КСиТ

6. Запросы: представления, оконные функции

Базы данных

Хисамутдинов М.А.
кафедра №12 НИЯУ МИФИ
2026

План занятия

1. Представления (VIEW) и материализованные представления (MATERIALIZED VIEW)
2. CTE (именованное табличное выражение): что и зачем
3. Оконные функции: базовые понятия и синтаксис
4. Ранжирующие функции
5. Агрегирующие окна
6. Смещения и окна
7. ROWS BETWEEN и фреймы
8. Рекурсивные запросы

Представление (VIEW)

Представление (VIEW) — это сохраненный SQL-запрос. Снаружи он выглядит как таблица, но данные вычисляются при обращении.

Свойства:

- хранит определение, а не данные
- упрощает доступ и переиспользование
- позволяет скрыть сложность и ограничить доступ
- VIEW сам по себе не “обходит” безопасность: доступ ограничивается моделью прав конкретной СУБД
- на практике проверяются права на VIEW и/или базовые таблицы (зависит от диалекта и режима выполнения)

Представление (VIEW) и таблица

Таблица:

- хранит данные физически
- изменяется операциями изменения данных (DML: `INSERT`, `UPDATE`, `DELETE`)
- имеет стоимость хранения

VIEW:

- не хранит данные
- выполняется при каждом обращении
- стоимость: вычисления, но без хранения

Когда VIEW дает выгоду

- безопасность
  - давать доступ только к нужным колонкам
- стабильный интерфейс доступа
  - прятать детали схемы
- переиспользование
  - единая бизнес-логика
- интеграция
  - “виртуальные” витрины

В современных системах это выглядит так:

- в хранилищах данных VIEW часто задают единый слой представлений
- в BI-инструментах VIEW дают стабильные витрины

Представление: пример

```
CREATE VIEW v_available AS
SELECT
  warehouse_id,
  product_id,
  qty_on_hand,
  qty_reserved,
  (qty_on_hand - qty_reserved) AS qty_available
FROM stock_balance;
```

Вывод:

warehouse_id	product_id	qty_on_hand	qty_reserved	qty_available
1	1	120	10	110
1	2	80	0	80
1	3	NULL	NULL	NULL

MATERIALIZED VIEW

Материализованное представление (MATERIALIZED VIEW, MV) — это таблица с результатом запроса, которая хранится на диске и обновляется отдельно.

Свойства:

- хранит данные физически
- требует обновления (refresh, то есть пересчета и перезаписи данных)
- ускоряет чтение, потому что результат запроса предвычислен и обычно не требует полного пересчета

VIEW и MATERIALIZED VIEW

VIEW:

- читает актуальные данные базовых таблиц в момент выполнения запроса
- чтение = пересчет

MATERIALIZED VIEW (MV):

- может быть неактуален
- чтение = быстро
- требует стратегии обновления

Стратегии обновления MV

- по расписанию (каждый час/день)
  - предсказуемая нагрузка, но данные запаздывают между запусками
- при событии (триггер/ETL, где ETL = извлечение, преобразование и загрузка данных)
  - данные свежее, но растет нагрузка на запись/ETL-контур
- по требованию (refresh on demand, то есть ручной запуск обновления)
  - полный контроль у пользователя или сервиса, но нужен явный

Стратегии обновления MV

1. Полное обновление - блокирует чтение на время обновления

```
REFRESH MATERIALIZED VIEW my_view;
```

2. Конкурентное обновление - старые данные доступны, пока строится новая версия

```
REFRESH MATERIALIZED VIEW CONCURRENTLY my_view;
```

По триггеру обновления:

1. По расписанию

```
SELECT cron.schedule('*/5 * * * *',
  'REFRESH MATERIALIZED VIEW CONCURRENTLY my_view');
```

2. По событию через триггеры (рассмотрим позже)

MATERIALIZED VIEW: пример

```
CREATE MATERIALIZED VIEW mv_daily_sales AS
SELECT
  date_trunc('day', moved_at) AS day,
  sum(qty_delta) AS qty_total
FROM stock_move
GROUP BY 1;

REFRESH MATERIALIZED VIEW mv_daily_sales;
```

Вывод:

day	qty_total
2026-01-10	30
2026-01-11	60
2026-01-13	-3

CTE (Common Table Expression)

CTE (именованное табличное выражение) — временный именованный блок внутри запроса. Его можно использовать как подзапрос несколько раз.

Цели:

- сделать запрос читаемым
- разделить логику на этапы
- повторно использовать подзапросы

CTE: логика этапов

Типичный паттерн:

- этап 1: фильтрация “сырых” данных
- этап 2: агрегация
- этап 3: финальный расчет

Кейс:

- считаем выручку, скидки и валовую маржу по категориям.
- каждый этап расчета оформляем отдельным CTE, чтобы упростить проверку формул.

CTE: структура

```
WITH cte_name AS (
    SELECT ...
    FROM ...
    WHERE ...
)
SELECT *
FROM cte_name;
```


CTE: пример

```sql
WITH base AS (
  SELECT product_id, qty_delta
  FROM stock_move
  WHERE moved_at >= date '2026-01-01'
),
agg AS (
  SELECT product_id, sum(qty_delta) AS total_delta
  FROM base
  GROUP BY product_id
)
SELECT p.sku, a.total_delta
FROM agg a
JOIN product p
ON p.product_id = a.product_id
ORDER BY a.total_delta DESC;
```

CTE: пример (вывод)

sku	total_delta
SKU-001	30
SKU-002	60
SKU-010	-3

CTE и оптимизация

В ряде СУБД (и в зависимости от версии и оптимизатора) CTE могут материализоваться или разворачиваться в основной запрос.

Следствие:

- CTE = читаемость, но не всегда ускорение
- для тяжелых расчетов иногда нужен MV

Оконные функции

Оконные функции добавляют вычисления по группе строк, но исходные строки остаются в результате.

Отличие от GROUP BY:

- GROUP BY уменьшает число строк
- оконные функции сохраняют все строки

Синтаксис окна

Ключевая конструкция:

```
функция() OVER (PARTITION BY ... ORDER BY .. ROWS/RANGE BETWEEN ... AND ...)
```

Компоненты:

- PARTITION BY — делит строки на группы. Функция вычисляется отдельно для каждой группы
- ORDER BY — задаёт порядок строк внутри окна. Влияет на накопительные вычисления и на фрейм по умолчанию
- ROWS/RANGE — уточняет, какие именно строки участвуют в вычислении относительно текущей строки

Синтаксис окна

Если не указывать:

- PARTITION BY — всё множество строк счиается одним окном
- ORDER BY — порядок не определён, и фрейм становится `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Но если указать ORDER BY, то фрейм по умолчанию меняется на `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
- из-за различий диалектов безопаснее явно писать `ROWS BETWEEN ...`

Синтаксис окна

```sql
SELECT name, salary,
    SUM(salary) OVER (
      PARTITION BY department ORDER BY salary
    ) AS running_total,
    SUM(salary) OVER (
      PARTITION BY department
      ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS another_sum
FROM employees;
```

```sql
OVER (
    ORDER BY date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    -- скользящее окно: текущая + 2 предыдущие
)
```


Оконные функции: пример

```
SELECT
  product_id,
  moved_at,
  qty_delta,
  sum(qty_delta) OVER (
    PARTITION BY product_id
    ORDER BY moved_at
  ) AS running_sum
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	running_sum
1	2026-01-01	10	10
1	2026-01-02	-3	7
1	2026-01-03	5	12

Границы фрейма

```
UNBOUNDED PRECEDING   -- от начала окна
N PRECEDING           -- N строк/диапазонов назад
CURRENT ROW           -- текущая строка
N FOLLOWING           -- N строк/диапазонов вперёд
UNBOUNDED FOLLOWING   -- до конца окна
```

Пример разницы ROWS vs RANGE:

```
-- Есть строки с salary: 100, 200, 200, 300
-- ROWS: берёт ровно 1 предыдущую физическую строку
SUM(salary) OVER (ORDER BY salary ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
-- результат для 3-й строки (200): 200 + 200 = 400
-- RANGE: берёт все строки с salary <= текущего значения
SUM(salary) OVER (ORDER BY salary RANGE BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW)
-- результат для 3-й строки (200): 100 + 200 + 200 = 500
```

Где применяются оконные функции

- аналитика продуктового поведения
- финансы: накопительный остаток по счёту
- логистика: накопительные объемы поставок
- мониторинг: ключевые показатели (KPI) по временным окнам
- системы HR/CRM: позиции и ранги в группе

Ранжирующие функции

Типичные:

- `row_number()` — уникальный номер строки: 1, 2, 3, 4
- `rank()` — пропуски при равенстве: 1, 2, 2, 4
- `dense_rank()` — без пропусков: 1, 2, 2, 3
- `ntile(n)` — разбиение на n групп

Кейс:

- нужно показать топ-10 товаров в каждой категории.
- для этого считаем ранг внутри категории и оставляем строки с рангом `<= 10`.

Ранжирование: назначение

Ранжирование отвечает на вопросы:

- “кто лучший в своей группе”
- “какие 5% клиентов дают 80% выручки”
- “как распределить сегменты по группам”

Как работает ранжирование

Ранги строятся по порядку строк внутри `PARTITION BY`. Итог зависит от сортировки `ORDER BY`, наличия равных значений и выбранной функции ранга.

Важно для равных значений:

- если сортировка не задает порядок однозначно, результат может быть недетерминированным
- добавляйте дополнительный критерий в `ORDER BY` (например, `ORDER BY sales DESC, emp_id ASC`)

Пример данных для ранжирования

dept	emp	sales
A	Ира	120
A	Олег	120
A	Павел	90
B	Анна	200
B	Денис	150

row_number()

`row_number()` — нумерует строки по порядку окна. Даже при равных значениях номера разные.

row_number(): пример

```
SELECT dept, emp, sales,
       row_number() OVER (
         PARTITION BY dept
         ORDER BY sales DESC
       ) AS rn
FROM sales_emp;
```

Вывод:

dept	emp	sales	rn
A	Ира	120	1
A	Олег	120	2
A	Павел	90	3
B	Анна	200	1
B	Денис	150	2

row_number(): ещё один вызов

```
SELECT dept, emp, sales,
       row_number() OVER (
         PARTITION BY dept
         ORDER BY sales DESC
       ) AS rn
FROM sales_emp;
```

Вывод:

dept	emp	sales	rn
A	Олег	120	1
A	Ира	120	2
A	Павел	90	3
B	Анна	200	1
B	Денис	150	2

rank()

`rank()` учитывает равные значения: одинаковые получают один ранг, следующий идет с пропуском.

rank(): пример

```
SELECT dept, emp, sales,
       rank() OVER (
         PARTITION BY dept
         ORDER BY sales DESC
       ) AS rnk
FROM sales_emp;
```

Вывод:

dept	emp	sales	rnk
A	Ира	120	1
A	Олег	120	1
A	Павел	90	3
B	Анна	200	1
B	Денис	150	2

dense_rank()

`dense_rank()` как `rank()`, но без пропусков: следующий ранг всегда +1.

dense_rank(): пример

```
SELECT dept, emp, sales,
       dense_rank() OVER (
         PARTITION BY dept
         ORDER BY sales DESC
       ) AS dr
FROM sales_emp;
```

Вывод:

dept	emp	sales	dr
A	Ира	120	1
A	Олег	120	1
A	Павел	90	2
B	Анна	200	1
B	Денис	150	2

ntile(n)

`ntile(n)` делит упорядоченные строки на n групп примерно равного размера.

ntile(2): пример

```
SELECT dept, emp, sales,
       ntile(2) OVER (
         PARTITION BY dept
         ORDER BY sales DESC
       ) AS bucket
FROM sales_emp;
```

Вывод:

dept	emp	sales	bucket
A	Ира	120	1
A	Олег	120	1
A	Павел	90	2
B	Анна	200	1
B	Денис	150	2

ntile(2): пример

```
SELECT dept, emp, sales,
       ntile(2) OVER (
         ORDER BY sales DESC
       ) AS bucket
FROM sales_emp;
```

Вывод:

dept	emp	sales	bucket
B	Денис	150	1
B	Анна	200	1
A	Ира	120	1
A	Олег	120	2
A	Павел	90	2

ntile(2): если строк меньше, чем групп

```
SELECT emp, sales,
    NTILE(5) OVER (ORDER BY sales DESC) AS tile
FROM sales_data
WHERE dept = 'B';
```

Вывод:

dept	emp	sales	bucket
B	Анна	200	1
B	Денис	150	2

Группы 3, 4, 5 просто не появятся.

NULL в ORDER BY

```sql
SELECT emp, sales,
    RANK() OVER (ORDER BY sales DESC) AS rank
FROM (VALUES
    ('Ира', 120),
    ('Олег', NULL),
    ('Павел', 90)
) AS t(emp, sales);
```

emp	sales	rank
Ира	120	1
Павел	90	2
Олег	NULL	3

По умолчанию в PostgreSQL `ORDER BY DESC` → `NULL` идут в конец (`NULLS LAST`).

При `ORDER BY ASC` → `NULL` идут в начало (`NULLS FIRST`).

Пустое OVER()

```
SELECT
  emp,
  sales,
  RANK() OVER () AS rank_empty,
  RANK() OVER (ORDER BY sales DESC) AS rank_ordered
FROM sales_data;
```

emp	sales	rank_empty	rank_ordered
Ира	120	1	3
Олег	120	1	3
Павел	90	1	5
Анна	200	1	1
Денис	150	1	2

Агрегирующие окна

Окна часто используют агрегаты:

- `sum()` — накопительные суммы
- `avg()` — скользящее среднее
- `count()` — количество событий

Кейс:

- считаем 7-дневную скользящую среднюю по продажам.
- это сглаживает выбросы и помогает увидеть общий тренд.

Разница: GROUP BY и окна

GROUP BY:

- 1 строка на группу
- подходит для отчетов “итоги”

Окна:

- строка на событие
- подходит для аналитики “истории” и динамики

Как работают агрегаты в окнах

Агрегат (`sum`, `avg`, `count`) считается по фрейму окна и повторяется для каждой строки внутри этого фрейма.

Пример данных для агрегатов

product_id	moved_at	qty_delta
1	2026-01-01	10
1	2026-01-02	-3
1	2026-01-03	5
1	2026-01-04	8
1	2026-01-05	12

sum(): пример

```
SELECT product_id, moved_at, qty_delta,
       sum(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS running_sum
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	running_sum
1	2026-01-01	10	10
1	2026-01-02	-3	7
1	2026-01-03	5	12
1	2026-01-04	8	20
1	2026-01-05	12	32

avg(): пример

```
SELECT product_id, moved_at, qty_delta,
       round(avg(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
         ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
       ), 2) AS running_avg
FROM stock_move;
```

Вывод: (Скользящее среднее за 3 дня)

product_id	moved_at	qty_delta	running_avg
1	2026-01-01	10	10.0
1	2026-01-02	-3	3.5
1	2026-01-03	5	4.0
1	2026-01-04	8	3.33
1	2026-01-05	12	8.33

count(): пример

```
SELECT product_id, moved_at, qty_delta,
       count(*) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
         ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS running_cnt
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	running_cnt
1	2026-01-01	10	1
1	2026-01-02	-3	2
1	2026-01-03	5	3
1	2026-01-04	8	4
1	2026-01-05	12	5

min(), max(): пример

```
SELECT product_id, moved_at, qty_delta,
    MIN(qty_delta) OVER (ORDER BY moved_at ROWS UNBOUNDED PRECEDING) AS
min_ever,
    MAX(qty_delta) OVER (ORDER BY moved_at ROWS UNBOUNDED PRECEDING) AS
max_ever
FROM stock_move;
```

Вывод: (на каждую дату видно — какой был минимум и максимум за всю историю до этого момента.)

product_id	moved_at	qty_delta	min_ever	max_ever
1	2026-01-01	10	10	10
1	2026-01-02	-3	-3	10
1	2026-01-03	5	-3	10
1	2026-01-04	8	-3	10
1	2026-01-05	12	-3	12

Отклонение от среднего и доля от итога

```sql
SELECT product_id, moved_at, qty_delta,
    ROUND(qty_delta - AVG(qty_delta) OVER (), 2) AS diff_from_avg,
    ROUND(qty_delta * 100.0 / SUM(qty_delta) OVER (), 1) AS pct_of_total
FROM movements;
```

Вывод:

product_id	moved_at	qty_delta	diff_from_avg	pct_of_total
1	2026-01-01	10	3.6	31.3
1	2026-01-02	-3	-9.4	-9.4
1	2026-01-03	5	-1.4	15.6
1	2026-01-04	8	1.6	25.0
1	2026-01-05	12	5.6	37.5

Смещения (LAG/LEAD)

`lag()` и `lead()` берут значения из соседних строк в пределах окна.

Кейсы:

- сравнение с предыдущим месяцем
- вычисление дельт и темпов роста
- поиск “первого” события после другого

NULL и смещения

Важно:

- если соседней строки нет, возвращается NULL
- можно задать значение по умолчанию

Следствие:

- первый элемент в группе часто требует обработки

Как работают LAG/LEAD

`lag()` берет значение из предыдущей строки, `lead()` — из следующей. Оба зависят от порядка в окне и нужны для сравнений во времени.

lag(): пример

```
SELECT product_id, moved_at, qty_delta,
       lag(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
       ) AS prev_delta
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	prev_delta
1	2026-01-01	10	NULL
1	2026-01-02	-3	10
1	2026-01-03	5	-3
1	2026-01-04	8	5
1	2026-01-05	12	8

lead(): пример

```
SELECT product_id, moved_at, qty_delta,
       lead(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
       ) AS next_delta
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	next_delta
1	2026-01-01	10	-3
1	2026-01-02	-3	5
1	2026-01-03	5	8
1	2026-01-04	8	12
1	2026-01-05	12	NULL

Еще примеры

```
SELECT product_id,
       moved_at,
       qty_delta,
       lag(qty_delta, 2) OVER (PARTITION BY product_id ORDER BY moved_at) AS
two_days_ago
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	two_days_ago
1	2026-01-01	10	NULL
1	2026-01-02	-3	NULL
1	2026-01-03	5	10
1	2026-01-04	8	-3
1	2026-01-05	12	5

Еще примеры

```
SELECT product_id,
       moved_at,
       qty_delta,
       lag(qty_delta, 1, -1) OVER (PARTITION BY product_id ORDER BY
moved_at) AS prev_or_default
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	prev_or_default
1	2026-01-01	10	-1
1	2026-01-02	-3	10
1	2026-01-03	5	-3
1	2026-01-04	8	5
1	2026-01-05	12	8

Еще примеры

```
SELECT product_id,
       moved_at,
       qty_delta,
       qty_delta - lag(qty_delta) OVER (PARTITION BY product_id ORDER BY
moved_at) AS diff_from_prev
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	diff_from_prev
1	2026-01-01	10	NULL
1	2026-01-02	-3	-13
1	2026-01-03	5	8
1	2026-01-04	8	3
1	2026-01-05	12	4

Еще примеры

```
SELECT product_id, moved_at, qty_delta,
       ROUND(
         (qty_delta - lag(qty_delta) OVER (PARTITION BY product_id ORDER BY
moved_at))::numeric
         / NULLIF(lag(qty_delta) OVER (PARTITION BY product_id ORDER BY
moved_at), 0) * 100, 1
       ) AS pct_change
FROM stock_move;
```

Вывод:

product_id	moved_at	qty_delta	pct_change
1	2026-01-01	10	NULL
1	2026-01-02	-3	-130
1	2026-01-03	5	-266,7
1	2026-01-04	8	60
1	2026-01-05	12	50

NULLIF

`NULLIF(a, b)` — возвращает NULL если a = b, иначе возвращает a как есть.

Эквивалентно:

```
CASE WHEN a = b THEN NULL ELSE a END
```

Нужен для защиты от деления на ноль:

```
100.0 / NULLIF(qty, 0)  -- вернёт NULL если qty = 0, а не ошибку
```

Убрать пустые строки:

```
NULLIF(trim(name), '')  -- вернёт NULL если name состоит из пробелов
```

Важно! Оба аргумента должны быть одного типа.

ROWS BETWEEN: фрейм окна

Фрейм определяет, какие строки участвуют в расчете.

Примеры:

- `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` — накопление от начала раздела до текущей строки
- `ROWS BETWEEN 6 PRECEDING AND CURRENT ROW` — скользящее окно из 7 строк

ROWS и RANGE: различия

`ROWS` — считает строки физически.

`RANGE` — считает по значению ORDER BY (например, все события с одной датой попадают вместе).

При одинаковом `ORDER BY`-значении разница видна так:

- `ROWS ... CURRENT ROW` увеличивает агрегат построчно
- `RANGE ... CURRENT ROW` дает одинаковый результат всем строкам с одной и той же датой/ключом

Кейс:

- в финансовых данных на одну дату часто приходится несколько операций.
- `RANGE` учитывает такие строки вместе и дает корректный итог по дню.

Как работает ROWS BETWEEN

Фрейм ограничивает набор строк для агрегата. Можно включать предыдущие, текущую и следующие строки.

Пример данных для ROWS/RANGE

product_id	moved_at	qty_delta
1	2026-01-01	10
1	2026-01-02	-3
1	2026-01-03	5
1	2026-01-03	4
1	2026-01-04	2

ROWS BETWEEN: пример

```
SELECT product_id, moved_at, qty_delta,
       sum(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
         ROWS BETWEEN 1 PRECEDING AND CURRENT ROW
       ) AS sum_last_2
FROM stock_move;
```

product_id	moved_at	qty_delta	sum_last_2
1	2026-01-01	10	10
1	2026-01-02	-3	7
1	2026-01-03	5	2
1	2026-01-03	4	9
1	2026-01-04	2	6

RANGE

`RANGE` объединяет строки с одинаковым значением `ORDER BY`, поэтому фрейм может включать несколько строк на одну дату/время.

RANGE: пример

```
SELECT product_id, moved_at, qty_delta,
       sum(qty_delta) OVER (
         PARTITION BY product_id
         ORDER BY moved_at
         RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
       ) AS sum_by_date
FROM stock_move;
```

RANGE: пример

Вывод (если на одной дате несколько строк — все они попадают в один фрейм):

product_id	moved_at	qty_delta	sum_by_date
1	2026-01-01	10	10
1	2026-01-02	-3	7
1	2026-01-03	5	16
1	2026-01-03	4	16
1	2026-01-04	2	18

Рекурсивные запросы

Рекурсивные CTE позволяют обходить иерархии и графы:

- организационные структуры
- каталоги товаров
- граф зависимостей
- родословные, связи в соцграфе

На практике работа с иерархиями и графами выглядит так:

- в продуктовых каталогах и BOM (спецификация состава изделия) рекурсия — стандартный прием
- для графов на масштабе часто используют отдельные графовые СУБД (graph DB, базы для работы с графовыми связями), но SQL-рекурсия остается важной для отчетов

Принцип рекурсии в SQL

Два блока:

- anchor (базовая часть)
- рекурсивная часть (шаг)

Идея:

- из базовых строк строим новые, пока есть что добавлять

Рекурсивные запросы: ограничения

- важно контролировать глубину
- возможны циклы (нужна защита)
- производительность зависит от объема связей

Как останавливается рекурсия и как защититься:

- рекурсия завершается, когда рекурсивная часть перестает возвращать новые строки
- ограничение глубины обычно задают явно через поле `depth` и условие вида `WHERE depth < N`
- защиту от циклов делают через контроль уже посещенных узлов (например, в `path`) или встроенные механизмы СУБД
- в части СУБД есть отдельные ограничения глубины на уровне запроса/настроек, в других это контролируют только в SQL

Рекурсивные запросы: ограничения

Кейс:

- нужно получить всех подчиненных сотрудника, начиная с руководителя.
- глубину ограничиваем 5 уровнями, чтобы исключить бесконечный обход при циклах.

Рекурсивный CTE: пример

```
WITH RECURSIVE tree AS (
  SELECT category_id, parent_id, name
  FROM category_tree
  WHERE category_id = 10
  UNION ALL
  SELECT c.category_id, c.parent_id, c.name
  FROM category_tree c
  JOIN tree t ON t.category_id = c.parent_id
)
SELECT * FROM tree;
```

Вывод:

category_id	parent_id	name
10	NULL	Электроника
11	10	Кабели
12	11	USB

Рекурсивный CTE: безопасный шаблон

```
WITH RECURSIVE tree AS (
  SELECT category_id, parent_id, name,
    1 AS depth,
    '/' || category_id || '/' AS path
  FROM category_tree
  WHERE category_id = 10
  UNION ALL
  SELECT c.category_id, c.parent_id, c.name,
    t.depth + 1 AS depth,
    t.path || c.category_id || '/' AS path
  FROM category_tree c
  JOIN tree t ON t.category_id = c.parent_id
  WHERE t.depth < 5
    AND position('/' || c.category_id || '/' in t.path) = 0
)
SELECT category_id, parent_id, name, depth
FROM tree;
```

Рекурсивный CTE: безопасный шаблон

Идея: `depth < 5` ограничивает глубину, а проверка `path` не дает зациклиться.

category_id	parent_id	name	depth	path
10	NULL	Электроника	1	/10/
11	10	Кабели	2	/10/11/
12	11	USB	3	/10/11/12

Современные кейсы

- материализованные представления для аналитических дашбордов продаж
- цепочки именованных подзапросов (CTE) для многоэтапных расчетов в финтехе
- окна для метрик в продуктовой аналитике
- функции смещения (`LAG/LEAD`) для мониторинга SLA (соглашения об уровне сервиса)
- рекурсия для спецификаций состава изделия (BOM) и цепочек поставок