Рекурсии, CTE

Files & media 

▼ Common Table Expressions (CTE)

CTE (Common Table Expressions) — это временные результаты, которые можно использовать в пределах одного SQL-запроса. CTE удобны для **упрощения сложных запросов, улучшения читаемости и повышения производительности**.

1. Синтаксис CTE:

```
WITH cte_name AS (
    SELECT columns
    FROM table
    WHERE conditions
)
SELECT *
FROM cte_name;
```

- **WITH** — ключевое слово для объявления CTE.
- **cte_name** — имя временной таблицы (это может быть любое имя).
- **AS** — показывает, что далее идёт запрос, создающий временный набор данных.
- После создания CTE, можно использовать её как любую таблицу в основном запросе.

2. Пример простого CTE:

```
WITH employee_cte AS (
    SELECT employee_id, first_name, department_id
    FROM employees
```

```
    WHERE department_id = 1
)
SELECT *
FROM employee_cte;
```

Здесь создаётся временная таблица `employee_cte`, которая содержит только тех сотрудников, у которых `department_id = 1`. Затем основной запрос выбирает все строки из этого CTE.

▼ Матеарилизация

Обрати внимание!

До PostgreSQL 12:

- Когда ты создаёшь CTE, PostgreSQL вычисляет её один раз и сохраняет результат во временной таблице (это называется **материализация**).
- Если CTE использовалась только один раз, её материализация могла замедлять выполнение запроса, потому что данные сначала сохранялись, а потом использовались.

После PostgreSQL 12:

- PostgreSQL стал умнее! Теперь CTE **по умолчанию не материализуются**, если это не нужно. Это значит, что они работают как обычные **подзапросы (то есть оптимизатор их воспринимает так)**, и результат не сохраняется отдельно, что делает запросы быстрее.
- Если всё-таки нужно, чтобы CTE был материализован (например, если он используется несколько раз), ты можешь это указать явно с помощью ключевого слова `MATERIALIZED`.

Чтобы узнать, использует ли **Common Table Expression (CTE)** материализацию в PostgreSQL, можно воспользоваться командой `EXPLAIN`, которая показывает план выполнения запроса.

```
....
        ->  Seq Scan on employees e  (cost=...)
        ->  Hash  (cost=...)
              ->  CTE Scan on avg_salaries avg_s  (cost=.
 ....

   ->  Seq Scan on employees e  (cost=...)
   ->  Hash  (cost=...)
         ->  HashAggregate  (cost=...)
               Group Key: employees.department_id
               ->  Seq Scan on employees  (cost=...) --ниче
```

Где хранится CTE:

1. **Если данные небольшие**:
   - Если объем данных, возвращаемых CTE, относительно мал, PostgreSQL пытается хранить их в **оперативной памяти (RAM)**. Это обеспечивает более быструю обработку данных и улучшает производительность.
   - Память для временных данных CTE управляется параметром `work_mem`. Если объем данных CTE меньше значения этого параметра, данные остаются в памяти.
2. **Если данные большие**:
   - Если данные CTE превышают лимит, установленный для `work_mem`, PostgreSQL начинает сбрасывать их на **диск**, чтобы освободить оперативную память.
   - В таких случаях данные CTE сохраняются во временных файлах на диске, что может замедлить работу запроса, так как операции ввода-вывода (I/O) на диске медленнее, чем в памяти.

▼ Рекурсивные CTE

Рекурсивные CTE позволяют обрабатывать данные, которые зависят от самих себя, например, для иерархических данных (деревьев, графов) — такие, как структура подчинённых сотрудников или структура каталогов.

1. Синтаксис рекурсивного CTE:

```
WITH RECURSIVE cte_name AS (
    -- Анкерный запрос (начало рекурсии)
    SELECT columns
    FROM table
    WHERE base_condition

    UNION ALL

    -- Рекурсивный запрос
    SELECT columns
    FROM table
    JOIN cte_name ON conditions
)
SELECT *
FROM cte_name;
```

- **RECURSIVE** — указывает, что CTE будет рекурсивным.
- Анкерный запрос выполняется один раз и формирует базовый набор данных.
- Рекурсивный запрос выполняется на каждом шаге до тех пор, пока не перестанут добавляться новые строки.

2. Пример рекурсивного CTE:

Предположим, у нас есть таблица `employees`, которая содержит данные о сотрудниках и их менеджерах (у каждого сотрудника есть ссылка на менеджера через `manager_id`).

```
WITH RECURSIVE hierarchy AS (
    -- Анкерный запрос: начнем с верхнего уровня (наприм
ер, CEO, у кого нет менеджера)
    SELECT employee_id, first_name, manager_id, 1 AS lev
el
    FROM employees
```

```
    WHERE manager_id IS NULL

    UNION ALL

    -- Рекурсивный запрос: выберем подчинённых текущего
уровня
    SELECT e.employee_id, e.first_name, e.manager_id, h.
level + 1
    FROM employees e
    JOIN hierarchy h ON e.manager_id = h.employee_id
)
SELECT *
FROM hierarchy;
```

В этом примере CTE строит иерархию сотрудников, начиная с тех, у кого нет менеджера (CEO), и затем рекурсивно выбирает всех подчинённых на каждом уровне.

Важные моменты по рекурсивным CTE:

- Рекурсивные запросы обычно используют оператор `UNION ALL`, чтобы объединять результаты анкерного и рекурсивного запросов.
- Можно ограничить количество итераций рекурсивного запроса с помощью `LIMIT` или добавить условие, чтобы избежать бесконечной рекурсии.

Пример реальной задачи на собеседовании:

Задача: Построить дерево категорий товаров, начиная с верхнего уровня иерархии (категории без родителя), и показать подкатегории.

```
WITH RECURSIVE category_tree AS (
    -- Начинаем с корневых категорий (те, у которых нет
родителя)
    SELECT category_id, category_name, parent_category_i
d, 1 AS level
    FROM categories
    WHERE parent_category_id IS NULL
```

```
    UNION ALL

    -- Рекурсивно выбираем подкатегории
    SELECT c.category_id, c.category_name, c.parent_cate
gory_id, ct.level + 1
    FROM categories c
    JOIN category_tree ct ON c.parent_category_id = ct.c
ategory_id
)
SELECT *
FROM category_tree
ORDER BY level;
```

▼ Views (Представления)

Представление — это виртуальная таблица, которая не хранит данные физически, а предоставляет результат запроса.

- Представление является динамическим: когда к нему обращаются, выполняется запрос, на основе которого оно создано.
- Представления используются для упрощения сложных запросов и для ограничения доступа к данным.

▼ Нематериализованные

Пример представления:

```
CREATE VIEW active_employees AS
SELECT employee_id, first_name, last_name
FROM employees
WHERE is_active = true;
```

✅Преимущества представлений:

- **Чистота кода**: Упрощают сложные запросы, скрывая детали выборки данных.
- **Безопасность**: Ограничивают доступ к определённым данным, предоставляя пользователям только нужные столбцы.
- **Динамическое обновление**: Автоматически обновляется при изменении данных в базовых таблицах.

❌Недостатки представлений:

- Не могут использоваться для сохранения данных.
- Представления с очень сложными запросами могут замедлять выполнение.
- Некоторыми представлениями нельзя манипулировать (вставка, обновление, удаление) в зависимости от сложности запроса.

Обрати внимание! Проблемы с производительностью!

- Если представление включает сложные запросы с многократными соединениями (`JOIN`), подзапросами, агрегациями и фильтрацией, то эти операции также включаются в запрос и должны выполняться каждый раз, когда ты запрашиваешь данные из представления. Это может замедлить выполнение, так как при каждом вызове представления выполняется весь запрос, даже если часть данных уже могла быть использована ранее.
- PostgreSQL не всегда может **оптимизировать сложные представления** так же, как **обычные запросы**. Это связано с тем, что представления могут содержать сложные запросы, которые планировщик запросов не может эффективно разбить на части и оптимизировать отдельно.
- **Представления сами по себе не могут иметь индексы**. Индексы могут быть созданы только на реальные таблицы. Это означает, что любые сложные операции, такие как сортировка, фильтрация и объединение данных, не могут использовать индексы напрямую через представление. Если в представление включены сложные объединения, фильтрации или агрегации, производительность может ухудшиться, особенно при отсутствии индексов на соответствующих столбцах в базовых таблицах.

▼ Материализованные

Материализованное представление — это представление, которое сохраняет результат запроса на диск, и может обновляться вручную или по расписанию.

- В отличие от обычного представления, материализованное представление сохраняет результат запроса, что позволяет выполнять запросы быстрее, так как данные уже подготовлены и сохранены.
- **Используется для улучшения производительности** при сложных запросах или данных, которые изменяются не очень часто.

Пример материализованного представления:

```
CREATE MATERIALIZED VIEW active_employees_mv AS
SELECT employee_id, first_name, last_name
FROM employees
WHERE is_active = true;
```

✅ Преимущества материализованных представлений:

- **Повышенная производительность**: Запросы выполняются быстрее, так как данные уже сохранены на диске и не нужно каждый раз выполнять сложные вычисления.
- **Можно создавать индексы**: На материализованное представление можно создавать индексы для дальнейшего ускорения запросов.
- **Удобно для редко изменяющихся данных**: Если данные изменяются нечасто, то материализованные представления позволяют существенно ускорить работу с этими данными.

❌ Недостатки материализованных представлений:

- **Неавтоматическое обновление данных**: В отличие от обычных представлений, данные в материализованных представлениях не обновляются

автоматически. Нужно вручную обновлять представление командой `REFRESH MATERIALIZED VIEW`.

- **Занимают место на диске**: Поскольку результат запроса сохраняется, материализованное представление требует дополнительного дискового пространства.

Обновление материализованного представления:

```
REFRESH MATERIALIZED VIEW active_employees_mv;
```

Когда использовать обычные представления и материализованные:

- **Обычные представления** хороши, когда данные часто изменяются и необходимо всегда получать актуальные данные.
- **Материализованные представления** подходят, когда данные редко изменяются, но нужно ускорить выполнение сложных запросов или часто обращаться к одному и тому же результату запроса.

Как выбрать:

- Если важна **актуальность данных** и нет особых проблем с производительностью — используй обычные представления.
- Если важна **производительность** и данные обновляются редко — используй материализованные представления.

▼ ⚔️ CTE VS View

Когда использовать CTE:

- Когда нужно временно разделить сложный запрос на части.
- Когда нужна **рекурсия**.
- Когда вам нужно временное хранилище для результата запроса, который не будет использоваться повторно.

Когда использовать View:

- Когда нужно повторно использовать сложный запрос.
- Когда нужно скрыть сложные запросы от конечного пользователя или обеспечить уровень абстракции.
- Когда нужно организовать доступ к данным через более простой интерфейс.

▼ Subquery (Подзапрос):

 **Подзапрос** — это вложенный запрос внутри основного запроса. Он выполняется один раз, а затем результат используется в основном запросе.

Пример подзапроса:

```
SELECT first_name, last_name
FROM employees
WHERE department_id = (SELECT department_id
                       FROM departments
                       WHERE department_name = 'Sales');
```

✅ Преимущества подзапросов:

- **Гибкость**: Подзапросы можно использовать в разных частях SQL-запроса, что делает их очень гибкими.
- **Часто эффективны**: PostgreSQL может оптимизировать выполнение подзапросов, и иногда они могут работать быстрее, чем CTE.

❌ Недостатки подзапросов:

- **Читаемость**: Подзапросы, особенно вложенные, могут усложнять читаемость SQL-запросов.
- **Повторный расчет**: В некоторых случаях подзапросы могут пересчитываться несколько раз, что снижает производительность (если субзапрос не оптимизирован).

Где можно использовать подзапрос?

- **В операторе** `SELECT` (в выражении `SELECT` или в списке возвращаемых полей).
- **В операторе** `WHERE` (для фильтрации результатов на основе условий).
- **В операторе** `FROM` (как временная таблица).
- **В операторе** `HAVING` (для фильтрации агрегированных данных).
- **В операторе** `JOIN` (вместо таблицы для соединения с другими данными).
- **В операторе** `EXISTS` (для проверки наличия данных).
- **В операторе** `IN` (для проверки, входит ли значение в список).
- **В операторе** `UPDATE` (для определения значений в обновляемых полях).
- **В операторе** `DELETE` (для удаления данных на основе подзапроса).
- **В операторе** `INSERT INTO ... SELECT` (для вставки данных, полученных из подзапроса).

▼ ⚔️ CTE vs Subquery:

- **Читаемость**: CTE предпочтительнее для упрощения сложных запросов и улучшения читаемости. Подзапросы могут быть сложными для восприятия при многократном вложении.
- **Использование результатов**: CTE можно использовать несколько раз в одном запросе, в то время как результат подзапроса обычно используется один раз.
- **Производительность**: Подзапросы могут быть более производительными в простых случаях, но в сложных запросах CTE может оказаться эффективнее благодаря структурированности и возможности разбить задачи на этапы.
- **Рекурсия**: CTE поддерживают рекурсию, а подзапросы — нет.