Engineer

トランザクション

データベースの変更を、すべて成功かすべて失敗として扱う単位

どのようなものか

トランザクションは、データベースが一つの単位として適用する変更のまとまりです。すべてが反映されるか、一つも反映されないかのどちらかになります。支払いにも使う言葉と区別する場合は、データベーストランザクションと呼びます。PostgreSQLのチュートリアルは、「複数の手順を一つの、すべて成功かすべて失敗かの操作にまとめる」ものと説明し、途中の状態はほかの利用者には見えません。

PostgreSQLでは `BEGIN` で始まり、変更を永続化して他のセッションから見えるようにする `COMMIT` か、変更を破棄する `ROLLBACK` で終わります。一文だけを送る場合も、トランザクション内で動きます。PostgreSQLが暗黙の `BEGIN` で包み、成功すれば暗黙の `COMMIT` を行うためです。長いトランザクションでは `SAVEPOINT` を設定し、後で `ROLLBACK TO` で戻れます。セーブポイント以降の作業を捨て、それより前は残せます。

保証はACIDに要約されます。原子性とは、一単位としてすべてを完了するか、何も反映しないことです。途中までの結果がクラッシュ後に残ることもありません。一貫性は、データに対して宣言された制約がコミット時に満たされ、制約に違反した状態を残すトランザクションは代わりにロールバックされることです。分離性は、コミットするまで変更が並行トランザクションに見えないことです。永続性は、コミット済みの変更がサーバーのクラッシュ後も残ることです。

分離レベルには段階があり、SQL標準はRead Uncommitted、Read Committed、Repeatable Read、Serializableの四つを定義しています。PostgreSQLでは、特に指定しなければRead Committedで開始し、Read UncommittedもRead Committedとまったく同じ動作をします。Read Committedでは、各文はその文の開始前にコミットされたデータを見るため、同じトランザクション内の二つのクエリで異なる数値を見ることがあります。Repeatable Readでは、トランザク

ション制御文以外の最初の文を実行した時点の、一つのスナップショットを全体で使います。Serializableはさらに進め、並行トランザクションの結果を、順番に実行した場合と同じにします。

概念を判断に生かす

「トランザクションとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 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 →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 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 →

なぜ重要なのか

記録する価値のある操作のほとんどは、複数の行に触れます。注文は、あるテーブルへの一行、別のテーブルでの在庫減算、台帳への記録です。トランザクションがなければ、二番目と三番目の文の間にクラッシュ、タイムアウト、制約違反が起きると、在庫を引いていない注文や、存在しない注文のために引かれた在庫が残ります。後で誰かがレポートで見つけ、手作業で修正します。トランザクションがあれば、その中途半端な状態を誰かが見ることはなく、それを保証するのはアプリケーションではなくデータベースです。

二つ目の理由は並行処理です。同時に届いた二つの要求は並行して動き、何が見えるかは分離レベル、どちらが待つかはロックが決めます。チームが驚きやすいのは、コードを `BEGIN` と `COMMIT` で囲むだけでは、二つの要求が同じ残高を読み、両方が使うことを防げない点です。Read Committedでは各文が開始時点のコミット済みデータを読むため、一文で確認して次の文で書く間に、別セッションのコミットが入ります。確認のために読み取る前に行をロックするか、確認そのものを書き込みに含める必要があります。

トランザクションには、時間というコストもかかります。 `UPDATE` 、 `DELETE` 、 `SELECT ... FOR UPDATE` が取った行ロックは、通常トランザクション終了まで続きます。ただし、セーブポイントへ戻ると、その後に取得したロックは解放されます。競合するロックを求めるセッションは、タイムアウト、キャンセル、デッドロックの解消で止まらない限り待ちます。開いたトランザクションは、まだ見る可能性のある古い行バージョンを `VACUUM` が削除することも妨げるため、何時間も開いたままならテーブルが膨らみます。設計上の問いは、使うかどうかだけではありません。境界をどこに置き、その内側の作業をどれだけ減らすかです。

また、通常のローカルなデータベーストランザクションだけでは、外部への作用は含められません。決済事業者への呼び出し、メール、別のキューへ送ったメッセージは、データベースをロールバックしても取り消せません。ほかのシステムとの調整には追加の設計が必要で、外側を `BEGIN` で囲むだけでは不十分です。何を境界の内側に置き、コミットを外へどう伝えるかが、設計作業の大半を占めます。

実際の活用例

ポイント制度の会員が、映画の引換券に250ポイントを使います。三つの処理をまとめて行う必要があります。ウォレットからポイントを引き、台帳に理由を残し、交換の記録を作成します。一つのトランザクションとして書くと、次のようになります。

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- no row back: not enough points, so ROLLBACK and stop here

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'redeemed: cinema voucher');

INSERT INTO redemption (member_id, reward_id)
VALUES (42, 7);

COMMIT;
```

`UPDATE` は会員42のウォレットの行をロックします。 `COMMIT` までは、ほかのセッションが読む残高は以前のままです。行ロックは読み手を止めず、未コミットの変更は見えないためです。同じ会員が別のタブから少し遅れて交換すると、その要求は自分の `UPDATE` に到達して待ちます。最初がコミットされると、Read CommittedのPostgreSQLは、現在の行に対して二つ目の `WHERE` 条件を再評価します。残りが100ポイントしかなければ、条件を満たさず行は返らないので、二つ目の要求は何も書かずにロールバックします。反対に、最後の `INSERT` が、たとえば報酬7の取り下げに伴う外部キー制約違反で失敗した場合も、クライアントがロールバックすれば、ポイントは元のままです。

リーグとバッジを備えたポイントウォレットは、この形を取っています。ウォレットを変更する際は、読み取る前に必ずその行をロックし、二つ目の要求が競り合うのではなく、一つ目の完了を待ちます。ウォレットがほかのデータと同じデータベース内にあるため、残高と購入したものは、まとめてコミットするか、どちらもコミットしないかになります。ページは代償も明記しています。行ロックはより時間のかかる方法であり、会員は同じポイントを二度使う代わりに、自分の順番を待ちます。

最後のチケットを一度だけ売るノートは、同じ手段をフラッシュセールに使っています。在庫確認と減算を一つの条件付き `UPDATE` にまとめるのが対策です。`SELECT` と `UPDATE` の間でアプリケーションが判断するあいだもロックを保持すると、その一ミリ秒一ミリ秒が次の購入者の待ち時間になるためです。残る作業の一覧で最初に挙げるのは、外部の処理をすべてトランザクションから出すことです。決済呼び出し、HTTP要求、メッセージの発行は、その前か後に置きます。

問題が起きるところ

トランザクションを開いたままセッションが何もせず、後続がすべて待つ。 アプリケーションがトランザクションを開いて行を更新した後、遅いAPI、利用者の次のクリック、あるいは `COMMIT` を送る前に落ちたワーカーを待っています。ロックは残り、タイムアウトが無効なら、PostgreSQLは競合ロックを求めるセッションを、デッドロックを検出するか待機がキャンセルされるまで無期限に待たせます。公式ドキュメントは、利用者の入力を待つ間トランザクションを開くのは良くないとしています。 `pg_stat_activity` には状態 `idle in transaction` で表示され、 `xact_start` に開始時刻が出ます。

```
SELECT pid, now() - xact_start AS open_for, state, query
FROM pg_stat_activity
WHERE state LIKE 'idle in transaction%'
ORDER BY xact_start;
```

遅い処理は外へ移し、忘れられたトランザクションが何時間もロックを持たないよう、サーバーに上限を設定します。 `idle_in_transaction_session_timeout` は、開いたトランザクション内でアイドル状態が指定値より長く続いたセッションを終了します。既定値0では無効なので、たとえばアプリケーションのロールに `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` を設定します。

二つの要求が同じ残高を使う。 コードが一文で値を読み、アプリケーションで判断し、別の文で結果を書きます。二つのセッションが同じ数値を読み、両方が確認を通って書くと、一方の変更が他方を黙って上書きします。Read Committedでは `BEGIN` で囲んでも防げません。上の例のように `UPDATE` の条件へ確認を組み込むか、読む前に `SELECT ... FOR UPDATE` で行をロックして、二つ目を一つ目の終了まで待たせます。

Repeatable ReadやSerializableで、負荷時にエラーが出る。 この水準では、約束を破るトランザクションをPostgreSQLが拒否し、SQLSTATE `40001` の直列化失敗と、`could not serialize access due to concurrent update` などのメッセージを返します。これは設計どおりで、不具合ではありません。公式ドキュメントも、これらの水準を使うアプリケーションには再試行への備えを要求しています。`40001` を捕捉したら、読み取りも含めてトランザクション全体を `BEGIN` からやり直します。以前の判断が、その後変わったデータに基づいているためです。失敗した文だけを繰り返すと、同じ誤りになります。

二つのトランザクションがデッドロックする。 一方は注文1、次に注文2をロックし、他方は注文2、次に注文1をロックすると、互いに相手を待ちます。PostgreSQLは循環を検出し、一方を中止して他方を完了させます。どちらを選ぶかは予測しにくいものです。公式の対策は、すべてのコード経路で同じ順にロックすることです。たとえば主キーの昇順で行を更新し、デッドロックで中止されたトランザクションを再試行します。

取り消された注文のメールだけが届く。 アプリケーションがトランザクション内から確認メールを送ったりイベントを発行したりした後で、後続の文が失敗します。ロールバックは注文を消しますが、メールは送信前に戻せません。変更を説明するメッセージを、その変更と同じトランザクション内のoutboxテーブルへ記録し、コミット後に別のプロセスが送るようにします。そうすれば、メッセージの有無が、注文の有無と一致します。

参考資料

- トランザクション — PostgreSQLチュートリアルの章。銀行振込を例に、すべて成功かすべて失敗かの動作、暗黙のトランザクション、セーブポイントを説明しています。
- トランザクションの分離 — 標準の四水準、PostgreSQLで各水準から見えるもの、再試行が必要な直列化失敗です。
- 明示的なロック — 行とテーブルのロック、保持期間、デッドロックの検出と回避を扱います。

- クライアント接続の既定値 — `default_transaction_isolation`、`idle_in_transaction_session_timeout`、`lock_timeout` など、セッションの上限設定です。
- 定期的なVACUUM — どのトランザクションからも見えなくなるまで古い行バージョンが残る理由と、長時間開いたトランザクションが整理へ与える影響を説明しています。

用語集

実際の制約に合わせて選ぶ

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 →

用語を実務に結び付ける

「トランザクションとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

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/glossary/transaction/