PostgresPro

Postgres для фронтендеров

Иван Евгеньевич Панченко
Заместитель генерального директора
Postgres Professional

О докладчике

- Занимается прикладной разработкой с 1994 г
- Занимается Postgres с 1998 г
- Сооснователь компании Postgres Professional (2015 г)
- В компании отвечает за взаимодействие с заказчиками и формирование дорожной карты развития продукта

О чем этот доклад

PostgresPro

1) О Постгресе и СУБД вообще.
2) О JSON в постгресе

3) Как работать с Postgres, если Вы программируете на JavaScript.

- Серверный JS
- Клиентский JS

О чем будем говорить:

PostgresPro

1) Что такое СУБД? Реляционные и другие СУБД
2) Особенности Postgres как СУБД
3) Как установить Postgres и как работать с ним. Как залить данные ?
4) Как работать с Postgres на Javascript (и напишем простейшее приложение на Node.js)
5) Обсудим, как вообще работать с базой из веб-приложений
6) Посмотрим, как программировать на JS внутри Postgres. Т.е. Server-side.

Что такое БАЗА ДАННЫХ

1) Любые данные, хранящиеся в структурированной цифровой форме.
   Поэтому, файловая система или текстовый файл – тоже своеобразные БД

Что такое БАЗА ДАННЫХ

PostgresPro

1) Любые данные, хранящиеся в структурированной цифровой форме.
   Поэтому, файловая система или текстовый файл – тоже своеобразные БД

2) Данные, хранящиеся в структурированной форме, доступные через посредство специального софта (СУБД), который обеспечивает ряд типовых способов работы с данными.

Что такое БАЗА ДАННЫХ

PostgresPro

1) Любые данные, хранящиеся в структурированной цифровой форме.
   Поэтому, файловая система или текстовый файл – тоже своеобразные БД

2) Данные, хранящиеся в структурированной форме, доступные через посредство специального софта (СУБД), который обеспечивает ряд типовых способов работы с данными.

3) Что такое СУБД – см. п.2

Что такое реляционная база данных?

PostgresPro

1) В теории – это база данных, свойства которой описаны в статье Кодда (1970) https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

2) На практике - это база данных, состоящая (с точки зрения пользователя) из таблиц, в которых каждая строка представляет собой отдельную хранимую в базе запись; обеспечивающая при этом ACID* и выполняющая SQL-запросы. Язык SQL стандартизован с начала 90х (но стандарт очень изменился :)

*ACID придумал не Кодд, а Джим Грей: http://jimgray.azurewebsites.net/papers/thetransactionconcept.pdf

ACID (транзакционность)?

Транзакция – набор изменений, переводящих БД из одного согласованного состояния в другое.

Атомарность – Транзакция может совершиться либо полностью, либо никак.

Consistency (Целостность/согласованность) – Никто не увидит БД в несогласованном состоянии.

Изоляция – параллельно выполняющиеся транзакции не влияют друг на друга. (в реальности существуют Уровни Изоляции)

Durability (Устойчивость) – записанное в БД не пропадет.

Что такое SQL

«Язык структурированных запросов» . Декларативный язык.

DDL: (Data Definition Language)
CREATE TABLE, DROP TABLE и т.д.

DML: (Data Manipulation Language)
INSERT, UPDATE, DELETE

DQL: (Data Query Language)
SELECT

TCL: (Transaction Control Language)
BEGIN, COMMIT, ROLLBACK

DCL: (Data Control Language)
GRANT, REVOKE…

Чем хорош и плох SQL

Хорошо:

Декларативный язык.

Легко стандартизуется, слабо зависит от реализации.

Легко переносится между реализациями.

Декларативный язык.

Трудно подсказать тупой машине, как лучше выполнять конкретный запрос.

Сложно программировать алгоритмы => возникли процедурные языки на базе

Особенности Postgres (гуманитарная часть)

PostgresPro

- Нет единого центра владения
- Максимально свободная лицензия
- Рождён учеными

Краткая история Postgres

PostgresPro

1986-1995
Скрытый период

1996-1999
Время энтузиастов

2000-2009
Интернет-эпоха

2009-2015
Промышленный рост

2015 - . . .
Всеобщее принятие

Баланс двух стратегий разработки

PostgresPro

Developer-Driven

- Характерен для Open Source сообществ
- Игнорируется ряд потребностей крупных заказчиков
- Способствует инновациям

«Пользователь не знает, что ему нужно»

Customer-Driven

- Характерен для компаний
- Ориентирован на формально описанные потребности заказчиков
- Не способствует инновациям

*«Разработчик не понимает, что от него требуется, и занимается ерундой. Пусть лучше сделают, как в ****»*

В сообществе Postgres есть баланс между ними

Эволюция сообщества

Рост разработчиков следует за ростом потребителей

Меняется сообщество: основные действующие лица - компании

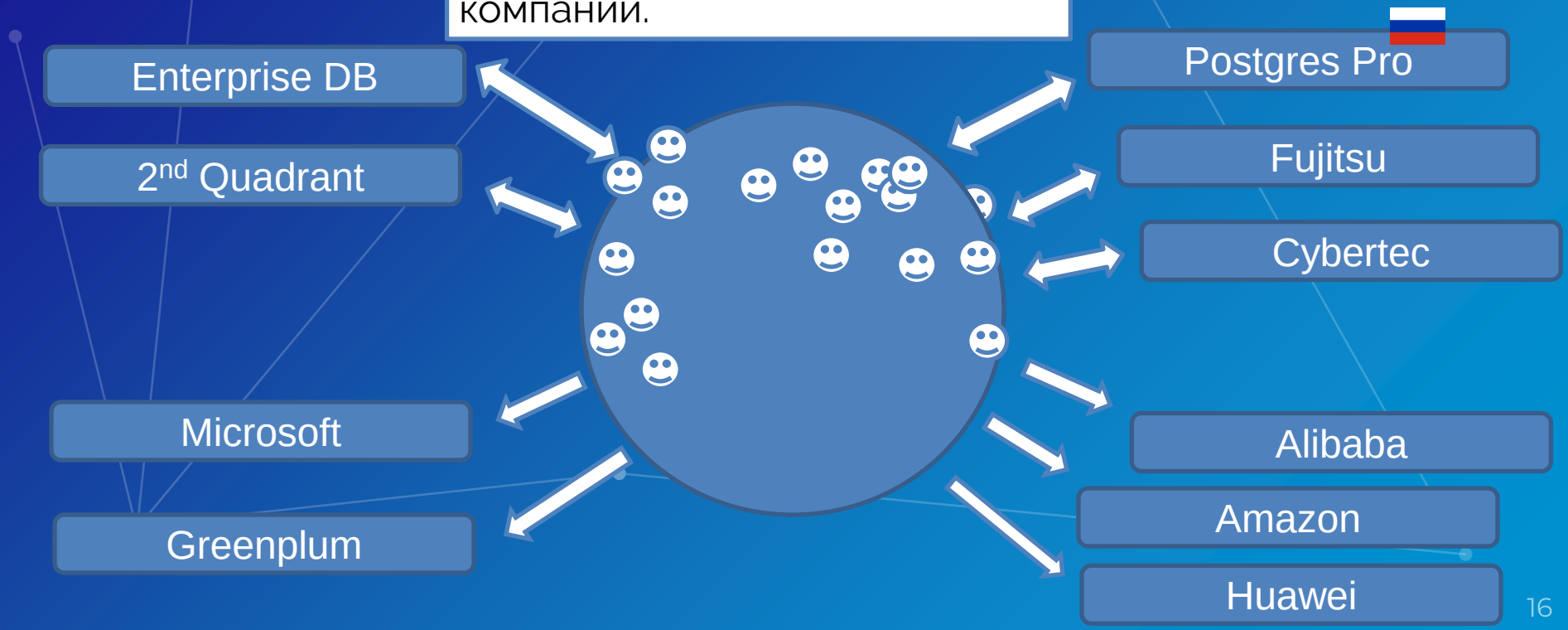
1й этап: Индивидуальные разработчики.
Делают, что хотят.

2й этап: Разработчики, нанятые компаниями.
Делают то, что нужно компаниям

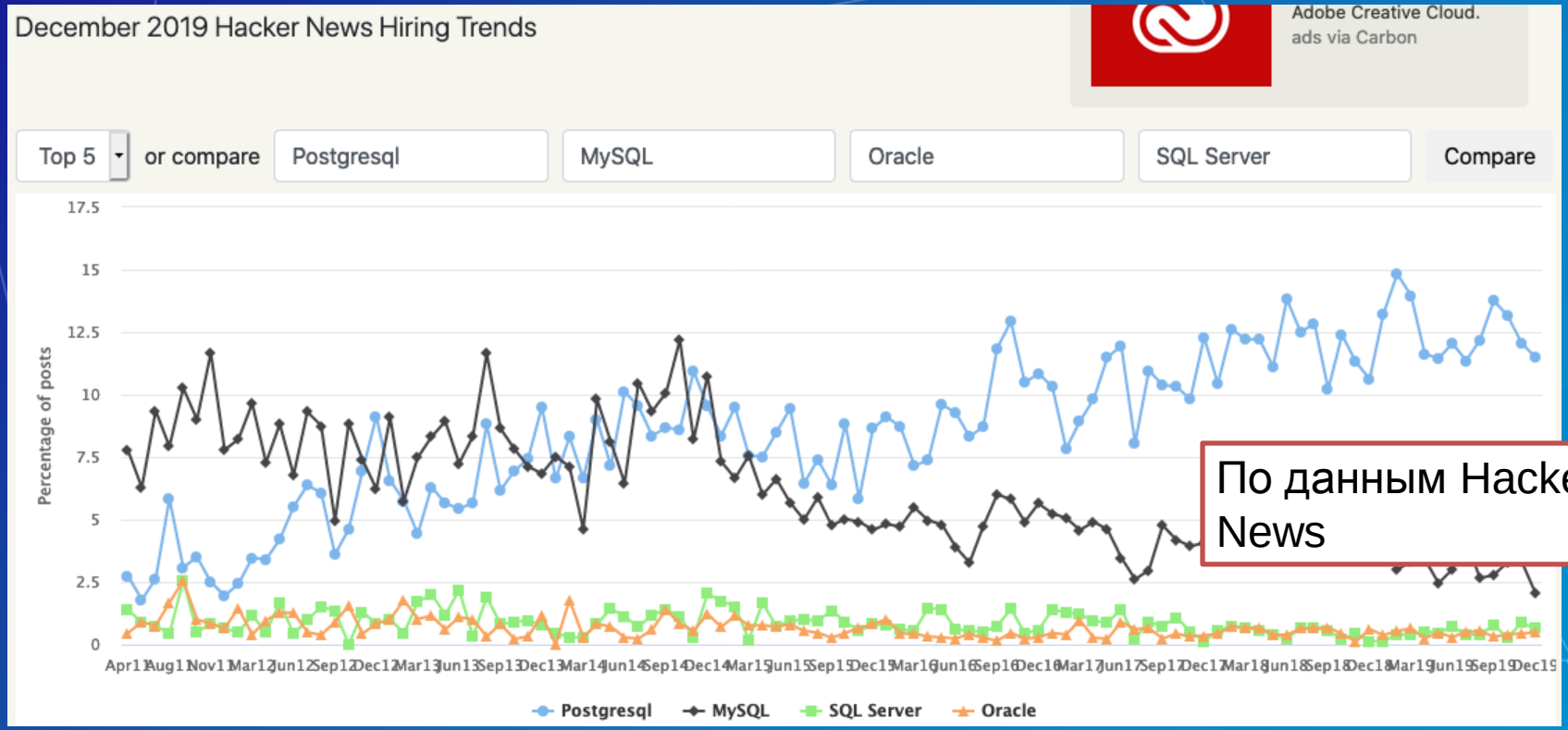
3й этап: Корпоративные разработчики.
Работают по своим планам развития для удовлетворения потребностей заказчиков.

Структура сообщества Postgres

Сейчас основной движущей силой сообщества являются компании.



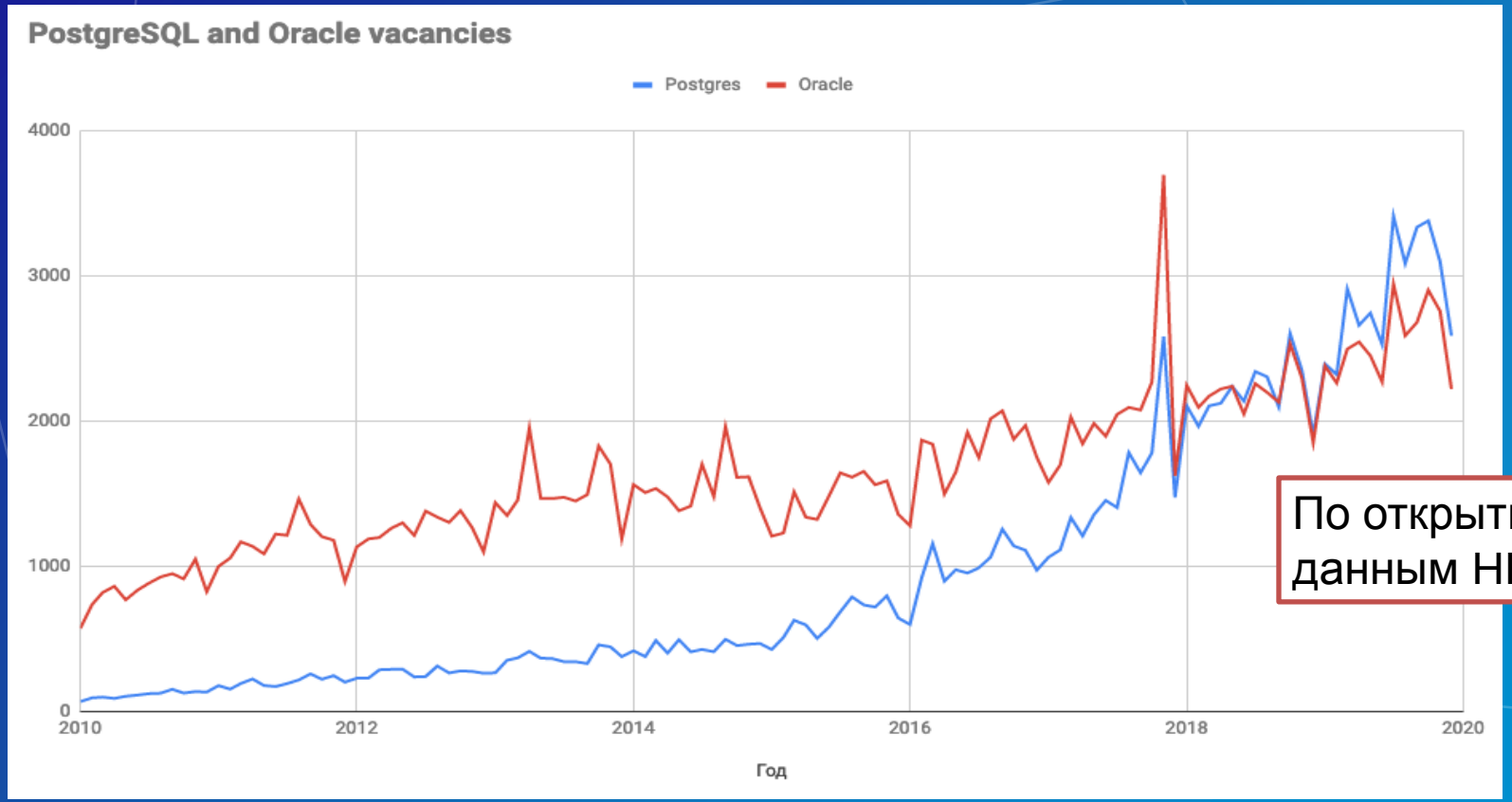
Популярность СУБД в мире



По данным Hacker News

Популярность СУБД в России

PostgresPro



По открытым данным HH.RU

Особенности Postgres (техническая часть). Не все!

Высокая степень соответствия стандартам SQL.
Высокая степень расширяемости. (и расширенности)
- типы данных; индексы; синтаксис языка; операторы; функции; (недавно) – стораджа.

Транзакционный DDL
Отличная работа со слабоструктурированными данными.
Временные таблицы – устроены так же, как обычные.
Процессная модель : одна сессия – один процесс (или больше).

https://www.sql-workbench.eu/dbms_comparison.html

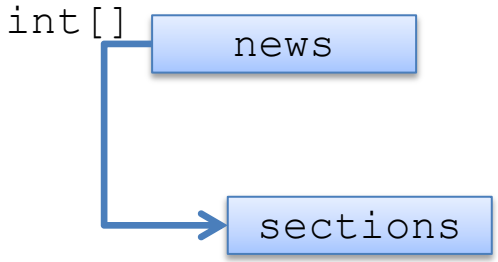
Слабоструктурированные данные в Postgres

PostgresPro

В классическом SQL нет массивов, и вообще композитных типов; тем более динамических объектов.
Неэлементарные атрибуты плохо лезут в классическую реляционную модель (на самом деле, нет).

В Postgres есть:
Массивы – давно
Hstore – с 2002 г (key1=>value1, key2=>value2 ….)
XML – давно (почти нет)
JSON – примерно с 2014 г

Началось с массивов




```
SELECT * FROM news
JOIN news_sections
  ON news.id = news_sections.news_id
WHERE news_sections.section_id = ?
```

```
SELECT * FROM news
WHERE section_ids && ARRAY[?]
```

Использование массивов оказалось быстрее чем классический JOIN благодаря созданию GiST-индексов.

Затем появился hstore…

EAV? Другие способы нагромождения таблиц?

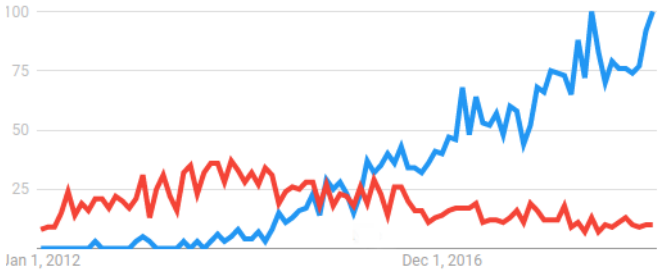
2003 (Pg 8.2) – расширение Hstore
– тип данных для набора пар "ключ-значение" (строка-строка)

```
'a=>1, b=>2'::hstore
```

Операторы:
- Извлечение: `hstore -> text`
- Есть ли ключ: `hstore ? text`
- Contains: `hstore @ hstore`

Индексная поддержка GiST, GIN (с 2006)

Google trends:
jsonb vs hstore



....но его вытеснил JSON(B)

PostgresPro

JSON:
- Стандартный
- Многоуровневый
- JSONB (быстрый)

HStore:
- Быстрый
- Многоуровневый Hstore (proposal)

- 2011: Joseph Adams предложил JSON-расширение для 9.1 (не вошло)
  https://postgrespro.ru/list/id/BANLkTik1k6bdafEzi7xvnB8vy+hcv4iD1Q@mail.gmail.com
- 2012: 9.2: тип данных json (хранение и верификация)
- 2012: Nested HStore proposal
- 2014: 9.4 JSONB (Бартунов, Коротков, Сигаев)
- 2016: Стандарт SQL:2016 поддержка JSON (Technical Report)
- 2019: 12 SQL/JSON (Бартунов, Коротков, Глухов)

https://commitfest.postgresql.org/17/1471/

JSON vs JSONB

PostgresPro

JSON:
- Текстовый
- Порядок ключей сохраняется
- Дублирующиеся ключи
- Пробелы
- Занимает меньше места (м.б. даже вдвое, для коротких полей. Для длинных разница → 0)
- Быстрее вставка
- Нет оператора @> (contains)

JSONB:
- Быстрый при выборке по ключу (в 1000 раз!)
- Ключи не дублируются и отсортированы по длине и ключу
- Поддерживается индексирование

Вообще-то, JSON не предназначался для хранения

Создание JSON

PostgresPro

1) Из текстового представления :

```
'{}'::jsonb
```

2) Функции-конструкторы:

```
jsonb_build_object(), array_to_json(), …
```

https://postgrespro.ru/docs/postgresql/12/functions-json

3) Из запросов БД

```
SELECT row_to_json(pg_class) from pg_class;
```

Извлечение данных из JSON

PostgresPro

1) Операторы `json->text, json->>text`:

```
WHERE js ->>'key' = 'value';
```

2) Операторы `json #> text[], json #>>text[]`

```
WHERE js #>>ARRAY['key'] = 'value';
```

3) Функции `json_each, json_each_text`

```
WHERE EXISTS (
  SELECT * FROM json_each_text(js)
     WHERE key='key' AND value='value'
);
```

Простой поиск по JSON(B)

PostgresPro

1) Операторы `json->text, json->>text:`

```
SELECT row_to_json(pg_class)->>'relname' from pg_class;
```

2) Операторы `json #> text[], json #>>text[]`

3) Функции `json_each, json_each_text, json_array_elements, json_array_elements_text`

```
SELECT * json_each_text('{"a":"foo","b":"bar"}')
          key | value
         -----+-------
          a   | foo
          b   | bar
```

Поиск с оператором @>

PostgresPro

```
{ "company_name": "Poupkine and Sons",
  "offices": [
     { "name": "Главный",
       "city": "Париж",
       "area": 1000
        },
     { "name": "Запасной",
       "city": "Москва",
       "area" : 200
        }
   ]
}

WHERE js @> '{"offices":[ {"city" : "Париж"} ]}'
```

Более сложный поиск

PostgresPro

У кого офис в Голицыно или Москве ?

```
WHERE
   js @> '{"offices":[ {"city" : "Голицыно"} ]}'
  OR
   js @> '{"offices":[ {"city" : "Москва"} ]}'
```

или

```
WHERE EXISTS (
    SELECT * FROM json_array_elements(js->'offices') t
      WHERE t->>'city' IN ('Голицыно', 'Москва')
);
```

Более сложный поиск

PostgresPro

У кого **главный** офис в Голицыно или Москве ?

```
WHERE (
   js @> '{"offices":[{ "city" : "Голицыно"} ]}'
  OR
   js @> '{"offices":[{ "city" : "Москва"} ]}'
  ) AND
   js @> '{"offices":[{ "name" : "Главный"} ]}';

WHERE (
 js @> '{"offices":[{
           "city": "Голицыно","name":"Главный"
       }]}'
 OR
 js @> '{"offices":[{
           "city": "Москва",   "name":"Главный”
       }]}';
```

Ещё более сложный поиск

У кого **главный** офис в Голицыно или Москве ?

```
WHERE (
   js @> '{"offices":[{ "city" : "Голицыно"} ]}'
  OR
   js @> '{"offices":[{ "city" : "Москва"} ]}'
  ) AND
   js @> '{"offices":[{ "name" : "Главный"} ]}';

WHERE (
 js @> '{"offices":[{
            "city": "Голицыно","name":"Главный"
       }]}'
 OR
 js @> '{"offices":[{
            "city": "Москва",   "name":"Главный”
       }]}';
```

Ещё более сложный поиск

PostgresPro

У кого **главный** офис в Голицыно или Москве ?

```
WHERE (
   js @> '{"offices":[{ "city" : "Голицыно"} ]}'
  OR
   js @> '{"offices":[{ "city" : "Москва"} ]}'
  ) AND
   js @> '{"offices":[{ "name" : "Главный"} ]}';

WHERE (
 js @> '{"offices":[{
            "city": "Голицыно","name":"Главный"
       }]}'
 OR
 js @> '{"offices":[{
            "city": "Москва",   "name":"Главный”
       }]}';
```

То же на чистом SQL

PostgresPro

У кого **главный** офис в Голицыно или Москве ?

```
WHERE EXISTS(
  SELECT * FROM json_array_elements(js->'offices') t
      WHERE t->>'city' IN ('Голицыно', 'Москва')
        AND t->>'name' = 'Главный'
);
```

На чистом SQL можно больше

PostgresPro

У кого **главный** офис в Голицыно или Москве
с площадью от 500 кв.м?

```
WHERE EXISTS(
  SELECT * FROM json_array_elements(js->'offices') t
      WHERE t->>'city' IN ('Голицыно', 'Москва')
        AND t->>'name' = 'Главный'
        AND (t->>'area')::float > 500
                        -- для SQL не проблема!
);
```

Немного выводов

- Поиск с –> хорош, ему помогают B-Tree функциональные индексы, но он может немного.
- Поиск с `@>` может больше, ему помогает GIN-индекс, но он недостаточно эффективен при сложных запросах.
- Вариант с разворачиванием в SQL наиболее универсален, но индексы не помогают совсем.

Нужно что-то делать.

Что было сделано

JSQuery – расширение (Александр Коротков)
JSONPATH – новые возможности с Pg12 (О.Бартунов, Н. Глухов, А.Коротков, Л.Мантрова, Ф.Сигаев) – в соответствии со стандартом SQL:2016
JSONTABLE – продолжение (в Pg13..14)

Извлечение с JSONPATH

```
SELECT jsonb_path_query_first(
    js, '$.offices[0].city'
)

SELECT jsonb_path_query_first(
    js, '$.offices[*] ? (@.area > 500 && @.name ==
    "Главный").city'
)
```

-- Город, в которым есть главный офис > 500 кв.м

Поиск с JSONPATH

```
SELECT * FROM t WHERE
 js @?
  '$.offices[*] ? (@.area > 500 && @.name == "Главный").city'
```

Подобнее о JSONPATH:
https://github.com/obartunov/sqljsondoc/blob/master/jsonpath.md

Таблица JSON-энтузиаста

```
CREATE TABLE data (
        fields jsonb
);
```

Таблица JSON-энтузиаста на следующий день

PostgresPro

```
CREATE TABLE data (
        fields jsonb
);

CREATE INDEX data_pk ON data (fields->>'id');
```

2 дня спустя...

```
CREATE TABLE foo(
       fields jsonb
);
CREATE UNIQUE INDEX data_pk  ON foo((fields->>'id'));
CREATE UNIQUE INDEX data_pki ON foo (((fields->>'id'):int));
INSERT INTO foo
   SELECT jsonb_build_object('q', random(), 'id', x)
   FROM generate_series(10000000,20000000);
```

Какой индекс лучше?
Каковы издержки каждого подхода?

Издержки хранения всех полей в JSON ч.1

PostgresPro

Ч 1. Издержки хранения первичного ключа в JSON

```
CREATE TABLE data_smth (
        data_id int4 REFERENCES …… ???? WTF
);
```

Не забыть:

```
  CHECK (fields->>'id' IS NOT NULL);
```

Это вообще не первичный ключ. Например, IDENTITY при логической репликации…

Издержки хранения всех полей в JSON ч.2

PostgresPro

Ч 2. Издержки хранения данных в JSON

- Больше места (храним ключи + оверхед JSONB) не всегда!!
- Извлекается дольше (особенно при TOAST)
- index scan ~ такой же, Seq scan 2-3 раза медленнее, даже без TOAST
- Но: Нет статистики. Поэтому планировщик будет ошибаться.
- Нет всего разнообразия типов. Даже дат.
- Меньше читаемость (иногда)

Хорошая статья Dan Robinson.

Как экономить место с JSON ?

PostgresPro

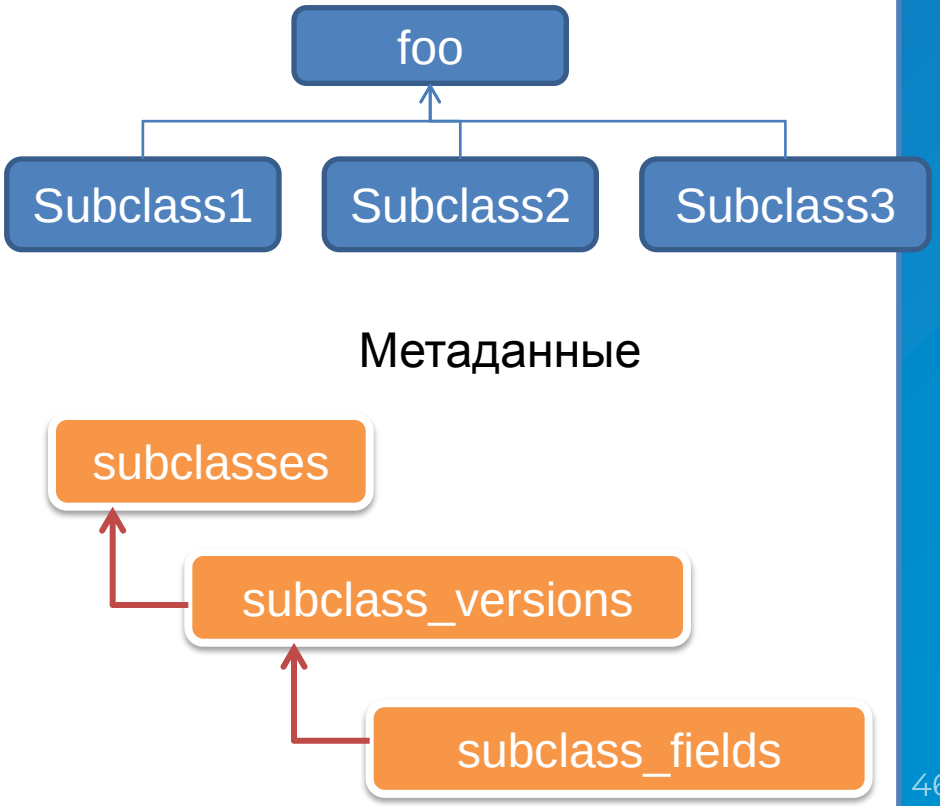
- Объединить несколько записей в одну.
- Каких?
  - Близких по смыслу, по времени, по пространственному расположению
- Например: часть временного ряда

Какие поля хранить в JSON ?

- Реально полиморфные
  - Но описание структуры надо всё же где-то хранить
  - И чем-то гарантировать отсутствие бардака (чем ближе к данным, тем лучше)
- Имеющие внутреннюю структуру
  - Особенно, если собираетесь использовать JSQuery или JSON Path
- Если надо хранить историю, несмотря на изменения в структуре

Пример версионно-полиморфной структуры

```
CREATE TABLE foo (
   id int primary key,
   subclass … ,
   schema_version …
/* SUPERCLASS_FIELDS */
   subclass_fields   jsonb
);
CREATE TRIGGER …
```



Пример поля, имеющего внутреннюю структуру

- Image

```
{ file_name: '...',
  file_path: '...',
  width:
  height:
}
```

(но для Image структура одинаковая, можно использовать композитный тип)

- Plan

(развесистое дерево, получаемое из EXPLAIN (FORMAT JSON)

Отступление

Как получить план в JSON и поместить его в таблицу

PostgresPro

- К сожалению, нельзя сделать SELECT FROM (EXPLAIN …)
- Или EXPLAIN INTO ….
- Нужно написать функцию, например такую:

```
CREATE FUNCTION explain (query text) RETURNS jsonb
   LANGUAGE plperl TRANSFORM FOR TYPE jsonb  AS $$
   my ($sql) = @_;
   my $res = spi_exec_query(
        "EXPLAIN(FORMAT JSON) $sql", 1);
   return $res->{rows}[0]->{"QUERY PLAN"};
$$;
```

JSON-агрегаты

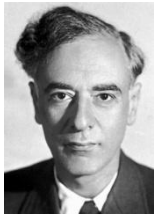
- Задача:
  Получить список книг с авторами, по порядку

Теория поля.// Ландау Л.Д., Лифшиц Е.М.
Язык программирования С.// Керниган Б.В., Ритчи Д.М.

Что хотим увидеть

PostgresPro

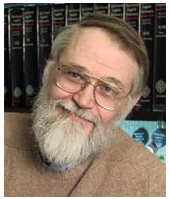
Теория поля.



Ландау Л.Д.,

Лифшиц Е.М.

Язык программирования C.



Керниган Б.В.,

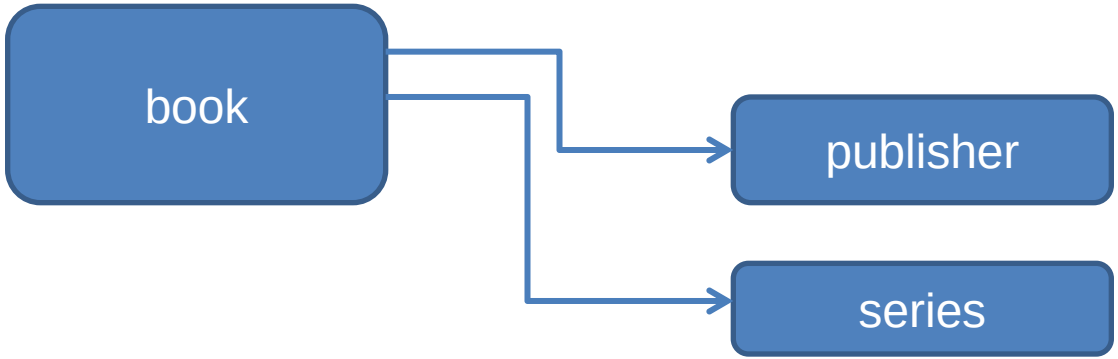
Ритчи Д.М.

Решение

```
SELECT book.*,
  ( SELECT json_agg(row_to_json(x))
     FROM (
            SELECT person.*
            FROM person
            JOIN authorship ON person.id = person_id
            WHERE book_id = book.id
            ORDER BY pos
      ) x
   ) authors
FROM book;
```

Ещё JSON-агрегаты

- Извлечение атрибутов по ссылкам из внешних таблиц




```
SELECT book.*,
    row_to_json(series)    AS $series,
    row_to_json(publisher) AS $publisher
FROM book
LEFT JOIN publisher ON published.id = book.publisher
LEFT JOIN series    ON series.id    = book.series
```

Задача посложнее: достать книгу с рубрикацией

PostgresPro

```
CREATE TABLE sections (     /* Рубрикатор */
    id      int PRIMARY KEY,
    parent int REFERENCES sections(id),
    title   text
);
CREATE TABLE book (    /* Книги */
    …
    sections[] int
    …
);
Хотим: { sections: [    /* Пути от корня до всех рубрик книги */
           [ {id:..., title:...}, {id:....} … ] -- 1я рубрика
           [ {id:...                            -- 2
             ...
         ]
}
```

Тестовые данные о книгах

```
INSERT INTO section VALUES
   (1, NULL, 'СУБД'),
       (2, 1, 'PostgreSQL'),
   (3, NULL, 'ОС'),
       (4, 3, 'Solaris');
INSERT INTO book VALUES (
    'Postgres for Solaris OS',
    ARRAY[2,4]
   );
```

Шаг 1: путь до узла в дереве

```
WITH RECURSIVE path AS (
   SELECT s.id, s.title, s.parent , 0 AS level
      FROM section s WHERE s.id = 4
    UNION ALL
    SELECT s.id, s.title, s.parent, path.level+1 AS level
      FROM section s JOIN path ON s.id = path.parent
)
SELECT id,title FROM path ORDER BY level DESC;
```

```
 id |  title
----+---------
  3 | ОС
  4 | Solaris
(2 rows)
```

Шаг 2: агрегируем путь

```
SELECT json_agg(row_to_json(path)) AS path
    FROM (
        WITH RECURSIVE path AS (
           SELECT s.id, s.title, s.parent , 0 AS level
           FROM section s WHERE s.id = 4
             UNION ALL
           SELECT s.id, s.title, s.parent, path.level+1 AS level
           FROM section s JOIN path ON s.id = path.parent
        )
      SELECT id,title FROM path ORDER BY level DESC
    ) path;
```

```
                                path
-----------------------------------------------------------------
 [{"id":3,"title":"OC"}, {"id":4,"title":"Solaris"}]
(1 row)
```

Шаг 3: получаем несколько путей сразу

PostgresPro

```
SELECT * FROM (
        SELECT unnest(ARRAY[2,4]) node_id
) nodes
JOIN LATERAL (
    SELECT json_agg(row_to_json(path)) AS path
        FROM (
          WITH RECURSIVE path AS (
             SELECT s.id, s.title, s.parent ,0 AS level
                FROM section s WHERE s.id = node_id
             UNION ALL
             SELECT s.id, s.title, s.parent, path.level+1 AS level
                FROM section s JOIN path ON s.id = path.parent
          )
           SELECT id,title FROM path ORDER BY level DESC
    ) path
) AS path ON true;
```

```
 node_id |                                  path
---------+-------------------------------------------------------------------
       2 | [{"id":1,"title":"СУБД"}, {"id":2,"title":"PostgreSQL"}]
       4 | [{"id":3,"title":"ОС"}, {"id":4,"title":"Solaris"}]
(2 rows)
```

Шаг 4: Подводим итог

```
SELECT title, (
   SELECT json_agg(path.path)
   FROM (
      SELECT unnest(book.sections) node_id
   ) nodes
   JOIN LATERAL (
      SELECT json_agg(row_to_json(path)) path
         FROM (
            WITH RECURSIVE path AS (
               SELECT s.id, s.title, s.parent , 0 AS level
                  FROM section s WHERE s.id = node_id
               UNION ALL
               SELECT s.id, s.title, s.parent, path.level+1 AS level
                  FROM section s JOIN path ON s.id = path.parent
            )
            SELECT id,title FROM path ORDER BY level DESC
         ) path
   ) path  ON true
) paths FROM book;
```

Вот и результат:

```
 -[ RECORD 1 ]-----------------------------------------------
title | Postgres for Solaris OS
paths |
   [
    [{"id":1,"title":"СУБД"}, {"id":3,"title":"PostgreSQL"}],
    [{"id":2,"title":"ОС"},   {"id":4,"title":"Solaris"}]
   ]
```

Disclaimer:

- Помните, что чем сложнее запрос, тем хуже оптимизатору

Доставая лишние данные, вы увеличиваете углеродный футпринт

Другая задача: 2 в одном

Задача: выполнить несколько запросов и вернуть результат вместе.

```
SELECT
   ( SELECT json_agg(row_to_json(book)) books
     FROM book )::jsonb ||
   ( SELECT json_agg(row_to_json(section)) sections
     FROM section)::jsonb;
```

Или

```
SELECT json_build_object(
   'books',
   (SELECT json_agg(row_to_json(book)) books FROM book ),
   'sections',
   (SELECT json_agg(row_to_json(section)) sections FROM
   section));
```

Задача 3. Собираем дерево

Задача: Собрать записи узлов дерева в многоуровневое дерево JSON.

```
[{  id: 1, title: 'СУБД', nodes: [
        { id: 2, title: 'Реляционные СУБД', nodes: [
                    ……
                                                  и т.д.
```

- Хочется RCTE, но непонятно, как его привинтить:
  - CTE умножает количество строк, а нам надо углублять дерево
- Эврика! Нужен агрегат. Он уменьшает количество строк!

Высаживаем дерево

План решения:

1) Готовим дерево обычным CTE
2) Агрегируем специальным хитрым агрегатом, который ставит каждый узел на свое место в дереве

Шаг 1. Вычислим путь для каждого узла

```
WITH RECURSIVE
   positioned_section AS (
     SELECT *,
       ROW_NUMBER() OVER (PARTITION BY parent ORDER BY id) -1 AS pos
     FROM section
   ),
   t AS (
     SELECT id,title,
       ARRAY['nodes', pos::text] AS path
       FROM positioned_section WHERE parent IS NULL
     UNION ALL
     SELECT s.id, s.title,
       t.path || ARRAY['nodes',pos::text] AS path
       FROM positioned_section s JOIN t ON s.parent = t.id
   )
 SELECT row_to_json(t)::jsonb node FROM t ORDER BY path;
```

Шаг 1: Результат

```
 node
------------------------------------------------------------
 {"id": 1, "path": ["nodes", "0"], "title": "СУБД"}
 {"id": 3, "path": ["nodes", "0", "nodes", "0"], "title":
   "PostgreSQL"}
 {"id": 2, "path": ["nodes", "1"], "title": "ОС"}
 {"id": 4, "path": ["nodes", "1", "nodes", "0"], "title":
   "Solaris"}
(4 rows)
```

Шаг 2. Агрегат, ставящий узлы на места

```
CREATE OR REPLACE FUNCTION tree_agg_f (state jsonb, item jsonb)
   RETURNS jsonb LANGUAGE sql IMMUTABLE STRICT AS $$
     SELECT jsonb_set(
        state,
        (SELECT array_agg(x)  /*
           FROM jsonb_array_elements_text(item->'path') x
        ),
        jsonb_build_object(
           'id', item->'id',
           'title', item->'title',
           'nodes','[]'::jsonb
        ),
        true);
$$ ;

        CREATE AGGREGATE tree_agg (item jsonb) (
          SFUNC=tree_agg_f, STYPE=jsonb, INITCOND='{"nodes":[]}');
```

Шаг 3. Объединяем шаги 1+2

```
SELECT jsonb_pretty(tree_agg(x.node)) FROM (
  WITH RECURSIVE
  positioned_section AS (
    SELECT *,
      ROW_NUMBER() OVER (PARTITION BY parent ORDER BY id) -1 AS pos
    FROM section
  ),
  t AS (
    SELECT id,title,
      ARRAY['nodes', pos::text] AS path
      FROM positioned_section WHERE parent IS NULL
    UNION ALL
    SELECT s.id, s.title,
      t.path || ARRAY['nodes',pos::text] AS path
      FROM positioned_section s JOIN t ON s.parent = t.id
  )
  SELECT row_to_json(t)::jsonb node FROM t ORDER BY path) x;
```

Ещё один пример - гистограмма

- Одним агрегатом получить целую гистограмму
- Возьмем упрощенную модель. Найдем частоты появления различных случайных чисел от 0 до 10
- Данные для примера:

```
SELECT( random()*20)::int
   FROM generate_series(1,20);
```

Создадим агрегат

```
CREATE OR REPLACE FUNCTION freq_agg_f (state jsonb, item
   int) RETURNS jsonb LANGUAGE sql IMMUTABLE STRICT AS $$
      SELECT jsonb_set(
       state,
       ARRAY[item]::text[],
       to_jsonb(COALESCE((state->>(item::text))::int, 0) +1),
       true
      );
$$;
CREATE AGGREGATE freq_agg (int) (
   SFUNC=freq_agg_f, STYPE=jsonb, INITCOND='{}');
```

Результат

```
SELECT freq_agg (  (random()*20)::int ) FROM
   generate_series(1,20);
```

```
 freq_agg
----------------------------------------------------
 {"2": 1, "3": 2, "5": 1, "6": 2, "10": 2, "11": 2, "13": 1,
   "15": 3, "16": 3, "17": 1, "18": 2}
(1 row)
```

Светлое будущее JSON

PostgresPro

- Единый тип данных для JSON
- Оптимизация хранения (спец-TOAST)
- Lazy transform
- Более быстрый поиск внутри

PL/v8

PostgresPro

- Postgres обладает необычайной расширяемостью.
- И, в том числе, позволяет создавать отдельные процедурные языки как расширения.
- Несколько языков:
- PL/PgSQL – «основной SQL-образный язык»
- PL/Perl, PL/Python, PL/TCL – языки-расширения из «коробки»
- PL/v8, PL/Lua, PL/Java, PL/Go – сторонние расширения

- PL/v8 – хорошее, но проблемное расширение

Установить PL/v8

```
sudo apt -y install postgresql-plv8-10
```

- https://github.com/AoAnima/PLV8-Postgresql11-Ubuntu-18

Или собирать самим: https://plv8.github.io/#building

Затем

```
CREATE EXTENSION plv8;
```

Самостоятельная сборка PL/v8

Осторожно, Трафик!!! И другие неприятности.

```
yum -y install postgresql12-devel make git
wget https://github.com/plv8/plv8/archive/v2.3.9.tar.gz
tar xzf v2.3.9.tar.gz
cd plv8-2.3.9/
make PG_CONFIG=/usr/pgsql-12/bin/pg_config static
make PG_CONFIG=/usr/pgsql-12/bin/pg_config install
```

PL/v8 – «доверенный язык»

PostgresPro

`TRUSTED LANGUAGE`

Это значит, что процедуры на нем может создавать доверено непривилегированным пользователям БД.

Но это оттого, что этому языку не разрешается ходить на диск, в сеть и т.д.

Как следствие: Весь код должен лежать в БД, подгружать библиотеки откуда-то не получится.

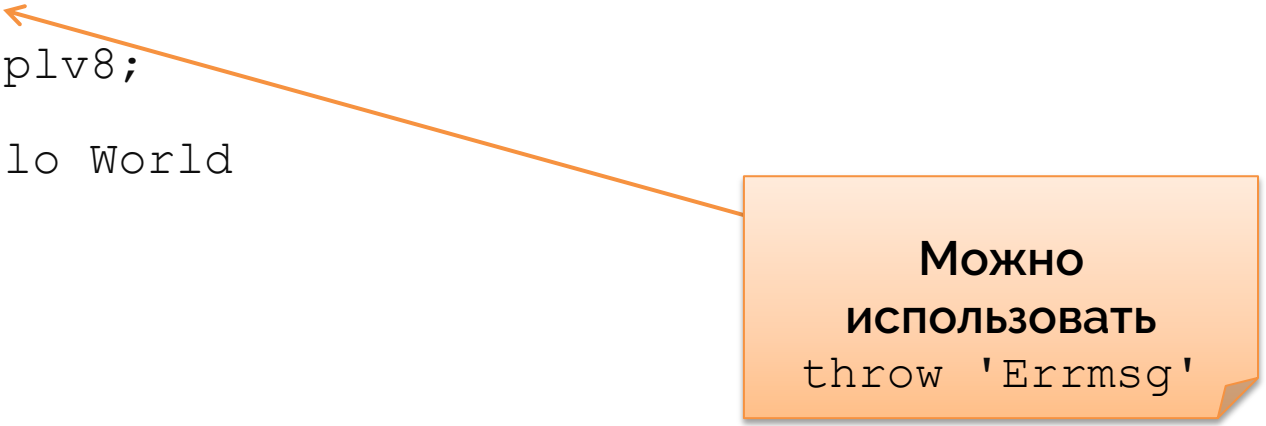
PL/v8 – другие особенности

- Автоматический маппинг JSON (JSONB)
- Возможность определять window functions
- Возможность делать подтранзакции
- Упрощенный вызов других функций PL/v8
- plv8.execition_timeout=300 (по умолчанию не включено при сборке)
- Инициализация:
- plv8.start_proc=my_start_func //(имя PLv8-функции)
- Подробнее см https://rymc.io/2016/03/22/a-deep-dive-into-plv8/

PL/v8 – Hello world

```
DO $$
   plv8.elog(NOTICE, 'Hello World');
$$ LANGUAGE plv8;

NOTICE:  Hello World
```



PL/v8 – работа с базой

```
DO $$ plv8.elog(NOTICE,
    JSON.stringify(
       plv8.execute('select 57 as x')

    ));
    $$ LANGUAGE plv8 ;

NOTICE:  [{"x":57}]
```

PL/v8 – экранирование

PostgresPro

```
plv8.quote_literal()

plv8.quote_nullable()

plv8.quote_ident()
```

```
DO $$
plv8.elog(NOTICE,
 plv8.quote_nullable(
    "Macy's"));
$$ LANGUAGE plv8 ;
NOTICE:  'Macy''s'
```

PL/v8 – запросы с параметром

```
DO LANGUAGE plv8 $$
var h= plv8.prepare('SELECT * FROM pg_class WHERE
       relname ~ $1', ['text'] );

plv8.elog(NOTICE, h.execute (['pg']));
h.free();
$$;
```

PL/v8 – производительность

```
DO $$ DECLARE a int; i int;
    BEGIN FOR i IN 0..999 LOOP
       SELECT count(*) INTO a FROM pg_class; END LOOP;
    END;
    $$ LANGUAGE plpgsql;
```

40ms

```
DO $$ for(var i=0;i<1000;i++)
    plv8.execute('SELECT count(*) FROM pg_class');
    $$ LANGUAGE plv8 ;
```

100ms

PL/v8 – производительность

```
DO $$
    var h=plv8.prepare(
        'SELECT count(*) FROM pg_class'
    );
    for(var i=0;i<1000;i++) h.execute();

$$  LANGUAGE plv8 ;
```

55ms

PL/v8 – производительность вычислений

```
DO $$
    DECLARE i int; a bigint;
    BEGIN a=0;
     FOR i IN 0..1000000 LOOP
       a=a+i*i::bigint;
     END LOOP;
    END;
    $$ LANGUAGE plpgsql;
```

280ms

```
DO $$
    var a=0;
    for(var i=0;i<=1000000;i++) a+=i*i;

$$ language plv8;
```

4ms !!

PL/v8 – обратная сторона производительности вычислений

```
DO $$
    var a=0;
    for(var i=0;i<=1000000;i++) a+=i*i;
    plv8.elog(NOTICE, a);

$$ language plv8;
    333333833333127550
```

- В Javascript целое представляется в виде float.
- Поэтому в предыдущих примерах результат получается быстро, но не точно:

333333833333127550 вместо 333333833333500000

$$\sum = n*(n+1)*(2n+1)/6$$

PL/v8 – память

```
CREATE OR REPLACE FUNCTION qqq() RETURNS int LANGUAGE
    plv8 AS $$

return plv8.execute(
    'select count(*) from pg_class'

)[0].count;

$$;
```

Хорошо (не течёт)

PL/v8 – память !!!

```
CREATE OR REPLACE FUNCTION qqq() RETURNS int LANGUAGE
    plv8 AS $$
    var h = plv8.prepare(
        'select count(*) from pg_class');

    return h.execute()[0].count;

$$;
```

Плохо (течёт)

PL/v8 – память !!!

```
CREATE OR REPLACE FUNCTION qqq() RETURNS int LANGUAGE
    plv8 AS $$
    var h = plv8.prepare(
        'select count(*) from pg_class');
    var r = h.execute()[0].count;
    h.free();
    return r;
    $$;
```

Снова хорошо (не течёт)

PL/v8 – параметры

PostgresPro

В каком виде параметры из SQL попадают в PL/v8?

```
CREATE OR REPLACE FUNCTION jdump(a int, b bytea, c int[], d
    jsonb ) RETURNS void LANGUAGE plv8 AS

        $$ plv8.elog(WARNING,a,b,c,d) $$;

SELECT jdump(1,'abcd', ARRAY[1,2,3],'{"a":2,"b":3}');

(1
    18,52,86 // массив байтов
    1,2,3    // массив чисел
    [object Object] // JSON !! ]
```

Проблемы: даты вне диапазона; бесконечность

PL/v8 – узнать версию

```
DO LANGUAGE plv8 $$
    plv8.elog(NOTICE,plv8.version);
    $$;
```

Быстрый доступ к функциям

Возвращает функцию по ее имени (полиморфизм – ошибка)

```
plv8.find_function(name);
```

```
DO LANGUAGE plv8 $$
    plv8.find_function('jdump')(1, 'abc');
    $$;
```

Инициализация PL/v8

Стартовая функция определяется параметром конфигурации start_proc

```
CREATE OR REPLACE FUNCTION my_init() RETURNS void
    LANGUAGE plv8 AS $$
    this.xxx = function() { return 57; }; this.qqq =
    157;  $$;

SET plv8.start_proc = 'my_init';

DO LANGUAGE plv8 $$
    plv8.elog(NOTICE, qqq, xxx(3) );
    $$;
```

Работа с курсором в PL/v8

```
var h = plv.prepare('SELECT …');
var cursor = h.cursor();
var row;
while(row = cursor.fetch()) {
…
}
cursor.close();
h.free();
```

Процедуры и функции в PL/v8

PostgresPro

- Функция – всегда внутри одной транзакции
- Процедура – может содержать много транзакций
- В Pg с 11-й версии появились процедуры
- Но в PL/v8 процедур пока нет

Подтранзакции в PL/v8

```
try {
    plv8.subtransaction(function() {
        plv8.execute('UPDATE…');
        plv8.execute('UPDATE…');
    });

}

catch(e) {
    …

}
```

Агрегат на PL/v8

Агрегирует сразу сумму и список значений

```
CREATE EXTENSION plv8;
CREATE FUNCTION v8_agg_f (state jsonb, item int) RETURNS
    jsonb AS $$
    state.list.push(item);
    state.sum += item;
    return state;
$$
LANGUAGE plv8 IMMUTABLE STRICT;
CREATE AGGREGATE v8_agg (int) (SFUNC=v8_agg_f,
    STYPE=jsonb,
    INITCOND='{"list":[], "sum": 0}');
```

Тот же агрегат на SQL

PostgresPro

```
CREATE FUNCTION sql_agg_f (state jsonb, item int) RETURNS
   jsonb LANGUAGE sql IMMUTABLE STRICT AS $$
   SELECT jsonb_set(
        jsonb_set(state, ARRAY['list', '2000000000'],
                  to_jsonb(item), true ),
        ARRAY['sum'],
        to_jsonb( (state->>'sum')::int + item ),
        true
   );
$$;

CREATE AGGREGATE sql_agg (int)( SFUNC=sql_agg_f,
   STYPE=jsonb, INITCOND='{"list":[], "sum": 0}');
```

Выводы: когда и для чего юзать PL/v8

- Работа со сложными структурами данных и алгоритмами
- Формирование динамических SQL (ORM, отчеты)
- Готовые библиотеки
- Прежде чем писать на C, попробуй на PL/v8*

Переходим на сторону клиента

(для кого-то это сервер – но для постгреса – клиент).

```
npm install pg
```

Первый запрос на node

```
const { Client } = require('pg')
const client = new Client({
    user: 'postgres',    database: 'test',
    port: 5435,          host: '/var/run/postgresql'
  })
client.connect();
client.query(
'SELECT $1::text as message, (SELECT count(*) FROM
   pg_tables) AS n', ['Hello world!'],
 function(err, res)  {
   console.log(err,res)
   client.end()
})
```

Пишем сервер

```
const { Pool, Client } = require('pg')
const pool = new Pool({max: 40, /* параметры соединения */})
var http = require('http');
const header = {'Content-Type' : 'application/json' }
http.createServer(function(req, response) {
 pool.query('/* некий запрос */', [/* параметры */],
  function(err, res)  {
     if(err) {
        response.writeHead(500, header);
        response.end(JSON.stringify({error: err}));
     } else {
        response.writeHead(200, header);
        response.end(JSON.stringify(res.rows[0]));
     }
  }
 );
}).listen(8080);
```

ORM, специфических для Pg, нет

Есть ORM для nodejs:

- `Sequelize`
- `TypeORM`

Но: ни одна из них нормально не поддерживает особенности Postgres
Например – работу с JSON.

Литература

- JSON, JSONB, JSQuery (PgConf.Russia 2017)
- PostgreSQL: Серверное программирование на «человеческом» языке
- JSONB в примерах (PgConf.EU 2019)
- JSONPATH manual
- PL/v8
- NodeJS и Postgres

Спасибо за внимание

Postgres Professional

Москва, ул. Дмитрия Ульянова 7А

+7 (495) 150-06-91

info@postgrespro.ru

www.postgrespro.ru