Engineer

Databasindex

en struktur som hittar rader utan att läsa hela tabellen

Vad det är

Ett databasindex är en separat struktur, lagrad bredvid en tabell, som låter databasen gå direkt till de rader som uppfyller ett villkor i stället för att läsa tabellen från första till sista raden. I vardagligt tal, och i större delen av PostgreSQL-dokumentationen, heter det bara ett index. Dokumentationen jämför det med registret längst bak i en bok: den som letar efter ett ämne slår upp det där och bläddrar direkt till sidorna som anges, i stället för att läsa igenom hela boken.

Ta en tabell med ordrar och villkoret `WHERE customer_id = 42`. Utan ett index på `customer_id` har PostgreSQL bara ett sätt att svara: en sekventiell skanning, som läser varenda rad och behåller dem som matchar. Det kostar lika mycket oavsett om kunden har två ordrar eller tvåtusen, och kostnaden växer med tabellen. Med ett index på kolumnen går PostgreSQL ner några nivåer i ett sökträd till posterna för 42 och hämtar bara de raderna. Ett index skapas med en enda sats, `DROP INDEX` tar bort det igen, och båda kan göras när som helst.

När indexet väl finns håller PostgreSQL det i takt med tabellen vid varje skrivning, och planeraren avgör från fråga till fråga om det lönar sig att använda det i stället för att skanna. Ingenting i SQL-koden namnger indexet, så samma index kan snabba upp en `SELECT`, en `UPDATE` eller `DELETE` med ett sökvillkor och en join.

PostgreSQL har flera sorter, var och en byggd för sin typ av fråga:

- **B-tree** är standard och det som `CREATE INDEX` bygger om inget annat anges. Det besvarar likhet och intervall (`<`, `<=`, `=`, `>=`, `>`), `BETWEEN`, `IN`, `IS NULL` och en `LIKE` där mönstret är förankrat i början. Det håller posterna sorterade och kan därför också leverera raderna i ordning.

- **Hash** lagrar en 32-bitars hashkod för varje värde och besvarar bara likhet.
- **GiST** och **SP-GiST** är ramverk för data som geometri, där frågan är ”överlappar”, ”innehåller” eller ”närmast” snarare än ”är lika med”.
- **GIN** är ett inverterat index för värden som består av många delar, till exempel arrayer, och hittar de rader vars värde innehåller ett visst element.
- **BRIN** lagrar bara en sammanfattning för varje intervall av tabellblock. Det passar en kolumn där värdena följer radernas fysiska ordning, som en tidsstämpel för insättningen ofta gör.

Utöver typen kan ett index formas efter frågan. Ett unikt index avvisar dubbletter. Ett sammansatt index täcker flera kolumner, upp till 32. Ett partiellt index har en `WHERE` -klausul och innehåller bara de rader som uppfyller den. Ett uttrycksindex lagrar ett beräknat värde som `lower(email)` . Och en `INCLUDE` -lista tar med extra kolumner i indexet utan att söka på dem, så att en SQL-fråga kan besvaras från indexet ensamt.

Använd begreppet i ett beslut

Använd ”Vad är ett databasindex?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Hittar databasfrågorna och de saknade indexen bakom ett system som har blivit tyngre för varje år, och åtgärdar dem som spelar roll.

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. 30 dagars eldata på 180 ms — 180 ms för att aggregera en månads mätaravläsningar

30 dagars eldata på 180 ms Hör av er →

Varför det spelar roll

Skillnaden ett index gör växer med tabellen. På en liten tabell är en skanning billig, och planeraren väljer ofta, med rätta, just den: att läsa en handfull sidor i ordning kostar mindre än att gå igenom ett träd för att nå dem. Med tiotals miljoner rader är skillnaden mellan en skanning och ett uppslag skillnaden mellan en sida som laddas och en som får timeout. En SQL-fråga som var snabb vid lanseringen kan bli långsam ett år senare utan att en enda kodrad har ändrats, bara för att tabellen den läser har vuxit.

Index är inte gratis, och notan kommer vid skrivning. Varje insättning lägger till en post i varje index på tabellen. En uppdatering kan bara slippa undan som en heap-only tuple-uppdatering (HOT), där den nya radversionen stannar på samma sida och inte behöver några nya indexposter, och det tillåter PostgreSQL bara när uppdateringen inte ändrar någon kolumn som tabellens index använder. BRIN-

index räknas inte, eftersom de bara lagrar en sammanfattning. Varje index gör alltså den billiga vägen smalare, och därför råder dokumentationen till att ta bort index som sällan eller aldrig används. En tabell med ett dussin index betalar ett dussin gånger vid varje insättning.

Det kostar också att bygga ett index. En vanlig `CREATE INDEX` låter andra sessioner läsa tabellen men blockerar skrivningar till den tills bygget är klart, och dokumentationen varnar för att det på en mycket stor tabell kan ta många timmar.

Till sist är ett index bara så bra som planerarens beslut att använda det, och planeraren arbetar utifrån statistik som `ANALYZE` har samlat in. Frågan att ställa är inte ”finns det ett index?” utan ”används det av de SQL-frågor som spelar roll?”, och `EXPLAIN` är sättet att ta reda på det.

Så ser det ut i praktiken

Kontosidan i en webbutik visar en kunds tjugo senaste ordrar:

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

Med bara en primärnyckel på `id` måste PostgreSQL läsa hela `orders`, behålla kundens rader, sortera dem och kasta alla utom tjugo. `EXPLAIN` visar det: en `Seq Scan` på `orders` med villkoret som `Filter` och en sortering ovanpå. `EXPLAIN ANALYZE` kör dessutom frågan och skriver ut de faktiska radantalen och tiderna bredvid uppskattningarna.

Det index som passar frågan har likhetskolumnen först och sorteringskolumnen därefter:

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

Kör `EXPLAIN` igen så blir skanningen en indexskanning med `orders_customer_created_idx`, där `customer_id = 42` nu står som `Index Cond` i stället för som filter. Sorteringen försvinner också. Ett B-trädindex håller posterna för kund 42 ordnade efter `created_at`, PostgreSQL kan läsa dem baklänges för att få de nyaste först, och med `LIMIT` slutar det efter tjugo poster utan att röra resten.

Kolumnernas ordning spelar roll. Samma index klarar en fråga på enbart `customer_id`, eftersom det är den första kolumnen, men en rapport över alla gårdagens ordrar som bara filtrerar på `created_at` får betydligt mindre ut av det. I ett sammansatt B-trädindex är det likhet på de första kolumnerna plus ett intervall på nästa som avgränsar den del av indexet som läses. PostgreSQL 18 kan göra skip scan och upprepa sökningen en gång för varje distinkt värde i den första kolumnen, men planeraren väljer vanligtvis det bara när den kolumnen har få distinkta värden, och kund-id:n finns det många av. Spelar rapporten roll får den ett eget index på `created_at`. Omvänt, som `(created_at, customer_id)`, skulle kund 42:s ordrar hamna utspridda över hela indexet.

Anteckningen om att optimera databasen innan du shardar den sätter index först bland förbättringarna den rangordnar efter vad de ger för insatsen, när planerna har pekat ut skanningarna som en prestandakritisk SQL-fråga kör. Anteckningens eget exempel är ett partiellt index, `units_unsold`, som bara innehåller de enheter som fortfarande är till salu, så att det krymper medan en rea pågår i stället för att växa. Anteckningen räknar också ut priset. På tabellen där var alla 2000 reservationer som satte `reserved`, `reserved_at` och `order_id` HOT-uppdateringar, medan ingen av 2000 uppdateringar som satte `sold`, en kolumn som det partiella indexet läser, var det. Rådet är att ta bort de index vars `idx_scan` i `pg_stat_user_indexes` blir stående på noll.

Fallstudien om ett bättre val av nyckel visar varför värdena som hamnar i ett B-trädindex betyder lika mycket som kolumnerna. Slumpmässiga UUID:n sprider insättningarna över indexet, vilket kan orsaka fragmentering. Plattformen gjorde tidsordnade UUID v7 till standard för entiteternas primärnycklar, med en tidsstämpel i de inledande bitarna så att nya identifierare får en tidsordning. Minskad fragmentering och snabbare databasfrågor var designmål; det dokumenterade resultatet är den konfigurerade nyckelkonventionen, och fallstudien ger inte belägg för ett prestandatest av en arbetslast med mätningar före och efter.

Var det går fel

Ett nytt index stoppar skrivningarna till tabellen. En vanlig `CREATE INDEX` på en upptagen tabell blockerar insättningar, uppdateringar och borttagningar under hela bygget, och en rutinmässig driftsättning blir ett avbrott. Använd `CREATE INDEX CONCURRENTLY` : det skannar tabellen två gånger och väntar på transaktioner som skulle kunna använda indexet, så det tar betydligt längre tid, men skrivningarna fortsätter. Det kan inte köras inuti ett transaktionsblock, så ett migreringsverktyg som lägger varje steg i ett sådant måste få veta att just den här satsen ska köras för sig. Bara ett samtidigt bygge kan köras på en tabell åt gången.

Ett samtidigt bygge misslyckas och lämnar ett ogiltigt index efter sig. Ett deadlock, eller ett dubblettvärde medan ett unikt index byggs, får `CREATE INDEX CONCURRENTLY` att misslyckas, men det påbörjade indexet ligger kvar i katalogen, markerat som ogiltigt. SQL-frågor ignorerar det, men varje skrivning betalar ändå för att underhålla det, och ett ogiltigt unikt index kan fortsätta att upprätthålla unikhet. `\d` i psql visar det med `INVALID` efter definitionen. Ta bort det och kör det samtidiga bygget igen, eller bygg om det med `REINDEX INDEX CONCURRENTLY` .

Frågan gömmer kolumnen inuti en funktion. Ett index på `email` gör ingenting för `WHERE lower(email) = 'ana@example.com'` , eftersom indexet innehåller `email` , inte `lower(email)` . Indexera uttrycket som frågan använder:

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

Uttrycket beräknas vid varje insättning och varje uppdatering som inte är HOT, och därför rekommenderar dokumentationen uttrycksindex där snabba uppslag betyder mer än snabba skrivningar.

Det samlas index som ingen använder. Varje index som lagts till ”för säkerhets skull” kostar vid varje skrivning och kan stänga vägen för HOT-uppdateringar. `pg_stat_user_indexes` räknar skanningarna som startats på varje index i `idx_scan` och noterar tiden för den senaste i `last_idx_scan` :

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

```
ORDER  BY relname;
```

Titta efter innan du tar bort något. Ett index bakom en primärnyckel eller en unik begränsning gör sitt jobb oavsett om en SQL-fråga någonsin läser det, och en nolla säger lite på en server som nyss har börjat ta emot trafik.

Planeraren förbiser ett bra index. Efter en stor inläsning släpar statistiken efter datan, uppskattningarna slår fel och en skanning vinner på papperet. Kör `ANALYZE` ; dokumentationen kallar det lönlöst att bedöma indexanvändning utan det. Testa också på realistiska volymer: på hundra rader som ryms på en enda disksida slår inget index att läsa den sidan. Vill du veta om ett index över huvud taget kan användas sätter du `enable_seqscan = off` i en testsession och kör `EXPLAIN` igen. Skannar planen fortfarande matchar villkoret inte indexet.

Läs mer

- Indexes: Introduction — vad ett index är till för, hur det skapas och tas bort, och vad det kostar vid varje skrivning.
- Index Types — B-tree, Hash, GiST, SP-GiST, GIN och BRIN, och vilka operatorer var och en kan besvara.
- Multicolumn Indexes — hur kolumnernas ordning avgör vad ett B-trädindex kan avgränsa, och när skip scan kommer in.
- CREATE INDEX — hela syntaxen, bland annat `CONCURRENTLY` , `INCLUDE` , partiella index och hur man städar upp efter ett ogiltigt bygge.
- Using EXPLAIN — att läsa en plan: skanningstyper, uppskattade kostnader och de faktiska siffrorna från `EXPLAIN ANALYZE` .
- Examining Index Usage — varför `ANALYZE` och riktiga data kommer först, och hur man testar om ett index kan användas.

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 ett databasindex?” 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/database-index/