Engineer

シャーディングの前にデータベースを最適化する

2026年10月14日

負荷でデータベースが遅くなると、最初に出る提案は、たいていマシンを増やすことです。それが正しい場合もあります。しかし、遅い部分は、誰もまだ読んでいない少数のSQL文であることのほうが多く、データを複数のサーバーに分散しても、それぞれの文の費用は減らず、多くのハードウェアに広がるだけです。解決不能に見える問題が、一つのクエリプランを読むだけで、きれいに解決することもあります。そして、すでにあるデータベースを誰も読まずに済むように、水平スケーリングが選ばれることもあります。

何かを分割する前に、私たちは次の順で進めます。調べ、効率化し、プライマリーから読み取りを外し、負荷の集中箇所の周囲のテーブルをパーティション分割し、一台が有益な処理で埋まったときだけシャーディングします。全購入者が一つの行へ書き込むフラッシュセールは、最後の一枚を、一度だけ販売するで詳しく扱っています。このノートは、その周囲の方法を説明し、最後の節では逆の提案、すなわち、すべてのサーバーから販売し、二重に売った分は謝罪する方法を検討します。

技術的な取り組みを自社のシステムに生かす

「シャーディングの前にデータベースを最適化する」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 →

まず調べる

知るべきことの大半は、PostgreSQLがすでに記録しています。変更する前に、四つの問いを投げかけます。

合計で最も費用のかかるSQL文は何か。 `shared_preload_libraries` で読み込む拡張機能 `pg_stat_statements` は、正規化した各SQL文と、その呼び出し回数、時間を保持します。平均ではなく合計で並べます。一時間に百万回実行する2 msのクエリは、二回実行する二秒のレポートより費用がかかります。

```
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;
```

最上位のSQL文は、実際に何をしているか。 `EXPLAIN (ANALYZE, BUFFERS)` はSQL文を実行し、選んだプラン、各段階で予想した行数と実際に返した行数、読み取ったページ数を示します。十行を返すために百万行を走査していたり、見積もりが百倍ずれていたりすれば、それだけでほぼ原因が分かります。 `ANALYZE` は説明対象を実行するため、書き込みは `BEGIN` と `ROLLBACK` で囲みます。

誰が、誰を待っているか。

```
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';
```

負荷が高い時間帯にここに行があれば、トランザクションはハードウェアではなく互いを待っています。マシンを高性能にしても、一つの共有行に対する更新を一つずつ順に実行する必要はなくなりません。アプリケーションサーバーを増やすと、書き込みを行うクライアントが、この待ち行列に加わることがあります。

サーバーは仕事をしているか、切り替えに追われているか。 `pg_stat_activity` でアクティブなセッション数を数え、コア数と比較します。アクティブな数がコア数の何倍にもなると、サーバーは切り替えに時間を費やします。ハードウェアを変える前に、接続プールで対処できます。次に、 `pg_stat_user_tables` の `n_dead_tup` を読みます。生きた行より不要な行が多いテーブルは、自分のごみを走査しています。データベースが小さすぎるのではなく、autovacuumが追い付いていません。

分かったことを書き留めます。その後のすべての手順を評価する基準であり、「シャーディングは必要か」への率直な答えが、すでにそこにある場合も多いからです。



すでにあるものを効率化する

おおむね、労力に対する効果が大きい順です。

問いに合うインデックス。 上記のプランで走査が分かります。頻繁なクエリが絞り込む列にインデックスを付けると、走査を検索に変えられます。部分インデックスには、例えばまだ販売中の在庫など、そのクエリが必要とする行だけを保持します。販売が進むにつれて、大きくなるのではなく小さくなります。

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

すべてのインデックスには、書き込みのたびにも費用がかかります。 `pg_stat_user_indexes` の `idx_scan` がゼロのままのものは削除します。

値の意味を示す型。 人でもモデルでも、急いで書いたスキーマでは、 `SMALLINT` や `BOOLEAN` で足りるところに `TEXT` を使いがちです。 `BOOLEAN` は一バイト、 `SMALLINT` は二バイト、テキストの `'yes'` は四バイトです。また、PostgreSQLは固定長の各列を、その列のアラインメントに合わせて埋めるため、偶然決まった列順にも余分なバイトがかかります。急いで書いた在庫一点の定義です。

```
CREATE TABLE units (
  sold        text,  -- 'yes'または'no'
  id          bigint,
```

```
  reserved    text,  -- 'yes'または'no'
  reserved_at timestamptz,
  item_id     bigint,
  order_id    bigint
);
```

同じ在庫を、値の意味を示す型で、幅の広いものから並べると、こうなります。

```
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
);
```

PostgreSQL 18で測定すると、最初の行は72バイト、二つ目は54バイトで、四分の一小さくなり、テーブルのより多くの部分がメモリーに入ります。型は、そうでなければアプリケーションが防ぐ必要のある `'maybe'` などの値も拒否します。範囲に対しては、 `quantity smallint CHECK (quantity > 0)` のような `CHECK` 制約で同じことができます。

往復を減らし、データのある場所で処理する。 アプリケーションが送るSQL文ごとに、通信の往復と解析の費用がかかります。百行を読み、一行ずつ更新するループは、一つの集合ベースのSQL文で済むところに、百一回の往復を使います。また、空きの数を数えるために商品の全在庫を取得するページは、一つの数字のために何千行も転送します。私たちの原則は、データを見る必要のあるもの、すなわち絞り込み、集計、現在の状態に依存する書き込みをデータベースに置くことです。業務ルールはアプリケーションに残します。システム内で最も増やしにくいCPUは、データベースのCPUだからです。

同じページに収まる更新。 一分間に何度も変わる行は、そのたびに新しい版を残します。インデックス付きの列を変えず、ページに空きがあれば、PostgreSQLは新しい版を同じページに置き、どのインデックスにも触れません。heap-only tuple、すなわちHOT更新です。そのため、頻繁な更新で変える列をインデックスから外し、ページに空きを残します。

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

設定は、それ以後に書くページに適用されるため、行を読み込む前に設定します。上記のテーブルでは、`reserved`、`reserved_at`、`order_id`を設定する2,000回の確保は、すべてHOTでした。一方、部分インデックスが参照する`sold`を設定する2,000回の更新には、HOTが一つもありませんでした。販売は一点につき一度ですが、確保は何度も起こり得ます。稼働中のシステムでは、`pg_stat_user_tables`の`n_tup_hot_upd`と`n_tup_upd`で割合を確認できます。

一行で、誰が先だったかを決める

在庫は、その行にSQL文が最初に到達した購入者が確保し、確保はデータベースの時計で期限切れになります。二つの列で表します。真偽値の`reserved`と、設定した時刻の`reserved_at`です。

```
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;
```

一行返れば、この購入者が確保しています。返らなければ、別の人が先だったか、販売済みです。同じ在庫に対する二つの確保は、行ロックで順番を待ちます。PostgreSQLの標準であるREAD COMMITTEDでは、後の処理が、先の処理でコミットされた行に対して`WHERE`句を再確認するため、正確に一方だけが行を受け取ります。10分より古い確保は空きとみなすため、正しい答えを得るために、別途期限切れの確保を掃除する必要はありません。

このため、真偽値だけでは完全な答えになりません。確保が期限切れになっても`reserved`はtrueのままで、在庫が確保中かどうかは二つの列を合わせて初めて分かります。問い合わせるクエリは、両方を見る必要があります。期限切れの確保を定期的に解除すれば、`reserved`だけを読む人にも実態を示せますが、正しさがその処理を待つことはありません。二つの値の整合性は、制約で維持します。

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

チケットのノートでは、開始時刻ではなく期限の `held_until` を保存しますが、同じ設計です。座席や番号付きの版画のように、購入者が在庫を選ぶ場合は、ここと同じくSQL文でそれを指定します。どの在庫でもよい場合は、そのノートの `SKIP LOCKED` 形式で、他の人が確保していないものを一つ選びます。



プライマリーから読み取りを外す

店舗のトラフィックの大半は、商品ページ、「残り3点」、買い物かご、注文履歴などの読み取りです。どれも、書き込みのために空けておくべき一台であるプライマリーを必要としません。ストリーミングレプリカに読み取りを受け持たせます。シャードの追加が一つのプロジェクトになるのに対し、レプリカの追加は設定で済みます。

注意点は二つあります。レプリカは少し遅れるため、在庫を確保した直後に買い物かごをレプリカから読むと、確保が見えない場合があります。利用者が書き込んだ直後のデータはプライマリーから読むか、書き込み後の数秒間、その利用者をプライマリーへ送ります。また、レプリカ上の長いレポートは、レプリカが再生する変更を妨げると、キャンセルされる場合があります。PostgreSQLの文書では、これをクエリの競合と呼び、その競合とレプリカの遅延を調整する設定を挙げています。

最も安い読み取りは、データベースに届かない読み取りです。商品の全在庫が確保されていれば、期限が切れるか解除されるまで、誰も購入できません。いつ空くかは、データベースが正確に答えられます。

```
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;
```

`free` がゼロなら、 `until` が最も早い確保の期限です。それを有効期限として「全数確保中」をキャッシュし、それより長くは保持しません。その間の訪問者には、データベースではなくキャッシュが販売不可と答えます。 `free` がゼロで `until` が空なら、全在庫が販売済みで、答えは確定です。カードの拒否や買い物かごの放棄により、確保が早く解除された場合は、キャッシュ項目を削除して、すぐに販売を再開します。

キャッシュは、不可としか答えません。可能という答えは常に上記の条件付き `UPDATE` から返します。そのため、古いキャッシュで数秒間販売機会を失うことはあっても、在庫を二重に売ることはありません。チケットのノートの「売り切れ」キャッシュに、有効期限を加えたものです。



負荷の高い行の周囲のテーブルをパーティション分割する

パーティション分割は、月やテナントなどのキーによって、一つの名前の背後でテーブルを複数に分けます。何をしないかも明確にすべきです。一つの競合する行を速くするものではありません。その行は依然として、一台のマシンの一つのパーティション内で、一つのロックの背後にあります。

できるのは、負荷が集中する箇所の周囲のテーブルを小さく保つことです。注文、確保履歴、イベントは増え続けますが、クエリの大半は最近の部分だけを必要とします。

```
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');
```

日付の範囲を指定したクエリは、その中のパーティションだけを読み、今月のインデックスは、メモリーに収まる大きさに保てます。保存期限の管理も、何百万行もの `DELETE` と、その後の肥大化やVACUUMではなく、一つの文になります。

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

主キーにはパーティションキーを含める必要があるため、`(id, created_at)` としています。キーを指定しないクエリはすべてのパーティションを読むため、すでにクエリが絞り込む項目で分割します。



シャーディングが妥当になるとき

シャーディングでも動かせない限界があります。各商品には、担当する所有者が一つ必要です。同じ在庫を二台のサーバーで同時に確保することはできないため、シャーディングでは商品を丸ごと移し、一商品の半分だけを移すことはしません。一つの商品が一台の処理能力を超える場合は、チケットのノートにあるバケットで在庫を小さく分け、それぞれの部分に、やはり正確に一つの所有者を割り当てます。

その条件の下で、次のいずれかが成り立ったときにだけ、シャーディングが妥当と考えます。

- **一台が有益な処理で埋まっている。** SQL文はすでに効率化され、一回の条件付き書き込みにまとまり、トランザクション内の外部処理はなく、更新は同じページに収まっています。それでも、購入する価値のある最大のマシンで、一つではなく多くの商品にわたり、プライマリーのCPUや先行書き込みログが飽和しています。一つの人気商品は、シャードではなくバケットの問題です。
- **データが保守能力を超えている。** 事業に約束した時間内にバックアップを復元できない、閑散時間にインデックスを作成したりテーブルをVACUUMしたりできない、または作業対象のデータがメモリーに収まらない場合です。
- **分離が要件である。** 契約や規制当局によってデータを分離する必要があるテナント、あるいは他の販売とディスクを共有してはいけないテナントです。

費用は、それ以外のすべてに及びます。マイグレーション、バックアップ、復元、レポートでは、別々のシャードを考慮する必要があります。シャード間の結合をアプリケーションのコードに移す場合もあり、シャードキーが分からないクエリは、すべてのシャードに問い合わせる必要が生じることがあります。ここまでの手順は、この負担を避けるためにあります。一度負うと、元に戻すのは困難です。

ロックより謝罪のほうが安い場合

逆の進め方にも、真剣に検討すべき主張があります。各サーバーが見える在庫から販売し、ときどき一点が二人に売れることを受け入れ、顧客対応が一方に謝罪して返金します。業務ロジックを結果整合性にし、競合しない99.9%のリクエストに合わせて構築し、非同期のループで後から競合を見つけて埋め合わせます。その代わり、店は「再試行してください」と言わずに済みます。HellandとCampbellは、_Building on Quicksand_で、この考え方を最も強く論じています。バックアップが受け取る前に処理を受け付けたと答えるシステムでは、すべての約束が確率的になり、企業は昔から守れなかった約束を謝罪で処理してきた、というものです。

その大半に賛成します。違うのは、どこに適用するかです。

店舗の大半は、結果整合性であるべきです。 商品一覧、検索インデックス、「残り3点」の表示、買い物かご、おすすめ、メール、倉庫向けの出力、レポートは、すべて数秒遅れてもよく、これらの処理をプライマリーの応答待ちにしても、速さを失うだけで得るものはありません。Dynamoは、まさにこのためにつくられました。Amazonの買い物かごは、マシン障害中も「カートに追加」を忘れず拒否せず、その代償として、マージにより購入者が削除した商品が戻ることがあります。確保後の購入手続きは、チケットのノートでもすでに非同期です。局所的な手順をつなぐSaga、少なくとも一回配信するメッセージ、注文をキーとする冪等な受信処理を使います。Webhookは失われるため、決済事業者とやり取りする設計には、照合処理のループが必要です。

確保だけは、この考え方が成り立ちません。 「まれ」という説明は、競合がまれだと仮定しています。しかしフラッシュセールでは、競合こそが販売そのものです。自分の見える状態から販売するサーバーは、他のサーバーの販売結果を聞くまで売り続けます。そのため、超過販売数は、おおよそ試行頻度とレプリケーション遅延の積になります。単純なシミュレーションでは、二つのリージョンが毎秒2,000回の試行を分け合い、それぞれ自分の件数から販売し、200ミリ秒後に相手の販売を知る設定にしました。10,000点の販売を五回実行すると、各回

で191点から205点の、存在しない在庫が売れました。計算では200点と予測します。試行頻度、遅延、他のリージョンで発生する販売の割合を掛けたものです。同じモデルで商品ごとに一つの所有者を置くと、正確に在庫数だけ売れました。これは、障害の有無にかかわらず、売り切れるたびに発生し、最も多くの購入者が見ている最高潮の瞬間に表面化します。

得られる稼働率は、聞こえるほど大きくありません。 GilbertとLynchは、結果整合性で行う交換を証明しました。ネットワークが分断されている間、応答することと整合性を保つことは、同時にできません。AbadiのPACELCは、もう一方も加えます。分断がないときは、遅延と整合性の間の選択になります。商品ごとに一つの所有者を置き、効率的なSQL文を使う方法は、すでに高速です。どちらの設計も100%には達しません。決済事業者、DNS、それらの間のネットワークにも障害があります。厳密な設計が諦める範囲は、限定的です。ある商品の在庫を持つ所有者に到達できなければ、フェイルオーバーにかかる数秒、その商品の販売が止まります。チケットのノートの販売停止の節では、停止を障害マシン上の商品に限定します。他の商品の販売は続きます。

謝罪は返金だけではありません。 返金は安い部分です。購入者は、買えたと言われ、すでに列車を予約し、休暇を取り、友人に知らせているかもしれません。一秒早く「売り切れ」と聞くのは残念なことです。確定した注文を取り消されたと聞くのは、約束を破られたということです。顧客対応の人員は通常の日に合わせていますが、超過販売は一年で最も忙しい一分間に集中します。確定した注文を取り消す計画を立てる前に、注文確認が店にどのような義務を生むか、法律の専門家に確認してください。

非同期処理を残し、承諾の前に移します。 この提案が正しいのは、購入者がロックを待つ必要はないという点です。すべてのクリックを、商品ごとに一つのパーティションを持つ、永続化して複製されたログに受け付け、回答を準備中と伝えます。クリックの受け付けはログだけに依存し、三台に複製したログなら、一台が故障しても書き込みを続けられます。商品ごとの一つの所有者が、ログを順にメモリーの速さで読んで判断し、数秒以内にメッセージで答えを返します。所有者がフェイルオーバー中なら、リクエストは拒否されたり二重に売られたりせず、ログで待ちます。承諾は取り置きを意味し、決済はチケットのノートの購入手続きと同様にその後に行うため、確保できなかった在庫に誰も支払いません。照合は引き続き行いますが、約束の後ではなく、その前に行います。

件数ではなく、在庫を分けます。 リージョン同士で問い合わせずに販売する必要があれば、各リージョンに在庫枠を与え、その枠から独立して販売させます。枠がなくなったリージョンは追加を要求し、販売の終盤には、一つの所有者に枠を戻してまとめます。Bailisらは、通信なしでサーバーが維持できるルールを分

析しており、在庫をゼロ未満にしないルールは、その一つではありません。在庫枠は通信を不要にはしません。頻繁な処理経路から、まれに枠を補充する時点へ移します。これが、不変条件を保つ結果整合性の設計です。唯一の費用は終盤にあり、別のリージョンには在庫があるのに、一方のリージョンで購入者を断る場合があります。

超過販売が適切な方針になるとき。 受注印刷、仕入れ先にまだ在庫があるTシャツ、ライセンスキーなど、追加で用意できる商品もあります。その場合、在庫数を超えた販売は、正直な納期を伴う取り寄せ注文であり、結果整合性が適切です。航空会社は意図的に過剰予約します。現れない人の割合を把握し、リスクを価格に反映し、事前に定めたルールで補償します。これが判断基準です。超過販売を許せるのは、事業側が販売前に、その規模と救済策を選んだ場合です。たまたま最後にクリックした人にしわ寄せが行くなら、不具合です。

謝罪を選ぶなら、決済と同じように構築します。 提案にあるループは返金処理であり、お金を動かすものに必要な要素を備えなければなりません。

- **すべての販売にIDを。** 再試行でも同じ販売として扱い、追記専用の販売台帳を設けます。在庫数は台帳から算出し、直接編集しません。
- **誰が在庫を得るかのルール。** 得られなかった購入者にも説明できるものにします。順序はログで決め、数ミリ秒の時計のずれで返金対象が決まる、別々のサーバーの時刻は使いません。
- **冪等で、記録される補償処理。** キャンセル、決済の無効化または返金、購入者への通知です。Sagaの取り消し手順として、機械で実行します。
- **期限。** 出荷前に、そして購入者が気付く前に、すべての超過販売を見つける必要があります。
- **測定する予算。** 販売ごとの超過販売数を、事業側が承認した数と比較します。不変条件と同じダッシュボードに並べ、予算を使い切ったら通知します。

私たちの答えは、確保の周囲は結果整合性、確保そのものは厳密、購入者への回答は非同期、という設計です。通常の経路に合わせて設計しますが、フラッシュセールでは競合が通常の経路です。何かと引き換えに少量の超過販売を認めるなら、事業側が選んだ数と、販売前に定めた救済策を用意し、測定します。

次に同じ課題に取り組む方へ

- **買い足す前に読みます。** `pg_stat_statements` と一つの `EXPLAIN` は、新しいサーバーより多くの容量上の疑問に答えます。

- **型でデータの意味を示します。** 真偽値を真偽値として保存すると、小さくなり、不正な値を拒否できます。
- **データを見る必要のある処理だけをデータベースに送ります。** 絞り込みと集計はそこに置き、業務ルールは置きません。
- **キャッシュは不可と答え、可能と答えるのはデータベースだけです。** 古いキャッシュで失うのは数秒であり、在庫ではありません。
- **パーティション分割は整理です。** 負荷の高い行の周囲のテーブルを小さく保ちますが、その行自体には何もしません。
- **一台が有益な処理で埋まったらシャーディングします。** 待ち時間で埋まったときではありません。
- **承諾した後ではなく、前に照合します。** 確保の周囲はすべて結果整合性にできます。確保そのものをそうすると、毎回の販売で、試行頻度と遅延の積に応じた超過販売が起きます。

出典

- PostgreSQL：pg_stat_statements
- PostgreSQL：EXPLAINの使い方
- PostgreSQL：累積統計システム
- PostgreSQL：数値型
- PostgreSQL：文字型
- PostgreSQL：真偽値型
- PostgreSQL：部分インデックス
- PostgreSQL：heap-only tuple（HOT）
- PostgreSQL：テーブルのパーティション分割
- PostgreSQL：ホットスタンバイとクエリの競合
- Pat Helland、Dave Campbell『Building on Quicksand』
- Seth Gilbert、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ほか『Dynamo: Amazon’s Highly Available Key-value Store』
- Hector Garcia-Molina、Kenneth Salem『Sagas』
- Bailisほか『Coordination Avoidance in Database Systems』

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「シャーディングの前にデータベースを最適化する」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/optimise-before-you-shard/