

Arquitectura de Software

Transacciones

Niveles de aislamiento · Cómo se vende dos veces un asiento · Sagas, idempotencia y outbox

Gerson Garrido

Semana 12

UNIVERSIDAD NACIONAL DE INGENIERÍA

Agenda

Sesión A: dentro de una base de datos

1. Qué garantiza una transacción
2. Cómo se vende dos veces un asiento
3. Niveles de aislamiento
4. Las anomalías que quedan
5. Tres maneras de impedirlo
6. La transacción como recurso

Sesión B: entre servicios

7. Cuando la transacción no alcanza
8. Two-phase commit
9. Sagas
10. Idempotencia
11. La escritura doble y la outbox

Lectura

A: Kleppmann y Riccomini, *Designing Data-Intensive Applications* (2.ª ed.), cap. 8; PostgreSQL 18, *Transaction Isolation* y *Explicit Locking*. **B:** Ford, Richards, Sadalage y Dehghani, *Software Architecture: The Hard Parts*, cap. 9 y 12; Richardson, *Saga* y *Transactional outbox*; Leach, *Designing robust and predictable APIs with idempotency* (Stripe, 2017).

SECCIÓN 01

Qué garantiza una transacción

01

Todo o nada

Reservar una noche son dos escrituras:

1. marcar la noche como reservada;
2. registrar la reserva del huésped.

Si el programa muere entre las dos, queda una noche que nadie puede reservar y que nadie reservó.

Una **transacción** las agrupa: o se aplican todas (`COMMIT`) o ninguna (`ROLLBACK`, o una falla).

Lo que se gana

Un manejo de errores simple: ante cualquier problema, se aborta y no pasó nada. Después se reintenta desde el principio.

ACID, letra por letra

	Garantía	Quién la da
Atomicidad	Todo o nada, también ante una caída	La base de datos
Consistencia	Los invariantes de la aplicación se mantienen	El código, con las restricciones que declare
A**i**slamiento	Las transacciones concurrentes no se estorban	La base de datos, según el nivel elegido
Durabilidad	Lo confirmado no se pierde	La base de datos (WAL, réplicas)

Dónde se decide

La atomicidad habla de fallas, no de concurrencia. Los trade-offs están en el aislamiento.

SECCIÓN 02

Cómo se vende dos veces un asiento

02

El problema, en el Tinkuy inicial

make tinkuy-oversell

```
Asiento 1 del evento 1 (fila A, número 1): 50 órdenes a la vez
  respuestas 201: 50
Órdenes aceptadas para el asiento 1: 50
Entradas emitidas para el asiento 1: 50
```

tinkuy/src/orders/create-order.ts (reducido)

```
1  await client.query('BEGIN');
2  const seats = await findSeatsForSale(client, input.seatIds);  // SELECT ... FROM seats
3  const taken = seats.filter((seat) => seat.status !== 'available');
4  if (taken.length > 0) throw conflict('SEAT_NOT_AVAILABLE', /* ... */);
5  const result = await charge({ /* ... */ });                    // la pasarela: 80 ms
6  await markSeatsSold(client, input.seatIds);                    // UPDATE seats SET status = 'sold'
7  await client.query('COMMIT');
```

Hay `BEGIN` y `COMMIT`: la atomicidad está bien. Lo que falla es el aislamiento.

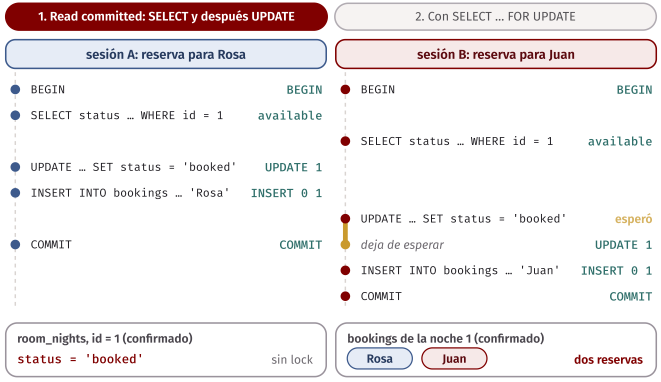
El ejemplo de la semana: un hospedaje

db/schema.sql (reducido)

```sql
CREATE TABLE room_nights (
    id     integer PRIMARY KEY,
    room   integer NOT NULL,
    night  date    NOT NULL,
    status text    NOT NULL DEFAULT 'available' CHECK (status IN ('available', '
     booked'))
);
CREATE TABLE bookings (
    id            integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    room_night_id integer NOT NULL REFERENCES room_nights (id),
    guest         text    NOT NULL
);
```

El invariante: una noche de una habitación tiene a lo más una reserva. Nada aquí lo respalda todavía.

La doble reserva, paso a paso



Una noche, dos reservas: la espera de B no salvó nada.

`make guion G=01` lo repite cuando quieran.

Lo que hizo la base de datos, y lo que no

make guion G=01 (el final)

```
B=# UPDATE room_nights SET status = 'booked' WHERE id = 1;
    (la sesión B queda esperando: el lock lo tiene la sesión A)
A=# COMMIT;
COMMIT
    (la sesión B deja de esperar; su terminal muestra:)
UPDATE 1
...
 id | room_night_id | guest
----+---------------+-------
  1 |             1 | Rosa
  2 |             1 | Juan
(2 rows)
```

- Sí hizo algo: el `UPDATE` de B esperó a que A terminara.
- No alcanzó: al continuar, `WHERE id = 1` seguía cumpliéndose.

Comprobar y después actuar

El programa lee, decide y actúa. Entre la lectura y la acción, el dato cambia.

- `SELECT` asiento libre → `UPDATE`
- `SELECT` correo no registrado → `INSERT`
- `SELECT saldo` → `if` → `UPDATE`

La señal: un `if` de la aplicación entre una lectura y una escritura.

Error común

Creer que “usar transacciones” basta. La pregunta es **qué nivel de aislamiento**, y qué anomalía de ese nivel le hace daño a **este** invariante.

SECCIÓN 03

Niveles de aislamiento

03

Serializable: el ideal y su precio

Serializable

El resultado de ejecutar varias transacciones a la vez es el mismo que el de ejecutarlas una tras otra, en algún orden.

- Quien escribe una transacción razona como si fuera la única.
- Cuesta: unas transacciones esperan a otras, o se abortan y se repiten.
- Por eso hay niveles más débiles. PostgreSQL usa read committed por omisión.

Trade-off

Más aislamiento: menos anomalías que prever, más esperas y abortos.

Read committed

Garantiza

- No se leen datos sin confirmar.
- No se sobrescriben: quien quiere la misma fila, espera.

No garantiza

- Que dos lecturas de la misma transacción vean lo mismo.

test/isolation.test.ts (reducido)

```
1  await a.run('BEGIN');
2  await a.run(placesLeft);   // A lee 10
3  await b.run('UPDATE tours SET
       places_left = 9 ...');
4  await a.run(placesLeft);   // A lee 9
```

Un detalle que usaremos

Un `UPDATE` que esperó a otra transacción vuelve a evaluar su `WHERE` sobre la fila ya confirmada.

Snapshot isolation: el repeatable read de PostgreSQL



- Cada transacción ve una **foto**: lo confirmado en un instante, y nada posterior.
- **MVCC**: un `UPDATE` escribe una versión nueva de la fila y deja la anterior.
- Leer nunca bloquea a quien escribe; escribir nunca bloquea a quien lee.

Los niveles en PostgreSQL 18

Nivel	Lectura sucia	No repetible	Fantasma	Serialización
Read committed (por omisión)	No	Posible	Posible	Posible
Repeatable read	No	No	No en PostgreSQL	Posible
Serializable	No	No	No	No

- PostgreSQL implementa tres niveles: su read uncommitted se comporta como read committed.
- El nombre no alcanza: otros motores usan los mismos nombres para garantías distintas.

PostgreSQL 18, *Transaction Isolation*, tabla 13.1

SECCIÓN 04

Las anomalías que quedan

04

Actualización perdida

Sesión A	Sesión B
`SELECT places_left ... → 10`	`SELECT places_left ... → 10`
`UPDATE tours SET places_left = 9 ...`	`UPDATE tours SET places_left = 9 ...` (espera)
`COMMIT;`	`COMMIT;`

Dos cupos vendidos, y el contador bajó uno

Cada sesión calculó $10 - 1 = 9$ en la aplicación. Dos escrituras sobre **la misma fila**, a partir de una lectura vieja.

Cuatro maneras de no perder la actualización

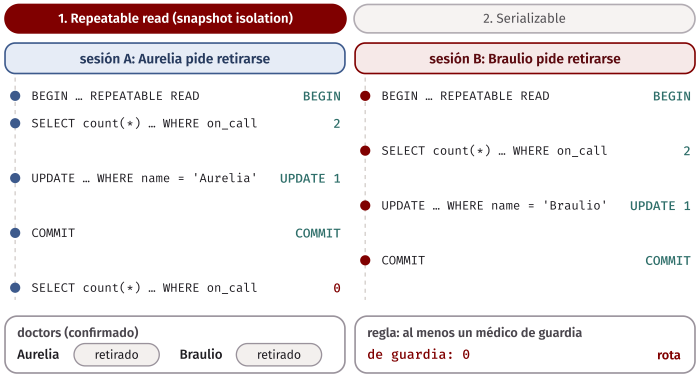
Manera	Cómo queda	Qué cuesta
El cálculo en la base de datos	`SET places_left = places_left - 1`	Nada, si cabe en SQL
Lock explícito	`SELECT ... FOR UPDATE`	Una espera y un lock
Columna de versión	`... WHERE id = 1 AND version = $2`	El reintento en el código
Repeatable read	La segunda aborta con `40001`	Repetir transacciones

Las cuatro están probadas

`test/isolation.test.ts`. En repeatable read, PostgreSQL detecta la actualización perdida: `could not serialize access due to concurrent update`.

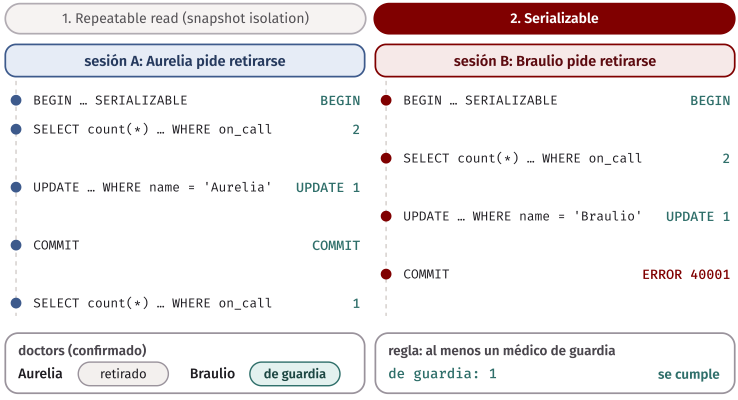
Write skew: los médicos de guardia

Regla: siempre queda al menos un médico de guardia. Aurelia y Braulio piden retirarse a la vez.



Nadie de guardia: cada una invalidó la premisa de la otra.

Las mismas sentencias, en serializable



Queda un médico de guardia: B tiene que repetir su transacción.

Snapshot isolation lo permite; serializable aborta una

make guion G=09 (el final)

```
A=# COMMIT;
COMMIT
B=# COMMIT;
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
B=# \errverbose
ERROR:  40001: could not serialize access due to read/write dependencies among transactions
```

- Snapshot isolation vigila las escrituras sobre una misma fila; no lo que se leyó para decidir.
- Serializable anota qué leyó cada una. Nadie esperó: el error llega en el `COMMIT`, y queda un médico.

Fantasmas: la decisión dependía de que una fila no existiera

```
make guion G=10

A=# SELECT id FROM room_reservations WHERE room = 'sala' AND day = '2027-07-28' FOR UPDATE;
 id
----
(0 rows)
B=# SELECT id FROM room_reservations WHERE room = 'sala' AND day = '2027-07-28' FOR UPDATE;
 id
----
(0 rows)
```

`FOR UPDATE` bloquea las filas que devuelve. No devolvió ninguna: las dos insertan su reserva.

- **Restricción de unicidad**: el segundo `INSERT` espera y falla con `23505`.
- **Crear las filas por adelantado**: es lo que hacen `room_nights` y los `seats` de Tinkuy.
- **Serializable**: una aborta con `40001`.

Qué nivel evita qué

Anomalía	Read committed	Repeatable read	Serializable
Dos lecturas ven datos distintos	Ocurre	No	No
Actualización perdida	Ocurre	Aborta una	Aborta una
Doble reserva (misma fila)	Ocurre	Aborta una	Aborta una
Write skew (los médicos)	Ocurre	Ocurre	Aborta una
Fantasma (la sala)	Ocurre	Ocurre	Aborta una

“Aborta una” no es “funciona”

La aplicación recibe un error `40001` y alguien tiene que escribir el reintento. Y todas las transacciones que tocan el invariante tienen que usar el nivel.

SECCIÓN 05

Tres maneras de no reservar dos veces

05

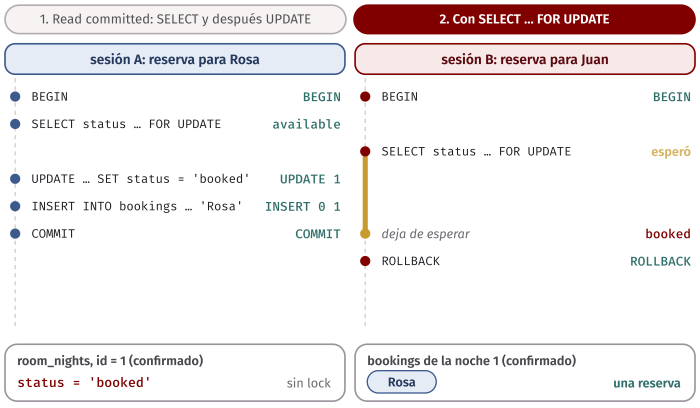
Solución 1: lock explícito

```
make guion G=02
```

```
A=# SELECT id, status FROM room_nights WHERE id = 1 FOR UPDATE;
...
B=# SELECT id, status FROM room_nights WHERE id = 1 FOR UPDATE;
    (la sesión B queda esperando: el lock lo tiene la sesión A)
...
A=# COMMIT;
COMMIT
    (la sesión B deja de esperar; su terminal muestra:)
 id | status
----+--------
  1 | booked
(1 row)
```

B ahora espera en la lectura, antes de decidir, y cuando continúa ve el estado que dejó A.

Con FOR UPDATE, paso a paso



Una noche, una reserva: B esperó antes de decidir.

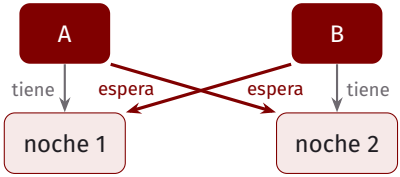
Qué detiene un `FOR UPDATE`

Otra sesión intenta, sobre la misma fila	Resultado
`SELECT` normal	No espera: lee la versión confirmada
`SELECT ... FOR UPDATE` o `FOR SHARE`	Espera hasta el `COMMIT` o `ROLLBACK`
`UPDATE` o `DELETE`	Espera igual
`... FOR UPDATE NOWAIT`	Falla enseguida (`55P03`)
`... FOR UPDATE SKIP LOCKED`	Se salta la fila
Cualquier cosa sobre **otra** fila	No espera

Trade-off

Fácil de razonar, y quien pierde no cobra. Pero quien pierde espera con una conexión ocupada, y hay que pedir el lock en todos los caminos que leen para decidir.

Deadlock: las noches 1 y 2, en orden inverso



- A tiene la noche 1 y espera la 2.
- B tiene la 2 y espera la 1.
- Ninguna va a soltar lo que tiene: solas no saldrían nunca.

```
make guion G=03

ERROR:  deadlock detected
DETAIL:  Process 4249 waits for ShareLock on transaction 41688; blocked by process 4247.
Process 4247 waits for ShareLock on transaction 41689; blocked by process 4249.
...
B=# \errverbose
ERROR:  40P01: deadlock detected
```

PostgreSQL revisa después de `deadlock_timeout` (1 s) y aborta a una. No se puede predecir a cuál.

La defensa: todos toman los locks en el mismo orden

booking/with-lock.ts

```ts
await client.query('BEGIN');
// ORDER BY id: todas las transacciones toman los locks en el mismo
// orden, pidan las noches en el orden que las pidan. Sin eso, dos
// reservas con noches en común pueden esperarse una a la otra (deadlock).
const nights = await client.query<NightRow>(
  'SELECT id, status FROM room_nights WHERE id = ANY($1::int[]) ORDER BY id FOR UPDATE',
  [request.nightIds],
);
// ... si alguna no está libre: ROLLBACK y "no disponible" ...
await charge(request); // con los locks tomados: nadie más avanza sobre estas noches
await client.query(`UPDATE room_nights SET status = 'booked' WHERE id = ANY($1::int[])`, [request.
    nightIds]);
const bookingIds = await insertBookings(client, request);
await client.query('COMMIT');
```

- `[1, 2]` y `[2, 1]` empiezan por la noche 1: la segunda espera sin tener nada tomado.
- Un `UPDATE` también toma locks de fila: el orden importa en las tres soluciones.

Solución 2: UPDATE con condición

comprobar y actuar, en una sola sentencia

```
1  UPDATE room_nights SET status = 'booked' WHERE id = $1 AND status = 'available';
```

make guion G=05

```
A=# UPDATE room_nights SET status = 'booked' WHERE id = 1 AND status = 'available';
UPDATE 1
...
B=# UPDATE room_nights SET status = 'booked' WHERE id = 1 AND status = 'available';
    (la sesión B queda esperando: el lock lo tiene la sesión A)
A=# COMMIT;
    (la sesión B deja de esperar; su terminal muestra:)
UPDATE 0
```

- Una fila cambiada: la noche es nuestra. Cero: ya no estaba libre.
- Read committed vuelve a evaluar el `WHERE` después de esperar.

La otra mitad: una restricción `UNIQUE`

db/unique-booking.sql

```sql
ALTER TABLE bookings ADD CONSTRAINT bookings_one_per_night UNIQUE (room_night_id);
```

un programa descuidado inserta de todos modos

```
B=# INSERT INTO bookings (room_night_id, guest) VALUES (1, 'Juan');
ERROR:  duplicate key value violates unique constraint "bookings_one_per_night"
DETAIL:  Key (room_night_id)=(1) already exists.
```

- El invariante deja de vivir solo en el código.
- Sola, actúa tarde: la versión ingenua con `UNIQUE` deja una reserva y dos cobros.
- **Error común**: comprobar el invariante solo con un `if`. Con dos instancias del servicio, el `if` de una no ve a la otra.

Solución 3: serializable

make guion G=06: el guion 1 con otro BEGIN

```
A=# BEGIN ISOLATION LEVEL SERIALIZABLE;
BEGIN
B=# BEGIN ISOLATION LEVEL SERIALIZABLE;
BEGIN
...
B=# UPDATE room_nights SET status = 'booked' WHERE id = 1;
    (la sesión B queda esperando: el lock lo tiene la sesión A)
A=# COMMIT;
COMMIT
    (la sesión B deja de esperar; su terminal muestra:)
ERROR:  could not serialize access due to concurrent update
B=# \errverbose
ERROR:  40001: could not serialize access due to concurrent update
```

El error no dice “la noche no está”. Dice: repitan la transacción completa.

La segunda parte de la solución es código

booking/retry.ts

```ts
for (let attempt = 1; ; attempt += 1) {
  stats.attempts += 1;
  try {
    return await transaction();
  } catch (error) {
    const code = sqlState(error);
    if (code === SERIALIZATION_FAILURE) stats.serializationFailures += 1;
    else if (code === DEADLOCK_DETECTED) stats.deadlocks += 1;
    else throw error;
    if (attempt >= maxAttempts) throw new TooManyRetries(attempt);
    // Backoff exponencial con jitter: si todas reintentan a la vez, vuelven a chocar.
    const ceiling = Math.min(200, 5 * 2 ** attempt);
    await sleep(Math.random() * ceiling);
  }
}
```

Solo `40001` y `40P01`. Espera antes del intento k: al azar entre 0 y mín$(200,\ 5 \cdot 2^k)$ ms.

Lo que cuesta serializable

- Hay que reintentar, y **todas** las transacciones que tocan el invariante tienen que usar el nivel.
- Un efecto fuera de la base de datos se repite con el reintento: el error puede llegar en el `COMMIT`.

```
make test

✓ serializable con reintento: 60 huéspedes, 4 noches: exactamente 4 reservas, una por noche (39.83925ms)
i cobros: 7 para 4 reservas; errores 40001: 13
```

- Aborta de más: anota lo leído por tabla o por página de índice. Con 300 noches, dos reservas de noches **distintas** ya se abortan entre sí.

Trade-off: cubre también los médicos y la sala, sin pensar qué bloquear. Se paga con abortos y reintentos.

Optimista y pesimista

Pesimista

Supone que habrá conflicto: toma un lock, y quien llega después espera.

Nadie descarta trabajo. Todos pagan la espera, y aparecen los deadlocks.

Optimista

Supone que no lo habrá: deja avanzar y comprueba al final.

Serializable, o una columna de versión. Si hubo conflicto, alguien repite su trabajo.

- El `UPDATE` con condición está en medio: sin lectura previa ni trabajo que descartar.
- Depende de la **contención**. Con poca, el optimista casi nunca repite nada. Con mucha, los reintentos se suman a la carga.

Mucha contención: 200 pedidos simultáneos por 10 noches

make compare-burst (rondas 2 y 5 de cada estrategia)

ronda	estrategia	201	409	5xx	reservas	dobles	cobros	transacciones	40001	p50 ms	p95 ms	max ms
2	naive	26	174	0	26	10	26	200	0	46	55	64
5	naive	22	178	0	22	10	22	200	0	34	39	50
2	lock	10	190	0	10	0	10	200	0	46	53	54
5	lock	10	190	0	10	0	10	200	0	34	40	41
2	conditional	10	190	0	10	0	10	200	0	39	40	41
5	conditional	10	190	0	10	0	10	200	0	34	39	40
2	serializable	10	190	0	10	0	55	313	113	201	258	270
5	serializable	10	190	0	10	0	34	292	92	122	171	185

- **Ingenua**: 20 a 26 reservas para 10 noches. Rápida porque no espera a nadie.
- **Lock** y **condición**: 10 reservas, 10 cobros; sin diferencia que se distinga del ruido.
- **Serializable**: 266 a 316 transacciones para 200 pedidos, 27 a 55 cobros, y un p95 varias veces mayor.

Tres corridas de cinco rondas, en una laptop con Apple Silicon dentro de Docker. Los tiempos tienen ruido; los conteos se repiten.

Poca contención: 200 huéspedes, 3 000 noches, 15 segundos

Estrategia	Reservas dobles	Errores 40001	Aceptados	Cobros	p95
naive	24	0	2 522	2 522	23,87 ms
lock	0	0	2 501	2 501	22,33 ms
conditional	0	0	2 519	2 519	21,74 ms
serializable	0	30	2 508	2 517	24,28 ms

Una de tres corridas de `make compare-load` (k6). En las tres, el p95 quedó entre 21 y 34 ms, sin un orden estable.

Idea clave

Con poca contención, las tres soluciones cuestan lo mismo, y la ingenua sigue vendiendo doble. La corrección no es un trade-off: el trade-off está entre las tres, y solo aparece con contención.

SECCIÓN 06

La transacción como recurso

06

Una transacción abierta ocupa cosas

Servidor con `STRATEGY=lock`, pool de 20 conexiones, y un cobro que tarda 3 segundos:

```
make activity, a mitad de la ráfaga

        state        | espera | conexiones
---------------------+--------+------------
 active              | Lock   |         10
 idle in transaction | Client |         10
```

- 10 conexiones con una transacción abierta y su lock, esperando el cobro. Otras 10, esperando ese lock.
- Los otros 180 pedidos no tienen ni conexión. Todos tardaron 3 s, también los 190 que recibieron un 409.

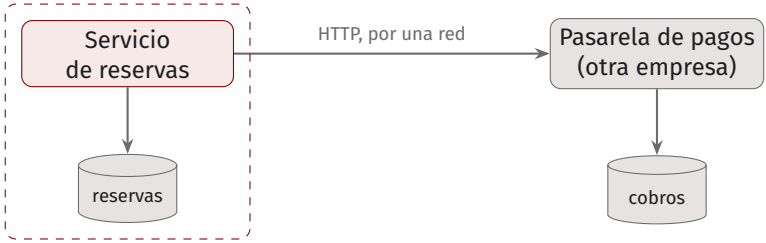
En Tinkuy: la pasarela se llama dentro de la transacción. Sacarla libera la base de datos y abre estados intermedios.

SECCIÓN 07

Cuando la transacción no alcanza

07

Dos sistemas, ninguna transacción



Orden	Si el proceso falla entre los dos pasos
Cobrar y después registrar	Un cobro sin reserva
Registrar y después cobrar	Una reserva sin pago

Un timeout no dice qué pasó

Una llamada entre dos nodos puede fallar de tres formas:

1. La conexión inicial falla. No pasó nada.
2. Falla a la mitad, mientras el servidor hace la operación. Quién sabe.
3. La operación se completa y la conexión se corta antes de la respuesta. Pasó todo.

Desde el cliente, las tres son un timeout

No se puede saber. Solo se puede diseñar para que **repetir sea seguro**.

Brandur Leach, *Designing robust and predictable APIs with idempotency* (Stripe, 2017)

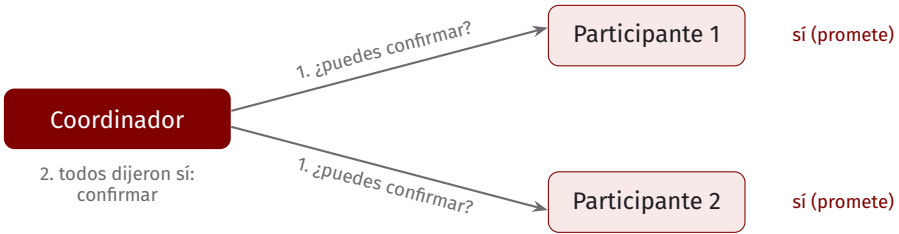
SECCIÓN 08

Two-phase commit

08

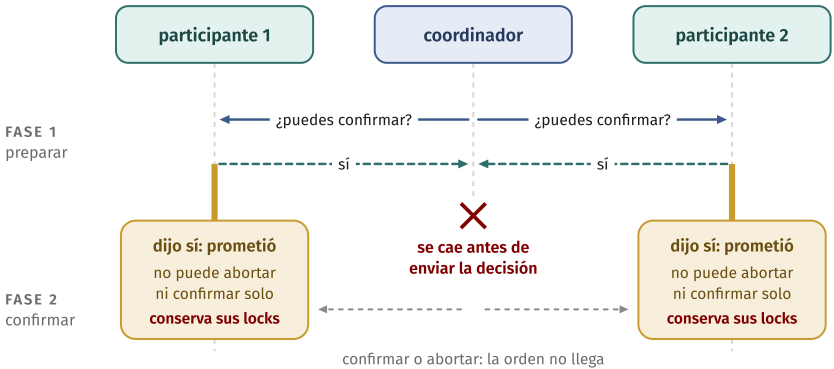


Two-phase commit



- **Preparar**: cada participante hace el trabajo, lo deja durable y responde sí o no.
- **Confirmar**: si todos dijeron sí, el coordinador ordena confirmar; si no, abortar.
- Da atomicidad entre varios nodos: o todos confirman o ninguno.

Si el coordinador se cae



Las filas de los dos participantes quedan bloqueadas hasta que el coordinador vuelva.

El precio de la promesa

Un participante que dijo sí ya no decide solo:

- no puede abortar (prometió confirmar);
- no puede confirmar (quizá otro dijo no);
- espera la orden con sus locks tomados.

Si el coordinador cae en ese momento, espera hasta que vuelva.

Entre servicios, casi nunca

Todos deben hablar el protocolo: una pasarela externa no lo hace.

La disponibilidad se multiplica: $A = A_1 \cdot A_2 \cdots A_k$.

Los locks cruzan servicios.

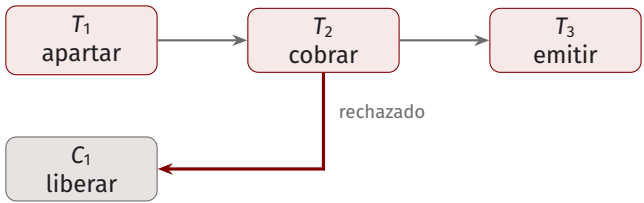
Se usa **dentro** de un sistema: una base de datos distribuida, entre sus shards.

SECCIÓN 09

Sagas

09

Saga: transacciones locales y compensaciones



- Cada paso **confirma de verdad** en su servicio. Nada queda preparado ni bloqueado.
- Si un paso falla, no hay `ROLLBACK`: se ejecutan **compensaciones**, en orden inverso.
- La compensación deshace desde el punto de vista del negocio, no borra lo ocurrido.
- No hay aislamiento: los estados intermedios se ven. Por eso se modelan: `held`, `pending`, `cancelled`.

Garcia-Molina y Salem, *Sagas* (1987); Richardson, *Saga*

El orden de los pasos

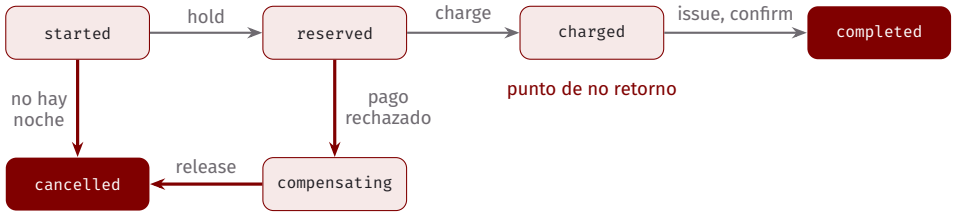
Tipo de paso	Qué es	Ejemplo	Dónde va
Compensable	Se puede deshacer con otra transacción	Apartar una noche	Al principio
Punto de no retorno	Si se completa, la saga ya no vuelve atrás	El cobro aprobado	En medio
Reintentable	Solo falla por razones técnicas; se repite	Emitir, enviar el correo	Al final

Antes del punto de no retorno, una falla se resuelve hacia atrás; después, solo hacia adelante.

Error común

Una compensación que puede fallar por una regla del negocio: la saga no puede ni avanzar ni retroceder.

La saga del hospedaje, orquestada



- El orquestador guarda el estado de cada saga en su tabla, `sagas`.
- Puede morir entre “el participante hizo el paso” y “lo anoté”. Al volver, repite el paso.
- Por eso cada operación de los participantes tiene que poder repetirse sin daño.

El orquestador muere justo después de cobrar

```
make saga

Después de la caída del orquestador:
  saga         estado      noche      apartada por  cobros  comprobantes
  saga-ines    reserved    held       saga-ines          1             0
  saga-juan    cancelled   available  -                  0             0
  saga-rosa    completed   booked     saga-rosa          1             1
Después de recover():
  saga         estado      noche      apartada por  cobros  comprobantes
  saga-ines    completed   booked     saga-ines          1             1
  saga-juan    cancelled   available  -                  0             0
  saga-rosa    completed   booked     saga-rosa          1             1
```

- La primera tabla es lo que una transacción nunca dejaría ver: `reserved`, y ya hay un cobro.
- Al retomar, el orquestador repite `charge`. No se cobra dos veces: el cobro lleva una clave de idempotencia.

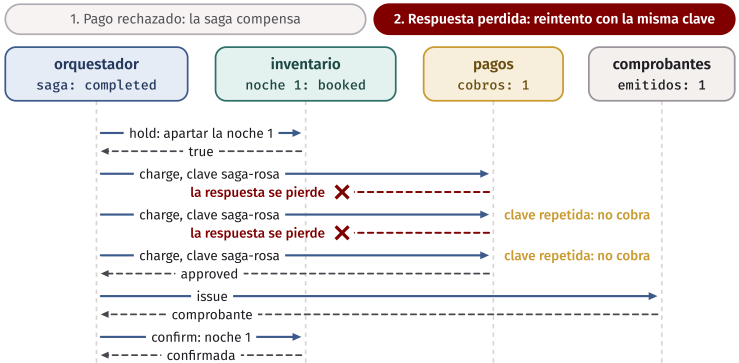
Fallas en cada paso

Falla inyectada	Qué hace la saga	Estado final
La noche no está libre	Termina	`cancelled`, sin cobro
El pago se rechaza	Ejecuta `release`	`cancelled`, noche libre
Se pierde el pedido o la respuesta de un paso	Reintenta el paso	`completed`, un cobro, un comprobante
El orquestador muere después de un paso	`recover()` lo repite	`completed`
`issue` sigue fallando después del cobro	No compensa	Queda en `charged` hasta que vuelva

Lo que se revisa

El estado final de los **tres** participantes, no solo el del orquestador (`test/saga.test.ts`, `saga/check.ts`).

Dos fallas, paso a paso



saga-rosa: completed, con un cobro y un comprobante.

Sin aislamiento: lo que se ve desde afuera

1. Rosa aparta la noche y empieza a pagar.
2. Juan llega, ve la noche `held` y su saga se cancela.
3. El pago de Rosa se rechaza. La compensación libera la noche.

Consistente, y nadie se quedó con la noche

Con una transacción y un lock, Juan habría esperado y la habría conseguido.

Se administra con decisiones de negocio: cuánto tiempo puede estar apartado un recurso, y qué se le dice a quien llega mientras tanto.

Coreografía y orquestación

Orquestación

Un coordinador guarda el estado y envía comandos.

El flujo y sus errores están en un solo lugar. El orquestador conoce a todos y tiene que estar arriba.

Coreografía

Cada servicio reacciona a los eventos del anterior.

Poco acoplamiento y sin punto central. El flujo no está escrito en ningún lado.

- Es la pareja mediator y broker de la semana 5.
- Con compensaciones, reintentos y un punto de no retorno, conviene poder leer el flujo en un archivo.

SECCIÓN 10

Idempotencia

10



Repetir sin daño

Idempotente

Aplicar la operación dos veces deja lo mismo que aplicarla una.

- `SET status = 'booked'` lo es. `SET places_left = places_left - 1`, un cobro o un correo, no.
- Se repite por un timeout, por un orquestador que retoma, por un `40001`.

La clave de idempotencia

Quien llama genera un identificador por operación (no por intento) y lo envía en cada reintento. Quien recibe guarda la clave con el resultado y, si se repite, devuelve el resultado guardado.

La clave, el efecto y la respuesta, en una transacción

idempotency/charges.ts (reducido)

```ts
await client.query('BEGIN');
// 1. Reclamar la clave. Si otra transacción está insertando la misma clave, este INSERT espera.
const claimed = await client.query(
  `INSERT INTO idempotency_keys (key, request_hash, status_code, response)
   VALUES ($1, $2, 0, '{}') ON CONFLICT (key) DO NOTHING`,
  [key, hash],
);
if (claimed.rowCount === 0) {
  // 2a. La clave ya existía: no se cobra. Se devuelve lo que se respondió la primera vez.
}
// 2b. La clave es nueva: el efecto, y su respuesta junto a la clave.
const reply = await insertCharge(client, input);
await client.query('UPDATE idempotency_keys SET status_code = $2, response = $3 WHERE key = $1', [
    /* ... */]);
await client.query('COMMIT');
```

O quedan las tres cosas o ninguna. En dos transacciones sería otra escritura doble.

La respuesta perdida, desde el cliente

```
make lost-response

Sin clave de idempotencia:
  intento 1: sin respuesta (TypeError, UND_ERR_SOCKET)
  intento 2: respuesta 201
  el cliente recibe: {"chargeId":2,"status":"approved","amountCents":18000}
  cobros registrados en el servicio: 2
...
Con clave de idempotencia:
  intento 1: sin respuesta (TypeError, UND_ERR_SOCKET)
  intento 2: respuesta 201 (guardada)
  el cliente recibe: {"chargeId":1,"status":"approved","amountCents":18000}
  cobros registrados en el servicio: 1
```

- Sin clave, el cliente recibe el cobro 2 y ni se entera de que existe el 1.
- Es un acuerdo entre dos partes: no sirve si el servidor ignora la clave, ni si el cliente la cambia en cada intento.

Duplicados concurrentes, y lo que hay que decidir

El duplicado puede llegar mientras el original todavía se procesa.

- `SELECT` de la clave y después cobrar: comprobar y después actuar, otra vez.
- `INSERT` sobre la clave primaria: el segundo espera y después lee la respuesta guardada.

Decisiones

- Quién genera la clave y con qué alcance.
- La misma clave con otro pedido: rechazar (422).
- Solo se guardan respuestas definitivas.
- Cuánto tiempo se conservan.
- Si el efecto sale a otro sistema, la clave viaja también.

SECCIÓN 11

La escritura doble y la outbox

11

La escritura doble

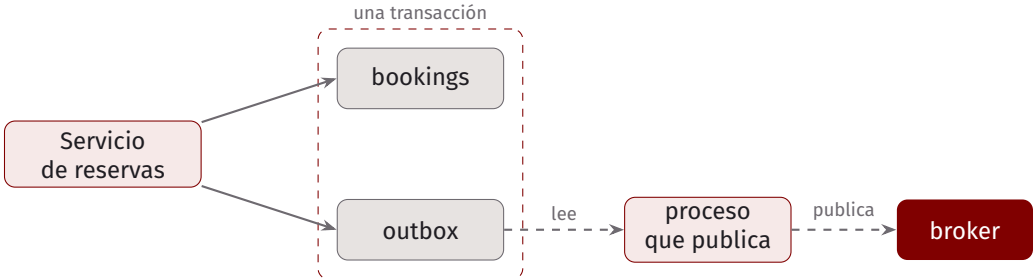
Guardar en la base de datos y publicar en el broker son dos operaciones. El proceso puede morir entre las dos.

Orden	Si muere en medio
Publicar y después confirmar	Se anunció una reserva que no existe
Confirmar y después publicar	La reserva existe y nadie se entera

Ningún orden lo arregla

Es el mismo hueco de “cobrar y registrar”. Las dos ramas están probadas en `test/outbox.test.ts`.

Outbox: escribir en un solo sistema



- El mensaje se inserta en `outbox` en la **misma transacción** que el cambio de estado.
- Otro proceso lee la tabla, publica y marca. Si muere antes de marcar, publica de nuevo.
- La entrega es at-least-once: los consumidores tienen que ser idempotentes.

Richardson, *Transactional outbox*

En Tinkuy, y cuándo no hacer nada de esto

P4 · Pagos como servicio

Una saga orquestada: reservar asientos (compensable), cobrar con clave de idempotencia (punto de no retorno), emitir y notificar (se reintentan).

Trade-off

Una saga cambia aislamiento por disponibilidad y autonomía. Si un invariante exige atomicidad, la primera opción es un mismo servicio y una misma base de datos.

Error común

Partir en dos servicios datos que cambian en la misma transacción, y después intentar recuperarla con una saga.

Laboratorio: no vender el mismo asiento dos veces, de tres maneras

En parejas. Pasos 1 a 7 en su copia de Tinkuy, con `code/week-12/` como referencia; el paso 8, en `code/week-12/lab/`.

1. `make oversell`: órdenes aceptadas y entradas emitidas para un asiento.
2. Reproducirlo con dos sesiones de `psql` y anotar el orden exacto de los pasos.
3. `SELECT ... FOR UPDATE`: ¿dónde espera la segunda sesión?
4. Deadlock con los asientos A y B en orden inverso; corregirlo ordenando los ids.
5. `UPDATE ... WHERE status = 'available'` y una restricción `UNIQUE`.
6. `SERIALIZABLE`: contar los errores `40001` y agregar el reintento.
7. Medir las tres con `make load-sale`: órdenes, ventas dobles, p95, errores.
8. `make lab-lost-response` y `make lab-test`: agregar la clave de idempotencia.

Para la próxima clase

- Sin entregas esta semana. **P4 · Pagos como servicio** se entrega en la semana 14.
- Terminar la tabla del paso 7 y decidir cuál solución elegirían para la apertura de una venta.
- Semana 13, mensajería. Leer de *Designing Data-Intensive Applications* (2.ª ed.): cap. 12 *Stream Processing* (Sesión A); cap. 3, *Event Sourcing and CQRS*, y cap. 10 *Consistency and Consensus* (Sesión B).
- Lecturas libres: Kreps, Narkhede y Rao, *Kafka: a Distributed Messaging System for Log Processing* (2011); Kleppmann, *Please stop calling databases CP or AP* (2015).

Pregunta

El proceso que lee la outbox publica un mensaje y muere antes de marcarlo. ¿Qué recibe el consumidor, y qué necesita para que no importe?

Lecturas de hoy: *Designing Data-Intensive Applications* (2.ª ed.), cap. 8; PostgreSQL 18, *Transaction Isolation* y *Explicit Locking*; *Software Architecture: The Hard Parts*, cap. 9 y 12; Richardson, *Saga* y *Transactional outbox*; Leach, *Designing robust and predictable APIs with idempotency*.