

Unidad 4: Realización de consultas

Índice de contenidos

1	Introducción.....	2
2	La sentencia <i>SELECT</i>	4
2.1	Orden de ejecución de las cláusulas de la sentencia <i>SELECT</i>	5
2.2	Cláusula <i>SELECT</i>	5
2.3	Cláusula <i>FROM</i>	6
2.4	Cláusula <i>WHERE</i>	7
2.5	Cláusula <i>GROUP BY</i> y <i>HAVING</i>	7
2.6	Cláusula <i>ORDER BY</i>	8
2.7	Cláusula <i>LIMIT</i>	9
2.8	Comentarios en sentencias <i>SQL</i>	9
3	Operadores.	10
3.1	Operadores de comparación.	10
3.2	Operadores aritméticos.	12
3.3	Operadores de concatenación.	12
3.4	Operadores lógicos.	13
3.5	Precedencia.	14
4	Consultas calculadas.	14
5	Funciones.	15
5.1	Funciones numéricas.	15
5.2	Funciones de cadena de caracteres.	16
5.3	Funciones de manejo de fechas y/o horas.	17
5.4	Funciones de conversión.	19
5.5	Funciones de listas.	20
5.6	Otras funciones: <i>NVL</i> , <i>IFNULL</i> , <i>COALESCE</i> e <i>IF</i>	20
6	Consultas resumen.	21
6.1	Funciones de agregado: <i>SUM</i> y <i>COUNT</i>	22
6.2	Funciones de agregado: <i>MIN</i> y <i>MAX</i>	23
6.3	Funciones de agregado: <i>AVG</i> , <i>VARIANCE</i> y <i>STD</i>	23
6.4	Agrupamiento de registros.	23
7	Consultas multitabla.	26
7.1	Composiciones cruzadas (producto cartesiano).	27
7.2	Composiciones internas (<i>INNER JOIN</i>).	27
7.3	Composiciones externas (<i>LEFT</i> , <i>RIGHT</i> y <i>FULL OUTER JOIN</i>).	31
7.4	Composiciones en la versión <i>SQL99</i> (resumen).	33
8	Otras consultas multitabla: <i>UNION</i> , <i>INTERSECT</i> y <i>EXCEPT</i>	34
9	Subconsultas.	35
9.1	Subconsultas en la cláusula <i>SELECT</i>	35
9.2	Subconsultas en la cláusula <i>FROM</i>	35
9.3	Subconsultas en la cláusula <i>WHERE</i>	36
9.3.1	Subconsultas que devuelven un único valor.	37
9.3.2	Subconsultas que devuelven más de un valor.	38
9.3.3	Comprobar si la subconsulta devuelve o no resultados.	39
9.4	Subconsultas en la cláusula <i>HAVING</i>	40
10	Rendimiento de consultas.	41
10.1	Introducción.	41
10.2	Actualizar y reordenar índices.	42
10.3	Optimización de consultas.	43
10.4	Optimizaciones.	44
10.4.1	Evitar <i>FULL SCAN</i>	44
10.4.2	Uso correcto de los índices.	44
10.4.3	Sentencias <i>OR</i>	44
10.4.4	<i>GROUP/ORDER BY</i>	44
10.4.5	Tablas derivadas versus subconsultas.	45
10.4.6	<i>INNER JOIN</i> con <i>GROUP/ORDER BY</i>	45
10.4.7	La sentencia <i>EXISTS</i>	45
10.4.8	Conclusiones.	45
11	Anexo I. <i>PIVOT</i> en <i>MariaDB</i>	46
12	Anexo II. Búsquedas de texto completo (<i>full-text</i>).	47

1 Introducción.

En unidades anteriores se ha visto que **SQL** es un conjunto de sentencias u órdenes que se necesitan para acceder a los datos. Este lenguaje es utilizado por la mayoría de las aplicaciones donde se trabaja con datos para acceder a ellos (es la **vía de comunicación entre el usuario/programa y el SGBD**).

SQL nació a partir de la publicación "A relational model of data for large shared data banks" de Edgar Frank Codd. IBM aprovechó el modelo que planteaba Codd para desarrollar un lenguaje acorde con el recién nacido modelo relacional, a este primer lenguaje se le llamó **SEQUEL** (*Structured English QUery Language*). Con el tiempo **SEQUEL** se convirtió en **SQL** (*Structured Query Language*). En 1979, la empresa *Relational Software* sacó al mercado la primera implementación comercial de SQL. Esa empresa es la que hoy se conoce como *Oracle*.

Actualmente SQL sigue siendo el **estándar en lenguajes de acceso a BD relacionales**.

En 1992, ANSI e ISO completaron la estandarización de SQL y se definieron las sentencias básicas que debía contemplar SQL para que fuera estándar. A este SQL se le denominó **ANSI-SQL** o **SQL92**.

Hoy en día todas las BD relacionales cumplen con este estándar, eso sí, cada fabricante añade sus mejoras al lenguaje SQL.

Para la **manipulación de datos** se utilizarán las sentencias del **DML** (*Lenguaje de Manipulación de Datos*). Este conjunto de sentencias está

orientado a **consultas y manejo de datos** de los objetos creados. Básicamente consta de **cuatro sentencias**: **SELECT**, **INSERT**, **DELETE** y **UPDATE**. En esta unidad se estudiará la **sentencia para consultas SELECT**. La primera fase del trabajo con cualquier BD comienza con sentencias **DDL** (*Lenguaje de Definición de Datos*), puesto que **antes de poder almacenar y recuperar información se debieron definir las estructuras donde agrupar la información (tablas)**.

Como se ha visto en la unidad anterior, las sentencias SQL que se verán a lo largo de la unidad pueden ser ejecutadas desde *HeidiSQL* o *phpMyAdmin*, siempre y cuando se tenga un usuario con permisos para ello. También se pueden ejecutar las sentencias SQL desde *MySQL Client*, al cual se puede acceder desde *Inicio* → *Todos los programas* → *MariaDB* → *MySQL Client* (o bien desde la *Shell* de *XAMPP*).

Si se opta por utilizar *MySQL Client*, lo primero será **conectarse al servidor de BD** utilizando un nombre de usuario con los permisos necesarios:

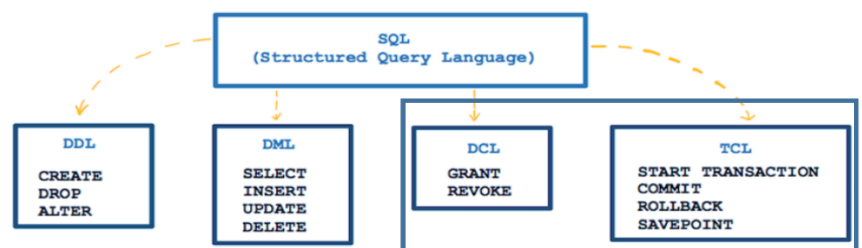
```
mysql [-h nombre_o_ip_servidor] -u nombre_usuario -p
```

Seguidamente, se solicitará la contraseña correspondiente a dicho usuario.

Una vez conectado habrá que **seleccionar la BD** utilizando la sentencia:

```
USE nombre_base_datos;
```

Seguidamente ya se podrán introducir las sentencias SQL que se requieran para obtener el resultado deseado.



```

MySQL Client (MariaDB 10.5 (x64)) - mariadb -h localhost -u root -p
C:\Program Files\MariaDB 10.5\bin>mariadb -h localhost -u root -p
Enter password: ****
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 14
Server version: 10.5.4-MariaDB mariadb.org binary distribution
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use prueba;
Database changed
MariaDB [prueba]>
  
```

En los archivos "BD-U4.- Conferencias.sql", "BD-U4.- Conferencias2.sql" y "BD-U4.- Script Pokemon.sql" se tienen los scripts SQL que permitirán crear las BDs con las que se trabajarán en esta unidad.

Lo primero que se va a hacer es **crear la BD "conferencias" con todo lo necesario y un usuario "conferencias"** que va a tener todos los permisos sobre dicha BD. Para ello los **pasos a seguir** en *HeidiSQL* son:

- 1) **Conectarse al servidor MariaDB** utilizando para ello el usuario "root" (recordar que el servidor de BD

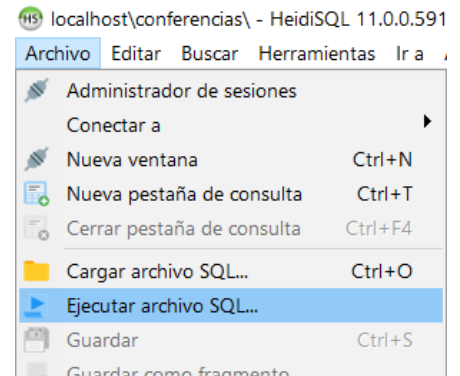
MariaDB se debe estar ejecutando, se debe haber iniciado previamente).

- 2) Seleccionar la **opción** del menú **Archivo** → **Ejecutar archivo SQL...**
- 3) En la **ventana Abrir** que se muestra **seleccionar** el archivo **“BD-U4.- Conferencias.sql”** y pulsar sobre el botón **Abrir**.
- 4) Al ejecutar el archivo SQL automáticamente **se creará la BD**, un **usuario conferencias** con **clave conferencias** que **tendrá todos los permisos sobre la BD creada**, las **tablas necesarias de la BD** y se **insertarán algunos registros de prueba**.

Para crear la **BD conferencias2** y el usuario **conferencias2** con clave **conferencias2**, así como la **BD pokemon** y el usuario **pokemon** con clave **pokemon** seguir los mismos pasos.

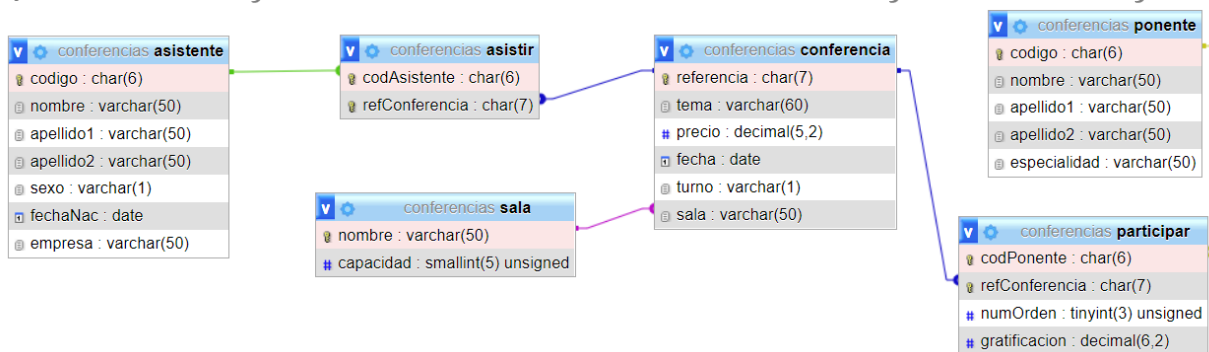
Cargar script SQL desde la *Shell*: `mysql -u root -p < archivo.sql`

Crear script SQL a partir de una *BD* desde la *Shell*: `mysqldump -u root -p base_datos > archivo.sql`

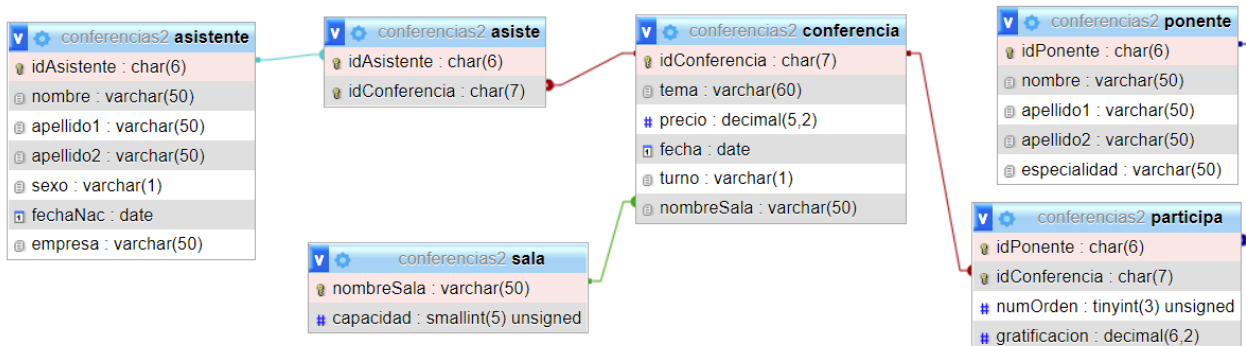


El **esquema relacional** correspondiente a cada una de las *BD* anteriores se muestra en las imágenes siguientes:

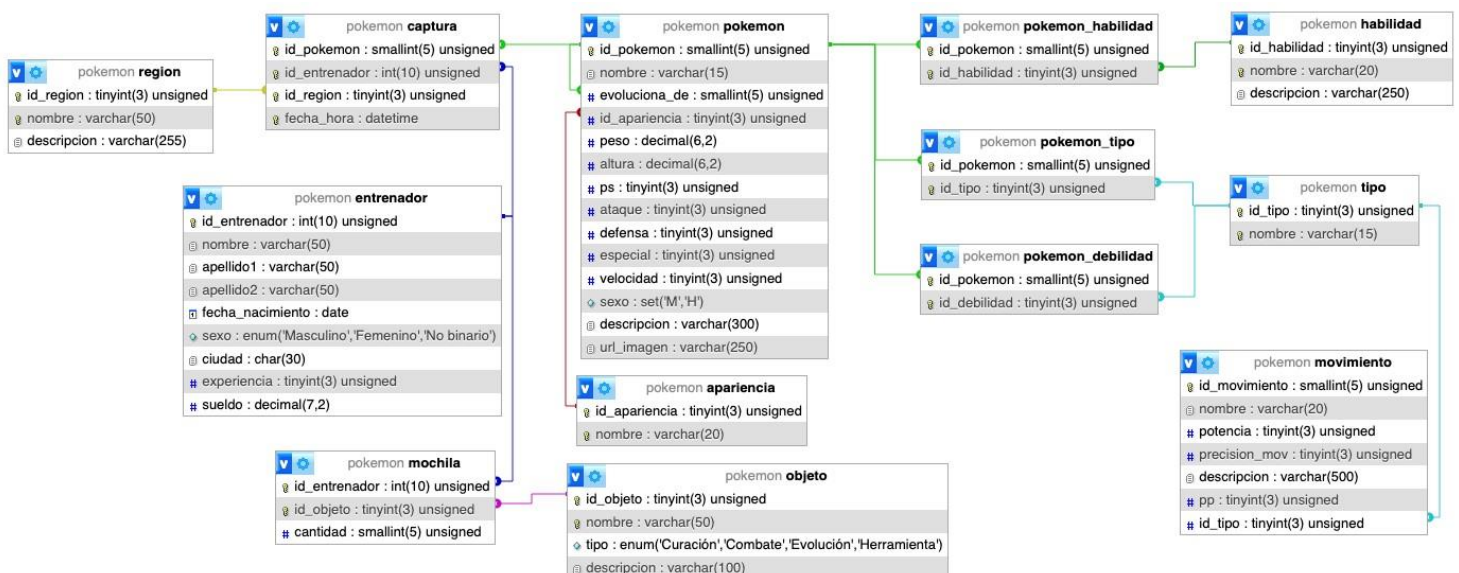
❖ **conferencias** Los ejercicios de esta unidad utilizarán la *BD* “conferencias” o “conferencias2”



❖ **conferencias2**



❖ **pokemon** Los ejemplos propuestos en esta unidad utilizarán la *BD* “pokemon”



2 La sentencia *SELECT*.

Para **recuperar los datos, de una o varias tablas** de una *BD* relacional se utilizará la **sentencia *SELECT***. Las **partes** que la conforman son:


- ✓ Cláusula ***SELECT*** seguida de la descripción de lo que se desea obtener, es decir, de **los nombres de las columnas o expresiones de selección que se quiere que se muestren separados por comas (",")**. Esta parte es **obligatoria**.
- ✓ Cláusula ***FROM*** seguida del nombre de la o las tablas de las que proceden las columnas que aparecen en la cláusula *SELECT*, es decir, de dónde se van a extraer los datos. Esta parte es **opcional**.
- ✓ Cláusula ***WHERE*** seguida de un **criterio de selección o condición**. Esta parte es **opcional**.
- ✓ Cláusulas ***GROUP BY* y *HAVING*** para **agrupar filas cuando se tengan columnas o valores calculados en común**. Esta parte es **opcional**.
- ✓ Cláusula ***ORDER BY*** seguida por un **criterio de ordenación**. Esta parte es **opcional**.
- ✓ Cláusula ***LIMIT*** seguida de un **desplazamiento y un cierto número de filas**. Esta parte es **opcional**.

```
SELECT
[ALL | DISTINCT | DISTINCTROW]
[HIGH_PRIORITY]
[STRAIGHT_JOIN]
[SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
[SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
select_expr [, select_expr ...]
[ FROM table_references
[WHERE where_condition]
[GROUP BY {col_name | expr | position} [ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_condition]
[ORDER BY {col_name | expr | position} [ASC | DESC], ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset} [ROWS EXAMINED rows_limit] } |
[OFFSET start { ROW | ROWS }]
[FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } { ONLY | WITH TIES } ]
procedure|[PROCEDURE procedure_name(argument_list)]
[INTO OUTFILE 'file_name' [CHARACTER SET charset_name] [export_options] |
INTO DUMPFILE 'file_name' | INTO var_name [, var_name] ]
[FOR UPDATE lock_option | LOCK IN SHARE MODE lock_option]

export_options:
[ {FIELDS | COLUMNS}
[TERMINATED BY 'string']
[[OPTIONALLY] ENCLOSED BY 'char']
[ESCAPED BY 'char']
]
[LINES
[STARTING BY 'string']
[TERMINATED BY 'string']
]

lock_option:
[WAIT n | NOWAIT | SKIP LOCKED]
```

Una primera sintaxis básica de la sentencia *SELECT* quedaría de la siguiente forma:



```
5 SELECT [DISTINCT] select_expr [,select_expr...]
1 [FROM table_references]
2 [WHERE where_condition]
3 [GROUP BY {col_name | expr | position} [ASC|DESC],...[WITH ROLLUP]]
4 [HAVING having_condition]
6 [ORDER BY {col_name | expr | position} [ASC|DESC],...]
7 [LIMIT {[offset,]row_count | row_count 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 que impone el estándar *SQL* se puede ver en la imagen anterior y se trata en el punto siguiente).

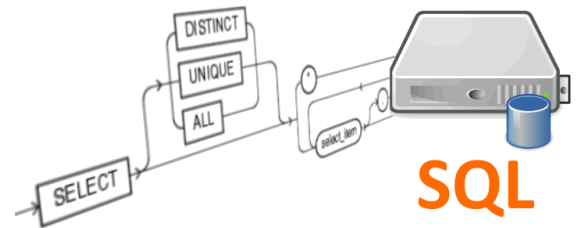
Hay que tener en cuenta que **el resultado de una consulta siempre será una tabla de datos**, que puede tener **una o varias columnas y ninguna, una o varias filas**.

El hecho de que el **resultado de una consulta** sea **una tabla** es muy interesante porque permite realizar cosas como almacenar los resultados en una nueva tabla de la *BD*. 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 ser consultados por otras nuevas consultas.

Si se incluye la **cláusula *DISTINCT*** después de *SELECT*, **se suprimirán aquellas filas del resultado que tengan igual valor que otras** (filas duplicadas).

2.1 Orden de ejecución de las cláusulas de la sentencia *SELECT*.

El lenguaje *SQL* es un potente lenguaje declarativo usado en las *BD* relacionales como *MariaDB*, *MySQL* o *PostgreSQL*. En el lenguaje *SQL* se declara que datos se quieren recuperar, qué condiciones han de cumplir y qué funciones se aplican a los datos pero no se define cómo han de recuperarse, es el *SGBD* el que decide cómo guardarlos e interpretando la sentencia *SELECT* cómo recuperarlos.



La **sentencia *SELECT*** se compone de varias cláusulas que se interpretan siguiendo una secuencia de operaciones tal que:

1. **FROM**: obtener los registros de todas las tablas indicadas. Si hay subconsultas en la cláusula *FROM* son evaluadas primero.
2. **JOIN**: realizar todas las posibles combinaciones descartando aquellas combinaciones que no cumplen las condiciones *JOIN* o estableciendo *NULL* en caso de *OUTER JOIN*.
3. **WHERE**: filtrar las combinaciones que cumplen las condiciones de la cláusula *WHERE*.
4. **GROUP BY**: construir los grupos basados en las expresiones de la lista de la cláusula *GROUP BY*.
5. **HAVING**: filtrar los grupos que cumplen las condiciones de la cláusula *HAVING*.
6. **SELECT**: evaluar las expresiones de la lista *SELECT* para seleccionar los datos.
7. **DISTINCT**: eliminar filas duplicadas si se especifica *DISTINCT*.
8. **UNION, EXCEPT, INTERSECT**: aplicar las operaciones *UNION*, *EXCEPT* e *INTERSECT*.
9. **ORDER BY**: ordenar las filas de acuerdo a la cláusula *ORDER BY*.
10. **OFFSET, LIMIT**: descartar los registros de acuerdo a *OFFSET* y *LIMIT*.

SQL queries run in this order

FROM + JOIN
↓
WHERE
↓
GROUP BY
↓
HAVING
↓
SELECT
↓
ORDER BY
↓
LIMIT

Este es el orden general, pero el algoritmo del planificador puede optimizar estos pasos realizándose en diferente orden e incluso simultáneamente. Por ejemplo, si se especifica un límite de 1 no es necesario obtener todas las filas de las tablas indicadas en la cláusula *FROM*, sino solo una que cumpla la condición *WHERE*.

Es importante tener en cuenta que no todas las cláusulas son obligatorias en la sentencia *SELECT*. Además, es posible que algunos *SGBD* específicos tengan características adicionales o permitan un orden de cláusulas ligeramente diferente, por lo que siempre es bueno consultar la documentación correspondiente.

*Los *SGBD* en realidad no ejecutan consultas literalmente en este orden porque implementan un montón de optimizaciones para hacer que las consultas se ejecuten más rápido.*

Comprender el orden de las consultas puede ayudar a diagnosticar por qué una consulta no se ejecutará e incluso a optimizar las consultas para que se ejecuten más rápido.

2.2 Cláusula *SELECT*.

La cláusula *SELECT* permite indicar cuáles serán las columnas que tendrá la tabla de resultados de la consulta que se está realizando. Las **opciones que se pueden indicar** (*select_expr*) son las siguientes:

- ✓ El **nombre de las columnas** sobre las que se está realizando la consulta, las cuales deben formar parte de las tablas que aparecen en la cláusula *FROM*.
- ✓ Una **constante** que aparecerá en todas las filas de la tabla resultado.
- ✓ Una **expresión** que permita calcular nuevos valores.
- ✓ Una **subconsulta** escalar (*SELECT*) que devuelva exactamente un valor de columna de una fila, aunque no se recomienda y no se especifica en la documentación. Si la subconsulta devuelve 0 filas, el valor de la expresión de la subconsulta escalar es *NULL* y si la subconsulta devuelve más de una fila, *MariaDB* devuelve un error. Es recomendable asignar un alias a la subconsulta y obligatorio incluirla entre ().

Se ha visto que a continuación de la sentencia SELECT se deben especificar cada una de las columnas que se quieren seleccionar, además, se debe **tener en cuenta** lo siguiente:

- ✓ Se puede **nombrar a las columnas anteponiendo el nombre de la tabla de la que proceden**, pero esto es **opcional, a no ser que se estén consultando campos de distintas tablas que tengan el mismo nombre**. Ej.:

```
SELECT pokemon.nombre, pokemon.peso FROM pokemon;
```

- ✓ Si se quieren **incluir todas las columnas de una o varias tablas** se puede **utilizar el comodín asterisco (*)**. Ej.:

```
SELECT * FROM pokemon;
```

- ✓ Se pueden **poner alias a los nombres de las columnas**. Cuando se consulta una BD, los nombres de las columnas se usan como cabeceras de presentación. Si éste resulta largo, corto o poco descriptivo, se puede usar un alias. Para ello **a continuación del nombre de la columna poner entre comillas dobles el alias que se quiere dar a esa columna** (si el nombre no contiene espacios no son necesarias las comillas). **Entre el nombre de la columna y el alias se puede incluir la palabra reservada AS**. Ej.:

```
SELECT nombre "Nombre Pokémon", peso "Peso kg." FROM pokemon;
SELECT nombre AS "Nombre Pokémon", peso AS "Peso kg." FROM pokemon;
SELECT nombre AS nombre_pokemon, peso AS peso_kg FROM pokemon;
```

- ✓ También se puede **sustituir el nombre de las columnas por constantes, expresiones o funciones SQL**. Ej.:


```
SELECT 4*3/100 "Expresión", peso, peso*0.5 "Mitad del peso", PI() Pi FROM pokemon;
```

Hay que tener en cuenta que **las palabras reservadas de SQL no son case sensitive**, por lo tanto, es posible escribir la sentencia `"SELECT * FROM pokemon;"` de la siguiente forma `"select * from pokemon;"` obteniendo el mismo resultado. En caso de trabajar en sistemas GNU/Linux con MariaDB el nombre de las tablas si son case sensitive (existe la variable del sistema `lower_case_table_names`, que afecta a cómo el servidor maneja la sensibilidad a mayúsculas y minúsculas en los nombres de tablas, el valor 0 indica que es sensible a mayúsculas/minúsculas y el valor 1 que no es sensible → `SELECT @@lower_case_table_names;`).

Otra consideración a tener en cuenta es que **una consulta SQL se puede escribir en una o varias líneas**. Es 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. Ej.:

```
SELECT nombre AS "Nombre Pokémon", peso AS "Peso kg."
FROM pokemon;
```

El **resultado de la consulta SQL mostrará las columnas que se hayan solicitado, siguiendo el orden en el que se hayan indicado en la cláusula SELECT** (no confundir con el orden de las filas, en este caso es el de las columnas).



Nombre Pokémon	Peso kg.
Bulbasaur	6,9
Ivysaur	13,0
Venasaur	100,0
Charmender	2,5

Los **modificadores ALL, DISTINCT y DISTINCTROW** indican si se deben incluir o no filas repetidas en el resultado de la consulta:

- ✓ **ALL**: indica que **se deben incluir todas las filas, incluidas las repetidas**. Es la **opción por defecto**, por lo tanto, **no es necesario indicarla**.
- ✓ **DISTINCT**: **elimina las filas repetidas en el resultado** de la consulta (filas que contienen los mismos valores en todas sus columnas).
- ✓ **DISTINCTROW**: es un **sinónimo de DISTINCT** (hace lo mismo).

Ejemplos:

```
SELECT nombre FROM pokemon; -- Mismo resultado que si se hubiera indicado el modificador ALL
SELECT DISTINCT sexo FROM pokemon; -- Mismo resultado que si se utiliza DISTINCTROW
SELECT DISTINCTROW sexo FROM pokemon; -- Igual resultado que si se utiliza DISTINCT
SELECT ALL nombre, peso, sexo FROM pokemon; -- El modificador ALL se puede omitir
SELECT nombre, (SELECT peso FROM pokemon WHERE pokemon.id_pokemon = pok.id_pokemon) AS peso
FROM pokemon AS pok; -- Subconsulta escalar
```

2.3 Cláusula FROM.

En la sentencia SELECT se debe **establecer de dónde se obtienen las columnas que se van a seleccionar**, para ello se dispone de la **cláusula FROM**. En la cláusula FROM se definen los **nombres de las tablas de las que proceden las columnas** (table_references). Las **consultas multitabla se estudiarán en el punto 7 de esta unidad**.

También se puede asociar un alias a las tablas para abreviar, en este caso no es necesario que se encierre entre comillas (el alias se podrá utilizar para hacer referencia a la tabla en otras cláusulas).

Ejemplos:

```
SELECT * FROM pokemon p; -- SELECT * FROM pokemon AS p;
SELECT p.nombre Nombre, p.peso Peso FROM pokemon AS p;
SELECT p.nombre "Nombre Pokémon", a.nombre apariencia
FROM pokemon p, apariencia a -- FROM pokemon AS p, apariencia AS a
WHERE p.id_apariencia = a.id_apariencia;
```

2.4 Cláusula WHERE.

Hasta ahora se ha utilizado la sentencia SELECT para obtener todas o un subconjunto de columnas de una o varias tablas. Pero esta selección afecta a todas las filas (registros) de la tabla. Si se quiere **restringir esta selección a un subconjunto de filas** se debe **especificar una condición que deben cumplir aquellos registros que se quieren seleccionar**. Para poder hacer esto se va a utilizar la cláusula WHERE.

A continuación de la palabra WHERE será donde se ponga la o las condiciones que han de cumplir las filas (where_condition) para salir como resultado de dicha consulta. 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 tenga el valor NULL).

El **criterio de búsqueda** o condición puede ser más o menos sencillo y para crearlo se pueden conjugar operadores de diversos tipos, funciones o expresiones más o menos complejas.

Se pueden diferenciar **cinco tipos de condiciones**:

- ✓ 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 se pueden usar en las condiciones pueden ser aritméticos, de comparación, lógicos, etc.

Si de la tabla *pokemon*, se necesita un listado de los Pokémon que sean machos únicamente, bastaría con crear la siguiente consulta:

```
SELECT nombre, sexo FROM pokemon WHERE sexo = 'M';
```

Si se necesita un listado de los Pokémon que sean machos (pero que puedan ser también hembras, se debe tener en cuenta que el campo sexo es de tipo SET y puede tomar varios valores), la consulta a utilizar sería cualquiera de las siguientes:

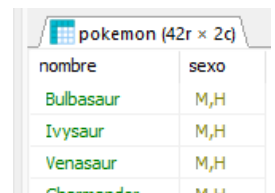
```
SELECT nombre, sexo FROM pokemon WHERE sexo LIKE '%M%';
SELECT nombre, sexo FROM pokemon WHERE FIND_IN_SET('M', sexo);
```

Para la búsqueda de valores en campos SET se puede utilizar el operador LIKE (se verá más adelante en esta unidad) o la función FIND_IN_SET (https://mariadb.com/kb/en/find_in_set/).

La función FIND_IN_SET devuelve 0 si el primer argumento (cadena) no se encuentra en el campo SET indicado como segundo argumento (lista de cadenas), en otro caso devuelve la posición del índice donde aparece. Esta función no funciona correctamente si el primer argumento contiene una coma.



nombre	sexo
Nidoran♂	M
Nidorino	M
Nidoking	M



nombre	sexo
Bulbasaur	M,H
Ivysaur	M,H
Venasaur	M,H
Charmander	M,H

2.5 Cláusula GROUP BY y HAVING.

La cláusula **GROUP BY** se usa para agrupar filas que tienen los mismos valores en una o más columnas.

GROUP BY se suele usar junto a **HAVING** cuando se necesita que los resultados agrupados deban cumplir ciertas condiciones, por lo que es **esencial cuando se necesita aplicar condiciones a los resultados de agregaciones**, como SUM, COUNT, AVG, etc., **después de agrupar los datos**. Mientras la cláusula WHERE se utiliza para filtrar filas individuales antes de que se formen grupos, la cláusula HAVING se utiliza para filtrar grupos de filas después de que se han formado.

Sobre la tabla Pokémon, se podría plantear la pregunta ¿cuántos Pokémon hay?, y la respuesta es un solo dato que hace referencia a un solo grupo. Ahora se podrían dividir los Pokémon en varios grupos de manera que uno lo formen los machos, otro las hembras y otro los que puedan ser machos/hembras. Una vez formados los grupos se podría preguntar ¿cuántos Pokémon hay en cada grupo?, y la respuesta es una cantidad para cada grupo (el resultado es en realidad una totalización por grupos). Primero se deben establecer los grupos por sexo y luego totalizar la cantidad de Pokémon de cada uno. Pues bien, **SQL permite agrupar totales mediante la cláusula GROUP BY**.

Ejemplo:

```
SELECT sexo, COUNT(*) cantidad
FROM pokemon
GROUP BY sexo;
```

sexo	cantidad
M	3
H	3
M,H	39

COUNT es una función de agregado, las **funciones de agregado se usan dentro de la cláusula SELECT para devolver un único valor que se aplica a un grupo de registros**. También se utilizan para obtener resultados estadísticos sobre las columnas de una tabla o sobre la misma tabla; en este caso, la consulta devuelve un solo registro con los resultados. Estas funciones se verán en los puntos 6 y 7.

Nota: las cláusulas **GROUP BY** y **HAVING** se ejecutan antes que la cláusula **SELECT** y por tanto no se deberían poder utilizar en ellas alias definidos en la cláusula **SELECT**, pero sin embargo **MariaDB/MySQL lo permiten** al implementar una extensión del **SQL** estándar que lo acepta.

La siguiente consulta funciona, aunque las cláusulas **GROUP BY** y **HAVING** se ejecuten antes de la cláusula **SELECT** y los alias aún no se hayan definido:

```
SELECT tipo AS tp, COUNT(*) AS total
FROM objeto
GROUP BY tp
HAVING total >= 3;
```

tp	total
Curación	7
Combate	6

2.6 Cláusula ORDER BY.

En las consultas de los ejemplos del punto 2.4 se ha obtenido una lista de nombres y sexo de los Pokémon que son machos. Sería conveniente que aparecieran ordenados por su nombre, ya que siempre quedará más profesional y será más práctico. De este modo, si se necesita localizar un registro concreto la búsqueda sería más rápida. ¿Cómo se hace? Para ello se usará la cláusula **ORDER BY**.

ORDER BY se utiliza **para especificar el criterio de ordenación de la respuesta a una consulta**. Se tendría:

```
SELECT [ALL | DISTINCT] columna1, columna2, ...
FROM tabla1, tabla2, ...
WHERE condición1 AND|OR condición2 AND|OR ...
ORDER BY columna1 [ASC | DESC], ..., columnaN [ASC | DESC];
```

Después de cada columna de ordenación se puede incluir el tipo de ordenación (ascendente o descendente) utilizando las palabras reservadas **ASC** o **DESC**. **Por defecto, y si no se pone nada, la ordenación es ascendente.**

Se debe saber que **es posible ordenar por más de una columna** (no es obligatorio que se incluyan en la cláusula **SELECT**). Es más, **se puede ordenar no solo por columnas, sino a través de una expresión creada con columnas, una constante** (no tendría mucho sentido) **o funciones SQL**. En el siguiente ejemplo, se ordena (de forma ascendente) por nombre y por sexo:

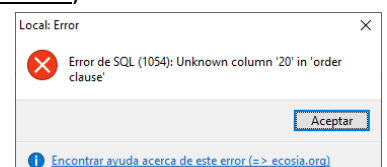
```
SELECT nombre, sexo FROM pokemon ORDER BY nombre, sexo;
```

nombre	sexo
Arbok	M,H
Beedrill	M,H
Blastoise	M,H
Bulbasaur	M,H
Rattata	M,H

Además, **se puede colocar la posición del campo por el que se quiere que se ordene en lugar de su nombre**, es decir, se pueden referenciar los campos por su número de orden en la lista de selección. Por ejemplo, si se quiere ordenar por los campos *evoluciona_de*, *peso* y *sexo* de forma ascendente se utilizaría la sentencia (los **valores NULL** ocuparán las primeras posiciones al ser por defecto ascendente, las últimas si se utiliza **DESC**):

```
SELECT nombre, evoluciona_de, sexo, peso, altura, ps
FROM pokemon ORDER BY 2, 4, 3;
```

Si se coloca un número mayor a la cantidad de campos de la lista de selección, aparece un mensaje de error y la sentencia no se ejecuta.



¿Se puede utilizar cualquier tipo de datos para ordenar? **No todos los tipos de campos servirán para ordenar, únicamente aquellos de tipo carácter, número o fecha.**

Ejemplos:

```
SELECT nombre, apellido1 FROM entrenador ORDER BY sueldo DESC, apellido1 ASC;
SELECT nombre, apellido1, apellido2 FROM entrenador ORDER BY 2 ASC, sueldo DESC;
SELECT nombre FROM pokemon WHERE sexo LIKE '%M%' ORDER BY ps DESC, nombre;
```

Cuando se utiliza la cláusula ORDER BY en la sentencia SELECT, los valores NULL aparecerán en primer lugar si se emplea el modo de ordenación ascendente (ASC) y al final si se usa el descendente (DESC).

2.7 Cláusula LIMIT.

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

La cláusula LIMIT se usa **para limitar los registros que se retornan en una consulta SELECT.**

Recibe 1 o 2 argumentos numéricos enteros positivos; **el primero indica el número del primer registro a retornar** (es decir, el número de filas a saltar antes de empezar a contar, la primera fila que se obtiene como resultado es la que está situada en la posición offset + 1), **el segundo, el número máximo de registros a retornar.** El número del registro inicial es 0 (no 1).

Si el segundo argumento supera la cantidad de registros de la tabla, se limita hasta el último registro.

Ejemplos:

```
SELECT * FROM pokemon LIMIT 0, 4; -- Muestra los primeros 4 registros
SELECT * FROM pokemon LIMIT 4 OFFSET 5; -- Recupera 4 registros a partir del sexto
SELECT * FROM pokemon LIMIT 8; -- Muestra los primeros 8 registros
SELECT * FROM pokemon LIMIT 6, 10000; -- Recupera 10000 registros a partir del séptimo
```

Más información sobre la sentencia SELECT: <https://mariadb.com/kb/en/select/>

2.8 Comentarios en sentencias SQL.

Se pueden escribir **comentarios en las sentencias SQL** de diferentes **formas**:

```
-- Esto es un comentario
SELECT nombre, peso -- Esto es otro comentario
FROM pokemon
ORDER BY nombre DESC;

/* Esto es un comentario
de varias líneas */
SELECT nombre, ps /* Esto es otro comentario */
FROM pokemon
LIMIT 3,4;

# Esto es un comentario
SELECT nombre, sexo, ps # Esto es otro comentario
FROM pokemon;
```

Además, **MariaDB** permite utilizar **comentarios ejecutables**, la sintaxis es la siguiente:

```
/*M! Código específico de MariaDB */          /*! Código específico de MariaDB o MySQL */
```

En caso de que el código deba ejecutarse solo a partir de una versión específica de **MariaDB**:

```
/*M![#]##### Código específico de MariaDB */          /*![#]##### Código específico de MariaDB o MySQL */
```

Los números, representados por **[#]#####** especifican las **versiones mínimas específicas de MariaDB que deben ejecutar el comentario**. El primer número es la versión principal ([#]#), los siguientes 2 números son la versión secundaria y los últimos 2 indican el nivel de parche.

Ejemplos:

```
SELECT 2 /* +1 */; -- Devuelve 2
SELECT 1 /*! +1 */; -- Devuelve 2, el ; se debe poner fuera del comentario ejecutable
SELECT 2 /*M! +1 */; -- Devuelve 3 si se ejecuta en MariaDB
/*M!100617 SELECT 1 */; -- Devuelve 1, se ejecuta a partir de la versión 10.6.17 de MariaDB
```



TAREA

Utilizando las tablas y datos de la BD “conferencias”:

1. Crear una única sentencia de consulta que calcule la suma, resta, producto y división de los números 10 y 5 respectivamente. Asignar un alias a cada una de las operaciones.
2. Realizar una consulta que muestre el nombre y los apellidos de los asistentes cuya empresa sea “BK Programación”.
3. Realizar una consulta que muestre el nombre y capacidad de las salas, ordenada de mayor a menor capacidad.
4. Realizar una consulta donde se obtenga toda la información de los asistentes de la empresa “BigSoft” ordenados por la fecha de nacimiento de forma descendente.
5. Realizar una consulta que muestre el total de asistentes por empresa (ver ejemplo punto 2.5).
6. Realizar una consulta que muestre el nombre y los apellidos de los primeros cinco asistentes, el resultado debe mostrarse ordenado por el primer apellido de forma ascendente.

3 Operadores.

Se ha visto que en la cláusula WHERE se pueden incluir expresiones para filtrar el conjunto de datos que se quiere obtener. Para **crear** esas expresiones se necesitan distintos operadores de modo que se pueda comparar, utilizar la lógica o elegir en función de una suma, resta, etc. Notar que los distintos operadores pueden usarse en diferentes sentencias y cláusulas de SQL y no únicamente en la cláusula WHERE de la sentencia SELECT.

MariaDB utiliza distintos tipos de operadores que se pueden clasificar en: relacionales o de comparación, aritméticos, de asignación, de bits, de concatenación y lógicos.

¿Cómo se utilizan y para qué sirven? En los siguientes apartados se responderá a estas cuestiones y se verán algunos de los más utilizados (tener en cuenta que **MariaDB** proporciona muchos otros).

3.1 Operadores de comparación.

Permiten comparar expresiones, que pueden ser valores concretos de campos, variables, etc.

Los operadores de comparación son símbolos que se usan como su nombre indica para comparar dos valores. Si el resultado de la comparación es correcto la expresión considerada es verdadera, en caso contrario es falsa.

En la tabla de la imagen se tienen los operadores de comparación más utilizados y su significado.

El valor NULL significa valor inexistente o desconocido y por tanto es tratado de forma distinta a otros valores.

Si se quiere verificar que un valor es NULL, se debe utilizar el operador IS NULL como se indica en la tabla o IS NOT NULL que devolverá verdadero si el valor del campo de la fila no es nulo.

OPERADOR	SIGNIFICADO
=	Igualdad.
!=, < >	Desigualdad.
<	<
>	Mayor que.
<=	Menor o igual que.
>=	Mayor o igual que.
IN	Igual que cualquiera de los miembros entre paréntesis.
NOT IN	Distinto que cualquiera de los miembros entre paréntesis.
BETWEEN	Entre. Contenido dentro del rango.
NOT BETWEEN	Fuera del rango.
LIKE '_abc%'	Se utiliza sobre todo con textos y permite obtener columnas cuyo valor en un campo cumpla una condición textual. Utiliza una cadena que puede contener los símbolos "%" que sustituye a un conjunto de caracteres o "_" que sustituye a un carácter.
IS NULL	Devuelve verdadero si el valor del campo de la fila que examina es nulo.

Si se quiere obtener el nombre de los Pokémon cuyo nombre comience por “Ch” ordenados alfabéticamente, la consulta sería:

```
SELECT nombre
FROM pokemon
WHERE nombre LIKE 'CH%'
ORDER BY nombre;
```

pokemon (3r x 1c)
nombre
Charizard
Charmander
Charmeleon

Si un campo de texto se intenta comparar con 0 (mediante =, >=, <=, BETWEEN, ...), **MariaDB/MySQL** intentará convertir el campo de texto en un número (salvo que tome el valor NULL) y en caso de no ser un número válido devolverá 0. Esto puede dar lugar a resultados inesperados en las consultas, por lo que es necesario asegurarse de comparar solo campos de cadena con valores de cadena.

Si se quiere obtener el nombre de los Pokémon cuyo nombre termine en “e” y en su segunda posición contenga la letra “a”, la consulta sería:

```
SELECT nombre
FROM pokemon
WHERE nombre LIKE '_a%e';
```

pokemon (3r x 1c)	
nombre	
Wartortle	
Caperpie	
Raticate	

Si se quiere seleccionar toda la información de los objetos que sean de los tipos “Combate” o “Herramienta” se utilizaría cualquiera de las siguientes sentencias SELECT (su uso es indiferente):

```
SELECT *
FROM objeto
WHERE tipo IN ('Combate', 'Herramienta');
```

```
SELECT *
FROM objeto
WHERE tipo = 'Combate' OR tipo = 'Herramienta';
```

Nota: los operadores AND y OR se verán en los siguientes puntos.

Si se quieren seleccionar los objetos que no sean de los tipos “Combate” o “Herramienta” se utilizaría:

```
SELECT *
FROM objeto
WHERE tipo NOT IN ('Combate', 'Herramienta');
```

objeto (9r x 4c)				
id_objeto	nombre	tipo	descripcion	
1	Poción	Curación	Restaura 20 puntos de salud	
2	Superpoción	Curación	Restaura 50 puntos de salud	
3	Revivir	Curación	Revive a un Pokémon debilitado con la ...	
4	Piedra lunar	Evolución	Permite evolucionar ciertos Pokémon	
8	Hiperpoción	Curación	Restaura 200 puntos de salud	
10	Elixir	Curación	Restaura 10 puntos de poder de todos l...	
12	Baya Oran	Curación	Restaura 10 puntos de salud	
13	Bayas	Curación	Restaura una pequeña cantidad de pun...	
15	Piedra trueno	Evolución	Permite evolucionar ciertos Pokémon	

Si se quiere consultar los entrenadores cuyo segundo apellido es NULL se utilizaría:

```
SELECT *
FROM entrenador
WHERE apellido2 IS NULL;
```

entrenador (10r x 9c)								
id_entrenador	nombre	apellido1	apellido2	fecha_nacimiento	sexo	ciudad	experiencia	suelo
1	Ash	Ketchum	(NULL)	1990-05-22	Masculino	Pueblo Paleta	100	5.000,0
2	Misty	Waterflower	(NULL)	1992-02-14	Femenino	Ciudad Celeste	80	4.000,5
3	Brock	Pewter	(NULL)	1988-11-27	Masculino	Ciudad Plateada	90	4.500,75
4	May	Maple	(NULL)	1994-04-03	Femenino	Ciudad Petalia	70	3.500,25
5	Max	Maple	(NULL)	2000-09-19	Masculino	Ciudad Petalia	50	3.000,0
6	Gary	Oak	(NULL)	1991-07-15	Masculino	Pueblo Paleta	110	5.500,0
8	Clemont	Forge	(NULL)	1990-03-08	No binario	Ciudad Luminaria	95	4.700,5
10	Iris	Dento	(NULL)	1995-09-02	Femenino	Ciudad Caolin	60	3.200,0
11	Cilan	Dent	(NULL)	1989-12-05	Masculino	Ciudad Caolin	70	3.600,5
14	Lt. Surge	Vermilion	(NULL)	1987-08-10	No binario	Ciudad Carmin	100	4.800,5

Si se quiere consultar el nombre de los Pokémon cuyo peso se encuentra comprendido entre 31 y 45 kg. (ambos valores incluidos) se podría utilizar cualquiera de las siguientes consultas:

```
SELECT nombre
FROM pokemon
WHERE peso BETWEEN 31 AND 45;
```

```
SELECT nombre
FROM pokemon
WHERE peso >= 31 AND peso <= 45;
```

Si se quiere consultar el nombre de los Pokémon cuya altura es inferior a 0.4 o superior a 2 metros se podría utilizar cualquiera de las siguientes consultas:

```
SELECT nombre
FROM pokemon
WHERE altura NOT BETWEEN 0.4 AND 2;
```

```
SELECT nombre
FROM pokemon
WHERE altura < 0.4 OR altura > 2;
```

En caso de tener que utilizar en el operador LIKE los símbolos % y _ se ha de utilizar algún carácter de escape. La sintaxis en *MariaDB* del operador LIKE es como sigue:

```
expr LIKE pat [ESCAPE 'escape_char']
expr NOT LIKE pat [ESCAPE 'escape_char']
```

La palabra clave **ESCAPE** se usa para escapar los caracteres de coincidencia de patrones, como el porcentaje % y el guion bajo _ si forman parte de los datos. Ej.:

```
SELECT * FROM entrenador WHERE apellido1 LIKE 'P!%' ESCAPE '!';
SELECT * FROM entrenador WHERE apellido1 LIKE 'P\%';
```

Ambas sentencias producen el mismo resultado. Si no se especifica un carácter de escape, *MariaDB* asume que \ es el carácter de escape.

3.2 Operadores aritméticos.

Los **operadores aritméticos** permiten realizar cálculos con valores numéricos (ver tabla).

Utilizando expresiones con operadores aritméticos es posible obtener salidas en las cuales una columna sea el resultado de un cálculo y no un campo de una tabla.

En el siguiente ejemplo, se van a obtener los entrenadores cuyo sueldo sea mayor o igual a 4500 aumentado en un 5%:

```
SELECT nombre, apellido1, apellido2, sueldo * 1.05 AS suel
FROM entrenador
WHERE sueldo >= 4500;
```

Si se desea obtener para cada entrenador la relación que existe entre su sueldo y la experiencia que posee, se podría utilizar la sentencia:

```
SELECT nombre, apellido1, apellido2, sueldo / experiencia AS
relacion
FROM entrenador;
```

Cuando una expresión aritmética se calcula sobre valores NULL, el resultado es el propio valor NULL. Ej.:

```
SELECT 5 * NULL, 5 + NULL, NULL MOD 5;
```

Operador	Nombre	Ejemplo
+	Adición	SELECT 3+5; -> 8 SELECT 3.5+2.5; -> 6.0 SELECT 3.5+2; -> 5.5
-	Sustracción	SELECT 3-5; -> -2
*	Multiplicación	SELECT 3 * 5; -> 15
/	División	SELECT 20 / 4; -> 5 SELECT 355 / 113; -> 3.1416 SELECT 10.0 / 0; -> NULL
DIV	División entera	SELECT 5 DIV 2; -> 2
% o MOD	Modulo	SELECT 7 % 3; -> 1 SELECT 15 MOD 4; -> 3 SELECT 15 MOD -4; -> 3 SELECT -15 MOD 4; -> -3 SELECT -15 MOD -4; -> -3 SELECT 3 MOD 2.5; -> 0.5

Resultado #1 (1r x 3c)		
5 * NULL	5 + NULL	NULL MOD 5
(NULL)	(NULL)	(NULL)

Más información sobre los operadores aritméticos: <https://mariadb.com/kb/en/arithmetic-operators/>

3.3 Operadores de concatenación.

En SQL para concatenar cadenas de caracteres se pueden utilizar los operadores + (este operador sirve también para sumar valores) y ||, sin embargo en MariaDB/MySQL hay que utilizar las funciones **CONCAT** y **CONCAT_WS**.

Para **concatenar cadenas** de caracteres existe la función **CONCAT(cadena1, cadena2, ...)**. **CONCAT** devuelve NULL si algún argumento es NULL.

Cuando es necesario **indicar un separador** para unir más de 2 cadenas, lo recomendable es usar la función **CONCAT_WS(separador, cadena1, cadena2, ...)**. El primer parámetro es el **separador** y **no puede ser nulo**. **CONCAT_WS** considera las cadenas vacías. Sin embargo, **salta cualquier valor NULL** después del argumento del separador.

MariaDB puede convertir automáticamente valores numéricos a cadenas para su concatenación.

En la tabla *entrenador* se tiene separado en dos campos los apellidos, si se necesita mostrarlos juntos se podría crear la siguiente consulta usando CONCAT:

```
SELECT nombre, CONCAT(apellido1, ' ', apellido2) AS apellidos
FROM entrenador;
```

Pero, rápidamente se puede comprobar que en muchas de las filas en la columna *apellidos* se muestra el valor NULL. Esto es debido a que existen entrenadores cuyo segundo apellido es nulo y al utilizar CONCAT el resultado devuelto será NULL. Para mostrar únicamente el *apellido1* cuando no exista el *apellido2* se puede utilizar la consulta:

```
SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS apellidos
FROM entrenador;
```

entrenador (15r x 2c)	
nombre	apellidos
Ash	(NULL)
Misty	(NULL)
Brock	(NULL)
Mav	(NULL)

entrenador (15r x 2c)	
nombre	apellidos
Ash	Ketchum
Misty	Waterflower
Brock	Pewter

En próximos apartados se verán las funciones NVL, IFNULL, COALESCE e IF que permitirán hacer uso de la función CONCAT aunque se tengan valores nulos en algunos campos.

Nota: si se desea utilizar || como operador de concatenación de cadenas (igual que CONCAT) en lugar de como un sinónimo de OR, se puede configurar el modo SQL PIPES_AS_CONCAT. Ej.:

```
SET @@SQL_MODE = CONCAT(@@SQL_MODE, ',PIPES_AS_CONCAT');
SELECT nombre || ' ' || apellido1 AS nombre_apellido1
FROM entrenador;
```



entrenador (15r × 1c)
nombre_apellido1
Ash Ketchum
Misty Waterflower
Brock Pewter
Mau Maria

3.4 Operadores lógicos.

Habrán ocasiones en las que se tenga que **evaluar más de una expresión y se necesite verificar que se cumple una única condición, otras veces comprobar si se cumple una u otra o ninguna de ellas**. Para poder hacer esto se utilizarán los **operadores lógicos**, los más utilizados son:

OPERADOR	SIGNIFICADO
AND	Devuelve verdadero si sus expresiones a derecha e izquierda son ambas verdaderas.
OR	Devuelve verdadero si alguna de sus expresiones a derecha o izquierda son verdaderas.
NOT	Invierte la lógica de la expresión que le precede, si la expresión es verdadera devuelve falsa y si es falsa devuelve verdadera.

Otro operador lógico menos usado es **XOR (OR exclusivo)**. Devuelve **NULL** si alguno de los operandos es **NULL**. Para operandos que no son **NULL**, se evalúa como **verdadero si un número impar de operandos es distinto de falso**; de lo contrario, se devuelve falso.

El **significado de los operadores** anteriores es:

- ✓ **AND**, significa "y". Los registros recuperados en una sentencia que une dos condiciones con el operador AND, **cumplen con las dos condiciones**.
- ✓ **OR**, significa "y/o". Los registros recuperados con una sentencia que une dos condiciones con el operador OR, **cumplen una de las condiciones o ambas**.
- ✓ **XOR**, significa "o". Los registros recuperados con una sentencia que une dos condiciones con el operador XOR, **cumplen una de las condiciones, no ambas**.
- ✓ **NOT**, significa "no", invierte el resultado. Los registros recuperados en una sentencia en la cual aparece el operador NOT, **no cumplen con la condición a la cual afecta el "NO"**.

CONDICION1	CONDICION2	RESULTADOS		
x	y	x and y	x or y	not x
VERDADERO	VERDADERO	VERDADERO	VERDADERO	FALSO
VERDADERO	FALSO	FALSO	VERDADERO	
VERDADERO	NULL	NULL	VERDADERO	
FALSO	VERDADERO	FALSO	VERDADERO	VERDADERO
FALSO	FALSO	FALSO	FALSO	
FALSO	NULL	FALSO	NULL	
NULL	VERDADERO	NULL	VERDADERO	NULL
NULL	FALSO	FALSO	NULL	
NULL	NULL	NULL	NULL	

Tabla de la verdad de los operadores lógicos.

XOR	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

Nota: también es posible utilizar && en lugar de AND, || en lugar de OR y ! en lugar de NOT.

Más información sobre los operadores lógicos: <https://mariadb.com/kb/en/logical-operators/>

Ejemplos:

- Obtener el nombre y apellidos de los entrenadores/as cuya experiencia esté entre 50 y 80.

```
SELECT nombre, apellido1, apellido2 FROM entrenador WHERE experiencia >= 50 AND experiencia <= 80;
```

- Obtener el nombre y apellidos de los entrenadores/as cuya experiencia esté entre 50 y 80, y su nombre comience por "Ma".

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE experiencia >= 50 AND experiencia <= 80 AND nombre LIKE 'Ma%';
```

- Obtener el nombre y primer apellido de las entrenadoras (sexo femenino) cuyo apellido2 sea nulo.

```
SELECT nombre, apellido1 FROM entrenador
WHERE apellido2 IS NULL AND sexo = 'Femenino';
```


- Obtener el nombre y apellidos de los entrenadores que hayan nacido entre el 01/01/1990 y el 31/12/1995, y cuya experiencia sea igual o mayor a 90.

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE fecha_nacimiento BETWEEN '1990-01-01' AND '1995-12-31' AND experiencia >= 90;
```

- Obtener el nombre de los objetos que no sean de tipo “Curación” y/o comience su nombre por “P”.

```
SELECT nombre FROM objeto WHERE tipo NOT IN ('Curación') OR nombre LIKE 'P%';
```

- Obtener el nombre de los objetos que no sean de tipo “Curación” o comience su nombre por “P” (o una condición o la otra, pero no ambas).

```
SELECT nombre FROM objeto WHERE tipo NOT IN ('Curación') XOR nombre LIKE 'P%';
```

- Recuperar el nombre de los Pokémon que no sean machos o puedan ser machos.

```
SELECT nombre FROM pokemon WHERE NOT (sexo = 'M' OR sexo = 'M,H');
SELECT nombre FROM pokemon WHERE NOT (sexo LIKE '%M%');
```

- Recuperar el nombre de los Pokémon cuyo nombre empiece por “B” o bien por “C”, y que sean machos o puedan ser machos. En este ejemplo se hace necesario el uso de paréntesis, ya que en caso de no utilizarlos el resultado podría ser diferente dado que el operador AND tiene mayor prioridad que el operador OR.

```
SELECT nombre
FROM pokemon
WHERE (nombre LIKE 'B%' OR nombre LIKE 'C%') AND (sexo = 'M' OR sexo = 'M,H');
```

3.5 Precedencia.

Con frecuencia se utilizará la sentencia SELECT acompañada de expresiones extensas y resultará difícil saber que parte de dicha expresión se evaluará primero, por ello es conveniente conocer el **orden de precedencia**:

- | | |
|--------------------------------------------------------------|----------------------------------------------------------|
| 1. Menos unario (-). | 7. NOT. |
| 2. Multiplicación (*), división (/ , DIV) y resto (% , MOD). | 8. AND, &&. |
| 3. Sumas (+) y restas (-). | 9. XOR. |
| 4. Concatenación. | 10. OR, //. |
| 5. Comparaciones (=, <, >, <=, >=, !=, ...). | 11. Asignación (=, :=). Se verá en unidades posteriores. |
| 6. Operadores IS NULL, IS NOT NULL, LIKE y BETWEEN. | |

Para cambiar el orden de precedencia habrá que utilizar paréntesis.

Más información sobre el orden de precedencia de los operadores: <https://mariadb.com/kb/en/operator-precedence/>

4 Consultas calculadas.

En algunas ocasiones será interesante **realizar operaciones con algunos campos para obtener información derivada de éstos**. Si se tuviera un campo *precio*, podría interesar calcular el precio incluyendo el IVA o si se tuvieran los campos *sueldo* y *paga_extra*, se podría necesitar obtener la suma de los dos campos. Estos son ejemplos simples, pero se pueden construir expresiones mucho más complejas. Para ello se hará uso de la creación de campos calculados.

Los operadores aritméticos se pueden utilizar para hacer cálculos en las consultas.

Estos **campos calculados se obtienen a través de la sentencia SELECT poniendo a continuación la expresión que se quiera**. Esta consulta no modificará los valores originales de las columnas ni de la tabla de la que se está obteniendo dicha consulta, únicamente mostrará una columna nueva con los valores calculados. Por ejemplo:

```
SELECT nombre, apellido1, apellido2, sueldo, sueldo + 1000
FROM entrenador;
```

Con esta consulta se mostrará una nueva columna que tendrá como nombre la expresión utilizada. Es posible ponerle un alias a la columna creada añadiéndolo detrás de la expresión un nombre. El ejemplo anterior quedaría así:

```
SELECT nombre, apellido1, apellido2, sueldo, sueldo + 1000 AS "Sueldo nuevo"
FROM entrenador;
```



TAREA

Utilizando las tablas y datos de la BD “conferencias”:

7. Realizar una consulta donde se obtenga el tema y la fecha de las conferencias que tengan turno de tarde y se celebren en las salas “Apolo” o “Zeus”.
8. Realizar una consulta donde se obtengan aquellos asistentes cuyo *apellido1* comience por la letra “M” y contenga la letra “R”.
9. Realizar una consulta que seleccione las conferencias cuyo precio esté comprendido entre 12 y 19 euros, y cuyo tema no sea “Programación Web”.
10. Realizar una consulta que seleccione los asistentes cuyas fechas de nacimiento estén comprendidas entre el 01/01/1974 y el 01/01/1985, ordenando los resultados por la fecha de nacimiento y concatenando las columnas *apellido1* y *apellido2* en una sola a la que se le asigne el alias “Apellidos”.
11. Realizar una consulta que seleccione las conferencias cuyo tema comience por “Programación”, ordenando los resultados por su precio de forma descendente (de mayor a menor precio).
12. Realizar una consulta que calcule los precios de las distintas conferencias teniendo en cuenta que se van a aplicar los siguientes descuentos: 0%, 5%, 10%, 15% y 20%. Asignar un alias a cada una de las nuevas columnas calculadas (“Descuento 0%”, “Descuento 5%”, ... respectivamente).

5 Funciones.

En casi todos los SGBD existen funciones ya creadas que **facilitan la creación de consultas más complejas**, dichas funciones varían según el SGBD, en este punto se verán algunas de ellas. En la unidad 6 (*Programación de bases de datos*) se aprenderá a programar funciones propias (no proporcionadas por el SGBD).

Las **funciones** son realmente **operaciones que se realizan sobre los datos y que realizan un determinado cálculo**. Para ello **necesitan unos datos de entrada llamados parámetros o argumentos** y en función de éstos, se realizará el cálculo de la función que se esté utilizando. Normalmente los parámetros **se especifican entre paréntesis**.

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

Las funciones se especifican de la siguiente manera: nombre_función([parámetro1[, parámetro2] ...])

Se pueden anidar funciones dentro de funciones.

Existe una gran variedad para cada tipo de datos (se verán en los siguientes apartados):

- | | | |
|----------------------------|------------------------|--------------|
| ✓ Numéricas. | ✓ De manejo de fechas. | ✓ De listas. |
| ✓ De cadena de caracteres. | ✓ De conversión. | ✓ Otras. |

5.1 Funciones numéricas.

Para trabajar con campos de tipo numérico se tienen las siguientes funciones:

- ✓ **ABS(n)**: calcula el **valor absoluto** de un número *n*. Ej.: `SELECT ABS(-17);` -- 17
- ✓ **EXP(n)**: calcula *eⁿ*, es decir, el **exponente en base e** del número *n*. Ej.: `SELECT EXP(2);` -- 7.38905609893065
- ✓ **CEIL(n)**: calcula el **valor entero inmediatamente superior o igual** al argumento *n*. Ej.: `SELECT CEIL(2.1);` -- 3
- ✓ **FLOOR(n)**: calcula el **valor entero inmediatamente inferior o igual** al parámetro *n*. Ej.: `SELECT FLOOR(2.1);` -- 2
- ✓ **MOD(m, n)**: calcula el **resto resultante de dividir m entre n**. Ej.: `SELECT MOD(5, 2);` -- 1
- ✓ **POWER(valor, exponente)**: eleva el **valor** al **exponente** indicado. Ej.: `SELECT POWER(5, 2);` -- 25
- ✓ **ROUND(n, decimales)**: redondea el número *n* al siguiente número con el número de decimales que se

indican. Ej.: `SELECT ROUND(12.5874, 2); -- 12.59`

- ✓ `SQRT(n)`: calcula la raíz cuadrada de n . Ej.: `SELECT SQRT(25); -- 5`
- ✓ `TRUNCATE(m, n)`: trunca un número m a la cantidad de decimales n especificada por el segundo argumento. Para truncar todos los decimales se asignará a n el valor 0. Si n es negativo, el número es truncado desde la parte entera. Ej.: `SELECT TRUNCATE(127.4567, 2); -- 127.45`
- ✓ `SIGN(n)`: si n es un valor positivo, retorna 1, si es negativo, devuelve -1 y 0 si es 0. Ej.: `SELECT SIGN(-23); -- -1`
- ✓ `PI()`: devuelve el valor del número π . Ej.: `SELECT PI(); -- 3.141593`

Ejemplos:

- Obtener el nombre y apellidos de los entrenadores/as, junto a su sueldo redondeado al número entero mayor.
`SELECT nombre, apellido1, apellido2, CEIL(sueldo) AS sueldo FROM entrenador;`
- Obtener el nombre y apellidos de los entrenadores/as, junto a su sueldo incrementado en un 5%. Redondear el nuevo sueldo obtenido a dos decimales.
`SELECT nombre, apellido1, apellido2, ROUND(sueldo*1.05, 2) AS sueldo_incrementado FROM entrenador;`
- Obtener el nombre y apellidos de los entrenadores/as, junto a su sueldo redondeado a miles (si el sueldo es 4565 devolverá 5000 y si es 4499 devolverá 4000).
`SELECT nombre, apellido1, apellido2, ROUND(sueldo, -3) AS sueldo_incrementado FROM entrenador;`
- Obtener el índice de masa corporal ($IMC = peso (kg) / [estatura (m)]^2$) de los Pokémon machos (únicamente). Truncar el valor obtenido del IMC a 3 decimales.

```
SELECT nombre, TRUNCATE(peso/POWER(altura, 2), 3) AS IMC
FROM pokemon
WHERE sexo = "M";
```

nombre	IMC
Nidoran♂	36,0
Nidorino	24,074
Nidoking	31,632

Más información sobre las funciones numéricas: <https://mariadb.com/kb/en/numeric-functions/>

5.2 Funciones de cadena de caracteres.

Es muy común manipular campos de tipo carácter o cadena de caracteres, como resultado se podrá obtener caracteres o números. **Las funciones más habituales son:**

- ✓ `CHR(n)`: devuelve el carácter cuyo valor codificado es n . Ej.: `SELECT CHR(81); -- Q`
- ✓ `ASCII(n)`: devuelve el valor *ASCII* de n . Ej.: `SELECT ASCII('Q'); -- 79`
- ✓ `CONCAT(cad1, cad2, ...)`: devuelve las cadenas unidas sin añadir ningún espacio entre ellas. Ej.: `SELECT CONCAT('Hola', 'Mundo'); -- HoLaMundo`
- ✓ `CONCAT_WS(sep, cad1, cad2, ...)`: similar a `CONCAT`, pero permite definir un carácter separador. Ej.: `SELECT CONCAT_WS('+', 'Hola', 'Mundo'); -- HoLa+Mundo`
- ✓ `LOWER(cad)`: devuelve la cadena cad con todos sus caracteres en minúsculas. Ej.: `SELECT LOWER('En Minúsculas'); -- en minúsculas`
- ✓ `UPPER(cad)`: devuelve cad con todos sus caracteres en mayúsculas. Ej.: `SELECT UPPER('En Mayúsculas'); -- EN MAYÚSCULAS`
- ✓ `LPAD(cad1, n, cad2)`: devuelve $cad1$ con longitud n , ajustada a la derecha, rellenando por la izquierda con $cad2$. Si la longitud de $cad1$ es mayor que n la recorta. Ej.: `SELECT LPAD('M', 5, '*'); -- *****M`
- ✓ `RPAD(cad1, n, cad2)`: devuelve $cad1$ con longitud n , ajustada a la izquierda, rellenando por la derecha con $cad2$. Si la longitud de $cad1$ es mayor que n la recorta. Ej.: `SELECT RPAD('M', 5, '*'); -- M*****`
- ✓ `REPLACE(cad, ant, nue)`: devuelve cad en la que cada ocurrencia de la cadena ant ha sido sustituida por la cadena nue . Ej.: `SELECT REPLACE('correo@gmail.es', 'es', 'com'); -- correo@gmail.com`
- ✓ `SUBSTR(cad, m, n)`: devuelve la cadena cad compuesta por n caracteres a partir de la posición m . Ej.: `SELECT SUBSTR('1234567', 3, 2); -- 34`
- ✓ `LENGTH(cad)`, `CHAR_LENGTH(cad)`: devuelve la longitud de cad . Ej.: `SELECT LENGTH('día'), CHAR_LENGTH('día');` -- `LENGTH` devuelve el resultado en bytes (4) y `CHAR_LENGTH` en caracteres (3)
- ✓ `TRIM(cad)`: elimina los espacios en blanco a la izquierda/derecha de cad . Ej.: `SELECT TRIM(' Hola de nuevo mundo '); -- Hola de nuevo mundo`
- ✓ `LTRIM(cad)`: elimina los espacios a la izquierda que posea cad . Ej.: `SELECT LTRIM(' Hola');` -- `Hola`
- ✓ `RTRIM(cad)`: elimina los espacios a la derecha que posea cad . Ej.: `SELECT RTRIM('Hola '); -- Hola`

- ✓ **STRCMP(cad1, cad2)**: permite **comparar cadenas de caracteres**. Devuelve **0** si las cadenas son iguales, **-1** si el primer argumento es más pequeño que el segundo según el orden de clasificación actual y **1** si las cadenas no son iguales. Devuelve **NULL** si cualquiera de los dos argumentos es **NULL**. Ej.: `SELECT STRCMP('texto', 'texto2'); -- -1`
- ✓ **INSTR(cad, cadBuscada)**: **obtiene la primera posición en la que se encuentra la cadena buscada en la cadena inicial cad**. Si no encuentra nada devuelve cero. Ej.: `SELECT INSTR('usuarios', 's'); -- 2`
- ✓ **LOCATE(cadBuscada, cad [, pos])**: similar a la anterior, invirtiendo el orden de los parámetros y pudiendo **especificar una posición de comienzo en la búsqueda**. Devuelve 0 si la cadena buscada no se encuentra. Ej.: `SELECT LOCATE('s', 'usuarios', 3); -- 8`

La función **CONCAT** devolverá **NULL** cuando alguna de las cadenas que se está concatenando sea igual a **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.

Ejemplos:

- Obtener el nombre y los apellidos concatenados de los entrenadores/as. Mostrar tanto el nombre como los apellidos en mayúsculas. En la consulta se debe mostrar la longitud en caracteres del nombre y los apellidos. En caso de que el *apellido2* sea nulo, únicamente se debe mostrar el *apellido1*.

```
SELECT UPPER(nombre) AS nombre, UPPER(CONCAT_WS(' ', apellido1, apellido2)) AS apellidos,
       CHAR_LENGTH(nombre) + CHAR_LENGTH(UPPER(CONCAT_WS(' ', apellido1, apellido2))) AS longitud
FROM entrenador;
```

nombre	apellidos	longitud
ASH	KETCHUM	10
MISTY	WATERFLOWER	16
BROCK	PEWTER	11
MAY	MAPLE	8
MAY	MARPLE	8

- Obtener el *nombre* y *apellido1* de los entrenadores/as. Mostrar tanto el nombre como los apellidos en minúsculas. En caso de que el *nombre* o el *apellido1* no llegue a 15 caracteres, completar por la derecha ambos campos con el carácter #.

```
SELECT RPAD(LOWER(nombre), 15, '#') AS nombre, RPAD(LOWER(apellido1), 15, '#') AS apellido1
FROM entrenador;
```

nombre	apellido1
ash#####	ketchum#####
misty#####	waterflower####
brock#####	pewter#####
mav#####	marple#####

- Obtener el *nombre* de los objetos de tipo combate cuyo nombre contiene la cadena “yo”. Reemplazar en el campo nombre la cadena “MT0” por “***”.

```
SELECT REPLACE(nombre, 'MT0', '***') AS nombre
FROM objeto
WHERE tipo = 'Combate' AND nombre LIKE '%yo%';
```

nombre
***3 - Rayo hielo
***5 - Rayo solar

Más información sobre las funciones de cadenas de caracteres: <https://mariadb.com/kb/en/string-functions/>

5.3 Funciones de manejo de fechas y/o horas.

En las **BD** se utilizan mucho las fechas y las horas, **MariaDB** proporciona **cuatro tipos de datos**:

- ✓ **DATE**: almacena **fechas concretas sin tener en cuenta la hora** en formato **'YYYY-MM-DD'**, el rango soportado va desde el **'1000-01-01'** al **'9999-12-31'** (3 bytes).
- ✓ **DATETIME**: considera valores de **fecha y hora** en formato **'YYYY-MM-DD HH:MM:SS'**, el rango soportado va desde el **'1000-01-01 00:00:00'** al **'9999-12-31 23:59:59'** (8 bytes).
- ✓ **TIMESTAMP**: considera valores **fecha y hora**, pero tiene un rango que va desde el **'1970-01-01 00:00:01' UTC** (*Universal Time Coordinated*) al **'2038-01-19 03:14:07' UTC** (4 bytes).
- ✓ **TIME**: considera valores de **hora** en formato **'HH:MM:SS'**, el rango soportado va desde las **'-838:59:59'** a las **'838:59:59'** (3 bytes).
- ✓ **YEAR**: almacena los años en formato **'YYYY'**, el rango soportado va desde el **'1901'** al **'2155'** (1 byte).

Se pueden realizar operaciones numéricas con las fechas:

- ✓ Se le pueden **sumar números** y esto se entiende como **sumarles días**, si el número tiene decimales se suman días, horas, minutos y segundos.
- ✓ La **diferencia entre dos fechas** dará un número de días.

Las **funciones más comunes** para trabajar con fechas en *MariaDB* son:

- ✓ **SYSDATE([precisión]):** devuelve la **fecha y hora actuales**. El parámetro **precisión** determina la **precisión en microsegundos** (máximo valor permitido 6). Ej.: `SELECT SYSDATE(4);` -- 2023-07-03 15:33:19.4042
- ✓ **NOW([precisión]):** devuelve la **fecha y hora actuales**. Ej.: `SELECT NOW();` -- 2023-07-03 15:34:54
- ✓ **CURDATE():** devuelve la **fecha actual** como un valor en formato 'AAAA-MM-DD' o 'AAAAMMDD', dependiendo de si la función se usa en una cadena o en un contexto numérico. Ej.: `SELECT CURDATE();` -- 2023-07-03
- ✓ **CURTIME([precisión]):** devuelve la **hora actual** como un valor en formato 'HH:MM:SS' o 'HHMMSS.uuuuuu', dependiendo de si la función se usa en una cadena o en un contexto numérico. La precisión es opcional y determina la precisión de microsegundos. Ej.: `SELECT CURTIME(4);` -- 15:36:24.4200
- ✓ **DATE(expr):** extrae la **parte de fecha de la expresión de fecha o fecha/hora expr**. Ej.: `SELECT DATE('2023-07-18 12:21:32');` -- 2023-07-18
- ✓ **DATEDIFF(expr1, expr2):** devuelve (**expr1 - expr2**) expresado como un valor en días de una fecha a la otra. *expr1* y *expr2* son expresiones de fecha o fecha y hora. En el cálculo solo se utilizan las partes de fecha de los valores. Ej.: `SELECT DATEDIFF('2022-12-31 23:59:59', '2022-12-30');` -- 1
- ✓ **DATE_SUB(fecha, INTERVAL valor unidad):** realiza **operaciones aritméticas con las fechas**. El argumento *fecha* especifica el valor de fecha y hora, el argumento *valor* es una expresión que especifica el valor del intervalo que se sumará o restará de la fecha y el argumento *unidad* indica las unidades en las que debe interpretarse el valor (SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.). Ej.: `SELECT DATE_SUB(NOW(), INTERVAL 1 YEAR);` -- 2022-07-03 16:18:06, suponiendo que hoy es 3 de julio de 2023
- ✓ **DATE_ADD(fecha, INTERVAL valor unidad):** similar a **DATE_SUB**. Si se indica un valor negativo hace lo mismo que si se utiliza **DATE_SUB** con el mismo valor en positivo. `SELECT DATE_ADD(NOW(), INTERVAL 1 YEAR);` -- 2024-07-03 16:18:06, suponiendo que hoy es 3 de julio de 2023
- ✓ **DATE_FORMAT(fecha, formato):** **formatea el valor de la fecha de acuerdo con la cadena de formato**. Para más información sobre los distintos formatos: <https://mariadb.com/kb/en/date-format/>. Ej.: `SELECT DATE_FORMAT(NOW(), '%d/%m/%Y');` -- 03/07/2023, muestra la fecha actual en formato 'DD/MM/YYYY'
- ✓ **STR_TO_DATE(cadena, formato):** es la **inversa de la función DATE_FORMAT(fecha, formato)**. En caso de **no ser correcta la fecha pasada en cadena devuelve NULL** por lo que nos permite comprobar la validez de fechas. Ej.: `SELECT STR_TO_DATE('2023-03-32', '%Y-%m-%d');` -- NULL
- ✓ **DAY(fecha):** devuelve el **día del mes para la fecha** dada en el rango 1 a 31. Ej.: `SELECT DAY(NOW());` -- 3, suponiendo que es el día 3 de julio
- ✓ **DAYOFWEEK(fecha):** devuelve el **índice del día de la semana para la fecha** (1=domingo, 2=lunes, 3=martes, ..., 7=sábado). Ej.: `SELECT DAYOFWEEK(NOW());` -- 2, suponiendo que es Lunes hoy
- ✓ **DAYOFYEAR(fecha):** devuelve el **día del año para la fecha** en el rango de 1 a 366. Ej.: `SELECT DAYOFYEAR(NOW());` -- 184, suponiendo que es 3 de julio de 2023
- ✓ **DAYNAME(fecha):** devuelve el **nombre del día de la semana para la fecha**. El idioma utilizado para el nombre está controlado por el valor de la variable de sistema *lc_time_names*. Ej.: `SELECT DAYNAME(NOW());` -- Monday, suponiendo que es 3 de julio de 2023
- ✓ **FROM_DAYS(n):** **dado un número de día n, devuelve un valor de fecha**. El recuento de días se basa en el número de días desde el inicio del calendario estándar (0000-00-00). Ej.: `SELECT FROM_DAYS(900000);` -- 2464-02-12
- ✓ **MONTH(fecha):** devuelve el **mes de la fecha dada** en el rango de 1 a 12 (de enero a diciembre). Ej.: `SELECT MONTH(NOW());` -- 7, suponiendo que es 3 de julio de 2023
- ✓ **MONTHNAME(fecha):** devuelve el **nombre completo del mes para la fecha**. El idioma utilizado para el nombre está controlado por el valor de la variable de sistema *lc_time_names*. Ej.: `SELECT MONTHNAME(NOW());` -- July, suponiendo que es 3 de julio de 2023
- ✓ **QUARTER(fecha):** devuelve el **trimestre del año para la fecha dada** en el rango de 1 a 4. Ej.: `SELECT QUARTER(NOW());` -- 3, suponiendo que es 3 de julio de 2023
- ✓ **WEEKOFYEAR(fecha):** devuelve el **número de semana para la fecha dada**. Ej.: `SELECT WEEKOFYEAR(NOW());` -- 27, suponiendo que es 3 de julio de 2023
- ✓ **YEAR(fecha):** devuelve el **año para la fecha dada** en el rango de 1000 a 9999. Ej.: `SELECT YEAR(NOW());` -- 2023, suponiendo que es 3 de julio de 2023

Se pueden emplear estas funciones enviando como argumento el **nombre de un campo de tipo fecha**. Por ejemplo, si se quiere ver el campo *fecha_nacimiento* de la tabla *entrenador* en el formato 'DD-MM-YYYY' se puede utilizar una de las sentencias siguientes:

```
SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS
      apellidos, sexo, DATE_FORMAT(fecha_nacimiento, '%d/%m/%Y') AS fecha_nac
```

entrenador (15r x 4c)			
nombre	apellidos	sexo	fecha_nac
Giovanni	Viridian Viridian	Masculino	02/11/1985
Lt. Surge	Vermilion	No binario	10/08/1987
Brock	Pewter	Masculino	27/11/1988
Cilan	Dent	Masculino	05/12/1989


```

FROM entrenador
ORDER BY fecha_nacimiento;

SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS apellidos, sexo,
      CONCAT(LPAD(DAY(fecha_nacimiento), 2, '0'), '/', LPAD(MONTH(fecha_nacimiento), 2, '0'),
            '/', YEAR(fecha_nacimiento)) AS fecha_nac
FROM entrenador
ORDER BY fecha_nacimiento;

```

Importante: tener en cuenta que para que los nombres de los meses y las abreviaciones aparezcan en español se deberá configurar la variable global `lc_time_names`. Esta variable afecta al resultado de las funciones `DATE_FORMAT`, `DAYNAME` y `MONTHNAME`.

Para configurar la variable global a español ejecutar lo siguiente:

```
SET @@lc_time_names = "es_ES"; -- SET GLOBAL lc_time_names = "es_ES";
```

Una vez establecido el valor de la variable, se podrá consultar ejecutando: `SELECT @@lc_time_names;`

Si se ejecuta la sentencia `SELECT DAYNAME(NOW());` devolverá el nombre del día en el que se esté en español. La sentencia `SELECT MONTHNAME(NOW());` devolverá el nombre del mes en el que se esté.

Ejemplos:

- Obtener el nombre y los apellidos concatenados de los entrenadores/as que tienen más de 35 años.

```

SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS apellidos
FROM entrenador
WHERE NOW() >= DATE_SUB(fecha_nacimiento, INTERVAL -35 YEAR);
-- WHERE NOW() >= DATE_ADD(fecha_nacimiento, INTERVAL 35 YEAR);

```

- Obtener el nombre, los apellidos concatenados y la fecha de nacimiento de los entrenadores/as que han nacido en el mes de noviembre. La fecha de nacimiento se debe mostrar en el formato 'DD-MM-YYYY'.

```

SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS apellidos,
      DATE_FORMAT(fecha_nacimiento, '%d/%m/%Y') AS fecha_nac
FROM entrenador
WHERE MONTH(fecha_nacimiento) = 11;

```

nombre	apellidos	fecha_nac
Brock	Pewter	27/11/1988
Giovanni	Viridian Viridian	02/11/1985

- Obtener el nombre, los apellidos concatenados y la fecha de nacimiento de los entrenadores/as que han nacido en los primeros 15 días de un año par. La fecha de nacimiento se debe mostrar en el formato 'nombre_día, DD de nombre_mes de YYYY'.

```

SET @@lc_time_names = "es_ES";
SELECT nombre, CONCAT_WS(' ', apellido1, apellido2) AS apellidos,
      DATE_FORMAT(fecha_nacimiento, '%W, %e de %M de %Y') AS fecha_nac
FROM entrenador
WHERE DAY(fecha_nacimiento) <= 15 AND MOD(YEAR(fecha_nacimiento), 2) = 0;

```

nombre	apellidos	fecha_nac
Misty	Waterflower	viernes, 14 de febrero de 1992
May	Maple	domingo, 3 de abril de 1994
Serena	Yvonne López	sábado, 12 de octubre de 1996
Clemont	Forge	jueves, 8 de marzo de 1990

Todas las funciones para trabajar con fechas y horas en MariaDB: <https://mariadb.com/kb/en/date-time-functions/>

5.4 Funciones de conversión.

Las funciones que se han visto en los apartados anteriores están asociadas a un tipo de datos en concreto: número, cadena de caracteres, fecha, etc. Existen funciones de conversión o de transformación de tipos que permiten transformar un dato de un tipo a otro.

Es importante destacar que **MariaDB sabe realizar implícitamente algunas conversiones de un tipo a otro**. Por ejemplo, un carácter que representa una cifra puede usarse directamente en una expresión numérica. **No es recomendable basarse en este tipo de funcionalidades ya que, por un lado, no se garantiza su existencia para todos o para algún tipo de datos y, por otro lado, tampoco se garantiza su disponibilidad con el transcurso del tiempo, a medida que vayan saliendo nuevas versiones.**

Existen **dos funciones de conversión**:

- ✓ **CAST**: tiene la siguiente sintaxis **CAST(expr AS tipo)**.
- ✓ **CONVERT**: tiene dos sintaxis posibles **CONVERT(expr, tipo)** y **CONVERT(expr USING transcoding_name)**.

El **tipo** puede ser **uno de los siguientes**:

- ✓ **BINARY**
- ✓ **CHAR**
- ✓ **DATE**
- ✓ **DATETIME**
- ✓ **DECIMAL[(M[,D])]**
- ✓ **DOUBLE**
- ✓ **FLOAT**
- ✓ **INTEGER**
- ✓ **SIGNED [INTEGER]**
- ✓ **TIME**
- ✓ **UNSIGNED [INTEGER]**

Las **funciones CONVERT y CAST** toman un valor de un tipo y producen un valor de otro tipo. Hay que tener en cuenta que en **MariaDB**, **INT** e **INTEGER** son lo mismo.

La principal **diferencia entre CAST y CONVERT** es que **CONVERT(expr, tipo)** es la sintaxis **ODBC**, mientras que **CAST(expr AS tipo)** y **CONVERT(... USING ...)** es la sintaxis **SQL92**.

Ejemplos:

```
SELECT CAST("casa" AS BINARY);
SELECT CAST("235" AS UNSIGNED INTEGER);
SELECT CAST(2342 AS CHAR CHARACTER SET utf8);
```

- Obtener todos los datos de los entrenadores y mostrar el resultado ordenado por la fecha de nacimiento de forma descendente convirtiendo ésta a tipo CHAR.

```
SELECT * FROM entrenador ORDER BY CAST(fecha_nacimiento AS CHAR) DESC;
```

5.5 Funciones de listas.

Las funciones de listas **trabajan sobre un grupo de columnas dentro de una misma fila**. Comparan los valores de cada una de las columnas en el interior de una fila para obtener el mayor o el menor valor de la lista:

- ✓ **GREATEST(valor1, valor2, ...)**: devuelve el **mayor valor de la lista**. Ej.: **SELECT GREATEST(10,2,25,4);** -- 25
- ✓ **LEAST(valor1, valor2, ...)**: devuelve el **menor valor de la lista**. Ej.: **SELECT LEAST(10, 2, 25, 4);** -- 2

Ejemplo: obtener el nombre de los Pokémon hembra (únicamente) junto al mayor y menor valor de sus campos *ps*, *ataque*, *defensa*, *especial* y *velocidad*.

```
SELECT nombre, GREATEST(ps, ataque, defensa, especial, velocidad) AS mayor,
       LEAST(ps, ataque, defensa, especial, velocidad) AS menor
FROM pokemon
WHERE sexo = "H";
```

nombre	mayor	menor
Nidoran♀	55	40
Nidorina	70	55
Nidoqueen	90	76

5.6 Otras funciones: NVL, IFNULL, COALESCE e IF.

Cualquier columna de una tabla puede contener un valor nulo independientemente del tipo de datos que tenga definido. Eso sí, esto no es así en los casos en que se defina esa columna como no nula (NOT NULL), o que sea clave primaria (PRIMARY KEY).

Cualquier operación que se haga con un valor NULL devuelve un NULL. Por ejemplo, si se intenta dividir por NULL, no aparecerá ningún error, sino que como resultado se obtendrá un NULL (no se producirá ningún error como sucede si se intenta dividir por cero). Ej.: **SELECT 5/NULL;** -- Devuelve NULL, no error

También es posible que el resultado de una función dé un valor nulo.

Por tanto, es habitual encontrarse con estos valores y es entonces cuando aparece la necesidad de poder hacer algo con ellos. Las **funciones para trabajar con nulos** permitirán hacer algo en caso de que aparezca un valor nulo.

La función **NVL(valor, expr1)** devuelve **valor** si no es nulo o bien **expr1** si es nulo. Tener en cuenta que **expr1** debe ser del mismo tipo que **valor**. Una función similar a **NVL** es **IFNULL(valor, expr1)**.

La función **COALESCE(valor1 [, valor2, ...])** devuelve el primer valor no NULL de la lista, o NULL si no hay valores no NULL.

Para **evaluar expresiones** se dispone de la función **IF(expr1, expr2, expr3)** que si **expr1** es TRUE

(expr1<>0 y expr1<>NULL) devolverá expr2, de lo contrario, devolverá expr3. IF puede devolver un valor numérico o de cadena, según el contexto en el que se utilice. Ej.: `SELECT IF(STRCMP('prueba', 'prueba1') = 0, 'Iguales', 'Diferentes');` -- Diferentes

Ejemplos:

- Obtener el nombre y los apellidos concatenados haciendo uso de la función CONCAT de los entrenadores/as que tienen más de 35 años. Realizar el tratamiento de valores nulos para que en el caso de tener únicamente el *apellido1* se muestren correctamente la columna apellidos resultante.

```
SELECT nombre, RTRIM(CONCAT(apellido1, ' ', NVL(apellido2, ''))) AS apellidos
FROM entrenador
WHERE NOW() >= DATE_ADD (fecha_nacimiento, INTERVAL 35 YEAR);
```

- Crear una consulta que muestre el listado de Pokémon junto al código del Pokémon desde el que evoluciona. En caso de que el Pokémon no evolucione desde ningún otro debe mostrar el valor “No evoluciona”.

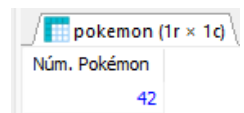
```
SELECT nombre, NVL(evolucion_de, 'No evoluciona') AS evolucion_de
FROM pokemon;
```



nombre	evoluciona_de
Bulbasaur	No evoluciona
Ivysaur	1
Venasaur	2
Charmander	No evoluciona
Charmeleon	4

- Crear una consulta que muestre el número de Pokémon que son o pueden ser machos (dos formas distintas que devuelven el mismo resultado, una de ellas haciendo uso de COUNT y la otra de SUM/IF).

```
SELECT COUNT(*) AS "Núm. Pokémon" FROM pokemon WHERE sexo LIKE '%M%';
SELECT SUM(IF(sexo LIKE '%M%', 1, 0)) AS "Núm. Pokémon" FROM pokemon;
```



Núm. Pokémon
42

TAREA

Utilizando las tablas y datos de la BD “conferencias”:

- Realizar una consulta que aplique un incremento del 5% a los precios de las conferencias, una vez aplicado redondee el precio al número entero menor más cercano y ordene finalmente las filas por el precio obtenido de forma descendente (de mayor a menor). Hacer que la fecha se muestre en formato “DD/MM/YYYY”.
- Realizar una consulta sobre la tabla *ponentes* que muestre todos los campos alfanuméricos en mayúsculas, se deben concatenar los apellidos y mostrar una única columna para ambos. El resultado se debe ordenar por la columna de apellidos resultante de forma ascendente.
- Repetir el ejercicio anterior para que en el caso de que el segundo apellido sea *NULL* muestre la cadena ‘*****’.
- Repetir el ejercicio anterior completando con * los nombres y apellidos que no lleguen a 10 caracteres.
- Realizar una consulta sobre la tabla *asistente* que muestre el nombre y apellidos de los asistentes, en el resultado que se muestre la palabra “José” debe ser sustituida por “Pepe”. Además, para cada fila se debe añadir una nueva columna que muestre la longitud total del nombre y apellidos del asistente sin tener en cuenta los espacios de separación entre el nombre y los apellidos.
- Realizar una consulta sobre la tabla *asistente* que muestre los días que lleva viviendo cada uno de ellos. Además de los días vividos debe mostrar el nombre y los apellidos.
- Realizar una consulta que muestre el nombre y los apellidos junto al día de la semana, el día del año y el número de la semana de la fecha de nacimiento de cada uno de los asistentes. Asignar un alias a cada uno de los campos mostrados.

6 Consultas resumen.

Muchas veces será necesario **realizar cálculos sobre un campo para obtener algún resultado global**, por ejemplo, si se tiene una columna donde se están guardando las notas que obtienen l@s alumn@s en base de datos, se podría estar interesado en saber cuál es la nota máxima que han obtenido o la nota media.

La sentencia **SELECT** va a **permitir obtener resúmenes de los datos de forma vertical**. Para ello dispone de varias **cláusulas específicas (GROUP BY y HAVING)** y también de unas **funciones** llamadas **de agrupamiento** o de agregado que **son las que dirán qué cálculos se quieren realizar sobre los datos** (sobre la columna).

Hasta ahora las consultas que se han visto daban como resultado un subconjunto de filas de la tabla de la que se extraía la información. Sin embargo, **este tipo de consultas no corresponde con ningún valor de la tabla, sino con un valor calculado sobre los datos de la tabla**. Esto hará que las consultas de resumen tengan limitaciones que se irán viendo.

Las **funciones** que se pueden utilizar se llaman **de agrupamiento o agregado**. Éstas **toman un grupo de datos** (una columna) **y producen un único dato que resume el grupo**. Por ejemplo, la función SUM acepta una columna de datos numéricos y devuelve la suma de estos.

El simple **hecho de utilizar una función de agregado en una consulta la convierte en consulta de resumen**. Todas las **funciones de agregado tienen una estructura muy parecida** `NOMBRE_FUNCIÓN([ALL|DISTINCT] expresión)` y se debe **tener en cuenta** que:

- ✓ **ALL** indica que se tienen que **tomar todos los valores de la columna**. Es el **valor por defecto** y no es necesario especificarlo (sería redundante si se pone).
- ✓ **DISTINCT** indica que **se considerarán todas las repeticiones del mismo valor como uno solo** (considera únicamente los valores distintos).
- ✓ El grupo de valores sobre el que actúa la función lo determina el resultado de **expresión** que será el **nombre de una columna o una expresión basada en una o varias columnas**. Por tanto, **en la expresión nunca puede aparecer ni una función de agregado ni una subconsulta**.
- ✓ Todas las funciones se aplican a las filas del origen de datos una vez ejecutada la cláusula WHERE (si se tuviera).
- ✓ Todas las funciones de agregado, excepto `COUNT(*)`, ignoran los valores NULL.
- ✓ Se puede encontrar una función de agrupamiento dentro de una lista de selección en cualquier sitio donde pudiera aparecer el nombre de una columna. Es por eso que puede formar parte de una expresión, pero no se pueden anidar funciones de este tipo.
- ✓ No se pueden mezclar funciones de columna con nombres de columna ordinarios, aunque hay excepciones que se verán más adelante (`GROUP BY`). Ej.: `SELECT nombre, COUNT(*) filas FROM pokemon;` ← FALLA

6.1 Funciones de agregado: SUM y COUNT.

Sumar (SUM) y contar (COUNT) filas o datos contenidos en los campos es algo bastante común.

La función `SUM([DISTINCT] expresión)` devuelve la **suma de los valores de la expresión**. Sólo puede utilizarse **con columnas cuyo tipo de dato sea numérico** (con *MariaDB* si se aplica a un campo de texto puede devolver 0 si no se puede realizar la conversión). El resultado será del mismo tipo, aunque puede tener una precisión mayor.

La función `COUNT([DISTINCT] expresión)` cuenta los elementos de un campo:

- ✓ **expresión contiene el nombre del campo que se desea contar**. Los operandos de expresión pueden incluir el nombre del campo, una constante o una función.
- ✓ **Puede contar cualquier tipo de datos** incluido texto.
- ✓ COUNT simplemente **cuenta el número de registros sin tener en cuenta qué valores se almacenan**.
- ✓ La función COUNT **no cuenta los registros que tienen valores NULL a menos que expresión sea el carácter comodín asterisco (*)**. Ej.: `SELECT COUNT(apellido2), COUNT(*) FROM entrenador;` -- 5 y 15 respectivamente
- ✓ `COUNT(*)` calcula el **total de filas**, incluyendo aquellas que contienen valores NULL.

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 se está usando en la función.

Recordar que **DISTINCT permite seleccionar valores distintos** (elimina las repeticiones).

Ejemplos:

- Obtener el número de entrenadores.

```
SELECT COUNT(*) AS num_entrenadores FROM entrenador; -- 15
```

- Obtener el total de sueldo de todos los entrenadores.

```
SELECT SUM(sueldo) AS sueldo_entrenadores FROM entrenador; -- 64204.75
```

- Obtener el número de apariencias distintas que toman los Pokémon registrados en la BD.

```
SELECT COUNT(DISTINCT id_apariencia) FROM pokemon; -- 25
```

- Obtener el número de objetos que tiene el entrenador 1 en su mochila.

```
SELECT SUM(cantidad) AS "Número de objetos" FROM mochila WHERE id_entrenador = 1; -- 10
```

- Obtener el número de entrenadores cuyo *apellido2* sea nulo (varias formas distintas).

```
SELECT COUNT(*) AS num_entrenadores_apellido2_nulo FROM entrenador WHERE apellido2 IS NULL; -- 10
SELECT COUNT(*) - COUNT(apellido2) AS num_entrenadores_apellido2_nulo FROM entrenador;
SELECT COUNT(IF(apellido2 IS NULL, 1, NULL)) AS num_entrenadores_apellido2_nulo FROM entrenador;
SELECT SUM(IF(apellido2 IS NULL, 1, 0)) AS num_entrenadores_apellido2_nulo FROM entrenador;
```

6.2 Funciones de agregado: MIN y MAX.

Para encontrar el valor máximo (MAX) y mínimo (MIN) se pueden utilizar las siguientes **funciones**:

- ✓ **MIN([DISTINCT] expresión)**: devuelve el **valor mínimo de la expresión sin considerar los nulos (NULL)**.
- ✓ **MAX([DISTINCT] expresión)**: devuelve el **valor máximo de la expresión sin considerar los nulos (NULL)**.

En *expresión* se puede incluir el nombre de un campo de una tabla, una constante o una función, pero no otras funciones agregadas.

Ejemplos:

- Obtener el *apellido1* más bajo (el primero al ordenar de forma ascendente) y más alto (el último al ordenar de forma ascendente) de todos los entrenadores.

```
SELECT MIN(apellido1), MAX(apellido1) FROM entrenador; -- Celadon y Yvonne respectivamente
```

- Obtener el *suelo* más alto de los entrenadores.

```
SELECT MAX(suelo) FROM entrenador; -- 6000.00
```

- Obtener el *peso* del Pokémon que más pesa.

```
SELECT MAX(peso) FROM pokemon; -- 100.00
```

6.3 Funciones de agregado: AVG, VARIANCE y STD.

Para **obtener información estadística** de los datos guardados en una BD se puede hacer uso de las **funciones**:

- ✓ **AVG([DISTINCT] expresión)**: devuelve el **promedio de los valores** de la expresión, para ello **se omiten los valores nulos (NULL)**.
- ✓ **VARIANCE([DISTINCT] expresión)**: devuelve la **varianza estadística** (medida de dispersión que representa la variabilidad de una serie de datos respecto a su media) **de todos los valores** de la *expresión*.
- ✓ **STD([DISTINCT] expresión)**: devuelve la **desviación típica** (es la raíz cuadrada de su varianza) **estadística de todos los valores** de la *expresión*.

Estas funciones **admiten únicamente columnas de tipo numérico** y el **tipo de dato del resultado puede variar según las necesidades** del sistema para representar el valor. Los **valores nulos (NULL)** **se omiten**.

Ejemplos:

- Obtener la media, varianza y desviación típica del peso de los Pokémon.

```
SELECT AVG(peso), VARIANCE(peso), STD(peso) FROM pokemon; -- 23.686667, 619.194933 y 24.883628
```

- Obtener el sueldo medio de los entrenadores.

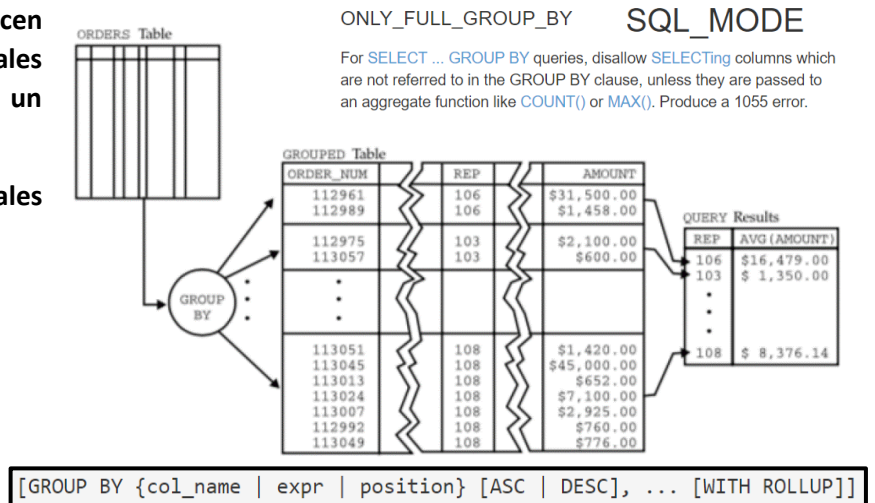
```
SELECT AVG(suelo) FROM entrenador; -- 4280.316667
SELECT SUM(suelo)/COUNT(suelo) FROM entrenador; -- 4280.316667
```

6.4 Agrupamiento de registros.

Hasta aquí las consultas de resumen que se han visto obtienen totales de todas las filas de un campo o una expresión calculada sobre uno o varios campos. Lo que se ha obtenido ha sido una única fila con un único dato.

En muchas ocasiones en las que se utilicen consultas de resumen interesará calcular totales parciales, es decir, agrupados según un determinado campo.

Se pueden obtener estos subtotales utilizando la cláusula **GROUP BY**.



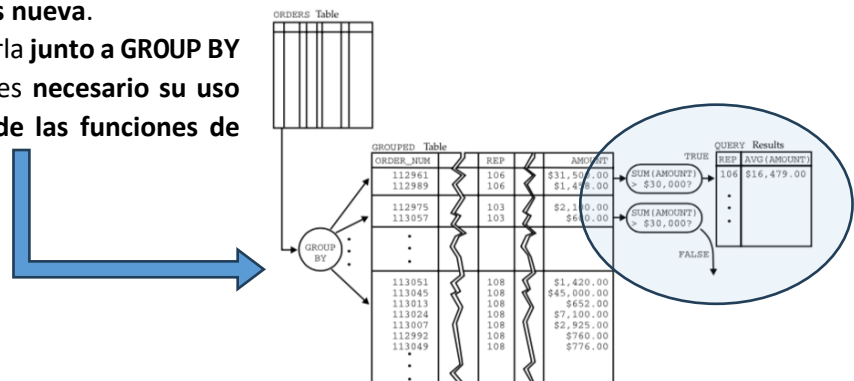
La **sintaxis simplificada es la siguiente:**

```
SELECT col1, [col2, ...,] FUNCIÓN_AGREGADO1(...) [, FUNCIÓN_AGREGADO2(...), ...]
FROM tabla(s)
WHERE condición(es)
GROUP BY col1 [DESC] [, col2 [DESC], ...] [WITH ROLLUP]
HAVING condición(es)
ORDER BY ordenación1[, ordenación2, ...];
```

En la cláusula **GROUP BY** se colocan las columnas por las que se va a agrupar. En la cláusula **HAVING** se especifica la condición que han de cumplir los grupos para que se realice la consulta.

Es muy importante fijarse bien en el **orden en el que se ejecutan las cláusulas:**

1. **WHERE:** que filtra las filas según las condiciones que se pongan (no se pueden usar funciones de agregado, usa los índices mientras que **HAVING** no por lo que es más eficiente al filtrar por valores de campos).
2. **GROUP BY:** que crea una tabla de grupos nueva.
3. **HAVING:** filtra los grupos (hay que utilizarla junto a **GROUP BY** y es menos eficiente que **WHERE**, pero es necesario su uso cuando se quiere filtrar haciendo uso de las funciones de agregado).



Las columnas que aparecen en la cláusula **SELECT** y que no aparezcan en la cláusula **GROUP BY** deben tener una función de agrupamiento. Si esto no se hace así se producirá un error (caso de usar **ONLY_FULL_GROUP_BY**).

WHERE CLAUSE	HAVING CLAUSE
WHERE Clause is used to filter the records from the table based on the specified condition.	HAVING Clause is used to filter record from the groups based on the specified condition.
WHERE Clause can be used without GROUP BY Clause	HAVING Clause cannot be used without GROUP BY Clause
WHERE Clause cannot contain aggregate function	HAVING Clause can contain aggregate function
WHERE Clause can be used with SELECT, UPDATE, DELETE statement.	HAVING Clause can only be used with SELECT statement.
WHERE Clause is used before GROUP BY Clause	HAVING Clause is used after GROUP BY Clause
WHERE Clause is used with single row function like UPPER, LOWER etc.	HAVING Clause is used with multiple row function like SUM, COUNT etc.

De forma predeterminada, si se tiene la cláusula **GROUP BY**, las filas de la salida se ordenarán por los campos utilizados en GROUP BY. También se puede especificar **ASC** o **DESC** después de los campos, como en **ORDER BY**. El valor predeterminado es **ASC**.

Si desea que las filas se ordenen por otro campo, se puede agregar la cláusula **ORDER BY**.

MariaDB/MySQL cuenta con un modificador para las consultas agregadas que puede resultar muy útil en la generación de informes. El **modificador WITH ROLLUP** añade algunas filas adicionales a los resultados generados. Estas filas representan operaciones de agregación de mayor nivel, es decir que **permite agregar los resultados ya agregados** (calcular subtotales adicionales, también conocidos como *rollup*).

Nota: **ONLY_FULL_GROUP_BY** es una configuración de modo (variable del sistema **sql_mode**) que se puede activar en *MariaDB/MySQL*. Esta configuración especifica cómo se deben manejar las consultas que contienen la cláusula **GROUP BY** en relación con las columnas de selección en la consulta.

Cuando **ONLY_FULL_GROUP_BY** está habilitado, todas las columnas que no están en la cláusula **GROUP BY** o en una función de agregación como **SUM**, **COUNT**, **AVG**, etc., no se pueden seleccionar directamente (se generará un error y la consulta no se ejecutará). **Esta configuración asegura que todas las columnas en la cláusula SELECT estén agregadas o incluidas en la cláusula GROUP BY**.

La finalidad de **ONLY_FULL_GROUP_BY** es garantizar que las consultas sean más explícitas y eviten ambigüedades en los resultados. Cuando esta configuración está desactivada, el **SGBD** puede devolver valores arbitrarios para las columnas no agrupadas, lo que puede conducir a resultados inesperados.

Si se intenta ejecutar la siguiente consulta con el modo **ONLY_FULL_GROUP_BY** activado fallará:

```
SELECT sexo, id_apariencia, COUNT(*) AS total_pokemon
FROM pokemon
GROUP BY sexo; -- Se debe agrupar también por la columna id_apariencia
```

Además el resultado arrojado por la consulta anterior carece de sentido.

Ejemplos:

- Obtener el número de Pokémon que se tienen de cada uno de los posibles sexos.

```
SELECT sexo, COUNT(*) AS num_pokemon
FROM pokemon
GROUP BY sexo;
```

sexo	num_pokemon
M	3
H	3
M,H	39

- Obtener el número de Pokémon que se tienen por sexo y apariencia. Ordenar el resultado por el sexo de forma ascendente y seguidamente por el *id_apariencia*.

```
SELECT sexo, id_apariencia, COUNT(*) AS total_pokemon
FROM pokemon
GROUP BY sexo, id_apariencia;
-- ORDER BY sexo, id_apariencia ← No es necesario, GROUP BY ya ordena
```

sexo	id_apariencia	total_pokemon
M	18	2
M	19	1
H	18	2
H	19	1
M,H	1	3
M,H	2	1
M,H	3	2
M,H	4	1

- Obtener el número total de entrenadores por sexo y ciudad.

```
SELECT sexo, ciudad, COUNT(*) AS total_entrenadores
FROM entrenador
GROUP BY ciudad DESC, sexo DESC;
```

sexo	ciudad	total_entrenadores
Masculino	Pueblo Paleta	2
Masculino	Ciudad Verde	1
Masculino	Ciudad Plateada	1
Femenino	Ciudad Pirita	1
Femenino	Ciudad Petalia	1
Masculino	Ciudad Petalia	1
Masculino	Ciudad Luminosa	1

- Repetir el ejemplo anterior añadiendo filas de resumen generales (para la ciudad y el total de entrenadores).

```
SELECT ciudad, sexo, COUNT(*) AS total_entrenadores
FROM entrenador
GROUP BY ciudad DESC, sexo DESC WITH ROLLUP;
```

ciudad	sexo	total_entrenadores
Pueblo Paleta	Masculino	2
Pueblo Paleta	(NULL)	2
Ciudad Verde	Masculino	1
Ciudad Verde	(NULL)	1
Ciudad Plateada	Masculino	1
Ciudad Plateada	(NULL)	1

- Obtener el número de entrenadores por ciudad. Mostrar únicamente aquellas ciudades en las que haya dos o más entrenadores.

```
SELECT ciudad, COUNT(*) AS total_entrenadores
FROM entrenador
GROUP BY ciudad
HAVING COUNT(*) >= 2; -- HAVING total_entrenadores >= 2
```

entrenador (4r × 2c)	
ciudad	total_entrenadores
Ciudad Caolín	2
Ciudad Luminalia	2
Ciudad Petalia	2
Pueblo Paleta	2

- Obtener el sueldo medio de los entrenadores por sexo. Redondear el sueldo medio obtenido a dos decimales. Agregar filas de resumen.

```
SELECT sexo, ROUND(AVG(sueldo),2) AS "Sueldo medio"
FROM entrenador
GROUP BY sexo WITH ROLLUP;
```

entrenador (4r × 2c)	
sexo	Sueldo medio
Masculino	4.600,21
Femenino	3.871,79
No binario	4.750,5
(NULL)	4.280,32

TAREA

Utilizando las tablas y datos de la BD “conferencias”:

- Realizar una consulta donde se obtenga el número total de salas cuya capacidad sea igual o superior a 200.
- Realizar una consulta donde se obtenga la media de las gratificaciones de cada uno de los ponentes.
- Realizar una consulta que muestre el total de salas distintas en función del turno: de mañana (M) o tarde (T).
- Repetir el ejercicio anterior teniendo en cuenta que la sala “Apolo” no debe salir en el resultado final.
- Repetir el ejercicio anterior de forma que se muestre el total del turno de mañana (M).
- Realizar una consulta que muestre el total de asistentes hombres (H) y mujeres (M) por cada empresa.
- Repetir el ejercicio anterior de forma que para los asistentes que tenga como valor de empresa NULL se agrupen en una categoría llamada “Sin empresa asignada”.
- Repetir el ejercicio anterior y hacer que solo se muestren los totales de hombres por cada una de las empresas.

7 Consultas multitable.

Una de las características de las BD relacionales es que se distribuye la información en distintas tablas que a su vez están relacionadas por algún campo común (así se evita repetir datos). Por tanto, también será frecuente que se tengan que consultar datos que se encuentren distribuidos en diferentes tablas.

Las consultas multitable **permiten consultar información en más de una tabla**. La única diferencia respecto a las consultas sobre una tabla es que **se va a tener que especificar en la cláusula FROM cuáles son las tablas que se van a usar y cómo se van a relacionar entre sí** (esto último también se puede hacer en la cláusula WHERE).

Hasta ahora las consultas realizadas hacen uso de una sola tabla, pero también es posible hacer **consultas que usen varias tablas en la misma sentencia SELECT**. Esto permitirá realizar **distintas operaciones** como son:

- ✓ La **composición cruzada** (CROSS JOIN).
- ✓ La **composición interna** (NATURAL JOIN, [INNER] JOIN USING/ON).
- ✓ La **composición externa** (LEFT/RIGHT/FULL [OUTER] JOIN).

En la **versión SQL99** se especifica una **nueva sintaxis para las consultas multitable** que *MariaDB* incorpora. La **razón** de esta nueva sintaxis **es separar las condiciones de asociación respecto a las condiciones de selección de registros**.

Por ejemplo, si se dispone de la tabla *pokemon* cuya clave principal es *id_pokemon* y esta tabla a su vez está relacionada con otra tabla *apariencia* a través del campo *id_apariencia* (clave foránea). Si se quisiera obtener la apariencia de los Pokémon, se necesitaría coger datos de ambas tablas pues el nombre de la apariencia se guarda en la tabla *apariencia* (esto significa que se cogerán filas de una y de otra). La sentencia SELECT podría quedar como sigue:

```
SELECT pokemon.nombre AS pokemon, apariencia.nombre AS apariencia
FROM pokemon JOIN apariencia USING (id_apariencia);
-- FROM pokemon, apariencia WHERE pokemon.id_apariencia = apariencia.id_apariencia;
```

Resultado #1 (45r × 2c)	
pokemon	apariencia
Bulbasaur	Semilla
Ivysaur	Semilla
Venasaur	Semilla
Charmander	flama

La **sintaxis** es la siguiente:

```
SELECT [tabla1.]columna1, [tabla1.]columna2, ..., [tabla2.]columna3, [tabla2.]columna4, ...
FROM tabla1
[CROSS JOIN tabla2] |
[NATURAL JOIN tabla2] |
[[INNER] JOIN tabla2 USING (columna [, ...])] |
[[INNER] JOIN tabla2 ON [(tabla1.columna=tabla2.columna [AND ...] [])] |
[LEFT|RIGHT|FULL [OUTER] JOIN tabla2 USING (columna [, ...])] |
[LEFT|RIGHT|FULL [OUTER] JOIN tabla2 ON [(tabla1.columna=tabla2.columna [AND ...] [])]
...;
```

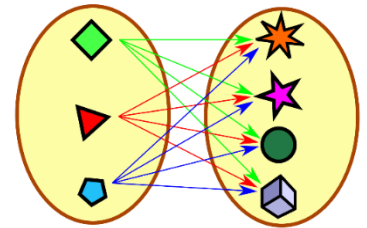
Composición cruzada

Composición interna

Composición externa

7.1 Composiciones cruzadas (producto cartesiano).

El **producto cartesiano** de dos conjuntos, es una **operación** que consiste en obtener otro conjunto cuyos elementos son todas las parejas que pueden formarse entre los dos conjuntos. Por ejemplo, se tendría que coger el primer elemento del primer conjunto y formar una pareja con cada uno de los elementos del segundo conjunto. Una vez hecho esto, repetir el mismo proceso para cada uno de los elementos del primer conjunto.



¿Qué ocurre si se combinan dos o más tablas sin ninguna restricción? El resultado será un **producto cartesiano**. El producto cartesiano entre dos tablas da como resultado todas las combinaciones de todas las filas de esas dos tablas.

Se indica poniendo en la cláusula FROM las tablas que se quieren componer separadas por comas o bien utilizando **CROSS JOIN**. Tener en cuenta que se puede obtener el producto cartesiano de las tablas que se quieren.

Como lo que se obtiene son todas las posibles combinaciones de filas, hay que tener especial cuidado con las tablas que se combinan. Si se tienen dos tablas de 10 filas cada una, el resultado tendrá 100 filas, a medida que aumenta el número de filas que contienen las tablas, mayor será el resultado final, por lo que se puede considerar una operación costosa.

Esta operación no es de las más utilizadas ya que coge una fila de una tabla y la asocia con todas y cada una de las filas de la otra tabla, independientemente de que tengan relación o no. Lo más normal es que se quiera seleccionar los registros según algún criterio.

Se necesitará discriminar de alguna forma para que únicamente aparezcan las filas de una tabla que estén relacionadas con las de otra tabla. A esto se le llama **asociar tablas (JOIN)**.

Ejemplos:

```
SELECT * FROM pokemon, apariencia; -- Devuelve 1260 filas (45 de pokemon x 28 de apariencia)
SELECT * FROM pokemon CROSS JOIN apariencia;
SELECT * FROM pokemon CROSS JOIN entrenador; -- No es necesario que tengan relación
```

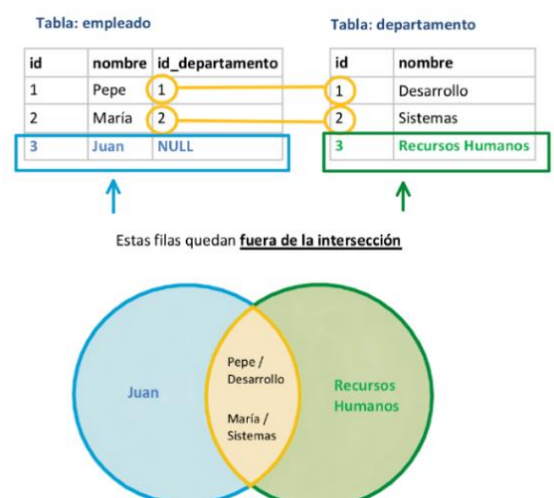
7.2 Composiciones internas (INNER JOIN).

La **intersección** de dos conjuntos es una **operación** que resulta en otro conjunto que contiene sólo los elementos comunes que existen en ambos conjuntos.

Para hacer una **composición interna** se parte de un producto cartesiano y se eliminan aquellas filas que no cumplen la condición de composición. Lo importante en las composiciones internas es emparejar los campos que han de tener valores iguales.

Las **reglas para las composiciones** son:

- ✓ Pueden combinarse tantas tablas como se desee.
- ✓ El criterio de combinación puede estar formado por más de



una pareja de columnas.

- ✓ En la cláusula **SELECT** pueden citarse columnas de todas las tablas, condicionen o no, la combinación.
- ✓ Si hay columnas con el mismo nombre en las distintas tablas, deben identificarse especificando la tabla de procedencia o utilizando un alias de tabla.

Las columnas de emparejamiento no tienen por qué estar incluidas en la lista de selección (cláusula SELECT). Se emparejarán tablas que estén relacionadas entre sí y, además, una de las columnas de emparejamiento será clave principal en su tabla.

Si los campos de unión de las tablas tienen el mismo nombre se puede utilizar **USING** u **ON**, en caso de no tener el mismo nombre hay que utilizar **ON** (ON se puede utilizar en todos los casos). Tener en cuenta que se puede mezclar el uso de **ON** y **USING** en una misma sentencia **SELECT**.

Ejemplo: crear una consulta que devuelva el nombre del Pokémon, su peso y el nombre de su apariencia. La sentencia **SELECT** a ejecutar sería (sintaxis anterior a **SQL99**):

```
SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon, apariencia
WHERE pokemon.id_apariencia = apariencia.id_apariencia;
```

Resultado #1 (45r x 3c)		
pokemon	peso	apariencia
Bulbasaur	6,9	Semilla
Ivysaur	13,0	Semilla
Venasaur	100,0	Semilla
Charmander	8,5	Lagartija
Charmeleon	19,0	Llama
Charizard	90,5	Llama
Squirtle	9,0	Tortuguita
Wartortle	22,5	Tortuna

La anterior consulta es similar a las siguientes (**SQL99**):

```
SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon JOIN apariencia ON pokemon.id_apariencia = apariencia.id_apariencia;

SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon INNER JOIN apariencia ON pokemon.id_apariencia = apariencia.id_apariencia;

SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon JOIN apariencia USING (id_apariencia);

SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon CROSS JOIN apariencia ON pokemon.id_apariencia = apariencia.id_apariencia;

SELECT pokemon.nombre AS pokemon, peso, apariencia.nombre AS apariencia
FROM pokemon JOIN apariencia
WHERE pokemon.id_apariencia = apariencia.id_apariencia;
```

En cualquier caso, la salida sólo contiene las filas que emparejan a los Pokémon con su apariencia. Las filas correspondientes a apariencias que no tienen ningún Pokémon asociado no aparecen. Para las apariencias con varios Pokémon, se repiten los datos de la apariencia para cada Pokémon.

Nota: tener en cuenta que con la operación de intersección sólo se obtendrán los elementos que existan en ambos conjuntos.

Ejemplo: crear una consulta que devuelva el nombre de los Pokémon junto al nombre de sus habilidades (sin repetirse) de aquellos Pokémon que pesen más de 50 kg. y en la descripción de sus habilidades aparezca el texto “ataque”. Ordenar el resultado de forma ascendente por el nombre del Pokémon y seguidamente por el nombre de las habilidades. Se proponen múltiples soluciones combinando diferentes posibles formas de obtener el mismo resultado.

```
SELECT P.nombre AS pokemon, H.nombre AS habilidad
FROM pokemon P, pokemon_habilidad PH, habilidad H
WHERE P.id_pokemon = PH.id_pokemon AND PH.id_habilidad = H.id_habilidad
AND peso >= 50 AND H.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;

SELECT pokemon.nombre AS pokemon, habilidad.nombre AS habilidad
FROM pokemon JOIN pokemon_habilidad USING (id_pokemon) JOIN habilidad USING (id_habilidad)
WHERE peso >= 50 AND habilidad.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;

SELECT P.nombre AS pokemon, H.nombre AS habilidad
FROM pokemon AS P JOIN pokemon_habilidad AS PH ON P.id_pokemon = PH.id_pokemon
JOIN habilidad AS H ON PH.id_habilidad = H.id_habilidad
WHERE peso >= 50 AND H.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;
```

pokemon (4r x 2c)	
pokemon	habilidad
Arbok	Intimidación
Golbat	Fuerza Mental
Nidoking	Punto Tóxico
Nidoqueen	Punto Tóxico


```
SELECT P.nombre AS pokemon, habilidad.nombre AS habilidad
FROM pokemon AS P JOIN pokemon_habilidad AS PH ON P.id_pokemon = PH.id_pokemon
JOIN habilidad USING (id_habilidad)
WHERE peso >= 50 AND habilidad.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;
```

Dado que INNER JOIN es el JOIN de SQL más importante, es posible omitir la palabra clave INNER y utilizar solamente JOIN.

Si dos tablas están enlazadas por columnas con el mismo nombre, los INNER JOIN se aplican como **NATURAL JOIN** y se puede omitir la parte del ON/USING (NATURAL JOIN se usa cuando los campos por los cuales se enlazan las tablas tienen el mismo nombre). Con NATURAL JOIN no se coloca la parte ON/USING que especifica los campos por los cuales se enlazan las tablas, **MariaDB** busca los campos con igual nombre y enlaza las tablas por ese campo. Hay que **tener cuidado** con este tipo de JOIN porque **si ambas tablas tienen más de un campo con igual nombre, MariaDB no sabrá por cual debe realizar la unión.**

Ejemplo: repetición del ejemplo anterior haciendo uso de NATURAL JOIN. Al ejecutar la consulta se puede comprobar que no devuelve ningún resultado, en este caso **MariaDB** enlaza por el campo *nombre* que se repite tanto en la tabla *pokemon* como *habilidad*. Se debe **tener** especial cuidado con este tipo de JOIN para no obtener resultados erróneos.

```
SELECT pokemon.nombre AS pokemon, habilidad.nombre AS habilidad
FROM pokemon NATURAL JOIN pokemon_habilidad NATURAL JOIN habilidad
WHERE peso >= 50 AND habilidad.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;
```



Resultado #1 (0r x 2c)	
pokemon	habilidad



pokemon (45r x 2c)	
nombre	num_habilidades
Arbok	2
Beedrill	1
Blastoise	1
Bulbasaur	1
Butterfree	1
Caperpie	1
Charizard	1
Charmander	1
Charmeleon	1
Clefable	2
Clefairy	2
Cleffa	2

Ejemplo: haciendo uso de NATURAL JOIN obtener un listado con el nombre de los Pokémon junto al número de habilidades que tiene cada uno de ellos. Ordenar el resultado por el nombre de los Pokémon de forma ascendente.

```
SELECT nombre, COUNT(*) AS num_habilidades
FROM pokemon NATURAL JOIN pokemon_habilidad
GROUP BY nombre
ORDER BY nombre;
```

Con **CROSS JOIN** se hace un producto cartesiano (multiplicación de todas las filas de una tabla por todas las filas de la otra tabla) si no se usa la cláusula ON/USING o WHERE para indicar los campos de unión. O sea que si una tabla tiene 30 filas y la otra tabla tiene 50 filas el resultado tendrá $30 * 50 = 1.500$ filas. Si se usa la cláusula WHERE u ON/USING con CROSS JOIN, funciona como INNER JOIN.

Ejemplo: repetición del ejemplo anterior haciendo uso de CROSS JOIN combinado con ON/USING.

```
SELECT pokemon.nombre AS pokemon, habilidad.nombre AS habilidad
FROM pokemon CROSS JOIN pokemon_habilidad USING (id_pokemon)
CROSS JOIN habilidad USING (id_habilidad)
WHERE peso >= 50 AND habilidad.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;

SELECT P.nombre AS pokemon, H.nombre AS habilidad
FROM pokemon AS P CROSS JOIN pokemon_habilidad AS PH ON P.id_pokemon = PH.id_pokemon
CROSS JOIN habilidad AS H ON PH.id_habilidad = H.id_habilidad
WHERE peso >= 50 AND H.descripcion LIKE '%ataque%'
ORDER BY pokemon, habilidad;
```

Normalmente la condición de combinación será una igualdad, pero se puede utilizar cualquier operador de comparación. A la combinación que utiliza comparaciones dentro del predicado JOIN se le llama **theta-join** (se pueden hacer comparaciones utilizando <, <=, =, <>, >= y >). Ej.: **SELECT * FROM tbA JOIN tbB ON tbA.X > tbB.Y;**

Las **composiciones internas son las operaciones que más se emplearán** ya que lo más frecuente es querer juntar los registros de una tabla relacionada con los registros correspondientes en la tabla de referencia (añadir a cada factura los datos de su cliente, añadir a cada línea de pedido los datos de su producto, etc.).

Se puede combinar una tabla consigo misma, pero se debe poner de manera obligatoria un alias a uno de los nombres de la tabla que se va a repetir.

Ejemplo: crear una consulta que devuelva un listado con los nombres de los Pokémon que evolucionan a partir de otros.

```
SELECT p1.nombre AS pokemon, p2.nombre AS evoluciona_de
FROM pokemon AS p1 JOIN pokemon AS p2 ON p1.evoluciona_de = p2.id_pokemon;
```

pokemon	evoluciona_de
Ivysaur	Bulbasaur
Venasaur	Ivysaur
Charmeleon	Charmander
Charizard	Charmeleon
Wartortle	Squirtle
Electroica	Wartortle

Ejemplos:

- Crear una consulta que muestre los nombres de los Pokémon cuya altura sea superior a 1 metro y sean de tipo “Eléctrico” y/o “Fuego”. Proponer dos consultas para obtener el resultado pedido, en una de ellas usar ON para hacer el JOIN de las tablas necesarias y en la otra usar USING. Ordenar el resultado por el nombre de forma ascendente.

```
SELECT pokemon.nombre AS nombre
FROM pokemon JOIN pokemon_tipo ON pokemon.id_pokemon = pokemon_tipo.id_pokemon
      JOIN tipo ON pokemon_tipo.id_tipo = tipo.id_tipo
WHERE altura >= 1 AND tipo.nombre IN ('Eléctrico','Fuego')
ORDER BY nombre;

SELECT pokemon.nombre AS nombre
FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
WHERE altura >= 1 AND tipo.nombre IN ('Eléctrico','Fuego')
ORDER BY nombre;
```

nombre
Charizard
Charmeleon
Ninetales

- Obtener un listado que devuelva para cada región el número de Pokémon distintos capturados cuya velocidad sea superior a 50. Ordenar el resultado por el nombre de la región de forma ascendente. Añadir al listado una fila resumen que muestre el número total de Pokémon con velocidad superior a 50 capturados en cualquiera de las regiones.

```
SELECT region.nombre AS region, COUNT(DISTINCT id_pokemon) AS cant_pokemon
FROM region JOIN captura USING (id_region) JOIN pokemon USING (id_pokemon)
WHERE velocidad >= 70
GROUP BY region WITH ROLLUP;
```

region	cant_pokemon
Alola	5
Galar	5
Hisui	9
Hoenn	9
Johto	5
Kalos	9
Kanto	5
Sinnoh	4
(NULL)	19

Nota: observar que no es necesario añadir la cláusula ORDER BY ya que al realizar la agrupación se ordena de forma automática por el campo por el que se agrupa de forma ascendente, que en este ejemplo es el mismo.

- Obtener para cada entrenador, el número de Pokémon que ha capturado en los meses de julio y agosto del 2023, y la velocidad media de todos ellos (redondear la velocidad media a dos decimales). Ordenar el resultado por la velocidad media de forma descendente.

```
SELECT CONCAT_WS(' ', apellido1, apellido2, entrenador.nombre)
      AS entrenador, COUNT(*) AS cant_pokemon,
      ROUND(AVG(velocidad),2) AS vel_media
FROM entrenador JOIN captura USING (id_entrenador)
      JOIN pokemon USING (id_pokemon)
WHERE fecha_hora BETWEEN '2023-07-01' AND '2023-08-31'
GROUP BY id_entrenador
ORDER BY vel_media DESC;
```

entrenador	cant_pokemon	vel_media
Maple Max	3	85,0
Maple May	4	75,0
Saffron Iturriaga Sabrina	4	74,5
Viridian Viridian Giovanni	4	73,75
Ketchum Ash	4	73,25
Celadon Ruiz Erika	4	73,0
Pewter Brock	4	71,75
Waterflower Misty	3	70,0
Sánchez Moreno Pedro	4	65,25
Dento Iris	4	60,5
Vermilion Lt. Surge	3	55,0
Dent Cilan	3	54,0
Oak Gary	4	52,25
Forge Clemont	2	45,0

Nota: observar que en este caso es necesario añadir la cláusula ORDER BY ya que el campo por el que se quiere ordenar es distinto al que se utiliza para realizar la agrupación.

- Obtener el nombre y apellidos de los entrenadores junto a la velocidad media de los Pokémon que han capturado. Mostrar únicamente aquellos entrenadores cuyos Pokémon tengan una velocidad media superior a 70. Ordenar el resultado de mayor a menor velocidad media.

```
SELECT CONCAT_WS(' ', apellido1, apellido2, entrenador.nombre) AS
      entrenador, ROUND(AVG(velocidad),2) AS vel_media
FROM entrenador JOIN captura USING (id_entrenador) JOIN pokemon USING (id_pokemon)
GROUP BY id_entrenador
HAVING vel_media > 70
ORDER BY vel_media DESC;
```

entrenador	vel_media
Celadon Ruiz Erika	72,45
Viridian Viridian Giovanni	70,36
Waterflower Misty	70,14

7.3 Composiciones externas (LEFT, RIGHT y FULL OUTER JOIN).

En determinadas situaciones, puede que interese **seleccionar algunas filas de una tabla, aunque éstas no tengan correspondencia** (se relacionen) **con las filas de la otra tabla**.

Imaginar que se tienen guardados en dos tablas los datos de los empleados de una empresa, por un lado (*cod_empleado*, *nombre*, *apellidos*, *salario* y *cod_departamento*) y por otro lado los departamentos (*cod_departamento* y *nombre*). Suponer que recientemente se ha remodelado la empresa y se han creado un par de departamentos más, pero aún no se les han asignado empleados. Si se tuviera que obtener un informe con los datos de los empleados por departamento y se quiere que aparezcan todos los departamentos, aunque no tengan empleados, con las composiciones internas no se podría conseguir. Para poder hacer este tipo de combinaciones se usarán las composiciones externas.

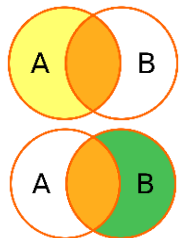
La **composición externa se escribe de forma similar al INNER JOIN indicando una condición de combinación, pero en el resultado se añaden filas que no cumplen la condición de combinación. Sintaxis:**

```
... FROM tabla1 {LEFT|RIGHT|FULL} [OUTER] JOIN tabla2 ON ([condición_combinación]) ...
... FROM tabla1 {LEFT|RIGHT|FULL} [OUTER] JOIN tabla2 USING (columna1 [, columna2, ...]) ...
```

La palabra **OUTER es opcional**, no añade ninguna funcionalidad y se utiliza si se quiere por cuestiones de estilo.

Las palabras **LEFT y RIGHT** indican la tabla de la cual se van a añadir las filas sin correspondencia:

- ✓ Si puede haber **filas de la primera tabla que no estén relacionadas con filas de la segunda tabla** e interesa que salgan en el resultado, entonces se utilizará **LEFT JOIN**.
- ✓ Si puede haber **filas de la segunda tabla que no estén relacionadas con filas de la primera tabla** e interesa que salgan en el resultado, entonces se utilizará **RIGHT JOIN**.



Una operación **LEFT JOIN** o **RIGHT JOIN** se puede anidar dentro de una operación **INNER JOIN**, pero **una operación INNER JOIN no se puede anidar dentro de LEFT JOIN o RIGHT JOIN. Los anidamientos de JOIN de distinta naturaleza no siempre funcionan**, a veces depende del orden en que se coloquen las tablas, en estos casos lo mejor es **probar y si no permite el anidamiento, cambiar el orden de las tablas** (y por tanto de los **JOINS**) dentro de la cláusula FROM.

La **composición FULL JOIN no está implementada en MariaDB**, por lo tanto, **para poder simular esta operación será necesario hacer uso del operador UNION**, que realiza la unión del resultado de dos consultas.

El resultado esperado de una **composición de tipo FULL JOIN** es obtener la intersección de las dos tablas, junto las filas de ambas tablas que no se puedan combinar. Dicho con otras palabras, el resultado sería el equivalente a **realizar la unión de una consulta de tipo LEFT JOIN y una consulta de tipo RIGHT JOIN sobre las mismas tablas**.

NATURAL LEFT JOIN realiza un **LEFT JOIN** entre las dos tablas, la única diferencia es que en este caso no es necesario utilizar **ON** o **USING** para indicar qué columna(s) va(n) a relacionar las dos tablas. En este caso las tablas se van a relacionar sobre aquellas columnas que tengan el mismo nombre. Por lo tanto, sólo se debería utilizar una composición de tipo **NATURAL LEFT JOIN** cuando se esté seguro de que los nombres de las columnas que relacionan las dos tablas se llaman igual en ambas. **NATURAL RIGHT JOIN funciona de la misma forma**.



Para más información sobre la sentencia JOIN en SQL: https://es.wikipedia.org/wiki/Sentencia_JOIN_en_SQL

Dadas las tablas *empleado* y *departamento* de la imagen anterior:

- ✓ **Ejemplo de LEFT JOIN:**

```
SELECT *
```

```
FROM empleado LEFT JOIN departamento ON (empleado.codigo_departamento = departamento.codigo);
```

Esta consulta devolverá todas las filas de la tabla que se ha colocado a la izquierda de la composición (*empleado*) y las relacionará con las filas de la tabla de la derecha (*departamento*) con las que encuentre una coincidencia. Si no encuentra ninguna coincidencia, se mostrarán los valores de la fila de la tabla izquierda (*empleado*) y en los valores de la tabla derecha (*departamento*) donde no encuentre una coincidencia mostrará el valor NULL.

El resultado de la operación LEFT JOIN es:

empleado. id	empleado. nombre	empleado. id_departamento	departamento. id	departamento. nombre
1	Pepe	1	1	Desarrollo
2	Maria	2	2	Sistemas
3	Juan	NULL	NULL	NULL

El resultado de la operación RIGHT JOIN es:

empleado. id	empleado. nombre	empleado. id_departamento	departamento. id	departamento. nombre
1	Pepe	1	1	Desarrollo
2	Maria	2	2	Sistemas
NULL	NULL	NULL	3	Recursos Humanos

✓ Ejemplo de RIGHT JOIN:

```
SELECT *
FROM empleado RIGHT JOIN departamento
ON (empleado.codigo_departamento = departamento.codigo);
```

Esta consulta devolverá todas las filas de la tabla de la derecha de la composición (*departamento*) y las relacionará con las filas de la tabla de la izquierda (*empleado*) con las que encuentre una coincidencia. Si no encuentra ninguna coincidencia, se mostrarán los valores de la fila de la tabla derecha (*departamento*) y en los valores de la tabla izquierda (*empleado*) donde no encuentre una coincidencia mostrará el valor NULL.

Cuando se utiliza LEFT JOIN, si no hay una coincidencia entre las filas de la tabla izquierda y la tabla derecha en función de la condición especificada, se devuelve NULL para las columnas de la tabla derecha. Lo mismo pasa a la inversa en el caso de RIGHT JOIN.

Ejemplos:

- Crear una consulta que devuelva un listado con los nombres de los Pokémon que evolucionan a partir de otros. En el listado deben aparecer también aquellos Pokémon que no evolucionan de ningún otro.

```
SELECT p1.nombre AS pokemon, p2.nombre AS evoluciona_de
FROM pokemon AS p1 LEFT JOIN pokemon AS p2 ON p1.evoluciona_de = p2.id_pokemon;
```

pokemon (45r x 2c)	
pokemon	evoluciona_de
Bulbasaur	(NULL)
Ivysaur	Bulbasaur
Venasaur	Ivysaur
Charmander	(NULL)
Charmeleon	Charmander
Charizard	Charmeleon
Squirtle	(NULL)

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores cuyo primer apellido contenga una "n" junto al número total de Pokémon capturados. En el caso de que el entrenador no haya capturado ningún Pokémon debe mostrarse el valor 0. Ordenar el resultado por el número de Pokémon capturados de forma descendente, en caso de coincidir, ordenar seguidamente por el primer apellido de forma ascendente. Realizar el tratamiento de valores nulos en el apellido2.
Nota: observar el valor 0 que se muestra en "Serena" (única entrenadora que aún no ha capturado ningún Pokémon, si se hace uso de un INNER JOIN su fila no se mostrará).

```
SELECT nombre, apellido1, NVL(apellido2, "") apellido2,
COUNT(id_pokemon) AS num_pok_capturados
FROM entrenador NATURAL LEFT JOIN captura
WHERE apellido1 LIKE '%n%'
GROUP BY id_entrenador
ORDER BY num_pok_capturados DESC, apellido1;
```

entrenador (8r x 4c)			
nombre	apellido1	apellido2	num_pok_capturados
Erika	Celadon	Ruiz	11
Iris	Dento		11
Sabrina	Saffron	Iturriaga	11
Pedro	Sánchez	Moreno	11
Giovanni	Viridian	Viridian	11
Cilan	Dent		8
Lt. Surge	Vermilion		6
Serena	Yvonne	López	0

- Crear una consulta que devuelva un listado con el nombre y apellidos de los entrenadores que no han capturado ningún Pokémon.

```
SELECT nombre, apellido1, NVL(apellido2, "") apellido2
FROM entrenador LEFT JOIN captura USING (id_entrenador)
WHERE id_pokemon IS NULL;
```

entrenador (1r x 3c)		
nombre	apellido1	apellido2
Serena	Yvonne	López

- Crear una consulta que devuelva un listado en el que aparezcan todas las regiones junto al número de Pokémon hembras capturados en ellas. En el caso de que en una región no se haya capturado ningún Pokémon hembra se debe mostrar el valor 0.

```
SELECT region.nombre AS region, COUNT(id_pokemon) AS num_pok_capturados_machos
FROM region LEFT JOIN captura USING (id_region)
LEFT JOIN pokemon USING (id_pokemon)
WHERE sexo LIKE '%H%' OR sexo IS NULL
```

region (9r x 2c)	
region	num_pok_capturados_machos
Aloa	12
Galar	16
Hisui	17
Hoenn	18
Johto	16
Kalos	16
Kanto	15
Sinnoh	13
Teselia	0

GROUP BY region;

- Realizar una consulta que muestre el nombre de cada Pokémon (aunque no evolucione de ningún otro), junto al nombre del Pokémon del que evoluciona y el nombre del Pokémon del que evoluciona el Pokémon desde el que evoluciona. En caso de no tener un Pokémon desde el que evolucione debe mostrar el valor "SIN EVOLUCIÓN". Por ejemplo, *Venasaur* evoluciona de *Ivysaur* que a su vez evoluciona de *Bulbasaur*.

```
SELECT pok.nombre AS pokemon, NVL(evol1.nombre, "SIN EVOLUCIÓN") AS "Evoluciona de 1",
      NVL(evol2.nombre, "SIN EVOLUCIÓN") AS "Evoluciona de 2"
FROM pokemon AS pok LEFT JOIN pokemon AS evol1 ON (pok.evoluciona_de=evol1.id_pokemon)
      LEFT JOIN pokemon AS evol2 ON (evol1.evoluciona_de=evol2.id_pokemon);
```

pokemon (45r x 3c)		
pokemon	Evoluciona de 1	Evoluciona de 2
Bulbasaur	SIN EVOLUCIÓN	SIN EVOLUCIÓN
Ivysaur	Bulbasaur	SIN EVOLUCIÓN
Venasaur	Ivysaur	Bulbasaur
Charmander	SIN EVOLUCIÓN	SIN EVOLUCIÓN
Charmeleon	Charmander	SIN EVOLUCIÓN
Charizard	Charmeleon	Charmander
Squirtle	SIN EVOLUCIÓN	SIN EVOLUCIÓN
Wartortle	Squirtle	SIN EVOLUCIÓN

7.4 Composiciones en la versión SQL99 (resumen).

Como se ha visto, SQL incluye en esta versión mejoras de la sintaxis a la hora de crear composiciones en consultas. Recordar que la **sintaxis** a utilizar con *MariaDB* es la siguiente:

```
SELECT [tabla1.]columna1, [tabla1.]columna2, ..., [tabla2.]columna3, [tabla2.]columna4, ...
FROM tabla1
      [CROSS JOIN tabla2] |
      [NATURAL JOIN tabla2] |
      [JOIN tabla2 USING (columna1, ...)] |
      [JOIN tabla2 ON [( )tabla1.columna = tabla2.columna [AND ...] ( )]] |
      [LEFT | RIGHT JOIN tabla2 USING (columna1[, columna2, ...])] |
      [LEFT | RIGHT JOIN tabla2 ON [( )tabla1.columna = tabla2.columna [AND ...] ( )]]
...;
```

- ✓ **CROSS JOIN**: creará un **producto cartesiano de las filas de ambas tablas** por lo que no es necesaria la cláusula WHERE/ON/USING.
- ✓ **NATURAL JOIN**: detecta automáticamente los campos de unión, basándose en el nombre de las columnas que coinciden en ambas tablas. Esta característica funcionará incluso si no están definidas las claves primarias o ajenas. Se debe tener cuidado con ella al utilizarla.
- ✓ **JOIN - USING**: esta cláusula **permite establecer relaciones indicando qué campo o campos comunes se quieren utilizar para ello, los nombres de las columnas deben coincidir en ambas tablas** (las tablas pueden tener más de un campo para relacionarlas y no siempre se requiere que se relacionen por todos los campos).
- ✓ **JOIN - ON**: se utiliza **para unir tablas en las que los nombres de las columnas que las relacionan no coinciden en ambas tablas** o se necesitan establecer asociaciones más complicadas.
- ✓ **[NATURAL] LEFT JOIN [ON/USING]**: composición externa izquierda. **Devuelve todas las filas de la tabla de la izquierda** (tabla izquierda en la cláusula LEFT JOIN) **y las filas coincidentes de la tabla de la derecha.**
- ✓ **[NATURAL] RIGHT JOIN [ON/USING]**: composición externa derecha. **Se devuelven todas las filas de la tabla de la derecha se devuelven y las filas coincidentes de la tabla de la izquierda.**



TAREA

Utilizando las tablas y datos de las **BD “conferencias”** y **“conferencias2”** (una sentencia para cada BD y ejercicio):

- Realizar una consulta donde se obtenga el nombre de los ponentes junto a la referencia y el tema de las conferencias en las que han participado.
- Realizar una consulta que obtenga un listado con el nombre y apellidos de los asistentes que hayan asistido a la conferencia con referencia “PWB1314”.
- Realizar una consulta que muestre el número de asistentes a cada una de las conferencias.
- Realizar una consulta que muestre la sala donde cada ponente realiza su conferencia. La consulta además de los datos del ponente debe mostrar el nombre de la sala, el tema de la conferencia y el orden de intervención. Los resultados se deben ordenar por el tema y el orden de intervención.
- Realizar una consulta que muestre el total de asistentes por conferencia y sala. El resultado se debe ordenar por el número de asistentes.
- Realizar una consulta que muestre los distintos ponentes que han utilizado la sala “Afrodita”.

8 Otras consultas multitabla: *UNION*, *INTERSECT* y *EXCEPT*.

En *SQL*, **UNION**, **INTERSECT** y **EXCEPT** son operadores que se utilizan para combinar resultados de múltiples consultas. Descripción breve de qué hace cada uno:

- ✓ **UNION**: combina los resultados de dos o más consultas en un solo conjunto de resultados. Si hay duplicados, se eliminan automáticamente del resultado final.
- ✓ **INTERSECT**: devuelve todos los registros que están presentes en ambas consultas. Es decir, devuelve la intersección de los conjuntos de resultados de las dos consultas.
- ✓ **EXCEPT**: devuelve todos los registros que están presentes en la primera consulta pero no en la segunda. Es decir, devuelve los resultados de la primera consulta que no están presentes en la segunda consulta.

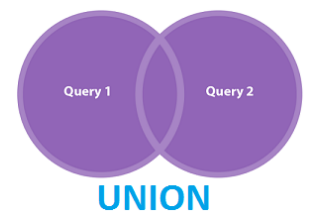
Tener en cuenta que las consultas en **UNION**, **INTERSECT** y **EXCEPT** deben tener la misma cantidad de columnas y los tipos de datos de esas columnas deben coincidir entre ellas para que la operación tenga éxito.

Estas operaciones se pueden combinar anidadas, pero es conveniente utilizar paréntesis para indicar que operación se quiere que se haga primero.

Ejemplos: las tablas que se utilizan no tienen nada que ver con las que se plantean en la BD “pokemon”.

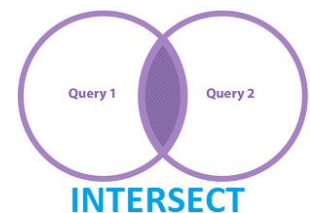
- ✓ **UNIÓN**: obtener los nombres y ciudades de todos los proveedores y clientes de “Alemania”.

```
SELECT nombre, ciudad FROM proveedor WHERE pais='Alemania'
UNION
SELECT nombre, ciudad FROM cliente WHERE pais='Alemania';
```



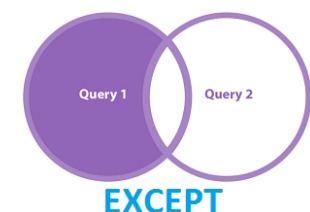
- ✓ **INTERSECCIÓN**: una academia de idiomas da clases de inglés, francés y portugués; almacena los datos de los alumnos en tres tablas distintas: *ingles*, *frances* y *portugues*. La academia necesita el nombre y domicilio de todos los alumnos que cursan los tres idiomas para enviarles información sobre los exámenes.

```
SELECT nombre, domicilio FROM ingles
INTERSECT
SELECT nombre, domicilio FROM frances
INTERSECT
SELECT nombre, domicilio FROM portugues;
```



- ✓ **DIFERENCIA**: ahora la academia necesita el nombre y domicilio solo de los alumnos que cursan inglés y no portugués pues les va a enviar publicidad referente al curso de portugués.

```
SELECT nombre, domicilio FROM ingles
EXCEPT
SELECT nombre, domicilio FROM portugues;
```



Ejemplos: sobre una misma tabla.

- Crear una consulta que devuelva un listado con el nombre y apellidos de los entrenadores que no han capturado ningún Pokémon.

```
SELECT nombre, apellido1, NVL(apellido2,"") apellido2
FROM entrenador
EXCEPT
SELECT nombre, apellido1, NVL(apellido2,"") apellido2
FROM entrenador JOIN captura USING (id_entrenador);
```

entrenador (1r × 3c)		
nombre	apellido1	apellido2
Serena	Yvonne	López

- Crear una consulta que devuelva el nombre de los Pokémon que son de tipo “Planta” y “Veneno”.

```
SELECT pokemon.nombre AS Pokemon
FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
WHERE tipo.nombre = "Planta"
INTERSECT
SELECT pokemon.nombre AS Pokemon
FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
WHERE tipo.nombre = "Veneno";
```

pokemon (3r × 1c)	
Pokemon	
Bulbasaur	
Ivysaur	
Venasaur	

- Crear una consulta que devuelva el nombre de los Pokémon que son de tipo “Planta” y/o “Veneno” sin que se repitan. Ordenar el resultado por el nombre de los Pokémon de forma ascendente.

```
SELECT pokemon.nombre AS Pokemon
FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
WHERE tipo.nombre = "Planta"
UNION
SELECT pokemon.nombre AS Pokemon
FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
WHERE tipo.nombre = "Veneno"
ORDER BY Pokemon;
```

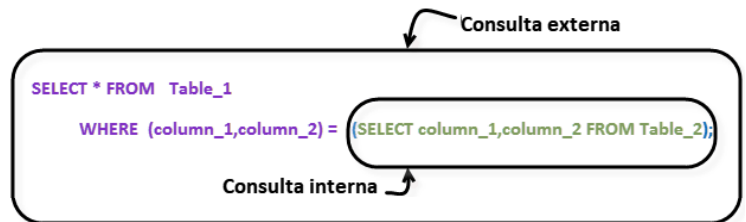
pokemon (16r x 4c)

Pokemon
Arbok
Beedrill
Bulbasaur
Ekans
Golbat
Ivysaur
Kakuna
Nidoking
Nidoqueen
Nidran O

9 Subconsultas.

Una subconsulta *SQL* es una consulta anidada dentro de otra consulta *SQL* (a veces se tendrá que utilizar en una consulta los resultados de otra).

La subconsulta puede ir dentro de las cláusulas SELECT, FROM, WHERE y/o HAVING.



9.1 Subconsultas en la cláusula SELECT.

La sintaxis básica de uso de **subconsultas en la cláusula SELECT**:

```
SELECT lista_expr1, ..., (SELECT columnaN FROM tabla2 [WHERE condición]) AS nombre_columna
FROM tabla1
[WHERE condición];
```

El resultado de la subconsulta se muestra como una columna adicional en el resultado de la consulta principal. Es importante tener en cuenta que la subconsulta en la cláusula SELECT debe devolver un solo valor y una sola columna. Si devuelve más de un valor o más de una columna, se producirá un error.

Ejemplos:

- Obtener el nombre y apellidos de los entrenadores como campos individuales y haciendo uso de subconsultas mostrar el nombre completo en el formato “apellido1 apellido2, nombre”.

```
SELECT nombre, apellido1, apellido2, (SELECT
  CONCAT(apellido1, IF(apellido2 IS NULL, '', CONCAT(' ',
    apellido2)), ', ', nombre)) AS nombre_completo
FROM entrenador;
```

entrenador (15r x 4c)

nombre	apellido1	apellido2	nombre_completo
Ash	Ketchum	(NULL)	Ketchum, Ash
Misty	Waterflower	(NULL)	Waterflower, Misty
Brock	Pewter	(NULL)	Pewter, Brock
May	Maple	(NULL)	Maple, May
Max	Maple	(NULL)	Maple, Max
Garv	Oak	(NULL)	Oak, Garv

- Obtener el nombre y apellidos de los entrenadores junto al número de Pokémon que han capturado.

```
SELECT nombre, apellido1, apellido2,
  (SELECT COUNT(*)
   FROM captura
   WHERE captura.id_entrenador = entrenador.id_entrenador)
  AS pok_capturados
FROM entrenador;
```

entrenador (15r x 4c)

nombre	apellido1	apellido2	pok_capturados
Ash	Ketchum	(NULL)	11
Misty	Waterflower	(NULL)	7
Brock	Pewter	(NULL)	11
May	Maple	(NULL)	11
Max	Maple	(NULL)	7

- Obtener el número total de Pokémon y entrenadores que se tienen en la BD.

```
SELECT (SELECT COUNT(*) FROM pokemon) AS TotalPokemon,
  (SELECT COUNT(*) FROM entrenador) AS TotalEntrenadores;
```

pokemon (1r x 2c)

TotalPokemon	TotalEntrenadores
45	15

9.2 Subconsultas en la cláusula FROM.

La sintaxis básica de uso de **subconsultas en la cláusula FROM**:

```
SELECT lista_expr1
FROM (SELECT columnaN FROM tabla2 [WHERE condición]) AS alias
[WHERE condición];
```

La subconsulta se coloca dentro de paréntesis y se utiliza como una tabla derivada en la cláusula FROM (conjunto de datos temporal que se puede utilizar en la consulta principal). La subconsulta puede tener sus propias cláusulas SELECT, FROM, WHERE y cualquier otra cláusula que sea necesaria para generar el resultado deseado.

Se asigna un alias a la subconsulta utilizando la palabra clave AS, y luego se utiliza ese alias para referirse a la subconsulta en la consulta principal.

Es importante tener en cuenta que la subconsulta utilizada en la cláusula FROM debe devolver un conjunto de filas y columnas, ya que se trata como una tabla en la consulta principal.

Ejemplos:

- Crear una consulta que muestre el nombre y apellidos de los entrenadores que han capturado menos de ocho Pokémon. Ordenar el resultado de forma ascendente por los apellidos y el nombre. Nota: este mismo ejemplo se resuelve haciendo uso de subconsultas en la cláusula WHERE con IN y EXISTS.

```
SELECT nombre, apellido1, apellido2
FROM entrenador JOIN (SELECT id_entrenador, COUNT(*) AS pok_capturados
                      FROM captura
                      GROUP BY id_entrenador) AS capt USING (id_entrenador)
WHERE pok_capturados < 8
ORDER BY apellido1, apellido2, nombre;
```

nombre	apellido1	apellido2
Clemont	Forge	(NULL)
Max	Maple	(NULL)
Lt. Surge	Vermilion	(NULL)
Misty	Waterflower	(NULL)

- Crear una consulta que devuelva el nombre del objeto del que se tiene más cantidad en las mochilas.

```
SELECT nombre
FROM objeto JOIN (SELECT id_objeto, SUM(cantidad) AS total
                  FROM mochila
                  GROUP BY id_objeto
                  ORDER BY total DESC
                  LIMIT 1) AS totales USING (id_objeto);
```

nombre
MT01 - Puño certero

- Devolver el nombre y apellidos del entrenador con menor sueldo (no se puede utilizar subconsultas en la cláusula WHERE, ni LIMIT ni ORDER BY).

```
SELECT nombre, apellido1, apellido2
FROM entrenador JOIN (SELECT MIN(sueldo) AS min_sueldo
                      FROM entrenador) AS subconsulta
ON entrenador.sueldo = subconsulta.min_sueldo;
```

nombre	apellido1	apellido2
Max	Maple	(NULL)

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores que han capturado Pokémon en dos o menos regiones distintas. No se puede hacer uso de HAVING ni LIMIT. Ordenar el resultado por los apellidos y el nombre de los entrenadores.

```
SELECT CONCAT(nombre, ' ', apellido1, ' ', NVL(apellido2, '')) AS nombre_completo
FROM entrenador JOIN (SELECT id_entrenador, COUNT(DISTINCT id_region) AS num_regiones
                      FROM captura JOIN region USING (id_region)
                      GROUP BY id_entrenador) AS capt_reg USING (id_entrenador)
WHERE num_regiones <= 2
ORDER BY apellido1, apellido2, nombre;
```

nombre_completo
Cilan Dent
Clemont Forge
Max Maple
Lt. Surge Vermilion
Misty Waterflower

9.3 Subconsultas en la cláusula WHERE.

Son consultas internas que se utilizan para filtrar los resultados de la consulta principal basándose en los resultados devueltos por la subconsulta. Estas subconsultas se ejecutan primero y el resultado se utiliza para filtrar las filas de la tabla o tablas especificadas en la consulta principal. Son útiles para realizar filtros más complejos.

La sintaxis básica de uso de subconsultas en la cláusula WHERE es:

```
SELECT lista_expr1
FROM tabla1
WHERE expresión OPERADOR (SELECT lista_expr2 FROM tabla2 [WHERE condición]);
```

El uso de subconsultas en la cláusula WHERE implica **colocar la subconsulta entre paréntesis y utilizarla en lugar de un valor o una expresión**. Esto permite realizar comparaciones más complejas y utilizar resultados de subconsultas para filtrar los resultados de la consulta principal.

Los **operadores que se pueden usar en las subconsultas** son los siguientes:

- ✓ **Operadores básicos de comparación:** >, >=, <, <=, !=, <> y = (la subconsulta devuelve un valor).
- ✓ Predicados **ALL** y **ANY** (resultado de la subconsulta devuelve más de un valor).
- ✓ Predicados **IN** y **NOT IN** (resultado de la subconsulta devuelve más de un valor).
- ✓ Predicados **EXISTS** y **NOT EXISTS** (para determinar si la subconsulta devuelve o no resultados).

9.3.1 Subconsultas que devuelven un único valor.

Los **operadores básicos de comparación** se van a utilizar para realizar comparaciones con **subconsultas que devuelven un único valor**, es decir, una columna y una fila. Si la subconsulta devolviera más de un valor se produciría un error. Las **subconsultas que devuelven un único valor son subconsultas escalares**.

Como se puede ver en la sintaxis, **las subconsultas deben ir entre paréntesis y a la derecha del operador**. El tipo de datos que devuelve la subconsulta y el de la columna con la que se compara han de ser el mismo.

Ejemplos:

- Crear una consulta que devuelva el nombre y apellidos del entrenador (o entrenadores) con menor sueldo (no se puede utilizar ni LIMIT ni ORDER BY).

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE sueldo = (SELECT MIN(sueldo) FROM entrenador);
```

entrenador (1r x 3c)		
nombre	apellido1	apellido2
Max	Maple	(NULL)

- Crear una consulta que devuelva el nombre y apellidos del entrenador más joven.

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE fecha_nacimiento = (SELECT MIN(fecha_nacimiento) FROM entrenador);
```

entrenador (1r x 3c)		
nombre	apellido1	apellido2
Giovanni	Viridian	Viridian

- Crear una consulta que devuelva el nombre de los Pokémon cuya velocidad sea superior a la de cualquier Pokémon de tipo 'Veneno'. Se proponen tres soluciones distintas.

```
SELECT nombre
FROM pokemon
WHERE velocidad > (SELECT MAX(velocidad)
FROM pokemon
WHERE id_pokemon IN (SELECT id_pokemon
FROM pokemon_tipo
WHERE id_tipo IN (SELECT id_tipo
FROM tipo
WHERE nombre = 'Veneno')));

SELECT nombre
FROM pokemon
WHERE velocidad > (SELECT MAX(velocidad)
FROM pokemon JOIN pokemon_tipo USING (id_pokemon)
JOIN tipo USING (id_tipo)
WHERE tipo.nombre = 'Veneno');
```

pokemon (6r x 2c)	
nombre	velocidad
Charizard	100
Pidgeot	100
Raticate	100
Fearow	100
Raichu	100
Ninetales	100

```
SELECT nombre
FROM pokemon
WHERE velocidad > ALL (SELECT velocidad
FROM pokemon JOIN pokemon_tipo USING (id_pokemon)
JOIN tipo USING (id_tipo)
WHERE tipo.nombre = 'Veneno');
```

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores que han capturado Pokémon en dos o menos regiones distintas. No se puede hacer uso de HAVING ni LIMIT. Ordenar el resultado por los apellidos y el nombre de los entrenadores.

```
SELECT CONCAT(nombre, ' ', apellido1, ' ', NVL(apellido2, '')) AS nombre_completo
FROM entrenador
WHERE (SELECT COUNT(DISTINCT id_region)
FROM captura JOIN region USING (id_region)
WHERE captura.id_entrenador = entrenador.id_entrenador) <= 2
AND (SELECT COUNT(DISTINCT id_region)
FROM captura JOIN region USING (id_region)
WHERE captura.id_entrenador = entrenador.id_entrenador) >= 1
ORDER BY apellido1, apellido2, nombre;
```

entrenador (5r x 2c)	
nombre_completo	id_entrenador
Cilan Dent	1
Clemont Forge	2
Max Maple	3
Lt. Surge Vermilion	4
Misty Waterflower	5

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores que son mayores que 'Cilan Dent'. Nota: se supone que la combinación de nombre y apellidos es una clave única (la subconsulta en ningún caso devuelve más de un valor).

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE fecha_nacimiento < (SELECT fecha_nacimiento
                           FROM entrenador
                           WHERE nombre='Cilan' AND apellido1='Dent' AND apellido2 IS NULL);
```

nombre	apellido1	apellido2
Brock	Pewter	(NULL)
Lt. Surge	Vermilion	(NULL)
Giovanni	Viridian	Viridian

9.3.2 Subconsultas que devuelven más de un valor.

En SQL, una subconsulta que devuelve más de un valor se puede utilizar en varias formas, dependiendo del contexto y los requisitos específicos de la consulta.

Cuando el **resultado de la subconsulta devuelve más de un valor**, se debe utilizar alguno de los siguientes operadores:

- ✓ **IN:** se utiliza para **verificar si un valor coincide con alguno de los valores devueltos por una subconsulta**.
Sintaxis básica:

```
SELECT columna
FROM tabla
WHERE columna IN (SELECT columna FROM otra_tabla WHERE condición);
```

- ✓ **NOT IN:** se utiliza para **verificar si un valor no coincide con ninguno de los valores devueltos por una subconsulta**. Sintaxis básica:

```
SELECT columna
FROM tabla
WHERE columna NOT IN (SELECT columna FROM otra_tabla WHERE condición);
```

- ✓ **ANY:** se utiliza para **comparar un valor con cualquier valor individual de una subconsulta**. La palabra reservada **SOME** es un alias de ANY. Sintaxis básica:

```
SELECT columna
FROM tabla
WHERE columna <operador_básico> ANY (SELECT columna FROM otra_tabla WHERE condición);
```

- ✓ **ALL:** se utiliza para **comparar un valor con todos los valores de una subconsulta**. La comparación resultará cierta, si es cierta toda la comparación con los registros de la subconsulta. Sintaxis básica:

```
SELECT columna
FROM tabla
WHERE columna <operador_básico> ALL (SELECT columna FROM otra_tabla WHERE condición);
```

Ejemplos:

- Crear una consulta que devuelva el nombre de los objetos que no tiene ningún entrenador en su mochila. Ordenar el resultado de forma ascendente por el nombre del objeto.

```
SELECT nombre
FROM objeto
WHERE id_objeto NOT IN (SELECT id_objeto FROM mochila)
ORDER BY nombre;
```

nombre
Baya Oran
Bayas
MT04 - Garra umbría
MT05 - Rayo solar
MT06 - Lanzamiento
Piedra trueno

- Crear una consulta que muestre el nombre y apellidos de los entrenadores que han capturado menos de ocho Pokémon. Ordenar el resultado de forma ascendente por los apellidos y el nombre.

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE id_entrenador IN (SELECT id_entrenador
                        FROM captura
                        GROUP BY id_entrenador
                        HAVING COUNT(*) < 8)
ORDER BY apellido1, apellido2, nombre;
```

nombre	apellido1	apellido2
Clemont	Forge	(NULL)
Max	Maple	(NULL)
Lt. Surge	Vermilion	(NULL)
Misty	Waterflower	(NULL)

- Crear una consulta que devuelva un listado con los nombres de los Pokémon de tipo 'Eléctrico' que no han sido capturados aún por ningún entrenador. Ordenar el resultado por el nombre de los Pokémon de forma

ascendente.

```
SELECT nombre
FROM pokemon
WHERE id_pokemon IN (SELECT id_pokemon
                     FROM pokemon_tipo JOIN tipo USING (id_tipo)
                     WHERE nombre = 'Eléctrico')
AND id_pokemon NOT IN (SELECT id_pokemon
                      FROM captura)
ORDER BY nombre;
```

pokemon (1r x 1c)	
nombre	Pichu

- Obtener el nombre de los Pokémon cuya debilidad sea 'Tierra' o 'Roca' y tengan una velocidad mayor o igual a la de todos los Pokémon del tipo 'Normal' (no se pueden utilizar las funciones MAX y MIN, ni la cláusula ORDER BY).

```
SELECT DISTINCT nombre
FROM pokemon
WHERE velocidad >= ALL (SELECT velocidad
                       FROM pokemon JOIN pokemon_tipo USING (id_pokemon) JOIN tipo USING (id_tipo)
                       WHERE tipo.nombre = 'Normal')
AND id_pokemon IN (SELECT id_pokemon
                  FROM pokemon_debilidad JOIN tipo ON pokemon_debilidad.id_debilidad = tipo.id_tipo
                  WHERE tipo.nombre IN ('Tierra', 'Roca'));
```

pokemon (4r x 1c)	
nombre	Charizard
	Fearow
	Raichu
	Ninetales

- Haciendo uso de subconsultas (no se puede utilizar ningún JOIN) obtener un listado con los nombres de los Pokémon que han sido capturados en la región 'Kanto', tienen alguna de las siguientes habilidades 'Velo Arena', 'Punto Tóxico' y/o 'Rivalidad', su velocidad está comprendida entre 30 y 80, y además los ha capturado algún entrenador que en su mochila tiene el objeto 'Poción'.

```
SELECT nombre
FROM pokemon
WHERE id_pokemon IN (SELECT id_pokemon
                     FROM captura
                     WHERE id_region IN (SELECT id_region
                                         FROM region
                                         WHERE nombre = 'Kanto'))
AND id_pokemon IN (SELECT id_pokemon
                  FROM pokemon_habilidad
                  WHERE id_habilidad IN (SELECT id_habilidad
                                         FROM habilidad
                                         WHERE nombre IN ('Velo Arena', 'Punto Tóxico', 'Rivalidad'))))
AND velocidad BETWEEN 30 AND 80
AND id_pokemon IN (SELECT id_pokemon
                  FROM captura
                  WHERE id_entrenador IN (SELECT id_entrenador
                                         FROM mochila
                                         WHERE id_objeto IN (SELECT id_objeto
                                                             FROM objeto
                                                             WHERE nombre = 'Poción'))));
```

pokemon (2r x 1c)	
nombre	Nidoqueen
	Sandslash

9.3.3 Comprobar si la subconsulta devuelve o no resultados.

EXISTS y NOT EXISTS son operadores utilizados en subconsultas para verificar la existencia de registros que cumplan ciertas condiciones. La diferencia entre ellos es que **EXISTS** devuelve verdadero si la subconsulta devuelve algún resultado, mientras que **NOT EXISTS** devuelve verdadero si la subconsulta no devuelve ningún resultado.

Sintaxis básica:

```
SELECT columna
FROM tabla
WHERE [NOT] EXISTS (SELECT columna FROM otra_tabla WHERE condición);
```

Es importante tener en cuenta que **las subconsultas pueden estar relacionadas con la consulta principal utilizando cláusulas de correlación**, como agregar una condición que asocie las tablas en la subconsulta y la consulta principal. Esto permite realizar comparaciones más específicas y contextuales en las subconsultas **EXISTS** y **NOT EXISTS**.

Notar que **EXISTS y NOT EXISTS se utilizan principalmente en combinación con otras cláusulas, como WHERE**, para filtrar los resultados en función de la existencia o no existencia de registros en una tabla relacionada.

Ejemplos:

- Crear una consulta que muestre el nombre y apellidos de los entrenadores que han capturado menos de ocho Pokémon. Ordenar el resultado de forma ascendente por los apellidos y el nombre.

```
SELECT nombre, apellido1, apellido2
FROM entrenador e
WHERE EXISTS (SELECT id_entrenador
              FROM captura c
              WHERE e.id_entrenador = c.id_entrenador
              GROUP BY id_entrenador
              HAVING COUNT(*) < 8)
ORDER BY apellido1, apellido2, nombre;
```

nombre	apellido1	apellido2
Clemont	Forge	(NULL)
Max	Maple	(NULL)
Lt. Surge	Vermilion	(NULL)
Misty	Waterflower	(NULL)

- Crear una consulta que devuelva el nombre de las regiones donde no se ha capturado ningún Pokémon.

```
SELECT nombre
FROM region
WHERE NOT EXISTS (SELECT id_pokemon -- 1
                  FROM captura
                  WHERE region.id_region = captura.id_region);
```

nombre
Teselia

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores que han capturado algún Pokémon con velocidad menor o igual a 20. Ordenar el resultado de forma ascendente por los apellidos y el nombre.

```
SELECT nombre, apellido1, apellido2
FROM entrenador
WHERE EXISTS (SELECT id_pokemon
              FROM captura JOIN pokemon USING (id_pokemon)
              WHERE entrenador.id_entrenador = captura.id_entrenador AND velocidad <= 20)
ORDER BY apellido1, apellido2, nombre;
```

nombre	apellido1	apellido2
Gary	Oak	(NULL)
Brock	Pewter	(NULL)
Sabrina	Saffron	Iturriaga
Pedro	Sánchez	Moreno
Giovanni	Viridian	Viridian

9.4 Subconsultas en la cláusula *HAVING*.

La cláusula **HAVING** se utiliza junto con la cláusula **GROUP BY** para **aplicar condiciones a los grupos resultantes después de agrupar los datos**. Aunque normalmente se utiliza para especificar condiciones de filtrado directamente, **también se pueden utilizar subconsultas en la cláusula HAVING para realizar condiciones más complejas**.

La sintaxis básica de uso de **subconsultas en la cláusula HAVING**:

```
SELECT lista_expr1
FROM tabla1
[WHERE condición]
GROUP BY columna1 [, columna2, ...]
HAVING expresión OPERADOR (SELECT lista_expr2
                             FROM tabla2
                             [WHERE condición]);
```

La **subconsulta se coloca dentro de paréntesis y se utiliza en lugar de un valor o una expresión en la cláusula HAVING**. Recordar que las **subconsultas en la cláusula HAVING deben devolver un solo valor**, ya que se comparan con una expresión agregada. Si la subconsulta devuelve más de un valor, se producirá un error.

Los **operadores** que se pueden utilizar son los **mismos que se pueden utilizar en la cláusula WHERE** (las situaciones que se pueden dar son las mismas).

Ejemplos:

- Devolver el nombre del o de los Pokémon de los que se han capturado menos cantidad.

```
SELECT nombre
FROM pokemon JOIN captura USING (id_pokemon)
GROUP BY id_pokemon
HAVING COUNT(id_entrenador) = (SELECT MIN(recuento)
                              FROM (SELECT id_pokemon, COUNT(id_entrenador) AS recuento
                                    FROM captura
                                    GROUP BY id_pokemon) AS subconsulta)
ORDER BY nombre;
```

nombre
Charmeleon
Cleffa
Ekans
Kakuna
Nidoran♂
Squirtle
Weedle
Wigglytuff
Zubat

- Crear una consulta que devuelva el nombre y apellidos de los entrenadores cuya cantidad total de Pokémon capturados esté por debajo de la media.

```
SELECT nombre, apellido1, apellido2,
       COUNT(DISTINCT id_pokemon) AS pok_capturados
FROM entrenador JOIN captura USING (id_entrenador)
GROUP BY id_entrenador
HAVING pok_capturados < (SELECT AVG(n_capturas)
                        FROM (SELECT id_entrenador, COUNT(DISTINCT id_pokemon) AS n_capturas
                              FROM captura
                              GROUP BY id_entrenador) AS subconsulta);
```

nombre	apellido1	apellido2	pok_capturados
Misty	Waterflower	(NULL)	7
Max	Maple	(NULL)	7
Clemont	Forge	(NULL)	6
Cilan	Dent	(NULL)	8
Lt. Surge	Vermilion	(NULL)	6

TAREA

Utilizando las tablas y datos de la BD “conferencias” (hacer uso de subconsultas si no se dice nada y es posible):

34. Realizar una consulta que seleccione los ponentes (nombre, apellido1 y apellido2) cuyo primer apellido sea igual al primer apellido del asistente de menor edad.
35. Realizar una consulta (utilizando subconsultas) que obtenga los distintos ponentes (nombre, apellido1 y apellido2) que han usado la sala “Afrodita” para dar una conferencia.
36. Realizar una consulta que muestre los asistentes (nombre, apellido1 y apellido2) de la empresa “BigSoft” que asisten a algunas de las sesiones de la conferencia sobre “Programación Web”.
37. Realizar una consulta que muestre los asistentes (nombre, apellido1 y apellido2) que sean hombres y hayan nacido antes del “01/01/1985”, y además hayan asistido a una conferencia sobre “Programación Web”.
38. Realizar una consulta que muestre el total de gratificaciones recibidas por cada uno de los ponentes.
39. Realizar una consulta que muestre los asistentes (nombre, apellido1 y apellido2) a cada una de las conferencias que se celebran el día “02/10/2013” (tema). El resultado debe mostrarse ordenado por el tema de la conferencia, así como por los apellidos y nombre de los asistentes.
40. Obtener el nombre de las salas que únicamente se utilizan en el turno de tarde y no en el de mañana. Proponer dos sentencias, una que haga uso de *EXCEPT* y otra con subconsultas.
41. Obtener el nombre de las salas que se utilizan en el turno de mañana y tarde. Proponer tres sentencias distintas, una que haga uso de *INTERSECT*, otra multitabla con *JOIN* y otra con subconsultas.
42. Obtener el nombre y apellidos (apellido1 y apellido2) del ponente o ponentes que cobra(n) la mayor gratificación. Hacer uso únicamente de subconsultas.
43. Obtener el nombre y apellidos (apellido1 y apellido2) del asistente de mayor edad que asiste a la conferencia más cara. Hacer uso únicamente de subconsultas.
44. Obtener el nombre, apellidos y gratificación de aquellos ponentes cuya gratificación sea mayor a la de todos los ponentes que hayan realizado la conferencia en la sala “Afrodita”. Mezclar el uso de *JOIN* y subconsultas.
45. Obtener el nombre o nombres de las salas y su capacidad donde participa el ponente de la especialidad “Seguridad Informática” que recibe la mayor gratificación. Mezclar el uso de *JOIN* y subconsultas.
46. Obtener los ponentes de la especialidad “Bases de Datos” que tienen una gratificación mayor a la de alguno de los ponentes de “Programación Orientada a Objetos”. Mezclar el uso de *JOIN* y subconsultas.
47. Seleccionar la sala o salas donde no se han celebrado conferencias. Utilizar subconsultas que usen *NOT EXISTS*.
48. Seleccionar los ponentes que han utilizado la sala “Zeus”. Utilizar subconsultas que hagan uso de *EXISTS*.

10 Rendimiento de consultas.

10.1 Introducción.

SQL es un lenguaje declarativo, cada consulta declara qué se quiere que haga el motor de *SQL* (*SGBD*), pero no dice cómo. Sin embargo, resulta que el cómo, el “plan”, es lo que afecta a la eficiencia de las consultas, así que es bastante importante. Por ejemplo, suponer que se tiene esta consulta sencilla:

```
SELECT * FROM libros WHERE autor = "J K Rowling";
```

Para esta consulta, hay dos **formas diferentes de encontrar los resultados**:

- ✓ Hacer una **exploración completa de la tabla**: buscar en cada fila y devolver las filas que coincidan.
- ✓ Crear un **índice**: hacer una copia de la tabla ordenada por autor, después hacer una búsqueda binaria para encontrar la fila en la que el autor es "J K Rowling", encontrar los *IDs* que coincidan y finalmente hacer una búsqueda binaria en la tabla original que devuelva las filas que coincidan con el *ID*.

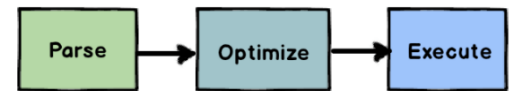
¿Cuál es más rápida? Depende de los datos y de qué tan frecuentemente sea ejecutada la consulta. Si la tabla solo tiene 10 filas, entonces una exploración completa solo requiere ver 10 filas y el primer plan funcionaría bien.

Si la tabla tuviera 10 millones de filas, entonces una exploración completa de la tabla requeriría ver 10 millones de filas. Sería más rápido hacer una búsqueda binaria en una tabla ordenada: solo se necesitan 23 búsquedas para encontrar un valor en 10 millones de filas. Sin embargo, crear la tabla ordenada tomaría un buen tiempo (~230 millones de operaciones, dependiendo del motor). Si se ejecutara esa consulta muchas veces (más de 23 veces) o si ya se tuviera esa tabla creada, entonces el segundo plan sería mejor.

¿Cómo decide un motor de SQL qué plan elegir? Ese es un paso importante acerca del cual no se ha hablado aún porque se ha estado enfocado en la sintaxis de las consultas, no en su implementación. **A medida que se llega a un uso más avanzado de SQL en BD grandes, el paso de planeación y optimización de consultas se vuelve cada vez más importante.**

El motor de SQL pasa por los siguientes pasos para cada consulta:

1. El **analizador de consultas** se asegura de que la consulta sea **sintácticamente y semánticamente correcta**. Si es correcta, entonces la convierte en una expresión algebraica y la pasa al siguiente paso.
2. El **planeador y optimizador de consultas** hace el trabajo pesado de pensar. Primero realiza optimizaciones directas (mejoras que siempre resultan en un mejor rendimiento, como simplificar $5*10$ en 50). Después considera diferentes "planes de consulta" que pueden tener diferentes optimizaciones, estima el costo (*CPU* y tiempo) de cada consulta con base en el número de filas en las tablas relevantes, después escoge el plan óptimo y lo pasa al siguiente paso.
3. El **ejecutor de la consulta** toma el plan y lo convierte en operaciones de la *BD*, regresando los resultados si es que hay algunos.



La planeación y optimización de la consulta sucede para cada consulta, y se podría estar toda la vida ejecutando consultas de SQL sin ser consciente de ello. Sin embargo, una vez que se empieza a lidiar con conjuntos de datos más grandes, habrá que preocuparse más por la velocidad de las consultas.

Muchas veces, especialmente para consultas complicadas, ciertamente hay maneras en las que se puede ayudar a optimizar una consulta, y eso se llama **afinación de la consulta**.

El primer paso es **identificar qué consultas se quieren afinar**, las cuales pueden averiguarse al ver cuáles de las llamadas al *SGBD* están tardando más o usando la mayor parte de los recursos, ¿cómo? con un analizador de SQL. Algunas veces, se puede descubrir una consulta con un muy mal desempeño después de que se tarda tanto que hace que se caiga toda la *BD* (*SGBD*). Con fortuna, se habrá averiguado eso antes.

El siguiente paso es **entender cómo un motor de SQL particular está ejecutando una consulta, y todos los sistemas de SQL vienen con una manera de preguntarle al motor.**

Ahora viene la parte difícil: la **optimización manual para mejorar el plan de ejecución**. Esta también es la parte que a menudo depende de las particularidades del motor de SQL que se esté usando, así como de las particularidades de los propios datos. **Crear índices a menudo puede hacer más eficientes las consultas que se repiten, pero también hay otros enfoques.**

10.2 Actualizar y reordenar índices.

OPTIMIZE TABLE permite desfragmentar una tabla, así como actualizar y reordenar los índices. La sintaxis en *MariaDB* es la siguiente:

```
OPTIMIZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE tbl_name [, tbl_name ...];
```

Ejemplo: `OPTIMIZE TABLE conferencia;`

Funciona con tablas que usen los **motores de almacenamiento InnoDB, Aria, MyISAM y ARCHIVE**, y debe usarse si se ha eliminado una gran parte de los registros de una tabla o si se han realizado muchos cambios en una tabla con filas de longitud variable (tablas que tienen columnas VARCHAR, VARBINARY, BLOB o TEXT).

ANALYZE TABLE analiza y almacena la distribución de claves en una tabla. Esta distribución se usa para determinar el orden que el servidor seguirá para combinar tablas en un JOIN, así como para decidir qué índices se usarán en una consulta. Es útil después de insertar una gran cantidad de datos y cuando se crea un nuevo índice.

`ANALYZE [NO_WRITE_TO_BINLOG | LOCAL] TABLE tbl_name [, tbl_name ...];`

Ejemplo: `ANALYZE TABLE conferencia;`

Esta sentencia **funciona con tablas que usen los motores de almacenamiento MyISAM, Aria e InnoDB**.

10.3 Optimización de consultas.

Antes de empezar a cambiar nada, lo primero que se debe hacer es **ejecutar la sentencia EXPLAIN sobre la consulta**. De esta forma se puede saber qué pasos sigue el **SGBD** para realizarla, y de qué manera accede a las tablas en cada uno de ellos. Con esto se podrán identificar posibles cuellos de botella.

La sentencia **EXPLAIN** permitirá obtener información sobre el plan de ejecución de las consultas realizadas contra la BD. De esta forma, se podrá analizar el plan de ejecución para saber cómo optimizar la ejecución de dichas consultas.

La sentencia **EXPLAIN** devuelve una tabla con una serie de filas con información sobre cada una de las tablas empleadas en la consulta a la que acompaña. **EXPLAIN** se puede utilizar con las siguientes sentencias: **SELECT, DELETE, INSERT, REPLACE y UPDATE**. Para emplearla, bastará con poner **EXPLAIN** seguido de la sentencia que se quiera analizar. Por ejemplo:

```
EXPLAIN SELECT * FROM conferencia JOIN sala ON conferencia.sala = sala.nombre;
```

La tabla que devuelve la sentencia **EXPLAIN** lista las tablas en el orden en el que **MariaDB** las leerá procesando la consulta (la primera fila que aparece sería la primera tabla que se lee y la última fila sería la última tabla que sería leída). Este orden es importante conocerlo ya que indicará el plan de ejecución de la consulta y, por tanto, información relevante para realizar las optimizaciones.

De forma adicional, **EXPLAIN** se puede combinar con **EXTENDED** para obtener más información del plan de ejecución. **EXTENDED** proporciona una columna más a la salida producida por **EXPLAIN**, esta columna será la de **filtered** e indicará el porcentaje aproximado de filas que han sido filtradas por la condición empleada en la tabla. Para emplear esta sentencia, bastará con poner **EXPLAIN EXTENDED** seguido de la consulta que se quiere analizar.

asistente (4r x 11c)										
id	select_type	table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	asistente	ALL	PRIMARY	(NULL)	(NULL)	(NULL)	13	100,00	Using where
1	PRIMARY	<subquery2>	eq_ref	distinct_key	distinct_key	24	func	1	100,00	
2	MATERIALIZED	conferencia	ALL	PRIMARY	(NULL)	(NULL)	(NULL)	5	100,00	Using where
2	MATERIALIZED	asistir	ref	PRIMARY,asi_con_FK	asi_con_FK	28	conferencias.conferencia.referencia	4	100,00	Using index

Si se emplea **EXTENDED**, también se puede emplear **SHOW WARNINGS** después de ejecutar la consulta **EXPLAIN**. La sentencia **SHOW WARNINGS** mostrará la consulta reescrita tras la actuación del optimizador. Para emplear esta sentencia, bastará con poner **SHOW WARNINGS** después de ejecutar la consulta **EXPLAIN EXTENDED**. Ejemplo:


```
EXPLAIN EXTENDED SELECT * FROM conferencia;
SHOW WARNINGS;
```

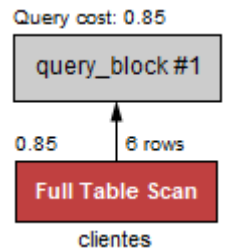
Del resultado de la sentencia **EXPLAIN**, en lo que hay que fijarse es:

- ✓ **El orden de las filas:** es el orden en que la **BD** consulta las tablas.
- ✓ **La columna Key:** indica el índice que se está empleando para acceder a esa tabla concreta (si lo hubiera).

✓ **La columna type:** indica el tipo de acceso a la tabla. Los posibles valores, de mejor a peor, son:

- | | | |
|-----------------|--------------------------|-------------------------|
| • <i>system</i> | • <i>fulltext</i> | • <i>index_subquery</i> |
| • <i>const</i> | • <i>ref_or_null</i> | • <i>range</i> |
| • <i>eq_ref</i> | • <i>index_merge</i> | • <i>index</i> |
| • <i>ref</i> | • <i>unique_subquery</i> | • <i>ALL</i> |

MySQL Workbench incluye Visual Explain, el cual permite ver como un gráfico el plan de acceso para las sentencias de SQL ejecutadas. Una vez ejecutada la consulta desde la barra de herramientas se puede hacer clic sobre el botón  y se mostrará de forma gráfica el plan de acceso (ver imagen del plan de acceso para la sentencia `SELECT * FROM automoviles.clientes;`).



10.4 Optimizaciones.

10.4.1 Evitar FULL SCAN.

Si algún acceso de la consulta es del tipo *index* o *ALL*, se debería revisar esa parte de la consulta, a menos que se quiera expresamente listar o calcular todos los registros de la tabla.

Si al ejecutar la sentencia `EXPLAIN`, se comprueba que el **tipo de acceso a alguna de las tablas es de ALL**, entonces **se trata de un FULL TABLE SCAN**, quiere decir que se están leyendo todos los registros de dicha tabla. Se debería **ver si se puede reescribir la consulta para que se acceda por un índice de la tabla**, o valorar la posibilidad de **crear un índice para las columnas por las que se está buscando**. Esto dependerá de lo frecuente o importante que sea esta consulta en la aplicación.

Si el tipo de acceso es *INDEX* no es mucho mejor, pues **indica que se están leyendo todos los accesos de un índice**. Se le llama **FULL INDEX SCAN** y **no llega a ser tan grave como un FULL TABLE SCAN**, porque se suele hacer en memoria y lee menos información por registro, pero **igualmente se está pasando por todos los nodos**. Este tipo de acceso se emplea **cuando todas las columnas que se quieren obtener forman parte del mismo índice** y, por tanto, **el optimizador de la BD entiende que no necesita ir a las tablas para obtener la información, ya que puede sacarla exclusivamente del índice**.

10.4.2 Uso correcto de los índices.

Hay que **ver los índices como diccionarios, donde el propio índice es la palabra y el registro entero de la tabla es la definición**. Si se quiere buscar la definición de “*escuela*” y se tantea por palabras que comiencen por “*esc*”, no se irá mal encaminado, mientras que si se quiere encontrarla buscando palabras que terminen por “*la*” no quedará más remedio que empezar por la primera página e ir pasando por todas hasta que se dé con la palabra. Esto explica que una búsqueda **LIKE ‘prefijo%’** es indexada y otra **LIKE ‘%sufijo’** no lo es.

Esto que parece una tontería cobra **especial importancia en el caso de los índices compuestos (formados por varias columnas)**, donde el **orden de las columnas que los forman es totalmente determinante**. Con este tipo de índices es mejor usar el ejemplo de la guía telefónica, pensar que no se será muy eficiente buscando el teléfono de alguien de quien sólo se conoce su segundo apellido.

10.4.3 Sentencias OR.

El optimizador de **MariaDB** no puede usar índices si se está empleando la sentencia **OR** y alguna de sus restricciones es una columna no indexada. Por ejemplo:

```
SELECT * FROM mi_tabla
WHERE columna_indexada='valor1' OR columna_no_indexada='valor2';
```

Se debería **tratar de evitar las sentencias OR siempre que sea posible**.

10.4.4 GROUP/ORDER BY.

Se va a tratar la cláusula **GROUP BY**, pero tener en cuenta que todo se aplica también a **ORDER BY**.

Esta cláusula puede suponer un verdadero cuello de botella cuando el número de registros a agrupar es muy elevado (independientemente de que se use la cláusula `LIMIT`, pues esta se aplica después del `GROUP BY`).

Hay que procurar que todas las columnas presentes en el `GROUP BY` formen parte del mismo índice de la tabla que se está consultando, y además en el mismo orden que en la consulta. Si la consulta es muy importante en la aplicación, se puede valorar la posibilidad de definir un índice para optimizarla. Se elegiría primero la/s columna/s filtrada/s en el `WHERE`, y después aquellas presentes en el `GROUP BY`.

10.4.5 Tablas derivadas versus subconsultas.

Una **subconsulta** no es más que una **sentencia `SELECT` dentro de otra sentencia**. Una **tabla derivada** es un tipo concreto de subconsulta que se caracteriza porque está dentro del `FROM` de la consulta “padre”. El tratamiento de ambas es diferente:

- ✓ En una subconsulta se ejecuta una búsqueda en la *BD* por cada registro de la consulta padre.
- ✓ En una tabla derivada se realiza una sola consulta a la *BD*, almacenándose los resultados en una tabla temporal en memoria. A esta tabla se accede una vez por cada registro de la consulta padre.

Con tablas derivadas, **las tablas que se crean en memoria como resultado de las subconsultas de los `INNER JOIN` no están indexadas**. Esto hace que, para cada registro de la consulta padre, deban leerse todas las filas de las tablas derivadas para comprobar cual cumple la condición de igualdad. En una tabla de 300.000 registros supone, en el peor de los casos, 300.000 lecturas en memoria.

Con subconsultas en el `SELECT`, para cada registro de la tabla “padre” se realizan varios accesos indexados a las tablas de las subconsultas. Esto supone, debido a la naturaleza de árbol binario del índice, menos lecturas en cada una de las subconsultas.

10.4.6 `INNER JOIN` con `GROUP/ORDER BY`.

En una consulta con varios `INNER JOIN` es el optimizador quien decide el orden de consulta de las tablas, dependiendo de las restricciones y el uso de los índices. Esta decisión suele ser acertada, con una excepción, puede no ser el orden óptimo para efectuar el `GROUP BY` (o el `ORDER BY`).

En este caso se necesita **colocar primero la tabla sobre la que se está haciendo el `GROUP BY` y forzar al optimizador a que lea primero esa tabla**. De esta manera, el `GROUP BY` se realizará de una sola pasada, debido a que los resultados ya estarán ordenados con respecto al índice. Esto se consigue con `STRAIGHT_JOIN` (no visto en la unidad), que no es más que un `INNER JOIN` con la particularidad de que fuerza la lectura de la tabla de la izquierda antes que la de la derecha.

Con los `LEFT JOIN` no ocurre este problema, pues en este caso la tabla de la izquierda siempre se leerá antes que su tabla dependiente.

10.4.7 La sentencia `EXISTS`.

Suponer que se tienen dos tablas: *BIG* y *HUGE*. La tabla *BIG* tiene miles de registros, y la tabla *HUGE* tiene miles de millones de registros. La relación entre ambas es tal que para cada registro de *BIG* puede haber de cero a varios millones de registros en *HUGE*.

Si se quieren obtener los registros de *BIG* para los cuales se cumplen ciertas condiciones en *HUGE*, puede que la primera aproximación sea realizar un `INNER JOIN`.

Una manera mejor de realizar la consulta sería con una subconsulta que emplee la cláusula `EXISTS`.

¿Cuál es la principal ventaja? Las consultas con `EXISTS` suelen ser muy eficientes, ya que *MariaDB* interrumpe la consulta cuando encuentra la primera coincidencia. Así se evita recorrer todos los registros de *HUGE* que cumplen la condición. Obviamente, esto sólo es aconsejable cuando se está accediendo a la tabla *HUGE* por algún índice.

10.4.8 Conclusiones.

Como se ha podido ver, la primera acción ante una consulta pesada deberá ser analizarla con la sentencia

EXPLAIN. Después del análisis, se intentará que todos los accesos (los que se pueda) se realicen a través de índices. Finalmente, si la consulta sigue siendo lenta, se intentará detectar si el tiempo se está yendo en alguna operación innecesaria o mejorable. Ante todo, no hay mejor estrategia que usar el sentido común.

11 Anexo I. *PIVOT* en *MariaDB*.

En *BD*, muchas veces se tiene estructurada la información en tablas de una forma que no se corresponde con las necesidades, sobre todo a la hora de representarla. Suelen ser tablas históricas con datos que se van guardando una y otra vez en las filas, pero cuando crecen es difícil hacerse una idea de los acumulados por un criterio en concreto.

En estos casos surge la **necesidad de cambiar las filas a columnas** haciendo algún tipo de agrupación para poder gestionar la información de una forma más eficiente. Para hacer esto **se necesita usar una función para poder pivotar dichas tablas**. Es lo que se conoce como hacer un *PIVOT*.

	COL 1	COL 2	COL 3
Fila 1	Identificador	cat1...catN	Dato1
Fila 2	Identificador	cat1...catN	Dato2
Fila 3	Identificador	cat1...catN	Dato3
....
Fila N	Identificador	cat1...catN	Dato4

	COL 1	Cat1	Cat2	...	CatN
Fila 1	Identificador	Dato	Dato	Dato
Fila 2	Identificador	Dato	Dato	Dato
Fila 3	Identificador	Dato	Dato	Dato
....
Fila N	Identificador	Dato	Dato	Dato

Para hacerlo se debe de tener en la tabla una columna categorizada. Cada una de las filas se volverá columna con respecto a dicha categoría de la siguiente forma.

Para ver el funcionamiento de *PIVOT* de una forma más clara se pondrá un ejemplo que servirá de ayuda. Suponer que se tiene una tabla en la que se va guardando un histórico con los pedidos y ventas de una tienda virtual, guardando una fila por cada pedido realizado. En dicha tabla se guardará el nombre del cliente, el mes, la cantidad de productos y el precio total del pedido efectuado.

Las columnas de la tabla pedidos serán:

- ✓ *cliente* → Nombre del cliente. Podría ser un identificador como el *DNI*, pero para el ejemplo será más claro el nombre.
- ✓ *mes* → mes en que se efectuó el pedido.
- ✓ *num_items* → cantidad de productos que tiene el pedido.
- ✓ *precio* → importe total del pedido efectuado.

Como se puede ver, los pedidos están uno por uno en las filas para cada cliente. Es decir, tal y como se han ido haciendo las compras en la página web se han ido insertando en la tabla. La información es correcta y completa, pero la forma de visualizarla no ayuda a ver los totales por cliente.

pedido (36r x 4c)

cliente	mes	num_items	precio
Laura Gil	Febrero	1	10,00
Miguel Gonzalez	Enero	2	4,50
Laura Gil	Enero	1	10,60
Andres Zaragoza	Marzo	6	45,80
Fernando Esteve	Abril	5	2,56
Andres Zaragoza	Enero	3	5,75
Fernando Esteve	Enero	8	7,80
Juan Martin	Febrero	2	82,40
Juan Martin	Enero	2	3,40
Adela Sanchez	Febrero	4	45,56
Andres Zaragoza	Febrero	5	45,70
Fernando Esteve	Febrero	6	52,80
Juan Martin	Marzo	3	34,80
Adela Sanchez	Marzo	5	23,00
Raquel Campos	Enero	4	3,77
Miguel Gonzalez	Marzo	7	45,78
Raquel Campos	Marzo	8	9,99
Adela Sanchez	Enero	5	10,00
Laura Gil	Marzo	21	125,50
Fernando Esteve	Marzo	7	34,70
Juan Martin	Abril	5	90,00
Adela Sanchez	Abril	3	87,12
Raquel Campos	Febrero	2	6,80
Miguel Gonzalez	Abril	4	12,65
Raquel Campos	Abril	8	51,13
Miguel Gonzalez	Febrero	2	2,95
Laura Gil	Abril	5	74,00
Andres Zaragoza	Abril	9	77,30
Juan Martin	Enero	2	3,40
Adela Sanchez	Febrero	3	10,00
Miguel Gonzalez	Marzo	2	4,50
Raquel Campos	Abril	3	3,77
Laura Gil	Febrero	2	10,60

Ahora se va a usar *PIVOT* para transformar filas a columnas y poder ver el resultado de una forma más clara.

MariaDB y forma estándar:

MariaDB no tiene una función específica para cambiar filas y columnas (*CROSSTAB*). Para estos casos se puede hacer el *PIVOT* manualmente con un *SELECT*. En dicho *SELECT* se tendrán que poner a mano todos los acumulados de las columnas que se quieran que salgan (las categorías que estaban en las filas), se hace la distinción de las categorías gracias a *IF* o *CASE WHEN*, dependiendo de la *BD* en la que se esté.

Ahora se verá un ejemplo de cómo pivotar en *MariaDB*. Crear la tabla e insertar los registros:

```
CREATE TABLE pedido (
  cliente VARCHAR(100) NOT NULL,
  mes VARCHAR(20) NOT NULL,
  num_items SMALLINT UNSIGNED DEFAULT 0,
  precio DECIMAL(8,2)
);
INSERT INTO pedido (cliente, mes, num_items, precio) VALUES ('Laura Gil','Febrero',1,10.00),
('Miguel Gonzalez','Enero',2,4.50), ('Laura Gil','Enero',1,10.60), ('Andrés Zaragoza','Marzo',6,45.80),
('Fernando Esteve','Abril',5,2.56), ('Andrés Zaragoza','Enero',3,5.75), ('Fernando Esteve','Enero',8,7.80),
('Juan Martin','Febrero',2,82.40), ('Juan Martin','Enero',2,3.40), ('Adela Sanchez','Febrero',4,45.56),
('Andrés Zaragoza','Febrero',5,45.70), ('Fernando Esteve','Febrero',6,52.80),
```

```
(('Juan Martin','Marzo',3,34.80), ('Adela Sanchez','Marzo',5,23.00), ('Raquel Campos','Enero',4,3.77),
('Miguel Gonzalez','Marzo',7,45.78), ('Raquel Campos','Marzo',8,9.99), ('Adela Sanchez','Enero',5,10.00),
('Laura Gil','Marzo',21,125.50), ('Fernando Esteve','Marzo',7,34.70), ('Juan Martin','Abril',5,90.00),
('Adela Sanchez','Abril',3,87.12), ('Raquel Campos','Febrero',2,6.80), ('Miguel Gonzalez','Abril',4,12.65),
('Raquel Campos','Abril',8,51.13), ('Miguel Gonzalez','Febrero',2,2.95), ('Laura Gil','Abril',5,74.00),
('Andrés Zaragoza','Abril',9,77.30), ('Juan Martin','Enero',2,3.40), ('Adela Sanchez','Febrero',3,10.00),
('Miguel Gonzalez','Marzo',2,4.50), ('Raquel Campos','Abril',3,3.77), ('Laura Gil','Febrero',2,10.60),
('Andrés Zaragoza','Marzo',3,5.75), ('Fernando Esteve','Febrero',2,7.80),
('Miguel Gonzalez','Marzo',7,14.50);
```

Para hacer el *PIVOT* manualmente se ponen como columnas los criterios que se quieran de una columna en concreto. Se hace el sumatorio y se usa la cláusula *CASE WHEN*. Ejemplos:

-- Obtener para cada cliente el número de pedidos que ha realizado en los 4 primeros meses del año: pedidos por mes

```
SELECT cliente,
    SUM(CASE WHEN mes='Enero' THEN 1 ELSE 0 END) enero,
    SUM(CASE WHEN mes='Febrero' THEN 1 ELSE 0 END) febrero,
    SUM(CASE WHEN mes='Marzo' THEN 1 ELSE 0 END) marzo,
    SUM(CASE WHEN mes='Abril' THEN 1 ELSE 0 END) abril
FROM pedido
GROUP BY cliente
ORDER BY cliente;
-- o utilizando COUNT():
SELECT cliente,
    COUNT(CASE WHEN mes='Enero' THEN 1 ELSE NULL END) enero,
    COUNT(CASE WHEN mes='Febrero' THEN 1 ELSE NULL END) febrero,
    COUNT(CASE WHEN mes='Marzo' THEN 1 ELSE NULL END) marzo,
    COUNT(CASE WHEN mes='Abril' THEN 1 ELSE NULL END) abril
FROM pedido
GROUP BY cliente
ORDER BY cliente;
-- Obtener para cada cliente el dinero gastado por meses
SELECT cliente,
    SUM(CASE WHEN mes='Enero' THEN precio ELSE 0 END) enero,
    SUM(CASE WHEN mes='Febrero' THEN precio ELSE 0 END) febrero,
    SUM(CASE WHEN mes='Marzo' THEN precio ELSE 0 END) marzo,
    SUM(CASE WHEN mes='Abril' THEN precio ELSE 0 END) abril
FROM pedido
GROUP BY cliente
ORDER BY cliente;
```

pedido (7r x 5c)				
cliente	enero	febrero	marzo	abril
Adela Sanchez	1	2	1	1
Andres Zaragoza	1	1	2	1
Fernando Esteve	1	2	1	1
Juan Martin	2	1	1	1
Laura Gil	1	2	1	1
Miguel Gonzalez	1	1	3	1
Raquel Campos	1	1	1	2

pedido (7r x 5c)				
cliente	enero	febrero	marzo	abril
Adela Sanchez	10,00	55,56	23,00	87,12
Andres Zaragoza	5,75	45,70	51,55	77,30
Fernando Esteve	7,80	60,60	34,70	2,56
Juan Martin	6,80	82,40	34,80	90,00
Laura Gil	10,60	20,60	125,50	74,00
Miguel Gonzalez	4,50	2,95	64,78	12,65
Raquel Campos	3,77	6,80	9,99	54,90

PIVOT utilizando la cláusula *IF*. Ejemplos:

-- Obtener para cada cliente el número de pedidos que ha realizado en los 4 primeros meses del año: pedidos por mes

```
SELECT cliente,
    SUM(IF(mes='Enero',1,0)) enero,
    SUM(IF(mes='Febrero',1,0)) febrero,
    SUM(IF(mes='Marzo',1,0)) marzo,
    SUM(IF(mes='Abril',1,0)) abril
FROM pedido
GROUP BY cliente
ORDER BY cliente;
-- o utilizando COUNT():
SELECT cliente,
    COUNT(IF(mes='Enero',1,NULL)) enero,
    COUNT(IF(mes='Febrero',1,NULL)) febrero,
    COUNT(IF(mes='Marzo',1,NULL)) marzo,
    COUNT(IF(mes='Abril',1,NULL)) abril
FROM pedido
GROUP BY cliente
ORDER BY cliente;
```

-- Obtener para cada cliente el precio total de sus pedidos por mes: dinero gastado por meses

```
SELECT cliente,
    SUM(IF(mes='Enero',precio,0)) enero,
    SUM(IF(mes='Febrero',precio,0)) febrero,
    SUM(IF(mes='Marzo',precio,0)) marzo,
    SUM(IF(mes='Abril',precio,0)) abril
FROM pedido GROUP BY cliente ORDER BY cliente;
```

12 Anexo II. Búsquedas de texto completo (*full-text*).

MariaDB soporta indexación y búsqueda *full-text*. Un índice *full-text* en *MariaDB* es un índice de tipo *FULLTEXT*. Los índices *FULLTEXT* pueden usarse sólo con tablas *MyISAM*; pueden ser creados desde columnas *CHAR*, *VARCHAR*, o *TEXT* como parte de un comando *CREATE TABLE* o añadido posteriormente usando *ALTER TABLE* o *CREATE INDEX*. Para conjuntos de datos grandes, es mucho más rápido cargar los datos en una tabla que no tenga índice

FULLTEXT y crear el índice posteriormente, que cargar los datos en una tabla que tenga un índice *FULLTEXT* existente.

Las búsquedas *full-text* se realizan con la función **MATCH()**.

MATCH (col1,col2,...) AGAINST (expr [search_modifier])

Ejemplo:

```
CREATE TABLE articles (
  id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
  title VARCHAR(200),
  body TEXT,
  FULLTEXT (title,body),
  FULLTEXT (title,body)
) ENGINE=MyISAM;

INSERT INTO articles (title,body) VALUES
('MySQL Tutorial','DBMS stands for DataBase ...'),
('How To Use MySQL Well','After you went through a ...'),
('Optimizing MySQL','In this tutorial we will show ...'),
('1001 MySQL Tricks','1. Never run mysqld as root. 2. ...'),
('MySQL vs. YourSQL','In the following database comparison ...'),
('MySQL Security','When configured properly, MySQL ...');

SELECT * FROM articles
WHERE MATCH (title, body) AGAINST ('database');

SELECT *, MATCH (title, body) AGAINST ('database') 'Puntuación'
FROM articles
WHERE MATCH (title, body) AGAINST ('database');
```

id	title	body
5	MySQL vs. YourSQL	In the following database comparison ...
1	MySQL Tutorial	DBMS stands for DataBase ...

id	title	body	Puntuación
5	MySQL vs. YourSQL	In the following database comparison ...	0,6626645922660828
1	MySQL Tutorial	DBMS stands for DataBase ...	0,6554583311080933

La función **MATCH** realiza una búsqueda de lenguaje natural para cadenas contra una colección de textos. Una colección es un conjunto de una o más columnas incluidas en un índice *FULLTEXT*. La cadena de búsqueda se da como argumento para **AGAINST**. Para cada registro en la tabla **MATCH** retorna una medida de similitud entre la cadena de búsqueda y el texto en el registro en las columnas mencionadas en la lista **MATCH**.

Por defecto, la búsqueda se realiza de forma insensible a mayúsculas. Sin embargo, puede realizar búsquedas sensibles a mayúsculas usando colaciones binarias para columnas indexadas.

Cuando se usa **MATCH** en una cláusula **WHERE**, como en los ejemplos anteriores, los registros devueltos se ordenan automáticamente con la similitud mayor primero. Los valores son números en coma flotante no negativos. El valor cero significa que no tiene similitud.

Las palabras de menos de 4 letras quedan excluidas de la búsqueda y si los términos de búsqueda están presentes en más del 50% de los resultados éstos no se mostrarán (parámetros definidos en el servidor).

Para búsquedas *full-text*, se requiere que las columnas nombradas en la función **MATCH** sean las mismas columnas incluidas en algún índice *FULLTEXT* de la tabla.

MariaDB permite realizar **búsquedas full-text booleanas** usando el modificador **IN BOOLEAN MODE**:

```
SELECT *
FROM articles
WHERE MATCH (title,body) AGAINST ('+MySQL -YourSQL' IN BOOLEAN MODE);
```

id	title	body
1	MySQL Tutorial	DBMS stands for DataBase ...
2	How To Use MySQL Well	After you went through a ...
3	Optimizing MySQL	In this tutorial we will show ...
4	1001 MySQL Tricks	1. Never run mysqld as root. 2. ...
6	MySQL Security	When configured properly, MySQL ...

Esta consulta recibe todos los registros que contienen la palabra “MySQL” pero que no contiene la palabra “YourSQL”. Tener en cuenta que no ordena los registros automáticamente por similitud.

Se puede comprobar que se puede buscar obligando a que estén todas las palabras indicadas indiferentemente del orden en el que puedan aparecer (siempre que su longitud sea mayor a 4 caracteres):

```
SELECT *
FROM articles
WHERE MATCH (title) AGAINST ('+YourSQL +mysql' IN BOOLEAN MODE);
```

id	title	body
5	MySQL vs. YourSQL	In the following database comparison ...

O bien localizar los registros en los que se encuentre alguna de las palabras indicadas, sin obligar a que se encuentren todas:

```
SELECT *
FROM articles
WHERE MATCH (title) AGAINST ('YourSQL mysql' IN BOOLEAN MODE);
```

id	title	body
1	MySQL Tutorial	DBMS stands for DataBase ...
2	How To Use MySQL Well	After you went through a ...
3	Optimizing MySQL	In this tutorial we will show ...
4	1001 MySQL Tricks	1. Never run mysqld as root. 2. ...
5	MySQL vs. YourSQL	In the following database comparison ...
6	MySQL Security	When configured properly, MySQL ...

Algunos de los **operadores que se pueden aplicar en las búsquedas booleanas** son:

- ✓ ***Sin operador*** indica que la palabra es opcional.
- ✓ El signo **+** indica que la palabra debe estar presente en cada resultado.
- ✓ El signo **-** indica que la palabra no debe estar presente en ningún resultado.
- ✓ El signo **~** indica que la palabra perderá peso en la relevancia de los resultados.
- ✓ El asterisco ***** sirve para buscar las palabras que comiencen por el término de búsqueda.
- ✓ Las comillas **""** devolverán las coincidencias exactas con el texto que contienen.
- ✓ **()** se usan para agrupar palabras en subexpresiones.

Los siguientes **ejemplos** muestran algunas cadenas de búsqueda que usan operadores booleanos *full-text*:

- ✓ **'apple banana'**: encuentra registros que contengan al menos una de las dos palabras.
- ✓ **'+apple +juice'**: encuentra registros que contengan ambas palabras.
- ✓ **'+apple macintosh'**: encuentra registros que contengan la palabra “apple”, pero clasifica mejor las que también contengan “macintosh”.
- ✓ **'+apple -macintosh'**: encuentra registros que contengan la palabra “apple” pero no “macintosh”.
- ✓ **'+apple +(>turnover <strudel)'**: encuentra registros que contengan las palabras “apple” y “turnover”, o “apple” y “strudel” (en cualquier orden), pero clasifican “apple turnover” mejor que “apple strudel”.
- ✓ **'apple*'**: encuentra registros que contenga palabras tales como “apple”, “apples”, “applesauce”, o “applet”.
- ✓ **'"some words"'**: encuentra registros que contienen la frase exacta “some words” (por ejemplo, registros que contengan “some words of wisdom” pero no “some noise words”). Tener en cuenta que el carácter **""** que envuelve la frase son caracteres operadores que delimitan la frase.

La búsqueda *full-text* soporta **expansión de consultas** (en particular, su variante “expansión de consultas ciega”). Generalmente esto es útil cuando una frase buscada es demasiado corta, lo que a menudo significa que el usuario se fía del conocimiento implícito que normalmente no tiene el motor de búsqueda *full-text*. Por ejemplo, un usuario buscando “database” puede referirse a “MySQL”, “Oracle”, “DB2”, y “RDBMS” todas son frases que deberían coincidir con “databases” y deberían retornarse también. Este es conocimiento implícito.

La expansión de consultas ciega (también conocida como *feedback* de relevancia automático) se activa añadiendo **WITH QUERY EXPANSION** siguiendo la frase de búsqueda. Funciona realizando la búsqueda dos veces, donde la frase de búsqueda para la segunda búsqueda es la frase de búsqueda original concatenada con los primeros documentos encontrados en la primera búsqueda. Por lo tanto, si uno de estos documentos contiene la palabra “databases” y la palabra “MySQL”, la segunda búsqueda los documentos que contienen la palabra “MySQL” incluso si no contienen la palabra “database”. El siguiente ejemplo muestra esta diferencia:

```
SELECT * FROM articles
WHERE MATCH (title,body) AGAINST ('database');
```

```
SELECT * FROM articles
WHERE MATCH (title,body)
AGAINST ('database' WITH QUERY EXPANSION);
```

articles (2r × 3c)		
id	title	body
5	MySQL vs. YourSQL	In the following database comparison ...
1	MySQL Tutorial	DBMS stands for DataBase ...

articles (3r × 3c)		
id	title	body
1	MySQL Tutorial	DBMS stands for DataBase ...
5	MySQL vs. YourSQL	In the following database comparison ...
3	Optimizing MySQL	In this tutorial we will show ...

Más información sobre los índices FULL-TEXT en <https://mariadb.com/kb/en/full-text-index-overview/>