

Bases de datos

Unidad Didáctica 04 - Lenguaje SQL - DML y TCL

Francisco Jesús Delgado Almirón



Ciclo formativo de grado superior
**Desarrollo de
aplicaciones web**

CONTENIDO

1.- Lenguaje de manipulación de datos - DML	2
1.1.- Insertar datos - INSERT INTO	2
1.2.- Borrar datos - DELETE	6
1.3.- Actualizar datos - UPDATE	7
1.3.1.- Borrado y actualización de datos en cascada	8
1.4.- Consultar datos - SELECT.....	10
1.4.1.- Cláusula ORDER BY	14
1.4.2.- Cláusula JOIN	16
1.4.3.- Cláusula GROUP BY y funciones agregadas	22
1.4.4.- Cláusulas TOP y DISTINCT.....	26
1.4.5.- Insertar datos en una tabla desde una consulta.....	28
1.5.- Comando GO	28
2.- Gestión de vistas.....	29
2.1.- Creación de vistas - CREATE VIEW	30
2.2.- Actualización de vistas - ALTER VIEW	31
2.3.- Borrado de vistas - DELETE VIEW.....	31
3.- Subconsultas - SUBSELECT	32
4.- Lenguaje de transacciones - TCL	37
4.1.- Transacciones.....	37
Bibliografía	40

1.- LENGUAJE DE MANIPULACIÓN DE DATOS - DML

El lenguaje de manipulación de datos (DML) proporciona operaciones que gestionan las peticiones de los usuarios, ofreciendo una forma de acceder y manipular los datos que éstos almacenan en una base de datos. Sus funciones comunes incluyen la inserción, la actualización y la recuperación de datos de la base de datos.

Aquí podemos ver la lista de sentencias del lenguaje DML:

- INSERT: Añade nuevos datos a la tabla de la base de datos existente.
- UPDATE: Cambia o actualiza los valores de la tabla.
- DELETE: Elimina registros o filas de la tabla.
- SELECT: Recupera datos de la tabla o de varias tablas.

Para todos los ejemplos que vamos a ver en esta unidad utilizaremos la base de datos de ventas que se encuentra en el fichero *BD_Ventas.sql*.

1.1.- INSERTAR DATOS - INSERT INTO

INSERT INTO es la sentencia DML para la inserción de registros en las tablas de base de datos. La sentencia básica es muy simple y puede realizarse de dos maneras:

```
1. --Opción 1
2. INSERT INTO nombreTabla (columna1, columna2...) VALUES (valor1, valor2, ...)
3.
4. --Opción 2
5. INSERT INTO nombreTabla VALUES (valor1, valor2, ...)
```

donde:

- nombreTabla: Es el nombre de la tabla en la que se va a insertar la fila.
- columna1, columna2, ...: Son las columnas en las que se van a insertar los valores. Deben especificarse en el mismo orden en el que aparecen los valores en la sección VALUES.
- valor1, valor2, ...: Son los valores que se insertarán en las columnas correspondientes. Estos valores deben coincidir en tipo de datos con las columnas definidas en la tabla.

La diferencia principal radica en que desde la primera opción podemos añadir los registros de acuerdo a únicamente las columnas que queremos insertar, mientras que la segunda opción nos obliga a insertar todos los valores de las columnas de la tabla de acuerdo a su posición, aunque sean a valor NULL.

Es más eficiente y recomendable utilizar la primera opción indicada, señalando únicamente las columnas que desean ser informadas.

Una vez tengamos nuestras tablas creadas usando la sentencia CREATE TABLE, podemos proceder a rellenarlas de nuevos registros desde la sentencia INSERT INTO.

En la tabla *cliente* insertamos los siguientes datos de clientes:

```
1. -- Insertar datos en la tabla cliente
2. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Aarón', 'Rivero', 'Gómez',
'Almería', 100);
3. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Adela', 'Salas', 'Díaz',
'Granada', 200);
4. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Adolfo', 'Rubio', 'Flores',
'Sevilla', NULL);
5. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Adrián', 'Suárez', NULL,
'Jaén', 300);
6. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Marcos', 'Loyola', 'Méndez',
'Almería', 200);
7. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('María', 'Santana', 'Moreno',
'Cádiz', 100);
8. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Pilar', 'Ruiz', NULL, 'Sevilla',
300);
9. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Pepe', 'Ruiz', 'Santana',
'Huelva', 200);
10. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Guillermo', 'López', 'Gómez',
'Granada', 225);
11. INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Daniel', 'Santana', 'Loyola',
'Sevilla', 125);
12. GO
```

Una vez insertados correctamente nuestra tabla *cliente* quedará de la siguiente forma:

Results		Messages				
	id	nombre	apellido1	apellido2	ciudad	categoria
1	1	Aarón	Rivero	Gómez	Almería	100
2	2	Adela	Salas	Díaz	Granada	200
3	3	Adolfo	Rubio	Flores	Sevilla	NULL
4	4	Adrián	Suárez	NULL	Jaén	300
5	5	Marcos	Loyola	Méndez	Almería	200
6	6	María	Santana	Moreno	Cádiz	100
7	7	Pilar	Ruiz	NULL	Sevilla	300
8	8	Pepe	Ruiz	Santana	Huelva	200
9	9	Guillermo	López	Gómez	Granada	225
10	10	Daniel	Santana	Loyola	Sevilla	125

Ilustración 1: Tabla cliente

En la tabla *comercial* insertamos los siguientes datos de comerciales:

```

1. -- Insertar datos en la tabla comercial
2. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Daniel', 'Sáez', 'Vega', 0.15);
3. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Juan', 'Gómez', 'López', 0.13);
4. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Diego', 'Flores', 'Salas', 0.11);
5. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Marta', 'Herrera', 'Gil', 0.14);
6. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Antonio', 'Carretero', 'Ortega',
0.12);
7. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Manuel', 'Domínguez', 'Hernández',
0.13);
8. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Antonio', 'Vega', 'Hernández', 0.11);
9. INSERT INTO comercial (nombre, apellido1, apellido2, comision) VALUES ('Alfredo', 'Ruiz', 'Flores', 0.05);
10. GO

```

Una vez insertados correctamente nuestra tabla *comercial* quedará de la siguiente forma:

Results		Messages			
	id	nombre	apellido1	apellido2	comision
1	1	Daniel	Sáez	Vega	0.15
2	2	Juan	Gómez	López	0.13
3	3	Diego	Flores	Salas	0.11
4	4	Marta	Herrera	Gil	0.14
5	5	Antonio	Carretero	Ortega	0.12
6	6	Manuel	Domínguez	Hernández	0.13
7	7	Antonio	Vega	Hernández	0.11
8	8	Alfredo	Ruiz	Flores	0.05

Ilustración 2: Tabla comercial

En la tabla *pedido* insertamos los siguientes datos de pedidos:

```

1. -- Insertar datos en la tabla pedido
2. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (150.5, '2017-10-05', 5, 2);
3. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (270.65, '2016-09-10', 1, 5);
4. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (65.26, '2017-10-05', 2, 1);
5. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (110.5, '2016-08-17', 8, 3);
6. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (948.5, '2017-09-10', 5, 2);
7. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (2400.6, '2016-07-27', 7, 1);
8. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (5760, '2015-09-10', 2, 1);
9. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (1983.43, '2017-10-10', 4, 6);
10. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (2480.4, '2016-10-10', 8, 3);
11. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (250.45, '2015-06-27', 8, 2);
12. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (75.29, '2016-08-17', 3, 7);
13. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (3045.6, '2017-04-25', 2, 1);
14. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (545.75, '2019-01-25', 6, 1);
15. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (145.82, '2017-02-02', 6, 1);
16. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (370.85, '2019-03-11', 1, 5);
17. INSERT INTO pedido (total, fecha, id_cliente, id_comercial) VALUES (2389.23, '2019-03-11', 1, 5);
18. GO

```

Una vez insertados correctamente nuestra tabla *pedido* quedará de la siguiente forma:

	id	total	fecha	id_cliente	id_comercial
1	1	150,5	2017-10-05	5	2
2	2	270,65	2016-09-10	1	5
3	3	65,26	2017-10-05	2	1
4	4	110,5	2016-08-17	8	3
5	5	948,5	2017-09-10	5	2
6	6	2400,6	2016-07-27	7	1
7	7	5760	2015-09-10	2	1
8	8	1983,43	2017-10-10	4	6
9	9	2480,4	2016-10-10	8	3
10	10	250,45	2015-06-27	8	2
11	11	75,29	2016-08-17	3	7
12	12	3045,6	2017-04-25	2	1
13	13	545,75	2019-01-25	6	1
14	14	145,82	2017-02-02	6	1
15	15	370,85	2019-03-11	1	5

Ilustración 3: Tabla pedido

Podemos insertar varias filas en una única instrucción INSERT INTO (soportado en versiones modernas de SQL Server):

```
1. INSERT INTO nombreTabla (columna1, columna2, columna3)
2. VALUES (valor1_1, valor1_2, valor1_3),
3.         (valor2_1, valor2_2, valor2_3),
4.         (valor3_1, valor3_2, valor3_3);
```

Ejemplo:

```
1. INSERT INTO empleado (id, nombre, apellido, salario)
2. VALUES (3, 'Laura', 'Martínez', 37000),
3.         (4, 'Pedro', 'López', 42000),
4.         (5, 'María', 'Rodríguez', 39000);
```

Reglas y consideraciones para la sentencia INSERT INTO

Es muy importante a la hora de realizar cualquier INSERT INTO, conocer muy bien como ha sido declarada la tabla junto a sus propiedades y restricciones para evitar errores fácilmente prevenibles.

Los tipos de datos en los valores proporcionados deben coincidir con los tipos de las columnas correspondientes.

Si una columna tiene una restricción de valor único (UNIQUE) o clave primaria (PRIMARY KEY), no podemos insertar un valor duplicado.

Si la tabla tiene columnas calculadas o valores por defecto (DEFAULT), no es necesario proporcionar un valor para ellas, a menos que queramos sobrescribir el valor predeterminado.

Restricciones

Es muy importante el control de los datos, especialmente en las tablas que tienen definidas claves **Foreign Key** para relacionarse a otras tablas. Y es que, al ejecutar un INSERT INTO, no solo debemos atender al tipo de dato y propiedades para escribir la sentencia, si no también habrá que dedicar una especial atención a las restricciones de la tabla y la coherencia de los datos.

1.2.- BORRAR DATOS - DELETE

DELETE es la sentencia DML para eliminar registros de las tablas de base de datos.

La sintaxis de la sentencia es sencilla:

```
1. DELETE FROM `nombreTabla` WHERE condiciones;
```

Es muy importante tener en cuenta la cláusula WHERE, para evitar que los cambios en la tabla se propaguen a todos los registros, eliminándolos en su totalidad.

En el caso de desear eliminar todos los registros de una tabla, es más óptimo utilizar la sentencia TRUNCATE, ya que es más rápida y eficiente, utilizando menos recursos del sistema y del registro de transacciones, además no elimina la estructura de la tabla ni sus definiciones (columnas, restricciones, índices, etc.). Hay que tener en cuenta que no se pueden truncar tablas que tengan claves foráneas activas.

La sintaxis de la sentencia TRUNCATE es la siguiente:

```
1. TRUNCATE TABLE nombre_de_tabla;
```

Ejemplos de borrado de datos

```
1. DELETE FROM cliente  
2. WHERE id = 5;
```

Explicación: Este comando eliminará de la tabla cliente al cliente con el id = 5.

```
1. DELETE FROM pedido  
2. WHERE id_cliente = 3  
3. AND fecha = '2023-01-01';
```

Explicación: Este comando eliminará todos los pedidos que pertenecen al cliente con id_cliente = 3 y que se realizaron en la fecha '2023-01-01'.

```
1. DELETE FROM comercial  
2. WHERE nombre = 'Juan'  
3. OR comision < 1000;
```

Explicación: Elimina todos los comerciales cuyo nombre sea 'Juan' o cuya comisión sea inferior a 1000.

```
1. DELETE FROM cliente
2. WHERE ciudad LIKE 'Madr%';
```

Explicación: Se eliminarán todos los clientes cuya ciudad comience con 'Madr', lo que podría incluir "Madrid", "Madrdejos", etc.

```
1. DELETE FROM pedido
2. WHERE fecha BETWEEN '2023-01-01' AND '2023-12-31';
```

Explicación: Se eliminarán todos los pedidos cuya fecha esté entre el 1 de enero de 2023 y el 31 de diciembre de 2023.

```
1. DELETE FROM cliente
2. WHERE (ciudad = 'Sevilla' OR ciudad = 'Córdoba')
3. AND categoria BETWEEN 1 AND 3
4. AND nombre LIKE 'A%';
```

Explicación: Este comando eliminará todos los clientes que vivan en Sevilla o Córdoba, cuya categoría esté entre 1 y 3, y cuyo nombre comience con 'A'. Esta combinación filtra clientes bastante específicos.

```
1. TRUNCATE TABLE pedido;
```

Explicación: Esto eliminará todos los registros de la tabla pedido, dejando la tabla vacía, pero manteniendo la estructura de la tabla y los índices intactos. A diferencia de DELETE, no se puede usar con condiciones ni se activan triggers.

1.3.- ACTUALIZAR DATOS - UPDATE

UPDATE es la sentencia DML que permite actualizar registros de base de datos. Dicho de otro modo, UPDATE permite modificar los valores de las columnas de las tablas respetando el tipo de dato correspondiente.

La sintaxis de la sentencia es sencilla:

```
1. UPDATE `nombreTabla` SET `nombreColumna` = nuevoValor WHERE condiciones;
```

Al igual que la sentencia DELETE, es muy importante tener en cuenta la cláusula WHERE, para evitar que los cambios en la tabla se propaguen a todos los registros.

Ejemplos de actualización de datos

```
1. UPDATE cliente
2. SET ciudad = 'Valencia'
3. WHERE id = 5;
```

Explicación: Este comando actualiza la ciudad del cliente con id = 5 y la cambia a "Valencia".


```
1. UPDATE pedido
2. SET total = total * 1.10
3. WHERE id_cliente = 3
4. AND fecha = '2023-01-01';
```

Explicación: Este comando incrementa en un 10% el total de los pedidos realizados por el cliente con `id_cliente = 3` en la fecha '2023-01-01'.

```
1. UPDATE comercial
2. SET comision = comision + 500
3. WHERE nombre = 'Juan'
4. OR comision < 2000;
```

Explicación: Este comando aumenta en 500 la comisión de los comerciales cuyo nombre es "Juan" o cuya comisión sea inferior a 2000.

```
1. UPDATE cliente
2. SET categoria = 2
3. WHERE apellido1 LIKE 'Mart%';
```

Explicación: Aquí se actualiza la categoría a 2 para todos los clientes cuyo primer apellido comience con "Mart" (por ejemplo, "Martínez" o "Martin").

```
1. UPDATE pedido
2. SET total = total * 0.90
3. WHERE fecha BETWEEN '2023-01-01' AND '2023-12-31';
```

Explicación: Este comando aplica un descuento del 10% en el total de todos los pedidos realizados entre el 1 de enero de 2023 y el 31 de diciembre de 2023.

```
1. UPDATE cliente
2. SET categoria = 1
3. WHERE (ciudad = 'Sevilla' OR ciudad = 'Málaga')
4. AND categoria BETWEEN 3 AND 5
5. AND nombre LIKE 'J%';
```

Explicación: En este ejemplo, se actualiza la categoría a 1 para los clientes que:

- Viven en Sevilla o Málaga.
- Tienen una categoría entre 3 y 5.
- Su nombre comienza con la letra 'J'.

1.3.1.- BORRADO Y ACTUALIZACIÓN DE DATOS EN CASCADA

Borrado en cascada (ON DELETE CASCADE)

El borrado en cascada se utiliza cuando queremos que, al eliminar un registro en la tabla padre (tabla referenciada), automáticamente se eliminen también los registros relacionados en la tabla hija (tabla que contiene la clave foránea). Esto es útil para mantener la consistencia referencial y evitar que queden registros "huérfanos" en la tabla hija.

Ejemplo de borrado en cascada

Sobre nuestra tabla Pedido, que tiene una clave foránea hacia la tabla cliente (la columna id_cliente), si configuramos el borrado en cascada, al eliminar un cliente, automáticamente se eliminarán todos sus pedidos.

```
1. -- Modificamos la tabla 'pedido' para agregar el borrado en cascada
2. ALTER TABLE pedido
3. DROP CONSTRAINT fk_pedido_cliente;
4.
5. ALTER TABLE pedido
6. ADD CONSTRAINT fk_pedido_cliente
7. FOREIGN KEY (id_cliente)
8. REFERENCES cliente(id)
9. ON DELETE CASCADE;
```

Explicación:

- Aquí eliminamos la restricción de clave foránea fk_pedido_cliente en la tabla pedido.
- Luego, volvemos a crear la clave foránea, pero ahora con la opción ON DELETE CASCADE.

Ahora, si ejecutamos la siguiente sentencia:

```
1. DELETE FROM cliente
2. WHERE id = 3;
```

Todos los pedidos asociados al cliente con id = 3 también serán eliminados automáticamente de la tabla pedido. Esto garantiza que no quedarán pedidos huérfanos.

Actualización en cascada (ON UPDATE CASCADE)

La actualización en cascada es similar, pero se utiliza para que cuando se actualiza una clave primaria en la tabla padre (tabla referenciada), las claves foráneas en las tablas hijas (tablas que contienen la clave foránea) también se actualicen automáticamente, manteniendo la consistencia de las relaciones.

Ejemplo de actualización en cascada

Supongamos que queremos que, si actualizamos el id de un cliente en la tabla cliente, todos los pedidos de ese cliente en la tabla pedido también se actualicen para reflejar la nueva id.

```
1. -- Modificamos la tabla 'pedido' para agregar la actualización en cascada
2. ALTER TABLE pedido
3. DROP CONSTRAINT fk_pedido_cliente;
4.
5. ALTER TABLE pedido
6. ADD CONSTRAINT fk_pedido_cliente
7. FOREIGN KEY (id_cliente)
8. REFERENCES cliente(id)
9. ON UPDATE CASCADE;
```

Explicación:

- Primero, eliminamos la clave foránea actual (fk_pedido_cliente) en la tabla pedido.
- Luego, la volvemos a crear, pero con la opción ON UPDATE CASCADE.

Ahora, si ejecutamos una sentencia como la siguiente:

```
1. UPDATE cliente
2. SET id = 10
3. WHERE id = 3;
```

La id del cliente con valor 3 será actualizada a 10 en la tabla cliente, y automáticamente todos los pedidos asociados a ese cliente también se actualizarán en la columna id_cliente en la tabla pedido.

Consideraciones sobre el uso de borrados y actualizaciones en cascada

- Sobre el borrado en cascada: Aunque útil, el borrado en cascada puede eliminar registros de múltiples tablas sin aviso si no se tiene cuidado.
- Sobre la actualización en cascada: Es relativamente raro actualizar claves primarias en una base de datos, ya que son campos que normalmente no cambian. Sin embargo, si necesitamos hacer esto, la actualización en cascada garantiza que las relaciones no se rompan.

1.4.- CONSULTAR DATOS - SELECT

Una instrucción SELECT, también conocida como query o consulta, es quizás el comando DML más popular y con más protagonismo en el trabajo con bases de datos. SELECT nos permite consultar los datos de una tabla.

La sintaxis básica es muy sencilla:

```
1. SELECT columna1, columna2... FROM nombreTabla
```

Comúnmente las sentencias SELECT van acompañadas después del FROM de la cláusula *WHERE*, que permite añadir filtros a la consulta para restringir o filtrar los registros a devolver. Existen además otras cláusulas después del WHERE que repercuten en cómo se visualizarán los registros y el orden en que se visualizarán, como por ejemplo el *GROUP BY* y *ORDER BY*.

Además, una SELECT permite devolver campos de varias tablas independientes pero relacionadas a través de una clave *Foreign Key* a través de la cláusula *JOIN*.

Dependiendo del modelo de la base de datos y los datos específicos que queramos devolver, una SELECT puede ser una sentencia muy sencilla de apenas una línea, o una muy compleja con muchos cruces de tablas y filtros específicos que debemos tener en cuenta para devolver la información requerida.

Algunos ejemplos sencillos de consulta son los siguientes:

```
1. SELECT *
2. FROM cliente;
```

Explicación: Esta consulta selecciona todos los datos de la tabla cliente. Usar SELECT * permite recuperar todas las columnas de la tabla sin especificarlas una a una. Este enfoque es útil cuando necesitamos ver todos los datos y no importa la cantidad de columnas. Sin embargo, en consultas de grandes bases de datos o cuando solo se necesitan algunas columnas, es más eficiente especificar las columnas necesarias para mejorar el rendimiento.

	id	nombre	apellido1	apellido2	ciudad	categoria
1	1	Aarón	Rivero	Gómez	Almería	100
2	2	Adela	Salas	Díaz	Granada	200
3	3	Adolfo	Rubio	Flores	Sevilla	NULL
4	4	Adrián	Suárez	NULL	Jaén	300
5	5	Marcos	Loyola	Méndez	Almería	200
6	6	María	Santana	Moreno	Cádiz	100
7	7	Pilar	Ruiz	NULL	Sevilla	300
8	8	Pepe	Ruiz	Santana	Huelva	200
9	9	Guillermo	López	Gómez	Granada	225
10	10	Daniel	Santana	Loyola	Sevilla	125

Ilustración 4: Resultado consulta 01

```
1. SELECT id, total, fecha
2. FROM pedido;
```

Explicación: Esta consulta selecciona solo tres columnas (id, total, fecha) de la tabla pedido. Es útil cuando necesitamos únicamente ciertos datos específicos de una tabla sin incluir todas sus columnas. Al limitar las columnas seleccionadas, la consulta suele ser más rápida y eficiente en términos de recursos, especialmente si la tabla tiene muchas columnas o si estás consultando una gran cantidad de datos.

	id	total	fecha
1	1	150,5	2017-10-05
2	2	270,65	2016-09-10
3	3	65,26	2017-10-05
4	4	110,5	2016-08-17
5	5	948,5	2017-09-10
6	6	2400,6	2016-07-27
7	7	5760	2015-09-10

Ilustración 5: Resultado consulta 02

Cláusula WHERE

La cláusula WHERE nos permite añadir filtros a nuestras consultas para devolver únicamente los registros que cumplan las condiciones especificadas del total del conjunto de resultados.

```
1. SELECT *
2. FROM cliente
3. WHERE ciudad = 'Granada';
```

Explicación: Esta consulta selecciona todos los clientes que viven en la ciudad de Granada. La cláusula WHERE establece una condición que debe cumplirse para que las filas sean incluidas en el resultado. En este caso, solo se seleccionan aquellos registros de la tabla cliente donde el valor de la columna ciudad es igual a 'Granada'. Esta cláusula ayuda a filtrar los datos para obtener solo los resultados que cumplen con el criterio indicado.

Results		Messages				
	id	nombre	apellido1	apellido2	ciudad	categoria
1	2	Adela	Salas	Díaz	Granada	200
2	9	Guillermo	López	Gómez	Granada	225

Ilustración 6: Resultado consulta 03

```
1. SELECT id, total, fecha
2. FROM pedido
3. WHERE total > 500;
```

Explicación: Esta consulta selecciona los pedidos cuyo total sea mayor a 500. Aquí, WHERE filtra los registros de la tabla pedido, devolviendo solo aquellos en los que el valor de la columna total es superior a 500. Este tipo de consulta es útil para obtener pedidos que cumplan ciertas condiciones, como un importe mínimo, sin necesidad de ver toda la tabla completa.

Results		Messages		
	id	total	fecha	
1	5	948,5	2017-09-10	
2	6	2400,6	2016-07-27	
3	7	5760	2015-09-10	
4	8	1983,43	2017-10-10	
5	9	2480,4	2016-10-10	
6	12	3045,6	2017-04-25	
7	13	545,75	2019-01-25	
8	16	2389,23	2019-03-11	

Ilustración 7: Resultado consulta 04

```
1. SELECT *
2. FROM cliente
3. WHERE ciudad = 'Almería' AND categoria = 100;
```

Explicación: Esta consulta selecciona todos los clientes que viven en Almería y pertenecen a la categoría 100. La condición AND requiere que ambas condiciones sean verdaderas para incluir el registro en el resultado. En este caso, solo se seleccionan clientes que cumplan simultáneamente con ambas condiciones: vivir en Almería y estar en la categoría 100.

Results		Messages				
	id	nombre	apellido1	apellido2	ciudad	categoria
1	1	Aarón	Rivero	Gómez	Almería	100

Ilustración 8: Resultado consulta 05

```
1. SELECT *
2. FROM comercial
3. WHERE comision < 0.12 OR apellido2 IS NULL;
```

Explicación: Esta consulta selecciona todos los comerciales que tienen una comisión menor a 0.12 o que no tienen un segundo apellido (apellido2 es NULL). La condición OR permite que cualquiera de las dos condiciones sea verdadera para incluir el registro en el resultado. Esto es útil cuando se desea incluir registros que cumplan con al menos una de las condiciones establecidas.

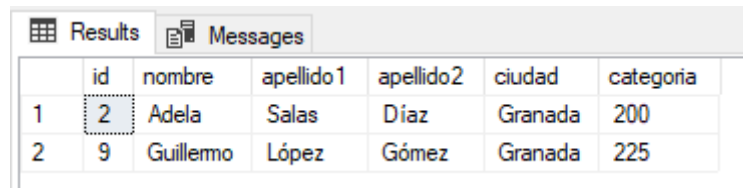


	id	nombre	apellido1	apellido2	comision
1	3	Diego	Flores	Salas	0.11
2	7	Antonio	Vega	Hernández	0.11
3	8	Alfredo	Ruiz	Flores	0.05

Ilustración 9: Resultado consulta 06

```
1. SELECT *
2. FROM cliente
3. WHERE ciudad = 'Granada' AND (categoria = 200 OR categoria = 225);
```

Explicación: Esta consulta selecciona los clientes que viven en Madrid y pertenecen a la categoría 1 o 3. Al combinar AND y OR, se crean condiciones más complejas. Aquí, la condición ciudad = 'Granada' debe ser verdadera, y, además, el cliente debe pertenecer a la categoría 200 o 225. Para asegurar el orden correcto de evaluación, se usan paréntesis, lo que garantiza que primero se evalúe la condición entre paréntesis (categoria = 200 OR categoria = 225) y luego se combine con ciudad = 'Granada'.

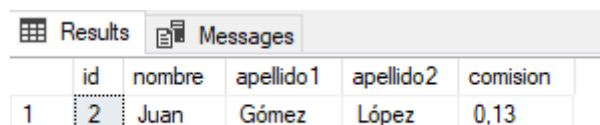


	id	nombre	apellido1	apellido2	ciudad	categoria
1	2	Adela	Salas	Díaz	Granada	200
2	9	Guillermo	López	Gómez	Granada	225

Ilustración 10: Resultado consulta 07

```
1. SELECT *
2. FROM comercial
3. WHERE apellido1 LIKE 'G%';
```

Explicación: Esta consulta selecciona todos los comerciales cuyo primer apellido (apellido1) comienza con "G". El operador LIKE se utiliza para realizar búsquedas de patrones, y el símbolo % actúa como un comodín que permite cualquier número de caracteres después de "G". Esto es útil para búsquedas parciales y patrones específicos en las columnas de texto.



	id	nombre	apellido1	apellido2	comision
1	2	Juan	Gómez	López	0.13

Ilustración 11: Resultado consulta 08

```
1. SELECT *
2. FROM cliente
3. WHERE ciudad = 'Sevilla'
4.   AND (categoria > 200 OR categoria IS NULL)
5.   AND apellido2 LIKE 'F%';
```

Explicación:

Esta consulta selecciona todos los clientes que cumplen con las siguientes condiciones:

- Viven en SEVILLA.
- Están en la categoría 200 o nula.
- Tienen el segundo apellido que comienza con "F".

La combinación de AND, OR y LIKE permite filtrar resultados con mucha precisión:

- La condición ciudad = 'Sevilla' se combina con (categoria = 200 OR categoria IS NULL) usando AND, por lo que ambas condiciones deben cumplirse.
- LIKE 'F%' se aplica con AND al apellido2, lo que hace que solo se seleccionen aquellos clientes el segundo apellido que comience con "F" y que además vivan en Sevilla y estén en la categoría 200 o nula.

Results

Messages

	id	nombre	apellido1	apellido2	ciudad	categoria
1	3	Adolfo	Rubio	Flores	Sevilla	NULL

Ilustración 12: Resultado consulta 09

1.4.1.- CLÁUSULA ORDER BY

La cláusula ORDER BY nos permite ordenar un conjunto de resultados devueltos por una SELECT a través de un campo de manera ascendente o descendente. La cláusula ORDER BY siempre se escribirá al final de la sentencia.

Ordenación descendente: Los registros se ordenarán de valor más alto a valor más bajo, siendo los más altos los primeros registros del conjunto de resultados.

```
1. SELECT *
2. FROM comercial
3. ORDER BY comision DESC;
```

Explicación: Esta consulta selecciona todos los registros de la tabla comercial y los ordena en función de la columna comision, de mayor a menor (orden descendente). El operador DESC especifica que los datos deben organizarse en orden decreciente, mostrando primero los comerciales con comisiones más altas. Esto es útil cuando queremos ver a los empleados con las comisiones más altas en la parte superior.



	id	nombre	apellido1	apellido2	comision
1	1	Daniel	Sáez	Vega	0,15
2	4	Marta	Herrera	Gil	0,14
3	2	Juan	Gómez	López	0,13
4	6	Manuel	Domínguez	Hernández	0,13
5	5	Antonio	Carretero	Ortega	0,12
6	3	Diego	Flores	Salas	0,11
7	7	Antonio	Vega	Hernández	0,11
8	8	Alfredo	Ruiz	Flores	0,05

Ilustración 13: Resultado consulta 10

Ordenación ascendente: Los registros se ordenarán de valor más bajo a valor más alto, siendo los más bajos los primeros registros del conjunto de resultados.

```
1. SELECT *  
2. FROM pedido  
3. ORDER BY fecha ASC;
```

Explicación: Esta consulta selecciona todos los registros de la tabla pedido y los ordena en función de la columna fecha, de la más antigua a la más reciente (orden ascendente). El operador ASC organiza los datos en orden creciente, por lo que en este caso se muestran los pedidos comenzando desde la fecha más antigua. Esto es útil para analizar el historial de pedidos en orden cronológico.



	id	nombre	apellido1	apellido2	comision
1	1	Daniel	Sáez	Vega	0,15
2	4	Marta	Herrera	Gil	0,14
3	2	Juan	Gómez	López	0,13
4	6	Manuel	Domínguez	Hernández	0,13
5	5	Antonio	Carretero	Ortega	0,12
6	3	Diego	Flores	Salas	0,11
7	7	Antonio	Vega	Hernández	0,11
8	8	Alfredo	Ruiz	Flores	0,05

Ilustración 14: Resultado consulta 11

Ordenación combinada: Permite organizar los datos jerárquicamente, con prioridades de orden en varios niveles. Esto es útil para análisis más complejo

```
1. SELECT *  
2. FROM cliente  
3. ORDER BY ciudad ASC, categoria DESC;
```

Explicación:

Esta consulta selecciona todos los registros de la tabla cliente y los ordena en dos niveles:

- Primero, por ciudad en orden ascendente (ASC), agrupando a los clientes por ciudad en orden alfabético.

- Luego, dentro de cada ciudad, ordena a los clientes por categoría en orden descendente (DESC), mostrando primero a los clientes de categoría más alta.

	id	nombre	apellido1	apellido2	ciudad	categoría
1	5	Marcos	Loyola	Méndez	Almería	200
2	1	Aarón	Rivero	Gómez	Almería	100
3	6	María	Santana	Moreno	Cádiz	100
4	9	Guillermo	López	Gómez	Granada	225
5	2	Adela	Salas	Díaz	Granada	200
6	8	Pepe	Ruiz	Santana	Huelva	200
7	4	Adrián	Suárez	NULL	Jaén	300
8	7	Pilar	Ruiz	NULL	Sevilla	300
9	10	Daniel	Santana	Loyola	Sevilla	125
10	3	Adolfo	Rubio	Flores	Sevilla	NULL

Ilustración 15: Resultado consulta 12

1.4.2.- CLÁUSULA JOIN

El modelo relacional nos permite ejecutar consultas sobre varias tablas relacionadas, de manera que podamos extraer e integrar la información de varias tablas desde una sola ejecución. Esto es posible gracias a la cláusula JOIN.

INNER JOIN

Es el JOIN más común. De hecho, si en nuestra SELECT solo indicamos la palabra JOIN, el sistema lo interpretará automáticamente como INNER JOIN. Esta cláusula establece una relación entre dos tablas a través de un campo común de la tabla principal y de la tabla secundaria devolviendo únicamente los registros que tengan valores coincidentes en ambas tablas.

Se pueden encadenar varios JOIN para establecer varias relaciones desde una única sentencia.

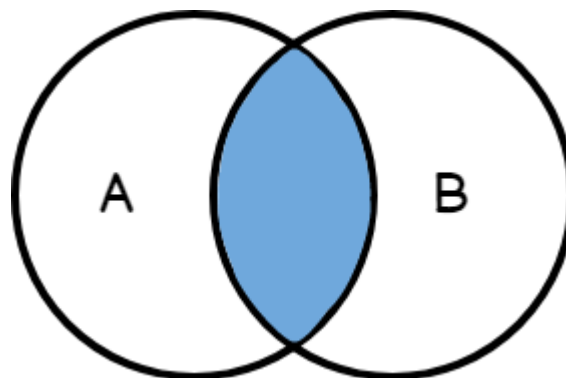


Ilustración 16: INNER JOIN

Fuente de la imagen:

<https://sqllearning.com/es/ejecucion-consultas/join/>

La sintaxis básica de la sentencia INNER JOIN es la siguiente:

```
1. SELECT tabla1.columna, tabla2.columna
2. FROM tabla1
3. INNER JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;
```

El INNER JOIN se utiliza para combinar filas de dos o más tablas que tienen una columna en común. Solo devuelve las filas en las que existe coincidencia en ambas tablas según la condición de emparejamiento en la cláusula ON. Se utiliza con las claves foráneas.

Ejemplo 1: Consulta para listar los pedidos junto con el nombre del cliente que realiza cada pedido.

```
1. SELECT pedido.id AS id_pedido, pedido.total, pedido.fecha, cliente.nombre AS nombre_cliente, cliente.apellido1
2. FROM pedido
3. INNER JOIN cliente ON pedido.id_cliente = cliente.id;
```

Explicación: Esta consulta utiliza INNER JOIN para combinar la tabla pedido con la tabla cliente, vinculándolas mediante la columna id_cliente de pedido y la columna id de cliente. Solo se seleccionarán aquellos registros donde el id_cliente en pedido coincida con un id en cliente.

En el resultado, se muestran:

- El id del pedido (pedido.id) renombrado como id_pedido.
- El total y fecha del pedido.
- El nombre y apellido1 del cliente que realizó cada pedido.

Esto permite ver los detalles de cada pedido junto con los datos del cliente que lo realizó.

	id_pedido	total	fecha	nombre_cliente	apellido1
1	1	150,5	2017-10-05	Marcos	Loyola
2	2	270,65	2016-09-10	Aarón	Rivero
3	3	65,26	2017-10-05	Adela	Salas
4	4	110,5	2016-08-17	Pepe	Ruiz
5	5	948,5	2017-09-10	Marcos	Loyola
6	6	2400,6	2016-07-27	Pilar	Ruiz

Ilustración 17: Resultado consulta 13

Ejemplo 2: Consulta para listar los pedidos con el nombre del comercial y del cliente involucrados en cada pedido.

```
1. SELECT pedido.id AS id_pedido, pedido.total, cliente.nombre AS nombre_cliente, comercial.nombre AS
   nombre_comercial
2. FROM pedido
3. INNER JOIN cliente ON pedido.id_cliente = cliente.id
4. INNER JOIN comercial ON pedido.id_comercial = comercial.id;
```

Explicación:

En esta consulta se realiza un INNER JOIN múltiple:

- Primero, se hace un INNER JOIN entre pedido y cliente para obtener el nombre del cliente asociado a cada pedido (a través de id_cliente en pedido e id en cliente).
- Luego, se realiza un INNER JOIN entre pedido y comercial para obtener el nombre del comercial que gestionó cada pedido (a través de id_comercial en pedido e id en comercial).

El resultado muestra:

- El id y el total de cada pedido.
- El nombre del cliente (cliente.nombre) y el nombre del comercial (comercial.nombre) asociado a cada pedido.

	id_pedido	total	nombre_cliente	nombre_comercial
1	1	150,5	Marcos	Juan
2	2	270,65	Aarón	Antonio
3	3	65,26	Adela	Daniel
4	4	110,5	Pepe	Diego
5	5	948,5	Marcos	Juan

Ilustración 18: Resultado consulta 14

Ejemplo 3: Listar los pedidos de clientes en una ciudad específica, gestionados por comerciales con comisión superior al 10%.

```
1. SELECT pedido.id AS id_pedido, pedido.total, pedido.fecha, cliente.nombre AS nombre_cliente, cliente.ciudad,
comercial.nombre AS nombre_comercial, comercial.comision
2. FROM pedido
3. INNER JOIN cliente ON pedido.id_cliente = cliente.id
4. INNER JOIN comercial ON pedido.id_comercial = comercial.id
5. WHERE cliente.ciudad = 'Granada' AND comercial.comision > 0.1;
```

Explicación:

- En esta consulta
 - INNER JOIN se utiliza para combinar las tablas pedido, cliente, y comercial:
 - Se relacionan pedido y cliente usando `pedido.id_cliente = cliente.id`, lo que permite acceder a los datos del cliente que realizó cada pedido.
 - Se combinan pedido y comercial usando `pedido.id_comercial = comercial.id`, para acceder a los datos del comercial que gestionó cada pedido.
 - Cláusula WHERE:
 - Se filtran los resultados para mostrar solo los pedidos en los que el cliente reside en Granada (`cliente.ciudad = 'Granada'`).
 - También se incluye la condición de que el comercial tenga una comisión superior a 0.1 (`comercial.comision > 0.1`).

El resultado de la consulta muestra:

- El id, total y fecha del pedido.
- El nombre y ciudad del cliente.
- El nombre y comisión del comercial.

Esta combinación de INNER JOIN y WHERE permite seleccionar solo aquellos pedidos que cumplen condiciones específicas tanto en la tabla de clientes como en la de comerciales, proporcionando un resultado más enfocado.

	id_pedido	total	fecha	nombre_cliente	ciudad	nombre_comercial	comision
1	3	65,26	2017-10-05	Adela	Granada	Daniel	0,15
2	7	5760	2015-09-10	Adela	Granada	Daniel	0,15
3	12	3045,6	2017-04-25	Adela	Granada	Daniel	0,15

Ilustración 19: Resultado consulta 15

LEFT JOIN

Esta cláusula devuelve todos los registros de la tabla izquierda más aquellos que intersecan ambas tablas.

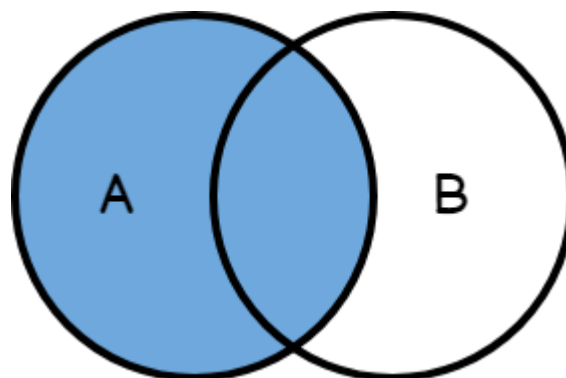


Ilustración 20: LEFT JOIN

Fuente de la imagen: <https://sqllearning.com/es/ejecucion-consultas/join/>

La sintaxis básica de la sentencia LEFT JOIN es la siguiente:

```
1. SELECT tabla1.columna, tabla2.columna
2. FROM tabla1
3. LEFT JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;
```

Ejemplo 1: Mostrar un listado de todos los clientes junto con los pedidos que tienen pedidos registrados. De los pedidos mostrar el total del pedido, la fecha del mismo y el comercial que lo atendió. Puede darse el caso de que un cliente no tenga pedidos a su nombre.

```
1. SELECT
2.     cliente.nombre AS NombreCliente,
3.     cliente.apellido1 AS ApellidoCliente,
4.     pedido.total AS TotalPedido,
5.     pedido.fecha AS FechaPedido,
6.     comercial.nombre AS NombreComercial
7. FROM cliente
8. LEFT JOIN pedido ON cliente.id = pedido.id_cliente
9. LEFT JOIN comercial ON pedido.id_comercial = comercial.id;
```

Explicación: Se utiliza LEFT JOIN entre la tabla cliente y la tabla pedido para asegurarnos de que todos los clientes aparezcan, incluso aquellos que no tienen pedidos (sus columnas relacionadas tendrán valores NULL). También se hace un segundo LEFT JOIN con la tabla comercial para incluir información del comercial asociado a cada pedido. El resultado mostrará detalles de clientes junto con sus pedidos (si existen) y el nombre del comercial correspondiente.

	NombreCliente	ApellidoCliente	TotalPedido	FechaPedido	NombreComercial
11	María	Santana	545,75	2019-01-25	Daniel
12	María	Santana	145,82	2017-02-02	Daniel
13	Pilar	Ruiz	2400,6	2016-07-27	Daniel
14	Pepe	Ruiz	110,5	2016-08-17	Diego
15	Pepe	Ruiz	2480,4	2016-10-10	Diego
16	Pepe	Ruiz	250,45	2015-06-27	Juan
17	Guillermo	López	NULL	NULL	NULL
18	Daniel	Santana	NULL	NULL	NULL

Ilustración 21: Resultado consulta LEFT JOIN 1

Ejemplo 2: Mostrar un listado de todos los comerciales junto con la suma total de los pedidos que gestionaron. Puede darse el caso de que haya comerciales que no hayan realizado pedidos todavía.

```
1. SELECT
2.     comercial.nombre AS NombreComercial,
3.     comercial.apellido1 AS ApellidoComercial,
4.     SUM(pedido.total) AS SumaTotalPedidos
5. FROM comercial
6. LEFT JOIN pedido ON comercial.id = pedido.id_comercial
7. GROUP BY comercial.nombre, comercial.apellido1;
```

Explicación: Se utiliza LEFT JOIN para garantizar que todos los comerciales aparezcan, incluso aquellos que no gestionaron ningún pedido. La función SUM(p.total) suma los totales de los pedidos gestionados por cada comercial. Si un comercial no tiene pedidos, el resultado será NULL. Se usa GROUP BY para agrupar los resultados por cada comercial y obtener un registro único por cada uno.

	NombreComercial	ApellidoComercial	SumaTotalPedidos
1	Antonio	Carretero	3030,73
2	Manuel	Domínguez	1983,43
3	Diego	Flores	2590,9
4	Juan	Gómez	1349,45
5	Marta	Herrera	NULL
6	Alfredo	Ruiz	NULL
7	Daniel	Sáez	11963,03
8	Antonio	Vega	75,29

Ilustración 22: Resultado consulta LEFT JOIN 2

RIGHT JOIN

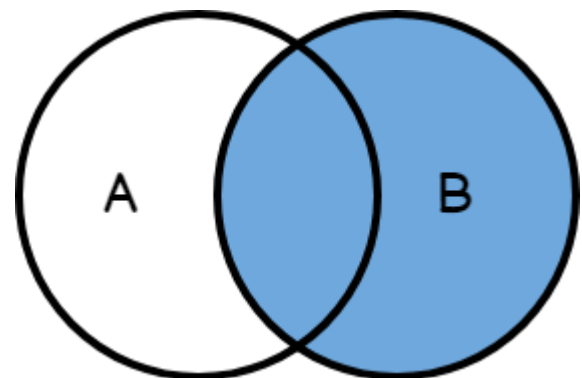


Ilustración 23: RIGHT JOIN

Esta cláusula devuelve todos los registros de la tabla derecha más aquellos que intersecan ambas tablas.

Fuente de la imagen: <https://sqllearning.com/es/ejecucion-consultas/join/>

La sintaxis básica de la sentencia RIGHT JOIN es la siguiente:

```
1. SELECT tabla1.columna, tabla2.columna
2. FROM tabla1
3. RIGHT JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;
```

Ejemplo 1: Mostrar un listado con todos los pedidos registrados junto con el comercial que los gestionó. Si un pedido no tiene un comercial asignado (lo cual es poco probable pero hipotéticamente posible), aún debe aparecer.

```
1. SELECT
2.     pedido.id AS IDPedido,
3.     pedido.total AS TotalPedido,
4.     pedido.fecha AS FechaPedido,
5.     comercial.nombre AS NombreComercial,
6.     comercial.apellido1 AS ApellidoComercial
7. FROM comercial
8. RIGHT JOIN pedido ON comercial.id = pedido.id_comercial;
```

Explicación: El RIGHT JOIN asegura que todos los registros de la tabla pedido estén presentes en el resultado, independientemente de si hay un comercial asignado. Si un pedido no tiene un comercial asociado (es decir, no hay coincidencia en la tabla comercial), las columnas relacionadas con el comercial mostrarán NULL. Esto es útil para analizar casos en los que los pedidos puedan estar mal asociados o faltan datos de comerciales.

	IDPedido	TotalPedido	FechaPedido	NombreComercial	ApellidoComercial
1	1	150,5	2017-10-05	Juan	Gómez
2	2	270,65	2016-09-10	Antonio	Carretero
3	3	65,26	2017-10-05	Daniel	Sáez
4	4	110,5	2016-08-17	Diego	Flores
5	5	948,5	2017-09-10	Juan	Gómez
6	6	2400,6	2016-07-27	Daniel	Sáez
7	7	5760	2015-09-10	Daniel	Sáez
8	8	1983,43	2017-10-10	Manuel	Domínguez

Ilustración 24: Resultado consulta RIGHT JOIN 1

FULL JOIN

Esta cláusula devuelve tanto los registros de la tabla izquierda y derecha, independientemente si intersecan o no.

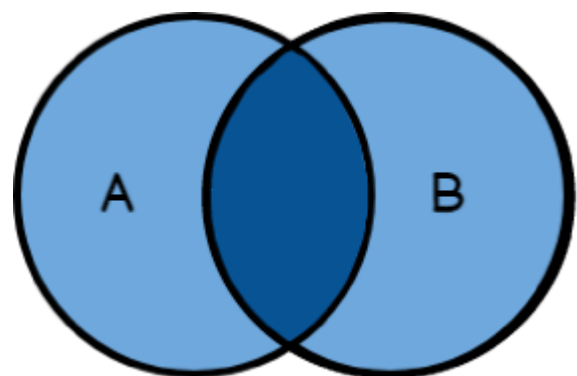


Ilustración 25: FULL JOIN

Fuente de la imagen: <https://sqllearning.com/es/ejecucion-consultas/join/>

La sintaxis básica de la sentencia FULL JOIN es la siguiente:

```
1. SELECT tabla1.columna, tabla2.columna
2. FROM tabla1
3. FULL JOIN tabla2 ON tabla1.columna_relacionada = tabla2.columna_relacionada;
```

Ejemplo 1: Mostrar un listado completo que incluya todos los clientes y sus pedidos. Si un cliente no tiene pedidos, igualmente debe aparecer. Por otro lado, si existe un pedido sin cliente, ese pedido también debe aparecer.

```
1. SELECT
2.     cliente.nombre AS NombreCliente,
3.     cliente.apellido1 AS ApellidoCliente,
4.     pedido.id AS IDPedido,
5.     pedido.total AS TotalPedido,
6.     pedido.fecha AS FechaPedido
7. FROM cliente
8. FULL JOIN pedido ON cliente.id = pedido.id_cliente;
```

Explicación:

- Se utiliza FULL JOIN para combinar todas las filas de las tablas cliente y pedido.
- Las filas que tienen coincidencias en ambas tablas se combinan.
- Los clientes que no tienen pedidos aparecen con valores NULL en las columnas de pedido.
- Los pedidos que no están asociados a un cliente (por ejemplo, si hay problemas de integridad referencial) aparecen con valores NULL en las columnas de cliente.

	NombreCliente	ApellidoCliente	IDPedido	TotalPedido	FechaPedido
11	Maria	Santana	13	545,75	2019-01-25
12	Maria	Santana	14	145,82	2017-02-02
13	Pilar	Ruiz	6	2400,6	2016-07-27
14	Pepe	Ruiz	4	110,5	2016-08-17
15	Pepe	Ruiz	9	2480,4	2016-10-10
16	Pepe	Ruiz	10	250,45	2015-06-27
17	Guillermo	López	NULL	NULL	NULL
18	Daniel	Santana	NULL	NULL	NULL

Ilustración 26: Resultado consulta FULL JOIN 1

1.4.3.- CLÁUSULA GROUP BY Y FUNCIONES AGREGADAS

La cláusula GROUP BY se utiliza para agrupar los resultados de acuerdo a una columna seleccionada, devolviendo una fila de resultados para cada grupo de dicha columna. GROUP BY se utiliza a menudo junto a funciones agregadas.

En cuanto a la sintaxis, la cláusula GROUP BY se sitúa al final de la sentencia después del WHERE, pero antes del ORDER BY en caso de aplicarse ambas cláusulas.

```
1. SELECT columna1, columna2, ..., función_agregada(columna)
2. FROM tabla
3. WHERE condiciones
4. GROUP BY columna1, columna2, ...;
```

donde:

- `columna1, columna2, ...`: Son las columnas que queremos agrupar. Deben aparecer en la lista de selección (SELECT) si no están dentro de una función de agregado.
- `función_agregada(columna)`: Función de agregado aplicada a una columna, como `SUM(columna)`, `COUNT(columna)`, `AVG(columna)`, etc.
- `tabla`: La tabla de la cual estás seleccionando los datos.
- `condiciones`: (Opcional) Filtro para seleccionar las filas antes de agrupar.

Las funciones de agregado utilizadas con GROUP BY permiten realizar cálculos sobre un conjunto de valores en una columna. Las funciones agregadas más comunes son:

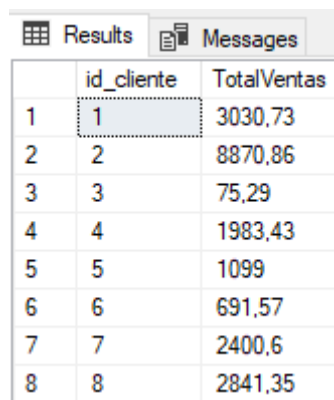
Función SUM

Calcula la suma total de valores en una columna numérica.

Ejemplo: Obtener el total de ventas por cliente. Esta consulta calcula el total de ventas (total) de cada cliente en la tabla pedido. Agrupamos por `id_cliente` para obtener la suma de todos los pedidos de cada cliente.

```
1. SELECT id_cliente, SUM(total) AS TotalVentas
2. FROM pedido
3. GROUP BY id_cliente;
```

Explicación: Para cada `id_cliente`, la consulta suma todos los valores en la columna total. Esto muestra cuánto ha gastado cada cliente en total.



	id_cliente	TotalVentas
1	1	3030,73
2	2	8870,86
3	3	75,29
4	4	1983,43
5	5	1099
6	6	691,57
7	7	2400,6
8	8	2841,35

Ilustración 27: Resultado consulta 16

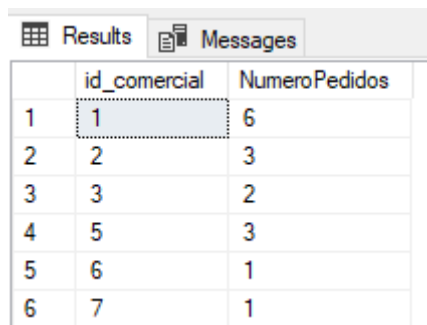
Función COUNT

Cuenta el número de filas en una columna. Puede contar todas las filas o solo las filas con valores distintos de NULL.

Ejemplo: Obtener el número de pedidos que ha hecho cada comercial. Esta consulta cuenta cuántos pedidos ha gestionado cada comercial. Usamos `COUNT(*)` para contar todas las filas asociadas a cada `id_comercial`.

```
1. SELECT id_comercial, COUNT(*) AS NumeroPedidos
2. FROM pedido
3. GROUP BY id_comercial;
```


Explicación: Para cada id_comercial, la consulta cuenta el número de filas en la tabla pedido. Así podemos ver cuántos pedidos ha gestionado cada comercial.



	id_comercial	NumeroPedidos
1	1	6
2	2	3
3	3	2
4	5	3
5	6	1
6	7	1

Ilustración 28: Resultado consulta 17

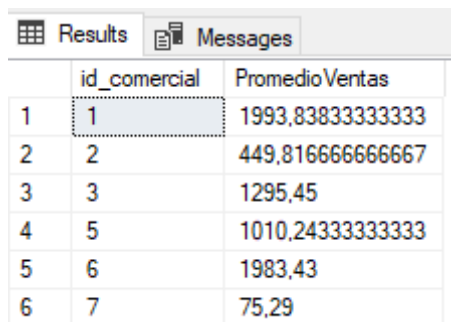
Función AVG

Calcula el promedio (media aritmética) de los valores en una columna numérica, ignorando los NULL.

Ejemplo: Obtener la media de las ventas que ha hecho cada comercial. Esta consulta calcula el promedio de ventas por cada comercial. Agrupamos por id_comercial y usamos AVG(total) para obtener el promedio de total.

```
1. SELECT id_comercial, AVG(total) AS PromedioVentas
2. FROM pedido
3. GROUP BY id_comercial;
```

Explicación: Para cada id_comercial, la consulta calcula el promedio de total. Esto muestra el valor medio de los pedidos que cada comercial ha gestionado.



	id_comercial	PromedioVentas
1	1	1993,838333333333
2	2	449,816666666667
3	3	1295,45
4	5	1010,243333333333
5	6	1983,43
6	7	75,29

Ilustración 29: Resultado consulta 18

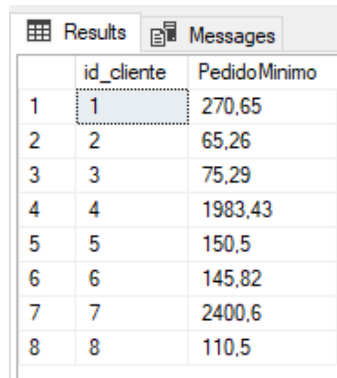
Función MIN

Devuelve el valor mínimo en una columna. Funciona con columnas numéricas, de texto y de fecha.

Ejemplo: Obtener el pedido más pequeño que ha hecho cada uno de los clientes. Esta consulta devuelve el valor mínimo del total de cada pedido de cada cliente. Usamos MIN(total) para encontrar el pedido de menor valor por cada id_cliente.

```
1. SELECT id_cliente, MIN(total) AS PedidoMinimo
2. FROM pedido
3. GROUP BY id_cliente;
```

Explicación: Para cada id_cliente, la consulta encuentra el valor mínimo de total. Esto indica el pedido de menor valor realizado por cada cliente.



	id_cliente	PedidoMinimo
1	1	270,65
2	2	65,26
3	3	75,29
4	4	1983,43
5	5	150,5
6	6	145,82
7	7	2400,6
8	8	110,5

Ilustración 30: Resultado consulta 19

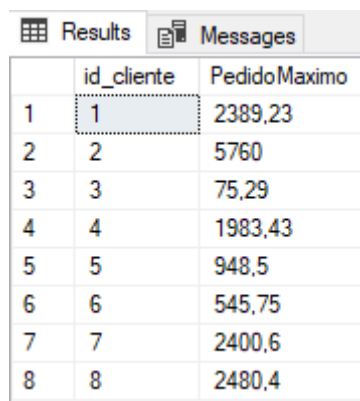
Función MAX

Devuelve el valor máximo en una columna. Similar a MIN, funciona con valores numéricos, de texto y de fecha.

Ejemplo: Obtener el pedido más grande que ha hecho cada uno de los clientes. Esta consulta devuelve el valor máximo del total de cada pedido por cliente, utilizando MAX(total) para encontrar el pedido de mayor valor por cada id_cliente.

```
1. SELECT id_cliente, MAX(total) AS PedidoMaximo
2. FROM pedido
3. GROUP BY id_cliente;
```

Explicación: Para cada id_cliente, la consulta obtiene el valor máximo de total, es decir, el pedido de mayor valor que cada cliente ha realizado.



	id_cliente	PedidoMaximo
1	1	2389,23
2	2	5760
3	3	75,29
4	4	1983,43
5	5	948,5
6	6	545,75
7	7	2400,6
8	8	2480,4

Ilustración 31: Resultado consulta 20

Ejemplo de consulta con agrupación de 2 columnas: Obtener el número de pedidos y el total de ventas por que ha hecho cada comercial por cliente.

```
1. SELECT
2.     id_cliente,
3.     id_comercial,
4.     COUNT(*) AS NumeroPedidos,
5.     SUM(total) AS TotalVentas
6. FROM pedido
7. GROUP BY id_cliente, id_comercial;
```

Explicación:

- COUNT(*): Cuenta el número de pedidos realizados entre cada cliente y comercial.
- SUM(total): Calcula el total de ventas para cada combinación cliente-comercial.
- Esta consulta proporciona tanto el total de pedidos como la cantidad de ventas que se han realizado entre cada par de cliente y comercial.

	id_cliente	id_comercial	NumeroPedidos	TotalVentas
1	2	1	3	8870,86
2	6	1	2	691,57
3	7	1	1	2400,6
4	5	2	2	1099
5	8	2	1	250,45
6	9	2	2	2500,6

Ilustración 32: Resultado consulta 21

Uso de GROUP BY con HAVING

Podemos filtrar después de agrupar, utilizamos para ello la cláusula HAVING en lugar de WHERE.

Ejemplo: Obtener los clientes que tienen gastando más de 1000€ comprando.

```
1. SELECT id_cliente, SUM(total) AS TotalVentas
2. FROM pedido
3. GROUP BY id_cliente
4. HAVING SUM(total) > 1000;
```

	id_cliente	TotalVentas
1	1	3030,73
2	2	8870,86
3	4	1983,43
4	5	1099
5	7	2400,6
6	8	2841,35

Ilustración 33: Resultado consulta 22

1.4.4.- CLÁUSULAS TOP Y DISTINCT

Existen ciertas cláusulas que generalmente se sitúan en el SELECT antes del FROM y ayudan en el filtrado de registros. Estas cláusulas son a menudo conocidas como predicados y en esta sección veremos su funcionamiento.

Cláusula TOP

TOP limita el número de registros que devuelve la consulta a la cantidad especificada en la cláusula.

Ejemplo: Obtener los 5 pedidos de mayor valor.

En este ejemplo, usamos TOP para seleccionar los 5 pedidos con el total de venta más alto.

```
1. SELECT TOP 5
2.     pedido.id,
3.     pedido.total,
4.     pedido.fecha,
5.     cliente.nombre AS NombreCliente,
6.     cliente.apellido1 AS ApellidoCliente,
7.     comercial.nombre AS NombreComercial,
8.     comercial.apellido1 AS ApellidoComercial
9. FROM pedido
10. INNER JOIN cliente ON pedido.id_cliente = cliente.id
11. INNER JOIN comercial ON pedido.id_comercial = comercial.id
12. ORDER BY pedido.total DESC;
```

Explicación:

- TOP 5: Limita el resultado a las 5 primeras filas.
- ORDER BY total DESC: Ordena los resultados de forma descendente por el campo total, de modo que los pedidos con valores más altos aparezcan primero.
- Esta consulta devuelve solo los 5 pedidos con el valor de venta más alto, junto con su fecha, cliente y comercial.

	id	total	fecha	NombreCliente	ApellidoCliente	NombreComercial	ApellidoComercial
1	7	5760	2015-09-10	Adela	Salas	Daniel	Sáez
2	12	3045,6	2017-04-25	Adela	Salas	Daniel	Sáez
3	9	2480,4	2016-10-10	Pepe	Ruiz	Diego	Flores
4	6	2400,6	2016-07-27	Pilar	Ruiz	Daniel	Sáez
5	16	2389,23	2019-03-11	Aarón	Rivero	Antonio	Carretero

Ilustración 34: Resultado consulta 23

Cláusula DISTINCT

DISTINCT omite los registros cuyos campos seleccionados contienen datos duplicados.

Ejemplo: Obtener las ciudades de los clientes.

Esta consulta usa DISTINCT para obtener una lista única de ciudades donde residen los clientes, sin duplicados.

```
1. SELECT DISTINCT ciudad
2. FROM cliente;
```

Explicación:

- DISTINCT ciudad: Elimina las filas duplicadas en la columna ciudad, de modo que cada ciudad aparezca solo una vez en los resultados.
- Esto proporciona una lista única de todas las ciudades en las que reside al menos un cliente.



	ciudad
1	Almería
2	Cádiz
3	Granada
4	Huelva
5	Jaén
6	Sevilla

Ilustración 35: Resultado consulta 24

1.4.5.- INSERTAR DATOS EN UNA TABLA DESDE UNA CONSULTA

Podemos insertar datos en una tabla desde otra tabla usando una instrucción SELECT:

```
1. INSERT INTO nombre_tabla_destino (columna1, columna2, columna3)
2. SELECT columna1, columna2, columna3
3. FROM nombre_tabla_origen
4. WHERE condición;
```

Es útil para transferir datos entre tablas o copiar información filtrada.

Ejemplo:

```
1. INSERT INTO empleados_backup (id, nombre, apellido, salario)
2. SELECT id, nombre, apellido, salario
3. FROM empleado
4. WHERE salario > 40000;
```

1.5.- COMANDO GO

GO es un comando utilizado por las herramientas de cliente de SQL Server como SQL Server Management Studio (SSMS) o el SQLCMD. Sirve para indicar el final de un bloque o lote de instrucciones SQL y señalar al servidor que ejecute ese bloque antes de continuar con las siguientes instrucciones.

Función de GO

- Separa grupos de instrucciones SQL en lotes para que el servidor las ejecute como una unidad.
- Permite crear bloques lógicos dentro de un script. Por ejemplo, después de un GO, el servidor ejecuta todas las instrucciones que hay antes de esa línea como un lote separado.
- Puede ser necesario cuando, por ejemplo, creas una tabla y luego necesitas usarla inmediatamente en el mismo script. SQL Server necesita procesar la creación de la tabla antes de que se pueda hacer referencia a ella, y GO asegura esto al forzar la ejecución.

El comando GO no pertenece a ninguno de los sublenguajes estándar de SQL, como DDL (Lenguaje de Definición de Datos), DML (Lenguaje de Manipulación de Datos), DCL (Lenguaje de Control de Datos) o TCL (Lenguaje de Control de Transacciones). En lugar de eso, GO es un comando específico de herramientas cliente de SQL Server, como SQLCMD y SQL Server Management Studio (SSMS).

Ejemplos de uso de GO

Separar DDL y DML.

```
1. CREATE TABLE ejemplo (  
2.     id INT PRIMARY KEY,  
3.     nombre VARCHAR(50)  
4. );  
5. GO  
6.  
7. INSERT INTO ejemplo (id, nombre) VALUES (1, 'Ejemplo');  
8. GO
```

En este ejemplo, el GO después de la creación de la tabla asegura que el servidor ejecute la instrucción CREATE TABLE antes de intentar el INSERT.

Usar después de crear una base de datos.

```
1. CREATE DATABASE mi_base_de_datos;  
2. GO  
3.  
4. USE mi_base_de_datos;  
5. GO
```

Aquí, el GO después de crear la base de datos asegura que esa instrucción se ejecute y la base de datos esté disponible antes de intentar usarla.

2.- GESTIÓN DE VISTAS

Una VIEW podemos definirla como una tabla virtual con su conjunto de filas y columnas, cuyo contenido está definido por una consulta SELECT sobre una o más tablas. Esto quiere decir, que los datos que contienen no están almacenados realmente en la vista, si no que se generan de manera dinámica de acuerdo a la consulta que ejecuten.

Las VIEW se utilizan a menudo como filtros de las tablas de las que recogen datos y es por ello que se utilizan a menudo como mecanismos de seguridad, ya que permiten al usuario tener acceso de visibilidad de los datos, pero no a las tablas reales subyacentes. También se utilizan a menudo para simplificar y personalizar la visualización de los datos para el usuario.

Generalmente no pueden utilizar ORDER BY a no ser que vayan precedidas de la cláusula TOP.

La SELECT que define la VIEW puede provenir también de otras vistas del sistema.

En SQL Server, existen diferentes tipos de vistas:

1. Vistas de usuario: Tablas virtuales definidas por el usuario cuyo contenido está definido por una consulta.
2. Vistas indexadas: Cuyos datos se han almacenado como una tabla real ya que se han indizado a través de un índice clúster único. Este tipo de vistas mejoran mucho el rendimiento en algunos tipos de consultas que devuelven muchos registros.
3. Vistas con particiones: Combinan datos horizontales con particiones de un conjunto de tablas miembro en uno o más servidores.
4. Vistas del sistema: Específicas para consultar metadatos del sistema.

2.1.- CREACIÓN DE VISTAS - CREATE VIEW

CREATE VIEW es la sentencia DDL para la creación de vistas.

La sintaxis es la siguiente:

```
1. CREATE VIEW nombreVista
2. AS SELECT ...
```

Ejemplo: Crear una vista que muestre el id, total, y fecha de los pedidos junto con los nombres y apellidos del cliente y del comercial asociados a cada pedido.

```
1. CREATE VIEW datos_pedidos AS
2. SELECT pedido.id AS pedido_id, pedido.total AS total_pedido, pedido.fecha AS fecha_pedido,
   CONCAT(cliente.nombre, ' ', cliente.apellido1, ' ', ISNULL(cliente.apellido2, '')) AS nombre_completo_cliente,
   CONCAT(comercial.nombre, ' ', comercial.apellido1, ' ', ISNULL(comercial.apellido2, '')) AS
   nombre_completo_comercial
3. FROM pedido
4. INNER JOIN cliente ON pedido.id_cliente = cliente.id
5. INNER JOIN comercial ON pedido.id_comercial = comercial.id;
```

Para ejecutar la nueva VIEW, basta con hacer una SELECT sobre ella:

```
1. SELECT * FROM datos_pedidos;
```

	pedido_id	total_pedido	fecha_pedido	nombre_completo_cliente	nombre_completo_comercial
1	1	150,5	2017-10-05	Marcos Loyola Méndez	Juan Gómez López
2	2	270,65	2016-09-10	Aarón Rivero Gómez	Antonio Carretero Ortega
3	3	65,26	2017-10-05	Adela Salas Díaz	Daniel Sáez Vega
4	4	110,5	2016-08-17	Pepe Ruiz Santana	Diego Flores Salas
5	5	948,5	2017-09-10	Marcos Loyola Méndez	Juan Gómez López
6	6	2400,6	2016-07-27	Pilar Ruiz	Daniel Sáez Vega
7	7	5760	2015-09-10	Adela Salas Díaz	Daniel Sáez Vega
8	8	1983,43	2017-10-10	Adrián Suárez	Manuel Domínguez Hernández
9	9	2480,4	2016-10-10	Pepe Ruiz Santana	Diego Flores Salas
10	10	250,45	2015-06-27	Pepe Ruiz Santana	Juan Gómez López
11	11	75,29	2016-08-17	Adolfo Rubio Flores	Antonio Vega Hernández

Ilustración 36: Datos de la vista datos_pedidos

Importante: La cláusula ORDER BY no es válida en vistas, funciones insertadas, tablas derivadas, subconsultas ni expresiones de tabla común, salvo que se especifique también TOP, OFFSET o FOR XML.

2.2.- ACTUALIZACIÓN DE VISTAS - ALTER VIEW

ALTER VIEW, es la sentencia DDL para la actualización de vistas.

La sintaxis es la siguiente:

```
1. ALTER VIEW nombreVista  
2. AS SELECT ...
```

Ejemplo: Modificar la vista datos_pedidos para que solo muestre la información completa de los pedidos cuyo total sea mayor o igual a 500€.

```
1. ALTER VIEW datos_pedidos AS  
2. SELECT pedido.id AS pedido_id, pedido.total AS total_pedido, pedido.fecha AS fecha_pedido,  
   CONCAT(cliente.nombre, ' ', cliente.apellido1, ' ', ISNULL(cliente.apellido2, '')) AS nombre_completo_cliente,  
   CONCAT(comercial.nombre, ' ', comercial.apellido1, ' ', ISNULL(comercial.apellido2, '')) AS  
   nombre_completo_comercial  
3. FROM pedido  
4. INNER JOIN cliente ON pedido.id_cliente = cliente.id  
5. INNER JOIN comercial ON pedido.id_comercial = comercial.id  
6. WHERE pedido.total >= 500;
```

Podemos ver al ejecutar un SELECT sobre la vista datos_pedidos que ya solo muestra los datos de los pedidos con un total mayor o igual a 500€.

	pedido_id	total_pedido	fecha_pedido	nombre_completo_cliente	nombre_completo_comercial
1	5	948,5	2017-09-10	Marcos Loyola Méndez	Juan Gómez López
2	6	2400,6	2016-07-27	Pilar Ruiz	Daniel Sáez Vega
3	7	5760	2015-09-10	Adela Salas Díaz	Daniel Sáez Vega
4	8	1983,43	2017-10-10	Adrián Suárez	Manuel Domínguez Hernández
5	9	2480,4	2016-10-10	Pepe Ruiz Santana	Diego Flores Salas
6	12	3045,6	2017-04-25	Adela Salas Díaz	Daniel Sáez Vega
7	13	545,75	2019-01-25	María Santana Moreno	Daniel Sáez Vega
8	16	2389,23	2019-03-11	Aarón Rivero Gómez	Antonio Carretero Ortega

Ilustración 37: Datos de vista datos_pedidos con pedidos con total >= 500

2.3.- BORRADO DE VISTAS - DELETE VIEW

DROP VIEW, es la sentencia DDL para la eliminación de vistas.

La sintaxis es la siguiente:

```
1. DROP VIEW nombreVista;
```


Pasemos ahora a eliminar nuestra VIEW: datos_pedidos

```
1. DROP VIEW datos_pedidos;
```

3.- SUBCONSULTAS - SUBSELECT

Una SUBSELECT o subconsulta, es una consulta anidada dentro de otra consulta SELECT, SUBSELECT o bien otras sentencias SQL como INSERT, UPDATE o DELETE. En este apartado nos vamos a centrar en las SUBSELECT dentro de las sentencias SELECT.

Una SELECT puede contener internamente otra SELECT en alguna de sus cláusulas. Esta SELECT interna encerrada entre paréntesis es la denominada SUBSELECT.

Las SUBSELECT ayudan en el filtrado de datos y atienden a las siguientes características y recomendaciones:

- Deben ir siempre entre paréntesis.
- Generalmente no pueden utilizar ORDER BY a no ser que vayan precedidas de la cláusula TOP.
- Suelen ir precedidas de cláusulas ALL, IN, NOT IN, EXISTS, NOT EXISTS.
- No es recomendable utilizar campos calculados ya que pueden ralentizar la consulta.

Las subconsultas se pueden situar en varios sitios.

Subconsultas en la cláusula SELECT

Una SUBSELECT, puede ubicarse en el listado de columnas de la expresión SELECT para devolver el resultado como un campo más del conjunto de resultados.

Ejemplo: Obtener una lista de los clientes junto con el número de pedidos que han realizado.

```
1. SELECT
2.     id AS cliente_id,
3.     nombre AS cliente_nombre,
4.     apellido1 AS cliente_apellido,
5.     ciudad,
6.     categoria,
7.     (SELECT COUNT(*)
8.      FROM pedido
9.      WHERE pedido.id_cliente = cliente.id) AS numero_pedidos
10. FROM cliente;
```

Esta subconsulta cuenta cuántos pedidos ha realizado cada cliente. El resultado de esta subconsulta se incluye como una columna adicional en la salida principal.

	cliente_id	cliente_nombre	cliente_apellido	ciudad	categoria	numero_pedidos
1	1	Aarón	Rivero	Almería	100	3
2	2	Adela	Salas	Granada	200	3
3	3	Adolfo	Rubio	Sevilla	NULL	1
4	4	Adrián	Suárez	Jaén	300	1
5	5	Marcos	Loyola	Almería	200	2
6	6	María	Santana	Cádiz	100	2
7	7	Pilar	Ruiz	Sevilla	300	1
8	8	Pepe	Ruiz	Huelva	200	3
9	9	Guillermo	López	Granada	225	0
10	10	Daniel	Santana	Sevilla	125	0

Ilustración 38: Resultado subconsulta 01

Subconsultas en la cláusula FROM

La cláusula FROM admite en lugar de la definición de una tabla, una SUBSELECT que devuelva resultados filtrados.

Ejemplo: Obtener los totales de ventas por cada cliente.

```
1. SELECT
2.     cliente.id AS cliente_id,
3.     cliente.nombre AS cliente_nombre,
4.     cliente.apellido1 AS cliente_apellido,
5.     cliente.ciudad,
6.     cliente.categoria,
7.     ventas.total_ventas
8. FROM cliente
9. LEFT JOIN
10.    (SELECT id_cliente, SUM(total) AS total_ventas
11.      FROM pedido
12.     GROUP BY id_cliente
13.    ) ventas
14. ON cliente.id = ventas.id_cliente;
```

Explicación

- Subconsulta en la cláusula FROM:
 - (SELECT id_cliente, SUM(total) AS total_ventas FROM pedido GROUP BY id_cliente) calcula el total de ventas por cliente en la tabla pedido.
 - La subconsulta retorna dos columnas: id_cliente y total_ventas.
- LEFT JOIN:
 - La subconsulta se une con la tabla cliente para incluir la información de ventas en la consulta principal.
 - Utilizamos un LEFT JOIN para asegurarnos de que todos los clientes aparezcan en el resultado, incluso si no tienen pedidos.

	cliente_id	cliente_nombre	cliente_apellido	ciudad	categoria	total_ventas
1	1	Aarón	Rivero	Almería	100	3030,73
2	2	Adela	Salas	Granada	200	8870,86
3	3	Adolfo	Rubio	Sevilla	NULL	75,29
4	4	Adrián	Suárez	Jaén	300	1983,43
5	5	Marcos	Loyola	Almería	200	1099
6	6	María	Santana	Cádiz	100	691,57
7	7	Pilar	Ruiz	Sevilla	300	2400,6
8	8	Pepe	Ruiz	Huelva	200	2841,35
9	9	Guillermo	López	Granada	225	NULL
10	10	Daniel	Santana	Sevilla	125	NULL

Ilustración 39: Resultado subconsulta 02

Prueba a cambiar LEFT JOIN por INNER JOIN, ¿qué ocurre?

Subconsultas en la cláusula WHERE

En la cláusula WHERE, es sin duda el lugar donde a menudo son utilizadas más SUBSELECT para realizar comparaciones y filtrar resultados.

Ejemplo 1: Obtener los clientes cuya categoría es mayor que la categoría de todos los clientes de la ciudad de "Huelva".

```

1. SELECT id, nombre, apellido1, ciudad, categoria
2. FROM cliente
3. WHERE categoria > ALL (
4.     SELECT categoria
5.     FROM cliente
6.     WHERE ciudad = 'Huelva'
7. );

```

Explicación:

- ALL: Compara la categoría de un cliente con todas las categorías de los clientes en "Huelva".
- El cliente será incluido si su categoría es mayor que todas las categorías de los clientes de "Huelva".
- Si la subconsulta retorna categorías [300, NULL], las filas con categorías superiores a 300 serán seleccionadas (se ignora NULL en las comparaciones).

	id	nombre	apellido1	ciudad	categoria
1	4	Adrián	Suárez	Jaén	300
2	7	Pilar	Ruiz	Sevilla	300
3	9	Guillermo	López	Granada	225

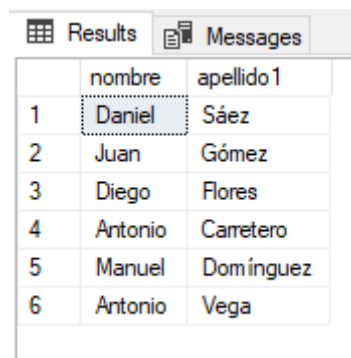
Ilustración 40: Resultado subconsulta 03

Ejemplo 2: Obtener los nombres de los comerciales que han gestionado algún pedido.

```
1. SELECT nombre, apellido1
2. FROM comercial
3. WHERE id IN (
4.     SELECT DISTINCT id_comercial
5.     FROM pedido
6. );
```

Explicación:

- IN: Busca si el id del comercial aparece en el conjunto de id_comercial de la tabla pedido.
- La subconsulta devuelve una lista de identificadores únicos de los comerciales que han gestionado al menos un pedido.



	nombre	apellido1
1	Daniel	Sáez
2	Juan	Gómez
3	Diego	Flores
4	Antonio	Carretero
5	Manuel	Domínguez
6	Antonio	Vega

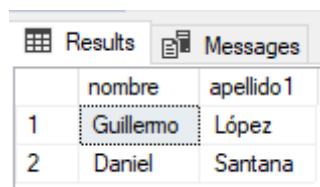
Ilustración 41: Resultado subconsulta 04

Ejemplo 3: Obtener los nombres de los clientes que no tienen ningún pedido.

```
1. SELECT nombre, apellido1
2. FROM cliente
3. WHERE id NOT IN (
4.     SELECT id_cliente
5.     FROM pedido
6. );
```

Explicación:

- NOT IN: Excluye los clientes cuyo id está presente en la lista de id_cliente de la tabla pedido.
- Devuelve todos los clientes que no aparecen en ningún pedido.



	nombre	apellido1
1	Guillermo	López
2	Daniel	Santana

Ilustración 42: Resultado subconsulta 05

Ejemplo 4: Obtener los clientes que han realizado al menos un pedido.

```
1. SELECT id, nombre, apellido1
2. FROM cliente c
3. WHERE EXISTS (
4.     SELECT *
5.     FROM pedido p
6.     WHERE p.id_cliente = c.id
7. );
```

Explicación:

- EXISTS: Evalúa si la subconsulta devuelve al menos una fila.
- La subconsulta verifica si existe al menos un pedido relacionado con el cliente actual.
- Si se encuentra al menos un pedido, el cliente se incluye en los resultados.



The screenshot shows a database interface with a 'Results' tab. It displays a table with 8 rows and 4 columns: 'id', 'nombre', and 'apellido1'. The first row is highlighted with a dashed border. The data is as follows:

	id	nombre	apellido1
1	1	Aarón	Rivero
2	2	Adela	Salas
3	3	Adolfo	Rubio
4	4	Adrián	Suárez
5	5	Marcos	Loyola
6	6	María	Santana
7	7	Pilar	Ruiz
8	8	Pepe	Ruiz

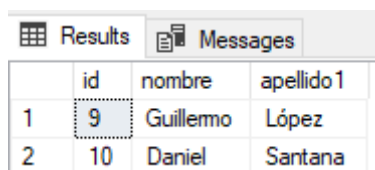
Ilustración 43: Resultado subconsulta 06

Ejemplo 5: Obtener los clientes que no han realizado ningún pedido.

```
1. SELECT id, nombre, apellido1
2. FROM cliente c
3. WHERE NOT EXISTS (
4.     SELECT *
5.     FROM pedido p
6.     WHERE p.id_cliente = c.id
7. );
```

Explicación:

- NOT EXISTS: Evalúa si la subconsulta no devuelve filas.
- La subconsulta verifica si no existe ningún pedido relacionado con el cliente actual.
- Si no hay pedidos, el cliente se incluye en los resultados.



The screenshot shows a database interface with a 'Results' tab. It displays a table with 2 rows and 4 columns: 'id', 'nombre', and 'apellido1'. The first row is highlighted with a dashed border. The data is as follows:

	id	nombre	apellido1
1	9	Guillermo	López
2	10	Daniel	Santana

Ilustración 44: Resultado subconsulta 07

4.- LENGUAJE DE TRANSACCIONES - TCL

El lenguaje de control de transacciones (TCL) gestiona las transacciones dentro de una base de datos. Las transacciones agrupan un conjunto de tareas relacionadas en una única tarea ejecutable. Todas las tareas deben tener éxito para que la transacción funcione.

4.1.- TRANSACCIONES

Una transacción puede definirse como un conjunto de instrucciones que funcionan como una unidad, donde si una parte falla, todo el proceso falla en su totalidad, revirtiéndose todos los cambios producidos en la base de datos.

Para declarar una transacción partimos de la siguiente sintaxis:

```
1. BEGIN TRANSACTION
```

También será válida la sentencia acortada:

```
1. BEGIN TRAN
```

Una transacción, por tanto, valida todas las instrucciones que contiene como un único conjunto o unidad única de trabajo. Si todas las instrucciones funcionan, los cambios sobre la base de datos se confirman haciéndose permanentes. Si cualquier instrucción falla, todos los cambios deben revertirse.

A menudo y, por tanto, es muy común que las transacciones contengan internamente estructuras de gestión de errores como **TRY...CATH**, para confirmar o revertir los cambios en base de datos. Si el código falla, se capturará el error en el CATCH y procederemos a revertir todos los cambios sobre la base de datos. Si todas las instrucciones del TRY funcionan correctamente, se procederá a guardar los cambios.

TRY...CATCH es una estructura de control de flujo que nos permite detectar y atrapar errores. El funcionamiento esencial es simple:

- **TRY**: Encierra una serie de instrucciones Transact SQL que intentarán ejecutarse.
- **CATCH**: Recoge cualquier error que se produzca en el TRY, ejecutándose las sentencias propias de este bloque CATCH, que generalmente estarán destinadas a proporcionar información acerca del error producido.

Su sintaxis es la siguiente:

```
1. BEGIN TRY
2.     [codigo]
3. END TRY
4. BEGIN CATCH
5.     [codigo]
6. END CATCH
```

Funciones ERROR

Existen una serie de funciones encargadas de devolver información acerca de los errores producidos, muy utilizadas a menudo dentro del bloque CATCH:

- `ERROR_NUMBER()`: Devuelve el número del error.
- `ERROR_SEVERITY()`: Devuelve la gravedad del error.
- `ERROR_STATE()`: Devuelve el número de estado de error.
- `ERROR_PROCEDURE()`: Devuelve el nombre del procedimiento almacenado o disparador donde ocurrió el error.
- `ERROR_LINE()`: Devuelve el número de línea dentro de la rutina que provocó el error.
- `ERROR_MESSAGE()`: Devuelve el texto completo del mensaje de error.

Ejemplo de TRY...CATCH

Para ejemplificar el funcionamiento de TRY...CATCH de manera sencilla, vamos a intentar dividir un número entre cero dentro de un bloque TRY...CATCH para capturar y devolver información del error:

```
1. BEGIN TRY
2.     SELECT 1/0;
3. END TRY
4. BEGIN CATCH
5.     SELECT
6.         ERROR_NUMBER() AS ErrorNumber,
7.         ERROR_SEVERITY() AS ErrorSeverity,
8.         ERROR_STATE() AS ErrorState,
9.         ERROR_PROCEDURE() AS ErrorProcedure,
10.        ERROR_LINE() AS ErrorLine,
11.        ERROR_MESSAGE() AS ErrorMessage;
12. END CATCH;
13. GO
```

	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	8134	16	1	NULL	2	Error de división entre cero.

Ilustración 45: Error capturado en bloque TRY CATCH

Dentro del bloque TRY...CATCH, **THROW** nos permite generar nuestras propias excepciones y devolverlas en el bloque CATCH. De esta manera, podremos manejar una gestión de errores personalizada.

La sintaxis es la siguiente:

```
1. THROW (error_number | variable,
2.        message | variable ,
3.        state | variable )
```

Como vemos en su sintaxis, la instrucción **THROW** utiliza tres argumentos importantes en su declaración:

- `error_number`: Indica el número del error. Será de tipo de dato INT y su valor deberá estar comprendido entre 50000 y 2147483647 (ambos incluidos).
- `message`: Devuelve el texto completo del mensaje de error que especifiquemos.
- `state`: Devuelve el número de estado de error. Será de tipo TINYINT con un valor comprendido entre 0 y 255 (ambos incluidos).

Ejemplo de THROW

Para ejemplificar el funcionamiento de THROW de manera sencilla, vamos volver a intentar dividir un número entre cero dentro de un bloque TRY...CATCH para capturar y devolver la información personalizada del error generado:

```
1. BEGIN TRY
2.     SELECT 1/0;
3. END TRY
4. BEGIN CATCH
5.     THROW 50100, 'La operación matemática es errónea.',1;
6. END CATCH;
7. GO
```

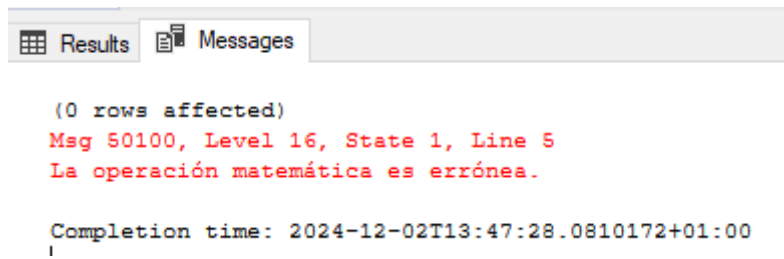


Ilustración 46: Ejemplo de excepción personalizada con THROW

Para entender bien el funcionamiento de una transacción y los conceptos de COMMIT, ROLLBACK y gestión de errores, supongamos que queremos añadir dos nuevos clientes dentro de una transacción:

```
1. BEGIN TRANSACTION;
2.     BEGIN TRY
3.         INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Fernando', 'López',
'Ayala', 'Sevilla', 2000);
4.         INSERT INTO cliente (nombre, apellido1, apellido2, ciudad, categoria) VALUES ('Sandra', NULL,
'Córcega', 'Almería', 2001);
5.     COMMIT TRANSACTION;
6.     END TRY
7.     BEGIN CATCH
8.         SELECT
9.             ERROR_NUMBER() AS ErrorNumber,
10.            ERROR_SEVERITY() AS ErrorSeverity,
11.            ERROR_STATE() AS ErrorState,
12.            ERROR_PROCEDURE() AS ErrorProcedure,
13.            ERROR_LINE() AS ErrorLine,
14.            ERROR_MESSAGE() AS ErrorMessage;
15.         ROLLBACK TRANSACTION
16.     END CATCH;
```

Si ejecutamos el código de arriba, vemos que nos devuelve el siguiente mensaje de error:

	ErrorNumber	ErrorSeverity	ErrorState	ErrorProcedure	ErrorLine	ErrorMessage
1	515	16	2	NULL	4	No se puede insertar el valor NULL en la columna 'apellido1', tabla 'ventas.dbo.cliente'. La columna no admite valores NULL. Error de INSERT.

Ilustración 47: Error en transacción

En este caso, aunque el primer INSERT INTO del registro 'Fernando López Ayala' funcione correctamente, al fallar el segundo, capturamos el error en el CATCH y hacemos ROLLBACK de la transacción. Por tanto, si consultamos la tabla cliente podemos comprobar que no se ha insertado ningún registro.

BIBLIOGRAFÍA

Calasan, M. (2023, January 13). ¿Qué son DDL, DML, DQL y DCL en SQL? LearnSQL.es; LearnSQL.com. <https://learnsql.es/blog/que-son-ddl-dml-dql-y-dcl-en-sql/>

Tipos de lenguajes de bases de datos y sus usos (con ejemplos). (2022, December 1). Historiadelaempresa.com. <https://historiadelaempresa.com/lenguajes-de-bases-de-datos>

Sentencia INSERT INTO en SQL. (2022, September 6). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/creacion-gestion-bases-datos/insert/>

Sentencia DELETE en SQL. (2022, September 19). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/creacion-gestion-bases-datos/delete/>

Sentencia UPDATE en SQL. (2022, September 19). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/creacion-gestion-bases-datos/update/>

Cláusula JOIN en SQL. (2022, September 13). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/join/>

Cláusula ORDER BY en SQL. (2022, September 15). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/order-by/>

Sentencia SELECT en SQL. (2022, September 12). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/select/>

Ejecución de consultas SQL. (n.d.). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. Retrieved October 28, 2024, from <https://sqllearning.com/es/ejecucion-consultas/>

Cláusulas TOP, ALL y DISTINCT en SQL. (2022, September 15). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/top-all-distinct/>

Sentencia GROUP BY y funciones agregadas en SQL. (2022, September 14). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/group-by/>

Vistas en SQL. (2022, September 20). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/elementos-avanzados/view/>

SUBSELECT en SQL. (2022, September 15). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/ejecucion-consultas/subselect/>

Gestión de errores en Transact SQL. (2022, October 3). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/elementos-lenguaje/gestion-errores/>

Transacciones en Transact SQL. (2022, October 3). SQLearning | Tutoriales de SQL y Transact-SQL; SQLearning. <https://sqllearning.com/es/elementos-lenguaje/transacciones/>