Rethinking JSONB

June, 2015, Ottawa, Canada

Alexander Korotkov, Oleg Bartunov, Teodor Sigaev

Postgres Professional

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

- Querying problems
  - Jsonb
  - Arrays
  - Hstore
  - Proposal
- Compression of jsonb
  - Idea 1
  - Idea 2

Jsonb (Apr, 2014)

- Documentation
  - JSON Types, JSON Functions and Operators
- There are many functionality left in nested hstore
  - Can be an extension
- Need query language for jsonb
  - <,>,&& … operators for values
    a.b.c.d && [1,2,10]
  - Structural queries on paths
    *.d && [1,2,10]
  - Indexes !

Builtin 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 querying an array: simple case

Find bookmarks with tag «NYC»:

```
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 is so.

```
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 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)

- # - 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)

```
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
         ........

key_any      ::= key
           | NOT
```

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

array    ::= '[' value_list ']'

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

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
```

IS BOOLEAN

IS NUMERIC

IS ARRAY

IS OBJECT

IS STRING

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

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()
```

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
  - SQL-rich
- Cons
  - No indexing support
  - Heavy syntax

JsQuery

- Pro
  - Indexing support
  - Rich enough for typical applications
- Cons
  - Not extendable

Still looking for a better solution!

Querying problem isn't new!

We already have similar problems with:

- Arrays
- Hstore

How to search arrays by their elements?

```
# CREATE TABLE t AS (SELECT
array_agg((random()*1000)::int) AS a FROM
generate_series(1,1000000) i GROUP BY (i-1)/10);
# CREATE INDEX t_idx ON t USING gin(a);

Simple query
# SELECT * FROM t WHERE 10 = ANY(a); -- ANY(a) = 10 ?
 Seq Scan on t
   Filter: (10 = ANY (a))
```

How to search arrays by their elements?

```
Search for arrays overlapping {10,20}.
# SELECT * FROM t WHERE 10 = ANY(a) OR 20 = ANY(a);
 Seq Scan on t
   Filter: ((10 = ANY (a)) OR (20 = ANY (a)))

Search for arrays containing {10,20}.
# SELECT * FROM t WHERE 10 = ANY(a) AND 20 = ANY(a);
 Seq Scan on t
   Filter: ((10 = ANY (a)) AND (20 = ANY (a)))
```

No index - hard luck :(

How to make it use index?

```
Overlaps «&&» operator.
# SELECT * FROM t WHERE a && '{10,20}'::int[];
 Bitmap Heap Scan on t
   Recheck Cond: (a && '{10,20}'::integer[])
   ->  Bitmap Index Scan on t_idx
         Index Cond: (a && '{10,20}'::integer[])

Contains «@>» operator.
# SELECT * FROM t WHERE a @> '{10,20}'::int[];
 Bitmap Heap Scan on t
   Recheck Cond: (a @> '{10,20}'::integer[])
   ->  Bitmap Index Scan on t_idx
         Index Cond: (a @> '{10,20}'::integer[])
```

Why do we need search operators for arrays?

Currently PostgreSQL planner can use index for:

- WHERE col op value
- ORDER BY col
- ORDER BY col op value (KNN)

We had to introduce array operators to use index.

What can't be expressed by array operators?

Search for array elements greater than 10.

```
SELECT * FROM t WHERE 10 < ANY(a);
```

Search for array elements between 10 and 20.

```
SELECT * FROM t WHERE EXISTS (
    SELECT *
    FROM unnest(a) e
    WHERE e BETWEEN 10 AND 20);
```

Why GIN isn't used?

- Planner limitations: every search clause must be expressed as a single operator, no complex expressions!
- Current GIN implementation has no support for such queries
  - **DOABLE!**

Array indexing support

Expression	gin (i)	gist (i)
i && '{1,2}', i@> '{1,2}'	+	+
i @@ '1&(2|3)'	+	+
1 = ANY(i)	–	–
1 < ANY(i)	–	–
Subselects	–	–

Array indexing support desired: add support of complex expressions

Expression	gin (i)	gist (i)
i && '{1,2}', i@> '{1,2}'	+	+
i @@ '1&(2|3)'	+	+
1 = ANY(i)	+	+
1 < ANY(i)	+	+
Subselects	+	+

How to search hstore?

```
# CREATE TABLE t AS (SELECT hstore(array_agg('k'||
(random()*10)::int::text), array_agg('v'||
(random()*100)::int::text)) h FROM
generate_series(1,1000000) g GROUP BY (g-1)/10);

# CREATE INDEX t_idx ON t USING gin(h);

# SELECT * FROM t WHERE h->'k0' = 'v0';
 Seq Scan on t
   Filter: ((h -> 'k0'::text) = 'v0'::text)
```

How to search hstore?

```
# SELECT * FROM t WHERE h @> '"k0"=>"v0"';
 Bitmap Heap Scan on t
   Recheck Cond: (h @> '"k0"=>"v0"'::hstore)
   ->  Bitmap Index Scan on t_idx
         Index Cond: (h @> '"k0"=>"v0"'::hstore)
```

@> operator can be used for index on hstore

How to search hstore?

What about btree indexes?

```
# DROP INDEX t_idx;
# CREATE INDEX t_idx ON t ((h->'k0'));

# SELECT * FROM t WHERE h @> '"k0"=>"v0"';
 Seq Scan on t
   Filter: (h @> '"k0"=>"v0"'::hstore)
```

Indexed expression should exactly match.
It should be in the form: h->'k0' opr value

How to search hstore?

```
# SELECT * FROM t WHERE h->'k0' = 'v0';
 Index Scan using t_idx on t
   Index Cond: ((h -> 'k0'::text) = 'v0'::text)

# SELECT * FROM t WHERE h->'k0' < 'v0';
 Index Scan using t_idx on t
   Index Cond: ((h -> 'k0'::text) < 'v0'::text)
```

Index is used because expressions exactly match.
You also use <, > etc which can't be expressed by @>.

Hstore index support

Expression	gin (h)	gist (h)	btree ((h->'key'))
h @> '"key" => "value"'	+	+	–
h -> 'key' = 'value'	–	–	+
h -> 'key' > 'value'	–	–	+

SQL becomes not so declarative: you specify what index query could use by the form of expression.

Hstore index support desired: add support of complex expressions

Expression	gin (h)	gist (h)	btree ((h->'key'))
h @> '"key" => "value"'	+	+	?
h -> 'key' = 'value'	+	+	+
h -> 'key' > 'value'	+	+	+

SQL becomes not so declarative: you specify what index query could use by the form of expression.

Jsonb indexing: cumulative problems of arrays and hstore

Expression	using gin			using btree ((js->'key'))
	default	path	jsquery	
js @> '{"key": ["value"]}'	+	+	–	–
js->'key' = 'value'	–	–	–	+
js->'key' > 'value'	–	–	–	+
js @@ 'key.# = "value""'	–	–	+	–
Subselects	–	–	–	–

Jsonb querying

@> operator

- Pro
  - Indexing support
- Cons
  - Checks only equality for scalars
  - Hard to explain complex logic

Using subselects

- Pro
  - Support all SQL features
- Cons
  - **No indexing support**
  - **Heavy syntax**

JsQuery

- Pro
  - Rich enough for typical applications
  - Indexing support
- Cons
  - Not extendable

It would be nice to workout these two!

Better syntax: «anyelement» feature

```
{ ANY | EACH } { ELEMENT | KEY | VALUE | VALUE
ANYWHERE } OF container AS alias SATISFIES
(expression)
```

https://github.com/postgrespro/postgres/tree/anyelement

Examples

- There is element divisible by 3 in array

```
SELECT * FROM array_test WHERE ANY ELEMENT OF a
AS e SATISFIES (e % 3 = 0);
```

- Each element of array is between 1 and 10

```
SELECT * FROM array_test WHERE EACH ELEMENT OF
a AS e SATISFIES (e BETWEEN 1 AND 10);
```

Examples

- All the scalars in jsonb are numbers

```
SELECT * FROM jsonb_test WHERE EACH VALUE
ANYWHERE OF j AS e SATISFIES (jsonb_typeof(e) =
'number');
```

- There is at least one object in jsonb array

```
SELECT * FROM jsonb_test WHERE ANY ELEMENT OF j
AS e SATISFIES (jsonb_typeof(e) = 'object');
```

Examples

- Find companies where CEO or CTO is called Neil.

```
SELECT * FROM companies
WHERE ANY ELEMENT OF c->'department' AS d
SATISFIES (
     ANY ELEMENT OF d->'staff' AS s SATISFIES (
          s->>'name' = 'Neil' AND
          s->>'post' IN ('CEO', 'CTO')));
```

Examples

- Find companies where exists department with all salaries greater than 1000.

```
SELECT * FROM companies WHERE ANY ELEMENT OF c
->'department' AS d SATISFIES (
     EACH ELEMENT OF d->'staff' AS s SATISFIES (
          (s->>'salary')::numeric > 1000));
```

Just a syntaxic sugar

```
CREATE VIEW v AS (SELECT * FROM (SELECT '{1,2,3}'::int[] a) t WHERE
ANY ELEMENT OF t.a AS e SATISFIES (e >= 1));

View definition:
 SELECT t.a
   FROM ( SELECT '{1,2,3}'::integer[] AS a) t
  WHERE ( SELECT
                CASE
                    WHEN count(*) = 0 AND t.a IS NOT NULL THEN false
                    ELSE bool_or_not_null(e.e >= 1)
                END AS bool_or_not_null
           FROM unnest_element(t.a, false) e(e));
```

How could we extend index usage?

- Opclass specify some grammar of index accelerated expressions.
- Now this grammar is trivial: limited set of «col opr value» expressiona.
- Need some better grammar to support wider set of expressions.

Use a-grammar for opclasses

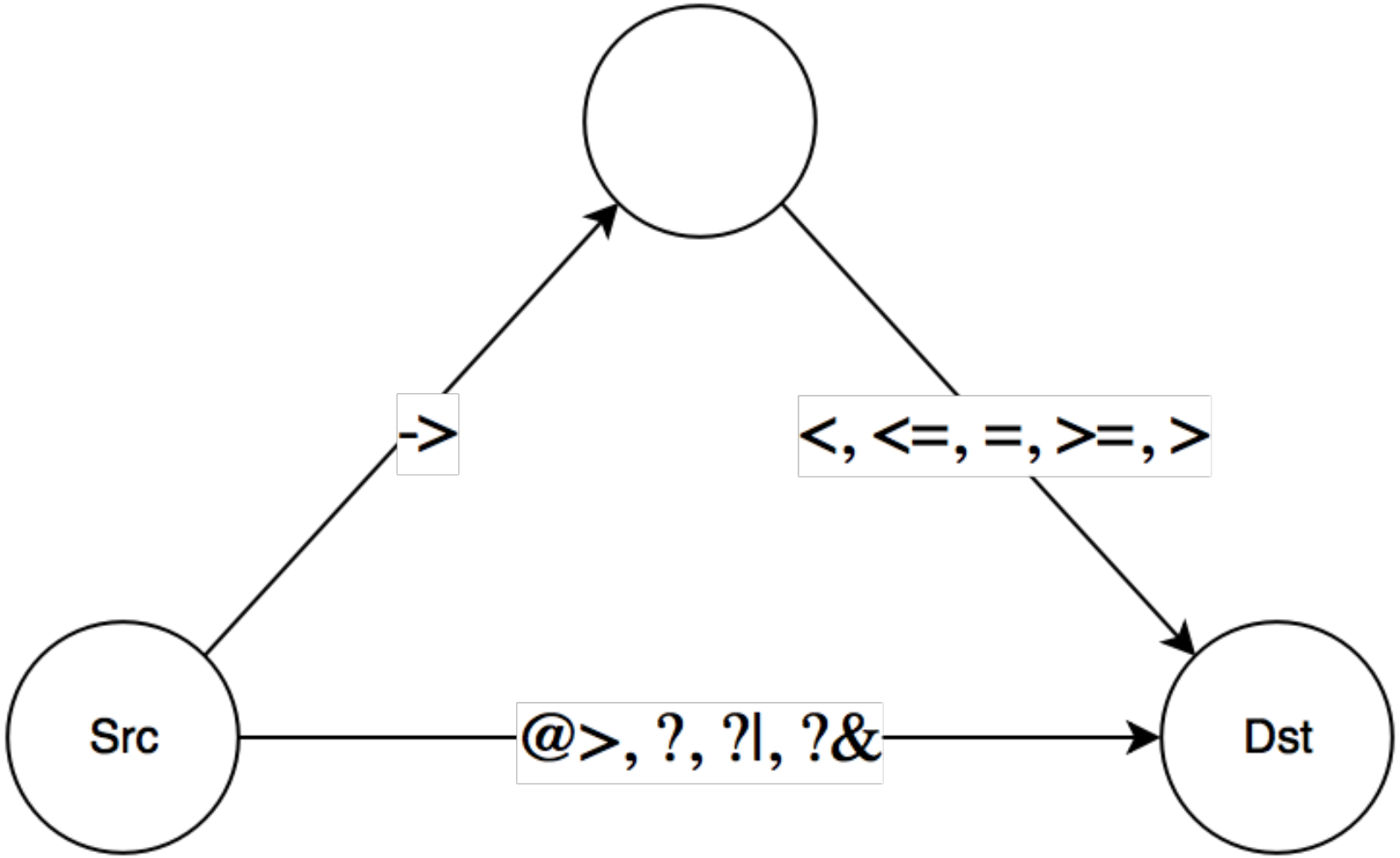
Automaton could be specified in opclass.

```
# \d pg_amop
       Table "pg_catalog.pg_amop"
     Column       |   Type   | Modifiers
------------------+----------+-----------
 amopfamily       | oid      | not null
 amoplefttype     | oid      | not null
 amoprighttype    | oid      | not null
 amopstrategy     | smallint | not null
 amoppurpose      | "char"   | not null
 amopopr          | oid      | not null
 amopmethod       | oid      | not null
 amopsortfamily   | oid      | not null
 amopsource       | smallint | not null
 amopdestination  | smallint | not null
```

New opclass format: hstore

```
CREATE OPERATOR CLASS gin_hstore_ops DEFAULT FOR TYPE hstore USING gin AS
OPERATOR        1       ->(hstore, text) 1 2,
OPERATOR        2       <(text, text) 2 0,
OPERATOR        3       <=(text, text) 2 0,
OPERATOR        4       =(text, text) 2 0,
OPERATOR        5       >=(text, text) 2 0,
OPERATOR        6       >(text, text) 2 0,
OPERATOR        7       @>,
OPERATOR        9       ?(hstore,text),
OPERATOR        10      ?|(hstore,text[]),
OPERATOR        11      ?&(hstore,text[]),
FUNCTION        1       bttextcmp(text,text),
FUNCTION        2       gin_extract_hstore(internal, internal),
FUNCTION        3       gin_extract_hstore_query(internal, internal, int2, interna
FUNCTION        4       gin_consistent_hstore(internal, int2, internal, int4, inte
STORAGE         text;
```

New opclass format: hstore



There is a lot of work!

- Changes in system catalog.
- Changes in planner to support complex expressions for indexes.
- Changes in access method interface. Array of ScanKeys isn't expressive enough! Need something called «ScanTrees».
- Changes in operator classes functions interface. «ScanTrees» should be passed into opclasses.

Postgres PROFESSIONAL

Thanks for support

WORLD OF TANKS

WARGAMING.NET

LET'S BATTLE

PROFESSIONAL Postgres

JSONB format compression

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"
  ]
}
```

Citus dataset: storage in jsonb

Heap size: 1588 MB
PK size: 65 MB
GIN index on product ids: 89 MB

```
                  Table "public.customer_reviews_jsonb"
 Column |  Type   |                        Modifiers
--------+---------+-----------------------------------------------------
 id     | integer | not null default
        |         | nextval('customer_reviews_jsonb_id_seq'::regclass)
 jr     | jsonb   |
Indexes:
    "customer_reviews_jsonb_pkey" PRIMARY KEY, btree (id)
    "customer_reviews_jsonb_similar_product_ids_idx" gin
        ((jr -> 'similar_product_ids'::text))
```

Citus dataset: normalized storage

Heap size: 434 MB (main table) + 598 MB (similar products) = 1032 MB

PK size: 65 MB (main table) + 304 MB (similar products) = 369 MB

Index on similar product id: 426 MB

```
    Table "public.customer_reviews_flat"
       Column        | Type    | Modifiers
---------------------+---------+-----------
 customer_id         | text    | not null
 review_date         | date    | not null
...
 product_subcategory | text    |
 id                  | integer | not null...
Indexes:
    "customer_reviews_flat_pkey"
        PRIMARY KEY, btree (id)
```

```
Table "public.customer_reviews_similar_product"
       Column        |  Type   | Modifiers
---------------------+---------+-----------
 product_id          | integer | not null
 similar_product_id  | bpchar  | not null
Indexes:
    "similar_product_product_id_idx"
        btree (product_id)
    "similar_product_similar_product_id_idx"
        btree (similar_product_id COLLATE "C")
```

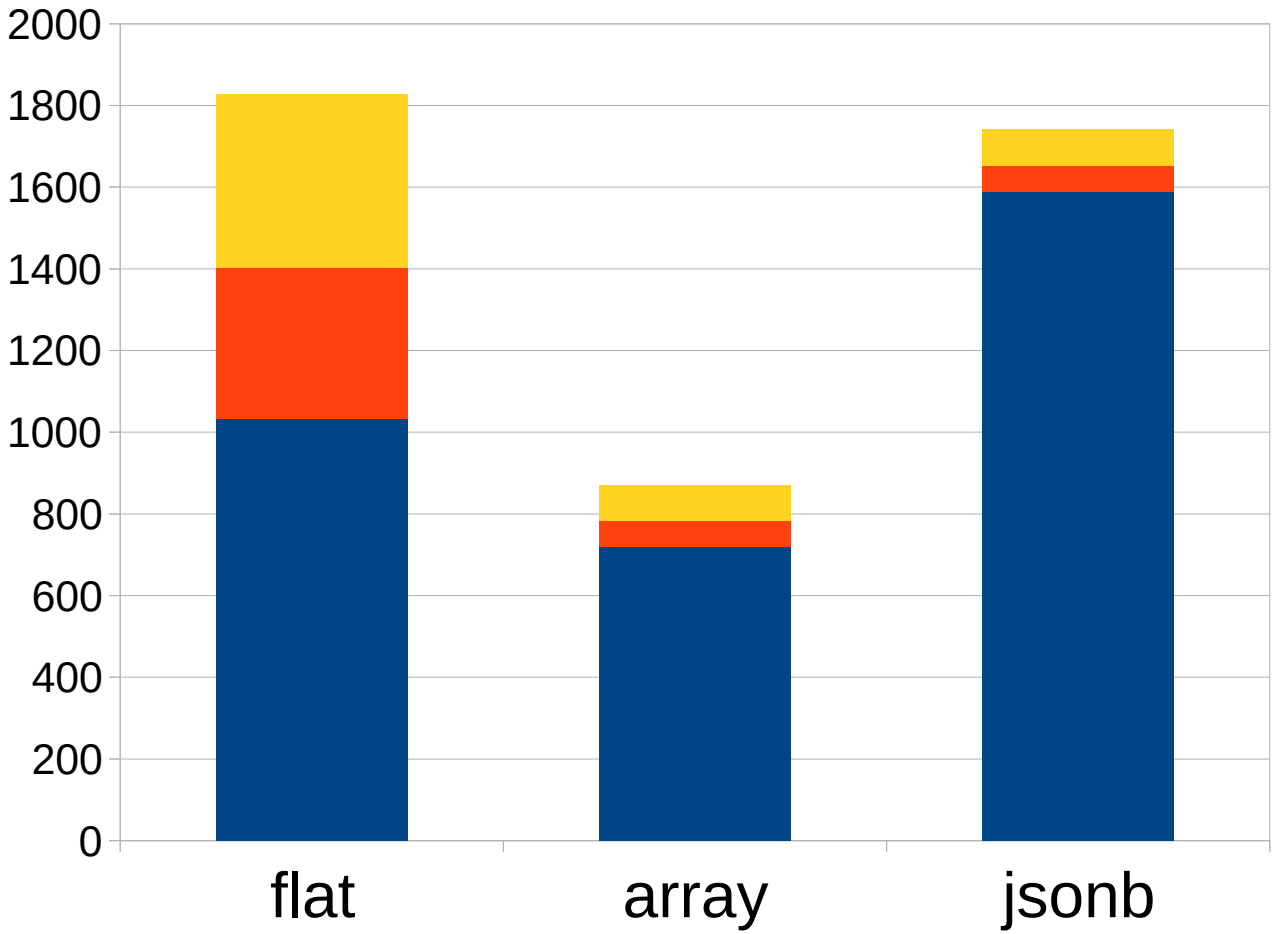
14 168 514 rows

Citus dataset: storage using array

Heap size: 719 MB
PK size: 65 MB
GIN index on product ids: 87 MB

```
         Table "public.customer_reviews_array"
        Column         |      Type        |  Modifiers
-----------------------+------------------+-------------
 id                    | integer          | not null …
 customer_id           | text             | not null
 review_date           | date             | not null
...
 similar_product_ids   | character(10)[] |
Indexes:
    "customer_reviews_array_pkey" PRIMARY KEY, btree (id)
    "customer_reviews_array_similar_product_ids_idx gin
                      (similar_product_ids COLLATE "C")
```

Citus dataset: storage size



Jsonb vs array overhead is huge!

Binary format of jsonb

```
{"product_id": "0613225783", "review_votes": 0, "product_group": "Book",
"product_title": "Walk in the Woods", "review_rating": 3, "product_sales_rank":
482809, "review_helpful_votes": 0}
```

```
07 00 00 20 0a 00 00 80  0c 00 00 00 0d 00 00 00  |... ............| Headers
0d 00 00 00 0d 00 00 00  12 00 00 00 14 00 00 00  |................|
0a 00 00 00 09 00 00 10  04 00 00 00 11 00 00 00  |................|
09 00 00 10 0a 00 00 10  08 00 00 10 70 72 6f 64  |............prod|
75 63 74 5f 69 64 72 65  76 69 65 77 5f 76 6f 74  |uct_idreview_vot| Key names
65 73 70 72 6f 64 75 63  74 5f 67 72 6f 75 70 70  |esproduct_groupp|
72 6f 64 75 63 74 5f 74  69 74 6c 65 72 65 76 69  |roduct_titlerevi|
65 77 5f 72 61 74 69 6e  67 70 72 6f 64 75 63 74  |ew_ratingproduct|
5f 73 61 6c 65 73 5f 72  61 6e 6b 72 65 76 69 65  |_sales_rankrevie|
77 5f 68 65 6c 70 66 75  6c 5f 76 6f 74 65 73 30  |w_helpful_votes0|
36 31 33 32 32 35 37 38  33 00 00 00 18 00 00 00  |613225783.......| Data
00 80 42 6f 6f 6b 57 61  6c 6b 20 69 6e 20 74 68  |..BookWalk in th|
65 20 57 6f 6f 64 73 00  20 00 00 00 00 80 03 00  |e Woods. .......|
28 00 00 00 01 80 30 00  f9 0a 00 00 18 00 00 00  |(.....0.........|
00 80                                             |..              |
```

Jsonb compression idea 1

- Maintain dictionary of keys
- Compress headers using varbyte encoding
- Compress small numbers using varbyte encoding

Cons:

- Maintain dictionary of schemas

Jsonbc extension: dictionary

```
CREATE TABLE jsonbc_dict
(
    id serial PRIMARY KEY,
    name text NOT NULL,
    UNIQUE (name)
);

extern int32 getIdByName(KeyName name);
extern KeyName getNameById(int32 id);
```

Binary format of jsonbc

```
{"product_id": "0613225783", "review_votes": 0, "product_group": "Book",
"product_title": "Walk in the Woods", "review_rating": 3, "product_sales_rank":
482809, "review_helpful_votes": 0}
```

```
3d 01 51 04 0b 01 21 01  89 01 01 0b 02 1b 03 0b  |=.Q...!.........| Headers
30 36 31 33 32 32 35 37  38 33 00 42 6f 6f 6b 57  |0613225783.BookW| Data
61 6c 6b 20 69 6e 20 74  68 65 20 57 6f 6f 64 73  |alk in the Woods|
06 f2 f7 3a 00                                    |...:.           |
```

Jsonbc extension

Customers reviews different format storage comparison

```
customer_reviews_jsonb          307 MB
customer_reviews_jsonbc         123 MB
customer_reviews_array          139 MB
```

Less than array because of numerics compression!

Jsonb compression idea 2

- Extract everything except raw data into schema
- Maintain dictionary of schemas
- Store only raw data and schema reference

Cons:

- Maintain dictionary of schemas
- There could be as many schemas as documents

Idea 2 estimation

Customers reviews different format storage comparison

```
customer_reviews_jsonb          307 MB
customer_reviews_jsonbc         123 MB
customer_reviews_jsonb2         106 MB (Estimation!)
customer_reviews_array          139 MB
```

16%
Does it worth the effort?

Postgres PROFESSIONAL

Thanks for support

WORLD OF TANKS

WARGAMING.NET

LET'S BATTLE