Tema 12. PROGRAMACIÓN EN BASES DE DATOS II

Contenido

1 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).
- **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 (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).

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
- Auditar las actualizaciones e, incluso, enviar alertas
- Gestionar réplicas remotas de la tabla
- Etc.

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;
```

1.1 Elementos de un trigger

En los disparadores podemos distinguir los siguientes elementos:

- **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 trigger. 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 la condición. Es opcional.

- **Cuerpo del trigger**. Es el código que se ejecutará cuando se cumplan las condiciones especificadas en la cabecera. Se trata de un bloque PL/SQL.

2 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_comision, que se disparará después de cada modificación de la columna comision de la tabla EMPLE:

Primero crear una tabla llamada `auditaremple`, que contenga una columna `col1`, donde introduciremos mensajes.

```
CREATE TABLE  AUDITAREMPLE
    (COL1  VARCHAR2(200));
Tabla creada.
```

El siguiente trigger se disparará después de cada modificación de la columna comision de la tabla emple:

```
CREATE OR REPLACE  TRIGGER audit_subida_comision
  AFTER  UPDATE  OF  comision
  ON  emple
  for each row
BEGIN
 INSERT  INTO  auditaremple
 VALUES ('Subida comision empleado '|| :old.emp_no);
END;
```

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.

Se compila

Se ejecuta cualquier procedimiento realizado anteriormente que modifique el salario

```
EXECUTE SUBIR_COMISION(7521,100);
Procedimiento PL/SQL terminado con éxito.
```

Se comprueba que ha funcionado el trigger anterior

```
SELECT * FROM EMPLE; /* compruebo que se ha modificado la
                        comision */
SELECT * FROM AUDITAREMPLE; /* compruebo que ha saltado
                        el trigger al modificar la comision */
COL1
---------------------------------
Subida comision empleado 7521
```

El formato para la creación de disparadores de tablas es:

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

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.
  - 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.

Ejemplo: Construir 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:

- En primer lugar, se habrá creado la tabla *auditaremple* (ver pag. 3).
- Cuando se produzca cualquier manipulación, se insertará una fila en dicha tabla que contendrá:
  - Fecha y hora
  - Numero de empleado
  - Apellido
  - La operación de actualización (INSERCIÓN o BORRADO)

```
CREATE OR REPLACE TRIGGER trig_ins_del_emple
   BEFORE INSERT OR DELETE ON emple FOR EACH ROW
BEGIN
   IF DELETING THEN
         INSERT INTO auditaremple
   VALUES(TO_CHAR(SYSDATE,'DD/MM/YY*HH24:MI*')  ||
   :OLD.emp_no|| '*' || :OLD.apellido || '* BORRADO ');
   ELSIF INSERTING THEN
         INSERT INTO auditaremple
VALUES(TO_CHAR(SYSDATE,'DD/MM/YY*HH24:MI*') ||
         :NEW.emp_no   ||   '*'   ||   :NEW.apellido||'*
INSERCION ');
   END IF;
END;
```

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'
```

2.1 Valores NEW y OLD

Se puede hacer referencia a los valores anterior y posterior a una actualización a nivel de fila. Normalmente 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`.
- 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.

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, lo indicaremos en la cláusula `REFERENCING`, tal como aparece en el formato ampliado que se verá después.

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

2.2 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:
  1. Se ejecutan los disparadores `BEFORE ... FOR EACH ROW`.
  2. Se ejecuta la actualización de la fila (`INSERT, UPDATE` o `DELETE`). En este momento se bloquea la fila hasta que la transacción se confirme.
  3. Se ejecutan los disparadores `AFTER ... FOR EACH ROW`.
- Una vez realizada la actualización se ejecutarán los disparadores `AFTER ... FOR EACH STATEMENT`.

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 misma tabla, podemos optar por la utilización de un solo trigger con múltiples eventos de disparo, tal como se explica en el apartado siguiente.

2.3 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 trigger fue un comando `INSERT`
`DELETING`	Devuelve `TRUE` si el evento que disparó el trigger fue un comando `DELETE`
`UPDATING`	Devuelve `TRUE` si el evento que disparó el trigger fue un comando `UPDATE`
`UPDATING('nombrecolumna')`	Devuelve `TRUE` si el evento que disparó el trigger fue un comando `UPDATE` y la columna especificada ha sido actualizada

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;
```

2.4 Restricciones para la creación de triggers

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:

1. El bloque PL/SQL **no puede** contener sentencias de control de transacciones como COMMIT, ROLLBACK o SAVEPOINT.
2. Tampoco se pueden hacer llamadas a procedimientos que transgredan la restricción anterior.
3. **No** se pueden utilizar **comandos DDL**.
4. Un trigger **no puede contener instrucciones que consulten o modifiquen tablas mutantes**. Una tabla mutante es aquella que está modificada por una sentencia UPDATE, DELETE o INSERT en la misma sesión.
5. **No se pueden cambiar valores** de las columnas que sean **claves primarias, únicas o ajenas de tablas de restricción**. Una tabla de restricción es una tabla que debe ser consultada 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.

2.5 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]  TRIGGER  nombretrigger
   INSTEAD OF {DELETE|INSERT|UPDATE[OF <lista_columnas>]}
    [OR {DELETE|INSERT|UPDATE[OF <lista_columnas>}] …
ON  nombrevista
 [FOR  EACH  ROW]       [WHEN (condicion)]
/*  aquí comienza el bloque PL/SQL  */
[DECLARE
<declaraciones>]
BEGIN
<acciones>
[EXCEPTION
<gestión de excepciones>]
END;
```

Los disparadores de sustitución tienen las siguientes características:

- 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 se puede conseguir una funcionalidad similar utilizando estructuras alternativas dentro del bloque PL/SQL)

Ejemplo:

1º Creamos una vista con el número de empleado, el apellido, el oficio, el nombre del departamento y la localidad.

```
CREATE VIEW  empleado_vista AS
   SELECT emp_no,apellido, oficio, dnombre, loc
   FROM  emple,depart
   WHERE emple.dept_no = depart.dept_no;
Vista creada.
```

```
select * from empleado_vista;
```

Veremos:

```
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 de datos dan error:

```
INSERT INTO  empleado_vista VALUES
     (7999,'MARTINEZ','VENDEDOR','CONTABILIDAD','SEVILLA');

ERROR en línea 1:
ORA-01776: cannot modify more than one base table through a join view


UPDATE empleado_vista SET dnombre='CONTABILIDAD'
    WHERE apellido='SALA';

ERROR en línea 1:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

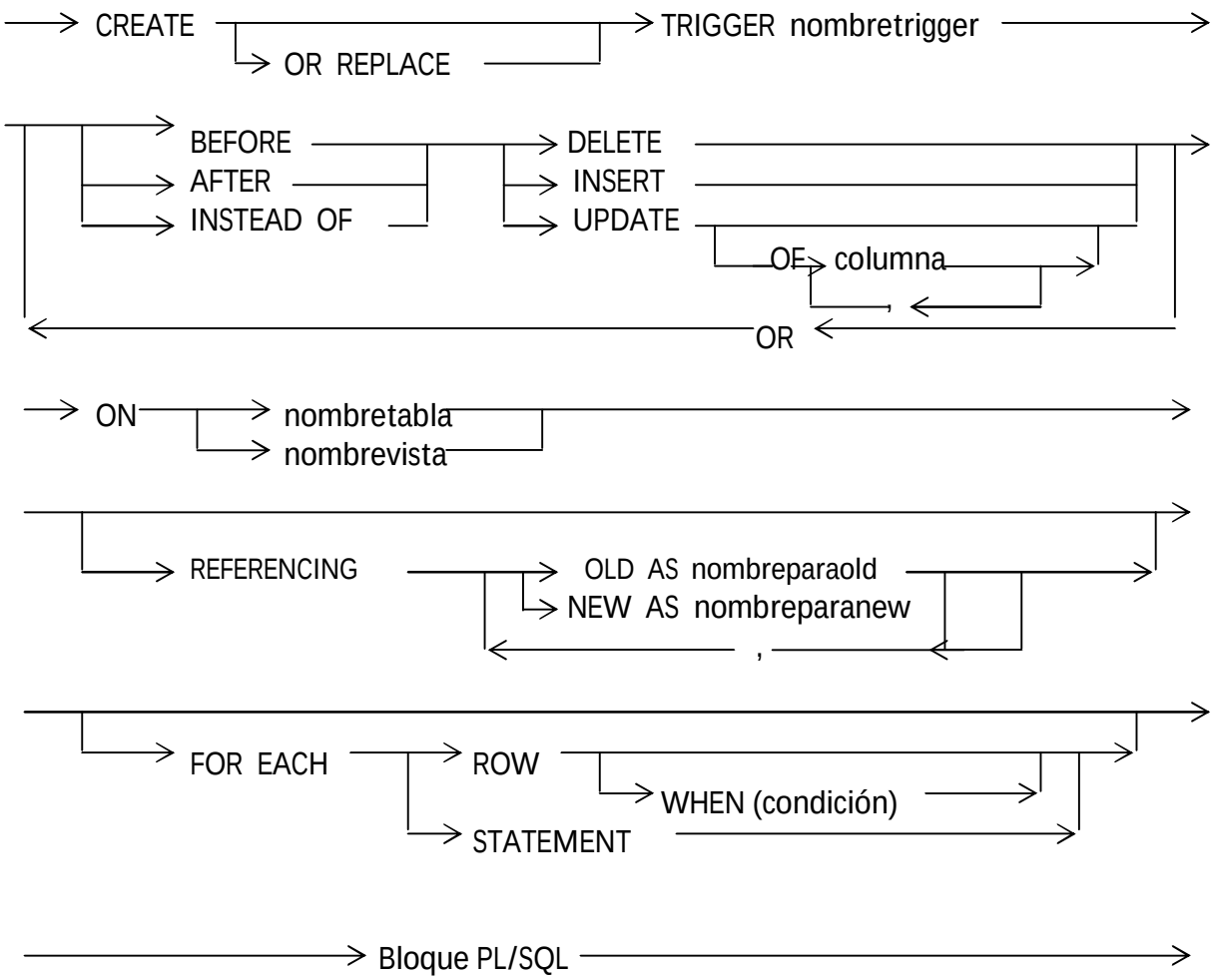
Para facilitar estas operaciones de manipulación se creará el siguiente disparador de sustitución:

```
CREATE OR REPLACE  TRIGGER  t_ges_empleado_vista
    INSTEAD OF  DELETE  OR  INSERT OR UPDATE
    ON empleado_vista
    FOR  EACH  ROW
DECLARE
    v_dept  depart.dept_no%TYPE;
BEGIN
 IF  DELETING  THEN       /* si se pretende  borrar una fila */
    DELETE  FROM  emple            /* la borra de empleado */
   WHERE emp_no = :old.emp_no; /* old porque es DELETE */
 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 se puede cambiar a un empleado de departamento indicando solamente 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 contemple entre las alternativas, el trigger levantará un error en la aplicación haciendo que falle toda la actualización.

2.6 Formato ampliado

Con las nuevas opciones y características estudiadas, podemos ampliar el formato básico inicial de creación de triggers al siguiente:



2.7 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*, ya que cuando un *trigger* se dispara, trabaja con los privilegios del propietario del *trigger*, no con los del usuario actual.
- Con el nombre del *trigger* se puede especificar el esquema en el que queremos crear el *trigger* 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 *triggers* en cualquier esquema.

- Normalmente un *trigger* se asocia a una tabla, pero también puede asociarse a una vista. Asimismo, la tabla o vista 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 etc.

2.8 Acciones a realizar en un trigger

- **Activarlo**: ALTER TRIGGER nombretrigger ENABLE.
- **Desactivarlo:** ALTER TRIGGER nombretrigger DISABLE.
- **Compilar:** ALTER TRIGGER nombretrigger COMPILE.
- **Eliminarlo:** DROP TRIGGER nombretrigger.

2.9 Vistas con información sobre los triggers

Las vistas **dba_triggers** y **user_triggers** contienen toda la información sobre los triggers.

DESC SYS.USER_TRIGGERS;

Resultado de la consulta:

Name	Null?	Type
TRIGGER_NAME	NOT NULL	VARCHAR2(30)
TRIGGER_TYPE		VARCHAR2(16)
TRIGGERING_EVENT		VARCHAR2(26)
TABLE_OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
REFERENCING_NAMES		VARCHAR2(87)
WHEN_CLAUSE		VARCHAR2(4000)
STATUS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
TRIGGER_BODY		LONG

```
select trigger_name,table_name, triggering_event, trigger_type,status
 from user_triggers;

TRIGGER_NAME              TABLE_NAME              TRIGGERING_EVENT
TRIGGER_TYPE              STATUS

------------------------- ------------------------- ----------------------------------------
---------------------------------------------------------------

T_GES_EMPLEADO_VISTA      EMPLEADO_VISTA          INSERT OR
UPDATE OR DELETE          INSTEAD OF    ENABLED


TRIG_INS_DEL_EMPLE        EMPLE                   INSERT OR DELETE
            BEFORE EACH ROW  ENABLED

T AUDIT_SUBIDA_COMISION   EMPLE     UPDATE   AFTER EACH ROW
ENABLED
```

3 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, etc.) o una instrucción de definición de datos (creación, modificación de un objeto). Para crear *triggers* del sistema hay que tener el privilegio `ADMINISTER DATABASE TRIGGER`.

```
CREATE [OR REPLACE ] TRIGER nombretrigger
{BEFORE | AFTER} {<lista de eventos definición> |<lista eventos del sistema>}
ON {DATABASE |SCHEMA}  {WHEN {condicion}]
<cuerpo del trigger (bloque pl/sql)>
```

- 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
- 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

Eventos del sistema y eventos de definición

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
DDL	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

Pl/sql dispone de algunas funciones que permiten acceder a los atributos del evento del disparo ORA_SYSEVENT, ORA_LOGIN_USER, ORA_DICT_OBJ_TYPE, etc

- 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 se puede comprobar el tipo y el nombre del objeto que se está modificando (y el ID o nombre de usuario).

Ejemplo:

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 del usuario (USER), la fecha y hora en la que se produce el evento de conexión, y la operación CONEXIÓN que realiza en usuario.

Creamos la tabla control_conexiones:

```
CREATE TABLE control_conexiones (usuario VARCHAR2(20),
momento TIMESTAMP, evento varchar2(20));
```

Ahora creamos el trigger:

```
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 crear este trigger 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.

4 Registros y colecciones

Tanto las tablas como los registros son estructuras de datos compuestas de otras más simples.

4.1 Registros en PL/SQL

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

Hasta ahora hemos utilizado el atributo ROWTYPE para crear una estructura de datos idéntica a la fila de una tabla. Esta estructura de datos es un *registro*. Esta forma de crear registros es muy sencilla y rápida pero tiene algunas limitaciones:

- Tanto los nombre 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 o 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 que definamos nosotros el registro y declarar las variables del tipo que necesitemos en los siguientes pasos:

1. **Se define el tipo** genérico del registro

```
TYPE  nombre_tipo  IS RECORD
  (nombre_campo1Tipo_campo1     [ [NOT  NULL]{:= | DEFAULT}
      valorinicial1]
   nombre_campo2Tipo_campo2     [ [NOT  NULL]{:= | DEFAULT}
      valorinicial2]
     ……);
```

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

Los campos que tengan el especificador NOT NULL 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;
```

Ejemplo:

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
```

Debemos tener cuidado con expresiones del tipo

nom_var_reg1 := nom_var_reg2

Ya que solamente funcionará 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, localidad`. Posteriormente, se podrá hacer referencia a estos campos utilizando la notación de punto, tal como aparece en los ejemplos:

```
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;
```

```
DECLARE
TYPE t_registro  IS RECORD(
      tabla CHAR(10),
      fecha DATE,
      contador    NUMBER);
 vreg t_registro;

BEGIN
  vreg.tabla:='EMPLE';
  vreg.fecha:=SYSDATE;
  SELECT  COUNT(*) INTO vreg.contador
  FROM  emple;
  DBMS_OUTPUT.PUT_LINE(vreg.tabla || ' ** ' || vreg.fecha || ' ** ' ||
                       vreg.contador);
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.

4.2 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.

4.2.1 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];
```

   - `nombre_tipovarray` es un especificador que indica el tipo base de la tabla y que posteriormente se utilizará para definir tablas.
   - `tipo_elementos` 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:

```
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);
     ```

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:

`nobrevariablevarray(i).nombrecampo`

4.2.2 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:

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_tipoTablaAnidada ();`

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;
```

Las tablas anidadas y los VARRAYS permiten su uso en tablas de la base de datos y su manejo mediante instrucciones SQL. Aunque no los veremos porque exceden los objetivos del curso.

4.2.3 Tablas indexadas (o arrays asociativos)

Este tipo de colección sólo permite su uso desde PL/SQL.

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. Los elementos de una tabla indexada no tienen que ser necesariamente contiguos. 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)];
```

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) FROM emple
    WHERE emp_no = 7900;
  …
  DBMS_OUTPUT.PUT_LINE(tab_emple(7900).apellido);
  …
  tab_emple(7900).salario := 3500;
…
```

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 siguientete 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;
...
```

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.

4.2.4 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:

`variabledetabla.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`

- 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
```

5 Paquetes

Los paquetes se utilizan para guardar subprogramas y otros objetos en la base de datos.

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

5.1 Elementos de un paquete

En los paquetes nos encontramos dos elementos claramente diferenciados:

- **Especificación**. Contiene declaraciones públicas (accesibles desde cualquier parte de la aplicación) de subprogramas, tipos, constantes, variables, cursores, excepciones, etc. Los objetos declarados en la especificación son accesibles también desde fuera del paquete Actúa como una interfaz con otros programas
- **Cuerpo**. Contiene los detalles de implementación de todos los objetos del paquete (el código de los programas, etcétera). Y también puede incluir declaraciones privadas accesibles solamente desde los objetos del paquete. Es una caja negra para los demás programas

5.2 Creación de un paquete

Tanto la especificación como el cuerpo se pueden crear desde SQL*PLUS mediante los comandos:

```
CREATE [OR REPLACE] PACKAGE
```

```
CREATE [OR REPLACE] PACKAGE BODY
```

El formato genérico es:

- **Creación de cabecera o especificación:**

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

- **Creación del cuerpo del paquete**

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

Se puede codificar y compilar la especificación del paquete independientemente del cuerpo

El bloque anónimo que se incluye, opcionalmente, al final del paquete se ejecutará la primera vez que se haga referencia, en la sesión, a alguno de los objetos incluidos en el mismo. Se suele utilizar para realizar operaciones de inicialización

Ejemplo:

- En primer lugar se crea la **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;
```

Podemos observar 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.

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 escribiremos:

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

- **Creación del cuerpo del paquete y declaración de objetos locales**

Una vez creada la cabecera procederemos a crear 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, etc. Estos objetos serán accesibles desde cualquier parte del paquete, pero no desde fuera

/* cuerpo del paquete */

```
CREATE OR REPLACE PACKAGE BODY buscar_emple
AS
   vg_emple   t_reg_emple;  /* variable local al paquete
*/
   PROCEDURE ver_emple;  /* declaracion del procedimiento
local al paquete, se coloca porque los procedimientos
llaman a esta procedure que se encuentra fisícamente detrás
*/
   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;  /* llamada al procedimiento ver_emple  */
   END ver_por_numero;
   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
    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;
```

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.

5.3 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 se utilizará el formato:

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

Por supuesto, solamente se podrán ejecutar desde fuera los subprogramas que hayan sido declarados en la especificación

```
execute buscar_emple.ver_por_apellido('SALA');
7521 * SALA * VENDEDOR * 178750 * 30
Procedimiento PL/SQL terminado con éxito.
```

```
EXECUTE buscar_emple.ver_por_numero(7902);
7902 * FERNANDEZ * ANALISTA * 429000 * 20
Procedimiento PL/SQL terminado con éxito.
```

```
DECLARE
  vr_emple  buscar_emple.t_reg_emple;
BEGIN
  vr_emple:= buscar_emple.datos(7902);
  dbms_output.put_line(vr_emple.apellido
||'*'||vr_emple.salario||'*'||vr_emple.oficio);
END;
```

Se pueden utilizar los subprogramas declarados en la especificación desde otros subprogramas que se encuentran fuera del paquete.

5.4 Declaración de cursores en paquetes

Para declarar cursores en paquetes de forma que estén accesibles en la especificación 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 indicando el ***nombre del cursor, los parámetros (si procede) y el tipo devuelto***. Este último se indicará mediante `RETURN tipodedato;` para cursores 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 >100000;
   …
END empleados_acc;
```

5.5 Ámbito y otras características de las declaraciones

Los objetos declarados en la especificación del paquete son globales al paquete y locales al contexto. Asimismo 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.

Podemos destacar lo siguiente:

- 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.

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

5.6 Características de almacenamiento y compilación

Tanto el código fuente como el código compilado de los paquetes se almacenan 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*, y 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 recompilar 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 recompilar 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.

5.7 Paquetes suministrados por Oracle

Oracle incluye con su gestor de base de datos diversos paquetes como **STANDARD, DBMS_OUTPUT, DBMS_STANDARD, DBMS_SQL** y muchos otros que incorporan diversas funcionalidades.

- En **STANDARD** se declaran tipos, excepciones, funciones etc, 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;
```

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(..)…

- **DBMS_STANDARD** incluye utilidades, como el procedimiento RAISE_APLICATION_ERROR, que facilitan la interacción de nuestras aplicaciones con Oracle.
- En **DBMS_OUTPUT** se encuentra el procedimiento PUT_LINE que hemos visto para visualizar datos, y otros como ENABLE y DISABLE que permiten figurar y purgar el buffer utilizado por PUT_LINE.
- **DBMS_SQL** incorpora procedimientos y funciones que permiten utilizar SQL dinámico en nuestros programas.