
UD6: MODELO FÍSICO DQL

Parte 3. DQL Avanzado

Tema Extendido

Bases de Datos (BD) CFGs DAM/DAW

Abelardo Martínez y Pau Miñana.

Basado y modificado de Sergio Badal y Raquel Torres.


Curso 2023-2024

ÍNDICE

- [1. Consultas reflexivas](#)
- [2. Subconsultas](#)
 - [2.1. Subconsultas en la proyección](#)
 - [2.2. Subconsultas en el filtrado \(WHERE/HAVING\)](#)
 - [2.2.1. Operadores para filtrar con varias filas](#)
 - [\[NOT\] IN](#)
 - [ALL](#)
 - [ANY/SOME](#)
 - [\[NOT\] EXISTS](#)
 - [2.3. Subconsultas en el FROM. Tablas derivadas](#)
 - [2.4. DQL EN INSTRUCCIONES DML](#)
 - [2.4.1. INSERT](#)
 - [2.4.2. UPDATE](#)
 - [2.4.3. DELETE](#)
- [3. ORDER BY y LIMIT 1 para calcular máximos y mínimos](#)
- [4. Uniones](#)
- [5. Vistas](#)
- [6. Bibliografía](#)

1. Consultas reflexivas

Las consultas reflexivas involucran varias veces a la misma tabla en la consulta. No suelen ser muy frecuentes, pero aparecen cuando existen relaciones reflexivas en nuestro modelo de datos o en ciertas consultas donde se necesitan varias copias de la misma tabla para poder realizar el filtrado correctamente.

 Para poder usar **varias copias de una tabla** simplemente **se añade varias veces al FROM**, pero es imprescindible ponerles **alias** para poder identificar a cada una.

Veamos algunos ejemplos.

Ejemplo 1

Puesto que las BD que estamos usando como ejemplo no tienen relaciones reflexivas, vamos a crear una tabla que las tenga. Vamos a añadir a la tabla de *empleados* una columna *supervisor*, que será una clave foránea que haga referencia al *dni* del propio empleado. De esta forma, un empleado puede ser el supervisor de otro (1:N Reflexiva). Además actualizamos los datos para que Ana Silván sea la supervisora de Mariano Sanz, ambos del departamento de informática.

```
mysql> ALTER TABLE empleados ADD COLUMN supervisor VARCHAR(10);
Query OK, 0 rows affected (0,04 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> ALTER TABLE empleados ADD CONSTRAINT emp_sup_fk
-> FOREIGN KEY(supervisor) REFERENCES empleados(dni);
Query OK, 6 rows affected (0,11 sec)
Records: 6 Duplicates: 0 Warnings: 0

mysql> UPDATE empleados
-> SET supervisor = '45678901D'
-> WHERE dni = '23456789B';
Query OK, 1 row affected (0,00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Para mostrar los datos de los empleados del departamento de informática junto con el dni de su supervisor no hay problema, todos esos datos están en la misma fila de la tabla... ¿pero qué pasa si queremos mostrar los datos del supervisor también? Estos están en otra fila de la tabla por lo que no podemos mostrarlos juntos, de hecho, si sólo se muestran empleados que tengan supervisor esa fila

incluso se eliminará en el filtro. Esto significa que necesitamos **usar la tabla empleados dos veces**, una para obtener la información del empleado y otra para obtener la información de los supervisores.

Supongamos que queremos **mostrar el nombre del empleado y el nombre de su supervisor para los empleados del departamento de informática**. Por ello, asignamos el alias **E** (*empleado*) a la primera copia y **S** (*supervisor*) a la segunda; de este modo las podemos tratar como si fueran dos tablas distintas. Una vez llegados a esto la consulta es igual que si se tratase de dos tablas distintas con una clave ajena relacionándolas, simplemente se añade la condición de que el campo *supervisor* del *empleado(E)* sea igual al *dni* de *supervisor(S)*, para poder seleccionar el *nombre* en esa fila de la copia de la tabla.

```
mysql> SELECT E.nombre_emp AS Empleado, S.nombre_emp AS Supervisor
-> FROM empleados E
-> INNER JOIN empleados S ON E.supervisor = S.dni
-> WHERE E.dpt = 'INF'
-> ORDER BY E.nombre_emp;
+-----+-----+
| Empleado | Supervisor |
+-----+-----+
| Mariano Sanz | Ana Silván |
+-----+-----+
1 row in set (0,00 sec)
```

Ejemplo 2

Supongamos ahora que necesitamos **mostrar los nombres de los empleados en el mismo departamento que Ana Silván**. De nuevo, parece una consulta trivial, pero eso es porque ya conocemos que su departamento es 'INF'.

No podemos dar por supuesto que conocemos la información en la base de datos para cambiar los parámetros de la consulta a nuestro antojo. Por ello, necesitamos **usar la tabla empleados dos veces**, una para obtener la información de 'Ana Silván', en este caso su departamento, y otra para obtener los nombres de los empleados de ese departamento, puesto que sus filas en la primera copia se han perdido en el filtrado.


En este caso no hay una clave ajena para relacionar las 2 copias, la condición de unión de las 2 tablas es el campo *dpt*, ya que buscamos a los empleados del mismo departamento.

```
mysql> SELECT E2.nombre_emp FROM empleados E1
      -> INNER JOIN empleados E2 ON E1.dpt = E2.dpt
      -> WHERE E1.nombre_emp = 'Ana Silván';
+-----+
| nombre_emp |
+-----+
| Mariano Sanz |
| Ana Silván   |
+-----+
2 rows in set (0,00 sec)
```

En caso de querer mostrar solo a los compañeros y no a la propia 'Ana Silván', se puede añadir también esa condición al WHERE.

```
mysql> SELECT E2.nombre_emp FROM empleados E1
      -> INNER JOIN empleados E2 ON E1.dpt = E2.dpt
      -> WHERE E1.nombre_emp = 'Ana Silván'
      ->       AND E2.nombre_emp <> 'Ana Silván';
+-----+
| nombre_emp |
+-----+
| Mariano Sanz |
+-----+
1 row in set (0,00 sec)
```

2. Subconsultas

 Una subconsulta es una instrucción **SELECT** anidada dentro de otra consulta o DML (INSERT, UPDATE o DELETE). Se escribe cada subconsulta entre paréntesis y pueden usarse en la proyección, en el FROM y en el filtrado (WHERE/HAVING) de las consultas.

Las subconsultas son muy útiles para realizar consultas más complejas, puesto que permiten realizar operaciones sobre los resultados de otras consultas, por ejemplo, de este modo se pueden por fin encadenar funciones agregadas. Además resultan muy versátiles, puesto que pueden contener campos de las tablas de la consulta en la que estén incluidas e incluso se pueden anidar, es decir, una subconsulta puede contener a su vez otra subconsulta.

Existen 2 tipos de subconsultas que funcionan de modo diferente:

- **Correlacionadas:** *Usan campos de la consulta principal*, por tanto, *se ejecutan una vez por cada fila* de la consulta principal, ya que su resultado depende de los valores de la fila sobre la que se ejecutan.

- **No correlacionadas:** *Son independientes de la consulta principal.* Son las más comunes y *se ejecutan una única vez* para toda la consulta, al no depender de los valores de esta.

La eficiencia de las subconsultas correlacionadas es mucho menor que la de las no correlacionadas, pero otorgan una gran flexibilidad para diseñar consultas y resultan imprescindibles en ciertos casos.

2.1. Subconsultas en la proyección

Se pueden incluir subconsultas en la proyección para mostrar el resultado de una operación sobre los datos de la tabla u otras tablas. Por ejemplo es muy típico usarlas para mostrar u operar con el resultado de una función agregada, como la suma, la media, el máximo o el mínimo de una columna.

 Las subconsultas en la proyección **deben devolver una única columna y un solo valor (fila).**

Esto resulta evidente, puesto que el resultado de la subconsulta se mostrará en una columna de la respuesta de la consulta, y por tanto no puede mostrar varias columnas a la vez ni devolver más de un valor por fila. La excepción a esto es cuando se anidan consultas, mientras la consulta incluida directamente en la proyección debe cumplir esta restricción las anidadas podrían devolver varias filas y columnas, siempre que no estén también en la proyección de otra subconsulta.

Para comprender como funcionan las subconsultas lo mejor es analizar algunos ejemplos.

Ejemplo 1: Subconsulta no correlacionada

Supongamos que queremos el **nombre de un producto, su precio y la diferencia entre ese precio y el precio medio de los productos.**

Para calcular el precio medio emplearemos una subconsulta:

```
SELECT AVG(precio) FROM producto;
```

El uso de alias en esta consulta es opcional. Aunque se use la tabla producto 2 veces, como la subconsulta no se relaciona con la consulta principal (no correlacionada) no hay ambigüedad y los campos se pueden usar sin prefijos, por lo que el alias no le aporta nada al SGBD. Sin embargo, es recomendable usar alias siempre que se use una tabla más de una vez en una consulta, puesto que facilita la identificación de a qué tabla se refiere cada campo.

Aunque no es necesario, tampoco es mala idea usar alias y prefijos siempre que se usen distintas tablas en las consultas para facilitar la identificación de los campos, aunque no entren en conflicto.

```
mysql> SELECT P1.nombre_prod, P1.precio,
-> P1.precio - (SELECT AVG(P2.precio)
-> FROM producto P2) AS Diferencia
-> FROM producto P1;
```

nombre_prod	precio	Diferencia
AVION FK20	31.75	12.857143
BOLA BOOM	22.20	3.307143
HOOP MUSICAL	12.80	-6.092857
NAIPES PETER PARKER	3.00	-15.892857
PATINETE 3 RUEDAS	22.50	3.607143
PELUCHE MAYA	15.00	-3.892857
PETER PAN	25.00	6.107143

7 rows in set (0,00 sec)

Es muy importante comprender el proceso interno que sigue el SGBD para realizar las consultas y subconsultas, sobretodo para comprender el funcionamiento de las subconsultas correlacionadas o para cuando se combinan con agrupamientos y anidaciones. El proceso que sigue el SGBD para resolver esta consulta es el siguiente:

1. *Resuelve la subconsulta* obteniendo el precio medio de los productos.
 1. *Resuelve el FROM*, obteniendo la tabla producto para la subconsulta.
 2. Puesto que *no hay condiciones en el WHERE*, se *resuelve la proyección*, obteniendo el precio medio de los productos.
 3. Ahora este valor está disponible para la consulta principal sin calcularse de nuevo.
2. *Resuelve la consulta principal, empieza por el *FROM*, obteniendo la tabla producto.
3. Puesto que *no hay condiciones en el WHERE*, se *resuelve la proyección*, mostrando el nombre, el precio y calculando la Diferencia.

Ejemplo 2: Subconsulta correlacionada

Supongamos ahora que deseamos *mostrar la referencia, el nombre y las unidades vendidas de cada producto ordenados por la referencia del producto*. Para calcular las unidades vendidas emplearemos una subconsulta correlacionada que sume la cantidad del producto *ref_prod* de la fila en la que se ejecuta:

```
mysql> SELECT P.ref_prod, P.nombre_prod,
->      (SELECT SUM(D.Cantidad)
->      FROM detalle_pedido D
->      WHERE D.ref_prod = P.ref_prod) AS ud_vendidas
-> FROM producto P
-> ORDER BY P.ref_prod;
```

ref_prod	nombre_prod	ud_vendidas
AFK11	AVION FK20	42
BB75	BOLA BOOM	17
HM12	HOOP MUSICAL	10
NPP10	NAIPES PETER PARKER	13
P3R20	PATINETE 3 RUEDAS	43
PM30	PELUCHE MAYA	20
PT50	PETER PAN	NULL

7 rows in set (0,00 sec)

El producto 'PT50' no tiene resultado porque no está en ningún pedido; es decir, nunca se ha pedido. Un caso así no devuelve 0 ya que esa fila se elimina en el JOIN.

Se puede observar que la subconsulta calcula la suma de la cantidad de productos pedidos sobre la tabla *detalle_pedido* filtrando que **la ref_prod sea la misma que en la tabla de la consulta principal**, por tanto, la subconsulta es correlacionada. Se ejecuta una vez en cada fila de la consulta principal, ya que la suma de la subconsulta cambia según el *P.ref_prod* de la fila en la que se ejecuta.

⚠ Los campos de la consulta principal pueden ser usados en la subconsulta, pero nunca al revés.

Es decir, en este caso se puede usar la tabla *producto (P)* en la subconsulta, pero no se puede usar la tabla *detalle_pedido (D)* en la consulta principal.

Como tenemos campos con el mismo nombre, en este caso la referencia del producto en las dos tablas que vamos a manejar, **estos campos necesitan el prefijo de la tabla** a la que pertenecen para evitar ambigüedades. Al ser dos tablas distintas, sigue siendo opcional, aunque muy recomendable, el uso de alias para las tablas.

⚠ En consultas correlacionadas el uso de alias es imprescindible si se incluye la misma tabla en la consulta principal y en la subconsulta.

El proceso que sigue el SGBD para resolver esta consulta es el siguiente:

1. *Resuelve el FROM*, obteniendo la tabla producto para la consulta principal.
2. *Selecciona la primera fila* de la tabla producto (AFK11).
3. *Resuelve la proyección*, mostrando referencia y nombre del producto. Entonces se *encuentra la subconsulta correlacionada*.
4. *Resuelve la subconsulta* obteniendo las ud_vendidas del producto concreto para mostrar en la proyección.
 1. *Resuelve el FROM*, obteniendo la tabla detalle_pedido para la subconsulta.
 2. *Resuelve el WHERE*, se queda solo las filas con la *ref_prod igual a la de la consulta principal* (AFK11).
 3. *Resuelve la proyección*, sumando las cantidades.
5. *Repite los pasos 2-4 para cada fila* de la tabla producto (*P.ref_prod cambia*, luego la subconsulta devuelve otro valor).

En estos casos la eficiencia de la consulta es mucho menor que en las subconsultas no correlacionadas, puesto que la subconsulta se ejecuta una vez por cada fila de la consulta principal, lo que puede resultar muy costoso si la tabla es grande. En este caso no hay problema, puesto que la tabla producto es pequeña, pero en tablas grandes se debe tener en cuenta. Por ejemplo si la tabla del FROM tuviese 1000 filas, la subconsulta se ejecutaría 1000 veces.

En ocasiones se puede mejorar la eficiencia substituyendo subconsultas correlacionadas por JOINS, agrupamientos o tablas derivadas, que se introducen más adelante en esta unidad.



Muchas de las consultas que incluyen subconsultas pueden reformularse usando únicamente JOINS, aumentando su eficiencia. Muchas veces se usan subconsultas por desconocimiento de los JOINS, comodidad o por costumbre, pero en realidad los JOINS suelen ser más eficientes y claros.

De todos modos, los SGDB actuales suelen optimizar las consultas y pueden reescribirlas internamente para que sean más eficientes.

Aquí tienes más información, aunque es mejor consultarla tras haber comprendido mejor las distintas subconsultas:

<https://styde.net/uso-de-joins-versus-subconsultas-en-bases-de-datos-mysql/>

Como ejemplo, la consulta anterior se ha usado para mostrar una consulta correlacionada de estructura simple, pero es completamente innecesaria. Se puede reformular usando un JOIN y un agrupamiento:

```
SELECT P.ref_prod, P.nombre_prod, SUM(D.Cantidad) AS ud_vendidas
FROM producto P
LEFT JOIN detalle_pedido D ON P.ref_prod = D.ref_prod
GROUP BY P.ref_prod
ORDER BY P.ref_prod;
```


Esta consulta es bastante más eficiente que la anterior, puesto que los JOINS son mucho más rápidos que las consultas correlacionadas y el agrupamiento solo se realiza una vez. Además, es más clara y fácil de entender.

Se usa LEFT JOIN para mantener exactamente la misma respuesta, puesto que si se usa INNER JOIN no se mostraría el producto 'PT50' que no tiene pedidos y se elimina en el INNER JOIN al no cumplir la condición en el ON; en cambio con LEFT JOIN se mantienen todos los productos.

2.2. Subconsultas en el filtrado (WHERE/HAVING)

Las subconsultas en el filtrado se suelen emplear cuando los datos de la condición que estamos estableciendo no los tenemos a priori y es necesario obtenerlos de la base de datos a través de una subconsulta.

Las subconsultas en el filtrado pueden devolver varias columnas y filas pero lo más habitual es que devuelvan una sola columna y un solo valor. En los siguientes apartados se introducirán los operadores que permiten filtrar con subconsultas que devuelvan varias filas.

 Las subconsultas en filtros con operadores relacionales ($=$, $<$, $>$, $<=$ y $>=$) deben devolver un solo valor (fila).

Ejemplo 1

Supongamos que queremos mostrar todos los productos cuyo precio es mayor de 15 euros, la condición sería "Precio > 15". Sin embargo, si queremos **mostrar todos los artículos cuyo precio es mayor que la media de los precios de los artículos** la cosa cambia, ya que no tenemos ese dato en ninguna tabla. Hay que calcularlo y, para eso, emplearemos una subconsulta. La condición sería:

```
precio > (SELECT AVG(Precio) FROM producto)
```

La consulta completa que nos mostraría el nombre de los artículos y su precio siempre que éste se encuentre por encima del precio medio de los productos que tenemos y ordenado por el nombre del artículo sería:

```
mysql> SELECT P1.nombre_prod, P1.precio
-> FROM producto P1
-> WHERE P1.precio > (SELECT AVG(P2.precio) FROM producto P2)
-> ORDER BY P1.nombre_prod;
+-----+-----+
| nombre_prod | precio |
+-----+-----+
| AVION FK20   | 31.75  |
| BOLA BOOM    | 22.20  |
| PATINETE 3 RUEDAS | 22.50  |
| PETER PAN    | 25.00  |
+-----+-----+
4 rows in set (0,01 sec)
```

Recuerda que la subconsulta va entre paréntesis y solamente puede devolver un valor (fila).

Ejemplo 2:

Supongamos que queremos **mostrar la referencia y cantidad de ventas de los artículos que han vendido más unidades que el producto 'PM30'**. Primero necesitamos una subconsulta para obtener las unidades vendidas de 'PM30' y después filtrar los productos que han vendido más que esa cantidad.

```
SELECT SUM(cantidad) FROM detalle_pedido WHERE ref_prod = 'PM30';
```


Pero en este caso el resultado de la subconsulta debe compararse con el total de unidades vendidas de cada producto, por tanto, debe estar en el HAVING, no en el WHERE. La consulta completa sería:

```
mysql> SELECT D1.ref_prod, SUM(D1.cantidad) AS ud_vendidas
-> FROM detalle_pedido D1
-> GROUP BY D1.ref_prod
-> HAVING SUM(D1.cantidad) > (SELECT SUM(D2.cantidad)
-> FROM detalle_pedido D2
-> WHERE D2.ref_prod = 'PM30');
+-----+-----+
| ref_prod | ud_vendidas |
+-----+-----+
| AFK11    | 42          |
| P3R20    | 43          |
+-----+-----+
2 rows in set (0,00 sec)
```

Recuerda que este tipo de subconsultas no correlacionadas se ejecutan una única vez para toda la consulta y no afectan mucho a la eficiencia de la misma.

2.2.1. Operadores para filtrar con varias filas

En el filtrado de las consultas se pueden usar subconsultas que devuelvan varias filas, pero para ello se deben usar operadores especiales que permitan comparar un valor con varios valores a la vez.

 Generalmente en subconsultas que devuelven varias filas es muy importante que estas no se repitan, puesto que solo ralentizan la comparación. Por ello, se añade **DISTINCT** cuando puede haber repeticiones.

[NOT] IN

Comprueba si un valor está o no en un conjunto. Este conjunto puede obtenerse de una subconsulta que devuelva varias filas.


Por ejemplo, si queremos **mostrar los productos que no se han vendido nunca**, podemos usar la siguiente consulta:

```
mysql> SELECT ref_prod, nombre_prod
-> FROM producto
-> WHERE ref_prod NOT IN (SELECT DISTINCT ref_prod
->                        FROM detalle_pedido);
```

ref_prod	nombre_prod
PT50	PETER PAN

1 row in set (0,00 sec)

La subconsulta devuelve el conjunto de referencias de productos que se han vendido alguna vez, y la consulta principal muestra los productos cuya referencia no está (NOT IN) en ese conjunto.

 **[NOT] IN** no puede comparar con valores NULL por tanto estos valores nunca superan el filtro y conviene eliminarlos antes de la comparación. Además si el conjunto está vacío ningún valor superará el filtro.

ALL

Cambia los operadores relacionales (**=**, **<**, **>**, **<=**, **>=** y **>=**) para que funcionen con los valores de un conjunto **y devuelve TRUE** si la comparación es **cierta para todos los valores del conjunto**.

Por ejemplo, para **buscar el producto más caro de la tienda** se puede usar la siguiente consulta:

```
mysql> SELECT P1.nombre_prod, P1.precio
-> FROM producto P1
-> WHERE P1.precio >= ALL (SELECT P2.precio FROM producto P2);
+-----+-----+
| nombre_prod | precio |
+-----+-----+
| AVION FK20  | 31.75  |
+-----+-----+
1 row in set (0,01 sec)
```

Esta consulta compara cada **precio** con todos los de la tabla producto y devuelve solo los productos cuyo **precio** es "mayor o igual que todos los demás" lo que obviamente solo puede ser el producto o los productos más caros. Una forma equivalente de resolver esta consulta sin operador ALL sería:

```
SELECT P1.nombre_prod, P1.precio
FROM producto P1
WHERE P1.precio = (SELECT MAX(P2.precio) FROM producto P2);
```

Esta segunda opción es más eficiente, puesto que no necesita comparar cada precio con todos los demás sino solo con uno. Pero lo cierto es que **hoy en día la mayoría de SGBD analizan y optimizan las consultas**, por lo que cambiarán internamente la consulta con el ALL y la eficiencia de ambas consultas será similar. De todos modos conviene diseñar las consultas de modo eficiente y no depender de la optimización del SGBD.

Es común usar el operador **ALL con <>** (sería el equivalente a **NOT IN**) o con los operadores relacionales para mostrar los valores que superan o no superan a todos los valores del conjunto. Mucho menos habitual es usarlo con =, puesto que si se tiene un conjunto de valores distintos es imposible que un solo valor sea igual a todos.



Mucho cuidado con estos 2 comportamientos:

- Si el conjunto está vacío la comparación siempre será cierta y **todos los valores superarán el filtro**.
- Si uno de los valores del conjunto es NULL, en cambio, la comparación siempre será falsa y **la consulta no devuelve nada**.

El siguiente ejemplo muestra estos comportamientos:

```
-- Insertamos un producto con precio NULL
mysql> INSERT INTO producto VALUES ('1','Borrar',NULL);

-- Comprobamos que ahora la consulta no devuelve nada
mysql> SELECT P1.nombre_prod, P1.precio
  -> FROM producto P1
  -> WHERE P1.precio >= ALL (SELECT P2.precio FROM producto P2);
Empty set (0,00 sec)

-- La opción sin ALL sigue funcionando sin problemas
mysql> SELECT P1.nombre_prod, P1.precio
  -> FROM producto P1
  -> WHERE P1.precio = (SELECT MAX(P2.precio) FROM producto P2);
+-----+-----+
| nombre_prod | precio |
+-----+-----+
| AVION FK20  | 31.75  |
+-----+-----+
1 row in set (0,00 sec)

-- Comprobamos ahora que si la subconsulta no devuelve nada
-- todos los productos pasan el filtro y la consulta es incorrecta
-- para ello simplemente ponemos un WHERE FALSE en la subconsulta
-- lo que implica que el filtro nunca se cumple y no devuelve nada.
mysql> SELECT P1.nombre_prod, P1.precio
  -> FROM producto P1
  -> WHERE P1.precio >= ALL (SELECT P2.precio
  ->                        FROM producto P2
  ->                        WHERE FALSE);
+-----+-----+
| nombre_prod | precio |
+-----+-----+
| Borrar      | NULL   |
| AVION FK20  | 31.75  |
| BOLA BOOM   | 22.20  |
| HOOP MUSICAL | 12.80  |
| NAIPES PETER PARKER | 3.00  |
| PATINETE 3 RUEDAS | 22.50  |
| PELUCHE MAYA | 15.00  |
| PETER PAN   | 25.00  |
+-----+-----+
8 rows in set (0,00 sec)

-- Borramos el producto con precio NULL
mysql> DELETE FROM producto WHERE ref_prod = '1';
Query OK, 1 row affected (0,01 sec)
```

ANY/SOME

ANY o **SOME** (son equivalentes) **cambian los operadores relacionales** (**=**, **<>**, **<**, **>**, **<=** y **>=**) para que funcionen con los valores de un conjunto y **devuelve TRUE** si la comparación es **cierta para al menos uno de los valores del conjunto**.

Por ejemplo, vamos a **buscar los productos de los que se han hecho pedidos de la misma cantidad que alguno de los productos del pedido 1** (que se hayan pedido 10 o 12 unidades, si se observa la tabla detalle_pedido). Obviamente debemos descartar los productos del mismo pedido 1. La consulta sería:

```
mysql> SELECT D1.* FROM detalle_pedido D1
-> WHERE D1.num_ped <> 1
->      AND D1.cantidad = ANY (SELECT D2.cantidad
->                             FROM detalle_pedido D2
->                             WHERE D2.num_ped = 1);
```

num_ped	ref_prod	cantidad
3	HM12	10
3	P3R20	10
4	BB75	12

3 rows in set (0,00 sec)


⚠ A diferencia de **ALL** si el conjunto está vacío la comparación siempre será **falsa** y ningún valor superará el filtro. Los valores **NULL** dentro del conjunto en este caso no afectan a la comparación, si algún otro valor cumple con la comparación el elemento pasará el filtro correctamente, aunque **conviene eliminarlos para aligerar la consulta**.

```
-- No aparecen resultados inesperados si la subconsulta esta vacía.
mysql> SELECT D1.* FROM detalle_pedido D1
-> WHERE D1.cantidad = ANY (SELECT D2.cantidad
->                          FROM detalle_pedido D2
->                          WHERE FALSE);
Empty set (0,00 sec)
```

💡 Tanto **ALL** como **ANY/SOME** son operadores muy potentes que permiten realizar comparaciones con conjuntos de valores, pero su uso es poco común en el entorno productivo, ya que se suelen usar alternativas que permiten consultas similares como **IN** o **EXISTS**, que veremos a continuación.

[NOT] EXISTS

Comprueba si existe o no al menos una fila en el resultado de una subconsulta. Si la subconsulta **devuelve alguna fila, EXISTS devuelve TRUE**, en caso contrario devuelve **FALSE**.

 **[NOT] EXISTS** se usa como **filtro sin comparación** con una *expr_columna*, **directamente [NOT] EXISTS (subconsulta)**. Además este operador se usa con subconsultas **correlacionadas**.


La explicación es bastante simple, el hecho de que la subconsulta esté vacía o devuelva valores no se puede comparar con otros valores y por otro lado, si la consulta no es correlacionada entonces todas las filas de la consulta principal pasan el filtro o no la pasa ninguna, sin depender en absoluto de ellas, solo de la subconsulta, que es independiente. Un filtro así no suele ser útil, su utilidad es ver para cada fila si una consulta correlacionada con los valores de la misma devuelve valores o no.

Volviendo al ejemplo usado con el [NOT] IN, si queremos **mostrar los productos que no se han vendido nunca**, podemos usar la siguiente alternativa:

```
mysql> SELECT ref_prod, nombre_prod
-> FROM producto P1
-> WHERE NOT EXISTS (SELECT 1 FROM detalle_pedido D1
->                      WHERE D1.ref_prod = P1.ref_prod);

+-----+-----+
| ref_prod | nombre_prod |
+-----+-----+
| PT50     | PETER PAN   |
+-----+-----+
1 row in set (0,00 sec)
```


En este caso la subconsulta devuelve 1 si hay algún pedido del producto en la fila de la consulta principal o nada si no lo hay. Por tanto, la consulta principal muestra los productos que "no devuelven nada" en la subconsulta (**NOT EXISTS**), es decir, los que no se han vendido nunca.

 Usar 1 como proyección, que resultaría absurdo en otras consultas, es una convención para indicar que no nos importa el valor que devuelve la subconsulta, solo si devuelve algo o no.

En realidad, se podría usar cualquier columna o incluso *, pero puesto que no se van a usar esos campos la consulta sería un poco menos eficiente. La realidad es

que los SGBD normalmente optimizan estas consultas y no importa que proyección se use, pero es una buena práctica usar 1.


2.3. Subconsultas en el FROM. Tablas derivadas

 Las **tablas derivadas** son tablas temporales que **se crean a partir de una subconsulta en el FROM** y que se pueden usar como si fueran una tabla normal en la consulta principal o en otras subconsultas.

Es decir, realizaremos una consulta sobre el resultado de una subconsulta o podremos usar este en combinación (*JOIN*) con otras tablas. **Las tablas derivadas pueden contener uno o varios campos y una o varias filas, que se pueden combinar con otras tablas con cualquier tipo de JOIN para formar la tabla más adecuada para la consulta** sin perder eficiencia con subconsultas correlacionadas.

Esto es especialmente útil cuando se necesita filtrar elementos usando subconsultas con funciones agregadas sobre agrupamientos (*GROUP BY*), encadenar funciones agregadas o combinar los resultados de varias subconsultas.

Se debe tener en cuenta que las tablas derivadas no se almacenan en la base de datos, sino que se crean en memoria al ejecutar la consulta y se eliminan una vez se ha terminado de ejecutar la consulta. Por tanto, no se pueden usar en otras consultas o subconsultas que no sean las que la han creado.

 Se debe **añadir siempre un alias a una tabla derivada** después del paréntesis de cierre.

Por otro lado, puesto que los campos de la tabla derivada muy probablemente contengan operaciones o funciones agregadas, es **altamente recomendable añadir alias a los campos para acceder a ellos desde la consulta principal**. Este es uno de los pocos casos en los que los alias de columna están disponibles, puesto que la subconsulta con los alias no es más que otra tabla y los alias se convierten en los nombres de las columnas de la tabla derivada.

Ejemplo 1

Supongamos que queremos **mostrar el producto más vendido y la cantidad de unidades vendidas de ese producto**. Es similar a buscar el producto más caro (el ejemplo del apartado del operador *ALL*), solo que ahora no disponemos de la lista de productos con sus unidades vendidas, sino que debemos calcularla. Para ello, necesitamos una subconsulta que sume las unidades vendidas de cada producto. Por simplicidad, nos centramos en mostrar solo esta la cantidad máxima y ya añadiremos la referencia del producto.

```
SELECT SUM(D1.cantidad) AS ud
FROM detalle_pedido D1
GROUP BY D1.ref_prod;
```

A) Sobre esta tabla derivada, que está agrupada, podemos aplicar la función MAX para obtener el máximo de unidades vendidas. La consulta sería:

```
mysql> SELECT MAX(PV.suma) as max_vendido
-> FROM (SELECT SUM(D1.cantidad) AS suma
->        FROM detalle_pedido D1
->        GROUP BY D1.ref_prod) PV;
+-----+
| max_vendido |
+-----+
|          43 |
+-----+
1 row in set (0,00 sec)
```

Se puede observar que se ha añadido un alias *PV* a la tabla derivada (productos vendidos); esto es estrictamente necesario o se devolverá un error, aunque se podría no usar este prefijo al llamar al campo *suma*, puesto que no hay conflicto.

Con esto se puede comprender el funcionamiento de las tablas derivadas. La subconsulta que hemos usado para calcular las unidades vendidas se ha convertido en una tabla temporal usada en la consulta principal para calcular el máximo de esta tabla derivada.

Con esto el ejemplo estaría completo en cuanto a tablas derivadas se refiere. Pero ahora se procede a estudiar otras alternativas para comprender mejor las diferencias entre las distintas opciones en el uso de subconsultas y las ventajas y desventajas de cada una.

☁️ Conviene tener presente que en la práctica la eficiencia de las consultas depende mucho del SGBD que se use y de la optimización que haga de las consultas. En los SGBD actuales la consulta real ejecutada pocas veces se corresponde a la que se escribe.

B) Se puede llegar a este mismo resultado mediante **>= ALL** y sin usar tablas derivadas, pero es menos eficiente puesto que necesita comparar con una lista de valores y además ejecuta la agrupación y calcula la suma de cantidades tanto para la consulta principal como para la subconsulta. Por tanto 2 veces en lugar de una.

```
SELECT SUM(D1.cantidad) AS max_vendido
FROM detalle_pedido D1
GROUP BY D1.ref_prod
HAVING SUM(D1.cantidad) >= ALL (SELECT SUM(D2.cantidad)
                                FROM detalle_pedido D2
                                GROUP BY D2.ref_prod);
```

C) Se puede sortear el `>= ALL` usando **MAX en el HAVING**, pero puesto que ese máximo se tiene que calcular sobre la suma de cantidades, esta vez estamos usando **una tabla derivada dentro de la subconsulta**. Esto es porque ahora necesitamos calcular la función agregada **MAX** sobre una subconsulta que contenga la función agregada **SUM**. Por tanto este ejemplo contiene 2 subconsultas anidadas.

```
SELECT SUM(D1.cantidad) AS max_vendido
FROM detalle_pedido D1
GROUP BY D1.ref_prod
HAVING SUM(D1.cantidad) = (SELECT MAX(suma)
                           FROM (SELECT SUM(D2.cantidad) as suma
                                FROM detalle_pedido D2
                                GROUP BY D2.ref_prod) PV);
```

Se puede observar además que esta opción en realidad usa como condición del **HAVING** la consulta entera que se muestra como primera opción (A) para resolver el caso, por lo que obviamente es menos eficiente, pero de una eficiencia superior a B en la teoría. Aunque necesita ejecutar el proceso agrupamiento y suma 2 veces y calcular el máximo, no compara con un listado de valores. En realidad, la opción B y esta tienen una eficiencia muy similar pues en la optimización de B los SGBD suelen sustituir el `>=ALL` y se ejecuta de forma muy similar a esta.

Pero esta no es la consulta que se ha pedido realmente, puesto que se pide el producto más vendido, no solo la cantidad de unidades vendidas. Vamos a ver ahora en cada caso cómo añadir este dato a la consulta.

Añadir este detalle es trivial con las soluciones B y C solo es necesario añadir D1.ref_prod a la proyección, dado que está en el **GROUP BY** y no hay problema con ello.

Esta es la gran ventaja de estos tipos de estructura en la consulta; a costa de una posible pérdida de eficiencia las variaciones y ampliaciones de la consulta son más sencillas y basta con modificar la consulta principal, probablemente no necesiten modificaciones en las subconsultas ni en la estructura de la consulta.

```

--- Opción B incluyendo ref_prod.
SELECT D1.ref_prod, SUM(D1.cantidad) AS max_vendido
FROM detalle_pedido D1
GROUP BY D1.ref_prod
HAVING SUM(D1.cantidad) = (SELECT MAX(suma)
                           FROM (SELECT SUM(D2.cantidad) as suma
                                FROM detalle_pedido D2
                                GROUP BY D2.ref_prod) PV);

```

Sin embargo, con la tabla derivada en la consulta principal (A) estas adaptaciones requieren más cambios y pueden resultar menos obvias. En este caso el **MAX** en la proyección de la consulta principal agrupa las filas y hace inviable mostrar los otros datos, por lo que hay que moverlo de allí. Usar subconsultas en el filtrado para calcular el máximo, por otro lado, nos llevaría a las soluciones B y C. Por tanto la opción que queda es **incluir el cálculo del máximo en la propia tabla derivada**.

Dependiendo de casos puede que se necesiten 2 o más tablas derivadas para mostrar toda la información, o que baste con una, combinada con JOINS a tablas de la base de datos. En este caso se puede usar directamente una tabla derivada que calcule el máximo de unidades vendidas, es decir que la consulta A por completo se convierte ahora en la tabla derivada, con un solo valor que contiene el máximo de unidades vendidas.

Añadimos con un **CROSS JOIN** esta tabla derivada a la consulta principal, que será muy parecida a la usada en C, pero moviendo la subconsulta del **HAVING** a una tabla derivada. Con el **CROSS JOIN** unimos el valor **maximo** (se puede poner como alias) a cada fila de la consulta principal y solo se necesita comparar la suma de cantidades con ese **maximo** anexo a cada fila en vez de a una subconsulta. La consulta quedaría así:

```

mysql> SELECT D1.ref_prod, SUM(D1.cantidad) AS max_vendido
-> FROM detalle_pedido D1
-> CROSS JOIN (SELECT MAX(PV.suma) as maximo
->              FROM (SELECT SUM(D2.cantidad) as suma
->                    FROM detalle_pedido D2
->                    GROUP BY D2.ref_prod) PV) MX
-> GROUP BY D1.ref_prod, MX.maximo
-> HAVING SUM(D1.cantidad) = MX.maximo;
+-----+-----+
| ref_prod | max_vendido |
+-----+-----+
| P3R20    |          43 |
+-----+-----+
1 row in set (0,00 sec)

```

En este caso la eficiencia deja de ser superior en general y no cambia tanto respecto a *B* o *C* con *ref_prod* añadido, puesto que en todos los casos necesita hacer la agrupación 2 veces y comparar los valores. De hecho, la estructura de las consultas es similar, solo que esta calcula el máximo en una tabla derivada y las otras en el filtro.

Pero de nuevo, dependiendo del SGBD y la forma que tenga de optimizar las consultas, la eficiencia puede variar, incluso ser menor que en las anteriores. Lo cierto es que en MySQL la eficiencia de esta consulta sí es netamente superior a las otras porque con la optimización detecta que la tabla derivada solo se usa para la condición del *HAVING* y que ese máximo se puede calcular desde la consulta principal. Tras optimizar, no hace el *JOIN* ni la tabla derivada, simplemente calcula el número y lo usa en el *HAVING*, ahorrándose hacer la subconsulta.

Curiosamente con las estructuras *B* y *C* MySQL no es capaz de detectar que podría hacer lo mismo, ya que la subconsulta no la analiza hasta resuelto el FROM, y por tanto sí que hace la subconsulta completa para calcular el máximo.

Ejemplo 2

Supongamos ahora que queremos **mostrar los artículos cuyo precio es mayor que la media de los precios de los artículos**. Este ejemplo ya se ha visto en las consultas en el filtrado, pero ahora se adapta para usar una tabla derivada.

La consulta sería:

```
mysql> SELECT P1.nombre_prod, P1.precio
-> FROM producto P1
-> CROSS JOIN (SELECT AVG(P2.precio) as media
-> FROM producto P2) M
-> WHERE P1.precio > M.media;
```

nombre_prod	precio
AVION FK20	31.75
BOLA BOOM	22.20
PATINETE 3 RUEDAS	22.50
PETER PAN	25.00

4 rows in set (0,00 sec)

Unimos el dato de la media (tabla derivada) a cada fila de la consulta principal con un *CROSS JOIN* y solo necesitamos comparar el precio con la media anexada.

Ejemplo 3

Por último, se analiza un ejemplo en el que **una tabla derivada puede substituir a una subconsulta correlacionada**. Supongamos que se quieren **obtener los productos de los que se ha pedido una cantidad superior a la media de las cantidades pedidas en ese mismo pedido**.

Primero se muestra con la subconsulta correlacionada. Se usa una subconsulta que agrupa las medias de las cantidades por pedido, pero filtra el pedido para sólo considerar cada vez el pedido de la fila de la consulta principal. Esa media se compara con la **cantidad** del producto para ver si es superior o no

```
mysql> SELECT D1.ref_prod, D1.num_ped, D1.cantidad
-> FROM detalle_pedido D1
-> WHERE D1.cantidad > (SELECT AVG(D2.cantidad)
->                        FROM detalle_pedido D2
->                        WHERE D2.num_ped = D1.num_ped
->                        GROUP BY D2.num_ped);
```

ref_prod	num_ped	cantidad
AFK11	1	12
PM30	3	20
AFK11	4	30
P3R20	5	18


4 rows in set (0,00 sec)

La consulta con la tabla derivada usa una tabla donde se calculan las medias de todos los pedidos y se une la media del pedido correspondiente a cada fila de la tabla detalle_pedido. La consulta sería:

```
SELECT D1.ref_prod, D1.num_ped, D1.cantidad
FROM detalle_pedido D1
INNER JOIN (SELECT D2.num_ped, AVG(D2.cantidad) as media
            FROM detalle_pedido D2
            GROUP BY D2.num_ped) M
ON M.num_ped = D1.num_ped
WHERE D1.cantidad > M.media;
```

La clave está en el **JOIN** de las tablas; con **ON M.num_ped = D1.num_ped** se une cada media a su pedido correspondiente y se puede comparar la cantidad con la media anexada.

Puesto que las consultas correlacionadas se ejecutan por cada fila mientras que la tabla derivada, se ejecuta una única vez, la eficiencia de la tabla derivada es mayor en términos generales. Como siempre, la optimización de las consultas del SGBD, o incluso la existencia de índices, podría variar este comportamiento.

 Aunque se ha estado comentando sobre la eficiencia de las distintas posibilidades y estructuras a usar en las consultas, esto se ha hecho para poder entender mejor las subconsultas y por motivos puramente didácticos. **Mientras las consultas sean funcionales y cumplan con los requisitos que se piden sin redundancias innecesarias o errores, se considerarán igualmente correctas.**

El análisis de la eficiencia de las consultas es una importante tarea que se realiza en entornos productivos, pero depende de demasiados factores como para trabajarla en profundidad en este curso.

2.4. DQL EN INSTRUCCIONES DML

En las instrucciones DML también se pueden usar subconsultas para filtrar, redistribuir o agrupar los datos para la operación, siempre que estén incluidos en alguna tabla de la base de datos.

2.4.1. INSERT

Es posible insertar en una tabla los resultados de una subconsulta, a esta operación se la denomina adición de datos por subconsulta. La sintaxis es:

```
INSERT INTO nombre_tabla [(columna1, columna2, ...)]
consulta
```

Evidentemente, la consulta debe devolver la misma cantidad de columnas que se hayan especificado en el **INSERT** o que la propia tabla de destino, si no se especifican. Además estos campos deben coincidir en tipo de datos y orden.

Por ejemplo, supongamos que se ha creado una tabla **empleados_inf** con los mismos campos que la vista del apartado anterior y se desea **copiar allí los empleados del departamento de informática**, se podría hacer así:

```
INSERT INTO empleados_inf (dni, nombre_emp, dpt)
SELECT dni, nombre_emp, dpt
FROM empleados
WHERE dpt = 'INF';
```

Mientras el tipo de datos y los campos sean compatibles los datos pueden venir de cualquier subconsulta, con varias tablas, agrupaciones o cualquier otra operación; también se pueden usar subconsultas o vistas.

2.4.2. UPDATE

En las actualizaciones de datos se pueden usar subconsultas tanto en el filtro (*WHERE*) que determina qué filas se van a actualizar, como en el valor (*SET*) que se va a asignar a las columnas. En el caso del *SET* la subconsulta debe devolver un único valor compatible con la columna que se actualiza.

Pero existe un limite en MySQL, **no se pueden usar consultas que modifiquen la misma tabla que se está actualizando**. Esto se debe a que se estaría leyendo y actualizando los datos de la misma tabla a la vez, con lo que se podría entrar en un bucle infinito y en numerosas contradicciones.

Por ejemplo, si se quiere actualizar el departamento de los empleados del mismo departamento que 'Mariano Sanz' a 'INF' no se puede hacer directamente con una consulta, puesto que esta usa la propia tabla de empleados. En cambio, para actualizar el cambiar a los empleados del departamento de 'INF' al departamento que lleva por nombre 'Informática', no hay problema, pues la consulta usa la tabla departamentos y no empleados.

Se usan estos ejemplos un poco absurdos para evitar que la actualización realice cambios reales, puesto que en ambas actualizaciones se "modifica" a 'INF' el departamento de los empleados de ese mismo departamento.

```
mysql> UPDATE empleados
-> SET dpt = 'INF'
-> WHERE dpt = (SELECT dpt
->                FROM empleados
->                WHERE nombre_emp = 'Mariano Sanz');
ERROR 1093 (HY000): You can't specify target table 'empleados'
for update in FROM clause
mysql> UPDATE empleados
-> SET dpt = (SELECT cod_dpt
->                FROM departamentos
->                WHERE nombre_dpt = 'Informática')
-> WHERE dpt = 'INF';
Query OK, 0 rows affected (0,00 sec)
Rows matched: 2  Changed: 0  Warnings: 0
```

Curiosamente, **para evitar la limitación** de la consulta que usa la misma tabla que se está actualizando, **se puede usar una tabla derivada**. La tabla derivada sí que puede usar la propia tabla a actualizar, puesto que primero se calcula la tabla

derivada y ya después se procede a actualizar las filas de la tabla original, con lo que la consulta ya no se ve afectada por los datos actualizados.

Si se repite la actualización anterior con una tabla derivada, MySQL ya es capaz de realizarla (aunque no haga cambios reales, como ya se ha comentado).

```
mysql> UPDATE empleados
-> SET dpt = 'INF'
-> WHERE dpt = (SELECT E.dpt
->                FROM (SELECT dpt
->                FROM empleados
->                WHERE nombre_emp = 'Mariano Sanz') E);
Query OK, 0 rows affected (0,00 sec)
Rows matched: 2  Changed: 0  Warnings: 0
```

2.4.3. DELETE

En la eliminación de datos también se pueden usar subconsultas en el filtro (*WHERE*) que determina qué filas se van a eliminar. De nuevo, tanto la limitación de usar en la consulta la tabla de la que se está eliminando información como la posibilidad de usar una tabla derivada para esquivarla se aplican aquí.

Hay empleados del departamento de informática como responsables de proyecto, lo que evita que se los elimine por la restricción de clave foránea. Se elige esta vez usar una *transacción* para **eliminar a todos los empleados del mismo departamento que 'Roberto Milán'** (aunque que en realidad es solo él mismo) y deshacer el cambio con *ROLLBACK* para no afectar a la tabla.

```
mysql> DELETE FROM empleados
-> WHERE dpt = (SELECT dpt
->                FROM empleados
->                WHERE nombre_emp = 'Roberto Milán');
ERROR 1093 (HY000): You can't specify target table 'empleados' for
update
in FROM clause
mysql> DELETE FROM empleados
-> WHERE dpt = (SELECT E.dpt
->                FROM (SELECT dpt
->                FROM empleados
->                WHERE nombre_emp = 'Roberto Milán') E);
Query OK, 1 row affected (0,00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0,00 sec)
```

Aunque a priori pueda parecer extraño, el uso de subconsultas en las instrucciones DML es una práctica muy común en el día a día de todo administrador de una base de datos o de un desarrollador de software con acceso directo (o indirecto vía software) a la misma.

Algunos SGBD, como MySQL, incluyen también la posibilidad de realizar borrados y actualizaciones en varias tablas a la vez con una nomenclatura específica, pero el estudio de estas instrucciones no se considera necesario para los contenidos de este curso.

3. ORDER BY y LIMIT 1 para calcular máximos y mínimos

En el apartado de subconsultas se ha visto cómo calcular valores máximos y mínimos con subconsultas, pero también se puede hacer con *ORDER BY* y mostrando sólo la primera fila del resultado.

Esto tiene la ventaja de que **si existen índices para los campos de los que se quiere hallar el máximo o mínimo el resultado de la consulta es inmediato** y más eficiente que cualquier otra opción. Pero en cambio existen **2 desventajas** que se deben tener presentes:

- Para limitar los resultados mostrados en la consulta no existe nomenclatura común a todos los SGBD, **no es SQL estándar**. En MySQL se usa *LIMIT*, en otros SGBD se usa *TOP* (SQL SERVER) o *FETCH FIRST* (Oracle). Ni siquiera estas cláusulas se sitúan en todas en el mismo lugar de la consulta, por lo que **es necesario conocer la sintaxis de cada SGBD**.
- Si hay varios valores máximos o mínimos, *LIMIT 1* solo mostrará uno de ellos. Para obtener el valor numérico funciona bien, pero si lo que se quiere es mostrar los datos de las filas con valor máximo o mínimo solo veremos una fila y no todos los elementos. En otros SGBD los comandos equivalentes a *LIMIT* tienen una opción para mostrar todos los valores "empatados", lo que evita el problema, pero no es el caso de MySQL.

⚠ Nunca se debe usar LIMIT 1 para mostrar los datos de la/s fila/s máxima/mínima, solo para obtener el valor de esa columna.

Ejemplo 1

Volviendo al ejemplo usado con el operador *ALL*, **buscar el producto más caro de la tienda**, se puede usar esta opción y ganar en eficiencia respecto a las opciones mostradas anteriormente si existe un índice para el campo *precio*.


```
SELECT P1.ref_prod, P1.precio
FROM producto P1
WHERE P1.precio = (SELECT P2.precio
                  FROM producto P2
                  ORDER BY P2.precio DESC
                  LIMIT 1);
```

Ejemplo 2

También con un ejemplo anterior, el que pedía **mostrar el producto más vendido y la cantidad de unidades vendidas de ese producto** y donde se han estudiado varias opciones para resolverlo, se puede sustituir la subconsulta por esta opción, tanto en el *WHERE* como en la tabla derivada.

En este caso concreto, puesto que la ordenación se hace sobre *SUM(cantidad)*, que no se puede indexar, no se puede considerar una mejora en la eficiencia. Aunque en este tipo de consultas ayuda que existan índices sobre los campos usados en el *GROUP BY*, como sucede en este caso al ser *ref_prod* una clave foránea, esta ayuda en la agrupación también se da en las otras opciones.

```
SELECT D1.ref_prod, SUM(D1.cantidad) AS max_vendido
FROM detalle_pedido D1
GROUP BY D1.ref_prod
HAVING SUM(D1.cantidad) = (SELECT SUM(D2.cantidad) as suma
                          FROM detalle_pedido D2
                          GROUP BY D2.ref_prod
                          ORDER BY 1 DESC
                          LIMIT 1);
```

 Si no existen índices que ayuden a la ordenación, usar *ORDER BY* y *LIMIT 1* no es en absoluto más eficiente que las opciones estudiadas anteriormente, ya que la ordenación no es precisamente un proceso eficiente.

4. Uniones

La **unión (UNION [ALL])** nos permite unir en un solo resultado los resultados de varias consultas independientes (de la misma o distintas tablas) siempre que cumplan las siguientes condiciones en todas las consultas unidas:

- Deben tener el mismo número de columnas en el resultado.
- Deben colocar las columnas en el mismo orden.
- Deben tener el mismo tipo de datos de cada columna.

La sintaxis de **UNION** es:

```
consulta1
UNION [ALL]
consulta2
[ORDER BY ...];
```

Esto se usa para unir los resultados de consultas distintas, que no se puedan obtener fácilmente en una sola consulta, o unir datos en una misma columna que pertenezcan a campos distintos en las tablas pero que se puedan considerar equivalentes para el resultado de una consulta, como por ejemplo unir los nombres de los clientes de 2 tablas distintas.

El **ORDER BY** se usa al final puesto que no tiene sentido ordenar una consulta parcial a la que se le pueden añadir filas con la unión.

La opción **ALL muestra todas las filas, incluso las duplicadas en ambas consultas**. Es decir, si en una unión de consultas hay filas en la primera consulta que son iguales a filas en la segunda consulta, se mostrarán ambas. Si no se usa **ALL**, se mostrarán solo una vez.

Además de **UNION**, existen otros operadores de conjuntos que permiten mostrar los resultados de las consultas de distintas formas. Estas opciones son:

- **INTERSECT**: Muestra solo las filas que están en ambas consultas. Es decir que muestra solo las filas que la opción **ALL** de **UNION** estaba añadiendo; la intersección de las consultas.
- **EXCEPT**: Muestra solo las filas que están en la primera consulta y no en la segunda. También llamado **MINUS** en algunos SGBD, aunque MySQL lo reconoce solo como **EXCEPT**. Esta unión no es simétrica como las anteriores, el orden en el que se sitúan las consultas importa.

La opción **ALL** no es de aplicación a estos otros operadores.

Ejemplo 1



MySQL no implementa el **FULL OUTER JOIN**, pero se puede realizar una consulta equivalente usando **UNION**.

Supongamos que se necesita una **lista con TODOS los empleados y TODOS los proyectos, mostrando a los empleados responsables de cada proyecto**.

```
mysql> SELECT E.nombre_emp, P.nombre AS proyecto
-> FROM empleados E
-> LEFT JOIN proyectos P ON P.responsable = E.dni
-> UNION
-> SELECT E.nombre_emp, P.nombre
-> FROM empleados E
-> RIGHT JOIN proyectos P ON P.responsable = E.dni;
```

nombre_emp	proyecto
Alberto Gil	Repsol, S.A.
Mariano Sanz	Consejería de Educación
Iván Gómez	NULL
Ana Silván	NULL
María Cuadrado	NULL
Roberto Milán	NULL
NULL	Oceanográfico

7 rows in set (0,00 sec)

Esta unión muestra todos los empleados (*LEFT*) y proyectos (*RIGHT*), aunque no haya correspondencia entre el empleado y el responsable del proyecto.

Las columnas toman su nombre de la primera consulta. En la segunda solo importa que el numero y tipo de los campos sea el mismo, no los nombres.

Ejemplo 2

Mostramos ahora los empleados y proyectos donde hay un responsable, usando INTERSECT. Esta consulta se realiza de forma más sencilla con un INNER JOIN en una consulta simple, se muestra así sólo para comprender su funcionamiento.

```
mysql> SELECT E.nombre_emp, P.nombre AS proyecto
-> FROM empleados E
-> LEFT JOIN proyectos P ON P.responsable = E.dni
-> INTERSECT
-> SELECT E.nombre_emp, P.nombre
-> FROM empleados E
-> RIGHT JOIN proyectos P ON P.responsable = E.dni;
```

nombre_emp	proyecto
Alberto Gil	Repsol, S.A.
Mariano Sanz	Consejería de Educación

2 rows in set (0,00 sec)

UNION ALL mostraría lo mismo que *UNION* pero duplicando las que aparecen aquí, es decir que estos 2 trabajadores y proyectos aparecerían 2 veces en el resultado.

Ejemplo 3

Por último, se pide una **lista con los datos de los empleados que no son responsables de ningún proyecto**.

En realidad lo más común para este tipo de consulta es usar *NOT IN* o *NOT EXISTS*. Cabe destacar que con *NOT IN* necesitamos eliminar los valores nulos de responsable primero o la consulta no devuelve nada

```
SELECT E.nombre_emp
FROM empleados E
WHERE E.dni NOT IN (SELECT P.responsable
                   FROM proyectos P
                   WHERE P.responsable IS NOT NULL);
```

```
mysql> SELECT E.nombre_emp
-> FROM empleados E
-> WHERE NOT EXISTS (SELECT 1
->                   FROM proyectos P
->                   WHERE P.responsable = E.dni);

+-----+
| nombre_emp |
+-----+
| Iván Gómez |
| Ana Silván |
| María Cuadrado |
| Roberto Milán |
+-----+
4 rows in set (0,00 sec)
```

Pero para el caso que nos ocupa, también se puede usar *EXCEPT* para obtener el mismo resultado, eliminando de la lista de empleados los que sí son responsables de algún proyecto. Esta vez no se necesita usar *OUTER JOIN*.

```
SELECT E.nombre_emp
FROM empleados E
EXCEPT
SELECT E.nombre_emp
FROM empleados E
INNER JOIN proyectos P ON P.responsable = E.dni;
```

Las uniones y operadores de conjuntos son muy útiles para mostrar los resultados de consultas que no se pueden obtener fácilmente en una sola consulta, pero su uso es poco común en el entorno productivo, ya que se suelen usar alternativas que permitan consultas similares.

5. Vistas

Una vista es una tabla virtual que se compone de los resultados de una consulta.

Una vista no es más que una **consulta almacenada con un nombre** a fin de utilizarla tantas veces como se desee, como si de una tabla real se tratase.

Las vistas **no contienen realmente los datos sino solo la instrucción** necesaria para lanzar la consulta; eso asegura que los datos sean coherentes y se actualicen cada vez que se llama a la vista, ya que se ejecuta la consulta de nuevo en ese mismo momento. Debido a esto, las vistas gastan muy poco espacio de disco pero externamente aparentan funcionar como las propias tablas.

⚠ Las instrucciones **DML (INSERT, UPDATE, DELETE)** se pueden realizar directamente sobre una vista, afectando a los datos de las tablas originales.

Las vistas se usan principalmente por **simplicidad y/o por seguridad**. Sus funciones más habituales son:

- *Como medida de seguridad* a la hora de asignar permisos a los diferentes usuarios que hagan uso de la base de datos. Creamos vistas y permitimos a los diferentes usuarios que usen las vistas en lugar de las tablas directamente.
- *Proporcionar tablas con datos completos*, distribuidos de modo distinto a como están en la base de datos o con datos calculados.
- Para *simplificar consultas de un grado de dificultad alta*. Permiten dividir una consulta en partes de forma que es más sencilla de diseñar.
- Ser utilizadas *como cursores de datos* en los lenguajes procedimentales (como PL/SQL).

Hay dos tipos de vistas:


- **Simples**: Las forman una sola tabla y no contienen funciones de agrupación. Su ventaja es que permiten siempre realizar operaciones DML sobre ellas.
- **Complejas**. Obtienen datos de varias tablas, pueden utilizar funciones de agrupación. No siempre permiten operaciones DML, pues dependiendo de la definición de la vista, no siempre se puede determinar la tabla o filas afectadas por la operación.

La sintaxis simplificada que usaremos para la creación de vistas es la siguiente:

```
CREATE [OR REPLACE] VIEW nombre_vista [(alias1, alias2, ...)]
AS consulta
[WITH CHECK OPTION]
```

Siendo:

- **OR REPLACE**: Si la vista ya existe, la reemplaza por la actual.
- **alias1, alias2, ...**: Alias de las columnas de la vista, si se usan reemplazarán a los nombres originales de las columnas de la consulta. Deben coincidir en número con las columnas de la consulta.
- **consulta**: Definición completa de la consulta que define la vista.
- **WITH CHECK OPTION**: Si se usa, solo las filas que pertenecen o pertenecerán a la vista son afectadas por las operaciones DML.

 Conviene tener cuidado con las operaciones DML sobre vistas. En otros SGBD se pueden hacer de solo lectura con la opción *WITH READ ONLY* para no permitir las, pero MySQL no la implementa.

En general es más fácil restringir los permisos de modificación de las tablas a usuarios que solo tengan acceso a vistas.

Para consultar o modificar mediante DML las vistas basta con incluirlas en la instrucción como si fueran una tabla normal. De hecho, MySQL incluso muestra las vistas en el resultado de un `SHOW TABLES;`, pero este comportamiento no es general, otros SGBD almacenan las vistas en listas distintas de las de las tablas.

Cuando una vista ya no sea necesaria se puede eliminar utilizando la instrucción:

```
DROP VIEW nombre_vista;
```

Ejemplo 1

Supongamos que se necesita **crear una vista que incluya solamente el dni, nombre y el departamento de los empleados del departamento de Informática** (INF).

Las instrucciones para crear la vista, mostrar las tablas y así ver que la lista se encuentra entre ellas y consultar la vista creada serían:


```
mysql> CREATE OR REPLACE VIEW empleados_informatica
-> AS SELECT dni, nombre_emp, dpt FROM empleados
-> WHERE dpt = 'INF';
Query OK, 0 rows affected (0,01 sec)
```

```
mysql> SHOW TABLES;
+-----+
| Tables_in_db_empresa |
+-----+
| departamentos        |
| empleados             |
| empleados_informatica |
| proyectos            |
+-----+
5 rows in set (0,00 sec)
```

```
mysql> SELECT * FROM empleados_informatica;
+-----+-----+-----+
| dni      | nombre_emp | dpt  |
+-----+-----+-----+
| 23456789B | Mariano Sanz | INF  |
| 45678901D | Ana Silván   | INF  |
+-----+-----+-----+
2 rows in set (0,00 sec)
```

Esta es una vista simple, que permite realizar operaciones DML sobre ella, puesto que solo contiene una tabla y no usa funciones de agrupación. Pero evidentemente este proceso tiene sus condiciones:

- No se pueden insertar, actualizar ni filtrar datos en los campos que no están en la vista.
- Si se insertan filas, los datos que no aparecen en la vista se pondrán a NULL o a DEFAULT. Si alguno de estos campos fuese NOT NULL y no tuviese DEFAULT, la operación fallará.
- Si la vista no incluye la clave primaria de la tabla no se podrán insertar filas, a menos que esta clave tenga algún valor por defecto o sea auto-incremental.

Además, como no se ha usado la opción *WITH CHECK OPTION*, no se restringe la modificación de las filas a las de la vista por lo que:

- Se pueden añadir filas que posteriormente no aparezcan en la vista, por ejemplo añadir empleados a cualquier otro departamento.
- Lo mismo ocurriría con la actualización; si se cambia el departamento de un empleado de Informática a otro, este empleado desaparecerá de la vista.

Con *WITH CHECK OPTION*, las anteriores acciones no estarían permitidas.

Borrar filas que no se muestran en la tabla, excepto las que se borren por borrados en cascada, no es posible normalmente, pues no se encuentran al filtrar, da igual si *WITH CHECK OPTION* está o no.

Se puede comprobar ejecutando la siguiente **transacción**:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

mysql> INSERT INTO empleados_informatica
-> VALUES ('1', 'Borrar', 'CONT');
Query OK, 1 row affected (0,00 sec)

mysql> UPDATE empleados_informatica SET dpt='CONT';
Query OK, 2 rows affected (0,00 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> select * from empleados_informatica;
Empty set (0,01 sec)

mysql> select * from empleados;
+-----+-----+-----+-----+-----+
| dni      | nombre_emp | especialidad | fecha_alta | dpt |
+-----+-----+-----+-----+-----+
| 1        | Borrar     | NULL        | NULL       | CONT |
| 12345678A | Alberto Gil | Contable    | 2010-12-10 | CONT |
| 23456789B | Mariano Sanz | Informática | 2011-10-04 | CONT |
| 34567890C | Iván Gómez | Ventas      | 2012-07-20 | COM  |
| 45678901D | Ana Silván | Informática | 2012-11-25 | CONT |
| 56789012E | María Cuadrado | Ventas    | 2013-04-02 | COM  |
| 67890123A | Roberto Milán | Logística  | 2010-02-05 | ALM  |
+-----+-----+-----+-----+-----+
7 rows in set (0,00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0,01 sec)
```

Se puede observar que se han añadido y actualizado filas a través de la vista que no aparecerán en la misma, al no ser del departamento de informática. Esto afecta a la tabla original de empleados, y sería definitivo, pero al estar en una transacción, se anulan los cambios con un *ROLLBACK*.

Si se repiten el *INSERT* o el *UPDATE* con la opción *WITH CHECK OPTION* activada, fallarán directamente, no están permitidos. Solo se permiten cambios que seguirían apareciendo en la vista. Como insertar o modificar empleados del departamento de informática

```
mysql> CREATE OR REPLACE VIEW empleados_informatica
-> AS SELECT dni, nombre_emp, dpt FROM empleados
-> WHERE dpt = 'INF'
-> WITH CHECK OPTION;
Query OK, 0 rows affected (0,00 sec)

mysql> INSERT INTO empleados_informatica
-> VALUES ('1', 'Borrar', 'CONT');
ERROR 1369 (HY000): CHECK OPTION failed
'db_empresa.empleados_informatica'
mysql> UPDATE empleados_informatica SET dpt='CONT';
ERROR 1369 (HY000): CHECK OPTION failed
'db_empresa.empleados_informatica'

mysql> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)

mysql> INSERT INTO empleados_informatica
-> VALUES ('1', 'Borrar', 'INF');
Query OK, 1 row affected (0,00 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0,00 sec)
```

De nuevo se ha usado una transacción con *ROLLBACK* para evitar cambios reales en la tabla.

Ejemplo 2

Supongamos ahora que se pide **crear una vista que muestre las líneas de factura de la tienda, con el numero de pedido, nombre del producto, la cantidad y el precio total**.

Esta tabla, aún sin contener agregaciones, no es simple, puesto que contiene datos de varias tablas y elementos calculados, por lo que las operaciones DML sobre ella están limitadas (que no completamente descartadas) aún sin usar *WITH CHECK OPTION*.

Por ejemplo, insertar productos no es posible. En cambio, modificar el nombre de productos sí es posible, puesto que puede localizarlos por nombre y actualizar los campos que están incluidos en la vista.

Se puede observar como el error al intentar insertar un producto es diferente al de la vista simple, directamente los INSERTS sobre proyectos no están permitidos, aunque podrían estar permitidos en alguna de las tablas incluidas, según la definición de la vista.

```
mysql> CREATE VIEW facturacion (Pedido,Producto,Ud, Precio, Total) AS
-> SELECT num_ped, nombre_prod, cantidad, precio, precio*cantidad
-> FROM producto P
-> INNER JOIN detalle_pedido D ON P.ref_prod = D.ref_prod;
Query OK, 0 rows affected (0,01 sec)
```

```
mysql> SELECT * FROM facturacion;
```

Pedido	Producto	Ud	Precio	Total
1	AVION FK20	12	31.75	381.00
1	NAIPES PETER PARKER	10	3.00	30.00
2	PATINETE 3 RUEDAS	15	22.50	337.50
3	HOOP MUSICAL	10	12.80	128.00
3	PATINETE 3 RUEDAS	10	22.50	225.00
3	PELUCHE MAYA	20	15.00	300.00
4	AVION FK20	30	31.75	952.50
4	BOLA BOOM	12	22.20	266.40
5	BOLA BOOM	5	22.20	111.00
5	NAIPES PETER PARKER	3	3.00	9.00
5	PATINETE 3 RUEDAS	18	22.50	405.00

11 rows in set (0,01 sec)

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0,00 sec)
```

```
mysql> UPDATE facturacion SET Producto='AVION FK11'
mysql> WHERE Producto='AVION FK20';
Query OK, 1 row affected (0,00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> ROLLBACK;
Query OK, 0 rows affected (0,01 sec)
```

```
mysql> INSERT INTO facturacion (Producto) VALUES ('AVION FK11');
ERROR 1471 (HY000): The target table facturacion of the INSERT
is not insertable-into
```

En este ejemplo también se puede observar como los campos han adoptado los nombres especificados tras el nombre de la vista.

Ejemplo 3

En este ejemplo se muestra como las vistas ayudan a la simplificación de consultas. Se debe tener en cuenta que la simplificación es sólo para el usuario, el SGBD necesita ejecutar la consulta completa para obtener los datos de la vista y su eficiencia no mejora en absoluto.

Supongamos de nuevo que se necesita **mostrar el producto más vendido y la cantidad de unidades vendidas de ese producto**. Ya que probablemente es la consulta más compleja de las estudiadas en este documento, se puede usar una vista para simplificarla.

Existen múltiples opciones, depende de la destreza del programador o administrador para usarlas. Se muestran aquí algunas opciones, usando las vistas de forma progresiva de modo que sea más sencillo entender el proceso, aún cuando las primeras opciones no muestren grandes cambios en la estructura de las consultas.

- Crear una vista que muestre la referencia del producto y la suma de las cantidades vendidas por cada producto y usarla para **simplificar las subconsultas** de las distintas opciones propuestas.

```
-- Creación de la vista
CREATE OR REPLACE VIEW prod_vendidos AS
SELECT ref_prod, SUM(cantidad) as suma
FROM detalle_pedido
GROUP BY ref_prod;

-- Simplificar >= ALL
SELECT ref_prod, SUM(cantidad) AS max_vendido
FROM detalle_pedido
GROUP BY ref_prod
HAVING SUM(cantidad) >= ALL (SELECT suma
                             FROM prod_vendidos);

-- Simplificar subconsulta en el HAVING
SELECT ref_prod, SUM(cantidad) AS max_vendido
FROM detalle_pedido
GROUP BY ref_prod
HAVING SUM(cantidad) = (SELECT MAX(suma)
                        FROM prod_vendidos);

-- Simplificar tabla derivada
SELECT ref_prod, SUM(cantidad) AS max_vendido
FROM detalle_pedido
CROSS JOIN (SELECT MAX(suma) as maximo
            FROM prod_vendidos) MX
GROUP BY ref_prod, maximo
HAVING SUM(cantidad) = maximo;
```

De momento no parece una gran simplificación, pero se puede llevar un paso más adelante. De momento solo se ha usado **suma**, pero la vista incluye la **ref_prod** por un motivo...

- Usar la vista anterior tanto en la consulta principal como en la subconsulta.

```
-- >= ALL
SELECT ref_prod, suma AS max_vendido
FROM prod_vendidos
WHERE suma >= ALL (SELECT suma
                    FROM prod_vendidos);

-- Subconsulta en el WHERE (anteriormente en el HAVING)
SELECT ref_prod, suma AS max_vendido
FROM prod_vendidos
WHERE suma = (SELECT MAX(suma)
              FROM prod_vendidos);

-- Usando tablas derivadas
SELECT ref_prod, suma AS max_vendido
FROM prod_vendidos
CROSS JOIN (SELECT MAX(suma) as maximo
            FROM prod_vendidos) MX
WHERE suma = maximo;
```

Con esto la simplificación es mucho más evidente, se pierden de vista los agrupamientos y funciones agregadas casi por completo, ya que quedan ocultos dentro de las vistas y simplemente se usan los campos de las mismas. No obstante las operaciones ejecutadas no han cambiado, es simplemente que no se ven en el texto de la consulta.

Aún se puede simplificar más, ocultando por completo las funciones agregadas.

- Crear otra vista que muestre el valor máximo de la suma de las cantidades vendidas y usar ambas vistas en vez de las tablas de la base de datos.

```
-- Creación de la vista
CREATE OR REPLACE VIEW max_ud_vendidas AS
SELECT MAX(suma) as maximo
FROM (SELECT SUM(cantidad) as suma
      FROM detalle_pedido
      GROUP BY ref_prod) PV;

-- Para el >= ALL no tiene utilidad

-- Subconsulta en el WHERE
SELECT ref_prod, suma AS max_vendido
FROM prod_vendidos
WHERE suma = (SELECT maximo
              FROM max_ud_vendidas);
```

```
-- Usando las 2 vistas anteriores como tablas derivadas
SELECT PV.ref_prod, PV.suma AS max_vendido
FROM prod_vendidos PV
CROSS JOIN max_ud_vendidas MX
WHERE PV.suma = MX.maximo;
```

Alguna de estas consultas incluso se podría crear ahora como vista para que un usuario la pudiese usar sin necesidad de conocer la estructura de las tablas de la base de datos. De este modo para el usuario la base de datos contiene una "tabla" en la que se puede consultar directamente la referencia y el número de unidades vendidas del producto más vendido.

6. Bibliografía

- MySQL 8.0 Reference Manual.
<https://dev.mysql.com/doc/refman/8.0/en/>
- Oracle Database Documentation
<https://docs.oracle.com/en/database/oracle/oracle-database/index.html>
- W3Schools. MySQL Tutorial.
<https://www.w3schools.com/mysql/>
- GURU99. Tutorial de MySQL para principiantes Aprende en 7 días.
<https://guru99.es/sql/>
- SQL Tutorial - Learn SQL.
<https://www.sqltutorial.net/>
- SQLCourse.
<http://dotnetauthorities.blogspot.com/2013/12/Microsoft-SQL-Server-Training-Online-Learning-Classes-Sql-Sub-Queries-Nested-Co-related.html>