



СТЕ: общие табличные выражения

СТЕ - Common Table Expressions

Создание таблиц и представлений

Рекурсия

Рекурсивные запросы “дерево”

СТЕ - Common Table Expressions

```
WITH имя_отношения AS ( запрос )
```

- Определение временных отношений, существующих только для одного запроса
- Предназначение - разбиение сложных запросов на простые части
- Внутри with можно использовать SELECT, INSERT, UPDATE или DELETE
- with можно присоединять к операторам SELECT, INSERT, UPDATE или DELETE

WITH предоставляет способ записывать дополнительные отношения для применения в больших запросах.

Эти отношения можно представить как определения временных таблиц, существующих только для одного запроса.

👉 В postgresql отношение в with вычисляется один раз и сохраняется, независимо от того сколько обращений будет в основном запросе.

👉 В MySQL отношение в with может вычисляться один раз и сохраняться, может вычисляться при обращении из исходного запроса в зависимости от решения оптимизатора.

👉 В других системах возможно разное поведение - читайте документацию.

👉 SELECT в предложении WITH нужен, чтобы разбивать сложные запросы на простые части.

Пример

Запрос без применения WITH:

```
SELECT
    user_id, GREATEST(dtime - interval '30 days')::date, dtime, s
FROM (
    SELECT
        user_id, dtime, s,
        ROW_NUMBER() OVER(PARTITION BY user_id ORDER BY s DESC) rn
    FROM (
        SELECT
            user_id, dtime::date AS dtime,
            SUM(amount) OVER(PARTITION BY user_id ORDER BY dtime RANGE '30 days' PRECEDING) AS s
        FROM transactions
    ) t
    ) t WHERE rn = 1
```

Запрос с применением WITH:

```
WITH
    t_sum AS (
    SELECT
        user_id, dtime::date AS dtime,
        SUM(amount) OVER(PARTITION BY user_id ORDER BY dtime RANGE '30 days' preceding)
        AS s
    FROM transactions
    ),
    t_rn AS (
    SELECT
        user_id, dtime, s,
        ROW_NUMBER() OVER(PARTITION BY user_id ORDER BY s DESC) rn
    FROM t_sum
    )
    SELECT
        user_id, GREATEST(dtime - interval '30 days')::date, dtime, s
    FROM t_rn
    WHERE rn = 1
```

Структура запроса стала более логически последовательной: он читается слева направо, сверху вниз, и не требует разбора вложенных структур.

Создание таблиц и представлений

Для того чтобы использовать одну и ту же логику в разных запросах можно создать таблицу или представление.

```
CREATE TABLE имя_отношения AS ( запрос );
```

Создание таблицы выполняет sql код и сохраняет результат в новой таблице

```
CREATE VIEW имя_отношения AS ( запрос );
```

Создание представления сохраняет sql код. В запросах можно использовать имя представления как имя таблицы, но сохраненный запрос каждый раз будет выполняться заново

Рекурсия

```
WITH RECURSIVE имя_таблицы AS (  
    нерекурсивная часть  
    UNION [ALL]  
    рекурсивная часть  
)
```

-
- Позволяет запросу обращаться к собственному результату
 - Только в рекурсивной части можно обратиться к результату запроса

RECURSIVE превращает WITH из просто удобной синтаксической конструкции в средство реализации того, что невозможно в стандартном SQL.

Пример

Таким может быть рекурсивный запрос, который посчитает сумму набегающего итога:

В результате будет сумма арифметической прогрессии

```
WITH RECURSIVE t(n) AS (
  VALUES (1)
  UNION ALL
  SELECT n+1 FROM t WHERE n < 100
)
SELECT SUM(n) FROM t;
```

Sum
5050

Как работает рекурсия

```
WITH RECURSIVE имя_таблицы AS (
  нерекурсивная часть
  UNION [ALL]
  рекурсивная часть
)
```

1. Вычисляется результат нерекурсивной части
2. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки.
3. Оставшиеся строки записываются в две таблицы - в результат и во временную *рабочую*
4. Пока *рабочая* таблица не пуста, повторяются следующие действия:
 - a. в рекурсивной части ссылка на себя подменяется на текущее содержимое *рабочей* таблицы
 - b. Вычисляется рекурсивная часть.
 - c. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные
 - d. Оставшиеся строки добавляются к результату и еще записываются во временную *промежуточную* таблицу
 - e. Содержимое *рабочей* таблицы заменяется содержимым *промежуточной* таблицы, а затем *промежуточная* таблица очищается.

Таким образом запрос вычисляется несколько раз, добавляя данные в результат, пока очередной вызов возвращает новые строки.

! Этот процесс является **итерационным**, а не рекурсивным, но комитетом по стандартам SQL был выбран термин RECURSIVE.

В показанном выше примере в рабочей таблице на каждом этапе содержится всего одна строка и в ней последовательно накапливаются значения от 1 до 100.

На сотом шаге, благодаря условию WHERE, не возвращается ничего, так что вычисление запроса завершается.

Пример

Рекурсивный расчет факториала:

```
WITH RECURSIVE t(n, fact) AS (  
    VALUES (1, 1)  
    UNION ALL  
    SELECT n+1, fact * (n+1) FROM t WHERE n < 6  
)  
SELECT max(fact) FROM t;
```

В первый раз берется стартовая часть рекурсии, а в следующие итерации - результат предыдущей. Как только очередной вызов не вернет строк из-за условия $n < 6$ расчет остановится

💬 Это учебный пример - PostgreSQL, например, и так умеет вычислять факториал.

Расчет факториала

SELECT 10000!

Рекурсивные запросы “дерево”

В случае древовидных структур данных для обхода используются рекурсивные запросы.

Пример

Дана таблица geo (id, parent_id, name) с топонимами. Здесь id идентификатор некоторого географического объекта, name его название, а parent_id - идентификатор другого топонима с этой же таблицы, которому принадлежит данный.


Нужно выбрать всё, что относится к Европе, и добавить уровень вложенности узла дерева к результату.

```
WITH RECURSIVE eu(id, name, level) AS (  
    SELECT id, name, 1 FROM geo WHERE id = 2  
    UNION ALL  
    SELECT geo.id, geo.name, eu.level + 1  
    FROM eu  
    JOIN geo ON eu.id = geo.parent_id  
)  
SELECT*from eu
```

В запрос нужно добавить еще один столбец (level), который будет означать уровень, и начальное значение (1). В рекурсивном запросе используется предыдущее значение, к которому тоже добавляется 1.

Каждый уровень вложенности увеличивается на 1.

По сути строки и уровнем вложенности 2 - это содержимое временной таблицы после выполнения второго шага. С уровнем 3 - третьего. И так далее.

 А еще можно посмотреть [вики PostgreSQL](#), где есть бесполезные, но веселые примеры рекурсии.