CREATE INDEX … USING VODKA.
VODKA CONNECTING INDEXES!

Олег Бартунов, ГАИШ МГУ
Александр Коротков, «Интеро-Софт»
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
Agenda

- Introduction into jsonb
- Jsonb querying
- Jsonb indexing
- VODKA prototype
- Future
Semi-structured data in PostgreSQL

- FTS — OpenFTS in 2000, in-core since 8.3
- hstore — as contrib/hstore since 8.3
- XML — in-core text data type since 8.2, contrib/xml2 since 8.3
- json — in-core text data type since 9.2, access functions since 9.3
- jsonb — in-core binary data type since 9.4
## Jsonb vs Json

<table>
<thead>
<tr>
<th>json</th>
<th>jsonb</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{&quot;c&quot;:0,  &quot;a&quot;:2,&quot;a&quot;:1}</code></td>
<td><code>{&quot;a&quot;: 1, &quot;c&quot;: 0}</code></td>
</tr>
</tbody>
</table>

(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 bookmarks from Delicious in json format (js)
  • The same bookmarks in jsonb format (jb)
  • The same bookmarks as text (tx)

```sql
=# \dt+

<table>
<thead>
<tr>
<th>List of relations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
</tr>
<tr>
<td>---------</td>
</tr>
<tr>
<td>public</td>
</tr>
<tr>
<td>public</td>
</tr>
<tr>
<td>public</td>
</tr>
</tbody>
</table>
Currently (9.4), 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 querying an array: simple case

Find bookmarks with tag «NYC»:

```sql
SELECT * 
FROM js 
WHERE js @@ '{"tags": [{"term": "NYC"}]}'
```
Jsonb querying an array: complex case

Find companies where CEO or CTO is called Neil.
One could write...

```
SELECT * FROM company
WHERE js @@ '{"relationships":[{"person":{"first_name":"Neil"}}}]'
  AND
  (js @@ '{"relationships":[{"title":"CTO"}]}' OR
   js @@ '{"relationships":[{"title":"CEO"}]}
```
Jsonb querying an array: complex case

Each «@>» is processed independently.

```
SELECT * FROM company
WHERE js @> '{"relationships":[{"person":{"first_name":"Neil"}}}]'
  AND
  (js @> '{"relationships":[{"title":"CTO"}]}') OR
  js @> '{"relationships":[{"title":"CEO"}]}');
```

Actually, this query searches for companies with some CEO or CTO and someone called Neil...
Jsonb querying an array: complex case

The correct version of the query is:

```
SELECT * FROM company
WHERE js @> '{"relationships":
    [{"title":"CEO", "person":{"first_name":"Neil"}}]}') OR
js @> '{"relationships":
    [{"title":"CTO", "person":{"first_name":"Neil"}}]}';
```

When constructing complex conditions over the same array element, query length can grow exponentially.
Jsonb querying an array: another approach

Using subselect and jsonb_array_elements:

```
SELECT * FROM company
WHERE EXISTS (  
    SELECT 1  
    FROM jsonb_array_elements(js -> 'relationships') t  
    WHERE t->>'title' IN ('CEO', 'CTO') AND  
      t->'person'->>'first_name' = 'Neil');
```
Jsonb querying an array: summary

Using «@>»

• Pro
  • Indexing support

• Cons
  • Checks only equality for scalars
  • Hard to explain complex logic

Using subselect and jsonb_array_elements

• Pro
  • Full power of SQL can be used to express condition over element

• Cons
  • No indexing support
  • Heavy syntax
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
Jsonb query language (Jsquery)

```
Expr ::= path value_expr
   | path HINT value_expr
   | NOT expr
   | NOT HINT value_expr
   | NOT value_expr
   | path '(' expr ')'
   | '(' expr ')' 
   | expr AND expr
   | expr OR expr

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

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

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

value_list ::= scalar_value
   | value_list ',' scalar_value

array ::= [' value_list ']

scalar_value ::= null
   | STRING
   | true
   | false
   | NUMERIC
   | OBJECT

[diagram]
```

---

```
value_expr ::= value_list

array ::= array

scalar_value ::= scalar_value
```

---

```
Expr ::= path value_expr
   | path HINT value_expr
   | NOT expr
   | NOT HINT value_expr
   | NOT value_expr
   | path '(' expr ')'
   | '(' expr ')' 
   | expr AND expr
   | expr OR expr

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

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

value_list ::= scalar_value
   | value_list ',' scalar_value

array ::= [' value_list ']

scalar_value ::= null
   | STRING
   | true
   | false
   | NUMERIC
   | OBJECT

[diagram]
```
Jsonb query language (Jsquery)

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

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

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

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

- Use "double quotes" for key!
  
  ```sql
  select 'a1."12222" < 111'::jsquery;
  ```

```
pattern ::= key
  | path '.' key_any
  | key
  | path '. not' key_any

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

  key_any ::= key
  | key not
```

Note: The path syntax is used to select or filter JSON objects. It supports various operators like `#`, `%`, `*`, `$`, and others, allowing for flexible querying of nested JSON data.
Jsonb query language (Jsquery)

- Scalar

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

- Test for key existence

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

- Array overlap

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

- Array contains

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

- Array contained

```sql
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

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

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

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

```json
{
    "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"
    ]
}
```
Jsonb query language (Jsquery)

• 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()
Each usage of «#», «*», «%» means separate element

- Find companies where CEO or CTO is called Neil.
  SELECT count(*) FROM company WHERE js @@ 'relationships.# (title in ("CEO", "CTO") AND person.first_name = "Neil")'::jsquery;
  count
  -------
  12

- Find companies with some CEO or CTO and someone called Neil
  SELECT count(*) FROM company WHERE js @@ 'relationships(#.title in ("CEO", "CTO") AND #.person.first_name = "Neil")'::jsquery;
  count
  -------
  69
Jsonb query language (Jsquery)

```
explain( analyze, buffers) select count(*) from jb where jb @> '{"tags":[{"term":"NYC"}]}':jsonb;
```

**QUERY PLAN**

```
Aggregate  (cost=191514.16..191514.16 rows=1 width=0) (actual time=0.006..1039.310 rows=285 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=891.553..891.553 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
Planning time: 0.074 ms
Execution time: 891.745 ms
```
Jsquery (indexes)

- GIN opclasses with jsquery support
  - jsonb_value_path_ops
    - use Bloom filtering for key matching
    - Good for key matching (wildcard support)
    - not good for range query
  - jsonb_path_value_ops
    - hash path (like jsonb_path_ops)
    - No wildcard support, no problem with ranges

List of relations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>jb</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>1374 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_value_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_gin_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>544 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_path_value_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>251 MB</td>
<td></td>
</tr>
</tbody>
</table>
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)
explain( analyze, costs off) select count(*) from jb where jb @@ '*.term = "NYC"';

QUERY PLAN

<table>
<thead>
<tr>
<th>QUERY PLAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregate (actual time=0.688..0.688 rows=1 loops=1)</td>
</tr>
<tr>
<td>-&gt; Bitmap Heap Scan on jb (actual time=0.145..0.660 rows=285 loops=1)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-&gt; Bitmap Index Scan on jb_value_path_idx (actual time=0.113..0.113 rows=285 loops=1)</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Execution time: 0.716 ms

(7 rows)
**Summary: PostgreSQL 9.4 vs Mongo 2.6.0**

- **Search for tag on delicious bookmarks**
  - `json`: 10 s seqscan
  - `jsonb`: 8.5 ms GIN jsonb_ops
  - `jsonb`: 0.7 ms GIN jsonb_path_ops
  - `jsquery`: 0.6 ms GIN jsonb_path_value_ops
  - `jsquery`: 0.7 ms GIN jsonb_value_path_ops
  - `mongo`: 1.0 ms btree index

- **Index size**
  - `jsonb_ops`: - 636 Mb
  - `jsonb_path_ops`: - 295 Mb
  - `jsonb_path_value_ops`: - 306 Mb
  - `jsonb_value_path_ops`: - 306 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
Citus dataset

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

```json
{
"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",
   "0910827312",
   "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"]
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
(7 rows)
```
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"
            ]
        ],
    },
}
Jsquery (indexes)

- Jsquery is opaque to planner, we could rewrite query and use planner

```sql
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** vs **7 ms** MongoDB!

(9 rows)
Jsquery now has built-in simple optimiser.

```sql
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!**
Since GIN opclasses can't expose something special to explain output, jsquery optimiser has its own explain functions:

- `text gin_debug_query_path_value(jsquery) — explain for jsonb_path_value_ops`
  ```sql
  # SELECT gin_debug_query_path_value('x = 1 AND (*.y = 1 OR y = 2'));
  gin_debug_query_path_value
  ---------------------------
  x = 1, entry 0           +
  ```

- `text gin_debug_query_value_path(jsquery) — explain for jsonb_value_path_ops`
  ```sql
  # SELECT gin_debug_query_value_path('x = 1 AND (*.y = 1 OR y = 2'));
  gin_debug_query_value_path
  ---------------------------
  AND               +
  x = 1, entry 0     +
  OR                +
  *.y = 1, entry 1   +
  y = 2, entry 2    +
  ```
Jsquery (optimizer) — NEW!

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 ($x = *$)
AND children can be put into recheck.

```sql
# SELECT gin_debug_query_path_value('x = 1 AND y > 0');
gin_debug_query_path_value
----------------------------
x = 1 , entry 0 +
```

While OR children can't. We can't handle false negatives.

```sql
# SELECT gin_debug_query_path_value('x = 1 OR y > 0');
gin_debug_query_path_value
----------------------------
OR +
  x = 1 , entry 0 +
y > 0 , entry 1 +
```
Can't do much with NOT, because hash is lossy. After NOT false positives turns into false negatives, which we can't handle.

```sql
# SELECT gin_debug_query_path_value('x = 1 AND (NOT y = 0)');
gin_debug_query_path_value
------------------------
x = 1, entry 0 +
```
Jsquery (optimizer) — NEW!

- Jsquery optimiser pushes non-selective operators to recheck

```
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
Jsquery (HINTING) — NEW!

- Jsquery now has HINTING (if you don't like optimiser):

  ```sql
  explain (analyze, costs off) select count(*) from jr where jr @@ 'product_sales_rank > 10000'
  ```

  ![Query Execution Plan](image)

  ```sql
  Aggregate (actual time=2507.410..2507.410 rows=1 loops=1)
  -> Bitmap Heap Scan on jr (actual time=1118.814..2352.286 rows=2373140 loops=1)
  Recheck Cond: (jr @@ "product_sales_rank" > 10000::jsquery)
  Heap Blocks: exact=201209
  -> Bitmap Index Scan on jr_path_value_idx (actual time=1052.483..1052.48 rows=2373140 loops=1)
  Index Cond: (jr @@ "product_sales_rank" > 10000::jsquery)
  Execution time: 2524.951 ms
  ```

- Better not to use index — HINT /*--noindex */

  ```sql
  explain (analyze, costs off) select count(*) from jr where jr @@ 'product_sales_rank /*-- noindex */ > 10000'
  ```

  ```sql
  Aggregate (actual time=1376.262..1376.262 rows=1 loops=1)
  -> Seq Scan on jr (actual time=0.013..1222.123 rows=2373140 loops=1)
  Filter: (jr @@ "product_sales_rank" /*-- noindex */ > 10000::jsquery)
  Rows Removed by Filter: 650022
  Execution time: 1376.284 ms
  ```
• If you know that inequality is selective then use HINT /* --index */

```sql
# 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=11.1 79..11.1 79 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)
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!
  • We will release it after PostgreSQL 9.4 release
  • Need real sample data and queries!
PostgreSQL 9.4+
- Open-source
- Relational database
- Strong support of json
Why PostgreSQL is better than MongoDB

• Effective binary storage for json (jsonb type)
• GIN indexes (compact and fast)
• Durability and reliability are proven by dozens of years
• High concurrency and performance
JsQuery limitations

• Variables are always on the left size
  \[ x = 1 \quad – \quad OK \]
  \[ 1 = x \quad – \quad Error! \]

• No calculations in query
  \[ x + y = 0 \quad – \quad Error! \]

• No extra datatypes and search operators
  \[ \text{point}(x,y) <@ '((0,0),(1,1),(2,1),(1,0))'::\text{polygon} \]
JsQuery limitations

Users want jsquery to be as rich as SQL...
JsQuery limitations

Users want jsquery to be as rich as SQL ...

... But we will discourage them ;(
JsQuery language goals

• Provide rich enough query language for jsonb in 9.4.
• Indexing support for 'jsonb @@ jsquery':
  • Two GIN opclasses are in jsquery itself
  • VODKA opclasses was tested on jsquery

It's NOT intended to be solution for jsonb querying in long term!
What JsQuery is NOT?

It's **not** designed to be another **extendable, full weight**:

- Parser
- Executor
- Optimizer

It's NOT SQL inside SQL.
## Jsonb querying an array: summary –! no statistics!

<table>
<thead>
<tr>
<th>Using «@&gt;»</th>
<th>Using subselect and jsonb_array_elements</th>
<th>JsQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro</strong></td>
<td><strong>Pro</strong></td>
<td><strong>Pro</strong></td>
</tr>
<tr>
<td>Indexing support</td>
<td>SQL-rich</td>
<td>Indexing support</td>
</tr>
<tr>
<td>Cons</td>
<td>Cons</td>
<td>Cons</td>
</tr>
<tr>
<td>Checks only equality for scalars</td>
<td>No indexing support</td>
<td>Rich enough for typical applications</td>
</tr>
<tr>
<td>Hard to explain complex logic</td>
<td>Heavy syntax</td>
<td>Not extendable</td>
</tr>
</tbody>
</table>

Still looking for a better solution!
Jsonb query: future

Users want jsonb query language to be as rich as SQL. How to satisfy them?..
Jsonb query: future

Users want jsonb query language to be as rich as SQL. How to satisfy them?

Bring all required features to SQL-level!
Jsonb query: future

Functional equivalents:

• SELECT * FROM company WHERE EXISTS (SELECT 1 FROM jsonb_array_elements(js->'relationships') t WHERE t->>'title' IN ('CEO', 'CTO') AND t->'person'->>'first_name' = 'Neil');

• SELECT count(*) FROM company WHERE js @@ 'relationships(#.title in ("CEO", "CTO") AND #.person.first_name = "Neil")'::jsquery;

• SELECT * FROM company WHERE ANYELEMENT OF js-> 'relationships' AS t ( t->>'title' IN ('CEO', 'CTO') AND t->'person'->>'first_name' = 'Neil');
Jsonb query: ANYELEMENT

Possible implementation steps:

• Implement ANYELEMENT as syntactic sugar and only for arrays.
• Support for various data types (extendable?)
• Handle ANYELEMENT as expression not subselect (problem with alias).
• Indexing support over ANYELEMENT expressions.
Another idea about ANYELEMENT

Functional equivalents:

- SELECT t
  FROM company,
    LATERAL (SELECT t FROM 
      jsonb_array_elements(js->'relationships') t) el;

- SELECT t
  FROM company,
    ANYELEMENT OF js->'relationships' AS t;
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

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>jb</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>1374 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_uniq_paths</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>912 MB</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>jb_uniq_paths_btree_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb_uniq_paths</td>
<td>885 MB</td>
<td>text_pattern_ops</td>
</tr>
<tr>
<td>public</td>
<td>jb_uniq_paths_spgist_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb_uniq_paths</td>
<td>598 MB</td>
<td>now much less !</td>
</tr>
</tbody>
</table>
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
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 problems with jsonb

• Have to store hash or bloom of path (storing full paths would lead to very long keys)
• But we need to use complex conditions over keys
• We want to use different complex tree types for same jsonb dataset (B-tree for scalars, R-tree for geometry, RD-tree for sets etc.)
GIN index structure for jsonb

```json
{
    "product_group": "Book",
    "product_sales_rank": 15000
},
{
    "product_group": "Music",
    "product_sales_rank": 25000
}
```
Vodka index structure for jsonb

{
  "product_group": "Book",
  "product_sales_rank": 15000
},
{
  "product_group": "Music",
  "product_sales_rank": 25000
}
CREATE INDEX ... USING VODKA

• Delicious bookmarks, mostly text data

```
set maintenance_work_mem = '1GB';
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>jb</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>1374 MB</td>
<td>1252973 rows</td>
</tr>
<tr>
<td>public</td>
<td>jb_value_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td>98769.096</td>
</tr>
<tr>
<td>public</td>
<td>jb_gin_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>544 MB</td>
<td>129860.859</td>
</tr>
<tr>
<td>public</td>
<td>jb_path_value_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td>100560.313</td>
</tr>
<tr>
<td>public</td>
<td>jb_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>251 MB</td>
<td>68880.320</td>
</tr>
<tr>
<td>public</td>
<td>jb_vodka_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>409 MB</td>
<td>185362.865</td>
</tr>
<tr>
<td>public</td>
<td>jb_vodka_idx5</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>325 MB</td>
<td>174627.234 new spgist</td>
</tr>
</tbody>
</table>

(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';
```

| 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)
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')
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';

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>jb</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>1374 MB</td>
<td>1252973 rows</td>
</tr>
<tr>
<td>public</td>
<td>jb_value_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td>98769.096</td>
</tr>
<tr>
<td>public</td>
<td>jb_gin_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>544 MB</td>
<td>129860.859</td>
</tr>
<tr>
<td>public</td>
<td>jb_path_value_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>306 MB</td>
<td>100560.313</td>
</tr>
<tr>
<td>public</td>
<td>jb_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>251 MB</td>
<td>68880.320</td>
</tr>
<tr>
<td>public</td>
<td>jb_vodka_idx</td>
<td>index</td>
<td>postgres</td>
<td>jb</td>
<td>409 MB</td>
<td>185362.865</td>
</tr>
</tbody>
</table>

(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';
```

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>jr</td>
<td>table</td>
<td>postgres</td>
<td></td>
<td>1573 MB</td>
<td>3023162 rows</td>
</tr>
<tr>
<td>public</td>
<td>jr_value_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jr</td>
<td>196 MB</td>
<td>79180.120</td>
</tr>
<tr>
<td>public</td>
<td>jr_gin_idx</td>
<td>index</td>
<td>postgres</td>
<td>jr</td>
<td>235 MB</td>
<td>111814.929</td>
</tr>
<tr>
<td>public</td>
<td>jr_path_value_idx</td>
<td>index</td>
<td>postgres</td>
<td>jr</td>
<td>196 MB</td>
<td>73369.713</td>
</tr>
<tr>
<td>public</td>
<td>jr_path_idx</td>
<td>index</td>
<td>postgres</td>
<td>jr</td>
<td>180 MB</td>
<td>48981.307</td>
</tr>
<tr>
<td>public</td>
<td>jr_vodka_idx3</td>
<td>index</td>
<td>postgres</td>
<td>jr</td>
<td>240 MB</td>
<td>155714.777</td>
</tr>
</tbody>
</table>

(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

```sql
# 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
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
Rtree Vodka

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

- Posting list/tree is just a way of effective duplicate storage
- Entry tree can consist of multiple levels of different access methods
- VODKA is a way to combine different access method in single index: VODKA CONNECTING INDEXES
Summary

• jsonb in core in 9.4, jsquery extension for 9.4
• jsonb querying problems are identified
• Proposal for bringing jsonb querying at SQL-level
• VODKA as a new level of access method extendability (have a prototype and new concept)
• New VODKA concept
• Extendable AMs + Generalized WAL prototype (ability to release VODKA as an extension)
We invite to PGConf.RU in Moscow, February 2015!