Engineer

Transaktion

Datenbankänderungen, die gemeinsam gelingen oder scheitern

Was es ist

Eine Transaktion ist eine Gruppe von Änderungen, die eine Datenbank als Einheit anwendet: Entweder wirken alle oder keine. Zur Abgrenzung von einer Zahlung, die ebenfalls als Transaktion bezeichnet wird, heißt es Datenbanktransaktion. Das PostgreSQL-Tutorial beschreibt sie als etwas, das „mehrere Schritte zu einer einzigen Alles-oder-nichts-Operation bündelt“. Zwischenzustände sind für niemand anderen sichtbar.

In PostgreSQL beginnt eine Transaktion mit `BEGIN` und endet mit `COMMIT`, das ihre Änderungen dauerhaft und für andere Sitzungen sichtbar macht, oder mit `ROLLBACK`, das sie verwirft. Auch eine einzeln gesendete Anweisung läuft in einer Transaktion: PostgreSQL umgibt sie mit einem impliziten `BEGIN` und bei Erfolg mit einem impliziten `COMMIT`. Eine lange Transaktion kann einen `SAVEPOINT` setzen und später mit `ROLLBACK TO` dorthin zurückkehren. Die seitdem geleistete Arbeit wird verworfen; alles davor bleibt erhalten.

Die Garantien einer Transaktion werden als ACID zusammengefasst. Atomarität bedeutet, dass eine Transaktion als Einheit oder gar nicht abgeschlossen wird und kein halbfertiges Ergebnis einen Absturz überlebt. Konsistenz heißt, dass die für die Daten definierten Bedingungen nach dem Commit erfüllt sind. Eine Transaktion, die eine verletzen würde, wird stattdessen zurückgerollt. Isolation bedeutet, dass ihre Änderungen für gleichzeitig laufende Transaktionen bis zum Commit unsichtbar bleiben. Dauerhaftigkeit heißt, dass eine bestätigte Änderung einen Serverabsturz übersteht.

Isolation hat Stufen. Der SQL-Standard definiert vier: Read Uncommitted, Read Committed, Repeatable Read und Serializable. Ohne andere Vorgabe beginnt PostgreSQL jede Transaktion auf Read Committed. Sein Read Uncommitted verhält sich genauso wie Read Committed. Auf Read Committed sieht jede

Anweisung die vor ihrem Beginn bestätigten Daten. Zwei Abfragen innerhalb derselben Transaktion können deshalb unterschiedliche Zahlen sehen. Auf Repeatable Read arbeitet die ganze Transaktion mit einem Snapshot, der beim Ausführen ihrer ersten Anweisung entsteht, die keine Anweisung zur Transaktionssteuerung ist. Serializable geht weiter: Gleichzeitige Transaktionen erzeugen dasselbe Ergebnis wie bei einer Ausführung nacheinander.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist eine Transaktion?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Eine Struktur, die zur tatsächlichen Arbeitsweise Ihres Unternehmens passt, damit sich auch spätere Fragen noch beantworten lassen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

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

Warum es wichtig ist

Fast jede aufzeichnungswürdige Aktion betrifft mehr als eine Zeile. Eine Bestellung ist eine Zeile in einer Tabelle, eine Verringerung in einer anderen und ein Eintrag im Journal. Ohne Transaktion hinterlassen ein Absturz, eine Zeitüberschreitung oder eine verletzte Bedingung zwischen der zweiten und dritten Anweisung eine Bestellung ohne abgezogenen Bestand oder einen abgezogenen Bestand ohne Bestellung. Jemand findet es später in einem Bericht und korrigiert es von Hand. Mit einer Transaktion ist dieser halbfertige Zustand für niemanden sichtbar. Das garantiert die Datenbank, nicht die Anwendung.

Der zweite Grund ist Nebenläufigkeit. Zwei gleichzeitig eintreffende Anfragen laufen nebeneinander. Die Isolationsstufe entscheidet, was jede sieht; Sperren entscheiden, wer wartet. Hier erleben Teams die meisten Überraschungen: Code in `BEGIN` und `COMMIT` einzuschließen verhindert allein nicht, dass zwei Anfragen denselben Kontostand lesen und beide ausgeben. Auf Read Committed liest jede Anweisung, was bei ihrem Beginn bestätigt war. Zwischen einer Prüfung in einer Anweisung und einem Schreibvorgang in der nächsten kann deshalb ein Commit einer anderen Sitzung liegen. Die Prüfung muss vorher die Zeile sperren oder Teil des Schreibvorgangs werden.

Transaktionen haben auch Kosten, bezahlt in Zeit. Zeilensperren von `UPDATE`, `DELETE` und `SELECT ... FOR UPDATE` bleiben normalerweise bis zum Transaktionsende bestehen. Ein Rollback zu einem Savepoint gibt die danach erworbenen Sperren frei. Eine Sitzung, die eine dazu im Konflikt stehende Sperre benötigt, muss warten, sofern nicht der Ablauf einer Zeitgrenze, ein Abbruch oder eine Deadlock-Auflösung das Warten beendet. Eine offene Transaktion verhindert außerdem, dass `VACUUM` alte Zeilenversionen entfernt, die sie möglicherweise noch lesen muss. Eine stundenlang offene Transaktion lässt Tabellen deshalb aufblähen. Beim Entwurf geht es nie nur darum, ob eine Transaktion verwendet wird, sondern wo ihre Grenzen liegen und wie wenig dazwischen geschieht.

Schließlich umfasst eine gewöhnliche lokale Datenbanktransaktion nicht von selbst externe Wirkungen. Ein Aufruf bei einem Zahlungsanbieter, eine E-Mail oder eine in eine separate Warteschlange gestellte Nachricht wird durch ein Zurückrollen der Datenbank nicht rückgängig gemacht. Die Abstimmung mit anderen Systemen braucht einen zusätzlichen Entwurf; ein umschließendes `BEGIN` genügt nicht. Was innerhalb der Grenze liegt und wie die Außenwelt von einem Commit erfährt, macht den Großteil des Transaktionsentwurfs aus.

Wie es in der Praxis aussieht

Ein Mitglied eines Treueprogramms gibt 250 Punkte für einen Kinogutschein aus. Drei Dinge müssen gemeinsam geschehen: Die Punkte werden vom Wallet abgezogen, das Journal hält den Grund fest, und die Einlösung wird angelegt. Als eine Transaktion sieht das so aus:

```
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;
```

Das `UPDATE` nimmt eine Zeilensperre auf dem Wallet von Mitglied 42. Bis zum `COMMIT` sieht jede andere Sitzung, die das Wallet liest, weiter den alten Stand: Zeilensperren blockieren Leser nicht, und unbestätigte Änderungen sind unsichtbar. Eine zweite Einlösung für dasselbe Mitglied, kurz darauf von einem anderen Tab gesendet, erreicht ihr eigenes `UPDATE` und wartet. Sobald die erste Transaktion bestätigt ist, wertet PostgreSQL auf Read Committed die `WHERE`-Bedingung der zweiten erneut gegen den nun vorliegenden Zeilenstand aus. Sind nur noch 100 Punkte übrig, scheitert die Bedingung. Es kommt keine Zeile zurück, und die zweite Anfrage rollt zurück, ohne etwas geschrieben zu haben. Scheitert stattdessen das letzte `INSERT`, etwa weil Prämie 7 zurückgezogen wurde und ein Fremdschlüssel sie ablehnt, rollt der Client zurück. Die Punkte stehen dann genau dort, wo sie vorher waren.

Das ist die Form des Treue-Wallets mit Ligen und Abzeichen. Jede Änderung an einem Wallet nimmt dort vor dem Lesen eine Zeilensperre. Eine zweite Anfrage wartet deshalb auf den Abschluss der ersten, statt mit ihr um die Wette zu laufen. Weil das Wallet in derselben Datenbank wie alles andere liegt, werden der Kontostand und das damit Bezahlte gemeinsam bestätigt oder gar nicht. Die Seite nennt den Preis offen: Zeilensperren sind die langsamere Antwort. Mitglieder warten der Reihe nach, statt dieselben Punkte zweimal auszugeben.

Die Notiz darüber, das letzte Ticket genau einmal zu verkaufen, setzt dasselbe Werkzeug bei einem Blitzverkauf ein. Ihre Lösung verbindet Bestandsprüfung und Verringerung in einem bedingten `UPDATE`. Eine Sperre, die während der Überlegungen der Anwendung zwischen `SELECT` und `UPDATE` bestehen bleibt, lässt den nächsten Käufer jede einzelne Millisekunde warten. An erster Stelle der übrigen Arbeiten steht, alles Externe aus der Transaktion herauszuhalten: Ein Zahlungsaufruf, eine HTTP-Anfrage oder eine veröffentlichte Nachricht gehört davor oder danach.

Wo es schiefgeht

Eine Sitzung bleibt in einer Transaktion untätig, und alles dahinter wartet. Eine Anwendung öffnet eine Transaktion, aktualisiert eine Zeile und wartet dann auf eine langsame API, den nächsten Klick eines Nutzers oder einen abgestürzten Worker, der nie `COMMIT` sendet. Die Sperren bleiben. Sind Timeouts deaktiviert, lässt PostgreSQL eine Sitzung, die eine kollidierende Sperre anfordert, unbegrenzt warten, sofern es keinen Deadlock erkennt und die Wartezeit nicht abgebrochen wird. Die Dokumentation bezeichnet es als schlechte Idee, eine

Transaktion beim Warten auf Nutzereingaben offen zu halten. `pg_stat_activity` zeigt solche Sitzungen im Zustand `idle in transaction` ; `xact_start` gibt ihren Beginn an:

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

Die langsame Arbeit aus der Transaktion verlagern und dem Server eine Grenze geben, damit eine vergessene Transaktion keine Sperren stundenlang halten kann. `idle_in_transaction_session_timeout` beendet jede Sitzung, die länger als sein Wert innerhalb einer offenen Transaktion untätig bleibt. Standardmäßig ist der Wert 0 und die Funktion damit deaktiviert. Der Parameter muss also gesetzt werden, etwa für die Rolle der Anwendung mit `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

Zwei Anfragen geben denselben Kontostand aus. Der Code liest einen Wert in einer Anweisung, entscheidet in der Anwendung und schreibt das Ergebnis mit einer anderen. Zwei Sitzungen lesen dieselbe Zahl, bestehen beide die Prüfung und schreiben beide. Eine Änderung überschreibt dabei unbemerkt die andere. Ein umschließendes `BEGIN` verhindert das auf Read Committed nicht. Die Prüfung wie im obigen Beispiel als Bedingung im `UPDATE` zum Teil des Schreibvorgangs machen oder die Zeile vor dem Lesen mit `SELECT ... FOR UPDATE` sperren. Das lässt die zweite Sitzung auf die erste warten.

Unter Last erscheinen Fehler bei Repeatable Read oder Serializable. Auf diesen Stufen lehnt PostgreSQL eine Transaktion ab, die deren Zusicherung verletzen würde, und meldet einen Serialisierungsfehler, SQLSTATE `40001` , etwa `could not serialize access due to concurrent update` . So funktioniert der Entwurf; es ist kein Fehler im Programm. Laut Dokumentation müssen Anwendungen auf diesen Stufen für Wiederholungen gerüstet sein. `40001` abfangen und die ganze Transaktion ab `BEGIN` erneut ausführen, einschließlich ihrer Lesevorgänge. Ihre Entscheidung beruhte auf inzwischen geänderten Daten. Nur die gescheiterte Anweisung zu wiederholen wiederholt den Fehler.

Zwei Transaktionen geraten in einen Deadlock. Eine sperrt Bestellung 1 und danach Bestellung 2, die andere Bestellung 2 und danach Bestellung 1. Beide warten aufeinander. PostgreSQL erkennt den Kreis und bricht eine Transaktion

ab, damit die andere fertig werden kann. Welche es auswählt, lässt sich schwer vorhersagen. Die dokumentierte Abhilfe besteht darin, in jedem Codepfad Sperren in derselben Reihenfolge anzufordern, etwa durch das Aktualisieren der Zeilen nach aufsteigendem Primärschlüssel, und eine wegen eines Deadlocks abgebrochene Transaktion erneut auszuführen.

Eine E-Mail geht für eine zurückgerollte Bestellung heraus. Die Anwendung sendet innerhalb der Transaktion eine Bestätigung oder veröffentlicht ein Ereignis. Dann scheitert eine spätere Anweisung. Der Rollback entfernt die Bestellung, aber nichts kann die E-Mail zurückholen. Die Nachricht in derselben Transaktion wie die beschriebene Änderung in einer Outbox-Tabelle festhalten und nach dem Commit von einem separaten Prozess senden lassen. Dann existiert die Nachricht genau dann, wenn die Bestellung existiert.

Weiterführende Lektüre

- Transactions — das PostgreSQL-Tutorial: Alles-oder-nichts-Verhalten, implizite Transaktionen und Savepoints, gezeigt an einer Überweisung.
- Transaction Isolation — die vier Standardstufen, ihre jeweilige Sicht in PostgreSQL und die Serialisierungsfehler, die einen erneuten Versuch verlangen.
- Explicit Locking — Zeilen- und Tabellensperren, ihre Haltedauer sowie Erkennung und Vermeidung von Deadlocks.
- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` und die übrigen Sitzungsgrenzen.
- Routine Vacuuming — warum alte Zeilenversionen bleiben, solange eine Transaktion sie noch sehen könnte, und was lange offene Transaktionen für die Bereinigung bedeuten.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

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 →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist eine Transaktion?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

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/glossary/transaction/