EcomLabs

GUÍA PASO A PASO

Cómo evitar la sobreventa en PostgreSQL

Las dos soluciones, SELECT … FOR UPDATE y un UPDATE condicional, en SQL y en Node, con un script que lanza compras en paralelo contra la última unidad y comprueba que solo una gana.

PARA	Tiendas con PostgreSQL y Node.js
ACTUALIZADA	30 de octubre de 2026
DOCUMENTACIÓN	postgresql.org/docs/current/explicit-locking.html
ARTÍCULO	ecomlabs.dev/g/stock
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué vas a montar
Paso 01	Crea las tablas
Paso 02	Las dos soluciones en SQL
Paso 03	Las mismas compras en Node
Paso 04	Lanza compras en paralelo
Al terminar	Checklist

ANTES DE EMPEZAR

Qué vas a montar

Dos formas de descontar inventario en PostgreSQL sin vender la misma unidad dos veces, en SQL y en Node, y un script que lanza N compras en paralelo contra un producto con stock 1 para comprobar que solo una gana. El script también ejecuta la versión ingenua (leer y luego escribir) para que veas la sobreventa con tus propios datos.

- Un PostgreSQL desechable. Por ejemplo, con Docker: `docker run --rm -d --name pg-race -e POSTGRES_PASSWORD=test -p 55432:5432 postgres:17-alpine`.
- Node 20 o más reciente y el driver `postgres` (Postgres.js): `npm install postgres`.
- `psql` para cargar el esquema y para la prueba manual del paso 2. Con el contenedor, sirve `docker exec -i pg-race psql -U postgres < schema.sql`.

> **CUIDADO**
>
> Corre todo contra una base desechable. El script vacía la tabla `orders` y cambia el stock del producto 1 en cada vuelta.

PASO 01

Crea las tablas

Guarda esto como `schema.sql`. El `CHECK (stock >= 0)` es una red de seguridad, no la solución: la versión ingenua del paso 3 escribe 0, nunca un número negativo, así que el CHECK no salta.

```
CREATE TABLE products (
  id    int  PRIMARY KEY,
  name  text NOT NULL,
  stock int  NOT NULL CHECK (stock >= 0)
);

CREATE TABLE orders (
  id         serial PRIMARY KEY,
  product_id int NOT NULL REFERENCES products (id),
  qty        int NOT NULL CHECK (qty > 0),
  created_at timestamptz NOT NULL DEFAULT now()
);

INSERT INTO products (id, name, stock) VALUES (1, 'Hoodie M', 1);
```

PASO 02

Las dos soluciones en SQL

Solución 1: bloquear la fila con FOR UPDATE

El `SELECT … FOR UPDATE` bloquea la fila hasta el `COMMIT` o el `ROLLBACK`. Otra transacción que intente bloquearla o actualizarla espera, y después recibe la fila ya actualizada.

```
BEGIN;
SELECT stock FROM products WHERE id = 1 FOR UPDATE;
-- tu código decide: si stock < 1, ROLLBACK
UPDATE products SET stock = stock - 1 WHERE id = 1;
INSERT INTO orders (product_id, qty) VALUES (1, 1) RETURNING id;
COMMIT;
```

Para verlo a mano, abre dos sesiones de `psql` y ejecuta en las dos `BEGIN;` y el `SELECT … FOR UPDATE`: la segunda se queda esperando hasta que hagas `COMMIT;` en la primera. Con `NOWAIT` al final, daría un error en lugar de esperar.

Solución 2: un UPDATE condicional

La condición `stock >= 1` decide y el `UPDATE` resta en la misma sentencia. En Read Committed, si otra transacción ya cambió la fila, PostgreSQL espera a que termine y vuelve a evaluar el `WHERE` sobre la versión nueva.

```
BEGIN;
UPDATE products SET stock = stock - 1
WHERE id = 1 AND stock >= 1
RETURNING stock;
-- 0 filas: no hay stock, ROLLBACK; 1 fila: crea el pedido
INSERT INTO orders (product_id, qty) VALUES (1, 1) RETURNING id;
COMMIT;
```

PASO 03

Las mismas compras en Node

Guarda esto como `stock.mjs`. Son tres funciones: la ingenua, para comparar, y las dos soluciones. Cada una recibe el cliente `sql` de Postgres.js y devuelve el id del pedido, o `null` si no hay stock. `workMs` simula el trabajo entre leer y escribir.

```
const sleep = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

export async function buyNaive(sql, productId, qty, workMs = 0) {
  return sql.begin(async (tx) => {
    const [p] = await tx`SELECT stock FROM products WHERE id = ${productId}`;
    if (!p || p.stock < qty) return null;
    await sleep(workMs);
    await tx`UPDATE products SET stock = ${p.stock - qty} WHERE id = ${productId}`;
```

```
    const [order] = await tx`INSERT INTO orders (product_id, qty) VALUES (${productId}, ${qty}) RETURNING id`;
    return order.id;
  });
}

export async function buyWithLock(sql, productId, qty, workMs = 0) {
  return sql.begin(async (tx) => {
    const [p] = await tx`SELECT stock FROM products WHERE id = ${productId} FOR UPDATE`;
    if (!p || p.stock < qty) return null;
    await sleep(workMs);
    await tx`UPDATE products SET stock = stock - ${qty} WHERE id = ${productId}`;
    const [order] = await tx`INSERT INTO orders (product_id, qty) VALUES (${productId}, ${qty}) RETURNING id`;
    return order.id;
  });
}

export async function buyConditional(sql, productId, qty) {
  return sql.begin(async (tx) => {
    const [p] = await tx`
      UPDATE products SET stock = stock - ${qty}
      WHERE id = ${productId} AND stock >= ${qty}
      RETURNING stock`;
    if (!p) return null;
    const [order] = await tx`INSERT INTO orders (product_id, qty) VALUES (${productId}, ${qty}) RETURNING id`;
    return order.id;
  });
}
```

PASO 04

Lanza compras en paralelo

Guarda esto como `race.mjs` junto a `stock.mjs`. Abre N conexiones, lanza N compras a la vez con cada versión y comprueba el resultado con `assert`:

```
import assert from "node:assert/strict";
import postgres from "postgres";
import { buyNaive, buyWithLock, buyConditional } from "./stock.mjs";

const N = Number(process.argv[2] ?? 20);
const sql = postgres(process.env.DATABASE_URL, { max: N });
await Promise.all(Array.from({ length: N }, () => sql`SELECT 1`));

async function race(name, buy, stock) {
  await sql`TRUNCATE orders RESTART IDENTITY`;
  await sql`UPDATE products SET stock = ${stock} WHERE id = 1`;
  const start = performance.now();
  const results = await Promise.allSettled(Array.from({ length: N }, () => buy(1)));
  const ms = Math.round(performance.now() - start);
  const [{ orders }] = await sql`SELECT count(*)::int AS orders FROM orders`;
```

```
  const [{ stock: left }] = await sql`SELECT stock FROM products WHERE id = 1`;
  const failed = results.filter((r) => r.status === "rejected").length;
  console.log(`${name.padEnd(16)} stock ${stock}, ${N} buyers -> ${orders} orders, stock left ${left},
${failed} errors, ${ms} ms`);
  return { orders, left, failed };
}

try {
  const naive = await race("read then write", (qty) => buyNaive(sql, 1, qty, 50), 1);
  assert.ok(naive.orders > 1, "the naive version should oversell");

  for (const [name, buy] of [
    ["FOR UPDATE", (qty) => buyWithLock(sql, 1, qty, 50)],
    ["UPDATE ... WHERE", (qty) => buyConditional(sql, 1, qty)],
  ]) {
    assert.deepEqual(await race(name, buy, 1), { orders: 1, left: 0, failed: 0 });
    assert.deepEqual(await race(name, buy, 5), { orders: 5, left: 0, failed: 0 });
  }
  console.log("OK: with both solutions, never more orders than units");
} finally {
  await sql.end();
}
```

Carga el esquema y córrelo con 20 compradores:

```
export DATABASE_URL=postgres://postgres:test@localhost:55432/postgres
psql "$DATABASE_URL" -f schema.sql
node race.mjs 20
```

El script exige que la versión ingenua cree más de 1 pedido (la sobreventa) y que `FOR UPDATE` y `UPDATE … WHERE` creen exactamente 1 pedido con stock 1 y 5 con stock 5, sin errores y con el stock en 0.

Si termina con `OK`, las dos soluciones aguantaron N compras simultáneas. Si falla en la versión ingenua, sube `workMs` o N: la carrera depende del tiempo y a veces hace falta un hueco mayor para verla.

AL TERMINAR

Checklist

- [ ] Ninguna compra lee el stock y lo escribe después sin bloqueo.
- [ ] El descuento es un `UPDATE … WHERE stock >= cantidad RETURNING`, o va después de un `SELECT … FOR UPDATE`.
- [ ] Una respuesta sin filas se trata como «sin stock».
- [ ] No hay llamadas lentas (la pasarela de pagos) dentro de la transacción.
- [ ] Los productos de un carrito se bloquean siempre en el mismo orden.
- [ ] `node race.mjs 20` termina en `OK` contra tu esquema.