Engineer

Datenbankindex

eine Struktur, die Tabellenzeilen ohne Vollscan findet

Was es ist

Ein Datenbankindex ist eine getrennte, neben einer Tabelle gespeicherte Struktur. Sie führt die Datenbank direkt zu den Zeilen, die eine Bedingung erfüllen, statt die Tabelle von der ersten bis zur letzten Zeile zu lesen. Im Alltag und in den meisten Teilen der PostgreSQL-Dokumentation heißt er einfach Index. Die Dokumentation vergleicht ihn mit dem Register am Ende eines Buchs: Wer ein Thema sucht, schlägt dort nach und öffnet die aufgeführten Seiten, statt den ganzen Band durchzublättern.

Nehmen Sie eine Bestelltabelle und die Bedingung `WHERE customer_id = 42`. Ohne Index auf `customer_id` hat PostgreSQL nur eine Antwortmöglichkeit: einen sequenziellen Scan, der jede Zeile liest und die passenden behält. Das kostet gleich viel, ob der Kunde zwei oder zweitausend Bestellungen hat, und wächst mit der Tabelle. Mit einem Index auf der Spalte durchläuft PostgreSQL wenige Ebenen eines Suchbaums bis zu den Einträgen für 42 und ruft nur diese Zeilen ab. Ein Index entsteht mit einer einzigen Anweisung, `DROP INDEX` entfernt ihn wieder. Beides ist jederzeit möglich.

Sobald ein Index existiert, hält PostgreSQL ihn bei jedem Schreibvorgang mit der Tabelle synchron. Der Abfrageplaner entscheidet für jede Abfrage, ob seine Nutzung günstiger ist als ein Scan. Das SQL benennt den Index nirgends. Ein Index kann deshalb ein `SELECT`, ein `UPDATE` oder `DELETE` mit Suchbedingung sowie einen Join beschleunigen.

PostgreSQL bringt mehrere Arten mit, jeweils für eine andere Form von Frage:

- **B-Tree** ist der Standard und das Ergebnis von `CREATE INDEX`, solange nichts anderes angegeben wird. Er beantwortet Gleichheits- und Bereichsvergleiche (`<`, `<=`, `=`, `>=`, `>`), `BETWEEN`, `IN`, `IS NULL` und ein `LIKE`, dessen Muster am Anfang verankert ist. Die Einträge bleiben geordnet, sodass er Zeilen auch bereits sortiert zurückgeben kann.
- **Hash** speichert zu jedem Wert einen 32-Bit-Hashcode und beantwortet nur Gleichheit.
- **GiST** und **SP-GiST** sind Frameworks für Daten wie Geometrien, bei denen die Frage „überlappt", „enthält" oder „am nächsten zu" statt „gleich" lautet.
- **GIN** ist ein invertierter Index für mehrteilige Werte, etwa Arrays, und findet die Zeilen, deren Wert ein bestimmtes Element enthält.
- **BRIN** hält nur eine Zusammenfassung für jeden Bereich von Tabellenblöcken. Er passt zu einer Spalte, deren Werte der physischen Zeilenreihenfolge folgen, wie es ein Einfügezeitstempel häufig tut.

Zusätzlich zu seinem Typ lässt sich ein Index auf die Frage zuschneiden. Ein eindeutiger Index weist doppelte Werte zurück. Ein mehrspaltiger Index deckt mehrere Spalten ab, bis zu 32. Ein partieller Index besitzt eine `WHERE`-Klausel und enthält nur die Zeilen, die sie erfüllen. Ein Ausdrucksindex speichert einen berechneten Wert wie `lower(email)`. Eine `INCLUDE`-Liste nimmt zusätzliche Spalten auf, ohne auf ihnen zu suchen. So kann eine Abfrage allein aus dem Index beantwortet werden.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist ein Datenbankindex?" für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Die Abfragen und fehlenden Indizes hinter einem von Jahr zu Jahr schwerfälligeren System finden und die entscheidenden Probleme beheben.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 30 Tage Stromdaten in 180 ms — 180 ms für die Aggregation eines Monats an Zählerständen

30 Tage Stromdaten in 180 ms Kontakt aufnehmen →

Warum es wichtig ist

Der Unterschied durch einen Index wächst mit der Tabelle. Bei einer kleinen Tabelle ist ein Scan billig. Der Planer bevorzugt ihn oft zu Recht: Wenige Seiten der Reihe nach zu lesen kostet weniger, als sie über einen Baum zu erreichen. Bei mehreren zehn Millionen Zeilen ist der Unterschied zwischen Scan und gezieltem

Nachschlagen der zwischen einer ladenden Seite und einer Zeitüberschreitung. Eine beim Start schnelle Abfrage kann ein Jahr später langsam sein, ohne dass sich eine Codezeile geändert hat, nur weil ihre Tabelle gewachsen ist.

Indizes sind nicht kostenlos. Die Rechnung kommt bei Schreibvorgängen. Jedes Einfügen ergänzt jeden Index der Tabelle um einen Eintrag. Eine Aktualisierung vermeidet das nur als Heap-only-Tuple-Update (HOT), bei dem die neue Zeilenversion auf derselben Seite bleibt und die vorhandenen Indexeinträge weiterhin zu ihr führen. PostgreSQL erlaubt das nur, wenn alle von Tabellenindizes verwendeten Spalten unverändert bleiben. Ausgenommen sind BRIN-Indizes, weil sie nur Zusammenfassungen halten. Jeder Index verengt also den günstigen Pfad. Deshalb rät die Dokumentation, selten oder nie verwendete Indizes zu entfernen. Eine Tabelle mit einem Dutzend Indizes bezahlt bei jedem Einfügen ein Dutzend Mal.

Auch der Aufbau kostet etwas. Ein gewöhnliches `CREATE INDEX` erlaubt anderen Sitzungen das Lesen der Tabelle, blockiert aber ihre Schreibvorgänge bis zum Ende des Aufbaus. Die Dokumentation warnt, dass das bei einer sehr großen Tabelle viele Stunden dauern kann.

Schließlich ist ein Index nur so gut wie die Entscheidung des Planers, ihn zu nutzen. Der Planer arbeitet mit durch `ANALYZE` erhobenen Statistiken. Die Frage lautet nicht „Gibt es einen Index?“, sondern „Nutzen ihn die wichtigen Abfragen?“. Das zeigt `EXPLAIN` .

Wie es in der Praxis aussieht

Die Kontoseite eines Onlineshops listet die zwanzig neuesten Bestellungen eines Kunden auf:

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

Mit nur einem Primärschlüssel auf `id` muss PostgreSQL die ganze Tabelle `orders` lesen, die Zeilen des Kunden behalten, sie sortieren und alle bis auf zwanzig verwerfen. `EXPLAIN` zeigt das: einen `Seq Scan` auf `orders` mit der

Bedingung als `Filter` und einem Sortierschritt darüber. `EXPLAIN ANALYZE` führt die Abfrage außerdem aus und zeigt tatsächliche Zeilenzahlen und Laufzeiten neben den Schätzungen.

Der passende Index setzt die Gleichheitsspalte zuerst und die Sortierspalte danach:

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

Führen Sie `EXPLAIN` erneut aus. Der Scan wird zu einem Indexscan über `orders_customer_created_idx`, bei dem `customer_id = 42` jetzt als `Index Cond` statt als Filter erscheint. Auch der Sortierschritt entfällt. Ein B-Tree hält die Einträge für den Kunden 42 nach `created_at` geordnet. PostgreSQL kann sie rückwärts lesen, um die neuesten zuerst zu erhalten. Mit einem `LIMIT` stoppt es nach zwanzig Einträgen, statt den Rest zu berühren.

Die Spaltenreihenfolge ist wichtig. Derselbe Index bedient eine Abfrage allein auf seiner führenden Spalte `customer_id`. Ein Bericht über alle gestrigen Bestellungen, der nur auf `created_at` filtert, profitiert deutlich weniger: In einem mehrspaltigen B-Tree begrenzen Gleichheit auf den führenden Spalten und ein Bereich auf der nächsten den gelesenen Indexabschnitt. PostgreSQL 18 kann Skip Scans ausführen und dabei die Suche für jeden verschiedenen führenden Wert wiederholen. Der Planer wählt das aber meist nur bei wenigen unterschiedlichen Werten der führenden Spalte. Kunden-IDs haben viele. Ist der Bericht wichtig, bekommt er einen eigenen Index auf `created_at`. Umgedreht zu `(created_at, customer_id)` würde der Index die Bestellungen des Kunden 42 über seine gesamte Struktur verteilen.

Die Notiz zur Optimierung der Datenbank vor dem Sharding setzt Indizes an die erste Stelle der nach Ertrag geordneten Korrekturen, nachdem `EXPLAIN (ANALYZE, BUFFERS)` die Scans einer stark beanspruchten Abfrage gezeigt hat. Ihr Beispiel ist ein partieller Index `units_unsold`, der nur die noch angebotenen Einheiten enthält. Im Verlauf eines Verkaufs schrumpft er deshalb, statt zu wachsen. Die Notiz beziffert auch den Preis. In ihrer Tabelle waren 2.000 Reservierungen, die `reserved`, `reserved_at` und `order_id` setzten, allesamt HOT-Updates. Von 2.000 Aktualisierungen an `sold`, einer vom partiellen Index gelesenen Spalte, war keine ein HOT-Update. Ihr Rat ist, die Indizes zu entfernen, deren `idx_scan` in `pg_stat_user_indexes` bei null bleibt.

Die Fallstudie zur besseren Wahl eines Schlüssels zeigt, warum die Werte in einem B-Tree genauso wichtig sind wie die Spalten. Zufällige UUIDs verteilen Einfügungen über den Index, was Fragmentierung verursachen kann. Die Plattform machte zeitlich geordnete UUID v7 zum Standard für Primärschlüssel von Entitäten. Ein Zeitstempel in den führenden Bits gibt neuen Kennungen eine zeitliche Ordnung. Weniger Fragmentierung und schnellere Abfragen waren Entwurfsziele. Das dokumentierte Ergebnis ist die konfigurierte Schlüsselkonvention. Die Fallstudie belegt keinen Vorher-nachher-Benchmark einer Arbeitslast.

Wo es schiefgeht

Ein neuer Index stoppt die Schreibvorgänge der Tabelle. Ein gewöhnliches `CREATE INDEX` auf einer stark genutzten Tabelle blockiert Einfügungen, Aktualisierungen und Löschungen während des gesamten Aufbaus. Eine routinemäßige Bereitstellung wird so zum Ausfall. Verwenden Sie `CREATE INDEX CONCURRENTLY` : Es liest die Tabelle zweimal und wartet auf Transaktionen, die den Index verwenden könnten. Es dauert dadurch erheblich länger, aber Schreibvorgänge laufen weiter. Innerhalb eines Transaktionsblocks ist es nicht möglich. Ein Migrationswerkzeug, das jeden Schritt in einen solchen Block einpackt, muss deshalb angewiesen werden, diese Anweisung allein auszuführen. Auf einer Tabelle darf zu einem Zeitpunkt nur ein nebenläufiger Indexaufbau laufen.

Ein nebenläufiger Aufbau scheitert und hinterlässt einen ungültigen Index. Ein Deadlock oder ein doppelter Wert beim Aufbau eines eindeutigen Index lässt `CREATE INDEX CONCURRENTLY` scheitern. Der begonnene Index bleibt aber als ungültig markiert im Katalog. Abfragen ignorieren ihn, doch jeder Schreibvorgang bezahlt weiter seine Pflege. Ein ungültiger eindeutiger Index kann weiterhin Eindeutigkeit erzwingen. psqls `\d` zeigt nach seiner Definition `INVALID` an. Entfernen Sie ihn und starten Sie den nebenläufigen Aufbau erneut oder bauen Sie ihn mit `REINDEX INDEX CONCURRENTLY` neu auf.

Die Abfrage versteckt die Spalte in einer Funktion. Ein Index auf `email` hilft nicht bei `WHERE lower(email) = 'ana@example.com'` , weil er `email` enthält, nicht `lower(email)` . Indizieren Sie den Ausdruck der Abfrage:

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

Der Ausdruck wird bei jedem Einfügen und jedem Update ohne HOT berechnet. Deshalb empfiehlt die Dokumentation Ausdrucksindizes dort, wo Abrufgeschwindigkeit wichtiger ist als Schreibgeschwindigkeit.

Ungenutzte Indizes häufen sich. Jeder „vorsichtshalber“ hinzugefügte Index kostet bei jedem Schreibvorgang und kann HOT-Updates ausschließen. `pg_stat_user_indexes` zählt die begonnenen Scans jedes Index in `idx_scan` und hält den Zeitpunkt des letzten in `last_idx_scan` fest:

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

Prüfen Sie vor dem Löschen. Ein Index hinter einem Primärschlüssel oder einer Eindeutigkeitsbedingung erfüllt seine Aufgabe auch dann, wenn nie eine Abfrage ihn liest. Ein Zähler von null sagt auf einem Server wenig, der gerade erst Datenverkehr erhält.

Der Planer ignoriert einen guten Index. Nach einer Massenladung hinken die Statistiken den Daten hinterher. Die Schätzungen sind falsch, und auf dem Papier gewinnt ein Scan. Führen Sie `ANALYZE` aus. Die Dokumentation nennt es „aussichtslos“, die Indexnutzung ohne diesen Schritt zu beurteilen. Prüfen Sie auch mit realistischen Datenmengen: Bei hundert Zeilen auf einer einzigen Datenträgerseite schlägt kein Index das Lesen dieser Seite. Um zu erfahren, ob ein Index überhaupt nutzbar ist, setzen Sie in einer Testsitzung `enable_seqscan = off` und führen `EXPLAIN` erneut aus. Führt der Plan weiter einen Scan aus, passt die Bedingung nicht zum Index.

Weiterführende Lektüre

- Indizes: Einführung — wozu ein Index dient, wie er erstellt und entfernt wird und welchen Zusatzaufwand er bei jedem Schreibvorgang verursacht.
- Indextypen — B-Tree, Hash, GiST, SP-GiST, GIN und BRIN und die jeweils unterstützten Operatoren.
- Mehrspaltige Indizes — wie die Spaltenreihenfolge den eingrenzbaren Bereich eines B-Trees bestimmt und wann Skip Scan greift.
- CREATE INDEX — die vollständige Syntax einschließlich `CONCURRENTLY`, `INCLUDE`, partieller Indizes und der Wiederherstellung nach einem ungültigen Aufbau.

- EXPLAIN verwenden — einen Plan lesen: Scanarten, geschätzte Kosten und die tatsächlichen Zahlen aus `EXPLAIN ANALYZE`.
- Indexnutzung untersuchen — warum `ANALYZE` und echte Daten zuerst kommen und wie sich die mögliche Indexnutzung prüfen lässt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Bringen Sie einen repräsentativen Datensatz, die Fragen, die er beantworten muss, und den aktuellen Engpass mit. Wir können vereinbaren, wie Qualität, Durchsatz und Wiederherstellung gemessen werden.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist ein Datenbankindex?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/database-index/