EcomLabs

STEP-BY-STEP GUIDE

Prevent overselling in PostgreSQL

Both fixes, SELECT … FOR UPDATE and a conditional UPDATE, in SQL and in Node, with a script that fires purchases in parallel at the last unit and checks that only one wins.

FOR	Stores on PostgreSQL and Node.js
UPDATED	October 30, 2026
DOCS	postgresql.org/docs/current/explicit-locking.html
ARTICLE	ecomlabs.dev/g/stock-en
AUTHOR	Luis Morales · @luismoralesdev

Contents

Before you start	What you will build
Step 01	Create the tables
Step 02	Both fixes in SQL
Step 03	The same purchases in Node
Step 04	Fire purchases in parallel
Final	Checklist

BEFORE YOU START

What you will build

Two ways to decrement inventory in PostgreSQL without selling the same unit twice, in SQL and in Node, and a script that fires N purchases in parallel at a product with stock 1 to check that only one wins. The script also runs the naive version (read, then write) so you can watch the overselling on your own data.

- A throwaway PostgreSQL. With Docker, for example: `docker run --rm -d --name pg-race -e POSTGRES_PASSWORD=test -p 55432:5432 postgres:17-alpine`.
- Node 20 or newer and the `postgres` driver (Postgres.js): `npm install postgres`.
- `psql` to load the schema and for the manual test in step 2. With the container, `docker exec -i pg-race psql -U postgres < schema.sql` works too.

> **CAUTION**
> Run all of this against a throwaway database. The script empties the `orders` table and changes product 1's stock on every round.

STEP 01

Create the tables

Save this as `schema.sql`. The `CHECK (stock >= 0)` is a safety net, not the fix: the naive version in step 3 writes 0, never a negative number, so the CHECK never fires.

```
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);
```

STEP 02

Both fixes in SQL

Fix 1: lock the row with FOR UPDATE

`SELECT … FOR UPDATE` locks the row until `COMMIT` or `ROLLBACK`. Another transaction that tries to lock or update it waits, then gets the already updated row.

```
BEGIN;
SELECT stock FROM products WHERE id = 1 FOR UPDATE;
-- your code decides: if stock < 1, ROLLBACK
UPDATE products SET stock = stock - 1 WHERE id = 1;
INSERT INTO orders (product_id, qty) VALUES (1, 1) RETURNING id;
COMMIT;
```

To see it by hand, open two `psql` sessions and run `BEGIN;` and the `SELECT … FOR UPDATE` in both: the second one hangs until you `COMMIT;` the first. With `NOWAIT` at the end, it would raise an error instead of waiting.

Fix 2: a conditional UPDATE

The `stock >= 1` condition decides and the `UPDATE` subtracts in the same statement. Under Read Committed, if another transaction already changed the row, PostgreSQL waits for it to finish and re-evaluates the `WHERE` against the new version.

```
BEGIN;
UPDATE products SET stock = stock - 1
WHERE id = 1 AND stock >= 1
RETURNING stock;
-- 0 rows: out of stock, ROLLBACK; 1 row: create the order
INSERT INTO orders (product_id, qty) VALUES (1, 1) RETURNING id;
COMMIT;
```

STEP 03

The same purchases in Node

Save this as `stock.mjs`. It holds three functions: the naive one, for comparison, and both fixes. Each takes the Postgres.js `sql` client and returns the order id, or `null` when there is no stock. `workMs` simulates the work between reading and writing.

```
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;
  });
}
```

STEP 04

Fire purchases in parallel

Save this as `race.mjs` next to `stock.mjs`. It opens N connections, fires N purchases at once with each version and checks the result with `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();
}
```

Load the schema and run it with 20 buyers:

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

The script requires the naive version to create more than 1 order (the overselling), and `FOR UPDATE` and `UPDATE … WHERE` to create exactly 1 order with stock 1 and 5 with stock 5, with no errors and the stock at 0.

If it ends with `OK`, both fixes held up under N simultaneous purchases. If it fails on the naive version, raise `workMs` or N: the race depends on timing, and sometimes it needs a wider gap to show.

FINAL

Checklist

- [ ] No purchase reads the stock and writes it later without a lock.
- [ ] The decrement is an `UPDATE … WHERE stock >= quantity RETURNING`, or comes after a `SELECT … FOR UPDATE`.
- [ ] A result with no rows is handled as "out of stock".
- [ ] No slow calls (the payment gateway) run inside the transaction.
- [ ] A cart's products are always locked in the same order.
- [ ] `node race.mjs 20` ends with `OK` against your schema.