Guía de Disparadores

Un disparador (o *trigger*) es un procedimiento almacenado asociado a una tabla que se ejecuta al realizar una operación "básica" (INSERT, un DELETE o un UPDATE) sobre ésta. La operación básica que despierta al *trigger* es conocida como sentencia disparadora. La ejecución del disparador puede ser antes o después de llevar a cabo la sentencia disparadora. Es posible especificar condiciones adicionales para la ejecución del disparador (restrictores). Dado que una sentencia disparadora puede afectar una o más filas de una tabla, es necesario especificar si se quiere que el disparador se ejecute para cada una de las filas afectadas o para el bloque en general.

Diseño de disparadores

Los disparadores pueden ser utilizados para cumplir con alguna de las siguientes tareas:

- Evitar la ejecución de transacciones inválidas
- Garantizar el cumplimiento de restricciones de integridad
- Garantizar el cumplimiento de reglas del negocio
- Generar, automáticamente, valores de columnas derivadas

Cuando se diseñan disparadores es necesario tomar en cuenta las siguientes consideraciones:

- El disparador ***no debe ser utilizado para garantizar el cumplimiento de restricciones de integridad que puedan ser definidas a nivel de esquema***. Por ejemplo, no tiene sentido implementar un disparador para verificar que al insertar una tupla en la tabla Empleado que su tipo debe ser 'A', si es administrativo, 'O', si es obrero o 'D', si es docente. Esta restricción puede garantizarse al definir el atributo tipo_empleado de la tabla Empleado. La manera de hacerlo es colocando la restricción CHECK (tipo_empleado IN ('A','O','D'))
- Hay que ***evitar*** crear ***disparadores recursivos***. Por ejemplo, el crear un disparador que se active después de actualizar la tabla Empleado, que a su vez realiza una actualización de la tabla Empleado, provoca una ejecución recursiva del disparador que agota la memoria.
- Dado que ***los disparadores son compilados la primera vez que se activan***, se recomienda que la cantidad de instrucciones de un disparador no sea muy grande (máximo 60 líneas). De esta manera, el efecto que tiene la primera ejecución sobre el rendimiento del sistema será menor. Si un *trigger* tiene demasiadas líneas es preferible incluir el código de éste en un ***procedimiento almacenado*** (que se almacena ya compilado). De esta forma, el *trigger* puede llamar al procedimiento, reduciendo así el tiempo de compilación al momento de ejecución.

Para diseñar un *trigger* es necesario identificar cada uno de los elementos definidos para él (sentencia disparadora, etc). A continuación se presenta un ejemplo de un disparador implementado para garantizar una restricción de integridad. Suponga que se tienen las siguientes relaciones, asociadas a la concesión de préstamos en un banco:

Prestatario(ci, nombre, dir, tel, empresa, tel_ofic) almacena todos los prestatarios actuales del banco.
Préstamo(num_prest, ci, tasa, monto) almacena todos los préstamos que aun no han sido cancelados. El atributo ci referencia a Prestatario y es candidato a clave pues el banco no otorga simultáneamente dos préstamos a la misma persona.
Cuota(num_prest, num_cuota, f_venc, f_pago) almacena todas las cuotas de los préstamos actuales (tanto las pagadas como las pendientes). El atributo num_prest referencia a Préstamo. Se tiene como política que toda cuota debe ser pagada antes de su fecha de vencimiento.

Al implantar el esquema sobre el RDBMS ORACLE se obtiene lo siguiente:

```
CREATE TABLE Prestatario (
ci VARCHAR(8) NOT NULL PRIMARY KEY,
nombre VARCHAR(50) NOT NULL,
dir VARCHAR(100) NOT NULL,
tel VARCHAR(10) NOT NULL,
empresa VARCHAR(100) NOT NULL,
tel_ofic VARCHAR(10) NOT NULL);

CREATE TABLE Prestamo (
num_prest NUMBER(5) NOT NULL PRIMARY KEY,
ci VARCHAR(8)UNIQUE REFERENCES Prestatario(ci),
tasa NUMBER(4,2) NOT NULL,
monto NUMBER(8)NOT NULL CHECK(monto > 0));
```

```
CREATE TABLE Cuota (
num_prest NUMBER(5) NOT NULL,
num_cuota NUMBER(2) NOT NULL,
f_venc DATE NOT NULL,
f_pago DATE,
CONSTRAINT pk_cuota
PRIMARY KEY (num_prest, num_cuota),
CONSTRAINT fk_cuota_prest
num_prest REFERENCES Prestamo(num_prest));
```

En la creación de las tablas se incluyeron todas las restricciones, excepto aquella que dice que toda cuota debe ser pagada antes de su fecha de vencimiento. A continuación se presenta el diseño del disparador que garantiza el cumplimiento de esta restricción:

- Sentencia Disparadora: Como en la BD están todas las cuotas (pagadas o no) asociadas a los préstamos, la fecha de pago se actualiza. Por lo tanto, la sentencia disparadora es la *actualización de fecha de pago en la tabla Cuota*.
- Antes / Después: La restricción de integridad no se puede violar, por lo tanto el *trigger* debe ser disparado *antes* de realizar la actualización.
- Para Todas/Para el bloque: La verificación de la restricción se hace *para todas las filas* que se actualicen al ejecutar la sentencia disparadora.
- Restrictor: Se debe impedir la actualización, *sólo cuando la fecha de pago sea mayor que la fecha de vencimiento de la cuota*.
- Acción: *Dar un error* por violación de la restricción.

Creación de Disparadores

Los disparadores se crean utilizando el comando CREATE TRIGGER. Este comando puede utilizarse con cualquier herramienta interactiva (como SQL*Plus o SQL*DBA). Cuando se usan estas herramientas interactivas, se indica que se terminó de especificar la instrucción CREATE TRIGGER, colocando un *slash* ("/") en la última línea.

A continuación se especifica el *trigger* asociado con la tabla cuota:

```
CREATE TRIGGER BUpCUOTA
BEFORE UPDATE OF f_pago ON Cuota
FOR EACH ROW
WHEN (new.f_pago > old.f_venc)
BEGIN
        raise_application_error(-20000, 'Cuota ' || TO_CHAR(:old.num_cuota) || ' del prestamo ' ||
        TO_CHAR(:old.num_prest) || ' vencida. Por favor, dirigirse a la gerencia.');
END;
/
```

A continuación se presentan algunos **aspectos generales** con respecto a la definición de *triggers*:

- Los **nombres** de los *triggers* deben ser **únicos** dentro de un esquema dado. Los nombres no tienen por qué ser únicos con respecto a otros objetos del esquema (por ejemplo, tablas, vistas, etc.). Sin embargo, se recomienda usar nombres distintos para evitar confusiones.
- Alguna de las dos, BEFORE o AFTER, debe ser utilizada en el CREATE TRIGGER. De esta manera se especifica exactamente cuando se despierta el disparador, en relación con la ejecución de la sentencia activadora. La opción BEFORE o AFTER se especifica justo antes de especificar la sentencia activadora. El trigger BUpCUOTA es un before trigger.
- En algunos casos da lo mismo si el trigger se ejecuta antes o después de realizar la sentencia activadora. En estos casos, un after trigger es ligeramente más eficiente que un before trigger, pues con estos últimos los registros de datos afectados son leídos (lógicamente) dos veces: una para el disparador y otra para la sentencia disparadora. Con los after triggers se leen los registros de datos sólo una vez.
- La sentencia activadora especifica el tipo de operación que despierta el disparador (DELETE, INSERT o UPDATE). Una, dos o incluso las tres operaciones pueden ser incluidas en la especificación de la sentencia activadora.
- En la sentencia activadora se especifica la tabla asociada al trigger. Puede especificarse exactamente una tabla (no una vista) en la sentencia activadora.
- Si la sentencia activadora especifica un UPDATE se puede incluir una lista de columnas en dicha sentencia. Si se incluye la lista de columnas, el trigger se activa por un UPDATE sólo si una de las columnas especificadas es actualizada. Si se omite la lista, el trigger se activa cuando cualquier columna de la tabla se actualiza. No se puede especificar lista de columnas para INSERT o DELETE.
- La presencia o ausencia de la opción **FOR EACH ROW** determina si el disparador es a **nivel de filas (row trigger)** o a **nivel de sentencia activadora (statement trigger).** Si está presente, esta opción especifica que el cuerpo del trigger se ejecuta individualmente para cada una de las filas de la tabla que haya sido afectada por la sentencia activadora. La ausencia de la opción FOR EACH ROW implica que el trigger se ejecuta una sola vez, para la ejecución de una sentencia activadora.
- Opcionalmente, se pueden incluir ***restricciones en la definición de un row trigger***. Para ello se especifica, en una **cláusula WHEN**, una expresión booleana de SQL. Si se incluye una cláusula WHEN, la expresión se evalúa para cada una de las filas que el disparador afecta. Si el resultado de la evaluación es TRUE, se ejecuta el cuerpo del trigger sobre la fila que hizo cierta la expresión. Si el resultado de la evaluación es FALSE o NOT TRUE (desconocido, como con los valores nulos) para una fila dada, el cuerpo del trigger no se ejecuta para dicha fila. La expresión en una cláusula WHEN **no puede incluir subqueries**.

- Sólo se puede definir un *trigger* de cada tipo por tabla. Esto da doce posibilidades:

BEFORE UPDATE row	AFTER UPDATE row
BEFORE DELETE row	AFTER DELETE row
BEFORE INSERT row	AFTER INSERT row
BEFORE UPDATE statement	AFTER UPDATE statement
BEFORE DELETE statement	AFTER DELETE statement
BEFORE INSERT statement	AFTER INSERT statement

- Cada tabla puede tener hasta cuatro UPDATE triggers (BEFORE/AFTER, statement/row), sin importar si los *triggers* son disparados sólo cuando se actualizan algunas columnas de la tabla.
- Orden en la Ejecución de Disparadores. Una tabla puede tener distintos Tipos de Disparadores asociados a una misma **orden DML**.:

1. Ejecutar, si existe, el disparador ipo BEFORE a nivel de sentencia.
2. Para cada fila a la que afecte la orden: (esto es como un bucle, para cada fila)
   a) Ejecutar, si existe, el disparador BEFORE a nivel de fila, sólo si dicha fila cumple la condición de la cláusula WHEN (si existe).
   b) Ejecutar la propia orden.
   c) Ejecutar, si existe, el disparador AFTER a nivel de fila, sólo si dicha fila cumple la condición de la cláusula WHEN (si existe).
3. Ejecutar, si existe, el disparador tipo AFTER a nivel de orden.

CUERPO:

- El ***cuerpo*** de un trigger es un ***bloque de PL/SQL que puede incluir instrucciones de PL/SQL y de SQL***. Este bloque de instrucciones se realiza si se ejecuta la sentencia activadora especificada para el trigger y, si existe una cláusula WHEN ésta es TRUE. La instrucción CREATE TRIGGER falla si hay algún error en el bloque PL/SQL.
- Si un trigger puede ser activado por más de un tipo de operación (por ejemplo, "INSERT OR DELETE OR UPDATE OF Cuota"), el cuerpo del trigger puede utilizar los predicados condicionales INSERTING, DELETING y UPDATING para ejecutar bloques específicos de código, dependiendo del tipo de operación que activó el disparador. Por ejemplo, si se tiene INSERT OR UPDATE ON Cuota

  dentro del código del *trigger* se pueden incluir las siguientes condiciones:

  ```
  IF INSERTING THEN . . . END IF;
  IF UPDATING THEN . . . END IF;
  ```

  la primera condición es cierta sólo si la operación que disparó el *trigger* es un INSERT. La segunda condición es cierta sólo si la operación que disparó el *trigger* es un UPDATE.

  - En un UPDATE, se puede especificar el nombre de una columna en un predicado condicional UPDATING para determinar si la columna especificada ha sido actualizada. Por ejemplo:

    ```
    CREATE TRIGGER . . .
    . . . UPDATE OF sal, comm ON emp . . .
    BEGIN
    . . . IF UPDATING ('sal') THEN . . . END IF;
    END;
    ```

    La siguiente instrucción dispara el trigger anterior y hace que el predicado condicional the UPDATING (sal) se evalúe comoTRUE: UPDATE emp SET sal = sal + 100;
- El cuerpo de un ***row trigger*** puede incluir ciertos elementos especiales: **nombres de correlación** y los predicados condicionales **INSERTING, DELETING y UPDATING**.
  - En el cuerpo de un disparador, el código PL/SQL y las instrucciones de SQL tienen acceso a los valores nuevos y viejos de las columnas de la fila actual afectada por la sentencia activadora. Existen dos nombres de correlación para cada columna de la tabla que está siendo modificada: uno para el valor viejo y otro para el valor nuevo. Los ***valores nuevos*** son referenciados utilizando **":new."** antes del nombre de la columna, mientras que los ***viejos*** utilizan **":old."** (los ":" son necesarios dentro del bloque de PL/SQL para indicar que es un valor que viene de afuera de la expresión SQL).
  - En el ejemplo se usa ":new.num_prest" en el cuerpo del trigger. La expresión en una cláusula WHEN de un row trigger puede incluir nombres de correlación. Note que los calificadores se usan sin los ":".
  - Dependiendo del tipo de sentencia activadora, algunos nombres de correlación pueden no tener sentido. En un trigger activado por un INSERT sólo tiene sentido hablar de los valores nuevos. En un trigger activado por un UPDATE tiene sentido hablar de los valores nuevos y los viejos tanto para los before triggers como para los after triggers. En un trigger activado por un DELETE, sólo tiene sentido hablar de los valores viejos.
  - Si un before row trigger cambia el valor de algún ":new.columna", un after row trigger para la misma sentencia activadora verá el cambio realizado por el before row trigger.

- Si se produce una **excepción** o **condición de error (predefinida o definida por el usuario)** durante la ejecución del cuerpo del disparador, se hace ROLLBACK de todos los efectos del cuerpo del *trigger* y de la sentencia activadora (**a menos que se haga un manejo específico de la excepción**). Por lo tanto, el cuerpo de un *trigger* puede evitar la ejecución de una sentencia activadora produciendo una excepción.
- El cuerpo de un *trigger* puede incluir ***cualquier instrucción del DML SQL***, incluyendo SELECT (que debe ser un SELECT-INTO o un SELECT en la definición de un cursor), INSERT, UPDATE y DELETE. **No se permiten instrucciones del DDL ni instrucciones para el control de transacciones (ni ROLLBACK, ni COMMIT ni SAVEPOINT).**
- Una instrucción SQL dentro de un *trigger* puede insertar datos en una columna de tipo LONG o LONG RAW. Sin embargo, no se pueden declarar variables de estos tipos en el cuerpo del disparador. Además, ni :NEW ni :OLD pueden ser usados con columnas de tipo LONG o LONG RAW.
- No se deben crear *triggers* que dependan del orden en el cual las filas sean procesadas. Recuerde que una BD relacional no garantiza el orden de las filas al ser procesadas por una instrucción SQL.
- Cuando una instrucción de un *trigger* produce que se dispare otro *trigger*, se dice que estos están "en cascada". ***ORACLE permite hasta 32 triggers en cascada***.

TABLAS MUTANTES

- Una tabla que está "**mutando**" es una tabla que **está siendo modificada por un INSERT, un DELETE o un UPDATE o una tabla que podría modificarse por los efectos de un ON DELETE CASCADE (integridad referencial)**. Una tabla está "restringiendo" (tabla de restricción o tabla padre) si una sentencia activadora podría necesitar leerla directamente, por una sentencia SQL, o indirectamente, por una restricción de integridad referencial. Una tabla no se considera ni mutando ni restringiendo para los *triggers*. Sin embargo, hay **dos restricciones importantes con respecto a las tablas que mutan o que restringen:**
  1. Las instrucciones de ***un row trigger no pueden ni leer ni modificar una tabla que está mutando.***
  2. Las instrucciones de ***un row trigger no pueden cambiar ni la clave primaria, ni claves foráneas, ni atributos únicos de tablas que estén restringiendo***. Esta restricción tiene una **excepción**: un ***before row trigger disparado por un INSERT de una sola fila de una tabla con una clave foránea***, puede modificar cualquier columna de la tabla primaria siempre que no se viole ninguna de las restricciones de integridad referencial.
- Los ERRORES por Tablas Mutantes se detectan y se generan en Tiempo de Ejecución y no de Compilación (ORA-4091).
- Tablas Mutantes, Disparador de Ejemplo:
  - Disparador que modifique el número de empleados de un departamento (columna Departamentos.Num_Emp) cada vez que sea necesario. Ese número cambia al INSERTAR o BORRAR uno o más empleados, y al MODIFICAR la columna Dpto de la tabla Empleados, para uno o varios empleados. La tabla Departamentos es una tabla de restricción de la tabla Empleados, pero el Disparador es **correcto**, porque modifica Num_Emp, que no es la clave primaria. Este disparador no puede consultar la tabla Empleados, ya que esa tabla es mutante (por ejemplo, esta instrucción no sería válida: SELECT COUNT(*) INTO T FROM Empleados WHERE Dpto = :new.Dpto;)

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

- El disparador anterior Cuenta_Empleados tiene un inconvenientes: modifica el campo Num_Emp aunque este no tenga el valor correcto.

- Una solución al problema de la Tabla Mutante: Las tablas mutantes surgen básicamente en los disparadores a nivel de Fila. Como en estos no puede accederse a las tablas mutantes, la solución es crear dos disparadores:
  - A Nivel de Fila: En este guardamos los valores importantes en la operación, pero no accedemos a tablas mutantes. Estos valores pueden guardarse en:
    - Tablas de la BD especialmente creadas para esta operación.
    - Variables o tablas PL/SQL de un paquete: Como cada sesión obtiene su propia instancia de estas variables no tendremos que preocuparnos de si hay actualizaciones simultáneas en distintas sesiones.
  - A Nivel de sentencia AFTER: Utiliza los valores guardados en el disparador a Nivel de Fila para acceder a las tablas que ya no son mutantes.

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

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

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

Modificando disparadores

Un *trigger* no puede ser alterado explícitamente. Debe ser reemplazado por una nueva definición. Cuando se reemplaza un disparador se debe incluir la opción OR REPLACE en la instrucción CREATE TRIGGER. Por ejemplo, **CREATE OR REPLACE TRIGGER** BUpCUOTA . . .

Otra alternativa es eliminar el *trigger* y volverlo a crear. Por ejemplo,

```
DROP TRIGGER BUpCUOTA;
CREATE TRIGGER BUpCUOTA . . .
```

Los *triggers* pueden estar **habilitados o deshabilitados**. Los *triggers* están habilitados por *default*. Los *triggers* pueden deshabilitarse si:

- Un objeto al que hace referencia no está disponible.
- Se tienen que realizar cargas masivas de datos y se quiere proceder sin disparar *triggers*.
- Se están volviendo a cargar datos.
- Cualquier otro caso en el que se considere necesario.

Para deshabilitar un *trigger* se utiliza el comando ALTER TRIGGER con la opción DISABLE. Por ejemplo:

```
ALTER TRIGGER BUpCuota DISABLE;
```

Es posible deshabilitar todos los *triggers* asociados a una tabla utilizando la opción ALL TRIGGERS con el ALTER TABLE. Por ejemplo,

```
ALTER TABLE Cuota
DISABLE ALL TRIGGERS;
```

Para volver a habilitar un *trigger*, se utiliza el comando ALTER TRIGGER con la opción ENABLE. P.e.,

```
ALTER TRIGGER BupCuota ENABLE;
```

Para habilitar todos los *triggers* asociados a una tabla se utiliza la opción ALL TRIGGERS con el ALTER TABLE. Por ejemplo,

```
ALTER TABLE Cuota
ENABLE ALL TRIGGERS;
```

Ejemplos. A continuación se presentan algunos ejemplos de *triggers*:

- Para garantizar restricciones de integridad

```
CREATE TRIGGER salary_check
BEFORE INSERT OR UPDATE OF sal, job_classification ON emp FOR EACH ROW
DECLARE
  minsal               NUMBER;
  maxsal               NUMBER;
  salary_out_of_range   EXCEPTION;
BEGIN
/* Retrieve the minimum and maximum salary for the employee's new job classification from the
SALGRADE  table into MINSAL and MAXSAL. */

SELECT minsal, maxsal INTO minsal, maxsal FROM salgrade
  WHERE job_classification = :new.job_classification;

/* If the employee's new salary is less than or greater  than the job classification's limits, the exception is
  raised.  The exception message is returned and the  pending INSERT or UPDATE statement that fired
the  trigger is rolled back. */
  IF (:new.sal < minsal OR :new.sal > maxsal) THEN
    RAISE salary_out_of_range;
  END IF;
EXCEPTION
  WHEN salary_out_of_range THEN
    raise_application_error (-20300, 'Salary '||TO_CHAR(:new.sal)||' out of range for '
      ||'job classification '||:new.job_classification ||' for employee '||:new.name);
  WHEN NO_DATA_FOUND THEN
    raise_application_error(-20322, 'Invalid Job Classification ' ||:new.job_classification);
END;
```

Reglas de negocio:

Almacen(*CodPieza*,CantDisp,CantLim,CantPedido), **Pedidos**(*CodPieza*,CantPedido,Fecha)

```
CREATE TRIGGER TrigPedido
AFTER UPDATE OF CantDisp ON Almacen FOR EACH ROW
WHEN (NEW.CantDisp < NEW.CantLim)
DECLARE     x number;
BEGIN
          SELECT COUNT(*) INTO x FROM Pedidos WHERE CodPieza = :NEW.CodPieza;
          IF x = 0 THEN
                    INSERT INTO Pedidos VALUES (:NEW.CodPieza, :NEW.CantPedido, SYSDATE);
          END IF;
END;
```

- Para generar valores de columnas derivadas:

```
BEFORE INSERT OR UPDATE OF ename ON emp
/* Before updating the ENAME field, derive the values for  the UPPERNAME and SOUNDEXNAME fields.
Users should be  restricted from updating these fields directly. */
FOR EACH ROW
BEGIN
  :new.uppername := UPPER(:new.ename);
  :new.soundexname := SOUNDEX(:new.ename);
END;
```

RAISE_APPLICATION_ERROR permite que un programa PL/SQL pueda generar errores tal y como lo hace Oracle:

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

• Hay Excepciones Predefinidas que controlan errores particulares (excepto OTHERS que controla cualquier tipo de error). Algunas son:

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

Otros ejemplos:

```
CREATE VIEW ClientesTodos(Codcli,Direccion,Credito,Provincia) AS
SELECT Codcli,Direccion,Credito,'Castellon'
FROM ClientesCS
UNION
SELECT Codcli,Direccion,Credito,'Valencia'
FROM ClientesVC;

CREATE TRIGGER TrigInsClientesTodos
INSTEAD OF INSERT ON ClientesTodos
FOR EACH ROW
begin
        if (:NEW.Provincia = 'Castellon') then
        INSERT INTO ClientesCS
        VALUES(:NEW.Codcli,:NEW.Direccion,:NEW.Credito)
        else
        INSERT INTO ClientesVC
        VALUES(:NEW.Codcli,:NEW.Direccion,:NEW.Credito)
        end if;
end;
```

Ejemplo de tabla mutante.

```
CREATE OR REPLACE TRIGGER chequear_salario
BEFORE INSERT OR UPDATE ON empleado
FOR EACH ROW
WHEN (new.trabajo<>'presidente')
DECLARE
v_salariomin empleado.salario%TYPE;
v_salariomax empleado.salario%TYPE;
BEGIN
        SELECT MAX(salario), MIN(salario) FROM empleado
        INTO v_salariomin, v_salariomax
        FROM empleado
        WHERE trabajo=:new.trabajo;
        IF :new.salario<v_salariomin OR :new.salario> v_salariomax THEN
        RAISE_APPLICATION_ERROR(-20001, 'Sueldo fuera de rango');
        END IF;
END chequear_salario;
/
```

Al ejecutar la orden

```
UPDATE empleado SET salario=1500
WHERE nombre_emp = 'Cortecero';
```

Dará un error de tabla mutante ya que el disparador está intentando obtener información de la tabla sobre la que está definido (tabla empleado) y que, por lo tanto, es mutante.

Ejemplo de tabla de restricción. Supongamos dos tablas, empleado y departamento, de forma que en la tabla empleado tenemos almacenado el número de departamento, definido como clave ajena que referencia a la tabla departamento.

```
CREATE TRIGGER ejemplo
AFTER UPDATE ON nrodep OF departamento
FOR EACH ROW
BEGIN
        UPDATE empleado
        SET empleado.dep = :NEW.nrodep
        WHERE empleado.dep= :OLD.nrodep;
END ejemplo;
/
```

Al ejecutar:

```
UPDATE departamento
SET nrodep= 1
WHERE nrodep=7;
```

Dará un error de tabla mutante ya que estamos intentando modificar una columna correspondiente a una restricción de integridad.

```
CREATE TRIGGER TrigPrestamos
AFTER INSERT OR UPDATE OF Dni ON Prestamos
FOR EACH ROW
declare
x number;
begin
        SELECT COUNT(*) INTO x
        FROM Prestamos
        WHERE Dni = :NEW.Dni;
        if x >2 then
        raise application error(-20000,'El socio '||:NEW.Dni ||' excede el numero de prestamos
        permitidos.'); end if;
end;
```

Error de ejecución: Prestamos es una tabla mutante.

Una posible solución:

```
CREATE TRIGGER TrigPrestamos1
AFTER INSERT OR UPDATE OF Dni ON Prestamos FOR EACH ROW
begin
        INSERT INTO PrestamosTmp
        VALUES(:NEW.Dni,:NEW.Signatura);
end;

CREATE TRIGGER TrigPrestamos2
AFTER INSERT OR UPDATE OF Dni ON Prestamos
declare
        curdni char(9);
        x number;
        exceso_prestamos exception;
cursor CurPrestamos is
        SELECT DISTINCT Dni FROM PrestamosTmp
begin
        open CurPrestamos;
        fetch CurPrestamos into curdni;
        while (CurPrestamos%notfound) loop
                SELECT COUNT(*) INTO x
                FROM Prestamos
                WHERE Dni = curdni
                if x > 3 then raise exceso_prestamos; end if;
                fetch CurPrestamos into curdni;
        end loop;
        close CurPrestamos;
        DELETE FROM PrestamosTmp;
exception
        when exceso_prestamos then
                close CurPrestamos;
                DELETE FROM PrestamosTmp;
                raise_application_error(-20001,'El socio '|| curdni ||' excede el numero de prestamos
                permitidos.');
end;
```