# Capítulo 4

# Lenguaje SQL

## Introducción y objetivos

Las siglas SQL corresponden a *Structured Query Language*, un lenguaje estándar que permite manejar los datos de una base de datos relacional. La mayor parte de los SGBD relacionales implementan este lenguaje y mediante él se realizan todo tipo de accesos a la base de datos. En este capítulo se hace una presentación del lenguaje SQL, haciendo énfasis en la sentencia de consulta de datos, la sentencia `SELECT`.

Al finalizar este capítulo, el estudiante debe ser capaz de:

- Emplear la sentencia `CREATE TABLE` para crear tablas a partir de una especificación dada.
- Emplear las sentencias `INSERT, UPDATE, DELETE` para insertar, actualizar y borrar datos de tablas de una base de datos.
- Emplear la sentencia `SELECT` para responder a cualquier consulta de datos sobre una base de datos dada.
- Especificar una sentencia `SELECT` equivalente a otra dada que no haga uso de los operadores que se indiquen, con el objetivo de intentar acelerar el tiempo de respuesta.

## 4.1. Bases de datos relacionales

Como se ha visto en capítulos anteriores, una base de datos relacional está formada por un conjunto de relaciones. A las relaciones, en SQL, se las denomina *tablas*. Cada tabla tiene una serie de *columnas* (son los atributos). Cada columna tiene un nombre distinto y es de un tipo de datos (entero, real, carácter, fecha, etc.). En las tablas se insertan *filas* (son las tuplas), que después se pueden consultar, modificar o borrar.

No se debe olvidar que cada tabla tiene una clave primaria, que estará formada por una o varias columnas de esa misma tabla. Sobre las claves primarias se debe hacer respetar una regla de integridad fundamental: la regla de integridad de entidades. La mayoría de los SGBD relacionales se encargan de hacer respetar esta regla automáticamente.

Por otra parte, las relaciones entre los datos de distintas tablas se establecen mediante las claves ajenas. Una clave ajena es una columna o un conjunto de columnas de una tabla que hace referencia a la clave primaria de otra tabla (o de ella misma). Para las claves ajenas también se debe cumplir una regla de integridad fundamental: la regla de integridad referencial. Muchos SGBD relacionales permiten que el usuario establezca las reglas de comportamiento de las claves ajenas que permiten hacer respetar esta regla.

## 4.2. Descripción de la base de datos

En este apartado se presenta de nuevo la base de datos con la que se ha trabajado en capítulos anteriores y que es la que se utilizará para estudiar el lenguaje SQL en este capítulo. Para evitar problemas de implementación se han omitido las tildes en los nombres de tablas y columnas.

La base de datos está formada por las tablas que aparecen a continuación. Las columnas subrayadas representan la clave primaria de cada tabla.

```
  CLIENTES(codcli, nombre, direccion, codpostal, codpue)
VENDEDORES(codven, nombre, direccion, codpostal, codpue, codjefe)
   PUEBLOS(codpue, nombre, codpro)
PROVINCIAS(codpro, nombre)
 ARTICULOS(codart, descrip, precio, stock, stock_min, dto)
  FACTURAS(codfac, fecha, codcli, codven, iva, dto)
LINEAS_FAC(codfac, linea, cant, codart, precio, dto)
```

A continuación se especifican las claves ajenas y si aceptan nulos:

| | | | | |
|---|---|---|---|---|
| CLIENTES | $\xrightarrow{\texttt{codpue}}$ | PUEBLOS | : | No acepta nulos. |
| VENDEDORES | $\xrightarrow{\texttt{codpue}}$ | PUEBLOS | : | No acepta nulos. |
| VENDEDORES | $\xrightarrow{\texttt{codjefe}}$ | VENDEDORES | : | Acepta nulos. |
| PUEBLOS | $\xrightarrow{\texttt{codpro}}$ | PROVINCIAS | : | Acepta nulos. |
| FACTURAS | $\xrightarrow{\texttt{codcli}}$ | CLIENTES | : | Acepta nulos. |
| FACTURAS | $\xrightarrow{\texttt{codven}}$ | VENDEDORES | : | Acepta nulos. |
| LINEAS_FAC | $\xrightarrow{\texttt{codfac}}$ | FACTURAS | : | No acepta nulos. |
| LINEAS_FAC | $\xrightarrow{\texttt{codart}}$ | ARTICULOS | : | No acepta nulos. |

La información contenida en esta base de datos pertenece a una empresa de venta de artículos eléctricos. A continuación se describe el contenido de cada tabla.

La tabla `PROVINCIAS` almacena información sobre las provincias de España. De cada provincia se almacena su nombre (`nombre`) y un código que la identifica (`codpro`).

La tabla `PUEBLOS` contiene los nombres (`nombre`) de los pueblos de España. Cada pueblo se identifica por un código que es único (`codpue`) y tiene una referencia a la provincia a la que pertenece (`codpro`).

La tabla `CLIENTES` contiene los datos de los clientes: código que identifica a cada uno (`codcli`), nombre y apellidos (`nombre`), calle y número (`direccion`), código postal (`codpostal`) y una referencia a su población (`codpue`).

La tabla `VENDEDORES` contiene los datos de los vendedores de la empresa: código que identifica a cada uno (`codven`), nombre y apellidos (`nombre`), calle y número (`direccion`), código postal (`codpostal`), una referencia a su población (`codpue`) y una referencia al vendedor del que depende (`codjefe`), si es el caso.

En la tabla `ARTICULOS` se tiene el código que identifica a cada artículo (`codart`), su descripción (`descrip`), el precio de venta actual (`precio`), el número de unidades del artículo que hay en el almacén (`stock`), si se conocen, la cantidad mínima que se desea mantener almacenada (`stock_min`), si es que la hay, y si el artículo está en oferta, el descuento (`dto`) que se debe aplicar cuando se venda.

La tabla `FACTURAS` contiene las cabeceras de las facturas correspondientes a las compras realizadas por los clientes. Cada factura tiene un código único (`codfac`), la fecha en que se ha realizado (`fecha`), así como el IVA (`iva`) y el descuento que se le ha aplicado (`dto`). Si el IVA o el descuento no se especifican, se deben interpretar como el valor cero (sin IVA o sin descuento). Es imporante tener en cuenta que se está haciendo un mal uso de los nulos, ya que interpretar los nulos con valores supone un trabajo extra cuando se hacen las consultas. Sin embargo, en muchas bases de datos se hace este uso no apropiado de los nulos y, por lo tanto, el estudio del lenguaje SQL requiere aprender manejarse con ellos. Cada factura también hace referencia al cliente

al que pertenece (`codcli`) y al vendedor que la ha realizado (`codven`). Ambas claves ajenas aceptan nulos.

Las líneas de cada factura se encuentran en la tabla `LINEAS_FAC`, identificándose cada una por el número de línea que ocupa dentro de la factura (`codfac, linea`). En cada una de ellas se especifica la cantidad de unidades (`cant`) del artículo que se compra (`codart`), el precio de venta por unidad (`precio`) y el descuento que se aplica sobre dicho precio (`dto`), si es que el artículo está en promoción. Si el descuento no se especifica, se debe interpretar como sin descuento (valor cero).

La figura 4.1 muestra el esquema de la base de datos gráficamente.

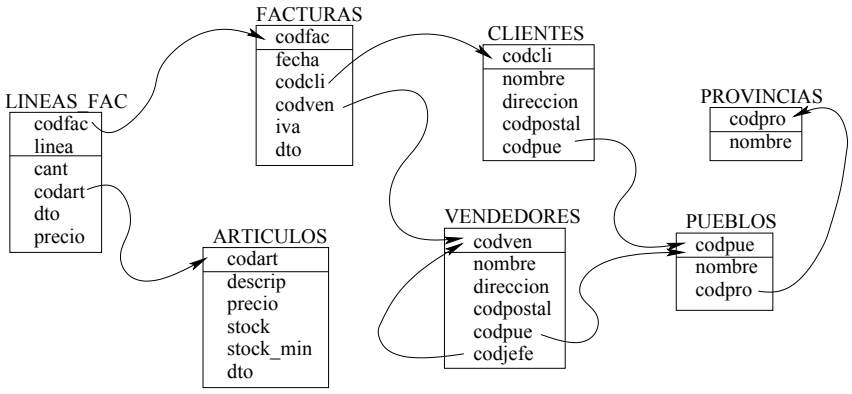


Figura 4.1: Esquema de la base de datos que se utilizará en los ejemplos.

## 4.3. Visión general del lenguaje

Normalmente, cuando un SGBD relacional implementa el lenguaje SQL, todas las acciones que se pueden llevar a cabo sobre el sistema se realizan mediante sentencias de este lenguaje. Dentro de SQL hay varios tipos de sentencias que se agrupan en tres conjuntos:

- *Sentencias de definición de datos*: son las sentencias que permiten crear tablas, alterar su definición y eliminarlas. En una base de datos relacional existen otros tipos de objetos además de las tablas, como las vistas, los índices y los disparadores, que se estudiarán más adelante. Las sentencias para crear, alterar y eliminar vistas e índices también pertenecen a este conjunto.

- *Sentencias de manejo de datos*: son las sentencias que permiten insertar datos en las tablas, consultarlos, modificarlos y borrarlos.

- *Sentencias de control*: son las sentencias que utilizan los administradores de la base de datos para realizar sus tareas, como por ejemplo crear usuarios y concederles o revocarles privilegios.

Las sentencias de SQL se pueden escribir tanto en mayúsculas como en minúsculas, y lo mismo sucede con los nombres de las tablas y de las columnas. Para facilitar la lectura de los ejemplos, se utilizará mayúsculas para las palabras clave del lenguaje y minúsculas para los nombres de tablas y de columnas. En los ejemplos se introducirán espacios en blanco para tabular las expresiones. Las sentencias de SQL terminan siempre con el carácter punto y coma (;).

### 4.3.1. Creación de tablas

Para crear una tabla en una base de datos se utiliza la sentencia `CREATE TABLE`. Su sintaxis es la siguiente:

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

donde `restricción_columna` es:

```
[ CONSTRAINT nombre_restricción ]
  { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (expr) |
    REFERENCES tablaref [ ( columnaref ) ]
    [ ON DELETE acción ] [ ON UPDATE acción ] }
  [ DEFERRABLE | NOT DEFERRABLE ]
  [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

y `restricción_tabla` es:

```
[ CONSTRAINT nombre_restricción ]
  { UNIQUE ( nombre_columna [, ... ] ) |
    PRIMARY KEY ( nombre_columna [, ... ] ) |
    CHECK ( expr ) |
    FOREIGN KEY ( nombre_columna [, ... ] )
        REFERENCES tablaref [ ( columnaref [, ... ] ) ]
        [ MATCH FULL | MATCH PARTIAL ]
        [ ON DELETE acción ] [ ON UPDATE acción ] }
```

A continuación se especifica el significado de cada identificador y de cada cláusula de la sentencia `CREATE TABLE`:

- `nombre_tabla`: nombre de la nueva tabla.
- `nombre_columna`: nombre de una columna de la tabla.

- `tipo_datos`: tipo de datos de la columna.
- `DEFAULT expr`: asigna un valor por defecto a la columna junto a la que aparece; este valor se utilizará cuando en una inserción no se especifique valor para la columna.
- `CONSTRAINT nombre_restricción`: a las restricciones que se definen sobre columnas y sobre tablas se les puede dar un nombre (si no se hace, el sistema generará un nombre automáticamente).
- `NOT NULL`: la columna no admite nulos.
- `NULL`: la columna admite nulos (se toma por defecto si no se especifica `NOT NULL`).
- `UNIQUE` especificada como restricción de columna indica que la columna sólo puede contener valores únicos. `UNIQUE (nombre_columna [,...])` especificada como restricción de tabla indica que el grupo de columnas sólo pueden contener grupos de valores únicos. Mediante esta cláusula se especifican las claves alternativas.
- `PRIMARY KEY` especificada como restricción de columna o bien `PRIMARY KEY (nombre_columna[,...])` especificada como restricción de tabla indica la columna o el grupo de columnas que forman la clave primaria de la tabla. Los valores de la clave primaria, además de ser únicos, deberán ser no nulos.
- `CHECK (expr)`: permite incluir reglas de integridad específicas que se comprueban para cada fila que se inserta o que se actualiza. La expresión es un predicado que produce un resultado booleano. Si se especifica a nivel de columna, en la expresión sólo puede hacerse referencia a esta columna. Si se especifica a nivel de tabla, en la expresión puede hacerse referencia a varias columnas. Por ahora no se puede incluir subconsultas en esta cláusula.
- Restricción de columna:
  ```
  REFERENCES tablaref [ ( columnaref ) ]
  [ ON DELETE acción ] [ ON UPDATE acción ]
  ```
  Restricción de tabla:
  ```
  FOREIGN KEY ( nombre_columna [, ... ] )
  REFERENCES tablaref [ ( columnaref [, ... ] ) ]
  [ MATCH FULL | MATCH PARTIAL ]
  [ ON DELETE acción ] [ ON UPDATE acción ]
  ```
  La restricción de columna `REFERENCES` permite indicar que la columna hace referencia a una columna de otra tabla. Si la referencia apunta a la clave primaria, no es necesario especificar el nombre de la columna a

la que se hace referencia (estamos definiendo una clave ajena). Cuando se añade o actualiza un valor en esta columna, se comprueba que dicho valor existe en la tabla referenciada. Cuando la restricción es a nivel de tabla (`FOREIGN KEY`) hay dos tipos de comprobación: `MATCH FULL` y `MATCH PARTIAL`. Con `MATCH FULL`, si la clave ajena está formada por varias columnas y admite nulos, esta comprobación es la que corresponde a la regla de integridad referencial: en cada fila, o todas las columnas de la clave ajena tienen valor o ninguna de ellas lo tiene (todas son nulas), pero no se permite que en una misma fila, algunas sean nulas y otras no. Con `MATCH PARTIAL`, si la clave ajena está formada por varias columnas y admite nulos, se permiten claves ajenas parcialmente nulas y se comprueba que en la tabla referenciada se podría apuntar a alguna de sus filas si los nulos se sustituyeran por los valores adecuados.

Además, se pueden establecer reglas de comportamiento para cada clave ajena cuando se borra o se actualiza el valor referenciado. En ambos casos hay cuatro posibles opciones que se enumeran a continuación. `NO ACTION` produce un error por intento de violación de una restricción. `RESTRICT` es igual que `NO ACTION`. `CASCADE` borra/actualiza las filas que hacen referencia al valor borrado/actualizado. `SET NULL` pone un nulo en las filas donde se hacía referencia al valor borrado/actualizado. `SET DEFAULT` pone el valor por defecto en las filas donde se hacía referencia al valor borrado/actualizado.

A continuación se muestra la sentencia de creación de la tabla `LINEAS_FAC`:

```
CREATE TABLE lineas_fac (
    codfac   NUMERIC(6,0) NOT NULL,
    linea    NUMERIC(2,0) NOT NULL,
    cant     NUMERIC(5,0) NOT NULL,
    codart   VARCHAR(8)   NOT NULL,
    precio   NUMERIC(6,2) NOT NULL,
    dto      NUMERIC(2,0),
    CONSTRAINT cp_lineas_fac PRIMARY KEY (codfac, linea),
    CONSTRAINT ca_lin_fac FOREIGN KEY (codfac)
               REFERENCES facturas(codfac)
               ON UPDATE CASCADE ON DELETE CASCADE,
    CONSTRAINT ca_lin_art FOREIGN KEY (codart)
               REFERENCES articulos(codart)
               ON UPDATE CASCADE ON DELETE RESTRICT,
    CONSTRAINT ri_dto_lin CHECK (dto BETWEEN 0 AND 50)
);
```

## 4.3.2. Inserción de datos

Una vez creada una tabla podemos introducir datos en ella mediante la sentencia INSERT, como se muestra en los siguientes ejemplos:

```
INSERT INTO facturas(codfac,fecha,       codcli,codven,iva,dto )
             VALUES(6600,  '30/04/2007',111,   55,    0,  NULL);
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  1,    4,   'L76425',3.16,  25 );
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  2,    5,   'B14017',2.44,  25 );
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  3,    7,   'L92117',4.39,  25 );
```

Mediante estas sentencias se ha introducido la cabecera de una factura y tres de sus líneas. Nótese que tanto las cadenas de caracteres como las fechas, se introducen entre comillas simples. Para introducir nulos se utiliza la expresión NULL.

Algunos SGBD relacionales permiten insertar varias filas en una misma tabla mediante una sola sentencia INSERT, y realizan las inserciones de un modo más eficiente que si se hace mediante varias sentencias independientes. Así, la tres inserciones que se han realizado en la tabla LINEAS_FAC también se pueden realizar mediante la siguiente sentencia:

```
INSERT INTO lineas_fac(codfac,linea,cant,codart,  precio,dto)
               VALUES(6600,  1,    4,   'L76425',3.16,  25 ),
                     (6600,  2,    5,   'B14017',2.44,  25 ),
                     (6600,  3,    7,   'L92117',4.39,  25 );
```

## 4.3.3. Consulta de datos

Una vez se ha visto cómo almacenar datos en la base de datos, interesa conocer cómo se puede acceder a dichos datos para consultarlos. Para ello se utiliza la sentencia SELECT. Por ejemplo:

```
SELECT *
FROM   facturas;
```

En primer lugar aparece la palabra SELECT, que indica que se va a realizar una consulta. A continuación, el * indica que se desea ver el contenido de todas las columnas de la tabla consultada. El nombre de esta tabla es el que aparece tras la palabra FROM, en este caso, la tabla facturas.

Esta sentencia es, sin lugar a dudas, la más compleja del lenguaje de manejo de datos y es por ello que gran parte de este capítulo se centra en su estudio.

### 4.3.4. Actualización y eliminación de datos

Una vez insertados los datos es posible actualizarlos o eliminarlos mediante las sentencias `UPDATE` y `DELETE`, respectivamente. Para comprender el funcionamiento de estas dos sentencias es imprescindible conocer bien el funcionamiento de la sentencia `SELECT`. Esto es así porque para poder actualizar o eliminar datos que se han almacenado es preciso encontrarlos antes. Y por lo tanto, la cláusula de estas sentencias que establece las condiciones de búsqueda de dichos datos (`WHERE`) se especifica del mismo modo que las condiciones de búsqueda cuando se hace una consulta.

Sin embargo, antes de pasar al estudio de la sentencia `SELECT` se muestran algunos ejemplos de estas dos sentencias.

```
UPDATE facturas        UPDATE facturas
SET    dto = 0         SET    codven = 105
WHERE  dto IS NULL;    WHERE  codven IN ( SELECT codven
                                          FROM   vendedores
                                          WHERE  codjefe = 105 );

DELETE FROM facturas   DELETE FROM facturas
WHERE  codcli = 333;   WHERE  iva = ( SELECT MIN(iva)
                                      FROM   facturas );
```

## 4.4. Estructura básica de la sentencia `SELECT`

La sentencia `SELECT` consta de varias cláusulas. A continuación se muestran algunas de ellas:

```
SELECT [ DISTINCT ] { * | columna [ , columna ] }
FROM   tabla
[ WHERE condición_de_búsqueda ]
[ ORDER BY columna [ ASC | DESC ]
           [,columna [ ASC | DESC ] ];
```

El orden en que se tienen en cuenta las distintas cláusulas durante la ejecución y la función de cada una de ellas es la siguiente:

- `FROM`: especifica la tabla sobre la que se va a realizar la consulta.
- `WHERE`: si sólo se debe mostrar un subconjunto de las filas de la tabla, aquí se especifica la condición que deben cumplir las filas a mostrar; esta condición será un predicado booleano con comparaciones unidas por `AND/OR`.
- `SELECT`: aquí se especifican las columnas a mostrar en el resultado; para mostrar todas las columnas se utiliza `*`.

- `DISTINCT`: es un modificador que se utiliza tras la cláusula `SELECT` para que no se muestren filas repetidas en el resultado (esto puede ocurrir sólo cuando en la cláusula `SELECT` se prescinde de la clave primaria de la tabla o de parte de ella, si es compuesta).

- `ORDER BY`: se utiliza para ordenar el resultado de la consulta.

La cláusula `ORDER BY`, si se incluye, es siempre la última en la sentencia `SELECT`. La ordenación puede ser ascendente o descendente y puede basarse en una sola columna o en varias.

La sentencia del siguiente ejemplo muestra los datos de todos los clientes, ordenados por el código postal, descendentemente. Además, todos los clientes de un mismo código postal aparecerán ordenados por el nombre, ascendentemente.

```
SELECT *
FROM   clientes
ORDER BY codpostal DESC, nombre;
```

### 4.4.1. Expresiones en `SELECT` y `WHERE`

En las cláusulas `SELECT` y `WHERE`, además de columnas, también se pueden incluir expresiones que contengan columnas y constantes, así como funciones. Las columnas y expresiones especificadas en la cláusula `SELECT` se pueden renombrar al mostrarlas en el resultado mediante `AS`.

Si el resultado de una consulta se debe mostrar ordenado según el valor de una expresión de la cláusula `SELECT`, esta expresión se indica en la cláusula `ORDER BY` mediante el número de orden que ocupa en la cláusula `SELECT`.

```
SELECT precio, ROUND(precio * 0.8, 2) AS rebajado
FROM   articulos
ORDER BY 2;
```

### 4.4.2. Nulos

Cuando no se ha insertado un valor en una columna de una fila se dice que ésta es nula. Un nulo no es un valor: un nulo implica ausencia de valor. Para saber si una columna es nula se debe utilizar el operador de comparación `IS NULL` y para saber si no es nula, el operador es `IS NOT NULL`.

Cuando se realiza una consulta de datos, los nulos se pueden interpretar como valores mediante la función `COALESCE(columna, valor_si_nulo)`. Esta función devuelve `valor_si_nulo` en las filas donde `columna` es nula; si no, devuelve el valor de `columna`.

```
SELECT codfac, fecha, codcli, COALESCE(iva, 0) AS iva,
       iva AS iva_null, COALESCE(dto, 0) AS dto
FROM   facturas
WHERE  codcli < 50
AND    (iva = 0 OR iva IS NULL);
```

La condición `(iva=0 OR iva IS NULL)` es equivalente a `COALESCE(iva,0)=0`.

### 4.4.3. Tipos de datos

Los tipos de datos disponibles se deben consultar en el manual del SGBD relacional que se esté utilizando. Puesto que las prácticas de las asignaturas para las que se edita este libro se realizan bajo PostgreSQL, se presentan aquí los tipos de datos que se han usado en este SGBD para crear las tablas. Todos ellos pertenecen al estándar de SQL.

- `VARCHAR(n)`: Cadena de hasta `n` caracteres.
- `NUMERIC(n,m)`: Número con `n` dígitos, de los cuales `m` se encuentran a la derecha del punto decimal.
- `DATE`: Fecha formada por día, mes y año. Para guardar fecha y hora se debe utilizar el tipo `TIMESTAMP`.
- `BOOLEAN`: Aunque este tipo no se ha utilizado en la base de datos de prácticas, es interesante conocer su existencia. El valor verdadero se representa mediante `TRUE` y el falso mediante `FALSE`. Cuando se imprimen estos valores, se muestra el carácter `'t'` para verdadero y el carácter `'f'` para falso.

Hay que tener siempre en cuenta que el nulo no es un valor, sino que implica ausencia de valor. El nulo se representa mediante `NULL` y cuando se imprime no se muestra nada.

## 4.5. Funciones y operadores

### 4.5.1. Operadores lógicos

Los operadores lógicos son `AND`, `OR` y `NOT`. SQL utiliza una lógica booleana de tres valores y la evaluación de las expresiones con estos operadores es la que se muestra en la siguiente tabla:

| a | b | a AND b | a OR b | NOT b |
|---|---|---|---|---|
| True | True | True | True | False |
| True | False | False | True | True |
| True | Null | Null | True | Null |
| False | False | False | False | |
| False | Null | False | Null | |
| Null | Null | Null | Null | |

### 4.5.2. Operadores de comparación

| | |
|---|---|
| `<` | Menor que. |
| `>` | Mayor que. |
| `<=` | Menor o igual que. |
| `>=` | Mayor o igual que. |
| `=` | Igual que. |
| `<>  !=` | Distinto de. |
| `a BETWEEN x AND y` | Equivale a: `a >= x AND a <= y` |
| `a NOT BETWEEN x AND y` | Equivale a: `a < x OR a > y` |
| `a IS NULL` | Devuelve True si `a` es nulo. |
| `a IS NOT NULL` | Devuelve True si `a` es no nulo. |
| `a IN (v1, v2, ...)` | Equivale a: `a = v1 OR a = v2 OR ...` |

### 4.5.3. Operadores matemáticos

| | |
|---|---|
| `+` | Suma. |
| `-` | Resta. |
| `*` | Multiplicación. |
| `/` | División (si es entre enteros, trunca el resultado). |
| `%` | Resto de la división entera. |
| `^` | Potencia ($3\hat{}2 = 9$). |
| `\|/` | Raíz cuadrada ($\|/25 = 5$). |
| `\|\|/` | Raíz cúbica ($\|\|/27 = 3$). |
| `!` | Factorial ($5! = 120$). |
| `!!` | Factorial como operador prefijo ($!!5 = 120$). |
| `@` | Valor absoluto. |

No se han incluido en esta lista los operadores que realizan operaciones sobre tipos de datos binarios.

### 4.5.4. Funciones matemáticas

| | |
|---|---|
| `ABS(x)` | Valor absoluto de `x`. |
| `SIGN(x)` | Devuelve el signo de `x` (-1, 0, 1). |
| `MOD(x,y)` | Resto de la división entera de `x` entre `y`. |
| `SQRT(x)` | Raíz cuadrada de `x`. |
| `CBRT(x)` | Raíz cúbica de `x`. |
| `CEIL(x)` | Entero más cercano por debajo de `x`. |
| `FLOOR(x)` | Entero más cercano por encima de `x`. |
| `ROUND(x)` | Redondea al entero más cercano. |
| `ROUND(x,n)` | Redondea `x` a `n` dígitos decimales, si `n` es positivo. Si `n` es negativo, redondea al entero más cercano a `x` múltiplo de $10^n$. |
| `TRUNC(x)` | Trunca `x`. |
| `TRUNC(x,n)` | Trunca `x` a `n` dígitos decimales, si `n` es positivo. Si `n` es negativo, trunca al entero más cercano por debajo de `x` múltiplo de $10^n$. |

Además de éstas, se suelen incluir otras muchas funciones para: calcular logaritmos, convertir entre grados y radianes, funciones trigonométricas, etc. Se aconseja consultar los manuales del SGBD que se esté utilizando, para conocer las funciones que se pueden utilizar y cuál es su sintaxis.

### 4.5.5. Operadores y funciones de cadenas de caracteres

En SQL, las cadenas de caracteres se delimitan por comillas simples: `'abc'`. Los operadores y funciones para trabajar con cadenas son los siguientes:

| | |
|---|---|
| `cadena \|\| cadena` | Concatena dos cadenas. |
| `cadena LIKE expr` | Devuelve `TRUE` si la cadena sigue el patrón de la cadena que se pasa en `expr`. En `expr` se pueden utilizar comodines: _ para un solo carácter y % para cero o varios caracteres. |
| `LENGTH(cadena)` | Número de caracteres que tiene la cadena. |
| `CHAR_LENGTH(cadena)` | Es la función del estándar equivalente a `LENGTH`. |
| `POSITION(subcadena IN cadena)` | Posición de inicio de la subcadena en la cadena. |
| `SUBSTR(cadena, n [, long])` | Devuelve la subcadena de la cadena que empieza en la posición `n` (`long` fija el tamaño máximo de la subcadena; si no se especifica, devuelve hasta el final). |

`SUBSTRING(cadena FROM n [FOR long])`
: Es la función del estándar equivalente a `SUBSTR`: devuelve la subcadena de la cadena que empieza en la posición `n` (`long` fija el tamaño máximo de la subcadena; si no se especifica, devuelve hasta el final).

`LOWER(cadena)`
: Devuelve la cadena en minúsculas.

`UPPER(cadena)`
: Devuelve la cadena en mayúsculas.

`BTRIM(cadena)`
: Elimina los espacios que aparecen por delante y por detrás en la cadena.

`LTRIM(cadena)`
: Elimina los espacios que aparecen por delante (izquierda) en la cadena.

`RTRIM(cadena)`
: Elimina los espacios que aparecen por detrás (derecha) de la cadena.

`BTRIM(cadena, lista)`
: Elimina en la cadena la subcadena formada sólo por caracteres que aparecen en la lista, tanto por delante como por detrás.

`LTRIM(cadena, lista)`
: Funciona como `BTRIM` pero sólo por delante (izquierda).

`RTRIM(cadena, lista)`
: Funciona como `BTRIM` pero sólo por detrás (derecha).

`TRIM(lado lista FROM cadena)`
: Es la función del estándar equivalente a `BTRIM` si lado es `BOTH`, equivalente a `LTRIM` si lado es `LEADING` y equivalente a `RTRIM` si lado es `TRAILING`.

`CHR(n)`
: Devuelve el carácter cuyo código ASCII viene dado por `n`.

`INITCAP(cadena)`
: Devuelve la cadena con la primera letra de cada palabra en mayúscula y el resto en minúsculas.

`LPAD(cadena, n, [, c])`
: Devuelve la cadena rellenada por la izquierda con el carácter `c` hasta completar la longitud especificada por `n` (si no se especifica `c`, se rellena de espacios). Si la longitud de la cadena es de más de `n` caracteres, se trunca por el final.

`RPAD(cadena, n, [, c])`
: Devuelve la cadena rellenada por la derecha con el carácter `c` hasta completar la longitud especificada por `n` (si no se especifica `c`, se rellena de espacios). Si la longitud de la cadena es de más de `n` caracteres, se trunca por el final.

## 4.5.6. Operadores y funciones de fecha

El tipo de datos `DATE` tiene operadores y funciones, como el resto de tipos.[1] En este apartado se muestran aquellos más utilizados, pero se remite al lector a los manuales del SGBD que esté utilizando para conocer el resto.

En primer lugar se verán las funciones que permiten convertir entre distintos tipos de datos. Todas ellas tienen la misma estructura: se les pasa un `dato` de un tipo, que se ha de convertir a otro tipo según el patrón indicado mediante un `formato`.

| | |
|---|---|
| `TO_CHAR(dato, formato)` | Convierte el `dato` de cualquier tipo a cadena de caracteres. |
| `TO_DATE(dato, formato)` | Convierte el `dato` de tipo cadena a fecha. |
| `TO_NUMBER(dato, formato)` | Convierte el `dato` de tipo cadena a número. |

A continuación se muestran algunos de los patrones que se pueden especificar en los formatos:

**Conversiones fecha/hora:**

| | |
|---|---|
| `HH` | Hora del día (1:12). |
| `HH12` | Hora del día (1:12). |
| `HH24` | Hora del día (1:24). |
| `MI` | Minuto (00:59). |
| `SS` | Segundo (00:59). |
| `YYYY` | Año. |
| `YYY` | Últimos tres dígitos del año. |
| `YY` | Últimos dos dígitos del año. |
| `Y` | Último dígito del año. |
| `MONTH` | Nombre del mes. |
| `MON` | Nombre del mes abreviado. |
| `DAY` | Nombre del día. |
| `DY` | Nombre del día abreviado. |
| `DDD` | Número del día dentro del año (001:366). |
| `DD` | Número del día dentro del mes (01:31). |
| `D` | Número del día dentro de la semana (1:7 empezando en domingo). |
| `WW` | Número de la semana en el año (1:53). |
| `W` | Número de la semana en el mes (1:5). |
| `Q` | Número del trimestre (1:4). |

[1] En PostgreSQL se puede escoger el modo de visualizar las fechas mediante `SET DATESTYLE`. Para visualizar las fechas con formato `día/mes/año` se debe ejecutar la orden `SET DATESTYLE TO EUROPEAN, SQL;`

**Conversiones numéricas:**

| | |
|---|---|
| 9 | Dígito numérico. |
| S | Valor negativo con signo menos. |
| . | Punto decimal. |
| , | Separador de miles. |

Cuando el formato muestra un nombre, utilizando en el patrón de forma adecuada las mayúsculas y minúsculas, se cambia el modo en que se muestra la salida. Por ejemplo, `MONTH` muestra el nombre del mes en mayúsculas, `Month` lo muestra sólo con la inicial en mayúscula y `month` lo muestra todo en minúsculas. Cualquier carácter que se especifique en el formato y que no coincida con ningún patrón, se copia en la salida del mismo modo en que está escrito. A continuación se muestran algunos ejemplos:

```
SELECT TO_CHAR(CURRENT_TIMESTAMP, 'HH12 horas MI m. SS seg.');
SELECT TO_CHAR(CURRENT_DATE, 'Day, dd of month, yyyy');
SELECT TO_NUMBER('-12,454.8','S99,999.9');
```

Las funciones de fecha más habituales son las siguientes:

| | |
|---|---|
| `CURRENT_DATE` | Función del estándar que devuelve la fecha actual (el resultado es de tipo `DATE`). |
| `CURRENT_TIME` | Función del estándar que devuelve la hora actual (el resultado es de tipo `TIME`). |
| `CURRENT_TIMESTAMP` | Función del estándar que devuelve la fecha y hora actuales (el resultado es de tipo `TIMESTAMP`). |
| `EXTRACT(campo FROM dato)` | Función del estándar que devuelve la parte del `dato` (fecha u hora) indicada por `campo`. El resultado es de tipo `DOUBLE PRECISION`. En `campo` se pueden especificar las siguientes partes:<br>`day`: día del mes (1:31)<br>`dow`: día de la semana (0:6 empezando en domingo)<br>`doy`: día del año (1:366)<br>`week`: semana del año<br>`month`: mes del año (1:12)<br>`quarter`: trimestre del año (1:4)<br>`year`: año<br>`hour`: hora<br>`minute`: minutos<br>`second`: segundos |

A continuación se muestran algunos ejemplos de uso de estas funciones:

```
SELECT CURRENT_TIMESTAMP;
SELECT 365 - EXTRACT(DOY FROM CURRENT_DATE) AS dias_faltan;
SELECT EXTRACT(WEEK FROM TO_DATE('24/09/2008','dd/mm/yyyy'));
```

Para sumar o restar días a una fecha se utilizan los operadores `+` y `-`. Por ejemplo, para sumar siete días a la fecha actual se escribe: `CURRENT_DATE+7`.

### 4.5.7. Función CASE

Los lenguajes de programación procedurales suelen tener sentencias condicionales: si una condición es cierta entonces se realiza una acción, en caso contrario se realiza otra acción distinta. SQL no es un lenguaje procedural; sin embargo, permite un control condicional sobre los datos devueltos en una consulta, mediante la función `CASE`.

A continuación se muestra un ejemplo que servirá para explicar el modo de uso de esta función:

```
SELECT codart, precio,
       CASE WHEN stock > 500 THEN precio*0.8
            WHEN stock BETWEEN 200 AND 500 THEN precio*0.9
            ELSE precio
       END AS precio_con_descuento
FROM   articulos;
```

Esta sentencia muestra, para cada artículo, su código, su precio y un precio con descuento que se obtiene en función de su stock: si el stock es superior a 500 unidades, el descuento es del 20 % (se multiplica el precio por 0.8), si el stock está entre las 200 y las 500 unidades, el descuento es del 10 % (se multiplica el precio por 0.9) y si no, el precio se mantiene sin descuento. La columna con el precio de descuento se renombra (`precio_con_descuento`). La función `CASE` termina con `END` y puede tener tantas cláusulas `WHEN ... THEN` como se precise.

### 4.5.8. Funciones COALESCE y NULLIF

La función `COALESCE` devuelve el primero de sus parámetros que es no nulo. La función `NULLIF` devuelve un nulo si `valor1` y `valor2` son iguales; si no, devuelve `valor1`. La sintaxis de estas funciones es la siguiente:

```
COALESCE( valor [, ...] )
NULLIF( valor1, valor2 )
```

Ambas funciones se transforman internamente en expresiones equivalentes con la función `CASE`.

Por ejemplo, la siguiente sentencia:

```
SELECT codart, descrip,
       COALESCE(stock, stock_min, -1)
FROM   articulos;
```

es equivalente a esta otra:

```
SELECT codart, descrip,
       CASE WHEN stock IS NOT NULL THEN stock
            WHEN stock_min IS NOT NULL THEN stock_min
            ELSE -1 END
FROM   articulos;
```

Del mismo modo, la siguiente sentencia:

```
SELECT codart, descrip,
       NULLIF(stock, stock_min)
FROM   articulos;
```

es equivalente a esta otra:

```
SELECT codart, descrip,
       CASE WHEN stock=stock_min THEN NULL
            ELSE stock END
FROM   articulos;
```

Hay que tener siempre mucha precaución con las columnas que aceptan nulos y tratarlos adecuadamente cuando se deba hacer alguna restricción (`WHERE`) sobre dicha columna.

## 4.5.9. Ejemplos

**Ejemplo 4.1** *Se quiere obtener un listado con el código y la fecha de las facturas del año pasado que pertenecen a clientes cuyo código está entre el 50 y el 80. El resultado debe aparecer ordenado por la fecha, descendentemente.*

Al consultar la descripción de la tabla de `FACTURAS` puede verse que la columna fecha es de tipo `DATE`. Por lo tanto, para obtener las facturas del año pasado se debe obtener el año en curso (`CURRENT_DATE`) y quedarse con aquellas cuyo año es una unidad menor. El año de una fecha se obtiene utilizando la función `EXTRACT` tal y como se muestra a continuación.

```
SELECT codfac, fecha
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1
AND    codcli BETWEEN 50 AND 80
ORDER BY fecha DESC;
```

**Ejemplo 4.2** *Mostrar la fecha actual en palabras.*

```
SELECT TO_CHAR(CURRENT_DATE,'Day, dd of month of yyyy');
```

Al ejecutar esta sentencia se observa que quedan huecos demasiado grandes entre algunas palabras:

```
Sunday   , 20 of july      of 2008
```

Esto es así porque para la palabra del día de la semana y la palabra del mes se está dejando el espacio necesario para mostrar la palabra más larga que puede ir en ese lugar. Si se desea eliminar los blancos innecesarios se debe hacer uso de la función `RTRIM`.

```
SELECT RTRIM(TO_CHAR(CURRENT_DATE, 'Day')) ||
       RTRIM(TO_CHAR(CURRENT_DATE, ', dd of month')) ||
       TO_CHAR(CURRENT_DATE, ' of yyyy');

Sunday, 20 of july of 2008
```

**Ejemplo 4.3** *Se quiere obtener un listado con los códigos de los vendedores que han hecho ventas al cliente cuyo código es el 54.*

La información que se solicita se extrae de la tabla de `FACTURAS`: el código de vendedor de las facturas de dicho cliente. Puesto que el cliente puede tener varias facturas con el mismo vendedor (`codven` no es clave primaria ni clave alternativa en esta tabla), se debe utilizar el modificador `DISTINCT`.

```
SELECT DISTINCT codven
FROM   facturas
WHERE  codcli = 54;
```

Es muy importante saber de antemano cuándo se debe utilizar el modificador `DISTINCT`.

## 4.6. Operaciones sobre conjuntos de filas

En el apartado anterior se han presentado algunos de los operadores y de las funciones que se pueden utilizar en las cláusulas `SELECT` y `WHERE` de la sentencia `SELECT`. Mediante estos operadores y funciones construimos expresiones *a nivel de fila*. Por ejemplo, en la siguiente sentencia:

```
SELECT DISTINCT EXTRACT(month FROM fecha) AS meses
FROM   facturas
WHERE  codcli IN (45, 54, 87, 102)
AND    EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

se parte de la tabla `FACTURAS` y se seleccionan las filas que cumplen la condición de la cláusula `WHERE`. A continuación, se toma el valor de la fecha de cada fila seleccionada, se extrae el mes y se muestra éste sin repeticiones.

En este apartado se muestra cómo se pueden realizar operaciones *a nivel de columna*, teniendo en cuenta todas las filas de una tabla (sin cláusula `WHERE`) o bien teniendo en cuenta sólo algunas de ellas (con cláusula `WHERE`). Además, se muestra cómo las funciones de columna se pueden aplicar sobre grupos de filas cuando se hace uso de la cláusula `GROUP BY`. Este uso se hace necesario cuando los cálculos a realizar no son sobre todas las filas de una tabla o sobre un subconjunto, sino que se deben realizar repetidamente para distintos grupos de filas.

### 4.6.1. Funciones de columna

En ocasiones es necesario contar datos: ¿cuántos clientes hay en Castellón? O también hacer cálculos sobre ellos: ¿a cuánto asciende el IVA cobrado en la factura 3752? SQL proporciona una serie de funciones que se pueden utilizar en la cláusula `SELECT` y que actúan sobre los valores de las columnas para realizar diversas operaciones como, por ejemplo, sumarlos u obtener el valor máximo o el valor medio, entre otros. Las funciones de columna más habituales son las que se muestran a continuación:

| | |
|---|---|
| `COUNT(*)` | Cuenta filas. |
| `COUNT(columna)` | Cuenta valores no nulos. |
| `SUM(columna)` | Suma los valores de la columna. |
| `MAX(columna)` | Obtiene el valor máximo de la columna. |
| `MIN(columna)` | Obtiene el valor mínimo de la columna. |
| `AVG(columna)` | Obtiene la media de los valores de la columna. |

Si no se realiza ninguna restricción en la cláusula `WHERE` de una sentencia `SELECT` que utiliza funciones de columna, éstas se aplican sobre todas las filas de la tabla especificada en la cláusula `FROM`. Sin embargo, cuando se realiza una restricción mediante `WHERE`, las funciones se aplican sólo sobre las filas que la restricción ha seleccionado.

A continuación, se muestran algunos ejemplos:

```
-- cantidad media por línea de factura
SELECT AVG(cant)
FROM   lineas_fac;

-- cantidad media por línea de factura del artículo
SELECT AVG(cant)
FROM   lineas_fac
WHERE  codart = 'TLFXK2';

-- se puede hacer varios cálculos a la vez
SELECT SUM(cant) AS suma, COUNT(*) AS lineas
FROM   lineas_fac;
```

La función `COUNT()` realiza operaciones distintas dependiendo de su argumento:

| | |
|---|---|
| `COUNT(*)` | Cuenta filas. |
| `COUNT(columna)` | Cuenta el número de valores no nulos en la columna. |
| `COUNT(DISTINCT columna)` | Cuenta el número de valores distintos y no nulos en la columna. |

A continuación, se muestra su uso mediante un ejemplo. Se ha creado una tabla `P` que contiene los datos de una serie de piezas:

```
SELECT * FROM P;

pnum |  pnombre   |   color    | peso |   ciudad
-----+------------+------------+------+-----------
P1   | tuerca     | verde      |   12 | París
P2   | perno      | rojo       |      | Londres
P3   | birlo      | azul       |   17 | Roma
P4   | birlo      | rojo       |   14 | Londres
P5   | leva       |            |   12 | París
P6   | engrane    | rojo       |   19 | París
```

y se ha ejecutado la siguiente sentencia:

```
SELECT COUNT(*) AS cuenta1, COUNT(color) AS cuenta2,
       COUNT(DISTINCT color) AS cuenta3
FROM   P;
```

El resultado de ejecutarla será el siguiente:

```
 cuenta1 | cuenta2 | cuenta3
---------+---------+---------
       6 |       5 |       3
```

A la vista de los resultados se puede decir que `cuenta1` contiene el número de piezas, `cuenta2` contiene el número de piezas con color y `cuenta3` contiene el número de colores de los que hay piezas.

Las funciones de columna (`SUM, MAX, MIN, AVG`) ignoran los nulos, es decir, los nulos no son tenidos en cuenta en los cálculos. Según esto, se plantea la siguiente pregunta: ¿coincidirá siempre el valor de `media1` y `media2` al ejecutar la siguiente sentencia?

```
SELECT AVG(dto) AS media1, SUM(dto)/COUNT(*) AS media2
FROM   lineas_fac;
```

La respuesta es negativa, ya que en `media1` se devuelve el valor medio de los descuentos no nulos, mientras que en `media2` lo que se devuelve es el valor medio de los descuentos (interpretándose los descuentos nulos como el descuento cero).

Como se ha visto, la función `AVG` calcula la media de los valores no nulos de una columna. Si la tabla de la cláusula `FROM` es la de artículos, la media es por artículo; si la tabla de la cláusula `FROM` es la de facturas, la media es por factura. Cuando se quiere calcular otro tipo de media se debe hacer el cálculo mediante un cociente. Por ejemplo, el número medio de facturas *por mes* durante el año pasado se obtiene dividiendo el número de facturas del año pasado entre doce meses:

```
SELECT COUNT(*)/12 AS media_mensual
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

Es importante tener en cuenta que la función `COUNT` devuelve un entero y que las operaciones entre enteros devuelven resultados enteros. Es decir, la operación `SELECT 2/4;` devuelve el resultado cero. Por lo tanto, es conveniente multiplicar uno de los operandos por `1.0` para asegurarse de que se opera con números reales. En este caso, será necesario redondear los decimales del resultado a lo que sea preciso:

```
SELECT ROUND(COUNT(*)*1.0/12,2) AS media_mensual
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

### 4.6.2. Cláusula `GROUP BY`

La cláusula `GROUP BY` *forma grupos* con las filas que tienen en común los valores de una o varias columnas. Sobre cada grupo se pueden aplicar las funciones de columna que se han estado utilizando hasta ahora (`SUM, MAX, MIN, AVG, COUNT`), que pasan a denominarse *funciones de grupo*. Estas funciones, utilizadas en la cláusula `SELECT`, se aplican una vez para cada grupo.

La siguiente sentencia cuenta cuántas facturas tiene cada cliente el año pasado:

```
SELECT codcli, COUNT(*)
FROM   facturas
WHERE  EXTRACT(year FROM fecha) =
       EXTRACT(year FROM CURENT_DATE)-1
GROUP  BY codcli;
```

El modo en que se ejecuta la sentencia se explica a continuación. Se toma la tabla de facturas (`FROM`) y se seleccionan las filas que cumplen la restricción (`WHERE`). A continuación, las facturas se separan en grupos, de modo que en un mismo grupo sólo hay facturas de un mismo cliente (`GROUP BY codcli`), con lo cual hay tantos grupos como clientes hay con facturas del año pasado. Finalmente, de cada grupo se muestra el código del cliente y el número de facturas que hay en el grupo (son las facturas de ese cliente): `COUNT(*)`.

### 4.6.3. Cláusula HAVING

En la cláusula HAVING, que puede aparecer tras GROUP BY, se utilizan las funciones de grupo para hacer restricciones sobre los grupos que se han formado. La sintaxis de la sentencia SELECT, tal y como se ha visto hasta el momento, es la siguiente:

```
SELECT  [ DISTINCT ] { * | columna [ , columna ] }
FROM    tabla
[ WHERE condición_de_búsqueda ]
[ GROUP BY columna [, columna ]
[ HAVING condición_para_el_grupo ] ]
[ ORDER BY columna [ ASC | DESC ]
           [,columna [ ASC | DESC ] ];
```

En las consultas que utilizan GROUP BY se obtiene una fila por cada uno de los grupos producidos. Para ejecutar la cláusula GROUP BY se parte de las filas de la tabla que cumplen el predicado establecido en la cláusula WHERE y se agrupan en función de los valores comunes en la columna o columnas especificadas. Mediante la cláusula HAVING se realiza una restricción sobre los grupos obtenidos por la cláusula GROUP BY, y se seleccionan aquellos que cumplen el predicado establecido en la condición.

Evidentemente, en la condición de la cláusula HAVING sólo pueden aparecer restricciones sobre columnas por las que se ha agrupado y también funciones de grupo sobre cualquier otra columna de la tabla. Lo mismo sucede en la cláusula SELECT: sólo es posible especificar de manera directa columnas que aparecen en la cláusula GROUP BY y también funciones de grupo sobre cualquier otra columna. Cuando en las cláusulas SELECT o HAVING aparecen columnas que no se han especificado en la cláusula GROUP BY y que tampoco están afectadas por una función de grupo, se produce un error.

### 4.6.4. Ejemplos

**Ejemplo 4.4** *Se quiere obtener el importe medio por factura, sin tener en cuenta los descuentos ni el IVA.*

El importe medio por factura se calcula obteniendo primero la suma del importe de todas las facturas y dividiendo después el resultado entre el número total de facturas. La suma del importe de todas las facturas se obtiene sumando el importe de todas las líneas de factura. El importe de cada línea se calcula multiplicando el número de unidades pedidas (cant) por el precio unitario (precio).

Por lo tanto, la solución a este ejercicio es la siguiente:

```
SELECT ROUND(SUM(cant*precio)/COUNT(DISTINCT codfac),2)
       AS importe_medio
FROM   lineas_fac;
```

Se ha redondeado a dos decimales porque el resultado es una cantidad en euros.

**Ejemplo 4.5** *Se quiere obtener la fecha de la primera factura del cliente cuyo código es el 210, la fecha de su última factura (la más reciente) y el número de días que han pasado entre ambas facturas.*

Como se ha comentado antes, algunas funciones de columna se pueden utilizar también sobre las fechas. En general, las funciones `MIN` y `MAX` pueden usarse sobre todo aquel tipo de datos en el que haya definida una ordenación: tipos numéricos, cadenas y fechas.

Ambas funciones sirven, por lo tanto, para obtener la fecha de la primera y de la última factura. Restando ambas fechas se obtiene el número de días que hay entre ambas.

```
SELECT MIN(fecha) AS primera, MAX(fecha) AS ultima,
       MAX(fecha) - MIN(fecha) AS dias
FROM   facturas
WHERE  codcli = 210;
```

**Ejemplo 4.6** *Se quiere obtener un listado con los clientes que tienen más de cinco facturas con 18 % de IVA, indicando cuántas de ellas tiene cada uno.*

Para resolver este ejercicio se deben tomar las facturas (tabla `FACTURAS`) y seleccionar aquellas con 18 % de IVA (`WHERE`). A continuación, se debe agrupar las facturas (`GROUP BY`) de manera que haya un grupo para cada cliente (columna `codcli`). Una vez formados los grupos, se deben seleccionar aquellos que contengan más de cinco facturas (`HAVING`). Por último, se debe mostrar (`SELECT`) el código de cada cliente y su número de facturas.

```
SELECT codcli, COUNT(*) AS facturas
FROM   facturas
WHERE  iva = 18
GROUP BY codcli
HAVING COUNT(*) > 5;
```

**Ejemplo 4.7** *Se quiere obtener un listado con el número de facturas que hay en cada año, de modo que aparezca primero el año con más facturas. Además, para cada año se debe mostrar el número de clientes que han hecho compras y en cuántos días del año se han realizado éstas.*

```
SELECT EXTRACT(year FROM fecha) AS año,
       COUNT(*) AS nfacturas,
       COUNT(DISTINCT codcli) AS nclientes,
       COUNT(DISTINCT codven) AS nvendedores,
       COUNT(DISTINCT fecha) AS ndias
FROM   facturas
```

```
GROUP BY EXTRACT(year FROM fecha)
ORDER BY nfacturas DESC;
-- nfacturas es el nombre que se ha dado a COUNT(*)
```

Como se ve en el ejemplo, es posible utilizar expresiones en la cláusula `GROUP BY`. El ejemplo también muestra cómo se puede hacer referencia a los nombres con que se renombran las expresiones del `SELECT`, en la cláusula `ORDER BY`. Esto es así porque la cláusula `ORDER BY` es la única que se ejecuta tras el `SELECT`.

**Ejemplo 4.8** *De los clientes cuyo código está entre el 240 y el 250, mostrar el número de facturas que cada uno tiene con cada IVA distinto.*

```
SELECT codcli, COALESCE(iva,0) AS iva, COUNT(*) AS facturas
FROM   facturas
WHERE  codcli BETWEEN 240 AND 250
GROUP BY codcli, COALESCE(iva,0);
```

Para resolver el ejercicio, se han agrupado las facturas teniendo en cuenta dos criterios: el cliente y el IVA. De este modo, quedan en el mismo grupo las facturas que son de un mismo cliente y con un mismo tipo de IVA. Puesto que en la base de datos con que se trabaja se debe interpretar el IVA nulo como cero, se ha utilizado la función `COALESCE`. Si no se hubiera hecho esto, las facturas de cada cliente con IVA nulo habrían dado lugar a un nuevo grupo (distinto del de IVA cero), ya que la cláusula `GROUP BY` no ignora los nulos sino que los toma como si fueran todos un mismo valor.

### 4.6.5. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan agrupaciones:

- Cuando se utilizan funciones de grupo en la cláusula `SELECT` sin que haya `GROUP BY`, el resultado de ejecutar la consulta tiene una sola fila.
- A diferencia del resto de funciones que proporciona SQL, las funciones de grupo sólo se utilizan en las cláusulas `SELECT` y `HAVING`, nunca en la cláusula `WHERE`.
- La sentencia `SELECT` tiene dos cláusulas para realizar restricciones: `WHERE` y `HAVING`. Es muy importante saber situar cada restricción en su lugar: las restricciones que se deben realizar a nivel de filas, se sitúan en la cláusula `WHERE`; las restricciones que se deben realizar sobre grupos (normalmente involucran funciones de grupo), se sitúan en la cláusula `HAVING`.

- El modificador `DISTINCT` puede ser necesario en la cláusula `SELECT` de una sentencia que tiene `GROUP BY` sólo cuando las columnas que se muestren en la cláusula `SELECT` no sean todas las que aparecen en la cláusula `GROUP BY`.
- Una vez formados los grupos mediante la cláusula `GROUP BY` (son grupos de filas, no hay que olvidarlo), del contenido de cada grupo sólo es posible conocer el valor de las columnas por las que se ha agrupado (ya que dentro del grupo, todas las filas tienen dichos valores en común), por lo que sólo estas columnas son las que pueden aparecer, directamente, en las cláusulas `SELECT` y `HAVING`. Además, en estas cláusulas, se pueden incluir funciones de grupo que actúen sobre las columnas que no aparecen en la cláusula `GROUP BY`.

## 4.7. Subconsultas

Una subconsulta es una sentencia `SELECT` anidada en otra sentencia SQL, que puede ser otra `SELECT` o bien cualquier sentencia de manejo de datos (`INSERT, UPDATE, DELETE`). Las subconsultas pueden anidarse unas dentro de otras tanto como sea necesario (cada SGBD puede tener un nivel máximo de anidamiento, que difícilmente se alcanzará). En este apartado se muestra cómo el uso de subconsultas en las cláusulas `WHERE` y `HAVING` otorga mayor potencia para la realización de restricciones. Además, en este apartado se introduce el uso de subconsultas en la cláusula `FROM`.

### 4.7.1. Subconsultas en la cláusula `WHERE`

La cláusula `WHERE` se utiliza para realizar restricciones a nivel de filas. El predicado que se evalúa para realizar una restricción está formado por comparaciones unidas por los operadores `AND/OR`. Cada comparación involucra dos operandos que pueden ser:

(a) Dos columnas de la tabla sobre la que se realiza la consulta.

```
-- artículos cuyo stock es el mínimo deseado
SELECT *
FROM   articulos
WHERE  stock = stock_min;
```

(b) Una columna de la tabla de la consulta y una constante.

```
-- artículos cuya descripción empieza como se indica
SELECT *
FROM   articulos
WHERE  UPPER(descrip) LIKE 'PROLONG%';
```

(c) Una columna o una constante y una subconsulta sobre alguna tabla de la base de datos.

```
-- artículos vendidos con descuento mayor del 45%
SELECT *
FROM   articulos
WHERE  codart IN ( SELECT codart FROM lineas_fac
                   WHERE dto > 45 );
```

Además de los dos operandos, cada comparación se realiza con un operador. Hay una serie de operadores que se pueden utilizar con las subconsultas para establecer predicados en las restricciones. Son los que se muestran a continuación:

`expresión operador ( subconsulta )`
: En este predicado la subconsulta debe devolver un solo valor (una fila con una columna). El predicado se evalúa a verdadero si la comparación indicada por el `operador` (`=, <>, >, <, >=, <=`), entre el resultado de la `expresión` y el de la subconsulta, es verdadero. Si la subconsulta devuelve más de un valor (una columna con varias filas o más de una columna), se produce un error de ejecución.

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto = ( SELECT MAX(dto) FROM facturas );
```

`(expr1, expr2, ...) operador ( subconsulta )`
: En un predicado de este tipo, la subconsulta debe devolver una sola fila y tantas columnas como las especificadas entre paréntesis a la izquierda del `operador` (`=, <>, >, <, >=, <=`).

  Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara, utilizando el `operador`, con la fila que devuelve la subconsulta.

  El predicado se evalúa a verdadero si el resultado de la comparación es verdadero para la fila devuelta por la subconsulta. En caso contrario, se evalúa a falso. Si la subconsulta no devuelve ninguna fila, se evalúa a nulo.[2]

  Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

[2]Hay que tener en cuenta que una restricción se cumple si el resultado de su predicado es verdadero; si el predicado es falso o nulo, se considera que la restricción no se cumple.

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

```
-- facturas con descuento máximo e IVA máximo
SELECT *
FROM   facturas
WHERE  (dto, iva) =
       ( SELECT MAX(dto), MAX(iva) FROM facturas );
```

`expresión IN ( subconsulta )`

El operador `IN` ya ha sido utilizado anteriormente, especificando una lista de valores entre paréntesis. Otro modo de especificar esta lista de valores es incluyendo una subconsulta que devuelva una sola columna. En este caso, el predicado se evalúa a verdadero si el resultado de la `expresión` es igual a alguno de los valores de la columna devuelta por la subconsulta.

El predicado se evalúa a falso si no se encuentra ningún valor en la subconsulta que sea igual a la `expresión`; cuando la subconsulta no devuelve ninguna fila, también se evalúa a falso.

Si el resultado de la `expresión` es un nulo, o ninguno de los valores de la subconsulta es igual a la `expresión` y la subconsulta ha devuelto algún nulo, el predicado se evalúa a nulo.

```
-- pueblos en donde hay algún cliente
SELECT codpue, nombre
FROM   pueblos
WHERE  codpue IN ( SELECT codpue FROM clientes);
```

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

En este predicado la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador `IN`.

Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, una a una.

El predicado se evalúa a verdadero si se encuentra alguna fila igual en la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila).

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador `IN`, el predicado se evalúa a nulo.

```
-- clientes que han comprado en algún mes en
-- que ha comprado el cliente con código 282
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( EXTRACT(month FROM fecha),
         EXTRACT(year FROM fecha) )
       IN ( SELECT EXTRACT(month FROM fecha),
                   EXTRACT(year FROM fecha)
            FROM   facturas
            WHERE  codcli = 282);
```

`expresión NOT IN ( subconsulta )`

Cuando `IN` va negado, el predicado se evalúa a verdadero si la `expresión` es distinta de todos los valores de la columna devuelta por la subconsulta.

También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila. Si se encuentra algún valor igual a la `expresión`, se evalúa a falso.

Si el resultado de la `expresión` es un nulo, o si la subconsulta devuelve algún nulo y valores distintos a la `expresión`, el predicado se evalúa a nulo.

```
-- número de clientes que no tienen facturas
SELECT COUNT(*)
FROM   clientes
WHERE  codcli NOT IN ( SELECT codcli
                       FROM   facturas
                       WHERE  codcli IS NOT NULL );
```

Nótese que en el ejemplo se ha incluido la restricción `codcli IS NOT NULL` en la subconsulta porque la columna `FACTURAS.codcli` acepta nulos. Un nulo en esta columna haría que el predicado `NOT IN` se evaluara a nulo para todos los clientes de la consulta principal.

`(expr1, expr2, ...) NOT IN ( subconsulta )`

En este predicado, la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador `NOT IN`. Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si no se encuentra ninguna fila igual en la subconsulta. También se evalúa a verdadero si la subconsulta no devuelve ninguna fila. Si se encuentra alguna fila igual, se evalúa a falso.

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es

distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos y el resto de las filas son distintas de la fila de la izquierda del operador `NOT IN`, el predicado se evalúa a nulo.

```
-- clientes que no tienen facturas con IVA y dto
-- como tienen los clientes del rango especificado
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( COALESCE(iva,0), COALESCE(dto,0) )
       NOT IN ( SELECT COALESCE(iva,0), COALESCE(dto,0)
                FROM   facturas
                WHERE  codcli BETWEEN 171 AND 174);
```

`expresión operador ANY ( subconsulta )`

En este uso de `ANY` la subconsulta debe devolver una sola columna. El `operador` es una comparación (`=`, `<>`, `>`, `<`, `>=`, `<=`).

El predicado se evalúa a verdadero si la comparación establecida por el `operador` es verdadera para alguno de los valores de la columna devuelta por la subconsulta. En caso contrario se evalúa a falso.

```
-- facturas con IVA como los de las facturas sin dto
SELECT *
FROM   facturas
WHERE  iva = ANY( SELECT iva
                  FROM facturas
                  WHERE COALESCE(dto,0) = 0 );
```

Si la subconsulta no devuelve ninguna fila, devuelve falso. Si ninguno de los valores de la subconsulta coincide con la expresión de la izquierda del operador y en la subconsulta se ha devuelto algún nulo, se evalúa a nulo.

En lugar de `ANY` puede aparecer `SOME`, son sinónimos. El operador `IN` es equivalente a `= ANY`.

`(expr1, expr2, ...) operador ANY ( subconsulta )`

En este uso de `ANY` la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del operador. Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para alguna de las filas devueltas por la subconsulta. En caso contrario se evalúa a falso (incluso si la subconsulta no devuelve ninguna fila).

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser falso (será verdadero o nulo).

```
-- clientes que han comprado algún mes
-- en que ha comprado el cliente especificado
SELECT DISTINCT codcli
FROM   facturas
WHERE  ( EXTRACT(month FROM fecha),
         EXTRACT(year FROM fecha) )
       = ANY( SELECT EXTRACT(month FROM fecha),
                     EXTRACT(year FROM fecha)
              FROM   facturas
              WHERE  codcli = 282);
```

En lugar de `ANY` puede aparecer `SOME`.

`expresión operador ALL ( subconsulta )`

En este uso de `ALL` la subconsulta debe devolver una sola columna. El `operador` es una comparación (`=`, `<>`, `>`, `<`, `>=`, `<=`).

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todos los valores de la columna devuelta por la subconsulta. También se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila. En caso contrario se evalúa a falso. Si la subconsulta devuelve algún nulo, el predicado se evalúa a nulo

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto >= ALL ( SELECT COALESCE(dto,0)
                    FROM facturas );
```

Nótese que, si en el ejemplo anterior, la subconsulta no utiliza `COALESCE` para convertir los descuentos nulos en descuentos cero, la consulta principal no devuelve ninguna fila porque al haber nulos en el resultado de la subconsulta, el predicado se evalúa a nulo.

El operador `NOT IN` es equivalente a `<>ALL`.

`(expr1, expr2, ...) operador ALL ( subconsulta )`

En este uso de `ALL`, la subconsulta debe devolver tantas columnas como las especificadas entre paréntesis a la izquierda del `operador`.

Las expresiones de la izquierda `expr1, expr2, ...` se evalúan y la fila que forman se compara con las filas de la subconsulta, fila a fila.

El predicado se evalúa a verdadero si la comparación establecida por el operador es verdadera para todas las filas devueltas por la subconsulta; cuando la subconsulta no devuelve ninguna fila también se evalúa a verdadero. En caso contrario se evalúa a falso.

Dos filas se consideran iguales si los atributos correspondientes son iguales y no nulos en ambas; se consideran distintas si algún atributo es distinto en ambas filas y no nulo. En cualquier otro caso, el resultado del predicado es nulo.

Si la subconsulta devuelve alguna fila de nulos, el predicado no podrá ser verdadero (será falso o nulo).

```
-- muestra los datos del cliente especificado si
-- siempre ha comprado sin descuento y con 18% de IVA
SELECT *
FROM   clientes
WHERE  codcli = 162
AND    ( 18, 0 ) =
       ALL (SELECT COALESCE(iva,0), COALESCE(dto,0)
            FROM   facturas
            WHERE  codcli = 162 );
```

Cuando se utilizan subconsultas en predicados, el SGBD no obtiene el resultado completo de la subconsulta, a menos que sea necesario. Lo que hace es ir obteniendo filas de la subconsulta hasta que es capaz de determinar si el predicado es verdadero.

### 4.7.2. Subconsultas en la cláusula `HAVING`

La cláusula `HAVING` permite hacer restricciones sobre grupos y necesariamente va precedida de una cláusula `GROUP BY`. Para hacer este tipo de restricciones también es posible incluir subconsultas cuando sea necesario.

La siguiente consulta obtiene el código del pueblo que tiene más clientes:

```
SELECT codpue
FROM   clientes
GROUP BY codpue
HAVING COUNT(*) >= ALL ( SELECT COUNT(*)
                         FROM   clientes
                         GROUP BY codpue );
```

En primer lugar se ejecuta la subconsulta, obteniéndose una columna de números en donde cada uno indica el número de clientes en cada pueblo. La

subconsulta se sustituye entonces por los valores de esta columna, por ejemplo:

```
SELECT codpue
FROM   clientes
GROUP BY codpue
HAVING COUNT(*) >= ALL (1,4,7,9,10);
```

Por último, se ejecuta la consulta principal. Para cada grupo se cuenta el número de clientes que tiene. Pasan la restricción del `HAVING` aquel o aquellos pueblos que en esa cuenta tienen el máximo valor.

### 4.7.3. Subconsultas en la cláusula `FROM`

También es posible incluir subconsultas en la cláusula `FROM`, aunque en este caso no se utilizan para construir predicados sino para realizar una consulta sobre la tabla que se obtiene como resultado de ejecutar otra consulta. Siempre que se utilice una subconsulta en el `FROM` se debe dar un nombre a la tabla resultado mediante la cláusula `AS`.

```
SELECT COUNT(*), MAX(ivat), MAX(dtot)
FROM   ( SELECT DISTINCT COALESCE(iva,0) AS ivat,
                         COALESCE(dto,0) AS dtot
           FROM   facturas ) AS t;
```

La consulta anterior cuenta las distintas combinaciones de IVA y descuento y muestra el valor máximo de éstos. Nótese que se han renombrado las columnas de la subconsulta para poder referenciarlas en la consulta principal. Esta consulta no se puede resolver si no es de este modo ya que `COUNT` no acepta una lista de columnas como argumento.

### 4.7.4. Ejemplos

**Ejemplo 4.9** *Se quiere obtener los datos completos del cliente al que pertenece la factura 5886.*

Para dar la respuesta podemos hacerlo en dos pasos, es decir, con dos consultas separadas:

```
SELECT codcli FROM facturas WHERE codfac = 5886;

 codcli
--------
    264

SELECT *
FROM   clientes
WHERE  codcli = 264;
```

Puesto que es posible anidar las sentencias SELECT para obtener el resultado con una sola consulta, una solución que obtiene el resultado en un solo paso es la siguiente:

```
SELECT *
FROM   clientes
WHERE  codcli = ( SELECT codcli
                  FROM facturas WHERE codfac = 5886 );
```

Se ha utilizado el operador de comparación = porque se sabe con certeza que la subconsulta devuelve un solo código de cliente, ya que la condición de búsqueda es de igualdad sobre la clave primaria de la tabla del FROM.

**Ejemplo 4.10** *Se quiere obtener los datos completos de los clientes que tienen facturas en agosto del año pasado. El resultado se debe mostrar ordenado por el nombre del cliente.*

De nuevo se puede dar la respuesta en dos pasos:

```
SELECT codcli FROM facturas
WHERE EXTRACT(month FROM fecha)=8
AND   EXTRACT(year FROM fecha) =
      EXTRACT(year FROM CURRENT_DATE)-1;

 codcli
--------
    105
     12
      .
      .
      .
    342
    309
    357

SELECT *
FROM   clientes
WHERE  codcli IN (105,12,...,342,309,357);
```

Se ha utilizado el operador IN porque la primera consulta devuelve varias filas. Esto debe saberse sin necesidad de probar la sentencia. Como esta vez no se seleccionan las facturas por una columna única (clave primaria o clave alternativa), es posible que se obtengan varias filas y por lo tanto se debe utilizar IN.

Tal y como se ha hecho en el ejemplo anterior, ambas sentencias pueden integrarse en una sola:

```
SELECT *
FROM   clientes
WHERE  codcli IN ( SELECT codcli FROM facturas
                   WHERE EXTRACT(month FROM fecha)=8
                   AND   EXTRACT(year FROM fecha) =
                         EXTRACT(year FROM CURRENT_DATE)-1 )
ORDER BY nombre;
```

### 4.7.5. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan subconsultas:

- Las subconsultas utilizadas en predicados del tipo `expresión operador (subconsulta)` o `(expr1, expr2, ...) operador (subconsulta)` deben devolver siempre una sola fila; en otro caso, se producirá un error. Si la subconsulta ha de devolver varias filas se debe utilizar `IN`, `NOT IN`, `operador ANY`, `operador ALL`.
- Es importante ser cuidadosos con las subconsultas que pueden devolver nulos. Una restricción se supera si el predicado se evalúa a verdadero; no se supera si se evalúa a falso o a nulo. Dos casos que no conviene olvidar son los siguientes:
  - `NOT IN` se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila; si la subconsulta devuelve un nulo/fila de nulos, se evalúa a nulo.
  - `operador ALL` se evalúa a verdadero cuando la subconsulta no devuelve ninguna fila; si la subconsulta devuelve un nulo/fila de nulos, se evalúa a nulo.
- Cuando se utilizan subconsultas en la cláusula `FROM` es preciso renombrar las columnas del `SELECT` de la subconsulta que son expresiones. De ese modo, será posible hacerles referencia en la consulta principal. Además, la tabla resultado de la subconsulta también se debe renombrar en el `FROM` de la consulta principal.

## 4.8. Consultas multitabla

En este apartado se muestra cómo hacer consultas que involucran a datos de varias tablas. Aunque mediante las subconsultas se ha conseguido realizar consultas de este tipo, aquí se verá que en ocasiones, es posible escribir consultas equivalentes que no hacen uso de subconsultas y que se ejecutan de modo más eficiente. El operador que se introduce es la *concatenación* (`JOIN`).

### 4.8.1. La concatenación: `JOIN`

La concatenación es una de las operaciones más útiles del lenguaje SQL. Esta operación permite combinar información de varias tablas sin necesidad de utilizar subconsultas para ello.

La concatenación natural (`NATURAL JOIN`) de dos tablas `R` y `S` obtiene como resultado una tabla cuyas filas son todas las filas de `R` concatenadas con todas las filas de `S` que en las columnas que se llaman igual tienen los mismos valores. Las columnas por las que se hace la concatenación aparecen una sola vez en el resultado.

La siguiente sentencia hace una concatenación natural de las tablas `FACTURAS` y `CLIENTES`. Ambas tablas tienen una columna con el mismo nombre, `codcli`, siendo `FACTURAS.codcli` una clave ajena a `CLIENTES.codcli` (clave primaria).

```
SELECT *
FROM   facturas NATURAL JOIN clientes;
```

Según la definición de la operación `NATURAL JOIN`, el resultado tendrá las siguientes columnas: `codfac, fecha, codven, iva, dto, codcli, nombre, direccion, codpostal, codpue`. En el resultado de la concatenación cada fila representa una factura que cuenta con sus datos (la cabecera) y los datos del cliente al que pertenece. Si alguna factura tiene `codcli` nulo, no aparece en el resultado de la concatenación puesto que no hay ningún cliente con el que pueda concatenarse.

Cambiando el contenido de la cláusula `SELECT`, cambia el resultado de la consulta. Por ejemplo:

```
SELECT DISTINCT codcli, nombre, direccion, codpostal, codpue
FROM   facturas NATURAL JOIN clientes;
```

Esta sentencia muestra los datos de los clientes que tienen facturas. Puesto que se ha hecho la concatenación, si hay clientes que no tienen facturas, no se obtienen en el resultado ya que no tienen ninguna factura con la que concatenarse.

A continuación, se desea modificar la sentencia anterior para que se obtenga también el nombre de la población del cliente. Se puede pensar que el nombre de la población se puede mostrar tras hacer una concatenación natural con la

tabla PUEBLOS. El objetivo es concatenar cada cliente con su población a través de la clave ajena codpue. Sin embargo, la concatenación natural no es útil en este caso porque las tablas PUEBLOS y CLIENTES tienen también otra columna que se llama igual: la columna nombre. CLIENTES.nombre contiene el nombre de cada cliente y PUEBLOS.nombre contiene el nombre de cada pueblo. Ambos nombres no significan lo mismo, por lo que la concatenación natural a través de ellas no permite obtener el resultado que se desea.

¿Qué se obtendrá como resultado al ejecutar la siguiente sentencia?

```
SELECT *
FROM   facturas NATURAL JOIN clientes NATURAL JOIN pueblos;
```

Se obtendrán las facturas de los clientes cuyo nombre completo coincide con el nombre de su pueblo, cosa poco probable que suceda.

Cuando se quiere concatenar varias tablas que tienen varios nombres de columnas en común y no todos han de utilizarse para realizar la concatenación, se puede disponer de la operación INNER JOIN, que permite especificar las columnas sobre las que hacer la operación mediante la cláusula USING.

```
SELECT DISTINCT codcli, clientes.nombre, codpue,
                pueblos.nombre
FROM   facturas INNER JOIN clientes USING (codcli)
                INNER JOIN pueblos  USING (codpue);
```

Nótese que, en la consulta anterior, algunas columnas van precedidas por el nombre de la tabla a la que pertenecen. Esto es necesario cuando hay columnas que se llaman igual en el resultado: se especifica el nombre de la tabla para evitar ambigüedades. Esto sucede cuando las tablas que se concatenan tienen nombres de columnas en común y la concatenación no se hace a través de ellas, como ha sucedido en el ejemplo con las columnas CLIENTES.nombre y PUEBLOS.nombre. En el resultado hay dos columnas nombre y, sin embargo, una sola columna codcli y una sola columna codpue (estas dos últimas aparecen sólo una vez porque las concatenaciones se han hecho a través de ellas).

En realidad, en SQL el nombre de cada columna está formado por el nombre de su tabla, un punto y el nombre de la columna (FACTURAS.iva, CLIENTES.nombre). Por comodidad, cuando no hay ambigüedad al referirse a una columna, se permite omitir el nombre de la tabla a la que pertenece, que es lo que se había estado haciendo hasta ahora en este capítulo.

Cuando las columnas por las que se hace la concatenación no se llaman igual en las dos tablas, se utiliza ON para especificar la condición de concatenación de ambas columnas, tal y como se ve en el siguiente ejemplo. En él se introduce también el uso de alias para las tablas, lo que permite no tener que escribir el nombre completo para referirse a sus columnas:

```
SELECT v.codven, v.nombre AS vendedor,
       j.codven AS codjefe, j.nombre AS jefe
```

```
FROM   vendedores AS v INNER JOIN vendedores AS j
       ON (v.codjefe=j.codven);
```

Esta sentencia obtiene el código y el nombre de cada vendedor, junto al código y el nombre del vendedor que es su jefe.

Es aconsejable utilizar siempre alias para las tablas cuando se hagan consultas multitabla, y utilizarlos para especificar todas las columnas, aunque no haya ambigüedad. Es una cuestión de estilo.

Ya que este tipo de concatenación (`INNER JOIN`) es el más habitual, se permite omitir la palabra `INNER` al especificarlo, tal y como se muestra en el siguiente ejemplo:

```
SELECT DISTINCT c.codcli, c.nombre, c.codpue, p.nombre
FROM   facturas AS f JOIN clientes AS c USING (codcli)
                     JOIN pueblos  AS p USING (codpue)
WHERE  COALESCE(f.iva,0) = 18
AND    COALESCE(f.dto,0) = 0;
```

Aunque la operación de `NATURAL JOIN` es la que originalmente se definió en el modelo relacional, su uso en SQL no es aconsejable puesto que la creación de nuevas columnas en tablas de la base de datos puede dar lugar a errores en las sentencias que las consultan, si estas nuevas columnas tienen el mismo nombre que otras columnas de otras tablas con las que se han de concatenar.

Es recomendable, al construir las concatenaciones, especificar las tablas en el mismo orden en el que aparecen en el diagrama referencial (figura 4.2). De este modo será más fácil depurar las sentencias, así como identificar qué hace cada una: en el resultado de una consulta escrita de este modo, cada fila representará lo mismo que representa cada fila de la primera tabla que aparezca en la cláusula `FROM` y en este resultado habrá, como mucho, tantas filas como filas hay en dicha tabla.

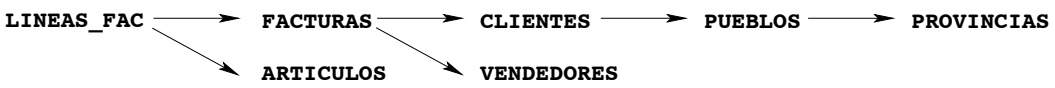


Figura 4.2: Diagrama referencial de la base de datos.

Hay un aspecto que todavía no se ha tenido en cuenta: los nulos en las columnas a través de las cuales se realizan las concatenaciones. Por ejemplo, si se quiere obtener un listado con las facturas del mes de diciembre del año pasado, donde aparezcan los nombres del cliente y del vendedor, se puede escribir la siguiente consulta:

```
SELECT f.codfac, f.fecha, f.codcli, c.nombre,
       f.codven, v.nombre
FROM   facturas AS f JOIN clientes AS c USING (codcli)
                     JOIN vendedores AS v USING (codven)
WHERE  EXTRACT(month FROM f.fecha) = 12
AND    EXTRACT(year FROM f.fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

De todas las facturas que hay en dicho mes, aparecen en el resultado sólo algunas. Esto es debido a que las columnas `FACTURAS.codcli` y `FACTURAS.codven` aceptan nulos. Las facturas con algún nulo en alguna de estas columnas son las que no aparecen en el resultado.

Para evitar estos problemas, se puede hacer uso de la operación `OUTER JOIN` con tres variantes: `LEFT, RIGHT, FULL`. Con `LEFT/RIGHT OUTER JOIN`, en el resultado se muestran todas las filas de la tabla de la izquierda/derecha; aquellas que no tienen nulos en la columna de concatenación, se concatenan con las filas de la otra tabla mediante `INNER JOIN`. Las filas de la tabla de la izquierda/derecha que tienen nulos en la columna de concatenación aparecen en el resultado concatenadas con una fila de nulos. Con `FULL OUTER JOIN` se hacen ambas operaciones: `LEFT OUTER JOIN` y `RIGHT OUTER JOIN`.

Teniendo en cuenta que, tanto `FACTURAS.codcli` como `FACTURAS.codven` aceptan nulos, el modo correcto de realizar la consulta en este último ejemplo será:

```
SELECT f.codfac, f.fecha, f.codcli, c.nombre,
       f.codven, v.nombre
FROM   facturas AS f
           LEFT OUTER JOIN clientes AS c USING (codcli)
           LEFT OUTER JOIN vendedores AS v USING (codven)
WHERE  EXTRACT(month FROM f.fecha) = 12
AND    EXTRACT(year FROM f.fecha) =
       EXTRACT(year FROM CURRENT_DATE)-1;
```

Como se ha visto, el `OUTER JOIN` tiene sentido cuando no se quiere perder filas en una concatenación en que una de las columnas que interviene acepta nulos. Otro caso en que esta operación tiene sentido es cuando las filas de una tabla no tienen filas para concatenarse en la otra tabla porque no son referenciadas por ninguna de ellas. Es el caso del siguiente ejemplo:

```
SELECT c.codcli, c.nombre, COUNT(f.codfac) AS nfacturas
FROM   facturas AS f
           RIGHT OUTER JOIN clientes AS c USING (codcli)
GROUP BY c.codcli, c.nombre
ORDER BY 3 DESC;
```

Esta sentencia obtiene un listado con todos los clientes de la tabla CLIENTES y el número de facturas que cada uno tiene. Si algún cliente no tiene ninguna factura (no es referenciado por ninguna fila de la tabla de FACTURAS), también aparecerá en el resultado y la cuenta del número de facturas será cero. Nótese que la cuenta del número de facturas se hace sobre la clave primaria de FACTURAS (COUNT(f.codfac)) ya que los clientes sin facturas tienen un nulo en esta columna tras la concatenación y las funciones de columna ignoran los nulos, por lo que la cuenta será cero.

### 4.8.2. Sintaxis original de la concatenación

En versiones anteriores del estándar de SQL la concatenación no se realizaba mediante JOIN, ya que esta operación no estaba implementada directamente. En el lenguaje teórico en el que se basa SQL, el álgebra relacional, la operación de concatenación sí existe, pero ya que no es una operación primitiva, no fue implementada en SQL en un principio. No es una operación primitiva porque se puede llevar a cabo mediante la combinación de otras dos operaciones: el producto cartesiano y la restricción. La restricción se lleva a cabo mediante la cláusula WHERE, que ya es conocida. El producto cartesiano se lleva a cabo separando las tablas involucradas por una coma en la cláusula FROM, tal y como se muestra a continuación:

```
SELECT *
FROM   facturas, clientes;
```

La sentencia anterior combina todas las filas de la tabla facturas con todas las filas de la tabla clientes. Si la primera tiene $n$ filas y la segunda tiene $m$ filas, el resultado tendrá $n \times m$ filas.

Para hacer la concatenación de cada factura con el cliente que la ha solicitado, se debe hacer una restricción: de las $n \times m$ filas hay que seleccionar aquellas en las que coinciden los valores de las columnas codcli.

```
SELECT *
FROM   facturas, clientes
WHERE  facturas.codcli = clientes.codcli;
```

La siguiente consulta, que utiliza el formato original para realizar las concatenaciones. Obtiene los datos de las facturas con 18 % de IVA y sin descuento, con el nombre del cliente:

```
SELECT facturas.codfac, facturas.fecha,
       facturas.codcli, clientes.nombre, facturas.codven
FROM   facturas, clientes
WHERE  facturas.codcli = clientes.codcli -- concatenación
AND    COALESCE(facturas.iva,0) = 18     -- restricción
AND    COALESCE(facturas.dto,0) = 0;     -- restricción
```

No hay que olvidar que la concatenación que se acaba de mostrar utiliza una sintaxis que ha quedado obsoleta en el estándar de SQL. La sintaxis del estándar actual es más aconsejable porque permite identificar más claramente qué son restricciones (aparecerán en el `WHERE`) y qué son condiciones de concatenación (aparecerán en el `FROM` con la palabra clave `JOIN`). Sin embargo, es importante conocer esta sintaxis porque todavía es muy habitual su uso.

### 4.8.3. Ejemplos

**Ejemplo 4.11** *Obtener los datos completos del cliente al que pertenece la factura 5886.*

Una versión que utiliza subconsultas es la siguiente:

```
SELECT *
FROM   clientes
WHERE  codcli = ( SELECT codcli
                  FROM facturas WHERE codfac = 5886 );
```

Una versión que utiliza `JOIN` es la siguiente:

```
SELECT c.*
FROM   facturas f JOIN clientes c USING (codcli)
WHERE  f.codfac = 5886;
```

**Ejemplo 4.12** *Obtener el código de las facturas en las que se ha pedido el artículo que tiene actualmente el precio más caro.*

Una versión en donde se utiliza el `JOIN` de subconsultas en el `FROM` es la siguiente:

```
SELECT DISTINCT l.codfac
FROM   lineas_fac AS l JOIN articulos AS a USING (codart)
       JOIN (SELECT MAX(precio) AS precio
             FROM articulos) AS t ON (a.precio = t.precio);
```

En la siguiente versión se utiliza la subconsulta para hacer una restricción.

```
SELECT DISTINCT l.codfac
FROM   lineas_fac AS l JOIN articulos AS a USING (codart)
WHERE  a.precio =  (SELECT MAX(precio)
                    FROM articulos) ;
```

A continuación se muestra una versión que utiliza sólo subconsultas:

```
SELECT DISTINCT codfac
FROM   lineas_fac
WHERE  codart IN
          (SELECT codart
           FROM   articulos
           WHERE  precio =
                    (SELECT MAX(precio) FROM articulos));
```

**Ejemplo 4.13** *Para cada vendedor de la provincia de Castellón, mostrar su nombre y el nombre de su jefe inmediato.*

```
SELECT v.codven, v.nombre AS vendedor, j.nombre AS jefe
FROM   vendedores AS v JOIN vendedores AS j
           ON (v.codjefe = j.codven)
       JOIN pueblos AS p ON (v.codpue = p.codpue)
WHERE  p.codpro = '12';
```

Nótese que ambas concatenaciones deben hacerse mediante `ON`: la primera porque las columnas de concatenación no tienen el mismo nombre, la segunda porque al concatenar con `PUEBLOS` hay dos columnas `codpue` en la tabla de la izquierda: `v.codpue` y `j.codven`.

### 4.8.4. Algunas cuestiones importantes

A continuación se plantean algunas cuestiones que es importante tener en cuenta cuando se realizan concatenaciones:

- Al hacer un `NATURAL JOIN` es importante fijarse muy bien en los nombres de las columnas de las tablas que participan en la operación. Como se sabe, mediante este operador se concatenan las filas de ambas tablas que en los atributos que tienen el mismo nombre tienen también los mismos valores. Por ejemplo, un `NATURAL JOIN` entre las tablas `PUEBLOS` y `CLIENTES` se realizará a través de las columnas `codpue` y `nombre`. El resultado, si contiene alguna fila, serán los datos de clientes que tienen como nombre el mismo nombre de su población. Si nuestro objetivo era realizar la concatenación a través de `codpue` podemos decir que el uso del `NATURAL JOIN` nos ha jugado una mala pasada. Concatenar filas por columnas no deseadas implica tener en cuenta más restricciones, con lo que los resultados obtenidos pueden no ser correctos. Es más aconsejable utilizar `INNER JOIN`, ya que pueden evitarse estos problemas al especificarse de manera explícita las columnas de concatenación.

- En la vida de una base de datos puede ocurrir que a una tabla se le deban añadir nuevas columnas para que pueda almacenar más información. Si esta tabla se ha utilizado para realizar algún `NATURAL JOIN` en alguna de las consultas de los programas de aplicación, hay que ser cuidadosos al escoger el nombre ya que si una nueva columna se llama igual que otra columna de la otra tabla participante en dicha operación, la concatenación que se hará ya no será la misma. Es posible evitar este tipo de problemas utilizando siempre `INNER JOIN` ya que éste requiere que se especifiquen las columnas por las que realizar la concatenación, y aunque se añadan nuevas columnas a las tablas, no cambiará la operación realizada por más que haya nuevas coincidencias de nombres en ambas tablas.

- Ordenar las tablas en el `FROM` tal y como aparecen en los diagramas referenciales ayuda a tener un mayor control de la consulta en todo momento: es posible saber si se ha olvidado incluir alguna tabla intermedia y es posible saber qué representa cada fila del resultado de la concatenación de todas las tablas implicadas. Además, será más fácil decidir qué incluir en la función `COUNT()` cuando sea necesaria, y también será más fácil determinar si en la proyección final (`SELECT`) es necesario el uso de `DISTINCT`.

## 4.9. Operadores de conjuntos

Los operadores de conjuntos del álgebra relacional son: el producto cartesiano, la unión, la intersección y la diferencia. El producto cartesiano se realiza en SQL especificando en la cláusula `FROM` las tablas involucradas en la operación, separadas por comas, tal y como se ha indicado anteriormente. A continuación se muestra cómo utilizar el resto de los operadores de conjuntos en las consultas en SQL.

La sintaxis para las uniones, intersecciones y diferencias es la siguiente:

```
sentencia_SELECT
UNION | INTERSECT | EXCEPT [ ALL ]
sentencia_SELECT
[ ORDER BY columna [ ASC | DESC ]
           [,columna [ ASC | DESC ] ];
```

Nótese que la cláusula `ORDER BY` sólo puede aparecer una vez en la consulta, al final de la misma. La ordenación se realizará sobre el resultado de la unión, intersección o diferencia.

Para poder utilizar cualquiera de estos tres nuevos operadores, las cabeceras de las sentencias `SELECT` involucradas deben devolver el mismo número de columnas, y las columnas correspondientes en ambas sentencias deberán ser del mismo tipo de datos.

### 4.9.1. Operador UNION

Este operador devuelve como resultado todas las filas que devuelve la primera sentencia `SELECT`, más aquellas filas de la segunda sentencia `SELECT` que no han sido ya devueltas por la primera. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave `ALL`. En este caso, si una fila aparece $m$ veces en la primera sentencia y $n$ veces en la segunda, en el resultado aparecerá $m + n$ veces.

Si se realizan varias uniones, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes o donde hay vendedores:

```
SELECT codpue FROM clientes
UNION
SELECT codpue FROM vendedores;
```

### 4.9.2. Operador INTERSECT

Este operador devuelve como resultado las filas que se encuentran tanto en el resultado de la primera sentencia `SELECT` como en el de la segunda sentencia `SELECT`. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave `ALL`. En este caso, si una misma fila aparece $m$ veces en la primera sentencia y $n$ veces en la segunda, en el resultado esta fila aparecerá $min(m, n)$ veces.

Si se realizan varias intersecciones, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto. La intersección tiene más prioridad, en el orden de evaluación, que la unión, es decir, `A UNION B INTERSECT C` se evalúa como `A UNION (B INTERSECT C)`.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes y también hay vendedores:

```
SELECT codpue FROM clientes
INTERSECT
SELECT codpue FROM vendedores;
```

### 4.9.3. Operador EXCEPT

Este operador devuelve como resultado las filas que se encuentran en el resultado de la primera sentencia SELECT y no se encuentran en el resultado de la segunda sentencia SELECT. En el resultado no se muestran duplicados.

Se puede evitar la eliminación de duplicados especificando la palabra clave ALL. En este caso, si una misma fila aparece $m$ veces en la primera sentencia y $n$ veces en la segunda, en el resultado esta fila aparecerá $max(m-n, 0)$ veces.

Si se realizan varias diferencias, éstas se evalúan de izquierda a derecha, a menos que se utilicen paréntesis para establecer un orden distinto. La diferencia tiene la misma prioridad, en el orden de evaluación, que la unión.

La siguiente sentencia muestra los códigos de las poblaciones donde hay clientes y no hay vendedores:

```
SELECT codpue FROM clientes
EXCEPT
SELECT codpue FROM vendedores;
```

La diferencia no es una operación conmutativa, mientras que el resto de los operadores de conjuntos sí lo son.

### 4.9.4. Sentencias equivalentes

En muchas ocasiones, una misma consulta de datos puede responderse mediante distintas sentencias SELECT que utilizan operadores diferentes. Cada una de ellas dará, por lo general, un tiempo de respuesta diferente, y se puede considerar que una es mejor que otra en este aspecto.

El que una sentencia sea mejor en unas circunstancias no garantiza que vaya a serlo siempre: puede que al evolucionar el estado de la base de datos, una sentencia que era la mejor, deje de serlo porque las tablas hayan cambiado de tamaño o se haya creado o eliminado algún índice.

Es por todo lo anterior, que se considera importante que, ante una consulta de datos, sea posible obtener varias sentencias alternativas. En este apartado se presentan algunas equivalencias entre operadores que se pueden utilizar para obtener sentencias equivalentes.

- Una concatenación es equivalente a una expresión con el operador IN y una subconsulta. Dependiendo del número de filas que obtenga la subconsulta, será más o menos eficiente que la concatenación con JOIN.
- Una restricción con dos comparaciones unidas por OR es equivalente a la unión de dos sentencias SELECT, y sitúa cada una de estas comparaciones en una sentencia distinta.
- Una restricción con dos comparaciones unidas por AND es equivalente a la intersección de dos sentencias SELECT, y sitúa cada una de estas comparaciones en una sentencia distinta.

- Una restricción con dos comparaciones unidas por `AND NOT` es equivalente a la diferencia de dos sentencias `SELECT`, y sitúa la primera comparación en la primera sentencia y la segunda comparación en la segunda sentencia (conviene recordar que esta operación no es conmutativa).
- El operador `NOT IN` puede dar resultados inesperados cuando la subconsulta devuelve algún nulo. En general, es más aconsejable trabajar con operadores en positivo (sin `NOT`) (en el ejemplo que se ofrece después se verá el porqué). Una restricción con el operador `NOT IN` y una subconsulta, es equivalente a una restricción con `IN` y una subconsulta con `EXCEPT`.

### 4.9.5. Ejemplos

**Ejemplo 4.14** *Obtener los datos de las poblaciones donde hay vendedores y no hay clientes.*

```
SELECT *
FROM   ( SELECT codpue FROM vendedores
         EXCEPT
         SELECT codpue FROM clientes ) AS t
       JOIN pueblos USING (codpue)
       JOIN provincias USING (codpro);
```

La tabla `t` contiene los códigos de las poblaciones en donde hay vendedores y no hay clientes. Tras concatenarla con `PUEBLOS` y `PROVINCIAS` se obtienen los datos completos de dichas poblaciones.

**Ejemplo 4.15** *¿Cuántos clientes hay que entre todas sus facturas no tienen ninguna con 18 % de IVA?*

La siguiente solución utiliza el operador `NOT IN`. Nótese que es preciso tener en cuenta dos restricciones: la primera es que en la subconsulta del `NOT IN` se debe evitar los nulos, y la segunda es que hay que asegurarse de que los clientes seleccionados hayan realizado alguna compra (deben tener alguna factura).

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli NOT IN ( SELECT codcli FROM facturas
                       WHERE  COALESCE(iva,0) = 18
                       AND    codcli IS NOT NULL )
AND    codcli IN (SELECT codcli FROM facturas);
```

Una sentencia equivalente sin NOT IN y que utiliza un operador de conjuntos, es la siguiente:

```
-- clientes con alguna factura
-- menos
-- clientes que tienen alguna con 18%
SELECT COUNT(*) AS clientes
FROM   ( SELECT codcli FROM facturas
         EXCEPT
         SELECT codcli FROM facturas
         WHERE  COALESCE(iva,0) = 18 ) AS t;
```

Trabajando con EXCEPT en lugar de NOT IN no es preciso preocuparse por los nulos en la clave ajena FACTURAS.codcli. Otra ventaja es que no aparecen en el resultado los clientes sin facturas. Además, suele suceder que las consultas así formuladas consiguen mejores tiempos de respuesta que las que utilizan NOT IN, quizá porque hay ciertas comprobaciones que se evitan.

## 4.10. Subconsultas correlacionadas

Una subconsulta correlacionada es una consulta anidada que contiene referencias a columnas de las tablas que se encuentran en el FROM de la consulta principal. Son lo que se denomina referencias externas.

Como ya se ha visto, las subconsultas dotan al lenguaje SQL de una gran potencia. Estas pueden utilizarse para hacer restricciones, tanto en la cláusula WHERE como en la cláusula HAVING, y también en la cláusula FROM. Hasta ahora, dichas subconsultas podían tratarse de modo independiente y, para comprender mejor el funcionamiento de la sentencia, se podía suponer que la subconsulta se ejecuta en primer lugar, y se sustituye ésta en la sentencia SELECT principal por su valor, como se muestra en el siguiente ejemplo:

```
-- facturas con descuento máximo
SELECT *
FROM   facturas
WHERE  dto = ( SELECT MAX(dto) FROM facturas );
```

en primer lugar se obtiene el descuento máximo de las facturas, se sustituye la subconsulta por este valor y, por último, se ejecuta la consulta principal.

### 4.10.1. Referencias externas

En ocasiones sucede que la subconsulta se debe recalcular para cada fila de la consulta principal, estando la subconsulta parametrizada mediante valores de columnas de la consulta principal. A este tipo de subconsultas se les llama subconsultas correlacionadas y a los parámetros de la subconsulta que pertenecen a la consulta principal se les llama referencias externas.

La siguiente sentencia obtiene los datos de las facturas que tienen descuento en todas sus líneas:

```
SELECT *
FROM   facturas AS f
WHERE  0 < ( SELECT MIN(COALESCE(l.dto,0))
             FROM   lineas_fac AS l
             WHERE  l.codfac = f.codfac );
```

La referencia externa es `f.codfac`, ya que es una columna de la consulta principal. En este caso, se puede imaginar que la consulta se ejecuta del siguiente modo. Se recorre, fila a fila, la tabla de las facturas. Para cada fila se ejecuta la subconsulta, sustituyendo `f.codfac` por el valor que tiene en la fila actual de la consulta principal. Es decir, para cada factura se obtiene el descuento mínimo en sus líneas. Si este descuento mínimo es mayor que cero, significa que la factura tiene descuento en todas sus líneas, por lo que se muestra en el resultado. Si no es así, la factura no se muestra. En cualquiera de los dos casos, se continua procesando la siguiente factura: se obtienen sus líneas y el descuento mínimo en ellas, etc.

### 4.10.2. Operadores `EXISTS, NOT EXISTS`

En un apartado anterior se han presentado los operadores que se pueden utilizar con las subconsultas para hacer restricciones en las cláusulas `WHERE` y `HAVING`. En aquel momento no se citó, intencionadamente, un operador, ya que éste se utiliza siempre con referencias externas: el operador `EXISTS`.

`EXISTS ( subconsulta )`
La subconsulta se evalúa para determinar si devuelve o no alguna fila. Si devuelve al menos una fila, se evalúa a verdadero. Si no devuelve ninguna fila, se evalúa a falso. La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

En la ejecución de la subconsulta, en cuanto se devuelve la primera fila, se devuelve verdadero, sin terminar de obtener el resto de las filas.

Puesto que el resultado de la subconsulta carece de interés (sólo importa si se devuelve o no alguna fila), se suelen escribir las consultas indicando una constante en la cláusula `SELECT` en lugar de `*` o cualquier columna:

```
-- facturas que en alguna línea no tiene dto
SELECT *
FROM   facturas AS f
WHERE  EXISTS ( SELECT 1
                FROM   lineas_fac AS l
                WHERE  l.codfac = f.codfac
                AND    COALESCE(dto,0)=0);
```

`NOT EXISTS ( subconsulta )`
La subconsulta se evalúa para determinar si devuelve o no alguna fila. Si devuelve al menos una fila, se evalúa a falso. Si no devuelve ninguna fila, se evalúa a verdadero. La subconsulta puede tener referencias externas, que actuarán como constantes durante la evaluación de la subconsulta.

En la ejecución de la subconsulta, en cuanto se devuelve la primera fila, se devuelve falso, sin terminar de obtener el resto de las filas.

Puesto que el resultado de la subconsulta carece de interés (sólo importa si se devuelve o no alguna fila), se suelen escribir las consultas indicando una constante en la cláusula `SELECT` en lugar de `*` o cualquier columna:

```
-- facturas que no tienen líneas sin descuento
SELECT *
FROM   facturas AS f
WHERE  NOT EXISTS ( SELECT 1
                    FROM   lineas_fac AS l
                    WHERE  l.codfac = f.codfac
                    AND    COALESCE(dto,0)=0);
```

### 4.10.3. Sentencias equivalentes

Algunos SGBD no son eficientes procesando consultas que tienen subconsultas anidadas con referencias externas, por lo que es muy conveniente saber encontrar sentencias equivalentes que no las utilicen, si es posible.

Por ejemplo, la siguiente sentencia también obtiene los datos de las facturas que tienen descuento en todas sus líneas. Utiliza una subconsulta en la cláusula `FROM` y no posee referencias externas.

```
SELECT *
FROM   facturas JOIN
       ( SELECT codfac
         FROM   lineas_fac
         GROUP BY codfac
         HAVING MIN(COALESCE(dto,0))>0 ) AS lf
       USING (codfac);
```

Una sentencia equivalente, que tampoco utiliza referencias externas, es la siguiente:

```
SELECT *
FROM   facturas
WHERE  codfac IN ( SELECT codfac
                   FROM   lineas_fac
                   GROUP BY codfac
                   HAVING MIN(COALESCE(dto,0))>0 );
```

## 4.10.4. Ejemplos

**Ejemplo 4.16** *¿Cuántos clientes hay que en todas sus facturas han pagado 18 % de IVA?*

En la primera versión se van a utilizar operadores de conjuntos:

```
SELECT COUNT(*) AS clientes
FROM   (SELECT codcli FROM facturas
        WHERE iva = 18
        EXCEPT
        SELECT codcli FROM facturas
        WHERE COALESCE(iva,0) <> 18) AS t;
```

La siguiente sentencia no utiliza la subconsulta del `FROM`, pero seguramente será más cara porque hay que acceder a la tabla clientes:

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  WHERE iva = 18
                  EXCEPT
                  SELECT codcli FROM facturas
                  WHERE COALESCE(iva,0) <> 18);
```

La siguiente versión utiliza `NOT IN`, aunque ya se sabe que puede dar problemas cuando hay nulos:

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  WHERE  iva = 18)
AND    codcli NOT IN (SELECT codcli FROM facturas
                      WHERE  COALESCE(iva,0) <> 18
                      AND codcli IS NOT NULL);
```

La siguiente sentencia sigue una estrategia diferente: se ha pagado siempre el 18 % de IVA si el IVA máximo y el mínimo son ambos 18.

```
SELECT COUNT(*) AS clientes
FROM   clientes
WHERE  codcli IN (SELECT codcli FROM facturas
                  GROUP BY codcli
                  HAVING MAX(COALESCE(iva,0)) = 18
                  AND    MIN(COALESCE(iva,0)) = 18 );
```

Con la subconsulta en el `FROM` es posible evitar la visita de la tabla de los clientes:

```
SELECT COUNT(*) AS clientes
FROM   (SELECT codcli FROM facturas
        GROUP BY codcli
        HAVING MAX(COALESCE(iva,0)) = 18
        AND    MIN(COALESCE(iva,0)) = 18 ) AS t;
```

**Ejemplo 4.17** *¿Cuántos pueblos hay en los que no tenemos clientes?*

Una versión con operadores de conjuntos es la siguiente:

```
SELECT COUNT(*) AS pueblos
FROM   (SELECT codpue FROM pueblos
        EXCEPT
        SELECT codpue FROM clientes) AS t;
```

Otra versión es la que utiliza `NOT IN`.

```
SELECT COUNT(*) AS pueblos
FROM   pueblos
WHERE  codpue NOT IN (SELECT codpue FROM clientes);
```

**Ejemplo 4.18** *Para proponer ofertas especiales a los buenos clientes, se necesita un listado con los datos de aquellos que en los últimos quince meses (los últimos 450 días) han hecho siempre facturas por un importe superior a 400 €.*

Se puede pensar en obtener el resultado recorriendo, uno a uno, los clientes. Para cada cliente, comprobar, mediante una subconsulta, la restricción: que todas sus facturas de los últimos 450 días tengan un importe superior a 400 €. Ya que la subconsulta se ha de ejecutar para cada cliente, llevará una referencia externa.

La restricción que se ha de cumplir sobre todas las facturas de ese periodo se puede comprobar con `ALL` o con `NOT EXISTS`: o bien todas las facturas del

cliente (en el periodo) tienen un importe superior a 400 €, o bien no existen facturas de ese cliente (en el periodo) con un importe igual o inferior a 400 €.

Se debe tener en cuenta que con los dos operadores (`ALL`, `NOT EXISTS`) se obtendrán también en el resultado los clientes que no tienen ninguna factura, por lo que será preciso asegurarse de que los clientes seleccionados hayan comprado en alguna ocasión en dicho periodo.

A continuación se muestran las dos versiones de la consulta que utilizan las referencias externas tal y como se ha explicado.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  400 < ALL ( SELECT SUM(l.cant*l.precio)
                   FROM   lineas_fac l JOIN facturas f
                          USING(codfac)
                   WHERE  f.fecha >= CURRENT_DATE - 450
                   AND    f.codcli = c.codcli  -- ref. externa
                   GROUP BY f.codfac )
AND    c.codcli IN ( SELECT f.codcli
                     FROM   facturas f
                     WHERE  f.fecha >= CURRENT_DATE - 450 )
ORDER BY c.nombre;
```

Nótese que con `NOT EXISTS` el predicado sobre el importe de las facturas es el único que debe aparecer negado.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  NOT EXISTS ( SELECT 1
                    FROM   lineas_fac l JOIN facturas f
                           USING(codfac)
                    WHERE  f.fecha >= CURRENT_DATE - 450
                    AND    f.codcli = c.codcli  -- ref. externa
                    GROUP BY f.codfac
                    HAVING SUM(l.cant*l.precio) <= 400 )
AND    c.codcli IN ( SELECT f.codcli
                     FROM   facturas f
                     WHERE  f.fecha >= CURRENT_DATE - 450 )
ORDER BY c.nombre;
```

En la siguiente versión se evitan las referencias externas utilizando operadores de conjuntos. Obsérvese la subconsulta: del conjunto de los clientes que alguna vez han comprado en ese periodo con facturas de más de 400 €, se debe eliminar a aquellos que además han comprado alguna de 400 € o menos. Puesto que se utiliza el operador `IN`, no es necesaria la restricción adicional que comprueba que los clientes seleccionados hayan comprado alguna vez en el periodo: si están en la lista es porque lo han hecho.

```
SELECT c.codcli, c.nombre
FROM   clientes c
WHERE  c.codcli IN ( SELECT f.codcli
                     FROM   lineas_fac l JOIN facturas f
                            USING(codfac)
                     WHERE  f.fecha >= CURRENT_DATE - 450
                     GROUP BY f.codcli, f.codfac
                     HAVING SUM(l.cant*l.precio) > 400
                     EXCEPT
                     SELECT f.codcli
                     FROM   lineas_fac l JOIN facturas f
                            USING(codfac)
                     WHERE  f.fecha >= CURRENT_DATE - 450
                     GROUP BY f.codcli, f.codfac
                     HAVING SUM(l.cant*l.precio) <= 400 )
ORDER BY c.nombre;
```