Engineer

最後の一枚を、一度だけ販売する

2026年10月13日

フラッシュセールでは、一万点の在庫に百万人が同時に向かいます。注目すべき障害は、店が停止することではありません。稼働し続けたまま、一万四十点を売ってしまうことです。その後の反射的な提案、「データベースが小さすぎるから、シャーディングしよう」は、誰も問うていない問題に答えています。データベースが大きすぎたことは、一度もありません。全購入者が、同じ行へ書き込んでいたのです。

プロジェクトで起きる前に、ここで問題を分解します。上限はどこから来るのか、その大半を取り除く一つのSQL文は何か、本当に一つのデータベースで足りなくなったら何を構築するのかを説明します。

技術的な取り組みを自社のシステムに生かす

「最後の一枚を、一度だけ販売する」で扱う技術的な取り組みは、次の変更を考える参考になります。お客様のソフトウェア、データ、インフラで起きる同様の挙動を調査できます。 実際の業務に合った構造を設計し、将来必要になる問いにも答えられるようにします。

サービスを見る お問い合わせ →

推論とその適用範囲をたどる

この技術ノートの推論と、その適用範囲をご覧ください。お客様の環境でも同じ診断が成り立つかを検証できます。 ETL/ELTパイプライン6本を設計・運用 — 6 3種類のDBエンジンにまたがるETLパイプライン

ETL/ELTパイプライン6本を設計・運用 お問い合わせ →

一行に、すべての購入者

クラスタ全体に二千万行を分散させるのは、すでに解かれた問題です。すべてのリクエストが変更する一行は、そうではありません。その行はいずれか一台に存在するため、マシンを増やしても解決しません。分散ストアも、この点を明確にしています。DynamoDBでは、一つのパーティションに毎秒約1,000書き込みユニットを割り当てます。他のパーティションがいくつあっても、一つの項目への書き込みは、その項目のパーティションに集中します。

この構造の背後には定理があります。Bailisらは、各レプリカが独自に書き込みを受け付けても、正しい状態へマージできる不変条件を、invariant-confluentと呼びます。`stock >= 0` は、それに当てはまらない典型です。残り一点と見える二つのレプリカが、それぞれ販売すると、どのようにマージしても二つ目の販売は取り消せません。超過販売を決して起こさないためには、協調が必要です。問題は、一回の販売でどれだけの協調の費用を払うかだけです。

上限を決める数値

一商品の上限は、おおよそ、一つのロックで行う販売数を、ロックの保持時間で割ったものです。

```
一商品当たりの最大販売数/秒 ≈ ロック一回当たりの販売数 ÷ ロック保持時間
```

行を保持する処理	一行当たりの上限
ロックしてから決済を呼び出す	~2/s
ロックしてからアプリのロジックを実行する	~100–200/s
一つの条件付き `UPDATE`	~500–2,000/s
一回のコミットに100件をまとめる	~50,000/s
メモリー上の単一の書き手	~100,000+/s

これらは桁の目安であり、ベンチマークではありません。数値はハードウェアで変わりますが、構造は変わりません。短いロックの大半は、コミットがログのディスク到達を待つ時間です。まとめて処理すると二桁の差が出るのは、百件の販売が一回のフラッシュを共有するためです。

調整できる点は四つで、以下の各設計はいずれかを使います。

- **ロックの保持時間を短くする。** 一つの文にし、その中にネットワーク処理を入れません。
- **一つのロックで多く処理する。** 減算をまとめます。
- **ロックの数を増やす。** 複数のキーに在庫を分けます。

- **ロックに届くリクエストを減らす。** 到達する前に大半の購入者へ不可と答えます。

最後の点は、思う以上に重要です。百万人の購入者に一万点の在庫なら、990,000件のリクエストは失敗しなければなりません。拒否がトラフィックの大半なので、「売り切れ」を最も安く返せる回答にします。販売前の待合室、在庫数程度の購入者を通すゲート、誰にも問い合わせずCDNが返す `sold_out` フラグ、アカウントごとのレート制限を使います。



まず一つのデータベースを直す

極端に遅い、または超過販売する実装は、通常この形です。

```
BEGIN;
SELECT available FROM inventory
 WHERE item_id = :item FOR UPDATE;
-- アプリケーションが在庫数を確認し、
-- 決済事業者を呼び出すこともある
UPDATE inventory SET available = :new
 WHERE item_id = :item;
COMMIT;
```

ロック自体は正しく、何の間保持するかが誤っています。アプリケーションが `SELECT` と `UPDATE` の間に使う一ミリ秒は、そのまま次の購入者が待つ一ミリ秒です。 `FOR UPDATE` がなければ、さらに悪化します。二人が1を読み、二人とも0を書き、一点が二重に売れます。

確認と減算を一つの文にするのが、修正方法です。

```
UPDATE inventory
   SET available = available - :qty
 WHERE tenant_id = :tenant
   AND item_id = :item
   AND available >= :qty
RETURNING available;
-- 一行返れば確保済み。返らなければ売り切れ。
```

PostgreSQLのREAD COMMITTEDでは、最初の購入者が行を保持している間に次の購入者が到達すると、待ってから、最初の処理がコミットした行に対して `WHERE` 句を再評価します。最初の販売で最後の一点がなくなれば、条件が満た

されず、次の購入者には負数ではなく、行なし、つまり売り切れが返ります。REPEATABLE READまたはSERIALIZABLEでは、次の購入者は代わりに直列化失敗を受け取ります。安全ですが、競合が再試行になるため、この文はREAD COMMITTED向けです。MySQLのInnoDBでも、`UPDATE`はどの分離レベルでも最新のコミット済みの行を読むため、同じ答えになります。制約も追加し、別の箇所の不具合が、存在しない在庫を販売するのではなく、明確に失敗するようにします。

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

一つのデータベースで行う残りの作業は、おおむね効果の大きい順に次のとおりです。

- **トランザクション内に外部処理を入れません。** 決済の呼び出し、HTTPリクエスト、メッセージ配信は、その前か後に行います。
- **負荷の高い行には最後に触れます。** 注文などを先に挿入し、競合する行のロックを、最後の文とコミットの間だけにします。
- **負荷の高いカウンターに楽観的ロックを使いません。** バージョン列は競合を再試行に変え、セール中には再試行が負荷になります。
- **読み取りを別の場所で処理します。** 商品ページの「残り12点」は、一秒遅れのレプリカやキャッシュから取得できます。プライマリーを必要とするのは、販売だけです。
- **在庫に専用のデータベースを与えます。** 販売が、同じディスクを使う商品一覧の処理を待たないようにします。
- **接続をプールします。** 十万人の購入者には、PgBouncer経由の数十の接続を用い、一つのロックを何千ものバックエンドが待たないようにします。

座席は一つずつ異なるため、別の構造で、別のSQL文が必要です。一座席を一行にし、他のトランザクションが保持する行を、後ろで待つ代わりに飛ばします。

```
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;
```

期限切れの取り置きは、そのまま再び空きになるため、正しい答えのために期限切れを掃除する処理は必要ありません。`now()` はデータベースの時計なので、時計がずれているアプリケーションサーバー同士でも、期限切れの時点は一致します。

多くの店舗では、この節だけで修正が完了します。以下のことより先に行ってください。シャーディングは、入るのは簡単でも、戻るには費用のかかる、一方通行の扉だからです。



先に確保し、次に支払う

最後の一点を確保しても、カード入力を放棄した購入者は、まだ購入していません。そのため、販売は一度の書き込みではなく、短いライフサイクルです。

```
AVAILABLE → HELD, 数分間 → SOLD
                         → RELEASED
```

- **取り置きのキーは、試行ではなく要求に結び付けます。** 各購入手続きに冪等性キーを付け、取り置きテーブルに `UNIQUE (tenant_id, idempotency_key)` を設けます。クリックを再試行しても、二点目ではなく、既存の取り置きを返します。PostgreSQLの注意点として、`INSERT ... ON CONFLICT DO NOTHING RETURNING id` は、行がすでにあると何も返しません。そのため、「行なし」は「失敗」ではなく「調べ直す」という意味です。
- **すべての状態遷移に条件を付けます。** 販売確定は、まだ `held` の場合だけ `held` から `sold` へ移し、解除は、その更新が正確に一行を変更した場合だけ在庫を戻します。タイムアウトとキャンセルボタンなどで解除が二度実行されても、在庫を戻すのは一度です。
- **取り置きは数分で期限切れにします。** 通常は五分から十分です。決済が始まったら一度だけ延長し、遅いカード処理によって、支払い中の在庫を失わないようにします。
- **取り置きの時点で購入者ごとの数量を制限します。** 一つのアカウントが、決めかねている間に全在庫を確保しないようにします。

すべての土台にあるのは、設計の中で最も役立つ一行です。

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

各トランザクションの後に成り立ち、一つのクエリで確認できます。成り立たなくなれば、顧客に言われる前に、どこかで超過販売または在庫の消失が起きたと分かります。

一つのデータベースでは足りない場合

多数のテナント、多数の同時販売、複数のリージョンなど、本当に足りない場合もあります。拡張する構造でも、一つのルールを守ります。各商品の在庫を所有するのは、正確に一つの在庫サービスであり、他は決定しません。そのサービスを `(tenant_id, item_id)` でシャーディングし、一商品の書き込みが二つのシャードにまたがらないようにします。最大のテナントには専用のセルを与え、その販売が他のテナントの処理能力を奪わないようにします。同期である必要があるのは予約だけで、その後はすべてメッセージにできます。

シャード内の人気商品には四つの手段があり、それぞれ上記の調整点を使います。

前段にゲートを置く。 Luaスクリプトで減算するRedisカウンターを使い、確認と減算を一つの手順にまとめ、在庫数程度の購入者を通して、他はマイクロ秒単位で断ります。これが最終判断者ではありません。Redisの複製は非同期であり、フェイルオーバー後の新しいプライマリーは、一部の入場を忘れて、少し多く通す場合があります。それでも、データベースの条件付き `UPDATE` が決定し、照合処理がデータベースの数値でゲートを再設定します。

在庫を分ける。 一万点を、百点ずつ百個のバケットに分け、それぞれを別の行またはシャードにして、各購入者を一つへ送れます。一つではなく百個のロックになります。この考え方は古く、1986年のO'Neilのエスクロートランザクション、続くBarbaráとGarcia-Molinaのdemarcation protocol、さらにBalegasらの上限・下限付きカウンターへとつながっています。本当の費用は一つ、終盤にあります。他に在庫があっても購入者のバケットが空になるため、別のバケットを試すか残りを移し、販売の終わり近くでは一つに戻してまとめます。同じ考え方で各リージョンに専用の在庫枠を与えれば、大半の販売で海を越えて問い合わせる必要はありません。

一商品に一つの書き手。 ある商品のすべての要求をKafkaの一つのパーティションに入れ、一つの受信処理が、メモリーに件数を保持し、判断をまとめてコミットします。一つのスレッドで、ロックはありません。LMAXの構造です。要求にはIDを付け、キューは二度配信することがあるため、書き手は判断済みのものを捨てます。

在庫を行にする。 座席や番号付きの限定品など、数が少なく一つずつ異なる場合は、事前に一つずつ行を作り、上記の `SKIP LOCKED` で確保します。または、在庫IDをRedisの集合に保持し、 `SPOP` で一つ取り出します。競合するカウンターはなくなります。



購入手続きの残り

決済、注文、領収書は別のシステムであり、それらをまたぐトランザクションはありません。教科書どおりなら二相コミットですが、ここでは誤った答えです。参加者のどれかが遅い間は止まり、最も負荷の高い経路に通信の往復を加え、そもそも決済事業者は参加しません。局所的な手順をつなぎ、それぞれに取り消しの手順を持たせるSagaを使います。

```
1. 在庫を確保する          取り消し：取り置きを解除する
2. 注文を保留状態にする   取り消し：注文を取り消す
3. 取り置きを延長する      （自動で期限切れになる）
4. 与信を取得する          取り消し：与信を取り消す
5. HELD → SOLDを確定する
6. 決済を確定する
```

最初に与信し、最後に売上を確定すれば、取り置きの期限後に到着した決済は、返金ではなく無効化で済みます。お金はまだ動いていないからです。各手順は、変更と同じトランザクションで、その変更を知らせるメッセージをoutboxテーブルに書き、変更データキャプチャーで配信します。保存と送信の間に失われるものはありません。各受信処理は注文をキーに冪等とします。メッセージは少なくとも一回届き、冪等性によって、実質的に一回になります。

ここに分散ロックを入れる必要はありません。Kleppmannの指摘は、ロックを持つ処理が、ガベージコレクションや仮想マシンの停止により、リース期限を超えて止まってから書き込み続ける可能性があることです。対処には、ストアが確認するフェンシングトークンが必要です。しかし、トークンを確認できるストアなら、 `available >= :qty` も確認でき、最初からロックは不要です。

超過販売せず、その商品の販売を止める

整合性と可用性は、システム単位ではなく、問いごとに選びます。「この一点を販売できるか」には整合性が必要です。在庫の所有者に到達できないときは、その商品の販売を止めます。商品一覧、「残り3点」の表示、注文履歴など、それ以外は、少し古くても提供を続けられます。

最も気付きにくい超過販売は、フェイルオーバーで起きます。非同期複製では、プライマリーが販売を承諾した後、レプリカに届く前に停止し、昇格したレプリカでは販売が失われ、同じ在庫が再び売れます。対処は、承諾する前に、複数のマシンで販売を永続化することです。PostgreSQLの同期スタンバイ（`synchronous_commit` と `synchronous_standby_names`）、またはRaftを基盤とするストアを使います。コミットごとに遅延が加わり、上限の式の分母に当たるロック保持時間が、その分長くなります。まとめて処理することとゲートで、その費用を取り戻します。停止の範囲も限定します。シャードの障害では、その上の商品だけを止め、店舗全体は止めません。

検証する

私たちの説明も含め、信じるだけで済ませるべきではありません。

- **不変条件を監視します。** `available + held + sold = total` を定期実行し、グラフにし、ずれがあれば緊急通知します。超過販売ゼロは、期待ではなく、通知を伴うサービス水準目標です。
- **意図して競合させます。** 百点の在庫に一万人が同時購入するテストを用意し、正確に百点が売れ、負数が一つもないことを検証します。
- **競合の最中に壊します。** Jepsenのように、プライマリーを停止し、ネットワークを切り、プロセスを一時停止し、その後で不変条件を確認します。
- **拒否する経路の負荷をテストします。** トラフィックの99%が売り切れでも、その経路を高速に保ち、エッジで吸収する必要があります。
- **指標を一つのダッシュボードに置きます。** ロック待ち、期限切れの取り置き、ゲートとデータベースの件数の差を表示します。

進める順序

- まず測定します。 `pg_stat_statements` とロック待ちから、そもそも負荷の高い行が原因かどうかが分かります。

- 書き込みを、一つの条件付きSQL文にします。
- 待合室、ゲート、キャッシュした「売り切れ」で、負荷を減らします。
- 在庫に専用のサービスと所有者を与えます。
- 最初の四つで足りない場合だけ、`(tenant_id, item_id)` でシャーディングします。
- 予定されているセールに向け、バケットまたは単一の書き手を使う人気商品モードを追加します。
- リージョンは最後にし、在庫枠を設けます。

次に同じ課題に取り組む方へ

- **シャーディングが答えるのは、別の問いです。** 多数の行を分散しますが、超過販売の問題は一行です。構築する前に、どちらの問題かを見極めます。
- **確認と減算を一つの文にします。** 正常な販売と超過販売の差の大半は、数値を読んでから書くまでの時間です。
- **トラフィックの大半は、不可という回答です。** 拒否を、システムが最も安くできることにします。
- **超過販売する前に止めます。** 三十秒販売を止めた商品には、謝罪が必要です。二重に売った商品には、返金、顧客対応の待ち行列、忘れない顧客が生まれます。
- **不変条件を書き出し、監視します。** 一行だけですが、レビュー上ではなく本番で設計が機能していると分かる、唯一のものです。

出典

- PostgreSQL：Read Committedの分離レベル
- PostgreSQL：`FOR UPDATE` と `SKIP LOCKED`
- PostgreSQL：同期複製
- Bailisほか『Coordination Avoidance in Database Systems』
- Balegasほか『Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants』
- Martin Kleppmann：分散ロックの方法
- Martin Fowler：LMAXのアーキテクチャ
- DynamoDB：パーティションキーの設計
- DynamoDB：パーティションとデータ分散
- microservices.io：Saga
- microservices.io：トランザクショナルoutbox
- Redis：複製
- Jepsen

すべてのブログ記事

変更の前に測る

代表的なデータセット、そこから答えを得たい問い、現在のボトルネックをお知らせください。品質、処理量、復旧の測り方を合意できます。

料金を見る お問い合わせ →

観測したことを相談する

ご自身のプロジェクトに即して「最後の一枚を、一度だけ販売する」についてご相談ください。何が観測されたか、どのような結果が役立つかをお聞かせください。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

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

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/notes/selling-limited-stock-without-overselling/