Engineer

# Selling the last ticket exactly once

October 13, 2026

A flash sale puts ten thousand units in front of a million people at once. The interesting failure is not that the shop goes down. It is that the shop stays up and sells ten thousand and forty. The instinct that follows — the database is too small, shard it — answers a question nobody asked. The database was never too big. Every buyer was writing to the same row.

We take the problem apart here before it reaches a project: where the ceiling comes from, the one statement that removes most of it, and what to build when a single database really is not enough.

Apply the engineering to your system

The engineering behind “Selling the last ticket exactly once” can inform your next change. We can investigate similar behaviour in your software, data or infrastructure. A structure that matches how the business actually works, so the questions you need answered later are still possible to ask.

**Explore our services** Get in touch →

Follow the reasoning and its limits

Read the reasoning and its limits in this note. We can test whether the same diagnosis holds in your environment. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

**Designed and ran 6 ETL/ELT pipelines** Get in touch →

## One row, every buyer

Twenty million rows spread over a cluster is a solved problem. One row that every request has to change is not, and adding machines does not touch it, because the row lives on one of them. Distributed stores say so plainly: DynamoDB gives a single partition about 1,000 write units a second, and every write to one item lands on that item’s partition however many others there are.

There is a theorem behind the shape. Bailis and colleagues call an invariant invariant-confluent when replicas can accept writes on their own and still merge into a valid state. `stock >= 0` is the textbook case of one that is not: two replicas that each see one unit left will each sell it, and no merge can take the second sale back. Never overselling needs coordination. The only question is how much of it each sale pays for.

## The number that decides it

The ceiling for one item is roughly how many sales one lock carries, divided by how long the lock is held:

```
max sales/s per item ≈ sales per lock ÷ lock hold time
```

| What holds the row | Ceiling per row |
|---|---|
| Lock, then call payment | ~2/s |
| Lock, then app logic | ~100–200/s |
| One conditional `UPDATE` | ~500–2,000/s |
| Batches of 100 per commit | ~50,000/s |
| One in-memory writer | ~100,000+/s |

These are orders of magnitude, not benchmarks; the hardware moves them. What it does not move is the shape. Most of a short lock is the commit waiting for its log to reach the disk, which is why batching wins by two orders: a hundred sales share one flush.

That leaves four levers, and every design below pulls one of them:

- **Hold the lock for less time** — one statement, nothing on the network inside it.
- **Do more per lock** — batch the decrements.
- **Have more locks** — split the stock across several keys.
- **Send fewer requests to the lock** — turn most buyers away before they reach it.

The last one matters more than it sounds. With a million buyers and ten thousand units, 990,000 requests have to fail. Rejections are most of the traffic, so “sold out” has to be the cheapest answer the system gives: a waiting room in front of the sale, a gate that admits about as many buyers as there are units, a `sold_out` flag the CDN serves without asking anyone, and a rate limit per account.

## Fix the one database first

The version that crawls, or oversells, usually looks like this:

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- the application checks the number,
-- and perhaps calls a payment provider
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

The lock is right. What it is held across is wrong: every millisecond the application spends between the `SELECT` and the `UPDATE` is a millisecond the next buyer waits. Without the `FOR UPDATE` it is worse. Two buyers read 1, both write 0, and one unit has been sold twice.

The fix makes the check and the decrement one statement:

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- one row back: reserved. none: sold out.
```

In PostgreSQL at READ COMMITTED, a second buyer who reaches the row while the first holds it waits, then evaluates the `WHERE` clause again against the row the first one committed. If that sale took the last unit, the condition fails and the second buyer gets no row back — sold out — rather than a negative number. At

REPEATABLE READ or SERIALIZABLE the second buyer gets a serialization failure instead, which is safe but turns contention into retries; READ COMMITTED is the level this statement is written for. MySQL's InnoDB reaches the same answer, because an `UPDATE` there reads the latest committed row at any level. Add the constraint as well, so a bug somewhere else fails loudly instead of selling stock that does not exist:

```
ALTER TABLE inventory
  ADD CONSTRAINT available_not_negative
  CHECK (available >= 0);
```

Then the rest of the single-database work, roughly in order of payoff:

- **Nothing external inside the transaction.** No payment call, no HTTP request, no message published. Those go before the transaction or after it.
- **Touch the hot row last.** Insert the order and everything else first, so the lock on the contended row lives only for the final statement and the commit.
- **No optimistic locking on a hot counter.** A version column turns contention into retries, and under a sale the retries are the load.
- **Serve the reads from somewhere else.** The product page's "12 left" can come from a replica or a cache a second behind. Only the sale needs the primary.
- **Give inventory a database of its own,** so a sale never waits behind the catalogue for the same disk.
- **Pool the connections.** A hundred thousand buyers should be a few dozen connections through PgBouncer, not thousands of backends queueing for one lock.

Seats are a different shape, because every unit is distinct, and they want a different statement. Each seat is a row, and a buyer skips the rows another transaction holds instead of queueing behind them:

```
UPDATE seats
   SET held_by = :order,
       held_until = now() + interval '10 min'
 WHERE id = (
   SELECT id FROM seats
    WHERE event_id = :event
      AND sold_at IS NULL
      AND (held_until IS NULL
           OR now() > held_until)
```

```
    LIMIT 1
    FOR UPDATE SKIP LOCKED)
RETURNING id;
```

An expired hold is simply available again, so no sweeper is needed for the answer to be right. And `now()` is the database's clock, so application servers whose clocks disagree cannot disagree about when a hold ran out.

For many shops this section is the whole fix. Do it before anything below, because sharding is a one-way door: easy to walk through, expensive to walk back.

## Claim first, pay second

A buyer who takes the last unit and then abandons the card form has not bought it. So a sale is not one write but a short lifecycle:

```
AVAILABLE → HELD, for minutes → SOLD
                              → RELEASED
```

- **The hold is keyed by the request, not the attempt.** Every checkout carries an idempotency key, and the holds table carries `UNIQUE (tenant_id, idempotency_key)`, so a retried click gets back the hold it already has instead of a second unit. One PostgreSQL detail bites here: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` returns nothing when the row already existed, so "no row back" means "look it up", not "failed".
- **Every transition is conditional.** Confirming a sale moves a hold from `held` to `sold` only where it is still `held`, and a release credits the stock back only if its own update changed exactly one row. A release that runs twice — the timeout and the cancel button, say — gives the unit back once.
- **Holds expire in minutes.** Five to ten is usual. Extend the hold once when payment starts, so a slow card does not lose the unit it is paying for.
- **Cap the units per buyer at the hold,** so one account cannot hold the whole stock while it decides.

Underneath all of it sits one line, the most useful in the design:

```
available + held + sold = total
```

It holds after every transaction, one query checks it, and when it stops holding, something has oversold or leaked a unit — before a customer says so.

## When one database is not enough

Sometimes it genuinely is not: many tenants, many sales at once, several regions. The structure that scales keeps one rule — every item has exactly one owner of its stock, an inventory service, and nothing else decides — and shards that service by `(tenant_id, item_id)`, so one item's writes never span two shards. The largest tenants get cells of their own, so their sale cannot starve anyone else's. Only the reservation has to be synchronous; everything after it can be a message.

For the hot item inside a shard there are four tools, and each pulls one of the levers above.

**A gate in front.** A Redis counter, decremented by a Lua script so the check and the decrement are one step, admits about as many buyers as there are units and turns everyone else away in microseconds. It is not the arbiter. Redis replicates asynchronously, so after a failover the new primary may have forgotten some admissions and let a few too many through. The database's conditional `UPDATE` still decides, and a reconciler resets the gate from the database's number.

**Split the stock.** Ten thousand units can be a hundred buckets of a hundred, each its own row or its own shard, with each buyer sent to one: a hundred locks instead of one. The idea is old — O'Neil's escrow transactions in 1986, then Barbará and Garcia-Molina's demarcation protocol, then the bounded counters of Balegas and colleagues — and its one real cost is the tail. A buyer's bucket runs dry while others still have stock, so either try another bucket or move the leftovers, and near the end of a sale collapse back to one. The same idea gives each region a quota of its own, so most sales never cross an ocean to ask.

**One writer per item.** Put every request for an item on one Kafka partition, read by one consumer that keeps the count in memory and commits its decisions in batches. One thread and no locks: the LMAX shape. Requests carry an id, and the writer drops one it has already decided, because a queue delivers twice.

**Units as rows.** When units are few and distinct — seats, numbered editions — create one row each in advance and claim them with `SKIP LOCKED` as above, or keep the unit ids in a Redis set and `SPOP` one. There is no counter left to fight over.

## The rest of checkout

Payment, the order and the receipt are separate systems, and no transaction spans them. Two-phase commit is the textbook answer and the wrong one here: it blocks while any participant is slow, it adds round trips to the hottest path, and payment providers do not take part in it. A saga does the job, a sequence of local steps, each with a step that undoes it:

```
1. reserve stock       undo: release the hold
2. pending order       undo: cancel the order
3. extend the hold     (expires on its own)
4. authorise payment   undo: void it
5. confirm HELD → SOLD
6. capture payment
```

Authorising first and capturing last means a payment that arrives after its hold has expired is voided rather than refunded: the money never moved. Each step writes its message to an outbox table in the same transaction as the change it announces, and change data capture publishes it, so nothing is lost between saved and sent. Every consumer is idempotent, keyed by the order. Messages arrive at least once, and idempotency makes that effectively once.

A distributed lock does not belong anywhere in this. Kleppmann's point is that a lock holder can pause past its lease — a garbage collection, a stalled virtual machine — and keep writing, and the cure is a fencing token the store checks. But a store that can check a token can check `available >= :qty`, and then it never needed the lock.

## Pause the item, never oversell it

Consistency and availability are chosen per question, not per system. “Can I sell this unit?” has to be consistent: when the owner of an item’s stock cannot be reached, that item stops selling. Everything else — the catalogue, the “3 left” badge, the order history — can stay available and be a little stale.

The quietest oversell comes from a failover. With asynchronous replication a primary acknowledges a sale and dies before its replica has it; the replica is promoted, the sale is gone, and the unit is sold again. The fix is to make a sale durable on more than one machine before acknowledging it: a synchronous standby in PostgreSQL ( `synchronous_commit` with `synchronous_standby_names` ), or a store built on Raft. That costs latency on every commit, which lengthens exactly the lock hold time the ceiling divides by, and batching and the gate buy it back. Scope the pause, too: a failed shard should stop the items on it, not the shop.

## Proving it

None of this should be taken on trust, ours included.

- **Watch the invariant.** Run `available + held + sold = total` on a schedule, chart it, and page on any drift. Zero oversells is a service level objective with an alert, not a hope.
- **Race it on purpose.** Ten thousand concurrent buyers for a hundred units, in the test suite, asserting exactly a hundred sold and no number below zero.
- **Break things during the race.** Kill the primary, cut the network, pause a process — Jepsen-style — and check the invariant afterwards.
- **Load-test the rejections.** The sold-out path has to stay fast when it is 99% of the traffic, and the edge has to absorb it.
- **Put the metrics on one dashboard:** lock waits, expired holds, and the distance between the gate’s count and the database’s.

## In what order

- Measure first. `pg_stat_statements` and the lock waits say whether the hot row is the problem at all.

- Make the write one conditional statement.
- Shed load: a waiting room, a gate, a cached “sold out”.
- Give inventory its own service and its own owner.
- Shard by `(tenant_id, item_id)`, and only if the first four were not enough.
- Add a hot-item mode, buckets or a single writer, for the sale you know is coming.
- Regions last, with quotas.

## What we would tell the next person

- **Sharding answers a different question.** It spreads many rows, and an oversold sale is one row. Find out which one you have before you build.
- **Make the check and the decrement one statement.** Most of the distance between fine and oversold is the time between reading a number and writing it.
- **Most of the traffic is the answer no.** Make rejection the cheapest thing the system does.
- **Pause before you oversell.** An item that stops selling for thirty seconds is an apology. An item sold twice is a refund, a support queue and a customer who remembers.
- **Write the invariant down and watch it.** It is one line, and it is the only thing that says the design works in production rather than in review.

## Sources


- PostgreSQL: Read Committed isolation
- PostgreSQL: `FOR UPDATE` and `SKIP LOCKED`
- PostgreSQL: synchronous replication
- Bailis et al., Coordination Avoidance in Database Systems
- Balegas et al., Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants
- Martin Kleppmann: How to do distributed locking
- Martin Fowler: The LMAX Architecture
- DynamoDB: designing partition keys
- DynamoDB: partitions and data distribution
- microservices.io: Saga
- microservices.io: Transactional outbox
- Redis: replication
- Jepsen

All blog posts

Measure before you change

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

**See what it costs** Get in touch →

# Discuss your observations

Discuss “Selling the last ticket exactly once” in the context of your own project. Tell us what you observed and what a useful result would look like.

**Explore our services**

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

**Carefully Crafted With Love by D.**

**CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N**

**Copyright © 2025 – present · Engineer ApS · All rights reserved.**

**Printed from https://engineer.company/notes/selling-limited-stock-without-overselling/**