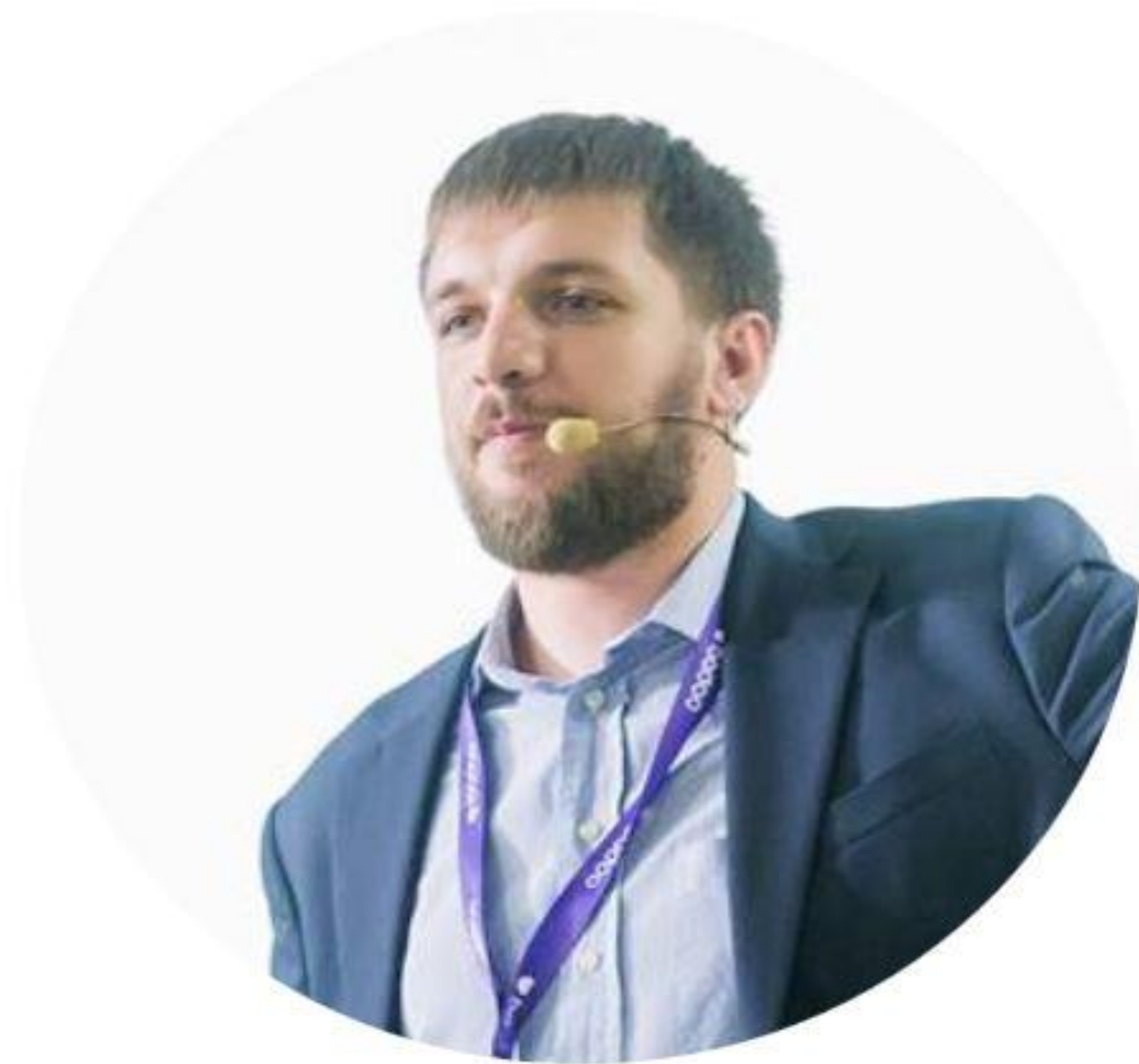


## ЗАНЯТИЕ 1.5

# Работа с PostgreSQL



# Алексей Кузьмин

Директор разработки; Data Scientist  
**ДомКлик.ру**



[aleksej.kyzmin@gmail.com](mailto:aleksej.kyzmin@gmail.com)

---

ЧТО СЕГОДНЯ ИЗУЧИМ



- Оконные и аналитические функции
- СТЕ
- Представления
- Ускорение запросов
- Схемы запросов

—


# Оконные (аналитические) функции



**Оконные функции** - полезный инструмент для построения сложных аналитических запросов.

Для их использования нужно задать параметры окна и функцию, которую хотим посчитать на каждом объекте внутри окна.

Простой пример - функция ROW\_NUMBER(). Эта функция нумерует строки внутри окна. Пронумеруем контент для каждого пользователя в порядке убывания рейтингов.



```
SELECT
  userId, movieId, rating,
  ROW_NUMBER() OVER (PARTITION BY userId ORDER BY rating DESC) as movie_rank
FROM (
  SELECT DISTINCT
    userId, movieId, rating
  FROM ratings
  WHERE userId <> 1 LIMIT 1000
) as sample
ORDER BY
  userId,
  rating DESC,
  movie_rank
LIMIT 20;
```

Результат:

userid | movieid | rating | movie\_rank

2	1356	5	1
2	339	5	2
2	648	4	3
2	605	4	4
2	1233	4	5
2	1210	4	6
2	377	4	7
2	260	4	8
2	79	4	9
2	628	4	10
2	64	4	11
2	58	3	12
2	25	3	13
2	762	3	14



## Параметры запроса:

- ROW\_NUMBER - функция, которую применяем к окну
- OVER - описание окна

Описание окна содержит:

- PARTITION BY - поле (или список полей), которые описывают группу строк для применения оконной функции
- ORDER BY - поле, которое задаёт порядок записей внутри окна. Для полей внутри ORDER BY можно применять стандартные модификаторы DESC, ASC

Оконная функция никак не меняет количество строк в выдаче, но к каждой строке добавляется полезная информация - например, про порядковый номер строки внутри окна. Названия функций обычно отражают их смысл. Ниже будут приведены примеры использования и результаты запросов.

## SUM()

Суммирует значения внутри окна. Посчитаем странную метрику - разделим каждое значение рейтинга на сумму всех рейтингов этого пользователя.

```
SELECT userId, movieId, rating,  
       rating / SUM(rating) OVER (PARTITION BY userId) as strange_rating_metric  
FROM (SELECT DISTINCT userId, movieId, rating FROM ratings WHERE userId <> 1  
LIMIT 1000) as sample  
ORDER BY userId, rating DESC  
LIMIT 20;
```

Результат:

userid | movieid | rating |  
strange\_rating\_metric

-----+-----+-----+-----

2 | 339 | 5 | 0.0684931506849315  
2 | 1356 | 5 | 0.0684931506849315  
2 | 648 | 4 | 0.0547945205479452  
2 | 64 | 4 | 0.0547945205479452  
2 | 79 | 4 | 0.0547945205479452  
2 | 260 | 4 | 0.0547945205479452  
2 | 1233 | 4 | 0.0547945205479452  
2 | 1210 | 4 | 0.0547945205479452  
2 | 377 | 4 | 0.0547945205479452  
2 | 605 | 4 | 0.0547945205479452  
2 | 628 | 4 | 0.0547945205479452  
2 | 762 | 3 | 0.0410958904109589  
2 | 141 | 3 | 0.0410958904109589

userid | movieid | rating | strange\_rating\_metric

-----+-----+-----+-----

2 | 780 | 3 | 0.0410958904109589  
2 | 5 | 3 | 0.0410958904109589  
2 | 58 | 3 | 0.0410958904109589  
2 | 25 | 3 | 0.0410958904109589  
2 | 1475 | 3 | 0.0410958904109589  
2 | 32 | 2 | 0.0273972602739726  
2 | 1552 | 2 | 0.0273972602739726

(20 rows)

## COUNT(), AVG()

Счётчик элементов внутри окна, а так же функция **Average()**. Для наглядности воспользуемся ими одновременно - результаты не должны отличаться. Вычислим полезную метрику - отклонение рейтинга пользователя от среднего рейтинга, который он склонен выставлять

```
SELECT userId, movieId, rating,  
       rating - AVG(rating) OVER (PARTITION BY userId) rating_deviance_simplex,  
       rating - SUM(rating) OVER (PARTITION BY userId) /COUNT(rating) OVER (PARTITION  
BY userId) as rating_deviance_complex  
FROM (SELECT DISTINCT userId, movieId, rating FROM ratings WHERE userId <>1 LIMIT  
1000) as sample  
ORDER BY userId, rating DESC  
LIMIT 20;
```

Результат:

userid | movieid | rating | rating\_deviance\_simplex | rating\_deviance\_complex

-----+-----+-----+-----+-----				
2	339	5	1.68181818181818	1.68181818181818
2	1356	5	1.68181818181818	1.68181818181818
2	648	4	0.681818181818182	0.681818181818182
2	64	4	0.681818181818182	0.681818181818182
2	79	4	0.681818181818182	0.681818181818182
2	260	4	0.681818181818182	0.681818181818182
2	1233	4	0.681818181818182	0.681818181818182
2	1210	4	0.681818181818182	0.681818181818182
2	377	4	0.681818181818182	0.681818181818182
2	605	4	0.681818181818182	0.681818181818182
2	628	4	0.681818181818182	0.681818181818182

—

ВРЕМЯ ПРАКТИКИ



## Практика 1

Выведите таблицу с 3-мя полями: название фильма, имя актера и количество фильмов, в которых он снимался

—

# PostgreSQL CTE



---

**CTE** - это временный результат запроса, который можно использовать с другими запросами. Временный = существует только в рамках запроса.

Синтаксис:

```
WITH cte_name (column_list) AS (  
    CTE_query_definition  
)  
statement;
```

- указывается название cte
- опционально список имен колонок
- запрос cte
- основной sql запрос

Обычно используются для упрощения сложных join-запросов и подзапросов. Кроме того поддерживают рекурсивные запросы

```
WITH cte_film AS (  
    SELECT  
        film_id,  
        title,  
        (CASE  
            WHEN length < 30 THEN 'Short'  
            WHEN length >= 30 AND length < 90 THEN  
'Medium'  
            WHEN length > 90 THEN 'Long'  
        END) length  
    FROM  
        film  
)
```

```
SELECT  
    film_id,  
    title,  
    length  
FROM  
    cte_film  
WHERE  
    length = 'Long'  
ORDER BY  
    title;
```

## ОТВ и рекурсивные запросы

Вычисление чего-то итерациями до того, как будет выполнено некоторое условие

WITH имя\_ОТВ (список\_\_столбцов) AS

( стартовый\_\_запрос

union [all]

рекурсивный\_\_запрос\_\_к\_\_имя\_ОТВ

) внешний\_запрос

## Задача

Посчитать факториал:

$$n! = 1234 \dots (n-1)n$$

[]

```
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;



Алгоритм примерно такой:

1. Извлечь стартовые данные

2. Подставить полученные данные с предыдущей итерации в «рекурсивную» часть запроса.

3. Если в текущей итерации рекурсивной части не пустая строка, то добавляем ее в результирующую выборку, а также пометить данные, как данные для следующего вызова рекурсивной части (п. 2), иначе завершить обработку

—

ВРЕМЯ ПРАКТИКИ

## Практика 2

При помощи СТЕ выведите таблицу со следующим содержанием  
Фамилия и  
Имя сотрудника (staff) и количество прокатов двд (retal), которые он продал



—

# Представления

---

View - это именованные запросы, которые помогают сделать представление (именно вид) данных, лежащий в таблицах PostgreSQL. View основывается на одной или нескольких базовых таблицах. Удобны для часто используемых запросов.

View (кроме materialized view) не хранят данные.

*Создание*

```
CREATE VIEW view_name AS query;
```

Пример с информацией о покупателях:

```
SELECT cu.customer_id AS id,  
        cu.first_name || ' ' || cu.last_name AS name,  
        a.address,  
        a.postal_code AS "zip code",  
        a.phone,  
        city.city,  
        country.country,  
        CASE  
            WHEN cu.activebool THEN 'active'  
            ELSE ''  
        END AS notes,  
        cu.store_id AS sid  
FROM customer cu  
        INNER JOIN address a USING (address_id)  
        INNER JOIN city USING (city_id)  
        INNER JOIN country USING (country_id);
```

---

и теперь для получения данных о покупателях можно использовать простой Select:

**SELECT**

\*

**FROM**

customer\_master;

*Изменение*

**CREATE OR REPLACE** view\_name

**AS**

query

*Удаление*

**DROP VIEW [ IF EXISTS ]** view\_name;

---

## Различия CTE и VIEW

- Представления могут быть проиндексированы, но ОТВ не могут
- ОТВ отлично работают с рекурсией
- Представления - физические объекты БД, можно обращаться из нескольких запросов:
  - гибкость
  - централизованный подход
- ОТВ - временные:
  - создаются, когда будут использоваться
  - удаляются после использования
  - не хранится статистика на сервере

—

ВРЕМЯ ПРАКТИКИ

## Практика 3

Создайте view с колонками клиент (ФИО; email) и title фильма, который он брал в прокат последним

---

Материализованное  
представление



---

Хранит результат запроса. Засчет этого доступ к информации происходит быстрее, но материализованное представление надо периодически обновлять

**CREATE** MATERIALIZED **VIEW** view\_name

**AS**

query

**WITH [NO] DATA;**

WITH DATA - загрузить данные сразу, WITH NO DATA - позже

*Обновление*

REFRESH MATERIALIZED **VIEW** view\_name;

*Удаление*

**DROP** MATERIALIZED **VIEW** view\_name;

—

# Ускорение запросов: Индексы

---

ускорить запрос можно с помощью создания индексов. Индексы можно создавать на лету

```
CREATE INDEX ON ratings(movieId);
```

**Результат:**

Time: 37427.672 ms (00:37.428)

После того, как индекс создан - запросы начинают выполняться быстрее, время сокращается в сотни раз

```
CREATE INDEX ON ratings(movieId);
```

**Результат:**

```
CREATE INDEX
```

Time: 38493.878 ms (00:38.494)



Выполним запрос ещё раз:

```
SELECT
```

```
    movieId,
```

```
    COUNT(*) num_rating
```

```
FROM public.ratings
```

```
WHERE
```

```
    ratings.movieID > 100000
```

```
GROUP BY 1
```

```
LIMIT 10;
```

**Результат:**

movieid | num\_rating

-----+-----

100001	2
100003	6
100006	6
100008	28
100010	88
100013	18
100015	4
100017	50
100032	30
100034	64

(10 rows)

Time: 5.289 ms

—

# Схема запроса

---

Оператор EXPLAIN демонстрирует этапы выполнения запроса и может быть использован для оптимизации.

```
EXPLAIN
```

```
SELECT
```

```
  userId, COUNT(*) num_rating
```

```
FROM public.links
```

```
LEFT JOIN public.ratings
```

```
  ON links.movieid=ratings.movieid
```

```
GROUP BY 1
```

```
LIMIT 10;
```

## Результат:

### QUERY PLAN

---

Limit (cost=1880431.03..1880431.13 rows=10 width=16)

-> HashAggregate (cost=1880431.03..1880749.83 rows=31880 width=16)

Group Key: ratings.userid

-> Hash Right Join (cost=1323.47..1620188.15 rows=52048576 width=8)

Hash Cond: (ratings.movieid = links.movieid)

-> Seq Scan on ratings (cost=0.00..903196.76 rows=52048576 width=16)

-> Hash (cost=750.43..750.43 rows=45843 width=8)



---

Полезные материалы

## ПОЛЕЗНЫЕ МАТЕРИАЛЫ

---

<https://habr.com/ru/post/269497/>

<https://medium.com/@hakibenita/be-careful-with-cte-in-postgresql-fca5e24d2119>

<https://postgrespro.ru/docs/postgrespro/9.5/using-explain>

<https://habr.com/ru/post/203320/>



НЕТОЛОГИЯ  
групп

Спасибо за  
внимание!

Алексей Кузьмин



[aleksej.kyzmin@gmail.com](mailto:aleksej.kyzmin@gmail.com)