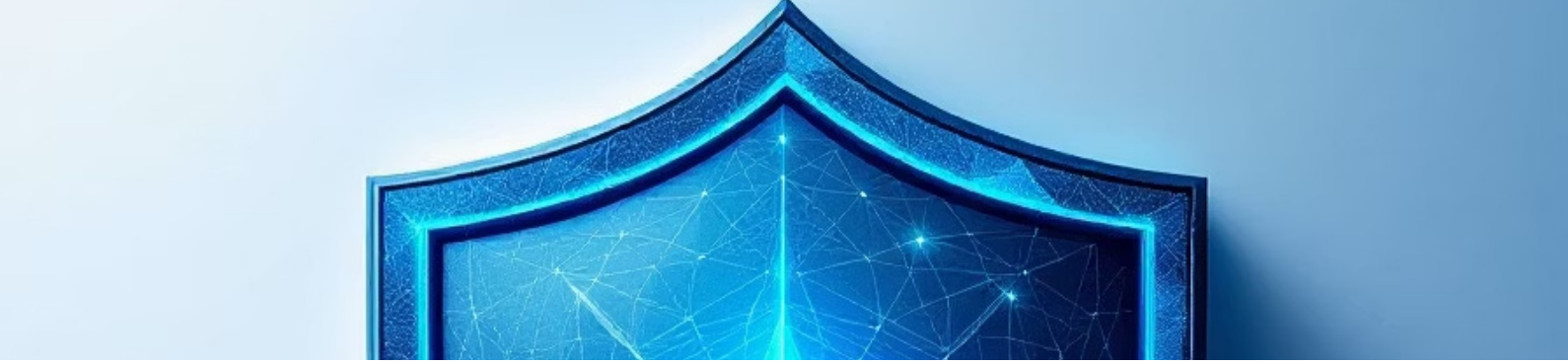
PROMETEO

# Unidad 6:
# Administración de bases de datos

Bases de datos

Técnico Superior de DAM / DAW



Sesión 23 – Usuarios, roles y privilegios en SQL

# Seguridad basada en usuarios, roles y privilegios

Cuando trabajas con bases de datos, no basta con que "funcione". También tienes que asegurarte de que solo las personas adecuadas puedan hacer las acciones adecuadas sobre los datos adecuados. Eso es, en esencia, la gestión de usuarios, roles y privilegios en SQL.

Un **usuario** es una cuenta que puede conectarse al SGBD (MySQL, PostgreSQL, SQL Server, Oracle, etc.). Normalmente representa a una persona o a una aplicación. Cada usuario tiene sus propias credenciales (usuario/contraseña, o autenticación integrada) y actúa con un determinado conjunto de permisos.

Un **privilegio** es el permiso para realizar una acción concreta: por ejemplo, SELECT para leer datos, INSERT para añadir filas, UPDATE para modificarlas, DELETE para eliminarlas, o privilegios administrativos como CREATE TABLE, ALTER, DROP, etc. Estos permisos se gestionan con dos comandos clave:

- **GRANT**: otorga privilegios.
- **REVOKE**: revoca (quita) privilegios.

Podrías asignar privilegios uno a uno a cada usuario, pero en la práctica eso es inviable en entornos profesionales. Imagina tener que actualizar decenas de permisos para cientos de usuarios cada vez que cambian las funciones de un equipo. Ahí entran los **roles**.

Un **rol** es un conjunto de privilegios con nombre. Creas, por ejemplo, el rol `rol_analista` con los permisos necesarios para análisis de datos (solo lectura de ciertas tablas o vistas) y luego simplemente asignas ese rol a todos los analistas. Si mañana ese colectivo necesita un privilegio extra, se lo das al rol y automáticamente todos los usuarios asociados lo heredan.

Este modelo encaja con el **principio de mínimo privilegio**: cada usuario debe tener solo lo estrictamente necesario para hacer su trabajo, ni más ni menos. Así reduces el riesgo de errores, abusos internos y el impacto de posibles ataques.

En un entorno real, el flujo suele ser este:

01

el DBA crea el usuario,

02

crea o reutiliza roles,

03

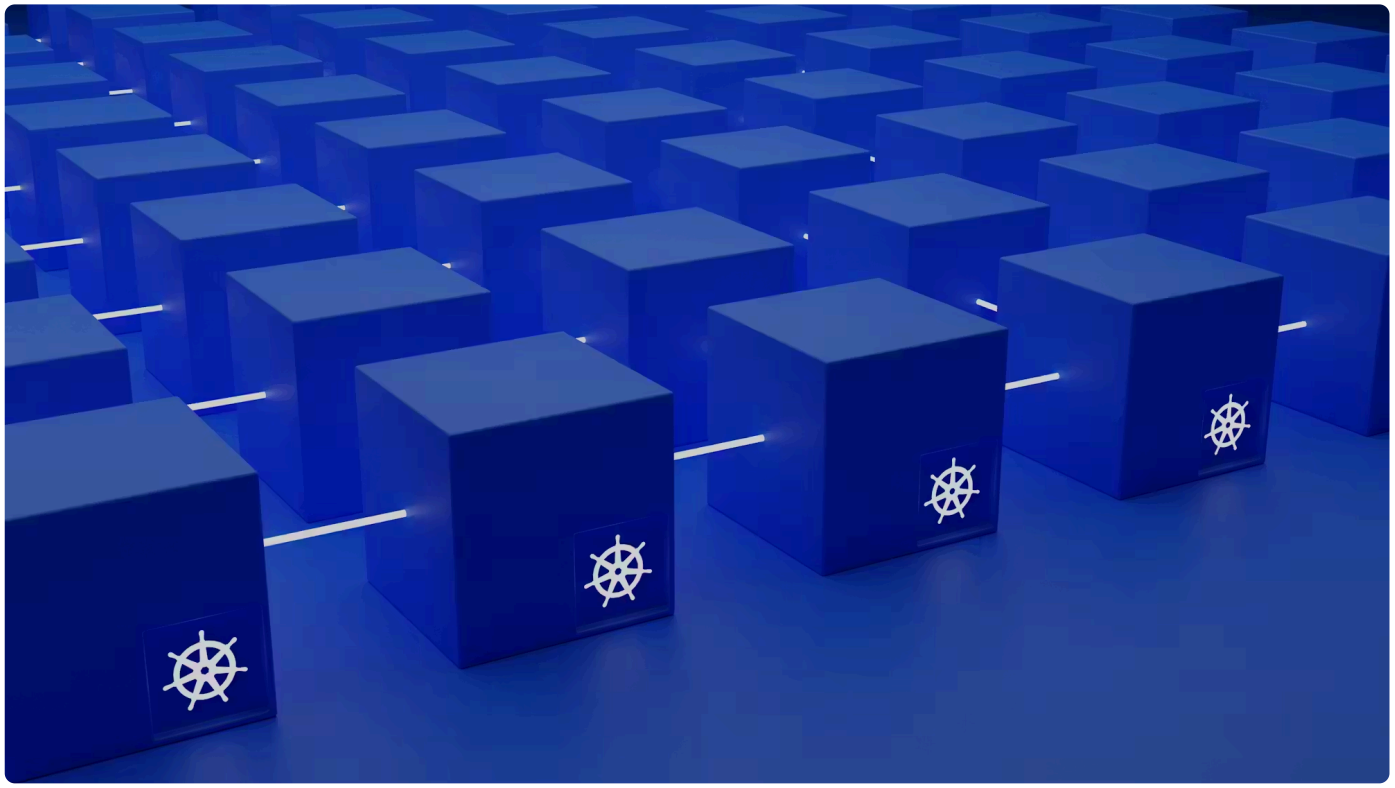
asigna privilegios a roles,

04

asigna roles a usuarios,

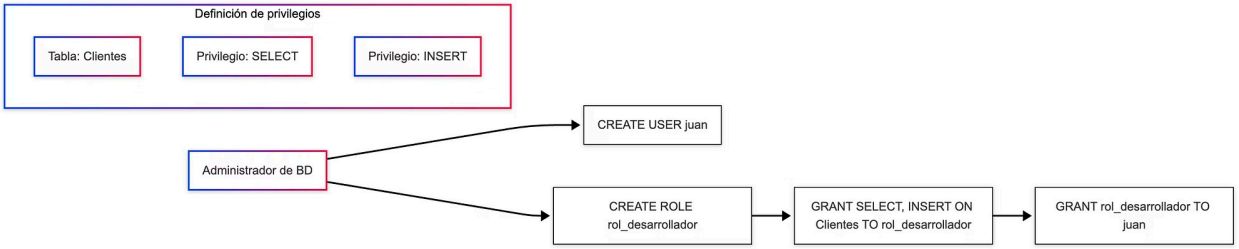
05

audita y revisa periódicamente que los permisos siguen teniendo sentido.



# Esquema Visual: Flujo de creación de usuarios, roles y privilegios

A continuación tienes un diagrama conceptual en formato Mermaid que representa el flujo típico de gestión de permisos usando usuarios, roles y privilegios sobre una tabla:




## Cómo leer el esquema:

- El nodo "Administrador de BD" representa a la persona responsable de la seguridad de la base de datos. Desde ahí salen las acciones de configuración.

- Primero se crea el usuario juan (`CREATE USER juan;`), que será una cuenta individual, con su propia contraseña y parámetros de seguridad.

- Después se crea el rol `rol_desarrollador` (`CREATE ROLE rol_desarrollador;`), que no es una persona, sino un contenedor lógico de permisos.

- En el bloque "Definición de privilegios" aparecen:
  - La tabla Clientes, que es el recurso que queremos proteger.
  - Los privilegios SELECT (leer datos) e INSERT (insertar registros).

- La instrucción conceptual `GRANT SELECT, INSERT ON Clientes TO rol_desarrollador;` significa: "le doy al rol todos los permisos necesarios sobre la tabla". Esto se refleja en la flecha de `rol_desarrollador` hacia los privilegios ligados a Clientes.

- Después, se ejecuta algo equivalente a `GRANT rol_desarrollador TO juan;`, lo que asocia el rol al usuario. A partir de ese momento, juan hereda los privilegios del rol.

Si más tarde decides retirar el permiso de inserción, usarías `REVOKE INSERT ON Clientes FROM rol_desarrollador;`. No hace falta tocar a juan ni a los demás usuarios: al modificar el rol, cambian automáticamente los permisos de todos los que lo tienen.

Este esquema refleja una gestión escalable (cambias el rol, no usuario por usuario) y basada en mínimo privilegio, manteniendo siempre el control sobre quién puede hacer qué sobre cada tabla.


# Caso de Estudio: La seguridad de datos en el Banco Santander

Para entender la importancia de usuarios, roles y privilegios, bajemos a un caso real de un banco como Banco Santander, donde la seguridad de los datos no es solo un tema técnico: es un requisito legal y de negocio.

## Contexto

Un banco gestiona información extremadamente sensible: saldos, movimientos, datos personales, préstamos, hipotecas... Además, está sometido a normativas estrictas de protección de datos (como el GDPR en Europa), regulación financiera y auditorías periódicas. Cualquier acceso indebido —incluso por parte de empleados internos— puede suponer multas millonarias, pérdida de confianza y daño reputacional.

En una entidad así, no todos los empleados necesitan el mismo nivel de acceso a la base de datos. Un cajero no debería ver lo mismo que un analista de riesgos, ni mucho menos tener los mismos permisos que un administrador de sistemas.

## Estrategia: modelo de seguridad basado en roles

El banco utiliza un modelo de control de acceso basado en roles (RBAC) sobre su SGBD:

### Rol "Cajero"

- Puede buscar un cliente (SELECT sobre vistas de CLIENTES con datos limitados).
- Puede ejecutar procedimientos almacenados para ingresos y reintegros (por ejemplo, EXEC sp_ingreso, EXEC sp_reintegro).
- No tiene permisos UPDATE directos sobre la columna saldo. Así se evita que nadie pueda "subirse el sueldo" en su propia cuenta o en la de un conocido.

### Rol "Analista de Marketing"

- Tiene SELECT sobre vistas anonimizadas de clientes y transacciones (por ejemplo, sin nombre, DNI, ni número de cuenta).
- Puede consultar tendencias, segmentaciones, patrones de compra, etc., pero no puede identificar a la persona concreta detrás de cada registro.

### Rol "Administrador de BD (DBA)"

- Tiene privilegios muy amplios: CREATE, ALTER, DROP, gestión de usuarios, roles, permisos, etc.
- Todas sus acciones están auditadas: quién hizo qué, a qué hora y sobre qué objetos. Esto es clave para trazabilidad y para pasar auditorías internas y externas.

Internamente, la administración de permisos se apoya en comandos como:

```
GRANT EXECUTE ON sp_ingreso TO rol_cajero;
GRANT SELECT ON vista_clientes_anonimizados TO rol_analista_mkt;
GRANT rol_cajero TO usuario_cajero_123;
GRANT rol_analista_mkt TO usuario_analista_45;
```

Los usuarios reales (personas) no se asocian a tablas directamente, sino a estos roles predefinidos.

## Resultado e impacto

Con este enfoque:

- Se reduce el riesgo de fraude interno, al evitar que un perfil operativo tenga permisos innecesarios.
- Se limitan los daños en caso de que un usuario sea comprometido (phishing, malware, etc.): el atacante solo hereda los privilegios de ese rol, no acceso total.
- Se facilita el cumplimiento de GDPR y otras normativas, porque solo ciertos roles pueden acceder a datos identificables.
- Se simplifica el día a día: cuando alguien cambia de puesto, basta con quitarle un rol y asignarle otro, en lugar de revisar uno a uno todos sus privilegios.

Este caso resume muy bien por qué, en entornos críticos, la gestión de usuarios, roles y privilegios no es una opción, es la única forma viable de operar con seguridad.

# Herramientas y Consejos para tu Futuro Profesional

Para que puedas aplicar todo esto en la práctica, aquí tienes recomendaciones concretas y herramientas útiles:

## Diseña la seguridad desde los roles, no desde los usuarios

- Antes de crear usuarios, define los principales perfiles de trabajo: por ejemplo, `rol_lectura_reporting`, `rol_app_web`, `rol_dba_junior`, `rol_marketing`.
- Piensa: "¿Qué necesita hacer este rol realmente?" y tradúcelo en GRANT sobre tablas, vistas o procedimientos.
- En SGBD como PostgreSQL, Oracle o SQL Server, los roles están totalmente integrados, y en MySQL/ MariaDB puedes usar roles desde versiones recientes.

## Apóyate en herramientas gráficas para gestionar permisos

Clientes como DBeaver, pgAdmin (PostgreSQL), MySQL Workbench o SQL Server Management Studio (SSMS) ofrecen paneles visuales para:

- Crear usuarios y roles.
- Asignar privilegios sin recordar toda la sintaxis.
- Ver rápidamente qué permisos tiene cada cuenta.

Aunque es importante que entiendas los comandos GRANT y REVOKE, estas herramientas te ayudan a evitar errores en entornos complejos.

## Usa vistas para limitar la exposición de datos

En lugar de dar SELECT directo sobre tablas sensibles (por ejemplo, CLIENTES con DNI, teléfono y dirección), crea vistas controladas:

```
CREATE VIEW vista_clientes_marketing AS
SELECT id_cliente, provincia, edad, segmento
FROM clientes;
```

Luego, da GRANT SELECT a esa vista, no a la tabla original. Así, ciertos roles solo verán los campos estrictamente necesarios.

### Gestiona credenciales con buenas prácticas

- Usa contraseñas fuertes y políticas de caducidad, especialmente para usuarios humanos.
- No compartas contraseñas entre varias personas. Si alguien se va de la empresa, revocas su usuario y se acabó.
- Para aplicaciones, almacena las credenciales en gestores seguros (por ejemplo, Azure Key Vault, AWS Secrets Manager, HashiCorp Vault o un gestor de secretos on-premise), nunca en texto plano en el código.

### Revisa y audita los permisos periódicamente

- Igual que revisas inventario, revisa también quién tiene acceso a qué.
- Detecta usuarios obsoletos, roles que se quedaron "inflados" con privilegios extra, y aplicaciones que usan permisos demasiado amplios.
- Muchos SGBD permiten consultar vistas del sistema (como INFORMATION_SCHEMA o tablas de catálogo) para listar usuarios, roles y permisos y exportarlos a informes.

# Mitos y Realidades

### ❌ Mito: "En un equipo pequeño, podemos compartir todos el mismo usuario de base de datos (ej: root)."

→ **FALSO.** Aunque pueda parecer cómodo al principio ("así todos tenemos acceso total y no hay líos"), es una de las peores prácticas que puedes adoptar. Si todo el mundo usa el mismo usuario:

- No sabes quién ha hecho qué: no hay trazabilidad, lo cual es crítico si ocurre un borrado masivo o una modificación peligrosa.
- Si esa contraseña se filtra (por ejemplo, porque alguien la reutiliza en otro servicio o la envía por email), el atacante obtiene los mismos permisos máximos que el equipo.
- Es imposible aplicar el principio de mínimo privilegio. Todos tienen acceso a todo, aunque solo lo necesite una persona.

La práctica correcta es que cada persona y cada aplicación tenga su propia cuenta, con permisos ajustados a su rol.

### ❌ Mito: "La seguridad de la base de datos no es mi problema, es cosa de los administradores de sistemas."

→ **FALSO.** En la realidad de las empresas, la seguridad es siempre una responsabilidad compartida:

- El DBA define la política general (roles, auditoría, backups).
- El equipo de sistemas se encarga del sistema operativo, cortafuegos, redes, copias de seguridad a nivel de infraestructura, etc.
- Los desarrolladores deben entender cómo funcionan usuarios, roles y privilegios para:
  - Pedir los permisos correctos para sus aplicaciones (por ejemplo, un usuario de aplicación que solo pueda SELECT y INSERT en ciertas tablas, pero no DROP).
  - Escribir código que no abra puertas a ataques como la inyección SQL. Si combinas consultas inseguras con usuarios demasiado privilegiados, el riesgo se dispara.
- Los analistas y usuarios de negocio también tienen responsabilidad: no deben compartir credenciales, ni descargar datos sensibles sin necesidad.

Cuanto antes interiorices que tú también formas parte de esta cadena, más fácil será que diseñes sistemas seguros en tu futuro trabajo.

## Resumen Final para el Examen

- La seguridad en bases de datos se apoya en usuarios, privilegios y roles: los usuarios ejecutan acciones, los privilegios definen qué pueden hacer y los roles agrupan esos privilegios.
- GRANT otorga permisos y REVOKE los revoca; lo habitual es conceder privilegios a roles y luego asignar esos roles a usuarios.
- El principio de mínimo privilegio exige dar a cada usuario solo los permisos necesarios para su trabajo, evitando accesos excesivos.
- Compartir usuarios (como root) y delegar "la seguridad" solo en sistemas o DBAs son malas prácticas; la seguridad es compartida y cada persona y aplicación debe tener credenciales propias y ajustadas a su rol.

## Sesión 24 – Transacciones, commit, rollback, concurrencia

# Transacciones y propiedades ACID

En una base de datos profesional casi ninguna operación importante es "un solo paso". Casi siempre estás encadenando acciones: insertar en una tabla, actualizar otra, registrar un histórico… Si algo falla a medias, puedes dejar los datos en un estado incoherente. Para evitarlo, los SGBD utilizan **transacciones**.

Una **transacción** es una unidad de trabajo atómica: o se ejecuta al completo, o no se aplica nada. Piensa en una transferencia bancaria clásica: restas 100 € de una cuenta y los sumas a otra. No tiene sentido que se reste sin sumarse, ni que se sume sin haberse restado en origen. Ambas operaciones se consideran parte de una sola transacción.

Las transacciones se apoyan en el famoso modelo **ACID**:

### **A**tomicity (Atomicidad)

"todo o nada". Si una parte de la transacción falla, el SGBD deshace todos los cambios.

### **C**onsistency (Consistencia)

la base de datos pasa de un estado válido a otro estado válido respetando reglas de negocio y restricciones (claves, integridad referencial, etc.).

### **I**solation (Aislamiento)

varias transacciones pueden ejecutarse a la vez, pero cada una debe comportarse como si fuera la única en el sistema, sin interferencias que rompan la lógica.

### **D**urability (Durabilidad)

una vez que se confirma una transacción, sus cambios persisten aunque se caiga el servidor o haya un corte de luz (gracias a logs y mecanismos de almacenamiento).

En SQL, los comandos básicos para trabajar con transacciones son:

- **START TRANSACTION** (o BEGIN): indica al SGBD que lo que viene a continuación forma parte de la misma unidad de trabajo.
- **COMMIT**: confirma todos los cambios de la transacción y los hace permanentes y visibles para el resto.

- **ROLLBACK**: deshace todos los cambios realizados desde el inicio de la transacción.

Por defecto, muchos SGBD trabajan en modo **autocommit**: cada sentencia INSERT, UPDATE, DELETE se considera implícitamente una transacción completa y se confirma sola. Para agrupar varias sentencias, desactivas autocommit o utilizas explícitamente START TRANSACTION.

El otro gran tema es la **concurrencia**: múltiples usuarios y procesos leyendo y escribiendo a la vez los mismos datos. Si no se gestiona bien, pueden aparecer problemas como:

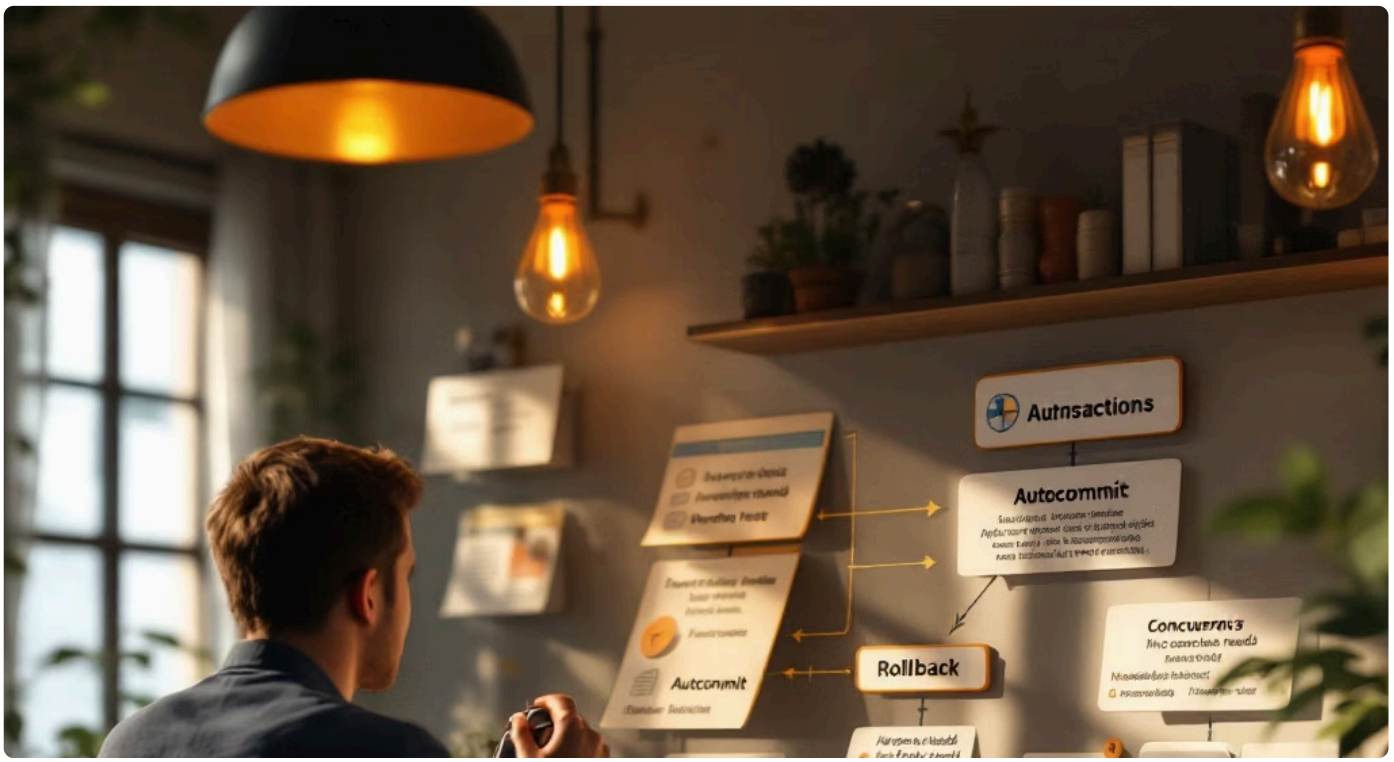
- **Lecturas sucias (dirty reads):** leer cambios que aún no se han confirmado y que podrían deshacerse.

- **Lecturas no repetibles:** leer un dato dos veces en la misma transacción y obtener valores distintos porque otra transacción lo modificó entre medias.

- **Lecturas fantasma (phantom reads):** en una segunda consulta aparecen nuevas filas que no estaban en la primera, insertadas por otra transacción.

Los SGBD controlan esto con **bloqueos (locks)** y **niveles de aislamiento** (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE). A mayor aislamiento, más seguridad frente a anomalías, pero más posible impacto en el rendimiento.

Como futuro profesional, lo clave es que entiendas que las transacciones no son un extra: son la base de la fiabilidad de cualquier sistema que maneje dinero, stock, reservas, notas, etc. Sin transacciones, los datos dejan de ser fiables y el negocio deja de ser creíble.



# Esquema Visual: De la operación bancaria al COMMIT/ROLLBACK

Vamos a representar una transferencia bancaria entre dos cuentas usando una transacción SQL. El objetivo es que veas el flujo completo: inicio, operaciones, decisión entre COMMIT o ROLLBACK.


- START TRANSACTION (A)

  marca el inicio de la unidad de trabajo. A partir de aquí, el SGBD trata todo lo que hagas como una sola transacción.

- Paso 1 – UPDATE cuenta origen (B)

  restas 100 € de la cuenta 1.

  Si este UPDATE falla (no existe la cuenta, error de bloqueo, cualquier problema), pasas por el camino NO de la decisión (C) y ejecutas ROLLBACK (F). Resultado: la base de datos vuelve exactamente al estado anterior al START TRANSACTION.

- Paso 2 – UPDATE cuenta destino (D)

  sumas 100 € a la cuenta 2.

  Si aquí hay un fallo (cuenta no encontrada, violación de constraint, etc.), el flujo te lleva de la decisión (E) a ROLLBACK (F). El dinero no sale de la cuenta origen: se deshace también el primer UPDATE.

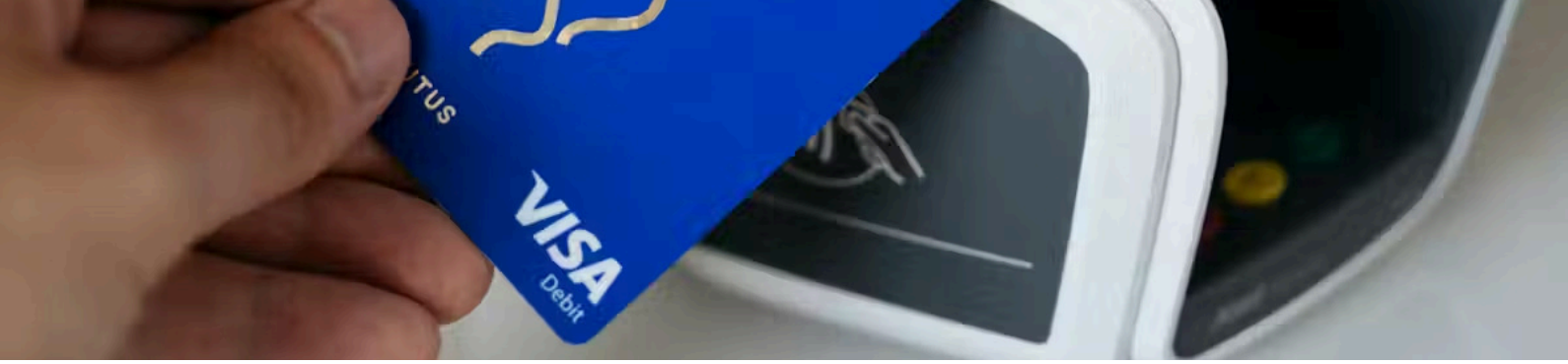
- COMMIT (G)

  solo si los dos pasos se ejecutan correctamente se llega al nodo de confirmación. A partir del COMMIT, los nuevos saldos quedan guardados de forma permanente y visibles para el resto de transacciones.

## Qué representa cada parte del esquema:

En segundo plano, el SGBD está manejando bloqueos sobre las filas de Cuentas implicadas. Mientras la transacción está en curso, otras transacciones pueden tener que esperar para evitar inconsistencias (por ejemplo, que otra operación lea un saldo "a medias"). Ese es el trabajo del motor de concurrencia y de los niveles de aislamiento.

Este mismo patrón se aplica en mil contextos: reservas de hotel, compra de entradas, gestión de stock en un ecommerce, matriculación en asignaturas, etc. Siempre que una operación requiera varios pasos que deben ir juntos, piensa automáticamente en "esto debe ir dentro de una transacción".




# Caso de Estudio: El procesamiento de transacciones de Visa

## Contexto

Visa procesa miles de transacciones por segundo en todo el mundo: compras en comercio físico, pagos online, suscripciones, devoluciones… Piensa en la cantidad de bancos, divisas, horarios y situaciones diferentes. El sistema no puede permitirse perder dinero, duplicar cargos ni dejar cuentas en estados contradictorios. Una caída de coherencia en un sistema así no es solo un problema técnico: puede significar pérdidas millonarias y pérdida masiva de confianza de usuarios y entidades financieras.

## Estrategia: transacciones ACID en cadena

Cada pago con tarjeta se trata como una transacción ACID completa, aunque por debajo implique múltiples sistemas y pasos:

### Autorización inicial

El banco emisor verifica:

- Que la tarjeta es válida.
- Que hay saldo o crédito disponible.
- Que no hay bloqueos por fraude.

Todo esto forma parte del inicio lógico de la transacción.

### Reserva del importe

Si la autorización es correcta, se realiza una retención en tu cuenta: se "marca" ese dinero como comprometido. En términos de BD, se actualizan saldos, se crean registros de movimiento, se generan logs, etc.

### Liquidación entre bancos

Más tarde (a veces en diferido), el sistema de Visa coordina la transferencia real de fondos entre el banco emisor (el tuyo) y el banco del comercio. Esto implica muchas operaciones en distintas bases de datos, pero conceptualmente siguen formando parte de una única transacción lógica de pago.

### Confirmación o cancelación

Si todo el flujo se completa sin problemas, se produce un COMMIT lógico: el cargo se consolida, el saldo se actualiza definitivamente y el comercio recibe su dinero.

Si en algún punto hay un fallo grave (error de comunicación, detección de fraude, fondos insuficientes reales, etc.), el sistema dispara un ROLLBACK lógico: la operación se cancela y el dinero retenido se libera.

## Impacto

Gracias a este enfoque:

- Los saldos de los clientes se mantienen siempre consistentes, incluso con miles de operaciones concurrentes.
- No se producen escenarios como "me han cobrado y al comercio no le ha llegado" o "me han cobrado dos veces por el mismo pago" (cuando pasa es por incidencias puntuales que se corrigen precisamente gracias a los registros de transacciones).
- La durabilidad está garantizada: un pago confirmado sigue siendo válido incluso si se cae un datacenter justo después, porque el sistema se apoya en logs de transacciones y mecanismos de recuperación.

El caso de Visa ilustra el mensaje clave de la sesión: sin transacciones ACID bien diseñadas, ningún sistema financiero global sería viable.

# Herramientas y Consejos para tu Futuro Profesional

Para que puedas llevar estos conceptos a la práctica, aquí van recomendaciones concretas y herramientas específicas:

## Practica las transacciones desde la consola del SGBD

En PostgreSQL, puedes usar psql:

```
BEGIN;
UPDATE cuentas SET saldo = saldo - 50 WHERE id_cuenta = 1;
UPDATE cuentas SET saldo = saldo + 50 WHERE id_cuenta = 2;
COMMIT; -- o ROLLBACK;
```

En MySQL/MariaDB, lo mismo desde el cliente CLI o MySQL Workbench:

```
START TRANSACTION;
...
COMMIT;
```

En SQL Server, con SQL Server Management Studio (SSMS):

```
BEGIN TRANSACTION;
...
COMMIT TRANSACTION;
```

Acostúmbrate a probar primero en un entorno de desarrollo y a observar qué pasa si haces ROLLBACK.

## Usa herramientas gráficas para ver el estado de las transacciones y bloqueos

- pgAdmin (PostgreSQL) permite ver sesiones activas, consultas en curso y bloqueos.
- SQL Server Management Studio tiene vistas de actividad donde puedes identificar transacciones que bloquean a otras o que se han quedado "colgadas".
- En MySQL/MariaDB, puedes consultar vistas del sistema como INFORMATION_SCHEMA.INNODB_LOCKS o usar interfaces gráficas en MySQL Workbench.

Esto es muy útil para diagnosticar problemas de concurrencia en entornos reales.

## Entiende los niveles de aislamiento y configúralos según el caso

En PostgreSQL o MySQL (InnoDB) puedes usar:

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
...
COMMIT;
```

En SQL Server:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION;
...
COMMIT;
```

Para informes o consultas de lectura pesada quizá te interese un aislamiento menor (por ejemplo, READ COMMITTED) para mejorar el rendimiento. Para operaciones críticas (cierres de caja, cierres contables), niveles más altos (REPEATABLE READ o SERIALIZABLE) pueden ser necesarios. El equilibrio entre coherencia y rendimiento es parte del trabajo profesional.

## Integra bien las transacciones en tu código de aplicación

Si usas un ORM como Hibernate (Java), Entity Framework (.NET) o Django ORM (Python), revisa cómo gestiona las transacciones:

En muchos casos puedes usar bloques de contexto, por ejemplo, en Django:

```
from django.db import transaction

with transaction.atomic():
    # operaciones de BD
```

Entiende cómo tu framework maneja autocommit y cómo definir bloques atómicos. No des por hecho que el framework "lo hace todo bien"; tienes que saber qué pasa por debajo.

## Simula problemas de concurrencia en entornos de prueba

- Abre dos conexiones diferentes (por ejemplo, dos ventanas de DBeaver o dos terminales de psql).
- Lanza transacciones que actualicen las mismas filas y mira:
  - Si una se queda esperando a la otra.
  - Qué pasa si haces COMMIT en una y luego consultas desde la otra.

Jugar con esto una tarde te da mucha intuición de cómo se comporta el motor y te prepara para resolver incidencias reales.

# Mitos y Realidades

### ❌ Mito: "Las transacciones hacen que la base de datos sea más lenta, así que es mejor no usarlas."

→ **FALSO.** Es cierto que las transacciones y los bloqueos tienen un coste, pero el objetivo de una base de datos no es ser "rápida a cualquier precio", sino ser fiable. En aplicaciones críticas (banca, stock, reservas, nóminas…) los errores de consistencia son muchísimo más caros que cualquier milisegundo ganado. Los motores modernos están optimizados para manejar transacciones de forma eficiente; la clave está en diseñar transacciones cortas y bien acotadas, no en evitarlas.

### ❌ Mito: "Puedo implementar la lógica de COMMIT/ROLLBACK en mi aplicación sin usar transacciones del SGBD."

→ **FALSO.** Intentar simular transacciones en la capa de aplicación ("si algo falla, borro lo que he insertado antes") es muy peligroso. Tu código:

- No controla los fallos de red a mitad de operación.
- No puede coordinar bien la concurrencia con otros procesos.
- No tiene acceso a los logs internos de la BD para garantizar durabilidad.

Solo el SGBD tiene la información y los mecanismos necesarios (journaling, redo logs, locking interno) para garantizar de verdad las propiedades ACID. La capa de aplicación debe pedir y usar las transacciones del SGBD, no intentar sustituirlas.

## Resumen Final para el Examen

- Una transacción es una unidad de trabajo atómica: o se ejecuta completa o se deshace; se gestiona con START TRANSACTION/BEGIN, COMMIT y ROLLBACK.
- El modelo ACID garantiza Atomicidad, Consistencia, Aislamiento y Durabilidad, clave para operaciones como transferencias bancarias o reservas.
- La concurrencia (muchos usuarios a la vez) se controla con bloqueos y niveles de aislamiento; mal gestionada, causa lecturas sucias, no repetibles o fantasmas.
- Las transacciones no son opcionales en sistemas serios: son la base de la fiabilidad de los datos; la lógica de commit/rollback debe delegarse siempre en el SGBD.

## Sesión 25 – Seguridad avanzada en BD: auditoría, backups, restauración y protección de datos

# Seguridad avanzada en la administración de bases de datos

Cuando administras una base de datos en un entorno profesional, la seguridad no se limita a controlar quién accede a qué. La seguridad real implica anticipar riesgos, monitorear acciones, crear defensas múltiples y planificar cómo recuperarte si algo falla. Este enfoque se conoce como **seguridad en profundidad** y se basa en varias capas que se complementan entre sí.

La primera capa avanzada es la **auditoría**, que responde a tres preguntas esenciales: ¿Quién ha accedido a los datos? ¿Qué ha hecho? ¿Cuándo? Los SGBD modernos permiten activar auditorías específicas: cada UPDATE en una tabla sensible, cada intento de conexión, cada modificación de permisos… Esta información es clave en dos escenarios:

- cuando quieres detectar comportamientos anómalos (fraude interno, accesos irregulares), y
- cuando debes demostrar ante auditores o instituciones regulatorias que tu empresa cumple con la ley (GDPR, PCI-DSS, normas internas del sector).

La segunda capa es la **resiliencia**, que depende directamente de una buena estrategia de backups y restauración. Los backups no son un trámite administrativo, sino un pilar para la supervivencia del negocio. Un fallo físico del servidor, un ransomware, un error humano al borrar una tabla o una aplicación que se comporta mal pueden dejar tu base de datos inoperativa.

Los tipos de backup más comunes son:

### Completo

copia toda la base.

### Diferencial

copia los cambios desde el último completo.

### Incremental

copia solo los cambios desde el último backup de cualquier tipo.

Una buena política combina estos tipos para equilibrar espacio, tiempo y velocidad de recuperación. La restauración es igual de importante. No basta con tener backups: tienes que asegurarte de que puedes restaurarlos, y hacerlo rápido. Esto se mide con dos métricas profesionales:

### RPO (Recovery Point Objective)

cuántos datos puedes permitirte perder medidos en tiempo (por ejemplo, "podemos perder como máximo 15 minutos de actividad").

### RTO (Recovery Time Objective)

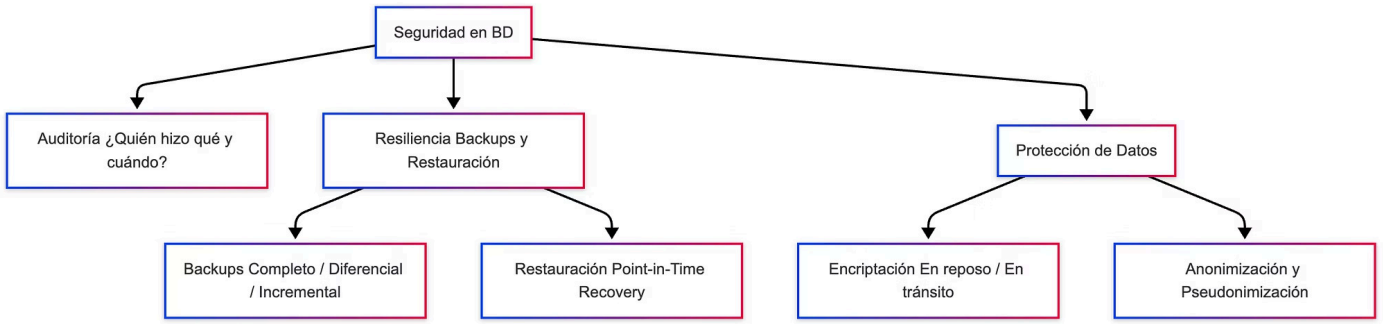
cuánto tiempo puedes tardar en recuperar el sistema antes de que el impacto sea crítico.

La tercera capa es la **protección de datos**, que protege incluso en caso de fuga o ataque. Aquí entran la **encriptación en reposo** (los archivos de BD en el disco están cifrados), la **encriptación en tránsito** (toda conexión BD-aplicación viaja cifrada con TLS), y la **anonimización o pseudonimización** de datos sensibles, indispensable cuando los programadores necesitan entornos de pruebas o cuando se comparten datos para análisis.

Cuando combinas auditoría + resiliencia + protección de datos, creas un sistema de seguridad completo que no confía en una sola barrera. Si una falla, otra protege. Por eso los administradores serios consideran estas prácticas como imprescindibles y no como "extras opcionales".

# Esquema Visual: Seguridad avanzada como arquitectura de defensa en profundidad

A continuación tienes un diagrama Mermaid que muestra cómo se estructuran las capas de seguridad mencionadas:




## Cómo interpretar el esquema:

- El nodo central "Seguridad en BD" representa la estrategia global.
- La primera rama, **Auditoría**, cubre el registro de acciones críticas.
- La segunda, **Resiliencia**, se divide en:
  - **Backups**, que almacenan copias de seguridad periódicas.
  - **Restauración**, que permite volver a un estado anterior, incluso a un momento concreto mediante "Point-in-Time Recovery".
- La tercera rama, **Protección de Datos**, contiene:
  - **Encriptación**, que asegura datos en reposo y en tránsito.
  - **Anonimización**, que protege información sensible cuando los datos se usan fuera del entorno de producción.

El objetivo del esquema es mostrar que la seguridad no depende de un solo punto, sino de múltiples mecanismos alineados entre sí.

# Caso de Estudio: Seguridad avanzada en Microsoft Azure SQL Database

## Contexto

Miles de empresas globales, desde hospitales hasta fintechs, alojan sus bases de datos en la nube de Microsoft Azure. Estas bases contienen datos personales, financieros y sanitarios, lo que requiere cumplir normativas como GDPR, HIPAA o PCI-DSS. Un error o una brecha de seguridad podría generar pérdidas económicas enormes y comprometer la confianza del cliente.

## Estrategia: seguridad multicapa por defecto

Azure SQL Database implementa un enfoque de seguridad completo:

### Auditoría detallada

Los administradores pueden activar auditorías para registrar:

- accesos,
- consultas sobre tablas sensibles,
- cambios en la configuración,
- ejecuciones de procedimientos críticos.

Estos logs pueden enviarse a Azure Storage, Log Analytics o un SIEM corporativo.

### Backups automáticos y redundantes

Azure realiza backups automáticos sin requerir intervención del usuario:

- copias completas, diferenciales e incrementales,
- almacenamiento georredundante,
- periodos de retención configurables,
- restauración a un punto exacto de los últimos 35 días.

Esto permite recuperar una BD minutos antes de un borrado accidental o de una corrupción.

### Encriptación en reposo y en tránsito

- **En reposo:** Transparent Data Encryption (TDE) cifra automáticamente los archivos físicos.
- **En tránsito:** todas las conexiones utilizan TLS, evitando que un atacante intercepte datos.

### Protección avanzada del dato (Data Masking)

Azure permite aplicar Dynamic Data Masking, mostrando datos sensibles ofuscados a usuarios con menor privilegio (por ejemplo, ****1234 en números de tarjeta).

## Impacto

- Las empresas cumplen requisitos regulatorios sin necesidad de configurar manualmente todas las capas de seguridad.
- Los datos están protegidos incluso ante robo físico o accesos no autorizados.
- La restauración rápida garantiza continuidad de negocio ante fallos.
- La auditoría permite detectar accesos irregulares y presentar evidencias ante auditorías externas.

En conjunto, Azure demuestra cómo una arquitectura moderna de bases de datos integra seguridad, recuperación y privacidad desde la base.

# Herramientas y Consejos para tu Futuro Profesional

## Define la estrategia de backups usando RPO y RTO desde el principio

Antes de crear una política de copias, pregúntate:

- ¿Cuánta pérdida de datos puedo permitir? (RPO)
- ¿Cuánto tiempo puedo estar caído? (RTO)

Un ecommerce puede tolerar perder 10 minutos de datos; un hospital, quizá ninguno.

## Usa herramientas nativas del SGBD para asegurar fiabilidad

- MySQL/MariaDB → mysqldump, mysqlpump
- PostgreSQL → pg_dump, pg_basebackup
- SQL Server → Copias automáticas desde SSMS o T-SQL (BACKUP DATABASE)
- Oracle → RMAN (muy robusto para backups corporativos)

Evita herramientas no oficiales o scripts improvisados en producción.

## Nunca lleves datos reales a entornos de pruebas

- Aplica anonimización antes de exportar datos.
- Herramientas útiles: pg_dump + scripts de anonimización, Faker, Mockaroo, Python pandas para sustituir campos sensibles.

## Encripta siempre la comunicación BD-aplicación

- Habilita TLS o SSL en MySQL, PostgreSQL o SQL Server.
- Si trabajas en la nube, revisa que las conexiones sean forzadas a cifrado.

## Documenta y prueba la restauración periódicamente

- Un backup que nunca se prueba puede estar corrupto sin saberlo.
- Programa pruebas de restauración trimestrales o mensuales.
- Crea un procedimiento claro para recuperar la base en caso de emergencia.

# Mitos y Realidades

### ❌ Mito: "Con hacer un backup diario ya estoy protegido."

→ **FALSO.** Un backup diario significa que podrías perder hasta 24 horas de datos. Si tu RPO es de 1 hora, tu estrategia es inaceptable. Además, no sirve de nada tener copias si nunca pruebas su restauración. Es habitual descubrir en el peor momento que un backup llevaba meses corrupto o incompleto.

### ❌ Mito: "Si la base de datos está encriptada, ya no pueden robar los datos."

→ **FALSO.** La encriptación no evita que un atacante con credenciales legítimas (por phishing, mala configuración, fuga interna) acceda a los datos ya descifrados. La encriptación protege si:

- te roban el disco,
- acceden al backup en bruto,
- interceptan tráfico sin TLS.

Pero no sustituye un buen control de permisos ni evita vulnerabilidades como la inyección SQL. La seguridad debe ser integral, no confiar solo en una capa.

## Resumen Final para el Examen

- La seguridad avanzada incluye auditoría, backups y restauración, y protección de datos (encriptación y anonimización).
- Los backups pueden ser completos, diferenciales o incrementales; la restauración debe planificarse según RPO y RTO.
- La encriptación protege datos en reposo y en tránsito, pero no sustituye a un buen control de accesos.
- La auditoría registra acciones críticas para detectar incidentes y cumplir normativas.
- Nunca uses datos reales en entornos de pruebas; anonimiza siempre.