Engineer

Das letzte Ticket genau einmal verkaufen

13. Oktober 2026

Ein Blitzverkauf stellt zehntausend Einheiten gleichzeitig einer Million Menschen gegenüber. Der interessante Fehler ist nicht der Ausfall des Shops. Er bleibt erreichbar und verkauft zehntausendvierzig. Der anschließende Reflex — die Datenbank ist zu klein, wir müssen sharden — beantwortet eine Frage, die niemand gestellt hat. Die Datenbank war nie zu groß. Jeder Käufer schrieb auf dieselbe Zeile.

Wir zerlegen das Problem hier, bevor es ein Projekt erreicht: woher die Grenze kommt, welche einzelne Anweisung den Großteil davon beseitigt und was zu bauen ist, wenn eine Datenbank tatsächlich nicht mehr genügt.

Die technischen Erkenntnisse auf Ihr System anwenden

Die technischen Überlegungen hinter „Das letzte Ticket genau einmal verkaufen“ 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 →

Eine Zeile, jeder Käufer

Zwanzig Millionen über einen Cluster verteilte Zeilen sind ein gelöstes Problem. Eine Zeile, die jede Anfrage ändern muss, ist es nicht. Mehr Maschinen helfen nicht, weil diese Zeile auf einer davon liegt. Verteilte Datenspeicher sagen das deutlich: DynamoDB stellt einer einzelnen Partition ungefähr 1.000 Schreibeinheiten pro Sekunde bereit. Jeder Schreibzugriff auf ein Element landet in dessen Partition, unabhängig von der Zahl der anderen Partitionen.

Dahinter steht ein Theorem. Bailis und Kollegen nennen eine Invariante invariant-konfluent, wenn Replikate eigenständig Schreibzugriffe annehmen und sich dennoch zu einem gültigen Zustand zusammenführen lassen. `stock >= 0` ist das Lehrbuchbeispiel einer Invariante, für die das nicht gilt: Zwei Replikate mit jeweils einer sichtbaren verbleibenden Einheit verkaufen beide diese Einheit. Kein Zusammenführen kann den zweiten Verkauf zurücknehmen. Überverkäufe sicher zu vermeiden erfordert Koordination. Die Frage ist nur, wie viel davon jeder Verkauf bezahlen muss.

Die entscheidende Zahl

Die Grenze für einen Artikel entspricht ungefähr der Zahl der Verkäufe pro Sperre geteilt durch die Dauer der Sperre:

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

Was die Zeile festhält	Grenze je Zeile
Sperren, dann Zahlung aufrufen	~2/s
Sperren, dann Anwendungslogik	~100–200/s
Ein bedingtes `UPDATE`	~500–2.000/s
Stapel von 100 je Commit	~50.000/s
Eine schreibende Instanz im Arbeitsspeicher	~100.000+/s

Dies sind Größenordnungen, keine Benchmarks; die Hardware verschiebt sie. Das Grundprinzip ändert sie nicht. Bei einer kurzen Sperre wartet der Commit hauptsächlich darauf, dass sein Protokoll auf der Festplatte ankommt. Deshalb gewinnt Stapelverarbeitung zwei Größenordnungen: Hundert Verkäufe teilen sich einen Flush.

Es bleiben vier Hebel; jede der folgenden Architekturen nutzt einen davon:

- **Die Sperre kürzer halten** — eine Anweisung, nichts über das Netzwerk darin.
- **Mehr je Sperre erledigen** — die Bestandsabzüge bündeln.

- **Mehr Sperren schaffen** — den Bestand auf mehrere Schlüssel aufteilen.
- **Weniger Anfragen an die Sperre schicken** — die meisten Käufer vorher abweisen.

  Der letzte Punkt zählt mehr, als er klingt. Bei einer Million Käufern und zehntausend Einheiten müssen 990.000 Anfragen scheitern. Ablehnungen bilden den Großteil des Verkehrs. „Ausverkauft“ muss deshalb die günstigste Antwort des Systems sein: ein Warteraum vor dem Verkauf, eine Schranke, die ungefähr so viele Käufer wie vorhandene Einheiten einlässt, ein vom CDN ohne Rückfrage ausgeliefertes `sold_out` -Flag und ein Ratenlimit je Konto.

Zuerst die eine Datenbank korrigieren

Die langsame oder überverkaufende Variante sieht meist so aus:

```
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;
```

Die Sperre ist richtig. Falsch ist, was währenddessen passiert: Jede Millisekunde der Anwendung zwischen `SELECT` und `UPDATE` ist eine Millisekunde Wartezeit für den nächsten Käufer. Ohne `FOR UPDATE` ist es schlimmer. Zwei Käufer lesen 1, beide schreiben 0, und eine Einheit ist doppelt verkauft.

Die Korrektur vereint Prüfung und Bestandsabzug in einer Anweisung:

```
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.
```

Erreicht in PostgreSQL unter READ COMMITTED ein zweiter Käufer die noch gesperrte Zeile, wartet er und prüft danach die `WHERE` -Bedingung erneut gegen die von der ersten Transaktion festgeschriebene Zeile. Hat dieser Verkauf die letzte Einheit genommen, schlägt die Bedingung fehl. Der zweite Käufer erhält keine Zeile — ausverkauft — statt einer negativen Zahl. Unter REPEATABLE READ oder SERIALIZABLE erhält er stattdessen einen Serialisierungsfehler. Das ist sicher, macht Konkurrenz aber zu Wiederholungsversuchen. Diese Anweisung ist für READ COMMITTED geschrieben. MySQLs InnoDB gelangt zur selben Antwort, weil ein `UPDATE` dort auf jeder Isolationsstufe die zuletzt festgeschriebene Zeile liest. Ergänzen Sie außerdem den Constraint, damit ein Fehler an anderer Stelle deutlich scheitert, statt nicht vorhandenen Bestand zu verkaufen:

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

Danach die übrige Arbeit an der einzelnen Datenbank, ungefähr nach Nutzen geordnet:

- **Nichts Externes innerhalb der Transaktion.** Kein Zahlungsaufruf, keine HTTP-Anfrage, keine veröffentlichte Nachricht. Das gehört davor oder danach.
- **Die umkämpfte Zeile zuletzt anfassen.** Zuerst Bestellung und alles Weitere einfügen, damit die Sperre nur für die letzte Anweisung und den Commit besteht.
- **Keine optimistische Sperrung für einen stark beanspruchten Zähler.** Eine Versionsspalte macht Konkurrenz zu Wiederholungen. Während eines Verkaufs sind diese Wiederholungen selbst die Last.
- **Lesezugriffe von anderswo bedienen.** Die Anzeige „12 übrig“ auf der Produktseite darf von einem Replikat oder Cache mit einer Sekunde Verzögerung stammen. Nur der Verkauf benötigt den Primärserver.
- **Dem Bestand eine eigene Datenbank geben**, damit ein Verkauf nicht hinter dem Katalog auf dieselbe Festplatte wartet.
- **Verbindungen bündeln.** Hunderttausend Käufer sollten über PgBouncer einige Dutzend Verbindungen belegen, nicht Tausende Backends, die auf eine Sperre warten.

Sitzplätze haben eine andere Struktur, weil jede Einheit unterscheidbar ist, und benötigen eine andere Anweisung. Jeder Sitz ist eine Zeile. Ein Käufer überspringt von anderen Transaktionen gehaltene Zeilen, statt dahinter

zu warten:

```
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;
```

Eine abgelaufene Reservierung ist einfach wieder verfügbar. Für die richtige Antwort ist kein Bereinigungslauf nötig. `now()` stammt zudem von der Datenbankuhr; Anwendungsserver mit unterschiedlichen Uhren können sich daher nicht über den Ablaufzeitpunkt uneinig sein.

Für viele Shops ist dieser Abschnitt bereits die gesamte Lösung. Setzen Sie ihn vor allem Folgenden um, denn Sharding ist eine Einbahnstraße: leicht hineinzugehen, teuer zurückzugehen.

Erst reservieren, dann bezahlen

Ein Käufer, der die letzte Einheit nimmt und danach das Kartenformular verlässt, hat sie nicht gekauft. Ein Verkauf ist deshalb kein einzelner Schreibzugriff, sondern ein kurzer Lebenszyklus:

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

- **Die Reservierung wird durch die Anfrage identifiziert, nicht den Versuch.** Jeder Checkout trägt einen Idempotenzschlüssel. Die Reservierungstabelle enthält `UNIQUE (tenant_id, idempotency_key)`. Ein wiederholter Klick erhält dadurch die vorhandene Reservierung statt einer zweiten Einheit. Ein PostgreSQL-Detail ist dabei wichtig: `INSERT ... ON CONFLICT DO NOTHING RETURNING id` gibt bei einer schon vorhandenen Zeile nichts zurück. „Keine Zeile“ bedeutet also „nachschlagen“, nicht „fehlgeschlagen“.
- **Jeder Übergang ist bedingt.** Bei der Verkaufsbestätigung wechselt eine Reservierung von `held` zu `sold`, aber nur, wenn ihr Zustand noch `held` ist. Eine Freigabe schreibt den Bestand nur zurück, wenn ihr eigenes Update genau eine Zeile geändert hat. Eine doppelte Freigabe — etwa durch Timeout und Abbrechen-Schaltfläche — gibt die Einheit nur einmal zurück.
- **Reservierungen laufen nach Minuten ab.** Fünf bis zehn sind üblich. Verlängern Sie beim Zahlungsbeginn einmal, damit eine langsame Kartenzahlung nicht die gerade bezahlte Einheit verliert.
- **Die Stückzahl je Käufer schon bei der Reservierung begrenzen**, damit ein Konto während seiner Entscheidung nicht den gesamten Bestand halten kann.

Unter allem steht eine einzige Zeile, die nützlichste der Architektur:

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

Sie gilt nach jeder Transaktion, wird durch eine Abfrage geprüft und zeigt bei einer Abweichung einen Überverkauf oder eine verlorene Einheit an — bevor ein Kunde es meldet.

Wenn eine Datenbank nicht reicht

Manchmal reicht sie tatsächlich nicht: viele Mandanten, viele gleichzeitige Verkäufe, mehrere Regionen. Die skalierende Struktur behält eine Regel bei: Für jeden Artikel gibt es genau eine Instanz, die über seinen Bestand entscheidet, einen Bestandsdienst. Dieser wird nach `(tenant_id, item_id)` aufgeteilt, damit die Schreibzugriffe eines Artikels nie zwei Shards betreffen. Die größten Mandanten erhalten eigene Zellen, damit ihr Verkauf niemanden anderen verdrängt. Nur die Reservierung muss synchron sein; alles danach kann als Nachricht erfolgen.

Für den stark nachgefragten Artikel innerhalb eines Shards gibt es vier Werkzeuge. Jedes nutzt einen der genannten Hebel.

Eine vorgeschaltete Schranke. Ein Redis-Zähler, durch ein Lua-Skript vermindert, damit Prüfung und Abzug ein Schritt sind, lässt ungefähr so viele Käufer wie verfügbare Einheiten durch und weist alle anderen in Mikrosekunden ab. Er ist nicht die entscheidende Instanz. Redis repliziert asynchron; nach einem Failover kann der neue Primärserver einige Zulassungen vergessen haben und zu viele Käufer durchlassen. Das bedingte `UPDATE` der Datenbank entscheidet weiterhin. Ein Abgleich setzt die Schranke anhand des Datenbankwerts zurück.

Den Bestand aufteilen. Zehntausend Einheiten können hundert Buckets zu je hundert sein, jeder mit eigener Zeile oder eigenem Shard, und jeder Käufer wird einem zugeteilt: hundert Sperren statt einer. Die Idee ist alt — O'Neils Escrow-Transaktionen von 1986, dann das Demarkationsprotokoll von Barbará und Garcia-Molina, später die begrenzten Zähler von Balegas und Kollegen. Ihr einziger echter Preis liegt im Auslauf. Der Bucket eines Käufers ist leer, während andere noch Bestand haben. Dann muss ein anderer versucht oder Restbestand verschoben werden; gegen Verkaufsende wird wieder auf einen zusammengeführt. Dieselbe Idee gibt jeder Region ein eigenes Kontingent, damit die meisten Verkäufe nicht für eine Rückfrage einen Ozean überqueren müssen.

Eine schreibende Instanz je Artikel. Alle Anfragen für einen Artikel landen in einer Kafka-Partition. Ein einzelner Empfänger hält den Zählwert im Arbeitsspeicher und schreibt seine Entscheidungen stapelweise fest. Ein Thread ohne Sperren: das LMAX-Prinzip. Anfragen tragen eine ID. Bereits entschiedene Anfragen verwirft die schreibende Instanz, weil eine Warteschlange doppelt zustellen kann.

Einheiten als Zeilen. Sind die Einheiten wenige und unterscheidbar — Sitzplätze, nummerierte Auflagen —, legen Sie jede vorab als Zeile an und reservieren Sie mit `SKIP LOCKED` wie oben. Oder halten Sie die Einheiten-IDs in einem Redis-Set und entnehmen eine mit `SPOP`. Es bleibt kein umkämpfter Zähler.

Der übrige Checkout

Zahlung, Bestellung und Beleg sind getrennte Systeme. Keine Transaktion umfasst sie alle. Two-phase Commit ist die Lehrbuchantwort und hier die falsche: Es blockiert bei jedem langsamen Teilnehmer, ergänzt Übertragungswege im meistbeanspruchten Pfad, und Zahlungsanbieter nehmen daran nicht teil. Eine Saga erledigt die Aufgabe als Folge lokaler Schritte, jeweils mit einem Rücknahmeschritt:

```
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
```

Zuerst autorisieren und zuletzt einziehen bedeutet: Trifft eine Zahlung nach Ablauf ihrer Reservierung ein, wird die Autorisierung aufgehoben statt Geld erstattet — es wurde nie Geld bewegt. Jeder Schritt schreibt seine Nachricht in derselben Transaktion wie die angekündigte Änderung in eine Outbox-Tabelle. Change Data Capture veröffentlicht sie; zwischen Speichern und Senden geht nichts verloren. Jeder Empfänger arbeitet idempotent mit der Bestellung als Schlüssel. Nachrichten kommen mindestens einmal an; Idempotenz macht daraus effektiv einmal.

Eine verteilte Sperre gehört nirgends hinein. Kleppmanns Argument: Ein Sperreninhaber kann länger pausieren als seine Lease — durch Garbage Collection oder eine stockende virtuelle Maschine — und danach weiterschreiben. Dagegen hilft ein vom Datenspeicher geprüftes Fencing Token. Kann ein Speicher aber ein Token prüfen, kann er auch `available >= :qty` prüfen. Dann brauchte er die Sperre nie.

Den Artikel pausieren, niemals überverkaufen

Konsistenz und Verfügbarkeit werden je Frage gewählt, nicht je System. „Kann ich diese Einheit verkaufen?“ muss konsistent beantwortet werden. Ist die zuständige Instanz für den Artikelbestand nicht erreichbar, stoppt der Verkauf

dieses Artikels. Alles andere — Katalog, Anzeige „3 übrig“, Bestellhistorie — darf verfügbar bleiben und etwas veraltet sein.

Der unauffälligste Überverkauf entsteht beim Failover. Bei asynchroner Replikation bestätigt der Primärserver einen Verkauf und fällt aus, bevor sein Replikat davon weiß. Das Replikat wird hochgestuft, der Verkauf fehlt, und die Einheit wird erneut verkauft. Die Lösung: Ein Verkauf muss vor der Bestätigung auf mehr als einer Maschine dauerhaft gespeichert sein. Dafür dienen ein synchroner Standby in PostgreSQL (`synchronous_commit` mit `synchronous_standby_names`) oder ein auf Raft aufgebauter Speicher. Das kostet bei jedem Commit Latenz und verlängert genau die Sperrdauer, durch die die Kapazitätsgrenze geteilt wird. Stapelverarbeitung und die Schranke gleichen das aus. Begrenzen Sie auch die Pause: Ein ausgefallener Shard sollte seine Artikel stoppen, nicht den Shop.



Den Nachweis führen

Nichts davon sollte ungeprüft geglaubt werden, auch nicht von uns.

- **Die Invariante überwachen.** `available + held + sold = total` regelmäßig prüfen, grafisch darstellen und bei jeder Abweichung alarmieren. Null Überverkäufe ist ein Service-Level-Ziel mit Alarm, keine Hoffnung.
- **Absichtlich Konkurrenz erzeugen.** Zehntausend gleichzeitige Käufer für hundert Einheiten in der Testsuite, mit der Zusicherung: genau hundert verkauft und keine Zahl unter null.
- **Währenddessen Ausfälle auslösen.** Primärserver beenden, Netzwerk trennen, Prozess pausieren — im Stil von Jepsen — und anschließend die Invariante prüfen.
- **Die Ablehnungen unter Last testen.** Der Ausverkaufspfad muss bei 99 % des Verkehrs schnell bleiben; die vorgelagerte Infrastruktur muss die Last abfangen.
- **Metriken auf einem Dashboard zusammenführen:** Wartezeiten auf Sperren, abgelaufene Reservierungen und Abstand zwischen Schranken- und Datenbankzählwert.

In welcher Reihenfolge

- Zuerst messen. `pg_stat_statements` und die Sperrwartezeiten zeigen, ob die umkämpfte Zeile überhaupt das Problem ist.
- Den Schreibzugriff zu einer bedingten Anweisung machen.
- Last abwerfen: Warteraum, Schranke, zwischengespeichertes „ausverkauft“.
- Dem Bestand einen eigenen Dienst und eine eigene zuständige Instanz geben.
- Nach `(tenant_id, item_id)` sharden, nur wenn die ersten vier Schritte nicht reichen.
- Für den absehbaren Verkauf einen Modus für stark nachgefragte Artikel ergänzen: Buckets oder eine einzelne schreibende Instanz.
- Regionen zuletzt, mit Kontingenten.

Was wir der nächsten betroffenen Person sagen würden

- **Sharding beantwortet eine andere Frage.** Es verteilt viele Zeilen; ein überverkaufter Verkauf betrifft eine Zeile. Klären Sie vor dem Aufbau, welcher Fall vorliegt.
- **Prüfung und Abzug in einer Anweisung vereinen.** Der größte Unterschied zwischen korrekt und überverkauft liegt oft in der Zeit zwischen Lesen und Schreiben einer Zahl.
- **Der Großteil des Verkehrs ist die Antwort Nein.** Ablehnung muss die günstigste Operation des Systems sein.
- **Pausieren, bevor Sie überverkaufen.** Ein Artikel, der dreißig Sekunden nicht verkauft wird, erfordert eine Entschuldigung. Ein doppelt verkaufter Artikel erfordert eine Erstattung, eine Supportwarteschlange und hinterlässt einen Kunden, der sich erinnert.
- **Die Invariante aufschreiben und überwachen.** Es ist eine Zeile und der einzige Nachweis dafür, dass die Architektur im Betrieb funktioniert statt nur bei der Überprüfung.

Quellen

- PostgreSQL: Isolationsstufe Read Committed
- PostgreSQL: `FOR UPDATE` und `SKIP LOCKED`
- PostgreSQL: synchrone Replikation
- 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: Partitionsschlüssel entwerfen
- DynamoDB: Partitionen und Datenverteilung
- microservices.io: Saga
- microservices.io: Transactional Outbox
- Redis: Replikation
- Jepsen

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 „Das letzte Ticket genau einmal verkaufen“ 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/selling-limited-stock-without-overselling/