Engineer

Databaseindeks

en struktur som finner rader uten å lese hele tabellen

Hva det er

En databaseindeks er en egen struktur, lagret ved siden av en tabell, som lar databasen gå rett til radene som oppfyller en betingelse, i stedet for å lese tabellen fra første til siste rad. I dagligtale, og i det meste av PostgreSQL-dokumentasjonen, heter den bare en indeks. Dokumentasjonen sammenligner den med stikkordregisteret bakerst i en bok: den som leter etter ett emne, slår opp der og blar rett til sidene som står oppført, i stedet for å lese gjennom hele boka.

Ta en tabell med ordrer og betingelsen `WHERE customer_id = 42`. Uten en indeks på `customer_id` har PostgreSQL bare én måte å svare på: en sekvensiell skanning, som leser hver eneste rad og beholder dem som passer. Det koster det samme enten kunden har to ordrer eller to tusen, og prisen vokser med tabellen. Med en indeks på kolonnen går PostgreSQL noen nivåer ned i et søketre til oppføringene for 42 og henter bare de radene. En indeks opprettes med én setning, `DROP INDEX` fjerner den igjen, og begge deler kan gjøres når som helst.

Når indeksen først finnes, holder PostgreSQL den i takt med tabellen ved hver skriving, og spørringsplanleggeren avgjør fra spørring til spørring om det lønner seg å bruke den framfor å skanne. Ingenting i SQL-en navngir indeksen, så den samme indeksen kan gjøre en `SELECT` raskere, likeså en `UPDATE` eller `DELETE` med en søkebetingelse og en join.

PostgreSQL har flere slags, hver laget for sin type spørsmål:

- **B-tree** er standarden og det `CREATE INDEX` bygger med mindre man ber om noe annet. Den besvarer likhet og intervaller (`<` , `<=` , `=` , `>=` , `>`), `BETWEEN` , `IN` , `IS NULL` og en `LIKE` der mønsteret er forankret i starten. Den holder oppføringene sortert og kan derfor også levere radene i rekkefølge.
- **Hash** lagrer en 32-biters hashkode for hver verdi og besvarer bare likhet.
- **GiST** og **SP-GiST** er rammeverk for data som geometri, der spørsmålet er «overlapper», «inneholder» eller «nærmest» snarere enn «er lik».
- **GIN** er en invertert indeks for verdier som består av mange deler, for eksempel arrayer, og finner radene der verdien inneholder et bestemt element.
- **BRIN** lagrer bare et sammendrag for hvert intervall av tabellblokker. Den passer for en kolonne der verdiene følger radenes fysiske rekkefølge, slik et innsettingstidspunkt ofte gjør.

I tillegg til typen kan en indeks formes etter spørsmålet. En unik indeks avviser duplikater. En sammensatt indeks dekker flere kolonner, opptil 32. En partiell indeks har en `WHERE` -klausul og inneholder bare radene som oppfyller den. En uttrykksindeks lagrer en beregnet verdi som `lower(email)` . Og en `INCLUDE` -liste tar med ekstra kolonner i indeksen uten å søke på dem, slik at en spørring kan besvares fra indeksen alene.

Bruk begrepet i en beslutning

Bruk «Hva er en databaseindeks?» 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

Forskjellen en indeks gjør, vokser med tabellen. På en liten tabell er en skanning billig, og planleggeren velger ofte, med rette, nettopp den: å lese en håndfull sider i rekkefølge koster mindre enn å gå gjennom et tre for å nå dem. Med titalls millioner rader er forskjellen mellom en skanning og et oppslag forskjellen mellom en side som lastes, og en som får tidsavbrudd. En spørring som var rask ved lansering, kan bli treg et år senere uten at en eneste kodelinje er endret, bare fordi tabellen den leser, har vokst.

Indekser er ikke gratis, og regningen kommer ved skriving. Hver innsetting legger til en oppføring i hver indeks på tabellen. En oppdatering kan bare slippe unna som en heap-only tuple-oppdatering (HOT), der den nye radversjonen blir liggende på samme side og ikke trenger nye indeksoppføringer, og det tillater PostgreSQL bare når oppdateringen ikke endrer noen kolonne som tabellens indekser bruker. BRIN-indekser regnes ikke med, siden de bare lagrer et sammendrag. Hver indeks gjør altså den billige veien smalere, og derfor råder dokumentasjonen til å fjerne indekser som sjelden eller aldri blir brukt. En tabell med et dusin indekser betaler et dusin ganger ved hver innsetting.

Det koster også å bygge en indeks. En vanlig `CREATE INDEX` lar andre økter lese tabellen, men blokkerer skriving til den til byggingen er ferdig, og dokumentasjonen advarer om at det på en svært stor tabell kan ta mange timer.

Til slutt er en indeks bare så god som planleggerens beslutning om å bruke den, og planleggeren arbeider ut fra statistikk som `ANALYZE` har samlet inn. Spørsmålet er ikke «finnes det en indeks?», men «bruker spørringene som betyr noe, den?», og `EXPLAIN` er måten å finne det ut på.

Slik ser det ut i praksis

Kontosiden i en nettbutikk viser en kundes tjue siste ordrer:

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

Med bare en primærnøkkel på `id` må PostgreSQL lese hele `orders`, beholde kundens rader, sortere dem og kaste alle unntatt tjue. `EXPLAIN` viser det: en `Seq Scan` på `orders` med betingelsen som `Filter` og en sortering over. `EXPLAIN ANALYZE` kjører også spørringen og skriver ut de faktiske radantallene og tidene ved siden av estimatene.

Indeksen som passer til spørsmålet, har likhetskolonnen først og sorteringskolonnen etterpå:

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Kjør `EXPLAIN` igjen, og skanningen blir en indeksskanning med `orders_customer_created_idx`, der `customer_id = 42` nå står som `Index Cond` i stedet for som filter. Sorteringen forsvinner også. En B-tree-indeks holder oppføringene for kunde 42 ordnet etter `created_at`, PostgreSQL kan lese dem baklengs for å få de nyeste først, og med `LIMIT` stopper den etter tjue oppføringer uten å røre resten.

Rekkefølgen på kolonnene betyr noe. Den samme indeksen dekker en spørring på `customer_id` alene, fordi det er den første kolonnen, men en rapport over alle gårsdagens ordrer som bare filtrerer på `created_at`, får langt mindre ut av den. I en sammensatt B-tree-indeks er det likhet på de første kolonnene pluss et intervall på den neste som avgrenser den delen av indeksen som blir lest. PostgreSQL 18 kan gjøre skip scan og gjenta søket én gang for hver ulik verdi i den første kolonnen, men planleggeren velger vanligvis det bare når det er få ulike verdier der, og kunde-ID-er finnes det mange av. Betyr rapporten noe, får den sin egen indeks på `created_at`. Snudd om, som `(created_at, customer_id)`, ville indeksen spre ordrene til kunde 42 over hele seg.

Notatet om å optimalisere databasen før du sharder den setter indekser først blant forbedringene det ordner etter gevinst, når planene har navngitt skanningene en ytelseskritisk spørring kjører. Notatets eget eksempel er en partiell indeks, `units_unsold`, som bare inneholder enhetene som fortsatt er til salgs, så den krymper etter hvert som et salg skrider frem, i stedet for å vokse. Notatet regner også ut prisen. På tabellen der var alle 2000 reservasjoner som satte `reserved`, `reserved_at` og `order_id`, HOT-oppdateringer, mens ingen av 2000 oppdateringer som satte `sold`, en kolonne den partielle indeksen leser, var det. Rådet er å fjerne indeksene der `idx_scan` i `pg_stat_user_indexes` blir stående på null.

Prosjektgjennomgangen om et bedre valg av nøkkel viser hvorfor verdiene som havner i en B-tree-indeks, betyr like mye som kolonnene. Tilfeldige UUID-er sprer innsettinger utover indeksen, noe som kan føre til fragmentering. Plattformen gjorde tidsordnede UUID v7 til standard for primærnøklene til entitetene, med et tidsstempel i de første bitene slik at nye identifikatorer har en tidsrekkefølge. Redusert fragmentering og raskere spørringer var designmål; det dokumenterte

resultatet er den konfigurerte nøkkelkonvensjonen, og prosjektgjennomgangen fastslår ingen sammenlignende ytelsesmåling av arbeidslasten før og etter endringen.

Hvor det går galt

En ny indeks stopper skrivingen til tabellen. En vanlig `CREATE INDEX` på en travel tabell blokkerer innsettinger, oppdateringer og slettinger så lenge byggingen varer, og en rutinemessig utrulling blir til nedetid. Bruk `CREATE INDEX CONCURRENTLY` : den skanner tabellen to ganger og venter på transaksjoner som kunne bruke indeksen, så den tar betydelig lengre tid, men skrivingen fortsetter. Den kan ikke kjøres inne i en transaksjonsblokk, så et migreringsverktøy som pakker hvert steg inn i én, må få beskjed om å kjøre akkurat den setningen for seg. Bare én samtidig bygging kan kjøre på en tabell om gangen.

En samtidig bygging feiler og etterlater en ugyldig indeks. En vranglås, eller en duplikatverdi under byggingen av en unik indeks, får `CREATE INDEX CONCURRENTLY` til å feile, men den påbegynte indeksen blir liggende i katalogen, merket som ugyldig. Spørringer ignorerer den, men hver skriving betaler fortsatt for å vedlikeholde den, og en ugyldig unik indeks kan fortsette å håndheve unikhet. `\d` i psql viser den med `INVALID` etter definisjonen. Fjern den og kjør den samtidige byggingen på nytt, eller bygg den om med `REINDEX INDEX CONCURRENTLY` .

Spørringen gjemmer kolonnen inne i en funksjon. En indeks på `email` gjør ingenting for `WHERE lower(email) = 'ana@example.com'` , fordi indeksen inneholder `email` , ikke `lower(email)` . Indekser uttrykket spørringen bruker:

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

Uttrykket beregnes ved hver innsetting og hver oppdatering som ikke er HOT, og derfor anbefaler dokumentasjonen uttrykksindekser der raske oppslag betyr mer enn rask skriving.

Det hoper seg opp indekser som ingen bruker. Hver indeks som er lagt til «for sikkerhets skyld», koster ved hver skriving og kan stenge for HOT-oppdateringer. `pg_stat_user_indexes` teller skanningene som er startet på hver indeks, i `idx_scan` og noterer tidspunktet for den siste i `last_idx_scan` :

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
ORDER  BY relname;
```

Se nøye etter før du sletter. En indeks bak en primærnøkkel eller en unik begrensning gjør jobben sin enten en spørring noen gang leser den eller ikke, og en null sier lite på en server som nettopp har begynt å ta imot trafikk.

Planleggeren overser en god indeks. Etter en stor innlasting henger statistikken etter dataene, estimatene bommer, og en skanning vinner på papiret. Kjør `ANALYZE`; dokumentasjonen kaller det nytteløst å vurdere indeksbruk uten. Test også på realistiske mengder: på hundre rader som får plass på én diskside, slår ingen indeks det å lese den ene siden. Vil du vite om en indeks i det hele tatt kan brukes, setter du `enable_seqscan = off` i en testøkt og kjører `EXPLAIN` igjen. Skanner planen fortsatt, passer ikke betingelsen til indeksen.

Les mer

- Indexes: Introduction — hva en indeks er til, hvordan den opprettes og fjernes, og hva den koster ved hver skriving.
- Index Types — B-tree, Hash, GiST, SP-GiST, GIN og BRIN, og hvilke operatorer hver av dem kan besvare.
- Multicolumn Indexes — hvordan rekkefølgen på kolonnene avgjør hva en B-tree-indeks kan avgrense, og når skip scan kommer inn.
- CREATE INDEX — hele syntaksen, blant annet `CONCURRENTLY`, `INCLUDE`, partielle indekser og opprydding etter en ugyldig bygging.
- Using EXPLAIN — å lese en plan: skanningstyper, estimerte kostnader og de faktiske tallene fra `EXPLAIN ANALYZE`.
- Examining Index Usage — hvorfor `ANALYZE` og ekte data kommer først, og hvordan man tester om en indeks kan brukes.

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 databaseindeks?» 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/database-index/