SQL

Índice

El select de SQL

1. Introducción

Prácticamente la totalidad de los SGBD relacionales utilizan para realizar consultas y solicitudes de modificación de los datos y/o estructuras de datos un lenguaje denominado SQL (*Structured Query Language* = Lenguaje de Consulta Estructurado). El SQL incluye algunas características del álgebra relacional, aunque en realidad está basado en gran parte en el cálculo relacional orientado a tuplas, pero con una sintaxis más amigable que esos lenguajes.

Originalmente, SQL se llamaba SEQUEL (*Structured English QUEry Language*) y fue diseñado e implementado por IBM Research como interfaz para el SGBD relacional experimental SYSTEM R. Un esfuerzo conjunto de ANSI (American National Standards Institute) e ISO (International Standards Organization) ha dado lugar a una versión estándar de SQL [Ame86] llamada SQL-86 o SQL1. Posteriormente, se desarrollaron dos versiones revisadas y más expandidas, la primera se llama SQL2 (también llamado SQL-92), la más reciente es el nuevo estándar, denominado SQL:1999 [MS02], que extiende SQL con conceptos de orientación a objetos y otros conceptos novedosos de bases de datos.

SQL es un lenguaje de BD global; cuenta con enunciados de definición, consulta y actualización de datos. Así pues, es tanto un lenguaje de definición de datos (LDD) como un lenguaje de manipulación de datos (LMD). Además, cuenta con mecanismos para especificar seguridad y autorización, para definir restricciones de integridad, y para especificar controles de transacciones. También tiene reglas para insertar sentencias de SQL en lenguajes de programación de propósito general como C, Pascal o Java.

Existe una diferencia muy importante entre SQL y el modelo relacional formal: SQL permite que tablas (relaciones) tengan dos o más tuplas idénticas en todos los valores de sus atributos. Por tanto, en general, *una tabla de SQL no es un CONJUNTO de tuplas* ya que los conjuntos no permiten dos miembros idénticos; más bien, es un *MULTICONJUNTO* (bolsa) de tuplas.

A pesar de que, como hemos visto, hay estándares, cada fabricante por unos motivos u otros introduce pequeños cambios. En nuestro caso ante tales situaciones presentaremos la alternativa de Oracle, ya que es el SGBD que se utiliza en esta Facultad.

Supondremos que el usuario está familiarizado con el uso de la herramienta `SQL*Plus` de Oracle.

2. Tipos de Datos

SQL emplea los términos *tabla, fila* y *columna* en lugar de relación, tupla y atributo, respectivamente. Nosotros usaremos de manera indistinta los términos correspondientes.

Como sabemos, las atributos están definidos sobre dominios, aunque en SQL2 se pueden definir dominios, en la mayoría de los casos los atributos en SQL se definen sobre tipos de datos que hacen las veces de dominio del atributo.

Cuando se define una nueva tabla, además de asignar nombres a sus columnas, se le asigna a cada una de ellas un determinado tipo de datos (o dominio). Con ello se está definiendo:

Algunas partes de este capítulo se ha realizado tomando parte de los apuntes de Eva Lorenzo Iglesias y Miguel Rodríguez Penabad.

- El conjunto de todos los valores posibles que puede tomar la columna. El SGBD se responsabiliza de que los valores que tome la columna en todo momento sean válidos; es decir, que estén incluidos en el dominio correspondiente.
- Las operaciones que se pueden realizar con los valores de la columna (por ejemplo, si se pueden hacer o no cálculos aritméticos).

La decisión de cuál es el tipo de datos más conveniente para una determinada columna suele tomarse al diseñar la base de datos. Pero, en todo caso, el usuario debe conocerlo, a fin de poder expresar correctamente las operaciones a realizar con los valores de las columnas.

Los tipos de datos pueden agruparse en tres categorías:

- Numéricos.
- Strings.
- De tiempo.

2.1. Datos numéricos

2.1.1. Oracle

Aunque existen diversas formas de representar datos numéricos en Oracle, las principales son las siguientes:

- `NUMERIC(m[,n])` o `NUMBER(m[,n])`, donde m es la precisión o anchura máxima del valor numérico, y n es la precisión decimal (la opción preferida por SQL estándar es `NUMERIC`, pero Oracle, en versiones anteriores, sólo aceptaba `NUMBER`). Por ejemplo, `NUMBER(3)` es un número entero de tres dígitos como máximo, y `NUMBER(5,2)` es un real con 5 dígitos máximo en total, de los que 2 (también como máximo) son decimales.
- `INT`: Es un número entero, que para Oracle es equivalente a `NUMBER(38)`.

Los valores numéricos en Oracle se representan como en cualquier lenguaje de programación, como por ejemplo `26` o `143.05`. No necesitan ir entre comillas, y el separador decimal es normalmente un punto.

2.1.2. SQL2

Incluyen números enteros de diversos tamaños (`INTEGER` o `INT` y `SMALLINT`) y números reales de diversas precisiones (`FLOAT`, `REAL`, `DOUBLE PRECISION`). También podemos declarar números con formato empleando `DECIMAL(m,n)` o `DEC(m,n)` o `NUMERIC(m,n)` donde m y n tienen el mismo significado que en el caso de Oracle.

2.2. Tipos de datos de cadenas de caracteres

2.2.1. Oracle

Existen en Oracle los siguientes tipos para representar un *string* o cadena de caracteres:

- `CHAR[(n)]`: Representa un carácter o una cadena de hasta n caracteres. Si no se indica longitud (indicando sólo `CHAR`) es equivalente a `CHAR(1)`.

- `VARCHAR2(n)` (o `CHARACTER VARYING(n)`): Una cadena de caracteres de hasta n caracteres. Para este tipo de datos es obligatorio indicar la longitud.

Los datos de este tipo se representan en Oracle entre comillas simples ('), como por ejemplo `'casa'`. A nivel de diseño, no hay diferencia entre los tipos `CHAR(n)` y `VARCHAR2(n)`. A nivel físico, el tipo `CHAR(n)` siempre almacena n bytes, mientras que `VARCHAR2(n)` sólo almacena los bytes necesarios para el dato a incorporar en la base de datos. Por ejemplo, si definimos un atributo como `CHAR(2000)` e insertamos un valor `'A'`, ocupará 2000 bytes de todas formas, mientras que si se define como `VARCHAR2(2000)`, ocupará un byte (más, posiblemente, otros dos para indicar la longitud ocupada). La ventaja de los tipos `VARCHAR2` es, pues, que no desperdician espacio. La ventaja del tipo `CHAR` es que, al ocupar siempre lo mismo, todas las tuplas de una tabla tendrán la misma longitud, por lo que es más rápido acceder a ellas en disco, ya que se conoce exactamente donde empieza y acaba cada una de ellas.

Los tipos `CHAR` y `VARCHAR2` admiten una longitud máxima de 2000 caracteres. Para textos de más de esa longitud se usará el tipo `LONG`. No debe confundirse este tipo con el tipo `long` de muchos lenguajes de programación en los que es un entero largo, en Oracle es de tipo carácter.

2.2.2. SQL2

Las cadenas de longitud fija pueden ser `CHAR(n)` o `CHARACTER(n)`, las de longitud variable `VARCHAR(n)`, `CHARVARYING(n)` o `CHARACTER VARYING(n)`.

2.3. Tipos de datos temporales (fechas, horas)

2.3.1. Oracle

Oracle usa únicamente el tipo de datos `DATE` para representar datos temporales, lo que incluye la fecha (día del mes, mes y año) y hora (hora, minutos y segundos, e incluso fracciones de segundos).

Estos tipos de datos siempre han sido en cierta medida problemáticos, debido a las distintas formas de representar el tiempo. Por ejemplo, podemos usar el formato de 12 o 24 horas, o el formato europeo (día-mes-año) o americano (mes-día-año) para representar una fecha.

Si se desea introducir una fecha (sin hora asociada), se puede hacer directamente, como si fuese una cadena de caracteres, entre comillas, usando el formato por defecto, que depende de la instalación. Alguna instalación usa por defecto el formato `dd-mmm-yyyy`, donde `dd` son dos dígitos numéricos para el día del mes, `mmm` representa las tres primeras letras del mes y `yyyy` cuatro dígitos para el año (también pueden usarse 2). Así, el dato `'06-JAN-2002'` representa el 6 de Enero de 2002. Otras instalaciones usan por defecto `dd/mm/yy`, donde `dd` son de nuevo dos dígitos numéricos para el día del mes, `mm` dos dígitos para el número del mes, y `yy` dos dígitos para el año.

Si no se indica la hora, se presupone que son las cero horas, cuando empieza el día indicado. Sin embargo, usar el formato por defecto puede no ser una buena idea, ya que este puede variar dependiendo de la instalación. Para evitar estos posibles problemas, es conveniente utilizar las siguientes funciones que convierten una fecha (u hora) a cadena de caracteres, y viceversa:

- `TO_CHAR(<fecha_hora>, <formato>)`, que convierte una fecha/hora a string.

- `TO_DATE(<fecha_hora>, <formato>)`, que convierte de string a fecha/hora.

El `<formato>` será una cadena de caracteres entre comillas, con espacios, guiones, y caracteres especiales para formatear las fechas y horas. Entre estos caracteres encontramos:

`dd`: Día del mes (2 dígitos).

`mm`: Número del mes (2 dígitos).

`mon`: Nombre del mes (3 primeras letras).

`yyyy`: Año (4 dígitos), aunque también se puede usar `yy` para usar sólo 2.

`hh`: horas (2 dígitos). Puede ser `hh12` o `hh24` para 12 o 24 horas (si sólo se especifica `hh`, se sobrentiende `hh12`)

`mi`: Minutos (2 dígitos).

`ss`: segundos (2 dígitos).

Veamos ahora algunos ejemplos: `TO_DATE('06/01/2002', 'dd/mm/yyyy')` devolvería un valor de tipo fecha almacenando el día 6 de enero de 2002, a las cero horas. `TO_CHAR(ATRIBUTO_FECHA, 'dd-mm-yy hh:mi')`, devolvería la cadena de caracteres `'06-01-2002 16:22'`, suponiendo que ese fuese el valor almacenado en el atributo `ATRIBUTO_FECHA`.

2.3.2. SQL2

El tipo de datos `DATE`, sin embargo, en SQL2 almacena fechas, con los componentes `YEAR`, `MONTH` y `DAY`, por lo regular en la forma `YYYY-MM-DD`. Para almacenar las horas se dispone del tipo de datos `TIME`, con los componentes `HOUR`, `MINUTE` y `SECOND`, normalmente en la forma `HH:MM:SS`. `TIME` tiene dos argumentos opcionales. `TIME(i)`, donde i es la precisión en fracciones de segundo, tendría el formato `HH:MM:SS:`f_1`, ..., `f_i. Un tipo de datos `TIME WITH TIME ZONE` contiene seis "caracteres" extra para especificar el *desplazamiento* respecto a la zona horaria estándar universal, que está en el intervalo de `+13:00` a `-12:59` en unidades `HOURS:MINUTES`. Si no se incluye WITH TIME ZONE, el valor por omisión es la zona horaria local de la sesión SQL.

Un tipo de datos `TIMESTAMP` incluye los campos DATE y TIME, más un mínimo de seis "caracteres" para fracciones de segundo y un calificador opcional `TIME WITH TIME ZONE`.

Otro tipo de datos relacionado con `DATE`, `TIME` y `TIMESTAMP` es el tipo de datos `INTERVAL`. Éste especifica un *intervalo*, un valor *relativo* que puede servir para incrementar o disminuir un valor absoluto de fecha, hora o marca temporal. Los intervalos se califican con `YEAR/MONTH` o `DATE/TIME`.

2.4. Valores nulos

Como veremos más adelante con más detalle, cuando se define un atributo, para cualquiera de los tipos admitidos, puede especificarse si admite valores nulos (`NULL`) o no (`NOT NULL`). Por defecto, si no se indica nada, supone `NULL`, con lo que sí admite valores nulos.

Ejemplo 2.1 Aunque trataremos más adelante el tema del LDD en SQL, para ilustrar el uso de los tipos de datos vamos a ver dos sentencias SQL.

La primera es una sencilla sentencia de creación de una tabla:

```
CREATE TABLE EMP(
    EMPNO    VARCHAR2(4) NOT NULL,
    ENAME    VARCHAR2(15) NOT NULL,
    JOB      VARCHAR2(9),
    MGR      VARCHAR2(4),
    HIREDATE DATE,
    SAL      NUMBER(7,2),
    COMM     NUMBER(7,2),
    DEPTNO   VARCHAR2(2))
```

La siguiente sentencia SQL inserta una tupla en la tabla creada:

```
INSERT INTO EMP VALUES('1245', 'José', 'Analista', NULL,
'12-Jan-1997', 34000, 300.05, '12')
```

En este ejemplo se ha utilizado una abreviatura del tipo de datos `DATE` que permite introducir fechas como una cadena de caracteres. Si utilizáramos la función `TO_DATE` la sentencia anterior quedaría:

```
INSERT INTO EMP VALUES('1245', 'José', 'Analista', NULL,
TO_DATE('12/01/1997', 'dd/mm/yyyy'), 34000, 300.05, '12')
```

□

3. Tablas de referencia

Antes de continuar, veamos las tablas con las que vamos a trabajar. Se utilizan las tablas EMP (empleados) y DEPT (departamentos). Para ver su descripción utilizaremos la sentencia `DESCRIBE <tabla>`. `DESCRIBE` no es una orden de SQL, es un comando del `SQL*Plus` de Oracle que nos indica las columnas (atributos) de una tabla, indicando además su tipo y si acepta o no valores nulos.

```
SQL> DESCRIBE EMP
 Nombre                                    +Nulo?   Tipo
 ----------------------------------------- -------- -------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
 JOB                                                VARCHAR2(9)
 MGR                                                NUMBER(4)
 HIREDATE                                           DATE
 SAL                                                NUMBER(7,2)
 COMM                                               NUMBER(7,2)
```

```
DEPTNO                                                NUMBER(2)

SQL> DESCRIBE DEPT
Nombre                                    +Nulo?   Tipo
----------------------------------------- -------- -------------
DEPTNO                                             NUMBER(2)
DNAME                                              VARCHAR2(14)
LOC                                                VARCHAR2(13)
```

La clave primaria de EMP es EMPNO (número de empleado) y la de DEPT es DEPTNO (número de departamento). En la tabla EMP hay dos claves externas, una es DEPTNO que hace referencia a la tabla DEPT. Evidentemente especifica el departamento (único) para el cual trabaja cada empleado. La otra clave externa en EMP es MGR, que hace referencia a la propia tabla EMP, indica para cada empleado el empleado que lo supervisa.

El contenido de las tablas se puede consultar en el Apéndice **??**.

4. Expresiones

Una expresión es la formulación de una secuencia de operaciones que, cuando se ejecuta, devuelve *un único valor como resultado*. Se formula como una combinación de operadores, operandos y paréntesis.

Los operandos son normalmente valores (números, tiempo o strings). De manera que, por ejemplo, una constante, nombres de columna u otra expresión, pueden actuar como operandos. Hay otros elementos que también pueden actuar como operandos como veremos más adelante (funciones, variables de programación y otros).

Los valores que actúan como operandos de una operación determinada han de ser homogéneos, es decir, o bien numéricos, o bien strings o bien fechas/horas.

Con los valores numéricos se pueden hacer operaciones aritméticas, cuyos operadores son: +, -, * y /. Si un operando es `Nulo`, el resultado también lo es.

Con los strings se puede usar el operador *concatenación*, que se escribe `CONCAT`, o también como dos rayas verticales `||`. Sus operandos han de ser dos strings y el resultado es otro string que se forma concatenando los dos strings. Si un operando es nulo el resultado también lo es.

Es válido que una expresión contenga un solo valor (nombre de columna, constante ...), sin especificar operaciones sobre él. También se considera una expresión válida una sentencia `SELECT`[1], entre paréntesis, que devuelva un solo valor (es decir, una sola fila con una sola columna). Este tipo de sentencias se denomina `Select` *escalar*. Si una de estas sentencias tuviera como resultado una tabla vacía, a efectos de considerarla una expresión es como si devolviera el valor *Nulo*.

Ejemplo 4.1 Ejemplos de expresiones:

```
3+2
'A'||'BC'
SAL*1.5
0.5*COMM
```

[1] Veremos pronto la definición de una sentencia Select.

```
3
'Casa'
ENAME
(Select COMM FROM EMP WHERE EMPNO = 7499)
```

El resultado de `(Select COMM FROM EMP WHERE EMPNO = 7499)` es 300, por lo tanto (en este caso), se puede ubicar esta sentencia Select en cualquier sitio donde se espere un número.□

Obsérvese que $COMM < 15$ NO es una expresión, ya que no devuelve un valor escalar.

5. SQL como DML (Lenguaje de Manipulación de Datos)

A continuación presentamos algunas *convenciones tipográficas* que se utilizarán.

> En este documento, el código SQL está escrito en letra `monoespaciada`. Cuando algún argumento es opcional, se muestra entre corchetes `[]`, y un argumento entre los símbolos menor y mayor indica que debe ser sustituido por una expresión o un identificador válido.
>
> Por ejemplo, `<nombre_atributo> <tipo> [NOT NULL][<Restriccion>]` podría sustituirse, por ejemplo, por `NIF CHAR(12) NOT NULL PRIMARY KEY` o por `NIF CHAR(12)`.
>
> Finalmente, `{ <a> | <b> }` significa una alternativa: en su lugar se puede escribir o bien `<a>` o bien `<b>` (realmente, las expresiones que lo sustituyan). Por supuesto, las llaves no se escribirán.

5.1. El select básico

Una de las operaciones más importantes que se realizan sobre los datos almacenados en una base de datos es la selección, o consulta, de datos. Todos los listados e informes que cualquier empresa emite provienen de una selección de datos.

En SQL existe una única sentencia para seleccionar datos de una o varias tablas: la sentencia `SELECT`. Aunque es una única sentencia, su potencia nos permite realizar directamente la mayoría de las selecciones de datos que necesitamos. La sintaxis general de esta sentencia es la siguiente:

```
SELECT [DISTINCT|ALL] {* | <expr1>[, <expr2>] ...}
   FROM <tabla1>[,<tabla2>,...]
   [WHERE <condicion_where>]
   [GROUP BY <groupexpr1>[,<groupexpr2>,...]
   [HAVING <condicion_having>]
   [ORDER BY <expr_orderby1>[,..]]
```

El propósito del `SELECT` es recuperar y mostrar datos de una o más tablas. Como veremos es una sentencia extremadamente potente capaz de realizar operaciones equivalentes a la *Selección, Proyección* y *Join* del álgebra relacional en una única sentencia.

La estructura básica del `Select` consiste en tres cláusulas:

1. La cláusula *Select* corresponde (en gran medida) a la operación de *Proyección* del álgebra relacional.

2. La cláusula *from* corresponde (en gran medida) a la operación del *producto cartesiano* del álgebra relacional.

3. La cláusula *where* corresponde a la *Selección* del producto cartesiano.

La elección de los nombres no ha sido muy afortunada, si los queremos comparar con el álgebra relacional.

Como vemos en la estructura de la sentencia `Select`, en la cláusula *Select* podemos especificar *expresiones*, generalmente nombres de columna de tablas separadas por comas. Esta lista la llamaremos *"lista SELECT"*. Si en su lugar se especifica un * sería equivalente a especificar una lista con los nombres de todas las columnas de todas las tablas abarcadas por la sentencia `Select`.

En la cláusula *from* especificaremos nombres de tablas (aunque como veremos más adelante hay otras posibilidades). En la cláusula *where* se especifica una condición compuesta por predicados de modo que la condición del *where* puede ser `TRUE` o `FALSE`.

5.1.1. Obtención del resultado de la sentencia `Select`

El resultado de una sentencia `Select` con el formato anterior puede obtenerse siguiendo los pasos que se exponen a continuación. Esto no quiere decir que el SGBD siga estos pasos exactamente. Puede seguir otros procedimientos, si estima que son más eficientes, siempre que produzcan el mismo resultado.

1. De entre todas las tablas existentes en la base de datos se seleccionan las nombradas en la cláusula *from*. Se realiza el producto cartesiano de todas esas tablas.

2. Se aplica el predicado de la cláusula where a todas las filas del producto cartesiano calculado en el punto anterior. Se eliminan las filas que no hagan la condición del *where* `TRUE`.

   La aplicación de la condición se *hace fila a fila*, una cada vez, hasta examinar todas. Cuando le toca el turno a una fila, se evalúa la cláusula *where* sustituyendo en ella los nombres de las columnas que intervengan en las comparaciones, por los respectivos valores que aquéllas tomen en la fila.

3. Del resultado obtenido en el punto anterior, se eliminan las columnas que no se mencionen en la lista SELECT. Las restantes se retienen y se ponen en la misma posición relativa en que se hayan escrito en la lista SELECT. También se calculan las expresiones si aparecen en la lista.

   La tabla así obtenida es el resultado final de la sentencia `Select`.

Ejemplo 5.1 Obtener los nombres de los empleados que trabajan en el departamento 10.

```
SQL> Select ENAME
     from EMP
     where DEPTNO=10;
```

```
ENAME
----------
CLARK
KING
MILLER

3 filas seleccionadas.
```

El resultado es una tabla de una sola columna (ENAME).□

Ejemplo 5.2 Seleccionar todos los datos del departamento 10.

```
SQL> select *
     from DEPT
     where DEPTNO=10;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK

1 fila seleccionada.
```

Aquí el resultado sólo tiene una fila, pero tres columnas. Se ha utilizado el comodín *, como alias de "todas las columnas".□

Ejemplo 5.3 Obtener los datos de todos los departamentos.

```
SQL> select *
     from DEPT

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 ACCOUNTING     NEW YORK
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO
        40 OPERATIONS     BOSTON

4 filas seleccionadas.
```

Como vemos la cláusula *where* es opcional.□

5.1.2. Orden de presentación de las filas del resultado

El resultado de una consulta en el modelo relacional (teórico) no devuelve las filas en ningún orden determinado. De hecho, realizando dos veces la misma consulta podría ocurrir (no ocurrirá normalmente en un gestor real, pero podría) que las tuplas obtenidas por la primera estuviesen en distinto orden a las obtenidas por la segunda. Para forzar un determinado orden, se puede utilizar la cláusula `ORDER BY` en la sentencia `SELECT`, que sigue la sintaxis:

```
ORDER BY <order_expr1> [ASC|DESC] [,<order_expr2> [ASC|DESC], ...]
```

Cada una de las `<order_expr>` significa un criterio de ordenación, y puede ser:

- Un atributo de la(s) tabla(s) en el `FROM`. Normalmente el atributo se corresponderá con una de las expresiones seleccionadas en el `SELECT`.
- Una referencia numérica a una de las expresiones en la lista SELECT. Así, un 1 referencia la primera expresión seleccionada, un 2 la segunda, etc. Si queremos ordenar el resultado de una consulta por una expresión que no es un atributo (por ejemplo, `SAL+COMM`), esta es la única opción posible hasta la versión 8.0.X de Oracle. A partir de la versión 8*i* (8.1.X) ya es posible ordenar por expresiones. Además, en algunos tipos de consultas (que incluyen la cláusula `UNION`) también es la única opción válida, con la misma consideración sobre las versiones de Oracle.

Los criterios de ordenación se siguen de izquierda a derecha. Es decir, se ordena como indique el primer criterio y, si hay repetidos, se considera el segundo criterio, y así sucesivamente. Después del criterio de ordenación, se puede indicar si la ordenación se desea ascendente o descendente. Si no se indica nada, o `ASC`, será ascendente, y si se indica `DESC`, descendente.

Ejemplo 5.4 Obtener los datos de los departamentos ordenado por el nombre de departamento:

```
SQL> select *
  2  from dept
  3  order by DNAME;


    DEPTNO DNAME          LOC
---------- -------------- ---------
        10 ACCOUNTING     NEW YORK
        40 OPERATIONS     BOSTON
        20 RESEARCH       DALLAS
        30 SALES          CHICAGO

4 filas seleccionadas.
```

Las referencias a nombres de columna (o expresiones) se pueden sustituir por el número de la lista SELECT.

```
SQL> select *
  2  from dept
  3  order by 2;
```

Esta consulta devuelve el mismo resultado que la anterior.□

Ejemplo 5.5 Obtener la comisión, departamento y nombre de los empleados cuyo salario sea inferior a 1.900 €, calificándolos por departamento en orden creciente, y por comisión en orden decreciente dentro de cada departamento.

```
SQL> select COMM, DEPTNO, ENAME
  2  from EMP
  3  where SAL < 1900
  4  order by DEPTNO, COMM DESC;

      COMM     DEPTNO ENAME
---------- ---------- ----------
                   10 MILLER
                   20 SMITH
                   20 ADAMS
                   30 JAMES
      1400         30 MARTIN
       500         30 WARD
       300         30 ALLEN
         0         30 TURNER

8 filas seleccionadas.
```

□

5.1.3. Asignación de nombres a las columnas del resultado

El SGBD genera en principio para las columnas de la tabla resultante de un `Select` el mismo nombre que la columna correspondiente o la definición de la expresión que da origen a esa columna. Pero el usuario puede cambiarlo. Para ello escribe el nuevo nombre en la lista SELECT detrás del nombre original de la expresión y de la palabra `AS` (aunque en algunos SGBD's, como Oracle, no es necesaria).

Ejemplo 5.6 Podemos utilizar la consulta del ejemplo anterior renombrando las columnas.

```
  select COMM AS COMISIÓN, DEPTNO AS DEPARTAMENTO, ENAME AS NOMBRE
  from EMP
  where SAL < 1900
  order by DEPTNO, COMM DESC

  COMISIÓN DEPARTAMENTO NOMBRE
---------- ------------ ----------
                    10 MILLER
                    20 SMITH
                    20 ADAMS
                    30 JAMES
      1400          30 MARTIN
       500          30 WARD
       300          30 ALLEN
         0          30 TURNER
```

□

5.1.4. Eliminación de filas repetidas

Como comentamos anteriormente, los SGBD's permiten filas duplicadas (bolsas). Además el resultado de una consulta sobre una tabla, aunque ésta no tenga filas duplicadas, sí puede tener filas duplicadas dependiendo de cómo se formule. El usuario puede dejar estas filas repetidas en el resultado, o puede eliminarlas.

En la cláusula *Select* puede incluirse para ello la palabra predefinida `DISTINCT` *antes* de la lista SELECT. Esto significa que **en el resultado no han de aparecer FILAS repetidas**, si las hubiere. En tal caso, el SGBD las eliminará de la tabla resultante antes de mostrarla al usuario.

Dos filas se consideran repetidas cuando tienen iguales valores en sus columnas. A estos efectos, dos valores *Nulos* se consideran iguales.

Ejemplo 5.7 Hallar todas las combinaciones diferentes de valores de puesto de trabajo (JOB) y año de contratación en el departamento 30.

Si no incluimos la palabra `DISTINCT`, como vemos a continuación, aparecen FILAS repetidas.

```
select job, to_char(HIREDATE,'yyyy') CONTRATADO
from emp
where deptno=30

JOB       CONT
--------- ----
SALESMAN  1981
SALESMAN  1981
SALESMAN  1981
MANAGER   1981
SALESMAN  1981
CLERK     1981

6 filas seleccionadas.
```

Al incluir la palabra `DISTINCT` al *comienzo* de la lista SELECT, se eliminan estas filas.

```
select DISTINCT job, to_char(HIREDATE,'yyyy') CONTRATADO
from emp
where deptno=30

JOB       CONT
--------- ----
CLERK     1981
MANAGER   1981
SALESMAN  1981

3 filas seleccionadas.
```

Obsérvese, que sigue habiendo valores repetidos, pero en el modelo relacional, la relación contiene *tuplas*, y por tanto, **la tupla es el elemento que se repite o no**.□

5.2. La condición del WHERE

Hasta ahora hemos visto algunos ejemplos de consultas con condiciones where muy simples, pero pueden especificarse condiciones más elaboradas. Estas condiciones se llaman predicados.

Un predicado expresa una condición entre valores, y según sean éstos, puede resultar `TRUE`, `FALSE` o `DESCONOCIDO`. La condición expresada por un predicado sólo se considera satisfecha cuando toma el valor `TRUE`, y por tanto el `Select` no devolverá las filas que al ser evaluadas sobre el predicado devuelvan `FALSE` o `DESCONOCIDO`.

Antes de continuar con los predicados del `WHERE`, necesitamos introducir una definición nueva.

5.2.1. Sentencias `Select` subordinadas

A las sentencias `Select` que están dentro de otras las llamaremos *subordinadas*.

5.2.2. Predicados simples

El formato de un predicado simple (o de comparación) es:

```
<expre1> operador <expre2>
```

Son predicados simples aquellos que expresan condiciones de comparación entre dos valores. `operador` se debe sustituir por uno de los operadores de comparación $\{<, \leq, =, \neq, \geq, >\}$.

Si alguno de los dos comparandos, o ambos, son *Nulos*, el predicado toma el valor de `DESCONOCIDO`.

Obsérvese que una expresión es también una `Select escalar` (select que devuelve una fila de una columna). Pero hay que tener cuidado con que la select que introduzcamos como expresión en un predicado simple sea ESCALAR, sino se producirá un error.

Ejemplo 5.8 Obtener por orden alfabético los nombres de los empleados cuyos sueldos igualan o superan al del empleado WARD en más de un 50 %.

```
  Select ENAME
    from EMP
    where sal >= (select sal*1.5
                  from emp
                  where ENAME='WARD');

ENAME
----------
JONES
BLAKE
CLARK
SCOTT
KING
FORD

6 filas seleccionadas.
```

Obsérvese que como sólo hay un empleado con nombre 'WARD' esta consulta es correcta, ya que la consulta subordinada devuelve una fila de una columna.□

Ejemplo 5.9 Para todos los empleados que tienen comisión, hallar sus salarios mensuales totales incluyendo la comisión. Hallar también el porcentaje que de su salario total supone la comisión. Mostrar el resultado por orden alfabético.

```
select ENAME, SAL+COMM AS SAL_TOTAL, (COMM/(SAL+COMM)*100) AS PORCENTAJE
from emp
where SAL+COMM >=0
order by ENAME;
```

```
ENAME       SAL_TOTAL PORCENTAJE
---------- ---------- ----------
ALLEN            1900 15,7894737
MARTIN           2650 52,8301887
TURNER           1500          0
WARD             1750 28,5714286

4 filas seleccionadas.
```

Obsérvese que en aquellos empleados (filas) donde el valor del atributo `COMM` es nulo, el resultado de la expresión `SAL+COMM` es nulo también, y por lo tanto el resultado de evaluar el predicado `SAL+COMM >=0` es `DESCONOCIDO`. Por consiguiente, esas filas no aparecen en el resultado.□

5.2.3. Predicado NULL

El formato de este predicado es:

```
<expre1> IS [NOT] NULL
```

Sirve para preguntar si el valor de una expresión es o no *Nulo*.

Ejemplo 5.10 La consulta del Ejemplo 5.9 puede formularse más claramente con la ayuda de este predicado.

```
select ENAME, SAL+COMM AS SAL_TOTAL, (COMM/(SAL+COMM)*100) AS PORCENTAJE
from emp
where COMM IS NOT NULL
order by ENAME;
```

□

Ejemplo 5.11 Nombre de los empleados que no tienen supervisor.

```
 select ENAME
  from EMP
  where MGR IS NULL;

ENAME
----------
KING

1 fila seleccionada.
```

□

5.2.4. Predicados cuantificados

Cuando se usa una sentencia `Select` subordinada en un predicado simple, su resultado debe ser un valor único, como ya se ha dicho (o sea, debe ser una `Select` ESCALAR).

Sin embargo se admite que el resultado tenga varias filas si la sentencia subordinada va precedida de alguna de las palabras cuantificadoras `ANY, SOME` o `ALL`. Los predicados así construidos se llaman cuantificados. Los cuantificadores `ANY` y `SOME` son equivalentes. El formato es como sigue:

```
<expr1> operador [SOME|ANY|ALL] <sentencia1>
```

o bien

```
<expr1>[,<expr2>,...] = [SOME|ANY|ALL] <sentencia2>
```

En este formato, `operador` es un operador de comparación. `<expr1>` y `<expr2>` son expresiones que no son sentencias `Select`. `<sentencia1>` y `<sentencia2>` son sentencias `Select` subordinadas. `<sentencia1>` debe devolver una tabla de una columna, y `<sentencia2>` debe devolver una tabla con tantas columnas como expresiones halla a la izquierda del igual (=). Con los valores de las expresiones a la izquierda del = se construye una fila que se compara con las filas devueltas por la sentencia select `<sentencia2>`. Para comparar una fila con otra, se comparan los valores de todas las columnas, una a una desde la primera columna (la primera con la primera, la segunda con la segunda,...).

Dicho de manera sencilla, si se usa `ALL`, el predicado cuantificado es `TRUE` si la comparación es `TRUE` para todas y cada una de las filas resultantes de la sentencia subordinada. Si se usa `SOME` o `ANY`, el predicado cuantificado es `TRUE` si la comparación es `TRUE` para una cualquiera de las filas resultantes de la sentencia subordinada. De un modo más detallado:

- Si se especifica ALL:
  - Si la sentencia subordinada devuelve una tabla con cero filas (vacía), el predicado cuantificado toma el valor `TRUE`.
  - Si devuelve una o más filas, y todas ellas al compararlas devuelven `TRUE`, el predicado toma el valor de `TRUE`.
  - Si devuelve una o más filas, y al menos hay una de ellas que al compararla devuelve `FALSE`, el predicado toma el valor `FALSE`.

- Si devuelve una o más filas, y ninguna de ellas da `FALSE` al compararla, y alguna da `DESCONOCIDO`, el predicado toma el valor de `DESCONOCIDO`.

- Si se especifica SOME/ANY:

  - Si la sentencia subordinada devuelve una tabla con cero filas (vacía), el predicado cuantificado toma el valor `FALSE`.
  - Si devuelve una o más filas, y al menos hay una de ellas que al compararla devuelve `TRUE`, el predicado toma el valor `TRUE`.
  - Si devuelve una o más filas, y todas ellas al compararlas devuelven `FALSE`, el predicado toma el valor de `FALSE`.
  - Si devuelve una o más filas, y ninguna de ellas da `TRUE` al compararla, y alguna da `DESCONOCIDO`, el predicado toma el valor de `DESCONOCIDO`.

Ejemplo 5.12 Obtener por orden alfabético los nombres de los empleados cuyo salario supera al máximo salario de los empleados del departamento 30.

```
 select ENAME
  from emp
  where sal > ALL (select SAL
                   from emp
                   where DEPTNO=30)
  order by ENAME;

ENAME
----------
FORD
JONES
KING
SCOTT

4 filas seleccionadas.
```

□

Si en la sentencia `select` anterior sustituyésemos el 30 por un 40, el resultado de la sentencia subordinada sería vacío. Por tanto, todas las filas del EMP satisfacen el predicado cuantificado y el resultado final es una relación de todos los empleados por orden alfabético. Este resultado puede ser inesperado para el usuario.□

Ejemplo 5.13 Obtener por orden alfabético los nombres de los empleados cuyo salario supera en tres veces y media o más al mínimo salario de los empleados del departamento 20.

```
Select ENAME
from emp
where sal/3.5 >= SOME (Select SAL
                       from emp
                       where DEPTNO=20)
```

```
ENAME
----------
JONES
BLAKE
SCOTT
KING
FORD

5 filas seleccionadas.
```

Una vez mas, si sustituimos el 20 por un 40 (departamento sin empleados), el resultado de la consulta subordinada es vacío. El predicado subordinado en tal caso devuelve `FALSE`, por lo tanto, el resultado final también será vacío.□

Ejemplo 5.14 Obtener el nombre, el puesto de trabajo y el número de departamento de los empleados que tienen el mismo puesto de trabajo y trabajan para el mismo departamento que el empleado 7499.

```
SELECT ENAME, JOB, DEPTNO
from emp
where (JOB,DEPTNO) = ALL (Select JOB, DEPTNO
                          from emp
                          where EMPNO=7499)
```

```
ENAME      JOB           DEPTNO
---------- --------- ----------
ALLEN      SALESMAN          30
WARD       SALESMAN          30
MARTIN     SALESMAN          30
TURNER     SALESMAN          30

4 filas seleccionadas.
```

Obsérvese que aunque la consulta subordinada sólo devuelve una fila, si no incluimos el `ALL` la consulta sería incorrecta, debido a que la fila tiene dos columnas.□

5.2.5. Predicado Between

Su formato es:

```
<expr1> [NOT] BETWEEN <expr2> AND <expr3>
```

Sirve para comparar si un valor está comprendido entre otros dos, ambos inclusive, o no. Si se omite el NOT, el predicado es `TRUE` si el valor de la expresión `<expr1>` está comprendido entre el de la expresión `<expr2>` y el de la expresión `<expr3>`, ambos inclusive. Si se especifica el NOT, el predicado es `TRUE` cuando no está comprendido en ese intervalo. La incursión de nulos crea, de nuevo, algunos problemas que debemos detallar:

Supongamos que escribimos `V1 BETWEEN V2 AND V3`:

- Si ninguno de los valores V1, V2 o V3 es nulo, el predicado es `TRUE` si V1 es mayor o igual que V2 y menor o igual que V3. En otro caso es `FALSE`.
- Si alguno de los valores, V1, V2 o V3 es nulo, el predicado toma el valor `DESCONOCIDO`.

Supongamos, ahora que escribimos `V1 NOT BETWEEN V2 AND V3`:

- Si ninguno de los valores V1, V2 o V3 es nulo, el predicado es `TRUE` si V1 es, o bien, menor que V2, o bien mayor que que V3. En otro caso es `FALSE`.
- Si V1 es nulo, el predicado toma el valor `DESCONOCIDO`.
- Si V1 no es nulo:
  - Si V2 y V3 son nulos, el predicado toma el valor `DESCONOCIDO`.
  - Si V2 es nulo y V3 no, el predicado toma el valor `TRUE` si V1 es mayor que V3. En otro caso es `DESCONOCIDO`.
  - Si V3 es nulo y V2 no, el predicado toma el valor `TRUE` si V1 es menor que V2. En otro caso es `DESCONOCIDO`.

Ejemplo 5.15 Obtener por orden alfabético los nombres de los empleados cuyo salario está entre 2.500 € y 3.000 €.

```
 select ENAME
  from emp
  where sal BETWEEN 2500 AND 3000
  order by 1;

ENAME
----------
BLAKE
FORD
JONES
SCOTT

4 filas seleccionadas.
```

□

5.2.6. Predicado Like

El formato de este predicado es:

```
<expr1> [NOT] LIKE <expr2>
```

Las expresiones `<expr1>, <expr2>` deben representar strings. Sirve para buscar combinaciones de caracteres de `<expr1>` que cumplan ciertas condiciones expresadas por `<expr2>`. Así `<expr2>` es una especie de máscara generadora de combinaciones de caracteres. La máscara puede contener cualquier carácter, pero dos de ellos, el guión de subrayar (_) y el

porcentaje (%), tienen un uso especial. El símbolo _ representa a cualquier string de longitud 1, mientras que % representa a cualquier string (de cualquier longitud, incluyendo de longitud cero).

Veamos el comportamiento más detallado:

- Si `<expr1>` o `<expr2>` son nulos, el predicado toma el valor `DESCONOCIDO`.
- Si `<expr1>` y `<expr2>` no son nulos ni cadenas vacías, el predicado es `TRUE` si el valor de `<expr1>` está incluido entre las cadenas que "encajan" en `<expr2>`. En caso contrario, toma el valor `FALSE`.
- Si `<expr1>` y `<expr2>` son ambas cadenas vacías, se conviene que el predicado es `TRUE`.
- Si una de las expresiones es una cadena vacía y la otra no, el predicado es `FALSE`.

Ejemplo 5.16 Obtener los nombres de los empleados cuyo nombre contenga la cadena 'NE'.

```
 select ename
  from emp
  where ename LIKE '%NE%';

ENAME
----------
JONES
TURNER

2 filas seleccionadas.
```

□

Ejemplo 5.17 Obtener los nombres de empleados que tengan un nombre de cinco letras.

```
 select ENAME
  from emp
  where  ENAME LIKE '_____';

ENAME
----------
SMITH
ALLEN
JONES
BLAKE
CLARK
SCOTT
ADAMS
JAMES

8 filas seleccionadas.
```

Y los de los empleados con cinco letras o más:

```
select ENAME
from emp
where ENAME LIKE '_____%'

ENAME
----------
SMITH
ALLEN
JONES
MARTIN
BLAKE
CLARK
SCOTT
TURNER
ADAMS
JAMES
MILLER

11 filas seleccionadas.
```

□

5.2.7. Predicado IN

El formato del predicado IN es como sigue:

```
<expr1> [NOT] IN {<sentencia1> | (<expr2>[,...]) | <expr3>}
```

O bien

```
(<expr4>[,...]) [NOT] IN <sentencia2>
```

El resultado de `<sentencia1>` sólo debe tener una columna. En cambio, el resultado de tanto `<sentencia1>` como `<sentencia2>`, pueden tener cualquier número de filas, inclusive cero. Este predicado sirve para preguntar si el valor de la `<expre1>` está incluido entre los valores especificados detrás de la palabra `IN`.

Veamos qué valor toma el predicado `IN` en las distintas formas en que se puede especificar:

- Supongamos que ser especifica un predicado `IN` de la forma:

  ```
  <expr1> IN (<expr2a>, <expr2b>,..., <expr2n>)
  ```

  Si el valor de `<expr1>` no es nulo y es igual a uno de los valores de la lista que hay entre paréntesis, el predicado es `TRUE`, en caso contrario es `FALSE`. Si `<expr1>` es nulo, el predicado toma el valor `DESCONOCIDO`.

- Si se especifica de la forma:

`<expr1> IN <expr3>`

es equivalente a `<expr1> = <expr3>`.

- Si se escribe de la forma:

  `<expr1> IN <sentencia1>`

  es equivalente al predicado cuantificado `<expr1> =SOME <sentencia1>`.

- Si se escribe una lista de expresiones antes del `IN`, en la forma:

  `(<expr4>[,...]) IN <sentencia2>`

  es equivalente al predicado cuantificado `(<expr4>[,...]) =SOME <sentencia2>`

- `NOT IN` será `TRUE` en los casos que `IN` sea `FALSE`, y viceversa.

Ejemplo 5.18 Obtener los nombres de los empleados cuyo puesto de trabajo sea CLERK, SALESMAN o ANALYST.

```
 select ENAME
  from EMP
  where JOB IN ('CLERK', 'SALESMAN', 'ANALYST');

ENAME
----------
SMITH
ALLEN
WARD
MARTIN
SCOTT
TURNER
ADAMS
JAMES
FORD
MILLER

10 filas seleccionadas.
```

□

Ejemplo 5.19 Obtener por orden alfabético los nombres de los empleados que trabajan en el mismo departamento que SMITH o MILLER.

```
  select ENAME
  from EMP
  where DEPTNO IN (select DEPTNO
```

```
                  from emp
                  where ENAME IN ('SMITH','MILLER'))
  order by 1

ENAME
----------
ADAMS
CLARK
FORD
JONES
KING
MILLER
SCOTT
SMITH

8 filas seleccionadas.
```

□

Ejemplo 5.20 Obtener una lista de los empleados cuyo puesto de trabajo y jefe coincida con los de alguno de los empleados del departamento 10.

```
  select ENAME
  from EMP
  where (JOB, MGR) IN (select JOB, MGR
                       from emp
                       where DEPTNO=10)

ENAME
----------
MILLER
JONES
CLARK
BLAKE

4 filas seleccionadas.
```

□

5.2.8. Predicado Exists

El formato del predicado `Exists` es:

```
EXISTS (<sentencia1>)
```

`<sentencia1>` representa un `Select` subordinado. Su resultado puede tener cualquier número de tuplas y columnas. Este predicado es `TRUE` si el resultado de la sentencia subordinada tiene una o más filas, es decir, si no es una tabla vacía. Es `FALSE` si es una tabla vacía.

Ejemplo 5.21 Obtener los nombres de los departamentos si hay alguno en 'Chicago'.

```
Select DNAME
  from DEPT
  where EXISTS (select *
                from dept
                where loc = 'CHICAGO');

DNAME
--------------
ACCOUNTING
RESEARCH
SALES
OPERATIONS

4 filas seleccionadas.
```

□

5.2.9. Predicados compuestos

Todos los predicados vistos hasta el momento son simples. Los compuestos son combinaciones de otros predicados, simples o compuestos, con los operadores `AND`, `OR` y `NOT`.

Estos conectores son bien conocidos por todos, pero hace falta tener en cuenta el valor `DESCONOCIDO`.

Valor X	Valor Y	X AND Y	X OR Y
TRUE	DESCONOCIDO	DESCONOCIDO	TRUE
FALSE	DESCONOCIDO	FALSE	DESCONOCIDO
DESCONOCIDO	TRUE	DESCONOCIDO	TRUE
DESCONOCIDO	FALSE	FALSE	DESCONOCIDO
DESCONOCIDO	DESCONOCIDO	DESCONOCIDO	DESCONOCIDO

Ejemplo 5.22 Obtener los nombres, salarios y fechas de contratación de los empleados que, o bien ingresaron después de 1-6-81, o bien tienen un salario inferior a 1.500 €. Clasificar los resultados por fecha y nombre.

```
  Select ENAME, SAL, HIREDATE
  from emp
  where hiredate > '1-jun-1981' or sal < 1500
  order by 3,1

ENAME             SAL HIREDATE
---------- ---------- --------
SMITH              800 17/12/80
WARD              1250 22/02/81
CLARK             2450 09/06/81
TURNER            1500 08/09/81
```

```
MARTIN          1250 28/09/81
KING            5000 17/11/81
FORD            3000 03/12/81
JAMES            950 03/12/81
MILLER          1300 23/01/82
SCOTT           3000 09/12/82
ADAMS           1100 12/01/83

11 filas seleccionadas.
```

□

Ejemplo 5.23 Nombre de los empleados que ganan más de 2.500€ en total (salario más comisión).

```
  Select ENAME
  from emp
  where sal > 2500 or (sal+comm)>2500

ENAME
----------
JONES
MARTIN
BLAKE
SCOTT
KING
FORD

6 filas seleccionadas.
```

□

5.3. Funciones

Como vimos, las expresiones pueden contener funciones. Además los operandos de los predicados simples también pueden ser el resultado de una función. Atendiendo al tipo de argumentos y resultado de una función, vamos a considerar dos tipos de funciones:

1. **Funciones escalares.** El resultado es un valor. Por tanto, una función escalar puede utilizarse dentro de una sentencia SQL en cualquier sitio donde se espere un valor. Por ejemplo como operando de una expresión o un predicado simple.

   Puede haber varios argumentos, que se escriben entre paréntesis en forma de lista. Cada argumento es un valor y tiene un significado particular que se describirá en cada caso. Naturalmente estos valores pueden ser expresiones.

2. **Funciones colectivas o de columna.** El resultado es un valor. Pero a diferencia de las escalares, no se pueden utilizar en cualquier sitio que se espere un valor, como veremos más adelante.

El único argumento que aceptan es en realidad un conjunto de valores, es decir es una expresión que representa un conjunto de valores (por ejemplo, un nombre de columna).

5.4. Funciones escalares

5.4.1. Funciones para descartar nulos

Existen dos funciones para descartar nulos: `NVL` y `COALESCE`. La función `NVL` toma dos argumentos:

```
NVL(<expresión1>, <expresión2>)
```

La función evalúa la expresión1, y si esta es un valor nulo, devuelve la expresión2 (que, por supuesto, debe ser del mismo tipo que la expresión1). Así, veamos la siguiente sentencia:

Ejemplo 5.24 Salario total (salario más comisión) de todos los empleados.

```
select ENAME,  NVL(sal+comm,sal) as sal_tot
from emp
```

```
ENAME         SAL_TOT
---------- ----------
SMITH             800
ALLEN            1900
WARD             1750
JONES            2975
MARTIN           2650
BLAKE            2850
CLARK            2450
SCOTT            3000
KING             5000
TURNER           1500
ADAMS            1100
JAMES             950
FORD             3000
MILLER           1300

14 filas seleccionadas.
```

□

Una función similar es `COALESCE`, pero con la diferencia de que permite tener más de dos expresiones.

```
COALESCE(<expresión1>, <expresión2>[,<expresión3>...])
```

El resultado es el primer argumento, de izquierda a derecha, que no sea *nulo*. El resultado es *nulo* si todos lo son.

Ejemplo 5.25 Nombre de los empleados, salario, comisión y el primer valor no nulo de la lista: comisión, salario+comisión y salario.

```
select ENAME, sal, comm, COALESCE(comm, sal+comm,sal) as sal_tot
from emp
```

```
ENAME             SAL       COMM    SAL_TOT
---------- ---------- ---------- ----------
SMITH              800 nulo              800
ALLEN             1600        300        300
WARD              1250        500        500
JONES             2975 nulo             2975
MARTIN            1250       1400       1400
BLAKE             2850 nulo             2850
CLARK             2450 nulo             2450
SCOTT             3000 nulo             3000
KING              5000 nulo             5000
TURNER            1500          0          0
ADAMS             1100 nulo             1100
JAMES              950 nulo              950
FORD              3000 nulo             3000
MILLER            1300 nulo             1300

14 filas seleccionadas.
```

□

5.4.2. Función DECODE

La función `DECODE` es más compleja, y puede tener un número variable de argumentos. También evalúa una expresión, y lo que sigue está formado por pares $(valor_i, resultado_i)$. Si la expresión se evalúa a $valor_i$, la función devuelve $resultado_i$. El último parámetro, `resultado_en_otro_caso`, es el valor que devuelve la función en el caso de que la evaluación de la expresión no coincida con ninguno de los valores indicados. La sintaxis general es la siguiente.

```
DECODE(<expresión>,
       <valor1>, <resultado1>
       [, <valor2>, <resultado2>, ...]
       [, <resultado_en_otro_caso>]
       )
```

Ejemplo 5.26 Supongamos que queremos para cada empleado su nombre y el nombre del departamento para el que trabajan en español.

```
select ENAME, DECODE(DEPTNO, '10', 'CONTABILIDAD',
                             '20', 'INVESTIGACIÓN',
                             '30', 'VENTAS')
 from emp;
```

```
ENAME      DECODE(DEPTNO
```

```
---------- -------------
SMITH      INVESTIGACIÓN
ALLEN      VENTAS
WARD       VENTAS
JONES      INVESTIGACIÓN
MARTIN     VENTAS
BLAKE      VENTAS
CLARK      CONTABILIDAD
SCOTT      INVESTIGACIÓN
KING       CONTABILIDAD
TURNER     VENTAS
ADAMS      INVESTIGACIÓN
JAMES      VENTAS
FORD       INVESTIGACIÓN
MILLER     CONTABILIDAD

14 filas seleccionadas.
```

□

5.4.3. Función LENGTH

Esta función sirve para obtener la longitud de un valor cualquiera, aunque es más apropiada para calcular la longitud de una cadena de texto. Su formato es:

```
LENGTH(<expre1>)
```

`<expre1>` puede ser de cualquier tipo, pero lo más normal es que sea una cadena de texto (string). Si el argumento es *nulo* el resultado de la función también lo es.

Ejemplo 5.27 Obtener la longitud del nombre de aquellos empleados que tengan un nombre de más de 5 letras.

```
 select ENAME, LENGTH(ENAME)
  from emp
  where LENGTH(ENAME)>5;

ENAME      LENGTH(ENAME)
---------- -------------
MARTIN                 6
TURNER                 6
MILLER                 6

3 filas seleccionadas.
```

□

5.4.4. Funciones para el tratamiento de strings

Hay un juego de funciones para manipular strings de caracteres. A modo de ejemplo describiremos una de ellas, el lector interesado puede consultar estas funciones en el manual de su SGBD o cualquier otro manual sobre SQL (por ejemplo, [RMLRO02]).

La función `SUBSTR` devuelve un string contenido en otro, es decir, un substring. Su formato es:

```
SUBSTR(<expre1>,<expre2>[,<expre3>])
```

El primer argumento es el string donde está contenido el substring. El segundo es la posición dentro del primer argumento, donde se empieza a extraer el substring. El tercero (opcional) es la longitud del substring.

El primer argumento es un string del que se desea extraer un string formado por varios caracteres contiguos. El substring se delimita con los dos argumentos restantes, que indican donde empieza y cuántos caracteres incluye. Si no se indica el tercer argumento, se extraen todos los caracteres hasta el final de la cadena original. Si cualquiera de los argumentos es *nulo*, el resultado también lo es.

Ejemplo 5.28 Obtener los nombres abreviados de los departamentos tomando sus primeras 6 letras, por orden alfabético.

```
  Select substr(DNAME, 1, 6) as NOMB_ABRV
   from dept

NOMB_A
------
ACCOUN
RESEAR
SALES
OPERAT

4 filas seleccionadas.
```

□

Otras funciones:

- `INITCAP (<expre1>)`: pone a mayúscula la primera letra de cada palabra contenida en `<expre1>`.
- `LOWER (<expre1>)`: transforma `<expre1>` a minúsculas.
- `UPPER (<expre1>)`: transforma `<expre1>` a mayúsculas.
- `LTRIM (<expre1> [, caracter])`: elimina el carácter *caracter* (o blancos) por la izquierda hasta encontrar el primer carácter que no está en *caracter*.
- `RTRIM (<expre1> [, caracter])`: elimina el carácter *caracter* (o blancos) por la derecha hasta encontrar el primer carácter que no está en *caracter*.

5.4.5. Funciones aritméticas y trigonométicas

También existen multitud de funciones trigonométricas y aritméticas, por ejemplo:

- `ABS(expre)`: valor absoluto.
- `CEIL(expre)`: entero más próximo por encima.
- `FLOOR(expre)`: entero más próximo por debajo.
- `MOD(expre, expre2)`: resto de dividir *expre* entre *expre2*. Si *expre2*=0 devuelve *expre*.
- `POWER(expre, expre2)`: $expre^{expre2}$.
- `ROUND(expre [, m])`: redondeo de *expre* a m dígitos.
- `TRUNC(expre [, m])`: trunca *expre* a m dígitos.
- `SIGN(expre)`:
  - Si $expre < 0$ devuelve (-1).
  - Si $expre = 0$ devuelve 0.
  - Si $expre > 0$ devuelve 1.
- `SQRT(expre)`: raíz cuadrada del valor del *expre*.

El lector interesado puede consultar el manual del SGBD que utiliza, o un manual sobre SQL como por ejemplo [RMLRO02], para encontrar más.

5.4.6. Funciones para datos de tipo fecha

Existe una amplia lista de funciones para ayudar a la manipulación de datos de tipo fecha. La información sobre la fecha se encuentra en una tabla del diccionario de datos, denominada *dual*. Las funciones más importantes son:

- `sysdate`: devuelve la fecha y hora actual.
  - Ej: `select sysdate from dual;`
    - Resultado: 28-FEB-03 si el día actual es 28 de febrero de 2003.
- `last_day`: último día del mes
  - Ej: `select last_day(sysdate) from dual;`
    - Resultado: 31-MAR-03 si el día actual es 12 de marzo de 2003.
- `add_months(d, n)`: suma o resta n meses a partir de la fecha d
  - Ej: `select add_months(sysdate, 2) from dual;`
    - Resultado: 18-MAY-03 si el día actual es 18 de marzo de 2003.
- `months_between(f, s)`: diferencia en meses entre la fecha f y la fecha s

- Ej:

```
select months_between('13-FEB-04','12-NOV-03') from dual

MONTHS_BETWEEN('13-FEB-04','12-NOV-03')
---------------------------------------
                                3,03225806

1 fila seleccionada.
```

- `next_day(d, day)`: la fecha del día especificado de la semana después del día actual
  - Ej: `select next_day(sysdate, 'Lunes') from dual;`
    - Resultado: 20-OCT-03 si el día actual es 14 de octubre de 2003.

5.5. Funciones colectivas o de columna

Estas funciones permiten obtener un solo valor como resultado de aplicar una determinada operación a los valores contenidos en una columna. Por ejemplo, la suma de todos los valores de la columna o su valor medio.

La colección de valores la especifica un argumento que es una expresión. Por ejemplo si A y B son columnas numéricas, sería válido especificar:

```
MAX(A)
MAX(3*A)
MAX((A+B)/2)
```

Estas funciones se pueden utilizar como operandos en la expresiones que se especifican en la lista SELECT. Así, podríamos escribir una cláusula como la siguiente:

`SELECT MAX(A), (MAX(A)+MAX(B*3))/2`

Hemos dicho que el argumento es un conjunto de valores, y por otra parte, que se especifica como una expresión, cuyo resultado, como siempre, es un solo valor, lo que parece una contradicción. En realidad esta expresión se evalúa varias veces, y el conjunto de estos resultados es el argumento de la función.

Para ello, antes de aplicar estas funciones se construyen uno o más grupos con las filas de la tabla. La forma de construir estos grupos la veremos más adelante, por ahora si no se especifican estos grupos, se entiende que hay un único grupo formado por todas las filas que satisfacen la condición del `WHERE`. Para cada una de estas filas se evalúa la expresión que se especifica como argumento de la función, y el conjunto de estos resultados es el argumento de la misma.

Al final el resultado de la sentencia `SELECT` será una tabla con una fila por cada uno de los grupos antes mencionados (como hasta ahora solo consideramos un grupo, la tabla resultante sólo tendrá una fila).

Las funciones de columna no pueden aparecer en el `WHERE` a no ser que aparezcan en la lista SELECT de una consulta subordinada.

Ejemplo 5.29 Obtener los salarios y los nombres de los empleados cuyo salario se diferencia con el máximo en menos de un 40 % de éste.

```
select ENAME, SAL
  from EMP
  where SAL >= (SELECT 0.6*MAX(SAL)
                from EMP)

ENAME             SAL
---------- ----------
SCOTT            3000
KING             5000
FORD             3000

3 filas seleccionadas.
```

Es decir ¡NO SERÍA CORRECTO!:

```
select ENAME, SAL
  from EMP
  where SAL >=  0.6*MAX(SAL)
```

□

En la expresión que se especifica como argumento no puede incluirse un `SELECT` escalar (los que devuelven un resultado).

5.5.1. Formato con una expresión en el argumento

`<func>([ALL|DISTINCT] <expre1>)`

`<func>` puede tomar (entre otros) los siguientes valores:

- `AVG` Halla el valor medio.
- `COUNT` Halla el número de valores (filas).
- `MAX` Halla el valor máximo.
- `MIN` Halla el valor mínimo.
- `STDDEV` Halla la desviación típica.
- `SUM` Halla la suma.
- `VAR` Halla la varianza (sinónimo: VARIANCE).

La función se aplica al conjunto de valores que se obtienen como resultado de evaluar `<expre1>` y que no sean *nulos*. Si este conjunto es vacío, la función `COUNT` da cero y el resto *nulo*.

La palabra `DISTINCT` no se considera parte del argumento de la función. Es una palabra predefinida que indica que antes de aplicar la función al conjunto de valores del argumento, hay que eliminar de éste los valores duplicados si los hubiera. Si no se pone nada o se usa `ALL`, no se eliminaran estos duplicados.

5.5.2. Formato con un asterisco

Este formato sólo es válido con la función `COUNT`:

```
COUNT(*)
```

En este caso el argumento no es un conjunto de valores, sino de filas. La función devuelve como resultado el número de filas que hay en él.

Ejemplo 5.30 Hallar el número de empleados de la empresa:

```
select COUNT(*) AS NUM_EMP
 from emp;

   NUM_EMP
----------
        14

1 fila seleccionada.
```

□

Ejemplo 5.31 Hallar el número de empleados y de distintos departamentos que tienen empleados.

```
  select COUNT(*) AS NUM_EMP, COUNT(DISTINCT DEPTNO) AS NUM_DEPT
  from emp

   NUM_EMP   NUM_DEPT
---------- ----------
        14          3

1 fila seleccionada.
```

□

Ejemplo 5.32 Nombre de los empleados y su salario de aquellos cuyo salario sea más del salario medio de la empresa.

```
  select ENAME, SAL
  from emp
  where sal > (select AVG(SAL)
               from emp)

ENAME             SAL
---------- ----------
JONES            2975
BLAKE            2850
CLARK            2450
```

```
SCOTT            3000
KING             5000
FORD             3000

6 filas seleccionadas.
```

□

Ejemplo 5.33 Hallar cuantas comisiones diferentes hay y su valor medio.

```
 select count(distinct COMM) As COMM, AVG(COMM) AS MEDIA
  from emp

      COMM      MEDIA
---------- ----------
         4        550

1 fila seleccionada.
```

□

5.6. Agrupamiento

En la sección anterior decíamos que las funciones colectivas se aplican a cada grupo, que hasta ahora era el formado por todas las tuplas que pasaban la condición del `WHERE`. En esta sección se va describir cómo se crean grupos más complejos, es decir, crear más de un grupo.

La cláusula `GROUP BY` es opcional, y sirve para crear los grupos antes mencionados. Si se especifica, se escribe detrás de la cláusula `WHERE`, si ésta existe, sino después del `FROM`.

```
GROUP BY (<expre1>[,<expre2>...])
```

La expresión/es suelen llamarse *expresiones de agrupamiento*. Esta cláusula indica que se han de agrupar las filas de la tabla de tal manera que se incluyan en el mismo grupo todas las que proporcionen iguales valores al evaluar las expresiones de agrupamiento. Puede haber grupos que sólo tengan una fila. Los valores *nulo* se consideran iguales a estos efectos.

Supongamos que ya están formados los grupos, y consideremos uno de ellos. Entonces hay que evaluar la lista SELECT para todas sus filas, de una en una. En la lista SELECT puede especificarse dos tipos de expresiones:

1. **Expresiones que dan el mismo resultado para todas las filas de cada uno de los grupos.** Son las expresiones de agrupamiento[2].
2. **Expresiones con funciones colectivas.** A partir de las filas del grupo producen un valor único.

En resumen, cada expresión de la lista SELECT produce un sólo valor para cada grupo. Con el resultado de cada expresión se construye una fila, es decir, para cada grupo se genera una ÚNICA fila.

[2]Esto es en Oracle, otros SGBD permiten expresiones distintas a las de agrupamiento, siempre que se observe la condición de que den el mismo valor para todas las tuplas de cada grupo.

Ejemplo 5.34 Hallar para cada departamento el salario medio de sus empleados.

```
select DEPTNO, AVG(SAL) AS MEDIA
 from emp
 GROUP BY DEPTNO;

    DEPTNO      MEDIA
---------- ----------
        10 2916,66667
        20       2175
        30 1566,66667

3 filas seleccionadas.
```

Obsérvese que en la lista SELECT hay una expresión de agrupamiento y otra con función colectiva. La expresión normal `DEPTNO` devuelve el mismo valor para todas las tuplas de cada grupo, de hecho es la expresión de agrupamiento. Oracle lo exige así para todas las expresiones que no sean funciones colectivas. Esto asegura que se pueda extraer una fila por grupo, al haber un único valor (para todas las expresiones de la lista SELECT) por grupo.

El resultado en el ejemplo anterior[3] se obtiene con los pasos siguientes:

1. Procesar el `FROM`. Seleccionar la tabla EMP.
2. Procesar el `GROUP BY`. Construir grupos con las filas que tengan igual valor en el atributo `DEPTNO`.
3. Procesar el `SELECT`. Evaluar las expresiones de la lista SELECT para las filas de cada grupo.
   - `DEPTNO`: todas las filas de cada grupo dan el mismo valor para esta expresión. Este valor es el que se incluye en la fila del resultado correspondiente a cada grupo.
   - `AVG(SAL)`: para cada grupo se calcula la media de los salarios de ese grupo. Esta media es la que forma parte de la fila de resultado de cada grupo.

   Todas las filas (cada una correspondiente a un grupo) forman la tabla resultado.

La siguiente consulta sería errónea:

```
select DEPTNO, SAL, AVG(SAL) AS MEDIA
 from emp
 GROUP BY DEPTNO;
```

debido a que el atributo `SAL` no es una expresión de agrupamiento, y por tanto, no puede aparecer en la lista SELECT por sí misma (en realidad, el problema es que no tiene un único valor para todas las filas de cada grupo).□

Ejemplo 5.35 Hallar el número de empleados de los departamentos 10 y 20.

[3] Obsérvese que no tiene WHERE.

```
  select DEPTNO, COUNT(*) AS EMPLE
  from emp
  where deptno IN (10,20)
  GROUP BY DEPTNO
  ORDER BY 1

    DEPTNO      EMPLE
---------- ----------
        10          3
        20          5

2 filas seleccionadas.
```

Obsérvese que el agrupamiento se hace con las filas que quedan después de aplicar el predicado de la cláusula `WHERE` y que el `ORDER BY` es la última cláusula en aplicarse. En resumen el orden de ejecución de la select es como sigue:

1. Procesar el `FROM`.
2. Procesar el `WHERE`.
3. Procesar el `GROUP BY`.
4. Procesar el `SELECT`.
5. Procesar el `ORDER BY`.

□

Ejemplo 5.36 Lista de los departamentos y la media de los salarios de las personas que trabajan en ellos y que ganan más de 2.000 €.

```
select DEPTNO, AVG(SAL)
  from emp
  where sal>2000
  group by DEPTNO;

    DEPTNO   AVG(SAL)
---------- ----------
        10       3725
        20 2991,66667
        30       2850

3 filas seleccionadas.
```

□

Ejemplo 5.37 Para cada departamento, obtener los diferentes puestos de trabajo de los que hay empleados y cuántos empleados hay con ese puesto de trabajo.

```
 select deptno, job, count(*)
  from emp
  group by deptno, job;

    DEPTNO JOB         COUNT(*)
---------- --------- ----------
        10 CLERK              1
        10 MANAGER            1
        10 PRESIDENT          1
        20 CLERK              2
        20 ANALYST            2
        20 MANAGER            1
        30 CLERK              1
        30 MANAGER            1
        30 SALESMAN           4

9 filas seleccionadas.
```

□

5.7. Cláusula HAVING

Es una cláusula opcional de la sentencia `SELECT` que sirve para descartar grupos de filas. Se especifica escribiendo la palabra `HAVING` seguida de una *condición*. Indica que después de haber separado las filas en uno o varios grupos, se descarten aquellos que no satisfagan la *condición*.

La separación de las filas en grupos se hace como ya se ha indicado. Es decir, según lo especificado en la cláusula `GROUP BY` (que precede a la cláusula `HAVING`).

La *condición* es un predicado, simple o compuesto, en el que podrá incluirse las *expresiones de agrupamiento y otras expresiones con funciones colectivas*. El formato es como sigue:

```
 HAVING (<condición>)
```

Ejemplo 5.38 Los puestos de trabajo cuyo salario medio sea mayor que el salario medio de la empresa y el número de empleados que lo tienen.

```
SQL> select Job,count(*), AVG(SAL)
  2  from emp
  3  group by job
  4  having AVG(SAL) > (select AVG(SAL) from emp);


JOB         COUNT(*)   AVG(SAL)
--------- ---------- ----------
ANALYST            2       3000
MANAGER            3 2758,33333
PRESIDENT          1       5000

3 filas seleccionadas.
```

Como vemos en la condición del `HAVING` se pueden incluir expresiones con funciones colectivas y cláusulas subordinadas. Observar que la expresión `AVG(SAL)` se evalúa para cada grupo de filas, dando (en este ejemplo) el salario medio de cada puesto de trabajo. La select subordinada es independiente del grupo que se está evaluando en un momento dado, obteniendo (siempre) el salario medio de TODOS los empleados (o desde otro punto de vista, de todos los puestos de trabajo). Aquellos grupos de filas que hagan cierta la condición del `HAVING` darán lugar a la correspondiente fila en el resultado.□

Ejemplo 5.39 Departamentos con al menos dos trabajadores que ganan más del salario medio de la empresa.

```
 select deptno
  from emp
  where sal > (select avg(sal) from emp)
  group by deptno
  having count(*)>=2;

    DEPTNO
----------
        10
        20

2 filas seleccionadas.
```

□

5.8. Orden de ejecución

Así una vez incluida la última cláusula del select básico, vamos a repasar el orden de ejecución de la sentencia `SELECT`.

1. Procesar el `FROM`. Es decir, seleccionar la tabla especificada.

2. Procesar el `WHERE`. Esto requiere eliminar de la tabla resultante las filas que no hagan verdadera la condición del `WHERE`.

3. Procesar el `GROUP BY`. Formar los grupos con las filas de la tabla resultante del paso anterior que den iguales valores en las expresiones de agrupamiento.

4. Procesar la cláusula `HAVING`. Descartar del resultado los grupos que no hagan verdadera la condición del `HAVING`.

5. Procesar el `SELECT`. Esto implica evaluar sus expresiones para cada uno de los grupos/filas obtenidos/as del paso anterior.

6. Procesar el `ORDER BY`.

5.9. Consultas sobre varias tablas

Hasta ahora las sentencias select que hemos visto sólo extraían datos de una sola tabla. Incluso en el caso de usar sentencias subordinadas, la lista SELECT sólo se refería a la tabla del `FROM` principal.

Sin embargo es posible manejar datos de varias tablas en una sentencia `SELECT`, para lo cual el SQL proporciona varias posibilidades:

1. Combinar las filas de una tabla con las de otra. Esta operación como vimos se denomina Join.

2. Usar sentencias `SELECT` subordinadas.

5.9.1. Calificación de los nombres de columnas

Como dentro de una tabla los nombres de columnas son todos diferentes, cuando una sentencia SQL opera sobre una tabla basta el nombre de una columna para designarla sin ambigüedad. Por el contrario, si interviene más de una tabla en la misma sentencia SQL puede ocurrir que algún nombre de columna se repita en más de una de ellas. En este caso, para referirnos sin ambigüedad a una columna determinada hay que indicar a qué tabla pertenece.

Para ello se escribe el nombre de la columna precedido del de su tabla y separados ambos por un punto.

Así por ejemplo, el nombre `EMP.DEPTNO` se refiere al tributo `DEPTNO` de la tabla de empleados.

También se pueden utilizar alias de los nombres de las tablas, si en la cláusula `FROM` se escribe el nombre de la tabla y luego (después de un espacio, o bien, después de la palabra clave `AS`) un alias, nos podremos referir al atributo por el alias punto nombre del atributo.

Ejemplo 5.40 En esta sentencia renombramos el nombre de tabla EMP por A.

```
select A.deptno
from emp A
where sal > (select avg(sal) from emp)
group by A.deptno
having count(*)>=2
```

□

5.9.2. Cláusula FROM con varias tablas

Como vimos en la Sección 5.1, la cláusula `FROM` puede tener más de una tabla (separada por comas), como por ejemplo `FROM EMP, DEPT`.

Esto significa que el SGBD construirá internamente una nueva tabla, que será el producto cartesiano de EMP y DEPT.

Si en lugar de especificar dos tablas, se especifican más, el proceso es análogo. Como el producto cartesiano puede dar lugar a una tabla extremadamente grande, el SGBD puede que no genere físicamente el producto cartesiano, pero a efectos de comprensión, es como si así fuera.

5.9.3. JOIN

En muy pocos casos va ser útil la operación de producto cartesiano. La inclusión de más de una tabla en la cláusula `FROM` suele ir acompañada de la inclusión de uno o más predicados en la condición del `WHERE` que especifican la condición de JOIN.

Ejemplo 5.41 Obtener los nombres de los empleados junto con el nombre del departamento para el cual trabajan.

```
select ENAME, DNAME
 from EMP A, DEPT B
 where A.DEPTNO=B.DEPTNO; /*condición de Join*/
```

```
ENAME      DNAME
---------- --------------
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER     ACCOUNTING
SMITH      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH
SCOTT      RESEARCH
JONES      RESEARCH
ALLEN      SALES
BLAKE      SALES
MARTIN     SALES
JAMES      SALES
TURNER     SALES
WARD       SALES

14 filas seleccionadas.
```

Dado que necesitamos incluir en la lista SELECT los atributos ENAME y DNAME, que están en dos tablas distintas, dichas tablas deben aparecer en la cláusula `FROM`. Pero si no incluimos la condición de JOIN, el resultado no tendría ningún sentido porque sería una proyección de esos dos atributos del producto cartesiano.

Para que el resultado sí tenga sentido, es necesario incluir en la condición del `WHERE` la condición del (en este caso) EQUIJOIN, `A.DEPTNO=B.DEPTNO`. En este caso como el atributo DEPTNO está presente en las dos tablas (en EMP es la clave externa) para deshacer ambigüedades, se califican los atributos con los nombres de los alias correspondientes.□

Ejemplo 5.42 Nombre de los empleados y el nombre del departamento de los empleados que trabajan en 'Chicago'.

```
SQL> select ENAME, DNAME
  2  from emp
  3  where deptno in (select DEPTNO
```

```
  4                   from DEPT
  5                   where LOC='CHICAGO');
select ENAME, DNAME
              *
ERROR en línea 1: ORA-00904: "DNAME": identificador no válido
```

Como vemos esta consulta da un error porque DNAME no es un atributo de EMP. Si sólo nos pidieran el nombre del empleado, esta podía ser una alternativa, pero dado que necesitamos el nombre del departamento nos vemos obligados a incluir la tabla DEPT en el `FROM`, y por lo tanto, incluir la correspondiente condición de Join en el `WHERE`.

```
    select ENAME, DNAME
    from emp a, dept b
    where a.deptno=b.deptno /*condición de Join*/
      and LOC='CHICAGO'

ENAME      DNAME
---------- --------------
ALLEN      SALES
BLAKE      SALES
MARTIN     SALES
JAMES      SALES
TURNER     SALES
WARD       SALES

6 filas seleccionadas.
```

□

Ejemplo 5.43 El nombre del departamento y el salario medio de sus empleados, de aquellos departamentos que tengan una media de salarios mayor que el salario medio de la empresa.

Una vez más nos encontramos con datos que deben aparecer en la lista SELECT procedentes de dos tablas. El nombre del departamento está en la tabla de departamentos, y el salario de los empleados en la tabla de empleados. Además en este ejemplo, debemos realizar una agrupamiento para poder calcular el salario medio y debemos descartar del resultado alguno de estos grupos.

```
  Select DNAME, AVG(SAL)
  from DEPT A, EMP B
  where a.deptno=b.deptno
  group by dname, a.deptno
  having AVG(SAL)>(select avg(sal) from emp)

DNAME            AVG(SAL)
-------------- ----------
ACCOUNTING     2916,66667
RESEARCH             2175

2 filas seleccionadas.
```

□

Para especificar un JOIN EXTERIOR, Oracle no sigue el estándar, primero veremos el modo en el que se puede especificar en Oracle y luego veremos el estándar.

Para especificar un JOIN EXTERIOR en Oracle, se añade `(+)` en el lado de la *condición de Join* incluida en la cláusula `WHERE` del siguiente modo:

- A la derecha del atributo a la izquierda del = si se quiere un JOIN EXTERIOR derecho.
- A la derecha del atributo a la derecha = si se quiere un JOIN EXTERIOR izquierdo.
- A la derecha de los dos atributos si se quiere un JOIN EXTERIOR completo.

Ejemplo 5.44 El nombre de los departamentos y el salario medio de los mismos. Si algún departamento no tiene empleados, debe aparecer en el resultado, y el campo "media de salarios" correspondiente debe tener valor *nulo*.

Si observamos las tablas del Apéndice **??**, existe el departamento 'Operations' que no tiene ningún empleado. Si hacemos un JOIN normal con la tabla de empleados (algo necesario para poder calcular el salario medio de cada departamento) no aparecerá ninguna fila correspondiente a ese departamento ya que no hay ningún empleado con el valor del atributo DEPTNO=40 (es decir, no hay un par de filas de las tablas EMP y DEPT que hagan cierto el predicado `EMP.DEPTNO=DEPT.DEPTNO`).

Obsérvese:

```
  Select A.DNAME, AVG(SAL)
  from DEPT A, EMP B
  where A.DEPTNO=B.DEPTNO
  group by A.DNAME, A.DEPTNO

DNAME            AVG(SAL)
-------------- ----------
ACCOUNTING     2916,66667
RESEARCH             2175
SALES          1566,66667

3 filas seleccionadas.
```

No produce ningún resultado para el departamento 'Operations', por lo antes comentando. Sin embargo si realizamos un JOIN EXTERIOR izquierdo para que aparezcan todas las tuplas del lado izquierdo a pesar de no "encajar" con ninguna fila del lado derecho obtenemos:

```
  Select A.DNAME, AVG(SAL)
  from DEPT A, EMP B
  where A.DEPTNO=B.DEPTNO (+)
  group by A.DNAME, A.DEPTNO

DNAME            AVG(SAL)
-------------- ----------
```

```
ACCOUNTING     2916,66667
RESEARCH             2175
SALES          1566,66667
OPERATIONS     nulo

4 filas seleccionadas.
```

□

En SQL2 se incluyó una modificación de la cláusula `FROM`[4] que permite especificar completamente el JOIN en dicha cláusula.

```
<tabla1> {INNER|{LEFT|RIGHT|FULL} [OUTER]} JOIN <tabla2> ON <condición>
```

`<tabla1>` y `<tabla2>` pueden ser dos tablas de la base de datos o pueden ser, a su vez, una expresión de Join como la anterior. Por ejemplo:

```
T1 LEFT JOIN T2 ON cond1 RIGHT JOIN T3 LEFT JOIN T4 ON cond2 ON cond3
```

equivalente a:

```
(T1 LEFT JOIN T2 ON cond1) RIGHT JOIN (T3 LEFT JOIN T4 ON cond2) ON cond3
```

La condición es un predicado simple o compuesto con algunas condiciones. Por ejemplo, no puede contener sentencias subordinadas ni funciones, y las columnas que se usen han de aparecen en `<tabla1>` y `<tabla2>`.

Ejemplo 5.45 Los nombres de los empleados junto con el nombre del departamento para el cual trabajan

```
  Select B.ENAME, A.DNAME
  from DEPT A INNER JOIN EMP B ON  A.DEPTNO=B.DEPTNO

ENAME      DNAME
---------- --------------
CLARK      ACCOUNTING
KING       ACCOUNTING
MILLER     ACCOUNTING
SMITH      RESEARCH
ADAMS      RESEARCH
FORD       RESEARCH
SCOTT      RESEARCH
JONES      RESEARCH
ALLEN      SALES
BLAKE      SALES
MARTIN     SALES
JAMES      SALES
```

[4]Que está incluida en Oracle.

```
TURNER     SALES
WARD       SALES

14 filas seleccionadas.
```

□

Ejemplo 5.46 El nombre de los departamentos y el salario medio de los mismos. Si algún departamento no tiene empleados, debe aparecer en el resultado, y el campo "media de salarios" correspondiente debe tener valor *nulo*.

```
  Select  A.DNAME, AVG(SAL)
  from DEPT A LEFT JOIN EMP B ON  A.DEPTNO=B.DEPTNO
  GROUP BY A.DNAME

DNAME            AVG(SAL)
-------------- ----------
ACCOUNTING     2916,66667
OPERATIONS     nulo
RESEARCH             2175
SALES          1566,66667

4 filas seleccionadas.
```

□

Ejemplo 5.47 Para todos los empleados de la empresa, obtener su nombre, salario, nombre de su jefe y salario del jefe.

```
 select e.ename, e.sal, j.ename, j.sal
  from emp e INNER JOIN emp j ON e.mgr=j.empno;

ENAME              SAL ENAME              SAL
---------- ---------- ---------- ----------
SCOTT            3000 JONES            2975
FORD             3000 JONES            2975
ALLEN            1600 BLAKE            2850
WARD             1250 BLAKE            2850
JAMES             950 BLAKE            2850
TURNER           1500 BLAKE            2850
MARTIN           1250 BLAKE            2850
MILLER           1300 CLARK            2450
ADAMS            1100 SCOTT            3000
JONES            2975 KING             5000
CLARK            2450 KING             5000
BLAKE            2850 KING             5000
SMITH             800 FORD             3000

13 filas seleccionadas.
```

Del modo que hemos escrito la consulta, en la primera columna falta un empleado, KING el presidente de la empresa. Pero como en el enunciado pide que aparezcan todos lo empleados de la empresa, tenemos que hacer un Join Exterior izquierdo para que aparezca también KING, rellenando las columnas tercera y cuarta con nulos.

```
  select e.ename, e.sal, j.ename, j.sal
 from emp e LEFT JOIN emp j ON e.mgr=j.empno

ENAME             SAL ENAME             SAL
---------- ---------- ---------- ----------
FORD             3000 JONES            2975
SCOTT            3000 JONES            2975
JAMES             950 BLAKE            2850
TURNER           1500 BLAKE            2850
MARTIN           1250 BLAKE            2850
WARD             1250 BLAKE            2850
ALLEN            1600 BLAKE            2850
MILLER           1300 CLARK            2450
ADAMS            1100 SCOTT            3000
CLARK            2450 KING             5000
BLAKE            2850 KING             5000
JONES            2975 KING             5000
SMITH             800 FORD             3000
KING             5000

14 filas seleccionadas.
```

□

5.10. Consultas correlacionadas

Hasta ahora hemos visto numerosos ejemplos de la inserción de consulta subordinadas en nuestras consultas. Pero en estos ejemplos, las sentencias subordinadas no hacen referencia a columnas de tablas que no estén en su propia cláusula `FROM`. Esto significa que el resultado de la sentencia subordinada puede evaluarse independientemente de sus sentencias antecedentes en todos los niveles, inclusive de la sentencia principal o de nivel 1. Por tanto, el SGBD la evalúa una sola vez y la reemplaza por su tabla de resultado en donde se encuentre. Esta manera de formular una sentencia subordinada se llama *no correlacionada*.

Sin embargo, en una sentencia subordinada puede haber referencias a columnas de otras tablas que no estén en su propia cláusula `FROM`. Estas sentencias se llaman *correlacionadas*, o también *consultas correlacionadas*.

Una sentencia subordinada correlacionada no puede evaluarse independientemente de las antecedentes, pues su resultado puede cambiar según qué filas se consideren en la evaluación de éstas en cada momento. El SGBD, por consiguiente, la evaluará múltiples veces. Hay que tener cuidado de que el número de veces sea lo menor posible, para evitar procesos inútiles al SGBD.

Cuando en una sentencia subordinada se especifica un nombre de columna, el SGBD debe saber sin ambigüedad en qué tabla se encuentra esa columna a la que nos referimos. Para

ello el sistema busca la primera tabla que contenga una columna con el nombre especificado, siguiendo el siguiente orden:

1. *Las tablas en la propia cláusula* `FROM` *de la sentencia correlacionada.* Si el nombre se encuentra en más de una de esas tablas, la referencia a la columna es ambigua y errónea. Para evitar este error deben usarse nombres de columna cualificados.

   Si se encuentra la tabla, la búsqueda acaba.

2. *Las tablas de la cláusula* `FROM` *de la primera sentencia antecedente.* Si el nombre se encuentra en más de una de esas tablas una vez más es necesaria la cualificación.

   Si se encuentra la tabla, la búsqueda acaba.

3. Se sigue el mismo proceso recorriendo consecutivamente las sentencias antecedentes de todos los niveles, hasta llegar a la sentencia externa si fuera necesario.

Si se desea alterar este orden de búsqueda, se deben cualificar los nombres de los atributos de modo que se indique al SGBD la tabla correspondiente a un atributo.

Ejemplo 5.48 Comenzamos con un ejemplo de consulta NO CORRELACIONADA. Obtener los nombres de los empleados que trabajan en 'Dallas'.

```
  Select ename
  from emp
  where deptno in (select deptno
                   from  dept
                   where loc='DALLAS')

ENAME
----------
SMITH
ADAMS
FORD
SCOTT
JONES

5 filas seleccionadas.
```

Como se puede apreciar el resultado de la consulta subordinada es siempre el mismo independientemente de la fila que se evalúe de la consulta principal (es siempre, 20). Por eso el SGBD la calcula una vez y el resultado (20) se utiliza para evaluar la condición `WHERE` de la consulta principal con todas las filas.□

Ejemplo 5.49 Ejemplo de sentencia correlacionada. Nombre y salario de los empleados que ganan más del salario medio de su departamento.

```
 select ename, sal
  from emp
  where sal > (select avg(sal)
               from   emp);
```

Esta consulta no responde a nuestro enunciado. Obsérvese que la consulta subordinada calcula el salario medio de todos los empleados de la empresa. De algún modo, debemos modificar esta consulta subordinada para calcular el salario medio del *empleado que en un momento dado se está evaluando en la consulta principal*.

Evidentemente el modo de hacer que la consulta subordinada calcule el salario medio de los empleados de un departamento concreto es incluyendo una condición en el `WHERE`. Pero, no podemos, a priori, especificar una condición del estilo `DEPTNO=XX`, donde XX es un número de departamento, puesto que este número depende de la fila que se está evaluando en la consulta principal.

La solución es la correlación, XX se sustituirá por el atributo DEPTNO de la tabla EMP que se está evaluando en la consulta principal, de tal modo que en cada momento este atributo tenga el número de departamento del empleado que se está evaluando en la consulta principal.

Para conseguir esto se utilizan atributos cualificados.

```
select ename, sal
from emp A
where sal > (select avg(sal)
             from   emp B
             where B.deptno=A.deptno)

ENAME             SAL
---------- ----------
ALLEN            1600
JONES            2975
BLAKE            2850
SCOTT            3000
KING             5000
FORD             3000

6 filas seleccionadas.
```

Ahora la consulta subordinada se recalcula (con el consiguiente costo computacional) por cada fila evaluada de la consulta principal en función del valor del atributo A.deptno (atributo de la consulta principal).□

Ejemplo 5.50 Nombre de departamento y cuantos empleados tiene dicho departamento de cada puesto de trabajo que ganen más que la media de trabajadores que trabajan en el mismo puesto.

```
Select DNAME, JOB, COUNT(*)
from dept D INNER JOIN emp E ON D.deptno=E.deptno
where E.sal > (select AVG(A.sal)
               from EMP A
               where A.job=E.job)
group by DNAME, JOB

DNAME          JOB         COUNT(*)
```

```
-------------- --------- ----------
SALES          MANAGER            1
SALES          SALESMAN           2
RESEARCH       CLERK              1
RESEARCH       MANAGER            1
ACCOUNTING     CLERK              1

5 filas seleccionadas.
```

□

5.11. Composición de consultas

El resultado de una consulta, es decir, una sentencia `SELECT`, es un conjunto de filas, con posibles repeticiones. Se pueden combinar varios de estos resultados unos con otros mediante operaciones de conjuntos, presentando una sola tabla como resultado final.

Una *subselect* es una sentencia `select` básica (la presentada en la Sección 5.1 sin la posibilidad de incluir el `ORDER BY`).

Así, las consultas compuestas son aquéllas cuyo resultado se obtiene realizando operaciones de unión, intersección y diferencia con dos conjuntos de filas obtenidas cada uno de ellos por medio de una *subselect*.

Las dos subselects que actúan como operandos de las operaciones de conjuntos deben tener el mismo número de columnas, y además las columnas que están en la misma posición relativa deben ser homogéneas, es decir, deben tener tipos de datos compatibles. El resultado es otra tabla con el mismo número de columnas y tipos de datos.

Los operandos pueden tener filas repetidas, es decir, filas con valores iguales en las columnas de la misma posición relativa. A estos efectos, dos nulos se consideran iguales.

Los operadores `UNION, INTERSECT` y `EXCEPT` (aunque en Oracle es `MINUS`), eliminan del resultado las filas repetidas, si las hubiere. Si deseamos que no se eliminen, se usan estos operadores añadiéndoles la palabra `ALL`.

Así el formato de un `SELECT`-compuesto es:

```
{subselect|SELECT-comp}{UNION [ALL]|INTERSECT [ALL]|MINUS [ALL]}{subselect|SELECT-comp}
```

Teniendo en cuenta que si no se utiliza Oracle se debe sustituir el `MINUS` por `EXCEPT`.

Ejemplo 5.51 Obtener el nombre y los ingresos totales (salario más comisión) de todos los empleados.

```
Select ename, sal AS INGRESOS
from emp
where comm is null
UNION
Select ename, sal+comm
from emp
where comm is not null
Order by 1
```

```
ENAME        INGRESOS
---------- ----------
ADAMS            1100
ALLEN            1900
BLAKE            2850
CLARK            2450
FORD             3000
JAMES             950
JONES            2975
KING             5000
MARTIN           2650
MILLER           1300
SCOTT            3000
SMITH             800
TURNER           1500
WARD             1750

14 filas seleccionadas.
```

□

5.12. Expresiones de tabla anidada

Hasta ahora hemos visto que en la cláusula `FROM` se especificaba una lista de tablas. Puesto que el resultado de un `SELECT` es una tabla, ¿no podría participar también en una cláusula `FROM` como una tabla más? Esto ya vimos en el álgebra relacional que era posible y SQL también lo permite.

El formato de una *expresión de tabla anidada* es como sigue:

```
(<select-compuesto>) [AS] [<nombre-correlación>] [(<nombre_col1>[,<nombre_col2>...])]
```

En este formato, el *nombre de correlación* es un nombre que se asigna a la tabla de resultado del *select compuesto*, a cuyas columnas también se puede opcionalmente asignar nombres, como se muestra en el formato. Además el *select compuesto* también incluye obviamente subselects (selects sencillas sin operación de conjuntos).

Estas expresiones pueden especificarse en una lista de tablas de una cláusula `FROM`, como una tabla más. Por tanto, una expresión de tabla anidada es una sentencia dentro de otra, es decir, una sentencia subordinada, siendo su sentencia antecedente inmediata la `SELECT` en cuyo `FROM` se encuentra. Esta sentencia antecedente puede referirse a las columnas de la expresión de tabla, como lo haría con cualquier otra tabla de su cláusula `FROM`.

Ejemplo 5.52 Queremos saber los departamentos y el número de empleados que tienen aquéllos en los que la media de salarios de sus empleados sea inferior a 2.000 €.

```
select NUMDEP, NUMEMP
 from (select A.DEPTNO AS NUMDEP, AVG(SAL) AS AVGSAL, COUNT(*) AS NUMEMP
       from EMP A, DEPT B
       where A.DEPTNO=B.DEPTNO
       GROUP BY A.DEPTNO)
```

```
  where avgsal < 2000
  order by 1;


    NUMDEP     NUMEMP
---------- ----------
        30          6

1 fila seleccionada.
```

Obsérvese que los nombres de columna de la expresión de tabla anidada se utilizan en la SELECT externa. Estas referencias pueden figurar tanto en la lista SELECT como en la condición del WHERE.□

Ejemplo 5.53 Para los departamentos 10 y 20, hallar el valor medio de los salarios medios de cada departamento.

```
 Select AVG(PREMED) AS MEDIA
  FROM (select avg(sal) as PREMED
        from emp
        where DEPTNO IN (10,20)
        group by DEPTNO);

     MEDIA
----------
2545,83333

1 fila seleccionada.
```

□

6. Continuación del SQL como DML (Lenguaje de Manipulación de Datos)

6.1. Inserción de datos

Para insertar datos en una tabla, usaremos la sentencia INSERT, que tiene la siguiente sintaxis:

```
INSERT INTO <nombre_tabla> [(<lista campos>)] VALUES (<lista valores>)
```

La lista de campos (atributos de la tabla) es opcional. Si se omite, es lo mismo que si se incluyese la lista de todos los atributos de la tabla, siguiendo el orden definido en la sentencia de creación de la tabla. Si se incluye, pueden indicarse sólo una parte de los campos (o todos) en el mismo o distinto orden de la definida en la creación de la tabla. Así, las dos sentencias siguientes son equivalentes:

```
INSERT INTO EMP (EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, DEPTNO)
VALUES (8555, 'Jose', 'ANALISTA', 7369, '12-FEB-02', 3000, 300,10);

INSERT INTO EMP VALUES (8555, 'Jose', 'ANALISTA', 7369, '12-FEB-02', 3000, 300,10);
```

La lista de valores, por supuesto, debe contener tantos valores como atributos hay en la lista de campos, y corresponderse un al tipo del atributo correspondiente. Por supuesto, el valor NULL puede ser usado, si el campo correspondiente fue definido aceptando valores nulos.

Si se usa la lista de campos, podemos variar el orden de los campos, o omitir alguno de ellos. En los campos omitidos se insertarán nulos, o el valor por defecto (se así se definió para el campo). Por ejemplo, las siguientes tres sentencias también serían equivalentes.

```
INSERT INTO DEPT(DEPTNO,DNAME) VALUES(50,'COMPRAS');
INSERT INTO DEPT (DNAME,DEPTNO) VALUES ('COMPRAS',50);
INSERT INTO DEPT VALUES (50,'COMPRAS', NULL);
```

Además de esta forma de insertar datos, que se realiza siempre de tupla en tupla, también se pueden insertar varias tuplas con una sentencia, obteniendo los datos mediante una sentencia `SELECT` (una consulta) sobre una o más tablas. La sintaxis general es la siguiente:

```
INSERT INTO <nombre_tabla> [(<lista de campos>)]
    <Sentencia SELECT>
```

La única restricción es que la sentencia `select` debe seleccionar el mismo número de atributos que campos hay en la lista de campos, y ser unión-compatibles. Como en el método anterior, si en la lista de campos se omite, se asume que el select devuelve una lista con todos los atributos de la tabla en el mismo orden que el establecido en la creación de la tabla. Por ejemplo, supongamos que tenemos una tabla `COPIA_DEPT` que tiene exactamente la misma estructura que a tabla `DEPT`. La siguiente sentencia serviría para insertar varias tuplas en la tabla `DEPT` a partir de una consulta.

```
INSERT INTO DEPT (DEPTNO, DNAME, LOC)
    SELECT C.DEPTNO, DNAME, LOC
        FROM COPIA_DEPT C, EMP E
        WHERE C.DEPTNO=E.DEPTNO
```

6.2. Borrado de datos

El borrado de datos se realiza mediante la sentencia `DELETE`, que borra tuplas de una tabla. La sintaxis general es:

```
DELETE FROM <nombre_tabla>
    [WHERE <Condicion>]
```

Donde la `Condicion` es cualquier condición válida en SQL, es decir, cualquier condición que podríamos poner en la cláusula `WHERE` de una sentencia `SELECT`. Por ejemplo,

```
DELETE FROM EMP
    WHERE SAL > 1500
      AND JOB <> 'MANAGER'
```

Es importante indicar que la sentencia `DELETE` borra siempre tuplas completas, nunca determinados atributos de una tupla. La cláusula `WHERE` es opcional, y se no se utiliza, se borrarán todas las tuplas de la tabla.

6.3. Modificación de Datos

La sentencia `UPDATE` permite modificar los valores almacenados en una tabla, es decir, cambiar el valor que hay en un atributo de una tupla por otro valor. Su sintaxis es:

```
UPDATE <nombre_tabla>
    SET <atributo_1> = <expre1>
        [,<atributo_2>=<expre2>...]
    WHERE <Condicion>
```

Un ejemplo,

```
UPDATE EMP
    SET  SAL = SAL*1.06,
    JOB = 'ANALYST'
    WHERE EMPNO = 7369
```

La sentencia de actualización examina todas las tuplas que satisfagan a condición impuesta en la cláusula `WHERE` (si no se indica, los cambios afectarán a todas las tuplas de la tabla). Para cada una de estas filas, el atributo tomará el nuevo valor, que puede ser una constante (como en `JOB = 'ANALYST'`) o una expresión. Las expresiones pueden contener atributos de la tabla que se está actualizando, incluso el mismo nombre de atributo que está a la izquierda del igual. En este caso, el valor del atributo a la derecha del igual es el valor de dicho atributo antes de realizarse la actualización. Es decir, la asignación `SAL=SAL*1.06` indica que el salario actual se va incrementar en un 6 % respecto al salario anterior.

7. SQL como DDL (Lenguaje de Definición de Datos)

Hasta ahora hemos utilizado el lenguaje SQL como un **DML** (Data Manipulation Language) o Lenguaje de Manipulación de Datos, es decir, para añadir, borrar, modificar o consultar datos. Pero además el SQL puede utilizarse como **DDL** (Data Definición Language) o Lenguaje de Definición de Datos; esto es, podemos utilizar SQL para crear, modificar o borrar diferentes objetos de bases de datos como tablas, vistas o índices.

7.1. Creación de tablas

La sentencia SQL de creación de tablas es `CREATE TABLE`, y en ella se definen por un lado los atributos que componen la tabla, y por otro las restricciones que afectan a la tabla, tales como claves primarias o externas. La sintaxis general de la sentencia de creación se compone de una lista (separada por comas) de definiciones de atributos seguida opcionalmente de una lista de definiciones de restricciones:

```
CREATE TABLE <nombre_tabla>(
    <definicion atributo 1>,
    <definicion atributo 2>,
    ...
    <definicion atributo n>,
    <definicion restriccion 1>,
    <definicion restriccion 2>,
```

```
...
<definicion restriccion m>
);
```

A continuación veremos cómo realizar cada una de estas definiciones.

7.1.1. Definición de atributos

Una definición de atributo describe el nombre y tipo de datos asociados al atributo, así como si admite o no valores nulos, valores por defecto y restricciones asociadas al atributo. La sintaxis general es la siguiente:

```
<nombre_atributo> <tipo> [DEFAULT <valor>] [NOT [NULL]] [<restriccion>]
```

Donde

`<nombre_atributo>` es el nombre del atributo.

`<tipo>` es el tipo del atributo, que será uno de los tipos de datos vistos anteriormente.

`DEFAULT <valor>` indica el valor por defecto que almacenará el campo si se inserta una tupla sin indicar un valor para ese atributo (si no se indica un valor por defecto, se insertará un valor nulo). `<valor>` será, evidentemente, una constante del tipo definido para el atributo. Existen algunos valores predefinidos para algunos tipos de datos. Así, tenemos valores como `SYSDATE`, que nos da la fecha actual, o `USER` que es un string que nos da el nombre del usuario (en el caso de Oracle). También se puede usar el valor `NULL` para cualquier tipo de datos, siempre que el atributo lo admita. Sin embargo, hacer esto no tiene demasiado sentido, ya que es el comportamiento por defecto si no se indica un valor `DEFAULT`.

`[NOT] NULL` indica si el atributo acepta valores nulos (`NULL`) o no los acepta (`NOT NULL`). Si no se indica nada se supone que sí los acepta.

`<restriccion>` es una restricción sobre el atributo. Veremos principalmente las restricciones que indican que un atributo es una clave primaria o una clave externa que referencia otra tabla. Si el atributo es una clave primaria, se indicará con `[CONSTRAINT <nombre_constr>] PRIMARY KEY`, y si es una clave externa, mediante `[CONSTRAINT <nombre_constr>] REFERENCES <tabla>[(<atributos>)]`. Veremos más adelante cómo indicar las restricciones de forma independiente de la definición de los atributos.

Otros tipos de restricción usados son `CHECK`, para realizar comprobaciones en los valores que se introducen en la tabla, o `UNIQUE`, que establece que un atributo (o lista) no puede tener duplicados en la tabla.

Ejemplo 7.1 Una creación de la tabla de departamentos podría ser:

```
CREATE TABLE DEPT(
DEPTNO NUMBER(4) CONSTRAINT rest1 PRIMARY KEY,
DNAME VARCHAR2(14),
LOC VARCHAR2(13));
```

7.1.2. Definición de restricciones

Una definición de restricción tiene el formato general siguiente:

```
[CONSTRAINT <nombre_constraint>] <tipo> <definicion>
```

donde la palabra `CONSTRAINT` y el nombre son opcionales, `<tipo>` indica el tipo de restricción. El nombre de la restricción, en caso de usarse, debe ser único para toda la base de datos (aunque esté en distinta tabla no puede tener el mismo nombre). Finalmente, `<definición>` describe completamente la restricción. Dado que esto último depende de cada restricción, veremos los distintos tipos por separado.

Definición de claves primarias Una clave primaria puede estar compuesta de un atributo o de varios, como ya hemos visto. En cualquier caso, todos los atributos que formen parte de la clave primaria, ya que van a identificar unívocamente una tupla, deben estar definidos como `NOT NULL` (si no se indica explícitamente, el SGBD lo hace de forma automática).

Si la restricción de clave primaria afecta a un solo atributo (como `DEPTNO` en la tabla `DEPT`), puede indicarse directamente en la definición del atributo, pero también como una restricción independiente, utilizando la sintaxis siguiente.

```
[CONSTRAINT <nombre>] PRIMARY KEY (<lista de atributos separados por comas>)
```

como por ejemplo, `CONSTRAINT pk_dept PRIMARY KEY(DEPTNO)`. Por convención, se suele preceder el nombre de una restricción de clave primaria por el prefijo `pk_` (por Primary Key).

Definición de claves externa Una clave externa establece una relación de integridad referencial entre dos tablas.

Es obligatorio que la tabla referenciada tenga una clave primaria (o, en su defecto, que exista una restricción única), y será precisamente esa clave primaria (o la lista de atributos con restricción única) la que se referencia en la clave externa. De la misma forma, nótese que para poder crear una tabla que referencia a otra, la tabla referenciada debe existir con anterioridad, es decir, debe ser creada antes.

Al igual que para las clave primarias, para establecer una clave externa tenemos dos opciones: incluirla en la definición del atributo, como ya se ha indicado:

```
<atributo> <tipo> REFERENCES <tabla_referenciada>[(<columnas_referenciadas>)]
```

Por ejemplo,

```
DEPTNO NUMBER(4) REFERENCES DEPT
```

Una segunda opción, la única utilizable cuando la clave externa está formada por varios atributos (pero válida también si es uno solo) es definir la clave externa como una restricción aparte:

```
[CONSTRAINT <nombre_constraint>] FOREIGN KEY (<lista atributos separados por comas>)
            REFERENCES <tabla_referenciada> [(<columnas_referenciadas>)]
            [ON DELETE CASCADE]
```

Por convención, normalmente se usa el prefijo `fk_` (Foreign Key) para nombrar las restricciones de clave externa. La cláusula opcional `ON DELETE CASCADE` indica cómo reacciona la tabla para mantener la integridad en caso de borrar una tupla de la tabla referenciada. Si se indica `ON DELETE CASCADE`, cuando se borra la tupla "padre" (la referenciada), se borran también los "hijos" (es decir, si se borra un departamento, se borrarán automáticamente los empleados que trabajan para él, para mantener la integridad referencial). Si no se usa esta cláusula, el gestor no dejará borrar la tupla "padre".

```
CREATE TABLE EMP (
EMPNO NUMBER(4)  PRIMARY KEY,
ENAME VARCHAR2(10),
JOB VARCHAR2(9),
MGR NUMBER(4) REFERENCES EMP,
HIREDATE DATE,
SAL DECIMAL(7,2),
COMM DECIMAL(7,2),
DEPTNO NUMBER(4),
PRIMARY KEY (EMPNO),
CONSTRAINT fk_deptno FOREIGN KEY (DEPTNO)
           REFERENCES DEPT(DEPTNO)
           ON DELETE CASCADE)
```

El poner nombre a las restricciones es importante por si más adelante se desean modificarlas o eliminarlas.

Por otro lado, aunque Oracle no lo soporta, en SQL2 se permitían otras opciones a la hora de borrar o actualizar una tupla "padre":

```
[ON {DELETE|UPDATE} SET {NULL|CASCADE}]:
```

En caso de borrado o actualización:

- SET NULL: se coloca el valor nulo en la clave externa.
- CASCADE: se borran las tuplas .

Restricciones de unicidad (`UNIQUE`) Una lista de atributos (o uno solo) puede establecerse como único, es decir, que no admita duplicados. Esto es útil, entre otras cosas, para garantizar que se cumplan las claves candidatas, ya que sólo es posible indicar una única clave primaria por tabla. La sintaxis es la siguiente:

```
<descripción del atributo> UNIQUE
```

o

```
[CONSTRAINT <nombre>] UNIQUE(<lista de atributos>)
```

Por ejemplo,

```
DNAME VARCHAR2(14) UNIQUE
```

o

```
CONSTRAINT u_dept_loc UNIQUE(DNAME, LOC)
```

Como se ha indicado anteriormente, una restricción de clave externa puede referenciar una lista de atributos que estén afectados por una restricción `UNIQUE`.

Restricciones de comprobación (`CHECK`) Se usa estas restricciones para realizar comprobaciones sobre los valores que se introducen en la tabla. Dependiendo de si se indica la restricción en la declaración del atributo o de forma aislada, la sintaxis general es la siguiente:

```
<descripción del atributo> CHECK (<condicion>)
```

o

```
[CONSTRAINT <nombre>] CHECK (<condicion>)
```

El nombre de la restricción es similar a las anteriores. La condición será simple (del tipo de las que vimos en las sentencias `SELECT`, aunque sin consultas subordinadas). Las restricciones tipo `CHECK` pueden afectar a más de un atributo. En el caso de que afecten a uno solo, se pueden indicar al final de la descripción del mismo. Veamos dos ejemplos.

```
... SAL NUMBER(7,2) CHECK (SAL > 0)
```

o

```
CONSTRAINT C_comm_job CHECK( (COMM < 2000) AND (JOB <> 'No lo sé'))
```

Como se ve, si la restricción se indica de forma independiente, como la segunda del ejemplo anterior, puede comprobar varias columnas. Si, como en el primer caso, se indica en la declaración de un atributo, sólo puede referenciarlo a él.

7.2. Borrado de Tablas

La sentencia SQL para borrar una tabla es `DROP TABLE <nome_tabla>`. Nótese que por "borrar" se entiende destruir totalmente el objeto tabla, no sólo los datos que contenga, sino también la definición de su estructura. Por ejemplo,

```
DROP TABLE EMP; DROP  TABLE DEPT;
```

borraría las tablas EMP y DEPT. Cuando se borra una tabla, también se borrarán otras estructuras asociadas a ella, tales como índices, restricciones, vistas o triggers (veremos más adelante algunos de estos conceptos).

Al igual que en la creación de tablas, existe una restricción en el proceso de borrado de tablas cuando hay claves externas: se debe borrar antes la tabla referenciadora que la referenciada. Tal como se ve en el ejemplo previo, se borra la tabla `EMP` (que tiene una clave externa que referencia a la tabla `DEPT`) antes de borrar la tabla `DEPT`.

7.3. Modifición (Alteración) de Tablas

La estructura de una tabla (creada con la sentencia `CREATE TABLE`) puede ser modificada, añadiendo y/o modificando o borrando atributos o restricciones. Para ello se utiliza la sentencia SQL `ALTER TABLE`. Veamos a continuación las opciones disponibles para esta sentencia.

7.3.1. Alteración de atributos

El estándar SQL2 permite añadir, borrar o modificar un atributo de una tabla. Sin embargo, Oracle, hasta la versión 8.0 inclusive, sólo permite añadirlos y modificarlos, pero no borrarlos. Para añadir un atributo (que se añadirá al final de los ya existentes) se utiliza la siguiente sintaxis:

```
ALTER TABLE <nombre_tabla>
      ADD <Definicion de atributo>;
```

Por ejemplo, si consideramos que la tabla de proveedores debe incluir la dirección del proveedor, podemos añadir ese atributo:

```
ALTER TABLE EMP
      ADD DIRECCION VARCHAR2(50);
```

El nuevo campo se incluye en la estructura de la tabla, evidentemente para todas sus tuplas. Las tuplas ya existentes incluirán un valor nulo en ese atributo, o el valor por defecto si se utiliza la cláusula `DEFAULT`. No sería posible, por lo tanto, añadir un atributo definido como `NOT NULL` y sin un valor por defecto, si la tabla ya contiene datos.

```
ALTER TABLE EMP                              -- Incorrecto!!
      ADD DIRECCION VARCHAR2(50) NOT NULL;

ALTER TABLE EMP                              -- Correcto
      ADD DIRECCION VARCHAR2(50) DEFAULT 'Desconocida' NOT NULL;
```

Para modificar un atributo existente (sólo se puede modificar su tipo, no el nombre), se utilizará

```
ALTER TABLE <nombre_tabla>
      MODIFY <Definicion de atributo existente>;
```

Por ejemplo, para indicar que necesitamos agrandar el atributo dirección hasta 120 caracteres, utilizaremos la sentencia siguiente:

```
ALTER TABLE EMP
      MODIFY DIRECCION VARCHAR2(120);
```

Cuando modificamos el tipo de una columna o atributo, debemos tener en cuenta los siguientes aspectos, que se explicarán tomando como base que la tabla EMP tiene el atributo dirección definido como `DIRECCION VARCHAR2(50) DEFAULT 'Desconocida' NOT NULL`:

- Si la columna tiene valores, no se puede reducir el tamaño de la columna, pero sí aumentarlo.

  ```
  ALTER TABLE EMP                            -- Incorrecto!!
        MODIFY DIRECCION VARCHAR2(10);

  ALTER TABLE EMP                            -- Correcto
        MODIFY DIRECCION VARCHAR2(150);
  ```

- Si la columna tiene valores nulos, no puede especificarse ahora que no se admiten nulos.
- Si se desea indicar que ahora se admiten valores nulos y/o no hay valores por defecto, debe indicarse explícitamente. Para ello, si queremos que ahora acepte nulos debemos indicarlo con `NULL`, y si queremos que no exista un valor por defecto, indicar `DEFAULT NULL` (es decir, el valor por defecto es `NULL`, que es el que toma si no se indica otro).

```
ALTER TABLE EMP     -- Sigue sin aceptar nulos y tiene el mismo
      MODIFY DIRECCION VARCHAR2(120);  -- valor por defecto

ALTER TABLE EMP     -- Correcto. Ahora admite nulos
      MODIFY DIRECCION VARCHAR2(120) NULL;

ALTER TABLE EMP     -- Correcto. Ahora no hay valor por defecto (es NULL)
      MODIFY DIRECCION VARCHAR2(120) DEFAULT NULL
```

Oracle, a partir de su versión 8*i* (en concreto, 8.1.6), también permite borrar campos de una tabla. Para ello se utiliza la sentencia

```
ALTER TABLE <tabla>
    DROP (<columna>) [CASCADE CONSTRAINTS]
```

Por ejemplo,

```
ALTER TABLE EMP
    DROP (DIRECCION) CASCADE CONSTRAINTS
```

La opción `CASCADE CONSTRAINTS` es necesaria, por ejemplo, si queremos borrar una columna que es referenciada por una clave externa:

```
SQL> ALTER TABLE EMP DROP (EMPNO);

ALTER TABLE EMP DROP (EMPNO)
                     *
ERROR en línea 1: ORA-12992: no se puede borrar la columna de
claves principales


SQL> ALTER TABLE EMP DROP (EMPNO) CASCADE CONSTRAINTS;

Tabla modificada.
```

En el segundo caso, en el que la sentencia funciona correctamente, la restricción es eliminada. Debemos hacer notar que, sin embargo, para otras restricciones como `UNIQUE` no es necesaria la cláusula `CASCADE CONSTRAINTS`, y la restricción asociada se borra igualmente.

Se permite añadir una restricción nueva a una tabla, utilizando la sintaxis siguiente:

```
ALTER TABLE <nombre_tabla>
      ADD <Definicion de restriccion>;
```

Por ejemplo, si hubiésemos creado la tabla EMP sin clave primaria y quisiésemos añadir esa restricción, utilizaríamos la sentencia

```
ALTER TABLE EMP
      ADD CONSTRAINT pk_empleado PRIMARY KEY(EMPNO);
```

Para borrar una restricción, usaremos:

```
ALTER TABLE <nombre_tabla>
      DROP CONSTRAINT <nombre de restriccion>;
```

Es evidente que para poder borrar una restricción debemos conocer su nombre, por lo que es recomendable darles un nombre cuando se crean. Si no se le ha dado nombre, es posible, examinando el catálogo (de Oracle), saber su nombre para poder borrarla, pero se recomienda en todo caso darle un nombre para facilitar estas operaciones. Así, para eliminar la restricción de clave primaria que acabamos de crear, usaríamos lo siguiente:

```
ALTER TABLE PROVEEDOR
      DROP CONSTRAINT pk_empleado;
```

7.4. Creación de índices con SQL

Utilizando SQL, la sentencia básica de creación de un índice es la siguiente:

```
CREATE INDEX <nombre índice> ON <tabla>(<lista de atributos>);
```

Por ejemplo, si planeamos realizar muchas consultas sobre el nombre y el puesto de trabajo, el siguiente índice sería útil.

```
SQL> CREATE INDEX idx_nom_puesto ON EMP(ENAME,JOB);

Índice creado.
```

Este tipo de índices permite valores duplicados, puesto que no se ha indicado lo contrario. Si deseamos crear un índice que no los admita (por ejemplo, si el índice se crea sobre el campo que es la clave primaria de una tabla y que por lo tanto no admite duplicados), utilizaremos la sintaxis siguiente:

```
CREATE UNIQUE INDEX <nombre índice> ON <tabla>(<lista de atributos>);
```

Por ejemplo, podríamos indexar crear un índice sobre la clave primaria de EMP, usando .

```
CREATE UNIQUE INDEX idx_pk_EMP ON EMP(EMPNO);
```

Originalmente se utilizaban los índices únicos para controlar la integridad de entidad, es decir, para controlar que no existiesen claves primarias duplicadas. De hecho, los gestores siguen creando un índice de este tipo sobre la clave primaria. Por ello, veamos lo que ocurre si intentamos crear ese índice sobre la tabla EMP donde se definió como clave primaria el atributo EMPNO:

```
SQL> CREATE UNIQUE INDEX idx_pk_EMP ON EMP(EMPNO);

CREATE UNIQUE INDEX idx_pk_dni ON EMP(EMPNO)
                                      *
ERROR en línea 1: ORA-01408: esta lista de columnas ya está indexada
```

Como se ve, esto produce un error. Esto es debido a que existe una restricción (muy lógica), que es que una `<lista de atributos>` sólo puede estar indexada por un único índice, y la creación de una clave primaria implica que el SGBD crea automáticamente un índice único sobre dicha clave. En otro caso estaríamos creando varios índices exactamente iguales, con lo que lo único que conseguiríamos sería desperdiciar espacio.

En el caso de Oracle, hay más tipos de índices soportados, que se crean de forma similar:

```
CREATE <tipo_indice> INDEX <nombre> ON....
```

Entre los tipos soportados se encuentran el tipo `BITMAP` (que no usa árboles B^+ sino un bitmap para almacenar el índice), o `UNSORTED` (que no ordena los datos en base a los campos indexados como hace si no se indica nada).

7.5. Vistas

La definición más simple de una vista es la que nos dice que *una vista es una consulta a la que se da un nombre*. Una vez que la vista se ha creado la definición de la vista, se puede acceder a ella exactamente igual que si fuese una tabla para ver su estructura o para seleccionar los datos que contiene. El mayor problema asociado a las vistas es determinar cuándo los datos de una vista se pueden actualizar (insertar, borrar o modificar filas).

Las tablas indicadas en la consulta que define una vista se denominan *tablas base*, y pueden a su vez ser vistas.

7.5.1. Creación de Vistas

La sentencia utilizada para crear una vista es `CREATE VIEW`, cuya sintaxis general se muestra a continuación.

```
CREATE [OR REPLACE] VIEW <nombre_vista> [( <esquema_vista> )]
   AS <sentencia select>
```

Si indicamos `CREATE VIEW...` y ya existe una vista con ese nombre, el SGBD dará un error. Si indicamos `CREATE OR  REPLACE VIEW...` y la vista existe, se sustituye la declaración de la vista existente por la nueva.

La `<sentencia select>` es una consulta realizada utilizando la sentencia `SELECT`.

El nombre `<nombre_vista>` especifica el nombre que se da a la vista que se crea. Así, por ejemplo, la siguiente sentencia define una vista sobre la tabla `EMP`, extrayendo solamente el EMPNO y nombre.

```
SQL> CREATE VIEW EMPLEADOS
       AS SELECT EMPNO, ENAME
            FROM EMP;

Vista creada.
```

Una vez definida, la vista se puede consultar como si fuese una tabla. Los nombres de los atributos de la vista serán los mismos atributos que se seleccionan de la(s) tabla(s). Así, veamos la vista anterior:

```
SQL> describe EMPLEADOS
 Nombre                                    +Nulo?   Tipo
 ----------------------------------------- -------- ------------
 EMPNO                                     NOT NULL NUMBER(4)
 ENAME                                              VARCHAR2(10)
```

Para renombrar los atributos (o expresiones de la select) se pueden tomar dos alternativas:

1. Especificar un alias para las expresiones que no corresponden a nombres de atributos:

```
SQL> CREATE VIEW V1
       AS SELECT ENAME AS NOMBRE, SAL+COMM AS SALARIO_TOTAL
          FROM EMP;

Vista creada.
```

   En este caso, la vista tendrá un atributo `NOMBRE` y otro `SALARIO_TOTAL`.

2. Utilizar la parte opcional `<esquema_vista>`, indicando (en una lista entre paréntesis) los nombres que se darán a los atributos de la vista que representan las expresiones seleccionadas:

```
SQL> CREATE OR REPLACE VIEW V1 (NOMBRE, SALARIO_TOTAL)
       AS SELECT ENAME, SAL+COMM
          FROM EMP;

Vista creada.
```

   El esquema de la vista sería el mismo que en el caso anterior.

En cualquier caso, en la primera de las opciones el uso de alias sólo es necesario para aquellas expresiones de nombre inválido. Es decir, si alguna expresión es válida, no es necesario especificar un alias (aunque sí se puede hacer).

Las consultas pueden ser todo lo complejas que se quieran, incluyendo las cláusulas `WHERE`, `GROUP BY` o `HAVING`. Así, la siguiente consulta nos ofrece una relación en la que se ven los nombres de los departamento, el número de empleados de cada departamento, y el montante total de los salarios que pagan:

```
CREATE OR REPLACE VIEW UNA_VISTA (NOMBRE, NUM_EMP, SALARIO_TOTAL)
       AS SELECT DNAME, COUNT(*), SUM(SAL)
         FROM EMP E INNER JOIN DEPT D ON E.DEPTNO=D.DEPTNO
         GROUP BY DNAME
```

Una consulta sobre esta vista nos daría el resultado siguiente:

```
SQL> SELECT * FROM UNA_VISTA;

NOMBRE            NUM_EMP SALARIO_TOTAL
-------------- ---------- -------------
ACCOUNTING              3          8750
RESEARCH                5         10875
SALES                   6          9400

3 filas seleccionadas.
```

7.5.2. Borrado de vistas

Para borrar una vista (sólo la definición, no los datos) se utiliza la sentencia siguiente:

```
DROP VIEW <nombre vista>
```

Por ejemplo,

```
SQL> DROP VIEW UNA_VISTA;

Vista borrada.
```

7.5.3. Ventajas del uso de vistas

Existen múltiples ventajas derivadas del uso de vistas en la definición de un esquema de bases de datos. Entre ellas destacan las siguientes, de las que se da algún ejemplo:

- Ocultación de información: Un empleado puede gestionar datos de los empleados, pero quizás no debería ver cierta información, como por ejemplo el salario. Para esto se puede definir una vista que obtenga los datos de los empleados, pero no el atributo salario ni comisión. Así, la persona que gestiona los empleados no accede a información a la que no debe.

  ```
  SQL> CREATE OR REPLACE VIEW EMPLEADOS
    2     AS SELECT EMPNO, ENAME, JOB, HIREDATE,
    3               MGR, DEPTNO
    4     FROM EMP;

  Vista creada.
  ```

- Independencia con respecto al diseño de tablas: Si se cambia la estructura de alguna tabla, como por ejemplo de algún atributo de la misma (que non está incluido en la definición de la vista), eso no afecta a los programas que acceden a la vista. Por ejemplo, si añadimos un atributo "fecha de nacimiento" para los empleados, los programas que acceden a la vista `EMPLEADOS` no se verán afectados.

- Facilitar la realización de consultas: Una vista puede definir una consulta relativamente compleja, y se pueden hacer a su vez consultas sobre esta vista. Así, si en algún caso un usuario debe realizar consultas complejas y no es experto en SQL, el uso de una vista que haga parte de la consulta de de gran ayuda. Por ejemplo, en una base de datos de contabilidad, existe una tabla en la que se almacenan los apuntes de los asientos contables. Podríamos definir una vista para calcular los saldos del debe y el haber de las cuentas, y un usuario normal tendría que hacer una consulta realmente simple para obtener el saldo de una determinada cuenta (del tipo `SELECT * FROM VISTA WHERE CUENTA=<cuenta>`) en vez de tener que definir la consulta completa, con sumas, agrupamientos, etc.

7.5.4. Consultas sobre vistas

Como ya se ha indicado, realizar una consulta sobre una vista es exactamente igual (sintácticamente) que realizar una consulta sobre una tabla real.

La ejecución (teórica) de una consulta que usa una vista seguiría 2 pasos:

1. Ejecutar la consulta que define la vista, obteniendo una relación temporal que contiene los datos de la vista.
2. Ejecutar la consulta pedida sobre esa relación temporal.

Lógicamente, la realización de una consulta sobre una visa se traslada a una consulta a las tablas base sobre las que se define la vista. Sin embargo, normalmente, los gestores de bases de datos implementan algoritmos de optimización sobre las consultas, por lo que las condiciones establecidas en la consulta se suelen añadir a las establecidas en la definición de la vista (automáticamente y de forma transparente para el usuario que tecleó la consulta) de modo que es esta última consulta la ejecutada.

7.5.5. Actualización de vistas

Al igual que cuando se hace una consulta, cuando se lanza una sentencia de actualización de datos (`INSERT`, `UPDATE`, `DELETE`) sobre una vista, el gestor de bases de datos tratará de trasladar esa actualización a la tabla o tablas base sobre las que está definida la vista. Al intentar una actualización, en algunos casos es posible realizarla, pero en otros no es posible y el gestor dará un error.

De forma genérica, podemos dividir las vistas en las siguientes categorías:

Todas las vistas: Todas las vistas que pueden ser definidas.

Vistas teóricamente actualizables: Es un subconjunto del anterior, formado por las vistas que en teoría se podrían actualizar. Es decir, la traducción de actualizar los datos en la vista a actualizar los datos en las tablas base es teóricamente posible.

Vistas realmente actualizables: Es un subconjunto de las vistas teóricamente actualizables, formado por aquellas vistas en realidad son consideradas actualizables por lo gestores de bases de datos. Probablemente cada gestor de bases de datos permitirá la actualización de datos en un subconjunto (reducido) de este conjunto de vistas. Veremos concretamente la reacción de Oracle ante el intento de actualización de varios tipos de vistas.

A continuación veremos varios tipos de consultas que se podrían usar para definir una vista, y qué es lo que pasaría, utilizando Oracle, si tratamos de actualizar datos en la vista. Como regla general, una vista es actualizable si las tuplas de las tablas base son "back-traceable", esto es, cuando es posible identificar la tupla de la tabla base que corresponde a cada tupla de la vista. Sin embargo, veremos por separado si se pueden insertar, borrar o modificar datos de las vistas creadas.

Proyecciones de atributos: Veamos una vista definida proyectando atributos de una sola tabla, y las posibles actualizaciones sobre ella.

Consideremos el siguiente ejemplo de vista, que obtiene el nombre y salario de todos los empleados (es decir, se hace solamente una proyección de datos):

```
SQL> CREATE VIEW V1
  2     AS SELECT ENAME, SAL
  3           FROM EMP;
```

Consideremos las posibles actualizaciones sobre esta vista. Evidentemente, se tratará de traducir todas las actualizaciones sobre la vista a actualizaciones sobre la tabla `EMP`.

- Inserción: Se pueden dar varios casos. Si alguno de los atributos de la tabla base que no están en la vista no admite valores nulos ni tiene valores por defecto, la inserción no es posible (ya que la traducción intentaría insertar valores nulos). Este es el caso de `V1`, ya que el atributo `EMPNO`, que no está en la definición de la vista, no admite valores nulos, por ser la clave primaria.

  ```
  SQL> INSERT INTO V1
    2     VALUES('NOMBRE',0);
  INSERT INTO V1
  *
  ERROR en línea 1:
      ORA-01400: no se puede realizar una inserción NULL en
      ("SCOTT"."EMP"."EMPNO")
  ```

  Como regla general, en el resto de los casos sí es posible la inserción, es decir, cuando los atributos de la tabla base no incluidos en la definición de la vista o bien admiten valores nulos o tienen definidos valores por defecto. Sin embargo, hay algunos casos en los que no es posible.

- Borrado: Siempre es posible, ya que las tuplas de la vista en este caso se corresponden una a una con las de la tabla base. Por lo tanto, las tuplas que se borren de la vista se borran realmente de la tabla base.

  En este sentido, debemos matizar algo. Veamos la siguiente vista y una consulta sobre ella:

  ```
  SQL> CREATE VIEW VSALARIO
    2     AS SELECT SAL
    3           FROM EMP;
  ```

```
Vista creada.

SQL> SELECT * FROM VSALARIO
  2     ORDER BY SAL;

       SAL
----------
       800
       950
      1100
      1250
      1250
      1300
      1500
      1600
      2450
      2850
      2975
      3000
      3000
      5000

14 filas seleccionadas.
```

Como se ve, hay dos tuplas con salario 1250. Si intentamos borrar de la vista `VSALARIO`, no podemos distinguir entre esas dos tuplas, por lo que si ejecutamos la sentencia

```
SQL> DELETE FROM VSALARIO
  2     WHERE SALARIO=1250;

2 filas suprimidas.

SQL> SELECT COUNT(*) FROM EMP;

  COUNT(*)
----------
        12

1 fila seleccionada.
```

Vemos que se borran las dos tuplas correspondientes en la tabla base. Que *nosotros* no podamos diferenciar las tuplas no indica que Oracle no pueda, internamente, hacerlo. Por ello, si la consulta es **sólo** una proyección de atributos, siempre se podrán borrar los datos de ella.

- Modificación: Al igual que el borrado, siempre es posible modificar los datos de una vista de este tipo. Evidentemente, sólo es posible modificar aquellos atributos que aparecen en la definición de la vista.

Sólo selecciones de datos: Consideremos el siguiente ejemplo de vista, que obtiene (todos los atributos) de aquellos empleados que ganan más de 1.250 €. Es importante hacer notar que no se eliminan los duplicados de las filas obtenidas, es decir, no se usa la palabra clave `DISTINCT` en la definición de la vista.

```
SQL> CREATE VIEW VISTA2
  2    AS SELECT *
  3       FROM EMP
  4       WHERE SAL > 1250;
```

Veamos las posibles actualizaciones sobre este tipo de vista:

- Inserción: Siempre es posible insertar tuplas, ya que cada una de las tuplas de la vista se corresponde directamente a una tupla de la tabla base, y se conocen todos los atributos. Sin embargo, las inserciones pueden dar lugar a situaciones extrañas. Por ejemplo, si insertamos una fila en la anterior vista, con un salario menor o igual a 1250, la fila se inserta en la tabla base `EMP`, pero no aparece en la vista ya que no satisface la condición.

- Borrado: También es siempre posible, por el mismo motivo. Las tuplas que se borren de la vista se borran realmente de la tabla base. En este caso las situaciones extrañas del caso de la inserción no suceden, ya que a la condición del borrado, si existe, se añade la condición de la vista antes de realizar el borrado.

  ```
  SQL> DELETE FROM VISTA2;

  9 filas suprimidas.

  SQL> SELECT COUNT(*) FROM EMP;

    COUNT(*)
  ----------
           5

  1 fila seleccionada.
  ```

  Como se ve, la sentencia borra 9 tuplas, las que aparecen en la vista, pero no las demás tuplas de la tabla `EMP`. De todas formas, se sugiere actuar con cuidado en estos casos y comprobar el comportamiento de cada versión de SGBD.

- Modificación: También es posible siempre modificar los datos de una vista de este tipo, actuando con las mismas consideraciones que con el borrado de filas.

  Además, hay otro tipo de consideraciones. Veamos el siguiente ejemplo:

```
UPDATE VISTA2 SET SAL=800
    WHERE EMPNO=7499;

1 fila actualizada.

select ENAME, EMPNO, SAL
 from VISTA2;
```

ENAME	EMPNO	SAL
JONES	7566	2975
BLAKE	7698	2850
CLARK	7782	2450
SCOTT	7788	3000
KING	7839	5000
TURNER	7844	1500
FORD	7902	3000
MILLER	7934	1300

```
8 filas seleccionadas.
```

El resultado es que la fila indicada se actualiza en la tabla base `EMP`, pero como resultado de ello, esa fila desaparece de la vista (como si fuese borrada) porque ya no satisface la condición de ganar más de 1250 €.

Combinación de proyecciones y selecciones de datos: Combinando selecciones y proyecciones sobre una tabla se pueden obtener vistas con una gran potencia. Los problemas de este tipo de vistas combinarían los aspectos indicados individualmente para las selecciones o las proyecciones.

Consultas con `DISTINCT` o agrupamiento sobre una tabla Consideremos la siguiente definición de vista, que utiliza `DISTINCT` para eliminar tuplas duplicadas:

```
SQL> CREATE VIEW VISTA3
  2    AS SELECT DISTINCT JOB, DEPTNO
  3      FROM EMP;
```

A diferencia de las vistas definidas anteriormente, para esta no es posible saber la tupla original de la tabla base sobre la que se ha definido la vista, ya que se eliminan los duplicados.

- Inserción: No es posible, ya que al intentar trasladar la inserción a la tabla base, no se sabría cómo realizar la inserción. Por ejemplo, no se sabría cuantas copias de la fila se podrían insertar, ya que al usar `DISTINCT` en la definición de la vista se eliminarían los duplicados.

- Borrado: Aunque sería teóricamente posible, borrando todas las copias de la tabla base que dan lugar a cada fila de la vista que se desea borrar, no se permite el borrado, ya que eso no sería el resultado deseado en muchos casos.
- Modificación: Por el mismo motivo, aún siendo teóricamente posible la actualización, no se permite en este tipo de vistas.

Consideremos ahora una vista que incluye agrupamiento y funciones de agregación:

```
SQL> CREATE VIEW EMP_POR_DEPT
  2     AS SELECT DEPTNO, COUNT(*) AS NUMEMP
  3          FROM EMP
  4          GROUP BY DEPTNO;
```

De nuevo, en este caso las tuplas de la vista no se corresponden una a una con las tuplas de la tabla base. Veamos caso por caso por qué las actualizaciones no se podrán realizar:

- Inserción: No es posible, ya que de nuevo al intentar trasladar la inserción a la tabla base no se sabría cómo realizar la inserción. Por ejemplo, intentar ejecutar `INSERT INTO EMP_POR_DEPT VALUES(50,2)` habría que insertar 2 filas en la tabla `EMP`, pero no se sabe qué poner en ninguno de sus atributos excepto el número de departamento.
- Borrado: Sería teóricamente posible borrar las filas de la vista, pero por ejemplo intentar ejecutar `DELETE FROM EMP_POR_DEPT  WHERE DEPTNO=30`, dado que la vista sólo nos dice cuántos empleados tiene el departamento, no parece lógico borrar todas las filas del la tabla `EMP` de ese departamento. Por lo tanto, el borrado no se permite en vistas definidas con agrupamientos y/o funciones de agregación.
- Modificación: Tampoco se permiten las modificaciones en este tipo de vistas. En algún caso sería teóricamente posible, por ejemplo, `UPDATE EMP_POR_DEPT SET DEPTNO=50 WHERE DEPTNO=30`, pero el resultado, que sería modificar el número de departamento de los empleados correspondientes, no sería lógico. Otro tipo de modificaciones, como `UPDATE EMP_POR_DEPT SET NUMEMP=2 WHERE DEPTNO=10`, se podría llevar a cabo borrando una fila de la tabla `EMP` de ese departamento (ya que actualmente tiene 3), pero tampoco sería una actuación correcta, por lo que no se permite.

Por lo tanto, como se ha visto, ninguna vista definida utilizando `DISTINCT` ni agrupamientos y/o funciones de agregación permite la actualización de los datos directamente a través de la vista.

Vistas definidas sobre más de una tabla Es común definir vistas sobre más de una tabla, haciendo joins entre varias. Por ejemplo, la siguiente vista nos daría los nombres de empleados y los nombres de los departamentos a los que pertenecen:

```
SQL> CREATE VIEW EMP_DEPT
  2     AS SELECT ENAME, DNAME
  3          FROM EMP E, DEPT D
  4          WHERE E.DEPTNO=D.DEPTNO;
```

Según el estándar SQL2 (el que *supuestamente* sigue Oracle), cuando hay más de una tabla en la cláusula `FROM` de la sentencia `SELECT` que define la vista, no es posible realizar ningún tipo de actualización directamente sobre la vista. La razón es que no hay forma genérica fiable de trasladar los cambios deseados de la vista a las tablas base. Oracle permite, bajo algunas circunstancias, hacer algunas actualizaciones sobre este tipo de vistas, pero ya que no es estándar y es algo que puede cambiar, no entraremos en detalles sobre eso y consideraremos que no se pueden realizar actualizaciones sobre vistas definidas sobre más de una tabla.

7.5.6. With check option

Hemos visto que a la hora de actualizar, insertar o borrar filas se pueden producir fenómenos extraños como insertar una fila en la vista que luego no es visible en dicha vista o actualizar una fila y que desaparezca de la vista.

Esto es así porque las filas existen en una vista porque satisfacen la condición `WHERE` de la consulta de definición. Si una fila es modificada de modo que ya no satisface la condición `WHERE`, entonces desaparecerá de la vista. Del mismo modo, aparecerán nuevas filas en la vista cuando una inserción o actualización haga que dichas filas satisfagan la condición `WHERE`. Estas filas que entran o salen de la vista se llaman *filas emigrantes*.

En el estándar de SQL se introdujo una cláusula opcional en la sentencia de creación de vistas para controlar estas filas. El formato la sentencia de creación con esta variación sería:

```
CREATE [OR REPLACE] VIEW <nombre_vista> [( <esquema_vista> )]
   AS <sentencia select>
   [WITH [CASCADED|LOCAL] CHECK OPTION]
```

Generalmente, la cláusula `WITH CHECK OPTION` prohíbe la migración de tuplas hacia o desde la vista. Los cualificadores `LOCAL/CASCADED` son aplicables a jerarquías de vistas: es decir, vistas que son derivadas de otras vistas. En este caso, si se especifica `WITH LOCAL CHECK OPTION` en la definición de una vista, entonces cualquier fila insertada o actualizada en la vista y sobre cualquier vista directamente o indirectamente definida sobre dicha vista no debe causar la migración de la fila en la vista, a no ser que la fila también desaparezca de la tabla/vista/s de la que es creada.

Si se especifica `WITH CASCADED CHECK OPTION` en la definición de la vista (modo por defecto), entonces la inserción o actualización sobre esta vista o sobre cualquier vista directamente o indirectamente definida sobre la vista en cuestión no puede provocar la migración de filas.

Esta opción es útil porque cuando un `INSERT` o un `UPDATE` sobre la vista viola la condición `WHERE` de la definición de la vista, la operación es rechazada. `WITH CHECK OPTION` sólo se puede especificar en vistas actualizables.

7.6. Seguridad y permisos (en Oracle)

7.6.1. Los conceptos de "Role" y "PUBLIC"

Un *role* en un sistema de bases de datos puede entenderse como cualquiera de los siguientes dos conceptos, que son equivalentes:

- Un conjunto de permisos.

- Un papel que un usuario desempeña ante la base de datos.

Por ejemplo, un usuario o conjunto de usuarios pueden desempeñar el papel de "facturador" en una empresa, con lo que entenderíamos el role como el segundo concepto, el papel que desempeñan esos usuarios. Equivalentemente, podríamos verlo como un conjunto de permisos, los necesarios para que estos usuarios realicen su labor. Por ejemplo, podrían acceder (seleccionar) datos en una tabla de facturas, y actualizar los campos que indican que la factura ha sido emitida o cobrada, pero no podrían acceder a la tabla de empleados para ver sus salarios.

Un role se crea con la sentencia `CREATE ROLE <nombre de role>`. Por ejemplo, `CREATE ROLE facturador`.

Como veremos, un usuario de la base de datos puede actuar bajo más de un role. Existe el comando `SET ROLE <nombre de role>` que hace que el usuario adquiera los permisos asociados a ese role (si tiene permiso para adquirir ese role) y pierda los permisos asociados al role anterior.

Existen varios roles predefinidos en Oracle. Entre ellos destacan `CONNECT` (el más básico), `RESOURCE`, o `DBA` (el máximo). Si un usuario intenta cambiar de role usando uno para el que no tiene permiso, recibirá un mensaje de error y el role no se cambia. Puede hacerse una prueba con un usuario normal al intentar ejecutar `SET ROLE DBA`.

Existe además un role especial llamado `PUBLIC`, al que tienen acceso todos los usuarios. Representa al usuario "genérico" de la base de datos, lo que engloba a todos los usuarios. Cuando un permiso se da a `PUBLIC`, se da a todos los usuarios de la base de datos. A veces se considera a `PUBLIC` como un usuario, sin embargo hay que tener en cuenta que no lo es realmente, ya que no sirve para identificarse ante la base de datos y conectarse a ella. Para esto es necesario disponer de un login y una clave "normales".

7.6.2. Gestión de permisos

Los permisos de un gestor de bases de datos se pueden clasificar en dos tipos:

- Permisos que afectan a todo el sistema: por ejemplo, la posibilidad de crear o borrar usuarios, crear una base de datos, etc.

- Permisos que afectan a un objeto de la base de datos: insertar datos en una tabla, ejecutar un procedimiento almacenado, etc.

De todas formas, las sentencias para dar o retirar permisos serán las mismas para ambos tipos, aunque con ligeras variaciones de sintaxis. Para dar un permiso se utiliza la sentencia `GRANT`, y para retirarlo la sentencia `REVOKE`. De la misma forma, dado que un role se puede ver como un conjunto de permisos, se usan las mismas sentencias para dar o retirar los permisos que tiene un role a otro role o a un usuario. Evidentemente, un usuario sólo podrá dar o quitar permisos que tenga (y que tenga el poder de dar o quitar).

7.6.3. Permisos globales

Los permisos que afectan a todo el sistema se pueden ver en la tabla `SYSTEM_PRIVILEGE_MAP` del catálogo de la base de datos. La sentencia `GRANT` para dar permisos de este tipo tiene el siguiente formato:

```
GRANT {Permiso|Role} [, {Permiso|Role}...]
   TO {usuario|Role|PUBLIC} [, ...]
   [WITH ADMIN OPTION]
```

Es decir, se puede dar un permiso, un conjunto de permisos individuales, o uno o varios roles, a una lista de usuarios, roles, o a `PUBLIC`.

Por ejemplo, la siguiente sentencia da los permisos `DROP ANY TABLE` (borrar cualquier tabla) y `CREATE USER` (crear usuarios), además del role `CONNECT` (permiso para conectarse a la base de datos) a los usuarios `scott` y `maria`.

```
GRANT CONNECT, DROP ANY TABLE, CREATE USER
   TO scott, maria
```

La cláusula opcional `WITH ADMIN OPTION` se usa para dar un permiso con la posibilidad de que el receptor pueda dar ese permiso a otros.

Para revocar un permiso se utiliza la sentencia `REVOKE`:

```
REVOKE {Permiso|Role|ALL PRIVILEGES} [, {Permiso|Role}...]
   FROM {usuario|Role|PUBLIC} [, ...]
```

La siguiente sentencia evita que el usuario `scott` se pueda conectar a nuestra base de datos:

```
REVOKE CONNECT
   FROM scott
```

Debemos tener además en cuenta un aspecto sobre los usuarios: los permisos individuales a un usuario toman precedencia sobre los roles, en particular sobre `PUBLIC`. Así, por ejemplo, si ejecutamos la siguiente secuencia:

```
GRANT CONNECT TO scott; REVOKE CONNECT FROM PUBLIC;
```

Impide que la generalidad de los usuarios se conecten a la base de datos, pero como el usuario `scott` tiene ese permiso dado explícitamente a él, sí podrá conectarse.

Si se utiliza `REVOKE ALL PRIVILEGES`, el usuario o role perderá todos los permisos.

Permisos a nivel de tabla El segundo tipo de permisos afectaban a objetos individuales de la base de datos. Dado que los más importantes afectan a las tablas, veremos estos en profundidad.

Cuando un usuario crea una tabla, será su dueño o *owner*, por lo que tiene todos los permisos sobre ella, y la potestad de traspasarlos a otros usuarios, roles o a `PUBLIC`. También podrá darlos aquél usuario que los reciba con el poder de traspasarlos. Los permisos que afectan a las tablas pueden verse en la tabla `TABLE_PRIVILEGE_MAP` del catálogo de la base de datos.

La sentencia básica para dar permisos es de nuevo `GRANT`, ahora con la siguiente sintaxis:

```
GRANT Permiso [, Permiso...]
   ON <nombre tabla>
   TO {usuario|Role|PUBLIC} [, ...]
   [WITH GRANT OPTION]
```

Los permisos a nivel de tabla son básicamente de seleccionar, insertar, modificar y borrar datos, alterar tablas, etc. (`INSERT`, `UPDATE`, `DELETE`, `ALTER TABLE`, ...). Para algunos de ellos, como por ejemplo el de `UPDATE`, se puede incluso especificar qué atributos de la tabla se pueden actualizar. En cambio, el permiso `SELECT` no tiene esa capacidad.

Si un permiso se da `WITH GRANT OPTION`, el usuario que lo recibe lo podrá dar a otros.

La siguiente sentencia da permisos para seleccionar cualquier dato y actualizar sólo el salario de la tabla `EMP`, al usuario `scott` y al role `facturador`. Además estos podrán a su vez dar esos permisos a otros.

```
GRANT SELECT, UPDATE (SAL)
   ON EMP
   TO scott, facturador
   WITH GRANT OPTION
```

Para sacar un permiso, de nuevo usaremos la sentencia `REVOKE`:

```
REVOKE {Permiso|ALL} [, {Permiso}...]
   ON <tabla>
   FROM {usuario|Role|PUBLIC} [, ...]
```

La siguiente sentencia impide que el usuario `scott` inserte datos en la tabla `EMP`.

```
REVOKE INSERT
   ON EMP
   FROM scott
```

De nuevo, si se indica `REVOKE ALL ...` el usuario o role perderá todos los permisos que tenía sobre la tabla.

Cuando se retira un permiso que fue dado con la opción `WITH GRANT OPTION` los permisos se retiran en cascada. Veamos un ejemplo, en el que se indica entre paréntesis el usuario que ejecuta cada sentencia:

```
(DBA)    GRANT SELECT ON EMP TO scott WITH GRANT OPTION;
(scott)  GRANT SELECT ON EMP TO miguel;
```

Ahora el usuario `miguel` puede acceder a la tabla `EMP`, pero tras la siguiente sentencia

```
(DBA)    REVOKE SELECT ON EMP FROM scott
```

tanto `scott` como `miguel` dejan de tener los permisos para acceder a la tabla `EMP`.

7.7. Transacciones

Un aspecto importante a decidir es el nivel o granularidad al que se bloquea. El nivel de bloqueo mayor es el que bloquea la base de datos completa, y que suele usarse sólo para tareas de administración y/o realización de copias de seguridad de la misma. Un nivel algo inferior será bloquear sólo la tabla afectada por la transacción, pero aún así es un nivel demasiado alto. Un nivel más bajo es el de bloqueo de página, que bloquea la página física en disco, y finalmente, el nivel de bloqueo de granularidad más fina es a nivel de tupla: sólo se bloquean aquellas tuplas que se van a leer o modificar. Los bloqueos pueden hacerse de forma explícita,

pero normalmente, si no se indica, el gestor de bases de datos lo hará automáticamente dependiendo de las sentencias que ejecutemos contra la base de datos.

En Oracle, cualquier acceso a la base de datos está dentro de una transacción. La primera se inicia en el momento de la conexión (por ejemplo si usamos el comando `CONNECT` desde `SQL*Plus`). Cuando se acaba una transacción, automáticamente comienza la siguiente.

Hay dos formas de acabar una transacción:

- `COMMIT`: Acabar "aceptando los cambios". Todas actualizaciones indicadas desde el inicio de la transacción se graban definitivamente en la base de datos.
- `ROLLBACK`: Acabar descartando los cambios, es decir, todos las actualizaciones a la base de datos desde el inicio de la transacción se deshacen, quedando como antes del inicio de la transacción.

Los comandos `COMMIT` y `ROLLBACK` pueden ejecutarse desde `SQL*Plus`, con el significado anteriormente indicado. Además, existe a mayores un comando para crear puntos de seguridad:

```
SAVEPOINT <nombre de punto de seguridad>
```

que da un nombre a un punto específico dentro de una transacción. En cualquier momento de la transacción podemos volver al estado en que estaba la base de datos en ese punto, utilizando el comando

```
ROLLBACK TO <nombre de punto de seguridad>
```

Finalmente, debemos hacer notar que las transacciones sólo afectan a las sentencias DML de manipulación de datos, tales como `INSERT`, `UPDATE` o `DELETE`, pero no a sentencias DML como la de `CREATE TABLE`.

Ejemplo:

```
SQL> -- Creamos una tabla SQL> CREATE TABLE tabla(
  2       CAMPO int  );
Tabla creada.

SQL> -- la sentencia DML no se ve afectada por el siguiente rollback
SQL> ROLLBACK;

Rollback terminado.

SQL> DESCRIBE tabla;
 Nombre                                            ¿Nulo?   Tipo
 ------------------------------------------------- -------- ------------
 CAMPO                                                      NUMBER(38)

SQL> --Introducimos datos SQL> INSERT INTO TABLA (CAMPO) VALUES
(1);

1 fila creada.
```

```
SQL> INSERT INTO TABLA (CAMPO) VALUES (2);

1 fila creada.

SQL> SELECT * FROM TABLA;

     CAMPO
----------
         1
         2

2 filas seleccionadas.

SQL> -- Si hacemos ROLLBACK, no se insertan los datos anteriores
SQL> ROLLBACK;

Rollback terminado.

SQL> SELECT * FROM TABLA;

ninguna fila seleccionada

SQL> INSERT INTO TABLA (CAMPO) VALUES (10);

1 fila creada.

SQL> INSERT INTO TABLA (CAMPO) VALUES (20);

1 fila creada.

SQL> SELECT * FROM TABLA;

     CAMPO
----------
        10
        20

2 filas seleccionadas.

SQL> -- Validamos los datos, terminando la transacción
SQL> COMMIT;

Validación terminada.

SQL> -- Insertamos un dato más
SQL> INSERT INTO TABLA (CAMPO) VALUES(66);
```

```
1 fila creada.

SQL> -- Creamos punto de seguridad
SQL> SAVEPOINT Despues_del_66;

Punto de grabación creado.

SQL> INSERT INTO TABLA (CAMPO) VALUES(67);

1 fila creada.

SQL> -- Creamos otro punto de seguridad
SQL> SAVEPOINT Despues_del_67;

Punto de grabación creado.

SQL> INSERT INTO TABLA (CAMPO) VALUES(68);

1 fila creada.

SQL> -- Volvemos al segundo punto de seguridad SQL> ROLLBACK TO
Despues_del_67;

Rollback terminado.

SQL> SELECT * FROM tabla;
     CAMPO
----------
        10
        20
        66
        67
4 filas seleccionadas.

SQL> DISCONNECT; Desconectado de Oracle9i Enterprise Edition
Release 9.2.0.1.0 - Production With the Partitioning, OLAP and
Oracle Data Mining options JServer Release 9.2.0.1.0 - Production
```

Como nota final, debemos indicar que el comando `DISCONNECT`, que nos desconecta de la base de datos, hace un `COMMIT` implícito, pero si cerramos el programa que accede a la base de datos de forma anormal (por ejemplo, pulsando en la "X" que cierra la ventana de `SQL*Plus WorkSheet` en Windows, o si cerramos un `xterm` en donde estemos ejecutando `sqlplus`), se hace un `ROLLBACK` implícito.

Si estamos trabajando en `SQL*Plus` podemos activar la opción `AUTOCOMMIT`, que hace un `COMMIT` automático después de cada sentencia de actualización.

```
SQL> SHOW AUTOCOMMIT autocommit OFF
SQL> SET AUTOCOMMIT ON
```

```
SQL> INSERT INTO TABLA(CAMPO) VALUES (111);

1 fila creada. Validación terminada.
```

8. El catálogo

El catálogo o diccionario del sistema contiene multitud de tablas y vistas que permiten administrar el sistema. Las tablas y vistas varían de un producto a otro.

Nosotros sólo vamos a repasar unas pocas tablas del catálogo de Oracle que nos pueden resultar imprescindibles.

Podemos consulta las tablas que componen el catálogo accediendo a la tabla `DICT`:

```
SQL> describe dict
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 TABLE_NAME                                         VARCHAR2(30)
 COMMENTS                                           VARCHAR2(4000)
```

Como se puede observar, para cada tabla hay una descripción. Esta tabla, como todas las del catálogo se consulta con una sentencia `Select` normal.

```
SQL> select * from dict where TABLE_NAME='USER_TABLES';

TABLE_NAME  COMMENTS
----------- ---------------------------------------------
USER_TABLES Description of the user's own relational tables
```

`USER_TABLES` como vemos es otra tabla fundamental, contiene gran cantidad de información sobre las tablas del usuario.

```
SQL> describe USER_TABLES
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 TABLE_NAME                                NOT NULL VARCHAR2(30)
 TABLESPACE_NAME                                    VARCHAR2(30)
 CLUSTER_NAME                                       VARCHAR2(30)
...
```

La tabla `ALL_TABLES` contiene la misma información pero de todas las tablas del sistema.

Para acceder a las restricciones que se han especificado sobre las tablas que hemos creado podemos acceder a la tabla `USER_CONSTRAINTS`:

```
SQL> describe USER_CONSTRAINTS
 Nombre                                    ¿Nulo?   Tipo
 ----------------------------------------- -------- ----------------------------
 OWNER                                     NOT NULL VARCHAR2(30)
 CONSTRAINT_NAME                           NOT NULL VARCHAR2(30)
```

```
 CONSTRAINT_TYPE                                    VARCHAR2(1)
 TABLE_NAME                                 NOT NULL VARCHAR2(30)
...
```

Aquí podemos obtener el nombre de la restricción en caso de que no lo especificáramos cuando creamos la restricción. Este nombre es fundamental para poder manipular dicha restricción con `ALTER TABLE` como hemos visto.

El atributo `CONSTRAINT_TYPE` es un carácter con el siguiente significado:

Carácter	Constraint Type
C	Check constraint (incluye NOT NULL)
P	PRIMARY KEY constraint
R	FOREIGN KEY constraint
U	UNIQUE constraint
V	WITH CHECK OPTION constraint (para vistas)

Una vez más accediendo a `ALL_CONSTRAINTS` podemos consultas todas las restricciones del sistema.

Otras tablas interesantes son: `ALL_USERS` que contiene información de los usuarios del sistema, `USER_INDEXES` (`ALL_INDEXES` los del sistema en general) con información de los índices, `TABLE_PRIVILEGES` y `COLUMN_PRIVILEGES` que muestran los privilegios sobre tablas y columnas respectivamente y, USER_VIEWS (ALL_VIEWS) que muestran información sobre las vistas.

Referencias

[Ame86] American National Standards Institute. *ANSI X3.135: The Database Language — SQL*. 1986.

[MS02] Jim Melton and Alan R. Simon. *SQL:1999 Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[RMLRO02] E. Rivero, L. Martinez, J. Benavides L. Reina, and J. Olaizola. *Introducción al SQL para Usuarios y Programadores*. Thomson, 2002.