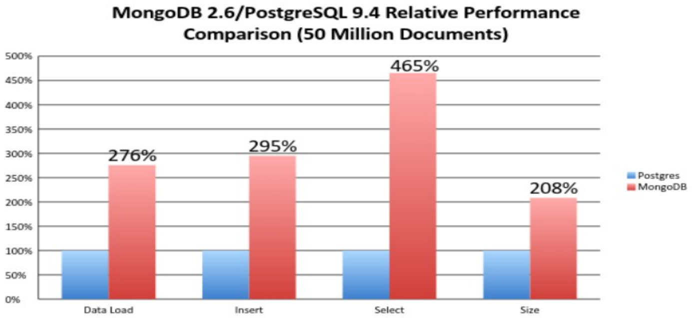
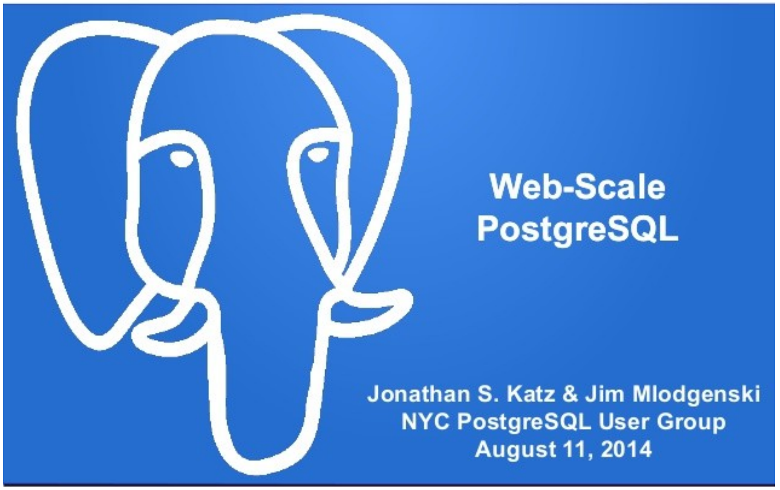
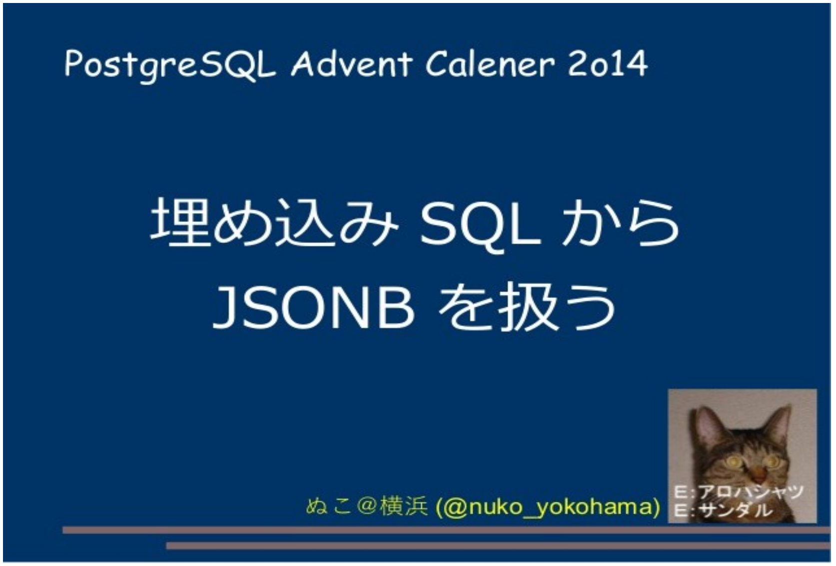
18 декабря 2014





	Postgres	MongoDB
Data Load (s)	4,732	13,046
Insert (s)	29,236	86,253
Select (s)	594	2,763
Size (GB)	69	145



Postgres' NoSQ
Capabilities

- HSTORE
  - Key-value pair
  - Simple, fast and easy
  - Postgres v 8.2 – pre-dates many NoSQL-only solutions
  - Ideal for flat data structures that are sparsely populated
- JSON
  - Hierarchical document model
  - Introduced in Postgres 9.2, perfected in 9.3
- JSONB
  - Binary version of JSON
  - Faster, more operators and even more robust
  - Postgres 9.4

JSONB Features

- Equality operator
  - `SELECT '{"a": 1, "b": 2}'::jsonb = '{"b":2, "a":1}'::jsonb`
- Containment operator (Softserve)
  - `SELECT '{"a": 1, "b": 2}'::jsonb @> {"b":2}::jsonb`
- Existence
  - `SELECT '{"a": 1, "b": 2}'::jsonb ? 'b';`

serve works as well)
}'::jsonb = '{"a":[1,2]}'::jsonb

Oleg Bartunov, Teodor Sigaev

- Locale support
- Extendability (indexing)
  - GiST (KNN), GIN, SP-GiST
- Full Text Search (FTS)
- Jsonb, VODKA
- Extensions:
  - intarray
  - pg_trgm
  - ltree
  - hstore
  - plantuner

https://www.facebook.com/oleg.bartunov
obartunov@gmail.com, teodor@sigaev.ru
https://www.facebook.com/groups/postgresql/

Alexander Korotkov

- Indexed regexp search
- GIN compression & fast scan
- Fast GiST build
- Range types indexing
- Split for GiST
- Indexing for jsonb
- jsquery
- Generic WAL + create am (WIP)

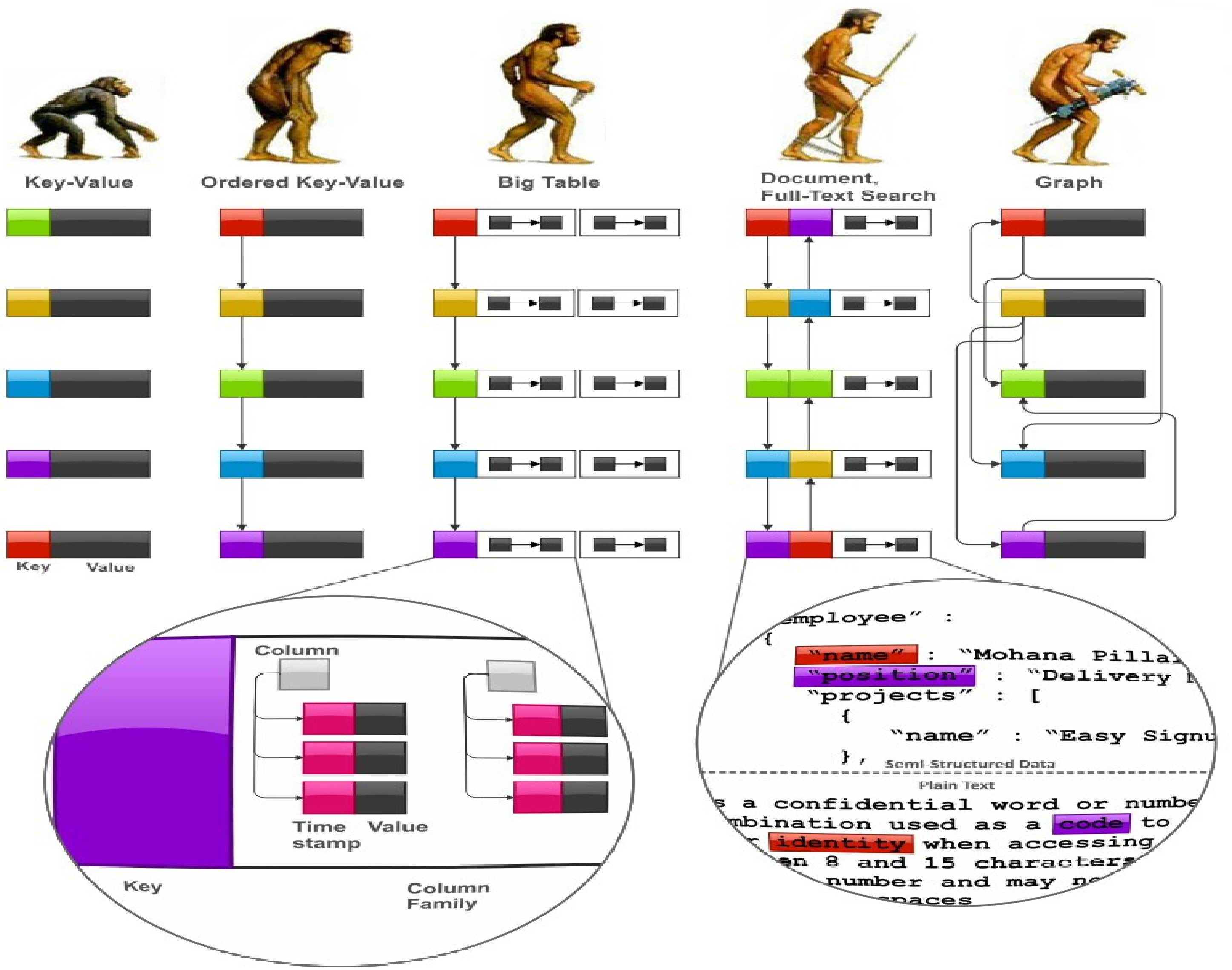
aekorotkov@gmail.com

The problem

- The world of data and applications is changing
- BIG DATA (**V**olume of data,**V**elocity of data in-out, **V**ariety of data)
- Web applications are service-oriented
  - Service itself can aggregate data, check consistency of data
  - High concurrency, simple queries
  - Simple database (key-value) is ok
  - Eventual consistency is ok, no ACID overhead
- Application needs faster releases
- NoSQL databases match all of these — scalable, efficient, fault-tolerant, no rigid schema, ready to accept any data.

NoSQL

- Key-value databases
  - Ordered k-v for ranges support
- Column family (column-oriented) stores
  - Big Table — value has structure:
    - column families, columns, and timestamped versions (maps-of maps-of maps)
- Document databases
  - Value has arbitrary structure
- Graph databases — evolution od ordered-kv



Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
Key
Value
Column
Time
stamp
Value
Key
Column
Family
Semi-Structured Data
Plain Text

The problem

- What if application needs ACID and flexibility of NoSQL ?
- Relational databases work with data with schema known in advance
- One of the major compaints to relational databases is rigid schema.
  It's not easy to change schema online (ALTER TABLE … ADD COLUMN...)
- Application should wait for schema changing, infrequent releases
- NoSQL uses json format, why not have it in relational database ?

JSON in PostgreSQL

This is the challenge !

Challenge to PostgreSQL !

- Full support of semi-stuctured data in PostgreSQL
  - Storage
  - Operators and functions
  - Efficiency (fast access to storage, indexes)
  - Integration with CORE (planner, optimiser)
- Actually, PostgreSQL is schema-less database since 2003 — hstore, one of the most popular extension !

Introduction to Hstore

id	col1	col2	col3	col4	col5	A lot of columns key1, …. keyN

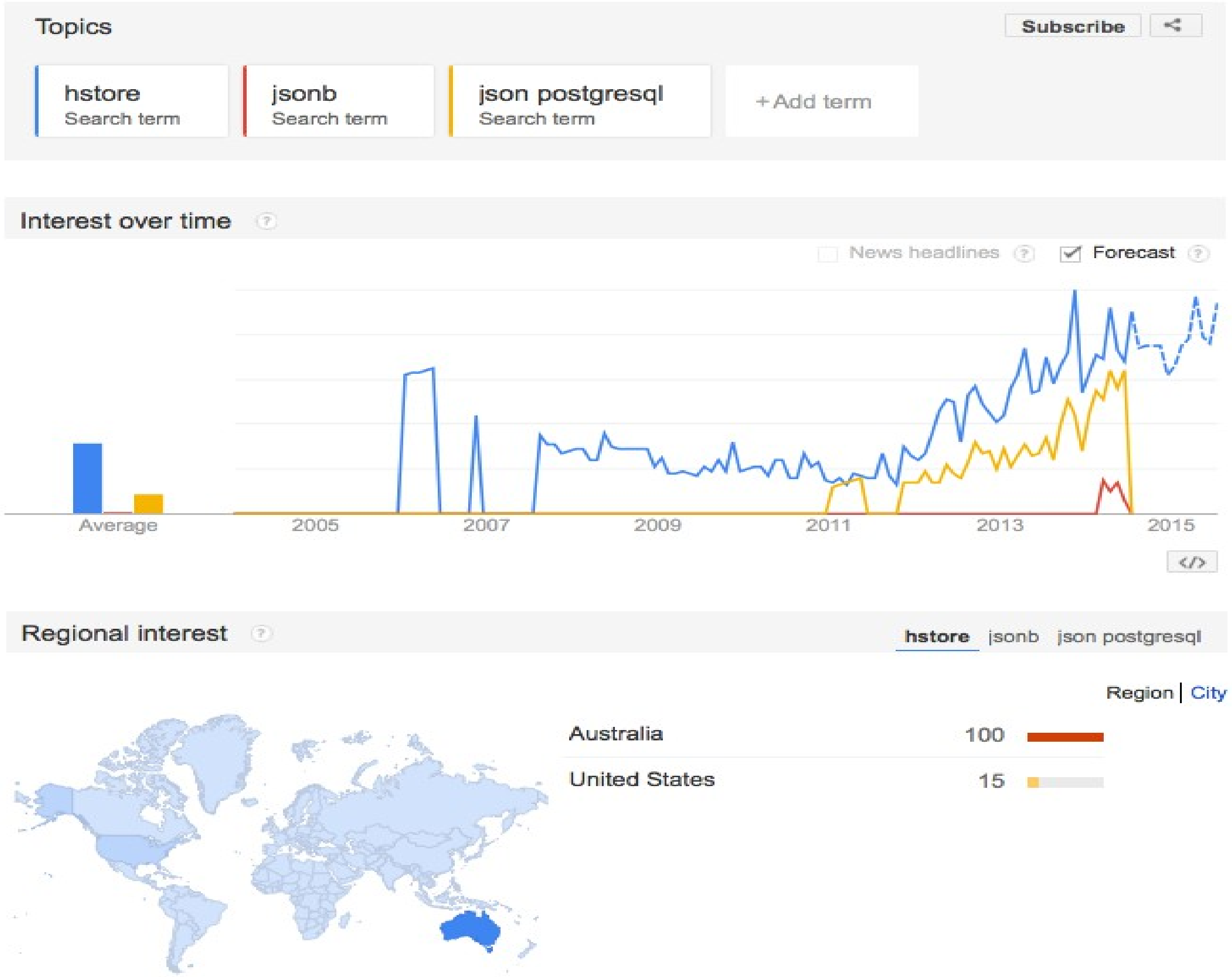
- The problem:
- Total number of columns may be very large
- Only several fields are searchable (used in WHERE)
- Other columns are used only to output
  - These columns may not known in advance
- Solution
  - New data type (hstore), which consists of (key,value) pairs (a'la perl hash)

Introduction to Hstore

id	col1	col2	col3	col4	col5	**Hstore** **key1=>val1, key2=>val2,.....**

- Easy to add key=>value pair
- No need change schema, just change hstore.
- Schema-less PostgreSQL in 2003 !

Google insights about hstore



Introduction to hstore

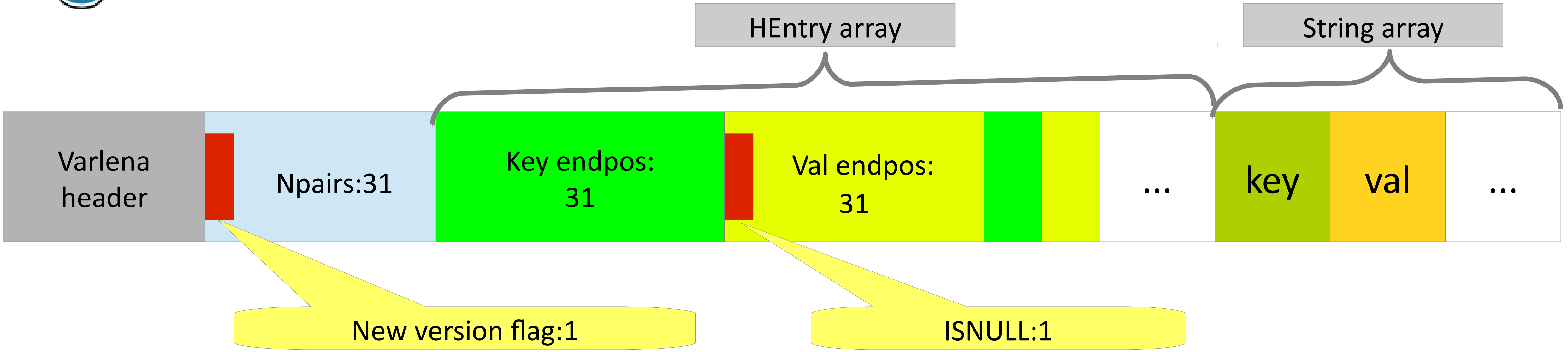
- Hstore — key/value binary storage (inspired by perl hash)

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

  - Key, value — strings
  - Get value for a key: hstore -> text
  - Operators with indexing support (GiST, GIN)

    Check for key: hstore ? text
    Contains: hstore @> hstore
  - check documentations for more
  - Functions for hstore manipulations (akeys, avals, skeys, svals, each,......)
- Hstore provides PostgreSQL schema-less feature !
  - Faster releases, no problem with schema upgrade

Hstore binary storage



	Start	End
First key	0	HEntry[0]
i-th key	HEntry[i*2 - 1]	HEntry[i*2]
i-th value	HEntry[i*2]	HEntry[i*2 + 1]

Pairs are lexicographically ordered by key

Hstore limitations

- Levels: unlimited
- Number of elements in array: 2^31
- Number of pairs in hash: 2^31
- Length of string: 2^31 bytes

2^31 bytes = 2 GB

Introduction to hstore

- Hstore benefits
  - In provides a flexible model for storing a semi-structured data in relational database
  - hstore has binary storage and rich set of operators and functions, indexes
- Hstore drawbacks
  - Too simple model !
    Hstore key-value model doesn't supports tree-like structures as json (introduced in 2006, 3 years after hstore)
- Json — popular and standartized (ECMA-404 The JSON Data Interchange Standard, JSON RFC-7159)
- Json — PostgreSQL 9.2, textual storage

Hstore vs Json

- hstore is faster than json even on simple data

```
CREATE TABLE hstore_test AS (SELECT
'a=>1, b=>2, c=>3, d=>4, e=>5'::hstore AS v
FROM generate_series(1,1000000));

CREATE TABLE json_test AS (SELECT
'{"a":1, "b":2, "c":3, "d":4, "e":5}'::json AS v
FROM generate_series(1,1000000));
```

```
SELECT sum((v->'a')::text::int) FROM json_test;
851.012 ms

SELECT sum((v->'a')::int) FROM hstore_test;
330.027 ms
```

Hstore vs Json

- PostgreSQL already has json since 9.2, which supports document-based model, but
  - It's slow, since it has no binary representation and needs to be parsed every time
  - Hstore is fast, thanks to binary representation and index support
  - It's possible to convert hstore to json and vice versa, but current hstore is limited to key-value
  - Need hstore with document-based model. Share it's binary representation with json !

Nested hstore

abstract

Oleg Bartunov <obartunov@gmail.com> 12/18/12

to Teodor

Поправь, дополни.

Title: One step forward true json data type. Nested hstore with array support.

We present a prototype of nested hstore data type with array support. We consider the new hstore as a step forward true json data type.

Recently, PostgreSQL got json data type, which basically is a string storage with validity checking for stored values and some related functions. To be a real data type, it has to have a binary representation, which could be a big project if started from scratch. Hstore is a popular data type, we developed years ago to facilitate working with semi-structured data in PostgreSQL. Our idea is to extend hstore to be nested (value can be hstore) data type and add support of arrays, so its binary representation can be shared with json. We present a working prototype of a new hstore data type and discuss some design and implementation issues.

Nested hstore & jsonb

- Nested hstore at PGCon-2013, Ottawa, Canada (May 24) — thanks Engine Yard for support !

  One step forward true json data type.Nested hstore with arrays support

- Binary storage for nested data at PGCon Europe — 2013, Dublin, Ireland (Oct 29)

  Binary storage for nested data structuresand application to hstore data type

- November, 2013 — binary storage was reworked, nested hstore and jsonb share the same storage. Andrew Dunstan joined the project.
- January, 2014 - binary storage moved to core

Nested hstore & jsonb

- Feb-Mar, 2014 - Peter Geoghegan joined the project, nested hstore was cancelled in favour to jsonb (Nested hstore patch for 9.3).
- Mar 23, 2014 Andrew Dunstan committed jsonb to 9.4 branch !
  pgsql: Introduce jsonb, a structured format for storing json.

Introduce jsonb, a structured format for storing json.

The new format accepts exactly the same data as the json type. However, it is stored in a format that does not require reparsing the orgiginal text in order to process it, making it much more suitable for indexing and other operations. Insignificant whitespace is discarded, and the order of object keys is not preserved. Neither are duplicate object keys kept - the later value for a given key is the only one stored.

Jsonb vs Json

```
SELECT '{"c":0,  "a":2,"a":1}'::json, '{"c":0,   "a":2,"a":1}'::jsonb;
         json          |      jsonb
-----------------------+------------------
 {"c":0,  "a":2,"a":1} | {"a": 1, "c": 0}
(1 row)
```

- json: textual storage «as is»
- jsonb: no whitespaces
- jsonb: no duplicate keys, last key win
- jsonb: keys are sorted

Summary: PostgreSQL 9.4 vs Mongo 2.6.0

- Поиск ключ=значение (contains @>)
  - json : 10 s seqscan
  - jsonb : 8.5 ms GIN jsonb_ops
  - **jsonb : 0.7 ms GIN jsonb_path_ops**
  - mongo : 1.0 ms btree index
- Index size
  - jsonb_ops - 636 Mb (no compression, 815Mb)
    jsonb_path_ops - 295 Mb
  - jsonb_path_ops (tags) - 44 Mb USING gin((jb->'tags') jsonb_path_ops
  - mongo (tags) - 387 Mb
    mongo (tags.term) - 100 Mb

- Table size
  - postgres : 1.3Gb
  - mongo : 1.8Gb
- Input performance:
  - Text : 34 s
  - Json : 37 s
  - Jsonb : 43 s
  - mongo : 13 m

Что сейчас может Jsonb ?

- Contains operators - jsonb @> jsonb, jsonb <@ jsonb (GIN indexes)

```
jb @> '{"tags":[{"term":"NYC"}]}'::jsonb
```

  Keys should be specified from root

- Equivalence operator — jsonb = jsonb (GIN indexes)
- Exists operators — jsonb ? text, jsonb ?! text[], jsonb ?& text[] (GIN indexes)

```
jb WHERE jb ?| '{tags,links}'
```

  Only root keys supported

- Operators on jsonb parts (functional indexes)

```
SELECT ('{"a": {"b":5}}'::jsonb -> 'a'->>'b')::int > 2;
CREATE INDEX ....USING BTREE ( (jb->'a'->>'b')::int);
```

  Very cumbersome, too many functional indexes

Найти что-нибудь красное

```
•      Table "public.js_test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
 value  | jsonb   |

select * from js_test;

 id |                                 value
----+-----------------------------------------------------------------------
  1 | [1, "a", true, {"b": "c", "f": false}]
  2 | {"a": "blue", "t": [{"color": "red", "width": 100}]}
  3 | [{"color": "red", "width": 100}]
  4 | {"color": "red", "width": 100}
  5 | {"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"}
  6 | {"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}
  7 | {"a": "blue", "t": [{"color": "blue", "width": 100}], "colr": "red"}
  8 | {"a": "blue", "t": [{"color": "green", "width": 100}]}
  9 | {"color": "green", "value": "red", "width": 100}
(9 rows)
```

Найти что-нибудь красное

```
WITH RECURSIVE t(id, value) AS ( SELECT * FROM
js_test
 UNION ALL
    (
      SELECT
        t.id,
        COALESCE(kv.value, e.value) AS value
      FROM
        t
        LEFT JOIN LATERAL
jsonb_each(
CASE WHEN jsonb_typeof(t.value) =
'object' THEN t.value
          ELSE NULL END) kv ON true
       LEFT JOIN LATERAL
jsonb_array_elements(
          CASE WHEN
jsonb_typeof(t.value) = 'array' THEN t.value
          ELSE NULL END) e ON true
        WHERE
          kv.value IS NOT NULL OR e.value IS
NOT NULL
     )
)
```

```
SELECT
  js_test.*
FROM
  (SELECT id FROM t WHERE value @> '{"color":
"red"}' GROUP BY id) x
  JOIN js_test ON js_test.id = x.id;
```

- Весьма непростое решение !

Что хочется ?

- Need Jsonb query language
  - Simple and effective way to search in arrays (and other iterative searches)
  - More comparison operators (сейчас только =)
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support

Jsonb query

- Need Jsonb query language
  - Simple and effective way to search in arrays (and other iterative searches)
  - More comparison operators
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support
- Introduce Jsquery - textual data type and @@ match operator

```
jsonb @@ jsquery
```

Postgres PROFESSIONAL

PGCon-2014, Май, Оттава

Jsonb query language (Jsquery)

- # - any element array

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# = 2';
```

- % - any key

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ '%.b.# = 2';
```

- * - anything

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ '*.# = 2';
```

- $ - current element

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# ($ = 2 OR $ < 3)';
```

- Use "double quotes" for key !

```
select 'a1."12222" < 111'::jsquery;
```

```
path     ::= key
           | path '.' key_any
           | NOT '.' key_any

key      ::= '*'
           | '#'
           | '%'
           | '$'
           | STRING
       ........

key_any  ::= key
           | NOT
```

Jsonb query language (Jsquery)

- Scalar

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# IN (1,2,5)';
```

- Test for key existence

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b = *';
```

- Array overlap

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b && [1,2,5]';
```

- Array contains

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b @> [1,2]';
```

- Array contained

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b <@ [1,2,3,4,5]';
```

```
value_expr
    ::= '=' scalar_value
      | IN '(' value_list ')'
      | '=' array
      | '=' '*'
      | '<' NUMERIC
      | '<' '=' NUMERIC
      | '>' NUMERIC
      | '>' '=' NUMERIC
      | '@' '>' array
      | '<' '@' array
      | '&' '&' array
      | IS ARRAY
      | IS NUMERIC
      | IS OBJECT
      | IS STRING
      | IS BOOLEAN
```

Jsonb query language (Jsquery)

- Type checking

```
select '{"x": true}' @@ 'x IS boolean'::jsquery,
       '{"x": 0.1}'  @@ 'x IS numeric'::jsquery;
 ?column? | ?column?
----------+----------
 t        | t
```

```
select '{"a":{"a":1}}' @@ 'a IS object'::jsquery;
 ?column?
----------
 t
```

```
select '{"a":["xxx"]}' @@ 'a IS array'::jsquery, '["xxx"]' @@ '$ IS array'::jsquery;
 ?column? | ?column?
----------+----------
 t        | t
```

IS BOOLEAN

IS NUMERIC

IS ARRAY

IS OBJECT

IS STRING

Jsonb query language (Jsquery)

- How many products are similar to "B000089778" and have product_sales_rank in range between 10000-20000 ?
- SQL

```
SELECT count(*) FROM jr WHERE (jr->>'product_sales_rank')::int > 10000
and (jr->> 'product_sales_rank')::int < 20000 and
 ....boring stuff
```

- Jsquery

```
SELECT count(*) FROM jr WHERE jr @@ ' similar_product_ids &&
["B000089778"] AND product_sales_rank( $ > 10000 AND $ < 20000)'
```

- Mongodb

```
db.reviews.find( { $and :[ {similar_product_ids: { $in ["B000089778"]}},
{product_sales_rank:{$gt:10000, $lt:20000}}] } ).count()
```

Найти что-нибудь красное

- ```
  WITH RECURSIVE t(id, value) AS ( SELECT * FROM
  js_test
   UNION ALL
      (
        SELECT
          t.id,
          COALESCE(kv.value, e.value) AS value
        FROM
          t
          LEFT JOIN LATERAL
  jsonb_each(
  CASE WHEN jsonb_typeof(t.value) =
  'object' THEN t.value
            ELSE NULL END) kv ON true
         LEFT JOIN LATERAL
  jsonb_array_elements(
            CASE WHEN
  jsonb_typeof(t.value) = 'array' THEN t.value
            ELSE NULL END) e ON true
          WHERE
            kv.value IS NOT NULL OR e.value IS
  NOT NULL
       )
  )
  ```

```
SELECT
  js_test.*
FROM
  (SELECT id FROM t WHERE value @> '{"color":
"red"}' GROUP BY id) x
  JOIN js_test ON js_test.id = x.id;
```

- **Jsquery**

  SELECT * FROM js_test
  WHERE
  value @@ '*.color = "red"';
```

Еще пример

- SQL

```
SELECT * FROM js_test2 js
WHERE NOT EXISTS (
  SELECT 1
  FROM
jsonb_array_elements(js.value) el
  WHERE EXISTS (
    SELECT 1
    FROM jsonb_each(el.value) kv
    WHERE NOT
kv.value::text::numeric BETWEEN
0.0 AND 1.0));
```

- Jsquery

```
SELECT * FROM js_test2 js
WHERE '#:.%:($ >= 0 AND $ <= 1)';
```

Jsonb query language (Jsquery)

```
explain( analyze, buffers) select count(*) from jb where jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                                QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=191517.30..191517.31 rows=1 width=0) (actual time=1039.422..1039.423 rows=1 loops=1)
   Buffers: shared hit=97841 read=78011
   ->  Seq Scan on jb  (cost=0.00..191514.16 rows=1253 width=0) (actual time=0.006..1039.310 rows=285 loops=1)
         Filter: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Rows Removed by Filter: 1252688
         Buffers: shared hit=97841 read=78011
 Planning time: 0.074 ms
 Execution time: 1039.444 ms

explain( analyze,costs off) select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
                             QUERY PLAN
--------------------------------------------------------------------
 Aggregate (actual time=891.707..891.707 rows=1 loops=1)
   ->  Seq Scan on jb (actual time=0.010..891.553 rows=285 loops=1)
         Filter: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
         Rows Removed by Filter: 1252688
 Execution time: 891.745 ms
```

Jsquery (indexes)

- GIN opclasses with jsquery support
  - jsonb_value_path_ops — use Bloom filtering for key matching
    `{"a":{"b":{"c":10}}} → 10.( bloom(a) or bloom(b) or bloom(c) )`
    - Good for key matching (wildcard support) , not good for range query
  - jsonb_path_value_ops — hash path (like jsonb_path_ops)
    `{"a":{"b":{"c":10}}} → hash(a.b.c).10`
    - No wildcard support, no problem with ranges

```
                          List of relations
 Schema |          Name          | Type  |  Owner   |    Table     |  Size   | Description
--------+------------------------+-------+----------+--------------+---------+-------------
 public | jb                     | table | postgres |              | 1374 MB |
 public | jb_value_path_idx      | index | postgres | jb           | 306 MB  |
 public | jb_gin_idx             | index | postgres | jb           | 544 MB  |
 public | jb_path_value_idx      | index | postgres | jb           | 306 MB  |
 public | jb_path_idx            | index | postgres | jb           | 251 MB  |
```

Jsquery (indexes)

```
explain( analyze,costs off) select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
                                                QUERY PLAN
--------------------------------------------------------------------------------------
 Aggregate (actual time=0.609..0.609 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb (actual time=0.115..0.580 rows=285 loops=1)
         Recheck Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on jb_value_path_idx (actual time=0.073..0.073 rows=285
             loops=1)
               Index Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
 Execution time: 0.634 ms
(7 rows)
```

Jsquery (indexes)

```
explain( analyze,costs off) select count(*) from jb where jb @@ '*.term = "NYC"';
                                               QUERY PLAN
--------------------------------------------------------------------------------------
 Aggregate (actual time=0.688..0.688 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb (actual time=0.145..0.660 rows=285 loops=1)
         Recheck Cond: (jb @@ '*."term" = "NYC"'::jsquery)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on jb_value_path_idx (actual time=0.113..0.113 rows=285
             loops=1)
               Index Cond: (jb @@ '*."term" = "NYC"'::jsquery)
 Execution time: 0.716 ms
(7 rows)
```

Jsquery (indexes)

```
explain (analyze, costs off) select count(*) from jr where
jr @@  ' similar_product_ids && ["B000089778"]';
                                                 QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate (actual time=0.359..0.359 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.084..0.337 rows=185 loops=1)
         Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=0.057..0.057 rows=185
             loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.394 ms
(7 rows)
```

Jsquery (indexes)

- No statistics, no planning :(

Not selective, better not use index!

```
explain (analyze, costs off) select count(*) from jr where
 jr @@ ' similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)';
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=126.149..126.149 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=126.057..126.143 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] &
 "product_sales_rank"($ > 10000 & $ < 20000))'::jsquery)
         Heap Blocks: exact=45
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=126.029..126.029
             rows=45 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] &
"product_sales_rank"($ > 10000 & $ < 20000))'::jsquery)
 Execution time: 129.309 ms !!!   No statistics
```

MongoDB 2.6.0

```
db.reviews.find(  {  $and :[ {similar_product_ids: { $in:["B000089778"]}},    {product_sales_rank:{$gt:10000, $lt:20000}}] } )
.explain()
{
        "n" : 45,
         ....................
        "millis" : 7,
        "indexBounds" : {
                "similar_product_ids" : [
                        [
                                "B000089778",
                                "B000089778"
                        ]
                ]
        },
}
```

index size = 400 MB just for similar_product_ids !!!

Jsquery (indexes)

- If we rewrite query and use planner

```
explain (analyze,costs off) select count(*) from jr where
jr @@ ' similar_product_ids && ["B000089778"]'
and  (jr->>'product_sales_rank')::int>10000 and (jr->>'product_sales_rank')::int<20000;
---------------------------------------------------------------------------------------
 Aggregate (actual time=0.479..0.479 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.079..0.472 rows=45 loops=1)
         Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
         Filter: ((((jr ->> 'product_sales_rank'::text))::integer > 10000) AND
(((jr ->> 'product_sales_rank'::text))::integer < 20000))
         Rows Removed by Filter: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=0.041..0.041 rows=185
             loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.506 ms   Potentially,  query could be faster Mongo !
```

Jsquery (optimizer)

- Jsquery now has built-in simple optimiser.

```
explain (analyze, costs off) select count(*) from jr where
jr @@ 'similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)'

-------------------------------------------------------------------------------

 Aggregate (actual time=0.422..0.422 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.099..0.416 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
         Rows Removed by Index Recheck: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=0.060..0.060
rows=185 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
 Execution time: 0.480 ms vs 7 ms MongoDB !
```

Jsquery (optimizer)

Jsquery now has built-in optimiser for simple queries.
Analyze query tree and push non-selective parts to recheck (like filter)

Selectivity classes:

1) Equality (x = c)
2) Range (c1 < x < c2)
3) Inequality (c > c1)
4) Is (x is type)
5) Any (x = *)

Jsquery (HINTING)

- If you know that inequality is selective then use HINT /* --index */

```
# explain (analyze, costs off) select count(*) from jr where jr @@ 'product_sales_rank /*-- index*/ >
3000000 AND review_rating = 5'::jsquery;
                                                       QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=12.307..12.307 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=11.259..12.244 rows=739 loops=1)
         Recheck Cond: (jr @@ '("product_sales_rank" /*-- index */  > 3000000 AND "review_rating" =
5)'::jsquery)
         Heap Blocks: exact=705
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=11.179..11.179 rows=739 loops=1)
               Index Cond: (jr @@ '("product_sales_rank" /*-- index */  > 3000000 AND "review_rating" =
5)'::jsquery)
 Execution time: 12.359 ms vs 1709.901 ms (without hint)
(7 rows)
```

Jsquery use case: schema specification

```
CREATE TABLE js (
    id serial primary key,
    v jsonb,
    CHECK(v @@ 'name IS STRING AND
                coords IS ARRAY AND
                NOT coords.# ( NOT (
                    x IS NUMERIC AND
                    y IS NUMERIC ) )'::jsquery));
```

Non-numeric coordinates don't exist => All coordinates are numeric

Jsquery use case: schema specification

```
# INSERT INTO js (v) VALUES ('{"name": "abc", "coords": [{"x": 1, "y": 2}, {"x": 3, "y": 4}]}');
INSERT 0 1
```

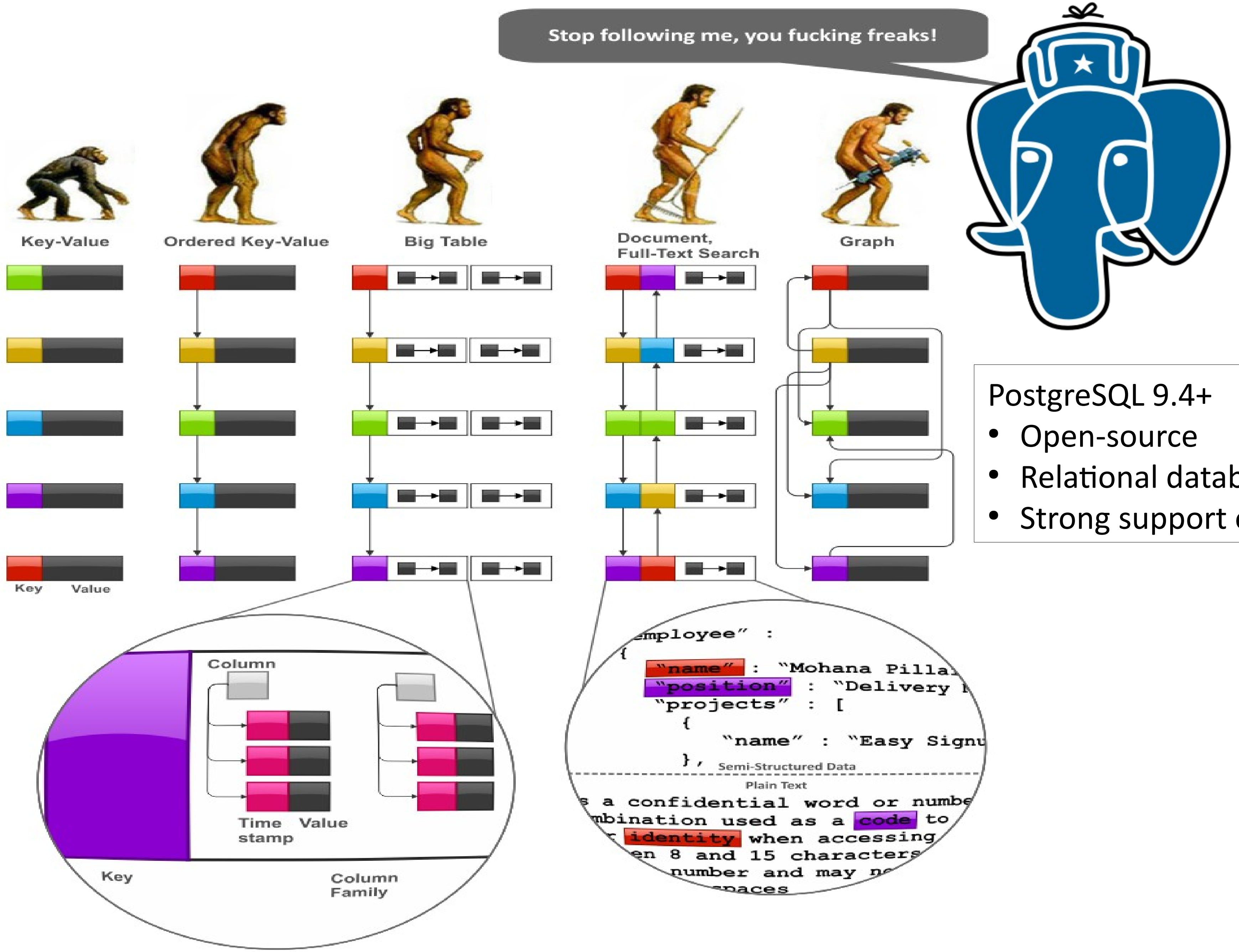
- ```
  # INSERT INTO js (v) VALUES ('{"name": 1, "coords": [{"x": 1, "y": 2}, {"x": "3", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint "js_v_check"
  ```

- ```
  # INSERT INTO js (v) VALUES ('{"name": "abc", "coords": [{"x": 1, "y": 2}, {"x": "zzz", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint "js_v_check"
  ```

Contrib/jsquery

- Jsquery index support is quite efficient (0.5 ms vs Mongo 7 ms !)
- Future direction
  - Make jsquery planner friendly
  - Need statistics for jsonb
- Availability
  - Jsquery + opclasses are available as extensions
  - Grab it from https://github.com/akorotkov/jsquery (branch master) , we need your feedback !

PROFESSIONAL Postgres

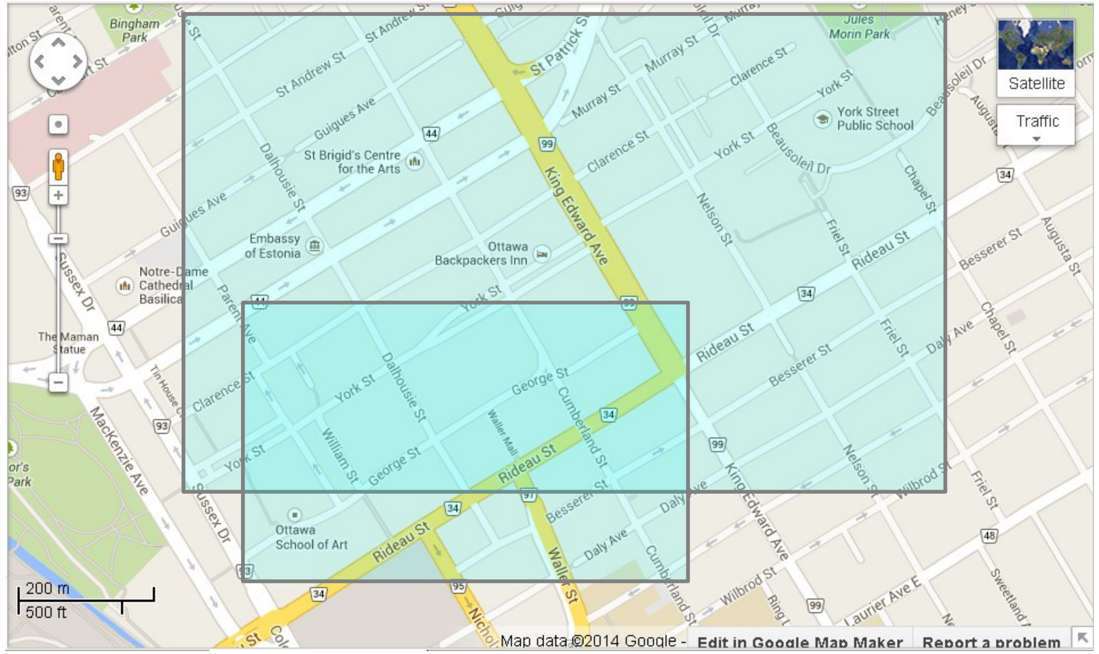


PostgreSQL 9.4+

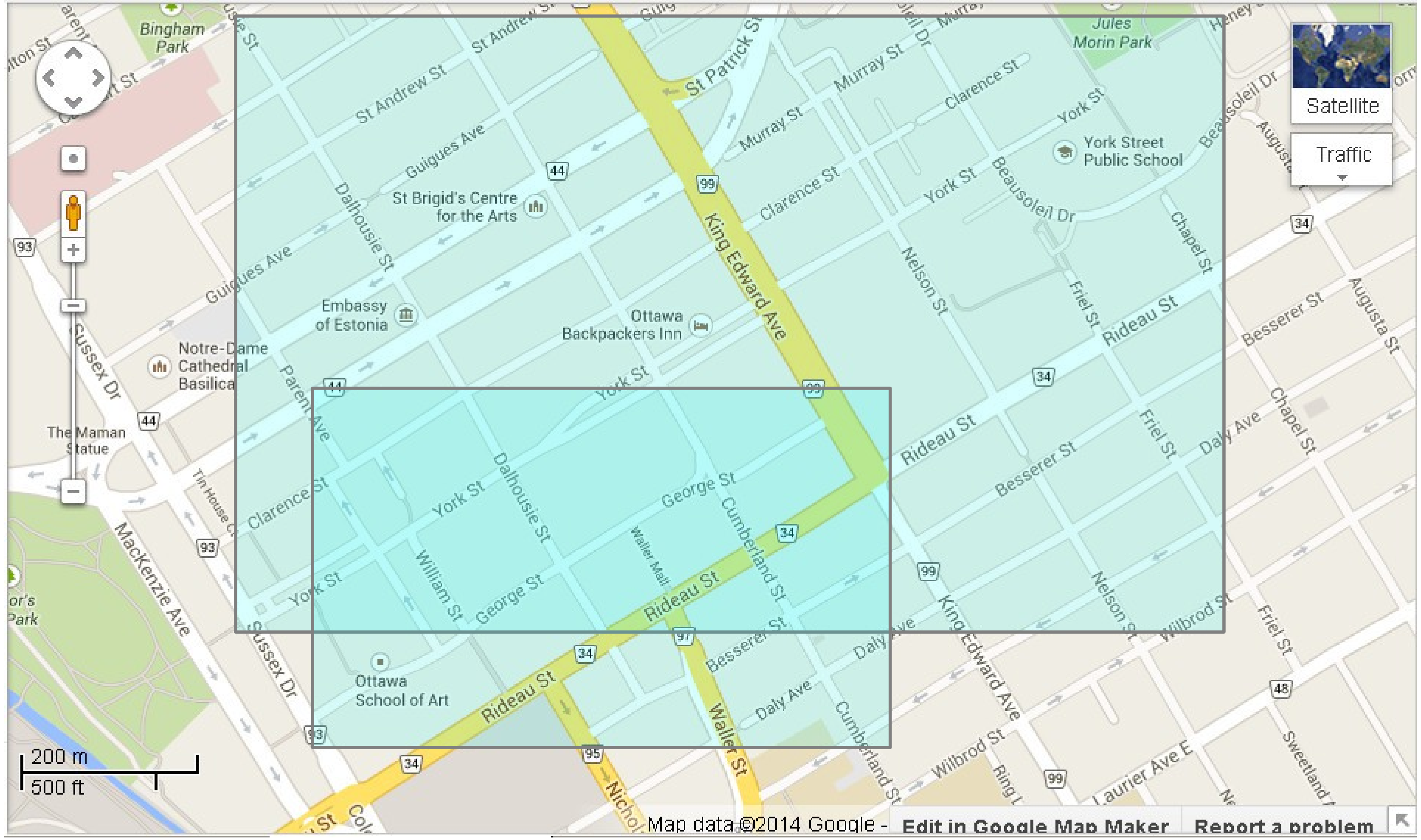
- Open-source
- Relational database
- Strong support of json

Что дальше ?

- SQL-level jsquery (расширяемость, статистика)
- VODKA access method ! VODKA Optimized Dendriform Keys Array
  - Комбинация произвольных методов доступа

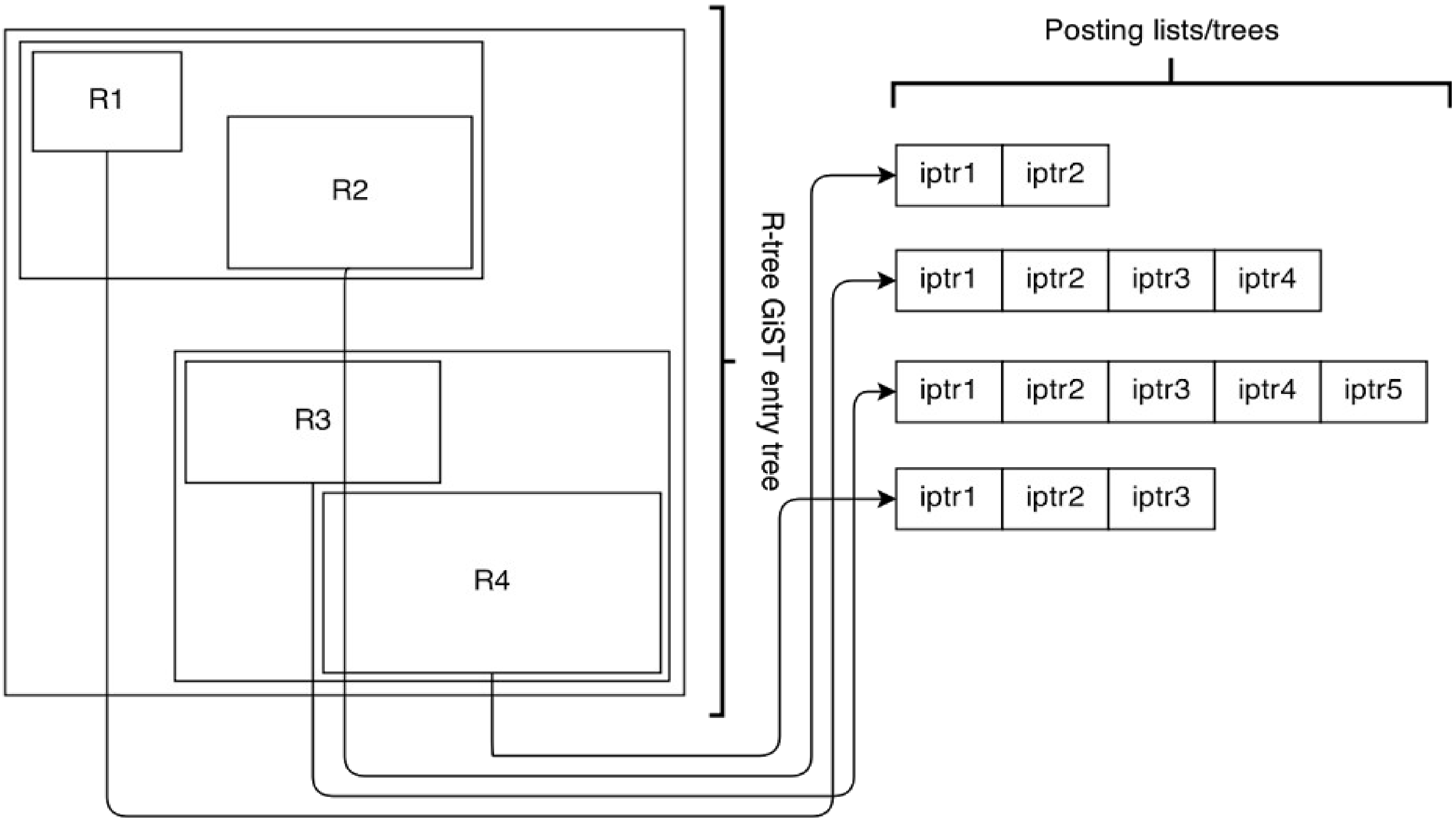


Ottawa downtown: York and George streets



Idea: Use multiple boxes

Rtree Vodka



Posting lists/trees
R1
R2
R3
R4
R-tree GiST entry tree
iptr1 iptr2
iptr1 iptr2 iptr3 iptr4
iptr1 iptr2 iptr3 iptr4 iptr5
iptr1 iptr2 iptr3

PROFESSIONAL
Postgres

Thanks for support

WORLD OF TANKS

WARGAMING.NET

LET'S BATTLE

postgrespro.ru

- Ищем инженеров:
  - 24x7 поддержка
  - Консалтинг & аудит
  - Разработка админских приложений
  - Пакеты
- Ищем си-шников для работы над постгресом:
  - Неубиваемый и масштабируемый кластер
  - Хранилища (in-memory, column-storage...)