Engineer

Erst optimieren, dann die Datenbank aufteilen

14. Oktober 2026

Wenn eine Datenbank unter Last langsamer wird, lautet der erste Vorschlag meist: mehr Maschinen. Manchmal ist das richtig. Häufiger sind einige wenige, noch ungelesene Anweisungen langsam. Verteilt man die Daten auf Server, verteilt man diese Anweisungen auf mehr Hardware, ohne eine davon günstiger zu machen. Für ein scheinbar unlösbares Problem liegt eine saubere Antwort oft nur einen Abfrageplan entfernt. Horizontale Skalierung wird bisweilen gerade deshalb gewählt, damit niemand die bereits vorhandene Datenbank lesen muss.

Bevor wir etwas aufteilen, arbeiten wir in dieser Reihenfolge: hinsehen, straffen, Lesezugriffe vom Primärserver nehmen, die Tabellen um den Engpass partitionieren und erst dann sharden, wenn ein Rechner mit sinnvoller Arbeit ausgelastet ist. Der Blitzverkauf, bei dem alle Käufer auf dieselbe Zeile schreiben, wird in Das letzte Ticket genau einmal verkaufen durchgespielt. Diese Notiz beschreibt die Methode darum herum. Ihr letzter Abschnitt untersucht den entgegengesetzten Vorschlag: von jedem Server aus verkaufen und sich für doppelt verkaufte Einheiten entschuldigen.

Die technischen Erkenntnisse auf Ihr System anwenden

Die technischen Überlegungen hinter „Erst optimieren, dann die Datenbank aufteilen“ können Ihre nächste Änderung unterstützen. Wir können ähnliches Verhalten in Ihrer Software, Ihren Daten oder Ihrer Infrastruktur untersuchen. Eine Struktur, die zur tatsächlichen Arbeitsweise Ihres Unternehmens passt, damit sich auch spätere Fragen noch beantworten lassen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Argumentation und ihre Grenzen nachvollziehen

Lesen Sie die Argumentation und ihre Grenzen in dieser Notiz. Wir können prüfen, ob dieselbe Diagnose in Ihrer Umgebung zutrifft. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

6 ETL/ELT-Pipelines entworfen und betrieben Kontakt aufnehmen →

Zuerst hinsehen

PostgreSQL erfasst bereits die meisten benötigten Informationen. Stellen Sie ihm vor jeder Änderung vier Fragen.

Welche Anweisungen kosten insgesamt am meisten? `pg_stat_statements`, eine über `shared_preload_libraries` geladene Erweiterung, erfasst jede normalisierte Anweisung samt Aufrufzahl und Laufzeit. Sortieren Sie nach Gesamtlaufzeit, nicht nach Mittelwert: Eine 2-ms-Abfrage, die eine Million Mal pro Stunde läuft, kostet mehr als ein zweimal ausgeführter Bericht mit zwei Sekunden Laufzeit.

```
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;
```

Was tut die teuerste Anweisung tatsächlich? `EXPLAIN (ANALYZE, BUFFERS)` führt sie aus und zeigt den gewählten Plan, die tatsächlich gelieferten Zeilen jedes Schritts im Vergleich zur Erwartung und die Zahl gelesener Seiten. Ein Scan über eine Million Zeilen für zehn Ergebnisse oder eine um den Faktor hundert falsche Schätzung erklärt meist das ganze Problem. `ANALYZE` führt aus, was es erläutert; fassen Sie einen Schreibzugriff deshalb in `BEGIN` und `ROLLBACK` ein.

Wer wartet auf wen?

```
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';
```

Wenn hier in einer Minute mit hoher Last Zeilen erscheinen, warten Transaktionen aufeinander, nicht auf Hardware. Ein leistungsfähigerer Rechner ändert nichts daran, dass Updates derselben gemeinsam genutzten Zeile

nacheinander ausgeführt werden müssen. Zusätzliche Anwendungsserver können weitere schreibende Clients in diese Warteschlange bringen.

Arbeitet der Server oder jongliert er? Vergleichen Sie die aktiven Sitzungen in `pg_stat_activity` mit der Zahl der Kerne. Sind um ein Vielfaches mehr Sitzungen aktiv als Kerne vorhanden, verbringt der Server seine Zeit mit dem Wechsel dazwischen. Ein Verbindungspool hilft dagegen, bevor neue Hardware es tut. Lesen Sie danach `n_dead_tup` in `pg_stat_user_tables` : Eine Tabelle mit mehr toten als lebenden Zeilen wird durch ihren eigenen Müll hindurch gescannt. Hier kommt Autovacuum nicht hinterher; die Datenbank ist nicht zu klein.

Schreiben Sie die Befunde auf. Sie bilden die Ausgangsbasis für jeden späteren Schritt. Oft enthalten sie bereits die ehrliche Antwort auf die Frage, ob Sharding nötig ist.



Das Vorhandene straffen

Ungefähr nach dem Verhältnis von Nutzen zu Aufwand geordnet.

Indizes, die zu den Fragen passen. Die Pläne oben zeigen die Scans. Ein Index auf den Filterspalten einer häufigen Abfrage macht aus einem Scan einen gezielten Zugriff. Ein partieller Index enthält nur die Zeilen, die eine Abfrage interessieren können — etwa noch verkäufliche Einheiten. Er schrumpft also während des Verkaufs, statt zu wachsen:

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

Jeder Index verursacht auch bei jedem Schreibzugriff Aufwand. Entfernen Sie deshalb diejenigen, deren `idx_scan` in `pg_stat_user_indexes` bei null bleibt.

Typen, die den Wert beschreiben. Schnell geschriebene Schemas, ob von Menschen oder Modellen, greifen zu `TEXT` , wo `SMALLINT` oder `BOOLEAN` genügen würden. Ein `BOOLEAN` benötigt ein Byte, ein `SMALLINT` zwei und das als

Text gespeicherte `'yes'` vier. PostgreSQL ergänzt zudem jede Spalte fester Breite um Auffüllbytes für ihre Ausrichtung. Eine zufällige Spaltenreihenfolge kostet daher ebenfalls Bytes. Eine schnell geschriebene Bestandseinheit:

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

Und dieselbe Einheit mit aussagekräftigen Typen, die breitesten zuerst:

```
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
);
```

Unter PostgreSQL 18 misst die erste Zeile 72 Bytes und die zweite 54, ein Viertel weniger. So passt mehr von der Tabelle in den Arbeitsspeicher. Die Typen verweigern außerdem Werte wie `'maybe'`, die sonst die Anwendung abfangen müsste. Ein `CHECK`-Constraint tut dasselbe für Wertebereiche: `quantity smallint CHECK (quantity > 0)`.

Weniger Hin- und Rückwege, die Arbeit bei den Daten. Jede von der Anwendung gesendete Anweisung kostet einen Übertragungsweg hin und zurück sowie einen Parserdurchlauf. Eine Schleife, die hundert Zeilen liest und einzeln aktualisiert, benötigt hundertundeinen solchen Weg, wo eine mengenbasierte Anweisung genügte. Eine Seite, die sämtliche Einheiten eines Artikels abruft, um freie zu zählen, überträgt Tausende Zeilen für eine einzige Zahl. Unsere Regel: Was die Daten sehen muss, kommt in die Datenbank — Filtern, Zählen, ein vom vorhandenen Zustand abhängiger Schreibzugriff. Geschäftsregeln bleiben in der Anwendung, denn zusätzliche CPU-Kapazität lässt sich an der Datenbank am schwersten ergänzen.

Updates, die auf ihrer Seite bleiben. Eine mehrmals pro Minute geänderte Zeile hinterlässt jedes Mal eine neue Version. Ändert ein Update keine indexierte Spalte und hat die Seite Platz, legt PostgreSQL die neue Version auf derselben Seite ab und fasst keinen Index an: ein Heap-only Tuple oder HOT-Update. Halten Sie deshalb häufig geänderte Spalten aus den Indizes heraus und lassen Sie Platz auf den Seiten:

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

Die Einstellung gilt für danach geschriebene Seiten; setzen Sie sie also vor dem Laden der Zeilen. In der Tabelle oben waren 2.000 Reservierungen, die `reserved`, `reserved_at` und `order_id` setzten, sämtlich HOT. Von 2.000 Updates auf `sold`, das der partielle Index liest, war keines HOT. Verkauft wird jede Einheit einmal; reserviert werden kann sie mehrfach. Das Verhältnis von `n_tup_hot_upd` zu `n_tup_upd` in `pg_stat_user_tables` zeigt die Quote im laufenden System.

Eine Zeile sagt, wer zuerst kam

Eine Einheit geht an den ersten Käufer, dessen Anweisung sie erreicht. Die Reservierung läuft nach der Uhr der Datenbank ab. Zwei Spalten tragen sie: `reserved`, ein boolescher Wert, und `reserved_at`, der Zeitpunkt seiner Setzung.

```
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;
```

Eine zurückgegebene Zeile: Dieser Käufer hat die Einheit. Keine: Jemand anderes war früher, oder sie ist verkauft. Zwei Reservierungen derselben Einheit warten an ihrer Zeilensperre. Unter READ COMMITTED, dem PostgreSQL-Standard, prüft die zweite Transaktion die `WHERE`-Bedingung erneut gegen die von der ersten

festgeschriebene Zeile. Genau eine erhält daher eine Zeile zurück. Eine mehr als zehn Minuten alte Reservierung gilt als frei; für die korrekte Antwort muss sie nicht erst aufgeräumt werden.

Der boolesche Wert allein ist deshalb keine vollständige Antwort. `reserved` bleibt nach Ablauf einer Reservierung wahr. Nur beide Spalten zusammen sagen, ob eine Einheit gehalten wird; jede entsprechende Abfrage muss beide prüfen. Ein periodischer Bereinigungslauf, der abgelaufene Reservierungen zurücksetzt, hält die Tabelle für Leser von `reserved` allein verständlich. Die Korrektheit wartet aber niemals darauf. Ein Constraint hält das Paar konsistent:

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

Die Ticket-Notiz speichert die Frist `held_until` statt des Beginns; das Prinzip ist dasselbe. Wählt der Käufer eine bestimmte Einheit — einen Sitzplatz oder einen nummerierten Druck —, benennt die Anweisung sie wie hier. Ist jede Einheit geeignet, wählt die dortige `SKIP LOCKED`-Variante eine aus, die niemand anders hält.



Lesezugriffe vom Primärserver nehmen

Der meiste Verkehr eines Shops liest: Produktseite, die Anzeige „3 übrig“, Warenkorb, Bestellhistorie. Nichts davon benötigt den Primärserver, der für Schreibzugriffe frei bleiben muss. Ein Streaming-Replikat übernimmt diese Zugriffe. Eines hinzuzufügen ist eine Konfigurationsänderung, während ein Shard ein Projekt ist.

Zwei Einschränkungen gehören dazu. Ein Replikat hinkt etwas hinterher. Ein Käufer, der gerade eine Einheit reserviert hat und danach seinen Warenkorb vom Replikat liest, sieht die Reservierung womöglich nicht. Lesen Sie gerade selbst geschriebene Daten vom Primärserver oder leiten Sie den Benutzer nach jedem Schreibzugriff für einige Sekunden dorthin. Außerdem kann ein langer Bericht auf einem Replikat abgebrochen werden, wenn er das Nachspielen von

Änderungen blockiert. Die PostgreSQL-Dokumentation nennt dies Abfragekonflikte und beschreibt die Einstellungen, die solche Konflikte gegen Replikationsverzögerung abwägen.

Der günstigste Lesezugriff erreicht keine Datenbank. Sind alle Einheiten eines Artikels reserviert, kann bis zum Ablauf oder zur Freigabe einer Reservierung niemand kaufen. Die Datenbank kann genau sagen, wann das ist:

```
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;
```

Ist `free` null, bezeichnet `until` den frühesten Reservierungsablauf: Speichern Sie „vollständig reserviert“ genau bis zu diesem Zeitpunkt im Cache, niemals länger. Bis dahin weist der Cache jeden Besucher ab statt der Datenbank. Ist `free` null und `until` leer, sind alle Einheiten verkauft und die Antwort ist endgültig. Eine vorzeitig freigegebene Reservierung — eine abgelehnte Karte, ein aufgegebener Warenkorb — löscht den Cacheeintrag, sodass die Einheit sofort wieder verkäuflich ist.

Der Cache sagt ausschließlich Nein. Ein Ja kommt immer vom bedingten `UPDATE` oben. Ein veralteter Cache kann daher für wenige Sekunden einen Verkauf verhindern, aber niemals eine Einheit doppelt verkaufen. Das ist das zwischengespeicherte „ausverkauft“ der Ticket-Notiz, ergänzt um eine Frist.



Die Tabellen um die umkämpfte Zeile partitionieren

Partitionierung teilt eine Tabelle anhand eines Schlüssels — etwa Monat oder Mandant — unter einem gemeinsamen Namen in mehrere Teile. Was sie nicht leistet, muss klar sein: Eine einzelne umkämpfte Zeile wird nicht schneller. Sie liegt weiterhin in einer Partition auf einer Maschine hinter einer Sperre.

Klein bleiben dadurch die Tabellen rund um den Engpass. Bestellungen, Reservierungshistorie und Ereignisse wachsen ständig; die meisten Abfragen benötigen nur den jüngsten Teil:

```
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');
```

Eine Abfrage mit Datumsbereich liest nur die darin liegenden Partitionen. Die Indizes dieses Monats bleiben klein genug für den Arbeitsspeicher.
Die Aufbewahrung erfordert kein `DELETE` über Millionen Zeilen mehr, samt anschließender Aufblähung und Vacuum-Arbeit, sondern eine einzige Anweisung:

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

Der Primärschlüssel muss den Partitionsschlüssel enthalten; deshalb lautet er `(id, created_at)`. Eine Abfrage ohne diesen Schlüssel liest jede Partition. Partitionieren Sie daher nach dem bereits verwendeten Abfragefilter.

Wann Sharding sinnvoll ist

Eine Grenze kann Sharding nicht verschieben: Jeder Artikel hat genau eine zuständige Instanz. Eine Einheit kann nicht gleichzeitig auf zwei Servern reserviert werden. Sharding verschiebt deshalb ganze Artikel, niemals einen

halben. Übersteigt ein einzelner Artikel die Kapazität seiner Maschine, teilen die Buckets der Ticket-Notiz seinen Bestand in kleinere Teile. Jeder Teil hat wiederum genau eine zuständige Instanz.

Innerhalb dieser Grenze halten wir Sharding für sinnvoll, sobald einer der folgenden Fälle zutrifft — vorher nicht:

- **Ein Rechner ist mit sinnvoller Arbeit ausgelastet.** Die Anweisungen sind bereits schlank — je ein bedingter Schreibzugriff, nichts Externes innerhalb einer Transaktion, auf ihrer Seite bleibende Updates —, und dennoch sind CPU oder Write-ahead Log des Primärservers auf der größten wirtschaftlich sinnvollen Maschine ausgelastet, über viele Artikel hinweg statt nur einen. Ein einzelner stark nachgefragter Artikel ist ein Fall für Buckets, nicht für Sharding.
- **Die Daten sind über ihre Wartbarkeit hinausgewachsen.** Eine Sicherung lässt sich nicht innerhalb der zugesagten Zeit wiederherstellen, ein Index nicht in den ruhigen Stunden aufbauen oder eine Tabelle nicht darin bereinigen, oder die aktiv genutzten Daten passen nicht mehr in den Arbeitsspeicher.
- **Isolation ist eine Anforderung.** Ein Mandant muss seine Daten laut Vertrag oder Aufsicht getrennt halten oder darf beim Verkauf keine Festplatte mit anderen teilen.

Die Kosten fallen bei allem anderen an. Migrationen, Sicherungen, Wiederherstellungen und Berichte müssen die getrennten Shards berücksichtigen; ein Join über Shards kann in den Anwendungscode verlagert werden, und eine Abfrage, die ihren Shard-Schlüssel nicht kennt, muss unter Umständen alle Shards anfragen. Diese Rechnung sollen die vorherigen Abschnitte vermeiden. Ist sie einmal bezahlt, lässt sich das schwer zurückdrehen.

Wenn eine Entschuldigung günstiger ist als eine Sperre

Für den umgekehrten Weg gibt es ein ernstzunehmendes Argument. Jeder Server verkauft aus dem für ihn sichtbaren Bestand. Gelegentlich geht eine Einheit an zwei Käufer; der Kundendienst erstattet einem davon das Geld mit einer Entschuldigung. Die Geschäftslogik wird letztlich konsistent, ausgelegt auf die 99,9 % der Anfragen ohne Kollision. Eine asynchrone Schleife findet die Kollisionen anschließend und gleicht sie aus. Dafür muss der Shop nie „versuchen Sie es erneut“ sagen. Helland und Campbell formulieren das Argument in

Building on Quicksand besonders stark: Ein System, das Arbeit bestätigt, bevor seine Sicherung sie hat, macht jedes Versprechen probabilistisch. Unternehmen haben nicht einzuhaltende Zusagen schon immer mit Entschuldigungen bereinigt.

Dem stimmen wir weitgehend zu. Wir widersprechen darin, wo dieser Preis gezahlt wird.

Der Großteil eines Shops sollte letztlich konsistent sein. Katalog, Suchindex, die Anzeige „3 übrig“, Warenkorb, Empfehlungen, E-Mails, Lagerexport und Berichte dürfen alle Sekunden hinterherhinken. Sie auf den Primärserver warten zu lassen kostet Tempo und bringt nichts. Genau dafür wurde Dynamo entwickelt: Amazons Warenkorb vergisst oder verweigert ein Hinzufügen auch bei Maschinenausfällen nie. Der Preis ist, dass ein Zusammenführen einen bereits entfernten Artikel zurückbringen kann. Nach der Reservierung läuft der Checkout der Ticket-Notiz bereits asynchron: eine Saga lokaler Schritte, mindestens einmal zugestellte Nachrichten und idempotente Empfänger mit der Bestellung als Schlüssel. Ein Abgleichslauf gehört in jede Anbindung an einen Zahlungsanbieter, weil Webhooks verloren gehen.

Die Reservierung ist die eine Ausnahme. „Selten“ setzt seltene Kollisionen voraus; im Blitzverkauf sind die Kollisionen der Verkauf. Ein Server, der aus seiner eigenen Sicht verkauft, macht weiter, bis er von den Verkäufen der anderen erfährt. Der Überverkauf entspricht deshalb ungefähr der Versuchsrate mal Replikationsverzögerung. In einer einfachen Simulation teilen zwei Regionen 2.000 Versuche pro Sekunde. Jede verkauft aus ihrem eigenen Zählwert und erfährt 200 Millisekunden später von den Verkäufen der anderen. Bei fünf Durchläufen eines Verkaufs von 10.000 Einheiten wurden je Lauf zwischen 191 und 205 nicht vorhandene Einheiten verkauft. Die Rechnung sagt 200 voraus: Versuchsrate mal Verzögerung mal Anteil der Verkäufe in der anderen Region. Mit einer zuständigen Instanz je Artikel verkaufte dasselbe Modell genau den Bestand. Das geschieht bei jedem Ausverkauf, ob etwas ausfällt oder nicht — und zwar am Höhepunkt, wenn die meisten Käufer zusehen.

Der Verfügbarkeitsgewinn ist kleiner, als er klingt. Gilbert und Lynch bewiesen die Abwägung hinter letztlicher Konsistenz: Während einer Netzpartition kann ein System antworten oder konsistent sein, nicht beides. Abadis PACELC ergänzt die andere Hälfte. Ohne Partitionierung geht es um Latenz gegenüber Konsistenz; eine zuständige Instanz je Artikel ist mit schlanken Anweisungen bereits schnell. Keine der beiden Architekturen erreicht 100 %: Zahlungsanbieter,

DNS und die Netze dazwischen haben eigene Ausfälle. Die strenge Architektur gibt nur einen engen Bereich auf. Ist die zuständige Instanz für einen Artikelbestand unerreichbar, stoppt dessen Verkauf für die Sekunden des Failovers. Der Abschnitt der Ticket-Notiz zum Pausieren eines Artikels begrenzt die Pause auf Artikel der ausgefallenen Maschine. Der übrige Shop verkauft weiter.

Eine Entschuldigung ist keine Erstattung. Die Erstattung ist der günstige Teil. Der Käufer hat eine Zusage erhalten und vielleicht schon den Zug gebucht, einen Tag freigenommen und Freunden Bescheid gesagt. „Ausverkauft“ eine Sekunde zu früh ist eine Enttäuschung. Eine zurückgenommene bestätigte Bestellung ist ein gebrochenes Versprechen. Der Kundendienst ist für normale Tage besetzt; Überverkäufe treffen gemeinsam in der geschäftigsten Minute des Jahres ein. Bevor Sie die Rücknahme bestätigter Bestellungen einplanen, holen Sie juristischen Rat dazu ein, wozu eine Bestätigung den Shop verpflichtet.

Die Asynchronität behalten und vor die Zusage verlagern. Richtig an dem Vorschlag ist, dass der Käufer nicht auf eine Sperre warten muss. Nehmen Sie jeden Klick in ein dauerhaftes, repliziertes Protokoll auf, eine Partition je Artikel, und teilen Sie dem Käufer mit, dass die Antwort folgt. Die Annahme eines Klicks hängt dann nur vom Protokoll ab. Ein über drei Maschinen repliziertes Protokoll nimmt bei Ausfall einer davon weiter Schreibzugriffe an. Eine zuständige Instanz je Artikel liest es der Reihe nach, entscheidet mit Arbeitsspeichergeschwindigkeit und sendet die Antwort binnen Sekunden als Nachricht zurück. Bei ihrem Failover warten Anfragen im Protokoll, statt abgelehnt oder doppelt verkauft zu werden. Die Zusage ist eine Reservierung; danach folgt die Zahlung wie im Checkout der Ticket-Notiz. Niemand bezahlt also für eine nicht erhaltene Einheit. Der Abgleich bleibt erhalten, findet aber vor statt nach dem Versprechen statt.

Den Bestand verteilen, nicht den Zählwert. Müssen Regionen ohne gegenseitige Rückfrage verkaufen, erhält jede ein Bestandskontingent und verkauft daraus eigenständig. Ist es erschöpft, fordert die Region mehr an; gegen Verkaufsende werden die Kontingente wieder bei einer zuständigen Instanz zusammengeführt. Bailis und Kollegen untersuchen, welche Regeln Server ohne Kommunikation einhalten können. Ein Bestand, der nicht unter null sinken darf, gehört nicht dazu. Kontingente beseitigen diese Kommunikation nicht, sondern verlagern sie aus dem häufigen Zugriffspfad auf den seltenen Moment des Nachfüllens. Das ist die letztlich konsistente Architektur, die die Invariante erhält. Ihr einziger Preis liegt im Auslauf: Ein Käufer wird in einer Region abgewiesen, während eine andere noch Bestand hat.

Wann Überverkauf die richtige Regel ist. Manche Güter lassen sich nachproduzieren: ein Druck auf Bestellung, ein noch beim Lieferanten liegendes T-Shirt, ein Lizenzschlüssel. Ein Verkauf über den Zählwert hinaus ist dann eine Nachbestellung mit ehrlichem Liefertermin; letztliche Konsistenz ist die richtige Architektur. Fluggesellschaften überbuchen bewusst. Sie kennen ihre Nichterscheinensquoten, kalkulieren das Risiko und entschädigen nach vorher festgelegten Regeln. Das ist der Maßstab: Überverkauf ist akzeptabel, wenn das Unternehmen Umfang und Abhilfe vor dem Verkauf gewählt hat. Er ist ein Fehler, wenn er die Person trifft, die zufällig zuletzt klickt.

Wenn Sie Entschuldigungen wählen, bauen Sie den Prozess wie Zahlungen. Die vorgeschlagene Schleife ist ein Erstattungsprozess. Sie benötigt alles, was jeder Prozess mit Geldbewegungen braucht:

- **Eine ID je Verkauf**, damit eine Wiederholung derselbe Verkauf bleibt, und ein nur ergänzbares Verkaufsbuch. Der Bestandswert wird daraus abgeleitet, niemals direkt geändert.
- **Eine Regel darüber, wer die Einheit behält**, die sich auch dem benachteiligten Käufer erklären lässt. Ordnen Sie nach dem Protokoll, nicht nach den Uhren verschiedener Server, deren Abweichung von wenigen Millisekunden sonst über die Erstattung entscheidet.
- **Idempotente, protokollierte Ausgleichsaktionen:** stornieren, Zahlung aufheben oder erstatten, den Käufer informieren. Das sind die maschinell ausgeführten Rücknahmeschritte der Saga.
- **Eine Frist.** Der Lauf muss jeden Überverkauf vor dem Versand entdecken — und bevor der Käufer ihn entdeckt.
- **Ein gemessenes Budget.** Überverkäufe je Verkaufsaktion im Vergleich zur vom Unternehmen freigegebenen Zahl, neben der Invariante auf demselben Dashboard, mit Alarm bei ausgeschöpftem Budget.

Unsere Antwort lautet: letztliche Konsistenz überall um die Reservierung herum, strenge Konsistenz bei der Reservierung selbst und eine asynchrone Antwort an den Käufer. Für den häufigen Fall entwickeln — doch beim Blitzverkauf ist die Kollision der häufige Fall. Wünscht das Unternehmen im Tausch für etwas einige Überverkäufe, muss es die Zahl selbst wählen, die Abhilfe vor dem Verkauf festlegen und das Ergebnis messen.

Was wir der nächsten betroffenen Person sagen würden

- **Vor dem Kauf lesen.** `pg_stat_statements` und ein `EXPLAIN` beantworten mehr Kapazitätsfragen als ein neuer Server.
- **Die Typen sollen die Daten beschreiben.** Ein als Boolean gespeicherter Boolean ist kleiner und verweigert Unsinn.
- **Nur die Arbeit an die Datenbank senden, die die Daten sehen muss.** Filtern und Zählen gehören dorthin. Geschäftsregeln nicht.
- **Der Cache sagt Nein; nur die Datenbank sagt Ja.** Ein veralteter Cache kostet dann Sekunden, niemals eine Einheit.
- **Partitionierung ist Aufräumen.** Sie hält die Tabellen rund um eine umkämpfte Zeile klein und hilft der Zeile selbst nicht.
- **Sharden, wenn eine Maschine mit sinnvoller Arbeit ausgelastet ist**, nicht wenn sie voller wartender Vorgänge ist.
- **Vor der Zusage abgleichen, nicht danach.** Alles um die Reservierung herum kann letztlich konsistent sein. Eine so ausgeführte Reservierung selbst führt bei jedem Verkauf zu Überverkäufen in Höhe der Versuchsrate mal Verzögerung.

Quellen

- PostgreSQL: pg_stat_statements
- PostgreSQL: EXPLAIN verwenden
- PostgreSQL: das kumulative Statistiksystem
- PostgreSQL: numerische Typen
- PostgreSQL: Zeichentypen
- PostgreSQL: Boolean-Typ
- PostgreSQL: partielle Indizes
- PostgreSQL: Heap-only Tuples (HOT)
- PostgreSQL: Tabellenpartitionierung
- PostgreSQL: Hot Standby und Abfragekonflikte
- Pat Helland und Dave Campbell, Building on Quicksand
- Seth Gilbert und 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 und Kenneth Salem, Sagas
- Bailis et al., Coordination Avoidance in Database Systems

Alle Blogbeiträge

Messen, bevor Sie ändern

Bringen Sie einen repräsentativen Datensatz, die Fragen, die er beantworten muss, und den aktuellen Engpass mit. Wir können vereinbaren, wie Qualität, Durchsatz und Wiederherstellung gemessen werden.

Preise ansehen Kontakt aufnehmen →

Ihre Beobachtungen besprechen

Besprechen Sie „Erst optimieren, dann die Datenbank aufteilen“ im Kontext Ihres eigenen Projekts. Erzählen Sie uns, was Sie beobachtet haben und wie ein nützliches Ergebnis aussehen würde.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von **D.**

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/notes/optimise-before-you-shard/