Creando nuestra primera base

En esta clase trabajaremos creando una base de datos y manipulando sus datos. Los objetivos de la misma serán:

Ganar experiencia en MySQL Workbench.

Crear una base de datos/Schema.

Crear tablas con sus columnas.

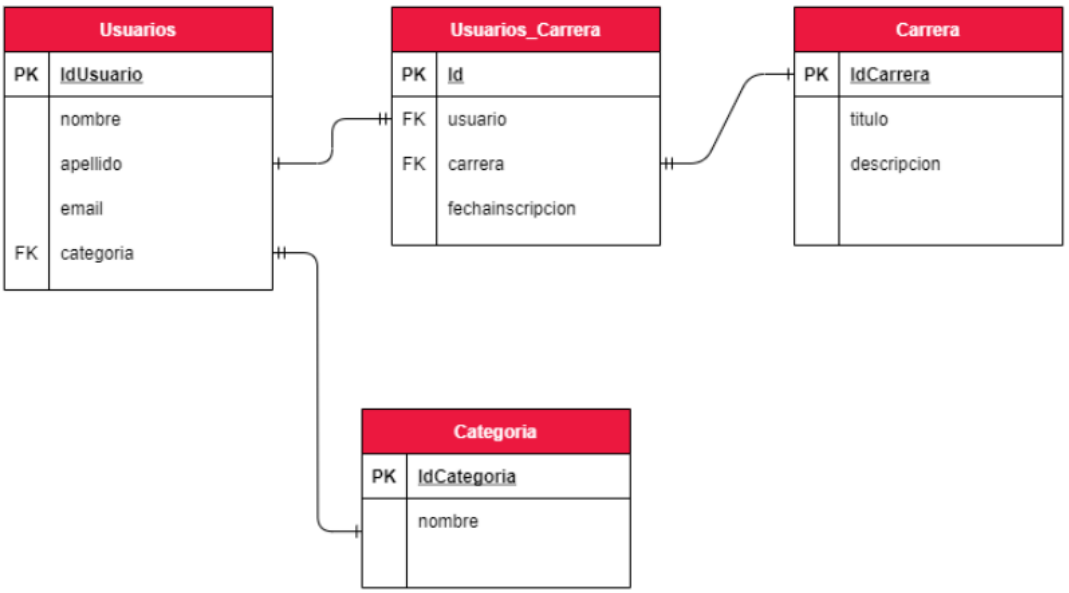
Hacer consultas que permitan insertar, actualizar y eliminar registros de una tabla.

Tus docentes van a realizar un paso a paso de ejemplo para que después te puedas guiar al realizar el desafío grupal.

¡Manos a la obra!

Tomando como base el ejercicio de DER Playground realizado, vamos a realizar el paso a paso para **crear una parte de la base de datos**:

DER - Playground

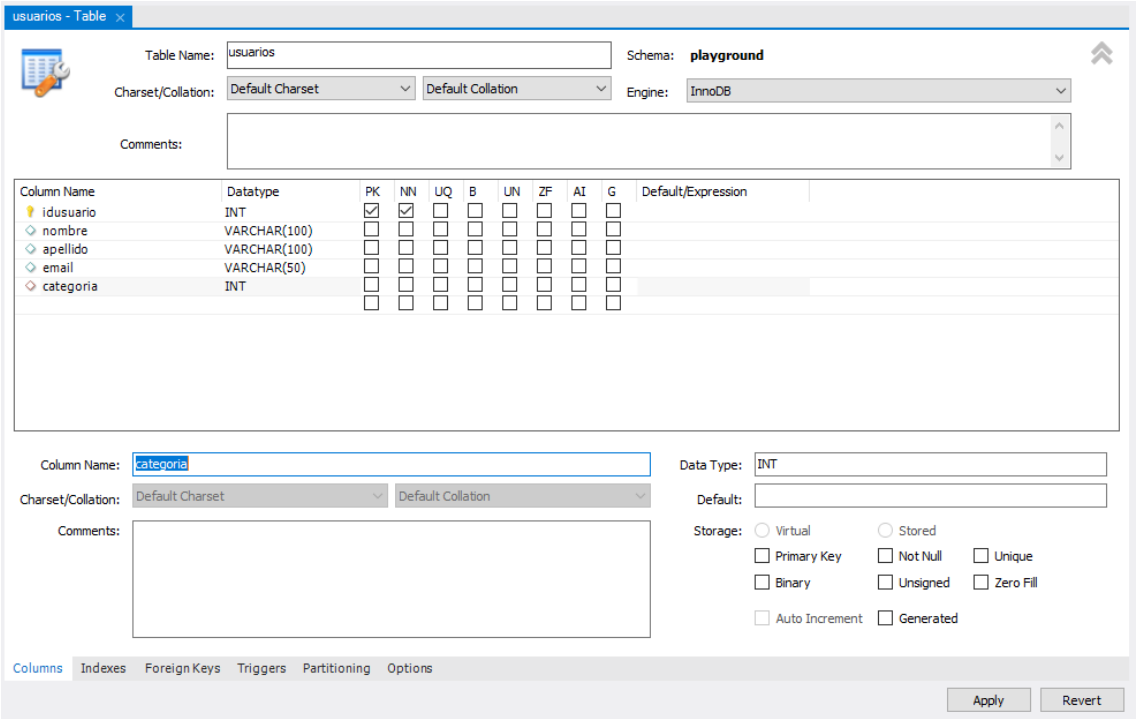


Ejemplo CREATE TABLE "categorias"

SQL

```sql
CREATE TABLE `playground`.`categorias` (
  `idcategoria` INT NOT NULL,
  `nombre` VARCHAR(100) NULL,
  PRIMARY KEY (`idcategoria`));
```

Ejemplo CREATE TABLE "usuarios"



SQL

```sql
CREATE TABLE `playground`.`usuarios` (
  `idusuario` INT NOT NULL,
  `nombre` VARCHAR(100) NULL,
  `apellido` VARCHAR(100) NULL,
  `email` VARCHAR(50) NULL,
  `categoria` INT NULL,
  PRIMARY KEY (`idusuario`),
  INDEX `FKcategoria_idx` (`categoria` ASC) VISIBLE,
  CONSTRAINT `FKcategoria`
    FOREIGN KEY (`categoria`)
    REFERENCES `playground`.`categorias` (`idcategoria`)
  );
```

Ejemplo CREATE TABLE “carrera”

```sql
CREATE TABLE `playground`.`carrera` (
  `idcarrera` INT NOT NULL,
  `titulo` VARCHAR(45) NULL,
  `descripcion` VARCHAR(100) NULL,
  PRIMARY KEY (`idcarrera`));
```

Ejemplo CREATE TABLE “usuarios_carrera”

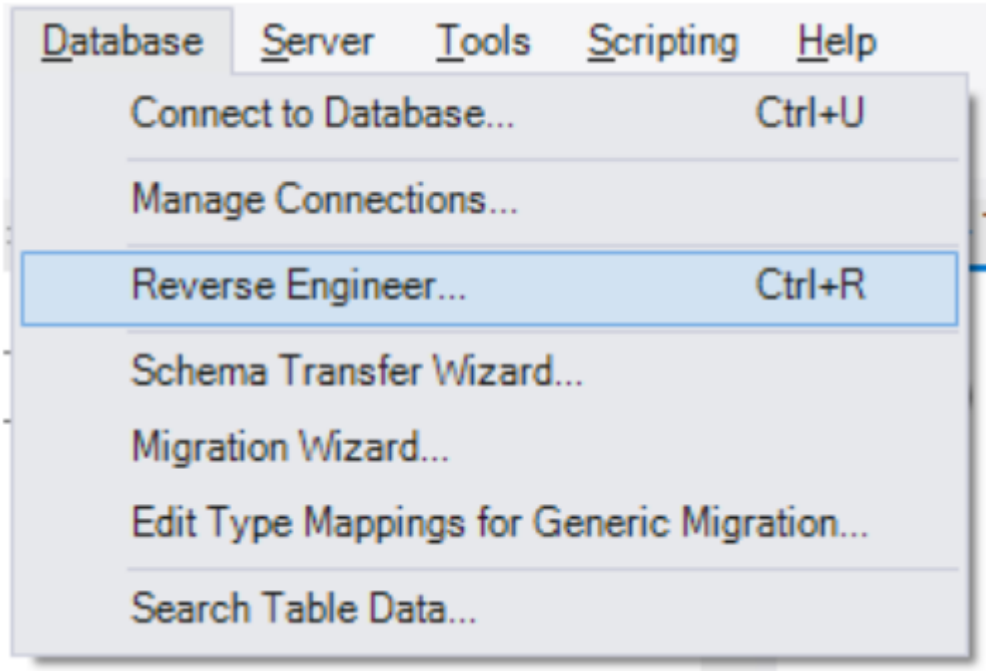
```sql
CREATE TABLE `playground`.`usuarios_carrera` (
  `id` INT NOT NULL,
  `usuario` INT NULL,
  `carrera` INT NULL,
  `fechainscripcion` DATE NULL,
  PRIMARY KEY (`id`),
  INDEX `usuario_idx` (`usuario` ASC) VISIBLE,
  INDEX `carrera_idx` (`carrera` ASC) VISIBLE,
  CONSTRAINT `usuario`
    FOREIGN KEY (`usuario`)
    REFERENCES `playground`.`usuarios` (`idusuario`),
  CONSTRAINT `carrera`
    FOREIGN KEY (`carrera`)
    REFERENCES `playground`.`carrera` (`idcarrera`));
```

Validar el modelo creado

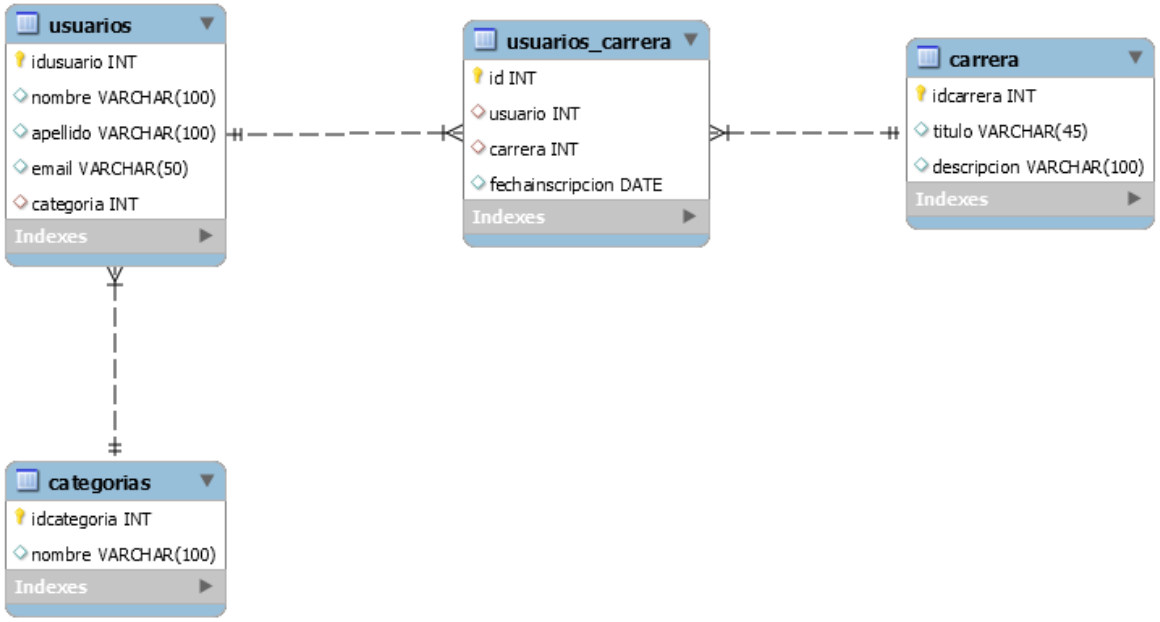
Continuamos con las tablas de **carrera** y de **Usuarios_Carrera** con las claves foráneas a las tablas de **usuarios** y **carrera**.

Luego, podemos realizar ingeniería inversa para controlar

que el modelo relacional es el correcto.



WORKBENCH - DER - Playground



Insertar datos

Vamos a insertar datos en las tablas:

- Categorías: “Alumno”, “Docente”, “Editor” y

“Administrador”.

- Usuario: “Juan Perez jperez@gmail.com categoria alumno”.
- Carrera: “Certified Tech Developer Carrera de programación y desarrollo de productos digitales”.
- Juan se inscribió el 1/3/2021 a CTD.

```sql
INSERT INTO `playground`.`categorias`
(`idcategoria`, `nombre`)
VALUES
(1, "Alumno"),
(2, "Docente"),
(3, "Editor"),
(4, "Administrador");
```

Desde la parte de administración nos avisan que no existe más la categoría “Editor”.

```sql
DELETE FROM `playground`.`categorias`
WHERE nombre = "Editor";
```

Ahora, ¿Qué sucede si intentamos eliminar la categoría “Alumno”?

```sql
DELETE FROM `playground`.`categorias`
WHERE nombre = "Alumno";
```

Nos va a generar un error similar a *"Cannot delete or update a parent row: a foreign key constraint fails"*.
Esto indica que no se puede eliminar la categoría "Alumno" dado que debe haber algún alumno bajo esta categoría.

Iniciemos con SQL

Bienvenidos a un nuevo módulo de Base de Datos.

En este módulo seguimos adentrándonos en el mundo de las bases de datos relacionales y, en particular, aprenderemos a:

Crear bases de datos.

Hacer consultas que nos permitan insertar, actualizar y eliminar registros de una tabla.

Traer datos de una tabla de manera general o aplicando diversos filtros de búsqueda.

Limitar la cantidad de resultados generados por una determinada búsqueda.

Usar algunas funciones nativas de MySQL para que podamos filtrar y organizar los datos obtenidos de una manera mucho más eficiente.

Relacionar distintas tablas.

Hacer foco en cómo obtener reportes o informes específicos.

En conclusión, es un módulo en donde vamos a conocer los principales secretos del trabajo con bases de datos. Así que manos a la obra y a poner mucha atención en los conceptos aquí expuestos.

¿Qué bases de datos vamos a utilizar?

Para este nuevo módulo vamos a aprender el **lenguaje de consultas de SQL.**

Este lenguaje posee un conjunto de sentencias que iremos aprendiendo gradualmente y sumaremos complejidad clase a clase.

Las sentencias SQL se agrupan en dos categorías según su funcionalidad o propósito:

Lenguaje de definición de datos (DDL): son sentencias para la creación de tablas y registros. Es decir, se utilizan para realizar modificaciones sobre la estructura de la base de datos.

Lenguaje de manipulación de datos (DML): son sentencias para la consulta, actualización y borrado de datos. Es decir, se utilizan para realizar consultas y modificaciones sobre los registros almacenados dentro de cada una de las tablas.

En esta clase vamos a aprender sobre el primer grupo que nos permitirá implementar el DER (diagrama entidad-relación) dentro de una base de datos.

¿Cómo vamos a trabajar?

Tendremos que descargar una base de datos que nos va a servir de ejemplo para realizar las prácticas virtuales. Esta base la encontrarás para descargar en el link de abajo.

En cada cuestionario propondremos una consigna que implica trabajar con Workbench y la base de datos descargada, para luego copiar y pegar los resultados que obtengamos en los cuestionarios de Playground.

Create, drop, alter

¿Cómo creamos una tabla con sentencias de SQL? Y si la queremos eliminar, ¿cómo lo podemos hacer? Además, ¿será posible modificar una tabla ya existente?

Las directrices **create, drop y alter** nos van a permitir llevar a cabo cada una de estas acciones que, vale la pena mencionar, son bastante habituales dentro del proceso de trabajo con bases de datos.

Sin más, démosle una vista al siguiente video para que conocer mucho más al respecto.

Comando CREATE DATABASE

Con **CREATE DATABASE** podemos crear una base de datos desde cero.

```sql
CREATE DATABASE miprimerabasededatos;
USE miprimerabasededatos;
```

Comando CREATE TABLE

Con **CREATE TABLE** podemos crear una tabla desde cero, junto con sus columnas, tipos y constraints.

```sql
CREATE TABLE nombre_de_la_tabla (
    nombre_de_la_columna_1 TIPO_DE_DATO CONSTRAINT,
    nombre_de_la_columna_2 TIPO_DE_DATO CONSTRAINT
)
```

```sql
CREATE TABLE post (
    id INT PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200)
)
```

Ejemplo CREATE TABLE

```sql
CREATE TABLE peliculas (
    id  INT PRIMARY KEY AUTO_INCREMENT,
    title VARCHAR(500) NOT NULL,
    rating DECIMAL(3,1) NOT NULL,
    awards INT DEFAULT 0,
    release_date DATE NOT NULL,
    length INT NOT NULL
);
```

FOREIGN KEY

Cuando creemos una columna que contenga una id foránea, será necesario usar la sentencia **FOREIGN KEY** para aclarar a qué tabla y a qué columna hace referencia aquel dato.

Es importante remarcar que la tabla "**clientes**" deberá existir antes de correr esta sentencia para crear la tabla "ordenes".

```sql
CREATE TABLE ordenes (
    orden_id INT NOT NULL,
    orden_numero INT NOT NULL,
    cliente_id INT,
    PRIMARY KEY (orden_id),
    FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);
```

Comando DROP TABLE

DROP TABLE borrará la tabla que le especifiquemos en la sentencia.

```sql
DROP TABLE IF EXIST peliculas;
```

Comando ALTER TABLE

ALTER TABLE permite alterar una tabla ya existente y va a operar con tres comandos:

- **ADD**: para agregar una columna.
- **MODIFY**: para modificar una columna.
- **DROP**: para borrar una columna.

Ejemplos ALTER TABLE

```sql
ALTER TABLE peliculas
ADD rating DECIMAL(3,1) NOT NULL;
```

Agrega la columna **rating**, aclarando tipo de dato y constraint.

```sql
ALTER TABLE peliculas
MODIFY rating DECIMAL(4,1) NOT NULL;
```

Modifica el decimal de la columna **rating**. Aunque el resto de las configuraciones de la tabla no se modifiquen, es necesario escribirlas en la sentencia.

```sql
ALTER TABLE peliculas
DROP rating;
```

Borra la columna **rating**.

Insert, update, delete

Al momento de trabajar con tablas, indefectiblemente vamos a querer insertar, actualizar o eliminar registros. Estas tres funciones son llevadas a cabo gracias a las tres directrices principales que tiene SQL para esta finalidad.

Insert nos va a permitir agregar datos, con **delete** podremos borrarlos y con **update** podremos modificar los registros existentes en una tabla.

Vamos entonces a conocer cómo podemos trabajar con ellos.

INSERT

Existen dos formas de agregar datos en una tabla:

- Insertando datos en **todas** las **columnas**.
- Insertando datos en las **columnas** que **especifiquemos**.

Todas las columnas

Si estamos insertando datos en todas las columnas, no hace falta aclarar los nombres de cada columna. Sin embargo, el orden en el que insertemos los valores, deberá ser el mismo orden que tengan asignadas las columnas en la tabla.

```sql
INSERT INTO table_name (columna_1, columna_2, columna_3, ...)
VALUES (valor_1, valor_2, valor_3, ...);
```

```sql
INSERT INTO artistas (id, nombre, rating)
VALUES (DEFAULT, 'Shakira', 1.0);
```

Columnas específicas

Para insertar datos en una columna en específico, aclaramos la tabla y luego escribimos el nombre de la o las columnas entre los paréntesis.

```sql
INSERT INTO artistas (nombre)
VALUES ('Calle 13');
```

```sql
INSERT INTO artistas (nombre, rating)
VALUES ('Maluma', 1.0);
```

UPDATE

UPDATE modificará los registros existentes de una tabla. Al igual que con **DELETE**, es importante no olvidar el **WHERE** cuando escribimos la sentencia, aclarando la condición.

```sql
UPDATE nombre_tabla
SET columna_1 = valor_1, columna_2 = valor_2, ...
WHERE condición;
```

```sql
UPDATE artistas
SET nombre = 'Charly Garcia', rating = 1.0
WHERE id = 1;
```

DELETE

Con **DELETE** podemos borrar información de una tabla. Es importante recordar utilizar siempre el **WHERE** en la

sentencia para agregar la condición de cuáles son las filas que queremos eliminar. Si no escribimos el **WHERE**, estaríamos borrando **toda** la **tabla** y no un registro en particular

```sql
DELETE FROM nombre_tabla WHERE condición;
```

```sql
DELETE FROM artistas WHERE id = 4;
```

SELECT

Cómo usarlo

Toda consulta a la base de datos va a empezar con la palabra **SELECT**.

Su funcionalidad es la de realizar consultas sobre **una** o **varias columnas** de una tabla.

Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

```sql
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

Ejemplo - Tabla Peliculas

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

De esta tabla completa, para conocer solamente los títulos y ratings de las películas guardadas en la tabla **películas,** podríamos hacerlo ejecutando la siguiente consulta:

```sql
SELECT id, titulo, rating
FROM peliculas;
```

WHERE

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT** que se realizan a una base de datos.

```sql
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;
```

Teniendo una tabla **clientes**, podría consultar primer nombre y apellido, filtrando con un **WHERE** solamente los usuarios **que su país es igual a Argentina** de la siguiente manera:

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE pais = 'Argentina';
```

Operadores

=	Igual a	IS NULL	Es nulo
>	Mayor que	BETWEEN	Entre dos valores
>=	Mayor o igual que	IN	Lista de valores
<	Menor que	LIKE	Se ajusta a...
<=	Menor o igual que		
<>	Diferente a		
!=	Diferente a		

Queries de ejemplo

SQL

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE pais <> 'Argentina';
```

SQL

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE id < 15;
```

SQL

```sql
SELECT primer_nombre, apellido
FROM clientes
WHERE id > 5;
```

```sql
SELECT *
FROM canciones
      WHERE id >= 3
      AND id < 8;
```

```sql
SELECT *
FROM canciones
      WHERE id = 2
      OR id = 6;
```

```sql
DELETE FROM usuarios
WHERE id = 2;
```

Si en esta query quitáramos el WHERE...
¡Borraríamos toda la tabla!

ORDER BY

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente *(ASC)* según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

```sql
SELECT nombre_columna1, nombre_columna2
FROM tabla
WHERE condicion
ORDER BY nombre_columna1;
```

Query de ejemplo

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un **WHERE** solamente los usuarios **mayores de 21 años** y ordenarlos de forma descendente tomando como referencia la columna nombre.

```sql
SELECT nombre, rating
FROM artistas
WHERE rating > 1.0
ORDER BY nombre DESC;
```

¿Qué es un CRUD?

CRUD es el acrónimo de "Crear, Leer, Actualizar y Borrar" (Claramente que en inglés, Create, Read, Update and Delete).

Este acrónimo se utiliza mucho en bases de datos, y técnicamente lo vemos en forma de sentencias SQL.

En esta nueva clase, vamos a consolidar lo aprendido sobre:

- Creación de bases de datos.

Manipulación de datos.

Generación de consultas.

Cabe aclarar que esta clase es el primer **CHECK POINT** de nuestro recorrido. ¿Nos detenemos y repasamos lo aprendido?

Antes de continuar te recomendamos repasar las sentencias SQL aprendidas. Principalmente los videos de Spoiler que te propusimos: SELECT y WHERE y ORDER BY.

Between y Like

Las consultas a las bases de datos, tradicionalmente, están compuestas de varios filtros.

¿Qué pasaría si queremos traer las facturas emitidas del 2010? Seguramente vamos a necesitar buscar las mismas por fecha en la base de datos.

Y si queremos saber ¿qué películas comienzan con la palabra "Toy"? Seguramente vamos a necesitar consultar

textos en la base de datos.

Las directrices **Between y Like** son fundamentales para poder poder hacer este tipo de consultas y más. Veamos entonces cómo se pueden llevar a cabo.

BETWEEN

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- BETWEEN **incluye** los **extremos**.
- BETWEEN funciona con **números**, **textos** y **fechas**.
- Se usa como un filtro de un WHERE.

Por ejemplo, coloquialmente:

- Dados los números: 4, 7, 2, 9, 1

Si hiciéramos un BETWEEN entre 2 y 7 devolvería 4, 7, 2 *(excluye el 9 y el 1, e incluye el 2).*

Query de ejemplo

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** solo cuando las edades estén **entre** 6 y 12.

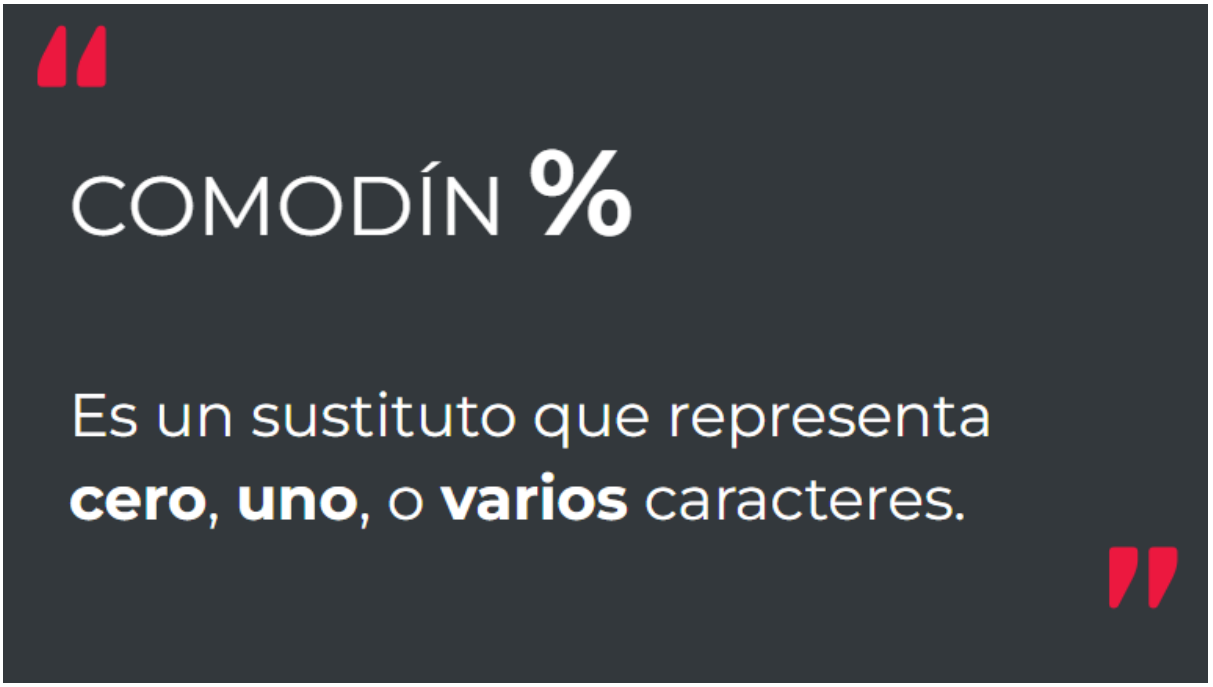
```sql
SELECT nombre, edad
FROM alumnos
WHERE edad BETWEEN 6 AND 12;
```

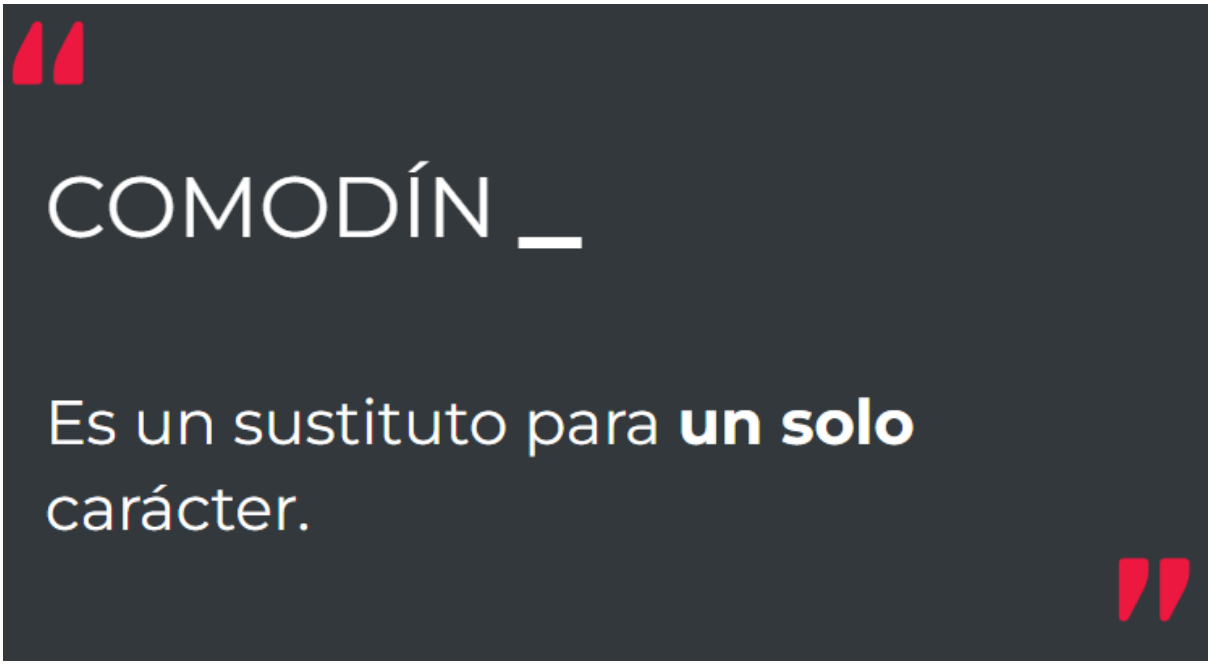
LIKE

Cuando hacemos un filtro con un **WHERE**, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** *(wildcards)*.

Por ejemplo, podríamos querer buscar:

- ➔ Los nombres que tengan la letra 'a' como segundo carácter.
- ➔ Las direcciones postales que incluyan la calle 'Monroe'.
- ➔ Los clientes que empiecen con 'Los' y terminen con 's'.





Queries de ejemplo

```sql
SELECT nombre
FROM usuarios
WHERE nombre LIKE '_a%';
```

Devuelve aquellos nombres que tengan la letra 'a' como segundo carácter.

```sql
SELECT nombre
FROM usuarios
WHERE direccion LIKE '%Monroe%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

```sql
SELECT nombre
FROM clientes
WHERE nombre LIKE 'Los%s';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

Filtros

Consolidemos lo aprendido sobre:

Select

Where

Order by

Like

Between

AND/OR

Limit y Offset

Imaginemos que nos solicitan hacer el buscador de la aplicación en la que estamos trabajando y, cuando realizamos la respectiva consulta, se muestran los resultados de búsqueda y son cientos y cientos de filas, pero nuestro Cliente nos solicitó que los resultados se mostrarán de 20 en 20.

¿Cómo podemos hacer entonces para que la consulta nos traiga de a 20 filas? Y más importante aún, ¿cómo podemos hacer para pedir las 20 siguientes en cada paginación?

Bien. Las directrices LIMIT y OFFSET son las herramientas necesarias para llevar a cabo tal fin. Así que no perdamos más tiempo y veamos cómo se implementan.

Limit

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

```sql
SELECT nombre_columna1, nombre_columna2
FROM nombre_tabla
LIMIT cantidad_de_registros;
```

Query de ejemplo

Teniendo una tabla **peliculas**, podríamos armar un top 10 con las películas que tengan más de 4 premios usando un **LIMIT** en la siguiente consulta:

```sql
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

Offset

- En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20.
- *¿Pero cómo haríamos si quisiéramos recuperar sólo 20 películas pero salteando las primeras 10 de la tabla?*

➔ **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

{código}

```
SELECT id, nombre, apellido
FROM alumnos
LIMIT 20
OFFSET 20;
```

```
SELECT id, nombre, apellido
```
Seleccionamos las columnas id, nombre y apellido.

```
FROM alumnos
LIMIT 20
OFFSET 20;
```

```
SELECT id, nombre, apellido
FROM alumnos
```
de la tabla alumnos.

```
LIMIT 20
OFFSET 20;
```

```
SELECT id, nombre, apellido
FROM alumnos
LIMIT 20
```
Limitamos los registros de la tabla resultante a **20** registros.

```
OFFSET 20;
```

```
SELECT id, nombre, apellido
FROM alumnos
LIMIT 20
OFFSET 20;
```

Desplazamos los resultados 20 posiciones para que se muestre desde la posición **21**.

Alias

Imaginemos que estamos trabajando con una base de datos cuyas columnas de las tablas están en inglés, pero nos solicitan expresamente que los resultados de las consultas deben traer el nombre de las columnas en español.

¿Cómo podemos resolver esto?

Para ello vamos a hacer uso de los Alias de MySQL. Veamos entonces qué puede hacer esta directriz.

Alias

Los **alias** se usan para darle un nombre temporal y más amigable a las **tablas**, **columnas** y **funciones**. Los **alias** se definen durante una consulta y persisten **solo** durante esa consulta.

Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias

```sql
SELECT nombre_columna1 AS alias_nombre_columna1
FROM nombre_tabla;
```

Alias para una columna

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

Seleccionamos la **columna** *razon_social_cliente* y le asignamos el **alias** nombre.

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

En el **FROM** elegimos tabla cliente.
Con el **ORDER BY** ordenamos los registros con la columna nombre.

Alias para una tabla

```sql
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

```sql
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

Seleccionamos las columnas nombre, apellido y edad.

```
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

Hacemos la consulta sobre la tabla *alumnos_comision_inicial* y le **asignamos** el **alias** alumnos.

No es recomendable asignar más de una palabra dentro de un alias. En el caso de necesitarlo, utilizar "_".

> De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.

RESUMEN: Queries ML

En la clase vamos a poner en práctica lo aprendido sobre la sintaxis SQL de extracción de datos.

Es por esto que:

Vamos a realizar un repaso de la sintaxis vista.

Vamos a practicar y a sumar horas de vuelo en SQL.

¿Estás listo?

SELECT - Cómo usarlo

Toda consulta a la base de datos va a empezar con la palabra **SELECT**.

Su funcionalidad es la de realizar consultas sobre **una** o **varias columnas** de una tabla.

Para especificar sobre qué tabla queremos realizar esa consulta usamos la palabra **FROM** seguida del nombre de la tabla.

```sql
SELECT nombre_columna, nombre_columna, ...
FROM nombre_tabla;
```

Ejemplo - Tabla Peliculas

id	título	rating	fecha_estreno	país
1001	Pulp Fiction	9.8	1995-02-16	Estados Unidos
1002	Kill Bill	9.5	2003-11-27	Estados Unidos

De esta tabla completa, para conocer solamente los títulos y ratings de las películas guardadas en la tabla **películas,** podríamos hacerlo ejecutando la siguiente consulta:

```sql
SELECT id, titulo, rating
FROM peliculas;
```

Where

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas **SELECT** que se realizan a una base de datos.

```sql
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;
```

Teniendo una tabla **usuarios**, podríamos consultar nombre y edad, filtrando con un **WHERE** solamente los usuarios **mayores de 17 años** de la siguiente manera:

```sql
SELECT nombre, edad
FROM usuarios
WHERE edad > 17;
```

Operadores

=	Igual a
>	Mayor que
>=	Mayor o igual que
<	Menor que
<=	Menor o igual que
<>	Diferente a
!=	Diferente a

IS NULL	Es nulo
BETWEEN	Entre dos valores
IN	Lista de valores
LIKE	Se ajusta a...

Queries de ejemplo

```sql
SELECT nombre, edad
FROM usuarios
WHERE edad > 17;
```

```sql
SELECT *
FROM movies
WHERE title LIKE 'Avatar';
```

```sql
SELECT *
FROM movies
      WHERE awards >= 3
      AND awards < 8;
```

```sql
SELECT *
FROM movies
      WHERE awards = 2
      OR awards = 6;
```

```sql
DELETE FROM usuarios
WHERE id = 2;
```

Si en esta query quitáramos el WHERE...
¡Borraríamos toda la tabla!

Order By

ORDER BY se utiliza para ordenar los resultados de una consulta **según el valor de la columna especificada**. Por defecto, se ordena de forma ascendente *(ASC)* según los valores de la columna. También se puede ordenar de manera descendente (DESC) aclarándolo en la consulta.

```sql
SELECT nombre_columna1, nombre_columna2
FROM tabla
WHERE condicion
ORDER BY nombre_columna1;
```

Query de ejemplo

Teniendo una tabla **usuarios**, podría consultar los nombres, filtrar con un **WHERE** sólo los usuarios **mayores de 21 años** y ordenarlos de forma descendente tomando como referencia la columna nombre.

```sql
SELECT nombre, edad
FROM usuarios
WHERE edad > 21
ORDER BY nombre DESC;
```

BETWEEN

Cuando necesitamos obtener valores **dentro de un rango**, usamos el operador BETWEEN.

- ➔ BETWEEN **incluye** los **extremos**.
- ➔ BETWEEN funciona con **números**, **textos** y **fechas**.
- ➔ Se usa como un filtro de un WHERE.

Por ejemplo, coloquialmente:

- ➔ Dados los números: 4, 7, 2, 9, 1

Si hiciéramos un BETWEEN entre 2 y 7 devolvería 4, 7, 2 *(excluye el 9 y el 1, e incluye el 2).*

Query de ejemplo

Con la siguiente consulta estaríamos seleccionando **nombre** y **edad** de la tabla **alumnos** sólo cuando las edades estén **entre** 6 y 12.

```sql
SELECT nombre, edad
FROM alumnos
WHERE edad BETWEEN 6 AND 12;
```

LIKE

Cuando hacemos un filtro con un **WHERE**, podemos especificar un patrón de búsqueda que nos permita especificar algo concreto que queremos encontrar en los registros. Eso lo logramos utilizando **comodines** *(wildcards)*.

Por ejemplo, podríamos querer buscar:

- ➔ Los nombres que tengan la letra 'a' como segundo carácter.
- ➔ Las direcciones postales que incluyan la calle 'Monroe'.
- ➔ Los clientes que empiecen con 'Los' y terminen con 's'.

> COMODÍN %
>
> Es un sustituto que representa **cero**, **uno**, o **varios** caracteres.

> COMODÍN _
>
> Es un sustituto para **un solo** carácter.

Queries de ejemplo

```sql
SELECT nombre
FROM usuarios
WHERE nombre LIKE '_a%';
```

Devuelve aquellos nombres que tengan la letra 'a' como segundo carácter.

```sql
SELECT nombre
FROM usuarios
WHERE direccion LIKE '%Monroe%';
```

Devuelve las direcciones de los usuarios que incluyan la calle 'Monroe'.

```sql
SELECT nombre
FROM clientes
WHERE nombre LIKE 'Los%s';
```

Devuelve los clientes que empiecen con 'Los' y terminen con 's'.

Limit

Su funcionalidad es la de **limitar el número de filas** (registros/resultados) devueltas en las consultas SELECT. También establece el **número máximo** de registros a eliminar con DELETE.

```sql
SELECT nombre_columna1, nombre_columna2
FROM nombre_tabla
LIMIT cantidad_de_registros;
```

Query de ejemplo

Teniendo una tabla **peliculas**, podríamos armar un top 10 con las películas que tengan más de 4 premios usando un

LIMIT en la siguiente consulta:

```sql
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

Offset

- En un escenario en donde hacemos una consulta de todas las películas de la base de datos, la misma nos devolvería muchos registros. Usando un **LIMIT** podríamos aclarar un límite de 20.
- *¿Pero cómo haríamos si quisiéramos recuperar sólo 20 películas pero salteando las primeras 10 de la tabla?*
- **OFFSET** nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

{código}

```
SELECT id, nombre, apellido
FROM alumnos
LIMIT 20
OFFSET 20;
```

Alias

Los **alias** se usan para darle un nombre temporal y más amigable a las **tablas**, **columnas** y **funciones**. Los **alias** se definen durante una consulta y persisten **solo** durante esa consulta.

Para definir un alias usamos las iniciales **AS** precediendo a la columna que estamos queriendo asignarle ese alias

```sql
SELECT nombre_columna1 AS alias_nombre_columna1
FROM nombre_tabla;
```

Alias para una columna

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

Seleccionamos la **columna** *razon_social_cliente* y le asignamos el **alias** nombre.

```sql
SELECT razon_social_cliente AS nombre
FROM cliente
ORDER BY nombre;
```

En el **FROM** elegimos tabla cliente.
Con el **ORDER BY** ordenamos los registros con la columna nombre.

Alias para una tabla

```sql
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

```sql
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

Seleccionamos las columnas nombre, apellido y edad.

```
SELECT nombre, apellido, edad
FROM alumnos_comision_inicial AS alumnos;
```

Hacemos la consulta sobre la tabla *alumnos_comision_inicial* y le **asignamos** el **alias** alumnos.

No es recomendable asignar más de una palabra dentro de un alias. En el caso de necesitarlo, utilizar "_".

> De este modo, podemos darle alias a las **columnas** y **tablas** que vamos trayendo y hacer más legible la manipulación de datos, teniendo siempre presente que los alias **no modifican** los nombres originales en la base de datos.

Funciones de agregación

MySQL trae consigo una gran cantidad de funcionalidades. Entre ellas, las funciones de agregación son una herramienta que podríamos considerar como un as bajo la manga.

Estas nos van a permitir hacer que el resultado de las

consultas muestren información, como la cantidad de registros, el promedio, el total de una determinada información almacenada en una columna, entre otras.

Las funciones de agregación más comunes (y con las que trabajaremos en este video) serán:

- COUNT
- AVG
- SUM
- MIN
- MAX

Como podemos inferir, el propio nombre deja visualizar cuál será el resultado que podemos obtener al momento de implementar una de estas funciones. Veamos entonces cada una de ellas en profundidad.

> Las funciones de agregación **realizan cálculos** sobre un conjunto de datos y **devuelven** un **único resultado**. Excepto **COUNT**, las funciones de agregación **ignorarán** los valores **NULL**.

COUNT

Devuelve un **único** resultado indicando la cantidad de **filas/registros** que cumplen con el criterio.

```sql
SELECT COUNT(*) FROM movies;
```

Devuelve la cantidad de registros de la tabla movies.

```sql
SELECT COUNT(id) AS total FROM movies WHERE genre_id=3;
```

Devuelve la cantidad de películas de la tabla movies con el genero_id 3 y lo muestra en una columna denominada total.

AVG, SUM

AVG (*average*) devuelve un **único** resultado indicando el **promedio** de una columna cuyo tipo de datos debe ser numérico.

SUM (suma) devuelve un **único** resultado indicando la **suma** de una columna cuyo tipo de datos debe ser numérico.

```sql
SELECT AVG(rating) FROM movies;
```

Devuelve el promedio del rating de las películas de la tabla movies.

```sql
SELECT SUM(length) FROM movies;
```

Devuelve la suma de las duraciones de las películas de la tabla movies.

MIN, MAX

MIN devuelve un **único** resultado indicando el valor **mínimo** de una columna cuyo tipo de datos debe ser numérico.

MAX devuelve un **único** resultado indicando el valor **máximo** de una columna cuyo tipo de datos debe ser numérico.

```sql
SELECT MIN(rating) FROM movies;
```

Devolverá el rating de la película menos ranqueada.

```sql
SELECT MAX(length) FROM movies;
```

Devolverá el rating de la película mejor ranqueada.

GROUP BY

Pensemos en un escenario en donde nuestro líder técnico nos solicita un reporte de la cantidad de autos agrupados por marca. Es decir, se desea visualizar cuántos autos hay en stock de la marca Chevrolet, Fiat, Renault y similares.

No hace falta volverse loco, la solución –como ya es habitual en MySQL– es más sencilla de lo que se cree.

La directriz **GROUP BY** nos va a permitir agrupar los registros de la tabla resultante de una consulta por una o más columnas, según nos sea necesario.

Veamos entonces toda la magia que trae consigo **GROUP BY** para que de esta manera el líder técnico siga viendo todas nuestras capacidades.

Sintaxis

GROUP BY se usa para **agrupar los registros** de la tabla resultante de una consulta por una o más columnas.

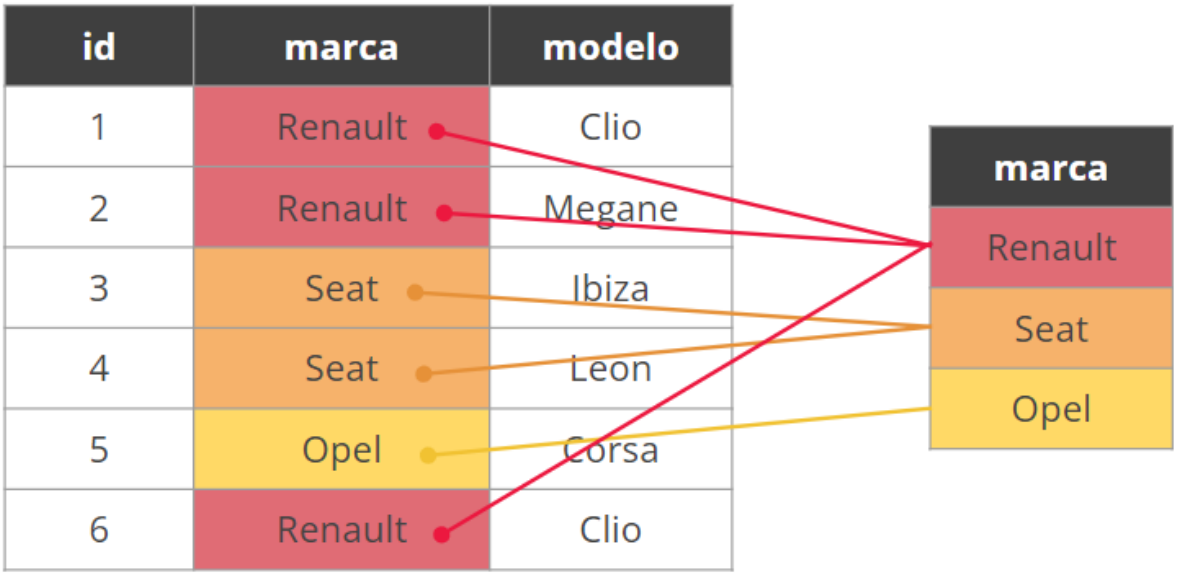
```sql
SELECT columna_1
FROM nombre_tabla
WHERE condition
GROUP BY columna_1;
```

Ejemplo

En el siguiente ejemplo, se utiliza **GROUP BY** para agrupar los

coches por **marca** mostrando aquellos que tienen el año de fabricación igual o superior al año **2010**

```sql
SELECT marca
FROM coche
WHERE anio_fabricacion>= 2010
GROUP BY marca;
```



Agrupación de datos

Dado que **GROUP BY** agrupa la información, perdemos el detalle de cada una de las filas. Es decir, ya no nos interesa el valor de cada fila, sino un resultado consolidado entre todas las filas. Veamos la siguiente consulta:

```sql
SELECT id, marca
FROM coche
GROUP BY marca;
```

Si agrupamos los coches por **marca**, ya no podremos visualizar el ID de cada fila. Posiblemente, en la fila nos muestre —para el campo **ID**— el primero de cada grupo de registros.

Ejemplos

```sql
SELECT marca, MAX(precio) AS precio_maximo
FROM coche
GROUP BY marca;
```

Devuelve la marca y el precio más alto de cada grupo de marcas.

```sql
SELECT genero, AVG(duracion) AS duracion_promedio
FROM pelicula
GROUP BY genero;
```

Devuelve el género y la duración promedio de cada grupo de géneros.

Conclusión

En resumen, la cláusula **GROUP BY**:

- Se usa para **agrupar filas** que contienen los **mismos valores**.
- Opcionalmente, se utiliza junto con las **funciones de agregación** (SUM, AVG, COUNT, MIN, MAX) con el

objetivo de producir reportes resumidos.

- Las consultas que contienen la cláusula GROUP BY se denominan **consultas agrupadas** y solo devuelven una **sola fila** para cada elemento agrupado.

```sql
SELECT marca, MAX(precio)
FROM coche
GROUP BY marca;
```

Having

Muchas veces, cuando estamos implementando las funciones de agregación, vamos a querer filtrar los resultados obtenidos. Seguramente, lo primero que se nos viene a la cabeza es hacer uso de un WHERE. El problema es que las funciones de agregación no son muy amigas del WHERE.

Es por esta razón que la directriz **HAVING** cumple la misma función, pero –¡ojo!– esta solo se va a poder usar en conjunto con las funciones de agregación para filtrar datos agregados. Es importante tener en cuenta esto porque para cualquier otro escenario la herramienta que tendremos que utilizar es el WHERE.

Veamos entonces, cómo trabaja el HAVING.

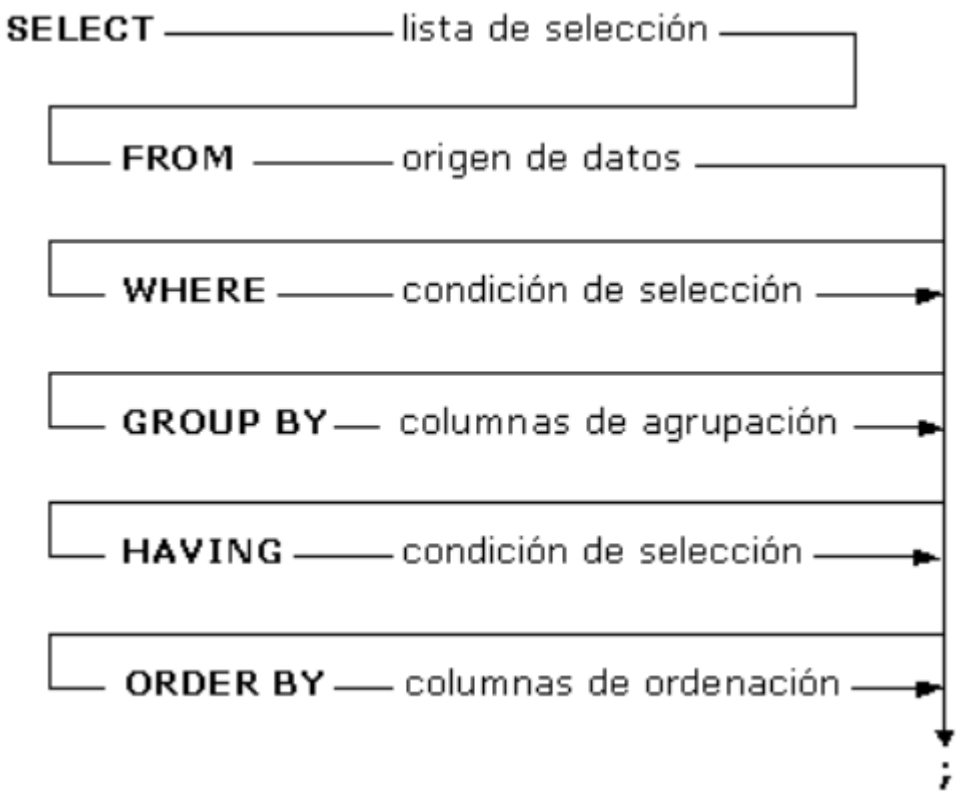
Sintaxis

Cumple la misma función que **WHERE**, a diferencia de que **HAVING** permite la implementación de **alias** y **funciones de agregación** en las condiciones de la selección de **datos.**

```SQL
SELECT columna_1
FROM nombre_tabla
WHERE condition
GROUP BY columna_1
HAVING condition_Group
ORDER BY columna_1;
```

Esta consulta devolverá la cantidad de clientes por país (agrupados por país). Solamente se incluirán en el resultado aquellos países que tengan al menos 3 clientes.

```SQL
SELECT pais, COUNT(clienteId)
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId)>=3;
```



Queries XL

En la clase de hoy vamos a poner en práctica lo aprendido sobre la sintaxis SQL de extracción de datos.

Es por esto que vamos a:

Realizar un repaso de la sintaxis vista.

Practicar y a sumar horas de vuelo en SQL.

Table reference

Ahora que aprendimos gran parte de la sintaxis de SQL para la extracción de datos,

empezamos a tener la necesidad de relacionar más de una tabla.

Ya vimos en el DER que las tablas se relacionan entre sí, pero ¿cómo hacemos para consultar los datos de más de una tabla?

Si bien en clase lo vamos a hacer con la sentencia llamada JOIN, veamos un adelanto:

Consultas a más de una tabla

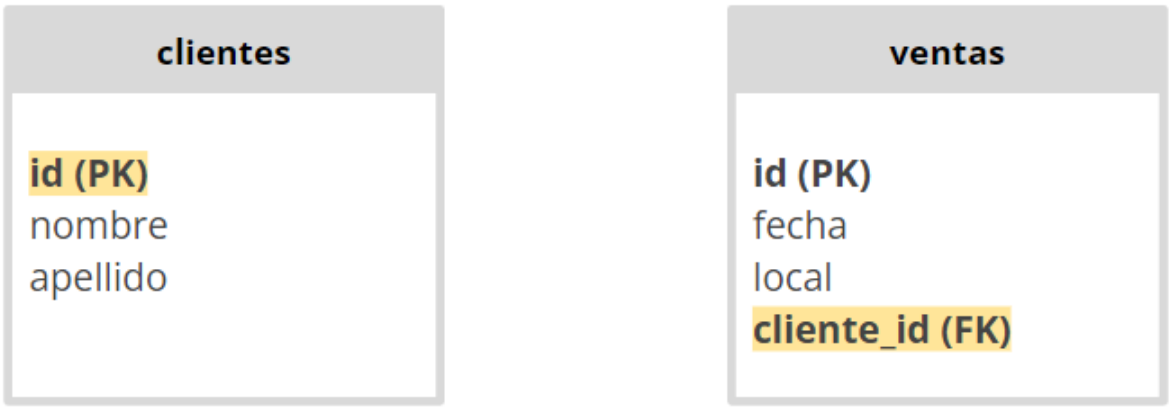
Hasta ahora vimos consultas (SELECT) dentro de una **tabla**. Pero también es posible y necesario hacer consultas a distintas tablas y unir los resultados.

Por ejemplo, un posible escenario sería querer consultar una tabla en donde están los **datos** de los **clientes** y otra tabla en donde están los **datos** de las **ventas a esos clientes.**

Seguramente, en la tabla de **ventas**, existirá un campo con el ID del cliente **(cliente_id).**

Si quisiéramos mostrar **todas** las ventas de un cliente concreto, necesitaremos usar datos de **ambas tablas** y

vincularlas con algún **campo** que **compartan**. En este caso, el **cliente_id.**



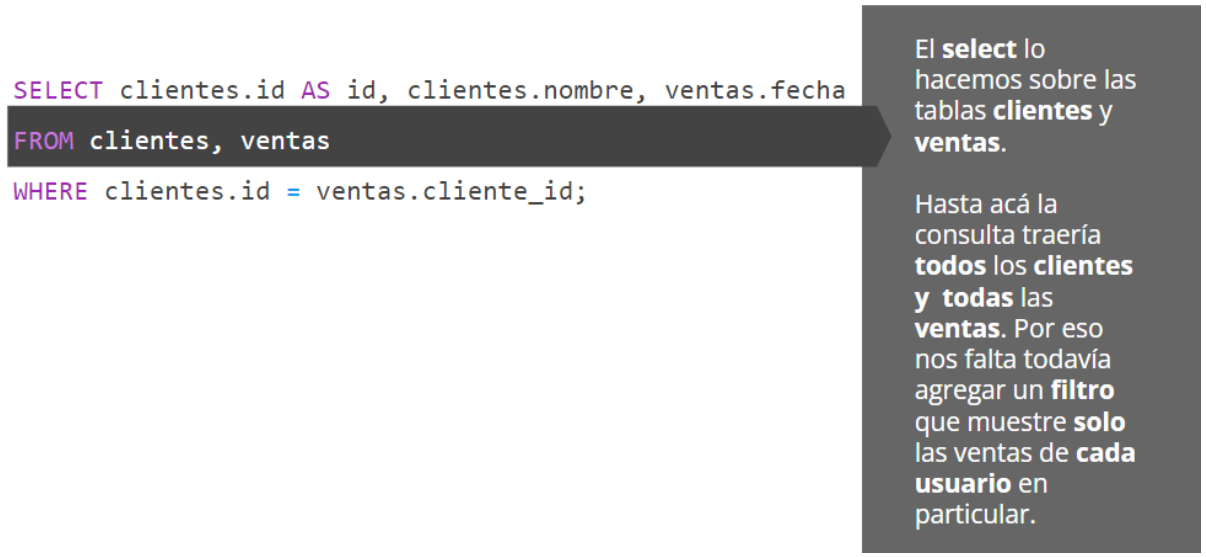
Consulta SQL

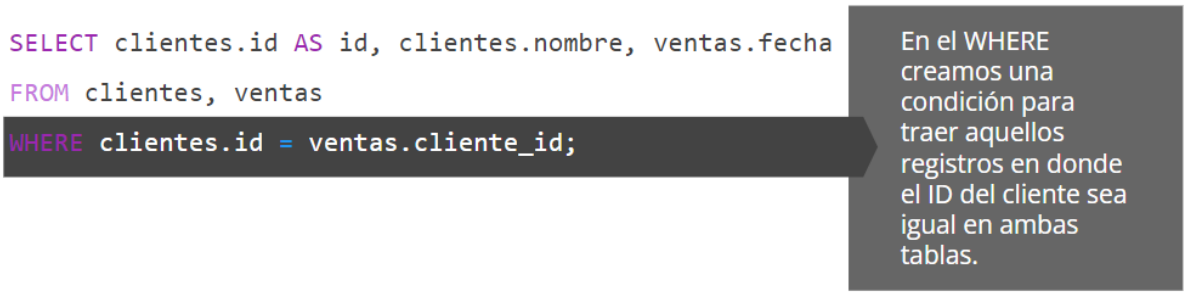
```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
```

```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
```

Seleccionamos:

- La columna **id** de la tabla **clientes** y le asignamos el alias **ID**.
- La columna **nombre** de la tabla **clientes**.
- La columna **fecha** de la tabla **ventas**.





JOINS

Imaginémonos el siguiente escenario: tenemos una tabla que almacena los datos de una persona, pero sabemos que esa tabla —a su vez— está asociada con otra que almacena las imágenes que postea una persona.

Sin embargo, en esta última tabla, el único dato que se tiene para identificar a la persona dueña de esa imagen es el ID de la misma. Es decir, el ID que le corresponde a esa persona en la tabla de personas.

Si este es el caso, ¿cómo podríamos, por ejemplo, traer en una sola consulta las imágenes que pertenecen al ID 5? Es decir, ¿cómo podemos traer todas las imágenes que

pertenecen a una persona si esa información está presente en otra tabla?

Para lograr esto, MySQL nos proporciona una herramienta llamada **JOIN**. Esta, tal como su nombre lo indica, establece uniones entre distintas tablas que tienen algún tipo de relación entre sí.

Con esto en mente, veámosla en detalle en el siguiente video.

¿Por qué usar JOIN?

Además de realizar consultas dentro de una tabla y de haber empleado **table reference** para consultas en múltiples tablas, existe la herramienta **JOIN** que nos permite hacer consultas a distintas tablas y unir los resultados.

Ventajas del uso de **JOIN**:

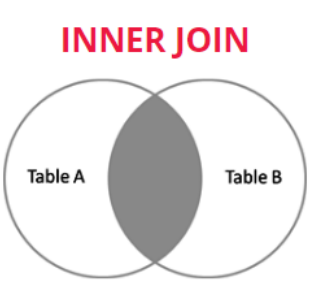
- Su sintaxis es mucho más comprensible.
- Presentan una mejor performance.
- Proveen de ciertas flexibilidades.

INNER JOIN

El **INNER JOIN** es la opción predeterminada y nos devuelve **todos los registros** donde se **cruzan dos o más tablas**. Por ejemplo, si tenemos una tabla cliente y otra factura, al

cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García



factura		
id	cliente_id	fecha
1	2	12/03/2019
2	2	22/08/2019
3	1	04/09/2019

Consulta a múltiples tablas

Antes con **table reference** escribíamos:

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente, factura;
```

Ahora con **INNER JOIN** escribimos:

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente
INNER JOIN factura;
```

> Si bien ya dimos el primer paso (que es **cruzar** ambas tablas), aún nos falta aclarar **dónde** está ese cruce.
>
> Es decir, qué **clave primaria (PK)** se cruzará con qué **clave foránea (FK)**.

Definiendo el INNER JOIN

Para definir el **INNER JOIN** tenemos que indicar el filtro por el cual se **evaluará** el **cruce**. Para esto, debemos utilizar la palabra reservada **ON**. Es decir, que lo que antes escribíamos en el **WHERE** de table reference, ahora lo escribiremos en el **ON** de INNER JOIN.

```sql
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente
INNER JOIN factura
ON cliente.id = factura.cliente_id;
```

Distinct

Imaginemos el siguiente escenario: viene nuestro líder de equipo y nos solicita un reporte que traiga todos los productos de la base de datos, pero solamente aquellos que no se encuentran duplicados. Es decir, se desea que en dicho reporte se vean solamente los valores únicos, porque se conoce que –para una determinada columna de esa tabla– puede llegar a haber valores duplicados.

Antes de entrar en pánico y salir corriendo de la oficina, dejanos decirte que la solución a dicha petición se resuelve de una manera bastante sencilla gracias a la cláusula **DISTINCT** de MySQL.

Veamos entonces todo lo que DISTINCT puede hacer por nosotros.

Cómo funciona

Al realizar una consulta en una tabla, puede ocurrir que en los resultados existan dos o más **filas idénticas**. En algunas situaciones, nos pueden solicitar un listado con registros **no duplicados**, para esto, utilizamos la cláusula **DISTINCT** que devuelve un listado en donde cada fila es distinta.

```sql
SELECT DISTINCT columna_1, columna_2
FROM nombre_tabla;
```

Ejemplo

Partiendo de una tabla de **usuarios**, si ejecutamos la consulta:

```sql
SELECT pais FROM usuarios;
```

Obtendremos cinco filas:

usuarios
Perú
Perú
Argentina
Colombia
Argentina

Si agregamos la cláusula **DISTINCT** en la consulta:

```sql
SELECT DISTINCT pais FROM usuarios;
```

Obtendremos tres filas:

usuarios
Perú
Argentina
Colombia

{código}

```sql
SELECT DISTINCT actor.nombre, actor.apellido
FROM actor
INNER JOIN actor_pelicula
ON actor_pelicula.actor_id = actor.id
INNER JOIN pelicula
ON pelicula.id = actor_pelicula.pelicula_id
WHERE pelicula.titulo LIKE '%Harry Potter%';
```

En este ejemplo vemos una query que **pide** los **actores** que hayan actuado en **cualquier película** de **Harry Potter**.

Si no escribiéramos el **DISTINCT**, los actores que hayan participado en más de una película, aparecerán repetidos en el resultado.

CONCAT

Usamos **CONCAT** para **concatenar** dos o más expresiones:

```sql
SELECT CONCAT('Hola', ' a ', 'todos.');
```

```
> 'Hola a todos.'
```

```sql
SELECT CONCAT('La respuesta es: ', 24, '.');
```

```
> 'La respuesta es 24.'
```

```sql
SELECT CONCAT('Nombre: ', apellido, ', ', nombre, '.')
FROM actor;
```

```
> 'Nombre: Clarke, Emilia.'
```

COALESCE

Usamos **COALESCE** para sustituir el valor **NULL** en una sucesión de expresiones o campos. Es decir, si la primera expresión es Null, se sustituye con el valor de una segunda expresión, pero si este valor también es Null, se puede sustituir con el valor de una tercera expresión y así sucesivamente.

```sql
SELECT COALESCE(NULL, 'Sin datos');
```

```
> 'Sin datos'
```

```sql
SELECT COALESCE(NULL, NULL, 'Digital House');
```

```
> 'Digital House'
```

Los tres clientes de la siguiente tabla poseen uno o más datos nulos:

```sql
SELECT id, apellido, nombre, telefono_movil, telefono_fijo
FROM cliente;
```

cliente				
id	**apellido**	**nombre**	**telefono_movil**	**telefono_fijo**
1	Pérez	Juan	1156685441	43552215
2	Medina	Rocío	Null	43411722
3	López	Matías	Null	Null

Usando **COALESCE** podremos sustituir los **datos nulos** en cada registro, indicando la columna a evaluar y el valor de sustitución.

```sql
SELECT id, apellido, nombre, COALESCE(telefono_movil, telefono_fijo, 'Sin datos')
AS telefono FROM cliente;
```

cliente			
id	**apellido**	**nombre**	**telefono**
1	Pérez	Juan	1156685441
2	Medina	Rocío	**43411722**
3	López	Matías	**Sin datos**

DATEDIFF

Usamos **DATEDIFF** para devolver la **diferencia** entre dos fechas,
tomando como granularidad el intervalo especificado.

```sql
SELECT DATEDIFF('2021-02-03 12:45:00', '2021-01-01 07:00:00');
```

```
> 33
```

Devuelve 33 porque es la cantidad de días de la diferencia entre las fechas indicadas.

```sql
SELECT DATEDIFF('2021-01-15', '2021-01-05');
```

```
> 10
```

Devuelve 10 porque es la cantidad de días de la diferencia entre las fechas indicadas.

TIMEDIFF

Usamos **TIMEDIFF** para devolver la **diferencia** entre dos horarios,
tomando como granularidad el intervalo especificado

```sql
SELECT TIMEDIFF('2021-01-01 12:45:00', '2021-01-01 07:00:00');
```

```
> 05:45:00
```

```sql
SELECT TIMEDIFF('18:45:00', '12:30:00');
```

```
> 06:15:00
```

EXTRACT

Usamos **EXTRACT** para **extraer** partes de una fecha:

```sql
SELECT EXTRACT(SECOND FROM '2014-02-13 08:44:21');
```

```
> 21
```

```sql
SELECT EXTRACT(MINUTE FROM '2014-02-13 08:44:21');
```

```
> 44
```

```sql
SELECT EXTRACT(HOUR FROM '2014-02-13 08:44:21');
```

```
> 8
```

```sql
SELECT EXTRACT(DAY FROM '2014-02-13 08:44:21');
```

```
> 13
```

```sql
SELECT EXTRACT(WEEK FROM '2014-02-13 08:44:21');
```

```
> 6
```

```sql
SELECT EXTRACT(MONTH FROM '2014-02-13 08:44:21');
```

```
> 2
```

```sql
SELECT EXTRACT(QUARTER FROM '2014-02-13 08:44:21');
```

```
> 1
```

```sql
SELECT EXTRACT(YEAR FROM '2014-02-13 08:44:21');
```

```
> 2014
```

REPLACE

Usamos **REPLACE** para reemplazar una cadena de caracteres por otro valor. Cabe aclarar que esta función hace distinción entre minúsculas y mayúsculas.

```sql
SELECT REPLACE('Buenas tardes', 'tardes', 'Noches');
```

```
> Buenas Noches
```

```sql
  SELECT REPLACE('Buenas tardes', 'a', 'A');
```

```
> BuenAs tArdes
```

```sql
SELECT REPLACE('1520', '2', '5');
```

```
> 1550
```

DATE_FORMAT

Usamos **DATE_FORMAT** para cambiar el formato de salida de una fecha según una condición dada.

```sql
SELECT DATE_FORMAT('2017-06-15', '%Y');
```

```
> 2017
```

```sql
SELECT DATE_FORMAT('2017-06-15', '%W %M %e %Y');
```

```
> Thursday June 15 2017
```

Para mostrar la fecha escrita en español se debe configurar el idioma con la siguiente instrucción:

```sql
SET lc_time_names = 'es_ES';
SELECT DATE_FORMAT('2017-06-15', '%W, %e de %M de %Y');
```

```
> jueves, 15 de junio de 2017
```

DATE_ADD

Usamos **DATE_ADD** para sumar o agregar un período de tiempo a un valor de tipo DATE o DATETIME.

```sql
SELECT DATE_ADD('2021-06-30', INTERVAL '3' DAY);
```

```
> 2021-07-03
```

```sql
SELECT DATE_ADD('2021-06-30', INTERVAL '9' MONTH);
```

```
> 2022-03-30
```

```sql
SELECT DATE_ADD('2021-06-30 09:30:00', INTERVAL '4' HOUR);
```

```
> 2021-06-30 13:30:00
```

DATE_SUB

Usamos **DATE_SUB** para restar o quitar un período de tiempo a un valor de tipo DATE o DATETIME.

```sql
SELECT DATE_SUB('2021-06-30', INTERVAL '3' DAY);
```

```
> 2021-06-27
```

```sql
SELECT DATE_SUB('2021-06-30', INTERVAL '9' MONTH);
```

```
> 2020-09-30
```

```sql
SELECT DATE_SUB('2021-06-30 09:30:00', INTERVAL '4' HOUR);
```

```
> 2021-06-30 05:30:00
```

CASE

Usamos **CASE** para **evaluar condiciones** y devolver la primera condición que se cumpla. En este ejemplo, la tabla resultante tendrá 4 columnas: id, titulo, rating, calificacion. Esta última columna mostrará los valores: Mala, Regular, Buena y Excelente; **según** el **rating** de la película.

```sql
SELECT id, titulo, rating,
  CASE
    WHEN rating < 4 THEN 'Mala'
    WHEN rating BETWEEN 4 AND 6 THEN 'Regular'
    WHEN rating BETWEEN 7 AND 9 THEN 'Buena'
    ELSE 'Excelente'
  END AS calificacion
FROM pelicula;
```

A continuación, se muestra la tabla resultante después de haber aplicado la función **CASE**.

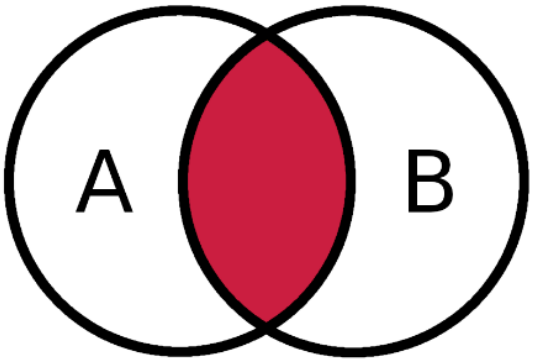
pelicula			
id	titulo	rating	calificacion
1	El Padrino	6	Regular
2	Tiburón	4	Regular
3	Jurassic Park	9	Buena
4	Titanic	10	Excelente
5	Matrix	3	Mala

INNER JOIN

El **INNER JOIN** entre dos tablas devuelve únicamente los registros que cumplen la condición indicada en la cláusula **ON**.

SQL

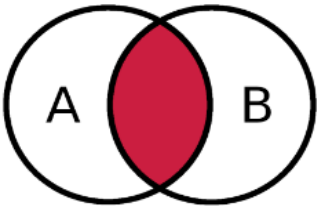
```
SELECT columna1, columna2, …
FROM tabla A
INNER JOIN tabla B
ON condicion
```



El **INNER JOIN** es la opción predeterminada y nos devuelve **todos los registros** donde **se cruzan dos o más tablas**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas con **INNER JOIN**, nos devuelve aquellos registros o filas donde haya un valor coincidente en ambas tablas.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

INNER JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	null	20/09/2019
13	1	24/09/2019

El ejemplo anterior, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
INNER JOIN factura
ON cliente.id = factura.cliente_id;
```

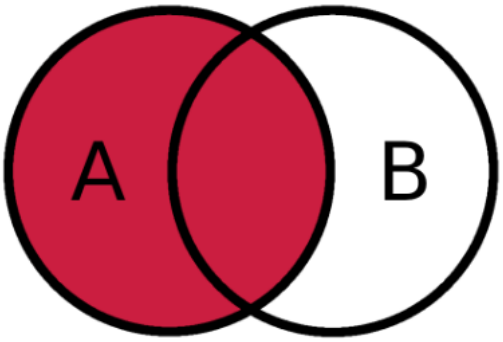
A continuación, se muestran los datos obtenidos:

nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019

LEFT JOIN

El **LEFT JOIN** entre dos tablas devuelve todos los registros de la primera tabla (en este caso sería la tabla A), incluso cuando los registros no cumplan la condición indicada en la cláusula **ON**.

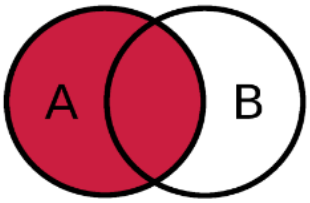
```sql
SELECT columna1, columna2, …
FROM tabla A
LEFT JOIN tabla B
ON condicion
```



Entonces, **LEFT JOIN** nos devuelve **todos** los registros donde se **cruzan dos o más tablas**. Incluso los registros de una primera tabla (A) que **no cumplan** con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellos clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

LEFT JOIN



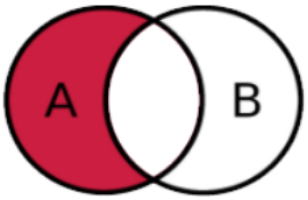
factura		
id	cliente_id	fecha
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

LEFT Excluding JOIN

Este tipo de **LEFT JOIN** nos devuelve únicamente los **registros** de una primera tabla (A), excluyendo los registros que cumplan con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de clientes que no tengan una factura asignada.

cliente		
id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

LEFT Excluding JOIN



factura		
id	cliente_id	fecha
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

LEFT Excluding JOIN

Continuando con el ejemplo, se podría construir de la siguiente manera:

SQL

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
LEFT JOIN factura
ON cliente.id = factura.cliente_id
WHERE factura.id IS NULL;
```

A continuación, se muestran los datos obtenidos:

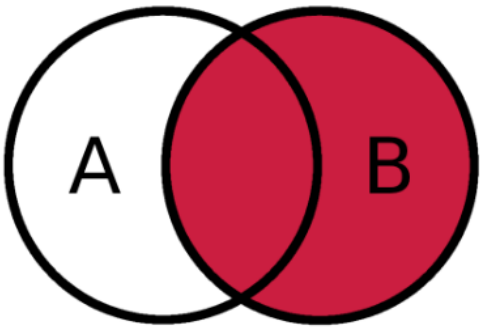
nro_factura	apellido	nombre	fecha
null	García	Marta	**null**

RIGHT JOIN

El **RIGHT JOIN** entre dos tablas devuelve todos los registros de la segunda tabla, incluso cuando los registros no cumplan la condición indicada en la cláusula **ON**.

SQL

```
SELECT columna1, columna2, …
FROM tabla A
RIGHT JOIN tabla B
ON condicion
```

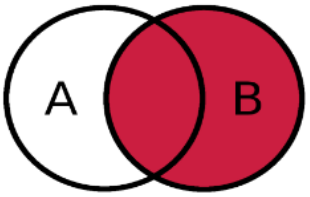


Entonces, **RIGHT JOIN** nos devuelve **todos** los registros donde se **cruzan dos o más tablas**. Incluso los registros de una segunda tabla (B) que **no cumplan** con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve aquellos registros donde haya un valor coincidente entre ambas, más los registros de aquellas facturas que no tengan un cliente asignado.

cliente

id	nombre	apellido
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT JOIN



factura

id	cliente_id	fecha
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

El ejemplo anterior, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
RIGHT JOIN factura
ON cliente.id = factura.cliente_id;
```

A continuación, se muestran los datos obtenidos:

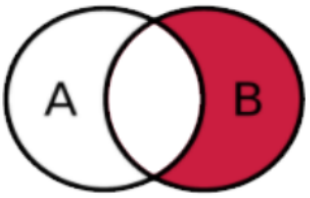
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
12	**null**	**null**	20/09/2019
13	Perez	Juan	24/09/2019

RIGHT Excluding JOIN

Este tipo de **RIGTH JOIN** nos devuelve únicamente los registros de una segunda tabla (B), **excluyendo** los registros que cumplan con la condición indicada en la cláusula **ON**. Por ejemplo, si tenemos una tabla cliente y otra factura, al cruzarlas, nos devuelve solo aquellos registros de facturas que no tengan asignado un cliente.

cliente		
id	**nombre**	**apellido**
1	Juan	Perez
2	Clara	Sanchez
3	Marta	García

RIGHT Excluding JOIN



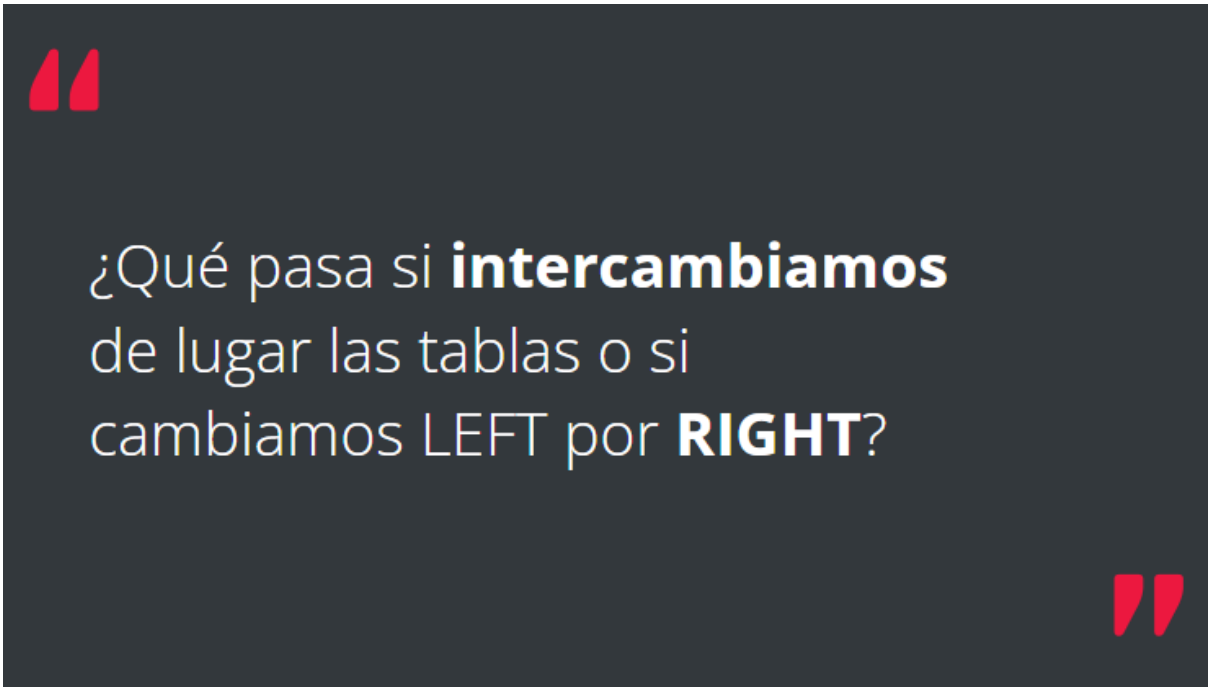
factura		
id	**cliente_id**	**fecha**
11	2	12/09/2019
12	**null**	20/09/2019
13	1	24/09/2019

Continuando con el ejemplo, se podría construir de la siguiente manera:

```sql
SELECT factura.id AS nro_factura, apellido, nombre, fecha
FROM cliente
RIGHT JOIN factura
ON cliente.id = factura.cliente_id
WHERE cliente.id IS NULL;
```

A continuación, se muestran los datos obtenidos:

nro_factura	apellido	nombre	fecha
12	**null**	**null**	20/09/2019



LEFT JOIN -> RIGHT JOIN

Experimentemos con el último ejemplo que vimos.

```sql
SELECT factura.id AS factura, apellido
FROM cliente
LEFT JOIN factura
ON factura.cliente_id = cliente.id;
```

```sql
SELECT factura.id AS factura, apellido
FROM cliente
RIGHT JOIN factura
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
13	Perez
null	García

factura	apellido
11	Sanchez
12	**null**
13	Perez

Intercambiando tablas

Ahora, intercambiamos el lugar de las tablas implicadas.

```sql
SELECT factura.id AS factura, apellido
FROM factura
LEFT JOIN cliente
ON factura.cliente_id = cliente.id;
```

```sql
SELECT factura.id AS factura, apellido
FROM factura
RIGHT JOIN cliente
ON factura.cliente_id = cliente.id;
```

Comparemos los resultados:

factura	apellido
11	Sanchez
12	**null**
13	Perez

factura	apellido
11	Sanchez
13	Perez
null	García