Engineer

Optimise the database before you shard it

October 14, 2026

When a database slows down under load, the first proposal is usually more machines. Sometimes that is right. More often the slow part is a handful of statements nobody has read yet, and spreading the data across servers spreads those statements over more hardware without making one of them cheaper. A problem that looks unsolvable is often one query plan away from a neat answer, and horizontal scaling is sometimes chosen precisely so that nobody has to read the database you already have.

This is the order we work in before we split anything: look, tighten, take the reads off the primary, partition the tables around the hot spot, and shard only when one machine is full of good work. The flash sale, where every buyer writes to one row, is worked through in Selling the last ticket exactly once. This note is the method around it, and its last section takes on the opposite proposal: sell from every server, and apologise for the units sold twice.

Apply the engineering to your system

The engineering behind “Optimise the database before you shard it” 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 →

Look first

PostgreSQL already records most of what there is to know. Before changing anything, ask it four questions.

Which statements cost the most in total? `pg_stat_statements`, an extension loaded through `shared_preload_libraries`, keeps every normalised statement with its call count and its time. Sort by total, not by mean: a 2 ms query run a million times an hour costs more than a two-second report run twice.

```
SELECT calls,
       round(total_exec_time) AS total_ms,
       round(mean_exec_time::numeric, 1)
         AS mean_ms,
       left(query, 60) AS query
  FROM pg_stat_statements
 ORDER BY total_exec_time DESC
 LIMIT 10;
```

What does the top one actually do? `EXPLAIN (ANALYZE, BUFFERS)` runs the statement and shows the plan it chose, the rows each step really returned against the rows it expected, and how many pages it read. A scan of a million rows to return ten, or an estimate off by a factor of a hundred, is usually the whole story. `ANALYZE` executes what it explains, so wrap a write in `BEGIN` and `ROLLBACK`.

Who is waiting, and on whom?

```
SELECT pid, wait_event_type, wait_event,
       pg_blocking_pids(pid) AS blocked_by,
       left(query, 60) AS query
  FROM pg_stat_activity
 WHERE wait_event_type = 'Lock';
```

Rows here in a busy minute mean transactions are queueing for each other, not for hardware, and a bigger machine does not remove the serialization of updates to one shared row. More application servers can add writers to that queue.

Is the server working or juggling? Count the active sessions in `pg_stat_activity` against the cores. When many times more are active than there are cores, the server spends its time switching between them, and a connection pool fixes that before any hardware does. Then read `n_dead_tup` in `pg_stat_user_tables` : a table with more dead rows than live ones is being scanned through its own garbage, which is autovacuum falling behind, not a database that is too small.

Write down what you find. It is the baseline every later step is measured against, and the honest answer to “do we need to shard?” is often in it already.

Tighten what is there

Roughly in order of payoff for the effort.

Indexes that match the questions. The plans above name the scans. An index on the columns a hot query filters by turns a scan into a lookup, and a partial index holds only the rows a query can want — units still for sale, say — so it shrinks as a sale goes on instead of growing:

```
CREATE INDEX units_unsold
    ON units (item_id)
 WHERE NOT sold;
```

Every index is also paid for on every write, so drop the ones whose `idx_scan` in `pg_stat_user_indexes` stays at zero.

Types that say what the value is. Schemas written fast, by a person or a model, reach for `TEXT` where a `SMALLINT` or a `BOOLEAN` would do. A `BOOLEAN` is one byte, a `SMALLINT` two, and `'yes'` stored as text is four. PostgreSQL also pads each fixed-width column to its own alignment, so a column order chosen by accident costs bytes too. One unit of stock, written fast:

```
CREATE TABLE units (
  sold        text,  -- 'yes' or 'no'
  id          bigint,
  reserved    text,  -- 'yes' or 'no'
  reserved_at timestamptz,
```

```
  item_id     bigint,
  order_id    bigint
);
```

And the same unit with types that say what the values are, widest first:

```
CREATE TABLE units (
  id          bigint PRIMARY KEY,
  reserved_at timestamptz,
  order_id    bigint,
  item_id     integer NOT NULL,
  sold        boolean NOT NULL DEFAULT false,
  reserved    boolean NOT NULL DEFAULT false
);
```

On PostgreSQL 18 the first row measures 72 bytes and the second 54, a quarter less, so more of the table fits in memory. The types also refuse values such as `'maybe'` that the application would otherwise have to defend against, and a `CHECK` constraint does the same for ranges: `quantity smallint CHECK (quantity > 0)`.

Fewer round trips, and the work where the data is. Every statement the application sends costs a round trip and a parse. A loop that reads a hundred rows and updates them one by one is a hundred and one round trips where one set-based statement would do, and a page that fetches every unit of an item to count the free ones ships thousands of rows to compute one number. The rule we use: what has to see the data goes to the database — filtering, counting, a write that depends on what is there — and business rules stay in the application, because the database's CPU is the hardest one in the system to add more of.

Updates that stay on their page. A row that changes many times a minute leaves a new version of itself each time. When an update changes no indexed column and the page has room, PostgreSQL keeps the new version on the same page and touches no index at all: a heap-only tuple, or HOT update. So keep the columns a frequent update changes out of the indexes, and leave room on the pages:

```
ALTER TABLE units SET (fillfactor = 90);
```

The setting applies to pages written from then on, so set it before the rows are loaded. On the table above, 2,000 claims that set `reserved`, `reserved_at` and `order_id` were all HOT, and of 2,000 updates that set `sold`, which the partial index reads, none were HOT. Selling happens once per unit; claiming can happen many times. `n_tup_hot_upd` against `n_tup_upd` in `pg_stat_user_tables` shows the ratio on a live system.

One row says who was first

A unit goes to the first buyer whose statement reaches it, and the claim runs out on the database's clock. Two columns carry it: `reserved`, a boolean, and `reserved_at`, the moment it was set.

```
UPDATE units
   SET reserved = true,
       reserved_at = now(),
       order_id = :order
 WHERE id = :unit
   AND NOT sold
   AND (NOT reserved
        OR reserved_at
           < now() - interval '10 minutes')
RETURNING id;
```

One row back: this buyer has it. None: someone else was first, or it is sold. Two claims on the same unit queue on its row lock, and on READ COMMITTED, PostgreSQL's default, the second checks the `WHERE` clause again against the row the first one committed, so exactly one of them gets a row back. A claim older than ten minutes counts as free, so nothing has to sweep it up for the answer to be right.

That makes the boolean alone an incomplete answer. `reserved` stays true after a claim lapses, and only the two columns together say whether a unit is held, so every query that asks has to ask both. A periodic sweep that resets lapsed claims keeps the table honest for anyone reading `reserved` on its own, but correctness never waits for it. A constraint keeps the pair consistent:

```
ALTER TABLE units
  ADD CONSTRAINT reserved_has_time
  CHECK (NOT reserved
         OR reserved_at IS NOT NULL);
```

The ticket note stores the deadline, `held_until`, rather than the start; it is the same design. When the buyer picks the unit — a seat, a numbered print — the statement names it, as here. When any unit will do, that note's `SKIP LOCKED` form picks one nobody else is holding.

Reads off the primary

Most of a shop's traffic is reading: the product page, the "3 left", the basket, the order history. None of it needs the primary, the one machine that has to stay free for writes. A streaming replica takes those reads, and adding one is configuration where a shard is a project.

Two cautions come with it. A replica runs a little behind, so a buyer who has just claimed a unit and then reads their basket from it may not see the claim: read what a user has just written from the primary, or send that user to the primary for a few seconds after each write. And a long report on a replica can be cancelled when it holds up the changes the replica is replaying. The PostgreSQL documentation calls these query conflicts and lists the settings that trade replica lag against them.

The cheapest read is one that never reaches a database. When every unit of an item is claimed, nobody can buy it until a claim lapses or is released, and the database can say exactly when that is:

```
SELECT count(*) FILTER (
         WHERE NOT reserved
            OR reserved_at
               < now() - interval '10 minutes')
         AS free,
       min(reserved_at) + interval '10 minutes'
         AS until
  FROM units
 WHERE item_id = :item
   AND NOT sold;
```

When `free` is zero, `until` is the moment the earliest claim lapses: cache "fully reserved" with that as its expiry, never longer, and every visitor until then is turned away by the cache rather than the database. When `free` is zero and `until` is empty, every unit is sold and the answer is final. A claim released early — a declined card, an abandoned basket — deletes the cache entry, so the unit is back on sale at once.

The cache only ever says no. A yes always comes from the conditional `UPDATE` above, so a stale cache can cost a sale for a few seconds and can never sell a unit twice. It is the ticket note's cached "sold out", with a deadline.



Partition the tables around the hot row

Partitioning splits one table into several by a key — a month, a tenant — behind one name. It is worth being plain about what it does not do: it does not make one contended row any faster. That row still lives in one partition, on one machine, behind one lock.

What it does is keep the tables around the hot spot small. Orders, claim history and events grow for ever, and most queries want the recent part:

```
CREATE TABLE orders (
  id          bigint NOT NULL,
  created_at  timestamptz NOT NULL
              DEFAULT now(),
  item_id     integer NOT NULL,
  quantity    smallint NOT NULL
              CHECK (quantity > 0),
  paid        boolean NOT NULL DEFAULT false,
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE orders_2026_10
  PARTITION OF orders
  FOR VALUES FROM ('2026-10-01')
              TO ('2026-11-01');
```

A query that names a date range reads only the partitions inside it, and this month's indexes stay small enough to live in memory. Retention stops being a `DELETE` of millions of rows, with the bloat and the vacuuming that follow, and

becomes one statement:

```
ALTER TABLE orders
  DETACH PARTITION orders_2025_10;
```

The primary key has to include the partition key, which is why it is `(id, created_at)`. A query that does not name the key reads every partition, so partition by what the queries already filter on.

When sharding is reasonable

There is one limit sharding cannot move: every item has one owner. A unit cannot be claimed on two servers at once, so sharding moves whole items, never half of one. When a single item outgrows its machine, the ticket note's buckets split its stock into smaller pieces, and each piece again has exactly one owner.

Within that, we call sharding reasonable when one of these is true, and not before:

- **One machine is full of good work.** The statements are already lean — one conditional write each, nothing external inside a transaction, updates that stay on their page — and the primary's CPU or write-ahead log is still saturated, on the largest machine worth buying, across many items rather than one. One hot item is a bucket problem, not a shard problem.
- **The data has outgrown its maintenance.** A backup cannot be restored inside the time promised to the business, an index cannot be built or a table vacuumed in the quiet hours, or the working set no longer fits in memory.
- **Isolation is the requirement.** A tenant whose contract or regulator says their data lives apart, or whose sale must not share a disk with anyone else's.

The cost lands on everything else. Migrations, backups, restores and reports have to account for separate shards; a join across shards may move into application code; and a query that does not know its shard key may have to ask all of them. That is the bill the sections above exist to avoid, and once it is paid it is hard to get back.

When an apology is cheaper than a lock

There is a serious argument for going the other way. Let every server sell from the stock it can see, accept that now and then a unit goes to two buyers, and let customer support refund one of them with an apology. Business logic becomes eventually consistent, built for the 99.9% of requests that never collide, and an asynchronous loop finds the collisions afterwards and makes them good. In return the shop never has to say “try again”. Helland and Campbell make the strongest form of the case in *Building on Quicksand*: a system that acknowledges work before its backup has it makes every promise probabilistic, and businesses have always settled the promises they could not keep with apologies.

We agree with most of that. We disagree about where it is spent.

Most of a shop should be eventually consistent. The catalogue, the search index, the “3 left” badge, the basket, the recommendations, the emails, the warehouse export and the reports can all run seconds behind, and making them wait for the primary costs speed and buys nothing. Dynamo was built for exactly this: Amazon’s shopping cart never forgets or rejects an “add to cart”, even while machines fail, and the price is that a merge can bring back an item the buyer had removed. After the claim, the ticket note’s checkout is already asynchronous: a saga of local steps, messages delivered at least once, and consumers that are idempotent, keyed by the order. A reconciliation loop belongs in any design that talks to a payment provider, because webhooks get lost.

The claim is the one place it does not hold. “Rare” assumes collisions are rare, and in a flash sale the collisions are the sale. A server that sells from its own view keeps selling until it hears what the others sold, so the oversell is roughly the attempt rate times the replication lag. In a simple simulation, two regions split 2,000 attempts a second, each sells from its own count and hears of the other’s sales 200 milliseconds later. Across five runs of a 10,000-unit sale, each run sold between 191 and 205 units that did not exist. The arithmetic predicts 200: the attempt rate, times the lag, times the share of sales made in the other region. With one owner per item, the same model sold exactly the stock. That happens on every sell-out, whether anything failed or not, and it lands at the climax, when the most buyers are watching.

The uptime is smaller than it sounds. Gilbert and Lynch proved the trade that eventual consistency makes: while the network is partitioned, a system can answer or it can be consistent, not both. Abadi's PACELC adds the other half. When nothing is partitioned, the trade is latency against consistency, and one owner per item, with lean statements, is already fast. Neither design reaches 100%: the payment provider, DNS and the networks between them have outages of their own. What the strict design gives up is narrow. When the owner of one item's stock cannot be reached, that item stops selling for the seconds a failover takes, and the ticket note's section on pausing the item scopes the pause to the items on the failed machine. The rest of the shop keeps selling.

An apology is not a refund. The refund is the cheap part. The buyer was told yes, and may already have booked the train, taken the day off and told their friends. Hearing "sold out" a second too early is a disappointment. Hearing that a confirmed order has been withdrawn is a broken promise. Support is staffed for an ordinary day, and oversells arrive together, in the busiest minute of the year. Before planning to withdraw confirmed orders, ask a lawyer what a confirmation commits the shop to.

Keep the asynchronous part, and move it before the yes. What the proposal gets right is that the buyer does not have to wait on a lock. Accept every click into a durable, replicated log, one partition per item, and tell the buyer the answer is coming. Accepting a click then depends only on the log, and a log replicated across three machines keeps taking writes when one of them fails. One owner per item reads the log in order and decides, at memory speed, and the answer goes back as a message within seconds. When that owner fails over, the requests wait in the log instead of being refused or sold twice. The yes is a hold, and payment follows it as in the ticket note's checkout, so nobody pays for a unit they did not get. The reconciliation still happens, but before the promise instead of after it.

Share the stock, not the count. When the business needs regions to sell without asking each other, give each region a quota of the stock and let it sell from its own quota without asking anyone. A region whose quota runs dry asks for more, and near the end of the sale the quotas collapse back to one owner. Bailis and colleagues work out which rules servers can keep without talking to each other, and a stock that must not go below zero is not one of them. Quotas do not remove that conversation; they move it off the hot path, to the rare moment a quota is refilled. That is the eventually consistent design that keeps the invariant. Its one cost is the tail, where a buyer is turned away in one region while another region still has stock.

When an oversell is the right policy. Some goods can be made again: a print on demand, a T-shirt the supplier still holds, a licence key. Selling past the count is then a backorder with an honest date, and eventual consistency is the right design. Airlines overbook on purpose. They know their no-show rates, price the risk and compensate by a rule written in advance. That is the test. An oversell is acceptable when the business chose its size and its remedy before the sale. It is a defect when it falls on whoever happened to click last.

If you choose apologies, build them like payments. The loop the proposal describes is a refund process, and it needs what anything that moves money needs:

- **An id on every sale,** so a retry is the same sale, and an append-only ledger of sales. The stock number is derived from the ledger, never edited.
- **A rule for who keeps the unit** that you could explain to the buyer who lost. Order by the log, not by the clocks of different servers, where a few milliseconds of skew would decide who gets the refund.
- **Compensations that are idempotent and logged:** cancel, void or refund, tell the buyer. They are the saga's undo steps, run by a machine.
- **A deadline.** The loop has to find every oversell before anything ships, and before the buyer finds it.
- **A budget, measured.** Oversells per sale, against the number the business signed for, next to the invariant on the same dashboard, with an alert when the budget is spent.

Our answer is to be eventually consistent everywhere around the claim, strict at the claim, and asynchronous in how the answer reaches the buyer. Design for the common path, but in a flash sale the collision is the common path. If the business wants a few oversells in return for something, make it a number the business chose, with a remedy written before the sale, and measure it.



What we would tell the next person

- **Read before you buy.** `pg_stat_statements` and one `EXPLAIN` answer more capacity questions than a new server does.
- **Make the types say what the data is.** A boolean stored as a boolean is smaller, and it refuses nonsense.
- **Send the database what has to see the data, and nothing else.** Filtering and counting belong there. Business rules do not.

- **The cache says no; only the database says yes.** Then a stale cache costs seconds, never a unit.
- **Partitioning is housekeeping.** It keeps the tables around a hot row small and does nothing for the row.
- **Shard when one machine is full of good work,** not when it is full of waiting.
- **Reconcile before the yes, not after.** Everything around the claim can be eventually consistent. The claim itself, made that way, oversells by the attempt rate times the lag, on every sale.

Sources

- PostgreSQL: pg_stat_statements
- PostgreSQL: Using EXPLAIN
- PostgreSQL: the cumulative statistics system
- PostgreSQL: numeric types
- PostgreSQL: character types
- PostgreSQL: Boolean type
- PostgreSQL: partial indexes
- PostgreSQL: heap-only tuples (HOT)
- PostgreSQL: table partitioning
- PostgreSQL: hot standby and query conflicts
- Pat Helland and Dave Campbell, Building on Quicksand
- Seth Gilbert and Nancy Lynch, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services
- Daniel Abadi, Consistency Tradeoffs in Modern Distributed Database System Design
- DeCandia et al., Dynamo: Amazon's Highly Available Key-value Store
- Hector Garcia-Molina and Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems

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 “Optimise the database before you shard it” 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/optimise-before-you-shard/