Transacciones

Bases de Datos 1 (CC3088) - 2026

Transacciones en PostgreSQL

Semestre 01, 2026

Problemática

Dos cajeros procesan operaciones simultáneamente sobre la misma cuenta.

El sistema falla a la mitad de una transferencia.

¿Los datos quedan en un estado inconsistente?

Ejemplo

Se transfieren Q500 de la cuenta de Ana a la de Carlos.

```
UPDATE cuentas SET saldo = saldo - 500 WHERE nombre = 'Ana';
-- El sistema falla aquí
UPDATE cuentas SET saldo = saldo + 500 WHERE nombre = 'Carlos';
```

- Ana perdió Q500.
- Carlos no recibió nada.
- Los datos quedaron inconsistentes.

Se necesita una forma de garantizar que ambas operaciones ocurran juntas o ninguna.

Transacción

Una transacción es un conjunto de operaciones SQL que se ejecutan como una unidad.

O todas se completan correctamente, o ninguna tiene efecto.

Analogía

Una transacción es como un contrato.

- Se hacen todas las operaciones acordadas.
- Si algo falla, se regresa al estado inicial.
- No existe un estado intermedio visible para otros.

Lo que garantiza

- Que los datos nunca quedan a medias.
- Que una falla no deja inconsistencias.
- Que múltiples operaciones relacionadas se traten como una sola.

ACID

Las transacciones cumplen cuatro propiedades fundamentales.

Conocidas como ACID.

Atomicity — Atomicidad

Todo o nada.

- Si todas las operaciones tienen éxito → los cambios se aplican.
- Si alguna falla → se deshacen todas las anteriores.

No existen estados intermedios.

Consistency — Consistencia

La base de datos siempre pasa de un estado válido a otro estado válido.

- Las restricciones, claves foráneas y reglas se mantienen.
- Una transacción no puede dejar datos que violen las reglas del esquema.

Isolation — Aislamiento

Las transacciones en ejecución no se interfieren entre sí.

- Los cambios de una transacción no son visibles para otras hasta que se confirmen.
- Dos transacciones concurrentes producen el mismo resultado que si se ejecutaran en serie.

Durability — Durabilidad

Una vez confirmados, los cambios persisten.

- Un fallo del sistema después del `COMMIT` no revierte los datos.
- Los cambios se escriben en disco antes de confirmar.

Sintaxis básica

BEGIN

Inicia la transacción. Todo lo que sigue es parte de ella.

```
BEGIN;
```

COMMIT

Confirma todos los cambios. Los hace permanentes y visibles para otros.

```
COMMIT;
```

ROLLBACK

Cancela todos los cambios desde el `BEGIN`.

```
ROLLBACK;
```

Transacciones implícitas

En PostgreSQL, cada sentencia sin `BEGIN` es automáticamente una transacción.

```
-- Esto
INSERT INTO pedidos (cliente_id, total) VALUES (5, 1200);

-- es equivalente a:
BEGIN;
INSERT INTO pedidos (cliente_id, total) VALUES (5, 1200);
COMMIT;
```

Una sola sentencia siempre es atómica por sí misma.

Para agrupar varias operaciones se debe usar `BEGIN` explícitamente.

La transferencia correcta

```
BEGIN;

UPDATE cuentas SET saldo = saldo - 500
WHERE nombre = 'Ana';

UPDATE cuentas SET saldo = saldo + 500
WHERE nombre = 'Carlos';

COMMIT;
```

¿Qué pasa si falla?

```
BEGIN;

UPDATE cuentas SET saldo = saldo - 500
WHERE nombre = 'Ana';

-- El sistema detecta que Carlos no existe
UPDATE cuentas SET saldo = saldo + 500
WHERE nombre = 'Carlos';

ROLLBACK; -- Se deshace la deducción de Ana
```

Ana conserva sus Q500.

Estados de una transacción

Estado: Active

La transacción está ejecutando comandos.

Ningún cambio es visible para otras sesiones todavía.

Estado: Partially Committed

Todos los comandos se ejecutaron sin errores.

La transacción espera el `COMMIT` para confirmar.

Estado: Committed

El `COMMIT` fue procesado.

Los cambios son permanentes y visibles para todos.

Estado: Failed

Se detectó un error durante la ejecución.

La transacción no puede continuar y espera `ROLLBACK`.

Estado: Aborted

Se ejecutó `ROLLBACK` (manual o automático).

Todos los cambios fueron deshechos. La base queda como antes del `BEGIN`.

Transiciones

- **Active** → comando exitoso → **Partially Committed**
- **Active** → error → **Failed**
- **Partially Committed** → `COMMIT` → **Committed**
- **Partially Committed** → error → **Failed**
- **Failed** → `ROLLBACK` → **Aborted**

Savepoints

Un savepoint define un punto de retorno dentro de una transacción activa.

Permite deshacer parte de la transacción sin cancelarla completamente.

Sintaxis

```
SAVEPOINT nombre_sp;

ROLLBACK TO nombre_sp;  -- deshace solo lo que vino después

RELEASE SAVEPOINT nombre_sp;  -- libera el savepoint
```

Ejemplo: pedido con error en un ítem

```
BEGIN;

INSERT INTO pedidos (cliente_id, total) VALUES (3, 0);

SAVEPOINT antes_detalle;

INSERT INTO pedido_detalle (pedido_id, producto_id, cantidad)
VALUES (currval('pedidos_id_seq'), 99, 2);
-- Error: producto 99 no existe

ROLLBACK TO antes_detalle;

INSERT INTO pedido_detalle (pedido_id, producto_id, cantidad)
VALUES (currval('pedidos_id_seq'), 12, 2);

UPDATE pedidos SET total = 240 WHERE id = currval('pedidos_id_seq');

COMMIT;
```

El pedido se crea correctamente aunque uno de los ítems falló.

Race conditions

Una race condition ocurre cuando dos procesos acceden al mismo recurso compartido al mismo tiempo y el resultado depende de quién llega primero.

El nombre lo dice todo

"Condición de carrera" — dos operaciones compiten por el mismo dato.

El resultado correcto depende del orden de ejecución.

Pero en un sistema concurrente, ese orden no está garantizado.

Race condition clásica fuera de bases de datos

```
# Dos hilos leen y modifican el mismo contador

saldo = 1000  # valor compartido

# Hilo A lee: 1000
# Hilo B lee: 1000
# Hilo A escribe: 1000 - 200 = 800
# Hilo B escribe: 1000 - 300 = 700  ← ignora el cambio de A

# Resultado: 700 (deberían ser 500)
```

Ambos leyeron el valor viejo antes de que el otro escribiera.

Se perdió una operación completa.

El mismo problema en bases de datos

```
-- Dos cajeros procesan retiros simultáneos de la misma cuenta (saldo: Q1000)

-- Cajero A                          -- Cajero B
```

```
SELECT saldo FROM cuentas            SELECT saldo FROM cuentas
WHERE id = 1;  -- 1000               WHERE id = 1;  -- 1000

UPDATE cuentas SET saldo = 800 ...   UPDATE cuentas SET saldo = 700 ...
COMMIT;                              COMMIT;

-- Resultado final: Q700
-- Debería ser: Q500
```

El banco "perdió" un retiro de Q200.

Nadie cometió un error de lógica. El problema fue el orden de ejecución.

Por qué las bases de datos son especialmente vulnerables

Una base de datos puede recibir cientos de transacciones por segundo.

Todas sobre las mismas tablas, las mismas filas, los mismos valores.

Sin mecanismos de control, las race conditions son inevitables.

Las tres formas en que aparece

Las race conditions en bases de datos se manifiestan como tres problemas concretos.

Cada uno es una variante de "dos transacciones interfieren entre sí de forma inesperada".

Los niveles de aislamiento existen precisamente para eliminar estas interferencias.

Niveles de aislamiento

El aislamiento protege contra tres problemas de concurrencia:

Dirty Read

Leer datos no confirmados de otra transacción.

Una transacción ve cambios que aún podrían deshacerse.

Nonrepeatable Read

Releer un dato que otra transacción ya modificó.

El mismo `SELECT` produce resultados distintos dentro de la misma transacción.

Phantom Read

Releer una consulta que devuelve filas distintas.

Otra transacción insertó o eliminó filas entre dos lecturas.

El dilema del aislamiento

Más aislamiento = menos problemas.

Pero más aislamiento también = menos concurrencia.

El costo real

| Nivel | Concurrencia | Bloqueos | Rendimiento | |---|---|---|---| | Read Committed | Alta | Mínimos | Máximo | | Repeatable Read | Media | Moderados | Menor | | Serializable | Baja | Agresivos | El más bajo |

No existe aislamiento gratis. Cada nivel sacrifica algo.

¿Qué es concurrencia?

Cuántas transacciones puede procesar el sistema al mismo tiempo.

Un banco procesa miles de transferencias por segundo.

Si cada una bloquea a las demás, el sistema se convierte en una fila de un solo cajero.

Niveles

Read Committed

Nivel por defecto en PostgreSQL.

Protege contra: Dirty Read

Sacrifica: Lecturas pueden cambiar dentro de la misma transacción

Cuándo usarlo: Operaciones cotidianas — inserts, updates, consultas simples

Por qué es el default: Máxima concurrencia con protección mínima útil

Read Committed — el riesgo

```
-- Transacción A
BEGIN;
SELECT saldo FROM cuentas WHERE id = 1; -- devuelve 1000

-- (en este momento, Transacción B hace COMMIT con saldo = 500)

SELECT saldo FROM cuentas WHERE id = 1; -- devuelve 500 ← cambió
COMMIT;
```

Dos lecturas del mismo dato, dos resultados distintos.

Aceptable si el resultado final no depende de comparar las dos lecturas.

Repeatable Read

Protege contra: Dirty Read, Nonrepeatable Read, Phantom Read

Sacrifica: Otras transacciones deben esperar para modificar las filas que esta transacción leyó

Cuándo usarlo: Reportes que leen múltiples tablas y necesitan un snapshot consistente

Repeatable Read — el riesgo

```
-- Transacción A (Repeatable Read)
BEGIN;
SELECT SUM(saldo) FROM cuentas; -- 50,000

-- Transacción B intenta actualizar una cuenta → espera bloqueada

SELECT SUM(saldo) FROM cuentas; -- 50,000 (consistente)
COMMIT;

-- Solo ahora Transacción B puede continuar
```

Protege la consistencia de A, pero B tuvo que esperar.

En sistemas de alta carga, muchas transacciones esperando = cuello de botella.

Serializable

El nivel más estricto.

Protege contra: Todo — las transacciones se comportan como si se ejecutaran una por una

Sacrifica: Concurrencia real y requiere reintentos

Cuándo usarlo: Operaciones críticas donde una anomalía tendría consecuencias graves

Serializable — el riesgo real

PostgreSQL puede abortar una transacción serializable si detecta un conflicto:

```
ERROR:  could not serialize access due to read/write dependencies
DETAIL:  Process ... waits for ...
HINT:   The transaction might succeed if retried.
```

La aplicación **debe** capturar este error y reintentar la operación.

En carga alta, muchos reintentos = degradación severa del rendimiento.

Serializable — ejemplo de conflicto

```
-- Ambas transacciones leen el mismo dato y escriben basándose en él

-- Transacción A
SELECT COUNT(*) FROM turnos WHERE medico_id = 1 AND fecha = '2026-04-13';
-- devuelve 4, procede a insertar un turno más

-- Transacción B (simultánea)
SELECT COUNT(*) FROM turnos WHERE medico_id = 1 AND fecha = '2026-04-13';
-- también devuelve 4, también procede a insertar

-- Sin Serializable: el médico queda con 6 turnos (límite era 5)
-- Con Serializable: una de las dos transacciones es abortada y debe reintentar
```

Balance

Un sistema bancario que procesa 10,000 transacciones por segundo con Read Committed puede bajar a 500 con Serializable.

El 99% de las operaciones no necesitan ese nivel de protección.

El costo de reintentos y bloqueos supera el riesgo real del problema que previene.

La regla práctica

Usar el nivel **mínimo** que proteja contra los problemas que realmente pueden ocurrir en esa operación específica.

- Insertar un pedido → Read Committed
- Generar un reporte de cierre contable → Repeatable Read
- Asignar un número de turno único sin duplicados → Serializable

Cambiar el nivel

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

-- operaciones...

COMMIT;
```

Errores comunes

Error 1: olvidar el COMMIT

```
BEGIN;
UPDATE cuentas SET saldo = saldo - 500 WHERE nombre = 'Ana';
-- Se cierra la sesión sin COMMIT
```

PostgreSQL hace `ROLLBACK` automático al cerrar la conexión.

Los cambios se pierden.

Error 2: continuar después de un error

```
BEGIN;
INSERT INTO pedidos VALUES (...);
-- Error de constraint
INSERT INTO pedido_detalle VALUES (...); -- no se ejecuta
COMMIT; -- falla: la transacción está en estado Failed
```

Después de un error, PostgreSQL exige `ROLLBACK` antes de iniciar otra transacción.

Error 3: transacciones muy largas

Una transacción abierta por mucho tiempo:

- Bloquea otros procesos que necesitan las mismas filas.
- Impide que PostgreSQL limpie versiones antiguas de los datos (VACUUM).
- Puede causar degradación del rendimiento general.

Buenas prácticas

- Hacer las transacciones lo más cortas posible.
- Nunca dejar una transacción abierta mientras se espera entrada del usuario.
- Usar `SAVEPOINT` cuando una parte de la transacción puede fallar de forma esperada.
- Verificar siempre el resultado de cada operación antes de `COMMIT`.
- En aplicaciones, manejar errores con `ROLLBACK` explícito y reintentar si es necesario.