EcomLabs

GUÍA PASO A PASO

Cola de trabajos en Postgres con SKIP LOCKED

La tabla jobs, la función para encolar dentro de la transacción del pedido y el worker en Node con FOR UPDATE SKIP LOCKED, reintentos con espera exponencial y recuperación de trabajos colgados. Todo probado, con la checklist para llevarlo a producción.

PARA	Apps en Node.js con PostgreSQL que mandan correos o atienden webhooks
PROBADO CON	Node 22.22.3 · PGlite 0.5.10 (PostgreSQL 18.6) · Postgres.js 3.4.9 · Windows 11
ACTUALIZADA	3 de noviembre de 2026
DOCUMENTACIÓN	postgresql.org/docs/current/sql-select.html
ARTÍCULO	ecomlabs.dev/g/colas
AUTOR	Luis Morales · @luismoralesdev

Índice

Antes de empezar	Qué vas a montar
Paso 01	La tabla de trabajos
Paso 02	Encolar en la transacción del pedido
Paso 03	El worker
Paso 04	Arranca el worker
Paso 05	Lo que probamos
Al terminar	Checklist

ANTES DE EMPEZAR

Qué vas a montar

Una tabla `jobs`, una función que encola en la transacción del pedido y un worker en Node que toma cada trabajo con `FOR UPDATE SKIP LOCKED`, lo reintenta y recupera los colgados. Lo probamos el 2 de noviembre de 2026 con Node 22.22.3, Postgres.js 3.4.9 (`npm install postgres`) y PGlite 0.5.10, que es PostgreSQL 18.6 en WebAssembly.

PASO 01

La tabla de trabajos

Guárdalo como `schema.sql` y ejecútalo en tu base (o conviértelo en una migración):

```
CREATE TABLE jobs (
  id           bigint      GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  type         text        NOT NULL,
  payload      jsonb       NOT NULL,
  status       text        NOT NULL DEFAULT 'pending'
               CHECK (status IN ('pending', 'running', 'done', 'failed')),
  attempts     int         NOT NULL DEFAULT 0,
  max_attempts int         NOT NULL DEFAULT 5,
  run_at       timestamptz NOT NULL DEFAULT now(),
  locked_at    timestamptz,
  last_error   text,
  created_at   timestamptz NOT NULL DEFAULT now(),
  updated_at   timestamptz NOT NULL DEFAULT now()
);

CREATE INDEX jobs_pending_run_at ON jobs (run_at) WHERE status = 'pending';
```

PostgreSQL solo usa el índice parcial si el `WHERE` de la consulta implica su condición: el worker pide `status = 'pending'` tal cual.

PASO 02

Encolar en la transacción del pedido

Guarda `enqueue.mjs`:

```
export async function enqueue(sql, type, payload) {
  const [job] = await sql`
    INSERT INTO jobs (type, payload)
    VALUES (${type}, ${sql.json(payload)})
    RETURNING id`;
  return job.id;
}
```

Y llámalo con el `tx` de la transacción que crea el pedido, para que los dos se confirmen juntos o ninguno:

```
async function placeOrder(email) {
  return sql.begin(async (tx) => {
    const [order] = await tx`INSERT INTO orders (email) VALUES (${email}) RETURNING id`;
    await enqueue(tx, "order-confirmation", { orderId: order.id, email });
    return order.id;
  });
}
```

Usa `sql.json(payload)`: con `JSON.stringify`, nuestra columna `jsonb` guardó un string.

PASO 03

El worker

Guarda `worker.mjs` completo:

```
export async function claimJob(sql) {
  const [job] = await sql`
    UPDATE jobs
    SET status = 'running', locked_at = now(), attempts = attempts + 1, updated_at = now()
    WHERE id = (
      SELECT id FROM jobs
      WHERE status = 'pending' AND run_at <= now()
      ORDER BY run_at
      LIMIT 1
      FOR UPDATE SKIP LOCKED
    )
    RETURNING *`;
  return job ?? null;
}

export async function runOne(sql, handlers, { backoffSeconds = 30 } = {}) {
  const job = await claimJob(sql);
  if (!job) return null;
  try {
    const handler = handlers[job.type];
    if (!handler) throw new Error(`no handler for job type ${job.type}`);
    await handler(job.payload, job);
    await sql`
      UPDATE jobs SET status = 'done', locked_at = NULL, updated_at = now()
      WHERE id = ${job.id} AND attempts = ${job.attempts}`;
    return job;
  } catch (err) {
    const status = job.attempts >= job.max_attempts ? "failed" : "pending";
    const delay = backoffSeconds * 2 ** (job.attempts - 1);
    await sql`
      UPDATE jobs SET status = ${status}, locked_at = NULL, last_error = ${String(err?.message ?? err)},
        run_at = now() + make_interval(secs => ${delay}), updated_at = now()
```

```
      WHERE id = ${job.id} AND attempts = ${job.attempts}`;
    return job;
  }
}

export async function recoverStuck(sql, minutes = 15) {
  return sql`
    UPDATE jobs
    SET status = CASE WHEN attempts >= max_attempts THEN 'failed' ELSE 'pending' END,
        locked_at = NULL, last_error = 'stuck in running, lock expired', updated_at = now()
    WHERE status = 'running' AND locked_at < now() - make_interval(mins => ${minutes})
    RETURNING id, status`;
}

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

export async function work(sql, handlers, { idleMs = 1000, signal } = {}) {
  while (!signal?.aborted) {
    if (!(await runOne(sql, handlers))) await sleep(idleMs);
  }
}
```

- `claimJob` toma un trabajo en una sola sentencia; `SKIP LOCKED` salta las filas que otra transacción ya bloqueó.
- `runOne` reintenta tras 30, 60, 120 y 240 s y deja el quinto fallo en `failed`. `AND attempts = ...` evita que un worker atrasado pise a otro.
- `recoverStuck` libera los trabajos en `running` de más de 15 minutos.

PASO 04

Arranca el worker

Guarda `start.mjs` y cambia el envío falso por el cliente de tu proveedor de correo:

```
import postgres from "postgres";
import { work, recoverStuck } from "./worker.mjs";

// Fake sender: replace it with your email provider's client.
async function sendOrderEmail({ orderId, email }) {
  await new Promise((resolve) => setTimeout(resolve, 300));
  console.log(`email sent to ${email} for order ${orderId}`);
}

const sql = postgres(process.env.DATABASE_URL, { max: 1 });
const handlers = { "order-confirmation": sendOrderEmail };

setInterval(() => recoverStuck(sql).catch(console.error), 60_000);
await work(sql, handlers);
```

Córrelo aparte de tu servidor web, con `DATABASE_URL` apuntando a la misma base: `node start.mjs`. Cada proceso es un worker con una conexión.

PASO 05

Lo que probamos

Con `node --test` contra PGlite, las 8 pruebas pasaron:

- 4 workers, 50 trabajos: 50 correos. Tomando en dos sentencias (`SELECT` y luego `UPDATE`): 200.
- Un handler que siempre falla: esperas de 30, 60, 120 y 240 s, y luego `failed`.
- Un trabajo colgado se recupera y el worker atrasado no lo pisa; sin la guarda de `attempts`, la prueba falla.
- Una transacción revertida no deja trabajo, y la consulta usa el índice parcial.

> **CUIDADO**
> PGlite ejecuta una consulta a la vez y nunca solapa dos transacciones. Las pruebas demuestran que cada trabajo se procesa una vez, no que `SKIP LOCKED` evite esperas: eso lo dice la documentación de PostgreSQL. Repite la prueba de concurrencia en un PostgreSQL de verdad, nunca en tu base de producción.

AL TERMINAR

Checklist

- [ ] El request guarda el trabajo en la misma transacción que el pedido y responde.
- [ ] La tabla tiene `CHECK` en `status` y el índice parcial sobre `pending`.
- [ ] El worker toma cada trabajo en una sola sentencia con `FOR UPDATE SKIP LOCKED`.
- [ ] El correo se manda fuera de cualquier transacción, y los fallos acaban en `failed`.
- [ ] `recoverStuck` corre cada minuto, con un límite mayor que el trabajo más lento.
- [ ] Las escrituras finales llevan `AND attempts = ...` y los handlers son idempotentes.
- [ ] Los trabajos `done` viejos se borran con un `DELETE` periódico.