REALIZACIÓN DE CONSULTAS

Tabla de Contenidos

BLOQUE I

CONSULTAS SIMPLES

1.- Introducción

En unidades anteriores hemos aprendido a pasar la especificación de un problema a un modelo físico implementado en un SGBD concreto, y también que los SGBD relacionales utilizan SQL como mecanismo para realizar todas las operaciones necesarias sobre la base de datos.

La finalidad de estas operaciones clasifica las sentencias SQL en tres subconjuntos del lenguaje: DDL, DML y DCL, siendo las correspondientes a DDL las primeras que se deben utilizar y que ya trabajamos en la unidad anterior.

Corresponde ahora aprender a manipular los datos a través de las sentencias correspondientes a DML que básicamente son: `SELECT`, `INSERT`, `DELETE` y `UPDATE`.

2.- La sentencia SELECT

Las consultas en lenguaje SQL se realizan utilizando una única sentencia: `SELECT`. Esta sentencia es muy versátil e implementa gran cantidad de variaciones y opciones que se irán desgranando a lo largo de la unidad. La forma general de la sentencia `SELECT` es:

```
SELECT [cualificador] [columnas_expresiones]
     FROM [tablas]
     WHERE [condicion_filas]
     GROUP BY [columnas_agrupacion]
     HAVING [condicion_agrupada]
     ORDER BY [columnas_ordenar]
```

donde:

- **cualificador** es una opción que indica el modo en que se devuelven las filas. Puede ser uno de:
  - **`ALL`**. Devuelve todas las filas que cumplen las condiciones de la consulta, incluyendo filas duplicadas. Es la opción por defecto si no indicamos ningún cualificador, por lo que si queremos filas duplicadas podemos poner `ALL` o no poner nada.
  - **`DISTINCT`**. Devuelve sólo filas con valores únicos. Este tipo de consulta es menos eficiente que `ALL` porque el SGBD se tiene que asegurar de que se eliminan las filas duplicadas por lo que sólo hay que utilizarla si es realmente necesario.
- **columnas_expresiones** son las columnas que va a tener la relación resultado. Pueden ser columnas de tablas ya existentes o ser el resultado de un cálculo realizado sobre otras columnas utilizando expresiones.
- **tablas** son las tablas de las cuales se van a obtener las columnas anteriores. Puede ser una tabla o varias tablas combinadas de distintas formas.

- **condicion_filas**. Condición que debe cumplir cada fila para que los valores que contiene aparezca en el resultado. Se aplica a cada fila de las tablas y determina si la misma va a aparecer en el resultado o no.
- **campos_agrupacion** es una lista con las columnas sobre las que se va a agrupar.
- **condicion_agrupada** es una condición que se aplica a cada elemento de la tabla después de agrupar.
- **columnas_ordenar** es un criterio de ordenación de filas. Las filas del resultado aparecerán ordenadas por el criterio que se especifique en esta claúsula.

La sentencia SELECT realiza una consulta sobre una tabla o tablas de la base de datos y devuelve una tabla temporal como resultado. Esta tabla no existe realmente en la base de datos y se construye expresamente en el momento de ejecutar la consulta y sólo es válida hasta el momento en que se le indica al SGBD que ya no se va a utilizar más, momento en que se descarta. Asimismo no se pueden modificar los contenidos de esta tabla, sólo se puede leer pero no se puede modificar.

Se irán abordando las distintas cláusulas según vayan siendo necesarias para implementar consultas más complejas.

3.- Seleccionar las columnas en una consulta

Para empezar a conocer la sentencia SELECT se va a estudiar su comportamiento sobre consultas que implican a una sola tabla. En este caso, la forma más simple de la sentencia es:

```
SELECT columnas FROM tabla
```

donde **columnas** es la especificación de las columnas que aparecerán en el resultado y **tabla** es la tabla desde la cual se va a realizar la consulta. En este caso, la operación que se realizaría sería muy similar a la operación de proyección del álgebra relacional.

Para especificar las columnas se dispone de varias opciones.

- **nombre de una columna**

  Si se especifica un nombre de una columna, ésta aparecerá en el resultado con el mismo nombre.

- ***** **(asterisco)**

  Un asterisco especifica que se desean incluir en el resultado TODAS las columnas de la tabla origen, con el nombre original.

- **Expresión**

  Una expresión puede utilizar los operadores aritméticos y funciones para realizar un cálculo. En el resultado aparecerá una columna con el resultado de ese cálculo. **La expresión se calcula para cada fila del resultado utilizando los valores de esa misma fila, por tanto no se pueden utilizar valores de otra fila para calcular una expresión.**

Por ejemplo, si la expresión indica que se desea el resultado de sumar dos columnas, se sumarán, para cada fila, el contenido de esas columnas para dicha fila y el resultado aparecerá en la columna del resultado.

Como no se garantiza el orden en que aparecerán las columnas en el resultado siempre hay que utilizar el nombre de columna para referirse a un dato en concreto en lugar de fiar la localización del dato a la posición que ocupa la columna entre el resto de ellas en el resultado.

EJEMPLO

Supongamos que tenemos una tabla con la siguiente definición:

```
CREATE TABLE empleado (
    dni CHAR(9) NOT NULL,
    nombre VARCHAR(200) NOT NULL,
    sueldo_base NUMERIC(6,2) NOT NULL,
    complementos NUMERIC(6,2) NOT NULL,
    CONSTRAINT PRIMARY KEY (dni)
);
```

Y con los datos:

dni	nombre	sueldo_base	complementos
11111111A	José López	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Para consultar los `nombres` y `dni` de los empleados se podría emplear la consulta:

```
SELECT nombre,dni FROM empleado;
```

que devolvería el resultado

nombre	dni
José López	11111111A
Ana Sánchez	22222222B
Julia Ramírez	33333333C

Si se deseara consultar la tabla completa:

```
SELECT * FROM empleado;
```

con el resultado

dni	nombre	sueldo_base	complementos
11111111A	José López	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Si se supone que el sueldo completo es el resultado de sumar el `sueldo_base` y los `complementos`, se podría obtener una tabla con `dni`, `nombre` y `sueldo_completo` utilizando la siguiente consulta:

```
SELECT dni, nombre, sueldo_base + complementos FROM empleado;
```

con el resultado:

dni	nombre	sueldo_base+complementos
11111111A	José López	1809,00
22222222B	Ana Sánchez	1780,64
33333333C	Julia Ramírez	3110,98

Como se puede comprobar, la última consulta ha devuelto una columna con un nombre un poco especial, ya que proviene del resultado de una expresión y no proviene de una columna ya existente, por lo que no tiene un nombre. Por lo tanto el sistema "se inventa" un nombre que usualmente tiene que ver con la expresión utilizada. Si esta tabla va a ser utilizada para algo más que para un vistazo a los datos, probablemente sería conveniente dar a la columna un nombre a la vez más corto y más manejable y que represente mejor lo que realmente contiene. En este caso, el nombre más conveniente sería algo como `sueldo_completo`. Para renombrar una columna en el resultado se utiliza la expresión `AS nuevo_nombre`, donde `nuevo_nombre` es el nuevo nombre que se quiere dar a esa columna. Por lo tanto, se podría rehacer la consulta introduciendo el renombrado y quedaría:

```
SELECT dni, nombre, sueldo_base + complementos AS sueldo_completo
FROM empleado;
```

Y el resultado sería ahora:

dni	nombre	sueldo_completo
11111111A	José López	1809,00
22222222B	Ana Sánchez	1780,64
33333333C	Julia Ramírez	3110,98

Es importante hacer notar que la palabra clave AS es opcional, por lo que la consulta anterior y la que se describe a continuación serían equivalentes:

```
SELECT dni, nombre, sueldo_base + complementos sueldo_completo
FROM empleado;
```

3.1.- Operadores aritméticos

Como se ha visto, se pueden utilizar operadores para calcular valores de columna sobre la marcha durante una consulta. Asimismo se pueden utilizar funciones que realizan cálculos más complejos.

Los operadores aritméticos que se pueden utilizar son los habituales:

- +

  Tanto para sumar dos números como para indicar que un número es positivo (en cuyo caso es opcional).
- –

  Tanto para restar dos números como para indicar que un número es negativo.
- *

  Para multiplicar dos números
- /

  Para dividir dos números. Hay que tener cuidado cuando el contenido de un campo puede ser cero ya que el resultado puede ser un error que detenga la consulta. Esto depende del SGBD utilizado. En caso contrario se realiza la división. Si esta no es exacta el resultado tendrá decimales.

Estos son los operadores estándar. El SGBD MariaDB añade unos cuantos no estándar:

- `DIV`
  División entera. Divide dos enteros y sólo devuelve la parte entera del cociente, sin decimales.
- `%` o `MOD` (se pueden utilizar ambos indistintamente)
  Realiza la división entera entre dos argumentos enteros y devuelve el *resto* de la división.

En cuanto a las funciones, su número y disponibilidad dependen del SGBD utilizado aunque hay algunas que son estándar. Las veremos más adelante.

EJEMPLO

Supongamos la misma tabla de los ejemplos anteriores. Se nos dice que para calcular el `sueldo_neto` del empleado es necesario sumar el `sueldo_base` y los `complementos` y deducir un 15% de retenciones a cuenta del IRPF. La consulta que realizaría esto sería:

```
SELECT dni, nombre, (sueldo_base + complementos) * 0.85 AS sueldo_neto
FROM empleado;
```

Al multiplicar por 0.85 se le está calculando el 85 % de la cantidad, lo que es lo mismo que restarle un 15%.

4.- Seleccionar las filas en una consulta

Para seleccionar las filas que devuelve una consulta se utilizan las clausulas `WHERE` y `HAVING`.

El uso es muy similar pero por ahora sólo se utilizará la clausula `WHERE` ya que `HAVING` sólo tiene sentido cuando se realiza agrupamiento de filas utilizando la clausula `GROUP BY`. Ambas se verán más adelante.

La clausula `WHERE` tiene como parámetro una condición sobre las columnas de la tabla. Es importante tener en cuenta que se puede utilizar cualquiera de las columnas de la tabla, aunque algunas de ellas no aparezcan en la lista de columnas del resultado.

La selección funciona de la siguiente manera. Se toman una a una las filas de la tabla y se le aplica la condición a los valores en cada una de ellas. Si la condición es cierta (devuelve `TRUE` o `VERDADERO`) la fila se incorpora al resultado. Si la condición no es cierta (devuelve `FALSE` o `FALSO`) la fila se descarta y no se incorporará al resultado.

Para realizar las condiciones se necesitan más operadores de los que se han presentado hasta ahora. En especial se necesitan los operadores relacionales y los operadores lógicos.

4.1.- Operadores relacionales

Los operadores relacionales o de comparación comparan dos valores para ver si cumplen una condición determinada. Si la cumplen devuelven VERDADERO o TRUE. Si no la cumplen devuelven FALSO o FALSE.

Operador	Nombre	Descripción
<	Menor que	Devuelve VERDADERO si el primer operando es menor que el segundo o FALSO en cualquier otro caso
<=	Menor o igual que	Devuelve VERDADERO si el primer operando es menor que el segundo o es igual al segundo o FALSO en cualquier otro caso.
>	Mayor que	Devuelve VERDADERO si el primer operando es mayor que el segundo o FALSO en cualquier otro caso.
>=	Mayor o igual que	Devuelve VERDADERO si el primer operando es mayor que el segundo o es igual al segundo.
=	Igual que	Devuelve VERDADERO si el primer operando tiene igual valor que el segundo o FALSO en cualquier otro caso.
<>	Distinto que	Devuelve VERDADERO si el valor del primer operando es distinto al del segundo o FALSO en cualquier otro caso

Los valores a comparar pueden ser directamente valores de columnas o también se pueden utilizar resultados de expresiones aritméticas o funciones.

4.2.- Operadores lógicos

Los operadores lógicos toman valores VERDADERO o FALSO y los combinan para producir otro valor VERDADERO o FALSO. Se utilizan para crear condiciones complejas en las que se tienen que dar varias condiciones distintas, ya sea a la vez o alternativamente.

Operador	Nombre	Descripción
NOT	No	Toma un solo operando y devuelve VERDADERO si el operando es FALSO o FALSO si el operando es VERDADERO.
AND	Y	Toma dos operandos y devuelve VERDADERO si ambos son VERDADERO. En cualquier otro caso devuelve FALSO.
OR	O	Toma dos operandos y devuelve VERDADERO si alguno de los dos (o los dos) es VERDADERO. Si ambos son FALSO devuelve FALSO.
XOR	O exclusivo	Toma dos operandos y devuelve VERDADERO sólo si uno de los dos es VERDADERO y el otro es FALSO. En cualquier otro caso devuelve FALSO.

4.3.- Precedencia de los operadores

Cuando se crea una expresión se pueden utilizar todos los operadores vistos hasta ahora para crearla: aritméticos, relacionales y lógicos. Es importante conocer en qué orden se van a realizar las operaciones para evitar problemas, ya que es posible que un orden de evaluación diferente produzca resultados diferentes.

Por ejemplo, si se tiene la expresión 1 + 2 * 3 ¿Cuánto vale?. Si se supone que la suma se realiza primero la expresión valdría 9. Si se supone que la multiplicación se realiza primero, la operación valdría 7. Se puede ver, por tanto, que es necesario definir una regla que especifique qué orden se sigue al evaluar los operadores, de forma que no se produzcan errores de cálculo.

Respecto a los tipos de operadores, los operadores aritméticos se calculan antes que los relacionales y estos a su vez antes que los lógicos.

Esto tiene sentido puesto que los operadores relacionales pueden utilizar los resultados de operadores aritméticos para sus propios cálculos, pero lo inverso no es aplicable. Por lo tanto tiene lógica que se realicen antes las operaciones aritméticas que las relacionales. Algo similar ocurre entre los operadores relacionales y los lógicos.

Pero, ¿qué ocurre si hay varios operadores del mismo tipo? Pues entonces hay que establecer un orden entre ellos.

En el caso de los operadores aritméticos, el orden es:

a) `-` (cambio de signo)

b) `*`, `/`, `DIV`, `%`, `MOD`

c) `+`, `-`

Si se encuentran varios operadores del mismo nivel (por ejemplo una suma y una resta) se realizan de izquierda a derecha.

Los operadores relacionales no tienen precedencia entre ellos porque no pueden mezclarse directamente, es decir, no se puede utilizar el resultado de un operador relacional como operando de otro operador relacional.

Los operadores lógicos tienen el siguiente orden:

1) `NOT`

2) `AND`

3) `XOR`

4) `OR`

Al igual que con los aritméticos, si se combinan varios del mismo nivel, se evalúan de izquierda a derecha.

Como recordar estas reglas puede ser difícil, existe una forma de forzar una precedencia, es decir, de indicar que queremos que unas operaciones se realicen antes que otras. La forma de lograrlo es utilizando paréntesis.

Las expresiones que estén entre paréntesis se realizan antes que cualquier otra, sea cual sea su precedencia. Si hay paréntesis dentro de otros paréntesis, primero se realizan los cálculos de los paréntesis más internos, luego los siguientes hacia afuera hasta llegar a los más externos.

Es una buena práctica el utilizar paréntesis para indicar el orden de precedencia de los operadores aunque según las reglas anteriormente descritas se sepa que la expresión se evalúa correctamente. Así, un lector ocasional o el mismo creador de la sentencia puede saber de un vistazo cómo se va a evaluar una expresión sin necesitar recurrir al manual o a otra documentación.

4.4.- Ejemplos de consultas

Una vez que se conocen las herramientas necesarias para crear condiciones se pueden realizar selecciones de filas más sofisticadas.

Si se tiene la misma tabla de los ejemplos anteriores:

dni	nombre	sueldo_base	complementos
11111111A	José López	1232,12	576,88
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Si se quiere acceder al empleado cuyo `dni` es `11111111A`, se podría emplear la sentencia:

```
SELECT * FROM empleado WHERE dni = '11111111A';
```

Nótese que los valores constantes de cadena de texto se deben delimitar con comillas simples.

El resultado sería:

dni	nombre	sueldo_base	complementos
11111111A	José López	1232,12	576,88

Si se quieren consultar los datos de todos los empleados excepto el que tiene el `dni 11111111A`, la sentencia sería:

```
SELECT * FROM empleado WHERE dni <> '11111111A';
```

con el resultado:

dni	nombre	sueldo_base	complementos
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Si se quieren obtener los empleados que cobran un `sueldo_base` de más de `1500` euros al mes, la consulta sería:

```
SELECT * FROM empleado WHERE sueldo_base > 1500;
```

con el resultado:

dni	nombre	sueldo_base	complementos
22222222B	Ana Sánchez	1656,87	123,77
33333333C	Julia Ramírez	2333,76	777,22

Que el resultado sea idéntico al del ejemplo anterior es pura coincidencia.

Los empleados que cobran un sueldo total mayor que `2500` euros se pueden consultar con:

```
SELECT * FROM empleado WHERE (sueldo_base + complementos) > 2500;
```

dni	nombre	sueldo_base	complementos
33333333C	Julia Ramírez	2333,76	777,22

5.- Operadores avanzados de selección

5.1.- Operador BETWEEN

Un tipo de selección que se presenta muy a menudo es el de seleccionar filas en las que el valor de una columna está comprendido en un intervalo determinado.

Siguiendo con la tabla del ejemplo anterior, si se quiere seleccionar a los empleados que tienen un sueldo base de 1000 a 1500 euros se podría emplear la sentencia:

```
SELECT * FROM empleado WHERE sueldo_base <= 1000 AND sueldo_base >=1500;
```

Esta sentencia, aunque correcta es engorrosa de crear y no queda clara a simple vista la intención de la condición. Es para este tipo de consultas para las que se creó el operador BETWEEN:

```
SELECT ..... WHERE expresion [NOT] BETWEEN valor_min AND valor_max
```

Para cada fila se evalúa expresión, que puede ser una columna o una expresión aritmética sobre una o más columnas y si el valor obtenido es mayor o igual a valor_min y a la vez es menor o igual a valor_max, devuelve VERDADERO. En caso contrario devuelve FALSE. Si se incluye la palabra opcional NOT, entonces devuelve justamente el valor opuesto.

Hay que tener en cuenta un detalle muy importante. El comportamiento que se ha dado es el estándar pero hay SGBDs que no incluyen los límites inferior y superior en el intervalo, es decir, si expresión vale exactamente igual a valor_min o valor_max, el valor devuelto no es VERDADERO sino FALSO. Otros motores incluyen los dos, otros incluyen uno si y otro no, por lo que hay que consultar la documentación del SGBD para saber cómo se comporta exactamente BETWEEN para ese SGBD. Por esta razón muchos usuarios de SGBDs prefieren utilizar la forma con >= y <= en lugar de BETWEEN, ya que de esta manera se sabe perfectamente el comportamiento que va a tener la consulta en cualquier SGBD.

EJEMPLO

Si se tiene la tabla anterior, la consulta para buscar a los empleados con sueldo base entre 1000 y 1500 quedaría

```
SELECT * FROM empleado WHERE sueldo_base BETWEEN 1000 AND 1500;
```

Si se quisiera localizar a los empleados que cobran en total de 1000 a 2000 euros:

```
SELECT * FROM empleado WHERE sueldo_base + complementos
BETWEEN 1000 AND 2000;
```

Y, por último, si se quisiera localizar a los empleados que cobran menos de 1000 o más de 2000 euros:

```
SELECT * FROM empleado WHERE sueldo_base + complementos
NOT BETWEEN 1000 AND 2000;
```

5.2.- Operador IN

Imaginemos que se quiere obtener al empleado cuyo dni es 11111111A. Para ello se emplearía la consulta:

```
SELECT * FROM empleado WHERE dni = '11111111A';
```

Hasta ahora nada nuevo pero, ¿y si se quieren los empleados con dni 11111111A o 33333333C? La respuesta podría ser:

```
SELECT * FROM empleado WHERE dni = '11111111A' OR dni = '33333333C';
```

Conforme se vayan añadiendo empleados la lista se hace más larga y complicada. SQL proporciona una manera abreviada de hacer esto utilizando el operador IN.

Su sintaxis es:

```
SELECT ..... WHERE expresion [NOT] IN (valor1, valor2, ...., valorN)
```

Se evalúa el valor de expresión y se compara con los distintos valores contantes que aparecen en la lista después de **IN**. Si hay coincidencia con alguno de ellos, se devuelve VERDADERO. Si no hay coincidencia con ninguno, se devuelve FALSE. Si se quiere saber si un valor NO está en la lista, se utiliza la opción NOT, que devuelve justo lo contrario.

EJEMPLO

La consulta anterior de los dnis 11111111A o 33333333C quedaría así utilizando IN:

```
SELECT * FROM empleado WHERE dni IN ('11111111A', '33333333C');
```

Si se quisieran obtener todos los empleados excepto los que tienen como dni 11111111A ó 33333333C, entonces se utilizaría:

```
SELECT * FROM empleado WHERE dni NOT IN ('11111111A', '33333333C');
```

5.3.- Operador LIKE

El operador LIKE es un poco especial debido a que los nuevos operadores introducidos en esta sección (BETWEEN e IN) servían para expresar algo que ya se podía hacer pero de forma más sencilla o legible. El operador LIKE, sin embargo, sirve para realizar una función que hasta ahora no se podía realizar: la búsqueda de patrones en columnas de tipo cadena. Se puede ver también como comparaciones de cadena pero que se hacen sobre una parte de la misma en lugar de la cadena completa, que son las comparaciones que se realizan con los operadores ya descritos (= y <>).

La sintaxis de LIKE es:

```
SELECT .... WHERE expresion [NOT] LIKE 'patron';
```

expresión puede ser cualquier expresión que devuelva una cadena y **patron** puede constar de:

- El caracter % (porcentaje) Este carácter equivale a cero o más caracteres cualesquiera.
- El caracter _ (subrayado) Este carácter equivale a un carácter cualquiera.
- Cualquier otro carácter Exactamente el carácter proporcionado.

Como cabría esperar, si se añade la partícula NOT, la condición se invierte.

Si se desea encontrar de forma literal un carácter % o _, es decir, que no actúe como comodín sino por su valor propio, se deben '*escapar*', colocando delante un carácter \ (backslash). Asimismo, el carácter backslash debe también 'escaparse' para poder ser utilizado (\\).

EJEMPLO

Patrón	Cadenas que corresponden
no%	no, nos, norte, nostalgia, novedad, 'no veo el momento de llegar'
%no	no, verano, langostino, vacuno, 'vamos de camino'
%no%	no, esnob, noble, verano, 'ellos no vienen'
no_	nos, noe, nom
_no	ono, uno, vno
_no_	anoe, vnod, hnog
no\%	no%
no\\	no\
no_	no_

Encontrar todos los empleados cuyo nombre es Jose:

```
SELECT * FROM empleado WHERE nombre LIKE 'Jose%';
```

Encontrar los empleados para los que la segunda letra del nombre no es una 'a':

```
SELECT * FROM empleado WHERE nombre NOT LIKE '_a%';
```

6.- Manejo de valores nulos

En el modelo relacional se necesita que se pueda especificar que el contenido de una casilla de una tabla pueda estar vacío. Para ello se utiliza el valor especial NULL.

La adición de NULL complica ligeramente la creación de expresiones ya que hay que definir cómo se comportan los operadores cuando alguno de los operando es NULL, circunstancia que se puede presentar en muchas situaciones.

6.1.- Operadores aritméticos

Si cualquiera de los operandos de un operador aritmético es NULL, el resultado también es NULL.

6.2.- Operadores relacionales

Si cualquiera de los operandos de un operador relacional es NULL, el resultado también es NULL.

Además se añaden los siguientes operadores relacionales para tratar con valores nulos.

IS NULL	Es NULL	Tiene un solo operando y devuelve VERDADERO si el valor del mismo es NULL o FALSO en otro caso.
IS NOT NULL	No es NULL	Tiene un solo operando y devuelve VERDADERO si el valor del mismo no es NULL o falso en otro caso.

La introducción de estos operadores es necesaria ya que la expresión

```
SELECT ... WHERE campo = NULL
```

siempre devuelve NULL (ni VERDADERO ni FALSO).

Por ejemplo, la consulta:

```
SELECT * FROM empleado WHERE nombre IS NULL;
```

devolvería aquellas filas de la tabla `empleado` en las que el campo `nombre` esté vacío (suponiendo que dicho campo permita valores vacíos).

6.3.- Operadores lógicos

Al igual que el resto de operadores, si alguno de los operandos es NULL, el resultado también lo será.

6.4.- Operador COALESCE

Dado que la mayoría de operaciones que involucran valores NULL devuelven valores NULL, se hace necesario utilizar algún mecanismo que permita realizar operaciones en las que puedan aparecer involucrados valores NULL de una forma predecible. Este mecanismo es el operador COALESCE.

El operador COALESCE tiene la forma:

```
COALESCE(expr1, expr2, ......, exprn)
```

donde **expr1**, **expr2**, **exprn**, son expresiones.

El resultado de COALESCE es el valor de la primera expresión que aparezca en su lista que no valga NULL. Dicho de otra forma, se van examinando una a una las expresiones que aparecen y se calcula su resultado. Si este es NULL, se pasa a la siguiente expresión, si no se devuelve el resultado obtenido. Si se llega a la última expresión de la lista y no se ha calculado ningún valor distinto de NULL, entonces se devuelve NULL.

EJEMPLO

```
COALESCE(1,2,3)
```

devuelve 1

```
COALESCE(NULL,2,3)
```

devuelve 2

```
COALESCE(NULL,NULL,3)
```

devuelve 3

```
COALESCE(NULL,NULL,NULL)
```

devuelve NULL

7.- Funciones

Al hablar de las expresiones se comentó que podrían constar tanto de operadores como de funciones. En esta sección se describirán algunas de las funciones existentes, ya que cada SGBD define su propio juego de funciones con su propia sintaxis y significado.

No obstante, hay algunas funciones que provienen del estándar SQL y que se implementan más o menos de forma similar en todos los SGBDs:

- `CURRENT_DATE()`

  Devuelve la fecha actual, es decir, la fecha del sistema en el momento de realizar la consulta.
- `CURRENT_TIME()`

  Igual que el anterior pero para la hora.
- `CURRENT_TIMESTAMP()`

  Devuelve la fecha y hora actuales.
- `ABS(expr)`

  Devuelve el valor absoluto de `expr`, que debe ser un valor numérico. El valor absoluto de un número es el mismo valor pero sin signo.
- `CHAR_LENGTH(expr)`

  Devuelve un valor entero indicando la longitud (en caracteres) de la cadena dada en `expr`.
- `POSITION(cad1 IN cad2)`

  Devuelve la posición (valor entero) en que aparece la cadena `cad1` dentro de la cadena `cad2`. Si `cad1` no aparece dentro de `cad2`, devuelve 0.
- `POWER(base, exponente)`

  Devuelve el resultado de elevar `base` a `exponente`, esto es, realiza el cálculo de potencias.
- `SQRT(expr)`

  Calcula la raíz cuadrada de `expr`, que tiene que ser una expresión numérica.
- `LOWER(expr)`

  `expr` debe ser una cadena y devuelve la misma cadena expresada completamente en letras minúsculas.
- `UPPER(expr)`

  Igual que `LOWER` pero pasa toda la cadena a mayúsculas.
- `SUBSTRING(cadena FROM comienzo [FOR longitud])`

  Extrae una parte de `cadena`, comenzando por el carácter que ocupa la posición `comienzo`. Si se especifica `FOR longitud`, extrae `longitud` caracteres y si no se especifica extrae hasta el final de la cadena de origen. Algunos SGBD no soportan `FOR` y extraen siempre hasta el final de la cadena.

Además de estas funciones se tienen las que proporcione cada SGBD y las agregadas que se verán más adelante.

8.- Ordenar los resultados de una consulta

El modelo relacional no garantiza ningún orden tanto en las columnas como en las filas del resultado de una operación de consulta, por lo que no se pueden hacer suposiciones sobre en qué posición aparecerá cada resultado ni en el orden en que se presentarán los mismos. Si se desea especificar la ordenación, hay que especificar la clausula `ORDER BY`.

La sintaxis de la claúsula `ORDER BY` es:

```
ORDER BY espec1, espec2, ..., especN
```

donde **`especX`** es una especificación de ordenación.

Se pueden dar más de una y el comportamiento es ordenar por la primera especificación. La segunda y siguientes se utilizan en caso de que haya ambigüedad al usar el primero. Por ejemplo, si en una tabla especificáramos que queremos ordenar por apellido primero y por nombre después, se ordenaría por apellido y se utilizaría la ordenación por nombre sólo en caso de que se encontrara más de una fila con el mismo apellido.

La especificación de ordenación tiene dos partes:

- **Indicador de columna**

  Indica sobre que columna, de entre las seleccionadas en la sentencia `SELECT`, se va a ordenar. La columna se puede indicar de tres formas:
  - Por nombre de columna.
  - Por alias (asignado utilizando `AS` alias)
  - Por posición, siendo 1 la posición de la primera columna seleccionada, 2 el de la segunda,...
- **Indicador de dirección**

  Indica si la ordenación sobre esa columna se desea en orden ascendente (los menores primero) o descendente (los mayores primero). En el primer caso se indica mediante la palabra clave `ASC` o no indicando ninguna, ya que es la opción por defecto. En el segundo se emplearía la palabra `DESC`.

EJEMPLO

Si se quiere consultar todos los `empleados`, ordenados de menor a mayor `sueldo_base` se podría emplear:

```
SELECT * FROM empleado ORDER BY sueldo_base;
```

Si sólo se desea saber el `dni` y `nombre` de los empleados, ordenados por `nombre` de forma descendente, se podría utilizar:

```
SELECT dni, nombre FROM empleado ORDER BY nombre DESC;
```

o bien

```
SELECT dni, nombre FROM empleado ORDER BY 2 DESC;
```

Si se quieren ordenar los empleados por `sueldo_base` y, en caso de que los sueldos sean iguales, por nombre, se emplearía:

```
SELECT * FROM empleado ORDER BY sueldo_base, nombre;
```

Por supuesto se puede combinar selección de columnas con filas y ordenación:

```
SELECT dni, nombre, (sueldo_base + complementos) AS sueldo_total
FROM empleado
WHERE (sueldo_base + complementos) > 1000 ORDER BY nombre, sueldo_total;
```

Mostraría el dni, nombre y sueldo total de aquellos empleados cuyo sueldo total (base + complementos) es mayor de 1000 euros, ordenados por nombre y si coinciden estos, por sueldo total.

9.- Funciones de columna

En apartados anteriores se han descrito los operadores y funciones, pero todos tenían algo en común: todos operaban a nivel de fila. A veces, sin embargo, es conveniente realizar operaciones que impliquen los valores de una columna de la tabla, en lugar de fila a fila.

Por ejemplo, puede que interese realizar la suma de los importes de las facturas para saber lo que se ha facturado. Esto no se puede hacer con lo que se ha descrito hasta ahora sin utilizar un programa externo que consulte la base de datos y realice los cálculos por su cuenta. Cabe pensar que como probablemente este tipo de cálculos son relativamente frecuentes, podrían ser realizados por el SGBD.

Para realizar este tipo de tareas se introdujeron las llamadas funciones de columna o funciones agregadas. Las funciones de columna son funciones cuyo ámbito de aplicación son un conjunto de valores, en lugar de un único valor. Si se aplica a una columna, realizará la operación indicada sobre todos los valores de dicha columna.

Hay que tener en cuenta, asimismo, que las funciones de columna sólo incluyen los valores seleccionados mediante la claúsula `WHERE`, es decir, si una consulta utiliza una función de columna sobre una columna determinada, sólo se realiza sobre los valores de esa columna correspondientes a filas que cumplan la condición dada en la claúsula `WHERE`, no sobre todas las filas de la tabla. Dicho en otras palabras: primero se realiza la selección de filas mediante `WHERE` y al resultado se le aplica la función de columna. Si se quiere hacer algún tipo de selección sobre los resultados de las funciones de columna, habría que utilizar la claúsula `HAVING` que se expondrá más adelante.

Las funciones de columna disponibles son:

- AVG

  Calcula la media aritmética de los valores de la columna. El tipo de datos de la columna debe ser numérico.
- COUNT

  Cuenta el número de valores no nulos en la columna. Es posible que hayan valores repetidos y cada uno cuenta.
- MAX

  Devuelve el mayor valor de la columna, según el tipo de dato. En el caso de números es comparación simple. En el caso de las columnas de tipo texto depende del cotejamiento.
- MIN

  Devuelve el menor valor de la columna, con las mismas consideraciones que MAX.
- SUM

  Devuelve la suma de los valores de la columna, que debe ser de tipo numérico.

EJEMPLO

Siguiendo con el ejemplo de los empleados, si se quisiera calcular el salario neto (`sueldo_base` más `complementos`) medio de la empresa se podría utilizar:

```
SELECT AVG(sueldo_base + complementos) AS salario_medio FROM empleado;
```

Si, en cambio, se deseara saber el número de empleados de la empresa:

```
SELECT COUNT(*) AS empleados FROM empleado;
```

O el numero de empleados cuyo salario neto es mayor de 1800 euros:

```
SELECT COUNT(*) AS empleados FROM empleado
WHERE (sueldo_base + complementos) > 1800;
```

El mayor salario de la empresa:

```
SELECT MAX(sueldo_base + complementos) AS sueldo_maximo FROM empleado;
```

El importe mensual de la nomina de la empresa se podría calcular fácilmente con:

```
SELECT SUM (sueldo_base + complementos) AS total_nomina FROM empleado;
```

10.- Agrupación de filas

Las funciones de columna que se han visto en el apartado anterior son muy útiles pero en la forma que hemos descrito anteriormente están limitadas a realizar operaciones sobre toda la tabla, lo que limita un poco su uso.

Supongamos la siguiente tabla:

oficina					
oficina	**ciudad**	region	dir	objetivo	venta
11	Valencia	Este	106	575000	693000
12	Alicante	Este	104	800000	735000
13	Castellón	Este	105	350000	368000
21	Badajoz	Oeste	108	725000	836000
22	A Coruña	Oeste	108	300000	186000
23	Madrid	Centro	108	225000	195000
24	Madrid	Centro	108	250000	150000
26	Pamplona	Norte		0	
28	Valencia	Este		900000	700000

Con la sentencia:

```
SELECT SUM(venta) FROM oficina;
```

se obtendría el total de ventas de la empresa, el cual es un dato útil pero, ¿y si se desea el total de ventas *de cada* `ciudad`? Con lo visto hasta ahora no se podría realizar tal tarea puesto que las funciones de columna, que son las que realizan la tarea lo hacen sobre **toda** la columna. La solución es la *agrupación de filas*.

La agrupación de filas utiliza el valor de una columna de la tabla para crear grupos de filas, cada uno de ellos con las filas que tienen un valor distinto de la tabla.

Por ejemplo, si se agrupara la tabla anterior por el nombre de la `ciudad` se formarían 6 grupos. Uno formado por las filas 1 y 9 (Valencia), otro por la fila 2 (Alicante), otro por la fila 3 (Castellón), otro por la fila 4 (Badajoz), otro por la fila 5 (A Coruña), otro por las filas 6 y 7 (Madrid) y, por último, otro con la fila 8 (Pamplona).

Si se aplica una función de columna sobre una agregación, se realiza la operación indicada *por grupo* en lugar de hacerlo sobre toda la tabla, con lo que se consigue efectivamente el realizar funciones de columna sobre subconjuntos de la tabla.

La agrupación de filas también se puede realizar sobre más de una columna. En este caso los grupos estarán formados por las filas que tengan una combinación específica de los valores contenidos en las columnas especificadas.

Cuando se realiza agrupación de filas, en la sección de selección de columnas de la sentencia `SELECT` no puede aparecer cualquier nombre de columna sino o bien sólo aquellas por las que se agrupa o bien funciones de columna sobre las demás (las que no se agrupan). No puede aparecer una columna sobre la que no se agrupe si no es dentro de una función de columna.

La agrupación de filas se lleva a cabo mediante la claúsula `GROUP BY`, en la forma:

```
GROUP BY columna1, columna2, ..., columnaX
```

donde **`columna1, columna2,... columnaX`** son las columnas sobre las que se desea agrupar.

EJEMPLO

En el caso de la tabla `oficina`, vista anteriormente, si se desean conocer las ventas totales por `ciudad`, la consulta que lo realizaría sería:

```
SELECT ciudad, SUM(venta) AS ventas FROM oficina GROUP BY ciudad;
```

que produciría el siguiente resultado:

ciudad	ventas
Valencia	1393000
Alicante	735000
Castellón	368000
Badajoz	836000
A Coruña	186000
Madrid	300000
Pamplona	0

Como se puede ver las filas correspondientes a Valencia y Madrid incluyen los totales calculados y segmentados.

El resto de ciudades no se ve afectado porque están solas en sus respectivos grupos por lo que la suma no hace nada.

10.1.- Selección de filas agrupadas

La clausula `GROUP BY` realiza agrupación de columnas y produce datos resumidos, ideales para informes o resúmenes.

Desgraciadamente, con lo que se ha visto hasta ahora es imposible seleccionar *en el resumen* aquellas filas que cumplan un criterio determinado. Esto es así porque la clausula `WHERE` selecciona las filas *antes* de realizar la agrupación, por lo que no afecta a las filas agrupadas.

Para proporcionar esta funcionalidad se introduce la claúsula `HAVING` con la siguiente sintaxis:

```
HAVING expresion
```

donde **`expresion`** es una expresión sobre las columnas del resumen que se evalúa por cada fila del resumen.

Si al aplicar la expresión sobre la fila se obtiene un valor `TRUE` o `VERDADERO`, la fila se incorporará al resultado. Si no lo obtiene la fila no se incorporará al resultado.

EJEMPLO

Si se desease conocer las ventas totales por `ciudad`, ya se ha visto la consulta que habría que realizar en el ejemplo del apartado anterior pero, ¿Y si se quisiera obtener las ventas totales para las `oficinas` que han vendido más de `500000` euros?. La solución sería utilizar la clausula `HAVING`:

```
SELECT ciudad, SUM(venta) AS ventas FROM oficina
GROUP BY ciudad HAVING ventas > 500000;
```

BLOQUE II

CONSULTAS COMPUESTAS Y VISTAS

1.- Introducción

En el bloque anterior se ha introducido la sintaxis básica de la sentencia `SELECT` para realizar consultas sobre una sola tabla. De esta forma se pueden realizar consultas bastante elaboradas pero no permite utilizar datos que están relacionados entre tablas, con lo que se pierde gran parte de la potencia del modelo relacional.

Mediante los JOINS se puede consultar información procedente de varias tablas de una base de datos. En este bloque se describirá cómo incorporar los JOINS a la sentencia `SELECT`, así como la forma de almacenar consultas utilizando el mecanismo de las vistas.

2.- Tipos de JOIN

Un JOIN de las tablas `tabla1` y `tabla2` es el resultado de aplicar las siguientes operaciones:

- El producto cartesiano (CROSS JOIN) de `tabla1` y `tabla2`.
- Un filtro que contiene condiciones en las que intervienen campos del producto cartesiano aplicado a ambas tablas. Los registros que cumplan estas condiciones serán el resultado del JOIN.

Hay varios tipos de JOIN o combinaciones entre tablas:

- **Internas**

  Obtienen solamente los registros que están relacionados (se corresponden sus valores) en las tablas combinadas.

  Las implementan los operadores [INNER] JOIN y NATURAL JOIN.

- **Externas**

  Obtienen los registros de la combinación interna más otros registros no coincidentes en alguna de las tablas combinadas.

  Las implementan los operadores LEFT JOIN, RIGHT JOIN y FULL JOIN.

En la cláusula FROM se pueden incluir varios operadores JOIN para combinar ciertas tablas que requiera cualquier consulta.

Cuando se referencia un campo cuyo nombre es el mismo en varias tablas combinadas hay que anteponer al nombre del campo el nombre de la tabla de la cual procede, utilizando la notación: `nombre_tabla.nombre_campo`. De esta forma el nombre de columna queda completamente cualificado, sin lugar a ambigüedad.

3.- Producto cartesiano (CROSS JOIN)

El producto cartesiano de dos tablas está compuesto por la suma de sus columnas (campos) y por el producto de sus filas (tuplas). Para realizar el producto cartesiano de dos o más tablas sólo hay que indicar las tablas que van a participar en la clausula `FROM`, separadas por comas.

EJEMPLO

Si se tiene la siguiente base de datos:

empleado		
nss	nombre	puesto
111	Juan Pérez	Jefe de Área
222	José Sánchez	Administrativo
333	Ana Díaz	Administrativo

email	
nss	email
111	jefe2@ecn.es
111	juanp@ecn.es
222	jsanchez@ecn.es
333	adiaz@ecn.es
333	ana32@ecn.es

puesto	
puesto	salario
Administrativo	1500
Jefe de Área	3000

y se realiza la consulta:

```
SELECT * FROM empleado, puesto;
```

se obtendría el siguiente resultado:

nss	nombre	puesto	puesto	salario
111	Juan Pérez	Jefe de Área	Administrativo	1500
111	Juan Pérez	Jefe de Área	Jefe de Área	3000
222	José Sánchez	Administrativo	Administrativo	1500
222	José Sánchez	Administrativo	Jefe de Área	3000
333	Ana Díaz	Administrativo	Administrativo	1500
333	Ana Díaz	Administrativo	Jefe de Área	3000

Como se puede ver hay dos columnas con el nombre `puesto`, lo cual puede dar lugar a confusiones.

Como se indicó anteriormente, para distinguir entre varias columnas con el mismo nombre en diferentes tablas se utiliza la notación `tabla.columna` para referirse a una columna sin ambigüedades.

Por tanto, la consulta anterior se podría hacer de la siguiente forma para que sólo se muestre la columna `puesto` proveniente de `empleado`:

```
SELECT empleado.*, puesto.salario FROM empleado, puesto;
```

que produciría el siguiente resultado:

nss	nombre	puesto	salario
111	Juan Pérez	Jefe de Área	1500
222	José Sánchez	Administrativo	1500
333	Ana Díaz	Administrativo	1500
111	Juan Pérez	Jefe de Área	3000
222	José Sánchez	Administrativo	3000
333	Ana Díaz	Administrativo	3000

Es posible mostrar solo algunos campos de ambas tablas en el producto cartesiano.

```
SELECT nss, nombre, puesto.salario FROM empleado, puesto;
```

El producto cartesiano en si no es una operación muy útil a la hora de obtener datos relacionados entre distintas tablas, ya que combina cualquier fila de una tabla con todas las de la otra, tengan o no tengan relación. Sin embargo el operador JOIN permite seleccionar las filas que cumplan ciertas condiciones del producto cartesiano y descartar el resto. Lo vamos a ver en los siguientes apartados.

4.- Combinación natural interna

Permite asociar tablas dentro de una sentencia SELECT de forma que se pueden unir varias tablas para que devuelva un único conjunto y la unión asocia las filas correctas en cada tabla sobre la marcha. La combinación es creada cuando se necesita y persiste durante la ejecución de la consulta.

4.1.- INNER JOIN

Se define como el producto cartesiano de varias tablas del que se eliminan aquellas filas en las que no coinciden los valores de los campos que sirven para relacionar las tablas (claves primarias y ajenas).

Hay distintas sintaxis para conseguir una combinación interna.

4.1.1.- INNER JOIN con cláusula WHERE

En este caso se realiza un producto cartesiano y en la cláusula `WHERE` se incluye una condición para dejar sólo las filas en las que los campos clave coincidan.

EJEMPLO

Siguiendo con el ejemplo anterior:

```
SELECT * FROM empleado, puesto WHERE empleado.puesto = puesto.puesto;
```

4.1.2.- `INNER JOIN` con cláusula FROM

En este caso se introduce una nueva cláusula que sirve exclusivamente para realizar este tipo de combinaciones. La cláusula se introduce dentro de la clausula `FROM` con la siguiente sintaxis:

```
tabla1 [INNER] JOIN tabla2 ON condicion;
```

donde,

- **`tabla1`** y **`tabla2`** son las dos tablas a combinar.
- **`condicion`** es la condición que deben cumplir las filas que pasarán al resultado.

La palabra `INNER` es opcional, pudiéndose utilizar únicamente `JOIN`.

EJEMPLO

Siguiendo con el ejemplo de los apartados anteriores, con esta sintaxis quedaría:

```
SELECT * FROM empleado
INNER JOIN puesto
ON empleado.puesto = puesto.puesto;
```

O bien:

```
SELECT * FROM empleado
JOIN puesto
ON empleado.puesto = puesto.puesto;
```

Como se puede ver, la condición de unión se especifica utilizando la cláusula especial `ON` en lugar de la cláusula `WHERE`. La condición que se pasa a `ON` es la misma que se pasaría a `WHERE`.

4.2.- NATURAL JOIN

En el caso de que la(s) columna(s) clave que une(n) las tablas **tenga el mismo nombre en ambas tablas**, se puede utilizar una sintaxis alternativa y abreviada de la cláusula `INNER JOIN`:

```
tabla1 [INNER] JOIN tabla2 USING (columnas_clave);
```

Tanto en `tabla1` como en `tabla2` debe haber unas columnas con esos nombres y son sobre las que se realizará la combinación. Si hay más de una columna se deberán separar por comas.

Un efecto adicional del uso de esta sintaxis es que la clave por la que se realiza la combinación no aparecerá dos veces en el resultado, sino que sólo aparecerá una vez.

EJEMPLO

Siguiendo con el ejemplo:

```
SELECT * FROM empleado INNER JOIN puesto USING(puesto);
```

O bien:

```
SELECT * FROM empleado JOIN puesto USING(puesto);
```

NOTA

El `puesto` *que aparece entre los paréntesis se refiere al nombre del campo, no de la tabla.*

En el caso de que la columna clave que une las tablas tenga el **mismo nombre en ambas tablas, y sean las únicas columnas de entre las dos tablas cuyos nombres coincidan**, se puede utilizar una sintaxis alternativa y mucho más abreviada de la clausula `INNER JOIN`:

```
tabla1 NATURAL JOIN tabla2;
```

donde todo es igual que en el caso anterior excepto que en la sentencia desaparece cualquier mención de las columnas clave.

Se examinan ambas tablas y si hay alguna coincidencia entre los nombres de alguna columna en ambas tablas se realiza automáticamente un `USING` sobre dichas columnas. Si hay más de una columna cuyos nombres coincidan, se realiza la unión sobre todas ellas simultáneamente.

Al igual que antes, un efecto adicional del uso de esta sintaxis es que la(s) columna(s) por la(s) que se realiza la combinación no aparecerá(n) dos veces en el resultado.

EJEMPLO

Siguiendo con nuestro caso:

```
SELECT * FROM empleado NATURAL JOIN puesto;
```

Ya que la única columna cuyo nombre coincide en ambas tablas es `puesto`, se realiza la unión por ésta.

5.- Combinación externa (OUTER JOIN)

La combinación natural debe su nombre a que es la forma natural de combinar tablas pero cuando se presentan campos con valor nulo en las claves, las filas que los contienen no aparecen en los resultados. Usualmente, este es el comportamiento deseado pero a veces se desea que aparezcan en el resultado de una combinación filas que no participan en la misma porque contienen valores nulos en las claves de relación. La combinación que produce estos resultados es la combinación externa (`OUTER JOIN`).

En la combinación externa u `OUTER JOIN`, siempre hay una tabla maestra y una subordinada. En el resultado aparecerán **TODAS las filas de la tabla maestra** y las filas correspondientes de la tabla subordinada con el mismo valor del campo clave. Si una fila de la tabla maestra contiene `NULL` en el valor clave o no se encuentra una coincidencia del valor de su campo clave con el campo correspondiente en la tabla subordinada, las columnas correspondientes a la tabla subordinada aparecerán todos con valor `NULL`.

Por ejemplo, se puede utilizar este tipo de uniones para las siguientes tareas:

- Contar cuántos pedidos ha hecho cada cliente, incluyendo los clientes que aún no han hecho un pedido.
- Listar todos los productos con cantidades de pedidos, incluyendo los productos que no han sido pedidos por nadie.

En estos ejemplo, la unión incluirá filas de la tabla que no tienen filas asociadas en la tabla relacionada.

En resumen, se puede decir que una combinación externa obtiene el resultado de una combinación interna más todas aquellas filas que no cumplan el filtro del `JOIN`.

Hay dos tipos de combinación externa según las posiciones relativas de las tablas maestra y subordinada: combinación externa izquierda (`LEFT JOIN`) o combinación externa derecha (`RIGHT JOIN`). Realmente sólo haría falta una de las dos puesto que son la misma operación y sólo cambia el orden en que aparecen las tablas.

5.1.- Combinación externa izquierda (LEFT JOIN)

Se incluiría dentro de la clausula FROM con la siguiente sintaxis:

```
tabla1 LEFT [OUTER] JOIN tabla2 ON condicion
```

donde:

- **tabla1** es la tabla maestra, todas las filas de esta tabla aparecerán en el resultado.
- **tabla2** es la tabla subordinada, en el resultado aparecerán las filas de esta tabla relacionadas con valores de la tabla maestra.
- **condicion** es la condición que se debe cumplir para que las filas se unan.

Una sintaxis alternativa sería:

```
tabla1 LEFT [OUTER] JOIN tabla2 USING (columnas)
```

donde **columnas** es una lista de nombres de columna separados por comas. Ambas tablas deben tener columnas con esos nombres y se mostrarán en el resultado todas las filas de tabla1 y las que tengan los mismos valores en los campos dados en la tabla2. Si no hay ninguna con los mismos valores, los campos de tabla2 en el resultado tendrán el valor NULL.

EJEMPLO

Supongamos las tablas:

empleado		
numempl	nombre	oficina
101	Antonio Viguer	12
102	Álvaro Jaumes	21
103	Juan Rovira	12
104	José González	NULL
105	Vicente Pantall	13
106	Luis Antonio Guzmán	NULL
107	Jorge Gutiérrez	22
108	Ana Bustamante	NULL
109	María Sunta	11
110	Juan Víctor	15

oficina	
oficina	ciudad
11	Valencia
12	Alicante
13	Castellón
21	Badajoz
22	A Coruña
23	Madrid
24	Madrid
26	Pamplona
28	Valencia

Si se hace una combinación externa izquierda entre `empleado` y `oficina`, utilizando:

```
SELECT * FROM empleado LEFT JOIN oficina USING (oficina);
```

proporcionaría el resultado:

numempl	nombre	oficina	ciudad
101	Antonio Viguer	12	Alicante
102	Álvaro Jaumes	21	Badajoz
103	Juan Rovira	12	Alicante
104	José González	NULL	NULL
105	Vicente Pantall	13	Castellón
106	Luis Antonio Guzmán	NULL	NULL
107	Jorge Gutiérrez	22	A Coruña
108	Ana Bustamante	NULL	NULL
109	María Sunta	11	Valencia
110	Juan Víctor	15	NULL

5.2.- Combinación externa derecha (RIGHT JOIN)

La combinación externa derecha es idéntica a la izquierda pero con la sintaxis:

```
tabla1 RIGHT [OUTER] JOIN tabla2 ON condicion
```

o

```
tabla1 RIGHT [OUTER] JOIN tabla2 USING (columnas)
```

El significado es exactamente el mismo que el de la combinación izquierda pero cambiando los papeles de `tabla1`, que ahora es la tabla subordinada, y `tabla2`, que ahora es la maestra. De hecho, a fin de evitar confusiones se recomienda realizar siempre combinaciones izquierdas.

6.- Combinaciones reflexivas

En todas las combinaciones que se han realizado hasta ahora se han utilizado dos tablas. Sin embargo ocurren casos en que las combinaciones hay que realizarlas entre una tabla consigo misma debido a una relación reflexiva.

Para ilustrar la situación se presenta la siguiente tabla:

persona					
dni	**nombre**	**apellidos**	**direccion**	**dnipadre**	**dnimadre**
11111111A	Juan	Gallardo	C/ Ventura 23	NULL	NULL
22222222B	Lucía	Sánchez	C/ Ventura 23	NULL	NULL
33333333C	Santiago	Gallardo	C/ Pozo 16	11111111A	22222222B

Esta tabla presenta dos relaciones consigo misma. La primera, a partir de la columna `dnipadre`, refleja la relación de una persona con otra que es su padre. Si el padre no está registrado, el valor de `dnipadre` es `NULL`. La otra relación es idéntica pero referente a la madre y se realiza a partir del atributo `dnimadre`.

Si ahora se quisiera consultar el nombre de la persona con `dni` 33333333C, la consulta sería:

```
SELECT nombre FROM persona WHERE dni='33333333C';
```

que devolvería:

nombre
Santiago

Hasta aquí ninguna novedad pero, ¿qué pasaría si además del nombre de la persona, se quisiera conocer el nombre del padre (o de la madre)? ¿Cómo debería ser la consulta? Una primera (apresurada y errónea) contestación podría ser:

```
SELECT persona.nombre, persona.nombre FROM persona
INNER JOIN persona
ON persona.dnipadre = persona.dni
WHERE persona.dni='33333333C';
```

El problema es que esto no funcionaría por varias razones:

- ¿A qué columna se refiere con `persona.nombre`? ¿A la de la persona o a la del padre?
- En la clausula `ON persona.dnipadre = persona.dni`, ¿a cual de las dos tablas se refiere `dnipadre`? y ¿Y `dni`?
- En la clausula `WHERE`, ¿a qué tabla se refiere `persona.dni`?

Para evitar este tipo de problemas se introduce el *alias* de tabla. El alias de tabla se incluye en la cláusula `FROM` y sirve para dar un nuevo nombre para alguna tabla de las que allí aparecen. Dicha tabla se podrá referenciar con su nuevo nombre o alias en el resto de clausulas de la sentencia `SELECT`. Esta funcionalidad no sólo se puede utilizar con consultas con relaciones reflexivas sino que se puede utilizar cuando se desee. La única consideración es que, en caso de que se quieran realizar combinaciones entre una tabla consigo misma, este renombrado no sólo es conveniente sino que es necesario.

Ahora, la consulta "imposible" anterior quedaría como:

```
SELECT persona.nombre, padre.nombre AS nombrePadre FROM persona
INNER JOIN persona AS padre
ON persona.dnipadre = padre.dni
WHERE persona.dni='33333333C';
```

Ahora si queda perfectamente claro a qué columnas se refieren tanto la lista de selección como las clausulas `FROM` y `WHERE`, y se tendría el resultado:

nombre	nombrePadre
Santiago	Juan

7.- Consultas combinadas

Todas las consultas vistas contienen una sola sentencia `SELECT` que devuelve datos de una o más tablas. En este apartado aprenderemos a realizar múltiples consultas (múltiples sentencias `SELECT`) y devolver los resultados como un único conjunto de resultados de la consulta.

Existen básicamente dos escenarios en los que se pueden utilizar las consultas combinadas:

- Para devolver datos de estructura similar de diferentes tablas en una sola consulta.
- Para realizar múltiples consultas contra una sola tabla devolviendo los datos como una sola consulta.

En la mayoría de los casos, la combinación de dos consultas a la misma tabla consigue lo mismo que una sola consulta con múltiples condiciones de la cláusula `WHERE`. Sin embargo, el rendimiento de una y otra puede variar, por lo que es bueno experimentar cuál es preferible.

7.1.- Unión

El uso del operador UNION es bastante sencillo. Lo único que hay que hacer es especificar cada sentencia SELECT y colocar la palabra clave UNION entre cada una de ellas.

La sintaxis es:

```
SELECT columnas1 [resto_select1]
UNION
SELECT columnas2 [resto_select2]
UNION
…
UNION
SELECT columnasX [resto_selectX];
```

donde:

- **columnasN** son las columnas que se seleccionan en cada SELECT
- **resto_selectN** son las clausulas SELECT usuales (WHERE, ORDER BY,...)

Los distintos juegos de columnas (columnas1, columnas2,...) deben contener las mismas columnas, expresiones o funciones de agregación (aunque no es necesario que aparezcan en el mismo orden) y los tipos de datos de las columnas deben ser compatibles.

Por defecto, el operador UNION elimina cualquier fila duplicada del conjunto de resultados de la consulta. Esto se puede cambiar utilizando UNION ALL en lugar de UNION.

EJEMPLO

empleado		
dni	**nombre**	**fecha_nac**
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Luis Sánchez	06/08/1986
77777777M	Juana Martín	01/02/1954

propietario		
dni	**fecha_nac**	**nombre_completo**
77777777M	01/02/1954	Juana Martín
99999999Z	02/02/1964	Héctor Rodríguez

Si se tienen las tablas:

La consulta:

```
SELECT dni, nombre, fecha_nac FROM empleado
UNION
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

produciría el resultado:

empleado		
dni	**nombre**	**fecha_nac**
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Luis Sánchez	06/08/1986
77777777M	Juana Martín	01/02/1954
99999999Z	Héctor Rodríguez	02/02/1964

Sólo hay una fila para el `dni` '`77777777M`' a pesar de que aparece en las dos tablas.

En cuanto a la ordenación de los resultados de las consultas combinadas con `UNION`, solo se puede utilizar una cláusula `ORDER BY`, que debe aparecer después de la última sentencia `SELECT`.

7.2.- Intersección

La intersección permite seleccionar sólo las filas que están en dos resultados.

Su sintaxis es:

```
SELECT columnas1 [resto_select]
INTERSECT
SELECT columnas2 [resto_select];
```

donde **`columnas1`** y **`columnas2`** deben cumplir las mismas condiciones que para las uniones.

EJEMPLO

Si se toman las mismas tablas de ejemplo que para la unión, la sentencia:

```
SELECT dni, nombre, fecha_nac FROM empleado
INTERSECT
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

produciría

dni	nombre	fecha_nac
77777777M	Juana Martín	01/02/1954

7.3.- Diferencia

La diferencia entre dos tablas devuelve las filas que están en la primera tabla pero que no están en la segunda.

Su sintaxis es:

```
SELECT columnas1 [resto_select]
MINUS
SELECT columnas2 [resto_select]
```

donde `columnas1` y `columnas2` deben cumplir las mismas condiciones que para las uniones o intersecciones.

EJEMPLO

Si se siguen utilizando las mismas tablas, la sentencia:

```
SELECT dni, nombre, fecha_nac FROM empleado
MINUS
SELECT dni, nombre_completo, fec_nac FROM propietario;
```

devolvería

dni	nombre	fecha_nac
11111111A	Juan García	17/03/1998
22222222B	Ana López	18/07/1983
33333333C	Luis Sánchez	06/08/1986

NOTA

En MariaDB el operador MINUS se escribe como EXCEPT.

8.- Subconsultas

Una subconsulta es una consulta que aparece dentro de otra consulta, por tanto, encontraremos una sentencia SELECT que se utiliza dentro de otra sentencia SELECT. La subconsulta obtiene unos resultados intermedios (uno o más datos) que se emplean en la consulta principal.

Las subconsultas proporcionan un mecanismo muy potente para realizar consultas que de otra forma, serían muy difíciles o imposibles de realizar. De hecho, esta posibilidad es el motivo de la palabra "estructurada" que da el nombra a SQL (Lenguaje de Consultas Estructuradas, Structured Query Language).

Sintaxis y características

Una subconsulta tiene la misma sintaxis que una sentencia SELECT normal pero con las siguientes apreciaciones:

- Debe aparecer siempre entre paréntesis.
- No se puede utilizar cláusulas `ORDER BY` ya que los resultados de una subconsulta se utilizan internamente y no son visibles al usuario.
- Se pueden anidar subconsultas dentro de subconsultas.
- Los nombres de columna que aparecen en las expresiones en una subconsulta pueden referirse a columnas de la tabla de la consulta principal y se conocen como referencias externas.
- No se pueden utilizar columnas de la subconsulta en el resultado de la consulta que la contiene.

Se puede limitar el número de filas devueltas en un SELECT mediante la clausula `TOP` junto con un número que indica el número de filas a devolver.

NOTA

MariaDB utiliza la clausula `LIMIT`, con la misma funcionalidad pero después de `ORDER BY`.

Para los ejemplos se utilizará la BD `empleado` de la relación de ejercicios.

El siguiente muestra una consulta con subconsultas para listar los empleados que no han cubierto aún su cuota de ventas:

```
SELECT nombre FROM empleado
WHERE cuota < (SELECT SUM(importe) FROM pedido WHERE rep = numempl);
```

Funcionamiento

Para cada fila de la consulta se ejecuta la subconsulta y con ese resultado se evalúa la fila correspondiente de la consulta, mostrándose si el resultado de la evaluación es verdadero.

En el ejemplo anterior, por cada fila de la tabla de `empleado` (consulta externa) se calcula la subconsulta y se evalúa la condición `(<)`. Como se puede ver, hay que realizar una consulta *por cada fila* de la tabla `empleado`. Una opción muy potente, pero que puede ralentizar el rendimiento.

Un excesivo anidamiento puede penalizar fuertemente el rendimiento de la consulta en su conjunto, por lo que no se deben anidar a la ligera. La consulta más externa se denomina consulta externa y las subconsultas, consultas internas.

Las consultas que utilizan subconsultas se pueden realizar en la mayoría de los casos utilizando otros métodos, pero como se puede ver en el ejemplo, son más fáciles de seguir e interpretar que las consultas que utilizan composición y, en los nuevos SGBDs su eficiencia está cerca de estar a la par.

Referencias

A menudo es necesario acceder, desde dentro de una subconsulta, al valor de una columna de la fila actual en la consulta externa. La referencia a la columna de la consulta externa desde el interior de la subconsulta se denomina *referencia externa*.

En el ejemplo anterior, dentro de la subconsulta se hace referencia a la columna `numempl`. Esta es una referencia externa, ya que dicha columna no está entre las columnas de la tabla que aparecen en la clausula `FROM` de la subconsulta.

Cuando aparece un nombre de columna dentro de una subconsulta se busca primero entre los nombres de columna de las que forman parte de las tablas en la clausula `FROM` de la subconsulta. Si no aparece allí, intenta buscarla en la consulta externa. Si se encuentra allí, se considera como una referencia externa. Si tampoco se encuentra allí, se busca la siguiente consulta externa, hasta llegar a la más externa. Si se sigue sin encontrar es un error.

Cuando se encuentra una referencia externa, su aparición se sustituye por el valor que tiene en la fila que se está procesando en cada momento en la consulta externa y se ejecuta la subconsulta como si se tratara de un valor constante. Por ejemplo, en la consulta anterior, para cada fila se obtiene el valor de `numempl` y se coloca en el lugar que ocupa el nombre de la columna en la subconsulta. Acto seguido se ejecuta la subconsulta y el resultado se utiliza para compararlo con el valor de cuota *de esa misma fila*.

Clasificación

Atendiendo al número de filas y columnas que devuelve la subconsulta distinguimos:

- Subconsultas que devuelven un único valor (una fila con una única columna).
- Subconsultas que devuelven una única fila con más de una columna.
- Subconsultas que devuelven un conjunto de filas.

8.1.- Subconsultas que devuelven una única fila y una única columna

Puede ser usada prácticamente en cualquier lugar de una consulta principal aunque los lugares más habituales son las cláusulas `WHERE` y `HAVING`.

Este tipo de subconsultas se pueden utilizar para:

- Incluirlo en la lista de selección de la consulta externa.
- Como operando de un operador relacional (`<`, `>`, `<=`, `>=`, `=`, `<>`) en la cláusula WHERE o HAVING de la consulta externa.

El valor devuelto se utilizará para:

- Incluirlo en la fila del resultado correspondiente.
- Realizar la comparación con otro valor.

Si la subconsulta no devuelve ninguna fila, se considerará que devuelve NULL.

Si la subconsulta devuelve más de una fila, se producirá un error.

EJEMPLO

La consulta de ejemplo en el apartado anterior, es una consulta con subconsulta de resultado único.

```
SELECT nombre FROM empleado
WHERE cuota < (SELECT SUM(importe) FROM pedido WHERE rep = numempl);
```

Y esta también es una subconsulta de resultado único.

```
SELECT p.id_producto as codigo, p.descripcion,
    (SELECT SUM(pe.cantidad)
     FROM pedido pe
     WHERE pe.producto = p.id_producto) AS total
FROM producto p;
```

Ejecuta lo siguiente y compáralo con la sentencia anterior.

```
ALTER TABLE pedido CHANGE COLUMN producto id_producto varchar(50);

SELECT p.id_producto as codigo, p.descripcion, sum(pe.cantiad) as total
FROM producto p LEFT JOIN pedido pe USING (id_producto)
GROUP BY p.id_producto, p.descripcion
ORDER BY id_producto;
```

¿Por qué ha sido necesario realizar previamente `ALTER TABLE`?

8.2.- Subconsultas que devuelven muchas filas y una única columna

Si la subconsulta devuelve un conjunto de valores (filas) (cero, uno o más) entonces no se pueden comparar con un comparador relacional (<, >, <=, >=, =, <>). En tal caso hay que recurrir a los operadores especiales: IN, NOT IN, ALL, ANY, EXISTS, NOT EXISTS.

A continuación veremos cada uno de estos operadores.

8.2.1.- Operador IN

Se utiliza para determinar si una expresión pertenece a un conjunto de valores:

```
expresion IN (subconsulta)
```

El resultado puede ser:

- VERDADERO si expresión coincide con alguno de los valores devueltos por la subconsulta.
- FALSO si expresión no coincide con ninguno de los valores devueltos por la consulta.
- FALSO si la subconsulta no devuelve ningún valor.
- NULL si expresión es NULL.
- NULL si ninguno de los valores devueltos por la subconsulta coincide con la expresión y además se ha devuelto algún NULL en la subconsulta.

EJEMPLO

Seleccionar todos los productos de los que hay algún pedido realizado.

```
SELECT p.id_producto, p.descripcion FROM producto p
WHERE p.id_producto IN (SELECT pe.producto FROM pedido pe);
```

8.2.2.- Operador NOT IN

Se utiliza para determinar si un valor no pertenece a un conjunto de valores:

```
expresion NOT IN (subconsulta)
```

El resultado puede ser:

- VERDADERO si expresión es distinta de todos los valores devueltos por la subconsulta.
- VERDADERO si la subconsulta no devuelve ningún valor. (¡Cuidado!)
- FALSO si expresión coincide con alguno de los valores devueltos por la consulta.
- NULL si expresion es NULL.
- NULL si ninguno de los valores devueltos por la subconsulta coincide con la expresión y además se ha devuelto algún NULL en la subconsulta.

8.2.3.- Operador ANY

El operador debe ser una comparación: <, >, <=, >=, =, <>

```
expresion operador ANY (subconsulta)
```

El resultado puede ser:

- VERDADERO si la comparación es cierta para alguno de los valores de la subconsulta.
- FALSO si la comparación es falsa para todos los valores de la subconsulta.
- FALSO si la subconsulta no devuelve ninguna fila.
- NULL si ninguno de los valores devueltos por la subconsulta coincide con la expresión y además se ha devuelto algún NULL en la subconsulta.

El operador IN es equivalente a = ANY.

EJEMPLO

Para consultar los empleados de oficinas del Este:

```
SELECT * FROM empleado
WHERE oficina = ANY (SELECT oficina FROM oficina WHERE region = 'Este');
```

o, utilizando IN

```
SELECT * FROM empleado
WHERE oficina IN (SELECT oficina FROM oficina WHERE region = 'Este');
```

8.2.4.- Operador ALL

El operador debe ser una comparación: <, >, <=, >=, =, <>

```
expresion operador ALL (subconsulta)
```

El resultado puede ser:

- VERDADERO si la comparación es cierta para todos los valores de la subconsulta.
- VERDADERO si la subconsulta no devuelve ninguna fila. (¡Cuidado!)
- FALSO si la comparación no es cierta para algún valor de la subconsulta.
- NULL si la subconsulta devuelve algún nulo.

El operador NOT IN es equivalente a <> ANY.

EJEMPLO

Si se quiere obtener los empleados de las oficinas que NO son del Este, la consulta podría ser:

```
SELECT * FROM empleado
WHERE oficina <> ALL (SELECT oficina FROM oficina
                      WHERE region = 'Este');
```

o bien

```
SELECT * FROM empleado
WHERE oficina NOT IN (SELECT oficina FROM oficina
                      WHERE region = 'Este');
```

8.2.5.- Operador EXISTS

El operador `EXISTS` y su contrario (`NOT EXISTS`) suelen necesitar emplear las referencias externas.

Las referencias externas se presentan cuando una subconsulta está parametrizada mediante valores de columnas de la consulta principal. A este tipo de subconsultas se les llama correlacionadas y a los parámetros de la subconsulta que pertenecen a la consulta principal se les llama referencias externas.

Tanto `EXISTS` como `NOT EXISTS` se pueden usar tanto en la cláusula `WHERE` como `HAVING`.

La forma de uso de `EXISTS` es:

```
EXISTS (subconsulta)
```

El resultado puede ser:

- `VERDADERO` si la subconsulta devuelve al menos una fila.
- `FALSO` si la subconsulta no devuelve ninguna fila.

Este operador no necesita terminar de ejecutar completamente la subconsulta, ya que puede terminar en cuanto encuentre una fila que le permita devolver verdadero.

Como sólo se valora la existencia o no de resultados, sin que importen las columnas que tengan estos, lo usual es utilizar * como lista de selección de columnas.

EJEMPLO

Si se quieren obtener los empleados que tienen algún pedido del fabricante `ACI`, la consulta se podría hacer de la siguiente forma:

```
SELECT * FROM empleado
WHERE EXISTS (SELECT * FROM pedido WHERE numempl = rep AND fab = 'ACI');
```

8.3.- Subconsultas que devuelven una única fila y muchas columnas

En este caso el operador que se puede utilizar es solamente =.

```
(expr1, expr2,…) operador (subconsulta)
```

La subconsulta debe devolver una sola fila y tantas columnas como las existentes entre paréntesis a la izquierda del operador, por tanto el número de columnas debe coincidir.

Si la subconsulta no devuelve ninguna fila se evalúa como NULL.

Si la subconsulta devuelve más de una fila se produce un error.

Hay que tener en cuenta que, tanto en la cláusula WHERE como HAVING, se cumple cuando el resultado que evalúan es verdadero. Si el predicado que evalúan es falso o NULL, se considera que la condición no se cumple.

8.4.- Subconsultas que devuelven muchas filas y muchas columnas

Si la subconsulta devuelve un conjunto de filas (cero, uno o más) entonces no se pueden comparar con un comparador relacional (`<, >, <=, >=, =, <>`). En tal caso hay que recurrir a los operadores `IN`, NOT IN.

También se puede utilizar EXISTS ya que realmente solo evalúa si la subconsulta devuelve un conjunto no vacío de filas.

8.4.1.- Operador IN

Se utiliza para determinar si una fila pertenece a un conjunto de filas:

```
(expr1, expr2,...) IN (subconsulta)
```

La subconsulta debe devolver tantas columnas como las especificadas entre paréntesis para las expresiones. Estas expresiones se evalúan y conforman una fila que se compara una a una con las filas de la subconsulta.

El resultado puede ser:

- `VERDADERO` si encuentra alguna fila igual en la subconsulta.
- `FALSO` si no encuentra ninguna fila igual en la subconsulta.
- `FALSO` si la subconsulta no devuelve ninguna fila.
- `NULL` si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila formada por las expresiones.

EJEMPLO

Seleccionar los pedidos y su fecha donde se haya comprado un mismo producto y en la misma cantidad que en el pedido 112997.

```
SELECT DISTINCT p.num_pedido, p.fecha_pedido
FROM pedido p JOIN producto p2
WHERE p.num_pedido <> 112997
AND (p.ide_producto, p.cantidad) IN (SELECT p3.id_producto, p3.cantidad
                                     FROM pedido p3
                                     WHERE p3.num_pedido = 112997);
```

8.4.2.- Operador NOT IN

Se utiliza para determinar si una fila no pertenece a un conjunto de filas:

```
(expr1, expr2,...) IN (subconsulta)
```

La subconsulta debe devolver tantas columnas como las especificadas entre paréntesis para las expresiones. Estas expresiones se evalúan y conforman una fila que se compara una a una con las filas de la subconsulta.

El resultado puede ser:

- VERDADERO si no se encuentra ninguna fila igual en la subconsulta.
- VERDADERO si la subconsulta no devuelve ninguna fila. (¡Cuidado!)
- FALSO si se encuentra alguna fila igual en la subconsulta.
- NULL si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila formada por las expresiones.

8.5.- Subconsultas en la cláusula FROM

Las subconsultas se pueden utilizar también para “crear” tablas que se pueden utilizar en la consulta externa como si fueran tablas “reales” de la base de datos. Esto es lo que se conoce por *tablas derivadas*.

Para poder utilizar las tablas derivadas las subconsultas deben aparecer dentro de la clausula FROM de la consulta externa, en la forma:

```
SELECT lista_de_campos FROM (subconsulta) AS nombre_tabla
```

La “tabla” así creada se puede utilizar como si fuera una tabla real en la consulta externa.

Cuando se utilizan de este modo, las subconsultas tienen diferencias con lo que se ha visto hasta ahora:

1. La subconsulta **no** se realiza varias veces sino que sólo se realiza una vez. Por lo tanto **no** pueden utilizar valores de campos que no estén en la lista `FROM` de la subconsulta.
2. Es obligatorio dar un nombre a la tabla derivada utilizando la clausula `AS` después de la subconsulta.
3. **Todas las columnas de la subconsulta** deben tener un nombre válido.

Si se cumplen todas las condiciones necesarias, las tablas derivadas se pueden utilizar en el lugar en que esté permitido utilizar una tabla, incluidas composiciones (`JOIN`).

EJEMPLO

La siguiente consulta obtiene los datos de los empleados que no han cubierto su cuota.

```
SELECT * FROM empleado
WHERE cuota >= (SELECT SUM(importe) FROM pedido WHERE rep = numempl);
```

Si se quieren obtener los clientes de estos empleados habría que hacer una composición entre este resultado y la tabla `cliente`. Esto se puede hacer utilizando una subconsulta en la clausula `FROM` de la forma:

```
SELECT cliente.nombre FROM cliente
INNER JOIN (SELECT * FROM empleado
            WHERE cuota >= (SELECT SUM(importe) FROM pedido
                            WHERE rep = numempl)) AS empleados2
ON cliente.repclie = empleados2.numempl;
```

Como se puede ver, se utiliza la “tabla” resultado de la consulta previa nombrada como `empleados2` y se compone con `cliente`. También se pueden ver dos consultas anidadas.

9.- Vistas

Muchas bases de datos son utilizadas por distintos usuarios que tienen distintas necesidades y distintos niveles de seguridad.

Otra necesidad que surge a veces es la de usar con mucha frecuencia datos agregados (medias, sumas totales,...) pero en las tablas hay que almacenar los datos en detalle, ya que la información que contienen puede ser necesaria en otros contextos. En este caso sería interesante disponer de tablas “virtuales” que permitieran consultar los datos agregados mientras que se dispone de los datos desagregados en las tablas “reales”.

La solución a este problema es el concepto de *vista*.

Una vista es una tabla virtual que se crea a partir de una consulta sobre otras tablas (o vistas). Cuando se utiliza una vista, el SGBD realiza la consulta almacenada en la vista y proporciona al usuario la ilusión de que existe una tabla con la estructura y datos de la vista. Esto soluciona los dos problemas mencionados anteriormente:

- Si se desea dar acceso a un usuario a un conjunto restringido de columnas de una tabla, se crea una vista sobre la misma con sólo los campos necesarios y obviando el resto. El usuario utiliza la vista en lugar de la tabla "real". Si la tabla real se actualiza (se modifican los datos), la vista reflejará este hecho.
- Si se quiere hacer una nueva tabla con datos agregados de otras se puede hacer una vista sobre la misma utilizando las funciones agregadas. La tabla real permanece con los datos desagregados pero la virtual los tiene ya agregados y se actualizará cuando cambien aquellos.

9.1.- Creación de vistas

Para crear una vista es necesario saber realizar consultas (sentencia `SELECT`), ya que cada vista está basada en una consulta subyacente. La sintaxis para la creación de vistas es:

```
CREATE VIEW nombre_vista AS sentencia_select;
```

donde:

- `nombre_vista`

  Es el nombre de la nueva vista a crear. No puede coincidir con el nombre de otra vista o tabla ya existente en la base de datos.
- `sentencia_select`

  Es la sentencia `SELECT` que va a proporcionar el esquema y los datos de la vista.

EJEMPLO

Utilizando la siguiente tabla, vamos a crear la vista de visitante `empleado_v`

empleado								
codigo	nombre	edad	oficina	titulo	fecha_contrato	codigo_jefe	cuota_ventas	ventas_obtenidas
101	Antonio Viguer	45	12	Representante	20/10/06	104	300000	305000
102	Álvaro Jaumes	48	21	Representante	10/12/06	108	350000	474000
103	Juan Rovira	29	12	Representante	01/03/07	104	275000	286000
104	José González	33	12	Dir Ventas	19/05/07	106	200000	143000
105	Vicente Pantall	37	13	Representante	12/01/08	104	350000	368000
106	Luis Antonio Rodríguez	52	11	Dir General	14/06/08	NULL	275000	299000
107	Jorge Gutiérrez	49	22	Representante	14/11/08	108	300000	186000
108	Ana Bustamante	62	21	Dir Ventas	12/10/09	106	350000	361000
109	María Sunta	31	11	Representante	12/10/16	106	300000	392000
110	Juan Víctor	41	22	Representante	13/01/10	104	500000	760000

La sintaxis sería:

```
CREATE VIEW empleado_v AS SELECT nombre, oficina FROM empleado;
```

La sentencia SELECT puede utilizar todas las cláusulas y subconsultas que necesite, aunque sólo puede especificar una cláusula ORDER BY si también incluye una clausula TOP. De todas formas, no se garantiza el orden de las filas de la vista. Si se desean ordenar por algún criterio hay que especificarlo al crear la sentencia SELECT que consulte los datos de la vista.

EJEMPLO

Si se quiere crear una vista con los cinco empleados con más ventas, se podría hacer utilizando:

```
CREATE VIEW empleado_top5
AS SELECT TOP 5 * FROM empleado ORDER BY ventas_obtenidas DESC;
```

Esto crearía una vista llamada empleado_top5 con los empleados con más ventas pero no garantiza que al hacer:

```
SELECT * FROM empleado_top5;
```

las filas aparezcan ordenadas por las ventas del empleado. Si se quiere asegurar que las filas devueltas estarán ordenadas por este criterio, hay que declararlo expresamente al realizar la consulta sobre la vista:

```
SELECT * FROM empleado_top5 ORDER BY ventas_obtenidas DESC;
```

Otra consideración importante del uso de vistas es que los cambios a las tablas subyacentes (las tablas sobre las que se basa la consulta) no modifican esta automáticamente, por lo que se pueden producir problemas. Por ejemplo, si se elimina un campo de una tabla subyacente que es utilizado en la vista, la vista fallará al utilizarla. Esto es así porque el sistema almacena sólo la consulta y la intenta ejecutar al utilizar la vista. Como ahora la consulta no es válida porque se utiliza un campo no existente se produce un error.

Por último hay que decir que, dado que la vista genera una "tabla", es necesario que en la lista de selección se den nombre (utilizando AS) a las columnas calculadas, de forma que la vista tenga una columna con un nombre válido.

9.2.- Modificación y eliminación de vistas

Para modificar una vista hay dos sintaxis distintas pero que realizan una función equivalente:

```
CREATE OR REPLACE VIEW nombre_vista AS sentencia_select;
```

o bien

```
ALTER VIEW nombre_vista AS sentencia_select;
```

En ambas tanto `nombre_vista` como `sentencia_select` tienen el mismo significado que en la sentencia `CREATE VIEW`. En principio la mayoría de los SGBDs soportan ambas sintaxis aunque hay que revisar la documentación del SGBD en que se esté trabajando para ver si una sintaxis es soportada.

Una sutil diferencia entre las dos sintaxis es que `CREATE OR REPLACE VIEW` puede utilizarse siempre, aunque la vista no esté ya creada, mientras que `ALTER VIEW` necesita que la vista ya esté creada. Por tanto, se recomienda el uso de `CREATE OR REPLACE VIEW` siempre que sea posible.

EJEMPLO

Si ahora en nuestro "directorio" de empleados se desea incluir el número del mismo para facilitar su localización, se podría hacer utilizando:

```
CREATE OR REPLACE VIEW empleado_v
AS SELECT codigo, nombre, oficina FROM empleado;
```

O bien:

```
ALTER VIEW empleado_v AS SELECT codigo, nombre, oficina FROM empleado;
```

Para eliminar una vista la sintaxis es:

```
DROP VIEW nombre_vista;
```

EJEMPLO

```
DROP VIEW empleado_v;
```