

PROMETEO

Unidad 5: Lenguaje SQL – DDL y DML

Bases de datos

Técnico superior Administración y sistemas informáticos

Sesión 16: Introducción a SQL, tipos de instrucciones

SQL, cuyas siglas significan *Structured Query Language*, es el lenguaje universal que permite comunicarse con las bases de datos relacionales. A diferencia de los lenguajes de programación tradicionales como Python o Java, SQL no te pide que describas *cómo* debe realizarse una tarea paso a paso, sino que simplemente declares *qué* resultado quieres obtener. Por eso decimos que es un lenguaje **declarativo**: tú defines el objetivo, y el sistema gestor de bases de datos (SGBD) se encarga de encontrar la forma más eficiente de alcanzarlo.

Su propósito es claro: **consultar, crear y modificar datos estructurados** que se almacenan en tablas con filas y columnas. Estas tablas se relacionan entre sí mediante claves, lo que permite que sistemas complejos —como las bases de datos de un banco, un ecommerce o una red social— se mantengan organizados y coherentes.

El estándar SQL se apoya en varios sublenguajes, entre los cuales dos son fundamentales:

DDL (Data Definition Language) o

Lenguaje de Definición de Datos. Es el conjunto de instrucciones que definen la estructura de la base de datos: crear nuevas tablas (*CREATE*), modificar su estructura (*ALTER*) o eliminarlas (*DROP*). Si lo comparas con el trabajo de un arquitecto, el DDL sería el plano del edificio. No contiene datos, sino las instrucciones que dicen qué espacios existirán y qué características tendrá cada uno.

DML (Data Manipulation Language) o

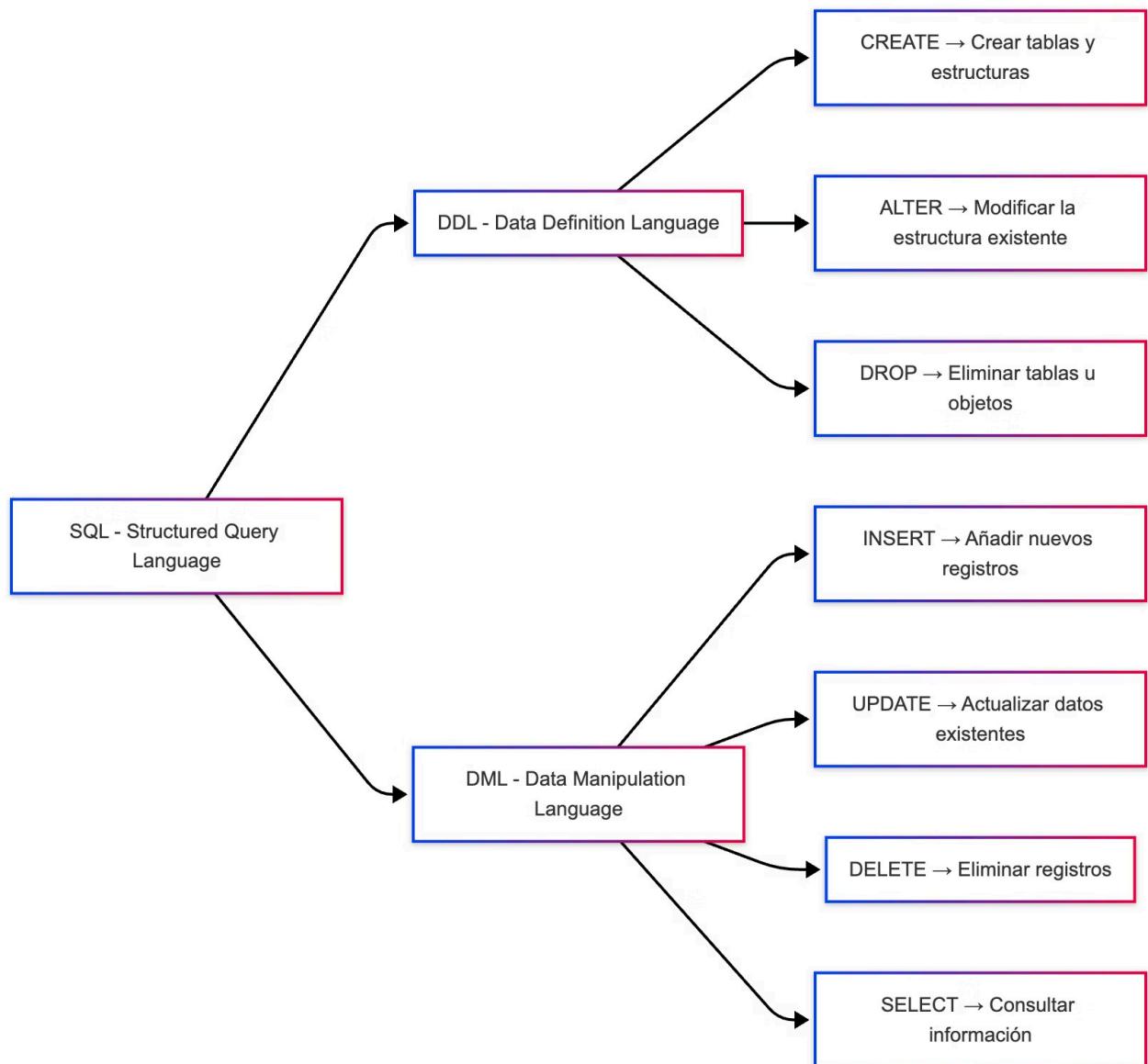
Lenguaje de Manipulación de Datos. Este conjunto de comandos sirve para interactuar con los datos almacenados dentro de las tablas creadas por el DDL. Incluye las instrucciones *INSERT* (insertar registros), *UPDATE* (modificarlos), *DELETE* (eliminarlos) y *SELECT* (consultarlos). En la analogía del arquitecto, el DML es la parte del constructor o del decorador: el que coloca los muebles, cambia su posición o busca un objeto concreto en una habitación.

Ambos lenguajes son complementarios. No puedes manipular datos si antes no existe una estructura donde almacenarlos, y una estructura sin datos no aporta valor. Entender esta diferencia y saber en qué momento usar cada uno es el primer paso para dominar SQL.

En el entorno profesional actual, SQL sigue siendo un **estándar transversal** que une perfiles técnicos y analíticos. Los desarrolladores backend lo utilizan para integrar las bases de datos en las aplicaciones; los analistas de datos lo aplican para extraer información de negocio; los científicos de datos lo combinan con lenguajes como Python para preparar datasets; e incluso los profesionales de marketing lo emplean para segmentar audiencias o medir campañas directamente desde el servidor.

Lejos de quedar obsoleto, SQL es hoy una **competencia esencial en la era del dato**, donde casi todas las decisiones empresariales se basan en información almacenada en bases de datos relacionales.

Esquema visual



- ⓘ **Descripción del esquema:** El nodo principal, **SQL**, representa el lenguaje global que engloba todas las operaciones sobre bases de datos relacionales. De él se derivan dos ramas:
- **DDL**, encargada de definir la estructura. Sus tres comandos principales –CREATE, ALTER y DROP– establecen, modifican o eliminan objetos como tablas, vistas o índices.
 - **DML**, dedicada a la manipulación de los datos almacenados. Contiene las acciones de insertar (*INSERT*), modificar (*UPDATE*), borrar (*DELETE*) y consultar (*SELECT*), esta última siendo la más utilizada en el día a día.

Visualmente, el esquema permite entender que SQL opera en **dos niveles complementarios**: la construcción de la base (DDL) y el trabajo con los datos (DML).

Caso de estudio

Oracle y el nacimiento del estándar SQL

Contexto

En los años 70, IBM desarrollaba internamente un sistema llamado **System R**, basado en el modelo relacional propuesto por Edgar F. Codd. Con él surgió un prototipo de lenguaje de consulta llamado **SEQUEL (Structured English Query Language)**. Sin embargo, el proyecto permaneció experimental y no llegó a comercializarse a tiempo.

En paralelo, **Larry Ellison**, cofundador de una pequeña empresa llamada *Software Development Laboratories* (más tarde *Oracle Corporation*), vio la oportunidad de convertir ese modelo teórico en un producto comercial. Apostó por desarrollar su propio SGBD implementando las ideas de Codd y su lenguaje de consultas: SQL.

Estrategia

Oracle lanzó en 1979 la primera base de datos comercial basada en SQL, llamada Oracle Database, antes incluso de que IBM sacara su propio sistema. Su estrategia fue clara: adoptar un lenguaje declarativo que simplificara el acceso a los datos y lo hiciera comprensible para distintos perfiles, no solo para programadores expertos. Mientras IBM se centraba en la robustez técnica, Oracle apostó por la **estandarización y la portabilidad**, haciendo que su lenguaje funcionara en diferentes sistemas operativos.

El éxito de Oracle impulsó la **estandarización de SQL** por parte de los organismos ANSI e ISO durante los años 80, asegurando su compatibilidad entre distintos SGBD (MySQL, PostgreSQL, SQL Server, DB2...).

Resultado

Oracle se consolidó como líder mundial del mercado de bases de datos empresariales, con clientes en banca, telecomunicaciones y administración pública. Hoy, gracias a aquella apuesta temprana, SQL es **el idioma común de los datos estructurados**, adoptado por todas las grandes plataformas. Su existencia permitió que los profesionales pudieran aprender un único lenguaje y aplicarlo en distintos sistemas, reduciendo barreras tecnológicas y unificando el ecosistema global de gestión de datos.

Herramientas y consejos

- **Empieza con entornos online gratuitos**

Si estás aprendiendo, no necesitas instalar nada complejo. Plataformas como **DB Fiddle** (db-fiddle.com) o **SQL Fiddle** (sqlfiddle.com) te permiten escribir sentencias DDL y DML directamente en el navegador. Puedes crear una tabla con CREATE TABLE, insertar registros con INSERT INTO y probar consultas con SELECT sin configuración previa.

- **Utiliza herramientas profesionales para proyectos reales**

Cuando pases a un entorno profesional, familiarízate con herramientas como:

- **DBeaver**: gestor universal que conecta con casi cualquier SGBD.
- **MySQL Workbench**: entorno visual oficial para bases MySQL.
- **pgAdmin**: interfaz de administración para PostgreSQL.

Estas aplicaciones permiten ejecutar scripts, visualizar esquemas, generar diagramas entidad-relación y exportar datos fácilmente.

- **Adopta buenas prácticas desde el principio**

- Termina siempre tus sentencias con punto y coma (;), incluso si el sistema no lo exige.
- Escribe las palabras clave de SQL en mayúsculas (SELECT, FROM, WHERE) para mejorar la legibilidad.
- Usa nombres de tablas y columnas descriptivos (clientes, fecha_pedido), evitando abreviaturas crípticas.
- Guarda tus consultas frecuentes en archivos .sql para reutilizarlas.

- **Conecta teoría y práctica con datasets reales**

Puedes descargar conjuntos de datos públicos (por ejemplo, de *Kaggle* o data.gov.es) y practicar consultas sobre ellos. Ejecutar un SELECT sobre miles de registros reales te permitirá comprender la potencia del lenguaje y cómo optimizarlo.

- **Aprende a leer los mensajes del SGBD**

Cada sistema (MySQL, PostgreSQL, SQL Server, Oracle) devuelve mensajes específicos ante errores de sintaxis o violaciones de restricciones. Leerlos con atención y entender su causa es parte esencial del aprendizaje. SQL no solo te dice que algo falló, sino exactamente qué comando o qué tabla causó el problema.

Mitos y realidades

✗ MITO: "SQL ya no sirve porque ahora todo es NoSQL."

→ **FALSO.** Aunque las bases de datos NoSQL (como MongoDB o Cassandra) han ganado terreno en entornos de big data, **SQL sigue siendo el estándar más usado** en empresas, gobiernos y startups. De hecho, muchas soluciones modernas incorporan interfaces compatibles con SQL por su facilidad de uso y su enorme comunidad. Incluso Google BigQuery, Snowflake y Apache Spark ofrecen versiones de *SQL extendido* para trabajar con grandes volúmenes de información.

✓ Realidad: SQL no ha sido reemplazado, sino **evolucionado**. Se usa tanto en bases relationales clásicas como en sistemas distribuidos y analíticos.

✗ MITO: "SQL es solo para administradores de bases de datos."

→ **FALSO.** Hoy, casi todos los perfiles digitales trabajan con datos. Un analista de marketing necesita SQL para segmentar clientes; un científico de datos lo usa para preparar datasets; un periodista de datos lo emplea para investigar bases públicas; incluso en recursos humanos se consulta información mediante SQL para elaborar informes internos.

✓ Realidad: SQL es una **habilidad transversal** en el mercado laboral. No se limita a la administración técnica, sino que forma parte del trabajo diario de muchos roles profesionales.

❑ Resumen final

- **SQL:** lenguaje declarativo para interactuar con bases de datos relationales.
- **DDL:** define la estructura (CREATE, ALTER, DROP).
- **DML:** manipula los datos (INSERT, UPDATE, DELETE, SELECT).
- **SELECT:** comando más utilizado, permite consultar información.
- **SQL** sigue siendo esencial en todos los sectores profesionales.
- **Buena práctica:** escribir palabras clave en mayúsculas y terminar con ;.



Sesión 17: DDL: creación de bases de datos y tablas, tipos de datos

El **DDL (Data Definition Language)** o **Lenguaje de Definición de Datos** es la piedra angular de toda base de datos. Si SQL es el idioma que usamos para hablar con los sistemas gestores de bases de datos, el DDL es la parte del idioma que se ocupa de **definir su estructura**: dónde se almacenará la información, cómo se organizará y qué reglas se aplicarán a cada campo.

Cuando creas una base de datos desde cero, el primer paso es establecer el entorno donde residirán las tablas. Esto se hace con el comando:

```
CREATE DATABASE nombre_bd;
```

Este comando genera un "contenedor lógico" dentro del sistema gestor (como MySQL, PostgreSQL o SQL Server) que agrupará todas las tablas, vistas y relaciones que formen parte del proyecto. Una vez creada, el siguiente paso es **entrar en ella** con:

```
USE nombre_bd;
```

A partir de ahí, puedes empezar a construir las tablas que darán forma al modelo de datos de tu organización. La instrucción más importante del DDL es, sin duda, **CREATE TABLE**. Con ella defines cada tabla, especificas sus columnas, los tipos de datos que contendrán y las restricciones que garantizarán la integridad de los registros.

Por ejemplo:

```
CREATE TABLE Clientes (
    id_cliente INT PRIMARY KEY,
    nombre VARCHAR(100) NOT NULL,
    fecha_registro DATE,
    presupuesto DECIMAL(10,2)
);
```

Esta sentencia define una tabla con cuatro columnas y varias características esenciales:

- **id_cliente**: un número entero (*INT*) que identifica de forma única a cada cliente y actúa como *PRIMARY KEY*.
- **nombre**: un texto de hasta 100 caracteres, obligatorio (*NOT NULL*).
- **fecha_registro**: almacena fechas en formato *DATE*.
- **presupuesto**: un número decimal con 10 dígitos totales y 2 decimales.

💡 El DDL se centra en las estructuras, no en los datos. Es decir, te permite establecer el "esqueleto" que sostendrá la información. Cada decisión en este punto tiene un impacto a largo plazo. Escoger el tipo de dato correcto no solo ahorra espacio, sino que evita errores lógicos y mejora el rendimiento. Si defines una columna con un tipo demasiado amplio (por ejemplo, *VARCHAR(255)* para un campo que solo almacena códigos de 5 caracteres), estarás desperdiciando recursos. Si lo defines demasiado restringido, podrías provocar errores al insertar valores válidos.

Los tipos de datos más usados en SQL se agrupan en cuatro grandes familias:



Numéricos

- *INT*: números enteros (por ejemplo, 42).
- *DECIMAL(p, s)*: números con decimales y precisión exacta (por ejemplo, 10 dígitos en total y 2 decimales).
- *FLOAT* o *DOUBLE*: números con decimales en coma flotante (para cálculos científicos o financieros).



Texto o cadenas

- *CHAR(n)*: texto de longitud fija. Ideal para códigos o campos siempre del mismo tamaño.
- *VARCHAR(n)*: texto de longitud variable, hasta un máximo definido.
- *TEXT*: textos largos como descripciones o comentarios.



Fechas y tiempo

- *DATE*: almacena fechas (YYYY-MM-DD).
- *TIME*: horas del día (HH:MM:SS).
- *DATETIME* o *TIMESTAMP*: fecha y hora simultáneamente.



Lógicos y otros

- *BOOLEAN*: valores de verdadero o falso.
- *BLOB*: datos binarios (imágenes, archivos, etc.).

Además de los tipos de datos, en la definición de cada tabla se añaden **restricciones (constraints)**. Estas son reglas que el sistema aplica automáticamente para proteger la integridad de los datos:

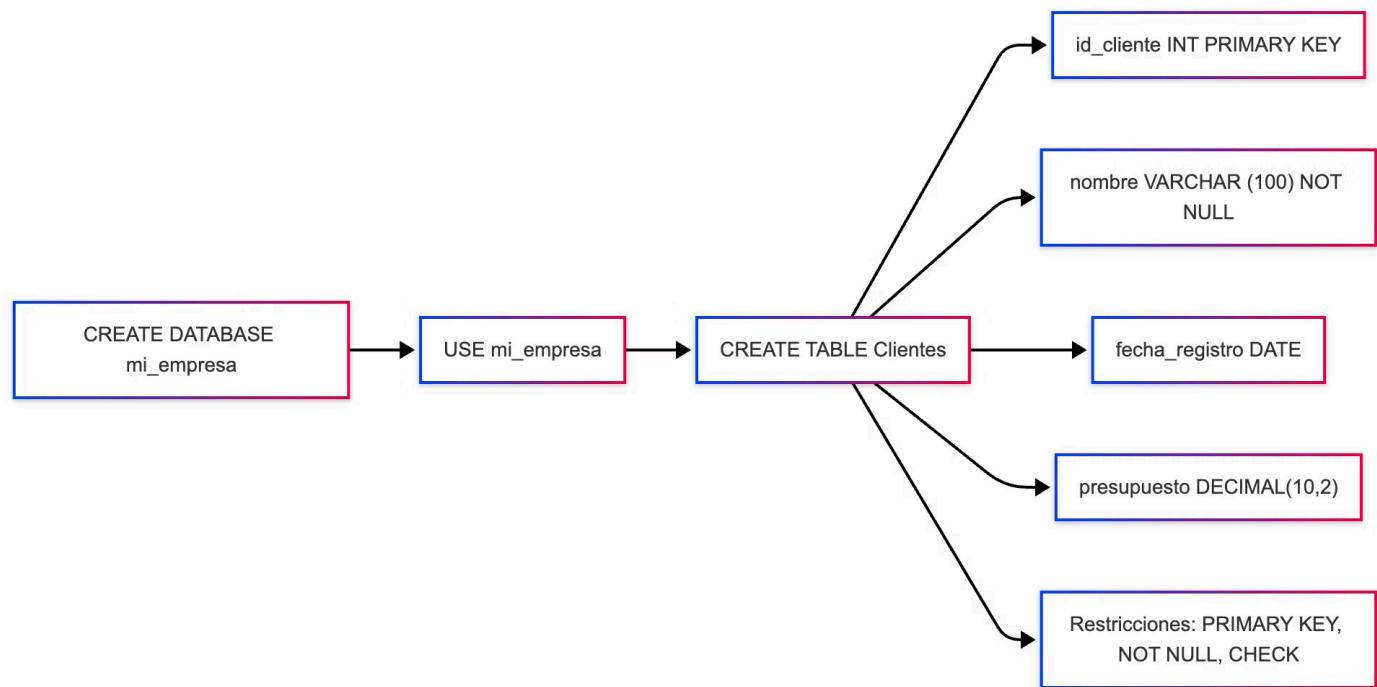
- **PRIMARY KEY**: identifica de forma única cada fila.
- **FOREIGN KEY**: relaciona una tabla con otra.
- **NOT NULL**: impide que una columna quede vacía.
- **UNIQUE**: obliga a que los valores no se repitan.
- **CHECK**: define condiciones lógicas que los datos deben cumplir (por ejemplo, que el salario sea mayor que cero).

El DDL es, en esencia, **el lenguaje de la arquitectura de los datos**. Antes de manipular información (con DML), debes tener un diseño sólido y coherente.

En la práctica profesional, diseñar bien una base de datos con DDL es tan importante como diseñar los planos de un edificio: si la estructura es débil, todo lo demás fallará.

```
31     def __init__(self, path=None):
32         self.file = None
33         self.fingerprints = set()
34         self.logdups = True
35         self.debug = debug
36         self.logger = logging.getLogger(__name__)
37         if path:
38             self.file = open(os.path.join(path, 'fingerprint.log'), 'a')
39             self.file.seek(0)
40             self.fingerprints.update(self._read_file())
41
42     @classmethod
43     def from_settings(cls, settings):
44         debug = settings.getbool('SUPERVISOR_DEBUG')
45         return cls(job_dir(settings), debug)
46
47     def request_seen(self, request):
48         fp = self.request_fingerprint(request)
49         if fp in self.fingerprints:
50             return True
51         self.fingerprints.add(fp)
52         if self.file:
53             self.file.write(fp + os.linesep)
54
55     def request_fingerprint(self, request):
56         return request_fingerprint(request)
```

Esquema visual



- ⓘ **Descripción del esquema:** El proceso parte del comando **CREATE DATABASE**, que crea el contenedor donde residirán las tablas. Después, con **USE**, se selecciona la base de datos activa. A continuación, **CREATE TABLE** define las estructuras de datos, especificando cada columna junto con su tipo de dato y sus restricciones. El diagrama refleja el orden jerárquico del DDL: primero el entorno, luego las estructuras y finalmente las reglas que garantizan la coherencia.

Caso de estudio

La arquitectura de datos de Salesforce

Contexto

Salesforce es una de las plataformas de CRM (Customer Relationship Management) más utilizadas del mundo, con millones de usuarios que acceden simultáneamente a sus datos. Dado que todos los clientes comparten la misma infraestructura en la nube, Salesforce utiliza un modelo **multitenant**, en el que una única base de datos física sirve a miles de empresas distintas.

Estrategia

En una base de datos convencional, cada empresa tendría sus propias tablas creadas mediante comandos DDL (CREATE TABLE, ALTER TABLE, etc.). Sin embargo, Salesforce no puede crear millones de tablas físicas diferentes: sería ineficiente y difícil de mantener. Para resolverlo, utiliza un **sistema de metadatos dinámico**. Cuando un cliente "crea un objeto personalizado" en su interfaz (por ejemplo, una tabla llamada "Pedidos"), el sistema no ejecuta realmente un CREATE TABLE. En su lugar, almacena la definición del objeto (sus campos, tipos de datos, relaciones y restricciones) en una tabla central de metadatos.

Los datos reales de todos los clientes se almacenan en unas pocas tablas físicas de gran tamaño. Cada registro incluye un identificador que indica a qué cliente pertenece, asegurando el aislamiento de la información. De este modo, Salesforce permite a cada usuario crear su propio "DDL lógico" sin necesidad de modificar la estructura física de la base de datos.

Resultado

Este enfoque de arquitectura ha permitido a Salesforce escalar hasta millones de usuarios sin perder coherencia ni rendimiento. Aunque los clientes creen que están ejecutando comandos DDL, lo que hacen en realidad es definir metadatos que luego se interpretan dinámicamente. La enseñanza clave es clara: **todo sistema de datos eficiente comienza con una definición estructurada y precisa**. El DDL no solo define tablas, sino que sienta las bases de la seguridad, la escalabilidad y la integridad de la información.

Herramientas y consejos

Comprueba las dependencias antes de modificar o eliminar

En bases de datos con relaciones, eliminar una tabla que actúa como referencia (clave foránea) puede generar errores o romper integridad referencial. Antes de usar `DROP TABLE`, revisa con herramientas como el **Database Diagram** o consultas al `INFORMATION_SCHEMA` qué objetos dependen de ella.

Usa herramientas visuales para aprender y diseñar

Plataformas como **MySQL Workbench** o **pgAdmin** permiten diseñar bases de datos gráficamente. Puedes arrastrar tablas, definir sus campos y restricciones, y la herramienta genera automáticamente el código DDL correspondiente. Son perfectas para visualizar relaciones entre tablas y comprobar dependencias de claves foráneas.

Define con precisión tus tipos de datos

Cada columna debe tener un tipo que se ajuste exactamente a su contenido esperado.

- Para números de serie o identificadores → INT.
- Para precios → DECIMAL(p,2).
- Para códigos fijos (por ejemplo, códigos postales) → CHAR(5) en lugar de VARCHAR(100).

Una definición precisa evita errores, ahorra espacio y mejora el rendimiento en consultas.

Controla los cambios estructurales

ALTER TABLE: permite añadir o modificar columnas sin borrar los datos existentes. Ejemplo:

```
ALTER TABLE Clientes ADD email  
VARCHAR(100);
```

DROP TABLE: elimina por completo una tabla y todos sus datos. Úsalo con extrema precaución, especialmente en entornos productivos.

```
DROP TABLE Clientes;
```

Utiliza un sistema de control de versiones para tus scripts SQL

En entornos profesionales, los equipos guardan los archivos .sql con los comandos DDL en repositorios de código (como **GitHub** o **GitLab**). Esto permite mantener un historial de cambios y revertir versiones en caso de errores.

Mitos y realidades

✗ MITO: "Puedo usar VARCHAR(255) o TEXT para todas mis columnas de texto."

→ **FALSO** Aunque técnicamente es posible, es una mala práctica. Los tipos de datos excesivamente amplios consumen memoria innecesaria, ralentizan las búsquedas y hacen que los índices sean menos eficientes.

✓ Realidad: Un diseño profesional ajusta el tamaño de los campos al uso real. Por ejemplo, si sabes que un nombre de usuario no superará los 50 caracteres, define VARCHAR(50); si un código de país tiene 2 letras, usa CHAR(2).

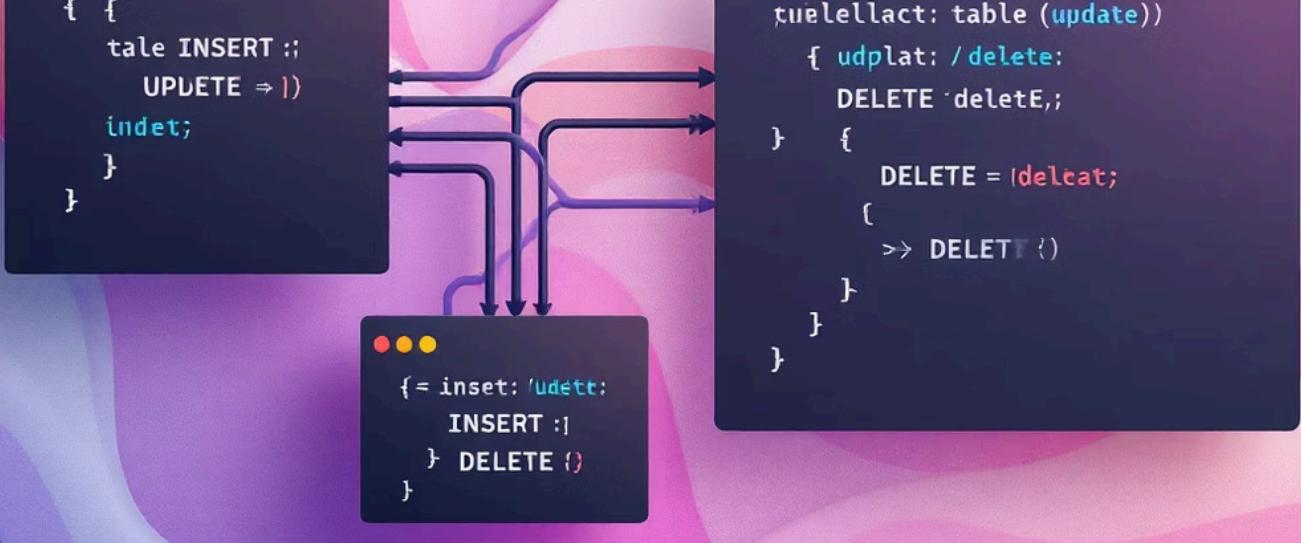
✗ MITO: "Una vez creada una tabla, su estructura no se puede cambiar."

→ **FALSO:** El DDL incluye el comando **ALTER TABLE**, precisamente para modificar estructuras existentes. Puedes añadir columnas, cambiar tipos de datos o ajustar restricciones según evolucione el sistema.

✓ Realidad: En entornos de producción, los cambios estructurales deben planificarse. Operaciones grandes con ALTER TABLE pueden bloquear temporalmente la tabla, afectando a los usuarios que la usan. Las empresas suelen programar estas modificaciones en horarios de mantenimiento para evitar interrupciones.

Resumen final

- **DDL:** define la estructura de las bases de datos (tablas, columnas, tipos, restricciones).
- **CREATE DATABASE:** crea un nuevo contenedor lógico.
- **CREATE TABLE:** define una tabla y sus columnas.
- **Tipos de datos comunes:** INT, DECIMAL, VARCHAR, CHAR, DATE, BOOLEAN.
- **Restricciones:** PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, CHECK.
- **ALTER TABLE:** modifica estructuras existentes.
- **DROP TABLE:** elimina una tabla permanentemente.
- Elección del tipo de dato correcto = rendimiento + integridad + eficiencia.



Sesión 18: DML: inserción, modificación y borrado de datos

Si con el DDL has levantado la estructura de tu "casa de datos", con el **DML (Data Manipulation Language)** vas a vivir dentro de ella: **insertar, modificar y borrar** información real en las tablas. El DML es el sublenguaje de SQL pensado para **gestionar el ciclo de vida del dato**. Te permite: crear registros nuevos cuando ocurre un evento (una compra, un registro de usuario), actualizarlos cuando cambian (un nuevo email, un estado de pedido) y eliminarlos cuando dejan de ser necesarios o válidos (borrado lógico o físico, según la política de la empresa).

Los tres comandos clave del DML que debes dominar desde hoy son:

INSERT INTO: añade filas nuevas a una tabla. Sintaxis básica:

```
INSERT INTO nombre_tabla (columna1, columna2, ...)  
VALUES (valor1, valor2, ...);
```

- Debes **respetar tipos de dato y restricciones** (NOT NULL, UNIQUE, claves foráneas). Si no indicas el listado de columnas, el orden y la cardinalidad de VALUES deben coincidir exactamente con la definición de la tabla.

UPDATE: modifica filas existentes. Sintaxis básica:

```
UPDATE nombre_tabla  
SET columna1 = nuevo_valor1, columna2 = nuevo_valor2  
WHERE condicion;
```

- La **cláusula WHERE es vital**: sin ella, actualizarás **todas** las filas. La buena práctica es probar primero la condición con un SELECT para ver qué filas se verán afectadas.

DELETE FROM: elimina filas existentes. Sintaxis básica:

`DELETE FROM nombre_tabla WHERE condicion;`

- De nuevo, el **WHERE** es imprescindible para acotar el alcance. Un DELETE sin WHERE vacía la tabla (pero no la borra; la estructura sigue existiendo).

```
attachEvent("onreadystatechange",H),e.attachE  
boolean Number String Function Array Date RegE  
_={};function F(e){var t=_[e]={};return b.ea  
t[1]==!=1&&e.stopOnFalse){r=!1;break}n=!1,u&  
?o=u.length:r&&(s=t,c(r))}return this},remove  
unction(){return u=[],this},disable:function()  
re:function(){return p.fireWith(this,argument  
ending"},r={state:function(){return n},always:  
romise)?e.promise().done(n.resolve).fail(n.re  
dd(function(){n=s},t[1^e][2].disable,t[2][2].  
=0,n=h.call(arguments),r=n.length,i=1!==r||e&  
(r),l=Array(r);r>t;t++)n[t]&&b.isFunction(n[t]  
><table></table><a href='/a'>a</a><input type  
/TagName("input")[0],r.style.cssText="top:1px  
test(r.getAttribute("style")),hrefNormalized:
```

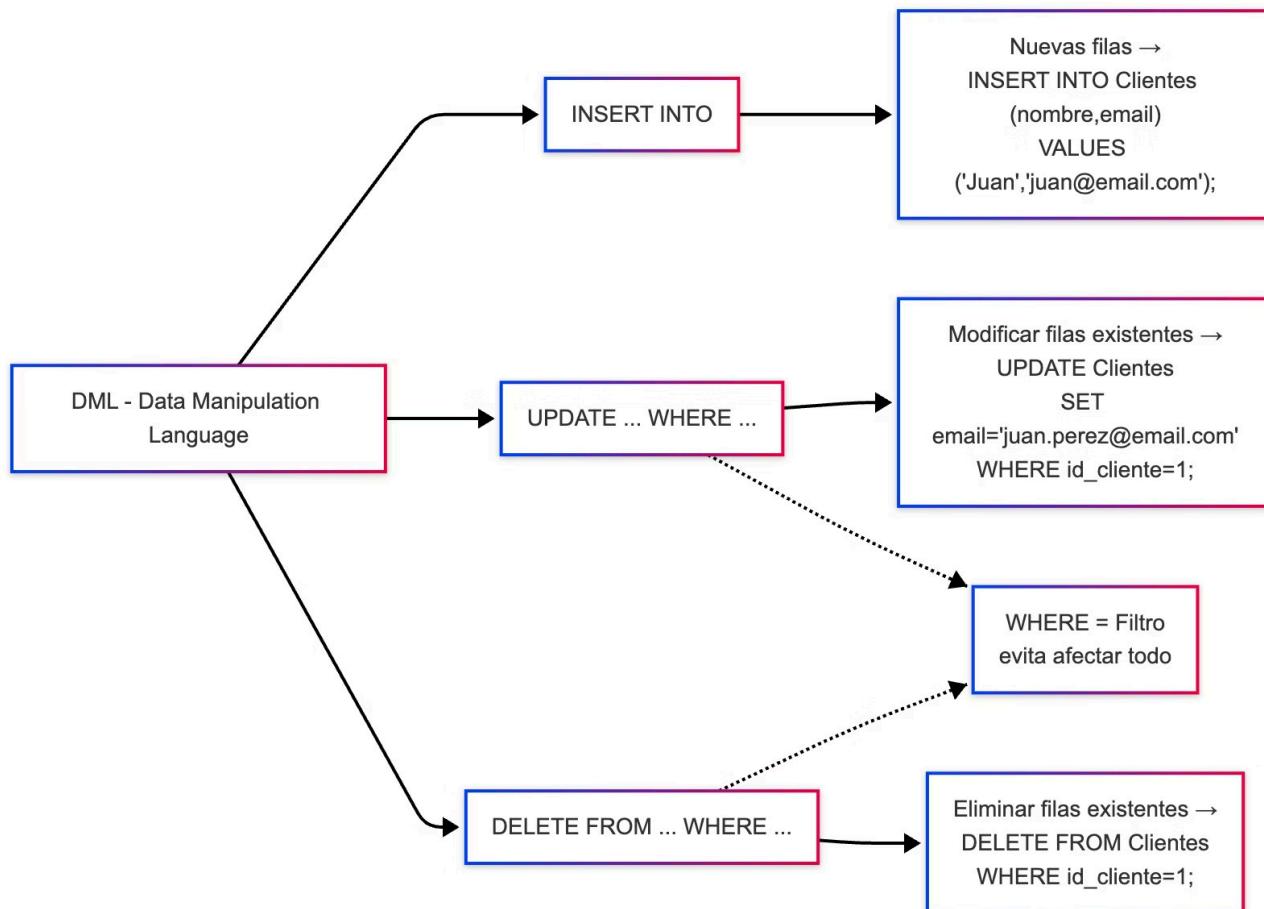
En el entorno profesional, DML trabaja mano a mano con otros elementos:

- **Transacciones** (BEGIN/COMMIT/ROLLBACK o START TRANSACTION): te permiten agrupar varios DML en una unidad atómica. Si algo falla, haces ROLLBACK y la base vuelve a un estado consistente.
 - **Restricciones e índices**: protegen la integridad (PRIMARY/FOREIGN KEY, UNIQUE, CHECK) y aceleran búsquedas. Un UPDATE o DELETE sobre columnas indexadas suele ser más rápido, pero también puede requerir cuidado para no bloquear excesivamente.
 - **Políticas de borrado**: en sistemas críticos se prefiere el **borrado lógico** (marcar activo = 0 o deleted_at con fecha) para mantener histórico y auditoría; el **borrado físico** (DELETE) se reserva para casos muy controlados.

La idea clave: **DML es precisión**. Una condición mal puesta puede afectar millones de filas. Por eso, tus mejores aliados serán la disciplina, el SELECT previo de comprobación y, cuando proceda, las transacciones.

Esquema visual

El siguiente diagrama resume los tres comandos DML y su flujo mental de uso (añadir, modificar, borrar), destacando el papel central de la **cláusula WHERE** para UPDATE y DELETE.



ⓘ Lectura del esquema:

- **INSERT** agrega registros nuevos y no necesita WHERE.
- **UPDATE** y **DELETE** siempre deben acompañarse de WHERE (tu “número de portal y piso”). Es la única forma de apuntar al registro correcto sin “pulsar todos los timbres”.
- Antes de un UPDATE o DELETE, **ensaya tu WHERE con un SELECT** para confirmar el conjunto de filas impactadas.

Caso de estudio

La gestión de mensajes en WhatsApp

Contexto

Cada interacción en WhatsApp dispara operaciones DML en tiempo real a una escala descomunal. Al enviar, recibir, leer o eliminar un mensaje, hay inserciones, actualizaciones y borrados sucediendo en la base de datos del servidor y, en ocasiones, también en la base de datos local del dispositivo.

Ejecución (tras bambalinas, simplificado):

INSERT (nuevo mensaje):

```
INSERT INTO MENSAJES (id_mensaje,  
id_chat, emisor, receptor, cuerpo,  
fecha_hora, estado)  
VALUES (... , ... , 'userA', 'userB', 'Hola', '2025-  
10-29 14:35:10', 'enviado');
```

- El sistema registra el contenido, quién lo envía/recibe, hora y estado inicial.

DELETE/UPDATE (eliminar):

"Eliminar para mí" (en tu dispositivo):

```
DELETE FROM MENSAJES_LOCAL WHERE  
id_mensaje = ...;
```

"Eliminar para todos" (en otros clientes puede reflejarse como actualización del contenido):

```
UPDATE MENSAJES  
SET cuerpo = 'Este mensaje fue eliminado',  
es_eliminado = 1  
WHERE id_mensaje = ...;
```

UPDATE (cambio de estado):

Recibido:

```
UPDATE MENSAJES  
SET estado = 'entregado', fecha_entrega =  
CURRENT_TIMESTAMP  
WHERE id_mensaje = ...;
```

Leído (doble tick azul):

```
UPDATE MENSAJES  
SET estado = 'leido', fecha_lectura =  
CURRENT_TIMESTAMP  
WHERE id_mensaje = ...;
```

Resultado

Miles de millones de **INSERT**, **UPDATE** y **DELETE** por día sostienen la experiencia fluida de mensajería. La eficiencia de estas operaciones (índices adecuados, transacciones, colas de escritura) y la **precisión** de las condiciones WHERE y claves son críticas para garantizar **consistencia, rendimiento y experiencia de usuario** en tiempo real.

Herramientas y consejos

1. Verifica con SELECT antes de UPDATE/DELETE

La regla de oro: **previsualiza** el impacto.

-- Ensayo del filtro:

```
SELECT * FROM Clientes WHERE  
id_cliente = 1;
```

-- Si el resultado es el esperado:

```
UPDATE Clientes SET email =  
'nuevo@email.com' WHERE id_cliente =  
1;
```

Este hábito te ahorra sustos y evita daños masivos "por accidente".

2. Inserta varias filas en un único INSERT

Ahorra viajes al servidor y acelera cargas iniciales:

```
INSERT INTO Productos (id_producto,  
nombre, precio, stock, id_categoria)  
VALUES  
(1, 'Portátil Pro', 1200.50, 50, 1),  
(2, 'Monitor 27"', 299.90, 120, 2),  
(3, 'Ratón Inalámbrico', 24.95, 500, 3);
```

3. Usa transacciones para cambios relacionados

Asegura consistencia en operaciones múltiples:

```
START TRANSACTION;
```

```
UPDATE Pedidos SET estado = 'ENVIADO' WHERE id_pedido = 1001;  
UPDATE Inventario SET stock = stock - 1 WHERE id_producto = 55;
```

```
COMMIT; -- o ROLLBACK si algo falla
```

Si cualquiera de las sentencias falla, revierte todo con ROLLBACK para no dejar datos "a medias".

4. Apóyate en clientes profesionales

DBeaver, MySQL Workbench, pgAdmin ofrecen vista en cuadrícula para editar "como Excel". Cuando guardas, generan los UPDATE que harías a mano. Úsalos para aprender y para tareas rutinarias, pero **revisa siempre** el SQL que lanzan.

5. Protege el WHERE con claves

Asegúrate de que tu WHERE se base en **claves primarias** o **campos únicos** cuando sea posible. Evita filtros ambiguos que puedan coincidir con más filas de las previstas.

6. Prefiere actualización a borrado-reinserción

Cambiar datos con UPDATE es más **seguro** y **eficiente** que borrar y reinsertar; así no rompes claves foráneas ni historiales asociados.

7. Borrado lógico vs. físico

En negocios con auditoría o normativa (finanzas, salud), **borrado lógico**:

```
UPDATE Clientes SET activo = 0, fecha_baja = CURRENT_DATE WHERE id_cliente = 1;
```

Mantienes historial y cumples trazabilidad. El borrado físico (**DELETE**) debe ir acompañado de copias de seguridad y políticas claras.

8. Cuida el rendimiento

- **Índices** en columnas usadas frecuentemente en WHERE.
- **Límites** en actualizaciones masivas: aplicar por lotes (por ejemplo, de 10 000 en 10 000) para evitar bloqueos prolongados.
- **Registra** cuántas filas afectaste (ROW_COUNT() o salida de la herramienta) para validar el impacto esperado.

Mitos y realidades

✗ MITO: "DELETE FROM tabla; es lo mismo que DROP TABLE tabla;"

→ **FALSO:** DELETE FROM tabla; (sin WHERE) **borra todas las filas** pero **no** elimina la estructura (la tabla sigue existiendo). Es DML. DROP TABLE tabla; **elimina la tabla completa**, su definición y sus datos. Es DDL e **irreversible** sin copia de seguridad.

✓ Realidad: Son operaciones distintas, con alcances y riesgos diferentes.

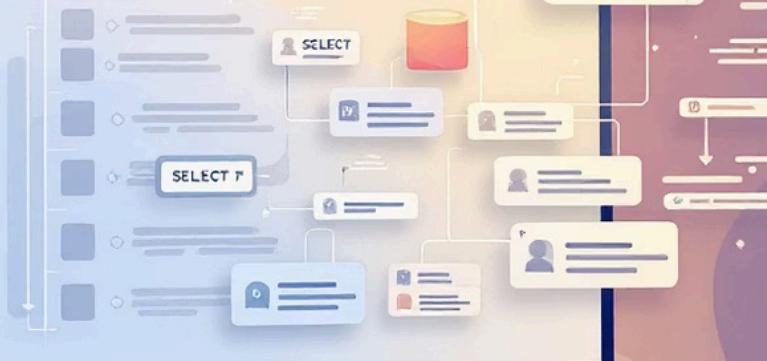
✗ MITO: "Para cambiar un dato, mejor borrar la fila y volver a insertarla."

→ **FALSO:** Borrar-reinsertar rompe integridad referencial (claves foráneas apuntan a otra PK), **pierde histórico** y es **menos eficiente**.

✓ Realidad: Usa **UPDATE**. Está diseñado para modificar de forma segura un registro existente conservando sus relaciones e identificadores.

□ Resumen final

- **DML** = manipular datos en tablas: **INSERT, UPDATE (con WHERE), DELETE (con WHERE)**.
- **INSERT** añade filas; **UPDATE** modifica; **DELETE** elimina filas existentes.
- **WHERE es vital** en UPDATE/DELETE para acotar el impacto.
- **DELETE ≠ DROP**: DELETE borra filas (DML); DROP borra la tabla entera (DDL).
- **Buena práctica**: ejecutar antes un SELECT ... WHERE ... de verificación; usar transacciones cuando el cambio sea crítico.



Sesión 19: Consultas simples con SELECT y operadores básicos

El comando **SELECT** es el núcleo del lenguaje SQL (Structured Query Language) y el punto de partida de toda interacción con una base de datos relacional. Dominarlo significa entender cómo transformar una masa de datos en información útil, comprensible y accionable. En entornos empresariales, casi todas las decisiones basadas en datos comienzan con una consulta SELECT.

Imagina una base de datos como una gran biblioteca. Cada tabla es una estantería, cada fila es un libro y cada columna es una característica de ese libro (autor, año, género...). El comando SELECT sería el bibliotecario que te ayuda a encontrar los títulos que cumplen tus criterios: "muéstrame todos los libros escritos después de 2015 y que sean de ciencia ficción". SQL traduce esta solicitud en una instrucción formal que el sistema puede entender y ejecutar.

La estructura básica de una consulta SELECT sigue una lógica natural:

```
SELECT columna1, columna2, ...
FROM nombre_tabla
WHERE condición;
```

SELECT indica qué información deseas recuperar. Si quieres todas las columnas, puedes usar el comodín *, aunque solo se recomienda para pruebas o exploraciones iniciales.

FROM define de dónde proviene esa información, es decir, el nombre de la tabla o las tablas que contienen los datos.

WHERE establece qué *condiciones* deben cumplir las filas para ser incluidas en el resultado. Es opcional, pero en la práctica se utiliza en la mayoría de las consultas para filtrar información.

En la cláusula WHERE entran en juego los **operadores**, que son los elementos lógicos que permiten construir las condiciones de filtrado. Los más importantes son:

- **Operadores de comparación:** =, !=, <, >, <=, >=
 - **Operadores lógicos:** AND, OR, NOT
 - **Operadores especiales:**
 - BETWEEN: para seleccionar valores dentro de un rango.
 - IN: para buscar coincidencias dentro de una lista.
 - LIKE: para realizar búsquedas por patrones en texto.
 - IS NULL / IS NOT NULL: para evaluar valores nulos.

Por ejemplo, si trabajas con una tabla **Productos**, podrías ejecutar:

```
SELECT nombre, precio  
FROM Productos  
WHERE precio > 100 AND id_categoria = 2;
```

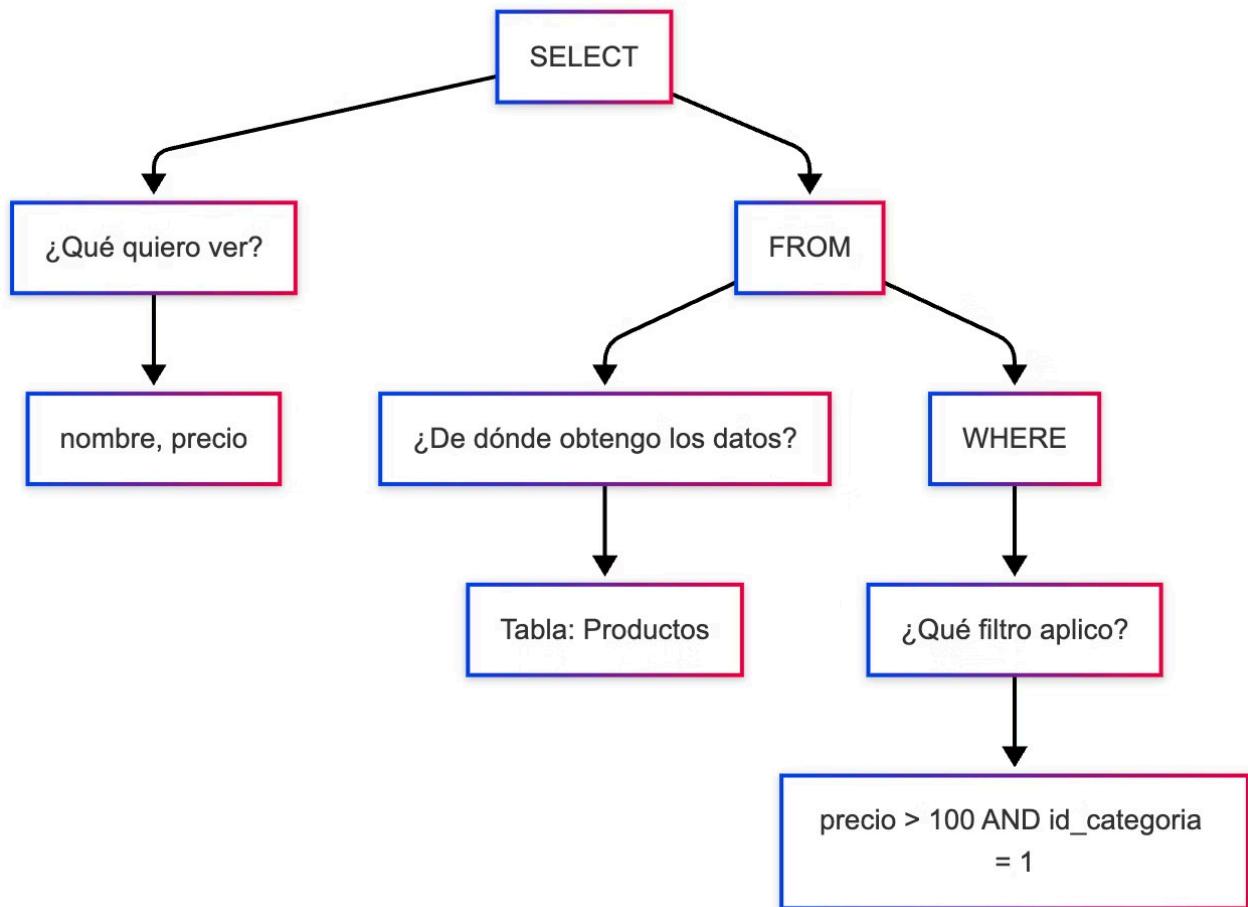
Esta instrucción selecciona únicamente los nombres y precios de los productos de la categoría 2 cuyo precio supere los 100 €. En una empresa real, una consulta de este tipo podría alimentar un informe de ventas, una aplicación web o incluso un dashboard financiero.

La potencia de SELECT radica en su **flexibilidad**. Con unas pocas palabras clave puedes construir consultas que van desde una simple lista hasta complejas combinaciones con filtros, ordenaciones o cálculos. Por eso se dice que SELECT es la herramienta que convierte los datos en conocimiento: permite a los analistas formular preguntas y obtener respuestas inmediatas, con precisión milimétrica.

```
    <div class="height-20p"></div>
    <div class="container">
        <section id="feature-post" class="py-5">
            <div class="row align-items-center">
                <div class="col-lg-7">
                    <div>
                        <h1 class="text-lg font-weight-bold text-gray-800">
                            Having a
                            Very Good Experience
                            With Laravel Development
                        </h1>
                        <div class="height-20p"></div>
                        <p class="text-regular font-weight-300 text-gray-800">
                            This repository provides you a development environment
                            within a GitHub
                        </p>
                        <div class="height-30p"></div>
                        <div class="d-flex">
                            <a href="/" class="btn btn-primary font-weight-500 px-2 py-1">Go to GitHub</a>
                            <a href="/" class="d-flex align-items-center font-weight-300 px-2 py-1">
                                
                                <span class="ml-2 font-weight-300">Go to GitHub repository</span>
                            </a>
                        </div>
                    </div>
                </div>
            </div>
        </section>
    </div>
```

Esquema visual

Anatomía de una consulta SELECT



ⓘ Descripción del esquema:

1. **SELECT** define las columnas que se mostrarán en el resultado. Es la respuesta a "¿qué quiero ver?".
2. **FROM** señala la tabla o tablas que contienen la información. Es la respuesta a "¿de dónde la obtengo?".
3. **WHERE** establece los filtros o condiciones. Es la respuesta a "¿qué registros quiero excluir o incluir?".

La ejecución sigue un flujo interno: primero el sistema accede a la tabla (FROM), luego aplica los filtros (WHERE) y finalmente muestra las columnas solicitadas (SELECT). Entender esta secuencia es clave para optimizar las consultas y evitar errores lógicos.

YouTube Search

Nuevo video



Mi Perfil | Mis Subcripciones | Mis Compras

Caso de estudio

Las búsquedas de productos en YouTube

Contexto

Cada vez que escribes "mejores portátiles 2024" en YouTube, el sistema ejecuta una consulta **SELECT** sobre una de las bases de datos más grandes del planeta. Detrás de esa simple búsqueda hay millones de filas y miles de condiciones que se procesan en milisegundos.

Estrategia

En una versión muy simplificada, la consulta podría representarse así:

```
SELECT titulo, url_video, nombre_canal  
FROM Videos  
WHERE (titulo LIKE '%mejores portátiles 2024%'  
      OR descripcion LIKE '%mejores portátiles 2024%')  
      AND fecha_publicacion >= '2024-01-01';
```

- El operador **LIKE** con los comodines % permite buscar patrones dentro del texto.
- El **OR** combina dos condiciones para que se cumpla al menos una.
- El **AND** añade un segundo filtro temporal, limitando los resultados a vídeos publicados en 2024.

Resultado

Gracias a esta lógica, el sistema devuelve una lista precisa de vídeos recientes y relevantes. Este mismo principio se replica en todos los motores de búsqueda modernos (Google, Amazon, Spotify, Netflix). SELECT y WHERE son la base que permite navegar por billones de datos y obtener información exacta al instante.

En empresas de análisis de datos o e-commerce, dominar SELECT es tan importante como saber leer un balance contable: define la capacidad de interpretar la realidad del negocio.

Herramientas y consejos

- Practica con entornos interactivos de SQL

Plataformas como **SQLZoo**, W3Schools SQL Tryit o Mode Analytics te permiten escribir y ejecutar consultas sin necesidad de instalar nada. Son ideales para afianzar la lógica de SELECT.

- Evita SELECT * en entornos productivos

Aunque es útil en pruebas, seleccionar todas las columnas sobrecarga el sistema y puede romper el código si se modifica la estructura de la tabla. Nombra solo lo que necesitas.

- Cuidado con los valores nulos

En SQL, NULL significa "sin valor", no "cero" ni "vacío". Por eso no puedes comparar con = NULL. Usa IS NULL o IS NOT NULL. Ejemplo:

```
SELECT nombre FROM Productos WHERE  
precio IS NULL;
```

- Experimenta con condiciones combinadas

Combina AND, OR y NOT para filtrar con precisión. Por ejemplo:

```
SELECT * FROM Empleados  
WHERE salario > 30000  
AND (departamento = 'Ventas' OR  
departamento = 'Marketing');
```

- Usa un entorno visual para aprender más rápido

Si prefieres trabajar con interfaz gráfica, **DBeaver**, **HeidiSQL** o **SQLite Studio** son herramientas gratuitas que te permiten ver las tablas, ejecutar consultas y visualizar los resultados en tiempo real.

- Domina los comodines del operador LIKE

- % representa *cero o más caracteres*: LIKE '%phone%' encuentra "iPhone", "Smartphone" o "Headphone".
- _ representa *un solo carácter*: LIKE 'a_p' encuentra "app" o "a3p", pero no "aerop".

- Ordena tus resultados con ORDER BY

Añade ORDER BY columna ASC (ascendente) o DESC (descendente) para organizar tus datos.

```
SELECT nombre, precio FROM Productos  
ORDER BY precio DESC;
```

- Analiza tus consultas

Muchos sistemas (como MySQL Workbench o PostgreSQL) incluyen la opción **EXPLAIN** para ver cómo se ejecuta tu consulta internamente. Es útil para aprender sobre optimización.

Mitos y realidades

X MITO: "SELECT * es la forma más fácil y mejor de consultar los datos."

→ **FALSO.** Aunque es útil al explorar una tabla nueva, usar SELECT * en entornos reales es ineficiente. Carga datos innecesarios, ralentiza la transmisión y puede provocar errores si la tabla cambia.

✓ Realidad: Selecciona solo las columnas necesarias:

```
SELECT nombre, precio FROM Productos;
```

Esto hace tus consultas más rápidas y tu código más mantenable.

X MITO: "Las consultas SQL son siempre lentas."

→ **FALSO.** La lentitud rara vez está en SQL como lenguaje. Generalmente, el problema está en un mal diseño de la base de datos (falta de índices, consultas redundantes, estructuras mal normalizadas).

✓ Realidad: Un SELECT optimizado sobre una base bien indexada puede analizar millones de filas en milisegundos. Empresas como Amazon o YouTube realizan millones de consultas simultáneas con una eficiencia extrema.

❑ Resumen final

- **SELECT:** comando para recuperar información.
- **FROM:** indica la tabla de origen.
- **WHERE:** filtra las filas según condiciones.
- **Operadores de comparación:** =, !=, <, >, <=, >=
- **Operadores lógicos:** AND, OR, NOT
- **LIKE:** búsqueda de patrones (usa % y _)
- **BETWEEN:** define rangos. **IN:** listas de valores.
- **ORDER BY:** ordena resultados (ASC o DESC).
- **IS NULL / IS NOT NULL:** evalúa valores nulos.
- *Evita SELECT *:* usa solo las columnas necesarias



Sesión 20: Funciones, agrupaciones y ordenaciones

El comando **SELECT** de SQL no solo sirve para recuperar filas individuales, sino también para **realizar cálculos y análisis sobre conjuntos de datos**. Aquí entran en juego las **funciones de agregación**, la **agrupación de resultados** con GROUP BY, y la **ordenación** con ORDER BY.

Estas tres herramientas son la base del análisis de datos dentro de una base de datos relacional. Gracias a ellas, pasamos de ver *datos aislados* a obtener *información resumida y significativa*.

Funciones de agregación

Son funciones matemáticas aplicadas a una o más columnas. Devuelven un único valor por conjunto de filas. Las más comunes son:

Función	Descripción
COUNT()	Cuenta el número de filas o valores no nulos.
SUM()	Suma los valores numéricos de una columna.
AVG()	Calcula el promedio.
MAX() / MIN()	Obtiene el valor máximo o mínimo, respectivamente.

Por ejemplo, si queremos saber cuántos productos hay en la tabla *Productos*:

```
SELECT COUNT(*) AS total_productos FROM Productos;
```

Y si deseamos calcular el precio medio de todos los productos:

```
SELECT AVG(precio) AS precio_medio FROM Productos;
```

Agrupación con GROUP BY

Cuando queremos aplicar funciones de agregación **por categorías**, usamos la cláusula GROUP BY. Por ejemplo:

```
SELECT id_categoria, AVG(precio) AS precio_medio  
FROM Productos  
GROUP BY id_categoria;
```

Esto devuelve un promedio por cada categoría, no un único resultado global.

Filtrado de grupos con HAVING

A diferencia de WHERE, que filtra filas antes de agrupar, HAVING filtra **después** de la agregación. Por ejemplo:

```
SELECT id_cliente, SUM(importe) AS total_gastado  
FROM Pedidos  
GROUP BY id_cliente  
HAVING SUM(importe) > 1000;
```

Este comando muestra solo a los clientes cuyo gasto total supera los 1000 €.

Ordenación con ORDER BY

Para ordenar los resultados, usamos la cláusula ORDER BY, al final de la consulta:

```
SELECT pais, AVG(salario) AS salario_medio  
FROM Empleados  
GROUP BY pais  
ORDER BY salario_medio DESC;
```

ASC (por defecto) ordena de menor a mayor, y DESC de mayor a menor.

En conjunto, **SELECT + funciones + GROUP BY + ORDER BY** forman la base de todo análisis descriptivo en SQL.

Esquema visual

Flujo lógico de una consulta con agregación

01

FROM / WHERE

Seleccionan las filas base

02

GROUP BY

Agrupa los registros según uno o varios campos

03

HAVING

Filtra los grupos ya formados

04

SELECT con funciones

Aplica funciones de agregación y define las columnas de salida

05

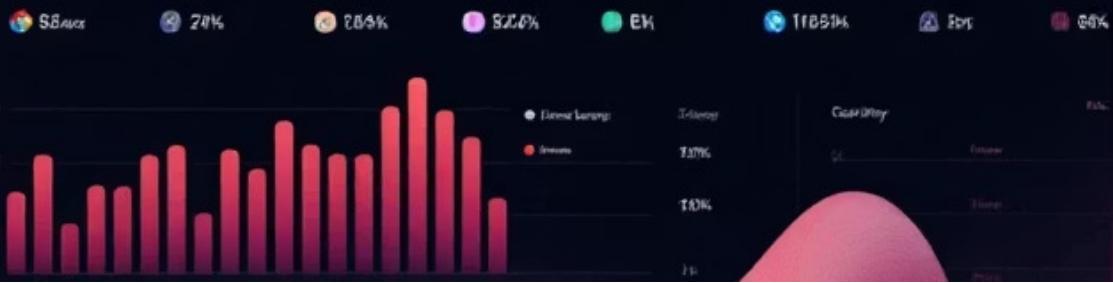
ORDER BY

Organiza los resultados para su presentación

ⓘ Descripción:

1. **FROM / WHERE:** seleccionan las filas base.
2. **GROUP BY:** agrupa los registros según uno o varios campos.
3. **HAVING:** filtra los grupos ya formados.
4. **SELECT:** aplica funciones de agregación y define las columnas de salida.
5. **ORDER BY:** organiza los resultados para su presentación.

Entender este orden **lógico** (que no siempre coincide con el orden de escritura) es esencial para escribir consultas precisas y eficientes.



Caso de estudio

El análisis de visualizaciones de Netflix

Contexto

Netflix analiza diariamente miles de millones de reproducciones. Para saber qué contenido funciona mejor, necesita **agrupar y resumir grandes volúmenes de datos**: país, tipo de contenido, dispositivo, etc.

Estrategia

Una consulta simplificada para conocer las series más vistas por país en el último mes podría ser:

```
SELECT id_serie, pais, COUNT(*) AS visualizaciones
FROM Visualizaciones
WHERE fecha >= '2024-10-01'
GROUP BY id_serie, pais
ORDER BY visualizaciones DESC;
```

- COUNT(*) cuenta cada visualización.
- GROUP BY agrupa por serie y país.
- ORDER BY muestra los resultados ordenados desde el más popular.

Resultado

Netflix obtiene una clasificación precisa de qué series funcionan mejor en cada región. Con esos datos puede:

- Promocionar contenido local.
- Tomar decisiones de producción.
- Ajustar sus recomendaciones personalizadas.

Este tipo de consultas son la base del **Business Intelligence (BI)**: convertir datos operativos en **insights estratégicos**.

Herramientas y consejos

Domina los alias con AS

Asigna nombres descriptivos a las columnas agregadas:

```
SELECT COUNT(*) AS total_pedidos  
FROM Pedidos;
```

Distingue entre WHERE y HAVING

- WHERE → filtra **filas** antes de agrupar.
- HAVING → filtra **grupos** después de agrupar.

Por ejemplo:

```
SELECT id_cliente, SUM(importe) AS  
total  
FROM Pedidos  
WHERE fecha >= '2025-01-01'  
GROUP BY id_cliente  
HAVING total > 500;
```

Usa DISTINCT cuando necesites valores únicos

```
SELECT DISTINCT pais FROM Clientes;
```

Recuerda el orden de ejecución real en SQL

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY. Este flujo interno explica por qué no se puede usar un alias de columna dentro del WHERE.

Analiza tus consultas con EXPLAIN

En MySQL o PostgreSQL te muestra cómo el motor ejecuta la consulta, ayudando a optimizar rendimiento.

Evita errores comunes

- No incluyas columnas no agregadas sin incluirlas en el GROUP BY.
- No confundas COUNT(*) con COUNT(columna) (esta última no cuenta valores NULL).

Herramientas recomendadas

- **Mode Analytics** o **SQLPad** para practicar consultas con datasets reales.
- **DBeaver / DataGrip** para ejecutar SQL visualmente.

Mitos y realidades

✗ MITO: "Las funciones de agregación son lentas."

→ **FALSO.** Los sistemas de bases de datos están optimizados para calcular agregaciones rápidamente.

✓ Realidad: Calcular una media con AVG(precio) en SQL es más rápido y eficiente que hacerlo desde tu aplicación en Python o Excel.

✗ MITO: "HAVING sustituye a WHERE."

→ **FALSO.** Cumplen roles distintos. HAVING no se ejecuta si no hay agrupación.

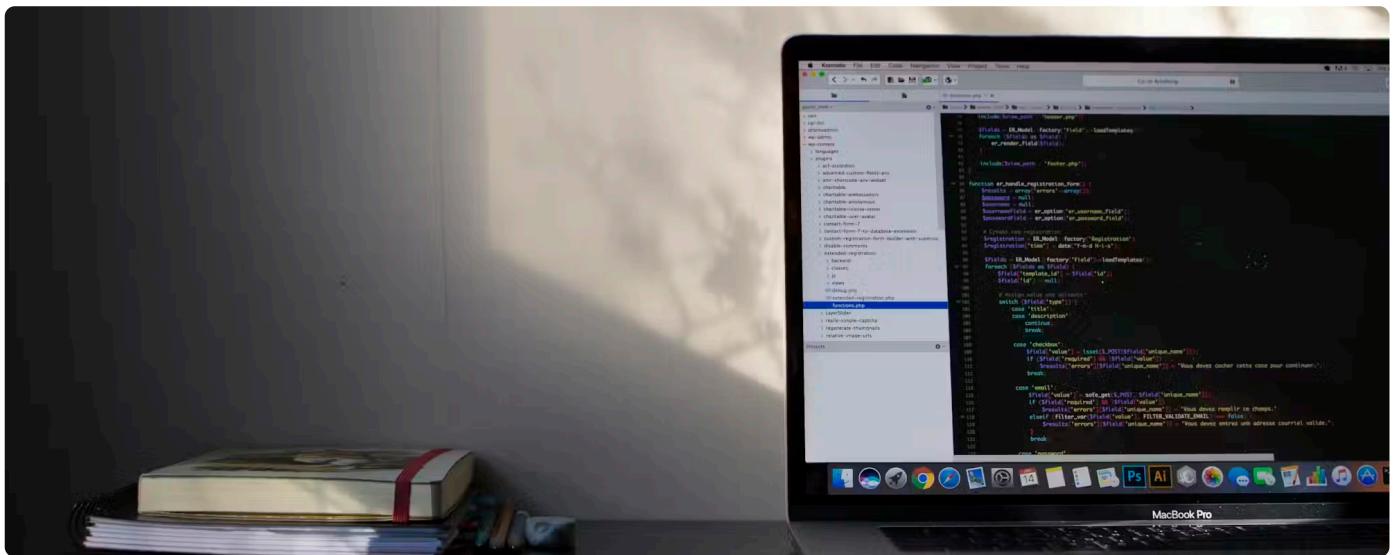
✓ Realidad: Usa WHERE para filtrar registros y HAVING para filtrar **grupos**.

✗ MITO: "No se puede usar WHERE y HAVING juntos."

→ **FALSO:** Es común combinarlos. WHERE filtra antes, HAVING después.

✓ Realidad:

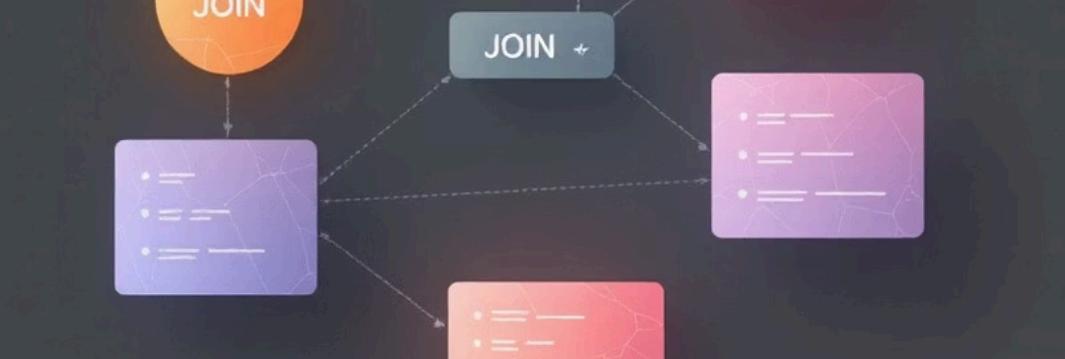
```
SELECT pais, COUNT(*) AS num_empleados
FROM Empleados
WHERE activo = TRUE
GROUP BY pais
HAVING COUNT(*) > 10;
```



☐ Resumen final

Concepto	Descripción
Funciones de agregación	COUNT, SUM, AVG, MAX, MIN.
GROUP BY	Agrupa registros que comparten valores.
HAVING	Filtira grupos después de aplicar una función de agregación.
AS	Crea alias descriptivos.
DISTINCT	Elimina duplicados.
ORDER BY	Ordena los resultados (ASC/DESC).
Orden lógico de ejecución	FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY.





Sesión 21: Consultas multitable: JOINS y subconsultas

El verdadero poder de una base de datos relacional no está en almacenar información, sino en **conectarla**. La razón de ser de los sistemas relacionales es precisamente su capacidad para **relacionar** datos que están distribuidos en diferentes tablas, permitiendo consultas complejas que extraen valor real de la información. Aquí entran en juego dos herramientas esenciales: los **JOINS** y las **subconsultas (subqueries)**.

Un **JOIN** combina filas de dos o más tablas basándose en una relación lógica entre sus columnas, normalmente una **clave primaria (PK)** y una **clave foránea (FK)**. Gracias a los JOINs, puedes reconstruir una visión coherente de datos que fueron **normalizados** para evitar redundancia y asegurar integridad. Por ejemplo, en una base de datos de ventas, los datos de los clientes pueden estar en una tabla y los de sus pedidos en otra. Sin un JOIN, sería imposible cruzar ambos mundos y responder preguntas como "¿qué clientes han comprado en el último mes?".

Existen varios tipos de JOIN, cada uno con un propósito específico:

INNER JOIN: Devuelve solo las filas que tienen coincidencias en ambas tablas. Es el más utilizado porque filtra automáticamente lo que no coincide.

LEFT JOIN (LEFT OUTER JOIN): Devuelve todas las filas de la tabla izquierda, aunque no tengan correspondencia en la derecha. Muy útil para detectar ausencias o registros huérfanos.

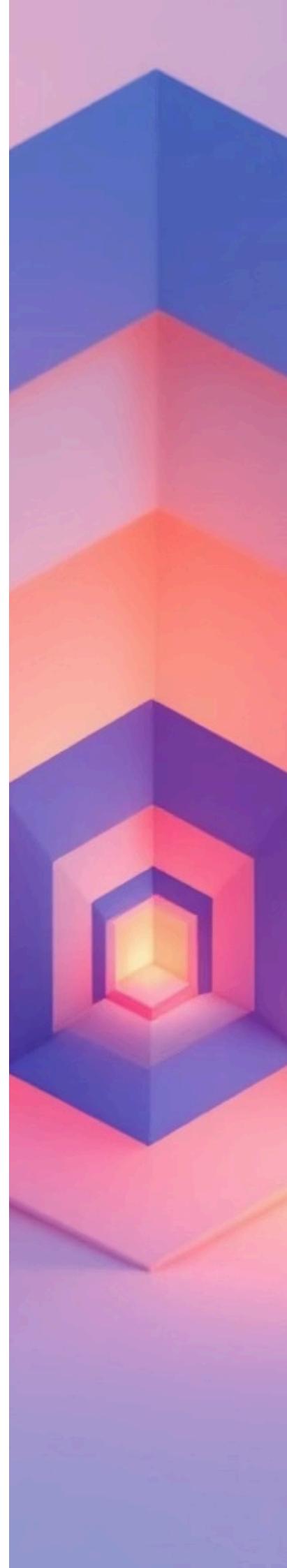
RIGHT JOIN (RIGHT OUTER JOIN): Igual que el anterior, pero partiendo de la tabla derecha.

FULL OUTER JOIN: Devuelve todas las filas de ambas tablas, uniendo cuando hay coincidencias y mostrando NULL cuando no las hay.

Por otro lado, una **subconsulta** es una **consulta anidada dentro de otra**. Sirve para filtrar, calcular o incluso generar tablas intermedias sobre las que luego se hace otra operación. Por ejemplo, puedes buscar todos los clientes cuyo id_cliente aparece en una subconsulta que obtiene los pedidos recientes. La sintaxis más común es colocar la subconsulta dentro de una cláusula WHERE, pero también puede usarse en FROM (como tabla virtual) o incluso en SELECT (para calcular valores derivados).

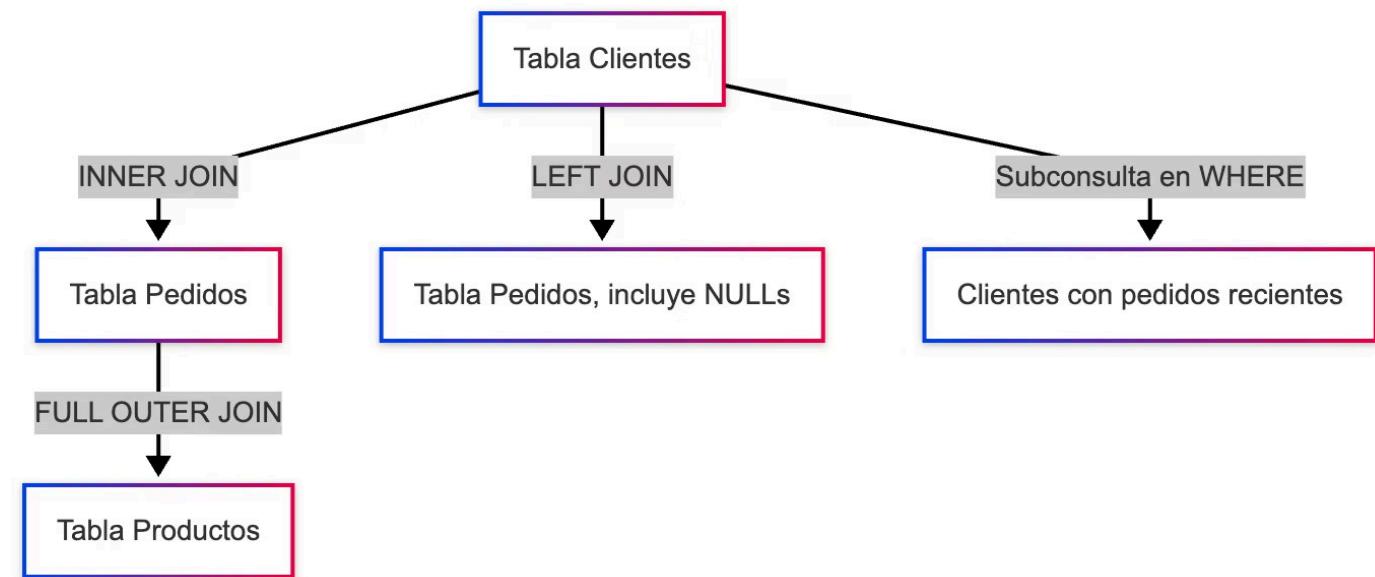
- ⓘ En la práctica profesional, ambos mecanismos —JOINS y subconsultas— son complementarios. Los **JOINS** son más legibles y eficientes para combinar datos de forma directa, mientras que las **subconsultas** son ideales para filtrados complejos o cuando necesitas calcular algo antes de compararlo. Un analista o desarrollador SQL experto sabe cuándo usar cada uno para lograr el equilibrio entre claridad, rendimiento y mantenibilidad.

En un entorno real como el de una empresa de ecommerce, estas operaciones permiten construir dashboards de ventas, informes de clientes o control de stock en tiempo real. Saber dominar los JOINs y las subconsultas te convierte en alguien capaz de **traducir la estructura técnica de la base de datos en información de negocio útil y accionable**.



Esquema visual

Los tipos de JOIN y su relación con las subconsultas



Alternativa: Subconsulta en WHERE

Filtrá clientes con pedidos recientes usando una subconsulta:

```
SELECT nombre FROM Clientes  
WHERE id_cliente IN  
(SELECT id_cliente FROM Pedidos  
WHERE fecha > '2024-01-01');
```

ⓘ Descripción del esquema:

- **INNER JOIN** crea combinaciones directas mostrando solo coincidencias entre tablas.
- **LEFT JOIN** incluye todos los clientes aunque no tengan pedidos (los campos de la tabla derecha aparecerán como NULL).
- **FULL OUTER JOIN** muestra todos los pedidos y todos los productos, uniendo la información cuando hay coincidencia.
- **Subconsulta en WHERE** simboliza la alternativa de filtrar los clientes con pedidos recientes, usando una subconsulta como filtro.

El esquema refleja cómo los JOINs crean combinaciones directas entre tablas, mientras que las subconsultas se utilizan para filtrar o derivar subconjuntos de datos antes de aplicar la consulta principal.



Caso de estudio

Amazon y las consultas de pedidos

Contexto

Amazon gestiona cientos de millones de clientes, productos y pedidos cada día. Cada vez que un usuario entra en la sección "**Mis pedidos**", el sistema debe combinar información procedente de múltiples fuentes: datos del cliente, pedidos realizados, artículos comprados, detalles de envío, facturación y más.

Estrategia

Para mostrar en pantalla una lista coherente de pedidos, Amazon ejecuta consultas con múltiples **JOINs**. Simplificadamente, una de ellas podría representarse así:

```
SELECT P.id_pedido, P.fecha, PR.nombre AS producto, LP.cantidad  
FROM Pedidos AS P  
INNER JOIN LineaPedido AS LP ON P.id_pedido = LP.id_pedido  
INNER JOIN Productos AS PR ON LP.id_producto = PR.id_producto  
WHERE P.id_cliente = 12345;
```

Esta consulta combina tres tablas:

- **Pedidos (P)** → contiene los datos generales del pedido.
- **LineaPedido (LP)** → detalla los productos incluidos en cada pedido.
- **Productos (PR)** → almacena los nombres y precios de los artículos.

La clave está en las relaciones `P.id_pedido = LP.id_pedido` y `LP.id_producto = PR.id_producto`, que permiten unir la información en una sola vista.

Resultado

El resultado final es una lista de productos por pedido para un cliente específico, incluyendo cantidades y fechas. Esto demuestra cómo los **JOINs** son el puente que conecta los datos dispersos por la normalización. En sistemas de este tamaño, la eficiencia de las uniones es crítica: cada milisegundo cuenta. Amazon optimiza estas operaciones mediante índices, caché de consultas y particionado de datos, pero la base conceptual sigue siendo la misma: **JOINs bien diseñados que reconstruyen la historia completa del cliente**.

Herramientas y consejos

1 Usa alias para mejorar legibilidad

Cuando trabajas con múltiples tablas, los alias hacen tu código más claro. Por ejemplo:

```
SELECT C.nombre, P.fecha  
FROM Clientes AS C  
INNER JOIN Pedidos AS P ON  
C.id_cliente = P.id_cliente;
```

Esto evita confusión y facilita la lectura del SQL, sobre todo cuando las columnas tienen nombres similares.

2 Aprovecha el LEFT JOIN para detectar "ausencias"

Una consulta muy útil para auditorías o informes de inactividad es encontrar qué registros no tienen correspondencia.

```
SELECT C.nombre  
FROM Clientes AS C  
LEFT JOIN Pedidos AS P ON  
C.id_cliente = P.id_cliente  
WHERE P.id_pedido IS NULL;
```

Este patrón te permite descubrir, por ejemplo, qué clientes nunca han hecho un pedido.

3 Usa subconsultas para comprobaciones complejas

Cuando necesites filtrar basándote en resultados agregados o condicionales, las subconsultas son ideales. Por ejemplo, para buscar clientes con más de tres pedidos:

```
SELECT nombre FROM Clientes  
WHERE id_cliente IN (  
    SELECT id_cliente FROM Pedidos  
    GROUP BY id_cliente  
    HAVING COUNT(*) > 3  
)
```

4 Elige la técnica más eficiente

En la mayoría de los sistemas de gestión (MySQL, PostgreSQL, SQL Server), los JOINs suelen ser más rápidos y fáciles de optimizar. Pero en algunos escenarios (como validaciones o consultas con funciones agregadas) las subconsultas pueden resultar más limpias.

5 Herramientas recomendadas

- **DBeaver** o **Azure Data Studio** para escribir y visualizar consultas con múltiples JOINs.
- **DBVisualizer** o **pgAdmin** para analizar gráficamente los planes de ejecución.
- **SQLBolt** o **Mode SQL School**, plataformas gratuitas para practicar JOINs interactivos.

Mitos y realidades

✗ MITO: "Los JOINs son lentos; es mejor tener toda la información en una sola tabla."

→ **FALSO.** Este pensamiento surge del miedo a la complejidad. Sin embargo, las bases de datos relacionales están optimizadas para ejecutar JOINs de manera extremadamente eficiente, siempre que las columnas estén indexadas. Las tablas "gigantes" con toda la información duplicada provocan errores de consistencia, uso excesivo de espacio y enormes problemas de mantenimiento.

✓ Realidad: La **normalización** más JOINs bien diseñados son la base del rendimiento y la integridad.

✗ MITO: "JOINs y subconsultas son intercambiables siempre."

→ **FALSO.** Aunque en muchos casos se puede lograr el mismo resultado de ambas formas, no son equivalentes. Hay consultas en las que una subconsulta correlacionada es la única opción, y otras donde un JOIN resulta mucho más claro y rápido.

✓ Realidad: Un profesional SQL sabe decidir con criterio: las subconsultas son ideales para filtros, los JOINs para combinar información directa entre entidades relacionadas.

❑ Resumen final

- JOINs y subconsultas permiten **combinar información** de múltiples tablas.
- **INNER JOIN:** devuelve coincidencias entre tablas.
- **LEFT JOIN:** incluye todos los registros de la tabla izquierda, incluso sin coincidencia.
- **ON** define la condición de unión entre claves primaria y foránea.
- **Subconsulta:** una consulta dentro de otra, útil para filtrados y cálculos previos.
- Los JOINs reconstruyen la información que fue separada por la normalización y son el corazón del modelo relacional.

Sesión 22: Optimización de consultas SQL

Escribir una consulta SQL que funcione no es suficiente en el mundo profesional. En entornos reales —como un ecommerce, una app de banca o una red social— cada milisegundo cuenta. Miles de usuarios pueden estar ejecutando la misma operación al mismo tiempo, y una consulta mal diseñada puede ralentizar un sistema completo. Por eso, la **optimización de consultas SQL** es una habilidad clave para cualquier desarrollador, analista o administrador de bases de datos.

- ⓘ La optimización consiste en hacer que las consultas **se ejecuten en el menor tiempo posible** usando los recursos adecuados. La piedra angular de este proceso son los **índices**, estructuras de datos que permiten al sistema localizar información de forma casi instantánea, sin tener que recorrer toda la tabla.

Un **índice** funciona igual que el índice de un libro: en lugar de leer todas las páginas hasta encontrar lo que buscas, vas directamente al capítulo o al término indicado. En bases de datos, los índices se crean sobre una o varias columnas para **acelerar búsquedas, filtrados y uniones (JOINs)**. Por ejemplo, si una tabla de clientes tiene un millón de filas y ejecutas constantemente consultas como:

```
SELECT * FROM clientes WHERE email = 'usuario@ejemplo.com';
```

crear un índice sobre la columna email permite que el motor de la base de datos localice el registro en milisegundos, en lugar de escanear las millones de filas una por una (lo que se llama un *full table scan*).

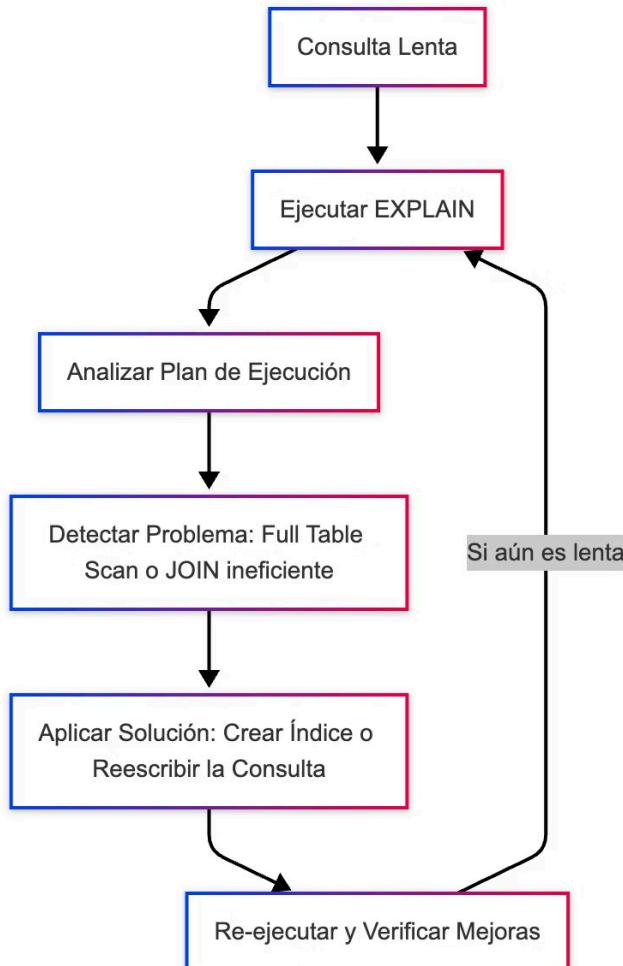
Los sistemas de gestión de bases de datos (SGBD) crean automáticamente índices sobre las **claves primarias** y las **claves foráneas**, pero es responsabilidad del profesional SQL analizar y definir **índices adicionales** en las columnas que más se utilizan para filtrar (WHERE) o para unir (JOIN).

Otra herramienta esencial es el comando **EXPLAIN** (o EXPLAIN ANALYZE, según el motor de base de datos). Este comando revela cómo el SGBD planea ejecutar una consulta: qué tablas recorrerá, qué tipo de unión usará, si aprovechará un índice o si realizará un escaneo completo. Interpretar el plan de ejecución que devuelve EXPLAIN permite identificar los **cuellos de botella** y actuar antes de que la lentitud se convierta en un problema.

En resumen, **optimizar una consulta SQL es como afinar un motor**: no se trata solo de que funcione, sino de que lo haga con precisión y eficiencia. Los índices, los planes de ejecución y las buenas prácticas son las herramientas que te permiten lograrlo.

Esquema visual

Ciclo de optimización de una consulta SQL



ⓘ Descripción del esquema:

1. **Consulta Lenta:** punto de partida, una sentencia SQL que tarda demasiado en ejecutarse.
2. **EXPLAIN:** se solicita al motor que muestre el plan de ejecución.
3. **Análisis:** se identifican los pasos más costosos y los posibles cuellos de botella.
4. **Diagnóstico:** típicamente, el problema es un *full table scan* (lectura completa de la tabla) o un JOIN no optimizado.
5. **Solución:** se crean índices (CREATE INDEX) o se reestructura la consulta para que aproveche mejor los existentes.
6. **Verificación:** se vuelve a ejecutar la consulta; si sigue lenta, se repite el proceso.

Este ciclo se repite hasta alcanzar el equilibrio óptimo entre velocidad y consumo de recursos. Optimizar no siempre es cuestión de añadir índices, sino de **comprender cómo el motor piensa**.



Caso de estudio

Facebook y la optimización de sus búsquedas

Contexto

Facebook (hoy Meta) es una de las plataformas con mayor volumen de datos del planeta. Cada segundo se realizan millones de búsquedas de personas, páginas o publicaciones. Si cada búsqueda tuviera que recorrer todas las filas de sus tablas de usuarios —que superan los 3.000 millones—, el sistema simplemente colapsaría. Por eso, la optimización de consultas es vital.

Estrategia

El equipo de ingeniería de Facebook utiliza un enfoque obsesivo con el rendimiento. Cada consulta que afecta a los sistemas centrales (usuarios, amigos, publicaciones) se analiza con herramientas internas basadas en el comando **EXPLAIN**. Un ejemplo simple de consulta sería:

```
SELECT id, nombre, ciudad FROM usuarios WHERE nombre = 'Juan Pérez';
```

Sin un índice en la columna nombre, esta consulta obligaría al sistema a revisar **cada fila de la tabla**. En cambio, con un índice bien diseñado:

```
CREATE INDEX idx_usuarios_nombre ON usuarios(nombre);
```

el motor puede saltar directamente a las filas relevantes.

Además, Facebook evita usar `SELECT *` (que devuelve todas las columnas) y especifica solo las necesarias, reduciendo el volumen de datos transferido. También monitoriza constantemente los **planes de ejecución** para asegurarse de que las consultas sigan usando los índices previstos, algo que puede cambiar con el tiempo si la distribución de los datos varía.

Resultado

Gracias a esta estrategia de **indexación proactiva y revisión constante**, Facebook consigue devolver resultados de búsqueda en milisegundos, incluso manejando bases de datos distribuidas a escala planetaria. La clave está en una combinación de hardware potente, pero sobre todo, **software optimizado**: las consultas SQL son tan rápidas como su diseño lo permita.

Herramientas y consejos

Crea índices estratégicos, no indiscriminados

Usa índices en columnas que se emplean con frecuencia en WHERE, JOIN, o como parte de una clave foránea. Un índice innútil ocupa espacio y ralentiza las operaciones de escritura. Ejemplo:

```
CREATE INDEX idx_pedidos_fecha ON pedidos(fecha);
```

Equilibrio entre lectura y escritura

Los índices aceleran los SELECT, pero cada INSERT, UPDATE o DELETE debe actualizar también los índices. Si una tabla tiene demasiados índices, su rendimiento de escritura se degrada. En sistemas de ecommerce, por ejemplo, se priorizan índices sobre columnas de búsqueda y no sobre cada campo.

Analiza el plan de ejecución con EXPLAIN

Antes de "arreglar" una consulta, entiende cómo se está ejecutando realmente. En PostgreSQL:

```
EXPLAIN ANALYZE SELECT * FROM pedidos WHERE ciudad =  
'Madrid';
```

El resultado te mostrará si se está usando un **Index Scan (eficiente)** o un **Seq Scan (recorrido completo)**.

Evita el SELECT *

Seleccionar todas las columnas consume tiempo y memoria. En entornos profesionales, se recomienda especificar solo las columnas necesarias. Ejemplo:

```
SELECT nombre, email FROM usuarios WHERE activo = TRUE;
```

Optimiza tus JOINs

Asegúrate de que las columnas de unión estén indexadas en ambas tablas. Si haces un JOIN entre clientes y pedidos, indexa clientes.id_cliente y pedidos.id_cliente. Así el motor no tendrá que comparar fila por fila.

Limita los resultados cuando sea posible

Si solo necesitas los 10 primeros resultados, usa LIMIT 10. Cuantas menos filas devuelva la consulta, menos tiempo y memoria requerirá.

Herramientas recomendadas para análisis y tuning

- **MySQL Workbench** y **pgAdmin** → muestran gráficamente el plan de ejecución.
- **Explain.depesz.com** → interpreta el resultado del EXPLAIN de PostgreSQL de manera visual.
- **SolarWinds Database Performance Analyzer** → monitorea el rendimiento de las consultas en producción.

Reescribe antes de escalar hardware

Un error común en las empresas es comprar servidores más potentes para resolver lentitud. Antes de eso, revisa tus consultas: optimizar el SQL suele ser **10 veces más efectivo y mucho más barato** que ampliar infraestructura.

The screenshot shows the browser's developer tools with the element inspector open. It displays the DOM structure of a 'todo-application' component. The DOM tree includes a root element with an id of 'app', which contains sections for '#shadow-root (open)', '', and '#todo-application'. Inside '#todo-application', there are elements like '', '', and ''. A right-hand sidebar provides detailed CSS information for the selected element, including filters, element styles, and inheritance from parent elements like 'ul.list' and 'ul, user agent style'.

Mitos y realidades

X MITO: "Optimizar SQL es tarea exclusiva de los administradores de bases de datos (DBA)."

→ **FALSO.** Todo profesional que escriba consultas debe entender los fundamentos de optimización. No se trata de una tarea exclusiva del DBA, sino de una competencia básica de cualquier programador backend o analista.

✓ Realidad: Un buen desarrollador escribe código limpio y también SQL eficiente.

X MITO: "Si mi consulta es lenta, compro más hardware."

→ **FALSO.** Añadir CPU o memoria puede mitigar temporalmente un problema, pero no lo soluciona. Una consulta ineficiente puede saturar cualquier servidor por potente que sea.

✓ Realidad: El verdadero rendimiento proviene de optimizar el diseño, los índices y las sentencias SQL, no de depender del hardware.

❑ Resumen final

- La **optimización de consultas** busca que el SQL se ejecute más rápido y consuma menos recursos.
- Un **índice** acelera las búsquedas evitando los *full table scans*.
- El comando **EXPLAIN** permite analizar cómo el motor ejecutará la consulta.
- **Buenas prácticas:**
 - Indexa columnas de WHERE y JOIN.
 - Evita SELECT *.
 - Lee e interpreta los planes de ejecución.
 - Reescribe consultas lentas antes de añadir hardware.
- Los índices aceleran las lecturas (SELECT), pero pueden ralentizar las escrituras (INSERT, UPDATE, DELETE).
- Optimizar SQL no es un lujo técnico: es una **responsabilidad profesional**.