



13.1 Introducción

En las unidades anteriores hemos utilizado el lenguaje PL/SQL para crear procedimientos que se almacenan de manera aislada en la base de datos y que deben ser invocados explícitamente para que se ejecuten.

En esta unidad aprenderemos a crear disparadores que se ejecutarán automáticamente al producirse determinados eventos, así como a empaquetar los procedimientos junto con variables, tipos, cursores y otros elementos del lenguaje. Definiremos nuestros propios tipos, utilizaremos SQL dinámico para crear y modificar la definición de tablas y usuarios, y trabajaremos con objetos cuyas características se desconocen en tiempo de compilación. También conoceremos las facilidades de la Programación Orientada a Objetos (POO) implementada en PL/SQL.

13.2 Triggers de base de datos

Los **triggers** o **disparadores de base de datos** son bloques PL/SQL almacenados que se ejecutan o disparan automáticamente cuando se producen ciertos eventos.

Hay tres tipos de disparadores de bases de datos:

- **Disparadores de tablas.** Asociados a una tabla. Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).
- **Disparaciones de sustitución.** Asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas).
- **Disparadores del sistema.** Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etcétera) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).

Las operaciones permitidas con el cursor son: **declarar, abrir, recoger información y cerrar el cursor.** No se le pueden asignar valores ni utilizarlo en expresiones.

Se suelen utilizar para:

- Implementar restricciones complejas de seguridad o integridad.
- Posibilitar la realización de operaciones de manipulación sobre vistas.
- Prevenir transacciones erróneas.
- Implementar reglas administrativas complejas.
- Generar automáticamente valores derivados.
- Auditir las actualizaciones e, incluso, enviar alertas.

En realidad, un **cursor** es una estructura que apunta a una región de la PGA que contiene toda la información relativa a la consulta asociada, en especial las filas de datos recuperadas.



13. Programación avanzada

13.2 Triggers de base de datos

- Gestionar réplicas remotas de la tabla.
- Etcétera.

A. Creación de un trigger

Aunque existen diferencias dependiendo del tipo de disparador, el formato básico para la creación de un *trigger* es:

```
CREATE [OR REPLACE]
/* aquí comienza la cabecera del trigger */
TRIGGER nombretrigger
    { BEFORE | AFTER | INSTEAD OF } evento_de_disparo
    [ WHEN condición_de_disparo ]
/* aquí comienza el cuerpo del trigger. Es un bloque
PL/SQL */
[DECLARE      ---- opcional
  <declaraciones>]
BEGIN
  <acciones>
[EXCEPTION      ---- opcional
  <gestión de excepciones>]
END];
```

Podemos distinguir:

- **Cabecera del trigger.** En ella se define:
 - Nombre del *trigger*: es el nombre o identificador del disparador.
 - Evento de disparo: es el suceso que producirá la ejecución del disparador. Puede ser:
 - Una orden DML (INSERT, DELETE o UPDATE) sobre una tabla o vista.
 - Una orden de definición de datos (CREATE, ALTER, etcétera).
 - O un suceso del sistema.
 - Restricción del *trigger*: condiciona la ejecución del *trigger* al cumplimiento de una condición. Es opcional.
- **Cuerpo del trigger.** Es el código que se ejecutará cuando se produzca el evento que cumplan las condiciones especificadas en la cabecera. Se trata de un bloque PL/SQL.

El cuerpo del *trigger* es un bloque PL/SQL con el formato anteriormente estudiado.

La sección declarativa, si se incluye, comenzará con la cláusula DECLARE.

A continuación, estudiaremos cada uno de los tres tipos de disparadores con sus formatos y opciones específicos.

13. Programación avanzada

13.2 Triggers de base de datos



B. Disparadores de tablas

Son disparadores asociados a una determinada tabla de la base de datos que se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).

El siguiente ejemplo crea el *trigger* audit_subida_salario, que se disparará después de cada modificación de la columna salario de la tabla EMPLE:

```
CREATE OR REPLACE TRIGGER audit_subida_salario
    AFTER UPDATE OF salario
    ON emple
    FOR EACH ROW
BEGIN
    INSERT INTO auditareemple
        VALUES ('SUBIDA SALARIO EMPLEADO '||:old.emp_no );
END;
/
Disparador creado.
```

El calificador *:old* permite acceder a las columnas de la fila que acaba de ser modificada o borrada.

Tanto *:old* como *:new* se estudiarán en detalle más adelante.

El disparador insertará una fila en la tabla auditareemple con el texto SUBIDA SALARIO EMPLEADO y el número del empleado al que se ha subido el salario.

El formato para la creación de disparadores de tablas es:

```
CREATE [OR REPLACE]
TRIGGER nombretrigger
{ BEFORE | AFTER }
{ DELETE | INSERT | UPDATE [OF <lista_columnas>] }
ON nombrertabla
[FOR EACH {STATEMENT | ROW [WHEN (condicion)]}]
< CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

Se pueden especificar varios eventos de disparo para un mismo *trigger* utilizando la cláusula OR.

```
... OR { BEFORE | AFTER }
{DELETE | INSERT | UPDATE
[OF listacolumnas]}
```

Cabe señalar que

- El **evento de disparo** será una orden de manipulación: INSERT, DELETE o UPDATE. En el caso de esta última, se podrán especificar opcionalmente las columnas cuya modificación producirá el disparo.
- El **momento en que se ejecuta el trigger** puede ser antes (BEFORE) o después (AFTER) de que se ejecute la orden de manipulación.
- El **nivel de disparo del trigger** puede ser a *nivel de orden* o a *nivel de fila*.
 - A **nivel de orden** (STATEMENT). El *trigger* se activará una sola vez para cada orden, independientemente del número de filas afectadas por ella. Se puede incluir la cláusula FOR EACH STATEMENT, aunque no es necesario, ya que se asume por omisión.

La combinación de:

- Tres posibles órdenes (INSERT, UPDATE, DELETE).
- Dos posibles momentos de disparo (BEFORE, AFTER).
- Dos niveles de disparo (ROW, STATEMENT).

Da 12 posibles tipos de disparadores:

BEFORE DELETE FOR EACH ROW, AFTER UPDATE, BEFORE INSERT FOR EACH ROW, ...



13. Programación avanzada

13.2 Triggers de base de datos

- A **nivel de fila**: el *trigger* se activará una vez para cada fila afectada por la orden. Para ello, se incluirá la cláusula FOR EACH ROW.
- La **restricción del trigger**. La cláusula WHEN seguida de una condición restringe la ejecución del *trigger* al cumplimiento de la condición especificada. Esta condición tiene algunas limitaciones:
 - Solamente se puede utilizar con *triggers* a nivel de fila (FOR EACH ROW).
 - Se trata de una condición SQL, no PL/SQL.
 - No puede incluir una consulta a la misma o a otras tablas o vistas.

El siguiente ejemplo crea un *trigger* que se disparará cada vez que se borre un empleado, guardando su número de empleado, apellido y departamento en una fila de la tabla auditaremple:

```
CREATE OR REPLACE TRIGGER audit_borrado_emple
BEFORE DELETE
ON emple
FOR EACH ROW
BEGIN
    insert INTO auditaremple
    VALUES ('BORRADO EMPLEADO' || '*' || :old.emp_no || '*'
           || :old.apellido || '*Dpto.' || :old.dept_no);
END;
/
Disparador creado.
```

Este disparador requiere de la tabla auditaremple, que habrá sido creada:

```
CREATE TABLE auditaremple (col1 VARCHAR2(200));
```

Si quisiéramos incluir una restricción para que sólo se ejecute el disparador cuando el empleado borrado sea el PRESIDENTE lo indicaremos insertando una cláusula WHEN antes del cuerpo del *trigger*:

```
WHEN old.oficio = 'PRESIDENTE'
```

Valores NEW y OLD

Se puede hacer referencia a los valores anterior y posterior a una actualización a nivel de fila. Lo haremos como *:old.nombrecolumna* y *:new.nombrecolumna* respectivamente.

Por ejemplo: ... IF :new.salario < :old.salario ...

Al utilizar los valores *old* y *new* deberemos tener en cuenta el evento de disparo:

- Cuando el evento que dispara el *trigger* es DELETE, deberemos hacer referencia a *:old.nombrecolumna*, ya que el valor de *new* es NULL.

13. Programación avanzada

13.2 Triggers de base de datos



- Paralelamente, cuando el evento de disparo es INSERT, deberemos referirnos siempre a `:new.nombrecolumna`, puesto que el valor de `old` no existe (es NULL).
- Para los triggers cuyo evento de disparo es UPDATE, tienen sentido los dos valores.

Conviene recordar que solamente se puede hacer referencia a los valores **new** y **old** en disparadores a nivel de fila.

En el caso de que queramos hacer referencia a los valores **new** y **old**, al indicar la restricción del trigger (en la cláusula WHEN), lo haremos sin poner los dos puntos. Por ejemplo:

```
WHEN new.salario < old.salario.
```

Si queremos que, en lugar de **old** y **new**, aparezcan otras palabras, podemos usar la cláusula REFERENCING justo antes de WHEN.

Por ejemplo, ...REFERENCING new AS nuevo, old AS anterior ...

Actividades propuestas



- 1 Escribe un disparador que inserte en la tabla `auditareemple` (col1 (VARCHAR2(200)) cualquier cambio que supere el 5% del salario del empleado indicando la fecha y hora, el empleado, y el salario anterior y posterior.

● Orden de ejecución de los disparadores

Una misma tabla puede tener varios disparadores. El orden de disparo será el siguiente:

- Antes de comenzar a ejecutar la orden que produce el disparo, se ejecutarán los disparadores BEFORE ... FOR EACH STATEMENT.
- Para cada fila afectada por la orden:
 - Se ejecutan los disparadores BEFORE ... FOR EACH ROW.
 - Se ejecuta la actualización de la fila (INSERT UPDATE o DELETE). En este momento se bloquea la fila hasta que la transacción se confirme.
 - Se ejecutan los disparadores AFTER ... FOR EACH ROW.
- Una vez realizada la actualización se ejecutarán los disparadores AFTER ... FOR EACH STATEMENT.



13. Programación avanzada

13.2 Triggers de base de datos

Observaciones:

- Cuando se dispara un *trigger*, éste forma parte de la operación de actualización que lo disparó, de manera que si el *trigger* falla, Oracle dará por fallida la actualización completa. Aunque el fallo se produzca a nivel de una sola fila, Oracle hará ROLLBACK de toda la actualización.
- En ocasiones, en lugar de asociar varios *triggers* a una tabla, podremos optar por la utilización de un solo *trigger* con múltiples eventos de disparo, tal como se explica en el apartado siguiente.

◆ Múltiples eventos de disparo y predicados condicionales

Un mismo *trigger* puede ser disparado por distintas operaciones o eventos de disparo. Para indicarlo, se utilizará el operador OR.

Por ejemplo, ... BEFORE DELETE OR UPDATE ON empleados ...

En estos casos, es probable que una parte de las acciones del bloque PL/SQL dependa del tipo de evento que disparó el *trigger*. Para facilitar este control Oracle permite la utilización de predicados condicionales que devolverán un valor verdadero o falso para cada una de las posibles operaciones:

INSERTING	Devuelve TRUE si el evento que disparó el <i>trigger</i> fue un comando INSERT.
DELETING	Devuelve TRUE si el evento que disparó el <i>trigger</i> fue un comando DELETE.
UPDATING	Devuelve TRUE si el evento que disparó el <i>trigger</i> fue una instrucción UPDATE.
UPDATING ('nombrecolumna')	Devuelve TRUE si el evento que disparó el <i>trigger</i> fue una instrucción UPDATE y la columna especificada ha sido actualizada.

Por ejemplo:

```
CREATE TRIGGER ...
BEFORE INSERT OR UPDATE OR DELETE ON empleados ...
BEGIN
    IF INSERTING THEN
        ...
    ELSIF DELETING THEN
        ...
    ELSIF UPDATING('salario') THEN
        ...
    END IF
    ...
END;
```

13. Programación avanzada

13.2 Triggers de base de datos



Continuación

◆ Restricciones

El código PL/SQL del cuerpo del *trigger* puede contener instrucciones de consulta y de manipulación de datos, así como llamadas a otros subprogramas. No obstante, existen restricciones que deben ser contempladas:

- El bloque PL/SQL no puede contener sentencias de control de transacciones como COMMIT, ROLLBACK o SAVEPOINT.
- Tampoco se pueden hacer llamadas a procedimientos que transgredan la restricción anterior.
- No se pueden utilizar comandos DDL.
- Un *trigger* no puede contener instrucciones que consulten o modifiquen *tablas mutantes*. Una **tabla mutante** es aquella que está siendo modificada por una sentencia UPDATE, DELETE o INSERT en la misma sesión.
- No se pueden cambiar valores de las columnas que sean claves primaria, única o ajena de *tablas de restricción*. Una **tabla de restricción** es una tabla que debe ser consultada o actualizada directa o indirectamente por el comando que disparó el *trigger* (normalmente, debido a una restricción de integridad referencial) en la misma sesión.

Los *triggers* a nivel de comando (FOR EACH STATEMENT) no se verán afectados por las restricciones que acabamos de enunciar para las tablas mutantes y tablas de restricción, excepto cuando el *trigger* se dispare como resultado de una restricción ON DELETE CASCADE.

C. Disparadores de sustitución

Son disparadores *asociados a vistas* que arrancan al ejecutarse una instrucción de actualización sobre la vista a la que están asociados. Se ejecutan en lugar de (INSTEAD OF) la orden de manipulación que produce el disparo del *trigger*; por eso se denominan disparadores de sustitución.

El formato genérico para la creación de estos disparadores de sustitución es:

```
CREATE [OR REPLACE] triggers
TRIGGER nombretrigger
    INSTEAD OF {DELETE | INSERT | UPDATE [OF <lista_columnas>]}
    ON nombrevista
    [FOR EACH ROW] [WHEN (condicion)]
<CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

Los disparadores de sustitución se ejecutan siempre a nivel de fila.

La cláusula FOR EACH ROW es opcional pero no hay otra posibilidad.



13. Programación avanzada

13.2 Triggers de base de datos

Los disparadores de sustitución tienen estas características diferenciales:

- Solamente se utilizan en *triggers* asociados a vistas, y son especialmente útiles para realizar operaciones de actualización complejas.
- Actúan siempre a nivel de fila, no a nivel de orden, por tanto, a diferencia de lo que ocurre en los disparadores asociados a una tabla, la opción por omisión es FOR EACH ROW.
- No se puede especificar una restricción de disparo mediante la cláusula WHEN (pero no se puede conseguir una funcionalidad similar utilizando estructuras alternativas dentro del bloque PL/SQL).

Caso práctico

1 Supongamos que disponemos de la siguiente vista:

```
CREATE VIEW EMPLEAD AS
SELECT EMP_NO, APELLIDO, OFICIO, DNOMBRE, LOC
FROM EMPLE, DEPART
WHERE EMPLE.DEPT_NO = DEPART.DEPT_NO;
```

Los usuarios verán los datos:

```
SQL> select * from emplead;
EMP_NO    APELLIDO   OFICIO      DNOMBRE      LOC
-----  -----
7839      REY        PRESIDENTE  CONTABILIDAD  SEVILLA
7876      ALONSO     EMPLEADO    INVESTIGACIÓN MADRID
7521      SALA       VENDEDOR    VENTAS      BARCELONA
...
...
```

Las siguientes operaciones de manipulación sobre los datos de la vista darán como resultado:

```
SQL> INSERT INTO EMPLEAD VALUES (7999, 'MARTINEZ', 'VENDEDOR', 'CONTABILIDAD', 'SEVILLA');
ERROR en línea 1: ORA-01776: no se puede modificar más de una tabla base a través de una vista.
```

```
SQL> UPDATE EMPLEAD SET DNOMBRE = 'CONTABILIDAD' WHERE APELLIDO = 'SALA';
ERROR en línea 1:ORA-01779: no se puede modificar una columna que se corresponde con una tabla reservada por clave
```

(Continúa)

13. Programación avanzada

13.2 Triggers de base de datos



(Continuación)

Para facilitar estas operaciones de manipulación crearemos el siguiente disparador de sustitución:

```
CREATE OR REPLACE TRIGGER t_ges_emplead
INSTEAD OF DELETE OR INSERT OR UPDATE
ON emplead
FOR EACH ROW
DECLARE
    v_dept depart.dept_no%TYPE;
BEGIN
    IF DELETING THEN      /* Si se pretende borrar una fila */
        DELETE FROM EMPLE WHERE emp_no = :old.emp_no;
    ELSIF INSERTING THEN   /* Si se intenta insertar una fila */
        SELECT dept_no INTO v_dept  FROM depart
        WHERE depart.dnombre = :new.dnombre
        AND loc = :new.loc;
        INSERT INTO EMPLE (emp_no, apellido, oficio, dept_no)
        VALUES (:new.emp_no, :new.apellido, :new.oficio, v_dept);
    ELSIF UPDATING('dnombre') THEN    /* Si se trata de actualizar
                                       la columna dnombre*/
        SELECT dept_no INTO v_dept FROM depart
        WHERE dnombre = :new.dnombre;
        UPDATE emple SET dept_no = v_dept
        WHERE emp_no = :old.emp_no;
    ELSIF UPDATING('oficio') THEN    /* Si se pretende actualizar
                                       la columna oficio */
        UPDATE emple SET oficio = :new.oficio
        WHERE emp_no = :old.emp_no;
    ELSE
        RAISE_APPLICATION_ERROR(-20500,'Error en la actualización');
    END IF;
END;
/
```

Ahora podemos realizar las operaciones anteriormente indicadas. Se puede cambiar a un empleado de departamento indicando el nombre del departamento nuevo. El disparador se encargará de comprobar y asignar el número de departamento que corresponda. También se han limitado las columnas que hay que actualizar. En caso de que la operación que pretende el usuario no se contempla entre las alternativas, el trigger levantará un error en la aplicación haciendo que falle toda la actualización.

D. Disparadores del sistema

Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etcétera) o una instrucción de definición de datos (creación, modificación o eliminación de un objeto).

Para crear triggers del sistema hay que tener el privilegio ADMINISTER DATABASE TRIGGER.



13. Programación avanzada

13.2 Triggers de base de datos

La sintaxis para su creación es:

```
CREATE [OR REPLACE]
TRIGGER nombretrigger
[BEFORE | AFTER] {<lista eventos definición> | <lista
eventos del sistema>}
ON {DATABASE | SCHEMA} [WHEN (condición)]
<CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

Los disparadores STARTUP y SHUTDOWN sólo tienen sentido a nivel ON DATABASE, no asociarlos a un esquema pues, aunque Oracle permite la asociación, nunca se dispararán.

Donde:

- El nombre del *trigger* puede incluir el esquema mediante la notación de punto.
- <lista eventos definición> puede incluir uno o más eventos DDL separados por OR.
- <lista eventos del sistema> puede incluir uno o más eventos del sistema separados por OR.
- El especificador ON SCHEMA|DATABASE indica el *nivel de disparo del trigger*:
 - **ON DATABASE** se disparará siempre que ocurra el evento de disparo.
 - **ON SCHEMA** se disparará solamente si el evento ocurre en el esquema determinado por el *trigger*. Por defecto, este esquema es aquel al que pertenece el *trigger*, pero se puede indicar otro mediante la notación de punto: ON esquema.SCHEMA

Al asociar un disparador a un evento del sistema debemos indicar el momento del disparo. Algunos eventos sólo pueden disparar ANTES de producirse, otros DESPUES y otros admiten las dos posibilidades, tal como se muestra en la tabla siguiente:

Evento	Momento	Se disparan:
STARTUP	AFTER	Después de arrancar la instancia.
SHUTDOWN	BEFORE	Antes de apagar la instancia.
LOGON	AFTER	Después de que el usuario se conecte a la base de datos.
LOGOFF	BEFORE	Antes de la desconexión de un usuario.
SERVERERROR	AFTER	Cuando ocurre un error en el servidor.
CREATE	BEFORE AFTER	Antes o después de crear un objeto en el esquema.
DROP	BEFORE AFTER	Antes o después de borrar un objeto en el esquema.
ALTER	BEFORE AFTER	Antes o después de cambiar un objeto en el esquema.
TRUNCATE	BEFORE AFTER	Antes o después de ejecutar un comando TRUNCATE.
GRANT	BEFORE AFTER	Antes o después de ejecutar un comando GRANT.
REVOKE	BEFORE AFTER	Antes o después de ejecutar un comando REVOKE.
DLL	BEFORE AFTER	Antes o después de ejecutar cualquier comando de definición de datos (excepto algunos, como CREATE DATABASE).
Otros comandos del sistema	BEFORE AFTER	RENAME, ANALYZE, AUDIT, NO AUDIT, COMMENT, SUSPEND, ASSOCIATE STATISTICS, DISASSOCIATE STATISTICS.

Tabla 13.1. Eventos del sistema y eventos de definición.

13. Programación avanzada

13.2 Triggers de base de datos



PL/SQL dispone de algunas funciones que permiten acceder a los atributos del evento del disparo ORA_SYSEVENT, ORA_LOGIN_USER, ORA_DICT_OBJ_NAME, ORA_DICT_OBJ_TYPE, etcétera. En los manuales del producto podemos encontrar un listado completo de las funciones accesibles desde estos disparadores. Estas funciones pueden usarse en la cláusula WHEN o en el cuerpo del disparador. Cabe mencionar que:

Los STARTUP y SHUTDOWN no pueden llevar la cláusula WHEN pues no permiten condiciones.

- Desde un disparador LOGON o LOGOFF se puede comprobar el identificador de usuario, o el nombre de usuario (ID, USERID, USERNAME).
- Desde un disparador DDL puede comprobar el tipo y el nombre del objeto que se está modificando (y el ID o nombre de usuario).

Caso práctico

2 Escribiremos un disparador que controlará las conexiones de los usuarios en la base de datos.

Para ello introducirá en la tabla control_conexiones el nombre de usuario (USER), la fecha y hora en la que se produce el evento de conexión, y la operación CONEXIÓN que realiza el usuario.

```
CREATE OR REPLACE TRIGGER ctrl_conexiones
AFTER LOGON
ON DATABASE
BEGIN
    INSERT INTO control_conexiones (usuario, momento, evento)
    VALUES (ORA_LOGIN_USER, SYSTIMESTAMP, ORA_SYSEVENT);
END;
```

Para que el disparador pueda crearse deberá estar creada la tabla control_conexiones:

```
CREATE TABLE control_conexiones (usuario VARCHAR2(20),
momento TIMESTAMP, evento VARCHAR2(20));
```

Para crear este disparador a nivel ON DATABASE hay que tener el privilegio ADMINISTER DATABASE TRIGGER, de lo contrario sólo nos permitirá crearlo ON SCHEMA.

Una vez creado el disparador cualquier evento de conexión en el esquema producirá el disparo del trigger y la consiguiente inserción de la fila en la tabla.

USUARIO	MOMENTO	EVENTO
FERNANDO	29/03/06 10:54:51,145000	LOGON

Por otro parte, crearemos un trigger que inserte en la tabla control_eventos cualquier instrucción de definición de datos:

```
CREATE TABLE control_eventos (usuario VARCHAR2(20), momento TIMESTAMP, evento
VARCHAR2(40));
```

(Continúa)



13. Programación avanzada

13.2 Triggers de base de datos

(Continuación)

```
CREATE OR REPLACE TRIGGER ctrl_eventos
AFTER DDL
ON DATABASE
BEGIN
    INSERT INTO control_eventos (usuario, momento, evento)
    VALUES (USER, SYSTIMESTAMP, ORA_SYSEVENT || '*' ||ORA_DICT_OBJ_NAME);
END;
```

Una vez que probemos el disparador deberemos borrarlo con DROP.

E. Consideraciones y opciones de utilización

- Para crear un *trigger* hay que tener privilegios de CREATE TRIGGER, así como los correspondientes privilegios sobre la tabla o tablas y otros objetos referenciados por el *trigger*.
- Con el nombre del *trigger* se puede especificar el esquema en el que queremos crearlo, utilizando la notación de punto. Por omisión, Oracle asumirá nuestro esquema actual en el momento de crear el *trigger*. El privilegio CREATE ANY TRIGGER permite crear *trigger* en cualquier esquema.
- La tabla, vista u objeto al que se asocia el disparador puede pertenecer a un esquema distinto del actual, siempre que se tengan los privilegios correspondientes; en este caso, se utilizará la notación de punto. No se puede asociar un *trigger* a una tabla del esquema SYS.
- Como ya hemos señalado, un *trigger* forma parte de la operación de actualización que lo disparó y si éste falla, Oracle dará por fallida la actualización. Esta característica puede servir para impedir desde el *trigger* que se realice una determinada operación, ya que si levantamos una excepción y no la tratamos, el *trigger* y la operación fallarán.
- Los disparadores son una herramienta muy útil, pero su uso indiscriminado puede degradar el comportamiento de la base de datos y ser fuente de problemas. Por ello, cuando se trata de implementar restricciones de integridad, se deberán utilizar preferentemente las restricciones ya disponibles: PRIMARY KEY, FOREIGN KEY, CHECK, NOT NULL, UNIQUE, CASCADE, SET NULL, SET DEFAULT, etcétera.

F. Activar, desactivar, compilar y eliminar disparadores

Un *trigger* puede estar activado o desactivado. Cuando se crea está activado, pero podemos variar esta situación mediante: ALTER TRIGGER *nombretrigger* DISABLE.

Para volver a activarlo utilizaremos: ALTER TRIGGER *nombretrigger* ENABLE.

Para volver a compilar emplearemos: ALTER TRIGGER *nombretrigger* COMPILE.

Para eliminar un *trigger* escribiremos: DROP TRIGGER *nombretrigger*.

13. Programación avanzada

13.2 Triggers de base de datos



G. Vistas con información sobre los triggers

Las vistas `dba_triggers` y `user_triggers` contienen toda la información sobre los *trigger*. La estructura de estas vistas se muestran a continuación con algunos ejemplos.

SQL> describe dba_triggers

Name	Null?	Type
OWNER		VARCHAR2 (30)
TRIGGER_NAME		VARCHAR2 (30)
TRIGGER_TYPE		VARCHAR2 (16)
TRIGGERING_EVENT		VARCHAR2 (227)
TABLE_OWNER		VARCHAR2 (30)
BASE_OBJECT_TYPE		VARCHAR2 (16)
TABLE_NAME		VARCHAR2 (30)
COLUMN_NAME		VARCHAR2 (4000)
REFERENCING_NAMES		VARCHAR2 (128)
WHEN_CLAUSE		VARCHAR2 (4000)
STATUS		VARCHAR2 (8)
DESCRIPTION		VARCHAR2 (4000)
ACTION_TYPE		VARCHAR2 (11)
TRIGGER_BODY		LONG

SQL> SELECT TRIGGER_NAME, TABLE_NAME, TRIGGERING_EVENT FROM USER_TRIGGERS;

TRIGGER_NAME	TABLE_NAME	TRIGGERING_EVENT
AUDIT_SUBIDA_SALARIO	EMPLEADOS	UPDATE
T_GES_EMPLEAD	EMPLEAD	INSERT OR UPD...

TRIGGER_TYPE	STATUS
AFTER EACH ROW	ENABLED
INSTEAD OF	ENABLED

SQL> SELECT TRIGGER_BODY FROM USER_TRIGGERS;

TRIGGER_BODY

```
BEGIN
  insert INTO auditaremple VALUES ('SUBIDA SALARIO EMPLEADO
  ' || :old.e...
  ...
  DECLAR
  v_dept depart.dept_no%TYPE;
  BEGIN
    IF DELETING THEN
      ...
    END;
  END;
END;
```



13. Programación avanzada

13.3 Registros y colecciones

(Continuación)

13.3 Registros y colecciones

Los **registros** y las **colecciones** son estructuras de datos compuestas de otras más simples.

A. Registros en PL/SQL

El concepto de **registro** en PL/SQL es similar al de la mayoría de los lenguajes de programación (en C se llaman *estructuras*). Se trata de una estructura compuesta de otras más simples (*campos*) que pueden ser de distintos tipos.

A lo largo de este libro hemos venido utilizando el atributo ROWTYPE para crear una estructura de datos idéntica a la fila de una tabla, vista o cursor. Esta estructura de datos es un **registro**. Esta forma de crear registros es muy sencilla y rápida pero tiene algunas limitaciones:

- Tanto los nombres de los campos como sus tipos quedarán determinados por los nombres y los tipos de las columnas de la tabla, sin que exista posibilidad de variar alguno de ellos ni de excluir algunos campos o incluir otros.
- No se incluyen restricciones como NOT NULL ni valores por defecto.
- La posibilidad de creación de variables de registro utilizando este formato quedará condicionada a la existencia de la tabla de referencia y a sus posibles variaciones.

PL/SQL permite salvar estas limitaciones definiendo nosotros mismos el registro, y declarando después todas las variables de ese tipo que necesitemos. Para ello procederemos de este modo:

1. Se define el tipo genérico del registro:

```
TYPE nombre_tipo IS RECORD  
  (nombre_campo1 Tipo_campo1 [ NOT NULL ] { := | DEFAULT }  
   valorinicial1],  
   nombre_campo2 Tipo_campo2 [ NOT NULL ] { := | DEFAULT }  
   valorinicial2],  
   ...);
```

Al definir el tipo podemos usar el especificador NOT NULL. En ese caso los campos deben ser inicializados:

```
TYPE nombre_tipo IS RECORD  
  (nombre_campo1 Tipo_campo1 NOT NULL := valor1,  
   nombre_campo2 Tipo_campo2 := valor2,  
   ...);
```

2. Se declaran las variables que se necesiten de ese tipo según el formato:

```
nombre_variable  nombre_tipo;
```

Los tipos de los campos se pueden especificar también utilizando %TYPE y %ROWTYPE.

13. Programación avanzada

13.3 Registros y colecciones



El siguiente ejemplo define el tipo **t_domicilio** y posteriormente declara la variable **v_domici** de ese tipo:

```
TYPE t_domicilio IS RECORD
  calle VARCHAR2(30),
  numero SMALLINT,
  localidad VARCHAR2(25) );
v_domici t_domicilio;
```

Para referirnos a la variable de registro completa, indicaremos su nombre. Para referenciar solamente un campo, lo haremos utilizando la notación de punto según el formato:

```
nombre_variable_registro.nombre_campo
```

Podemos usar esta notación para asignar valores como si se tratase de cualquier otra variable. Pero debemos tener cuidado con expresiones del tipo **Nom_var_reg1 := Nom_var_reg2**, pues solamente funcionarán cuando ambas variables hayan sido definidas sobre el mismo tipo-base. En caso contrario, aun cuando coincidan los tipos, longitudes e, incluso, nombres de los campos, dará error.

Un campo de un registro puede ser, a su vez, un registro. En este caso se les llama **registros anidados**. En el siguiente ejemplo, al declarar el campo **domicilio** se indica que es de tipo **t_domicilio**, definido previamente y que incluye los campos **calle**, **numero** y **localidad**. Posteriormente, se podrá hacer referencia a estos campos utilizando la notación de punto, tal como aparece en el ejemplo:

```
DECLARE
  TYPE t_domicilio IS RECORD
    calle VARCHAR2(30),
    numero SMALLINT,
    localidad VARCHAR2(25) );
  TYPE t_datospersona IS RECORD
    (nombre VARCHAR2(35),
     domicilio t_domicilio,
     fecha_nacimiento DATE);
  v_persona t_datospersona;
BEGIN
  v_persona.nombre := 'ALONSO FERNÁNDEZ, JOAQUÍN';
  v_persona.domicilio.calle := 'C/ ALBUFERA';
  v_persona.domicilio.numero := 14;
END;
```

Los registros se pueden usar como parámetros y como valor de retorno de procedimientos y funciones. En estos casos, es conveniente definir el tipo base del registro externamente, de manera que esté accesible para todos los programas que vayan a usarlo.

Por ejemplo, en una declaración pública de un paquete o como un objeto de la base de datos. Ambos casos los estudiaremos más adelante en esta misma unidad.

Podemos definir tipos de registro, como objetos de la base de datos, aplicando el formato de definición de los registros junto con el de los objetos tal como se explica al final de esta unidad.



13. Programación avanzada

13.3 Registros y colecciones

B. Colecciones PL/SQL

Las **colecciones** son estructuras compuestas por listas de elementos. Se usan para guardar datos en formato de múltiples filas similar a las tablas de la base de datos.

Oracle dispone de tres tipos de colecciones:

- *Varrays*.
- Tablas anidadas.
- Tablas indexadas o *arrays asociativos*.

Todas ellas son listas de una dimensión aunque sus elementos pueden ser compuestos.

◆ Varrays

Son equivalentes a los *arrays* (tablas de una dimensión) de los lenguajes de programación tradicionales.

- Pueden usarse en tablas de la base de datos.
- Se puede acceder a ellas desde SQL y desde PL/SQL.
- Tienen un índice secuencial que permite el acceso a sus elementos. A diferencia de otros lenguajes de programación (Java, C, etcétera), el índice comienza en uno.
- Tienen una longitud fija determinada en el momento de su creación.

Para poder usar el tipo VARRAY debemos:

1. **Definir el tipo.** Podemos optar por definirlo a nivel de programa en la sección declarativa según el formato:

```
TYPE nombre_tipovarray IS VARRAY (numelementos) OF  
tipoelementos [NOT NULL];
```

```
TYPE nombre_tipovarray IS VARRAY (numelementos) OF tipoelementos  
[NOT NULL];
```

- *nombre_tipovarray* es un especificador que indica el tipo base de la tabla y que posteriormente se utilizará para definir tablas.
- *tipoelementos* especifica el tipo de los elementos que va a contener. Pueden ser tipos predefinidos de la base de datos o tipos definidos por el usuario.

También podemos definirlo como un **objeto de la base de datos** (especialmente si queremos que esté disponible para otros programas) usando el formato:

13. Programación avanzada

13.3 Registros y colecciones



```
CREATE OR REPLACE
TYPE nombre_tipovarray AS VARRAY(numelementos) OF
    tipoelementos [NOT NULL];
```

2. Declarar variables de ese tipo en la sección declarativa del programa:

```
nombre_variable nombre_tipovarray;
```

3. Inicializamos la variable cargando valores. Esto podemos hacerlo:

- En la misma declaración de la variable según el formato:

```
nombre_variable nombre_tipovarray :=
    nombre_tipovarray(lista de valores);
```

- En la zona ejecutable haciendo asignaciones individuales:

```
nombre_variable := nombre_tipovarray (lista de valores);
```

Al asignar una lista a una variable tipo VARRAY indicaremos el tipo sobre el que está definida la lista:

```
tipovarray(lista);
```

4. Para hacer referencia a los elementos en el programa mediante el nombre de la variable y, entre paréntesis, el número del elemento.

```
nombre_variable(numelemento)
```

Los elementos deben ser inicializados antes de ser referenciados (aunque sea con valores nulos), pues de lo contrario nos encontraremos con el error:

ORA-06531: Reference to uninitialized collection.

Por ejemplo, podemos definir el tipo *t_meses* y declarar una variable de ese tipo incluyendo los meses:

```
DECLARE
    TYPE t_meses IS VARRAY (12) OF VARCHAR2(10);
    va_meses t_meses;
BEGIN
    va_meses := t_meses('ENERO', 'FEBRERO', 'MARZO', 'ABRIL',
    'MAYO', 'JUNIO', 'JULIO', 'AGOSTO', 'SEPTIEMBRE',
    'OCTUBRE', 'NOVIEMBRE', 'DICIEMBRE');
    FOR i IN 1..12 LOOP
        DBMS_OUTPUT.PUT_LINE( TO_CHAR(i) || '-' || va_meses(i));
    END LOOP;
END;
```

El tipo sobre el que se define un VARRAY (u otra colección) puede ser a su vez un tipo definido por el usuario, frecuentemente un registro. En este caso, para referirnos a un campo elemento *i* de la variable lo haremos:

```
nombrevariable(i).nombrecampo
```



13. Programación avanzada

13.3 Registros y colecciones



Caso práctico

3 Escribiremos un bloque PL/SQL que realizará lo siguiente:

- Declarar un cursor basado en una consulta.
- Definir un tipo de registro compatible con el cursor.
- Definir un tipo de VARRAY cuyos elementos son del tipo registro previamente definido.
- Declarar inicializar y usar una variable de tipo VARRAY cargando el contenido del cursor en los elementos y posteriormente mostrando el contenido de estos.

```
DECLARE
    /* Declaramos un cursor basado en una consulta */
    CURSOR c_depar IS
        SELECT dnombre, count(emp_no) numemple
        FROM depart, emple
        WHERE depart.dept_no = emple.dept_no
        GROUP BY depart.dept_no, dnombre;

    /* Definimos un tipo compatible con el cursor */
    TYPE tr_dept IS RECORD (
        nombredep depart.dnombre%TYPE,
        numemple INTEGER
    );

    /* Definimos un tipo VARRAY basado en el tipo anterior */
    TYPE tv_dept IS VARRAY (6) OF tr_dept;

    /* Declaramos e inicializamos una variable del tipo VARRAY definido arriba */
    va_departamentos tv_dept := tv_dept(NULL,NULL,NULL,NULL,NULL,NULL);

    /* Declaramos una variable para usarla como índice */
    n INTEGER := 0;
BEGIN
    /* Cargar valores en la variable */
    FOR vc IN c_depar LOOP
        n := c_depar%ROWCOUNT;
        va_departamentos(n) := vc;
    END LOOP;

    /* Mostrar los datos de la variable */
    FOR i IN 1..n LOOP
        DBMS_OUTPUT.PUT_LINE(' * Dnombre:' || va_departamentos(i).nombredep ||
                            ' * N°Empleados:' || va_departamentos(i).numemple);
    END LOOP;
END;
```

13. Programación avanzada

13.3 Registros y colecciones



◆ Tablas anidadas PL/SQL

Son estructuras similares a los VARRAYS estudiados en el epígrafe anterior y comparten con ellos muchas características estructurales y funcionales (lista de elementos, índice para acceso, acceso desde SQL y PL/SQL, posibilidad de uso en tablas de la base de datos, etcétera).

Pero también existen importantes diferencias, por ejemplo, las tablas anidadas no tienen una longitud fija. Para crearlas:

El tipo de dato TABLE en PL/SQL sirve para definir estructuras de datos de tipo array. Es completamente distinto del tipo TABLE que utilizan los SGBDR para almacenar la información.

1. **Definimos el tipo**, a nivel de programa, en la sección declarativa según el formato:

```
TYPE nombre_tipoTablaAnidada IS TABLE OF tipoelementos  
[NOT NULL];
```

También podemos definirlo como un objeto de la base de datos usando el formato:

```
CREATE OR REPLACE  
TYPE nombre_tipoTablaAnidada AS TABLE OF tipoelementos  
[NOT NULL];
```

2. **Declaramos e inicializamos las variables**, igual que en el caso de los VARRAY, pero en este caso las tablas anidadas permiten inicializar la variable con una lista incompleta o vacía:

```
nombre_variable := nombre_tipovarray();
```

Y añadir filas nuevas a la variable ya inicializada usando el método EXTEND según el formato:

```
nombre_variable_de_tabla_anidada.EXTEND;
```

El siguiente ejemplo ilustra las similitudes y diferencias de las tablas anidadas con los VARRAY y el uso del método EXTEND.

```
DECLARE  
    TYPE t_meses IS TABLE OF VARCHAR2(10);  
    va_meses t_meses;  
BEGIN  
    va_meses := t_meses('ENERO', 'FEBRERO', 'MARZO', 'ABRIL',  
                        'MAYO', 'JUNIO', 'JULIO', 'AGOSTO', 'SEPTIEMBRE',  
                        'OCTUBRE');  
    va_meses.EXTEND;  
    va_meses(11) := 'NOVIEMBRE';  
    va_meses.EXTEND;  
    va_meses(12) := 'DICIEMBRE';  
    FOR i IN 1..12 LOOP  
        DBMS_OUTPUT.PUT_LINE( TO_CHAR(i) || '-' || va_meses(i));  
    END LOOP;  
END;
```



13. Programación avanzada

13.3 Registros y colecciones



Actividades propuestas

- 3 Reescribe el bloque PL/SQL del caso práctico del epígrafe anterior usando una tabla anidada en lugar de un VARRAY. Debemos tener presente que no es necesario inicializar a NULL varios elementos, sino inicializar con una lista vacía y, después, podemos crear nuevos elementos en el bucle de carga usando el método EXTEND.

Las tablas anidadas y los VARRAYS permiten su uso en tablas de la base de datos y su manejo mediante instrucciones SQL. Ambos extremos exceden nuestros objetivos, pero remitimos a la documentación del producto para ampliaciones sobre esta materia.

El siguiente tipo de colección que estudiaremos sólo permite su uso desde PL/SQL.

◆ Tablas indexadas (o arrays asociativos)

Las tablas indexadas han evolucionado de la estructura conocida como TABLAS-PL/SQL de las versiones de Oracle 7 y 8.

Son estructuras tipo *array*, similares a las anteriores pero con diferencias importantes:

- No pueden usarse en tablas de la base de datos.
- No pueden manipularse con comandos SQL, sólo con PL/SQL.
- No son objetos.
- No tienen una longitud predeterminada.
- No requieren inicialización.
- Todos los elementos se crean dinámicamente.
- El índice no es secuencial, suele ser de tipo BINARY_INTEGER o PLS_INTEGER y puede tomar cualquier valor de los permitidos por estos tipos (positivo, negativo o cero).

Son tablas exclusivas de PL/SQL cuyos elementos se crean dinámicamente y cuyo índice no es secuencial. Tienen funcionalidades similares a las listas enlazadas.

Las operaciones a realizar para usar estas tablas son:

1. **Definir el tipo base de la tabla.** Igual que en los anteriores pero en este caso no indicaremos número de elementos y sí el tipo de índice.

```
TYPE nombre_tipoTabla IS TABLE OF tipo_elementos  
                           [NOT NULL]  
                           INDEX BY [PLS_INTEGER | BINARY_INTEGER | VARCHAR2(longitud)];
```

13. Programación avanzada

13.3 Registros y colecciones



También pueden crearse como un tipo de la base de datos mediante el comando CREATE OR REPLACE TYPE ...

2. Declarar variables de ese tipo utilizando el formato:

nombrevariableTabla nombre_tipotabla;

En el caso de las tablas indexadas, las variables no requieren inicialización y tampoco hay que reservar memoria para los nuevos elementos, simplemente introduciremos un valor indicando el índice del elemento.

3. Para hacer referencia a los elementos en el programa mediante el nombre de la variable y, entre paréntesis, el número del elemento: *nombre_variable(indice)*.

```
DECLARE
  TYPE T_tabla_emple IS TABLE OF emple%ROWTYPE;
    INDEX BY BINARY_INTEGER;
  tab_emple T_tabla_emple;
  ...
BEGIN
  ...
  SELECT * INTO tab_emple(7900)
    WHERE emp_no = 7900;
  ...
  DBMS_OUTPUT.PUT_LINE(tab_emple(7900).apellido);
  ...
  tab_emple(7900).salario := 3500;
  ...
END;
```

Los arrays indexados son muy útiles para cargar datos de una tabla de la base de datos usando como índice la clave primaria.

```
DECLARE
  TYPE T_tabla_emple IS TABLE OF emple%ROWTYPE;
    INDEX BY BINARY_INTEGER;
  tab_emple T_tabla_emple;
  CURSOR c_emple IS SELECT * FROM emple;
  ...
BEGIN
  /* El siguiente bucle carga en la tabla las filas de emple */
  FOR v_reg_emple IN c_emple LOOP
    tab_emple(v_reg_emple.emp_no) := v_reg_emple;
  END LOOP;
  ...
END;
```

La flexibilidad del índice no secuencial plantea también un problema a la hora de recorrer la tabla. Oracle dispone de una API para el manejo de colecciones que facilita esta tarea.

Debemos recordar que los elementos de una tabla indexada no tienen que ser necesariamente contiguos.

Cuando los elementos de la tabla son de tipo registro utilizaremos la notación de punto según el siguiente formato:

Nombretabla (indiceelemento).nombrecampo.



13. Programación avanzada

13.3 Registros y colecciones

◆ Atributos de colecciones PL/SQL

Facilitan la gestión de las variables de colecciones permitiendo recorrer la tabla, contar y borrar los elementos, etcétera. En general, para utilizar los atributos se empleará el siguiente formato:

variabletabla.atributo[(listadeparámetros)]

Los parámetros hacen referencia, normalmente, a valores de índice:

- FIRST. Devuelve el valor de la clave o índice del primer elemento de la tabla:

variabletabla.FIRST

- LAST. Devuelve el valor de la clave o índice del último elemento de la tabla:

variabletabla.LAST

Por ejemplo, para recorrer una tabla cuyo índice sabemos que tiene valores consecutivos podemos escribir:

FOR i IN variabletabla.FIRST .. variabletabla.LAST LOOP

- PRIOR. Devuelve el valor de la clave o índice del elemento anterior al elemento *n*:

variabletabla.PRIOR(n)

- NEXT. Devuelve el valor de la clave o índice del elemento posterior al elemento *n*:

variabletabla.NEXT(n)

PRIOR y NEXT se pueden utilizar para recorrer una tabla (en cualquiera de los dos sentidos) incluso con valores de índice no consecutivos. No obstante, deberemos tener cuidado con los valores que devolverán en ambos extremos, ya que PRIOR del primer elemento devuelve NULL y lo mismo ocurre con NEXT del último elemento.

```
...
i := variabletabla.FIRST
WHILE i IS NOT NULL LOOP
```

```
...
i := variabletabla.NEXT(i);
END LOOP;
```

```
...
```

- COUNT. Devuelve el número de filas que tiene una tabla:

variabletabla.COUNT

13. Programación avanzada

13.3 Registros y colecciones



- EXISTS. Devuelve TRUE si existe el elemento *n*; en caso contrario devolverá FALSE. Se utiliza para evitar el error que se produce cuando intentamos acceder a un elemento que no existe en la tabla:

variabletabla.EXISTS(n)

- DELETE. Se utiliza para borrar elementos de una tabla:

- **variabletabla.DELETE**. Borra todos los elementos de la tabla.

- **variabletabla.DELETE(n)**. Borra el elemento indicado por *n*. Si el valor de *n* es NULL no hará nada.

- **variabletabla.DELETE(n1, n2)**. Borra las filas comprendidas entre *n1* y *n2*, siendo *n1>=n2* (en caso contrario no hará nada).

Todos estos atributos están disponibles para todas las colecciones (VARRAYS, TABLAS ANIDADAS y TABLAS INDEXADAS); pero además existen otros que sólo están disponibles para alguna de las dos primeras:

- EXTEND. Reserva espacio para un nuevo elemento (VARRAYS y TABLAS ANIDADAS). Opcionalmente puede incluirse un parámetro que indique el número de elementos nuevos (por defecto se asume uno):

variabletabla.EXTEND; o bien **variabletabla.EXTEND(n);**

El atributo EXTEND se puede usar también en los VARRAYS, siempre que no se exceda el límite indicado en la declaración (por ejemplo, cuando se inicializó con menos elementos de los previstos en la declaración) pues de lo contrario dará error.

- TRIM. Elimina el último elemento (el índice más alto en VARRAYS y TABLAS ANIDADAS). Opcionalmente puede incluirse un parámetro que indique el número de elementos a eliminar (comenzando en el último, penúltimo, etcétera):

variabletabla.TRIM; o bien **variabletabla.TRIM(n);**

- LIMIT. Devuelve el valor más alto permitido en un VARRAY:

variabletabla.LIMIT

» Creación del cuerpo del proyecto

Crear los recursos básicos más importantes de la declaración de tipos, variables, funciones, procedimientos y clase, así como sus respectivas implementaciones.



13. Programación avanzada

13.3 Registros y colecciones



Caso práctico

- 4 A continuación reescribiremos el código del Caso práctico 3 usando una tabla indexada y los atributos disponibles para recorrer la tabla.

```
DECLARE
    CURSOR c_depar IS          /* Declaramos un cursor basado en una consulta */
        SELECT depart.dept_no, dnombre, count(emp_no) numemple
        FROM depart, emple
        WHERE depart.dept_no = emple.dept_no
        GROUP BY depart.dept_no, dnombre;
    TYPE tr_dept IS RECORD (      /* Definimos un tipo compatible con el cursor */
        nombredep depart.dnombre%TYPE,
        numemple INTEGER);
    /* Definimos un tipo TABLA INDEXADA basado en el tipo anterior */
    TYPE ti_dept IS TABLE OF tr_dept INDEX BY PLS_INTEGER;
    va_departamentos ti_dept; /* Declaramos la variable del tipo TABLA INDEXADA */
    n PLS_INTEGER := 0;         /* Declaramos una variable para usarla como índice */
BEGIN
    FOR vc IN c_depar LOOP      /* Cargar valores. El indice es el NºDpto */
        va_departamentos(vc.dept_no).nombredep := vc.dnombre;
        va_departamentos(vc.dept_no).numemple := vc.numemple;
    END LOOP;
    n := va_departamentos.FIRST;
    WHILE va_departamentos.EXISTS(n) LOOP /* Mostrar los datos de la variable */
        DBMS_OUTPUT.PUT_LINE(' * Dep N° : ' || n || ' * Dnombre: ' || va_departamentos(n).nombredep ||
            ' * NºEmpleados: ' || va_departamentos(n).numemple );
        n := va_departamentos.NEXT(n);
    END LOOP;
END;
/
```



13.4 Paquetes

Se utilizan para agrupar y guardar programas y otros objetos en la base de datos. En un entorno de producción, todos los programas y objetos (procedimientos, funciones cursores, etcétera) de una determinada aplicación o que gestionen un área de negocio, se agruparán en un paquete o en varios. También se pueden crear paquetes con utilidades que den soporte a distintas aplicaciones.

Oracle dispone de paquetes predefinidos (DBMS_OUTPUT, DBMS_STANDARD, ...) donde se encuentran muchas de las funciones y utilidades que hemos venido utilizando hasta ahora.

A. Elementos de un paquete

En los paquetes hay dos elementos claramente diferenciados:

- **Especificación:** contiene declaraciones públicas de subprogramas, tipos, constantes, variables, cursores, excepciones, etcétera. Los objetos declarados en la especificación son accesibles también desde fuera del paquete. Actúa como una interfaz con el exterior.
- **Cuerpo:** contiene los detalles de implementación de todos los objetos del paquete (el código de los programas, etcétera). También puede incluir declaraciones de objetos accesibles solamente desde los objetos del paquete.

B. Creación de un paquete

Tanto la especificación como el cuerpo se pueden crear desde SQL*PLUS mediante los comandos CREATE [OR REPLACE] PACKAGE y CREATE [OR REPLACE] PACKAGE BODY, respectivamente. El formato genérico es:

- **Creación de cabecera o especificación:**

```
CREATE [OR REPLACE] PACKAGE nombredepaquete AS
<declaraciones de tipos, variables, cursores, excepciones
y otros objetos públicos>
<especificación de subprogramas>
END [nombredepaquete];
```

- **Creación del cuerpo del paquete:**

```
CREATE [OR REPLACE] PACKAGE BODY mombredepaquete AS
<declaraciones de tipos, variables, cursores, excepciones
y otros objetos privados>
    <cuerpo de los subprogramas>
[BEGIN
    <instrucciones iniciales>
END [nombredepaquete];
```



13. Programación avanzada

13.4 Paquetes

El siguiente ejemplo servirá para ilustrar la utilización de paquetes en PL/SQL. En primer lugar se crea la cabecera o especificación del paquete según el formato indicado.

A continuación describiremos el contenido de la cabecera del paquete:

```
/* cabecera o especificación del paquete */
CREATE OR REPLACE PACKAGE buscar_emple
AS
    TYPE t_reg_emple IS RECORD
        (num_empleado emple.emp_no%TYPE,
         apellido emple.apellido%TYPE,
         oficio emple.oficio%TYPE,
         salario emple.salario%TYPE,
         departamento emple.dept_no%TYPE);

    PROCEDURE ver_por_numero
        (v_emp_no emple.emp_no%TYPE);

    PROCEDURE ver_por_apellido
        (v_apellido emple.apellido%TYPE);

    FUNCTION datos
        (v_emp_no emple.emp_no%TYPE)
        RETURN t_reg_emple;
END buscar_emple;
```

El sistema responderá con el mensaje: PAQUETE CREADO.

O bien: Aviso: Paquete creado con errores de compilación.

En este caso habrá que depurar los posibles errores antes de proceder a crear el cuerpo del paquete.

◆ Declaración de objetos en la cabecera o especificación del paquete

Podemos observar en el ejemplo que se han declarado:

- Un tipo: TYPE t_reg_emple
- Dos procedimientos: PROCEDURE ver_por_numero
PROCEDURE ver_por_apellido
- Una función: FUNCTION datos

Las declaraciones de procedimientos y funciones en la cabecera deben ser **declaraciones formales**, es decir, contendrán únicamente la cabecera de los subprogramas, el nombre, la definición de los parámetros y el tipo de retorno en las funciones.

13. Programación avanzada

13.4 Paquetes



Todos los objetos declarados en la cabecera del paquete son accesibles tanto desde el propio paquete como desde el exterior. En este último caso se deberá utilizar el nombre del objeto precedido por el nombre del paquete utilizando la notación de punto:

Nombre_de_paquete.nombre_de_objeto

Por ejemplo, para ejecutar el procedimiento *ver_por_numero* desde SQL*Plus escribiremos:

```
SQL> execute buscar_emple.ver_por_numero(7902);
```

Se pueden utilizar, aplicando la notación indicada, los otros objetos declarados en la cabecera. Por ejemplo, se puede utilizar desde otro programa el tipo *t_reg_emple* para definir datos:

```
DECLARE
  ...
  Mi_empleado t_reg_emple;
  ...
```

◆ Creación del cuerpo del paquete y declaración de objetos locales

Una vez creada la cabecera crearemos el cuerpo del paquete, el cual:

- Incluirá el código correspondiente a las declaraciones formales realizadas en la cabecera.
- Podrá incluir otros objetos locales al paquete: tipos, variables, excepciones, procedimientos, funciones, etcétera. Estos objetos serán accesibles desde cualquier parte del paquete, pero no desde fuera.

Para poder crear el cuerpo del paquete deberemos haber creado previamente la cabecera.

```
/* Cuerpo del paquete */
CREATE OR REPLACE PACKAGE BODY buscar_emple
AS

  vg_emple t_reg_emple; /* variable local al paquete */
  PROCEDURE ver_emple; /* declaración del procedimiento
  local al paquete */
  PROCEDURE ver_por_numero
    (v_emp_no emple.emp_no%TYPE)
  IS
  BEGIN
    SELECT emp_no, apellido, oficio, salario, dept_no
    INTO vg_emple
    FROM emple
    WHERE emp_no = v_emp_no;
    ver_emple; /* llama al procedimiento ver_emple */
  END ver_por_numero;
```



13. Programación avanzada

13.4 Paquetes

```
PROCEDURE ver_por_apellido
  (v_apellido emple.apellido%TYPE)
IS
BEGIN
  SELECT emp_no, apellido, oficio, salario, dept_no
  INTO vg_emple
  FROM emple
  WHERE apellido = v_apellido;
  ver_emple;
END ver_por_apellido;

FUNCTION datos
  (v_emp_no emple.emp_no%TYPE)
RETURN t_reg_emple
IS
BEGIN
  SELECT emp_no, apellido, oficio, salario, dept_no
  INTO vg_emple
  FROM emple
  WHERE emp_no = v_emp_no;
  RETURN vg_emple;
END datos;

PROCEDURE ver_emple /* Implementación del procedimiento*/
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(vg_emple.num_empleado ||
'*'||vg_emple.apellido||'*'||vg_emple.oficio ||
'*'||vg_emple.salario||'*'||vg_emple.departamento);

  END ver_emple;
END buscar_emple;
/
```

El sistema responderá con el mensaje: **CUERPO DEL PAQUETE CREADO.**

Hemos de tener en cuenta que no se podrá compilar el cuerpo del paquete hasta que se haya creado correctamente la cabecera, ya que, de lo contrario, se producirá el siguiente error:

```
LINE/COL ERROR
-----
0/0  PL/SQL: Compilation unit analysis terminated
1/14 PLS-00905: el objeto FERNANDO.BUSCAR_EMPL no es
válido
1/14 PLS-00304: no se puede compilar el cuerpo de 'BUS-
CAR_EMPL' sin su especificación
```



En nuestro ejemplo, además del código correspondiente a los subprogramas declarados en la cabecera, el cuerpo del procedimiento incluye:

- Una variable: `vg_emple` de tipo `t_reg_emple`;
- Un procedimiento: `PROCEDURE ver_emple;`

C. Utilización de los objetos definidos en el paquete

- **Desde el mismo paquete.** Todos los objetos declarados en el paquete pueden ser utilizados por los demás objetos del mismo, con independencia de que hayan sido o no declarados en la especificación. Además, no necesitan utilizar la notación de punto para referirse a los objetos del mismo paquete.

Los procedimientos `ver_por_numero` y `ver_por_apellido` llaman al procedimiento `ver_emple`, que no está en la especificación. Asimismo, utilizan la variable `vg_emple`, que tampoco se encuentra en la especificación.

- **Desde fuera del paquete.** Los subprogramas y otros objetos contenidos en el paquete son accesibles desde fuera solamente si han sido declarados en la especificación. En nuestro ejemplo no se podría hacer referencia desde fuera del paquete al procedimiento `ver_emple`, ya que no ha sido declarado en la cabecera.

Para ejecutar un procedimiento de un paquete desde SQL*Plus se utilizará el formato:

```
SQL> EXECUTE nombrepaquete.nomsubprog(listaparam);
```

Solamente se podrán ejecutar desde fuera los subprogramas que hayan sido declarados en la especificación.

A continuación, veremos dos ejemplos de ejecución:

```
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE buscar_emple.ver_por_numero(7902);
7902*FERNANDEZ*ANALISTA*3900*20
```

```
SQL> EXECUTE buscar_emple.ver_por_apellido('JIMENO');
7900*JIMENO*EMPLEADO*1235*30
```

Se pueden utilizar los subprogramas declarados en la especificación desde otros subprogramas que se encuentran fuera del paquete. El siguiente procedimiento visualiza el apellido, el salario y el oficio de un empleado cuyo número se pasa en la llamada, haciendo uso del tipo `buscar_emple.t_reg_emple` y de la función `buscar_emple.datos`:

```
CREATE OR REPLACE PROCEDURE ver_datos_empleado
(v_num_emple emple.emp_no%TYPE)
AS
    vr_emple buscar_emple.t_reg_emple;
```



13. Programación avanzada

13.4 Paquetes

```
zona BEGIN
    vr_emple := buscar_emple.datos(v_num_emple);
    DBMS_OUTPUT.PUT_LINE(vr_emple.apellido ||'*'|||
                           vr_emple.salario||'*'|||
                           vr_emple.oficio);
END ver_datos_empleado;
```

D. Declaración de cursos en paquetes

Para declarar cursos en paquetes, si queremos que estén accesibles desde fuera, deberemos separar la declaración del cursor del cuerpo (en éste es donde va la cláusula SELECT).

La declaración del cursor se incluirá en la cabecera del paquete (para que pueda estar accesible desde fuera del paquete) indicando el nombre del cursor, los parámetros (si procede) y el tipo devuelto. Este último se indicará mediante RETURN *tipodedato*; para cursos que devuelven filas enteras normalmente se usará %ROWTYPE.

```
CREATE PACKAGE empleados_acc AS
...
CURSOR c1 RETURN emple%ROWTYPE;
...
END empleados_acc;
CREATE PACKAGE BODY empleados_acc AS
...
CURSOR c1 RETURN emple%ROWTYPE;
SELECT * FROM emple WHERE salario > 1000;
...
END empleados_acc;
```

E. Ámbito y otras características de las declaraciones

Se pueden utilizar paquetes exclusivamente para declarar tipos, constantes, variables, excepciones, cursos, etcétera. En estos casos no es necesario el cuerpo del paquete.

Los objetos declarados en la especificación son accesibles desde el propio paquete y también desde fuera (en este caso, usando la notación de punto). Además, todas las variables declaradas en la especificación del paquete mantienen su valor durante la sesión, por tanto, el valor no se pierde entre las llamadas de los subprogramas.

Respecto a las variables constantes y cursos declarados en un paquete:

- El propietario es la sesión.
- En la sesión no se crean los objetos hasta que se referencia el paquete.
- Cuando se crean los objetos su valor será nulo (salvo que se inicialice).
- Durante la sesión los valores pueden cambiarse.
- Al salir de la sesión los valores se pierden.



F. Características de almacenamiento y compilación

Tanto el código fuente como el código compilado de los paquetes se almacena en la base de datos. Al igual que ocurría con los subprogramas almacenados, el *paquete* (la especificación) puede tener dos estados:

- Disponible (*valid*).
- No disponible (*invalid*).

Cuando se borra o modifica alguno de los objetos referenciados el paquete pasará a *invalid*. Si la especificación del paquete se encuentra «no disponible», Oracle invalida (pasa a *invalid*) cualquier objeto que haga referencia al paquete.

Cuando recompilamos la especificación, todos los objetos que hacen referencia al paquete pasan a «no disponible» hasta que sean compilados de nuevo. Cualquier referencia o llamada a uno de estos objetos antes de ser recompilados producirá que Oracle automáticamente los recompile. Esto ocurre también con el cuerpo del paquete, pero en este caso se puede recomilar el paquete sin invalidar la especificación. Esto evita las recompilaciones en cascada innecesarias.

Si se cambia la definición o implementación (*cuerpo*) de una función o procedimiento incluido en un paquete, no hay que recomilar todos los programas que llaman al subprograma (como ocurre con los subprogramas almacenados), a no ser que también se cambie la especificación de dicha función o procedimiento.

G. Paquetes suministrados por Oracle

Oracle incluye con su gestor de bases de datos diversos paquetes como STANDARD, DBMS_OUTPUT, DBMS_STANDARD, DBMS_SQL, y muchos otros que incorporan diversas funcionalidades. A continuación, veremos una breve reseña del contenido de estos paquetes:

- En STANDARD se declaran tipos, excepciones, funciones, etcétera, disponibles desde el entorno PL/SQL. Por ejemplo, en el paquete STANDARD están definidas las funciones:

```
FUNCTION ABS (n number) RETURN NUMBER;
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

Todos los paquetes suministrados por Oracle están documentados en los manuales del producto.

Si, por alguna circunstancia, volvemos a declarar alguna de esas funciones desde un programa PL/SQL, siempre podremos hacer referencia a la original mediante la notación de punto. Por ejemplo, ... STANDARD.ABS(...)



13. Programación avanzada

13.5 SQL Dinámico

- DBMS_STANDARD incluye utilidades, como el procedimiento RAISE_APPLICATION_ERROR, que facilitan la interacción de nuestras aplicaciones con Oracle.
- En DBMS_OUTPUT se encuentra el procedimiento PUT_LINE que hemos venido utilizando para visualizar datos, y otros como ENABLE y DISABLE, que permiten configurar y purgar el buffer utilizado por PUT_LINE.
- DBMS_SQL incorpora procedimientos y funciones que permiten utilizar SQL dinámico en nuestros programas, tal como veremos en el apartado siguiente.

13.5 SQL Dinámico

Cuando se compila un procedimiento, PL/SQL comprueba en el diccionario de datos completa todas las referencias a objetos de la base de datos (tablas, columnas, usuarios, etcétera). De esta forma, si algún objeto no existe en el momento de la compilación, el proceso fallará, indicando el error correspondiente.

Esta manera de trabajar se denomina **SQL estático** y tiene importantes ventajas, especialmente en cuanto a velocidad en la ejecución y a la seguridad de que las referencias a los objetos han sido comprobadas previamente; pero también tiene algunos inconvenientes:

- Todos los objetos tienen que existir en el momento de la compilación.
- No se pueden crear objetos nuevos con el programa.
- No se puede cambiar la definición de los objetos.
- Cualquier referencia a un objeto debe ser conocida y resuelta en tiempo de compilación.

Existe la posibilidad de superar estas limitaciones y ejecutar instrucciones de definición de datos, así como resolver referencias a objetos en el momento de la ejecución. A esta forma de trabajar se le denomina **SQL dinámico**.

Hasta la versión 9i la única posibilidad de trabajar con SQL dinámico era usar el paquete **DBMS_SQL**. A partir de esta versión Oracle incorpora la posibilidad de trabajar con **SQL dinámico nativo (NDS)**. Nosotros estudiaremos las dos posibilidades pues, aunque la tendencia actual es trabajar con NDS, hay mucho código desarrollado con DBMS_SQL y es la única posibilidad soportada en instalaciones anteriores.

A. SQL dinámico con DBMS_SQL

El siguiente programa es capaz de ejecutar órdenes de definición y manipulación sobre objetos que sólo se conocerán al ejecutar el programa, ya que el comando se pasará en la llamada al procedimiento:

13. Programación avanzada

13.5 SQL Dinámico



```
CREATE OR REPLACE PROCEDURE ejsqldin
  (instrucion VARCHAR2)
AS
  id_cursor INTEGER;
  v_dummy INTEGER;
BEGIN
  id_cursor := DBMS_SQL.OPEN_CURSOR;           /*Abrir*/
  DBMS_SQLPARSE(id_cursor, instrucion, DBMS_SQL.V7);
/*Analizar*/
  v_dummy := DBMS_SQL.EXECUTE(id_cursor);        /*Ejecutar*/
  DBMS_SQL CLOSE_CURSOR(id_cursor);             /*Cerrar*/
EXCEPTION
  WHEN OTHERS THEN
    DBMS_SQL CLOSE_CURSOR(id_cursor);           /*Cerrar*/
RAISE;
END ejsqldin;
/
```

En este ejemplo podemos observar resaltadas las líneas correspondientes a las cuatro operaciones más frecuentes que se suelen realizar al programar con SQL dinámico, así como los subprogramas incluidos en DBMS_SQL que se encargan de realizar dichas operaciones:

- DBMS_SQL.OPEN_CURSOR abre el cursor y devuelve un número de identificación para poder utilizarlo. Recordemos que todos los comandos SQL se ejecutan dentro de un cursor.
- DBMS_SQLPARSE analiza la instrucción que se va a ejecutar. Este paso es necesario, ya que cuando se compiló el programa, la instrucción no pudo ser analizada, pues aún no se había construido.
- DBMS_SQL.EXECUTE se encarga de la ejecución.
- DBMS_SQL CLOSE_CURSOR cierra el cursor utilizado.

Debemos tener en cuenta que, cuando se definen objetos dinámicamente, el propietario del procedimiento deberá tener concedidos los privilegios necesarios (CREATE USER, CREATE TABLE, etcétera) de forma directa, no mediante un rol.

```
SQL> EXECUTE EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY
DUMM1');
begin EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY DUMM1');
end;
*
ERROR en línea 1:
ORA-01031: privilegios insuficientes
ORA-06512: en "SYSTEM.EJSQLDIN", línea 14
ORA-06512: en línea 1
```

Los objetos se crean con éxito en el esquema del usuario que hace el procedimiento. Si se pretende crearlos en otro esquema, se deberá indicar explícitamente.



13. Programación avanzada

13.5 SQL Dinámico

Observemos que se trata del usuario SYSTEM, pero tiene el privilegio CREATE USER mediante un rol. Para solucionar el problema se concede el privilegio de forma directa:

```
SQL> GRANT CREATE USER TO SYSTEM;
Concesión terminada con éxito.
SQL> EXECUTE EJSQLDIN('CREATE USER DUMM1 IDENTIFIED BY
DUMM1');
Procedimiento PL/SQL terminado con éxito.
```

A continuación, podemos observar otros ejemplos de ejecución:

```
SQL> EXECUTE EJSQLDIN('CREATE TABLE PR1 (C1 CHAR)');
Procedimiento PL/SQL terminado con éxito.

SQL> EXECUTE EJSQLDIN('ALTER TABLE PR1 ADD COMENTARIO
VARCHAR2(20)')
Procedimiento PL/SQL terminado con éxito.

SQL> DESCRIBE PR1
Name          Null?    Type
-----  -----
C1           CHAR(1)
COMENTARIO   VARCHAR2(20)
```

● Pasos para utilizar SQL Dinámico con DBMS_SQL

Aunque hay diferencias dependiendo del tipo de instrucción que se va a procesar, los pasos que hay que seguir son los siguientes:

1. Abrir el cursor (OPEN_CURSOR) y guardar su número de identificación para posteriores referencias.
2. Analizar (PARSE) el comando que se va a procesar.
3. Si se van a pasar valores al comando, acoplar las variables de entrada (BIND).
4. Si el comando es una consulta:
 - Definir columnas (DEFINE_COLUMN).
 - Ejecutar (EXECUTE).
 - Recuperar los valores con FETCH_ROWS y COLUMN_VALUE.
5. Si se trata de un comando de manipulación, ejecutar (EXECUTE).
6. Cerrar el cursor (CLOSE_CURSOR).



◆ Comandos de definición de datos con DBMS_SQL

Utilizando los procedimientos y funciones incluidos en el paquete DBMS_SQL podemos crear objetos en tiempo de ejecución. Por ejemplo, el siguiente procedimiento creará una tabla de una sola columna cuyos datos se pasarán en la llamada al procedimiento:

```

CREATE OR REPLACE PROCEDURE creartabla
    (nombretabla VARCHAR2,
     nombrecol VARCHAR2,
     longitudcol POSITIVE)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
BEGIN
    id_cursor := DBMS_SQL.OPEN_CURSOR; /*Abre el cursor*/
    v_comando := 'CREATE TABLE '|| nombretabla || '(
        '|| nombrecol || ' VARCHAR2('|| longitudcol || '))';
    /*Analiza y ejecuta*/
    DBMS_SQLPARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
    DBMS_SQLCLOSE_CURSOR(id_cursor); /*Cierra el cursor*/
EXCEPTION
    WHEN OTHERS THEN
        DBMS_SQLCLOSE_CURSOR(id_cursor); /*Cierra el cursor*/
        RAISE;
END creartabla;
/

```

- **OPEN_CURSOR.** Esta función abre un cursor nuevo y retorna un identificador del mismo, que se guarda en una variable y servirá para hacer referencia al cursor desde el programa.
- **PARSE.** Analiza el comando comprobando posibles errores. Si se trata de un comando DDL (como en este caso), PARSE, además, lo ejecuta. Recibe tres parámetros:
 - El identificativo del cursor.
 - La cadena que contiene el comando (sin el ;).
 - El indicador del lenguaje, que determina la manera en que Oracle manejará el comando SQL. Están disponibles las siguientes opciones:
 - V7: estilo Oracle versión 7 (es el mismo para la versión 8 y la 9).
 - NATIVE: estilo por defecto de la base de datos en la que esté conectada la sesión.
- **CLOSE_CURSOR.** Cierra el cursor especificado y libera la memoria utilizada por el cursor.

Al utilizar SQL Dinámico aumentan las posibilidades de que aparezcan errores durante la ejecución del programa. Por eso, es muy importante incluir un manejador WHEN OTHERS que cierre el cursor.

Los objetos se crearán por defecto en el esquema del usuario propietario del procedimiento. Si se pretende crearlos en otro esquema, se deberá indicar explícitamente.



13. Programación avanzada

13.5 SQL Dinámico

◆ Comandos de manipulación de datos con DBMS_SQL

También se pueden crear dinámicamente *instrucciones de manipulación de datos*.

El siguiente ejemplo muestra la utilización de PL/SQL Dinámico para introducir datos en una tabla que se indicará en la llamada (las llamadas a PUT_LINE se utilizan para ilustrar el funcionamiento del programa):

```
CREATE OR REPLACE PROCEDURE introducir_fila
    (nombretabla VARCHAR2,
     valor VARCHAR2)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
    v_filas INTEGER;
BEGIN
    id_cursor := DBMS_SQL.OPEN_CURSOR;
    v_comando := 'INSERT INTO ' || nombretabla || '
VALUES (:val_1)';
    DBMS_OUTPUT.PUT_LINE(v_comando);
    DBMS_SQLPARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
    DBMS_SQLBIND_VARIABLE(id_cursor, ':val_1', valor);
    /*Acopla la variable que pasará el valor de entrada*/
    v_filas := DBMS_SQL.EXECUTE(id_cursor); /*Ejecuta el
comando*/
    DBMS_SQLCLOSE_CURSOR(id_cursor);
    DBMS_OUTPUT.PUT_LINE('Filas introducidas: ' || v_filas);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_SQLCLOSE_CURSOR(id_cursor);
        RAISE;
END introducir_fila;
```

La línea `INSERT INTO mitabla VALUES (:val_1)` corresponde a la visualización del comando construido, que se incluye con fines didácticos.

En este ejemplo podemos apreciar que, para introducir dinámicamente comandos de manipulación de datos, utilizamos los procedimientos y funciones estudiados en el apartado anterior: OPEN_CURSOR, PARSE y CLOSE_CURSOR.

Pero, además, utilizaremos el procedimiento EXECUTE (necesario para comandos de manipulación y consulta). También se utiliza en este caso el procedimiento BIND_VARIABLE para acoplar la variable con el valor de entrada.

A continuación, podemos observar un ejemplo de ejecución del procedimiento cuyo resultado es insertar una fila en la tabla mitabla que suponemos creada previamente:

```
SQL> execute introducir_fila('mitabla', 'HOLA MUNDO');
SQL> INSERT INTO mitabla VALUES (:val_1)
SQL> Filas introducidas: 1
SQL> Procedimiento PL/SQL terminado con éxito.
```

13. Programación avanzada

13.5 SQL Dinámico



Podemos comprobar el resultado de la ejecución introduciendo la siguiente consulta:

```
SQL> select * from mitabla;
COL1
```

```
HOLA MUNDO
```

Consultas con DBMS_SQL

El siguiente ejemplo permite obtener el número de empleado, el apellido y el oficio de los empleados que cumplan la condición que se especificará en la llamada:

```
CREATE OR REPLACE PROCEDURE consultar_emple
    condicion VARCHAR2,
    valor VARCHAR2)
AS
    id_cursor INTEGER;
    v_comando VARCHAR2(2000);
    v_dummy NUMBER;
    v_emp_no emple.emp_no%TYPE;
    v_apellido emple.apellido%TYPE;
    v_oficio emple.oficio%TYPE;
BEGIN
    id_cursor := DBMS_SQL.OPEN_CURSOR;
    v_comando := 'SELECT emp_no, apellido, oficio
                  FROM emple
                 WHERE ' || condicion || ':val_1';

    DBMS_OUTPUT.PUT_LINE(v_comando);
    DBMS_SQLPARSE(id_cursor, v_comando, DBMS_SQL.NATIVE);
    DBMS_SQLBIND_VARIABLE(id_cursor, ':val_1', valor);

    /* Se especifican las variables que recibirán los valores
     de la selección*/
    DBMS_SQL.DEFINE_COLUMN(id_cursor, 1, v_emp_no);
    DBMS_SQL.DEFINE_COLUMN(id_cursor, 2, v_apellido, 14);
    DBMS_SQL.DEFINE_COLUMN(id_cursor, 3, v_oficio, 14);
    v_dummy := DBMS_SQL.EXECUTE(id_cursor);

    /* La función FETCH_ROWS recupera filas y retorna el
     número de filas que quedan */
    WHILE DBMS_SQL.FETCH_ROWS(id_cursor)>0 LOOP

        /* A continuación se depositarán los valores recuperados
         en las variables PL/SQL */
    END LOOP;
```



13. Programación avanzada

13.5 SQL Dinámico

```
DBMS_SQL.COLUMN_VALUE(id_cursor, 1, v_emp_no);
DBMS_SQL.COLUMN_VALUE(id_cursor, 2, v_apellido);
DBMS_SQL.COLUMN_VALUE(id_cursor, 3, v_oficio);
DBMS_OUTPUT.PUT_LINE(v_emp_no || '*' || v_apellido
                     || '*' || v_oficio);
END LOOP;
DBMS_SQL.CLOSE_CURSOR(id_cursor);
EXCEPTION
  WHEN OTHERS THEN
    DBMS_SQL.CLOSE_CURSOR(id_cursor);
    RAISE;
END consultar_emple;
/
```

La principal diferencia de este programa con los anteriores es que, en este caso, como ocurre en todas las consultas, hay que recoger los valores obtenidos por la consulta. Para ello, se utilizan los siguientes procedimientos y funciones:

El paquete DBMS_SQL dispone de otras posibilidades que quedan fuera del objetivo de este libro. Para profundizar más en este tema, se recomienda consultar el manual de Oracle *Application Developer's Guide*.

- **DEFINE_COLUMN:** sirve para especificar las variables que recibirán los valores de la selección. Se utilizará una llamada para cada variable indicando:
 - El identificador del cursor.
 - El número de la columna de la cláusula SELECT de la que recibirá el valor.
 - El nombre de la variable.
 - Si la variable es de tipo CHAR, VARCHAR2 o RAW, hay que especificar la longitud.
- **FETCH_ROWS:** es similar al comando FETCH de PL/SQL estático, pero con dos particularidades muy importantes:
 1. Devuelve un valor numérico que indica el número de filas que quedan por recuperar en el cursor. Puede servir, como en el ejemplo, para controlar el número de iteraciones del bucle.
 2. Para introducir los datos recuperados en las variables PL/SQL todavía hay que utilizar el siguiente procedimiento.
- **COLUMN_VALUE:** deposita el valor de un elemento del cursor (recuperado con FETCH_ROWS) cuya posición se especifica en una variable PL/SQL. Normalmente se trata de la variable indicada en COLUMN_VALUE para el mismo elemento del cursor.

La función EXECUTE con instrucciones DML devuelve el número de filas afectadas. Sin embargo, en el caso de las consultas (y también en órdenes DDL si se usa) el valor que devuelve es indefinido.

```
SQL> EXECUTE CONSULTAR_EMPL('SALARIO > ', 3850)
SELECT emp_no, apellido, oficio
FROM emple
WHERE SALARIO > :val_1
```

13. Programación avanzada

13.5 SQL Dinámico



```
7566*JIMENEZ*DIRECTOR  
7788*GIL*ANALISTA  
7839*REY*PRESIDENTE  
7902*FERNÁNDEZ*ANALISTA
```

B. SQL dinámico con NDS

Las versiones más recientes de Oracle permiten trabajar con NDS (*Native Dynamic SQL*) que tiene importantes ventajas respecto al uso del paquete DBMS_SQL:

- La sintaxis es más sencilla y parecida a SQL.
- Manejo directo de FETCH desde PL/SQL.
- Soporta objetos y tipos definidos por el usuario.
- Los programas se ejecutan más rápido.

Una de las principales novedades de NDS es el comando EXECUTE IMMEDIATE que recibe una cadena de tipo VARCHAR2 conteniendo el comando a ejecutar, lo analiza y ejecuta.

```
CREATE OR REPLACE PROCEDURE ejsqldin_nds  
    (instrucion VARCHAR2)  
AS  
BEGIN  
    EXECUTE IMMEDIATE instrucion;  
END ejsqldin_nds;  
/
```

La instrucción se puede recibir en uno o en varios parámetros, completa o incompleta. El comando se formará concatenando los parámetros, variables, literales, etcétera.

Por ejemplo, el procedimiento de abajo recibirá el nombre de una tabla, el de una columna de dicha tabla y un valor de dicha columna, y eliminará la fila o filas cuyo valor en la columna especificada coincida con el indicado.

```
CREATE OR REPLACE PROCEDURE borrar  
    (nombre_tabla VARCHAR2,  
     columnaref VARCHAR2,  
     valor      VARCHAR2)  
AS  
    instrucion_borrado VARCHAR2(2000);  
BEGIN  
    instrucion_borrado := 'DELETE FROM ' || nombre_tabla ||  
    ' WHERE ' || columnaref || ' = ' || valor;  
    EXECUTE IMMEDIATE instrucion_borrado;  
END;
```



13. Programación avanzada

13.6 Objetos con PL/SQL

Para probarlo podemos escribir:

```
SQL> EXECUTE BORRAR('EMPLE', 'EMP_NO', 7902);
```

También se pueden usar *variables de transferencia*. En este caso se indicará mediante opción USING las variables (o valores) que se asignaran a las variables de transferencia contenidas en la instrucción.

```
SQL> EXECUTE IMMEDIATE instruccion USING lista de parámetros;
```

La asignación de los parámetros o valores de USING a las variables de transferencia es posicional.

Por ejemplo, el procedimiento anterior usando una variable de transferencia para indicar el valor que queremos borrar (y evitar problemas con valores de tipos distintos al numérico) sería:

```
CREATE OR REPLACE PROCEDURE borrar
  (nombre_tabla VARCHAR2,
   columnaref VARCHAR2,
   valor VARCHAR2)
AS
  instruccion_borrado VARCHAR2(2000);
BEGIN
  instruccion_borrado := 'DELETE FROM ' || nombre_tabla
    || ' WHERE ' || columnaref || ' = :vt1'; -- vt1 es la
    variable de transferencia.
  EXECUTE IMMEDIATE instruccion_borrado
    USING valor;
END;
```

13.6 Objetos con PL/SQL

Oracle incluye la posibilidad de trabajar con objetos desde la versión 8. En sucesivas versiones se ha implementado soporte para características avanzadas propias de la programación orientada a objetos mediante PL/SQL (herencia, polimorfismo, etcétera).

A. El tipo OBJECT

El tipo **OBJECT** es una definición o molde de objeto. Es el equivalente a una clase en Java. Al definir un tipo OBJECT podemos definir atributos y métodos:

- Los **atributos** son las características que sirven para describir físicamente el objeto.
- Los **métodos** son las acciones que se pueden realizar con el objeto (procedimientos y funciones).

13. Programación avanzada

13.6 Objetos con PL/SQL



```
CREATE OR REPLACE
TYPE nombre_de_tipo AS OBJECT (lista_atributos, [lista_
métodos]);
```

Por ejemplo, podemos definir el tipo COCHE_OBJ con los atributos *marca*, *modelo*, *nbastidor*, *fechaf*, *valorff*, *matrícula*:

```
CREATE OR REPLACE TYPE coche_obj AS OBJECT (
    marca      VARCHAR2(20),
    modelo     VARCHAR2(10),
    nbastidor  NUMBER(15),
    fechaf     DATE,
    valorff    NUMBER(8,2),
    matricula   VARCHAR2(10)
);
```

El sistema responderá: Tipo creado.

Observamos que en este caso sólo hemos incluido atributos dentro de la declaración, pues no es obligatorio definir ningún método al crear un tipo OBJECT. En cambio, sí es obligatorio definir al menos un atributo. Además, los atributos deben cumplir:

- Se puede usar cualquier tipo soportado por la base de datos excepto ROWID, UROWID, LONG, LONG RAW, NCHAR, NVARCHAR2, NCLOB.
- No se pueden usar tipos específicos de PL/SQL no soportados por la base de datos.
- No se pueden incluir calificadores NOT NULL ni DEFAULT.
- Tampoco podemos usar el atributo %TYPE o %ROWTYPE para definir el tipo.
- Se pueden incluir tipos definidos como OBJECT dentro de la definición de otros. A estos se les llama **tipos OBJECT complejos**.

Una vez declarado el tipo podemos usarlo para declarar e inicializar objetos. Esto lo haremos en la sección declarativa igual que si se tratase de cualquier otro tipo predefinido:

```
DECLARE
    v_coche COCHE_OBJ := COCHE_OBJ('SEAT', 'LEON TDI', NULL,
                                     NULL, NULL, NULL);
```

Observamos que al declarar el objeto lo hemos inicializado pues todo objeto debe ser inicializado antes de usarlo. Para ello, se utiliza el constructor por defecto del tipo, seguido de la lista de valores para los atributos (entre paréntesis).

En caso de que haya algún error de compilación al crear el objeto aparecerá:

... Warning: Type created with compilation errors.

En estos casos, usaremos SHOW ERRORS y depuraremos los errores.



13. Programación avanzada

13.6 Objetos con PL/SQL

B. Métodos

Normalmente, cuando creamos un objeto también crearemos los métodos que definen el comportamiento de ese objeto y que permiten actuar sobre él.

Los métodos son procedimientos y funciones que se especificarán después de los atributos del objeto indicando también el tipo de método, que puede ser:

- **MEMBER.** Son métodos que sirven para interactuar con los objetos. Se trata de funciones y procedimientos con un comportamiento y formato similares a los estudiados en los paquetes.

En nuestro caso podríamos crear un método que se llame MATRICULAR que consiste en una función que asigna una matrícula a un objeto (una variable) de tipo COCHE_OBJ:

```
...  
MEMBER PROCEDURE matricular (mat VARCHAR2(10))
```

- **STATIC.** Son métodos estáticos independientes de las instancias del objeto. Existe una similitud en su comportamiento con las variables estáticas definidas en la cabecera de los paquetes dentro de la sección ejecutable opcional.
- **CONSTRUCTOR.** Sirve para inicializar un objeto. Se trata de una función cuyos argumentos son los valores de los atributos del objeto y que devuelve el objeto inicializado.

Para cada objeto existe un constructor predefinido por Oracle (cuyo nombre y atributos son los mismos que los del objeto).

Por tanto, no es necesario crear un constructor, pues disponemos de uno por defecto. No obstante, podemos sobrescribirlo y/o crear otros constructores adicionales; además, creando nuestros propios constructores podemos incluir valores por defecto, restricciones, etcétera.

Los constructores llevarán en la cláusula `return` la expresión `RETURN SELF AS RESULT`.

Cabe subrayar que los métodos STATIC y MEMBER pueden ser procedimientos y funciones, mientras que los constructores CONSTRUCTOR son siempre funciones.

Una vez declarada la especificación de los métodos crearemos el cuerpo del nuevo tipo OBJECT (de manera similar a como lo hacímos con el cuerpo de un paquete) mediante la instrucción:

```
CREATE OR REPLACE TYPE BODY nombre_de_tipo AS  
<implementación de los métodos>
```

Donde `<implementación de los métodos>` incluye el código de todos los métodos ademas del tipo de los mismos según los formatos:

13. Programación avanzada

13.6 Objetos con PL/SQL



Para los procedimientos	Para las funciones
[STATIC MEMBER] PROCEDURE nombre_procedimiento [(lista_de_parámetros)] IS declaraciones; ... BEGIN instrucciones; ... END;	[STATIC MEMBER CONSTRUCTOR] FUNCTION nombre_funcion [(lista_de_parámetros)] RETURN tipo_valor_retorno IS declaraciones; ... BEGIN instrucciones; ... END;

En nuestro ejemplo, podemos añadir a la declaración del tipo COCHE_OBJ los siguientes métodos:

- mostrar_coche: visualiza todos los datos del coche.
- cambiar_precio_pct: modifica el precio aplicando el porcentaje de subida o bajada.
- ver_precio_total: devuelve el precio con IVA.

La especificación de COCHE_OBJ con los tres métodos enunciados será:

```
CREATE OR REPLACE TYPE coche_obj AS OBJECT (  
    marca      VARCHAR2(20),  
    modelo     VARCHAR2(10),  
    nbastidor  NUMBER(15),  
    fechaf     DATE,  
    valorff    NUMBER(8,2),  
    matricula   VARCHAR2(10),  
    MEMBER PROCEDURE mostrar_coche,  
    MEMBER PROCEDURE cambiar_precio_pct (pct NUMBER),  
    MEMBER FUNCTION ver_precio_total RETURN NUMBER  
)
```

Una vez creada la especificación crearemos el cuerpo:

```
CREATE OR REPLACE TYPE BODY coche_obj  
AS  
-----  
MEMBER PROCEDURE mostrar_coche  
IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE(marca || ' ' || modelo || ' ' ||  
                         nbastidor || ' ' || fechaf || ' ' ||  
                         valorff || ' ' || matricula);  
END;
```



13. Programación avanzada

13.6 Objetos con PL/SQL

Para referirnos a la instancia actual del objeto podemos usar la palabra reservada **SELF** (igual que en otros lenguajes OO).

```
-----  
MEMBER PROCEDURE cambiar_precio_pct(pct NUMBER)  
IS  
BEGIN  
  
    SELF.valorff := SELF.valorff * (1 + (pct/100));  
END;  
-----  
MEMBER FUNCTION ver_precio_total  
RETURN NUMBER  
IS  
    iva CONSTANT NUMBER(2,2) := 0.16;  
    precio_total NUMBER(10,2);  
BEGIN  
    precio_total := SELF.valorff * (1 + iva);  
    RETURN precio_total;  
END;  
-----  
END;
```

Ahora podemos usar el tipo COCHE_OBJ para crear instancias o variables de ese tipo que aceptarán los métodos implementados:

```
SET SERVEROUTPUT ON  
DECLARE  
/* declaramos y creamos una instancia. El identificador se usa dos veces, la primera se refiere al tipo, la segunda (en negrita) invoca al constructor del tipo con los valores se especifican entre paréntesis */  
v_coche COCHE_OBJ := COCHE_OBJ('SEAT', 'LEON TDI',  
123456767, SYSDATE, 20000, NULL);  
BEGIN  
  
/* invocaremos los métodos disponibles para el objeto mediante nombreinstancia.nombremetodo */  
v_coche.mostrar_coche;  
v_coche.cambiar_precio_pct(+10);  
v_coche.matricula := '5678-EZX';  
DBMS_OUTPUT.PUT_LINE('Nuevo precio incrementado en 10% con IVA:' || v_coche.ver_precio_total);  
v_coche.mostrar_coche;  
END;
```

El resultado de la ejecución será:

```
SEAT LEON TDI 123456767 22/09/05 20000  
Nuevo precio incrementado en 10% con IVA:25520  
SEAT LEON TDI 123456767 22/09/05 22000 5678-EZX  
--- Fin 170 mm ---
```



C. Ordenaciones y comparaciones con objetos

En muchas ocasiones necesitaremos poder comparar, e incluso ordenar, datos de tipos definidos como OBJECT. Este problema aparece especialmente cuando se usan operadores relacionales o cláusulas SQL como GROUP BY, ORDER BY, DISTINCT, etcétera, pues no hay un criterio de ordenación o comparación predefinido para estos datos, por tanto, normalmente deberemos crearlo mediante *métodos MAP*.

Los **métodos MAP** consisten en una función que devuelve un valor de tipo escalar (CHAR, VARCHAR2, NUMBER, DATE, ...) que será el que se utilice en las comparaciones y ordenaciones aplicando los criterios preestablecidos para estos tipos de datos.

En nuestro ejemplo, podemos establecer que los valores a partir de los cuales se ordenará serán la matrícula (VARCHAR2). También podríamos haber optado por el número de bastidor, o por el precio, o por una combinación de atributos como marca||modelo, etcétera. Una vez decidido el criterio incluiremos el método en el objeto usando el especificador MAP:

- Primero en la especificación del tipo indicaremos el nombre y el tipo de valor devuelto:

```
MAP MEMBER FUNCTION por_matricula RETURN CHAR
```

- Después en el cuerpo de la definición del tipo escribiremos el código correspondiente a la función:

```
MAP MEMBER FUNCTION por_matricula RETURN CHAR
IS
BEGIN
    RETURN SELF.matricula;
END;
```

En lugar de MAP pueden usarse métodos ORDER, que no entraremos a detallar pues suelen ser menos funcionales y eficientes.



13. Programación avanzada

Conceptos básicos



Conceptos básicos

- **Los disparadores de base de datos** son programas PL/SQL que se disparan cuando se producen ciertos eventos:
 - **Disparadores de tablas:** asociados a una tabla. Se disparan cuando se produce un evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).
 - **Disparadores de sustitución:** asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas).
 - **Disparadores del sistema:** se disparan cuando ocurre un evento del sistema o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).
- **Los registros** son estructuras compuestas de otras más simples (campos) que pueden ser de distintos tipos. Se crean:
 1. Se define el tipo genérico del registro: `TYPE nombre_tipo IS RECORD (definición_de_campos);`
 2. Se declaran las variables que se necesiten de ese tipo:
`nombre_variable nombre_tipo;`
- **Los colecciones** son estructuras compuestas por listas de elementos. Pueden ser de tres tipos:
 - **Varrays:** índice secuencial, longitud fija, acceso desde SQL y PL/SQL, soportados en tablas de la base de datos.
 - **Tablas anidadas:** igual que los varrays pero se pueden inicializar elementos adicionales dinámicamente.
 - **Tablas indexadas o arrays asociativos:** el índice no es secuencial, todos los elementos se crean dinámicamente, no requieren inicialización, no soportadas en tablas ni en SQL, no son objetos.

- **Los paquetes** se utilizan para agrupar y guardar programas y otros objetos en la base de datos. Hay dos elementos:

- **Especificación:** contiene declaraciones públicas de subprogramas, tipos, constantes, variables, cursos, excepciones, etcétera. Los objetos declarados en la especificación son accesibles también desde fuera del paquete.
- **Cuerpo:** contiene los detalles de implementación de todos los objetos del paquete (el código de los programas, etcétera). También puede incluir declaraciones de objetos accesibles solamente desde los objetos del paquete.

Los objetos declarados o definidos en un paquete se pueden usar:

- Desde el mismo paquete por los demás objetos del mismo, con independencia de que hayan sido o no declarados en la especificación. No necesitan utilizar la notación de punto para referirse a los objetos del mismo paquete.

- Desde fuera del paquete solamente si han sido declarados en la especificación. Se usará la notación de punto.

- **SQL dinámico** permite ejecutar instrucciones resolviendo referencias a objetos en el momento de la ejecución. Hay dos formas de trabajar con SQL dinámico:

- El paquete DBMS_SQL.
- SQL dinámico nativo (NDS), disponible sólo a partir de la versión 9i.

- El **tipo OBJECT** es una definición o molde de objeto. En un tipo OBJECT podemos definir atributos y métodos:

- Los **atributos** son las características que sirven para describir físicamente el objeto.
- Los **métodos** son las acciones que se pueden realizar con el objeto (procedimientos y funciones).



Actividades complementarias



1 Escribe un disparador de base de datos que permita auditar las operaciones de inserción o borrado de datos que se realicen en la tabla EMPLE según las siguientes especificaciones:

- Se creará desde SQL*Plus la tabla *auditaremple* con la columna *col1 VARCHAR2(200)*.
- Cuando se produzca cualquier manipulación, se insertará una fila en dicha tabla que contendrá: fecha y hora, número de empleado, apellido y la operación de actualización INSERCIÓN o BORRADO.

2 Escribe un trigger que permita auditar las modificaciones en la tabla EMPLEADOS, insertando los siguientes datos en la tabla *auditaremple*: fecha y hora, número de empleado, apellido, la operación de actualización MODIFICACIÓN y el valor anterior y el valor nuevo de cada columna modificada (sólo en las columnas modificadas).

3 Suponiendo que disponemos de la vista:

```
CREATE VIEW DEPARTAM AS
SELECT DEPART.DEPT_NO, DNOMBRE, LOC,
COUNT(EMP_NO) TOT_EMPLE
FROM EMPLE, DEPART
WHERE EMPLE.DEPT_NO (+) =
DEPART.DEPT_NO
GROUP BY DEPART.DEPT_NO, DNOMBRE, LOC;
```

Construye un disparador que permita realizar actualizaciones en la tabla *depart* a partir de la vista *departam*, de forma similar al ejemplo del trigger *t_ges_emplead*. Se contemplarán las siguientes operaciones:

- Insertar y borrar departamento.
- Modificar la localidad de un departamento.

4 Escribe un paquete para gestionar los departamentos. Se llamará *gest_depart* e incluirá, al menos, los siguientes subprogramas:

- *insertar_nuevo_depart*: inserta un departamento nuevo. Recibe el nombre y la localidad del nuevo departamento. Creará el nuevo departamento comprobando que el nombre no se duplique y le asignará como número de departamento la decena siguiente al último número de departamento utilizado.

- *borrar_depart*: borra un departamento. Recibirá dos números de departamento: el primero corresponde al departamento que queremos borrar y el segundo, al departamento al que pasarán los empleados del departamento que se va a eliminar. El procedimiento se encargará de realizar los cambios oportunos en los números de departamento de los empleados correspondientes.

- *modificar_loc_depart*: modifica la localidad del departamento. Recibirá el número del departamento que se modifica y la nueva localidad, y realizará el cambio solicitado.

- *visualizar_datos_depart*: visualizará los datos de un departamento cuyo número se pasará en la llamada. Además de los datos relativos al departamento, se visualizará el número de empleados que pertenecen actualmente al departamento.

- *visualizar_datos_depart*: versión sobrecargada del procedimiento anterior que, en lugar del número del departamento, recibirá el nombre del departamento. Realizará una llamada a la función *buscar_depart_por_nombre* que se indica en el apartado siguiente.

- *buscar_depart_por_nombre*: función local al paquete. Recibe el nombre de un departamento y devuelve el número del mismo.

5 Crea un procedimiento que permita consultar todos los datos de la tabla *depart* a partir de una condición que se indicará en la llamada al procedimiento.