Engineer

Transaktion

en gruppe databaseændringer, der lykkes eller fejler samlet

Hvad det er

En transaktion er en gruppe ændringer, som en database udfører som én enhed: enten får de alle virkning, eller også gør ingen af dem. Man skriver databasetransaktion, når det skal være tydeligt, at der ikke er tale om en betaling, som også kaldes en transaktion. PostgreSQL's tutorial beskriver den som en alt-eller-intet-operation, der samler flere trin til ét, og ingen andre kan se tilstandene undervejs.

I PostgreSQL begynder en transaktion med `BEGIN` og slutter enten med `COMMIT`, der gør ændringerne permanente og synlige for andre sessioner, eller med `ROLLBACK`, der kasserer dem. En sætning, der sendes alene, kører også i en transaktion: PostgreSQL omgiver den med et implicit `BEGIN` og, hvis den lykkes, et implicit `COMMIT`. En lang transaktion kan sætte et `SAVEPOINT` og senere vende tilbage til det med `ROLLBACK TO`, så arbejdet siden da forkastes, mens alt før bevares.

De garantier, en transaktion giver, kaldes under ét ACID. Atomicitet betyder, at transaktionen gennemføres som én enhed eller slet ikke, og at intet halvfærdigt resultat overlever et nedbrud. Konsistens betyder, at de constraints, der er erklæret på dataene, holder, når den committer, og at en transaktion, der ville bryde en af dem, rulles tilbage i stedet. Isolation betyder, at dens ændringer er usynlige for samtidige transaktioner, indtil den committer. Holdbarhed betyder, at en committet ændring overlever, at serveren går ned.

Isolation findes i flere niveauer, og SQL-standarden definerer fire: Read Uncommitted, Read Committed, Repeatable Read og Serializable. PostgreSQL starter hver transaktion på Read Committed, medmindre andet er angivet, og dens Read Uncommitted opfører sig præcis som Read Committed. På Read Committed ser hver sætning de data, der var committet, før netop den sætning

begyndte, så to forespørgsler i samme transaktion kan se forskellige tal. På Repeatable Read arbejder hele transaktionen ud fra ét øjebliksbillede, taget når dens første sætning, som ikke styrer transaktionen, kører. Serializable går videre og får samtidige transaktioner til at give samme resultat, som hvis de var kørt én ad gangen.

Brug begrebet i en beslutning

Brug »Hvad er en transaktion?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. En struktur, der passer til, hvordan forretningen faktisk fungerer, så de svar, I får brug for senere, stadig kan hentes.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har designet og drevet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tre databasemotorer

Har designet og drevet 6 ETL/ELT-pipelines Kontakt os →

Hvorfor det betyder noget

Næsten enhver handling, der er værd at registrere, rører mere end én række. En ordre er en række i én tabel, en nedskrivning af lageret i en anden og en postering i en hovedbog. Uden en transaktion kan et nedbrud, en timeout eller en constraint, der fejler mellem anden og tredje sætning, efterlade en ordre, hvis varer aldrig blev trukket fra lageret, eller varer trukket fra til en ordre, der ikke findes. Nogen opdager det senere i en rapport og retter det i hånden. Med en transaktion opstår den halve tilstand aldrig for nogen at se, og det er databasen, ikke applikationen, der står inde for det.

Den anden grund er samtidighed. To requests, der ankommer på samme tid, kører side om side, og isolationsniveauet afgør, hvad hver af dem ser, mens låse afgør, hvem der må vente. Her bliver teams oftest overraskede: at pakke koden ind i `BEGIN` og `COMMIT` forhindrer ikke i sig selv, at to requests læser den samme saldo og begge bruger den. På Read Committed læser hver sætning det, der var committet, da den startede, så et tjek i én sætning og en skrivning i den næste kan blive skilt ad af en anden sessions commit. Tjekket skal enten låse rækken først eller være en del af selve skrivningen.

Transaktioner har også en pris, og den betales i tid. Rækkelåse fra `UPDATE`, `DELETE` og `SELECT ... FOR UPDATE` holdes normalt, til transaktionen slutter; en rollback til et savepoint frigiver låse, der blev taget efter det savepoint. En session, der ønsker en modstridende lås, må vente, medmindre en tidsoverskridelse, en annullering eller løsningen af en deadlock afbryder

ventningen. En åben transaktion forhindrer desuden `VACUUM` i at fjerne gamle rækkeversioner, som den stadig kan få brug for at se, så en transaktion, der står åben i timevis, lader tabellerne svulme op. Spørgsmålet er derfor aldrig kun, om man skal bruge en transaktion, men hvor dens grænser skal gå, og hvor lidt der sker mellem dem.

Endelig omfatter en almindelig lokal databasetransaktion ikke i sig selv eksterne virkninger. Et kald til en betalingsudbyder, en e-mail eller en besked sendt til en separat kø trækkes ikke tilbage ved at rulle databasen tilbage. Koordinering med andre systemer kræver et yderligere design; en omgivende `BEGIN` er ikke nok. At afgøre, hvad der hører til inden for grænsen, og hvordan omverdenen får besked om et commit, er det meste af arbejdet, når man designer en transaktion.

Sådan ser det ud i praksis

Et medlem af en loyalitetsordning bruger 250 point på en biografbillet. Tre ting skal ske samlet: walleten mister pointene, en postering registrerer hvorfor, og indløsningen oprettes. Skrevet som én transaktion ser det sådan ud:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- ingen række tilbage: for få point, så ROLLBACK og stop her

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

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

COMMIT;
```

`UPDATE` tager en rækkelås på medlem 42's wallet. Indtil `COMMIT` ser enhver anden session, der læser walleten, stadig den gamle saldo, fordi rækkelåse ikke blokerer læsere, og fordi ændringer, der ikke er committet, er usynlige. En anden indløsning for det samme medlem, sendt fra en anden fane et øjeblik senere, når frem til sin egen `UPDATE` og venter. Når den første transaktion committer, evaluerer PostgreSQL på Read Committed den andens `WHERE` -betingelse igen

mod rækken, som den ser ud nu. Er der kun 100 point tilbage, holder betingelsen ikke, ingen række kommer tilbage, og den anden request rulles tilbage uden at have skrevet noget. Fejler den sidste `INSERT` i stedet, for eksempel fordi belønning 7 er trukket tilbage, og en fremmednøgle afviser rækken, ruller klienten tilbage, og pointene står præcis, hvor de stod.

Det er netop formen på loyalitetspoint med ligaer og badges. Hver ændring af en wallet tager dér en row lock på den, før den læses, så en anden request venter på, at den første bliver færdig, i stedet for at kappes med den. Og fordi walleten ligger i samme database som alt det andet, committes en saldo og det, den købte, sammen eller slet ikke. Siden lægger ikke skjul på prisen: row locking er det langsommere svar, og medlemmerne må vente på tur i stedet for at bruge de samme point to gange.

Noten om at sælge den sidste billet præcis én gang sætter det samme værktøj under pres i et flash-salg. Dens rettelse gør lagertjekket og nedskrivningen til én betinget `UPDATE`, fordi en lås, der holdes, mens applikationen tænker mellem en `SELECT` og en `UPDATE`, lader den næste køber vente hvert eneste millisekund af det. Øverst på dens liste over resten af arbejdet står at holde alt eksternt ude af transaktionen: et betalingskald, en HTTP-forespørgsel eller en udsendt besked hører til før transaktionen eller efter den.

Hvor det går galt

En session står inaktiv i en transaktion, og alt bag den må vente. En applikation åbner en transaktion, opdaterer en række og venter så på et langsomt API, på en brugers næste klik eller på en worker, der gik ned uden nogensinde at sende `COMMIT`. Låsene bliver hængende, og med timeouts slået fra lader PostgreSQL en session, der ønsker en modstridende lås, vente på ubestemt tid, medmindre den opdager en deadlock, eller ventningen afbrydes. Dokumentationen kalder det en dårlig idé at holde en transaktion åben, mens man venter på input fra en bruger. `pg_stat_activity` viser den slags sessioner med tilstanden `idle in transaction`, og `xact_start` fortæller, hvornår transaktionen begyndte:

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

Flyt det langsomme arbejde ud af transaktionen, og giv serveren en grænse, så en glemt transaktion ikke kan holde låse i timevis.
`idle_in_transaction_session_timeout` afslutter enhver session, der har stået inaktiv i en åben transaktion længere end den angivne værdi. Standardværdien er 0, som slår den fra, så den skal sættes eksplicit, for eksempel for applikationens rolle med `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

To requests bruger den samme saldo. Koden læser en værdi i én sætning, træffer beslutningen i applikationen og skriver resultatet i en anden sætning. To sessioner læser det samme tal, begge består tjekket, og begge skriver, så den ene ændring i stilhed overskriver den anden. Et omsluttende `BEGIN` forhindrer det ikke på Read Committed. Gør tjekket til en del af skrivningen med en betingelse i `UPDATE` , som i eksemplet ovenfor, eller lås rækken, før den læses, med `SELECT ... FOR UPDATE` , så den anden session venter på den første.

Der dukker fejl op under belastning på Repeatable Read eller Serializable. På de niveauer afviser PostgreSQL en transaktion, der ville bryde niveauets løfte, og melder en serialiseringsfejl, SQLSTATE `40001` , med beskeder som `could not serialize access due to concurrent update` . Det er ikke en bug, men designet, der virker, og dokumentationen siger, at applikationer, der bruger disse niveauer, skal være forberedt på at prøve igen. Fang `40001` , og kør hele transaktionen igen fra `BEGIN` , også dens læsninger, for beslutningen byggede på data, der siden har ændret sig. Kører man kun den sætning, der fejlede, igen, gentager man fejlen.

To transaktioner ender i deadlock. Den ene låser ordre 1 og derefter ordre 2, den anden låser ordre 2 og derefter ordre 1, og hver venter på den anden. PostgreSQL opdager kredsløbet og afbryder en af transaktionerne, så den anden kan blive færdig, og hvilken den vælger, er svært at forudsige. Det dokumenterede forsvar er, at alle kodestier tager deres låse i samme rækkefølge, for eksempel ved at opdatere rækker i stigende orden efter primærnøgle, og at en transaktion, der blev afbrudt på grund af en deadlock, prøves igen.

Der går en e-mail ud om en ordre, der blev rullet tilbage. Applikationen sender en bekræftelse eller publicerer en hændelse inde fra transaktionen, og bagefter fejler en senere sætning. Rollbacken fjerner ordren, men intet kan kalde e-mailen tilbage. Gem i stedet beskeden i en outbox-tabel i samme transaktion som den ændring, den beskriver, og lad en separat proces sende den, når commit er sket. Så findes beskeden præcis, når ordren gør.

Læs mere

- Transactions — kapitlet i PostgreSQL's tutorial: alt-eller-intet, implicitte transaktioner og savepoints, vist med en bankoverførsel.
- Transaction Isolation — de fire standardniveauer, hvad hvert af dem ser i PostgreSQL, og de serialiseringsfejl, der kræver et nyt forsøg.
- Explicit Locking — række- og tabellåse, hvor længe de holdes, og hvordan deadlocks opdages og undgås.
- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` og de øvrige grænser for en session.
- Routine Vacuuming — hvorfor gamle rækkeversioner bliver liggende, så længe en transaktion stadig kan se dem, og hvad lange åbne transaktioner gør ved oprydningen.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring et repræsentativt datasæt, de spørgsmål det skal besvare, og den aktuelle flaskehals. Vi kan aftale målinger af kvalitet, gennemløb og gendannelse.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er en transaktion?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

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

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/transaction/