циан | PiterPy

2025

Оптимизация PostgreSQL-запросов:

все, что нужно знать на практике

Александр Киверин,
Staff Engineer, Head of Python

Обо мне

Staff Engineer /Head of Python

- 5 лет в Циан
- Успел поработать со всеми основными компонентами (поиск, карточка, форма подачи, личный кабинет)
- В гильдии улучшаю процессы
- И автоматизирую все, что только можно

циан

1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы

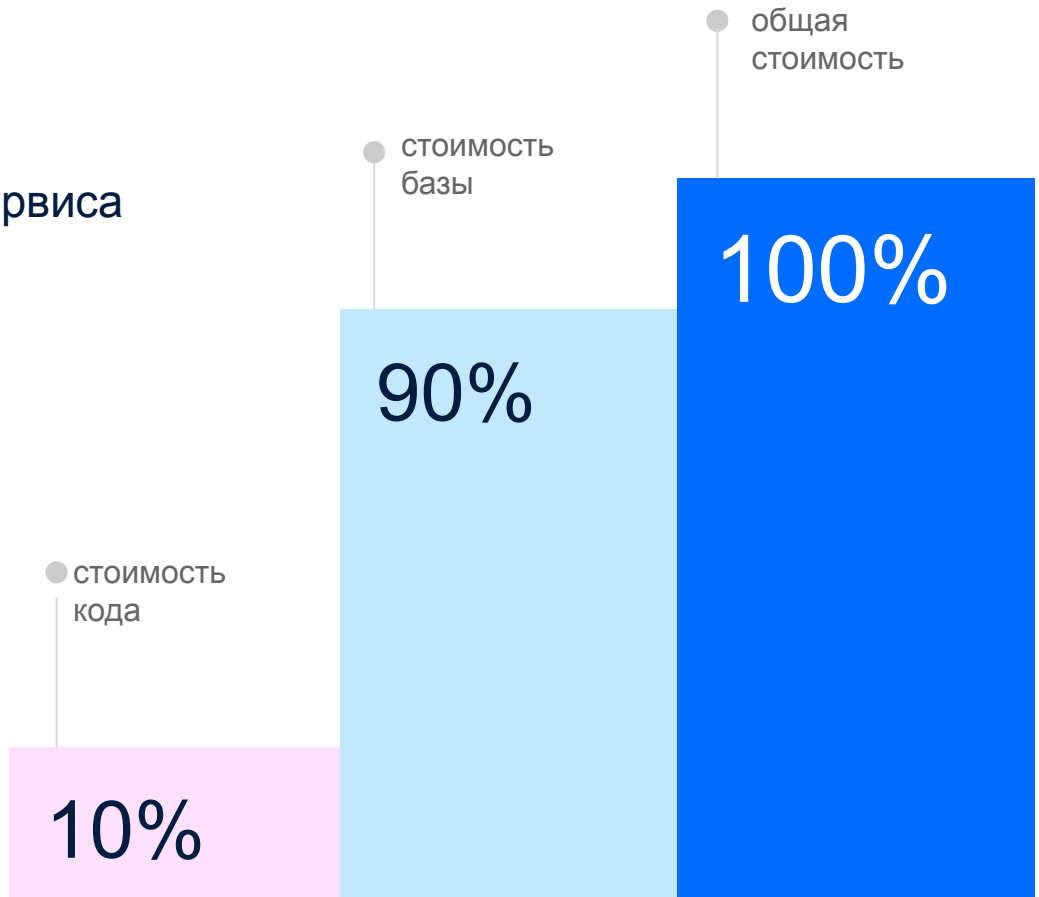
1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы

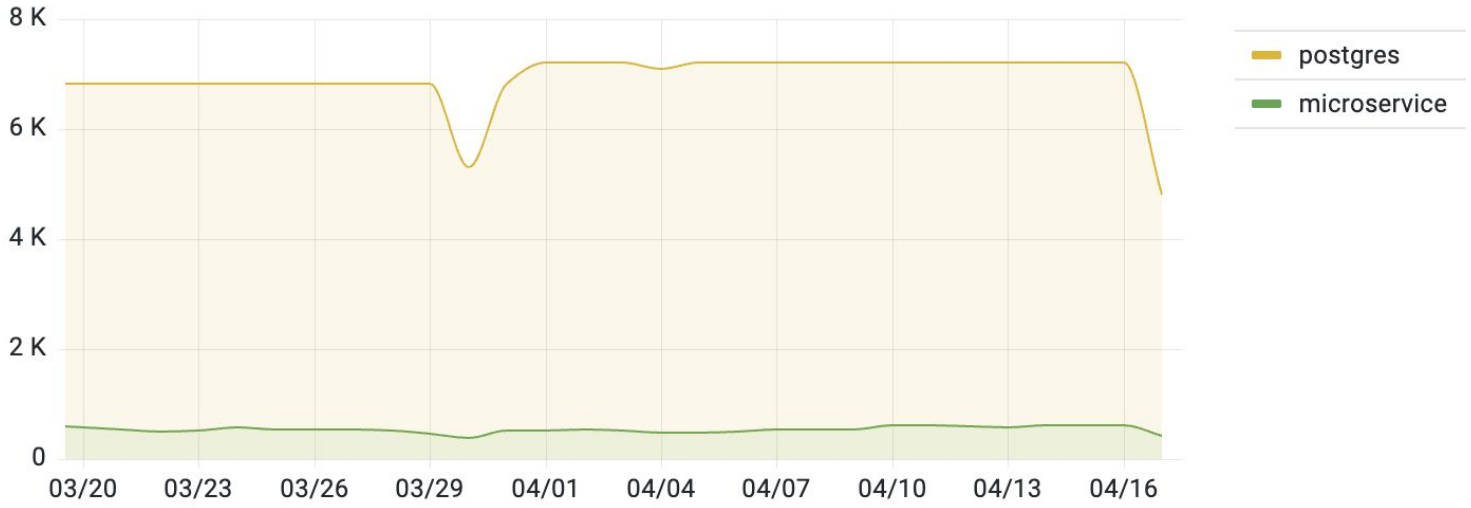
Нужно ли нам вообще что-то оптимизировать?

Вопрос стоимости

Сколько стоит база в облаке?

На порядок дороже экземпляров сервиса
(даже если база не managed виде)





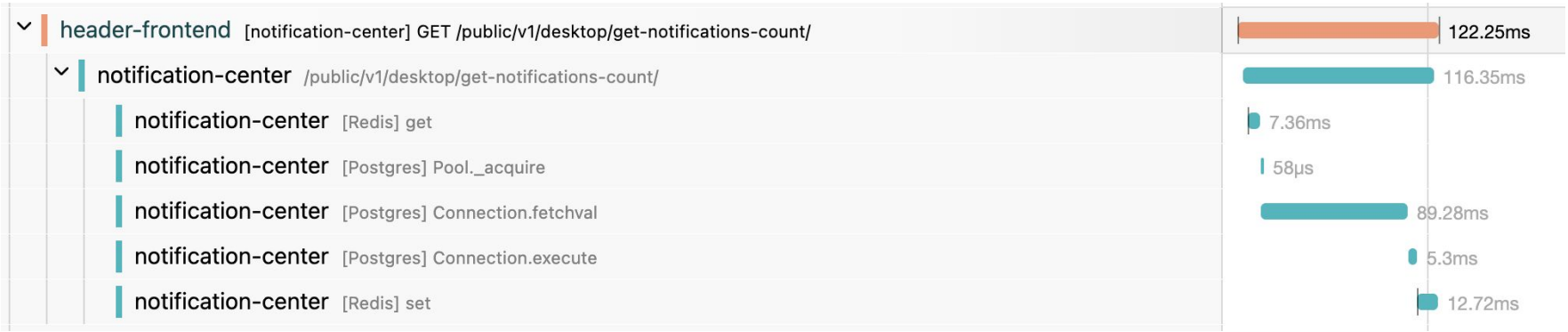
Пример реальной стоимости в облаке за каждый день

Вопрос скорости

циан

Как походы в базу влияют на время ответа?

- В большинстве web приложений на походы в базу приходится основная доля времени ответа
- Медленнее ответы - хуже индексируются страницы



Вопрос изменения

циан

Сколько стоит поменять формат хранения?

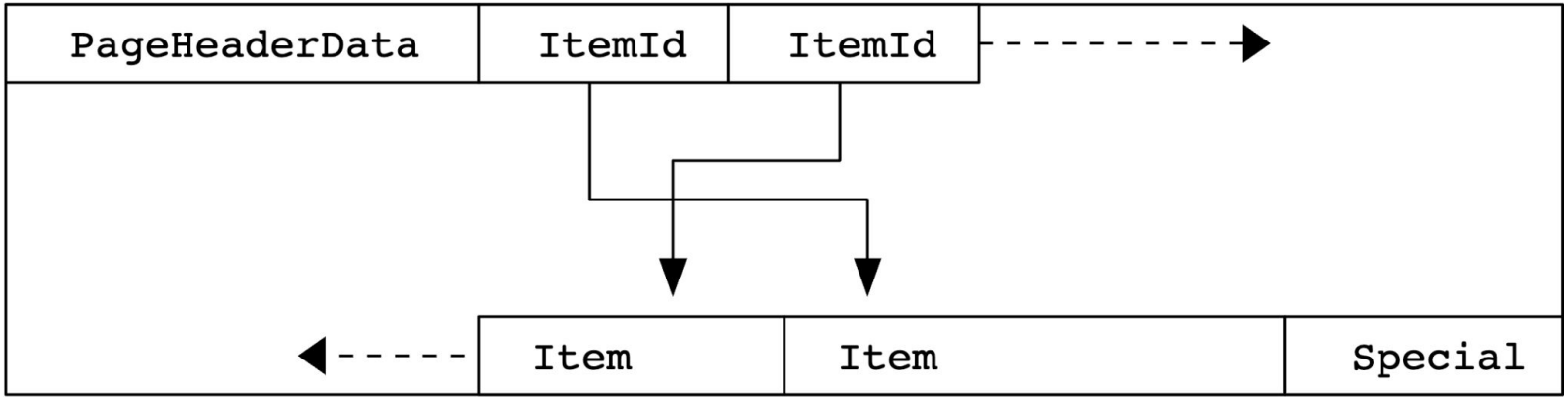
Слой данных сложнее всего менять - вопрос переливки данных в новое хранилище без просадки uptime

циан

1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы
6. Продуктовые подходы

Устройство хранения

- У нас есть таблицы и строки (они же **tuples** или **кортежи**)
- Строки хранятся в страницах (они же **блоки** по **8 КБ**),
- Страницы в куче (**heap**) на диске



TOAST хранение

- Храним отдельно или сжимаем строки:
  - Которые не помещаются в страницу
  - Превышают **TOAST_TUPLE_THRESHOLD** (2 КБ)
- **TOAST** добавляет overhead на чтение, запись и очистку данных
  - Например, нужно быть осторожным с хранением JSON

Multiversion Concurrency Control (MVCC)

циан

- Движок оптимистичных блокировок
    - Блокировать каждую изменяемую строку от чтения - дорого
- Нужно для согласованного видения данных транзакциями, чтобы не мешали друг другу
- У транзакции есть идентификатор **xid**
- У каждой строки есть идентификатор **xmin** (с какой транзакции видна) и **xmax** (с какой уже нет)
- **Read Uncommited** в **PostgreSQL** нет

Side Effects от MVCC

- Создается копия строки при каждом изменении
  - Переиспользуется после **VACUUM**
  - Очистятся при **VACUUM FULL**
- Как с этим жить?
  - Разбиваем широкие строки по частоте обновления
  - Например, не храним модель объявления вместе с количеством просмотров

shared memory buffers

- Least Recently Used кэш в RAM
- Живет отдельно от кэша ОС
- Все страницы проходят через него
- Если страница есть, то не пойдем на диск (ускорение на порядок)

Выводы

- Меньше обновлений
  - Меньше копий строк хранится на диске
- Меньше строки
  - Больше строк влезет в кэш
  - Медленнее рост базы

циан

1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы

Виды запросов

Транзакционные или короткие или OLTP

Для результата нужно небольшое количество строк

- Найти пользователя по идентификатору
- Выгрузить конфиг при старте приложения

Аналитические или длинные или OLAP

Для результата нужно прочитать большое количество строк

- Подсчет общего числа записей
- Поиск максимального значения

Взято из PostgreSQL Query Optimization: The Ultimate Guide to Building Efficient Queries

Как строится?

- На входе есть SQL декларативный запрос
- Есть N вариантов выполнения каждой операции внутри:
    - Как считать нужные строки
    - Как соединить таблицы
- Каждый узел оценивается:
    - Стоимость чтения страницы с диска
    - Стоимость обработки строки
- Выбирается самое оптимальное дерево
- Ориентируется на статистику таблицы (например, распределение значений в столбце)

Как управлять?

- Вам это не нужно
- PostgreSQL принимает решает “самостоятельно”
- Проверьте точность статистики - представление **pg_stats**
- Для экспериментов - расширение **pg_hint_plan**

```
 /*+
    HashJoin(a b)
    SeqScan(a)
  */
 EXPLAIN SELECT *
    FROM pgbench_branches b
    JOIN pgbench_accounts a ON b.bid = a.bid
    ORDER BY a.aid;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------
 Sort  (cost=31465.84..31715.84 rows=100000 width=197)
   Sort Key: a.aid
   ->  Hash Join  (cost=1.02..4016.02 rows=100000 width=197)
         Hash Cond: (a.bid = b.bid)
         ->  Seq Scan on pgbench_accounts a  (cost=0.00..2640.00 rows=100000 width=97)
         ->  Hash  (cost=1.01..1.01 rows=1 width=100)
               ->  Seq Scan on pgbench_branches b  (cost=0.00..1.01 rows=1 width=100)
(7 rows)
```

Как его получить?

- **EXPLAIN (ANALYZE, BUFFERS) запрос**
- EXPLAIN - построит план
- ANALYZE - выполнит запрос и сравнит с планом
- BUFFERS - покажет статистику по блокам (hit, read, dirtied, written)

Как его читать?

- Это дерево
- Начинаем читать изнутри
- Параметры:
  - cost
  - rows (выдал узел)
  - width (в байтах)
  - loops
  - actual time (**для 1 цикла,** с cost не сравниваем)

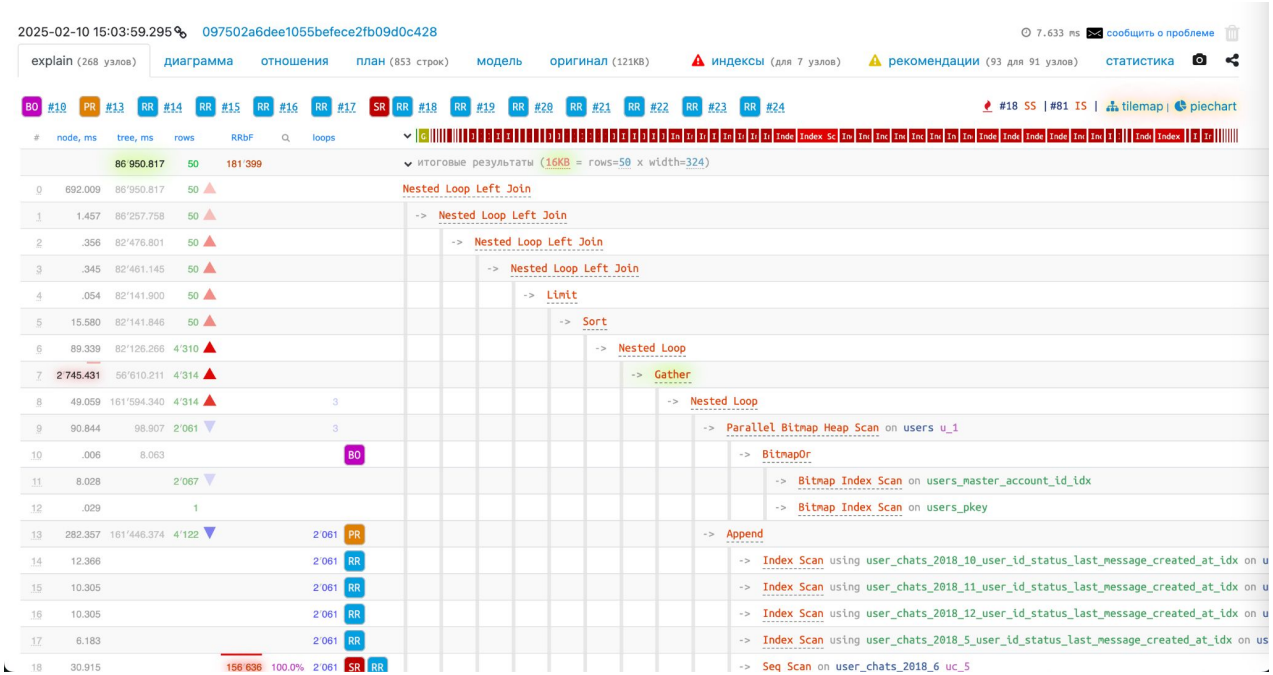
	QUERY PLAN text
1	GroupAggregate (cost=762870.42..762872.26 rows=1 width=24)
2	Group Key: f.flight_id
3	-> Sort (cost=762870.42..762871.03 rows=244 width=20)
4	Sort Key: f.flight_id
5	-> Hash Join (cost=366897.02..762860.74 rows=244 width=20)
6	Hash Cond: (p.booking_id = bl.booking_id)
7	-> Seq Scan on passenger p (cost=0.00..334784.93 rows=16313693 width=8)
8	-> Hash (cost=366896.70..366896.70 rows=26 width=20)
9	-> Hash Join (cost=9419.14..366896.70 rows=26 width=20)
10	Hash Cond: (bl.flight_id = f.flight_id)
11	-> Seq Scan on booking_leg bl (cost=0.00..310506.66 rows=17893566 width=8)
12	-> Hash (cost=9419.12..9419.12 rows=1 width=16)
13	-> Bitmap Heap Scan on flight f (cost=194.99..9419.12 rows=1 width=16)
14	Recheck Cond: (departure_airport = 'JFK'::bpchar)
15	Filter: ((actual_departure >= '2020-08-14 00:00:00-05'::timestamp with time zone) AND (actual_departure <= '2020-08-15 00:0...
16	-> Bitmap Index Scan on flight_departure_airport (cost=0.00..194.99 rows=10475 width=0)
17	Index Cond: (departure_airport = 'JFK'::bpchar)

Виды сканирований

- Seq Scan - читаем подряд все страницы в **heap**
- Index Scan - читаем по 1 строке сначала индекс, затем страницу в **heap**
- Index Only Scan - читаем только индекс (есть нюанс)
- Bitmap Index Scan + Bitmap Heap Scan - нужен когда хотим прочитать много строк из индекса или объединить несколько индексов

Как читать удобно?

- explain.tensor.ru



Как не строить вручную?

- Модуль **auto_explain**
- log_min_duration - минимальная длительность для замера
- log_analyze - выполнять ли ANALYZE
- Overhead - https://www.pgmustard.com/blog/auto-explain-overhead-with-timing

Test	Mean latency	Overhead vs baseline
Baseline	1.585 ms	–
Min duration 0	1.994 ms	25.8%
Min duration 10	1.597 ms	0.8%
Analyze on, Timing off, Min 10	1.590 ms	0.3%
Analyze on, Timing on, Min 10	1.620 ms	2.2%
Everything on, Min 10	1.613 ms	1.8%

циан

1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы

Какая основная цель?

Читать меньше страниц с диска

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months

Индексы

Пройдемся по основам

- B-tree создается по умолчанию
- Это не бинарное дерево, но сбалансированное и оптимизированное под последовательное чтение (специально под HDD)
- Индекс (a,b,c) будет использоваться для запросов по:
    - a=?
    - a=? AND b=?
    - a=? AND b=? AND c=?
- Side Effect от **MVCC** - в индексе указатель на каждую версию строки

Почему не используется индекс?

- В запросе неправильный порядок столбцов
  - Индекс на a и b, в запросе условия по b и a
- Индекс создавался через **CONCURRENTLY** и процесс завершился с ошибкой

```
SELECT * FROM pg_class, pg_index WHERE pg_index.indisvalid =
false AND pg_index.indexrelid = pg_class.oid;
```

- Проверить через **SET enable_seqscan=off** для своей сессии
  - Планировщик будет **по возможности** избегать **Seq Scan**
  - Возможно, так действительно быстрее - маленькая таблица
- Не хватает статистики по таблице:
  - ANALYZE + посмотреть **default_statistics_target**

Частичный индекс

- Создаем с помощью **WHERE** в **CREATE INDEX**
- Например, исключаем удаленные записи по флагу
  - CREATE INDEX idx_active_offers ON offers (offer_id) WHERE not deleted;
- Зачем?
  - Сокращаем размер индекса
  - И расходы на его обновление

Index Only Scan

- Ускоряет чтение, не идем в **heap**
- Можно хранить дополнительные данные в индексе
  - CREATE INDEX idx_offers_title ON offers (offer_id) INCLUDE (title);
- В чем подвох?
  - Увеличивает размер индекса - замедляется чтение
  - Часть строк может быть не видна текущий транзакции из-за **MVCC**
    - Нужна перепроверка в **heap**
    - Оптимизировано с помощью **Visibility Map**
    - Точность зависит от количества обновлений и когда был **ANALYZE**

Партиции

Зачем нужны?

- Разбиваем большую таблицу на более мелкие
- Например, по дате или по хэшу
- Ищем по каждой партиции отдельно

Какая опасность?

- Читать по множеству партиций хуже чем по одной таблице
- Возможен перекос в размере между партициями
- Нельзя перемещать данные между партициями при **UPDATE**
- По умолчанию нужно создавать вручную!

Default партиции

- Партиция для значений, которые не попадают ни в одну из существующих
- Поможет если забыли заранее создать партиции
- **Используйте осторожно:** нельзя будет создать новые партиции, пока в default есть подходящие для них значения

Как снизить риски?

- Заранее определяйте ваш паттерн чтения:
    - Если партиция по месяцам, то будет ли всегда известен точный диапазон?
- Следите за размерами партиций и распределением строк
    - Системные таблицы **pg_class** и **pg_inherits**
- Проверяйте на ревью использование ключа партиции в запросе. Ошибки при запросе без ключа не будет
- Для автоматического создания партиций - **pg_cron + pg_partman**

Динамическое исключение

- Можно отфильтровывать партиции в момент выполнения запроса
- Например, при **JOIN** таблицы с партиционированной
- Может помочь в случаях, когда заранее не знаем, какие партиции понадобятся

```
->  Index Scan using chats_2018_8_pkey on chats_2018_8 chats_7  (cost=0.14..5.59 rows=1 width=80) (never executed)
```

Помогаем планировщику

Избыточные условия

- Не можем явно указывать планировщику, но можем помогать ему
- Докидываем избыточные условия фильтрации
- Возможно помогут выбрать иной план запроса

Ранняя фильтрация

- Чем меньше строк нужно перекидывать между узлами плана тем лучше
- Следим за **rows** и **loops** в плане запроса
- Фильтруем и используем **SEMI** и **ANTI JOIN** (EXISTS)

Со стороны приложения

Используем pool соединений

- Зачем?
  - Есть overhead по времени на каждое подключение
  - Под каждое соединение выделяется >10 МБ
- Что делаем?
  - **asyncpg, sqlalchemy.pool** для локального, помогаем приложению
  - **pgbouncer, Odyssey** для глобального, помогаем базе

Контролируем долгие транзакции

- Зачем?
  - Мешают **VACUUM** - не переиспользовать строки, которые видны самой старой транзакции из-за **MVCC**
  - Держат блокировки на строках и таблицах
  - Помешают **ALTER TABLE** и всем после него (**lock queue**)
- Что делаем?
  - Смотрим на **xact_start** в **pg_stat_activity** - время старта последней транзакции
  - Выставить **idle_in_transaction_session_timeout** - останавливаем зависшие транзакции

Немного настроек

Настройка shared_buffers

- Размер shared memory буфера
  - По умолчанию - 128 МБ
  - Документация рекомендует поднимать сразу до 25% от памяти
- Зачем?
  - Больше памяти = больше hit rate страниц при чтении

Настройка random_page_cost

- Стоимость случайного чтения страниц в памяти
- По умолчанию 4.0 (**seq_page_cost** - 1.0)
- Насколько часто **PostgreSQL** будет выбирать **Seq Scan**, а не множество случайных походов через индекс
- Было актуально во времена HDD, если ваша база с SSD, то можно ставить в 1.1

Настройка fillfactor

- Определяется для таблицы
  - Насколько INSERT новых строк может заполнить страницу (по умолчанию 100%). Остаток места только под UPDATE
  - Нужно для **Heap-Only Tuples (HOT)** обновлений - изменяем только запись в **heap**, не трогая индексы
  - Статистика:
    - SELECT relname, n_tup_upd, n_tup_hot_upd, (n_tup_hot_upd::float / NULLIF(n_tup_upd,0))*100 AS hot_ratio FROM pg_stat_user_tables;
- Зачем?
  - Уменьшаем overhead обновлений индекса

1. Введение
2. Основы
3. План запроса
4. PostgreSQL подходы
5. Архитектурные подходы

Счетчики…

А нужны ли нам вообще счетчики?

- В уведомлениях возможно стоит перейти на bubble вид
- Не ускорит кейс, когда нет уведомлений и нет индекса
- Продукт часто будет против

Нужны ли нам точные счетчики?

- Есть компромисс
- Для продукта
  - Перейти на “10+ непрочитанных сообщений”
- Для внутренних графиков
  - SELECT reltuples::bigint AS estimate FROM pg_class where relname = 'mytable';
  - Точность зависит от вашего кейса

Eventually точные счетчики

- Для тех, кому нужна точность
- Используем **Transactional Outbox** паттерн
- В 1 транзакции меняем объект и записываем в таблицу событие
- Предагрегируем счетчики в отдельном процессе
- Не забываем мониторить задержку в обработке

Про денормализацию

Что это?

- Дублирование данных в нескольких таблицах/хранилищах
- Зачем?
    - Сокращаем время на соединение таблиц
    - Предагрегируем и сокращаем время на чтение
    - Облегчаем таблицы для быстрого поиска

Локальная

- Как выглядит?
  - Выносим поля для поиска в небольшие по размеру строк таблицы
  - Например описание объявления в отдельную таблицу
- Зачем?
  - Поможет если запросов на чтение больше чем на запись
  - Больше строк влезает в **shared buffers**

Глобальная

- Можно выносить в отдельные хранилища
    - Elasticsearch для ранжирования и fulltext search
- **Eventual Consistency**
- Все изменения все равно через PostgreSQL!
- Используем **Transactional Outbox**
- Поможет и с переездом целиком на другое хранилище

Read Only реплика

- Реплика уже есть у многих баз
- Используем и ее ресурсы для чтения (например, аналитики)
- Только для кейса с 1 **master** и 2 репликами (защита от перегрузки)

Особенности

- При асинхронной репликации есть задержка в актуальности данных
- При синхронной репликации проблемы реплики влияют на **master**
- Из-за конфликта очистки **VACUUM** с репликой могут быть ошибки:
    - ERROR: canceling statement due to conflict with recovery
- Помогают настройки:
    - **max_standby_streaming_delay** - сколько ждем перед отменой на реплике
    - **hot_standby_feedback** - меньше будет отмен из-за изменения данных на мастере

Выводы

- Не боимся денормализации, если не хватает производительности
- Но делаем в рамках транзакции или через **Transactional Outbox**
- Используем возможности **Read Only** реплики
- Но следим за отменой запросов из-за конфликтов

Общие выводы

- Минимизируем чтение с диска
  - Ориентируемся по плану запроса
  - Увеличиваем **shared buffers**
  - Накидываем индексы
  - В последнюю очередь партиции и денормализация
- Замедляем рост базы
  - Сокращаем размеры строк
  - Выносим крупные части в отдельные таблицы

Что еще почитать

- Мартин Клеппман “Высоконагруженные приложения. Программирование, масштабирование, поддержка”
- Алекс Петров “Распределенные данные. Алгоритмы работы современных систем хранения информации”
- Домбровская Г. Р., Новиков Борис “Оптимизация запросов в PostgreSQL”

Спасибо за внимание

Телеграм: @v2v21

Почта: kiverad@gmail.com

циан