CTE: WITH i zapytania rekurencyjne

> CTE to nazwany blok zapytania, który stoi przed właściwym SELECT-em i można się do niego odwoływać po nazwie tak, jakby był zwykłą tabelą — a jego rekurencyjny wariant potrafi rozwinąć hierarchię albo wygenerować ciąg wartości, czego zwykły SELECT zrobić nie potrafi.

Skróty użyte na tej stronie (3)

Skrót	Co znaczy
CTE	Common Table Expression: nazwany kawałek zapytania SQL pisany w WITH
ISO	International Organization for Standardization: organizacja, która wydaje normy techniczne używane na całym świecie
SQL	Structured Query Language: język zapytań do relacyjnych baz danych

Fiszka

TYLKO ODCZYT

Nazwany blok tymczasowy: WITH

```
WITH suma_klienta AS (
    SELECT klient_id, SUM(wartosc) AS suma
    FROM zamowienia
    GROUP BY klient_id
)
SELECT k.nazwa, sk.suma
FROM klienci k
JOIN suma_klienta sk ON sk.klient_id = k.id;
```

> **Pojęcie: CTE (Common Table Expression).** Nazwany blok zapytania zdefiniowany klauzulą `WITH`, do którego reszta zapytania odwołuje się po nazwie jak do zwykłej tabeli. `suma_klienta` istnieje tylko na czas tego jednego zapytania — nie zostaje w bazie jako obiekt, który dałoby się użyć gdzie indziej.

Przykład:

```
            nazwa             |  suma
------------------------------+---------
 Nowak i Wspólnicy Sp. z o.o. | 1680.50
 Kowalski Handel              |  150.00
 Zielona Dolina S.A.          | 2890.00
 Baltic Trade Sp. z o.o.      | 3200.00
 Podlaski Handel              |    0.00
(5 rows)
```

Wynik jest identyczny z tabelą pochodną w `FROM` ze strony Podzapytania: skalarne, wierszowe, skorelowane, EXISTS — różnica jest wyłącznie w miejscu zapisu i w czytelności. CTE stoi na górze zapytania, nazwany i wydzielony, zamiast być zagnieżdżony w środku klauzuli `FROM`.

□□ **CTE** = Common Table Expression: nazwany kawałek zapytania SQL pisany w WITH · **join** = połącz dwa pliki tekstowe po wspólnej kolumnie, jak złączenie tabel w bazie danych

□□ **ZMIENIA SYSTEM**

Wiele CTE w jednym zapytaniu

```
WITH suma_klienta AS (
    SELECT klient_id, SUM(wartosc) AS suma
    FROM zamowienia
    GROUP BY klient_id
),
duzi_klienci AS (
    SELECT * FROM suma_klienta WHERE suma > 1000
)
SELECT k.nazwa, dk.suma
FROM klienci k
JOIN duzi_klienci dk ON dk.klient_id = k.id
```

```
ORDER BY dk.suma DESC;
```

Drugie CTE, `duzi_klienci`, odwołuje się do pierwszego, `suma_klienta`, tak jakby to była zwykła tabela — CTE mogą się między sobą łańcuchowo odwoływać, każde kolejne widzi wszystkie zdefiniowane przed nim.

Przykład:

```
            nazwa             |  suma
------------------------------+---------
 Baltic Trade Sp. z o.o.      | 3200.00
 Zielona Dolina S.A.          | 2890.00
 Nowak i Wspólnicy Sp. z o.o. | 1680.50
(3 rows)
```

Rozbicie długiego zapytania na kilka nazwanych kroków czyta się jak przepis: „najpierw policz sumę na klienta, potem zostaw tylko dużych klientów, potem dołącz ich nazwy" — każdy krok osobno jest prosty do zweryfikowania, w przeciwieństwie do jednego zagnieżdżonego podzapytania w podzapytaniu.

□□ **CTE** = Common Table Expression: nazwany kawałek zapytania SQL pisany w WITH ·**join** = połącz dwa pliki tekstowe po wspólnej kolumnie, jak złączenie tabel w bazie danych

□□ **TYLKO ODCZYT**

Rozwinięcie hierarchii: WITH RECURSIVE

```
WITH RECURSIVE hierarchia AS (
    SELECT id, imie, nazwisko, 1 AS poziom
    FROM pracownicy
    WHERE przelozony_id IS NULL

    UNION ALL

    SELECT p.id, p.imie, p.nazwisko, h.poziom + 1
    FROM pracownicy p
    JOIN hierarchia h ON p.przelozony_id = h.id
)
SELECT * FROM hierarchia ORDER BY poziom, id;
```

Pojęcie: część kotwicząca i część rekurencyjna. Zapytanie rekurencyjne składa się zawsze z dwóch części sklejonych `UNION ALL`. Część kotwicząca (tu pierwszy `SELECT`, szukający pracownika bez przełożonego) uruchamia się raz i daje punkt startowy. Część rekurencyjna (drugi `SELECT`, ze złączeniem do samego CTE) uruchamia się wielokrotnie, za każdym razem dokładając kolejny poziom hierarchii, dopóki cokolwiek zwraca.

Przykład:

```
 id |   imie    | nazwisko  | poziom
----+-----------+-----------+--------
  1 | Ewa       | Kowalczyk |      1
  2 | Marek     | Zawadzki  |      2
  3 | Tomasz    | Wróbel    |      2
  4 | Anna      | Sikora    |      3
  5 | Paweł     | Górski    |      3
  6 | Katarzyna | Baran     |      3
(6 rows)
```

Pojęcie: warunek stopu. To, co kończy rekurencję — moment, w którym część rekurencyjna przestaje zwracać nowe wiersze. Tutaj dzieje się to naturalnie: gdy w kolejnym przebiegu żaden pracownik nie ma `przelozony_id` wskazującego na kogoś dodanego w poprzednim kroku, `JOIN` nie znajduje nic nowego, i rekurencja się kończy sama. Przy hierarchii z cyklem (co w poprawnie zaprojektowanej strukturze przełożonych nie powinno się zdarzyć, ale zdarza się w danych z błędami) brak warunku stopu prowadzi do nieskończonej pętli — dobrym nawykiem jest dopisanie górnego limitu głębokości, na przykład `WHERE h.poziom < 20` w części rekurencyjnej, jako zabezpieczenie.

□□ **CTE** = Common Table Expression: nazwany kawałek zapytania SQL pisany w WITH ·**join** = połącz dwa pliki tekstowe po wspólnej kolumnie, jak złączenie tabel w bazie danych

□□ **TYLKO ODCZYT**

Wygenerowanie ciągu dat

```
WITH RECURSIVE kalendarz AS (
    SELECT DATE '2024-01-01' AS dzien
```

```
    UNION ALL

    SELECT dzien + 1
    FROM kalendarz
    WHERE dzien < DATE '2024-01-07'
)
SELECT * FROM kalendarz;
```

Ten sam mechanizm, zupełnie inne zastosowanie — część kotwicząca daje pierwszy dzień, część rekurencyjna dokłada po jednym dniu na przebieg, a warunek stopu (`dzien < DATE '2024-01-07'`) jest tym razem jawny i konieczny, bo licznik dat nie zatrzyma się sam z siebie tak jak skończona hierarchia pracowników.

Przykład:

```
   dzien
------------
 2024-01-01
 2024-01-02
 2024-01-03
 2024-01-04
 2024-01-05
 2024-01-06
 2024-01-07
(7 rows)
```

Ten wzorzec zastępuje osobną, ręcznie utrzymywaną tabelę kalendarza tam, gdzie zapytanie potrzebuje ciągłego zakresu dat — na przykład do raportu, który ma pokazać zero sprzedaży w dniach bez żadnego zamówienia, a nie po prostu pominąć te dni milczeniem.

Po co to istnieje

CTE nie dodaje SQL-owi żadnej nowej mocy obliczeniowej ponad to, co dało się zapisać podzapytaniami zagnieżdżonymi w `FROM` — strona Podzapytania: skalarne, wierszowe, skorelowane, EXISTS pokazała dokładnie to samo zapytanie o sumie na klienta zapisane bez `WITH`. To, co CTE realnie daje, to czytelność przy zapytaniach wieloetapowych: zamiast podzapytania zagnieżdżonego w podzapytaniu, zagnieżdżonego w kolejnym, każdy krok dostaje własną nazwę i stoi osobno, w kolejności logicznej „najpierw to, potem tamto" — dokładnie w takiej kolejności, w jakiej człowiek naturalnie opisałby ten sam proces słowami.

`WITH RECURSIVE` jest jedynym miejscem w standardowym SQL, w którym da się wyrazić operację powtarzaną nieznaną z góry liczbę razy — rozwinięcie hierarchii o nieznanej głębokości, listę wszystkich elementów podrzędnych w strukturze drzewiastej, ciąg wartości o długości zależnej od danych. Bez rekurencji te same zadania wymagałyby albo sztywnej liczby złączeń `pracownicy` samej ze sobą (jedno na każdy poziom hierarchii, z góry ograniczonej liczby), albo przetwarzania po stronie aplikacji.

Jak to działa

Rozwiń: jak to działa pod spodem

Materializacja CTE w PostgreSQL. Do wersji 11 włącznie PostgreSQL zawsze materializował każde CTE — liczył je raz, zapisywał wynik jako tymczasową tabelę w pamięci i dopiero potem wykonywał resztę zapytania na tym materializowanym wyniku. To dawało przewidywalne zachowanie, ale czasem szkodziło wydajności: optymalizator nie mógł „zajrzeć" do środka CTE i na przykład przesunąć do niego filtr z zewnętrznego zapytania.

Od PostgreSQL 12 domyślne zachowanie się zmieniło — silnik traktuje proste, jednorazowo używane CTE jak zwykłe podzapytania w `FROM` i może je zoptymalizować razem z resztą zapytania (inline), o ile nic w zapytaniu na to nie wymusza inaczej. Jawna kontrola wróciła jako dwa słowa kluczowe:

```
WITH suma_klienta AS MATERIALIZED (
    SELECT klient_id, SUM(wartosc) AS suma FROM zamowienia GROUP BY klient_id
)
SELECT * FROM suma_klienta WHERE klient_id = 1;
```

`MATERIALIZED` wymusza stare zachowanie sprzed wersji 12 — CTE liczone osobno, jako odrębny krok. `NOT MATERIALIZED` wymusza nowe zachowanie nawet tam, gdzie optymalizator sam mógłby wybrać inaczej. Domyślne zachowanie bez żadnego z tych słów zależy od tego, czy CTE jest użyte więcej niż raz (wtedy zwykle materializowane, żeby nie liczyć go dwukrotnie) i czy zawiera operacje z efektami ubocznymi, takie jak `INSERT`, `UPDATE` czy `DELETE` w `WITH` (wtedy zawsze materializowane, bo efekt uboczny musi wykonać się dokładnie raz).

Silnik	Domyślne zachowanie CTE
PostgreSQL ≤ 11	zawsze materializowane
PostgreSQL ≥ 12	optymalizator decyduje, chyba że jawnie wymuszono `MATERIALIZED` / `NOT MATERIALIZED`

Silnik	Domyślne zachowanie CTE
SQL Server	traktowane jak podzapytanie inline, bez ręcznej kontroli materializacji
Oracle	optymalizator decyduje (`MATERIALIZE` / `INLINE` jako podpowiedzi `/*+ ... */`)

Wykonanie WITH RECURSIVE. Silnik wykonuje część kotwiczącą dokładnie raz, zapamiętuje jej wynik, a potem w pętli wykonuje część rekurencyjną, za każdym razem podstawiając pod nazwę CTE wynik z poprzedniego przebiegu, nie z wszystkich przebiegów naraz. Nowe wiersze z każdego przebiegu dokładają się do wyniku końcowego przez `UNION ALL`. Pętla kończy się, gdy przebieg rekurencyjny nie zwróci żadnego nowego wiersza.

Częste błędy

UNION zamiast UNION ALL w części rekurencyjnej. Składniowo dozwolone w niektórych silnikach, ale kosztowne — każdy przebieg rekurencji musiałby wtedy porównywać nowe wiersze ze wszystkimi dotychczas wygenerowanymi, żeby usunąć duplikaty, zamiast po prostu je doklejać. Przy hierarchii bez duplikatów (a taka powinna być każda poprawna struktura przełożonych) `UNION ALL` daje identyczny wynik szybciej.

Brak warunku stopu przy generowaniu ciągu. Pominięcie `WHERE dzien < DATE '2024-01-07'` w części rekurencyjnej generatora dat sprawia, że rekurencja nie ma żadnego naturalnego końca — silnik dolicza kolejne dni bez ograniczenia, aż napotka wbudowany limit głębokości rekurencji (w PostgreSQL domyślnie sto poziomów) i przerwie zapytanie błędem, zamiast zwrócić rozsądny wynik.

```
ERROR:  recursion "kalendarz" does not terminate
HINT:  Use a recursive CTE that gets smaller each iteration.
```

Cykl w hierarchii bez zabezpieczenia. Jeśli przez błąd w danych pracownik A ma przełożonym B, a B ma przełożonym A, `WITH RECURSIVE` bez dodatkowego zabezpieczenia (na przykład limitu poziomu albo tablicy odwiedzonych identyfikatorów) wpadnie w nieskończoną pętlę na tych samych dwóch wierszach, tak samo jak przy braku warunku stopu w generatorze dat.

□□ Ćwiczenia

> Zadania sprawdzające.

Ćwiczenie 1 — poziomy hierarchii

W tabeli `pracownicy` Ewa Kowalczyk nie ma przełożonego, Marek Zawadzki i Tomasz Wróbel mają przełożonym Ewę, a Anna Sikora, Paweł Górski i Katarzyna Baran mają przełożonych odpowiednio Marka i Tomasza. Ile poziomów wygeneruje `WITH RECURSIVE` z Fiszki tej strony, i ile pracowników znajdzie się na najgłębszym poziomie?

Odpowiedź

Trzy poziomy. Poziom 1 to sama Ewa (jeden pracownik), poziom 2 to Marek i Tomasz (dwóch), poziom 3 to Anna, Paweł i Katarzyna (trzech) — to jest najgłębszy poziom w tej hierarchii, bo żaden z tej trójki nie jest przełożonym kogokolwiek innego.

Pytania kontrolne

1. Czym różni się CTE od tabeli pochodnej w FROM, skoro oba są podzapytaniami dającymi ten sam wynik?

Odpowiedź

Różnica jest głównie w czytelności i możliwości odwołania się do tego samego bloku wielokrotnie po nazwie — CTE stoi na górze zapytania, nazwane, i kolejne CTE mogą się do niego odwoływać, budując łańcuch nazwanych kroków. Tabela pochodna jest zagnieżdżona bezpośrednio w miejscu użycia i nie ma własnej nazwy poza aliasem nadanym przy tym jednym użyciu. Logicznie, bez rekurencji, oba dają identyczne możliwości.

2. Dlaczego część rekurencyjna WITH RECURSIVE musi być połączona z częścią kotwiczącą przez UNION ALL, a nie przez zwykły JOIN?

Odpowiedź

Bo część kotwicząca i część rekurencyjna to dwa niezależne zapytania dające wiersze tego samego kształtu na różnych etapach — kotwicząca daje punkt startowy, rekurencyjna dokłada kolejne poziomy. `UNION ALL` sklejenia je w jeden strumień wynikowy, dokładnie tak jak każde inne sklejenie dwóch zapytań o tym samym kształcie, opisane na stronie UNION vs UNION ALL. `JOIN` zestawiałby je poziomo, obok siebie, co nie miałoby tu żadnego sensu.

Powiązane: Podzapytania: skalarne, wierszowe, skorelowane, EXISTS · UNION vs UNION ALL · CROSS JOIN i SELF JOIN · Funkcje okna — OVER, PARTITION BY, ROW_NUMBER, RANK, LAG/LEAD

Źródła: ISO/IEC 9075-2:2016 § 7.13, § 7.17 (with clause) · dokumentacja PostgreSQL, rozdział „WITH Queries (Common Table Expressions)"

Revision #47
Created 2026-09-24 22:42:08 UTC by Tomasz Konieczny
Updated 2026-10-26 14:11:10 UTC by Tomasz Konieczny