JSON in Postgres - The present and Future

Oleg Bartunov
Postgres Professional
Moscow University

NoSQL Postgres briefly

• 2003 — hstore (sparse columns, schema-less)
• 2006 — hstore as demo of GIN indexing, 8.2 release
• 2012 (sep) — JSON in 9.2 (verify and store)
• 2012 (dec) — nested hstore proposal
• 2013 — PGCon: nested hstore
• 2013 — PGCon.eu: binary storage for nested data
• 2013 (nov) — nested hstore & jsonb (better/binary)
• 2014 (feb-mar) — forget nested hstore for jsonb
• Mar 23, 2014 — jsonb committed for 9.4
• Autumn, 2018 — SQL/JSON for 10.X or 11?
JSONB - 2014
- Binary storage
- Nesting objects & arrays
- Indexing

HSTORE - 2003
- Perl-like hash storage
- No nesting
- Indexing
Google trends: `jsonb` vs `hstore`
Two JSON data types !!!
SELECT j::json AS json, j::jsonb AS jsonb FROM (SELECT '{"cc":0, "aa": 2, "aa":1,"b":1}' AS j) AS foo;

<table>
<thead>
<tr>
<th>json</th>
<th>jsonb</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;cc&quot;:0, &quot;aa&quot;: 2, &quot;aa&quot;:1,&quot;b&quot;:1}</td>
<td>{&quot;b&quot;: 1, &quot;aa&quot;: 1, &quot;cc&quot;: 0}</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 by (length, key)
- jsonb has a binary storage: no need to parse, has index support
Very detailed talk about JSON[B]

The One-Slide Oversimplification.

- Use relational data for the basic set of attributes.
- Use either array fields or jsonb for extended attributes.
- Use file-system storage for really big stuff.
- Always use jsonb. No reason to use json.

JSONB is great, BUT
No good query language —
jsonb is a «black box» for SQL
Find something «red»

- Table "public.js_test"
  
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td>not null</td>
</tr>
<tr>
<td>value</td>
<td>jsonb</td>
<td></td>
</tr>
</tbody>
</table>

```sql
select * from js_test;
```

<table>
<thead>
<tr>
<th>id</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1, &quot;a&quot;, true, {&quot;b&quot;: &quot;c&quot;, &quot;f&quot;: false}]</td>
</tr>
</tbody>
</table>
| 2  | {"a": "blue", "t": [{"color": "red", "width": 100}]}
| 3  | [{"color": "red", "width": 100}]
| 4  | {"color": "red", "width": 100}                |
| 5  | {"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"} |
| 6  | {"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"} |
| 7  | {"a": "blue", "t": [{"color": "blue", "width": 100}], "colr": "red"} |
| 8  | {"a": "blue", "t": [{"color": "green", "width": 100}]}
| 9  | {"color": "green", "value": "red", "width": 100} |

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

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

• Not easy!
Find something «red»

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

• **Jsquery**

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

[https://github.com/postgrespro/jsquery](https://github.com/postgrespro/jsquery)

• A language to query jsonb data type
• Search in nested objects and arrays
• More comparison operators with indexes support
JSON in SQL-2016

- BNF
- Discussed at Developers meeting Jan 28, 2017 in Brussels
- Post -hackers, Feb 28, 2017 (March commitfest)
  «Attached patch is an implementation of SQL/JSON data model from SQL-2016 standard (ISO/IEC 9075-2:2016(E)), which was published 2016-12-15 ...»
- Patch was too big (now about 16,000 loc) and too late for Postgres 10 :(
• It’s not a new data type, it’s a JSON data model for SQL
• PostgreSQL implementation is a subset of standard:
  • JSONB - ORDERED and UNIQUE KEYS
  • jsonpath data type for SQL/JSON path language
  • nine functions, implemented as SQL CLAUSEs
• **Jsonpath** provides an ability to operate (in standard specified way) with json structure at SQL-language level
  • Dot notation — $.a.b.c
  • Array - [*]
  • Filter ? - $.a.b.c ? (@.x > 10)
  • Methods - $.a.b.c.x.type()

```
SELECT * FROM js WHERE JSON_EXISTS(js, 'strict $.tags[*] ? (@.term == "NYC")');
```

```
SELECT * FROM js WHERE js @> '{"tags": [{"term": "NYC"}]}';
```
SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
    PASSING 0 AS x, 2 AS y);

<table>
<thead>
<tr>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
</tr>
</tbody>
</table>

(1 row)

SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
    PASSING 0 AS x, 1 AS y);
SQL/JSON in PostgreSQL

- The SQL/JSON **construction** functions:
  - `JSON_OBJECT` - serialization of an JSON object.
    - `json[b]_build_object()`
  - `JSON_ARRAY` - serialization of an JSON array.
    - `json[b]_build_array()`
  - `JSON_ARRAYAGG` - serialization of an JSON object from aggregation of SQL data
    - `json[b]_agg()`
  - `JSON_OBJECTAGG` - serialization of an JSON array from aggregation of SQL data
    - `json[b]_object_agg()`
The SQL/JSON retrieval functions:

- **JSON_VALUE** - Extract an SQL value of a predefined type from a JSON value.
- **JSON_QUERY** - Extract a JSON text from a JSON text using an SQL/JSON path expression.
- **JSON_TABLE** - Query a JSON text and present it as a relational table.
- **IS [NOT] JSON** - test whether a string value is a JSON text.
- **JSON_EXISTS** - test whether a JSON path expression returns any SQL/JSON items
SQL/JSON examples: JSON_VALUE

```
SELECT
  x,
  JSON_VALUE(
    jsonb '{"a": 1, "b": 2}',
    '$.* ? (@ > $x)' PASSING x AS x
  ) y
RETURNING int
DEFAULT -1 ON EMPTY
DEFAULT -2 ON ERROR
) y
FROM
  generate_series(0, 2) x;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
</tbody>
</table>

(3 rows)
SELECT
    JSON_QUERY(js FORMAT JSONB, '$_'),
    JSON_QUERY(js FORMAT JSONB, '$' WITHOUT WRAPPER),
    JSON_QUERY(js FORMAT JSONB, '$' WITH CONDITIONAL WRAPPER),
    JSON_QUERY(js FORMAT JSONB, '$' WITH UNCONDITIONAL ARRAY WRAPPER),
    JSON_QUERY(js FORMAT JSONB, '$' WITH ARRAY WRAPPER)
FROM
    (VALUES
        ('null'),
        ('12.3'),
        ('true'),
        ('"aaa"'),
        ('[1, null, "2"]'),
        ('{"a": 1, "b": [2]}')
    ) foo(js);
CREATE TABLE test_json_constraints (  
    js text,  
    i int,  
    x jsonb DEFAULT JSON_QUERY(jsonb '[1,2]', '$[*]' WITH WRAPPER)  
CONSTRAINT test_json_constraint1 
    CHECK (js IS JSON)  
CONSTRAINT test_json_constraint2  
CHECK (JSON_EXISTS(js FORMAT JSONB, '$.a' PASSING i + 5 AS int, i::text AS txt))  
CONSTRAINT test_json_constraint3  
CHECK (JSON_VALUE(js::jsonb, '$.a' RETURNING int DEFAULT ('12' || i)::int  
    ON EMPTY ERROR ON ERROR) > i)  
CONSTRAINT test_json_constraint4  
    CHECK (JSON_QUERY(js FORMAT JSONB, '$.a'  
    WITH CONDITIONAL WRAPPER EMPTY OBJECT ON ERROR) < jsonb '[10]')  
);
• Creates a relational view of JSON data.
• Think about UNNEST — creates a row for each object inside JSON array and represent JSON values from within that object as SQL columns values.
• Example: Delicious bookmark
• Convert JSON data (1369 MB) to their relational data

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Collation</th>
<th>Nullable</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>js</td>
<td>jsonb</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Delicious bookmarks

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

    * 

FROM

    js, JSON_TABLE(
        js, '$' AS root
    )

    id text,
    link text,
    author text,
    title text,
    NESTED PATH '$.title_detail' AS title_detail

    base text,
    title_type text PATH '$.type',
    value text,
    language text

),

    updated timestamp PATH '$.updated.datetime("Dy, DD Mon YYYY H24:MI:SS TZHTZM")',
    comments text,
    wfw_commentrss text,
    guid_is_link boolean PATH '$.guidislink',
    NESTED PATH '$.tags[*]' AS tags

    tag_term text PATH '$.term',
    tag_scheme text PATH '$.scheme'

),

    NESTED PATH '$.links[*]' AS links

    link_rel text PATH '$.rel',
    link_href text PATH '$.href',
    link_type text PATH '$.type'

) 

PLAN (root INNER (tags UNION links) CROSS title_detail))
jt;
Example: Delicious bookmark

Convert JSON data (1369 MB) to their relational data (2615 MB)

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Collation</th>
<th>Nullable</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>link</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>author</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>title</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>base</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>title_type</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>language</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>updated</td>
<td>timestamp with time zone</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comments</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wfw_commentrss</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>guid_is_link</td>
<td>boolean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag_term</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tag_scheme</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>link_rel</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>link_href</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>link_type</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Find something «red»

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

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

- Jquery

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

- SQL/JSON 2016

SELECT * FROM js_test WHERE JSON_EXISTS(value,'$.**.color ? (@ == "red")');
SQL/JSON availability

• Github Postgres Professional repository
  https://github.com/postgrespro/sqljson

• More examples (thousands !)
  https://github.com/postgrespro/sqljson/blob/sqljson/src/test/regress/sql/sql_json.sql

• BNF is available

• We need your feedback, bug reports and suggestions
• Help us writing documentation !
JSON Roadmap

• Push SQL/JSON to Postgres 11 (Postgres Pro 10)
• Dictionary compression to Postgres 11 (Postgres Pro 10)
Transparent compression of jsonb
+ access to the child elements without full decompression
jsonb compression: motivation

- Long object keys repeated in each document waste a lot of space
- Fixed-size object/array entries overhead is significant for short fields:
  - 4 bytes per array element
  - 8 bytes per object field
- Numbers stored as postgres numerics which have a quite long representation for the short integers:
  - 1-4-digit integers – 8 bytes
  - 5-8-digit integers – 12 bytes
jsonb compression: ideas

- Keys replaced by their id in the external dictionary
- Delta coding for sorted key ID arrays
- Variable-length encoded entries instead of 4-byte fixed-size entries
- Chunked encoding for entry arrays
- Storing integer numerics falling into int32 range as variable-length encoded 4-byte integers
jsonb compression: implementation

• Custom column compression methods:

CREATE COMPRESSION METHOD name HANDLER handler_func
CREATE TABLE table_name (column_name data_type [ COMPRESSED cm_name [ WITH (option 'value' [, ... ]) ] ] ...)  
ALTER TABLE table_name ALTER column_name SET COMPRESSED cm_name [ WITH (option 'value' [, ... ]) ]  
ALTER TYPE data_type SET COMPRESSED cm_name

• attcompression, attcmoptions in pg_attributes
jsonb compression: jsonbc

- **jsonbc** is a compression method for jsonb type using dictionary compression for object keys and more compact variable-length encoding.

- All key dictionaries for all jsonbc compressed columns are stored in the single catalog relation **pg_jsonbc_dict**:

  ```
  pg_jsonbc_dict (  
    dict oid,  
    id integer,  
    name text  
  )
  ```

- Dictionary used by jsonb column is identified by:
  - sequence oid – automatically updated
  - enum type oid – manually updated
Examples:

-- automatical test_js_jsonbc_dict_seq creation for generating key IDs
CREATE TABLE test (js jsonb COMPRESSED jsonbc);

-- manual dictionary sequence creation
CREATE SEQUENCE test2_dict_seq;
CREATE TABLE test2 (js jsonb COMPRESSED jsonbc WITH (dict_id 'test2_dict_seq'));

-- enum type as a manually updatable dictionary
CREATE TYPE test3_dict_enum AS ENUM ('key1', 'key2', 'key3');
CREATE TABLE test3 (js jsonb COMPRESSED jsonbc WITH (dict_enum 'test3_dict_enum'));

-- manually updating enum dictionary (before key4 insertion into table)
ALTER TYPE test3_dict_enum ADD VALUE 'key4';
jsonb compression: results

Two datasets:
- js – Delicious bookmarks, 1.2 mln rows
  - Mostly string values
  - Relatively short keys
  - 2 arrays (tags and links) of 3-field objects
- jr – Citus Data dataset: customer reviews data from Amazon, 3mln
  - Rather long keys
  - A lot of short integer numbers

Also, jsonbc compared with CFS (Compressed File System) – page level compression and encryption in Postgres Pro Enterprise 9.6.
jsonb compression: table size

size by datatype

storage engine
- cfs zstd
- pg
- wt snappy
jsonb compression: performance

```
SELECT count(*) FROM js WHERE js @> '{"tags": [{"term": "NYC"}]}'
db.js.find({ tags: { $elemMatch: { term: "NYC"} } }).count()
```
jsonb compression: performance

SELECT js->>'id', js->>'title', js->>'updated' FROM js;

- rel
- jsonb
- jsonbc
- bson

storage engine:
- cfs zstd
- pg
- wt snappy

run time, ms
jsonb compression: performance

```
SELECT js->>'product_group', avg(js->>'review_rating')::int) FROM jr GROUP BY 1;
```

```
db.jr.aggregate([[${group: {_id: "$product_group", rating: { $avg: "$review_rating"}}}}])
```
jsonb compression: jsonbc problems

• Transactional dictionary updates

  Currently, automatic dictionary updates uses background workers, but autonomous transactions would be better

• Cascading deletion of dictionaries not yet implementing. Need to track dependency between columns and dictionaries

• User compression methods for jsonb are not fully supported (should we ?)
jsonb compression: summary

- jsonbc can reduce jsonb column size to its relational equivalent size
- jsonbc has a very low CPU overhead over jsonb and sometimes even can be faster than jsonb
- jsonbc compression ratio is significantly lower than in page level compression methods
- jsonbc still needs some work
- Availability:
  https://github.com/postgrespro/postgrespro/tree/jsonbc
BENCHMARKS:
How NoSQL Postgres is fast
Summary: PostgreSQL 9.4 vs Mongo 2.6.0

• Search key=value (contains @>)
  • json : 10 s  seqscan
  • jsonb : 8.5 ms  GIN jsonb_ops
  • jsonb : 0.7 ms  GIN jsonb_path_ops
  • mongo : 1.0 ms  btree index

• Index size
  • jsonb_ops : - 636 Mb (no compression, 815Mb)
  • jsonb_path_ops : - 295 Mb
  • jsonb_path_ops (tags) : - 44 Mb  USING gin((jb->'tags') jsonb_path_ops
  • mongo (tags) : - 387 Mb
  • mongo (tags.term) : - 100 Mb

• Table size
  • postgres : 1.3Gb
  • mongo : 1.8Gb

• Input performance:
  • Text : 34 s
  • Json : 37 s
  • Jsonb : 43 s
  • mongo : 13 m

First (non-scientific) benchmark !
EDB NoSQL Benchmark

MongoDB 2.6 vs PostgreSQL 9.4 Performance

https://github.com/EnterpriseDB/pg_nosql_benchmark
Benchmarking NoSQL Postgres

• Both benchmarks are homemade by postgres people
• People tend to believe independent and «scientific» benchmarks
  • Reproducible
  • More databases
  • Many workloads
  • Open source
YCSB Benchmark

• Yahoo! Cloud Serving Benchmark - https://github.com/brianfrankcooper/YCSB/wiki

• De-facto standard benchmark for NoSQL databases


• We run YCBS for Postgres master and MongoDB 3.4.2
  • 1 server with 24 cores, 48 GB RAM for clients
  • 1 server with 24 cores, 48 GB RAM for database
  • 10Gbps switch
YCSB Benchmark: Core workloads

- Workload A: Update heavy - a mix of 50/50 reads and writes
- Workload B: Read mostly - a 95/5 reads/write mix
- Workload C: Read only — 100% read
- Workload D: Read latest - new records are inserted, and the most recently inserted records are the most popular
- Workload E: Short ranges - short ranges of records are queried
- Workload F: Read-modify-write - the client will read a record, modify it, and write back the changes
- All workloads uses Zipfian distribution for record selections
YCSB Benchmark: details

• Postgres (9.6, master), asynchronous commit=on
  Mongodb 3.4.2 (w1, j0) — 1 and 5 mln. rows

• Postgres (9.6, master), asynchronous commit=off
  Mongodb 3.4.2 (w1, j1) — 100K rows (Mongo is too slow !)

• fastupdate=off for GIN index

• We tested:
  • functional btree index for jsonb, jsonbc, sqljson
  • Functional btree index for jsonb on cfs (compressed)
  • Gin index for jsonb, jsonb_build_object
  • Mongodb (wiredtiger with snappy compression)
  • Return a whole json, just one field
  • 10 fields, 200 fields (TOASTed)
HOT update for jsonb

- If we have functional index on keyA and update keyB, then HOT update doesn’t work (black line).
- We know YCSB doesn’t update indexed keys.
- Quick and Dirty fix (blue line) in HeapSatisfiesHOTandKeyUpdate() used in all runs.
- The problem exists not only for jsonb, need a real fix!
1 mln rows, 10 fields

- Postgres is better in all workloads!
- Jsonb ~ jsonb(cfs) ~ Jsonbc ~ sqljson
- Gin is not good for updates
5 mln rows, 10 fields

- Postgres is better in all workloads!
- Gin is not good for updates
1 mln rows, 10 fields, (journaled to disk, asynchronous_commit off)

- Postgres is better in all workloads!
1mln rows, 200 fields (TOASTed)

MONGODB is better on some workloads.
PostgreSQL still beats Mongodb!

on one server
Summary

• Postgres is already a good NoSQL database

• SQL/JSON will provide better flexibility and interoperability
  • Expect it in Postgres 11 (Postgres Pro 10)
  • Need community help (testing, documentation)
  • What’s about index support?

• JSONB dictionary compression (jsonbc) is really useful
  • Expect it in Postgres 11 (Postgres Pro 10)

• Postgres beats Mongodb in one node configuration
  • HOT for JSONB, type specific compression
  • YCSB benchmarks in distributed mode (Citus ?)

• Postgres will help NoSQL users to avoid nightmare
  • Need sharding, community is working on it. Expect in 2-3 years!
PEOPLE BEHIND JSON[B]

Nikita Glukhov
Thanks !