Engineer

CTE

外側のクエリーからテーブルのように読める、名前付きのサブクエリー

どのようなものか

CTEはcommon table expressionの略で、日本語では共通テーブル式と呼びます。文の先頭にあるWITH句でクエリーに名前を付け、文の残りから、その行をテーブルのように読めるようにします。一つの文で複数のCTEを定義でき、各CTEは、それより前に名前を付けたものを読めます。最後のクエリーで、それらを組み合わせます。PostgreSQLのマニュアルは、「一つの問い合わせの間だけ存在する一時テーブル」と説明しています。この捉え方は適切ですが、注意点が一つあります。その名前で何かを保存するわけではなく、次の文からは見えません。各CTEはSELECTか、PostgreSQLではINSERT、UPDATE、DELETE、MERGEにもできます。WITH句を付ける文も、そのいずれかにできます。

「直近30日間のある日に、そのメーター自身の普段の使用量の三倍を超える使用量を記録したのは、どのメーターか」という問いは、自然に段階へ分けられます。

```
WITH daily AS (
  SELECT meter_id, read_at::date AS day, sum(kwh) AS kwh
  FROM readings
  WHERE read_at >= current_date - 30
  GROUP BY meter_id, read_at::date
),
typical AS (
  SELECT meter_id, avg(kwh) AS avg_kwh
  FROM daily
  GROUP BY meter_id
)
SELECT d.meter_id, d.day, d.kwh, t.avg_kwh
FROM daily AS d
JOIN typical AS t USING (meter_id)
WHERE d.kwh > 3 * t.avg_kwh
ORDER BY d.day, d.meter_id;
```

最初の段階は、各メーターの読み取り値を日ごとに合計します。二つ目は、日別の合計をメーターごとに平均し、最後に、それぞれの日を、そのメーター自身の平均と比較します。同じ答えは、入れ子のサブクエリーでも書けます。ただし、日別合計が、片方をもう片方の内側に置く形で二回現れ、読む人は内側から外側へ追う必要があります。WITHなら、上から下へ順に論理を読め、検査中は各段階だけを実行できます。

RECURSIVEを加えると、CTEが自分の出力を読めます。通常のSQLでは、ほかの方法で表せない部分です。再帰CTEは、最初のクエリー、UNIONかUNION ALL、CTE自身を参照するクエリーという順で構成します。PostgreSQLは、最初のクエリーを実行し、その行を作業テーブルに保持します。その後、最新の行を使って二つ目のクエリーを繰り返し、新しい行が加わらなくなったら止まります。一つの見出しの下のカテゴリーなど、深さが分からない木構造を、一つの文でたどる仕組みです。

```
WITH RECURSIVE subtree AS (
  SELECT id, parent_id, name, 1 AS depth
  FROM categories
  WHERE id = 42
  UNION ALL
  SELECT c.id, c.parent_id, c.name, s.depth + 1
  FROM categories AS c
  JOIN subtree AS s ON c.parent_id = s.id
)
SELECT id, name, depth FROM subtree ORDER BY depth, name;
```

概念を判断に生かす

「CTE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

第一の理由は、読みやすさです。四段の入れ子を持つ百行の分析クエリーは、レビューしにくく、変更はさらに困難です。同じ論理を名前付きの段階につなぎ直せば、レビューする人は一段階を一つの文として捉えられ、書く人にはコメント

を置く場所ができます。誰でも、次の段階を信用する前に、一つの段階だけを実行し、行が正しそうか確かめられます。

第二の理由は、データベースが段階をどう評価するかです。この詳細は、時代とともに変わりました。PostgreSQLは通常、一つの文の実行につき、WITHのクエリーを一回だけ評価します。その文が複数回参照していても同じなので、二か所で必要な高コストの計算も、一回で済みます。その保証の代償は、複数回参照するCTEの中へ、外側のクエリーのフィルターをプランナーが押し込めないことです。そのフィルターは、一つの利用箇所には適用しても、ほかには適用すべきでないためです。CTE全体を計算してから、不要な行を捨てます。

PostgreSQL 12より前は、すべてのCTEがこの扱いでした。クエリーの残りより先に評価し、その中へ統合しませんでした。バージョン12で既定の動作が変わり、再帰ではなく、副作用がなく、一度だけ参照するCTEは、外側のクエリーへ組み込むようになりました。プランナーが二つの階層をまとめて最適化し、外側のフィルターを、元のテーブルのインデックスまで押し込めます。
MATERIALIZEDは以前の振る舞いを強制します。NOT MATERIALIZEDは、複数回使うCTEでも組み込みを強制し、処理を二回行う危険を伴います。SQLiteも、同じ二つのキーワードを、拘束力のないヒントとして受け入れ、PostgreSQLから着想を得たとしています。ほかのエンジンは、それぞれに判断するため、実行計画で確かめます。

第三の理由は、CTEでデータを変更できることです。WITH句内のDELETEがRETURNINGで出力し、それを主文のINSERTへ渡せば、一つのテーブルから別のテーブルへ、一つの文で行を移せます。全体を実行するか、一切実行しないかのどちらかです。データを変更するCTEは、出力を何かが読むかどうかに関係なく、必ず一回、最後まで実行します。そのため、結果を記述するためというより、複数の書き込みをまとめて行うための道具になります。

実際の活用例

PostgreSQLの性能を10倍にした仕事の対象は、利用者アカウント、テナント、場所、日ごとの電力読み取り値を持つデータベースでした。データが増えるにつれて遅くなっていました。対策は三つです。対象を絞ったインデックスとGiSTインデックス、読み取り値テーブルの時間ベースのパーティション分割、クエリーの書き直しを行いました。よくアクセスするデータにはCTEとマテリアライズドビューを使い、テーブル全体の走査を避けました。組み替える価値のある結合やサブクエリーは、EXPLAIN ANALYZEで見つけました。応答時間が元の十分の一になり、リソース使用量が40%減ったのは、CTEだけではなく、この三つを組み

合わせた成果です。二つを組み合わせた点にも意味があります。CTEは、一つの文の間だけ中間結果に名前を付けますが、マテリアライズドビューは、更新するまで、多数の文から読める結果を保存します。

CTEのもう一つの面である読みやすさは、電力とGISデータの500本の分析レポートを支えました。すべてのレポートを、PostgreSQLとBigQueryに対する純粋なSQLで書きました。座標、メーターID、時刻付きの使用量を持つスマートメーターのデータを対象とし、スクリプトでは、CTE、ウィンドウ関数、サブクエリー、地理空間の結合を組み合わせました。各レポートには、同僚が分析を再現し、監査できるよう、注記付きのSQLと、GPS座標の欠落やメーター値の破損など、よくあるデータ品質の問題についての説明を付けました。段階を追って読めるSQLは、共有しやすいものです。レポートは部門をまたいだ標準の参照資料となり、運用計画の効率を10–15%高めるのに役立ちました。この種のSQLへ注記を付けられるのも、段階に名前があるためです。各段階の上に、何を取り除き、何を前提にするかを書けます。

問題が起きるところ

すべてを処理してから、フィルターを適用します。 二回参照するCTEや、MATERIALIZEDを指定したCTEは、外側のクエリーが絞り込む前に、全体を計算します。そのコピーには、インデックスがありません。少数の行に触れるはずのクエリーが、テーブル全体を読みます。実行計画を見ます。CTEが独立した段階として現れ、行数が最終的に返す行数を大きく上回るなら、NOT MATERIALIZEDを試すか、CTEを一回だけ参照する形に書き直します。高コストの関数を呼ぶCTEでは、実体化する形を保ちます。参照のたびではなく、一行につき一回だけ関数を実行するためです。

再帰クエリーが、止まりません。 互いを親として登録した二つのカテゴリーなど、循環のあるデータでは、UNION ALLを使う再帰CTEが、サーバーの許す限り巡回し続けます。UNIONが役立つのは、行全体が重複する場合だけです。深さの列を加えると、その条件はほとんど満たされません。訪問済みのキーの経路を配列で追い、同じキーが再び現れたら止めるか、PostgreSQL 14で追加したCYCLE句を使います。外側のクエリーのLIMITも、テスト用の工夫にはなりますが、マニュアルは、本番で頼るべきではないとしています。SQL Serverは、既定で再帰を100階層までに制限し、超えると文を取り消します。同じバグでも、動き続けずに止まります。

文の残りから、変更が見えません。 データを変更するCTEと主クエリーは、同じスナップショット上で並行して実行します。そのため、対象テーブルを直接読む主クエリーには、変更前の行が見えます。両者の間で情報を渡す唯一の経路は、

RETURNINGです。一つの文の二つの部分が、同じ行を変更する場合は、さらに問題になります。有効になる変更は一つだけで、どちらになるかは予測しにくいためです。変更した行は、CTEのRETURNINGの出力から読み、一つの文の中では、各行を書き込む部分を一つに限ります。

アップグレードで、実行計画が変わります。 PostgreSQL 11以前に書いたコードは、CTEが先に評価されることを、望ましくない計画への意図的な障壁として使っている場合があります。12以降へ更新すると、同じCTEを外側のクエリーへ組み込み、プランナーが別の計画を選びます。通常は速くなりますが、ときには大幅に遅くなります。遅い文のEXPLAIN出力を、更新前後で比較し、障壁を意図した箇所にはMATERIALIZEDを加えます。

参考資料

- 7.8. WITHクエリー（共通テーブル式） — CTEのマニュアルの章です。再帰、循環の検出、実体化、データを変更する文を説明します。
- E.23. リリース12 — 一度だけ使うCTEを、外側のクエリーへ組み込むようになったリリースです。
- E.25. リリース14 — SQL標準のSEARCH句とCYCLE句を追加したリリースです。
- WITH句 — SQLiteでの実装です。実体化のヒントと、その着想元も説明します。
- WITH common_table_expression（Transact-SQL） — SQL ServerのCTEの規則です。再帰のMAXRECURSION上限も含みます。

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

「CTE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/cte/