Engineer

データベースインデックス

テーブル全体を走査せずに行を探すための構造

どのようなものか

データベースインデックスは、テーブルとは別に、そのそばへ保存する構造です。データベースが、最初から最後まで全行を読む代わりに、条件に合う行へ直接進めるようにします。普段の会話でも、PostgreSQLの文書の大半でも、単にインデックスと呼びます。文書は、本の巻末の索引にたとえています。ある話題を探す読者は、本全体をめくる代わりに、索引を調べ、そこに載るページを開きます。

注文テーブルと、`WHERE customer_id = 42`という条件を考えます。`customer_id`にインデックスがなければ、PostgreSQLの答え方は、シーケンシャルスキャンです。すべての行を読み、合う行を残します。その顧客の注文が二件でも二千件でも、コストは同じで、テーブルが大きくなるほど増えます。列にインデックスがあれば、検索木の数階層をたどって42の項目へ進み、その行だけを取得します。作成は一つの文で済み、`DROP INDEX`で削除でき、どちらもいつでも行えます。

インデックスを作ると、PostgreSQLは、書き込みのたびに、テーブルと一致するように保ちます。クエリープランナーは、クエリーごとに、インデックスを使う方が走査より処理コストが低いかを判断します。SQL内でインデックスの名前を指定する必要はないので、一つのインデックスが、`SELECT`、検索条件のある`UPDATE`や`DELETE`、結合を速くできます。

PostgreSQLには、問いの形に合わせて作った、複数の種類があります。

- **B-tree**は既定の種類で、別の指定がなければ、`CREATE INDEX`が作ります。等値と範囲の比較（`<`、`<=`、`=`、`>=`、`>`）、`BETWEEN`、`IN`、`IS NULL`、先頭が固定されたパターンを使う`LIKE`に答えます。項目を順番に保持するので、行を整列済みで返すこともできます。
- **Hash**は、各値の32ビットのハッシュコードを保存し、等値だけに答えます。

- **GiST**と**SP-GiST**は、幾何データなどを扱う枠組みです。「等しい」ではなく、「重なる」「含む」「最も近い」といった問いに使います。
- **GIN**は、配列など、多数の部分からなる値向けの転置インデックスで、指定した要素を含む値を持つ行を探します。
- **BRIN**は、テーブルのブロックの範囲ごとに、要約だけを保持します。挿入時刻のタイムスタンプのように、値が行の物理的な順番に沿う列に向いています。

種類に加え、問いに合わせて形を変えられます。一意インデックスは、重複値を拒否します。複合インデックスは、最大32の列を対象にします。部分インデックスは `WHERE` 句を持ち、それを満たす行だけを保持します。式インデックスは、`lower(email)` のような計算結果を保存します。`INCLUDE` リストは、検索には使わない追加の列をインデックスに載せ、インデックスだけでクエリーへ答えられるようにします。

概念を判断に生かす

「データベースインデックスとは？」を実際の判断に生かしましょう。その概念を、開発または購入するシステムに結び付けて考えるお手伝いをします。 年々重くなってきたシステムのクエリと不足するインデックスを調べ、効果のある箇所を改善します。

サービスを見る お問い合わせ →

用語の背後にある前提を検証する

解説とリンク先の実績を参考に、プロジェクトについての質問を整理してください。ツールを選ぶ前に、前提を一緒に検討できます。 30日分の電力データを180 msで処理 — 180 ms 1か月分のメーター値を集計

30日分の電力データを180 msで処理 お問い合わせ →

なぜ重要なのか

インデックスの効果は、テーブルとともに大きくなります。小さなテーブルなら走査のコストが低く、プランナーがそちらを選ぶのは、しばしば正しい判断です。数ページを順番に読む方が、それを得るために木をたどるより処理コストが低いためです。数千万行になると、全走査と索引検索の差は、ページが表示されるか、タイムアウトするかの差になります。公開時は速かったクエリーが、一行もコードを変えずに、一年後には遅くなることがあります。読むテーブルが大きくなっただけでも起こります。

インデックスには代償があり、その負担は書き込みに現れます。挿入のたびに、そのテーブルのすべてのインデックスへ項目を追加します。更新でそれを避けられるのは、heap-only tuple（HOT）更新の場合だけです。新しい行バージョンが同じページにとどまり、既存のインデックス項目から、その行へ到達できます。PostgreSQLがそれを認めるのは、テーブルのインデックスが使う列を、更新が一つも変えない場合だけです。ただし、要約しか持たないBRINインデックスは

除きます。インデックスを一つ増やすごとに、低コストで処理できる条件が狭くなります。そのため、文書は、ほとんど使わない、またはまったく使わないインデックスの削除を勧めます。十二のインデックスを持つテーブルは、挿入ごとに十二個分の更新コストを負担します。

作成にも処理コストがかかります。普通の `CREATE INDEX` では、ほかのセッションからテーブルを読めますが、作成が終わるまで書き込みをブロックします。非常に大きなテーブルでは、何時間もかかり得ると、文書は警告しています。

最後に、インデックスは、プランナーが使うと判断してこそ役立ちます。判断には、 `ANALYZE` が集めた統計を使います。問うべきなのは、「インデックスがあるか」ではなく、「重要なクエリーが使っているか」です。それを知る方法が、 `EXPLAIN` です。

実際の活用例

オンラインショップのアカウントページが、顧客の直近二十件の注文を表示します。

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

`id` の主キーしかなければ、PostgreSQLは、 `orders` 全体を読み、顧客の行を残し、並べ替え、二十件以外を捨てます。 `EXPLAIN` には、それが現れます。 `orders` の `Seq Scan` に条件が `Filter` として付き、その上にソートがあります。 `EXPLAIN ANALYZE` は、クエリーも実行し、推定値の横に、実際の行数と所要時間を表示します。

この問いに合うインデックスは、等値条件の列を先、並べ替えの列を次に置きます。

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

もう一度 `EXPLAIN` を実行すると、走査は、 `orders_customer_created_idx` を使うインデックススキャンになり、 `customer_id = 42` は、フィルターではなく、 `Index Cond` として現れます。ソートもなくなります。B-treeは、顧客42の項目を `created_at` 順に保ちます。PostgreSQLは、逆向きに読んで新しいものを先に取得でき、 `LIMIT` があれば、残りに触れず、二十項目で止まります。

列の順番は重要です。同じインデックスは、先頭列の `customer_id` だけを使うクエリーにも対応します。しかし、昨日の全注文を調べ、 `created_at` だけで絞るレポートには、はるかに効果が小さくなります。複合B-treeで、読む部分を狭めるのは、先頭からの列の等値条件と、その次の列の範囲条件だからです。PostgreSQL 18は、先頭列の異なる値ごとに検索を繰り返す、スキップスキャンを使えます。ただし、プランナーが通常それを選ぶのは、先頭列の値の種類が少ない場合だけで、顧客IDには多くの種類があります。重要なレポートなら、 `created_at` に専用のインデックスを作ります。 `(created_at, customer_id)` と逆にすると、顧客42の注文が、インデックス全体へ散らばります。

シャーディングする前にデータベースを最適化するというノートは、 `EXPLAIN (ANALYZE, BUFFERS)` で負荷の高いクエリーの走査を特定した後、効果の順に並べた改善策の先頭に、インデックスを置きます。例は、まだ販売中の商品単位だけを保持する部分インデックス、 `units_unsold` です。販売が進むにつれて、大きくなるのではなく、小さくなります。ノートは、その代償も数えています。対象のテーブルでは、 `reserved` 、 `reserved_at` 、 `order_id` を設定する2,000回の予約取得は、すべてHOT更新でした。一方、部分インデックスが読む列の `sold` を設定する2,000回の更新には、HOT更新が一つもありませんでした。 `pg_stat_user_indexes` 内の `idx_scan` がゼロのままのインデックスは、削除することを勧めています。

よりよいキーの選択という事例は、B-treeへ入る値が、列と同じくらい重要な理由を示します。ランダムなUUIDは、インデックス全体に挿入を散らし、断片化を引き起こすことがあります。プラットフォームは、エンティティーの主キーの既定を、時刻順のUUID v7にしました。先頭のビットにタイムスタンプを持たせ、新しい識別子が時刻の順序を持つようにしています。断片化の軽減とクエリーの高速化は、設計上の狙いでした。記録された結果は、キーの規約を設定したことで、この事例からは、実際の負荷を使った変更前後の性能比較は確認できません。

問題が起きるところ

インデックスの追加で、テーブルの書き込みが止まります。 書き込みの多いテーブルで普通の `CREATE INDEX` を実行すると、作成中ずっと、挿入、更新、削除をブロックし、定例のデプロイがサービス停止になります。 `CREATE INDEX CONCURRENTLY` を使います。テーブルを二回走査し、インデックスを使い得るトランザクションを待つので、大幅に時間が増えますが、書き込みは続けられます。トランザクションブロック内では実行できません。各段階をトランザクションで包む移行ツールには、この文だけを単独で実行するように指定する必要があります。一つのテーブルで、同時に実行できる並行作成は一つだけです。

並行作成に失敗し、無効なインデックスが残ります。 デッドロックや、一意インデックスの作成中の重複値によって、 `CREATE INDEX CONCURRENTLY` が失敗します。しかし、作り始めたインデックスは、無効と印を付けたままカタログに残ります。クエリーは無視しますが、すべての書き込みで、それを維持する処理コストがかかり続けます。無効な一意インデックスは、一意性を強制し続ける場合もあります。psqlの `\d` は、定義の後ろに `INVALID` と表示します。削除して並行作成をやり直すか、 `REINDEX INDEX CONCURRENTLY` で再作成します。

クエリーが、列を関数の中に隠します。 `email` のインデックスは、 `WHERE lower(email) = 'ana@example.com'` には役立ちません。保持しているのは `email` で、 `lower(email)` ではないためです。クエリーが使う式に、インデックスを作ります。

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

式は、挿入のたびと、HOTではない更新のたびに計算します。そのため、文書は、書き込み速度より取得速度が重要な場所で、式インデックスを勧めています。

誰も使わないインデックスが、積み重なります。 「念のため」に足すたびに、すべての書き込みに処理コストがかかり、HOT更新を不可能にする場合もあります。 `pg_stat_user_indexes` は、各インデックスで始めた走査を `idx_scan` で数え、最後の走査時刻を `last_idx_scan` へ記録します。

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

```
ORDER  BY relname;
```

削除する前に、確認します。主キーや一意制約を支えるインデックスは、クエリーから一度も読まれなくても、役割を果たします。また、トラフィックを受け始めたばかりのサーバーでは、ゼロという数値に大きな意味はありません。

プランナーが、よいインデックスを無視します。 大量読み込みの後は、統計がデータに追い付かず、推定が外れ、計算上は全走査が有利になります。`ANALYZE` を実行します。文書は、それなしでインデックスの利用を評価することを、「見込みのない試み」と呼びます。現実的な量でもテストします。ディスクの一ページに収まる百行なら、そのページを読むより速いインデックスはありません。そもそも使えるかを知るには、テスト用のセッションで `enable_seqscan = off` を設定し、`EXPLAIN` を再実行します。それでも走査する計画なら、条件がインデックスに合っていません。

参考資料

- インデックス：概要 — インデックスの用途、作成と削除、書き込みごとに加わる負担です。
- インデックスの種類 — B-tree、Hash、GiST、SP-GiST、GIN、BRINと、それぞれが答えられる演算子です。
- 複合インデックス — B-treeで絞り込める範囲を列の順番がどう決めるかと、スキップスキャンが適用される場合です。
- CREATE INDEX — 構文全体です。`CONCURRENTLY`、`INCLUDE`、部分インデックス、無効になった作成からの回復も含みます。
- EXPLAINの利用 — 走査の種類、推定コスト、`EXPLAIN ANALYZE` の実測値など、実行計画の読み方です。
- インデックス利用の確認 — `ANALYZE` と現実のデータを先に用意する理由と、インデックスを使えるかを試す方法です。



用語集

実際の制約に合わせて選ぶ

代表的なデータセット、そこから答えを得たい問い、現在のボトルネックをお知らせください。品質、処理量、復旧の測り方を合意できます。

料金を見る お問い合わせ →

用語を実務に結び付ける

「データベースインデックスとは？」がお客様の組織にどう当てはまるかをご相談ください。現在のシステムの具体例が一つあれば、話を始められます。

サービスを見る

welcome@engineer.company +45 50 10 61 84 デンマーク、コペンハーゲン

このページを持ち歩く

連絡先カード

心を込めて丁寧に制作： D.

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

著作権 © 2025 – 現在 · Engineer ApS · すべての権利を留保します。

印刷元： https://engineer.company/ja/glossary/database-index/