Engineer

Транзакція

група змін у базі даних за принципом «все або нічого»

Що це таке

Транзакція — це група змін, яку база даних застосовує як одне ціле: або набирають чинності всі, або жодна. Коли її треба відрізнити від платежу, який теж називають транзакцією, пишуть «транзакція бази даних». Підручник PostgreSQL пояснює суть так: кілька кроків об'єднуються в одну операцію за принципом «все або нічого», а проміжних станів ніхто інший не бачить.

У PostgreSQL транзакція починається з `BEGIN` і закінчується або `COMMIT`, який фіксує зміни й робить їх видимими для інших сеансів, або `ROLLBACK`, який їх відкидає. Інструкція, надіслана окремо, теж виконується всередині транзакції: PostgreSQL неявно обгортає її в `BEGIN` і, якщо все минуло успішно, у `COMMIT`. Довга транзакція може поставити точку збереження командою `SAVEPOINT`, а згодом повернутися до неї через `ROLLBACK TO`: усе, що зроблено після неї, відкидається, а попередня робота лишається.

Гарантії транзакції підсумовує абревіатура ACID. Атомарність означає, що транзакція виконується повністю або не виконується зовсім, і жоден напівготовий результат не переживає збою. Узгодженість означає, що оголошені для даних обмеження виконуються після фіксації, а транзакцію, яка порушила б одне з них, буде відкочено. Ізольованість означає, що зміни транзакції лишаються невидимими для паралельних транзакцій, доки її не зафіксовано. Довговічність означає, що зафіксована зміна переживе аварію сервера.

Ізоляція має рівні, і стандарт SQL визначає чотири: Read Uncommitted, Read Committed, Repeatable Read і Serializable. PostgreSQL починає кожну транзакцію на рівні Read Committed, якщо не вказано інше, а його Read Uncommitted поводиться точнісінько як Read Committed. На Read Committed кожна інструкція бачить дані, зафіксовані до її початку, тож два запити в

одній транзакції можуть побачити різні числа. На Repeatable Read уся транзакція працює з одного знімка, зробленого під час виконання першої інструкції, яка не керує транзакцією. Serializable іде ще далі: паралельні транзакції дають той самий результат, ніби виконувалися одна за одною.

Застосуйте поняття до рішення

Застосуйте «Що таке транзакція?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Структура, що відповідає реальній роботі бізнесу, тож потрібні згодом запитання до даних усе ще можна буде поставити.

Переглянути наші послуги Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Спроєктувала й підтримувала 6 пайплайнів ETL/ELT — 6 пайплайнів ETL на трьох рушіях баз даних

Спроєктувала й підтримувала 6 пайплайнів ETL/ELT Зв'язатися з нами →

Чому це важливо

Майже кожна дія, яку варто записати, зачіпає більше ніж один рядок. Замовлення — це рядок в одній таблиці, списання залишку в іншій і запис у реєстрі. Без транзакції збій, тайм-аут чи порушене обмеження між другою і третьою інструкціями лишить замовлення, під яке так і не списали товар, або списаний товар під замовлення, якого не існує. Згодом хтось натрапить на це у звіті й виправлятиме вручну. З транзакцією такого проміжного стану ніхто ніколи не побачить, і гарантує це база даних, а не застосунок.

Друга причина — паралельність. Два запити, що прийшли одночасно, виконуються пліч-о-пліч: рівень ізоляції визначає, що бачить кожен із них, а блокування — хто з них чекатиме. Саме тут команди найчастіше дивуються: якщо обгорнути код у `BEGIN` і `COMMIT`, це саме собою не завадить двом запитам прочитати той самий баланс і обом його витратити. На Read Committed кожна інструкція читає те, що було зафіксовано на момент її початку, тож між перевіркою в одній інструкції та записом у наступній може вклинитися фіксація іншого сеансу. Перевірка має або спершу заблокувати рядок, або стати частиною запису.

У транзакцій є й ціна, і платять її часом. Блокування рядків, які беруть `UPDATE`, `DELETE` і `SELECT ... FOR UPDATE`, зазвичай тримаються до кінця транзакції; відкат до точки збереження знімає блокування, отримані після цієї точки. Сеанс, якому потрібне конфліктне блокування, має чекати, якщо тільки очікування не зупинить тайм-аут, скасування чи розв’язання взаємоблокування. Відкрита транзакція також не дає `VACUUM` прибрати старі

версії рядків, які їй ще можуть знадобитися, тож транзакція, забута відкритою на години, роздуває таблиці. Тому питання проєктування ніколи не зводиться лише до того, чи потрібна транзакція. Важливо, де пролягають її межі й як мало відбувається між ними.

Нарешті, звичайна локальна транзакція бази даних сама собою не охоплює зовнішніх ефектів. Виклик платіжного провайдера, лист чи повідомлення, опубліковане в окремій черзі, не скасовуються відкатом бази даних. Координація інших систем потребує додаткового проєктування; обгортки з `BEGIN` недостатньо. Вирішити, що опиниться всередині межі і як зовнішній світ дізнається про фіксацію, — це більша частина роботи над транзакцією.

Як це виглядає на практиці

Учасник програми лояльності витрачає 250 балів на квиток у кіно. Три речі мають статися разом: гаманець втрачає бали, реєстр фіксує причину, і з’являється запис про обмін. Як одна транзакція це виглядає так:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- рядка не повернуто: балів замало, тож ROLLBACK і стоп

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'обмін: квиток у кіно');

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

COMMIT;
```

`UPDATE` блокує рядок гаманця учасника 42. До `COMMIT` будь-який інший сеанс, що читає гаманець, бачить старий баланс, бо блокування рядків не заважають читанню, а незафіксовані зміни невидимі. Друге списання для того самого учасника, надіслане за мить з іншої вкладки, доходить до власного `UPDATE` і чекає. Коли перша транзакція фіксується, PostgreSQL на рівні Read Committed ще раз перевіряє умову `WHERE` другої на рядку в його новому стані. Якщо лишилося тільки 100 балів, умова не виконується, рядок

не повертається, і другий запит відкочується, нічого не записавши. Якщо ж збій станеться на останньому `INSERT` — скажімо, винагороду 7 зняли з пропозиції і зовнішній ключ її не пропускає, — клієнт відкочує транзакцію, і бали лишаються рівно там, де були.

Саме так побудовано систему лояльності з лігами й бейджами. Кожна зміна гаманця там спершу блокує його рядок і лише потім читає, тож другий запит чекає, доки завершиться перший, замість того щоб змагатися з ним. А оскільки гаманець лежить у тій самій базі даних, що й усе інше, баланс і те, що на нього куплено, фіксуються разом або не фіксуються зовсім. Сторінка відверто називає ціну: блокування рядків — повільніший варіант, за конкуренції записи виконуються почергово, зате учасник не витратить ті самі бали двічі.

Нотатка про те, як продати останній квиток рівно один раз, випробовує той самий інструмент флеш-розпродажем. Її виправлення об'єднує перевірку залишку і списання в один умовний `UPDATE`, бо блокування, яке тримається, поки застосунок думає між `SELECT` і `UPDATE`, змушує наступного покупця чекати кожну мілісекунду цього часу. Першим у її списку подальшої роботи стоїть правило не пускати в транзакцію нічого зовнішнього: виклику оплати, HTTP-запиту чи надісланому повідомленню місце до неї або після неї.

Де виникають проблеми

Сеанс простоює всередині транзакції, і все за ним чекає. Застосунок відкриває транзакцію, оновлює рядок, а потім чекає на повільний API, на наступний клік користувача або на фоновий процес, який упав, так і не надіславши `COMMIT`. Його блокування лишаються, а коли тайм-аути вимкнені, PostgreSQL змушує сеанс, якому потрібне конфліктне блокування, чекати без обмежень, якщо тільки не виявить взаємоблокування або очікування не буде скасовано. Документація прямо називає поганою ідеєю тримати транзакцію відкритою, поки триває очікування введення від користувача. `pg_stat_activity` показує такі сеанси зі станом `idle in transaction`, а `xact_start` — коли почалася транзакція:

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

Винесіть повільну роботу з транзакції й задайте серверу межу, щоб забута транзакція не тримала блокування годинами.
`idle_in_transaction_session_timeout` завершує будь-який сеанс, що простоює у відкритій транзакції довше за задане значення. За замовчуванням параметр дорівнює 0, тобто вимкнений, тож його треба задати явно, наприклад для ролі застосунку: `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

Два запити витрачають той самий баланс. Код читає значення однією інструкцією, ухвалює рішення в застосунку, а результат записує іншою. Два сеанси читають те саме число, обидва проходять перевірку й обидва записують, тож одна зміна мовчки затирає іншу. Обгортка з `BEGIN` на рівні Read Committed цьому не запобігає. Зробіть перевірку частиною запису, додавши умову в `UPDATE` , як у прикладі вище, або заблокуйте рядок перед читанням через `SELECT ... FOR UPDATE` , і тоді другий сеанс чекатиме на перший.

Під навантаженням на Repeatable Read чи Serializable з'являються помилки. На цих рівнях PostgreSQL відхиляє транзакцію, яка порушила б обіцянку рівня, і повідомляє про помилку серіалізації, SQLSTATE `40001` , з повідомленнями на кшталт `could not serialize access due to concurrent update` . Це не баг, а задум, і документація прямо каже, що застосунки на цих рівнях мають бути готові повторити транзакцію. Перехоплюйте `40001` і запускайте всю транзакцію заново від `BEGIN` , разом із читаннями, бо рішення в ній ухвалено на даних, які відтоді змінилися. Якщо повторити лише ту інструкцію, що впала, повториться й помилка.

Дві транзакції потрапляють у взаємоблокування. Одна блокує замовлення 1, а потім замовлення 2, інша — замовлення 2, а потім 1, і кожна чекає на іншу. PostgreSQL виявляє цикл і перериває одну з транзакцій, щоб інша могла завершитися, а яку саме — передбачити важко. Документація радить, щоб кожен шлях у коді брав блокування в тому самому порядку, наприклад оновлював рядки за зростанням первинного ключа, а транзакцію, перервану через взаємоблокування, повторював.

Лист іде за замовленням, яке відкотили. Застосунок надсилає підтвердження чи публікує подію зсередини транзакції, а потім наступна інструкція падає. Відкат прибирає замовлення, але лист уже не повернеш. Записуйте повідомлення в таблицю outbox у тій самій транзакції, що й зміну, яку воно описує, і нехай окремий процес надсилає його вже після фіксації. Тоді повідомлення існує рівно тоді, коли існує замовлення.

Що почитати далі

- Transactions — розділ підручника PostgreSQL: атомарність, неявні транзакції й точки збереження на прикладі банківського переказу.
- Transaction Isolation — чотири стандартні рівні, що бачить кожен із них у PostgreSQL, і помилки серіалізації, після яких транзакцію треба повторити.
- Explicit Locking — блокування рядків і таблиць, як довго вони тримаються, як PostgreSQL виявляє взаємоблокування і як їм запобігти.
- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` та інші обмеження сеансу.
- Routine Vacuuming — чому старі версії рядків лишаються, доки їх може бачити хоч одна транзакція, і що довгі відкриті транзакції роблять із прибиранням.

Словник

Обирайте за фактичними умовами

Надайте репрезентативний набір даних, запитання до нього та поточне вузьке місце. Узгодимо вимірювання якості, пропускної здатності та відновлення.

Дивитися, скільки це коштує Зв'язатися з нами →

Зробіть термінологію практичною

Запитайте, як «Що таке транзакція?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

Переглянути наші послуги

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

Дбайливо зроблено з любов'ю — D.

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

Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.

Надруковано з https://engineer.company/uk/glossary/transaction/