

SQL

SELECT * FROM
learners;

WHERE level
= ADVANCED;

WHERE level
= ADVANCED;

SELECT * FROM
learners;

SELECT * FROM
learners;



Curso Completo de SQL

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México

lucasegaacademy.com

ÍNDICE

PRÓLOGO.....	4
UNIDAD 1: PRIMEROS PASOS CON LAS BASES DE DATOS.....	11
UNIDAD 2: DISEÑO DE TABLAS.....	43
UNIDAD 3: CONSULTAS BÁSICAS: SELECT Y WHERE.....	74
UNIDAD 4: FILTRADO AVANZADO Y ORDENAMIENTO.....	107
UNIDAD 5: BÚSQUEDA DE PATRONES Y RESÚMENES.....	138
UNIDAD 6: MODIFICACIÓN DE DATOS Y RELACIONES.....	173
UNIDAD 7: COMBINAR TABLAS: LOS JOINS.....	211
UNIDAD 8: CONSULTAS AVANZADAS.....	240
UNIDAD 9: PROYECTO: TIENDA EN LÍNEA COMPLETA.....	274
UNIDAD 10: GRADUACIÓN: CONSOLIDACIÓN Y SIGUIENTE PASO.....	307
EXAMEN FINAL.....	345

PRÓLOGO

Bienvenido a "Curso Completo de SQL", el libro de Lucas EGA Academy para aprender desde cero el idioma con el que se guardan, se consultan y se administran los datos del mundo entero. Este libro nació del curso en video de 30 clases de la academia, pero aquí tienes todo explicado con mucho más detalle: conceptos a fondo, analogías de la vida real, errores comunes de principiantes, buenas prácticas de la industria, ejemplos paso a paso para probar en tu computadora, prácticas con soluciones explicadas y un examen final para medir lo que aprendiste.

Cada vez que abres una aplicación en tu celular, que compras en línea o que el banco muestra tu saldo, hay una base de datos respondiendo preguntas detras de todo eso. El idioma para hacerle esas preguntas se llama SQL. Aprender SQL es aprender a hablar con los datos, y eso hoy es un superpoder.

No importa si nunca has programado. Las instrucciones de SQL se parecen al ingles cotidiano: cuando escribes "SELECT nombre FROM clientes", literalmente estas pidiendo el nombre de la tabla clientes. En pocas clases harás preguntas reales a bases de datos reales, y esa sensación es lo que hace que SQL enganche tan rápido.

A quien va dirigido este libro

Este libro es para ti si empiezas desde cero en tecnología y quieres una habilidad útil, concreta y con salida laboral rápida. Es para ti si ya programas en otro lenguaje, como Python o JavaScript, y te falta la pieza de los datos para completar tu perfil. Es para ti si vives en hojas de calculo y sientes que ya les sacaste todo el jugo: SQL es el siguiente paso natural y multiplica lo que puedes hacer con la información.

También es para estudiantes, para quien quiere cambiar de carrera, para emprendedores que quieren entender los datos de su negocio y para curiosos. No necesitas matemáticas avanzadas ni experiencia programando: solo curiosidad, una computadora y ganas de practicar.

Si vienes de los cursos de HTML, CSS o JavaScript de Lucas EGA Academy, este libro encaja perfecto: ya sabes construir páginas y aplicaciones, y ahora aprenderas donde viven los datos que esas aplicaciones muestran.

Por qué SQL es el idioma universal de los datos

SQL significa Structured Query Language, Lenguaje Estructurado de Consultas. Nació en los años setenta en IBM y después de más de cincuenta años sigue siendo el estandar mundial para trabajar con bases de datos relacionales. En tecnología, donde todo cambia cada dos años, un lenguaje con medio siglo vigente no es casualidad: es porque funciona.

Los datos viven en tablas: filas y columnas organizadas con reglas estrictas, como hojas de calculo superpoderosas. SQL sirve para crear esas tablas, llenarlas de información, buscar en ellas, combinarlas, actualizarlas y protegerlas. Casi todo sistema serio del planeta usa bases de datos relacionales: bancos, hospitales, aerolineas, tiendas en línea, gobiernos y redes sociales. Todos hablan SQL.

Además es portable: lo que aprendas con MySQL te sirve para PostgreSQL, SQL Server, SQLite y Oracle. Aprendes una vez y lo usas en todas partes: es la moneda comun de todo el que trabaja con información.

La habilidad que más piden las empresas

Hablemos de frente: SQL aparece en más ofertas de empleo que casi cualquier otra habilidad técnica. Desarrolladores, analistas de datos, científicos de datos, administradores de sistemas, gente de marketing, de finanzas y de logistica: todos terminan necesitando SQL tarde o temprano. En muchas vacantes es requisito obligatorio, y en las demás es lo que separa a los candidatos.

La razón es simple: las empresas viven de sus datos, pero los datos crudos no sirven si nadie sabe preguntarles cosas. Quien sabe SQL responde en minutos preguntas que a otros les toman días, como las ventas por sucursal o los clientes que dejaron de comprar. Convertir datos en respuestas vale dinero, y las empresas lo saben.

Lo mejor es que SQL tiene una de las curvas de aprendizaje más amables de la tecnología. Con semanas de práctica constante llegas a un nivel útil para un empleo real, y este libro esta diseñado justo para eso: llevarte de cero a un nivel sólido, con proyectos que puedes mostrar, en 30 clases bien ordenadas.

Que vas a aprender: recorrido por las 10 unidades

El libro está organizado en 10 unidades, cada una agrupa 3 de las 30 clases originales del curso en video. Cada unidad sigue la misma estructura:

- Introducción: que verás y por qué importa.
- Explicación detallada: conceptos a fondo, analogías, errores comunes y buenas prácticas.
- Ejemplos paso a paso: código SQL completo para copiar y probar, explicado línea por línea.
- Práctica: ejercicios con soluciones explicadas paso a paso. Los datos se aprenden practicando, no solo leyendo.
- Resumen: los puntos clave para repasar rápido.

Al final hay un EXAMEN FINAL con 50 preguntas: opción múltiple, verdadero o falso y preguntas abiertas donde escribiras tus propias consultas. Cada respuesta viene explicada en las soluciones. Si lo resuelves con confianza, estas listo para trabajar con bases de datos reales.

Este es tu mapa:

Unidad 1: Tus primeros pasos con SQL (Clases 1 a 3)

Desde cero absoluto. Clase 1: qué es SQL, que es una base de datos y por qué el mundo depende de ellas. Clase 2: instalación paso a paso de MySQL Community Server en tu computadora. Clase 3: tu primera base de datos y tu primera consulta real. Terminas con un servidor de bases de datos funcionando en tu máquina.

Unidad 2: Cómo se guardan los datos (Clases 4 a 6)

La base de todo: las tablas. Clase 4: crear tablas con CREATE TABLE. Clase 5: tipos de datos, textos, números y fechas, y como elegir el tipo correcto para cada columna. Clase 6: llenar tus tablas con INSERT. Aprenderas a diseñar y poblar las estructuras donde viven los datos.

Unidad 3: Tu primera forma de preguntar (Clases 7 a 9)

La estrella del curso: SELECT. Clase 7: tu primer SELECT para ver la información de una tabla. Clase 8: pedir solo las columnas que te interesan, cómo se hace en la vida real. Clase 9: WHERE, el filtro para pedir exactamente las filas que cumplen una condición. Ya podrás responder preguntas utiles con datos reales.

Unidad 4: Consultas con precisión (Clases 10 a 12)

Filo para tus consultas. Clase 10: operadores, rangos con BETWEEN, listas con IN y combinaciones con AND, OR y NOT. Clase 11: ORDER BY para ordenar resultados como quieras. Clase 12: LIMIT para pedir solo los primeros resultados, esencial con tablas enormes. Tus consultas serán precisas y profesionales.

Unidad 5: Buscar y resumir información (Clases 13 a 15)

Clase 13: LIKE para buscar textos por patrón, como nombres que empiezan con cierta letra. Clase 14: funciones agregadas, COUNT, SUM, AVG, MIN y MAX, que convierten miles de filas en un número útil. Clase 15: GROUP BY para totales por categoría, como ventas por mes o clientes por ciudad. La unidad favorita de los analistas de datos.

Unidad 6: Filtrar grupos y modificar datos (Clases 16 a 18)

Clase 16: HAVING, el filtro que se aplica después de agrupar. Clase 17: UPDATE para corregir y actualizar información, con la regla de oro de usar siempre WHERE. Clase 18: DELETE para borrar filas, entendiendo la diferencia entre vaciar una tabla y eliminarla. Sabrás mantener los datos vivos y correctos.

Unidad 7: El poder de relacionar tablas (Clases 19 a 21)

Aquí cambia tu forma de pensar. Clase 19: llaves primarias, el identificador único de cada fila. Clase 20: llaves foráneas, que conectan tablas entre sí. Clase 21: INNER JOIN, la operación más importante de SQL, para combinar información de varias tablas: clientes con sus pedidos, alumnos con sus calificaciones.

Unidad 8: Consultas de nivel profesional (Clases 22 a 24)

Clase 22: LEFT JOIN, para traer todos los registros de una tabla aunque no tengan pareja en la otra. Clase 23: DISTINCT para eliminar duplicados. Clase 24: alias, apodos para tablas y columnas que hacen tus consultas más cortas y legibles. Escribirás SQL como los profesionales.

Unidad 9: Técnicas avanzadas (Clases 25 a 27)

Clase 25: subconsultas, consultas dentro de consultas para preguntas en varios pasos. Clase 26: CREATE INDEX y como las bases de datos buscan información a toda velocidad. Clase 27: vistas, consultas guardadas que funcionan como tablas virtuales. Tu nivel ya será avanzado de verdad.

Unidad 10: Proyecto final y graduación (Clases 28 a 30)

Todo se junta aquí. Clase 28: construiras una tienda completa, diseno de tablas, relaciones y datos de ejemplo. Clase 29: reportes profesionales sobre esa tienda, como los que piden en las empresas. Clase 30: graduación y el siguiente paso, hacia donde seguir creciendo y cómo usar esto para conseguir trabajo. Terminas con un proyecto real en tu portafolio.

Como estudiar con este libro

No leas este libro como una novela. SQL se aprende con las manos en el teclado, haciendo consultas reales. Mi recomendación:

1. Ritmo sugerido: una clase por día, y terminas en un mes. Si tienes más tiempo, haz dos, pero no corras: entender bien cada unidad vale más que terminar rápido y a medias.
2. Escribe tú mismo cada ejemplo en tu computadora, no copies y pegues: escribir a mano hace que tu cerebro lo asimile de verdad. Luego rompe la consulta a proposito: quita el WHERE, cambia el orden, prueba otro JOIN. Ahí ocurre el verdadero aprendizaje.
3. Resuelve los ejercicios ANTES de mirar las soluciones. Si te atorras más de 30 minutos, mira la solución, entiendela, cierrala y resuelve de nuevo sin mirar. Así se fija el conocimiento.
4. Al terminar cada unidad lee el resumen y preguntate: "puedo explicar esto con mis palabras?". Si la respuesta es sí, avanza. Si es no, repasa sin pena: repasar no es retroceder, es construir cimientos.
5. El examen final es tu prueba de fuego. Hazlo completo sin ver las soluciones, como si fuera una entrevista de trabajo. Si lo pasas con confianza, tu nivel es real y puedes demostrarlo.

Un consejo honesto: habrá consultas que no te salgan a la primera, sobre todo los JOIN y las subconsultas. Le pasa a todo el mundo, incluso a programadores con experiencia. La diferencia entre quien aprende SQL y quien no, no es el talento: es seguir intentando un día más.

Requisitos: lo que necesitas para empezar

Necesitas muy poco, y casi todo es gratis:

- Una computadora con Windows, Mac o Linux. No tiene que ser potente: cualquier máquina reciente sirve para aprender SQL.
- MySQL Community Server, la versión gratuita del manejador de bases de datos más usado del mundo. En la Clase 2 tienes la instalación guiada paso a paso.
- Un cliente como MySQL Workbench, también gratis, que instalaremos juntos en la Clase 2.
- Ganas de practicar todos los días, aunque sean 30 minutos.

No necesitas saber programar, ni matemáticas avanzadas, ni pagar nada. La única inversión que te pido es tu tiempo y tu constancia.

Sobre Lucas EGA Academy

Este libro es parte del ecosistema de Lucas EGA Academy (lucasegaacademy.com), donde encontraras cursos gratuitos de tecnología, los videos de cada clase de este curso y más libros como este. Todo el contenido de la academia es gratis porque creemos que la educación en tecnología no debería tener barreras de dinero. Si este libro te sirve, compártelo: la mejor forma de agradecer es ayudar a que otra persona también aprenda.

Un último mensaje antes de empezar

Te hablo de frente: mucha gente cree que la tecnología es solo para genios o para quienes empezaron a programar de niños. Eso es mentira. La tecnología es para quien decide aprenderla, punto. Y SQL es una de las mejores puertas de entrada: útil desde la primera semana, pedida por las empresas y con una comunidad enorme dispuesta a ayudar.

Va a haber días en que una consulta no te salga y sientas que esto no es para ti. En esos días acuérdate de esto: cada experto en bases de datos empezó exactamente donde estas tu ahora, frente a una pantalla sin entender nada. La única diferencia es que ellos no se rindieron. Tu tampoco lo hagas.

En unas semanas miraras atrás y te sorprendera lo lejos que llegaste: de no saber que era una base de datos a diseñar una tienda completa con reportes profesionales. Ese camino empieza en la página siguiente, con la Unidad 1. Nos vemos ahí.

A darle con todo.

Esteban *Lucas EGA Academy - lucasegaacademy.com*

UNIDAD 1: PRIMEROS PASOS CON LAS BASES DE DATOS

Las bases de datos están en todas partes, aunque nunca las hayas visto. Cada vez que abres WhatsApp y lees un mensaje, cada vez que compras algo en Amazon, cada vez que tu banco te muestra tu saldo, cada vez que ves un video en YouTube, detrás de todo eso hay una base de datos trabajando en silencio para ti.

Piensa en esto: hoy mismo, sin darte cuenta, probablemente interactuaste con decenas de bases de datos. Pediste un Uber, y el sistema consultó una base de datos para encontrar al conductor más cercano. Abriste Instagram, y otra base de datos buscó tus fotos y tus mensajes en menos de un segundo. Checaste tu correo, y ahí estaba una base de datos organizando miles de mensajes. Es invisible, pero es el motor de todo el mundo digital.

Y aquí viene lo mejor: el lenguaje que usan casi todas esas bases de datos para comunicarse se llama SQL. Tres letras que valen oro en el mercado laboral. Si aprendes SQL, aprendes el idioma universal de los datos. Es el idioma que hablan los bancos, las tiendas en línea, las redes sociales, los hospitales, los gobiernos y las empresas de todo el planeta.

Esta unidad es tu puerta de entrada. Vamos a empezar desde cero absoluto, sin suponer que sabes nada de programación. Primero entenderás qué es una base de datos de verdad (más allá de la idea vaga de "algo donde se guardan datos"). Después conocerás a SQL: qué es, de dónde viene y por qué es tan importante. Luego instalarás MySQL, el sistema de bases de datos gratuito más usado del mundo, junto con MySQL Workbench, su herramienta visual. Y al final de la unidad crearás tu primera base de datos con tus propias manos.

No necesitas experiencia previa. Solo necesitas ganas de aprender y una computadora. Si puedes abrir un programa y escribir texto, puedes hacer todo lo que veremos aquí.

Al terminar esta unidad vas a saber: qué es una base de datos y cómo se organiza por dentro, qué es SQL y por qué es el estándar mundial, cómo instalar y configurar MySQL y MySQL Workbench en Windows, Mac o Linux, y cómo crear, ver, usar y eliminar bases de datos con tus primeros comandos SQL.

Prepárate, porque estás a punto de aprender una de las habilidades más valiosas y mejor pagadas en tecnología. Vamos paso a paso, sin prisa y sin miedo. Esteban, tu profesor, te acompaña en cada línea.

CÓMO SACARLE EL MÁXIMO PROVECHO A ESTE LIBRO

Antes de empezar, tres consejos que harán la diferencia entre leer sobre SQL y aprender SQL de verdad.

Primero: no solo leas el código, escríbelo. Cada ejemplo de este libro está diseñado para que lo copies en MySQL Workbench y lo ejecutes tú mismo. Leer un comando te da la ilusión de entenderlo; ejecutarlo y ver el resultado te da el entendimiento real. Ten Workbench abierto mientras lees.

Segundo: equivócate a propósito. Cuando un ejemplo funcione, prueba variarlo: cambia un nombre, quita el punto y coma, escribe una palabra clave en minúsculas. Ver qué se rompe y por qué es una de las formas más rápidas de aprender. Tu laboratorio local es a prueba de errores: no puedes romper nada importante.

Tercero: repite en voz alta lo que hace cada comando. Parece un truco tonto, pero funciona: decir "CREATE DATABASE crea una base de datos llamada tienda" fija el concepto en tu memoria mucho mejor que solo leerlo. Los mejores estudiantes que he tenido hablan solos frente a la computadora. No les dio pena, y aprendieron rapidísimo.

Este libro está escrito para que avances unidad por unidad, sin saltarte nada. Cada unidad se apoya en la anterior, como los pisos de un edificio. La Unidad 1 que estás por leer es la cimentación: si queda sólida, todo lo demás se construye fácil.

Explicación detallada

Empecemos con la pregunta más básica de todas: qué es una base de datos.

Una base de datos es un conjunto organizado de información que se guarda de forma que sea fácil buscarla, modificarla y usarla. Eso es todo. No es magia, no es algo misterioso. Es información organizada con un propósito.

Para entenderlo, piensa en una biblioteca. Una biblioteca no es un montón de libros tirados en el piso. Los libros están organizados por secciones, por autor, por tema. Hay un catálogo que te dice exactamente dónde está cada libro. Gracias a esa organización, puedes encontrar un libro específico entre millones en cuestión de minutos.

Una base de datos funciona igual. Es la "biblioteca" donde viven los datos, y tiene su propio "catálogo" que permite encontrar cualquier dato en milisegundos, incluso entre millones de registros.

Otro ejemplo: el archivero de una oficina. Imagina una oficina de gobierno con archiveros metálicos llenos de carpetas. Cada cajón tiene una etiqueta, cada carpeta tiene un nombre,

cada hoja dentro de la carpeta tiene un formato. Si alguien necesita el expediente de un ciudadano, no revisa hoja por hoja: va al cajón correcto, saca la carpeta correcta y listo. Una base de datos es el archivero digital llevado a la perfección: nunca se pierde nada, todo tiene su lugar y la búsqueda es instantánea.

Ahora, ¿por qué no guardar todo en archivos de texto o en hojas de cálculo? Buena pregunta, y es importante entenderla porque marca la diferencia entre un aficionado y un profesional.

Puedes guardar una lista de 50 clientes en Excel sin problema. Pero ¿qué pasa cuando son 50 millones de clientes? ¿Qué pasa cuando 1,000 empleados necesitan consultar y modificar esa información al mismo tiempo? ¿Qué pasa si dos personas editan el mismo archivo a la vez y se corrompe? ¿Qué pasa si se va la luz a mitad de una operación importante?

Las hojas de cálculo y los archivos de texto no están diseñados para eso. Las bases de datos sí. Una base de datos profesional puede manejar millones de datos, permitir que miles de usuarios trabajen al mismo tiempo sin que nada se rompa, proteger la información con contraseñas y permisos, y recuperarse de fallos sin perder datos.

Además, las bases de datos garantizan algo que los archivos sueltos no pueden: la integridad de los datos. Esto significa que los datos siempre tienen sentido y están completos. Por ejemplo, en una base de datos bien diseñada no puede existir un pedido sin un cliente que lo haya hecho, ni un producto con precio negativo. La base de datos misma vigila que las reglas se cumplan.

Ahora que entiendes qué es una base de datos, vamos a conocer sus piezas internas. Porque una base de datos no es una caja negra: tiene una estructura clara que vas a dominar.

La pieza fundamental es la TABLA. Una tabla es como una hoja de cálculo dentro de la base de datos: organiza la información en filas y columnas. Cada tabla guarda un tipo de cosa: una tabla de clientes, una tabla de productos, una tabla de pedidos.

Las COLUMNAS de una tabla definen qué tipo de información se guarda. Por ejemplo, la tabla "clientes" puede tener las columnas: id, nombre, apellido, correo y teléfono. Cada columna tiene un nombre y un tipo de dato (texto, número, fecha, etc.). Las columnas son las categorías, los "campos" del formulario.

Las FILAS son los datos reales, cada una representa un elemento individual. Si la tabla es "clientes", cada fila es un cliente distinto: la fila 1 es Ana García, la fila 2 es Luis Pérez, y así sucesivamente. Cada fila también se llama REGISTRO. Fila y registro son lo mismo: un renglón completo de datos.

Fíjate en la lógica: las columnas dicen QUÉ se guarda (nombre, correo, teléfono) y las filas dicen QUIÉNES o CUÁLES son (Ana, Luis, María). Es una estructura simple pero poderosísima, y es la base de todo lo que aprenderás en este curso.

Una base de datos puede tener una tabla o cientos de tablas. Lo importante es que las tablas se relacionan entre sí. Por ejemplo, la tabla "pedidos" se relaciona con la tabla "clientes" porque cada pedido pertenece a un cliente. Esa capacidad de relacionar tablas es lo que hace tan potentes a las bases de datos relacionales, que son el tipo más usado en el mundo y el que aprenderás en este curso.

Hablemos ahora de SQL, el protagonista de este libro.

SQL significa Structured Query Language, que en español es Lenguaje de Consulta Estructurado. Es el lenguaje estándar para comunicarse con las bases de datos relacionales. Con SQL puedes hacer cuatro cosas fundamentales: guardar datos, leer datos, modificar datos y borrar datos. Todo lo demás son variaciones y combinaciones de esas cuatro operaciones.

La historia de SQL es interesante y vale la pena conocerla. Nació en los años 70 en los laboratorios de IBM. Dos investigadores, Donald Chamberlin y Raymond Boyce, crearon un lenguaje llamado SEQUEL (Structured English Query Language) para trabajar con un modelo teórico de bases de datos relacionales propuesto por Edgar Codd, otro genio de IBM. Con el tiempo, SEQUEL se convirtió en SQL.

En 1986, SQL fue adoptado como estándar oficial, y desde entonces es EL lenguaje de las bases de datos relacionales. Han pasado casi 40 años y sigue siendo el estándar. En tecnología, donde todo cambia cada seis meses, que un lenguaje siga vigente después de cuatro décadas dice muchísimo de su importancia.

¿Por qué SQL triunfó? Por tres razones. Primera: es declarativo, es decir, le dices a la base de datos QUÉ quieres, no CÓMO conseguirlo. Tú dices "dame todos los clientes de México" y la base de datos se encarga de buscarlos de la forma más eficiente. Segunda: es legible, casi parece inglés: SELECT, FROM, WHERE son palabras que cualquiera entiende. Tercera: es universal, funciona (con pequeñas variaciones) en MySQL, PostgreSQL, SQL Server, Oracle y SQLite.

Esa universalidad es clave para tu carrera. Aprendes SQL una vez y lo usas en todas partes. Si mañana cambias de trabajo y la empresa usa otro sistema de base de datos, tu conocimiento de SQL sigue valiendo. Es una inversión que no se devalúa.

Ahora aclaremos los términos que suelen confundir a los principiantes, porque es muy común mezclarlos.

Una cosa es el LENGUAJE (SQL) y otra cosa es el SISTEMA que lo ejecuta. MySQL, PostgreSQL, SQL Server y Oracle son sistemas gestores de bases de datos, conocidos como DBMS (Database Management System) o también como RDBMS (la R es de relacional). Piensa en esto: SQL es el idioma español, y MySQL es una persona que habla español. El idioma es uno, pero hay muchas personas (sistemas) que lo hablan, cada una con su acento y sus modismos.

En este curso usaremos MySQL, y hay buenas razones para ello. MySQL es gratuito y de código abierto. Es el sistema de bases de datos más usado del mundo para aplicaciones web: lo usan Facebook, Twitter, YouTube, Wikipedia y millones de sitios más. Tiene una comunidad enorme, así que cualquier duda que tengas ya la resolvió alguien en internet. Y lo que aprendas en MySQL te sirve para cualquier otro sistema, porque todos hablan SQL.

Además de MySQL, existen otros tipos de bases de datos que conviene conocer de nombre, aunque no las estudiaremos a fondo en este curso.

Las bases de datos RELACIONALES (como MySQL, PostgreSQL, SQL Server, Oracle) organizan los datos en tablas relacionadas entre sí y usan SQL. Son ideales cuando los datos tienen una estructura clara y las relaciones importan: bancos, tiendas, escuelas, hospitales.

Las bases de datos NO relacionales, también llamadas NoSQL (como MongoDB, Redis, Cassandra), guardan los datos de otras formas: documentos, pares clave-valor, grafos. Son útiles para casos especiales, como datos sin estructura fija o sistemas que necesitan crecer de forma masiva y rápida. Pero ojo: "NoSQL" no significa "mejor", significa "diferente". Para la inmensa mayoría de las aplicaciones del mundo, las relacionales siguen siendo la mejor opción.

Mi recomendación como profesor: aprende primero SQL y las bases de datos relacionales. Es la base (nunca mejor dicho) de todo. Después, si tu camino lo requiere, exploras las NoSQL. Pero el 90% de los empleos que piden bases de datos, piden SQL.

Antes de instalar nada, hablemos de los errores más comunes que cometen los principiantes. Conocerlos de antemano te ahorrará horas de frustración.

Error 1: confundir la base de datos con la tabla. La base de datos es el contenedor (la biblioteca), la tabla es una parte de su contenido (un estante de libros). Cuando creas una base de datos, está vacía: es como comprar el archivero sin las carpetas. Las tablas se crean después, dentro de la base de datos.

Error 2: no seleccionar la base de datos antes de trabajar. Es como querer guardar un documento sin haber abierto ninguna carpeta: el sistema no sabe dónde ponerlo. Por eso existe el comando USE, que le dice a MySQL "voy a trabajar en esta base de datos". Lo verás en detalle más adelante.

Error 3: olvidar el punto y coma al final de los comandos. En SQL, el punto y coma (;) marca el fin de una instrucción. Si lo olvidas, MySQL se queda esperando, como si no hubieras terminado de hablar. No es grave, pero confunde mucho al principio.

Error 4: usar mayúsculas y minúsculas de forma inconsistente en los nombres. En algunos sistemas, "Tienda" y "tienda" son dos bases de datos distintas. Acostúmbrate desde hoy a usar minúsculas en los nombres de bases de datos y tablas: es la convención más segura.

Error 5: poner espacios o caracteres raros en los nombres. "mi tienda" con espacio te va a dar problemas. Usa guion bajo: "mi_tienda". Simple y sin dolores de cabeza.

Error 6: no hacer respaldos. Cuando trabajes con datos reales, respalda siempre antes de borrar o modificar en masa. En este curso practicaremos en bases de datos de prueba, así que no hay riesgo, pero que se te quede grabado como hábito profesional.

Y ahora, las buenas prácticas que distinguen a un profesional desde el primer día.

Buena práctica 1: escribe las palabras clave de SQL en MAYÚSCULAS (SELECT, FROM, WHERE, CREATE). No es obligatorio, MySQL las entiende en minúsculas, pero el código se lee mucho mejor y es la convención universal. En este libro siempre las verás en mayúsculas.

Buena práctica 2: termina cada instrucción con punto y coma. Siempre. Sin excepciones.

Buena práctica 3: usa nombres descriptivos y en minúsculas para tus bases de datos y tablas: "tienda", "escuela", "veterinaria". Que el nombre diga qué contiene.

Buena práctica 4: comenta tu código. Los comentarios son notas que le dejas a tu yo del futuro (o a otros programadores) explicando qué hace cada parte. En SQL hay dos formas, que practicarás en esta unidad.

Buena práctica 5: prueba primero en un entorno seguro. Todo lo que hagas en este curso será en tu computadora local, en bases de datos de práctica. Experimenta sin miedo: romper una base de datos de prueba es la mejor forma de aprender.

Con la teoría clara, es hora de ensuciarnos las manos. Vamos a instalar MySQL paso a paso.

Necesitas dos programas. El primero es MySQL Community Server: es el motor, el sistema gestor de bases de datos como tal. Es el que guarda los datos, ejecuta tus comandos SQL y hace todo el trabajo pesado. Funciona como un servicio en segundo plano en tu computadora.

El segundo es MySQL Workbench: es la herramienta visual para trabajar con MySQL. En lugar de escribir comandos en una pantalla negra intimidante, Workbench te da una interfaz gráfica amigable donde puedes escribir SQL, ver tus tablas, explorar tus bases de datos y administrar todo con comodidad. Es como la diferencia entre manejar un carro estándar y uno automático: el motor es el mismo, pero la experiencia es mucho más cómoda.

Ambos son gratuitos. Vamos con la instalación según tu sistema operativo.

INSTALACIÓN EN WINDOWS (paso a paso):

Paso 1: entra a la página oficial de descargas de MySQL. Busca "MySQL Community Server download" en Google y entra al sitio oficial de MySQL (mysql.com). Siempre descarga del sitio oficial, nunca de páginas de terceros.

Paso 2: descarga el instalador de MySQL para Windows. Verás una opción que dice "MySQL Installer". Es un programa que te guía en la instalación de todos los componentes. Elige la versión más reciente estable.

Paso 3: ejecuta el instalador. Te preguntará qué tipo de instalación quieres. Elige la opción "Developer Default" (opción predeterminada para desarrolladores): instala el servidor, Workbench y las herramientas más útiles. Es la opción recomendada para este curso.

Paso 4: el instalador descargará los componentes y los instalará uno por uno. Ten paciencia, puede tardar varios minutos dependiendo de tu conexión a internet.

Paso 5: llegará el momento de la configuración del servidor. Aquí te pedirá elegir el tipo de configuración: selecciona "Development Computer" (computadora de desarrollo). Esto ajusta MySQL para que use pocos recursos, ideal para aprender.

Paso 6: configuración de red. MySQL usa por defecto el puerto 3306. Déjalo así, no lo cambies a menos que sepas lo que haces. Este puerto es por donde tu computadora se comunica con MySQL.

Paso 7: el paso más importante: la cuenta root. El usuario "root" es el superadministrador de MySQL, el que puede hacer todo. El instalador te pedirá que definas una CONTRASEÑA para root. Elige una contraseña segura: mínimo 8 caracteres, combinando letras, números y algún símbolo. Y APÚNTALA en un lugar seguro. Si la olvidas, recuperar el acceso es un dolor de cabeza. En este curso la necesitarás cada vez que te conectes.

Paso 8: el instalador te preguntará si quieres crear usuarios adicionales. Para este curso no es necesario: con root es suficiente. Puedes saltar ese paso.

Paso 9: finaliza la instalación. El instalador configurará MySQL como un servicio de Windows que arranca automáticamente. Esto significa que MySQL estará corriendo en segundo plano cada vez que enciendas tu computadora, listo para usarse.

Paso 10: abre MySQL Workbench desde el menú inicio. Verás una pantalla con conexiones. Si todo salió bien, habrá una conexión local lista para usarse. Haz clic en ella, escribe tu contraseña de root, y entrarás al mundo de las bases de datos.

INSTALACIÓN EN MAC (paso a paso):

Paso 1: entra al sitio oficial de MySQL y descarga MySQL Community Server para macOS. Elige el archivo DMG correspondiente a tu versión de macOS y a tu chip (hay versiones para Intel y para Apple Silicon M1/M2/M3: descarga la correcta).

Paso 2: abre el archivo DMG y ejecuta el instalador PKG. Sigue el asistente: es un proceso guiado muy sencillo.

Paso 3: durante la instalación te pedirá definir la contraseña del usuario root. Igual que en Windows: contraseña segura, mínimo 8 caracteres, y apúntala en un lugar seguro.

Paso 4: al terminar, MySQL quedará instalado como un servicio. Puedes verificarlo en Preferencias del Sistema, donde aparecerá un panel de MySQL para iniciar o detener el servidor.

Paso 5: ahora descarga MySQL Workbench para macOS desde el sitio oficial (también en archivo DMG). Instálalo arrastrando el icono a la carpeta Aplicaciones, como cualquier programa de Mac.

Paso 6: abre MySQL Workbench. Crea una nueva conexión: ponle un nombre como "Local", en hostname escribe "127.0.0.1" (esa es tu propia computadora), el puerto 3306, y el usuario "root". Guarda la conexión.

Paso 7: haz doble clic en la conexión, escribe tu contraseña de root cuando te la pida, y listo: estás dentro.

INSTALACIÓN EN LINUX (paso a paso, ejemplo con Ubuntu/Debian):

Paso 1: abre una terminal. Actualiza la lista de paquetes con el comando de tu distribución. En Ubuntu o Debian, el sistema trae MySQL en sus repositorios oficiales.

Paso 2: instala el servidor de MySQL desde la terminal con el gestor de paquetes de tu distribución. El paquete se llama mysql-server.

Paso 3: una vez instalado, el servicio de MySQL debería arrancar automáticamente. Puedes verificar que esté corriendo con el comando de estado de servicios de tu sistema.

Paso 4: ejecuta el script de configuración segura que incluye MySQL. Este asistente te guiará para definir la contraseña de root, eliminar usuarios anónimos, desactivar el acceso remoto de root y recargar los permisos. Responde "sí" a todas las opciones de seguridad que te proponga: es la configuración recomendada.

Paso 5: instala MySQL Workbench. En Ubuntu puedes instalarlo desde los repositorios o descargando el paquete DEB del sitio oficial de MySQL. También está disponible como paquete Snap en muchas distribuciones.

Paso 6: abre MySQL Workbench, crea la conexión local (hostname 127.0.0.1, puerto 3306, usuario root) y conéctate con tu contraseña.

Independientemente del sistema operativo, hay conceptos de la configuración que debes entender bien.

El usuario ROOT es el administrador total. Puede crear y borrar bases de datos, crear usuarios, cambiar configuraciones: todo. Por eso su contraseña debe ser segura. En un entorno de aprendizaje como el tuyo, usar root está bien. En un trabajo real, jamás usarías root para las aplicaciones del día a día: crearías usuarios con permisos limitados. Pero eso es tema de unidades avanzadas.

La CONEXIÓN LOCAL significa que MySQL corre en tu misma computadora. El "hostname" 127.0.0.1 es una dirección especial que siempre apunta a tu propia máquina: se le llama "localhost". Cuando en el futuro trabajes con bases de datos en servidores remotos, cambiarás ese hostname por la dirección del servidor, pero todo lo demás será igual. Lo que aprendas en local te sirve para el mundo real.

El PUERTO 3306 es el canal de comunicación de MySQL. Piensa en los puertos como puertas numeradas: cada programa usa la suya para que no se mezclen las comunicaciones. Si algún día MySQL no se conecta, una causa común es que otro programa esté usando el puerto 3306 o que un firewall lo esté bloqueando.

¿QUÉ HACER SI ALGO FALLA? La instalación a veces presenta problemas, y es normal. Aquí están las soluciones a los fallos más comunes.

Problema 1: "No puedo conectarme al servidor" o "Can't connect to MySQL server". Causa más común: el servicio de MySQL no está corriendo. Solución: verifica que el servicio esté iniciado. En Windows, abre el administrador de servicios (services.msc), busca "MySQL" y asegúrate de que esté en ejecución; si no, inícialo. En Mac, revisa el panel de MySQL en Preferencias del Sistema. En Linux, revisa el estado del servicio mysql.

Problema 2: "Access denied for user root". Casi siempre significa que la contraseña está mal escrita. Solución: verifica que estés escribiendo la contraseña correcta, cuidando mayúsculas y minúsculas. Si definitivamente la olvidaste, tendrás que restablecerla: busca en la documentación oficial el procedimiento de "reset root password" para tu sistema operativo. Por eso insistimos tanto en apuntarla.

Problema 3: el puerto 3306 está ocupado. Algunos programas (como ciertas instalaciones de XAMPP, Skype en versiones viejas, u otro MySQL) pueden usar ese puerto. Solución: identifica qué programa lo usa y ciérralo, o configura MySQL en otro puerto durante la instalación (por ejemplo 3307), recordando usar ese puerto en Workbench.

Problema 4: en Windows, el instalador se queja de requisitos faltantes (como Visual C++ Redistributable). Solución: el propio instalador de MySQL suele ofrecer descargarlos; acepta y deja que los instale antes de continuar.

Problema 5: en Mac con chip Apple Silicon, descargaste la versión para Intel por error. Solución: desinstala y descarga la versión ARM64 del sitio oficial. Los instaladores están claramente etiquetados.

Problema 6: Workbench se conecta pero no ves tus bases de datos. Solución: asegúrate de haber iniciado sesión con un usuario que tenga permisos (root los tiene todos). También verifica que estés en la conexión correcta y no en una de prueba.

Un consejo general para los problemas: lee el mensaje de error completo antes de entrar en pánico. Los mensajes de MySQL suelen ser descriptivos y te dicen exactamente qué falló. Copia el mensaje en Google entre comillas y casi siempre encontrarás la solución en los primeros resultados. Saber buscar errores es una habilidad de programador tan importante como saber programar.

Una vez instalado todo y conectado en Workbench, verás una ventana con un editor de SQL en el centro. Ahí es donde escribirás tus comandos. A la izquierda verás el panel de "Schemas" (esquemas): ahí aparecerán tus bases de datos. En MySQL, "schema" y "database" son sinónimos: es lo mismo. No te confundas cuando veas esa palabra.

En la siguiente sección vas a escribir tus primeros comandos SQL reales. Pero antes de pasar a la práctica, asegúrate de tener MySQL instalado y Workbench abierto con una conexión local

funcionando. Tómate el tiempo que necesites: una buena instalación te ahorrará problemas durante todo el curso.

Recuerda la idea central de esta unidad: una base de datos es información organizada, una tabla organiza esa información en filas y columnas, SQL es el idioma universal para hablar con las bases de datos, y MySQL es el sistema gratuito donde practicarás todo. Con eso claro, ya estás listo para crear tu primera base de datos.

Antes de cerrar la teoría, profundicemos en algunos temas que te darán una ventaja enorme: entenderás no solo el "qué", sino el "por qué" de las decisiones importantes.

Hablemos del MODELO RELACIONAL con más detalle, porque es el corazón de todo lo que harás. La idea central es simple: en lugar de guardar toda la información en una tabla gigante, la divides en tablas pequeñas y especializadas que se conectan entre sí.

Imagina una tienda en línea. Un principiante guardaría todo en una sola tabla: cada fila tendría el nombre del cliente, su dirección, su teléfono, el producto comprado, el precio, la fecha. El problema: si Ana García compra 10 veces, su nombre, dirección y teléfono se repiten 10 veces. Si Ana cambia de teléfono, hay que actualizar 10 filas. Si se te olvida una, los datos quedan inconsistentes: el mismo cliente con dos teléfonos distintos.

La solución relacional: una tabla "clientes" con los datos de Ana una sola vez, y una tabla "pedidos" donde cada pedido apunta al cliente correspondiente. Los datos de Ana existen en un solo lugar. Si cambia su teléfono, lo actualizas una vez y listo. Esa es la magia del modelo relacional: cada dato vive en un solo lugar (principio de "fuente única de verdad") y las tablas se conectan mediante identificadores.

Esos identificadores que conectan tablas se llaman LLAVES. La llave primaria (PRIMARY KEY) es la columna que identifica de forma única cada fila de una tabla: por ejemplo, el "id" del cliente. Ningún par de clientes comparten el mismo id. La llave foránea (FOREIGN KEY) es una columna en una tabla que apunta a la llave primaria de otra: por ejemplo, la columna "cliente_id" en la tabla "pedidos" apunta al "id" de la tabla "clientes". Así se construyen las relaciones. No te preocupes si suena abstracto: en la Unidad 3 lo practicarás con ejemplos reales hasta que sea natural.

Otro concepto que conviene conocer desde hoy son los TIPOS DE DATOS. Cada columna de una tabla tiene un tipo de dato que define qué clase de información puede guardar. Los más comunes son: INT para números enteros (edades, cantidades), DECIMAL para números con decimales (precios), VARCHAR para textos de longitud variable (nombres, correos), TEXT para textos largos (descripciones), DATE para fechas y DATETIME para fecha y hora juntas. Elegir el tipo correcto importa: ocupa menos espacio, evita errores y hace las búsquedas más rápidas. En la Unidad 2 los usarás al crear tus primeras tablas.

Hablemos también de MySQL frente a sus parientes, porque es una duda muy común. Existe MariaDB, que es un "hermano" de MySQL: nació cuando parte de la comunidad quiso mantener el proyecto 100% abierto. MariaDB es casi idéntico a MySQL y todo lo que aprendas aquí te sirve para ambos. También existe la edición Enterprise de MySQL (de pagó, para empresas grandes) y la edición Community (gratuita, la que usas tú). Para aprender y para la mayoría de los proyectos, Community es más que suficiente: es el mismo motor.

¿Por qué MySQL y no otro sistema para este curso? Ya mencionamos que es el más usado en la web, pero hay más razones prácticas. Primera: la instalación es sencilla y hay instaladores oficiales para todos los sistemas. Segunda: MySQL Workbench es una herramienta visual excelente y gratuita. Tercera: la documentación y la comunidad son enormes; cualquier error que te salga, alguien ya lo resolvió y lo publicó. Cuarta: si después quieres aprender PostgreSQL o SQL Server, el 95% de tu conocimiento de SQL se transfiere directamente.

Dediquemos un momento a conocer mejor MySQL Workbench, porque será tu taller durante todo el curso. Al abrirlo verás varias zonas. Arriba está la barra de herramientas con los botones para ejecutar consultas (el famoso rayo). En el centro está el editor de SQL, donde escribes tus comandos: tiene colores que distinguen las palabras clave, los textos y los comentarios, lo que facilita leer el código. Abajo del editor aparece el panel de resultados, donde ves lo que devuelve cada consulta en forma de tabla. A la izquierda está el navegador de esquemas: ahí verás tus bases de datos en forma de árbol; puedes expandir cada una para ver sus tablas.

Un truco muy útil de Workbench: no necesitas escribir todo a mano. Si haces clic derecho sobre una base de datos en el navegador, verás opciones como "Create Schema" o "Drop Schema". Pero te recomiendo escribir los comandos a mano durante este curso: la memoria muscular que desarrolles escribiendo SQL vale oro. La interfaz gráfica es cómoda, pero el profesional escribe código.

Hablemos de SEGURIDAD básica, porque aunque estés aprendiendo, los buenos hábitos se forman desde el primer día. La contraseña de root es la llave maestra de tu MySQL: quien la

tenga puede hacer todo. No la compartas, no la pegues en chats, no la subas a internet. Si algún día trabajas en equipo, cada persona debe tener su propio usuario con solo los permisos que necesita: ese principio se llama "mínimo privilegio". Además, cuando publiques proyectos en internet, jamás incluyas contraseñas en el código: existen formas seguras de manejarlas que aprenderás más adelante.

Otro hábito profesional: los RESPALDOS (backups). Un respaldo es una copia de seguridad de tu base de datos. Si algo sale mal (un DROP accidental, un disco dañado), el respaldo te salva. MySQL Workbench tiene una herramienta de exportación para crear respaldos con unos clics, y también existe el comando `mysqldump` en la terminal. En este curso trabajarás con datos de práctica, así que no hay riesgo real, pero cuando manejes datos importantes, respalda antes de cualquier operación peligrosa. Los profesionales no confían en su memoria ni en su suerte: confían en sus respaldos.

Volvamos a la instalación con algunos consejos extra que no suelen venir en los tutoriales. Si instalas MySQL en una computadora con poco espacio en disco o poca memoria, elige siempre las opciones de "Development Computer": están diseñadas para consumir pocos recursos. Si tu computadora es muy vieja y la instalación se siente lenta, cierra otros programas durante el proceso.

Si ya tenías instalado algún paquete como XAMPP, WAMP o MAMP (que traen su propio MySQL), puedes tener conflictos: dos servidores MySQL peleando por el mismo puerto 3306. Solución: usa solo uno a la vez. Detén el servicio del paquete que no uses mientras prácticas con tu instalación oficial, o cambia el puerto de una de las dos instalaciones. No es complicado, pero es una causa clásica del error "can't connect".

Para los usuarios de Linux avanzado: si prefieres no instalar Workbench con interfaz gráfica (por ejemplo, en un servidor sin pantalla), puedes trabajar directamente con el cliente de línea de comandos "mysql" en la terminal. Los comandos SQL son exactamente los mismos; solo cambia el entorno. De hecho, saber usar la terminal te hace ver muy profesional, pero para este curso Workbench es perfecto.

Una pregunta frecuente: ¿necesito internet para usar MySQL? No. Una vez instalado, MySQL corre en tu computadora y no necesita conexión. Puedes practicar en un avión, en el campo, donde sea. Solo necesitas internet para descargarlo la primera vez y para buscar ayuda cuando te atores.

Otra pregunta frecuente: ¿MySQL sirve para conseguir trabajo? Absolutamente. Las ofertas de empleo que piden SQL suelen mencionar MySQL o aceptar cualquier experiencia en bases de datos relacionales. Además, muchas empresas usan MySQL o MariaDB en producción, así que tu práctica local se traduce directamente en experiencia laboral. Y cuando domines lo

básico, el camino natural es aprender a conectar MySQL con un lenguaje de programación (PHP, Python, JavaScript), que es cómo se construyen las aplicaciones reales.

Para cerrar esta sección teórica, una reflexión importante. Aprender bases de datos no es memorizar comandos: es aprender a pensar en estructuras. Cuando veas una aplicación (una tienda, un banco, una red social), empieza a preguntarte: ¿qué tablas tendrá por dentro? ¿Cómo se relacionan? ¿Qué datos se repiten y cuáles son únicos? Ese ejercicio mental, hacerlo todos los días, te convertirá en un diseñador de bases de datos. Los comandos se aprenden en semanas; la forma de pensar se entrena toda la vida. Y tú ya empezaste hoy.

Ejemplos paso a paso

Es hora de escribir SQL de verdad. Abre MySQL Workbench, conéctate a tu servidor local y abre una ventana de consulta (el editor de SQL). Todo lo que sigue lo vas a escribir y ejecutar tú mismo. Para ejecutar un comando en Workbench, escríbelo en el editor y presiona el botón del rayo o las teclas Ctrl + Enter.

Ejemplo 1: ver las bases de datos que ya existen.

Antes de crear nada, vamos a ver qué bases de datos trae MySQL por defecto. Escribe este comando y ejecútalo:

```
SHOW DATABASES;
```

SHOW DATABASES le pide a MySQL: "muéstrame todas las bases de datos que existen". El resultado será una lista con nombres como "information_schema", "mysql", "performance_schema" y "sys". Esas son bases de datos del sistema: MySQL las usa para su funcionamiento interno. No las toques, no las borres, no les cambies nada. Son como los órganos vitales del sistema: no se operan sin necesidad.

Fíjate en dos detalles. Primero, las palabras clave están en MAYÚSCULAS: SHOW DATABASES. Segundo, el comando termina con punto y coma (;). Acostúmbrate a ese formato desde hoy.

Ejemplo 2: crear tu primera base de datos.

Ahora sí, el momento histórico: vas a crear tu primera base de datos. Se va a llamar "tienda", como una tienda en línea de ejemplo que usaremos durante el curso. Escribe y ejecuta:

```
CREATE DATABASE tienda;
```

CREATE DATABASE significa "crea una base de datos" y "tienda" es el nombre que le das. MySQL te responderá con un mensaje de confirmación, algo como "1 row affected" o simplemente "OK". Eso significa que la base de datos se creó correctamente.

Ahora vuelve a ejecutar SHOW DATABASES y verás que "tienda" aparece en la lista. Acabas de crear tu primera base de datos. Felicidades: ya hiciste algo que millones de personas no saben hacer.

Nota sobre el nombre: usamos "tienda" en minúsculas, sin espacios ni caracteres especiales. Esa es la convención que seguiremos en todo el curso.

Ejemplo 3: intentar crear una base de datos que ya existe (y entender el error).

¿Qué pasa si ejecutas CREATE DATABASE tienda; otra vez? Pruébalo:

```
CREATE DATABASE tienda;
```

MySQL te responderá con un error: "Can't create database 'tienda'; database exists". Traducción: "no puedo crear la base de datos 'tienda'; ya existe". Esto no es grave: es MySQL protegiéndote de crear duplicados por accidente. Los errores en SQL son tus amigos: te dicen exactamente qué salió mal.

Si quieres crear una base de datos solo cuando no exista (evitando el error), existe una variante:

```
CREATE DATABASE IF NOT EXISTS tienda;
```

IF NOT EXISTS significa "si no existe". Con esta versión, si la base de datos ya existe, MySQL no hace nada y no marca error: solo te avisa con una advertencia. Es muy útil en scripts que se ejecutan varias veces.

Ejemplo 4: seleccionar la base de datos con la que vas a trabajar (USE).

Crear la base de datos no significa que ya estés "dentro" de ella. Tienes que decirle a MySQL: "de ahora en adelante, voy a trabajar en la base de datos tienda". Eso se hace con USE:

```
USE tienda;
```

Después de ejecutar USE tienda;, todos los comandos que escribas se aplicarán a la base de datos "tienda". Es como entrar a una carpeta antes de guardar archivos en ella. Si olvidas el USE, MySQL no sabrá dónde crear tus tablas y te dará error.

¿Cómo sabes en qué base de datos estás? Workbench lo muestra en la interfaz, y además puedes ejecutar:

```
SELECT DATABASE();
```

Este comando te devuelve el nombre de la base de datos actualmente seleccionada. Si acabas de hacer `USE tienda;`, te devolverá "tienda". Si no has seleccionado ninguna, te devolverá NULL (qué significa "vacío" o "sin valor" en SQL).

Ejemplo 5: borrar una base de datos (con mucho cuidado).

Vamos a crear una base de datos de prueba solo para practicar cómo borrarla. Primero créala:

```
CREATE DATABASE prueba_borrar;
```

Verifica que existe:

```
SHOW DATABASES;
```

Ahí debe aparecer "prueba_borrar". Ahora bórrala:

```
DROP DATABASE prueba_borrar;
```

`DROP DATABASE` significa "elimina la base de datos". Vuelve a ejecutar `SHOW DATABASES` y comprobarás que "prueba_borrar" ya no está. Desapareció por completo, con todo lo que tuviera dentro.

ADVERTENCIA IMPORTANTE: `DROP DATABASE` no pide confirmación y no hay papelera de reciclaje. Lo que borras, se borra para siempre. Por eso en la vida real jamás ejecutarías un `DROP` sin estar 100% seguro y sin tener un respaldo. En este curso solo lo usaremos con bases de datos de prueba que creamos para ese fin. Grábate esta regla: `DROP` es el comando más peligroso de SQL.

Existe también la versión segura:

```
DROP DATABASE IF EXISTS prueba_borrar;
```

Si la base de datos no existe, no marca error. Útil en scripts, igual que `IF NOT EXISTS`.

Ejemplo 6: comentarios en SQL.

Los comentarios son notas dentro del código que MySQL ignora. Sirven para explicar qué hace cada parte, dejar recordatorios o desactivar temporalmente una línea. Hay dos formas de escribir comentarios.

Comentario de una línea, con dos guiones:

```
-- Este es un comentario de una sola línea  
SHOW DATABASES; -- Este comentario está al final de un comando
```

Todo lo que sigue a "--" en esa línea es ignorado por MySQL. Ojo: después de los dos guiones debe haber un espacio para que MySQL lo reconozca como comentario.

Comentario de varias líneas, con /* y */:

```
/*  
Este es un comentario  
que puede ocupar  
varias líneas  
*/  
SHOW DATABASES;
```

Todo lo que esté entre /* y */ es ignorado, sin importar cuántas líneas ocupe. Es ideal para encabezados de scripts o explicaciones largas.

Los comentarios no afectan la ejecución: son solo para humanos. Úsalos siempre que escribas algo que tu yo del futuro podría no entender. Un código bien comentado es un código profesional.

Ejemplo 7: poner todo junto en un pequeño script.

Un script es un conjunto de comandos que se ejecutan en orden. Vamos a escribir uno que resume todo lo aprendido en esta unidad:

```

/*
Script de práctica - Unidad 1
Autor: estudiante de Lucas EGA Academy
Descripción: crear y explorar bases de datos
*/

-- Ver las bases de datos existentes
SHOW DATABASES;

-- Crear la base de datos de la tienda
CREATE DATABASE IF NOT EXISTS tienda;

-- Verificar que se creó
SHOW DATABASES;

-- Entrar a la base de datos tienda
USE tienda;

-- Confirmar en qué base de datos estoy
SELECT DATABASE();

```

Copia este script completo en Workbench y ejecútalo. Verás cómo cada comando se ejecuta en orden y produce su resultado. Así es como trabajan los profesionales: escriben scripts, los guardan y los ejecutan cuando los necesitan.

Ejemplo 8: crear varias bases de datos de una sola vez.

Puedes ejecutar varios comandos CREATE seguidos, uno por línea:

```

CREATE DATABASE IF NOT EXISTS escuela;
CREATE DATABASE IF NOT EXISTS biblioteca;
CREATE DATABASE IF NOT EXISTS veterinaria;

```

Después ejecuta SHOW DATABASES y verás las tres nuevas bases de datos en la lista. Así de fácil es crear varias a la vez. En la sección de práctica usarás estas bases de datos para los ejercicios.

Ejemplo 9: cambiar de una base de datos a otra.

El comando USE no es permanente: puedes cambiar de base de datos cuando quieras. Mira:

```

USE escuela;
SELECT DATABASE();

```

Te devolverá "escuela". Ahora cambia a otra:

```
USE biblioteca;  
SELECT DATABASE();
```

Te devolverá "biblioteca". Así de simple: USE te mueve entre bases de datos como si cambiaras de carpeta en tu computadora. Recuerda siempre verificar con SELECT DATABASE() en qué base estás antes de crear tablas o hacer operaciones importantes. Es un hábito que te salvará de errores tontos.

Ejemplo 10: ver información del servidor.

Para cerrar esta sección, dos comandos útiles que te dan información del entorno:

```
SELECT VERSION();
```

Te muestra la versión de MySQL que tienes instalada. Útil cuando buscas ayuda en internet: la versión importa porque algunas funciones cambian entre versiones.

```
SELECT USER();
```

Te muestra con qué usuario estás conectado (debería ser algo como "root@localhost"). Confirma que estás trabajando como root en tu servidor local, que es lo esperado en este curso.

Con estos ejemplos ya dominas las operaciones básicas sobre bases de datos: verlas, crearlas, seleccionarlás y borrarlas. En la Unidad 2 entraremos a las tablas, que es donde viven los datos de verdad. Pero primero, práctica.

Ejemplo 11: explorar las bases de datos del sistema (solo mirar, no tocar).

Ya viste que MySQL trae bases de datos del sistema: information_schema, mysql, performance_schema y sys. Vamos a asomarnos a una de ellas para entender para qué sirven, sin modificar nada.

```
USE information_schema;  
SHOW TABLES;
```

Verás una lista larga de tablas con nombres como TABLES, COLUMNS, SCHEMATA. La base "information_schema" es el catálogo interno de MySQL: guarda información sobre todas las demás bases de datos (qué tablas existen, qué columnas tienen, etc.). Es de solo lectura para ti: MySQL la actualiza sola.

Ahora mira la lista de "esquemas" registrados ahí:

```
SELECT SCHEMA_NAME FROM SCHEMATA;
```

SCHEMATA es la tabla que lista todas las bases de datos (recuerda: en MySQL, schema y database son lo mismo). El resultado debe coincidir con lo que te muestra SHOW DATABASES. Es la misma información, vista desde adentro del sistema.

Por último, regresa a tu base de datos de trabajo:

```
USE tienda;  
SELECT DATABASE();
```

Este ejercicio te enseña algo valioso: en MySQL todo es consultable, incluso la información sobre el propio sistema. Más adelante usarás information_schema para inspeccionar tus tablas de forma automática.

Ejemplo 12: script de limpieza y recreación (reset de práctica).

Cuando prácticas mucho, tus bases de datos se llenan de experimentos. Este script borra tus bases de práctica y las vuelve a crear limpias. Es el "botón de reinicio" de tu laboratorio. Guárdalo, te servirá durante todo el curso:


```

/*
Script: reiniciar el laboratorio de la Unidad 1
Borra las bases de práctica y las crea de nuevo, limpias.
CUIDADO: solo usar con bases de datos de práctica.
*/

-- Borrar las bases de práctica (si existen)
DROP DATABASE IF EXISTS tienda;
DROP DATABASE IF EXISTS escuela;
DROP DATABASE IF EXISTS veterinaria;
DROP DATABASE IF EXISTS biblioteca;
DROP DATABASE IF EXISTS cafeteria;
DROP DATABASE IF EXISTS mi_tienda;

-- Volver a crearlas, limpias
CREATE DATABASE tienda;
CREATE DATABASE escuela;
CREATE DATABASE veterinaria;
CREATE DATABASE biblioteca;
CREATE DATABASE cafeteria;
CREATE DATABASE mi_tienda;

-- Verificar el resultado
SHOW DATABASES;

```

Ejecútalo y observa: primero desaparecen, luego reaparecen. Este patrón de "borrar y recrear" es muy común en el desarrollo real: los programadores reinician sus entornos de prueba constantemente. Eso sí, nota que en este script NO usamos IF NOT EXISTS al recrear, porque acabamos de borrarlas y sabemos que no existen. Cada variante tiene su momento: IF NOT EXISTS cuando no estás seguro, la versión simple cuando tienes el control total.

Ejemplo 13: documentar con comentarios un script de un compañero.

Imagina que un compañero te pasa este script sin comentarios y te pide que lo documentes. Leer código ajeno y comentarlo es una habilidad profesional clave:

```

CREATE DATABASE IF NOT EXISTS gimnasio;
USE gimnasio;
SELECT DATABASE();
SHOW DATABASES;

```

Tu trabajo: agregar comentarios que expliquen cada línea. La versión documentada quedaría así:

```

/*
Script documentado: base de datos para un gimnasio
Explica paso a paso lo que hace cada comando
*/

-- Crea la base de datos "gimnasio" solo si aún no existe,
-- para evitar errores si el script se ejecuta dos veces
CREATE DATABASE IF NOT EXISTS gimnasio;

-- Selecciona "gimnasio" como base de datos de trabajo:
-- los comandos siguientes se aplicarán dentro de ella
USE gimnasio;

-- Muestra la base de datos actualmente seleccionada
-- para confirmar que estamos dentro de "gimnasio"
SELECT DATABASE();

-- Lista todas las bases de datos del servidor
-- para verificar que "gimnasio" aparece en la lista
SHOW DATABASES;

```

Fíjate en el estilo de los comentarios: explican el PORQUÉ, no solo el QUÉ. "Crea la base de datos" es obvio al leer el código; lo valioso es explicar "para evitar errores si el script se ejecuta dos veces". Ese es el nivel de documentación que distingue a un profesional.

Con estos 13 ejemplos ya no solo sabes los comandos: sabes explorar el sistema, reiniciar tu laboratorio y documentar código. Eso es pensar como alguien que trabaja con bases de datos todos los días.

Práctica

La teoría sin práctica no sirve de nada. Aquí tienes 8 ejercicios para consolidar todo lo aprendido en la unidad. Haz cada ejercicio tú mismo en MySQL Workbench ANTES de leer la solución. Si te atorras, relee la sección de ejemplos. Las soluciones están explicadas paso a paso para que entiendas el porqué de cada comando.

Ejercicio 1: crea la base de datos de una escuela.

Escenario: te contratan para crear el sistema de una escuela primaria. Lo primero que necesitas es la base de datos que contendrá toda la información: alumnos, maestros, calificaciones. Crea una base de datos llamada "escuela" y verifica que se creó correctamente.

Solución paso a paso:

Paso 1: crear la base de datos. Usamos IF NOT EXISTS para evitar errores si ya la creamos antes:

```
CREATE DATABASE IF NOT EXISTS escuela;
```

¿Por qué IF NOT EXISTS? Porque si ejecutas el script dos veces, la segunda vez no marcará error. Es un hábito profesional.

Paso 2: verificar que existe en la lista:

```
SHOW DATABASES;
```

En el resultado debes ver "escuela". Si la ves, el ejercicio está completo.

Paso 3 (verificación extra): entra a la base de datos y confirma:

```
USE escuela;  
SELECT DATABASE();
```

El segundo comando debe devolver "escuela". Listo: la base de datos de la escuela está creada y seleccionada.

Ejercicio 2: crea las bases de datos de una veterinaria y una biblioteca.

Escenario: ahora tienes dos clientes más. Una veterinaria necesita una base de datos llamada "veterinaria" para sus mascotas y citas. Una biblioteca necesita una base de datos llamada "biblioteca" para sus libros y préstamos. Crea ambas y muestra la lista completa.

Solución paso a paso:

Paso 1: crear las dos bases de datos:

```
CREATE DATABASE IF NOT EXISTS veterinaria;  
CREATE DATABASE IF NOT EXISTS biblioteca;
```

Recuerda: un comando por línea, cada uno terminado en punto y coma. Los nombres en minúsculas, sin espacios.

Paso 2: mostrar todas las bases de datos:

```
SHOW DATABASES;
```

Deberías ver "veterinaria" y "biblioteca" en la lista, junto con "escuela", "tienda" y las bases de datos del sistema. Ya tienes 4 bases de datos de práctica: tienda, escuela, veterinaria y biblioteca.

Paso 3: práctica cambiar entre ellas:

```
USE veterinaria;  
SELECT DATABASE();  
USE biblioteca;  
SELECT DATABASE();
```

El primer SELECT debe devolver "veterinaria" y el segundo "biblioteca". Con esto confirmas que sabes moverte entre bases de datos.

Ejercicio 3: crea y elimina una base de datos temporal.

Escenario: quieres practicar el comando DROP sin riesgo. Crea una base de datos llamada "temporal", verifica que existe, bórrala y verifica que ya no existe.

Solución paso a paso:

Paso 1: crear la base de datos temporal:

```
CREATE DATABASE IF NOT EXISTS temporal;
```

Paso 2: verificar que existe:

```
SHOW DATABASES;
```

Busca "temporal" en la lista. Debe estar ahí.

Paso 3: borrarla:

```
DROP DATABASE temporal;
```

MySQL confirmará la eliminación. Recuerda: este comando no pide confirmación y no hay forma de deshacerlo. Solo lo usamos aquí porque "temporal" es una base de prueba vacía.

Paso 4: verificar que desapareció:

```
SHOW DATABASES;
```

"temporal" ya no debe aparecer en la lista. Ejercicio completo: creaste y eliminaste una base de datos de forma controlada.

Ejercicio 4: escribe un script comentado para un negocio nuevo.

Escenario: un amigo abre una cafetería y te pide que le prepares la base de datos inicial. Escribe un script completo y comentado que: muestre las bases de datos existentes, cree la base "cafeteria" solo si no existe, entre a ella y confirme la base de datos actual.

Solución paso a paso:

Paso 1: escribir el script con comentarios explicativos:

```
/*
Script: base de datos inicial para cafetería
Crea la base de datos "cafeteria" y la selecciona
*/

-- Paso 1: ver las bases de datos que ya existen
SHOW DATABASES;

-- Paso 2: crear la base de datos de la cafetería (solo si no existe)
CREATE DATABASE IF NOT EXISTS cafeteria;

-- Paso 3: entrar a la base de datos cafeteria
USE cafeteria;

-- Paso 4: confirmar en qué base de datos estoy trabajando
SELECT DATABASE();
```

Paso 2: ejecutar el script completo en Workbench y revisar cada resultado. El último resultado debe mostrar "cafeteria".

¿Por qué este ejercicio importa? Porque en el trabajo real no ejecutas comandos sueltos: escribes scripts comentados que otros pueden leer y entender. Este ejercicio te entrena en esa forma profesional de trabajar.

Ejercicio 5: identifica y corrige los errores.

Escenario: un compañero te pasa este script, pero tiene errores. Encuéntralos, explícalos y escribe la versión corregida.

Script con errores:

```
show databases
CREATE DATABASE Mi Tienda;
USE tienda
select database()
```

Solución paso a paso:

Error 1: "show databases" no termina en punto y coma. Sin el punto y coma, MySQL se queda esperando el fin de la instrucción. Además, por convención las palabras clave van en mayúsculas.

Error 2: "CREATE DATABASE Mi Tienda;" tiene dos problemas. Primero, el nombre tiene un espacio ("Mi Tienda"), lo cual causa error de sintaxis: los nombres no pueden llevar espacios. Segundo, usa mayúsculas en el nombre, lo cual puede causar problemas en algunos sistemas. Debe ser "mi_tienda" en minúsculas con guion bajo.

Error 3: "USE tienda" no termina en punto y coma. Mismo problema que el error 1.

Error 4: "select database()" no termina en punto y coma y no usa mayúsculas en las palabras clave. Funciona, pero no sigue las buenas prácticas.

Versión corregida:

```
-- Ver las bases de datos existentes
SHOW DATABASES;

-- Crear la base de datos con nombre válido en minúsculas
CREATE DATABASE IF NOT EXISTS mi_tienda;

-- Entrar a la base de datos tienda
USE tienda;

-- Confirmar la base de datos actual
SELECT DATABASE();
```

Nota: en la versión corregida separamos "mi_tienda" (la que se crea) de "tienda" (a la que entramos), porque son dos bases de datos distintas. El script original mezclaba los nombres, otro error de lógica común en principiantes.

Ejercicio 6: el reto de las 5 bases de datos.

Escenario: imagina que trabajas en una empresa de software y necesitas preparar el entorno de pruebas con 5 bases de datos: "clientes_demo", "ventas_demo", "inventario_demo", "rh_demo" y "pruebas_finales". Escríbelas en un solo script, ejecútalo y luego verifica el resultado.

Solución paso a paso:

Paso 1: escribir el script:

```
/*  
Script: entorno de pruebas con 5 bases de datos demo  
*/  
  
CREATE DATABASE IF NOT EXISTS clientes_demo;  
CREATE DATABASE IF NOT EXISTS ventas_demo;  
CREATE DATABASE IF NOT EXISTS inventario_demo;  
CREATE DATABASE IF NOT EXISTS rh_demo;  
CREATE DATABASE IF NOT EXISTS pruebas_finales;  
  
-- Verificar que las 5 se crearon  
SHOW DATABASES;
```

Paso 2: ejecutar y contar. En el resultado de SHOW DATABASES debes encontrar las 5 bases nuevas. Si falta alguna, revisa que cada línea termine en punto y coma.

Paso 3 (limpieza): cómo son bases de prueba, práctica borrar una:

```
DROP DATABASE IF EXISTS pruebas_finales;  
SHOW DATABASES;
```

"pruebas_finales" debe haber desaparecido. Las otras 4 quedan como evidencia de tu trabajo.

Ejercicio 7: documenta tu servidor.

Escenario: tu profesor te pide un reporte con la información básica de tu instalación de MySQL: la versión del servidor, el usuario con el que estás conectado y la lista de tus bases de datos de práctica. Obtén los tres datos con comandos SQL.

Solución paso a paso:

Paso 1: versión del servidor:

```
SELECT VERSION();
```

Anota el número de versión que te devuelva (por ejemplo, 8.0.x). Te servirá cuando busques ayuda en internet.

Paso 2: usuario actual:

```
SELECT USER();
```

Debe mostrar algo como "root@localhost": estás conectado como root en tu máquina local.

Paso 3: lista de bases de datos:

```
SHOW DATABASES;
```

Cuenta cuántas bases de datos de práctica creaste en esta unidad: tienda, escuela, veterinaria, biblioteca, cafeteria, mi_tienda, clientes_demo, ventas_demo, inventario_demo, rh_demo. Si hiciste todos los ejercicios, debes tener al menos 10 bases de datos propias, más las 4 del sistema.

Puedes juntar los tres comandos en un script comentado y guardarlo como tu primer "reporte de sistema":

```
/*
Reporte del servidor MySQL - Unidad 1
*/

-- Versión del servidor
SELECT VERSION();

-- Usuario conectado
SELECT USER();

-- Bases de datos existentes
SHOW DATABASES;
```

Guarda este script: en el futuro, cuando algo falle, lo primero que te preguntará cualquier persona que te ayude será justamente esta información.

Ejercicio 8: el examen sorpresa del cliente.

Escenario: un cliente te dice por teléfono: "Necesito una base de datos que se llame 'floreria', y quiero que verifiques que quedó creada y seleccionada. Mándame el script por correo". Escríbelo completo, comentado y listo para enviar, sin probarlo primero en tu cabeza más de una vez: como lo harías en el trabajo real.

Solución paso a paso:

Paso 1: escribir el script como si fuera a producción: claro, comentado y a prueba de errores:

```
/*
Script para el cliente: base de datos "floreria"
Fecha: hoy. Autor: tu nombre.
Crea la base de datos y la deja seleccionada.
*/

-- Ver el estado inicial de las bases de datos
SHOW DATABASES;

-- Crear la base de datos de la florería (sin error si ya existe)
CREATE DATABASE IF NOT EXISTS floreria;

-- Seleccionarla como base de datos de trabajo
USE floreria;

-- Confirmar que estamos dentro de "floreria"
SELECT DATABASE();

-- Mostrar la lista final para que el cliente verifique
SHOW DATABASES;
```

Paso 2: ejecutarlo en Workbench y revisar que el último SELECT DATABASE() devuelva "floreria" y que "floreria" aparezca en el SHOW DATABASES final.

Paso 3: reflexión profesional. ¿Qué hiciste bien en este script? Usaste IF NOT EXISTS (a prueba de re-ejecuciones), comentaste cada paso (el cliente entiende qué recibe), verificaste al final (evidencia del trabajo hecho) y seguiste las convenciones (mayúsculas, punto y coma, minúsculas). Eso es exactamente lo que un cliente paga: no solo el resultado, sino la confianza de que está bien hecho.

Si completaste los 8 ejercicios, ya no eres un principiante total: sabes instalar MySQL, conectarte, crear bases de datos, moverte entre ellas, borrarlas con cuidado, escribir scripts comentados, detectar errores comunes y documentar tu entorno. Eso es muchísimo para una primera unidad.

Resumen de la unidad

Llegaste al final de la Unidad 1. Tómate un momento para reconocer lo que lograste: hace unas páginas no sabías nada de bases de datos, y ahora tienes MySQL instalado, sabes qué es SQL y ya creaste tus primeras bases de datos con tus propias manos. Eso es un avance real.

Repasemos todo lo aprendido, punto por punto.

Primero: QUÉ ES UNA BASE DE DATOS. Es un conjunto organizado de información, diseñado para guardarla, buscarla y modificarla de forma eficiente y segura. Usa las analogías que más te sirvan: la biblioteca con su catálogo, el archivero de oficina con sus cajones etiquetados. Lo esencial es la organización: sin ella, los datos son solo un montón inútil.

Segundo: POR QUÉ NO BASTA CON ARCHIVOS O HOJAS DE CÁLCULO. Las bases de datos manejan millones de datos, permiten que miles de usuarios trabajen al mismo tiempo, protegen la información con permisos y garantizan la integridad de los datos. Excel es una gran herramienta, pero no es una base de datos.

Tercero: LAS PIEZAS INTERNAS. La tabla organiza la información en filas y columnas. Las columnas definen qué se guarda (nombre, correo, teléfono). Las filas (o registros) son los datos reales, uno por elemento. Una base de datos contiene tablas, y las tablas se relacionan entre sí: esa es la esencia del modelo relacional.

Cuarto: QUÉ ES SQL. El Lenguaje de Consulta Estructurado, nacido en IBM en los años 70, estándar mundial desde 1986. Es declarativo (dices qué quieres, no cómo conseguirlo), legible (casi parece inglés) y universal (funciona en MySQL, PostgreSQL, SQL Server, Oracle y más). Aprenderlo una vez te sirve para toda la vida profesional.

Quinto: LA DIFERENCIA ENTRE EL LENGUAJE Y EL SISTEMA. SQL es el idioma; MySQL es uno de los sistemas que lo hablan (un DBMS). Otros sistemas: PostgreSQL, SQL Server, Oracle, SQLite, MariaDB. También existen las bases de datos NoSQL (como MongoDB), útiles para casos especiales, pero las relacionales con SQL siguen siendo el estándar para la mayoría de las aplicaciones.

Sexto: INSTALACIÓN Y CONFIGURACIÓN. Instalaste MySQL Community Server (el motor) y MySQL Workbench (la herramienta visual). Configuraste el usuario root con una contraseña segura que apuntaste en un lugar seguro. Aprendiste qué es una conexión local (127.0.0.1, localhost), qué es el puerto 3306 y cómo solucionar los problemas más comunes: servicio detenido, contraseña incorrecta, puerto ocupado y requisitos faltantes.

Séptimo: TUS PRIMEROS COMANDOS SQL. SHOW DATABASES para ver las bases de datos. CREATE DATABASE (con IF NOT EXISTS) para crearlas. USE para seleccionar con cuál trabajas. SELECT DATABASE() para confirmar dónde estás. DROP DATABASE (con IF EXISTS) para borrarlas, con la advertencia de que no hay forma de deshacerlo. Y los comentarios con -- y /* */ para documentar tu código como un profesional.

Octavo: BUENAS PRÁCTICAS Y ERRORES COMUNES. Palabras clave en mayúsculas, punto y coma al final de cada instrucción, nombres en minúsculas sin espacios, comentar el código, probar en entornos seguros y respaldar antes de operaciones peligrosas. Y los errores clásicos que ya sabes evitar: confundir base de datos con tabla, olvidar el USE, olvidar el punto y coma y usar nombres con espacios.

Noveno: CONCEPTOS AVANZADOS QUE YA CONOCES DE NOMBRE. El modelo relacional y su principio de "cada dato en un solo lugar". Las llaves primarias y foráneas que conectan las tablas. Los tipos de datos (INT, VARCHAR, DATE y más). La seguridad básica del root y los respaldos. Todo esto lo practicarás a fondo en las próximas unidades, pero ya no te sonarán extraños.

Si hiciste los 8 ejercicios de práctica, además tienes un portafolio inicial: unas 10 bases de datos de práctica creadas por ti (tienda, escuela, veterinaria, biblioteca, cafetería, mi_tienda y las demo), scripts comentados guardados y un reporte básico de tu servidor. Nada mal para una primera unidad.

¿QUÉ VIENE EN LA UNIDAD 2? Ahora que dominas los contenedores (las bases de datos), vamos a llenarlos: entraremos a las TABLAS. Aprenderás a crear tablas con CREATE TABLE, a definir columnas con sus tipos de datos, a usar llaves primarias, a ver la estructura de tus tablas, a modificarlas y a borrarlas. Es la unidad donde las bases de datos empiezan a guardar información de verdad, y donde todo lo que aprendiste aquí cobrará aún más sentido.

Un último consejo de Esteban antes de cerrar: no corras. Si algo de esta unidad no te quedó claro, repásalo antes de seguir. Las bases de datos se construyen por capas, y cada unidad se apoya en la anterior. Es mejor dominar bien estos fundamentos que avanzar con dudas. La velocidad no importa; la solidez sí.

Nos vemos en la Unidad 2. Sigue practicando, sigue preguntando y sigue construyendo. El mundo de los datos te estaba esperando.

AUTOEVALUACIÓN DE LA UNIDAD 1

Antes de pasar a la Unidad 2, verifica honestamente que puedes hacer cada uno de estos puntos. Si alguno te cuesta, regresa a la sección correspondiente y practícalo hasta dominarlo.

1. Explicar con tus palabras qué es una base de datos, usando al menos una analogía (biblioteca, archivero).
2. Explicar la diferencia entre base de datos, tabla, fila (registro) y columna.
3. Explicar qué es SQL, qué significan sus siglas y por qué es el estándar mundial.
4. Explicar la diferencia entre SQL (el lenguaje) y MySQL (el sistema).
5. Mencionar al menos tres sistemas gestores de bases de datos además de MySQL.
6. Explicar con tus palabras la diferencia entre bases de datos relacionales y NoSQL.
7. Tener MySQL Community Server instalado y corriendo como servicio.
8. Tener MySQL Workbench instalado y una conexión local funcional con el usuario root.
9. Saber qué es el puerto 3306 y qué significa 127.0.0.1 (localhost).
10. Escribir y ejecutar `SHOW DATABASES` y reconocer las bases de datos del sistema.
11. Crear una base de datos con `CREATE DATABASE` y con `CREATE DATABASE IF NOT EXISTS`, explicando la diferencia.
12. Seleccionar una base de datos con `USE` y verificarla con `SELECT DATABASE()`.
13. Borrar una base de datos con `DROP DATABASE` explicando por qué es peligroso.
14. Escribir comentarios de una línea con `--` y de varias líneas con `/* */`.
15. Escribir un script completo, comentado y en el orden correcto: ver, crear, seleccionar, verificar.
16. Identificar al menos tres errores comunes de principiantes en un script ajeno.
17. Obtener la versión del servidor con `SELECT VERSION()` y el usuario actual con `SELECT USER()`.
18. Aplicar las buenas prácticas: mayúsculas en palabras clave, punto y coma final, nombres en minúsculas sin espacios.

Si marcaste los 18 puntos, estás listo para la Unidad 2. Si te faltó alguno, no es un fracaso: es tu mapa de repaso. Los fundamentos bien aprendidos hoy son la velocidad de mañana.

UNIDAD 2: DISEÑO DE TABLAS

En la Unidad 1 aprendiste a pensar como una base de datos: qué es, para qué sirve, cómo está organizada y cómo se habla con ella. Instalaste MySQL, hiciste tu primera conexión y aprendiste las primeras consultas. Todo eso era el cimiento. Ahora empieza la construcción de verdad.

Esta unidad se trata del diseño de tablas, y quiero que entiendas algo desde el principio: las tablas son el corazón de una base de datos. Puedes tener el servidor más rápido, el motor más moderno y las consultas más elegantes, pero si tus tablas están mal diseñadas, todo lo demás va a doler. Y va a doler mucho.

Un mal diseño de tabla es como construir una casa sobre arena. Al principio todo se ve bien: las paredes están derechas, la pintura está fresca, nadie nota nada raro. Pero pasa el tiempo, crecen los datos, llegan los usuarios, se agregan funciones, y de pronto todo empieza a agrietarse. Consultas que tardan una eternidad. Datos duplicados por todas partes. Campos que nadie sabe para qué sirven. Cambios que deberían tomar cinco minutos y terminan tomando cinco días. Todo eso casi siempre se rastrea hasta un mismo origen: alguien diseñó las tablas a la carrera, sin pensar.

La buena noticia es que diseñar bien una tabla no es complicado. No necesitas matemáticas avanzadas ni años de experiencia. Necesitas entender unos cuantos conceptos claros, conocer los tipos de datos disponibles y desarrollar el hábito de pensar antes de escribir el primer CREATE TABLE. Eso es exactamente lo que vamos a hacer en esta unidad.

En la Clase 4 vas a aprender a crear tablas: qué columnas lleva, qué tipos de datos usa cada columna y cómo definir una llave primaria que se genera sola con AUTO_INCREMENT. Vas a ver que crear una tabla no es solo "ponerle columnas", sino tomar decisiones conscientes sobre cada una.

En la Clase 5 vamos a profundizar en los tipos de datos, que es uno de los temas más subestimados de SQL. Muchos programadores usan VARCHAR para todo y DATE nunca, y luego se preguntan por qué su base de datos pesa el triple y sus consultas son lentas. Aquí vas a aprender INT, DECIMAL, VARCHAR, DATE y sus parientes cercanos, y sobre todo vas a aprender cuándo usar cada uno. La diferencia entre un buen diseñador de bases de datos y uno mediocre casi siempre está en la elección de tipos de datos.

En la Clase 6 vas a aprender a meter datos en tus tablas con INSERT INTO. Parece lo más fácil del mundo, y en el fondo lo es, pero hay detalles que importan: cómo especificar columnas, cómo insertar varias filas de un tirón y cómo evitar los errores clásicos que dejan filas con datos incompletos o nulos donde no deberían estar.

Al final de esta unidad vas a ser capaz de mirar un problema del mundo real -una lista de alumnos, un catálogo de productos, una nómina de empleados- y convertirlo en tablas bien diseñadas, con los tipos de datos correctos, las restricciones necesarias y datos de prueba insertados. Esa es la habilidad fundamental sobre la que se construye todo lo demás en SQL. Las consultas complejas, los JOINS, las subconsultas y los reportes que verás en las próximas unidades solo funcionan bien cuando las tablas están bien diseñadas. Así que pon atención: esta unidad paga intereses compuestos.

Antes de entrar en materia, un recordatorio de la Unidad 1: una base de datos es un conjunto de tablas relacionadas. Una tabla es un conjunto de filas y columnas. Cada columna tiene un nombre y un tipo de dato. Cada fila es un registro. El esquema de la tabla es la descripción de sus columnas. Si esto te suena familiar, perfecto: aquí es donde esos conceptos se vuelven concretos, porque los vas a escribir con tus propias manos.

Explicación detallada

Clase 4: Crear tablas - CREATE TABLE

Vamos a empezar por lo más básico y a la vez lo más importante: la anatomía de una tabla. Cuando miras una tabla, ves una cuadrícula con filas y columnas, como una hoja de cálculo. Pero debajo de esa apariencia sencilla hay una estructura rigurosa que tienes que entender.

Una tabla tiene un nombre, y ese nombre debe describir qué guarda: clientes, productos, pedidos, alumnos, empleados. Cada tabla tiene columnas, también llamadas campos. Cada columna tiene tres cosas obligatorias: un nombre, un tipo de dato y una posición en el esquema. Y cada columna representa una característica de lo que la tabla guarda. La tabla clientes tiene columnas como id, nombre, email y fecha_registro. Cada fila de esa tabla es un cliente concreto: Juan Pérez con su email y su fecha de registro.

El esquema de la tabla es la definición completa de sus columnas: qué se llaman, de qué tipo son, cuáles aceptan valores nulos, cuál es la llave primaria. El esquema es el plano arquitectónico de la tabla. Los datos son los habitantes de la casa. Puedes cambiar los datos mil veces al día sin tocar el esquema; cambiar el esquema es remodelar la casa con la gente adentro, así que conviene hacerlo bien desde el principio.

Para crear una tabla se usa la sentencia CREATE TABLE. Su forma general es la siguiente: primero la instrucción, luego el nombre de la tabla, y entre paréntesis la lista de columnas separadas por comas. Cada columna se define con su nombre, su tipo de dato y opcionalmente sus restricciones.

Mira este ejemplo sencillo para crear una tabla de clientes:

```
CREATE TABLE clientes (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    email VARCHAR(150) NOT NULL UNIQUE,  
    telefono VARCHAR(20),  
    fecha_registro DATE DEFAULT (CURDATE())  
);
```

Vamos a desmenuzar esta tabla línea por línea, porque cada detalle importa.

La columna id es de tipo INT, es decir, un número entero. Lleva AUTO_INCREMENT, lo que significa que MySQL genera su valor automáticamente: el primer registro recibe el 1, el segundo el 2, y así sucesivamente. Tú nunca tienes que pensar qué número le toca a cada fila. Y lleva PRIMARY KEY, que la convierte en la llave primaria de la tabla: el identificador único de cada fila. Más adelante en esta misma clase veremos a fondo qué significa eso.

La columna nombre es VARCHAR(100), texto de longitud variable con un máximo de 100 caracteres. Lleva NOT NULL, que significa que no acepta valores vacíos: todo cliente tiene que tener nombre, sin excepción. Es una restricción, una regla que la base de datos hace cumplir por ti.

La columna email también es VARCHAR, pero de 150 caracteres, y lleva dos restricciones: NOT NULL y UNIQUE. UNIQUE significa que no puede haber dos filas con el mismo email. Es la forma de decirle a la base de datos: "el email identifica a cada cliente, no permitas duplicados". Si intentas insertar un cliente con un email que ya existe, MySQL te va a rechazar la operación con un error. Eso es bueno: es la base de datos protegiendo la integridad de tus datos.

La columna teléfono es VARCHAR(20) sin restricciones. Puede quedar vacía (acepta NULL) porque no todos los clientes dan su teléfono. Fíjate que el teléfono se guarda como texto, no como número: nunca vas a sumar ni multiplicar teléfonos, y además pueden llevar guiones, espacios, paréntesis o el signo de más del código de país. Este es un ejemplo perfecto de que el tipo de dato no lo decide la apariencia del valor, sino lo que vas a hacer con él.

La columna `fecha_registro` es de tipo `DATE`, una fecha sin hora. Lleva `DEFAULT (CURDATE())`, que significa que si no especificas un valor al insertar, MySQL pone automáticamente la fecha actual. Los valores por defecto te ahorran trabajo y evitan olvidos.

Esa es la anatomía completa de una tabla bien diseñada en miniatura: identificador automático, textos con longitudes razonables, restricciones que protegen los datos y valores por defecto donde tienen sentido.

La llave primaria y `AUTO_INCREMENT`

La llave primaria (`PRIMARY KEY`) es la columna -o combinación de columnas- que identifica de forma única cada fila de la tabla. Es como el número de matrícula de un coche o la CURP de una persona: no hay dos iguales, y sirve para encontrar un registro específico sin ambigüedad.

Toda tabla debería tener una llave primaria. No es una sugerencia amable, es una regla de diseño fundamental. Sin llave primaria no puedes actualizar ni eliminar una fila específica con seguridad, no puedes relacionar tablas entre sí de forma confiable, y muchos sistemas y herramientas asumen que existe. Si diseñas una tabla sin llave primaria, estás construyendo una casa sin puerta de entrada: técnicamente habitable, prácticamente un problema.

La llave primaria tiene tres propiedades: sus valores son únicos (no se repiten), nunca son nulos (toda fila tiene que tener uno) y son estables (no deberían cambiar con el tiempo). Por eso un número generado automáticamente es la llave primaria ideal en la mayoría de los casos.

`AUTO_INCREMENT` es el mecanismo de MySQL para generar esos números automáticamente. Cuando defines una columna `INT AUTO_INCREMENT PRIMARY KEY`, le estás diciendo a la base de datos: "tú encárgate de numerar las filas". Cada `INSERT` nuevo recibe el siguiente número disponible, sin que tengas que calcularlo ni preocuparte por duplicados. Funciona incluso cuando varios usuarios insertan al mismo tiempo, porque MySQL maneja la concurrencia internamente.

Hay una pregunta que siempre aparece: ¿por qué no usar como llave primaria algo que ya existe, como el email del cliente o su número de teléfono? Porque los datos del mundo real cambian: la gente cambia de email, cambia de teléfono, hasta cambia de nombre. Y a veces lo que creías único resulta no serlo: dos clientes pueden compartir un teléfono de oficina, o un email familiar. Un número artificial generado por la base de datos nunca cambia y nunca se duplica. Por eso la práctica estándar es usar una llave primaria sustituta (un id `AUTO_INCREMENT`) y dejar los datos reales -email, teléfono, nombre- como columnas normales, protegidas con `UNIQUE` cuando corresponda.

El nombre de la columna de la llave primaria suele ser simplemente `id`. Es una convención tan extendida que casi es un estándar: cualquier programador que vea una columna llamada `id` asume que es la llave primaria `AUTO_INCREMENT`. Usa esa convención y tu base de datos será inmediatamente legible para otros.

Restricciones: las reglas que protegen tus datos

Las restricciones (constraints) son reglas que defines en el esquema y que la base de datos hace cumplir automáticamente, sin que tu aplicación tenga que acordarse. Son tu primera línea de defensa contra datos basura. Estas son las principales:

`NOT NULL` significa que la columna no acepta valores vacíos. Úsala en toda columna que sea esencial para el registro. El nombre de un cliente, el precio de un producto, la fecha de un pedido: si el registro no tiene sentido sin ese dato, márcalo `NOT NULL`. Pero no abuses: si marcas todo `NOT NULL`, te obligas a tener todos los datos desde el primer momento, y en la vida real los datos llegan por partes.

`UNIQUE` significa que no puede haber dos filas con el mismo valor en esa columna. Úsala para datos que identifican, como emails, nombres de usuario, códigos de producto o números de serie. Es la forma declarativa de decir "esto no se repite".

`DEFAULT` define el valor que se asigna automáticamente cuando no proporcionas uno en el `INSERT`. Sirve para fechas de registro (la fecha actual), contadores (cero), estados iniciales ("activo", "pendiente") y banderas. Un buen `DEFAULT` reduce la cantidad de cosas que tienes que especificar en cada inserción y evita filas con valores nulos accidentales.

`CHECK` permite definir una condición que los datos deben cumplir. Por ejemplo, que un precio sea mayor que cero, o que una calificación esté entre 1 y 5. No todos los motores la hacían cumplir en versiones viejas de MySQL, pero en las versiones modernas (8.0.16 en adelante) sí funciona. Es útil para reglas de negocio simples que quieres garantizar a nivel de base de datos.

`PRIMARY KEY` ya la vimos: unicidad más obligatoriedad en una sola restricción, con la ventaja adicional de que MySQL crea automáticamente un índice sobre ella para búsquedas rápidas.

La filosofía detrás de las restricciones es sencilla: la base de datos es el último guardián de la calidad de tus datos. Tu aplicación puede tener errores, los formularios web pueden fallar, los scripts de importación pueden traer basura. Si las reglas viven en el esquema, se cumplen siempre, sin importar quién o qué esté insertando datos. Diseñar bien las restricciones es diseñar una base de datos que se defiende sola.

Clase 5: Tipos de datos a fondo

Elegir el tipo de dato correcto para cada columna es una de las decisiones de diseño con más impacto y menos glamour. Nadie presume en una fiesta de que eligió bien sus tipos de datos, pero la diferencia entre una base de datos rápida y compacta y una lenta y gorda muchas veces está ahí.

Un tipo de dato le dice a MySQL tres cosas: qué valores puede guardar la columna, cuánto espacio ocupa cada valor y qué operaciones se pueden hacer con esos valores. Si defines una columna como INT, MySQL sabe que ahí van números enteros, que cada uno ocupa 4 bytes, y que puedes sumarlos, compararlos y ordenarlos numéricamente. Si la defines como VARCHAR, sabe que van textos de longitud variable y que las operaciones numéricas no aplican.

Vamos a recorrer las familias de tipos de datos que vas a usar el 99% del tiempo.

Tipos numéricos: INT, TINYINT, BIGINT, DECIMAL, FLOAT

Los tipos numéricos se dividen en dos grandes grupos: enteros y decimales. La elección entre ellos depende de si necesitas fracciones y de qué tan grandes pueden ser los números.

INT es el entero estándar: guarda números enteros entre -2,147,483,648 y 2,147,483,647, ocupando 4 bytes. Es el caballo de batalla de las bases de datos: sirve para llaves primarias, cantidades, edades, contadores y casi cualquier conteo. Si no estás seguro de qué entero usar, INT es la respuesta correcta el 90% de las veces.

TINYINT es el entero pequeño: ocupa 1 byte y guarda valores de -128 a 127 (o de 0 a 255 si es UNSIGNED). Es perfecto para datos con rangos pequeños y conocidos: el día del mes, la calificación de 1 a 5 estrellas, el número de intentos de inicio de sesión, una bandera de verdadero/falso. Usar TINYINT en lugar de INT para estos casos ahorra 3 bytes por fila, que en una tabla de millones de filas sí se nota.

BIGINT es el entero grande: ocupa 8 bytes y guarda números enormes. Lo necesitas para contadores que pueden crecer sin límite conocido (visitas a un sitio web, identificadores en sistemas gigantes) o para cantidades en centavos cuando manejas dinero a gran escala. Para una llave primaria normal, INT alcanza para más de 2 mil millones de filas; solo usa BIGINT si de verdad esperas acercarte a ese límite.

SMALLINT y MEDIUMINT existen como puntos intermedios (2 y 3 bytes), pero en la práctica se usan poco: la diferencia de espacio rara vez justifica la complicación. Menciónalos en tu vocabulario, pero no te obsesiones.

El modificador UNSIGNED merece una explicación aparte. Un entero normal (SIGNED) guarda negativos y positivos. Si le agregas UNSIGNED, el rango se recorre hacia arriba: un INT UNSIGNED va de 0 a 4,294,967,295. Es ideal para datos que nunca son negativos: edades, cantidades en inventario, precios en centavos, llaves primarias AUTO_INCREMENT. Duplicas el rango positivo sin gastar un byte extra. Para llaves primarias AUTO_INCREMENT, UNSIGNED es una excelente práctica.

DECIMAL es el tipo para números con decimales exactos. Se define con dos parámetros: DECIMAL(precisión, escala). La precisión es el total de dígitos y la escala es cuántos de esos dígitos van después del punto decimal. DECIMAL(10,2) guarda números de hasta 10 dígitos con 2 decimales: perfecto para dinero, donde necesitas exactamente dos decimales para los centavos.

Y aquí viene una regla de oro que te va a ahorrar dolores de cabeza reales: para dinero, SIEMPRE usa DECIMAL, nunca FLOAT. La razón es que FLOAT (y su hermano DOUBLE) guardan los números en formato binario de punto flotante, que no puede representar exactamente todos los decimales. Es el mismo problema de las calculadoras viejas: $0.1 + 0.2$ no da exactamente 0.3 en binario. En una aplicación científica ese error minúsculo no importa; en una factura, un centavo de diferencia es un problema contable. DECIMAL guarda los dígitos exactamente como son, así que las sumas de dinero cuadran al centavo. Memoriza esto: dinero = DECIMAL, mediciones aproximadas = FLOAT.

FLOAT ocupa 4 bytes y DOUBLE ocupa 8 bytes. Sirven para cantidades donde la precisión exacta no es crítica: temperaturas, distancias, pesos aproximados, coordenadas geográficas, resultados de cálculos científicos. Si estás guardando la temperatura de un sensor o la distancia entre dos ciudades, FLOAT está bien. Si estás guardando el precio de un producto o el saldo de una cuenta, DECIMAL es obligatorio.

Tipos de texto: CHAR vs VARCHAR vs TEXT

El texto es probablemente el tipo de dato que más vas a usar, y también donde más errores se cometen. MySQL ofrece tres opciones principales, cada una con su propósito.

CHAR(n) guarda texto de longitud FIJA de n caracteres. Si guardas "hola" en un CHAR(10), MySQL lo rellena con espacios hasta completar 10 caracteres. Es eficiente cuando todos los valores tienen exactamente la misma longitud: códigos de país de 2 letras ("MX", "US", "ES"), claves CURP de 18 caracteres, códigos postales de formato fijo, hashes MD5 de 32 caracteres. En esos casos, CHAR es más rápido porque MySQL sabe exactamente dónde empieza y termina cada valor.

VARCHAR(n) guarda texto de longitud VARIABLE hasta un máximo de n caracteres. Si guardas "hola" en un VARCHAR(100), ocupa solo 4 caracteres más un pequeño overhead para registrar la longitud. Es la opción correcta para casi todo el texto de longitud impredecible: nombres, emails, direcciones, títulos, descripciones cortas. El número entre paréntesis es el máximo permitido, no el tamaño reservado: VARCHAR(100) con un nombre de 8 letras ocupa 8 letras, no 100.

La pregunta eterna es: ¿qué número pongo en el VARCHAR? La respuesta honesta es: un máximo razonable para ese dato en el mundo real. Nombres: 100 es generoso. Emails: 150 o 255, porque el estándar permite emails largos. Títulos de productos: 200. No pongas VARCHAR(10) para un nombre (alguien con nombre largo no va a caber) ni VARCHAR(5000) para todo "por si acaso" (funciona, pero desperdicia la capacidad de MySQL para optimizar y hace tu esquema menos legible). Piensa en el dato real y ponle un límite sensato.

TEXT es para textos largos de longitud impredecible: descripciones de productos, contenido de artículos, comentarios de usuarios, biografías. Guarda hasta 65,535 caracteres en su versión básica, y tiene hermanas mayores (MEDIUMTEXT y LONGTEXT) para textos todavía más grandes. La diferencia práctica con VARCHAR es que TEXT no puede tener un valor DEFAULT en versiones viejas, no se puede indexar completo con la misma facilidad, y MySQL lo maneja de forma menos eficiente en memoria. Regla práctica: si el texto cabe cómodamente en unos cientos de caracteres y quieres ponerle un límite, usa VARCHAR; si es contenido libre y potencialmente largo, usa TEXT.

Un error clásico que debes evitar: guardar fechas, números de teléfono o códigos como tipos incorrectos "porque se ven como texto". Las fechas van en DATE o DATETIME (los veremos enseguida), porque así puedes compararlas, ordenarlas y hacer cálculos con ellas. Los teléfonos van en VARCHAR porque no se calculan con ellos. Los códigos postales van en VARCHAR o CHAR porque pueden empezar con cero (un 06600 guardado como número se convierte en 6600 y pierdes información). El tipo de dato lo decide el USO del valor, no su apariencia.

Tipos de fecha y hora: DATE, DATETIME, TIME, TIMESTAMP

Las fechas merecen su propio tipo de dato, y la razón es simple: con fechas quieres hacer operaciones de fecha. ¿Cuántos días pasaron entre dos fechas? ¿Qué registros son de este mes? ¿Cuáles vencen la próxima semana? Si guardas las fechas como texto ("19/09/2026"), esas preguntas se vuelven un infierno. Si las guardas como DATE, MySQL te da funciones para responderlas en una línea.

DATE guarda una fecha sin hora: año, mes y día. Formato interno YYYY-MM-DD. Úsala para fechas de nacimiento, fechas de registro, fechas de vencimiento, cualquier fecha donde la hora no importe. Ocupa 3 bytes y su rango va del año 1000 al 9999.

DATETIME guarda fecha Y hora: YYYY-MM-DD HH:MM:SS. Úsala para marcas de tiempo de eventos: cuándo se creó un pedido, cuándo se publicó un artículo, cuándo inició sesión un usuario. Ocupa 8 bytes en versiones modernas (5 bytes más fracciones de segundo si las necesitas).

TIME guarda solo la hora del día: HH:MM:SS. Sirve para horarios de apertura, duración de eventos, horas de cita cuando la fecha va en otra columna. Es menos común que DATE y DATETIME, pero tiene su lugar.

TIMESTAMP es parecido a DATETIME pero con dos diferencias importantes: ocupa solo 4 bytes, su rango es más limitado (de 1970 a 2038), y -esto es lo interesante- MySQL lo convierte automáticamente entre la zona horaria del servidor y la de la conexión. Además, TIMESTAMP tiene un truco muy útil: puedes definirlo con DEFAULT CURRENT_TIMESTAMP para que registre automáticamente el momento de la inserción, y con ON UPDATE CURRENT_TIMESTAMP para que se actualice solo cada vez que la fila cambia. Es perfecto para columnas como creado_en y actualizado_en en tablas de auditoría.

La regla práctica es: DATE para fechas puras, DATETIME para marcas de tiempo normales, TIMESTAMP cuando quieras el registro automático de creación/actualización o trabajos con zonas horarias. Y nunca, bajo ninguna circunstancia, guardes fechas como VARCHAR: pierdes las validaciones (MySQL rechaza "2026-13-45" como DATE pero lo acepta feliz como texto), pierdes las funciones de fecha y pierdes el ordenamiento cronológico correcto.

Booleanos y otros tipos

MySQL no tiene un tipo BOOLEAN puro como otros motores; en su lugar, BOOL y BOOLEAN son sinónimos de TINYINT(1). Guardas 0 para falso y 1 para verdadero. Funciona bien para banderas: activo/inactivo, verificado/no verificado, disponible/agotado. Algunos prefieren usar los literales TRUE y FALSE en sus consultas, que MySQL traduce a 1 y 0 automáticamente. Ambas formas son válidas; lo importante es ser consistente.

ENUM es un tipo especial que limita la columna a una lista fija de valores que defines al crear la tabla: por ejemplo, ENUM('pequeno','mediano','grande') para tallas, o ENUM('pendiente','pagado','enviado','cancelado') para el estado de un pedido. Tiene ventajas: ocupa poco espacio, garantiza que nadie meta un estado inventado y documenta los valores posibles en el esquema mismo. Tiene desventajas: agregar un valor nuevo requiere ALTER TABLE, y la lista vive en el esquema en lugar de en una tabla de catálogo. Úsalo para listas

cortas y estables; si la lista puede crecer o necesita descripciones adicionales, mejor crea una tabla de catálogo aparte.

BLOB es la contraparte binaria de TEXT: guarda datos binarios grandes como imágenes, archivos PDF o documentos. En la práctica moderna, casi siempre es mejor guardar los archivos en el disco o en un servicio de almacenamiento y guardar solo la ruta en la base de datos como VARCHAR. Los BLOBs inflan la base de datos, complican los respaldos y rara vez se consultan con SQL. Mención honorífica: existe, úsalo solo si tienes una razón muy buena.

JSON es un tipo relativamente moderno que guarda documentos JSON con validación automática. Es útil para datos semiestructurados: configuraciones, atributos variables de productos, metadatos. Pero ojo: si un dato merece su propia columna con su propio tipo (precio, fecha, nombre), no lo escondas dentro de un JSON. El tipo JSON es para lo verdaderamente variable, no para flojera de diseño.

Cómo elegir el tipo correcto: espacio, precisión y rendimiento

Elegir tipos de datos no es adivinanza, es un proceso con criterios claros. Aquí tienes la guía de decisión que usan los diseñadores experimentados.

Primero, pregunta qué clase de dato es: ¿número, texto, fecha u otro? Esa sola pregunta elimina el 75% de las opciones. Un precio es número; un nombre es texto; una fecha de nacimiento es fecha. Parece obvio, pero el error más común de los principiantes es tratar todo como texto "porque así es más fácil". No lo es: es más fácil hoy y más difícil para siempre.

Segundo, si es número, pregunta si necesita decimales. Si no, es un entero: elige el tamaño según el rango esperado (TINYINT para rangos pequeños, INT para lo general, BIGINT para contadores masivos). Si sí necesita decimales, pregunta si la precisión exacta importa. Dinero, impuestos, saldos, porcentajes financieros: DECIMAL siempre. Mediciones, cálculos científicos, coordenadas: FLOAT o DOUBLE.

Tercero, si es texto, pregunta qué tan largo puede ser y si la longitud es fija. Longitud fija y corta: CHAR. Longitud variable con un máximo razonable: VARCHAR con un límite sensato. Texto largo e impredecible: TEXT. Y recuerda la prueba del cero a la izquierda: si el dato puede empezar con cero y ese cero importa (códigos postales, números de teléfono, folios), es texto aunque esté hecho de dígitos.

Cuarto, si es fecha, pregunta si necesitas la hora. Solo fecha: DATE. Fecha y hora: DATETIME. Registro automático de cuándo se creó o modificó la fila: TIMESTAMP con sus opciones automáticas.

El espacio importa porque se multiplica. Un byte desperdiciado por fila en una tabla de 10 millones de filas son 10 megabytes. Parece poco hasta que sumas todas las columnas mal elegidas de todas las tablas. Pero más importante que el espacio es el rendimiento: los índices sobre columnas compactas son más rápidos, las comparaciones numéricas son más rápidas que las de texto, y las fechas reales permiten optimizaciones que el texto jamás permitirá. Elegir bien los tipos es una de las optimizaciones de rendimiento más baratas que existen: se hace una vez, al diseñar, y paga dividendos para siempre.

La precisión importa porque los errores de redondeo son silenciosos. Nadie nota que un FLOAT perdió una fracción de centavo hasta que el contador descuadra la contabilidad a fin de mes. En dinero y en cualquier cálculo donde el resultado deba ser exacto, DECIMAL no es una recomendación, es una obligación.

Errores comunes al elegir tipos de datos

Vamos a repasar los errores que veo una y otra vez, para que no los cometas tú.

Error 1: VARCHAR sin longitud o con longitud absurda. En MySQL, VARCHAR siempre requiere especificar la longitud máxima: VARCHAR(100), no VARCHAR a secas. Y poner VARCHAR(65535) "para que quepa todo" es una mala idea: comunica que no pensaste en el dato y puede afectar el uso de memoria en operaciones temporales. Pon un límite que refleje la realidad del dato.

Error 2: usar FLOAT para dinero. Ya lo dijimos, pero lo repito porque es el error más caro de está lista: los centavos se pierden, las sumas no cuadran, y el problema aparece meses después cuando ya hay miles de registros afectados. Dinero = DECIMAL(10,2) o la precisión que necesites. Punto.

Error 3: guardar fechas como texto. "19/09/2026" como VARCHAR te impide ordenar cronológicamente (el orden alfabético no es el orden de fechas), te impide validar (nada impide guardar "45/99/2026") y te impide usar las funciones de fecha de MySQL. Las fechas van en DATE, DATETIME o TIMESTAMP. Siempre.

Error 4: usar INT para todo lo numérico sin pensar. El teléfono no es INT (lleva guiones y códigos de país, y no se suma). El código postal no es INT (puede empezar con cero). Un porcentaje de 0 a 100 no necesita INT (TINYINT UNSIGNED basta). Cada número tiene su tipo según su rango y su uso.

Error 5: CHAR para textos de longitud variable. Guardar nombres en CHAR(100) desperdicia espacio en cada fila y te obliga a lidiar con espacios de relleno en las comparaciones. CHAR es solo para longitudes fijas reales.

Error 6: no usar NOT NULL donde corresponde. Una columna de precio que acepta NULL te obligará a manejar nulos en cada consulta, en cada reporte, en cada suma. Si el dato es obligatorio en el mundo real, que sea obligatorio en el esquema.

Buenas prácticas de nomenclatura

Los nombres que elijas para tablas y columnas van a ser leídos cientos de veces: en consultas, en código de aplicaciones, en reportes, por ti dentro de seis meses cuando ya no recuerdes qué significaba esa columna críptica. Nombres buenos ahorran tiempo; nombres malos generan confusión permanente.

Usa snake_case: palabras en minúsculas separadas por guion bajo. fecha_nacimiento, total_pedido, id_cliente. Es la convención dominante en MySQL y en el mundo SQL en general. Evita camelCase (fechaNacimiento) y evita espacios o caracteres especiales en los nombres: te obligarían a usar comillas invertidas en cada consulta.

Sé consistente con singular o plural en los nombres de tablas, y mantén la misma decisión en toda la base de datos. ¿clientes o cliente? ¿productos o producto? Ambas opciones son defendibles; lo indefendible es mezclar: tener clientes junto a producto y pedidos. Elige una y respétala. En este libro usaremos plural para las tablas (clientes, productos, pedidos) porque describe lo que la tabla contiene: muchos clientes, muchos productos.

Los nombres deben ser descriptivos pero no novelas. nombre es mejor que n. email es mejor que e. fecha_registro es mejor que fr. Pero tampoco te vayas al extremo: la_columna_que_guarda_el_nombre_completo_del_cliente es un exceso. Busca el punto medio: lo suficientemente claro para entenderse sin comentarios, lo suficientemente corto para escribirse cómodamente.

Para llaves foráneas (las veremos a fondo en la Unidad 3), usa el patrón id_tabla: id_cliente en la tabla pedidos referencia al id de la tabla clientes. Esta convención hace que las relaciones sean obvias con solo leer los nombres de columna.

Evita palabras reservadas de SQL como nombres de columna: order, group, table, select, date (esta última es especialmente traicionera porque parece inofensiva). Si necesitas una columna de fecha, llámala fecha_pedido o creado_en, no date. Si algún día no tienes alternativa, MySQL permite escapar con comillas invertidas, pero es mejor no llegar ahí.

Y un consejo final de nomenclatura: nombra las columnas booleanas como preguntas o estados claros. activo, verificado, disponible, enviado. Cuando leas "WHERE activo = 1" se entiende solo. Evita nombres ambiguos como flag, status_sin_mas o el temido campo1, campo2, campo3, que es la firma inconfundible de un diseño hecho a las prisas.

Cierre de la explicación: el diseño como hábito

Antes de pasar a los ejemplos, quiero dejarte con la idea central de esta unidad: diseñar una tabla es tomar decisiones, y las buenas decisiones se toman con información. Cada columna que defines responde preguntas: ¿qué dato guardó? ¿de qué tipo es? ¿puede faltar? ¿se repite? ¿quién lo genera? Si te haces esas preguntas para cada columna antes de escribir el CREATE TABLE, tus diseños van a ser sólidos. Si las omites, la base de datos te va a cobrar la factura más tarde, con intereses.

En la siguiente sección vas a ver todo esto en acción: crearás tablas reales, las inspeccionarás, las modificarás y les meterás datos. Manos a la obra.

Ejemplos paso a paso

En esta sección vas a construir con tus propias manos. Abre tu cliente de MySQL y ejecuta cada ejemplo. No te limites a leerlos: escríbelos, ejecútalos, equivócate, corrige. La memoria muscular también cuenta en SQL.

Antes de empezar, asegúrate de estar usando una base de datos de práctica. Si seguiste la Unidad 1, ya sabes cómo crearla y seleccionarla:

```
CREATE DATABASE IF NOT EXISTS tienda_practica;  
USE tienda_practica;
```

CREATE DATABASE crea la base de datos si no existe (el IF NOT EXISTS evita el error si ya la creaste antes). USE la selecciona como tu base de datos de trabajo: a partir de aquí, todas las tablas que crees vivirán dentro de tienda_practica.

Ejemplo 1: crear la tabla clientes

Vamos a crear nuestra primera tabla del mundo real: clientes de una tienda en línea. Aplica todo lo que aprendiste en la explicación: id AUTO_INCREMENT como llave primaria, VARCHAR con longitudes sensatas, NOT NULL donde el dato es obligatorio, UNIQUE en el email y un DEFAULT para la fecha de registro.

```
CREATE TABLE clientes (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    apellido VARCHAR(100) NOT NULL,  
    email VARCHAR(150) NOT NULL UNIQUE,  
    telefono VARCHAR(20),  
    fecha_nacimiento DATE,  
    fecha_registro DATE DEFAULT (CURDATE()),  
    activo TINYINT(1) DEFAULT 1  
);
```

Fíjate en las decisiones: el id es INT UNSIGNED porque nunca será negativo y así duplicamos el rango positivo. Dividimos el nombre en nombre y apellido porque en el mundo real se ordena y se busca por apellido, y tenerlos separados facilita eso. El email es UNIQUE porque identifica al cliente y no queremos duplicados. El teléfono es VARCHAR porque no se hacen cálculos con él. La fecha de nacimiento es DATE sin NOT NULL porque no todos los clientes la proporcionan. La fecha de registro tiene DEFAULT con la fecha actual: un cliente nuevo queda registrado hoy sin que tengamos que decirlo. Y activo es TINYINT(1) con DEFAULT 1: todo cliente nuevo nace activo, y si algún día se da de baja, cambiamos ese 1 por un 0 en lugar de borrar su historial.

Si la sentencia se ejecutó sin errores, MySQL responde con algo como "Query OK, 0 rows affected". La tabla existe, vacía, esperando datos.

Ejemplo 2: crear la tabla productos

Ahora la tabla de productos. Aquí aparece un tipo nuevo: DECIMAL para el precio, y un ENUM para la categoría cuando las opciones son fijas y conocidas.

```
CREATE TABLE productos (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    codigo VARCHAR(30) NOT NULL UNIQUE,  
    nombre VARCHAR(200) NOT NULL,  
    descripcion TEXT,  
    precio DECIMAL(10,2) NOT NULL,  
    stock INT UNSIGNED NOT NULL DEFAULT 0,  
    categoria ENUM('electronica', 'ropa', 'hogar', 'libros', 'otros') NOT NULL,  
    fecha_alta DATETIME DEFAULT CURRENT_TIMESTAMP  
);
```

Las decisiones aquí: código es un identificador legible para humanos (como "LAP-DELL-001") y por eso es UNIQUE pero no es la llave primaria; la llave primaria sigue siendo el id numérico. El precio es DECIMAL(10,2): hasta 10 dígitos con 2 decimales, exacto al centavo, como debe ser el dinero. El stock es INT UNSIGNED con DEFAULT 0: un producto nuevo nace con cero existencias hasta que se registra su inventario, y nunca puede ser negativo a nivel de tipo (aunque más adelante podrías agregar un CHECK para impedirlo del todo). La descripción es TEXT porque puede ser larga e impredecible. La categoría es ENUM porque la tienda maneja esas cinco categorías y no van a cambiar cada semana. Y fecha_alta es DATETIME con DEFAULT CURRENT_TIMESTAMP: registra automáticamente el momento exacto en que se dio de alta el producto, con fecha y hora.

Ejemplo 3: crear la tabla pedidos

La tercera tabla conecta a los clientes con los productos: los pedidos. Aquí aparece id_cliente, que más adelante será una llave foránea formal; por ahora es simplemente la columna que nos dice qué cliente hizo cada pedido.

```
CREATE TABLE pedidos (  
  id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
  id_cliente INT UNSIGNED NOT NULL,  
  fecha_pedido DATETIME DEFAULT CURRENT_TIMESTAMP,  
  total DECIMAL(10,2) NOT NULL,  
  estado ENUM('pendiente', 'pagado', 'enviado', 'entregado', 'cancelado') NOT NULL  
  DEFAULT 'pendiente',  
  notas TEXT  
);
```

Observa el patrón: id_cliente sigue la convención de nomenclatura para referenciar otra tabla (id_ más el nombre de la tabla referenciada). El estado es ENUM con DEFAULT 'pendiente': todo pedido nace pendiente, y su ciclo de vida (pagado, enviado, entregado, cancelado) está limitado a esos valores. El total es DECIMAL(10,2) NOT NULL: un pedido siempre tiene un total, y es dinero, así que DECIMAL. Las notas son TEXT opcional para observaciones libres.

Con estas tres tablas ya tienes el esqueleto de una tienda en línea: quién compra (clientes), qué se vende (productos) y las transacciones (pedidos). En la Unidad 3 aprenderás a conectarlas formalmente con llaves foráneas.

Ejemplo 4: inspeccionar lo que creaste - SHOW TABLES y DESCRIBE

Crear tablas a ciegas no sirve; tienes que saber verificar. MySQL te da dos comandos de inspección esenciales.

SHOW TABLES lista todas las tablas de la base de datos actual:

```
SHOW TABLES;
```

Después de los ejemplos anteriores, deberías ver clientes, productos y pedidos. Es el equivalente a abrir el cajón y comprobar que tus carpetas están ahí.

DESCRIBE muestra el esquema de una tabla: sus columnas, tipos, restricciones y valores por defecto. Es tu herramienta de verificación por excelencia.

```
DESCRIBE clientes;
```

La salida es una tabla con columnas Field (nombre de la columna), Type (tipo de dato), Null (YES si acepta nulos, NO si no), Key (PRI para llave primaria, UNI para UNIQUE), Default (valor por defecto) y Extra (información adicional como auto_increment). Cada vez que crees una tabla, corre DESCRIBE inmediatamente después y verifica que el esquema sea el que diseñaste. Es un hábito de dos segundos que evita horas de depuración.

También puedes usar la forma corta DESC clientes;, que hace exactamente lo mismo. Y si quieres ver la sentencia CREATE TABLE exacta que generó la tabla (útil para documentar o replicar tu diseño), usa:

```
SHOW CREATE TABLE productos;
```

Esto te devuelve el CREATE TABLE completo tal como MySQL lo tiene registrado, incluyendo detalles que no escribiste explícitamente pero que MySQL asumió. Es la forma más fiable de ver "la verdad" sobre tu tabla.

Ejemplo 5: modificar tablas existentes - ALTER TABLE

Rara vez un diseño sale perfecto al primer intentó. Los requisitos cambian, se te olvida una columna, un tipo de dato resulta insuficiente. Para eso existe ALTER TABLE: modifica el esquema de una tabla que ya existe y que puede ya tener datos.

Agregar una columna nueva con ADD COLUMN. Supón que decides guardar la ciudad de cada cliente:

```
ALTER TABLE clientes ADD COLUMN ciudad VARCHAR(80) AFTER telefono;
```

La cláusula AFTER teléfono coloca la columna justo después de teléfono en el esquema. Si no especificas posición, la columna se agrega al final. También puedes usar FIRST para ponerla al principio. El orden de las columnas no afecta el funcionamiento, pero un orden lógico hace el DESCRIBE más legible.

Modificar una columna existente con MODIFY COLUMN. Supón que los nombres de 100 caracteres se quedan cortos para algunos clientes con nombres compuestos largos:

```
ALTER TABLE clientes MODIFY COLUMN nombre VARCHAR(150) NOT NULL;
```

Ojo con MODIFY: tienes que repetir la definición COMPLETA de la columna (tipo y restricciones), no solo lo que cambia. Si olvidas el NOT NULL, la columna quedará aceptando nulos. Es un error común: MODIFY reemplaza la definición entera.

Cambiar el nombre de una columna (y opcionalmente su tipo) con CHANGE COLUMN. Supón que quieres renombrar teléfono a celular:

```
ALTER TABLE clientes CHANGE COLUMN telefono celular VARCHAR(20);
```

CHANGE COLUMN recibe el nombre viejo, el nombre nuevo y la definición completa. Si solo quieres renombrar sin cambiar el tipo, repites el mismo tipo.

Eliminar una columna con DROP COLUMN. Supón que la columna notas de pedidos resultó innecesaria:

```
ALTER TABLE pedidos DROP COLUMN notas;
```

Cuidado: DROP COLUMN borra la columna Y todos sus datos de forma irreversible. No hay papelera de reciclaje. Antes de eliminar una columna en una base de datos real, asegúrate de que nadie la usa y considera respaldar los datos. En práctica, está bien experimentar; en producción, piénsalo dos veces.

Agregar una restricción a una tabla existente. Supón que quieres evitar precios negativos en productos con un CHECK:

```
ALTER TABLE productos ADD CONSTRAINT chk_precio_positivo CHECK (precio > 0);
```

Esto agrega una regla que rechaza cualquier precio menor o igual a cero, tanto en inserciones nuevas como en actualizaciones. Las restricciones también se pueden agregar después: el diseño no tiene que ser perfecto desde el primer día, pero cada regla que agregues protege los datos futuros.

Ejemplo 6: eliminar una tabla - DROP TABLE

A veces una tabla de práctica ya cumplió su propósito, o creaste una con el nombre equivocado. DROP TABLE la elimina por completo: esquema y datos.

```
DROP TABLE IF EXISTS tabla_prueba;
```

El IF EXISTS evita el error si la tabla no existe. Y repito la advertencia porque es importante: DROP TABLE es irreversible. No pregunta dos veces, no hay deshacer. En tu entorno de práctica, úsalo con libertad para experimentar. En una base de datos real, un DROP TABLE accidental es una de las peores pesadillas de un administrador. Por eso los buenos hábitos se practican aquí: verifica el nombre de la tabla dos veces antes de ejecutar.

Ejemplo 7: insertar datos - INSERT INTO básico (Clase 6)

Con las tablas creadas, es hora de llenarlas. La sentencia INSERT INTO agrega filas nuevas. Su forma más explícita y recomendada incluye la lista de columnas:

```
INSERT INTO clientes (nombre, apellido, email, telefono, fecha_nacimiento, ciudad)
VALUES ('Maria', 'Gonzalez', 'maria.gonzalez@correo.com', '555-123-4567',
'1990-05-14', 'Ciudad de Mexico');
```

Fíjate en varios detalles. Primero, especificamos las columnas entre paréntesis después del nombre de la tabla. Esto es una buena práctica fundamental: hace el INSERT legible y resistente a cambios en el esquema (si mañana agregas una columna, este INSERT sigue funcionando). Segundo, NO incluimos el id: cómo es AUTO_INCREMENT, MySQL lo genera solo. Tercero, NO incluimos fecha_registro ni activo: tienen valores DEFAULT que se aplican automáticamente. Cuarto, los textos van entre comillas simples y la fecha en formato YYYY-MM-DD.

Existe una forma abreviada sin lista de columnas:

```
INSERT INTO clientes
VALUES (NULL, 'Juan', 'Perez', 'juan.perez@correo.com', '555-987-6543',
'1985-11-02', CURDATE(), 'Guadalajara', 1);
```

Aquí tienes que proporcionar un valor para CADA columna en el orden del esquema, incluyendo NULL para el id AUTO_INCREMENT (MySQL lo interpreta como "genera el siguiente"). Esta forma es más corta pero más frágil: si alguien agrega o reordena columnas, el INSERT se rompe o, peor, inserta datos en las columnas equivocadas. Úsala solo en pruebas rápidas; en código real, siempre lista las columnas.

Después de insertar, verifica con un SELECT:

```
SELECT * FROM clientes;
```

Deberías ver la fila de María González con su id generado automáticamente (1), su fecha de registro con la fecha de hoy y activo en 1, todo sin que tuvieras que especificarlo.

Ejemplo 8: inserción múltiple - varias filas en un solo INSERT

Insertar fila por fila es tedioso cuando tienes muchos datos iniciales. MySQL permite insertar varias filas en una sola sentencia separando los grupos de VALUES con comas:

```
INSERT INTO productos (codigo, nombre, precio, stock, categoria)
VALUES
    ('LAP-DELL-001', 'Laptop Dell Inspiron 15', 12999.99, 25, 'electronica'),
    ('TEL-SAM-002', 'Telefono Samsung Galaxy A54', 7499.50, 40, 'ropa'),
    ('LIB-SQL-003', 'Libro Curso Completo de SQL', 349.00, 100, 'libros'),
    ('SIL-OFI-004', 'Silla de oficina ergonomica', 2899.99, 15, 'hogar');
```

Un solo INSERT, cuatro productos. Espera, ¿viste el error? El teléfono Samsung quedó con categoría 'ropa', lo cual es obviamente incorrecto. Es un error intencional para enseñarte algo: los datos de ejemplo también pueden tener errores, y detectarlos es parte del trabajo. Lo corregiremos en el siguiente ejemplo con UPDATE (aunque UPDATE se ve a fondo en la Unidad 4, aquí tienes un adelanto porque es la herramienta natural para corregir datos).

La inserción múltiple no solo es más cómoda: también es más rápida. Cada sentencia SQL tiene un costo de comunicación con el servidor; un INSERT con 100 filas es mucho más eficiente que 100 INSERTs de una fila. Para cargas iniciales de datos y scripts de prueba, la inserción múltiple es la forma correcta.

Nota que en este INSERT omitimos descripción y fecha_alta: descripción acepta NULL y fecha_alta tiene DEFAULT CURRENT_TIMESTAMP. Solo incluimos las columnas para las que tenemos valores. Esa es la flexibilidad que te da listar las columnas explícitamente.

Ejemplo 9: corregir datos con UPDATE

Vamos a corregir la categoría del teléfono Samsung. La sentencia UPDATE modifica filas existentes; su compañera inseparable es WHERE, que dice CUÁLES filas modificar.

```
UPDATE productos SET categoria = 'electronica' WHERE codigo = 'TEL-SAM-002';
```

Lee la sentencia en voz alta: "actualiza productos, establece categoría igual a electrónica, donde el código sea TEL-SAM-002". El WHERE es crítico: sin él, el UPDATE modificaría TODAS las filas de la tabla. Imagina ejecutar "UPDATE productos SET categoría = 'electrónica'" sin WHERE: todos tus productos terminarían como electrónica. Es uno de los errores más temidos en SQL, así que desarrolla el reflejo: UPDATE siempre va con WHERE, y antes de ejecutarlo, lee el WHERE dos veces.

Verifica la corrección:

```
SELECT codigo, nombre, categoria FROM productos;
```

Aquí usamos un SELECT con columnas específicas en lugar de SELECT *: es más legible cuando solo te interesan algunas columnas, y es la forma que usarás en el código real.

Ejemplo 10: INSERT con datos de otra tabla (INSERT ... SELECT)

Hay una forma avanzada de INSERT que vale la pena conocer desde ahora: insertar filas tomando los datos de otra tabla con un SELECT. Imagina que quieres crear una tabla de respaldo de clientes activos:

```
CREATE TABLE clientes_activos (  
    id INT UNSIGNED PRIMARY KEY,  
    nombre_completo VARCHAR(205),  
    email VARCHAR(150)  
);  
  
INSERT INTO clientes_activos (id, nombre_completo, email)  
SELECT id, CONCAT(nombre, ' ', apellido), email  
FROM clientes  
WHERE activo = 1;
```


El `INSERT ... SELECT` toma el resultado de la consulta y lo inserta fila por fila en la tabla destino. Las columnas del `SELECT` deben coincidir en número y ser compatibles en tipo con las columnas listadas en el `INSERT`. Aquí usamos `CONCAT` para unir nombre y apellido en una sola columna: es un adelanto de las funciones de texto que verás en la Unidad 4.

Este patrón es muy usado en la vida real: migraciones de datos, tablas de resumen, respaldos parciales, poblar tablas de prueba a partir de datos reales. No necesitas exportar e importar: SQL mueve los datos dentro del propio servidor.

Ejemplo 11: ponerlo todo junto - el flujo completo de una tienda

Cerremos los ejemplos con un flujo completo y realista: registrar un pedido. Esto combina `INSERT` en varias tablas y muestra cómo trabajan juntas.

Primero, un cliente nuevo:

```
INSERT INTO clientes (nombre, apellido, email, ciudad)
VALUES ('Ana', 'Lopez', 'ana.lopez@correo.com', 'Monterrey');
```

Luego, el pedido de ese cliente. Necesitamos su id; supongamos que es el 2 (en la vida real lo obtendrías con un `SELECT` o con la función `LAST_INSERT_ID()`, que devuelve el id generado por el último `INSERT`):

```
INSERT INTO pedidos (id_cliente, total, estado)
VALUES (2, 13249.49, 'pagado');
```

Y para ver el panorama completo, una consulta que junta la información (adelanto de los `JOINS` de la Unidad 3, para que veas hacia dónde vamos):

```
SELECT c.nombre, c.apellido, p.id AS numero_pedido, p.total, p.estado
FROM clientes c, pedidos p
WHERE c.id = p.id_cliente;
```

No te preocupes si la sintaxis del `FROM` con dos tablas te resulta extraña: en la Unidad 3 la veremos con calma. Lo importante ahora es que entiendas el concepto: las tablas no viven aisladas. El `id_cliente` en `pedidos` apunta al `id` en `clientes`, y esa conexión es lo que permite responder preguntas como "¿qué pedidos hizo Ana López?" o "¿cuánto ha comprado cada cliente?". Diseñar bien las tablas -con llaves primarias claras y columnas de referencia con nombres consistentes- es lo que hace posibles esas preguntas.

Práctica

La teoría sin práctica se olvida en una semana. Estos ejercicios están diseñados para que tomes decisiones de diseño de verdad: no hay una sola respuesta correcta, pero sí hay respuestas mejores y peores, y las soluciones explican el porqué de cada decisión. Haz cada ejercicio por tu cuenta antes de leer la solución.

Ejercicio 1: diseña la tabla alumnos

Una escuela en línea necesita una tabla para sus alumnos. Cada alumno tiene: un identificador único, nombre completo, email (que no se puede repetir), fecha de nacimiento, fecha de inscripción (que debe registrarse automáticamente con la fecha actual), y un indicador de si está activo.

Escribe el CREATE TABLE correspondiente, eligiendo tipos de datos y restricciones. Luego verifica con DESCRIBE.

Solución paso a paso:

Paso 1: identifica las columnas y su naturaleza. Identificador único: número entero generado automáticamente. Nombre completo: texto de longitud variable. Email: texto que no se repite. Fecha de nacimiento: fecha pura. Fecha de inscripción: fecha con valor automático. Activo: bandera verdadero/falso.

Paso 2: elige los tipos. El identificador es INT UNSIGNED AUTO_INCREMENT PRIMARY KEY: la llave primaria estándar. El nombre completo es VARCHAR(150): 150 caracteres dan espacio para nombres compuestos largos sin exagerar. El email es VARCHAR(150) con UNIQUE y NOT NULL: el estándar para emails, único porque identifica al alumno. La fecha de nacimiento es DATE: fecha pura, sin hora. La fecha de inscripción es DATE DEFAULT (CURDATE()): se registra sola. Activo es TINYINT(1) DEFAULT 1: bandera compacta, todo alumno nuevo nace activo.

Paso 3: escribe la sentencia.

```
CREATE TABLE alumnos (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    nombre_completo VARCHAR(150) NOT NULL,  
    email VARCHAR(150) NOT NULL UNIQUE,  
    fecha_nacimiento DATE,  
    fecha_inscripcion DATE DEFAULT (CURDATE()),  
    activo TINYINT(1) DEFAULT 1  
);
```

Paso 4: verifica.

```
DESCRIBE alumnos;
```

Revisa que cada columna tenga el tipo esperado, que email muestre UNI en la columna Key, que id muestre PRI y auto_increment en Extra, y que fecha_inscripcion y activo muestren sus valores por defecto. Si algo no coincide, compáralo con tu sentencia y encuentra la diferencia: ese es el aprendizaje.

Decisiones que vale la pena notar: usamos nombre_completo en una sola columna en lugar de nombre y apellido separados. ¿Es correcto? Depende del uso: si la escuela nunca va a ordenar o buscar por apellido, una sola columna es más simple. Si sí lo va a hacer, conviene separarlas como hicimos en clientes. No hay respuesta universal: el diseño responde al uso. En un examen, cualquiera de las dos es válida si la justificas.

Ejercicio 2: diseña la tabla libros

Una librería necesita una tabla para su catálogo. Cada libro tiene: identificador único, código ISBN (13 caracteres, único, formato fijo), título, autor, precio (con centavos exactos), año de publicación, número de páginas, género (solo puede ser: novela, ensayo, poesía, infantil, técnico), y existencias en inventario.

Escribe el CREATE TABLE. Presta especial atención al ISBN, al precio y al género: cada uno pone a prueba un concepto distinto de esta unidad.

Solución paso a paso:

Paso 1: analiza los casos especiales. El ISBN tiene exactamente 13 caracteres siempre: longitud fija, así que es CHAR(13), no VARCHAR. Además puede empezar con cero y nunca se hacen cálculos con él, así que definitivamente es texto. El precio necesita centavos exactos: DECIMAL, nunca FLOAT. El género tiene una lista cerrada de valores: ENUM es el candidato natural.

Paso 2: el resto de las columnas. Identificador: INT UNSIGNED AUTO_INCREMENT PRIMARY KEY. Título: VARCHAR(200) porque los títulos pueden ser largos. Autor: VARCHAR(150). Año de publicación: aquí hay opciones; YEAR es un tipo específico de MySQL para años (ocupa 1 byte), pero INT también funciona y es más portable; usaremos YEAR para practicar, aunque en un diseño real ambas son aceptables. Número de páginas: SMALLINT UNSIGNED es suficiente (ningún libro tiene más de 65,535 páginas) y ahorra espacio frente a INT. Existencias: INT UNSIGNED NOT NULL DEFAULT 0.

Paso 3: escribe la sentencia.

```
CREATE TABLE libros (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    isbn CHAR(13) NOT NULL UNIQUE,  
    titulo VARCHAR(200) NOT NULL,  
    autor VARCHAR(150) NOT NULL,  
    precio DECIMAL(8,2) NOT NULL,  
    anio_publicacion YEAR,  
    paginas SMALLINT UNSIGNED,  
    genero ENUM('novela','ensayo','poesia','infantil','tecnico') NOT NULL,  
    existencias INT UNSIGNED NOT NULL DEFAULT 0  
);
```

Paso 4: prueba los límites. Intenta insertar un libro con un género que no está en la lista y observa el error: es el ENUM protegiéndote. Intenta insertar dos libros con el mismo ISBN y observa el error del UNIQUE. Estas pruebas te enseñan más que diez lecturas: son las restricciones trabajando en vivo.

```
INSERT INTO libros (isbn, titulo, autor, precio, anio_publicacion, paginas, genero,  
existencias)  
VALUES ('9786073161234', 'Cien años de soledad', 'Gabriel Garcia Marquez', 299.00,  
1967, 471, 'novela', 30);
```

Nota: el título real lleva tildes y eñes ("Cien años de soledad", "García Márquez"), y en tu base de datos real deberías escribirlos correctamente. Aquí los simplificamos solo para mantener el ejemplo compatible con cualquier configuración de caracteres; en la práctica, asegúrate de que tu base de datos use utf8mb4 para guardar tildes y eñes sin problemas.

Ejercicio 3: diseña la tabla empleados y justifica cada tipo de dato

Una empresa necesita una tabla de empleados con: identificador único, nombre, apellido, email corporativo único, fecha de contratación, salario mensual (con centavos), departamento (lista fija: ventas, sistemas, rh, finanzas, dirección), teléfono de contacto y número de empleado asignado por la empresa (formato fijo de 6 caracteres, por ejemplo "EMP001" - bueno, en realidad 6 caracteres como "A00123").

El reto de este ejercicio no es solo escribir el CREATE TABLE, sino escribir al lado de cada columna UNA FRASE que justifique el tipo de dato elegido. Esa justificación es la habilidad real: en el trabajo, tendrás que defender tus decisiones de diseño.

Solución paso a paso:

```

CREATE TABLE empleados (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    -- Identificador interno: entero sin signo generado automaticamente.
    -- Nunca es negativo y el AUTO_INCREMENT evita duplicados.
    nombre VARCHAR(100) NOT NULL,
    -- Texto de longitud variable: los nombres varían mucho en longitud.
    -- 100 caracteres cubren nombres compuestos sin desperdiciar.
    apellido VARCHAR(100) NOT NULL,
    -- Separado del nombre para permitir búsquedas y ordenamientos por apellido.
    email VARCHAR(150) NOT NULL UNIQUE,
    -- El email corporativo identifica al empleado: UNIQUE impide duplicados.
    fecha_contratacion DATE NOT NULL,
    -- Fecha pura sin hora: para antigüedad y cálculos de fechas se usa DATE,
    -- nunca texto, para poder ordenar y calcular diferencias.
    salario DECIMAL(10,2) NOT NULL,
    -- Dinero con centavos exactos: DECIMAL evita los errores de redondeo
    -- de FLOAT. Un centavo de error en nómina es inaceptable.
    departamento ENUM('ventas','sistemas','rh','finanzas','direccion') NOT NULL,
    -- Lista cerrada y estable de departamentos: ENUM garantiza que nadie
    -- escriba un departamento inventado o con faltas de ortografía.
    telefono VARCHAR(20),
    -- Texto porque lleva guiones y código de país, y nunca se hacen
    -- cálculos matemáticos con un teléfono. Opcional: puede ser NULL.
    numero_empleado CHAR(6) NOT NULL UNIQUE
    -- Formato fijo de 6 caracteres: CHAR porque todos miden lo mismo.
    -- Es texto porque puede empezar con cero y no se calcula con él.
    -- UNIQUE porque identifica al empleado en los sistemas de la empresa.
);

```

Los comentarios con -- son tus notas de diseño dentro del propio SQL. MySQL los ignora al ejecutar, pero documentan tus decisiones para quien lea el esquema después. Acostúmbrate a comentar las decisiones no obvias: tu yo del futuro te lo va a agradecer.

Verifica con DESCRIBE empleados; y luego inserta dos empleados de prueba para comprobar que todo funciona, incluyendo que el DEFAULT y el AUTO_INCREMENT hacen su trabajo.

Ejercicio 4: inserta datos variados y verifica

Usando la tabla alumnos del Ejercicio 1, realiza las siguientes inserciones:

a) Inserta un alumno completo especificando todas las columnas necesarias (sin id, sin fecha_inscripcion, sin activo: deja que los valores automáticos trabajen).

b) Inserta tres alumnos más en una sola sentencia de inserción múltiple.

c) Intenta insertar un alumno con un email que ya existe y observa qué pasa. Lee el mensaje de error: ¿qué te está diciendo MySQL?

d) Consulta todos los alumnos y verifica que los ids se generaron en secuencia, que las fechas de inscripción son la fecha de hoy y que todos están activos.

Solución paso a paso:

Paso a): inserción individual con columnas explícitas.

```
INSERT INTO alumnos (nombre_completo, email, fecha_nacimiento)
VALUES ('Carlos Hernandez Ruiz', 'carlos.hr@correo.com', '2001-03-22');
```

No incluimos id (AUTO_INCREMENT lo genera), ni fecha_inscripcion (DEFAULT pone hoy), ni activo (DEFAULT pone 1). Solo damos lo que la base de datos no puede deducir sola.

Paso b): inserción múltiple.

```
INSERT INTO alumnos (nombre_completo, email, fecha_nacimiento)
VALUES
    ('Lucia Fernandez', 'lucia.f@correo.com', '1999-07-10'),
    ('Pedro Sanchez', 'pedro.s@correo.com', '2002-12-01'),
    ('Sofia Ramirez', 'sofia.r@correo.com', NULL);
```

Fíjate en el tercer alumno: pasamos NULL explícito para fecha_nacimiento porque no la conocemos. Funciona porque esa columna acepta nulos (no le pusimos NOT NULL). Pasar NULL explícito es válido y a veces más claro que simplemente omitir la columna.

Paso c): el experimento del duplicado.

```
INSERT INTO alumnos (nombre_completo, email)
VALUES ('Carlos Duplicado', 'carlos.hr@correo.com');
```

MySQL responde con un error de entrada duplicada (Duplicate entry). Ese error NO es un fracaso: es la restricción UNIQUE haciendo su trabajo, protegiendo tus datos de un duplicado. En una aplicación real, tu código capturaría ese error y le mostraría al usuario un mensaje amable como "ese email ya está registrado". Aprende a leer los errores de MySQL: casi siempre te dicen exactamente qué salió mal.

Paso d): verificación.

```
SELECT id, nombre_completo, email, fecha_inscripcion, activo FROM alumnos;
```

Comprueba: los ids son 1, 2, 3, 4 en secuencia (el intento duplicado no consumió un id válido... bueno, en realidad con InnoDB el AUTO_INCREMENT sí puede dejar huecos cuando un INSERT falla; si ves un salto, ya sabes por qué). Las fechas de inscripción son hoy. La columna activo muestra 1 en todas las filas. Todo automático, todo correcto.

Ejercicio 5: altera una tabla existente

La escuela del Ejercicio 1 creció y ahora necesita cambios en la tabla alumnos:

- a) Agrega una columna teléfono VARCHAR(20) después del email.
- b) Agrega una columna promedio DECIMAL(4,2) con DEFAULT 0.00 para el promedio de calificaciones del alumno. (Pista: DECIMAL(4,2) permite hasta 99.99, suficiente para un promedio de 0 a 10 o de 0 a 100 según la escala... piensa qué escala usarías tú.)
- c) Cambia la columna nombre_completo a VARCHAR(200) porque algunos nombres internacionales son más largos.
- d) Agrega una restricción CHECK para que el promedio esté entre 0 y 10.
- e) Verifica el esquema final con DESCRIBE y luego inserta un alumno con promedio para probar que todo funciona.

Solución paso a paso:

Paso a): agregar columna en una posición específica.

```
ALTER TABLE alumnos ADD COLUMN telefono VARCHAR(20) AFTER email;
```

Paso b): agregar columna con valor por defecto.

```
ALTER TABLE alumnos ADD COLUMN promedio DECIMAL(4,2) DEFAULT 0.00;
```

DECIMAL(4,2) da 4 dígitos totales con 2 decimales: el máximo es 99.99. Para una escala de 0 a 10 sobra espacio, lo cual está bien; para una escala de 0 a 100 también alcanza. Si tu escuela usa escala de 0 a 10, podrías usar DECIMAL(3,1), pero DECIMAL(4,2) es más flexible. Lo importante es que pensaste en el rango real del dato.

Nota importante: al agregar una columna a una tabla que YA tiene filas, las filas existentes reciben el valor DEFAULT (0.00) o NULL si no hay default. MySQL no te deja con datos inconsistentes: siempre resuelve qué poner en las filas viejas.

Paso c): modificar el tipo de una columna.

```
ALTER TABLE alumnos MODIFY COLUMN nombre_completo VARCHAR(200) NOT NULL;
```

Recuerda: MODIFY exige la definición completa. Si olvidas el NOT NULL original, la columna quedaría aceptando nulos. Siempre revisa la definición actual con DESCRIBE antes de modificar.

Paso d): agregar la restricción CHECK.

```
ALTER TABLE alumnos ADD CONSTRAINT chk_promedio_rango CHECK (promedio >= 0 AND promedio <= 10);
```

Ahora prueba insertar un promedio de 15 y observa cómo MySQL lo rechaza. Luego inserta uno de 8.5 y verifica que pasa. Las restricciones CHECK convierten reglas de negocio ("el promedio va de 0 a 10") en garantías de la base de datos.

Paso e): verificación final.

```
DESCRIBE alumnos;

INSERT INTO alumnos (nombre_completo, email, telefono, promedio)
VALUES ('Diego Torres', 'diego.t@correo.com', '555-444-3322', 9.25);

SELECT nombre_completo, telefono, promedio FROM alumnos WHERE email =
'diego.t@correo.com';
```

El DESCRIBE final debe mostrar las 8 columnas con sus tipos, defaults y restricciones. El SELECT confirma que la inserción respetó el nuevo esquema. Si todo coincide, completaste el ciclo completo de vida de una tabla: crear, poblar, modificar y volver a poblar.

Ejercicio 6 (reto): diseña desde cero la base de una biblioteca

Este es el reto integrador de la unidad. Diseña TRES tablas para una biblioteca pequeña: socios, libros y prestamos. Los préstamos conectan socios con libros: cada préstamo registra qué socio se llevó qué libro, en qué fecha y con qué fecha de devolución esperada.

Requisitos: socios tiene id, nombre, apellido, email único y fecha de registro automática. libros tiene id, isbn CHAR(13) único, título, autor, precio DECIMAL y existencias. prestamos tiene id, id_socio, id_libro, fecha_prestamo con default de hoy, fecha_devolucion (DATE) y estado ENUM('activo','devuelto','vencido') con default 'activo'.

Después de crearlas, inserta 2 socios, 3 libros y 2 préstamos, y verifica todo con SELECT.

Solución (esquema; las inserciones quedan como práctica guiada):

```
CREATE TABLE socios (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(100) NOT NULL,  
    apellido VARCHAR(100) NOT NULL,  
    email VARCHAR(150) NOT NULL UNIQUE,  
    fecha_registro DATE DEFAULT (CURDATE())  
);  
  
CREATE TABLE libros (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    isbn CHAR(13) NOT NULL UNIQUE,  
    titulo VARCHAR(200) NOT NULL,  
    autor VARCHAR(150) NOT NULL,  
    precio DECIMAL(8,2) NOT NULL,  
    existencias INT UNSIGNED NOT NULL DEFAULT 0  
);  
  
CREATE TABLE prestamos (  
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,  
    id_socio INT UNSIGNED NOT NULL,  
    id_libro INT UNSIGNED NOT NULL,  
    fecha_prestamo DATE DEFAULT (CURDATE()),  
    fecha_devolucion DATE,  
    estado ENUM('activo','devuelto','vencido') NOT NULL DEFAULT 'activo'  
);
```

Fíjate en el patrón que se repite: cada tabla tiene su id AUTO_INCREMENT, las columnas de referencia siguen la convención id_tabla, los ENUMs modelan estados con valores por defecto sensatos, y el dinero va en DECIMAL. Si tus tres tablas se parecen a estas, ya estás diseñando como profesional. Las inserciones te las dejo a ti: si llegaste hasta aquí, sabes exactamente cómo hacerlas.

Resumen de la unidad

Llegamos al final de la Unidad 2. Hagamos un recuento de todo lo que ahora sabes hacer, porque es mucho más de lo que parece.

En la Clase 4 aprendiste la anatomía de una tabla: nombre, columnas, filas y esquema. Aprendiste a crear tablas con CREATE TABLE, definiendo cada columna con nombre, tipo de dato y restricciones. Entendiste la llave primaria como el identificador único de cada fila y

AUTO_INCREMENT como el mecanismo que genera esos identificadores automáticamente. Aprendiste las restricciones fundamentales: NOT NULL para datos obligatorios, UNIQUE para datos que no se repiten, DEFAULT para valores automáticos, CHECK para reglas de negocio y PRIMARY KEY como la combinación de unicidad y obligatoriedad. Y entendiste la filosofía detrás de todo esto: la base de datos es la última guardiana de la calidad de tus datos.

En la Clase 5 profundizaste en los tipos de datos, la decisión de diseño más subestimada y con más impacto. Recorraste los numéricos (INT para lo general, TINYINT para rangos pequeños, BIGINT para contadores masivos, DECIMAL para dinero y valores exactos, FLOAT para mediciones aproximadas), los de texto (CHAR para longitud fija, VARCHAR para longitud variable con límite sensato, TEXT para contenido largo), los de fecha (DATE para fechas puras, DATETIME para marcas de tiempo, TIMESTAMP para registro automático) y los especiales (TINYINT(1) como booleano, ENUM para listas cerradas). Aprendiste las reglas de oro: dinero siempre en DECIMAL nunca en FLOAT, fechas siempre en tipos de fecha nunca en texto, y el tipo lo decide el uso del dato no su apariencia. También viste los errores clásicos (VARCHAR sin pensar el límite, teléfonos como INT, fechas como texto) y las buenas prácticas de nomenclatura (snake_case, consistencia en singular/plural, nombres descriptivos, convención id_tabla para referencias, evitar palabras reservadas).

En la Clase 6 aprendiste a poblar tus tablas con INSERT INTO: la forma explícita con lista de columnas (la recomendada), la forma abreviada sin lista (solo para pruebas rápidas), la inserción múltiple para cargar varias filas de un tirón, y el INSERT ... SELECT para mover datos entre tablas. Viste cómo los valores AUTO_INCREMENT, DEFAULT y las restricciones trabajan automáticamente en cada inserción, y tuviste un adelanto de UPDATE con WHERE para corregir datos.

En los ejemplos paso a paso construiste una mini tienda en línea completa: clientes, productos y pedidos, con decisiones de diseño justificadas en cada columna. Aprendiste a inspeccionar con SHOW TABLES, DESCRIBE y SHOW CREATE TABLE, a modificar esquemas con ALTER TABLE (ADD, MODIFY, CHANGE y DROP COLUMN), a eliminar tablas con DROP TABLE, y viste el flujo completo de registrar un pedido conectando las tres tablas.

En la práctica diseñaste tablas de alumnos, libros y empleados justificando cada tipo de dato, insertaste datos variados incluyendo un experimento con duplicados que te mostró las restricciones en acción, alteraste una tabla existente agregando columnas y restricciones, y cerraste con el reto integrador de una biblioteca de tres tablas.

La idea central que debes llevarte: diseñar bien desde el inicio evita dolores de cabeza después. Cada columna que defines es una decisión; cada tipo de dato que eliges tiene consecuencias en espacio, precisión y rendimiento; cada restricción que agregas es una regla

que la base de datos hará cumplir por ti para siempre. El tiempo que inviertes pensando antes del primer CREATE TABLE se paga solo con creces.

En la Unidad 3 vamos a conectar todo: las llaves foráneas que relacionan formalmente tus tablas y los JOINS que te permiten consultar datos combinados de varias tablas a la vez. Es donde tus tablas dejan de ser islas y se convierten en una verdadera base de datos relacional. Nos vemos ahí.

Esteban - Lucas EGA Academy - lucasegaacademy.com

UNIDAD 3: CONSULTAS BÁSICAS: SELECT Y WHERE

Introducción

Hay una pregunta que distingue a quien apenas esta aprendiendo SQL de quien ya lo domina: cuando se enfrenta a una base de datos nueva, que hace primero? La respuesta de un profesional casi siempre es la misma: consulta. Antes de crear tablas, antes de insertar registros, antes de configurar permisos, un analista de datos abre la base, escribe un SELECT y empieza a preguntar. Porque consultar es la habilidad número uno en SQL, y no es una exageración: alrededor del ochenta por ciento del trabajo diario con bases de datos consiste en pedir información que ya existe y que necesitamos ver de cierta forma.

Piénsalo así. La empresa ya tiene los datos: los clientes están registrados, las ventas se guardaron, los empleados tienen su historial. El trabajo del día a día no es inventar datos nuevos, sino responder preguntas sobre los que ya existen. Cuantos clientes compraron este mes? Cual es el producto más vendido? Quienes son los empleados que entraron este año y todavía no tienen asignado un bono? Cada una de esas preguntas se responde con SELECT. Si dominas SELECT, dominas el ochenta por ciento de SQL. Todo lo demás (crear tablas, insertar, actualizar, borrar) es el veinte por ciento restante.

SELECT es el comando para pedir datos. Su nombre lo dice todo: selecciona. Pero no es solo "dame todo". SELECT te permite ser preciso: que columnas quieres ver, de que tabla las quieres, y lo más importante, cuáles filas te interesan. Esa precisión es la que convierte a SQL en una herramienta profesional y no en un simple visor de tablas. Sin SELECT, una base de datos seria un cajón lleno de papeles sin poder buscar nada.

En esta unidad vas a aprender las dos piezas más fundamentales del lenguaje: SELECT y WHERE. SELECT responde a la pregunta "que quiero ver?" (las columnas) y "de donde?" (la tabla). WHERE responde a "cuáles filas me interesan?" (las condiciones). Juntas forman la consulta básica que vas a escribir miles de veces en tu carrera.

La unidad se organiza en tres clases. La clase 7 cubre el SELECT básico: la sintaxis SELECT * FROM tabla y que significa realmente pedir "todas las columnas". La clase 8 te ensena a seleccionar columnas especificas, que es lo que harás el noventa y nueve por ciento de las veces en trabajo real. La clase 9 introduce WHERE, el filtro que convierte una consulta ordinaria en una respuesta útil: condiciones con números, textos, fechas, rangos con BETWEEN, listas con IN y el manejo correcto de valores nulos con IS NULL.

Al terminar esta unidad seras capaz de abrir cualquier tabla y extraer exactamente la información que necesitas, ni más ni menos. Ese es el fundamento de todo lo que sigue: los ordenamientos, las agrupaciones, las funciones y las uniones de tablas son adornos sobre esta base. Si SELECT y WHERE te quedan claros, el resto del curso se siente natural. Si te quedan flojos, todo lo demás cuesta el doble. Así que vamos despacio y a fondo: esta unidad se lo merece.

Antes de empezar, una nota sobre la práctica. En la Unidad 1 aprendiste a instalar MySQL y en la Unidad 2 creaste tus primeras tablas. Si todavía no tienes la tabla empleados que usaremos en los ejemplos, no te preocupes: en la sección de ejemplos la construiremos paso a paso con CREATE TABLE e INSERT para que puedas replicar cada consulta en tu computadora. Cada ejemplo esta pensado para que lo ejecutes tú mismo. SQL se aprende escribiendo consultas, no leyendo sobre consultas. Ten tu terminal o tu herramienta gráfica lista.

Explicación detallada

La anatomía de una consulta SELECT

Toda consulta SELECT, por simple que parezca, esta compuesta por clausulas. Una cláusula es cada bloque de la consulta: SELECT, FROM, WHERE, y más adelante verás ORDER BY, GROUP BY, HAVING, LIMIT y JOIN. Las clausulas no se escriben en cualquier orden: SQL exige un orden sintactico fijo. El orden en que las ESCRIBES es este:

1. SELECT - que columnas quieres ver.
2. FROM - de que tabla (o tablas) vienen los datos.
3. WHERE - que filas te interesan (filtro).
4. GROUP BY - como agrupar las filas (lo verás en la Unidad 5).
5. HAVING - filtro sobre los grupos (Unidad 5).
6. ORDER BY - en que orden mostrar el resultado (Unidad 4).
7. LIMIT - cuantas filas mostrar como máximo (Unidad 4).

Pero aquí viene algo que sorprende a todos los principiantes: el orden en que las ESCRIBES no es el orden en que el motor las EJECUTA. El motor de la base de datos procesa las clausulas en un orden lógico distinto, y entender ese orden te va a ahorrar muchisimos dolores de cabeza en el futuro. El orden lógico de ejecución es:

1. FROM - primero el motor ubica la tabla y carga las filas.
2. WHERE - después filtra esas filas según la condición.
3. GROUP BY - luego agrupa (si hay).
4. HAVING - filtra los grupos (si hay).
5. SELECT - hasta aquí el motor decide que columnas mostrar.
6. ORDER BY - ordena el resultado.
7. LIMIT - recorta el número de filas.

Fíjate en lo importante: WHERE se ejecuta ANTES que SELECT. Esto significa que en el WHERE no puedes usar los alias de columnas que defines en el SELECT, porque cuando el motor esta filtrando filas todavía no ha procesado la lista de columnas. Es un error clásico que verás más adelante:

```
SELECT salario * 12 AS salario_anual
FROM empleados
WHERE salario_anual > 3000000;
```

Esta consulta falla en la mayoría de los motores porque salario_anual no existe todavía cuando se ejecuta el WHERE. La forma correcta es repetir la expresión o usar una subconsulta:

```
SELECT salario * 12 AS salario_anual
FROM empleados
WHERE salario * 12 > 3000000;
```

Guarda este detalle en la memoria: FROM primero, WHERE después, SELECT al final. Cuando una consulta te de un error raro con un alias, casi siempre la causa es esta.

SELECT * : pedirlo todo

La forma más simple de consultar es pedir todas las columnas de una tabla con el asterisco:

```
SELECT * FROM empleados;
```

El asterisco significa "todas las columnas". Esta consulta devuelve cada columna y cada fila de la tabla empleados. Es perfecto para explorar: cuando te encuentras una tabla nueva, lo primero que haces es un SELECT * para ver que columnas tiene, que tipo de datos guarda y cómo se ven los valores reales.

Pero aquí va una de las lecciones más importantes de esta unidad: `SELECT *` es una herramienta de exploración, no una práctica de producción. En código real, en reportes, en aplicaciones, debes evitar el asterisco y escribir las columnas explícitamente. ¿Por qué? Hay varias razones de peso.

Primera: rendimiento. Si la tabla tiene treinta columnas y tu solo necesitas tres, el asterisco obliga al motor a leer y transferir las treinta. En una tabla con millones de filas eso significa mover datos innecesarios por la red y por la memoria. Es como pedir el menu completo del restaurante cuando solo quieres un taco.

Segunda: fragilidad. Si alguien agrega, quita o reordena columnas en la tabla, tu consulta con asterisco devuelve un resultado distinto sin avisar. El código que consume esos datos (una aplicación, un reporte) puede romperse de formas misteriosas. Con columnas explícitas, tu consulta siempre devuelve exactamente lo que pediste, en el orden que lo pediste.

Tercera: claridad. Cuando otro programador lee `SELECT nombre, puesto, salario FROM empleados` sabe exactamente que datos usa la consulta. Cuando lee `SELECT * FROM empleados` tiene que ir a revisar la estructura de la tabla para entender qué esta pasando.

Cuarta: seguridad. Las tablas a veces contienen columnas sensibles: contraseñas, números de tarjeta, datos personales. Un `SELECT *` las expone todas sin querer. Pedir solo las columnas necesarias es también una buena práctica de protección de datos.

La regla de oro: usa `SELECT *` para explorar y aprender; usa columnas explícitas para todo lo demás.

SELECT con columnas explícitas

Seleccionar columnas específicas es tan simple como listarlas separadas por comas:

```
SELECT nombre, puesto, salario FROM empleados;
```

Esta consulta devuelve solo tres columnas, en ese orden. Puedes pedir las en el orden que quieras, aunque en la tabla esten en otro orden:

```
SELECT salario, nombre FROM empleados;
```

También puedes repetir una columna o combinar columnas con expresiones. SQL permite calcular sobre la marcha:

```
SELECT nombre, salario, salario * 12 AS salario_anual FROM empleados;
```

Aquí aparecen dos conceptos nuevos. El primero es que SELECT no solo muestra columnas guardadas: puede mostrar el resultado de expresiones y calculos. El segundo es el alias de columna con AS.

Alias de columnas con AS

Un alias es un nombre temporal que le das a una columna en el resultado. Se escribe con la palabra clave AS (en MySQL el AS es opcional, pero escribirlo hace la consulta más legible):

```
SELECT nombre AS empleado, salario * 12 AS salario_anual FROM empleados;
```

El resultado mostrara las columnas con los encabezados "empleado" y "salario_anual" en lugar de "nombre" y la expresión cruda. Los alias sirven para tres cosas: dar nombres legibles a columnas calculadas, acortar nombres largos y preparar el resultado para reportes.

Si el alias lleva espacios, encierralo entre comillas dobles o acentos graves según el motor:

```
SELECT nombre AS "Nombre del empleado", salario AS "Salario mensual" FROM empleados;
```

En MySQL también puedes usar acentos graves (backticks) para nombres con espacios o caracteres especiales. Lo importante es que el alias solo existe en el resultado de la consulta: no cambia el nombre real de la columna en la tabla.

Recuerda la advertencia del orden lógico: el alias se define en SELECT, así que no puedes usarlo en WHERE. Pero si puedes usarlo en ORDER BY, porque ORDER BY se ejecuta después de SELECT. Lo verás en la Unidad 4.

WHERE a fondo: el filtro

Si SELECT responde "que columnas", WHERE responde "cuáles filas". Es la cláusula que convierte una consulta en una respuesta útil. Sin WHERE, siempre obtienes todas las filas. Con WHERE, obtienes solo las que cumplen una condición.

```
SELECT nombre, salario FROM empleados WHERE salario > 20000;
```

Lee esta consulta en voz alta: "selecciona nombre y salario de empleados donde el salario sea mayor a 20000". Así de directo es SQL: casi se lee como español.

La condición del WHERE se evalúa fila por fila. El motor toma la primera fila de la tabla, revisa si cumple la condición; si la cumple, la incluye en el resultado; si no, la descarta. Repite con la

segunda fila, la tercera, y así hasta el final. Este modelo mental ("fila por fila") te va a servir para entender condiciones complejas más adelante: el WHERE no es magia, es un filtro que prueba cada fila.

Operadores de comparación

Los operadores más básicos comparan un valor de la columna con un valor que tu das:

- = : igual a. WHERE puesto = 'Programador'
- <> o != : distinto de. WHERE ciudad <> 'Monterrey'
- > : mayor que. WHERE salario > 20000
- < : menor que. WHERE salario < 15000
- >= : mayor o igual que. WHERE salario >= 20000
- <= : menor o igual que. WHERE salario <= 15000

En SQL el operador de igualdad es un solo signo = (no doble como en muchos lenguajes de programación). Y para "distinto de", el estándar es <>, aunque MySQL también acepta !=. Acostúmbrate a usar <>, que funciona en todos los motores.

Los operadores funcionan con números, textos y fechas. Con textos, la comparación es alfabética según el orden del idioma configurado:

```
SELECT nombre FROM empleados WHERE nombre > 'M';
```

Esto devuelve los empleados cuyo nombre empieza con letra posterior a M. Con fechas funciona cronológicamente:

```
SELECT nombre, fecha_contratacion FROM empleados WHERE fecha_contratacion >= '2024-01-01';
```

Devuelve los contratados a partir del primero de enero de 2024. Nota el formato de fecha: 'AAAA-MM-DD' entre comillas simples. Es el formato estándar que MySQL entiende sin ambigüedades.

El detalle de las comillas en textos

Este es el error número uno de los principiantes, así que pon atención. Los valores de texto (cadenas) en SQL van entre comillas SIMPLES:

```
SELECT * FROM empleados WHERE puesto = 'Programador';
```

Si olvidas las comillas, el motor cree que Programador es el nombre de una columna y te lanza un error de "columna desconocida". Los números NO llevan comillas:

```
SELECT * FROM empleados WHERE salario > 20000;
```

Si pones '20000' entre comillas, MySQL igual lo convierte y funciona, pero es una mala práctica: confunde al lector y en algunos motores puede impedir que se usen índices. Regla simple: texto con comillas simples, números sin comillas, fechas con comillas simples en formato AAAA-MM-DD.

BETWEEN: rangos

Cuando necesitas un rango (entre dos valores), puedes escribir dos condiciones... o usar BETWEEN, que es más legible:

```
SELECT nombre, salario FROM empleados WHERE salario BETWEEN 15000 AND 25000;
```

BETWEEN incluye los extremos: esta consulta trae salarios desde 15000 hasta 25000, incluyendo ambos. Equivale a:

```
SELECT nombre, salario FROM empleados WHERE salario >= 15000 AND salario <= 25000;
```

BETWEEN funciona también con fechas y textos:

```
SELECT nombre, fecha_contratacion FROM empleados  
WHERE fecha_contratacion BETWEEN '2023-01-01' AND '2023-12-31';
```

Esto trae a todos los contratados durante el año 2023. Cuidado con un detalle: el valor menor va primero. Si escribes BETWEEN 25000 AND 15000 (al revés), el resultado será vacío porque ningún valor puede ser a la vez mayor o igual a 25000 y menor o igual a 15000.

IN: listas de valores

Cuando quieres filtrar por varios valores posibles de la misma columna, IN es tu mejor amigo:

```
SELECT nombre, ciudad FROM empleados WHERE ciudad IN ('Monterrey', 'Guadalajara',  
'Puebla');
```

Esto trae los empleados que viven en cualquiera de esas tres ciudades. Sin IN tendrías que escribir:

```
SELECT nombre, ciudad FROM empleados
WHERE ciudad = 'Monterrey' OR ciudad = 'Guadalajara' OR ciudad = 'Puebla';
```

IN es más corto, más legible y menos propenso a errores. También existe NOT IN para lo contrario (los que NO están en la lista):

```
SELECT nombre, ciudad FROM empleados WHERE ciudad NOT IN ('Monterrey',
'Guadalajara');
```

Un aviso importante sobre NOT IN que te va a servir en tu carrera: si la lista contiene un valor NULL, NOT IN se comporta de forma extraña y puede devolver cero filas. Es un tema avanzado relacionado con la lógica de tres valores que verás más adelante; por ahora, solo recuerda: con NOT IN, asegurate de que no haya nulos en juego.

IS NULL / IS NOT NULL: el manejo correcto de los nulos

NULL significa "sin valor", "desconocido", "no aplica". No es cero, no es texto vacío, no es falso: es la ausencia de dato. Y tiene una regla de oro que debes grabarte: NULL no se compara con =. Nunca.

```
SELECT * FROM empleados WHERE bono = NULL;
```

Esta consulta NO devuelve nada, ni siquiera las filas donde bono es nulo. ¿Por qué? Porque comparar cualquier cosa con NULL da como resultado "desconocido", no verdadero ni falso, y el WHERE solo incluye filas donde la condición es verdadera. Es como preguntar "es lo desconocido igual a lo desconocido?": la respuesta honesta es "no se sabe".

La forma correcta de buscar nulos es con IS NULL:

```
SELECT nombre, bono FROM empleados WHERE bono IS NULL;
```

Esto si devuelve los empleados que no tienen bono asignado. Y para los que SI tienen:

```
SELECT nombre, bono FROM empleados WHERE bono IS NOT NULL;
```

Este es uno de los errores más comunes en exámenes, entrevistas y trabajo real. Si alguna vez un filtro "no trae nada" y no entiendes por qué, revisa si hay nulos involucrados y si usaste = en lugar de IS NULL.

Como piensa el motor la condición fila por fila

Volvamos al modelo mental del filtro fila por fila, porque con condiciones más ricas vale la pena verlo en acción. Toma esta consulta:

```
SELECT nombre, puesto, salario FROM empleados WHERE salario >= 18000 AND ciudad = 'Monterrey';
```

(Nota: AND y OR los verás a fondo en la Unidad 4, pero ya puedes intuirlos: AND exige que ambas condiciones se cumplan.)

El motor hace esto con cada fila:

1. Toma la fila de Ana: salario 22000, ciudad Monterrey. 22000 >= 18000? Si. Ciudad = Monterrey? Si. Ambas verdaderas: la fila entra al resultado.
2. Toma la fila de Bruno: salario 16000, ciudad Monterrey. 16000 >= 18000? No. Se descarta, sin importar lo demás.
3. Toma la fila de Carla: salario 25000, ciudad Puebla. 25000 >= 18000? Si. Ciudad = Monterrey? No. Se descarta.
4. Y así con todas.

Este proceso es exactamente lo que hace el motor, millones de veces por segundo. Cuando escribas un WHERE, imagina que eres el motor revisando una fila: "cumple? no cumple?". Si puedes responder eso para una fila de ejemplo, tu condición está bien planteada.

Errores comunes (y como evitarlos)

Error 1: olvidar las comillas en textos. WHERE puesto = Programador falla porque el motor busca una columna llamada Programador. Solución: comillas simples siempre para texto.

Error 2: comparar NULL con =. WHERE bono = NULL nunca devuelve filas. Solución: IS NULL o IS NOT NULL.

Error 3: WHERE sin sentido por confusión de tipos. WHERE fecha_contratacion > 2024 (sin comillas ni formato de fecha) puede dar resultados raros. Solución: fechas entre comillas simples en formato 'AAAA-MM-DD'.

Error 4: usar el alias del SELECT dentro del WHERE. Como el WHERE se ejecuta antes que el SELECT, el alias aun no existe. Solución: repite la expresión en el WHERE.

Error 5: BETWEEN con los limites al revés. BETWEEN 25000 AND 15000 devuelve vacío. Solución: el menor primero.

Error 6: SELECT * en consultas de producción. Funciona, pero es lento, fragil y poco claro. Solución: lista las columnas que necesitas.

Error 7: punto y coma olvidado o de más. En la mayoría de las herramientas cada consulta termina con ;. Si ejecutas varias consultas juntas sin separador, obtendras un error de sintaxis.

Buenas prácticas

Pide solo lo necesario. Cada columna que agregas al SELECT es trabajo extra para el motor y ruido extra para quien lee el resultado. Antes de escribir la consulta, preguntate: que columnas necesito realmente? Si la respuesta es tres, escribe tres.

Escribe las palabras clave en mayusculas. No es obligatorio (SQL no distingue mayusculas en palabras clave), pero SELECT ... FROM ... WHERE ... en mayusculas se lee mucho mejor que todo en minusculas, sobre todo cuando las consultas crecen. Los nombres de tablas y columnas, en minusculas con guion bajo: buena costumbre desde el día uno.

Comenta tus consultas. Cuando una condición no es obvia, deja una nota con -- (doble guion) o /* ... */:

```
-- Empleados de Monterrey con salario competitivo para el reporte de compensaciones
SELECT nombre, puesto, salario
FROM empleados
WHERE ciudad = 'Monterrey' AND salario >= 20000;
```

Tu yo del futuro (y tus companeros) te lo van a agradecer. Una consulta sin comentarios es un acertijo; con comentarios, es documentación.

Una consulta por proposito. Si necesitas dos respuestas distintas, escribe dos consultas claras en lugar de una consulta monstruosa que intenta responder todo. La simplicidad es una virtud en SQL.

Prueba con pocos datos primero. Antes de correr un WHERE sobre una tabla de diez millones de filas, pruebalo con LIMIT 10 (lo verás en la Unidad 4) o en una tabla de prueba. Es más rápido iterar y más seguro experimentar.

Ejemplos paso a paso

Para que puedas ejecutar cada ejemplo en tu computadora, primero creamos la tabla empleados con la que trabajaremos en toda la unidad. Copia y ejecuta este bloque:

```
CREATE TABLE empleados (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    nombre VARCHAR(100),  
    puesto VARCHAR(50),  
    salario DECIMAL(10,2),  
    ciudad VARCHAR(50),  
    bono DECIMAL(10,2),  
    fecha_contratacion DATE  
);
```

Ahora insertamos diez empleados de ejemplo. Fíjate que algunos tienen bono NULL (no se les ha asignado) y que hay variedad de puestos, salarios y ciudades para que los filtros tengan sentido:

```
INSERT INTO empleados (nombre, puesto, salario, ciudad, bono, fecha_contratacion)  
VALUES  
( 'Ana Lopez', 'Programadora', 22000.00, 'Monterrey', 3000.00, '2022-03-15'),  
( 'Bruno Diaz', 'Disenador', 16000.00, 'Monterrey', NULL, '2023-06-01'),  
( 'Carla Ruiz', 'Programadora', 25000.00, 'Guadalajara', 5000.00, '2021-11-20'),  
( 'Diego Torres', 'Analista', 18000.00, 'Puebla', 2000.00, '2023-01-10'),  
( 'Elena Mora', 'Gerente', 35000.00, 'Monterrey', 8000.00, '2020-05-25'),  
( 'Fabian Castro', 'Programador', 21000.00, 'Guadalajara', NULL, '2024-02-14'),  
( 'Gabriela Soto', 'Analista', 17500.00, 'Puebla', 1500.00, '2023-09-30'),  
( 'Hector Vega', 'Soporte', 14000.00, 'Monterrey', NULL, '2024-07-08'),  
( 'Irene Pardo', 'Disenadora', 16500.00, 'Guadalajara', 1000.00, '2022-12-05'),  
( 'Jorge Luna', 'Programador', 23000.00, 'Puebla', 4000.00, '2021-08-17');
```

Verifica que los datos quedaron bien con un conteo rápido:

```
SELECT COUNT(*) FROM empleados;
```

Resultado esperado: 10. COUNT(*) cuenta las filas y lo verás a fondo en la Unidad 5; por ahora solo confirma que la tabla tiene diez registros.

Ejemplo 1: SELECT * para explorar la tabla

```
SELECT * FROM empleados;
```

Explicación: el asterisco pide todas las columnas (id, nombre, puesto, salario, ciudad, bono, fecha_contratacion) de todas las filas. Resultado esperado: las diez filas completas. Usalo cuando explores una tabla nueva para conocer su estructura y sus datos. Recuerda: es para exploración, no para código de producción.

Ejemplo 2: seleccionar columnas específicas

```
SELECT nombre, puesto, salario FROM empleados;
```

Explicación: pedimos solo tres columnas, en ese orden. Resultado esperado: diez filas, cada una con nombre, puesto y salario. Nota que id, ciudad, bono y fecha_contratacion no aparecen: el motor ni siquiera las incluye en el resultado. Así se ve una consulta profesional: precisa y sin ruido.

Ejemplo 3: cambiar el orden de las columnas

```
SELECT ciudad, nombre FROM empleados;
```

Explicación: las columnas salen en el orden en que las escribes, no en el orden de la tabla. Resultado esperado: diez filas con ciudad primero y nombre después. Útil cuando preparas un reporte donde el orden de presentación importa.

Ejemplo 4: columnas calculadas y alias con AS

```
SELECT nombre, salario, salario * 12 AS salario_anual FROM empleados;
```

Explicación: la tercera columna no existe en la tabla: se calcula multiplicando el salario mensual por doce. El alias salario_anual le da un encabezado legible. Resultado esperado: diez filas; por ejemplo, Ana Lopez mostrara 22000.00 y 264000.00. Los alias hacen que los reportes se entiendan sin explicación adicional.

Ejemplo 5: alias con espacios para reportes

```
SELECT nombre AS "Nombre del empleado", puesto AS "Puesto actual" FROM empleados;
```

Explicación: cuando el alias lleva espacios se encierra entre comillas dobles. Resultado esperado: diez filas con encabezados "Nombre del empleado" y "Puesto actual". Ideal para resultados que vera gente no técnica.

Ejemplo 6: WHERE con números (mayor que)

```
SELECT nombre, salario FROM empleados WHERE salario > 20000;
```

Explicación: el filtro conserva solo las filas donde el salario es estrictamente mayor a 20000. Resultado esperado: Ana Lopez (22000), Carla Ruiz (25000), Elena Mora (35000), Fabian Castro (21000) y Jorge Luna (23000). Cinco filas. Bruno, Diego, Gabriela, Hector e Irene quedan fuera por no cumplir la condición.

Ejemplo 7: WHERE con texto (igualdad exacta)

```
SELECT nombre, puesto FROM empleados WHERE puesto = 'Programador';
```

Explicación: comparamos la columna puesto con el texto 'Programador', entre comillas simples. Resultado esperado: Fabian Castro y Jorge Luna. Ojo: 'Programadora' (en femenino) NO coincide porque la comparación de textos es exacta, carácter por carácter. Ana Lopez y Carla Ruiz son 'Programadora', así que no aparecen. Este detalle muerde a muchos principiantes: revisa cómo están escritos los datos reales.

Ejemplo 8: WHERE con distinto de

```
SELECT nombre, ciudad FROM empleados WHERE ciudad <> 'Monterrey';
```

Explicación: <> significa "distinto de". Resultado esperado: los empleados que NO viven en Monterrey: Carla Ruiz, Fabian Castro e Irene Pardo (Guadalajara), Diego Torres, Gabriela Soto y Jorge Luna (Puebla). Seis filas.

Ejemplo 9: WHERE con fechas

```
SELECT nombre, fecha_contratacion FROM empleados
WHERE fecha_contratacion >= '2023-01-01';
```

Explicación: la fecha va entre comillas simples en formato AAAA-MM-DD. Resultado esperado: Bruno Diaz (2023-06-01), Diego Torres (2023-01-10), Fabian Castro (2024-02-14), Gabriela Soto (2023-09-30) y Hector Vega (2024-07-08). Cinco filas: todos los contratados desde el inicio de 2023.

Ejemplo 10: BETWEEN con números

```
SELECT nombre, salario FROM empleados WHERE salario BETWEEN 16000 AND 22000;
```

Explicación: BETWEEN incluye los extremos. Resultado esperado: Ana Lopez (22000), Bruno Diaz (16000), Diego Torres (18000), Fabian Castro (21000), Gabriela Soto (17500) e Irene Pardo (16500). Seis filas. Carla (25000), Elena (35000), Hector (14000) y Jorge (23000) quedan fuera del rango.

Ejemplo 11: BETWEEN con fechas

```
SELECT nombre, fecha_contratacion FROM empleados
WHERE fecha_contratacion BETWEEN '2023-01-01' AND '2023-12-31';
```

Explicación: trae a los contratados durante todo el 2023. Resultado esperado: Bruno Diaz, Diego Torres y Gabriela Soto. Tres filas. Fabian y Hector son de 2024, así que no entran; el resto es de 2021 o 2022.

Ejemplo 12: IN con lista de valores

```
SELECT nombre, puesto, ciudad FROM empleados
WHERE ciudad IN ('Monterrey', 'Puebla');
```

Explicación: IN verifica si la ciudad está en la lista. Resultado esperado: siete filas, todos los de Monterrey (Ana, Bruno, Elena, Hector) y Puebla (Diego, Gabriela, Jorge). Los tres de Guadalajara quedan fuera.

Ejemplo 13: NOT IN

```
SELECT nombre, puesto FROM empleados WHERE puesto NOT IN ('Programador', 'Programadora');
```

Explicación: trae los que NO son programadores. Resultado esperado: Bruno Diaz (Disenador), Diego Torres (Analista), Elena Mora (Gerente), Gabriela Soto (Analista), Hector Vega (Soporte) e Irene Pardo (Disenadora). Seis filas.

Ejemplo 14: IS NULL (el correcto para nulos)

```
SELECT nombre, puesto, bono FROM empleados WHERE bono IS NULL;
```

Explicación: busca los empleados sin bono asignado. Resultado esperado: Bruno Diaz, Fabian Castro y Hector Vega. Tres filas. Esta es la forma correcta; con = NULL no obtendrias nada.

Ejemplo 15: IS NOT NULL

```
SELECT nombre, bono FROM empleados WHERE bono IS NOT NULL;
```

Explicación: los que SI tienen bono. Resultado esperado: las otras siete filas (Ana, Carla, Diego, Elena, Gabriela, Irene, Jorge).

Ejemplo 16: combinando filtro y columnas calculadas

```
SELECT nombre, salario, salario + bono AS total FROM empleados WHERE bono IS NOT NULL;
```

Explicación: primero el WHERE filtra las siete filas con bono, y luego el SELECT calcula salario + bono para cada una con el alias "total". Resultado esperado: por ejemplo, Ana Lopez mostrara 22000.00 y 25000.00. Observa el orden lógico en acción: el filtro ocurre antes del calculo.

Ejemplo 17: WHERE con texto y mayusculas

```
SELECT nombre FROM empleados WHERE nombre > 'G';
```

Explicación: comparación alfabética: nombres que empiezan con letra posterior a G. Resultado esperado: Hector Vega e Irene Pardo (H e I). Gabriela Soto empieza con G, pero la comparación es de la cadena completa: 'Gabriela Soto' > 'G' es verdadero porque al comparar carácter por carácter, después de la G viene 'a', y cualquier letra es mayor que "nada". Así que Gabriela SI aparece. Este ejemplo muestra que las comparaciones de texto comparan cadena completa, no solo la primera letra. Resultado final: Gabriela Soto, Hector Vega, Irene Pardo y Jorge Luna. Cuatro filas.

Ejemplo 18: varias condiciones con AND

```
SELECT nombre, puesto, salario, ciudad FROM empleados
WHERE puesto = 'Analista' AND salario > 17000;
```

Explicación: AND exige que ambas condiciones se cumplan. Resultado esperado: Diego Torres (Analista, 18000). Gabriela Soto es Analista pero gana 17500... espera, 17500 > 17000 si se cumple. Revisemos fila por fila: Diego Torres: puesto Analista? Si. Salario 18000 > 17000? Si. Entra. Gabriela Soto: puesto Analista? Si. Salario 17500 > 17000? Si. Entra. Resultado: Diego Torres y Gabriela Soto. Dos filas. (AND y OR se estudian a fondo en la Unidad 4; este es solo un adelanto para que veas cómo se combinan filtros.)

Ejemplo 19: consulta comentada lista para reporte

```
-- Programadores con buen salario: candidatos a aumento
SELECT nombre AS "Empleado", salario AS "Salario actual", ciudad AS "Ciudad"
FROM empleados
WHERE puesto IN ('Programador', 'Programadora') AND salario >= 21000;
```

Explicación: el comentario con -- explica el propósito. El WHERE combina IN con una comparación numérica. Resultado esperado fila por fila: Ana Lopez (Programadora, 22000: cumple ambas), Carla Ruiz (Programadora, 25000: cumple), Fabian Castro (Programador, 21000: cumple, el >= incluye el límite), Jorge Luna (Programador, 23000: cumple). Cuatro filas. Así se ve una consulta profesional completa: comentada, con alias legibles y filtro preciso.

Ejemplo 20: el error clásico con NULL (para que lo veas fallar)

```
SELECT nombre FROM empleados WHERE bono = NULL;
```

Explicación: esta consulta es un ERROR didactico: ejecutala y verás que devuelve cero filas aunque tres empleados no tienen bono. El = nunca funciona con NULL. La versión correcta es la del ejemplo 14 con IS NULL. Ver el falló con tus propios ojos es la mejor forma de no olvidarlo.

Práctica

Es momento de ensuciarte las manos. Los ejercicios usan la tabla empleados que creaste en la sección anterior. Intenta resolver cada uno ANTES de ver la solución: escribe tu consulta, ejecutala y compara tu resultado con el esperado. Si te atorras más de diez minutos, mira la pista y vuelve a intentarlo. Equivocarse aquí es gratis y es la forma más rápida de aprender.

Ejercicio 1: filtrar por puesto

Pregunta: muestra el nombre y el salario de todos los empleados cuyo puesto sea 'Analista'.

Pista: necesitas dos columnas en el SELECT y una condición de igualdad sobre texto en el WHERE. Recuerda las comillas simples.

Solución:

```
SELECT nombre, salario FROM empleados WHERE puesto = 'Analista';
```

Explicación paso a paso:

1. Que columnas? nombre y salario. Van en el SELECT separadas por coma.
2. De que tabla? empleados. Va en el FROM.
3. Que filas? las que tengan puesto igual a 'Analista'. El valor es texto, así que va entre comillas simples.
4. El motor revisa fila por fila: Diego Torres (Analista: si), Gabriela Soto (Analista: si), el resto no. Resultado esperado: dos filas, Diego Torres con 18000.00 y Gabriela Soto con 17500.00.

Error típico: escribir WHERE puesto = Analista sin comillas. El motor buscara una columna llamada Analista y lanzara error.

Ejercicio 2: rango de salario con BETWEEN

Pregunta: muestra nombre, puesto y salario de los empleados que ganen entre 17000 y 23000 (incluyendo los limites), usando BETWEEN.

Pista: BETWEEN va seguido del limite inferior, la palabra AND y el limite superior.

Solución:

```
SELECT nombre, puesto, salario FROM empleados WHERE salario BETWEEN 17000 AND 23000;
```

Explicación paso a paso:

1. Columnas: nombre, puesto, salario.
2. Tabla: empleados.
3. Condición: salario BETWEEN 17000 AND 23000. Los números van sin comillas y el menor va primero.
4. Revisión fila por fila: Ana (22000: si), Bruno (16000: no), Carla (25000: no), Diego (18000: si), Elena (35000: no), Fabian (21000: si), Gabriela (17500: si), Hector (14000: no), Irene (16500: no), Jorge (23000: si, el limite se incluye).
5. Resultado esperado: cinco filas (Ana, Diego, Fabian, Gabriela, Jorge).

Variante: resuelvelo también sin BETWEEN, con >= y AND, y comprueba que da el mismo resultado. Es el mismo filtro escrito de otra forma.

Ejercicio 3: filtrar por ciudad con IN

Pregunta: muestra nombre y ciudad de los empleados que vivan en Guadalajara o en Puebla.

Pista: son dos valores posibles para la misma columna: el caso perfecto para IN.

Solución:

```
SELECT nombre, ciudad FROM empleados WHERE ciudad IN ('Guadalajara', 'Puebla');
```

Explicación paso a paso:

1. Columnas: nombre, ciudad.
2. Tabla: empleados.

3. Condición: ciudad IN ('Guadalajara', 'Puebla'). Cada texto entre comillas simples, separados por comas, todo entre paréntesis.
4. El motor conserva las filas cuya ciudad aparezca en la lista: Carla Ruiz, Fabian Castro, Irene Pardo (Guadalajara), Diego Torres, Gabriela Soto, Jorge Luna (Puebla).
5. Resultado esperado: seis filas. Los cuatro de Monterrey quedan fuera.

Reflexion: compara con la versión usando OR: WHERE ciudad = 'Guadalajara' OR ciudad = 'Puebla'. Da lo mismo, pero IN es más limpio. Cuando la lista crezca a cinco o diez valores, la diferencia se nota muchísimo.

Ejercicio 4: encontrar los nulos

Pregunta: muestra el nombre y el puesto de los empleados que NO tienen bono asignado.

Pista: "no tienen bono" significa que la columna bono es NULL. Y recuerda la regla de oro de los nulos.

Solución:

```
SELECT nombre, puesto FROM empleados WHERE bono IS NULL;
```

Explicación paso a paso:

1. Columnas: nombre, puesto.
2. Tabla: empleados.
3. Condición: bono IS NULL. No se usa =, no se usan comillas: IS NULL es el operador especial para nulos.
4. Revisión: Bruno Díaz (bono NULL: si), Fabian Castro (NULL: si), Hector Vega (NULL: si). El resto tiene un número, así que no.
5. Resultado esperado: tres filas.

Trampa clásica: si escribes WHERE bono = NULL obtienes cero filas y ningún error. El motor no te avisa: simplemente ninguna comparación con NULL da verdadero. Por eso este ejercicio existe: para que el reflejo "nulo = IS NULL" te quede grabado.

Ejercicio 5: combinar filtros (puesto y salario)

Pregunta: muestra nombre, puesto, salario y ciudad de los programadores y programadoras que ganen más de 22000.

Pista: el puesto puede ser 'Programador' o 'Programadora' (usa IN), y el salario debe ser mayor a 22000. Une ambas condiciones.

Solución:

```
SELECT nombre, puesto, salario, ciudad FROM empleados
WHERE puesto IN ('Programador', 'Programadora') AND salario > 22000;
```

Explicación paso a paso:

1. Columnas: nombre, puesto, salario, ciudad.
2. Tabla: empleados.
3. Condición 1: puesto IN ('Programador', 'Programadora') para atrapar ambas variantes del puesto.
4. Condición 2: salario > 22000 (estrictamente mayor, sin incluir el 22000).
5. AND exige que se cumplan las dos a la vez. Revisión fila por fila:
 - Ana Lopez: Programadora? Si. 22000 > 22000? No (es igual, no mayor). Fuera.
 - Carla Ruiz: Programadora? Si. 25000 > 22000? Si. Dentro.
 - Fabian Castro: Programador? Si. 21000 > 22000? No. Fuera.
 - Jorge Luna: Programador? Si. 23000 > 22000? Si. Dentro.
6. Resultado esperado: dos filas, Carla Ruiz y Jorge Luna.

Lección: el > estricto excluye el límite. Si el enunciado dijera "22000 o más", usarías >=. Leer con cuidado si el límite se incluye o no es media batalla en los filtros numéricos.

Ejercicio 6: fechas de contratación

Pregunta: muestra nombre y fecha de contratación de los empleados contratados en el año 2024, ordenados como tu quieras (el orden lo veremos en la Unidad 4, no te preocupes por eso ahora).

Pista: "en el año 2024" significa desde '2024-01-01' hasta '2024-12-31'. BETWEEN con fechas.

Solución:

```
SELECT nombre, fecha_contratacion FROM empleados
WHERE fecha_contratacion BETWEEN '2024-01-01' AND '2024-12-31';
```

Explicación paso a paso:

1. Columnas: nombre, fecha_contratacion.
2. Tabla: empleados.
3. Condición: rango de fechas completo del 2024. Las fechas van entre comillas simples en formato AAAA-MM-DD.
4. Revision: Fabian Castro (2024-02-14: si), Hector Vega (2024-07-08: si). Nadie más es de 2024.
5. Resultado esperado: dos filas.

Nota: si hubieras escrito WHERE fecha_contratacion >= '2024-01-01' también funcionaria con estos datos, pero BETWEEN expresa mejor la idea de "durante el año 2024" y protege contra datos futuros.

Ejercicio 7: los que SI tienen bono, con calculo

Pregunta: muestra el nombre, el salario, el bono y el total (salario + bono) de los empleados que tengan bono asignado. Usa un alias para la columna calculada.

Pista: "tengan bono asignado" = IS NOT NULL. La columna calculada es salario + bono con AS.

Solución:

```
SELECT nombre, salario, bono, salario + bono AS total
FROM empleados
WHERE bono IS NOT NULL;
```

Explicación paso a paso:

1. Columnas: nombre, salario, bono y la expresión salario + bono con el alias "total".
2. Tabla: empleados.
3. Condición: bono IS NOT NULL para excluir a Bruno, Fabian y Hector.
4. El motor primero filtra (WHERE) y después calcula (SELECT): para cada una de las siete filas con bono, suma salario + bono. Por ejemplo, Ana: 22000 + 3000 = 25000.
5. Resultado esperado: siete filas con cuatro columnas cada una.

Lección doble: practicaste IS NOT NULL y viste el orden lógico en acción (el WHERE filtra antes de que el SELECT calcule). Además, el alias "total" hace que el resultado se lea como un reporte.

Ejercicio 8: reto combinado

Pregunta: muestra el nombre, el puesto y la ciudad de los empleados que NO vivan en Monterrey, que ganen menos de 20000 y que SI tengan bono asignado.

Pista: son tres condiciones unidas con AND: ciudad distinta de 'Monterrey', salario < 20000, bono IS NOT NULL.

Solución:

```
SELECT nombre, puesto, ciudad FROM empleados
WHERE ciudad <> 'Monterrey' AND salario < 20000 AND bono IS NOT NULL;
```

Explicación paso a paso:

1. Columnas: nombre, puesto, ciudad.
2. Tabla: empleados.
3. Tres condiciones con AND (las tres deben cumplirse):
 - ciudad <> 'Monterrey': excluye a Ana, Bruno, Elena y Hector.
 - salario < 20000: de los restantes, Diego (18000: si), Gabriela (17500: si), Carla (25000: no), Fabian (21000: no), Irene (16500: si), Jorge (23000: no).
 - bono IS NOT NULL: Diego (2000: si), Gabriela (1500: si), Irene (1000: si).
4. Resultado esperado: tres filas: Diego Torres, Gabriela Soto e Irene Pardo.

Si llegaste aquí sin mirar las soluciones, felicidades: ya sabes filtrar como un profesional. Si tuviste que mirar un par, no pasa nada: vuelve a hacerlos mañana sin mirar. La fluidez con WHERE se construye con repetición, no con lectura.

Autoevaluación rápida

Responde mentalmente (o en voz alta) estas preguntas. Si alguna te cuesta, regresa a la sección correspondiente.

1. Cual es el orden lógico de ejecución de FROM, WHERE y SELECT? (Respuesta: FROM, luego WHERE, luego SELECT.)

2. Por qué se evita `SELECT *` en producción? (Rendimiento, fragilidad ante cambios, claridad y seguridad.)
3. Cómo se buscan valores nulos? (Con `IS NULL`; nunca con `= NULL`.)
4. Que incluye `BETWEEN`: los extremos o no? (Los incluye.)
5. Los textos en el `WHERE` llevan comillas simples, dobles o ninguna? (Simples.)
6. Se puede usar en el `WHERE` un alias definido en el `SELECT`? (No, porque el `WHERE` se ejecuta antes.)
7. Que devuelve `WHERE ciudad NOT IN ('A', 'B')` si hay un `NULL` en la lista? (Comportamiento extraño: puede devolver cero filas. Evítalo o filtra los nulos.)

Las tres clases de la unidad, en un mapa

Esta unidad agrupa tres clases. Ya viste todo su contenido en la explicación, pero vale la pena ver como encajan entre si.

Clase 7: `SELECT` básico. La consulta más simple posible: `SELECT * FROM tabla`. Aquí aprendiste qué significa el asterisco, cuando usarlo (exploración) y cuando no (producción), y la anatomía de la consulta con sus cláusulas y el orden lógico de ejecución. Es la clase del "que es una consulta".

Clase 8: `SELECT` con columnas. El paso de "dame todo" a "dame exactamente esto": listar columnas explícitas, cambiar su orden, crear columnas calculadas y ponerles alias con `AS`. Es la clase de la precisión, y representa lo que harás el noventa y nueve por ciento de las veces en trabajo real.

Clase 9: `WHERE`. El filtro que selecciona filas por condición: comparaciones, `BETWEEN`, `IN`, `IS NULL`. Es la clase que convierte consultas ordinarias en respuestas útiles, y donde vive la mayoría de los errores de principiante (comillas, nulos, límites de rango).

El patrón de las tres clases es siempre el mismo: primero pides (`SELECT`), luego afinas que columnas (columnas explícitas y alias), luego afinas que filas (`WHERE`). Cuando escribas una consulta, recorre mentalmente esas tres preguntas en orden y casi nunca te equivocarás.

Un día típico con `SELECT` y `WHERE`

Para cerrar la explicación, imagina un día normal de Mariana, analista de datos en una empresa de logística. Todo lo que hace en el día usa lo que aprendiste en esta unidad.

9:00. Mariana abre su computadora y lo primero que hace es explorar una tabla nueva que le pasaron: envios. Escribe `SELECT * FROM envios LIMIT 10` (el `LIMIT` lo verás en la Unidad 4; sirve para asomarse sin traer millones de filas). En segundos sabe que columnas existen y cómo se ven los datos.

9:20. Su jefe le pide "los envios retrasados de esta semana". Mariana traduce: columnas que importan (`id`, `cliente`, `fecha_prometida`, `fecha_entrega`, `ciudad`) y filtro (`fecha_entrega` mayor a `fecha_prometida`, y `fecha_prometida` dentro de la semana). Escribe el `SELECT` con columnas explícitas y el `WHERE` con comparaciones de fecha. Cinco minutos después el reporte está en el correo del jefe.

11:00. Auditoria: "cuáles envios no tienen ciudad de destino registrada?" `WHERE ciudad_destino IS NULL`. Salen doce. Mariana los reporta para corrección de captura. Sin saber manejar nulos, esos doce envios invisibles habrían contaminado todos los reportes por ciudad.

14:00. "Dame los envios con costó entre 500 y 2000 pesos para la revisión de precios." `WHERE costó BETWEEN 500 AND 2000`. "Y solo de las sucursales norte y sur." `AND sucursal IN ('Norte', 'Sur')`. Cada petición nueva es solo un filtro más en el `WHERE`.

16:30. Antes de irse, deja documentadas sus consultas con comentarios -- para que su compañero del turno vespertino entienda que hace cada una.

Ninguna de esas tareas requirió crear tablas, ni índices, ni nada avanzado. Solo `SELECT` y `WHERE`, usados con precisión. Ese es el ochenta por ciento del trabajo diario del que habla la introducción: no es una estadística abstracta, es la agenda de Mariana. Cuando domines esta unidad, podrás hacer su trabajo.

Checklist: antes de ejecutar tu consulta

Pasale está lista mental a cada consulta antes de ejecutarla. Con práctica se vuelve automática.

1. Las columnas del `SELECT` son exactamente las que necesito? (Nada de `SELECT *` por flojera.)
2. Los textos van entre comillas simples? Los números van sin comillas?
3. Si buscó nulos, use `IS NULL` y no `= NULL`?
4. En el `BETWEEN`, el límite menor va primero?
5. Las fechas están en formato AAAA-MM-DD entre comillas?

6. Si hay varias condiciones, cada una funciona por separado? (Pruebalas una por una si algo falla.)
7. La consulta tiene un comentario -- que explique su proposito?
8. Las palabras clave están en mayusculas y el código es legible?

Si respondes que si a todo, ejecutala con confianza.

Preguntas frecuentes

P: Si `SELECT *` es tan malo en producción, por qué todos los tutoriales lo usan? R: Porque en un tutorial lo importante es que veas el resultado completo y entiendas la idea rápido. En código real, donde la tabla puede tener cincuenta columnas y millones de filas, el asterisco cuesta rendimiento y claridad. Aprende con `*`, trabaja sin `*`.

P: Las palabras clave tienen que ir en mayusculas? R: No es obligatorio: `select * from empleados` funciona igual. Pero es la convención universal porque distingue a simple vista las instrucciones SQL de los nombres de tablas y columnas. Adoptala desde ahora y tu código se vera profesional.

P: Puedo usar comillas dobles para los textos? R: En MySQL las comillas dobles también funcionan para cadenas en la configuración por defecto, pero el estandar SQL reserva las comillas dobles para identificadores (nombres de tablas o columnas). Usa comillas simples para texto y evitaras sorpresas si algun día trabajas con otro motor.

P: Por qué mi consulta con `WHERE` no devuelve nada y no marca error? R: Las causas más comunes, en orden: comparaste `NULL` con `=` en lugar de `IS NULL`; el texto buscado no coincide exactamente (mayusculas, espacios, tildes); el rango `BETWEEN` tiene los limites al reves; o la condición es lógicamente imposible (`salario > 50000 AND salario < 10000`). Depura probando cada condición por separado.

P: Como buscó textos parecidos, no exactos? Por ejemplo, nombres que empiecen con "A". R: Con el operador `LIKE` y el comodin `%`: `WHERE nombre LIKE 'A%'`. `LIKE` se estudia a fondo más adelante, junto con `AND`, `OR` y `NOT` en la Unidad 4. Por ahora, quédate con que existe.

P: El punto y coma al final es obligatorio? R: Cuando ejecutas una sola consulta en la mayoría de las herramientas, no. Cuando ejecutas varias consultas juntas (un script), si: el `;` separa una consulta de la siguiente. Acostumbrate a ponerlo siempre y te evitaras errores de sintaxis.

P: Puedo seleccionar de una tabla que no existe? R: No: el motor respondera con error de "tabla desconocida". Si te pasa, revisa el nombre (mayusculas y minusculas importan en algunos sistemas) y que estes conectado a la base de datos correcta con `USE nombre_base`.

P: Qué pasa si dos columnas se llaman igual en el resultado? R: Con una sola tabla no suele pasar, pero si calculas dos veces lo mismo sin alias, verás dos columnas con el mismo encabezado confuso. Otro motivo para usar alias siempre que calcules algo.

Retos extra para practicar por tu cuenta

Si los diez ejercicios te supieron a poco, aquí tienes cinco retos más. Esta vez no hay solución escrita: solo la pregunta y cuantas filas debe devolver tu consulta. Inventa tu propia solución y verifica el conteo. Si tu conteo no coincide, revisa tu filtro condición por condición.

Reto A: muestra el nombre y el puesto de los empleados contratados antes del 2022 que ganen más de 20000. Debe devolver 2 filas. (Pista: fecha < '2022-01-01' AND salario > 20000.)

Reto B: muestra el nombre, la ciudad y el bono de los empleados de Guadalajara o Monterrey que SI tengan bono. Debe devolver 5 filas. (Pista: IN para las ciudades, IS NOT NULL para el bono.)

Reto C: muestra el nombre y el salario anual (con alias) de los tres empleados con el salario más bajo. Debe devolver 3 filas. (Pista: WHERE salario < 17000... verifica cuales caen en ese rango y ajusta si es necesario. No uses ORDER BY todavía; el orden exacto no importa.)

Reto D: muestra el nombre y la fecha de contratación de los empleados cuyo nombre empiece con las letras A a la F. Debe devolver 6 filas. (Pista: comparación alfabética con < 'G'. Cuidado: revisa el ejemplo 17 para recordar cómo funcionan estas comparaciones.)

Reto E: muestra todas las columnas de los empleados que no sean ni programadores ni analistas y que tengan bono mayor o igual a 1000. Debe devolver 4 filas. (Pista: NOT IN para los puestos, >= para el bono. Recuerda que las filas con bono NULL no cumplen >= 1000.)

Cuando puedas resolver los cinco sin dudar, esta unidad quedó atrás. En serio: es el punto donde SELECT y WHERE dejan de ser "temas de estudio" y se vuelven herramienta.

Resumen de la unidad

Llegaste al final de la unidad más importante del curso. Vamos a recapitular lo esencial, porque todo lo que sigue se construye sobre esto.

SELECT es el comando para pedir datos y es la habilidad número uno en SQL: alrededor del ochenta por ciento del trabajo diario con bases de datos consiste en consultar. Su estructura básica es SELECT (que columnas) FROM (de que tabla) WHERE (que filas). Las cláusulas se escriben en un orden fijo, pero el motor las ejecuta en orden lógico: primero FROM, después

WHERE y al final SELECT. Esa diferencia explica por qué no puedes usar en el WHERE un alias definido en el SELECT.

SELECT * pide todas las columnas y es perfecto para explorar una tabla nueva, pero en código de producción debes listar las columnas explícitamente: es más rápido, más robusto ante cambios en la tabla, más claro para quien lee el código y más seguro con datos sensibles. En el SELECT también puedes incluir expresiones calculadas y darles nombres legibles con alias usando AS.

WHERE es el filtro: se evalúa fila por fila y conserva solo las que cumplen la condición. Dominas ya los operadores de comparación (=, <>, >, <, >=, <=), que funcionan con números, textos y fechas; BETWEEN para rangos incluyendo los extremos; IN para listas de valores (y NOT IN para excluir); e IS NULL / IS NOT NULL para buscar valores nulos, porque NULL jamás se compara con =.

También aprendiste las reglas de formato que te acompañarán siempre: textos y fechas entre comillas simples, números sin comillas, palabras clave en mayúsculas, comentarios con -- para documentar tus consultas, y pedir solo las columnas que necesitas.

Los errores más comunes ya no te sorprenderán: olvidar comillas en textos, comparar NULL con =, poner los límites de BETWEEN al revés, usar el alias del SELECT dentro del WHERE y abusar de SELECT * en producción. Cada uno tiene una solución simple que ya practicaste.

En la práctica resolviste ocho ejercicios sobre la tabla empleados: filtrar por puesto, por rango de salario, por ciudad con IN, buscar nulos, combinar condiciones, filtrar por fechas, calcular columnas con alias y un reto que mezcló tres condiciones a la vez. Si puedes hacer eso sin mirar las soluciones, tienes el nivel de esta unidad.

Lo que sigue: en la Unidad 4 aprenderás a ordenar los resultados con ORDER BY, a limitar la cantidad de filas con LIMIT y a combinar condiciones con AND, OR y NOT, incluyendo como agrupar condiciones con paréntesis para que la lógica sea exactamente la que quieres. Después, en la Unidad 5, las funciones de agregación (COUNT, SUM, AVG, MIN, MAX) y GROUP BY te permitirán resumir miles de filas en una sola respuesta. Todo eso usa SELECT y WHERE como base: cada consulta avanzada que escribas en tu carrera llevará estas dos piezas dentro.

Un último consejo de Esteban: no avances a la Unidad 4 hasta que puedas escribir un SELECT con WHERE sin pensar demasiado. Abre tu MySQL, inventa preguntas sobre la tabla empleados ("quienes ganan más de 20000 y viven en Puebla?", "quienes no tienen bono y entraron antes de 2023?") y respondelas con consultas. Diez o quince consultas inventadas por ti valen más que cien ejemplos leídos. La fluidez se construye escribiendo. Nos vemos en la Unidad 4.

Lo que te llevas en la mochila

Si tuvieras que resumir esta unidad en una tarjeta que quepa en el bolsillo, diría esto:

SELECT elige columnas. FROM elige la tabla. WHERE elige filas. El motor ejecuta FROM, luego WHERE, luego SELECT. El asterisco es para explorar; las columnas explícitas son para trabajar. Los textos van entre comillas simples, los números sin comillas, las fechas en formato AAAA-MM-DD. NULL no se compara con igual: se busca con IS NULL. BETWEEN incluye los extremos, IN revisa listas, y cada condición se puede probar por separado cuando algo sale mal.

Con eso en la cabeza y las manos en el teclado, ya eres capaz de responder preguntas reales sobre datos reales. Las siguientes unidades te darán más herramientas (ordenar, agrupar, resumir, combinar tablas), pero ninguna será tan decisiva como esta: todo lo demás es SELECT y WHERE con superpoderes. Felicidades por llegar hasta aquí, y a seguir consultando.

Casos de la vida real: para que sirve todo esto

La teoría ya la tienes; ahora veamos cómo se usa SELECT y WHERE en situaciones de trabajo reales. Estos mini casos usan la misma tabla empleados y cada uno termina con la consulta que lo resuelve.

Caso 1: el reporte de nomina. El contador necesita la lista de empleados con su salario mensual y anual para calcular el presupuesto del próximo año. No necesita ver el id ni la fecha de contratación, solo datos de compensación.

```
-- Reporte de nomina: salario mensual y anual por empleado
SELECT nombre AS "Empleado", puesto AS "Puesto",
       salario AS "Mensual", salario * 12 AS "Anual"
FROM empleados;
```

Caso 2: la auditoria de bonos. Recursos humanos quiere saber a quienes todavía no se les asigne bono para regularizarlos antes del cierre del trimestre.

```
-- Empleados sin bono asignado: regularizar antes del cierre
SELECT nombre, puesto, fecha_contratacion
FROM empleados
WHERE bono IS NULL;
```

Caso 3: la búsqueda de candidatos a aumento. El gerente quiere ver a los programadores y programadoras que ganan menos de 22000 para evaluar un ajuste salarial.

```
-- Candidatos a aumento: programadores con salario menor a 22000
SELECT nombre, salario, ciudad
FROM empleados
WHERE puesto IN ('Programador', 'Programadora')
AND salario < 22000;
```

Caso 4: el filtro por antigüedad. Se acerca el aniversario de la empresa y quieren felicitar a quienes cumplen al menos dos años este 2026, es decir, contratados antes de 2024.

```
-- Empleados con al menos 2 años de antigüedad (contratados antes de 2024)
SELECT nombre, fecha_contratacion
FROM empleados
WHERE fecha_contratacion < '2024-01-01';
```

Caso 5: la limpieza de datos. El administrador sospecha que hay empleados registrados fuera de las tres ciudades donde la empresa tiene oficinas (Monterrey, Guadalajara, Puebla) y quiere detectarlos.

```
-- Posibles errores de captura: ciudades fuera de las oficinas conocidas
SELECT nombre, ciudad
FROM empleados
WHERE ciudad NOT IN ('Monterrey', 'Guadalajara', 'Puebla');
```

Con los datos de ejemplo este último devuelve cero filas, lo cual es una buena noticia: significa que no hay errores. En una tabla real, cada fila que aparezca es un dato sospechoso para revisar. Así se usa SQL para auditar calidad de datos: filtras lo que "no debería existir" y lo que salga es tu lista de trabajo.

Caso 6: el rango salarial por puesto. Para una revisión de equidad, se piden los analistas cuyo salario este entre 17000 y 19000.


```
-- Revision de equidad: analistas en el rango 17000-19000
SELECT nombre, salario
FROM empleados
WHERE puesto = 'Analista' AND salario BETWEEN 17000 AND 19000;
```

Fíjate en el patrón que se repite en todos los casos: primero defines que columnas necesita quien va a leer el resultado, después describes con precisión que filas le sirven. Ese es el oficio de consultar: traducir una necesidad de negocio ("quienes necesitan bono?") a un filtro exacto (WHERE bono IS NULL). Cuando domines esa traducción, serás la persona a la que todos le piden datos. Y eso, en cualquier empresa, es un superpoder.

Guía rápida de tipos de datos en el WHERE

Una duda frecuente: como escribo el valor según el tipo de la columna? Esta tabla te lo resume:

- Números enteros o decimales: sin comillas. WHERE salario > 20000. WHERE id = 5.
- Texto (VARCHAR, CHAR, TEXT): comillas simples. WHERE ciudad = 'Puebla'. WHERE nombre = 'Ana Lopez'.
- Fechas (DATE): comillas simples, formato AAAA-MM-DD. WHERE fecha_contratacion = '2023-06-01'.
- Fecha y hora (DATETIME): comillas simples, formato AAAA-MM-DD HH:MM:SS. WHERE creado = '2024-01-15 10:30:00'.
- Booleanos: según el motor; en MySQL se usan 1 y 0 o TRUE y FALSE. WHERE activo = 1.
- NULL: sin comillas, con IS NULL o IS NOT NULL. WHERE bono IS NULL.

Si comparas una columna numerica con un texto entre comillas ('20000'), MySQL hará la conversión y probablemente funcione, pero no te confíes: en otros motores puede fallar o impedir el uso de índices. Acostumbrate desde ahora a respetar el tipo de dato.

Más errores reales y como detectarlos

Error: el filtro devuelve más filas de las esperadas. Causa típica: comparación de texto inexacta. Si buscas WHERE puesto = 'programador' en minúsculas y los datos están en mayúscula inicial, no coincide nada (o al revés, según la configuración de cotejamiento). Solución: revisa con un SELECT DISTINCT puesto FROM empleados (DISTINCT lo verás pronto) cómo están escritos los valores reales antes de filtrar.

Error: el BETWEEN de fechas no trae el último día. Si la columna es DATETIME (fecha + hora) y escribes BETWEEN '2024-01-01' AND '2024-12-31', los registros del 31 de diciembre con hora distinta de medianoche quedan fuera, porque '2024-12-31' equivale a '2024-12-31 00:00:00'. Solución: usa < '2025-01-01' como límite superior o incluye la hora: BETWEEN '2024-01-01' AND '2024-12-31 23:59:59'. Con columnas DATE puras no hay problema.

Error: resultados duplicados que confunden. Si la tabla tiene filas repetidas (mismo empleado dos veces por un error de carga), tu conteo mental no cuadra. Solución temporal: SELECT DISTINCT para ver valores únicos mientras investigas el duplicado. La limpieza definitiva se hace con DELETE, que verás en la Unidad 7.

Error: la consulta tarda demasiado. Con tablas grandes, un WHERE sobre una columna sin índice puede tardar segundos o minutos. Solución: no te preocupes ahora; los índices se estudian más adelante. Pero graba la idea: filtrar por columnas indexadas (como las llaves primarias) es rapidísimo.

La técnica de depuración universal para filtros es la reducción: si una consulta con varias condiciones devuelve algo raro, prueba cada condición por separado. Ejecuta solo la primera, luego solo la segunda, y compara. En segundos encuentras cual condición es la culpable. Los profesionales depuran así todos los días.

Ejercicio 9: salarios altos fuera de la capital de operaciones

Pregunta: muestra el nombre, el puesto, el salario y la ciudad de los empleados que ganen 21000 o más y que NO vivan en Monterrey.

Pista: "21000 o más" es >=. "No vivan en Monterrey" es <> o NOT IN. Unelos con AND.

Solución:

```
SELECT nombre, puesto, salario, ciudad FROM empleados
WHERE salario >= 21000 AND ciudad <> 'Monterrey';
```

Explicación paso a paso:

1. Columnas: nombre, puesto, salario, ciudad.
2. Condición 1: salario >= 21000 (el 21000 se incluye).
3. Condición 2: ciudad <> 'Monterrey'.
4. Revisión fila por fila: Ana (22000 >= 21000? Si. Monterrey? Si, queda fuera por la segunda condición). Carla (25000? Si. Guadalajara? Distinta de Monterrey? Si. Dentro).

Elena (35000? Si. Monterrey? Fuera). Fabian (21000 >= 21000? Si, el limite incluido. Guadalajara distinta de Monterrey? Si. Dentro). Jorge (23000? Si. Puebla distinta de Monterrey? Si. Dentro).

5. Resultado esperado: tres filas: Carla Ruiz, Fabian Castro y Jorge Luna.

Lección: cuando combines condiciones, verifica cada una por separado sobre una fila de ejemplo antes de unir las. Así detectas si el problema está en el >=, en el <>, o en como las uniste.

Ejercicio 10: reporte final combinado

Pregunta: genera un reporte con el nombre del empleado, su salario anual (salario * 12, con alias "Anual") y su ciudad, solo para los empleados contratados a partir de 2022 (inclusive) que tengan bono asignado.

Pista: tres piezas: columna calculada con alias, filtro de fecha >= '2022-01-01', filtro de bono IS NOT NULL, todo unido con AND.

Solución:

```
-- Reporte: salario anual de empleados recientes con bono
SELECT nombre AS "Empleado", salario * 12 AS "Anual", ciudad AS "Ciudad"
FROM empleados
WHERE fecha_contratacion >= '2022-01-01' AND bono IS NOT NULL;
```

Explicación paso a paso:

1. Columnas: nombre con alias "Empleado", salario * 12 con alias "Anual", ciudad con alias "Ciudad".
2. Condición 1: fecha_contratacion >= '2022-01-01' (contratados desde 2022, inclusive).
3. Condición 2: bono IS NOT NULL.
4. Revisión fila por fila:
 - Ana Lopez: 2022-03-15 >= 2022-01-01? Si. Bono 3000? Si. Dentro. Anual: 264000.
 - Bruno Diaz: 2023-06-01? Si. Bono NULL? Fuera.
 - Carla Ruiz: 2021-11-20 >= 2022-01-01? No. Fuera.
 - Diego Torres: 2023-01-10? Si. Bono 2000? Si. Dentro. Anual: 216000.
 - Elena Mora: 2020-05-25? No. Fuera.
 - Fabian Castro: 2024-02-14? Si. Bono NULL? Fuera.

- Gabriela Soto: 2023-09-30? Si. Bono 1500? Si. Dentro. Anual: 210000.
 - Hector Vega: 2024-07-08? Si. Bono NULL? Fuera.
 - Irene Pardo: 2022-12-05? Si. Bono 1000? Si. Dentro. Anual: 198000.
 - Jorge Luna: 2021-08-17? No. Fuera.
5. Resultado esperado: cuatro filas (Ana, Diego, Gabriela, Irene) con encabezados "Empleado", "Anual" y "Ciudad".

Este ejercicio junta todo lo de la unidad: columnas calculadas, alias legibles, filtro de fecha, filtro de nulos y condiciones combinadas. Si lo resolviste, la unidad esta dominada.

Glosario de la unidad

- Cláusula: cada bloque de una consulta SQL (SELECT, FROM, WHERE, etc.).
- Orden sintactico: el orden en que se escriben las clausulas.
- Orden lógico de ejecución: el orden en que el motor procesa las clausulas (FROM, WHERE, SELECT...).
- SELECT *: pide todas las columnas; útil para explorar, desaconsejado en producción.
- Alias: nombre temporal para una columna del resultado, con AS.
- WHERE: cláusula que filtra filas según una condición, evaluada fila por fila.
- Operador de comparación: =, <>, >, <, >=, <=.
- BETWEEN: filtro de rango que incluye los extremos.
- IN: verifica si un valor está dentro de una lista; NOT IN hace lo contrario.
- NULL: ausencia de valor; no es cero ni texto vacío.
- IS NULL / IS NOT NULL: la única forma correcta de filtrar nulos.
- Condición: expresión del WHERE que resulta verdadera, falsa o desconocida por fila.
- Comentario: texto ignorado por el motor (-- o /* */) para documentar consultas.

UNIDAD 4: FILTRADO AVANZADO Y ORDENAMIENTO

Bienvenido a la Unidad 4 del Curso Completo de SQL de Lucas EGA Academy.

En las unidades anteriores aprendiste a crear bases de datos, a llenar tablas con datos y a recuperarlos con consultas basicas. SELECT, WHERE, INSERT y DELETE ya son parte de tu caja de herramientas. Y eso está muy bien. Pero aquí viene la verdad que nadie te dice al principio: una consulta sin filtro ni orden es solo ruido.

Imagina que entras a un almacén gigante lleno de cajas y pides "todo lo que hay". El encargado te tira una montaña de cajas encima sin ningún criterio. Eso es una consulta SELECT sin WHERE, sin ORDER BY y sin LIMIT. Técnicamente te dio lo que pediste. Prácticamente no te sirvió de nada.

Piensa ahora en el caso contrario: el encargado te pregunta que necesitas. "Las cajas de electrónica que llegaron esta semana, ordenadas por fecha, y solo las primeras veinte porque no me caben más en el camión". Eso es una consulta profesional. Misma bodega, misma información disponible, pero una pregunta bien hecha te devuelve exactamente lo útil. La diferencia entre un principiante y alguien que domina SQL no está en conocer más comandos raros: está en hacer mejores preguntas. Esta unidad te enseña a hacerlas.

Los datos, por sí solos, no valen nada. Lo que vale es encontrar el dato correcto, en el momento correcto, ordenado de la forma correcta. En esta unidad aprenderás exactamente eso: como dominar tus consultas para que te devuelvan solo lo que necesitas, ordenado como lo necesitas, y en la cantidad que necesitas.

La unidad tiene tres clases. La clase 10 cubre los operadores: comparación y lógicos AND, OR y NOT. Con ellos podrás combinar condiciones y preguntar cosas complejas como "quiero los productos de la categoría electrodomesticos que cuesten menos de 500 pesos y tengan stock disponible". La clase 11 cubre ORDER BY: como ordenar tus resultados de forma ascendente o descendente, usando varias columnas a la vez. Y la clase 12 cubre LIMIT: como cortar tus resultados para obtener solo los primeros N registros o para implementar paginación, eso que ves cuando una página web muestra "10 resultados por página".

Estas tres clases juntas cambian por completo tu nivel como programador de bases de datos. Después de esta unidad, tus consultas dejarán de ser "dame todo" para convertirse en "dame exactamente esto, ordenado así, y solo estos".

Cómo usar esta unidad: lee primero la explicación detallada completa, con calma, porque ahí están los conceptos que todo lo demás usa (tablas de verdad, precedencia, NULL). Después trabaja los ejemplos paso a paso ejecutando cada consulta tú mismo: no te saltes ninguno, porque cada uno muestra un patrón que volveras a usar. Luego intenta los ejercicios sin mirar las soluciones; si te atorras más de diez minutos, mira la solución, entiendela y vuelve a escribirla de memoria. Cierra con el resumen y la lista de verificación.

Antes de empezar, recuerda que el autor de este curso es Esteban, fundador de Lucas EGA Academy (lucasegaacademy.com), y que todo el material esta pensado para que lo practiques de verdad. Abre tu gestor de base de datos, crea la tabla de productos que usaremos en los ejemplos y escribe cada consulta con tus propias manos. Leer SQL no es aprender SQL. Escribir SQL si lo es.

Vamos a ello.

Explicación detallada

Operadores de comparación

Los operadores de comparación son los símbolos que te permiten comparar un valor con otro dentro de una consulta. Son la base del filtrado en SQL, porque la cláusula WHERE depende de ellos para decidir que filas pasan el filtro y cuales no.

SQL tiene seis operadores de comparación basicos. Vamos a verlos uno por uno.

El operador de igualdad es = (un solo signo de igual, no dos como en otros lenguajes). Sirve para pedir exactamente un valor. Por ejemplo, WHERE categoría = 'Ropa' trae solo las filas cuya categoría sea exactamente "Ropa". Ojo: en SQL se usa un solo igual. Si pones == te marcara error en la mayoría de los gestores.

El operador de diferencia es <> (algunos gestores también aceptan !=, pero <> es el estandar SQL y funciona en todos lados). Sirve para excluir un valor. WHERE categoría <> 'Ropa' trae todo menos la ropa.

Los operadores de orden son < (menor que), > (mayor que), <= (menor o igual que) y >= (mayor o igual que). Funcionan con números, con fechas y también con texto (donde comparan por orden alfabetico según el cotejamiento de la base de datos). Por ejemplo, WHERE precio > 100 trae los productos que cuestan más de 100. WHERE fecha >= '2026-01-01' trae los registros de este año en adelante.

Hay además tres operadores de comparación especiales que merecen su propia atención: BETWEEN, IN y LIKE.

BETWEEN sirve para rangos. WHERE precio BETWEEN 100 AND 500 es lo mismo que WHERE precio >= 100 AND precio <= 500. Incluye los extremos: el 100 y el 500 si entran. Es muy útil para rangos de fechas: WHERE fecha BETWEEN '2026-01-01' AND '2026-12-31' trae todo el año 2026.

IN sirve para listas. WHERE categoría IN ('Ropa', 'Calzado', 'Accesorios') es lo mismo que WHERE categoría = 'Ropa' OR categoría = 'Calzado' OR categoría = 'Accesorios', pero mucho más corto y legible. Cuando la lista crece, IN se vuelve tu mejor amigo.

LIKE sirve para patrones de texto, usando dos comodines: % (cualquier cantidad de caracteres) y _ (exactamente un carácter). WHERE nombre LIKE 'Cam%' trae todo lo que empiece con "Cam": "Camisa", "Cámara", "Camión". WHERE nombre LIKE '%pro' trae todo lo que termine en "pro". WHERE nombre LIKE '_osa' trae palabras de cuatro letras que terminen en "osa", como "rosa" o "cosa". LIKE es sensible a mayúsculas y minúsculas dependiendo del cotejamiento; en MySQL con cotejamiento latin1 o utf8 general, normalmente no distingue, pero en otros gestores sí. Si necesitas comparar sin importar mayúsculas, una práctica común es convertir ambos lados: WHERE UPPER(nombre) = UPPER('camisa').

Un detalle importante: los operadores de comparación devuelven tres resultados posibles, no dos. TRUE (verdadero), FALSE (falso) y UNKNOWN (desconocido). Ese tercer valor aparece cuando comparas con NULL, y es la fuente de muchos errores silenciosos. Lo veremos a fondo en la sección de NULL.

Lógica booleana: AND, OR y NOT

Cuando una sola condición no basta, SQL te da los operadores lógicos AND, OR y NOT para combinar condiciones. Esta es la herramienta que convierte filtros simples en filtros de verdad.

AND exige que todas las condiciones sean verdaderas. WHERE precio > 100 AND stock > 0 trae productos caros Y con existencias. Si alguna condición falla, la fila no pasa.

OR exige que al menos una condición sea verdadera. WHERE categoría = 'Ropa' OR categoría = 'Calzado' trae ropa O calzado. Basta con que una se cumpla.

NOT invierte el resultado. WHERE NOT (precio > 100) es lo mismo que WHERE precio <= 100. WHERE categoría NOT IN ('Ropa', 'Calzado') excluye esas dos categorías. NOT se usa menos, pero cuando lo necesitas, no hay sustituto.

Ahora lo serio: las tablas de verdad. Son la forma matematica de entender exactamente cómo se comporta cada operador. Memorizalas o tenlas a la mano; te salvaran de muchos dolores de cabeza.

Tabla de verdad de AND:

- TRUE AND TRUE = TRUE
- TRUE AND FALSE = FALSE
- FALSE AND TRUE = FALSE
- FALSE AND FALSE = FALSE

En resumen: AND solo es verdadero cuando todo es verdadero.

Tabla de verdad de OR:

- TRUE OR TRUE = TRUE
- TRUE OR FALSE = TRUE
- FALSE OR TRUE = TRUE
- FALSE OR FALSE = FALSE

En resumen: OR solo es falso cuando todo es falso.

Tabla de verdad de NOT:

- NOT TRUE = FALSE
- NOT FALSE = TRUE

Simple: NOT invierte.

Hasta aquí todo es lógica de primaria. El problema empieza con el tercer valor: UNKNOWN, que aparece con NULL. Las tablas se extienden así.

AND con UNKNOWN:

- TRUE AND UNKNOWN = UNKNOWN
- FALSE AND UNKNOWN = FALSE
- UNKNOWN AND UNKNOWN = UNKNOWN

Fíjate en algo interesante: FALSE AND UNKNOWN da FALSE, no UNKNOWN. Porque AND solo necesita un falso para ser falso, y ya lo tiene.

OR con UNKNOWN:

- TRUE OR UNKNOWN = TRUE
- FALSE OR UNKNOWN = UNKNOWN
- UNKNOWN OR UNKNOWN = UNKNOWN

Y NOT UNKNOWN = UNKNOWN.

Esto importa porque la cláusula WHERE solo deja pasar las filas donde la condición completa da TRUE. Las que dan FALSE o UNKNOWN se quedan fuera. Así que si filtras WHERE precio > 100 y hay un producto con precio NULL, ese producto NO aparece. No porque sea falso, sino porque el resultado es UNKNOWN y WHERE solo acepta TRUE. Este es uno de los errores más comunes en SQL y lo desarrollaremos en la sección de NULL.

Precedencia de operadores y el uso de paréntesis

Cuando mezclas AND, OR y NOT en una misma condición, el orden en que se evalúan importa muchísimo. SQL tiene reglas de precedencia: primero se evalúa NOT, después AND, y al final OR. Es parecido a las matemáticas, donde primero va la multiplicación y después la suma.

Mira este ejemplo:

WHERE categoría = 'Ropa' OR categoría = 'Calzado' AND precio < 100

Por precedencia, primero se evalúa el AND: (categoría = 'Calzado' AND precio < 100). Así que la consulta trae: toda la ropa (sin importar el precio) más el calzado barato. Probablemente eso NO es lo que querías. Lo más probable es que quisieras: (ropa o calzado) que además sea barato.

La solución: paréntesis. Los paréntesis siempre ganan a la precedencia y dejan claro lo que quieres.

WHERE (categoría = 'Ropa' OR categoría = 'Calzado') AND precio < 100

Esta versión sí trae ropa barata y calzado barato. Misma consulta, mismos operadores, resultado completamente distinto. Solo cambiaron los paréntesis.

La regla de oro: cuando mezcles AND con OR, usa paréntesis siempre. Aunque creas que la precedencia te favorece. Aunque estes seguro. Los paréntesis no cuestan nada, hacen tu intención explícita y evitan que otro programador (o tú mismo dentro de seis meses) malinterprete la consulta.

Otro caso clásico:

WHERE NOT categoría = 'Ropa' AND precio > 100

NOT tiene la precedencia más alta, así que se evalúa primero: (NOT categoría = 'Ropa') AND precio > 100. Trae lo que no es ropa y cuesta más de 100. Si lo que querías era negar todo, necesitas paréntesis: WHERE NOT (categoría = 'Ropa' AND precio > 100).

Los paréntesis también sirven para documentar. Una condición larga como WHERE (a = 1 OR a = 2) AND (b = 3 OR b = 4) se lee mucho mejor que su versión sin paréntesis, aunque el resultado sea el mismo. La legibilidad es una característica, no un lujo.

El problema de NULL en las comparaciones

NULL es el valor más traicionero de SQL. No es cero. No es texto vacío. No es "nada". NULL significa "desconocido" o "no aplica". Y con lo desconocido no se puede comparar de forma normal.

La regla fundamental: cualquier comparación con NULL da UNKNOWN, nunca TRUE ni FALSE. Veamos:

- NULL = NULL da UNKNOWN (no TRUE)
- NULL <> 1 da UNKNOWN
- NULL > 100 da UNKNOWN
- NULL = 'texto' da UNKNOWN

Esto tiene consecuencias prácticas muy concretas. Supongamos que tienes una tabla de productos y algunos tienen el precio en NULL porque aun no se les asigna precio. Haces esta consulta:

```
SELECT * FROM productos WHERE precio <> 200;
```

Esperarías obtener todos los productos excepto los que cuestan 200. Pero los productos con precio NULL no aparecen. Para ellos, NULL <> 200 da UNKNOWN, y WHERE solo deja pasar TRUE. El resultado te miente por omisión: crees que viste "todo menos los de 200", pero en realidad viste "todo menos los de 200 y menos los sin precio".

La forma correcta de preguntar por NULL es con IS NULL e IS NOT NULL. Son operadores especiales diseñados para eso.

```
SELECT * FROM productos WHERE precio IS NULL;
```

Esta consulta sí trae los productos sin precio. Y su contraparte:

```
SELECT * FROM productos WHERE precio IS NOT NULL;
```

Trae los que si tienen precio. Nunca uses = NULL ni <> NULL: no funcionan como esperas. Algunos gestores incluso te lanzan advertencia, pero la mayoría simplemente te devuelve un resultado vacío o incompleto sin avisarte.

Otro caso: contar. La función COUNT(*) cuenta todas las filas, incluyendo las que tienen NULL. Pero COUNT(precio) solo cuenta las filas donde precio NO es NULL. Si tu tabla tiene 100 productos y 10 no tienen precio, COUNT(*) da 100 y COUNT(precio) da 90. No es un error: es el diseño. Las funciones de agregación ignoran los NULL.

Y en ORDER BY, los NULL también tienen comportamiento especial: en la mayoría de los gestores aparecen al principio en orden ascendente y al final en descendente, aunque esto varia. Lo veremos en la sección de ORDER BY.

Consejo práctico: cuando disenes una tabla, piensa si cada columna debería permitir NULL o no. Si un producto siempre debe tener precio, declara la columna como NOT NULL y te ahorras toda esta clase de problemas. NULL es necesario a veces, pero cada NULL que permites es una comparación futura que puede salir mal.

ORDER BY a fondo

ORDER BY es la cláusula que ordena tus resultados. Va al final de la consulta (después de WHERE) y le dice a la base de datos en que orden entregarte las filas.

La sintaxis básica es:

ORDER BY columna ASC

ASC significa ascendente: de menor a mayor en números, de A a Z en texto, de la fecha más vieja a la más nueva. Es el orden por defecto: si no pones nada, SQL asume ASC.

ORDER BY columna DESC

DESC significa descendente: de mayor a menor, de Z a A, de la fecha más nueva a la más vieja.

Hasta aquí es fácil. Lo interesante empieza cuando ordenas por varias columnas.

Ordenar por múltiples columnas

Puedes pasarle a ORDER BY una lista de columnas separadas por comas. La base de datos ordena por la primera, y cuando hay empates en la primera, desempatan con la segunda, y así sucesivamente.

ORDER BY categoría ASC, precio DESC

Esto ordena primero por categoría (de la A a la Z) y, dentro de cada categoría, ordena los productos del más caro al más barato. Cada columna puede tener su propia dirección: la primera ascendente y la segunda descendente, o al revés, o las dos iguales. Tú decides.

Piensa en un ejemplo concreto: una tienda quiere mostrar sus productos agrupados por categoría en orden alfabético, y dentro de cada categoría quiere que aparezcan primero los más caros. Con una sola columna no se puede. Con dos, es una línea.

El orden de las columnas en la lista importa muchísimo. ORDER BY categoría, precio no es lo mismo que ORDER BY precio, categoría. En el primero, la categoría manda y el precio solo desempata. En el segundo, el precio manda y la categoría solo desempata. Es como ordenar una biblioteca: primero por estante y luego por autor no es lo mismo que primero por autor y luego por estante.

Puedes ordenar por tantas columnas como necesites. Tres, cuatro, las que sean. Cada una desempata a la anterior.

Ordenar por alias

Cuando tu SELECT crea una columna calculada con un alias, puedes ordenar por ese alias directamente.

```
SELECT nombre, precio, precio * 1.16 AS precio_con_iva FROM productos ORDER BY
precio_con_iva DESC;
```

Esto ordena por el precio con IVA incluido, del más caro al más barato. Sin el alias tendrías que repetir la expresión completa en el ORDER BY: ORDER BY precio * 1.16 DESC. Funciona igual, pero el alias es más limpio y menos propenso a errores (si cambias la fórmula en el SELECT y se te olvida cambiarla en el ORDER BY, tendrás un bug silencioso).

Nota técnica: esto funciona porque en la mayoría de los gestores el ORDER BY se evalúa después del SELECT, así que ya conoce los alias. Hay excepciones en consultas muy complejas, pero para el uso normal puedes confiar en ello.

Ordenar por posición

También puedes ordenar por el número de la columna en el SELECT. ORDER BY 1 ordena por la primera columna del SELECT, ORDER BY 2 por la segunda, y así.

```
SELECT nombre, precio FROM productos ORDER BY 2 DESC;
```

Esto ordena por precio descendente, porque precio es la segunda columna del SELECT.

Es una sintaxis valida y corta, pero usala con cuidado: si alguien cambia el orden de las columnas en el SELECT, el ORDER BY cambia de significado sin que nadie se de cuenta. Para código que va a vivir mucho tiempo o que van a leer otras personas, el nombre de la columna o el alias es mejor. La posición es práctica para consultas rápidas que escribes y desechas.

Orden de textos: mayusculas, minusculas y tildes

Ordenar texto parece simple hasta que aparecen las mayusculas y las tildes. El orden alfabetico en SQL depende del cotejamiento (collation) de tu base de datos, que es la regla que define cómo se comparan los caracteres.

En muchos cotejamientos, las mayusculas van antes que las minusculas porque así están en la tabla de caracteres: "Zebra" iria antes que "árbol". En otros cotejamientos más modernos, la comparación ignora mayusculas y el orden es el que esperarías: "árbol" antes que "Zebra".

Con las tildes pasa algo parecido. En algunos cotejamientos, "camión" y "camión" se consideran equivalentes para ordenar. En otros, la tilde cambia el orden y "camión" puede aparecer lejos de "camión".

Consejo práctico: si el orden de tus textos se ve raro, revisa el cotejamiento de tu tabla o de tu base de datos. Para la mayoría de las aplicaciones en español, un cotejamiento como utf8mb4_spanish_ci o utf8mb4_unicode_ci ordena de forma sensata con nuestras tildes y enes. No necesitas memorizar los nombres: solo recuerda que el orden del texto no es magico, es una configuración, y que se puede ajustar.

Si necesitas un orden totalmente controlado sin depender del cotejamiento, puedes forzar la comparación en minusculas: ORDER BY LOWER(nombre). Esto normaliza las mayusculas, aunque las tildes seguiran dependiendo del cotejamiento.

NULLs al inicio o al final

Cuando ordenas una columna que tiene NULLs, esos valores tienen que ir en algun lugar. La mayoría de los gestores los ponen al principio en orden ASC y al final en orden DESC. Es decir, los tratan como si fueran "menos que todo".

Pero no todos los gestores se comportan igual, y depender del comportamiento por defecto es riesgoso. La forma explícita y portable es usar NULLS FIRST o NULLS LAST (disponibles en PostgreSQL, Oracle y otros):

`ORDER BY precio ASC NULLS LAST;`

Esto ordena de menor a mayor pero manda los productos sin precio al final, que suele ser lo que quieres mostrarle a un usuario: primero lo que tiene precio, al final lo indefinido.

En MySQL, que no tiene `NULLS FIRST/LAST` directamente, puedes lograr lo mismo con un truco: `ORDER BY precio IS NULL, precio ASC`. La expresión `precio IS NULL` vale 0 para los que tienen precio y 1 para los `NULL`, así que ordenando por ella primero, los `NULL` quedan al final.

La lección: si tu columna puede tener `NULLs` y el orden importa para lo que muestras, decide explícitamente donde van. No dejes que el gestor decida por ti.

LIMIT y OFFSET: cortar resultados

`LIMIT` es la cláusula que limita cuantos registros te devuelve la consulta. Es de las más usadas en aplicaciones reales, porque casi nunca quieres "todos los registros": quieres los primeros 10, los 5 más caros, o una página de resultados.

La sintaxis básica:

`SELECT * FROM productos LIMIT 10;`

Te devuelve como máximo 10 filas. Si la tabla tiene menos de 10, te devuelve las que haya. `LIMIT` nunca falla por pedir de más: simplemente te da lo que hay.

`OFFSET` salta un número de filas antes de empezar a contar el `LIMIT`:

`SELECT * FROM productos LIMIT 10 OFFSET 20;`

Salta las primeras 20 filas y te devuelve las siguientes 10 (de la 21 a la 30). `OFFSET` sin `LIMIT` también es válido en algunos gestores, pero lo normal es usarlos juntos.

En MySQL existe una sintaxis alternativa: `LIMIT 20, 10` significa lo mismo que `LIMIT 10 OFFSET 20` (primero el desplazamiento, luego la cantidad). Es válida, pero la forma con `OFFSET` es más clara y más estándar. Te recomiendo usar `LIMIT` con `OFFSET` explícito.

Top N: los más caros, los más nuevos, los más vendidos

La combinación de `ORDER BY` con `LIMIT` es una de las más poderosas de SQL. Te permite responder preguntas de tipo "top N": los 5 productos más caros, los 10 clientes más recientes, los 3 vendedores con más ventas.

`SELECT nombre, precio FROM productos ORDER BY precio DESC LIMIT 5;`

Esto ordena todos los productos del más caro al más barato y luego corta en los primeros 5. Resultado: los 5 más caros. Sin el ORDER BY, el LIMIT te daría 5 productos cualesquiera, que no sirven para nada. El orden define que significa "los primeros".

Otros ejemplos de top N:

Los 10 productos con menos stock (para saber que resurtir):

```
SELECT nombre, stock FROM productos ORDER BY stock ASC LIMIT 10;
```

Los últimos 5 productos agregados (suponiendo que id crece con el tiempo):

```
SELECT nombre FROM productos ORDER BY id DESC LIMIT 5;
```

Fíjate en el patrón: ORDER BY define el criterio, DESC o ASC define la dirección, y LIMIT define cuantos. Tres piezas, infinitas preguntas respondidas.

Paginación: página 1, 2, 3

La paginación es lo que ves en cualquier tienda en línea o buscador: los resultados divididos en páginas de, digamos, 10 elementos. La página 1 muestra los primeros 10, la página 2 los siguientes 10, y así. En SQL se implementa con LIMIT y OFFSET.

La fórmula es simple. Si cada página tiene N elementos y quieres la página P:

$\text{OFFSET} = (P - 1) * N$ $\text{LIMIT} = N$

Página 1 (N = 10): LIMIT 10 OFFSET 0. Las primeras 10 filas.

```
SELECT * FROM productos ORDER BY id LIMIT 10 OFFSET 0;
```

Página 2: LIMIT 10 OFFSET 10. Salta 10, toma las siguientes 10.

```
SELECT * FROM productos ORDER BY id LIMIT 10 OFFSET 10;
```

Página 3: LIMIT 10 OFFSET 20.

```
SELECT * FROM productos ORDER BY id LIMIT 10 OFFSET 20;
```

El patrón del OFFSET es 0, 10, 20, 30... múltiplos del tamaño de página. En tu aplicación, el código calcula el OFFSET a partir del número de página que pide el usuario y arma la consulta.

Un detalle crucial: la paginación solo funciona bien si el ORDER BY es estable y único. Si ordenas por una columna con muchos valores repetidos (como categoría), el orden entre los empatados es arbitrario y un producto podría aparecer en la página 1 y también en la página 2,

o desaparecer entre página y página. La solución es agregar una columna única como segundo criterio, normalmente el id:

`ORDER BY categoría ASC, id ASC`

Así el orden es totalmente determinístico: a igualdad de categoría, manda el id, y dos filas nunca empatan en id. A esto se le llama paginación estable, y es la diferencia entre una paginación que funciona y una que muestra resultados duplicados o perdidos.

La trampa de LIMIT sin ORDER BY

Este es uno de los errores conceptuales más comunes: usar LIMIT sin ORDER BY y asumir que obtienes "los primeros" en algún orden lógico.

`SELECT * FROM productos LIMIT 5;`

Que te devuelve esto? Cinco filas. Cuales? Las que el motor de base de datos encuentre primero según su plan de ejecución interno. Podrían ser las de menor id hoy, y otras mañana después de que se borraron e insertaron filas. El orden sin ORDER BY no está garantizado: es no determinístico.

Esto significa que si ejecutas la misma consulta dos veces, podrías obtener resultados distintos. En desarrollo, con tablas pequeñas y sin cambios, suele "parecer" estable y te confías. En producción, con miles de filas y actividad constante, te da sorpresas.

La regla es simple: LIMIT sin ORDER BY es válido sintácticamente pero semánticamente vago. Si te importan cuáles filas obtienes (y casi siempre te importan), pon ORDER BY. Si de verdad te da igual y solo quieres una muestra rápida para explorar, entonces está bien, pero hazlo conscientemente.

La trampa gemela: ORDER BY sin LIMIT en tablas gigantes. Si tu tabla tiene un millón de filas y pides ORDER BY precio sin LIMIT, la base de datos tiene que ordenar el millón completo antes de devolverte la primera fila. Eso es lento y consume memoria. Si solo necesitas los primeros 10, el LIMIT le permite al motor optimizar: no necesita ordenar todo, solo encontrar los 10 extremos. Siempre que puedas, acompaña tu ORDER BY con un LIMIT.

Errores comunes

Vamos a repasar los errores que más se repiten con estos temas, para que los reconozcas cuando aparezcan.

Olvidar los paréntesis con OR. Ya lo vimos, pero vale repetirlo porque es el error número uno: WHERE a = 1 OR b = 2 AND c = 3 no significa lo que parece. El AND se evalúa primero. Paréntesis siempre que mezcles AND y OR.

Comparar con NULL usando = o <>. WHERE precio = NULL no trae nada, nunca, en ningún gestor. Usa IS NULL. Y WHERE precio <> NULL tampoco excluye los NULL como esperarías: también da UNKNOWN. Usa IS NOT NULL.

Paginación inestable. Pagar con ORDER BY sobre una columna no única (sin desempatar con el id) produce duplicados y huecos entre páginas. Agrega siempre una columna única al final del ORDER BY cuando pagues.

LIMIT sin ORDER BY esperando "los primeros". No hay "primeros" sin orden. Si necesitas los más recientes, ordena por fecha descendente. Si necesitas los de menor id, ordena por id. El LIMIT solo corta; el ORDER BY decide que se corta.

Confundir BETWEEN con rangos exclusivos. BETWEEN incluye los extremos. WHERE precio BETWEEN 100 AND 500 incluye el 100 y el 500. Si necesitas excluir un extremo, usa > o < explícitos en lugar de BETWEEN.

LIKE sin comodines. WHERE nombre LIKE 'Camisa' es lo mismo que WHERE nombre = 'Camisa', pero más lento y más confuso. Si no usas % ni _, no necesitas LIKE: usa =.

Olvidar que NOT IN con NULL en la lista da vacío. WHERE categoría NOT IN ('Ropa', NULL) no trae nada, porque compararse con NULL da UNKNOWN. Si tu lista puede contener NULL, filtralos antes o usa NOT EXISTS. Este es un error sutil que hasta programadores con experiencia cometen.

Buenas prácticas y rendimiento

Algunas reglas para que tus consultas con filtros y ordenamiento sean rápidas y sanas.

Filtra antes de ordenar. La base de datos primero aplica el WHERE y luego ordena lo que queda (ese es el orden lógico de ejecución). Aprovechalo: un buen WHERE que reduzca mucho las filas hace que el ORDER BY tenga poco trabajo. Si puedes filtrar agresivo, hazlo.

Evita ORDER BY en tablas gigantes sin índice. Ordenar es caro: la base de datos tiene que comparar todas las filas entre sí. Si ordenas frecuentemente por una columna (por ejemplo, fecha de creación para mostrar "lo más reciente"), considera crear un índice sobre esa columna. Un índice es como el índice alfabético de un libro: permite encontrar y ordenar sin recorrer todo. No abuses de los índices (cada índice cuesta en las inserciones), pero para columnas de ordenamiento frecuente son oro.

No ordenes lo que no vas a mostrar ordenado. Si tu aplicación va a reordenar los datos en memoria de todos modos, o si el orden no le importa al usuario, no pongas ORDER BY. Cada ORDER BY tiene un costo; no lo pagues si no lo necesitas.

Usa LIMIT siempre que solo necesites una muestra. "Dame los 10 más recientes" con LIMIT 10 es muchísimo más barato que traer miles y cortar en la aplicación. Deja que la base de datos haga el trabajo pesado: para eso está.

Cuidado con OFFSET grande. La paginación con OFFSET tiene un problema: para darte la página 1000 con OFFSET 10000, la base de datos tiene que recorrer y descartar 10000 filas. En paginaciones profundas (miles de páginas) esto se vuelve lento. La alternativa profesional es la paginación por cursor (keyset pagination): en lugar de saltar N filas, pides "las siguientes 10 después del id X" con WHERE id > X ORDER BY id LIMIT 10. Es mucho más eficiente porque usa el índice directamente. Para las primeras páginas no importa, pero si tu aplicación página profundo, investiga esta técnica.

Escribe condiciones que puedan usar índices. Una condición como WHERE YEAR(fecha) = 2026 obliga a la base de datos a calcular YEAR() en cada fila y no puede usar un índice sobre fecha. En cambio, WHERE fecha BETWEEN '2026-01-01' AND '2026-12-31' si puede usarlo. La regla general: evita aplicar funciones a la columna en el WHERE; aplica la lógica al valor con el que comparas.

Lo mismo aplica al ORDER BY: ordenar por una expresión calculada (ORDER BY precio * 1.16) impide usar un índice para el ordenamiento, mientras que ORDER BY precio si puede aprovecharlo. Si necesitas ordenar por un calculo con frecuencia, considera si vale la pena guardar el valor calculado en una columna (o usar una columna generada, que varios gestores soportan).

Y la más importante: prueba tus consultas con datos reales. Una consulta que vuela con 100 filas de prueba puede arrastrarse con 10 millones en producción. Si tu tabla va a crecer, diseña pensando en ese crecimiento desde el día uno. Cuando algo se ponga lento, la herramienta para diagnosticarlo es el plan de ejecución (EXPLAIN en MySQL y PostgreSQL): te muestra en que orden el motor ejecuta tu consulta, que índices usa y donde gasta el tiempo. No necesitas dominarlo hoy, pero tenlo en el radar: es el siguiente nivel después de esta unidad.

El orden lógico de ejecución: por qué WHERE va antes que ORDER BY

Escribes la consulta en este orden: SELECT, FROM, WHERE, ORDER BY, LIMIT. Pero la base de datos la ejecuta en un orden distinto. Entender ese orden te explica por qué algunas cosas funcionan y otras no.

El orden lógico de ejecución es: primero FROM (de que tabla sacó las filas), después WHERE (que filas pasan el filtro), después SELECT (que columnas muestro y que calculos hago), después ORDER BY (en que orden las entrego) y al final LIMIT (cuantas entrego).

Esto explica varias cosas que ya viste. Explica por qué puedes ordenar por un alias del SELECT: cuando se ejecuta el ORDER BY, el SELECT ya se calculo y el alias ya existe. Explica por qué NO puedes usar ese alias en el WHERE: cuando se ejecuta el WHERE, el SELECT todavía no existe. Si intentas WHERE precio_con_iva > 100 usando un alias definido en el SELECT, te marcara error en la mayoría de los gestores. Tienes que repetir la expresión: WHERE precio * 1.16 > 100.

También explica por qué el WHERE filtra antes de ordenar: primero reduces las filas y luego ordenas las que quedan. Por eso un buen WHERE hace más rápido al ORDER BY. Y explica por qué el LIMIT va al final: primero se filtra, luego se ordena todo lo filtrado, y al final se corta. (En la práctica, los motores optimizan y no siempre ordenan todo antes de cortar, pero el resultado lógico es el mismo.)

Hay una excepción famosa que debes conocer: en consultas con GROUP BY y HAVING, el HAVING se ejecuta después del agrupamiento y si puede usar alias del SELECT en algunos gestores. Pero para el nivel de esta unidad, quédate con el orden básico: FROM, WHERE, SELECT, ORDER BY, LIMIT. Cuando una consulta te de un error raro del tipo "columna desconocida", revisa este orden: casi siempre estas usando algo antes de que exista.

Combinando todo: el patrón profesional

Ya viste cada pieza por separado. Ahora veamos cómo se combinan en consultas reales, porque en la práctica nunca usas una sola: las usas todas juntas siguiendo un patrón.

El patrón es: filtra con WHERE (usando comparación, lógica booleana y NULL bien manejados), ordena con ORDER BY (una o varias columnas, dirección explícita), y corta con LIMIT (y OFFSET si páginas).

Ejemplo: una tienda en línea quiere su sección "Lo más vendido en electrónica con descuento". La consulta seria algo así:

```
SELECT nombre, precio, descuento
FROM productos
WHERE categoria = 'Electronica' AND descuento > 0 AND stock > 0
ORDER BY descuento DESC, precio ASC
LIMIT 12;
```

Lee la consulta como una frase: "de los productos, quédate con los de electrónica que tengan descuento y stock; ordenalos por descuento de mayor a menor y, a igual descuento, por precio de menor a mayor; dame los primeros 12". Cada cláusula hace su trabajo y el resultado es exactamente lo que la página necesita mostrar. Sin traer de más, sin ordenar en la aplicación, sin filtrar en memoria.

Otro patrón comun: reportes. "Los 10 clientes con más compras este ano" seria WHERE fecha BETWEEN ... AND ... para el ano, ORDER BY total DESC, LIMIT 10. "Productos por resurtir" seria WHERE stock < 10, ORDER BY stock ASC. Una vez que interiorizas el patrón filtro-orden-corte, empiezas a traducir preguntas de negocio a SQL casi automáticamente.

Un último consejo sobre legibilidad: escribe cada cláusula en su propia línea, como en los ejemplos. Las consultas de una sola línea larguísima son difíciles de leer y de depurar. El SQL no se queja de los saltos de línea: usalos. Tu yo del futuro te lo agradecerá.

NOT: negaciones bien usadas

NOT parece el operador más simple, pero tiene sus matices. La regla de oro: NOT se aplica a la condición que tiene inmediatamente a la derecha, salvo que uses paréntesis para ampliar su alcance.

WHERE NOT precio > 100 niega solo la comparación precio > 100, y equivale a WHERE precio <= 100. Hasta aquí todo bien. Pero WHERE NOT precio > 100 AND stock > 0 se lee como (NOT precio > 100) AND (stock > 0), porque NOT tiene más precedencia que AND. Si lo que querías era negar toda la combinación, necesitas paréntesis: WHERE NOT (precio > 100 AND stock > 0).

Las formas más comunes de NOT en la práctica son tres: NOT IN para excluir listas (WHERE categoría NOT IN ('Ropa', 'Calzado')), NOT LIKE para excluir patrones (WHERE nombre NOT LIKE 'Cam%'), y NOT BETWEEN para excluir rangos (WHERE precio NOT BETWEEN 100 AND 500). Las tres son azúcar sintáctico: NOT IN equivale a negar cada igualdad con AND, pero son mucho más legibles.

Cuidado con dos trampas. Primera: NOT IN con NULL en la lista devuelve vacío, como ya vimos. Segunda: negar una condición con NULL no "rescata" esos NULL. WHERE NOT (precio > 100) no trae los productos con precio NULL, porque NOT UNKNOWN sigue siendo UNKNOWN. Si quieres incluir los NULL en una negación, tienes que pedirlos explícito: WHERE NOT (precio > 100) OR precio IS NULL. Parece redundante, pero es la única forma correcta.

Un consejo de estilo: a veces es más claro escribir la condición positiva que negar. WHERE precio <= 100 se lee mejor que WHERE NOT precio > 100. Reserva NOT para cuando la negación sea lo natural: excluir listas, patrones o rangos con NOT IN, NOT LIKE y NOT BETWEEN.

Preguntas frecuentes y trampas típicas

Para cerrar la parte teorica, repasemos las preguntas que más se hacen los estudiantes con estos temas, porque ahí es donde suelen atorarse.

Puedo usar AND y OR sin paréntesis si "se entiende"? Tecnicamente si, la consulta corre. Pero "se entiende" es subjetivo: lo que tu entiendes hoy, otro programador puede entenderlo distinto mañana, y la base de datos lo entiende según la precedencia, no según tu intención. Los paréntesis son gratis y eliminan la ambigüedad. Usalos.

Por qué mi consulta con NOT IN no trae nada? Casi seguro tu lista o tu subconsulta incluye un NULL. Recuerda: compararse con NULL da UNKNOWN, y NOT IN exige que la comparación sea claramente falsa para cada elemento. Un solo NULL en la lista contamina todo el resultado. Soluciones: filtra los NULL de la lista, o usa NOT EXISTS, que maneja los NULL correctamente.

Por qué COUNT(*) y COUNT(columna) me dan distinto? Porque COUNT(*) cuenta filas y COUNT(columna) cuenta valores no nulos de esa columna. Si hay NULLs, difieren. No es un bug: es el comportamiento definido. Si quieres contar filas, usa COUNT(*).

Mi paginación muestra repetidos, que hago? Tu ORDER BY no es deterministico: hay empates y el motor los ordena como quiere en cada página. Agrega una columna única al final, normalmente el id: ORDER BY fecha DESC, id DESC. Con eso cada fila tiene una posición fija y la paginación queda estable.

Es lo mismo WHERE a = 1 AND b = 2 que WHERE b = 2 AND a = 1? En resultado, si: AND es conmutativo. En rendimiento, normalmente también, porque el optimizador de la base de datos reordena las condiciones como le convenga usando estadísticas. Escribe las condiciones en el orden que sea más legible para ti; el motor se encarga del resto.

Debo poner punto y coma al final? En la mayoría de los gestores el punto y coma marca el fin de una sentencia y es obligatorio cuando ejecutas varias seguidas (como en un script). En consultas sueltas interactivas muchos lo hacen opcional. Acostumbrate a ponerlo siempre: te evita errores cuando copies tus consultas a un script.

Ejemplos paso a paso

En esta sección vamos a trabajar con una sola tabla, "productos", para que todos los ejemplos sean consistentes y puedas seguirlos en tu propio gestor. Primero la creamos y la llenamos.

```
CREATE TABLE productos (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    categoria VARCHAR(50),  
    precio DECIMAL(10,2),  
    stock INT  
);
```

```
INSERT INTO productos (id, nombre, categoria, precio, stock) VALUES  
(1, 'Camisa azul', 'Ropa', 299.00, 50),  
(2, 'Pantalon negro', 'Ropa', 499.00, 30),  
(3, 'Tenis deportivos', 'Calzado', 899.00, 20),  
(4, 'Sandalias', 'Calzado', 249.00, 0),  
(5, 'Cinturon de piel', 'Accesorios', 199.00, 40),  
(6, 'Gorra roja', 'Accesorios', 149.00, 60),  
(7, 'Chamarra de mezclilla', 'Ropa', 799.00, 15),  
(8, 'Botas de trabajo', 'Calzado', 1099.00, 10),  
(9, 'Calcetines pack x3', 'Ropa', 99.00, 200),  
(10, 'Mochila urbana', 'Accesorios', 549.00, 25),  
(11, 'Vestido floreado', 'Ropa', 649.00, 12),  
(12, 'Zapatos de vestir', 'Calzado', 999.00, 8),  
(13, 'Bufanda de lana', 'Accesorios', NULL, 35),  
(14, 'Playera blanca', 'Ropa', 129.00, 0),  
(15, 'Sombrero de palma', 'Accesorios', 279.00, 18);
```

Fíjate que el producto 13, "Bufanda de lana", tiene precio NULL: aun no se le asigna precio. Lo dejamos así a proposito para ver el comportamiento de NULL en los ejemplos.

Ejemplo 1: filtro simple con operador de comparación.

Queremos los productos que cuestan más de 500 pesos.

```
SELECT nombre, precio  
FROM productos  
WHERE precio > 500;
```

Explicación: el WHERE evalua `precio > 500` en cada fila. Las que dan TRUE pasan. Resultado: Tenis deportivos (899), Chamarra de mezclilla (799), Botas de trabajo (1099), Mochila urbana

(549), Vestido floreado (649) y Zapatos de vestir (999). Son 6 filas. La bufanda (precio NULL) no aparece porque NULL > 500 da UNKNOWN.

Ejemplo 2: combinación con AND.

Queremos ropa que cueste menos de 500 y tenga stock disponible.

```
SELECT nombre, precio, stock
FROM productos
WHERE categoria = 'Ropa' AND precio < 500 AND stock > 0;
```

Explicación: las tres condiciones deben ser TRUE. Revisemos la ropa: Camisa azul (299, stock 50) si pasa. Pantalón negro (499, stock 30) si pasa. Chamarra (799) no pasa por precio. Calcetines (99, stock 200) si pasa. Vestido (649) no pasa por precio. Playera blanca (129) tiene precio y categoría bien, pero stock = 0, así que no pasa. Resultado: Camisa azul, Pantalón negro y Calcetines pack x3. Tres filas.

Ejemplo 3: combinación con OR.

Queremos todo el calzado o todos los accesorios.

```
SELECT nombre, categoria
FROM productos
WHERE categoria = 'Calzado' OR categoria = 'Accesorios';
```

Explicación: basta con que una condición sea TRUE. Resultado: los 4 de calzado (Tenis, Sandalias, Botas, Zapatos) más los 5 de accesorios (Cinturón, Gorra, Mochila, Bufanda, Sombrero). Nueve filas. La versión corta con IN sería WHERE categoría IN ('Calzado', 'Accesorios') y da lo mismo.

Ejemplo 4: el caso que exige paréntesis.

Queremos ropa o calzado, pero solo lo barato (menos de 300). La versión MAL escrita sería:

```
SELECT nombre, categoria, precio
FROM productos
WHERE categoria = 'Ropa' OR categoria = 'Calzado' AND precio < 300;
```

Explicación del problema: por precedencia, el AND se evalúa primero. Esta consulta trae TODA la ropa (sin importar precio) más el calzado barato. Resultado real: Camisa azul, Pantalón negro, Chamarra, Calcetines, Vestido, Playera (toda la ropa) más Sandalias (calzado de 249). Siete filas, incluyendo la chamarra de 799 que no queríamos.

La versión CORRECTA con paréntesis:

```
SELECT nombre, categoria, precio
FROM productos
WHERE (categoria = 'Ropa' OR categoria = 'Calzado') AND precio < 300;
```

Explicación: primero se resuelve el paréntesis (ropa o calzado) y luego se exige precio < 300. Resultado: Camisa azul (299), Calcetines (99), Playera blanca (129) y Sandalias (249). Cuatro filas. Compara los dos resultados: la única diferencia son los paréntesis, y el resultado cambia por completo.

Ejemplo 5: NOT para excluir.

Queremos todo excepto accesorios, y solo con stock disponible.

```
SELECT nombre, categoria, stock
FROM productos
WHERE categoria NOT IN ('Accesorios') AND stock > 0;
```

Explicación: NOT IN excluye la categoría, y el AND exige stock mayor a cero. Resultado: Camisa azul, Pantalón negro, Tenis deportivos, Chamarra de mezclilla, Botas de trabajo, Calcetines, Vestido floreado y Zapatos de vestir. Ocho filas. Las sandalias y la playera blanca quedan fuera por stock = 0; todos los accesorios quedan fuera por la categoría.

Ejemplo 6: BETWEEN para rangos de precio.

Queremos productos entre 200 y 600 pesos, incluyendo los extremos.

```
SELECT nombre, precio
FROM productos
WHERE precio BETWEEN 200 AND 600;
```

Explicación: BETWEEN incluye el 200 y el 600. Resultado: Camisa azul (299), Pantalón negro (499), Mochila urbana (549), Sombrero de palma (279). Cuatro filas. La bufanda (NULL) no aparece: NULL BETWEEN ... da UNKNOWN.

Ejemplo 7: LIKE para buscar por texto.

Queremos productos cuyo nombre empiece con "C".


```
SELECT nombre
FROM productos
WHERE nombre LIKE 'C%';
```

Explicación: el % representa "cualquier cosa después". Resultado: Camisa azul, Cinturon de piel, Chamarra de mezclilla, Calcetines pack x3. Cuatro filas.

Y si queremos los que contienen "de" en cualquier parte del nombre:

```
SELECT nombre
FROM productos
WHERE nombre LIKE '%de%';
```

Resultado: Cinturon de piel, Chamarra de mezclilla, Zapatos de vestir, Bufanda de lana, Sombrero de palma. Cinco filas.

Ejemplo 8: IS NULL para encontrar los sin precio.

```
SELECT nombre, categoria
FROM productos
WHERE precio IS NULL;
```

Explicación: la única forma correcta de preguntar por NULL. Resultado: Bufanda de lana. Una fila. Si hubieras usado WHERE precio = NULL, el resultado habría sido cero filas, sin error y sin aviso.

Ejemplo 9: ORDER BY simple descendente.

Queremos ver los productos del más caro al más barato.

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

Explicación: ordena por precio de mayor a menor. Resultado: Botas de trabajo (1099), Zapatos de vestir (999), Tenis deportivos (899), Chamarra (799), Vestido (649), Mochila (549), Pantalón (499), Camisa (299), Sombrero (279), Sandalias (249), Cinturón (199), Gorra (149), Playera (129), Calcetines (99) y al final la Bufanda (NULL, porque en la mayoría de los gestores los NULL van al final en DESC... espera, en DESC los NULL van al final? En la mayoría van al principio en ASC y al final en DESC. En MySQL, los NULL se consideran los valores más bajos: van primero en ASC y últimos en DESC. Así que en este ejemplo la bufanda queda al final).

Ejemplo 10: ORDER BY múltiple.

Queremos los productos agrupados por categoría (A a Z) y dentro de cada categoría del más caro al más barato.

```
SELECT categoria, nombre, precio
FROM productos
ORDER BY categoria ASC, precio DESC;
```

Explicación: primero ordena por categoría ascendente: Accesorios, Calzado, Ropa. Dentro de Accesorios, ordena por precio descendente: Mochila (549), Sombrero (279), Cinturón (199), Gorra (149), Bufanda (NULL al final). Dentro de Calzado: Botas (1099), Zapatos (999), Tenis (899), Sandalias (249). Dentro de Ropa: Chamarra (799), Vestido (649), Pantalón (499), Camisa (299), Playera (129), Calcetines (99). Cada grupo queda ordenado por precio de mayor a menor.

Ejemplo 11: top 5 más caros.

```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 5;
```

Explicación: ordena del más caro al más barato y corta en 5. Resultado: Botas de trabajo (1099), Zapatos de vestir (999), Tenis deportivos (899), Chamarra de mezclilla (799), Vestido floreado (649). Sin el ORDER BY, el LIMIT 5 te habría dado 5 filas arbitrarias.

Ejemplo 12: paginación, página 1.

Supongamos páginas de 5 productos, ordenados por id para que la paginación sea estable.

```
SELECT id, nombre
FROM productos
ORDER BY id ASC
LIMIT 5 OFFSET 0;
```

Explicación: OFFSET 0 no salta nada, LIMIT 5 toma las primeras 5. Resultado: ids 1 al 5 (Camisa azul, Pantalón negro, Tenis deportivos, Sandalias, Cinturón de piel).

Ejemplo 13: paginación, página 2.

```
SELECT id, nombre
FROM productos
ORDER BY id ASC
LIMIT 5 OFFSET 5;
```

Explicación: salta las primeras 5 y toma las siguientes 5. Resultado: ids 6 al 10 (Gorra roja, Chamarra de mezclilla, Botas de trabajo, Calcetines pack x3, Mochila urbana).

Ejemplo 14: paginación, página 3.

```
SELECT id, nombre
FROM productos
ORDER BY id ASC
LIMIT 5 OFFSET 10;
```

Explicación: salta 10, toma 5. Resultado: ids 11 al 15 (Vestido floreado, Zapatos de vestir, Bufanda de lana, Playera blanca, Sombrero de palma). Entre las tres páginas cubrimos los 15 productos sin repetir ni saltar ninguno. Esa es una paginación estable: ORDER BY id garantiza el orden determinístico.

Ejemplo 15: combinación completa, el ejemplo "de la vida real".

Una tienda quiere mostrar en su página principal los 3 productos de Ropa con stock disponible, ordenados del más barato al más caro, para una sección de "ofertas en ropa".

```
SELECT nombre, precio, stock
FROM productos
WHERE categoria = 'Ropa' AND stock > 0
ORDER BY precio ASC
LIMIT 3;
```

Explicación paso a paso: el WHERE filtra ropa con stock (Camisa 299, Pantalón 499, Chamarra 799, Calcetines 99, Vestido 649; la playera queda fuera por stock 0). El ORDER BY los ordena por precio ascendente (Calcetines 99, Camisa 299, Pantalón 499, Vestido 649, Chamarra 799). El LIMIT corta en 3. Resultado final: Calcetines pack x3 (99), Camisa azul (299), Pantalón negro (499). Filtro, orden y corte trabajando juntos: eso es una consulta profesional.

Ejemplo 16: ordenar por alias (precio con IVA).

Queremos ver los productos ordenados por su precio con IVA del 16% incluido, del más caro al más barato, mostrando el precio ya calculado.

```
SELECT nombre, precio * 1.16 AS precio_con_iva
FROM productos
WHERE precio IS NOT NULL
ORDER BY precio_con_iva DESC
LIMIT 5;
```

Explicación: el SELECT calcula precio_con_iva y le pone alias. El WHERE excluye la bufanda (precio NULL) para que el cálculo no sea NULL. El ORDER BY usa el alias directamente, sin repetir la fórmula. El LIMIT corta en 5. Resultado: Botas de trabajo (1274.84), Zapatos de vestir (1158.84), Tenis deportivos (1042.84), Chamarra de mezclilla (926.84), Vestido floreado (752.84). Fíjate como el alias hace la consulta legible: se entiende a la primera que estamos ordenando por el precio con IVA.

Ejemplo 17: paginación estable con desempate.

Supongamos que agregamos dos productos más con el mismo precio de 299 para forzar empates: una "Camisa verde" (id 16, 299) y unos "Guantes" (id 17, 299). Si paginamos ordenando solo por precio, los tres productos de 299 (Camisa azul id 1, Camisa verde id 16, Guantes id 17) podrían aparecer en distinto orden en cada página, o repetirse. La solución es desempatar con el id:

```
SELECT id, nombre, precio
FROM productos
ORDER BY precio ASC, id ASC
LIMIT 5 OFFSET 0;
```

Explicación: el primer criterio ordena por precio; cuando hay empate (los tres de 299), el id decide el orden de forma determinística: id 1, luego 16, luego 17. Cada fila tiene una posición fija y única en el orden total. Página 1: los 5 más baratos en orden totalmente predecible.

Página 2 (OFFSET 5): los siguientes 5, sin repetir ninguno de la página 1 y sin saltarse ninguno. Esta es la paginación estable que debes usar en producción: ORDER BY con columna única al final, siempre.

Práctica

Usa la misma tabla "productos" de la sección de ejemplos (15 filas, con la bufanda en precio NULL). Resuelve cada ejercicio por tu cuenta antes de ver la solución. La gracia está en intentarlo.

Ejercicio 1: filtro con AND y OR combinados

Escribe una consulta que traiga los productos de categoría 'Accesorios' o 'Calzado' que cuesten más de 200 pesos. El resultado debe incluir nombre, categoría y precio.

Solución:

```
SELECT nombre, categoria, precio
FROM productos
WHERE (categoria = 'Accesorios' OR categoria = 'Calzado') AND precio > 200;
```

Explicación paso a paso. Paso 1: el paréntesis resuelve primero la parte del OR: nos quedamos con accesorios y calzado (9 filas). Paso 2: el AND exige precio > 200 sobre esas 9. Revisemos: Tenis deportivos (899) si. Sandalias (249) si. Cinturon (199) no. Gorra (149) no. Botas (1099) si. Mochila (549) si. Zapatos (999) si. Bufanda (NULL) no, porque NULL > 200 es UNKNOWN. Sombrero (279) si. Resultado: Tenis deportivos, Sandalias, Botas de trabajo, Mochila urbana, Zapatos de vestir y Sombrero de palma. Seis filas.

Punto clave del ejercicio: sin los paréntesis, la consulta habría traído todos los accesorios (sin importar precio) más el calzado caro. Los paréntesis cambian el significado.

Ejercicio 2: el top 3 más barato con stock

Escribe una consulta que traiga los 3 productos más baratos que tengan stock disponible (stock mayor a cero). Muestra nombre, precio y stock, ordenados del más barato al más caro.

Solución:

```
SELECT nombre, precio, stock
FROM productos
WHERE stock > 0
ORDER BY precio ASC
LIMIT 3;
```

Explicación paso a paso. Paso 1: el WHERE deja fuera a Sandalias y Playera blanca (stock = 0). Quedan 13 filas. Paso 2: ORDER BY precio ASC los ordena de menor a mayor: Calcetines (99), Gorra (149), Cinturon (199), Sombrero (279), Camisa (299)... y la bufanda (NULL) queda primera o última según el gestor, pero ojo: tiene stock 35, así que paso el WHERE. En MySQL los NULL van primero en ASC, así que la bufanda aparecería antes que los calcetines. Paso 3: LIMIT 3 corta. Resultado en MySQL: Bufanda de lana (NULL), Calcetines pack x3 (99), Gorra roja (149).

Este ejercicio tiene trampa a propósito: el NULL se cuela en el top 3. Si no quieres productos sin precio en tu "top más barato", agrega AND precio IS NOT NULL al WHERE. La consulta corregida:

```
SELECT nombre, precio, stock
FROM productos
WHERE stock > 0 AND precio IS NOT NULL
ORDER BY precio ASC
LIMIT 3;
```

Resultado corregido: Calcetines pack x3 (99), Gorra roja (149), Cinturon de piel (199). Moraleja: cuando trabajes con top N sobre columnas que pueden tener NULL, decide explícitamente que hacer con ellos.

Ejercicio 3: paginación de ofertas

La tienda muestra 4 productos por página, ordenados por precio descendente. Escribe las consultas para la página 1 y la página 2. Muestra nombre y precio.

Solución página 1:

```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 4 OFFSET 0;
```

Solución página 2:

```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 4 OFFSET 4;
```

Explicación paso a paso. El orden por precio descendente queda: Botas (1099), Zapatos (999), Tenis (899), Chamarra (799), Vestido (649), Mochila (549), Pantalón (499), Camisa (299), Sombrero (279), Sandalias (249), Cinturón (199), Gorra (149), Playera (129), Calcetines (99), Bufanda (NULL al final en DESC en MySQL). Página 1 (OFFSET 0): Botas, Zapatos, Tenis, Chamarra. Página 2 (OFFSET 4): Vestido, Mochila, Pantalón, Camisa.

Nota importante: esta paginación ordena por precio, que no es una columna única. Si dos productos costaran exactamente lo mismo, su orden relativo sería arbitrario y podrían repetirse o saltarse entre páginas. Para una paginación estable en producción, agregaría el id como desempate: ORDER BY precio DESC, id ASC. En este ejercicio no hay precios repetidos, así que funciona, pero en datos reales el desempate es obligatorio.

Ejercicio 4: búsqueda con LIKE y NOT

Escribe una consulta que traiga los productos cuyo nombre contenga la palabra "de", pero excluyendo la categoría 'Ropa'. Muestra nombre y categoría, ordenados por nombre de la A a la Z.

Solución:

```
SELECT nombre, categoria
FROM productos
WHERE nombre LIKE '%de%' AND categoria <> 'Ropa';
```

Explicación paso a paso. Paso 1: LIKE '%de%' encuentra los nombres que contienen "de" en cualquier posición: Cinturón de piel, Chamarra de mezclilla, Zapatos de vestir, Bufanda de lana, Sombrero de palma. Cinco filas. Paso 2: el AND con categoría <> 'Ropa' elimina la Chamarra de mezclilla (es ropa). Quedan 4. Paso 3: ORDER BY nombre ASC los ordena alfabéticamente: Bufanda de lana, Cinturón de piel, Sombrero de palma, Zapatos de vestir.

Variante con NOT: en lugar de categoría <> 'Ropa' podrías escribir AND NOT categoría = 'Ropa', o AND categoría NOT IN ('Ropa'). Las tres formas son equivalentes. Elige la que te parezca más legible.

Ejercicio 5: rango de precios con BETWEEN y orden múltiple

Escribe una consulta que traiga los productos con precio entre 150 y 600 pesos (incluyendo los extremos), ordenados por categoría ascendente y, dentro de cada categoría, por precio descendente. Muestra categoría, nombre y precio.

Solución:

```
SELECT categoria, nombre, precio
FROM productos
WHERE precio BETWEEN 150 AND 600
ORDER BY categoria ASC, precio DESC;
```

Explicación paso a paso. Paso 1: BETWEEN 150 AND 600 filtra: Camisa (299), Pantalón (499), Sandalias (249), Cinturón (199), Mochila (549), Sombrero (279). Seis filas. La gorra (149) queda fuera por un peso: BETWEEN incluye el 150 exacto, pero 149 es menor. La bufanda (NULL) queda fuera. Paso 2: ORDER BY categoría ASC agrupa: Accesorios primero, luego Calzado, luego Ropa. Paso 3: dentro de cada grupo, precio DESC. Resultado final: Accesorios - Mochila urbana (549), Sombrero de palma (279), Cinturón de piel (199); Calzado - Sandalias (249); Ropa - Pantalón negro (499), Camisa azul (299).

Ejercicio 6: detectar el error (depuración)

La siguiente consulta tiene un error lógico. Encuéntralo, explica que resultado da realmente y escribe la versión corregida.

```
SELECT nombre, precio
FROM productos
WHERE categoria = 'Ropa' AND precio < 200 OR stock = 0;
```

Análisis del error. Por precedencia, el AND se evalúa antes que el OR. La consulta real es: (categoría = 'Ropa' AND precio < 200) OR (stock = 0). Eso trae: la ropa barata (Calcetines 99, Playera 129... la camisa cuesta 299, no entra) MÁS todos los productos con stock cero sin importar la categoría (Sandalias de calzado y Playera blanca de ropa).

Si la intención era "ropa que sea barata o que este agotada", la consulta necesita paréntesis:

```
SELECT nombre, precio
FROM productos
WHERE categoria = 'Ropa' AND (precio < 200 OR stock = 0);
```


Explicación de la versión corregida: primero el paréntesis (precio < 200 OR stock = 0), luego el AND exige que sea ropa. Resultado: Calcetines pack x3 (99, barato), Playera blanca (129, barata y agotada). La camisa azul (299, con stock) no entra. Las sandalias (agotadas pero de calzado) no entran. Dos filas.

Moraleja del ejercicio: cuando veas AND y OR mezclados sin paréntesis, desconfía. Lee la consulta aplicando la precedencia (AND primero) y pregunta: es eso lo que realmente se quiere? Nueve de cada diez veces, faltan paréntesis.

Ejercicio 7: reto integrador (todo junto)

Una tienda quiere armar su página de "Ofertas de la semana": necesita los productos que cumplan TODO lo siguiente: que sean de categoría 'Ropa' o 'Calzado', que cuesten entre 100 y 700 pesos, que tengan stock disponible, y que su nombre NO empiece con la letra 'P'. El resultado debe mostrar nombre, categoría, precio y stock, ordenado por precio ascendente, mostrando solo los primeros 4 (la página 1 de las ofertas, con 4 por página). Escribe la consulta completa.

Solución:

```
SELECT nombre, categoria, precio, stock
FROM productos
WHERE (categoria = 'Ropa' OR categoria = 'Calzado')
    AND precio BETWEEN 100 AND 700
    AND stock > 0
    AND nombre NOT LIKE 'P%'
ORDER BY precio ASC
LIMIT 4 OFFSET 0;
```

Explicación paso a paso. Paso 1: el paréntesis (categoría = 'Ropa' OR categoría = 'Calzado') selecciona 10 filas (6 de ropa + 4 de calzado). Paso 2: AND precio BETWEEN 100 AND 700 elimina la chamarara (799), las botas (1099), los tenis (899) y los zapatos (999). Quedan: Camisa azul (299), Pantalón negro (499), Sandalias (249), Calcetines (99)... espera, calcetines cuesta 99 y el rango empieza en 100, así que queda fuera. También queda fuera la playera blanca (129)? No: 129 si esta entre 100 y 700, así que pasa este filtro. Lista hasta aquí: Camisa azul, Pantalón negro, Sandalias, Playera blanca, Vestido floreado (649). Paso 3: AND stock > 0 elimina sandalias (0) y playera blanca (0). Quedan: Camisa azul, Pantalón negro, Vestido floreado. Paso 4: AND nombre NOT LIKE 'P%' elimina los que empiecen con P: Pantalón negro y Playera blanca (ya eliminada). Quedan: Camisa azul y Vestido floreado. Paso 5: ORDER BY precio ASC los ordena: Camisa azul (299), Vestido floreado (649). Paso 6: LIMIT 4 OFFSET 0 toma hasta 4: trae las 2 filas.

Resultado final: Camisa azul (Ropa, 299, stock 50) y Vestido floreado (Ropa, 649, stock 12). Solo 2 filas aunque el LIMIT pedía 4: LIMIT es un máximo, no un mínimo. Si hay menos filas, trae las que haya sin error.

Este ejercicio junta todo lo de la unidad: OR con paréntesis, BETWEEN, AND múltiple, NOT LIKE, ORDER BY y LIMIT con OFFSET. Si lo resolviste sin ver la solución, ya dominas el filtrado avanzado y el ordenamiento. Si te costó, repasa los ejemplos 4 y 15: ahí están los mismos patrones.

Resumen de la unidad

En esta unidad aprendiste a filtrar, ordenar y cortar tus consultas. Repasemos lo esencial.

Los operadores de comparación (=, <>, <, >, <=, >=) comparan valores en el WHERE. Los especiales BETWEEN (rangos), IN (listas) y LIKE (patrones de texto con % y _) cubren los casos más comunes de filtrado. IS NULL e IS NOT NULL son la única forma correcta de preguntar por valores nulos.

Los operadores lógicos AND, OR y NOT combinan condiciones. AND exige que todo sea verdadero, OR exige que al menos una lo sea, NOT invierte. La precedencia es NOT, luego AND, luego OR. Cuando mezcles AND con OR, usa paréntesis siempre: hacen tu intención explícita y evitan el error más común de SQL.

NULL es traicionero: cualquier comparación con NULL da UNKNOWN, y el WHERE solo deja pasar TRUE. Por eso los NULL desaparecen silenciosamente de tus filtros si no los manejas con IS NULL o IS NOT NULL. Diseña tus tablas pensando en esto y usa NOT NULL donde el valor siempre deba existir.

ORDER BY ordena tus resultados. ASC es ascendente (por defecto) y DESC es descendente. Puedes ordenar por varias columnas: la primera manda y las siguientes desempatan. Puedes ordenar por alias, y por posición aunque es menos recomendable. El orden de los textos depende del cotejamiento (cuidado con mayúsculas y tildes). Decide explícitamente donde van los NULL con NULLS FIRST/LAST o su equivalente.

LIMIT corta los resultados: top N con ORDER BY + LIMIT, paginación con LIMIT + OFFSET donde el OFFSET de la página P es $(P - 1) * N$. Para paginación estable, ordena siempre con una columna única como desempate (normalmente el id). Nunca uses LIMIT sin ORDER BY si te importa que filas obtienes: el orden sin ORDER BY no está garantizado.

En rendimiento: filtra agresivo con WHERE antes de ordenar, evita ORDER BY sobre tablas gigantes sin índice, no ordenes lo que no necesitas ordenado, usa LIMIT para no traer de más, y cuidado con OFFSET muy grandes (considera paginación por cursor para páginas profundas). Escribe condiciones que puedan usar índices: evita funciones sobre la columna en el WHERE.

Con estas tres clases tu nivel dio un salto real. Ya no pides "todo": pides exactamente lo que necesitas, ordenado como lo necesitas, en la cantidad que necesitas. En la siguiente unidad seguiremos construyendo sobre esta base.

Lista de verificación: si puedes hacer todo esto sin dudar, dominas la unidad.

- Explicar con tus palabras la diferencia entre AND y OR, y decir cuál se evalúa primero.
- Escribir una condición con AND y OR mezclados usando paréntesis correctamente.
- Decir que devuelve NULL = NULL y por qué, y escribir la forma correcta de buscar NULLs.
- Ordenar por dos columnas con direcciones distintas y explicar que columna desempatan a cuál.
- Obtener el top 5 de cualquier criterio con ORDER BY + LIMIT.
- Escribir las consultas de la página 1, 2 y 3 de una paginación de 10 por página.
- Explicar por qué LIMIT sin ORDER BY es una mala idea.
- Detectar en una consulta ajena la falta de paréntesis con OR y la paginación inestable.

Si algún punto te cuesta, vuelve a su sección y practícalo con la tabla productos hasta que salga natural. La fluidez en SQL, como en cualquier idioma, viene de la repetición con atención, no de leer muchas páginas.

Sigue practicando en lucasegaacademy.com. Nos vemos en la Unidad 5.

- Esteban

UNIDAD 5: BÚSQUEDA DE PATRONES Y RESÚMENES

Introducción

Hasta aquí has aprendido a guardar datos, a leerlos y a filtrarlos con precisión quirúrgica. Sabes crear tablas, insertar filas, pedir columnas específicas y decirle a la base de datos cosas como "dame los clientes de Mexico" o "muestrame las ventas mayores a mil pesos". Eso ya te pone por encima del noventa por ciento de las personas que trabajan con datos a diario.

Pero hay un salto que aun no hemos dado, y es el salto que separa a quien consulta una base de datos de quien de verdad la domina. Ese salto es pasar de ver filas sueltas a ver patrones, y de contar registros uno por uno a resumir miles de filas en un solo número que dice algo importante.

Pongamos un ejemplo con la vida real. Imagina que trabajas en una tienda en línea y tu jefe llega con tres preguntas. Primera: "Necesito la lista de todos los clientes cuyo correo sea de Gmail, porque vamos a mandarles una promoción especial". Segunda: "Cuántos productos vendimos el mes pasado y cuanto dinero entro en total". Tercera: "Dime que categorías de producto nos dejan más dinero, pero solo me interesan las que superen los cincuenta mil pesos".

Fíjate en algo: ninguna de esas tres preguntas se responde leyendo fila por fila. La primera requiere buscar un patrón dentro de un texto (todos los correos que terminen en "@gmail.com"). La segunda requiere resumir miles de ventas en dos números (el conteo y la suma). La tercera requiere agrupar las ventas por categoría, calcular el total de cada grupo y después quedarse solo con los grupos que pasen un umbral. Son exactamente las cuatro herramientas que vas a dominar en esta unidad.

La primera herramienta es LIKE. En la unidad anterior aprendiste el operador igual (=), que es perfecto cuando sabes exactamente lo que buscas. Pero en la vida real casi nunca sabes el texto exacto. Sabes que el correo "termina en gmail.com", que el nombre "empieza con Mar" o que el producto "contiene la palabra pro". LIKE es el operador que convierte esas descripciones vagas en búsquedas exactas, usando dos comodines: el porcentaje (%) que representa cualquier cantidad de caracteres, y el guion bajo (_) que representa exactamente un carácter. Es como buscar en Google con asteriscos, pero dentro de tu base de datos.

La segunda herramienta son las funciones agregadas: COUNT, SUM, AVG, MIN y MAX. Estas funciones hacen algo magico: toman un monton de filas y las comprimen en un solo valor. COUNT cuenta cuantas filas hay, SUM suma una columna numerica, AVG calcula el promedio, MIN encuentra el valor más pequeño y MAX el más grande. Con ellas respondes preguntas como "cuantos clientes tengo", "cuál fue la venta más alta del año" o "cual es el precio promedio de mis productos". Son las funciones que más se usan en reportes, tableros y análisis de negocio.

La tercera herramienta es GROUP BY. Si las funciones agregadas comprimen todas las filas en un solo número, GROUP BY te permite comprimir por grupos: ventas por categoría, clientes por ciudad, ingresos por mes. Es la cláusula que convierte una tabla plana de diez mil ventas en un resumen legible de diez renglones donde cada renglon es una categoría con su total. Aprender GROUP BY es aprender a pensar en conjuntos, y eso cambia por completo la forma en que lees una base de datos.

La cuarta herramienta es HAVING. Ya conoces WHERE, que filtra filas antes de resumir. HAVING filtra después de resumir: te quedas solo con los grupos cuyo total supere cierta cantidad, o con las categorias que tengan más de cien productos. WHERE trabaja sobre las filas crudas, HAVING trabaja sobre los resúmenes. Confundirlos es uno de los errores más comunes de quienes empiezan con SQL, y en esta unidad lo vas a dejar clarísimo con ejemplos que muestran la diferencia fila por fila.

Lo bonito de esta unidad es que las cuatro herramientas se combinan entre si como piezas de LEGO. Puedes buscar clientes con LIKE, agruparlos por ciudad con GROUP BY, contarlos con COUNT y quedarte solo con las ciudades que tengan más de cincuenta clientes usando HAVING. Todo en una sola consulta. Cuando domines esas combinaciones, vas a poder responder casi cualquier pregunta de negocio que te pongan enfrente sin salir de SQL.

Una advertencia honesta antes de empezar: esta unidad tiene más conceptos que las anteriores, y algunos son tramposos. La diferencia entre COUNT(*) y COUNT(columna) con valores NULL, el comportamiento de AVG con números enteros, la regla de que no puedes seleccionar columnas que no esten agrupadas... todo eso suena a detalle menor hasta que te muerde en un reporte real. Por eso cada concepto viene con ejemplos paso a paso, errores comunes explicados y una sección de práctica con soluciones detalladas. Tomate tu tiempo, prueba cada ejemplo en tu base de datos y no avances hasta que cada pieza te quede clara.

Al final de esta unidad vas a saber buscar texto por patrones, resumir datos con funciones agregadas, agrupar resultados y filtrar grupos. En otras palabras: vas a saber convertir una base de datos llena de filas sueltas en inteligencia de negocio. Empecemos.

Explicación detallada

LIKE: buscar patrones en texto

En la unidad anterior usaste el operador = para comparar valores exactos. Funciona muy bien cuando sabes exactamente lo que buscas: ciudad = 'Mexico', precio = 100. Pero la vida real rara vez es tan exacta. Lo normal es que sepas una parte del texto: el correo "termina en @gmail.com", el nombre del producto "empieza con Lap", la descripción "contiene la palabra inalámbrico". Para esos casos existe LIKE, un operador que compara texto contra un patrón en lugar de contra un valor exacto.

Un patrón es una cadena de texto donde algunos caracteres son literales (deben coincidir tal cual) y otros son comodines (representan caracteres desconocidos). SQL tiene dos comodines: el signo de porcentaje (%) y el guion bajo (_). Con solo esos dos símbolos puedes expresar casi cualquier patrón de búsqueda.

El porcentaje (%) significa "cero o más caracteres cualesquiera". Es el comodín más usado. Si escribes LIKE 'Mar%', estás diciendo "que empiece con Mar y después venga lo que sea": coincide con "Maria", "Mario", "Marta", "Marmota" e incluso con "Mar" solito, porque cero caracteres también cuenta. Si escribes LIKE '%ez', pides "que termine en ez": "Lopez", "Martinez", "Perez", "Gonzalez". Y si escribes LIKE '%pro%', pides "que contenga pro en cualquier parte": "Producto", "Laptop Pro", "Promocion", "Aprobado".

El guion bajo (_) significa "exactamente un carácter cualquiera". Es más estricto que el porcentaje. LIKE 'M_r_' coincide con "Mara", "Moro", "Mira", pero no con "Maria" (tiene cuatro letras después de la M y el patrón solo permite tres en total) ni con "Mar" (le falta un carácter). Es útil cuando sabes la longitud del texto o la posición exacta de las letras que buscas. Por ejemplo, para buscar códigos de producto con formato fijo como "A-123" usarías LIKE 'A-___': la A, el guion y exactamente tres caracteres más.

Veamos ejemplos concretos con una tabla de clientes. Supongamos que quieres todos los clientes cuyo nombre empiece con "J".

```
SELECT nombre, correo
FROM clientes
WHERE nombre LIKE 'J%';
```

Esta consulta devuelve "Juan", "Juana", "Jose", "Jimena"... todo lo que empiece con J. Ahora, los clientes cuyo correo sea de Gmail:

```
SELECT nombre, correo
FROM clientes
WHERE correo LIKE '%@gmail.com';
```

El patrón '%@gmail.com' significa "cualquier cosa, seguida de @gmail.com". Coincide con "juan@gmail.com", "maria.lopez@gmail.com", etc. Y si buscas productos cuya descripción mencione "inalambrico":

```
SELECT nombre, descripcion
FROM productos
WHERE descripcion LIKE '%inalambrico%';
```

Los dos porcentajes, uno al inicio y otro al final, significan "que contenga la palabra en cualquier posición".

LIKE vs = : cuando usar cada uno

El operador = exige coincidencia exacta de toda la cadena. nombre = 'Juan' solo devuelve las filas donde el nombre sea exactamente "Juan", ni "Juan Perez" ni "Juana". LIKE sin comodines se comporta igual que =: nombre LIKE 'Juan' es lo mismo que nombre = 'Juan'. La diferencia aparece cuando metes comodines al patrón.

Una regla práctica: si sabes el valor completo y exacto, usa =. Es más rápido y más claro. Si solo sabes una parte (inicio, final, fragmento intermedio) o quieres flexibilidad, usa LIKE con comodines. Nunca uses LIKE '%texto%' cuando un = bastaría, porque LIKE con porcentaje al inicio es mucho más lento, como veremos en la sección de rendimiento.

Otra diferencia importante: = con valores NULL nunca es verdadero. Ya lo viste en la unidad de WHERE: para buscar NULL se usa IS NULL. Con LIKE pasa lo mismo: columna LIKE '%algo%' devuelve falso (en realidad NULL) cuando la columna es NULL, así que esas filas no aparecen en el resultado. Si tu columna puede tener NULL y te importan, combinalo: WHERE correo LIKE '%@gmail.com' OR correo IS NULL.

Mayúsculas, minúsculas y acentos

Aquí hay un detalle que confunde a muchos: si LIKE distingue entre mayúsculas y minúsculas depende de la configuración de tu base de datos, no de SQL en general. En MySQL con la configuración más común, LIKE 'juan%' encuentra "Juan", "JUAN" y "juan" porque la comparación no distingue mayúsculas. En PostgreSQL, en cambio, LIKE si distingue: 'juan%' no encuentra "Juan", y para ignorar mayúsculas se usa ILIKE. En SQL Server depende de la configuración de la base de datos.

Para escribir código portable (que funcione igual en cualquier motor), convierte ambos lados a minúsculas con LOWER:

```
SELECT nombre
FROM clientes
WHERE LOWER(nombre) LIKE 'juan%';
```

Así no importa como este guardado el dato ni como este configurado el motor: siempre comparas minúsculas contra minúsculas. Es un hábito que te ahorra dolores de cabeza cuando cambias de base de datos.

El carácter ESCAPE: buscar % y _ literales

Hay un problema evidente: ¿qué pasa si quieres buscar un texto que contenga un porcentaje o un guion bajo de verdad. Por ejemplo, buscar productos con "50% de descuento" en la descripción, o códigos que contengan un guion bajo literal. Si escribes LIKE '%50%%', el motor no sabe cuál % es comodín y cuál es literal.

La solución es la cláusula ESCAPE, que define un carácter de escape. Todo lo que vaya después del carácter de escape se trata como literal:

```
SELECT nombre, descripcion
FROM productos
WHERE descripcion LIKE '%50!%%' ESCAPE '!';
```

Aquí el ! marca que el % siguiente es literal: el patrón significa "contenga 50% seguido de cualquier cosa". Sin el ESCAPE, ese patrón sería ambiguo. Para guiones bajos literales funciona igual:

```
SELECT codigo
FROM productos
WHERE codigo LIKE 'A!_001' ESCAPE '!';
```

Esto busca literalmente el código "A_001", no "A más cualquier carácter más 001". El carácter de escape puede ser el que quieras (!, \, #); lo importante es declararlo con ESCAPE. Si no lo declaras, el motor no aplica ningún escape y los comodines siempre son comodines.

NOT LIKE

Igual que existe NOT para negar condiciones, existe NOT LIKE para pedir "todo lo que NO coincida con el patrón":


```
SELECT nombre, correo
FROM clientes
WHERE correo NOT LIKE '%@gmail.com';
```

Esto devuelve todos los clientes cuyo correo no sea de Gmail. Ojo con los NULL otra vez: si el correo es NULL, NOT LIKE tampoco es verdadero, así que esas filas se excluyen. Es el comportamiento estandar de SQL con NULL y vale para todos los operadores de comparación.

Rendimiento: LIKE e índices

Esto es importante para la vida real. Las bases de datos usan índices para encontrar filas rápido, como el índice de un libro. Un patrón como 'Mar%' (porcentaje solo al final) si puede usar un índice sobre la columna, porque el motor sabe que todo lo que busca empieza con "Mar" y puede saltar directo a esa sección del índice. Es una búsqueda eficiente.

En cambio, un patrón como '%mar%' (porcentaje al inicio) no puede usar el índice de forma normal, porque el texto buscado puede estar en cualquier posición. El motor no tiene más remedio que revisar fila por fila toda la tabla. En una tabla de cien filas no lo notas; en una tabla de diez millones de filas, la consulta puede tardar minutos.

Consejos prácticos: prefiere patrones con el porcentaje al final ('texto%') cuando puedas; evita '%texto%' sobre tablas grandes si la consulta se va a ejecutar seguido; y si necesitas búsquedas de texto completo frecuentes (buscar palabras dentro de descripciones largas), investiga las funciones de full-text search de tu motor (MATCH en MySQL, to_tsvector en PostgreSQL), que están diseñadas justo para eso y son muchísimo más rápidas que LIKE '%...%'.

Alternativas: REGEXP y SIMILAR TO (mención breve)

LIKE cubre el 90 por ciento de las búsquedas por patrón, pero a veces necesitas patrones más complejos: "un correo valido", "un teléfono con formato (55) 1234-5678", "un código de tres letras seguidas de cuatro digitos". Para eso existen las expresiones regulares. MySQL y MariaDB tienen el operador REGEXP (o RLIKE), PostgreSQL tiene ~ y SIMILAR TO, Oracle tiene REGEXP_LIKE. Son más potentes pero también más lentas y su sintaxis varia entre motores, así que usalas solo cuando LIKE no alcance. En este curso nos quedamos con LIKE porque es estandar, portable y suficiente para la mayoría de los casos.

Funciones agregadas: resumir muchas filas en un valor

Las funciones agregadas toman un conjunto de filas y devuelven un solo valor que resume ese conjunto. Son cinco: COUNT, SUM, AVG, MIN y MAX. Si WHERE y LIKE sirven para encontrar filas, las funciones agregadas sirven para responder preguntas sobre grupos de filas: cuantos hay, cuanto suman, cual es el promedio, cual es el mínimo, cual es el máximo.

Piensa en una tabla de ventas con miles de filas. Cada fila es una venta individual con su monto. Las preguntas interesantes casi nunca son sobre una venta individual, sino sobre el conjunto: "cuanto vendimos en total", "cuál fue la venta más grande", "cuantas ventas hicimos". Cada una de esas preguntas es una función agregada.

COUNT: contar filas

COUNT cuenta filas o valores. Tiene tres formas y es crucial entender la diferencia.

COUNT(*) cuenta todas las filas del conjunto, sin importar si tienen NULL:

```
SELECT COUNT(*)  
FROM ventas;
```

Si la tabla ventas tiene 1,250 filas, devuelve 1250. Punto. Es la forma más común y la que debes usar cuando quieres saber "cuantas filas hay".

COUNT(columna) cuenta solo las filas donde esa columna NO es NULL:

```
SELECT COUNT(telefono)  
FROM clientes;
```

Si tienes 500 clientes pero solo 420 tienen teléfono registrado, devuelve 420. La diferencia entre COUNT(*) y COUNT(columna) es exactamente la cantidad de NULL en esa columna. Este detalle es fuente clásica de reportes mal calculados: si cuentas COUNT(correo) creyendo que cuentas clientes, y algunos clientes no tienen correo, tu número sale menor que el real.

COUNT(DISTINCT columna) cuenta valores distintos, ignorando duplicados y NULL:

```
SELECT COUNT(DISTINCT ciudad)  
FROM clientes;
```

Si tus clientes están en 30 ciudades distintas (aunque haya miles de clientes), devuelve 30. Es perfecto para preguntas como "en cuantas ciudades vendemos" o "cuantos productos diferentes se vendieron".

SUM: sumar valores numericos

SUM suma todos los valores de una columna numerica, ignorando los NULL:

```
SELECT SUM(monto)
FROM ventas;
```

Si las ventas son 100, 200 y 300, devuelve 600. Si alguna venta tiene monto NULL (raro, pero puede pasar), simplemente no se suma, como si no existiera. SUM solo tiene sentido con columnas numericas; si la aplicas a texto, la mayoría de los motores devuelve error.

Un uso típico: total de ingresos del mes.

```
SELECT SUM(monto) AS total_ingresos
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01';
```

Fíjate en el alias AS total_ingresos: cuando usas funciones agregadas, el nombre de la columna resultante seria algo feo como "SUM(monto)", así que siempre ponle un alias descriptivo. Es una buena práctica que verás en todos los ejemplos de esta unidad.

AVG: el promedio

AVG calcula el promedio aritmetico: suma los valores no NULL y divide entre la cantidad de valores no NULL:

```
SELECT AVG(precio) AS precio_promedio
FROM productos;
```

Aquí hay dos trampas clásicas. Primera: AVG ignora los NULL, así que el promedio se calcula solo sobre las filas con valor. Si tienes precios 100, 200 y NULL, el promedio es 150 (300 entre 2), no 100 (300 entre 3). Segunda trampa: en algunos motores, si la columna es de tipo entero, AVG puede devolver un entero truncado. En MySQL, AVG de enteros devuelve decimal con cuatro decimales, pero en SQL Server, AVG de una columna INT devuelve INT (trunca los decimales). Para ir a lo seguro, convierte a decimal: AVG(CAST(precio AS DECIMAL(10,2))). Lo veremos de nuevo en errores comunes.

MIN y MAX: los extremos

MIN devuelve el valor más pequeño y MAX el más grande del conjunto:

```
SELECT MIN(precio) AS precio_minimo,  
       MAX(precio) AS precio_maximo  
FROM productos;
```

A diferencia de SUM y AVG, MIN y MAX funcionan con cualquier tipo de dato: números, fechas y texto. Con fechas, MIN(fecha) es la fecha más antigua y MAX(fecha) la más reciente. Con texto, usan orden alfabético: MIN(nombre) sería el primer nombre alfabéticamente. Ignoran los NULL igual que las demás.

Combinar MIN y MAX con fechas responde preguntas muy útiles: "cuál fue la primera y la última venta del año", "desde cuándo tenemos registros".

Combinar varias agregadas en una consulta

Puedes usar varias funciones agregadas en el mismo SELECT, y es lo más normal en reportes:

```
SELECT COUNT(*) AS total_ventas,  
       SUM(monto) AS ingresos_totales,  
       AVG(monto) AS ticket_promedio,  
       MIN(monto) AS venta_minima,  
       MAX(monto) AS venta_maxima  
FROM ventas  
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01';
```

Una sola consulta te da el resumen completo del mes: cuántas ventas, cuánto dinero, ticket promedio, la venta más chica y la más grande. Esto es lo que en la práctica alimenta tableros y reportes: una consulta, cinco números, cero vueltas.

Agregadas y WHERE

Las funciones agregadas se calculan sobre las filas que pasan el WHERE. Es decir, primero se filtran las filas y después se resumen. Esta consulta:

```
SELECT AVG(monto) AS promedio  
FROM ventas  
WHERE categoria = 'Electronica';
```

calcula el promedio solo de las ventas de electrónica, porque WHERE eliminó las demás filas antes de que AVG hiciera su trabajo. El orden lógico es: FROM, luego WHERE, luego las agregadas del SELECT. Ten esto presente porque con GROUP BY y HAVING el orden se vuelve más interesante.

Agregadas con DISTINCT

Todas las agregadas aceptan DISTINCT para operar solo sobre valores distintos: SUM(DISTINCT columna), AVG(DISTINCT columna). Es poco común, pero existe. El caso más usado es COUNT(DISTINCT columna), que ya vimos. Para SUM y AVG con DISTINCT, piensa dos veces si de verdad quieres ignorar duplicados, porque normalmente cada fila cuenta aunque el valor se repita.

Agregación condicional: resumir con condiciones dentro de la función

Hay una técnica que combina lo mejor de WHERE y de las agregadas: meter un CASE dentro de la función agregada. Sirve para responder varias preguntas condicionales en una sola consulta, sin necesidad de varios SELECT.

Supongamos que quieres saber, en una sola fila, cuántas ventas fueron grandes (monto mayor a 10,000) y cuántas fueron chicas:

```
SELECT COUNT(*) AS total_ventas,  
       SUM(CASE WHEN monto > 10000 THEN 1 ELSE 0 END) AS ventas_grandes,  
       SUM(CASE WHEN monto <= 10000 THEN 1 ELSE 0 END) AS ventas_chicas  
FROM ventas;
```

El CASE devuelve 1 cuando la fila cumple la condición y 0 cuando no; SUM suma esos unos y ceros, y el resultado es un conteo condicional. Es equivalente a hacer dos consultas con WHERE distintos, pero en una sola pasada por la tabla. También funciona con SUM para sumar solo lo que cumpla la condición:

```
SELECT SUM(CASE WHEN categoria = 'Electronica' THEN monto ELSE 0 END) AS  
total_electronica,  
       SUM(CASE WHEN categoria = 'Ropa' THEN monto ELSE 0 END) AS total_ropa  
FROM ventas;
```

Esto pivota las categorías a columnas: un renglón con el total de cada categoría. Es la base de los reportes cruzados y una de las técnicas más valoradas en entrevistas de SQL. La regla es simple: el CASE decide que filas aportan a la suma (o al conteo) y cuáles aportan cero.

GROUP BY: agrupar filas y calcular por grupo

Las funciones agregadas que acabas de ver comprimen todas las filas en un solo número. Útil, pero limitado: qué pasa si no quieres el total general sino el total por categoría, por ciudad o por mes. Para eso está GROUP BY: divide las filas en grupos según una o más columnas, aplica la función agregada dentro de cada grupo y devuelve un renglón por grupo.

Imagina una tabla de ventas con una columna categoría que tiene valores como "Electrónica", "Ropa" y "Hogar". Esta consulta:

```
SELECT categoria,  
       COUNT(*) AS num_ventas,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY categoria;
```

hace lo siguiente, paso a paso. Primero toma todas las filas de ventas. Después las separa en montones: todas las de Electrónica juntas, todas las de Ropa juntas, todas las de Hogar juntas. Después, dentro de cada montón, cuenta las filas y suma los montos. El resultado tiene un renglón por categoría:

categoria	num_ventas	total
Electronica	450	89000
Ropa	620	54000
Hogar	310	41000

De 1,380 filas sueltas pasaste a 3 renglones que cuentan una historia: electrónica vende menos unidades pero deja más dinero. Eso es inteligencia de negocio en una consulta de cuatro líneas.

Cómo se forman los grupos

GROUP BY agrupa por valores iguales. Todas las filas que tengan el mismo valor en la columna de agrupación caen en el mismo grupo. Si agrupas por ciudad, hay un grupo por cada ciudad distinta. Las filas donde la columna sea NULL forman su propio grupo (el grupo de los NULL), porque en agrupación los NULL se consideran iguales entre sí.

El orden de los grupos en el resultado no está garantizado. Si quieres los grupos ordenados, agrega ORDER BY. Esto es importante: GROUP BY agrupa, no ordena. En la práctica muchos motores devuelven los grupos ordenados por casualidad, pero no debes confiar en eso. Si el reporte debe salir ordenado, escribe ORDER BY explícitamente:

```
SELECT categoria,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY categoria  
ORDER BY total DESC;
```

Así las categorías salen de mayor a menor ingreso, que es como normalmente quieres ver un ranking.

Que columnas puedes seleccionar con GROUP BY

Esta es la regla de oro de GROUP BY y la causa del error más común de los principiantes: en el SELECT solo puedes poner columnas que estén en el GROUP BY, o columnas dentro de una función agregada. Nada más.

Esto es válido:

```
SELECT categoria, COUNT(*)  
FROM ventas  
GROUP BY categoria;
```

categoría está en el GROUP BY, COUNT(*) es una agregada. Perfecto. Esto NO es válido:

```
SELECT categoria, nombre_producto, COUNT(*)  
FROM ventas  
GROUP BY categoria;
```

nombre_producto no está agrupado ni dentro de una agregada. Por qué está prohibido? Porque dentro del grupo "Electrónica" hay muchos productos distintos: ¿cuál nombre mostraría el motor en el renglón del grupo? No hay una respuesta correcta, así que SQL lo prohíbe. MySQL en configuraciones permisivas lo permite y devuelve un valor arbitrario del grupo, lo cual es peligrosísimo porque el resultado parece correcto pero es aleatorio. PostgreSQL y SQL Server lo rechazan con error, que es lo correcto. Aprende la regla y no dependas de la permisividad de tu motor.

Si necesitas un dato del grupo que no es la columna agrupada, úsalo dentro de una agregada: MIN(nombre_producto) te da el primero alfabéticamente, MAX(fecha) la fecha más reciente del grupo. Así siempre hay una respuesta bien definida.

Agrupar por varias columnas

Puedes agrupar por más de una columna separandolas con comas. El grupo se forma por la combinación de valores:

```
SELECT categoria, ciudad,  
       COUNT(*) AS num_ventas,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY categoria, ciudad;
```

Esto crea un grupo por cada combinación categoría-ciudad: "Electrónica-Mexico", "Electrónica-Guadalajara", "Ropa-Mexico", etc. Es como una tabla dinámica de Excel con dos dimensiones. El número de grupos es el número de combinaciones distintas que existan en los datos.

El orden de las columnas en el GROUP BY no cambia los grupos (agrupar por categoría, ciudad es lo mismo que por ciudad, categoría en cuanto a que grupos se forman), pero si afecta el orden en que suelen aparecer si no hay ORDER BY. Para controlar la presentación, usa ORDER BY con las columnas que quieras.

Agrupar por expresiones y fechas

No estas limitado a columnas simples: puedes agrupar por expresiones. El caso más útil es agrupar por partes de una fecha para hacer resúmenes por año o por mes:

```
SELECT YEAR(fecha) AS ano,  
       MONTH(fecha) AS mes,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY YEAR(fecha), MONTH(fecha)  
ORDER BY ano, mes;
```

Esto te da el total de ventas mes por mes, que es la base de cualquier reporte de tendencias. YEAR y MONTH son funciones de fecha (en MySQL; en PostgreSQL usarías EXTRACT(YEAR FROM fecha)). También puedes agrupar por expresiones aritmeticas o por CASE, por ejemplo para crear rangos de precio:


```

SELECT CASE
    WHEN precio < 100 THEN 'Economico'
    WHEN precio < 500 THEN 'Medio'
    ELSE 'Premium'
END AS rango_precio,
COUNT(*) AS num_productos
FROM productos
GROUP BY CASE
    WHEN precio < 100 THEN 'Economico'
    WHEN precio < 500 THEN 'Medio'
    ELSE 'Premium'
END;

```

Agrupar por CASE te permite crear categorías que no existen en la tabla. Es una técnica muy usada en reportes: rangos de edad, rangos de precio, segmentos de clientes.

GROUP BY con WHERE

WHERE filtra filas antes de agrupar. Si solo te interesan las ventas de 2026, el WHERE elimina las demás y GROUP BY agrupa lo que queda:

```

SELECT MONTH(fecha) AS mes,
       SUM(monto) AS total
FROM ventas
WHERE YEAR(fecha) = 2026
GROUP BY MONTH(fecha)
ORDER BY mes;

```

El orden lógico de ejecución es: FROM (toma la tabla), WHERE (filtra filas), GROUP BY (forma grupos), SELECT (calcula agregadas y columnas). Entender este orden es la llave para no confundirte con HAVING, que viene justo después.

HAVING: filtrar grupos (WHERE filtra filas, HAVING filtra resúmenes)

Ya sabes filtrar filas con WHERE. Pero hay preguntas que WHERE no puede responder: "dime las categorías con más de 100 ventas", "muéstrame las ciudades cuyo total supere los 50,000 pesos", "que meses tuvieron un ticket promedio mayor a 1,000". En todas ellas, la condición se aplica sobre el resultado de una función agregada, es decir, sobre el resumen del grupo, no sobre las filas individuales. Para eso existe HAVING.

HAVING se escribe después del GROUP BY y filtra grupos completos según una condición que normalmente usa funciones agregadas:

```
SELECT categoria,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
GROUP BY categoria
HAVING SUM(monto) > 50000;
```

Paso a paso: se agrupan las ventas por categoría, se calcula el total de cada grupo, y después HAVING elimina los grupos cuyo total no supere 50,000. Solo sobreviven las categorías "grandes". WHERE no podría hacer esto porque WHERE se ejecuta antes de agrupar, cuando todavía no existe ningún total por categoría.

La diferencia en una frase: WHERE filtra filas antes de resumir, HAVING filtra grupos después de resumir. Memorízala, porque es la idea central de esta sección.

Veamos un ejemplo que muestra la diferencia con datos concretos. Supongamos estas ventas:

categoria	monto
Electronica	30000
Electronica	40000
Ropa	10000
Ropa	15000
Hogar	5000

Consulta con WHERE:

```
SELECT categoria, SUM(monto) AS total
FROM ventas
WHERE monto > 12000
GROUP BY categoria;
```

WHERE elimina primero las filas con monto menor o igual a 12,000 (se van Ropa 10000 y Hogar 5000). Después agrupa lo que queda: Electrónica suma 70,000, Ropa suma 15,000. Hogar desaparece por completo.

Consulta con HAVING:

```
SELECT categoria, SUM(monto) AS total
FROM ventas
GROUP BY categoria
HAVING SUM(monto) > 12000;
```

Aquí se agrupa todo primero: Electrónica 70,000, Ropa 25,000, Hogar 5,000. Después HAVING elimina los grupos con total menor o igual a 12,000: se va Hogar, quedan Electrónica y Ropa. Fíjate que Ropa sobrevivió con HAVING (su total es 25,000) pero habría quedado reducida con WHERE (solo 15,000). Misma tabla, misma condición numérica, resultados distintos. La posición de la condición (antes o después de agrupar) lo cambia todo.

Puedes combinar WHERE y HAVING en la misma consulta, y es muy común:

```
SELECT categoria,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
WHERE YEAR(fecha) = 2026
GROUP BY categoria
HAVING COUNT(*) > 100 AND SUM(monto) > 50000
ORDER BY total DESC;
```

Lee la consulta en orden lógico: de las ventas de 2026 (WHERE), agrupadas por categoría (GROUP BY), quédate con las que tengan más de 100 ventas y más de 50,000 de total (HAVING), ordenadas de mayor a menor (ORDER BY). Cada cláusula hace su parte y el resultado es un reporte listo para presentar.

Se puede usar el alias en HAVING?

En el SELECT definimos SUM(monto) AS total. Sería cómodo escribir HAVING total > 50000. En MySQL funciona, porque MySQL permite usar alias del SELECT en HAVING. En PostgreSQL y SQL Server no: ahí debes repetir la función, HAVING SUM(monto) > 50000. Para escribir código portable, repite la función agregada en el HAVING en lugar de usar el alias. Es un poco más verboso pero funciona en todos los motores.

HAVING sin GROUP BY

Técnicamente puedes escribir HAVING sin GROUP BY. En ese caso, toda la tabla se trata como un solo grupo:

```
SELECT SUM(monto) AS total
FROM ventas
HAVING SUM(monto) > 10000000;
```

Si el total supera el millon, devuelve un renglon; si no, devuelve cero renglones. Funciona, pero es raro y confuso. Casi siempre que veas HAVING debería haber un GROUP BY cerca. Si lo usas sin GROUP BY, asegurate de tener una buena razón y de comentarlo, porque quien lea tu código (incluido tu yo del futuro) va a fruncir el ceno.

Errores comunes

Uno: seleccionar columnas no agrupadas. Ya lo vimos en la regla de oro: todo lo que este en el SELECT debe estar en el GROUP BY o dentro de una agregada. Si tu motor lo permite y no da error, peor: te devuelve datos arbitrarios disfrazados de correctos.

Dos: poner condiciones de agregadas en el WHERE. Esto falla:

```
SELECT categoria, SUM(monto) AS total
FROM ventas
WHERE SUM(monto) > 50000
GROUP BY categoria;
```

WHERE no conoce las agregadas porque se ejecuta antes de que existan. El motor devuelve error. La condición sobre SUM va en HAVING, siempre.

Tres: olvidar que COUNT(columna) ignora NULL. Si agrupas clientes por ciudad y cuentas con COUNT(teléfono), las ciudades donde muchos clientes no tengan teléfono saldrán con números menores que su cantidad real de clientes. Para contar filas del grupo, usa COUNT(*)

Cuatro: AVG de enteros truncado. En algunos motores, AVG sobre una columna INT devuelve INT. Si tus montos son enteros y el promedio real es 99.6, verás 99. Convierte a decimal antes de promediar: AVG(CAST(monto AS DECIMAL(10,2))).

Cinco: confundir el orden de WHERE y HAVING. Recuerda el orden lógico: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. WHERE filtra filas (antes), HAVING filtra grupos (después). Si la condición habla de una columna cruda (categoría = 'Ropa'), va en WHERE. Si habla de un resumen (SUM(monto) > 50000), va en HAVING.

Seis: usar HAVING para algo que WHERE haría mejor. Si quieres las ventas de electrónica por ciudad, filtra con WHERE categoría = 'Electrónica' antes de agrupar, no con HAVING después. Filtrar antes es más eficiente (menos filas que agrupar) y más claro.

Buenas prácticas

Pon siempre alias descriptivos a las columnas agregadas: total_ventas, ticket_promedio, num_clientes. Un reporte con columnas llamadas "SUM(monto)" es difícil de leer y de usar desde una aplicación.

Agrega ORDER BY cuando el orden importe. GROUP BY no ordena; si tu reporte es un ranking, ordenalo explícitamente, normalmente por la agregada de mayor a menor.

Filtra con WHERE todo lo que puedas antes de agrupar. Mientras menos filas entren al GROUP BY, más rápida la consulta y más claro el código. Reserva HAVING para condiciones sobre agregadas, que es lo único que WHERE no puede hacer.

Cuidado con agrupar por columnas con muchos valores distintos (como un ID o una fecha exacta con hora): obtendras miles de grupos de una sola fila y el resumen no resumira nada. Agrupa por categorías con sentido de negocio: categoría, ciudad, mes, rango.

Cuando uses LIKE junto con agregadas, recuerda el tema de rendimiento: un WHERE con LIKE '%texto%' sobre una tabla enorme antes de un GROUP BY puede ser lento. Si el reporte se corre seguido, considera si el patrón puede reescribirse con el porcentaje al final, o si conviene una columna calculada.

Por último, prueba tus agregadas con datos pequenos donde puedas verificar a mano. Toma cinco filas, calcula el total con calculadora y compara con lo que devuelve tu consulta. Esa costumbre te va a salvar de los errores silenciosos (NULL ignorados, enteros truncados, columnas no agrupadas) que son los más peligrosos porque no dan error: dan un número equivocado con cara de correcto.

El orden lógico de ejecución: tu brújula

Todo lo de esta unidad encaja cuando entiendes el orden en que el motor ejecuta una consulta. No es el orden en que la escribes: es este.

1. FROM: toma las tablas y las filas.
2. WHERE: filtra filas individuales. Aquí viven LIKE, =, >, IS NULL y todas las condiciones sobre columnas crudas.
3. GROUP BY: forma los grupos con las filas que sobrevivieron.
4. HAVING: filtra grupos completos según sus resúmenes.
5. SELECT: calcula las columnas finales, incluyendo las funciones agregadas, y aplica los alias.
6. ORDER BY: ordena el resultado.

Este orden explica cada regla que viste. Por qué no puedes usar un alias del SELECT en el WHERE? Porque WHERE (paso 2) se ejecuta antes que SELECT (paso 5): el alias aun no existe. Por qué no puedes poner SUM(monto) > 50000 en el WHERE? Porque las agregadas se calculan en el SELECT, después del WHERE. Por qué HAVING puede usar agregadas?

Porque HAVING (paso 4) va después de GROUP BY (paso 3), cuando los grupos y sus resúmenes ya existen.

Cuando una consulta con GROUP BY te de un error o un resultado raro, recorre estos seis pasos mentalmente con tus datos y casi siempre encontraras el problema: un filtro puesto en el paso equivocado, una columna que no pertenece a ningun grupo, un alias usado antes de tiempo. Esta brújula te va a servir también en las unidades siguientes, cuando agreguemos JOIN y subconsultas al mapa.

Un apunte sobre GROUP BY con números de posición

Verás código que agrupa por número de columna en lugar de por nombre: GROUP BY 1, 2 significa "agrupa por la primera y segunda columna del SELECT". Funciona en MySQL y PostgreSQL, pero es una mala práctica: si alguien reordena las columnas del SELECT, el agrupamiento cambia en silencio y el reporte se rompe sin dar error. Escribe siempre los nombres de columna o las expresiones completas en el GROUP BY. Lo explícito siempre le gana a lo corto cuando otro humano (o tu yo de dentro de seis meses) tenga que leer la consulta.

Cuando GROUP BY no es la respuesta

GROUP BY es poderoso, pero no todo resumen necesita grupos. Si solo quieres un número global (total de ventas del mes, promedio general), no agrupes: usa las funciones agregadas directamente sin GROUP BY y obtendras un solo renglon. Agregar un GROUP BY innecesario no solo complica la consulta, también la hace más lenta porque el motor hace trabajo de agrupación para nada.

Tampoco uses GROUP BY para eliminar duplicados: para eso existe DISTINCT en el SELECT. SELECT DISTINCT ciudad FROM clientes te da la lista de ciudades sin necesidad de agrupar ni de contar nada. GROUP BY sin funciones agregadas es, en la práctica, un DISTINCT disfrazado y más confuso de leer.

Y si necesitas los grupos pero también el detalle de las filas (por ejemplo, cada venta junto con el total de su categoría), GROUP BY solo no basta porque comprime las filas. Para esos casos existen las funciones de ventana (como SUM(monto) OVER (PARTITION BY categoría)), que calculan el resumen sin comprimir las filas. Las veremos en una unidad avanzada; por ahora, que sepas que existen y que GROUP BY no es la única forma de resumir.

Ejemplos paso a paso

Para todos los ejemplos de esta sección usaremos dos tablas consistentes: clientes y ventas. Imaginemos una pequeña tienda en línea. La tabla clientes guarda los datos de cada cliente, y la tabla ventas guarda cada venta con el monto, la categoría del producto, la ciudad de entrega y la fecha.

```
CREATE TABLE clientes (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    correo VARCHAR(100),  
    ciudad VARCHAR(50),  
    telefono VARCHAR(20)  
);  
  
CREATE TABLE ventas (  
    id INT PRIMARY KEY,  
    cliente_id INT,  
    categoria VARCHAR(50),  
    monto DECIMAL(10,2),  
    ciudad VARCHAR(50),  
    fecha DATE  
);
```

Y estos son los datos de ejemplo. Tabla clientes:

id	nombre	correo	ciudad	telefono
1	Maria Lopez	maria.l@gmail.com	Mexico	5512345678
2	Juan Perez	juan.perez@hotmail.com	Guadalajara	3312345678
3	Ana Martinez	ana.m@yahoo.com	Mexico	NULL
4	Luis Garcia	luis.g@gmail.com	Monterrey	8112345678
5	Carmen Ruiz	carmen.r@gmail.com	Guadalajara	NULL
6	Pedro Sanchez	pedro.s@outlook.com	Mexico	5598765432
7	Lucia Fernandez	lucia.f@gmail.com	Puebla	2212345678
8	Miguel Torres	miguel.t@hotmail.com	Monterrey	8187654321

Tabla ventas:

id	cliente_id	categoria	monto	ciudad	fecha
1	1	Electronica	12500	Mexico	2026-09-05
2	2	Ropa	850	Guadalajara	2026-09-06
3	1	Electronica	3200	Mexico	2026-09-08
4	4	Hogar	4600	Monterrey	2026-09-10
5	3	Ropa	1200	Mexico	2026-09-12
6	5	Electronica	18900	Guadalajara	2026-09-15
7	6	Hogar	2300	Mexico	2026-09-18
8	2	Electronica	7400	Guadalajara	2026-09-20
9	7	Ropa	950	Puebla	2026-08-22
10	8	Electronica	15600	Monterrey	2026-08-25
11	4	Ropa	1100	Monterrey	2026-08-28
12	6	Electronica	9800	Mexico	2026-08-30

Con los datos a la vista, empecemos con LIKE.

Ejemplo 1: clientes cuyo nombre empiece con "M". El patrón 'M%' significa "empieza con M, lo demás no importa".

```
SELECT nombre, ciudad
FROM clientes
WHERE nombre LIKE 'M%';
```

Resultado: Maria Lopez, Miguel Torres. (Martin no existe en la tabla, pero "M%" también encontraría "M", "Mar", "Marta", cualquier cosa que empiece con M.)

Ejemplo 2: clientes con correo de Gmail. El patrón '%@gmail.com' significa "cualquier cosa que termine en @gmail.com".

```
SELECT nombre, correo
FROM clientes
WHERE correo LIKE '%@gmail.com';
```

Resultado: Maria Lopez (maria.l@gmail.com), Luis Garcia (luis.g@gmail.com), Carmen Ruiz (carmen.r@gmail.com), Lucia Fernandez (lucia.f@gmail.com). Cuatro clientes.

Ejemplo 3: clientes cuyo nombre contenga "ez". El patrón '%ez%' busca el fragmento en cualquier posición.

```
SELECT nombre
FROM clientes
WHERE nombre LIKE '%ez%';
```


Resultado: Maria Lopez, Juan Perez, Ana Martinez, Pedro Sanchez, Lucia Fernandez. Casi todos, porque "ez" es un fragmento muy comun en apellidos espanoles. Este ejemplo muestra por qué '%texto%' es flexible pero a veces demasiado amplio.

Ejemplo 4: búsqueda insensible a mayusculas de forma portable. Queremos los clientes cuyo nombre contenga "maria" sin importar como este escrito.

```
SELECT nombre, correo
FROM clientes
WHERE LOWER(nombre) LIKE '%maria%';
```

Resultado: Maria Lopez. Con LOWER en la columna, el patrón en minusculas coincide sin importar si el dato está en mayusculas, minusculas o combinado. Habito recomendado para código que deba funcionar en cualquier motor.

Ejemplo 5: correos que NO sean de Hotmail, usando NOT LIKE.

```
SELECT nombre, correo
FROM clientes
WHERE correo NOT LIKE '%@hotmail.com';
```

Resultado: todos menos Juan Perez y Miguel Torres, que son los dos de Hotmail. Recuerda: si hubiera un cliente con correo NULL, no apareceria aquí.

Ahora pasemos a las funciones agregadas.

Ejemplo 6: resumen general de las ventas de septiembre. Una sola consulta, cinco números.

```
SELECT COUNT(*) AS total_ventas,
       SUM(monto) AS ingresos_totales,
       AVG(monto) AS ticket_promedio,
       MIN(monto) AS venta_minima,
       MAX(monto) AS venta_maxima
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01';
```

Las ventas de septiembre son las de id 1 a 8 (ocho ventas). El resultado: total_ventas = 8, ingresos_totales = 50950, ticket_promedio = 6368.75, venta_minima = 850, venta_maxima = 18900. Verifica a mano: $12500 + 850 + 3200 + 4600 + 1200 + 18900 + 2300 + 7400 = 50950$. Correcto.

Ejemplo 7: la diferencia entre COUNT(*) y COUNT(columna). Contemos clientes y telefonos registrados.

```
SELECT COUNT(*) AS total_clientes,  
       COUNT(telefono) AS con_telefono,  
       COUNT(DISTINCT ciudad) AS num_ciudades  
FROM clientes;
```

Resultado: total_clientes = 8, con_telefono = 6 (Ana Martinez y Carmen Ruiz tienen NULL), num_ciudades = 4 (Mexico, Guadalajara, Monterrey, Puebla). Este ejemplo cabe en una tarjeta: tres formas de COUNT, tres respuestas distintas, cada una correcta para su pregunta.

Ejemplo 8: primera y última venta. Con fechas, MIN y MAX dan el rango temporal de los datos.

```
SELECT MIN(fecha) AS primera_venta,  
       MAX(fecha) AS ultima_venta,  
       COUNT(*) AS total_ventas  
FROM ventas;
```

Resultado: primera_venta = 2026-08-22, ultima_venta = 2026-09-20, total_ventas = 12. Útil para saber de que periodo cubren tus datos antes de hacer cualquier análisis.

Ahora GROUP BY.

Ejemplo 9: ventas por categoría. El ejemplo clásico: de filas sueltas a resumen por grupo.

```
SELECT categoria,  
       COUNT(*) AS num_ventas,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY categoria  
ORDER BY total DESC;
```

Calculamos a mano. Electrónica: 12500 + 3200 + 18900 + 7400 + 15600 + 9800 = 67400 (6 ventas). Ropa: 850 + 1200 + 950 + 1100 = 4100 (4 ventas). Hogar: 4600 + 2300 = 6900 (2 ventas). El resultado ordenado por total descendente:

categoria	num_ventas	total
Electronica	6	67400
Hogar	2	6900
Ropa	4	4100

La historia que cuentan los datos: electrónica es la categoría estrella con diferencia, ropa vende más unidades que hogar pero deja menos dinero. Todo eso en una consulta.

Ejemplo 10: ventas por ciudad y categoría (dos columnas de agrupación).

```
SELECT ciudad, categoria,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
GROUP BY ciudad, categoria
ORDER BY ciudad, total DESC;
```

Cada combinación ciudad-categoría forma un grupo. Por ejemplo, Mexico-Electrónica: ventas 1, 3 y 12 ($12500 + 3200 + 9800 = 25500$, 3 ventas). Guadalajara-Electrónica: ventas 6 y 8 ($18900 + 7400 = 26300$, 2 ventas). Monterrey-Hogar: venta 4 (4600, 1 venta). Y así con cada combinación presente en los datos. Las combinaciones que no existen (por ejemplo, Puebla-Electrónica) simplemente no aparecen: GROUP BY solo crea grupos para combinaciones que si hay en los datos.

Ejemplo 11: ingresos por mes usando funciones de fecha.

```
SELECT YEAR(fecha) AS ano,
       MONTH(fecha) AS mes,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
GROUP BY YEAR(fecha), MONTH(fecha)
ORDER BY ano, mes;
```

Agosto 2026: ventas 9, 10, 11, 12 ($950 + 15600 + 1100 + 9800 = 27450$, 4 ventas). Septiembre 2026: ventas 1 a 8 (50950, 8 ventas). El resultado muestra la tendencia: septiembre casi duplica a agosto. Este es el patrón básico de todo reporte mensual.

Ejemplo 12: agrupar por rangos con CASE. Clasificamos las ventas en "Chica" (menos de 2000), "Mediana" (2000 a 10000) y "Grande" (más de 10000).

```

SELECT CASE
    WHEN monto < 2000 THEN 'Chica'
    WHEN monto <= 10000 THEN 'Mediana'
    ELSE 'Grande'
END AS tamano_venta,
COUNT(*) AS num_ventas,
AVG(monto) AS ticket_promedio
FROM ventas
GROUP BY CASE
    WHEN monto < 2000 THEN 'Chica'
    WHEN monto <= 10000 THEN 'Mediana'
    ELSE 'Grande'
END
ORDER BY ticket_promedio;

```

Chicas: 850, 1200, 950, 1100 (4 ventas, promedio 1025). Medianas: 3200, 4600, 2300, 7400, 9800 (5 ventas, promedio 5060). Grandes: 12500, 18900, 15600 (3 ventas, promedio 15666.67). Los rangos no existen en la tabla: los creamos con CASE y agrupamos por ellos.

Ahora HAVING, donde se pone interesante.

Ejemplo 13: categorías con más de 3 ventas. La condición usa COUNT, así que va en HAVING.

```

SELECT categoria,
    COUNT(*) AS num_ventas,
    SUM(monto) AS total
FROM ventas
GROUP BY categoria
HAVING COUNT(*) > 3;

```

Grupos: Electrónica (6), Hogar (2), Ropa (4). HAVING elimina Hogar. Resultado: Electrónica y Ropa. WHERE no podría hacer esto porque el conteo por categoría no existe hasta después de agrupar.

Ejemplo 14: ciudades cuyo total de ventas supere los 20,000, solo en septiembre. Aquí se combinan WHERE y HAVING.

```

SELECT ciudad,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01'
GROUP BY ciudad
HAVING SUM(monto) > 20000
ORDER BY total DESC;

```

Primero WHERE deja solo las 8 ventas de septiembre. Después se agrupa por ciudad: Mexico (12500 + 3200 + 1200 + 2300 = 19200), Guadalajara (850 + 18900 + 7400 = 27150), Monterrey (4600). HAVING elimina Mexico (19200 no supera 20000) y Monterrey. Resultado: solo Guadalajara con 27150. Fíjate que sin el WHERE, Mexico habría sumado también sus ventas de agosto y el resultado sería distinto. El orden importa.

Ejemplo 15: la combinación completa. Categorías buscadas por patrón, agrupadas por ciudad, con umbral mínimo. "Dame, por ciudad, el total de las categorías relacionadas con tecnología ('%nica%' o '%tro%'), quedandote solo con las ciudades donde ese total supere los 10,000".

```

SELECT ciudad,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total_tecnologia
FROM ventas
WHERE categoria LIKE '%nica%'
GROUP BY ciudad
HAVING SUM(monto) > 10000
ORDER BY total_tecnologia DESC;

```

LIKE '%nica%' encuentra "Electrónica". WHERE deja solo las 6 ventas de electrónica. Se agrupa por ciudad: Mexico (12500 + 3200 + 9800 = 25500), Guadalajara (18900 + 7400 = 26300), Monterrey (15600). Todas superan 10000, así que HAVING no elimina ninguna. Resultado ordenado: Guadalajara 26300, Mexico 25500, Monterrey 15600. Una sola consulta combina las cuatro herramientas de la unidad: LIKE para buscar, GROUP BY para agrupar, SUM para resumir y HAVING para filtrar grupos.

Ejemplo 16: el guion bajo en acción. Busquemos nombres con el patrón 'M_r_a': M, un carácter cualquiera, r, un carácter cualquiera, a.

```

SELECT nombre
FROM clientes
WHERE nombre LIKE 'M_r_a';

```

"Maria" coincide: M, luego "a" ocupa el primer _, luego "r", luego "i" ocupa el segundo _, luego "a". Resultado: Maria Lopez. En cambio "Mara" no coincide porque le falta un carácter, y "Mariana" tampoco porque le sobran. Este ejemplo muestra la precisión del guion bajo frente a la flexibilidad del porcentaje: con '%', el patrón 'M%a' habría encontrado también "Mariana", "Mica" no (no tiene r)... en fin, cada comodín tiene su carácter.

Ejemplo 17: primera y última venta por categoría, con MIN y MAX sobre fechas dentro de grupos.

```
SELECT categoria,  
       MIN(fecha) AS primera_venta,  
       MAX(fecha) AS ultima_venta,  
       COUNT(*) AS num_ventas  
FROM ventas  
GROUP BY categoria  
ORDER BY primera_venta;
```

Electrónica: primera 2026-08-25, última 2026-09-20, 6 ventas. Ropa: primera 2026-08-22, última 2026-09-12, 4 ventas. Hogar: primera 2026-09-10, última 2026-09-18, 2 ventas. Este patrón (MIN/MAX de fecha por grupo) responde "desde cuando y hasta cuando" para cada segmento de tu negocio, y es la base para calcular antigüedad, frecuencia y recencia de clientes.

Ejemplo 18: en cuantas ciudades se vende cada categoría, con COUNT(DISTINCT) dentro de grupos.

```
SELECT categoria,  
       COUNT(DISTINCT ciudad) AS num_ciudades,  
       SUM(monto) AS total  
FROM ventas  
GROUP BY categoria;
```

Electrónica se vende en 3 ciudades (Mexico, Guadalajara, Monterrey), Ropa en 4 (Guadalajara, Mexico, Puebla, Monterrey), Hogar en 2 (Monterrey, Mexico). Fíjate que COUNT(DISTINCT ciudad) dentro de cada grupo cuenta las ciudades distintas de ese grupo, no del total. Es una de las combinaciones más útiles: GROUP BY abre los grupos y DISTINCT cuenta variedades dentro de cada uno.

Ejemplo 19: agregación condicional. En una sola consulta, total de ventas, cuantas fueron grandes (monto mayor a 10,000) y el monto total de esas grandes.

```
SELECT COUNT(*) AS total_ventas,
       SUM(CASE WHEN monto > 10000 THEN 1 ELSE 0 END) AS ventas_grandes,
       SUM(CASE WHEN monto > 10000 THEN monto ELSE 0 END) AS monto_grandes
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01';
```

En septiembre, las ventas mayores a 10,000 son la de 12500 y la de 18900: ventas_grandes = 2, monto_grandes = 31400, total_ventas = 8. Sin el CASE habrias necesitado dos consultas o una subconsulta. Con el CASE, una sola pasada responde las tres preguntas.

Ejemplo 20: HAVING con varias condiciones. Categorías con más de 2 ventas O con ticket promedio mayor a 8,000, en septiembre.

```
SELECT categoria,
       COUNT(*) AS num_ventas,
       AVG(monto) AS ticket_promedio,
       SUM(monto) AS total
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01'
GROUP BY categoria
HAVING COUNT(*) > 2 OR AVG(monto) > 8000;
```

Grupos de septiembre: Electrónica (4 ventas, promedio 10500), Ropa (2 ventas, promedio 1025), Hogar (2 ventas, promedio 3450). HAVING con OR: Electrónica pasa por ambas condiciones, Ropa y Hogar no pasan ninguna. Resultado: solo Electrónica. Cambia el OR por AND y el resultado seria el mismo aquí, pero en otros datos la diferencia es enorme. En HAVING puedes usar AND, OR y NOT igual que en WHERE: son condiciones logicas normales, solo que evaluadas sobre grupos.

Práctica

Resuelve cada ejercicio usando las tablas clientes y ventas de la sección anterior. Intenta escribir la consulta antes de leer la solución. Las soluciones están explicadas paso a paso para que entiendas el razonamiento, no solo la respuesta.

Ejercicio 1: Encuentra todos los clientes cuyo nombre termine en "ez" y cuyo correo NO sea de Yahoo. Muestra nombre y correo.

Piensa: "termine en ez" es un patrón con el porcentaje al inicio: '%ez'. "No sea de Yahoo" es NOT LIKE '%@yahoo.com'. Las dos condiciones se unen con AND.

```
SELECT nombre, correo
FROM clientes
WHERE nombre LIKE '%ez'
      AND correo NOT LIKE '%@yahoo.com';
```

Paso a paso: primero el motor evalúa nombre LIKE '%ez' (termina en "ez"): Maria Lopez, Juan Perez, Ana Martinez, Pedro Sanchez, Lucia Fernandez. Después aplica correo NOT LIKE '%@yahoo.com': Ana Martinez tiene correo ana.m@yahoo.com, así que se elimina. Resultado final: Maria Lopez, Juan Perez, Pedro Sanchez, Lucia Fernandez.

Ejercicio 2: Calcula el ticket promedio de las ventas de la categoría "Ropa" en septiembre de 2026. Muestra también cuantas ventas fueron y el monto total.

Piensa: necesitas filtrar filas (categoría Ropa, fecha en septiembre) con WHERE, y después resumir con COUNT, SUM y AVG. Como no hay GROUP BY, todo se resume en un solo renglón.

```
SELECT COUNT(*) AS num_ventas,
       SUM(monto) AS total,
       AVG(monto) AS ticket_promedio
FROM ventas
WHERE categoria = 'Ropa'
      AND fecha >= '2026-09-01'
      AND fecha < '2026-10-01';
```

Paso a paso: WHERE deja solo las ventas de Ropa en septiembre: id 2 (850) e id 5 (1200). La venta 9 es de Ropa pero de agosto, así que se excluye. Sobre esas 2 filas: COUNT = 2, SUM = 2050, AVG = 1025. Si hubieras olvidado el filtro de fecha, el promedio saldría distinto (4100 entre 4 = 1025 también en este caso por coincidencia, pero el total sería 4100 en vez de 2050). Moraleja: verifica tus filtros con COUNT antes de confiar en un promedio.

Ejercicio 3: Muestra, por ciudad, cuantos clientes hay y cuantos tienen teléfono registrado. Ordena de mayor a menor cantidad de clientes.

Piensa: agrupar por ciudad (GROUP BY), contar filas del grupo con COUNT(*) y contar teléfonos no NULL con COUNT(teléfono). Ordenar con ORDER BY.


```
SELECT ciudad,
       COUNT(*) AS total_clientes,
       COUNT(telefono) AS con_telefono
FROM clientes
GROUP BY ciudad
ORDER BY total_clientes DESC;
```

Paso a paso: los grupos son Mexico (Maria, Ana, Pedro: 3 clientes), Guadalajara (Juan, Carmen: 2), Monterrey (Luis, Miguel: 2), Puebla (Lucia: 1). Telefonos por grupo: Mexico tiene 2 (Ana es NULL), Guadalajara tiene 1 (Carmen es NULL), Monterrey tiene 2, Puebla tiene 1. Resultado ordenado: Mexico (3, 2), Guadalajara (2, 1), Monterrey (2, 1), Puebla (1, 1). Nota que Guadalajara y Monterrey empatan en 2 clientes; el orden entre ellas no esta garantizado salvo que agregues un segundo criterio al ORDER BY.

Ejercicio 4: Encuentra las ciudades donde se hayan hecho más de 2 ventas en total, mostrando la cantidad de ventas y el monto total. Solo considera ventas mayores a 1,000 pesos.

Piensa: "ventas mayores a 1,000" filtra filas, así que va en WHERE. "Más de 2 ventas" filtra grupos según COUNT, así que va en HAVING. Agrupamos por ciudad.

```
SELECT ciudad,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total
FROM ventas
WHERE monto > 1000
GROUP BY ciudad
HAVING COUNT(*) > 2
ORDER BY total DESC;
```

Paso a paso: WHERE elimina las ventas de 850, 950 y 1100... espera, 1100 si es mayor que 1000. Revisemos: se eliminan las de 850 (id 2) y 950 (id 9). Quedan 10 ventas. Agrupando por ciudad: Mexico tiene id 1, 3, 5, 7, 12 (5 ventas, total 12500 + 3200 + 1200 + 2300 + 9800 = 29000). Guadalajara tiene id 6, 8 (2 ventas). Monterrey tiene id 4, 10, 11 (3 ventas, total 4600 + 15600 + 1100 = 21300). Puebla tenía solo la venta 9, eliminada por el WHERE, así que no forma grupo. HAVING COUNT(*) > 2 elimina Guadalajara (2 ventas). Resultado: Mexico (5, 29000), Monterrey (3, 21300). Este ejercicio muestra el error típico: si pones monto > 1000 en el HAVING en vez del WHERE, el motor da error porque HAVING no filtra filas.

Ejercicio 5: Muestra el total de ventas por mes, pero solo para los meses cuyo total supere los 30,000 pesos. Incluye el número de ventas y el ticket promedio.

Piensa: agrupar por año y mes usando YEAR y MONTH, y filtrar grupos con HAVING SUM(monto) > 30000.

```
SELECT YEAR(fecha) AS ano,
       MONTH(fecha) AS mes,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total,
       AVG(monto) AS ticket_promedio
FROM ventas
GROUP BY YEAR(fecha), MONTH(fecha)
HAVING SUM(monto) > 30000
ORDER BY ano, mes;
```

Paso a paso: agosto 2026 suma 27450 (4 ventas), septiembre 2026 suma 67950 (8 ventas). HAVING elimina agosto. Resultado: un solo renglón, 2026-09, 8 ventas, total 67950, promedio 8493.75. Si quisieras ver también agosto, bajarías el umbral o quitarías el HAVING. Los umbrales en HAVING son los que convierten un reporte descriptivo en un reporte de excepciones: "avisame solo cuando algo pase de cierto nivel".

Ejercicio 6 (reto): Encuentra los clientes de Gmail cuyo nombre empiece con vocal (A, E, I, O, U), y para cada ciudad cuenta cuantos hay. Muestra solo las ciudades con al menos 2 de estos clientes.

Piensa: combina LIKE para el correo ('%@gmail.com'), una condición para la vocal inicial y GROUP BY con HAVING. Para la vocal inicial puedes usar varias condiciones LIKE con OR, o SUBSTRING. Usemos LIKE, que es el tema de la unidad.

```
SELECT ciudad,
       COUNT(*) AS num_clientes
FROM clientes
WHERE correo LIKE '%@gmail.com'
      AND (nombre LIKE 'A%' OR nombre LIKE 'E%'
           OR nombre LIKE 'I%' OR nombre LIKE 'O%'
           OR nombre LIKE 'U%')
GROUP BY ciudad
HAVING COUNT(*) >= 2;
```

Paso a paso: clientes con Gmail: Maria Lopez (Mexico), Luis Garcia (Monterrey), Carmen Ruiz (Guadalajara), Lucia Fernandez (Puebla). De ellos, los que empiezan con vocal: Ana... no, Ana no es Gmail. Maria empieza con M, Luis con L, Carmen con C, Lucia con L. Ninguno empieza con vocal. El resultado es cero renglones. Y eso está bien: es una respuesta válida. El ejercicio está diseñado para que practiques la estructura completa (LIKE + OR + GROUP BY +

HAVING) aunque con estos datos el resultado sea vacío. Prueba cambiar la condición a nombres que empiecen con consonante (por ejemplo NOT LIKE con las vocales) y verás como aparecen grupos. En datos reales, un resultado vacío también informa: te dice que ningún grupo cumple la condición.

Ejercicio 7: Para cada categoría, muestra el monto mínimo, el máximo y el rango (la diferencia entre máximo y mínimo). Ordena por rango de mayor a menor.

Piensa: MIN y MAX por grupo con GROUP BY categoría. El rango es una operación aritmetica entre dos agregadas: MAX(monto) - MIN(monto). Eso esta permitido porque ambas son agregadas del mismo grupo.

```
SELECT categoria,
       MIN(monto) AS venta_minima,
       MAX(monto) AS venta_maxima,
       MAX(monto) - MIN(monto) AS rango
FROM ventas
GROUP BY categoria
ORDER BY rango DESC;
```

Paso a paso: Electrónica: min 3200, max 18900, rango 15700. Ropa: min 850, max 1200, rango 350. Hogar: min 2300, max 4600, rango 2300. Resultado ordenado: Electrónica (15700), Hogar (2300), Ropa (350). El rango te dice que tan dispersos están los montos dentro de cada categoría: en electrónica conviven ventas chicas y grandes, en ropa todas las ventas son parecidas. Puedes operar con agregadas (sumarlas, restarlas, dividir las) igual que con columnas normales, siempre que sea dentro del mismo nivel de agrupación.

Ejercicio 8: Muestra el total por mes de las ventas cuya categoría contenga las letras "lec" (como en "Electrónica"), incluyendo número de ventas y ticket promedio, pero solo para los meses con más de 2 ventas.

Piensa: LIKE '%lec%' en el WHERE filtra las filas de electrónica (todas las categorias que contengan "lec"). GROUP BY ano y mes. HAVING COUNT(*) > 2 filtra los meses.

```

SELECT YEAR(fecha) AS ano,
       MONTH(fecha) AS mes,
       COUNT(*) AS num_ventas,
       SUM(monto) AS total,
       AVG(monto) AS ticket_promedio
FROM ventas
WHERE categoria LIKE '%lec%'
GROUP BY YEAR(fecha), MONTH(fecha)
HAVING COUNT(*) > 2
ORDER BY ano, mes;

```

Paso a paso: WHERE deja las 6 ventas de Electrónica. Por mes: agosto tiene 2 (id 10 y 12: $15600 + 9800 = 25400$), septiembre tiene 4 (id 1, 3, 6, 8: $12500 + 3200 + 18900 + 7400 = 42000$). HAVING elimina agosto (2 no es mayor que 2). Resultado: un renglon, 2026-09, 4 ventas, total 42000, promedio 10500. Fíjate en el detalle del umbral: "más de 2" es > 2 , no $>= 2$. Con $>= 2$, agosto habría sobrevivido. Los operadores de comparación en HAVING merecen la misma atención que en WHERE: un símbolo cambia el reporte.

Ejercicio 9: En una sola consulta y sin GROUP BY, muestra: el total de ventas de septiembre, cuantas fueron de la categoría "Electrónica" y cuál fue el monto total de las ventas que NO fueron de electrónica.

Piensa: agregación condicional con CASE dentro de SUM. Total de ventas con COUNT(*), conteo de electrónica con SUM(CASE...1...0), monto de no electrónica con SUM(CASE...monto...0). Todo filtrado a septiembre con WHERE.

```

SELECT COUNT(*) AS total_ventas,
       SUM(CASE WHEN categoria = 'Electronica' THEN 1 ELSE 0 END) AS
ventas_electronica,
       SUM(CASE WHEN categoria <> 'Electronica' THEN monto ELSE 0 END) AS
monto_no_electronica
FROM ventas
WHERE fecha >= '2026-09-01' AND fecha < '2026-10-01';

```

Paso a paso: las 8 ventas de septiembre. De electrónica: id 1, 3, 6, 8 (4 ventas). Las no electrónica: id 2 (850), 4 (4600), 5 (1200), 7 (2300); su monto suma 8950. Resultado: total_ventas = 8, ventas_electronica = 4, monto_no_electronica = 8950. Y aquí viene la verificación cruzada que todo analista debería hacer: el total de electrónica en septiembre (42000, calculado en el ejercicio 8) más el monto no electrónica (8950) debe dar el total de septiembre (50950, calculado en el ejemplo 6). $42000 + 8950 = 50950$. Cuadra perfecto. Cuando dos consultas distintas sobre los mismos datos se reconcilian así, puedes confiar en ambas.

Resumen de la unidad

En esta unidad diste el salto de leer filas sueltas a extraer inteligencia de tus datos. Estas son las ideas que debes llevarte grabadas.

LIKE busca texto por patrones, no por valores exactos. El porcentaje (%) representa cero o más caracteres y el guion bajo (_) exactamente un carácter. 'Mar%' es "empieza con Mar", '%ez' es "termina en ez", '%pro%' es "contiene pro". Usa = cuando sepas el valor exacto y LIKE cuando solo conozcas una parte. Para buscar % o _ literales usa ESCAPE. Para código portable entre motores, envuelve con LOWER cuando no quieras distinguir mayúsculas de minúsculas. Y recuerda el tema de rendimiento: los patrones con % al inicio no usan índices y son lentos en tablas grandes.

Las funciones agregadas comprimen muchas filas en un solo valor. COUNT(*) cuenta filas, COUNT(columna) cuenta valores no NULL, COUNT(DISTINCT columna) cuenta valores distintos. SUM suma, AVG promedia, MIN y MAX encuentran los extremos (y funcionan también con fechas y texto). Las agregadas ignoran los NULL, se calculan sobre las filas que pasan el WHERE, y sus columnas resultantes siempre deben llevar un alias descriptivo.

GROUP BY divide las filas en grupos y aplica las agregadas dentro de cada grupo, devolviendo un renglón por grupo. La regla de oro: en el SELECT solo van columnas del GROUP BY o columnas dentro de una agregada. Puedes agrupar por varias columnas, por expresiones como YEAR(fecha) y MONTH(fecha), y por CASE para crear rangos que no existen en la tabla. GROUP BY agrupa, no ordena: si quieres orden, escribe ORDER BY.

HAVING filtra grupos después de resumir; WHERE filtra filas antes de resumir. Esa es toda la diferencia y es la idea más importante de la unidad. Las condiciones sobre columnas crudas van en WHERE, las condiciones sobre funciones agregadas van en HAVING. El orden lógico de ejecución es FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY. Nunca pongas una agregada en el WHERE: el motor lo rechaza porque las agregadas aun no existen en ese punto.

Los errores que más debes vigilar: seleccionar columnas no agrupadas, confundir COUNT(*) con COUNT(columna) cuando hay NULL, el AVG truncado con enteros en algunos motores, y poner en HAVING lo que debería ir en WHERE (o al revés). La defensa contra todos ellos es la misma: prueba con pocos datos y verifica a mano.

Con estas cuatro herramientas combinadas (LIKE + agregadas + GROUP BY + HAVING) puedes responder preguntas de negocio reales en una sola consulta: buscar clientes por patrón de correo, resumir ventas del mes, rankear categorías por ingreso y quedarte solo con las que superen un umbral.

Un último consejo antes de cerrar: la mejor forma de fijar esta unidad es aplicarla a datos que te importen. Toma cualquier tabla que tengas a la mano (ventas, gastos, calificaciones, lo que sea) y hazte cinco preguntas: una de búsqueda por patrón, una de resumen global, una de resumen por grupo, una de filtrado de grupos y una que combine todo. Escríbelas en SQL, verifica los números a mano con un punado de filas y compara. Cuando puedas responder tus propias preguntas sin dudar entre WHERE y HAVING, esta unidad quedó dominada.

En la siguiente unidad conectaremos tablas entre si con JOIN, y estos resúmenes se volverán aun más poderosos: podrás agrupar ventas por nombre de cliente, por region de una tabla de sucursales o por cualquier dimensión que viva en otra tabla. Lo que aprendiste aquí es la base; lo que sigue es la combinación.

UNIDAD 6: MODIFICACIÓN DE DATOS Y RELACIONES

Introducción

Hasta este punto del curso has hecho algo muy importante: aprendiste a crear bases de datos, a definir tablas, a insertar información, a consultar datos con `SELECT`, a filtrar con `WHERE`, a ordenar, a agrupar y hasta a resumir información con funciones agregadas. En pocas palabras, aprendiste a construir el edificio y a entrar a cada cuarto a ver lo que hay dentro.

Pero aquí viene la primera gran verdad de esta unidad: una base de datos que solo recibe datos y nunca los cambia es una base de datos muerta. Los datos reales cambian todos los días. Los clientes cambian de teléfono, los productos cambian de precio, los pedidos pasan de "pendiente" a "enviado" y luego a "entregado", los empleados reciben aumentos de sueldo. Si tu base de datos no sabe actualizarse, se vuelve un archivo historico bonito pero inútil.

La segunda gran verdad es esta: una tabla aislada, por si sola, no sirve de mucho. El poder real de SQL no está en tener una tabla con clientes y otra tabla con pedidos sin conexión entre ellas. El poder real está en las relaciones: saber que el pedido 105 lo hizo el cliente 12, que ese cliente vive en la ciudad 4, y que ese pedido contiene los productos 7, 9 y 11. Las bases de datos relacionales se llaman así precisamente por eso: porque relacionan tablas entre si.

En esta unidad vas a aprender dos habilidades que cambian por completo tu forma de trabajar con datos. La primera habilidad es modificar datos existentes: cambiar valores con `UPDATE` y eliminar filas con `DELETE`. Parece sencillo, y lo es, hasta que descubres que un `UPDATE` sin `WHERE` puede arruinar la tabla completa de un cliente en un segundo. Aquí vas a aprender a hacerlo con disciplina, con protecciones y con la mentalidad correcta, porque en el mundo real estas dos instrucciones son las más peligrosas de SQL.

La segunda habilidad es el corazon de todo lo relacional: las llaves primarias y las llaves foráneas. Vas a entender qué es la identidad única de cada fila, cómo se genera automáticamente con `AUTO_INCREMENT`, por qué no debes usar datos que cambian como llave primaria, y como las llaves foráneas conectan tablas entre si para garantizar que los datos siempre tengan sentido. Vas a ver los tres tipos de relaciones: uno a uno, uno a muchos y muchos a muchos, y vas a aprender a construir la famosa tabla intermedia que resuelve las relaciones de muchos a muchos.

Esta unidad es la que separa a quien apenas conoce SQL de quien puede diseñar una base de datos de verdad. Una tienda en línea, un sistema de escuela, un hospital, un banco: todos usan UPDATE, DELETE, llaves primarias y llaves foráneas todos los días. Lo que aprendas aquí lo vas a usar en cada proyecto serio de tu vida como programador.

Trabajemos con las tablas de ejemplo que ya conocemos. Imaginemos una tienda en línea con clientes y pedidos. Esta unidad completa gira alrededor de ese escenario, porque modificar datos y relacionar tablas es exactamente lo que una tienda en línea hace miles de veces al día.

Explicación detallada

CLASE 17: UPDATE - cambiar datos existentes

La instrucción UPDATE sirve para modificar los datos que ya existen en una tabla. Su sintaxis básica es muy sencilla:

La estructura es: UPDATE, el nombre de la tabla, la palabra SET seguida de las columnas que quieres cambiar con sus nuevos valores, y la cláusula WHERE que dice exactamente que filas se van a modificar.

Veamos el ejemplo más común de todos: un cliente cambió su número de teléfono.

```
UPDATE clientes
SET telefono = '555-987-1234'
WHERE id_cliente = 12;
```

Lee esa instrucción en voz alta: "Actualiza la tabla clientes, pon el teléfono 555-987-1234, en la fila donde el id_cliente sea 12". Eso es todo. Una fila cambiada, las demás intactas. Este es el uso correcto y seguro de UPDATE: una condición precisa en el WHERE que identifica exactamente las filas que quieres tocar.

Puedes cambiar varias columnas al mismo tiempo separandolas con comas después de SET:

```
UPDATE clientes
SET telefono = '555-987-1234',
    correo = 'maria.nueva@correo.com',
    ciudad = 'Guadalajara'
WHERE id_cliente = 12;
```


También puedes actualizar muchas filas de un golpe cuando la condición del WHERE abarca varias. Por ejemplo, aumentar el sueldo de todos los empleados del departamento de ventas:

```
UPDATE empleados
SET sueldo = sueldo * 1.10
WHERE departamento = 'ventas';
```

Fíjate en algo interesante: el nuevo valor puede calcularse a partir del valor anterior. La expresión "sueldo * 1.10" toma el sueldo actual de cada fila y le suma el 10 por ciento. No necesitas saber cuanto gana cada empleado; SQL hace el calculo fila por fila.

Otro uso poderoso es UPDATE con subconsulta: actualizar filas basandote en datos de otra tabla. Supongamos que quieres marcar como "VIP" a todos los clientes que han hecho más de 10 pedidos:

```
UPDATE clientes
SET categoria = 'VIP'
WHERE id_cliente IN (
    SELECT id_cliente
    FROM pedidos
    GROUP BY id_cliente
    HAVING COUNT(*) > 10
);
```

La subconsulta primero encuentra los id_cliente con más de 10 pedidos, y el UPDATE cambia la categoría solo de esos clientes. Esta técnica es muy usada en el trabajo real: actualizar una tabla con base en calculos hechos sobre otra tabla.

UPDATE también puede usar JOIN en algunos sistemas de bases de datos (MySQL lo permite) para actualizar una tabla con datos de otra directamente:

```
UPDATE pedidos p
JOIN clientes c ON p.id_cliente = c.id_cliente
SET p.estado = 'prioridad'
WHERE c.categoria = 'VIP';
```

Aquí estamos diciendo: "a todos los pedidos de clientes VIP, ponles estado prioridad". Es elegante y muy útil, aunque debes verificar que tu sistema de base de datos soporte esta sintaxis.

Ahora hablemos del tema serio: el peligro del UPDATE sin WHERE.

La historia de terror del UPDATE sin WHERE

Esto no es una leyenda urbana. Le ha pasado a programadores reales, en empresas reales, con consecuencias reales. Te cuento una historia típica, con nombres cambiados pero hechos muy parecidos a casos documentados:

Era viernes por la tarde. Carlos, un desarrollador con dos años de experiencia, recibió una petición urgente del área de ventas: "El cliente con id 452 cambió su correo, actualízalo por favor, urge para una factura". Carlos abrió su herramienta de base de datos, escribió la instrucción... y se le olvidó el WHERE. En lugar de esto:

```
UPDATE clientes SET correo = 'nuevo@empresa.com' WHERE id_cliente = 452;
```

Ejecutó esto:

```
UPDATE clientes SET correo = 'nuevo@empresa.com';
```

Sin WHERE. Sin condición. SQL hizo exactamente lo que le pidieron: actualizó TODOS los correos de TODOS los clientes al mismo valor. En menos de un segundo, 48,000 clientes quedaron con el mismo correo electrónico. La base de datos no preguntó "¿estás seguro?". No mostró una advertencia. Simplemente obedeció.

El lunes siguiente, el sistema de facturación envió 48,000 facturas al mismo correo. El área de soporte colapsó. Tardaron tres días en restaurar la información desde un respaldo, y aun así perdieron los cambios del fin de semana. Carlos no perdió su trabajo, pero pasó meses siendo "el del UPDATE" en todas las reuniones.

La lección es brutal pero simple: SQL no tiene botón de deshacer. Un UPDATE sin WHERE afecta a todas las filas de la tabla. Siempre. Sin excepción. Y no hay forma de revertirlo a menos que tengas un respaldo o estés dentro de una transacción.

Como protegerte: las tres reglas de oro

Regla de oro número uno: escribe el WHERE primero. Antes de escribir SET, escribe la condición. O mejor aun, escribe primero un SELECT con ese WHERE para ver exactamente que filas vas a afectar:

```
-- Paso 1: mira que filas vas a tocar
SELECT id_cliente, nombre, correo
FROM clientes
WHERE id_cliente = 452;

-- Paso 2: si el SELECT mostro la fila correcta, ahora si haz el UPDATE
UPDATE clientes
SET correo = 'nuevo@empresa.com'
WHERE id_cliente = 452;
```

Este habito te salva la vida. El SELECT es tu vista previa: si el SELECT devuelve 48,000 filas en lugar de una, ya sabes que tu WHERE está mal antes de hacer daño.

Regla de oro número dos: usa transacciones para cambios importantes. Una transacción es como un modo de prueba: haces los cambios, los revisas, y solo si todo está bien los confirmas. Si algo salio mal, los deshaces:

```
START TRANSACTION;

UPDATE empleados
SET sueldo = sueldo * 1.15
WHERE departamento = 'ventas';

-- Revisa el resultado antes de confirmar
SELECT nombre, sueldo FROM empleados WHERE departamento = 'ventas';

-- Si todo esta bien:
COMMIT;

-- Si algo salio mal, en lugar de COMMIT usas:
-- ROLLBACK;
```

Mientras no ejecutes COMMIT, puedes usar ROLLBACK para deshacer todo. Es tu red de seguridad. En cambios masivos, siempre trabaja dentro de una transacción.

Regla de oro número tres: respaldo antes de cambios masivos. Si vas a actualizar miles de filas, asegurate de que exista un respaldo reciente de la tabla. Muchos administradores hacen una copia de la tabla antes de operaciones grandes:

```
CREATE TABLE clientes_respaldo AS
SELECT * FROM clientes;
```

Así, si algo sale mal, tienes la tabla original guardada con otro nombre. Es una práctica simple que ha salvado más de un empleo.

Un consejo extra para herramientas graficas: muchos clientes de base de datos tienen un "modo seguro" que impide ejecutar UPDATE o DELETE sin WHERE. Activalo si tu herramienta lo ofrece. Es como el seguro del cinturon de seguridad: no lo notas hasta que lo necesitas.

CLASE 18: DELETE - borrar filas, con mucho cuidado

La instrucción DELETE elimina filas de una tabla. Su sintaxis es parecida a la de UPDATE:

```
DELETE FROM clientes
WHERE id_cliente = 30;
```

Eso borra la fila del cliente 30 y nada más. Igual que con UPDATE, el WHERE es lo que te mantiene a salvo. Y la misma historia de terror aplica aquí, multiplicada: un DELETE sin WHERE borra TODA la tabla en un segundo.

```
-- ESTO BORRA TODA LA TABLA. NO LO HAGAS sin estar 100 por ciento seguro.
DELETE FROM clientes;
```

La diferencia con UPDATE es que DELETE no deja rastro parcial: las filas desaparecen. Por eso la disciplina aquí debe ser aun mayor. Aplica las mismas tres reglas de oro: SELECT primero para verificar, transacción para cambios importantes, respaldo antes de borrados masivos.

Puedes borrar varias filas con una condición más amplia. Por ejemplo, eliminar los pedidos cancelados hace más de dos años:

```
DELETE FROM pedidos
WHERE estado = 'cancelado'
  AND fecha < '2023-01-01';
```

Y también puedes usar subconsultas en el DELETE, igual que en UPDATE:

```
DELETE FROM pedidos
WHERE id_cliente IN (
  SELECT id_cliente
  FROM clientes
  WHERE activo = 0
);
```

Esto borra todos los pedidos de clientes inactivos. Poderoso, y por lo mismo peligroso: verifica siempre con SELECT primero que filas caen dentro de la condición.

DELETE contra TRUNCATE: no son lo mismo

Existe otra instrucción para vaciar una tabla: TRUNCATE. A primera vista hace lo mismo que DELETE sin WHERE, pero hay diferencias importantes que debes conocer:

```
TRUNCATE TABLE pedidos;
```

Diferencia uno: velocidad. TRUNCATE es mucho más rápido porque no borra fila por fila, sino que libera el espacio de la tabla de golpe. En tablas con millones de filas la diferencia es enorme.

Diferencia dos: TRUNCATE reinicia los contadores AUTO_INCREMENT. Si tu tabla tenía el último id en 5000, después de un TRUNCATE el siguiente registro empezara de nuevo en 1. DELETE, en cambio, no reinicia el contador: el siguiente id seguiría en 5001.

Diferencia tres: TRUNCATE no se puede usar dentro de una transacción en algunos sistemas (en MySQL sí se puede en ciertas condiciones, pero en general es más riesgoso), y no activa disparadores (triggers) asociados a la tabla. DELETE sí respeta las transacciones y sí dispara los triggers.

Diferencia cuatro: TRUNCATE no admite WHERE. Es todo o nada: vacía la tabla completa o no hace nada.

Cuando usar cada uno: usa DELETE cuando necesites borrar filas específicas o quieras la protección de una transacción. Usa TRUNCATE cuando estes seguro de que quieres vaciar la tabla completa, por ejemplo al limpiar una tabla temporal o al reiniciar datos de prueba.

El borrado lógico: la práctica profesional

Aquí viene un secreto del mundo profesional: en los sistemas serios, casi nunca se borra de verdad. En lugar de DELETE físico, se usa el borrado lógico.

El borrado lógico consiste en agregar una columna como "activo" o "eliminado" a la tabla, y en lugar de borrar la fila, simplemente la marcas como inactiva:

```
-- En lugar de borrar al cliente, lo desactivamos
UPDATE clientes
SET activo = 0
WHERE id_cliente = 30;
```

Y todas tus consultas normales filtran los inactivos:

```
SELECT * FROM clientes WHERE activo = 1;
```

Por qué hacen esto las empresas? Por varias razones de peso. Primera: el historial. Si borras un cliente, pierdes para siempre la información de sus pedidos pasados, sus pagos, sus reclamos. Con borrado lógico, el historial se conserva. Segunda: los errores se pueden deshacer. Si desactivaste al cliente equivocado, un simple UPDATE lo reactiva. Tercera: las auditorías y las leyes. En muchos países las empresas están obligadas a conservar registros por años; borrar físicamente puede ser incluso ilegal en ciertos sectores como banca o salud.

El borrado físico con DELETE se reserva para casos específicos: datos de prueba, información duplicada por error, o cuando una regulación de privacidad te obliga a eliminar datos de verdad (como el derecho al olvido).

Mi recomendación: en tus proyectos, agrega la columna "activo" desde el diseño inicial. Te va a ahorrar muchos dolores de cabeza.

Restricciones de llave foránea al borrar

Aquí es donde DELETE se pone interesante y donde las relaciones entre tablas (que veremos a fondo en las clases 19 y 20) empiezan a protegerte.

Imagina que la tabla pedidos tiene una llave foránea que apunta a clientes. Eso significa que cada pedido debe pertenecer a un cliente que exista. Ahora intenta borrar un cliente que tiene pedidos:

```
DELETE FROM clientes WHERE id_cliente = 12;
-- ERROR: no se puede borrar porque el cliente 12 tiene pedidos relacionados
```

La base de datos te lo impide. Esa es la integridad referencial trabajando para ti: no permite que queden pedidos "huerfanos", es decir, pedidos que apuntan a un cliente que ya no existe. Sin esta protección, tu base de datos se llenaría de datos rotos e inconsistentes.

Pero a veces si quieres que el borrado se propague. Para eso existen las reglas ON DELETE, que se definen al crear la llave foránea:

ON DELETE RESTRICT (o no especificar nada): es el comportamiento por defecto. Impide borrar el registro padre si tiene hijos relacionados. Es la opción más segura y la recomendada en la mayoría de los casos.

ON DELETE CASCADE: borra automáticamente los registros hijos cuando se borra el padre. Si borras un cliente, se borran todos sus pedidos automáticamente.

```
CREATE TABLE pedidos (  
    id_pedido INT PRIMARY KEY AUTO_INCREMENT,  
    fecha DATE,  
    total DECIMAL(10,2),  
    id_cliente INT,  
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente)  
        ON DELETE CASCADE  
);
```

CASCADE es comodo pero peligroso: un solo DELETE puede borrar miles de filas en varias tablas sin que te des cuenta. Usalo solo cuando estes seguro de que los datos hijos no tienen valor sin el padre. Un ejemplo razonable: los renglones de detalle de un pedido no tienen sentido si el pedido desaparece.

ON DELETE SET NULL: cuando borras el padre, la llave foránea de los hijos se pone en NULL. El pedido se queda, pero ya no apunta a ningun cliente. Útil cuando quieres conservar el historial pero desvincularlo.

ON DELETE SET DEFAULT: parecido, pero pone un valor por defecto en lugar de NULL.

La regla práctica: por defecto usa RESTRICT y borra primero los hijos y luego el padre de forma explícita y controlada. Solo usa CASCADE cuando el diseño lo justifique claramente y todo el equipo lo entienda.

CLASE 19: Llaves primarias - la identidad única de cada fila

Una llave primaria (PRIMARY KEY) es la columna, o combinación de columnas, que identifica de forma única cada fila de una tabla. Es como la CURP de una persona o el número de serie de un producto: no puede haber dos filas con la misma llave primaria, y ninguna fila puede tenerla vacía.

Piensalo así: en una tabla de clientes puede haber dos personas que se llamen "Maria Garcia". El nombre no sirve para distinguirlas. El teléfono puede cambiar. El correo puede cambiar. Necesitas algo que sea único y permanente para decir "esta fila es el cliente 12 y no hay otro igual". Ese algo es la llave primaria.

Las llaves primarias cumplen tres reglas estrictas: primera, cada valor debe ser único en toda la tabla. Segunda, no puede ser NULL, es decir, toda fila debe tener su llave. Tercera, idealmente no debería cambiar nunca una vez asignada.

Se declaran al crear la tabla:

```
CREATE TABLE clientes (  
    id_cliente INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    correo VARCHAR(100),  
    telefono VARCHAR(20)  
);
```

O al final de la definición, lo cual es útil cuando la llave abarca varias columnas:

```
CREATE TABLE clientes (  
    id_cliente INT,  
    nombre VARCHAR(100),  
    correo VARCHAR(100),  
    PRIMARY KEY (id_cliente)  
);
```

Llave natural contra llave sustituta

Hay dos escuelas para elegir llaves primarias, y vale la pena entenderlas porque es una de las decisiones de diseño más discutidas en bases de datos.

La llave natural es un dato real del mundo que ya es único por sí mismo. Ejemplos: la CURP de una persona, el número de pasaporte, el código ISBN de un libro, la placa de un auto. La ventaja es que tiene significado: ves el valor y sabes de qué se trata. La desventaja es que los datos del mundo real cambian o tienen problemas: una persona puede tener dos CURP por error administrativo, un libro puede reeditarse con otro ISBN, y algunos datos que parecen únicos resultan no serlo.

La llave sustituta (o subrogada) es un número inventado por la base de datos que no significa nada en el mundo real: simplemente 1, 2, 3, 4... Su única misión es identificar la fila. No tiene significado, y esa es precisamente su virtud: como no significa nada, nunca hay razón para cambiarla.

La práctica profesional moderna favorece claramente las llaves sustitutas. La razón es simple y poderosa: los datos reales cambian, los números inventados no. Si usas el correo como llave primaria y el cliente cambia de correo, tienes que actualizar la llave en esa tabla y en todas las tablas relacionadas. Es un dolor de cabeza evitable.

La regla de oro: nunca uses como llave primaria un dato que pueda cambiar. Nombres, correos, telefonos, direcciones: todo eso cambia. Un id numerico no.

AUTO_INCREMENT: el contador automatico

La forma más comun de crear llaves sustitutas es con AUTO_INCREMENT (en MySQL) o IDENTITY (en SQL Server) o SERIAL (en PostgreSQL). Le dices a la base de datos: "encargate tu de generar el siguiente número":

```
CREATE TABLE clientes (  
    id_cliente INT PRIMARY KEY AUTO_INCREMENT,  
    nombre VARCHAR(100),  
    correo VARCHAR(100),  
    telefono VARCHAR(20)  
);
```

Ahora, cuando insertes un cliente, no necesitas dar el id: la base de datos asigna automáticamente el siguiente número disponible:

```
INSERT INTO clientes (nombre, correo, telefono)  
VALUES ('Juan Perez', 'juan@correo.com', '555-111-2222');  
-- La base de datos asigna id_cliente = 1 automaticamente  
  
INSERT INTO clientes (nombre, correo, telefono)  
VALUES ('Maria Lopez', 'maria@correo.com', '555-333-4444');  
-- La base de datos asigna id_cliente = 2 automaticamente
```

El contador siempre avanza, nunca se repite. Si borras al cliente 2, el siguiente insert no reutiliza el 2: sigue con el 3. Esto es intencional y es bueno: reutilizar números borrados puede causar confusiones terribles en historiales y auditorias.

Puedes saber que id se generó en tu último INSERT con la función LAST_INSERT_ID() en MySQL. Esto es útil cuando insertas un cliente y enseguida necesitas crearle un pedido que apunte a ese cliente.

Mencion especial: los UUID

Existe otra estrategia para llaves primarias: el UUID (identificador único universal). Es una cadena larga de letras y números generada aleatoriamente, algo como "550e8400-e29b-41d4-a716-446655440000". Su gracia es que puedes generarlo en cualquier computadora del mundo sin coordinarte con nadie y la probabilidad de que se repita es prácticamente cero.

Los UUID se usan cuando tienes sistemas distribuidos: varias sucursales, varios servidores o una app móvil que crea registros sin conexión y luego los sincroniza. En esos casos un contador AUTO_INCREMENT no funciona, porque dos servidores podrían generar el mismo número.

Sus desventajas: ocupan más espacio, son más lentos para buscar y ordenar, y son horribles de leer para un humano. Para la gran mayoría de los proyectos, AUTO_INCREMENT es suficiente y más simple. Conoce los UUID, usalos cuando el problema los pida, no por moda.

Llaves primarias compuestas

A veces una sola columna no basta para identificar una fila, y se usan dos o más columnas juntas como llave primaria compuesta. El ejemplo clásico es una tabla intermedia (la verás en la clase 20): la combinación de id_pedido e id_producto identifica cada renglon del detalle de un pedido.

```
CREATE TABLE pedido_detalle (  
    id_pedido INT,  
    id_producto INT,  
    cantidad INT,  
    precio DECIMAL(10,2),  
    PRIMARY KEY (id_pedido, id_producto)  
);
```

Aquí ni el id_pedido solo ni el id_producto solo son únicos (un pedido tiene varios productos, y un producto aparece en varios pedidos), pero la combinación de ambos si es única: el producto 7 aparece una sola vez en el pedido 105.

Las llaves compuestas son correctas y utiles en tablas intermedias. Pero como regla general, si puedes usar una llave simple con AUTO_INCREMENT, hazlo: es más fácil de manejar, de relacionar y de entender.

CLASE 20: Llaves foráneas - conectar tablas, construir relaciones

Si la llave primaria es la identidad de una fila, la llave foránea (FOREIGN KEY) es el puente entre tablas. Una llave foránea es una columna en una tabla que guarda el valor de la llave primaria de otra tabla. Gracias a ese puente, la base de datos sabe que filas están relacionadas.

Ejemplo concreto: la tabla pedidos tiene una columna id_cliente. Esa columna es llave foránea que apunta a la llave primaria id_cliente de la tabla clientes. Cuando un pedido tiene id_cliente = 12, significa "este pedido lo hizo el cliente 12". Así de simple, así de poderoso.

Se declara al crear la tabla:

```
CREATE TABLE pedidos (  
    id_pedido INT PRIMARY KEY AUTO_INCREMENT,  
    fecha DATE,  
    total DECIMAL(10,2),  
    estado VARCHAR(20),  
    id_cliente INT,  
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente)  
);
```

Lee la última línea así: "la columna id_cliente es llave foránea que hace referencia a la columna id_cliente de la tabla clientes". Desde ese momento, la base de datos vigila esa relación.

Integridad referencial: el guardián automatico

La integridad referencial es el conjunto de reglas que la base de datos aplica automáticamente gracias a las llaves foráneas. Es como un guardia que revisa cada INSERT, UPDATE y DELETE para que los datos nunca queden rotos. Estas son sus reglas:

Regla uno: no puedes insertar un pedido con un id_cliente que no exista. Si intentas crear un pedido para el cliente 999 y ese cliente no existe, la base de datos rechaza la operación con un error. Así se evitan los pedidos huérfanos.

Regla dos: no puedes borrar un cliente que tenga pedidos (con la regla RESTRICT, que es la de por defecto). Ya lo viste en la clase 18: la base de datos te detiene.

Regla tres: no puedes cambiar el id de un cliente si hay pedidos que apuntan a él, a menos que definas una regla ON UPDATE para ese caso (CASCADE, SET NULL, etc., igual que con ON DELETE).

Sin llaves foráneas, nada de esto existe: podrias borrar clientes a lo loco y dejar pedidos apuntando a la nada. Con llaves foráneas, la base de datos cuida la coherencia por ti, las 24 horas, sin que tengas que acordarte.

Un detalle importante: para que la llave foránea funcione, los tipos de datos deben coincidir. Si `id_cliente` es INT en clientes, debe ser INT en pedidos. No puedes relacionar un INT con un VARCHAR.

También puedes agregar una llave foránea a una tabla que ya existe con ALTER TABLE:

```
ALTER TABLE pedidos
ADD CONSTRAINT fk_pedidos_clientes
FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente);
```

Darle un nombre a la restricción (`fk_pedidos_clientes`) es buena práctica: si algun día necesitas eliminarla o modificarla, la puedes referenciar por su nombre.

Los tres tipos de relaciones

Todas las relaciones entre tablas caen en tres tipos. Entenderlos es entender el diseño de bases de datos.

Relacion uno a muchos (1-N): la más comun de todas. Un cliente puede tener muchos pedidos, pero cada pedido pertenece a un solo cliente. Un autor puede escribir muchos libros, pero cada libro (en este modelo simple) tiene un autor principal. Se implementa poniendo la llave foránea en la tabla del lado "muchos": la tabla pedidos lleva `id_cliente`.

Relacion uno a uno (1-1): un registro de una tabla se relaciona con un solo registro de otra, y viceversa. Ejemplo: cada empleado tiene una sola credencial de acceso, y cada credencial pertenece a un solo empleado. Se implementa con una llave foránea con restricción UNIQUE, o compartiendo la misma llave primaria en ambas tablas:

```
CREATE TABLE credenciales (
    id_empleado INT PRIMARY KEY,
    usuario VARCHAR(50) UNIQUE,
    clave VARCHAR(100),
    FOREIGN KEY (id_empleado) REFERENCES empleados(id_empleado)
);
```

Aquí `id_empleado` es a la vez llave primaria y foránea: eso garantiza el uno a uno. Las relaciones 1-1 se usan para separar datos sensibles, para dividir tablas muy anchas o cuando una parte de los datos es opcional.

Relacion muchos a muchos (N-M): un estudiante se inscribe en muchas materias, y cada materia tiene muchos estudiantes. Un producto aparece en muchos pedidos, y cada pedido contiene muchos productos. Este es el tipo más interesante porque no se puede implementar con una simple llave foránea: necesitas una tabla intermedia.

La tabla intermedia: resolviendo muchos a muchos

La tabla intermedia (también llamada tabla puente o tabla de union) convierte una relación N-M en dos relaciones 1-N. Para pedidos y productos, la tabla intermedia se llama pedido_detalle: cada fila dice "el pedido 105 incluye 2 unidades del producto 7".

```
CREATE TABLE pedido_detalle (  
    id_pedido INT,  
    id_producto INT,  
    cantidad INT,  
    precio_unitario DECIMAL(10,2),  
    PRIMARY KEY (id_pedido, id_producto),  
    FOREIGN KEY (id_pedido) REFERENCES pedidos(id_pedido),  
    FOREIGN KEY (id_producto) REFERENCES productos(id_producto)  
);
```

Observa el diseño: tiene dos llaves foráneas, una hacia pedidos y otra hacia productos. Su llave primaria es compuesta (id_pedido, id_producto), porque lo que identifica cada renglon es la combinación de pedido y producto. Además puede llevar columnas propias, como cantidad y precio_unitario, que son datos de la relación misma: no pertenecen ni al pedido ni al producto, sino al hecho de que ese producto está en ese pedido.

Con esta estructura puedes responder preguntas poderosas: que productos contiene el pedido 105, en cuantos pedidos aparece el producto 7, cual es el producto más vendido. Todo con JOINS, que ya sabes manejar de unidades anteriores... y si aun no los dominas, esta es la motivación perfecta para repasarlos, porque las llaves foráneas existen precisamente para hacer JOINS con sentido.

El patrón de la tabla intermedia aparece en todas partes: estudiantes y materias (inscripciones), actores y películas (reparto), usuarios y roles (asignaciones), etiquetas y artículos. Cuando detectes un "muchos a muchos" en tu diseño, ya sabes qué necesitas una tabla en medio.

Errores comunes de la unidad

Error uno: olvidar el WHERE en UPDATE o DELETE. Ya lo sabes, pero vale repetirlo hasta el cansancio: es el error número uno en SQL en el mundo real. SELECT primero, siempre.

Error dos: usar como llave primaria un dato que cambia, como el correo o el teléfono. Tarde o temprano cambia y el problema se multiplica por todas las tablas relacionadas.

Error tres: crear tablas relacionadas sin declarar las llaves foráneas "porque es más rápido". Al principio parece que ahorras tiempo; después pagas con datos inconsistentes que nadie sabe como limpiar. Declara siempre tus llaves foráneas.

Error cuatro: usar ON DELETE CASCADE sin pensarlo. Borrar un cliente y descubrir que se borraron también sus 200 pedidos, sus pagos y su historial es una experiencia que no quieres vivir. CASCADE solo donde tenga sentido.

Error cinco: tipos de datos distintos entre la llave primaria y la foránea. Un INT no se relaciona con un VARCHAR. La base de datos te dara error al crear la restricción, y con razón.

Error seis: intentar insertar un hijo antes que el padre. No puedes crear el pedido del cliente 12 si el cliente 12 no existe todavía. El orden correcto es: primero el padre, luego el hijo.

Error siete: borrar al padre antes que a los hijos con RESTRICT activo, y frustrarse con el error. No es un error de la base de datos: es la protección funcionando. Borra primero los hijos o cambia la estrategia.

Buenas prácticas de la unidad

Práctica uno: nunca ejecutes UPDATE ni DELETE sin WHERE, salvo que deliberadamente quieras afectar toda la tabla y lo hayas pensado dos veces.

Práctica dos: antes de un UPDATE o DELETE importante, ejecuta el SELECT equivalente para ver exactamente que filas vas a tocar.

Práctica tres: usa transacciones (START TRANSACTION, COMMIT, ROLLBACK) en cambios masivos o delicados. Revisa antes de confirmar.

Práctica cuatro: haz respaldo de las tablas antes de operaciones grandes. Una tabla de respaldo con CREATE TABLE ... AS SELECT ... toma segundos y puede salvarte días.

Práctica cinco: prefiere el borrado lógico (columna activo) sobre el borrado físico en sistemas reales. Conserva el historial.

Práctica seis: usa llaves sustitutas con AUTO_INCREMENT como llaves primarias, y nunca uses datos que cambian como llave.

Práctica siete: declara todas tus llaves foráneas. Son tu seguro automatico contra datos rotos.

Práctica ocho: modela los muchos a muchos con tabla intermedia desde el diseño inicial, con llave primaria compuesta y sus dos llaves foráneas.

Práctica nueve: nombra tus restricciones de llave foránea (fk_tabla_referencia) para poder administrarlas después.

Práctica diez: documenta tus relaciones. Un diagrama sencillo de que tabla se relaciona con cual vale más que mil comentarios.

Checklist del profesional antes de modificar datos

Con el tiempo, los buenos programadores desarrollan un checklist mental que ejecutan en automatico antes de cualquier UPDATE o DELETE. Aquí esta, paso por paso, para que lo adoptes desde ahora:

Paso uno: estoy en la base de datos correcta? Parece obvio, pero ejecutar un UPDATE en producción creyendo que estas en pruebas es un clásico. Verifica el nombre de la base de datos y del servidor antes de tocar nada.

Paso dos: tengo respaldo reciente? Para cambios masivos, la respuesta debe ser si. Si no hay respaldo, crealo tú mismo con CREATE TABLE respaldo AS SELECT antes de empezar.

Paso tres: mi WHERE esta probado con SELECT? Ejecuta el SELECT equivalente y cuenta las filas. El número debe coincidir con lo que esperas modificar. Si esperabas una fila y el SELECT devuelve mil, detente.

Paso cuatro: estoy dentro de una transacción? Para cambios delicados o masivos, usa START TRANSACTION. Te da la oportunidad de revisar y de usar ROLLBACK si algo sale mal.

Paso cinco: que efectos secundarios hay? Piensa en llaves foráneas con CASCADE, en triggers asociados a la tabla y en aplicaciones que lean esos datos. Un cambio en la base de datos nunca vive aislado.

Paso seis: verifique el resultado después de ejecutar. Un SELECT posterior confirma que el cambio quedó como esperabas y que nada más se afecto.

Memoriza este checklist. Los seis pasos juntos toman menos de un minuto y te ahorran los desastres que toman días en repararse.

Preguntas frecuentes de la unidad

Puedo usar UPDATE sin WHERE alguna vez? Si, pero solo cuando deliberadamente quieres cambiar toda la tabla, por ejemplo poner en cero una columna temporal o activar una bandera para todos los registros. Aun así, hazlo dentro de una transacción y con respaldo. La regla no es "nunca", es "nunca por accidente".

Qué pasa si olvido el WHERE y no tengo respaldo ni transacción? En la mayoría de los casos, los datos se perdieron. Algunas herramientas y servicios de nube ofrecen recuperación a un punto en el tiempo, pero no cuentes con eso. Por eso la prevención es todo.

DELETE o borrado lógico: cual elijo? Borrado lógico por defecto en sistemas reales. DELETE físico solo para datos de prueba, duplicados por error o cuando la ley te obliga a eliminar. Cuando dudes, desactiva en lugar de borrar.

AUTO_INCREMENT o UUID? AUTO_INCREMENT para el 90 por ciento de los proyectos: simple, rápido y legible. UUID cuando tengas varios servidores generando registros sin coordinación, apps móviles offline o necesidad de ids impredecibles por seguridad.

Puedo cambiar una llave primaria después de crearla? Técnicamente si, pero es una de las operaciones más dolorosas en bases de datos porque arrastra a todas las tablas relacionadas. Por eso se elige bien desde el diseño y se prefieren llaves sustitutas que nunca necesitan cambiar.

Por qué mi DELETE me da error de llave foránea si "yo se lo que hago"? Porque la base de datos no confía en tu memoria: confía en sus reglas. El error te está diciendo que hay filas hijas relacionadas. Bórralas primero de forma explícita, usa borrado lógico o reconsidera si de verdad quieres borrar.

Es normal tener muchas tablas intermedias? Si. Una base de datos bien diseñada suele tener varias tablas puente: son la señal de que los muchos a muchos están modelados correctamente en lugar de escondidos en columnas con listas separadas por comas, que es uno de los peores anti-patrones en SQL.

Glosario de terminos clave de la unidad

Llave primaria (PRIMARY KEY): columna o conjunto de columnas que identifica de forma única cada fila de una tabla. No se repite, no admite NULL y no debería cambiar.

Llave foránea (FOREIGN KEY): columna de una tabla que guarda valores de la llave primaria de otra tabla, creando un puente entre ambas.

Integridad referencial: conjunto de reglas automaticas que garantizan que las relaciones entre tablas siempre sean validas: sin hijos huérfanos ni referencias rotas.

Llave natural: dato real del mundo (CURP, ISBN, placa) usado como llave primaria. Tiene significado, pero puede cambiar o duplicarse.

Llave sustituta: número inventado por la base de datos (1, 2, 3...) sin significado en el mundo real, usado como llave primaria. Es la práctica recomendada.

AUTO_INCREMENT: mecanismo que genera automáticamente el siguiente número de una llave primaria sustituta en cada INSERT.

UUID: identificador único universal, cadena larga aleatoria útil cuando varios sistemas generan registros sin coordinarse.

Tabla intermedia (puente): tabla que resuelve una relación muchos a muchos con dos llaves foráneas y llave primaria compuesta.

Borrado lógico: técnica de marcar filas como inactivas con una columna activo en lugar de eliminarlas físicamente, conservando el historial.

Borrado físico: eliminación real de filas con DELETE. Irreversible sin respaldo.

Transacción: bloque de operaciones que se confirman juntas con COMMIT o se deshacen juntas con ROLLBACK. Es la red de seguridad para cambios delicados.

CASCADE: regla que propaga automáticamente un borrado o actualización del registro padre a sus hijos relacionados.

RESTRICT: regla que impide borrar o modificar un registro padre mientras tenga hijos relacionados. Es el comportamiento seguro por defecto.

Registro huérfano: fila hija cuya llave foránea apunta a un padre que ya no existe. Las llaves foráneas existen para impedirlos.

Ejemplos paso a paso

Vamos a trabajar con un escenario completo: la tienda en línea "La Esquina Digital". Primero creamos las tablas con sus llaves primarias y foráneas, luego practicamos UPDATE y DELETE de forma segura, y al final intentaremos violar la integridad referencial para ver como la base de datos nos protege.

Ejemplo 1: Crear las tablas con llaves primarias y foráneas

Empezamos creando la tabla clientes con su llave primaria AUTO_INCREMENT y una columna para borrado lógico:

```
CREATE TABLE clientes (  
    id_cliente INT PRIMARY KEY AUTO_INCREMENT,  
    nombre VARCHAR(100) NOT NULL,  
    correo VARCHAR(100) UNIQUE,  
    telefono VARCHAR(20),  
    ciudad VARCHAR(50),  
    activo TINYINT DEFAULT 1  
);
```

Explicación: id_cliente es la llave primaria sustituta que se genera sola. El correo lleva UNIQUE porque no queremos dos clientes con el mismo correo, pero no es la llave primaria: si el cliente cambia de correo, solo se actualiza ese dato sin romper relaciones. La columna activo con valor 1 por defecto nos servira para el borrado lógico.

Ahora la tabla pedidos, con su llave foránea hacia clientes:

```
CREATE TABLE pedidos (  
    id_pedido INT PRIMARY KEY AUTO_INCREMENT,  
    fecha DATE,  
    total DECIMAL(10,2),  
    estado VARCHAR(20) DEFAULT 'pendiente',  
    id_cliente INT,  
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente)  
);
```

Explicación: cada pedido tiene su propio id, y la columna id_cliente es el puente hacia la tabla clientes. Desde que se crea esta restricción, la base de datos impide pedidos de clientes inexistentes y borrados de clientes con pedidos.

La tabla productos:

```
CREATE TABLE productos (  
    id_producto INT PRIMARY KEY AUTO_INCREMENT,  
    nombre VARCHAR(100) NOT NULL,  
    precio DECIMAL(10,2),  
    existencia INT DEFAULT 0  
);
```

Y la tabla intermedia pedido_detalle para la relación muchos a muchos entre pedidos y productos:

```
CREATE TABLE pedido_detalle (  
    id_pedido INT,  
    id_producto INT,  
    cantidad INT,  
    precio_unitario DECIMAL(10,2),  
    PRIMARY KEY (id_pedido, id_producto),  
    FOREIGN KEY (id_pedido) REFERENCES pedidos(id_pedido),  
    FOREIGN KEY (id_producto) REFERENCES productos(id_producto)  
);
```

Explicación: la llave primaria compuesta (id_pedido, id_producto) garantiza que un producto aparezca una sola vez por pedido. Las dos llaves foráneas conectan con pedidos y productos. Las columnas cantidad y precio_unitario son datos propios de la relación.

Ejemplo 2: Insertar datos de prueba en el orden correcto

Recuerda la regla: primero los padres, luego los hijos. Primero clientes y productos, después pedidos, al final el detalle:

```
INSERT INTO clientes (nombre, correo, telefono, ciudad)  
VALUES ('Ana Torres', 'ana@correo.com', '555-101-0101', 'CDMX'),  
      ('Luis Gomez', 'luis@correo.com', '555-202-0202', 'Guadalajara'),  
      ('Sofia Ruiz', 'sofia@correo.com', '555-303-0303', 'Monterrey');  
  
INSERT INTO productos (nombre, precio, existencia)  
VALUES ('Teclado mecanico', 899.00, 50),  
      ('Mouse inalambrico', 349.00, 100),  
      ('Monitor 24 pulgadas', 2999.00, 20);  
  
INSERT INTO pedidos (fecha, total, estado, id_cliente)  
VALUES ('2026-09-10', 1248.00, 'pendiente', 1),  
      ('2026-09-12', 349.00, 'enviado', 1),  
      ('2026-09-15', 2999.00, 'pendiente', 2);  
  
INSERT INTO pedido_detalle (id_pedido, id_producto, cantidad, precio_unitario)  
VALUES (1, 1, 1, 899.00),  
      (1, 2, 1, 349.00),  
      (2, 2, 1, 349.00),  
      (3, 3, 1, 2999.00);
```

Explicación: los clientes reciben ids 1, 2 y 3 automáticamente. Los pedidos apuntan a clientes que ya existen. El detalle apunta a pedidos y productos existentes. Si intentaras insertar el pedido antes que el cliente, la base de datos lo rechazaría.

Ejemplo 3: UPDATE seguro con verificación previa

La clienta Ana Torres (id 1) cambió su teléfono. Aplicamos la disciplina completa: primero el SELECT, luego el UPDATE:

```
-- Paso 1: verificar que existe y ver el dato actual
SELECT id_cliente, nombre, telefono
FROM clientes
WHERE id_cliente = 1;
```

La consulta devuelve una sola fila: Ana Torres con su teléfono actual. Confirmado que el WHERE es correcto, hacemos el cambio:

```
-- Paso 2: ejecutar el UPDATE
UPDATE clientes
SET telefono = '555-909-0909'
WHERE id_cliente = 1;
```

La base de datos responde algo como "1 row affected". Una fila afectada es exactamente lo que esperábamos. Si hubiera dicho "3 rows affected", sabríamos que algo salió mal.

```
-- Paso 3: comprobar el resultado
SELECT id_cliente, nombre, telefono
FROM clientes
WHERE id_cliente = 1;
```

Verificamos que el teléfono nuevo quedó guardado. Este ciclo de tres pasos (ver, ejecutar, comprobar) debe volverse tu reflejo automático.

Ejemplo 4: UPDATE múltiple y UPDATE calculado

La tienda decide dar 10 por ciento de descuento a todos los productos con existencia mayor a 40. Primero vemos cuáles son:

```
SELECT id_producto, nombre, precio, existencia
FROM productos
WHERE existencia > 40;
```

Supongamos que devuelve el teclado (existencia 50) y el mouse (existencia 100). Aplicamos el descuento calculado sobre el precio actual:

```
UPDATE productos
SET precio = precio * 0.90
WHERE existencia > 40;
```

Explicación: el nuevo precio se calcula a partir del anterior, fila por fila. El teclado de 899.00 queda en 809.10 y el mouse de 349.00 queda en 314.10. El monitor, con existencia 20, no se toca.

Ahora un UPDATE con subconsulta: marcar como "frecuente" a los clientes con más de un pedido:

```
UPDATE clientes
SET ciudad = ciudad
WHERE id_cliente IN (
    SELECT id_cliente
    FROM pedidos
    GROUP BY id_cliente
    HAVING COUNT(*) > 1
);
```

(En este ejemplo el SET es ilustrativo; en un caso real cambiarías una columna como categoría.) La subconsulta encuentra que el cliente 1 tiene 2 pedidos, así que solo el se actualiza. Explicación: primero resolvemos mentalmente la subconsulta para saber que ids devuelve, y luego el UPDATE actúa solo sobre ellos.

Ejemplo 5: DELETE con WHERE y borrado lógico

Un pedido de prueba se capturo por error y hay que eliminarlo. Verificamos primero:

```
SELECT id_pedido, fecha, total, estado
FROM pedidos
WHERE id_pedido = 2;
```

Confirmamos que es el pedido correcto y lo borramos. Pero ojo: este pedido tiene un renglon en pedido_detalle, y la llave foránea de pedido_detalle apunta a pedidos con RESTRICT por defecto. Si intentamos borrar el pedido directamente, la base de datos nos detiene. El orden correcto es borrar primero el hijo y luego el padre:

```
-- Paso 1: borrar el detalle (hijo)
DELETE FROM pedido_detalle
WHERE id_pedido = 2;

-- Paso 2: borrar el pedido (padre)
DELETE FROM pedidos
WHERE id_pedido = 2;
```

Explicación: con RESTRICT, la base de datos protege al hijo. Borramos el renglon del detalle primero y después el pedido. Este orden (hijos antes que padres) es la forma controlada de borrar en cascada manual.

Ahora el borrado lógico: el cliente Luis Gomez pide que lo den de baja, pero queremos conservar su historial de pedidos. No usamos DELETE; lo desactivamos:

```
UPDATE clientes
SET activo = 0
WHERE id_cliente = 2;
```

Y a partir de ahora, las consultas de clientes activos filtran:

```
SELECT id_cliente, nombre, correo
FROM clientes
WHERE activo = 1;
```

Luis desaparece de las listas, pero su pedido 3 sigue existiendo con su historial intacto. Si algun día regresa, un UPDATE lo reactiva. Explicación: el borrado lógico conserva la integridad del historial y permite deshacer el cambio.

Ejemplo 6: Intentos de violar la integridad y sus errores

Esta es la parte divertida: vamos a intentar romper las reglas a proposito para ver como la base de datos nos protege. Cada intento fallido es una demostración de que las llaves foráneas están haciendo su trabajo.

Intentó uno: crear un pedido para un cliente que no existe:

```
INSERT INTO pedidos (fecha, total, estado, id_cliente)
VALUES ('2026-09-19', 500.00, 'pendiente', 999);
-- ERROR: Cannot add or update a child row: a foreign key constraint fails
```

Explicación: el cliente 999 no existe en la tabla clientes, así que la base de datos rechaza el INSERT. Sin la llave foránea, ese pedido huérfano se habría guardado y nadie se habría dado cuenta hasta que un reporte saliera mal.

Intentó dos: borrar un cliente que tiene pedidos:

```
DELETE FROM clientes
WHERE id_cliente = 1;
-- ERROR: Cannot delete or update a parent row: a foreign key constraint fails
```

Explicación: Ana (cliente 1) tiene pedidos relacionados, y la regla RESTRICT impide borrarla. La base de datos no te deja romper la relación. Para borrarla tendrías que borrar primero sus pedidos y sus detalles, o usar borrado lógico.

Intentó tres: insertar un renglón de detalle con un producto inexistente:

```
INSERT INTO pedido_detalle (id_pedido, id_producto, cantidad, precio_unitario)
VALUES (1, 999, 1, 100.00);
-- ERROR: Cannot add or update a child row: a foreign key constraint fails
```

Explicación: el producto 999 no existe. La tabla intermedia también está protegida por sus dos llaves foráneas.

Intentó cuatro: duplicar una llave primaria:

```
INSERT INTO clientes (id_cliente, nombre, correo)
VALUES (1, 'Pedro Sanchez', 'pedro@correo.com');
-- ERROR: Duplicate entry '1' for key 'PRIMARY'
```

Explicación: el id 1 ya está ocupado por Ana. La llave primaria garantiza unicidad y rechaza el duplicado. Por eso conviene dejar que AUTO_INCREMENT genere los ids en lugar de asignarlos a mano.

Intentó cinco: repetir un producto en el mismo pedido (violar la llave compuesta):

```
INSERT INTO pedido_detalle (id_pedido, id_producto, cantidad, precio_unitario)
VALUES (1, 1, 5, 899.00);
-- ERROR: Duplicate entry '1-1' for key 'PRIMARY'
```

Explicación: el pedido 1 ya contiene el producto 1. La llave primaria compuesta (id_pedido, id_producto) impide el duplicado. Si el cliente quiere 5 teclados, lo correcto es hacer UPDATE a la cantidad existente, no insertar otro renglon:

```
UPDATE pedido_detalle
SET cantidad = 5
WHERE id_pedido = 1 AND id_producto = 1;
```

Cada uno de estos errores es la base de datos cuidando tus datos. Cuando diseñes tus propias tablas, declara las llaves y deja que el motor haga este trabajo por ti.

Ejemplo 7: UPDATE dentro de una transacción

La tienda va a subir el precio de todos los monitores un 5 por ciento. Es un cambio masivo y queremos red de seguridad. Usamos una transacción:

```
START TRANSACTION;

UPDATE productos
SET precio = precio * 1.05
WHERE nombre LIKE '%Monitor%';

-- Revisamos antes de confirmar
SELECT id_producto, nombre, precio
FROM productos
WHERE nombre LIKE '%Monitor%';
```

Si los precios se ven correctos, confirmamos:

```
COMMIT;
```

Si algo se ve mal (por ejemplo, el LIKE atrapo un producto que no era), deshacemos todo:

```
ROLLBACK;
```


Explicación: entre START TRANSACTION y COMMIT, los cambios son visibles solo para ti y reversibles. Es cómo escribir con lápiz antes de pasar a tinta. Para cambios que afectan muchas filas, trabaja siempre así.

Ejemplo 8: Tabla con ON DELETE CASCADE (y por qué usarlo con cuidado)

Creemos una tabla de sesiones de usuario donde si tiene sentido el borrado en cascada: las sesiones temporales de un cliente no valen nada si el cliente desaparece:

```
CREATE TABLE sesiones (  
    id_sesion INT PRIMARY KEY AUTO_INCREMENT,  
    id_cliente INT,  
    inicio DATETIME,  
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente)  
        ON DELETE CASCADE  
);
```

Insertamos una sesión para el cliente 3:

```
INSERT INTO sesiones (id_cliente, inicio)  
VALUES (3, '2026-09-19 10:00:00');
```

Si ahora borráramos al cliente 3 (suponiendo que no tuviera pedidos que lo protejan), su sesión se borraría sola automáticamente. Explicación: CASCADE propaga el borrado a los hijos. Es cómodo para datos temporales o dependientes, pero peligroso para datos con valor propio como pedidos o pagos. La decisión de usar CASCADE debe ser consciente, nunca por defecto.

Ejemplo 9: Consultar las relaciones con JOIN

Para cerrar los ejemplos, comprobemos que las relaciones sirven para algo: una consulta que une las tres tablas para mostrar el detalle completo de un pedido:

```
SELECT p.id_pedido, p.fecha, c.nombre AS cliente,  
       pr.nombre AS producto, d.cantidad, d.precio_unitario  
FROM pedidos p  
JOIN clientes c ON p.id_cliente = c.id_cliente  
JOIN pedido_detalle d ON d.id_pedido = p.id_pedido  
JOIN productos pr ON pr.id_producto = d.id_producto  
WHERE p.id_pedido = 1;
```

Explicación: los JOINS usan exactamente las llaves foráneas que declaramos. Sin esas relaciones, esta consulta sería imposible o requeriría lógica manual propensa a errores. Las llaves foráneas no solo protegen: también son el mapa que permite navegar entre tablas.

Práctica

Los ejercicios usan las tablas de "La Esquina Digital" creadas en los ejemplos: clientes, pedidos, productos y pedido_detalle. Haz cada ejercicio por tu cuenta antes de mirar la solución. Recuerda la disciplina: SELECT primero, transacción en cambios grandes.

Ejercicio 1: Actualización segura de un dato

La cliente Sofia Ruiz (id_cliente 3) se mudó a Puebla y cambió su teléfono a 555-707-0707. Actualiza ambos datos con un solo UPDATE, siguiendo el protocolo seguro de tres pasos (ver, ejecutar, comprobar).

Solución paso a paso:

Paso 1: verificamos la fila actual con SELECT.

```
SELECT id_cliente, nombre, telefono, ciudad
FROM clientes
WHERE id_cliente = 3;
```

Debe devolver una sola fila: Sofia Ruiz. Si devuelve más de una fila o ninguna, el WHERE está mal y hay que corregirlo antes de seguir.

Paso 2: ejecutamos el UPDATE con las dos columnas separadas por coma.

```
UPDATE clientes
SET telefono = '555-707-0707',
    ciudad = 'Puebla'
WHERE id_cliente = 3;
```

La respuesta esperada es "1 row affected". Una sola fila tocada, que es exactamente lo que queríamos.

Paso 3: comprobamos el resultado.

```
SELECT id_cliente, nombre, telefono, ciudad
FROM clientes
WHERE id_cliente = 3;
```

Ahora debe mostrar el teléfono nuevo y la ciudad Puebla. Los demás clientes no cambiaron. Ejercicio completado con disciplina profesional.

Ejercicio 2: UPDATE calculado con condición

La tienda quiere liquidar los productos con existencia menor a 30: bajarles el precio 20 por ciento. Escribe el SELECT de verificación y luego el UPDATE. Después comprueba cuantos productos se afectaron.

Solución paso a paso:

Paso 1: identificamos los candidatos con SELECT.

```
SELECT id_producto, nombre, precio, existencia
FROM productos
WHERE existencia < 30;
```

Con nuestros datos, el monitor (existencia 20) es el único que cumple. Sabemos de antemano que esperamos afectar exactamente una fila.

Paso 2: cómo es un cambio calculado sobre dinero, lo hacemos dentro de una transacción para tener red de seguridad.

```
START TRANSACTION;

UPDATE productos
SET precio = precio * 0.80
WHERE existencia < 30;

SELECT id_producto, nombre, precio
FROM productos
WHERE existencia < 30;
```

Revisamos: el monitor de 2999.00 debe quedar en 2399.20. Si el número es correcto, confirmamos con COMMIT. Si algo no cuadra, ROLLBACK y todo vuelve a su estado anterior.

Paso 3: verificación final fuera de la transacción.

```
SELECT COUNT(*) AS productos_en_liquidacion
FROM productos
WHERE precio < 2500 AND nombre LIKE '%Monitor%';
```

La moraleja del ejercicio: en cambios calculados sobre varias filas, la transacción te permite revisar el resultado antes de comprometerte. Es la diferencia entre un profesional y alguien que reza después de presionar Enter.

Ejercicio 3: Borrado controlado con hijos relacionados

Se capturo por error el pedido 3 (el monitor de Luis Gomez) y hay que eliminarlo por completo, incluyendo su renglon en pedido_detalle. El cliente 2 (Luis) tiene borrado lógico activo... espera, no: Luis esta desactivado (activo = 0) pero su fila sigue existiendo, así que la llave foránea sigue protegiendo. Elimina el pedido 3 de forma segura.

Solución paso a paso:

Paso 1: verificamos que filas vamos a tocar en ambas tablas.

```
SELECT * FROM pedidos WHERE id_pedido = 3;
SELECT * FROM pedido_detalle WHERE id_pedido = 3;
```

El primer SELECT debe devolver una fila (el pedido del monitor) y el segundo una fila (el renglon del detalle). Ya sabemos exactamente el alcance del borrado: dos filas en total.

Paso 2: borramos en el orden correcto, primero el hijo y luego el padre, dentro de una transacción.

```
START TRANSACTION;

DELETE FROM pedido_detalle
WHERE id_pedido = 3;

DELETE FROM pedidos
WHERE id_pedido = 3;

COMMIT;
```

Paso 3: comprobamos que ambas tablas quedaron limpias.

```
SELECT * FROM pedidos WHERE id_pedido = 3;  
SELECT * FROM pedido_detalle WHERE id_pedido = 3;
```

Ambas consultas deben devolver cero filas. El cliente 2 sigue existiendo (solo está desactivado con borrado lógico), así que su historial queda coherente: simplemente ya no tiene pedidos.

Nota importante: si hubieras intentado borrar el pedido antes que el detalle, la base de datos te habría detenido con el error de llave foránea. Ese error no es tu enemigo: es la protección funcionando.

Ejercicio 4: Diseñar una tabla con llave primaria y foránea

La tienda quiere guardar una dirección de envío por pedido, con la regla de que cada pedido tiene máximo una dirección de envío (relación uno a uno). Crea la tabla direcciones_envio con: id_pedido como llave primaria y foránea hacia pedidos, y las columnas calle, ciudad y codigo_postal.

Solución paso a paso:

Paso 1: analizamos el diseño. La relación es uno a uno: un pedido, una dirección. La técnica es usar la misma columna como PRIMARY KEY y FOREIGN KEY. Al ser llave primaria, no puede repetirse: eso garantiza que un pedido no tenga dos direcciones.

Paso 2: creamos la tabla.

```
CREATE TABLE direcciones_envio (  
    id_pedido INT PRIMARY KEY,  
    calle VARCHAR(150),  
    ciudad VARCHAR(50),  
    codigo_postal VARCHAR(10),  
    FOREIGN KEY (id_pedido) REFERENCES pedidos(id_pedido)  
);
```

Explicación de cada decisión: id_pedido es INT porque debe coincidir con el tipo de la llave primaria de pedidos. Es PRIMARY KEY para forzar el uno a uno. Es FOREIGN KEY para garantizar que la dirección pertenezca a un pedido real. Las demás columnas son datos descriptivos sin restricciones especiales.

Paso 3: probamos que funciona insertando una dirección válida.

```
INSERT INTO direcciones_envio (id_pedido, calle, ciudad, codigo_postal)
VALUES (1, 'Av. Reforma 222', 'CDMX', '06600');
```

Debe insertarse sin problema porque el pedido 1 existe.

Paso 4: probamos las protecciones. Intentar una segunda dirección para el mismo pedido debe fallar por llave primaria duplicada, e intentar una dirección para un pedido inexistente debe fallar por llave foránea.

```
INSERT INTO direcciones_envio (id_pedido, calle, ciudad, codigo_postal)
VALUES (1, 'Otra calle 1', 'CDMX', '06600');
-- ERROR: Duplicate entry '1' for key 'PRIMARY'

INSERT INTO direcciones_envio (id_pedido, calle, ciudad, codigo_postal)
VALUES (999, 'Calle falsa 123', 'CDMX', '06600');
-- ERROR: Cannot add or update a child row: a foreign key constraint fails
```

Ambos errores son el diseño defendiéndose solo. Si tus restricciones rechazan los datos malos en las pruebas, van a rechazarlos también en producción.

Ejercicio 5: Tabla intermedia para muchos a muchos

La tienda quiere vender con cupones de descuento: un pedido puede usar varios cupones y un cupon puede usarse en varios pedidos (relación muchos a muchos). Tienes la tabla cupones ya creada:

```
CREATE TABLE cupones (
  id_cupon INT PRIMARY KEY AUTO_INCREMENT,
  codigo VARCHAR(20) UNIQUE,
  descuento DECIMAL(5,2)
);
```

Crea la tabla intermedia pedido_cupon con llave primaria compuesta y sus dos llaves foráneas, inserta datos de ejemplo y demuestra que la integridad funciona.

Solución paso a paso:

Paso 1: creamos la tabla intermedia siguiendo el patrón: dos foráneas, llave compuesta.

```
CREATE TABLE pedido_cupon (
    id_pedido INT,
    id_cupon INT,
    PRIMARY KEY (id_pedido, id_cupon),
    FOREIGN KEY (id_pedido) REFERENCES pedidos(id_pedido),
    FOREIGN KEY (id_cupon) REFERENCES cupones(id_cupon)
);
```

Explicación: la llave compuesta impide aplicar el mismo cupon dos veces al mismo pedido. Cada foránea garantiza que ambos lados de la relación existan.

Paso 2: insertamos cupones y los relacionamos con pedidos.

```
INSERT INTO cupones (codigo, descuento)
VALUES ('BIENVENIDA10', 10.00),
      ('FRECUENTE15', 15.00);

INSERT INTO pedido_cupon (id_pedido, id_cupon)
VALUES (1, 1),
      (1, 2);
```

El pedido 1 usa dos cupones: bienvenida y frecuente. El cupon 1 podría usarse después en otro pedido: esa es la gracia del muchos a muchos.

Paso 3: verificamos la relación con un JOIN.

```
SELECT p.id_pedido, c.codigo, c.descuento
FROM pedido_cupon pc
JOIN pedidos p ON p.id_pedido = pc.id_pedido
JOIN cupones c ON c.id_cupon = pc.id_cupon
WHERE p.id_pedido = 1;
```

Debe devolver dos filas: los dos cupones del pedido 1. La tabla intermedia esta haciendo su trabajo de puente.

Paso 4: intentamos romperla y comprobamos la defensa.

```
INSERT INTO pedido_cupon (id_pedido, id_cupon)
VALUES (1, 1);
-- ERROR: Duplicate entry '1-1' for key 'PRIMARY'
```

No se puede aplicar dos veces el mismo cupon al mismo pedido. El diseño impide el error de negocio automáticamente, sin que tengas que programar esa validación en tu aplicación.

Ejercicio 6: Rescate de un UPDATE sin WHERE

Un compañero ejecutó esto por accidente en la tabla productos:

```
UPDATE productos SET existencia = 0;
```

Sin WHERE. Todos los productos quedaron con existencia cero. Tienes un respaldo de la tabla llamado productos_respaldo creado ayer con CREATE TABLE productos_respaldo AS SELECT * FROM productos. Explica como diagnosticar el daño y como repararlo, y que habrias hecho para prevenirlo.

Solución paso a paso:

Paso 1: diagnosticar el alcance del daño. No entres en panico: primero mide.

```
SELECT id_producto, nombre, existencia
FROM productos;
```

Si todas las filas muestran existencia 0, el daño es total en esa columna. Revisa también si otras columnas se afectaron (en este caso solo fue existencia).

Paso 2: verificar que el respaldo este completo y sea confiable.

```
SELECT COUNT(*) FROM productos;
SELECT COUNT(*) FROM productos_respaldo;

SELECT id_producto, nombre, existencia
FROM productos_respaldo;
```

Ambas tablas deben tener el mismo número de filas, y el respaldo debe mostrar las existencias originales (50, 100, 20). Solo restaura desde un respaldo que hayas verificado: un respaldo corrupto es peor que no tener respaldo.

Paso 3: reparar con UPDATE usando el respaldo como fuente, dentro de una transacción. La técnica es actualizar cada fila con el valor que le corresponde según su id:


```
START TRANSACTION;

UPDATE productos p
JOIN productos_respaldo r ON p.id_producto = r.id_producto
SET p.existencia = r.existencia;

SELECT id_producto, nombre, existencia FROM productos;

COMMIT;
```

Explicación: el JOIN empareja cada producto con su fila gemela del respaldo y restaura la existencia original. Revisamos el resultado antes del COMMIT. Si los números coinciden con el respaldo, confirmamos.

Paso 4: prevención. Tres medidas para que no vuelva a pasar. Primera: el compañero debió escribir primero el SELECT con el WHERE para ver que filas iba a tocar. Segunda: en cambios masivos, trabajar dentro de una transacción para poder hacer ROLLBACK. Tercera: activar el modo seguro del cliente SQL que bloquea UPDATE y DELETE sin WHERE. Y la lección cultural: los respaldos automatizados diarios no son opcionales, son el seguro de vida de la empresa.

Este ejercicio es el más importante de la unidad porque simula lo que pasa en la vida real. La diferencia entre un desastre y un susto de diez minutos es la disciplina preventiva.

Ejercicio 7 (reto): Auditoria de una base de datos mal diseñada

Un compañero te pasa el diseño de su primer proyecto y te pide que lo audites. Encuentra al menos cuatro problemas de diseño relacionados con esta unidad y propone la corrección para cada uno.

```

CREATE TABLE usuarios (
    correo VARCHAR(100) PRIMARY KEY,
    nombre VARCHAR(100),
    telefono VARCHAR(20)
);

CREATE TABLE ordenes (
    id_orden INT PRIMARY KEY AUTO_INCREMENT,
    fecha DATE,
    total DECIMAL(10,2),
    correo_usuario VARCHAR(100)
);

CREATE TABLE productos_orden (
    id INT PRIMARY KEY AUTO_INCREMENT,
    id_orden INT,
    id_producto INT,
    cantidad INT
);

```

Solución paso a paso:

Problema uno: el correo como llave primaria de usuarios. El correo cambia (la gente cambia de proveedor o de trabajo) y entonces habría que actualizar la llave en todas las tablas relacionadas. Corrección: agregar `id_usuario INT PRIMARY KEY AUTO_INCREMENT` y dejar el correo como `UNIQUE`.

Problema dos: la tabla `ordenes` no declara llave foránea hacia usuarios. La columna `correo_usuario` existe pero sin restricción, así que nada impide ordenes de usuarios inexistentes. Corrección: agregar `FOREIGN KEY (id_usuario) REFERENCES usuarios(id_usuario)`, usando el nuevo id numerico.

Problema tres: la tabla `productos_orden` tampoco declara llaves foráneas hacia ordenes ni hacia productos, y además permite el mismo producto dos veces en la misma orden. Corrección: quitar el id sustituto innecesario o conservarlo, pero lo esencial es declarar ambas foráneas y, si se quiere impedir duplicados, usar `PRIMARY KEY (id_orden, id_producto)`.

Problema cuatro: no hay columna de borrado lógico en ninguna tabla. Si borran un usuario con `DELETE`, sus ordenes quedan huérfanas (o el borrado falla y nadie entiende por qué). Corrección: agregar `activo TINYINT DEFAULT 1` a usuarios y preferir desactivar sobre borrar.

Si detectaste los cuatro problemas y propusiste correcciones razonables, ya piensas como diseñador de bases de datos, no solo como escritor de consultas. Esa es exactamente la mentalidad que esta unidad quería construir.

Resumen de la unidad

En esta unidad aprendiste las dos habilidades que convierten una base de datos estatica en un sistema vivo y relacionado.

Primero, la modificación de datos. UPDATE cambia valores existentes con la estructura UPDATE, SET y WHERE, y puede modificar varias columnas a la vez, calcular nuevos valores a partir de los anteriores y hasta usar subconsultas o JOINS para actualizar con base en otras tablas. DELETE elimina filas con DELETE FROM y WHERE. Ambas instrucciones comparten la misma ley de hierro: sin WHERE afectan a toda la tabla, y SQL no tiene botón de deshacer. Por eso adoptaste la disciplina profesional: SELECT de verificación primero, transacciones con START TRANSACTION, COMMIT y ROLLBACK para cambios delicados, y respaldos antes de operaciones masivas. También conociste TRUNCATE para vaciar tablas rápido y aprendiste que en los sistemas serios casi siempre se usa borrado lógico con una columna activo en lugar de borrar físicamente, para conservar el historial y poder revertir errores.

Segundo, las relaciones. La llave primaria es la identidad única de cada fila: única, nunca nula y permanente. Aprendiste a preferir llaves sustitutas con AUTO_INCREMENT sobre llaves naturales, porque los datos reales cambian y los números inventados no, y conociste los UUID para sistemas distribuidos y las llaves compuestas para tablas intermedias. La llave foránea es el puente entre tablas: una columna que referencia la llave primaria de otra tabla y que activa la integridad referencial, el guardián automatico que impide hijos huérfanos, borrados que rompen relaciones e inserciones de datos inexistentes. Viste los tres tipos de relaciones: uno a muchos con la foránea en el lado muchos, uno a uno compartiendo la llave, y muchos a muchos resuelto con una tabla intermedia que tiene dos foráneas y llave primaria compuesta.

También aprendiste las reglas ON DELETE: RESTRICT protege por defecto, CASCADE propaga el borrado automáticamente, SET NULL desvincula conservando el historial. Y te llevaste la lista de errores comunes y buenas prácticas: nunca DELETE sin WHERE, respaldos antes de cambios masivos, declarar siempre las foráneas, y modelar los muchos a muchos desde el diseño.

Con lo de esta unidad ya puedes mantener datos reales que cambian todos los días y diseñar bases de datos donde las tablas trabajan en equipo en lugar de vivir aisladas. En la siguiente unidad vas a usar estas relaciones para consultas más avanzadas, porque ahora que las tablas están conectadas, los JOINS se vuelven tu herramienta favorita.

Fin de la Unidad 6.

UNIDAD 7: COMBINAR TABLAS: LOS JOINS

Bienvenido a la unidad más importante de este curso. Si las unidades anteriores te enseñaron a leer, filtrar, ordenar y resumir los datos de una sola tabla, esta unidad te enseña el verdadero superpoder de SQL: unir varias tablas en una sola consulta. Los datos del mundo real viven separados en muchas tablas, y el JOIN es la herramienta que los vuelve a juntar. Dominar los joins es la diferencia entre alguien que "sabe un poco de SQL" y alguien que puede trabajar con bases de datos de verdad. El autor, Esteban, escribió esta unidad pensando en que cada ejemplo se pueda probar en cualquier motor de base de datos relacional: MySQL, PostgreSQL, SQL Server, SQLite o MariaDB. La sintaxis que usamos aquí es SQL estandar y funciona en todos ellos.

Imagina que trabajas en la tienda en línea de Lucas EGA Academy. La base de datos tiene una tabla de clientes con nombres, correos y ciudades. Tiene otra tabla de pedidos con fechas, montos y cantidades. Y tiene otra tabla de productos con descripciones y precios. Cada tabla por separado es útil, pero la pregunta que de verdad le interesa al negocio es: "cuáles son los clientes que más compraron este mes, en que ciudad viven y que productos se llevaron?". Esa pregunta necesita datos de las tres tablas al mismo tiempo. Sin joins tendrías que consultar la tabla de pedidos, anotar los identificadores de los clientes, ir a buscar esos clientes uno por uno a la otra tabla, y luego buscar los productos. Con un JOIN lo haces en una sola consulta de unas diez líneas.

Por eso se dice que el JOIN es el superpoder de SQL. Una base de datos relacional bien diseñada guarda cada dato una sola vez y lo relaciona con los demás por medio de llaves. La tabla de clientes no repite los datos del pedido, y la tabla de pedidos no repite los datos del cliente: solo guarda el identificador del cliente, que es la llave foránea. Esta forma de guardar la información se llama normalización y evita la duplicidad y las inconsistencias. Pero tiene un costo: cuando quieres ver la foto completa tienes que volver a unir las piezas. El JOIN es exactamente eso: la operación que reconstruye la foto completa a partir de piezas separadas, sin duplicar nada.

Piensa en el JOIN como el pegamento de la base de datos. Tienes piezas de un rompecabezas regadas en varias cajas. Cada caja tiene sus piezas ordenadas: la caja de clientes, la caja de pedidos, la caja de productos. El JOIN toma la pieza de una caja, busca su pareja en la otra caja usando la llave que las relaciona, y las pega lado a lado para formar una fila de resultados más completa. Y lo hace millones de veces en fracciones de segundo.

Antes de entrar a la sintaxis, entiende la idea central: un JOIN compara dos tablas fila por fila usando una condición, que casi siempre es "la llave foránea de una tabla es igual a la llave

primaria de la otra". Cuando dos filas cumplen la condición, se combinan en una sola fila del resultado. Cuando no la cumplen, depende del tipo de JOIN que estes usando: el INNER JOIN las descarta, el LEFT JOIN las conserva del lado izquierdo con valores vacios del lado derecho, y así. Esa es toda la filosofía. El resto es sintaxis y práctica.

En esta unidad vas a aprender cuatro herramientas que van siempre juntas en la vida real. Primero el INNER JOIN, que combina solo las filas que tienen pareja en ambas tablas: es el join más usado y el que resuelve el ochenta por ciento de las necesidades. Segundo el LEFT JOIN, que trae todo lo de la tabla de la izquierda aunque no tenga pareja en la derecha: es el que usas para encontrar lo que falta, como los clientes que nunca han pedido. También verás el RIGHT JOIN, que es el espejo del LEFT, y mencionaremos el FULL JOIN y el CROSS JOIN para que los reconozcas cuando los veas. Tercero el DISTINCT, que elimina filas duplicadas del resultado: parece simple, pero combinado con joins se vuelve una herramienta de limpieza poderosísima. Y cuarto los alias con la palabra AS, que te permiten ponerle nombres cortos a las tablas y nombres bonitos a las columnas: en joins con varias tablas los alias pasan de ser un lujo a ser casi obligatorios.

Una advertencia importante antes de seguir. Los joins son también la fuente número uno de errores y de consultas lentas en SQL. Olvidar la condición del JOIN produce un producto cartesiano: cada fila de una tabla combinada con cada fila de la otra, lo que puede convertir una consulta inocente en millones de filas basura. Poner una condición en el WHERE sobre una columna de la tabla derecha convierte silenciosamente tu LEFT JOIN en un INNER JOIN, y el resultado "casi" se ve bien, por eso es un error tan traicionero. Y escribir consultas sin alias cuando hay tres o cuatro tablas termina en columnas ambiguas que el motor rechaza. En la sección de errores comunes vamos a diseccionar cada uno de estos casos para que los reconozcas antes de que te muerdan.

Al final de esta unidad vas a poder escribir consultas que unan dos, tres o más tablas con INNER JOIN, usar LEFT JOIN para detectar ausencias con IS NULL, limpiar duplicados con DISTINCT, y escribir consultas legibles usando alias. También vas a entender cuando un JOIN es innecesario y cuando el DISTINCT es la respuesta equivocada (pista: muchas veces un GROUP BY comunica mejor la intención). Los ejercicios de práctica están diseñados para que te equivoques en un ambiente seguro: cada ejercicio trae su solución explicada paso a paso para que compares tu razonamiento.

Prepara tu editor de SQL, crea las tablas de ejemplo de la sección de ejemplos y ejecuta cada consulta con tus propias manos. Los joins no se aprenden leyendo: se aprenden ejecutando, viendo el resultado, cambiando la condición y volviendo a ejecutar. Esta unidad es larga porque el tema lo merece. Vamos paso a paso.

Explicación detallada

Por qué los datos viven en tablas separadas

Para entender los joins hay que entender primero por qué las tablas están separadas. Cuando se diseña una base de datos relacional se sigue un principio: cada dato se guarda una sola vez, en un solo lugar. Si guardaras el nombre y la dirección del cliente dentro de cada pedido, estarías repitiendo esa información cientos de veces. Y si el cliente se cambia de ciudad, tendrías que actualizar cientos de filas. Eso es un desastre esperando a pasar. Por eso se separan: la tabla clientes guarda los datos del cliente una sola vez, la tabla pedidos guarda cada pedido una sola vez, y la tabla pedidos apunta a la tabla clientes mediante la llave foránea cliente_id.

Esta separación es sana, pero crea un problema práctico: ninguna tabla tiene la foto completa. La tabla pedidos sabe cuánto se vendió y cuándo, pero no sabe cómo se llama el cliente. La tabla clientes sabe el nombre y la ciudad, pero no sabe qué compró. Para responder preguntas de negocio reales necesitas juntarlas. El JOIN existe exactamente para eso: es la operación que vuelve a unir lo que el buen diseño separó.

Piensa en las llaves como los enchufes del rompecabezas. La llave primaria es el identificador único de cada fila: en clientes, la columna id. La llave foránea es una columna en otra tabla que guarda ese identificador: en pedidos, la columna cliente_id guarda el id del cliente que hizo el pedido. El JOIN usa esa coincidencia para emparejar filas. Cuando el pedido número 5 tiene cliente_id = 3, el JOIN lo empareja con el cliente número 3 de la tabla clientes. Simple, elegante y rapidísimo cuando hay índices.

La idea del JOIN con diagramas de Venn

La forma clásica de visualizar los joins es con diagramas de Venn: dos círculos que se traslapan. El círculo de la izquierda representa la tabla A, el círculo de la derecha representa la tabla B, y la zona donde se traslapan representa las filas que tienen pareja en ambas tablas según la condición del JOIN.

El INNER JOIN es solo el traslape: las filas que existen en ambas tablas. Si un cliente no tiene pedidos, no aparece. Si un pedido apunta a un cliente que no existe (algo que no debería pasar si hay integridad referencial, pero pasa en datos reales), tampoco aparece. Solo las parejas.

El LEFT JOIN es todo el círculo izquierdo: todas las filas de la tabla A, tengan o no tengan pareja en B. Las que sí tienen pareja salen con sus datos completos. Las que no tienen pareja

salen con valores NULL en las columnas de B. Es como decir "quiero ver a todos los clientes, y si tienen pedidos, muéstramelos también".

El RIGHT JOIN es todo el círculo derecho: el espejo del LEFT JOIN. Todas las filas de la tabla B, tengan o no tengan pareja en A. En la práctica casi nadie escribe RIGHT JOIN: es más natural voltear el orden de las tablas y usar LEFT JOIN. Pero tienes que reconocerlo cuando lo leas en código ajeno.

El FULL JOIN (o FULL OUTER JOIN) es la unión de los dos círculos completos: todo lo de A y todo lo de B, con NULL donde no haya pareja. No todos los motores lo soportan de forma nativa (MySQL, por ejemplo, no tiene FULL JOIN directo), pero el concepto es útil y se puede simular combinando un LEFT JOIN y un RIGHT JOIN con UNION.

El CROSS JOIN es otra bestia: es el producto cartesiano, cada fila de A combinada con cada fila de B, sin condición de emparejamiento. En el diagrama de Venn no tiene representación bonita porque no es un traslape: es una multiplicación. Casi nunca lo quieres, y cuando aparece suele ser un error. Pero tiene usos legítimos, como generar combinaciones (todas las tallas por todos los colores de una prenda).

Guarda esta imagen mental de los círculos porque te va a salvar muchas veces. Cada vez que dudes que tipo de JOIN usar, dibuja mentalmente los dos círculos y pregunta: "quiero solo el traslape, o quiero todo un círculo?".

INNER JOIN a fondo

El INNER JOIN es el join por defecto, el caballo de batalla. Su sintaxis básica es:

```
SELECT columnas
FROM tabla_a
INNER JOIN tabla_b ON tabla_a.llave = tabla_b.llave_foranea;
```

La palabra INNER es opcional: escribir solo JOIN es lo mismo que escribir INNER JOIN. La mayoría de los programadores escriben solo JOIN por brevedad, y está bien. Pero en esta unidad vamos a escribir INNER JOIN completo mientras aprendes, para que el tipo de join quede siempre explícito. Cuando ya lo domines, puedes acortar.

La cláusula ON es el corazón del JOIN: ahí va la condición de emparejamiento. Casi siempre es una igualdad entre la llave primaria de una tabla y la llave foránea de la otra. Pero el ON puede llevar cualquier condición: igualdades, desigualdades, rangos, incluso varias condiciones unidas con AND. Por ejemplo, podrías unir pedidos con clientes solo cuando el pedido sea del último año, poniendo esa condición extra en el ON. Eso sí: la condición del ON

define que filas se emparejan, y eso es distinto de filtrar el resultado final con WHERE. Más adelante veremos por qué esa distinción importa muchísimo en los LEFT JOIN.

Veamos el ejemplo canonico con nuestras tablas de la tienda. Queremos una lista de pedidos con el nombre del cliente:

```
SELECT pedidos.id, pedidos.fecha, clientes.nombre, pedidos.total
FROM pedidos
INNER JOIN clientes ON pedidos.cliente_id = clientes.id;
```

Lee la consulta en voz alta: "dame el id y la fecha del pedido, el nombre del cliente y el total, de la tabla pedidos unida con la tabla clientes donde el cliente_id del pedido sea igual al id del cliente". Cada fila del resultado combina un pedido con su cliente. Si hay un pedido cuyo cliente_id no existe en la tabla clientes, ese pedido desaparece del resultado. Si hay un cliente sin pedidos, ese cliente no aparece. Solo parejas.

Un detalle de estilo que verás en todo el código profesional: cuando las columnas se repiten en ambas tablas (como id), hay que calificarlas con el nombre de la tabla: pedidos.id, clientes.nombre. Si no lo haces, el motor no sabe a cuál id te refieres y te devuelve un error de "columna ambigua". Con los alias que veremos más adelante esto se vuelve mucho más comodo.

Joins de tres o más tablas

La vida real rara vez se resuelve con dos tablas. Volvamos a la pregunta del negocio: "que clientes compraron, en que ciudad viven y que productos se llevaron?". Necesitamos clientes, pedidos y productos. Y como un pedido puede tener varios productos, en un diseño real hay una tabla intermedia: detalle_pedido, que relaciona pedidos con productos y guarda cantidades. Entonces la cadena es: clientes se une con pedidos, pedidos se une con detalle_pedido, detalle_pedido se une con productos. Tres joins en una sola consulta:

```
SELECT clientes.nombre, clientes.ciudad,
       productos.nombre AS producto,
       detalle_pedido.cantidad
FROM clientes
INNER JOIN pedidos ON pedidos.cliente_id = clientes.id
INNER JOIN detalle_pedido ON detalle_pedido.pedido_id = pedidos.id
INNER JOIN productos ON productos.id = detalle_pedido.producto_id;
```

Observa el patrón: cada JOIN nuevo se apoya en una tabla que ya está en la consulta. Es una cadena: clientes -> pedidos -> detalle_pedido -> productos. Puedes encadenar tantos joins

como necesites: cuatro, cinco, ocho tablas en una consulta compleja no son raros en sistemas reales. La regla de oro es que cada JOIN debe tener su ON, y cada ON debe conectar la tabla nueva con algo que ya este unido. Si un JOIN queda sin ON, o con un ON que no conecta nada, obtienes un producto cartesiano parcial que multiplica tus filas sin sentido.

El orden en que escribes los joins también comunica intención. Empieza por la tabla que es el "sujeto" de tu pregunta. Si la pregunta es sobre clientes, empieza en clientes. Si la pregunta es sobre pedidos, empieza en pedidos. El resultado lógico es el mismo, pero la consulta se lee mejor y a veces el optimizador del motor trabaja mejor.

El self join: una tabla unida consigo misma

Hay un caso especial que parece truco de magia: unir una tabla consigo misma. Se llama self join y sirve cuando las filas de una tabla se relacionan entre si. El ejemplo clásico es una tabla de empleados donde cada empleado tiene un jefe, y el jefe también es un empleado de la misma tabla. La columna jefe_id es una llave foránea que apunta al id de la misma tabla empleados.

```
SELECT e.nombre AS empleado, j.nombre AS jefe
FROM empleados e
INNER JOIN empleados j ON e.jefe_id = j.id;
```

Aquí los alias son obligatorios, no opcionales: sin alias el motor no podría distinguir las dos copias de la tabla empleados. Le ponemos e a la copia de empleados y j a la copia de jefes. El resultado muestra cada empleado junto al nombre de su jefe. Los empleados sin jefe (el director general, con jefe_id NULL) no aparecen con INNER JOIN: para verlos necesitarías un LEFT JOIN, como veremos enseguida. El self join también sirve para comparar filas dentro de la misma tabla, como encontrar productos con el mismo precio o clientes de la misma ciudad.

LEFT JOIN: todo lo de la izquierda

El LEFT JOIN es el segundo join más importante y el favorito para las preguntas de "que falta". Su sintaxis es idéntica a la del INNER, solo cambia la palabra clave:

```
SELECT clientes.nombre, pedidos.id AS pedido_id, pedidos.total
FROM clientes
LEFT JOIN pedidos ON pedidos.cliente_id = clientes.id;
```

La tabla de la izquierda es la que aparece en el FROM (clientes), y la de la derecha es la del JOIN (pedidos). El LEFT JOIN trae TODOS los clientes, tengan o no tengan pedidos. Para los

clientes con pedidos, las columnas de pedidos salen con sus valores normales. Para los clientes sin pedidos, las columnas de pedidos salen como NULL.

Esos NULL son información valiosísima: te dicen "aquí no hay pareja". El patrón clásico para encontrar clientes sin pedidos combina LEFT JOIN con un filtro IS NULL:

```
SELECT clientes.nombre, clientes.correo
FROM clientes
LEFT JOIN pedidos ON pedidos.cliente_id = clientes.id
WHERE pedidos.id IS NULL;
```

Lee la lógica: primero el LEFT JOIN trae a todos los clientes con sus pedidos (o con NULL si no tienen). Después el WHERE se queda solo con las filas donde el id del pedido es NULL, es decir, los clientes que no tienen ningún pedido. Este es probablemente el truco más útil de todo SQL para análisis de negocio: clientes sin pedidos, productos sin ventas, empleados sin asignación. Memorízalo.

RIGHT JOIN: el espejo

El RIGHT JOIN hace lo mismo pero al revés: trae todas las filas de la tabla de la derecha, tengan o no tengan pareja en la izquierda:

```
SELECT clientes.nombre, pedidos.id, pedidos.total
FROM clientes
RIGHT JOIN pedidos ON pedidos.cliente_id = clientes.id;
```

Esto trae todos los pedidos, y el nombre del cliente cuando exista. En la práctica, esta consulta se escribe más naturalmente como un LEFT JOIN volteando las tablas:

```
SELECT clientes.nombre, pedidos.id, pedidos.total
FROM pedidos
LEFT JOIN clientes ON pedidos.cliente_id = clientes.id;
```

Ambas consultas devuelven lo mismo. La convención en la industria es preferir LEFT JOIN y ordenar las tablas para que la tabla "principal" quede a la izquierda. Así el código se lee siempre en la misma dirección. Pero cuando heredes código ajeno vas a encontrar RIGHT JOIN, así que debes entenderlo sin problema.

FULL JOIN y CROSS JOIN: menciones necesarias

El FULL JOIN (FULL OUTER JOIN) trae todo de ambas tablas: las parejas emparejadas, más las filas de la izquierda sin pareja (con NULL a la derecha), más las filas de la derecha sin pareja (con NULL a la izquierda). Es útil para conciliar dos fuentes de datos, como comparar la lista de clientes del sistema de ventas contra la del sistema de facturación para ver que falta en cada lado. Sintaxis:

```
SELECT clientes.nombre, pedidos.id, pedidos.total
FROM clientes
FULL JOIN pedidos ON pedidos.cliente_id = clientes.id;
```

Ojo: MySQL no soporta FULL JOIN directamente. En MySQL se simula con un LEFT JOIN más un RIGHT JOIN unidos con UNION. PostgreSQL, SQL Server y SQLite si lo soportan. Si tu motor no lo tiene y lo necesitas, busca "simular full join con union" en la documentación de tu motor.

El CROSS JOIN no lleva ON porque no empareja nada: combina cada fila de la izquierda con cada fila de la derecha. Con 100 clientes y 50 productos obtienes 5,000 filas. Su sintaxis:

```
SELECT clientes.nombre, productos.nombre AS producto
FROM clientes
CROSS JOIN productos;
```

Usos legítimos: generar todas las combinaciones posibles (tallas por colores, días por turnos), o crear tablas de números y fechas para reportes. Pero en el 99 por ciento de los casos, si tu consulta produce muchas más filas de las esperadas y no sabes por qué, sospecha de un CROSS JOIN accidental: casi siempre es un JOIN al que se le olvidó el ON.

Alias de tablas y columnas: casi obligatorios en joins

Un alias es un nombre temporal que le pones a una tabla o a una columna dentro de una consulta. Se declara con la palabra AS, aunque el AS es opcional para tablas en la mayoría de los motores. Para columnas, el AS también es opcional pero se recomienda escribirlo por claridad.

Alias de tabla:

```
SELECT c.nombre, p.total
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id;
```

Aquí c es alias de clientes y p es alias de pedidos. La consulta hace exactamente lo mismo que la versión sin alias, pero es mucho más corta y legible. Con dos tablas el ahorro es modesto; con cinco tablas y columnas calificadas por todas partes, los alias son la diferencia entre una consulta legible y una sopa de letras.

Reglas de los alias de tabla: se declaran justo después del nombre de la tabla en el FROM o en el JOIN, y a partir de ese momento debes usar el alias en toda la consulta (ya no puedes usar el nombre original en esa consulta). Los alias cortos de una letra son la convención más comun: c para clientes, p para pedidos, d para detalle_pedido, pr para productos. En consultas grandes algunos prefieren alias un poco más descriptivos como cli o ped. Lo importante es ser consistente.

Alias de columna:

```
SELECT c.nombre AS cliente, p.total AS total_pedido
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id;
```

El alias de columna renombra la columna en el resultado. Es útil cuando dos tablas tienen columnas con el mismo nombre (como nombre en clientes y nombre en productos: sin alias el resultado tendría dos columnas llamadas "nombre" y seria confuso), y también para poner encabezados bonitos en reportes. Observa que en el ejemplo anterior renombramos productos.nombre como producto para no chocar con clientes.nombre.

Un error comun con alias: intentar usar el alias de columna en el WHERE. No funciona, porque el WHERE se evalua antes que el SELECT. Los alias de columna si se pueden usar en el ORDER BY, porque el ORDER BY se evalua después. Este detalle del orden de evaluación (FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY) explica muchos errores misteriosos. Guardalo en la memoria.

DISTINCT: eliminar duplicados

El DISTINCT elimina las filas duplicadas del resultado, dejando una sola copia de cada combinación única:

```
SELECT DISTINCT ciudad
FROM clientes;
```

Si la tabla clientes tiene 200 clientes pero solo 12 ciudades distintas, el resultado trae 12 filas: una por ciudad. Sin DISTINCT traeria 200 filas con las ciudades repetidas.

DISTINCT funciona sobre la fila completa, no sobre una sola columna. Si pides DISTINCT sobre varias columnas, elimina las combinaciones duplicadas:

```
SELECT DISTINCT ciudad, nombre  
FROM clientes;
```

Esto trae cada combinación única de ciudad y nombre. Dos clientes llamados "Juan" en la misma ciudad colapsan en una sola fila; dos "Juan" en distintas ciudades son dos filas.

DISTINCT contra GROUP BY: para el caso simple de "dame los valores únicos de esta columna", DISTINCT y GROUP BY dan el mismo resultado:

```
SELECT ciudad FROM clientes GROUP BY ciudad;
```

La diferencia es de intención y de poder. DISTINCT dice "quiero valores únicos". GROUP BY dice "quiero agrupar para calcular algo por grupo", y se combina con funciones de agregación (COUNT, SUM, AVG). Si solo necesitas la lista de valores únicos, DISTINCT comunica mejor tu intención y suele ser más rápido. Si necesitas contar cuantos clientes hay por ciudad, necesitas GROUP BY con COUNT. Regla práctica: DISTINCT para listas unicas, GROUP BY para resúmenes con calculos.

DISTINCT con joins es donde se vuelve interesante. Un JOIN puede multiplicar filas: si un cliente tiene 5 pedidos, un INNER JOIN de clientes con pedidos repite al cliente 5 veces. Si solo quieres la lista de clientes que han comprado (sin repetirlos), el DISTINCT limpia el resultado:

```
SELECT DISTINCT c.nombre, c.ciudad  
FROM clientes c  
INNER JOIN pedidos p ON p.cliente_id = c.id;
```

Esto trae cada cliente comprador una sola vez, aunque tenga muchos pedidos. Alternativa elegante al mismo problema: usar EXISTS en lugar de JOIN, que veremos en unidades posteriores. Por ahora, quédate con que DISTINCT es el borrador que limpia los duplicados que los joins tienden a generar.

Cuidado con un malentendido comun: DISTINCT no es gratis. El motor tiene que ordenar o usar una tabla hash para detectar duplicados, lo que cuesta tiempo y memoria en resultados grandes. No pongas DISTINCT "por si acaso" en todas tus consultas. Si tu consulta no produce duplicados, el DISTINCT solo gasta recursos. Y si tu consulta produce duplicados inesperados, muchas veces el problema real es un JOIN mal planteado (un producto

cartesiano parcial), y el DISTINCT solo esconde el síntoma. Investiga primero por qué hay duplicados; usa DISTINCT cuando los duplicados sean legítimos y simplemente no los quieras en el reporte.

Errores comunes con joins (y como evitarlos)

Primer error: el producto cartesiano por olvidar el ON. Esta consulta es válida sintácticamente pero casi seguro no es lo que quieres:

```
SELECT c.nombre, p.total
FROM clientes c
INNER JOIN pedidos p;
```

Algunos motores la rechazan porque falta el ON; otros (como MySQL en ciertos modos, o si usas la sintaxis vieja con comas) la ejecutan y te devuelven cada cliente combinado con cada pedido. Con 1,000 clientes y 10,000 pedidos obtienes 10,000,000 de filas basura. Síntoma típico: "mi consulta trae muchísimas más filas de las que debería". Diagnóstico: revisa que cada JOIN tenga su ON y que cada ON conecte las tablas correctas.

Segundo error: columnas ambiguas. Cuando dos tablas tienen una columna con el mismo nombre y no la calificas, el motor se queja:

```
SELECT nombre, total
FROM clientes
INNER JOIN pedidos ON pedidos.cliente_id = clientes.id;
```

Si tanto clientes como pedidos tuvieran una columna llamada nombre, el motor no sabría cuál pedir y lanzaría error de ambigüedad. La solución es calificar siempre: clientes.nombre o usar alias: c.nombre. Acostúmbrate a calificar todas las columnas en consultas con joins, incluso las que no son ambiguas hoy: mañana alguien puede agregar una columna con el mismo nombre a la otra tabla y tu consulta se rompería.

Tercer error, el más traicionero: convertir un LEFT JOIN en INNER JOIN con un WHERE mal puesto. Mira:

```
SELECT c.nombre, p.total
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id
WHERE p.total > 100;
```

La intención era "todos los clientes, y entre sus pedidos solo los mayores a 100". Pero el WHERE se aplica después del JOIN, y para los clientes sin pedidos p.total es NULL, y NULL > 100 no es verdadero, así que esos clientes se eliminan. El resultado es idéntico a un INNER JOIN con ese filtro. Si quieres conservar a los clientes sin pedidos y solo filtrar los pedidos emparejados, la condición va en el ON, no en el WHERE:

```
SELECT c.nombre, p.total
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id AND p.total > 100;
```

Ahora el ON dice "empareja cada cliente con sus pedidos mayores a 100; si no tiene, déjalo con NULL". El cliente sin pedidos (o sin pedidos grandes) sigue apareciendo. Esta diferencia entre filtrar en el ON y filtrar en el WHERE es uno de los conceptos que más distinguen a quien domina SQL. En INNER JOIN no importa donde pongas el filtro (el resultado es el mismo); en LEFT JOIN cambia todo.

Cuarto error: unir por la columna equivocada. Si unes pedidos con clientes por pedidos.id = clientes.id en lugar de pedidos.cliente_id = clientes.id, el motor no se queja (ambas son columnas numéricas válidas), pero el resultado es un sinsentido: empareja el pedido 5 con el cliente 5 aunque no tengan relación. Síntoma: resultados "casi bien" pero con datos que no cuadran. Prevención: conoce tu modelo de datos, revisa cuáles son las llaves primarias y foráneas, y desconfía de resultados que se ven raros.

Quinto error: duplicar filas sin darte cuenta por joins en cadena con relaciones de uno a muchos en varios niveles. Si un cliente tiene 3 pedidos y cada pedido tiene 4 productos, el join triple repite la información del cliente 12 veces. Eso es correcto desde el punto de vista del JOIN (cada fila es una combinación válida), pero si luego sumas los totales de los pedidos sobre ese resultado, los contarás varias veces. La lección: entiende la cardinalidad de tus relaciones antes de agregar (SUM, COUNT) sobre resultados de joins múltiples. Muchas veces conviene agregar primero en una subconsulta y unir después el resultado ya resumido.

Buenas prácticas y rendimiento en joins

Primera práctica: índices en las columnas de unión. Un JOIN sin índice en la llave foránea obliga al motor a comparar cada fila de una tabla con cada fila de la otra: eso es lento con tablas grandes. Las llaves primarias suelen tener índice automático; las foráneas no siempre. Si tus joins son lentos, lo primero que se revisa es si las columnas del ON tienen índice. No necesitas ser administrador de bases de datos para esto: basta saber que existe y pedir ayuda cuando una consulta se arrastra.

Segunda práctica: une solo las tablas que necesites y pide solo las columnas que necesites. Cada JOIN extra es trabajo para el motor. El SELECT * en una consulta con joins es doblemente malo: trae columnas que no vas a usar y puede traer columnas duplicadas o ambiguas. Especifica tus columnas.

Tercera práctica: filtra antes de unir cuando puedas. Si solo te interesan los pedidos del último mes, filtrar los pedidos primero (en una subconsulta o con la condición bien puesta) reduce las filas que el JOIN tiene que procesar. Los motores modernos suelen optimizar esto solos, pero escribir la intención clara ayuda.

Cuarta práctica: consistencia en el estilo. Usa siempre la sintaxis explícita con JOIN y ON (no la sintaxis vieja con comas en el FROM, que es propensa al producto cartesiano accidental). Usa alias cortos y consistentes. Escribe las palabras clave en mayusculas. Califica las columnas. Tu yo del futuro, y tus compañeros de equipo, te lo van a agradecer cuando tengan que leer una consulta de seis joins a las tres de la mañana.

Quinta práctica: prueba tus joins por partes. Si una consulta de cuatro joins te da resultados raros, ejecútala con un solo join, revisa el resultado, agrega el segundo, revisa, y así. Aislar el problema es más rápido que mirar la consulta completa tratando de adivinar donde esta el error. Y cuenta las filas en cada paso: si al agregar un JOIN el conteo se multiplica de forma inesperada, ahí esta tu problema.

Sexta práctica: documenta la intención de los LEFT JOIN. Un LEFT JOIN siempre responde a la pregunta "que pasa con los que no tienen pareja?". Si la respuesta es "nada, solo quiero las parejas", usa INNER JOIN. Dejar un LEFT JOIN innecesario no solo es más lento: confunde al que lee el código, que va a pensar que los NULL importan. Cada tipo de JOIN comunica una intención; elige el que diga lo que realmente quieres.

El orden en que el motor evalúa tu consulta

Para entender por qué el alias de columna no funciona en el WHERE pero si en el ORDER BY, y por qué el WHERE puede arruinar un LEFT JOIN, necesitas conocer el orden lógico de evaluación de una consulta SQL. No es el orden en que la escribes. El motor la procesa así:

1. FROM y JOIN: primero une las tablas según los ON. Aquí nacen las filas combinadas, incluyendo los NULL de los LEFT JOIN.
2. WHERE: filtra esas filas combinadas. Por eso un WHERE sobre una columna de la tabla derecha elimina las filas con NULL del LEFT JOIN.
3. GROUP BY: agrupa las filas que sobrevivieron.
4. HAVING: filtra grupos.

5. **SELECT:** aquí se calculan las expresiones y se aplican los alias de columna. Por eso el **WHERE** (paso 2) no conoce los alias: todavía no existen cuando el **WHERE** se evalúa.
6. **ORDER BY:** ordena el resultado final. Como va después del **SELECT**, si puede usar los alias de columna.

Este orden explica también por qué no puedes usar una columna calculada en el **WHERE** pero sí en el **ORDER BY**, y por qué el **DISTINCT** se aplica al final sobre las filas ya formadas. No es un detalle académico: cada error misterioso de "columna no existe" o "mi **LEFT JOIN** no trae los **NULL**" se entiende en cuanto miras este orden. Cuando algo no cuadre, recorre mentalmente estos seis pasos con tus datos.

Los **NULL** nunca emparejan en un **JOIN**

Un detalle sutil pero importante: la condición del **ON** usa igualdad, y en **SQL** **NULL** no es igual a **NULL**. Si una tabla tiene filas con la llave foránea en **NULL**, esas filas nunca emparejan con nada, ni siquiera con otras filas que también tengan **NULL**. En un **INNER JOIN** simplemente desaparecen. En un **LEFT JOIN** aparecen con **NULL** del lado derecho, que es exactamente lo que esperas.

Esto tiene una consecuencia práctica: si tu tabla de pedidos permite `cliente_id` **NULL** (pedidos de invitados sin registro, por ejemplo), esos pedidos nunca se unirán con la tabla `clientes` por más **JOIN** que hagas. Si necesitas incluirlos en un reporte, el **LEFT JOIN** desde pedidos hacia clientes los conserva con el nombre del cliente en **NULL**, y ahí decides cómo presentarlos (con un **COALESCE** a 'Invitado', por ejemplo). Conocer este comportamiento te evita horas de depuración buscando filas "perdidas".

Condiciones de union más allá de la igualdad

Casi todos los joins usan igualdad entre llaves, pero el **ON** acepta cualquier condición. Un caso útil es unir por rangos: por ejemplo, una tabla de promociones con fecha de inicio y fin, unida con pedidos para saber qué promoción aplicaba a cada pedido:

```
SELECT p.id AS pedido, prm.nombre AS promocion
FROM pedidos p
INNER JOIN promociones prm
  ON p.fecha BETWEEN prm.fecha_inicio AND prm.fecha_fin;
```

Aquí no hay llaves involucradas: el emparejamiento es "la fecha del pedido cae dentro del rango de la promoción". Otro caso es unir con varias condiciones a la vez usando **AND**, como emparejar por cliente y además por año. La regla es la misma: el **ON** define qué significa

"estas dos filas van juntas". La igualdad de llaves es solo el caso más común, no el único posible. Eso sí: los joins por rango suelen ser más lentos que los joins por igualdad con índice, así que usalos con medida y solo cuando el problema lo pida.

Ejemplos paso a paso

Para que todos los ejemplos de esta sección funcionen igual en tu computadora, vamos a crear primero las tablas y a llenarlas con datos de prueba. Son las tablas de la tienda de Lucas EGA Academy: clientes, pedidos, productos y la tabla intermedia detalle_pedido. Copia y ejecuta este bloque completo antes de probar los ejemplos.

```
CREATE TABLE clientes (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    correo VARCHAR(100),  
    ciudad VARCHAR(50)  
);  
  
CREATE TABLE pedidos (  
    id INT PRIMARY KEY,  
    cliente_id INT,  
    fecha DATE,  
    total DECIMAL(10, 2)  
);  
  
CREATE TABLE productos (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    precio DECIMAL(10, 2)  
);  
  
CREATE TABLE detalle_pedido (  
    pedido_id INT,  
    producto_id INT,  
    cantidad INT  
);  
  
INSERT INTO clientes (id, nombre, correo, ciudad) VALUES  
(1, 'Ana Torres', 'ana@correo.com', 'Ciudad de Mexico'),  
(2, 'Luis Gomez', 'luis@correo.com', 'Guadalajara'),  
(3, 'Maria Ruiz', 'maria@correo.com', 'Ciudad de Mexico'),  
(4, 'Carlos Diaz', 'carlos@correo.com', 'Monterrey'),  
(5, 'Sofia Vega', 'sofia@correo.com', 'Guadalajara');
```

```

INSERT INTO pedidos (id, cliente_id, fecha, total) VALUES
(101, 1, '2026-09-01', 450.00),
(102, 1, '2026-09-10', 120.00),
(103, 2, '2026-09-05', 780.00),
(104, 3, '2026-09-12', 95.00);

INSERT INTO productos (id, nombre, precio) VALUES
(1, 'Curso de HTML5', 27.00),
(2, 'Curso de CSS3', 27.00),
(3, 'Libro de SQL', 35.00),
(4, 'Libro de Python', 35.00);

INSERT INTO detalle_pedido (pedido_id, producto_id, cantidad) VALUES
(101, 1, 2),
(101, 3, 1),
(102, 2, 1),
(103, 1, 1),
(103, 4, 3),
(104, 2, 2);

```

Fíjate en los datos porque los ejemplos los usan a proposito. Hay 5 clientes pero solo 3 tienen pedidos: Carlos Diaz (id 4) y Sofia Vega (id 5) no han comprado nada. Hay 4 pedidos y 4 productos. El pedido 101 es de Ana Torres e incluye 2 cursos de HTML5 y 1 libro de SQL. Con este escenario vamos a ver cada tipo de JOIN en acción.

Ejemplo 1: INNER JOIN de pedidos con clientes

Pregunta: quiero ver cada pedido con el nombre del cliente que lo hizo.

```

SELECT p.id AS pedido, p.fecha, c.nombre AS cliente, p.total
FROM pedidos p
INNER JOIN clientes c ON p.cliente_id = c.id
ORDER BY p.id;

```

Explicación paso a paso. El FROM dice que la tabla base es pedidos con alias p. El INNER JOIN trae la tabla clientes con alias c. El ON empareja p.cliente_id con c.id: el pedido 101 tiene cliente_id 1, así que se empareja con Ana Torres; el pedido 103 tiene cliente_id 2, así que se empareja con Luis Gomez. Cómo es INNER JOIN, solo aparecen las filas con pareja: los 4 pedidos tienen cliente valido, así que el resultado trae 4 filas. Carlos Diaz y Sofia Vega no aparecen porque no tienen pedidos: el INNER JOIN no muestra clientes sin pareja. El resultado se ve así:

pedido 101, fecha 2026-09-01, cliente Ana Torres, total 450.00. Pedido 102, fecha 2026-09-10, cliente Ana Torres, total 120.00. Pedido 103, fecha 2026-09-05, cliente Luis Gomez, total 780.00. Pedido 104, fecha 2026-09-12, cliente Maria Ruiz, total 95.00.

Observa que Ana Torres aparece dos veces porque hizo dos pedidos. Eso es normal y correcto: el resultado tiene una fila por pedido, no una por cliente. Si quisieras una fila por cliente necesitarías agrupar o usar DISTINCT, como veremos en el ejemplo 5.

Ejemplo 2: JOIN triple, del cliente al producto

Pregunta: quiero ver que producto se llevo cada cliente, con cantidades.

```
SELECT c.nombre AS cliente, pr.nombre AS producto, d.cantidad
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id
INNER JOIN detalle_pedido d ON d.pedido_id = p.id
INNER JOIN productos pr ON pr.id = d.producto_id
ORDER BY c.nombre, pr.nombre;
```

Explicación paso a paso. Esta es la cadena completa de cuatro tablas. Primer JOIN: clientes con pedidos, emparejando por cliente_id. Segundo JOIN: el resultado anterior con detalle_pedido, emparejando por pedido_id. Tercer JOIN: con productos, emparejando por producto_id. Cada fila final representa "un cliente compró cierta cantidad de un producto en un pedido". El resultado trae 6 filas, una por cada renglon de detalle_pedido que tenga su cadena completa:

Ana Torres compró 2 del Curso de HTML5. Ana Torres compró 1 del Libro de SQL. Ana Torres compró 1 del Curso de CSS3. Luis Gomez compró 1 del Curso de HTML5. Luis Gomez compró 3 del Libro de Python. Maria Ruiz compró 2 del Curso de CSS3.

Fíjate en lo que NO aparece: Carlos Diaz y Sofia Vega (sin pedidos) no salen, y ningun producto sin ventas saldria tampoco. El INNER JOIN en cadena es estricto: si un eslabon de la cadena falla, la fila completa desaparece. Por eso cuando agregues tablas a la cadena, verifica el conteo de filas en cada paso.

Ejemplo 3: LEFT JOIN para ver a todos los clientes

Pregunta: quiero la lista completa de clientes con el total de sus pedidos, incluyendo a los que no han comprado.

```
SELECT c.nombre AS cliente, c.ciudad, p.id AS pedido, p.total
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id
ORDER BY c.nombre, p.id;
```

Explicación paso a paso. Ahora la tabla izquierda es clientes y queremos TODOS los clientes. El LEFT JOIN empareja cada cliente con sus pedidos. Ana Torres tiene dos pedidos, así que aparece dos veces con sus dos totales. Luis Gomez y Maria Ruiz aparecen una vez cada uno. Carlos Diaz y Sofia Vega no tienen pedidos: aparecen una vez cada uno, con las columnas de pedido en NULL. El resultado trae 6 filas: 2 de Ana, 1 de Luis, 1 de Maria, 1 de Carlos (pedido NULL, total NULL) y 1 de Sofia (pedido NULL, total NULL).

Esos NULL son la señal de "este cliente no tiene pedidos". Este ejemplo muestra la diferencia esencial con el INNER JOIN del ejemplo 1: allá los clientes sin pedidos desaparecían; aquí están presentes con NULL. Si tu reporte es "clientes y sus compras", casi siempre quieres el LEFT JOIN, porque un reporte que omita silenciosamente a los clientes inactivos es un reporte mentiroso.

Ejemplo 4: LEFT JOIN con IS NULL, los clientes sin pedidos

Pregunta: cuales clientes NUNCA han hecho un pedido? Es la pregunta de marketing: a quienes les mandó la promoción de bienvenida?

```
SELECT c.nombre AS cliente, c.correo, c.ciudad
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id
WHERE p.id IS NULL;
```

Explicación paso a paso. El LEFT JOIN trae a los 5 clientes con sus pedidos o NULL. El WHERE p.id IS NULL filtra quedandose solo con las filas donde no hubo emparejamiento: aquellas donde el id del pedido quedó en NULL. El resultado trae exactamente 2 filas: Carlos Diaz (carlos@correo.com, Monterrey) y Sofia Vega (sofia@correo.com, Guadalajara). Son los clientes sin pedidos.

Este patrón (LEFT JOIN + IS NULL sobre la llave de la tabla derecha) es uno de los más usados en SQL profesional. Sirve para "clientes sin pedidos", "productos sin ventas", "empleados sin proyecto asignado", "cursos sin alumnos inscritos". Aprendetelo de memoria porque lo vas a usar toda tu carrera. Y recuerda la trampa: el IS NULL va en el WHERE sobre una columna de la tabla derecha que no puede ser NULL en datos reales (la llave primaria id es perfecta para eso).

Ejemplo 5: DISTINCT de ciudades

Pregunta: en que ciudades tenemos clientes? Quiero la lista sin repetidos.

```
SELECT DISTINCT ciudad
FROM clientes
ORDER BY ciudad;
```

Explicación paso a paso. La tabla tiene 5 clientes en 3 ciudades: Ciudad de Mexico (Ana y Maria), Guadalajara (Luis y Sofia) y Monterrey (Carlos). Sin DISTINCT obtendrias 5 filas con repetidos. Con DISTINCT el motor colapsa los duplicados y devuelve 3 filas: Ciudad de Mexico, Guadalajara, Monterrey. Simple y directo.

Ahora una variante con JOIN, que muestra por qué DISTINCT y JOIN son buenos amigos:

```
SELECT DISTINCT c.ciudad
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id
ORDER BY c.ciudad;
```

Pregunta: en que ciudades viven los clientes QUE SI han comprado? El INNER JOIN repite a Ciudad de Mexico dos veces (Ana tiene 2 pedidos, Maria tiene 1), pero el DISTINCT limpia y el resultado trae 2 filas: Ciudad de Mexico y Guadalajara. Monterrey no aparece porque Carlos Diaz no tiene pedidos. Este es el uso típico: el JOIN multiplica, el DISTINCT limpia.

Ejemplo 6: alias cortos en una consulta con agregación

Pregunta: cuanto ha comprado en total cada cliente, incluyendo a los que no han comprado nada?

```
SELECT c.nombre AS cliente,
       COUNT(p.id) AS num_pedidos,
       COALESCE(SUM(p.total), 0) AS total_comprado
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id
GROUP BY c.nombre
ORDER BY total_comprado DESC;
```

Explicación paso a paso. Usamos LEFT JOIN porque queremos a todos los clientes, incluso sin compras. Agrupamos por cliente. COUNT(p.id) cuenta los pedidos: como p.id es NULL para los clientes sin pedidos, el COUNT los cuenta como cero (COUNT ignora los NULL,

detalle útil). SUM(p.total) suma los totales, y COALESCE convierte el NULL en 0 para los que no compraron. Los alias hacen la consulta legible: c, p, y nombres de columna de salida claros (cliente, num_pedidos, total_comprado).

El resultado: Luis Gomez, 1 pedido, 780.00. Ana Torres, 2 pedidos, 570.00. Maria Ruiz, 1 pedido, 95.00. Carlos Diaz, 0 pedidos, 0. Sofia Vega, 0 pedidos, 0. Fíjate que Ana tiene más pedidos que Luis pero Luis compró más en total: el reporte con agregación cuenta una historia que la lista simple de pedidos no contaba. Y los clientes en cero aparecen gracias al LEFT JOIN: con INNER JOIN habrían desaparecido y el reporte habría mentido por omisión.

Ejemplo 7: self join, empleados y sus jefes

Para este ejemplo necesitamos una tablita extra. Crea y llénala:

```
CREATE TABLE empleados (  
    id INT PRIMARY KEY,  
    nombre VARCHAR(100),  
    jefe_id INT  
);  
  
INSERT INTO empleados (id, nombre, jefe_id) VALUES  
(1, 'Esteban', NULL),  
(2, 'Ana', 1),  
(3, 'Luis', 1),  
(4, 'Maria', 2);
```

Pregunta: quiero ver cada empleado junto al nombre de su jefe.

```
SELECT e.nombre AS empleado, j.nombre AS jefe  
FROM empleados e  
LEFT JOIN empleados j ON e.jefe_id = j.id  
ORDER BY e.id;
```

Explicación paso a paso. Unimos la tabla empleados consigo misma: la copia e representa a los empleados y la copia j representa a los jefes. El ON empareja e.jefe_id con j.id. Usamos LEFT JOIN (no INNER) porque Esteban no tiene jefe (jefe_id NULL) y con INNER desaparecería del resultado. El resultado: Esteban, jefe NULL. Ana, jefe Esteban. Luis, jefe Esteban. Maria, jefe Ana.

Este ejemplo demuestra dos cosas: que los alias de tabla son obligatorios en un self join (sin ellos el motor no distingue las dos copias), y que el LEFT JOIN es el que te permite ver la fila "huerfana" (el director sin jefe). Si hubieras usado INNER JOIN, Esteban no saldría y el reporte quedaría incompleto sin que nadie se diera cuenta.

Ejemplo 8: el error clásico del WHERE en un LEFT JOIN

Vamos a ver la trampa en acción con nuestros datos, para que la reconozcas cuando te pase. Esta consulta pretende decir "todos los clientes y sus pedidos mayores a 200":

```
SELECT c.nombre AS cliente, p.total
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id
WHERE p.total > 200;
```

Explicación de lo que sale mal. El LEFT JOIN genera las filas con NULL para Carlos y Sofia. Luego el WHERE evalúa `p.total > 200`: para los NULL el resultado no es verdadero, así que Carlos y Sofia se eliminan. También se eliminan los pedidos de 120 y 95. El resultado trae solo 2 filas: Ana Torres 450.00 y Luis Gomez 780.00. Los clientes sin pedidos desaparecieron: tu LEFT JOIN se convirtió silenciosamente en un INNER JOIN con filtro.

La versión correcta pone la condición en el ON:

```
SELECT c.nombre AS cliente, p.total
FROM clientes c
LEFT JOIN pedidos p ON p.cliente_id = c.id AND p.total > 200;
```

Ahora el ON empareja cada cliente solo con sus pedidos mayores a 200. Ana aparece con 450.00 (su pedido de 120 no empareja). Luis aparece con 780.00. Maria aparece con total NULL (su pedido de 95 no empareja, así que queda como si no tuviera). Carlos y Sofia aparecen con NULL. Los 5 clientes están presentes. Esta es LA diferencia entre filtrar en el ON y filtrar en el WHERE con LEFT JOIN. Práctica este ejemplo hasta que te salga natural.

Ejemplo 9: top de clientes con HAVING

Pregunta: que clientes han comprado más de 500 en total? Muestra el nombre y el total.

```
SELECT c.nombre AS cliente, SUM(p.total) AS total_comprado
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id
GROUP BY c.nombre
HAVING SUM(p.total) > 500
ORDER BY total_comprado DESC;
```

Explicación paso a paso. Unimos clientes con pedidos (INNER JOIN porque solo nos interesan los que compraron). Agrupamos por cliente y sumamos. El filtro "más de 500" no puede ir en el WHERE porque se aplica sobre el total ya sumado, no sobre filas individuales: para eso existe el HAVING, que filtra grupos después de agregar. Es el mismo orden de evaluación que vimos en la teoría: WHERE filtra filas, HAVING filtra grupos.

El resultado: Luis Gomez, 780.00. Ana Torres, 570.00. Maria Ruiz queda fuera con 95.00. Dos filas. Este ejemplo conecta los joins con la agregación de la unidad anterior: en la vida real casi siempre van juntos. Y nota que usamos el alias total_comprado en el ORDER BY (permitido, porque ORDER BY va después del SELECT) pero repetimos SUM(p.total) en el HAVING en lugar del alias: la mayoría de los motores acepta el alias en el HAVING, pero repetir la expresión es más portable y nunca falla.

Ejemplo 10: reporte de inventario vendido con COALESCE

Pregunta: para cada producto, muestra cuantas unidades se han vendido en total, incluyendo los productos con cero ventas.

```
SELECT pr.nombre AS producto,
       COALESCE(SUM(d.cantidad), 0) AS unidades_vendidas
FROM productos pr
LEFT JOIN detalle_pedido d ON d.producto_id = pr.id
GROUP BY pr.nombre
ORDER BY unidades_vendidas DESC;
```

Explicación paso a paso. La tabla izquierda es productos porque queremos todos los productos. El LEFT JOIN los empareja con sus renglones de venta. Agrupamos por producto y sumamos cantidades. Para un producto sin ventas, SUM(d.cantidad) da NULL (no hay filas que sumar), y COALESCE lo convierte en 0 para que el reporte se lea limpio.

El resultado con nuestros datos: Libro de Python, 3. Curso de CSS3, 3 (1 de Ana + 2 de Maria). Curso de HTML5, 3 (2 de Ana + 1 de Luis). Libro de SQL, 1. Todos los productos aparecen porque todos tuvieron ventas; si agregaras un quinto producto sin ventas, saldría con 0 gracias al LEFT JOIN. Con INNER JOIN habría desaparecido. Este patrón (LEFT JOIN +

GROUP BY + COALESCE) es el estandar para reportes de "todo el catálogo con sus metricas, incluyendo los ceros", y lo verás en dashboards de ventas, inventarios y paneles de control toda tu carrera.

Práctica

Usa las mismas tablas de la sección de ejemplos (clientes, pedidos, productos, detalle_pedido y empleados). Intenta resolver cada ejercicio por tu cuenta antes de leer la solución. La gracia está en equivocarte aquí y no en producción.

Ejercicio 1: pedidos con nombre de cliente y ciudad

Escribe una consulta que muestre el id del pedido, la fecha, el nombre del cliente y su ciudad, solo para los pedidos con total mayor a 100. Ordena por total de mayor a menor.

Solución:

```
SELECT p.id AS pedido, p.fecha, c.nombre AS cliente, c.ciudad, p.total
FROM pedidos p
INNER JOIN clientes c ON p.cliente_id = c.id
WHERE p.total > 100
ORDER BY p.total DESC;
```

Explicación paso a paso. Primero identificamos las tablas: pedidos tiene el id, la fecha y el total; clientes tiene el nombre y la ciudad. Las unimos con INNER JOIN por p.cliente_id = c.id, porque solo nos interesan los pedidos con su cliente (todos los pedidos tienen cliente valido en nuestros datos). El filtro p.total > 100 va en el WHERE porque es un INNER JOIN: aquí da igual si va en el ON o en el WHERE, el resultado es el mismo, y por convención los filtros sobre una sola tabla van en el WHERE. Ordenamos por total descendente.

El resultado: pedido 103, Luis Gomez, Guadalajara, 780.00. Pedido 101, Ana Torres, Ciudad de Mexico, 450.00. Pedido 102, Ana Torres, Ciudad de Mexico, 120.00. El pedido 104 (95.00) queda fuera por el filtro. Tres filas en total.

Ejercicio 2: productos que nadie ha comprado

Encuentra los productos que no aparecen en ningun pedido. Muestra su nombre y precio.

Solución:

```
SELECT pr.nombre AS producto, pr.precio
FROM productos pr
LEFT JOIN detalle_pedido d ON d.producto_id = pr.id
WHERE d.pedido_id IS NULL;
```

Explicación paso a paso. La pregunta es "que falta", así que el patrón es LEFT JOIN + IS NULL. La tabla izquierda es productos porque queremos revisar todos los productos. La unimos con detalle_pedido por producto_id: cada renglon de detalle indica que ese producto se vendió en ese pedido. El WHERE d.pedido_id IS NULL se queda con los productos que no emparejaron con ningun renglon de detalle, es decir, los nunca vendidos. Usamos d.pedido_id (columna que no puede ser NULL en datos reales) para detectar la ausencia.

El resultado: ningun producto? Revisemos los datos: el producto 1 (Curso de HTML5) aparece en los pedidos 101 y 103. El producto 2 (Curso de CSS3) aparece en el 102 y el 104. El producto 3 (Libro de SQL) aparece en el 101. El producto 4 (Libro de Python) aparece en el 103. Todos se vendieron al menos una vez, así que el resultado trae cero filas. Y eso también es una respuesta valida y útil: "todos los productos han tenido al menos una venta". Si el resultado vacío te sorprende, verifica tus datos antes de asumir que la consulta está mal: en este caso la consulta está bien, los datos son los que no tienen productos sin ventas. Para practicar el patrón con un resultado no vacío, inserta un producto nuevo sin ventas y vuelve a ejecutar.

Ejercicio 3: ticket promedio por ciudad

Calcula, por cada ciudad, cuantos pedidos se hicieron y cuál fue el total vendido. Incluye solo las ciudades con al menos un pedido.

Solución:

```
SELECT c.ciudad,
       COUNT(p.id) AS num_pedidos,
       SUM(p.total) AS total_vendido
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id
GROUP BY c.ciudad
ORDER BY total_vendido DESC;
```

Explicación paso a paso. Necesitamos la ciudad (tabla clientes) y los pedidos (tabla pedidos). Como solo nos interesan las ciudades con al menos un pedido, usamos INNER JOIN: las ciudades sin pedidos desaparecen solas, sin necesidad de filtro extra. Agrupamos por ciudad y

calculamos el conteo y la suma. COUNT(p.id) cuenta pedidos (p.id nunca es NULL en un INNER JOIN, así que cuenta todas las filas del grupo).

El resultado: Guadalajara, 1 pedido, 780.00 (solo Luis compró desde ahí). Ciudad de Mexico, 3 pedidos, 665.00 (450 + 120 de Ana, 95 de Maria). Monterrey no aparece porque Carlos no tiene pedidos y el INNER JOIN lo excluye. Dos filas. Observa como la elección del tipo de JOIN responde directamente a la pregunta: "solo ciudades con al menos un pedido" = INNER JOIN. Si la pregunta hubiera sido "todas las ciudades aunque no hayan vendido", habría sido LEFT JOIN con COALESCE.

Ejercicio 4: clientes y lo que compraron, con DISTINCT

Muestra la lista de clientes que han comprado el "Curso de HTML5", sin repetir nombres aunque hayan comprado varias veces. Incluye la ciudad.

Solución:

```
SELECT DISTINCT c.nombre AS cliente, c.ciudad
FROM clientes c
INNER JOIN pedidos p ON p.cliente_id = c.id
INNER JOIN detalle_pedido d ON d.pedido_id = p.id
INNER JOIN productos pr ON pr.id = d.producto_id
WHERE pr.nombre = 'Curso de HTML5';
```

Explicación paso a paso. La cadena es clientes -> pedidos -> detalle_pedido -> productos, igual que en el ejemplo 2. El WHERE filtra solo los renglones del Curso de HTML5. Sin DISTINCT, Ana Torres aparecería una vez por cada compra del curso (en nuestros datos lo compró una vez, en el pedido 101 con cantidad 2, pero si hubiera comprado en dos pedidos distintos saldría dos veces). El DISTINCT colapsa los duplicados y deja una fila por cliente.

El resultado: Ana Torres, Ciudad de Mexico. Luis Gomez, Guadalajara. Dos filas. Nota que el filtro por nombre del producto va en el WHERE: cómo es INNER JOIN, ponerlo en el WHERE o en el ON del último JOIN da el mismo resultado. Por legibilidad, los filtros sobre una sola tabla van en el WHERE.

Variante para pensar: si en lugar de DISTINCT usaras GROUP BY c.nombre, c.ciudad obtendrías lo mismo. Cual comunica mejor la intención? DISTINCT dice "quiero la lista sin repetidos". GROUP BY dice "quiero agrupar", y el lector esperaría una agregación (COUNT, SUM) que no hay. Para este caso, DISTINCT es la elección más clara.

Ejercicio 5: reporte completo con LEFT JOIN y alias

Genera un reporte con todos los clientes, el número de pedidos de cada uno y el total comprado. Los clientes sin pedidos deben aparecer con ceros. Ordena por total comprado de mayor a menor y usa alias en todo.

Solución:

```
SELECT c.nombre AS cliente,  
       c.ciudad,  
       COUNT(p.id) AS num_pedidos,  
       COALESCE(SUM(p.total), 0) AS total_comprado  
FROM clientes AS c  
LEFT JOIN pedidos AS p ON p.cliente_id = c.id  
GROUP BY c.nombre, c.ciudad  
ORDER BY total_comprado DESC;
```

Explicación paso a paso. Queremos TODOS los clientes, así que la base es clientes con LEFT JOIN hacia pedidos. Aquí escribimos los alias con la palabra AS explícita (FROM clientes AS c) para mostrar que también es válido; en los demás ejemplos la omitimos. Es cuestión de estilo: se consistente en tu código. Agrupamos por cliente (incluimos la ciudad en el GROUP BY porque aparece en el SELECT sin agregación: la mayoría de los motores lo exige). COUNT(p.id) da 0 para los sin pedidos porque COUNT ignora NULL. SUM(p.total) da NULL para los sin pedidos, y COALESCE lo convierte en 0.

El resultado: Luis Gomez, Guadalajara, 1, 780.00. Ana Torres, Ciudad de Mexico, 2, 570.00. Maria Ruiz, Ciudad de Mexico, 1, 95.00. Carlos Diaz, Monterrey, 0, 0. Sofia Vega, Guadalajara, 0, 0. Cinco filas, nadie falta. Este es el reporte que un gerente quiere ver: completo, sin omisiones silenciosas. Compara con lo que habría dado un INNER JOIN: solo 3 filas, y los dos clientes inactivos invisibles. Esa es la diferencia entre un reporte honesto y uno mentiroso.

Ejercicio 6: detecta el error

La siguiente consulta intenta mostrar "todos los empleados con el nombre de su jefe, incluyendo al director que no tiene jefe". Pero tiene un error y el director no aparece. Encuéntralo y corrígelo.

```
SELECT e.nombre AS empleado, j.nombre AS jefe  
FROM empleados e  
INNER JOIN empleados j ON e.jefe_id = j.id;
```

Solución: el error es el tipo de JOIN. Con INNER JOIN solo aparecen los empleados que tienen pareja en la tabla de jefes, es decir, los que tienen jefe_id con un id valido. Esteban tiene jefe_id NULL, así que no empareja con nadie y desaparece del resultado. La consulta corregida usa LEFT JOIN:

```
SELECT e.nombre AS empleado, j.nombre AS jefe
FROM empleados e
LEFT JOIN empleados j ON e.jefe_id = j.id
ORDER BY e.id;
```

Explicación paso a paso. El LEFT JOIN trae a todos los empleados (tabla izquierda = e). Los que tienen jefe se emparejan y muestran su nombre; Esteban queda con jefe en NULL. El resultado: Esteban, NULL. Ana, Esteban. Luis, Esteban. Maria, Ana. Cuatro filas.

La lección general: cada vez que una consulta "pierde" filas que esperabas ver, el primer sospechoso es un INNER JOIN donde debia ir un LEFT JOIN. Y al revés: si un LEFT JOIN te trae filas con NULL que no esperabas, probablemente querias un INNER. El tipo de JOIN es una decisión sobre que filas deben sobrevivir, no un detalle de sintaxis.

Ejercicio 7: reescribe un RIGHT JOIN como LEFT JOIN

La siguiente consulta usa RIGHT JOIN para mostrar todos los pedidos con el nombre de su cliente. Reescribela usando LEFT JOIN y explica por qué ambas devuelven lo mismo.

```
SELECT c.nombre AS cliente, p.id AS pedido, p.total
FROM clientes c
RIGHT JOIN pedidos p ON p.cliente_id = c.id;
```

Solución:

```
SELECT c.nombre AS cliente, p.id AS pedido, p.total
FROM pedidos p
LEFT JOIN clientes c ON p.cliente_id = c.id;
```

Explicación paso a paso. El RIGHT JOIN toma como "tabla completa" la de la derecha (pedidos): trae todos los pedidos, con el nombre del cliente cuando exista. La versión con LEFT JOIN simplemente voltea el orden: pone pedidos en el FROM (ahora es la tabla izquierda) y une clientes a la derecha. El LEFT JOIN trae todo lo de la izquierda, que otra vez son todos los pedidos. La condición del ON es la misma, solo cambia de que lado se escribe cada tabla.

Ambas devuelven las mismas 4 filas: los 4 pedidos con sus clientes. La moraleja es de estilo, no de lógica: los equipos prefieren leer siempre LEFT JOIN porque el ojo recorre la consulta en una sola dirección (la tabla importante primero, los complementos después). Cuando encuentres un RIGHT JOIN en código ajeno, tradúcelo mentalmente a LEFT JOIN volteando las tablas y todo se vuelve más claro. Y cuando escribas código nuevo, usa LEFT JOIN directamente: tu yo del futuro te lo agradecerá.

Resumen de la unidad

En esta unidad aprendiste el superpoder de SQL: combinar tablas con JOIN. La idea central es simple: las bases de datos relacionales guardan cada dato una sola vez en tablas separadas y relacionadas por llaves, y el JOIN reconstruye la foto completa emparejando filas según una condición, casi siempre la igualdad entre llave foránea y llave primaria.

El INNER JOIN trae solo las filas con pareja en ambas tablas: es el más usado y resuelve la mayoría de los casos. El LEFT JOIN trae todo lo de la tabla izquierda, con NULL donde no haya pareja: es la herramienta para las preguntas de "que falta", combinada con IS NULL para encontrar clientes sin pedidos, productos sin ventas y cualquier ausencia que necesites detectar. El RIGHT JOIN es su espejo y en la práctica se reemplaza volteando las tablas con un LEFT JOIN. El FULL JOIN trae todo de ambos lados y el CROSS JOIN multiplica cada fila con cada fila: reconoce ambos, usa el primero para conciliar fuentes y evita el segundo salvo que busques combinaciones a propósito.

Viste que los joins se encadenan: cada JOIN nuevo se apoya en una tabla ya unida, y puedes unir tres, cinco o más tablas en una sola consulta siguiendo la cadena de llaves. Conociste el self join, donde una tabla se une consigo misma usando alias obligatorios, útil para jerarquías como empleados y jefes.

Aprendiste los alias con AS: nombres cortos para tablas (c, p, d) que vuelven legibles las consultas con joins, y nombres claros para columnas que evitan ambigüedades en el resultado. El AS es opcional pero escribirlo comunica intención.

Dominaste el DISTINCT para eliminar duplicados: actúa sobre la fila completa, sirve para listas de valores únicos, y es el compañero natural del JOIN para limpiar las repeticiones que los joins generan. Aprendiste a distinguirlo del GROUP BY: DISTINCT para listas únicas, GROUP BY para resumir con cálculos.

Y lo más valioso: los errores que debes evitar. Nunca dejes un JOIN sin su ON o terminas con un producto cartesiano. Califica tus columnas o los nombres repetidos te darán errores de ambigüedad. No filtres columnas de la tabla derecha en el WHERE de un LEFT JOIN o lo

convertirás silenciosamente en un INNER: esas condiciones van en el ON. Une por las llaves correctas y desconfía de resultados que "casi" cuadran. Y cuando agregues sobre joins múltiples, entiende la cardinalidad primero para no contar de más.

Como buenas prácticas, quédate con estas: índices en las columnas del ON, pide solo las columnas que necesites, prefiere la sintaxis explícita con JOIN y ON, usa alias consistentes, prueba tus joins por partes contando filas en cada paso, y elige el tipo de JOIN según la intención (si los NULL no importan, no uses LEFT JOIN).

En la siguiente unidad seguiremos subiendo de nivel con subconsultas y operaciones de conjuntos, donde los joins que dominas aquí se combinan con nuevas herramientas. Práctica estos ejercicios hasta que el LEFT JOIN con IS NULL te salga sin pensar: es una de esas técnicas que separan al que consulta bases de datos del que de verdad las domina.

UNIDAD 8: CONSULTAS AVANZADAS

Ya llegaste lejos. Si dominas todo lo que vimos en las unidades anteriores, puedes crear bases de datos, llenarlas de datos, consultarlas con filtros, ordenar resultados, agrupar información y conectar tablas con JOINS. Con eso ya puedes resolver la gran mayoría de los problemas que aparecen en el trabajo diario de un desarrollador. Pero la diferencia entre alguien que "sabe SQL" y alguien que es un consultor de datos de verdad está justo aquí, en esta unidad.

Las consultas avanzadas son las que separan a los que apenas sobreviven de los que destacan. Cuando un gerente te pide un reporte que parece imposible, cuando la aplicación se vuelve lenta con miles de usuarios, cuando la consulta que escribiste regresa datos incorrectos en un caso especial: ahí es donde brillan las subconsultas, los índices y las vistas. Son las tres herramientas que vamos a dominar en esta unidad, correspondientes a las clases 25, 26 y 27 del curso.

Las subconsultas te permiten hacer preguntas dentro de preguntas. Imagina que quieres saber qué productos cuestan más que el precio promedio de todos los productos. Son dos preguntas en una: primero tienes que calcular el promedio, y después comparar cada producto contra ese promedio. Una subconsulta te deja escribir eso en una sola instrucción SQL elegante. También sirven para verificar existencia de datos relacionados, para filtrar con listas dinámicas y para crear tablas temporales al vuelo dentro de tu consulta.

Los índices son la herramienta de rendimiento más importante que tiene SQL. Una base de datos sin índices es como una biblioteca donde los libros están aventados al azar en el piso: para encontrar un libro específico tienes que revisar uno por uno hasta dar con él. Un índice es como el catálogo de la biblioteca, o el índice al final de un libro: te dice exactamente en qué página está cada tema sin tener que leer todo. Cuando aprendas a crear y usar índices correctamente, tus consultas pasarán de tardar segundos a tardar milisegundos. Y cuando los uses mal, aprenderás por qué a veces una consulta "optimizada" termina siendo más lenta que la original.

Las vistas son tablas virtuales que guardan una consulta con un nombre. Te permiten simplificar consultas complejas, dar acceso controlado a los datos y reutilizar lógica que se repite. Una vista bien diseñada puede convertir una consulta monstruosa de 50 líneas en algo tan simple como `SELECT * FROM vista_reporte_mensual`. Son una de las herramientas favoritas de los administradores de bases de datos y de los equipos que trabajan con reportes.

En esta unidad vas a ver cada tema a fondo, con la teoría bien explicada, analogías de la vida real, ejemplos paso a paso que puedes copiar y probar, errores comunes que cometen hasta

los programadores con experiencia, y buenas prácticas de la industria. Al final habrá práctica con ejercicios y soluciones explicadas, y un resumen para repasar rápido.

Un consejo antes de empezar: esta unidad es densa. No la leas de corrido como si fuera una novela. Lee una sección, abre tu gestor de base de datos, prueba los ejemplos, rompe cosas a propósito y arréglalas. La única forma de dominar las consultas avanzadas es escribiéndolas tú mismo. Vamos a darle.

Explicación detallada

Subconsultas: preguntas dentro de preguntas

Una subconsulta es una consulta `SELECT` escrita dentro de otra consulta. La consulta de adentro se ejecuta primero y su resultado lo usa la consulta de afuera. También se les llama consultas anidadas o consultas internas. La idea es simple: hay preguntas que no puedes responder con una sola consulta plana, porque necesitas el resultado de un cálculo intermedio para poder filtrar o comparar.

Piénsalo así. Tu jefe te pregunta: "¿Cuáles empleados ganan más que el salario promedio de la empresa?". Tú no puedes responder eso de un solo vistazo. Primero tienes que calcular el promedio de todos los salarios, y ya con ese número en la mano, revisar empleado por empleado quién lo supera. La subconsulta hace exactamente eso, pero en una sola instrucción:

```
SELECT nombre, salario
FROM empleados
WHERE salario > (SELECT AVG(salario) FROM empleados);
```

La parte entre paréntesis es la subconsulta. Se ejecuta primero, calcula el promedio, y ese número se convierte en el valor contra el que se compara cada salario. Nota que la subconsulta va entre paréntesis: eso es obligatorio siempre. Sin los paréntesis, el motor de base de datos no sabe dónde empieza y dónde termina la consulta interna.

Hay varios tipos de subconsultas según qué regresan y dónde se usan. Vamos a verlos uno por uno porque cada tipo tiene sus reglas y sus usos.

Subconsultas escalares: regresan un solo valor

Una subconsulta escalar regresa exactamente un valor: una sola fila y una sola columna. Es el tipo más simple y el más fácil de entender. Se usa en el WHERE para comparar contra un valor calculado, o en el SELECT para agregar una columna calculada a cada fila.

El ejemplo del salario promedio es una subconsulta escalar. Otro ejemplo clásico: quieres los productos más caros que el promedio.

```
SELECT nombre, precio
FROM productos
WHERE precio > (SELECT AVG(precio) FROM productos);
```

También puedes usar una subconsulta escalar en la lista del SELECT, para agregar una columna calculada:

```
SELECT nombre, precio,
       (SELECT AVG(precio) FROM productos) AS precio_promedio
FROM productos;
```

Esto agrega a cada fila una columna extra con el promedio general. Útil para reportes donde quieres comparar cada valor contra el promedio sin hacer dos consultas separadas.

Regla de oro de las subconsultas escalares: deben regresar un solo valor. Si la subconsulta regresa más de una fila, la base de datos te va a marcar error. Por ejemplo, esto falla:

```
SELECT nombre FROM empleados
WHERE salario > (SELECT salario FROM empleados WHERE departamento = 'Ventas');
```

Si hay más de un empleado en Ventas, la subconsulta regresa varios salarios y el operador > no sabe contra cuál comparar. Para esos casos necesitas IN, ANY o ALL, o asegurarte de que la subconsulta regrese un solo valor usando funciones de agregado como AVG, MAX, MIN, SUM o COUNT.

Subconsultas de una columna con IN y NOT IN

Cuando la subconsulta regresa una sola columna pero varias filas, la usas con el operador IN. IN verifica si un valor está dentro de una lista. La subconsulta genera esa lista dinámicamente.

Ejemplo: quieres los clientes que han hecho al menos un pedido. En lugar de escribir los IDs a mano, dejas que la subconsulta los genere:

```
SELECT nombre, email
FROM clientes
WHERE id IN (SELECT cliente_id FROM pedidos);
```

La subconsulta (SELECT cliente_id FROM pedidos) regresa la lista de todos los IDs de clientes que aparecen en la tabla de pedidos. El IN revisa cada cliente: si su ID está en esa lista, pasa el filtro.

NOT IN hace lo contrario: regresa los que NO están en la lista. Para encontrar clientes que nunca han comprado:

```
SELECT nombre, email
FROM clientes
WHERE id NOT IN (SELECT cliente_id FROM pedidos);
```

Cuidado con una trampa famosa de NOT IN: si la subconsulta regresa aunque sea un solo valor NULL, NOT IN no regresa ninguna fila. Esto pasa porque comparar algo contra NULL da como resultado desconocido, no verdadero ni falso, y el WHERE solo deja pasar lo que es verdadero. Es uno de los errores más confusos para los principiantes. Si sospechas que puede haber NULLs, usa NOT EXISTS en lugar de NOT IN, o filtra los NULLs dentro de la subconsulta:

```
SELECT nombre, email
FROM clientes
WHERE id NOT IN (SELECT cliente_id FROM pedidos WHERE cliente_id IS NOT NULL);
```

También existen los operadores ANY y ALL, que comparan un valor contra cada valor de la lista. Por ejemplo, "> ANY" significa "mayor que al menos uno de los valores", y "> ALL" significa "mayor que todos los valores". Son menos comunes en el día a día, pero es bueno saber que existen:

```
SELECT nombre, precio
FROM productos
WHERE precio > ANY (SELECT precio FROM productos WHERE categoria = 'Libros');
```

Esto regresa los productos que cuestan más que al menos un libro. En la práctica, la mayoría de los desarrolladores prefieren usar MIN o MAX con subconsultas escalares porque se leen más claro, pero ANY y ALL aparecen en exámenes y en código heredado.

Subconsultas en el FROM: tablas derivadas

Puedes poner una subconsulta en el FROM, y su resultado se comporta como una tabla temporal. A esto se le llama tabla derivada. Es muy útil cuando necesitas primero agregar o transformar datos y después consultar sobre ese resultado.

Ejemplo: quieres saber el total de ventas por cliente, pero solo de los clientes cuyo total supere los 10,000 pesos. Primero calculas los totales por cliente, y después filtras:

```
SELECT cliente, total_ventas
FROM (SELECT c.nombre AS cliente, SUM(p.monto) AS total_ventas
      FROM clientes c
      JOIN pedidos p ON c.id = p.cliente_id
      GROUP BY c.nombre) AS resumen
WHERE total_ventas > 10000;
```

La subconsulta del FROM calcula el total por cliente y ese resultado se trata como una tabla llamada "resumen". Después el WHERE filtra sobre esa tabla temporal.

Regla obligatoria: toda subconsulta en el FROM debe tener un alias. Sin el AS resumen (o al menos el nombre "resumen" sin el AS), la mayoría de los motores te marcan error de sintaxis. El alias le da nombre a la tabla temporal para que puedas referenciar sus columnas. Es una de las reglas que más se les olvida a los principiantes, así que grábatela: subconsulta en FROM, alias siempre.

Las tablas derivadas también sirven para resolver el problema de "no puedes usar un alias de columna en el WHERE de la misma consulta". Por ejemplo, esto falla en la mayoría de los motores:

```
SELECT nombre, precio * 1.16 AS precio_con_iva
FROM productos
WHERE precio_con_iva > 500;
```

El WHERE se evalúa antes que el SELECT, así que el alias todavía no existe cuando se filtra. La solución es envolverlo en una tabla derivada:

```
SELECT nombre, precio_con_iva
FROM (SELECT nombre, precio * 1.16 AS precio_con_iva
      FROM productos) AS con_iva
WHERE precio_con_iva > 500;
```

Subconsultas correlacionadas y EXISTS

Hasta ahora las subconsultas que vimos son independientes: se ejecutan una sola vez y la consulta exterior usa su resultado. Pero hay otro tipo más poderoso: las subconsultas correlacionadas. Estas hacen referencia a la consulta exterior, así que se evalúan una vez por cada fila de la tabla exterior. Suenan ineficientes, y a veces lo son, pero resuelven problemas que de otra forma serían muy complicados.

El operador EXISTS es el compañero clásico de las subconsultas correlacionadas. EXISTS regresa verdadero si la subconsulta encuentra al menos una fila, y falso si no encuentra nada. No le importa qué columnas regresa la subconsulta, solo si hay filas o no. Por eso es común ver SELECT 1 dentro de un EXISTS: el 1 es un marcador, lo que importa es la existencia.

Ejemplo: clientes que tienen al menos un pedido:

```
SELECT nombre, email
FROM clientes c
WHERE EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Fíjate en `p.cliente_id = c.id`. La subconsulta usa `c.id`, que viene de la consulta exterior. Por cada cliente, la base de datos revisa si existe al menos un pedido con ese ID. En cuanto encuentra uno, deja de buscar y pasa al siguiente cliente. Por eso EXISTS suele ser más rápido que IN con listas grandes: no necesita construir la lista completa, se detiene en la primera coincidencia.

NOT EXISTS hace lo contrario y no tiene la trampa del NULL que tiene NOT IN:

```
SELECT nombre, email
FROM clientes c
WHERE NOT EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Esto regresa los clientes sin pedidos, y funciona correctamente aunque haya NULLs. Por eso muchos desarrolladores experimentados prefieren EXISTS y NOT EXISTS sobre IN y NOT IN: son más seguros y a menudo más rápidos.

Las subconsultas correlacionadas también sirven para comparaciones fila por fila. Ejemplo clásico: empleados que ganan más que el promedio de SU departamento (no el promedio general):

```
SELECT nombre, departamento, salario
FROM empleados e
WHERE salario > (SELECT AVG(salario)
                 FROM empleados
                 WHERE departamento = e.departamento);
```

Aquí la subconsulta se ejecuta una vez por cada empleado, calculando el promedio solo de su departamento. Es elegante y legible, aunque con tablas enormes puede ser lenta. Más adelante veremos cuándo conviene reescribirla con JOIN.

Subconsulta vs JOIN: cuándo usar cada una

Esta es una de las decisiones más comunes en SQL y una de las preguntas favoritas en entrevistas de trabajo. Muchas consultas se pueden escribir tanto con subconsulta como con JOIN. ¿Cuál elegir?

Regla general: si necesitas columnas de ambas tablas en el resultado, usa JOIN. Si solo necesitas filtrar o verificar existencia basándote en otra tabla, la subconsulta (especialmente con EXISTS) suele ser más clara.

Ejemplo con JOIN: clientes con sus pedidos, mostrando datos de ambas tablas:

```
SELECT c.nombre, p.fecha, p.monto
FROM clientes c
JOIN pedidos p ON c.id = p.cliente_id;
```

Aquí necesitas columnas de las dos tablas, así que el JOIN es la herramienta natural.

Ejemplo con EXISTS: solo quieres la lista de clientes que tienen pedidos, sin datos de los pedidos:

```
SELECT nombre, email
FROM clientes c
WHERE EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Con JOIN tendrías que agregar un DISTINCT para evitar duplicados (un cliente con 5 pedidos aparecería 5 veces), lo cual es más confuso y a veces más lento. EXISTS expresa exactamente la intención: "clientes para los que existe al menos un pedido".

Comparativa rápida:

- Legibilidad: EXISTS y las subconsultas expresan mejor la intención cuando solo filtras. JOIN expresa mejor la intención cuando combinas datos.
- Duplicados: JOIN puede multiplicar filas cuando hay relaciones uno a muchos; EXISTS nunca duplica porque solo verifica existencia.
- Rendimiento: depende del motor y del tamaño de los datos. En general, los motores modernos optimizan bien ambos. EXISTS suele ganar cuando la subconsulta puede detenerse temprano. JOIN suele ganar cuando necesitas agregar datos de la tabla relacionada.
- Mantenibilidad: una subconsulta bien nombrada es más fácil de leer seis meses después que un JOIN con DISTINCT y GROUP BY pegados con cinta adhesiva.

Mi recomendación práctica: escribe primero la versión que te resulte más clara. Si la consulta es lenta con datos reales, mide y prueba la alternativa. No optimices por adivinanza.

Índices: el catálogo de la biblioteca

Un índice es una estructura de datos que la base de datos mantiene para encontrar filas rápidamente sin tener que revisar la tabla completa. La analogía clásica es el índice al final de un libro: si quieres encontrar todas las menciones de "fotosíntesis" en un libro de 500 páginas, no lees las 500 páginas. Vas al índice, buscas "fotosíntesis", y te dice que está en las páginas 42, 187 y 350. Un índice en base de datos hace exactamente lo mismo: guarda los valores de una columna ordenados junto con la ubicación de cada fila.

Sin índice, para encontrar un cliente por su email la base de datos tiene que hacer un "table scan": revisar fila por fila toda la tabla hasta encontrarlo. Con un millón de filas, eso es un millón de comparaciones. Con un índice sobre la columna email, la base de datos salta directamente a la fila correcta en milisegundos. La diferencia en tablas grandes es brutal: de segundos a milisegundos.

La mayoría de los motores implementan los índices como árboles B (B-trees), que permiten buscar, insertar y mantener el orden de forma eficiente. No necesitas entender los árboles B a fondo para usar índices, pero sí entender la idea: el índice mantiene tus valores ordenados y con apuntadores a las filas, así que buscar es rapidísimo.

CREATE INDEX: creando tu primer índice

La sintaxis básica es sencilla:

```
CREATE INDEX idx_clientes_email ON clientes(email);
```

Esto crea un índice llamado `idx_clientes_email` sobre la columna `email` de la tabla `clientes`. La convención de nombres `idx_tabla_columna` no es obligatoria, pero te recomiendo usarla: cuando tengas 50 índices, agradecerás poder saber de un vistazo a qué tabla y columna pertenece cada uno.

Después de crear el índice, las consultas que filtren u ordenen por `email` lo usarán automáticamente. Tú no tienes que decirle a la base de datos que use el índice; el optimizador de consultas decide solo. Tú solo escribes tu `SELECT` normal:

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

El motor detecta que hay un índice sobre `email` y lo usa. Si haces un `EXPLAIN` (una instrucción que muestra el plan de ejecución), verás que dice que usó el índice en lugar de hacer un `table scan`.

También puedes crear índices únicos, que además de acelerar búsquedas garantizan que no haya duplicados:

```
CREATE UNIQUE INDEX idx_clientes_email_unico ON clientes(email);
```

Esto es útil para columnas como `email`, `CURP`, `RFC` o número de serie, donde los duplicados serían un error de datos. Si intentas insertar un `email` repetido, la base de datos lo rechaza.

Índices compuestos: más de una columna

Un índice compuesto (o multicolumna) indexa varias columnas juntas en un orden específico:

```
CREATE INDEX idx_pedidos_cliente_fecha ON pedidos(cliente_id, fecha);
```

Este índice acelera las consultas que filtran por `cliente_id`, o por `cliente_id` y `fecha` juntos. Pero ojo con el orden: el índice funciona de izquierda a derecha. Una consulta que filtre solo por `fecha` (sin `cliente_id`) NO puede usar eficientemente este índice, porque la primera columna del índice no está en el filtro. Es como un directorio telefónico ordenado por apellido y luego por nombre: puedes buscar rápido por apellido, o por apellido y nombre, pero no puedes buscar rápido solo por nombre.

El orden de las columnas en el índice compuesto importa mucho. Pon primero la columna por la que filtras con más frecuencia o la que tiene más valores distintos (mayor cardinalidad). Si siempre consultas por `cliente_id` y a veces también por `fecha`, el orden (`cliente_id, fecha`) es el correcto.

Cuándo Sí crear índices y cuándo NO

Crear índices no es gratis. Cada índice ocupa espacio en disco y, lo más importante, cada vez que insertas, actualizas o borras una fila, la base de datos tiene que actualizar TODOS los índices de la tabla. En una tabla donde se escribe mucho y se lee poco, demasiados índices pueden volver lentas las inserciones.

Crea índices cuando:

- La columna aparece frecuentemente en cláusulas WHERE, especialmente con comparaciones de igualdad (=) o rangos (>, <, BETWEEN).
- La columna se usa en JOINS (las claves foráneas son candidatas naturales para índices).
- La columna se usa en ORDER BY o GROUP BY con frecuencia.
- La columna debe tener valores únicos (usa un índice UNIQUE).
- La tabla es grande y las consultas de lectura son frecuentes.

NO crees índices (o piénsalo dos veces) cuando:

- La tabla es pequeña (unas cientos de filas). El table scan es tan rápido que el índice no aporta nada.
- La columna tiene muy pocos valores distintos (baja cardinalidad), como una columna "generó" con solo dos o tres valores, o un booleano "activo". El índice no filtra lo suficiente para valer la pena, aunque hay excepciones.
- La tabla tiene muchísimas escrituras (INSERT, UPDATE, DELETE constantes) y pocas lecturas. Cada escritura paga el costo de actualizar los índices.
- Ya existe un índice que cubre esa consulta (por ejemplo, un índice compuesto que empieza con esa columna).
- La columna casi nunca aparece en filtros, JOINS ni ordenamientos. Un índice que nadie usa solo ocupa espacio y hace lentas las escrituras.

Una trampa común: crear un índice por cada columna "por si acaso". He visto tablas con 15 índices donde solo se usan 3. Los otros 12 solo están ahí haciendo lenta cada inserción. Los índices se crean con propósito, no por superstición.

Ver y eliminar índices

Para ver los índices de una tabla en MySQL:

```
SHOW INDEX FROM clientes;
```

Esto te muestra cada índice, sus columnas y si es único. En PostgreSQL usarías `\d nombre_tabla` en la terminal, y en SQL Server hay vistas del sistema. Cada motor tiene su forma, pero la idea es la misma: inspecciona tus índices de vez en cuando.

Para eliminar un índice que ya no sirve o que creaste por error:

```
DROP INDEX idx_clientes_email ON clientes;
```

En MySQL el `DROP INDEX` incluye el nombre de la tabla. En PostgreSQL y SQL Server la sintaxis varía ligeramente (`DROP INDEX idx_clientes_email;`), así que revisa la documentación de tu motor. Eliminar un índice no borra datos, solo borra la estructura de búsqueda. Es una operación segura en ese sentido, aunque las consultas que lo usaban se volverán más lentas.

El costo oculto: índices y escrituras

Vale la pena entender bien este punto porque es la razón número uno por la que los índices se usan mal. Un índice no es gratis: es una estructura que hay que mantener. Cada `INSERT` tiene que agregar una entrada en cada índice de la tabla. Cada `UPDATE` que modifica una columna indexada tiene que actualizar el índice. Cada `DELETE` tiene que quitar entradas de todos los índices.

En una tabla de lectura intensiva (un catálogo de productos que se consulta miles de veces por segundo y se actualiza una vez al día), los índices son una maravilla: pagas el costo de mantenimiento una vez y disfrutas la velocidad miles de veces. En una tabla de escritura intensiva (un registro de eventos que recibe miles de inserts por segundo y casi nunca se consulta), cada índice extra es un impuesto que pagas en cada escritura.

La estrategia profesional es: indexa para tus consultas más frecuentes e importantes, mide el impacto, y no indexas "por si acaso". Y cuando una tabla cambie su patrón de uso (de escritura intensiva a lectura intensiva, por ejemplo), revisa tu estrategia de índices.

Vistas: tablas virtuales

Una vista es una consulta `SELECT` guardada con un nombre, que se comporta como una tabla virtual. No almacena datos propios (salvo las vistas materializadas, que veremos al final): cada vez que consultas una vista, la base de datos ejecuta la consulta guardada y te regresa el resultado fresco.

La sintaxis básica:

```
CREATE VIEW vista_ventas_mensuales AS
SELECT DATE_FORMAT(fecha, '%Y-%m') AS mes,
       COUNT(*) AS num_pedidos,
       SUM(monto) AS total
FROM pedidos
GROUP BY DATE_FORMAT(fecha, '%Y-%m');
```

A partir de ese momento, puedes consultar la vista como si fuera una tabla:

```
SELECT * FROM vista_ventas_mensuales WHERE mes = '2026-09';
```

La base de datos ejecuta la consulta guardada, aplica tu WHERE sobre el resultado, y te regresa las filas. Tú escribes una línea simple; la vista esconde toda la complejidad del GROUP BY.

Para qué sirven las vistas: simplificar, seguridad, reutilización

Las vistas tienen tres grandes usos, y entenderlos te va a cambiar la forma de diseñar bases de datos.

Primero, simplificación. Las consultas complejas con varios JOINS y agregaciones son difíciles de escribir y fáciles de romper. Si tu equipo necesita el mismo reporte complicado cada semana, guárdalo como vista una vez y todos la consultan con un SELECT simple. Menos código repetido significa menos lugares donde equivocarse.

Segundo, seguridad y control de acceso. Una vista puede exponer solo ciertas columnas o ciertas filas de una tabla. Ejemplo clásico: la tabla empleados tiene la columna salario, que no todos deben ver. Creas una vista sin esa columna y le das permiso a ciertos usuarios solo sobre la vista, no sobre la tabla completa:

```
CREATE VIEW vista_empleados_publica AS
SELECT id, nombre, departamento, fecha_ingreso
FROM empleados;
```

Los usuarios con acceso solo a la vista nunca verán los salarios, aunque estén en la misma tabla base. Es una forma sencilla y efectiva de implementar seguridad a nivel de datos.

Tercero, reutilización y estabilidad. Si la estructura de las tablas cambia (se renombra una columna, se divide una tabla en dos), puedes actualizar la vista para que siga regresando los

mismos nombres de columnas, y las aplicaciones que usan la vista no se rompen. La vista actúa como una capa de abstracción entre tus tablas físicas y quienes consumen los datos.

Vistas actualizables y sus límites

En algunos casos puedes hacer INSERT, UPDATE o DELETE a través de una vista, y los cambios se aplican a la tabla base. Pero no todas las vistas son actualizables. Las reglas generales son:

- La vista debe basarse en una sola tabla (sin JOINS).
- No debe tener GROUP BY, DISTINCT ni funciones de agregado.
- No debe tener subconsultas en el SELECT.
- Debe incluir las columnas obligatorias (NOT NULL sin valor por defecto) de la tabla base si quieres insertar.

Ejemplo de vista actualizable:

```
CREATE VIEW vista_clientes_activos AS
SELECT id, nombre, email, ciudad
FROM clientes
WHERE activo = 1;
```

Cómo es un simple SELECT con WHERE sobre una tabla, puedes hacer:

```
UPDATE vista_clientes_activos SET ciudad = 'Guadalajara' WHERE id = 5;
```

Y se actualizará la tabla clientes. En cambio, una vista con GROUP BY o con JOINS no es actualizable directamente: la base de datos no sabría a qué fila base aplicar el cambio.

Consejo práctico: aunque puedas actualizar a través de vistas, en aplicaciones reales es más claro hacer las escrituras directamente sobre las tablas y usar las vistas principalmente para lectura y reportes. Así evitas sorpresas.

CREATE OR REPLACE y DROP VIEW

Si necesitas modificar una vista, no tienes que borrarla y crearla de nuevo. Usa CREATE OR REPLACE VIEW:

```
CREATE OR REPLACE VIEW vista_ventas_mensuales AS
SELECT DATE_FORMAT(fecha, '%Y-%m') AS mes,
       COUNT(*) AS num_pedidos,
       SUM(monto) AS total,
       AVG(monto) AS ticket_promedio
FROM pedidos
GROUP BY DATE_FORMAT(fecha, '%Y-%m');
```

Esto reemplaza la definición manteniendo los permisos que ya habías otorgado sobre la vista. Si la borraras con `DROP VIEW` y la crearas de nuevo, perderías esos permisos y tendrías que reotorgarlos.

Para eliminar una vista que ya no se usa:

```
DROP VIEW vista_ventas_mensuales;
```

Borrar una vista no afecta los datos de las tablas base, solo elimina la consulta guardada. Pero cuidado: si hay aplicaciones o reportes que usan esa vista, se van a romper. Antes de borrar, verifica quién la está usando.

Cuidado con el rendimiento de las vistas

Las vistas son cómodas, pero tienen un costo que debes conocer. Cada vez que consultas una vista, la base de datos ejecuta la consulta completa que la define. Si la vista hace `JOINS` pesados sobre tablas enormes y tú solo necesitas dos columnas de una tabla pequeña, la base de datos igual hace todo el trabajo pesado.

El problema se agrava con vistas anidadas: una vista que consulta otra vista que consulta otra vista. Cada capa esconde complejidad, y cuando algo se vuelve lento es muy difícil saber dónde está el problema. He visto sistemas donde una "simple" consulta a una vista tardaba 30 segundos porque detrás había 5 vistas anidadas con `JOINS` cruzados.

Buenas prácticas con vistas y rendimiento:

- No anides vistas más de un nivel si puedes evitarlo. Si necesitas combinar vistas, considera escribir la consulta directa.
- Si una vista se consulta mucho y sus datos cambian poco, investiga las vistas materializadas (disponibles en PostgreSQL, Oracle y otros). Una vista materializada sí guarda el resultado físicamente y se actualiza periódicamente: es rapidísima de consultar, a cambio de que los datos pueden estar ligeramente desactualizados.

- Revisa el plan de ejecución (EXPLAIN) de tus consultas importantes que usan vistas, para asegurarte de que el motor está optimizando bien.
- Las vistas son para simplificar y proteger, no para esconder consultas monstruosas. Si la vista es un monstruo, el problema no lo resuelve la vista: hay que optimizar la consulta base.

Patrones de subconsultas que verás en el trabajo real

Para cerrar el tema de subconsultas, veamos tres patrones que aparecen una y otra vez en sistemas reales. Reconocerlos te va a ahorrar mucho tiempo cuando leas código ajeno o diseñes tus propias consultas.

Patrón 1: "los N más altos por grupo" con subconsulta correlacionada. Quieres los 3 productos más caros de cada categoría. Una forma es comparar cuántos productos de la misma categoría son más caros:

```
SELECT nombre, categoria, precio
FROM productos p1
WHERE (SELECT COUNT(*)
       FROM productos p2
       WHERE p2.categoria = p1.categoria
       AND p2.precio > p1.precio) < 3
ORDER BY categoria, precio DESC;
```

La subconsulta cuenta cuántos productos de la misma categoría cuestan más que el actual. Si son menos de 3, el producto está entre los 3 más caros de su categoría. Es un patrón elegante aunque costoso en tablas enormes; en motores modernos también se resuelve con funciones de ventana (tema de unidades posteriores).

Patrón 2: filtrar con agregados de otra tabla. Quieres los clientes cuyo total de pedidos supere el total promedio de todos los clientes. Son dos niveles de agregación: primero totales por cliente, luego el promedio de esos totales:

```
SELECT c.nombre, SUM(p.monto) AS total
FROM clientes c
JOIN pedidos p ON c.id = p.cliente_id
GROUP BY c.nombre
HAVING SUM(p.monto) > (SELECT AVG(total_cliente)
                      FROM (SELECT SUM(monto) AS total_cliente
                            FROM pedidos
                            GROUP BY cliente_id) AS totales);
```


La subconsulta interior calcula el total por cliente, la del medio promedia esos totales, y el HAVING filtra. Son subconsultas anidadas a dos niveles: perfectamente válidas, solo cuida la legibilidad con buenos alias y formato.

Patrón 3: usar subconsulta para evitar el problema del WHERE con alias. Ya lo vimos con la tabla derivada y el precio con IVA. Este patrón aparece cada vez que calculas algo en el SELECT y necesitas filtrar por ese cálculo. Alternativas: repetir la expresión en el WHERE (feo y propenso a errores si la cambias en un lado y no en el otro) o envolver en tabla derivada (limpio). La tabla derivada casi siempre gana.

Cómo piensa el optimizador: EXPLAIN

Cuando escribes una consulta, tú dices QUÉ quieres, no CÓMO obtenerlo. El optimizador de consultas es el componente del motor que decide el cómo: en qué orden leer las tablas, si usa un índice o hace un table scan, si ejecuta primero la subconsulta o la convierte en JOIN. Casi siempre decide bien, pero no es infalible, y cuando una consulta es lenta necesitas ver qué decidió.

Para eso existe EXPLAIN. Antepones EXPLAIN a tu consulta y el motor te muestra su plan de ejecución sin correr la consulta:

```
EXPLAIN SELECT nombre FROM clientes WHERE email = 'ana@ejemplo.com';
```

La salida varía por motor, pero las columnas importantes son parecidas: type (cómo accede a la tabla: ALL significa table scan, ref o range significan que usa un índice), key (qué índice eligió, o NULL si no usó ninguno), rows (cuántas filas estima que va a revisar) y Extra (información adicional como "Using where" o "Using filesort").

Aprende a leer el EXPLAIN de tus consultas más importantes. Si ves type = ALL en una tabla de millones de filas, probablemente te falta un índice. Si ves que el optimizador eligió un índice distinto al que esperabas, pregúntate por qué: a veces tiene razón y a veces necesita ayuda (por ejemplo, reescribiendo la consulta o actualizando las estadísticas de la tabla con ANALYZE TABLE).

Un detalle importante: el optimizador a veces reescribe tus subconsultas internamente. Muchos motores convierten automáticamente una subconsulta con IN en un JOIN semi, o una tabla derivada la fusionan con la consulta exterior. Esto significa que la forma en que escribes la consulta no siempre es la forma en que se ejecuta. Por eso la regla de oro del rendimiento es medir, no adivinar: EXPLAIN te dice la verdad.

Vistas materializadas: cuando la vista sí guarda datos

Las vistas normales que vimos son virtuales: cada consulta ejecuta la definición desde cero. Pero hay casos donde eso es demasiado lento: un reporte que agrega millones de filas y que cien personas consultan al día. Para esos casos existen las vistas materializadas, disponibles en PostgreSQL, Oracle, SQL Server y otros motores (MySQL no las tiene de forma nativa hasta la fecha).

Una vista materializada sí guarda el resultado físicamente en disco, como una tabla. Consultarla es rapidísimo porque los datos ya están calculados. El truco es que los datos pueden estar desactualizados: la vista se refresca periódicamente (cada hora, cada noche) o manualmente.

```
CREATE MATERIALIZED VIEW reporte_ventas_diario AS
SELECT DATE(fecha) AS dia, COUNT(*) AS pedidos, SUM(monto) AS total
FROM pedidos
GROUP BY DATE(fecha);
```

Y para actualizarla:

```
REFRESH MATERIALIZED VIEW reporte_ventas_diario;
```

Cuándo usarlas: reportes pesados sobre datos que no necesitan estar al segundo, dashboards que mucha gente consulta, agregaciones sobre tablas enormes. Cuándo NO usarlas: cuando necesitas datos en tiempo real, o cuando la tabla base cambia constantemente y el refresco sería más caro que la consulta directa. Es el clásico trueque entre velocidad y frescura de datos.

Errores comunes en consultas avanzadas

Vamos a repasar los errores más frecuentes que cometen los estudiantes (y varios profesionales) con estos tres temas. Si te suena familiar alguno, no te preocupes: es normal. Lo importante es reconocerlos.

Error 1: subconsulta que regresa más de una fila donde se esperaba un solo valor. Usar = o > con una subconsulta que regresa varias filas marca error. Solución: usa IN, o agrega una función de agregado, o un LIMIT 1 si de verdad solo quieres un valor.

Error 2: olvidar el alias en la subconsulta del FROM. Casi todos los motores lo exigen. Si ves un error de sintaxis cerca del paréntesis de cierre de tu subconsulta en el FROM, lo más probable es que te falte el alias.

Error 3: la trampa del NOT IN con NULLs. Si la subconsulta puede regresar NULL, NOT IN te regresa cero filas y te vuelves loco buscando el error. Usa NOT EXISTS o filtra los NULLs.

Error 4: correlacionar mal la subconsulta. En una subconsulta correlacionada, asegúrate de que la condición realmente conecte la tabla interior con la exterior (p.cliente_id = c.id). Si olvidas esa condición, la subconsulta se evalúa igual para todas las filas y el resultado no tiene sentido.

Error 5: crear índices en todas las columnas "por si acaso". Ya lo vimos: cada índice cuesta en las escrituras. Indexa con propósito.

Error 6: crear un índice compuesto en el orden equivocado. Recuerda que el índice se usa de izquierda a derecha. Analiza tus consultas más frecuentes antes de decidir el orden de las columnas.

Error 7: pensar que el índice se usa automáticamente en todos los casos. Hay situaciones donde el optimizador ignora el índice: por ejemplo, si la consulta regresa un porcentaje alto de la tabla (ahí el table scan es más eficiente), o si aplicas una función a la columna indexada en el WHERE (WHERE YEAR(fecha) = 2026 no usa bien un índice sobre fecha en la mayoría de los motores; mejor usa un rango: WHERE fecha >= '2026-01-01' AND fecha < '2027-01-01').

Error 8: modificar una vista con DROP y CREATE en lugar de CREATE OR REPLACE, perdiendo los permisos otorgados.

Error 9: anidar vistas dentro de vistas sin control, creando consultas lentas e imposibles de depurar.

Error 10: usar SELECT * dentro de la definición de una vista. Si después agregas columnas a la tabla base, la vista puede comportarse de formas inesperadas según el motor. Mejor lista explícitamente las columnas que necesita la vista.

Buenas prácticas

Para cerrar la teoría, un resumen de buenas prácticas que usan los profesionales:

- Prefiere EXISTS sobre IN cuando solo verificas existencia, especialmente con NOT EXISTS sobre NOT IN por el tema de los NULLs.
- Dale alias claros a tus tablas en subconsultas correlacionadas (c, p, e). Hace el código mucho más legible.
- En subconsultas del FROM, el alias es obligatorio: hazlo un hábito y nunca se te olvidará.

- Antes de crear un índice, identifica tus consultas más lentas o más frecuentes. Indexa para ellas, no para hipótesis.
- Usa la convención de nombres `idx_tabla_columna` para tus índices. Tu yo del futuro te lo agradecerá.
- Revisa periódicamente qué índices se usan y cuáles no. Los índices sin uso se eliminan.
- Usa vistas para simplificar consultas repetidas y para controlar acceso a datos sensibles.
- Lista columnas explícitas en las vistas, nunca `SELECT *`.
- Usa `CREATE OR REPLACE VIEW` para modificar vistas sin perder permisos.
- Evita anidar vistas más de un nivel. Si la consulta es compleja, escríbela directa y optimízala.
- Mide antes de optimizar. Usa `EXPLAIN` para ver si tu índice realmente se está usando y si tu consulta hace lo que crees.

Ejemplos paso a paso

Vamos a trabajar con una base de datos de ejemplo de una tienda en línea. Primero la creamos para que puedas seguir cada ejemplo en tu computadora. Si ya tienes tus propias tablas, adapta los ejemplos.

```
CREATE DATABASE tienda_ejemplo;
USE tienda_ejemplo;

CREATE TABLE clientes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    email VARCHAR(100),
    ciudad VARCHAR(50),
    activo TINYINT(1) DEFAULT 1
);

CREATE TABLE productos (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    categoria VARCHAR(50),
    precio DECIMAL(10,2) NOT NULL,
    stock INT DEFAULT 0
);

CREATE TABLE pedidos (
```

```

    id INT PRIMARY KEY AUTO_INCREMENT,
    cliente_id INT,
    fecha DATE,
    monto DECIMAL(10,2),
    FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);

INSERT INTO clientes (nombre, email, ciudad) VALUES
('Ana Torres', 'ana@ejemplo.com', 'Ciudad de Mexico'),
('Bruno Diaz', 'bruno@ejemplo.com', 'Guadalajara'),
('Carla Ruiz', 'carla@ejemplo.com', 'Monterrey'),
('Diego Lopez', 'diego@ejemplo.com', 'Ciudad de Mexico'),
('Elena Mora', 'elena@ejemplo.com', 'Puebla');

INSERT INTO productos (nombre, categoria, precio, stock) VALUES
('Laptop Pro 15', 'Electronica', 18999.00, 12),
('Mouse inalambrico', 'Electronica', 349.00, 150),
('Teclado mecanico', 'Electronica', 1299.00, 60),
('Silla ergonomica', 'Muebles', 4599.00, 25),
('Escritorio roble', 'Muebles', 7999.00, 10),
('Lampara LED', 'Muebles', 599.00, 80),
('Curso de SQL', 'Libros', 499.00, 200),
('Curso de Python', 'Libros', 549.00, 180);

INSERT INTO pedidos (cliente_id, fecha, monto) VALUES
(1, '2026-08-02', 19348.00),
(1, '2026-08-15', 1299.00),
(2, '2026-08-10', 4599.00),
(3, '2026-09-01', 1048.00),
(3, '2026-09-05', 7999.00);

```

Con estos datos listos, vamos ejemplo por ejemplo.

Ejemplo 1: productos más caros que el promedio (subconsulta escalar)

Pregunta: ¿qué productos cuestan más que el precio promedio de todos los productos?

Primero piensa en los pasos: necesitas el promedio de todos los precios, y después comparar cada producto contra ese número. La subconsulta calcula el promedio:

```

SELECT nombre, precio
FROM productos
WHERE precio > (SELECT AVG(precio) FROM productos);

```

Cómo se ejecuta: primero corre la subconsulta (SELECT AVG(precio) FROM productos), que con nuestros datos da aproximadamente 5161.50. Después, la consulta exterior revisa cada producto: ¿su precio es mayor que 5161.50? La Laptop Pro 15 (18999) y el Escritorio roble (7999) pasan el filtro. El resultado muestra esos dos productos con sus precios.

Variante útil: si también quieres ver el promedio junto a cada producto para comparar visualmente, agrega la subconsulta en el SELECT:

```
SELECT nombre, precio,  
       (SELECT AVG(precio) FROM productos) AS precio_promedio,  
       precio - (SELECT AVG(precio) FROM productos) AS diferencia  
FROM productos  
WHERE precio > (SELECT AVG(precio) FROM productos);
```

La columna diferencia te dice cuánto más caro es cada producto respecto al promedio. En reportes reales esto es muy común: no solo filtras, también muestras el contexto.

Ejemplo 2: IN con subconsulta (clientes por ciudad de compra)

Pregunta: ¿qué clientes han comprado productos de la categoría 'Electrónica'? Supongamos que agregamos una tabla detalle_pedido para este ejemplo, o más simple: ¿qué clientes hicieron pedidos con monto mayor a 5000?

```
SELECT nombre, email  
FROM clientes  
WHERE id IN (SELECT cliente_id FROM pedidos WHERE monto > 5000);
```

La subconsulta genera la lista de IDs de clientes con pedidos grandes: con nuestros datos, el cliente 1 (pedido de 19348) y el cliente 3 (pedido de 7999). El IN filtra los clientes cuya ID está en esa lista: Ana Torres y Carla Ruiz.

Nota que si un cliente tiene dos pedidos grandes, su ID aparece dos veces en la lista de la subconsulta, pero el IN lo maneja sin problema: solo verifica pertenencia, no cuenta duplicados. Esa es una ventaja del IN sobre el JOIN para este tipo de preguntas: no necesitas DISTINCT.

La versión negativa, clientes que nunca han hecho un pedido grande:

```
SELECT nombre, email
FROM clientes
WHERE id NOT IN (SELECT cliente_id FROM pedidos WHERE monto > 5000 AND cliente_id
IS NOT NULL);
```

Agregamos el filtro IS NOT NULL por la trampa del NULL que vimos en la teoría. En este ejemplo no hay NULLs, pero es un buen hábito.

Ejemplo 3: EXISTS para clientes con pedidos (subconsulta correlacionada)

Pregunta: lista los clientes que tienen al menos un pedido, mostrando cuántos pedidos tiene cada uno. Primero la versión con EXISTS:

```
SELECT c.nombre, c.email
FROM clientes c
WHERE EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Por cada cliente, la base de datos busca al menos un pedido con su ID. Ana (id 1) tiene 2 pedidos, Bruno (id 2) tiene 1, Carla (id 3) tiene 2. Diego (id 4) y Elena (id 5) no tienen pedidos, así que no aparecen. En cuanto encuentra el primer pedido de cada cliente, deja de buscar: eficiente.

Ahora, si además quieres mostrar el número de pedidos, combinas EXISTS con una subconsulta escalar en el SELECT:

```
SELECT c.nombre, c.email,
       (SELECT COUNT(*) FROM pedidos p WHERE p.cliente_id = c.id) AS num_pedidos
FROM clientes c
WHERE EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

La subconsulta del SELECT cuenta los pedidos de cada cliente, y el EXISTS filtra solo a los que tienen al menos uno. El resultado: Ana con 2, Bruno con 1, Carla con 2.

Y la versión con NOT EXISTS, clientes sin ningún pedido (los candidatos para una campaña de reactivación):

```
SELECT c.nombre, c.email
FROM clientes c
WHERE NOT EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Regresa a Diego Lopez y Elena Mora. Esta consulta no tiene la trampa del NULL: funciona bien siempre.

Ejemplo 4: tabla derivada (resumen de ventas por cliente)

Pregunta: ¿qué clientes han gastado más de 8000 en total? Necesitas primero sumar por cliente y después filtrar. La tabla derivada lo resuelve en una sola consulta:

```
SELECT cliente, total_gastado
FROM (SELECT c.nombre AS cliente, SUM(p.monto) AS total_gastado
      FROM clientes c
      JOIN pedidos p ON c.id = p.cliente_id
      GROUP BY c.nombre) AS resumen_ventas
WHERE total_gastado > 8000;
```

Paso a paso: la subconsulta interior hace el JOIN entre clientes y pedidos, agrupa por nombre y suma los montos. Con nuestros datos: Ana Torres 20647, Bruno Díaz 4599, Carla Ruiz 9047. Ese resultado se convierte en la tabla temporal resumen_ventas. La consulta exterior filtra donde total_gastado > 8000: Ana Torres y Carla Ruiz.

Fíjate en el alias obligatorio AS resumen_ventas después del paréntesis de cierre. Sin él, la consulta marca error de sintaxis. También nota que las columnas de la tabla derivada (cliente, total_gastado) son los alias definidos adentro: puedes usarlos en el SELECT y WHERE exterior como si fueran columnas normales.

Ejemplo 5: crear y usar índices

Vamos a crear índices sobre nuestra base de ejemplo y comprobar su efecto. Primero, un índice simple sobre el email de clientes, porque las búsquedas por email son muy comunes (login, recuperación de cuenta):

```
CREATE INDEX idx_clientes_email ON clientes(email);
```

Ahora las consultas como esta usarán el índice automáticamente:

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

Para confirmar que el motor usa el índice, pide el plan de ejecución:

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


En la salida del EXPLAIN busca la columna "key": si dice idx_clientes_email, el índice se está usando. Si dice NULL, está haciendo un table scan. Con tablas pequeñas de ejemplo a veces el optimizador prefiere el table scan porque es igual de rápido; con miles de filas verás la diferencia.

Ahora un índice compuesto para una consulta típica de reportes: pedidos de un cliente en un rango de fechas:

```
CREATE INDEX idx_pedidos_cliente_fecha ON pedidos(cliente_id, fecha);
```

Esta consulta lo aprovecha al máximo:

```
SELECT * FROM pedidos  
WHERE cliente_id = 1 AND fecha BETWEEN '2026-08-01' AND '2026-08-31';
```

Filtra por la primera columna del índice (cliente_id) y luego por la segunda (fecha): uso perfecto del índice compuesto. En cambió, esta consulta no lo aprovecha bien:

```
SELECT * FROM pedidos WHERE fecha BETWEEN '2026-08-01' AND '2026-08-31';
```

Como no filtra por cliente_id (la primera columna del índice), el motor no puede usar el índice de forma eficiente. Si las consultas por fecha sola son frecuentes, necesitarías otro índice que empiece con fecha.

Para ver todos los índices de una tabla:

```
SHOW INDEX FROM pedidos;
```

Verás el índice de la llave primaria (que se crea automáticamente) y tu idx_pedidos_cliente_fecha. Y si un índice deja de ser útil, lo eliminas:

```
DROP INDEX idx_clientes_email ON clientes;
```

Recuerda: borrar un índice no borra datos, solo quita la estructura de búsqueda.

Ejemplo 6: crear vistas de reportes y consultarlas

Vamos a crear dos vistas útiles para nuestra tienda: un reporte de ventas por categoría y una vista de clientes con su total gastado.

Vista 1: resumen de productos por categoría:

```
CREATE VIEW vista_resumen_categorias AS
SELECT categoria,
       COUNT(*) AS num_productos,
       AVG(precio) AS precio_promedio,
       SUM(stock) AS unidades_en_stock
FROM productos
GROUP BY categoria;
```

Ahora el reporte mensual es trivial:

```
SELECT * FROM vista_resumen_categorias;
```

Y puedes filtrar sobre la vista como sobre cualquier tabla:

```
SELECT * FROM vista_resumen_categorias WHERE precio_promedio > 1000;
```

Vista 2: clientes con su total gastado y número de pedidos (reutiliza la lógica del ejemplo 4):

```
CREATE VIEW vista_clientes_resumen AS
SELECT c.id, c.nombre, c.email, c.ciudad,
       COUNT(p.id) AS num_pedidos,
       COALESCE(SUM(p.monto), 0) AS total_gastado
FROM clientes c
LEFT JOIN pedidos p ON c.id = p.cliente_id
GROUP BY c.id, c.nombre, c.email, c.ciudad;
```

Usamos LEFT JOIN para que aparezcan también los clientes sin pedidos (con total_gastado en 0 gracias al COALESCE). Ahora varias preguntas se vuelven simples:

Clientes que más han gastado:

```
SELECT nombre, total_gastado
FROM vista_clientes_resumen
ORDER BY total_gastado DESC;
```

Clientes sin compras (para campaña de reactivación):

```
SELECT nombre, email
FROM vista_clientes_resumen
WHERE num_pedidos = 0;
```

Si después necesitas agregar una columna al reporte, por ejemplo el ticket promedio, usas CREATE OR REPLACE:

```
CREATE OR REPLACE VIEW vista_clientes_resumen AS
SELECT c.id, c.nombre, c.email, c.ciudad,
       COUNT(p.id) AS num_pedidos,
       COALESCE(SUM(p.monto), 0) AS total_gastado,
       COALESCE(AVG(p.monto), 0) AS ticket_promedio
FROM clientes c
LEFT JOIN pedidos p ON c.id = p.cliente_id
GROUP BY c.id, c.nombre, c.email, c.ciudad;
```

Todas las consultas que usan la vista siguen funcionando y ahora tienen la columna extra disponible. Así de fácil es evolucionar un reporte sin tocar el código de la aplicación.

Ejemplo 7: vista para seguridad (ocultar datos sensibles)

Supongamos que agregamos una columna de teléfono a clientes y que el equipo de marketing puede ver nombres y ciudades pero no emails ni teléfonos. Creamos una vista limitada:

```
ALTER TABLE clientes ADD COLUMN telefono VARCHAR(20);

CREATE VIEW vista_marketing_clientes AS
SELECT id, nombre, ciudad
FROM clientes
WHERE activo = 1;
```

Al equipo de marketing le das permiso solo sobre vista_marketing_clientes, no sobre la tabla clientes. Ellos pueden hacer:

```
SELECT * FROM vista_marketing_clientes WHERE ciudad = 'Guadalajara';
```

Pero no pueden ver emails ni teléfonos porque la vista simplemente no los incluye. Es seguridad simple y efectiva: los datos sensibles nunca salen de la tabla base.

Ejemplo 8: reescribir una subconsulta correlacionada con JOIN

Las subconsultas correlacionadas son elegantes pero a veces lentas, porque se evalúan una vez por cada fila exterior. Muchas se pueden reescribir con JOIN y GROUP BY, que el optimizador suele ejecutar más rápido en tablas grandes. Veamos la transformación paso a paso con el ejemplo de empleados que ganan más que el promedio de su departamento.

Versión con subconsulta correlacionada:

```
SELECT nombre, departamento, salario
FROM empleados e
WHERE salario > (SELECT AVG(salario)
                  FROM empleados
                  WHERE departamento = e.departamento);
```

Versión con JOIN: primero calculas los promedios por departamento en una tabla derivada, y después haces JOIN contra ella:

```
SELECT e.nombre, e.departamento, e.salario
FROM empleados e
JOIN (SELECT departamento, AVG(salario) AS promedio
      FROM empleados
      GROUP BY departamento) AS prom
ON e.departamento = prom.departamento
WHERE e.salario > prom.promedio;
```

Paso a paso: la tabla derivada prom calcula el promedio una sola vez por departamento (una sola pasada por la tabla con GROUP BY). Después el JOIN pega ese promedio a cada empleado según su departamento, y el WHERE filtra. El resultado es idéntico, pero la tabla se recorre una vez para los promedios en lugar de una vez por empleado.

¿Cuándo vale la pena reescribir? Cuando la tabla exterior es grande y la subconsulta correlacionada se vuelve el cuello de botella (lo verás en el EXPLAIN o en los tiempos de ejecución). Con tablas pequeñas, quédate con la versión correlacionada: se lee mejor y la diferencia de velocidad es imperceptible. La legibilidad también es rendimiento: del programador que tendrá que mantener ese código.

Práctica

Usa la base de datos tienda_ejemplo de la sección anterior. Si la borraste, vuelve a crearla con el script del inicio de los ejemplos. Cada ejercicio tiene su solución explicada paso a paso, pero intenta resolverlo tú primero antes de ver la solución. La práctica es donde realmente se aprende.

Ejercicio 1: productos por encima del promedio de su categoría

Escribe una consulta que muestre los productos cuyo precio sea mayor que el precio promedio de SU categoría (no el promedio general). Muestra nombre, categoría, precio y el promedio de su categoría.

Piénsalo: necesitas comparar cada producto contra el promedio calculado solo para su categoría. Eso suena a subconsulta correlacionada.

Solución:

```
SELECT nombre, categoria, precio,
       (SELECT AVG(precio) FROM productos p2 WHERE p2.categoria = p1.categoria) AS
promedio_categoria
FROM productos p1
WHERE precio > (SELECT AVG(precio) FROM productos p2 WHERE p2.categoria =
p1.categoria);
```

Paso a paso: la subconsulta (SELECT AVG(precio) FROM productos p2 WHERE p2.categoría = p1.categoría) es correlacionada porque usa p1.categoría de la consulta exterior. Para cada producto, calcula el promedio solo de los productos de su misma categoría. El WHERE exterior filtra los que superan ese promedio. La subconsulta del SELECT muestra el promedio como columna de contexto. Con nuestros datos: en Electrónica el promedio es $(18999 + 349 + 1299) / 3 = 6882.33$, así que solo la Laptop Pro 15 lo supera. En Muebles el promedio es $(4599 + 7999 + 599) / 3 = 4399$, así que solo el Escritorio roble lo supera. En Libros el promedio es 524, así que el Curso de Python (549) lo supera.

Ejercicio 2: clientes sin pedidos usando NOT EXISTS

Escribe una consulta que liste nombre y ciudad de los clientes que no tienen ningún pedido registrado. Debe funcionar correctamente aunque la tabla pedidos tuviera valores NULL en cliente_id.

Solución:

```
SELECT c.nombre, c.ciudad
FROM clientes c
WHERE NOT EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id);
```

Paso a paso: por cada cliente, la subconsulta busca al menos un pedido con su ID. Si no encuentra ninguno, NOT EXISTS es verdadero y el cliente pasa el filtro. Elegimos NOT EXISTS en lugar de NOT IN precisamente por el requisito del ejercicio: NOT IN fallaría

silenciosamente si hubiera un NULL en cliente_id, mientras que NOT EXISTS funciona bien siempre. Con nuestros datos regresa a Diego Lopez (Ciudad de Mexico) y Elena Mora (Puebla).

Ejercicio 3: tabla derivada con doble filtro

Usando una tabla derivada, muestra el nombre y el total gastado de los clientes cuyo total gastado esté entre 4000 y 10000. Ordena de mayor a menor total.

Solución:

```
SELECT cliente, total_gastado
FROM (SELECT c.nombre AS cliente, SUM(p.monto) AS total_gastado
      FROM clientes c
      JOIN pedidos p ON c.id = p.cliente_id
      GROUP BY c.nombre) AS resumen
WHERE total_gastado BETWEEN 4000 AND 10000
ORDER BY total_gastado DESC;
```

Paso a paso: la subconsulta interior calcula el total por cliente con JOIN y GROUP BY (Ana 20647, Bruno 4599, Carla 9047). La tabla derivada se llama resumen (alias obligatorio). La consulta exterior filtra con BETWEEN 4000 AND 10000: Bruno Diaz (4599) y Carla Ruiz (9047) pasan; Ana queda fuera por pasarse del rango. El ORDER BY los ordena: primero Carla, luego Bruno. Nota que el ORDER BY va en la consulta exterior, no dentro de la subconsulta.

Ejercicio 4: índices para una consulta de reportes

La siguiente consulta se ejecuta cientos de veces al día en una tabla pedidos con 2 millones de filas, y tarda demasiado:

```
SELECT cliente_id, SUM(monto) AS total
FROM pedidos
WHERE fecha >= '2026-01-01' AND fecha < '2027-01-01'
GROUP BY cliente_id;
```

Propón el índice adecuado, créalo y explica por qué elegiste ese orden de columnas. Además, menciona un caso en el que NO convendría crear índices en esta tabla.

Solución:

```
CREATE INDEX idx_pedidos_fecha_cliente ON pedidos(fecha, cliente_id);
```

Paso a paso: la consulta filtra por un rango de fecha y agrupa por cliente_id. El índice compuesto (fecha, cliente_id) permite al motor primero localizar rápidamente las filas del rango de fechas usando la primera columna, y la segunda columna ya viene ordenada por cliente_id, lo que ayuda al GROUP BY. El orden es fecha primero porque el filtro (WHERE) es por fecha; la regla es poner primero la columna del filtro más selectivo o más frecuente. Si pusiéramos (cliente_id, fecha), el índice no ayudaría con el filtro por rango de fecha porque la primera columna no está en el WHERE.

Cuándo NO convendría indexar: si la tabla pedidos recibe miles de INSERTs por segundo (por ejemplo, un sistema de punto de venta en hora pico) y esta consulta de reporte se corre pocas veces al día, el costo de mantener el índice en cada inserción podría superar el beneficio. En ese caso sería mejor generar el reporte en horarios de baja actividad o usar una tabla de resumen precalculada.

Ejercicio 5: vista de reporte mensual

Crea una vista llamada vista_reporte_mensual que muestre, por cada mes, el número de pedidos, el monto total y el monto promedio. Luego escribe una consulta sobre la vista que muestre solo los meses con monto total mayor a 10000. Finalmente, modifica la vista con CREATE OR REPLACE para agregar la columna del monto máximo del mes.

Solución, parte 1 (crear la vista):

```
CREATE VIEW vista_reporte_mensual AS
SELECT DATE_FORMAT(fecha, '%Y-%m') AS mes,
       COUNT(*) AS num_pedidos,
       SUM(monto) AS monto_total,
       AVG(monto) AS monto_promedio
FROM pedidos
GROUP BY DATE_FORMAT(fecha, '%Y-%m');
```

Paso a paso: DATE_FORMAT(fecha, '%Y-%m') convierte cada fecha en un texto tipo '2026-08'. Agrupamos por ese valor y calculamos las tres métricas. La vista guarda esta lógica con un nombre.

Solución, parte 2 (consultar la vista):

```
SELECT mes, num_pedidos, monto_total, monto_promedio
FROM vista_reporte_mensual
WHERE monto_total > 10000;
```

La vista se consulta como una tabla normal. Con nuestros datos: agosto 2026 tiene 3 pedidos por 25246 en total, septiembre 2026 tiene 2 pedidos por 9047. Solo agosto pasa el filtro de 10000.

Solución, parte 3 (modificar la vista):

```
CREATE OR REPLACE VIEW vista_reporte_mensual AS
SELECT DATE_FORMAT(fecha, '%Y-%m') AS mes,
       COUNT(*) AS num_pedidos,
       SUM(monto) AS monto_total,
       AVG(monto) AS monto_promedio,
       MAX(monto) AS monto_maximo
FROM pedidos
GROUP BY DATE_FORMAT(fecha, '%Y-%m');
```

Usamos CREATE OR REPLACE en lugar de DROP + CREATE para no perder los permisos otorgados sobre la vista. La consulta de la parte 2 sigue funcionando igual, y ahora también puedes pedir la columna monto_maximo.

Ejercicio 6: reescribir IN con EXISTS (y explicar la diferencia)

Toma esta consulta que usa IN:

```
SELECT nombre FROM clientes
WHERE id IN (SELECT cliente_id FROM pedidos WHERE monto > 1000);
```

Reescribela usando EXISTS. Explica en qué caso la versión con EXISTS sería más rápida y por qué.

Solución:

```
SELECT c.nombre
FROM clientes c
WHERE EXISTS (SELECT 1 FROM pedidos p WHERE p.cliente_id = c.id AND p.monto > 1000);
```

Paso a paso: la subconsulta correlacionada verifica, para cada cliente, si existe al menos un pedido suyo con monto mayor a 1000. En cuanto encuentra el primero, se detiene y pasa al siguiente cliente.

Cuándo EXISTS es más rápido: cuando la tabla pedidos es muy grande y la mayoría de los clientes tienen al menos un pedido que cumple la condición. El IN tiene que construir la lista

completa de cliente_id (recorriendo todos los pedidos que cumplen el filtro, incluyendo duplicados) antes de comparar. El EXISTS se detiene en la primera coincidencia por cliente, así que hace mucho menos trabajo. Además, EXISTS nunca tiene la trampa del NULL que tiene NOT IN en su versión negativa. Como regla práctica: si solo verificas existencia, EXISTS expresa mejor tu intención y suele rendir mejor.

Ejercicio 7: detective de índices con EXPLAIN

Tienes la tabla pedidos con el índice idx_pedidos_cliente_fecha sobre (cliente_id, fecha) que creamos en el ejemplo 5. Para cada una de estas dos consultas, predice si el índice se usará de forma eficiente o no, explica por qué, y di qué harías en cada caso:

Consulta A:

```
SELECT * FROM pedidos WHERE cliente_id = 2;
```

Consulta B:

```
SELECT * FROM pedidos WHERE YEAR(fecha) = 2026;
```

Solución: la consulta A SÍ usa el índice eficientemente. Filtra por cliente_id, que es la primera columna del índice compuesto, así que el motor salta directo a las filas de ese cliente. Si corres EXPLAIN verás type = ref y key = idx_pedidos_cliente_fecha. No hay nada que arreglar.

La consulta B NO usa el índice de forma eficiente, por dos razones. Primera: filtra por fecha pero el índice empieza con cliente_id, y recuerda que el índice compuesto se usa de izquierda a derecha; sin filtrar por la primera columna, el motor no puede aprovecharlo bien. Segunda y más grave: aplica la función YEAR() sobre la columna fecha, lo que impide usar cualquier índice sobre fecha porque el motor tendría que calcular la función para cada fila antes de comparar. Es el error 7 de la sección de errores comunes.

Cómo arreglar la consulta B: reescribe el filtro como un rango sobre la columna sin funciones:

```
SELECT * FROM pedidos WHERE fecha >= '2026-01-01' AND fecha < '2027-01-01';
```

Y si las consultas por fecha sola son frecuentes en tu aplicación, crea un índice dedicado que empiece con fecha:

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

Moraleja del ejercicio: un índice solo sirve si la consulta está escrita de forma que el optimizador pueda usarlo. La forma más rápida de comprobarlo es EXPLAIN, no la intuición.

Resumen de la unidad

En esta unidad subiste de nivel: pasaste de escribir consultas básicas a pensar como consultor de datos. Estos son los puntos clave para repasar:

- Una subconsulta es un SELECT dentro de otro SELECT, siempre entre paréntesis. La interior se ejecuta primero y su resultado lo usa la exterior.
- Las subconsultas escalares regresan un solo valor y se usan con operadores de comparación (=, >, <). Si regresan más de una fila, marcan error.
- Las subconsultas de una columna con varias filas se usan con IN y NOT IN. Cuidado con NOT IN y los NULLs: si hay un NULL en la lista, no regresa nada. NOT EXISTS es la alternativa segura.
- Las subconsultas en el FROM crean tablas derivadas: resultados temporales que se consultan como tablas. El alias es obligatorio siempre.
- Las subconsultas correlacionadas referencian a la consulta exterior y se evalúan una vez por fila. EXISTS verifica si hay al menos una fila coincidente y se detiene en la primera: rápido y seguro.
- Subconsulta vs JOIN: usa JOIN cuando necesites columnas de ambas tablas; usa EXISTS o subconsulta cuando solo filtres o verifiques existencia. EXISTS no duplica filas como sí puede hacerlo un JOIN.
- Un índice es una estructura que permite encontrar filas sin revisar toda la tabla, como el índice al final de un libro. Se crea con CREATE INDEX y el optimizador lo usa automáticamente.
- Los índices compuestos indexan varias columnas en un orden específico y se usan de izquierda a derecha. Pon primero la columna por la que más filtras.
- Crea índices en columnas de WHERE, JOIN, ORDER BY y GROUP BY frecuentes, y en columnas que deben ser únicas (índice UNIQUE). No indexes tablas pequeñas, columnas con pocos valores distintos ni "por si acaso".
- Cada índice tiene un costo: cada INSERT, UPDATE y DELETE debe actualizarlo. En tablas de escritura intensiva, pocos índices; en tablas de lectura intensiva, indexa con confianza.

- Inspecciona tus índices con `SHOW INDEX` y elimina los que no se usen con `DROP INDEX`. Borrar un índice no borra datos.
- Una vista es una consulta guardada con un nombre que se comporta como tabla virtual. Se crea con `CREATE VIEW` y se consulta con `SELECT` normal.
- Las vistas sirven para simplificar consultas complejas, controlar el acceso a datos sensibles y reutilizar lógica de reportes.
- Solo las vistas simples (una tabla, sin `GROUP BY` ni agregados) son actualizables. Para modificar una vista usa `CREATE OR REPLACE VIEW` y así conservas los permisos.
- Cuidado con el rendimiento: cada consulta a una vista ejecuta su consulta completa. No anides vistas dentro de vistas y revisa el plan de ejecución de las consultas importantes.
- Usa `EXPLAIN` para ver el plan de ejecución: qué índice usa cada tabla y cuántas filas estima revisar. Mide antes de optimizar.
- Las vistas materializadas guardan el resultado físicamente y son rapidísimas de consultar, a cambio de datos que pueden estar desactualizados. Úsalas para reportes pesados que se consultan mucho.
- Errores clásicos que ya sabes evitar: subconsulta con más filas de las esperadas, alias olvidado en el `FROM`, `NOT IN` con `NULLs`, índices en orden equivocado, funciones sobre columnas indexadas en el `WHERE`, y `SELECT *` dentro de las vistas.

Con esto completas las clases 25, 26 y 27. Ya puedes escribir consultas que la mayoría de los desarrolladores no se atreve a intentar, acelerar tablas lentas con índices bien pensados y organizar el acceso a datos con vistas. En la siguiente unidad seguiremos avanzando. Sigue practicando: la diferencia entre saber SQL y dominar SQL está en las horas que pases escribiendo consultas tú mismo.

UNIDAD 9: PROYECTO: TIENDA EN LÍNEA COMPLETA

Bienvenido a la unidad más importante de todo el curso. No porque las anteriores no importen, al contrario: sin ellas no podrías estar aquí. Pero esta unidad es diferente porque ya no vamos a aprender comandos sueltos ni hacer ejercicios de juguete. En esta unidad vas a construir algo real.

Vas a diseñar y programar la base de datos de una tienda en línea de verdad, del tipo que usan las empresas que venden por internet todos los días. Clientes que se registran, productos con categorías y existencias, pedidos con sus detalles, y reportes de negocio que un gerente revisaría cada mañana para tomar decisiones. Cuando termines esta unidad, vas a tener un proyecto completo que puedes presumir en tu portafolio.

Piensa en lo que has recorrido. En las unidades anteriores aprendiste qué es SQL y cómo crear bases de datos y tablas. Aprendiste los tipos de datos y las restricciones. Dominaste INSERT, SELECT, UPDATE y DELETE. Luchaste con las condiciones WHERE y los operadores lógicos. Te peleaste con las funciones de agregado y GROUP BY. Sobreviviste a los JOINS y a las llaves foráneas. Pasaste por subconsultas, vistas, transacciones e índices. Todo eso fue entrenamiento. Esta unidad es el partido.

La diferencia entre saber SQL y saber usar SQL está justo aquí. Mucha gente termina un curso de SQL sabiendo hacer un SELECT con un JOIN, pero cuando les piden "arma la base de datos de mi negocio", se quedan en blanco. No saben por dónde empezar. ¿Qué tablas creó? ¿Qué columnas lleva cada una? ¿Cómo se relacionan? ¿Cómo respondo preguntas del negocio como "qué producto se vende más" o "qué clientes no han comprado en meses"? Esta unidad te enseña exactamente eso: el proceso completo, desde los requerimientos hasta los reportes.

Vamos a trabajar con un caso concreto: TiendaTech, una tienda en línea que vende tecnología. Computadoras, accesorios, componentes, audio. Tiene clientes registrados, catálogo de productos organizados por categorías, y cada día recibe pedidos que pueden incluir varios productos. El dueño de la tienda te contrata a ti, que ya sabes SQL, para que diseñes su base de datos desde cero y le entregues los reportes que necesita para dirigir su negocio.

La unidad está dividida en dos grandes bloques que corresponden a las clases 28 y 29 del curso.

En la primera parte vamos a hacer el análisis de requerimientos: qué necesita la tienda, qué entidades existen, cómo se relacionan. Vamos a dibujar el modelo entidad-relación con palabras, tabla por tabla. Vamos a revisar las formas normales, porque es aquí donde de verdad se entienden: vas a ver por qué repetimos datos es un error caro y cómo la normalización nos ahorra dolores de cabeza. Tomaremos decisiones de diseño reales: qué tipo de dato para cada columna, qué llaves usar, qué índices conviene crear y cuáles no. Y cerraremos con la estrategia de reportes: antes de escribir una sola consulta de negocio, hay que pensar qué preguntas se hace el gerente de la tienda.

En la segunda parte viene lo sabroso: los reportes. Top 5 de productos más vendidos, ventas por mes, clientes inactivos, ticket promedio, productos con bajo stock, el mejor cliente de la tienda, las categorías que más ingresos generan. Cada reporte es una consulta que combina JOINS, agregados, subconsultas y todo lo que ya dominas, pero ahora con un propósito de negocio detrás. Ya no es "haz un GROUP BY porque sí", es "el dueño necesita saber esto para decidir qué resurtir".

Y para cerrar, la sección de práctica te va a poner a prueba con ejercicios que extienden el proyecto: agregar una tabla de reseñas, sacar reportes por ciudad, encontrar productos que nunca se han vendido. Con soluciones explicadas paso a paso, como siempre.

Un consejo antes de empezar: no leas esta unidad como lees las demás. Abre tu gestor de base de datos, MySQL, PostgreSQL, el que uses, y ejecuta cada bloque de código. Rompe cosas. Cambia los datos. Pregúntate qué pasaría si haces las cosas de otra manera. Esta unidad se aprende con las manos en el teclado, no con los ojos en la pantalla.

¿Listo? Vamos a construir una tienda.

Explicación detallada

El análisis de requerimientos: antes del código, las preguntas

Todo proyecto de base de datos empieza mucho antes del primer CREATE TABLE. Empieza con preguntas. Cuando el dueño de TiendaTech te contrata, lo primero que haces no es abrir el editor de SQL, es sentarte con él y preguntar cómo funciona su negocio. Eso se llama análisis de requerimientos, y es la habilidad que separa al técnico que ejecuta órdenes del profesional que diseña soluciones.

Estas son las preguntas que le harías al dueño, y sus respuestas:

¿Qué vende la tienda? Productos de tecnología: laptops, accesorios, componentes de computadora, equipo de audio. Cada producto tiene un nombre, una descripción, un precio, una cantidad en existencia y pertenece a una categoría.

¿Cómo están organizados los productos? Por categorías: Computo, Accesorios, Componentes, Audio. Un producto pertenece a una sola categoría, pero una categoría agrupa muchos productos.

¿Quiénes compran? Clientes registrados. Cada cliente tiene nombre, correo electrónico, teléfono, ciudad y fecha de registro. Un cliente puede hacer muchos pedidos a lo largo del tiempo.

¿Cómo es un pedido? Un pedido lo hace un cliente en una fecha determinada. Tiene un estado: pendiente, pagado, enviado o cancelado. Y lo más importante: un pedido puede incluir varios productos, cada uno con su cantidad. Por ejemplo, el pedido 101 del cliente Ana incluye 1 laptop y 2 mouse.

¿Qué necesita saber el dueño? Quiere reportes: qué productos se venden más, cuánto se vende cada mes, qué clientes no han comprado últimamente, cuál es el ticket promedio, qué productos están por agotarse.

Fíjate en algo: cada respuesta a estas preguntas se convierte, casi directamente, en una tabla o en una relación. "Clientes registrados" se convierte en la tabla clientes. "Un pedido puede incluir varios productos" se convierte en la relación más importante del modelo. El análisis de requerimientos no es burocracia, es el plano del edificio. Sin plano, terminas construyendo paredes donde van las puertas.

El modelo entidad-relación explicado en texto

Un modelo entidad-relación es el dibujo de las tablas y cómo se conectan. Normalmente se dibuja con rectángulos y líneas, pero aquí lo vamos a explicar con palabras, tabla por tabla, porque lo importante es que entiendas la lógica, no que sepas dibujar diagramas bonitos.

Las entidades de TiendaTech son cinco: clientes, categorías, productos, pedidos y detalle_pedido. Vamos una por una.

CLIENTES. Representa a cada persona registrada en la tienda. Sus atributos son: un identificador único (id_cliente), nombre completo, correo electrónico, teléfono, ciudad y fecha de registro. El correo electrónico debe ser único, porque dos clientes no pueden compartir la misma cuenta de correo: es como su nombre de usuario. Un cliente existe independientemente de si ha comprado o no. Puede registrarse hoy y comprar dentro de seis meses.

CATEGORIAS. Agrupa los productos. Sus atributos son: identificador único (`id_categoria`), nombre de la categoría y una descripción. Ejemplos: Computo, Accesorios, Componentes, Audio. Es una tabla pequeña, de catálogo, pero muy útil: sin ella, cada producto tendría que repetir el nombre de su categoría como texto, y ya veremos por qué eso es un problema.

PRODUCTOS. El catálogo de la tienda. Sus atributos son: identificador único (`id_producto`), nombre del producto, descripción, precio, existencia (cuántas unidades hay en el almacén) y la categoría a la que pertenece. Aquí aparece la primera relación: cada producto pertenece a una categoría. Eso se representa con una llave foránea, `id_categoria`, que apunta a la tabla `categorias`. La relación es de uno a muchos: una categoría tiene muchos productos, pero cada producto tiene una sola categoría.

PEDIDOS. Cada compra que hace un cliente. Sus atributos son: identificador único (`id_pedido`), el cliente que lo hizo, la fecha del pedido y el estado (pendiente, pagado, enviado, cancelado). Segunda relación: cada pedido pertenece a un cliente, mediante la llave foránea `id_cliente`. Un cliente puede tener muchos pedidos, pero cada pedido es de un solo cliente. Uno a muchos, otra vez.

DETALLE_PEDIDO. Esta es la tabla más interesante del modelo, así que ponle atención. Representa cada renglón dentro de un pedido: qué producto se compró, cuántas unidades y a qué precio se vendió en ese momento. Sus atributos son: identificador único (`id_detalle`), el pedido al que pertenece (`id_pedido`), el producto comprado (`id_producto`), la cantidad y el `precio_unitario`.

¿Por qué existe esta tabla? Porque la relación entre pedidos y productos es de muchos a muchos: un pedido incluye muchos productos, y un producto aparece en muchos pedidos. En las bases de datos relacionales, una relación de muchos a muchos no se puede representar directamente con una llave foránea. Se necesita una tabla intermedia que la descomponga en dos relaciones de uno a muchos: un pedido tiene muchos detalles, y un producto aparece en muchos detalles. Esa tabla intermedia es `detalle_pedido`, y además nos sirve para guardar información propia de la relación: la cantidad y el precio al que se vendió.

Ojo con el `precio_unitario` en `detalle_pedido`. ¿Por qué guardar el precio ahí si el producto ya tiene un precio en la tabla `productos`? Porque los precios cambian. Si la laptop costaba 15000 pesos en enero y en marzo sube a 16500, el pedido de enero debe reflejar el precio de enero. Si solo guardáramos el precio actual del producto, los reportes históricos se volverían falsos. Esta es una de esas decisiones de diseño que distinguen a un profesional: pensar en el tiempo, no solo en el presente.

En resumen, el modelo queda así, en texto:

- categorías (1) ---< (muchos) productos. Una categoría agrupa muchos productos.
- clientes (1) ---< (muchos) pedidos. Un cliente hace muchos pedidos.
- pedidos (1) ---< (muchos) detalle_pedido. Un pedido tiene muchos renglones.
- productos (1) ---< (muchos) detalle_pedido. Un producto aparece en muchos renglones.

Cuatro relaciones de uno a muchos, cero relaciones de muchos a muchos directas. Así debe quedar un modelo bien diseñado: las relaciones muchos a muchos siempre se resuelven con una tabla intermedia.

Normalización: 1FN, 2FN y 3FN con ejemplos de la tienda

La normalización es un conjunto de reglas para organizar las tablas de forma que no haya datos repetidos ni inconsistencias. Suena teórico, pero en esta tienda vas a ver exactamente para qué sirve. Vamos con las tres primeras formas normales, con ejemplos sacados de TiendaTech.

PRIMERA FORMA NORMAL (1FN): cada celda debe tener un solo valor, no listas ni grupos repetidos.

Imagina que un diseñador novato propone esta tabla de pedidos:

pedidos_mal_diseñados: id_pedido, id_cliente, fecha, productos (texto con "laptop, mouse, teclado").

El problema es evidente: la columna productos guarda varios valores en una sola celda. ¿Cómo cuentas cuántas laptops se vendieron? Tendrías que partir el texto. ¿Cómo sabes la cantidad de cada uno? No puedes. ¿Cómo haces un reporte de top productos? Imposible sin trucos sucios.

La 1FN nos dice: cada renglón de detalle_pedido guarda UN producto con SU cantidad. Por eso existe la tabla detalle_pedido: es la aplicación directa de la primera forma normal. Cada celda, un solo valor. Sin listas, sin textos con comas.

SEGUNDA FORMA NORMAL (2FN): la tabla debe estar en 1FN y además cada columna que no sea llave debe depender de TODA la llave primaria, no solo de una parte.

Este problema aparece en tablas con llave primaria compuesta. Supón que alguien diseña detalle_pedido con llave primaria compuesta por (id_pedido, id_producto) y además agrega la columna nombre_cliente en esa misma tabla. El nombre del cliente depende solo de id_pedido (a través del cliente del pedido), no de la combinación completa. Eso viola la 2FN.

La solución es no poner el nombre del cliente en detalle_pedido. El nombre del cliente vive en clientes, y se llega a él por la cadena de relaciones: detalle_pedido -> pedidos -> clientes. Cada dato vive en un solo lugar. Si el cliente cambia su nombre, se actualiza en un solo renglón de una sola tabla, no en cientos de renglones de detalle.

TERCERA FORMA NORMAL (3FN): la tabla debe estar en 2FN y además ninguna columna que no sea llave debe depender de otra columna que tampoco sea llave. Dicho simple: nada de dependencias transitivas.

El ejemplo clásico en nuestra tienda: supón que en la tabla productos alguien agrega las columnas categoria_nombre y categoria_descripcion, además de id_categoria. El nombre de la categoría depende de id_categoria, no del producto directamente. Es una dependencia transitiva: producto -> categoría -> nombre de la categoría. Eso viola la 3FN.

¿Por qué es malo? Porque si la tienda decide renombrar la categoría "Computo" a "Computadoras", tendría que actualizar esa columna en todos los productos de esa categoría. Si se le olvida uno, la base queda inconsistente: el mismo id_categoria con dos nombres distintos. En cambio, con el diseño normalizado, el nombre de la categoría vive solo en la tabla categorias. Se cambia una vez, en un solo renglón, y listo.

Nuestro diseño de cinco tablas ya cumple con 1FN, 2FN y 3FN. Cada dato vive en un solo lugar. Esa es la esencia de la normalización: un solo lugar para cada dato, cero repetición, cero inconsistencias.

Un comentario honesto: en el mundo real, a veces se rompe la normalización a propósito por rendimiento, por ejemplo guardando un total calculado en la tabla pedidos para no sumar los detalles cada vez. Eso se llama desnormalización y es una decisión consciente que se toma cuando ya se entiende la normalización. Por ahora, quédate con el diseño normalizado: es la base correcta.

Decisiones de diseño: tipos de datos, llaves e índices

Diseñar no es solo decidir qué tablas existen, también es decidir cómo se construyen por dentro. Cada columna necesita un tipo de dato, cada tabla necesita su llave primaria, y hay que pensar en índices. Estas decisiones parecen pequeñas, pero en una tienda real con miles de pedidos al día, una mala decisión aquí se paga caro.

TIPOS DE DATOS. La regla de oro es: usa el tipo más preciso y más pequeño que represente bien el dato.

Para los identificadores usamos INT con AUTO_INCREMENT (en MySQL) o SERIAL (en PostgreSQL) o INTEGER PRIMARY KEY AUTOINCREMENT (en SQLite). Son números

enteros que la base genera sola: 1, 2, 3. Nunca uses el correo del cliente o el nombre del producto como llave primaria. ¿Por qué? Porque los datos de negocio cambian: la gente cambia de correo, los productos se renombran. Una llave primaria debe ser estable, sin significado de negocio, y generada por el sistema. Esa es una llave subrogada, y es la práctica estándar.

Para nombres, correos, teléfonos y ciudades usamos VARCHAR con un tamaño razonable: VARCHAR(100) para nombres, VARCHAR(150) para correos, VARCHAR(20) para teléfonos, VARCHAR(80) para ciudades. No pongas VARCHAR(255) a todo por flojera: tamaños adecuados documentan tu intención y en tablas enormes sí importan.

Para descripciones de productos usamos TEXT, porque pueden ser largas y no queremos limitarlas.

Para precios usamos DECIMAL(10,2). Nunca, jamás, uses FLOAT o REAL para dinero. Los tipos de punto flotante guardan aproximaciones: $0.1 + 0.2$ no es exactamente 0.3 en punto flotante. En una tienda, un centavo de error multiplicado por miles de pedidos es un problema contable real. DECIMAL guarda el valor exacto. El (10,2) significa hasta 10 dígitos en total, 2 después del punto: suficiente para precios de hasta 99 millones con centavos.

Para existencias y cantidades usamos INT. No pueden ser negativas en la lógica del negocio, así que agregamos una restricción CHECK (existencia ≥ 0) donde el motor lo soporte.

Para fechas usamos DATE para la fecha de registro del cliente y DATETIME o TIMESTAMP para la fecha del pedido, porque en un pedido sí importa la hora. Guarda siempre las fechas en el formato de la base de datos, no como texto: solo así puedes filtrar por mes, comparar rangos y agrupar por periodo.

Para el estado del pedido usamos VARCHAR(20) con un CHECK que limite los valores permitidos: pendiente, pagado, enviado, cancelado. Otra opción es ENUM en MySQL. Lo importante es que la base de datos impida estados inventados como "en camino más o menos".

LLAVES. Cada tabla lleva su PRIMARY KEY en su identificador. Las llaves foráneas llevan REFERENCES a la tabla padre. Y aquí viene una decisión importante: ¿qué pasa cuando se borra un registro padre?

Si se borra un cliente, ¿se borran sus pedidos? En una tienda real, no: los pedidos son historia contable, no se borran. Por eso en pedidos usamos ON DELETE RESTRICT para id_cliente: la base impide borrar un cliente que tenga pedidos. Si se borra una categoría, ¿qué pasa con sus productos? Tampoco queremos huérfanos, así que RESTRICT también, o mejor aún: obligamos a reasignar los productos a otra categoría antes de borrar.

En `detalle_pedido` usamos `ON DELETE CASCADE` para `id_pedido`: si por alguna razón administrativa se borra un pedido (por ejemplo, un pedido de prueba), sus renglones de detalle deben borrarse con él, porque un detalle sin pedido no tiene sentido. En cambio, para `id_producto` usamos `RESTRICT`: no puedes borrar un producto que ya aparece en pedidos históricos, porque romperías la historia de ventas.

Estas decisiones no son capricho: protegen la integridad del negocio. La base de datos es la última línea de defensa contra errores de la aplicación.

ÍNDICES. Ya sabes de la unidad de índices que aceleran las búsquedas pero cuestan en inserciones. En nuestra tienda, ¿dónde convienen?

Primero, el correo del cliente lleva `UNIQUE`, que crea un índice automáticamente: los inicios de sesión buscan por correo constantemente. Segundo, las llaves foráneas (`id_cliente` en `pedidos`, `id_categoria` en `productos`, `id_pedido` e `id_producto` en `detalle_pedido`) se benefician de índices porque los `JOINS` las usan todo el tiempo. La mayoría de los motores crean índices automáticamente para llaves primarias y únicas, pero para las foráneas conviene crearlos explícitamente.

Tercero, la fecha del pedido merece un índice: casi todos los reportes filtran o agrupan por fecha (ventas por mes, ventas del último trimestre). Sin índice, cada reporte haría un escaneo completo de la tabla `pedidos`.

¿Qué NO indexamos? La descripción `TEXT` de productos: indexar texto largo es caro y casi nadie busca por descripción exacta. El teléfono del cliente: se usa poco en búsquedas. Recuerda la regla: indexa lo que filtras, ordenas o unes con frecuencia; no indexes por si acaso.

Estrategia de reportes: qué preguntan los gerentes

Antes de escribir consultas, hay que pensar como gerente. Un reporte no es un ejercicio de SQL, es una respuesta a una pregunta de negocio. Si no entiendes la pregunta, la consulta más elegante del mundo no sirve para nada.

Estas son las preguntas típicas que hace el dueño de `TiendaTech`, y qué técnica de SQL responde cada una:

¿Qué se vende más? Es la pregunta del top de productos. Necesita sumar las cantidades vendidas por producto en `detalle_pedido`, ordenar de mayor a menor y quedarse con los primeros. Técnica: `JOIN` entre `productos` y `detalle_pedido`, `SUM(cantidad)`, `GROUP BY producto`, `ORDER BY DESC`, `LIMIT`.

¿Cuánto vendimos cada mes? La pregunta de la tendencia. Necesita agrupar los pedidos por mes y sumar el ingreso de cada mes. Técnica: extraer año y mes de la fecha, GROUP BY periodo, SUM(cantidad * precio_unitario). Aquí se ve por qué guardamos el precio_unitario en el detalle: el ingreso histórico debe calcularse con el precio de la venta, no con el precio actual.

¿Qué clientes no han comprado? La pregunta de los inactivos. Necesita todos los clientes MENOS los que tienen pedidos recientes. Técnica: LEFT JOIN de clientes a pedidos con filtro de fecha, y quedarse con los que quedaron en NULL. Es el patrón clásico de "los que no tienen".

¿Cuánto compra en promedio cada pedido? El ticket promedio. Técnica: sumar el total de cada pedido y promediar, o dividir el ingreso total entre el número de pedidos. Ojo: hay que decidir si se incluyen los pedidos cancelados. Normalmente no: un pedido cancelado no generó ingreso.

¿Qué productos están por agotarse? La pregunta del inventario. Necesita los productos con existencia baja. Técnica: WHERE existencia < umbral, ORDER BY existencia. Simple, pero salva a la tienda de quedarse sin stock del producto estrella.

¿Quién es mi mejor cliente? El que más dinero ha dejado. Técnica: JOIN clientes-pedidos-detalle, SUM por cliente, ORDER BY DESC, LIMIT 1. Con ese dato el dueño puede consentirlo: un descuento, un correo de agradecimiento.

¿Qué categorías dejan más dinero? No basta saber qué producto se vende más en unidades: una categoría puede vender pocas unidades pero caras. Técnica: JOIN de detalle_pedido con productos y categorías, SUM(cantidad * precio_unitario) agrupado por categoría.

Fíjate en el patrón: todas las preguntas de negocio se traducen a combinaciones de JOIN, GROUP BY, funciones de agregado y filtros. Ya dominas cada pieza por separado. Los reportes son el arte de combinarlas con un propósito. Esa es la mentalidad que debes llevarte de esta unidad: SQL no es el fin, es la herramienta para responder preguntas que valen dinero.

Errores comunes al diseñar (y cómo evitarlos)

Antes de pasar a los ejemplos, vale la pena repasar los errores que casi todos cometen la primera vez que diseñan una base de datos real. Conocerlos de antemano te ahorra semanas de retrabajo.

ERROR 1: guardar listas en una sola columna. Es el más común y el más dañino. "Voy a guardar los productos del pedido en una columna de texto separados por comas, es más fácil". No, no es más fácil: es imposible de consultar bien. No puedes sumar, no puedes filtrar, no

puedes unir. La solución siempre es una tabla intermedia, como `detalle_pedido`. Si sientes la tentación de meter comas en una celda, es señal de que falta una tabla.

ERROR 2: usar datos de negocio como llave primaria. El correo del cliente como llave primaria suena lógico hasta que el cliente cambia de correo y tienes que actualizar su identificador en pedidos, detalles, reseñas y donde sea que aparezca. Con una llave subrogada (un número sin significado), el correo cambia en un solo renglón y las relaciones ni se enteran. Las llaves primarias deben ser estables, únicas y sin significado. Punto.

ERROR 3: repetir datos "para no hacer JOINS". "Voy a guardar el nombre del cliente en cada pedido para no tener que unir tablas". Al principio funciona. Seis meses después, un cliente cambia su nombre y queda la mitad de sus pedidos con el nombre viejo. Ahora tienes dos versiones de la verdad y ningún reporte confiable. Los JOINS existen precisamente para no repetir datos. La computadora hace el JOIN en milisegundos; el humano corrigiendo inconsistencias tarda semanas.

ERROR 4: no pensar en el tiempo. Guardar solo el precio actual del producto y calcular el historial con él, no registrar cuándo cambió un estado, borrar pedidos viejos "porque ya pasaron". Los negocios viven del historial: comparan este mes contra el mismo mes del año pasado, analizan tendencias, auditan. Nuestra decisión de guardar `precio_unitario` en `detalle_pedido` es pensar en el tiempo. Cada vez que diseñes, pregúntate: ¿qué va a necesitar saber el dueño dentro de un año sobre lo que pasa hoy?

ERROR 5: olvidar las restricciones y confiar en la aplicación. "Mi programa nunca va a insertar un precio negativo, no necesito el CHECK". Hasta que otro programa se conecta a la base, o alguien importa un CSV a mano, o hay un bug. Las restricciones en la base de datos son la última línea de defensa: no confían en nadie y siempre están vigilando. Úsalas.

ERROR 6: indexar todo o no indexar nada. Indexar cada columna "por si acaso" vuelve lentas las inserciones y desperdicia espacio. No indexar nada vuelve lentos los reportes. El equilibrio está en indexar lo que filtras, unes y ordenas con frecuencia: llaves foráneas, fechas de reportes, columnas de búsqueda como el correo. Y medir: si un reporte tarda, EXPLAIN te dice si el índice se está usando.

ERROR 7: no probar con datos realistas. Diseñar con tres renglones de prueba y declarar victoria. Los problemas aparecen con datos de verdad: el cliente sin pedidos, el producto sin ventas, el pedido cancelado, el mes sin ventas. Por eso nuestros datos de ejemplo incluyen casos incómodos a propósito. Cuando pruebes tus reportes, busca activamente los casos borde: son los que rompen consultas en producción.

Si evitas estos siete errores, tu diseño ya está por encima del promedio de lo que se ve en el mundo real. No porque seas un genio, sino porque la mayoría de la gente no se detiene a pensar antes de crear tablas. Tú sí.

Ejemplos paso a paso

Manos a la obra. Vamos a construir TiendaTech completa: base de datos, tablas, datos de ejemplo y todos los reportes. Ejecuta cada bloque en tu gestor. Si usas MySQL, PostgreSQL o SQLite, el código funciona con ajustes mínimos que se indican donde importa.

Paso 1: crear la base de datos

Siempre empezamos creando la base y seleccionándola. Nada nuevo aquí, pero es el cimiento.

```
CREATE DATABASE tienda_linea
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_unicode_ci;

USE tienda_linea;
```

La especificación del juego de caracteres utf8mb4 es importante en MySQL: permite guardar tildes, eñes y cualquier carácter sin problemas. En PostgreSQL la base ya suele crearse en UTF-8 por defecto. En SQLite no necesitas este paso, solo abres o creas el archivo.

Paso 2: tabla de categorías

Creemos primero las tablas que no dependen de nadie, las tablas padre. categorías es la más simple: solo es un catálogo.

```
CREATE TABLE categorias (
  id_categoria INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(60) NOT NULL UNIQUE,
  descripcion VARCHAR(255)
);
```

Observa los detalles: id_categoria es la llave primaria autoincremental. nombre es NOT NULL y UNIQUE: no puede haber dos categorías con el mismo nombre ni categorías sin nombre. La descripción es opcional. Esta tabla es pequeña pero cumple una función grande: es la referencia que evita repetir el nombre de la categoría en cada producto, la 3FN en acción.

Nota para PostgreSQL: cambia INT AUTO_INCREMENT por SERIAL PRIMARY KEY. Para SQLite: INTEGER PRIMARY KEY AUTOINCREMENT.

Paso 3: tabla de clientes

```
CREATE TABLE clientes (  
  id_cliente INT AUTO_INCREMENT PRIMARY KEY,  
  nombre VARCHAR(100) NOT NULL,  
  correo VARCHAR(150) NOT NULL UNIQUE,  
  telefono VARCHAR(20),  
  ciudad VARCHAR(80) NOT NULL,  
  fecha_registro DATE NOT NULL  
);  
  
CREATE INDEX idx_clientes_ciudad ON clientes (ciudad);
```

El correo es UNIQUE porque funciona como identificador de cuenta: dos clientes no pueden compartirlo. El teléfono es opcional, por eso no lleva NOT NULL. La ciudad es obligatoria porque la usaremos en reportes geográficos. Creamos un índice en ciudad porque el reporte de ventas por ciudad filtrará por ella con frecuencia.

Paso 4: tabla de productos

```
CREATE TABLE productos (  
  id_producto INT AUTO_INCREMENT PRIMARY KEY,  
  nombre VARCHAR(120) NOT NULL,  
  descripcion TEXT,  
  precio DECIMAL(10,2) NOT NULL CHECK (precio > 0),  
  existencia INT NOT NULL DEFAULT 0 CHECK (existencia >= 0),  
  id_categoria INT NOT NULL,  
  CONSTRAINT fk_productos_categoria  
    FOREIGN KEY (id_categoria)  
    REFERENCES categorias (id_categoria)  
    ON DELETE RESTRICT  
    ON UPDATE CASCADE  
);
```

Aquí está la primera llave foránea: id_categoria apunta a categorias. ON DELETE RESTRICT impide borrar una categoría que tenga productos: primero reasigna o borra los productos. ON UPDATE CASCADE propaga cambios al identificador, aunque como usamos llaves subrogadas eso casi nunca pasará. El precio usa DECIMAL(10,2) y no puede ser cero ni

negativo. La existencia no puede ser negativa: la base de datos impide el error antes de que ocurra.

Paso 5: tabla de pedidos

```
CREATE TABLE pedidos (  
  id_pedido INT AUTO_INCREMENT PRIMARY KEY,  
  id_cliente INT NOT NULL,  
  fecha_pedido DATETIME NOT NULL,  
  estado VARCHAR(20) NOT NULL DEFAULT 'pendiente'  
  CHECK (estado IN ('pendiente', 'pagado', 'enviado', 'cancelado')),  
  CONSTRAINT fk_pedidos_cliente  
    FOREIGN KEY (id_cliente)  
    REFERENCES clientes (id_cliente)  
    ON DELETE RESTRICT  
    ON UPDATE CASCADE  
);  
  
CREATE INDEX idx_pedidos_fecha ON pedidos (fecha_pedido);  
CREATE INDEX idx_pedidos_cliente ON pedidos (id_cliente);
```

El estado tiene un valor por defecto 'pendiente': todo pedido nace pendiente. El CHECK limita los estados a los cuatro válidos. ON DELETE RESTRICT protege la historia: no puedes borrar un cliente con pedidos. El índice en fecha_pedido es clave para los reportes mensuales. El índice en id_cliente acelera los JOINS con clientes.

Paso 6: tabla de detalle_pedido, la tabla intermedia


```

CREATE TABLE detalle_pedido (
  id_detalle INT AUTO_INCREMENT PRIMARY KEY,
  id_pedido INT NOT NULL,
  id_producto INT NOT NULL,
  cantidad INT NOT NULL CHECK (cantidad > 0),
  precio_unitario DECIMAL(10,2) NOT NULL CHECK (precio_unitario > 0),
  CONSTRAINT fk_detalle_pedido
    FOREIGN KEY (id_pedido)
    REFERENCES pedidos (id_pedido)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  CONSTRAINT fk_detalle_producto
    FOREIGN KEY (id_producto)
    REFERENCES productos (id_producto)
    ON DELETE RESTRICT
    ON UPDATE CASCADE
);

CREATE INDEX idx_detalle_pedido ON detalle_pedido (id_pedido);
CREATE INDEX idx_detalle_producto ON detalle_pedido (id_producto);

```

Esta tabla resuelve la relación de muchos a muchos entre pedidos y productos. Fíjate en las dos llaves foráneas con comportamientos distintos: si se borra un pedido, sus detalles se borran en cascada (CASCADE), porque un renglón sin pedido no tiene sentido. Pero si intentas borrar un producto que ya se vendió, la base lo impide (RESTRICT), porque romperías la historia de ventas. El precio_unitario guarda el precio al momento de la venta: si el producto sube de precio después, el historial queda intacto. La cantidad debe ser mayor a cero: no existen renglones de cero productos.

Con estas cinco tablas, el modelo está completo y normalizado. Tómate un momento para apreciarlo: cada dato tiene un solo hogar, las relaciones están protegidas por llaves foráneas, y los índices están donde los reportes los van a necesitar.

Paso 7: llenar las categorías

Una tienda sin datos es un cascarón vacío. Vamos a poblar TiendaTech con datos realistas. Primero las categorías, porque productos depende de ellas.

```
INSERT INTO categorias (nombre, descripcion) VALUES
  ('Computo', 'Laptops, computadoras de escritorio y tabletas'),
  ('Accesorios', 'Mouse, teclados, mochilas y periféricos'),
  ('Componentes', 'Memorias, discos duros, tarjetas y procesadores'),
  ('Audio', 'Audífonos, bocinas y micrófonos');
```

Cuatro categorías, con identificadores 1 a 4 generados automáticamente. Verifica con un SELECT que quedaron bien antes de seguir: si las categorías quedan mal, todo lo demás se construye sobre un error.

Paso 8: registrar clientes

Ocho clientes de distintas ciudades, con fechas de registro variadas. Algunos comprarán mucho, otros poco, y un par no comprará nada: los necesitamos así para que el reporte de clientes inactivos tenga algo que encontrar.

```
INSERT INTO clientes (nombre, correo, telefono, ciudad, fecha_registro) VALUES
  ('Ana Martínez', 'ana.martinez@correo.com', '5551234567', 'Ciudad de México',
  '2025-06-10'),
  ('Carlos Ruiz', 'carlos.ruiz@correo.com', '5559876543', 'Guadalajara',
  '2025-07-02'),
  ('Lucía Fernández', 'lucia.fernandez@correo.com', '5554567890', 'Monterrey',
  '2025-08-15'),
  ('Miguel Torres', 'miguel.torres@correo.com', '5553210987', 'Ciudad de México',
  '2025-09-01'),
  ('Sofía Hernández', 'sofia.hernandez@correo.com', '5556543210', 'Puebla',
  '2025-10-20'),
  ('Diego Ramírez', 'diego.ramirez@correo.com', '5557890123', 'Guadalajara',
  '2026-01-12'),
  ('Valeria Castro', 'valeria.castro@correo.com', '5552345678', 'Querétaro',
  '2026-02-05'),
  ('Jorge Morales', 'jorge.morales@correo.com', '5558765432', 'Monterrey',
  '2026-03-18');
```

Los identificadores quedan del 1 al 8 en ese orden. Ana es la clienta 1, Jorge el cliente 8. Guárdate esa correspondencia en la cabeza porque los pedidos la van a usar. Nota que Diego, Valeria y Jorge se registraron en 2026 y todavía no tienen pedidos: serán nuestros clientes inactivos. Eso no es casualidad, es diseño de datos de prueba: cuando pruebes reportes, necesitas casos que ejerciten cada rama de la lógica.

Paso 9: cargar el catálogo de productos

Doce productos repartidos en las cuatro categorías, con precios y existencias realistas en pesos mexicanos. Algunos con mucho stock, un par casi agotados para probar el reporte de bajo stock.

```
INSERT INTO productos (nombre, descripcion, precio, existencia, id_categoria)
VALUES
  ('Laptop Pro 14', 'Laptop de 14 pulgadas, 16GB RAM, 512GB SSD', 18999.00, 25, 1),
  ('Laptop Básica 15', 'Laptop de 15 pulgadas para oficina y escuela', 9499.00, 40, 1),
  ('Tableta 10 pulg', 'Tableta con lápiz óptico incluido', 6299.00, 30, 1),
  ('Mouse inalámbrico', 'Mouse ergonómico inalámbrico 2.4GHz', 449.00, 150, 2),
  ('Teclado mecánico RGB', 'Teclado mecánico con retroiluminación RGB', 1299.00, 80, 2),
  ('Mochila para laptop', 'Mochila acolchada para laptops hasta 16 pulg', 799.00, 60, 2),
  ('Memoria RAM 16GB', 'Memoria DDR4 16GB 3200MHz para laptop', 1199.00, 100, 3),
  ('Disco SSD 1TB', 'Unidad de estado sólido NVMe 1TB', 1899.00, 70, 3),
  ('Tarjeta gráfica RTX', 'Tarjeta gráfica 8GB GDDR6 para gaming', 9499.00, 4, 3),
  ('Audífonos Bluetooth', 'Audífonos con cancelación de ruido', 1599.00, 90, 4),
  ('Bocina portátil', 'Bocina Bluetooth resistente al agua', 899.00, 55, 4),
  ('Micrófono USB', 'Micrófono condensador USB para podcast', 1399.00, 3, 4);
```

Los identificadores van del 1 al 12 en orden. Fíjate en los detalles diseñados a propósito: la tarjeta gráfica (id 9) tiene solo 4 unidades y el micrófono USB (id 12) solo 3. Esos van a aparecer en el reporte de bajo stock. Los precios están en DECIMAL con dos decimales, como debe ser. Las categorías 1 a 4 corresponden a Computo, Accesorios, Componentes y Audio.

Un consejo de profesional: cuando insertes datos de prueba, hazlo con intención. Cada dato raro que metas (un stock de 3, un cliente sin pedidos, un pedido cancelado) es un caso de prueba para tus reportes. Los datos aburridos y parejos no prueban nada.

Paso 10: registrar los pedidos

Doce pedidos distribuidos en varios meses, con distintos clientes y estados. La mayoría están pagados o enviados, uno sigue pendiente y uno fue cancelado: necesitamos el cancelado para probar que los reportes de ingresos lo excluyen correctamente.

```
INSERT INTO pedidos (id_cliente, fecha_pedido, estado) VALUES
  (1, '2026-01-15 10:30:00', 'enviado'),
  (2, '2026-01-22 14:15:00', 'enviado'),
  (1, '2026-02-08 09:45:00', 'enviado'),
  (3, '2026-02-19 16:20:00', 'pagado'),
  (4, '2026-03-05 11:00:00', 'enviado'),
  (2, '2026-03-18 13:30:00', 'cancelado'),
  (5, '2026-04-02 10:10:00', 'enviado'),
  (1, '2026-04-25 15:40:00', 'pagado'),
  (3, '2026-05-11 12:00:00', 'enviado'),
  (4, '2026-05-29 09:25:00', 'pendiente'),
  (2, '2026-06-14 17:05:00', 'enviado'),
  (5, '2026-07-03 11:50:00', 'pagado');
```

Los pedidos quedan con id del 1 al 12. Ana (cliente 1) tiene tres pedidos: es nuestra mejor cliente. Carlos (cliente 2) tiene tres pedidos pero uno cancelado. Lucía (3), Miguel (4) y Sofía (5) tienen dos cada uno. Los pedidos están repartidos entre enero y julio de 2026, así el reporte de ventas por mes tendrá siete meses que mostrar. El pedido 6 está cancelado y el 10 pendiente: esos dos no deben contar como ingreso.

Paso 11: el detalle de cada pedido

Ahora los renglones de cada pedido: qué producto, cuántos, a qué precio. Este es el bloque más largo, pero es el corazón de la tienda: sin detalles no hay ventas que reportar.

```

INSERT INTO detalle_pedido (id_pedido, id_producto, cantidad, precio_unitario)
VALUES
  (1, 1, 1, 18999.00),
  (1, 4, 2, 449.00),
  (2, 7, 2, 1199.00),
  (2, 8, 1, 1899.00),
  (3, 5, 1, 1299.00),
  (3, 4, 1, 449.00),
  (4, 10, 2, 1599.00),
  (4, 11, 1, 899.00),
  (5, 2, 1, 9499.00),
  (5, 6, 1, 799.00),
  (6, 9, 1, 9499.00),
  (7, 3, 1, 6299.00),
  (7, 12, 1, 1399.00),
  (8, 4, 3, 449.00),
  (8, 5, 2, 1299.00),
  (9, 8, 2, 1899.00),
  (9, 7, 1, 1199.00),
  (10, 11, 2, 899.00),
  (11, 1, 1, 18999.00),
  (11, 10, 1, 1599.00),
  (12, 6, 2, 799.00),
  (12, 4, 2, 449.00);

```

Veintidós renglones de detalle. Léelos con calma y verifica la lógica: el pedido 1 (Ana) lleva 1 laptop Pro y 2 mouse. El pedido 6 (Carlos, cancelado) lleva 1 tarjeta gráfica: ese ingreso no debe aparecer en los reportes. El pedido 11 (Carlos) lleva otra laptop Pro: las laptops caras se venden poco pero dejan mucho dinero, y eso se va a notar en los reportes por categoría.

Nota que el precio_unitario coincide con el precio actual del producto en todos los casos, porque son ventas recientes. En una tienda real con historial de años, verías diferencias: el precio_unitario congelado en el tiempo contra el precio actual que ya cambió. Nuestro diseño ya está preparado para eso.

Con esto, la tienda está completa y poblada. Tienes 4 categorías, 8 clientes, 12 productos, 12 pedidos y 22 renglones de detalle. Es una tienda pequeña, pero tiene todo lo que una grande tiene: solo cambia la escala. A partir de aquí, todo lo que hagas son consultas. Y aquí empieza la segunda mitad de la unidad: los reportes.

Paso 12: reporte de top 5 productos más vendidos

La pregunta del gerente: ¿qué se vende más? Unimos productos con detalle_pedido, sumamos cantidades por producto, ordenamos de mayor a menor y nos quedamos con cinco. Excluimos los pedidos cancelados, porque una venta cancelada no es una venta.

```
SELECT
  p.nombre AS producto,
  SUM(d.cantidad) AS unidades_vendidas,
  SUM(d.cantidad * d.precio_unitario) AS ingreso_total
FROM productos p
INNER JOIN detalle_pedido d ON p.id_producto = d.id_producto
INNER JOIN pedidos pe ON d.id_pedido = pe.id_pedido
WHERE pe.estado <> 'cancelado'
GROUP BY p.id_producto, p.nombre
ORDER BY unidades_vendidas DESC
LIMIT 5;
```

Tres tablas unidas: productos para el nombre, detalle_pedido para las cantidades, pedidos para filtrar el estado. El GROUP BY agrupa por producto, el SUM suma unidades e ingresos, el ORDER BY DESC pone al campeón primero y el LIMIT 5 corta la lista. Con nuestros datos, el mouse inalámbrico debería liderar en unidades: aparece en varios pedidos con cantidades de 2 y 3. La laptop Pro venderá pocas unidades pero generará mucho ingreso: por eso mostramos ambas columnas, unidades e ingreso. Un gerente que solo mira unidades toma malas decisiones.

Paso 13: reporte de ventas por mes

La pregunta de la tendencia: ¿cuánto vendimos cada mes? Agrupamos los pedidos por año y mes, y sumamos el ingreso de cada grupo. La función para extraer año-mes varía según el motor: en MySQL es DATE_FORMAT(fecha_pedido, '%Y-%m'), en PostgreSQL es TO_CHAR(fecha_pedido, 'YYYY-MM'), en SQLite es STRFTIME('%Y-%m', fecha_pedido). Aquí va la versión MySQL:

```

SELECT
    DATE_FORMAT(pe.fecha_pedido, '%Y-%m') AS mes,
    COUNT(DISTINCT pe.id_pedido) AS num_pedidos,
    SUM(d.cantidad * d.precio_unitario) AS ingreso_mes
FROM pedidos pe
INNER JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
WHERE pe.estado IN ('pagado', 'enviado')
GROUP BY mes
ORDER BY mes;

```

Fíjate en las decisiones: el WHERE solo incluye pedidos pagados y enviados, los que sí generaron ingreso. El pendiente todavía no es dinero y el cancelado nunca lo será. COUNT(DISTINCT pe.id_pedido) cuenta pedidos sin duplicarlos por el JOIN con el detalle: si contaras renglones, un pedido con tres productos contaría como tres pedidos. Ese es un error clásico, y aquí lo evitamos. El ingreso se calcula con cantidad por precio_unitario del detalle, el precio histórico real de cada venta. Con nuestros datos verás siete meses, de 2026-01 a 2026-07, cada uno con su total.

Paso 14: reporte de clientes inactivos

La pregunta de la retención: ¿qué clientes no han comprado? Usamos el patrón LEFT JOIN más IS NULL: traemos todos los clientes y tratamos de unir sus pedidos recientes. Los que no emparejan quedan con NULL, y esos son los inactivos.

```

SELECT
    c.nombre AS cliente,
    c.correo,
    c.fecha_registro,
    MAX(pe.fecha_pedido) AS ultima_compra
FROM clientes c
LEFT JOIN pedidos pe
    ON c.id_cliente = pe.id_cliente
    AND pe.estado <> 'cancelado'
GROUP BY c.id_cliente, c.nombre, c.correo, c.fecha_registro
HAVING ultima_compra IS NULL
    OR ultima_compra < '2026-04-01';

```

Hay dos tipos de inactivos y la consulta atrapa ambos. Primero, los que nunca han comprado: el LEFT JOIN no encuentra nada, MAX(fecha_pedido) es NULL, y el HAVING los incluye. En nuestros datos son Diego, Valeria y Jorge. Segundo, los que compraron hace mucho: su última compra es anterior a la fecha de corte. El HAVING con OR cubre los dos casos.

Nota el detalle fino: la condición `pe.estado <> 'cancelado'` va en el `ON` del `JOIN`, no en el `WHERE`. Si la pusieras en el `WHERE`, convertirías el `LEFT JOIN` en un `INNER JOIN` disfrazado y perderías a los clientes sin pedidos, que son justo los que buscas. Esta es una de las trampas más comunes con `LEFT JOIN`, y aquí la ves resuelta.

Paso 15: reporte de ticket promedio

La pregunta de la eficiencia: ¿cuánto gasta en promedio cada pedido? Hay dos formas de calcularlo. La forma correcta es promediar los totales por pedido, no dividir el ingreso total entre renglones. Primero calculamos el total de cada pedido en una subconsulta, luego promediamos.

```
SELECT
  AVG(total_pedido) AS ticket_promedio,
  MIN(total_pedido) AS pedido_minimo,
  MAX(total_pedido) AS pedido_maximo,
  COUNT(*) AS pedidos_contados
FROM (
  SELECT
    pe.id_pedido,
    SUM(d.cantidad * d.precio_unitario) AS total_pedido
  FROM pedidos pe
  INNER JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
  WHERE pe.estado IN ('pagado', 'enviado')
  GROUP BY pe.id_pedido
) AS totales;
```

La subconsulta interior calcula el total de cada pedido válido. La consulta exterior promedia esos totales y además muestra el mínimo, el máximo y cuántos pedidos se contaron. Con nuestros datos, el ticket promedio andará por varios miles de pesos, empujado hacia arriba por las laptops de 18999. El pedido mínimo será modesto, algún pedido de solo accesorios. Este reporte le dice al dueño si sus estrategias para aumentar el ticket (venta cruzada, envío gratis desde cierto monto) están funcionando.

Paso 16: reporte de productos con bajo stock

La pregunta del inventario: ¿qué se está por agotar? Simple y vital: filtramos productos con existencia menor a un umbral y ordenamos de menor a mayor para atender primero lo más urgente.


```

SELECT
    p.nombre AS producto,
    c.nombre AS categoria,
    p.existencia,
    p.precio
FROM productos p
INNER JOIN categorias c ON p.id_categoria = c.id_categoria
WHERE p.existencia < 10
ORDER BY p.existencia ASC;

```

El umbral de 10 es arbitrario: cada negocio define el suyo según su velocidad de venta. Con nuestros datos aparecen la tarjeta gráfica con 4 unidades y el micrófono USB con 3. Este es el reporte que evita el peor escenario de una tienda: que tu producto estrella se agote justo cuando la demanda sube. Un gerente inteligente revisa esto cada semana y hace pedidos de resurtido a tiempo.

Una mejora profesional sería comparar la existencia contra el ritmo de ventas, no contra un número fijo: un producto que vende 100 unidades al mes con stock de 10 está en emergencia, mientras uno que vende 2 al mes con stock de 10 está tranquilo. Eso se calcula uniendo este reporte con el de unidades vendidas. Te lo dejo como idea para que lo intentes por tu cuenta.

Paso 17: reporte del mejor cliente

La pregunta de la lealtad: ¿quién es el cliente que más dinero ha dejado en la tienda? Sumamos el gasto total por cliente en pedidos válidos y nos quedamos con el primero.

```

SELECT
    c.nombre AS cliente,
    c.ciudad,
    COUNT(DISTINCT pe.id_pedido) AS num_pedidos,
    SUM(d.cantidad * d.precio_unitario) AS total_gastado
FROM clientes c
INNER JOIN pedidos pe ON c.id_cliente = pe.id_cliente
INNER JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
WHERE pe.estado IN ('pagado', 'enviado')
GROUP BY c.id_cliente, c.nombre, c.ciudad
ORDER BY total_gastado DESC
LIMIT 1;

```

Tres tablas encadenadas: clientes a pedidos, pedidos a detalle. Agrupamos por cliente, contamos sus pedidos válidos y sumamos su gasto. Con nuestros datos, Ana Martínez debería ganar: tiene tres pedidos incluyendo una laptop Pro de 18999. Quita el LIMIT 1 y tienes el

ranking completo de clientes por gasto, que sirve para programas de lealtad: a los de arriba se les consiente, a los de en medio se les motiva, a los de abajo se les reactiva.

Paso 18: reporte de categorías con más ingresos

La última pregunta estratégica: ¿qué categorías dejan más dinero? No es lo mismo vender muchas unidades baratas que pocas unidades caras. Unimos las cuatro tablas necesarias y agrupamos por categoría.

```
SELECT
  c.nombre AS categoria,
  SUM(d.cantidad) AS unidades_vendidas,
  SUM(d.cantidad * d.precio_unitario) AS ingreso_total,
  ROUND(
    SUM(d.cantidad * d.precio_unitario) * 100.0 /
    (SELECT SUM(d2.cantidad * d2.precio_unitario)
     FROM detalle_pedido d2
     INNER JOIN pedidos pe2 ON d2.id_pedido = pe2.id_pedido
     WHERE pe2.estado IN ('pagado', 'enviado')),
    2
  ) AS porcentaje_del_total
FROM categorias c
INNER JOIN productos p ON c.id_categoria = p.id_categoria
INNER JOIN detalle_pedido d ON p.id_producto = d.id_producto
INNER JOIN pedidos pe ON d.id_pedido = pe.id_pedido
WHERE pe.estado IN ('pagado', 'enviado')
GROUP BY c.id_categoria, c.nombre
ORDER BY ingreso_total DESC;
```

Esta es la consulta más completa del proyecto y vale la pena desarmarla. El FROM encadena cuatro tablas: categorias a productos, productos a detalle, detalle a pedidos. El WHERE filtra pedidos válidos. El GROUP BY agrupa por categoría. Y la columna porcentaje_del_total usa una subconsulta escalar que calcula el ingreso total de la tienda para expresar cada categoría como porcentaje. Con nuestros datos, Computo debería dominar por las laptops caras, aunque Accesorios venda más unidades. Esa es exactamente la clase de hallazgo que un gerente necesita: no todo lo que más se vende es lo que más dinero deja.

Con estos siete reportes, TiendaTech tiene su panel de control completo: qué se vende, cuánto se vende, quién compra, quién no compra, cuánto gasta cada pedido, qué se agota y qué categorías mandan. Todo con SQL puro, sobre un modelo bien diseñado. Eso es lo que vendes como profesional: no consultas, sino respuestas.

Paso 19: reporte de crecimiento mes a mes (el favorito de los gerentes)

Hay una pregunta que todo dueño hace y que ningún reporte anterior responde: ¿vamos mejorando o empeorando? Saber que en julio vendiste 50000 no dice mucho; saber que en julio vendiste 20 por ciento más que en junio, eso sí dice algo. Para comparar cada mes contra el anterior usamos una función de ventana: LAG, que mira el renglón anterior.

```
SELECT
    mes,
    ingreso_mes,
    LAG(ingreso_mes) OVER (ORDER BY mes) AS ingreso_mes_anterior,
    ROUND(
        (ingreso_mes - LAG(ingreso_mes) OVER (ORDER BY mes)) * 100.0 /
        LAG(ingreso_mes) OVER (ORDER BY mes),
        2
    ) AS crecimiento_porcentaje
FROM (
    SELECT
        DATE_FORMAT(pe.fecha_pedido, '%Y-%m') AS mes,
        SUM(d.cantidad * d.precio_unitario) AS ingreso_mes
    FROM pedidos pe
    INNER JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
    WHERE pe.estado IN ('pagado', 'enviado')
    GROUP BY mes
) AS ventas_mensuales
ORDER BY mes;
```

La subconsulta interior es el reporte de ventas por mes que ya conoces. La consulta exterior le agrega dos columnas calculadas con LAG: el ingreso del mes anterior y el porcentaje de crecimiento. `LAG(ingreso_mes) OVER (ORDER BY mes)` significa "dame el ingreso del renglón anterior cuando los ordeno por mes". Para el primer mes no hay anterior, así que esas columnas salen NULL: es normal y correcto, no hay con qué comparar.

El cálculo del porcentaje es la fórmula clásica: (actual menos anterior) entre anterior, por cien. Si el resultado es positivo, el mes creció; si es negativo, cayó. Con nuestros datos verás qué meses subieron y cuáles bajaron, y el dueño puede correlacionar: ¿en qué mes hicimos la promoción? ¿Qué pasó en el mes que cayó?

Este reporte introduce las funciones de ventana en un contexto de negocio real, y es el tipo de análisis que en las empresas se hace todas las semanas. Si dominas LAG para esto, también puedes calcular promedios móviles, rankings y acumulados. Pero esa ya es otra historia, y tienes las bases para explorarla.

Paso 20: verificación final del proyecto

Todo profesional verifica su trabajo antes de entregarlo. Estas consultas de control confirman que la tienda quedó íntegra: conteos por tabla, y la prueba más importante, que no existan registros huérfanos.

```
SELECT 'categorias' AS tabla, COUNT(*) AS total FROM categorias
UNION ALL
SELECT 'clientes', COUNT(*) FROM clientes
UNION ALL
SELECT 'productos', COUNT(*) FROM productos
UNION ALL
SELECT 'pedidos', COUNT(*) FROM pedidos
UNION ALL
SELECT 'detalle_pedido', COUNT(*) FROM detalle_pedido
UNION ALL
SELECT 'resenas', COUNT(*) FROM resenas;
```

Deberías ver 4 categorías, 8 clientes, 12 productos, 12 pedidos, 22 detalles y 6 reseñas. Si algún número no cuadra, revisa los INSERTs antes de seguir.

Ahora la prueba de integridad: buscar detalles que apunten a pedidos o productos inexistentes. Con las llaves foráneas bien definidas esto no debería pasar nunca, pero verificarlo es gratis y te da tranquilidad.

```
SELECT d.id_detalle
FROM detalle_pedido d
LEFT JOIN pedidos pe ON d.id_pedido = pe.id_pedido
WHERE pe.id_pedido IS NULL;

SELECT d.id_detalle
FROM detalle_pedido d
LEFT JOIN productos p ON d.id_producto = p.id_producto
WHERE p.id_producto IS NULL;
```

Ambas consultas deben devolver cero renglones. El patrón es el mismo LEFT JOIN con IS NULL de los reportes de inactivos: aquí lo usamos como auditoría. Si alguna vez devuelve algo, significa que la integridad se rompió por algún lado y hay que investigar.

Y una última verificación de negocio: el ingreso total de la tienda según los detalles debe coincidir con la suma de los totales por pedido. Si no coincide, hay un error en alguna consulta.

```
SELECT SUM(d.cantidad * d.precio_unitario) AS ingreso_total
FROM detalle_pedido d
INNER JOIN pedidos pe ON d.id_pedido = pe.id_pedido
WHERE pe.estado IN ('pagado', 'enviado');
```

Guarda ese número. Es el ingreso oficial de TiendaTech con estos datos de ejemplo. Cada reporte de ingresos que construyas debe cuadrar con él: si el reporte por mes, el reporte por categoría y el reporte por ciudad suman lo mismo, tus consultas están bien. Esa es la disciplina del profesional de datos: los números siempre deben cuadrar entre sí.

Práctica

La teoría sin práctica se olvida en una semana. Estos ejercicios extienden el proyecto de TiendaTech con funcionalidades nuevas que toda tienda real necesita. Intenta resolver cada uno por tu cuenta antes de leer la solución. Si te atorras, la solución está explicada paso a paso para que entiendas el razonamiento, no solo para que copies el código.

Ejercicio 1: agregar una tabla de reseñas

Toda tienda en línea que se respete tiene reseñas de productos: las estrellitas que ves antes de comprar. Diseña y crea la tabla `resenas` con estas reglas: cada reseña pertenece a un cliente y a un producto, lleva una calificación del 1 al 5, un comentario opcional y la fecha. Un cliente no puede reseñar dos veces el mismo producto. Después inserta al menos 5 reseñas de ejemplo y escribe una consulta que muestre el promedio de calificación por producto.

Solución paso a paso. Primero pensamos el diseño: `resenas` es una tabla intermedia entre clientes y productos, igual que `detalle_pedido` lo es entre pedidos y productos. Necesita llaves foráneas a ambas tablas. La calificación lleva un CHECK entre 1 y 5. Para impedir reseñas duplicadas del mismo cliente al mismo producto, usamos una restricción UNIQUE sobre la pareja (`id_cliente`, `id_producto`).

```

CREATE TABLE resenas (
  id_resena INT AUTO_INCREMENT PRIMARY KEY,
  id_cliente INT NOT NULL,
  id_producto INT NOT NULL,
  calificacion INT NOT NULL CHECK (calificacion BETWEEN 1 AND 5),
  comentario TEXT,
  fecha_resena DATE NOT NULL,
  CONSTRAINT fk_resenas_cliente
    FOREIGN KEY (id_cliente)
    REFERENCES clientes (id_cliente)
    ON DELETE CASCADE,
  CONSTRAINT fk_resenas_producto
    FOREIGN KEY (id_producto)
    REFERENCES productos (id_producto)
    ON DELETE CASCADE,
  CONSTRAINT uq_resena_cliente_producto
    UNIQUE (id_cliente, id_producto)
);

```

Usamos ON DELETE CASCADE en ambas foráneas: si se borra un cliente o un producto, sus reseñas desaparecen con ellos, porque una reseña huérfana no tiene sentido. Ahora insertamos datos de ejemplo:

```

INSERT INTO resenas (id_cliente, id_producto, calificacion, comentario,
fecha_resena) VALUES
  (1, 1, 5, 'Excelente laptop, rapidísima y ligera.', '2026-02-10'),
  (2, 1, 4, 'Muy buena, aunque la batería dura menos de lo esperado.',
'2026-06-20'),
  (1, 4, 5, 'El mouse es comodísimo, lo recomiendo.', '2026-02-10'),
  (3, 10, 5, 'Cancelan el ruido de maravilla.', '2026-03-01'),
  (4, 2, 3, 'Cumple para oficina, nada espectacular.', '2026-03-20'),
  (5, 3, 4, 'Buena tableta por el precio.', '2026-04-15');

```

Y la consulta del promedio por producto, con la cantidad de reseñas para dar contexto: un promedio de 5 con una sola reseña vale menos que un 4.5 con cincuenta.

```

SELECT
  p.nombre AS producto,
  COUNT(r.id_resena) AS num_resenas,
  ROUND(AVG(r.calificacion), 2) AS calificacion_promedio
FROM productos p
LEFT JOIN resenas r ON p.id_producto = r.id_producto
GROUP BY p.id_producto, p.nombre
HAVING COUNT(r.id_resena) > 0
ORDER BY calificacion_promedio DESC;

```

Usamos LEFT JOIN para no perder productos, pero el HAVING filtra los que no tienen reseñas: no tiene sentido mostrar un promedio vacío. El ROUND a dos decimales deja el promedio legible.

Ejercicio 2: reporte de ventas por ciudad

El dueño quiere saber en qué ciudades vende más, para decidir dónde abrir una tienda física o dónde enfocar la publicidad. Escribe una consulta que muestre por cada ciudad: número de clientes, número de pedidos válidos e ingreso total. Ordénala por ingreso descendente.

Solución paso a paso. Necesitamos clientes para la ciudad, pedidos para filtrar los válidos y detalle_pedido para el ingreso. El truco está en contar bien: los clientes se cuentan con COUNT(DISTINCT c.id_cliente) porque el JOIN con pedidos y detalles multiplica los renglones por cliente. Lo mismo con los pedidos.

```

SELECT
  c.ciudad,
  COUNT(DISTINCT c.id_cliente) AS num_clientes,
  COUNT(DISTINCT pe.id_pedido) AS num_pedidos,
  SUM(d.cantidad * d.precio_unitario) AS ingreso_total
FROM clientes c
INNER JOIN pedidos pe ON c.id_cliente = pe.id_cliente
INNER JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
WHERE pe.estado IN ('pagado', 'enviado')
GROUP BY c.ciudad
ORDER BY ingreso_total DESC;

```

Con nuestros datos, Ciudad de México debería liderar: Ana y Miguel son de ahí y entre los dos suman varios pedidos incluyendo laptops. Guadalajara aparece con Carlos, Monterrey con Lucía, Puebla con Sofía. Querétaro no aparece porque Valeria no ha comprado: el INNER JOIN la excluye. Si quisieras mostrar todas las ciudades aunque no tengan ventas, cambiarías

a LEFT JOINs y usarías COALESCE para los nulos. Piensa cuál versión le sirve más al dueño: la que muestra solo donde hay ventas, o la que también muestra dónde no hay nada que vender.

Ejercicio 3: productos que nunca se han vendido

Toda tienda tiene productos rezagados: ocupan espacio en el almacén y capital parado. Encuentra los productos que no aparecen en ningún detalle de pedido válido. Muestra su nombre, categoría, precio y existencia.

Solución paso a paso. Es el mismo patrón de clientes inactivos: LEFT JOIN más IS NULL. Traemos todos los productos y tratamos de unirlos con sus ventas en pedidos válidos. Los que no emparejan son los nunca vendidos.

```
SELECT
  p.nombre AS producto,
  cat.nombre AS categoria,
  p.precio,
  p.existencia
FROM productos p
INNER JOIN categorias cat ON p.id_categoria = cat.id_categoria
LEFT JOIN detalle_pedido d ON p.id_producto = d.id_producto
LEFT JOIN pedidos pe
  ON d.id_pedido = pe.id_pedido
  AND pe.estado IN ('pagado', 'enviado')
WHERE d.id_detalle IS NULL
ORDER BY p.existencia DESC;
```

El detalle importante: el filtro de estado va en el ON del LEFT JOIN con pedidos, no en el WHERE. Si un producto solo se vendió en un pedido cancelado, para efectos del negocio es como si no se hubiera vendido: el dinero nunca entró. Con nuestros datos, el teclado mecánico RGB es candidato: revisa los detalles y verás si aparece en algún pedido válido. El WHERE d.id_detalle IS NULL se queda con los productos sin ninguna venta válida. Ordenamos por existencia descendente para ver primero dónde hay más capital parado.

Ejercicio 4: clientes que compraron en todos los meses con ventas

Este es el ejercicio difícil, el de la liga mayor. Encuentra a los clientes que tienen al menos un pedido válido en cada mes donde la tienda registró ventas. Es el patrón de división relacional: "los que cumplen con todos".

Solución paso a paso. Primero necesitamos la lista de meses con ventas. Luego, para cada cliente, contamos en cuántos meses distintos compró. Si ese número iguala al total de meses con ventas, el cliente compró en todos.

```
SELECT
  c.nombre AS cliente,
  COUNT(DISTINCT DATE_FORMAT(pe.fecha_pedido, '%Y-%m')) AS meses_con_compra
FROM clientes c
INNER JOIN pedidos pe ON c.id_cliente = pe.id_cliente
WHERE pe.estado IN ('pagado', 'enviado')
GROUP BY c.id_cliente, c.nombre
HAVING meses_con_compra = (
  SELECT COUNT(DISTINCT DATE_FORMAT(fecha_pedido, '%Y-%m'))
  FROM pedidos
  WHERE estado IN ('pagado', 'enviado')
);
```

La subconsulta del HAVING cuenta cuántos meses distintos tienen ventas válidas: con nuestros datos son 7, de enero a julio de 2026. La consulta principal cuenta, por cliente, en cuántos meses distintos compró. El HAVING se queda con los que igualan el total. Con nuestros datos probablemente nadie cumpla: Ana compró en enero, febrero y abril, le faltan meses. Y está bien: un reporte que a veces sale vacío también es información. Le dice al dueño que ningún cliente compra con esa regularidad, y eso sugiere una oportunidad: un programa de suscripción o de compras recurrentes.

Si quieres una versión menos exigente, cambia el HAVING a `meses_con_compra >= 3` para encontrar clientes frecuentes. Los reportes se ajustan al negocio, no al revés.

Ejercicio 5: actualizar existencias después de una venta

En una tienda real, cuando se vende un producto, la existencia debe bajar. Hasta ahora nuestros INSERTs no tocaron la existencia. Escribe las sentencias para registrar un pedido nuevo del cliente 3 (Lucía): 2 bocinas portátiles y 1 micrófono USB, y actualiza las existencias correspondientes. Todo dentro de una transacción.

Solución paso a paso. Primero iniciamos la transacción para que todo ocurra junto o no ocurra nada: si algo falla a la mitad, no queremos un pedido sin detalles ni existencias a medias. Insertamos el pedido, capturamos su id, insertamos los detalles y restamos las existencias.

```

START TRANSACTION;

INSERT INTO pedidos (id_cliente, fecha_pedido, estado)
VALUES (3, '2026-07-20 12:00:00', 'pagado');

SET @nuevo_pedido = LAST_INSERT_ID();

INSERT INTO detalle_pedido (id_pedido, id_producto, cantidad, precio_unitario)
VALUES
    (@nuevo_pedido, 11, 2, 899.00),
    (@nuevo_pedido, 12, 1, 1399.00);

UPDATE productos SET existencia = existencia - 2 WHERE id_producto = 11;
UPDATE productos SET existencia = existencia - 1 WHERE id_producto = 12;

COMMIT;

```

`LAST_INSERT_ID()` nos da el id del pedido recién creado sin tener que adivinarlo: es la forma segura de encadenar inserts. Los `UPDATE` restan la cantidad vendida de la existencia. Gracias al `CHECK` (`existencia >= 0`), si alguien intenta vender más de lo que hay, la base rechaza el `UPDATE` y la transacción se revierte con `ROLLBACK`. Así es como una tienda real evita vender productos que no tiene: la base de datos lo impide, no la buena voluntad del programador.

Verifica el resultado: la bocina portátil (id 11) debía quedar en 53 y el micrófono USB (id 12) en 2. El micrófono ahora está en zona crítica de stock: debería aparecer primero en el reporte de bajo stock. Todo conectado, como en un negocio de verdad.

Ejercicio 6: vista de resumen de pedidos para el panel del dueño

El dueño quiere un panel que muestre cada pedido con su cliente, fecha, estado, número de productos distintos y total. Crea una vista llamada `vista_resumen_pedidos` que calcule todo eso, y luego úsala para mostrar los 5 pedidos más recientes.

Solución paso a paso. Una vista guarda la consulta compleja una vez y la reutiliza como si fuera una tabla. Agrupamos por pedido y calculamos sus métricas.

```
CREATE VIEW vista_resumen_pedidos AS
SELECT
  pe.id_pedido,
  c.nombre AS cliente,
  c.ciudad,
  pe.fecha_pedido,
  pe.estado,
  COUNT(DISTINCT d.id_producto) AS productos_distintos,
  SUM(d.cantidad) AS unidades_totales,
  SUM(d.cantidad * d.precio_unitario) AS total_pedido
FROM pedidos pe
INNER JOIN clientes c ON pe.id_cliente = c.id_cliente
LEFT JOIN detalle_pedido d ON pe.id_pedido = d.id_pedido
GROUP BY
  pe.id_pedido, c.nombre, c.ciudad,
  pe.fecha_pedido, pe.estado;
```

Usamos LEFT JOIN con el detalle porque un pedido recién creado podría no tener detalles todavía: con INNER JOIN desaparecería del panel y el dueño pensaría que no existe. Ahora el panel es trivial:

```
SELECT *
FROM vista_resumen_pedidos
ORDER BY fecha_pedido DESC
LIMIT 5;
```

La vista encapsula la complejidad: quien consulta el panel no necesita saber de JOINS ni de agregados. Esa es la gracia de las vistas que aprendiste en su unidad: simplifican el acceso y protegen la lógica. Si mañana cambia la forma de calcular el total, se modifica la vista en un solo lugar y todos los reportes que la usan se actualizan solos.

Resumen de la unidad

Llegaste al final de la unidad más práctica del curso. Repasemos lo que construiste, porque es mucho.

Partiste de requerimientos en lenguaje de negocio: qué vende la tienda, quiénes compran, cómo son los pedidos, qué quiere saber el dueño. Convertiste esas respuestas en un modelo entidad-relación de cinco tablas: clientes, categorías, productos, pedidos y detalle_pedido. Aprendiste que las relaciones de muchos a muchos no existen directamente en el modelo relacional: se resuelven con una tabla intermedia, y detalle_pedido es el ejemplo perfecto.

Aplicaste normalización de verdad, no de examen: viste por qué la primera forma normal prohíbe listas en una celda, por qué la segunda exige que cada columna dependa de toda la llave, y por qué la tercera elimina las dependencias transitivas como el nombre de la categoría repetido en productos. Cada dato tiene un solo hogar.

Tomaste decisiones de diseño profesionales: llaves subrogadas autoincrementales en vez de datos de negocio como llaves, DECIMAL para dinero en vez de FLOAT, CHECKs para estados y valores válidos, comportamientos ON DELETE pensados según el negocio (RESTRICT para proteger la historia, CASCADE para los detalles huérfanos), e índices en las columnas que los reportes usan: fechas, foráneas y correos.

Poblaste la tienda con datos realistas e intencionales: clientes sin compras para probar inactivos, stock casi agotado para probar el inventario, un pedido cancelado para probar que los reportes lo excluyen. Los datos de prueba bien diseñados son la mitad del trabajo.

Y construiste siete reportes de negocio que responden preguntas reales: top 5 de productos, ventas por mes, clientes inactivos con el patrón LEFT JOIN e IS NULL, ticket promedio con subconsulta, bajo stock, mejor cliente y categorías por ingreso con subconsulta escalar para porcentajes. Cada reporte combina las técnicas de las unidades anteriores con un propósito.

En la práctica extendiste el proyecto: tabla de reseñas con restricción UNIQUE compuesta, ventas por ciudad, productos nunca vendidos, el reto de división relacional de clientes en todos los meses, una transacción completa de venta con actualización de existencias, y una vista para el panel del dueño.

Lo más importante no es el código, es la mentalidad: primero entender el negocio, luego diseñar el modelo, después poblar con intención y al final preguntar. SQL es la herramienta; las preguntas de negocio son el trabajo. Cuando alguien te pida la base de datos de su negocio, ya sabes por dónde empezar: con preguntas, no con CREATE TABLE.

Un último consejo práctico: guarda este proyecto. Súbelo a un repositorio, documenta el modelo con el texto del análisis de requerimientos, y tenlo a la mano. Cuando en una entrevista te pregunten "¿sabes SQL?", no respondas con teoría: muestra TiendaTech. Muestra el modelo, los reportes y las decisiones de diseño. Un proyecto completo y bien explicado vale más que cien ejercicios sueltos, porque demuestra lo que las empresas realmente compran: criterio.

En la siguiente unidad cerraremos el curso con buenas prácticas, seguridad y el camino a seguir como profesional de datos. Pero eso ya es la graduación. Por hoy, presume tu tienda: la construiste tú, de cero, y funciona.

UNIDAD 10: GRADUACIÓN: CONSOLIDACIÓN Y SIGUIENTE PASO

Esteban - lucasegaacademy.com

Llegaste al final.

Detente un momento. Mira hacia atrás. Hace diez unidades no sabías que era una base de datos, o al menos no sabías como pedirle las cosas. Hoy sabes diseñar tablas, relacionarlas, hacer consultas con filtros, ordenar, agrupar, sumar, promediar, unir tablas con JOIN, anidar consultas, insertar, actualizar y borrar datos sin destruir la base. Eso no es cualquier cosa. Eso es una habilidad profesional real, de las que pagan sueldo.

Esta unidad es tu ceremonia de graduación. No vamos a aprender un tema nuevo y complicado, vamos a hacer algo mejor: vamos a amarrar todo lo que aprendiste, a ver el mapa completo del curso desde arriba, a corregir los errores que más cometen los programadores (para que tu no los cometas), a conocer las buenas prácticas que separan a un aficionado de un profesional, y a mirar hacia adelante: cómo conectar todo esto con una aplicación real usando Node.js, y como convertir lo que sabes en trabajo.

Respira hondo. Esto es el repaso final antes de graduarte como alguien que domina SQL.

Cómo leer esta unidad

Las unidades anteriores te enseñaron una pieza a la vez. Esta unidad te muestra el rompecabezas armado.

Primero vas a ver el mapa mental completo del curso: un recorrido unidad por unidad de todo lo que aprendiste, para que veas como cada parte encaja con las demás. Después vienen los 15 errores más comunes en SQL: la lista negra que todo programador consulta alguna vez, para que aprendas de los tropiezos de otros antes de tropezarte tu. Luego, las buenas prácticas profesionales: como nombrar tus cosas, como comentar tu código, como respaldar tu trabajo, cómo usar transacciones para no romper nada, y una probadita de usuarios y permisos.

Después miramos hacia adelante: que puedes aprender después (funciones de fecha, procedimientos almacenados, optimización con EXPLAIN), y la puerta más emocionante de todas, cómo conectar tu base de datos MySQL con Node.js para construir aplicaciones completas. Esa conexión es lo que separa un curso de SQL de un camino full stack.

Terminamos con consejos prácticos para conseguir trabajo con SQL: portafolio, que piden las empresas, y certificaciones.

Y claro, hay práctica. Cinco ejercicios integradores que combinan todo lo del curso, estilo mini-proyectos reales: una biblioteca, un inventario, un gimnasio. Cuando los termines, vas a saber que ya no necesitas que te enseñen SQL paso a paso. Ya puedes aprender solo.

Tu viaje hasta aquí: el mapa mental del curso

Antes de repasar, vamos a celebrar el camino. Piensa en cada unidad como un nivel de un videojuego que ya pasaste.

En la UNIDAD 1 entendiste la idea más importante de todas: una base de datos es un almacén organizado de información, y una tabla es como una hoja de Excel superpoderosa con filas y columnas. Aprendiste que SQL es el idioma para hablar con esas tablas, y corriste tus primeras consultas SELECT. Sin eso, nada de lo demás existiría.

En la UNIDAD 2 te metiste a la cocina: CREATE DATABASE y CREATE TABLE. Aprendiste a diseñar tablas desde cero, a elegir tipos de datos (INT, VARCHAR, DATE, DECIMAL), y a decidir cuáles columnas no pueden quedar vacías. Aquí empezó tu carrera de arquitecto de datos.

En la UNIDAD 3 conociste las reglas del juego: PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, CHECK, DEFAULT. Las restricciones. Aprendiste que una buena base de datos se protege sola: si alguien intenta meter un dato inválido, la base dice que no. Eso es diseño profesional.

En la UNIDAD 4 poblaste tus tablas: INSERT para agregar filas, UPDATE para corregirlas, DELETE para eliminarlas. Y aprendiste la lección más valiosa de esa unidad: nunca uses UPDATE o DELETE sin WHERE, porque borras o cambias todo. Esa lección te va a salvar el empleo algún día.

En la UNIDAD 5 te volviste selectivo: WHERE con comparaciones, AND, OR, NOT, BETWEEN, IN, LIKE con comodines, IS NULL. Aprendiste a pedir exactamente los datos que necesitas, ni más ni menos. Filtrar es el 80 por ciento del trabajo diario con SQL.

En la UNIDAD 6 ordenaste el caos: ORDER BY para ordenar, LIMIT para paginar, DISTINCT para quitar duplicados, y alias con AS para que tus resultados se lean bonito. Aquí tus consultas empezaron a verse profesionales.

En la UNIDAD 7 aprendiste el superpoder de los números: COUNT, SUM, AVG, MIN, MAX, GROUP BY para agrupar y HAVING para filtrar grupos. De pronto podías responder preguntas de negocio: cuanto vendimos, cual es el promedio, cuantos clientes hay por ciudad. Eso es lo que las empresas pagan.

En la UNIDAD 8 conectaste el mundo: JOIN. INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, y los self join. Aprendiste que las tablas relacionadas se consultan juntas, y que elegir el tipo de JOIN correcto cambia completamente el resultado. Sin JOIN, tus tablas serian islas incomunicadas.

En la UNIDAD 9 subiste de nivel: subconsultas (consultas dentro de consultas), UNION para apilar resultados, vistas con CREATE VIEW, e índices con CREATE INDEX para que todo corra rápido. Aquí dejaste de ser usuario de SQL y empezaste a pensar como ingeniero de datos.

Y ahora, la UNIDAD 10: la graduación. El repaso, los errores que debes evitar, las buenas prácticas, el siguiente paso con Node.js, y el camino al trabajo.

Fíjate en algo: cada unidad dependía de la anterior. No puedes hacer JOIN si no sabes crear tablas relacionadas. No puedes usar GROUP BY si no filtras con WHERE. No puedes optimizar con índices si no entiendes cómo se consultan los datos. Construiste una torre, piso por piso. Y la torre esta completa.

Los 15 errores más comunes en SQL (y como evitarlos)

Esta es la sección más valiosa de la unidad. Estos errores los cometen todos: principiantes, intermedios, y hasta programadores con años de experiencia en un día de prisa. Conocelos ahora y te ahorraras horas de dolor de cabeza.

ERROR 1: Usar SELECT * en código de producción.

El asterisco trae todas las columnas. En tus prácticas está bien, es rápido y comodo. Pero en una aplicación real es peligroso: si alguien agrega una columna pesada a la tabla (una foto, un texto larguísimo), tu consulta de pronto trae megabytes de más. Además, si el orden de las columnas cambia, tu código se rompe.

Como evitarlo: lista siempre las columnas que necesitas.

En lugar de SELECT * FROM clientes, escribe SELECT id, nombre, email FROM clientes. Es más largo de escribir, pero más rápido, más seguro y más claro.

ERROR 2: UPDATE o DELETE sin WHERE.

Este es el error clásico, el que cuenta en las historias de terror. Un UPDATE sin WHERE actualiza TODAS las filas. Un DELETE sin WHERE borra TODA la tabla. En segundos.

Como evitarlo: antes de correr un UPDATE o DELETE, corre primero un SELECT con el mismo WHERE para ver cuantas filas vas a afectar. Si el SELECT trae 3 filas, tu UPDATE va a cambiar 3 filas. Si trae 3000, detente y revisa.

Habitualo como ritual: SELECT primero, UPDATE después.

ERROR 3: Comparar NULL con el signo igual.

NULL no es un valor, es la ausencia de valor. Por eso NULL = NULL no es verdadero, y column = NULL nunca trae nada. Es una de las trampas más confusas para principiantes.

Como evitarlo: usa IS NULL e IS NOT NULL. Siempre. Sin excepciones.

ERROR 4: Olvidar que los strings van entre comillas simples.

Los números van sin comillas: WHERE edad = 25. Los textos van con comillas simples: WHERE nombre = 'Ana'. Si pones WHERE nombre = Ana sin comillas, la base cree que Ana es una columna y te marca error.

Como evitarlo: texto entre comillas simples, números sin comillas, fechas entre comillas simples también: WHERE fecha = '2026-01-15'.

ERROR 5: Confundir WHERE con HAVING.

WHERE filtra filas ANTES de agrupar. HAVING filtra grupos DESPUÉS de agrupar. Si intentas usar una función de agregado en el WHERE (WHERE COUNT(*) > 5), te marca error. Los agregados solo viven en HAVING o en el SELECT.

Como evitarlo: preguntate, estoy filtrando filas individuales o grupos ya agrupados. Filas: WHERE. Grupos: HAVING.

ERROR 6: No poner alias claros en los JOIN.

Cuando unes dos tablas que tienen columnas con el mismo nombre (las dos tienen id, por ejemplo), y no especificas de cual tabla viene cada columna, la base se confunde y te marca error de ambigüedad. O peor: no te marca error y trae la columna equivocada.

Como evitarlo: usa alias cortos para las tablas (c para clientes, p para pedidos) y antepone el alias a cada columna: SELECT c.nombre, p.fecha FROM clientes c JOIN pedidos p ON c.id = p.cliente_id.

ERROR 7: Hacer JOIN sin condición ON (producto cartesiano accidental).

Si haces FROM clientes, pedidos sin condición de union, obtienes cada cliente combinado con cada pedido. Con 1000 clientes y 10000 pedidos, eso son 10 millones de filas. Tu consulta se congela, el servidor sufre, y tu jefe te mira feo.

Como evitarlo: todo JOIN lleva su ON. Si usas la sintaxis con comas, la condición va en el WHERE. Revisa siempre que la condición de union exista.

ERROR 8: Usar LIKE '%texto' al inicio cuando la tabla es enorme.

El comodín % al principio impide que la base use índices, así que tiene que leer la tabla completa fila por fila. En una tabla pequeña no se nota. En una tabla con millones de filas, la consulta tarda una eternidad.

Como evitarlo: si puedes, evita el % al inicio. LIKE 'texto%' si puede usar índice. Y si de verdad necesitas buscar dentro del texto en tablas grandes, investiga las búsquedas de texto completo (FULLTEXT), que es tema avanzado pero existe para eso.

ERROR 9: Guardar varios valores en una sola columna.

La columna telefonos con el valor "555-1234, 555-5678" parece práctica, pero rompe todo: no puedes buscar un teléfono individual fácil, no puedes contar cuantos hay, no puedes validar el formato. Es una violación a la primera forma normal que viste en diseño.

Como evitarlo: un dato por columna, un dato por fila. Si un cliente tiene varios telefonos, crea una tabla telefonos con una fila por teléfono.

ERROR 10: No usar transacciones para operaciones que deben ir juntas.

Si tu aplicación hace dos INSERT relacionados (el pedido y sus detalles) y el segundo falla, te quedas con un pedido sin detalles: datos rotos. Las transacciones existen exactamente para eso.

Como evitarlo: envuelve operaciones relacionadas en START TRANSACTION y COMMIT, con ROLLBACK si algo falla. Lo verás en detalle en los ejemplos de esta unidad.

ERROR 11: Guardar contraseñas en texto plano.

Nunca, jamás, bajo ninguna circunstancia guardes una contraseña tal como el usuario la escribió. Si alguien roba tu base de datos, tiene todas las contraseñas. Es el error más grave de seguridad de está lista.

Como evitarlo: guarda el hash de la contraseña (el resultado de pasarla por una función como bcrypt o SHA-256 con sal). Cuando el usuario inicia sesión, calculas el hash de lo que escribió y lo comparas con el guardado. Así ni tú mismo puedes ver las contraseñas reales.

ERROR 12: No hacer respaldos (o hacerlos y nunca probarlos).

El respaldo que nunca probaste restaurar es como el extintor que nunca revisaste: el día que lo necesitas, descubres que no sirve. Discos fallan, consultas malas borran datos, servidores se apagan.

Como evitarlo: programa respaldos automaticos con mysqldump (verás el comando en los ejemplos) y de vez en cuando restaura uno en una base de prueba para confirmar que funciona.

ERROR 13: Darle a la aplicación un usuario con todos los permisos.

Si tu aplicación se conecta a la base con el usuario root o con un usuario que puede borrar tablas, y alguien logra inyectar SQL malicioso, puede destruir todo. Es como darle la llave maestra del edificio al repartidor.

Como evitarlo: crea un usuario específico para la aplicación con solo los permisos que necesita (SELECT, INSERT, UPDATE en ciertas tablas). Lo verás en la sección de buenas prácticas.

ERROR 14: Ignorar los tipos de datos y guardar números como texto.

Guardar precios en VARCHAR parece inofensivo hasta que intentas sumarlos y la base los ordena como texto (donde "100" va antes que "20", porque compara carácter por carácter). O hasta que alguien guarda "caro" en la columna de precio y todo se rompe.

Como evitarlo: elige el tipo de dato correcto desde el diseño: INT para enteros, DECIMAL para dinero, DATE para fechas, VARCHAR para texto. La base te protege solo si le das las herramientas.

ERROR 15: No probar las consultas con datos reales (o sin datos de prueba).

Una consulta que funciona con 10 filas de prueba puede tardar minutos con 10 millones de filas reales. O puede traer resultados inesperados porque los datos reales tienen casos raros que no imaginaste (nombres con apostrofes, fechas futuras, valores NULL donde jurabas que no habría).

Como evitarlo: prueba con volúmenes parecidos a los reales cuando puedas, y mete casos raros en tus datos de prueba a propósito: NULL, textos vacíos, números negativos, fechas extremas. Si tu consulta sobrevive a los casos raros, sobrevivirá a la realidad.

Guía de buenas prácticas profesional

Saber SQL es una cosa. Usarlo como un profesional es otra. Estas prácticas son las que separan al que "sabe hacer consultas" del que las empresas quieren contratar. No son difíciles, son hábitos.

Convenciones de nombres: que todo se entienda sin explicación

Cuando otra persona (o tú mismo dentro de seis meses) abra tu base de datos, los nombres deben explicarse solos.

Usa minúsculas y guiones bajos para tablas y columnas: clientes, pedidos, fecha_nacimiento. Evita espacios, mayúsculas mezcladas y caracteres raros. Una tabla llamada "Tbl_Clientes 2024" es una pesadilla; una llamada clientes es un alivio.

Nombra las tablas en plural o singular, pero se consistente. Si usas clientes, usa pedidos y productos. Si usas cliente, usa pedido y producto. Mezclar los dos estilos en la misma base es cómo escribir un libro cambiando de idioma a mitad de página.

Las llaves primarias suelen llamarse id, simple y universal. Las llaves foráneas deben dejar claro a que apuntan: cliente_id en la tabla pedidos deja clarísimo que apunta al id de clientes. Nada de id_cliente en una tabla y clienteid en otra.

Evita palabras reservadas como nombres: order, group, user, table. La palabra user es especialmente traicionera porque casi todas las aplicaciones tienen tabla de usuarios. Si la llamas user, cada consulta necesitara comillas especiales. Llamala usuarios y duerme tranquilo.

Comenta tu código SQL

El SQL también se comenta, y el SQL complejo lo necesita desesperadamente. Una consulta con tres JOIN y una subconsulta es obvia hoy que la escribiste; en tres meses será un jeroglífico.

Usa -- para comentarios de una línea y /* */ para bloques. Explica el porqué, no el QUE. El código ya dice que hace; el comentario debe decir por qué lo hace así.

Un buen comentario: -- Excluye pedidos cancelados porque el reporte es solo de ventas efectivas. Un mal comentario: -- Selecciona los pedidos. El segundo no aporta nada.

Respalda siempre, automatiza el respaldo

Tu base de datos es el activo más valioso de cualquier aplicación. El código se puede reescribir; los datos de dos años de clientes, no.

La regla de oro: automatiza el respaldo diario y guarda copias en más de un lugar. Un respaldo en el mismo disco que la base de datos no te salva si el disco muere. Ideal: respaldo diario automatico más copia semanal en otro servidor o en la nube.

En los ejemplos de esta unidad verás el comando `mysqldump`, la herramienta clásica de MySQL para respaldos. Aprendetelo de memoria.

Transacciones: todo o nada

Imagina que tu aplicación transfiere dinero entre cuentas: resta 500 de la cuenta A y suma 500 a la cuenta B. Son dos UPDATE. Si el primero funciona y el segundo falla (se cae el servidor a la mitad), el dinero desaparece. 500 pesos que ya no están en A ni en B. Eso no puede pasar.

Las transacciones resuelven esto con una promesa: o se hacen TODAS las operaciones, o no se hace NINGUNA. Es el principio de "todo o nada".

Cómo funciona: empiezas con `START TRANSACTION` (o `BEGIN`). Ejecutas tus operaciones normalmente. Si todo salió bien, cierras con `COMMIT` y los cambios se guardan para siempre. Si algo salió mal, ejecutas `ROLLBACK` y la base deshace todo lo que hiciste desde el `START TRANSACTION`, como si nada hubiera pasado.

La mayoría de las bases de datos trabajan en "autocommit": cada sentencia se guarda automáticamente al ejecutarse. Las transacciones te permiten agrupar varias sentencias en un solo paquete atómico. Usalas siempre que varias operaciones dependan entre sí: transferencias, pedidos con sus detalles, registros que tocan varias tablas a la vez.

Verás ejemplos completos en la sección de ejemplos paso a paso.

Usuarios y permisos: el principio del menor privilegio

En MySQL puedes crear usuarios distintos con permisos distintos. La regla profesional se llama "principio del menor privilegio": cada usuario debe tener exactamente los permisos que necesita y ni uno más.

El usuario root puede hacer TODO: crear bases, borrar tablas, crear usuarios. Tu aplicación web no necesita todo eso. Si un atacante logra colarse por tu aplicación, heredará los permisos del usuario con el que se conecta. Menos permisos, menos dano posible.

En la práctica: crea un usuario para tu aplicación con permiso de SELECT, INSERT, UPDATE y DELETE solo en las tablas que usa, y nada más. Guarda otro usuario administrador (con más permisos) solo para tareas de mantenimiento que haces tu manualmente.

La sintaxis básica es CREATE USER para crear el usuario y GRANT para darle permisos. Por ejemplo, darle solo lectura a un usuario en una base: GRANT SELECT ON tienda.* TO 'reportes'@'localhost'. Y si un día ya no lo necesitas: REVOKE o DROP USER.

No necesitas memorizar toda la sintaxis de permisos hoy. Lo importante es el concepto: nunca conectes tu aplicación con root, y dale a cada quien lo mínimo necesario.

Otras prácticas que te hacen ver profesional

Valida los datos en la aplicación ANTES de mandarlos a la base, aunque la base también tenga restricciones. Dos capas de protección valen más que una.

Nunca construyas consultas pegando texto que escribió el usuario directamente en el SQL. Eso abre la puerta a la inyección SQL, el ataque más famoso contra bases de datos. Usa consultas parametrizadas (con ? como marcadores), donde la base trata el dato del usuario como dato y nunca como código. Cuando veas el ejemplo de Node.js más adelante, fíjate en los signos de interrogación: esos son parámetros, y son tu escudo.

Documenta tu diseño: un diagrama simple de tus tablas y sus relaciones vale oro cuando el proyecto crece. No tiene que ser fancy; un dibujo con las tablas y las flechas de las llaves foráneas basta.

Y mide antes de optimizar: no agregues índices "por si acaso" a todas las columnas. Los índices aceleran las lecturas pero hacen más lentas las escrituras. Agrega índices donde las consultas lentas los pidan, y usa EXPLAIN para comprobarlo (lo verás en los ejemplos).

Como seguir aprendiendo después de este curso

Graduarte no significa dejar de aprender. SQL es un mundo enorme y lo que viste aquí es la base sólida. Esto es lo que sigue, en orden de utilidad.

Funciones de fecha y hora

Las fechas están en todas partes: cuando se registro el usuario, cuando se hizo el pedido, cuantos días pasaron. MySQL tiene un arsenal de funciones: NOW() para la fecha y hora actual, CURDATE() para solo la fecha, DATEDIFF() para restar fechas, DATE_ADD() para sumar días o meses, YEAR(), MONTH(), DAY() para extraer partes, y DATE_FORMAT() para mostrarlas bonito.

Con esto puedes responder preguntas como: ventas del último mes, usuarios registrados esta semana, productos proximos a caducar. Es de lo primero que te pidiran en un trabajo real.

Procedimientos almacenados y funciones

Un procedimiento almacenado es un bloque de SQL guardado dentro de la base de datos, con nombre propio, que puedes ejecutar cuando quieras con CALL. Es como una función de programación pero viviendo en la base.

Sirven para encapsular lógica compleja que se repite: en lugar de que cada aplicación repita los mismos 40 renglones de SQL, los guardas una vez en la base y todos los llaman. También sirven para operaciones delicadas donde quieres controlar exactamente que se hace.

No son lo primero que necesitas, pero cuando trabajes con bases grandes los vas a encontrar. Saber que existen y cómo se ven te pone adelante.

Optimización con EXPLAIN

Cuando una consulta tarda demasiado, EXPLAIN es tu rayo X. Le pones EXPLAIN delante de cualquier SELECT y MySQL te muestra su plan: en que orden lee las tablas, si usa índices o lee todo, cuantas filas estima revisar.

Aprender a leer un EXPLAIN básico (verás un ejemplo en esta unidad) te convierte en la persona que resuelve los problemas de lentitud. Eso vale mucho en cualquier equipo.

Temas avanzados para después

En el camino largo te esperan: triggers (código que se ejecuta solo cuando pasa algo en una tabla), vistas materializadas, particionado de tablas gigantes, replicación (copias de la base para aguantar más carga), y diseño de data warehouses para análisis masivo. No te preocupes por ellos ahora; con lo que sabes puedes trabajar, y estos temas llegan solos cuando los necesitas.

El siguiente paso: conectar MySQL con Node.js

Aquí esta la puerta al siguiente nivel. Hasta ahora, tu has sido el que le habla a la base de datos directamente: escribes SQL, la base responde. Pero en el mundo real, casi nadie le habla a la base a mano. Las aplicaciones lo hacen por ti.

Piensa en una tienda en línea. El usuario hace clic en "comprar". El programa (escrito en JavaScript con Node.js, por ejemplo) recibe ese clic, y es EL PROGRAMA el que le manda a MySQL algo como `INSERT INTO pedidos...`. La base guarda el pedido y le responde al programa "listo". El programa le muestra al usuario "gracias por tu compra". Tu SQL sigue ahí, trabajando, pero ahora lo ejecuta código en lugar de tus manos.

Que es un driver

Tu programa de Node.js y tu base MySQL hablan idiomas distintos. El driver es el traductor: una librería que instalas en tu proyecto y que sabe como conectarse a MySQL, mandarle consultas y traerte los resultados de vuelta a JavaScript.

Para MySQL en Node.js, el driver más usado se llama `mysql2`. Lo instalas una vez con `npm install mysql2` y ya puedes conectar.

Cómo se ve la conexión (ejemplo conceptual)

Mira este ejemplo. No necesitas entender cada línea a fondo todavía; es una probadita, no un curso de Node.js. Solo observa la idea general: conectar, preguntar, recibir.

```

const mysql = require('mysql2/promise');

async function main() {
  const conexion = await mysql.createConnection({
    host: 'localhost',
    user: 'tienda_app',
    password: 'clave_segura',
    database: 'tienda'
  });

  const [filas] = await conexion.execute(
    'SELECT id, nombre, precio FROM productos WHERE precio < ?',
    [500]
  );

  console.log(filas);
  await conexion.end();
}

main();

```

Fíjate en tres cosas. Primera: la conexión lleva host, usuario, contraseña y base de datos, como cuando entras a MySQL desde la terminal. Segunda: la consulta SQL va dentro del código JavaScript como texto, y tu SQL de este curso sirve tal cual. Tercera: el signo ? es un parámetro: el valor 500 se manda por separado, y el driver lo coloca de forma segura. Así se evita la inyección SQL de la que hablamos en buenas prácticas.

El resultado llega como un arreglo de objetos JavaScript, uno por fila, que tu programa puede mostrar en una página web, mandar a una app móvil o lo que sea. Eso es el camino full stack: tu base de datos atrás, tu código en medio, el usuario enfrente. Y tu ya dominas la parte de atrás.

Si este tema te emociona, el siguiente curso natural es JavaScript y luego Node.js, donde aprenderas a construir esas aplicaciones completas. Todo lo que aprendiste aquí te va a servir tal cual.

Consejos para conseguir trabajo con SQL

Tienes la habilidad. Ahora hay que convertirla en oportunidades. Esto es lo que funciona en el mundo real.

Arma un portafolio con proyectos reales

Los reclutadores no contratan certificados, contratan evidencia. Un portafolio con 3 o 4 proyectos bien hechos vale más que cualquier papel.

Que poner: los mini-proyectos de la sección de práctica de esta unidad (biblioteca, inventario, gimnasio) son un excelente inicio. Para cada proyecto incluye el script SQL completo: creación de tablas, datos de ejemplo, y las consultas más interesantes con una breve explicación de que resuelven. Subelo a GitHub, que es gratis y es donde los reclutadores miran.

Un plus enorme: si además conectas uno de esos proyectos con una pequeña aplicación (aunque sea simple), demuestras el camino full stack. Una página que muestre los productos de tu inventario sacados de MySQL impresiona muchísimo más que solo scripts.

Que piden las empresas

Para puestos junior donde SQL es requisito (analista de datos, desarrollador backend junior, soporte de bases de datos), lo que más piden es exactamente lo de este curso: SELECT con WHERE, JOIN de varios tipos, GROUP BY con HAVING, subconsultas, INSERT/UPDATE/DELETE, y nociones de diseño de tablas.

En las entrevistas técnicas es común que te pongan a escribir consultas en vivo o en un editor: "dame los 5 clientes que más compraron", "encuentra los productos sin ventas este mes". Los ejercicios de práctica de esta unidad son entrenamiento directo para eso. Práctica escribir consultas sin copiar, de memoria, contra reloj. La velocidad y la seguridad impresionan.

También valoran: saber lo de NULL (muchos reprueban ahí), entender la diferencia entre WHERE y HAVING, y poder explicar un diseño de tablas en voz alta. Si puedes dibujar tres tablas relacionadas y explicar por qué las relacionaste así, ya estás arriba del promedio.

Certificaciones: útiles, pero no mágicas

Existen certificaciones de MySQL (de Oracle), de Microsoft (para SQL Server) y de otros proveedores. Ayudan a que tu currículum pase los filtros automáticos y demuestran disciplina. Pero ninguna certificación sustituye saber resolver problemas reales.

Mi consejo: primero el portafolio y la práctica, después la certificación si la quieres. Un GitHub con proyectos buenos más una certificación es una combinación muy fuerte. Solo la certificación, sin nada que mostrar, es una combinación débil.

Donde buscar y como presentarte

LinkedIn, Indeed, OCCMundial y Computrabajo son los lugares obvios. Pero el truco real es este: en tu perfil y curriculum, no pongas "se SQL". Pon lo que puedes HACER con SQL: "diseno y consulto bases de datos relacionales: modelado de tablas, consultas complejas con JOIN y agregados, optimización básica". Los verbos de acción venden; las etiquetas no.

Y práctica contar tu historia: "tome un curso completo de SQL donde construi estos proyectos..." y muestras el GitHub. Eso es una entrevista que empieza bien.

Explicación detallada: cierre del mapa

Ya tienes el mapa completo, los errores que debes evitar, las buenas prácticas, el camino para seguir aprendiendo, la puerta a Node.js y la ruta al trabajo. Antes de pasar a los ejemplos, graba estas tres ideas.

Primera: SQL es un idioma de preguntas, no de instrucciones paso a paso. Tu describes QUE quieres (dame los clientes que compraron más de 1000) y la base decide COMO conseguirlo. Esa es su magia y por eso EXPLAIN existe: para espiar como lo hace cuando algo va lento.

Segunda: el diseno manda. Una base bien disenada (tablas normalizadas, llaves correctas, tipos de datos adecuados) hace que todo lo demás sea fácil. Una base mal disenada convierte cada consulta en una lucha. El tiempo que inviertes en disenar se paga solo cien veces.

Tercera: la práctica es el verdadero maestro. Este curso te dio las herramientas; los ejercicios de práctica te dan las horas de vuelo. Un piloto no aprende volando solo en simulador: tu no aprendes SQL solo leyendo. Escribe, rompe cosas en tu base de prueba, arreglalas. Así se aprende de verdad.

Ahora si, manos a la obra con los ejemplos.

Chuleta rápida: todo el SQL del curso en una página

Cuando estes trabajando y necesites recordar la sintaxis, vuelve aquí. Esta es tu hoja de referencia.

Crear y borrar estructuras:

```
CREATE DATABASE tienda;
USE tienda;
CREATE TABLE clientes (id INT PRIMARY KEY AUTO_INCREMENT, nombre VARCHAR(100) NOT NULL);
ALTER TABLE clientes ADD COLUMN telefono VARCHAR(20);
DROP TABLE clientes;
```

Meter, cambiar y borrar datos:

```
INSERT INTO clientes (nombre, email) VALUES ('Ana', 'ana@mail.com');
UPDATE clientes SET email = 'nuevo@mail.com' WHERE id = 1;
DELETE FROM clientes WHERE id = 1;
```

Consultar y filtrar:

```
SELECT id, nombre FROM clientes WHERE ciudad = 'CDMX' AND edad >= 18;
SELECT * FROM clientes WHERE nombre LIKE 'A%' OR email IS NULL;
SELECT DISTINCT ciudad FROM clientes;
SELECT * FROM clientes ORDER BY nombre ASC LIMIT 10;
```

Agrupar y agregar:

```
SELECT ciudad, COUNT(*) AS total, AVG(edad) AS promedio
FROM clientes
GROUP BY ciudad
HAVING COUNT(*) > 5;
```

Unir tablas:

```
SELECT c.nombre, p.fecha FROM clientes c
INNER JOIN pedidos p ON c.id = p.cliente_id;
```

Vistas e índices:

```
CREATE VIEW vista_resumen AS SELECT ... ;
CREATE INDEX idx_email ON clientes(email);
```

Transacciones:

```
START TRANSACTION;
-- tus operaciones
COMMIT;
-- o ROLLBACK si algo fallo
```

Respaldo en terminal:

```
mysqldump -u root -p tienda > respaldo.sql
```

Pegala en tu pared. En serio.

Preguntas típicas de entrevista (con respuestas cortas)

Práctica responder estas en voz alta. Si las dominas, dominas la entrevista junior.

Qué es una llave primaria? Un identificador único para cada fila de una tabla. No se repite y no puede ser NULL.

Qué es una llave foránea? Una columna que apunta a la llave primaria de otra tabla. Es lo que crea la relación entre tablas.

Diferencia entre WHERE y HAVING? WHERE filtra filas antes de agrupar. HAVING filtra grupos después de agrupar con GROUP BY.

Por qué NULL no se compara con =? Porque NULL no es un valor, es ausencia de valor. Se usa IS NULL e IS NOT NULL.

Diferencia entre INNER JOIN y LEFT JOIN? INNER JOIN trae solo las filas que tienen pareja en ambas tablas. LEFT JOIN trae todas las de la izquierda aunque no tengan pareja (las de la derecha quedan en NULL).

Qué es una transacción? Un paquete de operaciones que se hacen todas o ninguna: START TRANSACTION, COMMIT para guardar, ROLLBACK para deshacer.

Qué hace un índice? Acelera las búsquedas y los JOIN, a cambio de hacer las escrituras un poco más lentas.

Qué es la inyección SQL y cómo se evita? Es meter código SQL malicioso donde la aplicación esperaba un dato. Se evita con consultas parametrizadas (el ? en lugar de pegar texto).

Qué es una vista? Una consulta guardada con nombre que se usa como si fuera una tabla.

¿Qué es la normalización? Organizar las tablas para evitar datos duplicados y dependencias raras: un dato, un lugar.

Si respondes estas nueve con seguridad y escribes las consultas de práctica sin temblar, la entrevista es tuya.

Ejemplos paso a paso

Ejemplo 1: La consulta "final boss" (repaso integral)

Esta consulta combina casi todo el curso en una sola sentencia: JOIN, GROUP BY, HAVING, ORDER BY y LIMIT. Es el tipo de pregunta que hacen los negocios reales y las entrevistas de trabajo.

El escenario: una tienda en línea quiere saber quienes son sus 5 mejores clientes del último año: solo clientes con más de 3 pedidos y un total comprado mayor a 5000, ordenados del que más gasto al que menos.

Primero, las tablas (ya las conoces de unidades anteriores):

```
CREATE TABLE clientes (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL  
);  
  
CREATE TABLE pedidos (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  cliente_id INT NOT NULL,  
  fecha DATE NOT NULL,  
  total DECIMAL(10,2) NOT NULL,  
  FOREIGN KEY (cliente_id) REFERENCES clientes(id)  
);
```

Ahora la consulta final:

```

SELECT
  c.nombre AS cliente,
  c.email,
  COUNT(p.id) AS num_pedidos,
  SUM(p.total) AS total_gastado,
  AVG(p.total) AS ticket_promedio
FROM clientes c
INNER JOIN pedidos p ON c.id = p.cliente_id
WHERE p.fecha >= '2025-01-01'
GROUP BY c.id, c.nombre, c.email
HAVING COUNT(p.id) > 3 AND SUM(p.total) > 5000
ORDER BY total_gastado DESC
LIMIT 5;

```

Vamos parte por parte, porque cada línea usa algo distinto del curso.

FROM clientes c INNER JOIN pedidos p ON c.id = p.cliente_id: unimos clientes con sus pedidos (UNIDAD 8). El alias c y p mantiene todo legible. Usamos INNER JOIN porque solo nos interesan clientes que SI tienen pedidos.

WHERE p.fecha >= '2025-01-01': filtramos filas individuales antes de agrupar (UNIDAD 5). Solo pedidos de este año.

GROUP BY c.id, c.nombre, c.email: agrupamos por cliente (UNIDAD 7). Agrupamos por id para que cada cliente sea un grupo único, y agregamos nombre y email porque los mostramos en el SELECT.

Las funciones de agregado: COUNT(p.id) cuenta los pedidos de cada cliente, SUM(p.total) suma lo que gasto, AVG(p.total) calcula su ticket promedio. Todo UNIDAD 7.

HAVING COUNT(p.id) > 3 AND SUM(p.total) > 5000: filtramos los grupos ya agrupados (UNIDAD 7). Solo clientes con más de 3 pedidos Y más de 5000 gastados. Aquí no serviría WHERE porque WHERE no entiende COUNT ni SUM.

ORDER BY total_gastado DESC: ordenamos del que más gasto al que menos (UNIDAD 6). Fíjate que ordenamos por el alias definido en el SELECT: MySQL lo permite y queda muy legible.

LIMIT 5: nos quedamos con los primeros 5 (UNIDAD 6). Los 5 mejores, ni uno más.

Si entendiste cada línea de esta consulta, entendiste el curso. Leela otra vez despacio y felicitate.

Ejemplo 2: Transacciones basicas (todo o nada)

Escenario: la tienda transfiere 500 pesos de saldo entre dos cuentas de clientes (quizás un reembolso interno). Son dos UPDATE que DEBEN ir juntos.

Sin transacción, si el servidor se cae entre el primer y el segundo UPDATE, el dinero se pierde. Con transacción, eso es imposible.

```
START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 500 WHERE id = 1;

UPDATE cuentas SET saldo = saldo + 500 WHERE id = 2;

COMMIT;
```

Que paso aquí: START TRANSACTION abre el paquete. Los dos UPDATE se ejecutan pero sus cambios quedan "pendientes", visibles solo dentro de esta sesión. COMMIT cierra el paquete y guarda todo junto, de forma atomica. Si entre los dos UPDATE algo falla, en lugar de COMMIT ejecutas ROLLBACK y el primer UPDATE se deshace como si nunca hubiera pasado.

La versión con manejo de error se ve así:

```
START TRANSACTION;

UPDATE cuentas SET saldo = saldo - 500 WHERE id = 1;

-- Supongamos que aqui detectamos un problema:
-- la cuenta 2 no existe, o el saldo quedaria negativo.
-- En lugar de seguir, deshacemos todo:

ROLLBACK;
```

Regla práctica: si una operación toca más de una tabla (o más de una fila donde la mitad seria un desastre), envuélvela en transacción. Pedidos con sus detalles, transferencias, registros compuestos: todo eso va en transacción.

Ojo: en MySQL las transacciones funcionan con el motor InnoDB, que es el que se usa por defecto en las versiones modernas. Si alguna vez ves una tabla con motor MyISAM (muy raro hoy), las transacciones no aplican ahí.

Ejemplo 3: EXPLAIN básico (el rayo X de tus consultas)

Cuando una consulta tarda demasiado, no adivines: investiga. EXPLAIN te muestra el plan que MySQL va a seguir.

```
EXPLAIN SELECT c.nombre, p.fecha, p.total
FROM clientes c
INNER JOIN pedidos p ON c.id = p.cliente_id
WHERE p.fecha >= '2026-01-01'
ORDER BY p.fecha DESC;
```

El resultado es una tablita con columnas como estas (los nombres exactos varían un poco según la versión):

La columna "type" es la más importante para empezar. Si dice ALL, significa que MySQL va a leer la tabla COMPLETA fila por fila: eso es lento en tablas grandes y es la señal de que probablemente falta un índice. Si dice ref o range, está usando un índice: buena señal. Si dice const, encontró la fila directamente: excelente.

La columna "rows" estima cuántas filas va a revisar. Mientras más pequeño, mejor. Si dice 1 millón en una tabla de 1 millón, está leyendo todo.

La columna "key" te dice QUÉ índice está usando. Si dice NULL, no está usando ninguno.

El flujo profesional es: consulta lenta, EXPLAIN, veo un ALL donde debería haber un índice, creo el índice con CREATE INDEX, vuelvo a correr EXPLAIN, confirmo que ahora dice ref, y mido el tiempo de nuevo. Así se optimiza con evidencia, no con superstición.

Ejemplo de antes y después:

```
-- Lento: busca pedidos por fecha sin índice
EXPLAIN SELECT * FROM pedidos WHERE fecha >= '2026-01-01';
-- type: ALL, rows: 1000000 (lee todo)

CREATE INDEX idx_pedidos_fecha ON pedidos(fecha);

EXPLAIN SELECT * FROM pedidos WHERE fecha >= '2026-01-01';
-- type: range, key: idx_pedidos_fecha, rows: 50000 (usa el índice)
```

Esa es la diferencia entre una consulta que tarda 8 segundos y una que tarda 0.2. Y todo empezó con un EXPLAIN.

Ejemplo 4: Respaldo con mysqldump (tu paracaidas)

mysqldump no se corre dentro de MySQL: se corre en la terminal del sistema, y genera un archivo .sql con todo lo necesario para reconstruir tu base (tablas, datos, todo).

Respaldo completo de una base de datos:

```
mysqldump -u root -p tienda > respaldo_tienda_2026-09-19.sql
```

Desglose: mysqldump es el programa. -u root dice con que usuario te conectas. -p te pide la contraseña (la escribes cuando te la pida, no se muestra en pantalla). tienda es la base a respaldar. El > guarda la salida en el archivo respaldo_tienda_2026-09-19.sql.

Ponle fecha al nombre del archivo. Tu yo del futuro te lo va a agradecer cuando tengas 40 respaldos y necesites el del martes.

Respaldo de una sola tabla (cuando solo te importa una):

```
mysqldump -u root -p tienda clientes > respaldo_clientes.sql
```

Y para restaurar un respaldo (en una base de prueba, para verificar que funciona, o en producción si algo salio mal):

```
mysql -u root -p tienda_restaurada < respaldo_tienda_2026-09-19.sql
```

Fíjate que para restaurar se usa mysql (el cliente normal), no mysqldump, y el < mete el archivo hacia adentro en lugar de sacarlo.

Automatizarlo es simple: ese comando en una tarea programada (cron en Linux) que corra cada noche, y listo. Respaldo diario sin que tengas que acordarte. Y una vez al mes, restaura uno en una base de prueba para confirmar que el paracaidas abre.

Práctica

Aquí es donde te graduas de verdad. Cinco ejercicios integradores, estilo mini-proyectos reales. Cada uno combina diseño de tablas, inserción de datos y consultas de todo el curso. Hazlos en tu base de prueba, sin mirar las soluciones primero. Las soluciones están explicadas paso a paso para que compares y aprendas.

Ejercicio 1: Sistema de biblioteca

Disena una base para una biblioteca pequena. Necesita tabla de libros (id, título, autor, año, generó, ejemplares disponibles), tabla de socios (id, nombre, email, fecha de registro) y tabla de prestamos (id, libro_id, socio_id, fecha de prestamo, fecha de devolución, donde la devolución puede ser NULL si aun no lo regresan).

Consultas a resolver: a) Libros que están prestados actualmente (sin devolver). b) Los 3 socios con más prestamos en total. c) Generó con más prestamos historicos. d) Libros que nunca han sido prestados.

Solución explicada:

Primero el diseno. Tres tablas, dos relaciones:

```
CREATE TABLE libros (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  titulo VARCHAR(150) NOT NULL,  
  autor VARCHAR(100) NOT NULL,  
  anio INT,  
  genero VARCHAR(50),  
  ejemplares INT NOT NULL DEFAULT 1  
);  
  
CREATE TABLE socios (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL,  
  fecha_registro DATE NOT NULL  
);  
  
CREATE TABLE prestamos (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  libro_id INT NOT NULL,  
  socio_id INT NOT NULL,  
  fecha_prestamo DATE NOT NULL,  
  fecha_devolucion DATE,  
  FOREIGN KEY (libro_id) REFERENCES libros(id),  
  FOREIGN KEY (socio_id) REFERENCES socios(id)  
);
```

Fíjate en fecha_devolucion sin NOT NULL: puede ser NULL porque un libro prestado aun no se devuelve. Ese NULL es información, no un error de diseno.

a) Libros prestados actualmente: prestamos donde la devolución es NULL. Aquí se usa IS NULL, el error 3 de la lista negra.

```
SELECT l.titulo, l.autor, s.nombre AS socio, p.fecha_prestamo
FROM prestamos p
INNER JOIN libros l ON p.libro_id = l.id
INNER JOIN socios s ON p.socio_id = s.id
WHERE p.fecha_devolucion IS NULL;
```

Dos JOIN en una consulta (libros y socios), filtro con IS NULL. Todo lo del curso en cuatro líneas.

b) Los 3 socios con más prestamos: agrupar por socio, contar, ordenar, limitar.

```
SELECT s.nombre, COUNT(p.id) AS total_prestamos
FROM socios s
INNER JOIN prestamos p ON s.id = p.socio_id
GROUP BY s.id, s.nombre
ORDER BY total_prestamos DESC
LIMIT 3;
```

c) Generó con más prestamos: igual pero agrupando por generó del libro, con un JOIN extra.

```
SELECT l.genero, COUNT(p.id) AS total_prestamos
FROM libros l
INNER JOIN prestamos p ON l.id = p.libro_id
GROUP BY l.genero
ORDER BY total_prestamos DESC
LIMIT 1;
```

d) Libros nunca prestados: aquí brilla el LEFT JOIN. Unimos libros con prestamos y nos quedamos con los que NO tienen pareja.

```
SELECT l.titulo, l.autor
FROM libros l
LEFT JOIN prestamos p ON l.id = p.libro_id
WHERE p.id IS NULL;
```

El truco: LEFT JOIN trae todos los libros aunque no tengan prestamos; donde no hay prestamo, las columnas de p quedan en NULL. El WHERE p.id IS NULL filtra justo esos. Este patrón (LEFT JOIN + IS NULL) es "encuentra los que no tienen relación" y aparece en entrevistas todo el tiempo.

Ejercicio 2: Control de inventario

Una tienda necesita: productos (id, nombre, categoría, precio, stock), proveedores (id, nombre, teléfono), y una tabla que relacione que proveedor surte que producto (un producto puede tener varios proveedores y un proveedor surte varios productos: relación muchos a muchos).

Consultas: a) Productos con stock menor a 10 (alerta de resurtido), con nombre del proveedor. b) Valor total del inventario por categoría (precio por stock, sumado). c) Proveedores que no surten ningún producto actualmente. d) Actualiza el stock después de una venta usando transacción.

Solución explicada:

La relación muchos a muchos necesita tabla intermedia. Es el diseño clásico:

```
CREATE TABLE productos (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  categoria VARCHAR(50),  
  precio DECIMAL(10,2) NOT NULL,  
  stock INT NOT NULL DEFAULT 0  
);  
  
CREATE TABLE proveedores (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  telefono VARCHAR(20)  
);  
  
CREATE TABLE producto_proveedor (  
  producto_id INT NOT NULL,  
  proveedor_id INT NOT NULL,  
  PRIMARY KEY (producto_id, proveedor_id),  
  FOREIGN KEY (producto_id) REFERENCES productos(id),  
  FOREIGN KEY (proveedor_id) REFERENCES proveedores(id)  
);
```

La tabla intermedia tiene llave primaria compuesta: la pareja producto-proveedor no se repite. Así se modela "muchos a muchos" en el mundo real.

a) Alerta de resurtido con proveedor:

```
SELECT pr.nombre AS producto, pr.stock, pv.nombre AS proveedor, pv.telefono  
FROM productos pr  
INNER JOIN producto_proveedor pp ON pr.id = pp.producto_id  
INNER JOIN proveedores pv ON pp.proveedor_id = pv.id  
WHERE pr.stock < 10  
ORDER BY pr.stock ASC;
```

Doble JOIN pasando por la tabla intermedia. Orden ascendente para ver primero los más críticos.

b) Valor del inventario por categoría:

```
SELECT categoria, SUM(precio * stock) AS valor_inventario  
FROM productos  
GROUP BY categoria  
ORDER BY valor_inventario DESC;
```

precio * stock se calcula por fila y SUM lo suma por grupo. Los calculos dentro de agregados son pan de cada día en reportes.

c) Proveedores sin productos: otra vez el patrón LEFT JOIN + IS NULL.

```
SELECT pv.nombre, pv.telefono  
FROM proveedores pv  
LEFT JOIN producto_proveedor pp ON pv.id = pp.proveedor_id  
WHERE pp.producto_id IS NULL;
```

d) Venta con transacción: vender 3 unidades del producto 5. Hay que bajar el stock Y registrar la venta; si una falla, la otra no debe pasar.

```
START TRANSACTION;  
  
UPDATE productos SET stock = stock - 3 WHERE id = 5;  
  
INSERT INTO ventas (producto_id, cantidad, fecha) VALUES (5, 3, CURDATE());  
  
COMMIT;
```

Si el INSERT falla, ROLLBACK y el stock vuelve a su valor. Todo o nada, como debe ser.

Ejercicio 3: Registro de gimnasio

Un gimnasio necesita: miembros (id, nombre, email, fecha de ingreso, membresia que puede ser básica, premium o vip), clases (id, nombre, instructor, capacidad), e inscripciones (que miembro va a que clase y en que fecha).

Consultas: a) Clases llenas (inscritos igual a la capacidad) para la próxima semana. b) Miembros premium o vip que no se han inscrito a ninguna clase este mes. c) Instructor con más clases impartidas y su promedio de asistencia. d) Borra las inscripciones de hace más de un año, con respaldo previo.

Solución explicada:

```
CREATE TABLE miembros (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL,  
  fecha_ingreso DATE NOT NULL,  
  membresia VARCHAR(20) NOT NULL DEFAULT 'basica'  
);  
  
CREATE TABLE clases (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  instructor VARCHAR(100) NOT NULL,  
  capacidad INT NOT NULL  
);  
  
CREATE TABLE inscripciones (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  miembro_id INT NOT NULL,  
  clase_id INT NOT NULL,  
  fecha DATE NOT NULL,  
  FOREIGN KEY (miembro_id) REFERENCES miembros(id),  
  FOREIGN KEY (clase_id) REFERENCES clases(id)  
);
```

a) Clases llenas: agrupamos inscripciones por clase, contamos, y comparamos con la capacidad usando HAVING (porque comparamos un agregado).

```

SELECT c.nombre, c.instructor, c.capacidad, COUNT(i.id) AS inscritos
FROM clases c
INNER JOIN inscripciones i ON c.id = i.clase_id
WHERE i.fecha BETWEEN '2026-09-21' AND '2026-09-27'
GROUP BY c.id, c.nombre, c.instructor, c.capacidad
HAVING COUNT(i.id) >= c.capacidad;

```

HAVING comparando COUNT con una columna (capacidad): se puede, HAVING filtra grupos y puede usar cualquier columna del grupo.

b) Miembros premium/vip sin clases este mes: combinamos IN con LEFT JOIN + IS NULL.

```

SELECT m.nombre, m.email, m.membresia
FROM miembros m
LEFT JOIN inscripciones i
  ON m.id = i.miembro_id
  AND i.fecha >= '2026-09-01'
WHERE m.membresia IN ('premium', 'vip')
  AND i.id IS NULL;

```

El truco fino: la condición de fecha va en el ON del LEFT JOIN, no en el WHERE. Si la pones en el WHERE, conviertes el LEFT JOIN en INNER JOIN sin querer y el truco se rompe. Así filtras las inscripciones del mes ANTES de buscar las que faltan.

c) Instructor estrella:

```

SELECT c.instructor,
       COUNT(DISTINCT c.id) AS clases_impartidas,
       COUNT(i.id) AS total_asistentes,
       ROUND(AVG(1.0 * 1), 1) AS nota
FROM clases c
LEFT JOIN inscripciones i ON c.id = i.clase_id
GROUP BY c.instructor
ORDER BY clases_impartidas DESC
LIMIT 1;

```

Mejor versión, con promedio de asistencia por clase calculado con subconsulta:

```

SELECT instructor,
       COUNT(*) AS clases_impartidas,
       AVG(asistentes) AS promedio_asistencia
FROM (
  SELECT c.instructor, c.id, COUNT(i.id) AS asistentes
  FROM clases c
  LEFT JOIN inscripciones i ON c.id = i.clase_id
  GROUP BY c.instructor, c.id
) AS resumen
GROUP BY instructor
ORDER BY clases_impartidas DESC
LIMIT 1;

```

La subconsulta interna calcula asistentes por cada clase; la externa agrupa por instructor. Subconsultas en acción (UNIDAD 9).

d) Limpieza con respaldo: primero respaldo de lo que vas a borrar (buena práctica: nunca borres sin poder recuperar).

```

-- Paso 1: respaldo en la terminal
-- mysqldump -u root -p gimnasio inscripciones > respaldo_inscripciones.sql

-- Paso 2: verifica cuantas filas vas a borrar
SELECT COUNT(*) FROM inscripciones WHERE fecha < '2025-09-19';

-- Paso 3: borra con WHERE (nunca sin WHERE)
DELETE FROM inscripciones WHERE fecha < '2025-09-19';

```

El ritual completo: respaldo, SELECT de verificación, DELETE con WHERE. Así borra un profesional.

Ejercicio 4: Mini red social (integrador total)

Este es el más ambicioso: combina todo. Tablas: usuarios (id, nombre, email, fecha_registro), publicaciones (id, usuario_id, texto, fecha), comentarios (id, publicacion_id, usuario_id, texto, fecha), y likes (usuario_id, publicacion_id, fecha) como tabla intermedia.

Consultas: a) Las 5 publicaciones con más likes, mostrando autor y número de likes y comentarios. b) Usuarios que nunca han publicado nada. c) Promedio de publicaciones por usuario (solo usuarios con al menos una). d) Encuentra publicaciones sin comentarios usando subconsulta.

Solución explicada:

```
CREATE TABLE usuarios (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  nombre VARCHAR(100) NOT NULL,  
  email VARCHAR(100) UNIQUE NOT NULL,  
  fecha_registro DATE NOT NULL  
);  
  
CREATE TABLE publicaciones (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  usuario_id INT NOT NULL,  
  texto VARCHAR(500) NOT NULL,  
  fecha DATETIME NOT NULL,  
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id)  
);  
  
CREATE TABLE comentarios (  
  id INT PRIMARY KEY AUTO_INCREMENT,  
  publicacion_id INT NOT NULL,  
  usuario_id INT NOT NULL,  
  texto VARCHAR(500) NOT NULL,  
  fecha DATETIME NOT NULL,  
  FOREIGN KEY (publicacion_id) REFERENCES publicaciones(id),  
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id)  
);  
  
CREATE TABLE likes (  
  usuario_id INT NOT NULL,  
  publicacion_id INT NOT NULL,  
  fecha DATETIME NOT NULL,  
  PRIMARY KEY (usuario_id, publicacion_id),  
  FOREIGN KEY (usuario_id) REFERENCES usuarios(id),  
  FOREIGN KEY (publicacion_id) REFERENCES publicaciones(id)  
);
```

La tabla likes es muchos a muchos con llave compuesta: un usuario no puede dar like dos veces a lo mismo. Diseño limpio.

a) Top 5 publicaciones:

```

SELECT p.id,
       u.nombre AS autor,
       p.texto,
       COUNT(DISTINCT l.usuario_id) AS num_likes,
       COUNT(DISTINCT c.id) AS num_comentarios
FROM publicaciones p
INNER JOIN usuarios u ON p.usuario_id = u.id
LEFT JOIN likes l ON p.id = l.publicacion_id
LEFT JOIN comentarios c ON p.id = c.publicacion_id
GROUP BY p.id, u.nombre, p.texto
ORDER BY num_likes DESC
LIMIT 5;

```

Ojo al detalle: COUNT(DISTINCT ...). Como unimos likes y comentarios a la vez, las filas se multiplican (cada like se combina con cada comentario). Sin DISTINCT, el conteo saldría inflado. Este es un error sutil y muy común en consultas con múltiples LEFT JOIN: cuando cuentas, usa DISTINCT.

b) Usuarios sin publicaciones: el patrón de siempre.

```

SELECT u.nombre, u.email
FROM usuarios u
LEFT JOIN publicaciones p ON u.id = p.usuario_id
WHERE p.id IS NULL;

```

c) Promedio de publicaciones por usuario activo:

```

SELECT AVG(num_pubs) AS promedio_por_usuario
FROM (
  SELECT usuario_id, COUNT(*) AS num_pubs
  FROM publicaciones
  GROUP BY usuario_id
) AS conteo;

```

La subconsulta cuenta publicaciones por usuario; la externa promedia esos conteos. Dos niveles de agregado, uno dentro del otro.

d) Publicaciones sin comentarios, con subconsulta y NOT IN:

```
SELECT p.id, p.texto, u.nombre AS autor
FROM publicaciones p
INNER JOIN usuarios u ON p.usuario_id = u.id
WHERE p.id NOT IN (SELECT publicacion_id FROM comentarios);
```

La subconsulta trae los ids de publicaciones que SI tienen comentarios; el NOT IN se queda con las que no están en esa lista. Alternativa equivalente con LEFT JOIN + IS NULL (que ya dominas). Dos caminos, mismo resultado: eso es SQL.

Ejercicio 5: El examen final (tienda completa)

Último ejercicio. Una tienda con clientes, productos, pedidos y detalle_pedidos (cada pedido tiene varios productos con cantidad y precio). Resuelve:

a) Ingreso total por mes en 2026. b) Los 3 productos más vendidos (por cantidad) con su ingreso generado. c) Clientes que compraron en todos los meses del primer trimestre (enero, febrero y marzo). d) Crea una vista con el resumen mensual de ventas para no repetir la consulta.

Solución explicada:

```

CREATE TABLE clientes (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  ciudad VARCHAR(50)
);

CREATE TABLE productos (
  id INT PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(100) NOT NULL,
  precio DECIMAL(10,2) NOT NULL,
  stock INT NOT NULL DEFAULT 0
);

CREATE TABLE pedidos (
  id INT PRIMARY KEY AUTO_INCREMENT,
  cliente_id INT NOT NULL,
  fecha DATE NOT NULL,
  FOREIGN KEY (cliente_id) REFERENCES clientes(id)
);

CREATE TABLE detalle_pedidos (
  pedido_id INT NOT NULL,
  producto_id INT NOT NULL,
  cantidad INT NOT NULL,
  precio_unitario DECIMAL(10,2) NOT NULL,
  PRIMARY KEY (pedido_id, producto_id),
  FOREIGN KEY (pedido_id) REFERENCES pedidos(id),
  FOREIGN KEY (producto_id) REFERENCES productos(id)
);

```

Nota de diseño: detalle_pedidos guarda precio_unitario aunque productos ya tiene precio. Por qué duplicar? Porque los precios cambian con el tiempo, y el detalle debe reflejar lo que se cobro ESE día, no el precio actual. Eso es pensar como profesional.

a) Ingreso por mes:

```

SELECT MONTH(p.fecha) AS mes,
       SUM(d.cantidad * d.precio_unitario) AS ingreso
FROM pedidos p
INNER JOIN detalle_pedidos d ON p.id = d.pedido_id
WHERE YEAR(p.fecha) = 2026
GROUP BY MONTH(p.fecha)
ORDER BY mes;

```

MONTH() extrae el mes de la fecha (funciones de fecha, de lo que sigue después del curso). Agrupamos por mes y sumamos cantidad por precio.

b) Top 3 productos:

```

SELECT pr.nombre,
       SUM(d.cantidad) AS unidades_vendidas,
       SUM(d.cantidad * d.precio_unitario) AS ingreso_generado
FROM productos pr
INNER JOIN detalle_pedidos d ON pr.id = d.producto_id
GROUP BY pr.id, pr.nombre
ORDER BY unidades_vendidas DESC
LIMIT 3;

```

c) Clientes fieles del trimestre: compraron en enero Y febrero Y marzo. La técnica es contar meses distintos por cliente.

```

SELECT c.nombre, c.email
FROM clientes c
INNER JOIN pedidos p ON c.id = p.cliente_id
WHERE p.fecha BETWEEN '2026-01-01' AND '2026-03-31'
GROUP BY c.id, c.nombre, c.email
HAVING COUNT(DISTINCT MONTH(p.fecha)) = 3;

```

COUNT(DISTINCT MONTH(fecha)) cuenta en cuantos meses distintos compró cada cliente. Si son 3, compró en los tres. Elegante y típico de entrevista.

d) La vista mensual (UNIDAD 9): guarda la consulta del inciso a) para reusarla.

```
CREATE VIEW resumen_mensual_ventas AS
SELECT MONTH(p.fecha) AS mes,
       YEAR(p.fecha) AS anio,
       COUNT(DISTINCT p.id) AS num_pedidos,
       SUM(d.cantidad * d.precio_unitario) AS ingreso
FROM pedidos p
INNER JOIN detalle_pedidos d ON p.id = d.pedido_id
GROUP BY YEAR(p.fecha), MONTH(p.fecha);
```

Y ahora usarla es trivial:

```
SELECT * FROM resumen_mensual_ventas WHERE anio = 2026 ORDER BY mes;
```

La vista esconde la complejidad: quien la usa no necesita saber de los JOIN internos. Eso es encapsular, y es de profesionales.

Si resolviste estos cinco ejercicios (o al menos entendiste cada solución línea por línea), estas listo. En serio.

Tu plan de 7 días para no olvidar nada

El conocimiento que no se usa se oxida. Aquí tienes un plan de una semana, una hora diaria, para que SQL se quede contigo para siempre. Hazlo justo después de terminar el curso, cuando todo esta fresco.

DÍA 1: Diseño desde cero. Inventar un negocio (taquería, papelería, taller mecánico) y diseñar su base completa en papel: tablas, columnas, tipos de datos, llaves primarias y foráneas. Luego crearla en MySQL. El objetivo es que diseñar te salga natural, sin dudar en los tipos de datos.

DÍA 2: Poblar y ensuciar. Meter al menos 50 filas repartidas en tus tablas, incluyendo casos raros a propósito: nombres con apostrofes (como D'Angelo), emails duplicados para probar el UNIQUE, fechas futuras, algún NULL donde este permitido. Una base con datos aburridos no te enseña nada; una con datos tramposos te enseña todo.

DÍA 3: Preguntas de negocio. Escribe 20 preguntas que el dueño del negocio haría ("cuál fue mi mejor día de ventas", "que producto se vende menos", "cuantos clientes nuevos llegaron este mes") y responderlas todas con consultas. Este es el día más importante: es exactamente lo que harás en un trabajo.

DÍA 4: El día de los JOIN difíciles. Escribe consultas que unan 3 o más tablas, práctica el patrón LEFT JOIN + IS NULL ("que no tiene..."), y resuelve todo con subconsultas y luego otra vez sin subconsultas. Compara cual se lee mejor. La fluidez con JOIN es lo que más distingue a un buen consultor SQL.

DÍA 5: Rompe y arregla. A proposito: corre un UPDATE sin WHERE en tu base de prueba y observa el desastre (en prueba, siempre en prueba). Luego práctica restaurarlo desde un respaldo que hiciste con mysqldump. Haz una transacción y un ROLLBACK. Corre un EXPLAIN a tu consulta más lenta y agregale un índice. Hoy aprendes con las manos, no con los ojos.

DÍA 6: Proyecto portafolio. Elige uno de los mini-proyectos de la práctica (biblioteca, inventario, gimnasio) y dejalos presentable: script SQL limpio y comentado, datos de ejemplo, y un archivo README corto explicando que resuelve. Subelo a GitHub. Hoy construyes la primera pieza de tu portafolio.

DÍA 7: Simulacro de entrevista. Pídele a alguien (o a una IA) que te haga las 9 preguntas de la sección anterior y que te pida escribir 5 consultas en vivo, con tiempo. Habla en voz alta mientras escribes: explica qué haces y por qué. En las entrevistas reales, pensar en voz alta vale tanto como la respuesta correcta.

Si completas los 7 días, SQL deja de ser "algo que estudiaste" y pasa a ser "algo que sabes hacer". Esa es la diferencia que notan los empleadores.

Glosario final del curso

Tu diccionario de bolsillo. Si algun terminó se te olvida, aquí esta.

Agregado: calculo sobre un grupo de filas, como COUNT, SUM, AVG, MIN, MAX.

Alias: nombre temporal para una tabla o columna en una consulta, con AS. Hace el SQL más corto y legible.

Autocommit: modo donde cada sentencia SQL se guarda automáticamente al ejecutarse. Las transacciones lo pausan temporalmente.

Base de datos: conjunto organizado de datos guardados en tablas relacionadas.

Cardinalidad: cuantos de cada lado participan en una relación: uno a uno, uno a muchos, muchos a muchos.

Check: restricción que obliga a que los valores cumplan una condición, como precio > 0.

Clave foránea (FOREIGN KEY): columna que referencia la llave primaria de otra tabla. Crea la relación.

Clave primaria (PRIMARY KEY): identificador único de cada fila. No se repite, no es NULL.

Commit: confirma una transacción y guarda sus cambios para siempre.

Consulta (query): petición de datos escrita en SQL.

Default: valor automatico que se usa cuando no se indica uno al insertar.

Delete: sentencia para borrar filas. Siempre con WHERE.

Driver: libreria que permite a un programa conectarse a una base de datos, como mysql2 para Node.js.

Explain: comando que muestra el plan de ejecución de una consulta. Tu rayo X para optimizar.

Grant: sentencia para dar permisos a un usuario de la base de datos.

Group by: agrupa filas con valores iguales para aplicarles funciones de agregado.

Having: filtra grupos ya agrupados por GROUP BY. El WHERE de los agregados.

Índice: estructura que acelera búsquedas y JOIN, creada con CREATE INDEX.

Inner join: une tablas trayendo solo las filas con pareja en ambas.

Insert: sentencia para agregar filas nuevas.

Inyeccion SQL: ataque que mete código SQL malicioso donde se esperaba un dato. Se evita con parámetros.

Join: operación que combina filas de dos o más tablas según una condición.

Left join: trae todas las filas de la tabla izquierda, aunque no tengan pareja.

Like: operador para buscar patrones en texto, con % y _ como comodines.

Limit: restringe cuantos resultados trae una consulta. Ideal para paginar.

Mysqldump: herramienta de terminal para respaldar bases MySQL en archivos .sql.

Normalizacion: técnica para organizar tablas evitando duplicados y dependencias incorrectas.

Not null: restricción que prohíbe valores vacios en una columna.

Null: ausencia de valor. No es cero ni texto vacío. Se compara con IS NULL.

Order by: ordena los resultados de forma ascendente (ASC) o descendente (DESC).

Rollback: deshace todos los cambios de una transacción en curso.

Select: sentencia para consultar datos. La más usada de todas.

Subconsulta: consulta anidada dentro de otra consulta.

Tabla: estructura de filas y columnas donde viven los datos.

Transacción: paquete atómico de operaciones: todas se hacen o ninguna.

Unique: restricción que impide valores duplicados en una columna.

Update: sentencia para modificar filas existentes. Siempre con WHERE.

Vista (view): consulta guardada con nombre que se usa como tabla virtual.

Where: filtra filas individuales antes de agrupar u ordenar.

Resumen de la unidad

Repasemos lo esencial de tu graduación.

Recorriste el mapa completo del curso: de que es una base de datos (unidad 1) hasta subconsultas, vistas e índices (unidad 9). Cada unidad construyo sobre la anterior, y ahora ves la torre completa.

Conociste los 15 errores más comunes: SELECT * en producción, UPDATE/DELETE sin WHERE, comparar NULL con =, confundir WHERE con HAVING, JOIN sin ON, guardar varios valores en una columna, no usar transacciones, contraseñas en texto plano, no respaldar, permisos de más, tipos de datos incorrectos, y no probar con datos reales. Evitar estos errores ya te pone por encima de muchos programadores.

Aprendiste buenas prácticas profesionales: nombres claros y consistentes, comentar el porqué, respaldos automaticos, transacciones para operaciones atómicas, y usuarios con el menor privilegio posible. Más el escudo contra inyección SQL: consultas parametrizadas, siempre.

Viste el camino hacia adelante: funciones de fecha, procedimientos almacenados, y EXPLAIN para optimizar. Y abriste la puerta más grande: conectar MySQL con Node.js usando el driver mysql2, donde tu SQL vive dentro del código de una aplicación real. Ese es el camino full stack.

Recibiste consejos de carrera: portafolio en GitHub con proyectos reales, lo que piden las empresas en entrevistas, como hablar de tus habilidades con verbos de acción, y el lugar de las certificaciones.

Practicaste con ejemplos integrales: la consulta "final boss" con JOIN, GROUP BY, HAVING, ORDER BY y LIMIT; transacciones con START TRANSACTION, COMMIT y ROLLBACK; EXPLAIN para diagnosticar lentitud; y mysqldump para respaldar y restaurar.

Y te graduaste con cinco mini-proyectos: biblioteca, inventario, gimnasio, red social y tienda completa, cada uno con soluciones explicadas paso a paso.

Cierre: lo que sigue es tuyo

Cuando empezaste este curso, SQL era un misterio. Ahora es una herramienta en tus manos. Sabes diseñar, poblar, consultar, relacionar, agrupar, proteger y respaldar bases de datos. Sabes lo que no debes hacer y sabes cómo se trabaja como profesional.

Pero aquí va la verdad que ningún curso te puede dar hecha: el conocimiento se vuelve habilidad solo con práctica. Los ejercicios de esta unidad son tu gimnasio. Vuelve a ellos. Inventa tus propios proyectos: la base de datos de tu colección de música, del negocio de un familiar, de tus gastos del mes. Cada base que diseñes te hace mejor.

Y cuando estes listo para el siguiente nivel, ahí está el camino: JavaScript, Node.js, conectar tu base a una aplicación real, y construir cosas que la gente use. El ecosistema full stack te espera, y tu ya tienes la mitad más difícil: los datos.

Gracias por llegar hasta aquí. No todos terminan lo que empiezan. Tu sí.

Ahora ve y construye algo. Esa es la verdadera graduación.

Esteban - lucasegaacademy.com

EXAMEN FINAL

Felicidades: llegaste al examen final del Curso Completo de SQL. Este examen cubre TODO lo que viste en las 10 unidades: primeros pasos, creación de tablas, SELECT y WHERE, filtrado y ordenamiento, LIKE y funciones de agregado, UPDATE, DELETE y llaves, JOINS, subconsultas, índices y vistas, el proyecto de la tienda y la consolidación final.

Instrucciones:

- Haz el examen SIN mirar tus apuntes y SIN copiar las consultas de las unidades. La idea es que compruebes lo que realmente se te quedó.
- Tiempo sugerido: 90 minutos. Si terminas antes, repasa las preguntas abiertas antes de irte a las respuestas.
- En las preguntas abiertas escribe el código tú mismo antes de ver la solución. No basta con "más o menos saberlo": el SQL se aprende escribiendo SQL.
- Como calificarte: suma 2 puntos por cada respuesta correcta, para un total de 100 puntos. 90 o más: excelente, dominas el curso. 70 a 88: bien, pero repasa las unidades donde fallaste. Menos de 70: vuelve a las unidades correspondientes y vuelve a intentarlo en unos días.

Las tablas del examen ----- Todas las preguntas abiertas usan estas tres tablas. Lee bien sus columnas antes de empezar:

```

CREATE TABLE clientes (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    ciudad TEXT,
    correo TEXT UNIQUE,
    fecha_registro DATE
);

CREATE TABLE productos (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    categoria TEXT,
    precio REAL NOT NULL,
    stock INTEGER DEFAULT 0
);

CREATE TABLE pedidos (
    id INTEGER PRIMARY KEY,
    cliente_id INTEGER,
    producto_id INTEGER,
    cantidad INTEGER NOT NULL,
    fecha DATE NOT NULL,
    FOREIGN KEY (cliente_id) REFERENCES clientes(id),
    FOREIGN KEY (producto_id) REFERENCES productos(id)
);

```

Datos de ejemplo que puedes asumir: la tabla clientes tiene clientes de varias ciudades como "Ciudad de Mexico", "Guadalajara" y "Monterrey". La tabla productos tiene categorias como "Electrónica", "Libros" y "Ropa". La tabla pedidos registra las compras de los clientes.

Ahora si: papel, pluma o editor de texto, y a darle.

Datos de ejemplo de las tablas ----- Para que las preguntas abiertas tengan contexto, estos son algunos datos de muestra que puedes asumir que existen en las tablas:

```
INSERT INTO clientes (id, nombre, ciudad, correo, fecha_registro) VALUES
(1, 'Juan Perez', 'Ciudad de Mexico', 'juan.perez@correo.com', '2026-01-15'),
(2, 'Ana Garcia', 'Guadalajara', 'ana.garcia@gmail.com', '2026-02-20'),
(3, 'Luis Torres', 'Monterrey', 'luis.torres@correo.com', '2026-03-10'),
(4, 'Sofia Ramos', 'Guadalajara', 'sofia.ramos@gmail.com', '2026-04-05');
```

```
INSERT INTO productos (id, nombre, categoria, precio, stock) VALUES
(1, 'Laptop Pro', 'Electronica', 15999.00, 10),
(2, 'Mouse inalambrico', 'Electronica', 499.00, 50),
(3, 'Curso de SQL', 'Libros', 299.00, 100),
(4, 'Playera basica', 'Ropa', 249.00, 200);
```

```
INSERT INTO pedidos (id, cliente_id, producto_id, cantidad, fecha) VALUES
(1, 1, 1, 1, '2026-05-01'),
(2, 2, 3, 2, '2026-05-03'),
(3, 1, 2, 3, '2026-06-10'),
(4, 3, 4, 5, '2026-07-22');
```

Nota: el cliente con id 4 (Sofia Ramos) no tiene pedidos registrados. Tenlo en cuenta para las preguntas sobre JOINS y clientes sin compras.

Estrategia para resolver el examen -----

- Primera pasada (30 minutos): responde todo lo que sepas de inmediato. Marca las preguntas donde dudes y sigue adelante; no te estanques.
- Segunda pasada (40 minutos): ataca las preguntas abiertas. Escribe la consulta completa aunque no estes seguro: un intentó escrito vale más que una idea a medias.
- Tercera pasada (20 minutos): revisa. Lee cada consulta en voz alta y preguntate: que tablas usa, que filtra el WHERE, que agrupa el GROUP BY. Los errores más comunes se cazan así.
- En opción múltiple, descarta primero las opciones imposibles. Casi siempre puedes eliminar dos y decidir entre las otras dos.
- En verdadero o falso, desconfía de las palabras absolutas: "siempre", "nunca", "todos los motores". Suelen esconder la trampa.

PREGUNTAS DE OPCIÓN MÚLTIPLE -----

PREGUNTA 1: Cual de las siguientes sentencias es la forma correcta de crear una base de datos llamada tienda en SQL estandar?

a) CREATE DATABASE tienda; b) MAKE DATABASE tienda; c) NEW DATABASE tienda; d) DATABASE CREATE tienda;

PREGUNTA 2: En la sentencia CREATE TABLE, que hace la restricción PRIMARY KEY?

a) Permite valores nulos y duplicados en la columna. b) Identifica de forma única cada fila y no permite nulos ni duplicados. c) Solo acelera las búsquedas, sin imponer reglas. d) Crea automáticamente una relación con otra tabla.

PREGUNTA 3: Qué tipo de dato es el más adecuado para guardar el precio de un producto, como 199.99?

a) INTEGER b) TEXT c) REAL d) DATE

PREGUNTA 4: Cual es la diferencia entre CHAR y VARCHAR?

a) No hay diferencia, son sinónimos. b) CHAR tiene longitud fija y VARCHAR longitud variable. c) VARCHAR solo acepta números y CHAR solo texto. d) CHAR es para fechas y VARCHAR para texto.

PREGUNTA 5: Que sentencia se usa para agregar una nueva columna a una tabla existente?

a) ALTER TABLE productos ADD COLUMN descuento REAL; b) UPDATE TABLE productos ADD descuento REAL; c) INSERT COLUMN descuento INTO productos; d) MODIFY productos ADD descuento REAL;

PREGUNTA 6: En un SELECT, para que sirve la cláusula DISTINCT?

a) Para ordenar los resultados de mayor a menor. b) Para eliminar filas duplicadas del resultado. c) Para filtrar filas con una condición. d) Para renombrar una columna con un alias.

PREGUNTA 7: Qué hace la siguiente consulta?

```
SELECT nombre, precio FROM productos WHERE precio > 100;
```

a) Muestra el nombre y precio de todos los productos. b) Muestra el nombre y precio solo de los productos que cuestan más de 100. c) Actualiza el precio de los productos a 100. d) Borra los productos que cuestan más de 100.

PREGUNTA 8: Cual operador lógico se usa para combinar dos condiciones que AMBAS deben cumplirse?

a) OR b) NOT c) AND d) LIKE

PREGUNTA 9: Que patrón LIKE encuentra los nombres que EMPIEZAN con "A"?

a) LIKE "%A" b) LIKE "A%" c) LIKE "%A%" d) LIKE "_A"

PREGUNTA 10: Cual es la diferencia entre WHERE y HAVING?

a) Son identicos, se pueden intercambiar libremente. b) WHERE filtra filas antes de agrupar y HAVING filtra grupos después de GROUP BY. c) HAVING filtra filas y WHERE filtra grupos. d) WHERE solo funciona con números y HAVING solo con texto.

PREGUNTA 11: Que función de agregado devuelve el número de filas de un grupo?

a) SUM() b) AVG() c) COUNT() d) MAX()

PREGUNTA 12: Quéhace la cláusula ORDER BY precio DESC?

a) Ordena los resultados por precio de menor a mayor. b) Ordena los resultados por precio de mayor a menor. c) Filtra los productos con precio descendente. d) Agrupa los productos por precio.

PREGUNTA 13: Cual es el peligro de ejecutar DELETE FROM clientes; sin WHERE?

a) No pasa nada, la sentencia falla sin WHERE. b) Borra unicamente la primera fila de la tabla. c) Borra TODAS las filas de la tabla clientes. d) Borra la tabla completa incluyendo su estructura.

PREGUNTA 14: Quéés una llave foránea (FOREIGN KEY)?

a) Una columna que identifica de forma única cada fila de su propia tabla. b) Una columna que hace referencia a la llave primaria de otra tabla y mantiene la integridad entre ambas. c) Una columna oculta que el motor crea automáticamente. d) Un índice que acelera las consultas pero no valida nada.

PREGUNTA 15: Quétipo de JOIN devuelve todas las filas de la tabla izquierda aunque no tengan coincidencia en la derecha?

a) INNER JOIN b) LEFT JOIN c) RIGHT JOIN d) CROSS JOIN

PREGUNTA 16: Quéhace una subconsulta en la cláusula WHERE con el operador IN?

a) Compara un valor contra una lista de valores devuelta por la subconsulta. b) Une dos tablas por una columna comun. c) Crea una tabla temporal en disco. d) Ordena los resultados de la subconsulta.

PREGUNTA 17: Para que sirve un índice (INDEX) en una tabla?

a) Para impedir valores duplicados siempre. b) Para acelerar las búsquedas y ordenamientos sobre las columnas indexadas. c) Para crear copias de seguridad automáticas. d) Para relacionar dos tablas entre sí.

PREGUNTA 18: Qué es una vista (VIEW)?

a) Una copia física de los datos de una tabla. b) Una consulta guardada que se comporta como una tabla virtual. c) Un permiso de usuario sobre la base de datos. d) Un tipo especial de índice.

PREGUNTA 19: ¿Cuál sentencia otorga permiso de solo lectura (SELECT) al usuario "ana" sobre la tabla productos?

a) GRANT SELECT ON productos TO ana; b) ALLOW ana READ productos; c) GIVE SELECT productos ana; d) PERMIT ana SELECT ON productos;

PREGUNTA 20: ¿Qué hace la sentencia BEGIN TRANSACTION; seguida de varias operaciones y luego ROLLBACK;?

a) Guarda permanentemente todas las operaciones. b) deshace todas las operaciones hechas desde el BEGIN, como si no hubieran pasado. c) Bloquea la base de datos para otros usuarios. d) Crea una copia de seguridad automática.

PREGUNTAS DE VERDADERO O FALSO -----

PREGUNTA 21: La sentencia SELECT * FROM clientes; muestra todas las columnas y todas las filas de la tabla clientes.

PREGUNTA 22: En SQL, los nombres de tablas y columnas distinguen entre mayúsculas y minúsculas en todos los motores de base de datos, así que Clientes y clientes siempre son tablas distintas.

PREGUNTA 23: La cláusula WHERE se escribe antes de GROUP BY en una consulta.

PREGUNTA 24: La función COUNT(*) cuenta también las filas donde hay valores NULL.

PREGUNTA 25: El operador BETWEEN 10 AND 20 incluye los valores 10 y 20 en el resultado.

PREGUNTA 26: Es posible actualizar varias columnas a la vez con una sola sentencia UPDATE usando comas: SET precio = 50, stock = 10.

PREGUNTA 27: Un INNER JOIN entre clientes y pedidos devuelve también los clientes que no tienen ningún pedido registrado.

PREGUNTA 28: La cláusula ON en un JOIN indica la condición que relaciona las dos tablas, normalmente igualando la llave foránea con la llave primaria.

PREGUNTA 29: Una subconsulta siempre se ejecuta después de la consulta externa que la contiene.

PREGUNTA 30: Crear un índice en una columna hace que los INSERT y UPDATE sobre esa tabla sean un poco más lentos, porque el índice también hay que mantenerlo.

PREGUNTA 31: Al borrar una vista con DROP VIEW, también se borran los datos de las tablas originales que la vista consultaba.

PREGUNTA 32: La restricción UNIQUE permite valores NULL en la mayoría de los motores, pero no permite dos valores iguales no nulos.

PREGUNTA 33: La sentencia TRUNCATE TABLE es equivalente a DELETE sin WHERE, pero normalmente es más rápida porque no registra cada fila borrada.

PREGUNTA 34: En un LEFT JOIN, si una fila de la tabla izquierda no tiene coincidencia en la derecha, las columnas de la derecha aparecen con valores NULL.

PREGUNTA 35: El alias definido con AS en el SELECT se puede usar dentro del WHERE de la misma consulta en todos los motores.

PREGUNTAS ABIERTAS CON CÓDIGO ----- Usa las tablas clientes, productos y pedidos descritas al inicio del examen. Escribe cada consulta completa y funcional.

PREGUNTA 36: Escribe una consulta que muestre el nombre y la ciudad de todos los clientes que viven en "Guadalajara", ordenados por nombre de la A a la Z.

PREGUNTA 37: Escribe una consulta que muestre el nombre y el precio de los productos de la categoría "Electrónica" que cuestan menos de 5000, ordenados por precio de mayor a menor.

PREGUNTA 38: Escribe una consulta que cuente cuantos clientes hay registrados en cada ciudad. El resultado debe mostrar la ciudad y el total de clientes, ordenado por el total de mayor a menor.

PREGUNTA 39: Escribe una consulta que muestre el nombre de cada producto junto con el total de unidades vendidas de ese producto (suma de cantidad en pedidos). Incluye los productos que no se han vendido nunca, mostrando 0 en esos casos. Ordena por total vendido de mayor a menor.

PREGUNTA 40: Escribe una sentencia UPDATE que aumente el precio un 10% a todos los productos de la categoría "Libros".

PREGUNTA 41: Escribe una consulta que muestre el nombre del cliente, el nombre del producto, la cantidad y la fecha de cada pedido, usando JOINS entre las tres tablas. Ordena por fecha de la más reciente a la más antigua.

PREGUNTA 42: Escribe una consulta que muestre los nombres de los clientes que han hecho al menos un pedido, usando una subconsulta con IN (sin usar JOIN).

PREGUNTA 43: Escribe una consulta que muestre las categorías de productos cuyo precio promedio sea mayor a 1000. Muestra la categoría y el precio promedio, y filtra usando HAVING.

PREGUNTA 44: Escribe la sentencia para crear un índice sobre la columna ciudad de la tabla clientes, y luego una vista llamada vista_pedidos_recientes que muestre id, cliente_id, producto_id, cantidad y fecha de los pedidos con fecha posterior al "2026-01-01".

PREGUNTA 45: Escribe una sentencia DELETE que borre todos los pedidos con fecha anterior al "2025-01-01". Explica en un comentario dentro del código por qué esta sentencia es peligrosa si se ejecuta sin revisar.

PREGUNTA 46: Escribe una consulta que muestre el nombre y el correo de los clientes cuyo correo termine en "@gmail.com", usando LIKE.

PREGUNTA 47: Escribe una consulta que muestre los 5 productos más caros de la tienda (nombre y precio), usando LIMIT.

PREGUNTA 48: Escribe una consulta que calcule el ingreso total generado por cada categoría de producto (precio multiplicado por cantidad vendida), mostrando la categoría y el ingreso total, ordenado de mayor a menor ingreso.

PREGUNTA 49: Escribe una sentencia INSERT que registre un cliente nuevo con id 101, nombre "Maria Lopez", ciudad "Monterrey", correo "maria.lopez@correo.com" y fecha de registro "2026-09-19".

PREGUNTA 50: Escribe una consulta que muestre el nombre de los clientes que NUNCA han hecho un pedido, usando LEFT JOIN y filtrando los NULL (sin usar subconsultas).

Cuando termines, ve a la sección de respuestas y califícate con honestidad. Recuerda: 2 puntos por respuesta correcta, 100 puntos en total.

Errores típicos que debes evitar -----

1. Confundir el orden de ejecución: SQL no se ejecuta en el orden en que se escribe. Primero FROM, luego WHERE, luego GROUP BY, luego HAVING, luego SELECT y al final ORDER BY. Por eso el alias del SELECT no sirve en el WHERE.
2. Olvidar el WHERE en UPDATE y DELETE: el error más caro del curso. Acostumbrate a escribir el WHERE primero o a probar con SELECT antes.
3. Usar comillas dobles para texto: en SQL estandar el texto va entre comillas simples ('Guadalajara'). Las comillas dobles son para identificadores en algunos motores.
4. Comparar con NULL usando =: NULL no es igual a nada. Usa IS NULL y IS NOT NULL, o tus filtros fallaran en silencio.
5. Poner funciones de agregado en el WHERE: WHERE AVG(precio) > 1000 es error. Los agregados se filtran con HAVING después de agrupar.
6. Pedir columnas sueltas junto a GROUP BY: si agrupas por ciudad, el SELECT solo puede llevar ciudad y funciones de agregado. Pedir nombre suelto es error lógico.
7. Olvidar el ON en los JOINS: sin condición de union obtienes un producto cartesiano que multiplica las filas y arruina los totales.
8. No calificar columnas ambiguas: si dos tablas tienen columna id, escribe c.id y no solo id, o el motor no sabra cual quieres.
9. Creer que DISTINCT ordena: DISTINCT quita duplicados, no ordena. Si quieres orden, agrega ORDER BY.
10. Mezclar HAVING sin GROUP BY: HAVING sin GROUP BY trata toda la tabla como un solo grupo. A veces es lo que quieres, pero casi siempre es un olvido.

GUÍA DE REPASO POR UNIDAD ----- Usa esta guía después de calificarte: busca las preguntas que fallaste y repasa la unidad indicada. Así no repasas todo el libro, solo lo que te falta.

Unidad 1 - Primeros pasos: qué es una base de datos, que es SQL, cómo se crea una base de datos y cómo se conecta un cliente al motor. Preguntas relacionadas: 1, 22. Si fallaste aquí, repasa los conceptos base: la diferencia entre el motor, la base de datos y las tablas, y la sintaxis mínima para empezar a trabajar.

Unidad 2 - Tablas: CREATE TABLE, tipos de datos (INTEGER, TEXT, REAL, DATE, CHAR, VARCHAR), restricciones NOT NULL, UNIQUE y DEFAULT, y ALTER TABLE para modificar la estructura. Preguntas relacionadas: 2, 3, 4, 5, 32, 49. Esta unidad es la base de todo: si no tienes claros los tipos de datos, las consultas más avanzadas se vuelven confusas. Práctica creando tablas desde cero sin mirar ejemplos.

Unidad 3 - SELECT y WHERE: la anatomía de una consulta, seleccionar columnas, el asterisco, filtros con comparadores (=, <>, >, <, >=, <=), operadores lógicos AND, OR y NOT, y el orden lógico de ejecución. Preguntas relacionadas: 6, 7, 8, 21, 23, 35, 36. La mayoría de los errores aquí vienen de confundir el orden en que SQL procesa las cláusulas: primero FROM, luego WHERE, luego SELECT y al final ORDER BY.

Unidad 4 - Filtrado y orden: BETWEEN, IN, IS NULL, ORDER BY con ASC y DESC, LIMIT para recortar resultados. Preguntas relacionadas: 12, 25, 47. Son herramientas simples pero muy usadas en la vida real: casi toda consulta de reporte termina con un ORDER BY y un LIMIT.

Unidad 5 - LIKE y agregados: patrones con % y _, funciones COUNT, SUM, AVG, MIN y MAX, GROUP BY para agrupar y HAVING para filtrar grupos. Preguntas relacionadas: 9, 10, 11, 24, 38, 43, 46. El punto más difícil de esta unidad es la diferencia entre WHERE y HAVING: si filtras con una función de agregado en el WHERE, la consulta falla. Memoriza el orden: WHERE filtra filas, GROUP BY agrupa, HAVING filtra grupos.

Unidad 6 - UPDATE, DELETE y llaves: modificar datos con UPDATE, borrar con DELETE, el peligro del WHERE olvidado, llaves primarias y foráneas, e INSERT de filas nuevas. Preguntas relacionadas: 13, 14, 26, 40, 45. La lección de oro de esta unidad: antes de un UPDATE o DELETE, corre el SELECT con la misma condición para ver exactamente que filas vas a tocar.

Unidad 7 - JOINS: INNER JOIN, LEFT JOIN, RIGHT JOIN, la cláusula ON, alias de tablas y cómo leer los NULL que deja un LEFT JOIN. Preguntas relacionadas: 15, 27, 28, 34, 39, 41, 50. Dibuja las tablas en papel y marca con flechas las llaves que las conectan: visualizar la relación hace que los JOINS dejen de ser magia.

Unidad 8 - Subconsultas, índices y vistas: subconsultas con IN, índices para acelerar búsquedas y su costo en escrituras, vistas como tablas virtuales, y control de acceso con GRANT. Preguntas relacionadas: 16, 17, 18, 19, 29, 30, 31, 42, 44. Recuerda la regla práctica: si la subconsulta devuelve una lista, IN es tu amigo; si devuelve un solo valor, compáralo directamente.

Unidad 9 - Proyecto tienda: todo junto en un caso realista. Las tablas clientes, productos y pedidos de este examen son las del proyecto, y las preguntas 37, 41, 42, 48 y 50 son mini versiones de los reportes que construiste ahí. Si fallaste varias de estas, vuelve al proyecto y reconstruye los reportes de ventas desde cero.

Unidad 10 - Consolidacion: transacciones con BEGIN, COMMIT y ROLLBACK, TRUNCATE TABLE contra DELETE, y buenas prácticas generales. Preguntas relacionadas: 20, 33. La idea central: la base de datos debe quedar siempre en un estado consistente, y las transacciones son la herramienta para garantizarlo cuando algo sale mal a la mitad de una operación.

Glosario rápido de sentencias del curso ----- Repasa esta tabla antes de empezar el examen. Si alguna línea no te dice nada, esa es tu señal de que debes repasar esa unidad.

Definición de datos:

- CREATE DATABASE tienda; Crea una base de datos nueva.
- CREATE TABLE ... (...); Crea una tabla con sus columnas, tipos y restricciones.
- ALTER TABLE productos ADD COLUMN descuento REAL; Agrega una columna a una tabla existente.
- ALTER TABLE productos DROP COLUMN descuento; Elimina una columna existente.
- DROP TABLE productos; Borra la tabla completa con su estructura y sus datos.
- CREATE INDEX idx ON clientes(ciudad); Crea un índice para acelerar búsquedas.
- CREATE VIEW v AS SELECT ...; Guarda una consulta como tabla virtual.

Manipulacion de datos:

- INSERT INTO clientes (nombre, ciudad) VALUES ('Ana', 'Lima'); Agrega una fila nueva.
- SELECT nombre, precio FROM productos; Lee columnas específicas.
- SELECT * FROM productos; Lee todas las columnas.
- SELECT DISTINCT ciudad FROM clientes; Quita filas duplicadas del resultado.
- UPDATE productos SET precio = 100 WHERE id = 5; Modifica filas que cumplan la condición.
- DELETE FROM pedidos WHERE fecha < '2025-01-01'; Borra filas que cumplan la condición.
- TRUNCATE TABLE pedidos; Vacía la tabla rápido, sin filtro.

Filtrado y orden:

- WHERE precio > 100 AND stock > 0; Filtra filas con condiciones combinadas.
- WHERE ciudad IN ('Lima', 'Quito'); Filtra por lista de valores.
- WHERE precio BETWEEN 10 AND 20; Filtra por rango inclusivo.

- WHERE nombre LIKE 'A%'; Filtra por patrón de texto.
- WHERE correo IS NULL; Filtra valores nulos.
- ORDER BY precio DESC; Ordena de mayor a menor.
- ORDER BY nombre ASC LIMIT 5; Ordena y recorta a 5 filas.

Agregados:

- SELECT COUNT(*) FROM clientes; Cuenta filas.
- SELECT AVG(precio), MAX(precio) FROM productos; Promedio y máximo.
- SELECT ciudad, COUNT(*) FROM clientes GROUP BY ciudad; Agrupa y cuenta por grupo.
- ... GROUP BY ciudad HAVING COUNT(*) > 10; Filtra grupos después de agrupar.

Relaciones:

- ... FROM pedidos pe JOIN clientes c ON pe.cliente_id = c.id; Une tablas por llave foránea.
- LEFT JOIN: conserva todas las filas de la izquierda aunque no haya coincidencia.
- WHERE id IN (SELECT cliente_id FROM pedidos); Subconsulta que devuelve una lista.
- FOREIGN KEY (cliente_id) REFERENCES clientes(id); Restricción que mantiene la integridad entre tablas.

Control y transacciones:

- GRANT SELECT ON productos TO ana; Da permiso de lectura a un usuario.
- REVOKE SELECT ON productos FROM ana; Quita un permiso.
- BEGIN TRANSACTION; ... COMMIT; Guarda un bloque de operaciones como una unidad.
- BEGIN TRANSACTION; ... ROLLBACK; deshace el bloque completo si algo salió mal.

Respuestas y soluciones

RESPUESTA 1: a) **CREATE DATABASE tienda;** Es la sintaxis estándar de SQL para crear una base de datos. La opción b) no existe como sentencia SQL, la c) tampoco es sintaxis válida y la d) invierte el orden de las palabras clave: el verbo **CREATE** siempre va primero.

RESPUESTA 2: b) **Identifica de forma única cada fila y no permite nulos ni duplicados.** Esa es la definición exacta de una llave primaria. La a) describe lo contrario, la c)

confunde la llave primaria con un índice (aunque muchos motores crean un índice automáticamente, su función principal es la unicidad) y la d) describe una llave foránea.

RESPUESTA 3: c) REAL. Los precios con decimales se guardan en tipos de punto flotante o decimal como REAL. INTEGER solo guarda enteros y perdería los centavos, TEXT guardaría el precio como texto y no permitiría operaciones matemáticas, y DATE es para fechas.

RESPUESTA 4: b) CHAR tiene longitud fija y VARCHAR longitud variable. CHAR(10) siempre ocupa 10 espacios aunque guardes menos caracteres, mientras que VARCHAR(10) ocupa solo lo necesario. La a) es falsa, la c) inventa una restricción que no existe y la d) confunde los tipos.

RESPUESTA 5: a) ALTER TABLE productos ADD COLUMN descuento REAL; ALTER TABLE es la sentencia para modificar la estructura de una tabla existente. UPDATE sirve para cambiar datos, no estructura; INSERT agrega filas, no columnas; y MODIFY no es sintaxis estandar para esto.

RESPUESTA 6: b) Para eliminar filas duplicadas del resultado. SELECT DISTINCT quita los repetidos del conjunto de resultados. Ordenar es trabajo de ORDER BY, filtrar es trabajo de WHERE y renombrar columnas se hace con AS.

RESPUESTA 7: b) Muestra el nombre y precio solo de los productos que cuestan más de 100. El WHERE filtra las filas antes de mostrarlas. La a) ignora el filtro, la c) confunde SELECT con UPDATE y la d) confunde SELECT con DELETE.

RESPUESTA 8: c) AND. AND exige que ambas condiciones sean verdaderas. OR pide que al menos una se cumpla, NOT niega una condición y LIKE es un operador de comparación de patrones, no un operador lógico.

RESPUESTA 9: b) LIKE "A%". El símbolo % representa cualquier cantidad de caracteres, así que "A%" significa "empieza con A". "%A" termina con A, "%A%" contiene A en cualquier parte y "_A" significa cualquier carácter seguido de A.

RESPUESTA 10: b) WHERE filtra filas antes de agrupar y HAVING filtra grupos después de GROUP BY. Esa es la diferencia clave y el orden lógico de ejecución. La a) es falsa, la c) invierte los papeles y la d) inventa una restricción por tipo de dato que no existe.

RESPUESTA 11: c) COUNT(). Cuenta el número de filas del grupo. SUM() suma valores, AVG() promedia y MAX() devuelve el valor máximo: ninguna de las tres cuenta filas.

RESPUESTA 12: b) Ordena los resultados por precio de mayor a menor. DESC significa descendente. ASC sería de menor a mayor. ORDER BY no filtra ni agrupa, solo ordena.

RESPUESTA 13: c) Borra TODAS las filas de la tabla clientes. Un DELETE sin WHERE no tiene filtro, así que afecta a todas las filas. No falla (la a es falsa), no borra solo la primera (la b es falsa) y la estructura de la tabla queda intacta, así que la d también es falsa.

RESPUESTA 14: b) Una columna que hace referencia a la llave primaria de otra tabla y mantiene la integridad entre ambas. Por ejemplo, pedidos.cliente_id referencia clientes.id. La a) describe una llave primaria, la c) es falsa y la d) describe un índice, no una llave foránea.

RESPUESTA 15: b) LEFT JOIN. Devuelve todas las filas de la tabla de la izquierda y NULL donde no hay coincidencia. INNER JOIN solo trae las coincidencias, RIGHT JOIN privilegia la tabla derecha y CROSS JOIN hace el producto cartesiano.

RESPUESTA 16: a) Compara un valor contra una lista de valores devuelta por la subconsulta. Por ejemplo: WHERE id IN (SELECT cliente_id FROM pedidos). La b) describe un JOIN, la c) es falsa y la d) describe ORDER BY.

RESPUESTA 17: b) Para acelerar las búsquedas y ordenamientos sobre las columnas indexadas. El índice es una estructura auxiliar que evita leer la tabla completa. No siempre impide duplicados (solo si es UNIQUE), no hace respaldos y no relaciona tablas.

RESPUESTA 18: b) Una consulta guardada que se comporta como una tabla virtual. No duplica datos físicamente (la a es falsa), no es un permiso (la c es falsa) y no es un índice (la d es falsa).

RESPUESTA 19: a) GRANT SELECT ON productos TO ana; Es la sintaxis estandar del control de permisos. Las demás opciones usan palabras que no existen en SQL estandar para este proposito.

RESPUESTA 20: b) Deshace todas las operaciones hechas desde el BEGIN, como si no hubieran pasado. ROLLBACK cancela la transacción. COMMIT seria la que guarda permanentemente (la a), y ni bloquea la base ni hace respaldos por si sola.

RESPUESTA 21: Verdadero. El asterisco (*) selecciona todas las columnas y, al no haber WHERE, se devuelven todas las filas.

RESPUESTA 22: Falso. La sensibilidad a mayusculas depende del motor y del sistema operativo. En MySQL sobre Linux los nombres de tablas si distinguen, pero en SQL Server o en Postgres el comportamiento varia; no es una regla universal de SQL.

RESPUESTA 23: Verdadero. El orden lógico es WHERE primero (filtra filas) y después GROUP BY (agrupa lo que sobrevivió al filtro).

RESPUESTA 24: Verdadero. COUNT(*) cuenta filas, sin importar si tienen NULL. Ojo: COUNT(columna) si ignora los NULL de esa columna. Esa diferencia es clásica de examen.

RESPUESTA 25: Verdadero. BETWEEN es inclusivo en ambos extremos: incluye el 10 y el 20.

RESPUESTA 26: Verdadero. Se pueden asignar varias columnas separándolas con comas dentro del mismo SET.

RESPUESTA 27: Falso. INNER JOIN solo devuelve las filas que tienen coincidencia en ambas tablas. Los clientes sin pedidos se quedan fuera; para incluirlos se necesitaría un LEFT JOIN.

RESPUESTA 28: Verdadero. ON define cómo se emparejan las filas, típicamente con la igualdad entre la llave foránea y la llave primaria.

RESPUESTA 29: Falso. En general la subconsulta se evalúa primero para producir los valores que la consulta externa necesita (por ejemplo la lista del IN). Hay subconsultas correlacionadas donde la ejecución se entrelaza, pero nunca "siempre después".

RESPUESTA 30: Verdadero. El índice acelera lecturas pero cada escritura debe actualizar también el índice, así que los INSERT, UPDATE y DELETE pagan un pequeño costo extra.

RESPUESTA 31: Falso. DROP VIEW solo elimina la definición de la vista. Las tablas base y sus datos quedan intactos.

RESPUESTA 32: Verdadero. UNIQUE impide duplicados entre valores no nulos, pero la mayoría de los motores permite varios NULL porque NULL no se considera igual a otro NULL.

RESPUESTA 33: Verdadero. TRUNCATE TABLE vacía la tabla más rápido porque no borra fila por fila con registro detallado. Ojo: en algunos motores no se puede deshacer con ROLLBACK, así que usalo con cuidado.

RESPUESTA 34: Verdadero. Esa es justo la marca de un LEFT JOIN: conserva la izquierda completa y rellena con NULL lo que no encontró a la derecha.

RESPUESTA 35: Falso. El WHERE se evalúa lógicamente antes que el SELECT, así que el alias aun no existe cuando el WHERE se procesa. Por eso muchos motores marcan

error si intentas usar el alias en el WHERE. (En ORDER BY si se puede usar, porque se evalua después.)

RESPUESTA 36:

```
SELECT nombre, ciudad
FROM clientes
WHERE ciudad = 'Guadalajara'
ORDER BY nombre ASC;
```

Explicación: el WHERE filtra solo los clientes de Guadalajara y el ORDER BY nombre ASC los ordena alfabeticamente. ASC es el orden por defecto, pero escribirlo deja clara la intención.

RESPUESTA 37:

```
SELECT nombre, precio
FROM productos
WHERE categoria = 'Electronica' AND precio < 5000
ORDER BY precio DESC;
```

Explicación: se combinan dos condiciones con AND (categoría exacta y precio menor a 5000) y se ordena por precio descendente para ver primero el más caro.

RESPUESTA 38:

```
SELECT ciudad, COUNT(*) AS total_clientes
FROM clientes
GROUP BY ciudad
ORDER BY total_clientes DESC;
```

Explicación: GROUP BY agrupa por ciudad y COUNT(*) cuenta los clientes de cada grupo. El alias total_clientes se puede usar en el ORDER BY porque este se evalua después del SELECT.

RESPUESTA 39:

```
SELECT p.nombre, COALESCE(SUM(pe.cantidad), 0) AS total_vendido
FROM productos p
LEFT JOIN pedidos pe ON pe.producto_id = p.id
GROUP BY p.id, p.nombre
ORDER BY total_vendido DESC;
```

Explicación: el LEFT JOIN conserva todos los productos aunque no tengan pedidos; en esos casos SUM() devuelve NULL y COALESCE lo convierte en 0. Se agrupa por producto y se ordena por lo vendido.

RESPUESTA 40:

```
UPDATE productos
SET precio = precio * 1.10
WHERE categoria = 'Libros';
```

Explicación: multiplicar por 1.10 aumenta el 10%. El WHERE es vital: sin el, todos los productos subirían de precio. Siempre revisa el WHERE antes de ejecutar un UPDATE.

RESPUESTA 41:

```
SELECT c.nombre AS cliente, p.nombre AS producto,
       pe.cantidad, pe.fecha
FROM pedidos pe
JOIN clientes c ON pe.cliente_id = c.id
JOIN productos p ON pe.producto_id = p.id
ORDER BY pe.fecha DESC;
```

Explicación: se encadenan dos INNER JOINs usando las llaves foráneas de pedidos. Los alias (c, p, pe) acortan la consulta y AS renombra las columnas de salida para que se lean mejor.

RESPUESTA 42:

```
SELECT nombre
FROM clientes
WHERE id IN (SELECT cliente_id FROM pedidos);
```

Explicación: la subconsulta devuelve los id de clientes que aparecen en pedidos y el IN filtra los clientes cuyo id esté en esa lista. No usa JOIN, tal como pedía la pregunta.

RESPUESTA 43:

```
SELECT categoria, AVG(precio) AS precio_promedio
FROM productos
GROUP BY categoria
HAVING AVG(precio) > 1000;
```

Explicación: GROUP BY agrupa por categoría y HAVING filtra los grupos cuyo promedio supera 1000. Aquí se ve la diferencia con WHERE: no se puede usar WHERE con AVG() porque el promedio aun no existe antes de agrupar.

RESPUESTA 44:

```
CREATE INDEX idx_clientes_ciudad ON clientes(ciudad);

CREATE VIEW vista_pedidos_recientes AS
SELECT id, cliente_id, producto_id, cantidad, fecha
FROM pedidos
WHERE fecha > '2026-01-01';
```

Explicación: el índice acelera las búsquedas por ciudad. La vista guarda la consulta como una tabla virtual: cada vez que consultes vista_pedidos_recientes verás los pedidos recientes con los datos actualizados.

RESPUESTA 45:

```
-- PELIGRO: sin un respaldo previo, los pedidos borrados no se pueden
-- recuperar. Revisa la fecha en el WHERE y prueba primero con un SELECT.
DELETE FROM pedidos
WHERE fecha < '2025-01-01';
```

Explicación: el WHERE limita el borrado a pedidos viejos, pero un error en la fecha (o un WHERE olvidado) borraría más de lo planeado. La buena práctica es correr primero el SELECT con la misma condición para ver que filas se afectarían.

RESPUESTA 46:

```
SELECT nombre, correo
FROM clientes
WHERE correo LIKE '%@gmail.com';
```

Explicación: el % al inicio significa "cualquier texto antes", así que el patrón coincide con correos que terminen exactamente en @gmail.com.

RESPUESTA 47:

```
SELECT nombre, precio
FROM productos
ORDER BY precio DESC
LIMIT 5;
```

Explicación: se ordena por precio descendente y LIMIT 5 corta el resultado en las primeras 5 filas, que son las más caras.

RESPUESTA 48:

```
SELECT p.categoria, SUM(p.precio * pe.cantidad) AS ingreso_total
FROM productos p
JOIN pedidos pe ON pe.producto_id = p.id
GROUP BY p.categoria
ORDER BY ingreso_total DESC;
```

Explicación: el JOIN conecta cada pedido con su producto para conocer el precio, se multiplica precio por cantidad y se suma por categoría. Solo cuentan las categorías con ventas, porque el INNER JOIN excluye productos sin pedidos.

RESPUESTA 49:

```
INSERT INTO clientes (id, nombre, ciudad, correo, fecha_registro)
VALUES (101, 'Maria Lopez', 'Monterrey', 'maria.lopez@correo.com', '2026-09-19');
```

Explicación: se listan las columnas en orden y luego los valores correspondientes. Nombrar las columnas es buena práctica: si la tabla cambia, el INSERT sigue funcionando.

RESPUESTA 50:

```
SELECT c.nombre
FROM clientes c
LEFT JOIN pedidos pe ON pe.cliente_id = c.id
WHERE pe.id IS NULL;
```

Explicación: el LEFT JOIN trae todos los clientes y pone NULL en las columnas de pedidos donde no hay compra. El WHERE pe.id IS NULL se queda justo con esos: los clientes sin ningún pedido. IS NULL es la forma correcta de comparar con NULL, porque NULL no es igual a nada, ni siquiera a otro NULL.

Fin del examen. Califica tus respuestas, anota en que unidades fallaste y repasalas antes de dar el curso por terminado. Si sacaste 90 o más, ya puedes decir que dominas SQL: ahora a construir proyectos de verdad en lucasegaacademy.com.

Hoja de respuestas rápida ----- 1: a - 2: b - 3: c - 4: b - 5: a - 6: b - 7: b - 8: c - 9: b - 10: b 11: c - 12: b - 13: c - 14: b - 15: b - 16: a - 17: b - 18: b - 19: a - 20: b 21: Verdadero - 22: Falso - 23: Verdadero - 24: Verdadero - 25: Verdadero 26: Verdadero - 27: Falso - 28: Verdadero - 29: Falso - 30: Verdadero 31: Falso - 32: Verdadero - 33: Verdadero - 34: Verdadero - 35: Falso 36 a 50: código abierto, revisa las soluciones explicadas arriba.

Colibrí Editorial EGA

Curso Completo de SQL

Curso completo de SQL: bases de datos, diseño de tablas, consultas, joins, proyecto de tienda en línea y graduación, con prácticas y examen final.

lucasegaacademy.com

© 2026 Esteban Gutiérrez · Colibrí Editorial EGA · Hecho en México