



Оптимизация через партиципирование

История оптимизации сервиса в условиях ограниченных ресурсов

Содержание доклада



О сервисе

Назначение, внутреннее устройство, предварительная подготовка к нагрузкам



О сбое и первых мерах

Как возвращали показатели работы к приемлемым значениям



Полноценное решение проблемы

Партиципирование как гарантия стабильности на будущее. Как делали его на действующей БД



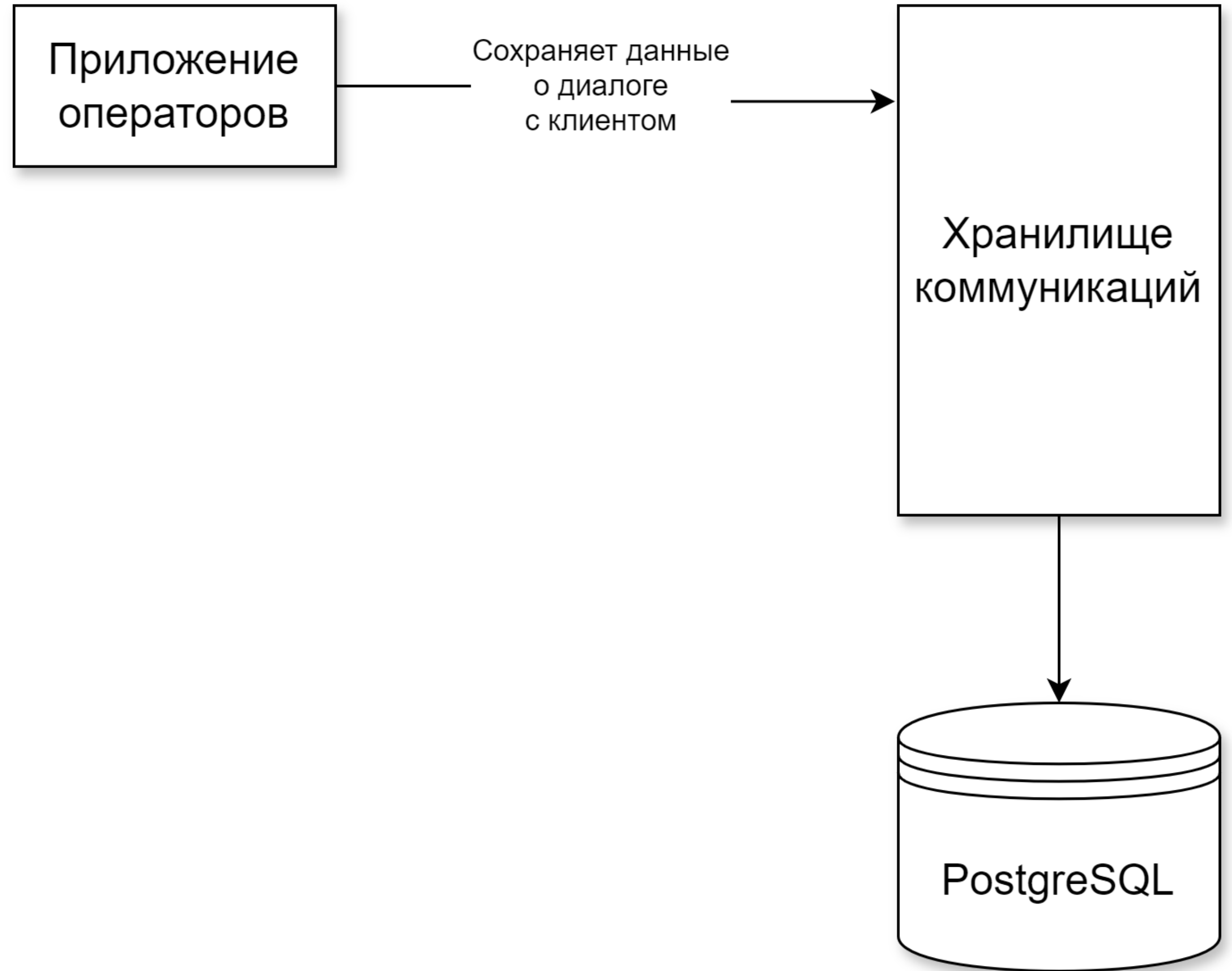
Подведение итогов

Выводы и рекомендации

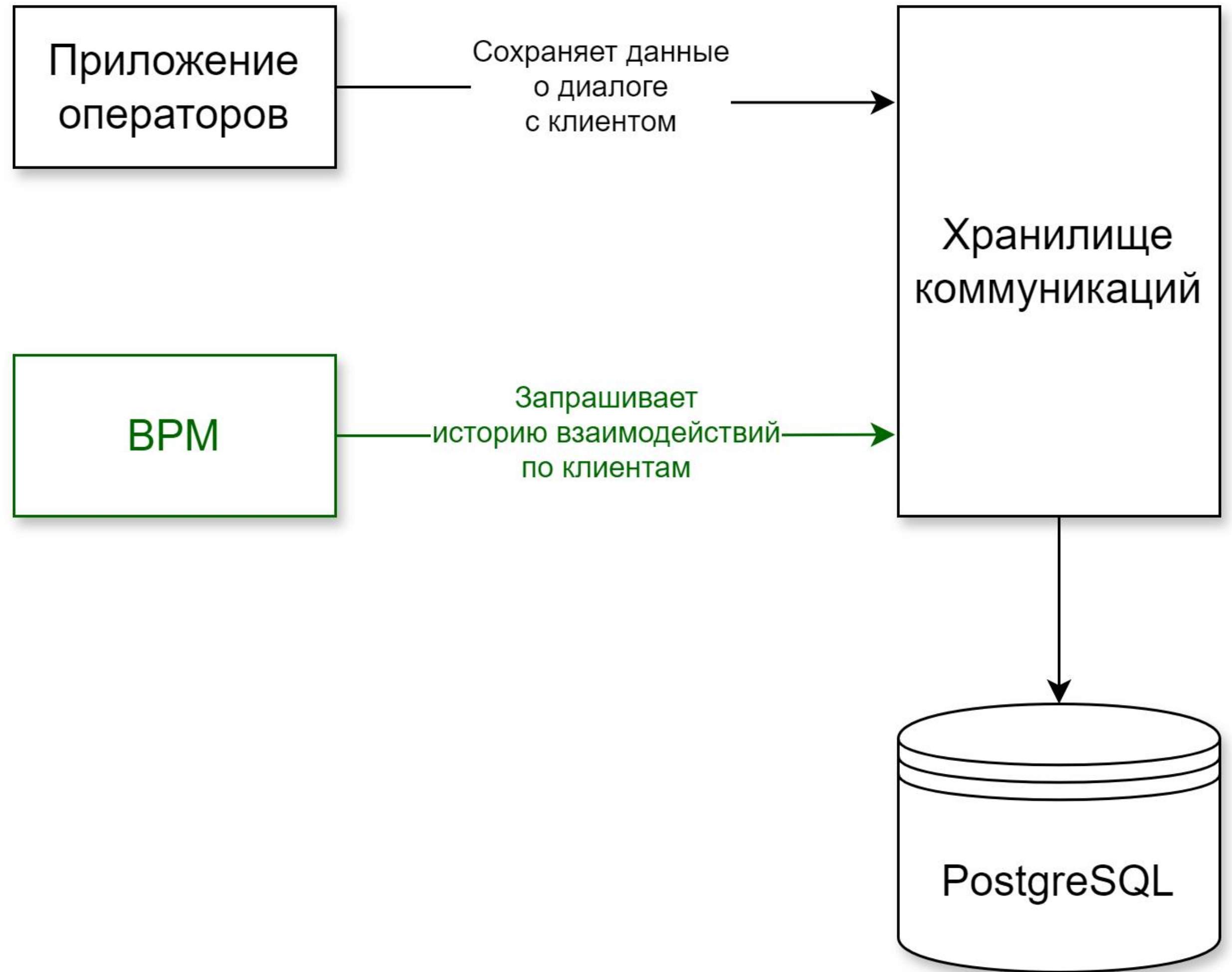
Сервис «Хранилище коммуникаций»

ака «Хранилище активностей»

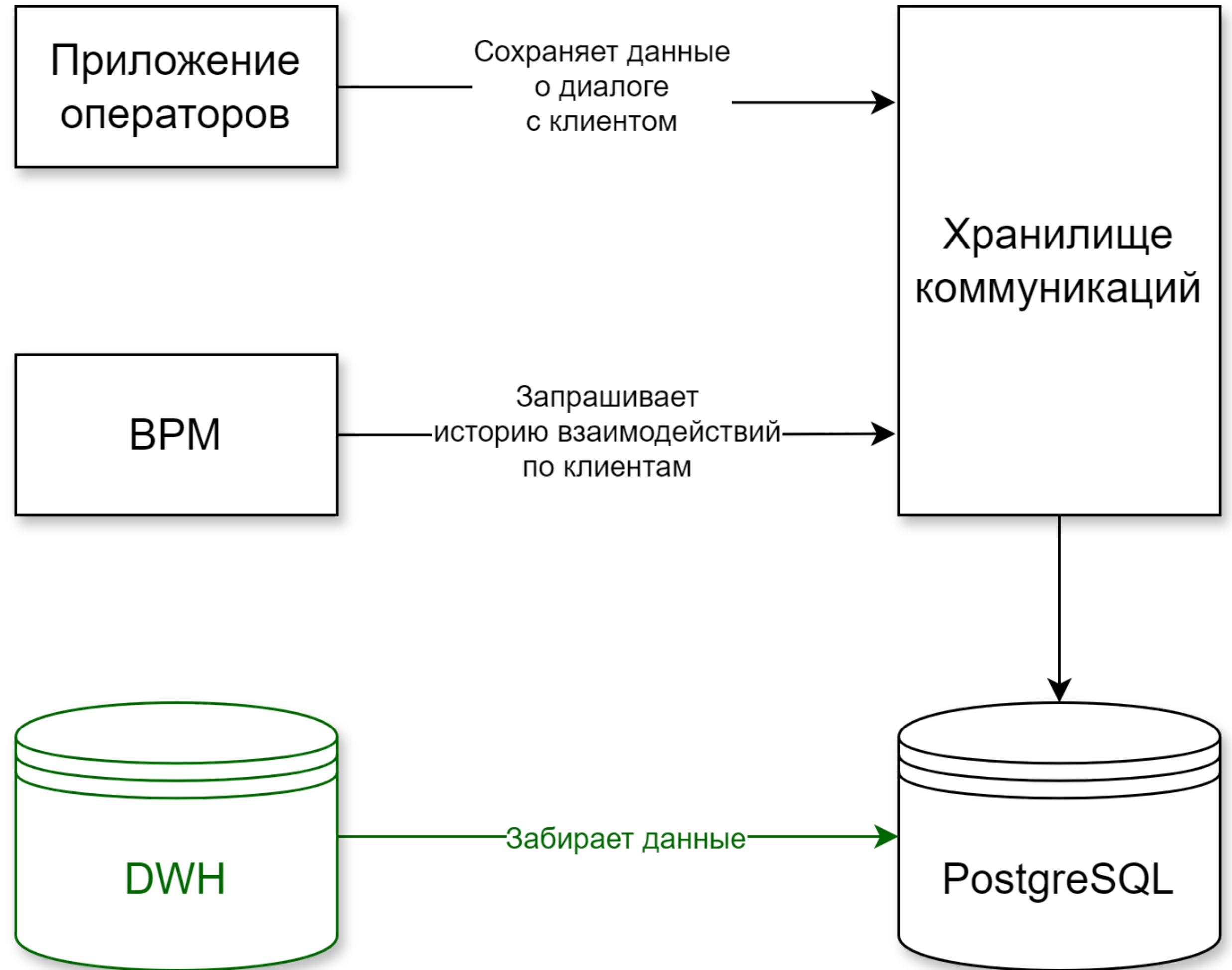
Назначение хранилища коммуникаций



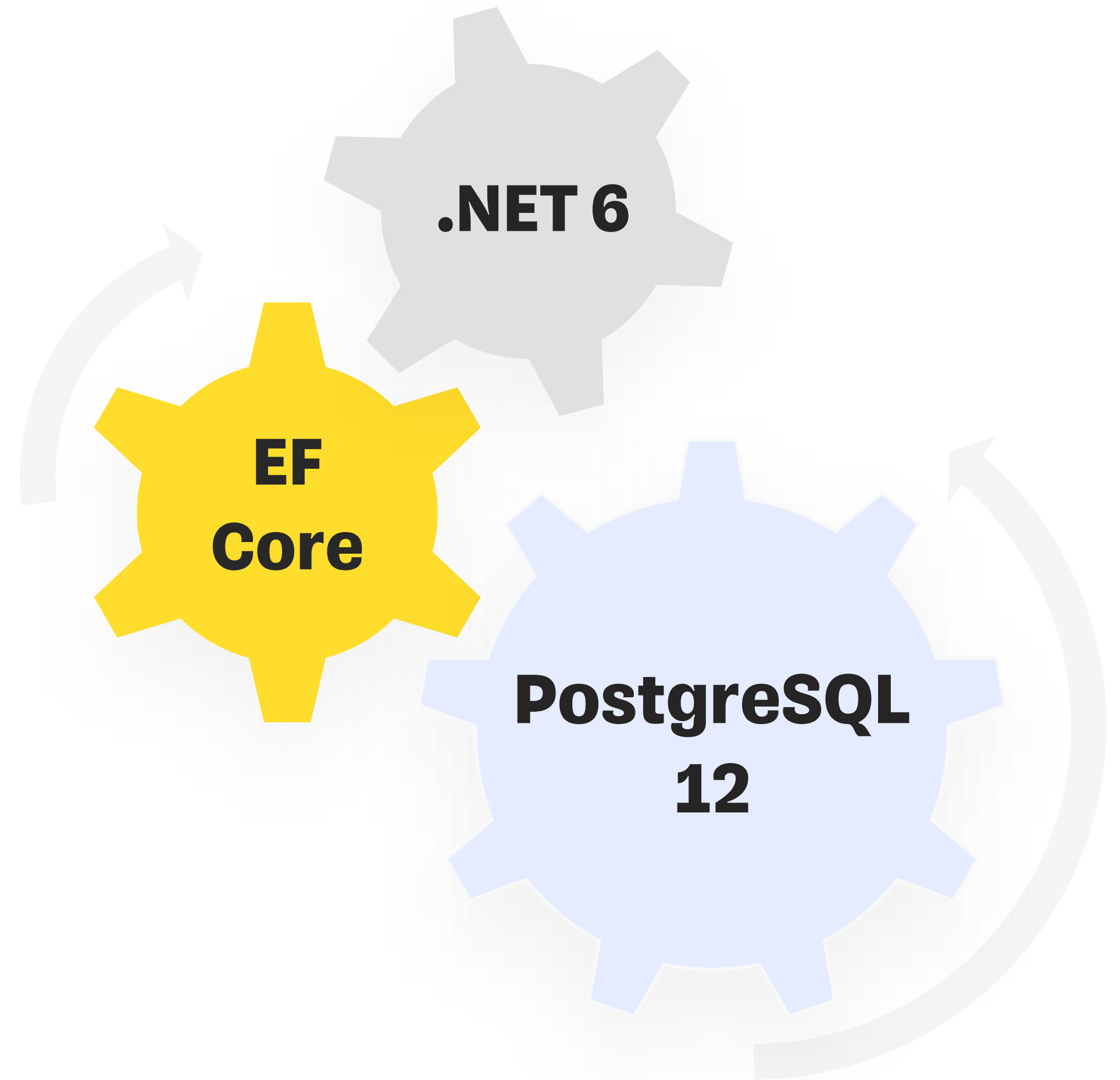
Назначение хранилища коммуникаций



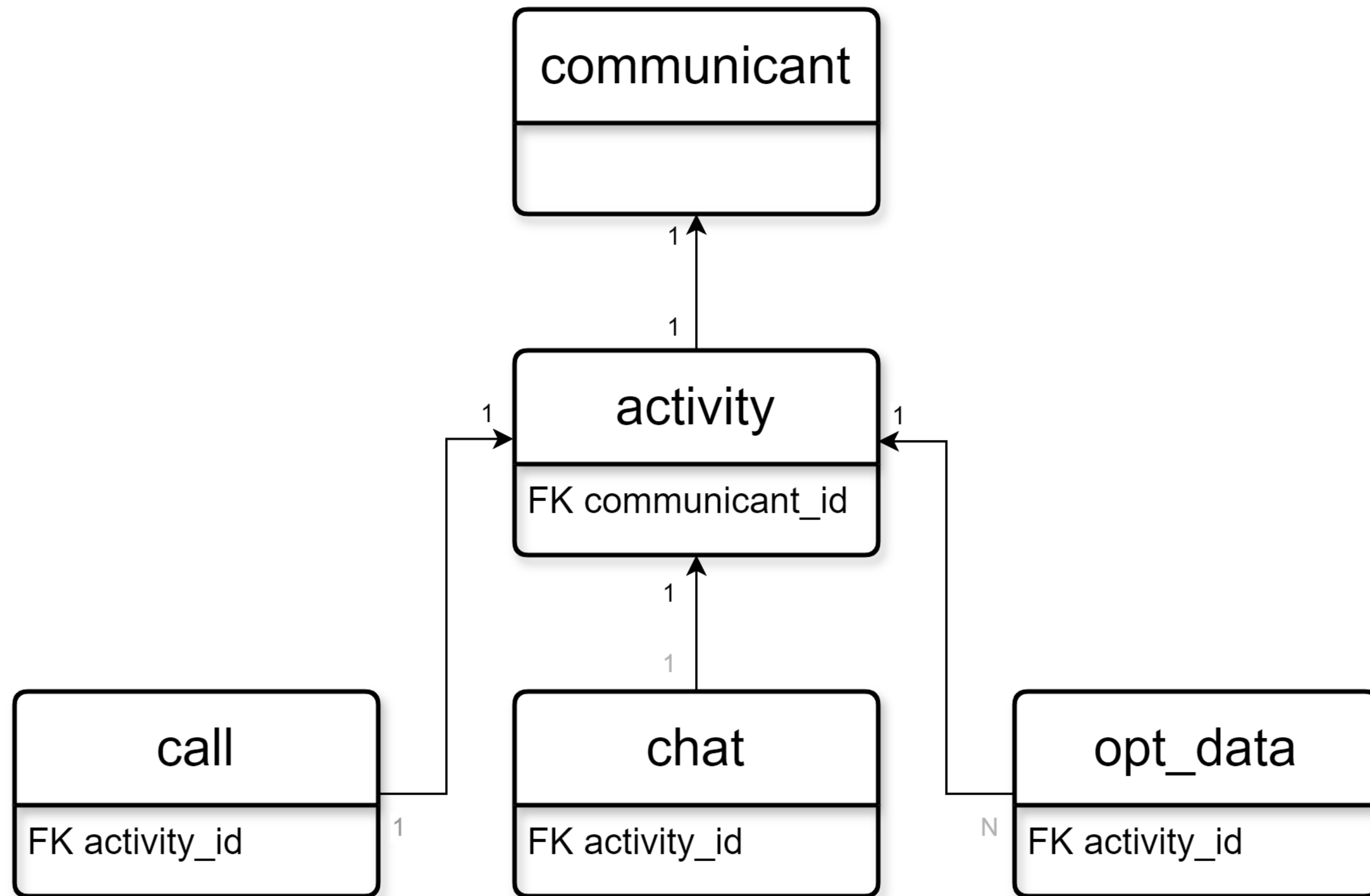
Назначение хранилища коммуникаций



Используемые технологии



Структура базы данных



Подготовка к будущим нагрузкам

Нагрузочные тесты

- 200 RPS
- БД: 1 CPU, 2GB mem, shared_buffers=512MB
- Сервис: 4 пода по 0.125CPU и 128MB mem.

01

03

Доработки

- Добавили недостающий индекс
- AsNoTracking
- PreCompiled LINQ

02

04

Первые результаты

Ошибки: 83%

Время ответа MAX: >60s

Время ответа p99: 30s

Время ответа p95: 22s

Время ответа p90: 15s

Итоговые результаты

Ошибки: 0.06%

Время ответа MAX: 60s

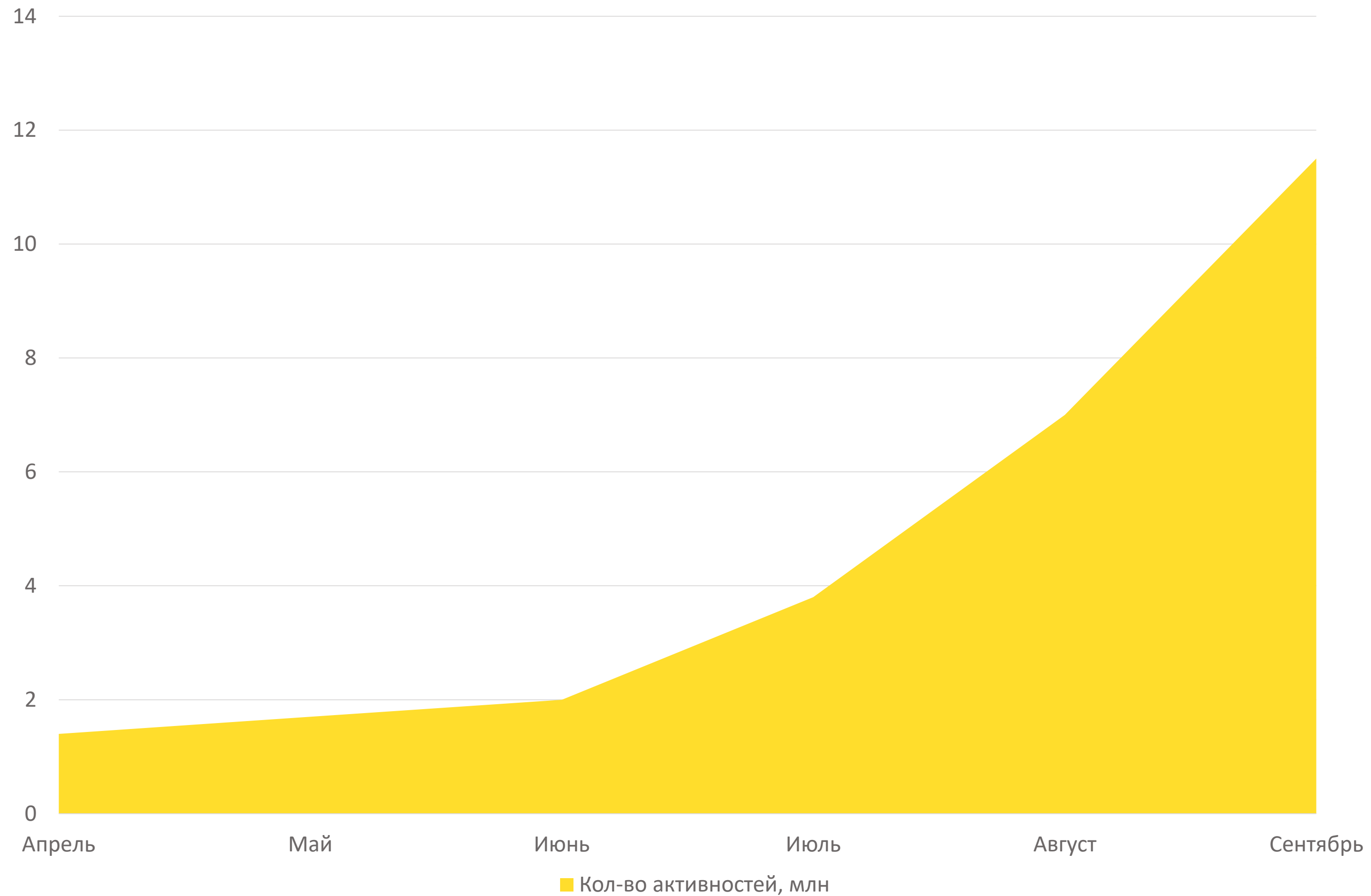
Время ответа p99: 460ms

Время ответа p95: 130ms

Время ответа p90: 90ms

Сбой и первые меры

Количество коммуникаций в БД



Сбой и его развитие



Первые алерты

Единичные ошибки таймаута выполнения запросов к БД



Рост количества ошибок

Таймауты получения подключения из пула, исчерпание лимита подключений в PG



Рост времени ответа

Время ответа GET-запроса выросло до **10 секунд и более**



Невозможность пересчёта

Система пересчёта не успевала обрабатывать всех клиентов из-за высокого времени получения истории взаимодействий

Неоптимальный индекс

- Фильтр: **client_id = '...' AND created_dt >= '90d ago'**
- Индексы: **client_id** и **created_dt**
- Для фильтрации по нескольким условиям, объединённым операцией **AND**, стоит использовать **составной индекс**
- В нашем случае необходим составной индекс **activity (client_id, created_dt)**

Неоптимальные запросы

- **Два LINQ-запроса** – отдельно запрашивались звонки, отдельно чаты
- В обоих запросах установлена опция **AsSplitQuery** – значит, EF мог разбить каждый из LINQ-запросов на несколько SQL

```
var calls = await dbContext.Calls
    .AsNoTracking()
    .AsSplitQuery()
    .Include(c => c.Activity)
    .Include(c => c.Activity.Communicant)
    .Include(c => c.Activity.OptData)
    .Where(c => c.Activity.ClientId == clientId)
    .Where(c => c.Activity.CreatedDt >= startDt)
    .ToListAsync();
```

```
var chats = await dbContext.Chats
    .AsNoTracking()
    .AsSplitQuery()
    .Include(c => c.Activity)
    .Include(c => c.Activity.Communicant)
    .Include(c => c.Activity.OptData)
    .Where(c => c.Activity.ClientId == clientId)
    .Where(c => c.Activity.CreatedDt >= startDt)
    .ToListAsync();
```

Неоптимальные запросы

На каждый вызов GET-метода получали несколько запросов с одной с одной и той же работой

```
-- Запрос звонков
SELECT c.*, a.*, c0.*
FROM "call" AS c
INNER JOIN activity AS a ON c.activity_id = a.id
INNER JOIN communicant AS c0 ON a.communicant_id = c0.id
WHERE
    a.client_id = @__clientId_0 AND
    a.created_dt >= @__startDt_1
ORDER BY c.id, a.id, c0.id
```

```
-- Запрос чатов
SELECT c.*, a.*, c0.*
FROM chat AS c
INNER JOIN activity AS a ON c.activity_id = a.id
INNER JOIN communicant AS c0 ON a.communicant_id = c0.id
WHERE
    a.client_id = @__clientId_0 AND
    a.created_dt >= @__startDt_1
ORDER BY c.id, a.id, c0.id
```

```
-- Запрос opt_data для звонков
SELECT o.*, c.id, c0.id, c1.id
FROM "call" AS c
INNER JOIN activity AS c0 ON c.activity_id = c0.id
INNER JOIN communicant AS c1 ON c0.communicant_id = c1.id
INNER JOIN "opt_data" AS o ON c0.id = o.activity_id
WHERE
    c0.client_id = @__customerId_0 AND
    c0.created_dt >= @__startDate_1
ORDER BY c.id, c0.id, c1.id
```

```
-- Запрос opt_data для чатов
SELECT o.*, c.id, c0.id, c1.id
FROM chat AS c
INNER JOIN activity AS c0 ON c.activity_id = c0.id
INNER JOIN communicant AS c1 ON c0.communicant_id = c1.id
INNER JOIN "opt_data" AS o ON c0.id = o.activity_id
WHERE
    c0.client_id = @__customerId_0 AND
    c0.created_dt >= @__startDate_1
ORDER BY c.id, c0.id, c1.id
```

Получение данных за один запрос

- Цель: **не выполнять** при получении данных **одну и ту же работу несколько раз**
- Делаем **единственный LINQ-запрос** к activity, присоединяя все связанные сущности
- **Не используем AsSplitQuery**

```
var activities = await dbContext.Activities
    .AsNoTracking()
    .Where(x => x.ClientId == clientId)
    .Where(x => x.CreatedDt >= startDt)
    .Include(x => x.Communicant)
    .Include(x => x.Calls)
    .Include(x => x.Chats)
    .Include(x => x.OptData)
    .ToListAsync();
```

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON a.communicant_id = c0.id
LEFT JOIN "call" AS c1 ON a.id = c1.activity_id
LEFT JOIN chat AS c2 ON a.id = c2.activity_id
LEFT JOIN opt_data AS o ON a.id = o.activity_id
WHERE a.client_id = 'ID9999' AND
      a.created_dt >= '2023-03-01T00:00:00' AND
ORDER BY a.id, c0.id, c1.id, c2.id
```


План для единственного запроса

Отбор по CustomerId и CreatedDate теперь выполняется один раз

```
-> Nested Loop Left Join (cost=2.28..988.40 rows=26 width=859)
  -> Nested Loop Left Join (cost=1.86..768.63 rows=26 width=816)
    -> Nested Loop Left Join (cost=1.43..548.87 rows=26 width=761)
      -> Nested Loop (cost=1.00..328.84 rows=26 width=508)
        -> Index Scan using idx_activity_client_id on activity a
            (cost=0.56..109.08 rows=26 width=392)
            Index Cond: (client_id = 'ID99999'::text)
            Filter: (created_dt >= '2023-03-01 00:00:00'::timestamp)
        -> Index Scan using communicant_pkey on communicant c0 (cost=0.43..8.45 rows=1 width=116)
            Index Cond: (id = c.communicant_id)
        -> Index Scan using idx_call_activity_id on "call" c1 (cost=0.43..8.45 rows=1 width=253)
            Index Cond: (activity_id = c.id)
        -> Index Scan using idx_chat_activity_id on chat c2 (cost=0.42..8.44 rows=1 width=55)
            Index Cond: (activity_id = c.id)
      -> Index Scan using idx_opt_data_activity_id on opt_data o (cost=0.42..8.44 rows=1 width=43)
            Index Cond: (activity_id = c.id)
```

Выполняется 1 раз
вместо 2-4

План для единственного запроса

Index Scan-ы для связанных сущностей тоже выполняются по одному разу

```
-> Nested Loop Left Join (cost=2.28..988.40 rows=26 width=859)
  -> Nested Loop Left Join (cost=1.86..768.63 rows=26 width=816)
    -> Nested Loop Left Join (cost=1.43..548.87 rows=26 width=761)
      -> Nested Loop (cost=1.00..328.84 rows=26 width=508)
        -> Index Scan using idx_activity_client_id on activity a
Index Scan других таблиц (cost=0.56..109.08 rows=26 width=392)
по 1 разу
      Index Cond: (client_id = 'ID99999'::text)
      Filter: (created dt >= '2023-03-01 00:00:00'::timestamp)
        -> Index Scan using communicant_pkey on communicant c0 (cost=0.43..8.45 rows=1 width=116)
          Index Cond: (id = c.communicant_id)
        -> Index Scan using idx_call_activity_id on "call" c1 (cost=0.43..8.45 rows=1 width=253)
          Index Cond: (activity_id = c.id)
        -> Index Scan using idx_chat_activity_id on chat c2 (cost=0.42..8.44 rows=1 width=55)
          Index Cond: (activity_id = c.id)
      -> Index Scan using idx_opt_data_activity_id on opt_data o (cost=0.42..8.44 rows=1 width=43)
        Index Cond: (activity_id = c.id)
```

План для единственного запроса

Для поиска в Calls и Chats оба раза используется полный список activity_id

```
-> Nested Loop Left Join (cost=2.28..988.40 rows=26 width=859)
  -> Nested Loop Left Join (cost=1.86..768.63 rows=26 width=816)
    -> Nested Loop Left Join (cost=1.43..548.87 rows=26 width=761)
      -> Nested Loop (cost=1.00..328.84 rows=26 width=508)
        -> Index Scan using idx_activity_client_id on activity a
            (cost=0.56..109.08 rows=26 width=392)
            Index Cond: (client_id = 'ID99999'::text)
            Filter: (created_dt >= '2023-03-01 00:00:00'::timestamp)
        -> Index Scan using communicant_pkey on communicant c0 (cost=0.43..8.45 rows=1 width=116)
            Index Cond: (id = c.communicant_id)
          -> Index Scan using idx_call_activity_id on "call" c1 (cost=0.43..8.45 rows=1 width=253)
              Index Cond: (activity_id = c.id)
        -> Index Scan using idx_chat_activity_id on chat c2 (cost=0.42..8.44 rows=1 width=55)
            Index Cond: (activity_id = c.id)
      -> Index Scan using idx_opt_data_activity_id on opt_data o (cost=0.42..8.44 rows=1 width=43)
          Index Cond: (activity_id = c.id)
```

Не учитываем a.activity_type

План для единственного запроса

Без SplitQuery JOIN таблицы opt_data может сильно раздуть результат запроса

```
-> Nested Loop Left Join (cost=2.28..988.40 rows=26 width=859)
  -> Nested Loop Left Join (cost=1.86..768.63 rows=26 width=816)
    -> Nested Loop Left Join (cost=1.43..548.87 rows=26 width=761)
      -> Nested Loop (cost=1.00..328.84 rows=26 width=508)
        -> Index Scan using idx_activity_client_id on activity a
            (cost=0.56..109.08 rows=26 width=392)
            Index Cond: (client_id = 'ID99999'::text)
            Filter: (created_dt >= '2023-03-01 00:00:00'::timestamp)
        -> Index Scan using communicant_pkey on communicant c0 (cost=0.43..8.45 rows=1 width=116)
            Index Cond: (id = c.communicant_id)
        -> Index Scan using idx_call_activity_id on "call" c1 (cost=0.43..8.45 rows=1 width=253)
            Index Cond: (activity_id = c.id)
      -> Index Scan using idx_chat_activity_id on chat c2 (cost=0.42..8.44 rows=1 width=55)
            Index Cond: (activity_id = c.id)
    -> Index Scan using idx_opt_data_activity_id on opt_data o (cost=0.42..8.44 rows=1 width=43)
            Index Cond: (activity_id = c.id)
```

Раздуваем
результат
запроса

Первая Оптимизация



Составной индекс

Создать **составной индекс** по колонкам `client_id` и `created_dt`



Едиственный запрос

Под фича-флагом делаем режим получения данных за **один запрос**



Нагрузочные тесты

Проверяем, не станет ли хуже от принятых нововведений

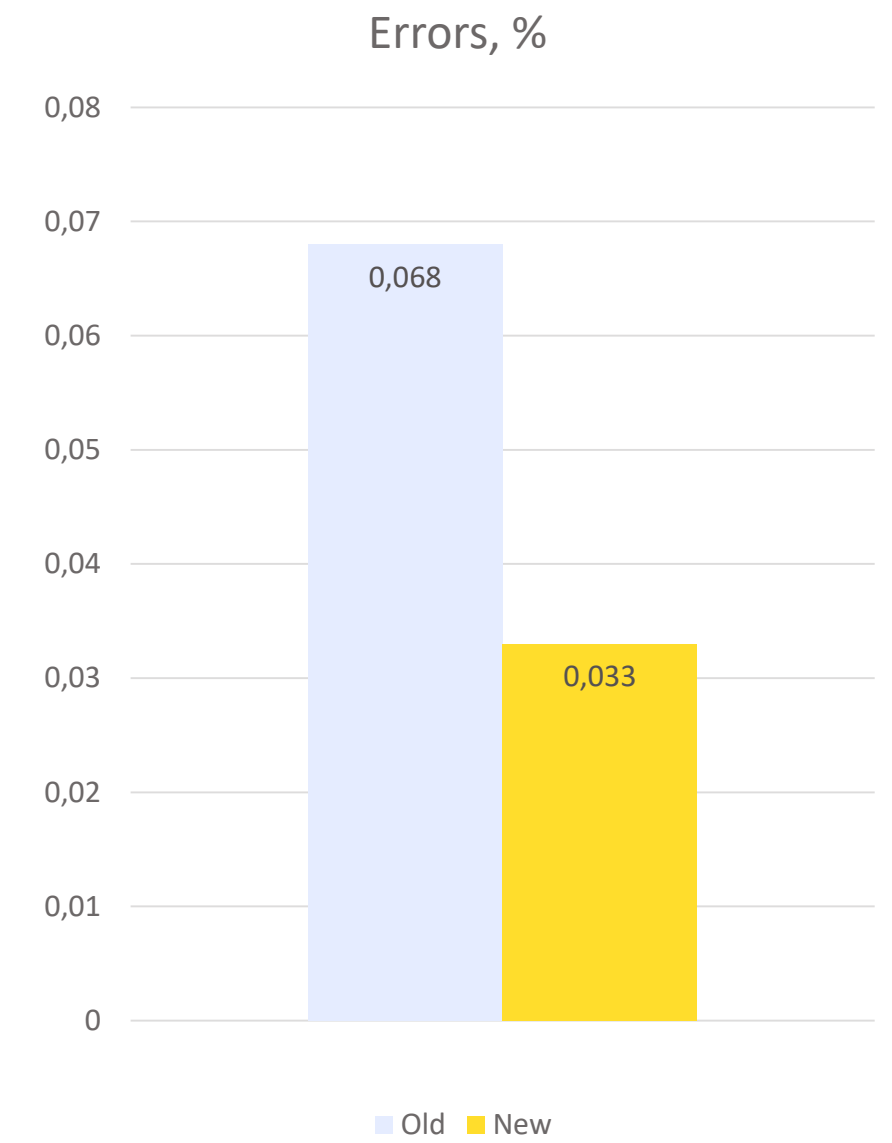
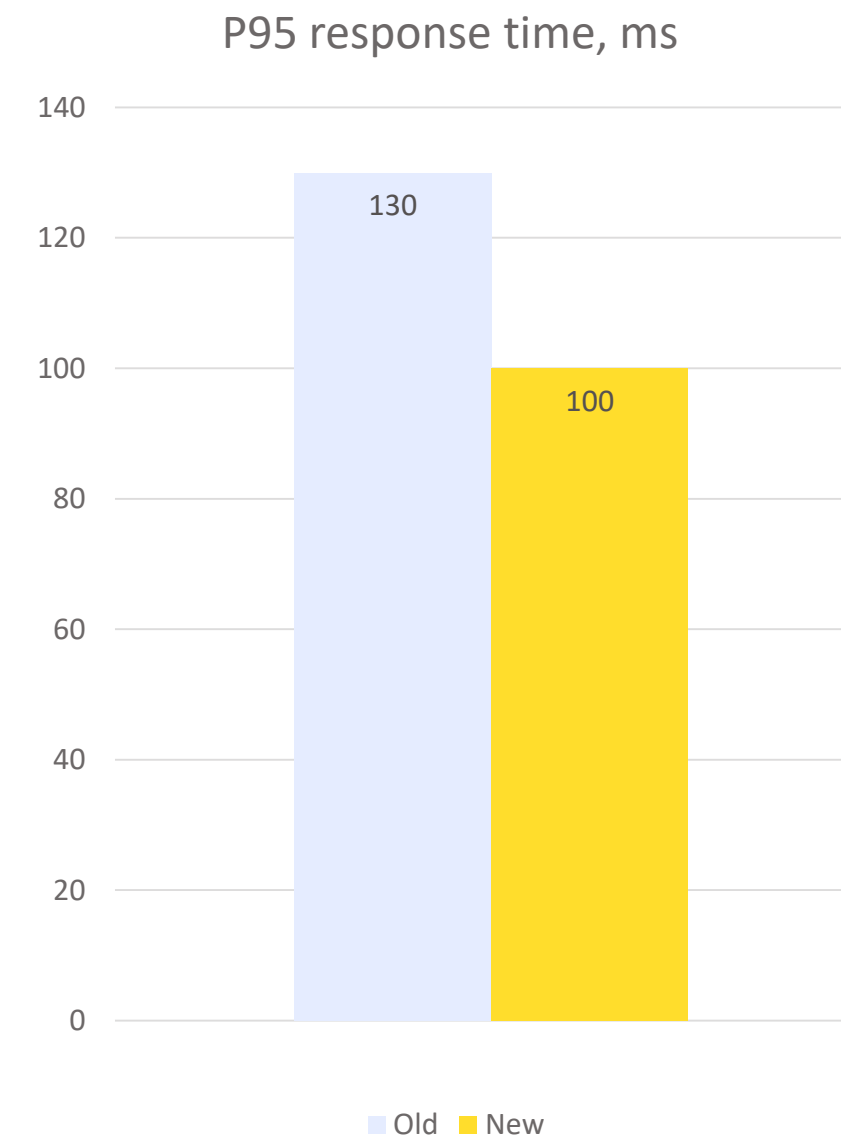
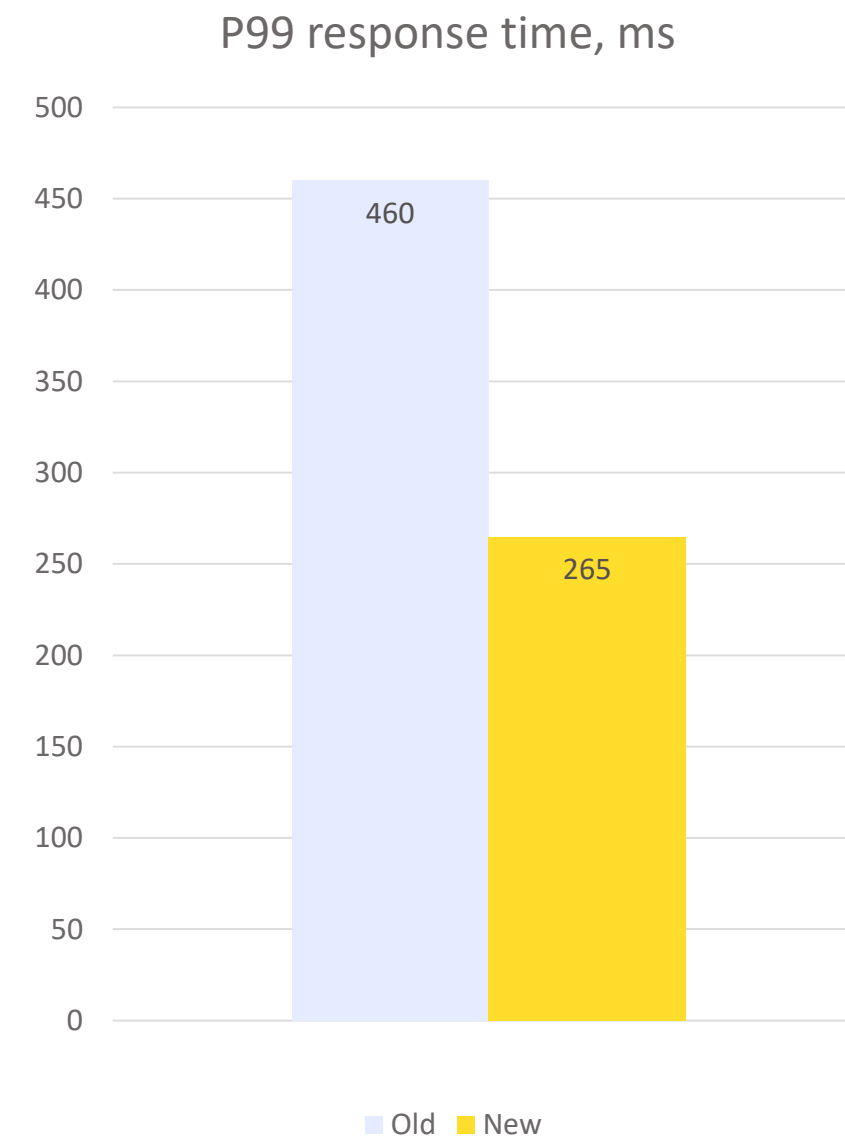
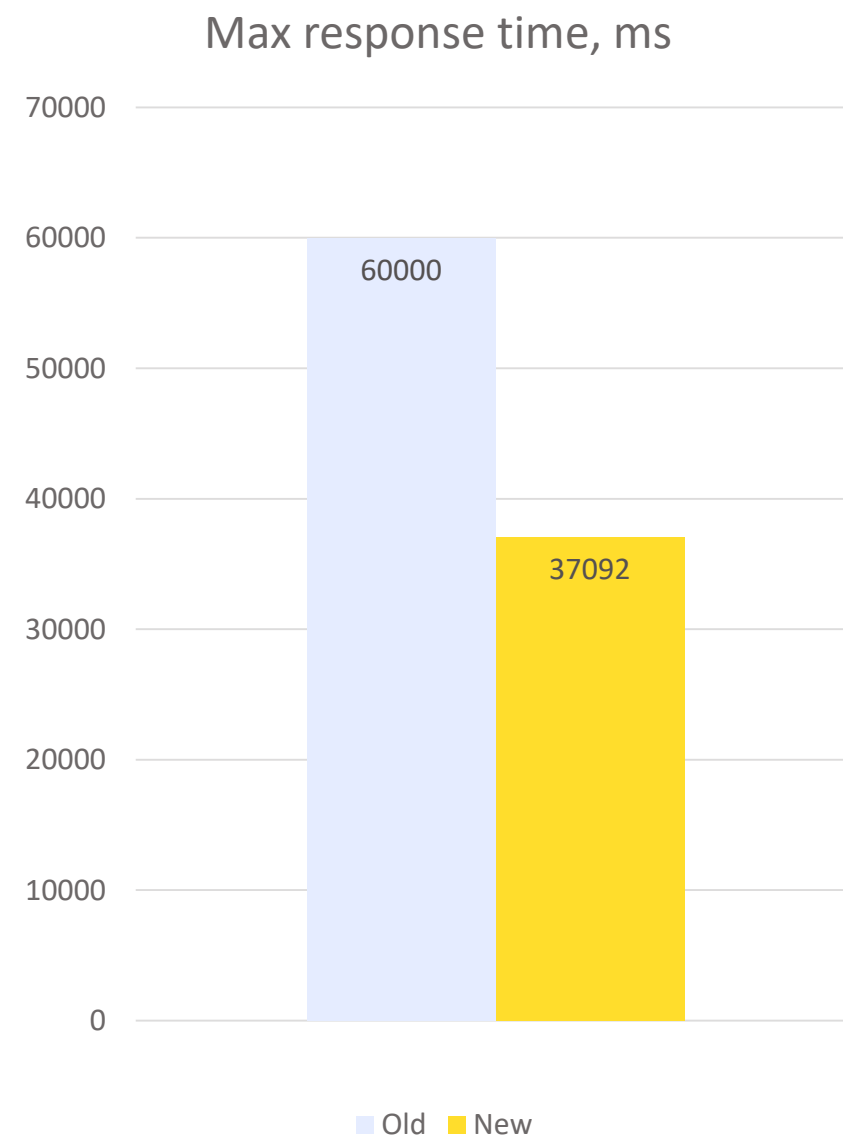


Проверка в проде

Релиз нововведений на `prod` и оценка результатов ночного пересчёта

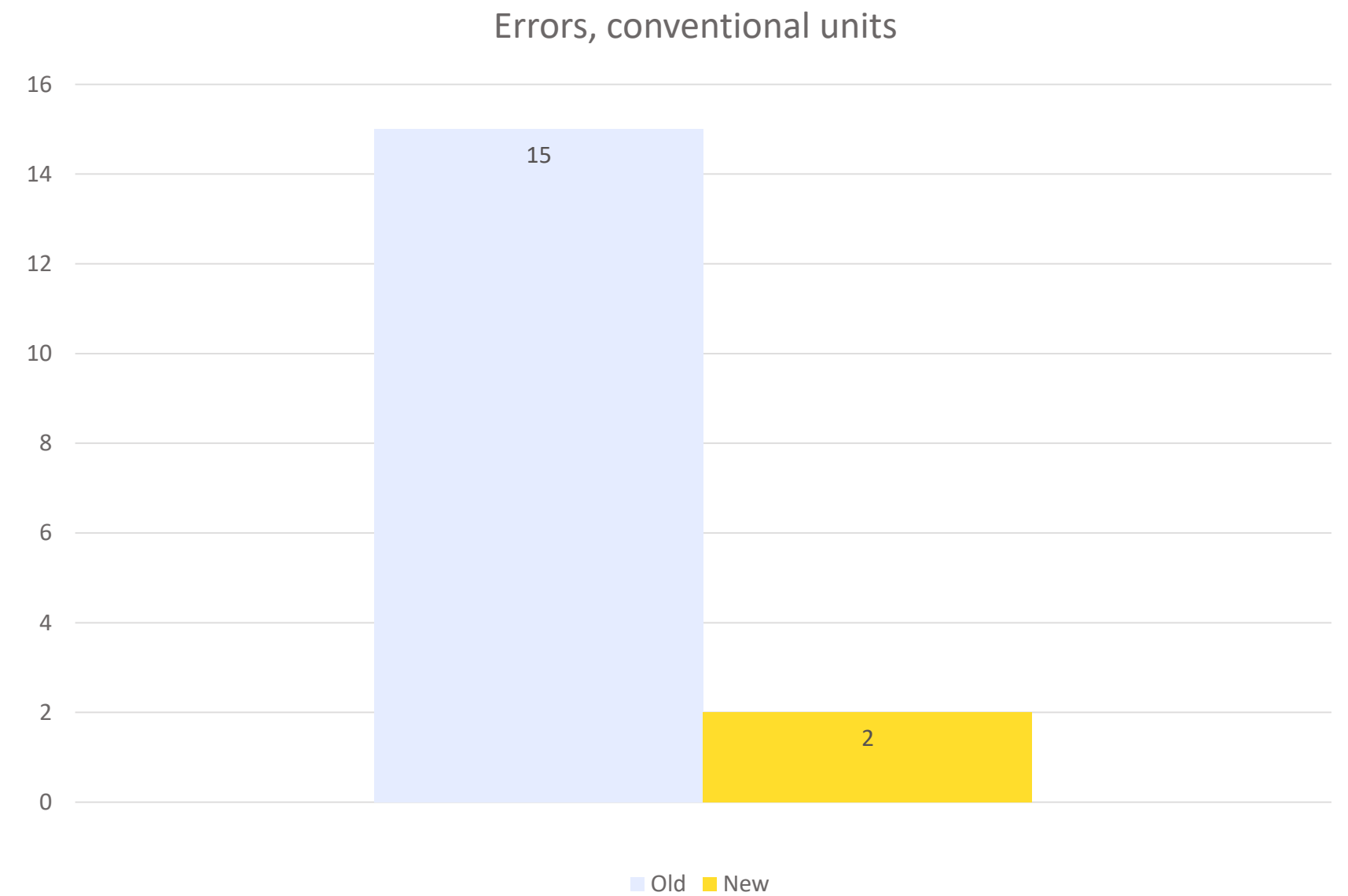
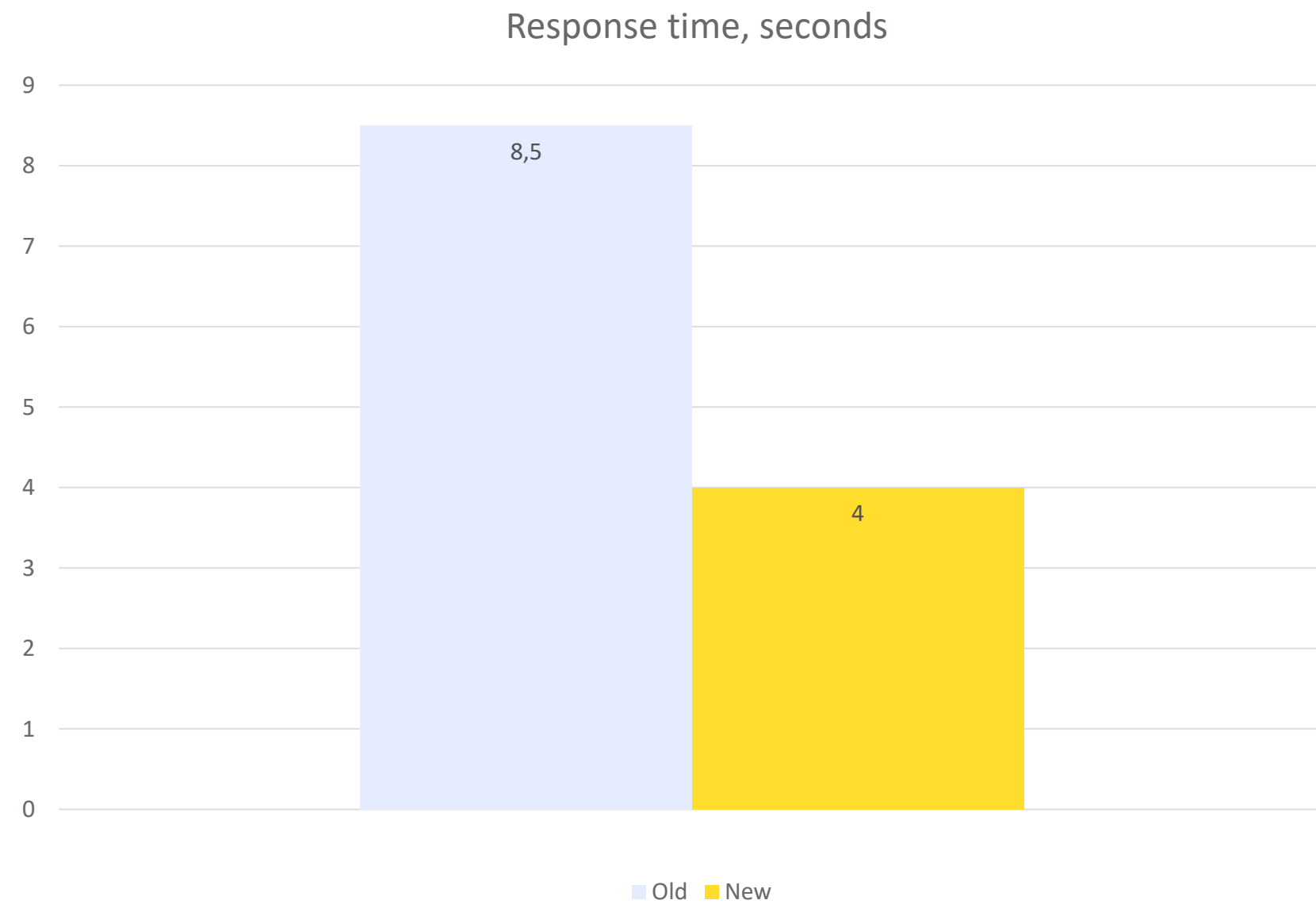
Составной индекс и получение данных за один запрос, результаты

Изменения в результатах нагрузочных тестов



Составной индекс и получение данных за один запрос, результаты

Изменения на production. Стало лучше, но нужно 300 ms



Дальнейший анализ и поиск решения

Суть проблемы



Долгое выполнение SQL-запросов

По метрикам БД: **очень много времени уходит на операции чтения с диска**



Много чтений с диска

- Запрос нередко считывает с диска **> 50 блоков**
- По правилам DBA это **недопустимо для OLTP базы**
- В идеале нужно стремиться к **3-4 блокам на запрос**



1. Каждый запрос
возвращает много строк



2. Неоптимальные типы
для колонок в таблицах



3. Излишняя
нормализация схемы БД



4. Нехватка ресурсов

Запрос возвращает много строк



История взаимодействий с клиентом за 90 дней составляла десятки активностей, в отдельных случаях запрос мог возвращать 100+ строк.



1. Каждый запрос возвращает много строк



2. Неоптимальные типы для колонок в таблицах



3. Излишняя нормализация схемы БД



4. Нехватка ресурсов

Запрос возвращает много строк



История взаимодействий с клиентом за 90 дней составляла десятки активностей, в отдельных случаях запрос мог возвращать 100+ строк.



Возможное решение: переработать контракт GET-метода, чтобы **добавить в него ограничение на размер выдачи** (пагинация).



1. Каждый запрос возвращает много строк



2. Неоптимальные типы для колонок в таблицах



3. Излишняя нормализация схемы БД



4. Нехватка ресурсов

Запрос возвращает много строк



История взаимодействий с клиентом за 90 дней составляла десятки активностей, в отдельных случаях запрос мог возвращать 100+ строк.



Возможное решение: переработать контракт GET-метода, чтобы **добавить в него ограничение на размер выдачи** (пагинация).




Ломает контракт API и требует привлечения ресурсов других команд – недоступная на тот момент для нас опция




Маловероятно что поможет: клиентская система будет просто делать несколько вызовов вместо одного, и **объём операций чтений с диска в совокупности останется прежний.**

Неоптимальные типы данных в БД

 1. Каждый запрос возвращает много строк

 2. Неоптимальные типы для колонок в таблицах


 3. Излишняя нормализация схемы БД

 4. Нехватка ресурсов





В таблицах есть **поля с типом TEXT**, при этом **множество возможных значений для них ограничено** несколькими десятками.

Неоптимальные типы данных в БД

 1. Каждый запрос возвращает много строк

 2. Неоптимальные типы для колонок в таблицах

 3. Излишняя нормализация схемы БД

 4. Нехватка ресурсов




В таблицах есть поля с типом TEXT, при этом множество возможных значений для них ограничено несколькими десятками.





Возможное решение: заменить на int (enum), чтобы **сократить объём чтений с диска.**

Неоптимальные типы данных в БД

 1. Каждый запрос возвращает много строк

 2. Неоптимальные типы для колонок в таблицах

 3. Излишняя нормализация схемы БД

 4. Нехватка ресурсов



В таблицах есть поля с типом TEXT, при этом множество возможных значений для них ограничено несколькими десятками.



Возможное решение: заменить на int (enum), чтобы **сократить объём чтений с диска**. При этом мы можем оставить контракт API прежним, выполняя маппинг на стороне .NET-сервиса.



Сломается интеграция с DWH, т.к. она завязана на схему данных. Потребуются работы на их стороне – недопустимый вариант.



Изменение схемы большой БД нетривиальная задача которая **потребует масштабных доработок и на нашей стороне**.

Излишняя нормализация схемы

➔ 1. Каждый запрос возвращает много строк


➔ 2. Неоптимальные типы для колонок в таблицах


➔ 3. Излишняя нормализация схемы БД

➔ 4. Нехватка ресурсов




Используется **5 таблиц**. При получении данных **в каждой таблице** необходимо выполнять Index Scan. **Расходуется память под индексы** в каждой таблице.

 1. Каждый запрос возвращает много строк

 2. Неоптимальные типы для колонок в таблицах

 **3. Излишняя нормализация схемы БД**

 4. Нехватка ресурсов


Излишняя нормализация схемы




Используется 5 таблиц. При получении данных в каждой таблице необходимо выполнять Index Scan. Расходуется память под индексы в каждой таблице.




Возможное решение: хранить активности **в одной таблице**. Тогда для GET-запроса будет нужен только **один Index Scan**, память нужна только под **один индекс**.

 1. Каждый запрос возвращает много строк

 2. Неоптимальные типы для колонок в таблицах

 **3. Излишняя нормализация схемы БД**

 4. Нехватка ресурсов

Излишняя нормализация схемы



Используется 5 таблиц. При получении данных в каждой таблице необходимо выполнять Index Scan. Расходуется память под индексы в каждой таблице.



Возможное решение: хранить активности в одной таблице. Тогда для GET-запроса будет нужен только **один Index Scan**, память нужна только под **один индекс**.



Нереализуемо по тем же причинам: **зависимость DWH от нашей схемы** данных, **сложность доработок** на нашей стороне, **сложность миграции** от старой схемы к новой.

Нехватка ресурсов сервера БД

➔ 1. Каждый запрос возвращает много строк

➔ 2. Неоптимальные типы для колонок в таблицах


➔ 3. Излишняя нормализация схемы БД


➔ 4. Нехватка ресурсов


i Используется инстанс c1r2. Всего **2ГБ** ОЗУ, из них **512МБ под shared_buffers**. Этого объёма не хватает под необходимые индексы.


Индекс	Объём
activity (client_id, created_dt)	633 МБ
communicants (id)	330 МБ
call (activity_id)	306 МБ
chat (activity_id)	23 МБ
opt_data (activity_id)	58 МБ
Всего	1350 МБ


Нехватка ресурсов сервера БД


 1. Каждый запрос возвращает много строк


 2. Неоптимальные типы для колонок в таблицах

 3. Излишняя нормализация схемы БД

 4. Нехватка ресурсов

 Используется инстанс c1r2. Всего **2ГБ** ОЗУ, из них **512МБ под shared_buffers**. Этого объёма не хватает под необходимые индексы.

 Возможное решение: запросить более мощный инстанс. Мы могли рассчитывать на **c2r4** с увеличенным **shared_buffers = 2ГБ**.

 На согласование потребуется какое-то время. И в будущем мы обязательно **снова столкнёмся с этой проблемой уже на новом сервере**, а бесконечно увеличивать ресурсы нам не будут.



**А что если поискать решение
на уровне бизнес-требований?**

Изменение логики GET-метода

- Почти **90%** коммуникаций составляли неудавшиеся звонки (**Автоответчики**).
- Автоответчики **не участвуют** в алгоритмах расчётов потребителя данных.
- Значит они могут быть **исключены из выдачи GET-метода** и из БД их тогда тоже тянуть не нужно.

```
var activities = await dbContext.Activities
    .AsNoTracking()
    .Where(x => x.ClientId == clientId)
    .Where(x => x.CreatedDt >= startDt)
    .Where(x => x.Result != "Автоответчик")
    .Include(x => x.Communicant)
    .Include(x => x.Calls)
    .Include(x => x.Chats)
    .Include(x => x.OptData)
    .ToListAsync();
```

План запроса с исключением Автоответчиков

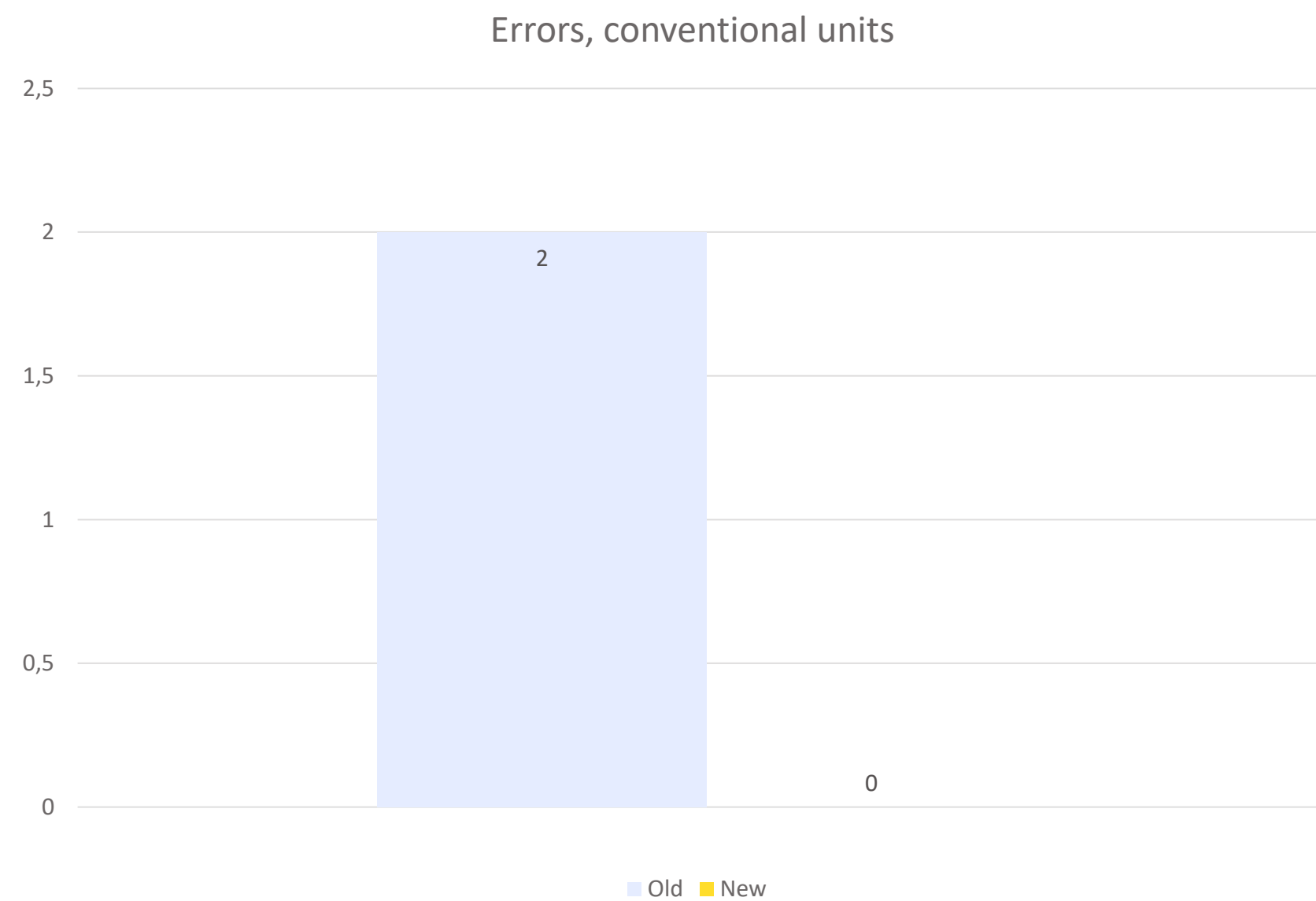
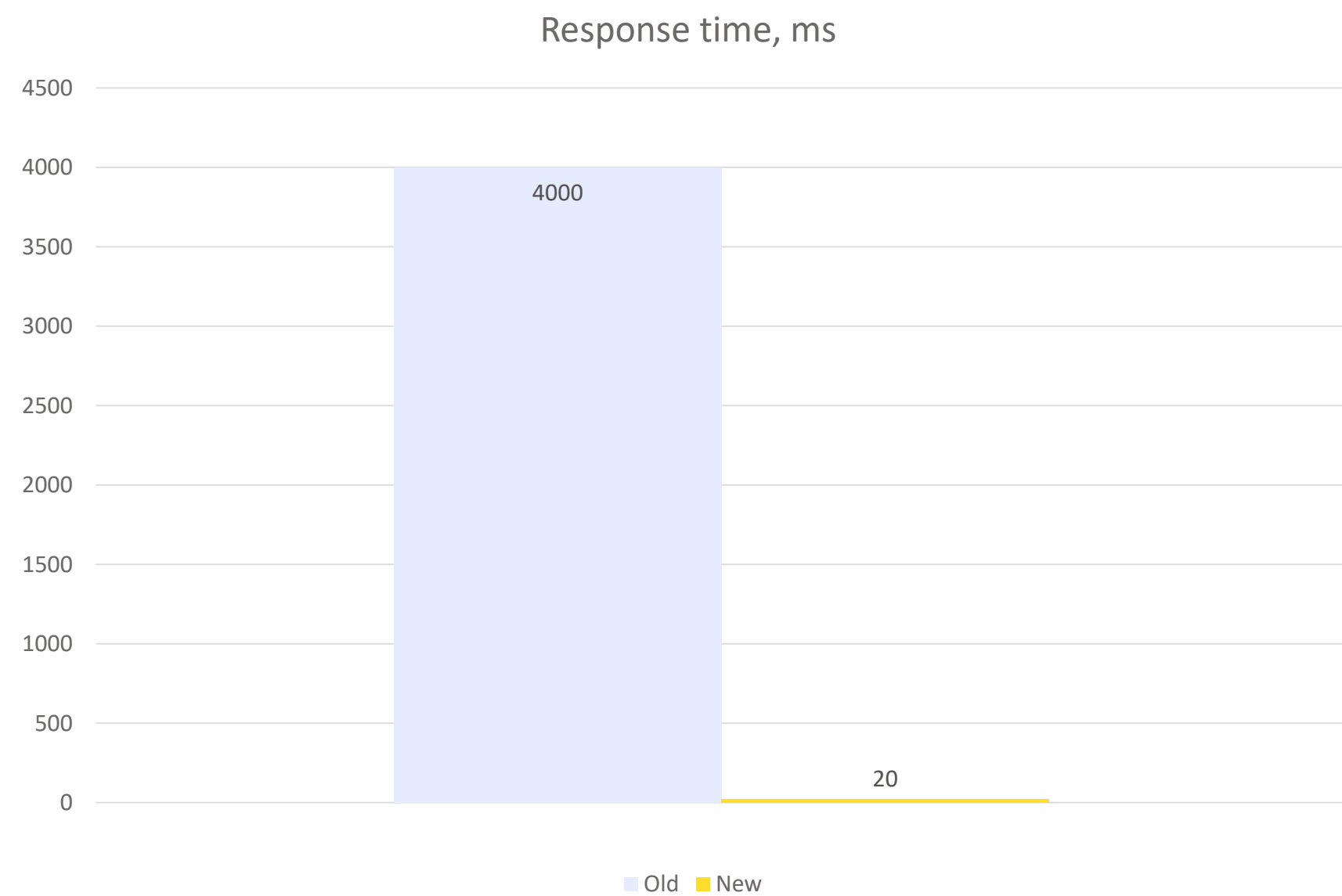
Внутренний Index Scan выдаст на 90% меньше строк, все остальные операции сократятся так же

```
-> Nested Loop Left Join (cost=2.28..379.70 rows=8 width=859)
  -> Nested Loop Left Join (cost=1.86..312.08 rows=8 width=816)
    -> Nested Loop Left Join (cost=1.43..244.46 rows=8 width=761)
      -> Nested Loop (cost=1.00..176.76 rows=8 width=508)
        -> Index Scan using idx_activity_client_id_created_dt on activity c
            (cost=0.56..109.14 rows=8 width=392)
            Index Cond: ((client_id = 'ID99999'::text)
            AND (created_dt >= '2023-03-01 00:00:00'::timestamp without time zone))
            Filter: (result <> 'Автоответчик'::text)
        -> Index Scan using communicant_pkey on communicant c0 (cost=0.43..8.45 rows=1 width=116)
            Index Cond: (id = c.communicant_id)
        -> Index Scan using idx_call_activity_id on "call" c1 (cost=0.43..8.45 rows=1 width=253)
            Index Cond: (activity_id = c.id)
        -> Index Scan using idx_chat_activity_id on chat c2 (cost=0.42..8.44 rows=1 width=55)
            Index Cond: (activity_id = c.id)
      -> Index Scan using idx_opt_data_activity_id on opt_data o (cost=0.42..8.44 rows=1 width=43)
            Index Cond: (activity_id = c.id)
```

Возвращает на ~90% меньше данных, которые идут на вход 4-м другим Index Scan

Фильтрация автоответчиков при получении коммуникаций из БД

Результаты на production. Проблема решена



Решение проблемы будущего роста

Оптимизация под будущий рост



Рост нагрузки

БД растёт с ускорением, доля автоответчиков в будущем снизится



Экономия и стабилизация

Задача: уменьшить размер активно используемых таблиц и индексов и стабилизировать его во времени



Данные старше 90 дней

Не нужны потребителю данных. Если их исключить из обработки, то можно сэкономить ресурсы



Автоответчики

Не нужны потребителю данных. Тоже кандидат на исключение из обработки

Удаление данных старше 90 дней



Дополнительная нагрузка

Запрос DELETE на большой таблице создаёт дополнительную нагрузку на БД



Не уменьшается объём данных

DELETE не уменьшает размер индексов и таблиц, место может быть переиспользовано после VACUUM



Нерекомендуемый метод

DBA не рекомендуют использовать DELETE для удаления неактуальных данных на больших таблицах



Сохранность исторических данных

DWH не даёт 100% гарантии по сохранности данных, у них должна быть возможность загрузить их от нас заново

Исключение автоответчиков из обработки



Не сохранять не можем

Автоответчики нужны в DWH аналитикам, а в DWH они могут попасть только из нашей БД



Другая таблица потребует изменений в DWH

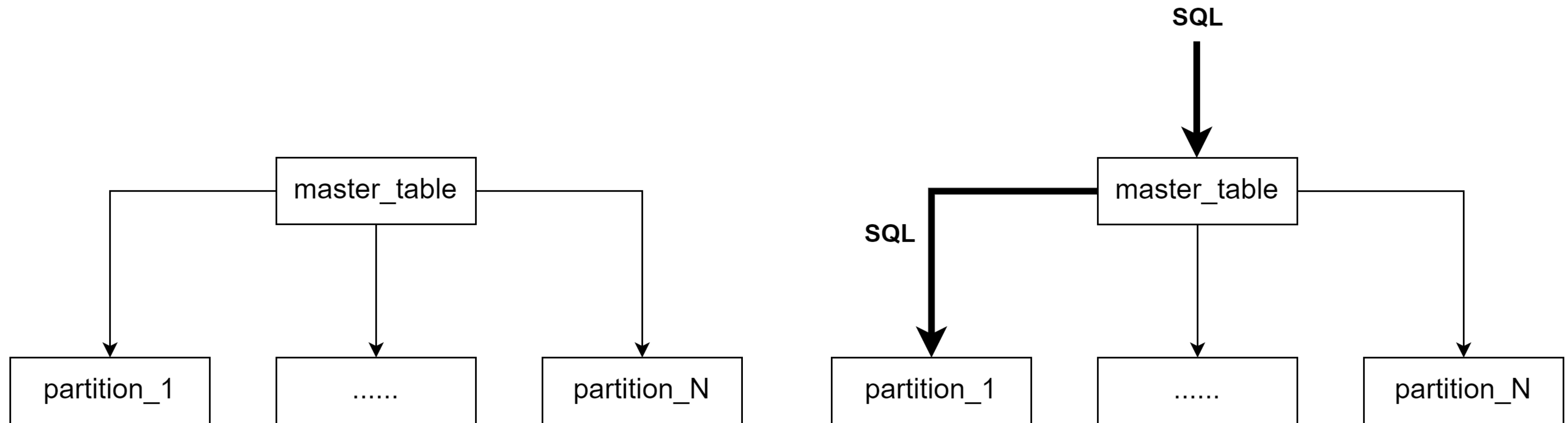
Механизмы репликации завязаны на нашу схему, поэтому без доработок на стороне DWH такой вариант невозможен



**Как вынести часть данных
в другую таблицу
чтобы для стороннего наблюдателя
они остались там же?**

Партицирование

- Множество строк одной таблицы (мастер-таблицы) разбивается на **непересекающиеся подмножества**, каждое из которых хранится **в отдельной таблице** (партиции).
- Для внешнего наблюдателя **выглядит как одна таблица**, к ней можно обращаться в SQL, фактически запрос будет выполнен на одной или нескольких партициях.



Создание партицированной таблицы

- **CREATE TABLE** с инструкцией **PARTITION OF**, после которой нужно указать **ключ** и **метод** партицирования
- Ключ – колонка или выражение
- Метод – **Range / List / Hash**. Указывает на **способ задать множество значений ключа**, которые соответствуют конкретной партиции

```
CREATE TABLE master_table (  
    id,  
    created_dt,  
    ...  
) PARTITION BY RANGE (created_dt);
```

МЕТОД ПАРТИЦИРОВАНИЯ

КЛЮЧ ПАРТИЦИРОВАНИЯ

Создание партиций для метода RANGE

- В партициях в соответствии с ключам и методом необходимо указать **подмножество соответствующих значений ключа**
- Для метода **RANGE** – это **интервал** закрытый слева и открытый справа

```
-- Партицирование по диапазону
CREATE TABLE master_table (
    id,
    created_dt,
    ...
) PARTITION BY RANGE (created_dt);
```

```
CREATE TABLE partition_january
PARTITION OF master_table
FOR VALUES
    FROM ('2024-01-01T00:00:00')
    TO ('2024-02-01T00:00:00');
```


Создание партиций для метода LIST

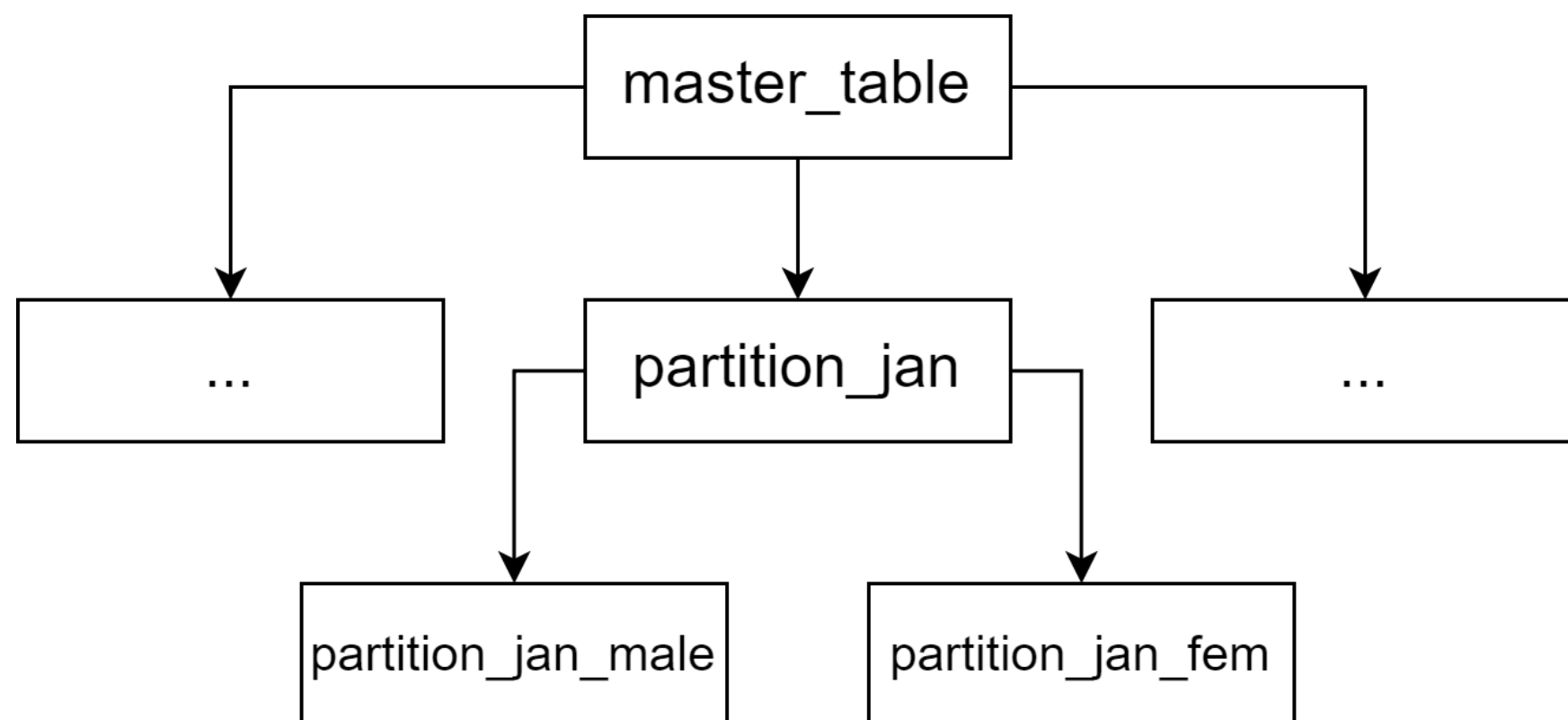
- В партициях в соответствии с ключам и методом необходимо указать **подмножество соответствующих значений ключа**
- Для метода **LIST** – это **перечисление** всех соответствующих значений

```
-- Партиционирование по списку значений
CREATE TABLE master_table (
    id,
    created_dt,
    ...
)
PARTITION BY
LIST (TO_CHAR(created_dt, 'DAY'));

CREATE TABLE partition_weekends
PARTITION OF master_table
FOR VALUES ('SATURDAY', 'SUNDAY');
```

Многоуровневое партицирование

- Партиции в свою очередь тоже могут быть партицированы
- В полученном дереве таблицами с данными будут только листья нижнего уровня



```
-- Список людей
-- Партицирован по дате рождения
CREATE TABLE master_table (
  id,
  birthday,
  sex
) PARTITION BY RANGE (birthday);

-- Партиция для рождённых в январе
CREATE TABLE partition_january
PARTITION OF master_table
FOR VALUES FROM('2024-01-01T00:00:00') TO ('2024-02-01T00:00:00')
PARTITION BY LIST (sex);

-- Партиция для рождённых в январе мальчиков
CREATE TABLE partition_january_male
PARTITION OF partition_january FOR VALUES ('male');

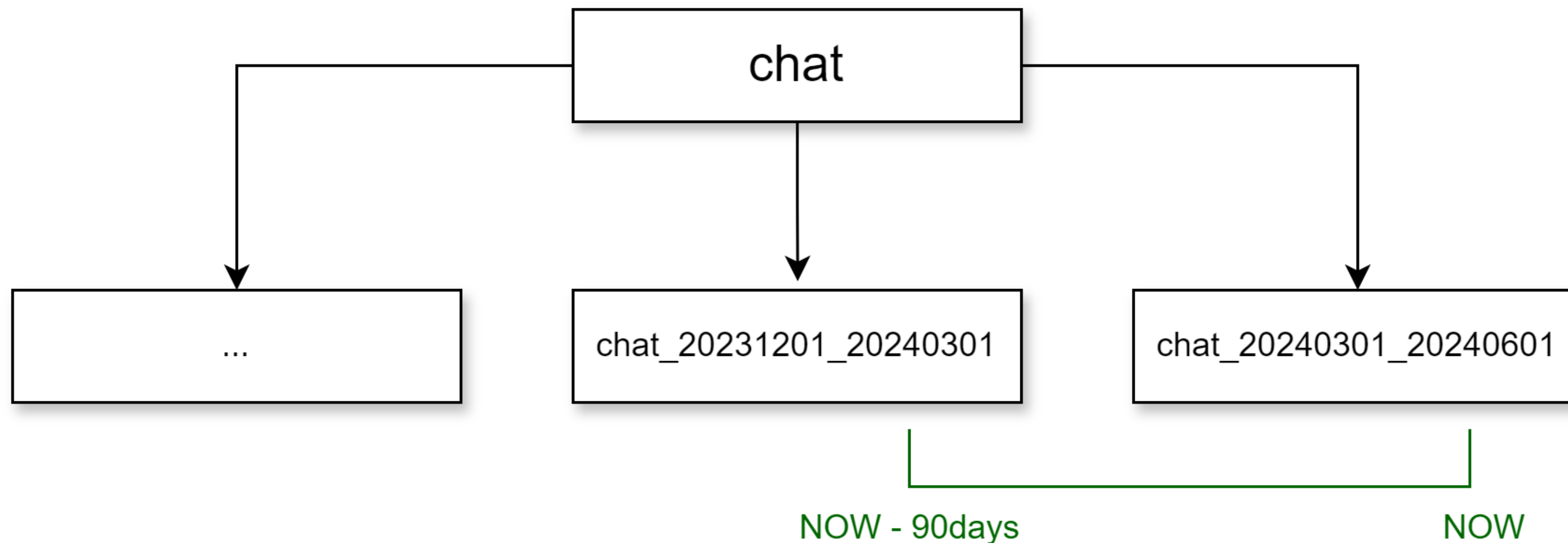
-- Партиция для рождённых в январе девочек
CREATE TABLE partition_january_female
PARTITION OF partition_january FOR VALUES ('female');
```

Другие факты о партицировании

- **DDL** запросы на мастер-таблице выполняются на всех партициях
- При **INSERT** должна быть возможность **однозначно определить подходящую партицию**. Если это невозможно, то INSERT завершится с ошибкой
- Если **UPDATE** меняет ключ партицирования, то запись будет перемещена в другую партицию. Поведение аналогично INSERT
- **SELECT / UPDATE / DELETE** выполняются только в тех партициях, в которых могут быть данные согласно фильтрам в запросе, и не выполняются в тех партициях, в которых данных заведомо нет
- В партицированную таблицу можно **добавить существующую таблицу как партицию (ATTACH)** или наоборот сделать партицию независимой таблицей (**DETACH**)

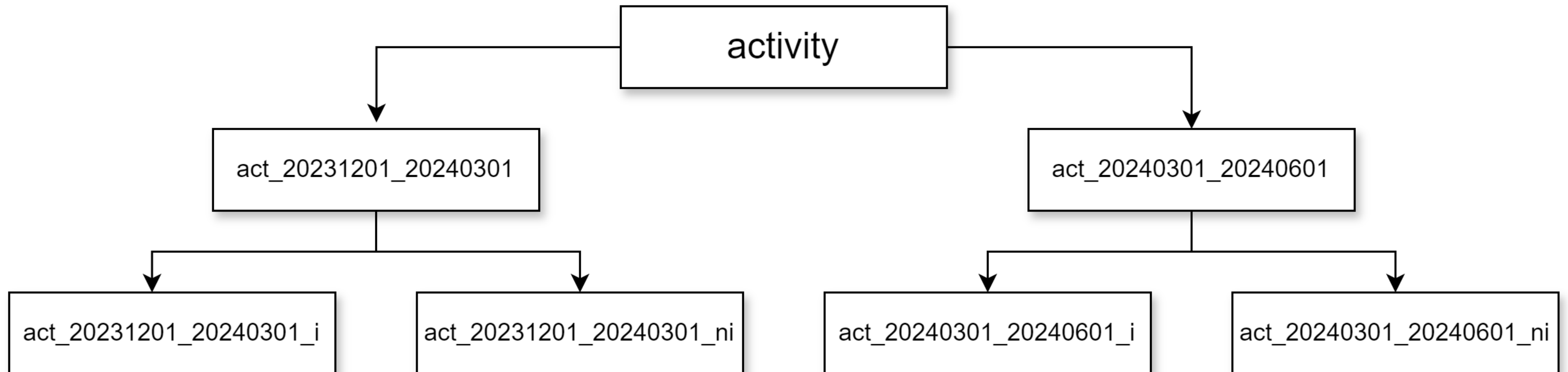
Партицирование chat и opt_data

- Таблицы chat и opt_data **гарантированно не содержат записей, относящихся к Автоответчикам**
- Поэтому для них нужно только решить проблему удаления старых записей, для чего подходит **партицирование по дате на трёхмесячные партии**



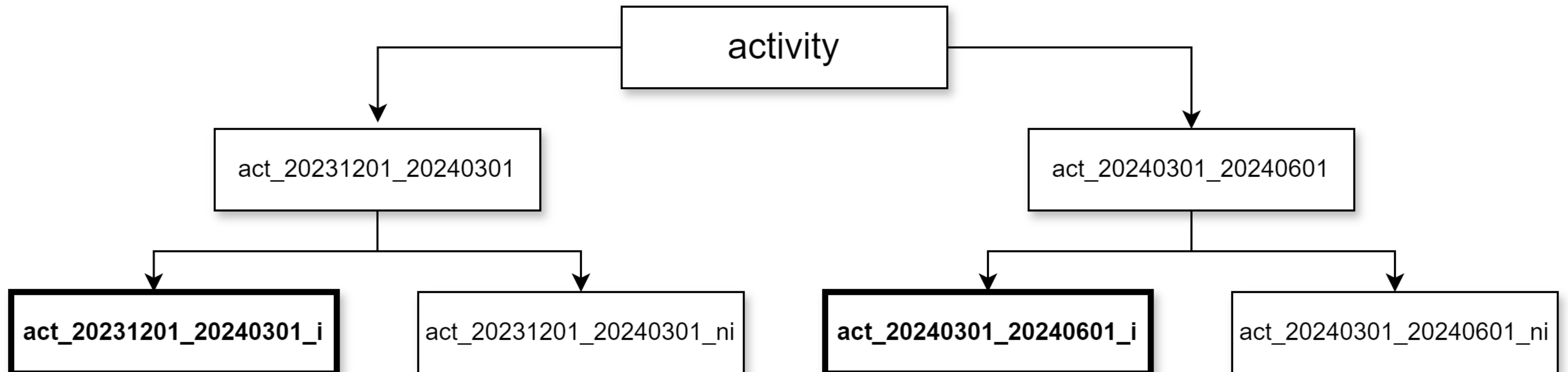
Двухуровневое партицирование

- Таблицы activity, communicant и call **могут содержать строки, относящиеся к Автоответчикам**
- Чтобы исключить такие строки из обработки мы можем **каждую трёхмесячную партицию дополнительно разделить на две партиции: *важные* для расчётов данные и *неважные* (автоответчики)**



Двухуровневое партиционирование

- Фактически запрос получения данных будет работать **только на 1-2 таблицах, содержащие меньшую часть от общего объёма данных**
- Индексы этих таблиц тоже занимают меньшую долю от объема индексов для всех данных, следовательно **они намного лучше будут влезать в кэш**



Партицирование по типу результата

- result – колонка с типом TEXT.
- **PARTITION BY RANGE** – не имеет смысла.
- **PARTITION BY LIST** – потребует указывать **все возможные значения**. Это возможно, т.к. множество всех значений result ограничено и известно нам.

```
CREATE TABLE activity_20231201_20240301
PARTITION OF activity
FOR VALUES FROM ('2023-12-01') TO ('2024-03-01')
PARTITION BY LIST (result);
```

```
-- Для партиции ni указываем Автоответчик
CREATE TABLE activity_20231201_20240301_ni
PARTITION OF activity_20231201_20240301
FOR VALUES ('Автоответчик');
```

```
-- Перечисляем всё кроме автоответчиков
CREATE TABLE activity_20231201_20240301_i
PARTITION OF activity_20231201_20240301
FOR VALUES ('Результат 1', ..., 'Результат N');
```

Партицирование по типу результата

- result – колонка с типом TEXT.
- **PARTITION BY RANGE – не имеет смысла.**
- **PARTITION BY LIST –** потребует указывать **все возможные значения.** Это возможно, т.к. множество всех значений result ограничено и известно нам.
- **Если появится новый тип результата «Результат N+1», то INSERT с таким значением не работает**
- **Надо делать DETACH / ATTACH с обновлённым условием (тяжёлая операция)**

```
-- Для обновления ограничения на ключ партицирования  
-- Нужно отключать партицию
```

```
ALTER TABLE activity_20231201_20240301  
DETACH PARTITION activity_20231201_20240301_i;
```

```
-- А затем включать с обновлённым ограничением  
-- Долгая операция с блокировкой таблицы
```

```
ALTER TABLE activity_20231201_20240301  
ATTACH PARTITION activity_20231201_20240301_i  
FOR VALUES ('Результат 1', ..., 'Результат N+1');
```


Партицирование по типу результата

- Добавим числовую колонку **result_code**
- **-100000 <= result_code < 0** для Автоответчиков
- **0 <= result_code < 100000** для важных результатов
- Можно делать **PARTITION BY RANGE**
- INSERT для новых значений будет работать для существующих партиций

```
CREATE TABLE activity_20231201_20240301
PARTITION OF activity
FOR VALUES FROM ('2023-12-01') TO ('2024-03-01')
PARTITION BY RANGE (result_code);
```

```
-- Для партиции ni указываем отрицательные
CREATE TABLE activity_20231201_20240301_ni
PARTITION OF activity_20231201_20240301
FOR VALUES FROM (-100000) TO (0);
```

```
-- Для партиции i указываем неотрицательные
CREATE TABLE activity_20231201_20240301_i
PARTITION OF activity_20231201_20240301
FOR VALUES FROM (0) TO (100000);
```

Препятствия на пути к партицированию

- Колонка **created_dt** есть только в таблице activity
- Колонка **result** (текстовая) есть только в таблице activity
- Колонки **result_code** (числовой) нет нигде
- **Невозможно обычную таблицу превратить в партицированную**
 - Партицированную таблицу необходимо создавать новую
 - Данные необходимо **либо копировать** из старых таблиц, **либо присоединять** через ATTACH

Метод копирования, ч. 1

- Для каждой таблицы **создать её партицированную версию** с суффиксом `_part`
- **Структура аналогична** существующим таблицам, но **добавляем колонки** `created_dt` и/или `result_code`
- Объявляем **партиции первого уровня по `created_dt`**, первую дату указываем в прошлом ~90 дней назад
- Где нужно, **разбиваем их еще на две по `result_code`** – на `important` и `not important`

```
CREATE TABLE communicant_part (  
    id int8 NOT NULL DEFAULT nextval('"communicant_id_seq"'),  
    ...  
    created_dt timestamp NOT NULL,  
    result_code int NOT NULL,  
) PARTITION BY RANGE (created_dt);
```

```
CREATE TABLE communicant_20231201_20240301  
PARTITION OF communicant_part  
FOR VALUES FROM ('2023-12-01T00:00:00') TO ('2024-03-01T00:00:00')  
PARTITION BY RANGE (result_code);
```

```
CREATE TABLE communicant_20231201_20240301_i  
PARTITION OF communicant_20231201_20240301  
FOR VALUES FROM (0) TO (100000);
```

```
CREATE TABLE communicant_20231201_20240301_ni  
PARTITION OF communicant_20231201_20240301  
FOR VALUES FROM (-100000) TO (0);
```

Метод копирования , ч. 2

- **Копируем данные** из исходной таблицы в партицированную, **подтягивая на лету значения для недостающих колонок**
- Из исходной таблицы берём не все строки, а только созданные после первой даты первой партиции

```
INSERT INTO communicant_part (  
    id,  
    ...  
    result_code,  
    created_dt  
)  
  
SELECT  
    -- Берём все имеющиеся в communicant колонки  
    c.id,  
    ...  
  
    -- created_dt берём из activity  
    a.created_dt,  
  
    -- result_code берём из справочника по текстовому значению в activity  
    (SELECT code FROM result_list r  
     WHERE r."name" = a.result)  
FROM activity a  
INNER JOIN communicant c ON a.communicant_id = c.id  
WHERE a.created_dt > '2023-07-01T00:00:00'; -- начиная с дня 0
```

Метод копирования , ч. 3

- **Добавить нужные индексы** на мастер-таблицах (при этом они будут добавлены во всех партициях автоматически)
- **Переименовать таблицы** так, чтобы место старых **непартицированных таблиц заняли новые партицированные**

```
CREATE INDEX ON communicant_part(client_id, created_dt);  
CREATE INDEX ON call_part(activity_id);  
CREATE INDEX ON chat_part(activity_id);  
CREATE INDEX ON opt_data_part(activity_id);  
.....
```

```
ALTER TABLE communicant RENAME TO communicant_old;  
ALTER TABLE communicant_part RENAME TO communicant;
```

```
ALTER TABLE activity RENAME TO activity_old;  
ALTER TABLE communicant_part RENAME TO activity;
```

```
...
```

Метод копирования



Положительные стороны

- Сразу отбросим старые данные. В новые таблицы попадут только активности не сильно старше 90 дней
- Сразу вынесем автоответчики в отдельную партицию



Отрицательные стороны

- Копирование большого объёма данных – **долгая и тяжёлая операция с даунтаймом сервиса**
- Рост объёма занимаемого дискового пространства
- Чтоб избежать большого даунтайма нужно разрабатывать фоновый процесс постепенного копирования

Метод присоединения

- Аналогично **создаём партицированные таблицы с недостающими колонками**
- Партиции **по дате** создаём начиная с **некоторого момента в будущем (день X)**
- Где нужно, **каждую партицию разбиваем еще на две по result_code**
- Существующие таблицы **присоединяем как партиции на диапазон дат от начала времён до дня X**

```
CREATE TABLE communicant_part (  
    id int8 NOT NULL DEFAULT nextval('"communicant_id_seq"'),  
    ...  
    created_dt timestamp NOT NULL,  
    result_code int NOT NULL,  
) PARTITION BY RANGE (created_dt);  
  
-- Партиция на будущее (после дня X)  
CREATE TABLE communicant_20231201_20240301  
PARTITION OF communicant_part  
FOR VALUES FROM ('2023-12-01T00:00:00') TO ('2024-03-01T00:00:00')  
PARTITION BY RANGE (result_code);  
  
-- Её так же разделяем на i / ni  
-- .....  
  
-- Существующие данные (до дня X) присоединяем целиком  
ALTER TABLE communicant_part  
ATTACH PARTITION communicant  
FOR VALUES FROM ('0001-01-01T00:00:00') TO ('2023-12-01T00:00:00');
```

Метод присоединения

- Аналогично **создаём партицированные таблицы с недостающими колонками**
- Партиции **по дате** создаём начиная с **некоторого момента в будущем (день X)**
- Где нужно, **каждую партицию разбиваем еще на две по result_code**
- Существующие таблицы **присоединяем как партиции на диапазон дат от начала времён до дня X**
- **ATTACH не выполнится, т.к. в исходной таблице нет колонок created_dt и/или result_code**

```
CREATE TABLE communicant_part (  
    id int8 NOT NULL DEFAULT nextval('"communicant_id_seq"'),  
    ...  
    created_dt timestamp NOT NULL,  
    result_code int NOT NULL,  
) PARTITION BY RANGE (created_dt);  
  
-- Партиция на будущее (после дня X)  
CREATE TABLE communicant_20231201_20240301  
PARTITION OF communicant_part  
FOR VALUES FROM ('2023-12-01T00:00:00') TO ('2024-03-01T00:00:00')  
PARTITION BY RANGE (result_code);  
  
-- Её так же разделяем на i / ni  
-- .....  
  
-- Существующие данные (до дня X) присоединяем целиком  
ALTER TABLE communicant_part  
ATTACH PARTITION communicant  
FOR VALUES FROM ('0001-01-01T00:00:00') TO ('2023-12-01T00:00:00');
```


Добавление ключей партицирования

- Добавить колонку **result_code** со значением по умолчанию **0** в необходимые таблицы
- Добавить колонку **created_dt** со значением по умолчанию **'0001-01-01'** в необходимые таблицы
- Наличие этих колонок с такими значениями **позволит сделать АТТАСН с нужным ограничением на ключ партицирования**

```
ALTER TABLE communicants  
ADD COLUMN result_code int  
| NOT NULL DEFAULT 0;
```

```
ALTER TABLE communicants  
ADD COLUMN created_dt timestamp  
| NOT NULL DEFAULT '0001-01-01T00:00:00';
```

Доработка метода создания активностей

- При добавлении новых активностей поле **created_dt** для всех таблиц устанавливаем в то же значение что и в таблице activity
- Поле **result_code** заполняем по справочнику на основе текстового значения result
- Эти доработки **позволят после дня X делать INSERT уже в новые таблицы**

Эффективное выполнение ATTACH

- ATTACH – долгая ресурсоёмкая операция с блокировкой таблицы
- Если на таблице есть **CONSTRAINT** аналогичный правилу партицирования – ATTACH выполняется мгновенно
- Но создание CONSTRAINT тоже выполняется долго и с блокировкой
- Можно избежать блокировки с помощью опции **NOT VALID**, вызвав валидацию позже

-- Быстрая операция т.к. есть опция NOT VALID

```
ALTER TABLE activity
```

```
  ADD CONSTRAINT activity_old CHECK (  
    created_dt >= '0001-01-01T00:00:00' AND  
    created_dt < '2023-12-01T00:00:00')  
  NOT VALID;
```

-- Долгая операция, но зато без блокировки

```
ALTER TABLE activity VALIDATE CONSTRAINT activity_old;
```

-- Аналогично делаем для остальных таблиц

АТТАСН всё равно выполнялся долго 😞

- При отработке на клоне продовой базы столкнулись с тем, что **АТТАСН выполняется очень долго и с высоким потреблением ресурсов даже при наличии констрейнта**
- Причина: **при АТТАСН запускался процесс создания индекса по id + created_dt + result_code**

Ошибка при создании мастер-таблиц

- Это случилось из-за неаккуратности при объявлении мастер-таблицы: **объявили составной PK**
- Составной PK требует индекс, которого в присоединяемой таблице не было
- Его создание запускалось при ATTACH автоматически

```
-- Скопировали существующий DDL вместе с PK
CREATE TABLE activity_part (
    ...
    result_code int4 NOT NULL DEFAULT 0
    CONSTRAINT activity_pkey PRIMARY KEY (id)
) PARTITION BY RANGE(created_dt);

-- Не работает :(

-- Необходимо включить в PK все ключи партицирования
CREATE TABLE activity_part (
    ...
    result_code int4 NOT NULL DEFAULT 0,
    CONSTRAINT activity_part_pkey
    PRIMARY KEY (id, created_dt, result_code)
) PARTITION BY RANGE(created_dt);

-- Но это требует соответствующий индекс в партициях
-- И его создание запустится при ATTACH
```

Ошибка при создании мастер-таблиц

- Решение: **не создавать РК на мастер-таблице**, но не забывать **создавать его на всех новых партициях последнего уровня**
- В партициях последнего уровня можно использовать в качестве РК одну колонку id

```
-- Решение: на мастер таблице не указываем РК
CREATE TABLE activity_part (
    ...
    result_code int4 NOT NULL DEFAULT 0
) PARTITION BY RANGE(created_dt);

-- На партиции первого уровня тоже не указываем
CREATE TABLE activity_20231201_20240301
PARTITION OF activity_part
FOR VALUES FROM ('2023-12-01') TO ('2024-03-01')
PARTITION BY RANGE(result_code);

-- А на партиции нижнего уровня РК уже добавляем
CREATE TABLE activity_20231201_20240301_i
PARTITION OF activity_20231201_20240301
FOR VALUES FROM (0) TO (100000);

ALTER TABLE activity_20231201_20240301_i
ADD PRIMARY KEY(id)
```

Оптимизация запроса под партиции

- В запрос добавим во все фильтры условия по ключам партицирования,
- Тогда планировщик сразу отбросит ненужные партиции

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON
    a.communicant_id = c0.id AND
    c0.created_dt >= '2023-03-01T00:00:00' AND
    c0.result_code >= 0
LEFT JOIN "call" AS c1 ON
    a.id = c1.activity_id AND
    c1.created_dt >= '2023-03-01T00:00:00' AND
    c1.result_code >= 0
LEFT JOIN chat AS c2 ON
    a.id = c2.activity_id AND
    c2.created_dt >= '2023-03-01T00:00:00'
LEFT JOIN opt_data AS o ON
    a.id = o.activity_id AND
    o.created_dt >= '2023-03-01T00:00:00'
WHERE a.client_id = '...' AND
    a.created_dt >= '2023-03-01T00:00:00' AND
    a.result_code >= 0 AND
    a.result != 'Автоответчик'
ORDER BY a.id, c0.id, c1.id, c2.id
```

Оптимизация запроса под партиции

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON
    a.communicant_id = c0.id AND
    c0.created_dt >= '2023-03-01T00:00:00' AND
    c0.result_code >=0
LEFT JOIN "call" AS c1 ON
    a.id = c1.activity_id AND
    c1.created_dt >= '2023-03-01T00:00:00' AND
    c1.result_code >=0
LEFT JOIN chat AS c2 ON
    a.id = c2.activity_id AND
    c2.created_dt >= '2023-03-01T00:00:00'
LEFT JOIN opt_data AS o ON
    a.id = o.activity_id AND
    o.created_dt >= '2023-03-01T00:00:00'
WHERE a.client_id = '...' AND
    a.created_dt >= '2023-03-01T00:00:00' AND
    a.result_code >= 0 AND
    a.result != 'Автоответчик'
ORDER BY a.id, c0.id, c1.id, c2.id
```

<=>

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON
    a.communicant_id = c0.id
LEFT JOIN "call" AS c1 ON
    a.id = c1.activity_id
LEFT JOIN chat AS c2 ON
    a.id = c2.activity_id
LEFT JOIN opt_data AS o ON
    a.id = o.activity_id
WHERE a.client_id = '...' AND
    a.created_dt >= '2023-03-01T00:00:00' AND
    a.result_code >= 0 AND
    a.result != 'Автоответчик' AND
    c0.created_dt >= '2023-03-01T00:00:00' AND
    c0.result_code >=0 AND
    c1.created_dt >= '2023-03-01T00:00:00' AND
    c1.result_code >=0 AND
    c2.created_dt >= '2023-03-01T00:00:00' AND
    o.created_dt >= '2023-03-01T00:00:00'
ORDER BY a.id, c0.id, c1.id, c2.id
```


Производительность упала в 50+ раз 😞

- На исходной БД: **~20 ms**
- На партицированной БД: **более 1000 ms**

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON
    a.communicant_id = c0.id AND
    c0.created_dt >= '2023-03-01T00:00:00' AND
    c0.result_code >= 0
LEFT JOIN "call" AS c1 ON
    a.id = c1.activity_id AND
    c1.created_dt >= '2023-03-01T00:00:00' AND
    c1.result_code >= 0
LEFT JOIN chat AS c2 ON
    a.id = c2.activity_id AND
    c2.created_dt >= '2023-03-01T00:00:00'
LEFT JOIN opt_data AS o ON
    a.id = o.activity_id AND
    o.created_dt >= '2023-03-01T00:00:00'
WHERE a.client_id = '...' AND
    a.created_dt >= '2023-03-01T00:00:00' AND
    a.result_code >= 0 AND
    a.result != 'Автоответчик'
ORDER BY a.id, c0.id, c1.id, c2.id
```

План выполнения на партицированной БД

В плане запроса **появился пункт JIT**, и EXPLAIN ANALYZE показывает что он **тратит всё время**

```
Gather Merge (cost=1191337307570179.50..1256362072643365.75 rows=557316216567106 width=916)
  Workers Planned: 2
  -> Sort (cost=1191337307569179.50..1192033952839888.50 rows=278658108283553 width=916)
      Sort Key: c.id, c.communicant_id, c1.id, c2.id
      -> Nested Loop (cost=1.86..594825841450.80 rows=278658108283553 width=916)
          -> Nested Loop Left Join (cost=1.43..114504587.25 rows=50154867119 width=788)
              .....
              .....
              -> Index Scan using "communicant_20231201_20240301_i_pkey" on "communicant_20231201_20240301_i" c0_1
                  Index Cond: (id = c.communicant_id)
                  Filter: ((created_dt >= '2023-03-01 00:00:00'::timestamp without time zone) AND (result_code >= 0))
```

JIT:

Functions: 54

Options: Inlining true, Optimization true, Expressions true, Deforming true

План выполнения на партицированной БД

Причина включения JIT – астрономическая сложность, прогнозируемая планировщиком

```
Gather Merge (cost=1191337307570179.50..1256362072643365.75 rows=557316216567106 width=916)
  Workers Planned: 2
  -> Sort (cost=1191337307569179.50..1192033952839888.50 rows=278658108283553 width=916)
    Sort Key: c.id, c.communicant_id, c1.id, c2.id
    -> Nested Loop (cost=1.86..594825841450.80 rows=278658108283553 width=916)
      -> Nested Loop Left Join (cost=1.43..114504587.25 rows=50154867119 width=788)
        .....
        .....
        -> Index Scan using "communicant_20231201_20240301_i_pkey" on "communicant_20231201_20240301_i" c0_1
          Index Cond: (id = c.communicant_id)
          Filter: ((created_dt >= '2023-03-01 00:00:00'::timestamp without time zone) AND (result_code >= 0))
JIT:
  Functions: 54
  Options: Inlining true, Optimization true, Expressions true, Deforming true
```

Если выключить JIT

- Время выполнения запроса становится аналогичным времени на непартицированной БД: **~20 ms**
- При этом cost в плане запроса всё ещё астрономический
- Вероятно, **баг планировщика**.
Проверялось на PostgreSQL 12.

```
SET jit = OFF;
```

```
SELECT a.*, c0.*, c1.*, c2.*, o.*
FROM activity AS a
INNER JOIN communicant AS c0 ON
    a.communicant_id = c0.id AND
    c0.created_dt >= '2023-03-01T00:00:00' AND
    c0.result_code >= 0
LEFT JOIN "call" AS c1 ON
    a.id = c1.activity_id AND
    c1.created_dt >= '2023-03-01T00:00:00' AND
    c1.result_code >= 0
LEFT JOIN chat AS c2 ON
    a.id = c2.activity_id AND
    c2.created_dt >= '2023-03-01T00:00:00'
LEFT JOIN opt_data AS o ON
    a.id = o.activity_id AND
    o.created_dt >= '2023-03-01T00:00:00'
WHERE a.client_id = '...' AND
    a.created_dt >= '2023-03-01T00:00:00' AND
    a.result_code >= 0 AND
    a.result != 'Автоответчик'
ORDER BY a.id, c0.id, c1.id, c2.id
```

Изменение запроса

- Снова **отказываемся от получения всего за один запрос**
- Получаем **activity + communicant**, здесь можно с одним INNER JOIN т.к. планировщик оценивает адекватно
- Из полученного списка **берём activity.id только для звонков и по ним получаем записи из таблицы call**
- Аналогично через WHERE IN забираем данных из других таблиц
- Т. о. исправляем проблему **неадекватного плана и неоптимальные особенности** старого большого запроса

```
var activities = await dbContext.Activities
    .AsNoTracking()
    .Include(x => x.Communicant)
    .Where(x => x.ClientId == clientId)
    .Where(x => x.CreatedDt >= startDt)
    .Where(x => x.ResultCode >= 0 && x.Result != "АВТООТВЕТЧИК")
    .Where(x => x.Communicant.CreatedDt >= startDt)
    .Where(x => x.Communicant.ResultCode >= 0)
    .ToListAsync();

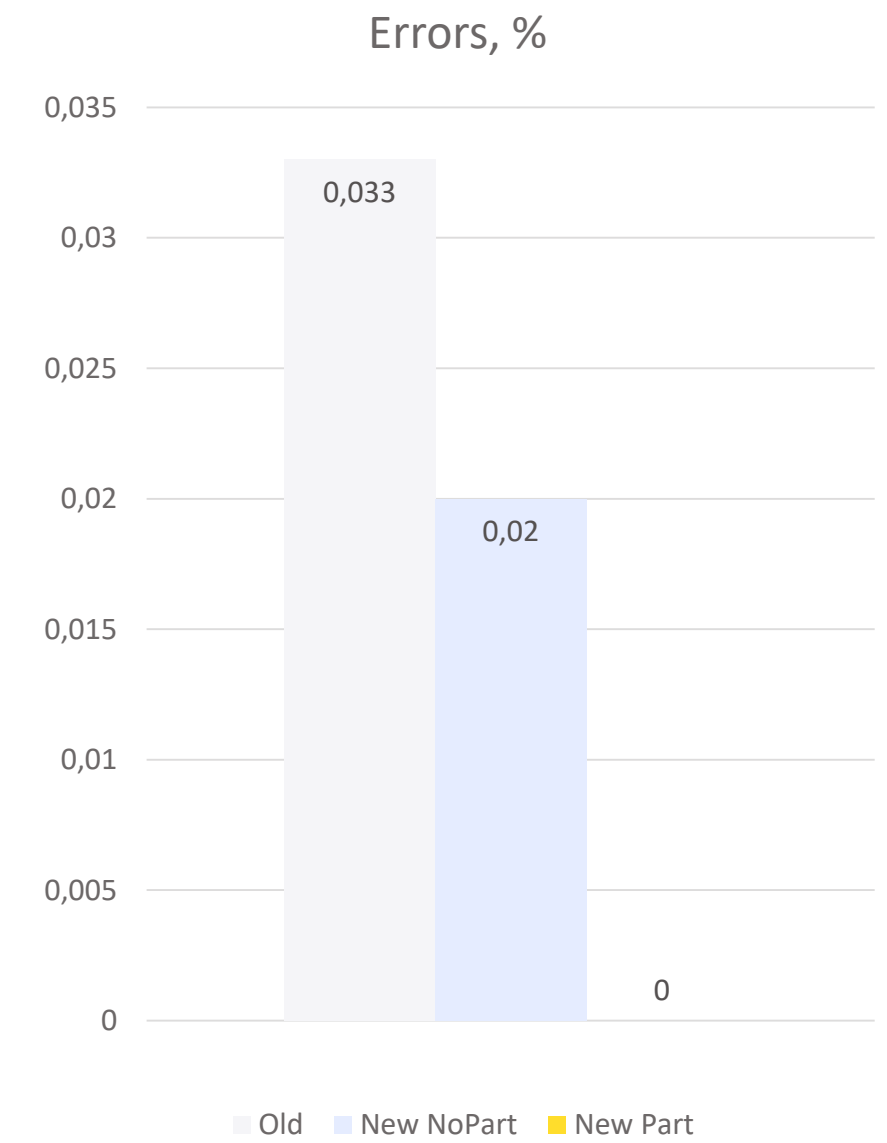
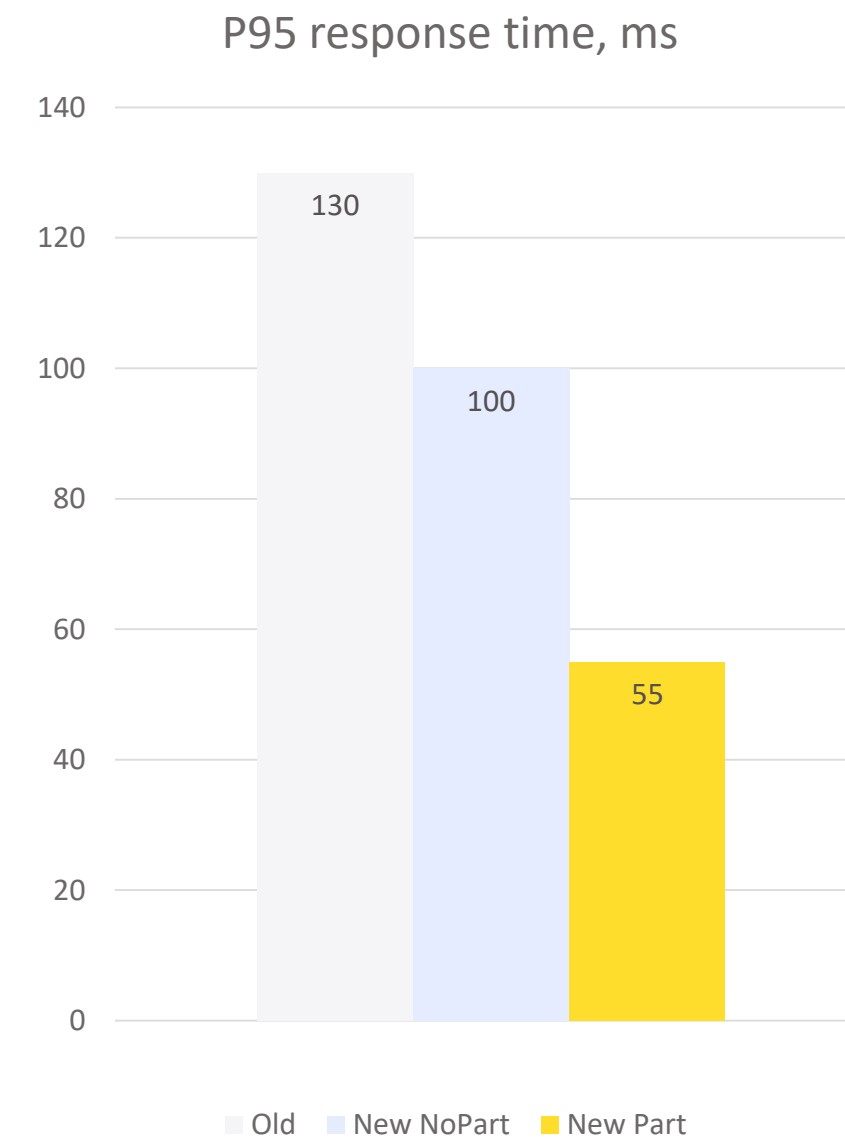
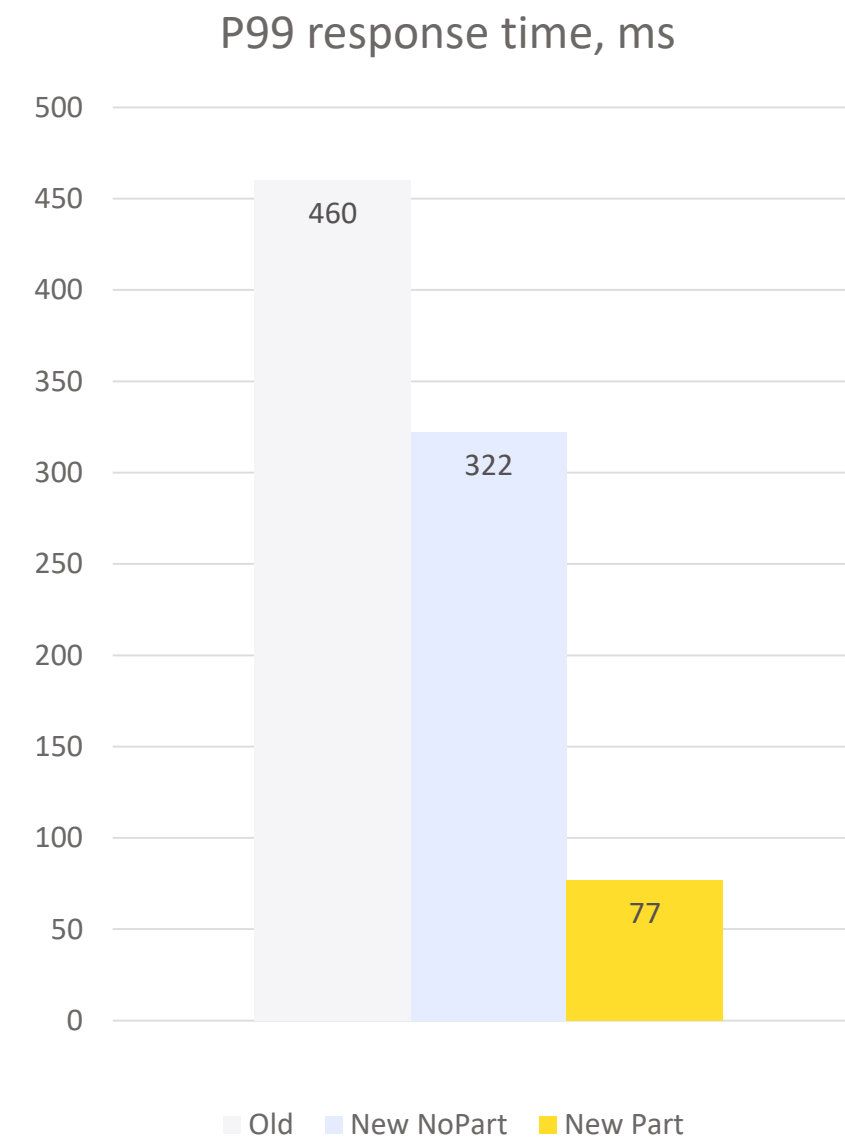
var callActIds = activities
    .Where(x => IsCall(x.ActivityType))
    .Select(x => x.Id);

var calls = await dbContext.Calls
    .AsNoTracking()
    .Where(x => callActIds.Contains(x.Id))
    .Where(x => x.CreatedDt >= startDt && x.ResultCode >= 0)
    .ToListAsync();

// аналогично получаем chats
// и opt_data по всем activity.id
```

Новая логика запроса данных без запроса с кучей JOIN-ов

Изменения в результатах нагрузочных тестов



Релиз

Этап 1

- Добавление новых колонок
- Доработки на стороне приложения

Этап 2

- Создание Constraint-ов на таблицах в окошко наименьшей загруженности сервиса

Этап 3

- Создание новых таблиц и пустых партиций на будущее
- Временная остановка репликации в DWH
- ATTACH и переименование таблиц

2023—2024

Итоги на сегодня

- **160+ млн** активностей, **160+ ГБ**
- БД росла с ускорением, каждая партиция на 30-40% больше предыдущей
- Response Time P90: **20-40ms при 60 RPS**
- **0% ошибок, ни одного сбоя**
- Ресурсы БД: **4ГБ ОЗУ, 2ГБ под Shared Buffers**



Что имели на момент сбоя

- **15 млн** активностей в базе
- Response Time **8.5s при 20-30 RPS**
- Очень **высокий процент ошибок**



Выводы

Интеграции

- **Не стоит допускать интеграций с другими системами, завязанных на схему БД** (как в нашем случае была реализована интеграция с DWH)
- Предпочтительный вариант: **взять на себя ответственность за отправку данных в другие системы**, например отправлять данные в Kafka по заранее оговоренным контрактам

Схема данных

- **Не стоит использовать TEXT** под поля с **ограниченным набором значений**
- Нормализация схемы может пагубно влиять на производительность запросов
- Поменять радикально схему данных на поздних этапах будет затруднительно

Нагрузочные тесты

- Важен не только RPS, но и размер базы данных
- **Структура данных и профиль нагрузки должны соответствовать ситуации на проде**
 - В наших тестах создавалось много клиентов
 - Но у каждого клиента была 1 активность
 - Поэтому тесты не смогли предсказать сбой на проде, где по каждому клиенту читались десятки активностей

Бизнес- требования

- Нужно быть **внимательным к бизнес-требованиям**, возможно ваш сервис делает никому не нужную работу
- Небольшое изменение функционала может дать очень большой прирост производительности при минимальных затратах

Партицирование

- Имеет смысл только **на больших таблицах**
- **Ускоряет чтение**, особенно в том случае, когда запросы **часто обращаются только к партициям с небольшим процентом от всех данных**
- Аналогично может **ускорить запись**
- Позволяет **эффективно удалять** неактуальные данные
- **Упрощает создание индексов** в больших таблицах

Партиципирование

- Нужно внимательно **следить за всеми запросами**
 - Не стоит допускать ситуаций, когда запросы затрагивают много партиций
- Для существующей большой нагруженной БД **процесс партиципирования требует усилий**
- Нужно **вовремя создавать новые партиции**



Спасибо!