



JSON OR NOT JSON

THAT IS THE QUESTION

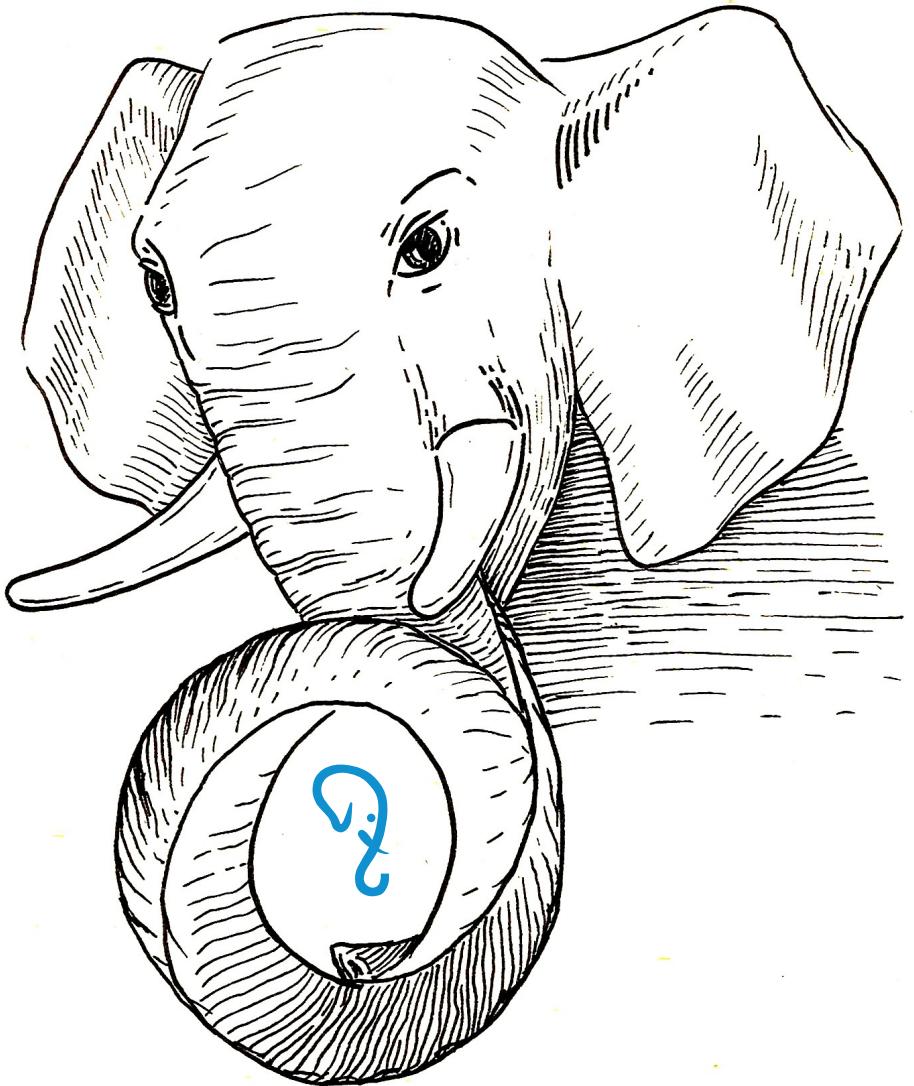
Since Postgres95

A circular word cloud centered on PostgreSQL features, including various indexing and search functions like GIST, GIN, SP-GIST, JSON, and Hstore. The words are arranged in concentric circles, with larger words representing more common features and smaller words representing less common ones. The colors of the words vary, creating a vibrant and dynamic visual effect.



Research scientist @
Moscow University
CEO Postgres Professional
Major PostgreSQL contributor

Nikita Glukhov



Senior developer @Postgres Professional
PostgreSQL contributor

Major CORE contributions:

- Jsonb improvements
- SQL/JSON (Jsonpath)
- KNN SP-GiST
- Opclass parameters

Current development:

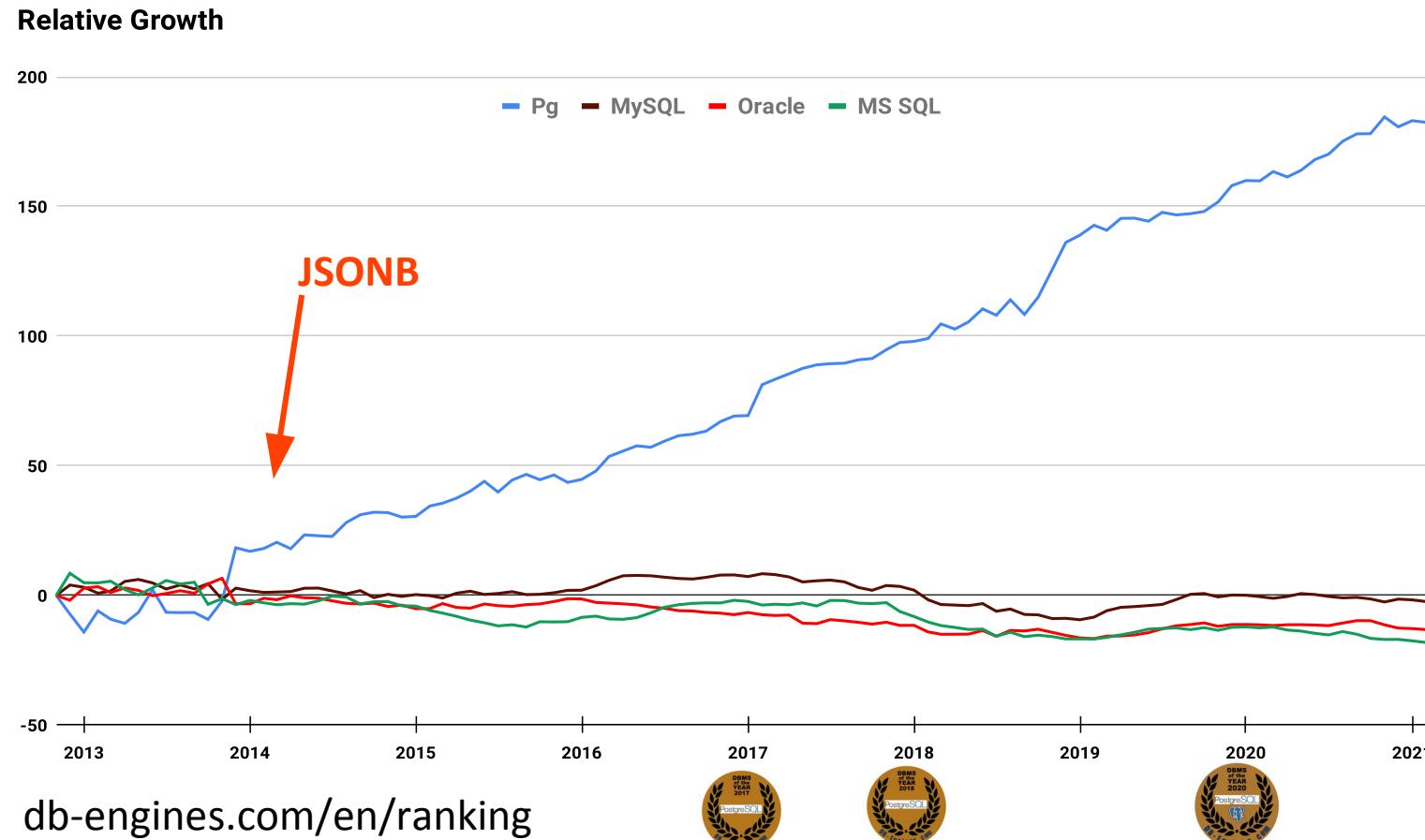
- SQL/JSON functions
- Jsonb performance

Why this talk ?

- My 20+ years interest and experience in extending Postgres
 - Arrays, hstore (2003), full text search, inexact search, index access methods (GiST, GIN, SP-GiST), spatial data (astronomy), now JSONB (2014), SQL/JSON
- JSONB (better JSON) performance is more important than compatibility
- Popular — microservices, clouds, startups
 - Ubiquitous format for data interchange, storing API messages (XML is too much)
 - Simple database design (simple queries), support of Agile development
 - Data migration (schema evolution). Old applications can easily accept new data.
 - Compact storage of metadata — one column for all
 - Client app, backend, database — one format, all server side languages support JSON, now SQL supports JSON, JSON relaxed code-centric vs data-centric
- Rash use of JSONB :)

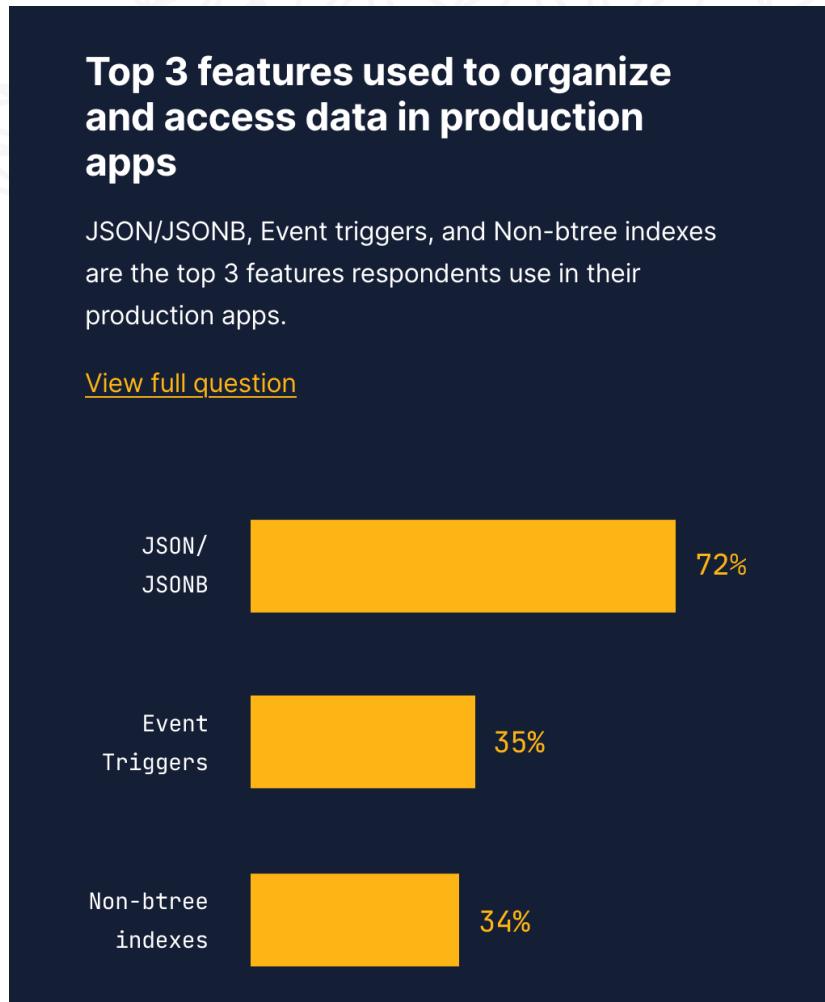
Postgres breathed a second life into relational databases

- Postgres innovation - the first relational database with NoSQL support
- NoSQL Postgres attracts the NoSQL users
- JSON became a part of SQL Standard 2016



JSONB Popularity - CREATE TABLE qq (js JSONB)

State of PostgreSQL 2021 ([Survey](#))

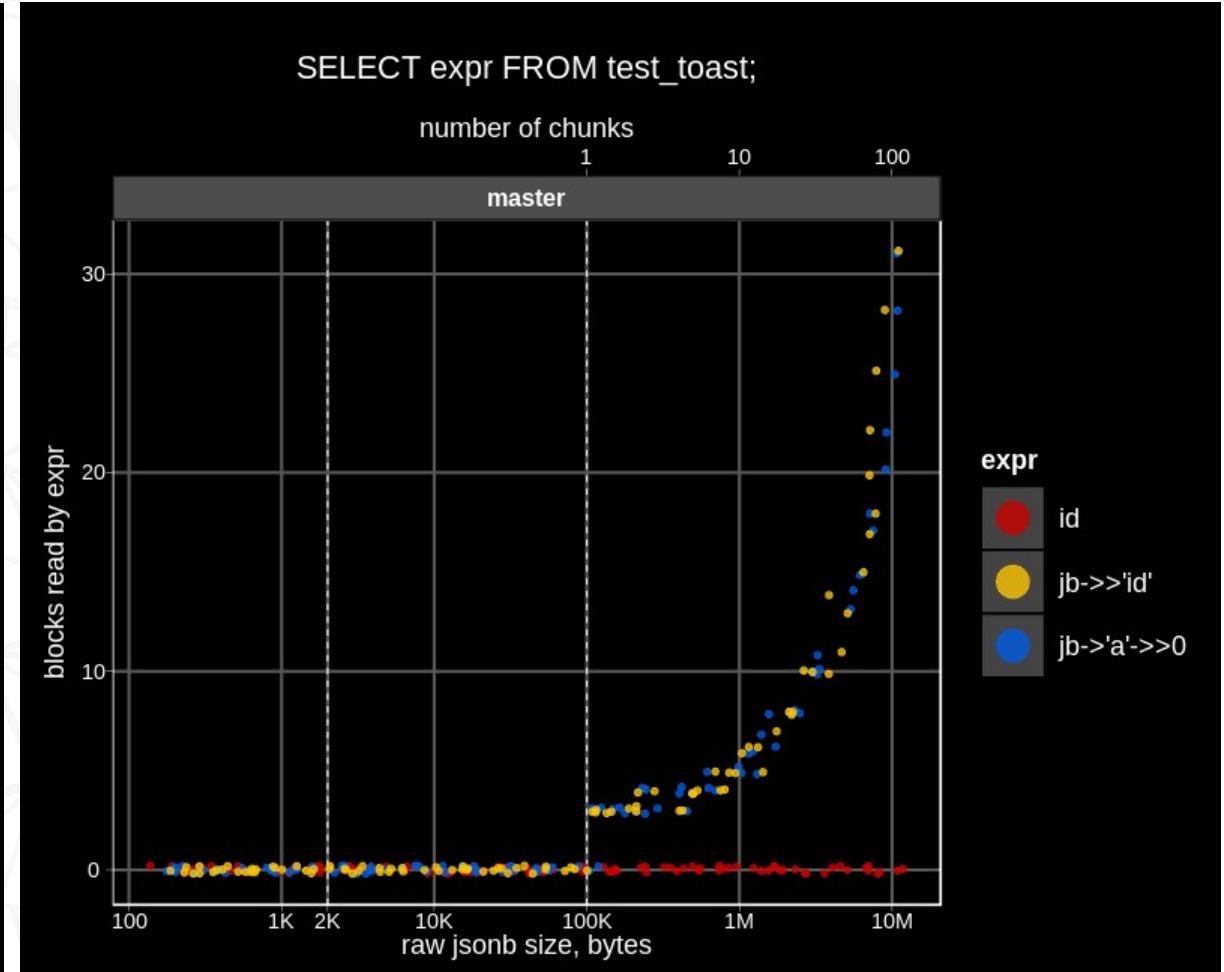
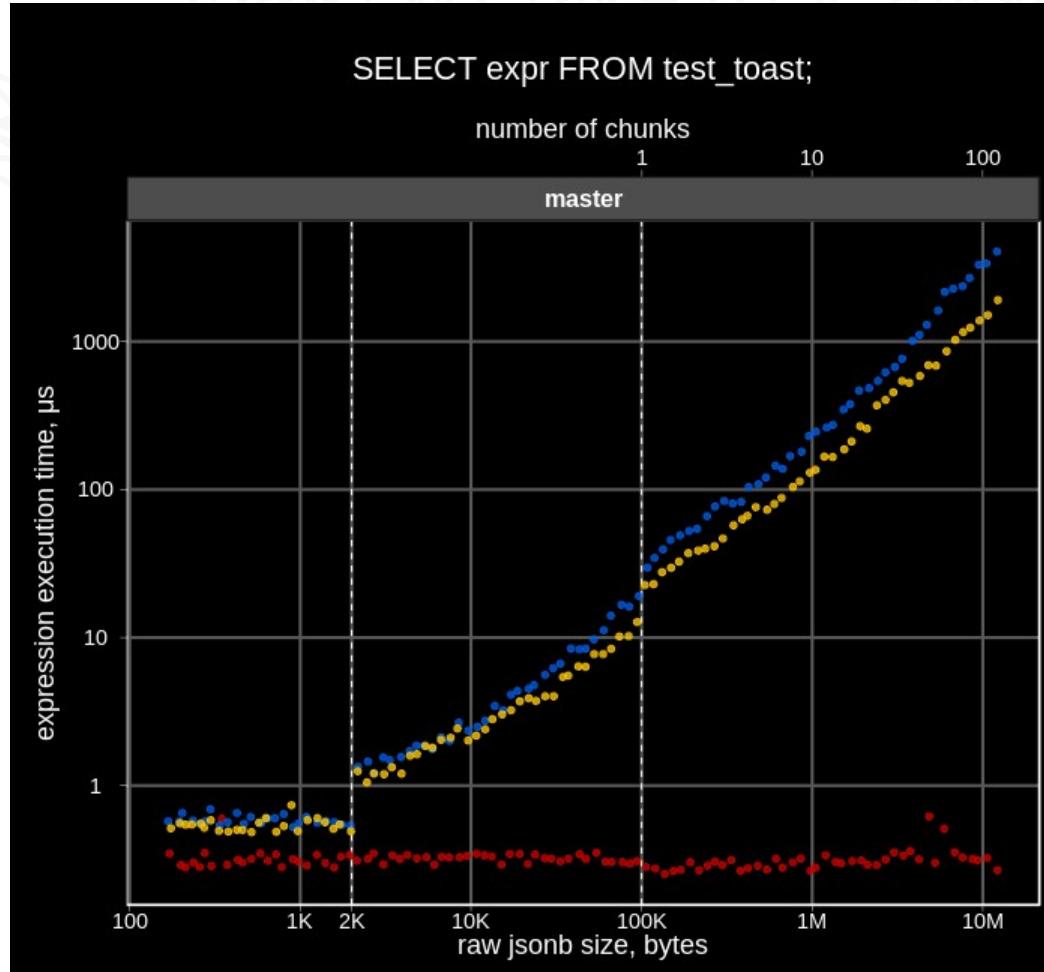


Pgsql telegram (7022) – 22.10.2021
<https://t.me/pgsql>

- SELECT 8061/312083
- SQL 4473/144789
- **JSON[B]** 3116/88234
- TABLE 2997/129936
- JOIN 2345/108860
- INDEX 1519/74327
- BACKUP 1484/42618
- VACUUM 1470/53919
- REPLICA 707/31036

Popular mistake: CREATE TABLE qq (jsonb)

(id**, {...}::jsonb) vs ({**id**,...}::jsonb)**



Large jsonb is TOASTed !

JSONB Projects: What we were working on

- SQL/JSON functions (SQL-2016) and JSON_TRANSFORM
- Generic JSON API (GSON). Jsonb as a SQL Standard JSON data type.
- *Better jsonb indexing (Jquery GIN opclasses)*
- *Parameters for jsonb operators (planner support functions for Jsonb)*
- *JSONB selective indexing (Jsonpath as parameter for jsonb opclasses)*
- *Jsonpath syntax extension*
- *Simple Dot-Notation Access to JSON Data*

Current TOP-priority project

- SQL/JSON functions (SQL-2016) and JSON_TRANSFORM
- Generic JSON API. Jsonb as a SQL Standard JSON data type.
- *Better jsonb indexing (Jquery GIN opclasses)*
- *Parameters for jsonb operators (planner support functions for Jsonb)*
- *JSONB selective indexing (Jsonpath as parameter for jsonb opclasses)*
- *Jsonpath syntax extension*
- *Simple Dot-Notation Access to JSON Data*
- **JSONB - 1st-class citizen in Postgres**
 - Efficient storage,select, update, API

Top-priority: JSONB - 1st-class citizen in Postgres

- Popularity of JSONB — it's mature data type, rich functionality
- Startups use Postgres and don't care about compatibility to Oracle/MS SQL
 - Jsonpath is important and committed
 - There is rich user API to Jsonb, so SQL/JSON functions are not in top-priority list
- Not enough resources in community (developers, reviewers, committers)
 - SQL/JSON — 4 years, 59 versions
 - JSON/Table — 4 years, 52 versions
 - Waiting for PG15
- We concentrate on efficient storage, select, update (OLTP+OLAP)
 - Extendability of JSONB format
 - Extendability of TOAST — data type aware TOAST, TOAST for non-atomic attributes

The Curse of TOAST. Unpredictable performance

Small update cause 10 times slowdown !

```
=# EXPLAIN(ANALYZE, BUFFERS) SELECT jb->'id' FROM test;  
          QUERY PLAN
```

```
Seq Scan on test (cost=0.00..2625.00 rows=10000 width=32) (actual time=0.014..6.128 rows=10000 loops=1)
```

Buffers: shared hit=**2500**

Planning:

Buffers: shared hit=5

Planning Time: 0.087 ms

Execution Time: **6.583 ms**

(6 rows)

```
=# UPDATE test SET jb = jb || '{"bar": "baz"}';  
=# VACUUM FULL test; -- remove old versions
```

```
=# EXPLAIN (ANALYZE, BUFFERS) SELECT jb->'id' FROM test;  
          QUERY PLAN
```

```
Seq Scan on test (cost=0.00..2675.40 rows=10192 width=32) (actual time=0.067..65.511 rows=10000 loops=1)
```

Buffers: shared hit=**30064**

Planning Time: 0.044 ms

Execution Time: **66.889 ms**
(4 rows)

Row gets TOASTed ! See TOAST explained slides

```
CREATE TABLE test (jb jsonb);  
ALTER TABLE test ALTER COLUMN jb SET STORAGE EXTERNAL;  
INSERT INTO test  
SELECT  
    jsonb_build_object(  
        'id', i,  
        'foo', (select jsonb_agg(0) from generate_series(1, 1960/12)) -- [0,0,0, ...]  
    ) jb  
FROM  
    generate_series(1, 10000) i;
```

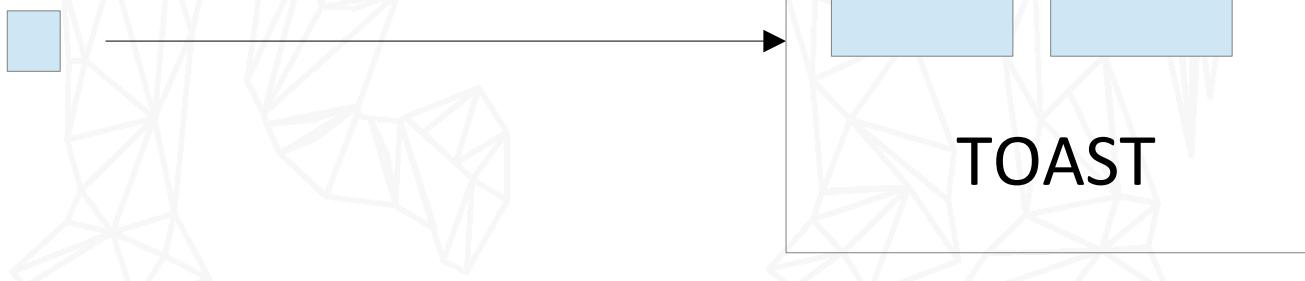
The Curse of TOAST

- Original JSONBs stored inline in heap tuples (2500 pages with 4 tuples per page):

```
CREATE EXTENSION pageinspect;
SELECT lp_len FROM heap_page_items(get_raw_page('test', 0));
lp_len
-----
2022
2022
2022
2022
(4 rows)
```

- JSONBs after update became larger than 2K and postgres replaced them by pointer to special TOAST relation (see TOAST explained slides), so the tuple length is greatly decreased (64 pages with 157 tuples per page):

```
SELECT lp_len FROM heap_page_items(get_raw_page('test', 0));
lp_len
-----
42
42
...
42
(156 rows)
```



The Curse of TOAST

- JSONB data has moved into TOAST relation:

```
SELECT reltoastrelid::regclass toast_rel FROM pg_class
WHERE oid = 'test'::regclass;
      toast_rel
-----
 pg_toast.pg_toast_16460
(1 row)
```

- Each JSONB is splitted into two TOAST chunks, that implicitly joined by index to attribute, when its value is fetched. Chunks belonging to the one attribute has the same chunk_id, which stored in TOAST pointer:

```
SELECT chunk_id, chunk_seq, length(chunk_data) FROM pg_toast.pg_toast_16460;
      chunk_id | chunk_seq | length
-----+-----+-----+
      16466 |         0 |     1996
      16466 |         1 |       10
      16467 |         0 |     1996
      16467 |         1 |       10
...
(20000 rows)
```

The Curse of TOAST

- Access to TOASTed JSONB requires reading at least 3 additional buffers:
 - 2 TOAST index buffers (B-tree height is 2)
 - 1 TOAST heap buffer
 - 2 chunks read from the same page, if JSONB size > Page size (8Kb), then more TOAST heap buffers

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF, TIMING OFF)
SELECT jb->'id' FROM test;
```

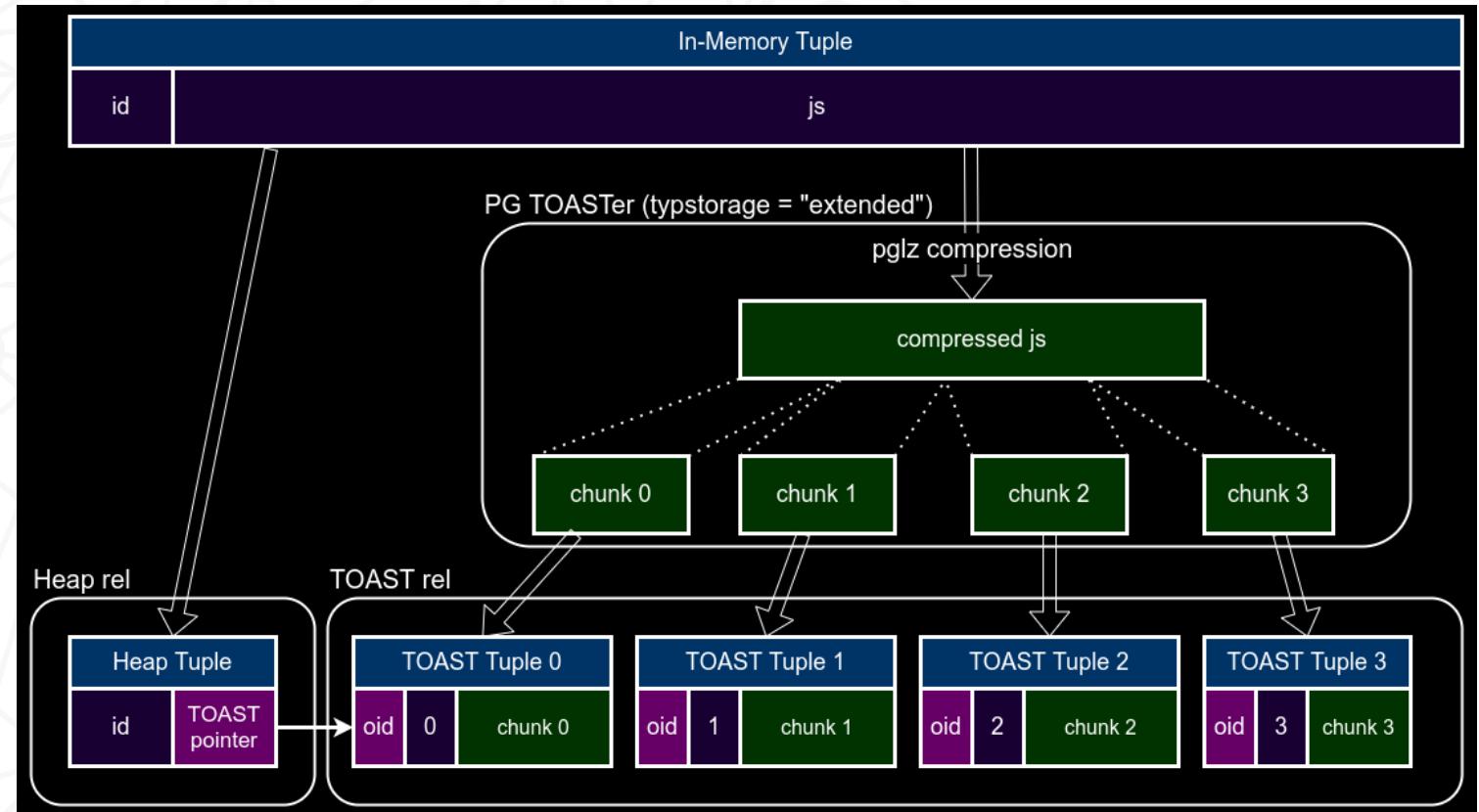
```
QUERY PLAN
-----
Seq Scan on test (actual rows=100 loops=1)
  Buffers: shared hit=30064
            Buffers: shared hit=301
Planning Time: 0.186 ms
Execution Time: 56 ms
(6 rows)
```

Table	TOAST
64 buffers + 3 buffers*10000	

TOAST Explained

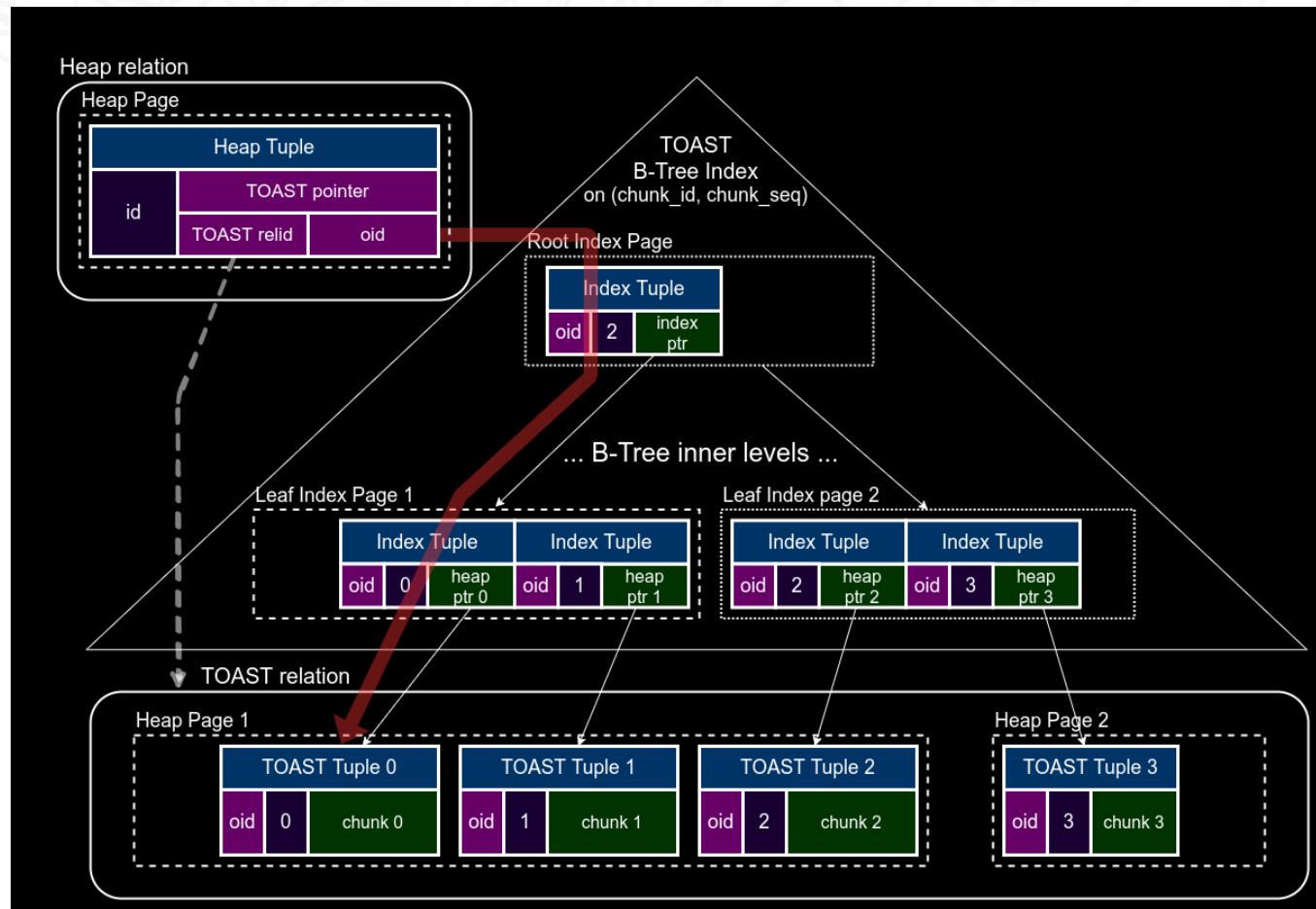
The Oversized-Attribute Storage Technique

- TOASTed (large field) values are compressed, then split into the fixed-size TOAST chunks (1996B for 8KB page)
- TOAST chunks (along with generated Oid chunk_id and sequence number chunk_seq) stored in special TOAST relation pg_toast.pg_toast_XXX, created for each table containing TOASTable attributes
- Attribute in the original heap tuple is replaced with TOAST pointer (18 bytes) containing chunk_id, toast_relid, raw_size, compressed_size



TOAST access

- TOAST pointers does not refer to heap tuples with chunks directly. Instead they contains Oid chunk_id, so one need to descent by index (chunk_id, chunk_seq).



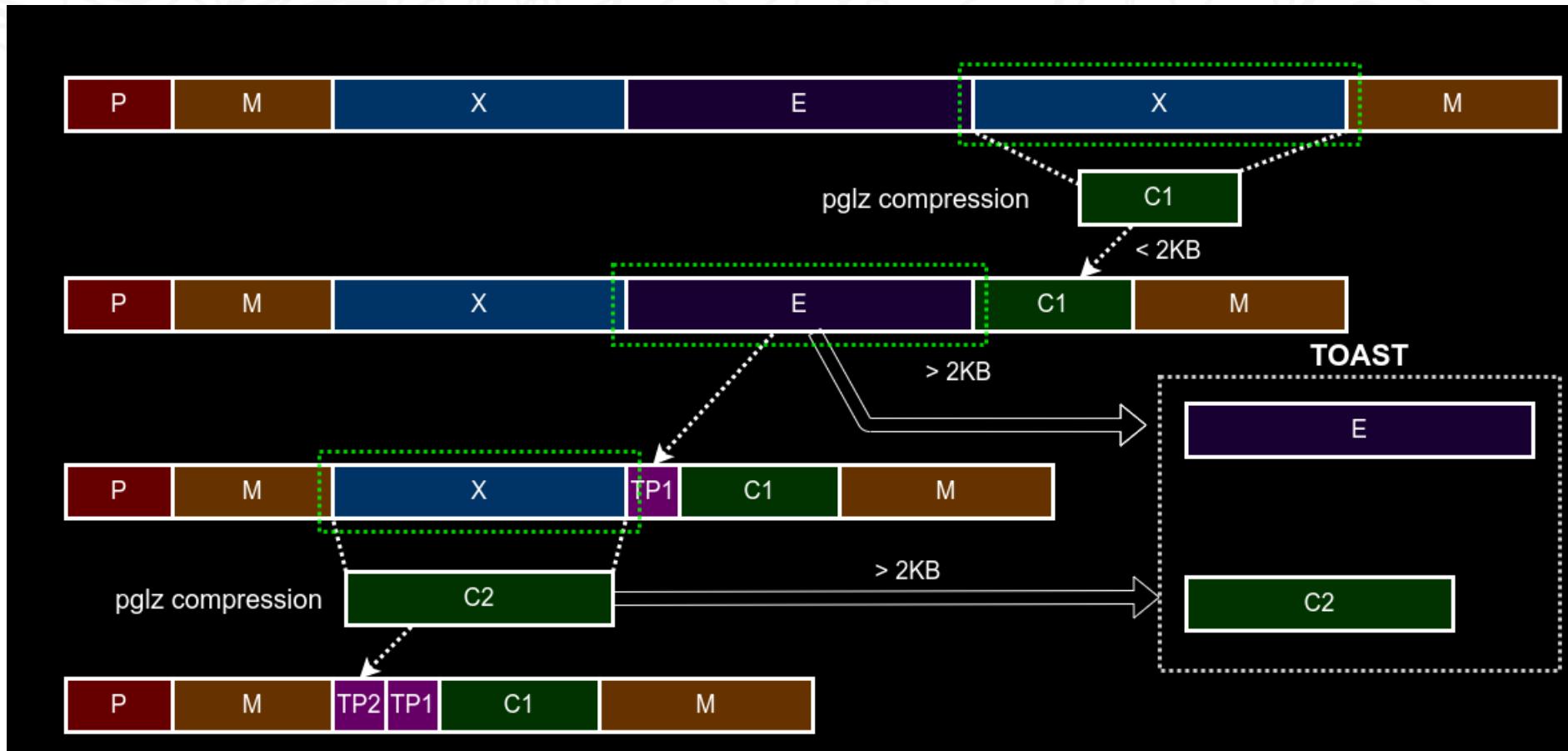
Overhead to read only a few bytes from the first chunk is 3,4 or even 5 additional index blocks.

TOAST passes

- Tuple is TOASTed if its size is more than 2KB (1/4 of page size).
- There are 4 TOAST passes.
- At each pass considered only attributes of the specific storage type (extended/external or main) starting from the largest one.
- Plain attributes are not TOASTed and not compressed at all.
- The process can stop at every step, if the resulting tuple size becomes less than 2KB.
- If the attributes were copied from the other table, they can already be compressed or TOASTed.
- TOASTed attributes are replaced with TOAST pointers.

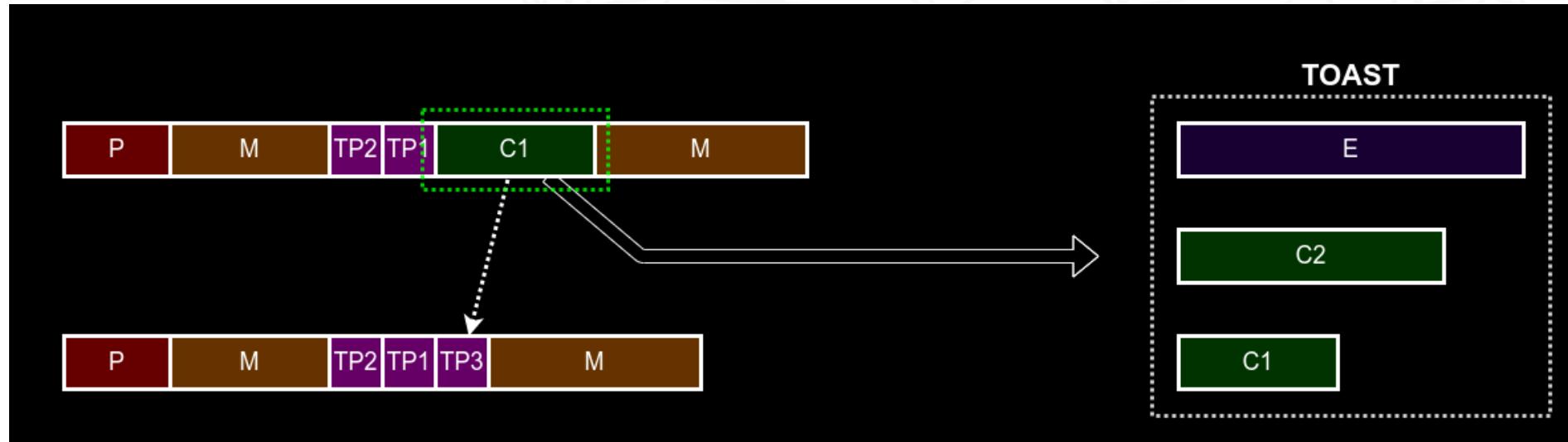
TOAST pass #1

- Only "extended" and "external" attributes are considered, "extended" attributes are compressed. If their size is more than 2KB, they are TOASTed.



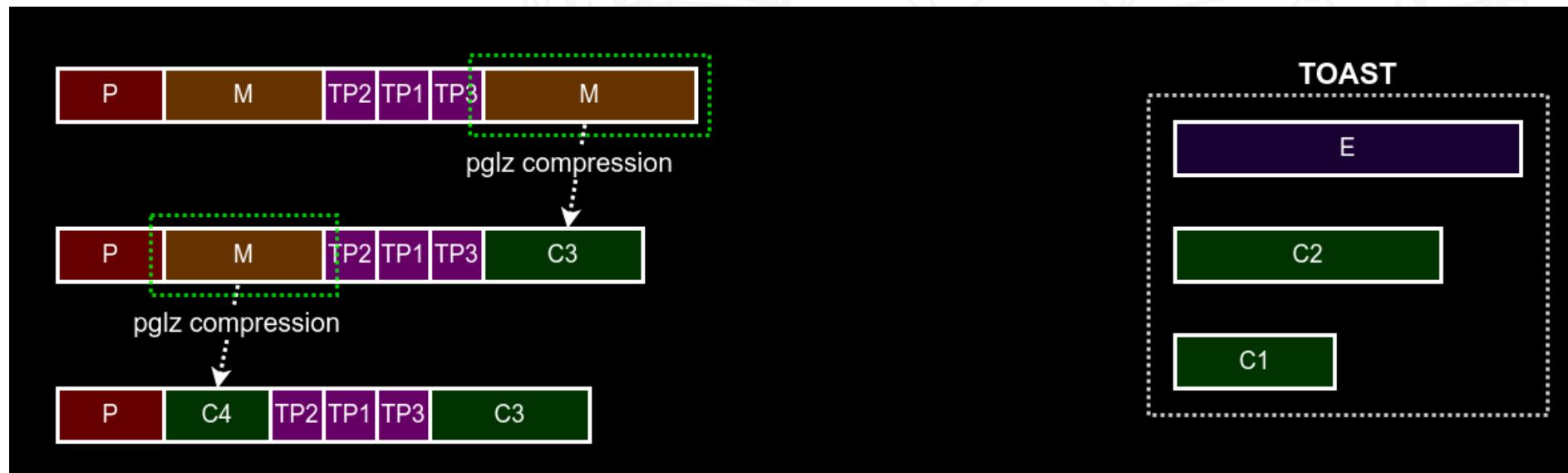
TOAST pass #2

- Only "extended" and "external" attributes (that were not TOASTed in the previous pass) are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.



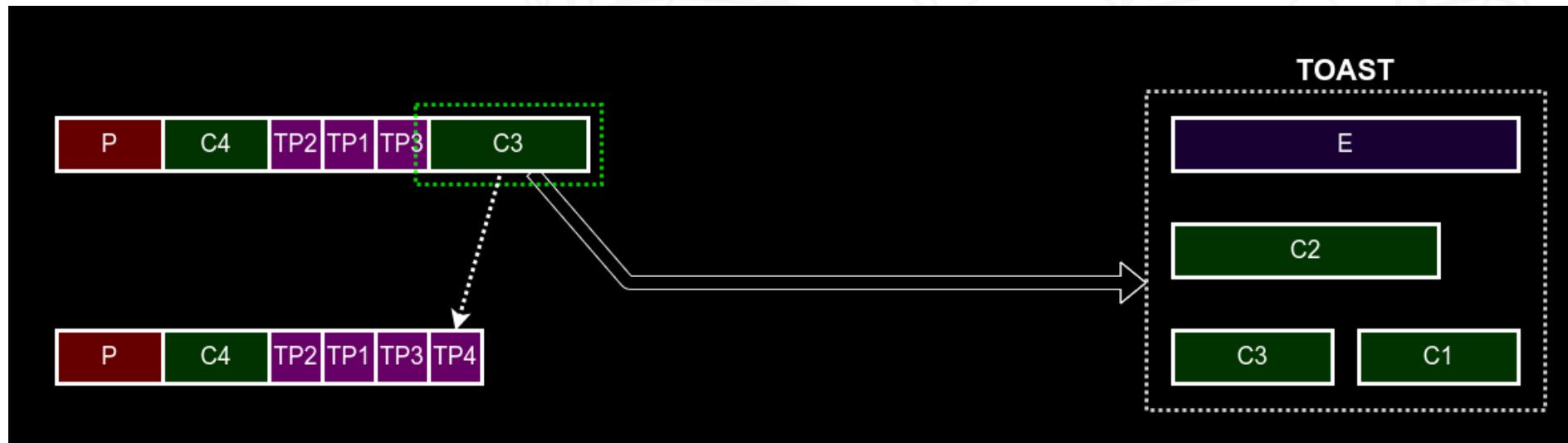
TOAST pass #3

- Only "main" attributes are considered.
- Each attribute is compressed, until the resulting tuple size < 2KB.



TOAST pass #4

- Only "main" attributes are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.



Motivational example (synthetic test)

- A table with 100 jsonbs of different sizes (130B-13MB, compressed to 130B-247KB):

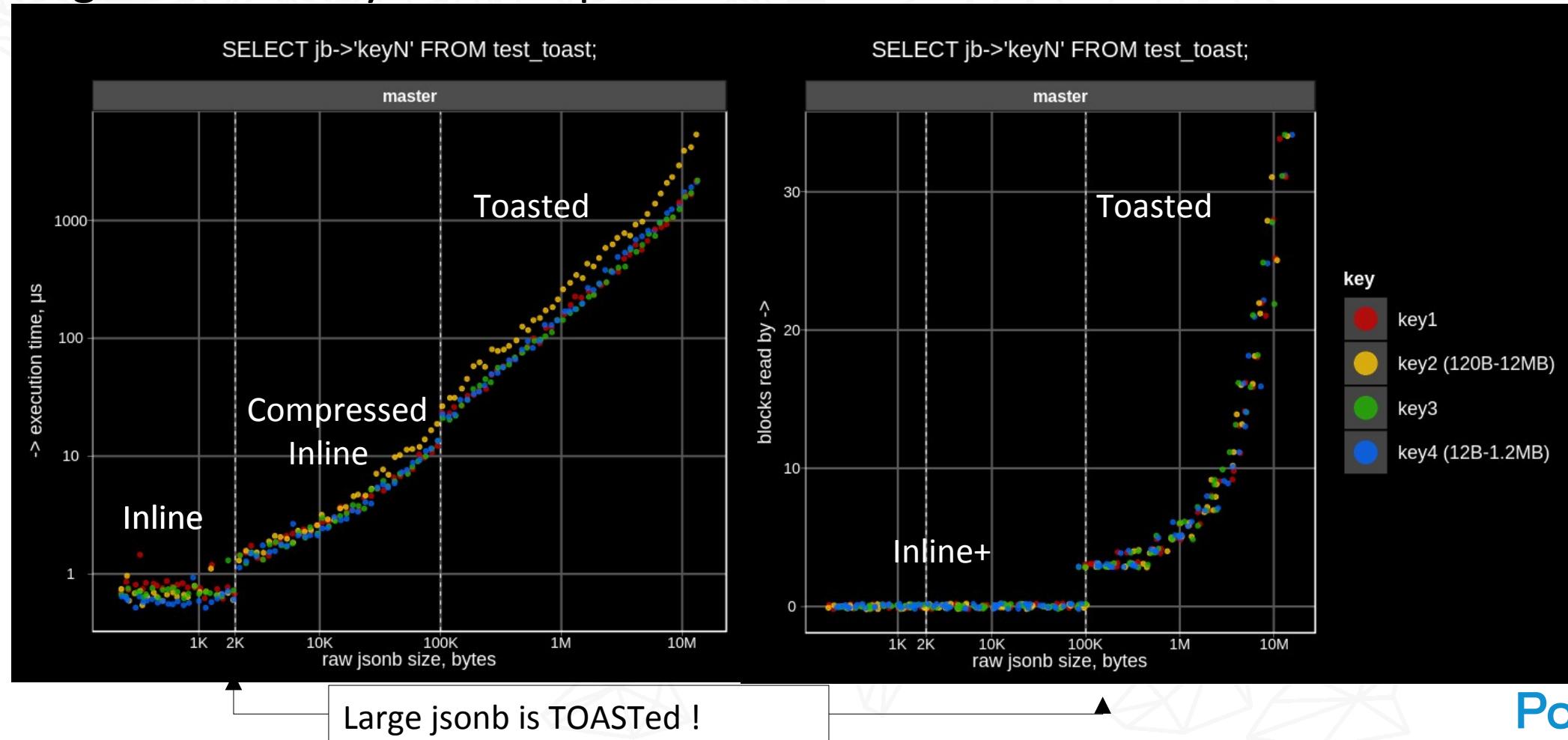
```
CREATE TABLE test_toast AS
SELECT
    i id,
    jsonb_build_object(
        'key1', i,
        'key2', (select jsonb_agg(0) from
                  generate_series(1, pow(10, 1 + 5.0 * i / 100.0)::int)),-- 10-100k elems
        'key3', i,
        'key4', (select jsonb_agg(0) from
                  generate_series(1, pow(10, 0 + 5.0 * i / 100.0)::int)) -- 1-10k elems
    ) jb
FROM generate_series(1, 100) i;
```

- Each jsonb looks like: key1, loooong key2, key3, long key4.
- We measure execution time of operator ->(jsonb, text) for each row by repeating it 1000 times in the query:

```
SELECT jb -> 'keyN', jb -> 'keyN', ... jb -> 'keyN' FROM test_toast WHERE id = ?;
```

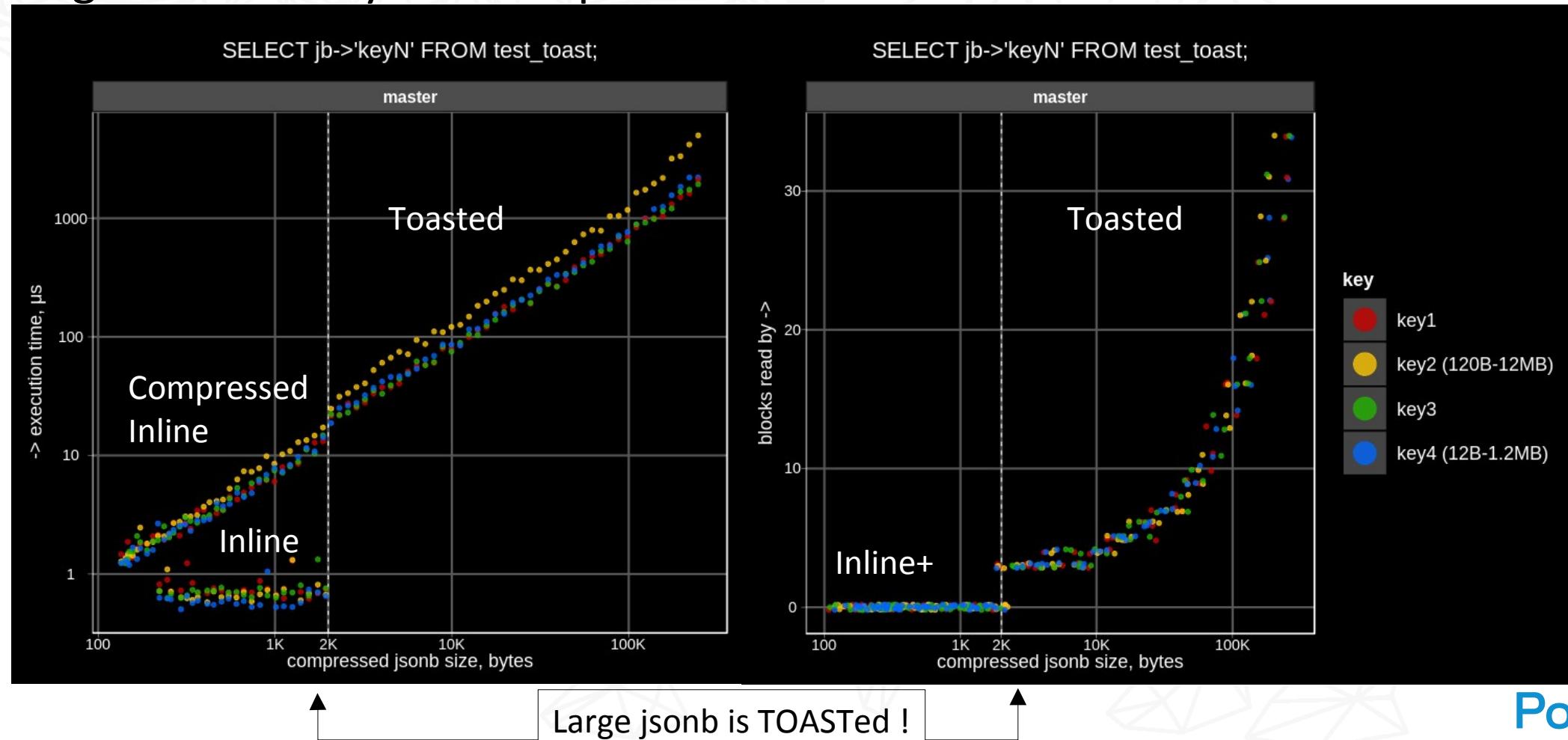
Motivational example (synthetic test)

Key access time for TOASTed jsonbs linearly increase with jsonb size, regardless of key size and position.



TOAST performance problems (synthetic test)

Key access time for TOASTed jsonbs linearly increase with jsonb size, regardless of key size and position.



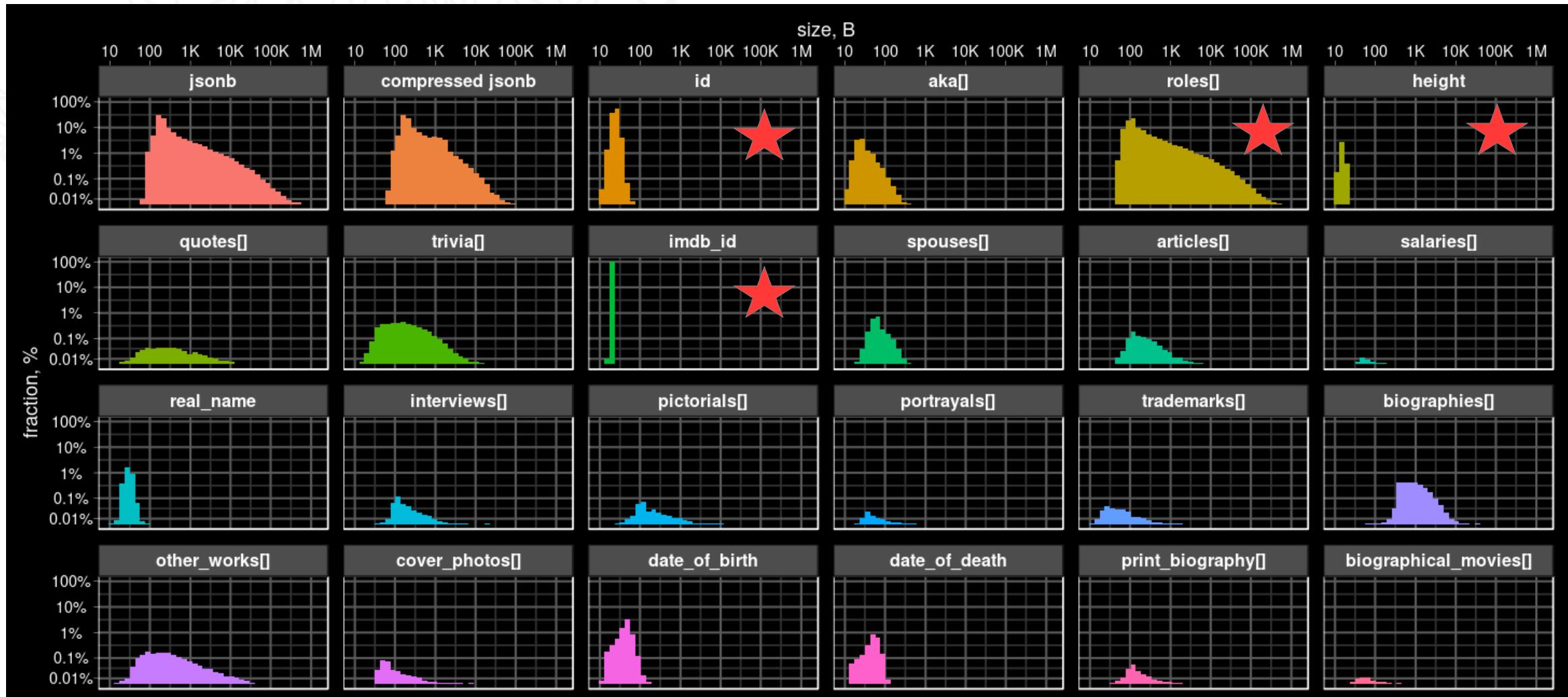
Motivational example (IMDB test)

- Real-world JSON data extracted from IMDB database ([imdb-22-04-2018-json.dump.gz](#))
- Typical IMDB «name» document looks like:

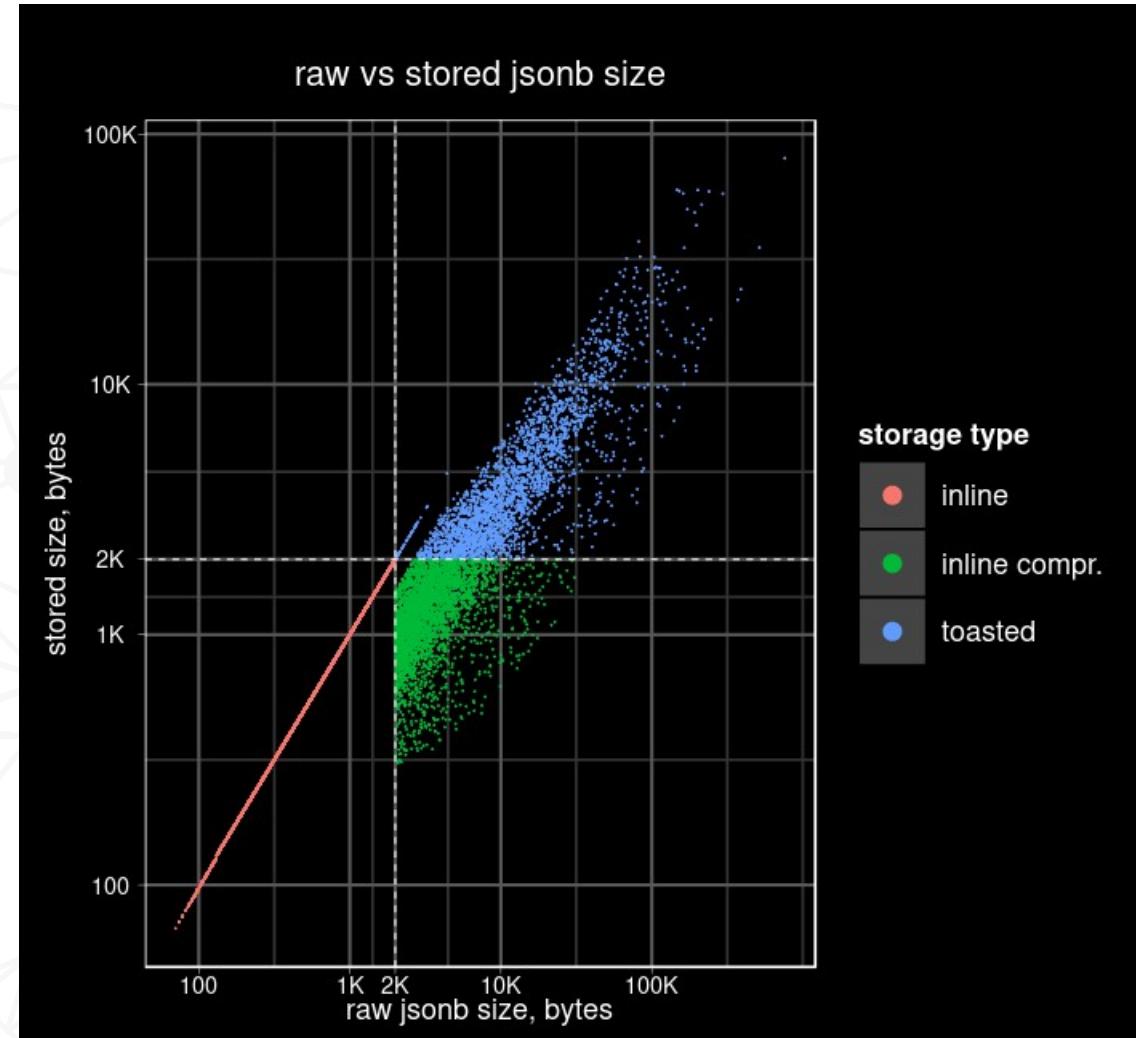
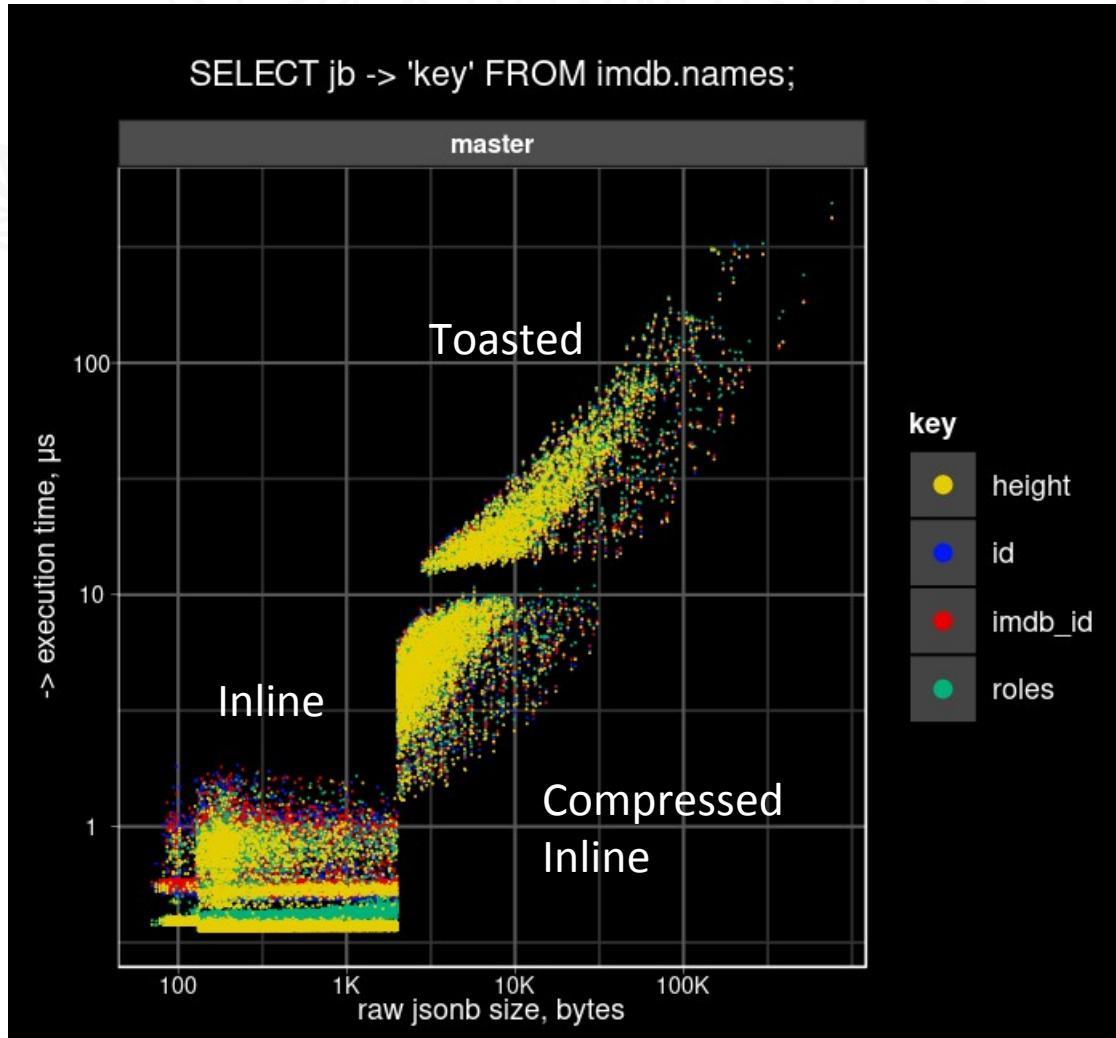
```
{  
    "id": "Connors, Steve (V)",  
    "roles": [  
        {  
            "role": "actor",  
            "title": "Copperhead Creek (????)"  
        },  
        {  
            "role": "actor",  
            "title": "Ride the Wanted Trail (????)"+  
        }  
    ],  
    "imdb_id": 1234567  
}
```

- There are many other infrequent fields, but only `id`, `imdb_id` are mandatory, and `roles` array is the **biggest** and most frequent (see next slide).

IMDB data set field statistics



Motivational example (IMDB test)



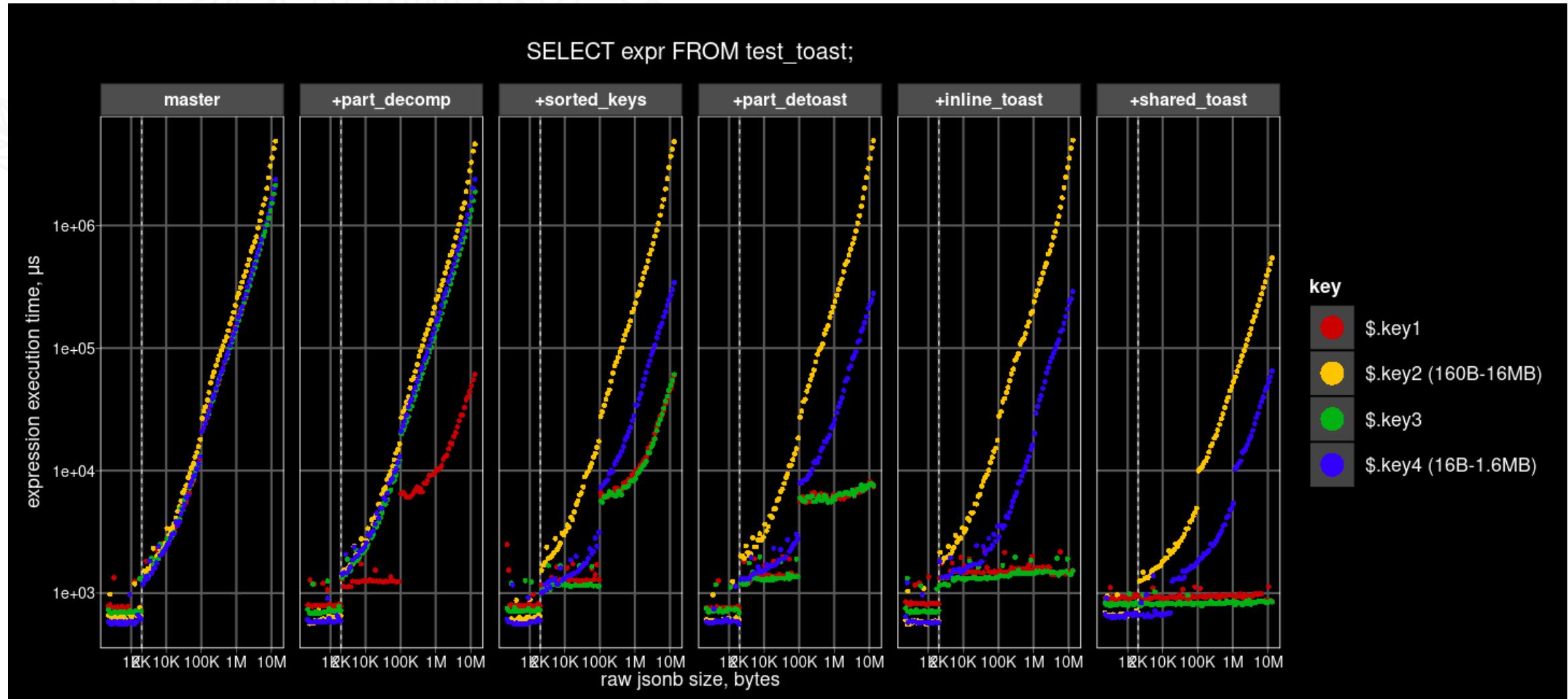
Motivation

- Decompression is the biggest problem. Big overhead of decompression of the whole jsonb limits the applicability of jsonb as document storage with partial access.
 - Need partial decompression
- Toast introduces additional overhead - read too many blocks
 - Read only needed blocks — partial detoast

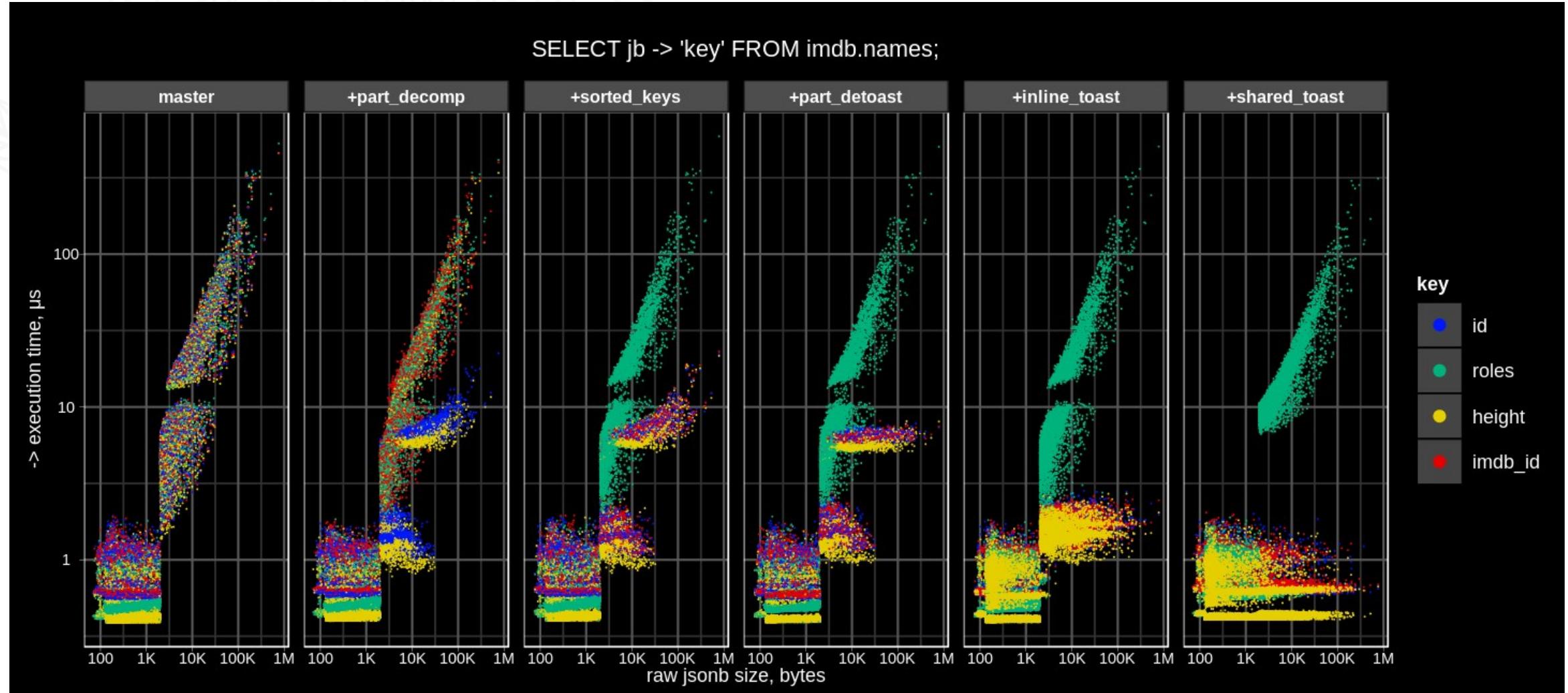
Jsonb deTOAST improvements

- Partial (prefix) decompression
- Sort jsonb object key by their length – good for short metadata
- Partial deTOASTing using TOAST iterators – decompress chunk by chunk
- Inline TOAST – store in heap tuple data from the first chunk
- Shared TOAST – store in heap tuple uncompresed short keys, compress chunks separately, share common chunks
 - Access
 - Update - share
 - In-place update – don't copy shared chunks if length is not changed, store new value inline if possible

Step-by-step results (access key,synthetic)

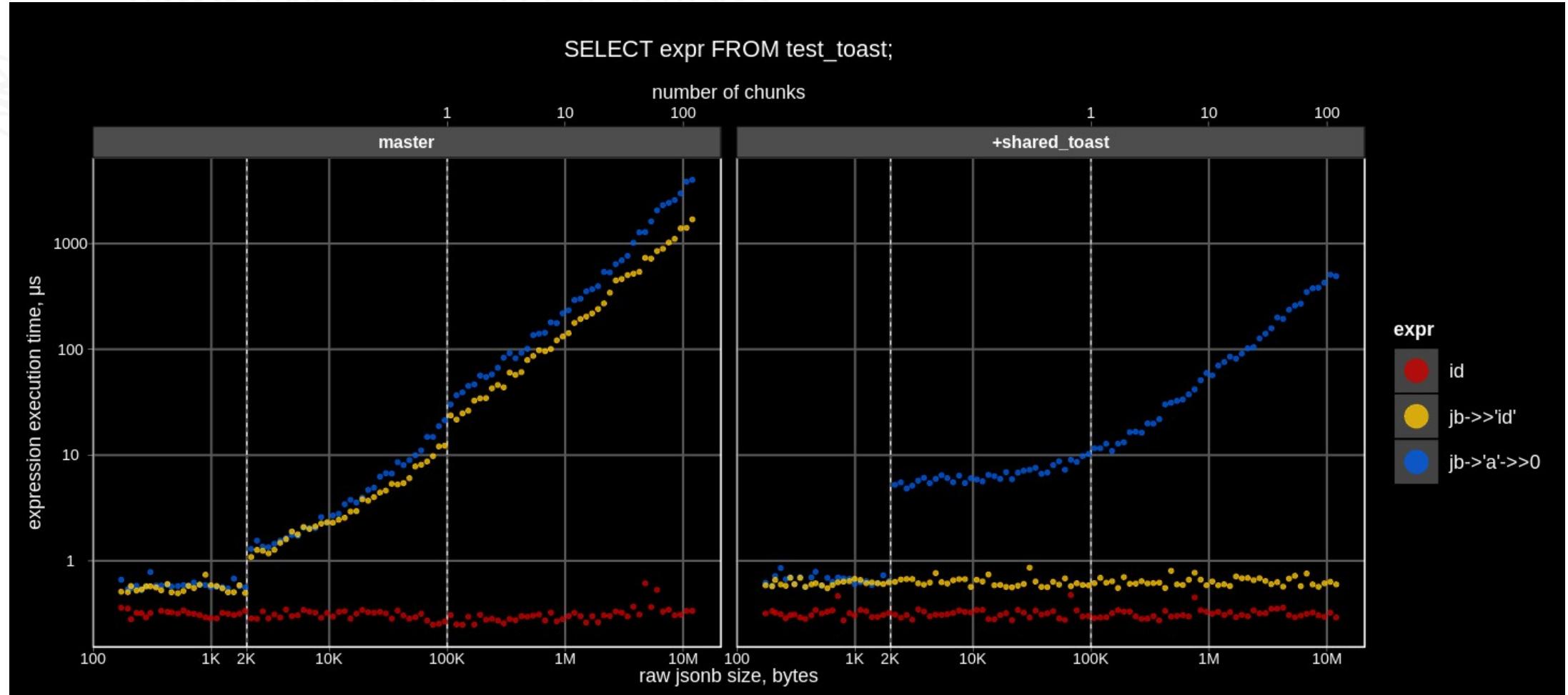


Step-by-step results (access key, IMDB)



Popular mistake: CREATE TABLE qq (jsonb)

(id**, {...}::jsonb) vs ({**id**,...}::jsonb)**



Large jsonb is TOASTed !

JSONB partial update

TOAST was originally designed for atomic data types, it knows nothing about internal structure of composite data types like jsonb, hstore, and even ordinary arrays.

TOAST works only with binary BLOBs, it does not try to find differences between old and new values of updated attributes. So, when the TOASTed attribute is being updated (does not matter at the beginning or at the end and how much data is changed), its chunks are simply fully copied. The consequences are:

- TOAST storage is duplicated
- WAL traffic is increased in comparison with updates of non-TOASTED attributes, because the whole TOASTed values is logged
- Performance is too low

JSONB partial update: The problem

Example: table with 10K jsonb objects with 1000 keys { "1": 1, "2": 2, ... }.

```
CREATE TABLE t AS
SELECT i AS id, (SELECT jsonb_object_agg(j, j) FROM generate_series(1, 1000) j) js
FROM generate_series(1, 10000) i;
```

```
SELECT oid::regclass AS heap_rel,
       pg_size.pretty(pg_relation_size(oid)) AS heap_rel_size,
       reltoastrelid::regclass AS toast_rel,
       pg_size.pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
FROM pg_class WHERE relname = 't';
```

heap_rel	heap_rel_size	toast_rel	toast_rel_size
t	512 kB	pg_toast.pg_toast_27227	78 MB

Each 19 KB jsonb is compressed into 6 KB and stored in 4 TOAST chunks.

```
SELECT pg_column_size(js) compressed_size, pg_column_size(js::text::jsonb) orig_size from t limit 1;
compressed_size | original_size
-----+-----
6043 | 18904
```

```
SELECT chunk_id, count(chunk_seq) FROM pg_toast.pg_toast_47235 GROUP BY chunk_id LIMIT 1;
chunk_id | count
-----+-----
57241 | 4
```

JSONB partial update: The problem

First, let's try to update of non-TOASTED int column id:

```
SELECT pg_current_wal_lsn(); --> 0/157717F0
```

```
UPDATE t SET id = id + 1; -- 42 ms
```

```
SELECT pg_current_wal_lsn(); --> 0/158E5B48
```

```
SELECT pg_size.pretty(pg_wal_lsn_diff('0/158E5B48','0/157717F0')) AS wal_size;
wal_size
-----
1489 kB (150 bytes per row)
```

```
SELECT oid::regclass AS heap_rel,
       pg_size.pretty(pg_relation_size(oid)) AS heap_rel_size,
       reltoastrelid::regclass AS toast_rel,
       pg_size.pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
FROM pg_class
WHERE relname = 't';
      heap_rel | heap_rel_size |          toast_rel          | toast_rel_size
-----+-----+-----+-----+
      t    | 1024 kB        | pg_toast.pg_toast_47235 | 78 MB
           (was 512 kB)          (not changed)
```

JSONB partial update: The problem

Next, let's try to update of TOASTED jsonb column js:

```
SELECT pg_current_wal_lsn(); --> 0/158E5B48
```

```
UPDATE t SET js = js - '1'; -- 12316 ms (was 42 ms, ~300x slower)
```

```
SELECT pg_current_wal_lsn(); --> 0/1DB10000
```

```
SELECT pg_size.pretty(pg_wal_lsn_diff('0/1DB10000','0/158E5B48')) AS wal_size;  
wal_size
```

```
-----  
130 MB (13 KB per row; was 1.5 MB, ~87x more)
```

```
SELECT oid::regclass AS heap_rel,  
       pg_size.pretty(pg_relation_size(oid)) AS heap_rel_size,  
       reltoastrelid::regclass AS toast_rel,  
       pg_size.pretty(pg_relation_size(reltoastrelid)) AS toast_rel_size
```

```
FROM pg_class
```

```
WHERE relname = 't';
```

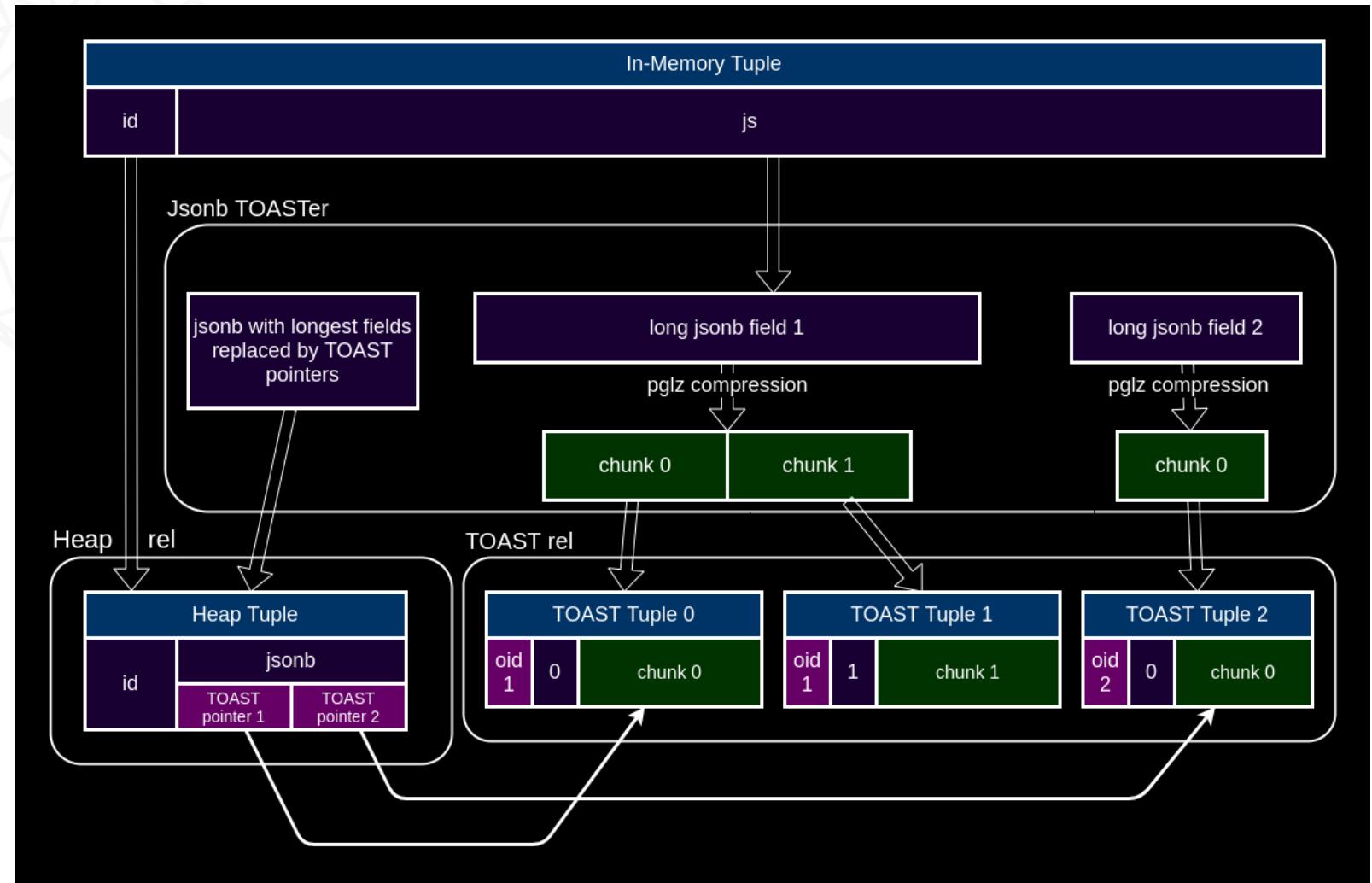
	heap_rel	heap_rel_size	toast_rel	toast_rel_size
t	1528 kB (was 1024 kB)		pg_toast.pg_toast_47235	156 MB (was 78 MB, 2x more)

Partial update using Shared TOAST

- The previous optimizations are great for SELECT, but don't help with UPDATE, since TOAST consider jsonb as an atomic binary blob – change part, copy the whole.
- Idea: Keep INLINE short fields (uncompressed) and TOAST pointers to long fields to let update short fields without modification of TOAST chunks, which will be shared between versions.
- Currently, this works only for root objects fields, so the longest fields of jsonb object are TOASTed until the whole tuple fits into the page (typically, remaining size of jsonb becomes < ~2000 bytes).
- But this technique can also be applied to array elements or element ranges. We plan to try to implement it later, it needs more invasive jsonb API changes.
- Currently, jsonb hook is hardcoded into TOAST pass #1, but in the future it will become custom datatype TOASTer using pg_type.typttoast.

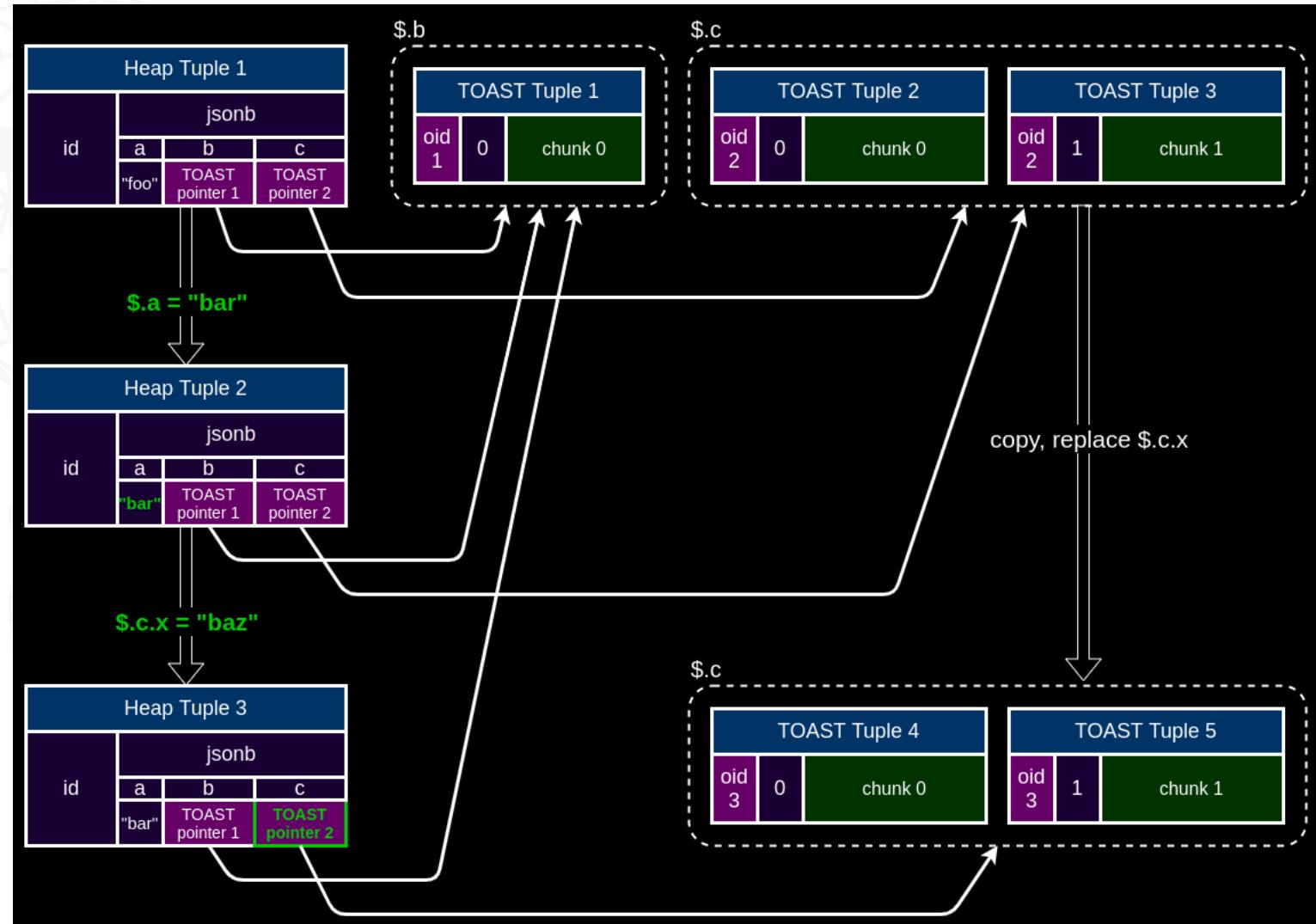
Shared TOAST – tuple structure

- In this example two largest fields of jsonb are TOASTed separately
- TOASTed jsonb contains two TOAST pointers
- Operators like `->` can simply return TOAST pointer as external datum, accessing only the inline part of jsonb



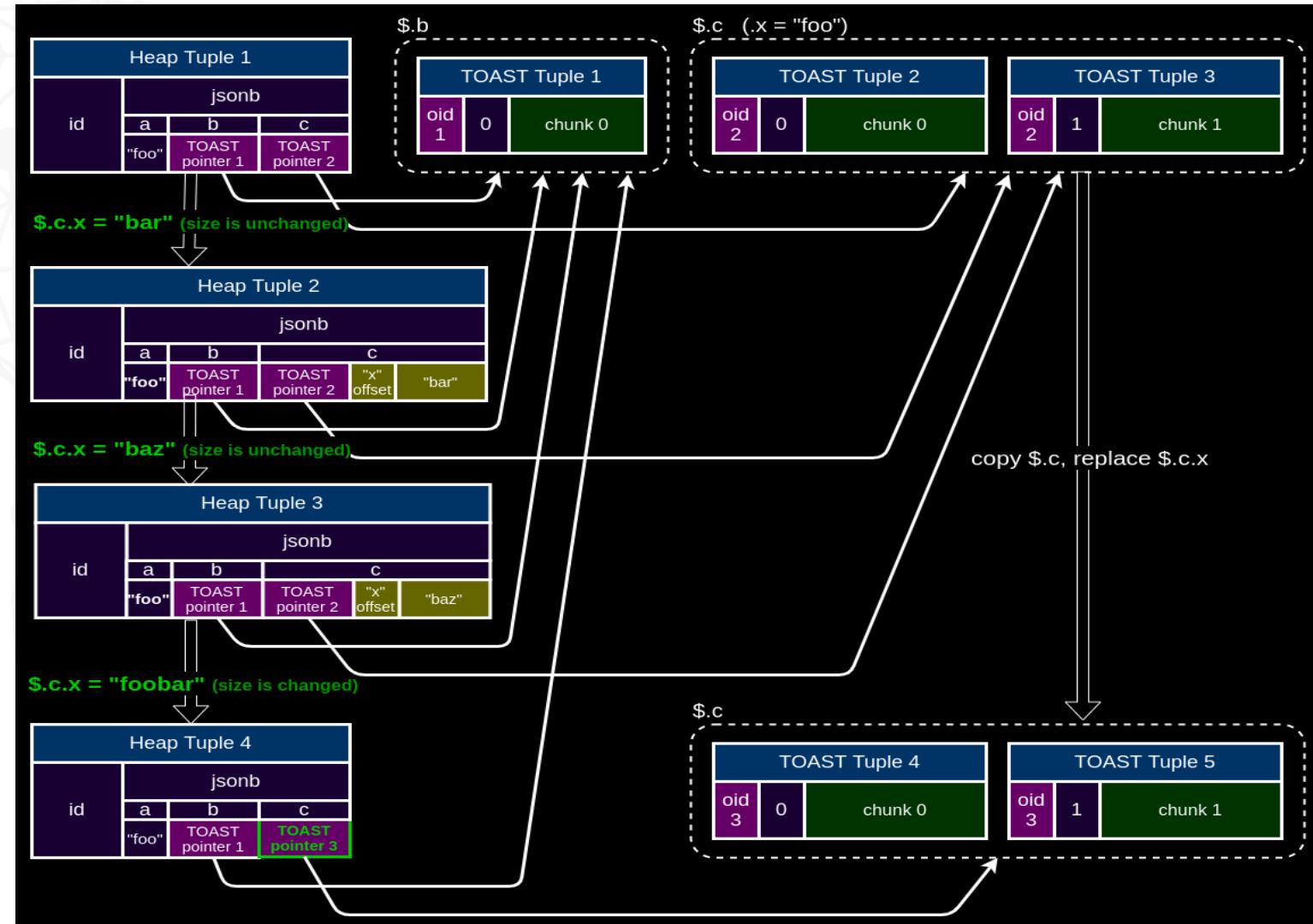
Shared TOAST – update

- When the short inline field is updated, only the new version of inline data is created.
- When some part of the long field is updated, the whole container is copied, updated and then TOASTed back with new oid (in the future oids can be shared).
- Unchanged TOASTed fields are always shared.



Shared TOAST – in-place updates

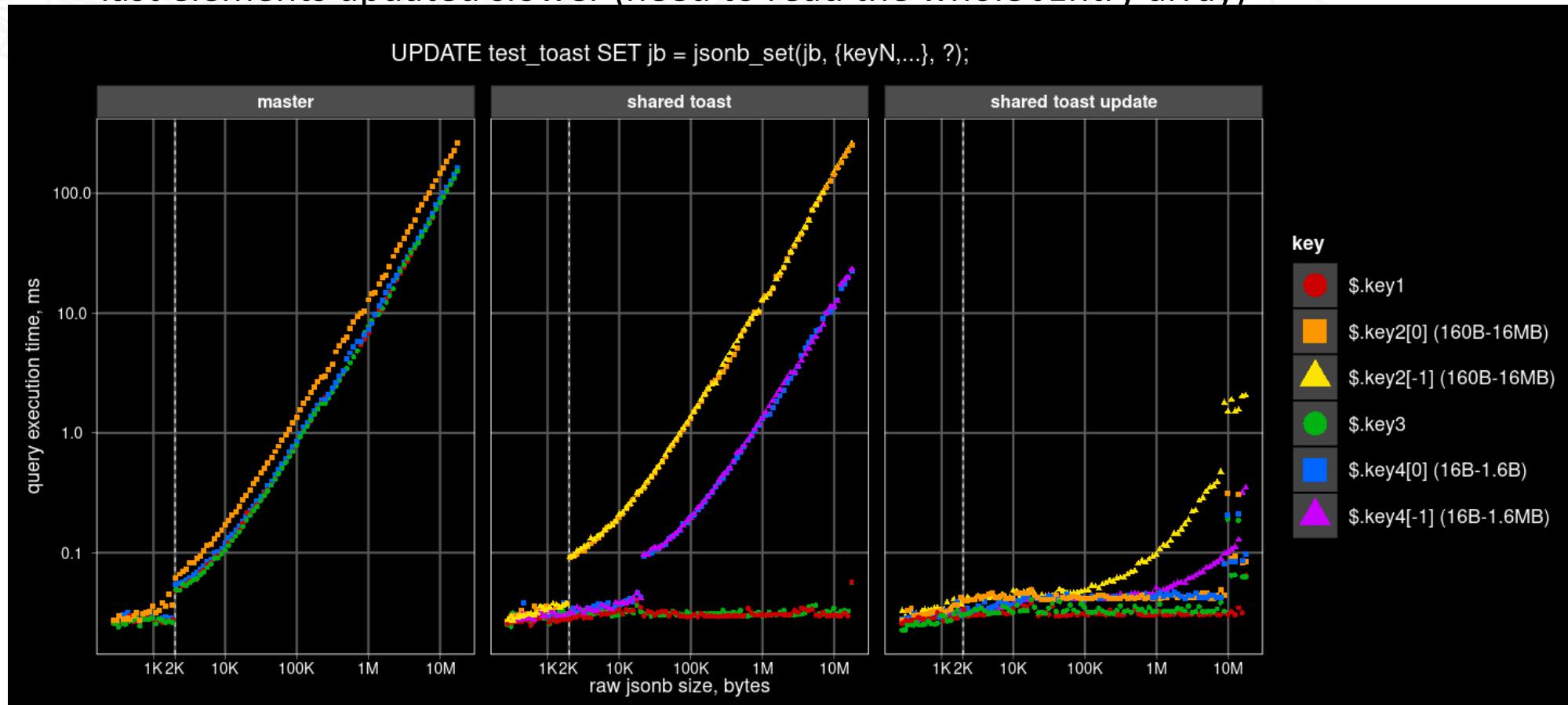
- Copying of shared TOASTs can be avoided when the size and type of updated part is not changed – there is no need to rewrite JEntries, only the value needs to be replaced
- `jsonb_set()` checks this special case accessing only the minimal header part needed for fetching offset, length and type of the old value
- If the length is not changed, created “diff” TOAST pointer with offset and new value



Shared TOAST – in-place update results (synthetic)

Update time of array elements depends on their position:

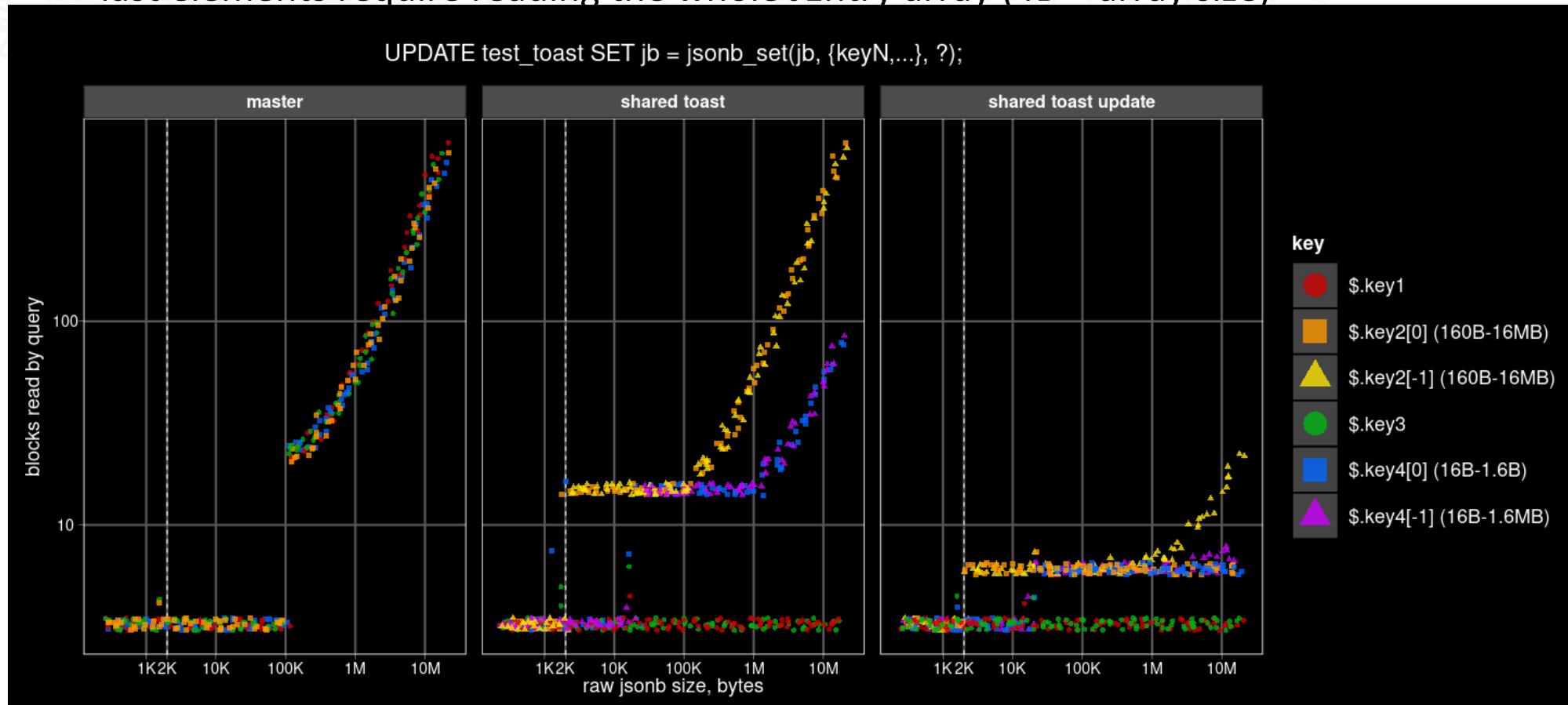
- first elements updated very fast (like inline fields)
- last elements updated slower (need to read the whole JEntry array)



Shared TOAST – in-place update results (synthetic)

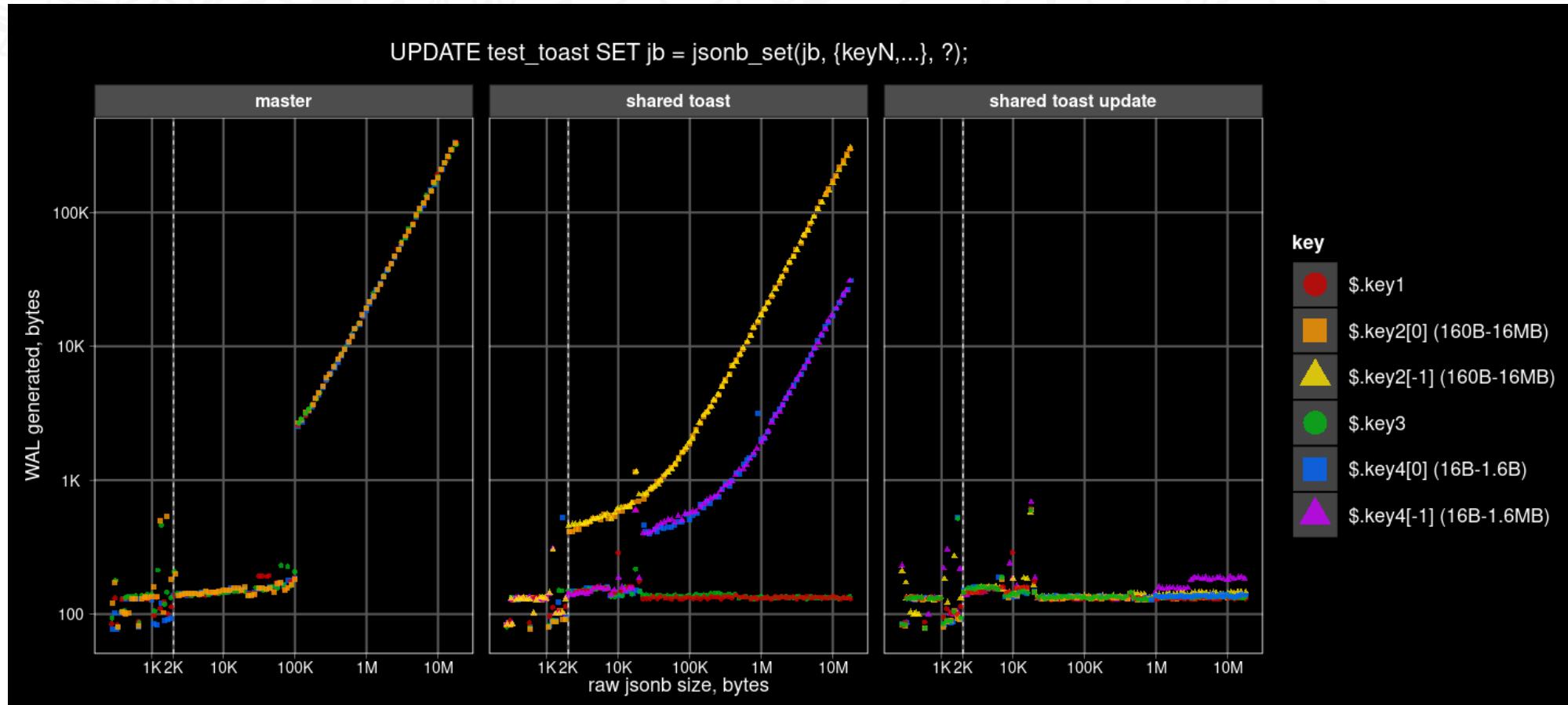
Number of blocks read depends on element position:

- first elements do not require reading of additional blocks
- last elements require reading the whole JEntry array ($4B * \text{array size}$)



Shared TOAST – in-place update results (synthetic)

- WAL size of in-place updates is almost independent on element position
- Only inline data with TOAST pointer diff are logged



JSONB vs Relational: access whole document

JSONB table – 25600 uncompressed arrays of various sizes (1 - 1000) with random string elements of various length (1-1000 bytes):

```
[{"id": 123, "val": "random string"}, ...]
```

```
CREATE TABLE test_jsonb_arrays (id int, array_size int, elem_size int, jb jsonb);
ALTER TABLE test_jsonb_arrays ALTER jb SET STORAGE external;
```

```
INSERT INTO test_jsonb_arrays
SELECT
    id + (array_size * 16 + elem_size) * 100 AS id,
    array_size,
    elem_size,
    obj AS jb
FROM
    generate_series(0, 15) array_size,
    generate_series(0, 15) elem_size,
    lateral (select jsonb_agg(
        jsonb_build_object('id', idx,
                           'val', random_string(pow(10, elem_size / 5.0)::int)))
        from generate_series(1, pow(10, array_size / 5.0)::int) idx
    ) o(obj),
    generate_series(0, 99) id;
```

```
CREATE INDEX ON test_jsonb_arrays (array_size, elem_size);
CREATE INDEX ON test_jsonb_arrays (id);
```

JSONB vs Relational: access whole document

Two relational tables – the first for arrays, the second for their elements:

```
CREATE TABLE test_jsonb_arrays_rel (id int, array_size int, elem_size int);
CREATE TABLE test_jsonb_arrays_rel_elems (id int, idx int, val text);

INSERT INTO test_jsonb_arrays_rel
SELECT
    id + (array_size * 16 + elem_size) * 100 AS id,
    array_size,
    elem_size
FROM
    generate_series(0, 15) array_size,
    generate_series(0, 15) elem_size,
    generate_series(0, 99) id;

INSERT INTO test_jsonb_arrays_rel_elems
SELECT
    id + (array_size * 16 + elem_size) * 100 AS id,
    idx,
    val
FROM
    generate_series(0, 15) array_size,
    generate_series(0, 15) elem_size,
    generate_series(0, pow(10, array_size / 5.0)::int - 1) idx,
    random_string(pow(10, elem_size / 5.0)::int) val,
    generate_series(0, 99) id;

CREATE INDEX ON test_jsonb_arrays_rel (array_size, elem_size);
CREATE INDEX ON test_jsonb_arrays_rel (id);
CREATE INDEX ON test_jsonb_arrays_rel_elems (id, idx);
```

JSONB vs Relational: access whole document

- JSONB document extraction in 3 variants:

```
SELECT jb FROM test_jsonb_arrays WHERE array_size = $1 AND elem_size = $2;
```

```
SELECT textsend(jb::text) ... -- plain text format
```

```
SELECT ubjson_send(jb::ubjson) ... -- binary ubjson format
```

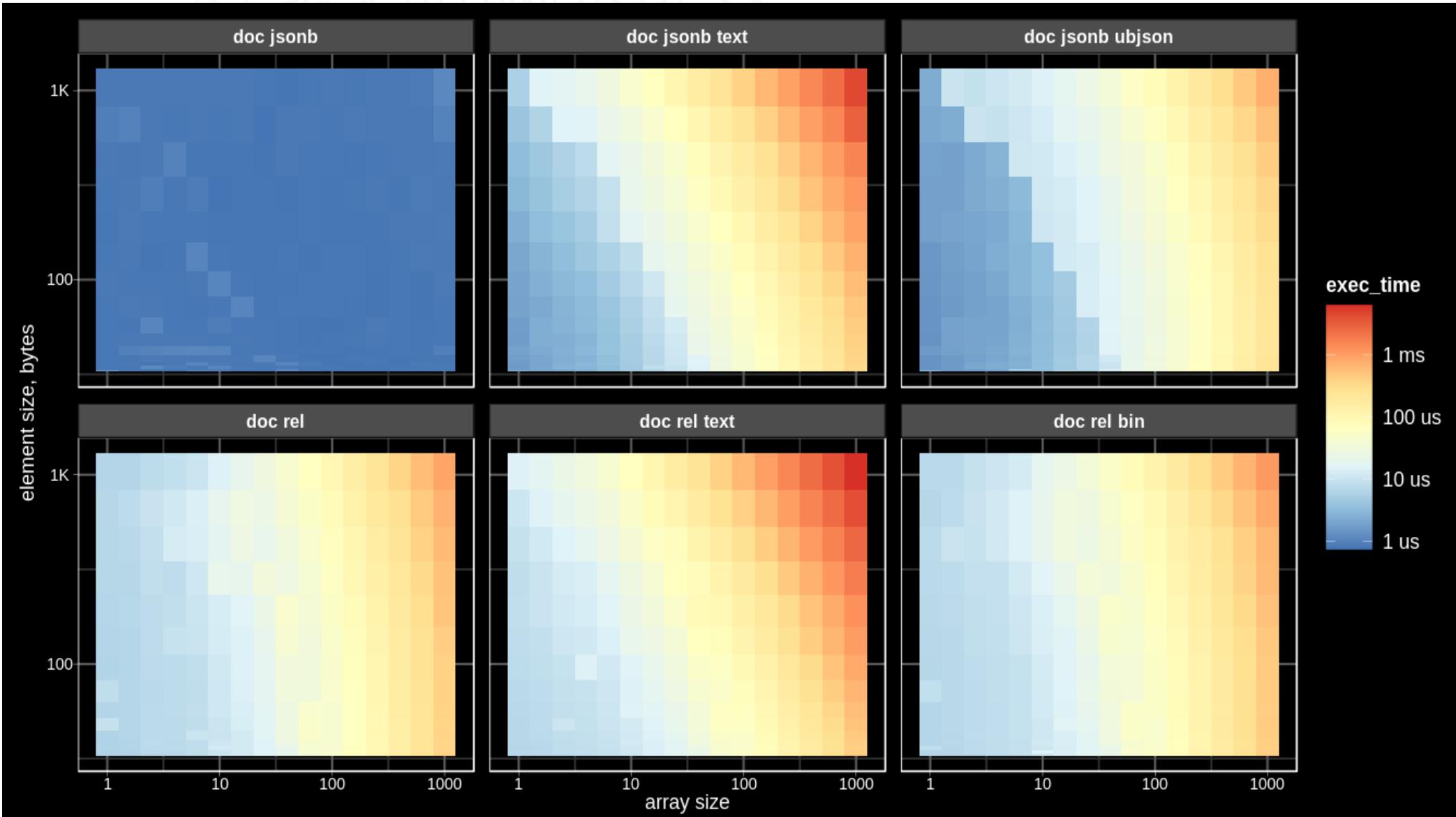
- Relational join with aggregation to array in 3 variants:

```
SELECT (SELECT array_agg(e.val)
        FROM test_jsonb_arrays_rel_elems e
        WHERE e.id = a.id)
FROM test_jsonb_arrays_rel a
WHERE array_size = $1 AND elem_size = $2;
```

```
SELECT textsend(SELECT array_agg(e.val) ...) ... -- plain text format
```

```
SELECT array_send(SELECT array_agg(e.val) ...) ... -- binary format
```

JSONB vs Relational: access whole document



JSONB vs Relational: access key, update

JSONB table – uncompressed objects of various sizes (up to 1.4MB) with 10 random string keys of various length (up to 1MB):

```
key 1 length: 100 B - 1 MB  
key 2 length: 30 B - 300 KB  
key 3 length: 10 B - 100 KB  
...  
.
```

```
CREATE TABLE test_jsonb_object (id int, size int, level int, jb jsonb);  
ALTER TABLE test_jsonb_object ALTER jb SET STORAGE external;
```

```
INSERT INTO test_jsonb_object  
SELECT id + size * 100 AS id, size, obj AS jb  
FROM  
    generate_series(20, 60) size,  
    LATERAL (  
        SELECT jsonb_object_agg('key' || k,  
            jsonb_build_array(random_string(pow(10, size / 10.0 - (k - 1) / 2.0)::int)))::text  
        FROM generate_series(1, 10) k  
    ) o(obj),  
    generate_series(0, 99) id;
```

```
CREATE INDEX ON test_jsonb_object (size);  
CREATE INDEX ON test_jsonb_object (id);
```

JSONB vs Relational: access key, update

Relation table with 10 key columns:

```
CREATE TABLE test_jsonb_object_rel AS
SELECT
    id + size * 100 id,
    size,
    arr[1] key1,
    arr[2] key2,
    arr[3] key3,
    arr[4] key4,
    arr[5] key5,
    arr[6] key6,
    arr[7] key7,
    arr[8] key8,
    arr[9] key9,
    arr[10] key10
FROM
    generate_series(20, 60) size,
    LATERAL (SELECT array_agg(random_string(pow(10, size / 10.0 - (k - 1) * 0.5)::int))
              FROM generate_series(1, 10) k a(arr),
    generate_series(0, 99) id;

CREATE INDEX ON test_jsonb_object_rel (size, level);
CREATE INDEX ON test_jsonb_object_rel (id);
```

JSONB vs Relational: access key, update

- Select single key:

```
SELECT textsend(jb #>> '{key$1,0}' ) FROM test_jsonb_object      WHERE size = $2;  
SELECT textsend(key$1)                      FROM test_jsonb_object_rel WHERE size = $2;  
  
$1 = 1-10 (key)  
$2 = 20-60 (size)
```

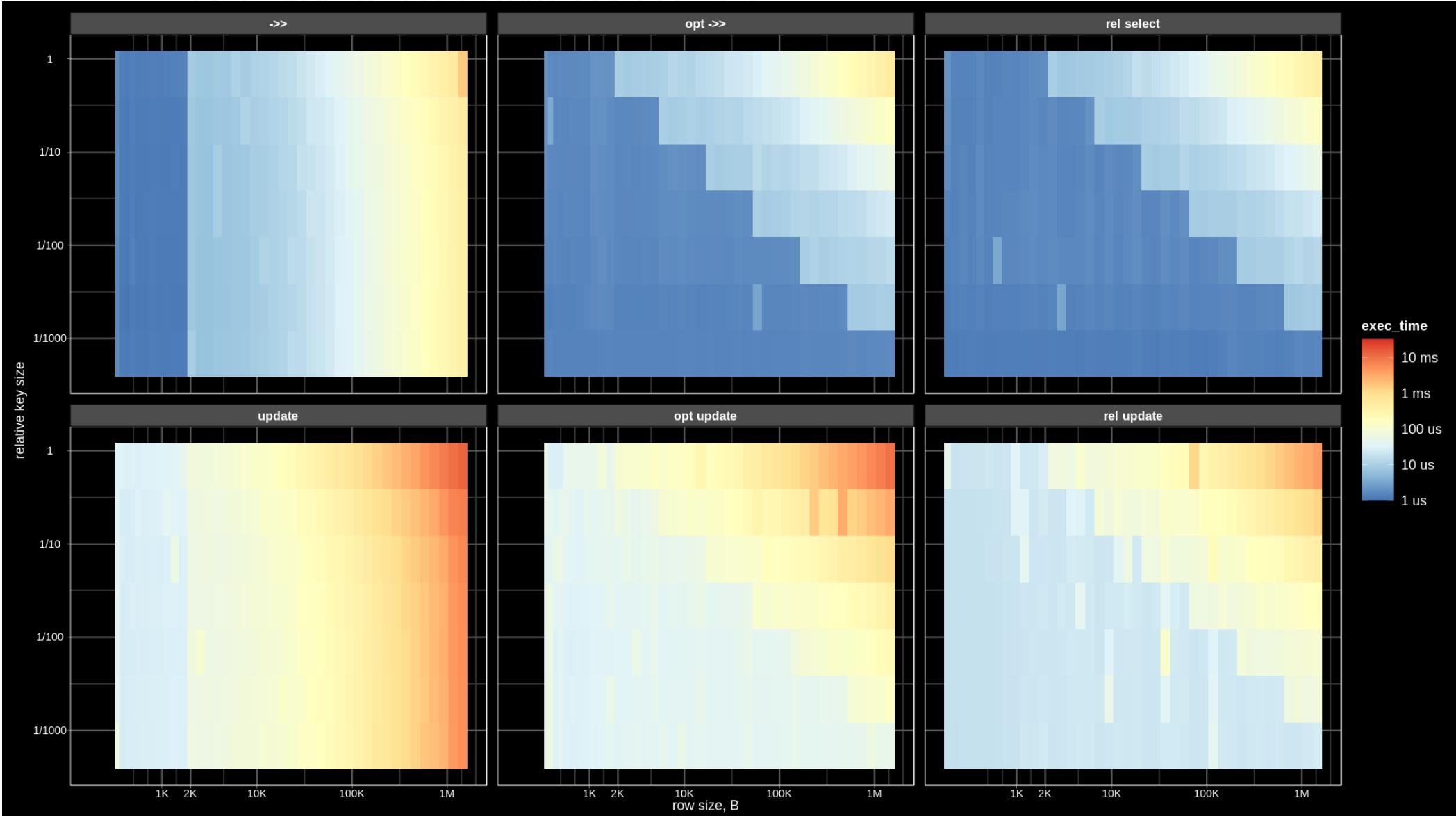
- Update single key and commit, repeat 100 times, key length not changed:

```
UPDATE test_jsonb_object  
SET jb = jsonb_set(jb, '{key$1,0}', to_jsonb($2))  
WHERE id = $3;
```

```
UPDATE test_jsonb_object_rel  
SET key$1 = $2  
WHERE id = $3;
```

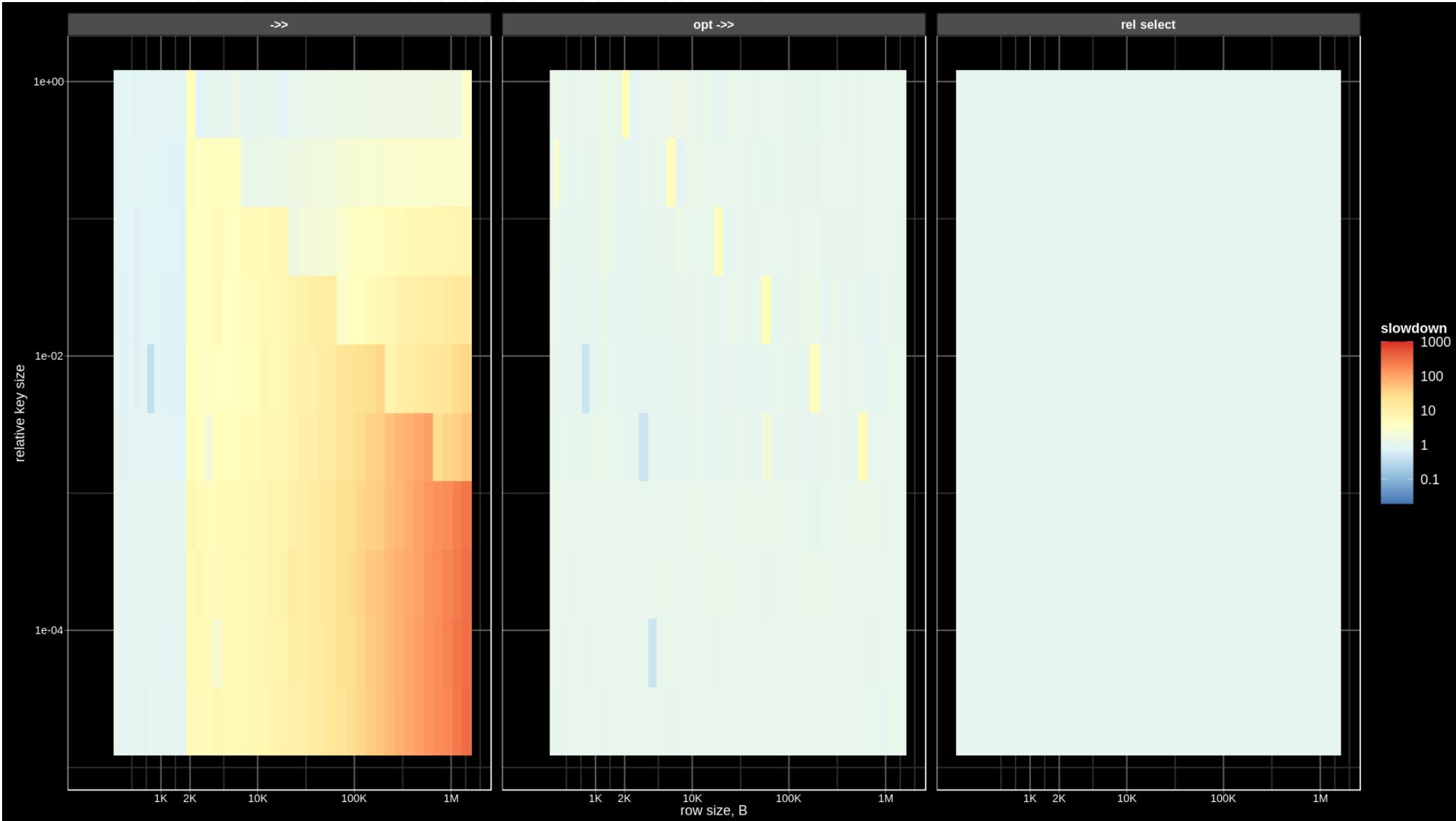
```
$1 = 1-10 (key)  
$2 = random_string(pow(10, size / 10.0 - (key - 1) / 2.0)::int))  
$3 = size * 1000
```

JSONB vs Relational: access key, update



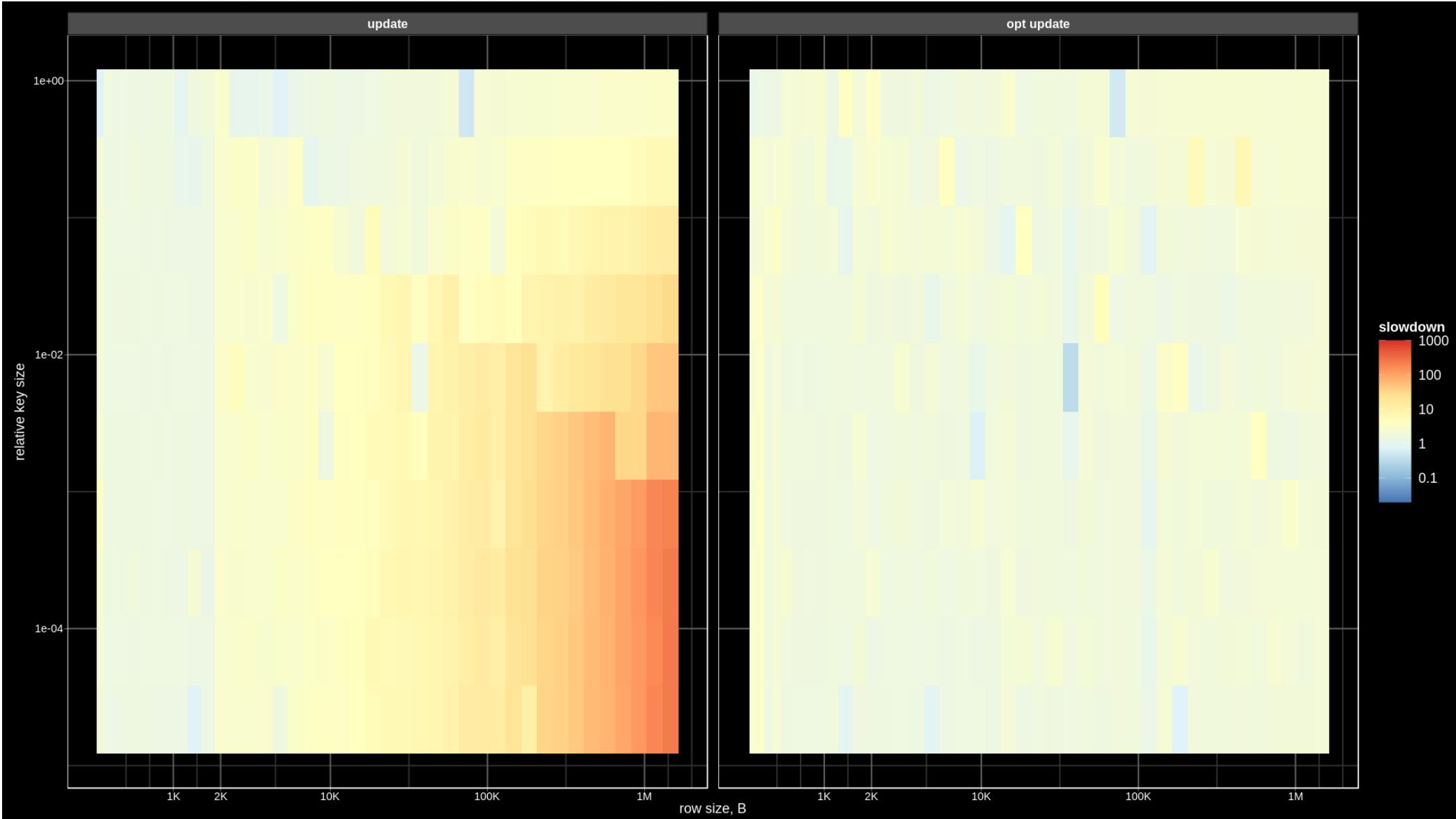


JSONB vs Relational: access key slowdown



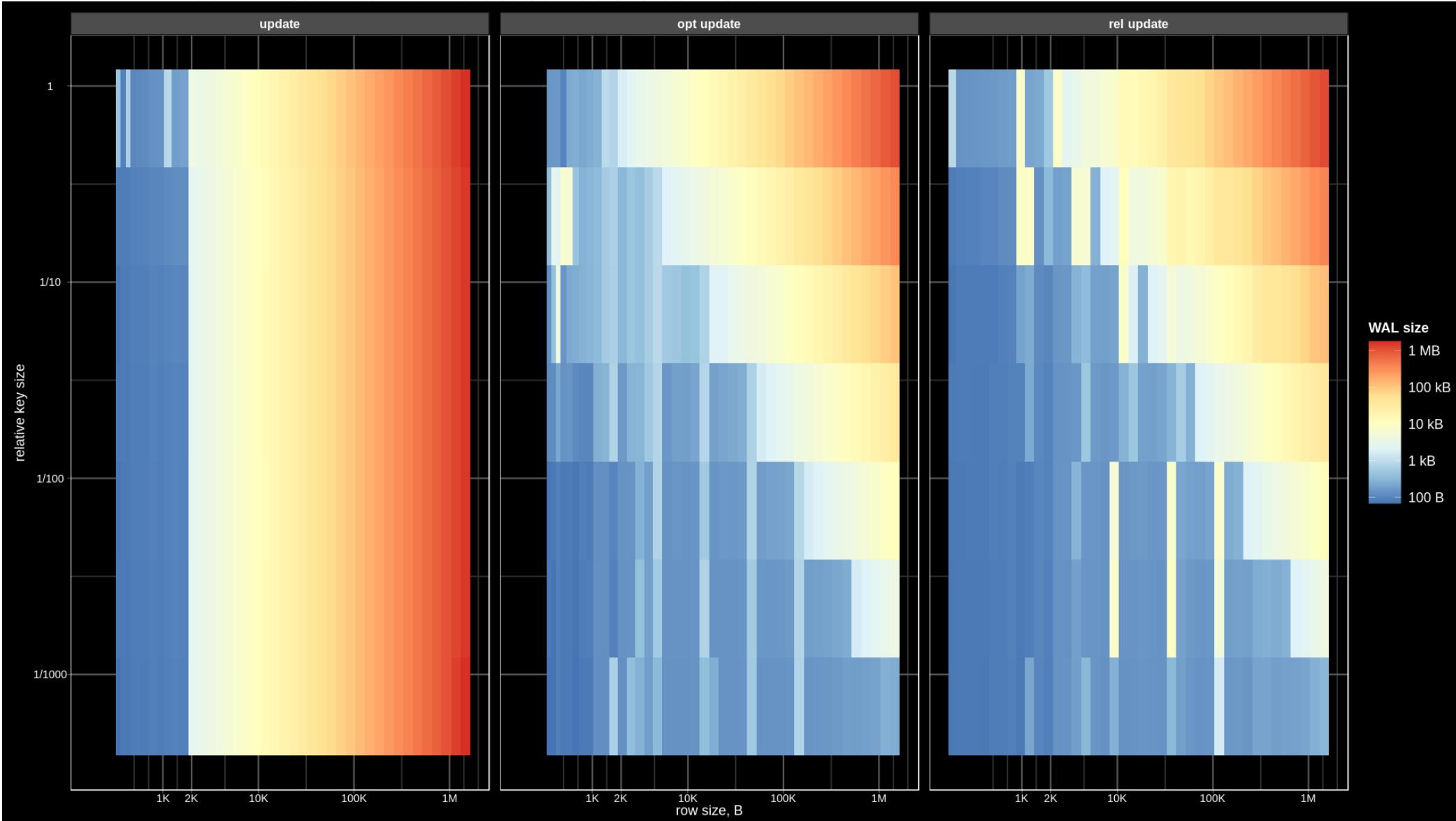


JSONB vs Relational: update key slowdown





JSONB vs Relational: update key WAL



JSONB vs Relational: access array member

- JSONB:

```
SELECT textsend(jb #>> ARRAY[$1::text, 'val'])
FROM test_jsonb_arrays
WHERE array_size = $2 AND elem_size = $3;
```

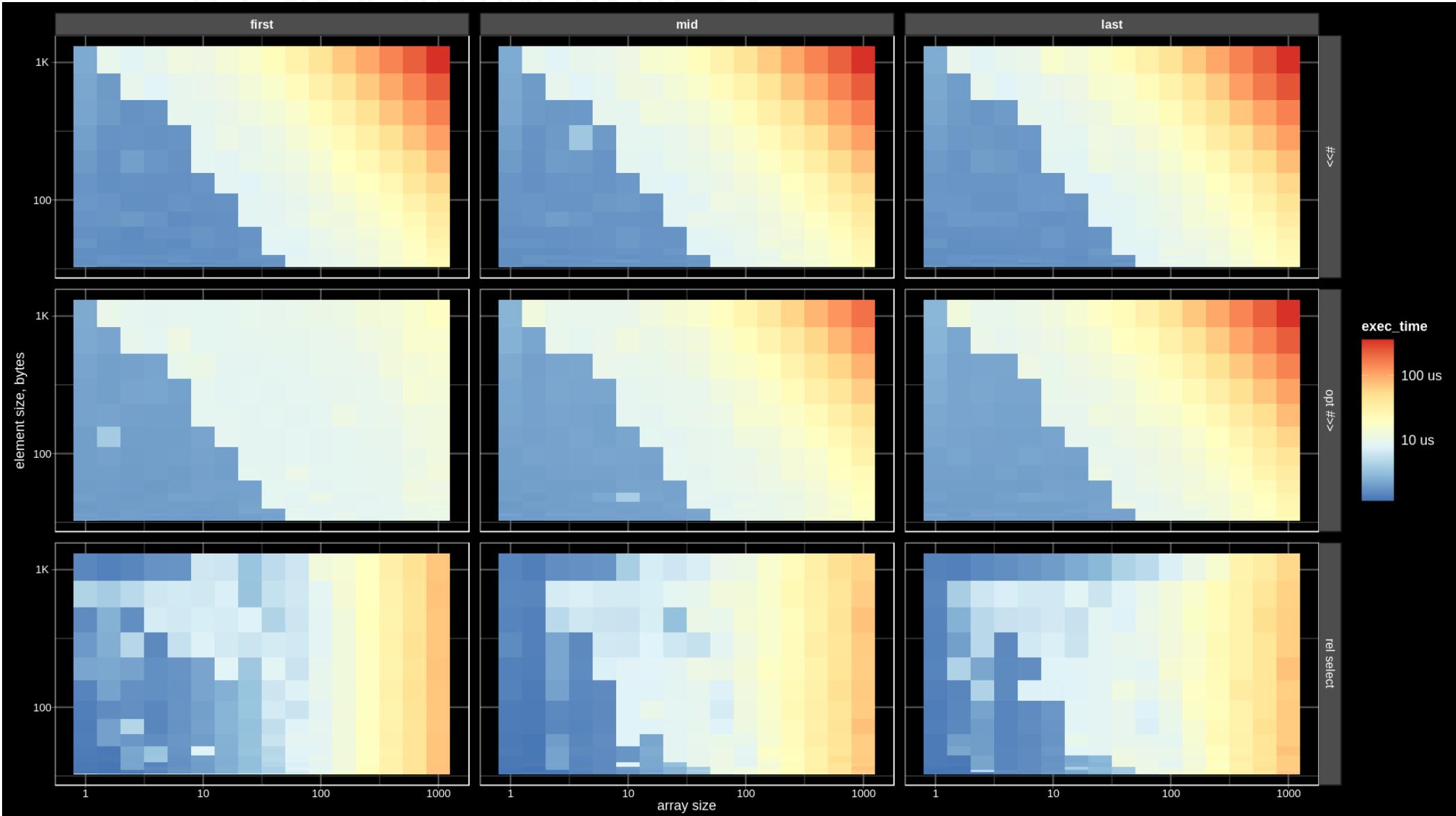
- Relational:

```
SELECT textsend(val)
FROM test_jsonb_arrays_rel_elems
WHERE idx = $1 AND array_size = $2 AND elem_size = $3;
```

- \$1 =

```
first  => 0
middle => array_length / 2
last   => array_length - 1
```

JSONB vs Relational: access array member



JSONB vs Relational: update array member

- JSONB:

```
UPDATE test_jsonb_arrays
SET jb = jsonb_set(jb, ARRAY[$1::text, 'val'], $3)
WHERE id = $2;
```

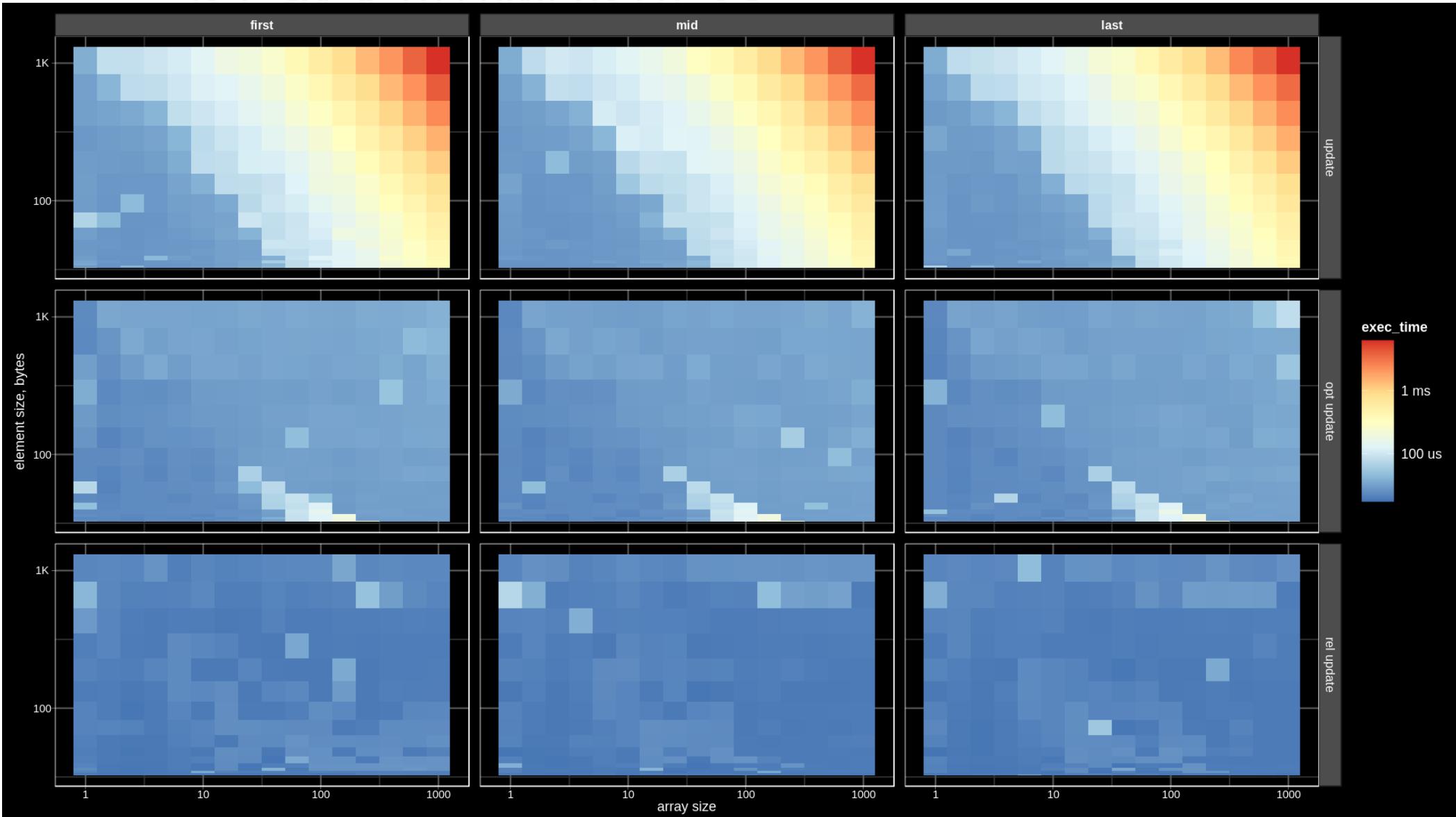
- Relational:

```
UPDATE test_jsonb_arrays_rel_elems
SET val = $3
WHERE id = $2 AND idx = $1;
```

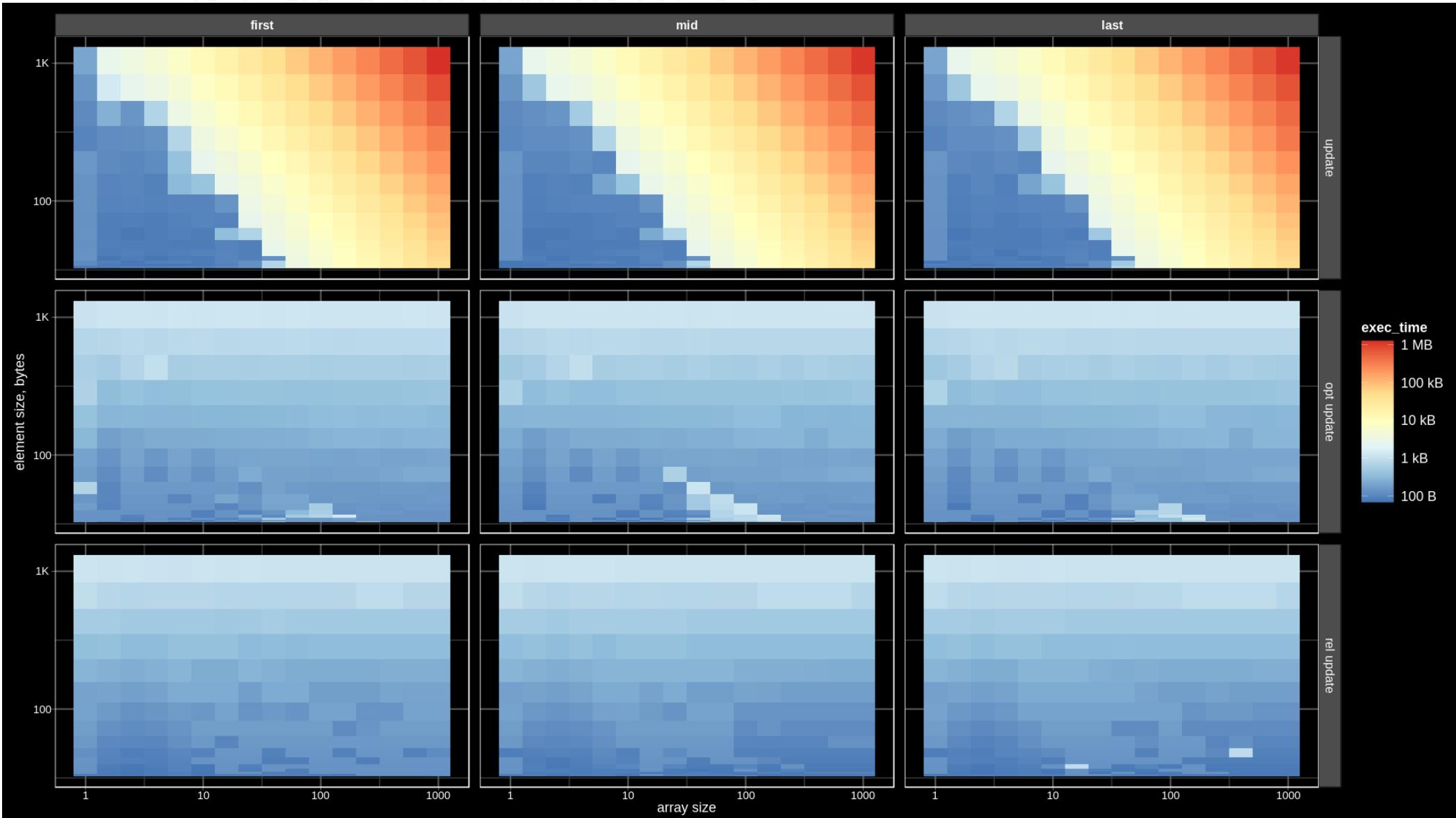
- \$1 =

```
first  => 0
middle => array_length / 2
last   => array_length - 1
```

JSONB vs Relational: update array member



JSONB vs Relational: WAL update array member



Conclusions

- JSONB is good
 - Full object access (microservices) — faster than relational way (joins, aggregate, difficult tuning)
 - Storing short metadata as a separate jsonb field
 - Ubiquitous format for data interchange, storing API messages (XML is too much)
 - Simple database design (simple queries), support of Agile development
 - Data migration (schema evolution). Old applications can easily accept new data.
 - Client app, backend, database — one format, all server side languages support JSON, now SQL supports JSON, JSON relaxed code-centric vs data-centric
- Currently not optimized for
 - TOASTed jsonb (updates)
 - Access to array members
- There are promising results !

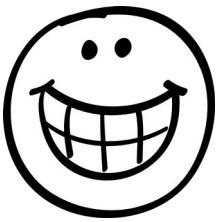
TODO

- Data type aware TOAST
- Better indexing — index only specified subset of json
<https://www.pgcon.org/2018/schedule/events/1169.en.html>
- JSON Schema - required fields, optional fields, data types, value constraints,...
- JSONB Access Method — graph-like access of k-v level ?

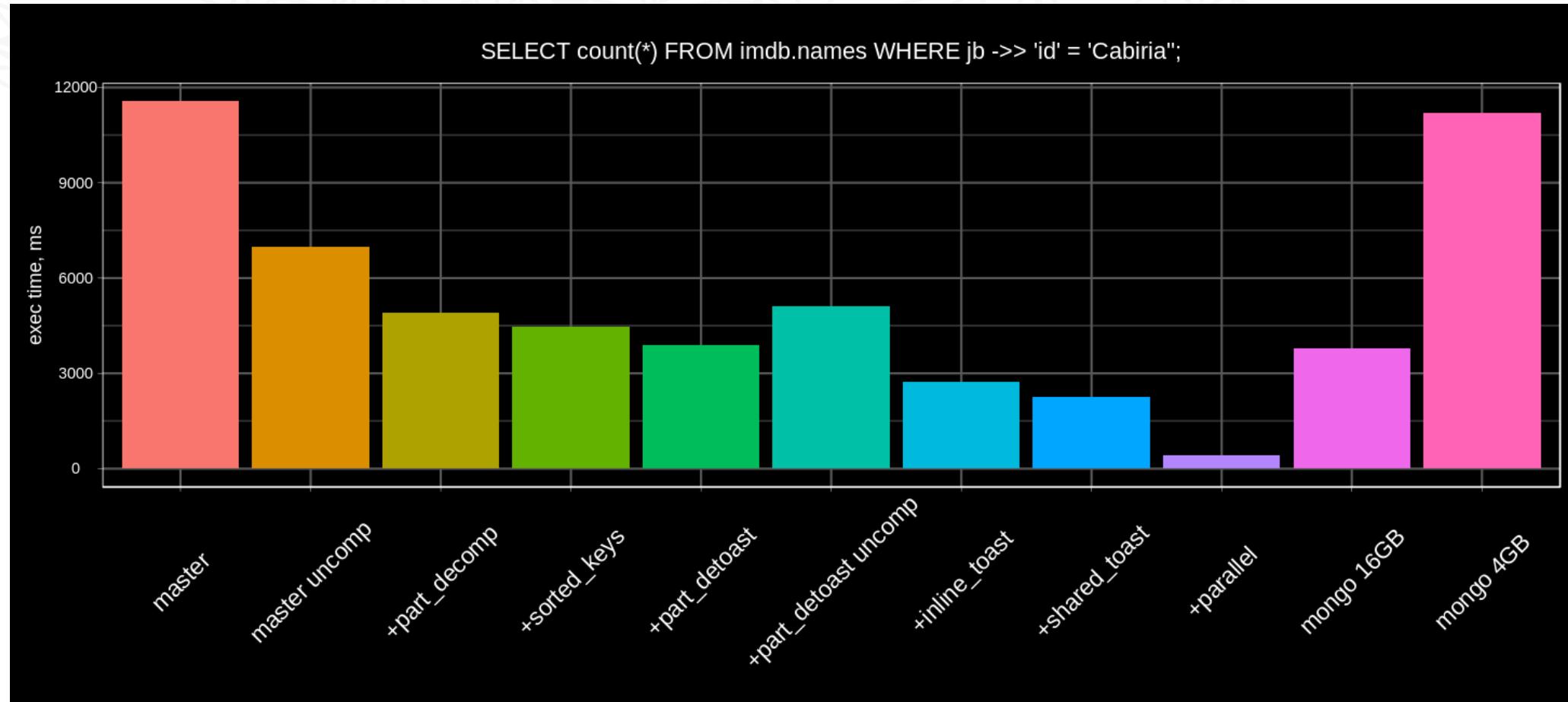
References

- A sequence of rather simple and straightforward algorithms and storage optimizations based on GSON API, without any major changes to the JSONB API, have lead to significant performance improvements (10X speedup for SELECT and much cheaper UPDATEs):
 - Details - <http://www.sai.msu.su/~megera/postgres/talks/jsonb-pgvision-2021.pdf>
 - Slides of this talk ([PDF](#))
- Jsonb is ubiquitous and is continuously developing
 - [JSON\[B\] Roadmap V2, Postgres Professional Webinar, Sep 17, 2020](#)
 - [JSON\[B\] Roadmap V3, Postgres Build 2020, Dec 8, 2020](#)

Non-scientific comparison PG vs Mongo



- Seqscan, PG - in-memory, Mongo (4.4.4): 16Gb (in-memory), 4GB (1/2)



ALL

YOU

NEED
POSTGRES

US

