TESTING-US

VALIDÁ TU APRENDIZAJE

CUADERNILLO DE ESTUDIO · NIVEL INTERMEDIO

SQL

consultas y uniones

Consultar, agrupar, unir tablas y modificar datos en bases relacionales, con ejemplos para probar en el navegador.

1 SELECT y WHERE

2 GROUP BY y HAVING

3 JOIN

4 Subconsultas, CASE y UNION

5 Modificar datos

\+ Ejercicios con respuestas

Testing-US · Validá tu aprendizaje
testing-us.com

Edición octubre 2026

Antes de empezar

SQL es el lenguaje para consultar y modificar bases de datos relacionales: lo usan los sistemas de gestión, las aplicaciones web y las herramientas de análisis. Sirve igual en PostgreSQL, MySQL, SQL Server, Oracle o SQLite, con pequeñas diferencias que se marcan cuando importan.

CÓMO APROVECHARLO

- Creá las tablas de ejemplo en un sitio como sqliteonline.com y ejecutá cada consulta.
- Antes de ejecutar, tratá de predecir el resultado: es lo que pide la evaluación.
- Cuando termines, rendí la evaluación **SQL: consultas y uniones** en testing-us.com.

TIP Base de ejemplo: clientes(id, nombre, ciudad, telefono), pedidos(id, cliente_id, fecha, total) y productos(id, nombre, categoria, precio).

Contenido

1. **Consultar: SELECT, WHERE y ORDER BY**
   Columnas, alias, filtros, IN, BETWEEN, LIKE, NULL, orden y LIMIT.
2. **Agrupar**
   COUNT, SUM, AVG, MIN, MAX, GROUP BY, HAVING y el orden lógico.
3. **Unir tablas: JOIN**
   Claves, INNER, LEFT, RIGHT, FULL y CROSS JOIN, self join.
4. **Subconsultas, CTE, CASE y UNION**
   IN, EXISTS, WITH, CASE, UNION, COALESCE y funciones útiles.
5. **Modificar datos**
   INSERT, UPDATE, DELETE, restricciones, transacciones e índices.

TEMA 1

1 Consultar: SELECT, WHERE y ORDER BY

LO QUE VAS A APRENDER

- Escribir consultas con columnas, alias y filtros.
- Combinar condiciones con AND, OR, IN, BETWEEN y LIKE.
- Trabajar bien con los valores NULL.
- Ordenar, limitar y quitar repetidos.

Estructura básica

- `SELECT columnas FROM tabla WHERE condición ORDER BY columna;`
- `SELECT *` trae todas las columnas; conviene nombrar solo las necesarias.
- Un **alias** renombra una columna o tabla en el resultado: `SELECT total * 1.21 AS total_con_iva FROM pedidos;`.
- Las palabras clave (`SELECT`, `FROM`) no distinguen mayúsculas; los textos entre comillas simples sí pueden distinguirlas según la base.
- Los textos y las fechas van entre **comillas simples**: `WHERE ciudad = 'Neuquén'`.

Filtrar con WHERE

- Comparaciones: `=`, `<>` (distinto; muchas bases aceptan también `!=`), `>`, `<`, `>=`, `<=`.
- Se combinan con `AND`, `OR` y `NOT`. `AND` se evalúa **antes** que `OR`: usá paréntesis para no equivocarte, por ejemplo `WHERE (ciudad = 'Neuquén' OR ciudad = 'Cipolletti') AND total > 1000`.
- `IN ('Neuquén', 'Cipolletti')` equivale a varios OR.
- `BETWEEN 100 AND 200` incluye **los dos extremos**.
- `LIKE` busca patrones: `%` reemplaza cualquier cantidad de caracteres y `_` uno solo. `nombre LIKE 'Ana%'` encuentra los que empiezan con Ana.

OJO AND se evalúa antes que OR. Ante la duda, poné paréntesis: además de evitar errores, la consulta se entiende mejor.

El valor NULL

- `NULL` significa "sin dato" (desconocido o vacío). No es cero ni texto vacío.
- Cualquier comparación con NULL da "desconocido", nunca verdadero: `WHERE telefono = NULL` **no devuelve filas**. Hay que usar `IS NULL` o `IS NOT NULL`.
- Una cuenta con NULL da NULL: `10 + NULL` es NULL.
- `COALESCE(telefono, 'sin teléfono')` devuelve el primer valor que no sea NULL.

Comparar y operar con NULL

`telefono = NULL`	→ desconocido
`telefono <> NULL`	→ desconocido
`telefono IS NULL`	→ verdadero o falso
`10 + NULL`	→ NULL
`COALESCE(telefono, '-')`	→ '-' si es NULL

Figura 1.1 Con NULL no se compara con =: se usa IS NULL.

Ordenar, limitar y quitar repetidos

- `ORDER BY total DESC` ordena de mayor a menor; `ASC` (de menor a mayor) es el orden por defecto. Se puede ordenar por varias columnas: `ORDER BY ciudad, nombre`.
- Sin `ORDER BY`, la base **no garantiza** ningún orden.
- `LIMIT 5` (PostgreSQL, MySQL, SQLite) devuelve solo las primeras 5 filas; en SQL Server se escribe `SELECT TOP 5 ...` y en el estándar `FETCH FIRST 5 ROWS ONLY`.
- `SELECT DISTINCT ciudad FROM clientes;` devuelve cada ciudad una sola vez.

Para practicar

1. Escribí una consulta con el nombre y el total de los pedidos de más de $50.000, de mayor a menor.
2. ¿Qué nombres encuentra LIKE '%ez'?
3. ¿Por qué WHERE email = NULL no devuelve filas?
4. ¿Cómo pedís las 3 ciudades distintas que aparecen primero en orden alfabético?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 2

2 Agrupar: funciones de agregación, GROUP BY y HAVING

LO QUE VAS A APRENDER

- Usar COUNT, SUM, AVG, MIN y MAX.
- Saber cómo tratan los NULL.
- Agrupar con GROUP BY y filtrar grupos con HAVING.
- Conocer el orden en que se resuelve una consulta.

Funciones de agregación

- `COUNT`, `SUM`, `AVG`, `MIN` y `MAX` resumen muchas filas en un solo valor.
- `COUNT(*)` cuenta **filas**; `COUNT(columna)` cuenta los valores que **no son NULL** en esa columna; `COUNT(DISTINCT columna)` cuenta los valores distintos.
- `SUM`, `AVG`, `MIN` y `MAX` **ignoran los NULL**. Por eso `AVG(nota)` promedia solo las notas cargadas.
- Sin `GROUP BY`, una consulta con agregaciones devuelve **una sola fila**.

GROUP BY

- Agrupa las filas que tienen el mismo valor y calcula la agregación por grupo: `SELECT ciudad, COUNT(*) FROM clientes GROUP BY ciudad;`.
- Regla (estándar, PostgreSQL y SQL Server): toda columna del `SELECT` que no esté dentro de una función de agregación tiene que estar en el `GROUP BY`. SQLite y algunas configuraciones de MySQL no dan error, pero muestran un valor cualquiera del grupo: no conviene aprovecharlo.
- Se puede agrupar por varias columnas: `GROUP BY ciudad, anio`.
- Los NULL forman **un grupo** propio.

```
SELECT ciudad, COUNT(*), SUM(total) ... GROUP BY ciudad
```

pedidos

id	ciudad	total
1	Neuquén	300
2	Cipolletti	120
3	Neuquén	80
4	Neuquén	200
5	Cipolletti	50

→

resultado: una fila por grupo

ciudad	count	sum
Neuquén	3	580
Cipolletti	2	170

Figura 2.1 GROUP BY arma un grupo por ciudad y calcula por grupo.

HAVING

- `WHERE` filtra **filas antes** de agrupar; `HAVING` filtra **grupos después** de agrupar.
- Ejemplo: clientes con más de 3 pedidos: `SELECT cliente_id, COUNT(*) FROM pedidos GROUP BY cliente_id HAVING COUNT(*) > 3;`.
- Una condición sobre una agregación (`SUM(total) > 100000`) va en `HAVING`; no se puede poner en `WHERE`.

Orden lógico de una consulta

Aunque se escribe empezando por SELECT, la base la resuelve en este orden:

1. `FROM` (y los `JOIN`)
2. `WHERE`
3. `GROUP BY`
4. `HAVING`
5. `SELECT`
6. `DISTINCT`
7. `ORDER BY`
8. `LIMIT`

Por eso, en muchas bases (PostgreSQL, SQL Server, MySQL), un alias definido en el SELECT **no** se puede usar en el WHERE, pero sí en el ORDER BY (SQLite sí lo permite).



Figura 2.2 Se escribe empezando por SELECT, pero se resuelve en este orden.

Para practicar

1. Una columna tiene 8 valores y 2 NULL. ¿Qué dan COUNT(*) y COUNT(columna)?
2. Escribí una consulta con el total vendido por cliente, solo de los que superan $200.000.
3. ¿Dónde va la condición fecha >= '2026-01-01': en WHERE o en HAVING?
4. ¿Por qué no se puede usar en el WHERE un alias del SELECT?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 3

3 Unir tablas: JOIN

LO QUE VAS A APRENDER

- Relacionar tablas con claves primarias y foráneas.
- Usar INNER JOIN y LEFT JOIN.
- Encontrar filas sin coincidencia.
- Evitar los errores más comunes al unir.

Claves y relaciones

- La **clave primaria** (*primary key*) identifica cada fila: `clientes.id`.
- Una **clave foránea** (*foreign key*) apunta a la clave de otra tabla: `pedidos.cliente_id` apunta a `clientes.id`. Así se relaciona cada pedido con su cliente (relación de uno a muchos).

INNER JOIN

- `SELECT c.nombre, p.total FROM clientes c INNER JOIN pedidos p ON p.cliente_id = c.id;`
- Devuelve solo las combinaciones que **coinciden** en las dos tablas: un cliente sin pedidos no aparece; un cliente con 3 pedidos aparece 3 veces.
- `JOIN` a secas es lo mismo que `INNER JOIN`.
- Los **alias** de tabla (`c`, `p`) acortan la consulta y evitan ambigüedades cuando dos tablas tienen una columna con el mismo nombre (como `id`).

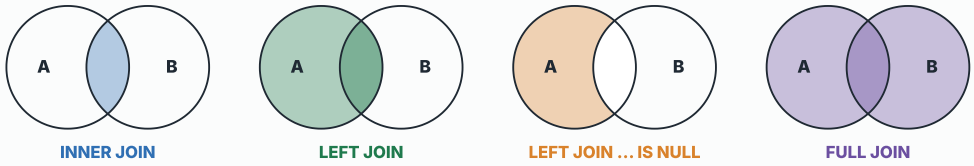


Figura 3.1 Qué parte de las tablas devuelve cada tipo de JOIN.

LEFT, RIGHT y FULL JOIN

- `LEFT JOIN` devuelve **todas** las filas de la tabla de la izquierda, aunque no tengan coincidencia; las columnas de la otra tabla quedan en **NULL**.
- Para encontrar los que **no** tienen coincidencia: `SELECT c.nombre FROM clientes c LEFT JOIN pedidos p ON p.cliente_id = c.id WHERE p.id IS NULL;` (clientes sin pedidos).
- `RIGHT JOIN` es lo mismo con la tabla de la derecha (casi siempre se reescribe como LEFT JOIN). `FULL JOIN` devuelve las filas de las dos tablas, coincidan o no (MySQL no lo tiene).
- Ojo: una condición sobre la tabla de la derecha puesta en el `WHERE` (por ejemplo `WHERE p.total > 1000`) descarta las filas con NULL y convierte el LEFT JOIN en un INNER JOIN. Si se quiere conservar a todos, la condición va en el `ON`.

clientes

id	nombre
1	Ana
2	Bruno
3	Carla

pedidos

id	cliente_id	total
10	1	500
11	1	300
12	3	900

clientes c LEFT JOIN pedidos p

nombre	p.id	total
Ana	10	500
Ana	11	300
Bruno	NULL	NULL
Carla	12	900

Bruno no tiene pedidos: aparece igual con NULL. Con INNER JOIN no aparec

Figura 3.2 Un LEFT JOIN conserva a los clientes sin pedidos.

Otros casos

- `CROSS JOIN` combina **cada** fila de una tabla con **cada** fila de la otra: 4 filas × 3 filas = 12 filas. Pasa lo mismo, sin querer, si se olvida la condición de unión.
- Se pueden unir varias tablas seguidas: pedidos con clientes y con productos.
- Un **self join** une una tabla consigo misma, por ejemplo empleados con su jefe: `FROM empleados e LEFT JOIN empleados j ON e.jefe_id = j.id`.
- Si un JOIN devuelve más filas de las esperadas, revisá que la condición `ON` use las columnas correctas y que la relación no sea de muchos a muchos.

Para practicar

1. Hay 4 productos; 2 se vendieron 3 veces cada uno y 2 nunca. ¿Cuántas filas da productos INNER JOIN ventas? ¿Y LEFT JOIN?
2. Escribí la consulta de los productos que nunca se vendieron.
3. ¿Qué pasa si te olvidás el ON en un JOIN?
4. ¿Por qué hace falta escribir c.id o p.id?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 4

4 Subconsultas, CTE, CASE y UNION

LO QUE VAS A APRENDER

- Usar subconsultas en el WHERE y en el FROM.
- Ordenar consultas largas con WITH.
- Clasificar con CASE.
- Juntar resultados con UNION y UNION ALL.

Subconsultas

- Una **subconsulta** es un SELECT dentro de otra consulta.
- En el WHERE: `SELECT * FROM pedidos WHERE total > (SELECT AVG(total) FROM pedidos);` (pedidos por encima del promedio). Esa subconsulta tiene que devolver **un solo valor**: si devuelve varias filas, PostgreSQL, MySQL y SQL Server dan error (SQLite, en cambio, usa la primera fila).
- Con `IN`: `SELECT nombre FROM clientes WHERE id IN (SELECT cliente_id FROM pedidos);` (clientes con al menos un pedido).
- `EXISTS (subconsulta)` es verdadero si la subconsulta devuelve **al menos una fila**.
- Cuidado con `NOT IN`: si la subconsulta devuelve algún NULL, `NOT IN` no devuelve ninguna fila. `NOT EXISTS` no tiene ese problema.
- Una subconsulta también puede ir en el FROM, como si fuera una tabla, con un alias.

OJO Si la lista de una subconsulta con NOT IN puede tener NULL, usá NOT EXISTS: con un solo NULL, NOT IN no devuelve nada.

CTE (WITH)

- Una **CTE** (*common table expression*) le pone nombre a una subconsulta para leerla mejor:

```
WITH ventas_por_cliente AS (SELECT cliente_id, SUM(total) AS gastado FROM pedidos GROUP BY cliente_id) SELECT * FROM ventas_por_cliente WHERE gastado > 5000
```

- Existe solo durante esa consulta.

CASE

- `CASE` devuelve valores según condiciones, como un SI de Excel:

```
CASE WHEN total >= 100000 THEN 'Grande' WHEN total >= 20000 THEN 'Mediano' ELSE 'Chico' END AS tamaño
```

- Devuelve el resultado de la **primera** condición verdadera; si ninguna se cumple y no hay ELSE, devuelve NULL.
- Combinado con SUM o COUNT sirve para contar por categorías en una sola consulta: `SUM(CASE WHEN estado = 'pagado' THEN 1 ELSE 0 END)`.



Figura 4.1 CASE devuelve el resultado de la primera condición verdadera.

UNION y funciones útiles

- `UNION` junta los resultados de dos SELECT con la **misma cantidad de columnas** y **quita los repetidos**; `UNION ALL` los deja (y es más rápido).
- `COALESCE(a, b)` devuelve el primer valor no NULL. `NULLIF(a, b)` devuelve NULL si a y b son iguales (útil para evitar dividir por cero).
- Funciones de texto comunes: `UPPER`, `LOWER`, `LENGTH` (en SQL Server, `LEN`), `TRIM`, `SUBSTRING`; para unir, `||` (estándar, PostgreSQL, SQLite, Oracle) o `CONCAT` (en MySQL solo `CONCAT`; en SQL Server, `+` o `CONCAT`).
- `ROUND(valor, 2)` redondea a 2 decimales. En PostgreSQL, SQLite y SQL Server, dividir dos enteros da un entero: `7 / 2` da `3` (en MySQL da `3.5000`).

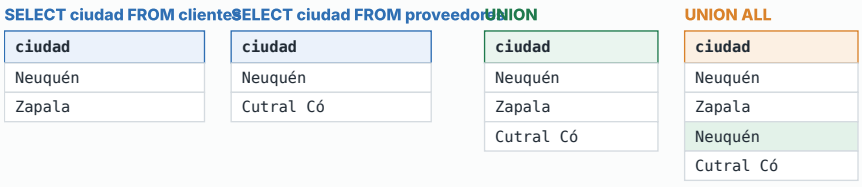


Figura 4.2 UNION quita repetidos; UNION ALL los deja.

Funciones de ventana (introducción)

- `ROW_NUMBER() OVER (PARTITION BY cliente_id ORDER BY fecha DESC)` numera las filas dentro de cada grupo **sin agruparlas**: sirve, por ejemplo, para quedarse con el último pedido de cada cliente.

Para practicar

1. Escribí los pedidos cuyo total supera el promedio.
2. ¿Qué devuelve CASE WHEN 5 > 10 THEN 'sí' END?
3. ¿Qué devuelve SELECT 9 / 2 en PostgreSQL? ¿Y SELECT 9 / 2.0?
4. ¿Cuándo usarías UNION ALL en lugar de UNION?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

TEMA 5

5 Modificar datos, restricciones y transacciones

LO QUE VAS A APRENDER

- Insertar, actualizar y borrar filas sin accidentes.
- Definir claves y restricciones.
- Usar transacciones con COMMIT y ROLLBACK.
- Conocer índices, normalización e inyección SQL.

INSERT, UPDATE y DELETE

- `INSERT INTO clientes (nombre, ciudad) VALUES ('Ana Paz', 'Neuquén');` agrega una fila. Las columnas que no se nombran toman su valor por defecto (o NULL).
- `UPDATE productos SET precio = precio * 1.10 WHERE categoria = 'Bebidas';` cambia filas existentes.
- `DELETE FROM pedidos WHERE fecha < '2020-01-01';` borra filas.
- **Sin WHERE, UPDATE y DELETE afectan a todas las filas de la tabla.** Buena práctica: antes de ejecutarlos, probar el mismo WHERE con un SELECT.
- `TRUNCATE TABLE` vacía una tabla entera, y `DROP TABLE` borra la tabla completa (estructura y datos).



Figura 5.1 Un UPDATE o un DELETE sin WHERE afecta a toda la tabla.

Crear tablas y restricciones

- `CREATE TABLE clientes (id INTEGER PRIMARY KEY, nombre VARCHAR(100) NOT NULL, email VARCHAR(150) UNIQUE, ciudad VARCHAR(60));`
- `PRIMARY KEY`: identifica la fila; no se repite y no puede ser NULL (SQLite tiene una excepción histórica: conviene sumar `NOT NULL`).
- `NOT NULL`: el dato es obligatorio. `UNIQUE`: no se puede repetir. `CHECK (precio >= 0)`: valida una condición. `DEFAULT`: valor por defecto.
- `FOREIGN KEY (cliente_id) REFERENCES clientes(id)`: impide cargar un pedido de un cliente que no existe y, según cómo se defina, impide borrar un cliente que tiene pedidos (o los borra en cascada con `ON DELETE CASCADE`). En SQLite hay que activarlas con `PRAGMA foreign_keys = ON;`.
- Tipos de datos frecuentes: `INTEGER`, `DECIMAL(10,2)` o `NUMERIC` (montos exactos, mejor que `FLOAT` para dinero), `VARCHAR(n)`, `DATE`, `TIMESTAMP`, `BOOLEAN`.
- `ALTER TABLE clientes ADD COLUMN telefono VARCHAR(30);` modifica la estructura.

Transacciones

- Una **transacción** agrupa varias operaciones que se tienen que hacer **todas o ninguna**: `BEGIN;` (o `START TRANSACTION;` en el estándar y MySQL; `BEGIN TRANSACTION;` en SQL Server) … `COMMIT;` confirma los cambios y `ROLLBACK;` los deshace.
- Ejemplo: una transferencia descuenta de una cuenta y suma a otra; si falla la segunda operación, la primera también se deshace.
- Las propiedades **ACID**: atomicidad (todo o nada), consistencia (respeta las reglas), aislamiento (las transacciones no se pisan) y durabilidad (lo confirmado no se pierde).



Figura 5.2 Una transferencia dentro de una transacción.

Índices y buenas prácticas

- Un **índice** acelera las búsquedas y los JOIN por una columna (como el índice de un libro), a cambio de ocupar espacio y hacer un poco más lentas las escrituras. Las claves primarias tienen índice automáticamente.
- **Normalizar** es evitar datos repetidos: en vez de escribir el nombre del cliente en cada pedido, se guarda su id y se une con la tabla de clientes.
- Para no sufrir **inyección SQL**, los valores que escribe un usuario se pasan como **parámetros** de la consulta, nunca pegados al texto del SQL.

Para practicar

1. Escribí la sentencia que sube 5 % los precios de la categoría Limpieza.
2. ¿Qué restricción usarías para que el stock no sea negativo?
3. ¿Qué diferencia hay entre DELETE FROM t y DROP TABLE t?
4. ¿Por qué no hay que armar el SQL pegando lo que escribe el usuario?

LAS RESPUESTAS ESTÁN AL FINAL DEL CUADERNILLO.

PARA CONTROLAR

Respuestas de los ejercicios

Tema 1

1. SELECT nombre, total FROM pedidos WHERE total > 50000 ORDER BY total DESC;
2. Los que terminan en ez, como Pérez o Gómez.
3. Porque cualquier comparación con NULL da desconocido; hay que usar IS NULL.
4. SELECT DISTINCT ciudad FROM clientes ORDER BY ciudad LIMIT 3;

Tema 2

1. 10 y 8.
2. SELECT cliente_id, SUM(total) FROM pedidos GROUP BY cliente_id HAVING SUM(total) > 200000;
3. En el WHERE: filtra filas, no grupos.
4. Porque el WHERE se resuelve antes que el SELECT, cuando el alias todavía no existe.

Tema 3

1. INNER: 6 filas. LEFT (productos a la izquierda): 8 filas.
2. SELECT p.nombre FROM productos p LEFT JOIN ventas v ON v.producto_id = p.id WHERE v.id IS NULL;
3. Se combinan todas las filas de una con todas las de la otra (producto cartesiano).
4. Porque las dos tablas tienen una columna id y la base no sabe cuál usar.

Tema 4

1. SELECT * FROM pedidos WHERE total > (SELECT AVG(total) FROM pedidos);
2. NULL: no se cumple y no hay ELSE.
3. 4 (división entera) y 4.5.
4. Cuando no puede haber repetidos o querés conservarlos: es más rápido.

Tema 5

1. UPDATE productos SET precio = precio * 1.05 WHERE categoria = 'Limpieza';
2. CHECK (stock >= 0).
3. DELETE borra las filas y deja la tabla; DROP TABLE borra la tabla completa.
4. Porque permite la inyección SQL: hay que pasar los valores como parámetros.

Validá lo que aprendiste

Rendí la evaluación **SQL: consultas y uniones** en **testing-us.com**. Es gratis: en cada intento te tocan 10 preguntas al azar, dos de cada tema de este cuadernillo. Tenés 20 minutos y aprobás con 70%.

Si aprobás, recibís un certificado con código de verificación que podés sumar a tu currículum y compartir con quien te contrate.

MIS NOTAS

© 2026 Testing-US. Material de estudio propio. Se permite imprimirlo y compartirlo sin modificaciones y sin fines comerciales.

Referencias: norma ISO/IEC 9075 (SQL) y la documentación de PostgreSQL, MySQL, SQLite y SQL Server. Los textos y dibujos de este cuadernillo son propios. Los nombres de productos mencionados son marcas de sus respectivos dueños y se citan solo para identificarlos.

El certificado de Testing-US acredita la aprobación de una evaluación de conocimientos. No es un título oficial ni habilita para el ejercicio profesional.