PostgresPro

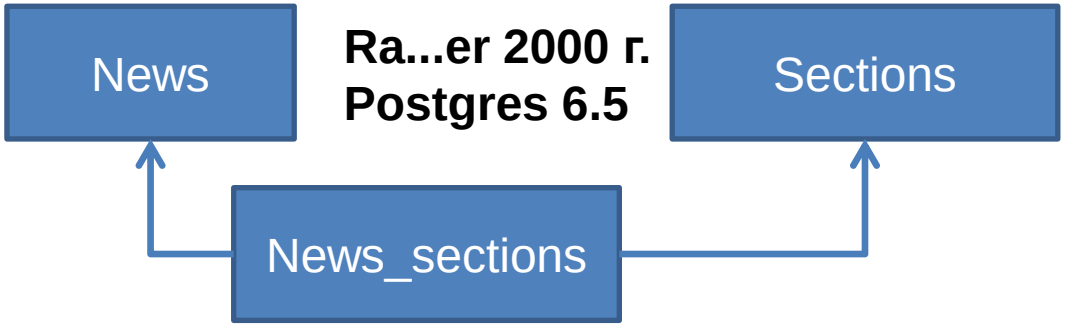
Интересные случаи использования JSON

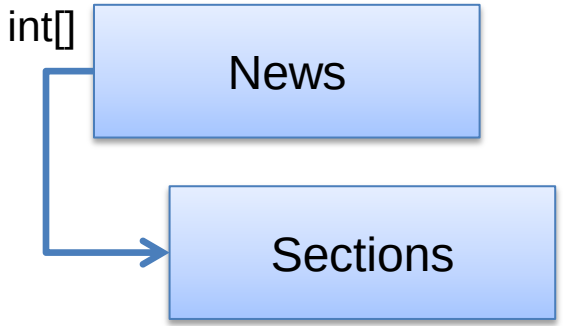
Иван Панченко

Postgres Professional

PROFESSIONAL Postgres

Откуда бысть пошла…






```
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.

Хранение полиморфных данных…

EAV? Другие способы нагромождения таблиц?

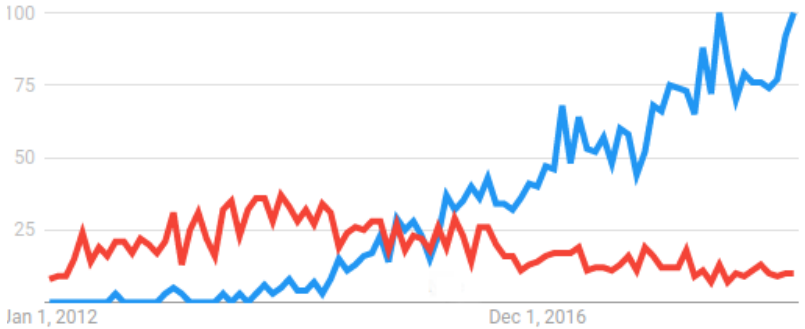
2003 (Pg 8.2) – расширение Hstore
– тип данных для набора пар "ключ-значение" (строка-строка)

```
'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



PROFESSIONAL Postgres

JSON(B) спешит на помощь

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/

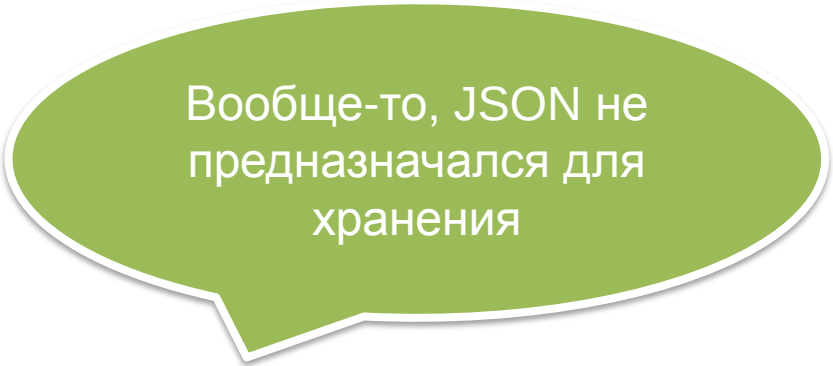
JSON или JSONB ?

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



Первый «интересный» случай

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

Первый «интересный» случай

```
CREATE TABLE data (
      fields jsonb
);

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

Первый «интересный» случай

```
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 этого подхода?

Издержки хранения всех полей в JSON

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. 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 ?

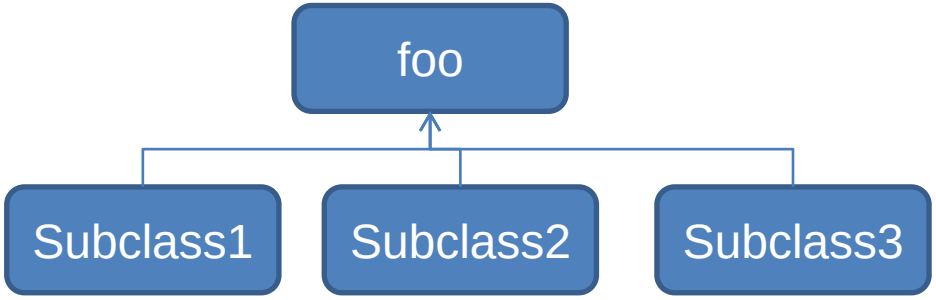
- Объединить несколько записей в одну.
- Каких?
  - Близких по смыслу, по времени, по пространственному расположению
- Например: часть временного ряда

Какие поля хранить в JSON ?

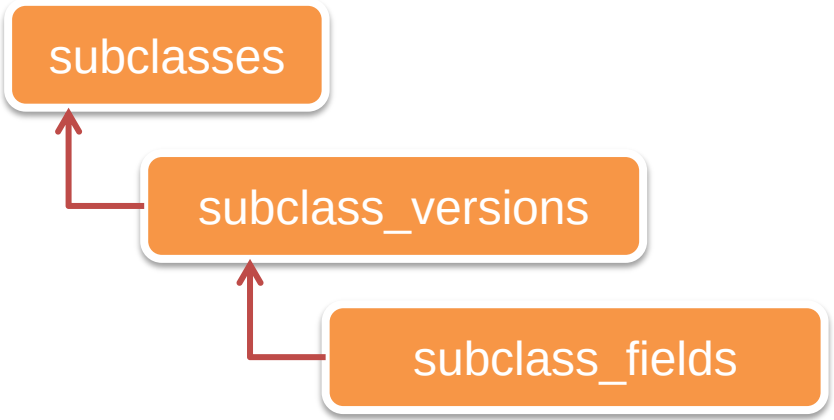
- Реально полиморфные
  - Но описание структуры надо всё же где-то хранить
  - И чем-то гарантировать отсутствие бардака (чем ближе к данным, тем лучше)
- Имеющие внутреннюю структуру
  - Особенно, если собираетесь использовать JSQuery или JSON Path
- Если надо хранить историю, несмотря на изменения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

Пример версионно-полиморфной структуры

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



Метаданные



Пример поля, имеющего внутреннюю структуру

- Image

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

- Plan
  (развесистое дерево, получаемое из EXPLAIN (FORMAT JSON)

Отступление

Как получить план в JSON и поместить его в таблицу

- 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 $$
   my ($sql) = @_;
   my $res = spi_exec_query(
        "EXPLAIN(FORMAT JSON) $sql", 1);
   return $res->{rows}[0]->{"QUERY PLAN"};
$$;
```

Postgres PROFESSIONAL

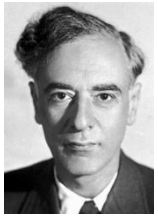
Второй интересный случай: Агрегаты

- 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.

JSON-агрегаты

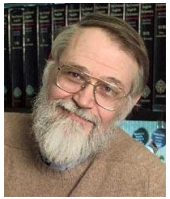
Теория поля.



Ландау Л.Д.,

Лифшиц Е.М.

Язык программирования C.



Керниган Б.В.,

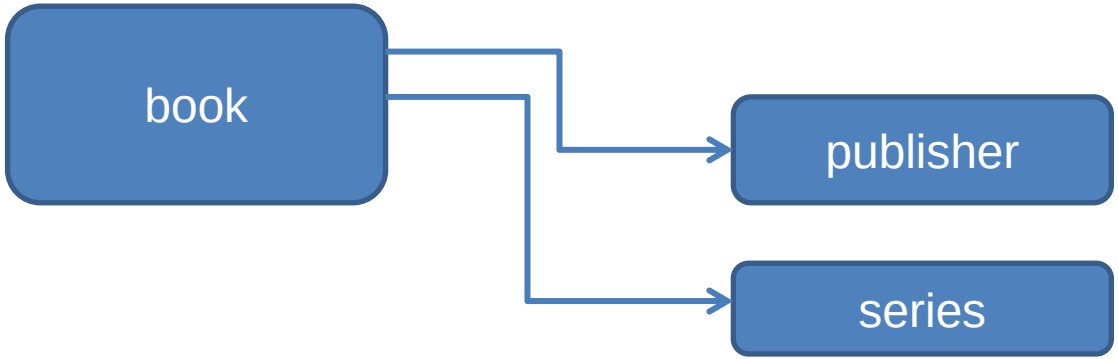
Ритчи Д.М.

JSON-агрегаты

```
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;
```

Другое использования JSON-агрегатов

- 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: достать книгу с рубрикацией

```
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
    ...
  ]
}
```

PROFESSIONAL Postgres

Задача: тестовые данные

```
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

```
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

Третий интересный случай: 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));
```

Четвёртый интересный случай: 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;
```

Четвертый интересный случай: 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)
```

Пятый интересный случай: JSON внутри PL/*

- SQL и PL/PgSQL с JSON работают громоздко, и по-видимому, неэффективно

А что PL/Perl, PL/Python, PL/v8 ?

- PL/Perl и PL/Python получают JSON текстом, если не использовать TRANSFORM
- PL/v8 получает JSON сразу во внутреннем представлении
- PL/v8 не входит в комплект PostgreSQL и обычно отстаёт по версиям

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 PL/Perl

```
CREATE EXTENSION plperl;
CREATE EXTENSION jsonb_plperl; -- определяет TRANSFORM
CREATE FUNCTION pl_agg_f (state jsonb, item int) RETURNS
    jsonb AS $$
    my ($state, $item) = @_;
    push @{$state->{list}}, $item;
    $state->{sum} += $item;
    return $state;
$$
LANGUAGE plperl TRANSFORM FOR TYPE jsonb IMMUTABLE STRICT;

CREATE AGGREGATE pl_agg (int)( SFUNC=pl_agg_f, STYPE=jsonb,
    INITCOND='{"list":[], "sum": 0}');
SELECT pl_agg (  (random()*20)::int ) FROM
    generate_series(1,20);
```

Тот же агрегат на SQL

```
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}');
```

Светлое будущее

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

Литература

- JSON, JSONB, JSQuery (PgConf.Russia 2017)
- PL/{Perl,Python,V8}
- JSONB в примерах (PgConf.EU 2019)

Спасибо за внимание!

Слайды, видео и статья на Хабре будут.

Вопросы: i.panchenko@postgrespro.ru