Engineer

Transaksjon

en gruppe databaseendringer som lykkes eller feiler samlet

Hva det er

En transaksjon er en gruppe endringer som en database utfører som én enhet: enten får alle virkning, eller så får ingen av dem det. Når den må holdes adskilt fra en betaling, som også kalles en transaksjon, skriver folk databasetransaksjon. PostgreSQL-veiledningen forklarer det slik: flere steg samles i én operasjon der alt eller ingenting skjer, og ingen andre ser noe av det som ligger imellom.

I PostgreSQL åpnes en transaksjon med `BEGIN` og avsluttes enten med `COMMIT`, som gjør endringene permanente og synlige for andre økter, eller med `ROLLBACK`, som forkaster dem. En setning som sendes alene, kjører også inne i en transaksjon: PostgreSQL legger en implisitt `BEGIN` rundt den og, hvis den lykkes, en implisitt `COMMIT`. En lang transaksjon kan sette et lagringspunkt med `SAVEPOINT` og senere gå tilbake til det med `ROLLBACK TO`, slik at arbeidet etter punktet forkastes mens alt før det beholdes.

Garantiene en transaksjon gir, oppsummeres som ACID. Atomisitet betyr at transaksjonen fullføres som én enhet eller ikke i det hele tatt, og at ikke noe halvferdig resultat overlever et krasj. Konsistens betyr at integritetsreglene som er definert på dataene, holder når den committes, og at en transaksjon som ville brutt en av dem, rulles tilbake i stedet. Isolasjon betyr at endringene er usynlige for samtidige transaksjoner til den er committet. Varighet betyr at en committet endring overlever at serveren krasjer.

Isolasjon kommer i nivåer, og SQL-standarden definerer fire: Read Uncommitted, Read Committed, Repeatable Read og Serializable. PostgreSQL starter hver transaksjon på Read Committed med mindre noe annet er angitt, og Read Uncommitted oppfører seg der nøyaktig som Read Committed. På Read Committed ser hver setning dataene som var committet før setningen startet, så to spørringer i samme transaksjon kan se forskjellige tall. På Repeatable Read

arbeider hele transaksjonen ut fra ett øyeblikksbilde, tatt når den første setningen som ikke er en transaksjonskontrollsetning, kjører. Serializable går lenger og sørger for at samtidige transaksjoner gir samme resultat som om de hadde kjørt én etter én.

Bruk begrepet i en beslutning

Bruk «Hva er en transaksjon?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. En struktur som samsvarer med hvordan virksomheten faktisk fungerer, slik at spørsmålene du trenger svar på senere, fortsatt er mulige å stille.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Designet og driftet 6 ETL/ELT-pipelines — 6 ETL-pipelines på tvers av tre databasemotorer

Designet og driftet 6 ETL/ELT-pipelines Ta kontakt →

Hvorfor det er viktig

Nesten enhver handling som er verdt å registrere, berører mer enn én rad. En ordre er en rad i én tabell, en nedtelling av lageret i en annen og en post i en hovedbok. Uten transaksjon kan et krasj, et tidsavbrudd eller en brutt integritetsregel mellom andre og tredje setning etterlate en ordre der lageret aldri ble trukket, eller lager trukket for en ordre som ikke finnes. Så finner noen det i en rapport og retter det for hånd. Med en transaksjon finnes den halve tilstanden aldri for noen å se, og det er databasen, ikke applikasjonen, som garanterer det.

Den andre grunnen er samtidighet. To forespørsler som kommer samtidig, kjører side om side, og isolasjonsnivået avgjør hva hver av dem ser, mens låser avgjør hvem som venter. Her blir team oftest overrasket: å pakke kode inn i `BEGIN` og `COMMIT` hindrer ikke i seg selv at to forespørsler leser samme saldo og begge bruker den. På Read Committed leser hver setning det som var committet da den startet, så en sjekk i én setning og en skriving i den neste kan skilles av en commit fra en annen økt. Sjekken må enten låse raden først eller bli en del av skrivingen.

Transaksjoner har også en pris, og den betales i tid. Radlåser som tas av `UPDATE`, `DELETE` og `SELECT ... FOR UPDATE`, varer normalt til transaksjonen avsluttes; en tilbakerulling til et lagringspunkt frigir låser tatt etter det lagringspunktet. En økt som vil ha en motstridende lås, må vente med mindre et tidsavbrudd, en avbrytelse eller løsning av en vranglås stopper ventingen. En åpen transaksjon hindrer også `VACUUM` i å fjerne gamle radversjoner den fortsatt kan trenge å se,

så en transaksjon som står åpen i timevis, får tabellene til å svulme opp. Designspørsmålet er derfor aldri bare om man skal bruke en transaksjon, men hvor grensene går og hvor lite som skjer mellom dem.

Til slutt omfatter ikke en vanlig lokal databasetransaksjon i seg selv eksterne virkninger. Et kall til en betalingsleverandør, en e-post eller en melding publisert på en separat kø blir ikke ugjort ved å rulle databasen tilbake. Koordinering med andre systemer krever ytterligere utforming; en omsluttende `BEGIN` er ikke nok. Å bestemme hva som skal ligge innenfor grensen, og hvordan verden utenfor får vite om en commit, er det meste av arbeidet med å utforme en transaksjon.

Slik ser det ut i praksis

Et medlem av et lojalitetsprogram bruker 250 poeng på en kinobillett. Tre ting må skje sammen: lommeboken mister poengene, hovedboken registrerer hvorfor, og innløsningen opprettes. Skrevet som én transaksjon ser det slik ut:

```
BEGIN;

UPDATE wallet
SET    points = points - 250
WHERE  member_id = 42
AND    points >= 250
RETURNING points;
-- ingen rad tilbake: for få poeng, så ROLLBACK og stopp her

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

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

COMMIT;
```

`UPDATE` tar en radlås på lommeboken til medlem 42. Fram til `COMMIT` ser enhver annen økt som leser lommeboken, fortsatt den gamle saldoen, fordi radlåser ikke blokkerer lesere og endringer som ikke er committet, er usynlige. En ny innløsning for samme medlem, sendt fra en annen fane et øyeblikk senere, når sin egen `UPDATE` og venter. Når den første transaksjonen committes, evaluerer PostgreSQL på Read Committed `WHERE`-betingelsen til den andre på nytt mot raden slik den står nå. Er det bare 100 poeng igjen, er betingelsen ikke oppfylt, ingen rad kommer tilbake, og den andre forespørselen rulles tilbake uten å ha

skrevet noe. Feiler i stedet den siste `INSERT`, for eksempel fordi belønning 7 er trukket tilbake og en fremmednøkkel avviser den, ruller klienten tilbake, og poengene er nøyaktig der de var.

Dette er formen på lojalitetslommeboken med ligaer. Hver endring der tar en radlås på lommeboken før den leses, slik at den andre forespørselen venter til den første er ferdig i stedet for å kappes med den, og fordi lommeboken ligger i samme database som alt annet, committes saldoen og det den kjøpte, sammen eller ikke i det hele tatt. Siden legger ikke skjul på prisen: radlåsing er det tregere svaret, og skrivingene må vente på tur, men alternativet er et medlem som bruker de samme poengene to ganger.

Notatet om å selge den siste billetten nøyaktig én gang setter det samme verktøyet på prøve under et lynsalg. Løsningen der gjør sjekken og nedtellingen til én betinget `UPDATE`, fordi en lås som holdes mens applikasjonen tenker mellom en `SELECT` og en `UPDATE`, får den neste kjøperen til å vente hvert eneste millisekund av det. Først på listen over resten av arbeidet står å holde alt eksternt utenfor transaksjonen: et betalingskall, en HTTP-forespørsel eller en publisert melding hører hjemme før den eller etter den.

Hvor det går galt

En økt står inaktiv i en transaksjon, og alt bak den må vente. En applikasjon åpner en transaksjon, oppdaterer en rad og venter så på et tregt API, på brukerens neste klikk eller på en bakgrunnsprosess som krasjet uten å sende `COMMIT`. Låsene blir stående, og med tidsavbrudd slått av lar PostgreSQL en økt som vil ha en motstridende lås, vente på ubestemt tid med mindre den oppdager en vranglås eller ventingen avbrytes. Dokumentasjonen kaller det en dårlig idé å holde en transaksjon åpen mens man venter på input fra brukeren. `pg_stat_activity` viser disse øktene med tilstanden `idle in transaction`, og `xact_start` forteller når transaksjonen begynte:

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

Flytt det trege arbeidet ut av transaksjonen, og gi serveren en grense slik at en glemt transaksjon ikke kan holde låser i timevis.
`idle_in_transaction_session_timeout` avslutter enhver økt som har stått inaktiv i en åpen transaksjon lenger enn verdien. Standardverdien er 0, som slår den av, så den må settes, for eksempel for applikasjonens rolle med `ALTER ROLE app SET idle_in_transaction_session_timeout = '30s';` .

To forespørsler bruker den samme saldoen. Koden leser en verdi i én setning, bestemmer seg i applikasjonen og skriver resultatet i en annen. To økter leser samme tall, begge består sjekken, og begge skriver, så den ene endringen overskriver den andre uten et ord. En `BEGIN` rundt det hele hindrer det ikke på Read Committed. Gjør sjekken til en del av skrivingen med en betingelse i `UPDATE` , som i eksempelet over, eller lås raden før den leses med `SELECT ... FOR UPDATE` , slik at den andre økten venter på den første.

Feil dukker opp under last på Repeatable Read eller Serializable. På disse nivåene avviser PostgreSQL en transaksjon som ville brutt nivåets løfte, og melder en serialiseringsfeil, SQLSTATE `40001` , med meldinger som `could not serialize access due to concurrent update` . Slik er det ment å virke, og dokumentasjonen sier rett ut at applikasjoner som bruker disse nivåene, må være forberedt på å prøve igjen. Fang `40001` , og kjør hele transaksjonen på nytt fra `BEGIN` , lesingene inkludert, siden avgjørelsen den tok, bygde på data som har endret seg siden. Prøver man bare setningen som feilet på nytt, gjentar man feilen.

To transaksjoner havner i vranglås (deadlock). Den ene låser ordre 1 og så ordre 2, den andre låser ordre 2 og så ordre 1, og hver venter på den andre. PostgreSQL oppdager sirkelen og avbryter én av transaksjonene slik at den andre kan bli ferdig, og hvilken den velger, er vanskelig å forutsi. Forsvaret dokumentasjonen anbefaler, er at alle kodestier tar låser i samme rekkefølge, for eksempel ved å oppdatere rader i stigende rekkefølge etter primærnøkkel, og at en transaksjon som ble avbrutt på grunn av vranglås, prøves på nytt.

En e-post går ut for en ordre som ble rullet tilbake. Applikasjonen sender en bekreftelse eller publiserer en hendelse inne fra transaksjonen, og så feiler en senere setning. Rollbacken fjerner ordren, men ingenting kan kalle tilbake e-posten. Skriv meldingen i en outbox-tabell i samme transaksjon som endringen den beskriver, og la en egen prosess sende den når commiten har skjedd. Da finnes meldingen nøyaktig når ordren gjør det.

Les mer

- Transactions — kapittelet i PostgreSQL-veiledningen: alt-eller-ingenting-oppførsel, implisitte transaksjoner og lagringspunkter, vist med en bankoverføring.
- Transaction Isolation — de fire standardnivåene, hva hvert av dem ser i PostgreSQL, og serialiseringsfeilene som krever et nytt forsøk.
- Explicit Locking — rad- og tabellåser, hvor lenge de holdes, og hvordan vranglåser oppdages og unngås.
- Client Connection Defaults — `default_transaction_isolation`, `idle_in_transaction_session_timeout`, `lock_timeout` og de andre grensene for en økt.
- Routine Vacuuming — hvorfor gamle radversjoner blir liggende så lenge en transaksjon fortsatt kan se dem, og hva lange åpne transaksjoner gjør med oppryddingen.

Ordliste

Velg ut fra deres faktiske rammer

Ta med et representativt datasett, spørsmålene det skal besvare og dagens flaskehals. Vi kan avtale målinger av kvalitet, gjennomstrømning og gjenoppretting.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er en transaksjon?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/transaction/