Servicio Andaluz de Salud
CONSEJERÍA DE SALUD

Oficina Técnica para la Gestión y Supervisión de Servicios TIC
Subdirección de Tecnologías de la Información

Best Practices de desarrollo en Oracle PL/SQL

Referencia documento: InfV5_JASAS_PLSQL_BestPractices_V920.doc
Fecha: 16 de noviembre de 2018
Versión: 9.2.0

Registro de Cambios

Fecha	Autor	Versión	Notas
12 de Enero de 2012	Oracle ACS	3.1.	Versión inicial
14 de marzo de 2013	Oracle ACS	4.1	Revisión de marzo de 2013, contrato 2012-2014
13 de junio de 2013	Oracle ACS	4.2	Revisión de junio de 2013, contrato 2012-2014
17 de octubre de 2013	Oracle ACS	4.3	Revisión de octubre de 2013, contrato 2012-2014
16 de Julio de 2015	Paola Juárez	6.1	Revisión de Julio de 2015, contrato 2014-2016
16 de diciembre del 2015	Paola Juárez	6.2	Revisión de diciembre de 2015, contrato 2014-2016
16 de junio de 2016	Paola Juárez	7.1	Revisión de diciembre de 2016, contrato 2014-2016
16 de junio de 2017	Paola Juárez	8.1	Revisión de diciembre de 2017, contrato 2016-2018
16 de noviembre de 2017	Paola Juárez	8.2	Revisión de noviembre de 2017, contrato 2016-2018
16 de junio de 2018	Paola Juárez	9.1	Revisión de junio de 2018, contrato 2016-2018
16 de noviembre de 2018	Paola Juárez	9.2	Revisión de noviembre de 2018, contrato 2016-2018

Revisiones

Nombre	Role
Jonathan Ortiz	Oracle ACS Service Engineer
José María Gómez	Oracle Technical Account Manager

Distribución

Copia	Nombre	Empresa
1	Subdirección de Tecnologías de la Información	Consejería de Salud, Junta de Andalucía
2	Servicio de Coordinación de Informática de la Consejería de Hacienda y Administración Pública	Consejería de Hacienda y Administración Pública, Junta de Andalucía

Índice de Contenidos

Control de cambios

Cambio	Descripción	Página
	No se realizan cambios en esta versión del documento.	N/A

Introducción

Este documento recoge una serie de recomendaciones de Oracle Soporte planteadas como buenas prácticas de desarrollo para aplicaciones que hagan uso de Oracle PL/SQL.

Todas las best practices recogidas en este documento son implementables en la versión de PL/SQL 11gR1, aunque existen algunas de ellas, que a pesar de corresponderse con la versión 11gR2, por su utilidad e importancia se ha recogido igualmente en este documento.

Finalmente, estas recomendaciones están encaminadas a minimizar los posibles problemas de rendimiento en sistema de cualquier tamaño y en la gran mayoría de los casos se basan en la experiencia de casos reales gestionados por Oracle Soporte.

Best practices de PL/SQL

Uso de DBMS_OUTPUT

Evite el uso de DBMS_OUTPUT.PUT_LINE de forma directa

Posiblemente DBMS_OUTPUT.PUT_LINE es el mecanismo más común de depuración empleado por los desarrolladores PL/SQL, pero es seguro que es uno de los peores de todos los que se pueden elegir.

A pesar de que DBMS_OUTPUT.PUT_LINE existe desde la versión 2.0 de PL/SQL y que antes se hacía casi imposible depurar el código y que ha sufrido modificaciones en las últimas versiones de Oracle RDBMS, DBMS_OUTPUT sigue dejando mucho que desear.

- Productividad, se tiene que escribir 20 caracteres para mostrar algo.
- La sobrecarga de tipos es insuficiente, se puede pasar sólo cadenas de texto individuales, fechas o números. No se puede pasar un valor booleano, ni puede devolver varios valores, a menos que se use la concatenación.
- La longitud de la cadena es limitado, sobre todo antes de Oracle Database 10g Release 2. Si se intenta mostrar una cadena con más de 255 caracteres, se obtenía uno de los dos errores siguientes: ORA-20000/ORU-10028 u ORA-06502.
- El número de líneas es limitado, sobre todo antes de Oracle Database 10g Release 2, un programa puede mostrar un máximo de 1 millón de líneas, que puede ser muy inferior si se olvida especificar un número alto en el comando SET SERVEROUTPUT. A partir de Oracle Database 10g Release 2, el tamaño del buffer es ilimitado.
- No hay retroalimentación incremental, es decir, no se ve nada en la pantalla hasta que el programa PL/SQL haya terminado de ejecutarse, ya sean cinco minutos o cinco horas.

Como alternativa al uso de DBMS_OUT existen diversos framework en el mercado, aunque casi con toda seguridad, el más extendido y potente es LOG4PL/SQL, disponible en http://log4plsql.sourceforge.net/.

Uso de %ROWTYPE y %TYPE

Una de las principales motivaciones de los equipos de desarrollo es que al escribir programas se esperara que duren mucho, mucho tiempo y que sean capaces de adaptarse a los cambios y a las nuevas dependencias.

Sin embargo, es una práctica común, definir variables, por ejemplo VARCHAR2, con una longitud máxima en vez realizad declaraciones de variables usando el atributo%TYPE tal como se muestra a continuación:

```
PROCEDURE informe
IS
CURSOR q_cur IS SELECT titulo, descripcion FROM cuentas;
l_titulo cuentas.titulo%TYPE;
l_desc cuentas.descripcion%TYPE;
```

Ahora este programa se adapta automáticamente a los cambios en la tabla subyacente. Cada vez que la estructura de datos contra el cual se ha construido el código PL/SQL cambies, provocará un cambio de estado de éste a INVALID pero al recompilar, se utilizará automáticamente la nueva la estructura de datos y pasará de nuevo a estado VALID.

Estas declaraciones también son "auto-documentación": una declaración de %TYPE permite que cualquiera que lo lea, conozca qué tipo de datos está usando la variable. Igualmente también se puede utilizar el atributo ROWTYPE% para cargar un registro completo desde un cursor, una tabla o una vista. De hecho, este tipo de declaración tiene mucho más sentido para el caso anterior. Si reescribimos el ejemplo anterior usando ROWTYPE% quedaría de la siguiente forma:

```
PROCEDURE informe IS
CURSOR mi_cursor IS SELECT titulo, descripcion FROM
cuentas;
registro mi_cursor%ROWTYPE;
BEGIN
OPEN mi_cursor;
LOOP
FETCH mi_cursor INTO registro;
EXIT WHEN mi_cursor%NOTFOUND;
DBMS_OUTPUT.put_line ( 'Titulo/descripcion: '
|| l_record.titulo || ' - ' || l_record.descripcion);
END LOOP;
END informe;
```

Ahora se puede declarar simplemente una variable, de tipo récord, que tiene la misma estructura que el cursor. Este código es aún más resistente a los cambios. Las longitudes de las columnas pueden cambiar sin que el programa produzca errores. Incluso se pueden añadir más valores al SELECT de la consulta, y el registro definido mediante %ROWTYPE tendrá los campos adicionales que corresponda.

Finalmente, se podría simplificar este código aún más mediante el uso de un cursor bucle FOR, ya que para iterar a través del conjunto de filas se puede evitar la declaración de registro completo de la siguiente manera:

```
PROCEDURE informe IS
CURSOR mi_cursor IS SELECT titulo, descripcion FROM
cuentas;
BEGIN
OPEN mi_cursor;
LOOP
FETCH mi_cursor INTO registro;
EXIT WHEN mi_cursor%NOTFOUND;
DBMS_OUTPUT.put_line ( 'Titulo/descripcion: '
|| l_record.titulo || ' - ' || l_record.descripcion);
END LOOP;
END informe;
```

Conversiones implícitas de tipos.

El tema de las conversiones explicitas siempre es un tema controvertido, especialmente cuando se delega este tipo de conversiones en automatismos de la base de dato que pueden verse afectados por parámetros de inicialización a nivel de instancia, como el ejemplo siguiente:

```
DECLARE
    my_cumpleanos DATE := TO_DATE ('04-JAN-75', 'DD-MON-
RR');
```

En el código anterior, la indicar implícitamente la conversión y establecer de forma fija la máscara de conversión, hacemos que el código sea más consistente y predecible en su comportamiento, ya que no asumimos comportamientos que pueden verse afectador por temas externos al programa.

Uso de paquetes y variables globales

Una variable global es una estructura de datos a la que se le puede hacer referencia desde fuera del alcance o del bloque en el que se declara.

Una variable declarada en el nivel de paquete, fuera de cualquier procedimiento individual o de la función en ese paquete, es global en uno de dos niveles:

- Si la variable se declara en el cuerpo del paquete, entonces es el global y accesible desde todos los programas definidos en dicho paquete.

- Si la variable se declara en la declaración del paquete, entonces se puede acceder desde cualquier programa ejecutado a través del permiso EXECUTE.

En el siguiente bloque, por ejemplo, el publicación_date es global para el procedimiento local muestra_información:

```
DECLARE
publicación_date DATE;
...
PROCEDURE muestra_informacion IS
BEGIN
DBMS_OUTPUT.PUT_LINE (publicación_date);
END;
```

Las variables globales son peligrosas y deben evitarse ya que crean dependencias ocultas y/o efectos colaterales. Hay que tener en cuenta que estas variables globales no tiene que pasar a través de la lista de parámetros, así que es difícil saber siquiera que se está haciendo referencia a una global en un subprograma.

Además, si esa es una variable global, no es una constante, y se declara en la especificación del paquete, entonces se tiene un efecto de pérdida de control de los datos, al no poder garantizar la integridad de su valor, ya que cualquier programa que se ejecuta a través de EXECUTE sobre el paquete puede cambiar dichos valores.

Para evitar el uso de variables globales y por tanto las modificaciones sin control de estas, aplique las siguientes best practices:

- Pase las variables globales como parámetros en sus procedimientos y funciones, no haga referencia a ellas directamente desde programa.

- Declarar variables, cursores, funciones y otros objetos tan "profundamente" como sea posible. Es decir, sería en el bloque más cercano a donde el objeto se utiliza. Al hacer esto se reduce la posibilidad de un uso inadecuado por otras secciones del código.

- Ocultar datos de paquete detrás de funciones GET y SET: Se trata de los subprogramas que controlan el acceso a los datos.

- El alcance o ámbito de las declaraciones se crea lo más localmente posible, si la variable se utiliza sólo en un subprograma, se declara allí. Si tiene que ser compartida entre múltiples subprogramas en el mismo paquete, se declara a nivel de paquete, pero nunca en la especificación de paquetes.

De esta forma, declarando las variables a nivel del cuerpo paquete y disponiendo de métodos GET y SET, los desarrolladores pueden acceder a los datos para manipularlos siguiendo las reglas establecidas sin ninguna otra opción.

Uso de estructuras de control de flujo

Simplificación del código a través de la sentencia CASE

Las estructuras de control tienden con el tiempo a complicarse, o como mínimo a ampliarse en casos como las estructuras condicionales basadas en IF. Es decir, podemos encontrar casos de multiplicidad de sentencias IF, anidadas o no, para poder capturar la complejidad del negocio que estamos modelando.

En este tipo de situaciones, la expresión CASE nos permite simplificar el código a la vez que reducimos la cantidad de éste necesario. Veamos un ejemplo:

```
FUNCTION envia_correo (
asunto_in IN correos.ID%TYPE, destino_in IN VARCHAR2
DEFAULT NULL
) RETURN VARCHAR2
IS
envio correos%ROWTYPE := formatea (asunto_in);
BEGIN
RETURN CASE
WHEN destino_in IS NOT NULL
THEN 'Apreciado/a ' || destino_in || ', '
ELSE NULL
END
|| envio.descripcion
|| CASE
WHEN envio.url IS NOT NULL
THEN ' "' || envio.url || '"'
ELSE NULL
END;
END envia_correo;
```

En el ejemplo anterior, ya no es necesario declarar una variable que contenga el valor a devolver. Simplemente se construye un valor que es devuelto directamente por la sentencia RETURN.

Sentencias CASE siempre con clausula ELSE

Siempre que se use una sentencia CASE se deber incluir la clausula ELSE. En caso contrario, en el que la sentencia CASE no es capaz de encontrar una expresión coincidente, se generará un error del tipo: "ORA-06592: CASE not found while executing CASE statement", tal como se muestra en el siguiente ejemplo:

```
FUNCTION traductor (letra_in IN VARCHAR2) RETURN
VARCHAR2
IS
retorno VARCHAR2(100);
BEGIN
CASE
WHEN letter_in = 'A' THEN retorno := 'Es una A';
WHEN letter_in = 'B' THEN retorno := 'Es una B';
END CASE;
RETURN retorno;
END traductor;

SQL> BEGIN DBMS_OUTPUT.put_line (traductor ('O')); END;
2 /
*
ERROR at line 1:
ORA-06592: CASE not found while executing CASE statement
```

Finalmente recordar que si realizamos el siguiente cambio:

```
FUNCTION traductor (letra_in IN VARCHAR2) RETURN
VARCHAR2
IS
BEGIN
RETURN CASE
WHEN letra_in = 'A' THEN 'Es una A'
WHEN letra_in = 'B' THEN 'Es una B'
END;
END traductor;
```

La función simplemente devolverá un valor NULL si no le pasamos por parámetro bien una A o bien una B. En este caso deberemos saber diferencia y tratar este caso particular del NULL.

Uso del valor NULL

Hay que recordar que el valor NULL no es igual a nada, incluso no es igual a otro valor NULL y por tanto, es necesario que anticipemos esa posibilidad y que nos preparamos para ello.
En el siguiente ejemplo, vemos la posibilidad descrita anteriormente:

```
IF accion = 'AVISO'
THEN
    Enviar_aviso;
ELSIF accion = 'RECORDATORIO'
THEN
    Enviar_recordatorio;
ELSIF accion IS NULL
THEN
<responder al NULL>
ELSE
<respuesta a cualquier otro valor>
END IF;
```

Como podemos ver, el valor NULL es una alternativa válida y no tiene porqué significar ningún tipo de problema. En el caso de que no tengamos que actuar ante esa posibilidad, bastará con añadir la posibilidad y comentad el código tal como se muestra a continuación.

```
ELSIF accion IS NULL
THEN
/* Nada que hacer… ☺ */
NULL;
```

En otras palabras, incluso si no debe hacerse nada, es conveniente y recomendable añadir el código y comentarlo para hacerlo todavía más explícito. Quizá en un futuro se necesario su uso, y ya hayamos ganado tiempo, al menos, inicialmente se cubrieron todas las posibilidades.

Uso de bucles

Siga las siguientes best practices a la hora de usar bucles en PL/SQL:

- bucle FOR: Para iterar a través de cualquiera de una serie de valores enteros o sobre todas las filas de la obtienen de un cursor.
- bucle WHILE: Existe una condición de frontera o finalización antes de ejecutar el cuerpo del bucle.
- bucle LOOP: se utiliza cuando la lógica de la salida o finalización se codifica en el cuerpo del bucle.

Visto esto, veamos ahora las pautas a alto nivel para determinar qué tipo de bucle elegir en cada momento:

- bucle FOR sobre numéricos: Use este tipo de bucle cuando sepa que quiere o necesita recorrer todos los valores enteros entre dos valores y no existe condición de fin del bucle, excepto, posiblemente por una excepción.
- bucle FOR sobre cursores: Use este tipo de bucle cuando sepa que quiere o necesita recuperar todas las filas identificadas por el cursor y no existe condición de fin del bucle, excepto, posiblemente por una excepción.
- bucle WHILE: Use este tipo de bucle cuando no se sabe de antemano el número de iteraciones del bucle o cuando tenga que terminar el bucle de forma condicional sobre la base de una expresión booleana o cuando no desea que el cuerpo del bucle para ejecutar una sola vez.
- bucle LOOP: Use este tipo de bucle cuando no se sabe de antemano el número de iteraciones del bucle o cuando tenga que terminar el bucle de forma condicional sobre la base de una expresión booleana o cuando quiero que el cuerpo del bucle para ejecutar al menos una vez.

Ahora echemos un vistazo a algunas de las mejores prácticas relacionadas con bucles.

Uso de EXIT o RETURN desde bucles WHILE o FOR

Una recomendación clásica de los días de la programación estructurada puede ser "una forma de entrar, una manera de salir." Es decir, debe haber una manera de entrar en un bucle y una sola manera de salir de él.

Cuando esté a punto de escribir un programa con un bucle FOR, lo más importante es preguntarse: ¿realmente siempre se requiere recorrer todos los valores o los registros? Si es así, mantener alejado de las sentencias EXIT. Si, por el contrario, se da cuenta de que puede necesitar salir del bucle bajo algunas circunstancias, directamente no utilice el bucle FOR.

En su lugar, cambie a un bucle de tipo LOOP. El cambio puede parecer trivial, pero ahora es más fácil de depurar y comprender el código generado y a medida que la lógica se vuelve más compleja, será más fácil incorporar esta lógica de negocio. Trate el cuerpo de un bucle como un subprograma, con un recorrido bien definido y un único punto de salida.

Por último, esta best practice se aplica igualmente al uso de la sentencia GOTO dentro de bucles, ya que generan múltiples puntos de salida.

Declaraciones implícitas para bucles FOR

En el siguiente ejemplo, se ha declarado un cursor sobre la sentencia SQL. Como se puede observar, igualmente se ha declarado de forma implícita una estructura para recoger cada fila que devuelva el cursor:

```
PROCEDURE elimina_filas (filtro_in IN cuentas.id%TYPE)
IS
CURSOR cuentas_cur
IS
SELECT id, titulo FROM cuentas
WHERE titulo LIKE filtro_in;
cuenta_reg cuentas%ROWTYPE;
BEGIN
FOR cuenta_reg IN cuentas_cur
LOOP
Cuentas_pkg.delete (cuenta_reg.id);
END LOOP;
END elimina_filas;
```

Lo más probable es que el código compile pero falle en su ejecución, casi con toda seguridad debido a que la estructura contiene un NULL que no se esperaba.

Esto se debe exclusivamente a que se ha declarado la estructura de forma implícita en el código. Por tanto la best practices pasa por no declarar este tipo de índices o estructuras ya que, en este caso, el bucle FOR se encarga de hacerlo por nosotros de la forma adecuada.

Use bucle FOR para conjuntos densos y bucles WHILE para conjuntos dispersos

Una colección en PL/SQL es como una matriz unidimensional. Sin embargo, una colección diferente de una matriz en el que dos de los tres tipos de colecciones (tablas anidadas y arrays asociativos-anteriormente conocido como tablas de tipo index-by) puede ser dispersa. Esto significa que las filas que se define en la colección no tienen por qué ser secuencial, es decir, se puede asignar un valor a la fila 10 y un valor a la fila 10.000 y que no existan filas entre los dos.

Si el desarrollador usa el ID de cliente como el valor del índice de forma que le permite emular la clave primaria dentro de la colección, posiblemente no tenga en cuenta que al usar un bucle de tipo FOR pueden existir huecos o gaps en dicha clave primaria y por tanto en el índice de la colección.

Cuando se escanea una colección con un bucle FOR, PL/SQL comprueba cada valor del índice dentro de los valores límites, independientemente de si se define o no una colección. El bucle FOR es totalmente ignorante de este hecho, por lo que tan pronto como el bucle FOR intenta acceder a una línea no definida, la base de datos Oracle lanzará una excepción de tipo NO_DATA_FOUND .

En esta situación, la best practice recomienda cambiar el bucle FOR por los métodos de acceso a colecciones FIRST, LAST y NEXT.

Al evitar el bucle FOR, no se hace ninguna suposición sobre el contenido de la colección. El código es un poco menos simple, pero es mucho menos probable que genere excepciones.

Veamos las pautas que se debe seguir para encontrar la mejor forma para recorrer las colecciones:

- Si su colección puede ser escasa o poco densa, use FIRST y NEXT para iterar desde los valores más bajos a los valores más altos del índice. Igualmente puede usar LAST y PRIOR para acceder a los valores más altos del índice.

- Si la colección se rellena con una consulta de tipo BULK COLLECT, entonces vacíe y llene siempre de forma secuencial. En este caso, puede utilizar un bucle FOR de forma segura a través de todos los elementos:
    - FOR indice_variable IN 1 .. coleccion_nombre.COUNT
    - El código del bucle FOR es mucho más simple de escribir y mantener y se debe utilizar siempre que sea posible ya que se trata de una opción completamente segura si se une a sentencias de tipo BULK COLLECT.

- Como variación de menor importancia sobre esta best practices, cuando se trabaja con bucles de tipo FORALL, se puede utilizar las clausulas INDEXES OF para recorrer automáticamente los valores del índice definido en la colección.

Uso de estructuras condicionales

Uso GOTO y CONTINUE exclusivamente cuando no tenga otra opción

Ambas declaraciones le permiten realizar ramificaciones no estructurados de la ejecución de su programa. Sin embargo, la best practice es simple: evite su uso siempre que sea posible.

Por lo general, se puede considerar el uso de cualquiera instrucción GOTO o CONTINUE cuando es necesario modificar la lógica compleja de un programa. En tal situación, se suele optar por "escapar" del control de flujo de una manera tan limpia como sea posible, dejando la lógica existente sin cambios. En este caso, lo más importante que puede hacer es documentar claramente la justificación y el uso este tipo de declaraciones.

Uso de excepciones

Antes de entrar en las best practices específicas de las excepciones, se debe estar seguro de entender cómo funciona el manejo de estas excepciones. Muchos de nosotros pensamos que conocemos todas las reglas y todo lo que PL/SQL tiene que ofrecernos, pero la realidad es que hay algunos aspectos muy poco intuitivo en la gestión de error de Oracle PL/SQL. Además, en casi cada nueva versión de la base de datos, Oracle agrega algunos cambios y características nuevas.

A continuación ofrecemos una lista con las funciones básicas y las capacidades de gestión de errores en PL/SQL:

- Sección de ejecución, una sección de tratamiento de excepciones manejará exclusivamente los errores provocados en el bloque de la sección de ejecución. Los errores provocados en la sección de declaración o en la propia sección de excepción no serán tratados por esta última.

- Propagación, se puede control la propagación de una excepción, es decir, hasta donde puede llegar una excepción no manejada, colocando el código que puede provocar la excepción en un bloque anónimo anidado con su propia sección de excepciones.

- SQLERRM, por regla general no es recomendable usar SQLERRM para conseguir el mensaje de error asociado al último error SQL ya que puede ser truncado. En vez de esto, es recomendable usar la función DBMS_UTILITY.FORMAT_ERROR_STACK.

- SAVE EXCEPTIONS and SQL%BULK_EXCEPTIONS, use SAVE EXCEPTIONS con los bucles FORALL para continuar después de una excepciones en operaciones de tipo BULK BIND. Posteriormente se podrá recorrer la vista SQL%BULK_EXCEPTIONS para recuperar los errores producidos.

- DBMS_UTILITY.FORMAT_BACK_TRACE: A partir de Oracle RDBMS 10gR2, puede usarse DBMS_UTILITY.FORMAT_BACK_TRACE para obtener el número de la línea desde donde se ha lanzado la excepción más reciente.

- DBMS_ERRLOG, a partir de Oracle RDBMS 10gR2, podemos usar DBMS_ERRLOG para continuar tras un error de tipo DML sin elevar una excepción. De forma similar a SAVE EXCEPTIONS, DBMS_ERRLOG se encargará de almacenar cada una de las excepciones en una tabla sin elevar excepciones. Posteriormente esta tabla puede ser consultada.

- AFTERSERVERERROR: Ese trigger nos permite definir la lógica necesaria para el tratamiento de excepciones a nivel de la instancia de Oracle RDBMS. Este triggers es disparado exclusivamente cuando la excepción no es capturada dentro de un bloque PL/SQL. Finalmente, es necesario recordar que este tipo de triggers no se dispara bajo ciertas excepciones como los errores ORA-600.

Tipos de excepciones

No todas las excepciones son iguales. Algunas excepciones, por ejemplo, las que indican un error grave a nivel de base de datos o problemas a bajo nivel como un ORA-600, no es lo mismo que una excepción del tipo NO_DATA_FOUND. Este último caso, ocurren de forma tan habitual que realmente no pensamos en ellas como errores. Normalmente se añade una sentencia IF mas que cubra esa posibilidad.

Por tanto, podemos categorizar las excepciones en diferentes tipos, veamos cuales:

- Deliberadas: La estructura del código en si mismo genera la excepción como parte de su funcionalidad. Es decir, el código anticipa la posibilidad de la generación de la excepción.
- Desafortunadas: Este tipo de excepciones son esperables pero no tienen porqué significar ningún error. Por ejemplo, un SELECT INTO que genera un NO_DATA_FOUND.
- No esperadas: Este tipo indican un claro problema en la aplicación. Por ejemplo, un SELECT INTO, es decir, se supone que se devuelve una única fila para una clave dada, y sin embargo se genera un error TOO_MANY_ROWS.

En los siguientes apartados veremos cada uno de estos tipos de forma detallada.

Estudio de las excepciones deliberadas

Veamos un ejemplo de este tipo de excepciones. Si usamos UTL_FILE.GET_LINE para leer el contenido de un fichero, llegará el momento en el que éste se acabará y se generará una excepción de tipo

NO_DATA_FOUND. Es el comportamiento esperado y por tanto es parte del uso de la funcionalidad. Veamos un ejemplo:

```
PROCEDURE lee_fichero (
dir_in IN VARCHAR2, fichero_in IN VARCHAR2
)
IS
l_fichero UTL_FILE.file_type;
l_linea VARCHAR2 (32767);
BEGIN
l_fichero := UTL_FILE.fopen (dir_in, fichero_in, 'R',
max_linesize => 32767);
LOOP
UTL_FILE.get_line (l_fichero, l_linea);
Trata_los_Datos;
END LOOP;
EXCEPTION
WHEN NO_DATA_FOUND
THEN
UTL_FILE.fclose (l_fichero);
END;
```

Estudio de las excepciones desafortunadas y excepciones no esperadas

Veamos estos dos tipos de excepciones con los ejemplos anteriores: NO_DATA_FOUND y TOO_MANY_ROWS. Supongamos un subprograma que devuelve el nombre completo de un empleado a partir de DNI:

```
FUNCTION dame_empleado (dni_in IN empleado.dni%TYPE)
RETURN VARCHAR2
IS
 retorno VARCHAR2 (32767);
BEGIN
 SELECT nombre || ' ' || apellido1
 INTO retorno
 FROM empleados
 WHERE dni = dni_in;
 RETURN retorno;
END dame_empleado;
```

En el ejemplo anterior, si llamamos a este subprograma con un DNI de empleado que no está en la tabla, la base de datos levantará la excepción de tipo NO_DATA_FOUND. Si por el contrario, lo llamamos con un DNI de empleado que se encuentra en más de una fila de la tabla, la base de datos levantará la excepción de tipo TOO_MANY_ROWS.

Analicemos las dos excepciones por separado:

- NO_DATA_FOUND: Indica que no se ha encontrado ninguna coincidencia. Esto puede ser un problema o no en función de la lógica de nuestra aplicación y de los datos tratados. Puede ser que simplemente se haya introducido un valor de de DNI de un empleado nuevo. Por tanto, podemos categorizar esta excepción como una excepción de tipo

desafortunada ya que en este caso no puede considerarse un error del subprograma.

- TOO_MANY_ROWS: Este tipo de excepciones suelen ser un problema critico en nuestras aplicaciones, ya que o es un error en los datos introducidos como parámetros o es un error en la clave de restricción de clave primaria. Por tanto, podemos clasificar claramente este tipo de excepciones como errores del subprograma.

Tratamiento de los diferentes tipos de excepciones

Veamos a continuación una serie best practices para tratamiento de cada uno de estos tipos de excepciones:

- Deliberadas: Tal como se comentó anteriormente, es un tipo de excepción anticipada o prevista, por lo que hemos generado código para su tratamiento. En este punto, es fundamental cumplir la best practice de no añadir lógica de negocio en el tratamiento de este tipo de excepciones. La sección de código correspondiente deberá exclusivamente contener lo necesario para manejar el error, es decir, almacenar el código y descripción del error, relanzar la excepción, etc. Cualquier otra opción, complica el entendimiento del código y por tanto su mantenimiento.

- Desafortunadas: Existen situaciones donde el código levanta excepciones que no tiene por qué interpretarse como un error por lo que no suelen propagarse más allá de la zona que la captura inicialmente. En este tipo de casos, se suele devolver un valor y/o activar algún tipo de flag que indica que la excepción se ha producido, dejando al desarrollador la posibilidad de decidir qué hacer con dicha excepción, tratarla, ignorar o levantarla de forma definitiva.

- Inesperadas: Este tipo de excepciones, sin excepción, debe ser tratada y gestionada en un contexto de aplicación lo más cercano posible a lugar de generación. Por regla general, este tipo de excepciones hacen que un subprograma termine, bien con dicha excepción bien con otra distinta a través de la sentencia RAISE con el fin de que se trate de forma obligatoria por parte del código del programa que realiza la llamada.

Uso de RAISE_APPLICATION_ERROR.

Veamos un ejemplo, donde se realiza la comprobación de una regla de negocio a través de un trigger. En dicho tratamiento se aplica dos best practices que vemos a continuación:

- No codifica reglas de negocio directamente en el código del trigger. Es recomendable usar funciones que almacenen esta misma regla de negocio con el fin de poder ser reutilizada desde todas las partes del código.

- No usa llamadas a RAISE_APPLICATION_ERROR de forma directa desde el código del trigger. En su lugar se llama a un paquete de gestión de errores, que toma la información necesaria para documentar la incidencia y la almacena para su tratamiento posterior.

Veamos el código con las best practices aplicadas:

```
CREATE OR REPLACE TRIGGER limita_edad
BEFORE INSERT OR UPDATE ON clientes FOR EACH ROW
BEGIN
IF reglas_pkg.mayor_de_edad (:NEW.fecha_nacimiento)
THEN
Log_pkg.error (
error__in => 'MENOR DE EDAD'
, nombre1_in => 'DNI'
, valor1_in => :NEW.dni
, nombre2_in => 'FECHA_NACIMIENTO';
, valor2_in => :NEW.fecha_nacimiento);
END IF;
END limita_edad;
```

Al usar un paquete de gestión de errores y no llamar directamente al procedimiento RAISE_APPLICATION_ERROR nos permite poder cambiar el comportamiento que deseamos para nuestras excepciones sin tener que cambiar todas las llamadas al dicho procedimiento. Bastaría con cambiar los parámetros o el código del paquete de gestión de errores para cambiar el comportamiento del trigger ante las excepciones o del código de todos nuestros subprogramas.

Uso de EXCEPTION_INIT

Veamos el siguiente ejemplo. En él se define una sección para el tratamiento de excepciones. Dado que se desea un tratamiento con un nivel de detalle muy bajo se decide filtrar los distintos errores que provocan las excepciones.

```
PROCEDURE insercion_masiva
IS
...
EXCEPTION
WHEN OTHERS
THEN
IF SQLCODE = -24381
THEN
FOR numero_errores IN 1 .. SQL%BULK_EXCEPTIONS.COUNT
LOOP
...
```

Como se puede observar, se ha utilizado un valor fijo en el tratamiento de las excepciones, es decir, se ha dejado "hardcoded" el valor 24381. Posiblemente el desarrollador tenga muy claro en el momento de la creación del código lo que significa dicho error, pero tenemos que pensar en el mantenimiento de dicho código por parte de otros desarrolladores.

En este escenario, la best practice ha aplicar pasa por dotar a dicho error de un nombre a través el programa EXCEPTION_INIT. Veamos el ejemplo anterior con la best practice aplicada:

```
PROCEDURE insercion_masiva
IS
errores_en_forall EXCEPTION
PRAGMA EXCEPTION_INIT (errors_in_forall, -24381);
...
EXCEPTION
WHEN errores_en_forall
THEN
FOR numero_errores IN 1 .. SQL%BULK_EXCEPTIONS.COUNT
LOOP
...
```

Al usar EXCEPTION_INIT, no solo estamos realizando un código más sencillo de entender y de mantener si no que está autodocumentado en sí mismo.

La ventaja de la best practice es clara. Sin embargo podemos ir un poco más lejos si pensamos en la cantidad de procedimientos y funciones donde podemos necesitar definir estas excepciones con EXCEPTION_INIT. En este caso podemos definir a nivel de proyecto todas las que consideremos convenientes en un paquete. Veamos el ejemplo de esta nuevo best practice:

```
PACKAGE oracle_excepciones
IS
errores_en_forall EXCEPTION
PRAGMA EXCEPTION_INIT (errors_en_forall, -24381);
...
END;

PROCEDURE insercion_masiva
IS
...
EXCEPTION
WHEN oracle_excepciones.errores_en_forall
THEN
FOR numero_errores IN 1 .. SQL%BULK_EXCEPTIONS.COUNT
LOOP
...
```

Uso de WHEN OTHERS THEN NULL.

El uso del patrón WHEN OTHERS THEN NULL es posiblemente una de las prácticas más comunes y uno de los patrones con resultados mas catastróficos en producción, ya detectar las causa y poder depurar este tipo de situaciones se hace muy complicado.

Hay que tener en cuenta, que la mayoría de las veces el código se desarrolla con un conjunto de datos, una base de datos de desarrollo, muy diferente al conjunto de datos que usará en producción, base de datos de producción, por tanto debemos anticiparnos a este tipo de situaciones y en ningún momento podemos ignorar las excepciones que puedas producirse con el uso de este tipo de patrón.

Uso de SQL en PL/SQL

SQL y DML siempre tras procedimientos y funciones

Puede parecer irónico pero una de las principales características de PL/SQL, la integración con SQL, puede suponer una de las peores prácticas que podemos realizar al codificar en PL/SQL. Esta situación suele corregirse cuando los desarrolladores de PL/SQL entienden que programar SQL y programa PL/SQL son dos temas totalmente distintos con particularidades distintas.

La clave está en no repetir la misma SQL o DML en el código. La repetición es la mayor fuente de errores y el gasto de horas en solventarlos tiende a infinito cuando el proyecto es de un tamaño considerable. La mejor forma de conseguir no repetir las sentencias SQL ni las sentencias DML es ocultarlas tras funciones y procedimientos. Cuando un subprograma necesite la sentencia SQL en cuestión, realizará una llamada al procedimiento o a la función que la almacena.

Esta best practices tiene una doble funcionalidad. Al no repetir las sentencias SQL, minimizamos el hardcoding de nuestro código respecto a la estructura de tablas y objetos de nuestra base de datos.

Hay que tener en cuenta que casi con toda seguridad, las sentencias SQL y DML es lo más íntimamente relacionado con los requisitos funcionales de todo el código que realizamos. Es decir, toda sentencia SQL o DML es un hardcoding en el código PL/SQL. Por tanto, es recomendable minimizar todo lo posible el hardcoding de SQL y DML en PL/SQL y para ello las funciones y procedimientos que encapsulan las sentencias SQL y DML son fundamentales.

En el caso de que los requisitos funcionales de la aplicación cambien o bien el rendimiento de una determinada SQL en producción no sea aceptable, no tendremos que buscar en múltiples lugares del código, bastará con cambiar la función o procedimiento en cuestión y recompilar.

Uso de AUTONOMOUS_TRANSACCTION

Tal como se ha comentado anteriormente, la clave está en encapsular las sentencias SQL y DML en funciones y procedimientos. En el caso de las DML, posiblemente nos interese realizar un control de excepciones,

implementar un sistema de auditoría, etc. Es decir, posiblemente tengamos la necesidad de escribir que no estrictamente de negocio o que necesitamos escribir de todas formas, sea o no, completada la información de negocio.

En la mayoría de los casos cuando se produce una excepción, el motor de PL/SQL genera y propaga una excepción que casi con toda seguridad termine como un rollback de toda las DML en vuelo.

Para este tipo de situaciones, la best practice recomienda el uso de una funcionalidad presente desde la versión 8i de Oracle RDBMS: autonomous transaction.

Esta funcionalidad permite que todo bloque PL/SQL, procedimiento, función, trigger o bloque anónimo, se ejecute como una transacción autónoma e independiente de la transacción en vuelo simplemente añadiendo el siguiente pragma:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

En el caso de usar este pragma y realizad cualquier operación DML en el subprograma, éste tendrá que realizar de manera independiente la ejecución de un COMMIT o ROLLBACK ya que no dependerá de la transacción de negocio principal.

Veamos un ejemplo:

```
PROCEDURE log_error (codigo IN PLS_INTEGER, msg IN
VARCHAR2)
IS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
INSERT INTO error_log (error_codigo, error_texto, fecha,
usuario)
VALUES (codigo, msg, SYSDATE, USER);
COMMIT;
EXCEPTION
WHEN OTHERS THEN ROLLBACK;
END log_error;
```

De esta forma, nuestro subprograma guardará la información de log independientemente del resultado final de nuestra operación de negocio:

```
BEGIN
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "HR.MAKE_CHANGES", line 21
ORA-06512: at line 2
SELECT * FROM error_log
/
ERRR_CODIGO ERROR_TEXTO              FECHA     USUARIO
----------- ------------------------ --------- -------
```

```
100         ORA-01403: no data found 01-JUN-07 HR
```

Hemos visto la potencia de la que disponemos a través de este pragma, pero hay que recordar que cada transacción se ejecuta en un espacio diferente. Es decir, no se comparten bloqueos entre ambas transacciones/sesiones, por lo que pueden aparecer situaciones como deadlocks. En este caso, la solución pasa por revisar el código para localizar aquellas filas que están siendo modificadas desde, al menos, dos lugares distintos.

Uso de COMMIT y ROLLBACK.

Tal como se ha comentado anteriormente en el documento, el hardcoding es una de las peores prácticas de programación en PL/SQL y en cualquier otro lenguaje.

Parte del problema está en saber que partes de nuestro código realmente están sufriendo esta mala práctica. Por ejemplo, cada vez que escribimos un COMMIT o ROLLBACK en nuestro código, estamos haciendo "hardcoded" toda la gestión transaccional de nuestro subprograma.

Este tipo de situaciones se da aún más cuando se está desarrollando contra un conjunto de datos que no se quiere modificar por lo que llenamos nuestro código de llamadas a ROLLBACK. Posteriormente cuando este código pasa a producción tenemos que modificarlo para sustituir todos los ROLLBACKs por COMMITs. La dificultad estriba en que puede que no sean todos y tengamos que dedicar más tiempo aún en realizad las sustituciones de forma correcta.

Para evitar este tipo de situaciones, podemos recurrir a un procedimiento almacenado al que llamaremos siempre al finalizar nuestras transacciones que en función del entorno y de la fase de desarrollo en la que se encuentra, realizará el ROLLBACK o el COMMIT en base un parámetro o bien a un ROLLBACK o un COMMIT, eso sí, "hardcoded" en dicho procedimiento. En el peor de los casos es solo una línea la que queda "hardcoded" y siempre muy bien localizada. Veamos un ejemplo:

```
PROCEDURE calculo (tipo_in IN VARCHAR2)
IS
BEGIN
 .... operaciones DMLs
 -- No realizado ningun commit ya que estoy probando.
END calculo;
```

En el anterior caso, podemos añadir una llamada a un procedimiento encargado de finalizar la transacción, quedando de la siguiente manera:

```
PROCEDURE Finaliza_transaccion (tipo_in IN VARCHAR2)
IS
BEGIN
 commit;
 -- rollback;
```

```
END Finaliza_transaccion;

PROCEDURE calculo (tipo_in IN VARCHAR2)
IS
BEGIN
… operaciones DMLs
 Finaliza_transaccion;
… mas operaciones DMLs;
 Finaliza_transaacion;
END calculo;
```

De esta forma podemos llamar al procedimiento finaliza_transaccion tantas veces como deseemos cuando deseemos distintos comportamientos a nivel transaccional de nuestro código.

Best Practices para consulta de datos desde PL/SQL

Uso de %ROWTYPE en cursores

Esta es una de las best practices más simples: siempre que se realice un fetch contra un cursor debe hacerse directamente contra un atributo de tipo %ROWTYPE.

Veamos un ejemplo:

```
PROCEDURE consulta_compleja
IS
 l_una_fila mi_paquete.mi_cursor%ROWTYPE;
BEGIN
 OPEN mi_paquete.mi_cursor;
 FETCH mi_paquete.mi_cursor INTO l_una_fila;
 ...
END consulta_compleja;
```

En el ejemplo anterior no solo hemos encapsulado la definición del cursor en un paquete para poder reusarlo tantas veces como deseemos sino que hemos incluido en su uso la declaración de una variable de tipo %ROWTYPE sobre dicho cursor.

De este modo, podremos cambiar cuantas veces necesitemos nuestro cursor que todo el código dependiente de él, no se verá alterado y bastará con una simple recompilación para que vuelva a funcionar.

Uso de COUNT

Uno de los problemas más típicos de rendimiento en producción viene por la típica confusión a la hora de implementar la solución a la pregunta: ¿Hay al menos un registro?, con la respuesta a la pregunta: ¿Cuántos registros existen?.

Aunque la diferencia es clara, por regla general en todas las bases de datos de producción existen sentencias SQL del tipo *SELECT COUNT(*) INTO VARIABLE FROM TABLA WHERE CONDICION* simplemente para averiguar si existe un elemento que cumpla el criterio de filtrado.

A pesar de todo, este tipo de patrones tiene fácil de solucionar como se puede ver en el ejemplo siguiente:

```
FUNCTION busca_periodo (cliente_id_in IN
clientes.ID%TYPE, fecha_inicio_in IN DATE, fecha_fin_in
IN DATE)
RETURN BOOLEAN
IS
 l_dummy PLS_INTEGER;
 l_es_usado BOOLEAN;
 CURSOR es_usado_cursor
```

```
 IS
  SELECT 1 valor_dummy FROM clientes
  WHERE cliente_id = cliente_id_in
  AND usado_on BETWEEN fecha_inicio_in AND fecha_fin_in;
BEGIN
 OPEN es_usado_cursor;
 FETCH usado_cursor INTO l_dummy;
 l_es_usado := usado_cursor%FOUND;
 CLOSE es_usado_cursor;
 RETURN l_es_usado;
END busca_periodo;
```

En la versión anterior, el SELECT COUNT(*) se ha sustituido por un cursor y por un fetch de la primera fila. Con esto nos vale para evaluar la condición de existencia con un consumo mínimo respecto al perfil de ejecución del COUNT(*). Finalmente no tenemos más que cerrar el cursor y devolver el valor lógico mediante el RETURN.

Como se puede observar, la especificación y la implementación coinciden con un gasto mínimo de recursos, por lo que podemos concluir que no es necesario usar COUNT para situaciones donde solo necesitemos saber la condición de existencia, dejando exclusivamente su función para situaciones donde realmente sea necesario saber exactamente el número de filas que cumplen con el criterio en cuestión.

Uso del cursores tipo FOR para múltiples filas

Es una práctica generalizada usar bucles FOR para recorrer cualquier tipo de conjunto de datos, ya sean grandes o simplemente una fila. Los bucles FOR, como casi todas las estructuras PL/SQL, durante su ejecución necesita una cantidad de recursos muy superior a su equivalente SQL. Es decir, el uso de cursores debe estar justificado y delimitado a situaciones donde es necesario hacer algo sobre cada una de las filas seleccionadas.

Sin embargo, este patrón no es el más típico, y lo normal es encontrar cursores usados exclusivamente para acceder a una única fila, ya sea por condición de filtrado o por la propia naturaleza de los datos, una restricción de unicidad por ejemplo.

En este tipo de casos, la best practices recomienda el uso de estructuras más simples tanto a la hora de ser escritas y mantenidas como a la hora de su ejecución, consume menos recursos, como el patrón SELECT INTO.

Dicho patrón es equivalente al uso de un cursor y a un fetch sobre su primera fila, tal como hemos comentado antes, algo muy normal en las aplicaciones, con un consumo mínimo. Veamos un ejemplo:

```
FUNCTION dame_cliente (nif_in IN clients.id%TYPE)
RETURN clientes%ROWTYPE
IS
l_unico_cliente clientes%ROWTYPE;
BEGIN
```

```
SELECT * INTO l_unico_cliente
FROM clientes
WHERE id = dni_in;
RETURN l_unico_cliente;
EXCEPTION
WHEN NO_DATA_FOUND THEN RETURN l_unico_cliente;
END dame_cliente;
```

Best Practices para modificación de datos desde PL/SQL

DML siempre tras procedimientos y funciones

Al igual que las sentencias SQL, las sentencias DML también son candidatas de encapsularse en funciones y procedimientos, incluso son más triviales de construir dado que la variedad de operaciones es mucho menor que en el caso de las sentencias SQL.

La best practice recomienda la creación de una serie de funciones y procedimientos de operaciones DML a modo de API sobre la tabla en cuestión. De esta forma no solo reusamos el código, sino que ahorramos tiempo en modificaciones, por la centralización de los cambios, unificamos criterios como la información de depuración que se va a generar, y un largo etc.

En el caso de que no se pueda seguir esta best practices, y se opte por escribir cada una de las DML necesarias a medida que se vayan necesitando, es aconsejable seguir los siguientes criterios:

- Añadir siempre una sección de excepciones.
- Prever todos los errores potenciales que podamos localizar.
- Guardar toda la información relevante como información de depuración de las excepciones.
- Generar un mensaje adecuado para el usuario del programa, no hace falta devolver el error ORA generado, eso debería haberse capturado convenientemente como información de depuración.

Uso de listas de columnas en operaciones de tipo INSERT

Veamos el siguiente ejemplo, donde queremos aplicar la anterior best practices, encapsular todas las operaciones DML sobre una tabla en procedimientos y funciones.

```
CREATE TABLE parametros (
  categoria VARCHAR2(100)
, nombre VARCHAR2(100)
, valor VARCHAR2(4000)
);
```

Para esta tabla, podemos generar el siguiente bloque anónimo de PL/SQL para dotarla de contenido:

```
BEGIN
INSERT INTO parametros (categoria, nombre, valor) VALUES
('PANTALLA', 'COLOR', 'VERDER');
INSERT INTO parametros (categoria, nombre, valor) VALUES
('PANTALLA', 'COLOR', 'AZUL');
INSERT INTO parametros (categoria, nombre, valor) VALUES
('PANTALLA', 'MOSTRAR_SIEMPRE', 'NO');
INSERT INTO parametros (categoria, nombre, valor) VALUES
('PANTALLA', 'MOSTRAR_SIEMPRE', 'SI');
END;
```

Sin embargo, nuestro objetivo es generar PL/SQL que se encargue de encapsular todas estas operaciones. Veamos la siguiente opción:

```
PROCEDURE aniade_opcion (
categoria_in IN VARCHAR2, nombre_in IN VARCHAR2,
valor_in IN VARCHAR2)
IS
BEGIN
INSERT INTO parametros VALUES (categoria_in, nombre_in,
valor_in);
END aniade_opcion;
```

El anterior código cumple con las necesidades planteadas y funciona correctamente. Funciona hasta que realizamos por necesidad del negocio la siguiente adición de una columna a la tabla:

```
ALTER TABLE parametros ADD por_defecto VARCHAR2(1)
DEFAULT 'N';
```

Inmediatamente, nuestro procedimiento dejará de funcionar al no compilar. Para evitar este tipo de situaciones la best practices recomienda el uso de los nombres de las columnas implicadas antes de la clausula VALUES en todas las operaciones de tipo INSERT tal como se muestra a continuación:

```
PROCEDURE aniade_opcion (
categoria_in IN VARCHAR2, nombre_in IN VARCHAR2,
valor_in IN VARCHAR2)
IS
BEGIN
 INSERT INTO parametros (categoria, nombre, valor)
 VALUES (categoria_in, nombre_in, valor_in)
END aniade_opcion;
```

Dado que la nueva columna tiene un valor por defecto no es necesario añadirlo a la sentencia INSERT, el procedimiento compilará y funcionará correctamente tras añadir la columna a la tabla.

De esta forma, añadiendo la lista de columnas a las sentencias INSERT, haremos un código más flexible, duradero y autodocumentado.

Uso y consideraciones de SQL%ROWCOUNT

Recordemos que las operaciones INSERT, UPDATE y DELETE siempre se ejecutan como cursores implícitos en PL/SQL. Al ser implícitos, no es necesario declararlos ni abrirlos y simplemente invocamos las operaciones y la base de datos se encarga de realizar la gestión de estos cursores.

De la última operación implícita podemos obtener información a través de los atributos de cursores que se listan a continuación:

- SQL%ROWCOUNT, número de filas que han sido afectadas por la sentencia DML.
- SQL%ISOPEN, siempre es FALSE, ya que el cursor es abierto y cerrado de forma implícita.
- SQL%FOUND, TRUE si la sentencia afecta al menos un fila.
- SQL%NOTFOUND, FALSE si la sentencia no afecta al menos a una fila.

Dado que solo existe un único conjunto de atributos SQL% por sesión y que reflejan exclusivamente la última operación implícita ejecuta, es recomendable mantener al mínimo la cantidad de código entre la ejecución de la operación DML y la referencia a estos atributos.

De cualquier otra manera, los valores retornados por los atributos no se corresponderán con la sentencia en cuestión, provocando bugs del código difíciles de localizar.

Finalmente recordar que en el caso de sentencias DML, el atributo SQL%ROWCOUNT devuelve el número de filas afectadas por la operación DML más reciente en la sesión. Por tanto, podemos usarla para comprobar en qué grado nuestra sentencia ha tenido éxito o no. Solo recordar, por ejemplo, que una operación de tipo UPDATE no genera excepciones en el caso de no actualizar ningún registro.

```
BEGIN
 UPDATE clientes
 SET nombre = 'ACME'
 WHERE nombre = 'ACME Corp.';
 IF SQL%ROWCOUNT < 2 THEN
  ROLLBACK;
 END IF;
END;
```

Alternativamente, podemos evitar el uso de los atributos SQL%, y usar la clausula RETURNING para obtener un valor equivalente, el número de filas afectadas por la operación, pero a diferencia del atributo, la clausula queda vinculada a la sintaxis de la operación de tipo UPDATE. Veamos un ejemplo:

```
DECLARE
 l_modificados PLS_INTEGER;
BEGIN
```

```
UPDATE clientes
 SET nombre = 'ACME'
 WHERE author = 'ACME Corp';
 RETURNING l_modificados;
IF l_modificados < 2
  ROLLBACK;
 END IF;
END;
```

Uso de parámetros

La cabecera de un subprograma consiste en su nombre, una lista de parámetros opcionales, y una cláusula de retorno si es el subprograma es una función. Si bien es posible definir los procedimientos y funciones sin ningún parámetro, no es una práctica recomendable.

En su lugar, se debe utilizar una lista de parámetros para describir claramente los valores que el programa necesita para hacer su trabajo, y los valores que el programa volverá al bloque de llamada de código.

En varias de las siguientes best practices nos referiremos a los parámetros formales y a los parámetros reales. Un parámetro formal" se refiere al parámetro que se declara en la lista de parámetros y es manipulado dentro del programa. Un "parámetro real" se refiere al valor, una variable o expresión que se pasa en el programa cuando se ejecuta.

Todos los argumentos de tipo IN debe tener un valor por defecto.

Los programadores suelen centrarse en hacer que el código generado cumplas los nuevos requisitos pero en ocasiones olvidan el hecho de que el código modificado ya está siendo llamado desde otros lugares del código.

Para asegurar que el código final será consistente y que compilará tanto él como sus dependencias, podemos usar las siguientes tres opciones:

- Proporcionar un valor predeterminado para el nuevo parámetro de tipo IN.
- Volver a cada invocación existente del código y añadir el nuevo parámetro.
- Crear una sobrecarga distinta del código que contenga el nuevo parámetro, dejando a la cabecera del programa existente sin cambios.

Es fácil ver que la primera opción es la más simple y más rápida de implementar. En un ejemplo simple, la best practices se implementaría de la siguiente forma:

```
FUNCTION estimacion (
cliente_id_in IN cliente.id%type
, check_frecuencia_in IN BOOLEAN
, check_variedad_in IN BOOLEAN
, check_usodirario_in IN BOOLEAN DEFAULT FALSE
```

```
) RETURN PLS_INTEGER
```

Ahora, todas las llamadas existentes para la función estimación seguirá siendo válida, y con el valor predeterminado es FALSE, no se verán afectados por el análisis de uso durante el día. Si, por otro lado, algunas de esas llamadas existentes para a la función estimación realmente tiene que tomar el tiempo de utilización en cuenta, los programadores tendrá que ir a cada una de esas llamadas y agregar este parámetro, con el valor apropiado.

Por último, si el modo del nuevo parámetro es IN OUT u OUT, no podemos confiar en los valores por defecto para ocultar el nuevo argumento y puede significar un cambio sustancial en la funcionalidad y por tanto queda fuera de esta best practice.

En este caso, lo más probable es que se desee crear otra versión del programa, dejando sin cambios el programa existente.

La mejor manera de hacerlo es que el programador se asegure de que su subprograma se define dentro de un paquete, y luego agregar una sobrecarga de ese subprograma: un procedimiento o función con el mismo nombre, pero con la nueva lista de parámetros que contiene la salida o en el argumento de tipo OUT. El runtime de PL / SQL se ajustará automáticamente a la invocación correcta del subprograma por el número y tipo de parámetros.

Al añadirse esta sobrecarga, se debe asegurar que no se termina copiando y pegando el código del programa original en las nuevas sobrecargas. En su lugar, debería haber un trozo de código que actuara de núcleo del programa que todas las llamadas que se diferenciarán en la lista de parámetros que cada una de las sobrecargas toma.

Uso de "named notation" para hacer el código más claro.

Con la "named notation", el programador indica en la lista de parámetros el nombre del parámetro formal además de los valores de los parámetros actuales usando el siguiente formato:

```
formal_parameter => actual_parameter
```

La alternativa a la "named notation" es la que la mayoría de nosotros utilizamos la mayoría de las veces y se llama "positional notation" o notación posicional. Con este otro enfoque, el parámetro real se asocia implícitamente con el parámetro formal a través de su posición.

Sin embargo la "named notation" es más legible, más clara y sobre todos más flexible ya que:

- Permite cambiar el orden en el que los parámetros actuales se suministran. Es posible que se desee hacer hincapié en determinados parámetros en una llamada de atención a un programa.
- Permite utilizar el valor predeterminado de un argumento a pesar de la posición en la lista de parámetros: Con la notación posicional, se puede evitar pasar un parámetro de tipo IN que tenga un valor por defecto. Pero si es un parámetro de tipo IN OUT u OUT el que aparece en la lista después del argumento de tipo IN con un valor predeterminado, se deberá "pasar por alto" el argumento de tipo IN.

En cuanto a las desventajas de este tipo de notación podemos recoger las siguientes:

- Se necesita saber o buscar los nombres de los parámetros, es decir, es necesario escribir el nombre de éstos. Esto lleva su tiempo y rara vez lo tenemos. Por otra parte, existen editores PL/SQL que cada vez son mejores a la hora de la generación de llamadas de programa con la notación de llamada. Si dispone de uno de ellos, sería interesante usar esta funcionalidad.

- Con la notación de llamada, se está codificando los nombres de parámetro. Es decir, en el futuro, si un programador cambia un nombre de parámetro, el programa puede dejar de compilar, aunque no ha habido ningún cambio en la funcionalidad real. Esta es ciertamente un problema y por tanto hay que extremar la preocupación cuando se modifican los interfaces de los programas existentes, aunque los nombres de parámetros no es probable que cambien. Igualmente con la notación posicional, si un programador de alguna manera cambia los parámetros de orden, los errores resultantes serían más difíciles de encontrar.

Retorno de múltiples valores desde un único punto del código.

La principal misión de una función es la de devolver un valor, un solo valor, un escalar o un tipo compuesto, como un registro o una colección. Por tanto, debe ser una de nuestras principales métricas de calidad de código. Y en el caso de que la función además vuelva valores a través de la lista de parámetros con argumentos de tipo OUT o de tipo IN OUT, lo único que hacen es contribuir a que la funcionalidad de ésta sea menos clara.

Si se necesita devolver múltiples piezas de información, es importante elegir una de los siguientes métodos recomendados:

- Devolver un registro o un conjunto de valores. Es importante asegurarse de que la publicación de la estructura de su registro o colección (la declaración de tipo) se realiza en la especificación de un paquete para que los desarrolladores puedan comprender y utilizar la función con mayor facilidad.

- Romper la función en múltiples funciones que devuelvan valores escalares. Con este enfoque, se pueden llamar a las funciones dentro de sentencias SQL.

- Cambiar una función en un procedimiento. A menos que necesite llamar a una función para devolver esta información, sólo tiene que cambiar a un procedimiento que retorne múltiples piezas de información a través de los argumentos en la lista de parámetros.

Si se siguen estas pautas, los subprogramas serán más susceptibles de ser utilizados y reutilizados, ya que se definirán de manera que sean fáciles de entender y aplicar en su propio código.

Adicionalmente, estas funciones también puede ser llamarse desde de una sentencia SQL, lo que fomenta el uso aún mayor de ésta. Tenga en cuenta, sin embargo, que existen restricciones en las llamadas a funciones de SQL: no se puede llamar a una función con un argumento de tipo OUT desde dentro de SQL ni tampoco se puede llamar a una función que devuelve un registro, es decir, todos los parámetros deben ser compatibles con SQL.

Uso de procedimientos y funciones

Limitar el tamaño de una sección a no más de 50 líneas.

Cuando una sección de ejecución de código PL/SQL crece y crece, puede fácilmente llegar a tener cientos o miles de líneas. Esta situación, por típica, no deja de ser algo no deseable ya que ese código posiblemente tenga los siguientes defectos:

- Visualización: Cada vez será más difícil entender y visualizar de forma complete el flujo del programa, es decir, no podremos ver todo el árbol de decisiones.
- Confusión: Al programador cada vez le resultará más complicado entender lo que estaba realizando con ese código PL/SQL. Esta situación empeorará si el código originalmente lo hizo otro programador diferente.
- Reusabilidad: El código será difícilmente reusable en otras partes de la aplicación. Para aumentar la reusabilidad debemos centrar el foco en cuanto a funcionalidades y sobre todo, evitar efectos colaterales.

Para evitar este tipo de defectos, los programadores aplican una técnica de diseño top-down, particionando el problema en capas y realizando dicha partición en distintos pasos que van generando piezas de código pequeñas, individuales y reusables.

En resumen el proceso se basa en aplicad el siguiente conjunto de reglas:

- Localizar o crear un resumen de lo que hace el programa.
- Traducir el resumen a alto nivel en código PL / SQL.
- Crear una sección de código de ejecución de este procedimiento local o resumen.
- Crear las siguientes capas de subprogramas locales, inicialmente como esqueletos de código.
- Asegúrese de que el programa se compila.

En ese proceso, puede usarse la métrica de las 50 líneas de código por pieza. 50 suelen ser las líneas de código que cualquier editor de PL/SQL puede mostrar sin usar el scroll. De esta forma de un único golpe de vista puede visualizarse todo el árbol del flujo de programa.

El código que se genere será mucho más legible, y por lo tanto fácil de mantener. Al trabajar con una baja granularidad, pueden inferirse la intención del código con más facilidad. Además, será mucho menos probable que el código se copie y pegue, es decir, que se repita la lógica de sus programas, por lo que los subprogramas locales se convierten inmediatamente en código reutilizable.

Por tanto, se tendrá que escribir un número significativamente menor de errores, ya que mediante el empleo de refinamiento paso a paso, el

programador se centra en describir el flujo lógico de los requisitos del usuario y encontrará mucho más difícil cometer errores, descuidar alguno de los requisitos al no perderse en la complejidad del algoritmo.

Se puede definir un subprograma en cualquiera de los siguientes lugares en su código, en orden creciente de visibilidad:

- local: sólo se puede llamar dentro del bloque en el que se define.
- paquete privado: Definido en el cuerpo del paquete único, este subprograma es privado para el paquete y sólo se puede llamar por otros subprogramas de ese paquete.
- paquete público: declarado en la especificación del paquete, este subprograma se puede llamar por cualquier programa que se ejecuta a partir de un esquema que tiene la autoridad EXECUTE sobre el paquete.
- a nivel de esquema de un procedimiento o una función, que no se definen dentro de un paquete: Tiene la misma visibilidad que un subprograma del paquete público; se puede llamar por cualquier programa que se ejecuta a partir de un esquema que ha EXECUTE autoridad en este programa.

Cada vez que se crea un nuevo procedimiento o función, se debe decidir en qué nivel o qué alcance se desea hacer disponible este subprograma. Como regla para tomar la decisión se puede usar la siguiente: definir el subprograma lo más cerca posible de su uso (s).

Especificaciones claras y sin dependencias ocultas

Si desea que los desarrolladores reutilicen los programas es importante tener en cuentas las siguientes cuestiones:

- Asegúrese de que sus programas están muy enfocado en un área particular de la funcionalidad: Cada necesidad del programa debe servir a un único propósito y el nombre debe reflejar claramente ese propósito. Cuanto más variado sea el conjunto de características de un programa, es menos probable que respondan a las necesidades de los demás, y por tanto será menos reusable.
- Limitar la funcionalidad de los programas: Es importante no escribir código sólo por el gusto de hacerlo. Un mayor volumen de código sólo se traducirá en una mayor complejidad, más errores y menos tiempo para implementar la funcionalidad crítica en su aplicación.
- Evitar los efectos secundarios y acciones ocultas o globales dentro de los programas: Por ejemplo, escribir en una tabla de traza como parte de un procedimiento de negocio. Esta operación DML es un efecto secundario.

Ya que el desarrollador no se da cuenta de que la operación de DML se ejecuta hasta que encuentran un comportamiento inesperado.

Finalmente, solo recordar las consecuencias de poner sentencias DML dentro de un subprograma:

- La operación DML no puede ser llamado dentro de una consulta SQL a menos que sea una transacción autónoma.
- La operación DML se convierte en una parte de la transacción de la aplicación, de nuevo, a menos que el subprograma es una operación autónoma.
- La operación DML puede ralentizar la operación de negocio.

Por lo tanto, la recomendación es no añadir operaciones DML a un programa sin una consideración previa. De hacerlo, sin duda deben quedar documentadas y en ningún momento se debe "esconder" estas operaciones DML en operaciones de otro propósito.

Una única sentencia RETURN por función

Veamos la siguiente sección de código PL/SQL:

```
FUNCTION dame_estado (cd_in IN VARCHAR2) RETURN VARCHAR2
IS
valor_return VARCHAR2 (32757);
BEGIN
valor_return :=
CASE UPPER (cd_in)
WHEN 'C' THEN 'CERRADO'
WHEN 'A' THEN 'ABIERTO'
WHEN 'I' THEN 'INACTIVO'
END;
RETURN valor_return;
END dame_estado;
```

En lugar de ejecutar una sentencia de RETURN dentro de cada cláusula IF, es mucho más apropiado declarar una variable que almacene el valor a retornar. Además, se usa una expresión CASE para establecer el valor de esta variable de retorno, y después de la última línea, la función devuelve dicho valor. Ahora no hay absolutamente ninguna posibilidad de que la función finalice sin devolver un valor.

Una buena regla a seguir al escribir programas en PL/SQL podría ser: "Un camino de entrada y una salida". En otras palabras, no debe haber una sola manera de entrar o llamar a un programa, además debe haber una única salida, sea exitosa o no la finalización de ésta. Siguiendo esta regla, se genera un código que es mucho más fácil de rastrear, depurar y mantener.

Una forma gráfica de entender esta best practices, es pensar que una función es como un embudo, todas las líneas de código deben coincidir en la última sentencia ejecutable, que no es más que un RETURN del valor correspondiente.

Finalmente, aunque es posible, es decir, la sintaxis es aceptable, el uso de una estructura como a la siguiente no es la más recomendable:

```
IF finalizado
THEN
    RETURN;
END IF;
```

En este caso, el procedimiento termina inmediatamente y devuelve el control. Tal como se ha comentado en otras secciones de este mismo documento, es una práctica no recomendable ya que da como resultado un código no estructurado que es difícil de depurar y mantener.

Nunca devuelva el valor NULL en funciones lógicas.

O dicho de otro modo, asegúrese que una función que devuelve un boolean, siempre devuelve y solo devuelve TRUE o FALSE. Hay que recordar que en Oracle, una variable de tipo boolean puede tener tres valores, TRUE, FALSE y NULL. Este último caso, que no suele ser el caso más típico, un valor NULL que se devuelva de forma deliberada, pueden ocurrir situaciones donde se genere de forma no esperada un valor NULL que finalmente es devuelto por la función, por ejemplo una sentencia SQL que no devuelve filas.

En este caso, el trozo de código que invoca a la función que puede devolver un boolean con valor NULL debe estar preparado para ese tipo de valor y para su posible comprobación:

```
IF NOT NVL (dame_titulo (titulo), TRUE)
THEN
 …
ELSE
 …
END IF;
```

Esta modificación tiene el inconveniente de que el programador tiene que recordar en cada llamada a la función anteponer el uso de la función NVL. No es deseable basarse en la memoria del usuario de la función por lo que es necesario asegurar que la función devuelve exclusivamente TRUE o FALSE dentro de la función.

Es evidente que quedan fuera de esta best practices las situaciones donde el valor NULL tiene sentido como tal y por tanto el código PL/SQL que invoca a la función deberá prepararse para dicha situación.

Uso de paquetes

Los paquetes son los pilares fundamentales de cualquier aplicación bien diseñada construida en el Oracle PL / SQL. Un paquete consiste en una especificación y un cuerpo, aunque en algunos casos el cuerpo no es

necesario. La especificación indica que un usuario lo que puede hacer con el paquete: qué programas se puede llamar, qué estructuras de datos y tipos definidos por el usuario se puede hacer referencia, y así sucesivamente. El cuerpo del paquete implementa algún programa en la especificación del paquete y así mismo puede contener de forma privada estructuras de datos y unidades de programa.

Uso de package.subprograma

Normalmente cuando se empieza a desarrollar un procedimiento o función, rara vez se plantea desde un inicio, encapsular en un paquete. Normalmente se desarrolla de forma independiente y quizá posteriormente se decida unificar todas las funciones y procedimientos similares bajo un mismo paquete.

Sin embargo, este tipo de cambios a posteriori no suelen hacerse ya que implicaría cambiar todas las llamadas del tipo subprograma a llamadas del tipo package.subprograma en todo el código, por lo que lo más común es que finalmente, las funciones y procedimientos más antiguos queden fuera del nuevo paquete, quedando si cabe, una situación peor que la inicial.

Vamos a hacer algunas recomendaciones generales de esta experiencia:

- Evite escribir procedimientos y funciones a nivel de esquema, comience siempre con un paquete. Incluso si sólo hay un programa en el paquete en el momento, es muy probable que haya más en el futuro. Por lo que "pone en el punto en el principio", y usted no tendrá que añadir más tarde.

- Use paquetes para agrupar funcionalidad relacionada: Un paquete da un nombre a un conjunto de elementos del programa: procedimientos, funciones, tipos definidos por el usuario, las declaraciones de variables y constantes, cursores, etc. Al crear un paquete para cada área distinta de la funcionalidad, se crea contenedores para que la funcionalidad. Los programas serán más fáciles de encontrar, y por lo tanto menos probable que se reinventen en diferentes lugares de su aplicación.

- Mantenga sus paquetes pequeños y muy específicos: no sirve de mucho tener sólo tres paquetes, cada uno con cientos de programas. Sin embargo, será difícil encontrar algo dentro de ese montón de código.

Uso de la sobrecarga

La sobrecarga, también conocido como polimorfismo estático en el mundo de lenguajes orientados a objetos, es la posibilidad de declarar dos o más programas en el mismo ámbito con el mismo nombre.

Las razones para el uso de la sobrecarga de los programas en un paquete son varias, veamos algunas de ellas:

- la facilidad de transmitir el funcionamiento del paquete: al existir varias funciones o procedimientos con el mismo nombre que solo se diferencian en el número y/o tipo de parámetros el usuario del paquete tiene que recordar menos nombres y por tanto su uso es más sencillo.

- la posibilidad de anticiparse a las necesidades de los futuros usuarios del paquete, al crear varias opciones evitando así que el usuario tenga que realizar costosas conversiones de tipos en tiempo real o llamadas complejas o innecesarias.

Veamos un ejemplo de sobrecarga de procedimientos y funciones sobrecargadas:

```
PACKAGE SYS.DBMS_SQL AUTHID CURRENT_USER
IS
FUNCTION execute(c IN INTEGER) RETURN INTEGER;
What I would do is add an overloading of this program,
but in a procedure form:
PACKAGE SYS.DBMS_SQL AUTHID CURRENT_USER
IS
FUNCTION execute(c IN INTEGER) RETURN INTEGER;
PROCEDURE execute(c IN INTEGER);
```

Uso de disparadores (triggers)

Los disparadores o triggers son parte esencial de una aplicación bien diseñada. Al colocar cierta parte de las necesidades del negocio en triggers, se consigue un acoplamiento casi perfecto entre los datos y las reglas de negocio. De esta forma, por ejemplo, se garantiza que dichas reglas son siempre aplicadas a cualquier operación realizada sobre los objetos de la base de datos.

Cláusula FOLLOWS

Existen situaciones donde existe dependencias entre dos o más triggers, y por tanto, el orden en el que se disparan es esencial. Este tipo de situaciones, lo más sencillo es recurrir a fusión de dichos triggers creando grande piezas de código que difícilmente se pueden mantener dada la cantidad de reglas que pueden albergar.

Ante este tipo de situaciones, la best practice pasa por usar la nueva funcionalidad de Oracle RDBMS 11g, la clausula FOLLOWS. Esta funcionalidad permite establecer el orden de ejecución de los triggers.

```
CREATE OR REPLACE TRIGGER TEST_TRIGGER_2
FOLLOWS TEST_TRIGGER_1
.... CÓDIGO PL/SQL ....
END;
```

En el ejemplo anterior el trigger test_trigger_2 siempre se ejecutará después de test_trigger_1.

Triggers compuestos en Oracle Database 11g

Sobre algunas tablas se tiene la necesidad de crear múltiples triggers con gran cantidad de lógica y reglas de negocio. En esta situación, se suele recurrir a un paquete para almacenar todo el código relacionado de forma conjunta y para disponer de una forma de compartir información entre los distintos triggers a través de variables de sesión asociadas al paquete. Por el contrario, este uso de variables globales al paquete puede ser contraproducente, ya que, por ejemplo, ante una excepción durante el disparo del trigger, provoca que no se realice correctamente las tareas de limpieza de estas variables globales y por tanto existe un memory leak.

Oracle RDBMS 11g permite la definición de triggers compuesto que permiten mover toda la lógica almacenada en paquete directamente a un de este tipo de trigger que combine todas las funcionalidades necesarias: Este tipo de trigger tiene 4 secciones diferentes:

- BEFORE STATEMENT: Esta parte consolida todo el código que necesita ejecutarse antes de que la sentencia sea ejecutada. Es el lugar correcto para realizar funciones de cacheo de datos.

- BEFORE ROW: Consolida todo el código que necesita ejecutarse antes de cada fila afectada por la ejecución de la sentencia.

- AFTER ROW: Consolida todo el código que necesita ejecutarse después de cada fila afectada por la ejecución de la sentencia. En el caso de que se quiera evitar el efecto de tabla mutante, puede usarse para extraer la fila a una variable o cache.

- AFTER STATEMENT: Esta parte consolida todo el código que necesita ejecutarse después de que la sentencia sea ejecutada. Suele ser el lugar para colocar el código que no afecta a nivel de fila o que puede provocar el efecto de tabla mutante.

```
CREATE TRIGGER TGR_CLIENTES
BEFORE UPDATE ON TBL_CLIENTES
COMPOUND TRIGGER
/*
DECLARACIÓN DE VARIABLES A LAS QUE PUEDE ACCEDERSE DESDE
CUALQUIER PARTE DEL CUERPO DEL TRIGGER.
*/
BEFORE STATEMENT IS
BEGIN
...
END BEFORE STATEMENT;
BEFORE ROW IS
BEGIN
...
END BEFORE ROW;
AFTER ROW IS
BEGIN
...
```

```
END AFTER ROW;
AFTER STATEMENT IS
BEGIN
...
END AFTER STATEMENT;
END TGR_CLIENTES;
```

En ejemplo anterior, también pueden usarse las funciones de INSERTING, UPDATING y DELETING dentro de cualquiera de las secciones con el fin de crear la lógica de negocio que necesitemos.

Con esta nueva funcionalidad, ya no es necesario usar variables o collection a nivel de paquetes para comunicar información entre distintos triggers. En este caso podemos definir una colección, por ejemplo, en la zona de declaración del triggers, tal como se muestra a continuación.

```
CREATE TRIGGER TGR_CLIENTES
BEFORE UPDATE ON TBL_CLIENTES
COMPOUND TRIGGER
TYPE T_COLECCION IS TABLE OF TBL_CLIENTES%ROWTYPE
INDEX BY TBL_CLIENTES.PK%TYPE;
COLECCION T_COLECCION
BEFORE STATEMENT IS
....
```

De esta forma se puede referencia cualquier variable definida dentro de un trigger compuesto. En el caso de una excepción durante el disparo de este trigger, la memoria asociada a estas variables es automáticamente liberada.

Reglas de negocio a bajo nivel a través de triggers

Normalmente en un sistema de información no existe una única forma de introducir o consultar los datos, es decir, existen varios aplicativos que realizan estas tareas.

En estas situaciones, es importante asegurar las reglas de negocio en todas las situaciones, si no, la corrupción lógica de los datos está asegurada. Para ello podemos valernos de triggers en la propia base de datos para asegurar la validación al más bajo nivel posible. De esta forma, nos será indiferente como lleguen los datos a la base de datos, todos cumplirán nuestras reglas de negocio al no ser posible la no validación de estos.

```
CREATE TRIGGER TGR_CLIENTES
BEFORE UPDATE ON TBL_CLIENTES
FOR EACH ROW
BEGIN
IF REGLA_NEGOCIO_NO_CUMPLIDA THEN
/*
GENERAR ESTADO DE ERROR Y DEVOLVERLO AL USUARIO
*/
...
END;
```

Campos calculados a través de triggers.

La misma lógica aplica a los campos calculados, es decir aquellos almacenados en la fila que independientemente de cómo son insertados, actualizados o eliminados, son transformados sobre la misma columna o sobre otra diferente.

Para garantizar la consistencia de estos cálculos y evitar corrupciones lógicas de los datos en el caso de múltiples aplicaciones, la recomendación y es usar triggers para realizarlos.

De esta forma, el trigger asegura tanto la aplicación para todas las filas y columnas como el correcto cálculo.

```
CREATE OR REPLACE TRIGGER TGR_CLIENTES
BEFORE UPDATE OR INSERT ON TBL_CLIENTES
FOR EACH ROW
BEGIN
:NEW.UPPER_NOMBRE := UPPER (:NEW.NOMBRE);
END;
```