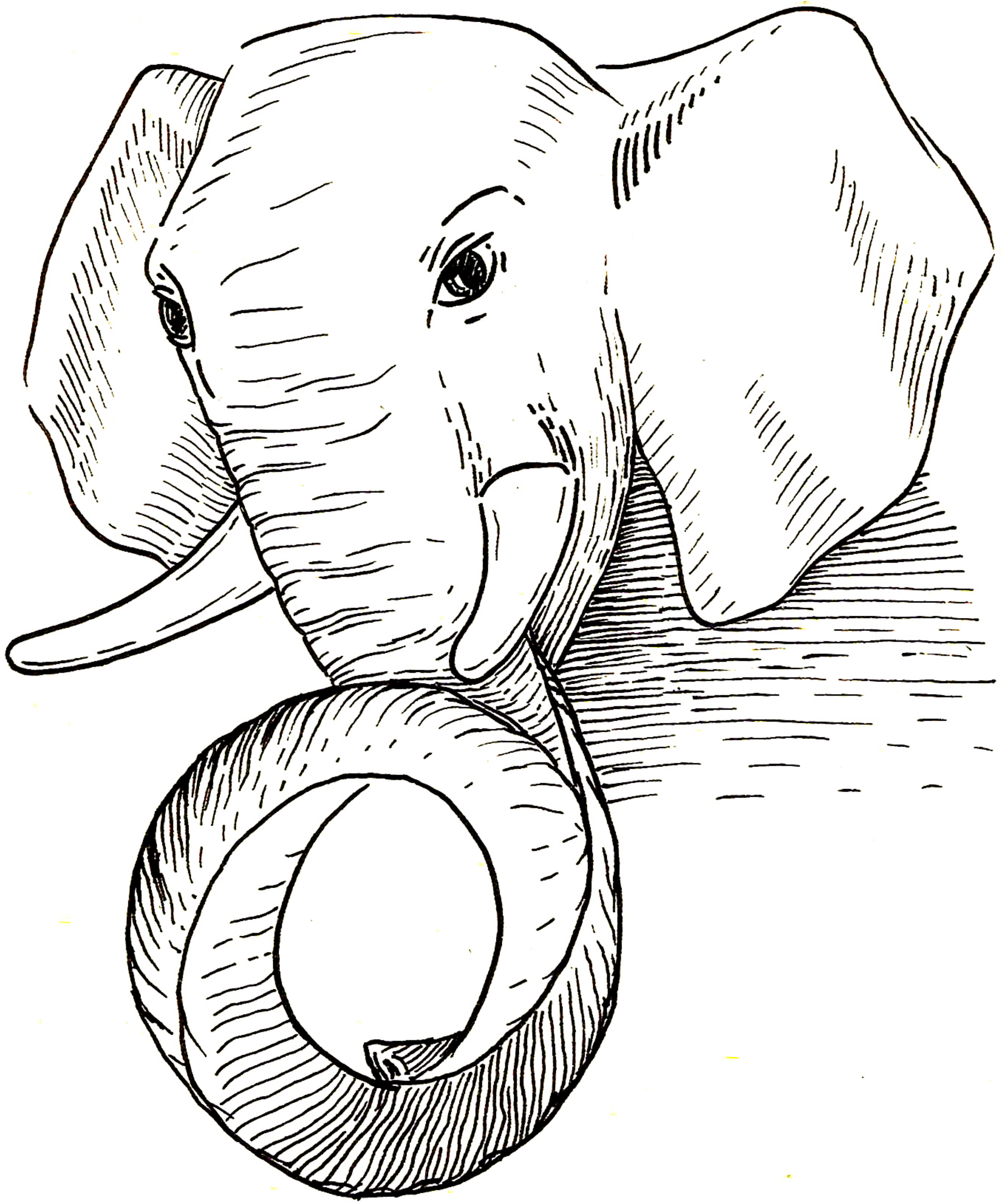


Postgres vs Mongo

Oleg Bartunov
Postgres Professional
Moscow University

2017

RIT, Moscow, June 6, 2017

NoSQL (предпосылки)

- Relational DBMS - integrational
  - All APPs communicaties through RDBMS
  - SQL — universal language to work with data
  - All changes in RDBMS are available to all
  - Changes of the scheme are difficult, so → slow releases
  - Mostly for interactive work
    - Aggregates are mostly interested, not the data itself, SQL is needed
    - SQL takes cares about transactions, consistency … instead of human

The problem

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

NoSQL databases (wikipedia) ...+++

Document store

* Lotus Notes
* CouchDB
* MongoDB
* Apache Jackrabbit
* Colayer
* XML databases
  o MarkLogic Server
  o eXist

Graph
* Neo4j
* AllegroGraph

Tabular
* BigTable
* Mnesia
* Hbase
* Hypertable

Key/value store on disk

* Tuple space
* Memcachedb
* Redis
* SimpleDB
* flare
* Tokyo Cabinet
* BigTable

Key/value cache in RAM

* memcached
* Velocity
* Redis

Eventually-consistent key-value store
* Dynamo
* Cassandra
* Project Voldemort

Ordered key-value store
* NMDB
* Luxio
* Memcachedb
* Berkeley DB

Object database
* Db4o
* InterSystems Caché
* Objectivity/DB
* ZODB

The problem

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

JSON in PostgreSQL

This is the challenge !

Challenge to PostgreSQL !

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

Introduction to Hstore

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

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

Introduction to Hstore

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

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

Introduction to hstore

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

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

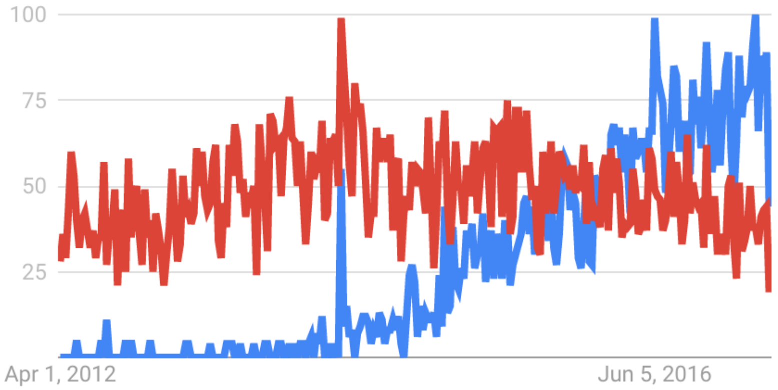
  - Key, value — strings
  - Get value for a key: hstore -> text
  - Operators with indexing support (GiST, GIN)

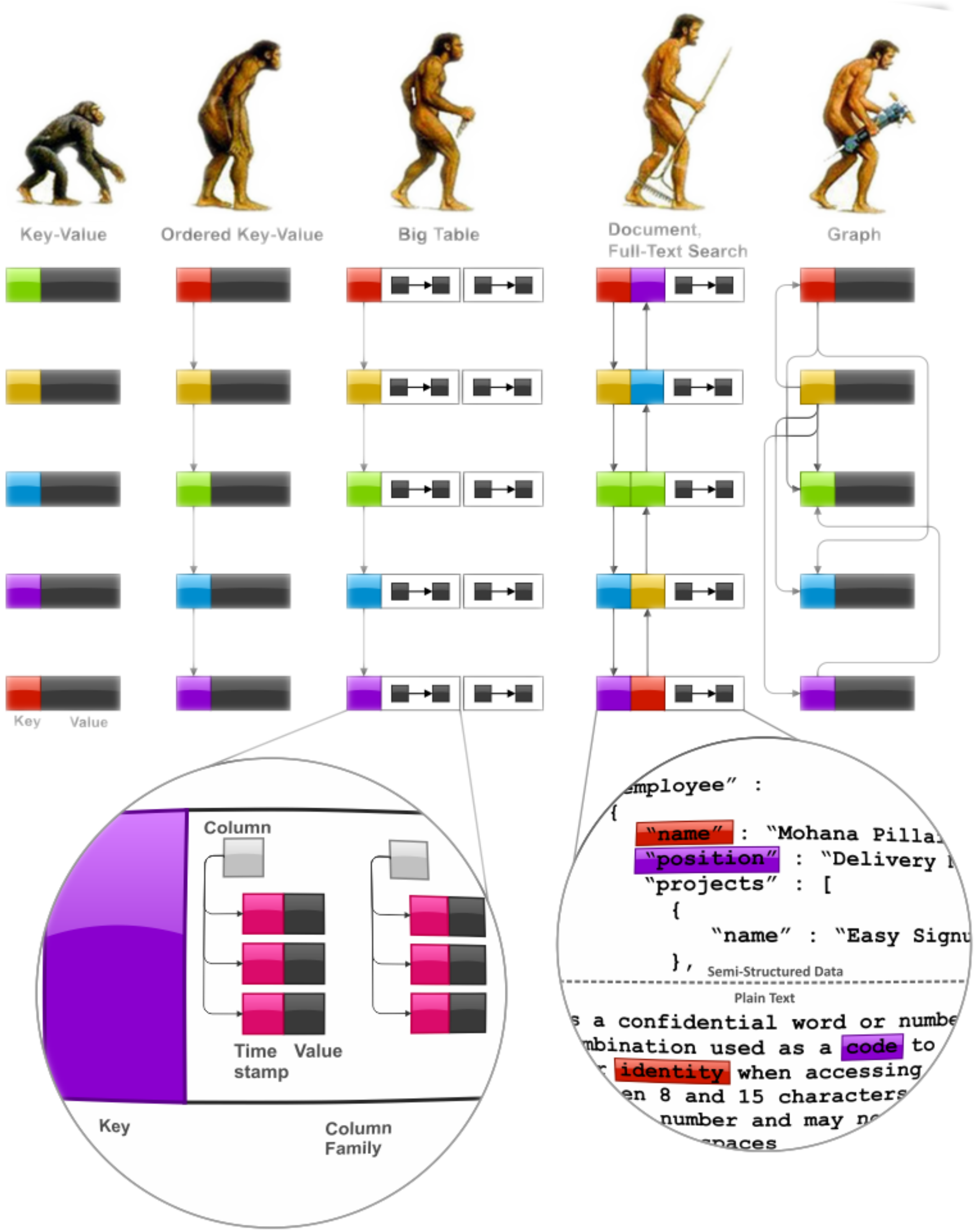
    Check for key: hstore ? text
    Contains: hstore @> hstore
  - check documentations for more
  - Functions for hstore manipulations (akeys, avals, skeys, svals, each,......)
- Hstore provides PostgreSQL schema-less feature !
  - Faster releases, no problem with schema upgrade
  - Better indexing support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

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, Ottawa: 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 vs hstore





JSONB - 2014

- Binary storage
- Nesting objects & arrays
- Indexing

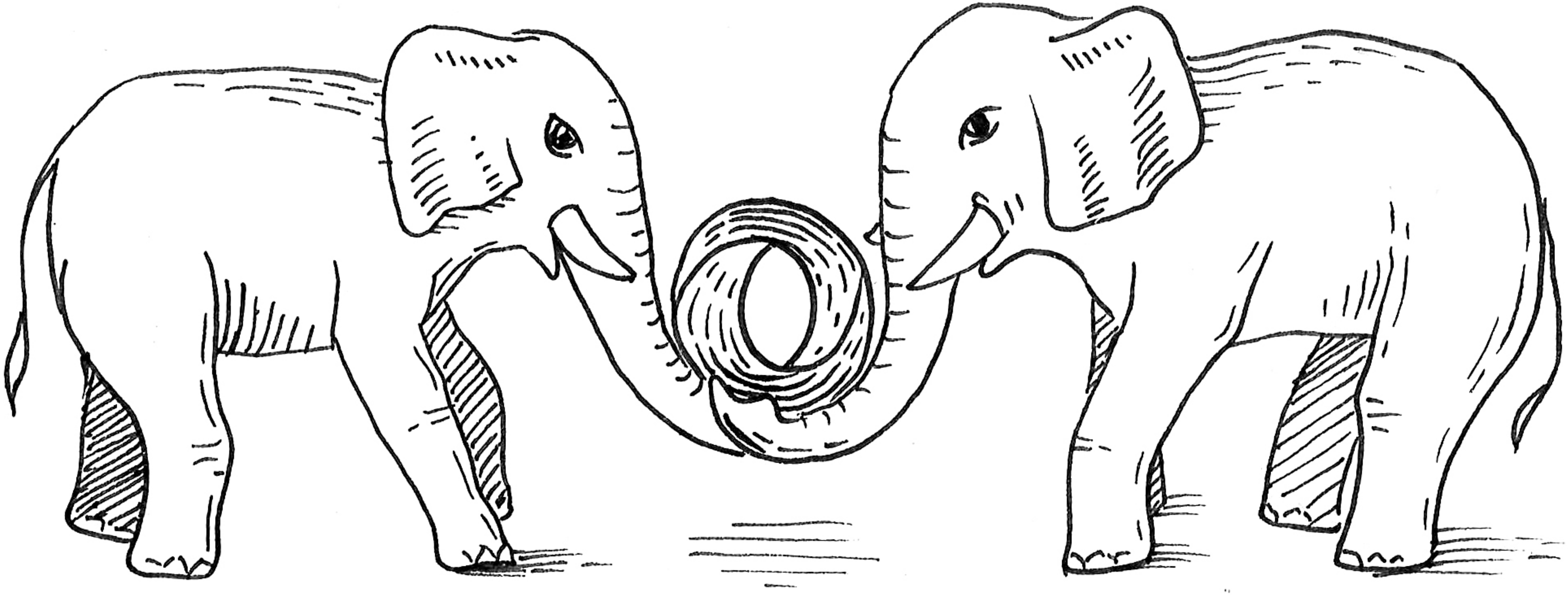
JSON - 2012

- Textual storage
- JSON verification

HSTORE - 2003

- Perl-like hash storage
- No nesting
- Indexing

Two JSON data types !!!

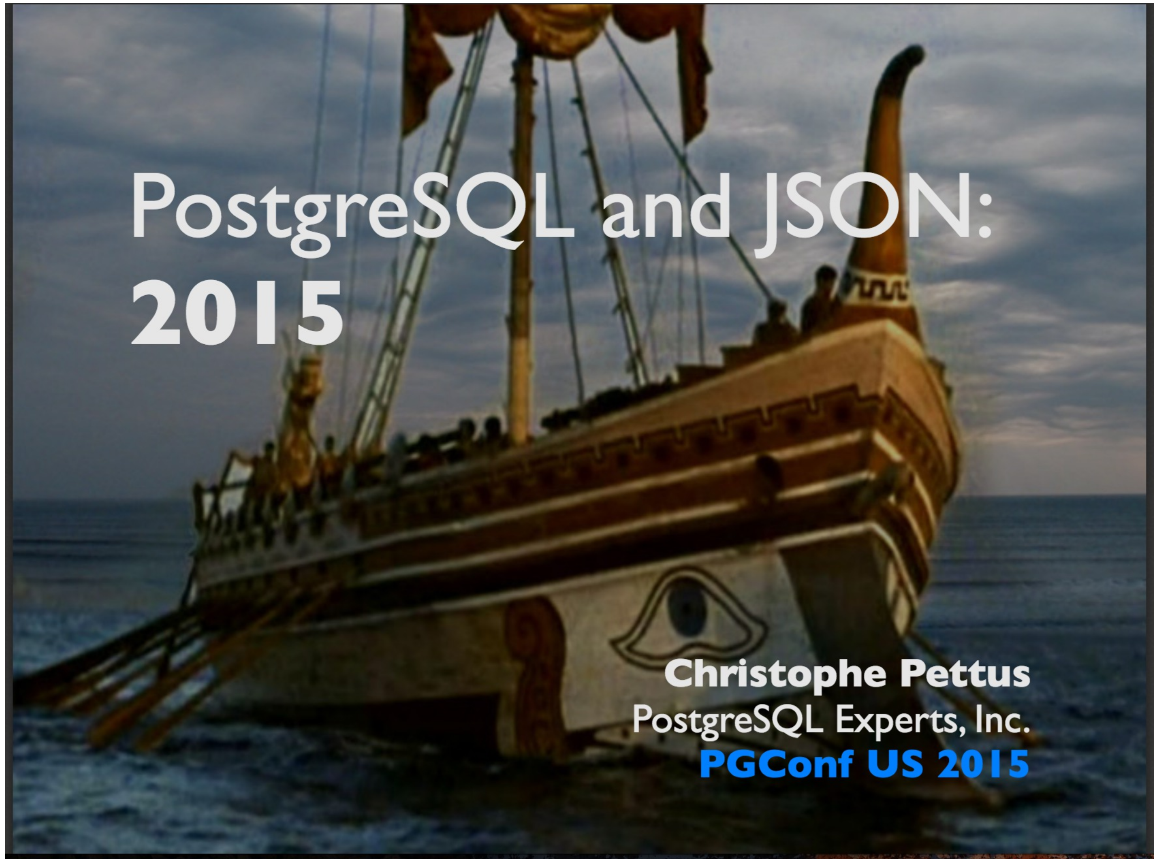


Jsonb vs Json

```
SELECT j::json AS json, j::jsonb AS jsonb FROM
(SELECT '{"cc":0, "aa":  2, "aa":1,"b":1}' AS j) AS foo;
               json               |              jsonb
----------------------------------+--------------------------------
 {"cc":0, "aa":  2, "aa":1,"b":1} | {"b": 1, "aa": 1, "cc": 0}
(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]



http://thebuild.com/presentations/json2015-pgconfus.pdf

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

Find something «red»

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

select * from js_test;

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

Find something «red»

- VERY COMPLEX SQL QUERY

```
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;
```

```
id |                    value
----+---------------------------------------------------------------
 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"}
(5 rows)
```

PGCon-2014, Ottawa

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;
```

- **Jsquery**

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

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

JSON in SQL-2016

- ISO/IEC 9075-2:2016(E) - https://www.iso.org/standard/63556.html
- BNF
  https://github.com/elliotchance/sqltest/blob/master/standards/2016/bnf.txt
- 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 :(

SQL/JSON in PostgreSQL

- 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

SQL/JSON in PostgreSQL

- **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"}]}';
```

SQL/JSON in PostgreSQL

```
SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
            PASSING 0 AS x, 2 AS y);
 ?column?
----------
 t
(1 row)

SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
            PASSING 0 AS x, 1 AS y);
 ?column?
----------
 f
(1 row)
```

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()

SQL/JSON in PostgreSQL

- 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
                     RETURNING int
                     DEFAULT -1 ON EMPTY
                     DEFAULT -2 ON ERROR
                    ) y
FROM
generate_series(0, 2) x;
 x | y
---+----
 0 | -2
 1 |  2
 2 | -1
(3 rows)
```

SQL/JSON examples: JSON_QUERY

```
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);
     ?column?       |      ?column?      |      ?column?      |       ?column?       |       ?column?
--------------------+--------------------+--------------------+----------------------+----------------------
 null               | null               | [null]             | [null]               | [null]
 12.3               | 12.3               | [12.3]             | [12.3]               | [12.3]
 true               | true               | [true]             | [true]               | [true]
 "aaa"              | "aaa"              | ["aaa"]            | ["aaa"]              | ["aaa"]
 [1, null, "2"]     | [1, null, "2"]     | [1, null, "2"]     | [[1, null, "2"]]     | [[1, null, "2"]]
 {"a": 1, "b": [2]} | {"a": 1, "b": [2]} | {"a": 1, "b": [2]} | [{"a": 1, "b": [2]}] | [{"a": 1, "b": [2]}]
(6 rows)
```

SQL/JSON examples: Constraints

```
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]')
);
```

SQL/JSON examples: JSON_TABLE

- 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 "public.js"
 Column | Type  | Collation | Nullable | Default
--------+-------+-----------+----------+---------
 js     | jsonb |           |          |
```

Delicious bookmarks


```
{                                                                                  +
    "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                                                           +
    },                                                                             +
    "wfw_commentrss": "http://feeds.delicious.com/v2/rss/url/b5b3cbf9a9176fe43c27d7b4af94a422"+
}
```

```sql
SELECT
    jt.*
FROM
    js,
    JSON_TABLE(
        js, '$' AS root
        COLUMNS (
            id text,
            link text,
            author text,
            title text,
            NESTED PATH '$.title_detail' AS title_detail COLUMNS (
                base text,
                title_type text PATH '$.type',
                value text,
                language text
            ),
            updated timestamptz PATH '$.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 COLUMNS (
                tag_term   text PATH '$.term',
                tag_scheme text PATH '$.scheme'
            ),
            NESTED PATH '$.links[*]' AS links COLUMNS (
                link_rel  text PATH '$.rel',
                link_href text PATH '$.href',
                link_type text PATH '$.type'
            )
        )
        PLAN (root INNER ((tags UNION links) CROSS title_detail))
    ) jt;
```

SQL/JSON examples: JSON_TABLE

- Example: Delicious bookmark
- Convert JSON data (1369 MB) to their relational data (2615 MB)

```
                         Table "public.js_rel"
     Column      |           Type           | Collation | Nullable | Default
-----------------+--------------------------+-----------+----------+---------
 id              | text                     |           |          |
 link            | text                     |           |          |
 author          | text                     |           |          |
 title           | text                     |           |          |
 base            | text                     |           |          |
 title_type      | text                     |           |          |
 value           | text                     |           |          |
 language        | text                     |           |          |
 updated         | timestamp with time zone |           |          |
 comments        | text                     |           |          |
 wfw_commentrss  | text                     |           |          |
 guid_is_link    | boolean                  |           |          |
 tag_term        | text                     |           |          |
 tag_scheme      | text                     |           |          |
 link_rel        | text                     |           |          |
 link_href       | text                     |           |          |
 link_type       | text                     |           |          |
```

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;
```

- Jsquery

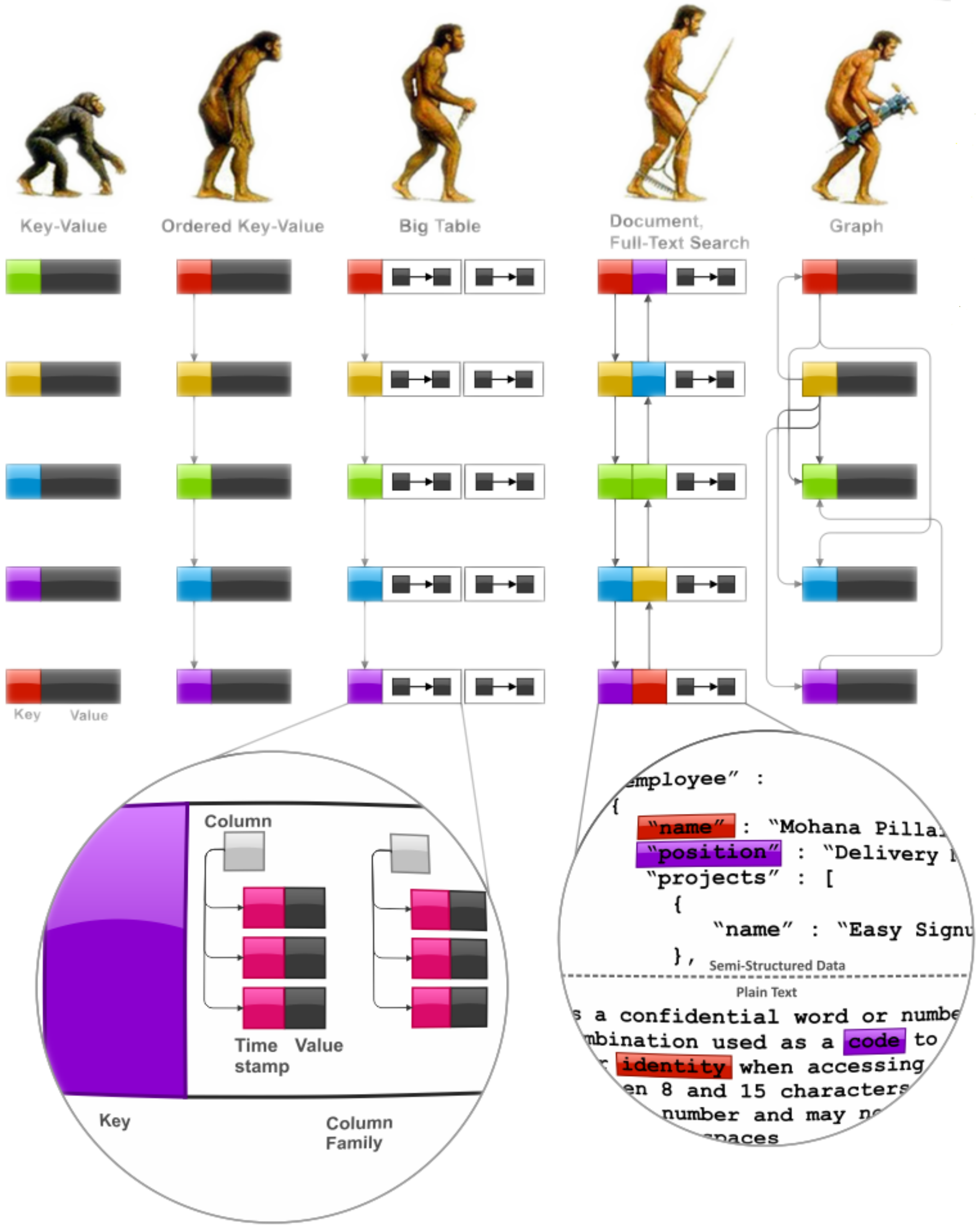
```
SELECT * FROM js_test
 WHERE
 value @@  '*.color = "red"';
```

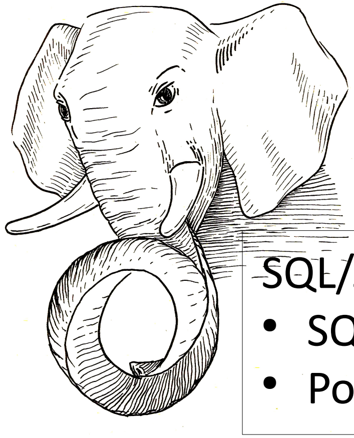
- **SQL/JSON 2016**

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

SQL/JSON availability

- Github Postgres Professional repository
  https://github.com/postgrespro/sqljson
- SQL/JSON examples
- WEB-interface to play with SQL/JSON
- BNF of SQL/JSON
- We need your feedback, bug reports and suggestions
- Help us writing documentation !





SQL/JSON - 2018
- SQL-2016 standard
- Postgres Pro - 2017

JSONB - 2014
- Binary storage
- Nesting objects & arrays
- Indexing

JSON - 2012
- Textual storage
- JSON verification

HSTORE - 2003
- Perl-like hash storage
- No nesting
- Indexing

SQL/JSON TODO

JSON path

:: casts

Postgres operators

'$.tags @> [{"term": "NYC"}]'
-- what about == and &&

'$.a::text || "abc"'

Item methods

'$.sum()'

'$.avg()'

'$.indexOf(5)'

'$.indexWhere(@ > 5)'

'$.maxBy(@.a)'

'$.sort(@.a)'

'$.sortBy(@.a)'

'$.sortWith($1 > $2)'

'$.partition(@ > 5)'

'$.groupBy(@.a)'

'$.indexWhere(@ > 5)'

'$.a.zip($.b)' or '$.a zip $.b'

'$.a.zipWithIndex()'

SQL/JSON ToDO

Postgres functions (including aggregate) support

'$.a[*].map(lower(@))' -- lower(text) used as a function

'$.a[*].lower()' -- lower(text) used as an item method

'$.a[*].concat('foo')' -- concat(any, ...) used as an item method

'avg($.a[*])' -- avg(numeric) aggregate function

Postgres operators support ??? (remember &&, ||, ==)

Item aliases, named parameters in lambdas ???

Reusing PostgreSQL executor???

SQL/JSON ToDO

SQL/JSON functions

SRF (it seems hard to implement now)

JSON_QUERY(json, jsonpath RETURNING SETOF type)

JSON_ITEMS(json, jsonpath)

Insert

JSON_INSERT('{"a": 1}', '$.b = 2') -- {a:1, b:2}

JSON_INSERT('[1,2,3]', '$[0] = 0') -- [0,1,2,3]

JSON_INSERT('[1,2,3]', '$[last] = 4') -- [1,2,3,4]

SQL/JSON ToDO

SQL/JSON functions

Delete

JSON_DELETE('{"a": {"b": 1}, "c": 2}', '$.a.b, $.c') -- {"a":{}}

JSON_DELETE('[1,2,3,4,5]', '$[*] ? (@ > 3)') -- [1,2,3]

Update

JSON_UPDATE('{"counter": 1}', '$.counter = $.counter + 1')

JSON_UPDATE('{"counter": 1}', '$.counter = (@ + $increment)' PASSING 5 AS increment)

JSON_UPDATE('[1,2,3]', '$[*] = (@ + 1)')

JSON_UPDATE('{"a": 1, "b": 2}', '$.a = (@ + 1), $.b = (@ - 1)')

It might be very useful to combine somehow multiple modification operations into a single expression

SQL/JSON ToDO

Index support?!

JSONB COMPRESSION

Transparent compression of jsonb

+ access to the child elements without full decompression

jsonb compression: motivation

- Long object keys repeated in each document is a waste of 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 — overhead 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_catalog.pg_attributes

jsonb compression: jsonbc

- **Jsonbc** - compression method for jsonb type:
  - dictionary compression for object keys
  - more compact variable-length encoding

- All key dictionaries for all jsonbc compressed columns are stored in the pg_catalog.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

json compression: jsonbc dictionaries

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';
```

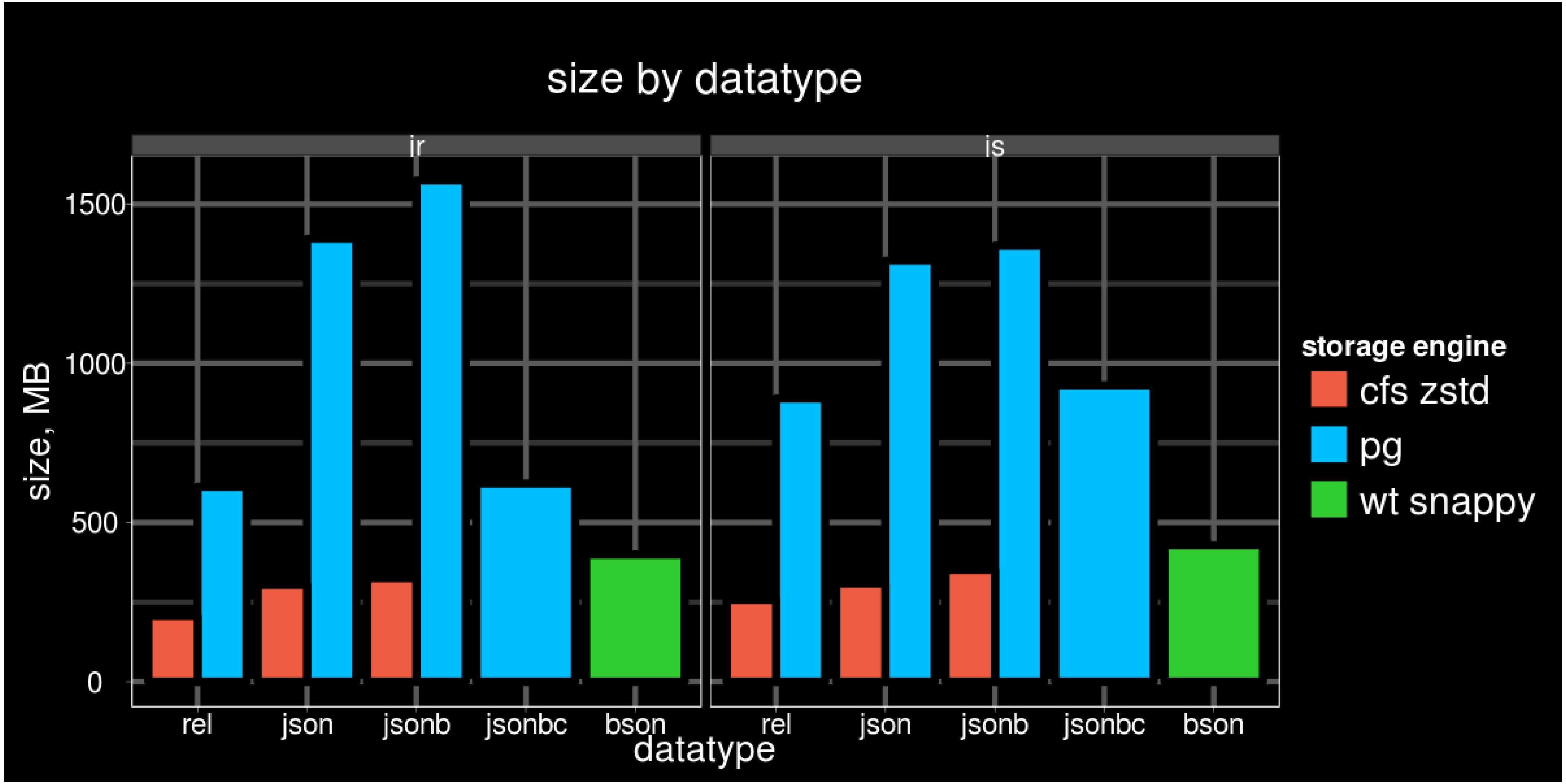
jsonb compression: results

Two datasets:

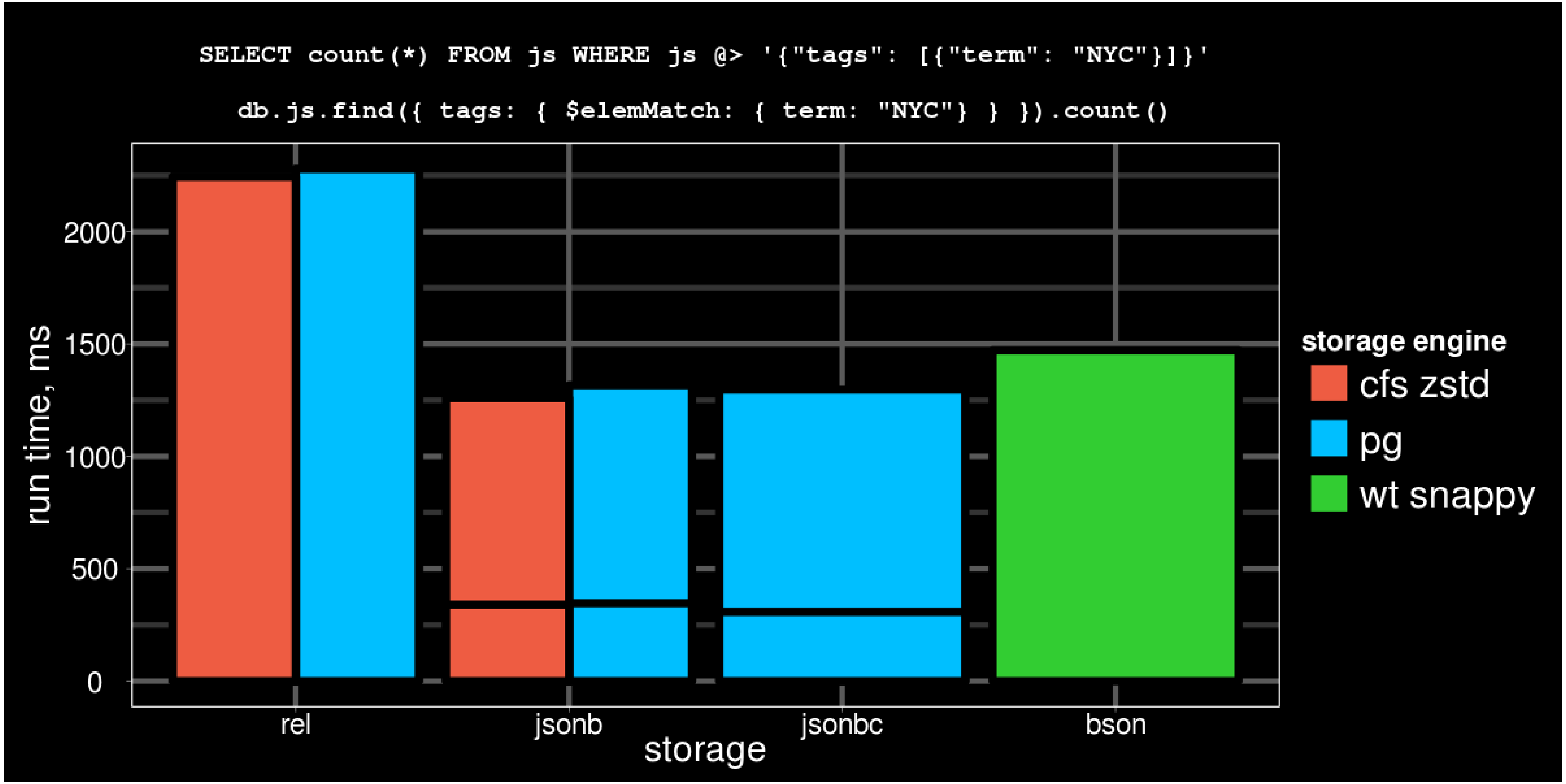
- js – Delicious bookmarks, 1.2 mln rows (js.dump.gz)
  - Mostly string values
  - Relatively short keys
  - 2 arrays (tags and links) of 3-field objects
- jr – customer reviews data from Amazon, 3mln (jr.dump.gz)
  - Rather long keys
  - A lot of short integer numbers

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

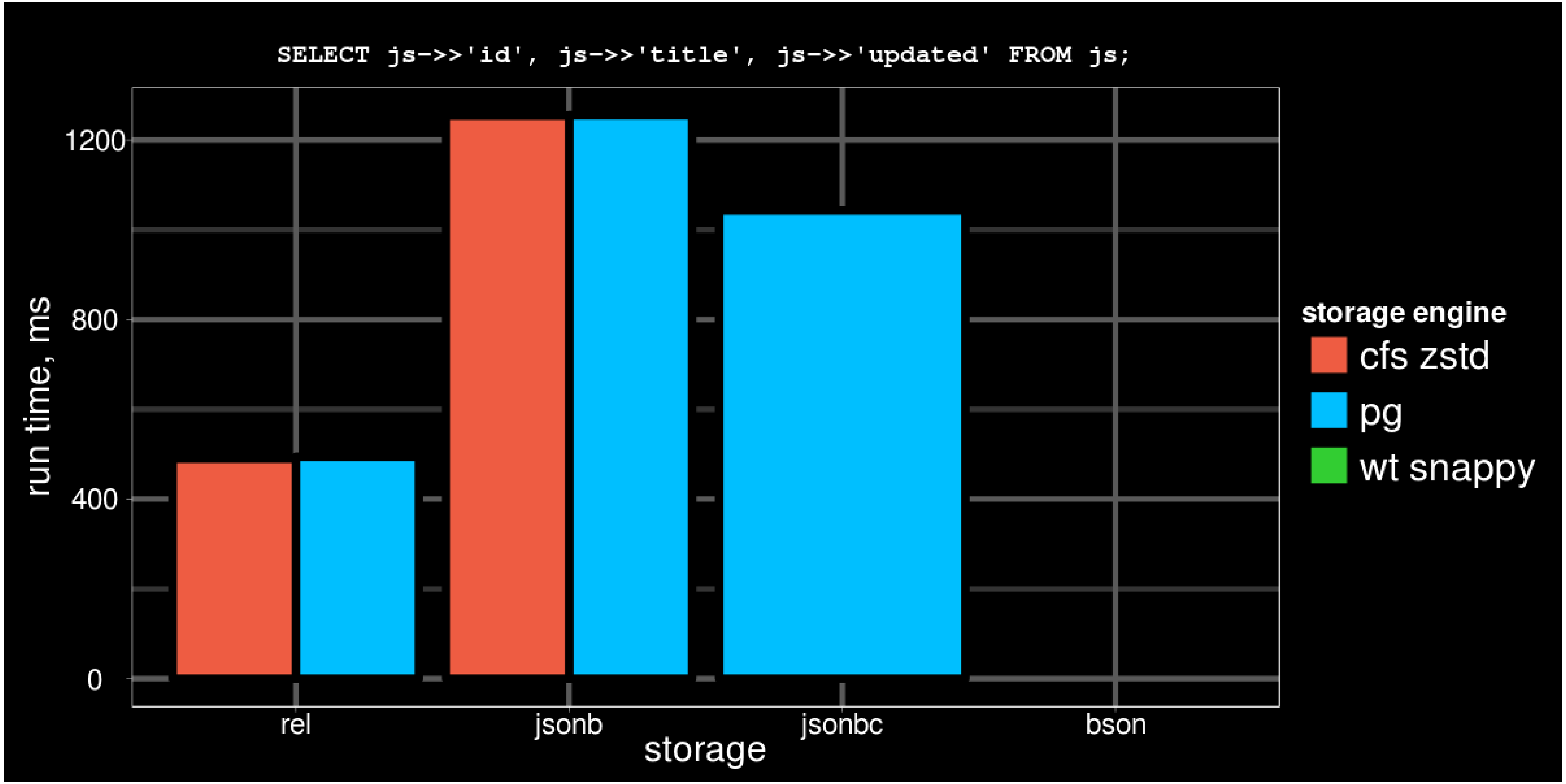
jsonb compression: table size



jsonb compression (js): performance

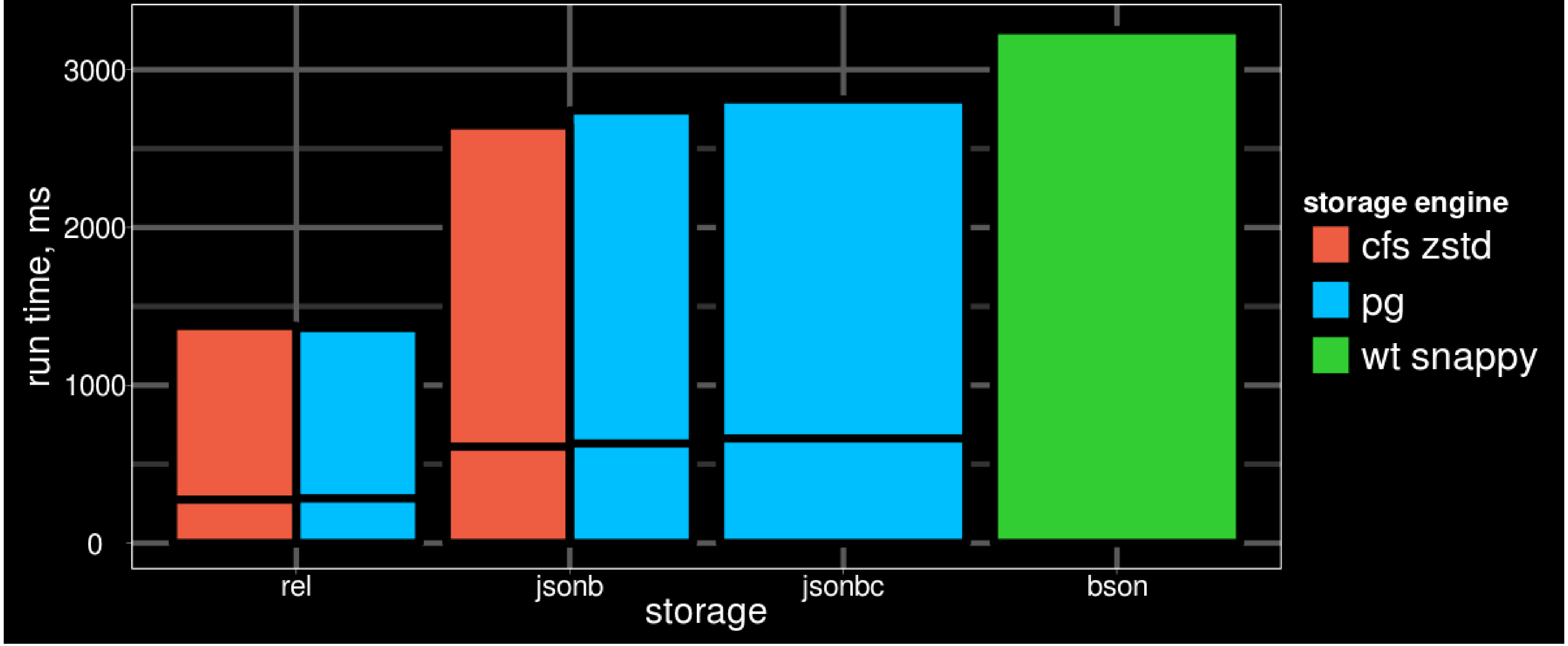


jsonb compression (js): performance



jsonb compression (jr): 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 can be even faster than jsonb
- jsonbc compression ratio is significantly lower than in page level compression methods
- Availability:

  https://github.com/postgrespro/postgrespro/tree/jsonbc

JSON[B] Text Search

- tsvector(configuration, json[b]) in Postgres 10

```
select to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::json)) foo(jb);
                                         to_tsvector
------------------------------------------------------------------------------------------
'fals':10 'hous':18 'intern':17 'long':5 'moscow':16 'peac':12 'stori':6 'true':8 'war':14
```

```
select to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
                                         to_tsvector
------------------------------------------------------------------------------------------
 'fals':14 'hous':18 'intern':17 'long':9 'moscow':16 'peac':1 'stori':10 'true':12 'war':3
```

JSON[B] Text Search

- Phrase search is [properly] supported !

```
select  phraseto_tsquery('english','war moscow') @@ to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 f
```

```
select  phraseto_tsquery('english','moscow international') @@ to_tsvector(jb) from
(values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 t
```

- Kudos to Dmitry Dolgov & Andrew Dunstan !

BENCHMARKS:
How NoSQL Postgres is fast

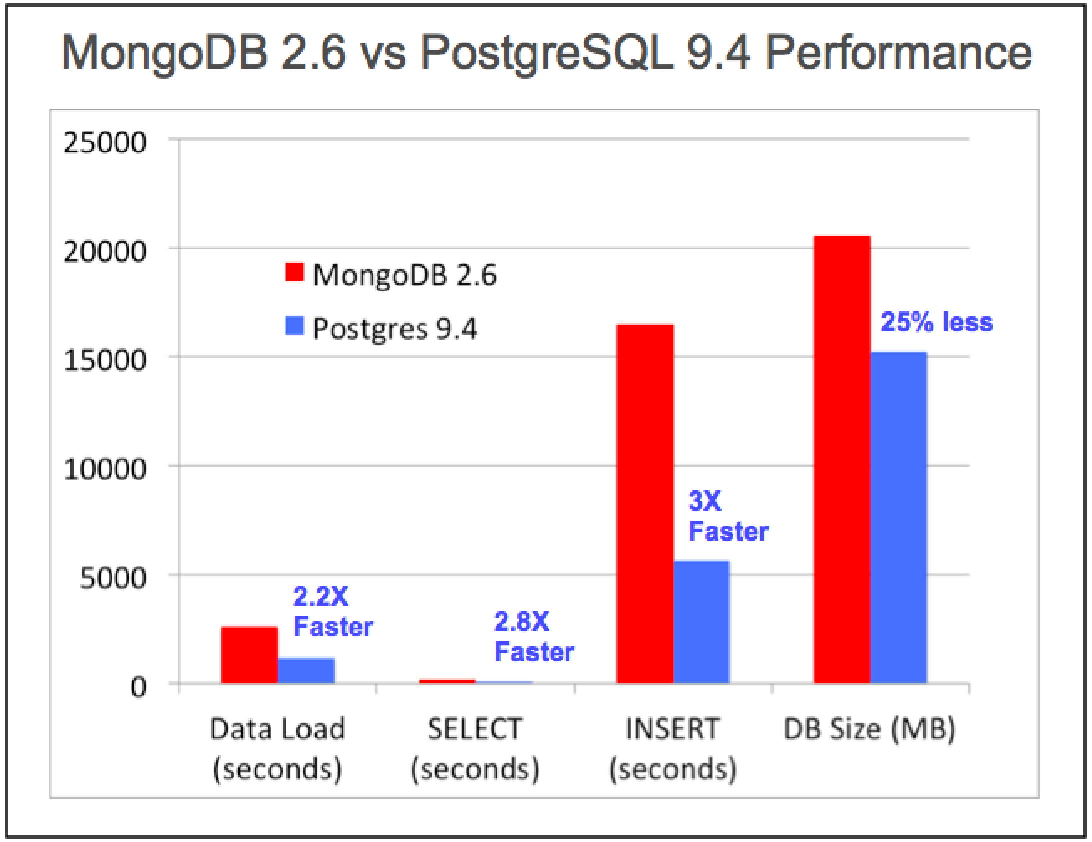
First (non-scientific) benchmark !

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

EDB NoSQL Benchmark



https://github.com/EnterpriseDB/pg_nosql_benchmark

Benchmarking NoSQL Postgres

- Both benchmarks were 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
- Scientific paper «Benchmarking Cloud Serving Systems with YCSB» https://www.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf
- We run YCBS for Postgres master, Postgres Pro Enterprise 2.0, MongoDB 3.4.2, Mysql 5.7.17
    - 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 (except D) workloads uses Zipfian distribution for record selections

YCSB Benchmark: details (1)

- 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
- MySQL 5.7.17 + all optimization
- We tested:
  - Functional btree index for jsonb, jsonbc, sqljson, cfs (compressed) storage
  - Gin index (fastupdate=off) for jsonb, jsonb_build_object
  - Mongodb (wiredtiger with snappy compression)
  - Return a whole json, just one field, small range
  - 10 fields, 200 fields (TOASTed)

YCSB Benchmark: details (2)

- Client machine load:
  - Postgres <= 30%
    Mongodb <= 55%
- Server machine load:
  - Postgres — 100%
  - MySQL — 100%
  - MongoDB — 70%

YCSB Benchmark: PostgreSQL

- Table:
  ```
  CREATE TABLE usertable(data jsonb);
  CREATE TABLE usertable(data jsonb COMPRESSED jsonbc);
  ```
- Btree index:
  CREATE INDEX usertable_bt_idx ON usertable ((data→>'YCSB_KEY'));
  - SELECT data FROM usertable WHERE data->>'YCSB_KEY' = ?;
  - SELECT data→>'field5' FROM usertable WHERE data->>'YCSB_KEY' = ?;
  - SELECT data→>'field5' FROM usertable WHERE data->>'YCSB_KEY' > ? LIMIT ?
  - UPDATE usertable SET data = data || ?WHERE data->>'YCSB_KEY' = ?;

YCSB Benchmark: PostgreSQL

- Btree SQL/JSON index:
  CREATE INDEX usertable_sqljson_idx ON usertable ((JSON_VALUE(data, '$.YCSB_KEY' RETURNINIG text));
  - SELECT data FROM usertable WHERE JSON_VALUE(data, '$.YCSB_KEY' RETURNINIG text) = ?;
  - SELECT JSON_VALUE(data, '$.field5' RETURNINIG text) FROM usertable WHERE JSON_VALUE(data, '$.YCSB_KEY' RETURNINIG text) = ?;
  - SELECT JSON_VALUE(data, '$.field5' RETURNINIG text) FROM usertable WHERE JSON_VALUE(data, '$.YCSB_KEY' RETURNINIG text) > ? LIMIT ?
  - UPDATE usertable SET data = data || ?
    WHERE JSON_VALUE(data, '$.YCSB_KEY' RETURNINIG text) = ?;

YCSB Benchmark: PostgreSQL

- GIN index:
  CREATE INDEX usertable_gin_idx ON usertable USING gin (data jsonb_path_ops);
  gin-jbo (JSONB_BUILD_OBJECT)
  - SELECT data FROM usertable WHERE data @> jsonb_build_object('YCSB_KEY', ?);
  - SELECT data→>'field5' FROM usertable WHERE data @> jsonb_build_object('YCSB_KEY', ?);
  - UPDATE usertable SET data = data || ? WHERE data @> jsonb_build_object('YCSB_KEY', ?);

YCSB Benchmark: PostgreSQL

- GIN index:
  CREATE INDEX usertable_gin_idx ON usertable USING gin (data jsonb_path_ops);
  gin-jsonb
  - SELECT data FROM usertable WHERE data @> ?::jsonb;
  - SELECT data→>'field5' FROM usertable WHERE data @> ?::jsonb;
  - UPDATE usertable SET data = data || ? WHERE data @> ?::jsonb;

YCSB Benchmark: MySQL

- Table
  CREATE TABLE usertable(
  data json,
  ycsb_key CHAR(255) GENERATED ALWAYS AS (data->>'$.YCSB_KEY'),
  INDEX ycsb_key_idx(ycsb_key)
  );
  - SELECT data FROM usertable WHERE ycsb_key = ?;
  - SELECT data→>'$.field5' FROM usertable WHERE ycsb_key = ?;
  - SELECT data FROM usertable WHERE ycsb_key >= ? LIMIT ?
  - UPDATE usertable SET data = json_set(data, '$.field5', ?) WHERE ycsb_key = ?;

YCSB Benchmark: MongoDB

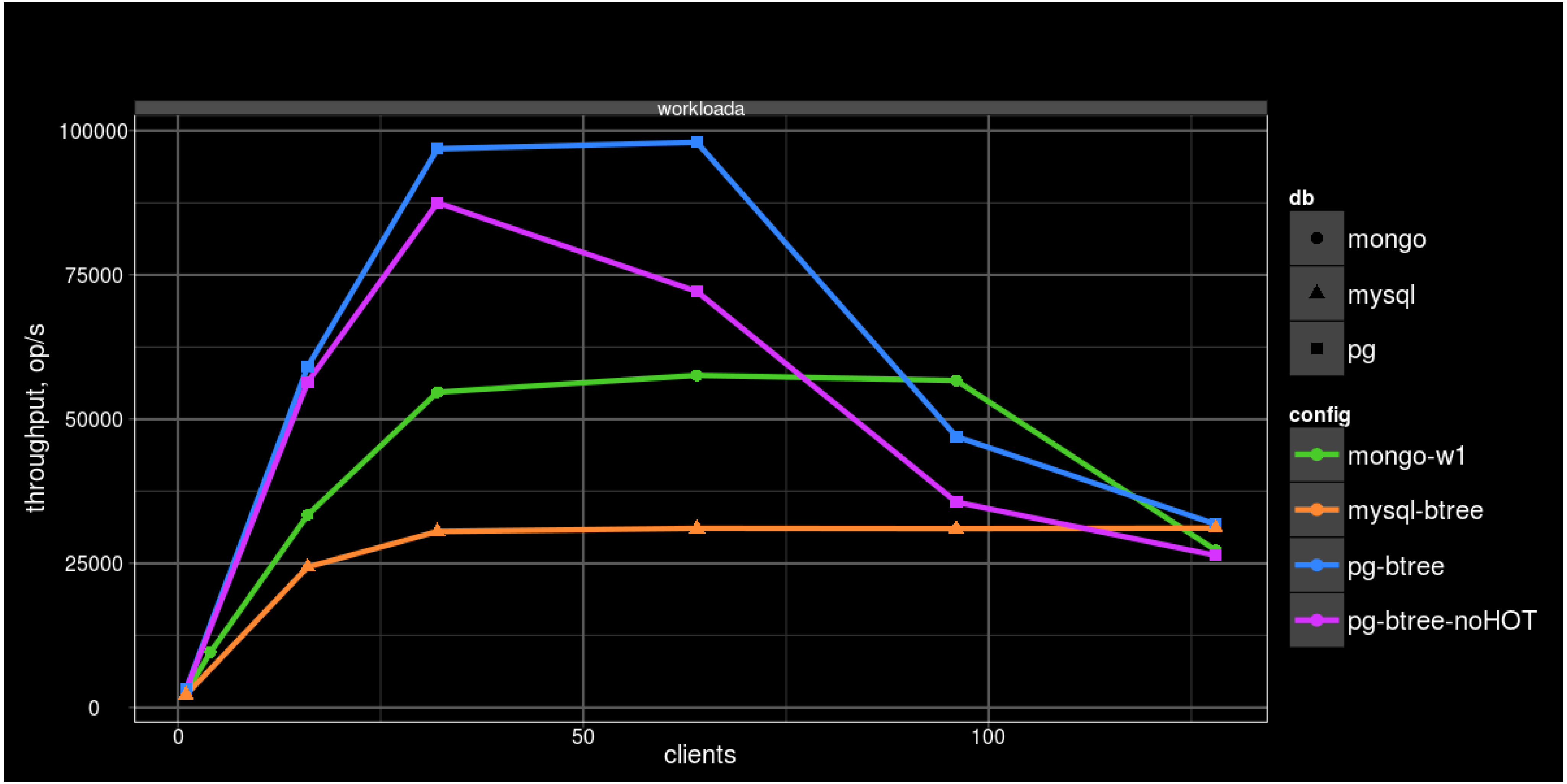
- Table

  - db.usertable.findOne({ _id: key })
  - db.usertable.findOne({ _id: key }).projection({ field5: 1 })
  - db.usertable.find({ _id: { $gte: startkey } }).sort({ _id: 1 }).limit(recordcount)
  - db.usertable.updateOne({ _id: key }, { $set: { field5: fieldval } })

HOT update for json[b]

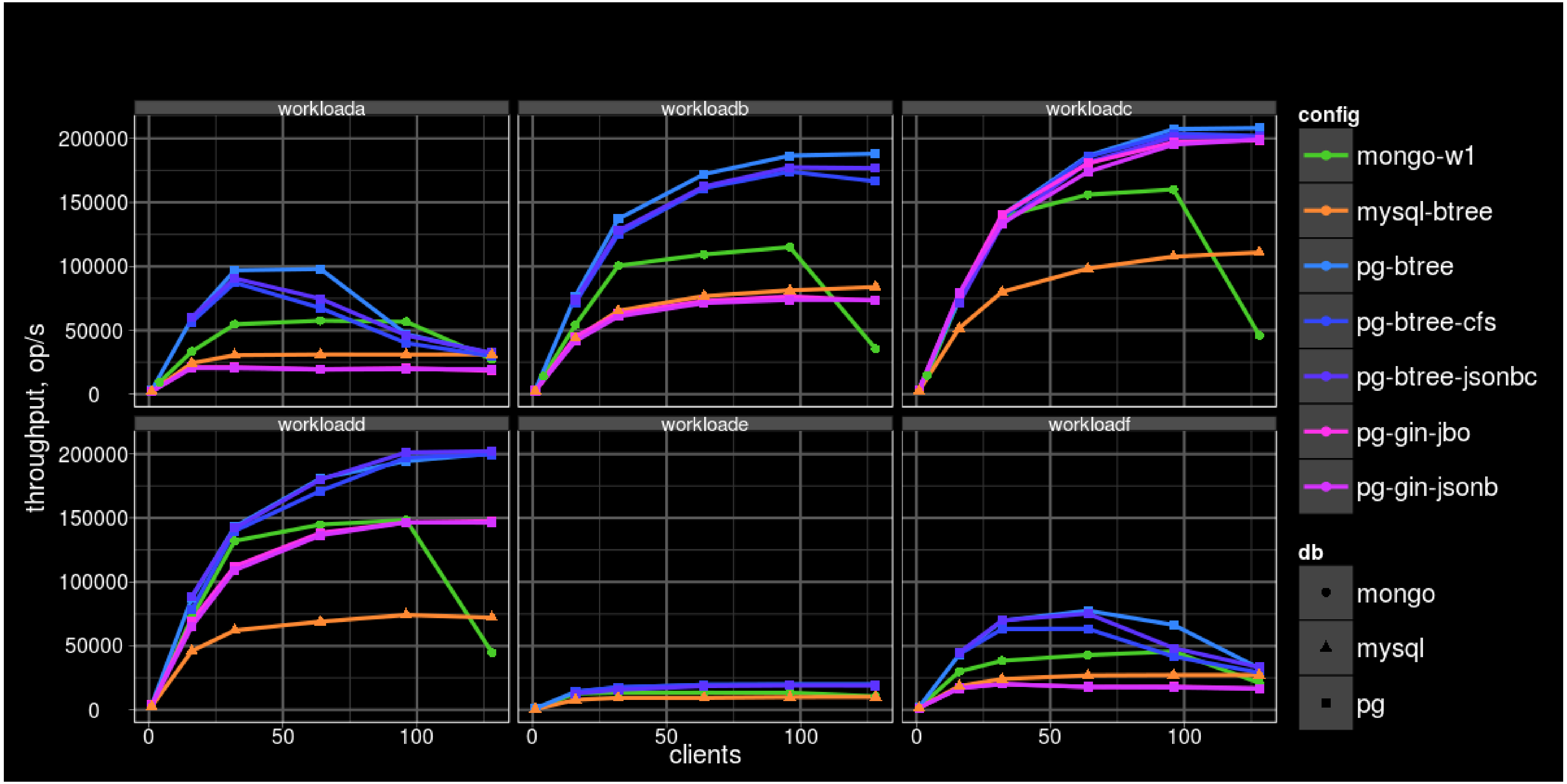
- HOT (Heap Only Tuple) — useful optimization for UPDATE performance
  - Dead tuple space can be automatically reclaimed at INSERT/UPDATE if no changes are made to indexed columns
  - New and old row versions «live» on the same page
- HOT doesn“t works well with functional indexes
  - Functional index on keyA and update keyB - (raspbeberry line)
- We fixed the problem in HeapSatisfiesHOTandKeyUpdate() and use it on all runs - (blue line)

HOT update for json[b]



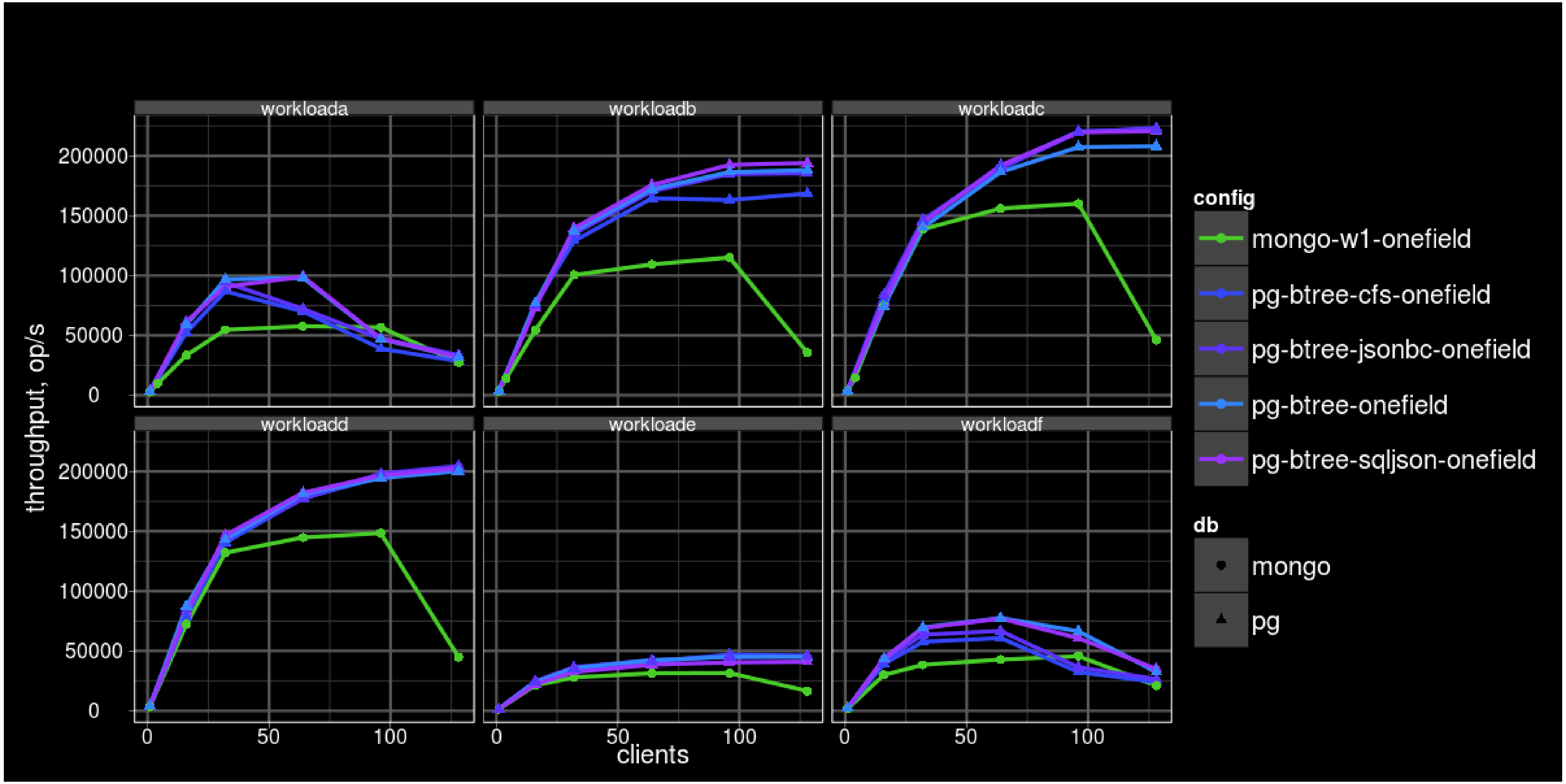
1 mln rows, 10 fields, select all fields

- Postgres is better in all workloads !
- All jsonb variants are the same for most read workloads
- Gin, jsonbc and jsonb(cfs) are not good for updates



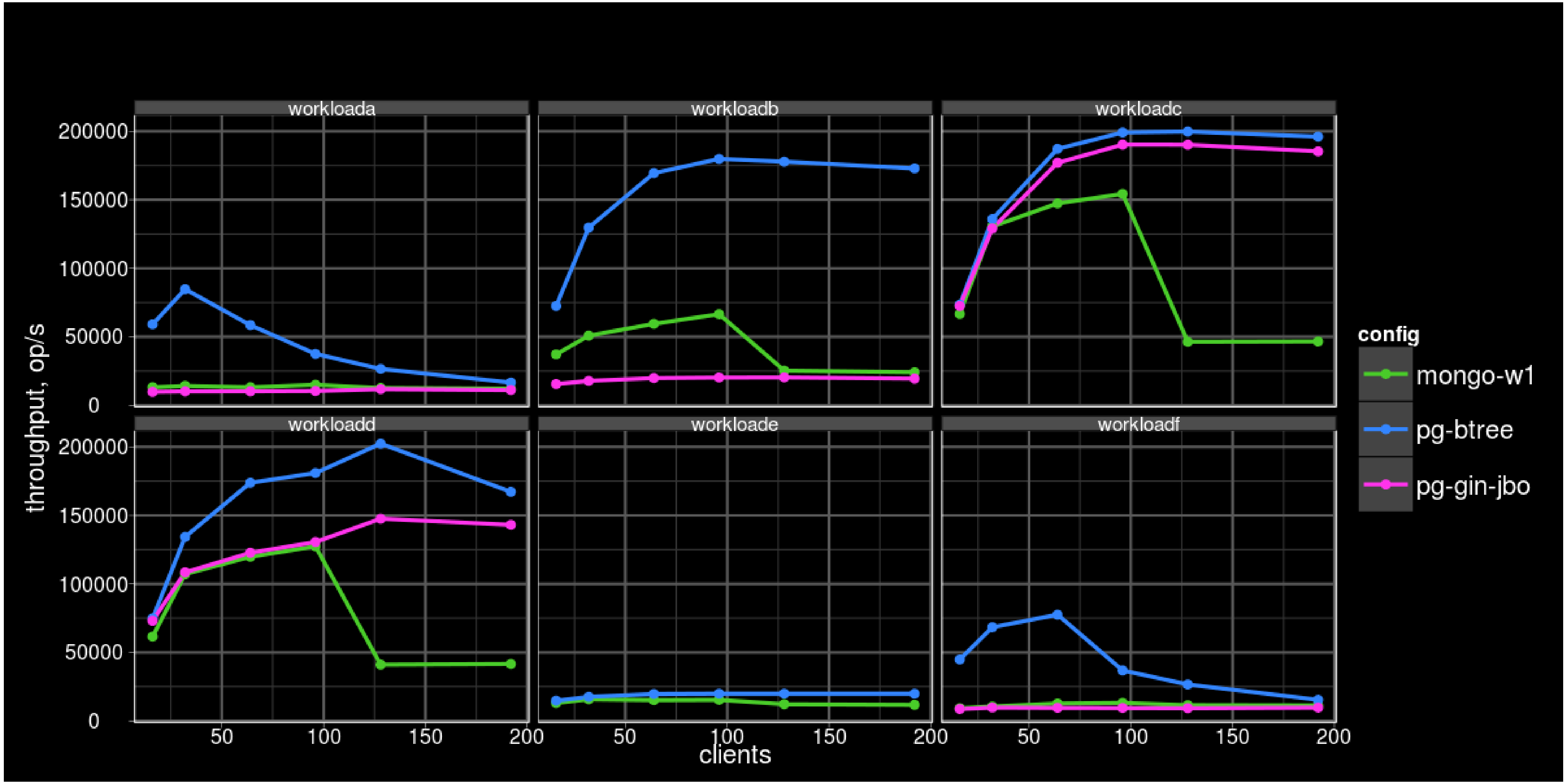
1 mln rows, 10 fields, select one fields

- Postgres is better in all workloads !
- Jsonb ~ jsonb(cfs) ~ Jsonbc ~ sqljson for most read workloads
- Jsonbc and jsonb(cfs) not good for updates



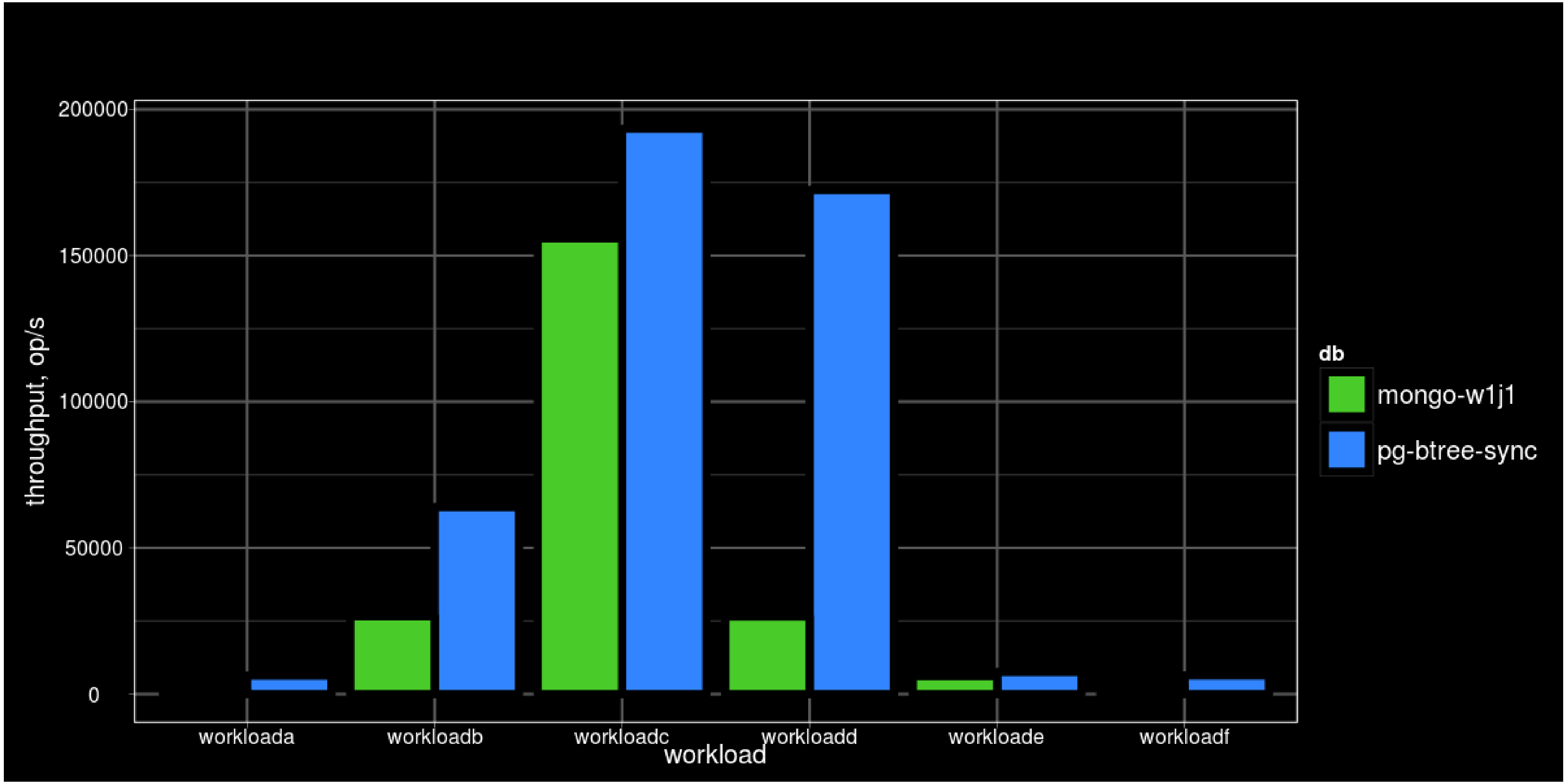
5 mln rows, 10 fields

- Postgres is better in all workloads !
- Gin is not good for updates



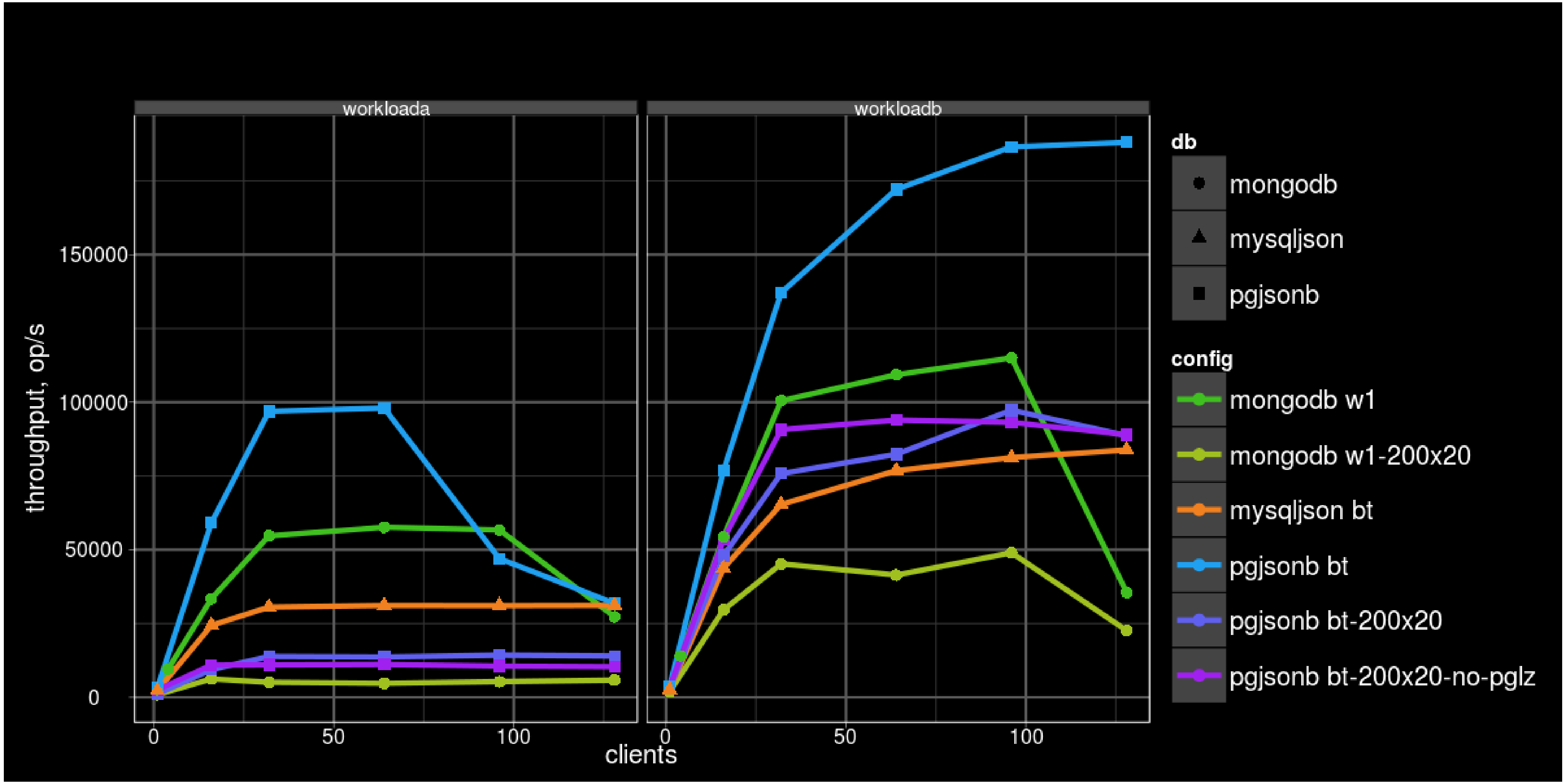
100K rows, 10 fields, journal on disk

- Mongo — j1 Postgres - async.commit is on
- Postgres is better in all workloads !



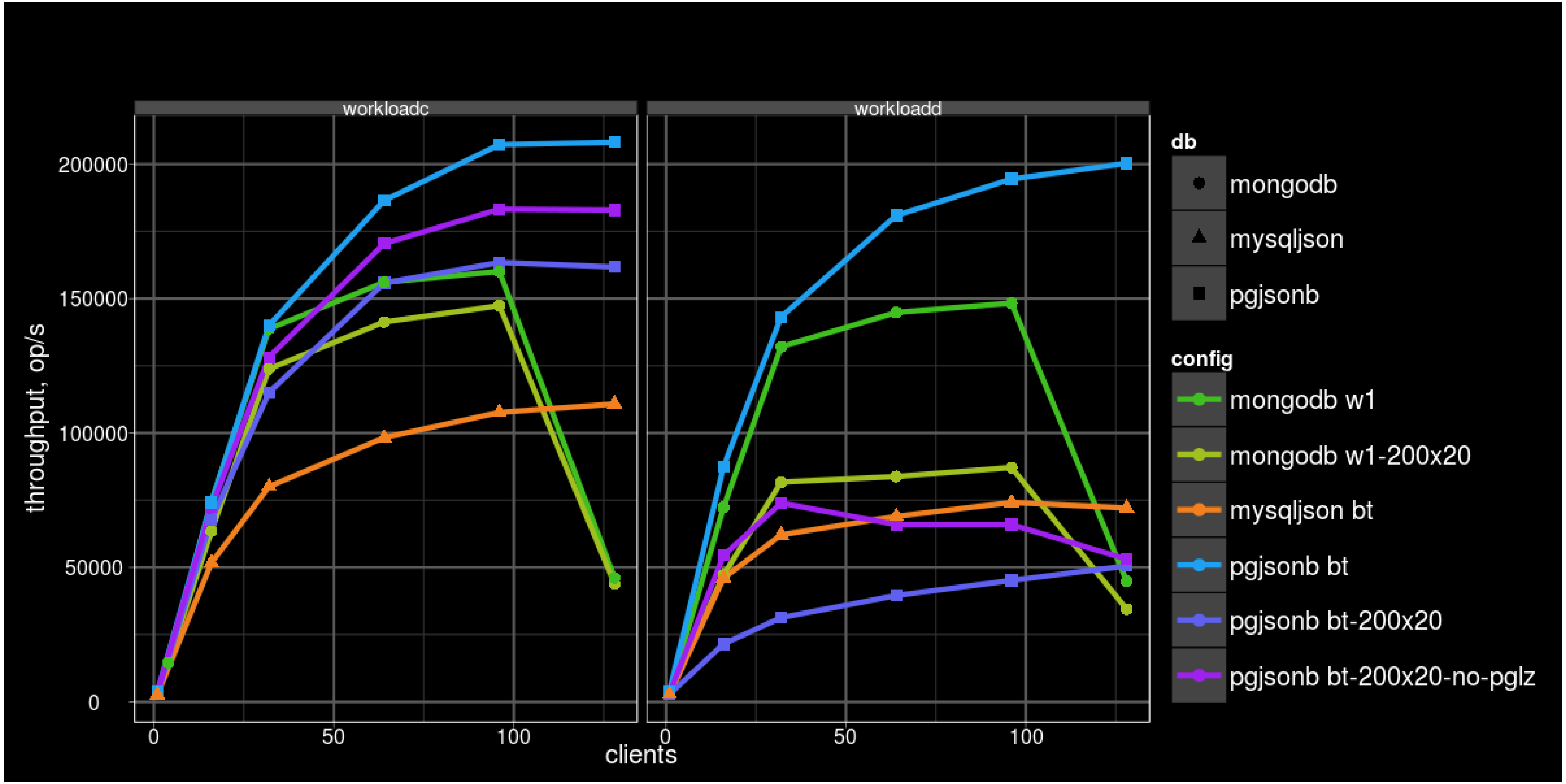
1mln rows, 200 fields, workloads a,b

- Postgres is better !



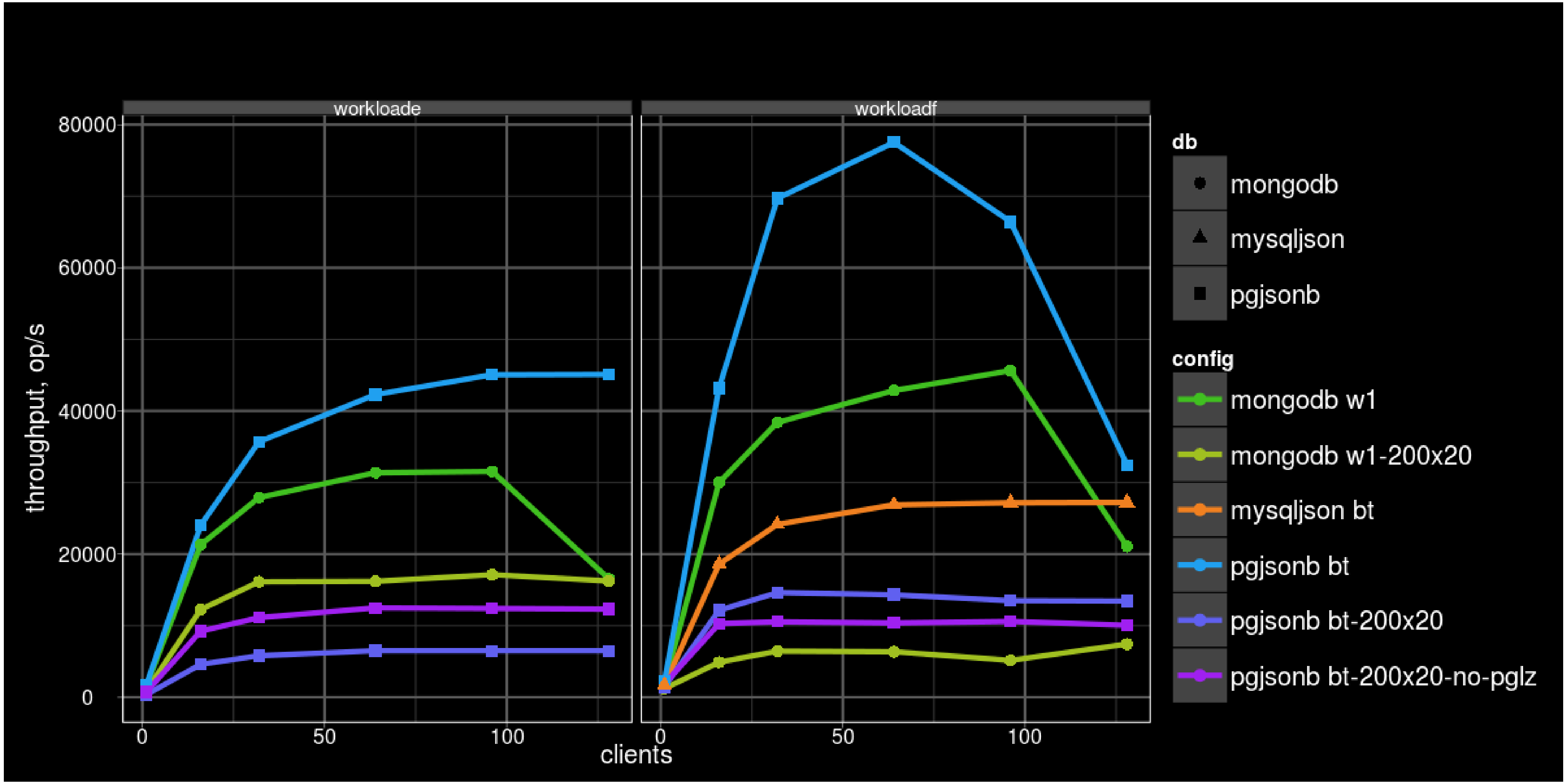
1mln rows, 200 fields, workloads c,d

- MongoDB is better in workload D
- Postgres is better in workload C



1mln rows, 200 fields, workloads e,f

- MongoDB is better in workload E
- Postgres is better in workload F

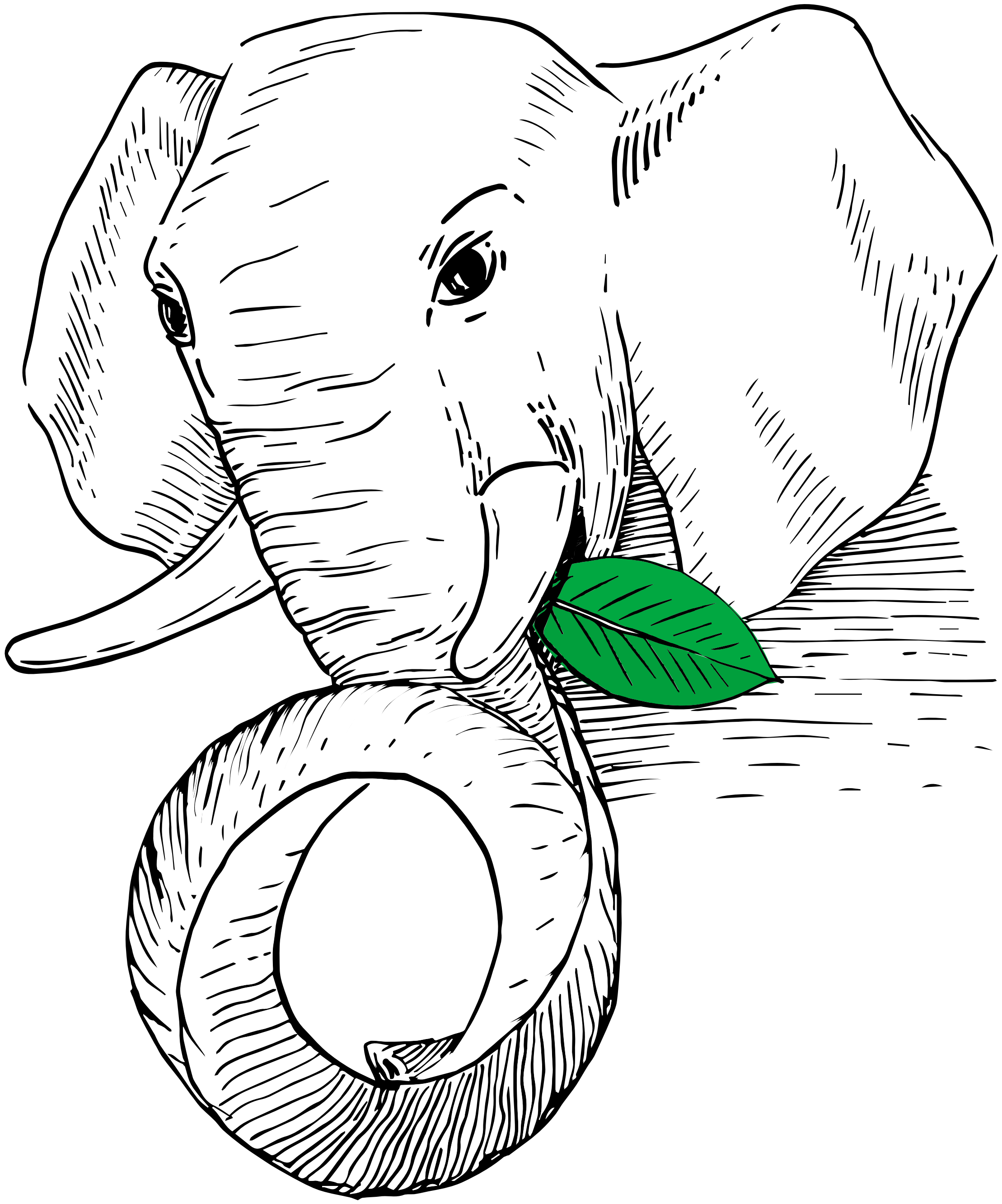


PROFESSIONAL Postgres

PostgreSQL

beats

MongoDB !



Still need more tps ?

Use partitioning

- Upcoming version of pg_pathman supports partitioning by expression
- Delicious bookmarks dataset — 5 partitions

```
SELECT pathman.create_hash_partitions('jb', 'jb->>''id''', 5);
 create_hash_partitions
------------------------
                      5
(1 row)

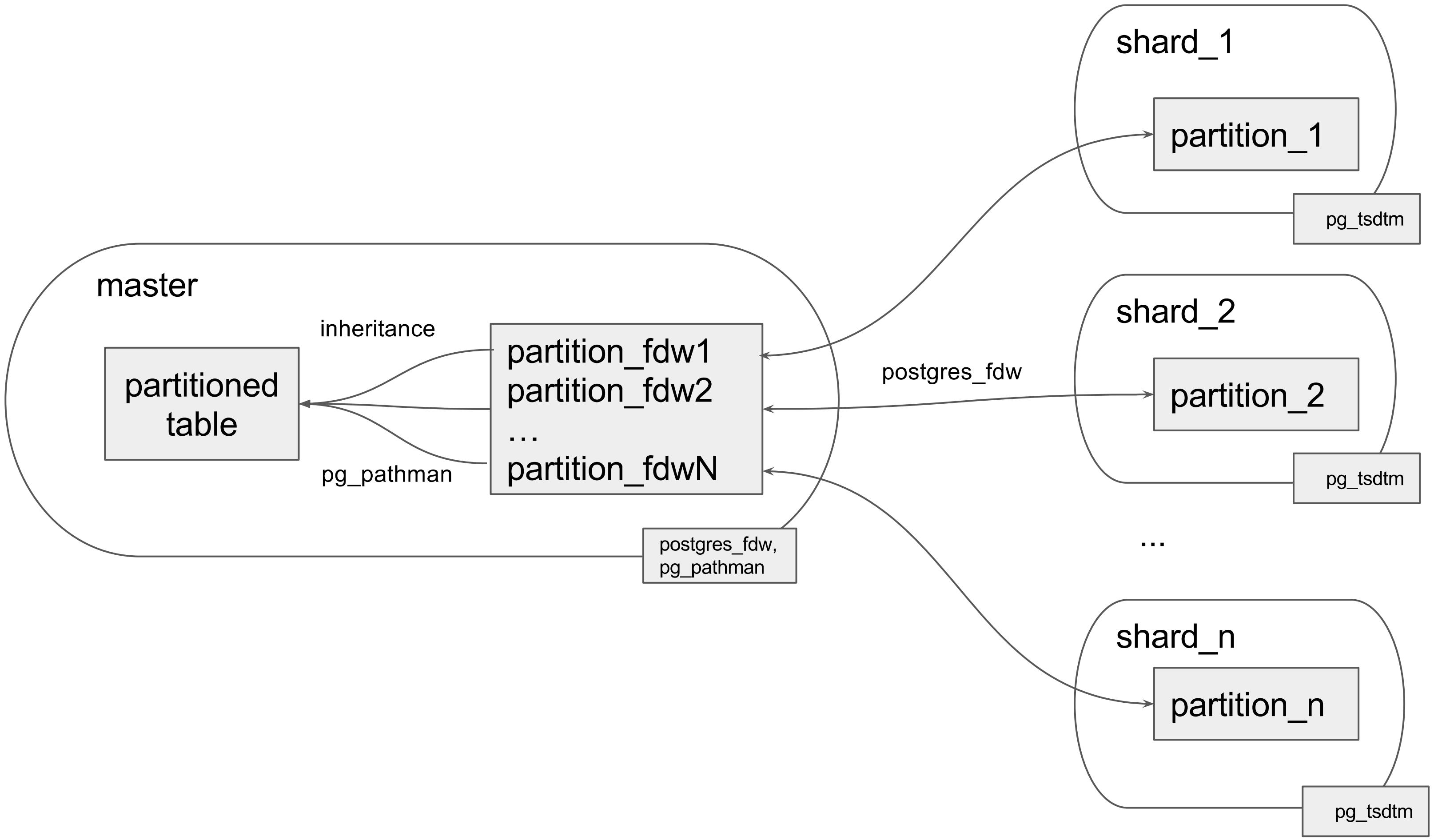
SELECT * FROM jb
WHERE (jb->>'id') = 'http://delicious.com/url/c91427110a17ad74de35eabaa296fa7a#kikodesign';
```

  - Vanilla 9.6 - 818, 274 (parallel) +pg_pathman - 173, 84 (parallel)
- Delicious bookmarks dataset — 1000 partitions
  - Vanilla 9.6 — 505 ms (27 ms) + pg_pathman — 1 ms (0.47 ms) !

Postgres PROFESSIONAL

Still need more tps ?

Use sharding !



Sharding with postgres_cluster

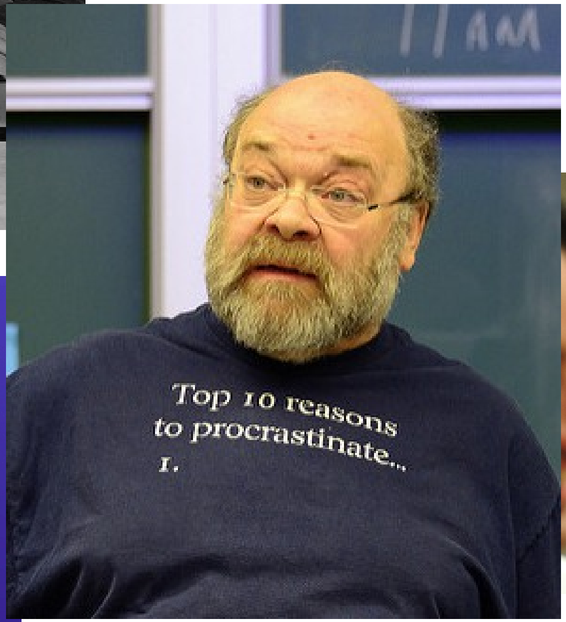
- Master: fork postgres_cluster
  https://github.com/postgrespro/postgres_cluster
- Shards: pg_tsdtm
  https://github.com/postgrespro/pg_tsdtm

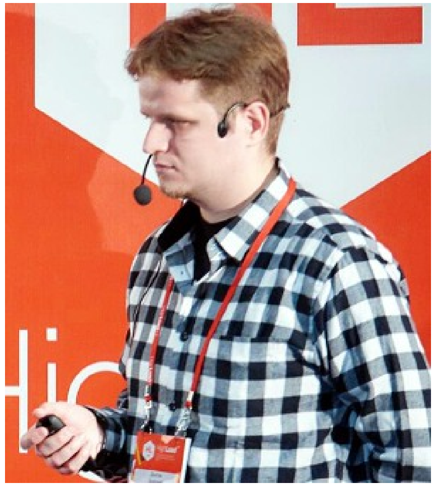
Summary

- Postgres is already a good NoSQL database + clear roadmap
- Move from NoSQL to Postgres to avoid nightmare !
- SQL/JSON will provide better flexibility and interoperability
  - Expect it in Postgres 11 (Postgres Pro 10)
  - Need community help (testing, documentation)
- JSONB dictionary compression (jsonbc) is really useful
  - Expect it in Postgres 11 (Postgres Pro 10)
- Postgres beats Mongodb and MySQL in one node configuration
  - Next: YCSB benchmarks in distributed mode

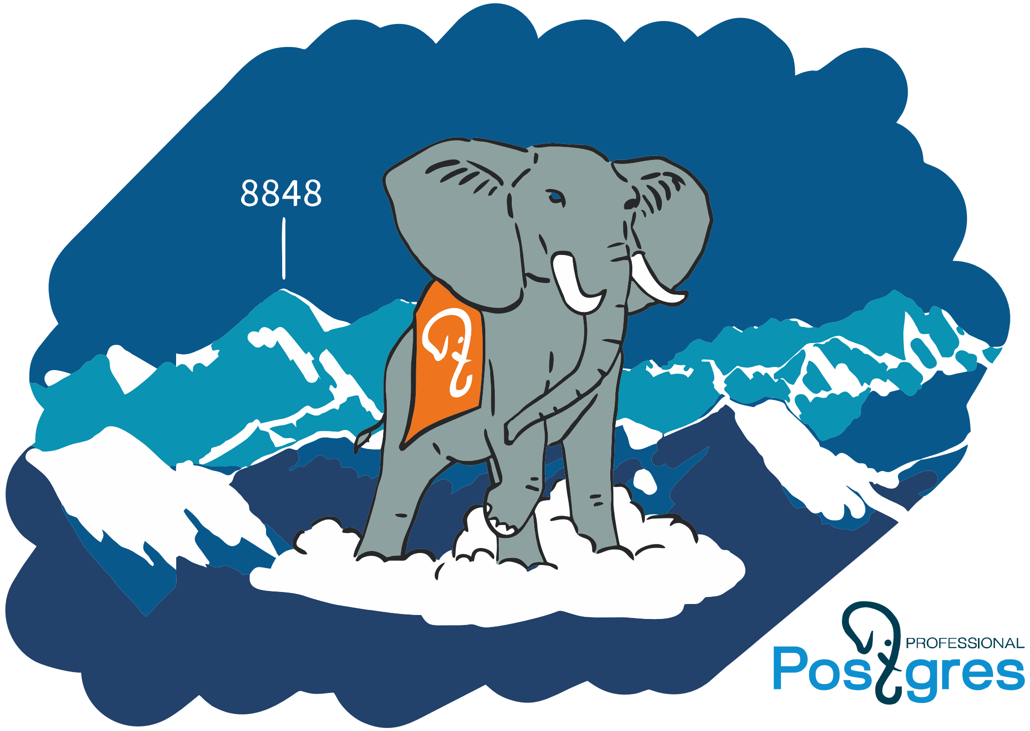
PEOPLE BEHIND JSON[B]

Nikita Glukhov





EVEREST MARATHON 2017



Thanks !