Санкт-Петербургский Национальный Исследовательский Университет ИТМО

Факультет Программной Инженерии и Компьютерной Техники



Вариант № 23445

Лабораторная работа №4

По дисциплине

Базы Данных

Выполнил студент группы P3119:

Ануфриев Андрей Сергеевич

Преподаватель:

Бойко Владислав Алексеевич

Санкт-Петербург 2025 г.

Оглавление

[Текст задания 4](#_Toc199711917)

[Первый запрос 4](#_Toc199711918)

[Реализация запрос 4](#_Toc199711919)

[Добавление индекса 5](#_Toc199711920)

[План 5](#_Toc199711921)

[Вывод EXPLAIN 6](#_Toc199711922)

[Второй запрос 7](#_Toc199711923)

[Реализация запроса 7](#_Toc199711924)

[Добавление индекса 8](#_Toc199711925)

[План 8](#_Toc199711926)

[Вывод EXPLAIN 9](#_Toc199711927)

[Вывод 11](#_Toc199711928)

[Ответы на вопросы 11](#_Toc199711929)

[**Типы соединения** 11](#_Toc199711930)

[Nested Loop 11](#_Toc199711931)

[Hash Join 12](#_Toc199711932)

[Merge Join 12](#_Toc199711933)

[Parallel Hash Join (Параллельное хеш-соединение) 12](#_Toc199711934)

[Parallel Nested Loop (Параллельные вложенные циклы) 13](#_Toc199711935)

[Bitmap Heap Scan + Bitmap Index Scan 13](#_Toc199711936)

[Semi Join и Anti Join 14](#_Toc199711937)

[Materialized Join (Материализованное соединение) 15](#_Toc199711938)

[LATERAL Join (Латеральное соединение) 15](#_Toc199711939)

[Index Only Join (Только индексное соединение) 16](#_Toc199711940)

[TID Scan Join (Соединение по идентификаторам кортежей) 16](#_Toc199711941)

[**Методы доступа** 17](#_Toc199711942)

[Index Only Scan 17](#_Toc199711943)

[Index Scan 17](#_Toc199711944)

[Bitmap Index Scan + Bitmap Heap Scan 17](#_Toc199711945)

[Sequential Scan (Seq Scan) 18](#_Toc199711946)

[Tid Scan (Сканирование по физическому адресу строки) 18](#_Toc199711947)

[Parallel Scan (Параллельное сканирование) 18](#_Toc199711948)

[**Типы индексов** 19](#_Toc199711949)

[1. B-Tree (Балансированное дерево) 19](#_Toc199711950)

[2. Hash (Хеш-индекс) 19](#_Toc199711951)

[3. GiST (Generalized Search Tree) 19](#_Toc199711952)

[4. SP-GiST (Space-Partitioned GiST) 20](#_Toc199711953)

[5. GIN (Generalized Inverted Index) 20](#_Toc199711954)

[6. BRIN (Block Range INdex) 21](#_Toc199711955)

[7. Bloom (Фильтр Блума) 21](#_Toc199711956)

[8. RUM (расширение GIN) 21](#_Toc199711957)

[9. Partial (Частичный индекс) 22](#_Toc199711958)

[10. Covering (Покрывающий индекс) 22](#_Toc199711959)

[11. Functional (Функциональный индекс) 22](#_Toc199711960)

[**Прочие вопросы** 23](#_Toc199711961)

[Heap Blocks 23](#_Toc199711962)

[Buckets (Корзины/Бакеты) 23](#_Toc199711963)

[Batches (Пакетная обработка) 24](#_Toc199711964)

[Когда индексы неэффективны или невыгодны? 24](#_Toc199711965)

[Что такое Recheck Cond? 25](#_Toc199711966)

[**Режимы кэширования (Cache Modes)** 25](#_Toc199711967)

[1.1. Write-Through (Сквозная запись) 25](#_Toc199711968)

[Алгоритмы вытеснения данных их кэша (Eviction Policies) 27](#_Toc199711969)

[2.1. LRU (Least Recently Used) 27](#_Toc199711970)

[2.2. LFU (Least Frequently Used) 27](#_Toc199711971)

[2.3. FIFO (First-In, First-Out) 28](#_Toc199711972)

[2.4. LIRS (Low Inter-reference Recency Set) 28](#_Toc199711973)

[2.5. ARC (Adaptive Replacement Cache) 28](#_Toc199711974)

[Переполнение кэша (Cache Overflows) 29](#_Toc199711975)

# Текст задания

Для каждого запроса предложить индексы, добавление которых уменьшит время выполнения запроса (указать таблицы/атрибуты, для которых нужно добавить индексы, написать тип индекса; объяснить, почему добавление индекса будет полезным для данного запроса).

Для запросов 1-2 необходимо составить возможные планы выполнения запросов. Планы составляются на основании предположения, что в таблицах отсутствуют индексы. Из составленных планов необходимо выбрать оптимальный и объяснить свой выбор.  
Изменятся ли планы при добавлении индекса и как?

Для запросов 1-2 необходимо добавить в отчет вывод команды EXPLAIN ANALYZE [запрос]

Подробные ответы на все вышеперечисленные вопросы должны присутствовать в отчете (планы выполнения запросов должны быть нарисованы, ответы на вопросы - представлены в текстовом виде).

1. Сделать запрос для получения атрибутов из указанных таблиц, применив фильтры по указанным условиям:  
   Таблицы: Н\_ЛЮДИ, Н\_СЕССИЯ.  
   Вывести атрибуты: Н\_ЛЮДИ.ИД, Н\_СЕССИЯ.ДАТА.  
   Фильтры (AND):  
   a) Н\_ЛЮДИ.ИД = 100865.  
   b) Н\_СЕССИЯ.УЧГОД > 2001/2002.  
   Вид соединения: LEFT JOIN.
2. Сделать запрос для получения атрибутов из указанных таблиц, применив фильтры по указанным условиям:  
   Таблицы: Н\_ЛЮДИ, Н\_ВЕДОМОСТИ, Н\_СЕССИЯ.  
   Вывести атрибуты: Н\_ЛЮДИ.ИД, Н\_ВЕДОМОСТИ.ДАТА, Н\_СЕССИЯ.ЧЛВК\_ИД.  
   Фильтры (AND):  
   a) Н\_ЛЮДИ.ФАМИЛИЯ = Ёлкин.  
   b) Н\_ВЕДОМОСТИ.ИД = 1457443.  
   c) Н\_СЕССИЯ.УЧГОД = 2011/2012.  
   Вид соединения: RIGHT JOIN.

# Первый запрос

## Реализация запрос

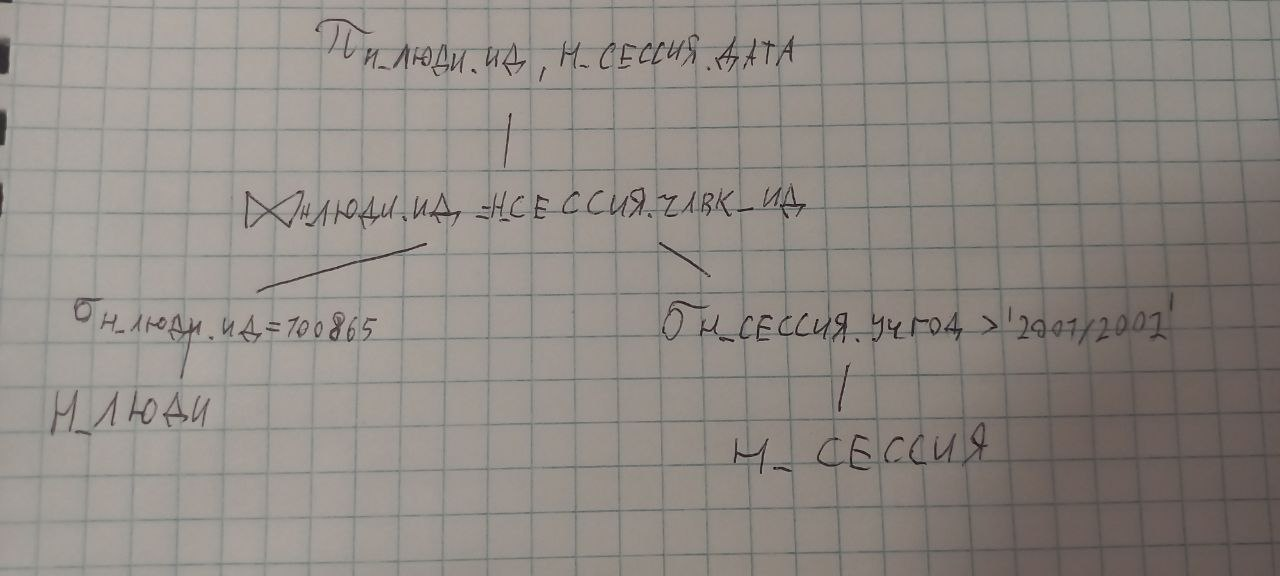
SELECT Н\_ЛЮДИ.ИД, Н\_СЕССИЯ.ДАТА  
FROM Н\_ЛЮДИ  
LEFT JOIN Н\_СЕССИЯ ON Н\_ЛЮДИ.ИД = Н\_СЕССИЯ.ЧЛВК\_ИД  
WHERE Н\_ЛЮДИ.ИД = 100865  
AND Н\_СЕССИЯ.УЧГОД > '2001/2002';

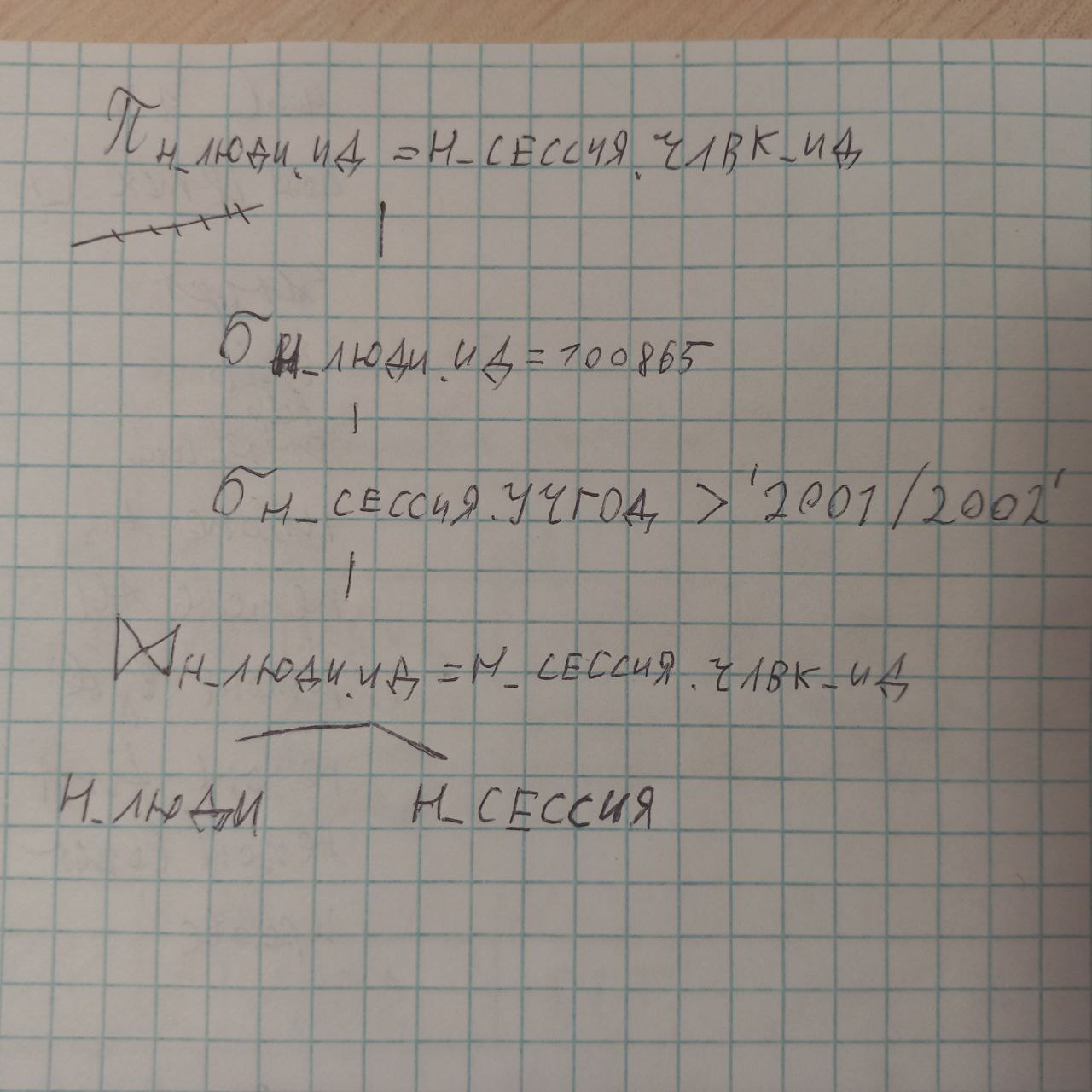
## Добавление индекса

Ускорить этот запрос мог бы индекс таблицы Н\_ЛЮДИ по ИД типа Hash.

Так как этот атрибут участвует в точном сравнение (= 100865). Используя hash субд сможет быстро найти нужные строки в Hash таблице.

## План

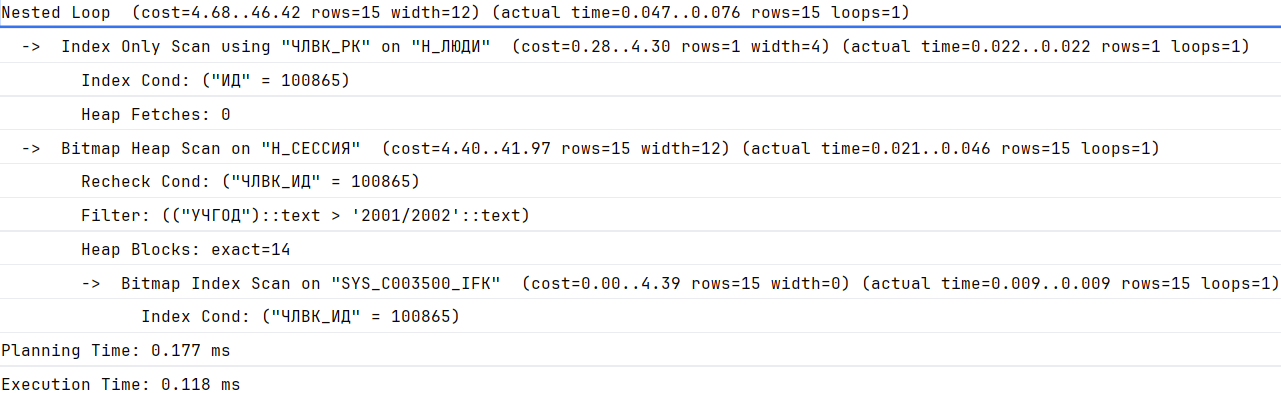




Лучшим является первый вариант, т.к при исполнении JOIN обрабатывается меньше строк, т.к на его вход будут идти только подходящие строки.

При добавление индекса, схема плана не изменяется, т.к индекс работает только в выборке из н\_люди просто находит соответствие в Hash таблице и возвращает все, которые указывают на него.

## Вывод EXPLAIN



Nested Loop (внешний цикл)

│

├── Index Only Scan по Н\_ЛЮДИ (используя индекс ЧЛВК\_PK)

│ │

│ └── Точный поиск по ИД = 100865 (данные только из индекса)

│

└── Bitmap Heap Scan по Н\_СЕССИЯ

│

├── Bitmap Index Scan (используя индекс SYS\_C003500\_IFK)

│ │

│ └── Условие: ЧЛВК\_ИД = 100865

│

└── Дополнительный фильтр: УЧГОД > '2001/2002'

Nested Loop (cost=4.68..46.42 rows=15 width=12) (actual time=0.047..0.076 rows=15 loops=1)

Nested Loop - алгоритм соединения таблиц (вложенный цикл)

cost=4.68..46.42 - оценка затрат (первой строки..всех строк) в условных единицах

rows=15 - ожидаемое количество строк результата

width=12 - ожидаемый средний размер строки в байтах

actual time=0.047..0.076 - реальное время выполнения (мс)

rows=15 - фактическое количество строк результата

loops=1 - количество итераций алгоритма

Первая часть соединения (внешний цикл)

Index Only Scan using ""ЧЛВК\_PK"" on ""Н\_ЛЮДИ"" (cost=0.28..4.30 rows=1 width=4) (actual time=0.022..0.022 rows=1 loops=1)"

Index Cond: (""ИД"" = 100865)"

Heap Fetches: 0

Index Only Scan - сканирование только индекса (данные не читаются из таблицы)

using "ЧЛВК\_PK" - используется первичный ключевой индекс

on "Н\_ЛЮДИ" - таблица, к которой обращаемся

cost=0.28..4.30 - оценка затрат на эту операцию

rows=1 - ожидается 1 строка

width=4 - размер строки 4 байта

actual time=0.022..0.022 - реальное время выполнения

rows=1 - действительно получена 1 строка

Index Cond: ("ИД" = 100865) - условие поиска по индексу

Heap Fetches: 0 - не потребовалось обращений к основной таблице (только индекс)

Вторая часть соединения (внутренний цикл)

Bitmap Heap Scan on ""Н\_СЕССИЯ"" (cost=4.40..41.97 rows=15 width=12) (actual time=0.021..0.046 rows=15 loops=1)"

" Recheck Cond: (""ЧЛВК\_ИД"" = 100865)"

" Filter: ((""УЧГОД"")::text > '2001/2002'::text)"

Heap Blocks: exact=14

" -> Bitmap Index Scan on ""SYS\_C003500\_IFK"" (cost=0.00..4.39 rows=15 width=0) (actual time=0.009..0.009 rows=15 loops=1)"

" Index Cond: (""ЧЛВК\_ИД"" = 100865)"

Bitmap Heap Scan - сканирование таблицы с использованием битовой карты

on "Н\_СЕССИЯ" - таблица, к которой обращаемся

cost=4.40..41.97 - оценка затрат

rows=15 - ожидается 15 строк

width=12 - размер строки 12 байт

actual time=0.021..0.046 - реальное время выполнения

rows=15 - получено 15 строк

Recheck Cond - повторная проверка условия (после битового сканирования)

Filter - дополнительное условие фильтрации

Heap Blocks: exact=14 - точно прочитано 14 блоков таблицы

Bitmap Index Scan - сканирование битового индекса

on "SYS\_C003500\_IFK" - имя используемого индекса

Index Cond - условие поиска по индексу

# Второй запрос

## Реализация запроса

SELECT  
 Н\_ЛЮДИ.ИД,  
 Н\_ВЕДОМОСТИ.ДАТА,  
 Н\_СЕССИЯ.ЧЛВК\_ИД  
FROM  
 Н\_ВЕДОМОСТИ  
RIGHT JOIN Н\_ЛЮДИ ON Н\_ВЕДОМОСТИ.ЧЛВК\_ИД = Н\_ЛЮДИ.ИД  
RIGHT JOIN Н\_СЕССИЯ ON Н\_ЛЮДИ.ИД = Н\_СЕССИЯ.ЧЛВК\_ИД  
WHERE  
 Н\_ЛЮДИ.ФАМИЛИЯ > 'Ёлкин' *--конкретно такого нет* AND (Н\_ВЕДОМОСТИ.ИД > 69965 ) *--1214405 максимальное* AND (Н\_СЕССИЯ.УЧГОД = '2011/2012');

## Добавление индекса

Можно добавить индекс B-Tree в таблицу Н\_ЛЮДИ для атрибутов ИД

CREATE INDEX idx\_Н\_ЛЮДИ\_ИД ON Н\_ЛЮДИ USING BTREE (ИД);

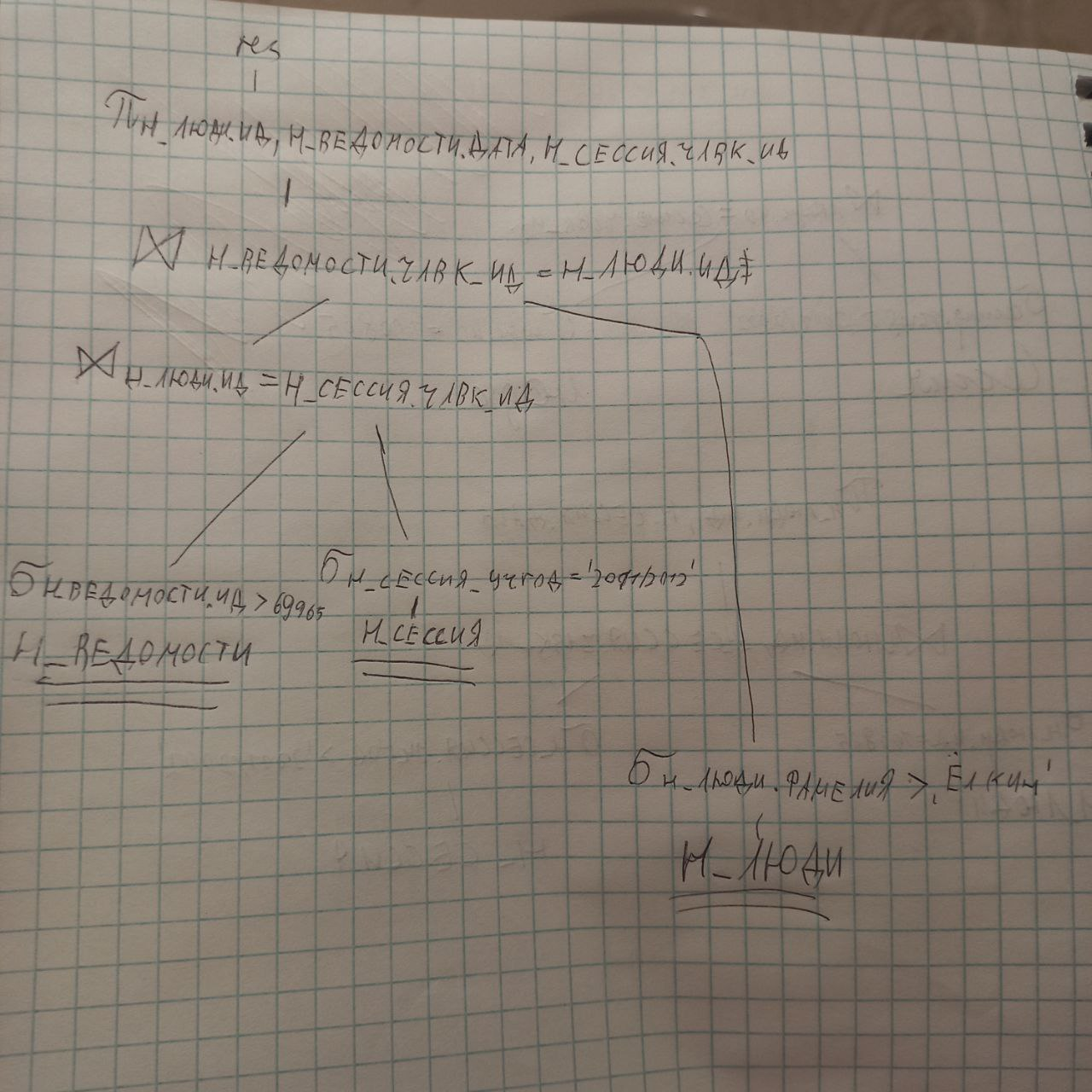
Также можно создать такой же индекс для таблицы Н\_ВЕДОМОСТИ и Н\_СЕССИЯ для атрибутов ЧЛВК\_ИД

CREATE INDEX idx\_Н\_ВЕДОМОСТИ\_ЧЛВК\_ИД ON Н\_ВЕДОМОСТИ USING BTREE (ЧЛВК\_ИД);

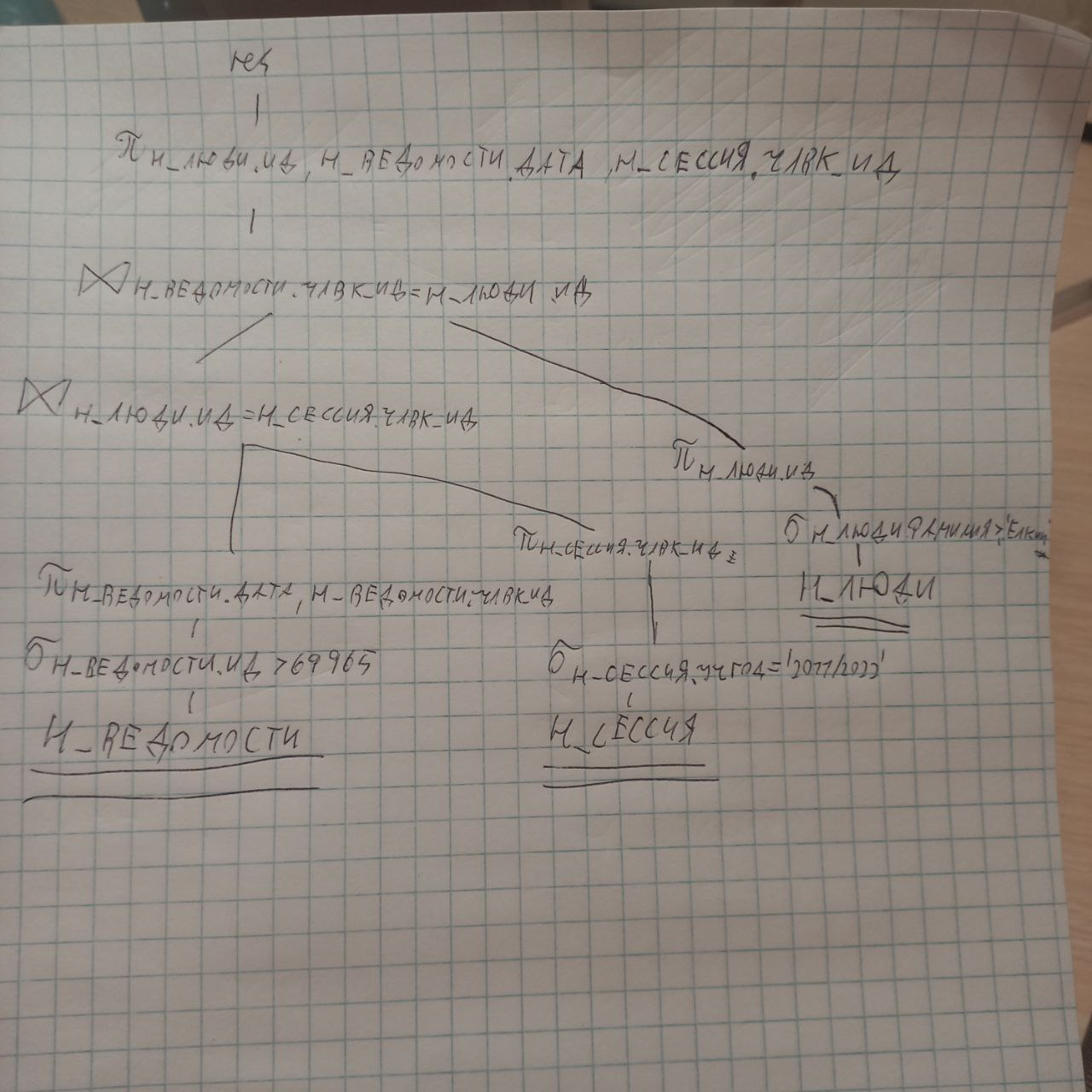
CREATE INDEX idx\_Н\_СЕССИЯ\_ЧЛВК\_ИД ON Н\_СЕССИЯ USING BTREE (ЧЛВК\_ИД);

B-Tree эффективен для сортировки и сравнения строк. Эти индексы ускорят операции JOIN и WHERE. Так как в дереве можно быстрее найти нужное значение и также понятно где брать следующие значения (для < > и т.д)

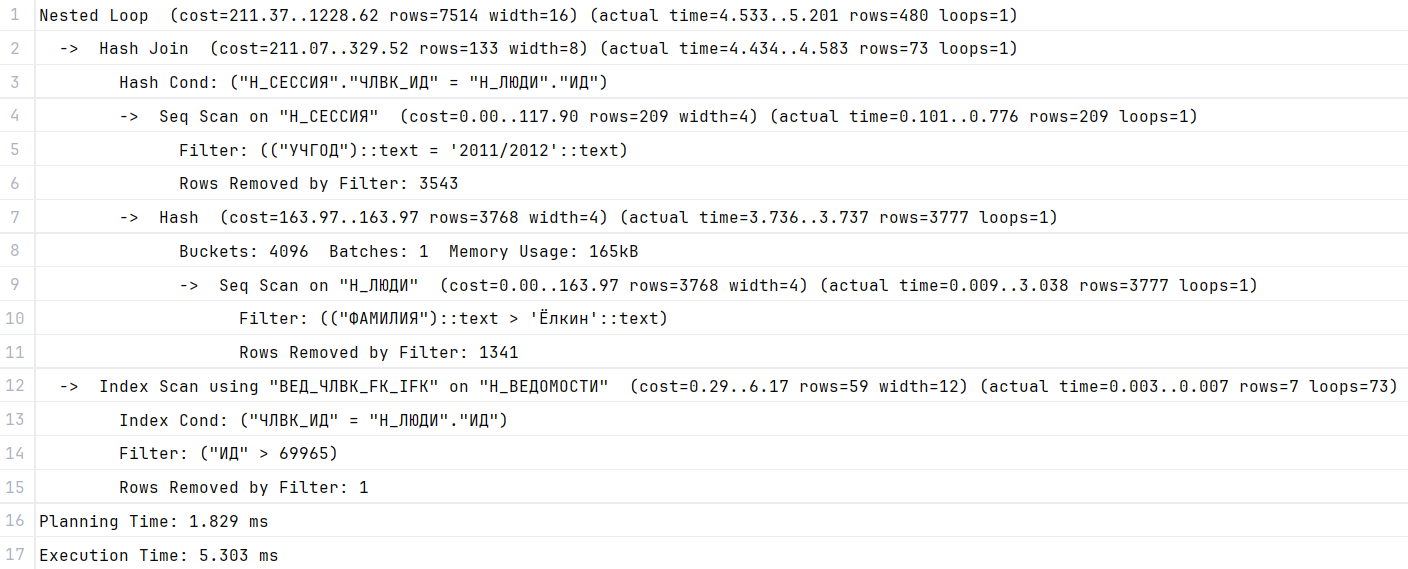
## План



Этот план можно улучшить, возвращая из нижних уровней только необходимые атрибуты, а не всю таблицу.



## Вывод EXPLAIN



Nested Loop (внешний цикл)

│

├── Hash Join (первая часть)

│ │

│ ├── Seq Scan по Н\_СЕССИЯ (с фильтром по учебному году)

│ │

│ └── Hash (построение хеш-таблицы)

│ │

│ └── Seq Scan по Н\_ЛЮДИ (с фильтром по фамилии)

│

└── Index Scan по Н\_ВЕДОМОСТИ (используя индекс ВЕД\_ЧЛВК\_FK\_IFK)

│

└── Фильтр по ИД > 69965

1. Внешний Nested Loop (cost=211.37..1228.62)

sql

Nested Loop (cost=211.37..1228.62 rows=7514 width=16) (actual time=4.533..5.201 rows=480 loops=1)

Тип: Вложенный цикл (основная операция соединения)

Оценка стоимости: от 211.37 до 1228.62

Ожидаемое количество строк: 7514

Фактическое время: 4.533..5.201 мс

Фактическое количество строк: 480

Количество итераций: 1

2. Hash Join внутри Nested Loop

sql

-> Hash Join (cost=211.07..329.52 rows=133 width=8) (actual time=4.434..4.583 rows=73 loops=1)

Тип: Хеш-соединение (первая часть вложенного цикла)

Условие соединения: Н\_СЕССИЯ.ЧЛВК\_ИД = Н\_ЛЮДИ.ИД

Оценка стоимости: 211.07..329.52

Ожидаемое количество строк: 133

Фактическое время: 4.434..4.583 мс

Фактическое количество строк: 73

3. Seq Scan по Н\_СЕССИЯ

sql

-> Seq Scan on "Н\_СЕССИЯ" (cost=0.00..117.90 rows=209 width=4) (actual time=0.101..0.776 rows=209 loops=1)

Filter: (("УЧГОД")::text = '2011/2012'::text)

Rows Removed by Filter: 3543

Тип: Полное сканирование таблицы

Фильтр: Учебный год = '2011/2012'

Ожидаемое количество строк: 209

Фактическое количество строк: 209

Отброшено строк: 3543 (не соответствуют условию)

4. Hash-таблица для Н\_ЛЮДИ

sql

-> Hash (cost=163.97..163.97 rows=3768 width=4) (actual time=3.736..3.737 rows=3777 loops=1)

Buckets: 4096 Batches: 1 Memory Usage: 165kB

-> Seq Scan on "Н\_ЛЮДИ" (cost=0.00..163.97 rows=3768 width=4) (actual time=0.009..3.038 rows=3777 loops=1)

Filter: (("ФАМИЛИЯ")::text > 'Ёлкин'::text)

Rows Removed by Filter: 1341

Размер хеш-таблицы: 4096 корзин, 1 партия, 165kB памяти

Полное сканирование Н\_ЛЮДИ:

Фильтр: Фамилия > 'Ёлкин'

Ожидаемое количество строк: 3768

Фактическое количество строк: 3777

Отброшено строк: 1341

5. Index Scan по Н\_ВЕДОМОСТИ

sql

-> Index Scan using "ВЕД\_ЧЛВК\_FK\_IFK" on "Н\_ВЕДОМОСТИ" (cost=0.29..6.17 rows=59 width=12) (actual time=0.003..0.007 rows=7 loops=73)

Index Cond: ("ЧЛВК\_ИД" = "Н\_ЛЮДИ"."ИД")

Filter: ("ИД" > 69965)

Rows Removed by Filter: 1

Используемый индекс: ВЕД\_ЧЛВК\_FK\_IFK

Условие индекса: ЧЛВК\_ИД = Н\_ЛЮДИ.ИД

Дополнительный фильтр: ИД > 69965

Количество итераций: 73 (по количеству строк из Hash Join)

Среднее время на итерацию: 0.003..0.007 мс

Среднее количество строк на итерацию: 7

Отброшено строк на итерацию: 1

# Вывод

В ходе работы я узнал, как выполняются запросы внутри субд. Изучил и применил на практике индексы. Узнал, как можно оптимизировать запросы.

# Ответы на вопросы

## **Типы соединения**

### Nested Loop

Описание: Вложенный цикл - базовый метод соединения.

Как работает:

Для каждой строки внешней таблицы:

Для каждой строки внутренней таблицы:

Если условие соединения выполняется -> добавить в результат

Преимущества:

Эффективен для маленьких таблиц

Не требует сортировки или хеширования

Хорош, когда есть индекс для внутренней таблицы

Недостатки:

O(n\*m) сложность для больших таблиц

### Hash Join

Описание: Использует хеш-таблицу для соединения.

Как работает:

Строит хеш-таблицу для меньшей таблицы

Сканирует большую таблицу и ищет совпадения в хеш-таблице

Преимущества:

Эффективен для больших таблиц без индексов

Хорош для равенств (=)

Недостатки:

Требует памяти для хеш-таблицы

Не подходит для неравенств (>, <)

### Merge Join

Описание: Использует сортированные данные для соединения.

Как работает:

Сортирует обе таблицы по ключу соединения

Проходит по обеим таблицам параллельно, находя совпадения

Преимущества:

Эффективен для больших отсортированных данных

Подходит для неравенств (>, <, BETWEEN)

Недостатки:

Требует предварительной сортировки (если данные не отсортированы)

Использует больше памяти

### Parallel Hash Join (Параллельное хеш-соединение)

Как работает:

Рабочие процессы (workers) совместно строят общую хеш-таблицу из одной из таблиц

Затем параллельно сканируют вторую таблицу и ищут совпадения в хеш-таблице

Лидер-процесс собирает результаты от всех рабочих процессов

Преимущества:

Значительное ускорение для больших таблиц (в 4-8 раз на многоядерных системах)

Эффективное использование ресурсов сервера

Хорошо масштабируется с увеличением числа ядер

Недостатки:

Требует больше памяти (каждый worker использует свою копию хеш-таблицы)

Накладные расходы на координацию между процессами

Не эффективен для маленьких таблиц

Когда используется:

Для соединения больших таблиц (обычно > 100,000 строк)

Когда max\_parallel\_workers\_per\_gather > 0 и work\_mem достаточно

Для равенств в условиях JOIN (=)

### Parallel Nested Loop (Параллельные вложенные циклы)

Как работает:

Внешняя таблица разделяется между рабочими процессами

Каждый worker выполняет свой фрагмент Nested Loop

Результаты объединяются лидер-процессом

Преимущества:

Ускорение для CPU-интенсивных операций

Хорошо работает, когда внутренний цикл использует индекс

Эффективен для маленьких/средних таблиц с хорошей селективностью

Недостатки:

Ограниченная масштабируемость (обычно 2-4 workers)

Не эффективен, если внутренний цикл требует полного сканирования

Когда используется:

Для соединения таблиц, где внешняя таблица среднего размера

Когда есть индексы для внутреннего цикла

Для сложных условий, которые нельзя хешировать

### Bitmap Heap Scan + Bitmap Index Scan

Как работает:

Bitmap Index Scan создает битовую карту совпадений по индексу

Bitmap Heap Scan использует эту карту для эффективного чтения нужных страниц таблицы

Битовая карта (bitmap) в PostgreSQL — это компактная структура данных, которая представляет соответствие между индексными записями и физическим расположением данных в таблице с помощью битов (0 и 1).

Применяются дополнительные фильтры (если есть)

Преимущества:

Эффективно комбинирует несколько индексов

Минимизирует случайные чтения с диска

Хорош для сложных условий с AND/OR

Недостатки:

Требует построения битовой карты в памяти

Не всегда оптимален для простых условий

Когда используется:

Для сложных условий фильтрации с несколькими индексами

Когда нужно объединить результаты нескольких индексов

Для частичного совпадения по индексу с дополнительными фильтрами

### Semi Join и Anti Join

Как работает:

Semi Join: Возвращает строки из первой таблицы, где есть хотя бы одно совпадение во второй (без дубликатов)

Anti Join: Возвращает строки из первой таблицы, где нет совпадений во второй

Преимущества:

Оптимизированная работа для EXISTS/NOT EXISTS

Не обрабатывает все совпадения, только первое/отсутствие

Может использовать специальные алгоритмы (Hash/Merge/Nested)

Недостатки:

Ограниченная применимость (только для подзапросов)

Не всегда выбирается оптимальный план

Когда используется:

Для подзапросов с EXISTS/NOT EXISTS

Для IN/NOT IN с большими подзапросами

Когда нужно проверить наличие/отсутствие без полного соединения

### Materialized Join (Материализованное соединение)

Как работает:

Одна из таблиц полностью материализуется во временной структуре

Затем выполняется соединение с материализованной версией

Может комбинироваться с другими методами (Hash, Nested Loop)

Преимущества:

Уменьшает повторные чтения одной таблицы

Полезно для сложных подзапросов

Может ускорить повторяющиеся соединения

Недостатки:

Требует дополнительной памяти

Накладные расходы на материализацию

Когда используется:

Для сложных подзапросов в JOIN

Когда одна таблица используется несколько раз

Для оптимизации общих табличных выражений (WITH)

### LATERAL Join (Латеральное соединение)

Как работает:

Позволяет подзапросу справа ссылаться на столбцы из таблиц слева

Выполняется для каждой строки левой таблицы

Может использовать разные методы соединения внутри

Преимущества:

Гибкость в сложных запросах

Возможность параметризации подзапросов

Поддержка LIMIT внутри соединения

Недостатки:

Может быть неэффективным без правильных индексов

Сложнее для оптимизатора

Когда используется:

Для TOP-N запросов в соединениях

Когда нужно передавать параметры в подзапрос

Для сложных иерархических запросов

### Index Only Join (Только индексное соединение)

Как работает:

Все необходимые данные берутся только из индексов

Полностью избегает чтения табличных данных (heap)

Требует покрывающих индексов (INCLUDE в PostgreSQL 11+)

Преимущества:

Максимальная скорость (только индексные чтения)

Минимальные блокировки

Эффективное использование кэша

Недостатки:

Требует специальных индексов

Ограниченный набор поддерживаемых операций

Когда используется:

Для запросов, которые можно полностью покрыть индексами

В высоконагруженных OLTP-системах

Когда важна скорость чтения

### TID Scan Join (Соединение по идентификаторам кортежей)

Как работает:

Использует физические адреса строк (TID - Tuple ID)

Прямой доступ к данным по физическому расположению

Редко используется в явном виде

Преимущества:

Максимально быстрое обращение к конкретным строкам

Полезно для внутренних операций

Недостатки:

Очень специфическое применение

Хрупкость (TID меняется при VACUUM FULL)

Когда используется:

Для внутренних системных запросов

При работе с курсорами

В некоторых типах блокировок

## **Методы доступа**

### Index Only Scan

Как работает:

СУБД читает данные только из индекса, не обращаясь к таблице (heap)

Возможен когда в SELECT только столбцы, входящие в индекс

Преимущества:

Максимальная скорость (нет обращений к таблице)

Минимальная нагрузка на I/O

### Index Scan

Описание: использует индекс для поиска, но затем обращается к таблице за дополнительными данными.

Как работает:

Находит записи в индексе

По указателям из индекса получает полные строки из таблицы

Преимущества:

Быстрее, чем полное сканирование таблицы

Позволяет эффективно находить конкретные строки

Недостатки:

Требует дополнительных обращений к таблице (heap fetches)

### Bitmap Index Scan + Bitmap Heap Scan

Описание: Двухэтапный процесс для эффективного поиска по нескольким условиям.

Как работает:

Bitmap Index Scan: создает битовую карту совпадений

Bitmap Heap Scan: использует битовую карту для доступа к таблице

Преимущества:

Эффективен для комбинации условий

Минимизирует случайные чтения из таблицы

### Sequential Scan (Seq Scan)

Описание: Полное сканирование таблицы.

Когда используется:

Нет подходящего индекса

Запрашивается большая часть таблицы

Для очень маленьких таблиц

Недостатки:

Медленно для больших таблиц

Высокая нагрузка на I/O

### Tid Scan (Сканирование по физическому адресу строки)

Описание:

Прямой доступ к строке по ее физическому идентификатору (TID — Tuple ID).

Преимущества:

✔ Максимально быстрое чтение конкретной строки.

Недостатки:

✖ Крайне редко используется в обычных запросах.

✖ TID меняется после VACUUM FULL.

Когда используется:

Внутренние операции PostgreSQL (например, обновление строк).

Запросы с ctid (физический адрес строки).

Пример:

sql

EXPLAIN SELECT \* FROM users WHERE ctid = '(123, 5)';

### Parallel Scan (Параллельное сканирование)

Описание:

Распределяет сканирование таблицы или индекса между несколькими worker-процессами.

Преимущества:

✔ Ускорение для больших таблиц (в 2-8 раз на многоядерных CPU).

Недостатки:

✖ Накладные расходы на координацию процессов.

✖ Требует настройки (max\_parallel\_workers\_per\_gather).

Когда используется:

Большие Seq Scan/Index Scan (обычно >100K строк).

Аналитические запросы (COUNT, SUM, GROUP BY).

## **Типы индексов**

### 1. B-Tree (Балансированное дерево)

Как работает:

Хранит данные в сбалансированной древовидной структуре

Поддерживает сортировку значений

Обеспечивает поиск за O(log n)

Для чего подходит:

Стандартные сравнения: =, >, <, >=, <=, BETWEEN

LIKE 'prefix%' (но не LIKE '%suffix')

IS NULL, IS NOT NULL

Композитные индексы (многоколоночные)

Особенности:

sql

CREATE INDEX idx\_name ON table\_name (column1, column2);

Поддерживает уникальность (UNIQUE CONSTRAINT)

Оптимален для OLTP-нагрузки

### 2. Hash (Хеш-индекс)

Как работает:

Создает хеш-таблицу значений

Быстрый поиск точного совпадения (O(1) в идеальном случае)

Для чего подходит:

Только операторы равенства (=)

Точечные запросы без диапазонов

Особенности:

sql

CREATE INDEX idx\_name ON table\_name USING HASH (column);

Не поддерживает сортировку

Не поддерживает уникальность до PostgreSQL 10

Требует перестройки при больших изменениях данных

### 3. GiST (Generalized Search Tree)

Как работает:

Обобщенная структура для различных типов данных

Позволяет реализовать собственные стратегии поиска

Для чего подходит:

Геометрические данные (близость, пересечение)

Полнотекстовый поиск

Диапазоны (range types)

Иерархические данные (ltree)

Пример:

sql

CREATE INDEX idx\_gist ON table USING GiST (geom\_column);

### 4. SP-GiST (Space-Partitioned GiST)

Как работает:

Разбивает пространство на непересекающиеся области

Альтернатива GiST для определенных типов данных

Для чего подходит:

IP-адреса (inet, cidr)

Геометрические данные (особые виды разбиения)

Текст (различные стратегии разбиения)

Пример:

sql

CREATE INDEX idx\_spgist ON table USING SPGiST (ip\_column);

### 5. GIN (Generalized Inverted Index)

Как работает:

Инвертированный индекс (хранит mapping: значение → строки)

Оптимизирован для составных значений

Для чего подходит:

JSON/JSONB данные

Полнотекстовый поиск

Массивы (contains, overlap)

hStore (ключ-значение)

Пример:

sql

CREATE INDEX idx\_gin ON table USING GIN (jsonb\_column);

### 6. BRIN (Block Range INdex)

Как работает:

Хранит метаданные о диапазонах физических блоков

Крайне компактен

Для чего подходит:

Очень большие таблицы с коррелированной физической и логической сортировкой

Временные ряды (timestamp, последовательные ID)

Данные, которые редко изменяются

Пример:

sql

CREATE INDEX idx\_brin ON table USING BRIN (timestamp\_column);

### 7. Bloom (Фильтр Блума)

Как работает:

Вероятностная структура данных

Быстро проверяет "возможное наличие" элемента

Для чего подходит:

Многоколоночные запросы с большим количеством комбинаций

Когда важнее скорость, чем точность

Особенности:

sql

CREATE EXTENSION bloom;

CREATE INDEX idx\_bloom ON table USING bloom (col1, col2, col3);

Могут быть ложные срабатывания

Компактнее, чем B-Tree для многих столбцов

### 8. RUM (расширение GIN)

Как работает:

Улучшенная версия GIN

Хранит дополнительную информацию о позициях токенов

Для чего подходит:

Фразовый поиск

Ранжирование в полнотекстовом поиске

Более сложные операции с массивами

Пример:

sql

CREATE EXTENSION rum;

CREATE INDEX idx\_rum ON table USING rum (text\_column rum\_tsvector\_ops);

### 9. Partial (Частичный индекс)

Как работает:

Индексирует только подмножество строк

Содержит условие WHERE

Для чего подходит:

Когда интересна только часть данных

Уменьшает размер индекса

Ускоряет запросы к определенным подмножествам

Пример:

sql

CREATE INDEX idx\_partial ON table (column) WHERE status = 'active';

### 10. Covering (Покрывающий индекс)

Как работает:

Включает в индекс дополнительные столбцы

Позволяет Index Only Scan

Для чего подходит:

Частые запросы, выбирающие определенные столбцы

Избегание обращений к таблице

Пример:

sql

CREATE INDEX idx\_covering ON table (id) INCLUDE (name, email);

### 11. Functional (Функциональный индекс)

Как работает:

Индексирует результат функции

Вычисляемое значение хранится в индексе

Для чего подходит:

Запросы с функциями в условиях

Нормализация данных перед индексацией

Пример:

sql

CREATE INDEX idx\_func ON table (lower(email));

## **Прочие вопросы**

### Heap Blocks

Ключевые характеристики:

Размер по умолчанию: 8KB (8192 байта)

Содержит:

Заголовок блока (24 байта)

Массив указателей на строки (ItemId array)

Фактические данные строк (кортежи)

Свободное пространство

Как PostgreSQL использует Heap Blocks?

1. Структура блока

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

| Заголовок | (24 байта)

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

|ItemId array | (массив указателей на строки)

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

|Свободное место|

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

| Кортежи | (фактические данные строк)

### Buckets (Корзины/Бакеты)

Buckets - это метод организации данных путем их распределения в "корзины" на основе хеш-функции или диапазонов значений.

Применение Buckets:

Хеш-партиционирование:

sql

CREATE TABLE sales (

id serial,

sale\_date date,

amount numeric

) PARTITION BY HASH (id);

Ускорение JOIN операций:

Данные с одинаковым хешем попадают в одну корзину

Соединение происходит только между соответствующими корзинами

Группировка в аналитических запросах:

sql

SELECT width\_bucket(amount, 0, 1000, 10) AS bucket, COUNT(\*)

FROM sales

GROUP BY bucket;

Преимущества Buckets:

Уменьшение объема обрабатываемых данных

Параллельная обработка разных корзин

Равномерное распределение данных

### Batches (Пакетная обработка)

Что такое Batches?

Batches - это группировка операций в пакеты для более эффективной обработки.

Применение Batches:

Пакетные вставки:

sql

INSERT INTO table VALUES (1), (2), (3); -- вместо 3 отдельных INSERT

Пакетные обновления:

sql

UPDATE table SET col = new\_val WHERE id IN (1, 2, 3);

Чтение данных порциями:

python

# Пример на Python с psycopg2

cursor.execute("SELECT \* FROM large\_table")

while True:

batch = cursor.fetchmany(1000)

if not batch:

break

process\_batch(batch)

Преимущества Batches:

Снижение накладных расходов на коммуникацию

Лучшее использование кэша CPU

Уменьшение количества транзакций

### Когда индексы неэффективны или невыгодны?

3.1. Частые операции записи

Проблема:

Каждое INSERT/UPDATE/DELETE требует обновления индексов

При высокой частоте изменений индексы становятся "узким местом"

Решение:

Уменьшить количество индексов для часто изменяемых таблиц

Использовать индексы только для часто читаемых колонок

3.2. Большие таблицы с низкой селективностью

Проблема:

Если условие выбирает >15-20% строк таблицы

Seq Scan может быть быстрее Index Scan + Heap Fetches

Пример:

sql

-- Неэффективно для индекса, если статусов немного:

SELECT \* FROM orders WHERE status IN ('new', 'processing', 'completed');

3.3. Очень маленькие таблицы

Проблема:

Накладные расходы на использование индекса могут превысить выгоду

PostgreSQL часто выбирает Seq Scan для таблиц < 10-20 страниц

### Что такое Recheck Cond?

Recheck Cond (повторная проверка условия) — это этап выполнения запроса в PostgreSQL, когда система после первоначального отбора строк по индексу должна дополнительно проверить условие непосредственно на данных таблицы (heap).

Как это работает?

Первичная проверка: PostgreSQL использует индекс (обычно Bitmap Index Scan) для быстрого нахождения потенциально подходящих строк.

Построение битовой карты: Создается карта страниц (blocks), где могут находиться нужные строки.

Recheck Cond: При чтении данных из этих страниц PostgreSQL перепроверяет условие, так как:

Индекс может дать ложноположительные срабатывания

Условие может содержать часть, не покрытую индексом

## **Режимы кэширования (Cache Modes)**

### 1.1. Write-Through (Сквозная запись)

Как работает:

Данные записываются одновременно в кэш и основное хранилище

Обеспечивает согласованность данных

Преимущества:

Высокая надежность данных

Простота реализации

Недостатки:

Более медленные операции записи

Увеличенная нагрузка на основное хранилище

Использование:

Системы, где критична сохранность данных

Финансовые транзакции

1.2. Write-Back (Отложенная запись)

Как работает:

Данные сначала записываются в кэш

Запись в основное хранилище происходит позже (асинхронно)

Преимущества:

Быстрые операции записи

Снижение нагрузки на основное хранилище

Недостатки:

Риск потери данных при сбое

Сложнее обеспечить согласованность

Использование:

Высоконагруженные системы

Когда производительность важнее мгновенной согласованности

1.3. Write-Around (Обход кэша)

Как работает:

Данные записываются напрямую в хранилище, минуя кэш

Кэшируются только при чтении

Преимущества:

Не засоряет кэш одноразовыми записями

Эффективен для данных с однократной записью и редким чтением

Недостатки:

Первое чтение будет медленным

Использование:

Логирование

Архивные данные

1.4. Read-Through (Чтение через кэш)

Как работает:

При промахе кэша система автоматически загружает данные из хранилища

Прозрачно для приложения

Преимущества:

Упрощает логику приложения

Автоматическое заполнение кэша

Недостатки:

Первое обращение к данным может быть медленным

1.5. Cache-Aside (Lazy Loading)

Как работает:

Приложение явно управляет кэшем

При промахе приложение само загружает данные в кэш

Преимущества:

Гибкость управления

Можно реализовать сложную логику кэширования

Недостатки:

Более сложная реализация

Возможность кэширования устаревших данных

## Алгоритмы вытеснения данных их кэша (Eviction Policies)

### 2.1. LRU (Least Recently Used)

Принцип работы:

Вытесняет редко используемые элементы

Поддерживает очередь по времени последнего доступа

Плюсы:

Эффективен для большинства шаблонов доступа

Простая реализация

Минусы:

Не учитывает частоту использования

Уязвим к сканированию (scan resistance)

### 2.2. LFU (Least Frequently Used)

Принцип работы:

Вытесняет наименее часто используемые элементы

Подсчитывает количество обращений к каждому элементу

Плюсы:

Хорош для часто используемых "горячих" данных

Минусы:

Может задерживать устаревшие данные

Дополнительные накладные расходы

### 2.3. FIFO (First-In, First-Out)

Принцип работы:

Вытесняет самые старые элементы (по времени добавления)

Плюсы:

Очень простая реализация

Низкие накладные расходы

Минусы:

Низкая эффективность для большинства сценариев

### 2.4. LIRS (Low Inter-reference Recency Set)

Принцип работы:

Комбинирует LRU с учетом "холодных" элементов

Более устойчив к сканированию, чем LRU

Плюсы:

Лучшая производительность для некоторых workload'ов

Хорошая scan resistance

Минусы:

Более сложная реализация

### 2.5. ARC (Adaptive Replacement Cache)

Принцип работы:

Адаптивно балансирует между LRU и LFU

Поддерживает два списка: недавние и частые элементы

Плюсы:

Автоматическая адаптация к шаблонам доступа

Высокая эффективность

Минусы:

Высокие накладные расходы

Сложная реализация

## Переполнение кэша (Cache Overflows)

3.1. Причины переполнения:

Недостаточный размер кэша

Неоптимальная политика вытеснения

Изменение паттернов доступа к данным

Утечки памяти в реализации

3.2. Методы обработки переполнения:

Динамическое масштабирование:

Автоматическое увеличение размера кэша

Используется в распределенных системах

Сегментирование кэша:

Разделение кэша на независимые части

Разные политики для разных сегментов

Гибридные стратегии:

Комбинация нескольких алгоритмов вытеснения

Адаптивное переключение между политиками

Интеллектуальное предварительное вытеснение:

Прогнозирование будущих обращений

Заблаговременное освобождение места