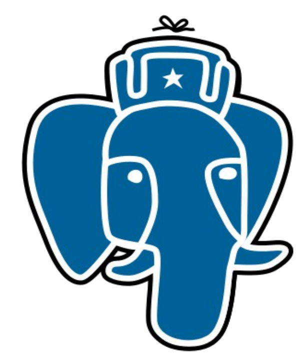
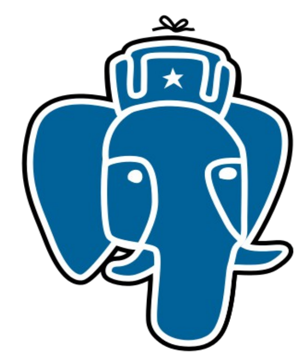


CREATE INDEX ... USING VODKA

An efficient indexing of nested structures

Oleg Bartunov (MSU), Teodor Sigaev (MSU),

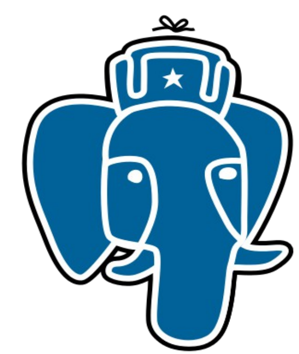
Alexander Korotkov (MEPhI)



Oleg Bartunov, Teodor Sigaev

- Locale support
- Extendability (indexing)
  - GiST, GIN, SP-GiST
- Extensions:
  - intarray
  - pg_trgm
  - ltree
  - hstore, hstore v2.0 → jsonb
  - plantuner
- Full Text Search (FTS)
- KNN-GiST

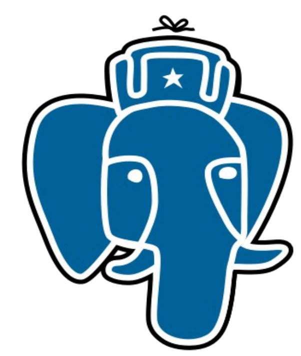
https://www.facebook.com/oleg.bartunov
obartunov@gmail.com, teodor@sigaev.ru



Alexander Korotkov

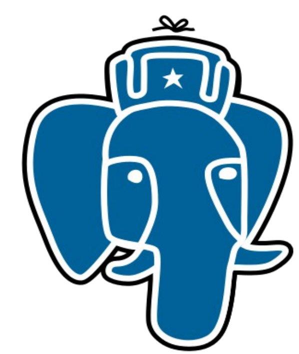
- Indexed regexp search
- GIN compression & fast scan
- Fast GiST build
- Range types indexing
- Split for GiST

aekorotkov@gmail.com



Agenda

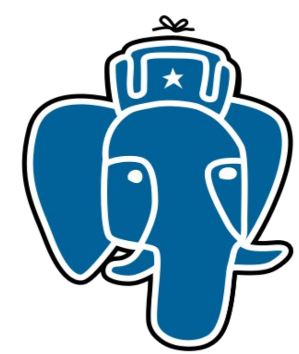
- Introduction to jsonb indexing
- Jsquery - Jsonb Query Language
- Exercises on jsonb GIN opclasses with Jsquery support
- VODKA access method



Schema-less data in PostgreSQL

key-value model → document-based model

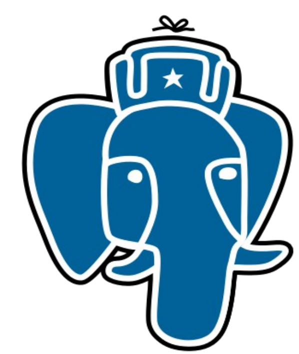
- First unpublished version of hstore (May 16, 2003)
- Dec 05, 2006 - hstore is a part of PostgreSQL 8.2
- May 23, 2007 - GIN index for hstore, PostgreSQL 8.3
- Sep, 20, 2010 - Andrew Gierth improved hstore, PostgreSQL 9.0
- Json data type (text), PostgreSQL 9.2
- One step forward true json data type.Nested hstore with arrays support - PGCon-2013
- Binary storage for nested data structuresand application to hstore data type -PGConf-2013
- pgsql: Introduce jsonb, a structured format for storing json. - 9.4dev, Mar 23, 2014
- A lot of community work on improving jsonb — still ...



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)
```

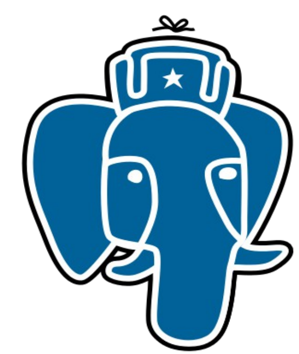
- json: textual storage «as is»
- jsonb: no whitespaces
- jsonb: no duplicate keys, last key win
- jsonb: keys are sorted



Jsonb vs Json

- Data
  - 1,252,973 Delicious bookmarks
- Server
  - MBA, 8 GB RAM, 256 GB SSD
- Test
  - Input performance - copy data to table
  - Access performance - get value by key
  - Search performance contains @> operator

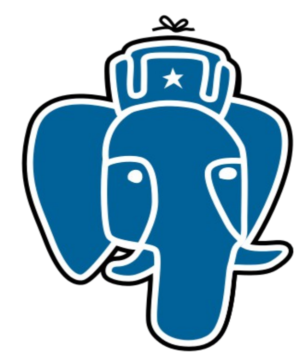

```
{
    "author": "mcasas1",
    "comments": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422",
    "guidislink": false,
    "id": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422#mcasas1",
    "link": "http://www.theatermania.com/broadway/",
    "links": [
        {
            "href": "http://www.theatermania.com/broadway/",
            "rel": "alternate",
            "type": "text/html"
        }
    ],
    "source": {},
    "tags": [
        {
            "label": null,
            "scheme": "http://delicious.com/mcasas1/",
            "term": "NYC"
        }
    ],
    "title": "TheaterMania",
    "title_detail": {
        "base": "http://feeds.delicious.com/v2/rss/recent?min=1&count=100",
        "language": null,
        "type": "text/plain",
        "value": "TheaterMania"
    },
    "updated": "Tue, 08 Sep 2009 23:28:55 +0000",
    "wfw_commentrss": "http://feeds.delicious.com/v2/rss/url/b5b3cbf9a9176fe43c27d7b4af94a422"
}
```



Jsonb vs Json

- Data
  - 1,252,973 bookmarks from Delicious in json format (js)
  - The same bookmarks in jsonb format (jb)
  - The same bookmarks as text (tx)

```
=# \dt+
                   List of relations
 Schema | Name | Type  |  Owner   |  Size   | Description
--------+------+-------+----------+---------+-------------
 public | jb   | table | postgres | 1374 MB | overhead is < 4%
 public | js   | table | postgres | 1322 MB |
 public | tx   | table | postgres | 1322 MB |
```



Jsonb vs Json

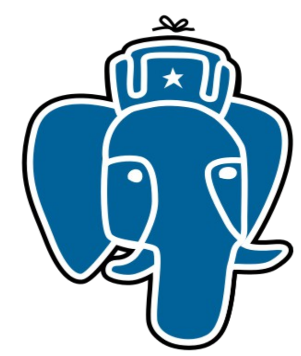
- Input performance (parser)
  Copy data (1,252,973 rows) as text, json,jsonb

  copy tt from '/path/to/test.dump'

Text: 34 s - as is

Json: 37 s - json validation

Jsonb: 43 s - json validation, binary storage

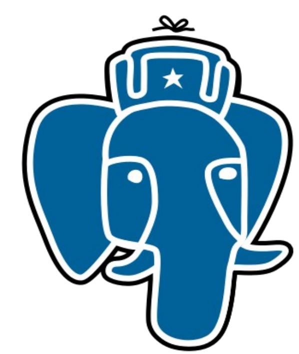


Jsonb vs Json (binary storage)

- Access performance — get value by key
  - Base: SELECT js FROM js;
  - Jsonb: SELECT j->>'updated' FROM jb;
  - Json: SELECT j->>'updated' FROM js;

Base:	0.6 s	
Jsonb:	1 s	0.4
Json:	9.6 s	9

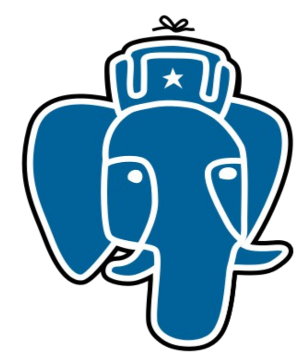
Jsonb ~ 20X faster Json



Jsonb vs Json

```
EXPLAIN ANALYZE SELECt count(*) FROM js WHERE js #>>'{tags,0,term}' = 'NYC';
                                                              QUERY PLAN
----------------------------------------------------------------------------
 Aggregate  (cost=187812.38..187812.39 rows=1 width=0)
(actual time=10054.602..10054.602 rows=1 loops=1)
   ->  Seq Scan on js  (cost=0.00..187796.88 rows=6201 width=0)
(actual time=0.030..10054.426 rows=123 loops=1)
         Filter: ((js #>> '{tags,0,term}'::text[]) = 'NYC'::text)
         Rows Removed by Filter: 1252850
 Planning time: 0.078 ms
 Execution runtime: 10054.635 ms
(6 rows)
```

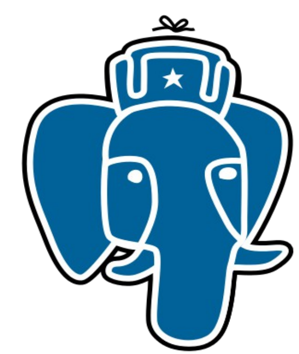
Json: no contains @> operator,
search first array element



Jsonb vs Json (binary storage)

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                                            QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate  (cost=191521.30..191521.31 rows=1 width=0)
(actual time=1263.201..1263.201 rows=1 loops=1)
   ->  Seq Scan on jb  (cost=0.00..191518.16 rows=1253 width=0)
 (actual time=0.007..1263.065 rows=285 loops=1)
         Filter: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Rows Removed by Filter: 1252688
 Planning time: 0.065 ms
 Execution runtime: 1263.225 ms          Execution runtime: 10054.635 ms
(6 rows)
```

Jsonb ~ 10X faster Json

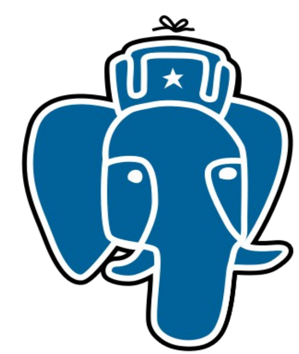


Jsonb vs Json (GIN: key && value)

CREATE INDEX gin_jb_idx ON jb USING gin(jb);

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                                          QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate  (cost=4772.72..4772.73 rows=1 width=0)
(actual time=8.486..8.486 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb  (cost=73.71..4769.59 rows=1253 width=0)
(actual time=8.049..8.462 rows=285 loops=1)
         Recheck Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on gin_jb_idx  (cost=0.00..73.40 rows=1253 width=0)
(actual time=8.014..8.014 rows=285 loops=1)
               Index Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
 Planning time: 0.115 ms
 Execution runtime: 8.515 ms           Execution runtime: 10054.635 ms
(8 rows)
```

Jsonb ~ 150X faster Json

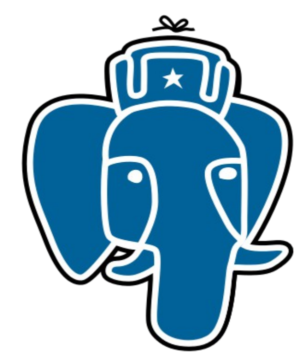


Jsonb vs Json (GIN: hash path.value)

CREATE INDEX gin_jb_path_idx ON jb USING gin(jb jsonb_path_ops);

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                                              QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=4732.72..4732.73 rows=1 width=0)
(actual time=0.644..0.644 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb  (cost=33.71..4729.59 rows=1253 width=0)
(actual time=0.102..0.620 rows=285 loops=1)
         Recheck Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on gin_jb_path_idx
(cost=0.00..33.40 rows=1253 width=0) (actual time=0.062..0.062 rows=285 loops=1)
               Index Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
 Planning time: 0.056 ms
 Execution runtime: 0.668 ms        Execution runtime: 10054.635 ms
(8 rows)
```

Jsonb ~ 1800X faster Json



MongoDB 2.6.0

- Load data - ~13 min **SLOW !** **Jsonb 43 s**

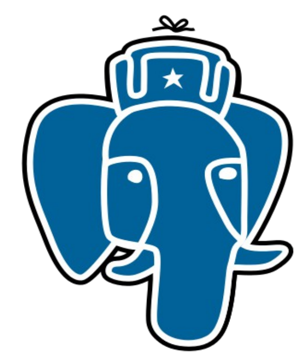
```
mongoimport --host localhost -c js --type json < delicious-rss-1250k
2014-04-08T22:47:10.014+0400                3700     1233/second
...
2014-04-08T23:00:36.050+0400                1252000  1547/second
2014-04-08T23:00:36.565+0400 check 9 1252973
2014-04-08T23:00:36.566+0400 imported 1252973 objects
```

- Search - ~ 1s (seqscan) **THE SAME**

```
db.js.find({tags: {$elemMatch:{ term: "NYC"}}}).count()
285
-- 980 ms
```

- Search - ~ 1ms (indexscan) **Jsonb 0.7ms**

```
db.js.ensureIndex( {"tags.term" : 1} )
db.js.find({tags: {$elemMatch:{ term: "NYC"}}}).
```

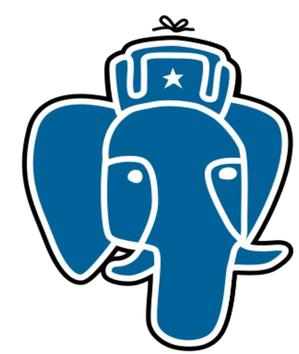


Summary: PostgreSQL 9.4 vs Mongo 2.6.0

- Operator 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
    jsonb_path_ops (tags.term) - 1.6 Mb
  - mongo (tags) - 387 Mb
    mongo (tags.term) - 100 Mb

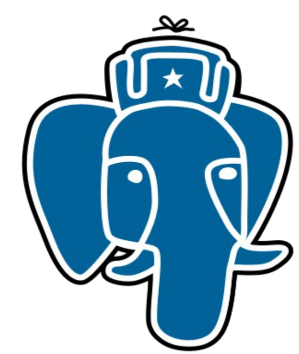
- Table size
  - postgres : 1.3Gb
  - mongo : 1.8Gb
- Input performance:
  - Text : 34 s
  - Json : 37 s
  - Jsonb : 43 s
  - mongo : 13 m

Engine Yard™



Jsonb query

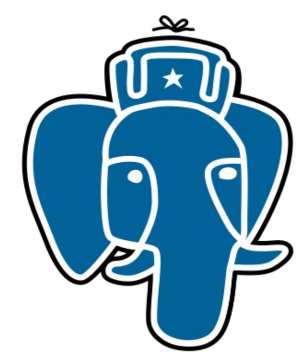
- Currently, one can search jsonb data using
  - Contains operators - jsonb @> jsonb, jsonb <@ jsonb (GIN indexes)
    `jb @> '{"tags":[{"term":"NYC"}]}'::jsonb`
    Keys should be specified from root
  - Equivalence operator — jsonb = jsonb (GIN indexes)
  - Exists operators — jsonb ? text, jsonb ?! 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



Jsonb query

- Need Jsonb query language
  - More operators on keys, values
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support
- Introduce Jsquery - textual data type and @@ match operator

```
jsonb @@ jsquery
```



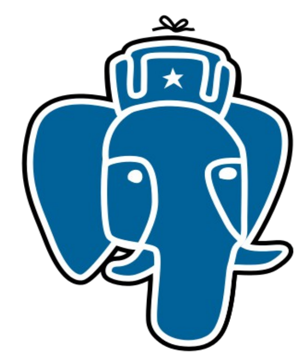
Jsonb query language (Jsquery)

```
expr     ::= path value_expr
           | path '(' expr ')'
           | '(' expr ')'
           | '!' expr
           | expr '&' expr
           | expr '|' expr
```

```
value_expr
        ::= '=' scalar_value
          | IN '(' value_list ')'
          | '=' array
          | '=' '*'
          | '<' NUMERIC
          | '<' '=' NUMERIC
          | '>' NUMERIC
          | '>' '=' NUMERIC
          | '@' '>' array
          | '<' '@' array
          | '&' '&' array
```

```
path     ::= path_elem
           | path '.' path_elem

path_elem
        ::= '*'
          | '#'
          | '%'
          | '$'
          | key

key     ::= STRING
          | true
          | false
          | NUMERIC
          | null
          | IN
```

```
value_list
        ::= scalar_value
          | value_list ',' scalar_value

array    ::= '[' value_list ']'

scalar_value
        ::= null
          | STRING
          | IN
          | true
          | false
          | NUMERIC
```



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';
```

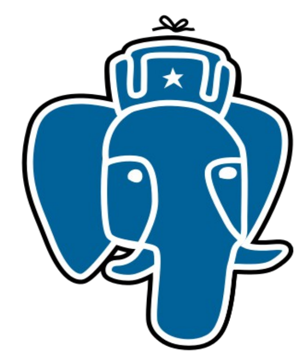
- $ - current element

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

- Use "double quotes" for key !

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

```
path      ::= path_elem
            | path '.' path_elem

path_elem
          ::= '*'
            | '#'
            | '%'
            | '$'
            | key

key       ::= STRING
            | true
            | false
            | NUMERIC
            | null
            | IN
```



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]';
```

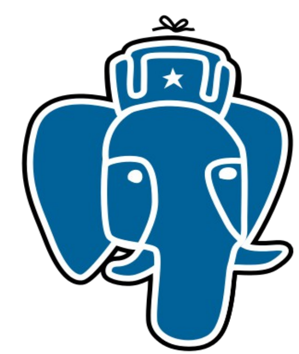
- Array contains

```
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
```



Jsonb query language (Jsquery)

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

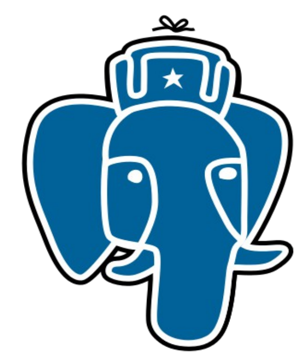
```
SELECT count(*) FROM jr WHERE (jr->>'product_sales_rank')::int > 10000
and (jr->> 'product_sales_rank')::int < 20000 and
 ....boring stuff
```

- Jsquery

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

- Mongodb

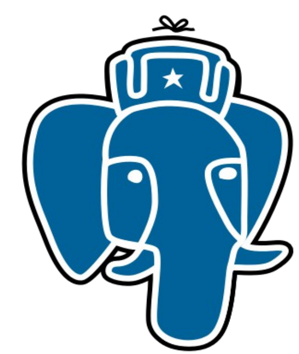
```
db.reviews.find( { $and :[ {similar_product_ids: { $in ["B000089778"]}},
{product_sales_rank:{$gt:10000, $lt:20000}}] } ).count()
```



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
```

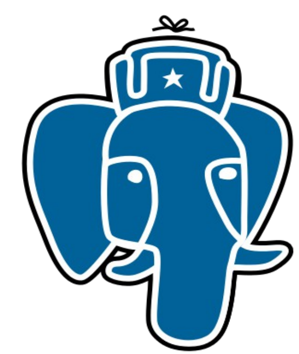


Jsquery (indexes)

- 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

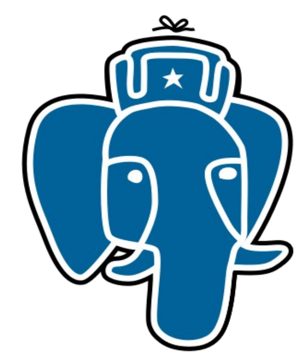
List of relations

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	



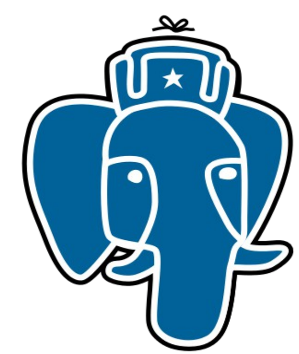
Jsquery (indexes)

```
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)

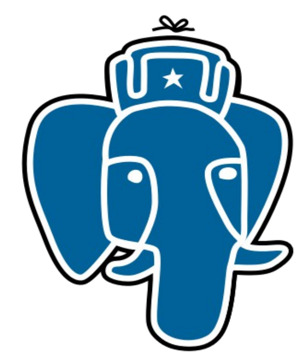
```
explain( analyze,costs off) select count(*) from jb where jb @@ '*.term = "NYC"';
                                      QUERY PLAN
--------------------------------------------------------------------------------------------------
 Aggregate (actual time=0.688..0.688 rows=1 loops=1)
   ->  Bitmap 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)
```



Citus dataset

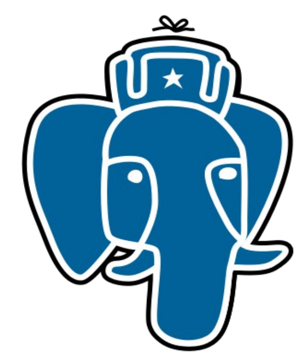
- 3023162 reviews from Citus 1998-2000 years
- 1573 MB


```
{
  "customer_id": "AE22YDHSBFYIP",
  "product_category": "Business & Investing",
  "product_group": "Book",
  "product_id": "1551803542",
  "product_sales_rank": 11611,
  "product_subcategory": "General",
  "product_title": "Start and Run a Coffee Bar (Start & Run a)",
  "review_date": {
    "$date": 31363200000
  },
  "review_helpful_votes": 0,
  "review_rating": 5,
  "review_votes": 10,
  "similar_product_ids": [
    "0471136174",
    "0910627312",
    "047112138X",
    "0786883561",
    "0201570483"
  ]
}
```



Jsquery (indexes)

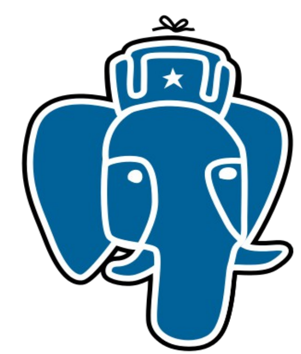
```
explain (analyze, costs off) select count(*) from jr where
jr @@  ' similar_product_ids && ["B000089778"]';
                                              QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=0.359..0.359 rows=1 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 :(

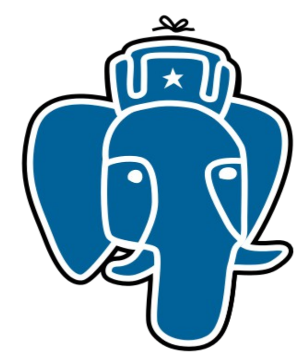
```
explain (analyze, costs off) select count(*) from jr where
 jr @@ ' similar_product_ids && ["B000089778"] & product_sales_rank( $ > 10000 & $  < 20000)';
                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------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



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"
                        ]
                ]
        },
}
```

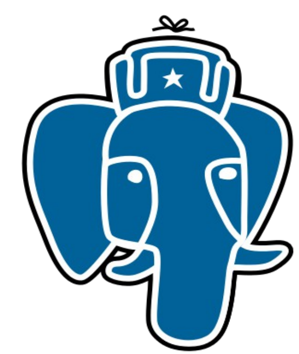
index size = 400 MB just for similar_product_ids !!!



Jsquery (indexes)

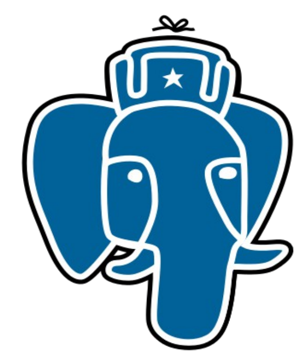
- Need statistics and planner support !

```
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
(9 rows)
```



Contrib/jsquery

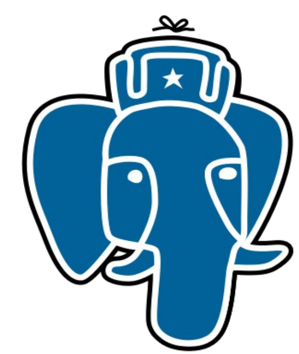
- PostgreSQL has potential to execute jsquery for 0.5 ms vs Mongo 7 ms !
  - Need statistics
  - Need planner
- Availability
  - Jsquery + opclasses are available as extensions
  - Grab it from https://github.com/akorotkov/jsquery (branch master) ,
    we need your feedback !
  - We will release it after PostgreSQL 9.4 release
  - Need real sample data and queries !



Better indexing ...

- GIN is a proven and effective index access method
- Need indexing for jsonb with operations on paths (no hash!) and values
  - B-tree in entry tree is not good - length limit, no prefix compression

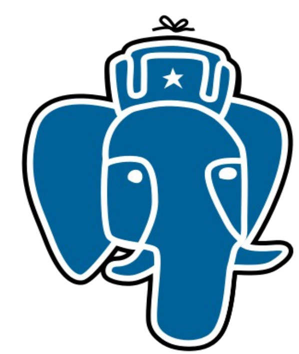
```
                                   List of relations
 Schema |             Name              | Type  |  Owner   |     Table       |  Size    | Description
--------+-------------------------------+-------+----------+-----------------+----------+-------------
 public | jb                            | table | postgres |                 | 1374 MB |
 public | jb_uniq_paths                 | table | postgres |                 | 912 MB  |
 public | jb_uniq_paths_btree_idx       | index | postgres | jb_uniq_paths | 885 MB  |text_pattern_ops
 public | jb_uniq_paths_spgist_idx      | index | postgres | jb_uniq_paths | 598 MB  |now much less !
```



Better indexing ...

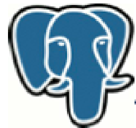
- Provide interface to change hardcoded B-tree in Entry tree
  - Use spgist opclass for storing paths and values as is (strings hashed in values)
- We may go further - provide interface to change hardcoded B-tree in posting tree
  - GIS aware full text search !
- New index access method

```
CREATE INDEX … USING VODKA
```



GIN History

- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev

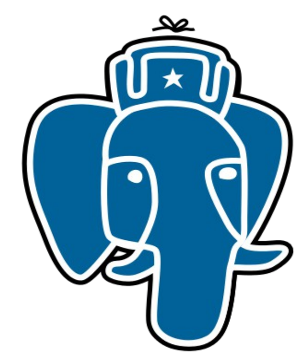


Generalized Inverted Index

- An inverted index is an index structure storing a set of (key, posting list) pairs, where 'posting list' is a set of documents in which the key occurs.

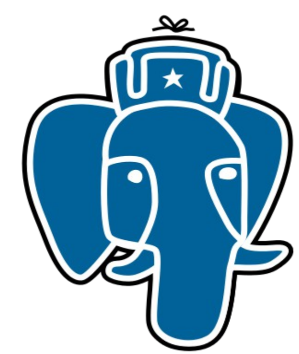
- Generalized means that the index does not know which operation it accelerates. It works with custom strategies, defined for specific data types. GIN is similar to GiST and differs from B-Tree indices, which have predefined, comparison-based operations.

Oleg Bartunov, Teodor Sigaev   PostgreSQL Summit, Toronto, July 8-9, 2006



GIN History

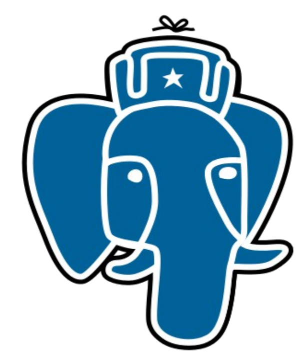
- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev
- Supported by JFG Networks (France)
- «Gin stands for Generalized Inverted iNdex and should be considered as a genie, not a drink.»
- Alexander Korotkov, Heikki Linnakangas have joined GIN++ development in 2013



GIN History

- From GIN Readme, posted in -hackers, 2006-04-26

```
TODO
----

Nearest future:

  * Opclasses for all types (no programming, just many catalog changes).

Distant future:

  * Replace B-tree of entries to something like GiST (VODKA ! 2014)
  * Add multicolumn support
  * Optimize insert operations (background index insertion)
```



GIN index structure for jsonb

```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```

product_sales_rank 15000

product_group 0x2F 0xF1 0x9A 0xB2

product_group 0xF5 0x17 0x50 0xD4

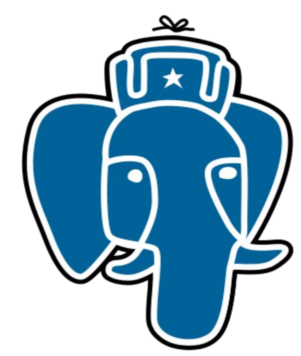
product_sales_rank 15000

product_sales_rank 25000

btree entry tree

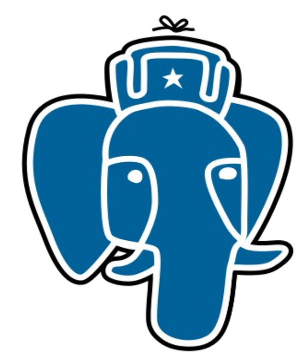
Posting lists/trees

iptr1	iptr2			
iptr1	iptr2	iptr3	iptr4	
iptr1	iptr2	iptr3	iptr4	iptr5
iptr1	iptr2	iptr3		



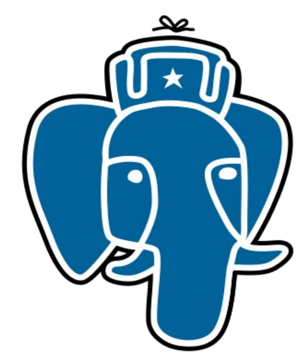
Vodka index structure for jsonb

```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```



Vodka distilling instructions

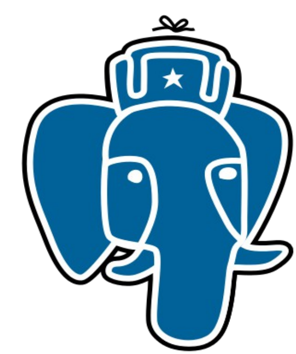
- config — configures parameters
  - entry tree opclass
  - equality operator
- compare — compares entry tree parameters (as in GIN)
- extract value — decompose datum into entries (as in GIN)
- extract query — decompose query into keys:
  - operator to scan entry tree
  - argument to scan entry tree
- consistent — check if item satisfies query (as in GIN)
- triconsistent — check if item satisfies query in ternary logic (as in GIN)



CREATE INDEX … USING VODKA

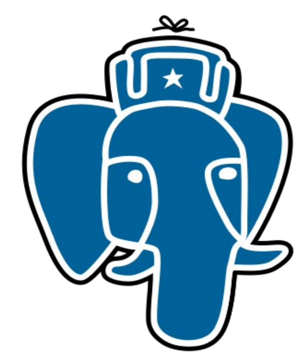
- Delicious bookmarks, mostly text data

```
set maintenance_work_mem = '1GB';

                              List of relations
 Schema |        Name         | Type  |  Owner    | Table |  Size    | Description
--------+---------------------+-------+-----------+-------+----------+-------------
 public | jb                  | table | postgres |       | 1374 MB |  1252973 rows
 public | jb_value_path_idx   | index | postgres | jb    | 306 MB  |   98769.096
 public | jb_gin_idx          | index | postgres | jb    | 544 MB  |  129860.859
 public | jb_path_value_idx   | index | postgres | jb    | 306 MB  |  100560.313
 public | jb_path_idx         | index | postgres | jb    | 251 MB  |   68880.320
 public | jb_vodka_idx        | index | postgres | jb    | 409 MB  |  185362.865
 public | jb_vodka_idx5       | index | postgres | jb    | 325 MB  |  174627.234 new spgist
(6 rows)
```



CREATE INDEX … USING VODKA

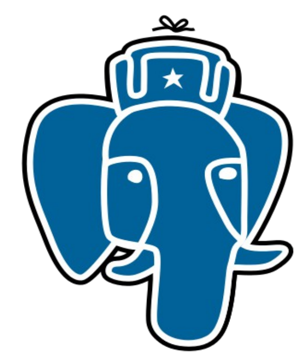
```
select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
--------------------------------------------------------------------------------------
 Aggregate (actual time=0.423..0.423 rows=1 loops=1)
   -> Bitmap Heap Scan on jb (actual time=0.146..0.404 rows=285 loops=1)
        Recheck Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on jb_vodka_idx (actual time=0.108..0.108 rows=285 loops=1)
             Index Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
 Execution time: 0.456 ms  (0.634 ms,  GIN jsonb_value_path_ops)

select count(*) from jb where jb @@ '*.term = "NYC"';
--------------------------------------------------------------------------------------
 Aggregate (actual time=0.495..0.495 rows=1 loops=1)
   -> Bitmap Heap Scan on jb (actual time=0.245..0.474 rows=285 loops=1)
        Recheck Cond: (jb @@ '*."term" = "NYC"'::jsquery)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on jb_vodka_idx (actual time=0.214..0.214 rows=285 loops=1)
             Index Cond: (jb @@ '*."term" = "NYC"'::jsquery)
 Execution time: 0.526 ms  (0.716 ms, GIN jsonb_path_value_ops)
```



CREATE INDEX … USING VODKA

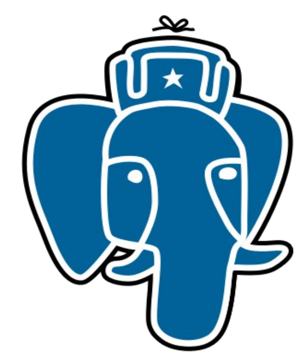
- CITUS data, text and numeric

```
set maintenance_work_mem = '1GB';

                              List of relations
 Schema |        Name          | Type  |  Owner   | Table |   Size    | Description
--------+----------------------+-------+----------+-------+-----------+-------------
 public | jr                   | table | postgres |       | 1573 MB   |  3023162 rows
 public | jr_value_path_idx    | index | postgres | jr    | 196 MB    |  79180.120
 public | jr_gin_idx           | index | postgres | jr    | 235 MB    | 111814.929
 public | jr_path_value_idx    | index | postgres | jr    | 196 MB    |  73369.713
 public | jr_path_idx          | index | postgres | jr    | 180 MB    |  48981.307
 public | jr_vodka_idx3        | index | postgres | jr    | 240 MB    | 155714.777
 public | jr_vodka_idx4        | index | postgres | jr    | 211 MB    | 169440.130 new spgist

(6 rows)
```



CREATE INDEX … USING VODKA

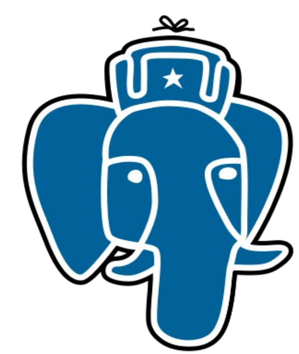
```
explain (analyze, costs off) select count(*) from jr where jr @@ ' similar_product_ids && ["B000089778"]';
                          QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate (actual time=0.200..0.200 rows=1 loops=1)
   -> Bitmap Heap Scan on jr (actual time=0.090..0.183 rows=185 loops=1)
        Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
        Heap Blocks: exact=107
        -> Bitmap Index Scan on jr_vodka_idx (actual time=0.077..0.077 rows=185 loops=1)
             Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.237 ms (0.394 ms, GIN jsonb_path_value_idx)
(7 rows)
```



CREATE INDEX … USING VODKA

- No statistics, no planning :(

```
select count(*) from jr where  jr @@ ' similar_product_ids && ["B000089778"]
 & product_sales_rank( $ > 10000 & $  < 20000)';
                                              QUERY PLAN
--------------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=127.471..127.471 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=127.416..127.461 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_vodka_idx (actual time=127.400..127.400 rows=45 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"]
& "product_sales_rank"($ > 10000 & $ < 20000))'::jsquery)
 Execution time: 130.051 ms
(7 rows)
```

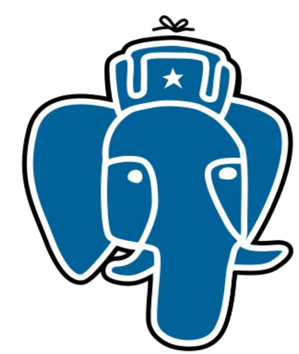


CREATE INDEX … USING VODKA

- No statistics, no planning :(

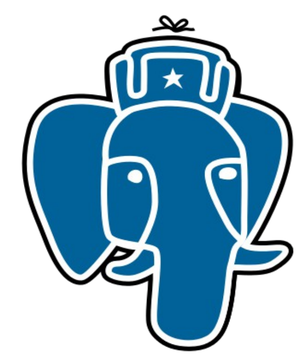
```
select count(*) from jr where jr @@ ' similar_product_ids && ["B000089778"]'
and  (jr->>'product_sales_rank')::int>10000 and (jr->>'product_sales_rank')::int<20000;
                                  QUERY PLAN
---------------------------------------------------------------------------------------------
 Aggregate (actual time=0.401..0.401 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.109..0.395 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_vodka_idx (actual time=0.079..0.079 rows=185 loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.431 ms  (7 ms, MongoDB)
(9 rows)
```

BIG Potential !



CREATE INDEX … USING VODKA

```
select count(*) from jr where jr @@ 'similar_product_ids && ["B000089778"] & review_rating($> 3 & $<5)';
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=98.313..98.314 rows=1 loops=1)
   -> Bitmap Heap Scan on jr (actual time=98.273..98.307 rows=32 loops=1)
        Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] & "review_rating"($ > 3 & $ < 5))'::jsquery)
        Heap Blocks: exact=16
        -> Bitmap Index Scan on jr_path_value_idx (actual time=98.254..98.254 rows=32 loops=1)
             Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] & "review_rating"($ > 3 & $ < 5))'::jsquery)
 Execution time: 99.873 ms

-----------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=1.521..1.521 rows=1 loops=1)
   -> Bitmap Heap Scan on jr (actual time=1.503..1.515 rows=32 loops=1)
        Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] & "review_rating"($ > 3 & $ < 5))'::jsquery)
        Heap Blocks: exact=16
        -> Bitmap Index Scan on jr_vodka_idx (actual time=1.498..1.498 rows=32 loops=1)
             Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] & "review_rating"($ > 3 & $ < 5))'::jsquery)
 Execution time: 1.550 ms  (FAST SCAN !)
```

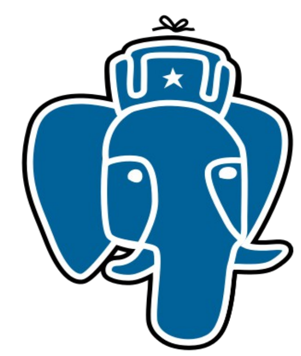


Need positional information for both GIN and VODKA

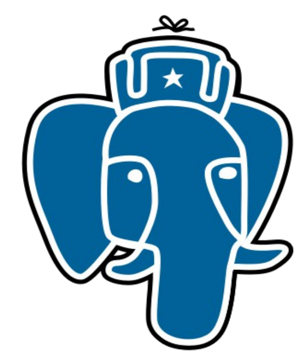
[{"f1": 23, "f2": 46}, {"f1": 42, "f2": 24}, {"f1": 98, "f2": 2}, {"f1": 62, "f2": 70},
{"f1": 66, "f2": 41}, {"f1": 32, "f2": 95}, {"f1": 11, "f2": 64}, {"f1": 20, "f2": 27},
{"f1": 45, "f2": 42}, {"f1": 14, "f2": 53}]

explain analyze select * from test where v @@ '#(f1 = 10 & f2 = 20)'; **- HUGE RECHECK!**

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on test  (cost=191.75..3812.68 rows=1000 width=32)
(actual time=14.576..35.039 rows=1005 loops=1)
   Recheck Cond: (v @@ '#("f1" = 10 & "f2" = 20)'::jsquery)
   Rows Removed by Index Recheck: 7998
   Heap Blocks: exact=8416
   -> Bitmap Index Scan on test_idx  (cost=0.00..191.50 rows=1000 width=0)
      (actual time=13.396..13.396 rows=9003 loops=1)
       Index Cond: (v @@ '#("f1" = 10 & "f2" = 20)'::jsquery)
 Execution time: 35.329 ms
```

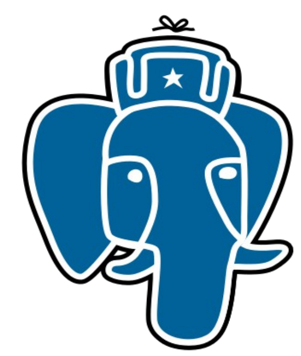


There are can be different flavors of Vodka

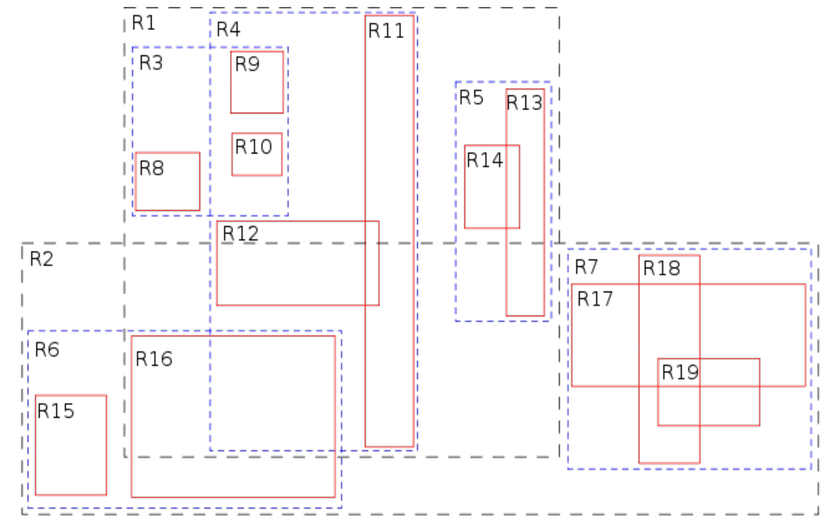


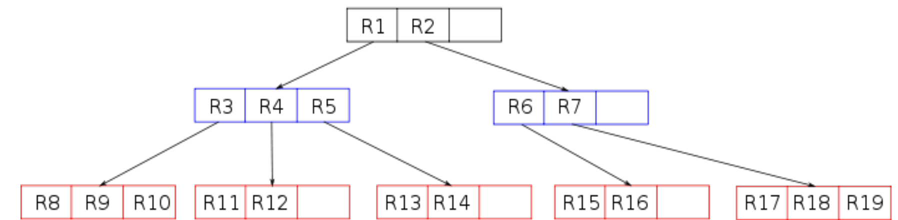
Spaghetti indexing ...

Find twirled spaghetti

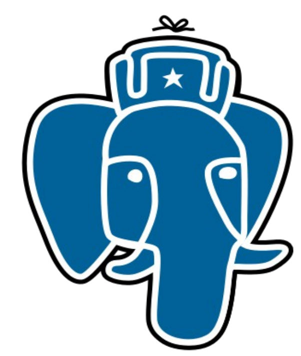


Spaghetti indexing ...

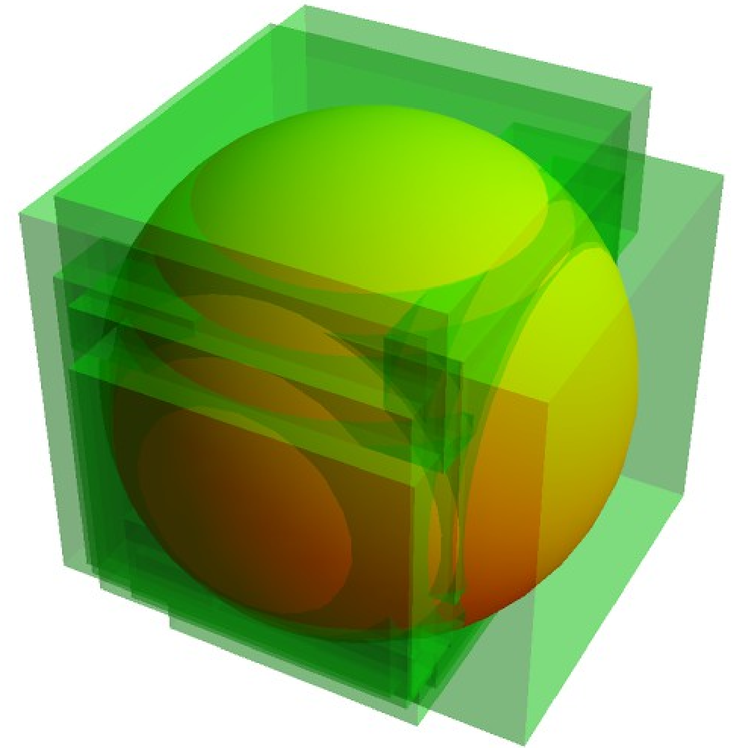




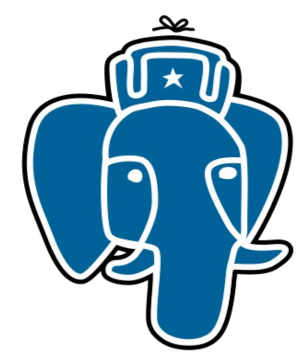
R-tree fails here – bounding box of each separate spaghetti is the same



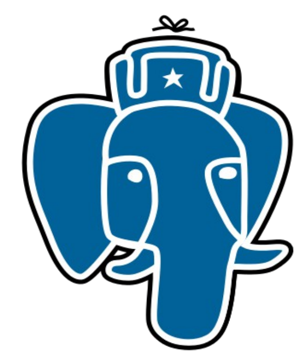
Spaghetti indexing ...



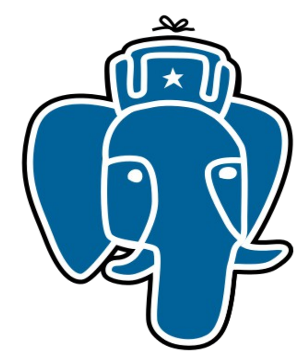
R-tree fails here – bounding box of each separate spaghetti is the same



Ottawa downtown: York and George streets



Idea: Use multiple boxes



Rtree Vodka

R1
R2
R3
R4
R-tree GiST entry tree

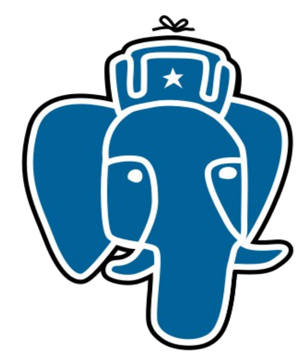
Posting lists/trees

iptr1	iptr2

iptr1	iptr2	iptr3	iptr4

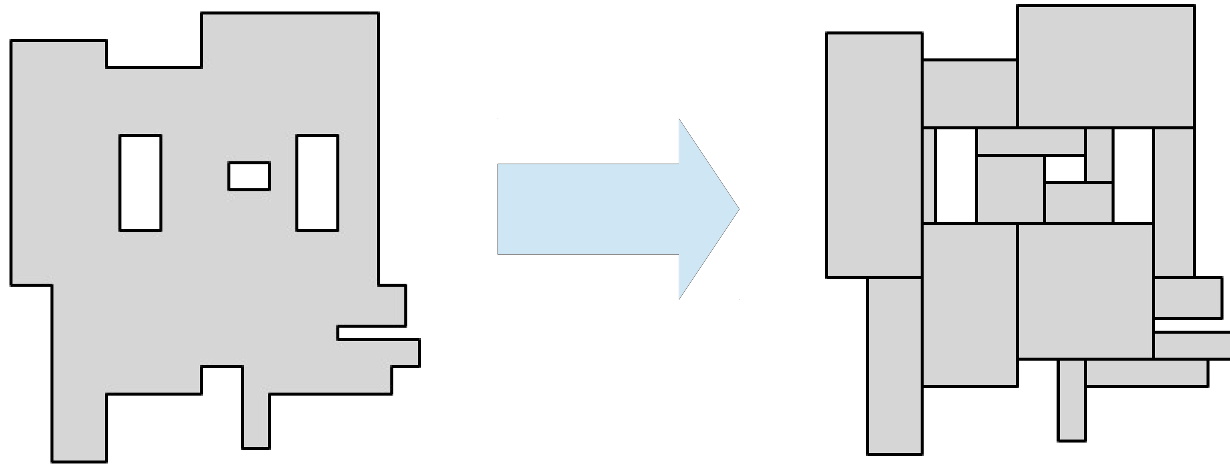
iptr1	iptr2	iptr3	iptr4	iptr5

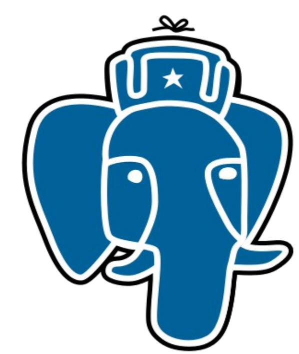
iptr1	iptr2	iptr3



Rtree Vodka

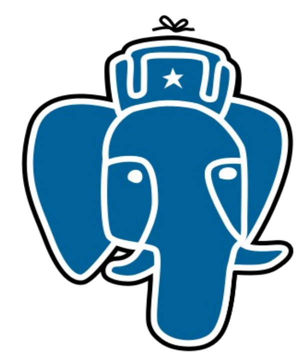
- R-tree based on GiST as Entry tree
- An algorithm for covering polygons with rectangles ?
- Need support — POSTGIS ?



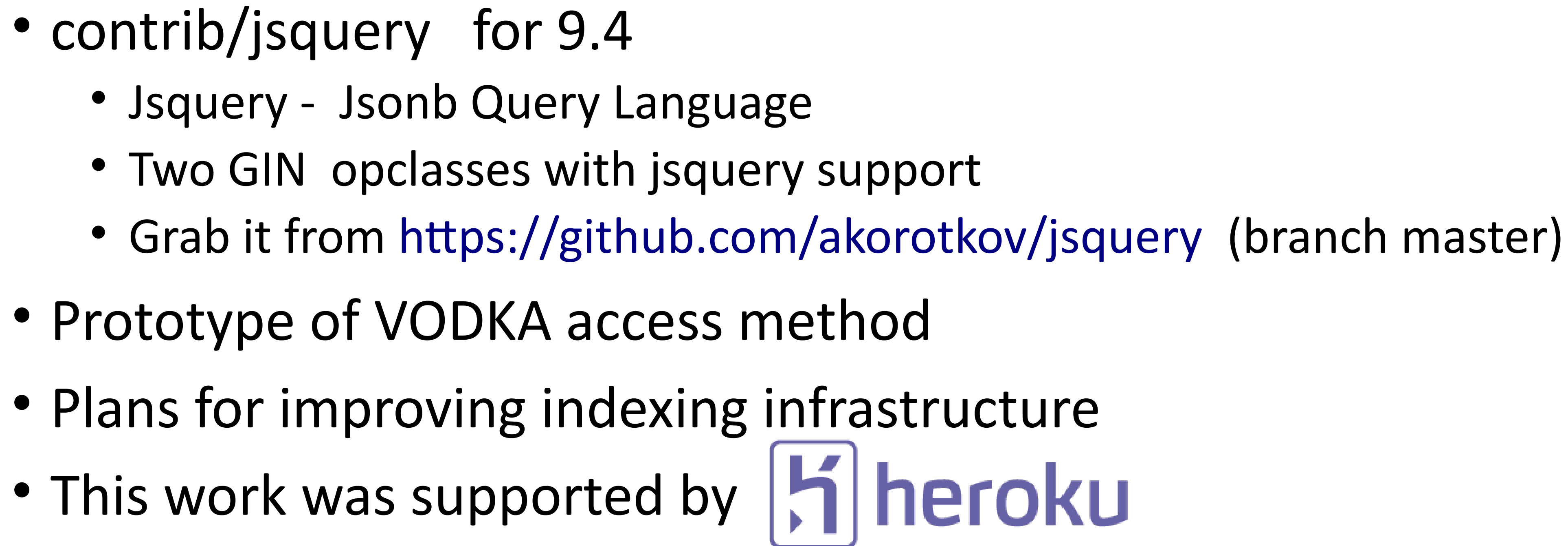


Vodka problems

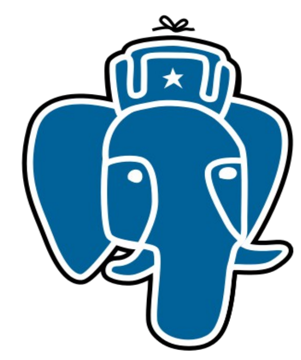
- How to update ItemPointer in entry tree? Insert new then vacuum is weird and expensive. amupdateiptr?
- Now it's hard to get OIDs of extension opclass or operator.
- Storing entry tree in separate file: is it correct? What infrastructure should handle it?
- TODO: replacing posting tree with arbitrary access method. Have to share multiple indexes in same file. How am interface can handle it? Pass root block number to am? How to vacuum?



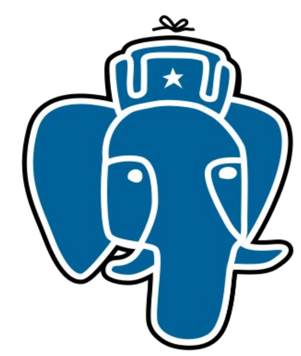
Summary



- contrib/jsquery for 9.4
  - Jsquery - Jsonb Query Language
  - Two GIN opclasses with jsquery support
  - Grab it from https://github.com/akorotkov/jsquery (branch master)
- Prototype of VODKA access method
- Plans for improving indexing infrastructure
- This work was supported by heroku



Meet us in Madrid with better VODKA !



VODKA Optimized Dendriform Keys Array