Oracle 21c XE
SQL – DML I

En este capítulo se verá el lenguaje SQL enfocado a la manipulación de datos en una base de datos. Un subconjunto de SQL, el DML (*Data Manipulation Language*) se centra en la ejecución de consultas a una base de datos.

Tabla de contenido

Oracle 21c XE
SQL - DML

1 Introducción

SQL es el lenguaje declarativo de alto nivel con el cual todos las aplicaciones y usuarios acceden a los datos en una BD Oracle. Aunque algunas herramientas y aplicaciones enmascarán el uso de SQL, todas las tareas sobre una base de datos se realizan utilizando SQL.

SQL suministra una interfaz a una base de datos relacional como Oracle. SQL unifica tareas como las siguientes:

- ✔ Crear, modificar y eliminar objetos.
- ✔ Insertar, actualizar y borrar filas de tablas.
- ✔ Consulta de datos.
- ✔ Controlar el acceso a la base de datos y sus objetos.
- ✔ Garantizar la consistencia e integridad de la base de datos.

SQL puede utilizarse interactivamente, lo que significa que las sentencias se introducen manualmente en un programa. Las sentencias SQL también se pueden escribir dentro de un programa escrito en un lenguaje como C o Java.

Oracle SQL incluye muchas extensiones del estándar SQL ANSI/ISO. Las herramientas y aplicaciones Oracle suministran sentencias adicionales. Las herramientas SQL*Plus, SQL

Developer y Oracle Enterprise Manager te permiten ejecutar sentencias estándar ANSI/ISO SQL y cualquier otra sentencia adicional o funciones disponibles para esas herramientas.

Las sentencias SQL se dividen en varias categorías entre las que se encuentran:

- ✔ Lenguaje de definición de datos (DDL) para definir los objetos de la BD.
- ✔ Lenguaje de manipulación de datos (DML) para acceso a los objetos de la BD.
- ✔ Control de transacciones.
- ✔ Control del acceso a datos

El DML (*Data Manipulation Language*) es el conjunto de sentencias que está orientadas a la consulta, y manejo de datos de los objetos creados. El DML es un subconjunto muy pequeño dentro de SQL, pero es el más importante, ya que su conocimiento y manejo con soltura es imprescindible para el acceso a las BD. Básicamente consta de cuatro sentencias: `SELECT`, `INSERT`, `DELETE`, `UPDATE`. La primera de ella se emplea para acceso a la información almacenada en la BD, mientras que el resto se emplea para la actualización de los datos, que se verá en un capítulo posterior.

Las sentencias DML acceden y manipulan los datos existentes en las tablas. En el entorno SQL*Plus podemos introducir una sentencia DML detrás del prompt `SQL>` y el resultado se mostrará a continuación. En el entorno SQL Developer, podemos introducir una sentencia SQL en una hoja de trabajo y veremos el resultado en la parte inferior.

2 Sentencia SELECT

Una consulta o sentencia `SELECT` selecciona datos de uno más tablas o vistas. Su sintaxis básica es:

```
SELECT [DISTINCT] <lista_ítems>
FROM <lista_tablas>
[WHERE <condición>]
[GROUP BY <expresión> [HAVING <condición>]]
[ORDER BY <expresión>]
```

Donde:

- ✔ `SELECT [DISTINCT] lista_ítems` ➜ Lista de columnas de una o más tablas separadas por comas que se desean recuperar los datos.
- ✔ `FROM lista_tablas` ➜ Tablas donde se encuentran las columnas que aparecen en SELECT.
- ✔ `WHERE condición` ➜ Condición que tienen que cumplir las filas seleccionadas para ser mostradas.
- ✔ `GROUP BY expresión [HAVING condición]]` ➜ Agrupamiento de las filas seleccionadas para crear grupos por la expresión indicada. Si se añade `HAVING` condición se define una condición que las filas agrupadas tienen que cumplir para

mostrarse.

- ✔ `ORDER BY expresión` .- El resultado final se ordena por alguna de las columnas de selección.

Todas las cláusulas son opcionales, excepto `SELECT` y `FROM`. En `SELECT` se deben indicar los ítems de selección que se desean obtener en el resultado. Los distintos ítems se separan por coma. Cada ítem de selección genera una única columna de resultados en la consulta. Un ítem de selección puede ser:

- ✔ Un nombre de columna identificando una columna de una tabla que aparezca en la cláusula `FROM`. Cuando un nombre de columna aparece como ítem de selección, SQL simplemente toma el valor de esa columna de cada fila de la tabla de la base de datos y lo coloca en la fila correspondiente de los resultados de la consulta. Opcionalmente se puede cualificar con el nombre de la tabla utilizando la sintaxis `TABLA.COLUMNA`. Esta cualificación es obligada cuando en la lista de ítems aparecen dos columnas con el mismo nombre de diferente tabla.
- ✔ Una constante, especificando que el mismo valor constante va a aparecer en todas las filas del resultado de la consulta.
- ✔ Una expresión SQL, indicando que SQL debe calcular el valor a colocar en los resultados. La expresión podrá estar compuesta de constantes, nombres de columnas, funciones SQL, etc.

La lista de tablas de la clausula `FROM` debe separar cada nombre de tabla con una coma. Cada identificador de tabla identifica una tabla del esquema que contiene datos a recuperar por la consulta. Estas tablas se denominan *tablas fuente* de la consulta, ya que constituyen la fuente de todos los datos que aparecen en los resultados. Las tablas deben de existir y si no existiera alguna ocurriría un error. Si alguna tabla pertenece a un esquema diferente se debe cualificar con `ESQUEMA.TABLA`.

El resultado de una consulta SQL es siempre una tabla de datos, semejante a las tablas de la base de datos. Si se escribe una sentencia SELECT utilizando SQL*Plus o SQL Developer se visualizarán los resultados de la consulta en forma tabular. Si una aplicación envía una consulta a la base de datos utilizando SQL programado, la tabla de resultados de la consulta se devuelve al programa que tendría la responsabilidad de mostrarlos en la interfaz de usuario de la aplicación. En cualquier caso, los resultados tienen siempre el mismo formato tabular de fila/columna que las tablas de la base de datos. Generalmente los resultados de la consulta formarán una tabla con varias columnas y varias filas.

Por ejemplo, una consulta sencilla solicita varias columnas de una única tabla.

```
SELECT nif, nombre, email
FROM cliente
```

La consulta anterior mostraría el `nif`, `nombre` e `email` de todas las filas de la tabla `cliente`.

2.1 Columnas calculadas

Como se indicó anteriormente, además de las columnas cuyos valores provienen directamente de la base de datos, una consulta SQL puede incluir columnas calculadas cuyos valores se calculan a partir de los valores de los datos almacenados, constantes y/o funciones SQL. Para solicitar una columna calculada, se especifica una expresión SQL en la lista de selección. Para construir las expresiones SQL se pueden utilizar los operadores y funciones vistos anteriormente. Por ejemplo

```
SELECT referencia, precio, precio - precio * dto
FROM lpedido;
```

Recuperará la `referencia`, el `precio` y el resultado de restar al `precio` la aplicación del porcentaje de descuento. En el resultado, la tercera columna se calcula a partir de los valores de datos de cada fila utilizando una expresión aritmética.

En el siguiente ejemplo recuperaría para cada factura su número, el número del mes de su fecha y el año. Para ello hemos utilizado las funciones `MONTH()` y `YEAR()`.

```
SELECT numero, MONTH(fecha), YEAR(fecha)
FROM factura;
```

2.2 Selección de todas las columnas de una tabla

A veces es conveniente visualizar el contenido de todas las columnas de una tabla. SQL permite utilizar un asterisco (`*`) en lugar de la lista de selección como abreviatura de todas las columnas. El resultado contendrá todas las columnas en el mismo orden en que están definidas en la tabla. Por ejemplo, la consulta

```
SELECT * FROM cliente;
```

recuperaría como resultado todas las columnas de la tabla cliente. También si se desea el asterisco puede cualificarse con el nombre de la tabla.

```
SELECT cliente.* FROM cliente;
```

Además del asterisco, que recuperaría todas las columnas de la tabla, puede añadirse más ítems de selección. Por ejemplo:

```
SELECT *, nif FROM cliente;
```

recuperaría todas las columnas de la tabla `cliente` y la columna `nif` de la misma tabla.

2.3 Encabezados de columna

Cuando los resultados de una consulta se muestran, el encabezado de columna por defecto es el propio nombre de la columna. Para visualizar una columna bajo un nuevo encabezado se puede especificar uno nuevo (un alias) inmediatamente después del nombre

de la columna. El alias renombra la columna solo para la consulta, pero no cambia su nombre en la base de datos.

Por ejemplo, podemos listar las líneas de pedido con su número de pedido, número de línea, referencia de artículo y precio con nuevos encabezados

```
SELECT npedido Pedido, nlinea Línea, referencia Artículo, precio
pvp
FROM lpedido;
```

Si ejecutamos la consulta anterior vemos que los encabezados están en letras mayúsculas y solamente contienen una palabra. Si queremos que el encabezado contenga espacios en blanco y respete la capitalización hay que encerrarlos entre comillas dobles. El mismo ejemplo anterior puede reescribirse así

```
SELECT npedido "Nº Pedido", nlinea "Nº Línea", referencia
"Referencia de artículo", precio "Precio Venta"
FROM lpedido;
```

2.4 Filas duplicadas

Si una consulta incluye una clave primaria de una tabla en su lista de selección, entonces cada fila de resultados será única. Si no se incluye la clave primaria en los resultados pueden producirse filas duplicadas. Por ejemplo

```
SELECT nif FROM pedido;
```

Aparecen filas duplicadas ya que un cliente tiene varios pedidos. Se pueden eliminar las filas duplicadas de los resultados de la consulta insertando la palabra clave `DISTINCT` en la sentencia SELECT justo antes de la lista de selección. Así

```
SELECT DISTINCT nif FROM pedido;
```

Ahora se suprimen las filas duplicadas.

Si se utiliza la cláusula `DISTINCT` retorna filas únicas, es decir, no habrá dos filas con los mismos valores en todos sus campos. Por ejemplo, si a la sentencia anterior le añadimos la columna `total_pedido`.

```
SELECT DISTINCT nif, total_pedido FROM pedido;
```

Como se puede observar, no hay dos filas iguales. También se puede especificar la palabra clave `ALL` para indicar explícitamente que las filas duplicadas se muestren, pero es innecesario ya que éste es el comportamiento por defecto.

2.5 Criterio de selección. Cláusula WHERE

Generalmente se deseará seleccionar parte de las filas de una tabla, y sólo se incluirán esas filas en los resultados. La cláusula `WHERE` se emplea para especificar qué condiciones deben cumplirse para que una fila entre dentro del resultado retornado. Para construir las

condiciones se podrán utilizar todos los operadores lógicos vistos anteriormente. Es posible construir condiciones complejas uniendo dos o más condiciones simples a través de los operadores lógicos `AND` y `OR`.

Por ejemplo, listar todos los pedidos cuyo importe supere 100 €.

```
SELECT * FROM pedido WHERE total_pedido > 100;
```

Si una fila tiene una columna con valor `NULL` y esa columna forma parte de una expresión condicional en la cláusula `WHERE`, el resultado siempre será falso y esa fila no aparecerá en el conjunto de resultados.

SQL ofrece un amplio conjunto de condiciones de búsqueda que permiten especificar muchos tipos diferentes de consultas, empleándolas en la cláusula `WHERE`. A continuación se exponen las más básicas.

2.5.1 Test de comparación

La condición de búsqueda más común utilizada en una consulta SQL es el test de comparación. En un test de comparación, SQL calcula y compara los valores de dos expresiones SQL por cada fila de datos. Las expresiones pueden ser tan simples como un nombre de columna o una constante, o pueden ser expresiones aritméticas complejas. Las comparaciones se harán con los operadores de comparación típicos. Por ejemplo

```
SELECT npedido, nif FROM pedido
WHERE total_pedido > 100;
```

recuperará el *número* y el *nif* de los pedidos con un importe superior a 100 €. Por otro lado si ejecutamos

```
SELECT *
FROM lpedido
WHERE dto = 0 OR precio < 200;
```

recuperará las líneas de albarán cuyo descuento es 0 o su precio es inferior a 200. Otro ejemplo, usando una columna calculada.

```
SELECT referencia, descripcion,  pvp - pvp * dto_venta
FROM articulo
WHERE pvp - pvp * dto_venta < 20 AND und_disponibles > 3;
```

Este último ejemplo, recupera la *referencia* la *descripción,* el precio neto (una vez hemos hecho el descuento) de los artículos cuyo precio neto es inferior a 20 € y las unidades disponibles superan 3. Como se puede observar, no es necesario que las columnas que intervienen en los test de comparación aparezcan en la lista de selección.

En el siguiente ejemplo vamos a utilizar funciones. Consiste en listar todos los envíos hechos durante el mes de enero de 2023.

```
SELECT * FROM envio
WHERE MONTH(fecha) = 1 AND YEAR(fecha) = 2023;
```

2.5.2 Test de rango BETWEEN

SQL proporciona una forma diferente de condición de búsqueda con el test de rango. Este comprueba si un valor de dato se encuentra entre dos valores especificados. Implica el uso de tres expresiones SQL. La primera expresión define el valor a comprobar, las expresiones segunda y tercera definen los extremos superior e inferior del rango a comprobar. Los tipos de datos de las tres expresiones deben ser comparables. Por ejemplo

```
SELECT npedido, nif, total_pedido, fecha FROM pedido
WHERE fecha BETWEEN '01/01/2023' AND '31/01/2023';
```

La consulta anterior lista los pedidos cuya fecha está entre el 1 de enero del 2023 y el 31 de enero del 2023, es decir, que el pedido sea del mes de enero del 2023. La sentencia anterior sería similar a:

```
SELECT npedido, nif, total_pedido, fecha FROM pedido
WHERE fecha >= '01/01/2018' AND fecha <= '31/01/2018';
```

Para que se devuelva una fila en el conjunto de resultados, es necesario que la primera expresión antes del `BETWEEN` sea mayor igual que la expresión que define el extremo inferior del rango y menor o igual que la expresión que define el extremo superior del rango.

2.5.3 Test de pertenencia a conjunto. Cláusula IN

Otra condición habitual es el test de pertenencia a conjunto (operador `IN`). Examina si un valor coincide con uno de una lista de valores objetivo. Por ejemplo

```
SELECT * FROM articulo WHERE dto_venta IN (0.1, 0.2, 0.3);
```

La sentencia anterior lista todos los artículos cuyo descuento de venta es 10%, 20% o 30%. Sería equivalente a

```
SELECT * FROM articulo
WHERE dto_venta = 0.1 or dto_venta = 0.2 or dto_venta = 0.3;
```

También se puede comprobar que un valor no coincida con ninguno de la lista de valores objetivo anteponiendo el operador lógico `NOT`. Por ejemplo

```
SELECT * FROM articulo
WHERE categoria NOT IN ( 'CARN', 'FRUT', 'CONG');
```

El ejemplo anterior visualizaría todos los artículos que no pertenecen a las categorías de carne, fruta y congelados.

2.5.4 Test de correspondencia con patrón. Cláusula LIKE

Se puede utilizar un test de comparación simple para recuperar las filas en donde el

contenido de una columna de texto se corresponde parcialmente con un cierto texto particular. Para ello se construye un patrón que es una cadena con uno o más caracteres comodín. Por ejemplo

```
SELECT * FROM cliente WHERE nombre LIKE 'J%';
```

devolvería todos los clientes cuyo nombre comienza por `'J'`, sin importar el resto del nombre. En la siguiente sentencia

```
SELECT * FROM direccion_envio WHERE poblacion LIKE 'S%a';
```

devolvería todos los clientes cuya población empieza por `'S'` y acaba en `'a'`. En un capítulo anterior se explican con detalle los dos caracteres comodín que existen.

También, si lo que se desea es que la expresión de texto no se ajuste al patrón, se puede anteponer el operador lógico `NOT` a `LIKE`. Por ejemplo

```
SELECT * FROM cliente
WHERE nombre NOT LIKE 'PE%';
```

Listaría todos los clientes cuyo nombre no comiencen por `'PE'`.

2.5.5 Test de valor nulo

A veces es útil comprobar explícitamente los valores `NULL` en una condición de búsqueda y manejarlos directamente. SQL proporciona un test especial de valor nulo. Esta consulta muestra todos los envíos cuya forma de envío es nulo.

```
SELECT * FROM envio WHERE forma_envio IS NULL;
```

La forma negada del test de valor nulo (`IS NOT NULL`) encuentra las filas que no contienen un valor `NULL`. Por ejemplo

```
SELECT * FROM direccion_envio
WHERE direccion IS NOT NULL;
```

2.6 Ordenación de los resultados de una consulta. Cláusula ORDER BY

Al igual que las filas de una tabla en la base de datos las filas de los resultados de una consulta no están dispuestas en ningún orden particular. Se puede pedir a SQL que ordene los resultados de una consulta incluyendo la cláusula `ORDER BY` en la sentencia `SELECT`. La cláusula `ORDER BY` seguida por una lista de especificadores de ordenación separadas por comas. Por ejemplo

```
SELECT * FROM cliente ORDER BY apellidos, nombre;
```

Lista los clientes ordenados alfabéticamente por `apellidos` y `nombre`. La primera especificación de ordenación (`apellidos`) es la clave de ordenación mayor, las que le sigan (`nombre`) son progresivamente claves de ordenación menores, utilizadas para

desempatar cuando dos filas de resultados tienen los mismos valores para las claves mayores. Utilizando la cláusula `ORDER BY` se puede solicitar la ordenación en secuencia ascendente o descendente y se puede ordenar con respecto a cualquier elemento, incluso si no aparece en la lista de selección.

Dentro de esta cláusula podrá aparecer cualquier expresión que pueda aparecer en el `SELECT`, es decir, pueden aparecer columnas, constantes (no tiene sentido, aunque está permitido), expresiones y funciones SQL.

Después de cada columna de ordenación se puede incluir una de las palabras reservadas `ASC` o `DESC`, para hacer ordenaciones ASCendentes o DESCendentes. Por defecto, si no se pone nada se hará ascendente. Por ejemplo

```
SELECT * FROM factura
ORDER BY fecha ASC, nfactura DESC
```

Lista las facturas ordenadas por fecha ascendetemente y después por número descendetemente.

Como característica adicional, se pueden incluir números en la ordenación, que serán sustituidos por la columna correspondiente del `SELECT` en el orden que indique el número. Por ejemplo

```
SELECT * FROM factura ORDER BY 1, fecha, 3;
```

Lista las facturas ordenados por `nfactura` (columna 1 del resultado), la `fecha` y por `nenvio` (columna 3 de la lista de resultados).

Si la columna de resultados de la columna utilizada para ordenación es una columna calculada, no tiene nombre de columna que se pueda emplear en una especificación de ordenación. En este caso, debe especificarse un número de columna en lugar de un nombre, como en este ejemplo.

```
SELECT numero, referencia, unidades * precio
FORM lpedido
ORDER BY 3;
```

2.7 Consultas sumarias

Una consulta agrupada se utiliza para considerar las filas cuyas ciertas columnas tienen el mismo valor, y procesarlas de la misma manera, para contar, sumar, hacer la media... Las consultas típicas son para contar las filas de cierto tipos, sumar los importes de cierto cliente, etc.

Una función de columna SQL acepta una columna con tipo de datos numérico como argumento y produce un único dato que sumariza la columna. Por ejemplo, la función `AVG()` acepta una columna de datos y calcula su promedio. He aquí una consulta que utiliza la función de columna `AVG()` para calcular el valor promedio de dos columnas de la tabla `lpedido`.

```
SELECT AVG(precio), AVG(dto) FROM lpedido;
```

La primera función toma valores de la columna `precio` y calcula su promedio. La segunda promedia los valores de la columna `dto`. La consulta produce una única fila de resultados que sumarizan los datos de la tabla `lpedido`. También los valores de entrada puede ser una expresión

```
SELECT AVG(precio - precio * dto) FROM lAlbaran;
```

Las funciones de grupo que pueden usarse son:

Función	Descripción
`SUM(columna)`	Devuelve la suma
`AVG(columna)`	Devuelve la media artimética
`MIN(columna)`	Devuelve el valor mínimo
`MAX(columna)`	Devuelve el valor máximo
`COUNT(*)`	Devuelve la cuenta de valores

Todas estas funciones permite incluir el modificador `DISTINCT` delante de la columa como argumento de función para que omita los repetidos.

2.7.1 Cálculo del total de una columna (SUM)

La función `SUM()` calcula la suma de una columna de valores de datos. Los datos de la columna deben ser de tipo numérico. El resultado es del mismo tipo que el dato básico de los valores de la columna, pero el resultado puede tener una precisión superior. Por ejemplo, vamos a sacar el total del importe de los pedidos.

```
SELECT SUM(total_pedido) FROM pedido
```

Esto nos sumara todos los importes de la tabla pedido y los ofrecerá en una única fila.

2.7.2 Cálculo del promedio de una columna (AVG)

La función `AVG()` calcula el promedio de una columna de valores de datos. Al igual que la función `SUM()`, los datos deben tener tipo numérico. Ya que la función `AVG()` suma los valores de la columna y luego los divide por el número de valores, su resultado puede tener un tipo de dato diferente al de los valores de la columna.

2.7.3 Determinación de valores extremos

Las funciones de columna `MIN()` y `MAX()` determinan los valores mayor y menor de una columna, respectivamente. Los datos de la columna pueden ser de tipo numérico, cadena o fecha/hora. El resultado de la función tiene el mismo tipo de dato que los datos de la columna. Por ejemplo, para listar la fecha más antigua de las facturas.

```
SELECT MIN(fecha) FROM factura;
```

Para obtener el precio del artículo más caro.

```
SELECT MAX(pvp) FROM articulo;
```

Para obtener el precio neto máximo y el mínimo en todas las líneas de pedido.

```
SELECT MAX(precio - precio * dto), MIN(precio - precio * dto)
FROM lpedido;
```

2.7.4 Cuenta de valores de datos (COUNT)

La función de columnas `COUNT()` cuenta el número de valores de datos que hay en una columna. Los datos de la columna pueden ser de cualquier tipo. La función `COUNT()` devuelve siempre un entero, independientemente del tipo de datos de la columna. Por ejemplo

```
SELECT COUNT(nif) FROM cliente;
```

Dará como resultado el número de filas en la tabla cliente.

```
SELECT COUNT(precio_coste) FROM articulo_proveedor
WHERE precio_coste > 10;
```

El resultado será el número de artículos cuyo precio de compra es superior a 10 €. Observese que la función `COUNT()` ignora los valores de los datos en la columna, simplemente cuenta cuántos datos hay. En consecuencia no importa realmente qué columna se especifica como argumento de la función `COUNT()`. El último ejemplo podría haberse escrito igualmente así

```
SELECT COUNT(referencia) FROM articulo_proveedor
WHERE precio_coste > 10;
```

De hecho, es extraño pensar en la consulta como "*contar cuántos precios de coste de artículo*" o "*contar cuántas referencias de artículo*"; es mucho más fácil imaginar que se pide "*contar cuántos artículos*". Por esta razón, SQL soporta una función de columna especial `COUNT(*)` QUE CUENTA FILAS EN LUGAR DE VALORES DE DATOS. He aquí la misma consulta, reescrita una vez más para utilizar la función `COUNT(*)`.

```
SELECT COUNT(*) FROM articulo_proveedor WHERE precio_coste > 10;
```

Hay que pensar en la función `COUNT()` como una función que cuenta filas, en lugar de valores, aunque a veces es necesario que cuente valores.

2.7.5 Valores NULL y funciones de columna

La funciones de columna `SUM()`, `AVG()`, `MIN()`, `MAX()` y `COUNT()` aceptan cada una de ellas una columna de valores de datos como argumento y producen un único valor como resultado. ¿Qué sucede si uno o más de los valores de la columna toma un valor NULL? El estándar SQL ANSI/ISO especifica que los valores NULL de la columna sean ignorados por

las funciones de columna, es decir, cuando haya que sumar, calcular la media, el mínimo, máximo o contar valores de columna, aquellas que tengan valor NULL no entrarán en el resultado.

2.7.6 Eliminación de filas duplicadas (DISTINCT)

Recordemos que se puede especificar la palabra clave `DISTINCT` al comienzo de la lista de selección para eliminar filas duplicadas de los resultados de la consulta. También se puede pedir a SQL que elimine valores duplicados de una columna antes de aplicarle una función de columna. Para eliminar valores duplicados, la palabra clave `DISTINCT` se incluye delante del argumento de la función de columna. Por ejemplo

```
SELECT COUNT( DISTINCT nif ) FROM pedido;
```

devolverá los `nif` de clientes no repetidos en la tabla pedido, la cual tiene más pedidos, cada uno con un cliente, pero hay clientes que tienen más de un pedido y su valor estará repetido.

El estándar permite usar `DISTINCT` en las funciones `SUM()` y `AVG()`, pero no en `MIN()` y `MAX()`, ya que no tiene impacto sobre los resultados. En `COUNT()` se puede utilizar siempre que se especifique una columna como argumento, no es posible utilizarla en `COUNT(*)` ya que esta forma de la función cuenta las filas.

Además, la palabra clave `DISTINCT` sólo puede ser especificada una vez en una consulta. Si aparece en el argumento de una función de columna, no puede aparecer en ninguna otra.

2.8 Consultas agrupadas

Las consultas sumarias descritas hasta ahora son como los totales al final de un informe. Condensan todos los datos detallados del informe en una única fila sumaria de datos. Al igual que los subtotales son útiles en informes impresos, con frecuencia es conveniente sumarizar los resultados de la consulta a un nivel "subtotal". La cláusula `GROUP BY` de la sentencia `SELECT` proporciona esta capacidad. Por ejemplo

```
SELECT referencia, SUM(unidades * (precio - precio * dto) )
FROM lpedido
GROUP BY referencia
```

Esto nos sumará los importes de las líneas de pedido que tenga el mismo valor de `referencia`. Cuando en la cláusula `SELECT` no se incluyen funciones de grupo una consulta `GROUP BY` es equivalente a una consulta `SELECT DISTINCT`.

SQL lo que ha hecho es lo siguiente:

1. SQL divide las líneas de pedidos en grupos de líneas de pedidos, un grupo por cada valor del atributo `referencia`. Dentro de cada grupo, todas las líneas de pedido tienen el mismo valor en la columna `referencia`.
2. Por cada grupo, SQL calcula la suma de la expresión `unidades * (precio -`

`precio * dto)` para todas las filas del grupo, y genera una única fila sumaria de resultados. La fila contiene el valor de la columna `referencia` y el total de los importes calculados.

Las primeras filas del resultado han sido como sigue:

```
BEBI0001 124,875
BEBI0002 202,5
BEBI0003 253,125
BEBI0004 531,5625
BEBI0005 885,9375
CARN0001 1996,5
CARN0002 701,25
CARN0003 670,3125
...
```

Hay más filas del resultado, pero por razones de espacio solo se muestran las primeras. Una consulta que incluya la cláusula `GROUP BY` se denomina consulta agrupada, ya que agrupa los datos de las tablas fuente y produce una única fila sumaria por cada grupo de filas. Las columnas indicadas en la cláusula `GROUP BY` se denominan columnas de agrupación de la consulta, ya que ellas son las que determinan cómo se dividen las filas en grupo.

Ejemplos de consultas agrupadas:

```
SELECT fecha, COUNT(*) FROM factura
GROUP BY fecha;
```

En el ejemplo anterior se listan las facturas que hay por fecha de factura.

```
SELECT categoria, AVG(pvp)
FROM articulo
GROUP BY categoria;
```

En el ejemplo anterior obtendríamos el precio medio de venta de los artículos por categoría de artículo. En el siguiente obtenemos el precio mínimo de venta de cada artículo en los pedidos.

```
SELECT referencia, MIN(precio) FROM lpedido
GROUP BY referencia;
```

El siguiente ejemplo es algo más complicado. Se listan los clientes, fecha y mayor total de pedido cuando la fecha del pedido es posterior al 15 de diciembre de 2022.

```
SELECT cliente, fecha, MAX( total_pedido )
FROM pedido
WHERE fecha > TO_DATE('15-DIC-2022')
GROUP BY cliente, fecha;
```

Hay una relación íntima entre las funciones de columna SQL y la cláusula `GROUP BY`. Cuando la cláusula `GROUP BY` está presente, informa a SQL que debe dividir los resultados

detallados en grupos y aplicar la función de columna separadamente a cada grupo, produciendo un único resultado por cada grupo. Es posible agrupar los resultados de la consulta en base a contenidos de dos o más columnas, tal y como aparece en el último ejemplo. La consulta proporciona una fila sumaria por cada combinación diferente de cliente y fecha.

Las consultas agrupadas están sujetas a algunas limitaciones bastante estrictas. Las columnas de agrupación deben ser columnas efectivas de las tablas designadas en la cláusula `FROM` de la consulta. No se pueden agrupar las filas basándose en el valor de una expresión calculada.

También hay restricciones sobre los elementos que pueden aparecer en la lista de selección de una consulta agrupada. Todos los elementos de la lista de selección deben tener un único valor por cada grupo de filas. Básicamente, esto significa que un elemento de selección en una consulta agrupada puede ser:

- ✔ Una constante.
- ✔ Una función de columna, que produce un único valor sumarizando las filas del grupo.
- ✔ Una columna de agrupación, que por definición tiene el mismo valor en todas las filas del grupo.
- ✔ Una expresión que afecte a combinaciones de los anteriores.

En la práctica, una consulta agrupada incluirá siempre una columna de agrupación y una función de columna en su lista de selección. Si no aparece una función de columna, la consulta puede expresarse más sencillamente utilizando `SELECT DISTINCT`, sin `GROUP BY`.

Un valor `NULL` presenta un problema especial cuando aparece en una columna de agrupación. Si el valor de la columna es desconocido, ¿en qué grupo debería colocarse la fila? En la cláusula `WHERE`, cuando se comparan dos valores `NULL` diferentes, el resultado es `NULL` (no `TRUE`), es decir, los dos valores `NULL` no se consideran iguales. Aplicando el mismo convenio a la cláusula `NULL` se forzaría a SQL a colocar cada fila con una columna de agrupación `NULL` en un grupo aparte.

En lugar de ello, el estándar SQL ANSI/ISO considera que dos valores `NULL` son iguales a efectos de la cláusula `GROUP BY`. Si dos filas tienen `NULL` en las mismas columnas de agrupación y valores idénticos en las columnas de agrupación no `NULL`, se agrupan dentro del mismo grupo de filas. Sin embargo, este comportamiento estándar no está aplicado a todos los SGBD.

2.8.1 Condiciones de búsqueda de grupos. Cláusula HAVING

Al igual que la cláusula `WHERE` puede ser utilizada para seleccionar y rechazar filas individuales que participan en una consulta, la cláusula `HAVING` puede ser utilizada para seleccionar y rechazar grupos de filas. El formato de la cláusula `HAVING` es análogo al de la cláusula `WHERE`, consistiendo en la palabra clave `HAVING` seguida de una condición de búsqueda. La cláusula `HAVING` especifica por tanto una condición de búsqueda para

grupos. Por ejemplo

```
SELECT nif, AVG(total_pedido) FROM pedido
GROUP BY nif
HAVING SUM(total_pedido) > 200;
```

En la consulta anterior, la cláusula `HAVING` elimina los grupos cuyo importe total no supera 200 €. Finalmente, la cláusula `SELECT` calcula el importe medio para cada uno de los grupos restantes y genera los resultados de la consulta. Otro ejemplo sería:

```
SELECT categoria, referencia, AVG(pvp)
FROM articulo
GROUP BY categoria, referencia
HAVING COUNT(*) >= 2;
```

La consulta anterior calcularía primero los grupos de `categoría` y `referencia` que superan las dos filas, y posteriormente para cada uno de esos grupos promedia el `pvp`. Otro ejemplo

```
SELECT cliente, MIN(total_pedido) FROM pedido
GROUP BY cliente
HAVING MIN(total_pedido) > 50;
```

En la consulta anterior, calcula los grupos de cliente que tienen importe mínimo superior a 50. Para todos ellos, genera resultados con el cliente e importe mínimo.

La cláusula `HAVING` se utiliza para incluir o excluir grupos de filas de los resultados de la consulta, por lo que la condición de búsqueda que especifica debe ser aplicable al grupo en su totalidad en lugar de a filas individuales. Esto significa que un elemento que aparezca dentro de la condición de búsqueda en una cláusula `HAVING` puede ser:

- ✔ Una constante.
- ✔ Una función de columna, que produzca un único valor que sumarice las filas del grupo.
- ✔ Una columna de agrupación, que por definición tiene el mismo valor para todas las filas del grupo.
- ✔ Una expresión, que afecte a combinaciones de las anteriores.

En la práctica, la condición de búsqueda de la cláusula `HAVING` incluirá siempre al menos una función de columna. Si no lo hiciera, la condición de búsqueda podría expresarse con la cláusula `WHERE` y aplicarse a filas individuales. El modo más fácil de imaginar si una condición de búsqueda pertenece a la cláusula `WHERE` o la cláusula `HAVING` es recordar como se aplican ambas cláusulas:

- ✔ La cláusula `WHERE` se aplica a filas individuales, por lo que las expresiones que contiene debe ser calculables para filas individuales.
- ✔ La cláusula `HAVING` se aplica a grupos de filas, por lo que las expresiones que

contengan deben ser calculables para un grupo de filas.

La cláusula `HAVING` se utiliza casi siempre juntamente con la cláusula `GROUP BY`, pero la sintaxis de la sentencia `SELECT` no lo precisa. Si una cláusula `HAVING` aparece sin una cláusula `GROUP BY`, SQL considera el conjunto entero de resultados detallados como un único grupo. En otras palabras, las funciones de columna de la cláusula `HAVING` se aplican a un solo y único grupo para determinar si el grupo está incluido o excluido de los resultados, y ese grupo está formado por todas las filas. El uso de una cláusula `HAVING` sin una cláusula corresponiente `GROUP BY` casi nunca se ve en la práctica.

2.9 Subconsultas

La característica de subconsulta SQL permite utilizar los resultados de una consulta como parte de otra. Una subconsulta es una consulta que aparece dentro de la cláusula `WHERE` o `HAVING` de otra sentencia SQL. He aquí un ejemplo: *Listar el nif, nombre, apellidos y dirección de los clientes que tienen pedidos con importe superior a 100 €*. Inicialmente, la sentencia SELECT de la consulta anterior sería la siguiente:

```
SELECT nif, nombre, apellidos
FROM cliente
WHERE nif = ???;
```

Como se puede observar, se lista el `nif`, `nombre` y `apellidos` de los clientes cuyo `nif` es igual a algo (`???`). Ese algo tiene que ser alguno de los `nif` de los clientes que tengan pedidos con importe superior a 100 €, que se pueden obtener con la siguiente sentencia:

```
SELECT DISTINCT nif FROM pedido
WHERE total_pedido > 100;
```

Como se puede observar, con esta sentencia se obtendrían todos los `nif` de los clientes con pedidos cuyo importe fuera superior a 100 €. Este conjunto es al que tiene que pertenecer el `nif` de la primera sentencia. Para ello, ambas se pueden unir de la siguiente manera:

```
SELECT nif, nombre, apellidos
FROM cliente
WHERE nif IN (SELECT DISTINCT nif FROM pedido
              WHERE total_pedido > 100 )
```

La subconsulta está encerrada siempre entre paréntesis, pero tiene el formato similar a una sentencia `SELECT`, con una cláusula `FROM` y cláusulas opcionales `WHERE`, `GROUP BY` y `HAVING`. El formato de estas cláusulas es idéntico al que tienen en una sentencia `SELECT`, y efectúan sus funciones normales cuando se utilizan dentro de una subconsulta. Sin embargo, hay unas cuantas diferencias entre una subconsulta y una sentencia `SELECT` real.

- ✔ Una subconsulta debe producir una única columna de datos como resultados. Esto significa que una subconsulta siempre tiene un único elemento de selección en su cláusula `SELECT`.

- ✔ La cláusula `ORDER BY` no puede ser especificada en una subconsulta. Los resultados de la subconsulta se utilizan internamente por parte de la consulta principal y nunca son visibles al usuario, por lo que tiene poco sentido ordenarlas de ningún modo.
- ✔ Una subconsulta no puede ser la `UNION` de varias sentencias `SELECT` diferentes sólo se permite una única `SELECT`.
- ✔ Los nombres de columna que aparecen en una subconsulta pueden referirse a columnas de tablas en la consulta principal. Estas referencias externas se describen a continuación.

Dentro del cuerpo de una subconsulta, con frecuencia es necesario referirse al valor de una columna en la fila "actual" de la consulta principal. Consideremos el siguiente ejemplo: *Listar el nif de cliente, número de pedido y fecha de pedido cuyo total supere el total promedio de los pedidos del cliente.*

```
SELECT nif, npedido, fecha, total_pedido
FROM pedido AS P1
WHERE total_pedido > ( SELECT AVG(total_pedido)
                       FROM pedido AS P2
                       WHERE P2.nif = P1.nif )
```

El papel de la subconsulta es obtener el importe medio de los pedidos de un cliente en particular, específicamente, el cliente que está siendo examinado por la cláusula `WHERE` de la consulta principal. Sin embargo, se observa que `P1.nif` de la cláusula `WHERE` se refiere al de la fila actual en la consulta principal. Tanto en la consulta principal como en la subconsulta ha sido necesario utilizar un alias (`P1` y `P2`) para distinguir cuando un cliente pertenece a la fila actual en la consulta principal (`P1.nif`) y cuando pertenece a la subconsulta (`P2.nif`). Esto es debido a que ambos tienen el mismo nombre, pero si la subconsulta fuera sobre otra tabla diferente a `pedido`, no sería necesario, aunque si habría que especializar el nombre de cada columna con el nombre de tabla, salvo que las columnas con las que se establece la igualdad en la cláusula `WHERE` de la subconsulta no se llamaran de la misma manera.

Por ejemplo, supongamos que la columna `referencia` en la tabla `lpedido` se llamara `ref_art`. En esta situación formulamos la siguiente consulta:

```
-- Listar la referencia y descripción de los artículos
-- cuyo pvp es superior al precio medio de las líneas de pedido
-- de cada artículo
SELECT referencia, descripcion
FROM articulo
WHERE pvp > ( SELECT AVG(precio)
              FROM lpedido
              WHERE ref_art = referencia )
```

Como se puede apreciar, en la subconsulta se calcula la media de los artículos pedidos de una referencia concreta, aquella que está siendo examinada por la cláusula `WHERE` de la consulta principal. La subconsulta lo lleva a cabo explorando en la tabla `lpedido`. Pero en la cláusula `WHERE` de la subconsulta se hace referencia a la columna `referencia` que no

existe en la tabla `lpedido`. Esta columna es de la tabla `articulo`, que forma parte de la consulta principal. Conforme SQL recorre cada fila de la tabla `articulo`, utiliza el valor de la columna `referencia` de la fila actual cuando lleva a cabo una subconsulta.

Como la clave externa en la tabla `lpedido`, `ref_art` tiene diferente nombre que la clave primaria de la tabla `articulo`, `referencia`, no es necesario utilizar alias ni realizar especificación con el nombre de tabla de cada una.

Una subconsulta forma parte siempre de una condición de búsqueda en la cláusula `WHERE` o `HAVING`. Anteriormente se describió las condiciones de búsqueda simples que pueden ser utilizadas en estas cláusulas. Además, SQL ofrece estas condiciones de búsqueda de subconsultas.

2.9.1 Test de comparación de subconsulta (=, <>, <, <=, >, >=)

El test de comparación subconsulta es una forma modificada del test de comparación simple, tal como se muestra en el siguiente ejemplo. Compara el valor de una expresión con el valor producido por una subconsulta, y devuelve un resultado `TRUE` si la comparación es cierta. Este test se utiliza para comparar un valor de la fila que está siendo examinada con un valor único producido por una subconsulta.

```
-- Listar los clientes cuyas ventas superan el importe
-- del pedido número 5

SELECT *
FROM cliente
WHERE ventas > (   SELECT total_pedido
                   FROM pedido
                   WHERE npedido = 5);
```

La subconsulta recupera un solo valor, el total del pedido n.º 5. El valor se utiliza entonces para seleccionar los clientes cuyas ventas son superiores o iguales al total del pedido.

El test de comparación subconsulta ofrece los mismos seis operadores de comparación (`=, <>, <, <=, >, >=`) disponibles con el test de comparación simple. La subconsulta especificada en este test debe producir una única fila de resultados. Si la subconsulta produce múltiples filas, la comparación no tiene sentido, y SQL informa de una condición de error. Si la subconsulta no produce filas o produce un valor `NULL`, el test de comparación devuelve NULL.

2.9.2 Test de pertenencia a conjunto. Clausula IN

El test de pertenencia a conjunto de subconsulta es una forma modificada del test de pertenencia a conjunto simple. Compara un único valor de datos con una columna de valores producida por una subconsulta y devuelve un resultado `TRUE` si el valor coincide con uno de los valores de la columna. Este test se utiliza cuando se necesita comparar un valor de la fila que está siendo examinada con un conjunto de valores producidos por una subconsulta, como se muestra en los ejemplos.

```
-- Lista los artículos que se han vendido con un descuento
-- superior al 20%
SELECT *
FROM articuo
WHERE ref IN (SELECT referencia
              FROM lpedido
              WHERE dto > 0.2 );

-- Listar los clientes que no tienen pedidos con importe
-- superior a 100 €.

SELECT * FROM cliente
WHERE nif NOT IN ( SELECT DISTINCT nif
                   FROM pedido
                   WHERE total_pedido > 100 );
```

En cada uno de estos ejemplos, la subconsulta produce una columna de valores, y la cláusula `WHERE` de la consulta principal comprueba si un valor de una fila de la consulta principal coincide con uno de los valores de la columna. El formato de la subconsulta en el test `IN` funciona por tanto exactamente igual al del test `IN` simple, excepto que el conjunto de valores producido por una subconsulta en lugar de ser valores literales escritos explícitamente en la lista de la clausula `IN`.

2.9.3 Test de existencia. Clausula EXISTS

El test de existencia (operador `EXISTS`) comprueba si una subconsulta produce alguna fila de resultados. Solamente se utiliza con subconsultas. Por ejemplo

```
-- Listar los clientes que tienen al menos un pedido con importe
-- superior a 100 €.

SELECT *
FROM cliente
WHERE EXIST ( SELECT npedido
              FROM pedido
              WHERE pedido.nif = cliente.nif
                   AND total_pedido > 100 );
```

Conceptualmente, SQL procesa esta consulta recorriendo la tabla `cliente` y efectuando la subconsulta para cada cliente. La subconsulta produce una columna que contiene los datos de un aquellos pedidos del cliente actual con importe superior a 100 €. Si hay alguna de tales facturas (es decir, la subconsulta arrojó algún resultado), el test `EXISTS` devuelve `TRUE`. Si la subconsulta no produce filas, el test `EXISTS` es `FALSE`. El test `EXISTS` no puede producir un valor `NULL`.

Nótese en el ejemplo como se obtienen los pedidos del cliente que se examina en la consulta principal. Se emplea el campo `nif` que pertenece a la tabla `cliente` para restringir el resultado de la subconsulta a los pedidos del cliente que se está examinando. Como la columna `nif` está tanto en la tabla `cliente` como en la tabla `pedido`, para distinguir una de otra tenemos que cualificarla con el nombre de la tabla.

Se puede invertir la lógica del test `EXISTS` utilizando la forma `NOT EXISTS`. En este caso, el test es `TRUE` si la subconsulta no produce filas, y `FALSE` en caso contrario. Observe que la condición de búsqueda `EXISTS` no utiliza realmente los resultados de la subconsulta. Simplemente comprueba si la subconsulta produce algún resultado.

2.9.4 Test cuantificados. Clausulas ANY y ALL

La versión subconsulta del test `IN` comprueba si un valor de dato es igual a algún valor en columna de los resultados de una subconsulta. SQL proporciona dos test cuantificados, `ANY` y `ALL`, que extienden esta noción a otros operadores de comparación, tales como mayor que (`>`) y menor que (`<`). Ambos test comparan un valor de dato con la columna de valores producidos por una subconsulta.

El test `ANY` se utiliza conjuntamente con uno de los seis operadores de comparación SQL (`=, <>, <, <=, >, >=`) para comparar un único valor de test con una columna de valores producidos por una subconsulta. Para efectuar el test, SQL utiliza el operador de comparación especificado para comparar el valor de test con cada valor de datos en la columna, uno cada vez. Si alguna de las comparaciones individuales producen un resultado `TRUE`, el test `ANY`, devuelve un resultado `TRUE`. Por ejemplo

```
-- Listar los clientes que han hecho algún pedido cuyo total
-- importe supere el 10% de sus ventas.
SELECT *
FROM cliente
WHERE ( ventas * 0.1 ) > ANY ( SELECT total_pedido
                               FROM pedido
                               WHERE pedido.nif = cliente.nif );
```

La consulta principal examina cada fila de la tabla `cliente`, una a una. La subconsulta encuentra todos los pedidos hechos por el cliente actual y devuelve una columna que contiene el total de esos pedidos. La cláusula `WHERE` de la consulta principal calcula entonces el diez por ciento de las ventas del cliente actual y la utiliza como valor de test, comparándolo con todos los totales de pedidos producidos por la subconsulta. Si hay algún total de pedido que exceda al valor de test calculado, el test `> ANY` devuelve `TRUE` y el cliente queda incluido en los resultados de la consulta.

La palabra clave `SOME` es una alternativa para `ANY` especificada por el estándar SQL ANSI/ISO. Cualquiera de las dos palabras clave puede ser utilizada generalmente, pero algunos SGBD no la soportan.

El test `ALL` también se utiliza conjuntamente con uno de los seis operadores de comparación SQL (`=, <>, <, <=, >, >=`) para comparar un único valor de test con una columna de valores de datos producidos por una subconsulta. Para efectuar el test, SQL utiliza el operador de comparación especificado para comparar el valor de test con todos y cada uno de los valores de datos de la columna. Si todas las comparaciones individuales producen un resultado `TRUE`, el test `ALL` devuelve un resultado `TRUE`, en caso contrario devuelve `FALSE`. Por ejemplo

```
-- Listar el nif de proveedor y la referencia de artículos
-- de los artículos cuyo descuento de compra es superior
-- a todos los descuentos de venta del mismo artículo en los
-- en los pedidos

SELECT nif, referencia
FROM articulo_proveedor AS ap
WHERE dto_compra < ALL ( SELECT dto
                         FROM lpedido AS lp
                         WHERE lp.referencia = ap.referencia);
```

Conceptualmente, la consulta principal examina cada fila de la tabla `articulo_proveedor` una a una. La subconsulta encuentra todos los descuentos de las filas de la tabla lpedido cuya referencia coincide con la fila actual en la consulta principal y devuelve una columna que contiene ese descuento. La cláusula `WHERE` de la consulta principal compra el descuento de compra del artículo para compararlo con cada uno de los descuentos de venta obtenidos en la subconsulta. Si todos los descuentos de los pedidos superan el valor del test calculado, el test `< ALL` devuelve `TRUE` y el artículo juntoo con su proveedor se incluye en los resultados de la consulta. Si no, no se incluye en los resultados de la consulta.

2.9.5 Subconsultas anidadas

Todas las consultas descritas hasta ahora en este capitulo han sido consultas de dos niveles, afectando a una consulta principal y una subconsulta. Del mismo modo que se puede utilizar una subconsulta dentro de una consulta principal, se puede utilizar una subconsulta dentro de otra subconsulta. Por ejemplo

```
-- Listar los artículos de los pedidos realizados en el
-- mes de enero de 2023


SELECT *
FROM articulos
WHERE referencia IN (SELECT referencia
                     FROM lpedido
                     WHERE npedido IN ( SELECT npedido
                                        FROM pedido
                                        WHERE fecha BETWEEN
                                        '01/01/23' AND
                                        '31/01/23' )
                    )
```

En este ejemplo la subconsulta más interna produce una columna que contiene los números de pedido hechos durante el mes de enero de 2023. La subconsulta siguiente produce una columna que contiene las referencias de artículo de los artículos que figuran en alguno de los pedidos seleccionados. Finalmente, la consulta más externa encuentra los artículos cuyas referencias tienen uno de las referencias producidos en la subconsulta.

La misma técnica utilizada en esta consulta de tres niveles puede utilizarse para construir consultas con cuatro o más niveles. El estándar SQL ANSI/ISO no especifica un

número máximo de niveles de anidación, pero en la práctica una consulta consume mucho más tiempo cuando se incrementa el número de niveles. La consulta también resulta mucho más difícil de leer, comprender y mantener cuando contiene más de uno o dos niveles de subconsultas.

2.9.6 Subconsultas en clausula HAVING

Aunque las subconsultas suelen encontrarse sobre todo en la cláusula `WHERE`, también pueden utilizarse en la cláusula `HAVING` de una consulta. Cuando una subconsulta aparece en la cláusula `HAVING`, funciona como parte de la selección de grupo de filas efectuada por la cláusula `HAVING`. Por ejemplo

```
-- Listar los clientes cuyo importe medio de pedido es
-- superior al tamaño del importe medio de todos los pedidos

SELECT nif, AVG(total_pedido)
FROM pedidos
GROUP BY nif
HAVING AVG(total_pedido) > ( SELECT AVG(total_pedido)
                             FROM pedidos );
```

La subconsulta calcula la media de todos los totales de pedido. La consulta principal recorre la tabla `pedidos`, hallando todos los pedidos y agrupándolos por cliente. La cláusula `HAVING` comprueba entonces cada grupo de filas para ver si la media del total de pedido de ese grupo es superior al promerio de todos los pedidos, calculado con antelación. Si es así, el grupo de filas es retenido; si no, el grupo de filas es descartado. Finalmente la cláusula `SELECT` produce una fila sumaria por cada grupo, mostrando el nif del cliente y el importe medio de pedido para cada uno.

2.10 Consultas multitabla

El posible que para consultas sencillas, todos los datos que necesítemos listar estén en una sola tabla. Pero... ¿y si están repartidos por una, dos o más tablas? Es posible hacer consultas que incluyan más de una tabla dentro de la cláusula `FROM`, pero en estas consultas hay que tener en cuenta ciertos factores. Para ilustrar esto supongamos que las tablas `cliente` y `pedido` tienen el siguiente estado:

Tabla `cliente`

nif	nombre	apellidos	...	email	ventas
300001A	José	Gómez López		jose@correo.es	1545.50
300002B	Juan	Martínez Salazar		juan@correo.es	2450.15
300003C	Antonio	Caracol Bueno		tony@correo.es	3840.25

Tabla `pedido`

npedido	nif	fecha	observaciones	total_pedido
111	300001A	15/12/2022		155.45
112	300002B	17/12/2022		75.90

113	300003C	17/12/2022		41.20
114	300003C	18/12/2022		35.75
115	300001A	19/12/2022		110.95

Sobre las dos tablas anteriores ejecutamos la siguiente consulta:

```
SELECT npedido, pedido.nif, cliente.nif, nombre, importe
FROM pedido, cliente;
```

El resultado de la consulta será el siguiente

npedido	pedido.nif	cliente.nif	nombre	importe
111	**300001A**	**300001A**	**José**	**155.45**
111	300001A	300002B	Juan	155.45
111	300001A	300003C	Antonio	155.45
112	300002B	300001A	José	75.90
112	**300002B**	**300002B**	**Juan**	**75.90**
112	300002B	300003C	Antonio	75.90
113	300003C	300001A	José	41.20
113	300003C	300002B	Juan	41.20
113	**300003C**	**300003C**	**Antonio**	**41.20**
114	300003C	300001A	José	35.75
114	300003C	300002B	Juan	35.75
A114	**300003C**	**300003C**	**Antonio**	**35.75**
A115	**300001A**	**300001A**	**José**	**110.95**
A115	300001A	300002B	Juan	110.95
A115	300001A	300003C	Antonio	110.95

Podemos ver que el resultado es el producto cartesiano de una tabla por otra tabla, es decir, todas las combinaciones posibles de la tabla `pedido` con la tabla `cliente`. Pero en realidad lo que a nosotros nos interesa es mostrar todas los pedidos, pero con la cliente de cada pedido, es decir, que cada pedido seleccione sólo su fila correspondiente de la tabla `cliente`. Las filas que a nosotros nos interesan están marcados en negrita en el esquema anterior, y en todos ellos se cumple que `pedido.nif = cliente.nif`, o dicho de otro modo, los campos que componen la relación igualados.

Entonces del resultado anterior, sólo nos interesan los registros marcados en negrita, y la sentencia `SELECT` que nos retorna ese resultado es:

```
SELECT npedido, pedido.nif, cliente.nif, nombre, importe
FROM pedido, cliente
WHERE pedido.nif = cliente.nif;
```

Nótese que cuando se listan columnas de diferentes tablas, pero que tienen el mismo nombre, es necesario cualificarlas con el nombre de la tabla a la que pertenecen. El resultado final es:

npedido	pedido.nif	cliente.nif	nombre	importe
111	**300001A**	**300001A**	**José**	**155.45**
112	**300002B**	**300002B**	**Juan**	**75.90**
113	**300003C**	**300003C**	**Antonio**	**41.20**
A114	**300003C**	**300003C**	**Antonio**	**35.75**
A115	**300001A**	**300001A**	**José**	**110.95**

Como norma general se puede decir que para combinar dos o más tablas estas se encuentran relacionadas mediante una clave externa y hay que poner como condición **la igualdad entre la clave primaria de una tabla y la clave externa de la otra**. Las condiciones dentro del `WHERE` que sirven para hacer el enlace entre tablas se denominan **JOIN** (unión, enlace).

2.10.1 Consultas multitabla con criterio de selección de fila

La condición de búsqueda que especifica las columnas de emparejamiento en una consulta multitabla puede combinarse con otras condiciones de búsqueda para restringir aún más los contenidos de los resultados. Supongamos que se desea ejecutar la cosulta anterior, pero mostrando únicamente los pedidos cuyo total superan los 100 €.

```
SELECT npedido, pedido.nif, cliente.nif, nombre, importe
FROM pedido, cliente
WHERE pedido.nif = cliente.nif
 AND total_pedido > 100;
```

Con la condición de búsqueda adicional, las filas que aparecen en los resultados están aún más restringidas. El primer test (`pedido.nif = cliente.nif`) selecciona solamente los pares de filas `pedido` y `cliente` que tienen la adecuada relación padre/hijo; el segundo test selecciona adicionalmente sólo aquellos pares de filas en donde el total del pedido está por encima de 100.

2.10.2 Múltiples columnas de emparejamiento

Centrémonos en las tablas `lenvio` y `lpedido` de nuestro modelo relacional de ejemplo. En la tabla `lenvio` hay una clave externa formada por dos columnas (`npedido`, `nlinea`) que referencia a la tabla `lpedido`. Para componer las tablas basándose en esta relación, deben especificarse ambos pares de columnas de emparejamiento, tal y como muestra este ejemplo:

```
-- Listar el número de envío, el número de pedido,
-- el número de línea de pedido, las unidades de las
-- líneas de envío y las unidades de
-- las líneas de pedido correspodientes.
-- Comprobar si se han enviado todas las unidades que se
-- pidieron solo para aquellas líneas de pedido son
-- superiores a 10

SELECT lenvio.nenvio, lenvio.npedido, lenvio.nlinea,
    lenvio.unidades, lpedido.unidades
FROM lenvio, lpedido
WHERE lenvio.npedido = lpedido.npedido
AND lenvio.nlinea = lpedido.nlinea
AND lpedido.unidades > 10;
```

La condición de búsqueda en la consulta dice a SQL que los pares de filas relacionados en las tablas `lenvio` y `lpedido` son aquellas en las que ambos pares de columnas coincidentes contienen los mismos valores, ya que la clave externa de la tabla `lenvio` está formada por la agregación de las columnas `npedido` y `nlinea`, la cual referencia a la clave principal de la tabla `lpedido`, que a su vez está formada por la agregación de las columnas `npedido` y `nlinea`.

2.10.3 Consultas de tres o más tablas

SQL puede combinar datos de tres o más tablas utilizando las mismas técnicas básicas utilizadas para las consultas de dos tablas. Veamos el siguiente ejemplo

```
-- Listar el número de envío, fecha, forma de envío,
-- descripción de la forma de envío y su coste,
-- línea de envío, referencia del artículo enviado y
-- unidades enviadas

SELECT envio.nenvio, fecha, forma_envio, descripcion,
  coste, nlinea, referencia, descripcion, unidades
FROM envio, forma_envio, lenvio
WHERE envio.nenvio = lenvio.nenvio
AND envio.forma_envio = forma_envio.id_fe;
```

Esta consulta usa dos claves externas: una en la tabla `lenvio` con `envio` y otra en la tabla `envio` con `forma_envio`. La columna `nenvio` es una clave externa para la tabla `envio`, que enlaza cada línea de envío con su correspondiente envío. Luego tenemos la columna `forma_envio` de la tabla `envio`, que es clave externa a la tabla `forma_envio`.

2.10.4 Join externo

Existe un caso especial cuando se establece un join entre tablas: *el outer join*. Este caso se da cuando los valores de los campos enlazados en alguna de las tablas, contiene el valor `NULL`. Al realizar un join, si algún campo enlazado contiene el valor `NULL`, ese registro quedará automáticamente excluido, ya que una condición en la que un operando sea `NULL` siempre se evalúa como falso.

Pensemos por ejemplo en la tabla cliente y pedido de nuestro ejemplo. Si ejecutamos la siguiente consulta

```
-- Listar el nif de cliente, nombre, apellidos, número de pedido
y fecha de todos los pedidos

SELECT cliente.nif, nombre, apellidos, npedido, fecha
FROM pedido, cliente
WHERE pedido.nif = cliente.nif
```

Al ejecutar la consulta anterior (los clientes y sus pedidos), no aparecerán los clientes que no hayan hecho ningún pedido, ya que no estarán emparejados con ningún pedido. Habrá clientes que no hayan hecho ningún pedido y por tanto su `nif` no aparecerá en ningún campo `cliente` de la tabla `pedido`. Al evaluar la condición de join (`WHERE pedido.nif = cliente.nif`), no se evaluará como verdadero.

Además, hay otro caso en los que no se producirá emparejamiento, que es cuando el valor de una clave externa es NULL. Por ejemplo, en la siguiente consulta queremos obtener los pedidos y la forma de pago que tienen.

```
-- Listar el número de factura, fecha, nenvio y nif
-- de cliente
SELECT nfactura, fecha, nenvio, nif
FROM factura, envio
WHERE factura.nenvio = factura.nenvio
```

Habrá facturas que tengan el valor de la columna `nenvio` a `NULL` ya que podría ser una factura correspondiente a una devolución. Con la condición `WHERE` anterior tendremos que esas facturas no pueden emparejarse con ningún envío al tener el valor de la clave externa `nenvio` a `NULL`.

¿Qué ocurre entonces con las filas de una tabla que en una consulta multitabla no se pueden emparejar con las filas de otras tablas? En el caso de que también queramos visualizarla debemos utilizar un outer join (join externo), que es un JOIN pero indicando que queremos considerar aquellos registros que se descartan por existencia de valores nulos en las columnas de emparejamiento. Dependiendo de la procedencia de los datos que queremos incluir en el resultado de la consulta existen tres tipos de join externo:

- ✔ Join externo completo (FULL OUTER JOIN).-. Se indica que el resultado debe incluir tantas filas como se puedan combinar con las dos tablas (join interno) más una fila por cada fila de la primera tabla que no se ha podido emparejar con la segunda tabla y más una fila por cada fila de la segunda tabla que no se ha podido emparejar con la primera tabla.
- ✔ Join externo izquierdo (LEFT OUTER JOIN).-. Se indica que el resultado debe incluir tantas filas como se puedan combinar con las dos tablas (join interno) más una fila por cada fila de la primera tabla (a la izquierda) que no se ha podido emparejar con la segunda tabla.
- ✔ Join externo derecho (RIGTH OUTER JOIN).-. Se indica que el resultado debe incluir

tantas filas como se puedan combinar con las dos tablas (join interno) más una fila por cada fila de la segunda tabla (a la derecha) que no se ha podido emparejar con la primera tabla.

Cuando se muestren los resultados de cada JOIN aquellos campos correspondiente a la tabla que no se han podido incluir aparecerán con su valor a `NULL`. Veamos un ejemplo de cada caso.

```
-- Listar el número de factura, fecha, nenvio y nif
-- de cliente de todas las facturas

SELECT nfactura, fecha, nenvio, nif
FROM factura
FULL OUTER JOIN envio ON nfactura.nenvio = envio.nenvio
ORDER BY nfactura
```

Como se puede observar, las primeras filas con valores en todas sus columnas son resultado de combinar las dos tablas. Sin embargo, algunas facturas que se basan en envíos no se han podido combinar con ninguna fila de la tabla `envio`, con lo que los valores de estas columnas en el resultado son `NULL`. Las últimas filas del resultado son filas de la segunda tabla (`envio`) que no se han podido combinar con ninguna de la primera (`factura`), y por tanto los valores de las columnas pertenecientes a la primera son `NULL`.

Si quisiéramos hacer un join externo por la izquierda, es decir, que el resultado tenga todas las posibles combinaciones de las dos tablas y además las filas de la tabla `factura` que no han podido emparejarse. La sentencia sería:

```
-- Listar el número de factura, fecha, nenvio y nif
-- de cliente de todas las facturas

SELECT nfactura, fecha, nenvio, nif
FROM factura
LEFT OUTER JOIN envio ON nfactura.nenvio = envio.nenvio
ORDER BY nfactura
```

El resultado es similar al anterior, excepto en que las ultimas filas no aparecen al añadir solamente las filas de la tabla de la izquierda.

De forma análoga, si quisiéramos hacer un join externo por la derecha, es decir, que el resultado tenga todas las posibles combinaciones de las dos tablas y además las filas de la tabla `envio` que no han podido emparejarse, la sentencia sería:

```
-- Listar el número de factura, fecha, nenvio y nif
-- de cliente de todas las facturas

SELECT nfactura, fecha, nenvio, nif
FROM factura
RIGHT OUTER JOIN envio ON nfactura.nenvio = envio.nenvio
ORDER BY nfactura
```

Como se puede observar, las primeras filas son resultado de combinar las dos tablas y

las últimas corresponden a filas de la tabla `envio` (derecha) que no se han podido combinar con ninguna de la tabla `factura` y por eso las columnas de esta tabla aparecen con valores nulos. Ahora las facturas que no se combinan con envios no aparecen al no haber podido combinar con ningún envío.

Por último, indicar que una consulta SQL con un join interno puede realizarse utilizando una sintaxis similar a la del join externo. Consiste en quitar la segunda tabla de la cláusula `FROM` y ponerla en una cláusula `INNER JOIN`. La condición de enlace se elimina de la cláusula `WHERE` y se añadiría a una cláusula `ON`. Veamos el siguiente ejemplo

```
-- Listar la referencia de artículo, descripción,
-- precio de venta y descripción de categoría
-- de los artículos cuyo precio de venta supera un euro

SELECT referencia, articulo.descripcion, pvp,
articulo.categoria, categoria.descripcion
FROM articulo
INNER JOIN categoria ON articulo.categoria = categoria.categoria
WHERE pvp > 1;
```

2.10.5 Cláusula USING

En el apartado anterior hemos visto la forma de realizar consultas multitabla o joins en las que se emplea una clave externa y una clave primaria para emparejar las filas de ambas tablas. En el caso de que las columnas de emparejamiento tengan el mismo nombre podemos emplear la cláusula `USING` en lugar de utilizar la cláusula `ON`. En este caso no es necesario cualificar las columnas de emparejamiento aunque tengan el mismo nombre. Si repetimos el ejemplo anterior tendríamos lo siguiente:

```
-- Listar la referencia de artículo, descripción,
-- precio de venta y descripción de categoría
-- de los artículos cuyo precio de venta supera un euro

SELECT referencia, articulo.descripcion, pvp,
articulo.categoria, categoria.descripcion
FROM articulo
INNER JOIN categoria USING (categoria)
WHERE pvp > 1;
```

Vemos en el ejemplo anterior que la columna `categoría` es empleada para hacer el join entre la tabla `artículo` y `categoría`. En la tabla `artículo` la columna `categoría` es clave externa a la tabla `categoría` que tiene como clave primaria la columna `categoría`. Al tener el mismo nombre la clave primaria y la clave externa que se emplean para emparejar las filas podemos usar `USING` que especifica la columna de emparejamiento en ambas tablas. En este caso no es necesario cualificar la columna `categoría` en la lista de columnas, de lo contrario resultaría un error. El resto de columnas que pudiera tener un nombre en común en ambas tablas si tiene que cualificarse con el nombre de la tabla.

Podemos usar la cláusula `USING` en cualquier JOIN, tanto interno como externo.

3 Bibliografía

KRISHNAMURTHY, Usha, *Oracle Database Express Edition – SQL Reference Guide – 21c Release*. ORACLE 2022

MOORE, Sheila, *Oracle Database Express Edition – 2 Day Developer's Guide – 11g Release*. ORACLE 2014