# ТЕМА 11. ОПЕРАТОР WITH

```
[ WITH [ RECURSIVE ]
имя_запроса_WITH [ ( имя_столбца [, ...] ) ] AS ( выборка | values | insert | update | delete )
SELECT ...
```

Предложение WITH позволяет задать один или несколько подзапросов, к которым затем можно обратиться по имени в основном запросе. Эти подзапросы по сути действуют как временные таблицы или представления в процессе выполнения главного запроса. Каждый подзапрос может представлять собой оператор SELECT, TABLE, VALUES, INSERT, UPDATE или DELETE.

При использовании в WITH оператора, изменяющего данные, (INSERT, UPDATE или DELETE) обычно добавляется предложение RETURNING. Если RETURNING опущено, оператор, тем не менее, выполняется, но не выдаёт никакого результата, так что на него нельзя сослаться как на таблицу в основном запросе.

Имя должно быть указано для каждого запроса WITH. Также можно задать необязательный список с именами столбцов; если он опущен, имена столбцов формируются из результата подзапроса.

Если указано RECURSIVE, подзапрос SELECT может ссылаться сам на себя по имени. Такой подзапрос должен иметь форму

```
нерекурсивная_часть UNION [ ALL | DISTINCT ] рекурсивная_часть
```

, где рекурсивная ссылка на сам запрос может находиться только справа от UNION. Для одного запроса допускается только одна рекурсивная ссылка на него же. Операторы, изменяющие данные, не могут быть рекурсивными, но результат рекурсивного запроса SELECT в таких операторах можно использовать.

Пример 1. Расчет факториала

```
WITH RECURSIVE r AS (
   -- стартовая часть рекурсии (т.н. "anchor")
   SELECT
      1 AS i,
      1 AS factorial

   UNION

   -- рекурсивная часть
   SELECT
      i+1 AS i,
      factorial * (i+1) as factorial
   FROM r
   WHERE i < 10
)
SELECT * FROM r;
```

**Вычисление рекурсивного запроса**

1. Вычисляется не рекурсивная часть. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *рабочую таблицу*.
2. Пока рабочая таблица не пуста, повторяются следующие действия:
    a. Вычисляется рекурсивная часть так, что рекурсивная ссылка на сам запрос обращается к текущему содержимому рабочей таблицы. Для UNION (но не UNION ALL) отбрасываются дублирующиеся строки и строки, дублирующие ранее полученные. Все оставшиеся строки включаются в результат рекурсивного запроса и также помещаются во временную *промежуточную таблицу*.

b. Содержимое рабочей таблицы заменяется содержимым промежуточной таблицы, а затем промежуточная таблица очищается.

Ещё одна особенность RECURSIVE в том, что запросы WITH могут быть неупорядоченными: запрос может ссылаться на другой, идущий в списке после него. (Однако циклические ссылки или взаимная рекурсия не поддерживаются.) Без RECURSIVE запрос в WITH может ссылаться только на запросы того же уровня в WITH, предшествующие ему в списке WITH.

Ключевое свойство запросов WITH состоит в том, что они вычисляются один раз для всего основного запроса, даже если в основном запросе содержатся несколько ссылок на них. В частности, гарантируется, что операторы, изменяющие данные, будут выполняться ровно один раз, вне зависимости от того, будет ли их результат прочитан основным запросом и в каком объёме.

Основной запрос и все запросы WITH, условно говоря, выполняются одновременно. Это значит, что действие оператора, изменяющего данные в WITH, не будут видеть другие части запроса, кроме как прочитав его вывод RETURNING. Если два таких оператора попытаются изменить одну строку, результат будет неопределённым.

Привет 2. Выборка из иерархической структуры

```
CREATE TABLE geo (
    id int not null primary key,
    parent_id int references geo(id),
    name varchar(1000)
);

INSERT INTO geo
(id, parent_id, name)
VALUES
(1, null, 'Планета Земля'),
(2, 1, 'Континент Евразия'),
(3, 1, 'Континент Северная Америка'),
(4, 2, 'Европа'),
(5, 4, 'Россия'),
(6, 4, 'Германия'),
(7, 5, 'Москва'),
(8, 5, 'Санкт-Петербург'),
(9, 6, 'Берлин');

WITH RECURSIVE r AS (
   SELECT id, parent_id, name
   FROM geo
   WHERE parent_id = 4

   UNION

   SELECT geo.id, geo.parent_id, geo.name
   FROM geo
      JOIN r
         ON geo.parent_id = r.id
)

SELECT * FROM r;
```

# ОКОННЫЕ ФУНКЦИИ

Оконные функции не изменяют выборку, а только добавляют некоторую дополнительную информацию о ней. Т.е. для простоты понимания можно считать, что postgres сначала выполняет весь запрос (кроме сортировки и limit), а потом только просчитывает оконные выражения.

Синтаксис примерно такой:
***функция OVER окно***

Окно — это некоторое выражение, описывающее набор строк, которые будет обрабатывать функция и порядок этой обработки. Причем окно может быть просто задано пустыми скобками (), т.е. окном являются все строки результата запроса.

Например, в этом селекте к обычным полям id, header и score просто добавится нумерация строк.

```
SELECT
    id,
    section,
    header,
    score,
    row_number() OVER ()  AS num
FROM news;
```

```
 id | section |  header   | score | num
----+---------+-----------+-------+-----
  1 |       2 | Заголовок |    23 |   1
  2 |       1 | Заголовок |     6 |   2
  3 |       4 | Заголовок |    79 |   3
  4 |       3 | Заголовок |    36 |   4
  5 |       2 | Заголовок |    34 |   5
  6 |       2 | Заголовок |    95 |   6
  7 |       4 | Заголовок |    26 |   7
  8 |       3 | Заголовок |    36 |   8
```

В оконное выражение можно добавить ORDER BY, тогда можно изменить порядок обработки.

```
SELECT
    id,
    section,
    header,
    score,
    row_number() OVER (ORDER BY score DESC)  AS rating
FROM news
ORDER BY id;
```

```
 id | section |  header   | score | rating
----+---------+-----------+-------+--------
  1 |       2 | Заголовок |    23 |      7
  2 |       1 | Заголовок |     6 |      8
  3 |       4 | Заголовок |    79 |      2
  4 |       3 | Заголовок |    36 |      4
  5 |       2 | Заголовок |    34 |      5
  6 |       2 | Заголовок |    95 |      1
  7 |       4 | Заголовок |    26 |      6
  8 |       3 | Заголовок |    36 |      3
```

В оконное выражение можно добавить слово PARTITION BY [expression],
например row_number() OVER (PARTITION BY section), тогда подсчет будет идти в каждой группе отдельно:

```
SELECT
    id,
    section,
    header,
    score,
    row_number() OVER (PARTITION BY section ORDER BY score DESC)  AS rating_in_section
FROM news
ORDER BY section, rating_in_section;
```

```
 id | section |  header   | score | rating_in_section
----+---------+-----------+-------+-------------------
  2 |       1 | Заголовок |     6 |                 1
  6 |       2 | Заголовок |    95 |                 1
  5 |       2 | Заголовок |    34 |                 2
  1 |       2 | Заголовок |    23 |                 3
  4 |       3 | Заголовок |    36 |                 1
  8 |       3 | Заголовок |    36 |                 2
  3 |       4 | Заголовок |    79 |                 1
  7 |       4 | Заголовок |    26 |                 2
```

Если не указывать партицию, то партицией является весь запрос.

Тут сразу надо немного сказать о функциях, которые можно использовать, так как есть очень важный нюанс.

В качестве функции можно использовать, так сказать, истинные оконные функции из мануала — это row_number(), rank(), lead() и т.д., а можно использовать функции-агрегаты, такие как: sum(), count() и т.д. Так вот, это важно, агрегатные функции работают слегка по-другому: если не задан ORDER BY в окне, идет подсчет по всей партиции один раз, и результат пишется во все строки (одинаков для всех строк партиции). Если же ORDER BY задан, то подсчет в каждой строке идет от начала партиции до этой строки.

Посмотрим это на примере. Например, у нас есть некая (сферическая в вакууме) таблица пополнений балансов.

```
SELECT
    transaction_id,
    change
FROM balance_change
ORDER BY transaction_id;
```

```
 transaction_id | change
----------------+--------
              1 |   1.00
              2 |  -2.00
              3 |  10.00
              4 |  -4.00
              5 |   5.50
```

и мы хотим узнать заодно, как менялся остаток на балансе при этом:

```
SELECT
    transaction_id,
    change,
    sum(change) OVER (ORDER BY transaction_id) as balance
FROM balance_change
ORDER BY transaction_id;
```

```
transaction_id | change | balance
---------------+--------+---------
              1 |   1.00 |    1.00
              2 |  -2.00 |   -1.00
              3 |  10.00 |    9.00
```

```
              4 |  -4.00 |    5.00
              5 |   5.50 |   10.50
```

Т.е. для каждой строки идет подсчет в отдельном фрейме. В данном случае фрейм — это набор строк от начала до текущей строки (если было бы PARTITION BY, то от начала партиции).

Если же мы для агрегатной фунции sum не будем использовать ORDER BY в окне, тогда мы просто посчитаем общую сумму и покажем её во всех строках. Т.е. фреймом для каждой из строк будет весь набор строк
от начала до конца партиции.

```
SELECT
    transaction_id,
    change,
    sum(change) OVER () as result_balance
FROM balance_change
ORDER BY transaction_id;
```

```
 transaction_id | change | result_balance
----------------+--------+----------------
              1 |   1.00 |          10.50
              2 |  -2.00 |          10.50
              3 |  10.00 |          10.50
              4 |  -4.00 |          10.50
              5 |   5.50 |          10.50
```

Результат работы оконной функции невозможно отфильтровать в запросе с помощью WHERE, потому что оконные фунции выполняются после всей фильтрации и группировки, т.е. с тем, что получилось. Поэтому чтобы выбрать, например, топ 5 новостей в каждой группе, надо использовать подзапрос:

```
SELECT *
FROM (
    SELECT
        id,
        section,
        header,
        score,
        row_number() OVER (PARTITION BY section ORDER BY score DESC)  AS
rating_in_section
    FROM news
    ORDER BY section, rating_in_section
) counted_news
WHERE rating_in_section <= 5;
```

# АГРЕГАТНЫЕ ВЫРАЖЕНИЯ

*Агрегатное выражение* представляет собой применение агрегатной функции к строкам, выбранным запросом. Агрегатная функция сводит множество входных значений к одному выходному, как например, сумма или среднее. Агрегатное выражение может записываться следующим образом:

```
агрегатная_функция (выражение [ , ... ] [ предложение_order_by ] ) [ FILTER ( WHERE
условие_фильтра ) ]

агрегатная_функция (ALL выражение [ , ... ] [ предложение_order_by ] ) [ FILTER (
WHERE условие_фильтра ) ]

агрегатная_функция (DISTINCT выражение [ , ... ] [ предложение_order_by ] ) [ FILTER (
WHERE условие_фильтра ) ]

агрегатная_функция ( * ) [ FILTER ( WHERE условие_фильтра ) ]

агрегатная_функция ( [ выражение [ , ... ] ] ) WITHIN GROUP ( предложение_order_by ) [
FILTER ( WHERE условие_фильтра ) ]
```

Здесь *агрегатная_функция* — имя ранее определённой агрегатной функции (возможно, дополненное именем схемы), *выражение* — любое выражение значения, не содержащее в себе агрегатного выражения или вызова оконной функции.

В первой форме агрегатного выражения агрегатная функция вызывается для каждой строки. Вторая форма эквивалентна первой, так как указание ALL подразумевается по умолчанию. В третьей форме агрегатная функция вызывается для всех различных значений выражения (или набора различных значений, для нескольких выражений), выделенных во входных данных. В четвёртой форме агрегатная функция вызывается для каждой строки, так как никакого конкретного значения не указано (обычно это имеет смысл только для функции count(*)). В последней форме используются *сортирующие* агрегатные функции, которые будут описаны ниже.

Большинство агрегатных функций игнорируют значения NULL, так что строки, для которых выражения выдают одно или несколько значений NULL, отбрасываются. Это можно считать истинным для всех встроенных операторов, если явно не говорится об обратном.

Например, count(*) подсчитает общее количество строк, а count(f1) только количество строк, в которых f1 не NULL (так как count игнорирует NULL), а count(distinct f1) подсчитает число различных и отличных от NULL значений столбца f1.

Обычно строки данных передаются агрегатной функции в неопределённом порядке и во многих случаях это не имеет значения, например функция min выдаёт один и тот же результат независимо от порядка поступающих данных. Однако некоторые агрегатные функции (такие как array_agg и string_agg) выдают результаты, зависящие от порядка данных. Для таких агрегатных функций можно добавить *предложение_order_by* и задать нужный порядок. Это *предложение_order_by* имеет тот же синтаксис, что и предложение ORDER BY на уровне запроса за исключением того, что его выражения должны быть просто выражениями, а не именами результирующих столбцов или числами. Например:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Заметьте, что при использовании агрегатных функций с несколькими аргументами, предложение `ORDER BY` идёт после всех аргументов. Например, надо писать так:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

а не так:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- неправильно
```

Агрегатное выражение может фигурировать только в списке результатов или в предложении `HAVING` команды `SELECT`. Во всех остальных предложениях, например `WHERE`, они запрещены, так как эти предложения логически вычисляются до того, как формируются результаты агрегатных функций.

### *Агрегатные функции общего назначения*

| **Функция** | **Типы аргумента** | **Описание** |
|---|---|---|
| `array_agg(`*выражение*`)` | любой тип не массива | входные значения, включая NULL, объединяются в массив |
| `avg(`*выражение*`)` | `smallint`, `int`, `bigint`, `real`, `double precision`, `numeric` или `interval` | арифметическое среднее для всех входных значений |
| `bool_and(`*выражение*`)` | `bool` | true, если все входные значения равны true, и false в противном случае |
| `bool_or(`*выражение*`)` | `bool` | true, если хотя бы одно входное значение равно true, и false в противном случае |
| `count(`*выражение*`)` | any | количество входных строк, для которых значение *выражения* не равно NULL |
| `max(`*выражение*`)` | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | максимальное значение *выражения* среди всех входных данных |
| `min(`*выражение*`)` | любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов | минимальное значение *выражения* среди всех входных данных |
| `string_agg(`*выражение*`, `*разделитель*`)` | (`text`, `text`) или (`bytea`, `bytea`) | входные данные складываются в строку через заданный разделитель |
| `sum(`*выражение*`)` | `smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, `interval` или `money` | сумма значений *выражения* по всем входным данным |