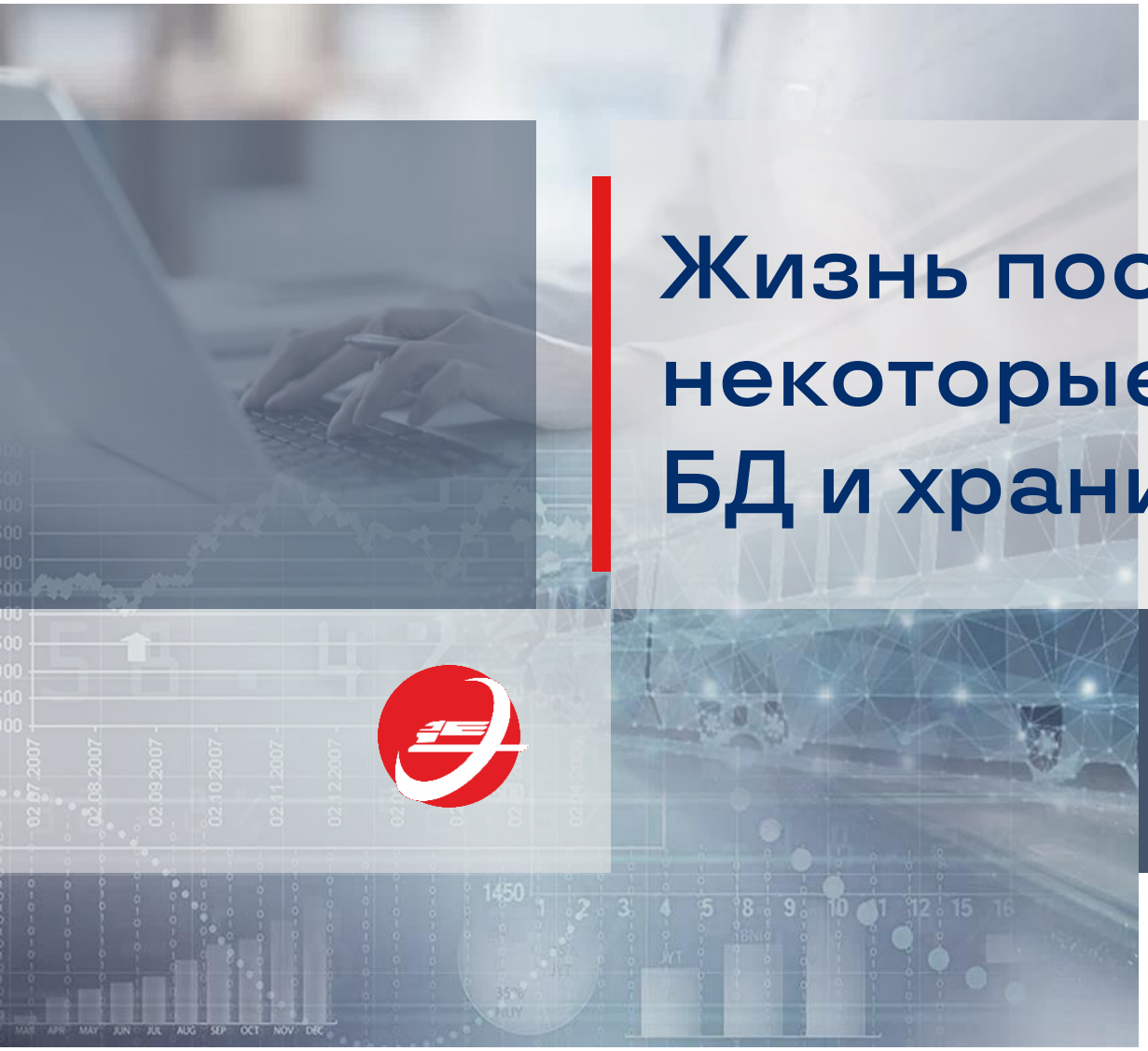
ВНИИЖТ
НЦ ЦМПЭ



Жизнь после импортозамещения: некоторые особенности настройки БД и хранимых процедур

Анатолий Анфиногенов

Краткое содержание предыдущих серий

2020 Как я перестал беспокоиться и перенес 60К строк из 150 процедур PL/SQL в Postgres

https://pgconf.ru/2020/274661

2021 Миграция приложения Oracle PL/SQL на Postgres pl/pgSQL: взгляд два года спустя

https://pgconf.ru/202110/308499

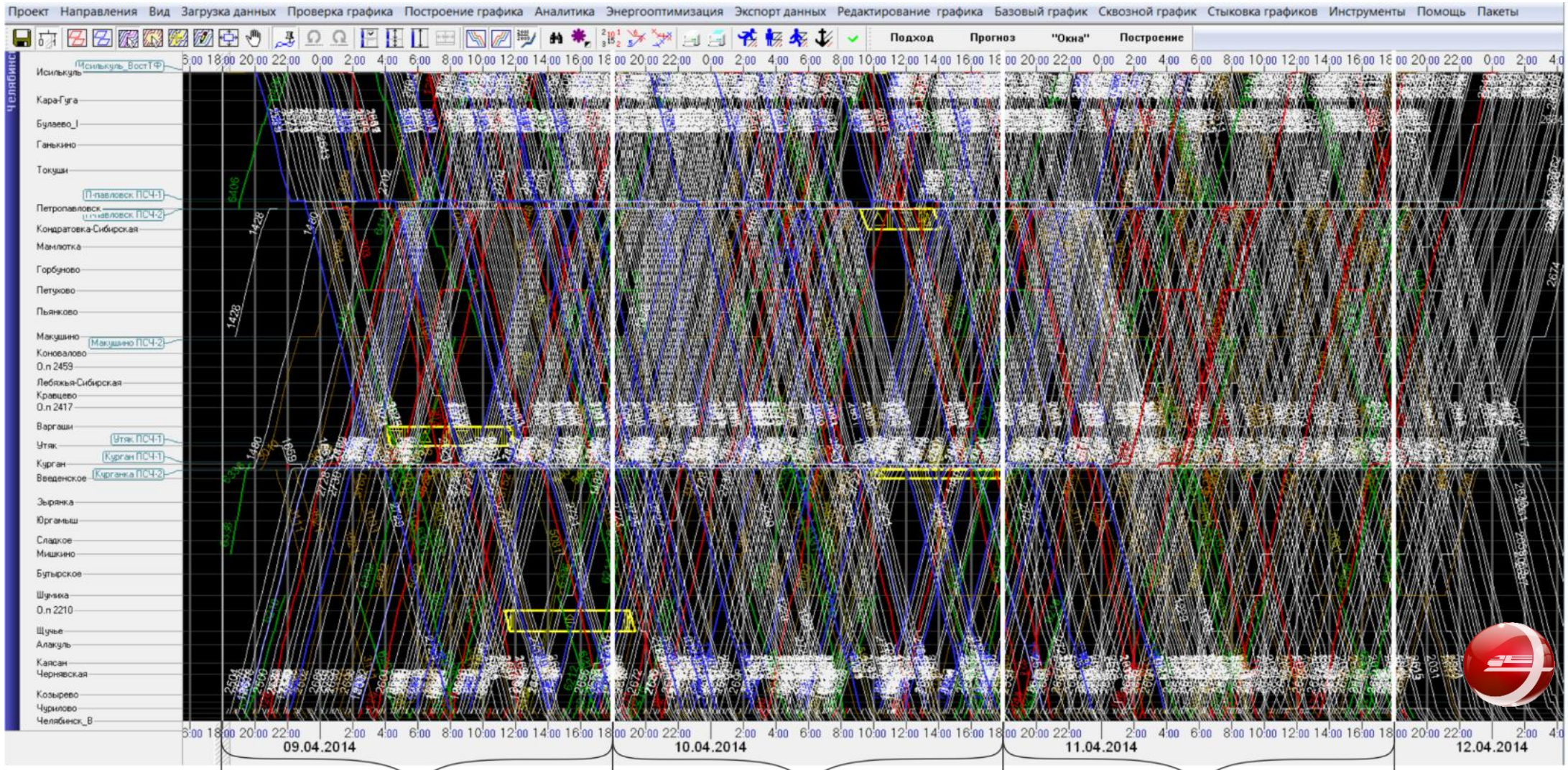
Что это и зачем это нужно?

ЭЛЬБРУС (2006 г. – н.в.) – это система планирования движения грузовых поездов по энергооптимальным расписаниям, пополнившаяся в 2021 году прогнозно-аналитической подсистемой ЭЛЬБРУС–М.

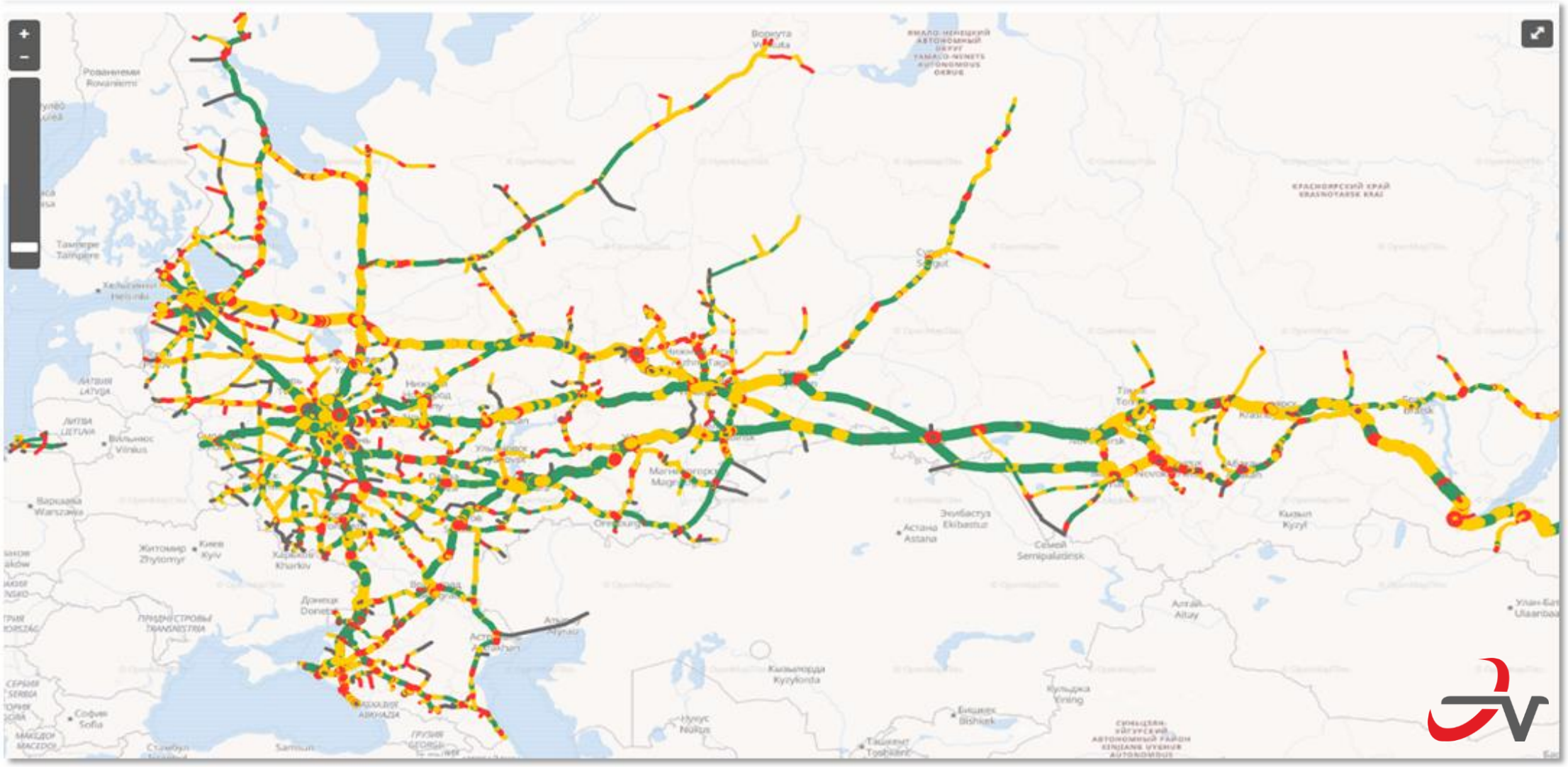
Первая премия UIC/МСЖД-2012 в области железнодорожных исследований и инноваций

ЭЛЬБРУС – это распределенное многозвенное технологическое приложение, работающее на всех железных дорогах России от Калининграда до Хабаровска

Так выглядит расписание поездов снаружи



Так выглядит прогнозная аналитика



Так устроено расписание поездов изнутри

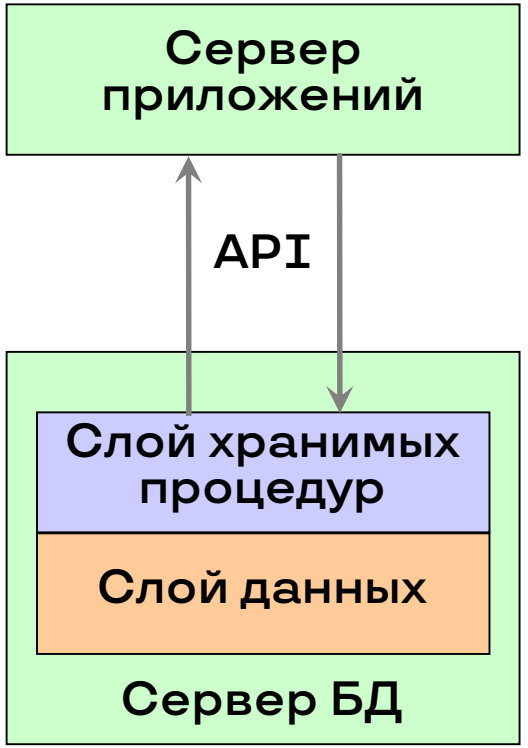
- До 2 млн объектов нескольких десятков типов в одном расписании
- Объем данных до 500 Мбайт на одно расписание одной железной дороги
- Для каждой из 16 железных дорог России каждый день разрабатывается и используется несколько расписаний различных типов и назначений
- В оперативной базе одновременно содержится порядка 500 активных расписаний + архивная БД
- Расписания из оперативной БД поступают в центральную базу прогнозной аналитики

Архитектура ЭЛЬБРУС: трехзвенная

- **Эксплуатация:** 24/7; регулярные обновления серверных приложений 3-6 раз в год, включая приложения БД
- **Толстый клиент:** Windows, C++, MQ
- **Тонкий клиент:** GWT, Angular
- **Сервер приложений:** Java, Tomcat, MQ
- **Сервер БД:** Ванильный Postgres 11.11,
  – 50000 строк хранимых процедур pl/pgSQL;
  – Очень крупные, но относительно редкие транзакции;
  – Работа с БД – только через API на основе хранимых процедур.

Почему только хранимые процедуры?

- Взаимодействие в предметных категориях приложения.
- Логика хранения с помощью API ХП отделена от бизнес-логики приложения;
- Можно вносить изменения в структуры данных и организацию БД без изменения сервера приложений (в пределах API).
- Возможно гладкое поэтапное обновление распределенного ПО за счет версионности API.
- Встроенный механизм диагностики, логирования, отладки и профилирования приложения БД без остановки сервиса, включаемый и управляемый параметрически.



Особенности нашей базы данных

- ~250 таблиц, 250 Гб оперативных данных, обновление до 40-60% содержимого БД за неделю
- ~200 хранимых процедур на `pl/pgSQL` (~50000 строк) в отдельной схеме
- Эмуляция (`dblink` + `pg_variables`) автономных транзакций для логирования вызовов API
- Эмуляция временных таблиц (в стиле Oracle) как средства обмена сервера БД данными большого объема (сотни Мбайт) с сервером приложений при вызовах процедур
- Небольшое число одновременно работающих пользователей (десятки), выполняющих продолжительные (до десятков секунд), но относительно редкие операции

Что нового?

1) Импортозамещение продолжается

- Накопление опыта эксплуатации БД нашего приложения для PostgreSQL: переход от борьбы с детскими болезнями к непрерывной диагностике и лечению хронических заболеваний.
- Опыты в части перехода от свободного ПО к отечественному ПО: CentOS –> RedOS/AstraLinux/?...

2) Разработка продолжается

- В дополнение к системе ЭЛЬБРУС и на ее основе создана система ЭЛЬБРУС–М (М – макромодель) для прогноза продвижения поездопотоков и оценки инфраструктурных и управляющих решений.
- Учтен опыт перехода ЭЛЬБРУС с Oracle на PostgreSQL и двух лет эксплуатации: улучшаем алгоритмы и структуры данных; проектируем и реализуем ЭЛЬБРУС–М правильно с учетом особенностей PostgreSQL!

История импортозамещения в ЭЛЬБРУС

2018 — Старт перехода

Пилотный проект, к началу 2019 г. перенесено 11 процедур на PostgreSQL 10

2019 — Разработка

Отработаны инфраструктурные решения, созданы структуры данных и перенесены 200 процедур (60000 строк) на PostgreSQL 11

2020 — Переход

Параллельная эксплуатация старой и новой БД в переходный период, перенос данных, начальная настройка производительности

2021 — Эксплуатация: детские болезни

Создание индексов, работа с временными таблицами, оптимизация SQL-команд

2022 — Эксплуатация: хронические заболевания

Настройка сбора статистики и автовакуума; борьба с «распуханием» таблиц и индексов

Проблема выбора: как переносить БД+ХП?

— Переносим как есть

— Перенос + мини-рефакторинг

— Напишем заново как следует

Пошаговая история импортозамещения – 1

1. Разработка общей технологии миграции: зачем, что, когда и как следует сделать.
2. Обучение особенностям работы с PostgreSQL.
3. Разработка инфраструктурных workaround, без которых наше приложение не перенести.
4. Перенос таблиц, VIEW, SEQUENCES и т. п.
5. Перенос (возможно, с переделкой) хранимых процедур.
6. Тестирование, отладка и первоначальная оптимизация производительности приложения БД.

Пошаговая история импортозамещения – 2

7. Разработка технологического процесса переключения.

8. Разработка эксплуатационной документации и инструкций для администраторов.

9. Обучение администраторов.

10. Переключение и начало переходного периода параллельной эксплуатации.

11. Организация сопровождения приложения и БД, включая мониторинг.

12. Разработка и внедрение технологии выполнения обновлений.

Пошаговая история импортозамещения – 3

13. Выявление и преодоление «детских болезней».
14. Изучение основ управления производительностью PostgreSQL и применение их на практике.
15. Пересмотр и расширение номенклатуры метрик мониторинга приложения и БД.
16. Расширение перечня расширений (sic!).
17. Диагностика «хронических заболеваний» и их лечение.

←Вы находитесь здесь

18. Переход с ванильной PostgreSQL на PostgresPro.
19. ???
20. PROFIT!

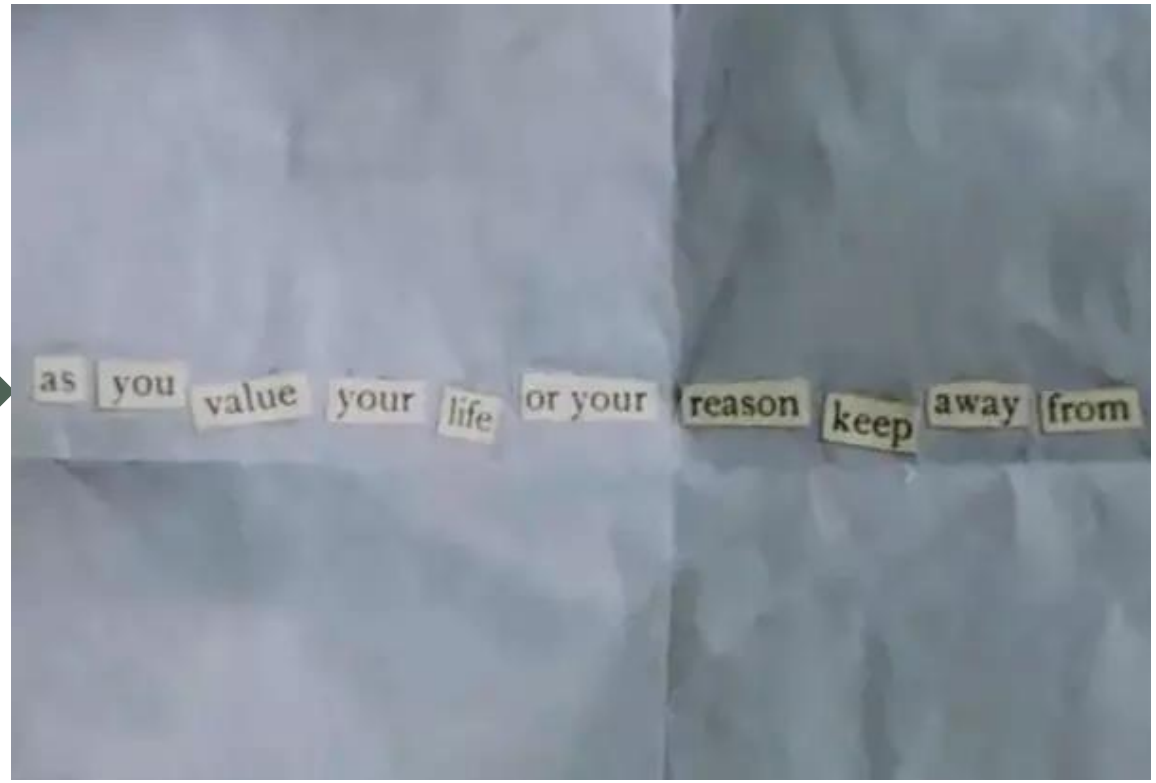
Некоторые инфраструктурные решения – 1

- Хранимые процедуры пишут логи в лог-таблицы вне зависимости от успешности транзакции – нужны автономные транзакции.
- В ванильном PostgreSQL автономных транзакций нет.
- Их эмуляция через **dblink** работает медленно, т.к. создание соединения – дорогая операция.
- Спасает расширение **pg_variables**; спасибо, Иван Фролков! В переменной храним открытое соединение, и логи начинают писаться с приемлемой скоростью.

Некоторые инфраструктурные решения – 2

- Для приема больших объемов данных от сервера приложений при вызове хранимых процедур нужны временные таблицы (в стиле Oracle).
- Ванильный PostgreSQL не поддерживает временные таблицы, сохраняющиеся в словаре данных СУБД по завершении сессии.
- Неверное решение: их эмуляция с помощью UNLOGGED-таблиц с уникальным GUID слоя данных.
- Верное решение: изменение API путем обязательного вызова функции `prepareCall('`*ИМЯ_ПРОЦЕДУРЫ*`')` для создания временных таблиц в данной сессии.

Несколько слов об UNLOGGED-таблицах



UNLOGGED TABLES

Некоторые инфраструктурные решения – 3

- Для выполнения задач архивирования устаревших данных, техобслуживания БД, диагностики и проактивного мониторинга мы использовали пакетные задания (JOB) Oracle, которые вызывали процедуры с оператором COMMIT внутри.
- Ванильный PostgreSQL не поддерживает JOB
- Для запуска процедур, реализующих эти функции, было создано Java-приложение **jobrunner** для Apache Tomcat. Плюсы: оно гибче, частично унифицировано с сервером приложений и может выполнять не только задания БД. Минусы: более сложная конструкция.

Запомнившиеся моменты

- В Oracle тип **DATE** подобен **TIMESTAMP** с меньшей точностью

```
TO_DATE('2020-10-15 14:00','YYYY-MM-DD HH24:MI') = 2020-10-15 14:00
```

  В PostgreSQL тип **DATE** округляется до даты

```
TO_DATE('2020-10-15 14:00','YYYY-MM-DD HH24:MI') = 2020-10-15
```

- При переносе операторов **MERGE** из Oracle они заменялись на **INSERT(…) ON CONFLICT DO UPDATE…**, но в случае отсутствия основания для конфликта (т.е. первичного ключа или уникального индекса) ветвь **DO UPDATE…** не выполнялась никогда, не вызывая замечаний с точки зрения синтаксиса.

Используемые расширения

Расширение	Назначение в проекте	Штатное?
dblink	Эмуляция автономных транзакций	да
pg_variables	Эмуляция автономных транзакций	нет (да в Pro)
pgcrypto	Генерация GUID	да
plpgsql_check	Проверка хранимых процедур	нет
plpythonu	Работа с файлами в ФС сервера	да
postgres_fdw	Связь с архивными серверами	да
oracle_fdw	Интеграция со смежными системами	нет (да в Pro)
pg_stat_statements	Мониторинг производительности БД	да
pgstattuple	Проверка «распухания» таблиц и индексов	да
pg_repack	Лечение «распухания» таблиц и индексов	да

Нештатные расширения затрудняют обновление СУБД!

Как повлиять на производительность?

Где и как измерять производительность?

Где?

Потребитель

Приложение БД

PostgreSQL

ОС

Как?

- Контроль времени выполнения типовых операций со стороны сервера приложений и клиента.
- Профилирование ХП, запись в БД **гистограмм** производительности; мониторинг производительности.
- Применение расширений и скриптов контроля производительности БД: `pg_stat_statements` и др.
- Контроль средствами ОС

Профилирование хранимых процедур

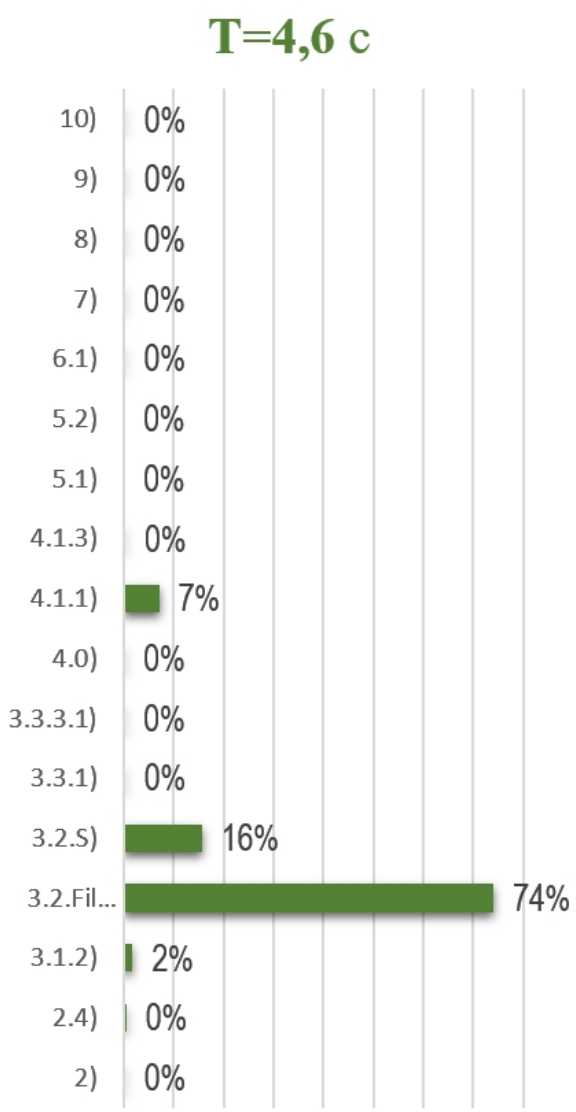
- Возможность профилировать хранимые процедуры должна быть заложена при проектировании.
- Процедура разбита на этапы (один или несколько сходных по типу SQL-операторов); времена их выполнения сохраняются в логи производительности.
- Профилирование должно включаться и управляться параметрически с возможностью отключения; времена выполнения должны попадать в мониторинг.
- Необходимо фиксировать планы выполнения проблемных запросов изнутри хранимой процедуры (это было в приложении для Oracle, но это ~~невозможно~~ трудновыполнимо для ванильной PostgreSQL — печаль).

Пример этапов хранимой процедуры

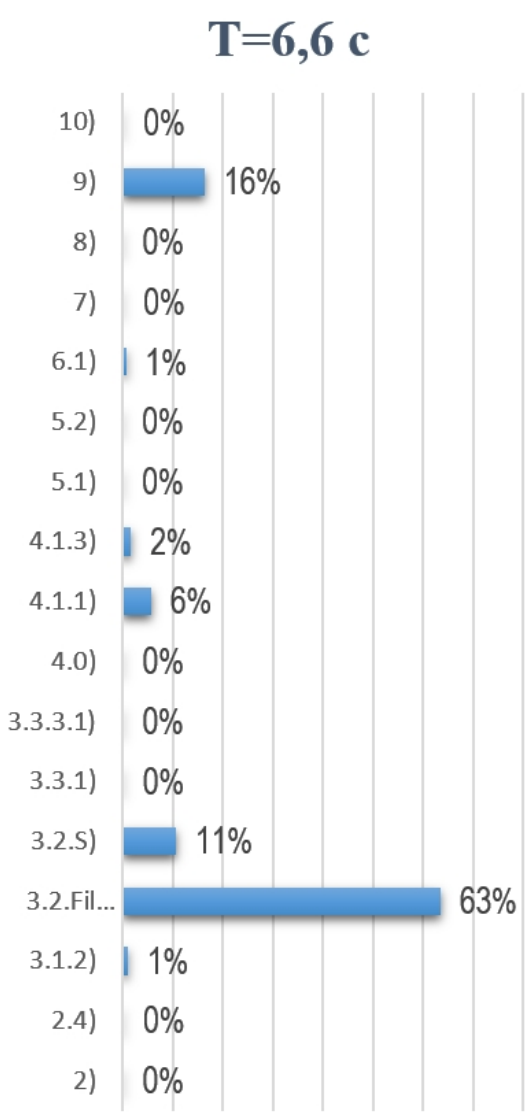
Краткое описание этапов процедуры `getTrainsData()`, выполняющей чтение расписания со сложной фильтрацией

2)	Чтение и проверка входных параметров
2.4)	Создание рабочих временных таблиц
3.1.2)	INSERT,SELECT (4K) : Чтение во временную таблицу всех поездов расписания с первичной фильтрацией
3.2.Filter)	INSERT,SELECT (1112K) : Чтение во временную таблицу всех точек расписания с первичной фильтрацией
3.2.S)	INSERT,SELECT (4K) : Чтение во временную таблицу лишних поездов по доп.критериям фильтрации
3.3.1)	DELETE: Удаляем лишние поезда из таблицы лишних поездов
3.3.3.1)	DELETE: Удаляем поезда с признаком неактивности
4.0)	ANALYZE: Анализируем временную таблицу поездов
4.1.1)	INSERT,SELECT (1000K) : Чтение во временную выходную таблицу точек расписания
4.1.3)	ANALYZE: Анализируем временную выходную таблицу точек расписания
5.1)	DELETE: Дополнительная тонкая фильтрация точек расписания
5.2)	DELETE: Дополнительная тонкая фильтрация точек расписания
6.1)	UPDATE: Вычисление и заполнение серийных номеров точек расписания при необходимости
7)	DELETE: Дополнительная фильтрация расписания по полигонам
8)	DELETE: Удаляем поезда без точек или удовлетворяющие доп.критериям
9)	INSERT,SELECT (225K) : Чтение во временную таблицу всех календарей отобранных поездов расписания
10)	SELECT: Открываем курсоры к временным таблицам и возвращаем результаты

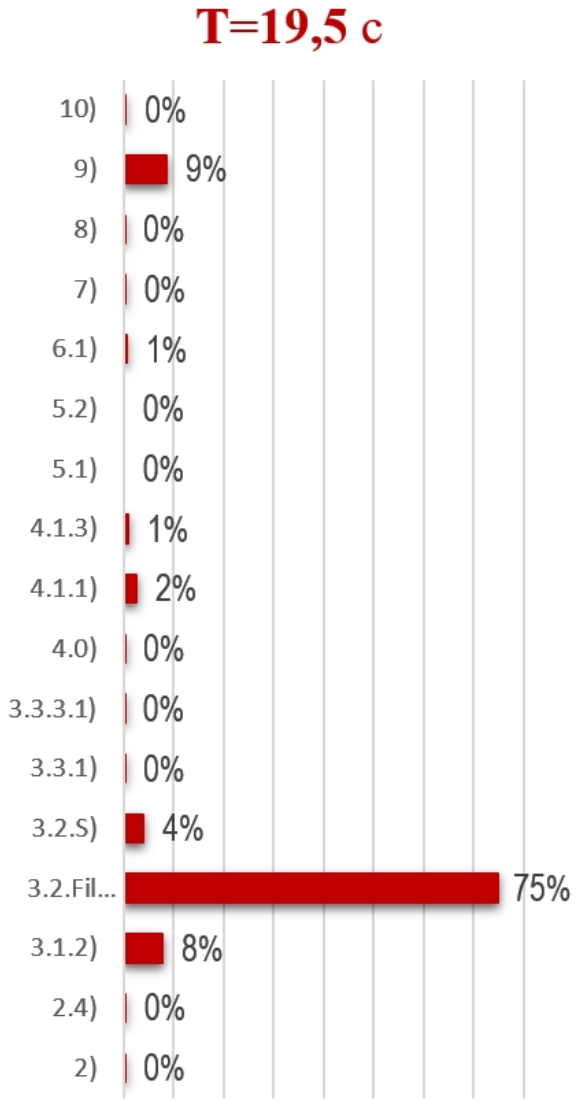
Гистограммы: THE GOOD, THE BAD AND THE UGLY



отлично



приемлемо



плохо

Ускоряем процедуру по гистограмме

Ускорение, раз	Этап процедуры	T, мс (до/после ускорения)
3,00	2)	6
	2)	2
1,30	2.4)	13
	2.4)	10
1,40	3.1.2)	2309
	3.1.2)	1648
17,09	3.2.S)	48468
	3.2.S)	2836
1,04	3.3.1)	55
	3.3.1)	53
0,89	3.3.3.1)	24
	3.3.3.1)	27
	4.0)	20
497,54	4.1)	97517
	4.1.1)	67
	4.1.3)	82
200,00	5.1)	2
	5.1)	0
	5.2)	0
	5.2)	0
1,00	6.1)	6
	6.1)	6
0,83	7)	10
	7)	12
1,00	8)	18
	8)	18
1,55	9)	22812
	9)	14733
1,38	10)	11
	10)	8

- Это — пример итерационного подхода к ускорению процедуры `getTrainsData()`.
- Пик на гистограмме — повод разобраться с SQL-операторами проблемного этапа выполнения процедуры.
- Применяя приемы оптимизации, можно оценивать их эффективность путем сравнения времен выполнения этапов.
- В приведенном примере процедура ускорилась в 9 раз: было 171 с, стало 19 с.
- Основной эффект достигнут на этапах «4.1)», «3.2.S)» и «9)», см. знак ⬭ .
- Знаком ⬭ выделены цели для дальнейшей оптимизации

Гистограмма до и после оптимизации

- Гистограмма времени выполнения этапов процедуры позволяет выявить «узкие места» — этапы, на которые ушло более чем 80% времени выполнения процедуры
- Цель — по возможности более равномерная гистограмма!

А за счет чего это было достигнуто?

«Детские болезни» производительности – 1

Способы ускорения SQL-операторов и хранимых процедур

1. В отличие от Oracle, в PostgreSQL внутри хранимых процедур можно не только создавать таблицы, но и собирать статистику.

Решение: `ANALYZE TMP_InputData;`

Результат: ускорение в 1,5 раза

2. Отсутствие индекса на таблице, которая выросла в размерах. Основная сложность – осознать это.

Решение: `CREATE INDEX …;`

Результат: ускорение в 14000 раз

«Детские болезни» производительности – 2

Проблема *что-то база тормозит*

```
SELECT query, calls, total_exec_time, mean_exec_time
FROM   pg_stat_statements
ORDER BY total_exec_time DESC LIMIT 10;
```

query	calls	total_exec_time	mean_exec_time
select * from M.getHashInfo($1,$2,$3,$4) as	40,827	124,643,313.050181	3,052.96282
FETCH ALL IN "P_RESULTSET"	159,883	41,421,988.167746	259.076876
select * from	13,476	33,627,829.13626	2,495.386549
SELECT $2 FROM ONLY "m"."tt_train_links" x	43,695	30,163,201.28449	690.312422
select * from	13,418	13,552,919.840955	1,010.055138
SELECT	13,418	13,157,557.284014	980.590049
select * from M.saveT_Threads($1,$2,$3) as	12,219	3,569,138.341333	292.097417
select * from M.saveT_Trains($1,$2,$3) as	13,083	3,527,987.256952	269.661947
select * from EAPI.doMonitoringJob( ) as	73,761	1,644,611.695368	22.296494
SELECT dblink_exec(v_ConnName,v_SQL)	837,924	1,489,619.495458	1.77775

Суть проблемы «что-то база тормозит»

Разберемся с первым запросом из Топ-10 pg_stat_statements:

T_bHash
bHash
TableName

650 строк

bGUID2Hash
bHash
TableName

1.1 млн.строк

bGUID2Hash имеет индекс (PK) по bHash

```
SELECT H.*
FROM   T_bHash T
       INNER JOIN bGUID2Hash H
       ON(    (T.bHash=H.bHash)
          AND(T.TableName=H.TableName)
         )
```

```
QUERY PLAN
Hash Left Join  (cost=929678.45..1137426.73 rows=650 width=96)
  Hash Cond: ((t.bhash = h.bhash) AND (t.table_name = h.tablename))
  ->  Seq Scan on t_bhash t  (cost=0.00..16.50 rows=650 width=96)
  ->  Hash  (cost=432011.98..432011.98 rows=19332698 width=61)
        ->  Seq Scan on bguid2hash h  (cost=0.00..432011.98 rows=19332698 width=61)"Planning Time: 5.428 ms
Planning Time: 896.667 ms
Execution Time: 9532.021 ms
```

Решение проблемы «что-то база тормозит»

Таблица bGUID2Hash подросла и Seq Scan по ней стал узким местом – необходимо создать индекс

```
CREATE UNIQUE INDEX bGUID2Hash_IDX1 ON bGUID2Hash(bHash, TableName)
```

```
QUERY PLAN
Nested Loop Left Join  (cost=0.56..5592.75 rows=650 width=96) (actual time=0.292..0.522 rows=3 loops=1)
  ->  Seq Scan on t_bhash t  (cost=0.00..16.50 rows=650 width=96)
      (actual time=0.020..0.025 rows=3 loops=1)
  ->  Index Scan using bguid2hash_idx1 on bguid2hash h  (cost=0.56..8.58 rows=1 width=61)
      (actual time=0.153..0.153 rows=1 loops=3)
        Index Cond: ((bhash = t.bhash) AND (tablename = t.table_name))
Planning Time: 2.212 ms
Execution Time: 0.682 ms
```

Ускорение в 14000 раз!

Кстати: а сколько стоит логирование?

Просуммируем времена связанных с логированием запросов из pg_stat_statements и сравним их с общим временем выполнения запросов:

query	calls	total_exec_time	mean_exec_time
select * from M.getHashInfo($1,$2,$3,$4) as	40,827	124,643,313.050181	3,052.96282
FETCH ALL IN "P_RESULTSET"	159,883	41,421,988.167746	259.076876
select * from	13,476	33,627,829.13626	2,495.386549
SELECT $2 FROM ONLY "m"."tt_train_links" x	43,695	30,163,201.28449	690.312422
select * from	13,418	13,552,919.840955	1,010.055138
SELECT	13,418	13,157,557.284014	980.590049
select * from M.saveT_Threads($1,$2,$3) as	12,219	3,569,138.341333	292.097417
select * from M.saveT_Trains($1,$2,$3) as	13,083	3,527,987.256952	269.661947
select * from EAPI.doMonitoringJob() as	73,761	1,644,611.695368	22.296494
SELECT dblink_exec(v_ConnName,v_SQL)	837,924	1,489,619.495458	1.77775

Общее время, мс	Логирование, мс	Потери, %
307,756,480	2,013,985	0.65

В нашем случае логирование можно не отключать никогда!

«Детские болезни» производительности – 3

3. Использование специфических только для PostgreSQL нестандартных форм операторов UPDATE и DELETE в нашем коде дает прирост скорости до 40% по сравнению с использованием подзапросов вида

… WHERE EXISTS (SELECT …) или … WHERE X IN (SELECT …)

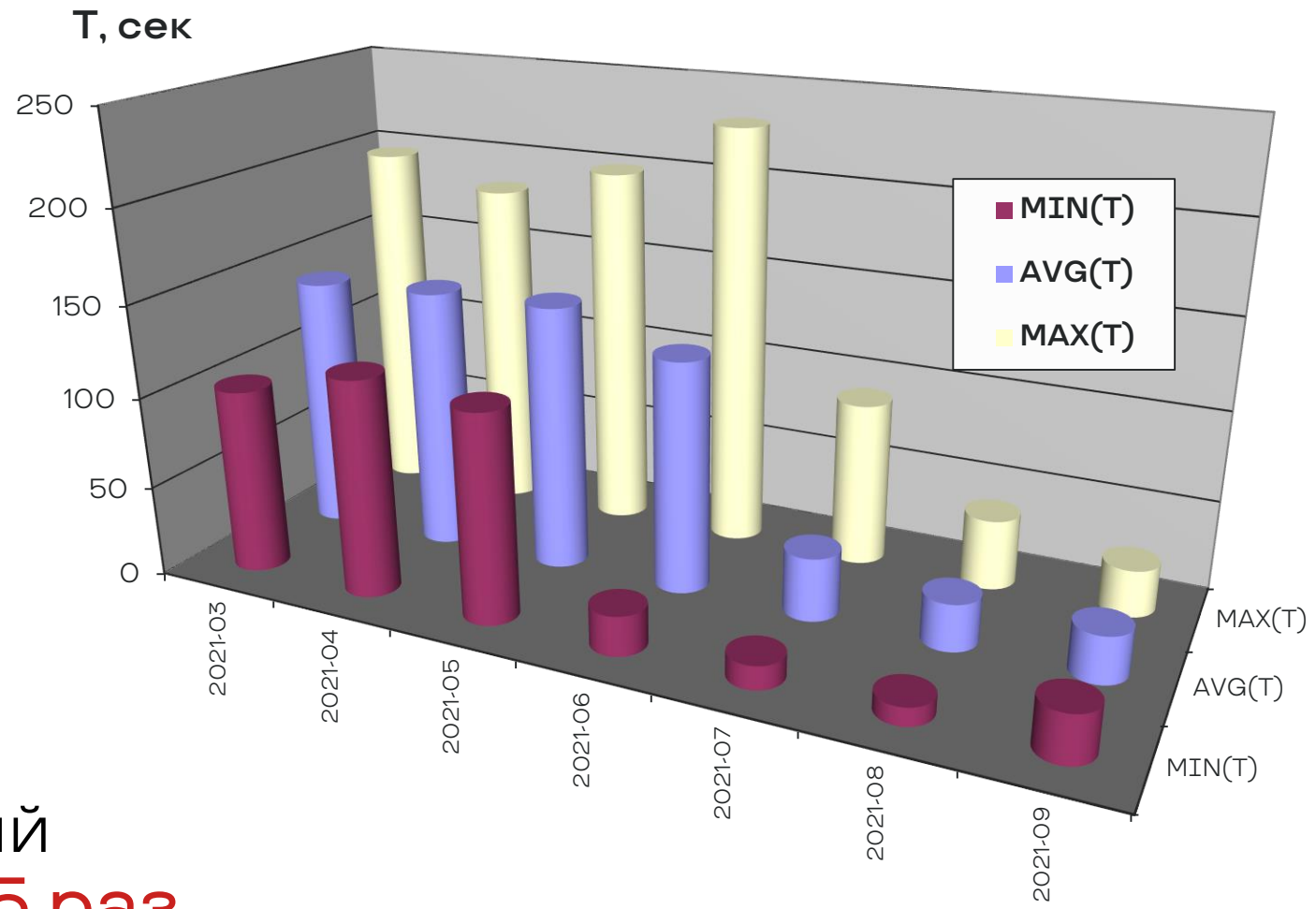
Примеры:

```
UPDATE IDs
SET    ID = Q.NEW_ID
FROM   (SELECT NEW_ID, ID FROM TMP_IDs) AS Q
WHERE (Q.ID = IDs.ID);

DELETE FROM IDs
USING  TMP_IDs
WHERE (IDs.ID=TMP_IDs.ID);
```

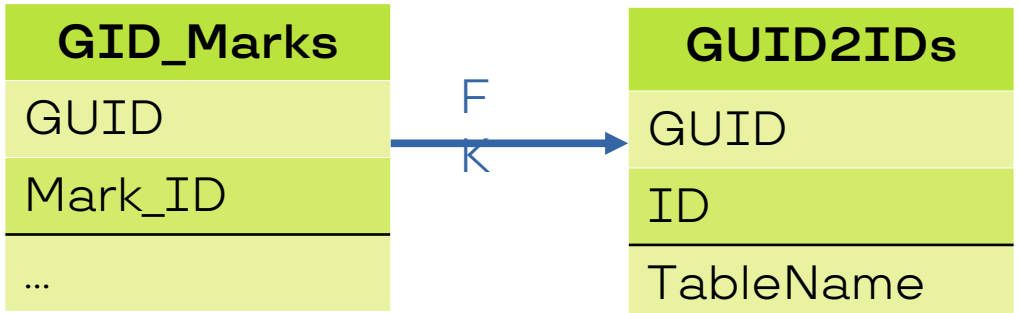
«Детские болезни» производительности — 4

4. Операции удаления постепенно стали выполняться слишком медленно — проблема с **ON DELETE CASCADE**



После принятых мер скорость удаления устаревших расписаний возросла в среднем в 5 раз

Суть проблемы с ON DELETE CASCADE




```
FOREIGN KEY (GUID)
REFERENCES  GUID2ID(GUID)
ON DELETE CASCADE
```

ванильный PostgreSQL 11.11

При выполнении оператора

```
DELETE FROM GUID2ID … ,
```

который должен был удалить 5 млн строк из GUID2ID и, заодно, из GID_Marks, оказалось, что БД выполнила 5 млн отдельных запросов вида:

```
DELETE FROM GID_Marks
WHERE GUID = '4e423f41'
```

Решение проблемы с ON DELETE CASCADE

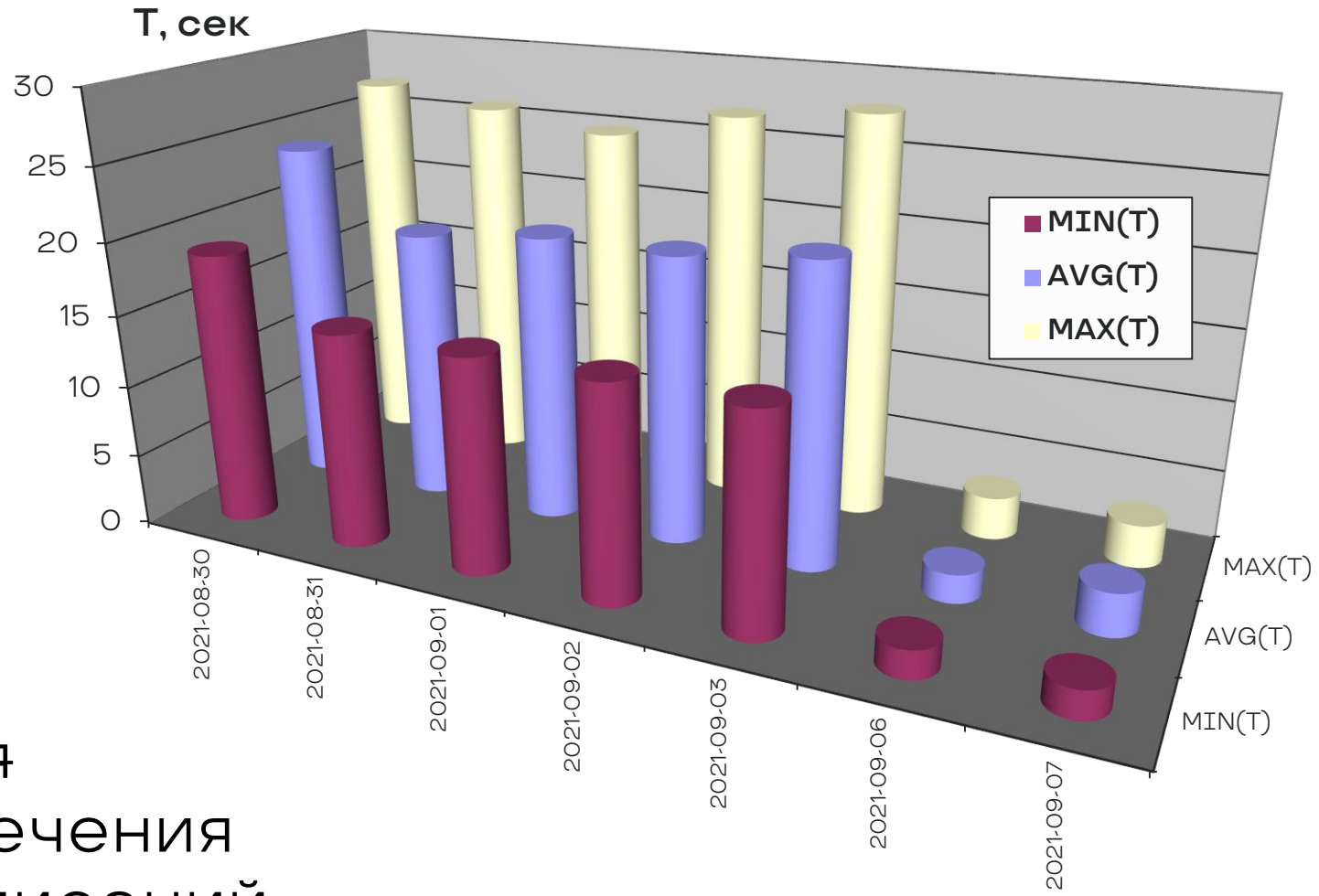
1) Разорвать связь ON DELETE CASCADE

2) Для эффективного одновременного удаления из родительской и подчинённой таблицы использовать SQL-операторы следующего вида:

```
WITH CTE_GUIDs AS
    (
      DELETE FROM GID_Marks
      WHERE (GID_Marks.TimetableID = v_TimetableID)
      RETURNING GID_Marks.GUID
    )
DELETE FROM GUID2ID
USING  CTE_GUIDs
WHERE (GUID2ID.GUID=CTE_GUIDs.GUID);
```

«Хронические заболевания» – 1

1. Первое проявление в виде проблемы с индексами: при объединении двух основных по размеру таблиц возникал **Sec Scan** вместо обращения по индексу.



После ~~исправления~~ симптоматического лечения скорость чтения расписаний возросла в среднем в 15 раз (временно).

Проявление проблемы с Sec Scan

GID_Events
Train_EID
…

142 млн.строк

TMP_Trains
Train_EID
…

3682 строки

GID_Events имеет индекс по Train_EID

```
SELECT E.*
FROM   GID_Events E INNER JOIN
       TMP_Trains T
       ON(T.Train_EID=E.Train_EID);
```

```
QUERY PLAN
Hash Semi Join  (cost=170.84..4792890.47 rows=11787245 width=286) (actual time=13254.652..135489.742
rows=816779 loops=1)
  Hash Cond: (e.train_eid = t.train_eid)
  ->  Seq Scan on gid_events e  (cost=0.00..4287162.24 rows=142637824 width=286) (actual
time=0.217..117560.619 rows=136429750 loops=1)
  ->  Hash  (cost=124.82..124.82 rows=3682 width=8) (actual time=2.882..2.892 rows=3682 loops=1)
        Buckets: 4096  Batches: 1  Memory Usage: 176kB
        ->  Seq Scan on tmp_trains t  (cost=0.00..124.82 rows=3682 width=8) (actual time=0.036..2.109
rows=3682 loops=1)
Planning Time: 5.428 ms
Execution Time: 135523.087 ms
```

Временное решение проблемы с Seq Scan

Увеличить глубину сбора статистики по колонке с ПК:

```
ALTER TABLE GID_Events ALTER COLUMN Train_EID SET STATISTICS 10000;
ANALYZE GID_Events;
```

```
QUERY PLAN
Nested Loop  (cost=134.59..774248.40 rows=193491 width=286) (actual time=30.496..6586.509 rows=816779
loops=1)
  ->  HashAggregate  (cost=134.03..170.84 rows=3682 width=8) (actual time=3.238..7.687 rows=3682 loops=1)
        Group Key: t.train_eid
        ->  Seq Scan on tmp_trains t (cost=0.00..124.82 rows=3682 width=8) (actual time=0.034..1.865
rows=3682 loops=1)
  ->  Index Scan using gid_events_pk on gid_events e  (cost=0.57..209.70 rows=53 width=286) (actual
time=0.762..1.674 rows=222 loops=3682)
        Index Cond: (train_eid = t.train_eid)
Planning Time: 6.667 ms
Execution Time: 6619.910 ms
```

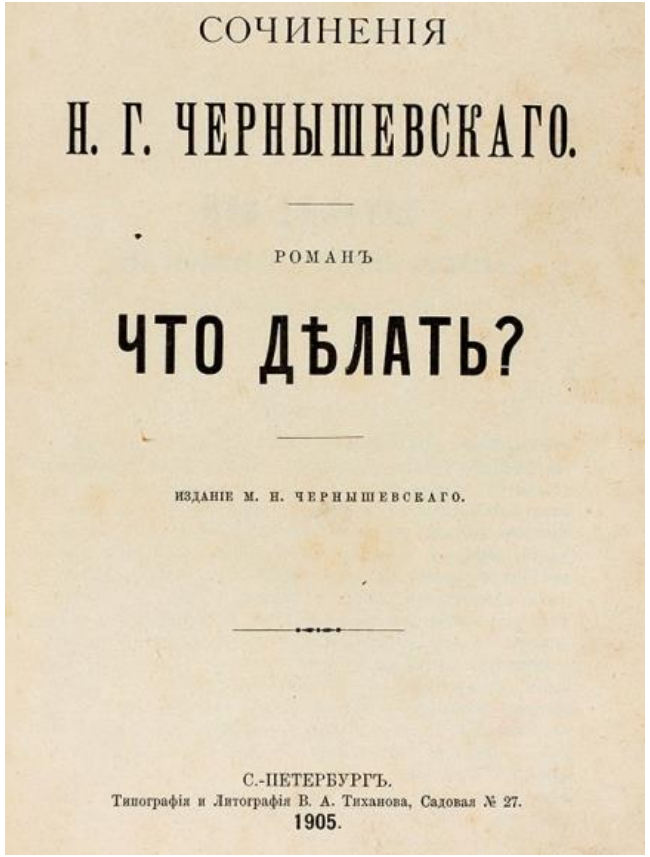
Ускорение в 20 раз!

«Хронические заболевания» – 2

2. Со временем скорость работы снизилась до неприемлемых значений, при объединении двух основных по размеру таблиц снова возникал `Sec Scan` вместо обращения по индексу.

Прежний метод не помогал.

Перестроение индексов не помогало тоже.


СОЧИНЕНІЯ

Н. Г. ЧЕРНЫШЕВСКАГО.

РОМАНЪ

ЧТО ДѢЛАТЬ?

ИЗДАНІЕ М. Н. ЧЕРНЫШЕВСКАГО.

С.-ПЕТЕРБУРГЪ.
Типографія и Литографія В. А. Тиханова, Садовая № 27.
1905.

Выход найден!

В Oracle были **хинты** – поищем их в PostgreSQL
и **заставим** использовать индексы!

«Хинты» в PostgreSQL (нет)

- Хинтов в стиле Oracle в PostgreSQL нет, но давать советы планировщику все же можно.
- Менять нужно только настройки текущей сессии и не забыть восстанавливать исходное состояние настроечного параметра.

```
SET LOCAL enable_seqscan = OFF; -- Порицаем использование Sec Scan
… -- Выполняем запрос
SET LOCAL enable_seqscan TO DEFAULT; -- Возвращаем, как было
```

Ненадолго помогает, но так как причина не выявлена и не устранена, вскоре снова становится плохо…

Настоящая причина Sec Scan — autovacuum

Настройки **autovacuum** по умолчанию вызывают такой эффект в случае нашей БД.

Потребовалось изучить вопрос с **autovacuum** и внести изменения в настройки по умолчанию!

Спасибо Егору Рогову и Алексею Лесовскому!
Их статьи по теме очень помогли.

https://habr.com/ru/company/postgrespro/blog/452762/
https://dataegret.com/category/autovacuum/

Симптом проблемы с autovacuum

- Простой SELECT из интерфейсной UNLOGGED-таблицы внезапно стал занимать 95% времени работы процедуры независимо от числа записей в этой таблице (плоть до 0).
- По состоянию UNLOGGED-таблиц мы видим, что autovacuum к ним не применяется!

table_name	n_live_tuples	n_dead_tuples
tmp_trains	0	521 457
tmp_timetablepoints	0	15 048 410
tmp_traincalendars	0	8 544 236

Зачем настраивать autovacuum?

- До 60% содержимого основных таблиц в нашей БД обновляется за неделю.
- Настройки **autovacuum** по умолчанию таковы, что отработать интенсивно обновляемые таблицы он не успевает. Кстати, начиная с версии 12 настройки по умолчанию более активные.
- Статистика по таблице и индексу собирается в ходе выполнения **autovacuum**; в итоге статистика не собиралась и устаревала; поэтому в запросах переставали использоваться индексы и возникал Sec Scan.
- Параметры работы **autovacuum** можно настроить индивидуально для отдельных таблиц.

Настройка autovacuum

- Устанавливаем индивидуальные значения для часто обновляемых таблиц командой `ALTER TABLE trains SET(…)`.

Параметр	Значение	Назначение параметра
`autovacuum_vacuum_scale_factor`	0	Пороговая доля строк в таблице для принятия решения о начале вакуумирования
`autovacuum_vacuum_threshold`	10000	Пороговое число «мертвых» строк в таблице для принятия решения о начале вакуумирования
`autovacuum_vacuum_cost_delay`	5	Перерыв между циклами автовакуума, мс (вместо 20 мс по умолчанию в 11-й версии)
`autovacuum_vacuum_cost_limit`	1000	Стоимость одного цикла автовакуума (вместо 200 по умолчанию)

Помогает надолго, но не навсегда…

Паллиатив: pg_repack

- 1 раз в месяц выполняем техобслуживание с краткосрочным простоем путем вызова **pg_repack** для ликвидации последствий «распухания» часто обновляемых таблиц и индексов.
- Возможны проблемы с репликацией.

```
TRUNCATE tmp_trains; -- Очищаем UNLOGGED-таблицы

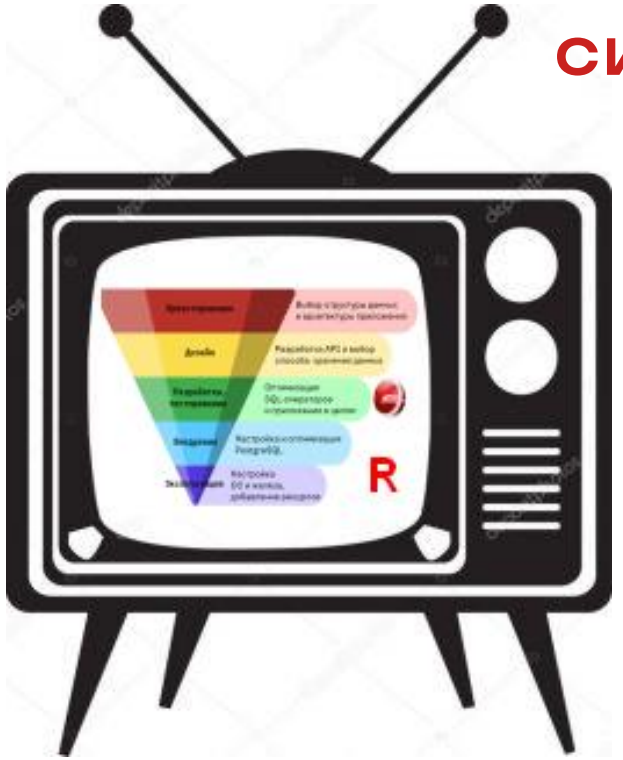
# Пересоздаём часто обновляемые таблицы, включая
UNLOGGED

$ pg_repack -t trains
```

Пока помогает. Продолжаем наблюдения...

Проектируем систему с «нуля»

Макромодель ЭЛЬБРУС–М для прогноза продвижения поездопотоков и оценки инфраструктурных и управляющих решений.



Можно повлиять на производительность системы на двух недоступных ранее уровнях!

- Выбор структуры данных и архитектуры приложения.
- Разработка API и выбор способа хранения данных.

Выбор структуры данных и архитектуры

- Выполнена глубокая нормализация данных о расписаниях

ОБЪЕКТЫ РАСПИСАНИЙ

1) Поезда
2) Точки ниток поездов
3) Календари

Нормализация

ОБЪЕКТЫ РАСПИСАНИЙ

1) Нитки поездов
2) Точки ниток поездов
3) Поезда
4) Изменения поездов по ходу движения
5) Календари
6) Маршруты
7) Перегоны маршрутов

Объем данных одного расписания уменьшился в 10-100 раз

Выбор API и способа хранения данных

- Активное использование партицирования.
- Шаблоны для создания временных таблиц.
- Предварительный расчет и хранение промежуточных данных для аналитических запросов.
- Отказ от использования UNLOGGED-таблиц в пользу традиционных временных таблиц PostgreSQL – это выигрыш до 40% в производительности на тяжелых ХП. Для этого перед вызовом ХП вызываем функцию `prepareCall('`*ИМЯ_ПРОЦЕДУРЫ*`')` для создания необходимых для работы API временных таблиц в данной сессии.

Изменения будут распространены и на родительскую систему!

Продолжение следует!

- Импортозамещение состоялось и приносит пользу.
- Новое приложение и новый тип БД – новые вызовы.
- Скучать не придется!

ВНИИЖТ
НЦ ЦМПЭ

P.S. Чего бы хотелось в PostgreSQL

Чего бы хотелось в PostgreSQL

- Блокировки строк с заданным таймаутом (a-la Oracle):
- ```
  SELECT ID FROM IDs FOR UPDATE WAIT 60;
  ```

- Ослабление чрезмерной строгости в указании типов для параметров хранимых функций при их вызове. Необходимость писать `f ( 'A'::VARCHAR)` в качестве аргумента функции вместо `f ( 'A')` на мой взгляд – перебор
  ```

Чего бы хотелось в PostgreSQL

- Усеченный функционал автономных транзакций хотя бы для записи логов
- В утилите pg_dump иметь возможность:
- — брать параметры не из командной строки, а из файла
- — задавать дополнительные фильтры для данных, извлекаемых из отдельных таблиц наподобие
- `--sqlfilter=IDS:"ID<10"`
- — сортировать объекты (хотя бы по именам) для того, чтобы иметь возможность сравнивать две БД при выгрузке в файл только метаданных

Спасибо!

О докладчике

anfinogenov.anatoly@vniizht.ru
+7 (499) 262-45-06

АО «ВНИИЖТ» (АО «Научно-исследовательский институт железнодорожного транспорта»)

Анфиногенов Анатолий Юрьевич

Заместитель директора научного центра – начальник отдела разработки ПО, кандидат физико-математических наук

Работаю уже два десятка лет во ВНИИЖТ над задачами имитационного моделирования и оптимизации железнодорожных перевозок.

Проектировал, разрабатывал и сопровождал БД и серверное ПО для этих задач (Postgres, Oracle, C++, Python), чем и продолжаю заниматься.