

Programación de bases de datos

Contenidos

- ☒ Introducción a los lenguajes de programación de bases de datos
- ☒ Tipos de datos, identificadores y variables
- ☒ Operadores y expresiones
- ☒ Estructuras de control
- ☒ Gestión de errores
- ☒ Transacciones en scripts

Objetivos

- ☒ Introducir el concepto de programación de bases de datos
- ☒ Aprender los mecanismos básicos de la programación de bases de datos
- ☒ Diseñar y codificar scripts para tareas complejas
- ☒ Incorporar el uso de las transacciones en tareas complejas
- ☒ Manejar cursos

En este tema el alumno aprenderá a programar una base de datos, desarrollando programas y guiones utilizando las técnicas apropiadas.

6.1. Introducción a la programación de bases de datos

Programar una base de datos es la forma en la que el desarrollador o administrador de la base de datos interactúa con ella.

Hay muchas maneras de programar una base de datos, la elección del modo más adecuado depende de con qué tecnología de base de datos se está trabajando (por ejemplo, Oracle), de qué compiladores se tengan instalados en la máquina objeto de ejecución (por ejemplo, C++), de qué ventajas e inconvenientes se persigan, o de la naturaleza de la aplicación de base de datos que se va a desarrollar. La aplicación de base de datos puede ser de tres tipos distintos:

- **Codificación en el lado del servidor:** La lógica de la aplicación reside enteramente en la base de datos. La aplicación está basada en la implementación de disparadores que se ejecutan automáticamente cuando ocurre algún cambio en los datos almacenados, y en el almacenamiento de procedimientos y funciones que son llamados explícitamente. Una ventaja importante es que, de esta forma, se puede reutilizar el mismo código para muchos clientes.
- **Modelo cliente/servidor (también llamado modelo en dos niveles):** El código de la aplicación corre en otra máquina distinta del servidor donde se encuentra la base de datos. Las llamadas a la base de datos son transmitidas desde la máquina cliente al servidor. Los datos son transmitidos desde el cliente al servidor para las operaciones de inserción y actualización, y desde el servidor al cliente durante las consultas. Los datos son procesados en la máquina cliente. Normalmente, estas aplicaciones se escriben usando precompiladores con las sentencias SQL embebidas en el código.
- **Modelo en tres niveles:** Al modelo en dos niveles se le añade un servidor de aplicación separado que procesa las peticiones. Este podría ser, por ejemplo, un servidor Web básico, o un servidor que realizará funciones de cacheo de datos y balanceo de carga. Aumentar la potencia de procesamiento de esta capa intermedia permite disminuir los recursos necesarios para los clientes, pudiendo estos llegar a ser simples navegadores Web.

Para el desarrollo de aplicaciones, además de necesitar un interfaz de programación donde hacer el desarrollo, se precisa de un software de cliente que enlace el código desarrollado con la base de datos. Los interfaces más comunes son:

- **SQL incorporado:** Las aplicaciones de bases de datos de SQL incorporado se conectan a las bases de datos y ejecutan directamente sentencias de SQL

incorporado que se encuentran insertadas dentro de una aplicación escrita en un lenguaje principal (típicamente C, C++ o COBOL). Las sentencias SQL se pueden ejecutar estáticamente o dinámicamente.

- **ODBC:** Microsoft desarrolló una interfaz SQL denominada Open Database Connectivity (ODBC) para los sistemas operativos de Microsoft. Los controladores ODBC específicos de la BBDD se cargan dinámicamente en el tiempo de ejecución, mediante un gestor de controladores basado en la fuente de datos (nombre de la base de datos) proporcionada en la petición de conexión. La aplicación se enlaza directamente con una única biblioteca del gestor de controladores, en lugar de con la biblioteca de cada motor de base de datos. El gestor de controladores media entre las llamadas de función de la aplicación en el momento de la ejecución y asegura que estas se dirijan hasta el controlador ODBC adecuado que sea específico para el motor. Dado que el gestor del controlador ODBC solo conoce las funciones específicas de ODBC, no se podrá acceder a las funciones específicas del motor de base de datos. ODBC no está limitado a los sistemas operativos de Microsoft; otras implementaciones están disponibles en varias plataformas. Para el desarrollo de aplicaciones ODBC, debe instalarse un ODBC Software Development Kit¹.
- **CLI:** Call Level Interface (interfaz a nivel de llamada). Es una interfaz de programación de aplicaciones C y C++ para el acceso a bases de datos relacionales, que utiliza llamadas de función para pasar sentencias de SQL dinámico como argumentos de función. Es una alternativa al SQL dinámico incorporado, pero a diferencia de SQL incorporado, CLI de DB2 no necesita variables del sistema principal o precompilador. Se basa en la especificación ODBC 3,51 y en la Norma Internacional para SQL/CLI. Estas especificaciones se escogieron como la base para proporcionar una curva de aprendizaje más corta para aquellos programadores de aplicaciones, ya familiarizados con cualquiera de estas interfaces de bases de datos.
- **JDBC:** Java Database Connectivity es un API (Applications Programming Interface, interfaz de programación de aplicaciones) que permite enviar sentencias SQL a una base de datos relacional.
- **SQLJ:** Es un estándar ANSI SQL-1999 para incorporar sentencias SQL en código fuente Java. Proporciona una alternativa al JDBC para el acceso a datos desde Java tanto en el lado cliente como en el lado servidor. Un código fuente SQLJ es más breve que su equivalente en código fuente JDBC.

¹Para la plataforma Windows, el SDK de ODBC es parte del SDK de Microsoft Data Access Components (MDAC), disponible para descarga en <http://www.microsoft.com/data>.

- **OCI y OCCI:** Oracle Call Interface (OCI) y Oracle C++ Call Interface (OCCI) son APIs que permiten crear aplicaciones que invocan procedimientos y funciones nativas para acceder y controlar todas las fases de ejecución de sentencias SQL en bases de datos Oracle. Permite desarrollar aplicaciones combinando la potencia de acceso a los datos del SQL, con la capacidad de hacer bucles, crear estructuras de control...
- **Interfaz de base de datos Perl:** Combina la potencia del lenguaje interpretado Perl y el SQL dinámico. Estas propiedades convierten a Perl en un lenguaje perfecto para crear y revisar rápidamente aplicaciones de bases de datos (DB2). El Módulo DBI de Perl utiliza una interfaz que es muy parecida a las interfaces CLI y JDBC, lo cual facilita la traducción de las aplicaciones Perl a aplicaciones CLI y JDBC, y viceversa.
- **Hypertext Preprocessor (PHP):** Es un lenguaje de programación interpretado. La primera versión de PHP fue creada por Rasmus Lerdorf y recibió contribuciones bajo una licencia de código abierto en 1995. Principalmente pensado para el desarrollo de aplicaciones Web, inicialmente era un motor de plantillas HTML muy sencillo, pero con el tiempo los desarrolladores de PHP han ido añadiendo funciones de acceso a bases de datos, han reescrito el intérprete, han incorporado soporte orientado a objetos y han mejorado el rendimiento. Actualmente, PHP se ha convertido en un lenguaje muy utilizado para el desarrollo de aplicaciones Web porque se centra en soluciones prácticas y da soporte a las funciones más utilizadas en aplicaciones Web².
- **OLE DB:** Microsoft OLE DB es un conjunto de interfaces OLE/COM que proporciona a las aplicaciones un acceso uniforme a datos almacenados en distintas fuentes de información. La arquitectura OLE DB define a los consumidores de OLE DB y a los proveedores de OLE DB. Un consumidor de OLE DB puede ser cualquier sistema o aplicación que utiliza interfaces OLE DB; un proveedor de OLE DB es un componente que expone las interfaces OLE DB.

El consejo del buen administrador...

Toda programación debe ir acompañada de su correspondiente depuración de errores. Hay estudios que concluyen que se dedica un 60-80 % del tiempo invertido en fabricar un programa a la detección y corrección de errores.

²Para Windows, se pueden encontrar versiones binarias precompiladas de PHP en <http://php.net>.

6.2. Los lenguajes de programación de bases de datos

Se puede programar una base de datos en multitud de lenguajes, desde el C, pasando por el Java, hasta el nuevo e innovador XQuery, pero todos, en mayor o menor medida, terminarán recurriendo a sentencias SQL para extraer o insertar datos de la base de datos, por lo que es imprescindible y fundamental conocer y desenvolverse adecuadamente con SQL.

Para añadir más potencia de ejecución a las sentencias SQL, Oracle desarrolló un lenguaje propio llamado **PL/SQL**, con él, es posible definir variables, crear estructuras de control de flujo y toma de decisiones, crear funciones, procedimientos, paquetes....

Con el mismo objetivo, en DB2 viene incorporado el lenguaje SQL/PL que, al contrario que el de Oracle, respeta la totalidad del estándar de SQL, pero además, DB2, desde su versión 9.7, ofrece la posibilidad de desarrollar directamente en PL/SQL, existiendo compatibilidad plena con Oracle, lo cual hace que las migraciones de la lógica de la aplicación de una base de datos Oracle a DB2 9.7 sean prácticamente transparentes para el desarrollador, facilitando además la reutilización del conocimiento y evitando la necesidad de tener que aprender las características del nuevo gestor (el equipo de trabajo sigue siendo productivo desde el mismo día de la migración).

Los programas PL/SQL, pueden ejecutarse haciendo uso del SQL*Plus; los tipos básicos de un programa PL/SQL son: procedimientos, funciones y bloques anónimos. Todos ellos están compuestos de tres bloques bien diferenciados:

Partes de un programa

```

DECLARE
    --declaración de variables que se usarán en la sección de ejecución
    --es opcional, ya que no siempre la complejidad del programa a
    --realizar precisa del uso de variables y constantes
BEGIN
    --sentencias a ejecutar, bloque de ejecución propiamente dicho.
EXCEPTION
    --chequeo de errores de interés ocurridos durante todo la ejecución
    --y acciones a tomar en consecuencia
    --Es opcional, pero si existe, está ubicado junto antes de END
END;
/

```

El consejo del buen administrador...

Cuantos más comentarios se pongan en un programa, más fácil será su mantenimiento y posterior modificación, sobre todo si la persona encargada de retocarlo no es la misma que la persona que creó el programa originalmente.

Los bloques anónimos solo se pueden utilizar en el momento de su creación, ya que al no tener nombre que los identifique, no pueden ser referenciados. Si lo que se pretende es reutilizar el código, habrá que guardar el programa PL/SQL en el catálogo de la base de datos, ya sea como procedimiento almacenado o función almacenada, de modo que quede identificado únicamente con un nombre y un tipo. Tanto el procedimiento como la función podrán aceptar parámetros de entrada, pero solo la función devolverá a su vez un valor al término de su ejecución. En caso de querer crear un procedimiento, se sustituirá la palabra *DECLARE* por la sentencia ***CREATE PROCEDURE***.

```
CREATE OR REPLACE PROCEDURE nombre_del_procedimiento (lista_de_parámetros) AS
```

Y si lo que se desea es crear una función, se sustituirá la palabra *DECLARE* por la sentencia ***CREATE FUNCTION***.

```
CREATE OR REPLACE FUNCTION nombre_de_la_función(lista_de_parámetros)  
RETURN tipo_de_dato_devuelto IS
```

En ambos casos, la lista de parámetros es opcional, de modo que si no se precisa pasar parámetro alguno, basta con omitir la lista (incluyendo los paréntesis).

Para hacer uso de un procedimiento basta con invocar la instrucción **CALL**.



```
CALL nombre_del_procedimiento(lista_de_valores);
```

En el caso de las funciones, puesto que estas tienen necesariamente que devolver un valor, para hacer uso de ellas es obligatorio recoger dicho valor. Una forma fácil sería asignándoselo a una variable que fuese del mismo tipo que el valor devuelto por la función, o también, invocarla desde una sentencia *SELECT*.

```
SELECT  
nombre_de_la_función(lista_de_valores)  
FROM DUAL;
```

También se pueden agrupar procedimientos y funciones en objetos más grandes llamados paquetes, los cuales están compuestos por dos componentes:

- **La cabecera:** encargada de almacenar la definición de todos los subprogramas (procedimientos y funciones) que incluye el paquete, así como la declaración de las variables y constantes globales al paquete (que podrán usarse en todos los subprogramas que lo componen).
- **El cuerpo:** que contiene el código de todos los subprogramas.

cabecera del paquete

```
CREATE OR REPLACE PACKAGE nombre_del_paquete AS
PROCEDURE nombre_del_procedimiento_1 (lista_de_sus_parámetros);
PROCEDURE nombre_del_procedimiento_2 (lista_de_sus_parámetros);
FUNCTION nombre_de_la_función_1 (lista_de_sus_parámetros) RETURN
tipo_de_dato_devuelto;
--declaración de variables y constantes globales al paquete
END nombre_de_paquete;
/
```

Cuerpo del paquete

```
CREATE OR REPLACE PACKAGE BODY nombre_del_paquete AS

PROCEDURE nombre_del_procedimiento_1 (lista_de_sus_parámetros) IS
    --declaración de variables y constantes locales al procedimiento
BEGIN
    --código del nombre_del_procedimiento_1
END nombre_del_procedimiento_1;

PROCEDURE nombre_del_procedimiento_2 (lista_de_sus_parámetros) IS
    --declaración de variables y constantes locales al procedimiento
BEGIN
    --código del nombre_del_procedimiento_2
END nombre_del_procedimiento_2;

FUNCTION nombre_de_la_funcion_1 (lista_de_sus_parámetros) RETURN
tipo_de_dato_devuelto IS
    --declaración de variables y constantes locales a la función
BEGIN
    --código del nombre_de_la_funcion_1
END nombre_de_la_funcion_1;

END nombre_del_paquete;
/
```

Recuerda. En Oracle, la tabla *DUAL* es una tabla auxiliar muy útil para ejecutar todo tipo de funciones, concretamente las que proporciona el gestor para por ejemplo, extraer el DDL de un objeto de la base de datos, o para conocer qué fecha es:

SELECT SYSDATE FROM DUAL;

En DB2, la tabla auxiliar se llama *SYSIBM.SYSDUMMY1*. El mismo ejemplo para obtener la fecha actual se haría con la sentencia:

SELECT CURRENT_TIMESTAMP FROM SYSIBM.SYSDUMMY1;

Los procedimientos y funciones que están dentro de paquetes, también pueden ser invocados individualmente y, de la misma manera que los procedimientos y funciones independientes, basta con calificarlos por delante con el nombre del paquete al que pertenecen, ya que esta es la única forma de identificarlos únicamente respecto de la totalidad de objetos que tiene la base de datos.

— Llamada dentro de un paquete —

```
--llamada a procedimiento  
CALL nombre_del_paquete.nombre_del_procedimiento(lista_de_valores);  
  
--llamada a función  
SELECT nombre_del_paquete.nombre_de_la_función(lista_de_valores)  
FROM dual;
```

- ◊ **Actividad 6.1:** Crea dos tablas que se llamen igual pero en distintos esquemas. Crear, en el mismo esquema, dos objetos que se llamen exactamente igual, por ejemplo, una tabla y un procedimiento.

A parte de los programas mencionados (procedimientos, funciones y paquetes), también se puede codificar lógica de negocio en los disparadores o *TRIGGERS* asociados a tablas, vistas o eventos, que podrán ejecutarse una vez (por ejemplo al intentar un usuario conectarse a la base de datos), o tantas veces como filas de una tabla se vayan a insertar, borrar o actualizar. El disparador podrá programarse para que se active inmediatamente antes o después del evento.

Para dar de alta o recrear un disparador basta con sustituir la palabra *DECLARE* por la sentencia de definición del disparador. Un ejemplo para definir un disparador de tabla que se ejecute en cada fila antes de ser insertada es el siguiente:

Creación o recreación de un disparador

```
CREATE OR REPLACE TRIGGER nombre_del_disparador BEFORE INSERT  
ON nombre_de_la_tabla FOR EACH ROW
```

En ese caso, el momento en el que se ejecuta el disparador es *BEFORE* (antes), y el evento que lo dispara es la instrucción DML *INSERT* (inserción en la tabla destino del disparador). Otro momento de ejecución podría ser *AFTER* (después), y otros eventos, además de la inserción, podrían ser el borrado (*DELETE*) y la actualización (*UPDATE OF lista_de_columnas*), o incluso varios a la vez (*INSERT OR DELETE OR UPDATE*). Además, también se podría añadir una condición (*WHEN*). Por ejemplo, se podría crear un disparador llamado *actualizaSueldoMinimo* definido antes (*BEFORE*) de la actualización (*UPDATE OF*) del campo *sueldo* de la tabla *nominas* del esquema *contratados*, que se activara solo cuando se cumpliera la condición de que el valor del campo que se quiere actualizar fuese menor de 1000 euros, de modo que impidiese que se pudieran asignar sueldos por debajo de los 1000 euros.

```
CREATE OR REPLACE TRIGGER contratados.actualizaSueldoMinimo BEFORE UPDATE OF  
sueldo ON contratados.nominas FOR EACH ROW  
WHEN (new.sueldo < 1000)  
BEGIN  
    :new.sueldo := 1000;  
END;  
/
```

¿Sabías que . . . ? Se puede definir más de un disparador para la misma tabla y para la misma combinación de momento y evento, pero el orden en el que Oracle los ejecutará será indeterminado, por ello es mejor opción agrupar la lógica de negocio en un solo disparador.

DB2, en cambio, los ejecutaría según el orden en el que fueron creados.

Dentro de la programación del disparador, *:old.nombre_del_campo* hará referencia al valor antiguo del campo llamado *nombre_del_campo*, mientras que *:new.nombre_del_campo* será el nuevo valor que pretende asignársele al campo llamado *nombre_del_campo*, pudiendo este valor *new* ser modificado en el bloque de ejecución del disparador. Nótese que en la cláusula *WHEN*, las variables *:old* y *:new* no deben ir precedidas de los dos puntos (:).

Recuerda. En una base de datos todos los objetos son únicos, es decir, que están perfectamente identificados por su nombre y por su tipo y no pueden repetirse. Por ejemplo, un mismo usuario o esquema de base de datos no podría tener dos tablas que se llamen exactamente igual, pero sí podría existir una tabla llamada *nominas* en dos esquemas distintos (*contratados* y *subcontratados*), ya que la base de datos las conocería como *contratados.nominas* y *subcontratados.nominas*.

En el caso de los procedimientos y funciones, existe además el nivel de paquetes, es decir, que podría existir un procedimiento independiente que se llamara *calculaNominas* que perteneciera al usuario *pepito*, pero además podría también haber otro procedimiento *calculaNominas* dentro de un paquete *paqueteDeContabilidad* del mismo esquema *contratados*. Se referenciarían por *contratados.calculaNominas*, *contratados.paqueteDeContabilidad.calculaNominas*, respectivamente. Incluso podría existir una función que se llamara también *contratados.calculaNominas*, pues al ser objetos de distinto tipo, en una sentencia SQL determinada, la base de datos los distinguiría por el contexto.

6.3. Tipos de datos, identificadores y variables

En Oracle existen varios juegos de datos soportados, unos específicos de Oracle, otros compatibles con el estándar ANSI y los tipos de datos del DB2 de IBM, otros definidos por el usuario a partir de los dos tipos anteriores mediante el uso de la instrucción *CREATE TYPE*, y otros como los tipos de datos XML (para la consulta y tratamiento de documentos XML) o los tipos de datos espaciales (para tratar la información geográfica). Aquí solo se tratarán los tipos básicos del primer juego:

- **CHAR(tamaño):** cadena de tamaño fijo. El número máximo de caracteres es 2000. Los valores de tipo cadena deberán ir entrecomillados con comilla simple.
- **VARCHAR2(tamaño):** cadena de longitud variable. El tamaño máximo es 4000 caracteres. Los valores de tipo cadena deberán ir entrecomillados con comilla simple.
- **NUMBER(precisión,escala):** número decimal. La precisión puede tomar valores entre 1 y 38. La escala puede estar entre -84 y 127. Un número de escala positivo indica cuántos dígitos significativos de entre el total de la precisión se guardarán a la derecha de la coma decimal. Un número de escala

negativo indica el número de dígitos que se redondearán del número a la izquierda de la coma. Tanto la precisión como la escala son opcionales, pero si hubiera escala, necesariamente tendría que haberse indicado la precisión. Por ejemplo, para crear una variable que pueda albergar valores del –999999 hasta el 999999, basta con definir una de tipo NUMBER(6). Una variable de tipo NUMBER(6,–2) aceptará valores de hasta seis cifras, pero las unidades y las decenas se redondearán, por ejemplo, el número 123456,44 se almacenará como 123500. En cambio, NUMBER(6,2) guardará cuatro dígitos a la izquierda de la coma y redondeará los dígitos a la derecha de la coma dejando solo dos. Los subtipos *INTEGER*, *INT*, y *SMALLINT* también pueden usarse para definir números enteros con un máximo de 38 dígitos de precisión.

- **BINARY_FLOAT:** número de coma flotante de 32 bits de precisión. Los valores positivos están en el intervalo (1, 17549E–38, 3, 40282E+38).
- **BINARY_DOUBLE:** número de coma flotante de 64 bits de precisión. Los valores positivos están en el intervalo (2, 22507485850720E–308, 1, 79769313486231E+308).
- **DATE:** fecha y hora. La función *TO_DATE* es muy útil para insertar datos tipo fecha o comparar dos fechas: *TO_DATE* ('31-01-2010', 'DD-MM-YYYY').
- **TIMESTAMP:** es una extensión del tipo *DATE* que guarda además fracciones de segundo.
- **BLOB,CLOB,BFILE**³: datos muy grandes y desestructurados, por ejemplo texto, imágenes, vídeos o datos espaciales. Pueden guardarse en tablespaces diferentes al resto de la tabla, o en el caso de los *BFILE*, en ficheros externos a la base de datos.
- **ROWID:** es el tipo de la pseudocolumna ROWID, una columna interna de la tabla que almacena la dirección física de cada fila. Mediante esta dirección se puede localizar el número de objeto, fichero de datos, tablespace donde se encuentra y la posición dentro del bloque de datos respecto del fichero de datos, formando con todo ello un valor único en toda la base de datos. Los accesos a los datos a través de la localización de un valor en esta columna son rapidísimos.

Además de estos tipos, existen otros no soportados por el estándar SQL, que no pueden usarse para definir columnas, pero sí en la programación de PL/SQL, por ejemplo el tipo *BOOLEAN*, que puede albergar los valores lógicos *TRUE* (verdadero), *FALSE* (falso) o *NULL* (nulo) muy útiles en comparaciones.

³Oracle recomienda sustituir los datos tipo LONG, LONG RAW y RAW por estos tipo LOB.

Basándose en estos tipos de datos, se podrá proceder a la definición de variables y constantes en los bloques *DECLARE*, y en la definición de los procedimientos y funciones.

Sintaxis para la definición de variables.

```
nombre_de_la_variable tipo_de_dato ;  
o  
nombre_de_la_variable tipo_de_dato := valor_de_inicialización;
```

Sintaxis para la definición de constantes

```
nombre_de_la_constante CONSTANT tipo_de_dato := valor_de_inicialización;
```

¿Sabías que ...? Los valores de tipo cadena van entrecomillados con comilla simple.

En PL/SQL, la asignación del valor a una variable o constante siempre va precedida de los signos `:=`. En la asignación u operación con valores, PL/SQL puede convertir implícitamente el tipo de dato para adecuarlo al tipo de la variable al que se asigna el valor o al tipo necesario en la operación. En el siguiente ejemplo, se puede hacer la resta gracias a la conversión implícita de los meses. En caso de que los meses tuvieran alguna letra, la operación arrojaría un error.

```
DECLARE  
    mesInicio CHAR(2) := '6';  
    mesFin CHAR(2) := '10';  
    mesesTranscurridos NUMBER(2);  
BEGIN  
    mesesTranscurridos := mesFin - mesInicio;  
END;  
/
```

Cuando se desee declarar una variable con el mismo tipo que otra variable conocida del mismo programa o el tipo de una columna de una tabla concreta, o incluso como una fila completa de una tabla, existen dos atributos que facilitan la definición y con ello el mantenimiento del programa PL/SQL, ya que la modificación del tipo original se extenderá al resto de variables derivadas sin necesidad de tener que cambiar ninguna línea de código. Esos atributos son el `%TYPE` y el `%ROWTYPE`.

Atributos %TYPE y %ROWTYPE

```
nombre_de_la_variable nombre_de_la_otra_variable%TYPE;
nombre_de_la_variable esquema.tabla.columna%TYPE;
nombre_de_la_variable_tipo_registro esquema.tabla%TYPE;
```

En el caso de los procedimientos y funciones, las variables de su lista de parámetros pueden ser de tres tipos: *IN* (parámetro de entrada), *OUT* (parámetro de salida) e *IN OUT* (parámetro de entrada y de salida). Si no se especifica nada, el parámetro en cuestión sería *IN*. Hacer que un procedimiento tenga parámetros *OUT* es un truco muy bueno para conseguir que el procedimiento devuelva datos.

Procedimiento con parámetro de salida

```
CREATE OR REPLACE PROCEDURE contratados.dameSueldoMinimo
(sueldoMinimo OUT contratados.nominas.sueldo%TYPE) AS
BEGIN
    sueldoMinimo:=1000;
END;
/
```

¿Sabías que ...? La función *SUBSTR(cadena_argumento, posición_inicial, longitud_de_la_subcadena)* sirve para extraer trozos de la *cadena_argumento*, de modo que se devolverán un total de *longitud_de_la_subcadena* caracteres comenzando por los que van desde la *posición_inicial*. En caso de omitir el parámetro *longitud_de_la_subcadena*, la función devolverá la subcadena desde la *posición_inicial* hasta el final de la *cadena_argumento*.

6.4. Operadores y expresiones

Para relacionar variables y constantes entre sí, se usan los operadores, que combinándolas con aquéllas darán lugar a expresiones cuyos resultados serán el fundamento de la lógica de programación de la base de datos.

Las operaciones que se pueden realizar y el orden en el que se ejecutarían (en ausencia de paréntesis) son:

Operador	Acción
**	potencia
+ - (unarios)	Signo positivo o negativo
*	multiplicación
/	división
+	suma
-	resta
	concatenación. Muy útil para juntar varias cadenas en una
=, <, >, <=	comparaciones: igual, menor, mayor, menor o igual
>=, <>, !=	mayor o igual, distinto, distinto
IS NULL, LIKE	es nulo, como
BETWEEN, IN	entre, en
NOT	negación lógica de un tipo boolean
AND	operador AND lógico entre tipos de dato boolean
OR	operador OR lógico entre tipos de dato boolean

Teniendo en cuenta esto, se podría decir que las siguientes expresiones son todas verdaderas (TRUE):

$(6 + 8) / 2 = 7$	$6 + 8 / 2 = 10$	$'6a2' = '6' 'a2'$
$7 != 10$	$0 <> \text{NULL}$	$0 \text{ IS NULL} = \text{FALSE}$
$0 \text{ IS NOT NULL} = \text{TRUE}$	$6 \text{ BETWEEN } 2 \text{ AND } 8$	$'6a2' \text{ LIKE } '%2'$
$'6a2' \text{ LIKE } '6 %'$	$6 \text{ IN } (6,8,2)$	$'a' \text{ IN } ('b','a')$
$'c' \text{ NOT IN } ('b','a')$	$(3 < 5) \text{ AND } (5 >= 5)$	$(3 < 5) \text{ OR } (5 <= 3)$

Además de los operadores, existen funciones tremadamente útiles para la toma de decisiones. Tal es el ejemplo de las funciones DECODE, NVL y REPLACE.

- **DECODE(argumento, patrón1, resultado1, patrón2, resultado2..., resultado_por_defecto):** Compara el valor del argumento con cada uno de los patrones y en cuando encuentra la coincidencia devuelve el resultado correspondiente, o el *resultado_por_defecto* en caso de que no encuentre coincidencia en ningún patrón proporcionado. Cualquiera de los patrones puede tomar el valor *NULL*.
- **NVL(valor1, valor2):** Si el valor1 es nulo, entonces se devuelve el valor2, en caso contrario se devuelve el valor1.
- **REPLACE(cad_argumento, cad_a_reemplazar, cad_que_reemplaza):** Devuelve la *cad_argumento* sustituyendo en ella cada ocurrencia de *cad_a_reemplazar* por *cad_que_reemplaza*, pudiendo tomar esta última el valor *NULL*, de modo que entonces se borrarían todas las *cad_a_reemplazar* que hubiera en *cad_argumento*.

6.5. Estructuras de control

Cuando se crea un programa, todas las sentencias se ejecutan siguiendo una secuencia, es decir, el flujo de ejecución del programa consiste en ejecutar una instrucción detrás de otra. Las estructuras de control sirven para controlar ese flujo del programa, la toma de decisiones y programar las acciones en consecuencia. Todas las estructuras de control están basadas en la evaluación de una expresión lógica más o menos compleja que constituye en sí misma una condición. En caso de cumplirse la condición, es decir, en caso de que la evaluación de la condición dé como resultado el valor *TRUE*, se tomará una acción, en caso negativo otra o nada. Hay dos tipos de sentencias de control de flujo, condicionales e iterativas. En las condicionales, la decisión que se toma es alternativa, es decir, se ejecuta un bloque de sentencias u otro. En las iterativas o *bucles*, la decisión que se toma es si repetir la ejecución de un bloque de instrucciones o no repetirlo.

6.5.1. IF..THEN-ELSIF..THEN-ELSE-END IF

Hay tres formas de usar la sentencia condicional *IF*: evaluando una única condición (*IF-THEN*), más de una condición (*ELSIF-THEN*), y/o añadiendo una opción de acción por defecto en caso de que no se cumpla ninguna condición de las exigidas (*ELSE*). Las cláusulas *ELSIF* y *ELSE* son opcionales.

Ejemplo de condicional

```
CREATE OR REPLACE PROCEDURE Calificacion(Nota NUMBER) as
BEGIN
    IF Nota >= 0 AND Nota < 5 THEN
        DBMS_OUTPUT.PUT_LINE('Suspenso');
    ELSIF Nota >= 5 AND Nota < 6 THEN
        DBMS_OUTPUT.PUT_LINE('Suficiente');
    ELSIF Nota >= 6 AND Nota < 7 THEN
        DBMS_OUTPUT.PUT_LINE('Bien');
    ELSIF Nota >= 7 AND Nota < 9 THEN
        DBMS_OUTPUT.PUT_LINE('Notable');
    ELSIF Nota >= 9 AND Nota <= 10 THEN
        DBMS_OUTPUT.PUT_LINE('Sobresaliente');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Calificación errónea');
    END IF;
END;
/
```

En el ejemplo anterior se hace uso de la función *DBMS_OUTPUT.PUT_LINE* para escribir el resultado de la evaluación de cada condición. La misión de dicha función

es la de mostrar por pantalla un mensaje visible al usuario, pero para que este pueda verlo, ha de activarse la variable de entorno del SQL*Plus llamada *SERVERROUT-PUT*, que es local a la sesión actual en curso.

6.5.2. CASE-WHEN..THEN-ELSE-END CASE

Es una estructura de control condicional que permite ejecutar distintas sentencias ante los distintos valores posibles de la misma condición.

En los ejemplos siguientes se exponen dos formas de programar esta estructura, una escribiendo cada vez la condición candidata, y la otra, poniéndola solo una vez.

Ejemplo 1 de CASE-WHEN..THEN-ELSE-END CASE

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE Calificacion(Nota NUMBER) as
BEGIN
CASE
    WHEN Nota >= 0 AND Nota < 5 THEN
        DBMS_OUTPUT.PUT_LINE('Suspenso');
    WHEN Nota >= 5 AND Nota < 6 THEN
        DBMS_OUTPUT.PUT_LINE('Suficiente');
    WHEN Nota >= 6 AND Nota < 7 THEN
        DBMS_OUTPUT.PUT_LINE('Bien');
    WHEN Nota >= 7 AND Nota < 9 THEN
        DBMS_OUTPUT.PUT_LINE('Notable');
    WHEN Nota >= 9 AND Nota <= 10 THEN
        DBMS_OUTPUT.PUT_LINE('Sobresaliente');
    ELSE DBMS_OUTPUT.PUT_LINE('Calificación errónea');
END CASE;
END;
/
```

Ejemplo 2 de CASE-WHEN..THEN-ELSE-END CASE

```

SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE Calificacion(Nota VARCHAR) as
BEGIN
CASE Nota
    WHEN 'Suspens' THEN
        DBMS_OUTPUT.PUT_LINE('Calificación menor de 5.');
    WHEN 'Suficiente' THEN
        DBMS_OUTPUT.PUT_LINE('Calificación entre 5 y 6.');
    WHEN 'Bien' THEN
        DBMS_OUTPUT.PUT_LINE('Calificación entre 6 y 7.');
    WHEN 'Notable' THEN
        DBMS_OUTPUT.PUT_LINE('Calificación entre 7 y 9.');
    WHEN 'Sobresaliente' THEN
        DBMS_OUTPUT.PUT_LINE('Calificación entre 9 y 10.');
    ELSE DBMS_OUTPUT.PUT_LINE('Calificación errónea');
END CASE;
END;
/

```

6.5.3. LOOP-EXIT WHEN-END LOOP

Para construir sentencias iterativas o bucles existen varias opciones, la primera de ellas es esta, que garantiza que por lo menos una vez, la ejecución del programa entra en la codificación hecha dentro del bucle.

Existen dos formas de salir del bucle, en cualquier circunstancia (*EXIT*), o poniendo una condición (*EXIT WHEN*).

Ejemplo de LOOP-EXIT WHEN-END LOOP

```

SET SERVEROUTPUT ON
DECLARE
    contador NUMBER(3);
BEGIN
    contador := 1;
    LOOP
        EXIT WHEN contador > 99;
        contador := contador + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('El valor de salida debería ser 100: ' || contador);
END;
/

```

Ejemplo equivalente de LOOP-EXIT WHEN-END LOOP

```
SET SERVEROUTPUT ON
DECLARE
    contador NUMBER(3);
BEGIN
    contador := 1;
    LOOP
        IF contador > 99 THEN
            EXIT;
        END IF;
        contador := contador + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('El valor de salida debería ser 100: ' || contador);
END;
/
```

6.5.4. WHILE..LOOP-END LOOP

Esta sentencia iterativa es otro tipo de bucle. Se diferencia del anterior en que la ejecución del programa tiene que evaluar la condición antes de ejecutar el bloque de sentencias del bucle.

Ejemplo de WHILE..LOOP-END LOOP

```
SET SERVEROUTPUT ON
DECLARE
    contador NUMBER(3);
BEGIN
    contador := 1;
    WHILE contador < 100 LOOP
        contador := contador + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE ('El valor de salida debería ser 100: ' || contador);
END;
/
```

6.5.5. FOR..IN..LOOP-END LOOP

Esta estructura es un bucle que automáticamente incrementa el valor de un contador dentro de un intervalo fijado, por lo que el número de veces que se ejecutará el bloque de instrucciones es conocido previamente. El contador puede tener cualquier nombre elegido por el usuario y se podrá referenciar como una variable más del programa, pero no necesita ser definido en el bloque *DECLARE*.

Usando en vez de *IN*, la opción *IN REVERSE*, se conseguiría que el bucle fuera contando de manera descendente, de mayor a menor.

En el siguiente ejemplo, el contenido del bucle se ejecuta 100 veces, desde el 1 hasta el 100. Si el límite inferior del contador coincide con el superior (*FOR i IN 100..100 LOOP*), el cuerpo del bucle se ejecutaría una sola vez, otorgando el valor 100 a la variable *i*.

Ejemplo de FOR..IN..LOOP-END LOOP

```
SET SERVEROUTPUT ON
DECLARE
    contador NUMBER(3);
BEGIN
    FOR contador IN 1..100 LOOP
        DBMS_OUTPUT.PUT_LINE ('El valor del contador es: ' || contador);
    END LOOP;
END;
/
```

Es posible interrumpir la ejecución del bucle y salir de él. Como en el caso de los bucles simples, pueden usarse en cualquier momento dentro de la programación del bucle *FOR* las cláusulas *EXIT* y *EXIT WHEN*.

Los bucles *FOR* son ideales para recorrer los registros resultantes de una consulta (sentencia *SELECT*), ya sea mediante el uso de cursosres⁵ o directamente incluyendo la sentencia en la cláusula *IN*. Véase los ejemplos.

Ejemplo de FOR..IN..LOOP-END LOOP con consulta

```
SET SERVEROUTPUT ON
DECLARE
BEGIN
    DBMS_OUTPUT.PUT_LINE ('Los sueldos ordenados de menor a mayor son:');
    FOR i IN (SELECT sueldo
               FROM contratados.nominas
              ORDER BY sueldo) LOOP
        DBMS_OUTPUT.PUT_LINE (i.sueldo);
    END LOOP;
END;
/
```

⁵Un cursor es una estructura de datos concebida para recorrer el resultado de una consulta (sentencia *SELECT*). Mediante el uso de la instrucción *FETCH nombre_del_cursor INTO lista_de_variables*, el registro objeto de la consulta se extrae del cursor y se introduce dentro de una variable, listo para ser referenciado.

Ejemplo de FOR..IN..LOOP-END LOOP con cursor

```
SET SERVEROUTPUT ON
DECLARE
    vSueldo contratados.nominas.sueldo%TYPE;
    CURSOR curSueldo IS SELECT sueldo FROM contratados.nominas
        ORDER BY sueldo DESC;
BEGIN
    OPEN curSueldo;
    DBMS_OUTPUT.PUT_LINE ('Los diez sueldos más altos (de mayor a menor):');
    FOR i IN 1..10 LOOP
        FETCH curSueldo INTO vSueldo;
        DBMS_OUTPUT.PUT_LINE (vSueldo);
    END LOOP;
    CLOSE curSueldo;
END;
/
```

Pero, ¿qué pasaría si en el ejemplo del bucle *FOR con cursor* no hubiera más de 3 registros en la tabla *contratados.nominas*? Al intentar imprimir el dato del sueldo tras hacer el *FETCH* cuando *i* fuese ≥ 4 , aparecería repetido el valor del último sueldo recuperado, esto es debido a que, como la tabla no tiene más registros, el *FETCH* no consigue volver a inicializar la variable *vSueldo*. Para evitar ese mal funcionamiento se podría optar por incluir una cláusula *EXIT WHEN* que obligara a abandonar el lazo al llegar al final del cursor (atributo *NOTFOUND* del cursor).

Ejemplo de FOR..IN..LOOP-END LOOP con cursor y EXIT WHEN

```
SET SERVEROUTPUT ON
DECLARE
    vSueldo contratados.nominas.sueldo%TYPE;
    CURSOR curSueldo IS SELECT sueldo FROM contratados.nominas
        ORDER BY sueldo DESC;
BEGIN
    OPEN curSueldo;
    DBMS_OUTPUT.PUT_LINE ('Los diez sueldos más altos (de mayor a menor):');
    FOR i IN 1..10 LOOP
        FETCH curSueldo INTO vSueldo;
        EXIT WHEN curSueldo%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (vSueldo);
    END LOOP;
    CLOSE curSueldo;
END;
/
```

6.6. Gestión de errores

Algo imprescindible en toda programación es la codificación de una sección para interceptar posibles errores que pudieran ocurrir durante la ejecución del programa (ya sean errores conocidos o desconocidos, previstos o imprevistos), y en consecuencia, tomar las medidas oportunas o escribir un mensaje apropiado. Para ello se usa el bloque *EXCEPTION* con la cláusula *WHEN..THEN* que sirve para filtrar el tipo de error deseado.

Algunos de los errores predefinidos que pueden interceptarse son:

- **CASE_NOT_FOUND:** ninguno de las condiciones de la sentencia *WHEN* en la estructura *CASE* se corresponde con el valor evaluado y no existe cláusula *ELSE*.
- **CURSOR_ALREADY_OPEN:** el cursor que intenta abrirse ya está abierto.
- **INVALID_CURSOR:** la operación que está intentando realizarse con el cursor no es válida, por ejemplo, porque quiera cerrarse un cursor que no se ha abierto previamente.
- **INVALID_NUMBER o VALUE_ERROR:** la conversión de una cadena a valor numérico no es posible porque la cadena no representa un valor numérico válido.
- **VALUE_ERROR:** error ocurrido en alguna operación aritmética, de conversión o truncado, por ejemplo, cuando se intenta insertar en una variable un valor de más tamaño.
- **LOGIN_DENIED:** un programa está intentando acceder a la base de datos con un usuario o password incorrectos.
- **NOT_LOGGED_ON:** un programa está intentando ejecutar algo en la base de datos sin haber formalizado previamente la conexión.
- **NO_DATA_FOUND:** una sentencia *SELECT INTO* no devuelve ningún registro.
- **TOO_MANY_ROWS:** una sentencia *SELECT INTO* devuelve más de un registro.
- **TIMEOUT_ON_RESOURCE:** se ha acabado el tiempo que el SGBD puede esperar por algún recurso.
- **ZERO_DIVIDE:** algún programa intenta hacer una división de un número entre cero.
- **OTHERS:** es la opción por defecto. Interceptará todos los errores no tenidos en cuenta en las condiciones *WHEN* de la cláusula *EXCEPTION* donde se encuentre.

Ejemplo de gestión de errores

```
SET SERVEROUTPUT ON
DECLARE
    mesInicio CHAR(2) := '6a';
    mesFin CHAR(2) := '10';
    mesesTranscurridos NUMBER(2);
BEGIN
    mesesTranscurridos := mesFin - mesInicio;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE('Error INVALID_NUMBER interceptado.');
    WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Error VALUE_ERROR interceptado.');
END;
/
```

Ejemplo de CASE-WHEN..THEN-END CASE sin ELSE y con gestión de errores

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE Nota(Calificacion VARCHAR) as
BEGIN
    CASE Calificacion
        WHEN 'Suspens' THEN
            DBMS_OUTPUT.PUT_LINE('Nota >=0 y <5');
        WHEN 'Suficiente' THEN
            DBMS_OUTPUT.PUT_LINE('Nota >=5 y <6');
        WHEN 'Bien' THEN
            DBMS_OUTPUT.PUT_LINE('Nota >=6 y <7');
        WHEN 'Notable' THEN
            DBMS_OUTPUT.PUT_LINE('Nota >=7 y <9');
        WHEN 'Sobresaliente' THEN
            DBMS_OUTPUT.PUT_LINE('Nota >=9 y <=10');
    END CASE;

    EXCEPTION
        WHEN CASE_NOT_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Calificación errónea');

    END;
/
```

Cuando se desee interceptar cualquier error, la condición *OTHERS* es ideal, pero para conocer qué error exactamente hizo saltar la excepción se deben usar las funciones especiales *SQLCODE* y *SQLERRM*.

Ejemplo de gestión de errores extrayendo los códigos de error

```
SET SERVEROUTPUT ON
DECLARE
    sueldoMaximo NUMBER(4); --saltará excepción si sueldo>9999
    codigoError NUMBER;
    textoError VARCHAR2(64);
BEGIN
    SELECT MAX(sueldo) INTO sueldoMaximo FROM contratados.nominas;
    DBMS_OUTPUT.PUT_LINE('El sueldo más alto es:'||sueldoMaximo);
EXCEPTION
    WHEN OTHERS THEN
        codigoError := SQLCODE;
        textoError := SUBSTR(SQLERRM, 1, 64);
        DBMS_OUTPUT.PUT_LINE('El código del error interceptado es: ' ||
            codigoError || ' y su texto: ' || textoError);
END;
/
```

La variable *sueldoMaximo* está definida como NUMBER(4) y solo admitirá valores entre -9999 y 9999. Si en la tabla de nóminas hay algún empleado que gane más de 9999 euros, la sentencia SELECT Max(sueldo) retornará un valor que no puede ser almacenada dentro de la variable *sueldoMaximo* generándose así una excepción:

El código del error interceptado es: -6502 y su texto: ORA-06502:
PL/SQL: error: precision de numero demasiado grande

6.7. Transacciones en scripts

Cuando en los programas además de consultar los objetos de la base de datos también se hacen modificaciones en los mismos, resulta de vital importancia conocer qué es una transacción. Es la herramienta que tienen las bases de datos para garantizar la integridad de los datos.

Tal y como se introdujo en el capítulo 5, una transacción es una serie de instrucciones de manipulación de datos (DML) que constituyen una unidad de trabajo completa. Si el programa fallara en mitad de una transacción, debería ser capaz de deshacer todos los cambios hechos hasta ese momento en la transacción para dejar las tablas de la base de datos tal y como estaban originalmente.

Por ejemplo, en cualquier operación de transferencia bancaria, una unidad completa de trabajo sería sumar el dinero a la cuenta destino y restarlo de la cuenta origen. Si después de sumar el dinero a la cuenta destino, la transacción fallara al restarlo de la cuenta origen, el programa debería deshacer la suma del dinero a la

cuenta destino, de otro modo, el banco origen perdería el dinero de la transferencia.

Recuerda. A la operación de deshacer se la conoce como hacer *ROLLBACK*, mientras que la operación de validar el total de la transacción es hacer *COMMIT*. Explícitamente, se pueden escribir esos comandos (*ROLLBACK* y *COMMIT*) en el programa para terminar la transacción, no obstante, debe tenerse en cuenta que existen operaciones que implican un commit implícito. Tal es el caso de las sentencias de definición de datos (DDL), es decir, si en medio de una transacción se creara una nueva tabla, implícitamente se estaría haciendo un commit, con lo que quedarían validadas todas las operaciones anteriores y se pondría fin a la transacción. Otras formas de hacer commit implícito es ejecutar un simple *DISCONNECT* para cerrar la conexión del usuario actual a la base de datos, o ejecutar *EXIT* o *QUIT* para salir de la sesión actual del SQL*Plus; en cambio, si en vez de salir ordenadamente del SQL*Plus se abortara su sesión, la base de datos haría un rollback implícito.

-
- ◊ **Actividad 6.2:** Para comprobar el commit o rollback implícito en Oracle, crea con SQL*Plus una tabla y haz pruebas con ella: insértale un registro, valídalo con *COMMIT*, inserta otro y aborta la sesión del SQL*Plus, vuelve a abrir una sesión de SQL*Plus, comprueba que el número de registros que hay en la tabla es solo uno, inserta otro, crea otra tabla (se hará commit implícito de la sesión), aborta la sesión de SQL*Plus y comprueba que esta vez sí que hay dos registros en la tabla original.
-

¿Sabías que . . . ? En muchos SGBD también es muy usado lo que se conoce como TWO_PHASE_COMMIT, que es una forma de definir transacciones en dos partes, de modo que una parte pertenecería a una base de datos y otra a otra. Esto es muy corriente en sistemas que están distribuidos en varias bases de datos, por ejemplo, los datos de banca tradicionalmente se han encontrado almacenados en bases de datos residentes en máquinas HOST, a las que tiene que accederse desde bases de datos localizadas en máquinas UNIX para hacer determinadas operaciones, las cuales solo podrán quedar validadas si las operaciones hechas en las bases de datos del HOST han tenido éxito, por lo que la transacción completa está dividida en dos partes, una en HOST y otra en UNIX, pero solo podrá ser validada si tuvo éxito en ambos entornos.

A continuación se ofrece un ejemplo en el que se actualiza el sueldo a todos los empleados contratados. La actualización consiste en subirles el IPC, pero solo deberá validarse en el caso de que todas las actualizaciones hayan tenido éxito.

Ejemplo de subida de sueldo

```
SET SERVEROUTPUT ON
DECLARE
    ipc NUMBER(2,1) := 3.2;
    codigoError NUMBER;
    textoError VARCHAR2(64);

BEGIN
    FOR i IN (SELECT sueldo, nif
               FROM contratados.nominas) LOOP
        UPDATE contratados.nominas SET sueldo = sueldo + sueldo * ipc / 100
        WHERE nif = i.nif;
    END LOOP;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        codigoError := SQLCODE;
        textoError := SUBSTR(SQLERRM, 1, 64);
        DBMS_OUTPUT.PUT_LINE('El código del error interceptado es: ' ||
            codigoError || ' y su texto: ' || textoError);
        ROLLBACK;
END;
/
```

Otro ejemplo servirá para ilustrar el mismo caso anterior de subida de sueldo masiva, pero en esta ocasión, en vez de validar la transacción solo si todas las actualizaciones han ido bien, se irá haciendo commit cada mil sueldos modificados, de modo que se minimice el espacio necesario en el tablespace de UNDO de la base de datos.

El consejo del buen administrador...

En actualizaciones, inserciones o borrados masivos, hay que tener presente el espacio para deshacer los cambios de que se dispone (espacio del tablespace de UNDO).

Ejemplo de subida de sueldo cada 1000 actualizaciones

```
SET SERVEROUTPUT ON
DECLARE
    ipc          CONSTANT NUMBER(2,1) := 3.2;
    contador     NUMBER(4) := 0;
    codigoError  NUMBER;
    textoError   VARCHAR2(64);
    contadorErrores NUMBER(38) := 0;
BEGIN
    FOR i IN (SELECT sueldo, nif
               FROM contratados.nominas) LOOP
        BEGIN
            UPDATE contratados.nominas SET sueldo = sueldo + sueldo * ipc / 100
              WHERE nif = i.nif;
            --Se pone un bloque de excepción aquí para evitar que en caso de error
            --en la sentencia de actualización el programa se acabe, de este modo,
            --el bucle continuaría actualizando el resto de sueldos
        EXCEPTION
            WHEN OTHERS THEN
                codigoError := SQLCODE;
                textoError := SUBSTR(SQLERRM, 1, 64);
                DBMS_OUTPUT.PUT_LINE('Error en el UPDATE: ' || codigoError ||
                                     ' ' || textoError);
                contadorErrores := contadorErrores + 1;
        END;
        contador := contador + 1;
        IF contador >= 1000 THEN
            COMMIT;
            contador := 0;
        END IF;
    END LOOP;
    -- Aquí se pone otro commit para validar los últimos sueldos actualizados
    -- (que no hayan cumplido la condición de que contador sea 1000)
    COMMIT;
    DBMS_OUTPUT.PUT_LINE('El nº de sueldos no actualizados por error fueron:' ||
                         ' ' || contadorErrores);
EXCEPTION
    WHEN OTHERS THEN
        codigoError := SQLCODE;
        textoError := SUBSTR(SQLERRM, 1, 64);
        DBMS_OUTPUT.PUT_LINE('Error inesperado:' || codigoError ||
                             ' ' || textoError);
        ROLLBACK;
END;
/
```

6.8. Las secuencias

En Oracle, al contrario que en MySQL no existe el tipo AUTO_INCREMENT. En su lugar, hay un objeto especial llamado *secuencia* que sirve para obtener valores secuenciales autoincrementados. Es más manual que el campo AUTO_INCREMENT de MySQL, pero también es mucho más versátil, pudiendo ser utilizado en otros sitios aparte de la clave primaria de una tabla.

La sintaxis para la creación de estos objetos es:

```
CREATE SEQUENCE secuencia
    INCREMENT BY n
    START WITH n
    {MAX VALUE n | NOMAXVALUE}
    {MIN VALUE n | NOMINVALUE}
    {CACHE n | NOCACHE};
```

Por ejemplo:

```
CREATE SEQUENCE NumFactura INCREMENT BY 1 START WITH 20
NOMAXVALUE CACHE 40;
```

Con este comando se obtiene un objeto secuencia llamado NumFactura que comenzará en 20 y se incrementará de uno en uno sin valor máximo. Además, en la memoria caché del SGBD se almacenarán los 40 números siguientes por lo que el acceso a la secuencia será muy rápido.

Se puede utilizar este tipo de objetos mediante una SELECT de forma muy sencilla haciendo uso de las funciones CURRVAL (valor actual) y NEXTVAL (valor siguiente).

```
--para incrementar la secuencia y devolver el número siguiente
SELECT NumFactura.NEXTVAL from DUAL;
--para devolver el número de factura actual
SELECT NumFactura.CURRVAL from DUAL;
```

6.9. Prácticas Resueltas

Práctica 6.1: Bajada de sueldo un 5%

Con motivo de una crisis económica se tiene que proceder a bajar el sueldo a todos los funcionarios un 5 %. Implementar un procedimiento en PL-SQL de Oracle que, basándose en una tabla inventada, simule cómo se haría esa disminución masiva del sueldo, para ello:

1. Crea el esquema *funcionarios* al que pertenecerán todos los objetos del problema, otorgarle permiso de *DBA* y abrir una sesión con él.

solución

```
CREATE USER funcionarios IDENTIFIED BY passfun
DEFAULT TABLESPACE users TEMPORARY TABLESPACE temp;
ALTER USER funcionarios QUOTA UNLIMITED ON users;
GRANT DBA TO funcionarios;
conn funcionarios/passfun
```

2. Crea la tabla *nominas* con un campo llamado *sueldo* tipo *NUMBER(8,2)* y otro llamado *nif* tipo *VARCHAR2(9)*.

solución

```
--La tabla se creará en el tablespace por defecto
CREATE TABLE nominas
  (nif VARCHAR2(9) NOT NULL,
   sueldo NUMBER(8,2) NOT NULL);
```

3. Crea una tabla donde se registren los errores en la actualización del sueldo. La tabla se llamará *erroresBajadaSueldo* y tendrá tres campos: *nif*, *codigoError* y *textoError*.

solución

```
--La tabla se creará en el tablespace por defecto
CREATE TABLE erroresBajadaSueldo
  (nif VARCHAR2(9) NOT NULL,
   codigoError NUMBER,
   textoError VARCHAR2(64));
```

4. Crea un paquete que se llame *paq-bajada-sueldo* que será el que contenga todas las funciones y procedimientos para llevar a cabo el problema. Define una

constante para el porcentaje de bajada de sueldo. El paquete deberá tener además:

- a) Un procedimiento *paq_bajada_sueldo.baja_sueldos* que abrirá un cursor para recorrer todos los sueldos de los funcionarios y proceder a su disminución. Además, el procedimiento será capaz de ir llenando la tabla *erroresBajadaSueldo* con los errores encontrados en la actualización del sueldo.
- b) Una función llamada *paq_bajada_sueldo.hubo_errores_bajando_sueldos* que evalúe la tabla de errores y determine si ha habido algún error o no. La función tendrá como parámetro de salida un booleano que indicará si hubo algún error o no.

solución

```

CREATE OR REPLACE PACKAGE paq_bajada_sueldo AS
  PROCEDURE baja_sueldos;
  FUNCTION hubo_errores_bajando_sueldos RETURN BOOLEAN;
  --El porcentaje de bajada se define como
  --una constante global al paquete
  porcentaje CONSTANT NUMBER := 5;
END paq_bajada_sueldo;
/
CREATE OR REPLACE PACKAGE BODY paq_bajada_sueldo AS
  PROCEDURE baja_sueldos IS
    codigoE      NUMBER;
    textoE       VARCHAR2(64);
  BEGIN
    FOR i IN (SELECT nif, sueldo
              FROM nominas) LOOP
      BEGIN
        UPDATE nominas SET sueldo = i.sueldo - i.sueldo * porcentaje / 100
        WHERE nif = i.nif;
        --Se pone un bloque de excepción aquí para evitar que
        --en caso de error en la sentencia de actualización el
        --programa se acabe, de este modo, el bucle continuaría
        --actualizando el resto de sueldos
      EXCEPTION
        WHEN OTHERS THEN
          codigoE := SQLCODE;
          textoE := SUBSTR(SQLERRM, 1, 64);
          INSERT INTO erroresBajadaSueldo (nif, codigoError, textoError)
                    VALUES (i.nif, codigoE, textoE);
      END;
      COMMIT;
    END LOOP;
  END baja_sueldos;

```

```
FUNCTION hubo_errores_bajando_sueldos RETURN BOOLEAN IS
    res BOOLEAN := FALSE;
    numErrores NUMBER;
BEGIN
    SELECT COUNT(*) into numErrores from erroresBajadaSueldo;
    IF numErrores > 0 THEN
        res := TRUE;
    END IF;
    RETURN res;
END hubo_errores_bajando_sueldos;

END paq_bajada_sueldo;
/
```

5. Escribe cómo se invocaría la operativa completa: borrado del contenido de la tabla de errores, llamada al procedimiento *baja_sueldos* y llamada a la función *hubo_errores_bajando_sueldos*.

solución

```
SET SERVEROUTPUT ON
--Se borra el contenido antiguo de la tabla de errores
DELETE FROM erroresBajadaSueldo;
--Se invoca el procedimiento de bajada de sueldos
CALL paq_bajada_sueldo.baja_sueldos();
--Se comprueba si hubo algún error
BEGIN
IF paq_bajada_sueldo.hubo_errores_bajando_sueldos THEN
    --Hubo errores. Se presentan por pantalla.
    FOR i IN (SELECT *
                FROM erroresBajadaSueldo) LOOP
        DBMS_OUTPUT.PUT_LINE('Error al actualizar el sueldo del ' ||
            'funcionario con NIF ' || i.nif || '. Código ' || i.codigoError ||
            ' y su texto: ' || i.textoError);
    END LOOP;
END IF;
END;
/
```

Práctica 6.2: Procedimientos almacenados en MySQL

Crea una tabla innodb en la BBDD *jardineria* llamada *ActualizacionLimiteCredito* con tres

campos: Fecha, CodigoCliente, Incremento. Crea un procedimiento almacenado para actualizar los límites de crédito de los clientes en un % del total pedido entre 2008 y 2010 registrando el incremento en la tabla creada. Al terminar, invoca al procedimiento para actualizar el límite de crédito en un 15 %.

solución

```
CREATE TABLE ActualizacionLimiteCredito(
    Fecha DATETIME, CodigoCliente INTEGER, Incremento NUMERIC(15,2)
) engine=innodb;

delimiter //
CREATE PROCEDURE IncrementaLimCredito (IN porcentaje INTEGER)
BEGIN
    DECLARE TotalPedidos,Credito,Incremento NUMERIC(15,2);
    DECLARE Cliente,Terminado INTEGER DEFAULT 0;
    #cursor para recorrer clientes
    DECLARE curClientes CURSOR FOR
        SELECT LimiteCredito,CodigoCliente FROM Clientes;
    #Al terminar de recorrerse, se activará la variable terminado
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET Terminado = 1;
    #Si ocurre alguna excepción se producirá un rollback
    DECLARE EXIT HANDLER FOR NOT FOUND rollback;
    DECLARE EXIT HANDLER FOR SQLEXCEPTION rollback;
    OPEN curClientes;
    START TRANSACTION;
    FETCH curClientes INTO Credito,Cliente; #primer cliente
    WHILE NOT Terminado DO
        SELECT SUM(Cantidad*PrecioUnidad) INTO TotalPedidos #Total de pedidos?
            FROM DetallePedidos
            NATURAL JOIN Pedidos WHERE YEAR(FechaPedido)
            BETWEEN 2008 AND 2010 AND Pedidos.CodigoCliente=Cliente;
        IF TotalPedidos IS NOT NULL THEN #Si hay pedidos
            SET Incremento=TotalPedidos*Porcentaje/100;
            UPDATE Clientes SET LimiteCredito=LimiteCredito+Incremento
                WHERE CodigoCliente=Cliente;
            INSERT INTO ActualizacionLimiteCredito VALUES(now(),Cliente,Incremento);
        END IF;
        FETCH curClientes INTO Credito,Cliente; #siguiente cliente
    END WHILE;
    COMMIT;
END;
//
delimiter ;
call IncrementaLimCredito(15); #Invocar al procedimiento
```



6.10. Prácticas Propuestas

Práctica 6.3: Bajada de sueldo v.2.0

Se va a efectuar una segunda versión del procedimiento de bajada de sueldo. Consiste en bajar el sueldo a todos los funcionarios, esta vez en base a un determinado porcentaje en función de su escala salarial. Extrapolar los objetos de la actividad anterior para contemplar esta nueva situación, para ello:

1. Añade a la tabla *funcionarios.nominas* un campo que represente la escala salarial. El campo se llamará *grupo* y será de tipo *CHAR(1)*.
2. Crea una función llamada *dame_porcentaje_segun_escala* dentro del paquete *paq_bajada_sueldo* que acepte como parámetro la escala salarial y devuelva el porcentaje de disminución a aplicar. La equivalencia entre uno y otro se hará mediante constantes globales al paquete: grupo A-10 %, grupo B-8 %, grupo C-5 %, grupo D-3 %.
3. Modifica la función *baja_sueldos* de la práctica anterior para que llame a la nueva *dame_porcentaje_segun_escala* para extraer el porcentaje.
4. Escribe cómo se invocaría la operativa completa: borrado del contenido de la tabla de errores, llamada al procedimiento *baja_sueldos* y llamada a la función *hubo_errores_bajando_sueldos*.

◊

Práctica 6.4: Bajada de sueldo v.3.0

Modifica la función *dame_porcentaje_segun_escala* para que en vez de extraer la información a través de constantes, la extraiga de una tabla creada a tal efecto. La tabla deberá tener dos campos, uno con la escala (A,B,...) y otro campo con el porcentaje a descontar.

Sustituye el último punto de la práctica anterior por un procedimiento que también esté almacenado en el paquete *funcionarios.paq_bajada_sueldo*. ◊

Práctica 6.5: Facturación Se desea codificar un programa en PL-SQL para poder lanzar el proceso de facturación de la BBDD *jardineria*. Para ello, se deberán crear tres tablas con los siguientes comandos DDL:

solución

```
CREATE TABLE Facturas (
    CódigoCliente integer,
    BaseImponible Numeric(15,2),
    IVA Numeric(15,2),
    Total Numeric(15,2),
    CódigoFactura INTEGER);
CREATE TABLE Comisiones(
    CódigoEmpleado integer,
    Comisión Numeric(15,2),
    CódigoFactura integer);
CREATE TABLE AsientoContable(
    CódigoCliente integer,
    DEBE Numeric(15,2),
    HABER Numeric(15,2));
```

A continuación programar, en un paquete llamado Facturación:

1. Una función llamada UltimaFactura que devuelva el último código de factura generado. Si no hay ningún registro en la tabla Facturas, devolverá un 1.
2. Procedimiento Facturar: Este procedimiento grabará un registro en la tabla de facturas por cada uno de los pedidos de la BBDD. Al mismo tiempo guardará en la tabla de comisiones una comisión del 5 % del total del pedido para el empleado que haya generado la factura.
3. Procedimiento GenerarContabilidad: Este procedimiento guardará en la tabla AsientoContable un registro por cada grupo de facturas, almacenando en el campo DEBE la suma de las facturas que han sido generadas.



6.11. Resumen

Los conceptos clave de este capítulo son los siguientes:

- Las aplicaciones de base de datos pueden ser de tres tipos distintos: codificada en el lado del servidor, modelo cliente/servidor y modelo en tres niveles.
- Las sentencias SQL se pueden ejecutar de forma estática o dinámica. Una sentencia estática debe precompilarse, vincularse y compilarse antes de ser ejecutada en la aplicación. Una sentencia dinámica es creada en el momento de la ejecución.
- El lenguaje PL/SQL fue desarrollado por Oracle para añadir más potencia de ejecución a las sentencias SQL, con él es posible definir variables, crear estructuras de control de flujo y toma de decisiones...
- Las partes de un programa PL/SQL son el bloque *DECLARE*, el bloque *BEGIN..END* y el subbloque *EXCEPTION*.
- Los distintos programas que pueden crearse con el uso del lenguaje PL/SQL son: procedimientos, funciones, bloques anónimos y triggers.
- Los procedimientos y funciones pueden estar contenidos dentro de otros objetos más grandes llamados paquetes, los cuales están compuestos por *la cabecera* y *el cuerpo*.
- SQL*Plus es la herramienta que proporciona Oracle para conectarse a la base de datos por línea de comandos y ejecutar todo tipo de comandos de base de datos, incluidos programas PL/SQL.
- En PL/SQL hay datos de tipo carácter, numérico, fecha, booleanos, y datos grandes entre otros. Además existen muchas funciones para gestionarlos, aunque aquí solo se ha tratado la *SUBSTR*. Existen también, operadores para hacer operaciones aritméticas, comparaciones y concatenaciones.
- Las estructuras de control sirven para controlar el flujo del programa, la toma de decisiones y programar las acciones en consecuencia. Aquí se han estudiado el *IF*, *CASE*, *LOOP*, *WHILE* y *FOR*. Los tres últimos son bucles, y son muy útiles para recorrer cursos.
- El control de excepciones es un bloque opcional dentro de la programación pero muy recomendable. Sirve para interceptar los errores de ejecución del programa y tomar las medidas oportunas en consecuencia, de otro modo el programa terminaría abrupta y descontroladamente.

6.12. Test de repaso

1. ¿Qué tipo de variable tendría que definirse para albergar el dato 3,2?

- a) CHAR(3)
- b) NUMBER(2,1)
- c) NUMBER(3)
- d) a) y b)

2. ¿Qué es el *ROWID*?

- a) Es una fila especial de la tabla
- b) Es una función ideal para generar filas automáticamente
- c) Es una columna interna que almacena la dirección física de cada fila
- d) Es un tipo de excepción muy utilizada

3. ¿Cómo se define una constante?

- a) nombre_de_la_constante CONSTANT tipo_de_dato := valor;
- b) nombre_de_la_constante tipo_de_dato := valor CONSTANT
- c) CONSTANT nombre_de_la_constante tipo_de_dato := valor;
- d) nombre_de_la_constante CONSTANT tipo_de_dato := valor

4. La expresión $5 = '5'$ es:

- a) FALSE
- b) TRUE

5. El extracto de definición
`dame_sueldo() RETURN NUMBER(6)`
IS corresponde a

- a) Un procedimiento
- b) Una función
- c) Un disparador
- d) Un paquete

6. Si la variable *suspens* es *TRUE*, qué valor tendrá la expresión *NOT suspens*

- a) NULL
- b) FALSE
- c) TRUE

7. Si la variable *gallo* es '*kikiriki*', qué valor resultará de la función *REPLACE(gallo, 'i', 'o')*:

- a) o
- b) oooooooo
- c) kokoroko
- d) kokorokí

8. ¿Qué función se usa para escribir mensajes por pantalla en los programas PL/SQL?

- a) PUT_LINE
- b) DBMS_OUTPUT.PUTLINE
- c) DBMSOUTPUT.PUT_LINE
- d) PUTLINE

9. ¿Qué variable de entorno del SQL*Plus hay que activar para que la función de la pregunta 8 escriba por la salida standard?

- a) La función escribe siempre
- b) SERVER_OUTPUT
- c) SERVEROUTPUT

Soluciones: 1.d, 2.c, 3.a, 4.b, 5.b, 6.b, 7.d, 8.a, 9.c

6.13. Comprueba tu aprendizaje

1. Define qué es un cursor, para qué sirve y distintas formas de recorrerlo.
2. Define los distintos bloques de que se compone un programa PL/SQL.
3. ¿Para qué se usan los paquetes?
4. Define qué es un disparador y los tipos distintos que hay. Pon un ejemplo.
5. Enuncia las diferencias entre el SQL dinámico y el estático.
6. ¿Qué son los datos tipo *LOB*? ¿Para qué sirven?
7. Enumera los tipos de datos tipo carácter.
8. Enumera los tipos y subtipos de datos tipo numérico.
9. Enumera los tipos de datos tipo fecha.
10. Ventajas del *CASE* frente al *IF..ELSIF* y al revés.
11. Ventajas del *LOOP* frente al *WHILE* y al revés.
12. ¿Qué error intercepta el evento *NO_DATA_FOUND*?
13. ¿Cuándo se producirá un *TOO_MANY_ROWS*?
14. Explica cómo se usan las funciones especiales *SQLCODE* y *SQ-LERRM*.
15. ¿Qué es una transacción?
16. ¿En qué consiste la validación transaccional implícita?
17. ¿Para qué se usa el atributo *%TYPE*? ¿y *%ROWTYPE*?
18. Pon ejemplos con todos los operadores de comparación.
19. Ventajas del uso del *ROWID*.
20. ¿Dónde se puede usar la cláusula *EXIT WHEN*?
21. ¿Para qué sirve la tabla de sistema *DUAL*?
22. ¿Cuál es el signo comodín en las comparaciones entre datos de tipo cadena?, ¿cómo se usa?
23. ¿Qué hace la instrucción *FETCH* en el contexto de los cursos?
24. El atributo *NOTFOUND*, ¿de dónde es, para qué es y cómo se usa?
25. Explica qué es la conversión implícita de tipos de datos.
26. ¿Qué significa que en una base de datos todos los objetos tienen que estar identificados únicamente?
27. Expon las diferencias entre procedimientos y funciones.
28. Explica los distintos tipos de aplicaciones de base de datos que existen.