Engineer

Materialisert visning

📷 en visning som lagrer spørringsresultatet som en tabell

Hva det er

En materialisert visning er en spørring der databasen tar vare på resultatet, slik at en lesing returnerer rader som er regnet ut tidligere, i stedet for å kjøre spørringen på nytt. På engelsk heter det materialized view, og det er også navnet kommandoene bruker. Den plasserer seg mellom to eldre objekter. En vanlig visning lagrer bare spørringen sin, og hver lesing kjører den mot tabellene under. En tabell laget med `CREATE TABLE AS` lagrer et resultat, men glemmer spørringen som ga det. En materialisert visning beholder begge deler: radene, som lesingene henter fra, og spørringen, som kan kjøres igjen for å erstatte dem.

PostgreSQL fikk materialiserte visninger i versjon 9.3, utgitt 9. september 2013, og utgivelsesnotatene krediterer Kevin Grittner for arbeidet. Ifølge dokumentasjonen bruker de regelsystemet på samme måte som vanlige visninger, men lagrer resultatet i en tabellignende form. For spørringsparseren er en materialisert visning en relasjon som enhver tabell eller visning, så en `SELECT` mot den leser de lagrede radene direkte; den lagrede spørringen brukes bare når visningen skal fylles. Innholdet kan ikke endres direkte. For å erstatte innholdet med et ferskt spørringsresultat må den lagrede spørringen kjøres på nytt.

To kommandoer gjør den jobben. `CREATE MATERIALIZED VIEW` definerer visningen og kjører spørringen med en gang for å fylle den, med mindre den får `WITH NO DATA`. `REFRESH MATERIALIZED VIEW` kjører den lagrede spørringen igjen og erstatter hele innholdet med det nye resultatet. Versjon 9.4, utgitt 18. desember 2014, la til `CONCURRENTLY`, som lar en oppdatering kjøre uten å blokkere øktene som leser visningen. Begge kommandoene er PostgreSQL-utvidelser og ikke en del av SQL-standarden; andre databaser tilbyr ideen med sin egen syntaks og sine egne regler.

Bruk begrepet i en beslutning

Bruk «Hva er en materialisert visning?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Finne spørringene og de manglende indeksene bak et system som har blitt tyngre for hvert år, og rette dem som betyr noe.

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. 30 dagers strømdata på 180 ms — 180 ms for å aggregere en måned med måleravlesninger

30 dagers strømdata på 180 ms Ta kontakt →

Hvorfor det er viktig

Noen spørringer er dyre og blir stilt ofte: et dashbord som summerer et års ordrer per dag, en rapport som joiner seks tabeller, en side som rangerer produkter etter salg. Kjørt mot kildetabellene betaler hver forespørsel for hele skanningen, joinen og aggregeringen på nytt, selv om svaret knapt endrer seg fra den ene forespørselen til den neste. En materialisert visning betaler den prisen én gang per oppdatering, og hver lesing etter det er en skanning av en liten tabell som er regnet ut på forhånd. Ifølge PostgreSQL-dokumentasjonen går det ofte mye raskere å lese de lagrede dataene enn å gå til tabellene under, enten det skjer direkte eller gjennom en visning.

Prisen er ferskhet. Mellom to oppdateringer viser visningen dataene slik de var ved den siste, og i PostgreSQL oppdaterer ingenting den med mindre noen, eller noe planlagt, ber om det. Mange bruksområder kan leve med det. Gårsdagens salg endrer seg ikke i dag, og en månedsrapport trenger ikke ordrene fra siste minutt. Det en materialisert visning tvinger frem, er en uttrykkelig beslutning: hvor gammelt svaret får være, og hvem som har ansvaret for at det ikke blir eldre.

Den gir også indekser et sted å bo: det lagrede resultatet kan få sine egne indekser, bygget for spørringene som leser det. PostgreSQL-dokumentasjonen viser en annen bruk av det samme: data hentet inn fra et annet system gjennom en foreign data wrapper, som kanskje ikke støtter indekser i det hele tatt, kan kopieres over i en materialisert visning som gjør det.

For et team som holder skjemaet sitt normalisert, er det ofte her den flate formen hører hjemme. Databasenormalisering lagrer hvert faktum ett sted og besvarer de fleste spørsmål gjennom joins. En materialisert visning kan holde det joinede og summerte resultatet som rapporteringen vil ha, mens kildetabellene beholder sin ene kopi av hvert faktum, og hver skriving fortsatt går til ett sted.

Slik ser det ut i praksis

Ta ordretabellene til en nettbutikk: `orders`, med én rad per ordre og datoen den ble lagt inn, og `order_lines`, med varenummer, antall og enhetspris for hver linje. Et salgsdashbord skal vise solgte enheter og omsetning per produkt per dag. Spørringen er kort, men over flere års ordrer leser den hver eneste linje:

```
CREATE MATERIALIZED VIEW daily_sales AS
SELECT o.ordered_on                  AS day,
       l.sku,
       sum(l.quantity)               AS units,
       sum(l.quantity * l.unit_price) AS revenue
FROM   orders o
JOIN   order_lines l USING (order_id)
WHERE  o.ordered_on < CURRENT_DATE
GROUP  BY o.ordered_on, l.sku;
```

`WHERE`-betingelsen utelater dagen i dag, der tallene fortsatt beveger seg, så dashbordet bare viser hele dager; eksempelet i PostgreSQLs egen dokumentasjon behandler en dag med uferdig salg på samme måte. Kommandoen kjører spørringen én gang og lagrer resultatet. Deretter kommer en indeks:

```
CREATE UNIQUE INDEX daily_sales_sku_day ON daily_sales (sku, day);
```

Indeksen har to oppgaver. Den lar dashbordet finne dagene for ett produkt uten å lese hele visningen, og det er den som senere gjør en samtidig oppdatering mulig. Spørringen grupperer på dag og varenummer, så hvert par forekommer én gang, og derfor kan indeksen være unik. Fra nå av leser dashbordet visningen som en tabell:

```
SELECT day, units, revenue
FROM   daily_sales
WHERE  sku = 'TEA-250'
ORDER  BY day;
```

Hver natt, kort etter midnatt, oppdaterer en planlagt jobb den:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY daily_sales;
```

Uten `CONCURRENTLY` bruker en oppdatering som berører mange rader, gjerne færre ressurser og blir raskere ferdig, men den kan blokkere lesingene fra dashbordet til den er ferdig. Med `CONCURRENTLY` fortsetter lesingene under oppdateringen, og dokumentasjonen påpeker at det til og med kan være det raskeste valget når bare få rader endres, som her, der én ny dag kommer til hver natt. Valget har to betingelser. Visningen må allerede inneholde data, og den må ha minst én unik indeks bygget av rene kolonnenavn som dekker alle radene, altså ingen uttrykksindeks og ingen `WHERE` -betingelse. Indeksen over oppfyller den andre; at visningen ble laget med dataene sine, oppfylte den første.

Andre databaser trekker grensen andre steder. I BigQuery blir materialiserte visninger vedlikeholdt i bakgrunnen: endringer i basistabellene legges til trinnvis uten at noen ber om det, en spørring mot visningen fletter inn de siste endringene slik at resultatet holder seg ferskt, og smart tuning kan sende en spørring som er skrevet mot basistabellene, videre til visningen i stedet. Det gjelder inkrementelle visninger, som bare godtar et begrenset utvalg av SQL. En visning definert med `allow_non_incremental_definition = true` godtar de fleste spørringer, men hver oppdatering kjører hele spørringen, og resultatene er ikke alltid ferske.

SQL Server når et lignende resultat med indekserte visninger. Lager man en unik clustered indeks på en visning, som må være laget `WITH SCHEMABINDING` og ha en deterministisk definisjon, lagres resultatet på samme måte som en tabell med en clustered indeks, og spørringsoptimalisereren kan bruke den, også for en spørring som ikke nevner visningen. SQL Server oppdaterer den indekserte visningen sammen med hver insert, update og delete på tabellene den er avhengig av, så det finnes ingen oppdatering å planlegge; kostnaden flytter over på skrivingene, og dokumentasjonen fra Microsoft advarer om at ytelsen deres kan falle betydelig.

Den samme avveiningen viser seg i produksjonsarbeid. I en tidligere rolle utviklet grunnleggeren ETL/ELT-pipelines i PL/pgSQL, som tok i bruk partisjonerte tabeller og materialiserte visninger for å optimalisere leseytelsen for store datasett, sammen med primær- og fremmednøkkelbegrensninger for referanseintegriteten og unike og sammensatte indekser for JOIN-operasjonene.

Hvor det går galt

Gamle tall som ser ferske ut. En oppdateringsjobb som sluttet å kjøre for en uke siden, etterlater en visning som svarer på hver spørring, raskt og feil, og ingenting i resultatet forteller hvor gammelt det er. La jobben registrere tidspunktet for hver vellykket oppdatering et sted der dashbordet kan vise det, og send et varsel når tidspunktet er eldre enn leserne har godtatt.

En oppdatering som blokkerer leserne. En vanlig `REFRESH MATERIALIZED VIEW` over en stor visning kan blokkere alle økter som prøver å lese den, til oppdateringen er ferdig. Legg til en unik indeks og bruk `CONCURRENTLY`, eller legg de vanlige oppdateringene til timer da ingen leser visningen. Når de fleste radene endres ved hver kjøring, kan den vanlige oppdateringen likevel være den billigste.

`CONCURRENTLY` **blir avvist.** Valget krever en unik indeks på rene kolonner som dekker alle radene. En unik indeks på `lower(email)`, eller en som er avgrenset av en `WHERE`-betingelse, holder ikke. Det krever også en visning som allerede inneholder data, så det feiler på en visning laget `WITH NO DATA`, og de to valgene kan ikke kombineres. Legg til en indeks som oppfyller kravene, og fyll visningen én gang med en vanlig oppdatering.

En visning som ikke kan skannes. `WITH NO DATA` lager visningen uten å kjøre spørringen og merker den som ikke skannbar, og enhver spørring mot den feiler frem til den første oppdateringen. Det kan være riktig valg i en migrering som ikke skal vente på en lang spørring, men da må den første oppdateringen planlegges, ikke glemmes.

En endret definisjon som aldri kom frem. `CREATE MATERIALIZED VIEW IF NOT EXISTS` i en migrering gjør ingenting utover å gi en notice når en visning med det navnet allerede finnes, og dokumentasjonen gir ingen garanti for at den eksisterende visningen ligner den nye definisjonen. En migrering som endrer spørringen på den måten, lar den gamle bli stående. Fjern visningen med `DROP MATERIALIZED VIEW`, og lag den på nytt i samme migrering.

Oppdateringsjobber som overlapper. Bare én oppdatering om gangen kan kjøre mot en gitt materialisert visning, også med `CONCURRENTLY`, så en tidsplan som er tettere enn selve oppdateringen, etterlater visningen bare lenger bak. Mål hvor lang tid en oppdatering tar, og legg tidsplanen ut fra det.

Å stole på rekkefølgen til radene. Har den definerende spørringen en `ORDER BY`, kommer det første innholdet i den rekkefølgen, men en oppdatering lover ikke å beholde den. Sett `ORDER BY` i spørringene som leser visningen.

Funksjoner som mister tabellene sine. Mens den definerende spørringen kjører, ved opprettelsen eller ved en oppdatering, setter PostgreSQL `search_path` midlertidig til `pg_catalog, pg_temp` og begrenser hva spørringen får gjøre: en funksjon som lager midlertidige tabeller, feiler. En funksjon som finner tabellene sine ved et ukvalifisert navn, kan bestå hver test for hånd og først knekke inne i oppdateringen. Kvalifiser tabellnavnene med skjemaet i alle funksjoner visningen kaller.

Feil rolle kjører jobben. En oppdatering krever privilegiet `MAINTAIN` på den materialiserte visningen. En jobb som kjører som en rapporteringsrolle som får lese visningen, men ikke vedlikeholde den, feiler hver natt. Gi rollen som kjører oppdateringen, det privilegiet.

En lagret kopi der en indeks ville holdt. Er en spørring treg fordi en indeks mangler, skjuler en materialisert visning problemet og legger gamle data på toppen. Les spørringsplanen først. Når en spørring bare trenger et navn, holder en vanlig visning, og den er alltid oppdatert; spar materialiserte visninger til resultater som er dyre å regne ut og trygge å lese litt gamle.

Les mer

- Materialized Views — kapittelet i PostgreSQL-manualen om hvordan materialiserte visninger forholder seg til visninger og tabeller, med en salgsoversikt som oppdateres hver natt og et eksempel med en foreign data wrapper.
- CREATE MATERIALIZED VIEW — kommandoreferansen, blant annet `WITH NO DATA`, `IF NOT EXISTS` og den begrensede `search_path` mens spørringen kjører.
- REFRESH MATERIALIZED VIEW — oppdateringskommandoen, betingelsene for `CONCURRENTLY` og privilegiet `MAINTAIN` som den krever.
- PostgreSQL 9.3 release notes — utgivelsen fra september 2013 som la til materialiserte visninger.
- Introduction to materialized views — inkrementelle og ikke-inkrementelle materialiserte visninger i BigQuery, automatisk oppdatering og smart tuning.
- Create indexed views — indekserte visninger i SQL Server, kravene til dem og hva de koster for skrivingene.

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 materialisert visning?» 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/materialized-view/