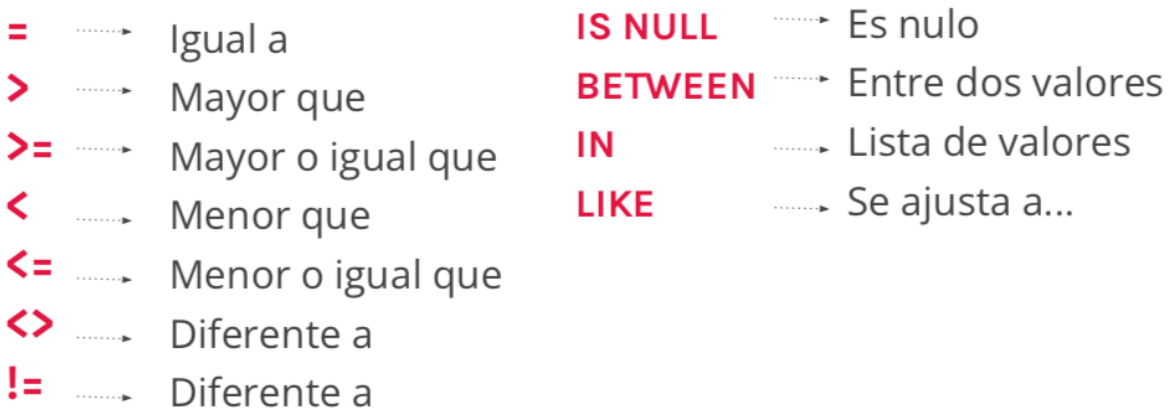
Consultas SQL

Operadores



`>` es `>` (greater than) mayor que

`<` es `<` (less than) menor que

BASE_DE_DATOS_I_-_Fuciones_de_alteracion.docx

CASE → https://www.w3schools.com/sql/sql_case.asp

▼ CREATE - DROP - ALTER

▼ Comandos SQL - CREATE

```
CREATE DATABASE miprimerabasededatos;
USE miprimerabasededatos;

// podemos crear una tabla desde cero, junto con sus columr
// tipos y constraints.
```

```
CREATE TABLE nombre_de_la_tabla (
    nombre_de_la_columna_1 TIPO_DE_DATO CONSTRAINT,
    nombre_de_la_columna_2 TIPO_DE_DATO CONSTRAINT
)
CREATE TABLE post (
    id INT PRIMARY KEY AUTO_INCREMENT,
    titulo VARCHAR(200)
)
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
);


// Cuando creemos una columna que contenga una id foránea,
// necesario usar la sentencia FOREIGN KEY para aclarar a c
// tabla y a qué columna hace referencia aquel dato.
// Es importante remarcar que la tabla “clientes” deberá ex
// antes de correr esta sentencia para crear la tabla “orde
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)
);
```

▼ Comandos SQL - DROP

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

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

Comandos SQL - ALTER

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.

```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

Comandos SQL - INSERT

Existen dos formas de agregar datos en una tabla:

Insertando datos en **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.

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

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

▼ Insertando datos en las **columnas** que **especifiquemos**.

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

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

INSERT INTO artistas (nombre, rating)
VALUES (‘Maluma’, 1.0);
```

▼ Comandos SQL - 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.

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

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

▼ Comandos SQL - DELETE

FCon **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.

```
DELETE FROM nombre_tabla WHERE condición;

DELETE FROM artistas WHERE id = 4;
```

▼ SELECT

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.

```
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 - ORDER BY

▼ Comandos SQL - WHERE

La funcionalidad del **WHERE** es la de condicionar y filtrar las consultas

SELECT que se realizan a una base de datos.

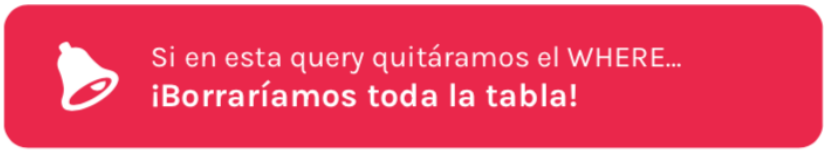
```
SELECT nombre_columna_1, nombre_columna_2, ...
FROM nombre_tabla
WHERE condicion;

//ejemplo
SELECT primer_nombre, apellido
FROM clientes
WHERE pais = 'Argentina';

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

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

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



▼ Comandos SQL - 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.

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

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

```
--

SELECT name, rating FROM actors
WHERE rating > 2
//se pasan dos parametros de ordenamiento.
ORDER BY name DESC, rating DESC
```

▼ BETWEEN Y LIKE

▼ Comandos SQL - 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.

```
-- Con la siguiente consulta estaríamos seleccionando nombr
-- de la tabla alumnos solo cuando las edades estén entre 6
SELECT nombre, edad
FROM alumnos
WHERE edad BETWEEN 6 AND 12;
```

▼ Comandos SQL - 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'.

COMODINES

- `%` → Es un sustituto que representa cero, uno, o varios caracteres.
- `_` → Es un sustituto para un solo carácter.

```
-- Devuelve aquellos nombres que tengan la letra 'a' como s
SELECT nombre
FROM usuarios
WHERE nombre LIKE '_a%';


-- Devuelve las direcciones de los usuarios que incluyan la
SELECT nombre
FROM usuarios
WHERE direccion LIKE '%Monroe%';
```

▼ LIMIT - OFSET

▼ Comandos SQL - BETWEEN

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.

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

-- Teniendo una tabla peliculas, podríamos armar un top 10
```

```
-- películas que tengan más de 4 premios usando un LIMIT en
-- consulta:
SELECT *
FROM peliculas
WHERE premios > 4
LIMIT 10;
```

▼ Comandos SQL - OFSET

Nos permite especificar a partir de qué fila comenzar la recuperación de los datos solicitados.

```
-- Desplazamos los resultados 20 posiciones para que se mue
-- posición 21.
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 **unciones**. 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.

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

▼ FUNCIONES DE AGREGACION

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.

▼ Comandos SQL - COUNT

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

```
-- Devuelve la cantidad de registros de la tabla movies.
SELECT COUNT(*) FROM movies;

-- Devuelve la cantidad de películas de la tabla movies
-- con el genero_id 3 y lo muestra en una columna denomina
SELECT COUNT(id) AS total FROM movies WHERE genre_id=3;
```

▼ Comandos SQL - 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.

```
-- Devuelve el promedio del rating de las películas de la t
SELECT AVG(rating) FROM movies;


-- Devuelve la suma de las duraciones de las películas de l
SELECT SUM(length) FROM movies;
```

▼ Comandos SQL - 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.

```
-- Devolverá el rating de la película menos ranqueada.
SELECT MIN(rating) FROM movies;


-- Devolverá la película con mayor duración
SELECT MAX(length) FROM movies;
```

▼ GROUP BY

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

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

-- Se utiliza GROUP BY para agrupar los coches por marca most
-- que tienen el año de fabricación igual o superior al año 2
SELECT marca
FROM coche
WHERE anio_fabricacion>= 2010
GROUP BY marca;
```

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:

id	marca	modelo
1	Renault	Clio
2	Renault	Megane
3	Seat	Ibiza
4	Seat	Leon
5	Opel	Corsa
6	Renault	Clio

marca
Renault
Seat
Opel

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.

▼ 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.

▼ HAVING

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**.

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

-- Ejemplos

-- Esta consulta devolverá la cantidad de clientes por país (
-- país). Solamente se incluirán en el resultado aquellos paí
-- al menos 3 clientes.
SELECT pais, COUNT(clienteId)
FROM clientes
GROUP BY pais
HAVING COUNT(clienteId)>=3;
```

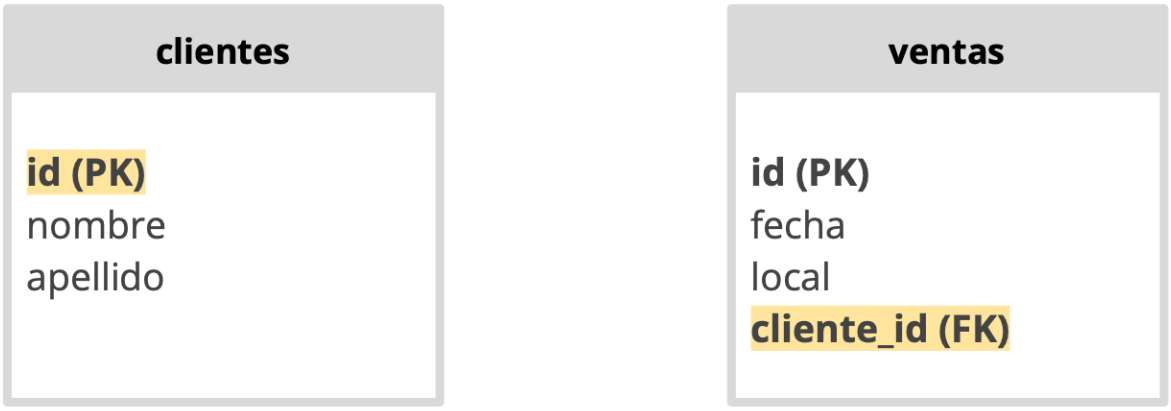
▼ TABLE REFERENCE

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.**




```
SELECT clientes.id AS ID, clientes.nombre, ventas.fecha
FROM clientes, ventas
WHERE clientes.id = ventas.cliente_id;
-- En el WHERE creamos una condición para traer aquellos regi
-- el ID del cliente sea igual en ambas tablas.
```

▼ JOIN

Permite hacer consultas a distintas tablas y unir los resultados.

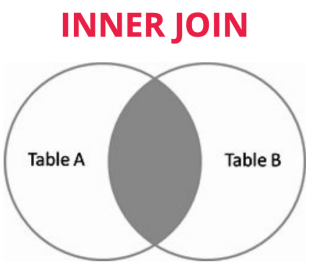
Ventajas del uso de **JOIN**:

- Su sintaxis es mucho más comprensible.

- Presentan una mejor performance.
- Proveen de ciertas flexibilidades.

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

```
-- Antes con table reference escribíamos:
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente, factura;
WHERE clientes.id = factura.cliente_id;

-- Ahora con INNER JOIN escribimos:
SELECT cliente.id, cliente.nombre, factura.fecha
FROM cliente
INNER JOIN factura;
```

▼ DISTINCT

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.

```
SELECT DISTINCT columna_1, columna_2
FROM nombre_tabla;

-- Ejemplo

-- En este ejemplo vemos una query que pide los actores que h
-- cualquier película de Harry Potter.
-- Si no escribiéramos el DISTINCT, los actores que hayan par
-- de una película, aparecerán repetidos en el resultado.
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%';
```

▼ FUNCIONES DE ALTERACION

▼ Comandos SQL - CONCAT

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

```
SELECT CONCAT('Nombre: ', apellido, ', ', nombre, '.')
FROM actor;
-- Nombre: Clarke, Emilia.'
```

▼ Comandos SQL - 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.

```
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**

▼ Comandos SQL - 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.

▼ Comandos SQL - 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
```

▼ Comandos SQL - 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
```

▼ Comandos SQL - 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
```

▼ Comandos SQL - 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
```

▼ Comandos SQL - 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
```

▼ Comandos SQL - 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

SELECT DATE_SUB('2021-06-30', INTERVAL '9' MONTH);
> 2020-09-30

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**

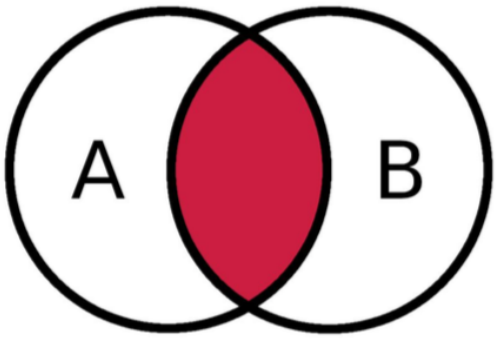
▼ JOIN 2da_Parte

▼ Comandos SQL - INNER JOIN

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

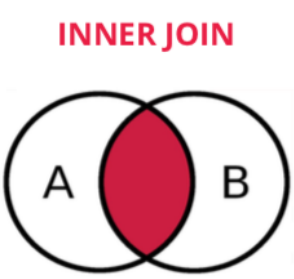
SQL

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



Ej.

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



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

```
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:

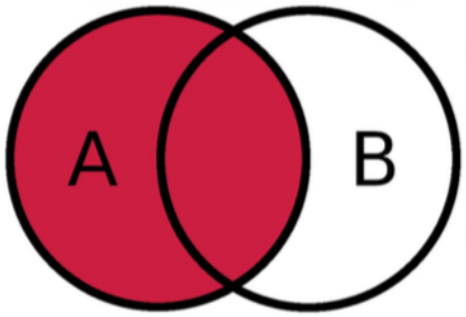
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019

▼ Comandos SQL - 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**.

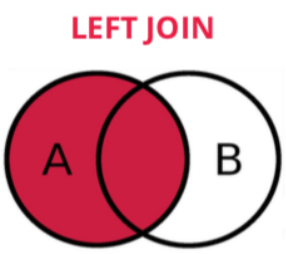
SQL

```
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



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
LEFT JOIN factura
ON cliente.id = factura.cliente_id;
```

A continuación, se muestran los datos obtenidos:

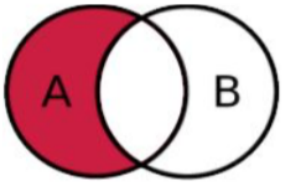
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
13	Perez	Juan	24/09/2019
null	García	Marta	**null**

▼ Comandos SQL - 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

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:

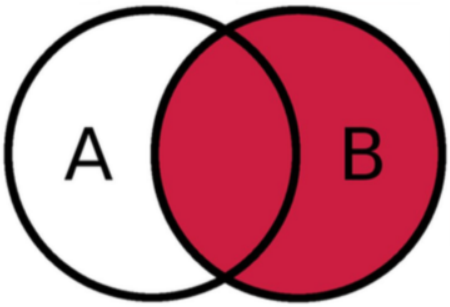
nro_factura	apellido	nombre	fecha
null	García	Marta	**null**

▼ Comandos SQL - 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

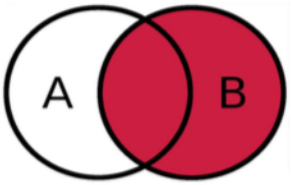
```sql
SELECT columna1, columna2, ...
FROM tabla A
RIGHT JOIN tabla B
ON condicion
```



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
indicada en la cláusula
ON. Por ejemplo, si tenemos una tabla cliente y otra
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:

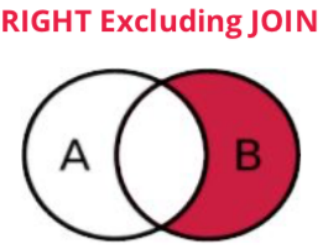
nro_factura	apellido	nombre	fecha
11	Sanchez	Clara	12/09/2019
12	**null**	**null**	20/09/2019
13	Perez	Juan	24/09/2019

▼ Comandos SQL - RIGHT Excluding JOIN

Este tipo de **RIGTH JOIN** nos devuelve únicamente los registros de una segunda
(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



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

```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

▼ Comandos SQL - 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

▼ Comandos SQL - 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

▼ C17 - DML Queries XXL - Parte II

▼ Vistas

Una **vista** es un elemento de la base de datos que facilita el acceso a los datos de las tablas. Básicamente, su función es **guardar** un **SELECT**. Este es un recurso que nos permite visualizar los resultados abstrayéndonos de cómo esté definida una consulta.

Las vistas tienen la misma **estructura** que una **tabla** (filas y columnas). La diferencia es que solo se almacena la **definición de la consulta**, no así los datos. En otras palabras, una vista es una tabla virtual basada en el conjunto de resultados de una **consulta SQL**.

▼ ¿Para qué sirven las vistas?

- Para simplificar el acceso a los datos cuando se requiere la implementación de consultas complejas.
- Para impedir la modificación de datos por terceros.

- Para facilitar la consulta de datos a aquellas personas que no conozcan el modelo de datos o que no sean expertos en SQL.

¿Qué hay que saber sobre las vistas?

- Una vista se **ejecuta** en el momento en que se invoca.
- Los **nombres** de las vistas deben ser **únicos** (no se pueden usar nombres de tablas existentes).
- Solamente se pueden incluir **sentencias SQL** de tipo **SELECT**.
- Los **campos** de las vistas heredan los **tipos de datos** de la tabla.
- El conjunto de resultados que devuelve una vista es **inmodificable**, a diferencia de lo que sucede con el conjunto de resultados de una tabla.

Creacion de vistas

Una vista se crea con la cláusula **CREATE VIEW**:

```
CREATE VIEW nombre_de_la_vista AS consulta SQL;

CREATE VIEW canciones_de_rock AS
SELECT canciones.id, canciones.nombre, generos.nombre AS
INNER JOIN generos
ON canciones.id_genero = generos.id
WHERE generos.nombre IN ('Rock', 'Rock And Roll');
```

Modificacion de vistas

Usamos la cláusula **ALTER VIEW** para modificar o reemplazar una vista.

```
ALTER VIEW nombre_de_la_vista AS consulta SQL;

ALTER VIEW vista_coche AS SELECT * FROM coche WHERE marc
```

Modificacion de vistas

Utilizamos la cláusula **DROP VIEW** para eliminar una vista.

```
DROP VIEW nombre_de_la_vista;

DROP VIEW vista_coche;
```

▼ Invocacion de vistas

Si bien las vistas son elementos diferentes a las tablas, la **invocación** es la misma
que la de una tabla. Es decir, se utiliza la cláusula
SELECT.

```
SELECT * FROM vista_coche;

SELECT * FROM vista_coche WHERE id > 10;
```