Consultas SQL (SELECT)

Consultas sencillas

Devolver todos los campos de una tabla (SELECT *)

```
SELECT *
FROM CLIENTES
```

Con el * indicamos que queremos devolver todos los campos. Si CLIENTES dispone de los campos *idCliente*, *nombre* y *descripcion*, lo anterior sería equivalente a:

```
SELECT idCliente, nombre, descripcion
FROM CLIENTES
```

Obviamente, al querer todos los campos, esto es innecesario y es por tanto más conveniente emplear el asterisco (*). También sería equivalente emplear la notación completa:

```
SELECT CLIENTES.idCliente, CLIENTES.nombre, CLIENTES.descripcion
FROM CLIENTES
```

Al tener únicamente una tabla involucrada, podemos referirnos a los campos sin calificar, dado que no hay duda de a qué tabla se refiere. Cuando veamos consultas sobre varias tablas comprenderemos la necesidad de incluir esta notación calificada (TABLA.campo).

Devolver un subconjunto de los campos de una tabla (SELECT DISTINCT)

```
SELECT cp, ciudad
FROM DIRECCION
```

Esta consulta devolverá únicamente los campos *cp* (código postal) y *ciudad* de la tabla DIRECCION. Al tener un subconjunto de los campos, éstos no tienen por qué incluir a la clave de la tabla, por lo que no tienen por qué ser únicos. Así, si tenemos muchos registros referidos a distintas calles y números de ese mismo código postal y ciudad, nos encontraremos muchos registros repetidos. Esto puede evitarse haciendo:

```
SELECT DISTINCT cp, ciudad
FROM CLIENTES
```

Así se eliminan los registros repetidos, devolviendo únicamente una vez cada par *cp*, *ciudad*. Esta selección de un subconjunto de los datos de la tabla y excluyendo repetidos se denomina en álgebra relacional *proyección*.

Devolver un subconjunto de los registros de una tabla (WHERE)

```
SELECT numero, calle
FROM DIRECCION
WHERE ciudad = 'Sevilla'
```

Esta consulta devolvería el número y la dirección de todas las direcciones pertenecientes a la ciudad de Sevilla. Como vemos, con WHERE indicamos la condición que deben cumplir los registros de la tabla para ser devueltos en la consulta. En este caso tenemos una condición simple dada por la comparación de igualdad (=) entre al campo (*ciudad*) y un literal de tipo

cadena, entre comillas simples (‘Sevilla’).

```
SELECT calle, ciudad
FROM DIRECCION
WHERE numero = 12
```

Esta otra consulta devolvería la calle y ciudad de todos los registros de la tabla con el número 12, en este caso un literal numérico. Las condiciones empleadas pueden ser mucho más complejas incluyendo otro tipo de operadores y combinaciones de los mismos.

Operadores relacionales

Al margen del signo de igualdad empleado anteriormente, se pueden usar n las condiciones simples de las consultas los operadores relacionales habituales, devolviendo siempre un valor booleano (lógico):

Operador Significado
< Menor que
> Mayor que
<> Distinto de
<= Menor ó Igual que
>= Mayor ó Igual que
= Igual que

```
SELECT nombre
FROM CLIENTES
WHERE edad <= 32
```

Adicionalmente, disponemos de operadores de comparación adicionales, también devolviendo valores booleanos (lógicos) True o False según si se cumplen o no las condiciones:

BETWEEN: para indicar un intervalo de valores.

```
SELECT nombre
FROM CLIENTES
WHERE edad BETWEEN 20 AND 35
```

LIKE: empleado para comparar patrones de texto pudiendo incluir comodines como los siguientes:

Comodín Descripción
% Sustituto para cero o más caracteres.
_ Sustituto para exactamente un carácter
[lista caracteres] Cualquier carácter de la lista

[^lista caracteres] Cualquier carácter que no esté en la lista
[!lista caracteres] Cualquier carácter que no esté en la lista

```
SELECT num, calle, cp
FROM DIRECCION
WHERE ciudad LIKE 'Val%'
```

Esta consulta devolvería los datos de las direcciones de toda ciudad que comience por *Val* y siga por cualquier número de caracteres, incluyendo cero caracteres. Por ejemplo, *Valladolid* o *Valencia*. En el enlace anterior sobre patrones de texto podemos practicar directamente desde la web.

IN: empleado para comparar con una lista de valores fijados de modo que devuelva True si el campo indicado pertenece a la lista.

```
SELECT num, calle, direccion
FROM DIRECCION
WHERE ciudad IN ('Sevilla', 'Córdoba', 'Huelva', 'Cádiz')
```

Operadores lógicos (AND, OR, NOT)

Los operadores lógicos nos sirven para componer expresiones de filtrado a partir de las anteriores:

Operador	Significado
AND	Y lógico
OR	O lógico
NOT	Negación lógica

La precedencia y asociatividad es la habitual definida en la Lógica. En cualquier caso, cuando incluya expresiones que empleen varios de estos operadores es recomendable usar paréntesis para evitar errores. Por ejemplo:

```
SELECT *
FROM DIRECCION
WHERE ciudad = 'Sevilla' AND cp = 41009 OR ciudad = 'Córdoba' AND NOT cp = 14010
```

Devuelve los registros pertenecientes a direcciones que tengan el código postal 41009 de Sevilla o bien que no tengan el 14010 de Córdoba. La mayor precedencia la adopta el operador *NOT* sobre la condición *cp = 14010*; a continuación los *AND* se aplican sobre *ciudad = 'Sevilla' AND cp = 41009* y *ciudad = 'Córdoba' AND NOT cp = 14010*; por último se aplica el *OR* sobre la fórmula completa. La misma consulta se puede expresar de forma más clara con paréntesis:

```
SELECT *
FROM DIRECCION
WHERE (ciudad = 'Sevilla' AND cp = 41009) OR
      (ciudad = 'Córdoba' AND (NOT cp = 14010))
```

O bien si el NOT nos parece más evidente, podemos excluir el paréntesis interior, a nuestra

gusto siempre conservando el significado que queríamos dar la operación.

Ordenación

Ordenar según criterios (ORDER BY)

Podemos ordenar los registros devueltos por una consulta por el campo o campos que estimemos oportunos:

```
SELECT *
FROM CIUDAD
ORDER BY provincia ASC, numhabitantes DESC
```

Esta consulta devolvería todas las ciudades ordenadas por provincia en orden ascendente, y dentro de los de la misma provincia ordenaría las ciudades por orden descendente del número de habitantes. Si no indicamos ASC ni DESC, el comportamiento por defecto será el orden ascendente (ASC).

Devolución de expresiones

Asignación de un alias a un dato devuelto (AS)

```
SELECT idCliente AS id, nombre AS cliente, descripcion AS desc
FROM CLIENTES
```

Uso de expresiones empleando operadores y/o funciones

Podemos practicar en SQLzoo a usar expresiones con operadores y funciones. Por ejemplo:

```
SELECT MOD(DAY(NOW()),7) AS numSemana, POW(2,3) AS potencia8
```

Devuelve el número de semana en la que nos encontramos dentro del mes, ya que NOW() nos devuelve la fecha/hora actual, de la cual extraemos el día con DAY, y posteriormente calculamos el módulo 7 de dicho día (de modo que para un día 26 devolvería un 5, por ejemplo). El dato con alias potencia8 devolvería 8 (2 elevado a 3, POW es power, potencia).

Se puede combinar naturalmente la potencia de este lenguaje de expresiones usando operadores y funciones sobre los datos de los registros de una tabla:

```
SELECT DAY(fechaLinea) AS dia, FLOOR(precioLinea * 0.85) AS precioDtoRedondeado
FROM LINEAPEDIDO
```

Consultas agrupadas (GROUP BY)

Las consultas anteriores recuperaban, trabajaban con, y mostraban información a nivel de cada registro individual de la base de datos. Así, si tenemos un producto con un determinado precio, podemos devolver el precio mediante *SELECT precioLinea* o bien operar sobre él como en *SELECT precioLinea * 0.85*.

Ahora bien, podemos querer obtener información que no proviene de un registro individual sino de la agrupación de información, como es el caso de contar el número de líneas de pedido, sumar el precio de todas las líneas por cada pedido, etc. Para ello, debemos emplear funciones agregadas y en la mayoría de los casos agrupar por algún campo.

Así, para ver el número total de registros podemos hacer:

```
SELECT COUNT(*)
```

```
FROM LINEAPEDIDO
```

Si por el contrario deseamos obtener el total de líneas por pedido, debemos indicar que agrupe por idPedido, lo que contará todos los registros con el mismo idPedido y calculará su cuenta:

```
SELECT idPedido, COUNT(*)
FROM LINEAPEDIDO
GROUP BY idPedido
```

Lo mismo se puede aplicar a otras funciones como la suma, indicando en ese caso aparte de la agrupación el campo que queremos sumar:

```
SELECT idPedido, SUM(precioLinea)
FROM LINEAPEDIDO
GROUP BY idPedido
```

¿Y si queremos hallar la media de los precios por cada pedido? En ese caso necesitamos de nuevo agrupar (GROUP BY) por pedido.

```
SELECT idPedido, AVG(precioLinea)
FROM LINEAPEDIDO
GROUP BY idPedido
```

Igualmente, podríamos aplicar un redondeo (ROUND) sobre la media, para dejar 4 decimales, y aplicarle un alias (AS) para el nombre del dato de salida.

```
SELECT idPedido, ROUND(AVG(precioLinea),4) AS media
FROM LINEAPEDIDO
GROUP BY idPedido
```

O podríamos establecer una **condición sobre el dato agrupado** (**HAVING**), de forma que solamente se muestren las medias menores o iguales que 10. Existe una gran cantidad de funciones de agregación definidas en SQL, pero hay que tener precaución porque pueden diferir de un SGBD a otro.

```
SELECT idPedido, ROUND(AVG(precioLinea),4)
FROM LINEAPEDIDO
GROUP BY idPedido
HAVING AVG(precioLinea) < 10
```

Para practicar un poco con las más comunes es muy recomendable este tutorial interactivo. En MySQL tendríamos las que aparecen en este enlace. Este es el desglose completo de las funciones de agregación estándar:

Function Usage
AVG(expression) Computes the average value of a column given by expression.
CORR(dependent, independent) Computes a correlation coefficient.
COUNT(expression) Counts the rows defined by the expression.
COUNT(*) Counts all rows in the specified table or view.

COVAR_POP (dependent, independent) Computes population covariance.	
COVAR_SAMP(dependent, independent) Computes sample covariance.	
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)	Computes the relative rank of a hypothetical row within a group of rows, where the rank is equal to the number of rows less than or equal to the hypothetical row divided by the number of rows in the group.
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a dense rank (no ranks are skipped) for a hypothetical row (value_list) in a group of rows generated by GROUP BY.

MIN(expression) Finds the minimum value in a column given by expression.	
MAX(expression) Finds the maximum value in a column given by expression.	
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.
PERCENTILE_CONT (percentile) WITHIN GROUP (ORDER BY sort_list)	Generates an interpolated value that, if added to the group, would correspond to the percentile given.
PERCENTILE_DISC (percentile) WITHIN GROUP (ORDER BY sort_list)	Returns the value with the smallest cumulative distribution value greater than or equal to percentile.
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a rank for a hypothetical row (value_list) in a group of rows generated by GROUP BY.
REGR_AVGX(dependent, independent) Computes the average of the independent	

variable.
REGR_AVGY(dependent, independent) Computes the average of the dependent variable.
REGR_COUNT(dependent, independent) Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated.
REGR_INTERCEPT(dependent,independent) Computes the y-intercept of the least-squares fit linear equation.
REGR_R2(dependent, independent) Squares the correlation coefficient.
REGR_SLOPE(dependent, independent) Determines the slope of the least-squares-fit linear equation.
REGR_SXX(dependent, independent) Sums the squares of the independent variables.
REGR_SXY(dependent, independent) Sums the products of each pair of variables.
REGR_SYY(dependent, independent) Sums the squares of the dependent variables.
STDDEV_POP(expression) Computes the population standard deviation of all expression values in a group.
STDDEV_SAMP(expression) Computes the sample standard deviation of all expression values in a group.
SUM(expression) Computes the sum of column values given by expression.
VAR_POP(expression) Computes the population variance of all expression values in a group.
VAR_SAMP(expression) Computes the sample standard deviation of all expression values in a group.

Devolución de filas nulas/no nulas (IS NULL / IS NOT NULL)

```
SELECT idPedido
FROM PEDIDO
WHERE direccionEnvio IS NULL
```

Devolvería los pedidos que no tienen dirección de envío asociada.

Consultas por intersecciones entre tablas (JOIN)

Podemos generar una consulta que obtenga datos de varias tablas, pudiendo establecer a su vez criterios sobre otras. Veamos los tipos fundamentales (una buena explicación se puede encontrar aquí). Por ejemplo, dados los clientes de nuestro concesionario y nuestras ventas de coches a clientes en los diversos

VENTA

concesionarios, **CLIENTE**

podemos obtener una consulta que obtenga datos del cliente y de la venta. Por ejemplo, nombre, apellidos, codcoche y color, mediante una consulta del tipo:

```
SELECT nombre, apellidos, codcoche, color
FROM CLIENTE, VENTA
WHERE CLIENTE.cifcl = VENTA.idCliente
```

INNER JOIN

INNER JOIN implícito

Por ejemplo, para obtener los datos del cliente y el pedido en la misma consulta:

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE, PEDIDO
WHERE CLIENTE.idCliente = PEDIDOpedido.idCliente
```

Como vemos, podemos empezar escribiendo tal cuál qué datos nos piden (SELECT), de dónde podemos obtenerlos (FROM) y qué criterio (WHERE). Esta es la versión más antigua de SQL, aunque se sigue empleando, conociéndose como JOIN implícito ya que no se usa por ningún lado la palabra JOIN, pero se está haciendo la intersección por la foreign key, en este caso la columna *idCliente*.

Si omitimos en el caso anterior el WHERE:

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE, PEDIDO
```

Obtendremos los datos solicitados para todos los pares de registros de CLIENTE y PEDIDO, el

producto cartesiano, de forma que si hay 3 registros en CLIENTE y 4 en PEDIDO devolveremos 12 registros. Puede probar estos comportamientos en este enlace. La explicación detallada de un ejemplo similar con coches y marcas se puede encontrar en esta entrada de blog. Este tipo de join se conoce como **CROSS JOIN**, y en este caso se ha hecho de forma implícita.

INNER JOIN natural (NATURAL JOIN)

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE NATURAL JOIN PEDIDO
```

Esta consulta obtiene lo mismo que el join implícito inicial, de forma que NATURAL JOIN establece la condición de igualdad entre los campos con el mismo nombre.

INNER JOIN explícito

La forma más habitual de INNER JOIN es la que intersecta las tablas indicadas en con INNER JOIN por el campo indicado por ON.

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE INNER JOIN PEDIDO
ON cliente.idCliente = pedido.idCliente
```

Este comportamiento se explica y puede probar aquí, en este otro enlace o finalmente aquí. Como vemos, es equivalente al INNER JOIN implícito, pero en lugar de establecer una condición sobre los campos de las relaciones en el WHERE, establece la condición asociada a la relación entre ambas tablas a través de la FOREIGN KEY, en este caso desde el atributo (campo) idCliente en la relación (tabla) PEDIDO, FOREIGN KEY que hace referencia al atributo idCliente de la relación CLIENTE.

Con la notación explícita también podemos expresar un CROSS JOIN como se indicó en el caso explícito:

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE CROSS JOIN PEDIDO
```

Nota: aunque lo más frecuente es que la condición del JOIN sea en términos de igualdad entre las claves, también se podrían establecer condiciones empleando otros operadores relacionales (operadores para comparar dos valores, devolviendo un resultado booleano, es decir, cierto o falso):

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE INNER JOIN PEDIDO
ON cliente.idCliente < pedido.idCliente
```

LEFT JOIN

El resultado de esta operación siempre contiene todos los registros de la relación izquierda (primera tabla que indicamos), y aquellos de la tabla derecha que cumplen la condición establecida. Para el resto aparecerá en los campos correspondientes a dicha tabla un NULL.

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE LEFT JOIN PEDIDO
ON cliente.idCliente = pedido.idCliente
```

Esta consulta devolverá todos los clientes con sus pedidos, y un registro por cada cliente que no tenga pedidos.

RIGHT JOIN

El RIGHT JOIN es análogo al LEFT JOIN, pero devolviendo todos los registros de la relación derecha (segunda tabla que aparece), y únicamente aquellos de la tabla izquierda que cumplen la condición del JOIN. El resultado de esta operación siempre contiene todos los registros de la relación derecha (segunda tabla que indicamos), de modo que en aquellos sin equivalente en la parte izquierda tendrán en los campos correspondientes a dicha tabla un NULL.

```
SELECT nombreCliente, idPedido, fechaPedido
FROM CLIENTE RIGHT JOIN PEDIDO
ON cliente.idCliente = pedido.idCliente
```

Asumiendo que pudiéramos tener en la base de datos pedidos sin cliente asociado, esta consulta devolverá todos los pedidos con sus clientes, y en caso de que el cliente no aparezca el nombreCliente sería NULL.

FULL OUTER JOIN

```
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO FULL OUTER JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
```

Esta consulta devolvería tanto los datos de las empresas sin empleados como los de los empleados sin empresa, apareciendo rellenos todos los datos de la consulta únicamente para aquellos registros que cumplen la condición del JOIN, y apareciendo NULL en unos u otros campos para el resto de registros.

Como se puede ver, esta consulta es equivalente a mostrar tanto los registros devueltos por el LEFT como por el RIGHT JOIN, eliminando los repetidos (aquellos registros que cumplan la condición del JOIN, que serían devueltos por ambas consultas).
El hecho de unir los registros de las consultas de LEFT y JOIN, ambas devolviendo los mismos tipos de campos (EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre) puede ser expresado en SQL a través de la cláusula UNION:

```
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO
LEFT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
UNION
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO
RIGHT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
```

Para devolver los registros de la consulta anterior excluyendo los repetidos haríamos:

```
SELECT DISTINCTROW
FROM (SELECT EMPLEADO.nombre, EMPLEADO.apellidos,
EMPRESA.nombre  FROM EMPLEADO
```

```
LEFT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
UNION
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO
RIGHT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif)
```

Ahora bien, podríamos haber evitado los duplicados mediante una solución más ingeniosa, devolviendo en una de las dos consultas (LEFT o RIGHT) solamente los registros de una de las partes, excluyendo así los que cumplen la condición del JOIN, que serán devueltos por la otra:

```
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO
LEFT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
UNION
SELECT EMPLEADO.nombre, EMPLEADO.apellidos, EMPRESA.nombre
FROM EMPLEADO
RIGHT JOIN EMPRESA
ON EMPLEADO.empresa = EMPRESA.cif
WHERE EMPLEADO.empresa IS NULL
```

Algunos ejercicios adicionales sobre JOIN pueden encontrarse en este tutorial y los primeros 4 apartados de este otro (los restantes ejercicios emplean conceptos que no hemos visto), ambos en SQLZOO.

Consultas anidadas

A veces se han de utilizar en una consulta los resultados de otra consulta, llamada subconsulta o consulta anidada. Podemos ver muchos ejemplos de ello aquí. Veamos que existen diversas variantes:

Empleando IN

Obtener el identificador de los clientes que han comprado algún coche a un concesionario de Madrid.

```
SELECT DISTINCT cifcl
FROM VENTA
WHERE cifc IN (SELECT cifc
FROM CONCESIONARIO
WHERE ciudad = 'Madrid')
```

Obtener el código de coche de los coches vendidos por algún concesionario de Madrid. SELECT DISTINCT codcoche
FROM VENTA

```
WHERE cifc IN (SELECT cifc
 FROM CONCESIONARIO
 WHERE ciudad = 'Madrid')
```

Obtener el nombre y el modelo de los coches vendidos por algún concesionario de Barcelona.

```
SELECT nombre, modelo
FROM COCHE
WHERE codcoche IN (SELECT codcoche
 FROM VENTA
 WHERE cifc IN (SELECT cifc
 FROM CONCESIONARIO
 WHERE ciudad = 'Barcelona'))
```

Empleando operadores relacionales

Un ejemplo típico sería conocer los nombres de los empleados cuyo salario está por encima de la media:

```
SELECT nombre
FROM EMPLEADOS
WHERE salario > (select avg(salario) from emp);
```

Obtener el nombre y apellidos de los clientes cuyo identificador de cliente es menor que el de Juan Martín.

```
SELECT nombre, apellidos
FROM CLIENTE
WHERE cifcl < (SELECT cifcl
 FROM CLIENTE
 WHERE nombre = 'Juan'
 AND apellido = 'Martín')
```

Empleando ALL

Obtener el nombre y apellidos de los clientes cuyo identificador de cliente es menor que el de los de Barcelona.

```
SELECT nombre, apellidos
FROM CLIENTE
WHERE cifcl < ALL (SELECT cifcl
FROM CLIENTE
WHERE ciudad = 'Barcelona')
```

Empleando ANY

Obtener el nombre y apellidos de los clientes cuyo identificador de cliente es mayor que el de alguno de los de Madrid, y cuyo nombre empieza por a.

```
SELECT nombre, apellidos
FROM CLIENTE
WHERE cifcl > ANY
```

```
                    (SELECT cifcl
FROM CLIENTE
WHERE ciudad = 'Madrid')
        AND nombre LIKE 'A%'
```

Empleando EXISTS

Obtener los identificadores de los clientes que sólo han comprado coches al concesionario

```
1. SELECT cifcl
FROM VENTA va
WHERE NOT EXISTS (SELECT *
 FROM VENTA vb
 WHERE cifc <> 1
 AND va.cifcl = vb.cifcl)
```

Obtener los identificadores de los clientes que han comprado coches al concesionario 1 y a algún otro.

```
SELECT cifcl
FROM VENTA va
WHERE cifc = 1
AND EXISTS (SELECT *
 FROM VENTA vb
 WHERE cifc <> 1
 AND va.cifcl = vb.cifcl)
```