

# NoSQL Postgres

Oleg Bartunov  
Ivan Panchenko

Postgres Professional  
Moscow University



PGDay, Saint Petersburg, July 6, 2017



# NoSQL (предпосылки)

- Relational DBMS - integrational
  - All APPs communicates 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 (Volume of data, Velocity of data in-out, Variety 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 NoSQL functionality is not enough ?
- What if application needs ACID and flexibility of NoSQL ?
  
- Relational databases work with data with schema known in advance
- One of the major complaints 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 ?



# Challenge to PostgreSQL !

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





# 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 !

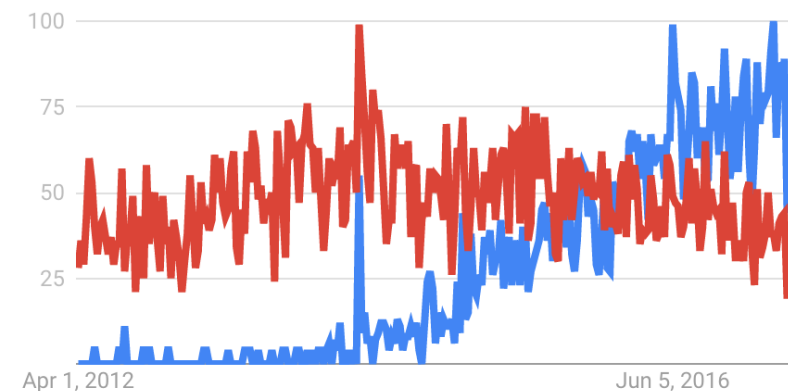


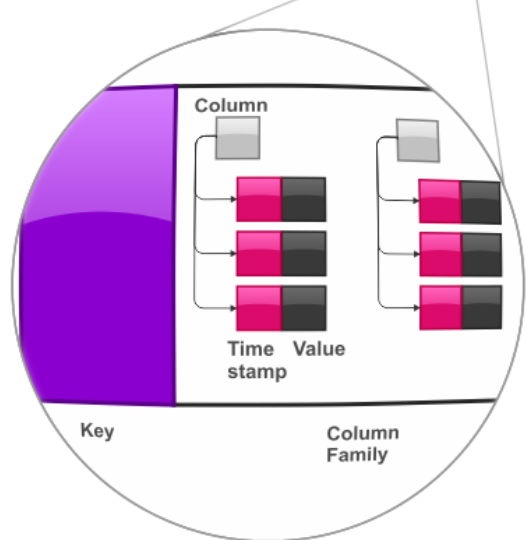
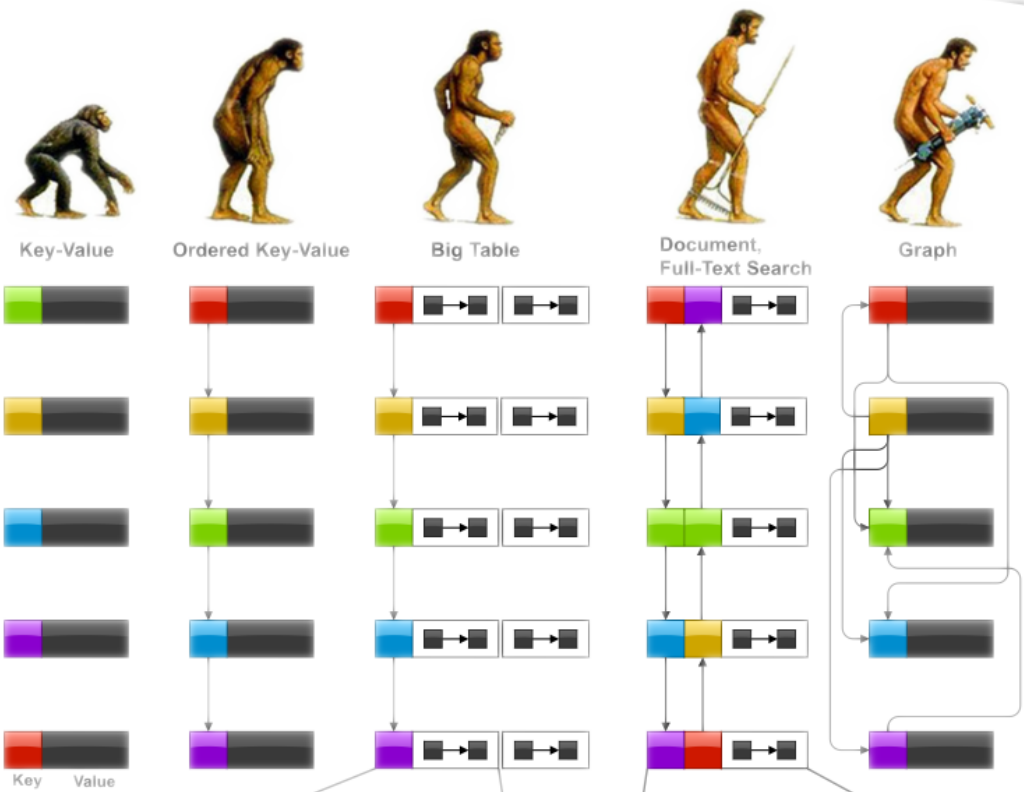
## NoSQL Postgres briefly

- 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





```

"employee" :
{
  "name" : "Mohana Pillai",
  "position" : "Delivery Manager",
  "projects" : [
    {
      "name" : "Easy Signatures"
    }
  ]
}, Semi-Structured Data
-----
Plain Text
is a confidential word or number
combination used as a code to
identify when accessing
between 8 and 15 characters
number and may not contain
spaces
  
```



JSONB - 2014

- Binary storage
- Nesting objects & arrays
- Indexing

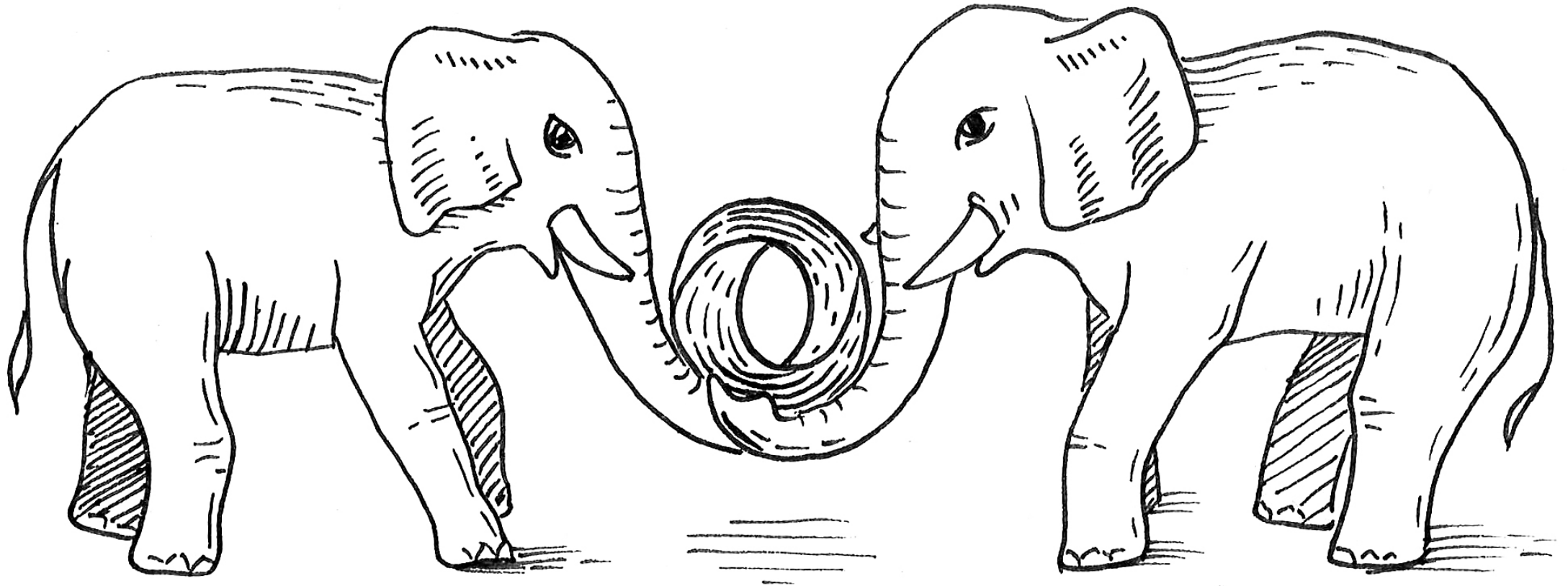
JSON - 2012

- 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;
```

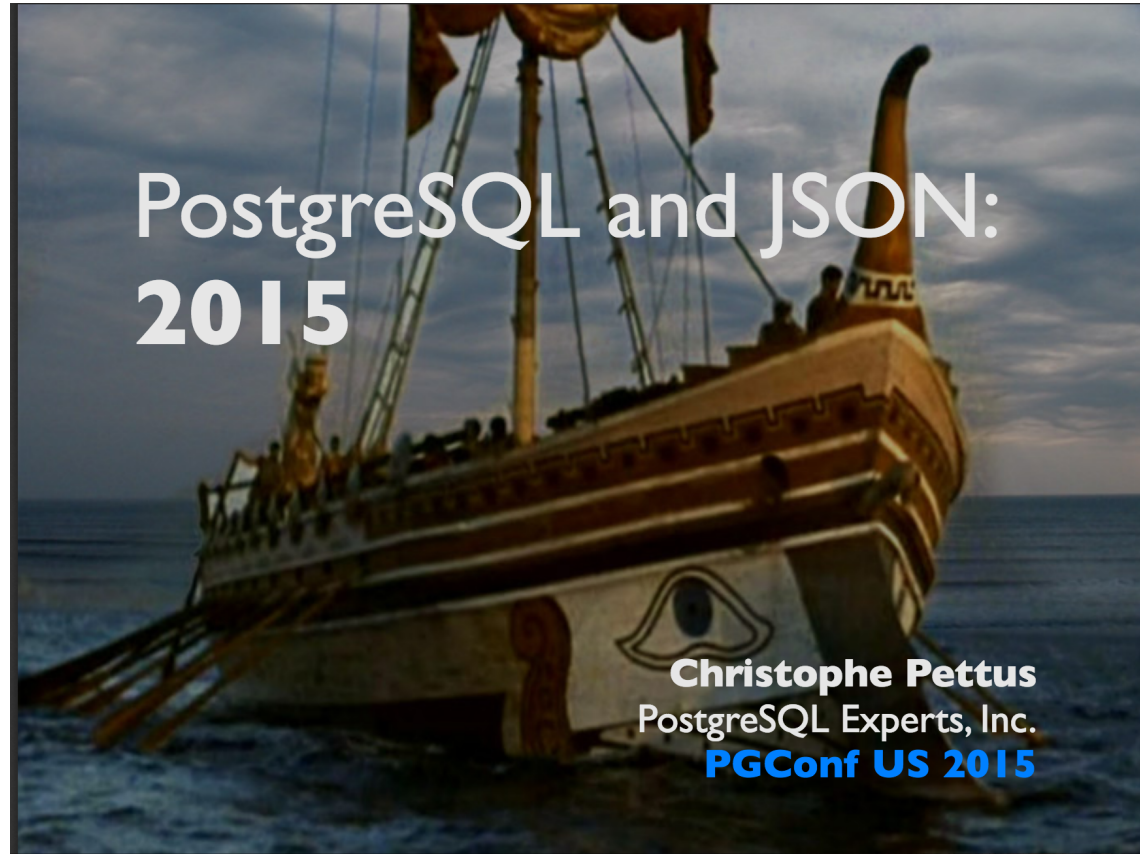
```
-----+-----
{"cc":0, "aa": 2, "aa":1,"b":1} | {"b": 1, "aa": 1, "cc": 0}
(1 row)
```

- 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 there is  
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)

# 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"';
```

<https://github.com/postgrespro/jquery>

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

4.46	<b>JSON</b> data handling in SQL. ....	174
4.46.1	Introduction. ....	174
4.46.2	Implied JSON data model. ....	175
4.46.3	SQL/JSON data model. ....	176
4.46.4	SQL/JSON functions. ....	177
4.46.5	Overview of SQL/JSON path language. ....	178
<b>5</b>	<b>Lexical elements. ....</b>	<b>181</b>
5.1	<SQL terminal character>. ....	181
5.2	<token> and <separator>. ....	185



## 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: 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]')  
);
```

# 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

```

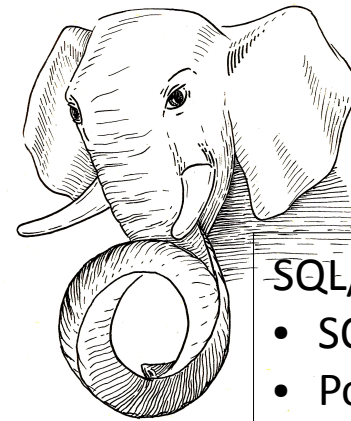
SELECT * FROM js_test WHERE
JSON_EXISTS( value, '$.**.color
? (@ == "red")');

```



# SQL/JSON availability

- Github Postgres Professional repository  
<https://github.com/postgrespro/sqljson>
- SQL/JSON examples
- WEB-interface to play with SQL/JSON
- Technical Report (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



# JSONB COMPRESSION

Transparent compression of jsonb

+ access to the child elements without full decompression





## 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



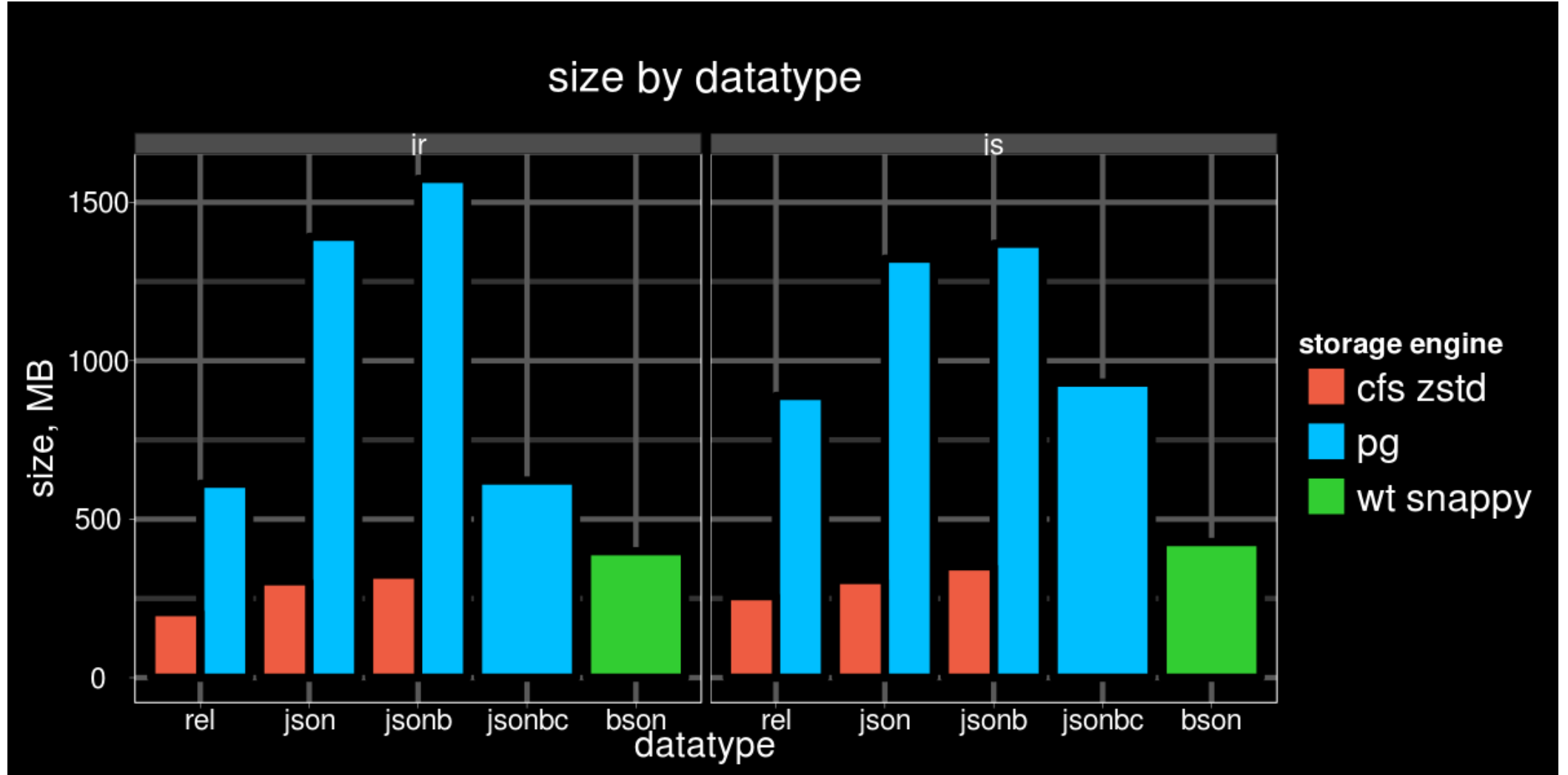
# jsonb compression: results

Two datasets:

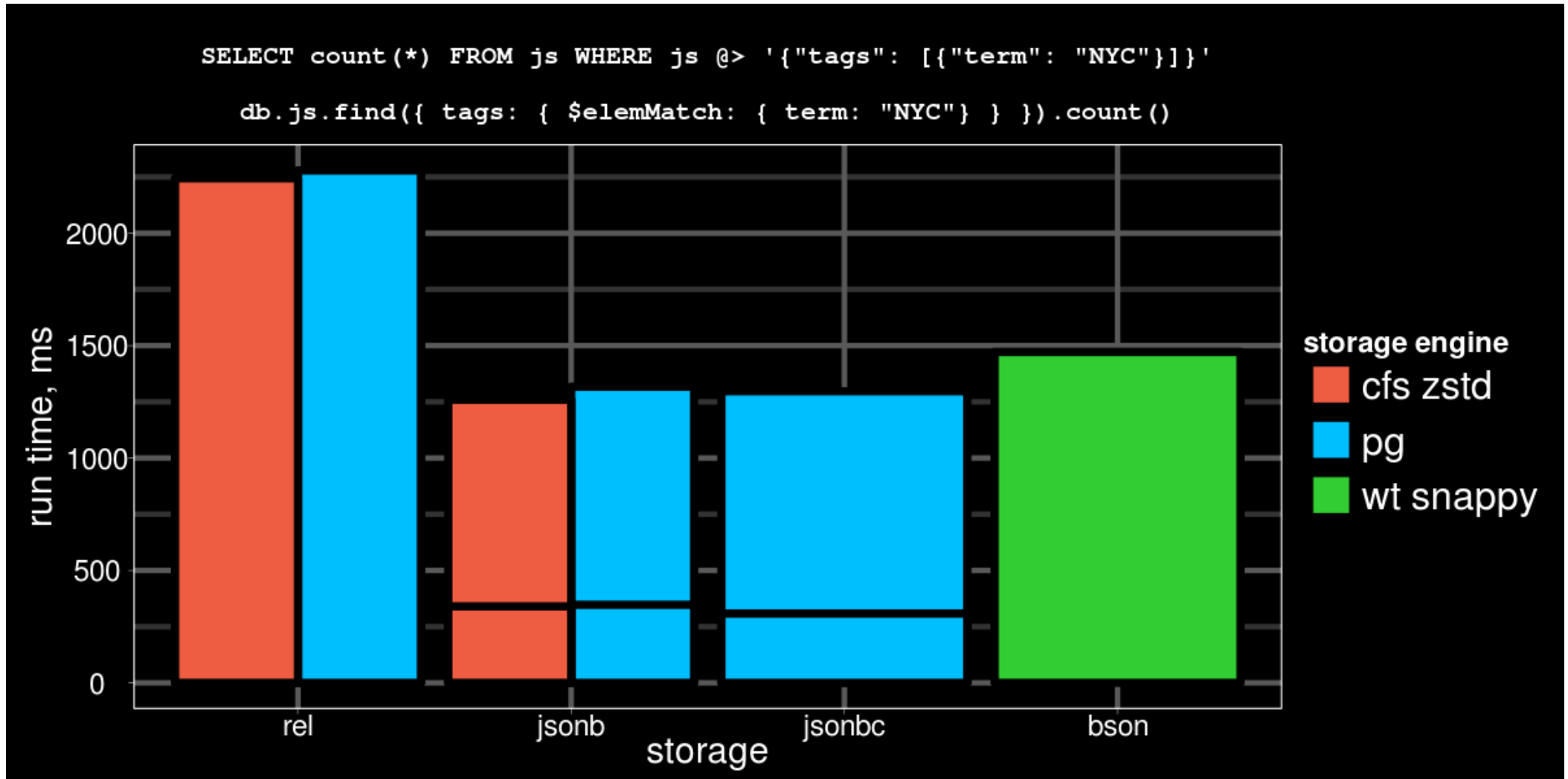
- js – Delicious bookmarks, 1.2 mln rows (js.dump.gz)
  - Mostly string values
  - Relatively short keys
  - 2 arrays (tags and links) of 3-field objects
- jr – customer reviews data from Amazon, 3mln (jr.dump.gz)
  - Rather long keys
  - A lot of short integer numbers

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

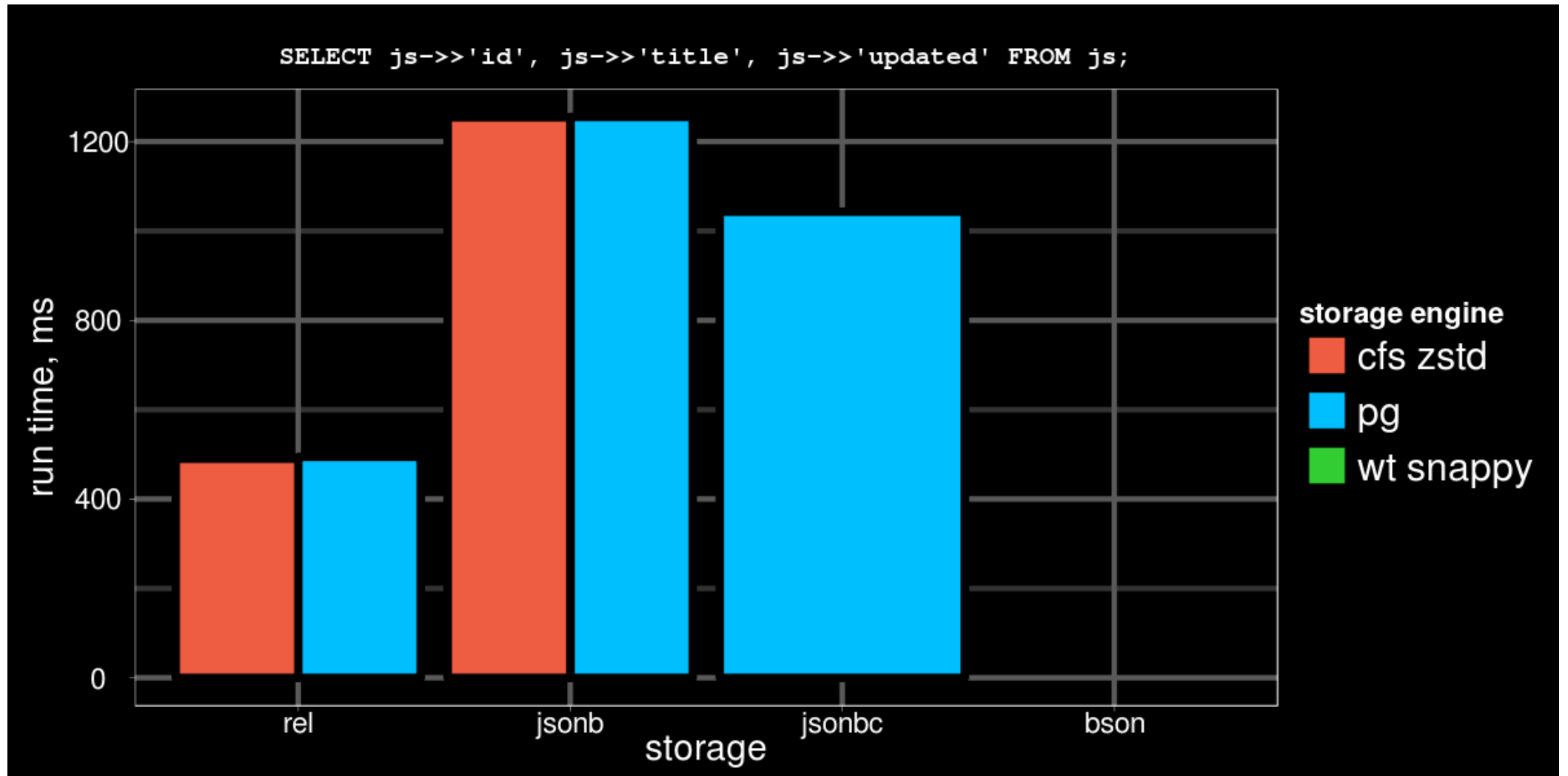
# jsonb compression: table size



# jsonb compression (js): performance

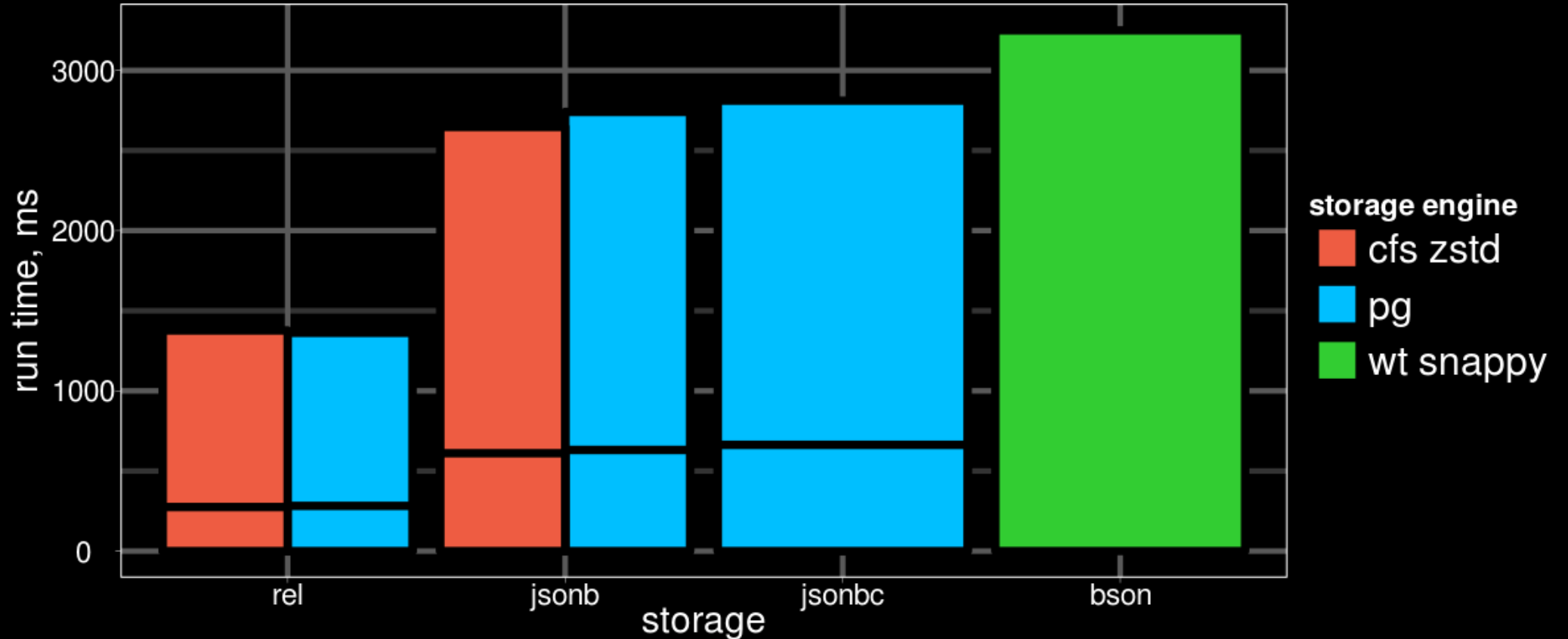


# jsonb compression (js): performance



# 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: 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

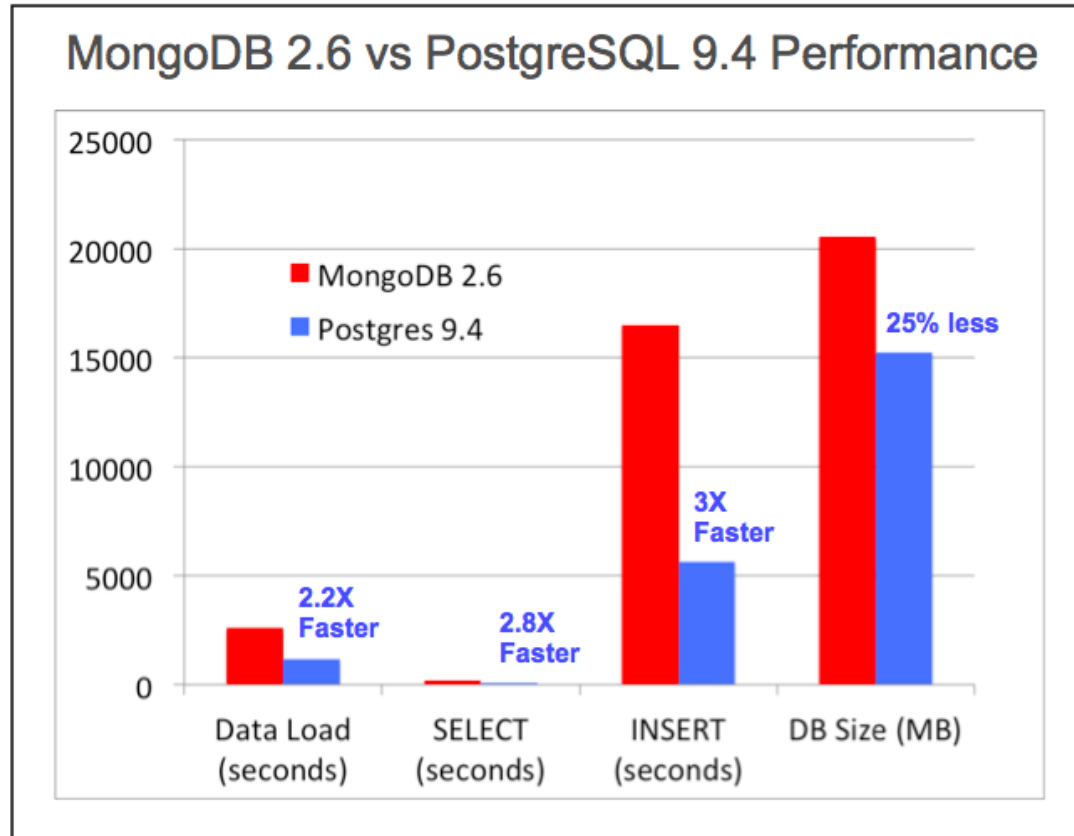
- postgres : 1.3Gb
- mongo : 1.8Gb

- Input performance:

- Text : 34 s
- Json : 37 s
- Jsonb : 43 s
- mongo : 13 m



# EDB NoSQL Benchmark



[https://github.com/EnterpriseDB/pg\\_nosql\\_benchmark](https://github.com/EnterpriseDB/pg_nosql_benchmark)

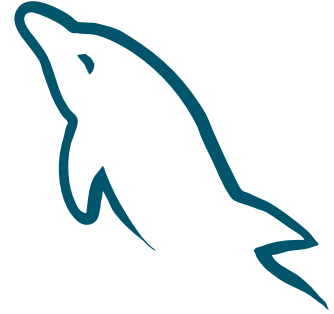
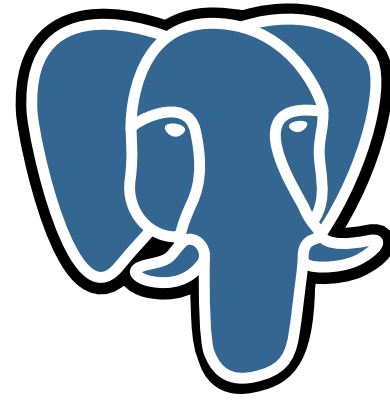


# Benchmarking NoSQL Postgres

- 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/yccb.pdf>
- We run YCBS for Postgres master, MongoDB 3.4.2, Mysql 5.7.18
  - 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

- Postgres (master), asynchronous commit=on  
Mongodb 3.4.2 (w1, j0) — 1 mln. rows
- Postgres ( master), asynchronous commit=off  
Mongodb 3.4.2 (w1, j1) — 100K rows
- MySQL 5.7.18 + all optimization (by Alexey Kopytov)
- We tested:
  - Functional btree index for jsonb, jsonbc, sqljson, cfs (compressed) storage
  - Mongodb: WiredTiger without compression
  - Return a whole json, just one field, small range
  - 10 fields, 200 fields (TOASTed)



# YCSB Benchmark: PostgreSQL

- Table:

```
CREATE TABLE usertable(data jsonb);
```

- 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

- `shared_buffers = 20GB` # min 128kB
- `temp_buffers = 512MB` # min 800kB
- `work_mem = 512MB` # min 64kB
- `dynamic_shared_memory_type = posix` # the default is the first option
  
- `synchronous_commit = off` # synchronization level;
- `commit_delay = 10` # range 0-100000, in microseconds
  
- `full_page_writes = off` # recover from partial page writes
- `wal_level = minimal` # minimal, replica, or logical
  
- `bgwriter_delay = 10ms` # 10-10000ms between rounds
- `bgwriter_lru_maxpages = 400` # 0-1000 max buffers written/round
- `bgwriter_lru_multiplier = 8.0` # 0-10.0 multiplier on buffers scanned/round
- `effective_io_concurrency = 4` # 1-1000; 0 disables prefetching



# YCSB Benchmark: PostgreSQL

- `log_autovacuum_min_duration = 0` # -1 disables, 0 logs all actions and
- `autovacuum_max_workers = 8` # max number of autovacuum subprocesses
- `autovacuum_naptime = 10s` # time between autovacuum runs
- `autovacuum_vacuum_scale_factor = 0.1` # fraction of table size before vacuum
- `autovacuum_vacuum_cost_delay = 0ms` # default vacuum cost delay for
- `autovacuum_vacuum_cost_limit = 10000` # default vacuum cost limit for
  
- `checkpoint_timeout = 60min` # range 30s-1d
- `max_wal_size = 8GB`
- `min_wal_size = 1GB`
- `checkpoint_completion_target = 0.9` # checkpoint target duration, 0.0 - 1.0
- `checkpoint_flush_after = 0` # measured in pages, 0 disables
  
- `max_wal_senders = 0` # max number of walsender processes



# YCSB Benchmark: MySQL

- Table

```
CREATE TABLE usertable(  
  data json,  
  ycsb_key CHAR(255) GENERATED ALWAYS AS (data->>'$.YCSB_KEY'),  
  STORED PRIMARY 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: MySQL

```
# general
table_open_cache = 1000
table_open_cache_instances=16
back_log=1500
query_cache_type=0
max_connections=4000
skip-name-resolve=1

# files
innodb_file_per_table
innodb_log_file_size=8G
innodb_log_files_in_group=2
#innodb_open_files=1000

# buffers
innodb_buffer_pool_size=16G
innodb_buffer_pool_instances=1
innodb_log_buffer_size=256M

# tune
innodb_checksum_algorithm=none
innodb_doublewrite=0
innodb_thread_concurrency=64
innodb_flush_log_at_trx_commit=0
innodb_flush_method=0_DIRECT_NO_FSYNC
innodb_max_dirty_pages_pct=90
innodb_max_dirty_pages_pct_lwm=10
innodb_numa_interleave=1

innodb_page_cleaners=16
innodb_use_native_aio=1
innodb_stats_auto_recalc=0
innodb_stats_persistent = 1

innodb_change_buffering=none

# perf special
innodb_adaptive_flushing = 1
innodb_adaptive_flushing_lwm=1
innodb_flush_neighbors = 1
innodb_io_capacity=1000
innodb_io_capacity_max=20000

# purge
innodb_max_purge_lag_delay=3000000
innodb_max_purge_lag=2000
innodb_adaptive_hash_index=1

# monitoring
innodb_monitor_enable = '%'
performance_schema=OFF

transaction_isolation=READ-COMMITTED
```



# 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 } })`

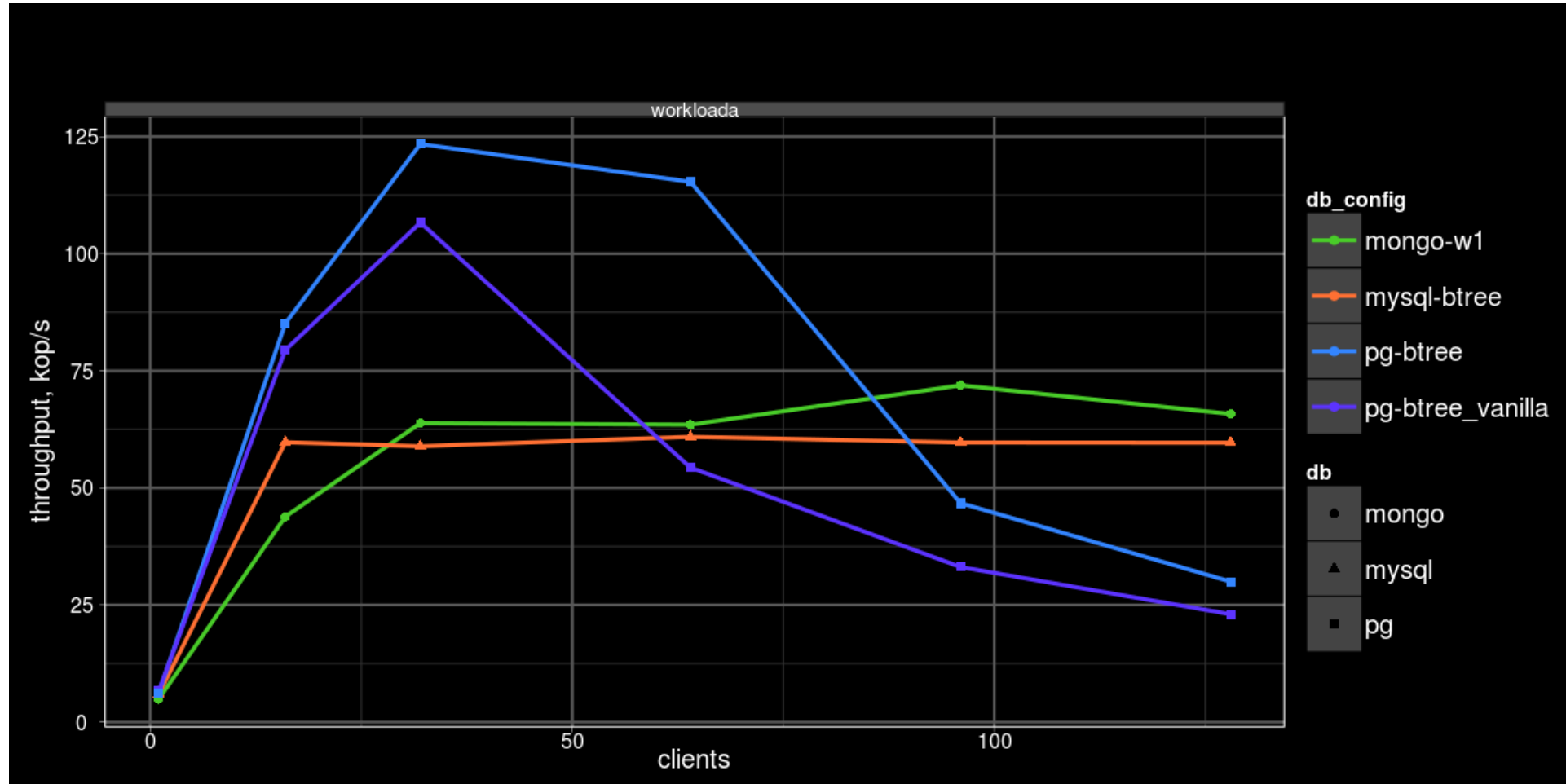


## HOT update for json[b]

- 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 does not work well with functional indexes
  - Functional index on keyA and update keyB - (raspberry line)
- We fixed the problem in `HeapSatisfiesHOTandKeyUpdate()` and use it on all runs - (blue line)

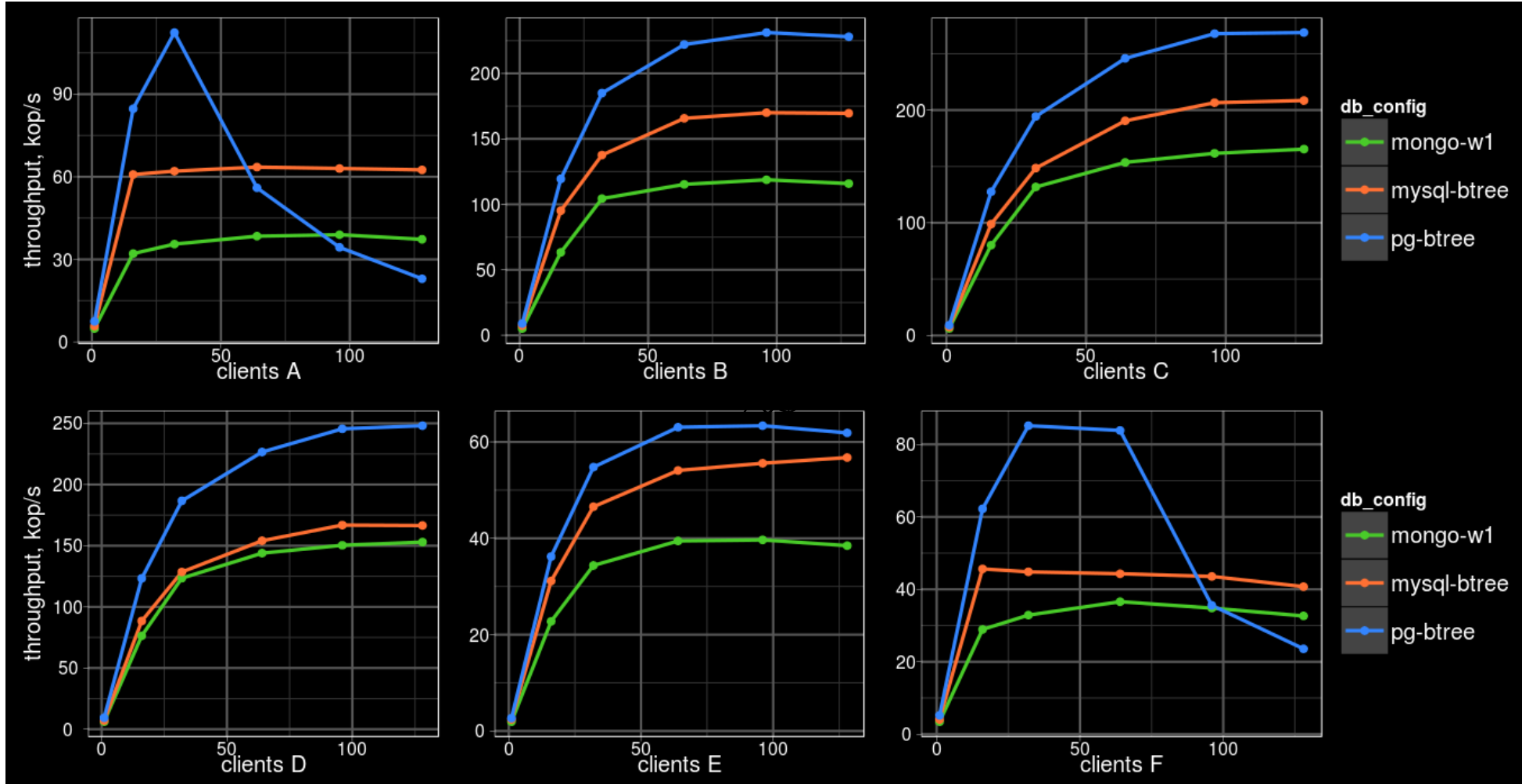


# HOT update for json[b]



# 1 mln rows, 10 fields, select 1 key

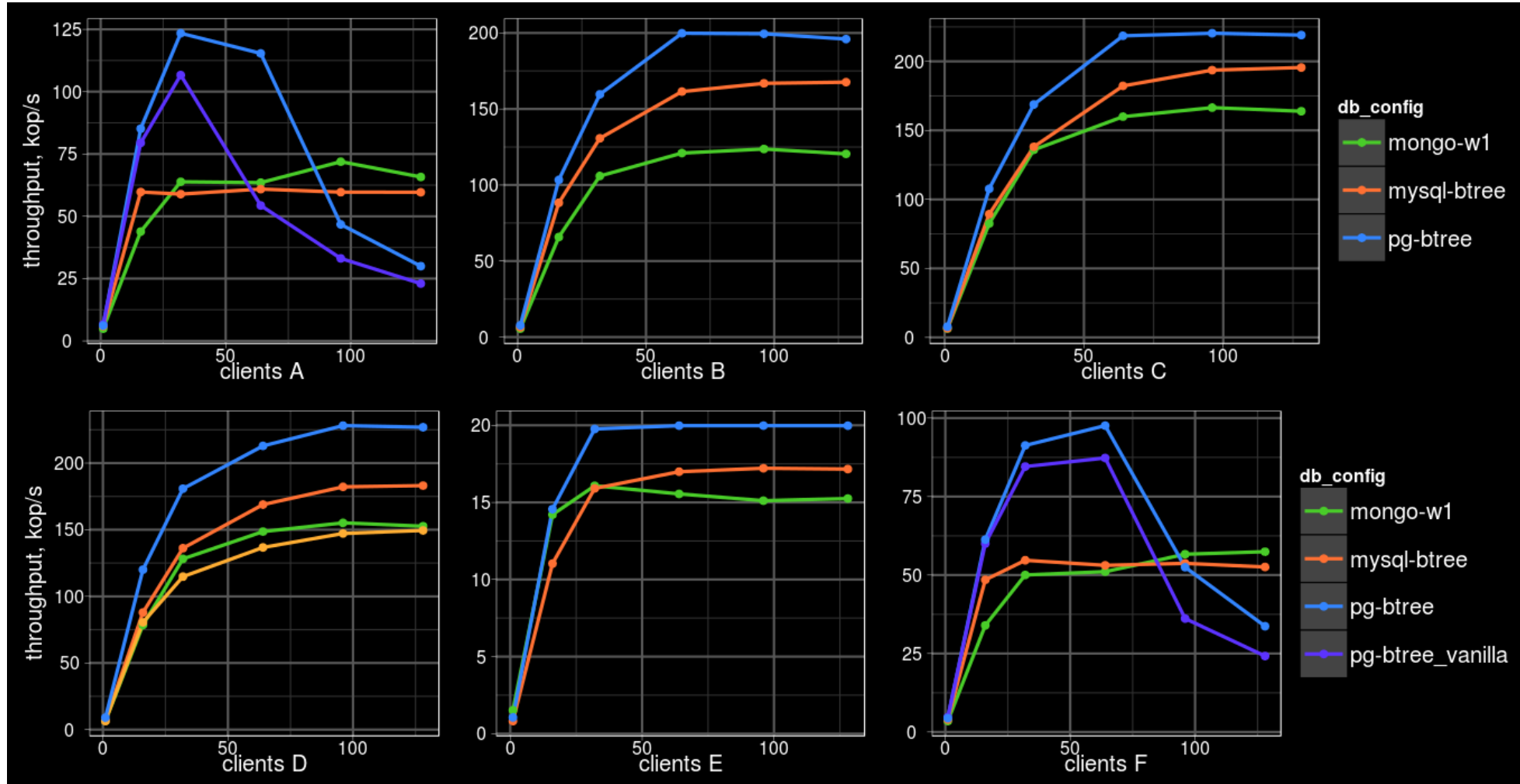
- Postgres is better in all R/O workloads
- Postgres is not scaling well for heavy R/W workloads (a,f)





# 1 mln rows, 10 fields, select all keys

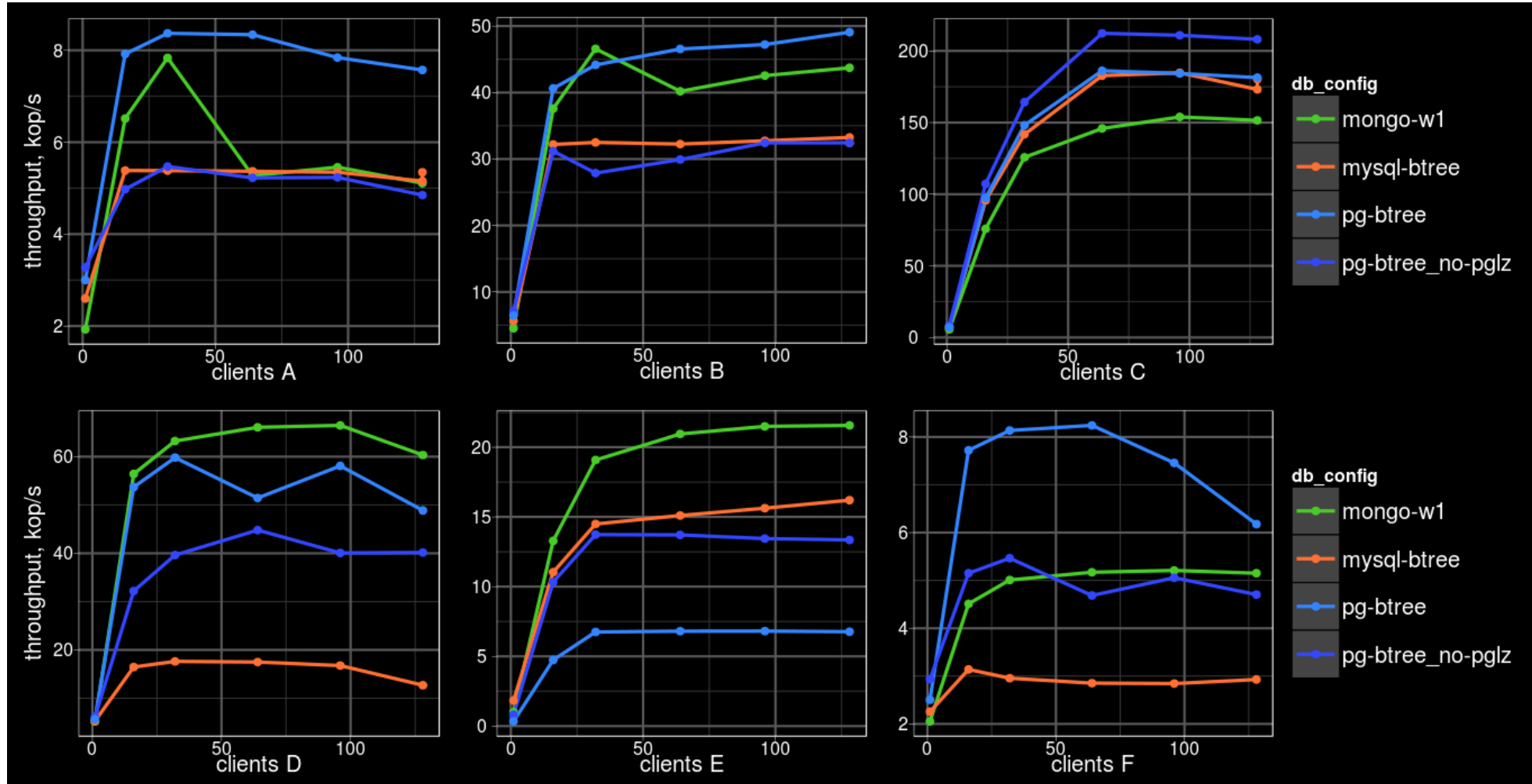
- Postgres is better in all R/O workloads
- Postgres isn't scaling well for heavy R/W workloads (a,f)





# 1mln rows, 200 fields, select 1 key

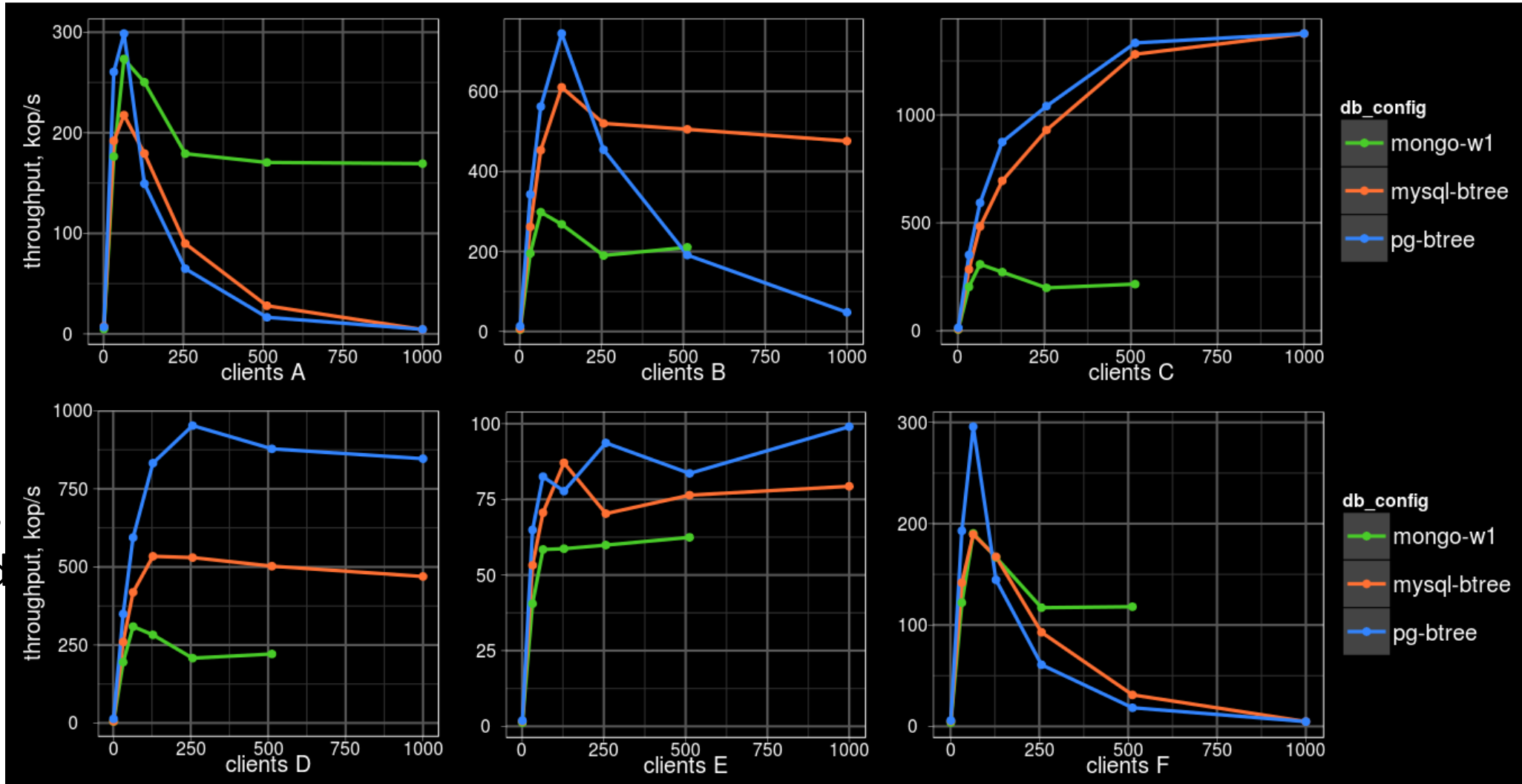
- TOASTed json are really bad
- Mongo win in D,E workloads





# 1 mln rows, 10 fields, select 1 key BIG 144 cores, 3TB ram, 2 Tb SSD

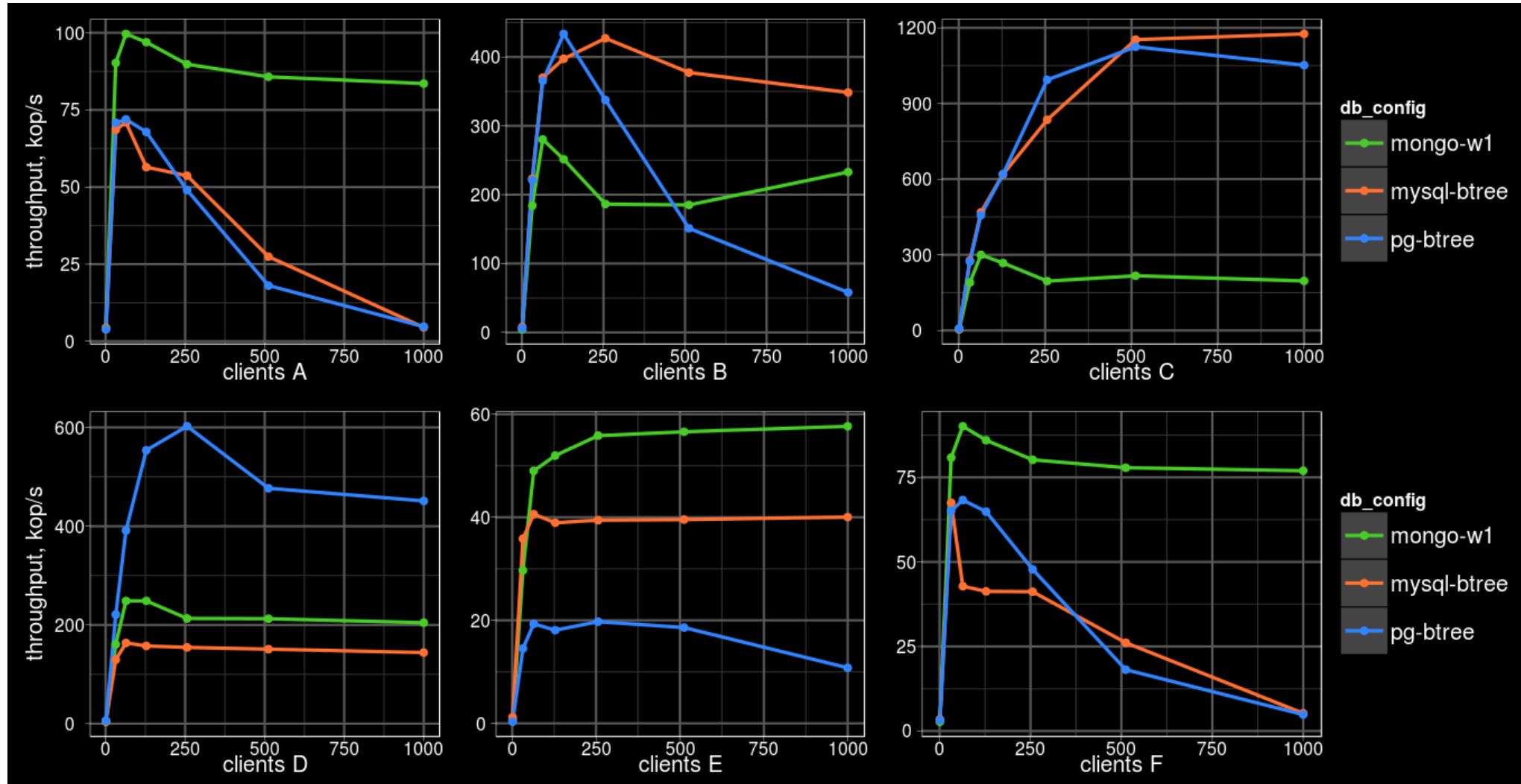
- Postgres and MySQL better use multiple cores (1.5 mln ops/sec !)
- Postgres not scaled well in R/W workloads (huge overhead in isolations)





# 1 mln rows, 200 fields, select 1 key BIG 144 cores, 3TB ram, 2 Tb SSD

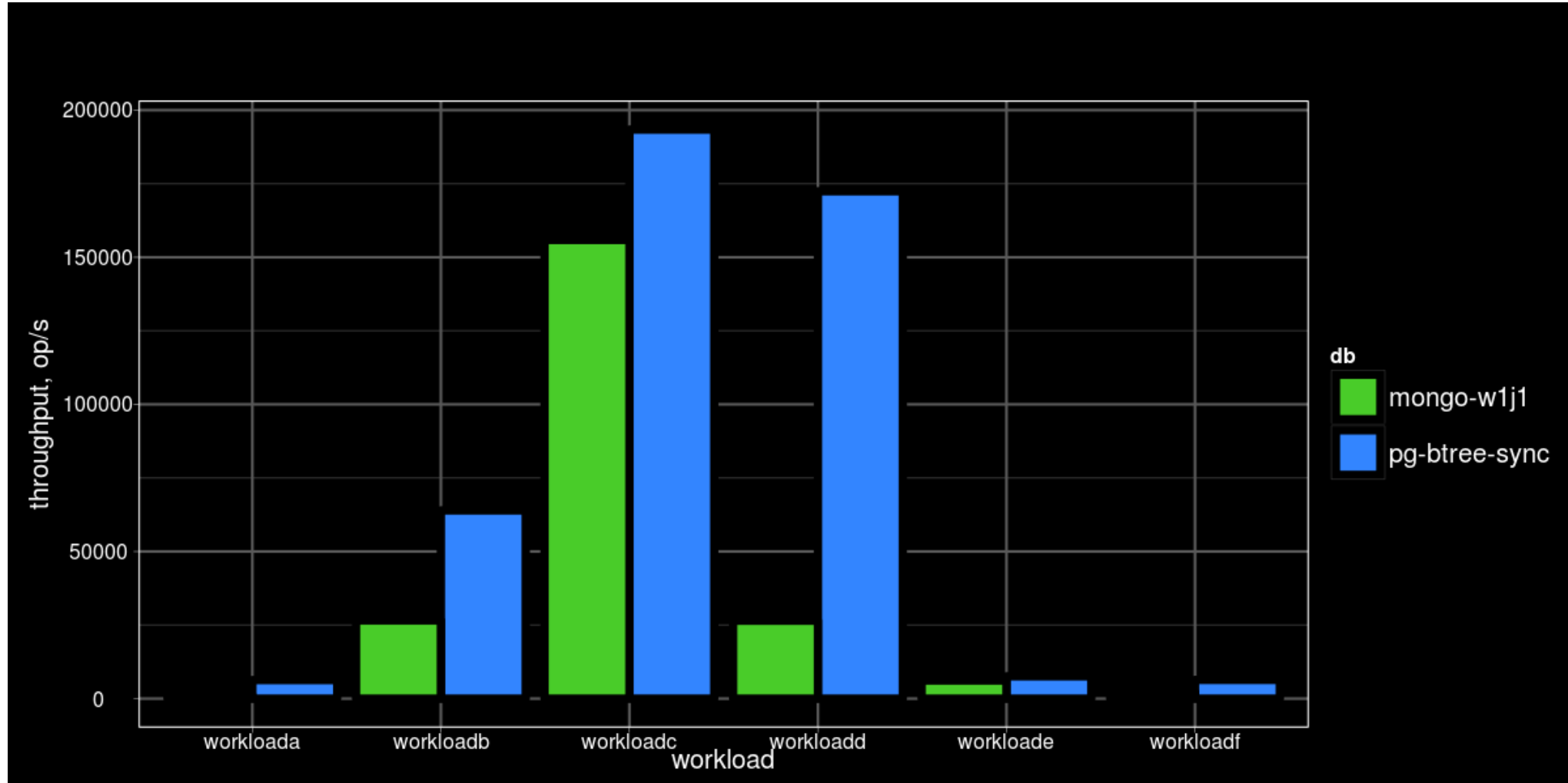
- MongoDB wins on A, E, F workloads !  
One writer is better for Zipf distribution.





100K rows, 10 fields, 64 clients

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





## Summary

Low durability: `synchronous_commit=off`, j0

- Postgres and MySQL scale well on R/O workloads
- Postgres has inefficient transactions locking (isolation) on high contention (zipf distribution and large number of clients)
- PGLZ in TOAST is cpu-hungry, range queries (workload E) suffer.
- MySQL is better than Postgres on R/W (zipf distribution and large number of clients), especially in workload B (5% update).
- Mongo does not degrade on R/W with high contention, especially on long json (one writer helps).
- Postgres (`synchronous_commit=on`) beats Mongo if durability enabled (j1).

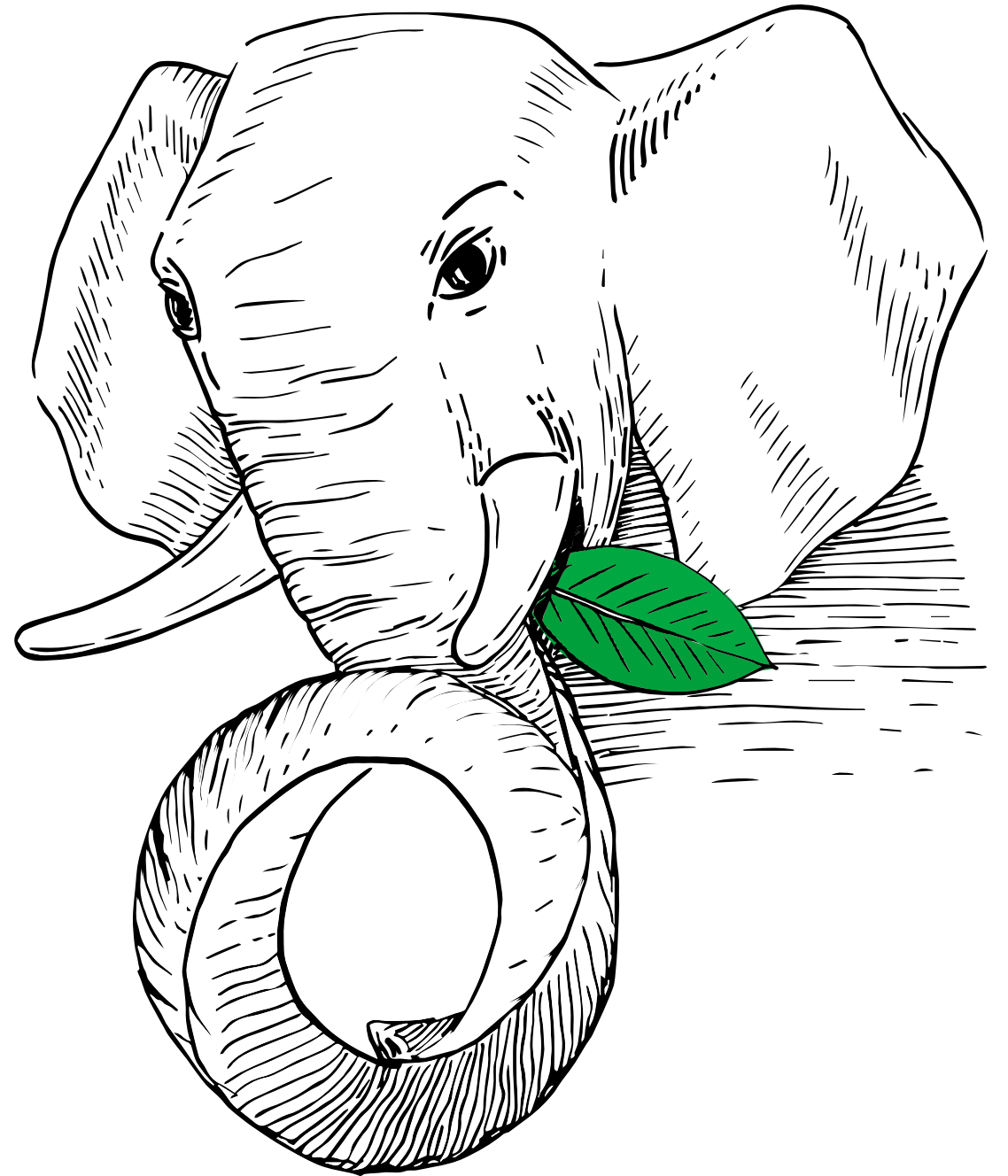




**PostgreSQL**

**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) !

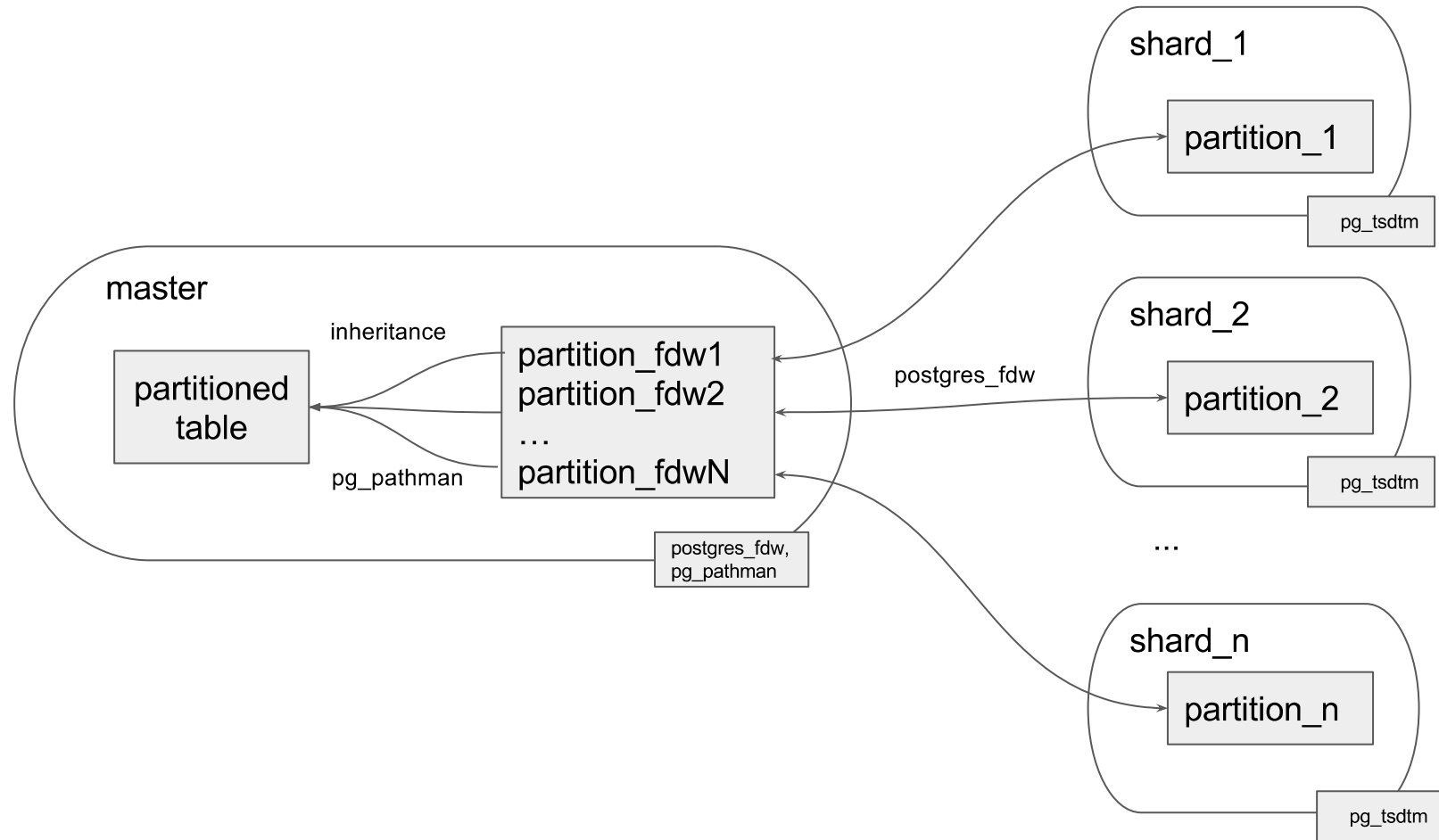


Still need more tps ?





# Use sharding !





# Sharding with postgres\_cluster

- Master: fork postgres\_cluster

[https://github.com/postgrespro/postgres\\_cluster](https://github.com/postgrespro/postgres_cluster)

- Shards: pg\_tsdtm

[https://github.com/postgrespro/pg\\_tsdtm](https://github.com/postgrespro/pg_tsdtm)



## Summary

- Postgres is already a good NoSQL database + clear roadmap
- 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 and MySQL beats MongoDB in one node configuration
  - Next: YCSB benchmarks in distributed mode
- Move from NoSQL to Postgres to avoid nightmare !





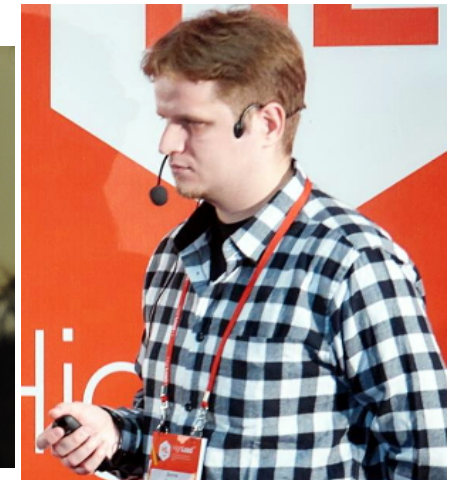
# PEOPLE BEHIND JSON[B]



Nikita Glukhov



Engine Yard™

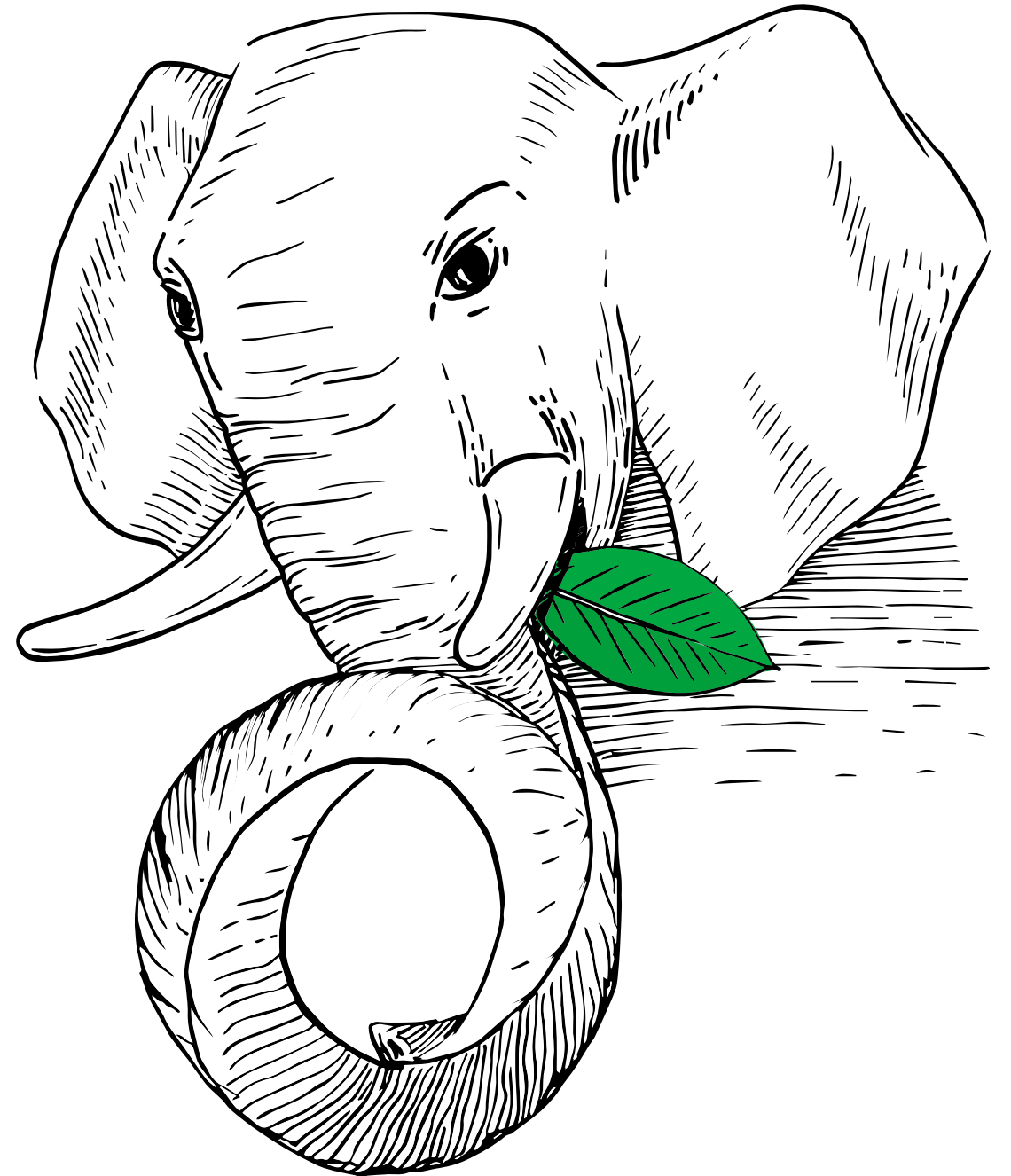
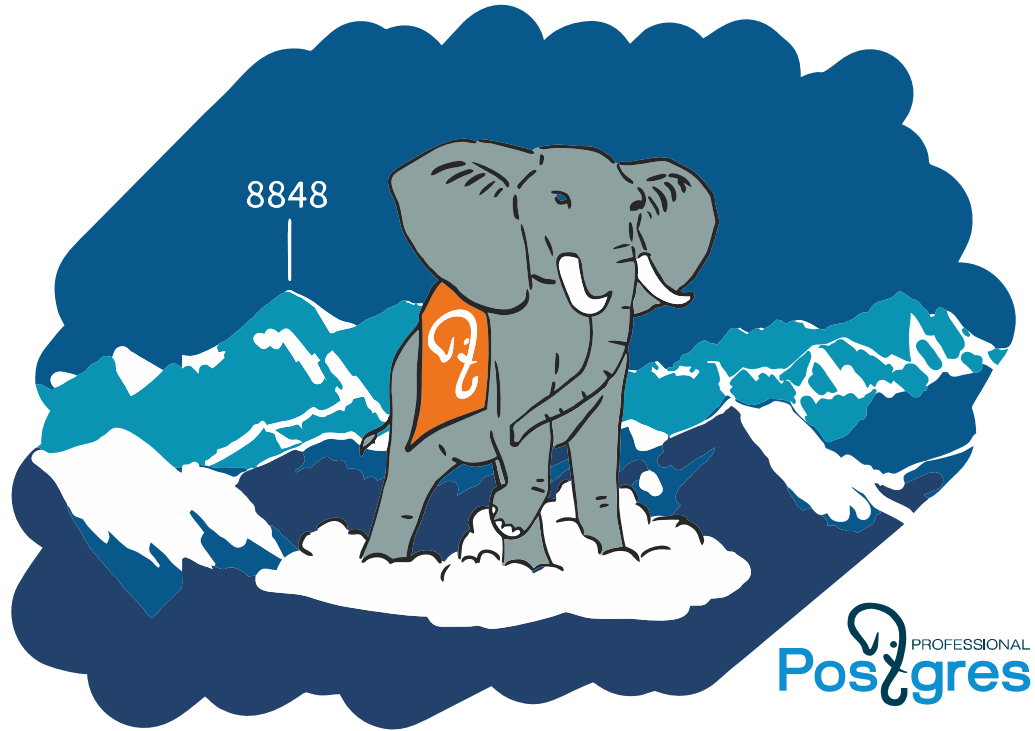




## Контакты:

- Олег Бартунов, [obartunov@postgrespro.ru](mailto:obartunov@postgrespro.ru)
- [www.postgrespro.ru](http://www.postgrespro.ru) - смотрите Образование
- [Реестр задач](#) для разработчиков
- [Hacking Postgres](#)
- [Developer FAQ](#)
- [Ресурсы для разработчиков на C](#)
- Мой ЖЖ: [obartunov.livejournal.ru](http://obartunov.livejournal.ru)  
(постгрес, горы, фото)
- Telegram: [@pgsql](#)
- Группа в FB: [PostgreSQL в России](#)

# EVEREST MARATHON 2017



# Thanks !