Transacciones

Lo que probablemente más importa de tu DB

(y sin animaciones)

Qué son las transacciones?

Una transacción es una unidad lógica de trabajo que agrupa una o más operaciones de base de datos.

Son:

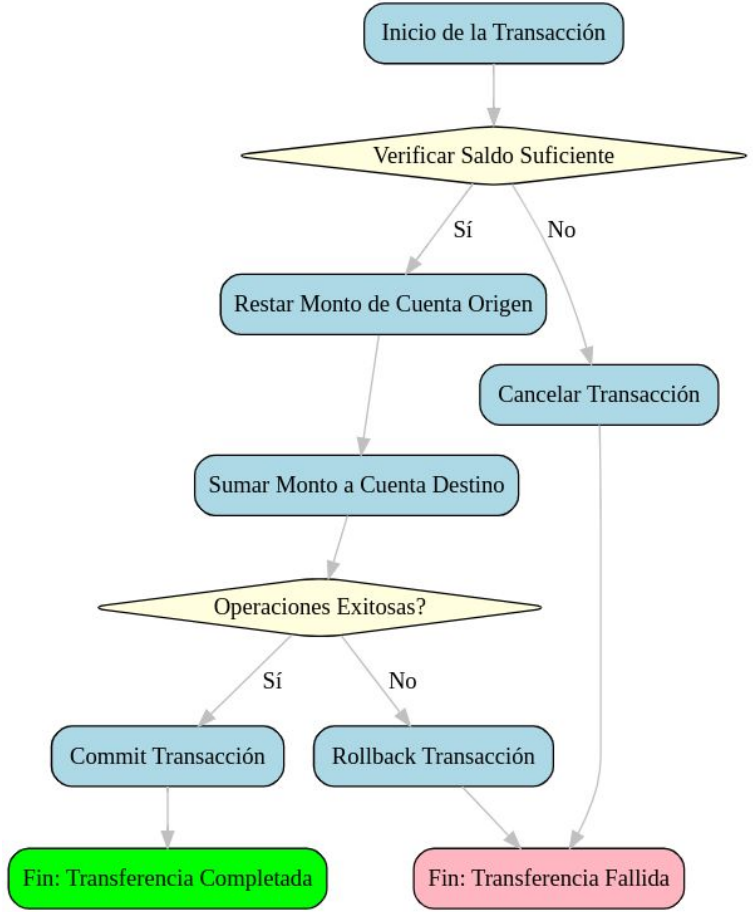
- Indivisible: se ejecuta completamente o no se ejecuta en absoluto.
- Mantiene la integridad de los datos.
- Garantiza la consistencia de la base de datos.
- Protege contra fallos del sistema.
- Mantiene la exactitud en operaciones concurrentes.
- Facilita la recuperación en caso de errores.

Ejemplo de transacción

Transferencia bancaria

1. Restar dinero de una cuenta
2. Sumar dinero a otra cuenta

Ambas operaciones deben completarse o ninguna debe realizarse.



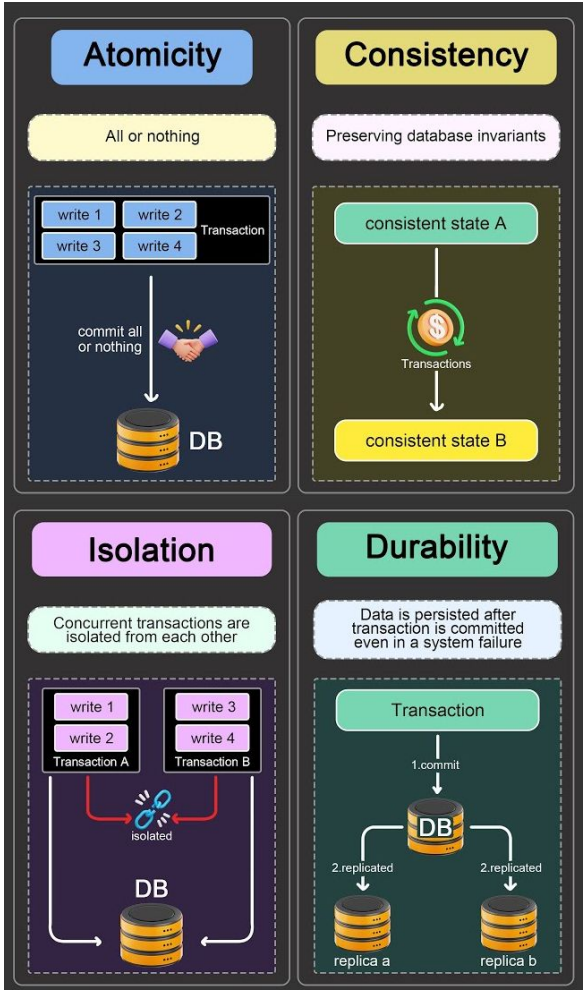
ACID

Atomicidad

- "Todo o nada": La transacción se completa totalmente o se deshace completamente.
- Ejemplo: Si falla la suma en la cuenta destino, se deshace la resta de la cuenta origen.

Consistencia

- La base de datos pasa de un estado válido a otro válido: Se mantienen todas las reglas de integridad.
- Ejemplo: El total de saldos antes y después de la transacción debe ser el mismo.



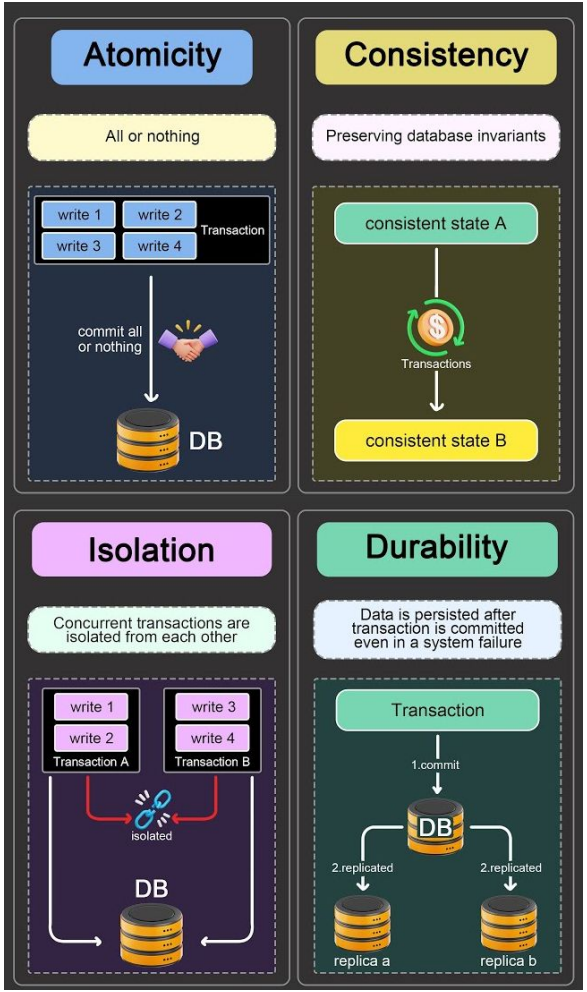
ACID

Aislamiento

- Las transacciones se ejecutan de forma aislada unas de otras.
- Los efectos intermedios de una transacción no son visibles para otras transacciones.
- Niveles de aislamiento: Read Uncommitted, Read Committed, Repeatable Read, Serializable.

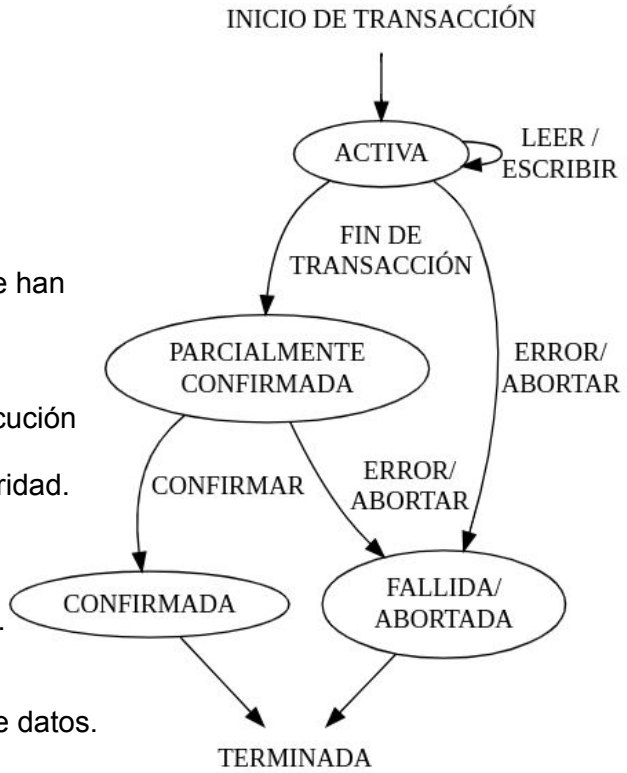
Durabilidad

- Una vez que una transacción se ha completado, sus efectos son permanentes.
- Los cambios persisten incluso en caso de fallos del sistema.
- Se implementa típicamente mediante logs de transacciones.



Estados de una Transacción

1. **Activa**
   - La transacción ha comenzado pero aún está en ejecución.
   - Las operaciones de lectura y escritura se están realizando.
2. **Parcialmente confirmada**
   - Todas las operaciones se han ejecutado, pero los cambios aún no se han guardado permanentemente.
   - Estado justo antes de la confirmación (commit).
3. **Fallida**
   - Se ha producido un error y la transacción no puede continuar su ejecución normal.
   - Puede deberse a errores lógicos, del sistema, o violaciones de integridad.
4. **Abortada**
   - La transacción ha sido rollback (deshecha).
   - Todos los cambios realizados por la transacción se han deshecho.
   - La base de datos vuelve al estado anterior al inicio de la transacción.
5. **Confirmada**
   - La transacción se ha completado con éxito.
   - Todos los cambios se han guardado permanentemente en la base de datos.
   - No se puede deshacer a partir de este punto.



Ejemplos de transacciones

Se confirma con **COMMIT**

```
BEGIN TRANSACTION;
  UPDATE Cuenta SET saldo = saldo - 100 WHERE id = 1;
  UPDATE Cuenta SET saldo = saldo + 100 WHERE id = 2;
COMMIT;
```

Se aborta con **ROLLBACK**

```
BEGIN TRANSACTION;
  UPDATE Producto SET stock = stock - 1 WHERE id = 5;
  UPDATE Pedido SET estado = 'Enviado' WHERE id = 100;
  -- Supongamos que aquí ocurre un error
  ROLLBACK;
```

Tipos de Transacciones

Planas

- La forma más simple de transacción.
- Consiste en una secuencia de operaciones tratadas como una única unidad.
- Se ejecuta completamente o se deshace completamente.

Anidadas

Distribuidas

```sql
BEGIN TRANSACTION;

UPDATE Cuentas
SET saldo = saldo - 1000
WHERE numero_cuenta = '123456';

UPDATE Cuentas
SET saldo = saldo + 1000
WHERE numero_cuenta = '789012';

INSERT INTO Transferencias (cuenta_origen, cuenta_destino, monto, fecha)
VALUES ('123456', '789012', 1000, CURRENT_TIMESTAMP);

COMMIT;
```

Tipos de Transacciones

Planas

Anidadas

- Contiene subtransacciones dentro de una transacción principal.
- En el ejemplo, se muestra un proceso de compra con tres pasos: actualizar inventario, procesar pago y registrar envío.
- Cada paso es una subtransacción (marcada con **`SAVEPOINT`**).
- Permite un control más granular: si falla un paso, se puede deshacer solo ese paso sin afectar a los demás.
- Se puede volver atrás a un savepoint con **`ROLLBACK TO <SAVEPOINT>`**

Distribuidas

```sql
BEGIN TRANSACTION; -- Transacción principal

    -- Paso 1: Actualizar inventario
    SAVEPOINT actualizar_inventario;
    UPDATE Productos SET stock = stock - 1 WHERE id_producto = 101;

    -- Paso 2: Procesar pago
    SAVEPOINT procesar_pago;
    INSERT INTO Pagos (id_cliente, monto, fecha) VALUES (1, 50.00, CURRENT_TIMESTAMP);
    UPDATE Clientes SET saldo = saldo - 50.00 WHERE id_cliente = 1;

    -- Paso 3: Registrar envío
    SAVEPOINT registrar_envio;
    INSERT INTO Envios (id_cliente, id_producto, fecha_envio) VALUES (1, 101, CURRENT_TIMESTAMP);

    -- Si todo está bien, confirmar la transacción principal
    COMMIT;
```

Ejemplo:

`ROLLBACK TO procesar_pago;`

Tipos de Transacciones

Planas

Anidadas

Distribuidas

- Involucra operaciones en múltiples bases de datos o sistemas.
- El ejemplo muestra una reserva de vuelo y hotel en sistemas separados.
- Requiere un coordinador de transacciones distribuidas para asegurar que ambas partes se completen o se deshagan juntas.
- Es más compleja de implementar y gestionar debido a la necesidad de coordinación entre sistemas.

ESTO ES PSEUDOCÓDIGO

```
BEGIN DISTRIBUTED TRANSACTION;

-- En el sistema de la aerolínea
CONNECT TO airline_db;
BEGIN TRANSACTION;
    UPDATE Vuelos SET asientos_disponibles = asientos_disponibles - 1 WHERE id_vuelo = 'FL123';
    INSERT INTO Reservas (id_cliente, id_vuelo, fecha) VALUES (1, 'FL123', CURRENT_TIMESTAMP);
COMMIT;

-- En el sistema del hotel
CONNECT TO hotel_db;
BEGIN TRANSACTION;
    UPDATE Habitaciones SET estado = 'Reservada' WHERE id_habitacion = 'H789';
    INSERT INTO Reservas (id_cliente, id_habitacion, fecha_entrada, fecha_salida)
    VALUES (1, 'H789', '2023-06-01', '2023-06-05');
COMMIT;

-- Si ambas transacciones son exitosas
COMMIT DISTRIBUTED TRANSACTION;
```

OJO!

- Las bases de datos distribuidas no suelen ser ACID, sino **BASE** (Estudienlo lo necesitan por el laburo).
- También lean **Teorema CAP**.

Problemas de Concurrencia

1. **Lectura Sucia (Dirty Read)**
   - Una transacción lee datos que no han sido confirmados por otra transacción.
2. **Lectura No Repetible (Non-Repeatable Read)**
   - Una transacción lee los mismos datos dos veces y obtiene valores diferentes.
3. **Lectura Fantasma (Phantom Read)**
   - Una transacción re-ejecuta una consulta y obtiene un conjunto diferente de filas.

OJO: en todos los caso dice “**una Transacción**”, no dice “una consulta”

Problemas de Concurrencia

Lectura Fantasma: El Último Croissant

- Alice ve el croissant pero se distrae con su teléfono
- Bob ve el croissant y lo compra inmediatamente
- Alice intenta comprar el croissant que ya no está

Problema: Alice vio datos que cambiaron antes de que pudiera actuar sobre ellos

Lectura Sucia: El Croissant en Oferta

- Sophie está actualizando precios en el sistema
- Cambia temporalmente el precio de 5 a 4 euros
- Alice ve el precio de 4 euros y compra rápidamente
- Sophie revierte el cambio de precio
- Bob entra y ve el precio normal de 5 euros

Problema: Alice actúa sobre datos no confirmados (lectura sucia)m Confusión entre Alice y Bob sobre el precio real

Lectura No Repetible: El Croissant Cambiante

- Sophie implementa un sistema de precios dinámicos
- El precio cambia según la demanda
- Alice ve el precio de 5 euros y decide volver en 10 minutos
- El sistema actualiza el precio a 6 euros debido a la demanda
- Alice regresa y se sorprende por el nuevo precio

• **Problema:** Alice experimenta una "lectura no repetible"

Niveles de aislamiento (No todas andan en todas los rbdms)

Read uncommitted isolation: Permite leer los datos que están siendo modificados por otras transacciones, incluso si éstas no han terminado o confirmado sus cambios.

Read committed isolation: Impide leer los datos que están siendo modificados por otras transacciones hasta que éstas terminen y confirmen sus cambios.

Read committed snapshot isolation (RCSI): Es, simplemente, una variación de Read Committed que en vez de generar bloqueos genera snapshots. Los snapshots son la versión original de los datos modificados, se almacenan en tempdb mientras dure la transacción bloqueante.

Repeatable read isolation: Este nivel garantiza que si una transacción lee un registro, nadie podrá modificarlo hasta que la transacción termine.

Serializable isolation: Impide cualquier modificación concurrente de los datos que lee una transacción.

Snapshot isolation: Este nivel permite leer los datos tal y como estaban al inicio de la transacción.

Niveles de aislamiento (No todas andan en todas los rbdms)

	Lectura Sucias	Lecturas Repetibles	Lecturas Fantasma
READ COMMITTED	NO	NO	SI
READ COMMITED SNAPSHOT	NO	NO	SI
READ UNCOMMITTED	**SI**	**NO**	**SI**
REPETIBLE READ	NO	SI	SI
SNAPSHOT*	NO	SI	NO
SERIALIZABLE	NO	SI	NO

	READ COMMITTED	READ COMMITED SNAPSHOT	**READ UNCOMMITTED**	REPETIBLE READ	SNAPSHOT*	SERIALIZABLE
Lector BLOQUEA A Lector	NO	NO	**NO**	NO	NO	NO
Lector BLOQUEA A Escritor	SI	NO	**NO**	SI	NO	SI**
Escritor BLOQUEA A Lector	SI	NO	**NO**	SI	NO	SI**
Escritor BLOQUEA A Escritor	SI	SI	**SI**	SI	ERROR	SI*

Niveles de aislamiento - MySQL

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] …

transaction_characteristic: {ISOLATION LEVEL level | access_mode}
level: {REPEATABLE READ  | READ COMMITTED | READ UNCOMMITTED | SERIALIZABLE}
access_mode: {READ WRITE | READ ONLY}
```

Ejemplo:

```
mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Técnicas de control de Concurrencia

Bloqueos (Locking)

- Bloqueos compartidos (para lectura) y exclusivos (para escritura).
- Bloqueos de dos fases: fase de crecimiento y fase de liberación.
- Pueden llevar a deadlocks (interbloqueos).

Mysql ofrece varios mecanismos como

- "`SELECT … FOR UPDATE …`": Otras transacciones no pueden hacer updates en las filas seleccionadas
- "`SELECT … FOR SHARE …`" Establece un bloqueo en modo compartido en cualquier fila que sea leída. Otras sesiones pueden leer las filas, pero no pueden modificarlas hasta que tu transacción haga commit.
- `LOCK TABLES` y `UNLOCK TABLES`

`hay muchas/muchísimas más opciones`

Técnicas de control de Concurrencia

Además del lock hay estas otras dos técnicas necesarias para niveles de aislamiento mas complejos:

Marcas de Tiempo (Timestamping)

- Cada transacción recibe una marca de tiempo única.
- Se utilizan para ordenar la ejecución de las transacciones.
- Evita deadlocks pero puede llevar a muchos abortos.

Control de Versiones Multiversión (MVCC)

- Mantiene múltiples versiones de los datos.
- Permite a los lectores acceder a versiones consistentes sin bloquear a los escritores.
- Utilizado en sistemas como **PostgreSQL y Oracle.**

Detalles

- TODO (**TODO**) en una BDD sucede dentro de una transacción.
- En las consultas que están corriendo tienen configurado “`autocomit`” en `true`.
- Los frameworks web suelen iniciar una transacción antes de arrancar una “vista” y dan “commit/rollback” según si sucedio una excepción o no
- Las transacciones son fundamentalmente para DML. Algunas bases de datos como Oracle y Postgres tienen DDL transaccional (que es genial!)