Engineer

Transaktion

en grupp databasändringar som genomförs helt eller inte alls

Vad det är

En transaktion är en grupp ändringar som en databas genomför som en enhet: antingen får alla verkan eller ingen av dem. När den behöver hållas isär från en betalning, som också kallas transaktion, skriver man databastransaktion. PostgreSQL-handledningen beskriver idén så här: flera steg samlas i en enda operation där allt eller inget sker, och ingen annan ser något av det som ligger däremellan.

I PostgreSQL öppnas en transaktion med `BEGIN` och avslutas antingen med `COMMIT`, som gör ändringarna beständiga och synliga för andra sessioner, eller med `ROLLBACK`, som kastar dem. En sats som skickas ensam körs också i en transaktion: PostgreSQL lägger ett implicit `BEGIN` runt den och, om den lyckas, ett implicit `COMMIT`. En lång transaktion kan sätta en sparpunkt med `SAVEPOINT` och senare gå tillbaka till den med `ROLLBACK TO`, så att arbetet efter punkten kastas medan allt före den behålls.

Garantierna en transaktion ger sammanfattas som ACID. Atomicitet betyder att transaktionen genomförs som en enhet eller inte alls och att inget halvfärdigt resultat överlever en krasch. Konsistens betyder att integritetsvillkoren som definierats för datan gäller när den committas, och att en transaktion som skulle bryta mot något av dem rullas tillbaka i stället. Isolering betyder att dess ändringar är osynliga för samtidiga transaktioner tills den har committats. Hållbarhet betyder att en committad ändring överlever att servern kraschar.

Isolering finns i nivåer, och SQL-standarden definierar fyra: Read Uncommitted, Read Committed, Repeatable Read och Serializable. PostgreSQL startar varje transaktion på Read Committed om inget annat anges, och dess Read Uncommitted beter sig exakt som Read Committed. På Read Committed ser varje sats de data som committats innan satsen började, så två frågor i samma

transaktion kan se olika siffror. På Repeatable Read arbetar hela transaktionen utifrån en ögonblicksbild, tagen när dess första sats som inte styr transaktionen körs. Serializable går längre och ser till att samtidiga transaktioner ger samma resultat som om de hade körts en i taget.

Använd begreppet i ett beslut

Använd ”Vad är en transaktion?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. En struktur som speglar hur verksamheten faktiskt fungerar, så att de frågor du behöver svar på senare fortfarande går att ställa.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Utformade och drev 6 ETL/ELT-pipelines — 6 ETL-pipelines över tre databasmotorer

Utformade och drev 6 ETL/ELT-pipelines Hör av er →

Varför det spelar roll

Nästan varje handling som är värd att registrera rör mer än en rad. En order är en rad i en tabell, en nedräkning av lagret i en annan och en post i en huvudbok. Utan transaktion kan en krasch, en timeout eller ett brutet integritetsvillkor mellan andra och tredje satsen lämna kvar en order vars lager aldrig drogs, eller lager som dragits för en order som inte finns. Sedan hittar någon det i en rapport och rättar det för hand. Med en transaktion finns det halva tillståndet aldrig för någon att se, och det är databasen, inte applikationen, som garanterar det.

Det andra skälet är samtidighet. Två anrop som kommer samtidigt körs sida vid sida, och isoleringsnivån avgör vad vart och ett ser medan lås avgör vilket som får vänta. Det är här team oftast blir överraskade: att slå in kod i `BEGIN` och `COMMIT` hindrar inte i sig två anrop från att läsa samma saldo och båda spendera det. På Read Committed läser varje sats det som var committat när den startade, så en kontroll i en sats och en skrivning i nästa kan skiljas åt av en commit från en annan session. Kontrollen måste antingen låsa raden först eller bli en del av skrivningen.

Transaktioner har också ett pris, och det betalas i tid. Radlås som tas av `UPDATE`, `DELETE` och `SELECT ... FOR UPDATE` hålls normalt tills transaktionen avslutas; en tillbakarullning till en sparpunkt frigör lås som tagits efter den sparpunkten. En session som vill ha ett konflikterande lås måste vänta om inte en tidsgräns, ett avbrott eller lösningen av ett deadlock stoppar väntan. En öppen transaktion hindrar också `VACUUM` från att ta bort gamla radversioner som den fortfarande

kan behöva se, så en transaktion som står öppen i timmar får tabellerna att svälla. Designfrågan är därför aldrig bara om en transaktion ska användas, utan var dess gränser går och hur lite som händer mellan dem.

Slutligen omfattar en vanlig lokal databastransaktion inte i sig externa effekter. Ett anrop till en betalleverantör, ett mejl eller ett meddelande som publicerats på en separat kö blir inte ogjort av att databasen rullas tillbaka. Samordning med andra system kräver ytterligare design; ett omgivande `BEGIN` räcker inte. Att bestämma vad som hamnar innanför gränsen, och hur omvärlden får veta om en commit, är det mesta av arbetet med att utforma en transaktion.

Så ser det ut i praktiken

En medlem i ett lojalitetsprogram löser in 250 poäng mot en biobiljett. Tre saker måste ske tillsammans: plånboken förlorar poängen, huvudboken registrerar varför och inlösen skapas. Skrivet som en transaktion ser det ut så här:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- ingen rad tillbaka: för få poäng, så ROLLBACK och stopp här

INSERT INTO points_entry (member_id, delta, reason)
VALUES (42, -250, 'inlöst: biobiljett');

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

COMMIT;
```

`UPDATE` tar ett radlås på plånboken för medlem 42. Fram till `COMMIT` ser varje annan session som läser plånboken fortfarande det gamla saldot, eftersom radlås inte blockerar läsare och ändringar som inte har committats är osynliga. En andra inlösen för samma medlem, skickad från en annan flik ett ögonblick senare, når sin egen `UPDATE` och väntar. När den första transaktionen committas utvärderar PostgreSQL på Read Committed den andras `WHERE` -villkor på nytt mot raden som den ser ut nu. Om bara 100 poäng finns kvar är villkoret inte uppfyllt, ingen rad kommer tillbaka och det andra anropet rullas tillbaka utan att

ha skrivit något. Om i stället den sista `INSERT` misslyckas, till exempel för att belöning 7 har dragits tillbaka och en främmande nyckel avvisar den, rullar klienten tillbaka och poängen är exakt där de var.

Det här är formen på lojalitetsplånboken med ligor. Varje skrivoperation där tar ett radlås på plånboken innan den läses, så att det andra anropet väntar på att det första ska bli klart i stället för att kapplöpa med det, och eftersom plånboken ligger i samma databas som allt annat committas saldot och det som köptes för det tillsammans eller inte alls. Sidan säger öppet vad det kostar: radlåsning är det långsammare svaret, och vid trängsel får transaktionerna vänta på sin tur, men alternativet är en medlem som spenderar samma poäng två gånger.

Anteckningen om att sälja den sista biljetten exakt en gång prövar samma verktyg under en blixtrea. Lösningen där gör kontrollen och nedräkningen till en enda villkorad `UPDATE`, eftersom ett lås som hålls medan applikationen tänker mellan en `SELECT` och en `UPDATE` får nästa köpare att vänta varenda millisekund av det. Först på listan över resten av arbetet står att hålla allt externt utanför transaktionen: ett betalningsanrop, en HTTP-begäran eller ett publicerat meddelande hör hemma före den eller efter den.

Var det går fel

En session står overksam i en transaktion och allt bakom den får vänta. En applikation öppnar en transaktion, uppdaterar en rad och väntar sedan på ett långsamt API, på användarens nästa klick eller på en bakgrundsprocess som kraschade utan att någonsin skicka `COMMIT`. Låsen ligger kvar, och med tidsgränser avstängda låter PostgreSQL en session som vill ha ett konflikterande lås vänta på obestämd tid om den inte upptäcker ett deadlock eller väntan avbryts. Dokumentationen kallar det en dålig idé att hålla en transaktion öppen medan man väntar på indata från användaren. `pg_stat_activity` visar sådana sessioner med tillståndet `idle in transaction`, och `xact_start` anger när transaktionen började:

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

Flytta ut det långsamma arbetet ur transaktionen, och ge servern en gräns så att en bortglömd transaktion inte kan hålla lås i timmar. `idle_in_transaction_session_timeout` avslutar varje session som har stått overksam i en öppen transaktion längre än värdet. Standardvärdet är 0, vilket stänger av den, så den måste sättas, till exempel för applikationens roll med `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

Två anrop spenderar samma saldo. Koden läser ett värde i en sats, fattar beslut i applikationen och skriver resultatet i en annan. Två sessioner läser samma tal, båda klarar kontrollen och båda skriver, så den ena ändringen skriver tyst över den andra. Ett omgivande `BEGIN` förhindrar det inte på Read Committed. Gör kontrollen till en del av skrivningen med ett villkor i `UPDATE` , som i exemplet ovan, eller lås raden innan den läses med `SELECT ... FOR UPDATE` , så att den andra sessionen väntar på den första.

Fel dyker upp under last på Repeatable Read eller Serializable. På de här nivåerna avvisar PostgreSQL en transaktion som skulle bryta nivåns löfte och rapporterar ett serialiseringsfel, SQLSTATE `40001` , med meddelanden som `could not serialize access due to concurrent update` . Det är så det är tänkt att fungera, och dokumentationen säger uttryckligen att applikationer som använder de här nivåerna måste vara beredda att försöka igen. Fånga `40001` och kör hela transaktionen igen från `BEGIN` , läsningarna inräknade, eftersom beslutet den fattade byggde på data som sedan har ändrats. Att bara köra om den sats som misslyckades upprepar misstaget.

Två transaktioner hamnar i deadlock (baklås). Den ena låser order 1 och sedan order 2, den andra låser order 2 och sedan order 1, och var och en väntar på den andra. PostgreSQL upptäcker cirkeln och avbryter en av transaktionerna så att den andra kan bli klar, och vilken den väljer är svårt att förutse. Försvaret som dokumentationen anger är att alla kodvägar tar lås i samma ordning, till exempel genom att uppdatera rader i stigande ordning efter primärnyckel, och att en transaktion som avbröts på grund av deadlock körs igen.

Ett mejl går ut för en order som rullades tillbaka. Applikationen skickar en bekräftelse eller publicerar en händelse inifrån transaktionen, och sedan misslyckas en senare sats. Rollbacken tar bort ordern, men inget kan kalla tillbaka mejlet. Skriv meddelandet i en outbox-tabell i samma transaktion som ändringen det beskriver, och låt en separat process skicka det när commiten har skett. Då finns meddelandet exakt när ordern finns.

Läs mer

- Transactions — kapitlet i PostgreSQL-handledningen: allt-eller-inget-beteende, implicita transaktioner och sparpunkter, visade med en banköverföring.
- Transaction Isolation — de fyra standardnivåerna, vad var och en ser i PostgreSQL och serialiseringsfelen som kräver ett nytt försök.
- Explicit Locking — rad- och tabellås, hur länge de hålls och hur deadlocks upptäcks och undviks.
- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` och de andra gränserna för en session.
- Routine Vacuuming — varför gamla radversioner ligger kvar så länge någon transaktion fortfarande kan se dem, och vad långa öppna transaktioner gör med städningen.

Ordlista

Välj utifrån era faktiska förutsättningar

Ta med ett representativt dataset, frågorna det ska besvara och dagens flaskhals. Vi kan avtala mått på kvalitet, genomströmning och återställning.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är en transaktion?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

welcome@engineer.company +45 50 10 61 84 Köpenhamn, Danmark

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/transaction/