Unidad 5 : Consultas básicas con SQL: DML

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Fecha	Versión	Descripción
28/11/2022	1.0.0	Versión completa

Unidad 5 : Consultas básicas con SQL: DML

1. Introducción

En el lenguaje SQL existen 4 sentencias que forman el DML (Data Manipulation Language, Lenguaje de Manipulación de Datos). Son aquellas sentencias que nos permiten manipular la información que almacenamos en las Bases de Datos.

Las sentencias DML son las siguientes:

- `SELECT` : se utiliza para realizar consultas y extraer información de la base de datos.
- `INSERT` : se utiliza para insertar registros en las tablas de la base de datos.
- `UPDATE` : se utiliza para actualizar los registros de una tabla.
- `DELETE` : se utiliza para eliminar registros de una tabla.

A lo largo de esta unidad (y en la siguiente) nos centraremos en la cláusula `SELECT`. Sin duda es el comando más versátil del lenguaje SQL ya que permite:

- Obtener datos de ciertas columnas de una tabla (proyección).
- Obtener registros (filas) de una tabla de acuerdo con ciertos criterios (selección).
- Mezclar datos de tablas diferentes (asociación, join).

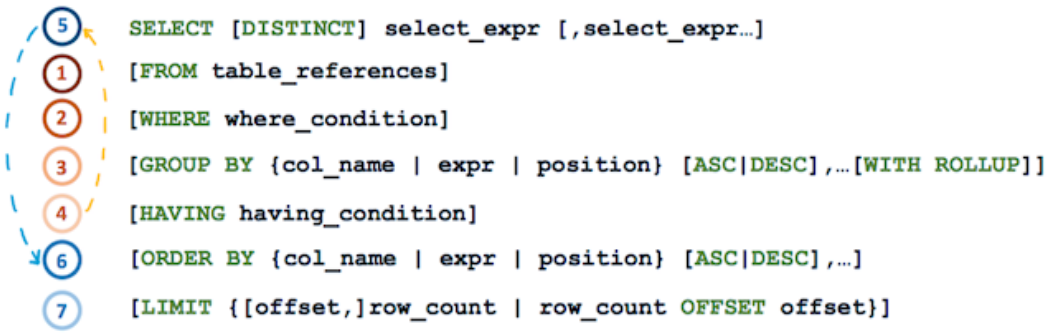
2. Sintaxis básica de la sentencia `SELECT`

Para realizar consultas a una base de datos relacional hacemos uso de la sentencia `SELECT`. La sintaxis básica del comando `SELECT` es la siguiente:

```
SELECT  * | {[ DISTINCT ] columna | expresión [[AS] alias ], ...}
[FROM tabla]
[WHERE condición]
[GROUP BY {columna | expresión | posición} [ASC | DESC], ... ]
[HAVING condición]
[ORDER BY {columna | expresión | posición} [ASC | DESC], ...]
[LIMIT {[offset,] número_filas | número_filas OFFSET offset}]
```

Es muy importante conocer en qué orden se ejecuta cada una de las cláusulas que forman la sentencia `SELECT`. El orden de ejecución es el siguiente:

- Cláusula `FROM`.
- Cláusula `WHERE` (Es opcional, puede ser que no aparezca).
- Cláusula `GROUP BY` (Es opcional, puede ser que no aparezca).
- Cláusula `HAVING` (Es opcional, puede ser que no aparezca).
- Cláusula `SELECT`.
- Cláusula `ORDER BY` (Es opcional, puede ser que no aparezca).
- Cláusula `LIMIT` (Es opcional, puede ser que no aparezca).



Hay que tener en cuenta que el resultado de una consulta siempre será una tabla de datos, que puede tener una o varias columnas y ninguna, una o varias filas.
El hecho de que el resultado de una consulta sea una tabla es muy interesante porque nos permite realizar cosas como almacenar los resultados como una nueva en la base de datos. También será posible combinar el resultado de dos o más consultas para crear una tabla mayor con la unión de los dos resultados. Además, los resultados de una consulta también pueden consultados por otras nuevas consultas.

En los siguientes apartados analizaremos para que sirve cada una de las cláusulas de esta sentencia.

3. Cláusula `SELECT`

La cláusula `SELECT` se utiliza para selecciona las columnas que se quieren visualizar como resultado de la consulta. Es decir, nos permite indicar cuáles serán las columnas que tendrá la tabla de resultados de la consulta que estamos realizando.

Las opciones que podemos indicar son las siguientes:

- El nombre de una columna de la tabla sobre la que estamos realizando la consulta. Será una columna de la tabla que aparece en la cláusula `FROM`.
- Una constante que aparecerá en todas las filas de la tabla resultado.
- Una expresión que nos permite calcular nuevos valores.
- El comodín '*' para indicar que se quieren visualizar todas las columnas de la tabla.

La cláusula `FROM` permite indicar con qué tablas se trabajará en la consulta (en esta unidad, solo aparecerá una tabla).

3.1. Cómo obtener los datos de las columnas de una tabla (`SELECT *`).

Vamos a utilizar la base de datos ***instituto*** para los ejemplos. Te puedes descargar el script de Aules (en el apartado de "*Recursos*").

Esta base de datos está formada por una sola tabla, llamada alumnos, con la información de los alumnos matriculados en un determinado curso:

La sintaxis de creación es:

```sql
CREATE TABLE alumno (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    apellido1 VARCHAR(100) NOT NULL,
    apellido2 VARCHAR(100),
    fecha_nacimiento DATE NOT NULL,
    es_repetidor ENUM('sí', 'no') NOT NULL,
    teléfono VARCHAR(9)
);
```

Y la información que contiene se puede ver en la siguiente tabla:

id	nombre	apellido1	apellido2	fecha_nacimiento	es_repetidor	telefono
1	María	Sánchez	Pérez	1990/12/01	no	NULL
2	Juan	Sáez	Vega	1998/04/02	no	618253876
3	Pepe	Ramírez	Gea	1988/01/03	no	NULL
4	Lucía	Sánchez	Ortega	1993/06/13	sí	678516294
5	Paco	Martínez	López	1995/11/24	no	692735409
6	Irene	Gutiérrez	Sánchez	1991/03/28	sí	NULL
7	Cristina	Fernández	Ramírez	1996/09/17	no	628349590
8	Antonio	Carretero	Ortega	1994/05/20	sí	612345633
9	Manuel	Domínguez	Hernández	1999/07/08	no	NULL
10	Daniel	Moreno	Ruiz	1998/02/03	no	NULL

Vamos a ver qué consultas sería necesario realizar para obtener ***todos los datos de todos los alumnos matriculados en el curso***.

```sql
SELECT *
FROM alumno;
```

El carácter * es un comodín que utilizamos para indicar que queremos seleccionar todas las columnas de la tabla. La consulta anterior devolverá todos los datos de la tabla.
Tenga en cuenta que las palabras reservadas de SQL no son ***case sensitive***, por lo tanto es posible escribir la sentencia anterior de la siguiente forma obteniendo el mismo resultado:

```sql
select *
from alumno;
```

Otra consideración que también debemos tener en cuenta es que una consulta SQL se puede escribir en una o varias líneas. Por ejemplo, la siguiente sentencia tendría el mismo resultado que la anterior:

```sql
SELECT * FROM alumno;
```

 NOTA

`A lo largo del curso vamos a considerar como una buena práctica escribir las consultas SQL en varias líneas, empezando cada línea con la palabra reservada de la cláusula correspondiente que forma la consulta.`

3.2 Cómo obtener los datos de algunas columnas de una tabla

Vamos a ver qué consultas sería necesario realizar para obtener los siguientes datos:

a) Obtener el nombre de todos los alumnos:

```
SELECT nombre
FROM alumno;
```

nombre
María
Juan
Pepe
Lucía
Paco
Irene
Cristina
Antonio
Manuel
Daniel

b) Obtener el nombre y los apellidos de todos los alumnos:

```
SELECT nombre, apellido1, apellido2
FROM alumno;
```

nombre	apellido1	apellido2
María	Sánchez	Pérez
Juan	Sáez	Vega
Pepe	Ramírez	Gea
Lucía	Sánchez	Ortega
Paco	Martínez	López
Irene	Gutiérrez	Sánchez
Cristina	Fernández	Ramírez
Antonio	Carretero	Ortega
Manuel	Domínguez	Hernández
Daniel	Moreno	Ruiz

Tenga en cuenta que el resultado de la consulta SQL mostrará las columnas que haya solicitado, siguiendo el orden en el que se hayan indicado. Por lo tanto la siguiente consulta:

```
SELECT apellido1, apellido2, nombre
FROM alumno;
```

Devolverá lo siguiente:

apellido1	apellido2	nombre
Sánchez	Pérez	María
Sáez	Vega	Juan
Ramírez	Gea	Pepe
Sánchez	Ortega	Lucía
Martínez	López	Paco
Gutiérrez	Sánchez	Irene
Fernández	Ramírez	Cristina
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Moreno	Ruiz	Daniel

3.3 Cómo realizar comentarios en sentencias SQL

Para escribir comentarios en nuestras sentencias SQL podemos hacerlo de diferentes formas.

```
-- Esto es un comentario
SELECT nombre, apellido1, apellido2 -- Esto es otro comentario
FROM alumno;
```

```
/* Esto es un comentario
de varias líneas */
SELECT nombre, apellido1, apellido2 /* Esto es otro comentario */
FROM alumno;
```

```
# Esto es un comentario
SELECT nombre, apellido1, apellido2 # Esto es otro comentario
FROM alumno;
```

3.4 Cómo obtener columnas calculadas

Dada la siguiente base de datos ***tienda***, formada por una sola tabla, llamada ventas, con la información de las ventas realizadas en una determinada tienda:

```
DROP DATABASE IF EXISTS tienda;
CREATE DATABASE tienda CHARACTER SET utf8mb4;
USE tienda;

CREATE TABLE ventas (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    cantidad_comprada INT UNSIGNED NOT NULL,
    precio_por_elemento DECIMAL(7,2) NOT NULL
);

INSERT INTO ventas VALUES (1, 2, 1.50);
INSERT INTO ventas VALUES (2, 5, 1.75);
INSERT INTO ventas VALUES (3, 7, 2.00);
INSERT INTO ventas VALUES (4, 9, 3.50);
INSERT INTO ventas VALUES (5, 6, 9.99);
```

Es posible realizar cálculos aritméticos entre columnas para calcular nuevos valores.

Por ejemplo, supongamos que tenemos la siguiente información en la tabla ventas.

id	cantidad_comprada	precio_por_elemento
1	2	1.50
2	5	1.75
3	7	2.00
4	9	3.50
5	6	9.99

Queremos calcular una nueva columna con el precio total de la venta, que sería equivalente a multiplicar el valor de la columna *cantidad_comprada* por *precio_por_elemento*.
Para obtener esta nueva columna podríamos realizar la siguiente consulta:

```
SELECT id, cantidad_comprada, precio_por_elemento, cantidad_comprada *
precio_por_elemento
FROM ventas;
```

Y el resultado sería el siguiente:

id	cantidad_comprada	precio_por_elemento	cantidad_comprada * precio_por_elemento
1	2	1.50	3.00
2	5	1.75	8.75
3	7	2.00	14.00
4	9	3.50	31.50
5	6	9.99	59.94

3.5 Cómo realizar alias de columnas con AS

Con la palabra reservada `AS` podemos crear alias para las columnas. Esto puede ser útil cuando estamos calculando nuevas columnas a partir de valores de las columnas actuales. En el ejemplo anterior de la tabla que contiene información sobre las ventas, podríamos crear el siguiente alias:

```
SELECT id, cantidad_comprada, precio_por_elemento, cantidad_comprada *
precio_por_elemento AS 'precio total'
FROM ventas;
```

Si el nuevo nombre que estamos creando para el alias contiene espacios en blanco es necesario usar comillas simples.
Al crear este alias obtendremos el siguiente resultado:

id	cantidad_comprada	precio_por_elemento	precio total
1	2	1.50	3.00
2	5	1.75	8.75
3	7	2.00	14.00
4	9	3.50	31.50
5	6	9.99	59.94

3.6 Cómo utilizar funciones de MySQL en la cláusula SELECT

Es posible hacer uso de funciones específicas de MySQL en la cláusula `SELECT`. MySQL nos ofrece funciones matemáticas, funciones para trabajar con cadenas y funciones para trabajar con fechas y horas. Algunos ejemplos de las funciones de MySQL que utilizaremos a lo largo del curso son las siguientes.

Funciones con cadenas

Función	Descripción
`CONCAT`	Concatena cadenas
`CONCAT_WS`	Concatena cadenas con un separador
`LOWER`	Devuelve una cadena en minúscula
`UPPER`	Devuelve una cadena en mayúscula
`SUBSTR`	Devuelve una subcadena

Funciones matemáticas

Función	Descripción
`ABS()`	Devuelve el valor absoluto
`POW(x,y)`	Devuelve el valor de x elevado a y
`SQRT()`	Devuelve la raíz cuadrada
`PI()`	Devuelve el valor del número `PI`
`ROUND()`	Redondea un valor numérico
`TRUNCATE()`	Trunca un valor numérico

Funciones de fecha y hora

Función	Descripción
`NOW()`	Devuelve la fecha y la hora actual
`CURTIME()`	Devuelve la hora actual

En la documentación oficial puede encontrar la referencia completa de todas las funciones y operadores disponibles en MySQL.

Ejemplo: *Obtener el nombre y los apellidos de todos los alumnos en una única columna.*

```
SELECT CONCAT(nombre, apellido1, apellido2) AS nombre_completo
FROM alumno;
```

nombre_completo
MaríaSánchezPérez
JuanSáezVega
PepeRamírezGea
LucíaSánchezOrtega
PacoMartínezLópez
IreneGutiérrezSánchez
CristinaFernándezRamírez
AntonioCarreteroOrtega
ManuelDomínguezHernández
DanielMorenoRuiz

En este caso estamos haciendo uso de la función `CONCAT` de MySQL y la palabra reservada `AS` para crear un alias de la columna y renombrarla como nombre_completo.
La función `CONCAT` de MySQL no añade ningún espacio entre las columnas, por eso los valores de las tres columnas aparecen como una sola cadena sin espacios entre ellas. Para resolver este problema podemos hacer uso de la función `CONCAT_WS` que nos permite definir un carácter separador entre cada columna.

En el siguiente ejemplo haremos uso de la función `CONCAT_WS` y usaremos un espacio en blanco como separador.

```
SELECT CONCAT_WS(' ', nombre, apellido1, apellido2) AS nombre_completo
FROM alumno;
```

nombre_completo
María Sánchez Pérez
Juan Sáez Vega
Pepe Ramírez Gea
Lucía Sánchez Ortega
Paco Martínez López
Irene Gutiérrez Sánchez
Cristina Fernández Ramírez
Antonio Carretero Ortega
Manuel Domínguez Hernández
Daniel Moreno Ruiz

 `Importante`

`La función CONCAT devolverá NULL cuando alguna de las cadenas que está concatenando es igual NULL, mientras que la función CONCAT_WS omitirá todas las cadenas que sean igual a NULL y realizará la concatenación con el resto de cadenas.`

Ejercicios:

1. Obtener el nombre y los apellidos de todos los alumnos en una única columna en minúscula.
2. Obtener el nombre y los apellidos de todos los alumnos en una única columna en mayúscula.
3. Obtener el nombre y los apellidos de todos los alumnos en una única columna. Cuando el segundo apellido de un alumno sea `NULL` se devolverá el nombre y el primer apellido concatenados en mayúscula, y cuando no lo sea, se devolverá el nombre completo concatenado tal y como aparece en la tabla.

3.7 Modificadores ALL, DISTINCT y DISTINCTROW

Los modificadores `ALL` y `DISTINCT` indican si se deben incluir o no filas repetidas en el resultado de la consulta.

1. `ALL` indica que se deben incluir todas las filas, incluidas las repetidas. Es la opción por defecto, por lo tanto no es necesario indicarla.
2. `DISTINCT` elimina las filas repetidas en el resultado de la consulta.
3. `DISTINCTROW` es un sinónimo de `DISTINCT`.

Ejemplo: *En el siguiente ejemplo vamos a ver la diferencia que existe entre utilizar* `DISTINCT` *y no utilizarlo.*

La siguiente consulta mostrará todas las filas que existen en la columna *apellido1* de la tabla alumno.

```
SELECT apellido1
FROM alumno;
```

apellido1
Sánchez
Sáez
Ramírez
Sánchez
Martínez
Gutiérrez
Fernández
Carretero
Domínguez
Moreno

Si en la consulta anterior utilizamos `DISTINCT` se eliminarán todos los valores repetidos que existan.

```
SELECT DISTINCT apellido1
FROM alumno;
```

apellido1
Sánchez
Sáez
Ramírez
Martínez
Gutiérrez
Fernández
Carretero
Domínguez
Moreno

Si en la cláusula `SELECT` utilizamos `DISTINCT` con más de una columna, la consulta seguirá eliminando **todas las filas** repetidas que existan.

Por ejemplo, si tenemos las columnas *apellido1, apellido2 y nombre*, se eliminarán todas las filas que tengan los mismos valores en las tres columnas.

```
SELECT DISTINCT apellido1, apellido2, nombre
FROM alumno;
```

apellido1	apellido2	nombre
Sánchez	Pérez	María
Sáez	Vega	Juan
Ramírez	Gea	Pepe
Sánchez	Ortega	Lucía
Martínez	López	Paco
Gutiérrez	Sánchez	Irene
Fernández	Ramírez	Cristina
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Moreno	Ruiz	Daniel

En este ejemplo no se ha eliminado ninguna fila porque no existen alumnos que tengan los mismos apellidos y el mismo nombre.

4. Cláusula ORDER BY

`ORDER BY` permite ordenar las filas que se incluyen en el resultado de la consulta. La sintaxis de MySQL es la siguiente:

```
[ORDER BY {col_name | expr | position} [ASC | DESC], ...]
```

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1;
```

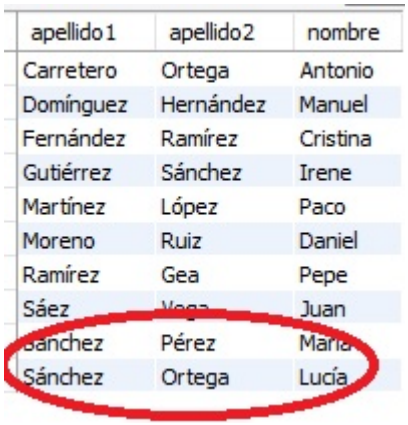
Si no indicamos nada en la cláusula `ORDER BY` se ordenará por defecto de forma ascendente. Por tanto, la consulta anterior es equivalente a esta otra.

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1 ASC;
```

El resultado de ambas consultas será:

apellido1	apellido2	nombre
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Fernández	Ramírez	Cristina
Gutiérrez	Sánchez	Irene
Martínez	López	Paco
Moreno	Ruiz	Daniel
Ramírez	Gea	Pepe
Sáez	Vega	Juan
Sánchez	Pérez	María
Sánchez	Ortega	Lucía

Las filas están ordenadas correctamente por el primer apellido, pero todavía hay que resolver cómo ordenar el listado cuando existen varias filas donde coincide el valor del primer apellido. En este caso tenemos dos filas donde el primer apellido es Sánchez:



apellido1	apellido2	nombre
Carretero	Ortega	Antonio
Domínguez	Hernández	Manuel
Fernández	Ramírez	Cristina
Gutiérrez	Sánchez	Irene
Martínez	López	Paco
Moreno	Ruiz	Daniel
Ramírez	Gea	Pepe
Sáez	Vega	Juan
Sánchez	Pérez	María
Sánchez	Ortega	Lucía

En este caso, para obtener un listado de todos los alumnos ordenados por el primer apellido, segundo apellido y nombre, de forma ascendente, será necesario indicar dichos campos y en ese orden en la cláusula `ORDER BY`, tal y como se muestra a continuación:

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1, apellido2, nombre;
```

En lugar de indicar el nombre de las columnas en la cláusula `ORDER BY` podemos indicar sobre la posición donde aparecen en la cláusula `SELECT`, de modo que la consulta anterior sería equivalente a la siguiente:

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY 1, 2, 3;
```

b) Cómo ordenar de forma descendente

***Ejemplo**: Obtener el nombre y los apellidos de todos los alumnos, ordenados por su primer apellido de forma descendente.*

```
SELECT apellido1, apellido2, nombre
FROM alumno
ORDER BY apellido1 DESC;
```

5. Cláusula LIMIT

`LIMIT` permite limitar el número de filas que se incluyen en el resultado de la consulta. La sintaxis de MySQL es la siguiente:

```
[LIMIT {[offset,] row_COUNT | row_COUNT OFFSET offset}]
```

donde `row_COUNT` es el número de filas que queremos obtener y `offset` el número de filas que nos saltamos antes de empezar a contar. Es decir, la primera fila que se obtiene como resultado es la que está situada en la posición offset + 1.

Dada la siguiente base de datos ***google***:

```sql
DROP DATABASE IF EXISTS google;
CREATE DATABASE google CHARACTER SET utf8mb4;
USE google;

CREATE TABLE resultado (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    descripcion VARCHAR(200) NOT NULL,
    url VARCHAR(512) NOT NULL
);

INSERT INTO resultado VALUES (1, 'Resultado 1', 'Descripción 1', 'http://....');
INSERT INTO resultado VALUES (2, 'Resultado 2', 'Descripción 2', 'http://....');
INSERT INTO resultado VALUES (3, 'Resultado 3', 'Descripción 3', 'http://....');
INSERT INTO resultado VALUES (4, 'Resultado 4', 'Descripción 4', 'http://....');
INSERT INTO resultado VALUES (5, 'Resultado 5', 'Descripción 5', 'http://....');
INSERT INTO resultado VALUES (6, 'Resultado 6', 'Descripción 6', 'http://....');
INSERT INTO resultado VALUES (7, 'Resultado 7', 'Descripción 7', 'http://....');
INSERT INTO resultado VALUES (8, 'Resultado 8', 'Descripción 8', 'http://....');
INSERT INTO resultado VALUES (9, 'Resultado 9', 'Descripción 9', 'http://....');
INSERT INTO resultado VALUES (10, 'Resultado 10', 'Descripción 10', 'http://....')
;
INSERT INTO resultado VALUES (11, 'Resultado 11', 'Descripción 11', 'http://....')
;
INSERT INTO resultado VALUES (12, 'Resultado 12', 'Descripción 12', 'http://....')
;
INSERT INTO resultado VALUES (13, 'Resultado 13', 'Descripción 13', 'http://....')
;
INSERT INTO resultado VALUES (14, 'Resultado 14', 'Descripción 14', 'http://....')
;
INSERT INTO resultado VALUES (15, 'Resultado 15', 'Descripción 15', 'http://....')
;
```

Ejercicios:

Supongamos que queremos mostrar en una página web las filas que están almacenadas en la tabla **resultados**, y queremos mostrar la información en diferentes páginas, donde cada una de las páginas muestra solamente 5 resultados.

4. ¿Qué consulta SQL necesitamos realizar para incluir los primeros 5 resultados de la primera página?
5. ¿Qué consulta necesitaríamos para mostrar resultados de la segunda página?
6. ¿Y los resultados de la tercera página?

6. Cláusula WHERE

La cláusula `WHERE` nos permite añadir filtros a nuestras consultas para seleccionar sólo aquellas filas que cumplen una determinada condición. Estas condiciones se denominan predicados y el resultado de estas condiciones puede ser verdadero, falso o desconocido.
Una condición tendrá un resultado desconocido cuando alguno de los valores utilizados tiene el valor `NULL`.
Podemos diferenciar cinco tipos de condiciones que pueden aparecer en la cláusula `WHERE`:

- Condiciones para comparar valores o expresiones.

- Condiciones para comprobar si un valor está dentro de un rango de valores.
- Condiciones para comprobar si un valor está dentro de un conjunto de valores.
- Condiciones para comparar cadenas con patrones.
- Condiciones para comprobar si una columna tiene valores a `NULL`.

Los operandos usados en las condiciones pueden ser nombres de columnas, constantes o expresiones. Los operadores que podemos usar en las condiciones pueden ser aritméticos, de comparación, lógicos, etc.

6.1 Operadores disponibles en MySQL

A continuación se muestran los operadores más utilizados en MySQL para realizar las consultas.

Operadores aritméticos:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

Operadores de comparación:

Operador	Descripción
<	Menor que
<=	Menor o igual
>	Mayor que
>=	Mayor o igual
<>	Distinto
!=	Distinto
=	Igual que

Operadores lógicos:

Operador	Descripción
`AND`	Y lógica
&&	Y lógica
`OR`	O lógica
`||`	O lógica
`NOT`	Negación lógica
`!`	Negación lógica

***Ejemplos**: Vamos a continuar con el ejemplo de la tabla **alumno** que almacena la información de los alumnos matriculados en un determinado curso. ¿Qué consultas serían necesarias para obtener los siguientes datos?*

a) Obtener el nombre de todos los alumnos que su primer apellido sea Martínez.

```
SELECT nombre
FROM alumno
WHERE apellido1 = 'Martínez';
```

b) Obtener todos los datos del alumno que tiene un id igual a 9.

```
SELECT *
FROM alumno
WHERE id = 9;
```

c) Obtener el nombre y la fecha de nacimiento de todos los alumnos nacieron después del 1 de enero de 1997.

```
SELECT nombre, fecha_nacimiento
FROM alumno
WHERE fecha_nacimiento >= '1997/01/01';
```

Ejercicios: Realice las siguientes consultas teniendo en cuenta la base de datos ***instituto***:

7. Devuelve los datos del alumno cuyo `id` es igual a 1.
8. Devuelve los datos del alumno cuyo `teléfono` es igual a 692735409.
9. Devuelve un listado de todos los alumnos que son repetidores.
10. Devuelve un listado de todos los alumnos que no son repetidores.
11. Devuelve el listado de los alumnos que han nacido antes del 1 de enero de 1993.
12. Devuelve el listado de los alumnos que han nacido después del 1 de enero de 1994.
13. Devuelve el listado de los alumnos que han nacido después del 1 de enero de 1994 y no son repetidores.
14. Devuelve el listado de todos los alumnos que nacieron en 1998.
15. Devuelve el listado de todos los alumnos que **no** nacieron en 1998.

6.2 Operador `BETWEEN`

Sintaxis:

```
expresión [NOT] BETWEEN cota_inferior AND cota_superior
```

Se utiliza para comprobar si un valor está dentro de un rango de valores. Por ejemplo, si queremos obtener los pedidos que se han realizado durante el mes de enero de 2018 podemos realizar la siguiente consulta:

```
SELECT *
FROM pedido
WHERE fecha_pedido BETWEEN '2018-01-01' AND '2018-01-31';
```

Ejercicios:

Realice las siguientes consultas teniendo en cuenta la base de datos ***instituto***.

16. Devuelve los datos de los alumnos que hayan nacido entre el 1 de enero de 1998 y el 31 de mayo de 1998.
17. Devuelve los datos de los alumnos que **no** hayan nacido entre el 1 de enero de 1998 y el 31 de mayo de 1998.

6.3 Operador `IN`

Este operador nos permite comprobar si el valor de una determinada columna está incluido en una lista de valores.

Ejemplo: *Obtener todos los datos de los alumnos que tengan como primer apellido* `Sánchez`, `Martínez` *o* `Domínguez`.

```
SELECT *
FROM alumno
WHERE apellido1 IN (`Sánchez`, `Martínez`, `Domínguez`);
```

Ejemplo: *Obtener todos los datos de los alumnos que* ***no tengan*** *como primer apellido* `Sánchez`, `Martínez` *o* `Domínguez`.

```
SELECT *
FROM alumno
WHERE apellido1 NOT IN (`Sánchez`, `Martínez`, `Domínguez`);
```

6.4 Operador `LIKE`

Sintaxis:

```
columna [NOT] LIKE patrón
```

Se utiliza para comparar si una cadena de caracteres coincide con un patrón. En el patrón podemos utilizar cualquier carácter alfanumérico, pero hay dos caracteres que tienen un significado especial, el símbolo del porcentaje (`%`) y el carácter de subrayado (`_`).

- `%` : Este carácter equivale a cualquier conjunto de caracteres.
- `_` : Este carácter equivale a cualquier carácter.

Ejemplos:

Devuelva un listado de todos los alumnos que su primer apellido empiece por la letra `S`.

```
SELECT *
FROM alumno
WHERE apellido1 LIKE 'S%';
```

Devuelva un listado de todos los alumnos que su primer apellido termine por la letra `z`.

```
SELECT *
FROM alumno
WHERE apellido1 LIKE '%z';
```

Devuelva un listado de todos los alumnos que su nombre tenga el carácter `a`.

```
SELECT *
FROM alumno
WHERE nombre LIKE '%a%';
```

Devuelva un listado de todos los alumnos que tengan un nombre de cuatro caracteres.

```
SELECT *
FROM alumno
WHERE nombre LIKE '____';
```

Devuelve un listado de todos los productos cuyo nombre empieza con estas cuatro letras 'A%BC'.

En este caso, el patrón que queremos buscar contiene el carácter %, por lo tanto, tenemos que usar un carácter de escape.

```
SELECT *
FROM producto
WHERE nombre LIKE 'A$%BC%' ESCAPE '$';
```

Por defecto, se utiliza el carácter '' como carácter de escape. De modo que podríamos escribir la consulta de la siguiente manera.

```
SELECT *
FROM producto
WHERE nombre LIKE 'A\%BC%';
```

6.5 Operadores `IS` e `IS NOT`

Estos operadores nos permiten comprobar si el valor de una determinada columna es `NULL` o no lo es.

Ejemplo:

Obtener la lista de alumnos que tienen un valor `NULL` en la columna `teléfono`.

```
SELECT *
FROM alumno
WHERE teléfono IS NULL;
```

Ejemplo:

Obtener la lista de alumnos que no tienen un valor `NULL` en la columna `teléfono`.

```
SELECT *
FROM alumno
WHERE teléfono IS NOT NULL;
```

7. Funciones disponibles en MySQL

A continuación se muestran algunas funciones disponibles en MySQL que pueden ser utilizadas para realizar consultas.

Las funciones se pueden utilizar en las cláusulas `SELECT`, `WHERE` y `ORDER BY`.

En la documentación oficial puede encontrar la referencia completa de todas las funciones disponibles en MySQL.

7.1 Funciones con cadenas

Función	Descripción
`CONCAT`	Concatena cadenas
`CONCAT_WS`	Concatena cadenas con un separador
`LOWER`	Devuelve una cadena en minúscula
`UPPER`	Devuelve una cadena en mayúscula
`SUBSTR`	Devuelve una subcadena
`LEFT`	Devuelve los caracteres de una cadena, empezando por la izquierda
`RIGHT`	Devuelve los caracteres de una cadena, empezando por la derecha
`CHAR_LENGTH`	Devuelve el número de caracteres que tiene una cadena
`LENGTH`	Devuelve el número de bytes que ocupa una cadena
`REVERSE`	Devuelve una cadena invirtiendo el orden de sus caracteres
`LTRIM`	Elimina los espacios en blanco que existan al inicio de una cadena
`RTRIM`	Elimina los espacios en blanco que existan al final de una cadena
`TRIM`	Elimina los espacios en blanco que existan al inicio y al final de una cadena
`REPLACE`	Permite reemplazar un carácter dentro de una cadena

Ejercicios:

18. Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre de los alumnos y en la segunda, el nombre con todos los caracteres invertidos.
19. Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos y en la segunda, el nombre y los apellidos con todos los caracteres invertidos.
20. Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos en mayúscula y en la segunda, el nombre y los apellidos con todos los caracteres invertidos en minúscula.
21. Devuelve un listado con tres columnas, donde aparezca en la primera columna el nombre y los apellidos de los alumnos, en la segunda, el número de caracteres que tiene en total el nombre y los apellidos y en la tercera el número de bytes que ocupa en total.
22. Devuelve un listado con dos columnas, donde aparezca en la primera columna el nombre y los dos apellidos de los alumnos. En la segunda columna se mostrará una dirección de correo electrónico que vamos a calcular para cada alumno. La dirección de correo estará formada por el nombre y el primer apellido, separados por el carácter `.` y seguidos por el dominio `@iescamp.es`. Tenga en cuenta que

la dirección de correo electrónico debe estar en minúscula. Utilice un alias apropiado para cada columna.

23. Devuelve un listado con tres columnas, donde aparezca en la primera columna el nombre y los dos apellidos de los alumnos. En la segunda columna se mostrará una dirección de correo electrónico que vamos a calcular para cada alumno. La dirección de correo estará formada por el nombre y el primer apellido, separados por el carácter `.` y seguidos por el dominio `@iescamp.es`. Tenga en cuenta que la dirección de correo electrónico debe estar en minúscula. La tercera columna será una contraseña que vamos a generar formada por los caracteres invertidos del segundo apellido, seguidos de los cuatro caracteres del año de la fecha de nacimiento. Utilice un alias apropiado para cada columna.

7.2 Funciones de fecha y hora

Función	Descripción
`NOW()`	Devuelve la fecha y la hora actual
`CURTIME()`	Devuelve la hora actual
`ADDDATE`	Suma un número de días a una fecha y calcula la nueva fecha
`DATE_FORMAT`	Nos permite formatear fechas
`DATEDIFF`	Calcula el número de días que hay entre dos fechas
`YEAR`	Devuelve el año de una fecha
`MONTH`	Devuelve el mes de una fecha
`MONTHNAME`	Devuelve el nombre del mes de una fecha
`DAY`	Devuelve el día de una fecha
`DAYNAME`	Devuelve el nombre del día de una fecha
`HOUR`	Devuelve las horas de un valor de tipo `DATETIME`
`MINUTE`	Devuelve los minutos de un valor de tipo `DATETIME`
`SECOND`	Devuelve los segundos de un valor de tipo `DATETIME`

Ejemplos:

`NOW()` devuelve la fecha y la hora actual.

```
SELECT NOW();
```

`CURTIME()` devuelve la hora actual.

```
SELECT CURTIME();
```

Ejercicios:

24. Devuelva un listado con cuatro columnas, donde aparezca en la primera columna la fecha de nacimiento completa de los alumnos, en la segunda columna el día, en la tercera el mes y en la cuarta el año. Utilice las funciones `DAY`, `MONTH` y `YEAR`.

25. Devuelva un listado con tres columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos, en la segunda el nombre del día de la semana de la fecha de nacimiento y en la tercera el nombre del mes de la fecha de nacimiento.
26. Resuelva la consulta utilizando las funciones `DAYNAME` y `MONTHNAME`.
27. Resuelva la consulta utilizando la función `DATE_FORMAT`.
28. Devuelva un listado con dos columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos y en la segunda columna el número de días que han pasado desde la fecha actual hasta la fecha de nacimiento. Utilice las funciones `DATEDIFF` y `NOW`. Consulte la documentación oficial de MySQL para `DATEDIFF`.
29. Devuelva un listado con dos columnas, donde aparezca en la primera columna la fecha de nacimiento de los alumnos y en la segunda columna la edad de cada alumno/a. La edad (aproximada) la podemos calcular realizando las siguientes operaciones:
    - Calcula el número de días que han pasado desde la fecha actual hasta la fecha de nacimiento. Utilice las funciones `DATEDIFF` y `NOW`.
    - Divida entre 365.25 el resultado que ha obtenido en el paso anterior. (El 0.25 es para compensar los años bisiestos que han podido existir entre la fecha de nacimiento y la fecha actual).
    - Trunca las cifras decimales del número obtenido.

7.3 Funciones matemáticas

Función	Descripción
`ABS()`	Devuelve el valor absoluto
`POW(x,y)`	Devuelve el valor de x elevado a y
`SQRT`	Devuelve la raíz cuadrada
`PI()`	Devuelve el valor del número `PI`
`ROUND`	Redondea un valor numérico
`TRUNCATE`	Trunca un valor numérico
`CEIL`	Devuelve el **entero inmediatamente superior o igual**
`FLOOR`	Devuelve el **entero inmediatamente inferior o igual**
`MOD`	Devuelve el resto de una división

Ejemplos:

`ABS` devuelve el valor absoluto de un número.

```
SELECT ABS(-25);
25
```

`POW(x, y)` devuelve el valor de x elevado a y.

```
SELECT POW(2, 10);
1024
```

`SQRT` devuelve la raíz cuadrada de un número.

```
SELECT SQRT(1024);
32
```

`PI()` devuelve el valor del número `PI`.

```
SELECT PI();
3.141593
```

`ROUND` redondea un valor numérico

```
SELECT ROUND(37.6234);
38
```

`TRUNCATE` Trunca un valor numérico.

```
SELECT TRUNCATE(37.6234, 0);
37
```

8. Consultas resumen

8.1 Funciones Agregadas

Ya hemos visto cómo hacer cálculos con los datos de una consulta e incluso como utilizar funciones en esos cálculos. Pero hasta ahora las funciones utilizadas solo podían usar información procedente de datos de la misma fila. Es decir, no podíamos sumar, por ejemplo, datos procedentes de distintas filas. Sin embargo, hacer cálculos con datos de diferentes filas es una necesidad muy habitual.

Las **funciones de agregación** son las encargadas de realizar cálculos en vertical (usando datos de diferentes filas) en lugar de en horizontal (usando datos procedentes de la misma fila en la que vemos el resultado). Estas funciones realizan una operación específica sobre todas las filas de un grupo.

Las funciones de agregación más comunes son:

Función	Descripción
`MAX(expr)`	Valor máximo del grupo
`MIN(expr)`	Valor mínimo del grupo
`AVG(expr)`	Valor medio del grupo
`SUM(expr)`	Suma de todos los valores del grupo
`COUNT(*)`	Número de filas que tiene el resultado de la consulta
`COUNT(columna)`	Número de valores no nulos que hay en esa columna

Ejemplos:

Supongamos que tenemos los siguientes valores en la tabla `trabajador`:

cod_emp	nombre	apellido1	apellido2	fecha_nacimiento	salario	teléfono
1	María	Sánchez	Pérez	1990/12/01	1200	`NULL`
2	Juan	Sáez	Vega	1998/04/02	1500	618253876
3	Pepe	Ramírez	Gea	1988/01/03	1650	`NULL`
4	Lucía	López	Ruiz	1993/06/13	980	678516294

a) La siguiente consulta devuelve el salario máximo de la tabla (**1650**):

```
SELECT MAX(salario)
FROM trabajador;
```

b) La siguiente consulta devuelve el salario mínimo de la tabla (**980**):

```
SELECT MIN(salario)
FROM trabajador;
```

c) La siguiente consulta nos devuelve la media de los salario de la tabla (**1332,5**):

```
SELECT AVG(salario)
FROM trabajador;
```

d) Las siguiente consulta nos devuelve la suma de los salarios de los trabajadores de la tabla (**5330**):

```
SELECT SUM(salario)
FROM trabajador;
```

En la documentación oficial de MySQL puede encontrar una lista completa de todas las funciones de agregación que se pueden usar.

 `Importante`

`Las funciones de agregación sólo se pueden usar en las cláusulas SELECT Y HAVING.`

8.1.1 Diferencia entre `COUNT(*)` y `COUNT(columna)

- `COUNT(*)` : Calcula el número de filas que tiene el resultado de la consulta.
- `COUNT(columna)` : Cuenta el número de valores no nulos que hay en esa columna.

 `Importante`

`Tener en cuenta la diferencia que existe entre las funciones COUNT(*) y COUNT(columna), ya que devolverán resultados diferentes cuando haya valores nulos en la columna que estamos usando en la función.`

Siguiendo con nuestra tabla ejemplo `trabajador` , la consulta:

```
SELECT COUNT(teléfono)
FROM trabajador;
```

devolverá:

COUNT(teléfono)
2

mientras que la consulta:

```
SELECT COUNT(*)
FROM trabajador;
```

devolverá:

COUNT(*)
4

8.1.2 Contar valores distintos `COUNT(DISTINCT columna)`

Supongamos que tenemos los siguientes valores en la tabla `producto`:

id	nombre	precio	código_fabricante
1	Disco duro SATA3 1TB	86	5
2	Memoria RAM DDR4 8GB	120	4
3	Disco SSD 1 TB	150	5
4	GeForce GTX 1050Ti	185	5

Y nos piden calcular el número de valores distintos de código de fabricante que aparecen en la tabla `producto`.

```
SELECT COUNT(DISTINCT código_fabricante)
FROM producto;
```

Esta consulta devolverá:

COUNT(DISTINCT código_fabricante)
2

Un error muy común que se comete a la hora de contar filas distintas es la colocación incorrecta de `DISTINCT`.

```
SELECT DISTINCT COUNT(codigo_fabricante)
FROM producto;
```

8.2 Cláusula `GROUP BY`

Es muy común utilizar consultas en las que se desee agrupar los datos a fin de realizar cálculos en vertical, es decir calculados a partir de datos de distintos registros. Para ello se utiliza la cláusula `GROUP BY` que permite indicar en base a qué registros se realiza la agrupación. En el apartado `GROUP BY`, se indican las columnas por las que se agrupa. La función de este apartado es crear un único registro por cada valor distinto en las columnas del grupo.

La cláusula `GROUP BY` nos permite crear **grupos de filas** que tienen los mismos valores en las columnas por las que se desea agrupar.

Ejemplos:

Supongamos que tenemos los siguientes valores en la tabla `existencias`:

Tipo	Modelo	N_Almacen	Cantidad
AR	6	1	2500
AR	6	2	5600
AR	6	3	2430
AR	9	1	250
AR	9	2	4000
AR	9	3	678
AR	15	1	5667
AR	20	3	43
BI	10	2	340
BI	10	3	23
BI	38	1	1100
BI	38	2	540
BI	38	3	700

Si ejecutamos la siguiente consulta:

```
SELECT tipo,modelo
FROM existencias
GROUP BY tipo,modelo;
```

Obtendremos el siguiente resultado:

Tipo	Modelo
AR	6
AR	9
AR	15
AR	20
BI	10
BI	38

Los datos se resumen. Por cada `tipo` y `modelo` distintos se creará un grupo de modo que solo aparece una fila por cada grupo. Los datos *`n_almacen`* y *`cantidad`* no están disponibles ya que varían en cada grupo.

Solo se podrán mostrar los datos agrupados o datos sobre los que realicemos cálculos.

Es decir, esta consulta es errónea:

```
SELECT tipo,modelo, cantidad
FROM existencias
GROUP BY tipo,modelo;
```

La razón es el uso de la columna *`cantidad`*, en el `SELECT` como no se ha agrupado por ella y no es un cálculo sobre *`tipo`* y/o *`modelo`* (las columnas por la que se está agrupando), se produce el error.

Lo normal es agrupar para obtener cálculos sobre cada grupo. Por ejemplo podemos modificar la consulta anterior de esta forma:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo;
```

y se obtiene este resultado:

Tipo	Modelo	SUM(Cantidad)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI	10	363
BI	38	1740

Se suman las cantidades para cada grupo.

Igualmente podríamos realizar cualquier otro tipo de cálculo (**AVG**, **COUNT**,...).

8.3 Cláusula `HAVING`

A veces se desea restringir el resultado de una expresión agrupada. La cláusula `HAVING` nos permite crear **filtros** sobre los grupos de filas que tienen los mismos valores en las columnas por las que se desea agrupar.

Por ejemplo, siguiendo con la tabla `existencias`, podríamos tener esta idea:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE SUM(Cantidad)>500
GROUP BY tipo,modelo;
```

Lo que se espera es que solo aparezcan los grupos cuya suma de la cantidad sea menor de 500. Pero, en su lugar, el SGBD devolvería un error, del tipo `... función de grupo no permitida aquí ...`

La razón reside en el orden en el que se ejecutan las cláusulas de la instrucción `SELECT`.

EL SGBD calcula primero el `WHERE` y luego los grupos (cláusula `GROUP BY`); por lo que no podemos usar en el `WHERE` la función de cálculo de grupos `SUM`, porque es imposible en ese momento conocer el resultado de la **SUMA** al no haberse establecido aún los grupos.

La solución es utilizar otra cláusula: `HAVING`, que se ejecuta una vez realizados los grupos. Es decir, si queremos ejecutar condiciones sobre las funciones de totales, se debe hacer en la cláusula `HAVING`.

La consulta anterior quedaría:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

Eso no implica que no se pueda usar `WHERE`. Ésta expresión sí es válida:

```
SELECT tipo,modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE tipo!='AR'
GROUP BY tipo,modelo
HAVING SUM(Cantidad)>500;
```

En definitiva, el orden de ejecución de la consulta marca lo que se puede utilizar con `WHERE` y lo que se puede utilizar con `HAVING`.

9. Referencias

- Wikibook SQL Exercises.
- Tutorial SQL de w3resource.
- MySQL Join Types by Steve Stedman.
- **Bases de Datos**. 2ª Edición. Grupo editorial Garceta. Iván López Montalbán, Manuel de Castro Vázquez y John Ospino Rivas.
- Apuntes Bases de datos de Codeandcoke.
- Apuntes Bases de datos del IES Luis Vélez de Guevara

10. Licencia