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

Анатолий Анфиногенов

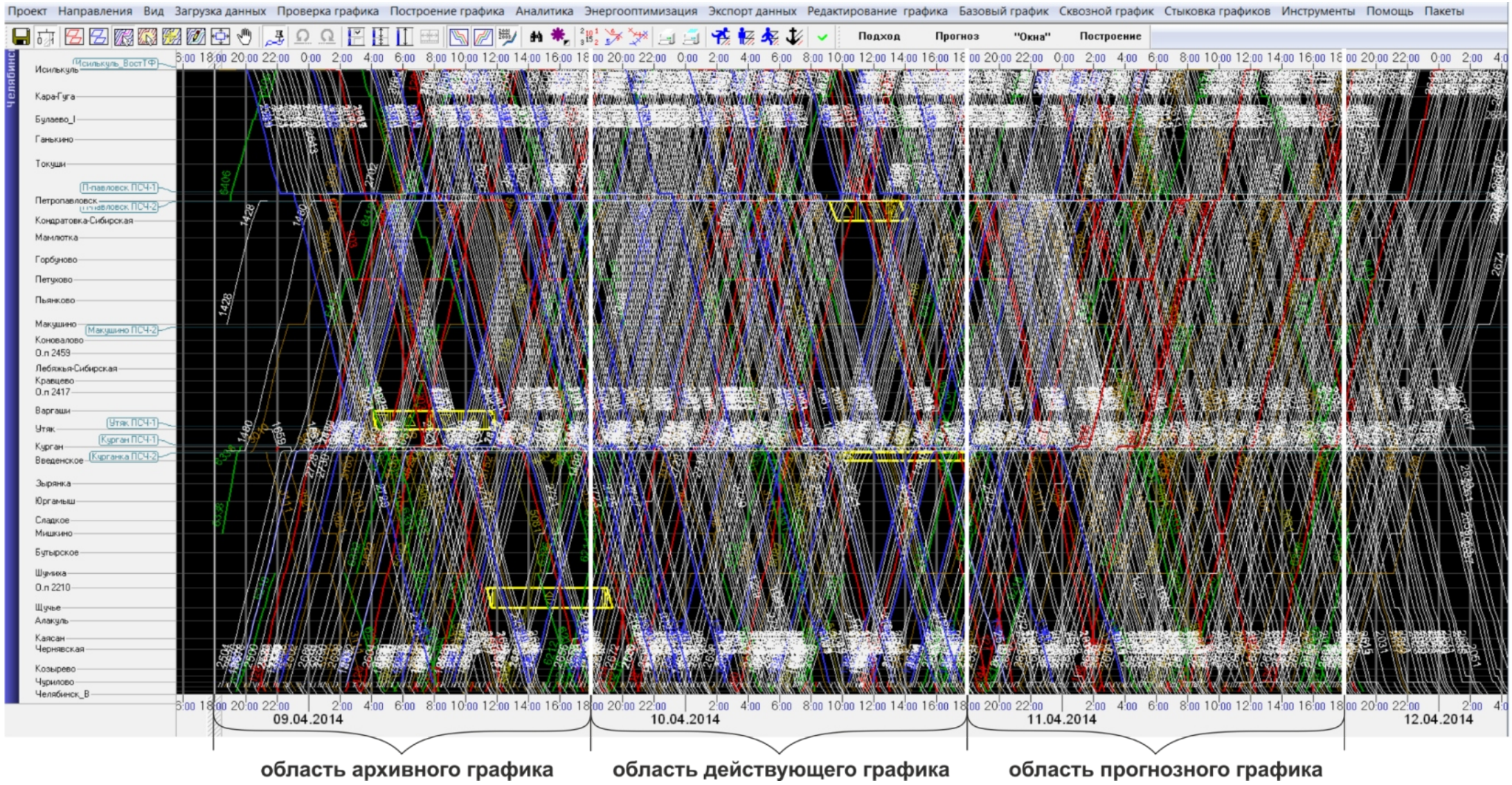
Что это и зачем это нужно?

ЭЛЬБРУС – это система для планирования движения грузовых поездов по энергооптимальным расписаниям

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 расписание поездов изнутри

- Для каждой из 16 железных дорог России каждый день разрабатывается и используется несколько расписаний различных типов и назначений
- До 2 млн объектов нескольких десятков типов в одном расписании
- Объем данных до 500 Мбайт на одно расписание
- В оперативной базе одновременно содержится порядка 500 активных расписаний + архивная БД
- **Очень крупные, но относительно редкие транзакции**

Начальные условия (2018 год)

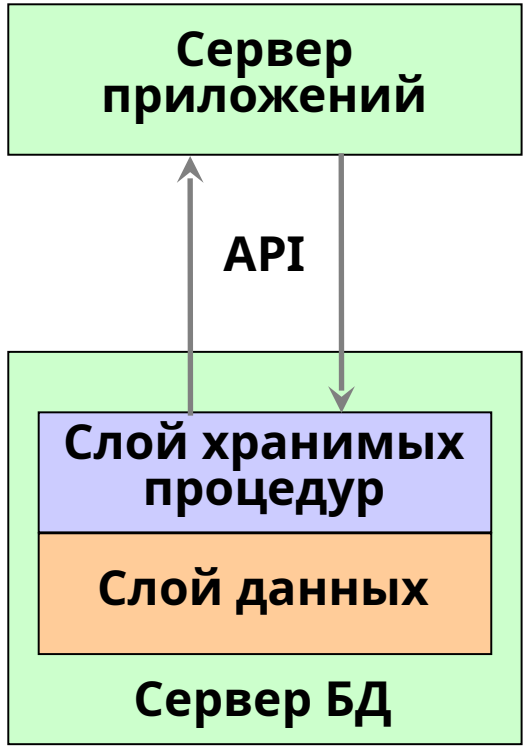
- Использовалась СУБД Oracle Database 11g Standard Edition One
- Всего более 100 взаимодействующих серверов
- Более 50 серверов БД
- Эксплуатация 24/7; регулярные обновления серверных приложений 3-6 раз в год, включая приложения БД

Архитектура: трехзвенная

- **Толстый клиент:** Windows, C++, MQ
- **Тонкий клиент:** GWT, Angular
- **Сервер приложений:** Java, OpenJDK, Tomcat, ActiveMQ
- **Сервер БД:** Oracle 11g SE1,
  60000 строк хранимых процедур PL/SQL

Почему хранимые процедуры?

- Логика хранения отделена (с помощью API) от бизнес-логики приложения
- Можно вносить изменения в структуру БД без изменения сервера приложений (в пределах API)
- Возможно обновление распределенного ПО за счет версионности API
- Можно диагностировать, логировать, отлаживать и профилировать приложение без остановки сервиса



Особенности нашей базы данных

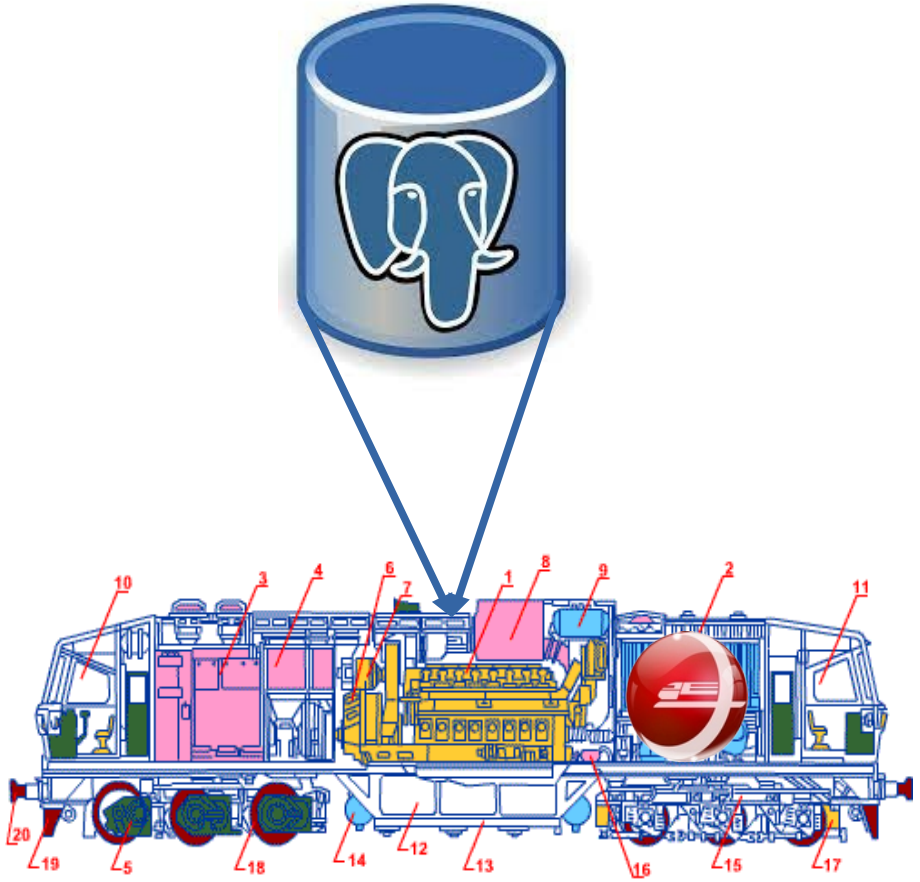
- Одна схема, 250 таблиц, 50 Гб оперативных данных
- 200 хранимых процедур на PL/SQL (60000 строк) в нескольких пакетах
- Автономные транзакции для логирования API
- Временные таблицы как средство обмена данными большого объема (сотни Мбайт) с сервером приложений при вызовах хранимых процедур
- Stabndy средствами приложения как часть API из-за ограничения редакции Oracle SE

ВНЕЗАПНО: импортозамещение

- Выбор PostgreSQL как ближайшей к Oracle из всех СУБД, подходящих для импортозамещения
- Язык pl/pgSQL максимально близок к PL/SQL
- Нет проблем с русскоязычной документацией, поддержкой, сертификацией и Реестром отечественного ПО
- Есть замечательная команда Postgres Professional и большое и дружелюбное сообщество разработчиков
- Есть примеры успешных миграций Oracle—>PostgreSQL

Что от нас требовалось?

Поменять на ходу двигатель у локомотива так, чтобы пассажиры этого не заметили, то есть перенести таблицы, данные и хранимые процедуры так, чтобы все это продолжало работать уже на базе СУБД PostgreSQL с тем же API прозрачно для приложения



Граничные условия

- Время — один год (2019-й)
- Силы – три человека: двое на PostgreSQL и один – на адаптацию сервера приложений
- Сервер приложений работает с прежним API и с БД Oracle, и с БД PostgreSQL
- Развитие функциональности системы продолжается и нужно выпускать новые версии для двух СУБД
- PostgreSQL – ванильный (не спрашивайте, почему!) версии 11.5

Но были и хорошие новости

- Годом ранее (в 2018) вспомогательная подсистема (20 таблиц и 11 процедур) была перенесена на PostgreSQL 10.6 — это дало первый опыт переноса, пока что полностью вручную

- Был получен первый опыт эксплуатации БД на PostgreSQL

- Переходом на PostgreSQL в своей предметной области мы занялись одними из первых;
  мы восприняли это как шанс сделать все самостоятельно и правильно (с нашей точки зрения)

(c) А.Куделин

Но что делать дальше?!

“Слона едят по частям”

Планируем переход: основные шаги

1. Разработка общей технологии миграции
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 оптимизация производительности приложения БД

Планируем переход: основные шаги

7. Разработка технологического процесса переключения
8. Разработка эксплуатационной документации и инструкций для администраторов
9. Обучение администраторов
10. Организация сопровождения приложения и БД, включая мониторинг
11. Разработка и внедрение технологии выполнения обновлений

1) Создаем технологию миграции

- Оценка объема работ
  (*ура! – переносить из 200 процедур надо только 150!*)
- Выбор версии и редакции PostgreSQL
  (*11.5, ванильная*)
- Согласуем с Заказчиком этапы и порядок работ в ходе миграции
- Планируем вычислительные ресурсы и заблаговременно организуем их выделение
- Продумываем порядок переноса данных

2) Учимся работе с PostgreSQL

- **PostgreSQL – это другая СУБД! Это – не Oracle**, хотя местами и похоже. Как следствие, **PL/pgSQL – это не PL/SQL**!
- Учимся правильно работать именно с PostgreSQL, не пытаясь механически воспроизвести рецепты из Oracle
- Узнаем об EXSTENSIONS; выбираем необходимые для проекта; учимся устанавливать и настраивать ОС, СУБД и расширения
- Выбираем инструмент для работы с СУБД PostgreSQL (*купили лицензии на EMS SQLManager for PostgreSQL*)

Читая классиков

Не надо бояться использовать специфические средства конкретной СУБД, — за них заплачено немало денег.

В каждой СУБД есть свой набор уникальных возможностей,
и в любой СУБД можно найти способ выполнить необходимое действие.

Используйте в текущей СУБД лучшее…

Том Кайт

3) Инфраструктурные workarounds–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
- Нас **спасает** расширение **pg_variables; спасибо, Иван Фролков!** В переменной храним открытое соединение, и логи начинают писаться с приемлемой скоростью

3) Инфраструктурные workarounds–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

3) Инфраструктурные workarounds–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

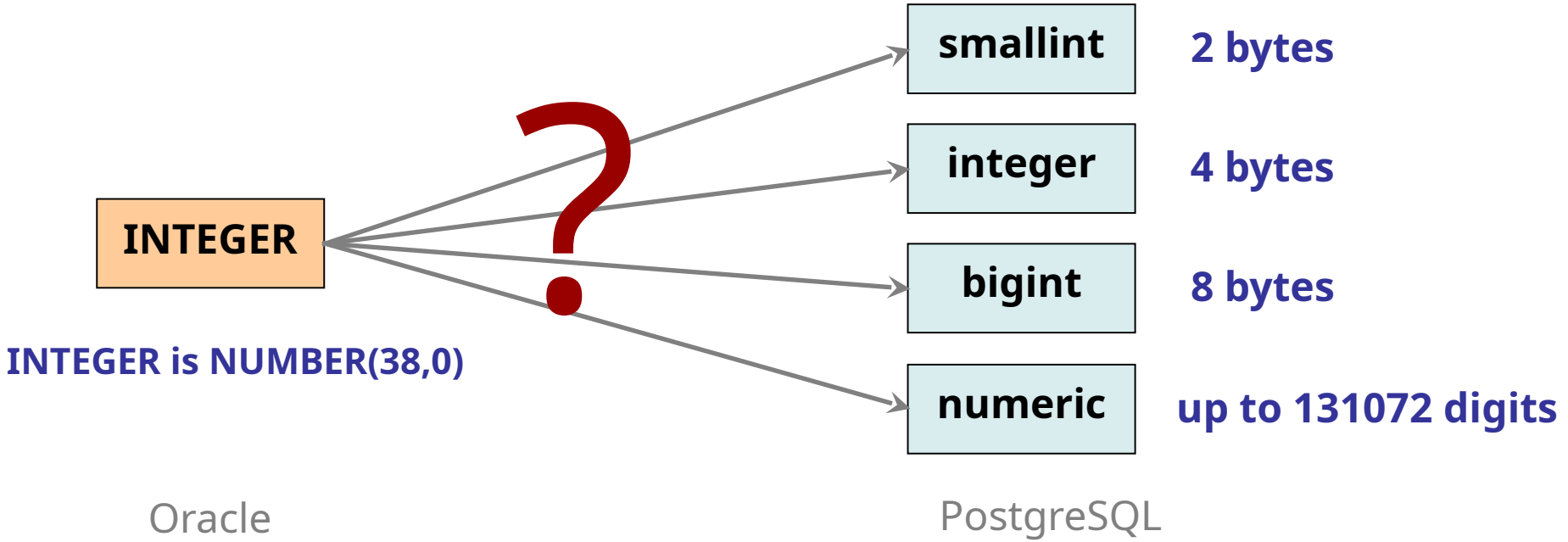
3) Workarounds–4: Extensions

Расширение	Назначение в проекте	Штатное?
dblink	Эмуляция автономных транзакций	да
postgres_fdw	Связь с архивными серверами	да
pgcrypto	Генерация GUID	да
pg_repack	Обслуживание БД	да
plpythonu	Работа с файлами в ФС сервера	да
pg_variables	Эмуляция автономных транзакций	**нет** (да в Pro)
oracle_fdw	Репликация данных при переходе	**нет** (да в Pro)
plpgsql_check	Проверка хранимых процеур	**нет**
pg_stat_satements	Мониторинг производительности БД	да

Нештатные расширения затрудняют обновление СУБД!

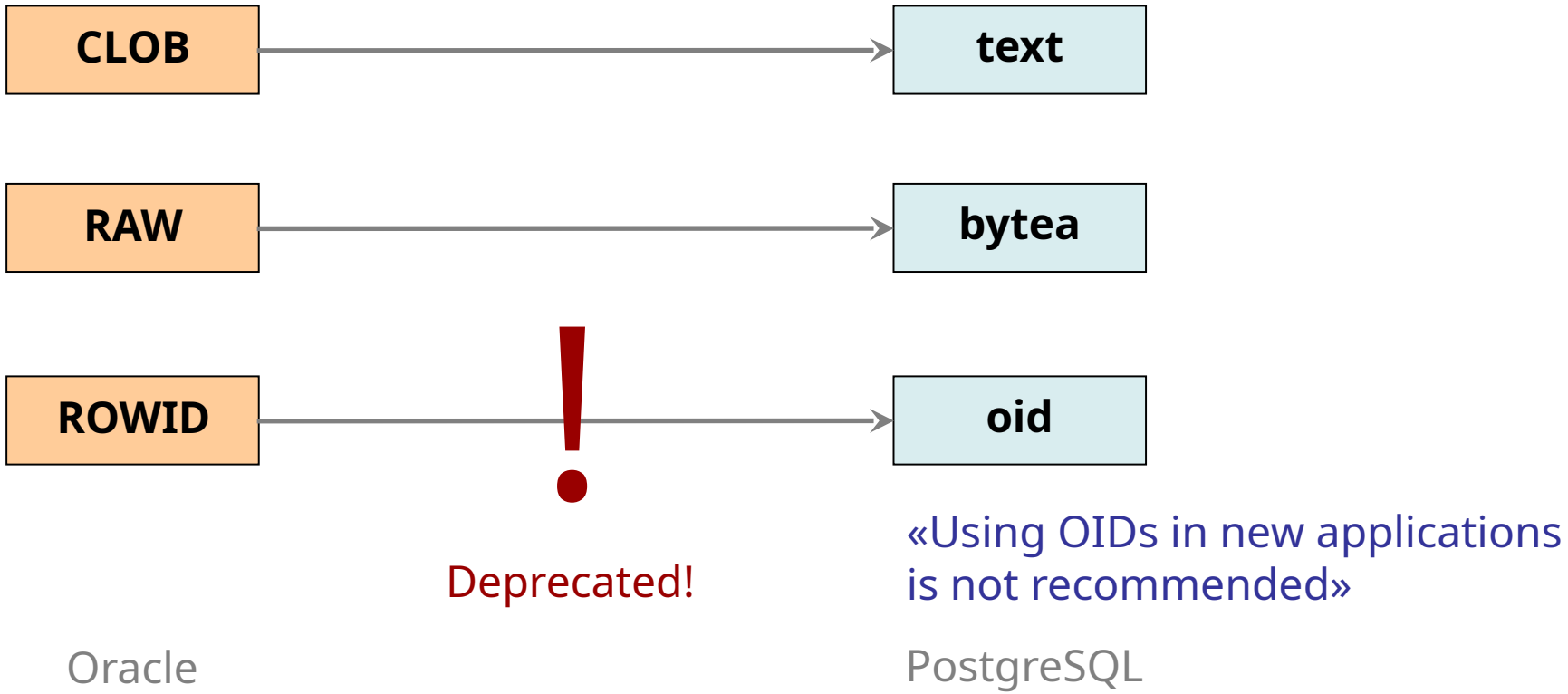
4) Перенос таблиц, VIEW и т.п...

- Главный вопрос при переносе таблиц – **как соотнести типы данных Oracle и PostgreSQL**. Этот **вопрос не решается механически** для многих полей даных



4) Перенос таблиц, VIEW и т.п...

- Еще один пример неполного соответствия типов



5) Перенос хранимых процедур – 1

- Как быть с отсутствием оператора **MERGE?**

  ⇨ В большинстве случаев годится **INSERT(…) ON CONFLICT DO UPDATE (…)**

- Как быть с отсутствием **PACKAGES**?

  ⇨ Используется **схема с названием пакета**

- **Не создавайте свои объекты в схеме public**: в неё пишут EXTENSIONS; возможен конфликт имен!
- **DDL не вызывает автоматический COMMIT**!

  ⇨ DDL можно вызывать внутри функций, а также не забывать фиксировать транзакцию после создания функции

5) Перенос хранимых процедур – 2

- Функции в PostgreSQL могут иметь одно имя, но разный набор аргументов (зачастую ничтожно отличающийся по типу). Это – потенциальный источник больших проблем. **Обеспечьте уникальность имен инструментально!**
- Обработка EXCEPTIONS отличается.

  ⇨ Используйте свою систему именованных ERRCODE, например:

```
RAISE EXCEPTION '%', v_ErrorMessage
    USING ERRCODE = 'EL023', HINT = 'ARCHIVE data not found!';
```

- Осторожнее с CTE – они могут материализоваться!
  В версиях PostgreSQL 12+ это управляемо

5) Перенос хранимых процедур – 3

- В Oracle **пустые строки и NULL** – одно и то же;
  в PostgreSQL – **не одно и то же**!

```
(" IS NULL) = TRUE
```
Oracle

```
(" IS NULL) = FALSE
```
PostgreSQL

- В PostgreSQL **NULL обнуляет строку**; в Oracle – нет!

```
select 'Hi!'||NULL AS Str from dual
```

```
STR
-----
Hi!
```
Oracle

```
select 'Hi!'||NULL AS Str;
```

```
STR
-----

```
PostgreSQL

- Всегда используйте определение типа через объект:


P_GUID GUID2ID.GUID%TYPE


P_GUID VARCHAR

5) Перенос хранимых процедур – 4

- Как силами двух человек за полгода перенести 150 хранимых процедур из Oracle в PostgreSQL?
  Без автоматизации – никак!

- Как частично автоматизировать перенос кода хранимых процедур?
  Спасение: **ora2pg**! **Спасибо, Gilles Darold!**

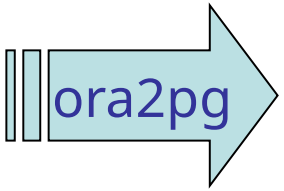
- Нет проверки корректности на этапе компиляции хранимых процедур (к которой мы привыкли в Oracle)?
  Используйте **plpgsql_check** (если подружите его с временными таблицами)! **Спасибо, Павел Стехуле!**

5) Перенос хранимых процедур – 5

- Как надо использовать **ora2pg**?

```
IF ( A = B ) THEN
    do_smth_1();
ELSIF (C IS NULL) THEN
    IF (NVL(D,0) = 0) THEN
       do_smth_2();
    ELSE
       do_smth_3();
    END IF;
ELSE
    RETURN 0;
END IF;
```

Oracle




```
IF ( A = B ) THEN
     PERFORM do_smth_1();
ELSIF (C IS NULL) THEN
   IF (COALESCE(D,0) = 0) THEN
       PERFORM do_smth_2();
    ELSE
       PERFORM  do_smth_3();
    END IF;
ELSE
   RETURN 0;
END IF;
```

PostgreSQL

- Результат трансляции – только канва (помним про MERGE, NVL2(), алиасы в UPDATE/DELETE, обработка комментариев и многое другое); правим руками

6) Настройка производительности – 1

- Сравним максимальные времена выполнения одной из ключевых процедур на одних и тех же данных на одном и том же не самом быстром тестовом сервере:

10 минут	**19 часов**
`Oracle`	`PostgreSQL`

!

- **Еще рано впадать в отчаяние – пора заняться оптимизацией**
- Вот теперь-то нам и пригодится встроенное логирование и профилирование хранимых процедур

Профилируем хранимые процедуры

- Возможность профилировать хранимые процедуры должна быть заложена **при проектированияи**
- Процедура разбита на **этапы**; 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 пробемных запросов изнутри хранимой процедуры** (это было в приложении для Oracle, но это невоможно для ванильной PostgreSQL — печаль)

Профилируем хранимые процедуры

Ускорение, раз	Этап процедуры	Т, мс (до 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 saveTrainTimetable
- Каждый **этап** — один или несколько сходных по типу SQL-операторов в теле хранимой процедуры
- Применяя приемы оптимизации, можно оценивать их эффективность путем сравнения времен выполнения этапов
- На данном примере процедура ускорилась в 9 раз: было 171 с, стало 19 с
- Основной эффект достигнут на этапах «4.1)», «3.2.S)» и «9)»

Профилируем хранимые процедуры

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

6) Настройка производительности–2

За счет чего повышалась производительность хранимых процедур? Наш опыт:

1. Использование UNLOGGED-таблиц вместо временных таблиц – удобно и похоже на то, как было в Oracle, но отсутствие индексов и статистики замедляет работу.

Решение: использовать CTE в запросах с UNLOGGED-таблицами

Результат: **ускорение в 3 раза**
(но **6 часов вместо 10 минут все равно плохо**)

6) Настройка производительности–3

2. Используем UNLOGGED-таблицы только для приема данных от сервера приложений, а далее создаем временные таблицы PostgreSQL внутри функции и работаем уже с ними

Решение:

```
CREATE TEMPORARY TABLE TMP_Input
    ON COMMIT DROP AS SELECT * FROM UL_Input;
```

Результат: **ускорение в 600 раз**

(стало 15 минут вместо 10 минут – **почти паритет**)

6) Настройка производительности–4

3. А если еще поднажать? В отличие от Oracle, в PostgreSQL внутри функций можно не только создавать таблицы, но и собирать статистику

Решение: `ANALYZE TMP_InputData;`

Результат: **ускорение в 600 раз**

(стало 15 минут вместо 10 минут – **почти паритет**)

4. Что еще можно сделать? **Отказаться от ошибочного архитектурного решения и сразу использовать временные таблицы PostgreSQL**, создавая их перед вызовом с помощью **prepareCall()**

6) Настройка производительности–5

5. 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:

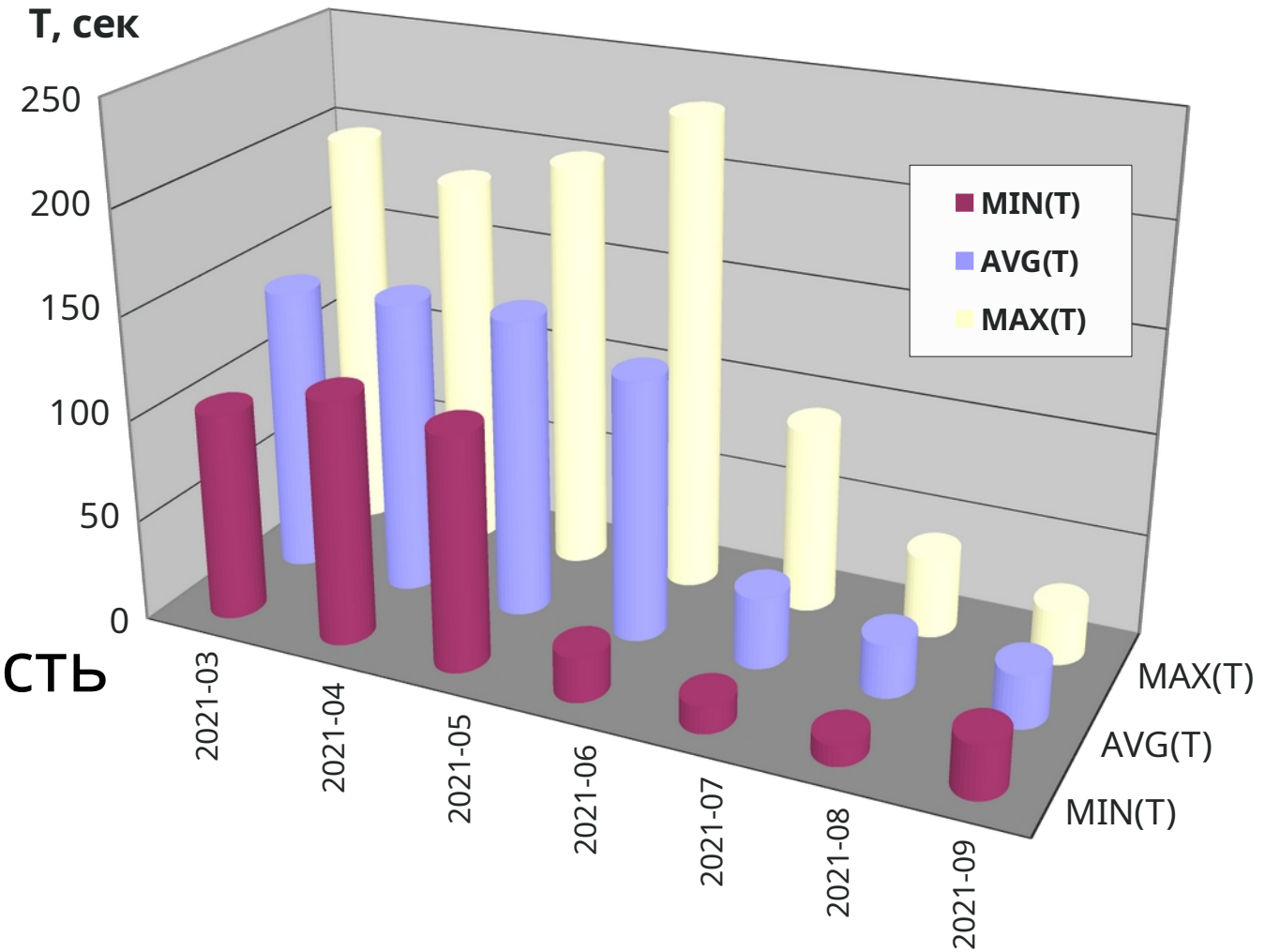
```
UPDATE IDs
SET    ID = Q.NEW_ID
FROM   (SELECT NEW_ID, ID FROM TMP_IDs) AS Q
WHERE (Q.ID = IDs.ID);

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

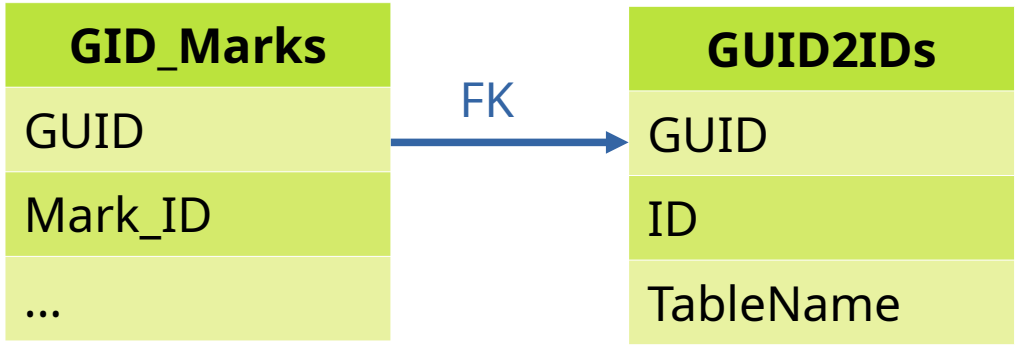
6) Настройка производительности – 6

6. Два года эксплуатации позволили открыть новые резервы для повышения производительности — **проблему** с ON DELETE CASCADE

После исправления 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 11.9

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'
```

Читая классиков

Когда приложение, без проблем работавшее в СУБД «А», не работает или работает весьма странно в СУБД «Б», сразу же возникает мысль, что "СУБД «Б» — плохая".

*Правда, однако, в том, что СУБД «Б» работает **иначе**. Ни одна из СУБД не ошибается и не является "плохой" — они просто разные.*

Знание и понимание особенностей их работы поможет успешно решить подобные проблемы..

Том Кайт

Решение проблемы ON DELETE CASCADE

1) Разорвать связь `ON DELETE CASCADE`

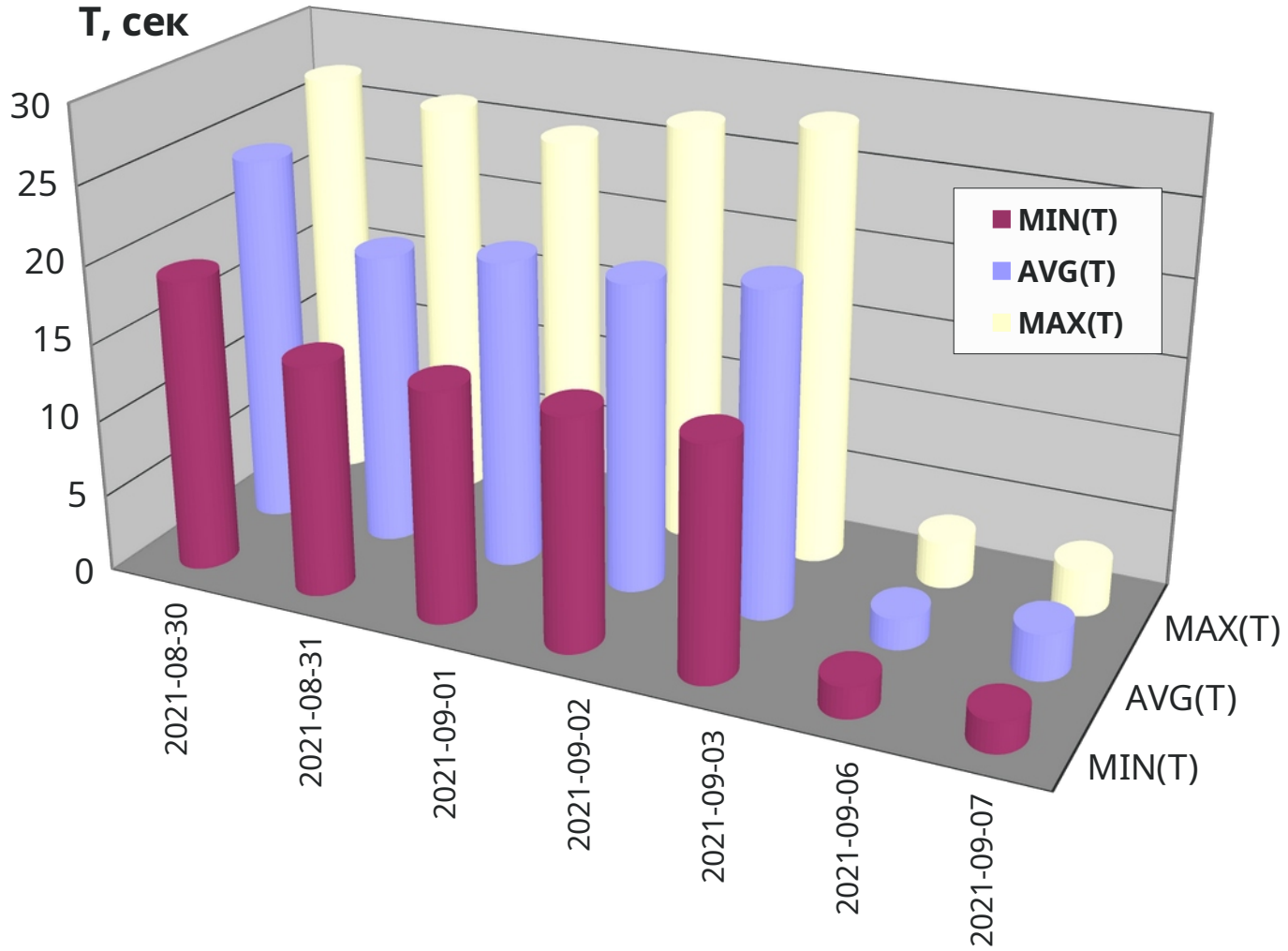
2) Для эффективного одновременного удаления из родительской и подчиненнй таблицы использовать SQL-операторы следующего вида:

```
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)
```

6) Настройка производительности – 7

7. Решение проблемы с индексами: при объединении двух основных таблиц возникал Sec Scan вместо использования индекса

После исправления скорость чтения расписаний **возросла в среднем в 15 раз**



Суть проблемы с **Sec Scan**

GID_Events
Train_EID
…

142 млн.строк

TMP_Trains
Train_EID
…

3682 строки

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

GID_Events имеет индекс по 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
```

Решение проблемы **Sec Scan**

1) Увеличить глубину сбора статистики по колонке Train_EID:

```
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!

6) Настройка производительности – 8

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:

TZ_bHash_getHashInfo
bHash
TableName

650 строк

bGUID2Hash
bHash
TableName

1.1 млн.строк

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

bGUID2Hash имеет индекс (PK) по bHash

```
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 tz_bhash_gethashinfo 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*

```
CREATE UNIQUE INDEX ELECT 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 tz_bhash_gethashinfo 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!

6) Настройка производительности – 9

Кстати: *А сколько стоит логирование?*

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

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

В нашем случае логирование можно не отключать никогда!

7) Подготовка к переключению

- Как перенести данные из Oracle в PostgreSQL?
- Как обеспечить **переходный период** – параллельную эксплуатацию приложений для Oracle и PostgreSQL с возможностью быстрого возвращения на Oracle в случае обнаружения сбоев или крупных и неприятных сюрпризов?
- Как поддерживать синхронность данных в БД Oracle и PostgreSQL в течение переходного периода?
- Какую выбрать технологию резервного копирования? (**pg_probackup**)

8) Разработка документации

- Разработка **подробных пошаговых инструкций на русском языке** по установке системы, включая установку и настройку ОС, СУБД, EXTENSIONS, импорта стартовой БД и всех объектов, а также настройки приложения БД
- Разработка подробного руководства администратора системы на русском языке, включая администрирование БД (**резервное копирование и восстановление, горячее резервирование, настройка параметров, мониторинг, техобслуживание** и т.п.)

9) Обучение администраторов

- Практическая отработка **подробных пошаговых инструкций** на русском языке из п.8) с внесением в них изменений и уточнений
- Проведение очного и дистанционного обучения
- Проведение семинаров на базе ВНИИЖТ

10) Поддержка и обновление ПО – 1

- В части обновления прикладного ПО БД **PostgreSQL обладает огромным преимуществом перед Oracle: операторы DDL не вызывают автоматический COMMIT**
- Обновление прикладного ПО БД в Oracle: восстановление из резервной копии после первого же оператора ALTER TABLE … в случае сбоя при обновлении
- Обновление прикладного ПО БД в PostgreSQL: в случае сбоя при обновлении восстановление к исходному состоянию производится одной командой ROLLBACK

10) Поддержка и обновление ПО – 2

- В части обновления ОС и СУБД ванильный PostgreSQL в нашей конфигурации обладает серьезным **недостатком**: некоторые **расширения** (pg_variables, oracle_fdw, plpgsql_check) **требуется компилировать из исходных кодов**, поэтому **обновляться командой yum update нельзя**

- Выход – использование PostgresPro

11) Ввод в эксплуатацию

- С октября 2019 года система ЭЛЬБРУС на базе СУБД PostgreSQL работает в режиме постоянной эксплуатации на одной из 16 железных дорог России
- Конечные пользователи ничего (плохого) не заметили; быстродействие и надежность на тех же вычислительных мощностях не уступают исходным
- На оставшихся 15 дорогах действовал переходный период и сервера ЭЛЬБРУС на базе СУБД PostgreSQL работали в качестве горячего резерва
- **С 01.06.2021 переходный период окончен; ЭЛЬБРУС работает на PostgreSQL на всех железных дорогах**

Эпилог

- Подведем итоги – **миграция** ЭЛЬБРУС на СУБД PostgreSQL **завершилась успешно**
- В 2019-2021 годах мы решали вопросы **сопровождения приложения и управления быстродействием** как приложения, так и БД в целом: перестроение индексов, оптимизация системы хранения, рефакторинг и т.п.
- Другой вопрос – организация обновлений до следующей мажорной версии с учетом зависимости от EXTENSIONS — **пока решается пересозданием серверов и импортом БД**
- Да, самое главное – наш **заказчик уже получает выгоду от импортозамещения!**

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

- Иметь возможность запросить план выполнения только что выполненного SQL-оператора, находясь внутри хранимой процедуры, для логирования и последующего анализа производительности
  ```

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.