La Biblia de SQL

De Principiante a Experto en Bases de Datos Relacionales

Creado por "Roberto Arce"

Este libro está basado en experiencias reales y contiene opiniones sobre el ejercicio profesional de la calidad en proyectos de software.
Nombres de productos, empresas o situaciones reales se mencionan únicamente con fines educativos.

Primera edición: 2025
Diseño y estrategia editorial: **QA sin filtros**

Publicado por el autor a través de Amazon Kindle Direct Publishing (KDP)
www.amazon.com/kdp

Prólogo

Vivimos en una era donde los datos lo son todo. Desde los historiales médicos hasta las recomendaciones de películas, desde las transacciones bancarias hasta las plataformas de comercio electrónico, **todo está respaldado por una base de datos**. Y si hay un lenguaje universal en ese mundo, ese es **SQL**.

Cuando comencé a trabajar con bases de datos, SQL me pareció una herramienta simple... hasta que descubrí su verdadero poder. Las consultas que parecían básicas escondían una lógica profunda, y detrás de cada buen sistema, había un diseño de datos sólido, eficiente y mantenible. Años después, comprendí que dominar SQL no era solo escribir `SELECT * FROM tabla`, sino entender cómo fluye la información, cómo se relaciona y cómo optimizarla para que responda en milisegundos aunque tenga millones de filas.

Este libro nace con un propósito claro: **llevarte de cero hasta un nivel profesional en SQL**, sin atajos, sin tecnicismos innecesarios, pero con toda la profundidad que requiere trabajar en entornos reales. Ya seas programador, tester, analista, estudiante o simplemente curioso, aquí encontrarás un camino ordenado y práctico para convertirte en un experto en bases de datos relacionales.

Cada capítulo está diseñado con ejemplos reales, ejercicios y mini-proyectos que reflejan escenarios del mundo laboral. No solo aprenderás a escribir buenas consultas: aprenderás a **modelar, optimizar, asegurar y pensar como un verdadero arquitecto de datos**.

Este es el primer paso de un viaje mayor. Porque después de SQL vienen los sistemas distribuidos, los data warehouses, el análisis avanzado y mucho más. Pero todo gran edificio comienza con unos buenos cimientos, y eso es justo lo que estás a punto de construir.

Bienvenido a *La Biblia de SQL*. El conocimiento está a unas páginas de distancia.

ÍNDICE GENERAL

Una guía completa para dominar SQL desde los fundamentos hasta la optimización avanzada, el modelado de datos, la seguridad y las mejores prácticas profesionales.

- Buenas prácticas y errores comunes
- **Ejercicios prácticos:** subconsultas y funciones

CAPÍTULO 3: Modelado de Bases de Datos

- ¿Qué es el modelado de datos y cuál es su objetivo?
- Tipos de modelos: Conceptual, Lógico y Físico
  - Modelo de datos conceptual
  - Modelo de datos lógico
  - Modelo de datos físico

- Normalización de datos
  - Primera Forma Normal (1FN)
  - Segunda Forma Normal (2FN)
  - Tercera Forma Normal (3FN)
  - Normalizar vs. Desnormalizar — buenas prácticas
  - **Ejercicios prácticos:** normalización

- Claves e índices
  - Claves primarias: identificadores únicos
  - Claves foráneas: relaciones e integridad referencial
  - Índices: acelerar el acceso a los datos
  - Errores comunes y cómo evitarlos
  - **Ejercicios prácticos:** claves e índices

- Diseño lógico y físico de bases de datos
  - Qué es el diseño lógico
  - Qué es el diseño físico
  - Diferencias y relación entre ambos
  - **Ejemplo práctico:** diseño de esquema completo
  - **Ejercicios prácticos:** creación de esquemas

CAPÍTULO 4: Funciones avanzadas y consultas complejas

- Funciones agregadas
  - SUM, COUNT, AVG, MIN, MAX

- Funciones analíticas y de ventana
  - OVER, PARTITION BY, ORDER BY, RANK, ROW_NUMBER, DENSE_RANK

- Expresiones condicionales
  - Sentencia CASE
  - Función COALESCE

- Funciones de texto y de fecha
  - UPPER, LOWER, CONCAT, LENGTH
  - NOW, DATE_ADD, DATEDIFF

- Vistas (*Views*)
- Tablas virtuales definidas por una consulta
- Procedimientos almacenados (*Stored Procedures*)
- Triggers (*Disparadores*)
  - Definición, uso y precauciones

- **Ejercicios prácticos:** consultas avanzadas

CAPÍTULO 5: Optimización y Buenas Prácticas

- **EXPLAIN y planes de ejecución**
  - Qué es `EXPLAIN` y cómo leer los resultados
  - Interpretación en MySQL y PostgreSQL
  - Consultas ineficientes vs. Optimizadas
  - Buenas prácticas para interpretar y optimizar
  - Errores comunes al analizar planes de ejecución
  - **Ejercicios prácticos:** análisis de rendimiento

- **Índices y su impacto**
  - Qué es un índice y cómo funciona
  - Tipos de índices (BTREE, HASH, GIN, etc.)
  - Creación y eliminación de índices
  - Comparación: consultas con y sin índice
  - Cuándo usar o evitar índices
  - Buenas prácticas de indexación
  - **Ejercicios prácticos:** índices en acción

- **Transacciones y control de concurrencia**
  - Uso de transacciones (`BEGIN`, `COMMIT`, `ROLLBACK`)
  - Control de concurrencia en entornos multiusuario
  - Niveles de aislamiento (`READ COMMITTED`, `REPEATABLE READ`, etc.)
  - Fenómenos clásicos: *dirty reads*, *phantom reads*, *non-repeatable reads*
  - Bloqueos y concurrencia en MySQL y PostgreSQL
  - Buenas prácticas para transacciones seguras
  - **Ejercicios prácticos:** manejo de transacciones

- **Seguridad básica: roles y permisos**
  - Autenticación y autorización
  - Usuarios vs. Roles
  - Creación y gestión de usuarios
  - Privilegios con `GRANT` y `REVOKE`
  - Ejemplos por departamento o función
  - Buenas prácticas en entornos multiusuario y cloud
  - **Ejercicios prácticos:** roles y permisos

EJERCICIOS RECOMENDADOS PARA EL LIBRO

- Parte 1: Fundamentos
- Parte 2: Operaciones CRUD

- Parte 3: Modelado de Datos
- Parte 4: Consultas Avanzadas
- Parte 5: Optimización y Seguridad

Proyectos finales sugeridos:

- Dashboard de reportes para un e-commerce
- Seguimiento de pruebas para un equipo QA
- Sistema de reservas con gestión de conflictos

BONUS: Recursos y herramientas profesionales

- Clientes SQL recomendados (DBeaver, DataGrip, TablePlus)
- Extensiones y herramientas de optimización
- Sitios de práctica: SQLZoo, LeetCode SQL, Mode Analytics
- Guía de comandos SQL más usados
- Checklist de buenas prácticas para producción

Capítulo 1: Fundamentos de SQL

Instalación de MySQL y PostgreSQL en Windows, macOS y Linux

Comenzaremos por el principio: **instalar un sistema de gestión de bases de datos relacionales (RDBMS)**. En esta sección veremos cómo instalar las dos plataformas SQL de código abierto más populares – **MySQL** y **PostgreSQL** – en los principales sistemas operativos (Windows, macOS y Linux). También presentaremos herramientas gráficas útiles (como **DBeaver** o **pgAdmin**) para administrar las bases de datos, así como la forma de verificar la instalación usando la terminal.

Instalación de MySQL

En Windows: MySQL ofrece un instalador gráfico fácil de usar.

1. Accede al sitio oficial de MySQL y descarga el **MySQL Installer** para Windows (MySQL Community Server). El instalador unificado incluye el servidor MySQL y opcionalmente herramientas como MySQL Workbench.
2. Ejecuta el instalador descargado y sigue los pasos del asistente de instalación. Durante el proceso podrás elegir el tipo de setup (típico o personalizado) e instalar componentes adicionales. Se te pedirá configurar una contraseña para el usuario **root** (administrador) de MySQL y opcionalmente crear usuarios adicionales. También puedes ajustar el puerto (por defecto **3306**) y otros parámetros, aunque generalmente se pueden dejar los valores por defecto.
3. Una vez completada la instalación, verifica que MySQL esté funcionando. Puedes hacerlo de dos formas:
   a) Abre la aplicación **MySQL Workbench** (si la instalaste) o el **MySQL Shell** y conéctate al servidor local usando el usuario root y la contraseña que definiste.
   b) Alternativamente, abre una ventana de *Símbolo del sistema* (CMD) y ejecuta el comando `mysql -u root -p` (o `mysql --version` para comprobar la versión). Si aparece el prompt de MySQL (tras ingresar tu contraseña de root) o la versión instalada, la instalación fue exitosa.

En macOS: La forma más sencilla es utilizar el instalador oficial para Mac:

1. Descarga el paquete **DMG** de MySQL Community Server desde el sitio oficial de MySQL. Elige la versión más reciente para macOS (por ejemplo, un archivo `.dmg` para MySQL 8.x en arquitectura Intel o ARM según tu hardware).
2. Abre el archivo DMG y ejecuta el instalador de MySQL. Sigue las instrucciones del asistente gráfico. Durante la instalación en macOS, también se te solicitará configurar una contraseña para el usuario **root** de MySQL. Es posible que el

instalador ofrezca la opción de instalar un panel de preferencias para iniciar/detener el servidor MySQL desde *Preferencias del Sistema*.

3. Tras finalizar, verifica la instalación abriendo la **Terminal** de macOS. Puedes ejecutar `mysql -u root -p` para conectarte (ingresa la contraseña que definiste) o `mysql --version` para ver la versión. Asimismo, MySQL suele iniciar automáticamente como servicio; de no ser así, puedes iniciarlo manualmente desde el panel de preferencias de MySQL o con el comando `sudo launchctl start com.mysql.mysql` en Terminal.

En Linux: La instalación varía ligeramente según la distribución, pero en general:

- **Distribuciones basadas en Debian/Ubuntu:** MySQL está disponible en los repositorios oficiales. Abre una terminal y ejecuta:

```
bash
sudo apt update && sudo apt install mysql-server
```

Esto instalará el servidor MySQL (y el cliente de línea de comandos). Durante la instalación en algunas versiones, se te pedirá establecer la contraseña de root; en otras, tendrás que configurarla luego usando el script `mysql_secure_installation`. Tras instalar, el servicio MySQL normalmente se inicia automáticamente. Puedes comprobar el estado con `sudo systemctl status mysql`. También verifica la versión con `mysql --version`.

- **Distribuciones basadas en Red Hat/Fedora/CentOS:** Si usas Fedora, CentOS Stream u otra variante, puedes instalar MySQL con *yum* o *dnf*. Por ejemplo, en Fedora:

```
bash
sudo dnf install mysql-server
sudo systemctl enable --now mysqld
```

En CentOS/RHEL puede ser necesario habilitar el repositorio de MySQL si no se incluye por defecto (ya que a veces se ofrece **MariaDB** como reemplazo compatible). MySQL proporciona sus propios repositorios YUM/APT con los paquetes más recientes. Consulta la documentación oficial si necesitas una versión específica. Tras la instalación, inicia el servicio (si no se inició automáticamente) con `sudo systemctl start mysqld` y verifica con `mysql --version` en la terminal.

Una vez instalado MySQL en Linux, ejecuta `mysql -u root -p` para confirmar que puedes conectarte (por primera vez es posible que el usuario root no tenga contraseña en algunas distros, o que debas usar autenticación Unix socket). Es **altamente recomendable** ejecutar `mysql_secure_installation` para configurar la seguridad básica (establecer contraseña de root, eliminar usuarios anónimos, deshabilitar accesos remotos de root, etc.).

Herramientas gráficas para MySQL: MySQL incluye su propia GUI oficial llamada **MySQL Workbench**, que es muy útil para diseño de bases de datos, ejecución de consultas SQL y administración en general. Puedes descargar MySQL Workbench gratuitamente desde el sitio oficial de MySQL, aunque en Windows suele venir incluido con el Installer. Otra herramienta popular es **DBeaver**, de la que hablaremos más abajo, que permite conectar a MySQL entre muchos otros motores.

Instalación de PostgreSQL

En Windows: PostgreSQL también ofrece un instalador gráfico para Windows, distribuido por EDB.

1. Dirígete a la sección de descargas del sitio oficial de PostgreSQL y descarga el instalador para Windows (archivo `.exe`). El sitio te dirigirá al instalador mantenido por EnterpriseDB, donde podrás seleccionar la versión de PostgreSQL deseada (por ejemplo, PostgreSQL 15 o 16) para Windows.
2. Ejecuta el instalador y sigue las indicaciones. Podrás elegir la ruta de instalación y los componentes. De forma predeterminada se instala el servidor PostgreSQL, el cliente de línea de comandos **psql**, y también la herramienta **pgAdmin 4** (entorno web/desktop para administrar PostgreSQL). Durante la instalación se te pedirá definir una contraseña para el usuario administrador por defecto (**postgres**), y puedes dejar el puerto en **5432** que es el estándar.
3. Al finalizar, verifica la instalación:
   a) Busca *SQL Shell (psql)* en el menú Inicio y ábrelo. Se abrirá la consola de PostgreSQL; ingresa los datos que te solicita (por defecto: servidor *localhost*, puerto 5432, usuario *postgres* y tu contraseña, base de datos *postgres*) y deberías obtener el prompt `postgres=#` si todo funcionó.
   b) Como alternativa, puedes abrir *pgAdmin 4* (se instala como aplicación de escritorio o accesible vía navegador). Al abrir pgAdmin por primera vez, te pedirá la contraseña de admin (puedes usar la de *postgres* que configuraste). Luego verás en el panel el servidor local PostgreSQL; al expandirlo e introducir la contraseña, podrás gestionarlo visualmente.

En macOS: Para macOS existen varias opciones. La más sencilla es usar el instalador gráfico de PostgreSQL:

1. Descarga el instalador para macOS desde la página oficial de PostgreSQL (también provisto por EDB, similar al de Windows). Obtendrás un paquete `.dmg` o `.pkg`.
2. Ejecuta el instalador y sigue los pasos. Te pedirá una contraseña para el usuario *postgres* y configurará el resto automáticamente.
3. Tras la instalación, puedes verificar abriendo la aplicación **Terminal** y ejecutando el cliente `psql`. Por ejemplo, ejecuta: `psql -U postgres` (posiblemente debas añadir el directorio bin de PostgreSQL al PATH, o invocar el comando con la ruta completa). Si aparece el prompt `postgres=#` tras ingresar la contraseña, la conexión fue exitosa. Alternativamente, inicia **pgAdmin 4** (en macOS se instala como una aplicación independiente) y conecta al servidor local para comprobar que todo esté en orden.

Nota: Otra opción popular en macOS es utilizar **Homebrew** para instalar PostgreSQL con el comando `brew install postgresql` y posteriormente iniciar el servicio con `brew services start postgresql`. Esto instala PostgreSQL de forma sencilla desde la terminal (aunque no incluye pgAdmin).

En Linux: PostgreSQL suele estar disponible directamente en los repositorios de la mayoría de distribuciones, lo que facilita su instalación:

- **Debian/Ubuntu:** Ejecuta en la terminal:

```
bash
sudo apt update && sudo apt install postgresql postgresql-client
```

Esto instala el servidor PostgreSQL y el cliente `psql`. El proceso suele crear automáticamente un usuario de sistema *postgres* y una base de datos inicial. El servicio se inicia automáticamente tras la instalación. Puedes cambiar a la cuenta *postgres* (`sudo -i -u postgres`) y luego ejecutar `psql` para entrar a la consola interactiva de PostgreSQL. También puedes conectarte como usuario normal usando `psql -U postgres -h localhost` si la autenticación está configurada para permitirlo. Comprueba la versión con `psql --version`.

- **Red Hat/CentOS/Fedora:** En distribuciones RHEL-based, posiblemente quieras añadir el repositorio oficial de PostgreSQL (ya que a veces las versiones incluidas por defecto pueden ser antiguas). Sin embargo, en las versiones recientes de Fedora/CentOS, puedes instalar directamente con dnf/yum. Por ejemplo, en CentOS 8+ o AlmaLinux/Rocky:

```
bash
sudo dnf install @postgresql
```

(Algunos sistemas ofrecen un grupo de instalación `@postgresql` o paquetes como `postgresql-server` y `postgresql-contrib`.) Después de instalar, puede ser necesario inicializar la base de datos con `postgresql-setup --initdb` (en RHEL) y luego iniciar el servicio: `sudo systemctl enable --now postgresql`. Consulta la documentación específica de tu distro si es necesario. La comprobación es similar: usar `psql` para verificar la conexión local.

Una vez instalado PostgreSQL en Linux, típicamente el usuario administrador es *postgres*. Para acceder a la consola, puedes usar `sudo -u postgres psql` directamente (sin contraseña, usando autenticación peer en localhost por defecto) y así obtener el prompt `postgres=#`. Desde allí puedes crear otros usuarios/roles y bases de datos según necesites.

Herramientas gráficas para PostgreSQL: La herramienta oficial es **pgAdmin 4**, que se instala junto con PostgreSQL en los instaladores de Windows/macOS, y está disponible como paquete en Linux o como aplicación web. pgAdmin permite administrar el servidor, crear bases, visualizar tablas y ejecutar consultas en una interfaz amigable. Es un proyecto de código abierto mantenido por la comunidad de PostgreSQL. Por otro lado, **DBeaver** (Community Edition) es una herramienta gráfica

universal que soporta PostgreSQL y muchos otros motores; es gratuita y de código abierto. Puedes descargar DBeaver para Windows, Mac o Linux desde su web oficial. DBeaver resulta muy útil si trabajas con múltiples sistemas de bases de datos, ya que ofrece una interfaz única para todos ellos. En cualquier caso, tanto pgAdmin como DBeaver permiten conectarte a tu servidor PostgreSQL local (o remoto) proporcionando host, puerto, usuario y contraseña, ofreciendo capacidades similares de administración.

Resumen de esta sección: A estas alturas deberías tener MySQL y/o PostgreSQL instalados en tu sistema operativo, junto con alguna herramienta para manejarlos. En todos los casos, recuerda que tras la instalación es importante **recordar las credenciales de administrador (usuario y contraseña)** que configuraste, pues las necesitarás para crear bases de datos y tablas. También, mantener el servicio de base de datos en ejecución (muchos instaladores configuran el servicio para que se inicie automáticamente con el sistema, pero puedes iniciarlo/detenerlo manualmente si hace falta).

Pasemos ahora a entender qué son las bases de datos relacionales que acabamos de instalar y cómo se estructuran sus datos.

¿Qué es una base de datos relacional?

Una **base de datos relacional** es un tipo de base de datos que almacena y organiza datos en forma de tablas relacionadas entre sí mediante claves comunes. Sigue el modelo relacional propuesto por Edgar F. Codd en 1970, el cual resultó ser una forma muy intuitiva y poderosa de gestionar información estructurada. En una base de datos relacional, los datos no se guardan en un único bloque, sino separados en múltiples tablas según su contexto, y luego se vinculan mediante relaciones lógicas.

Cada **tabla** representa una entidad o concepto (p. ej., *Empleados*, *Clientes*, *Productos*), y está compuesta por filas y columnas (también llamadas registros y campos). Todas las tablas de la base de datos en conjunto forman el esquema de la base de datos. Las tablas a su vez pueden relacionarse entre sí. Por ejemplo, es común tener una tabla principal de *Clientes* y otra tabla de *Pedidos*: cada pedido registrado en la tabla *Pedidos* incluye un identificador de cliente que referencia al registro correspondiente en la tabla *Clientes*. De este modo, los datos de clientes y pedidos permanecen separados pero vinculados lógicamente mediante ese identificador en común. Esta separación evita la duplicación de información y permite mantener la integridad: los datos del cliente se guardan solo en su tabla, y los pedidos simplemente hacen referencia a ellos.

La **estructura fundamental** de una base de datos relacional se puede resumir así: la base de datos contiene múltiples tablas (relaciones); cada tabla tiene **columnas** (campos) definidas con un tipo de dato específico, y múltiples **filas** (registros) que contienen los valores de datos. Según las características del modelo relacional:

- Una base de datos se compone de varias tablas, y **no pueden existir dos tablas con el mismo nombre**.

- Cada tabla tiene un conjunto definido de columnas (su *esquema* o *estructura*), y cada fila de la tabla corresponde a una instancia de esa entidad (un registro único) con un valor para cada columna.
- Es fundamental el concepto de **clave primaria** (**primary key**): una columna (o combinación de columnas) cuyas valores identifican de manera única a cada fila de la tabla. Por ejemplo, una tabla *Empleados* podría tener una columna `empleado_id` como clave primaria, de forma que no haya dos empleados con el mismo ID. La clave primaria asegura la **integridad de datos** en cuanto a unicidad de registros.
- Otro concepto clave son las **claves foráneas** (**foreign keys**), que implementan las relaciones entre tablas. Una clave foránea es una columna en una tabla "hija" que almacena valores que corresponden a la clave primaria de una tabla "padre". Por ejemplo, en la tabla *Pedidos*, puede haber una columna `cliente_id` que es clave foránea referenciando al `id` (clave primaria) de la tabla *Clientes*. Esto establece un vínculo relacional: ningún pedido tendrá un `cliente_id` que no exista en la tabla de clientes, garantizando la consistencia referencial. Las claves foráneas permiten las **relaciones uno-a-muchos** (un cliente puede tener muchos pedidos, pero cada pedido corresponde a un único cliente), entre otros tipos de relaciones.
- Además de claves, las bases de datos relacionales utilizan **restricciones de integridad** para asegurar la validez de los datos. Por ejemplo, podemos definir que una columna no acepte valores nulos (`NOT NULL`), o imponer restricciones de unicidad (`UNIQUE`) aparte de la clave primaria, entre otras reglas de negocio. Estas restricciones actúan como "reglas" que la base de datos hace cumplir automáticamente al insertar o actualizar datos.

Otra característica de las bases de datos relacionales es que operan bajo el principio ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) para transacciones, lo que garantiza que las operaciones de modificación de datos se realicen de forma confiable. Aunque los detalles de ACID y transacciones se verán más adelante en el libro, vale la pena saber que los sistemas relacionales se distinguen por mantener la integridad y consistencia de los datos incluso en escenarios de fallos o accesos concurrentes.

En resumen, una base de datos relacional organiza la información en tablas conectadas por relaciones lógicas. Esto permite modelar estructuras complejas de datos de manera coherente y eliminar redundancias. Casi todos los sistemas modernos de bases de datos relacionales usan el lenguaje **SQL (Structured Query Language)** para interactuar con los datos desde consultar información hasta insertar, borrar o modificar registros. En las siguientes secciones profundizaremos en cómo están compuestas las tablas y cómo escribir consultas básicas SQL para extraer datos de ellas.

Tablas, filas, columnas y tipos de datos comunes en SQL

Como hemos mencionado, las **tablas** son el elemento central del modelo relacional. A continuación definiremos con más detalle qué son las tablas, sus filas y columnas, y revisaremos los tipos de datos más comunes que podemos usar al definir esquemas en SQL.

- **Tabla:** es un conjunto de datos organizados en una estructura de *matriz* de dos dimensiones, con columnas verticales y filas horizontales. Cada tabla representa una entidad o categoría de datos. Una tabla de base de datos relacional contiene una colección de información sobre un tema específico. **Las tablas constan de filas y columnas**. Por ejemplo, podríamos tener una tabla llamada **Empleados** que registre información de los empleados de una empresa; las columnas podrían ser atributos como *ID*, *Nombre*, *Puesto*, *Salario*, *FechaContratación*, etc., y cada fila de la tabla corresponde a un empleado (un conjunto completo de valores, uno por columna, que describen a ese empleado).
- **Columna (campo):** es la unidad vertical en la tabla, identifica un atributo de la entidad representada por la tabla. Cada columna tiene un **nombre** único dentro de la tabla y está definida con un **tipo de dato** que determina qué clase de valores puede contener. Por ejemplo, en la tabla Empleados podríamos tener una columna `nombre` de tipo texto para almacenar el nombre del empleado, o una columna `salario` de tipo numérico para el salario. Todas las filas de la tabla comparten las mismas columnas, lo que significa que cada registro de empleado tendrá valores para las columnas *nombre*, *puesto*, *salario*, etc. Las columnas a veces se llaman **campos** o **atributos**.
- **Fila (registro o tupla):** es cada entrada horizontal en la tabla, que agrupa un valor por cada columna, representando una instancia individual de la entidad. Siguiendo el ejemplo, una fila de la tabla Empleados contendría los datos de un empleado específico: su ID, su nombre, su puesto, su salario, etc. Cada fila es un registro completo. Las filas no tienen un orden intrínseco definido a menos que se especifique en una consulta (el ordenamiento se logra mediante consultas, no por cómo se almacenan las filas).

Para ilustrar esto, veamos un ejemplo concreto de una tabla llamada **Empleados** con algunas filas de datos simulados:

Tabla **Empleados** (estructura y datos de ejemplo):

id	nombre	puesto	salario	fecha_contratacion
1	Ana López	Ventas	35000	2019-03-15
2	Luis Gómez	Desarrollo	42000	2020-07-01
3	Marta Ruiz	Ventas	38000	2018-11-20
4	Juan Pérez	Soporte	30000	2021-01-10
5	Laura García	Desarrollo	51000	2017-09-05

En esta tabla de ejemplo, tenemos 5 filas, cada una representando a un empleado, y 5 columnas que describen atributos de los empleados:

- **id:** identificador único de cada empleado (clave primaria de esta tabla, de tipo entero).
- **nombre:** nombre del empleado (cadena de texto).
- **puesto:** el departamento o rol del empleado (cadena de texto, por ejemplo "Ventas", "Desarrollo", "Soporte").
- **salario:** el salario anual del empleado (número, posiblemente entero o decimal).

- **fecha_contratacion:** la fecha en que fue contratado el empleado (tipo fecha).

Notemos que en este ejemplo simple, `id` es único para cada fila (1, 2, 3, 4, 5) – sería la clave primaria. La columna `puesto` tiene valores repetidos para "Ventas" y "Desarrollo", lo cual es normal ya que varios empleados pueden tener el mismo puesto. La información está organizada de tal forma que cada dato atómico (una celda específica, p. ej. el salario de Luis Gómez) está en la intersección de una fila con una columna.

Tipos de datos en SQL

Al definir las columnas de una tabla, debemos especificar el **tipo de dato** que contendrá cada una. El tipo de dato restringe el tipo de valores que se pueden almacenar en esa columna (por ejemplo, números enteros, cadenas de texto, fechas, valores booleanos, etc.). Usar los tipos de datos correctos es fundamental para mantener la integridad de los datos y optimizar el almacenamiento. SQL estándar define una variedad de tipos, y cada sistema de base de datos puede implementar tipos adicionales o variaciones. A continuación describimos algunos de los tipos más comunes que encontrarás en MySQL, PostgreSQL y otros sistemas SQL:

Tipo de dato	Descripción	Ejemplos de valores
INT (ENTERO)	Número entero (por lo general de 32 bits, sin parte decimal). Ideal para contadores, IDs, cantidades que no requieren decimales.	0, 1, -42, 10000
DECIMAL(p,s)	Número decimal de precisión fija, con *p* dígitos en total y *s* dígitos decimales. Útil para valores monetarios o medidos donde se requiere exactitud decimal.	DECIMAL(10,2): 12345.67 (p=7,s=2), 999.99
VARCHAR(n)	Cadena de texto de longitud variable hasta *n* caracteres. Se utiliza para almacenar textos cortos o medianos, como nombres, descripciones, etc. El límite *n* define la longitud máxima.	'Ana', 'Proyecto X', 'abc123'
CHAR(n)	Cadena de texto de longitud fija de *n* caracteres. Útil en casos especiales donde todas las entradas tienen exactamente la misma longitud, de lo contrario suele preferirse VARCHAR.	CHAR(2): 'ES', 'US'
TEXT	Cadena de texto de longitud muy grande (enorme), para almacenar textos extensos sin especificar un tamaño máximo fijo (por ejemplo comentarios, cuerpo de artículos). **Nota:** MySQL distingue subtipos como TEXT, MEDIUMTEXT, etc., y PostgreSQL usa TEXT sin longitud máxima.	(Texto libre, podría ser párrafos enteros)
DATE	Fecha (año, mes y día). Formato típico	'2019-12-31'

Tipo de dato	Descripción	Ejemplos de valores
	'YYYY-MM-DD'. Solo contiene la fecha sin hora.	
TIME	Hora (formato hora:minuto:segundo). Almacena un horario del día o duración temporal.	'23:59:59', '08:30:00'
DATETIME / TIMESTAMP	Fecha y hora combinadas (año, mes, día, hora, minuto, segundo). DATETIME es utilizado en MySQL para fechas/hora sin conversión de zona horaria, TIMESTAMP (en MySQL y en PostgreSQL) además puede almacenar automáticamente en UTC y convertir según la zona horaria. Ambos sirven para instantes de tiempo precisos.	'2025-07-29 10:45:00'
BOOLEAN o BOOL	Valor booleano lógico (verdadero/falso). En SQL estándar se espera TRUE o FALSE. PostgreSQL implementa un tipo boolean nativo. MySQL lo soporta como sinónimo de TINYINT(1) (donde 1 significa TRUE y 0 FALSE)secoda.co.	TRUE, FALSE (o 1, 0 en MySQL)
FLOAT / DOUBLE	Números de coma flotante de precisión simple o doble, para valores reales de magnitud grande o cuando se permiten pequeñas imprecisiones (ej. medidas científicas). *No se recomiendan para montos de dinero debido a errores de redondeo; para eso es mejor DECIMAL*.	3.14159, -0.001, 1.2e5
BLOB (Binary Large Object)	Tipo para datos binarios arbitrarios (imágenes, archivos binarios, etc.). Se almacena como secuencia de bytes. Hay variantes como TINYBLOB, MEDIUMBLOB, etc.	(datos binarios, ej. 0xFFD8... para imagen JPEG)

Estos son solo algunos ejemplos. Existen muchos más tipos: por ejemplo, PostgreSQL tiene tipos avanzados como `JSON/JSONB` para datos JSON, `UUID` para identificadores únicos globales, arreglos, tipos geométricos, etc. MySQL y otros sistemas también agregan sus propios. Sin embargo, **los tipos listados arriba son los fundamentales y más comunes** en prácticamente cualquier base de datos SQL (enteros, decimales, texto, fechas, booleanos).

Al crear una tabla, elegir el tipo correcto para cada columna garantiza almacenamiento eficiente y validación apropiada. Por ejemplo, usar `INT` para edades o cantidades, `VARCHAR(100)` para nombres (asumiendo que 100 caracteres es un máximo razonable), `DATE` para fechas de nacimiento, etc., ayuda a la base de datos a entender la naturaleza de los datos y aplicar reglas (no podrías insertar letras en una columna INT, o una fecha inválida en una columna DATE).

A modo de ejemplo, veamos cómo definiríamos en SQL la tabla *Empleados* que describimos antes, con sus columnas y tipos de datos apropiados:

```
sql
CREATE TABLE Empleados (
   id              INT         PRIMARY KEY,
   nombre           VARCHAR(100) NOT NULL,
   puesto           VARCHAR(50)  NOT NULL,
   salario          DECIMAL(10,2),
   fecha_contratacion DATE
);
```

En esta sentencia SQL DDL (Data Definition Language):

- Definimos una tabla llamada `Empleados`.
- La columna `id` es de tipo INT y se declara `PRIMARY KEY` (clave primaria), lo que automáticamente impone que sus valores sean únicos y no nulos.
- `nombre` es VARCHAR(100), indicando que puede almacenar hasta 100 caracteres, y `NOT NULL` significa que el nombre es obligatorio (no se permite que un empleado no tenga nombre).
- `puesto` es VARCHAR(50) `NOT NULL`, asumiendo que cada empleado debe tener un puesto asignado.
- `salario` es DECIMAL(10,2): esto permite cantidades con hasta 10 dígitos en total, de los cuales 2 son decimales (por ejemplo, 99999999.99 sería el máximo si p=10,s=2). No pusimos NOT NULL, así que podría haber empleados cuyo salario no esté registrado (valor NULL).
- `fecha_contratacion` es DATE, para guardar solo la fecha. Si no sabemos la fecha de contratación de un empleado, podría quedar como NULL (no especificamos NOT NULL).

> ***Nota:*** *SQL distingue entre NULL (un valor nulo, desconocido o no aplicable) y valores cero o cadenas vacías. Cuando no se pone NOT NULL en una definición de columna, significa que esa columna permite NULLs. Es importante entender que NULL no es "0" ni "vacío", sino la ausencia de valor. Muchas consultas requieren un cuidado especial al tratar con NULL (por ejemplo usando operadores específicos como IS NULL o IS NOT NULL en la cláusula WHERE, ya que = NULL no funciona como uno esperaría). En nuestros ejemplos iniciales, no profundizaremos en NULL, pero tenlo presente al diseñar esquemas.*

Con las tablas definidas y los tipos de datos explicados, ya podemos almacenar información estructurada. Lo siguiente es aprender a **consultar** esa información usando SQL, que es la tarea principal del día a día al trabajar con bases de datos. Pasemos a las consultas básicas.

Consultas básicas en SQL: SELECT, WHERE, ORDER BY, LIKE, BETWEEN

Una vez que tenemos datos almacenados en tablas, necesitamos extraerlos o consultarlos de diversas formas. La sentencia más básica y fundamental de SQL para leer datos es **SELECT**, que permite especificar qué columnas queremos ver y de qué tabla. A SELECT normalmente se le agregan cláusulas para filtrar filas (**WHERE**), para ordenarlas (**ORDER BY**), o para buscar patrones de texto (**LIKE**), o valores dentro de rangos (**BETWEEN**), entre otras posibilidades.

En esta sección nos enfocaremos en construir consultas simples para leer datos de una tabla. Usaremos de referencia la tabla *Empleados* que presentamos anteriormente para ejemplos prácticos. Escribir consultas es como formular preguntas a la base de datos; dominando estas sentencias básicas podremos obtener información valiosa de nuestras tablas.

Sentencia SELECT: leyendo datos de una tabla

La sintaxis general de una consulta básica es:

```sql
SELECT <columnas>
FROM <tabla>
[WHERE <condición>]
[ORDER BY <columna> [ASC|DESC]];
```

Donde:

- `SELECT` indica que vamos a realizar una consulta de selección de datos.
- `<columnas>` es la lista de columnas que queremos ver en el resultado (se separan por comas). Si queremos seleccionar *todas* las columnas de la tabla, usamos `SELECT *` (asterisco significa "todas las columnas").
- `FROM` especifica la tabla de donde obtendremos los datos.
- La cláusula `WHERE` es opcional; si se incluye, filtra las filas que cumplen cierta condición (lo veremos en detalle pronto).
- La cláusula `ORDER BY` también es opcional; se usa para ordenar las filas resultado según una o más columnas, ya sea en forma ascendente (`ASC`, la predeterminada) o descendente (`DESC`).
- Las partes entre corchetes [ ] son opcionales y dependen de lo que necesite la consulta.

Ejemplo 1: Obtener todos los registros y columnas de la tabla *Empleados*.
Supongamos que queremos listar todos los empleados con toda su información:

```sql
SELECT *
FROM  Empleados;
```

Esta consulta selecciona (*) todas las columnas de la tabla Empleados, devolviendo cada fila tal cual está almacenada. Con los datos de ejemplo, el resultado sería:

id	nombre	puesto	salario	fecha_contratacion
1	Ana López	Ventas	35000	2019-03-15
2	Luis Gómez	Desarrollo	42000	2020-07-01
3	Marta Ruiz	Ventas	38000	2018-11-20
4	Juan Pérez	Soporte	30000	2021-01-10
5	Laura García	Desarrollo	51000	2017-09-05

Como se puede observar, la consulta sin cláusula WHERE devuelve **todas** las filas de la tabla (en un orden no garantizado, que suele ser el orden de inserción, pero esto puede variar). Muchas veces no necesitaremos todas las columnas; en esos casos es mejor listar solo las columnas necesarias para reducir la salida.

Ejemplo 2: Obtener solo los nombres y puestos de todos los empleados:

```sql
SELECT nombre, puesto
FROM Empleados;
```

Esto retornaría únicamente dos columnas:

nombre	puesto
Ana López	Ventas
Luis Gómez	Desarrollo
Marta Ruiz	Ventas
Juan Pérez	Soporte
Laura García	Desarrollo

La sintaxis es bastante sencilla: listamos las columnas `nombre` y `puesto` después de SELECT, así obtenemos solo esa información.

Nota: El orden de las columnas en la salida seguirá el orden en que las enumeremos en la consulta. Además, podemos renombrar columnas en la salida usando alias (por ejemplo `SELECT nombre AS NombreEmpleado`...), pero en esta introducción mantendremos los nombres originales.

Ahora que sabemos seleccionar columnas y filas, veamos cómo **filtrar** cuáles filas queremos.

Filtrar resultados con WHERE

La cláusula **WHERE** se usa para especificar condiciones que deben cumplir las filas que queremos en el resultado de la consulta. Solo las filas que evalúen la condición como verdadera (TRUE) pasarán el filtro. Las que no la cumplan serán descartadas de la salida.

En una cláusula WHERE podemos usar operadores de comparación (`=`, `<>` o `!=`, `>`, `<`, `>=`, `<=`), operadores lógicos (`AND`, `OR`, `NOT`), así como operadores especiales como `BETWEEN` o `LIKE` (que veremos más adelante), entre otros. También podemos combinar varias condiciones.

Ejemplo 3: Supongamos que queremos listar únicamente los empleados cuyo puesto sea "Ventas". En nuestra tabla, eso corresponde a filtrar las filas donde la columna `puesto` tenga el valor 'Ventas'. La consulta sería:

```sql
SELECT id, nombre, puesto, salario
FROM Empleados
WHERE puesto = 'Ventas';
```

(Notamos que aquí listamos columnas específicas en SELECT para no traer la fecha de contratación, solo por brevedad del ejemplo.)

La condición `puesto = 'Ventas'` hace que solo se consideren aquellas filas donde el campo puesto sea exactamente el texto "Ventas". El resultado esperado sería:

id	nombre	puesto	salario
1	Ana López	Ventas	35000
3	Marta Ruiz	Ventas	38000

Como se observa, solo Ana y Marta tienen `puesto = 'Ventas'`. Las demás filas se excluyeron por no cumplir la condición.

La comparación de texto en SQL es por defecto **sensible a mayúsculas?** Depende del collation y del sistema. En MySQL, las comparaciones de texto típicamente *no distinguen mayúsculas* (collation predeterminado case-insensitive), de modo que 'ventas' = 'Ventas' = 'VENTAS' en un `WHERE` textual. En PostgreSQL, las comparaciones de texto *sí distinguen mayúsculas/minúsculas* por defecto, así que 'Ventas' != 'ventas'. Ten en cuenta estas diferencias según el motor que uses. Si necesitas que no distinga mayúsculas en PostgreSQL, puedes usar funciones como `LOWER()` para normalizar, o collations *CI* específicos. Pero nos desviamos; retornemos a las bases.

Ejemplo 4: Podemos usar múltiples condiciones combinadas con `AND` (todas deben ser ciertas) o con `OR` (cualquiera puede ser cierta). Por ejemplo, imaginemos que quisiéramos empleados del departamento de *Desarrollo* **y** que ganen más de 50000. O empleados que estén en *Ventas* **o** en *Soporte*. Veamos uno:

Consulta para obtener empleados de *Desarrollo* con salario superior a 50000:

```sql
SELECT nombre, puesto, salario
FROM Empleados
WHERE puesto = 'Desarrollo'
  AND salario > 50000;
```

La condición compuesta asegura que ambas cosas se cumplan. En nuestra tabla de ejemplo, `puesto = 'Desarrollo'` lo cumplen Luis y Laura, pero además `salario > 50000` solo lo cumple Laura García (51000). Así que la salida sería solo Laura.

Si utilizáramos OR, por ejemplo *empleados en Ventas* **o** *con salario > 50000* (`WHERE puesto = 'Ventas' OR salario > 50000`), obtendríamos quienes cumplan cualquiera de esas, etc.

Ejemplo 5: Podemos filtrar por valores de texto, numéricos, fechas, etc. Por ejemplo, empleados contratados después del 1 de enero de 2020:

```sql
SELECT nombre, fecha_contratacion
FROM Empleados
WHERE fecha_contratacion > '2020-01-01';
```

Aquí comparamos fechas. Suponiendo el formato 'YYYY-MM-DD' que entiende SQL, se obtendrían los empleados contratados después de esa fecha. En la tabla de ejemplo:

- Ana (2019-03-15) no cumple (es anterior),
- Luis (2020-07-01) **sí** cumple (es posterior a 2020-01-01),
- Marta (2018-11-20) no,
- Juan (2021-01-10) sí,
- Laura (2017-09-05) no.

Resultado esperado: Luis Gómez (2020-07-01) y Juan Pérez (2021-01-10).

Vemos la potencia del WHERE: con una simple condición podemos reducir drásticamente el conjunto de datos retornado a solo lo que nos interesa. Es común también usar condiciones más complejas, paréntesis para agrupar lógicas (`WHERE (cond1 AND cond2) OR cond3`, etc.), pero por ahora mantendremos condiciones sencillas.

Importante: Al comparar cadenas de texto en SQL para igualdad (como `puesto = 'Ventas'`), normalmente se debe escribir la cadena completa y exacta que se busca. Si se quiere hacer búsqueda parcial (por ejemplo, que el nombre *contenga* cierta subcadena, o empiece por algo), eso no se logra con `=` sino con el operador `LIKE` que veremos en breve.

Para concluir esta subsección, recordemos que **WHERE filtra filas según un criterio**. Si la cláusula WHERE se omite, la consulta devuelve todas las filas de la tabla. Si se incluye y ninguna fila cumple la condición, simplemente obtendremos cero filas de resultado (consulta válida pero vacía).

Ordenar resultados con ORDER BY

Cuando realizamos un SELECT, a menos que especifiquemos un orden, los registros pueden aparecer en cualquier secuencia (por lo general, el orden físico de inserción, pero esto no es garantizado ni parte del estándar). Para **ordenar** los resultados según uno o varios campos, utilizamos la cláusula **ORDER BY**.

La sintaxis básica es `ORDER BY <columna> [ASC|DESC]`. Podemos listar múltiples columnas si queremos ordenamientos secundarios. Por defecto, ASC (ascendente, de menor a mayor en números o alfabético A->Z en texto) es el orden usado. Si se desea descendente, se indica `DESC`.

Ejemplo 6: Supongamos que queremos obtener la lista de empleados del departamento de Ventas (como en el ejemplo anterior), pero ordenados por salario de mayor a menor, para ver primero quién gana más. La consulta:

```
sql
SELECT nombre, puesto, salario
FROM Empleados
WHERE puesto = 'Ventas'
ORDER BY salario DESC;
```

Combina lo que ya hicimos: filtra los `puesto = 'Ventas'` (Ana y Marta) y luego ordena los resultados por `salario` en orden descendente. Ana López tiene salario 35000 y Marta Ruiz 38000, así que Marta aparecerá antes que Ana en la salida debido al `DESC` (de mayor a menor):

nombre	puesto	salario
Marta Ruiz	Ventas	38000
Ana López	Ventas	35000

Si hubiésemos usado `ORDER BY salario ASC` (ascendente) o simplemente `ORDER BY salario`, el orden sería inverso (Ana primero, luego Marta).

Ejemplo 7: Otro caso, listar *todos* los empleados ordenados alfabéticamente por nombre:

```
sql
SELECT id, nombre, puesto
FROM Empleados
ORDER BY nombre ASC;
```

Salida ordenada por nombre de la A a la Z:

id	nombre	puesto
1	Ana López	Ventas
4	Juan Pérez	Soporte
5	Laura García	Desarrollo
2	Luis Gómez	Desarrollo
3	Marta Ruiz	Ventas

Aquí 'Ana' < 'Juan' < 'Laura' < 'Luis' < 'Marta' según el orden alfabético. Ten en cuenta que `ORDER BY` por defecto ordena considerando mayúsculas/minúsculas según la collation de la columna: en muchos casos "Laura" y "laura" serían equivalentes en orden (o podrían agruparse), esto es un detalle avanzado de collations.

También podemos ordenar por múltiples criterios. Ej: `ORDER BY puesto ASC, salario DESC` ordenaría primero por puesto (alfabético de departamento) y, dentro de cada puesto, por salario descendiente. Por ejemplo, Ventas y Desarrollo se agruparían, etc.

Ordenar es muy útil para reportes o para asegurar que la data viene en un cierto orden lógico. Sin ORDER BY, nunca asumamos un orden específico en la salida de una consulta SELECT.

Búsquedas parciales con LIKE

A veces necesitamos filtrar resultados no por igualdad exacta, sino por coincidencia parcial de texto. Por ejemplo: todos los empleados cuyo nombre *comienza con "Ma"* (María, Marcos, etc.), o que *contiene* cierta palabra. Para estos casos, SQL proporciona el operador **LIKE** dentro de la cláusula WHERE.

`LIKE` permite comparar una columna de texto con un **patrón** que puede incluir **comodines** (wildcards). Los comodines más utilizados en SQL son:

- **%** (porcentaje): representa *cualquier secuencia de caracteres* (de longitud variable, incluso cero caracteres).
- _ (guión bajo): representa *un solo caracter* cualquiera.

Con estos símbolos, podemos construir patrones. Por ejemplo:

- `'Ma%'` significa "empieza con 'Ma' seguido de cualquier secuencia de caracteres" (ejemplos que coincidirían: "Martín", "María", "Ma", "Mars", etc. Cualquier texto que comience por "Ma").
- `'%ana'` significa "termina con 'ana'" (coincidencias: "Ana", "Diana", "Montana", etc.).
- `'%lo%'` significa "contiene 'lo' en cualquier parte" (coincidencias: "López", "pablo", "Explorar", etc., notemos que la coincidencia también depende de mayúsculas según collation).

- '____' (cuatro guiones bajos) significa "exactamente cuatro caracteres cualesquiera". Por ejemplo, '201_' sería "empieza con 201 y luego un carácter cualquiera", útil para formatos fijos.
- Podemos combinar % y _ en un mismo patrón también, por ejemplo 'A_%_a' (un patrón algo raro que podría significar "empieza con A, luego cualquier caracter, luego cualquier secuencia, y termina con a").

El operador LIKE se utiliza en la cláusula WHERE en lugar de =. La sintaxis es:

```sql
WHERE columna LIKE 'patrón';
```

Y opcionalmente podemos negarlo con NOT LIKE para filtrar los que *no* coincidan con el patrón.

Ejemplo 8: Obtener todos los empleados cuyo nombre empieza con "Ma":

```sql
SELECT id, nombre
FROM Empleados
WHERE nombre LIKE 'Ma%';
```

En nuestra tabla de ejemplo, los nombres que comienzan con "Ma" son "Marta Ruiz". (Tenemos también "Laura", "Luis", "Ana", "Juan" que no cumplen). Entonces esta consulta debería devolver:

id	nombre
3	Marta Ruiz

Si tuviéramos "Mariana" o "Manuel" también aparecerían. Vemos que 'Ma%' captura tanto "Ma" seguido de cualquier cosa.

Ejemplo 9: Listar los empleados cuyo nombre contiene la subcadena "ar" (en cualquier posición):

```sql
SELECT id, nombre
FROM  Empleados
WHERE nombre LIKE '%ar%';
```

Patrón '%ar%' significa "tiene 'ar' en medio de la cadena, pudiendo tener cualquier cosa antes y después". Coincidencias en nuestros datos:

- "Marta Ruiz" contiene "ar".
- "Laura García" contiene "ar".
- "Carlos" (si hubiera algún Carlos) también. "Ana López" no tiene "ar", "Luis Gómez" no, "Juan Pérez" no.

Así que saldrían Marta y Laura, por ejemplo:

id	nombre
3	Marta Ruiz
5	Laura García

Ejemplo 10: Uso del guión bajo _. Supongamos que tuviéramos códigos de empleado con un formato fijo, por ejemplo "E-" seguido de tres dígitos (E-001, E-002, etc.). Si quisiéramos filtrar aquellos cuyo código termina en '5', podríamos usar `WHERE codigo LIKE 'E-__5'` (dos guiones bajos representan dos caracteres cualquiera, de modo que 'E-__5' casaría con "E-015", "E-125", etc. pero no con "E-25" porque faltaría un caracter ni con "E-0055" porque sobra uno).

En nuestro ejemplo de nombres no tiene mucho uso el _, así que consideremos otro caso: listar empleados cuyo nombre es de 4 letras. Podríamos hacer `WHERE nombre LIKE '____'` (4 guiones). En nuestros datos, "Juan" es de 4 letras, "Luis" también de 4 (sin contar apellido si estuvieran juntos). Sin embargo, dado que tenemos nombre y apellido en la misma columna separados por espacio, el patrón es más complejo. No profundizaremos en ese caso, pero es bueno saber que _ ayuda para buscar por longitud o patrones de un solo caracter.

En general, **LIKE es muy útil para búsquedas parciales o comodines**, y se utiliza muchísimo en consultas cuando se desconocen todos los caracteres. Por ejemplo, en aplicaciones se usa para autocompletar: `WHERE nombre LIKE 'Jo%'` para encontrar nombres que comiencen con "Jo".

Importante: En SQL estándar, LIKE es sensible o no a mayúsculas dependiendo del collation de la columna. En MySQL con collation predeterminado, "ANA" LIKE "ana" devuelve TRUE (porque no distingue mayúsculas). En PostgreSQL, "ANA" LIKE "ana" sería FALSE a menos que se use ilike (un operador no estándar que hace case-insensitive). Aquí asumiremos comparaciones case-insensitive para simplicidad. Siempre se puede hacer `WHERE LOWER(col) LIKE 'ana%'` para forzar ignorar mayúsculas, por ejemplo.

También, **el comodín % puede hacer que la búsqueda sea más lenta** en tablas grandes si no hay índices adecuados, especialmente si el patrón comienza con % (porque entonces no se puede aprovechar un índice de prefijo). Pero estos temas de rendimiento se analizan en partes avanzadas; por ahora, nos enfocamos en la funcionalidad.

Resumen: use **LIKE** para criterios de texto flexibles, usando % para varios caracteres y _ para uno solo[learnsql.es](). Por ejemplo, `WHERE apellido LIKE 'García%'` encontraría apellidos que empiezan por García (como "García", "Garciandía"). `WHERE nombre LIKE '%Luis%'` encontraría cualquier nombre que contenga "Luis" (podría traer "Luis", "San Luis", "Luisana" etc.).

Filtrar por rangos con BETWEEN

Para condiciones que implican verificar si un valor está **entre un rango** (por ejemplo, entre dos números, o entre dos fechas), SQL proporciona el operador **BETWEEN**. Este operador mejora la legibilidad en lugar de combinar condiciones con >= y <=. La sintaxis es:

```
sql
WHERE columna BETWEEN valor_inferior AND valor_superior
```

Esto es esencialmente equivalente a `columna >= valor_inferior AND columna <= valor_superior`, pero más conciso. Ten en cuenta que **BETWEEN es inclusivo** en ambos extremos del rango, es decir, incluye las filas cuyos valores sean exactamente iguales al límite inferior o superior.

Ejemplo 11: Listar los empleados cuyo salario está entre 30000 y 40000 (es decir, en el rango [30000, 40000], incluyendo ambos extremos):

```
sql
SELECT nombre, salario
FROM Empleados
WHERE salario BETWEEN 30000 AND 40000;
```

Esto capturará a cualquier empleado con salario de $30,000 hasta $40,000 inclusive. En nuestra tabla de ejemplo, veamos los salarios: Ana 35000 (sí entra), Luis 42000 (no, está encima de 40000), Marta 38000 (sí entra), Juan 30000 (sí entra, porque 30000 es el límite inferior incluido), Laura 51000 (no). Por tanto se obtendrían:

nombre	salario
Juan Pérez	30000
Ana López	35000
Marta Ruiz	38000

Obsérvese que Juan con exactamente 30000 apareció (límite inferior), y si tuviéramos alguien con 40000 exactos también aparecería (límite superior). Si quisiéramos excluir los límites, tendríamos que usar operadores > y < manualmente en lugar de BETWEEN, o la forma `NOT BETWEEN`.

Ejemplo 12: Filtrar por rango de fechas. Imaginemos que queremos todos los empleados contratados durante el año 2019 (desde el 1 de enero de 2019 hasta el 31 de diciembre de 2019 inclusive). Podemos hacer:

```
sql
SELECT nombre, fecha_contratacion
FROM Empleados
WHERE fecha_contratacion BETWEEN '2019-01-01' AND  '2019-12-31';
```

Between aquí funciona con fechas dadas en el formato correcto. En nuestra tabla de ejemplo, la única fecha en 2019 es Ana López (2019-03-15), por lo que ella sería el resultado. Si algún otro empleado tuviera fecha ese año, también saldría.

Ejemplo 13: Podemos usar BETWEEN con caracteres (cadenas) según el orden alfabético o collation, pero su uso principal es con números y fechas. Un ejemplo con texto: `WHERE apellido BETWEEN 'A' AND 'L'` podría traer apellidos que comienzan con A hasta L (dependiendo de la collation esto puede ser útil en listados alfabéticos por rango). Aun así, su semántica en texto es comparar cadenas en orden lexicográfico.

Recordemos: **BETWEEN X AND Y incluye X e Y**. Si necesitamos excluirlos, podríamos hacer `WHERE campo > X AND campo < Y`. Además, si `valor_inferior` es mayor que `valor_superior`, la consulta típicamente no devuelve filas (en algunos SQL podría dar error lógicamente). Así que asegúrate de poner el menor primero.

BETWEEN también se puede negar con `NOT BETWEEN` para tomar valores fuera de un rango dado.

Con lo anterior, hemos cubierto las consultas básicas de lectura en SQL:

- `SELECT ... FROM ...` para escoger columnas de una tabla.
- Cláusula `WHERE` para filtrar por condiciones (igualdad, comparaciones, etc.).
- Cláusula `ORDER BY` para ordenar los resultados.
- Operador `LIKE` (en WHERE) para búsquedas por patrones en texto.
- Operador `BETWEEN` (en WHERE) para filtrado por rangos.

Cada uno de estos elementos se puede combinar. Por ejemplo, podríamos hacer una consulta más compleja que use todo: *"Obtener el nombre y salario de los empleados del departamento de Ventas con salario entre 30000 y 40000, cuyos nombres empiecen por 'M', ordenados por salario descendente"*:

```
sql
SELECT nombre, salario
FROM Empleados
WHERE puesto = 'Ventas'
  AND nombre LIKE 'M%'
  AND salario BETWEEN 30000 AND 40000 ORDER BY salario DESC;
```

Esa sería una única consulta conjuncionando varios criterios y un orden final. (En nuestro dataset de ejemplo: puesto Ventas tenemos Ana y Marta, de ellas la que empieza por M es Marta Ruiz, y su salario 38000 está entre 30000 y 40000, por lo que Marta sería la única en el resultado.)

Como ves, SQL permite encadenar condiciones y modificaciones a la consulta básica para afinar exactamente la información que necesitamos. Estas son las bases de las consultas; más adelante en el libro veremos cláusulas adicionales (JOIN para

combinar tablas, GROUP BY para agregaciones, funciones de agregación como COUNT, SUM, etc., subconsultas, y muchas más características de SQL). Pero habiendo dominado SELECT junto con WHERE/ORDER BY/LIKE/BETWEEN, ya se puede hacer un amplio rango de consultas útiles sobre una sola tabla.

Antes de cerrar esta sección, recalcamos la importancia de la sintaxis y la lógica:

- La cláusula FROM define de dónde vienen los datos.
- SELECT qué columnas mostrar.
- WHERE qué filas cualificar.
- ORDER BY cómo presentarlas ordenadas.

El orden de las cláusulas en la sentencia es siempre SELECT -> FROM -> WHERE -> (opcional otras como GROUP BY/HAVING) -> (opcional ORDER BY) en SQL estándar. Memoriza este orden, ya que escribirlas en orden incorrecto producirá errores de sintaxis.

Finalmente, veamos unos **ejercicios prácticos** para afianzar estos conceptos, antes de continuar con más contenido en la siguiente parte del libro.

Ejercicios prácticos

A continuación se proponen algunos ejercicios relacionados con los temas cubiertos en esta Parte 1. Intenta resolverlos escribiendo las sentencias SQL o respuestas apropiadas. (Las soluciones se podrán consultar en el anexo de soluciones al final del libro).

1. **Instalación y verificación:** Después de instalar MySQL o PostgreSQL en tu sistema, ¿qué comando usarías en la terminal para comprobar la versión instalada del servidor de base de datos? *(Pista: busca un comando* `--version` *o el cliente por defecto.)*
2. **Consulta SELECT básica:** Considera la tabla **Empleados** usada en los ejemplos (con columnas `id, nombre, puesto, salario, fecha_contratacion`). Escribe una consulta SQL para obtener una lista de todos los empleados del departamento **"Ventas"** mostrando solo su nombre y salario. Los resultados deben aparecer ordenados por salario de mayor a menor.
3. **Búsqueda con LIKE:** Usando la misma tabla *Empleados*, escribe una consulta para encontrar todos los empleados cuyo nombre **contenga** la cadena "Luis" (ya sea al inicio, medio o final del nombre). Muestra el *id* y el *nombre* en los resultados.
4. **Filtro de rango:** Siempre con la tabla *Empleados*, escribe una consulta que muestre el nombre, puesto y fecha de contratación de aquellos empleados contratados **entre el 1 de enero de 2018 y el 31 de diciembre de 2019** (inclusive ambas fechas). Ordena la lista por fecha de contratación ascendente (del más antiguo al más reciente).
5. **Desafío adicional:** Imagina que tienes una tabla **Clientes** con columnas `cliente_id, nombre, ciudad`. También tienes una tabla **Pedidos** con columnas `pedido_id, fecha, cliente_id, monto`. ¿Qué tipo de relación existe entre estas dos tablas y qué columna actuaría como clave foránea? Escribe una sentencia

SELECT (simplificada) que combinaría datos de ambas tablas para obtener el nombre del cliente y el monto de cada pedido. *(Este último punto es conceptual y de adelanto, se puede responder en palabras o pseudocódigo SQL, ya que las JOIN se cubrirán en la siguiente parte.)*

Capítulo 2: Operaciones CRUD completas

Operaciones CRUD (Create, Read, Update, Delete) en SQL

En esta sección profundizaremos en las operaciones fundamentales para manipular datos en bases de datos relacionales, comúnmente referidas por las siglas **CRUD** (del inglés *Create, Read, Update, Delete*, que en español significa **Crear, Leer, Actualizar, Eliminar**). Estas cuatro operaciones cubren el ciclo de vida básico de los datos: **crear** nuevos registros, **leer** o consultar registros existentes, **actualizar** datos ya almacenados y **eliminar** datos que ya no se necesitan. En el contexto de SQL, cada operación CRUD se realiza mediante sentencias específicas del lenguaje:

- **Create (Crear)**: Se lleva a cabo con la sentencia `INSERT`, que permite **insertar** nuevos registros en una tabla.
- **Read (Leer)**: Se realiza con la sentencia `SELECT`, que permite **consultar** o leer los datos existentes (esta operación de lectura se ha tratado a profundidad en la parte 1 de este libro).
- **Update (Actualizar)**: Se efectúa con la sentencia `UPDATE`, usada para **modificar** los valores de registros existentes.
- **Delete (Eliminar)**: Se logra con la sentencia `DELETE`, empleada para **borrar** registros de una tabla.

A continuación, nos enfocaremos en detallar las operaciones de **inserción, actualización y eliminación** de datos (INSERT, UPDATE, DELETE), ya que la operación de lectura (SELECT) fue abordada previamente. Veremos la sintaxis general de cada comando y sus variaciones, ejemplificando cada caso con la tabla de ejemplo **Clientes**. La tabla *Clientes* consta de las columnas: `cliente_id` (identificador único de cliente), `nombre`, `email`, `ciudad` y `fecha_registro`. Usaremos esta estructura para demostrar cómo agregar nuevos clientes, cómo modificar los datos de clientes existentes y cómo eliminar registros, todo ello acompañado de buenas prácticas y señalando **errores comunes** que conviene evitar. Al final de la sección se proponen ejercicios prácticos (sin resolver) para afianzar lo aprendido.

Insertar datos con INSERT

La sentencia `INSERT` se utiliza para **crear nuevos registros** en una tabla, es decir, agregar filas de datos. Cada vez que necesitemos almacenar información nueva (por ejemplo, un nuevo cliente en nuestro sistema), emplearemos una consulta `INSERT`. Es parte del lenguaje de manipulación de datos (DML) de SQL. Antes de insertar datos, debemos asegurarnos de que la tabla de destino existe y de conocer qué columnas requiere y qué restricciones tienen (por ejemplo, columnas obligatorias, tipos de datos, valores por defecto, etc.).

Sintaxis básica de INSERT: La sintaxis general para insertar una nueva fila es la siguiente:

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

- **nombre_tabla**: es el nombre de la tabla en la que deseamos insertar el nuevo registro.
- **(columna1, columna2, ..., columnaN)**: es la lista de columnas para las cuales vamos a suministrar valores. Esta lista es opcional pero **altamente recomendada**; si se omite, la sentencia debe proporcionar un valor para **todas** las columnas de la tabla (en el orden exacto en que fueron definidas en la creación de la tabla).
- **VALUES (valor1, valor2, ..., valorN)**: especifica los valores que se insertarán correspondientes a cada columna listada. El número de valores debe coincidir con el número de columnas especificadas, y cada valor debe ser del **tipo de dato** apropiado para la columna destino (por ejemplo, cadenas entre comillas para columnas de texto, números sin comillas para columnas numéricas, fechas en formato adecuado para columnas de fecha, etc.).

Variaciones de la sintaxis INSERT: Además de la forma básica, existen algunas variaciones y consideraciones útiles al insertar datos:

- *Inserción sin lista de columnas:* Es posible escribir `INSERT INTO nombre_tabla VALUES (val1, val2, ...);` omitiendo la lista de columnas, **solo si** vamos a insertar un valor para **cada columna** de la tabla y en el orden exacto de definición de las columnas. No obstante, **no se recomienda omitir la lista de columnas**, ya que puede llevar a errores si la estructura de la tabla cambia o si se desconoce el orden exacto de las columnas. Es más seguro y legible especificar siempre las columnas afectadas.
- *Inserción de valores por defecto o nulos:* Si una columna tiene un valor por defecto definido en la tabla, podemos **omitir** esa columna en la lista para que el valor por defecto se asigne automáticamente al nuevo registro. De manera similar, si una columna permite valores nulos (NULL) y no tenemos un valor para ella, podemos ya sea omitir la columna (si no es obligatoria) o incluirla explicitando `NULL` en la lista de VALUES. Por ejemplo, para insertar un cliente sin conocer su `email`, podríamos poner el valor `NULL` en la posición correspondiente a la columna email. Siempre debemos asegurarnos de que la columna admite nulos; de lo contrario, la inserción generará un error.
- *Inserción de múltiples filas:* SQL permite insertar **varias filas a la vez** con una sola sentencia `INSERT`, separando cada tupla de valores con comas. Por ejemplo:

  ```sql
  INSERT INTO Clientes (nombre, email, ciudad, fecha_registro)
  VALUES
    ('Ana González', 'ana.gonzalez@example.com', 'Barcelona', '2025-07-01'),
    ('Luis Fernández', 'luis.fernandez@example.com', 'Madrid', '2025-07-02');
  ```

  Esta única instrucción agregaría dos filas nuevas en la tabla *Clientes*. La capacidad de insertar múltiples filas en un solo comando es útil para cargas

masivas de datos, ya que reduce el número de sentencias ejecutadas y puede mejorar el rendimiento.

- *Inserción a partir de una consulta (INSERT...SELECT):* Otra variación poderosa es usar los resultados de una consulta SELECT para insertar datos en una tabla. En vez de proporcionar valores fijos, se inserta el conjunto de resultados que devuelve un `SELECT`. Por ejemplo, podríamos copiar clientes de una tabla a otra tabla de respaldo así:

  ```sql
  INSERT INTO Clientes_backup (cliente_id, nombre, email, ciudad, fecha_registro)
  SELECT cliente_id, nombre, email, ciudad, fecha_registro
  FROM Clientes
  WHERE ciudad = 'Madrid';
  ```

  Este comando tomaría todos los clientes de la ciudad "Madrid" desde la tabla original *Clientes* e insertaría esos registros en la tabla *Clientes_backup*. La lista de columnas en el `INSERT` debe corresponder con las columnas seleccionadas en el `SELECT`. Esta técnica es muy útil para tareas como copias de seguridad lógicas, migración de datos o transformación de datos en bloque.

A continuación, veamos ejemplos concretos de inserciones en la tabla *Clientes*. Supondremos que la tabla *Clientes* ya existe con las columnas mencionadas (`cliente_id`, `nombre`, `email`, `ciudad`, `fecha_registro`). También asumiremos que `cliente_id` es la **clave primaria** de la tabla, y que probablemente sea un identificador único autogenerado (auto-incremental). En caso de ser auto-incremental, normalmente se omitiría esta columna en el INSERT para que el gestor de base de datos asigne el siguiente ID disponible automáticamente. No obstante, en los ejemplos incluiremos explícitamente el `cliente_id` con fines ilustrativos (imaginando que conocemos el identificador o que la columna no es auto-incremental), salvo que se indique lo contrario.

Ejemplo 1: Insertar un único cliente nuevo en la tabla *Clientes*.

Supongamos que queremos agregar un nuevo cliente con los siguientes datos: ID de cliente `101`, nombre "Carlos López", email "carlos.lopez@example.com", ciudad "Toledo" y fecha de registro "2025-07-29". La sentencia SQL para hacerlo sería:

```sql
INSERT INTO Clientes (cliente_id, nombre, email, ciudad, fecha_registro)
VALUES (101, 'Carlos López', 'carlos.lopez@example.com', 'Toledo', '2025-07-29');
```

Analicemos paso a paso esta instrucción de inserción:

1. **INSERT INTO Clientes (cliente_id, nombre, email, ciudad, fecha_registro)**: Indica que vamos a insertar datos en la tabla `Clientes`. A continuación del nombre de la tabla, especificamos entre paréntesis las columnas en las que vamos

a insertar valores. En este caso, listamos todas las columnas de la tabla *Clientes*: el identificador del cliente, el nombre, el email, la ciudad y la fecha de registro.

2. **VALUES (101, 'Carlos López', 'carlos.lopez@example.com', 'Toledo', '2025-07-29')**: Proporciona los **valores** para cada columna listada anteriormente. Estos valores aparecen en el mismo orden que las columnas:
   a) `101` corresponde a **cliente_id** (un número entero único para el cliente Carlos López).
   b) `'Carlos López'` corresponde a **nombre** (una cadena de texto con el nombre del cliente; obsérvese que las cadenas se colocan entre comillas simples `'...'` en SQL).
   c) `'carlos.lopez@example.com'` corresponde a **email** (otra cadena de texto que representa el correo electrónico).
   d) `'Toledo'` corresponde a **ciudad** (cadena de texto con la ciudad de residencia del cliente).
   e) `'2025-07-29'` corresponde a **fecha_registro** (fecha en formato **AAAA-MM-DD**; va entre comillas porque las fechas normalmente se tratan como cadenas con formato específico o literales de fecha según el SQL de cada sistema).

Al ejecutar esta sentencia, se añadirá una nueva fila en la tabla *Clientes* con los datos proporcionados. Es decir, habremos **creado** un nuevo registro de cliente. Tras la inserción, podríamos verificar la operación con una consulta SELECT (por ejemplo, `SELECT * FROM Clientes WHERE cliente_id = 101;`) para confirmar que el registro fue agregado correctamente.

Ejemplo 2: Insertar múltiples clientes en una sola sentencia.

Ahora imaginemos que tenemos dos nuevos clientes para registrar a la vez. Uno es Ana Gómez (email "ana.gomez@example.com", de ciudad "Madrid") y otro es Luis Prado (email "luis.prado@example.com", de ciudad "Barcelona"), ambos registrados el día 2025-08-01. Podemos insertar los dos registros con un solo comando `INSERT` como se muestra a continuación:

```
sql
INSERT INTO Clientes (cliente_id, nombre, email, ciudad, fecha_registro)
VALUES  (102, 'Ana Gómez', 'ana.gomez@example.com', 'Madrid', '2025-08-01'),
  (103, 'Luis Prado', 'luis.prado@example.com', 'Barcelona', '2025-08-01');
```

Lo que sucede aquí es que tras la cláusula `VALUES` hemos incluido **dos conjuntos** de valores, cada uno entre paréntesis y separado por una coma. Cada conjunto de valores representa una fila a insertar:

- La primera tupla `(102, 'Ana Gómez', 'ana.gomez@example.com', 'Madrid', '2025-08-01')` inserta el registro del cliente Ana Gómez con ID 102.
- La segunda tupla `(103, 'Luis Prado', 'luis.prado@example.com', 'Barcelona', '2025-08-01')` inserta el registro del cliente Luis Prado con ID 103.

Al ejecutar esta sentencia, ambos registros se agregarán a la tabla *Clientes* simultáneamente. Es más eficiente que ejecutar dos sentencias separadas `INSERT` por cada fila, especialmente en escenarios donde necesitemos cargar muchos registros.

Errores comunes al insertar datos:

- **Número de valores no coincide con columnas:** Si la lista de columnas especificada no tiene la misma cantidad de valores proporcionados tras `VALUES`, SQL devolverá un error. Por ejemplo, si enumeramos 5 columnas pero solo damos 4 valores (o viceversa), la inserción fallará. Siempre se debe asegurar que cada columna listada tenga un valor correspondiente.
- **Omisión de columnas obligatorias:** Si omitimos la lista de columnas esperando usar valores por defecto, pero la tabla contiene columnas *NOT NULL* sin valor por defecto, la sentencia fallará. Un error común es tratar de insertar un registro sin proveer un valor para una columna obligatoria (por ejemplo, olvidar insertar `email` cuando `email` no admite NULL ni tiene un DEFAULT definido).
- **Tipo de dato incompatible:** Cada valor debe ser del tipo adecuado. Un error típico es intentar insertar una cadena de texto en una columna numérica o un formato de fecha inválido en una columna de fecha. Esto provocará errores de conversión o rechazo por parte del motor de base de datos. Por ejemplo, insertar `'abc'` en una columna definida como INTEGER dará error de tipo de datos.
- **Falta de comillas en valores de texto o fecha:** Olvidar poner comillas alrededor de los valores de tipo *cadena* (VARCHAR, CHAR, TEXT, etc.) o *fecha* es un error de sintaxis común. Por ejemplo `VALUES (104, Juan, juan@example.com, Madrid, 2025-08-01)` es incorrecto, debería ser `VALUES (104, 'Juan', 'juan@example.com', 'Madrid', '2025-08-01')`.
- **Violación de restricciones (constraints):** Si la inserción viola alguna restricción de integridad de la base de datos, esta será rechazada. Por ejemplo, insertar un valor duplicado en una columna *PRIMARY KEY* o *UNIQUE* (como un `cliente_id` que ya existe) causará un error. Otro caso es intentar insertar un valor en una columna referenciada por una clave foránea que no exista en la tabla padre correspondiente (violación de foreign key).
- **Longitud de datos excedida:** Intentar insertar datos más largos que la longitud máxima definida para la columna. Por ejemplo, si `ciudad` está definido como VARCHAR(50) y se intenta insertar una cadena de 100 caracteres, la base de datos podría truncar el valor con warning o rechazar la inserción (dependiendo de la configuración).
- **Uso incorrecto de NULL:** Usar la palabra clave `NULL` en un contexto incorrecto. Por ejemplo, escribir `'NULL'` (entre comillas) en vez de `NULL` sin comillas insertará la cadena literal "NULL" en lugar de un valor nulo, lo cual es un error lógico. También, como se mencionó, tratar de insertar `NULL` en una columna que no lo permite causará error.

Buenas prácticas para INSERT:

- **Especificar las columnas**: Incluye siempre la lista de columnas en el `INSERT`. Esto hace las consultas más claras y resistentes a cambios en la estructura de la tabla. Además, permite insertar en un subconjunto de columnas dejando que valores por defecto se apliquen a las demás.
- **Orden y tipos de datos correctos**: Verifica que el orden de los valores corresponda exactamente con el orden de las columnas listadas y que cada valor

tenga el tipo de dato esperado. Esto evita errores de tipo y asegura que los datos entren en las columnas correctas.

- **Manejo de valores por defecto y nulos**: Aprovecha los valores por defecto definidos en la tabla para columnas que puedan no recibir dato en la inserción. Si un campo opcional no tiene dato, es preferible **omitir la columna** en el INSERT (si tiene default) o usar NULL explícitamente en el VALUES (si se permite), en vez de insertar datos ficticios o vacíos. Nunca dejes columnas obligatorias sin valor.
- **Inserciones masivas eficientes**: Para insertar muchos registros, prefiere usar inserciones por lotes (por ejemplo, múltiples filas en un solo INSERT como vimos) o métodos de carga masiva proporcionados por el SGBD (como utilidades de *bulk load* o importación de CSV). Esto es más eficiente que ejecutar miles de sentencias individuales.
- **Transacciones en inserciones múltiples**: Si vas a insertar numerosos registros relacionados, considera encapsular las inserciones dentro de una **transacción**. De este modo, si algo falla a mitad de camino, puedes revertir (ROLLBACK) la transacción completa y evitar datos inconsistentes (por ejemplo, mitad de los registros insertados y la otra mitad no). Además, en algunos motores de base de datos, agrupar operaciones en una transacción mejora el rendimiento al no confirmar cada operación por separado.
- **Seguridad y sanitización**: Si las sentencias SQL se construyen desde una aplicación o entrada de usuario, evita la **inyección SQL** usando consultas preparadas o parámetros en lugar de concatenar cadenas directamente. Esto es especialmente importante en inserciones, para que datos proporcionados por usuarios no rompan la sintaxis SQL ni causen daños.
- **Verificación posterior**: Después de una inserción, verifica que los datos se guardaron correctamente. Puedes usar una consulta SELECT para comprobar el nuevo registro, o si estás en un entorno de programación, verificar el número de filas afectadas que retorna la operación de inserción. Confirmar la operación es parte de mantener la integridad de los datos.

Actualizar datos con UPDATE

La sentencia UPDATE se utiliza para **modificar los valores** de columnas en registros existentes. En lugar de agregar nuevas filas, UPDATE nos permite cambiar información de filas que ya están almacenadas. Por ejemplo, si un cliente actualiza su dirección de correo electrónico o si necesitamos corregir una ciudad mal escrita en sus datos, utilizaremos una sentencia UPDATE en la tabla correspondiente.

Es fundamental usar UPDATE con precaución, ya que por defecto esta sentencia puede afectar **múltiples registros a la vez**. Siempre debemos especificar claramente **qué filas se van a actualizar** mediante una cláusula WHERE apropiada; de lo contrario, sin WHERE, la actualización se aplicará a **todas** las filas de la tabla (lo cual generalmente *no* es lo deseado).

Sintaxis básica de UPDATE: La forma general de una consulta de actualización es:

```sql
UPDATE nombre_tabla
SET columna1 = valor1, columna2 = valor2, ...
WHERE condición;
```

- **nombre_tabla**: es el nombre de la tabla cuyas filas queremos actualizar.
- **SET columna1 = valor1, columna2 = valor2, ...**: después de la palabra clave SET se enumeran las asignaciones de nuevos valores a las columnas que deseamos modificar. Podemos actualizar una o varias columnas a la vez, separando cada asignación con comas. Cada *columna* se iguala a un *valor* nuevo (que puede ser un literal, una expresión, el resultado de una subconsulta, etc., siempre que sea compatible con el tipo de dato de la columna).
- **WHERE condición**: opcional pero crítica en la mayoría de los casos. Es una expresión que filtra qué filas serán actualizadas. Solo las filas que cumplan la condición se verán afectadas por el UPDATE. Si se omite WHERE, **todas las filas** de la tabla serán actualizadas con los valores indicados, algo que usualmente solo se hace intencionalmente en casos muy particulares. Por lo general, siempre incluiremos una condición para limitar el alcance de la modificación (por ejemplo, actualizar un cliente específico por su cliente_id, o todos los clientes de una cierta ciudad, etc.).

Veamos cómo utilizar UPDATE con ejemplos en la tabla *Clientes*.

Ejemplo 1: Actualizar un solo registro identificado por su clave primaria.

Supongamos que el cliente con cliente_id = 101 (Carlos López) cambió de ciudad de residencia de "Toledo" a "Madrid". Para reflejar este cambio en nuestros datos, debemos actualizar la columna **ciudad** de ese cliente en la tabla *Clientes*. La sentencia SQL sería:

```sql
UPDATE Clientes
SET ciudad = 'Madrid'
WHERE cliente_id = 101;
```

Analicemos la consulta:

1. **UPDATE Clientes**: Especifica que vamos a actualizar la tabla Clientes.
2. **SET ciudad = 'Madrid'**: Indica que la columna **ciudad** debe tomar el valor 'Madrid'. Podemos actualizar varias columnas separándolas con coma (por ejemplo, SET ciudad = 'Madrid', email = 'nuevo@mail.com' si quisiéramos cambiar también el email), pero en este caso solo cambiamos la ciudad.
3. **WHERE cliente_id = 101**: Limita la actualización únicamente a aquellas filas cuyo cliente_id sea 101. Dado que cliente_id es clave primaria, esta condición se refiere a lo sumo a una fila específica. Solo el cliente Carlos López cumple esa condición, por lo que solo su registro se verá afectado. Si omitiéramos el WHERE, **todos** los clientes tendrían su ciudad cambiada a "Madrid", lo cual sería un error grave en este contexto. Por eso, insistimos: nunca olvidar la cláusula WHERE a menos que realmente se necesite actualizar todas las filas.

Tras ejecutar esta sentencia, los datos del cliente 101 reflejarán la nueva ciudad "Madrid". Ningún otro registro habrá cambiado. Conviene comprobar la actualización con un SELECT: por ejemplo, `SELECT nombre, ciudad FROM Clientes WHERE cliente_id = 101;` debería mostrarnos "Carlos López | Madrid" confirmando el cambio.

Ejemplo 2: Actualizar múltiples registros que cumplen una condición.

Consideremos ahora que la empresa decide corregir un error en la información de varios clientes. Supongamos que todos los clientes cuya ciudad está registrada como "Bcn" (abreviatura incorrecta de Barcelona) deben actualizarse para que la ciudad aparezca como "Barcelona" completo. Podemos hacer un único UPDATE que afecte a todas las filas necesarias:

```sql
UPDATE Clientes
SET ciudad = 'Barcelona'
WHERE ciudad = 'Bcn';
```

Explicación de esta sentencia:

- **UPDATE Clientes**: tabla objetivo de la actualización.
- **SET ciudad = 'Barcelona'**: nueva asignación para la columna ciudad.
- **WHERE ciudad = 'Bcn'**: condición que selecciona todas las filas donde la ciudad actualmente almacenada sea exactamente "Bcn". Podrían ser varios registros (todos aquellos clientes cuya ciudad esté mal escrita de esa forma). Cada fila que cumpla esto se actualizará, cambiando "Bcn" por "Barcelona". Las filas que ya tengan "Barcelona" o cualquier otra ciudad no se tocan.

Este comando actualizará potencialmente **múltiples filas a la vez** (todos los clientes en "Bcn"). Es una de las fortalezas de SQL: poder actualizar en masa con una sola instrucción declarativa. Nuevamente, verificar el resultado es útil: por ejemplo, un `SELECT COUNT(*) FROM Clientes WHERE ciudad = 'Bcn'` antes y después del UPDATE nos confirmaría que antes había X registros con "Bcn" y después del UPDATE debería haber 0, habiéndose transformado todos a "Barcelona".

Vale la pena mencionar que las condiciones en el WHERE pueden ser tan complejas como se requiera, incluyendo múltiples columnas (`WHERE ciudad = 'Bcn' AND nombre LIKE 'Luis%'`, por ejemplo), operadores de comparación, rangos, subconsultas, etc., para determinar precisamente los registros a modificar. Incluso se pueden hacer *updates condicionales* usando expresiones CASE dentro del SET, aunque eso es más avanzado. Siempre, el objetivo es que el WHERE identifique correctamente las filas que necesitan el cambio.

Errores comunes al actualizar datos:

- **Olvidar la cláusula WHERE:** Este es quizá el error más peligroso. Si se ejecuta `UPDATE Clientes SET ciudad = 'Madrid';` sin WHERE, *todas* las filas de la tabla *Clientes* tendrán la ciudad cambiada a "Madrid". Recuperar los datos originales sería difícil si no hay respaldo o si no se usa una transacción que

permita deshacer. Por ello, repetir la recomendación: asegúrate de poner la condición adecuada en el WHERE. Muchos administradores de bases de datos habilitan un "modo seguro" en herramientas de cliente que previene ejecutar updates (o deletes) sin where, debido a este riesgo.

- **Condición mal escrita o demasiado amplia:** Un WHERE incorrecto puede ser tan problemático como no tener WHERE. Por ejemplo, confundir operadores lógicos (`AND` vs `OR`) podría actualizar registros equivocados. Si escribimos `WHERE nombre = 'Ana' OR apellido = 'Gómez'` pensando en actualizar solo a Ana Gómez, en realidad actualizaremos *todos* los clientes que se llamen Ana *además de* todos los apellidados Gómez, incluyendo gente que no sea "Ana Gómez". Siempre pruebe su condición con un SELECT primero para ver qué filas selecciona.
- **Error en nombres de columnas o valores:** Es relativamente común escribir mal el nombre de una columna en el SET o en el WHERE. Si la columna no existe, el SGBD arrojará un error de sintaxis o de columna desconocida. Si la columna existe pero no era la que queríamos (p. ej., confundir `ciudad` con `direccion`), podríamos estar modificando el dato equivocado. Asimismo, asignar un valor del tipo erróneo (p. ej., `'123'` entre comillas a una columna numérica) generará error, y usar comillas incorrectamente (como `SET nombre = Juan` sin comillas) también.
- **Actualizar varias tablas a la vez incorrectamente:** En SQL estándar, un `UPDATE` solo puede afectar a una tabla a la vez. Algunos motores ofrecen sintaxis especial para updates con JOIN (por ejemplo, MySQL permite `UPDATE t1 JOIN t2 ON ... SET t1.col = ... WHERE ...`), pero si se desconoce esto, un usuario podría intentar algo no soportado como `UPDATE Clientes, Pedidos SET Clientes.ciudad = 'X', Pedidos.status = 'Y' WHERE ...` lo cual es inválido en la mayoría de sistemas. La actualización multi-tabla debe hacerse con consultas separadas o usando subconsultas/joins adecuados en la misma tabla objetivo.
- **Violación de restricciones al actualizar:** Al igual que con INSERT, las constraints pueden dar problemas si un update intenta romperlas. Ejemplos: cambiar un valor de clave primaria a uno que ya existe (duplicando PK), o alterar una clave foránea a un valor que no tiene correspondencia en la tabla padre (violando integridad referencial). También actualizar un valor de columna única a uno que otro registro ya posee puede arrojar error de duplicado.
- **No verificar el alcance de la actualización:** A veces, un UPDATE puede afectar 0 filas (si la condición no encontró coincidencias) y eso puede pasar desapercibido. O bien afectar más filas de las previstas. No comprobar el **número de filas afectadas** es un error de omisión; siempre es bueno revisar el mensaje del SGBD (que suele decir "X rows updated") o usar métodos en aplicaciones para obtener ese conteo, asegurándose de que coincide con lo esperado.

Buenas prácticas para UPDATE:

- **Siempre incluir WHERE (cuando corresponda):** Reiteramos, salvo en casos extremadamente justificados (como reinicializar todos los registros de una tabla), siempre usa la cláusula WHERE para delimitar qué registros modificar. Aún más, procura que el WHERE use una condición que identifique un conjunto específico (por ejemplo, por clave primaria o por algún criterio bien acotado).
- **Previsualizar con SELECT:** Una táctica muy útil es primero escribir la cláusula WHERE en un `SELECT *` para ver qué registros serían afectados. Si el SELECT

devuelve exactamente las filas que deseas cambiar, entonces usas la misma condición para el UPDATE. Esto actúa como verificación previa para evitar actualizaciones accidentales.

- **Actualizaciones graduales o en lote:** Si necesitas actualizar un enorme número de filas (digamos millones), considera hacerlo en lotes más pequeños dentro de una transacción o con varias transacciones, para no bloquear la tabla demasiado tiempo ni sobrecargar los logs de transacciones. Por ejemplo, actualizar de 10000 en 10000 filas por iteración, usando una condición que delimite por rangos de la clave primaria.
- **Uso de transacciones:** Para actualizaciones críticas, realiza la operación dentro de una transacción explícita (`BEGIN TRANSACTION ... COMMIT`) de modo que puedas revisar los resultados (con SELECT) antes de confirmar los cambios. Si algo no luce bien, puedes hacer un `ROLLBACK` para deshacer la actualización.
- **Respaldo antes de cambios masivos:** Si vas a hacer una actualización global o muy amplia (por ejemplo, aumentar el precio de todos los productos en una tabla de millones de registros, o cambiar un código que afecta muchas filas), es prudente respaldar la información antes (ya sea con una copia de seguridad de la base, o copiando las filas afectadas a una tabla temporal) por si la lógica de cambio no era correcta y necesitas restaurar los datos originales.
- **Verificar filas afectadas:** Después de un UPDATE, comprueba cuántas filas fueron modificadas. Esto se puede hacer consultando la salida del comando (muchas interfaces SQL te dicen "X rows affected") o mediante funciones de la API si estás programando (como `rowCount()` en PDO, etc.). Si el número es distinto al esperado (ya sea 0 cuando esperabas 1, o 100 cuando esperabas 10), puede indicar que la condición no era la correcta o hubo duplicados no previstos.
- **Mantener integridad referencial:** Si actualizas valores que son claves foráneas en otras tablas (por ejemplo, cambias un código de cliente que es referenciado en tabla de pedidos), asegúrate de actualizar también las tablas relacionadas o de que haya *ON UPDATE CASCADE* definido en las claves foráneas para que el SGBD lo haga automáticamente. De lo contrario, podrías dejar datos huérfanos o inconsistentes.
- **Evitar lógica compleja en UPDATE si es posible:** A veces es tentador hacer cálculos muy elaborados en la propia sentencia UPDATE (subconsultas, funciones, etc.). Está bien usar expresiones (ej: `SET saldo = saldo * 1.10` para incrementar un saldo en 10%), pero si la lógica es compleja, podría ser más claro dividirla en pasos o calcular valores en una aplicación y luego pasarlos al UPDATE. Lo importante es mantener la claridad y asegurarse de que la expresión hace lo esperado para cada fila.
- **Uso correcto de alias y JOIN si se soporta:** Si tu SGBD soporta `UPDATE ... FROM ... JOIN ...` (como T-SQL de SQL Server o MySQL), úsalo cuidadosamente. Los alias pueden ayudar a hacer la sentencia más legible. Por ejemplo, en MySQL podrías hacer `UPDATE Clientes AS c JOIN Ciudades AS d ON c.ciudad = d.nombre_corto SET c.ciudad = d.nombre_completo WHERE d.pais = 'ES';` para actualizar nombres de ciudad basándose en otra tabla. En estos casos, revisa bien el join para no multiplicar filas accidentalmente.
- **No sobrepasar límites de la transacción:** Si se ejecuta un UPDATE gigantesco, podría haber riesgo de bloquéos prolongados o llenado de logs. Evaluar usar `DELETE+INSERT` (reemplazo) u otras estrategias puede ser útil en algunos escenarios avanzados, pero eso depende del caso de uso y el entorno.

Eliminar datos con DELETE

La sentencia DELETE se utiliza para **eliminar registros** de una tabla, borrando filas que ya no son necesarias o que deben ser suprimidas por algún motivo. Esta operación es destructiva y permanente: una vez confirmada, los datos borrados no se recuperan fácilmente (a menos que se tenga una copia de seguridad o no se haya hecho COMMIT en una transacción todavía). Por lo tanto, DELETE también requiere un uso cuidadoso, especialmente al igual que con UPDATE, en cuanto a limitar correctamente qué filas van a eliminarse usando la cláusula WHERE.

Sintaxis básica de DELETE: La forma general de la sentencia de borrado es:

```sql
DELETE FROM nombre_tabla WHERE condición;
```

- **DELETE FROM nombre_tabla**: indica la tabla de la cual se quieren eliminar registros. A diferencia de SELECT, en DELETE no se lista columnas después del nombre de la tabla, ya que se eliminará la fila completa. (Un error común de principiantes es escribir DELETE * FROM tabla, lo cual es incorrecto; la sintaxis correcta es siempre DELETE FROM seguido del nombre de la tabla, sin asterisco).
- **WHERE condición**: especifica qué filas deben eliminarse, usando una condición lógica igual que en el caso de UPDATE. Solo aquellas filas que cumplan la condición serán borradas. Si se omite la cláusula WHERE, **todas las filas de la tabla serán eliminadas**. Esto vacía la tabla pero deja su estructura intacta (equivalente a una purga completa de datos).

Al igual que con UPDATE, la cláusula WHERE en un DELETE es fundamental para evitar eliminar más datos de los deseados. A menos que la intención sea vaciar la tabla entera, siempre incluiremos una condición. Cabe destacar que DELETE fila por fila puede ser más lento que otras técnicas para eliminar todos los registros (como la sentencia TRUNCATE TABLE, que es DDL y elimina *todas* las filas de forma más eficiente), pero TRUNCATE tiene consideraciones especiales (no puede filtrarse por condición, puede requerir permisos adicionales y no es reversible mediante ROLLBACK en muchos sistemas). Por tanto, para eliminar filas específicas usamos DELETE con WHERE; para eliminar todo el contenido de una tabla rápidamente podemos usar TRUNCATE bajo las precauciones adecuadas.

Veamos ejemplos de eliminación usando la tabla *Clientes*.

Ejemplo 1: Eliminar un único registro identificado por ID.

Supongamos que el cliente con cliente_id = 103 (Luis Prado) solicita la eliminación de sus datos personales de nuestra base de datos (por derecho al olvido, por ejemplo). Para borrar por completo ese registro de la tabla *Clientes*, ejecutamos:

```sql
DELETE FROM Clientes
WHERE cliente_id = 103;
```

Análisis de la sentencia:

- **DELETE FROM Clientes**: señala que vamos a remover datos de la tabla Clientes.
- **WHERE cliente_id = 103**: limita la acción a la fila cuyo identificador sea 103. Nuevamente, siendo cliente_id clave primaria, esta condición apunta a lo sumo a una fila específica. Solo el cliente Luis Prado tiene cliente_id 103, por lo que solo su registro será eliminado.

Tras ejecutar el comando, la fila correspondiente a Luis Prado desaparecerá de la tabla. Conviene confirmarlo haciendo, por ejemplo, SELECT * FROM Clientes WHERE cliente_id = 103; que ya no debería devolver resultados. Si devolviera, es que la eliminación no ocurrió (posiblemente porque no se hizo COMMIT aún en caso de estar en una transacción, o porque la condición no coincidía exactamente).

Ejemplo 2: Eliminar múltiples registros con una condición.

Consideremos ahora que tenemos registros de clientes de prueba o duplicados que queremos limpiar. Supongamos que queremos eliminar *todos* los clientes cuyo email sea NULL o esté vacío, ya que asumimos que esos registros están incompletos o no son válidos. Podríamos hacerlo con una sentencia:

```
sql
DELETE FROM Clientes
WHERE email IS NULL OR email = '';
```

Explicación:

- **DELETE FROM Clientes**: tabla objetivo de la eliminación.
- **WHERE email IS NULL OR email = ''**: condición que abarca dos casos: (1) filas donde el email es valor nulo (IS NULL) y (2) filas donde el email es la cadena vacía ''. Dependiendo de cómo se almacenen los datos, un email vacío podría haberse guardado como una cadena vacía en lugar de un verdadero NULL, por eso cubrimos ambas posibilidades. Todas las filas que cumplan cualquiera de esas condiciones serán eliminadas. Puede que sean varias filas o ninguna, según los datos. Si, por ejemplo, tuviéramos 5 clientes sin email (3 con NULL y 2 con cadena vacía), los 5 serían borrados con esta sentencia en un solo paso.

Después de ejecutarla, la tabla *Clientes* ya no contendrá registros con email nulo o vacío. Es recomendable, como siempre, hacer un SELECT de verificación antes o después. En este caso, podríamos haber hecho antes SELECT cliente_id, nombre FROM Clientes WHERE email IS NULL OR email = ''; para listar a quiénes íbamos a borrar, y tras el DELETE hacer el mismo SELECT para confirmar que no queden tales registros.

Ejemplo 3: Eliminar **todas** las filas de una tabla (uso con precaución).

Si por alguna razón quisiéramos vaciar por completo la tabla *Clientes* (por ejemplo, reiniciar el contenido de una tabla temporal, o borrar datos de una entidad que ya no utilizaremos), podríamos emplear un DELETE sin WHERE:

```sql
DELETE FROM Clientes;
```

Advertencia: Esto eliminará **todos los registros** de *Clientes*. La tabla quedará vacía, aunque la estructura (columnas, índices, etc.) permanece. Esta operación es altamente destructiva y suele ser más eficiente realizarla con `TRUNCATE TABLE Clientes;` en la mayoría de SGBD, pero `TRUNCATE` no es transaccional (no se puede deshacer) y puede estar restringido. Usando `DELETE FROM Clientes;` tenemos la opción de envolverla en una transacción para poder revertir si fuese necesario. En cualquier caso, ejecutar un borrado masivo así requiere estar 100% seguro de la decisión y haber hecho, idealmente, un respaldo previo.

Errores comunes al eliminar datos:

- **Olvidar la cláusula WHERE:** Igual que con UPDATE, omitir el WHERE en un DELETE puede resultar catastrófico, ya que eliminará todo el contenido de la tabla. Un pequeño descuido (como ejecutar el comando equivocado o en la base de datos equivocada) puede llevar a pérdida masiva de datos. Siempre verifica dos veces que incluiste la condición correcta. Algunas interfaces SQL, por protección, requieren confirmación extra si detectan un DELETE sin WHERE.
- **Condición mal formulada:** Un OR/AND mal utilizado, paréntesis omitidos o condiciones equivocadas pueden borrar registros no deseados. Por ejemplo, `DELETE FROM Clientes WHERE ciudad = 'Madrid' AND ciudad = 'Barcelona';` no borrará nada (condición imposible), mientras que `DELETE FROM Clientes WHERE ciudad = 'Madrid' OR ciudad = 'Barcelona';` borrará *todos* los clientes de Madrid y Barcelona. Asegúrate de entender la lógica booleana de tu condición y pruébala con SELECT primero.
- **Dependencias referenciales (clave foránea):** Si la tabla *Clientes* está relacionada con otras tablas (por ejemplo, una tabla de *Pedidos* donde cada pedido referencia a un cliente), puede haber restricciones de clave foránea con acciones al borrar. Si *Clientes* es padre de otras tablas y la clave foránea está definida con `ON DELETE RESTRICT` (por defecto, si no se especifica CASCADE), intentar borrar un cliente que tiene pedidos asociados resultará en un error de integridad referencial: el SGBD impedirá la eliminación para no dejar pedidos huérfanos. Un error común es ignorar estas restricciones y no entender por qué el DELETE "no funciona". La solución en estos casos puede ser borrar primero los registros hijos (por ejemplo, borrar pedidos del cliente, si es lógico hacerlo) o haber definido la relación con `ON DELETE CASCADE` para que la base de datos elimine automáticamente los hijos cuando eliminas el padre. Sin embargo, el uso de CASCADE también puede llevar a eliminar más datos de los previstos, así que hay que manejarlo con cuidado.
- **Eliminar más o menos filas de las esperadas:** Si la condición es demasiado amplia, podrías borrar inadvertidamente datos necesarios; si es demasiado

estrecha o tiene un error, puede que no borres nada. Por ejemplo, `DELETE FROM Clientes WHERE nombre = 'Luis'` borraría *todos* los clientes llamados Luis (quizá varios, no solo uno). Si la intención era borrar a Luis Prado específicamente, debería haberse usado una identificación única (como `cliente_id = 103` en el ejemplo anterior). Por otro lado, un error de sintaxis como `WHERE nombre = 'Luis' AND 'Prado'` (falta comparar el apellido con algo) puede no arrojar error pero tampoco hace lo esperado, resultando en borrar a todos los "Luis" independientemente del apellido en algunos motores, o en no borrar nada en otros.

- **No confirmar la eliminación:** A diferencia de UPDATE donde ves "X rows affected", con DELETE también se informa cuántas filas se borraron. Ignorar este resultado puede ser peligroso. Uno debería verificar que el número de filas eliminadas coincide con el que esperaba. Si esperabas borrar 1 y ves "100 rows deleted", sabrás inmediatamente que algo anduvo mal en la condición.
- **No usar transacción cuando corresponde:** Si vas a borrar registros en varias tablas relacionadas manualmente (sin CASCADE, por ejemplo, primero borrar pedidos de un cliente, luego el cliente), un error sería no hacerlo bajo una transacción. Podrías borrar los pedidos y fallar al borrar el cliente, quedando en un estado inconsistente (cliente sin pedidos, o pedidos huérfanos si fue al revés). Con una transacción, aseguras que o se borra todo o no se borra nada ante un problema.
- **Confundir DELETE con DROP/TRUNCATE:** A veces los principiantes confunden comandos. `DELETE` elimina filas (datos), `DROP` elimina objetos de la base de datos (por ejemplo, `DROP TABLE Clientes` borraría la tabla completa, estructura y datos), `TRUNCATE` elimina **todas** las filas de una tabla de manera rápida y resetea ciertos contadores pero no borra la tabla en sí. Cada uno tiene su uso; usar uno por otro puede ser desastroso (ej.: hacer DROP en vez de DELETE, borrando la tabla entera del esquema, o usar DELETE cuando se quería solo truncar en un entorno controlado). Hay que elegir el comando adecuado para la necesidad.
- **Seguridad e inyección SQL:** Similar a los otros casos, si la condición `WHERE` se construye a partir de entrada de usuario (por ejemplo, desde una aplicación web: borrar un usuario por id pasado en la URL), es susceptible a inyección SQL si no se parametriza la consulta. Un atacante podría manipular la entrada para forzar un `DELETE FROM Clientes;--` sin where, por ejemplo, y borrar toda la tabla. Por eso, aunque es un error más de desarrollo que de SQL en sí, nunca está de más recordar la importancia de las consultas preparadas o sanitizar los inputs.

Buenas prácticas para DELETE:

- **Where específico y verificado:** Igual que con UPDATE, define claramente la condición de borrado. Prueba la condición primero con un SELECT para asegurarte de que solo las filas previstas aparecen. Por ejemplo, antes de borrar clientes sin email, puedes listar cuántos son y cuáles son, para estar seguro.
- **Uso de LIMIT (en motores que lo soportan):** Algunos sistemas (MySQL, PostgreSQL con extensión, etc.) permiten añadir `LIMIT` a un DELETE para limitar el número de filas borradas. Esto puede ser útil para eliminar en lotes controlados, por ejemplo, `DELETE FROM Logs WHERE fecha < '2022-01-01'`

LIMIT 10000; en un loop hasta limpiar todo un histórico muy grande, sin bloquear la tabla completa en una sola transacción gigantesca.

- **Transacciones en operaciones críticas:** Si la eliminación es parte de una serie de cambios relacionados, o si es potencialmente riesgosa, realizarla dentro de una transacción es aconsejable. Puedes así hacer un ROLLBACK si verificas que la condición estaba mal o si ocurre algún problema en medio. Por ejemplo, al borrar datos de varias tablas relacionadas manualmente, haz primero BEGIN, luego los DELETE en hijos, luego en padre, verifica conteos, y finalmente COMMIT. Si algo no coincide o falla, ROLLBACK.
- **Respaldo de información importante:** Antes de borrar datos sensibles o difíciles de recuperar, haz un respaldo. Si vas a depurar clientes viejos, quizá guárdalos en una tabla de *Clientes_historico* mediante un INSERT … SELECT como se vio antes, y luego bórralos de la principal. Así tienes un resguardo por si se necesita consultar después. Borrar sin backup debería reservarse a datos ciertamente desechables.
- **Cascada consciente:** Si utilizas claves foráneas con ON DELETE CASCADE, recuerda que borrar una fila padre desencadenará el borrado automático de todas las filas hijas relacionadas. Esto puede ser muy útil (ej: borrar un cliente automáticamente borra sus pedidos y direcciones asociadas en otras tablas) pero también peligroso si se hace sin plena conciencia. Revisa el esquema relacional antes de hacer deletes en tablas interconectadas para anticipar qué más podría borrarse en cascada. En entornos de desarrollo, prueba este comportamiento; en producción, asegúrate de querer realmente eliminar todo ese conjunto.
- **Preferir TRUNCATE para vaciar tablas enteras:** Si tu intención es eliminar **todos** los registros de una tabla y estás seguro, TRUNCATE TABLE suele ser más eficiente que DELETE masivo, ya que no registra borrado fila por fila sino que simplemente libera la storage de la tabla. Pero recuerda: Truncate es irreversible (no soporta rollback en muchos sistemas) y no dispara triggers de delete fila por fila, etc. Úsalo para limpieza total de tablas no referenciadas o donde la pérdida total de datos está contemplada.
- **Comprobar filas afectadas:** Siempre observa el resultado de filas borradas. Esto confirmará que la cantidad es la esperada. Si esperabas borrar 1 y dice 0, quizás la condición no coincidió (tal vez un typo en el valor). Si esperabas 1 y dice 5, cancela la transacción (si estás dentro de una) o revisa qué ocurrió, porque posiblemente tu condición abarcó más de lo deseado.
- **Políticas de retención y auditoría:** En contextos profesionales, a veces no se permite un DELETE físico inmediato de ciertos datos (por razones legales o de auditoría). En lugar de borrar, a veces se marca un registro como "inactivo" o "borrado lógico" mediante una columna de estado. Esto ya es diseño de aplicación, pero es una buena práctica considerar si realmente necesitas eliminar o solo ocultar datos. Si el libro aborda solo SQL puro, esto quizá excede, pero es bueno mencionarlo: muchos sistemas evitan deletes frecuentes en favor de un flag de borrado lógico, para no perder historiales.

Ejercicios prácticos

A continuación se plantean una serie de **ejercicios prácticos** para que el lector ponga en práctica los conceptos de operaciones CRUD aprendidos. Todos los ejercicios se basan en la tabla **Clientes (cliente_id, nombre, email, ciudad, fecha_registro)**

utilizada en los ejemplos. **Nota:** No se incluyen aquí las soluciones, solo el planteamiento; las respuestas se pueden consultar en los apéndices o solucionario una vez haya intentado resolverlos por su cuenta.

1. **Inserción de un nuevo cliente:** Suponga que acaba de registrarse un nuevo cliente llamado **María Pérez** con email `"maria.perez@example.com"`, residente en **Valencia**, cuyo registro es en la fecha actual. Escriba la sentencia `INSERT` necesaria para añadir este cliente a la tabla *Clientes*. (Use `cliente_id = 104` para este nuevo registro, asumiendo que continúa la secuencia de IDs de los ejemplos anteriores, e incluya la fecha del día de hoy en el formato correcto).
2. **Inserción múltiple de clientes:** La empresa obtuvo una lista de dos nuevos clientes para agregar al mismo tiempo:
   a) Juan Ojeda, email "juan.ojeda@example.com", de **Madrid**, con fecha de registro `2025-08-15`.
   b) Elisa Márquez, email "elisa.marquez@example.com", de **Barcelona**, con fecha de registro `2025-08-16`.

   Plantee una sola sentencia SQL que inserte **ambos** registros en la tabla *Clientes* con sus respectivos datos (asigne los IDs 105 y 106 respectivamente a Juan y Elisa en la inserción).

3. **Actualización de datos de cliente:** El cliente con `cliente_id = 102` (Ana Gómez) ha actualizado su correo electrónico y ahora es `"ana.gomez@nuevoemail.com"`. Además, se mudó a **Sevilla**. Escriba una sentencia `UPDATE` para modificar **ambos** campos (email y ciudad) de Ana Gómez en la tabla *Clientes*. Asegúrese de identificar correctamente a Ana por su ID en la cláusula WHERE.
4. **Eliminación de registros según condición:** Debido a nuevas políticas de datos, se deben eliminar de la tabla *Clientes* todos los clientes que se registraron **antes del año 2021** (es decir, con `fecha_registro` anterior al `2021-01-01`). Formule la sentencia `DELETE` que borre los registros correspondientes en *Clientes*. (Pista: compare la fecha_registro de cada cliente con la fecha indicada).
5. **Consulta de verificación (ejercicio de lectura):** *(Ejercicio adicional de "Read")* Después de realizar las operaciones anteriores, escriba una consulta `SELECT` para verificar cuántos clientes quedan en la tabla *Clientes* y listar sus nombres, ciudades y fecha_registro. *(Este ejercicio refuerza la operación de lectura y permite comprobar los efectos de las operaciones CRUD ejecutadas.)*

JOINS en SQL

¿Qué es un JOIN y para qué sirve?

En SQL, un **JOIN** es una operación que permite **combinar datos de dos o más tablas** en una sola consulta, basándose en **columnas relacionadas** entre ellas. Es una herramienta fundamental para trabajar con bases de datos **relacionales**, ya que en estos sistemas la información suele estar **normalizada** (distribuida en múltiples tablas para evitar duplicación). Mediante los JOINs podemos **reunir nuevamente** esos datos dispersos: por ejemplo, unir la tabla de clientes con la de pedidos para obtener información completa de quién realizó cada pedido, sin tener que almacenar todos esos datos en una sola tabla gigante. En resumen, los JOINs se utilizan:

- **Para combinar datos de distintas tablas** que comparten alguna relación lógica (como un identificador común).
- **Para evitar duplicación de datos** en una única tabla: en lugar de repetir información, se divide en tablas separadas (clientes, pedidos, etc.) y luego se unen bajo demanda.
- **Para respetar el modelo relacional (normalización)** y mantener la integridad de los datos.
- **Para hacer consultas más eficientes y legibles**: es más claro y potente obtener datos relacionados con un JOIN que, por ejemplo, usando subconsultas complejas o almacenando todos los datos redundantes en una tabla.
-

Un JOIN funciona comparando los valores de ciertas columnas en las tablas involucradas, según una condición de emparejamiento definida en la cláusula `ON`. Cuando los valores cumplen la condición (por ejemplo, un **cliente_id** que aparece en ambas tablas), se **fusionan** las filas de ambas tablas en el resultado. Si una fila no cumple la condición con ninguna fila de la otra tabla, dependerá del tipo de JOIN si aparece o no en el resultado (como veremos en detalle más adelante).

Es importante destacar que, si no se especifica lo contrario, un JOIN en SQL por defecto es de tipo **INNER JOIN** (también llamado **JOIN interno**). Además, existen varios tipos de JOIN que determinan **qué filas se incluyen o excluyen** según si encuentran o no coincidencia en la otra tabla. A continuación, revisaremos los tipos principales de JOIN en SQL y cómo utilizarlos.

Tipos de JOIN en SQL

Los JOINs más comunes en SQL son cuatro:

1. **INNER JOIN** (unión interna)
2. **LEFT JOIN** (unión externa izquierda, o LEFT OUTER JOIN)
3. **RIGHT JOIN** (unión externa derecha, o RIGHT OUTER JOIN)
4. **FULL JOIN** (unión externa completa, o FULL OUTER JOIN)

Cada tipo de JOIN determina qué conjuntos de filas se devuelven al combinar dos tablas, dependiendo de si hay **coincidencias** entre ellas en la columna (o combinación de columnas) usada para unir. En las siguientes secciones explicaremos cada tipo en detalle, con su sintaxis SQL, un **diagrama conceptual** para visualizar el conjunto de datos resultado, y ejemplos prácticos usando dos tablas de ejemplo: `Clientes` y `Pedidos`. Las tablas de ejemplo tienen la siguiente estructura simplificada:

- **Clientes**: contiene datos de los clientes, con campos `(cliente_id, nombre, email, ciudad)`.
- **Pedidos**: contiene datos de pedidos realizados, con campos `(pedido_id, cliente_id, fecha_pedido, total)`.

Supondremos que `Pedidos.cliente_id` es una **clave foránea** que refiere al cliente que realizó el pedido (relacionada con `Clientes.cliente_id`). Usaremos estas tablas para ilustrar cómo funciona cada tipo de JOIN.

INNER JOIN (Unión interna)

Un **INNER JOIN** devuelve **solo las filas que tienen coincidencia en ambas tablas**, según la condición de unión indicada. Esto significa que **excluye** cualquier dato que exista en una tabla y no tenga correspondencia en la otra. En términos de conjuntos, un INNER JOIN toma la **intersección** de las dos tablas, considerando la columna común.

Figura 1: Diagrama conceptual de un INNER JOIN. El área sombreada (intersección de los dos círculos que representan las tablas) indica las filas devueltas: solo aquellas presentes tanto en la tabla izquierda (ejemplo: Clientes) como en la tabla derecha (Pedidos).

En SQL, la sintaxis básica de un INNER JOIN (o simplemente JOIN) es la siguiente:

```
sql
SELECT <columnas>
FROM Tabla1
INNER JOIN Tabla2
ON Tabla1.columna_comun = Tabla2.columna_comun;
```

- INNER JOIN: indica el tipo de unión (interna). Puede abreviarse simplemente como JOIN ya que INNER es el tipo por defecto.
- La cláusula ON Tabla1.col = Tabla2.col especifica la condición de unión: normalmente una columna común que relaciona ambas tablas (p. ej., el ID del cliente en ambos lados). Solo las filas donde esta condición sea verdadera se combinarán.

Ejemplo práctico: Supongamos que queremos obtener una lista de todos los pedidos junto con el nombre y la ciudad de cada cliente que lo realizó. Usando nuestras tablas Clientes y Pedidos, escribiríamos una consulta con INNER JOIN así:

```
sql
SELECT
  Clientes.cliente_id,
  Clientes.nombre,
  Clientes.ciudad,
  Pedidos.pedido_id,
  Pedidos.fecha_pedido,
  Pedidos.total
FROM Clientes
INNER JOIN Pedidos
  ON Clientes.cliente_id = Pedidos.cliente_id;
```

En esta consulta:

- La tabla **Clientes** es la tabla izquierda (después de FROM) y **Pedidos** la derecha (después de JOIN).

- El predicado `ON Clientes.cliente_id = Pedidos.cliente_id` indica que solo queremos combinar filas donde el `cliente_id` de ambas tablas coincida (es decir, el pedido pertenece a ese cliente).

El resultado de un INNER JOIN **solo incluirá los clientes que tienen al menos un pedido** en la tabla de Pedidos. Cualquier cliente que no tenga pedidos **no aparecerá** en el resultado, y cualquier pedido cuyo `cliente_id` no tenga correspondencia en la tabla de Clientes tampoco aparecerá (aunque en un sistema bien diseñado, todos los pedidos deberían referenciar a un cliente existente). En nuestro ejemplo, obtendríamos un listado donde cada fila es la combinación de un cliente con uno de sus pedidos. Si un cliente tiene varios pedidos, ese cliente aparecerá repetido en múltiples filas (uno por pedido). Si un cliente no tiene ningún pedido, simplemente no contribuye ninguna fila al conjunto de resultados del INNER JOIN.

Uso típico: el INNER JOIN se utiliza cuando *solo* nos interesan los datos que **coinciden en ambas tablas**. Por ejemplo, para listar todas las ventas realizadas (que requieren combinar la tabla de ventas con la de clientes y productos), o para encontrar entidades que tienen correspondencia en dos listas diferentes. Es el tipo de JOIN más utilizado cuando se analizan relaciones **uno a muchos** (como clientes con sus pedidos) y se quieren **solo los casos donde la relación existe** (clientes que hicieron pedidos).

LEFT JOIN (Unión externa izquierda)

Un **LEFT JOIN** devuelve **todas las filas de la tabla izquierda**, y agrega los datos coincidentes de la tabla derecha. Si alguna fila de la izquierda **no tiene coincidencia** en la derecha, de todos modos aparecerá en el resultado, con valores NULL en las columnas provenientes de la tabla derecha. En otras palabras, el LEFT JOIN toma **todos los registros de la izquierda** y agrega lo de la derecha solo cuando hay correspondencia; las filas de la derecha que no coinciden son descartadas. Por eso se le llama también **unión externa izquierda**, ya que expande el resultado para incluir los "exteriores" (no emparejados) de la izquierda.

La sintaxis de LEFT JOIN es muy similar a la de INNER JOIN, solo cambiando la palabra clave `LEFT`:

```sql
SELECT <columnas>
FROM Tabla1
LEFT JOIN Tabla2
ON Tabla1.columna_comun = Tabla2.columna_comun;
```

Aquí `Tabla1` es la tabla izquierda cuyo contenido completo queremos conservar. `Tabla2` es la tabla derecha, de la cual solo obtendremos las filas que emparejen con algo de la izquierda.

Ejemplo práctico: Supongamos que queremos **listar todos los clientes**, junto con los detalles de sus pedidos si los tuvieran. Esto incluye a clientes que *no han hecho ningún pedido*, los cuales igualmente deben aparecer en el listado (y podríamos indicar de alguna forma que no tienen pedidos). La consulta con LEFT JOIN sería:

```sql
SELECT
  C.nombre,
  C.email,
  C.ciudad,
  P.pedido_id,
  P.fecha_pedido,
  P.total
FROM Clientes C
LEFT JOIN Pedidos P
  ON C.cliente_id = P.cliente_id;
```

En esta consulta hemos usado **alias** (`C` para Clientes y `P` para Pedidos) por claridad. El LEFT JOIN asegura que **todos los registros de `Clientes` aparezcan** al menos una vez en el resultado. Si un cliente tiene pedidos, cada pedido será una fila con los datos del cliente repetidos; si un cliente no tiene ningún pedido, aparecerá una única fila con sus datos y `NULL` en las columnas de pedido.

Por ejemplo, si "María Pérez" no tiene pedidos en la tabla `Pedidos`, igualmente aparecerá en el resultado con su nombre, email, ciudad, y en las columnas de pedido verá `NULL` (pedido_id NULL, fecha_pedido NULL, total NULL indicando que no hay pedido asociado). Esto ocurre porque esas filas de la tabla izquierda no encontraron pareja en la tabla derecha, y el LEFT JOIN rellena con NULL las columnas faltantes.

En cambio, un cliente como "Juan López" que tenga dos pedidos aparecerá en dos filas, una por cada pedido, repitiendo sus datos de cliente en ambas filas junto a los datos de cada pedido.

Uso típico: los LEFT JOIN son ideales para **incluir todos los registros de una tabla principal aunque no tengan relación en la segunda tabla**. Por ejemplo, listar todos los clientes incluyendo aquellos que no hayan realizado pedidos (para quizás detectar inactividad), o un informe de productos mostrando también los productos sin ventas. En general, se usa cuando queremos identificar elementos "que no tienen correspondencia" en la otra tabla, ya que estos aparecerán con valores nulos en los datos de la tabla derecha. Un patrón común es usar LEFT JOIN seguido de una condición `WHERE TablaDerecha.columna IS NULL` para **encontrar registros huérfanos**, por ejemplo: "clientes sin pedidos" (se hace LEFT JOIN de clientes con pedidos y luego se filtra donde `Pedidos.cliente_id` es NULL). Esto es más eficiente y expresivo que intentar lograr lo mismo con subconsultas.

RIGHT JOIN (Unión externa derecha)

Un **RIGHT JOIN** es esencialmente el espejo del LEFT JOIN: devuelve **todas las filas de la tabla derecha**, emparejándolas con las de la izquierda cuando hay coincidencias; si alguna fila de la derecha no tiene correspondencia en la izquierda, también aparecerá en el resultado con NULLs en las columnas de la tabla izquierda.

Las filas de la izquierda que no coincidan con ninguna de la derecha se descartan. En resumen, el RIGHT JOIN conserva **todos los registros de la tabla derecha**.

La sintaxis es análoga a los anteriores, usando RIGHT JOIN:

```sql
SELECT <columnas>
FROM Tabla1
RIGHT JOIN Tabla2
ON Tabla1.columna_comun = Tabla2.columna_comun;
```

En este caso, Tabla2 (la que está a la derecha del JOIN) es la tabla cuyos registros **se mantienen todos** en el resultado.

Ejemplo práctico: Si quisiéramos listar **todos los pedidos realizados**, incluyendo incluso aquellos que no tengan un cliente asociado (situación anómala, pero supongamos que podría pasar por errores de datos), usaríamos un RIGHT JOIN. Por ejemplo:

```sql
SELECT
  C.nombre,
  C.email,
  C.ciudad,
  P.pedido_id,
  P.fecha_pedido,
  P.total
FROM Clientes C
RIGHT JOIN Pedidos P
  ON C.cliente_id = P.cliente_id;
```

Aquí la tabla Pedidos es la derecha, por lo que el resultado contendrá **todos los pedidos** existentes. Para cada pedido, se anexa la información del cliente correspondiente si lo hay. Si algún registro de Pedidos tiene un cliente_id que no coincide con ningún cliente en Clientes (por ejemplo, un pedido cuyo cliente fue borrado de la tabla de clientes), igualmente el pedido aparecerá y los campos de cliente (nombre, email, ciudad) saldrán como NULL indicando que falta esa relación. Los pedidos válidos (con cliente existente) mostrarán normalmente los datos del cliente. Las filas de clientes que no tienen pedidos no aparecerían en absoluto en este resultado, porque el enfoque está en la tabla derecha.

En la práctica, el RIGHT JOIN es **menos utilizado que el LEFT JOIN**, ya que cualquier consulta con RIGHT JOIN puede reescribirse alternativamente intercambiando el orden de las tablas y usando LEFT JOIN. Por ejemplo, la consulta anterior podría haberse escrito empezando con la tabla Pedidos en el FROM y luego LEFT JOIN Clientes ... obteniendo el mismo resultado. Muchas personas prefieren esta estrategia para evitar confusiones. Además, no todos los motores de base de datos soportan RIGHT JOIN; por ejemplo, SQLite no lo implementa. En sistemas que carecen de RIGHT JOIN, simplemente se invierte la lógica utilizando LEFT JOIN.

Uso típico: el RIGHT JOIN se usa en situaciones similares al left join pero donde la prioridad es la tabla derecha. Un caso real podría ser, por ejemplo, si quisiéramos obtener un listado de *todas* las citas médicas registradas (tabla derecha), incluyendo aquellas cuyo paciente (tabla izquierda) ya no exista en el sistema. Como se mencionó, es más común reformular este caso como un LEFT JOIN inverso.

FULL JOIN (Unión externa completa)

Un **FULL JOIN** (o **FULL OUTER JOIN**) devuelve **todas las filas de ambas tablas**, combinándolas cuando hay coincidencias, y dejando NULL en las columnas faltantes cuando no las hay. Es conceptualmente la **unión** del resultado de un LEFT JOIN y un RIGHT JOIN: incluye todos los emparejamientos posibles *y* además incluye las filas no emparejadas de una tabla o de la otra. En términos de conjuntos, sería la **unión de ambos conjuntos completos**, marcando dónde no hubo intersección.

La sintaxis es:

```
sql
SELECT <columnas>
FROM Tabla1
FULL [OUTER] JOIN Tabla2
ON Tabla1.columna_comun = Tabla2.columna_comun;
```

(Notar que la palabra `OUTER` es opcional; `FULL JOIN` es suficiente en SQL estándar. Pocos sistemas usan sintaxis ligeramente distinta para lograr esto.)

Ejemplo práctico: Imaginemos que necesitamos un reporte maestro de **todas las personas y todas las transacciones** en un sistema, de modo que aparezcan: los clientes que hicieron pedidos con sus pedidos, *más* los clientes que no han hecho ninguno (pero igual listados), *más* cualquier pedido que por alguna razón no tenga cliente asociado. Esta situación híbrida requeriría un FULL JOIN entre `Clientes` y `Pedidos`. La consulta sería:

```
sql
SELECT
  C.cliente_id,
  C.nombre,
  C.ciudad,
  P.pedido_id,
  P.fecha_pedido,
  P.totalFROM Clientes C
FULL JOIN Pedidos P
  ON C.cliente_id = P.cliente_id;
```

Al ejecutar un FULL JOIN, el resultado contendrá:

- Las filas donde hubo coincidencia entre cliente y pedido (igual que un inner join, combinadas en una sola fila).

- Las filas de clientes sin pedido (datos de cliente con NULL en columnas de pedido).
- Las filas de pedidos sin cliente (datos del pedido con NULL en columnas de cliente).

Por ejemplo, si el Cliente #100 no tiene pedidos, aparecerá con sus datos y columnas de pedido nulas; si el Pedido #500 tiene un `cliente_id` que no existe en Clientes, ese pedido aparecerá con sus datos y los campos de cliente como NULL.

En nuestros datos de ejemplo, un FULL JOIN permitiría ver **todas** las filas posibles. Esta operación es útil para tareas de **reconciliación de datos** o **auditoría**, donde queremos identificar discrepancias entre dos tablas. Por ejemplo, podría emplearse para cotejar dos listas de registros provenientes de fuentes diferentes: un FULL JOIN mostraría claramente qué elementos están solo en la fuente A, cuáles solo en la fuente B, y cuáles en ambas.

No obstante, cabe mencionar que **no todos los sistemas de base de datos soportan FULL JOIN de forma nativa**. Por ejemplo, MySQL (hasta versiones recientes) no tenía FULL JOIN; en tales casos se emula combinando un LEFT JOIN y un RIGHT JOIN con UNION. En SQL Server, Oracle, PostgreSQL, etc., sí existe FULL JOIN directamente. En un contexto de libro como "La Biblia de SQL" consideramos la sintaxis estándar.

Uso típico: el FULL OUTER JOIN se usa cuando queremos **ver absolutamente todas las filas de dos tablas** y sus relaciones, incluyendo lo que no coincide. Es útil para informes integrales o para encontrar registros que están en una tabla pero faltan en la otra (por ejemplo, comparar dos listas de clientes de dos sistemas distintos, etc.). Dado que retorna muchos resultados (a lo sumo la suma de filas de ambas tablas, si ninguna coincide), se emplea con cuidado. Un caso práctico: supongamos que tenemos una tabla de *clientes actuales* y otra de *clientes antiguos* y queremos un listado de *todos los clientes que han existido alguna vez*, indicando si actualmente están activos o no. Un FULL JOIN entre ambas tablas, con columnas indicando origen, permitiría ver todos los clientes (los que solo están en antiguos saldrán con NULL en campos de actuales, y viceversa).

Casos de uso de los diferentes tipos de JOIN

Resumiendo lo anterior, podemos destacar casos de uso comunes para cada tipo de JOIN:

- **INNER JOIN:** Úselo cuando necesite **solo los registros con correspondencia en ambas tablas**. Ejemplos: obtener todas las órdenes con sus clientes y detalles (solo verá órdenes que tengan cliente y clientes que tengan órdenes); combinar tablas maestro-detalle donde solo interesan las entradas con datos en ambas (p.ej., productos con ventas registradas, estudiantes que están inscritos en cursos, etc.).
- **LEFT JOIN:** Adecuado para **incluir todo el conjunto principal** aunque la otra tabla no tenga datos relacionados. Ejemplos: listar todos los clientes y sus pedidos (incluyendo clientes sin pedidos, que aparecerán con valores nulos); encontrar elementos "huérfanos" en la tabla izquierda al filtrar los NULL de la derecha (clientes sin pedido, categorías sin productos, etc.). También útil para reportes

donde la tabla izquierda contiene todas las entidades y la derecha datos opcionales o adicionales.

- **RIGHT JOIN:** Útil cuando la prioridad es **conservar todos los registros de la tabla derecha**. En la práctica es equivalente a un LEFT JOIN invirtiendo el orden. Ejemplos: obtener todos los registros de log de eventos (tabla derecha) intentando emparejarlos con usuarios en la tabla izquierda, de modo que incluso eventos sin usuario conocido aparezcan; o listar todas las citas agendadas (derecha) con la información del profesional a cargo si existe (izquierda). Si su base de datos no soporta RIGHT JOIN directamente, recuerde que puede lograr lo mismo reordenando las tablas y usando LEFT JOIN.
- **FULL JOIN:** Se usa para **combinar conjuntos completos y analizar diferencias o complementariedad**. Ejemplos: conciliación de datos de dos sistemas distintos (ver qué registros están solo en uno, solo en otro, o en ambos); informes globales que necesiten incluir *todas* las entidades de dos tablas aunque no tengan relación (por ejemplo, un informe de todas las cuentas bancarias y todas las transacciones, para detectar cuentas sin transacciones y transacciones sin cuenta, si las hubiera). Es muy útil para tareas de calidad de datos y obtención de panorama completo, pero recuerde verificar que su motor SQL ofrezca FULL JOIN.

(Nota: Además de estos, existen otros tipos de JOIN más avanzados o especializados, como el CROSS JOIN (producto cartesiano), SELF JOIN (unir una tabla consigo misma), o NATURAL JOIN, entre otros. Esos van más allá del alcance de esta sección pero conviene conocerlos. El CROSS JOIN devuelve todas las combinaciones posibles entre dos tablas – se suele evitar salvo casos específicos. Un SELF JOIN permite relacionar filas dentro de la misma tabla, por ejemplo para jerarquías. En general, siempre defina cuidadosamente la condición de unión para evitar resultados inesperados.)

Buenas prácticas al usar JOINs

Al trabajar con JOINS en SQL, tenga en cuenta las siguientes **buenas prácticas**, que ayudarán a escribir consultas correctas, eficientes y fáciles de entender:

- **Determinar la tabla principal:** Antes de escribir la consulta, identifique cuál es la tabla "principal" o de referencia, es decir, aquella cuyos registros quiere conservar sí o sí en el resultado. Esto guía si debe usar LEFT o RIGHT JOIN. Por ejemplo, si desea listar todos los clientes con sus pedidos, *Clientes* es la tabla principal (izquierda) y usará LEFT JOIN con Pedidos.
- **Usar alias de tabla descriptivos:** Asigne **alias cortos** a las tablas (por ejemplo, `C` para Clientes, `P` para Pedidos) y utilícelos en los nombres de columna en la consulta. Esto mejora la legibilidad y evita ambigüedades cuando columnas de diferentes tablas tienen el mismo nombre. Por ejemplo: `SELECT C.nombre, P.total FROM Clientes C JOIN Pedidos P ON C.cliente_id = P.cliente_id;`. Sin alias, la consulta sería más verbosa y propensa a errores.
- **Especificar siempre la condición de unión (`ON`):** Asegúrese de incluir la cláusula `ON` con la condición apropiada de emparejamiento. **Nunca** omita el `ON` en un JOIN, ya que si lo hace, el motor SQL producirá un **producto cartesiano** (CROSS JOIN) que combina todas las filas de ambas tablas, resultando en muchísimas filas indeseadas. Por ejemplo, unir una tabla de 100 clientes con

1000 pedidos sin ON daría 100*1000 = 100 000 filas combinando todo con todo, algo casi siempre incorrecto.

- **Verificar las columnas comunes e índices:** Normalmente los JOINs se realizan sobre columnas que son **claves primarias o foráneas**. Verifique que las columnas usadas en `ON` tengan el mismo tipo de datos y contenido compatible. Además, es recomendable que esas columnas estén indexadas (por ejemplo, la clave foránea en Pedidos) para mejorar el rendimiento de la consulta, especialmente si las tablas son grandes. Los índices en columnas de JOIN pueden acelerar significativamente la combinación.
- **Comenzar probando con pocas columnas:** Al construir un JOIN complejo, primero escriba una versión simple de la consulta que seleccione solo las columnas esenciales (por ejemplo, solo los IDs o counts) para comprobar que el número de filas devuelto es el esperado. Una vez validado que el JOIN está correcto (por ejemplo, el número de filas coincide con lo esperado: en un inner join, no más que la menor de las tablas; en un left join, igual al total de la tabla izquierda, etc.), entonces agregue más columnas al SELECT según necesite.
- **Usar condiciones adicionales adecuadamente:** Si necesita filtrar resultados (cláusula `WHERE`) o agregar más criterios, tenga cuidado con *dónde* los coloca. Por ejemplo, con LEFT JOIN, si quiere filtrar por una columna de la tabla derecha, a menudo deberá hacerlo dentro de la condición `ON` o permitir valores NULL, de lo contrario el filtro en `WHERE` podría eliminar las filas sin correspondencia. Ejemplo: para listar todos los clientes y sus pedidos de 2023, incluyendo clientes sin pedidos en 2023, debería hacer `FROM Clientes LEFT JOIN Pedidos ON Clientes.id = Pedidos.id AND YEAR(Pedidos.fecha)=2023` en lugar de poner `YEAR(Pedidos.fecha)=2023` en el WHERE, porque esto último removería los clientes sin pedidos (con Pedidos.fecha NULL). Entender este matiz evita errores lógicos. En general, las condiciones sobre la tabla opcional (derecha en left join, izquierda en right join) van en ON si queremos conservar las filas no emparejadas.
- **Limitar resultados si es posible durante pruebas:** En bases de datos grandes, un JOIN puede producir muchísimas filas. Si está explorando datos o armando la consulta, considere usar cláusulas como `WHERE` con condiciones restrictivas o `LIMIT` (o `TOP` en T-SQL) para acotar temporalmente el resultado mientras prueba. Así evita esperar mucho tiempo o sobrecargar el sistema accidentalmente. Una vez comprobado, puede ejecutar sin el límite para obtener el resultado completo.
- **Documentar y formatear la consulta:** En entornos profesionales, es útil **formatear** correctamente la consulta JOIN (una tabla por línea, indentaciones para ON, etc.) y añadir comentarios si la lógica es compleja. Esto facilita la lectura y mantenimiento por parte de otros desarrolladores.
- **Elegir el tipo de JOIN adecuado:** Si no está seguro inicialmente de qué tipo de JOIN necesita, un consejo es **empezar con INNER JOIN** (que es el más restrictivo) y ver qué pierde, y luego considerar si necesita un LEFT JOIN para reincorporar filas no coincidentes. También puede comenzar con LEFT JOIN y comprobar si las filas "faltantes" son relevantes. Entender la pregunta de negocio es clave: si requiere "todos los X con sus Y", probablemente sea un LEFT JOIN; si requiere "solo X que tienen Y", es un INNER JOIN; si requiere "todo de ambos lados", entonces FULL JOIN, etc.

Errores comunes al usar JOINs

Al utilizar JOINs, especialmente cuando se está aprendiendo, es fácil caer en algunos **errores típicos**. Reconocerlos le ayudará a evitarlos:

- **JOIN sin condición (ON) o mal especificada:** Como se mencionó, olvidar poner la cláusula `ON` o equivocarse en la columna de emparejamiento es un error grave. Sin la condición, el resultado es un **cruce cartesiano** exponencialmente grande e incorrecto. Asegúrese siempre de usar `ON tabla1.columna = tabla2.columna` con las columnas correctas (por ejemplo, unir por IDs correspondientes). Un síntoma de este error es obtener un número de filas inesperadamente enorme.
- **No utilizar alias y calificación de columnas:** Si dos tablas tienen columnas con el mismo nombre (caso común: ambas tienen una columna `cliente_id` o `nombre`), una consulta con JOIN *debe* referirse a ellas con el nombre de tabla o alias (por ejemplo `Clientes.nombre` vs `Pedidos.nombre`). De lo contrario, el SGBD lanzará un error de **columna ambigua**. Olvidar alias también dificulta la lectura; puede llevar a mezclar columnas por accidente. Es mejor acostumbrarse a siempre calificar las columnas, al menos en consultas multi-tabla, para evitar confusiones.
- **Olvidar las implicaciones de NULL en OUTER JOINs:** Un error lógico frecuente con LEFT/RIGHT JOIN es suponer que todas las filas resultantes tendrán datos completos. En realidad, las filas sin pareja tendrán **NULL** en las columnas de la tabla que no tuvo coincidencia. Si luego realiza filtros `WHERE` sobre esas columnas sin tenerlo en cuenta, podría filtrar esas filas. Por ejemplo, supongamos que hace un LEFT JOIN de Clientes con Pedidos y luego agrega `WHERE Pedidos.total > 100`. Esto **eliminará** del resultado a todos los clientes que no tenían pedidos (porque para ellos `Pedidos.total` es NULL y la condición `> 100` resulta falsa al comparar con NULL). La solución es mover ese filtro a la cláusula ON (si el objetivo era "clientes con pedidos mayores a 100, pero también clientes sin pedidos") o reformular la consulta. En resumen, recuerde que NULL en SQL nunca cumple comparaciones ordinarias, así que al usar outer joins, tenga presente dónde aplicar condiciones.
- **Suponer compatibilidad universal de JOINs:** No todos los sistemas de bases de datos implementan todos los tipos de JOIN. Ya mencionamos que SQLite no soporta RIGHT JOIN, y MySQL durante mucho tiempo no soportó FULL JOIN. Un error es escribir una consulta con FULL JOIN y asumir que funcionará en cualquier motor; es necesario conocer las capacidades de su SGBD y quizás preparar consultas alternativas (como UNION de left y right) si migra o necesita compatibilidad.
- **Problemas de rendimiento sin darse cuenta:** Si bien no es un error sintáctico, sí es un error de diseño común no anticipar el volumen de datos que un JOIN puede generar. Unir tablas muy grandes sin criterios adicionales puede resultar en millones de filas. Por eso se recalcó la importancia de probar con límites y asegurarse de tener índices adecuados. Otro error es hacer múltiples JOINs encadenados sin evaluar su necesidad: cada join adicional aumenta la complejidad; únalas solo si realmente necesita datos de ellas. Revise los **planes de ejecución** para comprender cómo el motor está realizando los joins, y agregue índices o ajuste la consulta si observa cuellos de botella (por ejemplo, *table scans* completos en cada join).

- **Uso incorrecto de la sintaxis antigua de JOIN en el WHERE:** En SQL existe una sintaxis antigua (ANSI-89) donde se listaban varias tablas en el FROM separadas por comas y las condiciones de unión se ponían en el WHERE (e.g., `FROM Clientes, Pedidos WHERE Clientes.id = Pedidos.id`). Esta forma, aunque funcional, es propensa a errores (es fácil olvidar una condición y generar un cross join) y menos expresiva. La **buena práctica** es usar la sintaxis moderna con `JOIN ... ON ...` para mayor claridad. Combinar ambas sintaxis en una misma consulta puede llevar a resultados equivocados o confusión, así que mantenga un estilo consistente.

En conclusión, siempre revise cuidadosamente sus consultas con JOIN. Si el resultado no cuadra con lo esperado (demasiadas filas, muy pocas filas, valores NULL inesperados), es probable que la causa sea alguna de las razones anteriores.

Ejercicios prácticos

A continuación se proponen algunos **ejercicios** para reforzar el uso de los JOINs. Intente resolver cada uno escribiendo la consulta SQL correspondiente. *(Nota: No se incluyen soluciones aquí, ya que se proporcionarán en los anexos. Estas tareas están diseñadas para aplicar los conceptos aprendidos.)*

1. **INNER JOIN básico:** Usando las tablas `Clientes` y `Pedidos`, escriba una consulta que liste el `nombre` del cliente, la `ciudad` y el `total` de cada pedido. Solo deben aparecer los pedidos que tengan un cliente asociado en la tabla de Clientes (hint: use un INNER JOIN entre `Clientes` y `Pedidos`).
2. **LEFT JOIN con filtrado de NULL:** Liste todos los clientes junto con el `pedido_id` de alguno de sus pedidos. Si el cliente no ha realizado ningún pedido, el campo de `pedido_id` deberá aparecer como NULL. *(Pista: Use LEFT JOIN. Después, modifique esta consulta para mostrar* **solo** *aquellos clientes que* **no** *tienen pedidos, utilizando una condición que filtre los NULL en los campos de Pedidos.)*
3. **FULL JOIN o combinación completa:** Suponga que dispone de dos tablas de clientes (`Clientes_Europeos` y `Clientes_America`) con estructura similar. Formule una consulta con `FULL JOIN` (unión completa) entre ambas para obtener un listado de **todos** los clientes de Europa y América, emparejando por alguna columna común (por ejemplo, email o número de cliente si hubiera coincidencias entre ambas tablas). Asegúrese de mostrar de forma distinguible qué registros provienen solo de Europa, solo de América, o de ambos. *(Si su sistema SQL no soporta FULL JOIN, piense cómo lograr el mismo resultado combinando LEFT JOIN, RIGHT JOIN y UNION).*

Subconsultas, Alias y Funciones Integradas en SQL

En esta sección exploraremos tres conceptos fundamentales para construir consultas SQL más potentes y legibles: las **subconsultas**, el uso de **alias** para columnas y tablas, y las **funciones integradas** (agregadas, escalares y de fecha). A través de ejemplos prácticos basados en dos tablas de ejemplo – **Clientes** (cliente_id, nombre, ciudad, email, fecha_registro) y **Pedidos** (pedido_id, cliente_id, fecha_pedido, total) –

explicaremos en detalle cada concepto, su sintaxis, buenas prácticas, errores comunes y ejercicios para afianzar el aprendizaje.

Subconsultas

¿Qué es una subconsulta?

Una **subconsulta**, también conocida como *consulta anidada*, es una consulta SQL colocada dentro de otra consulta SQLs. En otras palabras, es una instrucción `SELECT` interna cuyo resultado se utiliza en otra consulta externa. Las subconsultas permiten realizar consultas más complejas dividiendo el problema en partes: primero se obtiene un conjunto de datos intermedio mediante la subconsulta, y luego la consulta externa utiliza ese resultado para filtrar, comparar o derivar información.

Se puede incluir una subconsulta en diversas partes de una sentencia SQL, por ejemplo en las cláusulas `SELECT`, `FROM` y `WHERE`, e incluso en condiciones con operadores especiales como `EXISTS`. Según su uso y resultado, las subconsultas se clasifican en:

- **Subconsultas escalares**: Devuelven **un único valor**, es decir, exactamente una fila con una columna. Suelen emplearse cuando se necesita calcular o obtener un valor para usarlo en una expresión (por ejemplo, una media, un conteo o un valor relacionado con la fila externa).
- **Subconsultas de varias filas**: Devuelven **múltiples filas** (una columna con muchos valores) o incluso **varias columnas** formando una tablalearnsql.es. Normalmente se usan junto con operadores que manejan conjuntos de resultados, como `IN` (para listas de un solo campo) o se colocan en la cláusula `FROM` para actuar como una tabla derivada.
- **Subconsultas correlacionadas**: Son subconsultas que dependen de la consulta externa; es decir, en su condición interna hacen referencia a columnas de la consulta principallearnsql.es. La subconsulta correlacionada se evalúa repetidamente, una vez por cada fila candidata de la consulta externa, típicamente en conjunción con `EXISTS` u operadores que comparan valores fila por fila.

A continuación veremos cuándo y cómo utilizar subconsultas en las principales partes de una sentencia SQL, con ejemplos prácticos.

Subconsultas en la cláusula WHERE

Una de las formas más comunes de emplear subconsultas es dentro de la cláusula `WHERE` para filtrar resultados en base a otra consulta. En este caso, la subconsulta suele proporcionar un valor de comparación o un conjunto de valores que la cláusula `WHERE` usará con operadores como `=`, `>`, `<`, `IN`, `ANY`, `ALL`, etc.

Ejemplo: Supongamos que queremos listar los pedidos cuyo monto total sea superior al promedio de todos los pedidos. Esta es una típica situación donde utilizamos una subconsulta escalar en `WHERE` para calcular primero el valor promedio y luego filtrar:

```
sql
```

```
SELECT pedido_id, cliente_id, fecha_pedido, total
FROM Pedidos
WHERE total > (
   SELECT AVG(total)
   FROM Pedidos
);
```

En esta consulta, la subconsulta `SELECT AVG(total) FROM Pedidos` calcula el **promedio** de la columna `total` de la tabla *Pedidos*. Esa subconsulta devuelve un único valor (por ejemplo, digamos que el promedio es 500.00). La consulta externa luego compara el campo `total` de cada pedido contra ese valor promedio, gracias a la condición `total > (...)`. El resultado final listará solo aquellos pedidos cuyo total excede al promedio general de ventaslearnsql.es. Este uso de subconsulta en `WHERE` nos permite filtrar filas basándonos en un cálculo agregado de la misma tabla, algo que no podríamos lograr en una sola cláusula `WHERE` simple sin subconsulta.

Resultado esperado (ejemplo hipotético):

pedido_id	cliente_id	fecha_pedido	total
1003	12	2025-05-10	750.00
1017	7	2025-07-01	1200.00
...	...	...	...

En la lista, cada `total` mostrado es mayor que el promedio calculado. La subconsulta se evaluó **una vez** antes de filtrar los pedidos, ya que no depende de filas individuales de la consulta externa (es una subconsulta no correlacionada).

Podemos usar subconsultas en `WHERE` también para comparar conjuntos de valores. Por ejemplo, si quisiéramos obtener los clientes que viven en una ciudad donde existe al menos un pedido de más de 1000 euros, podríamos usar una subconsulta de múltiples filas combinada con `IN`:

```
sql
SELECT nombre, ciudad
FROM ClientesWHERE ciudad IN (
   SELECT c2.ciudad
   FROM Clientes c2
   JOIN Pedidos p2 ON p2.cliente_id = c2.cliente_id
   WHERE p2.total > 1000
);
```

En este caso, la subconsulta devuelve una lista de ciudades (posiblemente repetidas, pero el motor SQL las tratará en el contexto de IN como conjunto) donde se encontraron pedidos con total > 1000. La consulta externa luego filtra los clientes cuya `ciudad` esté dentro de esa lista. Este enfoque, aunque válido, a veces puede

reescribirse de forma equivalente con un JOIN y DISTINCT o usando EXISTS (como veremos más adelante) para mayor eficiencia o claridad, pero ilustra el uso de subconsulta en WHERE con un conjunto de resultados.

Subconsultas en la cláusula SELECT

También es posible colocar una subconsulta en la lista de columnas de la cláusula SELECT. En este caso, la subconsulta actúa esencialmente como una **columna derivada** cuyo valor se calcula para cada fila de la consulta externa. Por lo general, estas subconsultas están diseñadas para devolver un solo valor escalar **por cada fila** de la consulta padre; suelen ser *subconsultas correlacionadas* porque necesitan usar datos de la fila actual en la condición de la subconsulta.

Ejemplo: Listar cada cliente junto con la **fecha de su último pedido**. Para resolver esto con una subconsulta en SELECT, podemos calcular la fecha máxima (MAX(fecha_pedido)) dentro de Pedidos para cada cliente:

```
sql
SELECT
  c.cliente_id,
  c.nombre,
  c.email,
  /* Subconsulta escalar correlacionada para última fecha de pedido */
  (
    SELECT MAX(p.fecha_pedido)
    FROM Pedidos p
    WHERE p.cliente_id = c.cliente_id
  ) AS ultima_fecha_pedido
FROM Clientes c;
```

Aquí, la subconsulta (SELECT MAX(p.fecha_pedido) FROM Pedidos p WHERE p.cliente_id = c.cliente_id) busca la fecha más reciente (MAX(fecha_pedido)) en la tabla *Pedidos* para el cliente actual (c.cliente_id) procesado por la consulta externa. Notemos que la subconsulta hace referencia a c.cliente_id, es decir, a la fila corriente de *Clientes* en la consulta principal; esto la convierte en una subconsulta **correlacionada**. El resultado devuelto por la subconsulta será distinto según el cliente: para cada cliente traerá su última fecha de pedido (o NULL si el cliente no tiene pedidos en la tabla *Pedidos*).

La salida tendría columnas de cliente y la última fecha de pedido. Por ejemplo:

cliente_id	nombre	email	ultima_fecha_pedido
1	Juan Perez	jperez@dominio.es	2025-06-15
2	María Gómez	mgomez@dominio.es	2024-12-30
3	Luis Díaz	ldiaz@dominio.es	(NULL)
...	...	...	...

En este resultado hipotético, el cliente con `cliente_id` 3 no tiene fecha de último pedido (aparece `NULL`), lo que indicaría que no tiene ningún registro en *Pedidos*. Así, la subconsulta correlacionada en `SELECT` nos permitió agregar información derivada (la última fecha de compra) a cada fila de cliente sin necesidad de hacer un `JOIN` complejo.

Nota: Las subconsultas en la cláusula `SELECT` **deben retornar un único valor por fila**. Si la subconsulta podría devolver más de un valor, el SGBD lanzará un error de ejecución (por ejemplo: *"Subquery returns more than one row"*) porque no sabría cuál de ellos asignar a la columna derivada. Para asegurar que la subconsulta escalar sea válida, normalmente utilizamos funciones de agregado (como `MAX` en el ejemplo) o condiciones que limiten el resultado a una fila.

Subconsultas en la cláusula FROM (tablas derivadas)

Otra utilidad poderosa de las subconsultas es usarlas en la cláusula `FROM` como **tablas derivadas** (también llamadas *subconsultas en línea*). En este patrón, una subconsulta completa (con sus propias selecciones, filtros e incluso agrupaciones) se encierra entre paréntesis en la cláusula `FROM` y se le asigna un alias, de manera que el resultado de esa subconsulta se trate como una tabla virtual a la cual podemos hacerle `JOIN` o de la cual podemos seleccionar campos.

Usar subconsultas en `FROM` es útil para estructurar consultas complejas en pasos lógicos. Por ejemplo, puede permitirnos primero obtener un conjunto de resultados resumidos y luego combinarlo con otras tablas.

Ejemplo: Supongamos que queremos obtener una lista de clientes junto con el **número de pedidos** que cada uno ha realizado. Esto se puede lograr fácilmente agrupando la tabla de *Pedidos* por `cliente_id` y contando, pero si queremos además mostrar datos del cliente (por ejemplo el nombre), podríamos hacerlo con un `JOIN`. Una forma de enfocar la solución es usar una subconsulta en `FROM` que ya calcule el número de pedidos por cliente, y luego unirla con *Clientes*:

```
sql
SELECT
  c.nombre,
  COALESCE(sub.num_pedidos, 0) AS num_pedidos
FROM Clientes c
LEFT JOIN (
```

```
    SELECT cliente_id, COUNT(*) AS num_pedidos
    FROM Pedidos
    GROUP BY cliente_id
) AS sub
  ON sub.cliente_id = c.cliente_id;
```

Analicemos este ejemplo paso a paso:

- La subconsulta dentro de `FROM` obtiene dos columnas: `cliente_id` y `num_pedidos`. Lo hace seleccionando de *Pedidos*, contando cuántos registros hay por `cliente_id` (`COUNT(*)`), y agrupando por `cliente_id`. El resultado de esa subconsulta sería una especie de tabla resumida con cada cliente_id que tiene pedidos y la cantidad correspondiente.
- A la subconsulta se le asigna el alias `sub` para poder referenciar sus columnas. La sintaxis es `(... ) AS sub`. Esto es obligatorio: cada subconsulta en `FROM` debe tener un alias de tabla.
- Luego, en la consulta externa principal, hacemos un `LEFT JOIN` entre la tabla *Clientes* (`c`) y la tabla derivada `sub` en la clave común `cliente_id`.
- Usamos `LEFT JOIN` en lugar de `INNER JOIN` porque queremos incluir también a aquellos clientes que **no tengan pedidos** (en cuyo caso la subconsulta no tendría una fila para ellos, y el LEFT JOIN produciría valores nulos). Para esos casos utilizamos `COALESCE(sub.num_pedidos, 0)` al seleccionar, de forma que si `sub.num_pedidos` viene nulo (cliente sin pedidos), se muestre 0 en su lugar.

El resultado podría verse así:

```
nombre         | num_pedidos
-------------- | -----------
Juan Perez     | 5
María Gómez    | 2
Luis Díaz      | 0
...            | ...
```

Este resultado indica, por ejemplo, que Juan Pérez ha realizado 5 pedidos, María Gómez 2, y Luis Díaz 0 (ninguno). Todo esto se logró gracias a la subconsulta en `FROM` que resumió *Pedidos* por cliente. Un detalle importante es que podríamos haber obtenido el mismo resultado con una sola consulta usando `JOIN` y `GROUP BY` directamente, pero a veces las subconsultas en `FROM` hacen la consulta más legible o permiten cálculos que de otra forma requerirían expresiones más complejas.

Ventaja de tablas derivadas: La legibilidad y la modularidad. Podemos pensar la subconsulta del ejemplo como "primero obtengo la tabla de clientes con su número de pedidos, luego la uno con la tabla de clientes para los detalles". Esto divide la tarea en sub-tareas lógicas.

Subconsultas con EXISTS

El operador EXISTS se usa en SQL en conjunción con una subconsulta para verificar la **existencia de al menos una fila** que cumpla una cierta condición. A diferencia de usar IN (que comprueba pertenencia en una lista de valores) o de comparar con un valor directo, EXISTS simplemente devuelve *TRUE* o *FALSE* para cada fila de la consulta externa, dependiendo de si la subconsulta retorna **al menos una fila** (caso TRUE) o ninguna (FALSE). Típicamente, la subconsulta con EXISTS es **correlacionada**, ya que suele referirse a la fila actual de la consulta externa.

Ejemplo: Listar los clientes que **han realizado al menos un pedido** con un monto superior a 1000. Utilizaremos EXISTS para comprobar la condición por cada cliente:

```
sql
SELECT nombre, ciudad, email
FROM Clientes c
WHERE EXISTS (
   SELECT 1
   FROM Pedidos p
   WHERE p.cliente_id = c.cliente_id
     AND p.total > 1000
);
```

En la cláusula WHERE tenemos EXISTS (subconsulta). Esta subconsulta busca en *Pedidos* alguna fila (SELECT 1 es una sintaxis común; en realidad podríamos usar SELECT *, lo importante es la existencia de filas) donde p.cliente_id = c.cliente_id (coincidiendo con el cliente actual) **y** p.total > 1000. Si para un cliente dado existe al menos un pedido que cumple la condición (total > 1000), la subconsulta devuelve una o más filas, haciendo que EXISTS sea verdadero y por tanto ese cliente pase el filtro. Si la subconsulta no encuentra ningún pedido que satisfaga la condición, EXISTS será falso y ese cliente será descartado.

Importante: En el caso de EXISTS, la subconsulta normalmente no necesita devolver datos particulares; por convención se suele usar SELECT 1 o SELECT * dentro del EXISTS. El motor SQL detendrá la búsqueda en la subconsulta tan pronto como encuentre una fila que cumpla la condición, ya que no importa cuántas haya, solo le interesa saber si existe al menos una.

La ventaja de EXISTS sobre, por ejemplo, usar un IN en estos casos, es que puede ser más eficiente y claro cuando la lógica es *"quiero filas de la tabla externa para las cuales haya al menos una coincidencia en la tabla interna con tal condición"*. Además, EXISTS maneja bien casos con valores NULL (a diferencia de NOT IN, donde hay que tener cuidado especial si la subconsulta puede devolver nulos).

Podemos fácilmente modificar el ejemplo para obtener los clientes que **no** tienen pedidos mayores a 1000 utilizando NOT EXISTS:

```sql
SELECT nombre, ciudad, email
FROM Clientes c
WHERE NOT EXISTS (
   SELECT 1
   FROM Pedidos p
   WHERE p.cliente_id = c.cliente_id
     AND p.total > 1000
);
```

Esta consulta devolvería aquellos clientes para los cuales la subconsulta *no* encontró ningún pedido sobre 1000, es decir, clientes sin pedidos costosos (o sin ningún pedido en absoluto).

Subconsultas correlacionadas y rendimiento: Cabe mencionar que las subconsultas correlacionadas (como las usadas con `EXISTS`) se evalúan para cada fila candidata de la consulta externa, lo que puede implicar muchas ejecuciones de la subconsulta. Los sistemas de bases de datos modernos suelen optimizar estos casos (por ejemplo, convirtiéndolos internamente en un semi-join), pero es bueno ser consciente de ello. Si la subconsulta correlacionada resulta ser costosa, a veces se puede reescribir la consulta usando `JOIN` o subconsultas no correlacionadas para mejorar el rendimiento.

Alias (para columnas y tablas)

En SQL, un **alias** es un nombre alternativo que se asigna temporalmente a una columna o tabla dentro de una consulta. Los alias no cambian los nombres reales en la base de datos; solo existen durante la ejecución de la consulta actual y sirven para simplificar referencias o mejorar la legibilidad. Usar alias de forma adecuada hace que nuestras consultas sean más claras y menos propensas a ambigüedades, especialmente cuando trabajamos con múltiples tablas o cuando realizamos cálculos.

¿Por qué usar alias en SQL? Algunas razones comunes incluyen:

- **Simplificar nombres complejos:** Si nuestras tablas o columnas tienen nombres largos o poco descriptivos en el contexto de la consulta, podemos asignarles un alias corto y significativo. Esto reduce la escritura repetitiva y hace que la consulta sea más fácil de leer.
- **Claridad en consultas con JOIN múltiples:** Cuando se unen varias tablas, es útil (y a menudo necesario) **anteponer un alias a los nombres de columna** para ver fácilmente qué columna pertenece a qué tabla. Sin alias, tendríamos que escribir el nombre completo de la tabla cada vez y la consulta se volvería verbosa y potencialmente confusa.
- **Evitar ambigüedades:** Si dos tablas unidas tienen columnas con el mismo nombre, debemos diferenciarlas. Usar alias de tabla permite referirnos a cada columna precedida por el alias correspondiente (por ejemplo, `a.nombre` vs `b.nombre`). Además, podemos renombrar columnas calculadas o agregadas para darles una identidad en el resultado.
- **Uso en subconsultas y autouniones:** Cuando una subconsulta en `FROM` produce una "tabla derivada", debemos darle un alias para poder referenciar sus columnas.

Asimismo, en una *autounión* (self-join), donde una tabla se une consigo misma, es obligatorio usar alias diferentes para cada instancia de la tabla, o de lo contrario el servidor no sabrá distinguirlas.

- **Presentación de resultados:** Podemos usar alias de columna para que los encabezados en el conjunto de resultados sean más descriptivos o estén en un idioma específico (por ejemplo, mostrar `nombre_completo` como "Nombre Completo" en el resultado usando `AS "Nombre Completo"`).

A continuación, veremos la sintaxis y ejemplos de alias tanto para columnas como para tablas.

Alias de columna

La sintaxis básica para asignar un alias a una columna es escribir la expresión de columna seguida de la palabra clave `AS` y el alias deseado. Por ejemplo:

```
sql
SELECT columna_original AS alias_columna
FROM tabla;
```

La palabra `AS` es opcional en muchos sistemas de SQL, por lo que también podríamos escribir `columna_original alias_columna` directamente, pero usar `AS` suele mejorar la claridad. Si el alias incluye espacios o caracteres especiales, se debe encerrar entre comillas (simples o dobles dependiendo del SGBD), por ejemplo: `SELECT customer_name AS "Nombre Completo" ...`.

Ejemplo: Supongamos que queremos listar los clientes mostrando su identificador, nombre y ciudad, pero con encabezados más descriptivos en español. Podemos hacer:

```
sql
SELECT
  cliente_id AS id,
  nombre AS nombre_cliente,
  ciudad AS ciudad_residencia
FROM Clientes;
```

Esto no altera los datos, pero en el resultado veremos las columnas con títulos renombrados como *id*, *nombre_cliente* y *ciudad_residencia* en lugar de los nombres originales. Usar alias de columna es especialmente útil cuando nuestras columnas son el resultado de expresiones o funciones. Por ejemplo, si calculamos el total de pedidos por cliente, podríamos poner `COUNT(*) AS total_pedidos` para que esa columna calculada aparezca con el encabezado "total_pedidos".

Otra utilidad es usar alias para hacer más comprensible una fórmula. Por ejemplo:

```
sql
SELECT
  nombre,
  ROUND( DATEDIFF(NOW(), fecha_registro) / 365.0, 1 ) AS años_registrado
```

```
FROM Clientes;
```

En esta consulta hipotética, usamos una fórmula para calcular cuántos años lleva registrado el cliente (diferencia entre la fecha actual y `fecha_registro`, dividida por 365). Al asignarle el alias `años_registrado`, el resultado será más fácil de interpretar en la salida.

Alias de tabla

Al usar varias tablas en una consulta (especialmente con `JOIN`), los alias de tabla sirven para abreviar la referencia a cada tabla. La sintaxis es similar: se coloca el nombre de la tabla seguido de la palabra clave `AS` (opcional en muchos SGBD) y el alias de la tabla. Por ejemplo:

```sql
SELECT *
FROM nombre_tabla AS alias;
```

Ejemplo con JOIN: Consideremos una consulta que une las tablas *Clientes* y *Pedidos* para mostrar información combinada. Sin alias, tendríamos que referirnos a las columnas con el nombre de la tabla completo, así:

```sql
SELECT Clientes.nombre, Pedidos.total
FROM Clientes JOIN Pedidos
ON Pedidos.cliente_id = Clientes.cliente_id;
```

Esto es correcto pero verboso. Usando alias la misma consulta se ve más limpia:

```sql
SELECT c.nombre, p.total
FROM Clientes AS c
JOIN Pedidos AS p
  ON p.cliente_id = c.cliente_id;
```

Aquí asignamos `c` como alias para *Clientes* y `p` como alias para *Pedidos*. Ahora, en lugar de escribir `Clientes.nombre` podemos escribir `c.nombre`, y `p.total` en lugar de `Pedidos.total`. Además de ahorrar escritura, mejora la legibilidad al separar claramente qué campos provienen de cada tabla.

En el caso de *self-joins*, el uso de alias de tabla es obligatorio. Por ejemplo, imaginemos que en la tabla *Clientes* queremos encontrar pares de clientes que viven en la misma ciudad. Tendríamos que unir *Clientes* consigo misma en la columna ciudad:

```sql
SELECT A.nombre AS nombre_cliente_1,
      B.nombre AS nombre_cliente_2,
      A.ciudad
```

```
FROM Clientes AS AJOIN Clientes AS B
  ON A.ciudad = B.ciudad
  AND A.cliente_id < B.cliente_id;
```

En esta consulta, A y B son dos alias distintos para la **misma tabla** *Clientes*. Esto nos permite comparar registros dentro de la tabla. La condición A.cliente_id < B.cliente_id es un truco para no obtener cada par dos veces (evitando duplicados invertidos, ya que si Juan va emparejado con María, no necesitamos repetir María con Juan). Sin alias, esta consulta sería imposible de escribir porque el parser de SQL no sabría a qué instancia de *Clientes* nos referimos en cada caso.

Buenas prácticas con alias: Es recomendable elegir alias cortos pero descriptivos. Por ejemplo, c para *Clientes* es conciso; si tuviéramos *Clientes* y *Ciudades* quizás convenga cli y ciu para distinguir. Evite usar alias demasiado crípticos (como x, y) a menos que sea algo temporal en subconsultas. También hay que tener en cuenta que algunos SGBD tienen pequeñas diferencias: por ejemplo, en SQL Server y Oracle no se permite usar la palabra clave AS antes de un alias de tabla (simplemente se escribe FROM tabla alias), mientras que en otros como PostgreSQL o MySQL es aceptado usar AS o no usarlo, indistintamente. En todos los casos, el alias **no debe** colidir con otro nombre de tabla o columna en la misma consulta.

Funciones integradas en SQL

SQL proporciona numerosas **funciones integradas** (built-in) que podemos utilizar dentro de nuestras consultas para realizar transformaciones o cálculos sobre los datos. Estas funciones suelen dividirse en distintas categorías según su propósito. En esta sección nos enfocaremos en:

- **Funciones agregadas:** Aquellas que operan sobre un conjunto de filas y devuelven un resultado *único* para ese conjunto (por ejemplo, sumas, promedios, conteos).
- **Funciones escalares (de texto u otros tipos):** Aquellas que operan sobre un solo valor y devuelven un solo valor por cada fila (por ejemplo, convertir texto a mayúsculas, calcular la longitud de una cadena, etc.).
- **Funciones de fecha y hora:** Un conjunto muy útil de funciones para obtener la fecha actual, calcular diferencias entre fechas o modificar fechas sumando intervalos.

Estas funciones están "integradas" en el lenguaje SQL (aunque pueden variar ligeramente según el motor de base de datos) y se usan directamente en las consultas, ya sea en la lista de selección, en cláusulas WHERE/HAVING o incluso en órdenes de ORDER BY, siempre que tenga sentido. Veamos cada categoría con ejemplos prácticos.

Funciones agregadas

Las **funciones agregadas** realizan cálculos sobre un conjunto de valores (generalmente varias filas) y retornan un único valor resumido. Son esenciales para

obtener información como totales, promedios, mínimos, máximos y conteos de conjuntos de registros. Las más comunes en SQL son:

- SUM(): Devuelve la suma total de una columna numérica. Ignora los valores NULL. Útil para obtener, por ejemplo, el total de ingresos, la suma de cantidades, etc.
- COUNT(): Cuenta la cantidad de filas que cumplen la condición, o simplemente el total de filas de una consulta. COUNT(*) cuenta filas incluyendo duplicados y nulos; COUNT(columna) cuenta solo filas donde esa columna no es nula. Sirve para saber cuántos registros hay, por ejemplo cuántos pedidos se realizaron o cuántos clientes cumplen cierta característica.
- AVG(): Calcula el promedio (media aritmética) de los valores de una columna numérica. Ignora nulos. Se usa para hallar, por ejemplo, el precio medio de ventas, el promedio de edad de clientes, etc.
- MIN(): Retorna el valor mínimo de una columna. Funciona con números, fechas (retorna la más antigua/menor) e incluso texto (en orden alfabético, retorna el "menor"). Es útil para encontrar, por ejemplo, la fecha más antigua de registro o el menor monto de pedido.
- MAX(): Retorna el valor máximo de una columna. Análogo a MIN(), pero para el mayor valor (fecha más reciente, mayor número, etc.).

Importante: Las funciones agregadas (exceptuando COUNT(*)) suelen ignorar los valores NULL en sus cálculos. Por ejemplo, al promediar o sumar, las filas con valor nulo en la columna objetivo no se cuentan. Además, cuando se usan funciones agregadas junto con otras columnas en un SELECT, es necesario usar una cláusula GROUP BY para agrupar por las demás columnas no agregadas, de modo que la consulta tenga sentido lógico. Alternativamente, se pueden usar subconsultas anidadas para calcular agregados por grupo como vimos antes.

Veamos un ejemplo práctico que combine varias funciones agregadas: supongamos que queremos obtener, para cada **ciudad** en la que tenemos clientes, cuántos pedidos se han realizado y el monto total acumulado de esos pedidos. Esto implica combinar *Clientes* y *Pedidos*, agrupar por ciudad, contar pedidos y sumar sus totales:

```
sql
SELECT
  c.ciudad,
  COUNT(p.pedido_id) AS total_pedidos,
  SUM(p.total) AS monto_total
FROM Clientes c
JOIN Pedidos p ON p.cliente_id = c.cliente_id
GROUP BY c.ciudad;
```

Explicación: unimos *Clientes* con *Pedidos* para asociar cada pedido a la ciudad del cliente que lo hizo. Luego agrupamos los resultados por c.ciudad. Para cada grupo (cada ciudad), COUNT(p.pedido_id) contará cuántos pedidos hay en ese grupo y SUM(p.total) sumará los importes de esos pedidos. Los alias total_pedidos y monto_total se utilizan para que el resultado tenga encabezados claros.

El resultado de ejemplo podría ser:

```
ciudad       | total_pedidos | monto_total
------------ | ------------- | -----------
Madrid       | 15            | 45,230.50
Barcelona    | 8             | 22,100.00
Valencia     | 3             | 5,750.00
...          | ...           | ...
```

Esto nos dice, por ejemplo, que en Madrid hubo 15 pedidos por un total de 45,230.50 (moneda cualquiera), en Barcelona 8 pedidos sumando 22,100.00, etc. Detrás de escena, el SGBD agrupó todas las filas por ciudad y realizó los cálculos de agregación para cada grupo.

Podemos utilizar otras funciones agregadas de forma similar. Por ejemplo, si quisiéramos saber el **pedido máximo y mínimo** registrado en la tabla *Pedidos* podríamos hacer:

```sql
SELECT
  MAX(total) AS pedido_mayor,
  MIN(total) AS pedido_menor,
  AVG(total) AS pedido_promedio
FROM Pedidos;
```

Que podría retornar algo como:

```
pedido_mayor | pedido_menor | pedido_promedio
------------ / ------------ / ----------------
    2500.00  |     50.00    |    540.27
```

indicando que el pedido más alto fue de 2500, el menor de 50, y el promedio alrededor de 540.27. Estas funciones nos dan una visión resumida de los datos.

Nota: No es válido en SQL estándar usar funciones agregadas en la cláusula `WHERE` (porque `WHERE` filtra filas individuales, no puede filtrar sobre un valor agregado que aún no se ha calculado al nivel de grupo). Para eso existe la cláusula `HAVING`, que se aplica después de un `GROUP BY` para filtrar grupos enteros en base a condiciones agregadas (por ejemplo, podríamos agregar `HAVING SUM(p.total) > 20000` para mostrar solo ciudades con más de 20,000 en ventas totales). Si intenta usar, por ejemplo, `WHERE SUM(p.total) > 20000` obtendrá un error, ya que conceptualmente no encaja en la fase de filtrado por fila.

Funciones escalares (texto y otros tipos)

Las **funciones escalares** operan sobre valores individuales y retornan un valor por cada fila, sin condensar resultados de múltiples filas. Hay muchísimas funciones escalares integradas en SQL, que abarcan textos, números, fechas, conversión de tipos, etc. Aquí nos centraremos en algunas funciones de texto muy comunes y útiles: `UPPER`, `LOWER` y `LENGTH`.

- **LOWER(cadena):** Convierte todos los caracteres alfabéticos de la cadena de entrada a **minúsculas**. Caracteres no alfabéticos (dígitos, símbolos) no se afectan. Por ejemplo, `LOWER('Workiva')` devuelve `'workiva'`. Si se le pasa un valor que no es texto, normalmente se convierte implícitamente a texto.
- **UPPER(cadena):** Es el inverso de LOWER; convierte toda la cadena a **mayúsculas**. Ejemplo: `UPPER('Workiva')` resulta en `'WORKIVA'`.
- **LENGTH(cadena):** Devuelve la **longitud** (número de caracteres) de la cadena dada. Por ejemplo, `SELECT LENGTH('Workiva')` retornaría 7 (asumiendo que 'Workiva' tiene 7 letras). En algunos motores SQL se puede usar también `LEN()` (SQL Server) o funciones similares, pero `LENGTH` es bastante universal. Típicamente no cuenta delimitadores de strings, solo el contenido. En ciertos dialectos, LENGTH puede contar bytes si el texto es multibyte, pero para textos ASCII/UTF-8 normales equivale al número de caracteres.

Veamos un ejemplo utilizando estas funciones en la tabla *Clientes*. Supongamos que queremos obtener una lista de clientes con su nombre en mayúsculas y la longitud de su email:

```
sql
SELECT
  nombre,
  UPPER(nombre) AS nombre_mayus,
  email,
  LENGTH(email) AS longitud_email
FROM Clientes;
```

Esta consulta produce para cada cliente su nombre original, el nombre todo en mayúsculas, el email original y el número de caracteres de su email. Por ejemplo:

nombre	nombre_mayus	email	longitud_email
Juan Perez	JUAN PEREZ	jperez@dominio.es	17
María Gómez	MARÍA GÓMEZ	mgomez@otro.com	15
Luis Díaz	LUIS DÍAZ	ldiaz@ejemplo.com	16
...	...	...	...

Observamos en la salida que `nombre_mayus` es simplemente la versión en mayúsculas de `nombre`, y `longitud_email` nos dice cuántos caracteres tiene cada

dirección de correo. Estas funciones no modifican los datos en la base de datos; solo alteran la representación de los datos al proyectarlos en la consulta.

Otras funciones escalares útiles (aunque no listadas en el enunciado) incluyen por ejemplo CONCAT() para concatenar cadenas, SUBSTR() o SUBSTRING() para extraer subcadenas, TRIM() para eliminar espacios en blanco sobrantes, funciones matemáticas como ROUND() para redondear números, etc. Cada base de datos puede tener además funciones específicas.

Un uso práctico de funciones de texto podría ser, por ejemplo, hacer una búsqueda case-insensitive (insensible a mayúsculas/minúsculas) comparando valores en una sola case. Imaginemos que queremos buscar clientes cuyo nombre sea "luis" independientemente de cómo esté almacenado (Luis, LUIS, luIs, etc.). Podemos usar LOWER en la comparación:

```
sql
SELECT *
FROM Clientes
WHERE LOWER(nombre) = 'luis';
```

Aquí convertimos el nombre a minúscula en la comparación, y también ponemos 'luis' en minúscula, garantizando que coincidencias con mayúsculas diferentes se consideren iguales. Esta técnica es útil aunque hay que tener en cuenta que puede afectar el rendimiento si la columna está indexada (aplicar funciones podría inutilizar el índice en algunos SGBD, a menos que exista un índice funcional o la colación de la columna ya sea case-insensitive). Sin embargo, para consultas ad-hoc es perfectamente válido.

Funciones de fecha y hora

Trabajar con fechas es algo muy frecuente en bases de datos (por ejemplo, calcular edades, duraciones de suscripción, filtrar por fechas recientes, etc.). SQL proporciona funciones integradas para manejar estas necesidades. Nos centraremos en tres funciones comunes: NOW(), DATE_ADD y DATEDIFF.

- **NOW():** Retorna la fecha y hora actuales del sistema (timestamp exacto del momento en que se ejecuta la consulta). En muchos sistemas es equivalente a CURRENT_TIMESTAMP. Por ejemplo, SELECT NOW() podría devolver algo como 2025-07-30 10:30:09 dependiendo del formato de fecha/hora de la base de datos. Si solo se necesita la fecha actual sin tiempo, existen funciones como CURRENT_DATE en algunos dialectos, pero NOW() es útil para obtener la marca temporal completa.
- **DATE_ADD(fecha, INTERVAL n unidad):** Devuelve una nueva fecha resultante de **sumar un intervalo de tiempo** especificado a la fecha dada. Es muy usado en MySQL y algunos otros motores con esa sintaxis. Por ejemplo, DATE_ADD('2025-07-01', INTERVAL 7 DAY) produciría 2025-07-08 (sumamos 7 días). Se pueden usar diferentes unidades de tiempo: DAY (día), MONTH (mes), YEAR (año), HOUR, MINUTE, etc. Asimismo, muchos motores

ofrecen DATE_SUB para restar intervalos, o uno puede usar DATE_ADD con valores negativos dependiendo del dialecto.

- **DATEDIFF(fecha1, fecha2):** Calcula la diferencia entre dos fechas y normalmente devuelve el número de días entre ellas (dependiendo del sistema SQL). Su sintaxis varía: en MySQL es DATEDIFF(fecha1, fecha2) y devuelve fecha1 - fecha2 en días (un entero, positivo, negativo o cero). En SQL Server/Oracle hay una función DATEDIFF(unidad, fecha1, fecha2) que permite especificar la unidad (día, mes, año, etc.) en el primer parámetro. Con MySQL, si quisiéramos la diferencia en meses o años, tendríamos que combinar funciones (p.ej. TIMESTAMPDIFF en MySQL) o usar fórmulas.

Veamos ejemplos con nuestras tablas:

Ejemplo 1: Obtener la fecha actual y comparar con la fecha de registro de los clientes. Supongamos que queremos saber cuántos días han pasado desde que cada cliente se registró (fecha_registro) hasta hoy. Podemos usar NOW() y DATEDIFF:

```
sql
SELECT
 nombre,
 fecha_registro,
 DATEDIFF(NOW(), fecha_registro) AS dias_desde_registro
FROM Clientes;
```

Aquí cada fila mostrará el nombre del cliente, su fecha de registro original, y la diferencia en días entre la fecha de hoy (NOW()) y esa fecha de registro. Un fragmento de resultado podría ser:

```
nombre       | fecha_registro | dias_desde_registro
------------ | -------------- | -------------------
Juan Perez   | 2023-07-30     | 365
María Gómez  | 2025-06-01     | 59
Luis Díaz    | 2025-07-28     | 2
...          | ...            | ...
```

Suponiendo que *Juan Perez* se registró hace justo un año, veríamos 365 días; María Gómez hace un par de meses, 59 días; Luis Díaz hace 2 días, etc. Esto nos sirve, por ejemplo, para identificar la antigüedad de cada cliente en días. Si quisiéramos en años, podríamos dividir entre 365 o usar funciones específicas de año.

Ejemplo 2: Generar una fecha a futuro o pasado usando DATE_ADD. Quizá queremos saber, para cada pedido, una fecha estimada de entrega que es 7 días después de la fecha del pedido. Usaríamos DATE_ADD(fecha_pedido, INTERVAL 7 DAY):

```
sql
SELECT
```

```
  pedido_id,
  fecha_pedido,
  DATE_ADD(fecha_pedido, INTERVAL 7 DAY) AS fecha_entrega_estimada
FROM Pedidos;
```

Así, a cada `fecha_pedido` le suma 7 días calendario. Si un pedido se hizo el `2025-07-01`, la `fecha_entrega_estimada` resultante sería `2025-07-08`. Podemos sumar otros intervalos: por ejemplo, `INTERVAL 1 YEAR` para fechas un año después (útil para calcular aniversarios de registro de clientes, expiración de suscripciones, etc.), o `INTERVAL 3 MONTH` para trimestres, etc.

Ejemplo 3: Usar `DATEDIFF` para filtrar datos. Imaginemos que queremos listar los pedidos realizados en los últimos 30 días. Podemos comparar la fecha del pedido con la fecha actual usando `DATEDIFF` o usando directamente date arithmetic:

Opción A (usando DATEDIFF):

```
sql
SELECT pedido_id, cliente_id, fecha_pedido, total
FROM Pedidos
WHERE DATEDIFF(NOW(), fecha_pedido) <= 30;
```

Esto asume que `DATEDIFF(NOW(), fecha_pedido)` devuelve cuántos días han pasado desde la fecha del pedido hasta hoy, y nos quedamos con aquellos <= 30 (es decir, hasta 30 días de antigüedad).

Opción B (usando DATE_ADD o DATE_SUB):

```
sql
SELECT pedido_id, cliente_id, fecha_pedido, total
FROM Pedidos
WHERE fecha_pedido >= DATE_SUB(NOW(), INTERVAL 30 DAY);
```

Aquí usamos `DATE_SUB(NOW(), INTERVAL 30 DAY)` para obtener la fecha de hace 30 días desde hoy, y pedimos los pedidos cuya fecha_pedido sea mayor o igual a esa fecha (o sea, dentro de los últimos 30 días). Esta opción puede ser más eficiente en ciertos motores porque puede aprovechar índices sobre `fecha_pedido`.

Nota: Las funciones de fecha pueden variar en disponibilidad según el dialecto SQL. Por ejemplo, en Oracle se podría sumar intervalos con una sintaxis diferente (usando el operador `+` sobre fechas junto con la función `NUMTODSINTERVAL` o similares). En SQL Server, como vimos, `DATEADD` y `DATEDIFF` se usan con parámetros separados (p.ej., `DATEADD(day, 7, fecha)` para sumar 7 días). Siempre conviene revisar la documentación de la base de datos específica. No obstante, los conceptos de obtener la fecha actual, sumar intervalos y calcular diferencias existen en prácticamente todos los SQL relacionales.

Buenas prácticas y errores comunes

Al utilizar subconsultas, alias y funciones integradas, es importante tener en cuenta algunas recomendaciones para evitar errores típicos y aprovechar al máximo estas características:

- **Planifica el tipo de subconsulta adecuado:** Si tu subconsulta debe devolver un único valor (subconsulta escalar), asegúrate de que la lógica garantice eso (por ejemplo, usando funciones de agregado sin `GROUP BY` adicional, o condiciones que resulten en una sola fila). Si obtienes el error de "subconsulta devolvió más de un registro", quizás debas usar `IN` en lugar de `=` o revisar la condición. Recuerda la diferencia entre subconsultas escalares y de varias filas.
- **Usa subconsultas correlacionadas solo cuando sea necesario:** Las subconsultas correlacionadas pueden ser menos eficientes, ya que se ejecutan fila por fila. Úsalas para operaciones como `EXISTS` o cálculos que realmente dependan de cada fila externa. Si puedes lograr el mismo resultado con una subconsulta no correlacionada (por ejemplo, usando `IN` o un `JOIN`), considera esas alternativas, sobre todo si estás manejando grandes volúmenes de datos.
- **Alias siempre para tablas múltiples:** Cuando una consulta involucra más de una tabla, acostúmbrate a usar alias de tabla para **todas** las columnas, incluso si no es estrictamente necesario. Esto mejora la claridad y previene conflictos de nombres. Además, te prepara para agregar más tablas al JOIN sin confusiones. Por ejemplo, en lugar de `SELECT nombre FROM Clientes JOIN Pedidos ...` haz `SELECT c.nombre FROM Clientes c JOIN Pedidos p ...` y así sabrás de un vistazo que `nombre` es de clientes.
- **Nombres de alias significativos:** Evita usar alias que no tengan relación con el contenido. Un alias de una letra (`a`, `b`) está bien en subconsultas cortas o autojoins, pero en consultas largas tal vez prefieras alias más descriptivos como `cli` para clientes o `ped` para pedidos. Eso sí, no exageres haciendo alias largos; recuerda que la idea es simplificar.
- **Alias obligatorios en subconsultas en FROM:** Si usas una subconsulta como tabla derivada, no olvides darle un alias. Olvidarlo causará un error de sintaxis. Por ejemplo: `FROM (SELECT ... FROM Pedidos) AS sub` es correcto, pero sin `AS sub` la consulta fallará.
- **Cuidado al referenciar alias de columna en cláusulas:** En SQL estándar, no puedes usar un alias de columna definido en la cláusula `SELECT` dentro de la cláusula `WHERE` o `GROUP BY` de esa misma consulta. El motivo es el orden de procesamiento: `WHERE` y `GROUP BY` se evalúan antes de que se seleccione y nombre la columna. Por ejemplo, esto es **incorrecto**: `SELECT total AS monto FROM Pedidos WHERE monto > 100;` (fallará diciendo que `monto` no es una columna conocida). Debes repetir la expresión original en el WHERE (`Pedidos.total > 100`) o usar una subconsulta/CTE si quieres evitar duplicar lógica. Sin embargo, sí se permite usar alias de columna en la cláusula `ORDER BY` (y en `HAVING` en algunos sistemas) porque esas se evalúan después del SELECT.
- **Funciones agregadas y GROUP BY correctos:** Un error común es mezclar en el `SELECT` columnas sin agregar con funciones agregadas sin agrupar adecuadamente. Por ejemplo, `SELECT cliente_id, COUNT(*) FROM Pedidos;` lanzará error porque `cliente_id` no está agrupada ni es parte de una función

agregada. La forma correcta sería agrupar: `SELECT cliente_id, COUNT(*) FROM Pedidos GROUP BY cliente_id;`. Siempre que uses funciones de agregación junto con columnas normales, verifica que las columnas normales estén listadas en `GROUP BY`. Si la intención era usar una subconsulta escalar para obtener un valor agregado global, haz esa subconsulta aparte.

- **NULL en funciones agregadas:** Recuerda que funciones como SUM, AVG, etc., ignoran los nulos. Si necesitas contar registros aunque una columna tenga nulos, usa `COUNT(*)` en lugar de `COUNT(columna)`. Si necesitas considerar los nulos como cero en una suma, podrías usar `COALESCE(columna, 0)` dentro de SUM (aunque generalmente no es necesario porque SUM ignora nulls de todas formas, pero COALESCE puede ser útil en sumas de subconsultas o cálculos combinados).
- **Uso adecuado de COALESCE/IFNULL:** Si combinas tablas (por ejemplo con LEFT JOIN) y calculas funciones agregadas, puedes obtener NULL en resultados donde no hay datos relacionados. Ya vimos un ejemplo con `COALESCE(sub.num_pedidos, 0)`. Es buena práctica manejar esos nulos resultantes para que el resultado sea más interpretable (mostrar 0 en vez de NULL cuando corresponda a "ninguno").
- **Atención a las funciones de cadena y rendimiento:** Funciones como UPPER/LOWER en una condición `WHERE` impiden el uso de índices sobre esa columna en muchos casos. Si tienes un conjunto de datos grande y buscas filas de forma case-insensitive, puede ser preferible usar una colación insensible a mayúsculas en la base de datos, o indexar también la versión upper/lower de la columna. En contexto de aprendizaje o consultas ad-hoc, está bien usarlas, pero en producción ten en cuenta su impacto.
- **Formato de fecha y funciones de fecha:** Asegúrate de entender qué tipo de dato produce cada función. `NOW()` por ejemplo devuelve un *timestamp* completo; si lo comparas con una columna de solo fecha, el SGBD puede convertir implícitamente o la comparación podría obviar la hora. Para "fecha actual sin hora", podrías truncar `NOW()` usando funciones específicas (como `DATE(NOW())` en MySQL para obtener solo la parte de fecha). También ten cuidado con `DATEDIFF` según el motor: en MySQL devuelve siempre días, en otros depende de la unidad especificada.
- **No abusar de subconsultas cuando un JOIN basta:** A veces los principiantes usan subconsultas en cualquier situación, incluso cuando un simple JOIN resolvería el problema de forma más clara. Por ejemplo, para "obtener nombres de clientes y total de cada pedido", no hagas una subconsulta por pedido, simplemente une Clientes con Pedidos. Las subconsultas son ideales cuando necesitas anidar consultas para cálculos específicos o filtrados que no se logran fácilmente con un join directo.
- **Indentación y formato:** Aunque no es un error de ejecución, formatea tus consultas con sangrías y saltos de línea lógicos, especialmente si usas subconsultas. Por ejemplo, en una subconsulta en FROM, ponla en líneas separadas e indenta su contenido, y luego vuelve al nivel anterior para la consulta externa. Esto hará el SQL mucho más legible y reducirá la probabilidad de olvidarte de una condición o un paréntesis.

En resumen, escribir SQL profesional implica combinar correctamente subconsultas, alias y funciones. Hay que probar las consultas con cuidado, interpretar los errores

que el SGBD proporciona (que suelen dar pistas de por qué la consulta no es válida) y seguir buenas prácticas para evitar confusiones.

Ejercicios prácticos

A continuación se proponen algunos ejercicios para poner en práctica los conceptos de subconsultas, alias y funciones integradas. Cada ejercicio se basa en las tablas **Clientes** y **Pedidos** descritas anteriormente. Se recomienda intentar escribir primero la solución por cuenta propia y luego comparar con una solución ejemplo.

1. **Clientes sin pedidos:** Obtén el listado de los clientes (nombre y ciudad) que **no han realizado ningún pedido** hasta la fecha. *(Pista: Puedes resolverlo usando una subconsulta en la cláusula WHERE con* `NOT EXISTS` *o usando* `NOT IN`*.)*
2. **Resumen de pedidos por cliente:** Muestra el **número de pedidos** y el **monto total** acumulado de esos pedidos para cada cliente, junto con el nombre del cliente. Ordena el resultado de mayor a menor por monto total. *(Pista: Utiliza una función agregada COUNT y SUM, agrupando por cliente. Puedes hacerlo mediante un JOIN entre Clientes y Pedidos, usando alias, y una cláusula GROUP BY.)*
3. **Clientes antiguos:** Lista los clientes (nombre, email) que llevan registrados **más de 365 días** (más de un año) desde la fecha actual. Incluye en el resultado cuántos días han pasado desde su registro. *(Pista: Calcula la diferencia de días con* `DATEDIFF` *entre NOW() y fecha_registro, y filtra aquellas mayores a 365.)*
4. **Longitud de nombres:** Encuentra los clientes cuyo **nombre** tiene **más de 10 caracteres**. Muestra el nombre original y su longitud de caracteres. *(Pista: Utiliza la función LENGTH en la cláusula WHERE para filtrar por longitud.)*