

IT Bootcamp

# Sentencias SQL Avanzadas

DigitalHouse>



SCHOOL OF  
INNOVATION®



TECH  
ACADEMY

# Índice

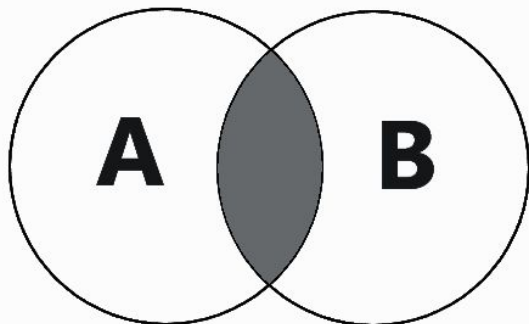
1. JOIN
2. GROUP BY
3. HAVING
4. SUBCONSULTAS

**1 | JOIN**

# Sentencia JOIN en SQL

La sentencia **INNER JOIN** es la sentencia **JOIN** por defecto. Se utiliza para obtener datos de varias tablas relacionadas entre sí.

Consiste en **combinar datos** de una tabla con datos de la otra tabla, a partir de una o varias condiciones en común.

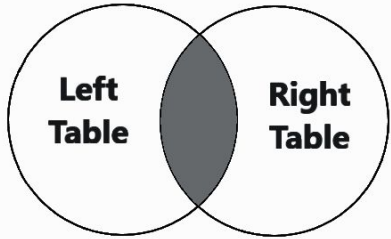


INTERSECCIÓN ( $A \cap B$ )

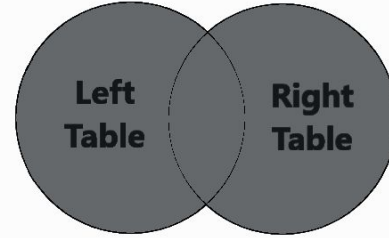


El INNER JOIN es el tipo de JOIN más utilizado.

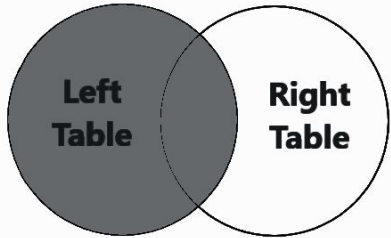
# Tipos de JOINS SQL



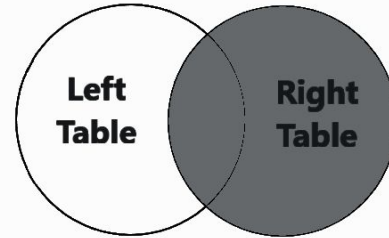
INTERSECCIÓN ( $A \cap B$ )



UNIÓN ( $A \cup B$ )



DIFERENCIA ( $A - B$ )



DIFERENCIA ( $B - A$ )

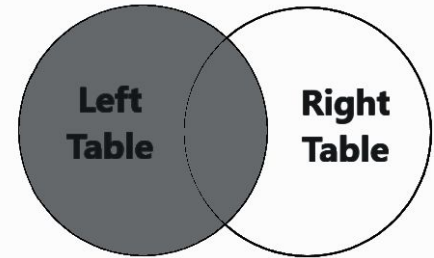
## Ejemplo INNER JOIN

SQL

```
SELECT movies.*, actors.first_name, actors.last_name  
FROM movies INNER JOIN actors  
ON movies.id = actors.favorite_movie_id;
```

# Ejemplo LEFT JOIN

```
SQL SELECT *  
FROM movies mo LEFT JOIN actors ac  
ON mo.id = ac.favorite_movie_id;
```



# **2 | GROUP BY**



# GROUP BY

- Agrupa los resultados según las columnas indicadas.
- Genera un solo registro por cada grupo de filas que compartan las columnas indicadas.
- Reduce la cantidad de filas de la consulta.
- Se suele utilizar en conjunto con **funciones de agregación**, para obtener **datos resumidos y agrupados** por las columnas que se necesiten.

## GROUP BY

Por ejemplo, si se tiene una tabla con los pagos realizados por cada persona y se quiere saber cuánto gastó cada una en total. ¿Cómo se podría realizar el reporte?

id	dni	fecha	pago
1	33.241.677	01/01/2017	50
2	35.186.928	02/01/2017	60
3	33.241.677	03/01/2017	70
4	35.186.928	04/01/2017	40

Tabla **clientes\_pagos** que almacena pagos de clientes

## GROUP BY

Si realizamos una consulta a la tabla **clientes\_pagos** de la siguiente forma:


```
SQL SELECT dni, SUM(pago) AS total  
FROM clientes_pagos  
GROUP BY dni;
```

# GROUP BY

¿Cómo se podría descomponer esta consulta?

**SELECT** dni

**SUM** (pago) = total



dni	total
35.186.928	100
33.241.677	120



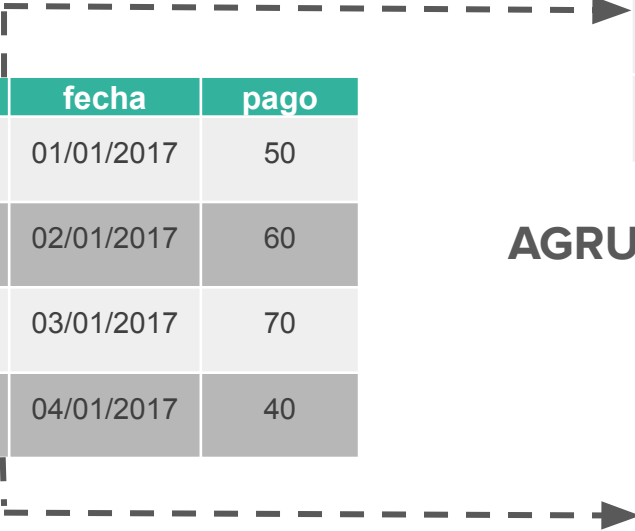
**GROUP BY** dni

*Resultado con los dni y montos pagados por cada cliente*

# GROUP BY

¿Cómo funciona para este caso? Agrupando por DNI, se crean grupos diferentes por cada DNI que exista en la tabla. En este ejemplo, existen dos grupos:

id	dni	fecha	pago
1	33.241.677	01/01/2017	50
2	35.186.928	02/01/2017	60
3	33.241.677	03/01/2017	70
4	35.186.928	04/01/2017	40



id	dni	fecha	pago
1	33.241.677	01/01/2017	50
3	33.241.677	03/01/2017	70

## AGRUPACIÓN

id	dni	fecha	pago
2	35.186.928	02/01/2017	60
4	35.186.928	04/01/2017	40

## GROUP BY

Sobre cada grupo, se aplica la función de agregación que se indicó en el SELECT. En este caso, se aplica la función de agregación SUM sobre la columna pago. El resultado de la consulta, es una tabla que contiene el resultado de cada grupo.

SUM			
id	dni	fecha	pago
1	33.241.677	01/01/2017	50
3	33.241.677	03/01/2017	70

SUM			
id	dni	fecha	pago
2	35.186.928	02/01/2017	60
4	35.186.928	04/01/2017	40

dni	total
35.186.928	100
33.241.677	120



**RESULTADO**

## Ejemplo: GROUP BY

Aquí podemos observar otro ejemplo de la sentencia GROUP BY con la BD movies.

SQL

```
SELECT COUNT(*), mo.title, mo.rating, mo.awards
FROM movies mo INNER JOIN actors ac
ON mo.id = ac.favorite_movie_id
GROUP BY title;
```

# 3 | HAVING



# HAVING

Es muy *similar* a la cláusula WHERE, pero en lugar de afectar a las filas de la tabla, afecta a los grupos obtenidos por el GROUP BY.

La cláusula **HAVING** se utiliza para incluir condiciones con algunas funciones SQL.



**WHERE** opera sobre registros individuales, mientras que **HAVING** lo hace sobre un grupo de registros.

# HAVING

Continuando con el ejemplo visto anteriormente, si se desea obtener solo los clientes que realizaron pagos totales superiores a 100.

dni	total
35.186.928	100
33.241.677	120



dni	total
35.186.928	100

**HAVING** total > 100

# HAVING

Si realizamos la consulta a la tabla **clientes\_pagos** podría ser de la siguiente forma:

SQL

```
SELECT dni, SUM(pago) AS total
FROM clientes_pagos
GROUP BY dni
HAVING total>100;
```

## Ejemplo: HAVING

Aquí podemos observar otro ejemplo de la sentencia HAVING con la BD movies.

```
SQL SELECT COUNT(*) AS tot_act, mo.title, mo.rating,  
      mo.awards  
      FROM movies mo INNER JOIN actors ac  
      ON mo.id = ac.favorite_movie_id  
      GROUP BY title HAVING tot_act > 2;
```

## WHERE - GROUP BY - HAVING

Para ver el orden de ejecución y diferencia entre estas cláusulas, sigamos este ejemplo.

Si se desea obtener solo las personas que realizaron pagos por un total superior a 100, pero considerando que cada compra individual haya sido superior a 50.

id	dni	fecha	pago
1	33.241.677	01/01/2017	50
2	35.186.928	02/01/2017	60
3	33.241.677	03/01/2017	70
4	35.186.928	04/01/2017	55

## WHERE - GROUP BY - HAVING

Si realizamos la consulta a la tabla **clientes\_pagos** podría ser de la siguiente forma:

```
SQL SELECT dni, SUM(pago) AS total
FROM clientes_pagos
WHERE pago>50
GROUP BY dni
HAVING total>100;
```

## WHERE - GROUP BY - HAVING

Resumiendo la ejecución, en primera instancia con WHERE se filtran los pagos mayores a 50, luego se realiza la agrupación (GROUP BY) por dni. Seguidamente, se aplica la función de agregación, sumando los pagos de cada agrupación por dni.

Por último, la cláusula HAVING filtra aquellos totales mayores a 100.



dni	total
35.186.928	115

**Resultado**

# WHERE - GROUP BY - HAVING

## ORDEN DE EJECUCIÓN





# 4 | Subconsultas

# ¿QUÉ SON LAS SUBCONSULTAS?

- SQL proporciona un mecanismo para las subconsultas anidadas.
- Una subconsulta es una expresión SELECT-FROM-WHERE que se anida dentro de otra consulta.
- Para resolver consultas complejas, se pueden utilizar estos anidamientos, obteniendo resultados basados en otros previos, que fueron obtenidos a través de otra consulta.

## Ejemplo: SUBCONSULTAS

Aquí podemos observar un ejemplo de Subconsulta con la BD movies.

SQL

```
SELECT *  
FROM actor_movie  
WHERE movie_id IN (SELECT id FROM movies WHERE  
rating=9.0);
```

# ALGUNAS BUENAS PRÁCTICAS

- Validar que los campos por el cual se realiza el JOIN sean del **mismo tipo de dato**.
- Evitar aplicar funciones sobre los campos por los cuales se realiza el JOIN.
- Revisar alias en los JOINS para evitar productos cartesianos en las consultas.
- Evitar **uso excesivo** de **JOINS**.
- Evitar el uso de subconsultas en tablas de gran volumen que no utilicen algún índice.
- Reescribir la consulta validando los JOINS. En ocasiones, es mejor utilizar un **DISTINCT** en lugar de un **GROUP BY**.

DigitalHouse>