Índice

1
Introducción

1. Introducción

Las prácticas de las asignaturas de Bases de Datos se van a realizar utilizando el Sistema de Gestión de Bases de Datos Relacional (RDBMS) **ORACLE**. Varias son las razones que justifican la impartición de estas prácticas utilizando ORACLE. En primer lugar, ORACLE es un producto comercial ampliamente extendido y utilizado, que cuenta con una importante cuota de mercado dentro del mundo de las bases de datos, estando disponible para prácticamente la totalidad de plataformas posibles (Windows, MAC, UNIX, LINUX, ...) con la ventaja de que las aplicaciones realizadas para una plataforma concreta pueden ser *portadas* de forma automática a cualquiera de las otras plataformas. ORACLE permite almacenar gran cantidad de información y su posterior manejo de forma rápida y segura, destacando además su valor educativo, ya que la herramienta que utiliza ORACLE para acceder a la base de datos es el lenguaje no procedural SQL, y este lenguaje es *relacionalmente completo*, es decir, implementa prácticamente toda la funcionalidad y características del modelo relacional teórico.

2. El RDBMS ORACLE

ORACLE como todo sistema de base de datos está constituido por los datos, esto es, un conjunto de ficheros que contienen la información que forma la base de datos, y por un software encargado de manipular la base de datos llamado RDBMS. Pero, además, ORACLE proporciona una serie de herramientas para trabajar con la base de datos, algunas de ellas son:

- SQL*PLUS: es una herramienta de programación y consulta que permite a los usuarios la manipulación directa de la información de la base de datos usando el lenguaje SQL.

- DEVELOPER/2000: es un conjunto de programas clientes que se ejecutan bajo plataforma WINDOWS y que permiten crear de una manera cómoda y rápida aplicaciones clientes ejecutables bajo WINDOWS para acceder, manipular y mostrar la información almacenada en un servidor ORACLE.

- Precompiladores: es un conjunto de utilidades que permiten insertar dentro de programas escritos en lenguajes de programación tradicionales (C, PASCAL, COBOL, ...) sentencias SQL y bloques PL/SQL (lenguaje estructurado de Oracle).

- SQL*DBA: es la utilidad que permite realizar las tareas de administración de la base de datos.

En los capítulos posteriores se van a estudiar, en primer lugar el lenguaje de programación SQL, estándar para el acceso a base de datos; posteriormente ampliaremos las posibilidades de programación del servidor ORACLE mediante el estudio del lenguaje PL/SQL que permite una

programación avanzada a través de la utilización de disparadores y prodecimientos almacenados; finalmente se desarrollará un estudio a modo de tutorial de parte de DEVELOPER/2000, en concreto de FORMS 4.5 y REPORTS 2.5, lo que permitirá desarrollar cualquier aplicación clásica de gestión de bases de datos en entorno cliente/servidor y dentro del marco de las herramientas proporcionadas por Oracle.

3. Nociones básicas de SQL*Plus

PLUS31 es un *cliente* SQL para bases de datos Oracle. El *servidor* Oracle se encuentra instalado en un servidor aparte. En este servidor reside el RDBMS Oracle (software + base de datos física) y el software de red (SQL*Net).

Una vez ejecutado el programa aparece en pantalla la ventana **LOG ON**, la cual pide los parámetros necesarios para realizar la conexión al servidor Oracle. Estos parámetros son el nombre de usuario, el password y la *cadena de conexión* que establece el protocolo de comunicación que se va a utilizar, la dirección de la máquina que corre el servidor Oracle y el nombre de la base de datos a la que se quiere acceder (a partir de SQL*Net v. 2 esta cadena de conexión queda enmascarada con un alias que se establece en el fichero **tnsnames.ora** que debe aparecer tanto en el servidor como en cada cliente).

Existen dos comandos básicos de SQL*Plus, no de SQL, que son **host** y **start**. El comando HOST sirve para ejecutar comandos del sistema operativo sin necesidad de abandonar ORACLE. Un ejemplo de este comando sería **host dir**. Si se ejecuta el comando **host** sin ningún argumento, se abre una ventana shell del sistema operativo que se cerrará al ejecutar **exit**.

El comando START (se puede abreviar utilizando el símbolo **@**) permite ejecutar ficheros que contengan conjuntos de comandos SQL ó SQL*Plus, ahorrando el tener que volver a introducirlos uno a uno. Para editar estos ficheros de comandos se puede utilizar cualquier editor de textos. Estos ficheros de comandos suelen llevar la extensión **.SQL**.

4. Usuarios de la base de datos, cambio de contraseña

Oracle es un RDBMS multiusuario, lo que significa que para poder acceder a una base de datos Oracle se debe tener cuenta en ella. Cada usuario dentro de la base de datos tiene un **nombre de usuario** (username) y una **palabra clave** (password).

La creación de usuarios dentro de la base de datos es una de las tareas del DBA (Administrador de la Base de Datos). Cuando el DBA añade un usuario a la base de datos, establece su username y su password, aunque el nuevo usuario puede posteriormente cambiar el password utilizando el comando SQL `ALTER USER`.

La forma de añadir usuarios a la base de datos y establecer los privilegios de que van a gozar es mediante el comando SQL `GRANT`. Una manera sencilla de crear un nuevo usuario con nombre de usuario **user1**, password **pp100** y con privilegios de conexión y de creación de objetos es:

```
GRANT connect, resource
TO user1
IDENTIFIED BY pp100;
```

Para tener un control más detallado (asignación de quotas, recursos, privilegios, ...) de la creación de usuarios se utiliza el comando `CREATE USER`. En un momento determinado puede interesar eliminar algún privilegio de un usuario concreto, para ello se utiliza el comando de SQL `REVOKE`.

Si el usuario con username **user1** quiere cambiar su password estableciéndolo como **pp453**, debe ejecutar el siguiente comando SQL desde SQL*Plus:

```
ALTER USER user1
IDENTIFIED BY pp453;
```

5. El Diccionario de Datos de ORACLE

El diccionario de datos (**DD**) de ORACLE es uno de los componentes más importantes del DBMS ORACLE. Contiene toda la información sobre las estructuras y objetos de la base de datos así como de las tablas, columnas, usuarios, ficheros de datos, etc. Los datos guardados en el diccionario de datos son también llamados metadatos. Aunque el DD es usualmente del dominio del administrador de base de datos (DBA), es una valiosa fuente de informacion para los usuarios y desarrolladores. El DD consta dos niveles:

- *nivel interno*: contiene todas las tablas base que son utilizadas por el SGBD y que no son normalmente accesibles para el usuario,

- *nivel externo*: proporciona numerosas vistas de estas tablas base para acceder a la información sobre objetos y estructuras a diferentes niveles con detalle.

5.1. Tablas del Diccionario de Datos

La instalación de una base de datos ORACLE siempre incluye la creación de tres usuarios standard ORACLE:

- SYS: este es el propietario de todas las tablas del DD y de las vistas. Esta utilidad tiene el enorme privilegio de manejar objetos y estructuras de una base de datos ORACLE así como crear nuevos usuarios.

- SYSTEM: es el propietario de las tablas usadas por las diferentes herramientas como SQL*Forms, SQL*Reports etc.

- PUBLIC: este es un usuario “dominante” en una base de datos ORACLE. Todos los privilegios asignados a este usuario son automáticamente asignados a todos los usuarios que se conocen en la base de datos.

Las tablas y vistas que proporciona el DD contienen información sobre:

- Usuarios y sus privilegios
- Tablas, columnas de tabla y sus tipos de datos, restricciones de integridad e índices
- Estadísticas sobre tablas e índices usados
- Privilegios concedidos a los objetos de la BD
- Estructuras de almacenamiento de la BD

El comando SQL

```
SELECT  *  FROM DICTIONARY;
```

lista todas las tablas y vistas del diccionario de base de datos que son accesibles para el usuario. La información seleccionada incluye el nombre y una corta descripción de cada tabla y vista.

La consulta

```
SELECT *  FROM TABS;
```

recupera los nombres de todas las tablas pertenecientes al usuario que emite este comando.

La consulta

```
SELECT * FROM COL;
```

devuelve toda la información sobre las columnas de algunas tablas.

Cada operación SQL requiere de varios accesos directos a las tablas y vistas del DD. Puesto que el diccionario de datos en sí mismo consiste en tablas, ORACLE tiene que generar numerosas sentencias SQL para chequear si el comando SQL emitido por el usuario es correcto y puede ser ejecutado. Por ejemplo, la consulta SQL

```
SELECT * FROM EMPLEADO  WHERE SALARIO> 2000;
```

requiere la verificación de si (1) la tabla EMPLEADO existe, (2) el usuario tiene el privilegio de acceder a esta tabla, (3) la columna SALARIO está definida en esta tabla, etc.

5.2. Las vistas del Diccionario de Datos

El nivel externo del DD proporciona un interface para acceder a la información relevante para los usuarios. Este nivel tiene numerosas vistas (en ORACLE7 aproximadamente 540) que representan (una porción de) los datos de las tablas base del DD de un modo legible y entendible. Estas vistas pueden utilizarse en consultas SQL como si fueran tablas normales.

Las vistas proporcionadas por el DD se dividen en tres grupos: USER, ALL y DBA. Los nombres de los grupos construyen el prefijo para cada nombre de tabla. Para algunas vistas, hay sinonimos asociados (descritos en las tablas adjuntas).

- **USER_**: las tuplas en la vista USER contienen información sobre objetos pertenecientes al usuario que realiza la consulta SQL (usuario actual).

USER_TABLES	Todas las tablas con su nombre, numero de columnas, información almacenada, etc. (TABS)
USER_CATALOG	Tablas, vistas y sinónimos (CAT)
USER_COL_COMMENTS	Comentarios en columnas
USER_CONSTRAINTS	Definiciones de restricciones para tablas
USER_INDEXES	Información sobre indices creados para las tablas (IND)
USER_OBJECTS	Todos los objetos de la BD propiedad del usuario(OBJ)
USER_TAB_COLUMNS	Columnas de las tablas y vistas propiedad del usuario (COLS)
USER_TAB__COMMENTS	Comentarios sobre tablas y vistas
USER_TRIGGERS	Disparadores definidos por el usuario
USER_USERS	Información sobre el usuario actual

USER_VIEWS	Vistas definidas por el usuario

• **ALL_**: las filas de las vistas ALL contienen idéntica información que las vistas USER pero de todos los usuarios. La estructura de estas vistas es análoga a la estructura de las vistas USER.

ALL_TABLES	Propietario, nombre de todas las tablas accesibles
ALL_CATALOG	Propietario, tipo y nombre de todas las tablas accesibles, vistas y sinónimos.
ALL_OBJECTS	Propietario, tipo y nombre de todos los objetos accesibles de la BD
ALL_TRIGGERS	….
ALL_USERS	….
ALL_VIEWS	….

• **DBA_**: Las vistas DBA incluyen información sobre todos los objetos de la BD sin tener a los propietarios de dichos objetos. Sólo los usuarios con privilegios DBA pueden acceder a estas vistas.

DBA_TABLES	Tablas de todos los usuarios de la BD
DBA_CATALOG	Tablas vistas y sinónimos definidos en la BD
DBA_OBJECTS	Objetos de todos los usuarios
DBA_DATA_FILES	Información sobre los ficheros de datos
DBA_USERS	Información sobre los usuarios de la BD

2
SQL: Lenguaje de Consulta Estructurado

1. Introducción

SQL (**S**tructured **Q**uery **L**anguage, Lenguaje de Consulta Estructurado) es un lenguaje de programación estándar para el acceso a bases de datos. La mayoría de bases de datos comerciales permiten ser accedidas mediante SQL, siendo mínimas las diferencias entre las distintas implementaciones. El lenguaje SQL que va a ser estudiado es el implementado en la versión 7 del RDBMS Oracle.

A diferencia de otros lenguajes de programación, programar en SQL es relativamente sencillo, debido principalmente a que es un lenguaje no procedural, es decir, se dice QUÉ es lo que se quiere hacer pero no CÓMO hacerlo; además de la sencillez conceptual del modelo relacional en que se basa SQL.

SQL, como cualquier lenguaje de base de datos, se divide básicamente en:

- Lenguaje de Definición de Datos (DDL): proporciona órdenes para definir, modificar o eliminar los distintos objetos de la base de datos (tablas, vistas, índices...).

- Lenguaje de Manipulación de Datos (DML): proporciona órdenes para insertar, suprimir y modificar tuplas de las tablas de la base de datos.

- Lenguaje de Control de Datos (DCL): que permite establecer derechos de acceso de los usuarios sobre los distintos objetos de la base de datos.

Existen básicamente dos formas de utilizar SQL para acceder a una base de datos: a través de un intérprete SQL conectado a la base de datos (como por ejemplo SQL*Plus de Oracle), o bien insertando código SQL dentro de programas escritos en otros lenguajes como C o COBOL. A este último SQL se le denomina *SQL empotrado* y requiere de un preproceso que convierta los comandos SQL en llamadas a librerías propias de ese lenguaje de programación.

Evidentemente no es el objetivo de esta publicación dar un manual completo del lenguaje, sino estudiar los comandos fundamentales que permitan implementar cualquier aplicación estándar. En la bibliografía se proponen varios libros que profundizan en todos los detalles del lenguaje.

NOTA: la notación utilizada para la especificación de los comandos de SQL es la siguiente:

- palabras clave en mayúsculas,
- los corchetes `[]` indican opcionalidad,
- las llaves `{}` delimitan alternativas separadas por `|` de las que se debe elegir una,
- los puntos suspensivos `...` indican repetición varias veces de la opción anterior.

2. Tipos de datos y conversión entre tipos

Los tipos de datos principales de ORACLE son los siguientes:

- **`VARCHAR(n)`**: dato de tipo carácter, de `n` caracteres de longitud.
- **`NUMBER`**: dato de tipo numérico de un máximo de 40 dígitos, además del signo y el punto decimal. Se puede utilizar notación científica (1.273E2 es igual a 127.3).
- **`NUMBER(n,d)`**: dato de tipo numérico con `n` dígitos en total como máximo y `d` dígitos decimales como mucho. `NUMBER(4,2)` tiene como máximo valor 99.99.
- **`DATE`**: datos de tipo fecha.

Cadenas de caracteres (`VARCHAR(n)`):

Se delimitan utilizando comillas simples: `'Hola'`, `'Una cadena'`. Además de los operadores de comparación e igualdad (<, >, =, !=, ...) otras funciones útiles para trabajar con cadenas son:

- `cad || cad` : concatena dos cadenas.
- `LENGTH(cad)` : devuelve la longitud de la cadena.
- `LOWER(cad)` : convierte todas las letras de la cadena a minúsculas.
- `UPPER(cad)` : ídem a mayúsculas.
- `SUBSTR(cad, comienzo [, cuenta])`: extrae la subcadena de cad empezando en la posición comienzo y con longitud (la subcadena) cuenta. El primer caracter de la cadena tiene como índice el número 1.

Números (`NUMBER`):

Además de las operaciones típicas con valores numéricos (+, -, *, /), otras funciones útiles son:

- `ABS(num)` : devuelve el valor absoluto.
- `SQRT(num)` : devuelve la raíz cuadrada.
- `POWER(b,e)` : devuelve la potencia b^e.
- `GREATEST(num1, num2, ...)` : devuelve el mayor valor de la lista de valores.
- `LEAST(num1, num2, ...)` : devuelve el menor valor de la lista.

Existen otras funciones para grupos de valores (suma, media, máximo, ...) que se verán en apartados posteriores.

Fechas (DATE):

El formato de una valor de tipo DATE es: 'dia-mes-año', donde tanto el día como el año tiene formato numérico y el mes se indica con las tres primeras letras del nombre del mes en el idioma soportado por el servidor ORACLE. Ejemplos: '1-JAN-96', '28-jul-74'. Además de esta información, un valor de tipo fecha almacena también la hora en formato hh:mm:ss. Las fechas se pueden comparar con los operadores típicos de comparación (<, >, !=, =, ...). La función SYSDATE devuelve la fecha actual (fecha y hora). Con las fechas es posible realizar operaciones aritméticas como sumas y restas de fechas, teniendo en cuenta que a una fecha se le suman días y que la diferencia entre dos fechas se devuelve también en días. Por ejemplo SYSDATE + 1 devuelve la fecha de mañana.

Oracle permite tanto la conversión de tipos implícita como la explícita. La conversión de tipos implícita significa que cuando Oracle encuentra en un lugar determinado (por ejemplo en una expresión) un dato de un tipo diferente al esperado, entonces aplica una serie de reglas para intentar convertir ese dato al tipo esperado. Por ejemplo, si un atributo de una tabla determinada es de tipo NUMBER y se intenta introducir el valor de tipo caracter '1221', entonces automáticamente se convierte en su valor numérico equivalente sin producirse ningún error.

La conversión de tipos explícita se realiza básicamente con las siguientes funciones:

- Conversión número-cadena: TO_CHAR(número [, formato]).
- Conversión cadena-número: TO_NUMBER(cadena [,formato]).
- Conversión fecha-cadena: TO_CHAR(fecha [, formato]).
- Conversión cadena-fecha: TO_DATE(cadena [, formato]).

La opción formato permite especificar un modelo de formato o máscara consistente en una cadena de caracteres que describe el formato en el que se quiere obtener el resultado o en el que se da el parámetro. Algunos ejemplos de la utilización de estas funciones son:

- TO_CHAR('25-dec-98', 'YY') devuelve '98'.
- TO_CHAR(SYSDATE, 'dd-mon-yyyy') devuelve '25-dec-1998'.
- TO_CHAR(123.34,'09999.999') devuelve '00123.340'.

3. Expresiones y operadores condicionales

Las condiciones son expresiones lógicas (devuelven verdadero o falso) que se sitúan normalmente junto a una cláusula SQL que utilizan muchos comandos, la cláusula **WHERE**. La cláusula WHERE selecciona un subconjunto de tuplas, justo aquellas que cumplen la condición especificada. Una condición también puede aparecer en otras cláusulas de determinados comandos SQL, como por ejemplo en la cláusula CHECK que sirve para establecer condiciones sobre los valores almacenados en una tabla.

Las condiciones se construyen utilizando los operadores de comparación y los operadores lógicos. A continuación se describen los operadores más importantes junto con ejemplos de su utilización.

- **=, <>, <=, >=, < y >**.

Ejemplos:

- `horas >= 10.5`
- `nombre = 'PEPE'`
- `fecha < '1-ene-93'`

• **`[NOT] IN lista_valores`**: Comprueba la pertenencia a la lista de valores. Generalmente, los valores de la lista se habrán obtenido como resultado de un comando `SELECT` (comando de consulta).

Ejemplo:

- `nombre NOT IN ('PEPE', 'LOLA')`

• **`oper {ANY | SOME} lista_valores`**: Comprueba que se cumple la operación `oper` con algún elemento de la lista de valores. `oper` puede ser **<**, **>**, **<=**, **>=**, **<>**.

Ejemplo:

- `nombre = ANY ('PEPE', 'LOLA')`

• **`oper ALL lista_valores`**: Comprueba que se cumple la operación `oper` con todos los elementos de la lista de valores. `oper` puede ser **<**, **>**, **<=**, **>=**, **<>**.

Ejemplo:

- `nombre <> ALL ('PEPE', 'LOLA')`

• **`[NOT] BETWEEN x AND y`**: Comprueba la pertenencia al rango `x - y`.

Ejemplo:

- `horas BETWEEN 10 AND 20` que equivale a h`oras >= 10 AND horas <= 20`

• **`[NOT] EXISTS lista_valores`**: Comprueba si la lista de valores contiene algún elemento.

Ejemplos:

- `EXISTS ('ALGO')` devuelve verdadero.
- `NOT EXISTS ('ALGO')` devuelve falso.

• **`[NOT] LIKE`**: Permite comparar cadenas alfanuméricas haciendo uso de símbolos comodín. Estos símbolos comodín son los siguientes:

_ : sustituye a un único carácter.
%: sustituye a varios caracteres.

Ejemplos:

- `nombre LIKE 'Pedro%'`
- `codigo NOT LIKE 'cod1_'`

Si dentro de una cadena se quieren utilizar los caracteres `'%'` o `'_'` tienen que ser *escapados* utilizando el símbolo `'\'`.

- **IS [NOT] NULL**: Cuando el valor de un atributo, o es desconocido, o no es aplicable esa información, se hace uso del valor nulo (NULL). Para la comparación de valores nulos se utiliza el operador `IS [NOT] NULL`.

  Ejemplo:

  - `telefono IS NULL`

Los operadores lógicos junto con el uso de paréntesis permiten combinar condiciones simples obteniendo otras más complejas. Los operadores lógicos son:

- **OR**: `nombre = 'PEPE' OR horas BETWEEN 10 AND 20`
- **AND**: `horas > 10 AND telefono IS NULL`
- **NOT**: `NOT (nombre IN ('PEPE','LUIS'))`

4. Creación, modificación y destrucción de TABLAS

Los tres comandos SQL que se estudian en este apartado son `CREATE TABLE`, `ALTER TABLE` y `DROP TABLE`, pertenecientes al DDL. Estos comandos permiten respectivamente crear y modificar la definición de una tabla y eliminarla de la base de datos.

4.1. Creación de TABLAS

Para la creación de tablas con SQL se utiliza el comando **CREATE TABLE**. Este comando tiene una sintaxis más compleja de la que aquí se expone, pero se van a obviar aquellos detalles que quedan fuera del ámbito de esta publicación.

La sintaxis del comando es la siguiente:

```
CREATE TABLE nombre_tabla (
   {nombre_columna tipo_datos [restricción_columna]
    ...
    | restricción_tabla}
   [,{nombre_columna tipo_datos [restricción_columna]
    ...
    | restricción_tabla}]
   ...
);
```

Donde **restricción_columna** tiene la sintaxis:

```
[CONSTRAINT nombre_restricción]
   {[NOT] NULL
    | {UNIQUE | PRIMARY KEY}
    | REFERENCES nombre_tabla [(nombre_columna)]
        [ON DELETE CASCADE]
    |  CHECK (condición)
   }
```

y **restricción_tabla** tiene la sintaxis:

```
[CONSTRAINT nombre_restricción]
  {{UNIQUE | PRIMARY KEY} (nombre_columna [,nombre_columna] ...)
   | FOREIGN KEY (nombre_columna [,nombre_columna] ...)
      REFERENCES nombre_tabla [(nombre_columna
                               [,nombre_columna] ...)]
         [ON DELETE CASCADE]
   |  CHECK (condición)
  }
```

El significado de las distintas opciones es:

- `UNIQUE`: impide que se introduzcan valores repetidos para ese atributo. No se puede utilizar junto con `PRIMARY KEY`.

- `NOT NULL`: evita que se introduzcan tuplas con valor NULL para ese atributo.

- `PRIMARY KEY`: establece ese atributo como la llave primaria de la tabla.

- `CHECK (condición)`: permite establecer condiciones que deben cumplir los valores introducidos en ese atributo. La condición puede ser cualquier expresión válida que sea cierta o falsa. Puede contener funciones, atributos (de esa tabla) y literales. Si un CHECK se especifica como una restricción de columna, la condición sólo se puede referir a esa columna. Si el CHECK se especifica como restricción de tabla, la condición puede afectar a todas las columnas de la tabla. Sólo se permiten condiciones simples, por ejemplo, no está permitido referirse a columnas de otras tablas o formular subconsulas dentro de un CHECK. Además las funciones **`SYSDATE`** y **`USER`** no se pueden utilizar dentro de la condición. En principio están permitidas comparaciones simples de atributos y operadores lógicos (AND, OR y NOT).

- `PRIMARY KEY lista_columnas`: sirve para establecer como llave primaria un conjunto de atributos.

- `FOREIGN KEY`: define una llave externa de la tabla respecto de otra tabla. Esta restricción especifica una columna o una lista de columnas como de clave externa de una tabla referenciada. La tabla referenciada se denomina *tabla padre* de la tabla que hace la referencia llamada *tabla hija*. En otras palabras, no se puede definir una restricción de integridad referencial que se refiere a una tabla antes de que dicha tabla haya sido creada. Es importante resaltar que una clave externa debe referenciar a una clave primaria completa de la tabla padre, y nunca a un subconjunto de los atributos que forman esta clave primaria.

- `ON DELETE CASCADE`: especifica que se mantenga automáticamente la integridad referencial borrando los valores de la llave externa correspondientes a un valor borrado de la tabla referenciada (tabla padre). Si se omite esta opción no se permitirá borrar valores de una tabla que sean referenciados como llave externa en otras tablas.

En la definición de una tabla pueden aparecer varias cláusulas `FOREIGN KEY`, tantas como llaves externas tenga la tabla, sin embargo sólo puede existir una llave primaria, si bien esta llave primaria puede estar formada por varios atributos.

La utilización de la cláusula `CONSTRAINT nombre_restricción` establece un nombre determinado para la restricción de integridad, lo cual permite buscar en el *Diccionario de Datos* de la base de datos con posterioridad y fácilmente las restricciones introducidas para una determinada tabla.

Ejemplo:

```
CREATE TABLE coches (
```

```
        mat       VARCHAR(8) CONSTRAINT pk_coches PRIMARY KEY,
        marca     VARCHAR(15),
        an_fab    NUMBER(2)
);
```

En los ejercicios de este tema se pueden encontrar más ejemplos del comando `CREATE TABLE` correspondientes a las tablas que van a ser utilizadas para ejemplificar los distintos conceptos introducidos en los temas relativos a Oracle.

El comando de SQL*PLUS **`describe nombre_tabla`** permite ver la definición de una tabla concreta de la base de datos. En esta descripción aparecerán los nombres de los atributos, el tipo de datos de cada atributo y si tiene o no permitidos valores `NULL`. Toda la definición y restricciones de una tabla creada con el comando `CREATE TABLE` pueden ser consultadas a través de las siguientes tablas y vistas del *Diccionario de Datos* de Oracle:

- `USER_TABLES`: almacena toda la información de almacenamiento físico relativa a una tabla.
- `USER_CONSTRAINTS`: almacena todas las restricciones de integridad definidas por un usuario concreto.
- `USER_CONS_COLUMNS`: almacena las restricciones de integridad definidas sobre cada atributo.

4.2. Modificación de TABLAS

Para modificar la definición de una tabla se utiliza el comando **`ALTER TABLE`**. Las posibles modificaciones que se pueden hacer sobre la definición de una tabla son añadir un nuevo atributo o modificar uno ya existente. Estas dos operaciones se podrán realizar pero teniendo en cuenta una serie de restricciones que posteriormente se detallarán. La sintaxis que a continuación se expone es también restringida, por las mismas razones expuestas para `CREATE TABLE`.

```
ALTER TABLE nombre_tabla
   [ADD { nombre_columna tipo [restricción_columna] ...
        | restricción_tabla ...}]
   [MODIFY {nombre_columna [tipo] [restricción_columna] ...];
```

La opción `ADD` permite añadir un nuevo atributo a la tabla, con un valor NULL inicial para todas las tuplas. `MODIFY` cambia la definición del atributo que ya existe en la tabla. Las restricciones a tener en cuenta son las siguientes:

- `restricción_columna` sólo puede ser NOT NULL.
- Se puede cambiar el tipo del atributo o disminuir su tamaño sólo si todas las tuplas tienen en ese atributo el valor NULL.
- Un atributo NOT NULL únicamente se puede añadir a una tabla sin tuplas.
- Un atributo ya existente sólo se puede hacer NOT NULL si todas las tuplas tiene en ese atributo un valor distinto de NULL.
- Cuando se modifica un atributo, todo lo no especificado en la modificación se mantiene tal y como estaba.

Ejemplos:

- ```
  ALTER TABLE  coches
    ADD (modelo VARCHAR(15));
  ```

- ```
  ALTER TABLE  coches
    ADD (CONSTRAINT pk_coches primary key (mat));
  ```

- ```
  ALTER TABLE coches
    MODIFY (an_fab VARCHAR(4));
  ```

4.2.1. Habilitación/deshabilitación de restricciones de integridad

Si una restricción de integridad se define mediante el comando `CREATE TABLE` o se añade con el comando `ALTER TABLE`, la condición se habilita automáticamente. Una restricción puede ser posteriormente deshabilitada con el comando `ALTER TABLE`:

```
ALTER TABLE nombre_tabla DISABLE
   {CONSTRAINT nombre_constraint
    | PRIMARY KEY
    | UNIQUE [lista_columnas]
   }
   [CASCADE];
```

Para deshabilitar una clave primaria, han de deshabilitarse previamente todas las restricciones de clave externa que dependan de esa clave primaria. La cláusula `CASCADE` deshabilita automáticamente las restricciones de clave externa que dependen de la (deshabilitada) clave primaria.

Ejemplo: Deshabilitar la clave primaria de la tabla `coches`:

```
ALTER TABLE coches DISABLE PRIMARY KEY CASCADE;
```

Para habilitar las restricciones de integridad, se utiliza la cláusula `ENABLE` en lugar de `DISABLE`. Una restricción puede ser habilitada de forma satisfactoria sólo si no hay ninguna tupla que viole la restricción que se desea habilitar. En el caso de esto ocurra, aparece un mensaje de error. Nótese que para habilitar/deshabilitar una restricción de integridad es importante haber dado nombre a todas las restricciones.

4.3. Eliminación de TABLAS

Para eliminar una tabla de la base de datos se utiliza el comando:

```
DROP TABLE nombre_tabla
   [CASCADE CONSTRAINTS];
```
```

La opción `CASCADE CONSTRAINTS` permite eliminar una tabla que contenga atributos referenciados por otras tablas, eliminando también todas esas referencias.

Evidentemente, toda la información almacenada en la tabla desaparecerá con ella. Si la llave primaria de la tabla es una llave externa en otra tabla y no utiliza la opción `CASCADE CONSTRAINTS`, entonces no se podrá eliminar la tabla.

Ejemplo: `DROP TABLE coches;`

5. Inserción, modificación y eliminación de DATOS

Una vez que se ha creado de forma conveniente las tablas, el siguiente paso consiste en insertar datos en ellas, es decir, añadir tuplas. Durante la vida de la base de datos será necesario, además, borrar determinadas tuplas o modificar los valores que contienen.

Los comandos de SQL que se van a estudiar en este apartado son `INSERT`, `UPDATE` y `DELETE`. Estos comandos pertenecen al DML.

5.1. Inserción de Datos

El comando **`INSERT`** de SQL permite introducir tuplas en una tabla o en una vista (estudiadas posteriormente) de la base de datos. La sintaxis del comando es la siguiente:

```
INSERT INTO {nombre_tabla | nombre_vista}
     [(nombre_columna [, nombre_columna] ...)]
  {VALUES (valor [, valor] ...)
   | sub_consulta
  };
```

Con el comando `INSERT` se añade una tupla a la tabla o a la vista. Si se da una lista de columnas, los valores deben emparejar uno a uno con cada una de estas columnas. Cualquier columna que no esté en la lista recibirá el valor NULL. Si no se da esta lista de columnas, se deberán dar valores para todos los atributos de la tabla y en el orden en que se definieron con el comando `CREATE TABLE`.

Si se elige la opción de `sub_consulta`, se introducirán en la tabla las tuplas resultantes de la subconsulta expresada como un comando `SELECT` que será estudiado posteriormente.

Ejemplos:

```
INSERT INTO coches
VALUES ('M2030KY','RENAULT',1995,'CLIO');
INSERT INTO coches (mat,marca,modelo,an_fab)
VALUES ('M2030KY','RENAULT','CLIO',1995);
```

5.2. Modificación de Datos

Para la modificación de tuplas dentro de una tabla o vista se utiliza el comando **`UPDATE`**. La sintaxis del comando es la siguiente:

```
UPDATE {nombre_tabla | nombre_vista}
   SET {nombre_col = expresión [, nombre_col = expresión, ...]
        | nombre_col [, nombre_col, ...] = (sub_consulta)
       }
   [WHERE condición];
```

Este comando modifica los valores de los atributos especificados en `SET` para aquellas tuplas que verifican `condición` (si existe). La subconsulta puede seleccionar de la tabla o vista que se está modificando (o de otras tablas o vistas), aunque debe devolver una única tupla. Si no se utiliza la cláusula `WHERE`, se modificarán todas las tuplas de la tabla o vista. Si se utiliza la cláusula `WHERE` sólo se modificarán aquellas tuplas que verifiquen la condición. Hay que tener en cuenta que las expresiones se van evaluando según se va ejecutando el comando `UPDATE`.

Ejemplos:

- El siguiente comando modifica el año de fabricación del coche con matrícula 'M2030KY':

```
UPDATE coches
SET an_fab = 1996
WHERE mat ='M2030KY';
```

- Para modificar el año de fabricación de todos los coches añadiéndoles una unidad:

```
UPDATE coches
SET an_fab = an_fab + 1;
```

5.3. Eliminación de Datos

Por último, se va a estudiar el comando que permite eliminar tuplas concretas de una tabla o vista determinada, el comando **`DELETE`**. Su sintaxis es la siguiente:

```
DELETE FROM {nombre_tabla | nombre_vista}
   [WHERE condición];
```

Si se omite la cláusula `WHERE`, este comando borrará todas las tuplas de la tabla o vista indicada, aunque ésta seguirá existiendo, pero sin ninguna tupla. Si se introduce la cláusula `WHERE`, sólo se borrarán aquellas tuplas que verifiquen la condición.

Ejemplos:

- ```
  DELETE FROM coches
     WHERE an_fab < 1910;
  ```

- ```
  DELETE FROM coches;
  ```

NOTA: **a partir de este punto se supone creada la base de datos con las tablas y datos corrrespondientes a las soluciones de los ejercicios 1 a 4 (ver apartado 8).**

6. Consultas sobre la base de datos

En este apartado se va a estudiar el comando SELECT de SQL. Se puede decir que SELECT es el comando principal del SQL, ya que tiene como objetivo prioritario el **obtener la información precisa de la base de datos**. Esta información se obtendrá de una o más tablas o vistas.

El comando SELECT, además, se utiliza dentro de otros comandos, como por ejemplo en CREATE TABLE, INSERT, UPDATE, ...

6.1. Conceptos básicos relativos al comando SELECT

La sintaxis fundamental del comando SELECT es la siguiente:

```
SELECT [ALL, DISTINCT]{[*|tabla.*],expresión[,expresión,...]}
FROM tabla [, tabla, ...]
[WHERE condición]
[ORDER BY {expresión | posición} [ASC | DESC]
          [expresión | posición} [ASC | DESC] , ...]]
[GROUP BY expresión [, expresión, ...]]
[HAVING condición];
```

Las distintas opciones tienen el siguiente significado:

- ALL: significa que se devolverán todas las tuplas que satisfagan las distintas condiciones, esta opción es la que se toma por defecto.

- DISTINCT: significa que se devolverán valores o conjuntos de valores únicos, es decir, no repetidos.

- *: indica que se realizará la proyección por todos los atributos de la/s tabla/s o vista/s indicadas en la cláusula FROM. Si se quiere que sólo se proyecte por los atributos de una tabla se utiliza tabla.*.

- expresión: se puede sustituir por:

  - nombre de un atributo

  - expresión conteniendo atributos y/o funciones.

  Si en el listado de la consulta debe aparecer un nombre distinto al del atributo o al de la expresión se puede utilizar un **ALIAS**, que consiste en una cadena de caracteres encerrada entre comillas dobles y situada junto a la expresión que sustituirá en el listado de la consulta.

- Otro tipo de **alias** es el que se utiliza para las tablas en la cláusula FROM. Estos alias se ponen junto al nombre de la tabla y se pueden utilizar en cualquier punto del comando SELECT sustituyendo así al nombre de la tabla.

- Las condiciones de las cláusulas WHERE y HAVING son cualquiera de las vistas en el apartado 3 de este capítulo (expresiones que devuelven verdadero o falso), además de algunas nuevas que se verán en el siguiente apartado.

- En la cláusula `ORDER BY`, `posición` sirve para referenciar un atributo que ocupa esa posición dentro de la cláusula `SELECT`.

Desde el punto de vista de las operaciones abstractas de consulta del modelo relacional, la cláusula `SELECT` corresponde a la operación de proyección; la cláusula `FROM` corresponde a la operación de producto cartesiano (nos permitirá hacer el JOIN); y la cláusula `WHERE` corresponde a la condición de la operación de selección.

Una típica consulta:

```
SELECT    a1, a2, ..., an
FROM      r1, r2
WHERE     p;
```

equivale a la expresión en términos de operaciones abstractas:

Proyección$_{a1, a2, ..., an}$ (Selección$_{p}$ (r_1 X r_2)), donde X es el producto cartesiano.

El resultado de un comando `SELECT` es siempre una tabla, lo que permite construir consultas complejas anidando varios `SELECT`. A continuación se explica el significado del resto de clausulas del comando `SELECT` viendo sus típicos usos.

- Obtener toda la información contenida en una tabla:

```
SELECT * FROM nombe_tabla ;
```

El asterisco indica que se quieren seleccionar todas la columnas de la tabla, y al no especificarse ninguna condición, todas la tuplas son seleccionadas.

Ejemplo: Obtener toda la información almacenada en la tabla `COCHES`:

```
SQL> SELECT * FROM coches;

MAT      MARCA              AN_FAB MODELO
-------- --------------- --------- ---------------
M3020KY  TOYOTA                1996 CARINA
J1234Z   RENAULT               1997 MEGANE
GR4321A  RENAULT               1978 5
B4444AC  PEUGEOT               1978 504
CA0000AD PEUGEOT               1996 205
GR1111AK PEUGEOT               1998 207
J9999AB  VW                    1998 BEATTLE
```

- Seleccionar únicamente algunas de las columnas, se deben especificar cuáles son estas columnas en la cláusula `SELECT`:

```
SELECT lista_nombre_columnas FROM nombre_tabla ;
```

Esta operación es equivalente a la operación proyección.

Ejemplo: Obtener, para cada coche, su marca y modelo:

```
SQL> SELECT marca,modelo FROM coches;

MARCA           MODELO
--------------- ---------------
TOYOTA          CARINA
RENAULT         MEGANE
RENAULT         5
PEUGEOT         504
PEUGEOT         205
PEUGEOT         207
VW              BEATTLE
```

• Para consultar sólo determinadas tuplas de una tabla se hace uso en el comando `SELECT` de la cláusula `WHERE condición`, donde la condición podrá venir expresada según lo visto en el punto 3 de este tema.

Ejemplos:

• Coches que no son fueron fabricados entre 1990 y 1999:

```
SQL> SELECT *
       FROM coches
       WHERE an_fab NOT BETWEEN 1990 AND 1999;

MAT      MARCA              AN_FAB MODELO
-------- --------------- --------- ---------------
GR4321A  RENAULT              1978 5
B4444AC  PEUGEOT              1978 504
```

• Marca y modelo de los coches de Granada:

```
SQL> SELECT marca,modelo
     FROM coches
       WHERE UPPER(SUBSTR(mat,1,2))='GR';

MARCA           MODELO
--------------- ---------------
RENAULT         5
PEUGEOT         207
```

• Para consultar información usando operaciones, en los comandos `SELECT` se pueden utilizar expresiones tales como operaciones aritméticas, operaciones con cadenas de caracteres.

Ejemplos:

• Obtener las dos primeras fechas de revisión ITV para los coches.

```
SQL> SELECT mat,an_fab,
       an_fab+5"Primera revision",
       an_fab+7"Segunda revision"
       FROM coches;

MAT         AN_FAB Primera revision Segunda revision
-------- --------- ---------------- ----------------
M3020KY       1996             2001             2003
J1234Z        1997             2002             2004
```

```
GR4321A       1978            1983            1985
B4444AC       1978            1983            1985
CA0000AD      1996            2001            2003
GR1111AK      1998            2003            2005
J9999AB       1998            2003            2005
```

- Obtener el número de caracteres que suman la marca y el modelo de cada coche.

```
SQL> SELECT marca,modelo,
       LENGTH(marca || modelo)"Numero caracteres"
     FROM coches;

MARCA           MODELO          Numero caracteres
--------------- --------------- -----------------
TOYOTA          CARINA                         12
RENAULT         MEGANE                         13
RENAULT         5                               8
PEUGEOT         504                            10
PEUGEOT         205                            10
PEUGEOT         207                            10
VW              BEATTLE                         9
```

- Para obtener la información de una forma ordenada, en los comandos `SELECT` se puede especificar el orden en que debe aparecer el resultado, este orden se consigue usando la opción:

```
ORDER BY {columna, posición} {ASC, DESC}
```

La ordenación puede hacerse por nombre de columna o por la posición de esa columna dentro de la cláusula `SELECT`.

Ejemplos:

- Obtener los coches ordenados por marca:

```
SQL> SELECT * FROM coches ORDER by marca;

MAT      MARCA              AN_FAB MODELO
-------- --------------- --------- ---------------
B4444AC  PEUGEOT              1978 504
CA0000AD PEUGEOT              1996 205
GR1111AK PEUGEOT              1998 207
J1234Z   RENAULT              1997 MEGANE
GR4321A  RENAULT              1978 5
M3020KY  TOYOTA               1996 CARINA
J9999AB  VW                   1998 BEATTLE
```

- Obtener los coches ordenados descendentemente por marca y para la misma marca ordenados ascendentemente por año de fabricación:

```
SQL> SELECT * FROM coches
     ORDER by marca DESC, an_fab ASC;

MAT      MARCA              AN_FAB MODELO
-------- --------------- --------- ---------------
J9999AB  VW                   1998 BEATTLE
```

```
M3020KY  TOYOTA               1996 CARINA
GR4321A  RENAULT              1978 5
J1234Z   RENAULT              1997 MEGANE
B4444AC  PEUGEOT              1978 504
CA0000AD PEUGEOT              1996 205
GR1111AK PEUGEOT              1998 207
```

- Obtener los coches ordenados por el número de caracteres que suman la marca y el modelo.

```
SQL> SELECT marca,modelo,
       LENGTH(marca || modelo)"Numero caracteres"
     FROM coches
     ORDER by 3;

MARCA           MODELO          Numero caracteres
--------------- --------------- -----------------
RENAULT         5                               8
VW              BEATTLE                         9
PEUGEOT         504                            10
PEUGEOT         205                            10
PEUGEOT         207                            10
TOYOTA          CARINA                         12
RENAULT         MEGANE                         13
```

- Para consultar información relacionada almacenada en varias tablas hay que implementar la operación **JOIN**. Se pueden realizar consultas donde intervengan más de una tabla de la base de datos, especificando las relaciones entre las distintas tablas en la cláusula `WHERE`.

Menos para el caso en el que se quiera realizar un producto cartesiano, CUANDO SE PONEN VARIAS TABLAS EN LA CLAUSULA `FROM` DEL COMANDO `SELECT`, DEBEN APARECER EN LA CLAUSULA `WHERE` LAS CORRESPONDIENTES CONDICIONES DE IGUALDAD QUE UNAN LAS DISTINTAS TABLAS (ESTO ES LO QUE REALIZA VERDADERAMENTE EL JOIN), además, pueden aparecer las condiciones que se requieran para realizar una consulta concreta.

Supongamos que las tablas `COCHES` y `TRABAJOS` tienen los siguientes datos:

COCHES:

```
MAT      MARCA              AN_FAB MODELO
-------- --------------- --------- ---------------
M3020KY  TOYOTA               1996 CARINA
GR4321A  RENAULT              1978 5
```

TRABAJOS:

```
MAT      DNI            HORAS FECHA_REP
-------- --------- --------- ---------
M3020KY  1111              1 23-FEB-96
M3020KY  2222            2.5 23-FEB-96
GR4321A  3333            2.1 01-JAN-98
M3020KY  5555              2 23-FEB-96
```

Si sólo se ponen las dos tablas en la claúsula `FROM`, entonces lo que se está realizando es un **producto cartesiano**, no un JOIN. Es decir, se generan nuevas tuplas combinando cada tupla de coches con cada tupla de trabajos, pero sin existir relación alguna en cada combinación:

```
SQL> SELECT c.mat,c.marca,c.modelo,t.mat,t.dni,t.horas
```

```
      FROM coches c,trabajos t;

MAT      MARCA      MODELO     MAT      DNI           HORAS
-------- ---------- ---------- -------- --------- ---------
M3020KY  TOYOTA     CARINA     M3020KY  1111              1
GR4321A  RENAULT    5          M3020KY  1111              1
M3020KY  TOYOTA     CARINA     M3020KY  2222            2.5
GR4321A  RENAULT    5          M3020KY  2222            2.5
M3020KY  TOYOTA     CARINA     GR4321A  3333            2.1
GR4321A  RENAULT    5          GR4321A  3333            2.1
M3020KY  TOYOTA     CARINA     M3020KY  5555              2
GR4321A  RENAULT    5          M3020KY  5555              2
```

Donde `R.x` indica que se refiere al atributo `x` de la tabla cuyo alias es `R`.

En la mayoría de los casos, cuando se requiere obtener información relacionada de varias tablas es necesario establecer la condición de unión entre ellas (JOIN). De esta manera lo que se está consiguiendo es recuperar la información inicial que fue particionada en tablas. Para el ejemplo anterior, como en la tabla `TRABAJOS` sólo se dispone de la matrícula de los coches, el JOIN con la tabla `COCHES` consigue añadir la información adicional de cada coche (tuplas en **negrita** en el resultado del producto cartesiano). Pero para ello es necesario indicar la condición de unión que es la que verdaderamente realiza el JOIN. Esta condición de unión se establece en la claúsula `WHERE`, y en la mayoría de los casos son condiciones de igualdad entre llaves primarias y llaves externas. Para el ejemplo anterior, la consulta que realiza correctamente el JOIN de `COCHES` y `TRABAJOS` y que por tanto obtiene sólo las tuplas en negrita del ejemplo anterior es:

```
SQL> SELECT c.mat,c.marca,c.modelo,t.mat,t.dni,t.horas
     FROM coches c,trabajos t
     WHERE c.mat = t.mat;

MAT      MARCA      MODELO     MAT      DNI           HORAS
-------- ---------- ---------- -------- --------- ---------
M3020KY  TOYOTA     CARINA     M3020KY  1111              1
M3020KY  TOYOTA     CARINA     M3020KY  2222            2.5
GR4321A  RENAULT    5          GR4321A  3333            2.1
M3020KY  TOYOTA     CARINA     M3020KY  5555              2
```

En los siguientes ejemplos, las condiciones de la cláusula `WHERE` que implementan el JOIN aparecen en **negrita**.

Ejemplos:

- Obtener todos los trabajos incluyendo los datos del mecánico que realiza el trabajo:

```
SQL> SELECT *
     FROM mecanicos,trabajos
     WHERE mecanicos.dni=trabajos.dni;

DNI      NOMBRE     PUESTO          P MAT      DNI           HORAS FECHA_REP
-------- ---------- --------------- - -------- --------- --------- -------
1111     ANTONIO    CHAPA           1 M3020KY  1111              1 23-FEB-96
2222     LUIS       MOTOR           0 M3020KY  2222            2.5 23-FEB-96
4444     LOLA       CHAPA           1 J1234Z   4444              7 19-MAR-97
2222     LUIS       MOTOR           0 J1234Z   2222              3 19-MAR-97
3333     PEPE       AMORTIGUACION   0 GR4321A  3333            2.1 01-JAN-98
3333     PEPE       AMORTIGUACION   0 B4444AC  3333            3.2 23-APR-96
3333     PEPE       AMORTIGUACION   0 CA0000AD 3333              8 23-APR-96
5555     LUISA      AMORTIGUACION   1 M3020KY  5555              2 23-FEB-96
```

```
6666      EMILIO     CHAPA           0 J9999AB  6666          1.2 05-MAR-98
5555      LUISA      AMORTIGUACION   1 J9999AB  5555           .7 05-MAR-98
2222      LUIS       MOTOR           0 J9999AB  2222            1 05-MAR-98
1111      ANTONIO    CHAPA           1 J1234Z   1111          2.2 19-MAR-97
3333      PEPE       AMORTIGUACION   0 GR1111AK 3333          5.5 01-JAN-98
3333      PEPE       AMORTIGUACION   0 J9999AB  3333          7.7 05-MAR-98
5555      LUISA      AMORTIGUACION   1 GR1111AK 5555          2.5 01-JAN-98
```

- Obtener parejas de mecánicos del mismo puesto:

```
SQL> SELECT m1.nombre,m1.puesto,m2.nombre,m2.puesto
     FROM mecanicos m1,mecanicos m2
     WHERE m1.puesto=m2.puesto AND
           m1.nombre < m2.nombre;

NOMBRE     PUESTO          NOMBRE     PUESTO
---------- --------------- ---------- ---------------
LUISA      AMORTIGUACION   PEPE       AMORTIGUACION
ANTONIO    CHAPA           EMILIO     CHAPA
ANTONIO    CHAPA           LOLA       CHAPA
EMILIO     CHAPA           LOLA       CHAPA
```

- Obtener las marcas y modelos de los coches reparados de CHAPA:

```
SQL> SELECT distinct marca,modelo
     FROM coches c, trabajos t,mecanicos m
     WHERE c.mat=t.mat AND t.dni=m.dni AND
           puesto='CHAPA';

MARCA          MODELO
-------------- --------------
RENAULT        MEGANE
TOYOTA         CARINA
VW             BEATTLE
```

6.2. Otros usos del comando SELECT

En este apartado se estudiarán algunos aspectos algo más avanzados del comando SELECT, pudiendo de esta forma generar consultas algo más complejas.

Existen una serie de funciones especiales, que no fueron tratadas en el apartado anterior, y que van a permitir trabajar con información relativa a no sólo una única tupla, sino a varias. Estas funciones se conocen con el nombre de *funciones agregadas o funciones de grupos de valores*. Un dato a tener en cuenta es que estas funciones agregadas ignoran el valor NULL y calculan resultados a pesar de su existencia.

A continuación se exponen algunas de las principales funciones de grupo:

- **COUNT**: devuelve el número de tuplas. Su formato es el siguiente:

```
COUNT ({* | [DISTINCT | ALL] columna})
```

`COUNT (*)`: cuenta todas las tuplas de la tabla, aunque tengan valores NULL (incluso en todos los atributos).

`COUNT (DISTINCT columna)`: devuelve el número de valores distintos para la columna, no incluyendo los nulos.

`COUNT (ALL columna)`: devuelve el número de valores de la columna, no incluyendo los NULL.

Ejemplos:

- Contar el número de tuplas de la tabla `COCHES`:

```
SQL> SELECT COUNT(*) FROM coches;

COUNT(*)
---------
        7
```

- Contar el número de puestos distintos en la tabla `MECANICOS`:

```
SQL> SELECT COUNT(distinct puesto) "Numero de puestos"
     FROM mecanicos;

Numero de puestos
-----------------
                3
```

- **`SUM`**: devuelve la suma de un conjunto de valores. El formato es:

```
SUM ([DISTINCT] columna | expresión)
```

Devuelve la suma de los valores de la columna o de los devueltos por expresión. La columna o expresión, evidentemente, debe ser numérica.

Ejemplos:

- Obtener el total de horas trabajadas por los mecánicos de CHAPA:

```
SQL> SELECT SUM(horas) "Horas de CHAPA"
     FROM mecanicos,trabajos
     WHERE puesto='CHAPA' AND
           mecanicos.dni=trabajos.dni;

Horas de CHAPA
--------------
          11.4
```

- Obtener la cantidad facturada por los mecánicos de CHAPA suponiendo que el precio de la hora de trabajo es 1000 para el año 96 y cada año se incrementa en un 5%:

```
SQL> SELECT SUM(horas*1000*
            POWER(1.05,TO_CHAR(fecha_rep,'YY')-96))
             "Total CHAPA"
     FROM mecanicos,trabajos
```

```
     WHERE mecanicos.dni=trabajos.dni AND
           puesto='CHAPA';

Total CHAPA
-----------
      11983
```

- **AVG**: calcula la media aritmética de un conjunto de valores. Su formato es:

```
AVG ([DISTINCT] columna | expresión)
```

Devuelve la media aritmética de los valores de la columna o de los devueltos por la expresión. La columna o expresión debe ser numérica.

Ejemplos:

- Obtener la media de horas trabajadas con los coches RENAULT:

```
SQL> SELECT AVG(horas)
     FROM coches,trabajos
     WHERE coches.mat=trabajos.mat AND
           marca='RENAULT';

AVG(HORAS)
----------
     3.575
```

- Obtener la media del número de caracteres formados por la marca y el modelo de cada coche:

```
SQL> SELECT AVG(LENGTH(marca||modelo)) FROM coches;

AVG(LENGTH(MARCA||MODELO))
--------------------------
                  10.285714
```

- **MAX**: calcula el máximo de un conjunto de valores. Su formato es:

```
MAX ([DISTINCT] columna | expresión)
```

Devuelve el valor máximo de la columna o la expresión.

Ejemplos:

- Obtener el número máximo de horas trabajadas sobre un coche:

```
SQL> SELECT MAX(horas)
     FROM trabajos;

MAX(HORAS)
----------
         8
```

- Obtener el precio del trabajo más caro:

```
SQL> SELECT MAX(horas*1000*
           POWER(1.05,TO_CHAR(fecha_rep,'YY')-96))
           "Maximo trabajo"
     FROM trabajos;

Maximo trabajo
--------------
       8489.25
```

- **MIN**: calcula el mínimo de un conjunto de valores. Su formato es:

```
MIN ([DISTINCT] columna | expresión)
```

Devuelve el valor mínimo de la columna o la expresión.

Ejemplos:

- Obtener la mitad del número mínimo de horas trabajadas en un coche:

```
SQL> SELECT MIN(horas)/2 FROM trabajos;

MIN(HORAS)/2
------------
         .35
```

6.2.1. Consulta de información agrupada

SQL permite obtener una tabla agrupada que contenga una fila con información resumen para cada grupo. Para obtener esta información agrupada se hace uso de la cláusula **`GROUP BY`** del comando `SELECT`. La cláusula `GROUP BY` tiene la siguiente sintaxis:

```
GROUP BY expresión [, expresión, ...]
```

donde `expresión` puede ser un atributo o una expresión conteniendo atributos.

Ejemplos:

-
```
SQL> SELECT mat,SUM(horas)
     FROM trabajos
     GROUP by mat
     ORDER by mat;

MAT      SUM(HORAS)
-------- ----------
B4444AC         3.2
CA0000AD          8
GR1111AK          8
GR4321A         2.1
J1234Z         12.2
```

```
J9999AB        10.6
M3020KY         5.5
```

Este comando devuelve para cada coche, su matrícula y la suma de horas trabajadas en dicho coche. Los trabajos se agrupan por matrícula y a cada grupo se le calcula la suma del atributo horas. Finalmente se devuelve una tupla por grupo, conteniendo la información de la matrícula y la suma de horas trabajadas.

En la siguiente tabla se muestra el proceso de generación del resultado anterior. En esta tabla se muestran las matrículas y las horas de cada trabajo ordenadas por matrícula. El proceso consiste por tanto en juntar las horas de los trabajos de igual matrícula (`GROUP BY mat`) y sumar sus horas, devolviendo la matrícula y la suma calculada.

MAT	HORAS	
B4444AC	3.2	SUM(horas) = 3.2
CA0000AD	8	SUM(horas) = 8
GR1111AK	5.5	
GR1111AK	2.5	SUM(horas) = 8
GR4321A	2.1	SUM(horas) = 2.1
J1234Z	2.2	
J1234Z	3	
J1234Z	7	SUM(horas) = 12.2
J9999AB	1	
J9999AB	7.7	
J9999AB	.7	
J9999AB	1.2	SUM(horas) = 10.6
M3020KY	1	
M3020KY	2.5	
M3020KY	2	SUM(horas) = 5.5

- Consultar el número de trabajos realizados a cada coche:

```
SQL> SELECT mat,COUNT(*)
     FROM trabajos
     GROUP by mat
     ORDER by mat;

MAT       COUNT(*)
--------- ---------
B4444AC           1
CA0000AD          1
GR1111AK          2
GR4321A           1
J1234Z            3
J9999AB           4
M3020KY           3
```

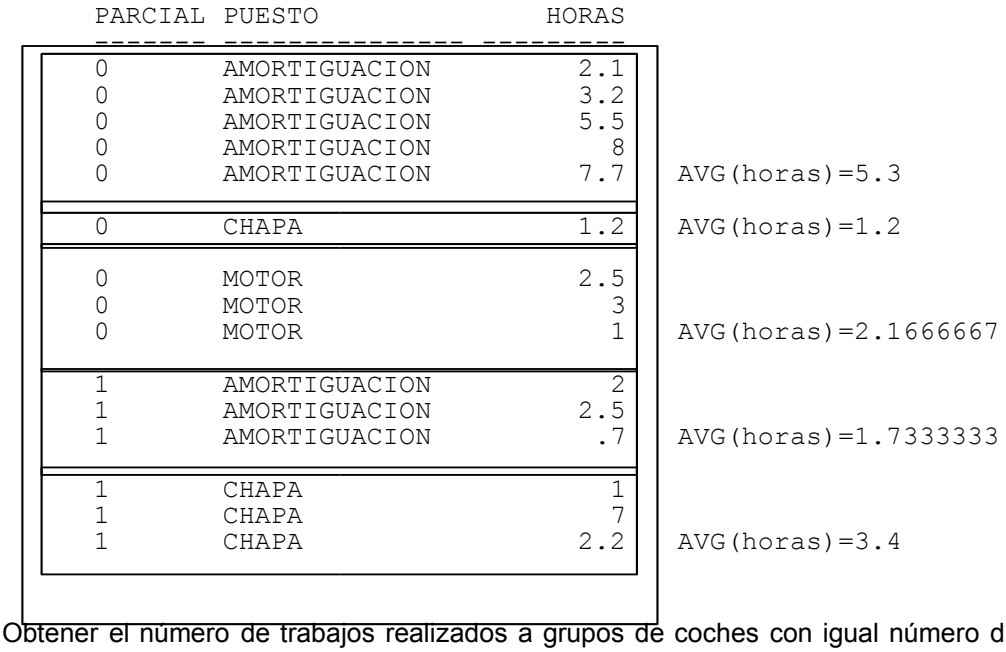
- Consultar el número medio de horas trabajadas por puesto clasificados por tipo de contrato (atributo parcial):

```
SQL> SELECT parcial,puesto,AVG(horas)
     FROM trabajos,mecanicos
```

```
      WHERE trabajos.dni=mecanicos.dni
      GROUP by parcial,puesto;
```

```
PARCIAL PUESTO          AVG(HORAS)
------- --------------- ----------
0       AMORTIGUACION          5.3
0       CHAPA                  1.2
0       MOTOR            2.1666667
1       AMORTIGUACION    1.7333333
1       CHAPA                  3.4
```

Lo que hace este comando es obtener una tupla con la información resumen de cada grupo de tuplas con el mismo valor para la pareja `(parcial, puesto)`. La forma en que se agrupan las tuplas para obtener la información resumen es la que se muestra en la siguiente tabla.



- Obtener el número de trabajos realizados a grupos de coches con igual número de caracteres sumando la marca y el modelo:

```
SQL> SELECT LENGTH(marca||modelo), COUNT(*)
        FROM coches,trabajos
      WHERE coches.mat=trabajos.mat
      GROUP by LENGTH(marca||modelo)
      ORDER by LENGTH(marca||modelo);

LENGTH(MARCA||MODELO)   COUNT(*)
--------------------- ---------
                    8         1
                    9         4
                   10         4
                   12         3
                   13         3
```

- Obtener el número máximo de horas en total trabajadas sobre un coche:

```
SQL> SELECT MAX(SUM(horas))
       FROM trabajos
     GROUP by mat;

  MAX(SUM(HORAS))
  ---------------
             12.2
```

De forma análoga a la cláusula `WHERE`, se pueden establecer restricciones sobre qué grupos aparecerán en el resultado de la consulta. Esto se realiza mediante la cláusula **`HAVING`** del comando `SELECT`.

Ejemplo: Obtener los coches en los que se ha trabajado más de 10 horas:

```
SQL> SELECT mat,SUM(horas)
     FROM trabajos
     GROUP by mat
     HAVING SUM(horas)>10;

  MAT      SUM(HORAS)
  -------- ----------
  J1234Z         12.2
  J9999AB        10.6
```

Si en una misma consulta aparecen las cláusulas `WHERE` y `HAVING`, primero se aplica la condición de la cláusula `WHERE`. Las tuplas que la satisfacen son colocadas en grupos por la cláusula `GROUP BY`, después se aplica la cláusula `HAVING` a cada grupo, y los grupos que satisfacen la condición de la cláusula `HAVING` son utilizados por la cláusula `SELECT` para generar tuplas del resultado de la consulta.

Ejemplo:

- Obtener los mecánicos de CHAPA o AMORTIGUACION que han reparado más de dos coches:

```
SQL> SELECT mecanicos.dni,
            COUNT(distinct mat) "Numero de coches"
     FROM mecanicos,trabajos
     WHERE mecanicos.dni=trabajos.dni AND
           puesto in('CHAPA','AMORTIGUACION')
     GROUP by mecanicos.dni
     HAVING COUNT(distinct mat)>2;

  DNI       Numero de coches
  --------- ----------------
  3333                     5
  5555                     3
```

Un detalle importante a tener en cuenta es que las expresiones que se pongan en la cláusula `SELECT` serán, o **valores constantes**, o **funciones agregadas**, o bien **alguna expresión incluida en la cláusula `GROUP BY`**. Esto es debido a que las funciones como `COUNT`, `SUM`, `AVG`, etc., devuelven valores sobre grupos de tuplas, no sobre tuplas individuales. En la cláusula `HAVING` sólo se podrán utilizar funciones agregadas.

Por ejemplo, si se quiere obtener el dni y el nombre de los mecánicos junto con las horas que han trabajado, la consulta:

```
SQL> SELECT m.dni,nombre,SUM(horas)
     FROM trabajos t,mecanicos m
      WHERE t.dni=m.dni
     GROUP by m.dni;
```

produciría el siguiente error:

```
   SELECT M.DNI,NOMBRE,SUM(HORAS)
                 *
   ERROR AT LINE 1:
   ORA-00979: NOT A GROUP BY EXPRESSION
```

indicando que la columna nombre no es ninguna de las expresiones incluidas en la cláusula `GROUP BY` del comando.

6.2.2. Subconsultas

Una subconsulta es una consulta dentro de otra, es decir, un `SELECT` anidado en la cláusula `WHERE` o `HAVING` de otro `SELECT`. Esta anidación permite realizar consultas complejas, que no sería posible, o poco eficiente realizar haciendo consultas simples a base de JOINS.

El formato de la subconsulta es el siguiente:

```
SELECT columnas
FROM tabla
WHERE {columna | expresión}    operador
           ( SELECT columna | expresión
             FROM ...) ;
```

El operador a utilizar dependerá del número de valores que se van a obtener en la subconsulta.

Las subconsultas pueden aparecer en cualquier lugar en el que se pueda colocar la cláusula `WHERE`, es decir en los comandos `SELECT`, `INSERT` y `UPDATE`.

Ejemplos:

- Obtener los mecánicos que pertenecen al mismo puesto que EMILIO;

```
SQL> SELECT nombre
     FROM mecanicos
     WHERE  puesto = (  SELECT puesto
                        FROM mecanicos
                        WHERE nombre = 'EMILIO')
               AND nombre!='EMILIO';

  NOMBRE
  ---------------
  ANTONIO
  LOLA
```

Esta consulta se puede hacer también con un único `SELECT` que implemente un producto cartesiano entre tuplas de la tabla de mecánicos:

```
SQL> SELECT m2.nombre
     FROM mecanicos m1,mecanicos m2
     WHERE m1.nombre='EMILIO' AND m2.puesto=m1.puesto
             AND m2.nombre!='EMILIO'
```

- Obtener los coches reparados por algún mecánico de CHAPA:

```
SQL> SELECT mat
     FROM trabajos
     WHERE dni IN (   SELECT dni
                      FROM mecanicos
                      WHERE puesto = 'CHAPA');

  MAT
  --------
  M3020KY
  J1234Z
  J9999AB
  J1234Z
```

Nótese como **cuando el resultado de la subconsulta es un conjunto de valores NO** se puede utilizar el operador de igualdad, sino el de inclusión.

6.2.3. Variables de tupla y consultas sincronizadas

SQL toma prestada la notación de variables de tupla del cálculo relacional orientado a tuplas. Una variable de tupla en SQL debe estar asociada a una tabla determinada. Las variables de tupla se definen en la cláusula `FROM` del comando `SELECT`, situándolas junto a la tabla asociada. Las variables de tupla son variables que van tomando como valores tuplas de una determinada tabla. Hasta el momento se han utilizado variables de tupla para ahorrar trabajo, dar mayor claridad al escribir las consultas y evitar ambigüedades.

Una variable de tupla se coloca después del nombre de la relación y separada por uno o más espacios en blanco. Cuando una consulta tiene subconsultas anidadas, se aplica la siguiente **regla de ámbito** para las variables de tuplas: en un subconsulta, está permitido usar sólo variables de tupla definidas en la misma subconsulta o en cualquier consulta que contenga a la subconsulta. Si existen dos variables de tupla con el mismo nombre se tiene en cuenta la más interior a la subconsulta.

Las variables de tupla son muy útiles para comparar dos tuplas de la misma relación. Por ejemplo, si se quieren obtener las parejas de mecánicos con el mismo tipo de contrato:

```
SQL> SELECT m1.nombre,m2.nombre,
            m1.parcial"Contrato parcial?"
     FROM mecanicos m1,mecanicos m2
     WHERE m1.parcial=m2.parcial AND
           m1.nombre<m2.nombre


  NOMBRE          NOMBRE          Contrato parcial?
  --------------- --------------- -----------------
  EMILIO          LUIS                            0
  LUIS            PEPE                            0
  EMILIO          PEPE                            0
  ANTONIO         LOLA                            1
  ANTONIO         LUISA                           1
  LOLA            LUISA                           1
```

Las **consultas sincronizadas** utilizan tanto subconsultas como variables de tupla. La consulta: dar los mecánicos que reparan tres o más coches, se podría resolver de la siguiente forma:

```
SQL> SELECT *
       FROM mecanicos m
       WHERE 3 <= ( SELECT COUNT(distinct mat)
                    FROM trabajos t
                    WHERE t.dni = m.dni);

   DNI       NOMBRE          PUESTO          P
   --------- --------------- --------------- -
   2222      LUIS            MOTOR           0
   3333      PEPE            AMORTIGUACION   0
   5555      LUISA           AMORTIGUACION   1
```

El hecho de llamarse sincronizadas se refiere a que se irán sustituyendo valores en `m.dni` en la subconsulta de forma sincronizada a como se van tratando en la consulta principal. Es decir, la consulta principal va analizando tupla a tupla y la subconsulta coge el valor de la tupla que se está analizando en ese momento. De esta forma, **para cada** mecánico, la subconsulta obtiene el número de coches que ha arreglado.

6.2.4. Operaciones entre conjuntos

A los resultados de las consultas (que se implementan como tablas temporales) se les pueden aplicar también los operadores tradicionales de conjuntos (unión, intersección y diferencia), obteniendo nuevas tablas. Los operadores de conjuntos son:

- **UNION**:

Realiza la operación de unión eliminando las tuplas duplicadas. Las dos tablas que se unen deben tener el mismo número de atributos (aunque no es necesario que tengan los mismos nombres) y éstos deben tener el mismo tipo de datos.

Ejemplo:

```
SQL>   (SELECT *
        FROM mecanicos
        WHERE nombre = 'PEPE')
       UNION
       (SELECT *
        FROM mecanicos
        WHERE puesto = 'AMORTIGUACION');

   DNI       NOMBRE          PUESTO          P
   --------- --------------- --------------- -
   3333      PEPE            AMORTIGUACION   0
   5555      LUISA           AMORTIGUACION   1
```

Como se puede apreciar, PEPE sólo aparece una vez en el resultado de la consulta, aunque

pertenece a los dos conjuntos.

- **INTERSECT**:

Realiza la operación de intersección de conjuntos. Las condiciones son las descritas anteriormente.

Ejemplo:

```
SQL> (SELECT *
      FROM mecanicos
      WHERE nombre = 'PEPE')
        INTERSECT
        (SELECT *
      FROM mecanicos
      WHERE puesto = 'AMORTIGUACION')

   DNI       NOMBRE          PUESTO          P
   --------- --------------- --------------- -
   3333      PEPE            AMORTIGUACION   0
```

- **MINUS**:

Realiza la operación resta de conjuntos. Es decir, A - B son las tuplas que están en A y no están en B. La operación de resta no es conmutativa, como muestra el siguiente ejemplo.

Ejemplo:

```
SQL> (SELECT *
      FROM mecanicos
      WHERE nombre = 'PEPE')
        MINUS
        (SELECT *
      FROM mecanicos
      WHERE puesto = 'AMORTIGUACION')

no rows selected



SQL> (SELECT *
      FROM mecanicos
      WHERE puesto = 'AMORTIGUACION')
       MINUS
        (SELECT *
      FROM mecanicos
      WHERE nombre = 'PEPE');

   DNI       NOMBRE          PUESTO          P
   --------- --------------- --------------- -
   5555      LUISA           AMORTIGUACION   1
```

Generalmente estos comandos no se utilizan ya que se pueden escribir de forma más simplificada introduciendo una nueva condición en la cláusula `WHERE`, haciendo, además, que la consulta sea más eficiente. Aunque, en algunos casos, resuelven consultas complicadas de forma bastante elegante, como es el caso del siguiente ejemplo, en donde se utilizan dos subconsultas sincronizadas además de operaciones entre conjuntos.

Ejemplo:

- Encontrar los mecánicos que han reparado todos los coches de alguna marca:

```
SQL> SELECT distinct nombre,marca
       FROM mecanicos m,coches c
         /* no se hace JOIN ya que se quiere comprobar
            cada mecánico con todas las marcas */
       WHERE not exists (
                 (SELECT distinct mat
                  FROM coches
                  WHERE marca = c.marca) /* obtiene los coches de
                                            cada marca */
                MINUS
                 (SELECT distinct mat
                FROM trabajos t
                WHERE t.dni = m.dni) /* obtiene los coches
                                        reparados por cada mecánico
       */
     );

  NOMBRE          MARCA
  --------------- ---------------
  ANTONIO         TOYOTA
  EMILIO          VW
  LUIS            TOYOTA
  LUIS            VW
  LUISA           TOYOTA
  LUISA           VW
  PEPE            PEUGEOT
  PEPE            VW
```

Esta consulta funciona de la siguiente manera: el `SELECT` inicial va obteniendo parejas dni-marca, y para cada pareja la primera subconsulta obtiene todos los coches reparados de esa marca y la segunda subconsulta obtiene todos los coches reparados por ese mecánico, de tal forma que cuando ambos conjuntos sean iguales (el mecánico ha reparado todos los coches de esa marca) su diferencia devuelve el conjunto vacío, verificándose la condición del `WHERE` del `SELECT` inicial.

7. Vistas

En este último apartado dedicado a SQL se va a estudiar el concepto de **VISTA**, viendo su utilidad, cómo se crean y cómo se destruyen.

7.1. Concepto de vista

Se puede definir una *vista* como una **tabla lógica** que permite acceder a los datos de otras tablas. Las tablas sobre las cuales la vista se crea se denominan *tablas base*.

El que una vista sea una tabla lógica significa que las vistas no contienen datos en sí mismas. Una vista puede ser entendida como una tabla virtual que no tiene datos propios, sino que se nutre de datos de otra(s) tabla(s). Las vistas son muy útiles ya que permiten establecer accesos selectivos a diferentes partes de la base de datos:

- Una vista puede proporcionar acceso a determinadas columnas de la(s) tabla(s) que definen la vista. Por ejemplo, se puede definir una vista de la tabla MECANICOS que sólo muestre los atributos DNI y NOMBRE, pero no el atributo PARCIAL.

- Una vista puede proporcionar seguridad de la información contenida en la(s) tabla(s) que la definen. Se puede definir una vista que muestre sólo determinadas tuplas de las tablas asociadas, por ejemplo, una vista que muestre sólo el sueldo del usuario que accede a ella, pero no el resto de sueldos de los empleados de la empresa.

En una base de datos, las vistas son utilizadas con los siguientes propósitos:

- Para proporcionar un nivel adicional de seguridad sobre las tablas, restringiendo el acceso a un predeterminado conjunto de tuplas o atributos de una tabla base.

- Para ocultar la complejidad de los datos. Por ejemplo, una vista puede ser utilizada para actuar como una única tabla, mientras que varias tablas se usan para construir el resultado.

- Para presentar los datos desde otra perspectiva. Por ejemplo, las vistas proporcionan la posibilidad de renombrar los atributos sin necesidad de cambiar la definición de la tabla base.

- Para obligar al RDBMS a realizar algunas operaciones, tales como los joins, sobre la base de datos que contiene la vista, en lugar de en otras posibles bases de datos referenciadas en la misma sentencia SQL.

Una vista, una vez definida, puede ser utilizada en cualquier lugar (incluso en la definición de otra vista) en donde se pueda utilizar una tabla en cualquiera de los siguientes comandos SQL:

- `DELETE`
- `INSERT`
- `UPDATE`
- `SELECT`

7.2. Definición de vistas en Oracle

La sintaxis del comando SQL para la creación de vistas es la siguiente:

```
CREATE    [OR REPLACE] [FORCE | NOFORCE] VIEW nombre_vista
          [(alias [,alias]...)]
AS subconsulta
[WITH READ ONLY ]
[WITH CHECK OPTION ] ;
```

donde:

- **OR REPLACE**: crea de nuevo la vista si ésta ya existe.

- **FORCE**: crea la vista sin tener en cuenta si las tablas base de la vista existen y sin tener en cuenta si el propietario del esquema que contiene la vista tiene los privilegios suficientes sobre dichas tablas. Nótese que estas dos condiciones tienen que ser ciertas antes de usar los comandos `SELECT`, `INSERT`, `UPDATE` o `DELETE` sobre la vista.

- **NOFORCE**: crea la vista sólo si las tablas base existen y el propietario de la vista tiene privilegios sobre ellas. **La opción por defecto es NOFORCE**.

- **nombre_vista**: es el nombre de la vista.

- **alias**: especifica nombres para las expresiones seleccionadas por la subconsulta de la vista (serán los nombres de los atributos de la vista). El número de alias debe coincidir con el número de expresiones seleccionadas por la subconsulta. Estos alias deben ser únicos dentro de la vista. Si se omiten los alias, Oracle los deriva de las columnas o de los alias de las columnas seleccionados en la subconsulta. Por esta razón se deben utilizar alias si la subconsulta de la vista contiene expresiones en lugar de sólo nombres de columnas.

- **AS subconsulta**: identifica los atributos y tuplas de la(s) tabla(s) en las que se basa la vista, es decir, establece el contenido de la vista. Esta subconsulta puede ser cualquier sentencia `SELECT` que no contenga la cláusula `ORDER BY`.

- **WITH READ ONLY**: especifica que no se pueden realizar borrados, inserciones o actualizaciones a través de la vista.

- **WITH CHECK OPTION**: especifica que las inserciones y las actualizaciones realizadas sobre la vista deben dar como resultado tuplas que la vista pueda seleccionar. La opción `CHECK OPTION` no garantiza esto si, a su vez, existe alguna subconsulta en la subconsulta de la vista o si la vista se basa a su vez en otra vista.

La restricción más importante que existe sobre las vistas es que si la subconsulta contiene alguno de los siguientes elementos, no se podrán realizar inserciones, actualizaciones o borrados sobre la vista:

- operaciones **JOIN**
- operadores de conjuntos
- funciones agregadas
- la cláusula `GROUP BY`
- el operador `DISTINCT`

Nótese que si la vista contiene expresiones, sólo se podrá actualizar la vista con una sentencia

UPDATE que no se refiera a ninguna de las expresiones.

Para borrar una vista se utiliza el siguiente comando:

```
DROP VIEW nombre_vista;
```

donde **nombre_vista** es el nombre de la vista que se quiere borrar.

Para ver las vistas que ha creado un usuario y sus subconsultas asociadas se puede consultar el Diccionario de Datos a través de la vista USER_VIEWS.

7.3. Ejemplos de utilización de vistas

• Crear una vista llamada CHAPA que muestre los mecánicos de CHAPA y las horas que han trabajado:

```
SQL> CREATE OR REPLACE VIEW chapa
     AS     SELECT dni, SUM(horas) "Total horas"
            FROM mecanicos m,trabajos t
            WHERE m.dni=t.dni AND
                puesto='CHAPA'
          GROUP BY m.dni,nombre;
```

Nótese que en la declaración de la vista no se necesita definir un nombre para el atributo basado en la expresión SUM(horas) debido a que la subconsulta utiliza un alias de atributo ("Total horas") para la expresión.

Una vez creada la vista se puede utilizar como si fuera una tabla:

```
SQL> SELECT * FROM chapa;

   DNI        Total horas
   ---------  -----------
   1111              3.2
   4444                7
   6666              1.2
```

• Crear una vista actualizable llamada Tiempo_parcial conteniendo los atributos indispensables de los mecánicos que estén a tiempo parcial.

```
SQL> CREATE VIEW Tiempo_parcial (dni,contrato)
     AS     SELECT dni,parcial
            FROM mecanicos
            WHERE parcial='1'
     WITH CHECK OPTION;
```

Debido al CHECK OPTION, no se pueden insertar nuevas tuplas en Tiempo_parcial si el nuevo mecánico no tiene un contrato a tiempo parcial:

```
SQL> INSERT INTO Tiempo_parcial VALUES ('7777','0');
     INSERT INTO Tiempo_parcial VALUES ('7777','0')
          *
     ERROR at line 1:
     ORA-01402: view WITH CHECK OPTION WHERE-clause violation
```

Si se inserta un nuevo mecánico a través de la vista, como no se han incluido los atributos NOMBRE y PUESTO de la tabla mecánicos, tomarán valor NULL, siendo esto posible ya que cuando se creo la tabla de mecánicos no se especificó la restricción NOT NULL para estos campos:

```
SQL> INSERT INTO Tiempo_parcial VALUES ('7777','1');

     1 row created.

SQL> SELECT * FROM mecanicos;

   DNI       NOMBRE          PUESTO          P
   --------- --------------- --------------- -
   1111      ANTONIO         CHAPA           1
   2222      LUIS            MOTOR           0
   3333      PEPE            AMORTIGUACION   0
   4444      LOLA            CHAPA           1
   5555      LUISA           AMORTIGUACION   1
   6666      EMILIO          CHAPA           0
   7777                                      1
```

Si en vez de haber utilizado la cláusula `WITH CHECK OPTION` se hubiera utilizado `WITH READ ONLY`, entonces no estarían permitidas las operaciones de inserción, borrado o actualización sobre la vista.

• Obtener el total de horas trabajadas por los mecánicos de chapa que tienen un contrato a tiempo parcial:

```
SQL>   SELECT c.dni,c."Total horas"
         FROM chapa c,tiempo_parcial tp
         WHERE c.dni=tp.dni;

   DNI       Total horas
   --------- -----------
   1111              3.2
   4444                7
```

8. Ejercicios Resueltos

8.1. Creación de tablas

COCHES:
- mat (matricula del coche, 8 caracteres y llave primaria)
- marca (15 caracteres)
- an_fab (año de fabricacion, número de 2 dígitos)

MECANICOS:
- dni (9 caracteres, llave primaria)
- nombre (15 caracteres)
- puesto (15 caracteres)
- parcial (1 caracter)

TRABAJOS:
- mat (la matrícula del coche, llave externa respecto de COCHES),
- dni (el dni del mecánico),
- horas (número de 3 cifras con 1 decimal, debe ser mayor de 0.5)
- fecha_rep (tipo fecha)

Solución:

```
CREATE TABLE coches (
          mat     VARCHAR(8) CONSTRAINT pk_coches PRIMARY KEY,
          marca   VARCHAR(15),
          an_fab  NUMBER(2)
);

CREATE TABLE mecanicos (
  dni     VARCHAR(9) CONSTRAINT pk_mecanicos PRIMARY KEY,
  nombre  VARCHAR(15),
  puesto  VARCHAR(15),
  parcial VARCHAR(1)
);

CREATE TABLE trabajos (
  mat       VARCHAR(8),
  dni       VARCHAR(9),
  horas     NUMBER(3,1) CONSTRAINT ck_trabajo
                            CHECK (horas>=0.5),
  fecha_rep DATE,
  CONSTRAINT fk1_trabajos
    FOREIGN KEY (mat) REFERENCES coches
);
```

8.2. Modificación de tablas

Añadir a la tabla `COCHES` el atributo `modelo` (15 caracteres)

Establecer los atributos `mat` y `dni` como la llave primaria de `TRABAJOS`.
Establecer el atributo `dni` de `TRABAJOS` como llave externa respecto a `MECANICOS`.
Ampliar a 4 la longitud del atributo `an_fab` de la tabla `COCHES`.

Solución:

```
ALTER TABLE coches
ADD modelo VARCHAR(15);

ALTER TABLE trabajos
ADD CONSTRAINT pk_trabajos
PRIMARY KEY (mat,dni);

ALTER TABLE trabajos
ADD  CONSTRAINT fk2_trabajos
FOREIGN KEY (dni) REFERENCES mecanicos;

ALTER TABLE coches
MODIFY an_fab NUMBER(4);
```

8.3. Inserción de tuplas

Realizar la inserción de valores de forma que las tablas queden de la siguiente forma:

COCHES

MAT	MARCA	MODELO	AN_FAB
M3020KY	TOYOTA	CARINA	1996
J1234Z	RENAULT	MEGANE	1997
GR4321A	RENAULT	5	1978
B4444AC	PEUGEOT	504	1978
CA0000AD	PEUGEOT	205	1996
GR1111AK	PEUGEOT	207	1998
J9999AB	VW	BEATTLE	1998

MECANICOS

DNI	NOMBRE	PUESTO	PARCIAL
1111	ANTONIO	MOTOR	1
2222	LUIS	MOTOR	0
3333	PEPE	AMORTIGUACION	0
4444	LOLA	CHAPA	1
5555	LUISA	AMORTIGUACION	1
6666	EMILIO	CHAPA	0
7777	ANGEL	CHAPA	0

TRABAJOS

MAT	DNI	HORAS	FECHA_REP
M3020KY	1111	1	23-FEB-96
M3020KY	2222	2.5	23-FEB-96

J1234Z	4444	7	19-MAR-97
J1234Z	2222	3	19-MAR-97
GR4321A	3333	2.1	1-JAN-98
B4444AC	3333	3.2	23-APR-96
CA0000AD	3333	8	23-APR-96
M3020KY	5555	2	23-FEB-96
J9999AB	6666	1	5-MAR-98
J9999AB	5555	0.6	5-MAR-98
J9999AB	2222	0.9	5-MAR-98
J1234Z	1111	2.2	19-MAR-97
GR1111AK	3333	5.5	1-JAN-98
J9999AB	3333	6.7	5-MAR-98
GR1111AK	5555	2.5	1-JAN-98
GR1111AK	7777	1	1-JAN-98

Solución:

```
INSERT INTO coches (MAT,MARCA,MODELO,AN_FAB)
VALUES ('M3020KY','TOYOTA','CARINA',1996);

INSERT INTO coches (MAT,MARCA,MODELO,AN_FAB)
VALUES ('J1234Z','RENAULT','MEGANE',1997);

INSERT INTO coches (MAT,MARCA,MODELO,AN_FAB)
VALUES ('GR4321A','RENAULT','5',1978);

...

INSERT INTO MECANICOS
VALUES ('1111','ANTONIO','MOTOR','1');

INSERT INTO MECANICOS
VALUES ('2222','LUIS','MOTOR','0');

INSERT INTO MECANICOS
VALUES ('3333','PEPE','AMORTIGUACION','0');

...

INSERT INTO TRABAJOS
VALUES ('M3020KY','1111',1,'23-FEB-96');

INSERT INTO TRABAJOS
VALUES ('M3020KY','2222',2.5,'23-FEB-96');

INSERT INTO TRABAJOS
VALUES ('J1234Z','4444',7,'19-MAR-97');

...
```

8.4. Actualización y borrado de tuplas

Modificar a `'CHAPA'` el puesto de `'ANTONIO'`.
Borrar a `'ANGEL'` de la base de datos.
Aumentar las horas de los trabajos hechos el '5-MAR-98' en un 15%.

Solución:

```
UPDATE mecanicos
SET puesto='CHAPA'
WHERE nombre='ANTONIO';
```

Como la llave externa `DNI` en `TRABAJOS` no se ha definido con la opción `ON DELETE CASCADE`, entonces primero se deben borrar todos los trabajos de Ángel para posteriormente eliminarle a él de la base de datos.

```
DELETE FROM trabajos
WHERE dni = ( SELECT dni
                     FROM mecanicos
                     WHERE nombre='ANGEL');

DELETE FROM mecanicos
WHERE nombre='ANGEL';

UPDATE trabajos
SET horas = horas * 1.15
WHERE fecha_rep='5-MAR-98';
```

8.5. Consultas

Parejas de coches de distintas marcas fabricados el mismo año y reparados ambos de AMORTIGUACIÓN:

```
SQL> SELECT c1.marca,c1.modelo,c2.marca,c2.modelo
     FROM coches c1,coches c2
     WHERE c1.an_fab=c2.an_fab AND c1.marca>c2.marca AND
           NOT EXISTS
              ((SELECT mat
                FROM coches
                   WHERE mat IN (c1.mat,c2.mat))
                MINUS
                  (SELECT mat
                   FROM trabajos t,mecanicos m
                WHERE t.dni=m.dni AND
                         m.puesto='AMORTIGUACION'));

MARCA           MODELO          MARCA           MODELO
--------------- --------------- --------------- ---------------
RENAULT         5               PEUGEOT         504
TOYOTA          CARINA          PEUGEOT         205
VW              BEATTLE         PEUGEOT         207
```

Esta misma consulta pero sin utilizar subconsultas sincronizadas:

```
SQL> SELECT distinct c1.marca,c1.modelo,c2.marca,c2.modelo
     FROM coches c1,coches c2,trabajos t1,trabajos t2,
          mecanicos m1,mecanicos m2
     WHERE c1.mat=t1.mat AND t1.dni=m1.dni AND
           c2.mat=t2.mat AND t2.dni=m2.dni AND
           c1.an_fab=c2.an_fab AND c1.marca>c2.marca AND
           m1.puesto='AMORTIGUACION' AND
           m2.puesto='AMORTIGUACION';
```

Obtener los coches que NO son reparados por 'PEPE':

```
SQL> SELECT mat,marca,modelo
     FROM coches
     WHERE mat not in (SELECT mat
                          FROM trabajos t,mecanicos m
                       WHERE t.dni=m.dni AND
                                m.nombre='PEPE');

   MAT      MARCA           MODELO
   -------- --------------- ---------------
   M3020KY  TOYOTA          CARINA
   J1234Z   RENAULT         MEGANE
```

Obtener los mecánicos que no han reparado ningún coche RENAULT:

```
SQL> SELECT nombre
     FROM mecanicos m
     WHERE not exists (
          (SELECT distinct mat
           FROM coches
           WHERE marca = 'RENAULT')
          INTERSECT
          (SELECT distinct mat
           FROM trabajos t
           WHERE t.dni = m.dni));

   NOMBRE
   ---------------
   EMILIO
   LUISA
```

Otras posibles formas de resolver esta consulta son:

```
SQL> (SELECT distinct nombre
      FROM mecanicos m,trabajos t,coches c
      WHERE m.dni=t.dni AND t.mat=c.mat AND c.marca!='RENAULT')
     MINUS
     (SELECT distinct nombre
      FROM mecanicos m,trabajos t,coches c
      WHERE m.dni=t.dni AND t.mat=c.mat AND c.marca='RENAULT');


SQL> SELECT nombre
     FROM mecanicos
     WHERE dni not in
            (SELECT dni
             FROM trabajos t,coches c
             WHERE t.mat=c.mat AND c.marca='RENAULT');
```

8.6. Vistas

Crear una vista que contenga la marca y el modelo de cada coche, el puesto en que fue reparado y el total de horas invertidas en ese puesto:

```
SQL> CREATE OR REPLACE VIEW total_coche_puesto
     as     SELECT marca,modelo,puesto,SUM(horas) "total horas"
          FROM mecanicos m, coches c,trabajos t
          WHERE m.dni=t.dni AND c.mat=t.mat
          GROUP by marca,modelo,puesto;
```

```
SQL> SELECT * FROM total_coche_puesto;

MARCA           MODELO          PUESTO          total horas
--------------- --------------- --------------- -----------
PEUGEOT         205             AMORTIGUACION             8
PEUGEOT         207             AMORTIGUACION             8
PEUGEOT         504             AMORTIGUACION           3.2
RENAULT         5               AMORTIGUACION           2.1
RENAULT         MEGANE          CHAPA                   9.2
RENAULT         MEGANE          MOTOR                     3
TOYOTA          CARINA          AMORTIGUACION             2
TOYOTA          CARINA          CHAPA                     1
TOYOTA          CARINA          MOTOR                   2.5
VW              BEATTLE         AMORTIGUACION           8.4
VW              BEATTLE         CHAPA                   1.2
VW              BEATTLE         MOTOR                     1
```

3
PL/SQL
Programación del servidor

1. Introducción

El desarrollo de una aplicación de BD requiere normalmente la utilización de estructuras alternativas, repetitivas, de declaración de datos…. similares a las que se pueden encontrar en lenguajes procedurales tales como C, C++, o Pascal. Esta construcción es necesaria para implementar tipos complejos de datos y algoritmos. El principal problema con el que se encuentra el lenguaje declarativo SQL es la ausencia de dicho tipo de estructuras, lo que impide que muchas tareas puedan llevarse a cabo. Para superar este inconveniente fue creado PL/SQL (Procedural Language/SQL). PL/SQL es una extensión procedural de Oracle-SQL que ofrece un lenguaje de construcción similar a los lenguajes de programación imperativos. PL/SQL permite al usuario y al diseñador desarrollar aplicaciones de BD complejas que requieren el uso de estructuras de control y elementos procedurales tales como procedimientos, funciones y módulos.

La construcciión básica en PL/SQL es el bloque (block). Los bloques permiten al diseñador combinar sentencias SQL en unidades. En un bloque, pueden ser declaradas constantes y variables, y las variables pueden ser utilizadas para almacenar resultados de consultas. Las sentencias en un bloque PL/SQL pueden incluir:

- Estructuras de control repetitivas (loops)
- Sentencias condicionales (if-then-else)
- Manejo de excepciones (exception hadling)
- Llamadas a otros bloques PL/SQL

Los bloques PL/SQL que especifican procedimientos y funciones pueden ser agrupados en paquetes (packages) similares a módulos. Oracle presenta varios paquetes predefinidos: rutinas input/output, ficheros handling, de planificación de trabajos, etc.

Otra característica importante de PL/SQL es que presenta un mecanismo para procesar los resultados de las consultas orientado a tuplas (tuple-oriented way), es decir, de tupla en tupla, mediante la definición de los cursores (cursors). Un cursor es básicamente un puntero a un resultado de una consulta y es utilizado para leer valores de atributos de tuplas seleccionadas. Un cursor se utiliza habitualmente en combinación con una estructura repetitiva de modo que cada tupla que sea leída por el cursor puede ser procesada individualmente.

En resumen, los objetivos del PL/SQL son:

- Hacer más expresivo el SQL
- Procesar los resultados de consultas orientado a tuplas (tuple-oriented way)
- Optimizar la combinación de sentencias SQL
- Desarrollar programas de aplicación de base de datos modulares
- Reutilizar el código del programa
- Reducir el coste por mantenimiento y cambio de aplicaciones

1.1. Tablas para los ejemplos

Los ejemplos que ilustran los conceptos presentados en este capítulo utilizan las siguientes tablas:

EMPLEADO (NUM_EMP, NOMBRE_EMP, TAREA, JEFE, FECHA_ALTA, SALARIO, NUM_DPTO)

NUM_EMP	Número identificador del empleado
NOMBRE_EMP	Nombre del empleado
TAREA	Tarea o puesto que desempeña el empleado en la empresa
JEFE	Número identificador del jefe del empleado
FECHA_ALTA	Fecha de entrada del empleado en la empresa
SALARIO	Salario mensual
NUM_DPTO	Número identificador del departamento en el que empleado trabaja

DPTO (NUM_DPTO, NOMBRE_DPTO, LOCALIDAD)

NUM_DPTO	Número identificador del departamento
NOMBRE_DPTO	Nombre del departamento
LOCALIDAD	Nombre de la localidad en la que se ubica el departamento

NIVEL (ID_NIVEL, SAL_MIN, SAL_MAX)

ID_NIVEL	Número identificativo del nivel del empleado
SAL_MIN	Tope mínimo salarial del nivel
SAL_MAX	Tope maximo salarial del nivel

PROYECTO (NUM_PRO, NOMBRE_PRO, JEFE_PRO, NUM_PER, PRESUPUESTO,FECHA_INI, FECHA_FIN)

NUM_PRO	Número identificador del proyecto
NOMBRE_PRO	Nombre del proyecto
JEFE_PRO	Número identificador del jefe del proyecto
NUM_PER	Número de personas que integrados en el proyecto
PRESUPUESTO	Presupuesto del proyecto
FECHA_INI	Fecha de inicio del proyecto
FECHA_FIN	Fecha de conclusión de proyecto

2. Estructura de los bloques PL/SQL

PL/SQL es un lenguaje estructurado en bloques. Cada bloque es una unidad de programa que puede ser o no etiquetada mediante una cabecera de bloque, siendo posible la existencia de bloques anidados. Los bloques que crean un procedimiento, una función, o un paquete deben ser etiquetados. Un bloque PL/SQL consta de dos secciones: una sección declarativa opcional y una sección que contiene sentencias PL/SQL. Dentro de esta última podemos incluir una parte opcional para el manejo de excepciones (exception-handling). La sintaxis de un bloque PL/SQL es la siguiente:

```
[<Cabecera de Bloque>]
[DECLARE
  <Constantes>
  <Variables>
    <Cursores>
    <Excepciones definidas por el usuario>]
BEGIN
  <Sentencias PL/SQL>
[EXCEPTION
    <Manejo de excepciones>]
END;
```

La cabecera del bloque especifica si el bloque PL/SQL es un procedimiento, una función o un paquete. Si no se especifica cabecera, se dice que el bloque es anónimo (anonymous block). En los bloques anidados el ámbito de las variables declaradas es análogo al ámbito de las variables en lenguajes de programación tales como C o Pascal.

3. Declaraciones

Las constantes, variables, cursores y excepciones que son usadas en un bloque PL/SQL deben ser declaradas en su sección declarativa. Las variables y constantes deben ser declaradas como sigue:

```
<nombre de variable> [CONSTANT] <tipo de dato> [NOT NULL] [:= <expresión>];
```

Los tipos de datos validos son los mimo que en SQL. Los datos booleanos pueden ser solo verdadero, falso, o nulo. La cláusula `NOT NULL` requiere que la variable declarada tenga un valor diferente de nulo. `<expresión>` es utilizada para inicializar una variable. Si no hay ninguna expresión especificada el valor null es asignado a la variable. La cláusula `[CONSTANT]` hace que una vez que el valor ha sido asignado a la variable, éste no puede ser modificado (la variable se convierte en una constante). Ejemplo:

```
DECLARE
   Fecha_contrato DATE; /*inicialización implícita a null*/
   Nombre_tarea VARCHAR2(80):='VENDEDOR';
   Emp_encontrado BOOLEAN; /*inicialización implícita a null*/
   Incremento_sal CONST NUMBER(3,2):= 1.5; /*constante/
   . . .
BEGIN  . . .  END;
```

En lugar de declarar directamente una variable con un tipo de dato SQL (varchar, number, …), es posible declarar el tipo de una variable asignándole a ésta, el tipo de dato asignado previamente a una columna de la tabla. Esta declaración se denomina declaración anclada (anchored). Por ejemplo `EMPLEADO.NUM_EMP%TYPE` se refiere al tipo de dato de la columna `NUM_EMP` en la tabla `EMPLEADO`. De la misma forma que una variable puede ser definida mediante una declaración anclada, también podemos declarar conjuntos de atributos que pueda almacenar una tupla completa de una tabla dada (o un resultado de una consulta). Por ejemplo, el tipo de dato `DPTO%ROWTYPE` especifica un registro apropiado para almacenar todos los valores de los atributos de una fila completa de la tabla `DPTO`. Normalmente, los registros así definidos se utilizan en combinación con un cursor. Se puede acceder a un campo en un registro utilizando `<nombre registro>.<nombre columna>`. De este modo realizamos la siguiente declaración:

```
DECLARE
   reg_dpto  DPTO%ROWTYPE;
```

el numero del departamento se referencia como `REG_DPTO.NUM_DPTO`.

Una declaración de un cursor especifica un conjunto de tuplas resultado de una consulta, que pueden ser procesadas secuencialmente tupla a tupla (tuple-oriented way) utilizando la sentencia fetch**.** Una declaración de un cursor tiene la forma

```
CURSOR <nombre de cursor> [(<lista de parámetros>)] IS <sentencia SELECT>;
```

El nombre del cursor es un identificador no declarado, no pudiendo ser el nombre de ninguna variable PL/SQL. Un parámetro tiene la forma `<nombre de parámetro> <tipo de parámetro>`. Los tipos de parámetros posibles son `CHAR`, `VARCHAR`, `NUMBER`, `DATE` y `BOOLEAN` así como sus subtipos correspondientes tal como `INTEGER`. Los parámetros son utilizados para asignar valores a las variables que utilizan las sentencias `SELECT`.

Ejemplo: queremos recuperar (tuple_oriented way) los valores de los atributos nombre del trabajo y el nombre de los empleados contratados después de una fecha dada y cuyo sueldo sea mayor a un valor.

```
CURSOR empleado_cur (fecha_inical DATE, dno NUMBER) IS
   SELECT tarea, nombre_emp
   FROM empleado e
   WHERE fecha_alta> fecha_inicial AND
          EXITSTS (SELECT *
                   FROM empleado
                   WHERE e.jefe = num_emp AND num_dpto = dno);
```

Si alguna de las tuplas seleccionadas fuera modificada en el bloque PL/SQL, la cláusula `FOR UPDATE[(<columna(s)>)]` debería ser añadida al final de la declaración del cursor. En este caso, las tuplas seleccionadas son bloqueadas y no pueden ser accedidas por otros usuarios hasta que sea ejecutado un comando `COMMIT`. Antes de utilizar un cursor declarado previamente, éste debe estar abierto. Una vez procesadas las tuplas definidas en él, debe de ser cerrado. Discutiremos con mas detalle el uso del cursor más adelante.

Las excepciones son utilizadas para procesar de forma controlada los errores y advertencias que ocurren durante la ejecución de sentencias PL/SQL. Algunas excepciones, como `ZERO_DIVIDE`, son definidas internamente. Otras excepciones pueden ser especificadas por el usuario al final de un bloque PL/SQL. Para que el usuario defina las excepciones necesita declararlas utilizando

```
<nombre de excepcion> EXCEPTION
```

Discutiremos el manejo de las excepciones en el apartado 6.

4. Elementos del lenguaje

Además de la declaración de variables, constantes y cursores, PL/SQL presenta varias elementos como la asignación de valores a variables, estructuras repetitivas (loops), estrucuturas alternativas (if-then-else), llamadas a procedimientos y funciones, etc. Sin embargo, PL/SQL no permite comandos del lenguaje de definición de datos DDL (data definition language) tales como la creación de tablas (create table). Además, PL/SQL utiliza una modificación en la sentencia `SELECT` que requiere que una tupla sea asignada a un registro (o una lista de variables).

En PL/SQL hay varias alternativas para la asignacón de un valor a una variable. El camino más sencillo para asignar un valor a una variable es

```
DECLARE
   contador INTEGER:=0;
   . . .
BEGIN
   contador:=contador + 1;
```

Los valores asignados a una variable también pueden ser recuperados de la base de datos usando la sentencia `SELECT`.

```
SELECT <columna(s)> INTO <lista de variables>
FROM <tabla(s)> WHERE <condición>;
```

Es muy importante asegurar que la sentencia `SELECT` recupera a lo sumo una tupla. Si no ocurre esto, no es posible asignar los valores del atributo a la lista de variables especificada y se produciría un error de ejecución. Si el `SELECT` recupera mas de una tupla, en lugar de éste, deberá ser utilizado un cursor. Además, los tipos de datos de las variables especificadas deben concordar con los valores de atributo recuperados. Para la mayoría de los tipos de datos, PL/SQL realiza una conversión automática (ej., de entero a real).

Tras la palabra reservada into, en lugar de una lista de variables podemos incluir un registro. Este caso, la sentencia `SELECT` debe recuperar como máximo una única tupla.

```
DECLARE
   REG_EMPLEADO EMPLEADO%ROWTYPE;
      MAXSAL EMPLEADO.SALARIO%TYPE;
BEGIN
   SELECT NUM_EMP, NOMBRE_EMP, TAREA, JEFE, SALARIO, FECHA_ALTA,
          NUM_DPTO
    INTO REG_EMPLEADO
    FROM EMPLEADO WHERE NUM_EMP=5698;
    SELECT MAX (SALARIO) INTO MAXSAL FROM EMPLEADO;
         . . .
END;
```

PL/SQL incluye bucles `WHILE`, dos estucturas de bucles `FOR`, y bucles continuos. Mas adelante utilizaremos algunos de ellos en combinación con cursores. Todos los tipos de bucles son utilizados para ejecutar multiples veces una secuencia de sentencias. La especificación de bucles se realiza de la misma forma que en los lenguajes de programación imperativos como C y Pascal.

La sintaxis de un bucle `WHILE` es la siguiente:

```
[<nombre de etiqueta>]
WHILE <condicion> LOOP
       <secuencia de sentencias>;
END LOOP [<nombre de etiqueta>];
```

En general, el etiquetado del bucle es opcional, aunque cuando se presentan varios bucles continuos (incondicionales) anidados, resulta obligatorio el etiquetado, ya que es necesario incluir la sentencia `EXIT <nombre de etiqueta>;`

Por tanto, el numero de iteraciones a través de un bucle `WHILE` no es conocido hasta que el bucle finaliza. El numero de iteraciones a través del bucle `FOR` puede ser precisado utilizando dos números enteros.

```
[<nombre de etiqueta>]
FOR <índice> IN [REVERSE] <limite inferior>...<limite superior> LOOP
     <secuencia de sentencias>;
END LOOP [<nombre de etiqueta>];
```

El contador de bucle `<índice>` esta implícitamente declarado. El ámbito del contador del bucle es sólo el bucle `FOR`. Esto anula el alcance de algunas variables que tienen el mismo nombre fuera del bucle. Dentro el bucle `FOR`, `<índice>` puede ser referenciado como una constante. `<índice>` puede aparecer en expresiones, pero no es posible asignarle valores. El uso de la palabra reservada `REVERSE`, provoca in incremento negativo del índice, desde el limite superior (`<límite superior>`) hasta el inferior (`<límite inferior>`).

5. Manejo de Cursores

Antes de que un cursor pueda ser utilizado, debe ser abierto usando la sentencia OPEN.

```
OPEN <nombre de cursor> INTO [<lista de parámetros>];
```

La sentencia SELECT asociada es entonces procesada, y el cursor hace referencia (apunta) a la primera tupla seleccionada. Las tuplas seleccionadas pueden ser entonces procesadas una a una usando el comando FETCH.

```
FETCH <nombre de cursor>  INTO <lista de variables>;
```

El comando FETCH asigna los valores de los atributos seleccionados de la tupla actual a la lista de variables. Después de ejecutar el comando FETCH, el cursor avanza a la siguiente tupla del conjunto seleccionado. Hemos de tener en cuenta las variables en la lista deben de tener los mismos tipos de datos que los valores seleccionados. Después de que todas las tuplas hayan sido procesadas, el cursor debe de ser deshabilitado mediante el comando CLOSE.

```
CLOSE <nombre de cursor>;
```

El siguiente ejemplo muestra como un cursor es utilizado junto a un bucle:

```
DECLARE
   CURSOR EMP_CUR IS SELECT * FROM EMPLEADO;
   REG_EMP EMPLEADO%ROWTYPE;
   SAL_EMP  EMPLEADO.SALARIO%TYPE;
BEGIN
   OPEN EMP_CUR;
   LOOP
     FETCH EMP_CUR INTO REG_EMP;
     EXIT WHEN EMP_CUR%NOTFOUND;
     SAL_EMP := REG_EMP.SAL;
     <SECUENCIA DE SENTENCIAS>
   END LOOP;
   CLOSE EMP_CUR;
   . . .
END;
```

Cada bucle puede ser completado incondicionalmente usando la cláusula EXIT.

```
EXIT  [<etiqueta de bloque>] [WHEN <condición>]
```

El uso de EXIT sin una etiqueta de bloque causa el fin de la ejecución del bucle que contiene la sentencia **exit**. La condición puede ser una simple comparación de valores. En la mayoría de los casos, la condición se refiere a un cursor. En el ejemplo anterior, %NOTFOUND es un operador que se evalúa como falso si el último comando FETCH ejecutado ha leído una tupla. Si no ha sido posible leer una tupla al haber sido ya todas procesadas, %NOTFOUND devuelve falso. El valor de <nombre de cursor>%NOTFOUND es nulo antes de que la primera tupla sea procesada con un FETCH. El operador %FOUND es la oposición lógica de %NOTFOUND.

El cursor de los bucles FOR puede ser usado para simplificar el uso del cursor:

```
<nombre de etiqueta>]
FOR <nombre registro> IN <nombre cursor> [(<lista parametros>)] LOOP
  <secuencia de sentencias>
END LOOP [<nombre de etiqueta>];
```

Con esta sintaxis, se declara de forma implícita el registro apropiado para almacenar una tupla del cursor. Además, este bucle ejecuta implícitamente un FETCH en cada iteración, así como un OPEN antes de que se entre en el bucle y un CLOSE después de que el bucle finalice. Si se intenta ejecutar un bucle con un cursor que no ha seleccionado ninguna tupla, éste finaliza automáticamente.

Esto es igualmente posible al especificar una consulta en lugar de <nombre de cursor> en un bucle FOR:

```
FOR <nombre de registro> IN (<sentencia select>) LOOP
  <secuencia de sentencias>
END LOOP;
```

Es decir, un cursor no necesita ser especificado antes de entrar el bucle, pero es definido en la sentencia SELECT. Ejemplo:

```
FOR REG_SAL IN (SELECT SALARIO+COMM TOTAL FROM EMPLEADO) LOOP
  ...;
END LOOP;
```

TOTAL es un alias para la expresión en la sentencia SELECT. Así, en cada iteración sólo se procesa una única tupla. El registro REG_SAL, que está implícitamente definido, contiene sólo una entrada a la que podemos acceder con REG_SAL.TOTAL. Los alias, por supuesto, no son necesarios si son seleccionado atributos exclusivamente, es decir, si la sentencia SELECT no contiene ni operadores aritméticos ni funciones agregadas.

Para las estructuras de control alternativas, PL/SQL ofrece la estructura IF-THEN-ELSE

```
IF <condición> THEN <secuencia de sentencias>
[ELSIF] <condición> THEN <secuencia de sentencias>
  ...
[ELSE] <secuencia de sentencias>
END IF;
```

Empezando con la primera condición, si ésta es verdadera, su correspondiente secuencia de sentencias son ejecutadas, de otra forma el control pasaría a la próxima condición. Así la conducta de este tipo de sentencias PL/SQL es análoga a las sentencias IF-THEN-ELSE en los lenguajes de programación imperativos.

Excepto en los comandos DDL (data definition lanbguaje) como CREATE TABLE, todos los tipos de sentencias SQL pueden ser utilizadas en los bloques PL/SQL, en particular DELETE, INSERT, UPDATE, y COMMIT. Notese que en PL/SQL sólo se permite la sentencia SELECT del tipo SELECT <column(s)> INTO <variable list>, es decir, los valores de atributo seleccionados sólo pueden ser asignados a variables. El uso de la sentencia SELECT SQL provoca un error de sintaxis. Si las sentencias UPDATE o DELETE se utilizan en combinación con un cursor, estos comandos pueden ser restringidos a la tupla en proceso (fetched). Para ello, se añade la clausula WHERE CURRENT OF <nombre de cursor>, tal y como se muestra en el siguiente ejemplo.

Ejemplo: El siguiente bloque PL/SQ ejecuta las siguientes modificaciones: Todos los empleados que tengan como jefe a 'JIMENEZ' incrementarán su salario en un 5%.

```
DECLARE
  MANAGER EMPLEADO.JEFE%TYPE;
```

```
   CURSOR EMP_CUR (NUM_JEFE NUMBER) IS
     SELECT SALARIO
     FROM EMPLEADO
     WHERE JEFE = NUM_JEFE FOR UPDATE OF  SALARIO;
BEGIN
   SELECT NUM_EMP INTO MANAGER
   FROM EMPLEADO
   WHERE NOMBRE_EMP  = 'JIMENEZ';
   FOR REG_EMP IN EMP_CUR(MANAGER) LOOP
     UPDATE EMPLEADO
     SET  SALARIO = REG_EMP.SALARIO * 1.05
     WHERE CURRENT OF EMP_CUR;
   END LOOP;
   COMMIT;
END;
```

6. Manejo de Excepciones

Un bloque PL/SQL puede contener sentencias que especifique rutinas de manejo de excepciones. Cada error o advertencia (warning) durante la ejecución del bloque PL/SQL genera una excepción. Podemos distinguir dos tipos de excepciones:

- Excepciones definidas por el sistema.

- Excepciones definidas por el usuario (que deben ser declaradas por éste en la parte del bloque donde la excepción es usada/ implementada)

Las excepciones definidas por el sistema son automáticamente generadas cuando se produce un error o advertencia. Por el contrario, las excepciones definidas por el usuario, deben ser generadas implícitamente mediante una secuencia de sentencias utilizando la cláusula `RAISE <nombre de excepción>`. Las rutinas del manejo de excepciones son implementadas por el usuario al final del bloque y tras la palabra reservada `EXCEPCTION`:

```
WHEN <nombre de excepción>  THEN  <secuencia de sentencias>;
```

Los errores más comunes que pueden darse durante la ejecución de los programas PL/SQL son manejados por excepciones definidas por sistema. La siguiente tabla muestra algunos de sus nombres y descripciones cortas.

Principales excepciones del sistema.

Nombre de la excepción	Numero	Observaciones
CURSOR_ALREADY_OPEN	ORA-06511	Se ha intentado abrir un cursor que está ya abierto.
INVALID_CURSOR	ORA-01001	Se ha intentado hacer un **fetch** sobre un cursor cerrado.
NOT_DATA_FOUND	ORA-01403	Una sentencia **select ... into** o **fetch** no devuelve ninguna tupla.
TOO_MANY_ROWS	ORA-01422	Una sentencia **select ... into** o **fetch** no devuelve más de uan tupla.
ZERO_DIVIDE	ORA-01476	Se ha intentado dividir un numero entre 0.

Ejemplo:

```
DECLARE
   SAL_EMP   EMPLEADO.SALARIO%TYPE;
   N_EMP      EMPLEADO.NUM_EMP%TYPE
   SALARIO_DEMASIADO_ALTO EXCEPCION;
BEGIN
   SELECT NUM_EMP, SALARIO INTO N_EMP, SAL_EMP
   FROM EMPLEADO WHERE NOMBRE_EMP = 'GOMEZ';
   IF SAL_EMP * 1.05 > 4000 THEN RAISE SALARIO_DEMASIADO_ALTO
   ELSE UPDATE EMPLEADO SET SQL. . .
   END IF;

   EXCEPCTION
      WHEN NO_DATA_FOUND – NO SE SELECCIONAN TUPLAS
      THEN ROLLBACK;
      WHEN SALARIO_DEMASIADO_ALTO THEN
```

```
        INSERT EMP_SAL_ALTOS VALUES( N_EMP);
        COMMIT;
END;
```

Tras la palabra reservada WHEN, podemos incluir una lista de nombres de excepciones conectados con el operador relacional OR. La última cláusula WHEN en la parte de EXCEPTION puede contener la excepción OTHERS. Ésta presenta la excepción por defecto de la rutina de manejo, por ejemplo, un ROLLBACK.

Si un programa PL/SQL es ejecutado desde la shell del SQL*Plus, las rutinas de manejo de excepciones pueden contener sentencias que muestren errores o mensaje de advertencias en la pantalla mediante el procedimiento RAISE_APPLICATION_ERROR. Este procedimiento tiene dos parámetros <numero de error> y <texto del mensaje>. <numero de error> es un entero negativo definido por el usuario y cuyo rango esta entre –20000 y –20999. <mensaje de error> es un string con una longitud no superior a 2048 caracteres. El operador de concatenación de caracteres '||' puede ser utilizado para concatenar cadenas de caracteres simples a una cadena de caracteres. Para mostrar variables numéricas, éstas deben de ser convertidas a strings usando la función TO_CHAR. Si RAISE_APPLICATION_ERROR es llamado por un bloque PL/SQL, la ejecución de dicho bloque finaliza y todas las modificaciones realizafdas sobre la BD son anuladas. Dicho de otro modo, se desarrolla un ROLLBACK implícito además de mostrar el mensaje de error.

Ejemplo:

```
IF SAL_EMP * 1.05 > 4000
THEN RAISE_APPLICATION_ERROR (-20010, 'El incremento salarial para el
   empleado número' || TO_CHAR (N_EMP) || ' es demasiado alto' );
```

7. Procedimientos y Funciones

PL/SQL proporciona un lenguaje sofisticado para construir procedimientos y funciones de programa que pueden ser invocados por otros bloques PL/SQL y por otros procedimientos y funciones. La sintaxis para la definición de procedimientos es

```
CREATE [OR REPLACE] PROCEDURE <nombre de procedimiento>
                     (<lista de parámetros>)] IS <declaración>
BEGIN
     <secuencia de sentencias>
     [EXCEPTION <rutinas de manejo de excepciones>]
END [<nombre de procedimiento>];
```

Una función puede ser especificada de una forma parecida:

```
CREATE [OR REPLACE] FUNCTION <nombre de función>
                     [(<lista de parámetros>)]
BEGIN
     <secuencia de sentencias>
   [EXCEPTION <rutinas de manejo de excepciones>]
END [<nombre de función>];
```

La cláusula opcional `OR REPLACE` vuelve a crear el procedimiento o función. Un procedimiento o función puede ser borrado respectivamente con los comandos

```
DROP PROCEDURE <nombre de procedimiento>

DROP FUNCTION <nombre de función>
```

Al contrario que en los bloques PL/SQL anónimos, la clausula `DECLARE` no puede ser utilizada en las funciones/procedimientos definidos.

Los parámetros válidos incluyen todos los tipos de datos. Sin embargo, para los tipos `CHAR`, `VARCHAR2` Y `NUMBER` no deben de especificar longitud ni escala. Por ejemplo, el parámetro `NUMBER(6)` da un error de compilación y debe ser reemplazado por `NUMBER`. Los tipos implícitos del tipo `%TYPE` y `%ROWTYPE` pueden ser utilizados sin ninguna restricción.

Los parámetros se especifican de la siguietne forma:

```
<nombre de parámetro> [IN | OUT | IN OUT]
  <tipo de dato> [{:= |DEFAULT } <expresión>]
```

Las cláusulas `IN`, `OUT`, y `IN OUT` especifican el modo en que es utilizado el parámetro. Por defecto, los parámetros son de modo `IN`. `IN` significa que el parámetro puede estar referenciado dentro del cuerpo del procedimiento, pero no puede ser cambiado. `OUT` significa que al parámetro se le puede asignar un valor, pero el valor del parámetro no puede ser referenciado. `IN OUT` permite ambas cosas, asignar valores a los parámetros y referenciarlos. Normalmente basta con usar el modo por defecto (`IN`).

Ejemplo: El siguiente procedimiento incrementa el salario de todos los empleados que trabajan en un departamento dado como parámetro de entrada al procedimiento. El porcentaje del incremento de salario también es un parámetro de entrada (por defecto 0.5).

```
CREATE PROCEDURE INCREMENTO_SALARIAL (DNO NUMBER,
                                      PORCENTAJE  NUMBER  DEFAULT  0.5)
                                   IS
  CURSOR EMP_CUR (N_DPTO NUMBER) IS
    SELECT SALARIO
    FROM EMPLEADO
    WHERE NUM_DPTO = N_DPTO
  FOR UPDATE OF SALARIO
  SAL_EMP NUMBER (8);
BEGIN
  OPEN EMP_CUR (DNO); - - ASIGNAMOS DNO A N_DPTO
  LOOP
    FETCH EMP_CUR INTO SAL_EMP;
    EXIT WHEN EMP_CUR%NOTFOUND;
    UPDATE EMPLEADO SET SALARIO = SAL_EMP*((100+PORCENTAJE) /100)
    WHERE CURRENT OF EMP_CUR;
  END LOOP;
  CLOSE EMP_CUR;
  COMMIT;
END  INCREMENTO_SALARIAL;
```

Este procedimiento puede ser invocado desde la shell SQL*Plus usando el comando

```
EXECUTE INCREMENTO_SALARIAL(10,3);
```

Si el procedimiento es llamado sólo con el valor 10, se entiende que el valor del parámetro porcentaje es el declarado por defecto: 0.5. Si un procedimiento es llamado desde el bloque PL/SQL la utilización de `EXECUTE` es omitida.

Las funciones tienen la misma estructura que los procedimientos. La única diferencia es que las funciones devuelven un valor de un tipo que debe ser declarado, no existiendo ninguna restricción para ello.

```
CREATE FUNCTION SALARIO_TOTAL_DPTO (DNO NUMBER) R
                                   ETURN NUMBER IS
  SALARIO_TOTAL NUMBER;.
BEGIN
  SALARIO_TOTAL := 0;
    FOR SAL_EMP IN (SELECT SALARIO FROM EMPLEADO
                 WHERE NUM_DPTO = DNO  AND  SALARIO IS NOT NULL) LOOP
    SALARIO_TOTAL := SALARIO_TOTAL + SAL_EMP.SALARIO;
    END LOOP;
  RETURN SALARIO_TOTAL;
END SALARIO_TOTAL_DPTO;
```

Para llamar a una función desde la shell de SQL*Plus es necesario en primer lugar definir la variable en la que se almacenará el valor devuelto. En SQL*Plus una variable puede ser definida utilizando el comando

```
VARIABLE <nombre de variable> <tipo de variable>;
```

por ejemplo, `VARIABLE sueldo NUMBER`. La función anterior puede ser entonces llamada utilizando el comando `EXECUTE`:

```
EXECUTE  :SUELDO := SALARIO_TOTAL_DPTO(20);
```

Tengase en cuenta que el simbolo ":" debe ponerse delante de la variable.

Utilizando el comando `HELP` en SQL*Plus puede obtenerse más información sobre los procedimientos y funciones, por ejemplo `HELP [CREATE] FUNCTION`, `HELP SUBPROGRAM`,

```
HELP STORED SUBPROGRAMS.
```

8. Disparadores (Triggers)

8.1. Visión General

Los diferentes tipos de restricciones de integridad vistos, están muy lejos de proporcionar un mecanismo declarativo para asociar condiciones simples a tablas tal como ocurre con las restricciones de llave primaria, llave foránea o de dominio. Las restricciones complejas de integridad que se refieren a varias tablas y atributos no pueden ser especificadas en la definición de la tabla. Los disparadores, por lo contrario, proporcionan una técnica procedural para especificar y mantener las restricciones de integridad. Los disparadores incluso permiten a los usuarios especificar las condiciones de integridad más complejas, ya que un disparador es esencialmente un procedimiento PL/SQL (asociado con una tabla) que es automáticamente llamado por el SGBD cuando ocurre alguna modificación (evento) en la tabla. Las modificaciones de la tabla pueden incluir operaciones `INSERT, UPDATE, Y DELETE.`

8.2. Estructura de los disparadores

La definición de un disparador consta de los siguientes componentes (opcionales):

- Nombre del disparador

  ```
  CREATE [OR REPLACE] TRIGGER <nombre de disparador>
  ```

- Tiempo o punto del disparador

  ```
  BEFORE | AFTER
  ```

- Eventos disparadores

  ```
  INSERT OR UPDATE [OF <columna(s)>] OR DELETE ON <tabla>
  ```

- Tipo de disparador (opcional)

  ```
  FOR EACH ROW
  ```

- Restricciones disparador (solo para disparador `FOR EACH ROW`)

  ```
  WHEN (<condición>)
  ```

- Cuerpo del disparador

  ```
  <Bloque PL/SQL>
  ```

La cláusula REPLACE vuelve a crear una definición de disparador con el mismo <nombre de disparador>. El nombre de un disparador puede ser elegido arbitrariamente, pero un buen estilo de programación es usar un nombre de disparador que refleje al de la tabla y el evento(s), por ejemplo: UPD_INS_EMP. Un disparador puede ser invocado antes o después del evento disparador (trigger event). Un evento simple es una sentencia INSERT, UPDATE o DELETE; los eventos pueden ser combinados usando la conectiva lógica OR. Si no se especifican columnas en un disparador UPDATE, el disparador es ejecutado cuando la tabla <tabla> es modificada. Si el disparador debe de ejecutarse cuando son actualizadas sólo determinadas columnas, estas columnas deben de ser especificadas después del evento UPDATE.

Para programar disparadores de forma eficiente y correcta es esencial comprender la diferencia entre un disparador-fila o disparador a nivel de fila (row level trigger) y un disparador-sentencia o disparador a nivel de sentencia (statement level trigger). Un disparador a nivel de fila se define utilizando la cláusula FOR EACH ROW. Si no aparece esta cláusula en la definición del disparador, se asume que es un disparador-sentencia. Un disparador-fila se ejecuta una vez por cada fila antes (o después) del evento. Por el contrario, un disparador-sentencia se ejecuta una única vez antes (o después) del evento, independientemente del número de las filas afectadas por el evento. Por ejemplo, un disparador-fila con la especificación de evento AFTER UPDATE es ejecutado una vez por cada fila afectada por la actualización. Por consiguiente, si la actualización afecta a 20 tuplas, el disparador es ejecutado 20 veces. En contraposición, un disparador-sentencia es ejecutado sólo una vez.

Si combinamos los diferentes tipos de disparadores, hay doce posibles configuraciones que pueden ser definidas por la tabla:

Tipos de disparadores.

Evento	Punto del disparador		Tipo de disparador	
	BEFORE	AFTER	SENTENCIA	FILA
INSERT	X	X	X	X
UPDATE	X	X	X	X
DELETE	X	X	X	X

Los disparadores-fila tienen algunas características especiales de la que carecen los disparadores-sentencias. Sólo con un disparador-fila es posible acceder a los valores de los atributos de una tupla antes y después de la modificación (ya que el disparador es ejecutado una vez por cada tupla). Para un disparador UPDATE podemos acceder al valor que posee el atributo antes de disparo (valor antiguo) utilizando :OLD.<columna> y podemos acceder al valor de atributo tras el disparo (valor nuevo) utilizando :NEW.<columna>. Para un disparador INSERT sólo puede ser utilizado :NEW.<columna> y para un disparador DELETE sólo puede ser utilizado :OLD.<columna> (ya que en un borrado no existen valores nuevos, unicamente existen valores viejos de tupla). En estos casos, :NEW.<columna> se refiere a valores de atributos de <columna> en la tupla insertada, y :OLD.<columna> se refiere al valor de atributo de la columna de la tupla borrada. En un disparador-fila es posible especificar comparaciones entre un nuevo y antiguo valor de atributo en el bloque PL/SQL. Por ejemplo, "IF :OLD.SALARIO := :OLD.SALARIO THEN... ". Si un disparador-fila es especificado como BEFORE, es incluso posible modificar los nuevos valores de la fila, ej., :NEW.SALARIO:=:NEW * 1.05 or :NEW.SALARIO:=:OLD.SALARIO. Estas modificaciones no son posibles con disparadores-fila AFTER. En general, es aconsejable usar un disparador-fila AFTER si la nueva fila no es modificada en el bloque PL/SQL. Los disparadores-sentencia son usados generalmente en combinacion con AFTER.

En una definición de disparador, la cláusula WHEN sólo puede ser utilizada en combinación con la opción FOR EACH ROW. Esta cláusula es utilizada para restringir cuándo se ejecuta el disparador. Para la especificación de la condicion en la cláusula WHEN, se siguen las mismas normas que para la clausula CHECK. Las únicas excepciones son que las funciones SYSDATE y USER pueden ser utilizadas, y que es posible referirse al viejo/nuevo valor de atributo de la fila actual. En el caso anterior, el símbolo ":" no debe ser usado (OLD.<atributo> y NEW.<atributo>).

El cuerpo del disparador consiste en un bloque PL/SQL. Todos los comandos PL/SQL pueden ser utilizados en un bloque disparador, a excepción de las sentencias `COMMIT` y `ROLLBACK`. Además, la utilización adicional de la estructura `IF` permite ejecutar ciertas partes del bloque PL/SQL dependiendo del evento disparador. Para ello, existen las tres construcciones `IF INSERTING`, `IF UPDATING[('<columna>')]`, y `IF DELETING`, que pueden ser utilizadas de la forma que muestra el siguiente ejemplo:

```
CREATE OR REPLACE TRIGGER COMPROBAR_EMPLEADO
     AFTER INSERT OR DELETE OR UPDATE ON  EMPLEADO
   FOR EACH ROW
BEGIN
   IF INSERTING THEN
     <BLOQUE PL/SQL>
   END IF;
   IF UPDATING THEN
     <BLOQUE PL/SQL>
   END IF;
   IF DELETING THEN
     <BLOQUE PL/SQL>
   END IF;
END;
```

8.3. Ejemplos de disparadores

Supongamos que tenemos que mantener la siguiente restricción de integridad:

> "El salario de un empleado diferente del presidente no puede ser incrementado ni decrementado más de un 10%. Además, dependiendo del trabajo, cada salario debe estar dentro de un cierto rango.

```
CREATE OR REPLACE TRIGGER COMPROBAR_INCREMENTO_SALARIO_EMPLEADO
   AFTER INSERT OR UPDATE OF  SALARIO, TAREA ON EMPLEADO
   FOR EACH ROW  WHEN (NEW.TAREA != 'PRESIDENTE')
     -- RESTRICCIÓN DE DISPARADOR
DECLARE
   MINSAL, MAXSAL NIVEL.SAL_MAX%TYPE;
BEGIN
   - RECUPERA EL SALARIO MÁXIMNO Y MÌNIMO PARA UNA TAREA
   SELECT SAL_MIN, SAL_MAX INTO MINSAL, MAXSAL
   FROM NIVEL
   WHERE TAREA = :NEW.TAREA;
   --   SI EL NUEVO SALARIO HA SIDO DECREMENTADO O NO ESTÁ DENTRO DE LOS
     LÍMITES, SE PRODUCE UNA EXCEPCIÓN
   IF ( :NEW.SALARIO < MINSAL OR :NEW.SALARIO > MAXSAL) THEN
     RAISE_APPLICATION_ERROR(-20230, 'EL SALARIO HA SIDO DECREMENTADO');
   ELSIF ( :NEW.SALARIO > 1.1 * :OLD.SALARIO) THEN
     RAISE_APPLICATION_ERROR(-20235, 'INCREMENTO SUPERIOR AL 10%');
   END IF;
END;
```

Hemos usado un disparador `AFTER` ya que las filas insertadas y actualizadas no cambian dentro del

bloque PL/SQL (ej., en caso de una violación de una restricción, podríamos restaurar el valor antiguo del atributo).

Téngase en cuenta que las modificaciones en la tabla NIVEL también pueden causar una violación de restricción. Para mantener una condición completa definimos el siguiente disparador en la tabla NIVEL. En caso de una violación por una modificación UPDATE, nosotros no podemos invocar a una excepción, pero podemos restaurar el valor antiguo del atributo.

```
CREATE OR REPLACE TRIGGER COMPROBAR_SALARIO_NIVEL
  BEFORE UPDATE OR DELETE ON NIVEL
  FOR EACH ROW
  WHEN (NEW.SAL_MIN > OLD.SAL_MIN OR NEW.SAL_MAX < OLD.SAL_MAX)
DECLARE
  TAREA_EMP NUMBER(3) := 0;
BEGIN
  IF DELETING THEN
    --PUEDE UN EMPLEADO TENER ASIGNADA UNA TAREA BORRADA?
    SELECT COUNT(*) INTO TAREA_EMP
    FROM EMPLEADO
    WHERE TAREA = :OLD.TAREA;
    IF TAREA_EMP != 0 THEN
      RAISE_APPLICATION_ERROR(-20240,
                'EXISTEN EMPLEADOS CON LA TAREA' || :OLD.TAREA);
    END IF;
  END IF;
  IF UPDATING THEN
    -- HAY EMPLEADOS CUYO SALARIO ESTÁ FUERA DEL RANGO?
     SELECT COUNT(*) INTO TAREA_EMP
    FROM EMPLEADO
    WHERE TAREA = :NEW.TAREA AND SALARIO NOT BETWEEN :NEW.SAL_MIN AND
                                                     :NEW.SAL_MAX;
       IF TAREA_EMP != 0 THEN - - RESTAURAR LOS ANTIGUOS RANGOS DE
SALARIO
      :NEW.SAL_MIN := :OLD.SAL_MIN;
      :NEW.SAL_MAX:= :OLD.SAL_MAX;
    END IF;
  END IF;
END;
```

En este caso hemos utilizado un disparador BEFORE para restaurar el valor antiguo de un atributo perteniciente a una fila actualizada.

Supongamos que además tenemos una columna PRESUPUESTO en nuestra tabla DPTO que es utilizada para almacenar el presupuesto del departamento. Asumamos la restricción de integridad que establece que el total de los salarios de un departamento no puede exceder el presupuesto del departamento. Las operaciones criticas para la tabla EMPLEADO son la inserción de nuevas tuplas y la actualización de los atributos SALARIO o NUM_DPTO.

```
CREATE OR REPLACE TRIGGER  COMPROBAR_PRESUPUESTO_PROYECTO
  AFTER INSERT OR UPDATE OF  SALARIO, NUM_DPTO ON EMPLEADO
DECLARE
  CURSOR DPTO_CUR IS
    SELECT NUM_DPTO, PRESUPUESTO
    FROM DPTO;
  DNO DPTO.NUM_DPTO%TYPE;
  TOTAL_SALARIOS   DPTO.PRESUPUESTO%TYPE,
  DPTO_SALARIOS    NUMBER;
BEGIN
  OPEN DPTO_CUR
  LOOP
    FETCH DPTO_CUR INTO DNO, TOTAL_SALARIOS;
```

```
      EXIT WHEN DPTO_CUR %NOTFOUND;
      SELECT SUM(SALARIO) INTO DPTO_SALARIOS
      FROM EMPLEADO
      WHERE NUM_DPTO = DNO;

      IF DPTO_SALARIOS > TOTAL_SALARIOS THEN
        RAISE_APPLICATION_ERROR(-20325,
          'EL TOTAL DE LOS SALARIOS DEL DEPARTAMENTO.  ' || TO_CHAR(DNO)
           || 'EXCEDE DEL PRESUPUESTO');
      END IF;
    END LOOP;
    CLOSE DPTO_CUR;
END;
```

En este caso, utilizamos una sentencia disparador en la tabla `EMPLEADO` porque tenemos que aplicar una función agregada al salario de todos los empleados que trabajan en un departamento en particular. Para la tabla `DPTO`, debemos definir otro disparador, pero en este caso, puede ser definido como un disparador-fila.

8.4. Programación de disparadores

A la hora de programar, los diparadores-fila son el tipo de disparador más problemático, ya que incluyen varias restricciones. Para asegurar la consistencia de lectura, Oracle realiza un bloqueo de la tabla desde el comienzo de las sentencias `INSERT, UPDATE,` o `DELETE`. Esto quiere decir, que otros usuarios no pueden acceder a esta tabla hasta que las inserciones/modificaciones/borrados se hayan completado satisfactoriamente. En este caso, la tabla sobre la que se realiza la operación se denomina tabla mutante (mutating table). La única forma de acceder a una tabla mutante en un disparador es utilizando `:OLD.<columna>` y `:NEW.<columna>` en conexión con una fila disparador.

Ejemplo de un disparador-fila erróneo:

```
CREATE TRIGGER COMPROBAR_SALARIO_EMPLEADO
  AFTER UPDATE OF SALARIO ON EMPLEADO
  FOR EACH ROW
DECLARE
  SUM_SAL NUMBER;
BEGIN
  SELECT SUM(SALARIO)
  INTO SUM_SAL
  FROM EMPLEADO;
 . . .;
END;
```

Si una sentencia `UPDATE` de la forma

```
UPDATE EMPLEADO SET SALARIO = SALARIO * 1.1;
```

es ejecutada en la tabla `EMPLEADO`, el disparador de arriba es ejecutado una vez por cada fila modificada. Mientras la tabla está siendo modificada por el comando `UPDATE`, no es posible acceder a todas las tuplas de la tabla usando el comando `SELECT`, porque este está cerrado. En este caso obtendremos los siguientes mensajes de error:

```
ORA-04091:  la tabla EMPLEADO es mutante, el disparador no debe ni leer ni
            modificar
ORA-06512:  en la línea 4
ORA-04088:  error durante la ejecución del disparador
            'comprobar_salario_empleado'
```

La única forma de acceder a la tabla, y más concretamente a la tupla, es usar :OLD.<columna> y :NEW.<columna>.

Se recomienda seguir las reglas de abajo para la definición de integridad manteniendo disparadores:

Identificar operaciones y tablas criticas para cada restricción de integridad
  Para cada tabla chequear
    If la restricción puede ser comprobada a nivel de fila **then**
      If las filas comprobadas son modificadas dentro del disparador **then**
          usar disparador-fila tipo **before**
      else
          usar disparador-fila tipo **after**
    else
    usar disparador-sentencia tipo **after**

Los disparadores no son utilizados exclusivamente para el mantenimiento de la integridad. Pueden se utilizados también para:

- Control de acceso a los usuarios y modificaciones en ciertas tablas
- Monitorización de operaciones sobre tablas:

```
CREATE TRIGGER LOG_EMP
   AFTER INSERT OR UPDATE OR DELETE ON EMPLEADO
BEGIN
   IF INSERTING THEN
      INSERT INTO EMP_LOG VALUES (USER, 'INSERT' , SYSDATE);
   END IF;
   IF  UPDATING  THEN
      INSERT INTO EMP_LOG VALUES(USER, 'UPDATE', SYSDATE);
   END IF;
   IF DELETING THEN
      INSERT INTO EMP_LOG VALUES(USER, 'DELETE', SYSDATE);
   END IF;
END;
```

- La propagación automática de las modificaciones. Por ejemplo, si un jefe es transferido a otro departamento, puede ser definido un disparador que automáticamente transfiera a los empleados del jefe al nuevo departamento.

8.5. Mas sobre disparadores

Si un disparador se define el SQL*Plus shell, la definición debe terminar con un punto "." en la ultima línea. Una definición disparador puede ser cargada desde un fichero utilizando el comando @. Téngase en cuenta que la última línea del fichero debe contener un slash " / "

Una definición disparador no puede ser cambiada, sólo puede ser recreada usando la cláusula `OR REPLACE`.. El comando `DROP TRIGGER <nombre de disparador>` borra un disparador.

Después de que una definición disparador haya sido compilada satisfactoriamente, el disparador es habilitado automáticamente. El comando

```
ALTER TRIGGER <nombre de disparador> DISABLE;
```

es utilizado para deshabilitar un disparador. Todos los disparadores definidos en una tabla pueden ser (des)habilitados utilizando el comando

```
ALTER TABLE <nombre de tabla> ENABLE | DISABLE ALL TRIGER;
```

El diccionario de datos guarda información acerca de los disparadores en la tabla `USER_TRIGGERS`. La información incluye el nombre del disparador, tipo, tabla y el código para el bloque PL/SQL.

4
Bibliografía

1. Koch, G. "*ORACLE 7 : manual de referencia*", McGraw Hill, 1994.

2. Muller, R.J. "*Oracle developer/2000 handbook*", McGraw-Hill, 1997

3. Oracle Corporation. "*Oracle7 Server SQL Reference*", 1995.

4. Oracle Corporation. "*Oracle7 Server Application Developer's Guide*", 1995.

5. Oracle Corporation. "*Forms 4.5 Developer's Guide Manual*", 1995.

6. Oracle Corporation. "*Reports 2.5 Building Reports Manual*", 1995.

7. Pepin, D. "*Oracle : guia de referencia para el programador*", Anaya multimedia, 1991.

8. Urman. "*ORACLE8. Programación PL/SQL*", McGraw Hill, 1998.