQL.

1. Si existe, se ejecuta el disparador de tipo BEFORE (disparador previo) con nivel de orden.
2. Para cada fila a la que afecte la orden:
   2.1. Se ejecuta si existe, el disparador de tipo BEFORE con nivel de fila.
   2.2. Se ejecuta la propia orden.
   2.3. Se ejecuta si existe, el disparador de tipo AFTER (disparador posterior) con nivel de fila.
3. Se ejecuta, si existe, el disparador de tipo AFTER con nivel de orden.

Restricciones de los Triggers.

El cuerpo de un disparador es un bloque PL/SQL. Cualquier orden que sea legal en un bloque PL/SQL , es legal en el cuerpo de un disparador, con las siguientes restricciones:

a. Un disparador no puede emitir ninguna orden de control de transacciones (COMMIT, ROLLBACK o SAVEPOINT). El disparador se activa como parte de la ejecución de la orden que provocó el disparo, y forma parte de la misma transacción qu dicha orden. Cuando la orden que provoca la orden es confirmada o cancelada, se confirma o se cancela también el trabajo realizado por el disparador.

b. Por las mismas razones, ningún procedure o función llamado por el disparador puede emitir órdenes de control de transacciones.

c. El disparador no puede declarar variables de tipo LONG.

Utilización de :old y :new en los disparadores con nivel de fila.

Un disparador con nivel de fila se ejecuta una vez por cada fila procesada por la orden que provoca el disparo. Dentro del disparador, puede accederse a la fila que está siendo actualmente procesada utilizando, para ello, dos pseudo-registros, **:old** y **:new.**

En principio tanto :old como :new son del tipo **tabla_disparo%ROWTYPE;**

Orden de disparo	:old	:new
INSERT	No definido; todos los campos toman el valor NULL.	Valores que serán insertados cuando se complete la orden.
UPDATE	Valores originales de la fila, antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE	Valores originales, antes del borrado de la fila.	No definido; todos los campos toman el valor NULL.

Ejemplo :

```
CREATE TRIGGER scott.emp_permit_changes
  BEFORE DELETE OR INSERT OR UPDATE ON scott.emp
  DECLARE
    dummy INTEGER;
  BEGIN
    /* If today is a Saturday or Sunday, then return an
    error.*/ IF (TO_CHAR(SYSDATE, 'DY') = 'SAT' OR
    TO_CHAR(SYSDATE, 'DY') = 'SUN')
     THEN raise_application_error( -20501,
   'May not change employee table during the weekend');
    END IF;
   /* Compare today's date with the dates of all
     company holidays. If today is a company
     holiday, then return an error.*/
   SELECT COUNT(*) INTO dummy FROM
     company_holidays WHERE day = TRUNC(SYSDATE);
   IF dummy > 0
    THEN raise_application_error( -20501,
     'May not change employee table during a holiday');
   END IF;
   /*If the current time is before 8:00AM or
     after 6:00PM, then return an error. */
   IF (TO_CHAR(SYSDATE, 'HH24') < 8 OR
     TO_CHAR(SYSDATE, 'HH24') >= 18)
     THEN raise_application_error( -20502,
   'May only change employee table during working hours');
    END
  IF; END;
```

La cláusula WHEN.

La cláusula WHEN sólo es válida para disparadores con nivel de fila. Si está presente, el cuerpo del disparador sólo se ejecutará para las filas que cumplan la condición especificada en la cláusula.

```
CREATE TRIGGER tr1
  BEFORE INSERT OR UPDATE OF salario ON scott.emp FOR
  EACH ROW WHEN (new.job <> 'PRESIDENT')
  BEGIN
    /* Cuerpo del disparador
  */ END;
```

Esto último es equivalente a :

```
CREATE TRIGGER tr1
  BEFORE INSERT OR UPDATE OF salario ON scott.emp
  FOR EACH ROW
  BEGIN
          IF :new.job <> 'PRESIDENT' THEN
    /* Cuerpo del disparador */
          END IF;
  END;
```

Para hacer que un trigger ejecute un ROLLBACK de la transacción que tiene activa y teniendo en cuenta que en las sentencias que componen el cuerpo de un trigger no puede haber este

tipo de sentencias (rollback, commit,…) hay que ejecutar "***error / excepcion***" mediante la sentencia raise_application_error cuya sintaxis es

```
RAISE_APPLICATION_ERROR(num_error,'mensaje');
```

El num_error es un número entero cualquiera, aunque se aconseja que tenga 5 dígitos.

Por ejemplo

```
raise_application_error( 20000,' No se puede modificar el cliente.');
```

Tabla mutando

Cuando se realiza un trigger sobre una tabla, dicha tabla se dice que está en "**proceso de mutación**", es decir, que se están realizando cambios sobre ella y que por tanto **dentro del trigger no se debe hacer ningún tipo de acceso a dicha tabla con operaciones DML (SELECT, INSERT, DELETE o UPDATE).**

Si queremos conocer los valores del registro de la tabla sobre la que se ha disparado el trigger, este último debe estar declarado de forma **FOR EACH ROW** y acceder a sus valores mediante las pseudocolumnas **:NEW** y **:OLD**.

Campos Calculados

En algunos SGBDR se permite el uso de campos calculados. Son campos cuyo valor se calcula de forma automática cuando cambia el valor de alguno de los campos o valores de los que depende.

Por ejemplo, en SQL SERVER se haría de la siguiente forma:

```
CREATE TABLE FacturasL(
        nNumFac         numeric(10) NOT NULL,
        nNumLin         numeric(10) NOT NULL,
        cCodArt         char(13) NOT NULL,
        nPvpArt         numeric(14,4) DEFAULT 0,
        nUniArt         numeric(14,4) DEFAULT 0,
        nDtoLin         numeric(14,4) DEFAULT 0,
        nBaseImp        AS nPvpArt * nUniArt,
        nAnnio          AS YEAR(GetDate()),
        CONSTRAINT PK_FacturasL PRIMARY KEY (nNumFac, nNumLin)
);
```

En el ejemplo anterior, los campos nBaseImp y nAnnio son calculados y su valor se calcula de forma automática; así que no pueden ser ni insertados ni modificados.

En Oracle no existen los campos calculados, así que, hemos de hacer uso de los triggers para simularlos. La forma de hacerlo es:

1. Creando una segunda tabla, con la misma PRIMARY KEY que la tabla sobre la cual se van a realizar los cálculos, más los campos "calculados"
2. Crear los triggers necesarios para actualizar los campos "calculados".

```
--EJEMPLO DE CAMPOS CALCULADOS EN UNA TABLA ORACLE

CREATE TABLE FacturasL(
        nNumFac         NUMBER(10) NOT NULL,
        nNumLin         NUMBER(10) NOT NULL,
```

```
        cCodArt         VARCHAR(13) NOT NULL,
        nPvpArt         NUMBER(14,4) DEFAULT 0,
        nUniArt         NUMBER(14,4) DEFAULT 0,
        nDtoLin         NUMBER(14,4) DEFAULT 0,
        CONSTRAINT PK_FacturasL PRIMARY KEY (nNumFac, nNumLin)
);

CREATE TABLE FacturasL_Calc(
        nNumFac         NUMBER(10) NOT NULL,
        nNumLin         NUMBER(10) NOT NULL,
        nBaseImp        NUMBER(14,4) DEFAULT 0,
        nSubTotal NUMBER(14,4) DEFAULT 0,
        CONSTRAINT PK_FacturasL_Calc PRIMARY KEY (nNumFac, nNumLin)
);

CREATE OR REPLACE TRIGGER tr_FacturasL_Ins_Upd
AFTER INSERT OR DELETE OR UPDATE
ON FacturasL
FOR EACH ROW
DECLARE
  PKE Exception;
BEGIN

-- No se puede cambiar la PK
        IF Updating('nNumFac') OR Updating('nNumLin')
                THEN RAISE PKE; -- Lanzar Excepcion
        END IF;
        IF Inserting THEN
                INSERT INTO FacturasL_Calc VALUES (:new.nNumFac,:new.nNumLin,
                        :new.nPvpArt * :new.nUniArt,
  :new.nPvpArt                                          * :new.nUniArt    -         (:new.nPvpArt  *
                                                                                    :new.nUniArt       *
:new.nDtoLin / 100)
                        );
        END IF;

        IF Deleting THEN
                DELETE FacturasL_Calc
                        WHERE nNumFac = :old.nNumFac AND nNumLin = :old.nNumLin;
        END IF;

        IF Updating('nPvpArt') OR Updating('nUniArt') OR
                Updating('nDtoLin') THEN UPDATE FacturasL_Calc
                        SET nBaseImp = :new.nPvpArt * :new.nUniArt,
                                nSubTotal = :new.nPvpArt * :new.nUniArt - (:new.nPvpArt * :new.nUniArt
* :new.nDtoLin / 100)
                        WHERE nNumFac = :new.nNumFac AND nNumLin =
                        :new.nNumLin; DBMS_OUTPUT.PUT_LINE('Cambio Realizado.');
        END IF;

  EXCEPTION
          WHEN PKE THEN
    RAISE_APPLICATION_ERROR (-20001,'Error: No se puede modificar ni Número de Factura ni
Línea de Factura.');
   WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR (-20001,'Error: Actualizando campos calculados
FacturasL_Calc.'); END ;

----------------------------------------
```

```
SET SERVEROUTPUT ON;
INSERT INTO FacturasL VALUES (1000,1,'HD-SEA01-1TB',60.5,3,10);
INSERT INTO FacturasL VALUES (1000,2,'TFT LG 19',195,1,0);
COMMIT;
SELECT * FROM FacturasL;
SELECT * FROM FacturasL_Calc;
-- Hasta aqui para probar el INSERT
SET SERVEROUTPUT ON;
UPDATE FacturasL SET cCodArt = 'HD-IBM01-3TB' WHERE nNumLin = 1;
COMMIT;
SELECT * FROM FacturasL;
SELECT * FROM FacturasL_Calc;
-- Hasta aqui para probar que el cambio de cCodArt no hace nada
-- ni siquiera lanza la salida OUTPUT : Cambio Realizado

SET SERVEROUTPUT ON;
UPDATE FacturasL SET nPvpArt = 99 WHERE nNumLin = 1;
COMMIT;
SELECT * FROM FacturasL;
SELECT * FROM FacturasL_Calc;
-- Hasta aqui para probar que el cambio de nPvpArt lanza el Trigger
-- Ademas lanza la salida OUTPUT : Cambio Realizado

DELETE FacturasL WHERE nNumLin = 1;
COMMIT;
SELECT * FROM FacturasL; SELECT
* FROM FacturasL_Calc;
-- Hasta aqui para probar que el borrado de una linea lanza el Trigger
```

Para más información sobre los disparadores consultar el ***Oracle SQL Reference***

Registros y tablas

Existen dos tipos de datos que no hemos mencionado anteriormente: los registros (o estructuras) y las tablas (o arrays o vectores).

Los dos tipos deben ser definidos como un nuevo tipo antes de declarar variables de ese nuevo tipo.

El modo de definir nuevos tipos de variables en PL/SQL es a través de la palabra reservada TYPE:

```
TYPE nuevo_tipo IS tipo_original.
```

Una vez definido en nuevo tipo, ya se pueden definir variables de ese nuevo tipo.

Registros:

Los registros no son más que agrupaciones de tipos de variables que se acceden con el mismo nombre.

La sintaxis de definición de registros es:

```
TYPE nombre_registro IS RECORD(
        Campo1 tipo,
        Campo2 tipo,
        Campo3 tipo );
```

Por ejemplo:

```
TYPE alumno IS RECORD( n_alumno
        VARCHAR2(5), nombre
        VARCHAR2(25), tlf
        VARCHAR2(15) );
```

Tablas.

Una tabla no es más que una colección de elementos identificados cada uno de ellos por un índice. En muchos lenguajes se les denomina **arrays o matrices**.

La sintaxis de definición de tablas es:

```
TYPE nombre_tabla IS TABLE OF tipo_de_elementos;
```

El tamaño de la tabla se define durante la declaración de la variable.

```
Nombre_variable nombre_tabla := nombre_variable( lista de elementos );
```

Por ejemplo:

```
DECLARE
TYPE array_enteros IS TABLE OF INTEGER;
Un_array array_enteros := array_enteros( 0, 0, 0, 0 );
```

Tunning básico de SQL

Una de las tareas más importantes de las propias de un desarrollador de bases de datos es la de puesta a punto o tuning. Hay que tener en cuenta que las sentencias SQL pueden llegar a ser muy complejas y conforme el esquema de base de datos va creciendo las sentencias son más complejas y confusas. Por es difícil escribir la sentencia correcta a la primera.

Por todo ello después de tener cada uno de los procesos escrito, hay que pasar por una etapa de tuning en la que se revisan todas las sentencias SQL para poder optimizarlas conforme a la experiencia adquirida.

Tanto por cantidad como por complejidad, la mayoría de las optimizaciones deben hacerse sobre sentencias SELECT, ya que son (por regla general) las responsables de la mayor pérdida de tiempos.

A continuación se dan unas normas básicas para escribir sentencias SELECT optimizadas.

- Las condiciones (tanto de filtro como de join) deben ir siempre en el orden en que esté definido el índice. Si no hubiese índice por las columnas utilizadas, se puede estudiar la posibilidad de añadirlo, ya que tener índices de más sólo penaliza los tiempos de inserción, actualización y borrado, pero no de consulta.
- Evitar la condiciones IN (SELECT…) sustituyéndolas por joins.
- Colocar la tabla que devuelve menor número de registros en el último lugar del FROM
- Una consulta cualificada con la cláusula DISTINTC debe ser ordenada por el servidor aunque no se incluya la cláusula ORDER BY.