Диагностика производительности PostgreSQL

или детектив под названием «что-то база тормозит»

Yandex Cloud

Технологическая платформа Яндекса
для создания и развития цифровых продуктов

Фомичев Степан

Старший разработчик программного обеспечения, Yandex Cloud

Работаю в команде Managed PostgreSQL

Управляемыми базами данных занимаюсь последние 2 года

Прежде очень много пользовался managed- и unmanaged-СУБД в качестве продуктового разработчика и руководителя инфраструктурной команды

О спикере

Личный рекорд в беге на 10км – 37:02 🏃

Как начинается среднестатистический инцидент

Что-то тормозит или совсем не работает в самый неподходящий момент. Чинить надо быстрее, а времени разбираться с причинами совсем нет

Путь «на удачу»

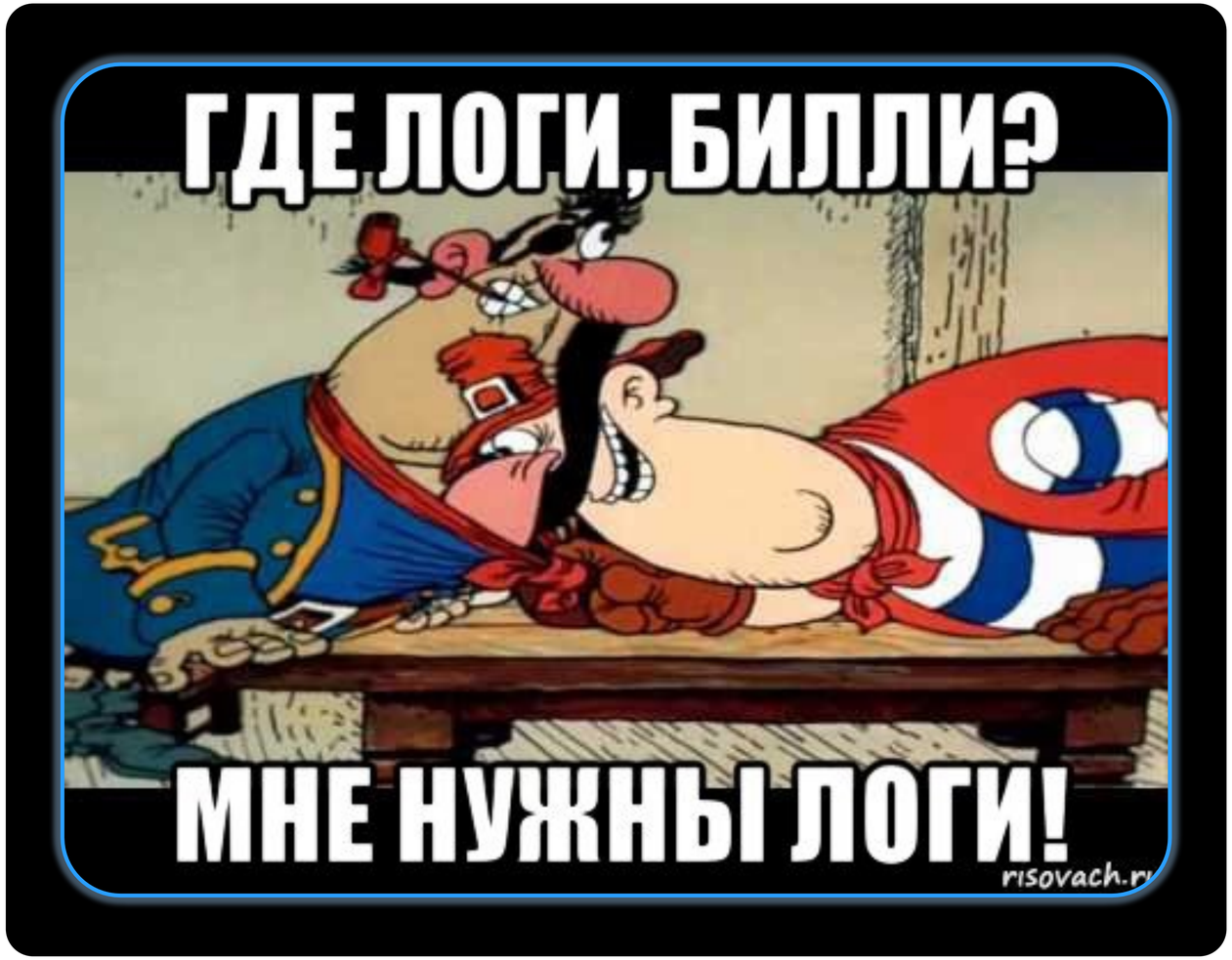
- Может помочь, но не самым оптимальным образом
- Может вообще не сработать
- А может и навредить

Попытка понять а что привело к инциденту на самом деле

- Позволит предотвратить подобные проблемы в дальнейшем
- Выбрать оптимальное решение по соотношению стоимость/эффективность

Краеугольный камень расследования

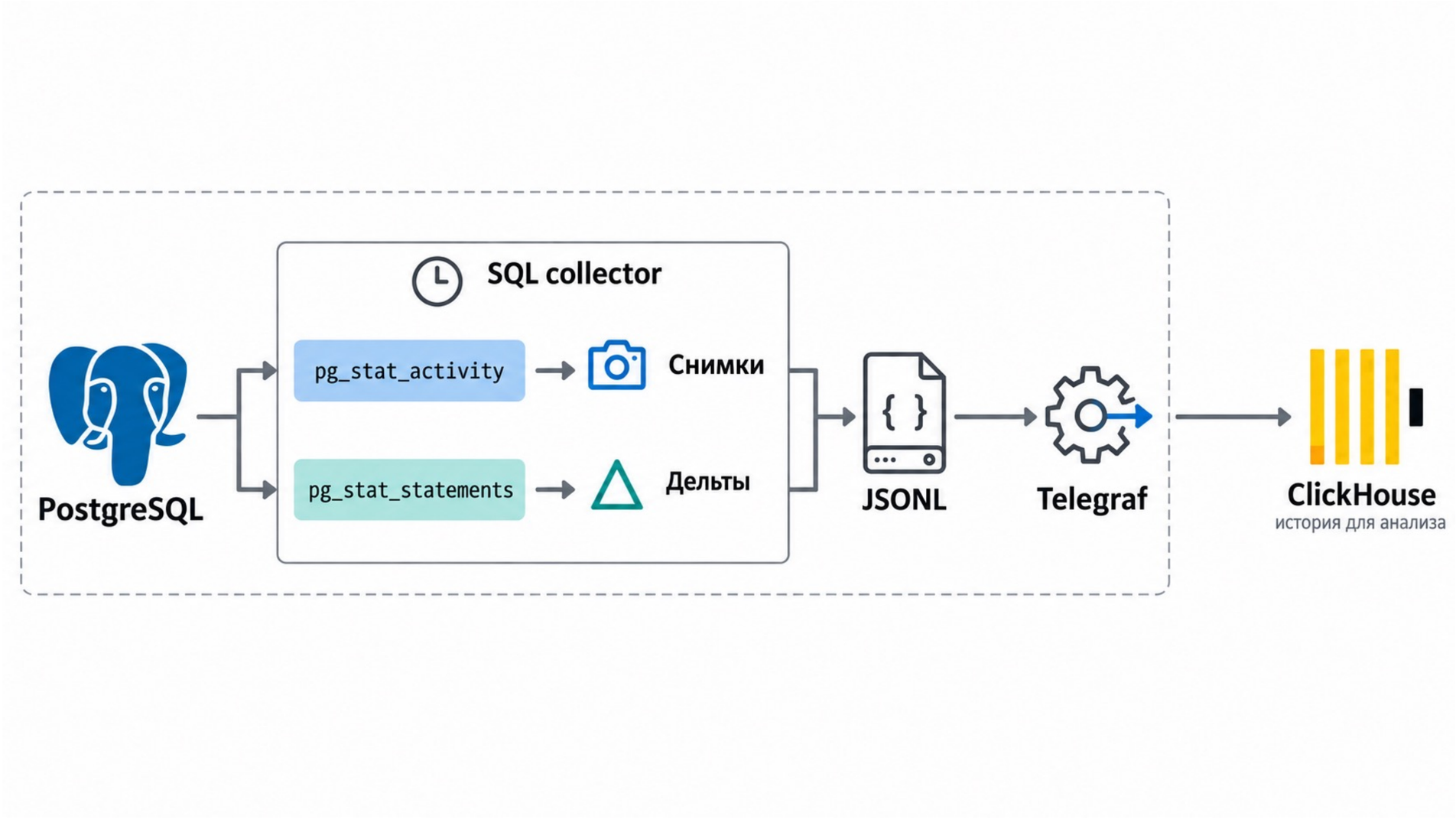
Для того, чтобы в моменте или постфактум понять реальную причину инцидента нужны данные

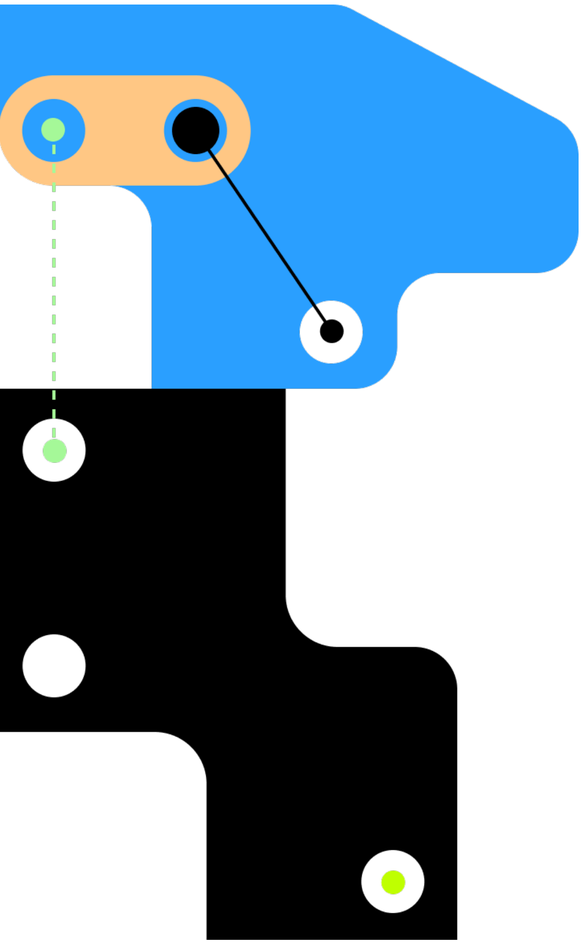


О сборе данных нужно подумать заранее

Что хочу узнать	Что смотрю	Что нужно включить заранее
Что происходит прямо сейчас	pg_stat_activity, pg_locks	log_lock_waits, писать application_name в приложении
Что менялось за период	pg_stat_statements, pg_stat_kcache	оба расширения
Сколько «мусора» в объектах	pg_stat_user_tables, pgstattuple	log_autovacuum_min_duration
Какие индексы реально используются	pg_stat_user_indexes	-
Как выполняется конкретный запрос	EXPLAN (ANALYZE, BUFFERS)	log_min_duration_statement
Какой план был в определенный момент времени	auto_explain, pg_stat_query_plans	оба расширения

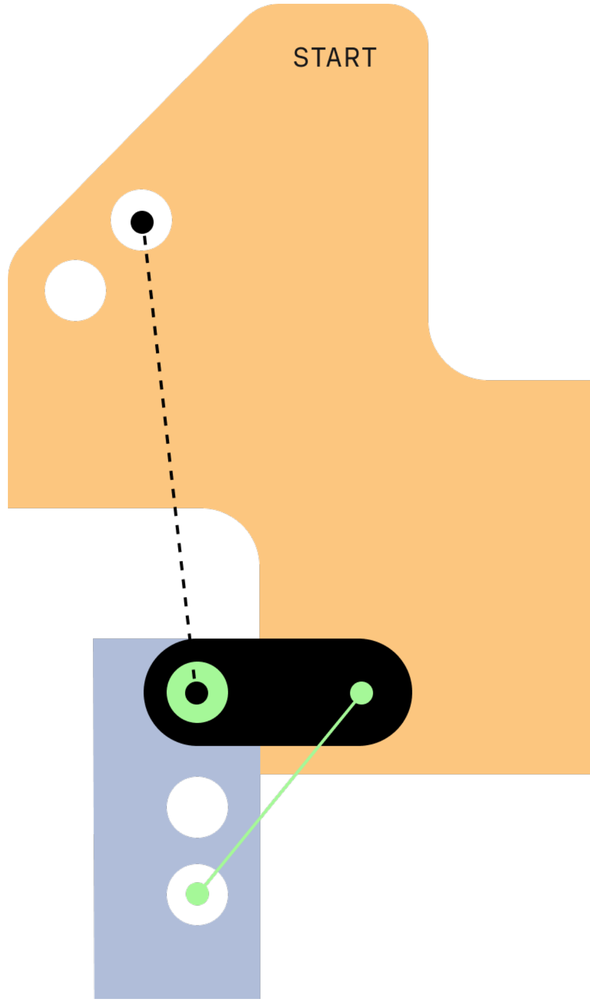
Про сбор информации





А проблема действительно в базе?

Вечер, пятница, дежурство



Пользователи массово жалуются на задержки и зависания

В логах приложения видим ошибки вида:

```
Caused by: java.sql.SQLTransientConnectionException:
    HikariPool-1 - Connection is not available,
    request timed out after 30000ms.
```

Логи и графики PostgreSQL 👀

Графики приложения 👀

Трассировки приложения 👀

Смотрим мониторинги PostgreSQL

Диагностика производительности

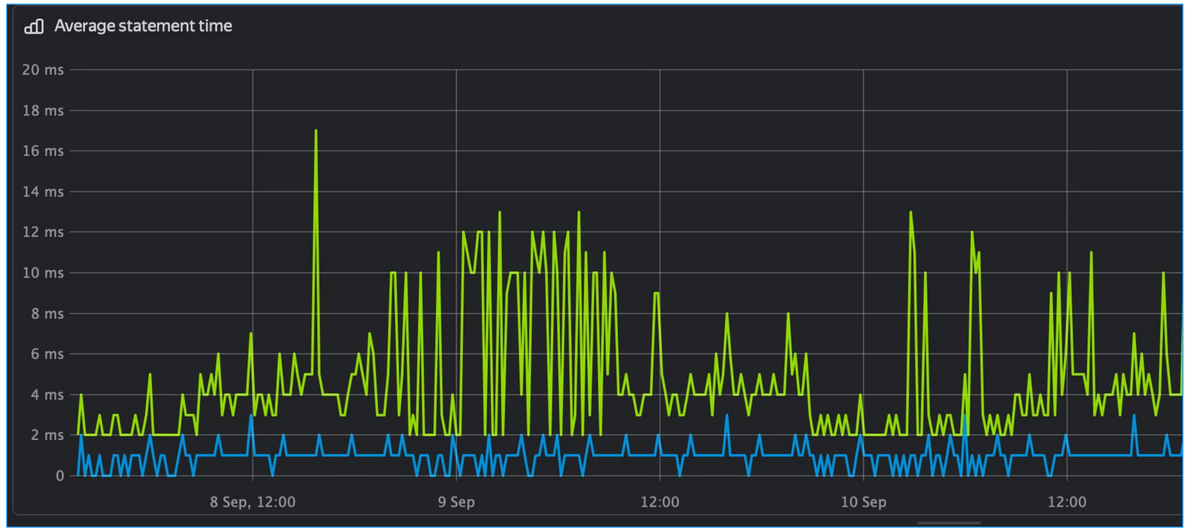
Сводная информация | Сессии | История сессий | Запросы

Фильтр | 10.09.2026 11:13:06

Idle in transaction: 147 | Active: 3

PID	State	Запрос	Transaction age, с	Wait event type	Wait event	Пользователь	База данных
807	idle in transaction	SELECT * FROM command WHERE transport_ord...	3	Client	ClientRead	conveyor_app	conveyor
804	idle in transaction	UPDATE transport_unit SET status = $1 WHERE id...	3	Client	ClientRead	conveyor_app	conveyor
805	idle in transaction	SELECT * FROM transport_order WHERE id = $1 A...	3	Client	ClientRead	conveyor_app	conveyor
806	idle in transaction	UPDATE command SET status = $1 WHERE trans...	3	Client	ClientRead	conveyor_app	conveyor
808	idle in transaction	UPDATE transport_order SET current_location = $...	3	Client	ClientRead	conveyor_app	conveyor
809	idle in transaction	SELECT * FROM transport_unit WHERE id = $1 A...	3	Client	ClientRead	conveyor_app	conveyor

- Среднее время исполнения запросов не выросло
- Есть живые сессии от нашего приложения
- State – idle in transaction, wait event - ClientRead

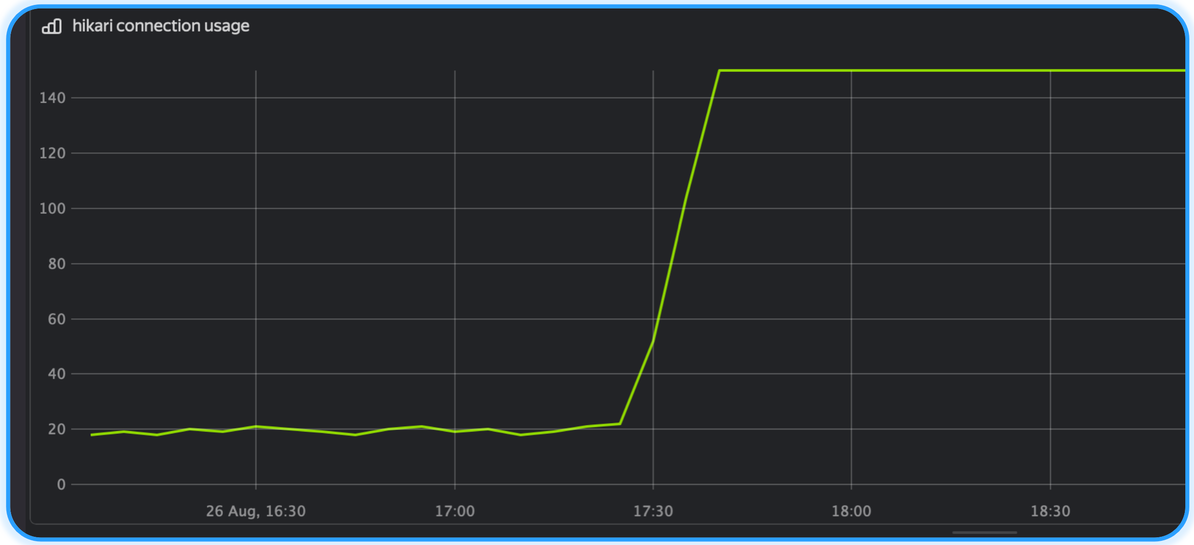


Сессии:
SELECT * FROM pg_stat_activity()

Время выполнения запроса:
avg_query_time (pgbounser/odyssey)

Смотрим мониторинги приложения

```yaml
datasource:
  pg:
    type: com.zaxxer.hikari.HikariDataSource
    platform: postgres
    driver-class-name: org.postgresql.Driver
    url: &jdbc-url jdbc:${PostgresDbMyServerUri}
    jdbc-url: *jdbc-url
    username: ${PostgresDbMyUserUser}
    password: ${PostgresDbMyUserPass}
    hikari:
      minimum-idle: 5
      maximum-pool-size: 50
      connection-timeout: 30000
```



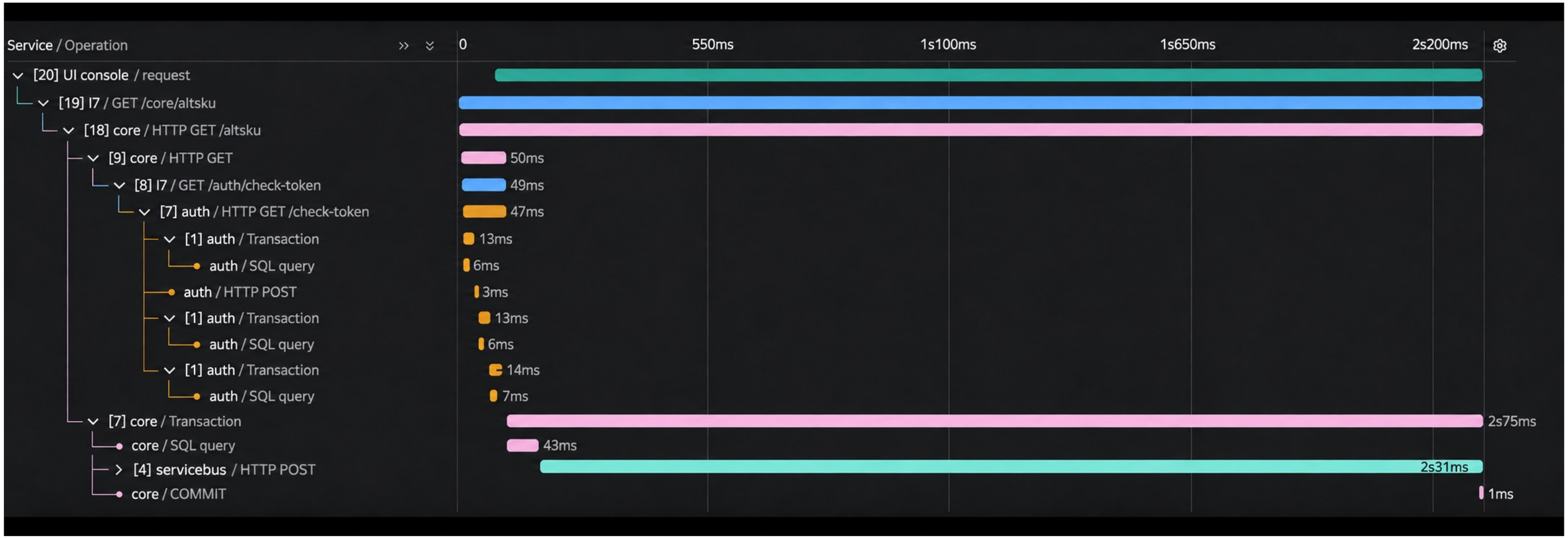
Имя	Разрешения	Лимит подключений	Способ авторизации	Настройки
my_user	db1	1000	Пароль	User password encryption: SCRAM-SHA-256

- Лимит соединений для my_user – **1000**, а в настройках hikariCP максимум - **50**
- У нас **3 инстанса приложения**. Получается, в пике мы можем использовать **максимум 150 соединений**

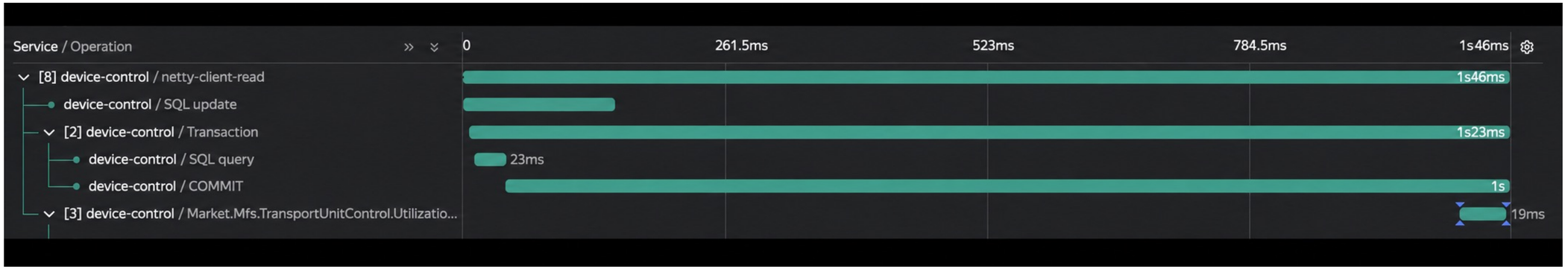
hikariCP + spring boot + prometheus:

Соединений активно: hikaricp_connections_active
Максимальное количество соединений: hikaricp_connections_max

Смотрим трассировки

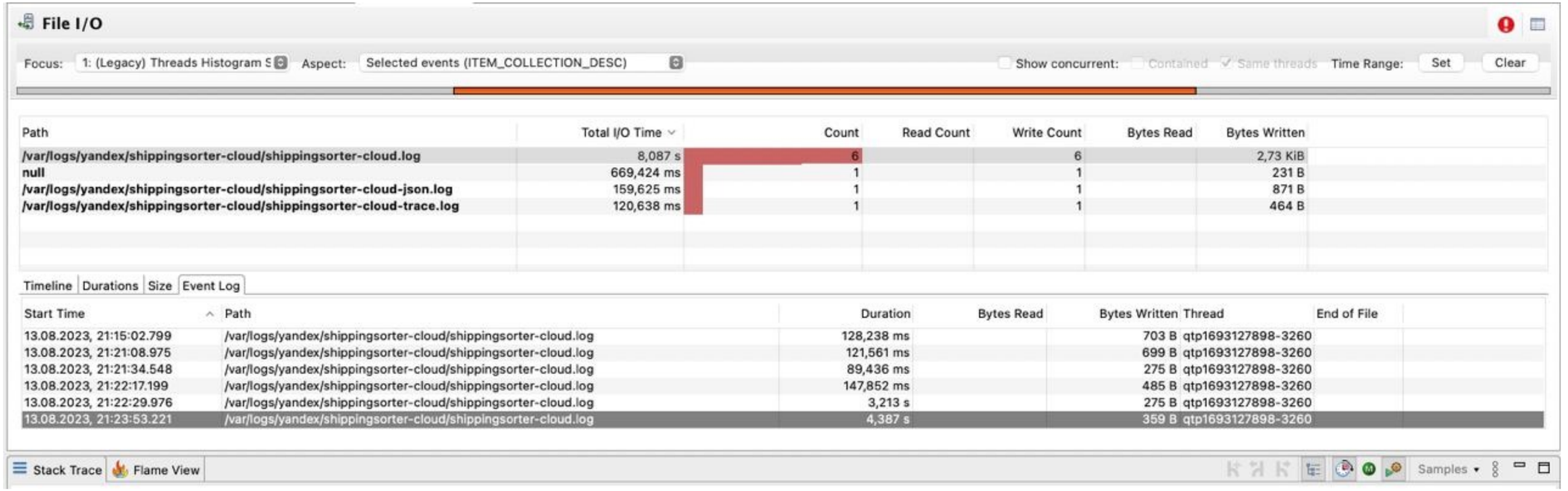


Почему нужно быть осторожнее с трассировками

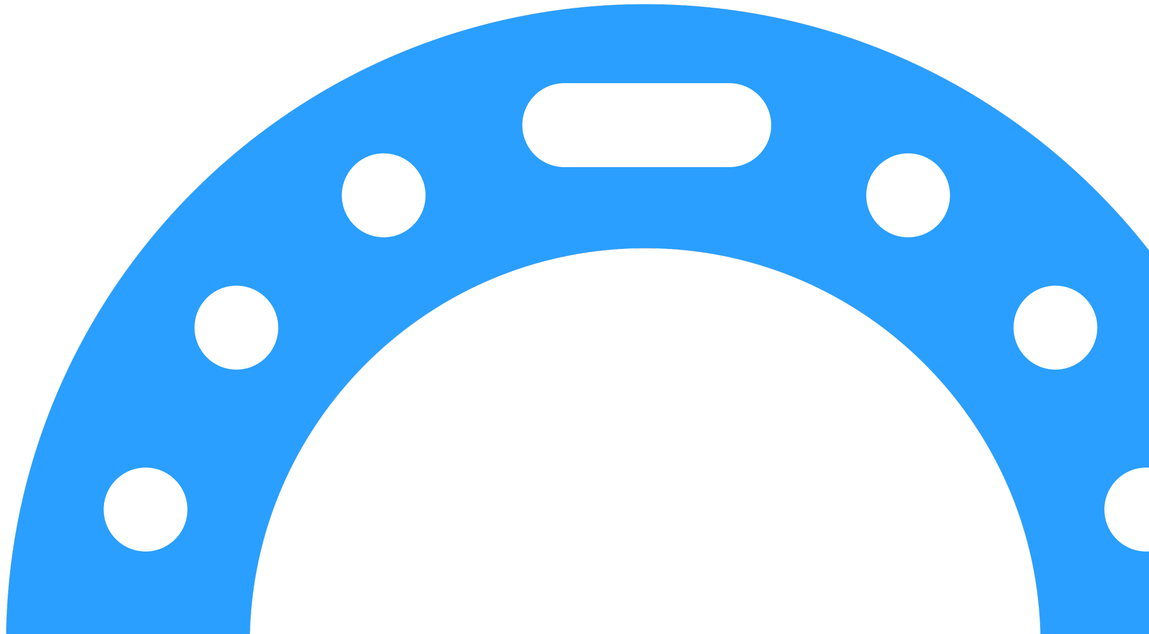


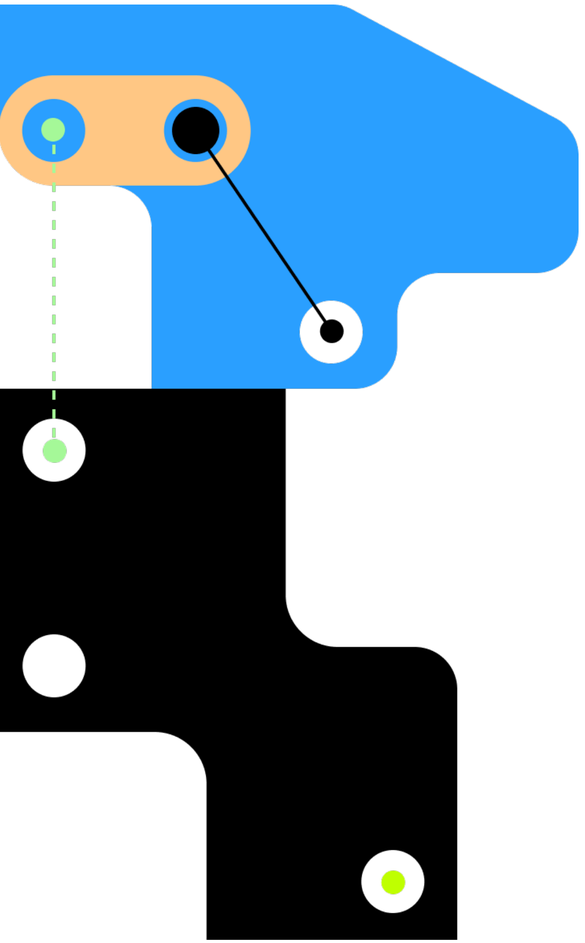
Очень долгий commit?

Почему нужно быть осторожнее с трассировками



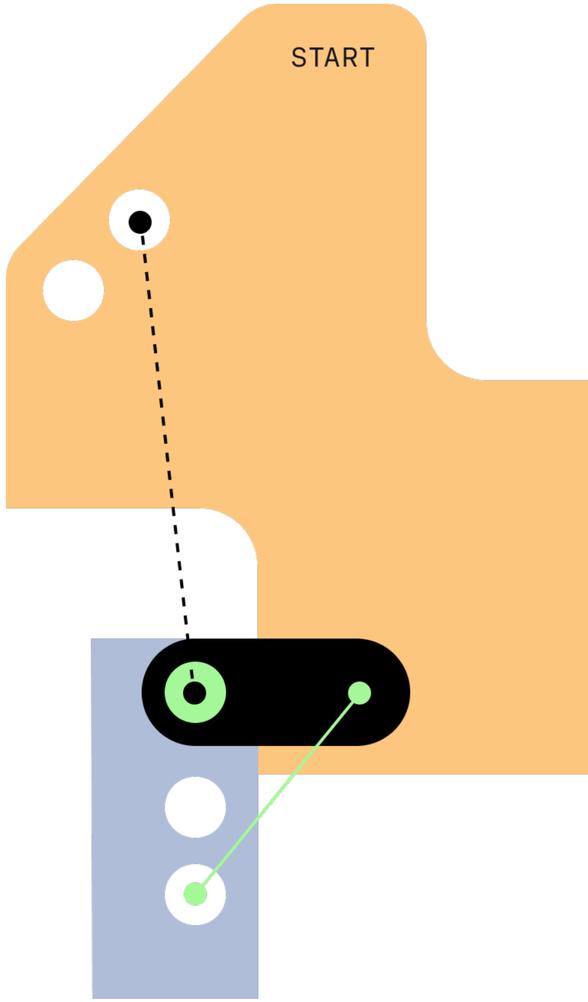
Даже если проблема выглядит как проблема с базой и пахнет как проблема с базой – есть ненулевой шанс, что она всё-таки не в базе





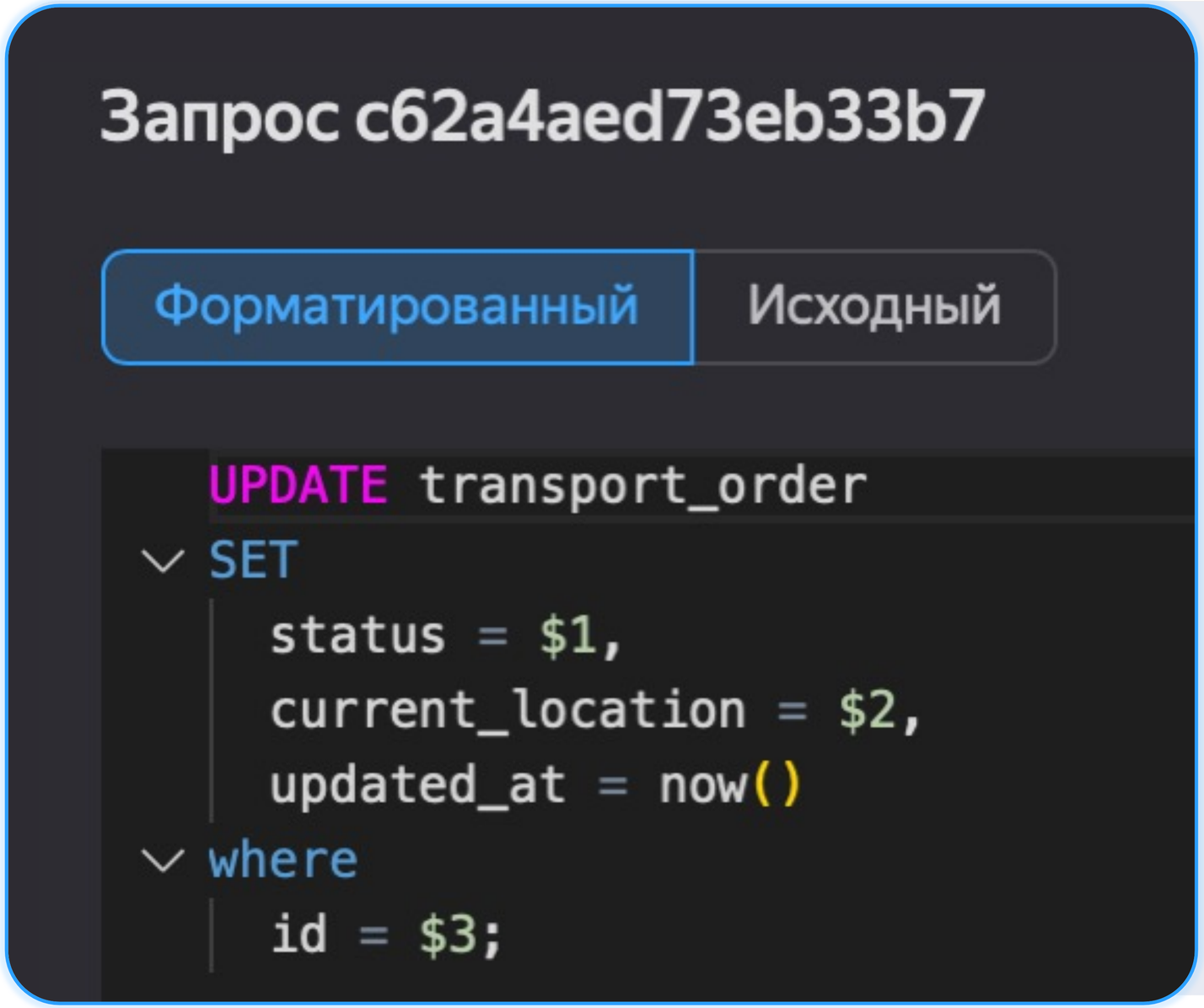
Тихая деградация

Или проблема, находящаяся в плоскости настроек СУБД

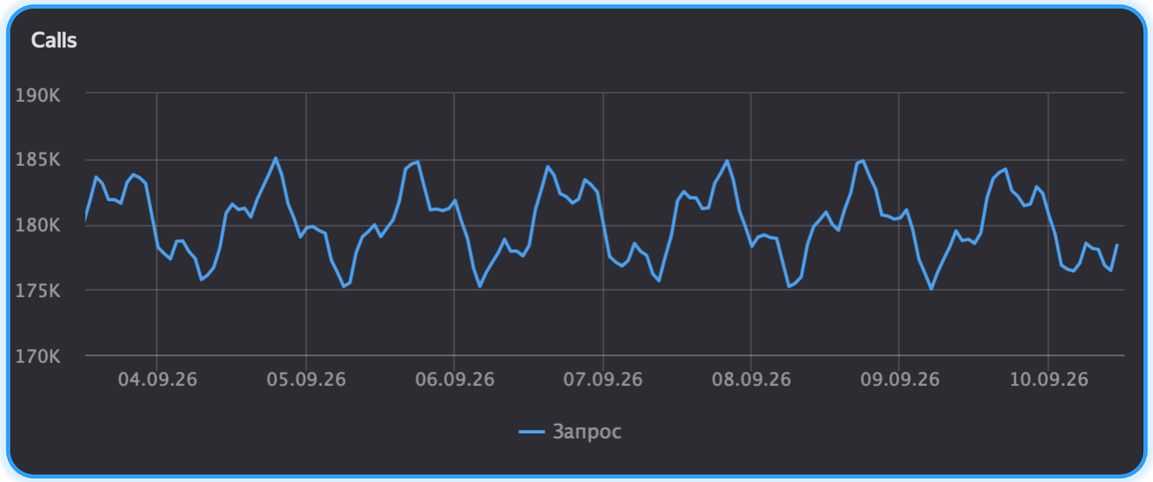


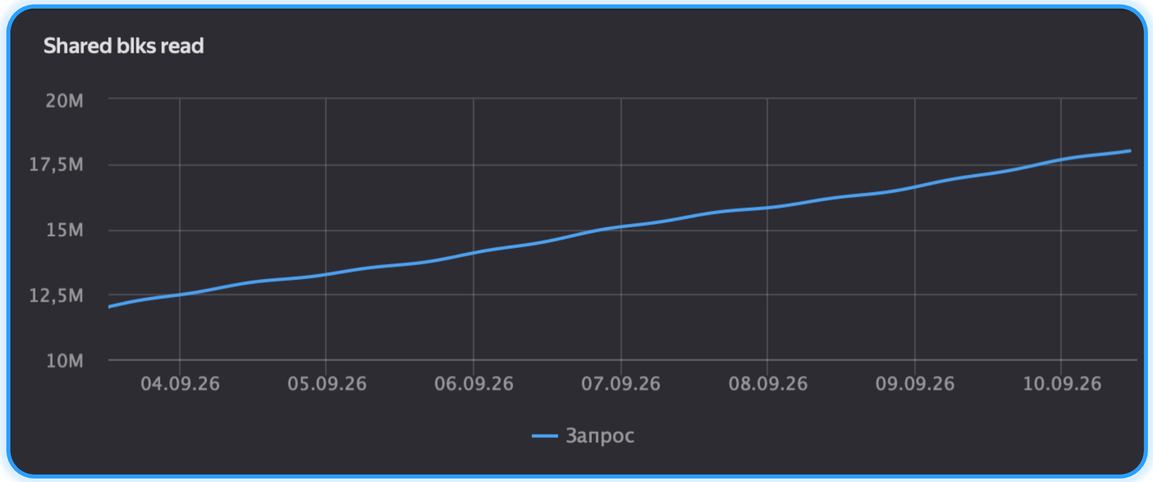
Обстановка

- Таблица **transport_order**: каждая посылка проходит несколько статусов, на каждый статус – **UPDATE**
- Всё работало нормально, но запросы к таблице **постепенно** замедлялись
- Со временем это стало заметно, начали загораться алерты на время ответа



Смотрим мониторинги приложения



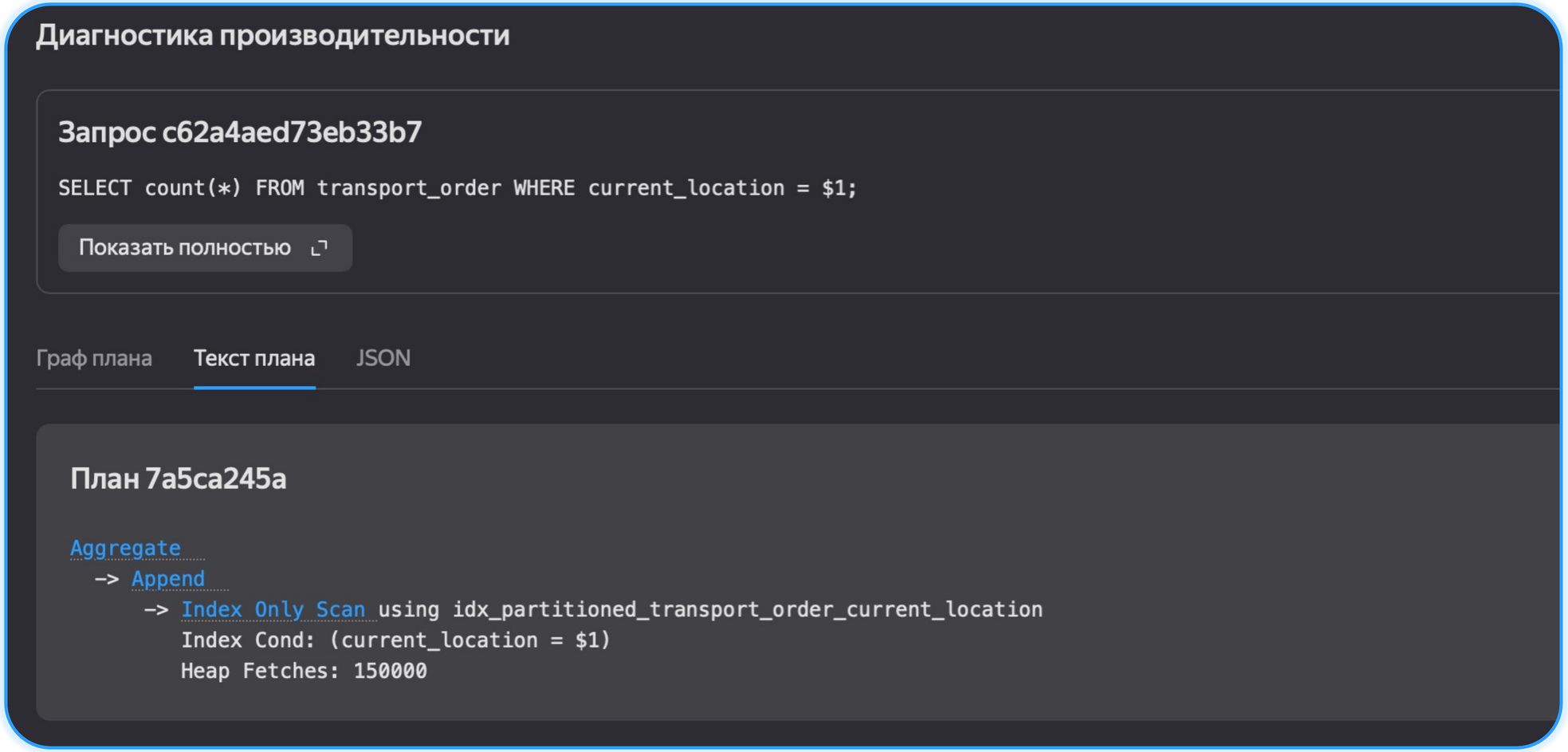


- **Количество запросов** к проблемной таблице либо не изменилось, либо незначительно выросло — и это никак не объясняет кратно выросшие тайминги
- Зато сильно выросла метрика **shared_blks_read** — значит, запрос стал читать больше страниц **не из shared buffers**

Получить значения calls и shared_blks_read для запросов можно с помощью расширений pg_stat_statements или pg_stat_query_plans

Дополнительно: расширение pg_stat_kcache для системных метрик

План одного из деградировавших запросов



План запроса в моменте можно получить выполнив запрос с опцией EXPLAIN
Для того, чтобы проблемные планы собирались автоматически можно использовать расширения auto_explain и pg_stat_query_plans

Почему Index-Only Scan перестал работать

MVCC

`UPDATE` в PostgreSQL создаёт новую версию строки и снимает признак **all-visible** со страницы

autovacuum

Пока autovacuum эту страницу не обработает, **Index-Only scan** вынужден проверять видимость строки **в heap**

Изменилось физическое состояние таблицы

```
relname             | transport_order_transport_conveyor
n_tup_upd           | 928145061
n_tup_newpage_upd   | 784312770
n_live_tup          | 10842617
n_dead_tup          | 14287319
last_autovacuum     | 2026-09-10 10:19:40.967484+03
autovacuum_count    | 1572
```

- Мёртвых строк больше, чем живых.
- Очень много обновлений - о чём мы и так знали.
- Autovacuum запускался недавно

Системное представление pg_stat_user_tables

Откуда берутся мертвые версии строк

В **PostgreSQL** для обеспечения уровней изоляции реализован **MVCC - Multiversion Concurrency Control**

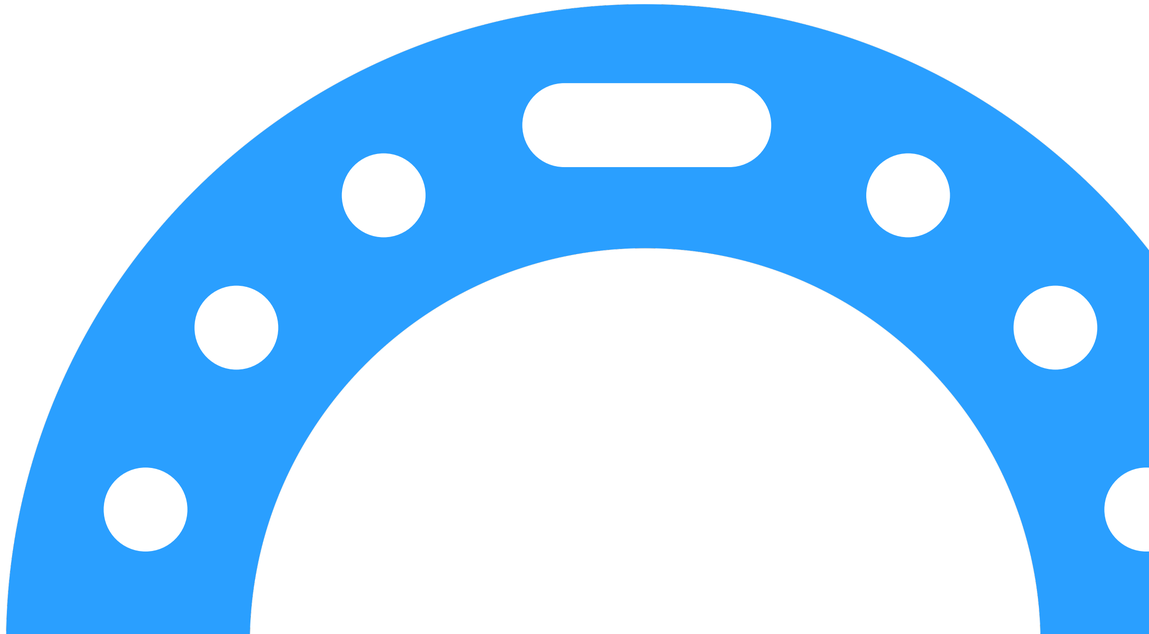
По сути каждая транзакция видит свой
особый snapshot базы, зафиксированный на момент начала транзакции

Каждый **`UPDATE`** создаёт новую версию строки. Старая версия становится ненужной для новых запросов, но ещё может быть видна транзакции, которая получила snapshot до обновления

Vacuum удаляет версию только тогда, когда гарантированно ни один существующий snapshot
больше не сможет её увидеть

Autovacuum может «не справляться» из-за не подходящих значений настроек.

Также очистку могут блокировать долго не завершающиеся транзакции



Три группы настроек autovacuum

Когда запускать

- autovacuum_vacuum_scale_factor
- autovacuum_vacuum_threshold

Сколько ресурсов может использовать

- autovacuum_vacuum_cost_limit
- autovacuum_vacuum_cost_delay

Параллельность

- autovacuum_max_workers

Долгие транзакции

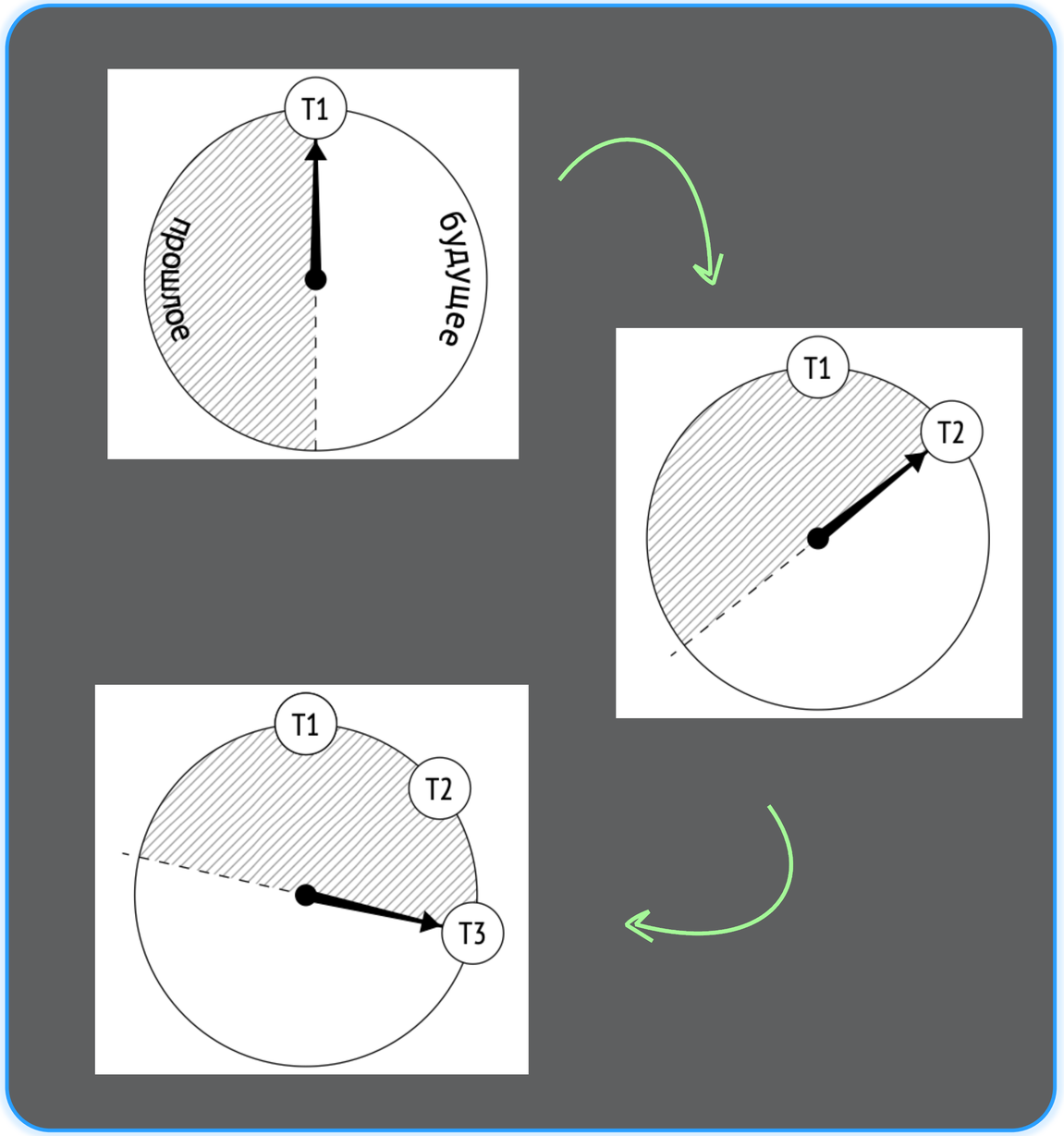
- Если есть долгие транзакции, то, как ни тюнингуй **vacuum**, мёртвые версии строк удалить не получится
- Тут уже нужно искать зависшие транзакции или запросы и разбираться с ними. Сделать это достаточно просто через **`pg_stat_activity`**

Мониторинг долгих транзакций - max(pg_stat_activity.xact_start)

Текущие сессии - системное представление pg_stat_activity

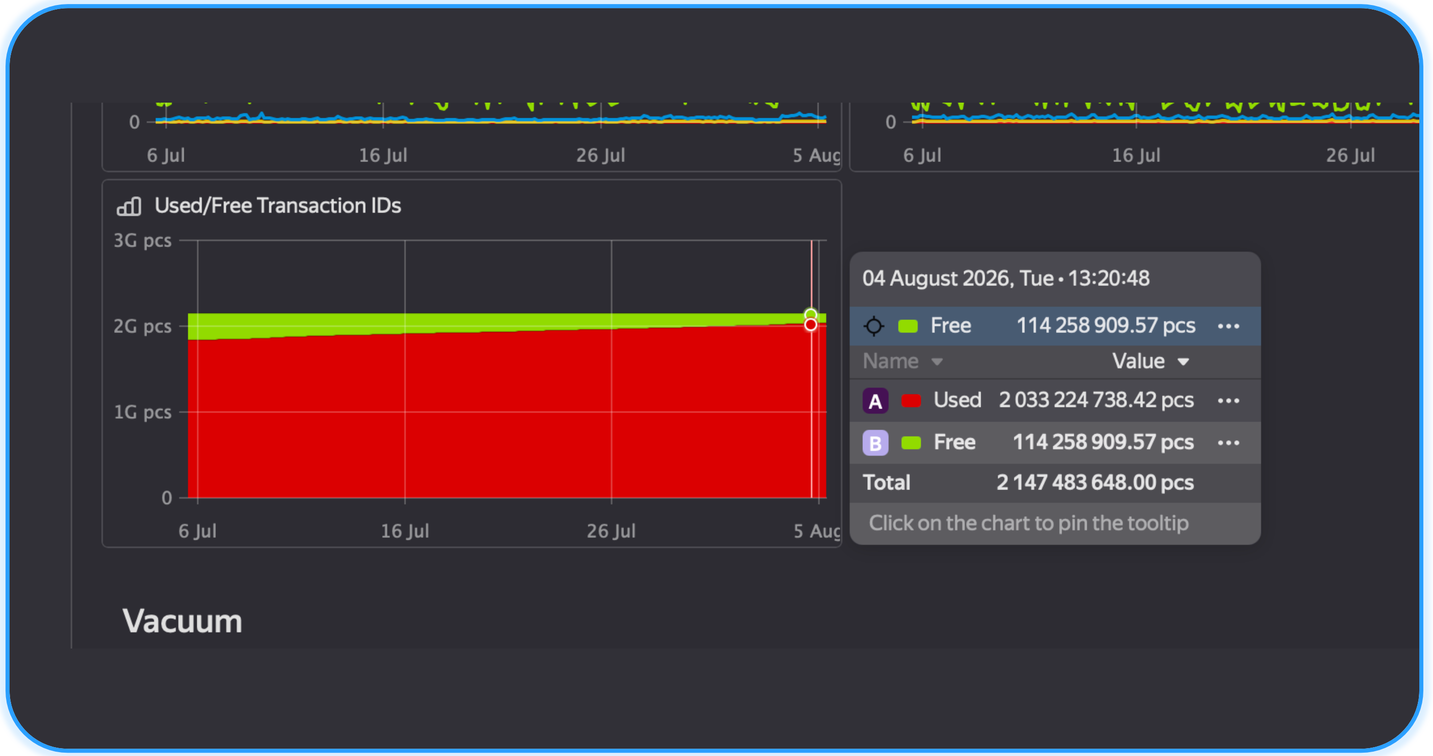
Зачем еще нужен vacuum: freeze

- Изменяющие данные транзакции получают **Transaction ID** - 32-битное число. Счётчик увеличивается, а достигнув конца диапазона, возвращается в начало
- PostgreSQL рассматривает пространство идентификаторов как **кольцо**: относительно текущей транзакции примерно половина значений считается прошлым, половина — будущим
- Каждая версия строки **хранит ID транзакции**, которая её создала — чтобы решить, видна ли эта версия запросу.
- Поэтому вакуум делает **freeze** для всё ещё актуальных строк: помечает, что транзакция точно в прошлом и видна всем, а идентификатор можно использовать дальше



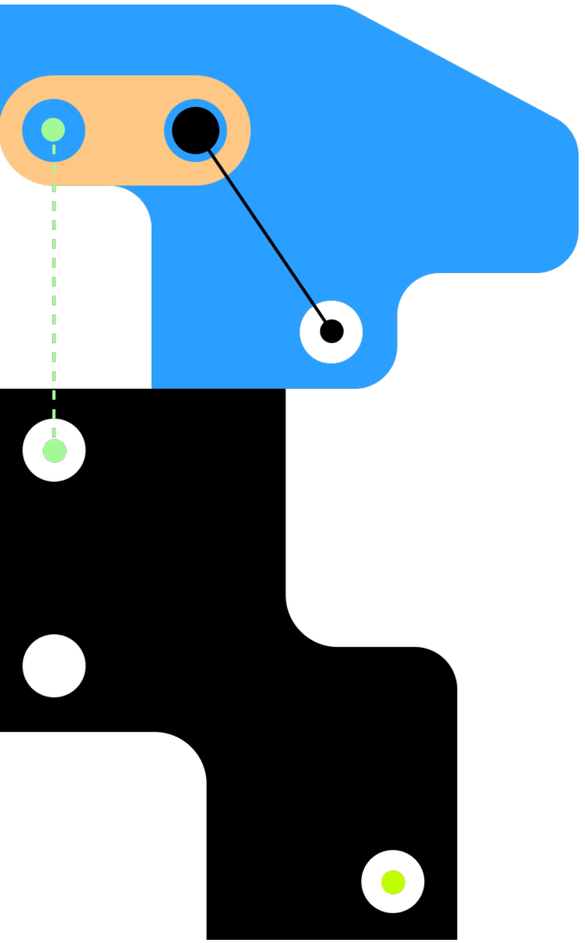
Если vacuum не успевает

- Мы используем больше идентификаторов за единицу времени, чем успеваем освобождать
- У PostgreSQL начинают постепенно заканчиваться свободные идентификаторы транзакций
- Если останется около **трёх миллионов** свободных - PostgreSQL уйдёт в аварийный **read-only** режим и будет позволять запускать только vacuum, во избежание коррупций данных

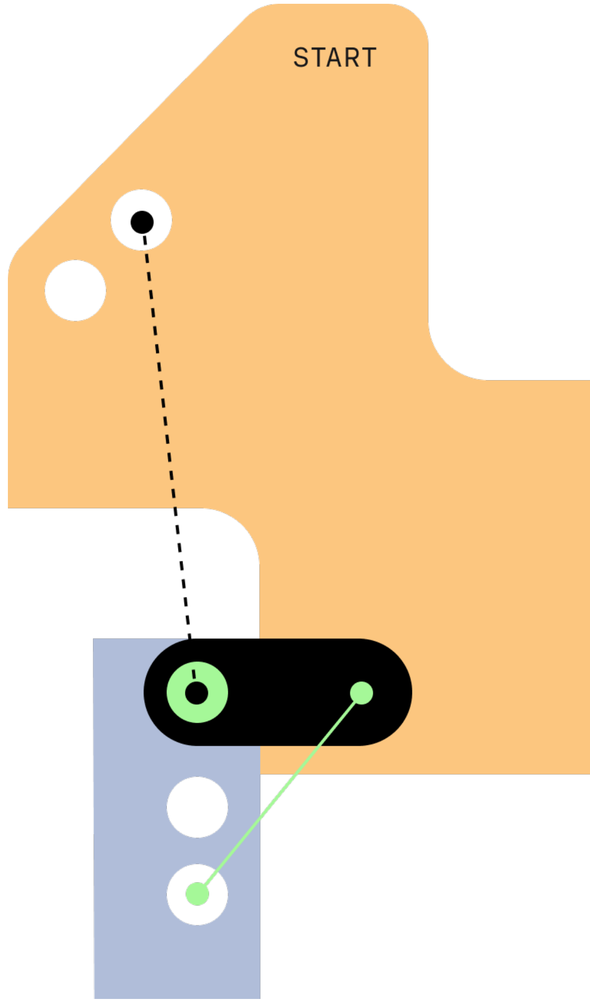


Сколько осталось XID:

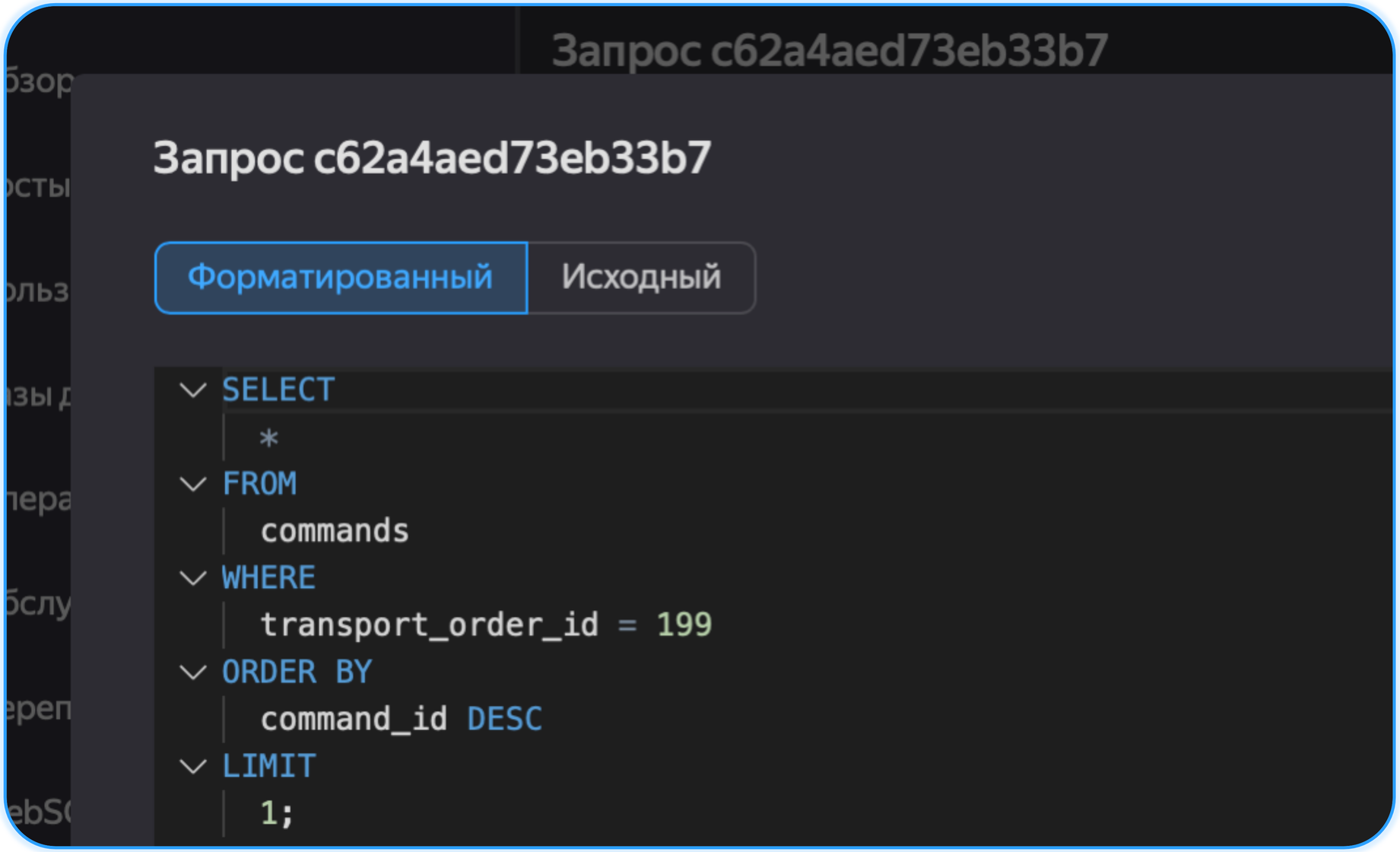
```
SELECT min(pow(2, 31) - 1 - age()) AS xid_left
FROM pg_database;
```



Проблема на уровне конкретного запроса



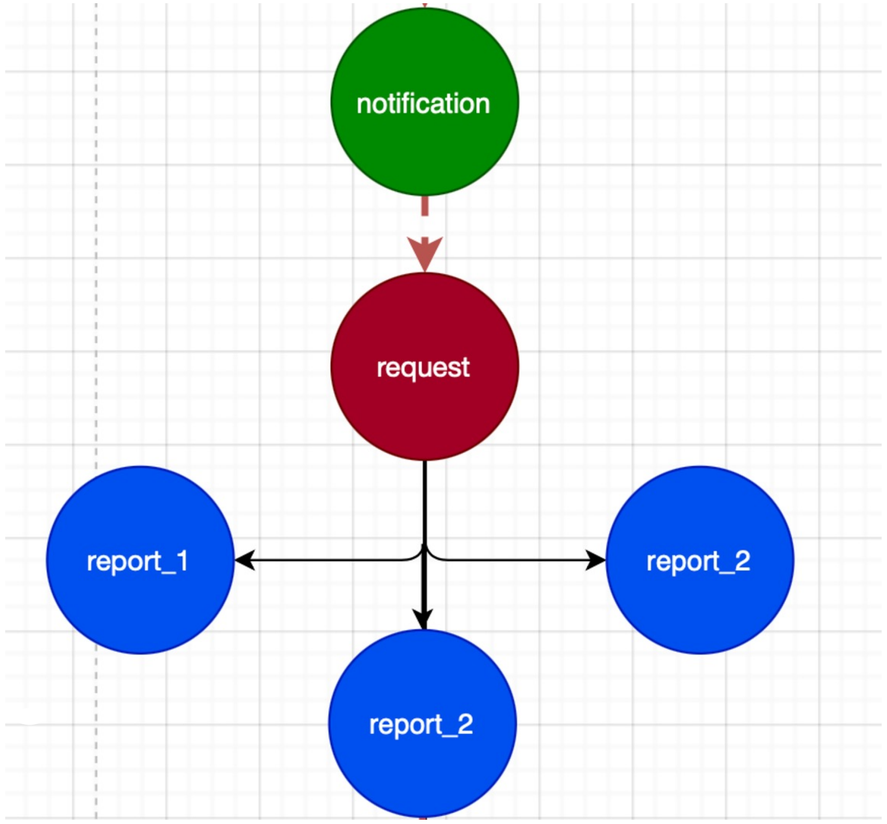
Утро субботы



Конвейер стоит, контейнеры не едут

Как это выглядело в моменте

Принцип работы

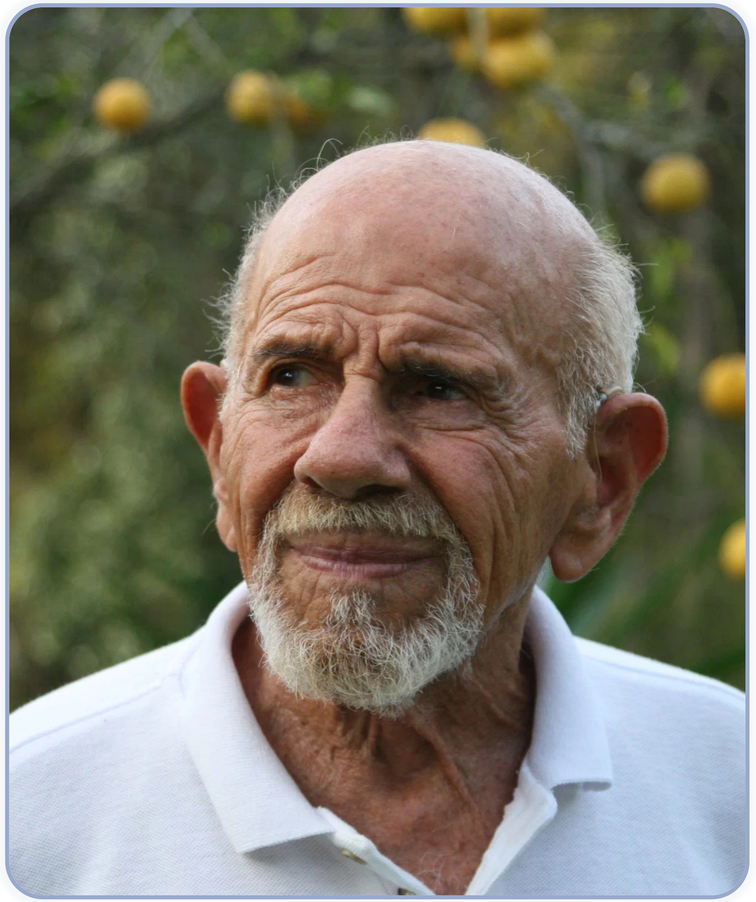


Загадка

Новых релизов с нашей стороны не было давно

Запрос не стал выполняться чаще

Необходимый индекс по **transport_order_id** есть – я это знаю, так как сам же его и создавал



EXPLAIN ANALYZE

```
Limit  (cost=0.43..1163.17 rows=1 width=30)
       (actual time=4180.282..4180.283 rows=0.00 loops=1)
  Buffers: shared hit=25745 read=78504 written=24915
  I/O Timings: shared read=3287.706 write=102.422
  ->  Index Scan Backward using commands_pkey on commands
        (cost=0.43..361611.43 rows=311 width=30)
        (actual time=4180.280..4180.281 rows=0.00 loops=1)
        Filter: (transport_order_id = 199)
        Rows Removed by Filter: 10000000
        Index Searches: 1
Planning Time: 0.510 ms
Execution Time: 4180.416 ms
```

Что здесь не так

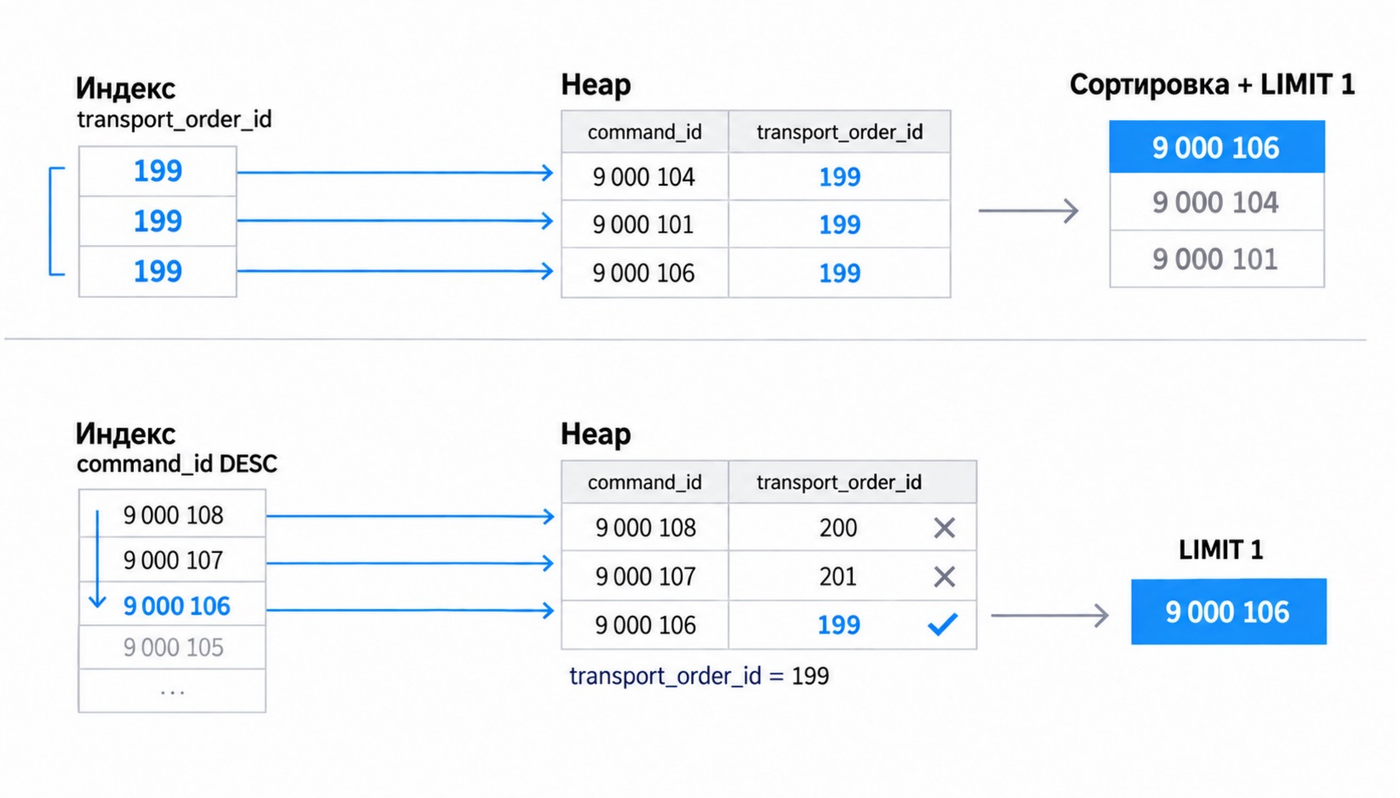
Вместо ожидаемого индекса по **`transport_order_id`** используется **primary-key index по command_id**

Страшные цифры в графе **`Rows Removed by Filter`** - как будто бы мы прошлись вообще по всем записям в таблице. Что в общем-то так и было

Планировщик ожидал найти **311 строк**, а нашёл **0** — то есть строки, которую мы искали, в таблице вообще нет

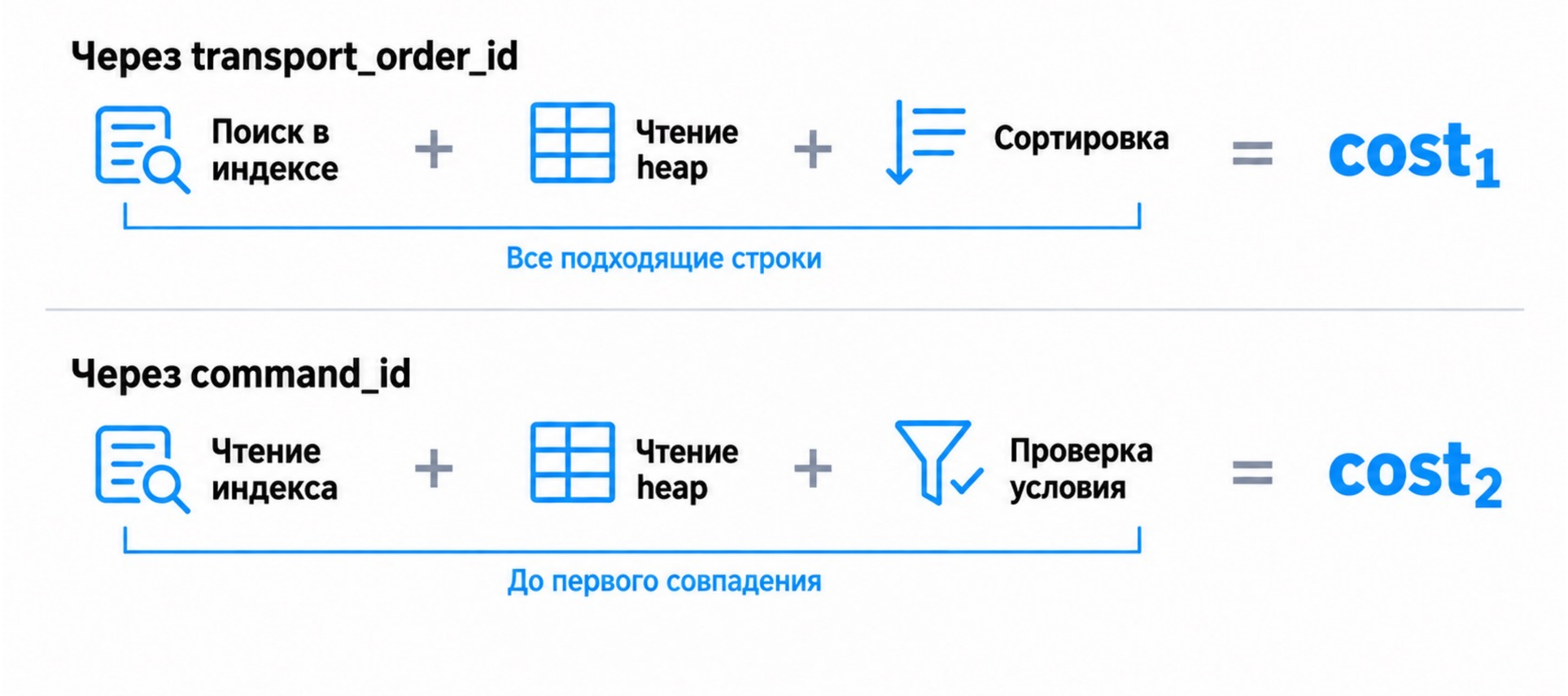
Два возможных пути исполнения

- Найти все команды для этого задания через индекс по transport_order_id и отсортировать их
- Идти в обратном порядке по индексу по `command_id`, где команды уже и так отсортированы, и просто фильтровать строки



Из чего складывается стоимость

Стоимость ≈ ∑ («объем работы» x «условная цена операции»)



Почему PostgreSQL выбирает неверный путь

- Строк с нужным **transport_order_id** у нас оценочно **10000000 / 32154 ~= 311**
- Они распределены **равномерно** по всей таблице
- Первую строку (для LIMIT 1) мы должны найти в перебрав **1/311 часть таблицы**

```
for (i = 0; i < sslot.nnumbers; i++)
    sumcommon += sslot.numbers[i];
selec = 1.0 - sumcommon - nullfrac;
CLAMP_PROBABILITY(selec);

/*
 * and in fact it's probably a good deal less. We approximate that
 * all the not-common values share this remaining fraction
 * equally, so we divide by the number of other distinct values.
 */
otherdistinct = get_variable_numdistinct(vardata, &isdefault) -
    sslot.nnumbers;
if (otherdistinct > 1)
    selec /= otherdistinct;
```

Какие были варианты решения

Переписать запрос

```
SELECT *
FROM commands
WHERE transport_order_id = $1;
```

Создать или изменить текущий индекс

```
CREATE INDEX CONCURRENTLY idx_commands_order_command
    ON commands (transport_order_id, command_id);
```

Подведем итоги

	Проблема не в СУБД	Проблема на уровне настроек СУБД	Проблема с конкретным запросом или группой запросов
Что требуется	Метрики приложения, инфраструктурные метрики, трассировки, профилировщики	Метрики СУБД, pg_stat_user_tables, pg_stat_user_indexes, pg_stat_progress_vacuum etc	EXPLAIN (ANALYZE, BUFFERS) auto_explain, pg_stat_query_plan

Выводы

1. Для качественной диагностики нужно много чего настроить заранее
2. Данные – ключ к поиску правильного решения
3. СУБД – это одна из самых важных частей наших систем, которая очень часто является bottleneck
4. Диагностика может быть действительно интересной, увлекательной и остросюжетной

Дополнительные материалы

1. **«PostgreSQL 18 изнутри»** - очень интересно и доходчиво про то, как работает PG
   https://postgrespro.ru/education/books/internals

2. **«Dasha - дашборд PostgreSQL»** - пример того, как можно настроить сбор метрик
   https://habr.com/ru/articles/1062540/

Спасибо
за внимание!

Степан Фомичев

Разработчик Managed PostgreSQL,
Yandex Cloud

Tg: @instaHipsta

Cloud