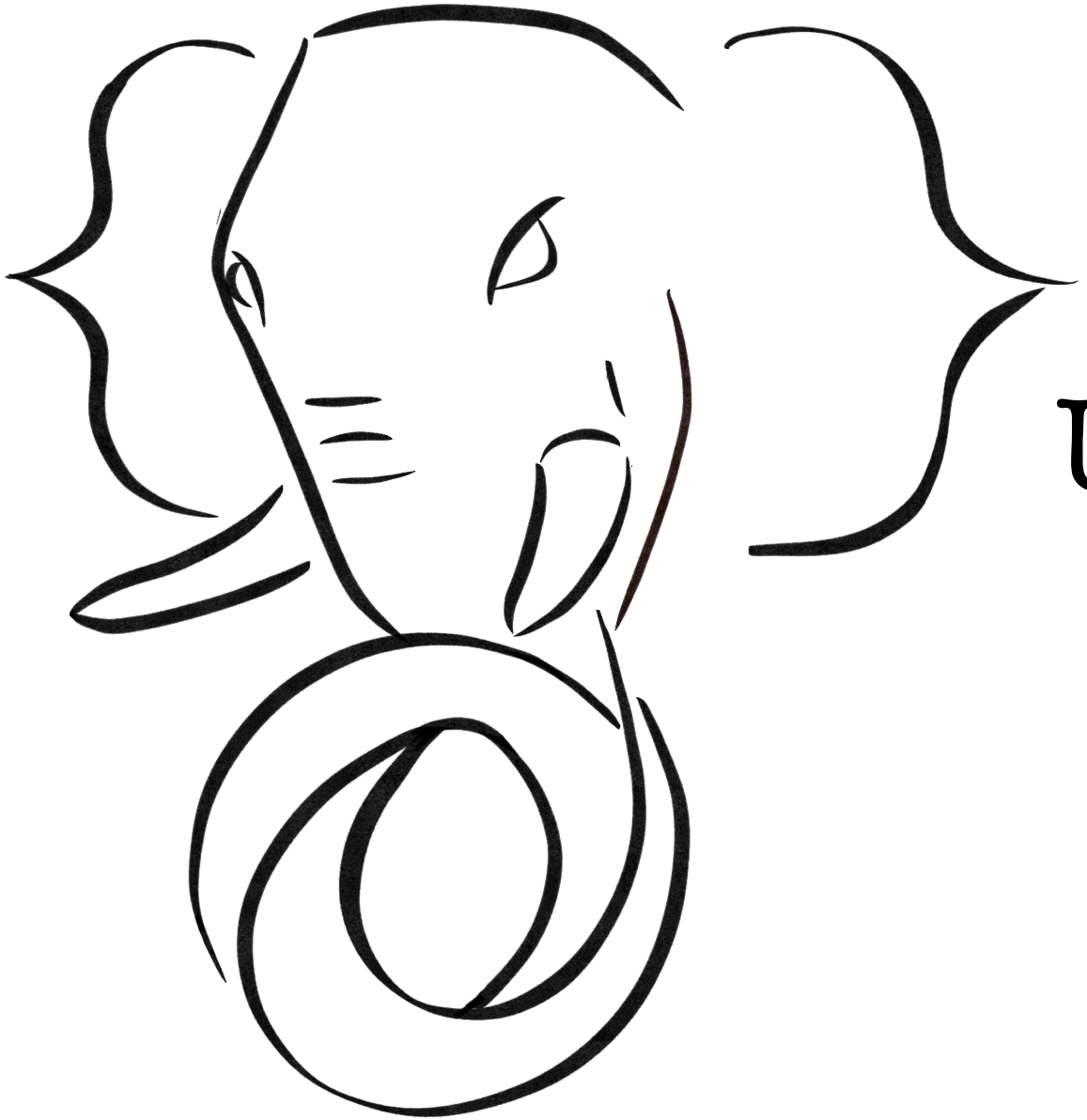


PGCONG.NYC 2021

Understanding
JSONB Performance

Oleg Bartunov
Nikita Glukhov

Postgres PROFESSIONAL

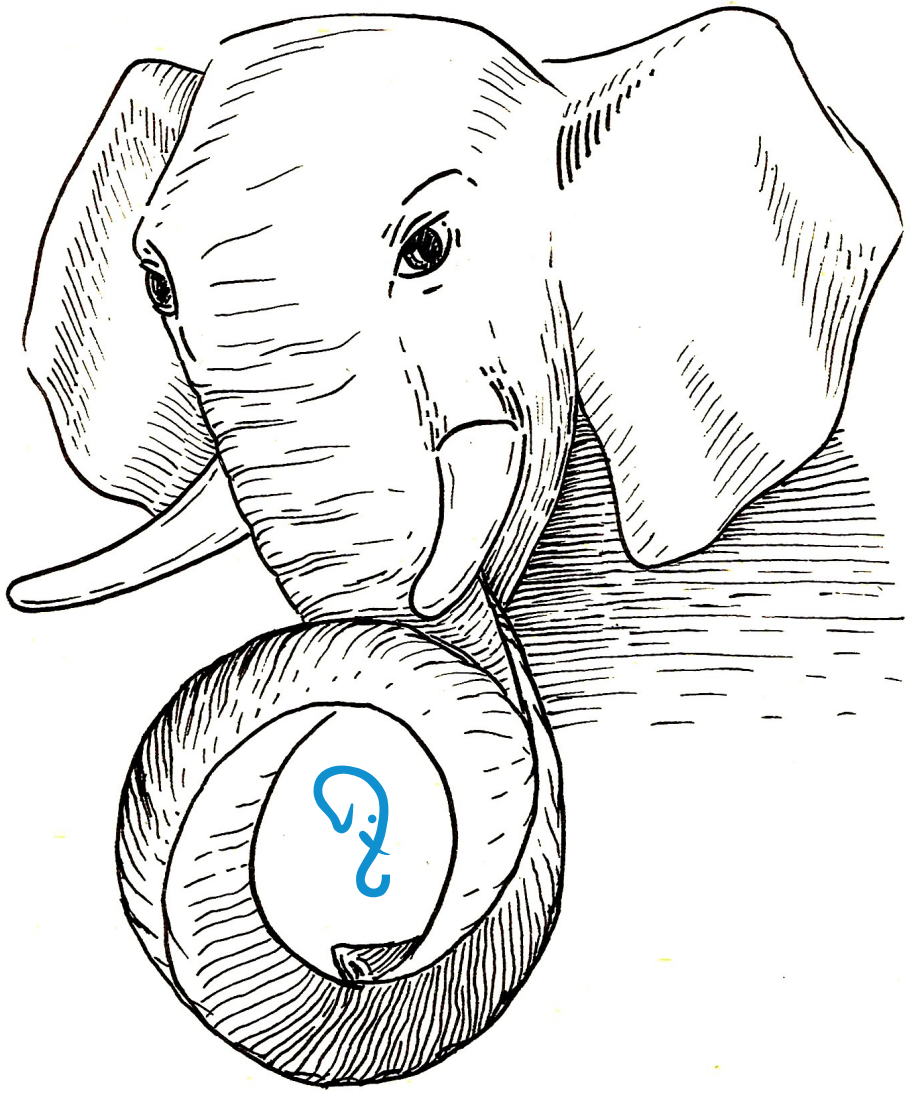
http://www.sai.msu.su/~megera/postgres/talks/jsonb-pgconfnyc-2021.pdf

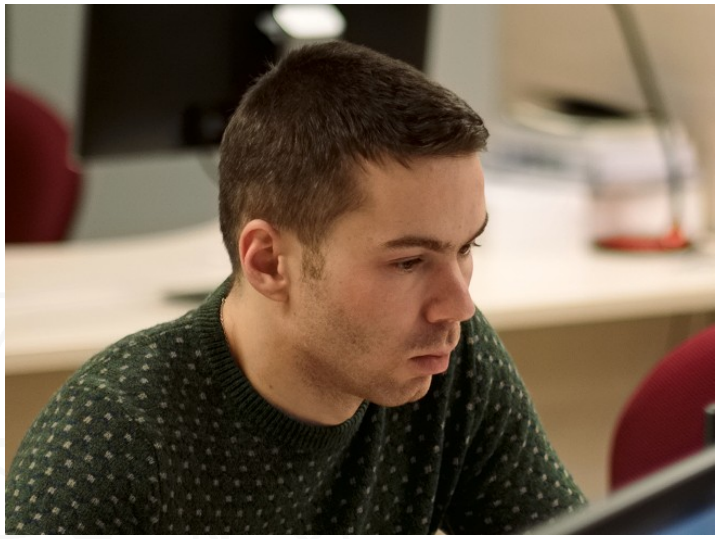
Since Postgres95

Research scientist @
Moscow University
CEO Postgres Professional
Major PostgreSQL contributor

PostgresPro

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

PostgresPro

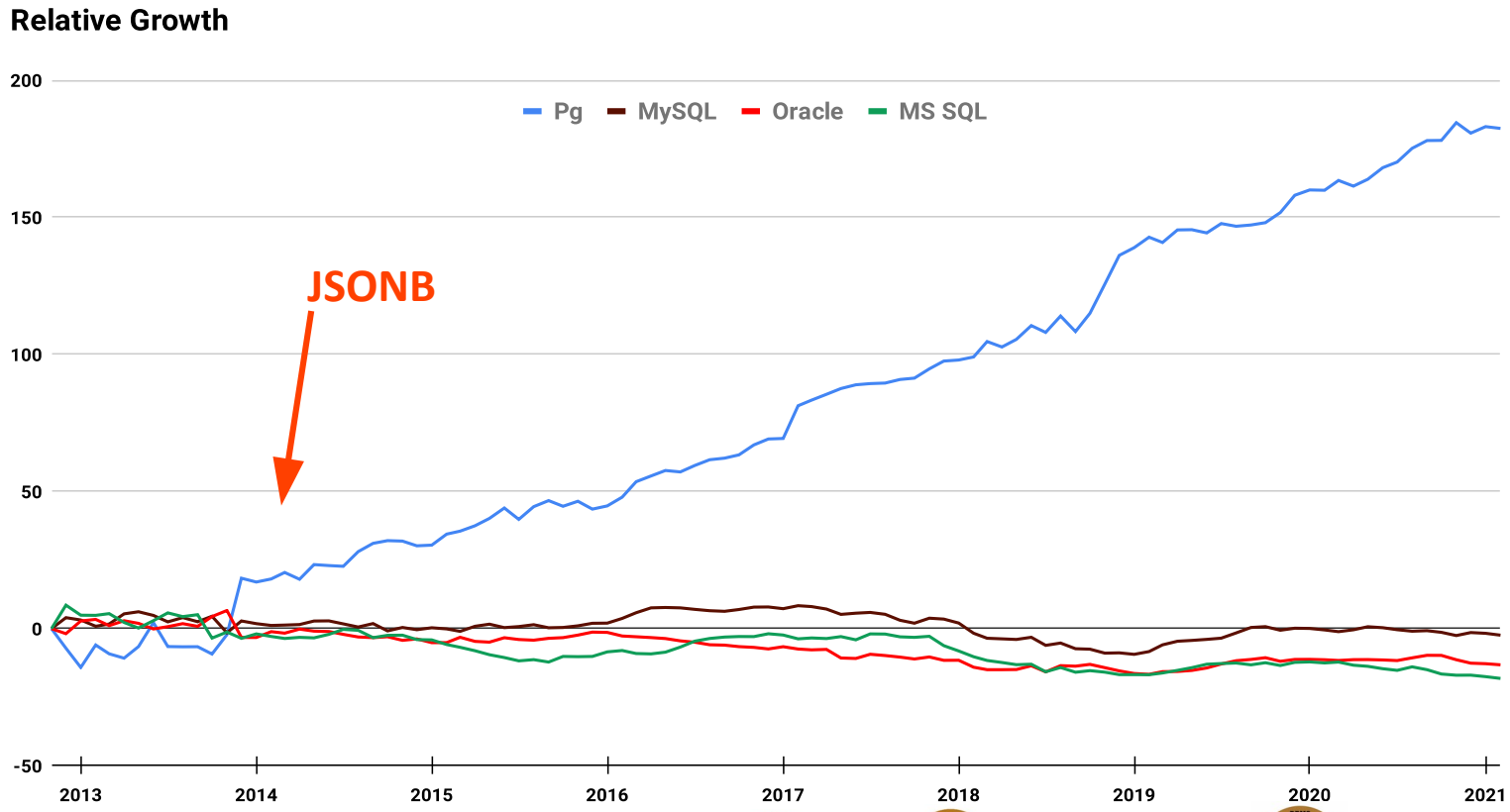
Why this talk ?

- Blossom of Microservice architecture, startups need JSON
- One-Type-Fits-All
  - Client app — Backend - Database use JSON
  - All server side languages support JSON, now SQL support JSON
  - JSON relaxed ORM (Object-Relational Mismatch), mitigate contradictions between code-centric and data-centric paradigms.
- ~~Gold~~ Jsonb Rush
  - JSONB is one of the main driver of Postgres popularity
  - CREATE TABLE … (jb JSONB); – common mistake to put everything into JSONB
  - Need comparison of performance of jsonb operators and roadmap
- My 20+ interest and experience in extending Postgres to support semi-structural data: arrays, hstore (2003), full text search, index AM (GiST, GIN, SP-GiST), JSONB (2014), SQL/JSON

PostgresPro

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

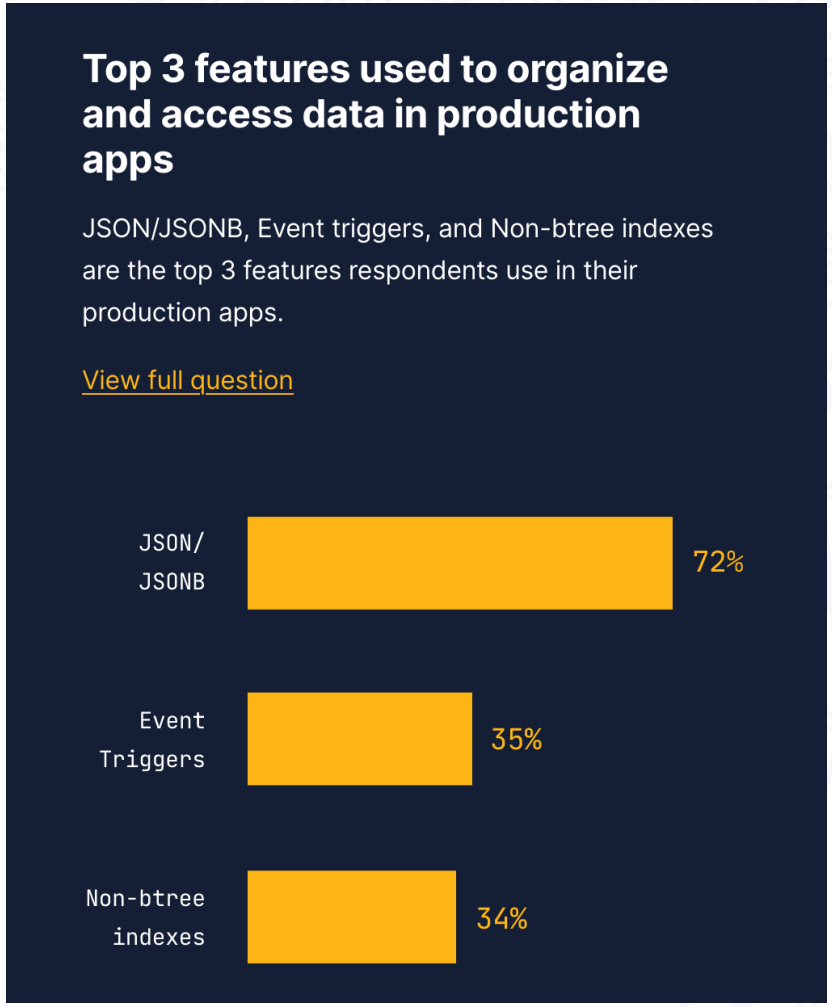


db-engines.com/en/ranking

PostgresPro

JSONB Popularity

State of PostgreSQL 2021 (Survey)



Pgsql telegram (6170) — 26.02.2021
https://t.me/pgsql

- SELECT 8061
- SQL 4473
- JSON[B] 3116
- TABLE 2997
- JOIN 2345
- INDEX 1519
- BACKUP 1484
- VACUUM 1470
- REPLICA 707

PostgresPro

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 (Jsquery 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*

PostgresPro

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 (Jsquery 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

PostgresPro

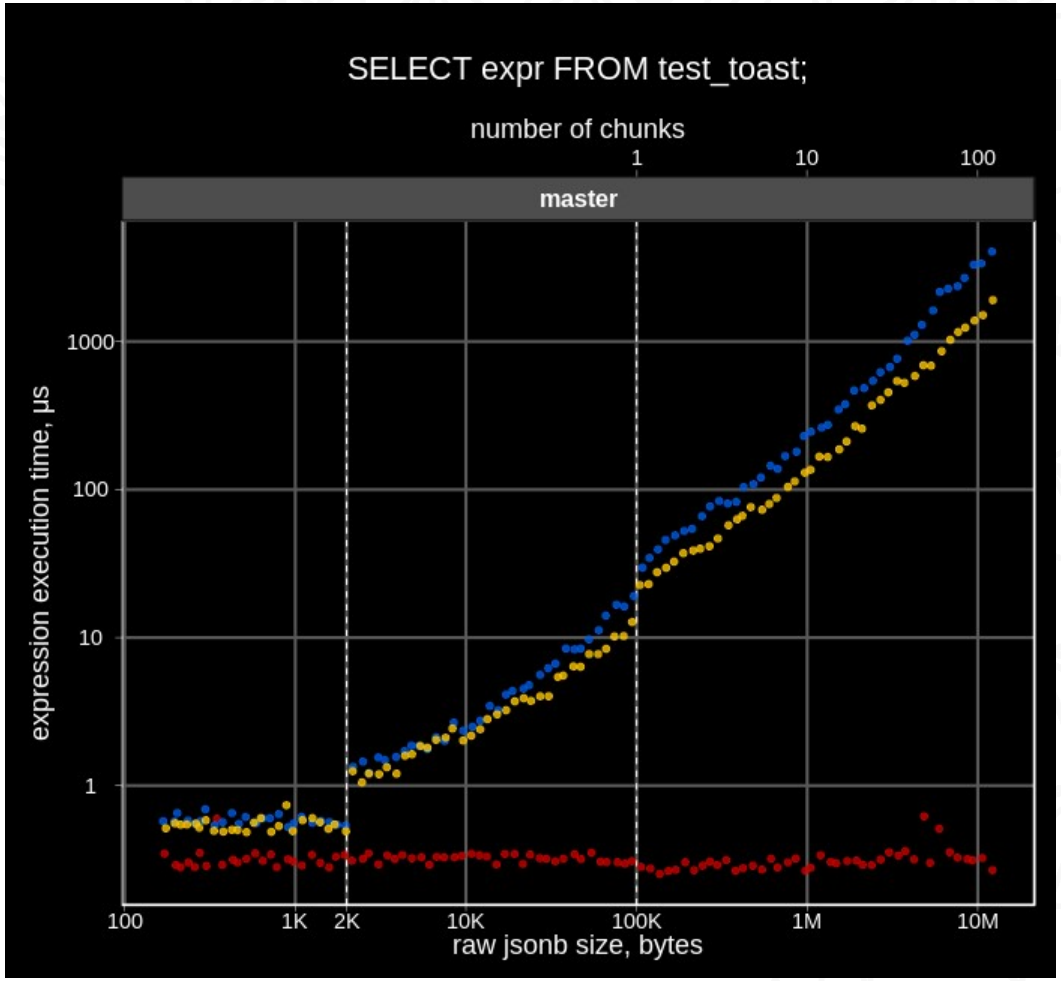
Top-priority: JSONB - 1st-class citizen in Postgres

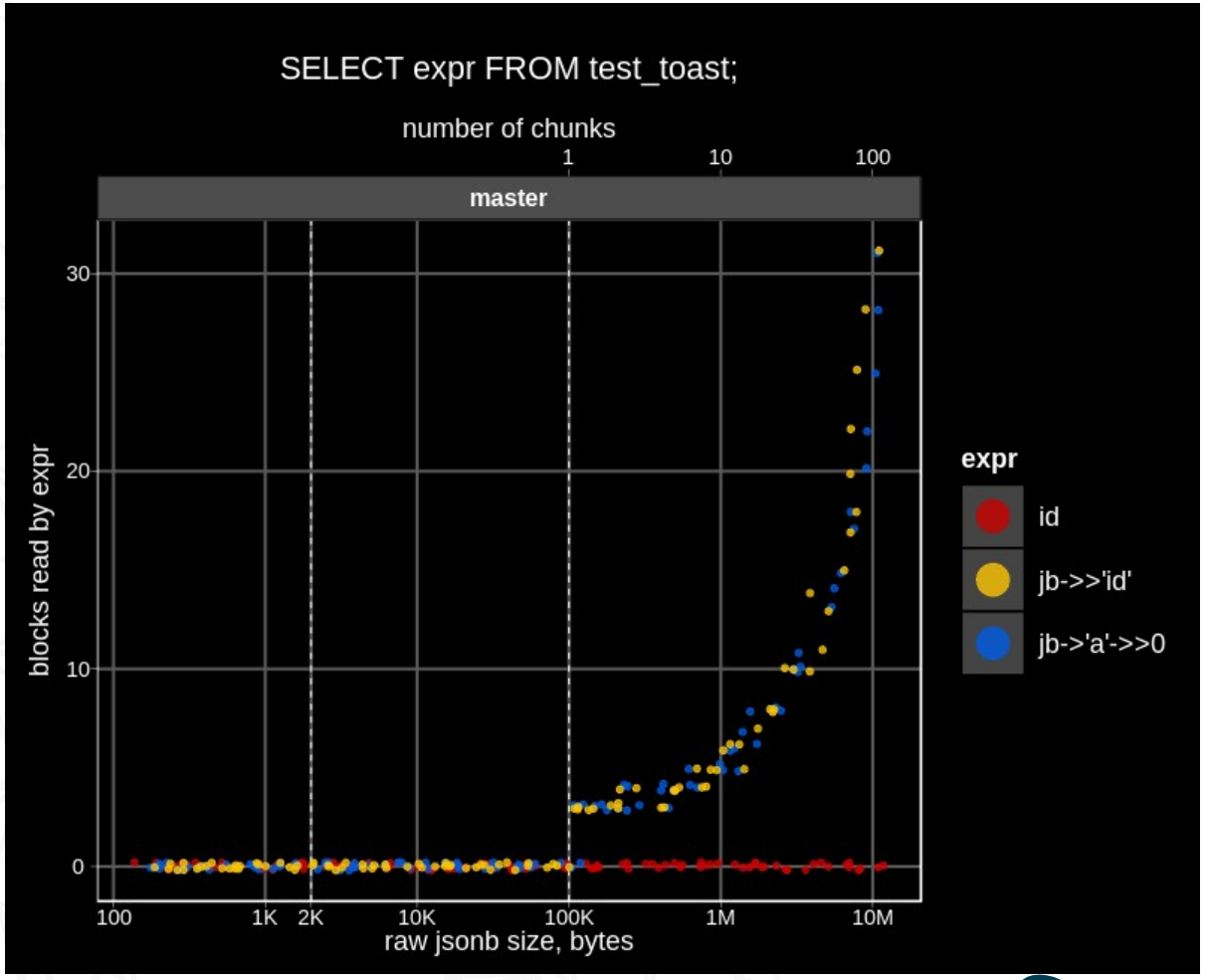
- Popularity of JSONB — it's mature data type, rich functionality
- Startups use Postgres and don't care about compatibilty to Oracle/MS SQL
  - Jsonpath is important and committed
  - There is rich user API to Jsonb, so SQL/JSON functions are not in top-priority list
- Not enough resources in community (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
  - IS JSON is important for validation using JSON Schema !

PostgresPro

Popular mistake: CREATE TABLE qq (jsonb)

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





Large jsonb is TOASTed !

PostgresPro

Nested containers performance, which is the best

- Sample table with nested objects of various sizes (1 – 1MB) and various nesting levels (0 – 9) with one short and one long key:

```
CREATE TABLE test_jsonb_nesting AS
SELECT id / 10 size, id % 10 level,
    (repeat('{"obj": ', id % 10) ||
     jsonb_build_object('key', id, 'long_str', str) ||
     repeat('}', id % 10))::jsonb jb
FROM
  generate_series(0, 1200) id,
  repeat('a', pow(10, id / 200.0)::int) str;
```

```
{"key": 10, "long_str": "a"}
{"obj": {"key": 11, "long_str": "a"}}
{"obj": {"obj": {"key": 12, "long_str": "a"}}}
{"obj": {"obj": {"obj": {"key": 13, "long_str": "a"}}}}
{"obj": {"obj": {"obj": {"obj": {"key": 14, "long_str": "a"}}}}}
```

- Test query:

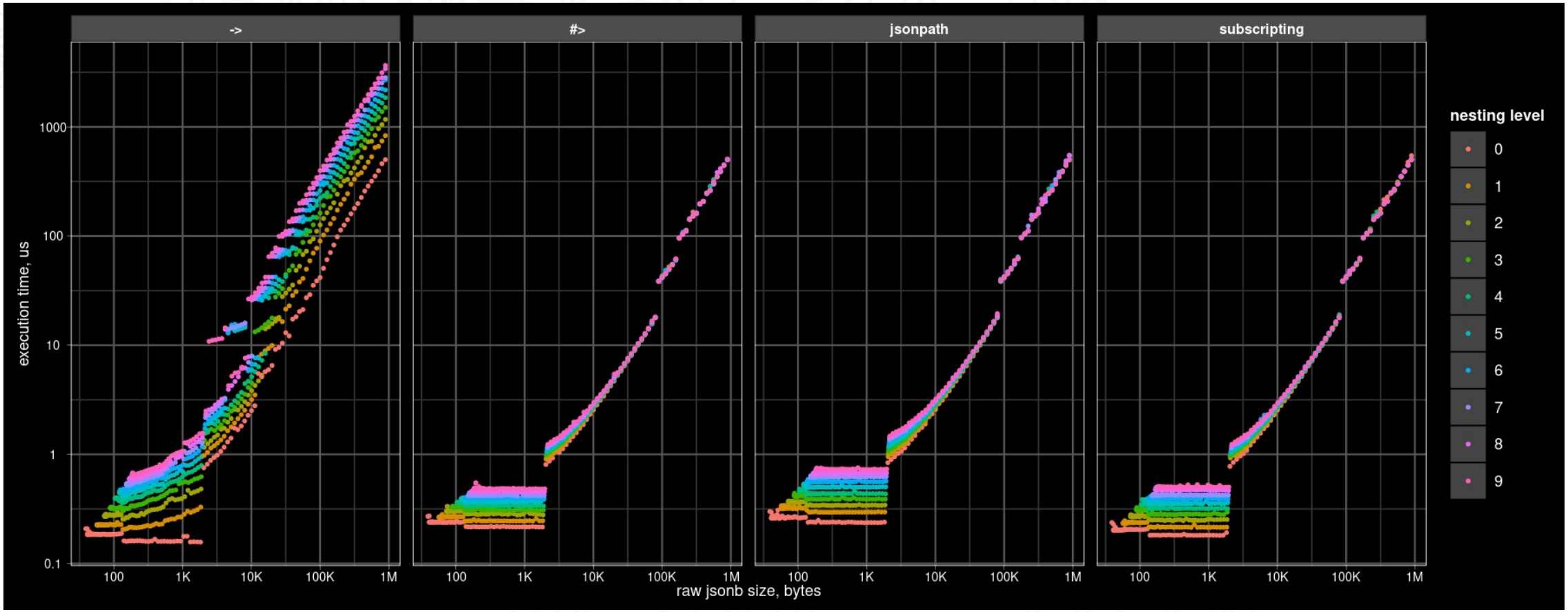
```
SELECT expr FROM test_jsonb_nesting WHERE size = ? AND level = ?;
```

- Test expressions:
  - `->:` `jb -> 'obj' -> 'obj' -> … -> 'obj' -> 'key'`
  - `#>:` `jb #> '{obj,obj,…,obj,key}'`
  - `subscripting:` `jb['obj']['obj']…['obj']['key']`
  - `jsonpath:` `jsonb_path_query_first(jb, '$.obj.obj.…,obj.key')`

PostgresPro

Nested containers performance

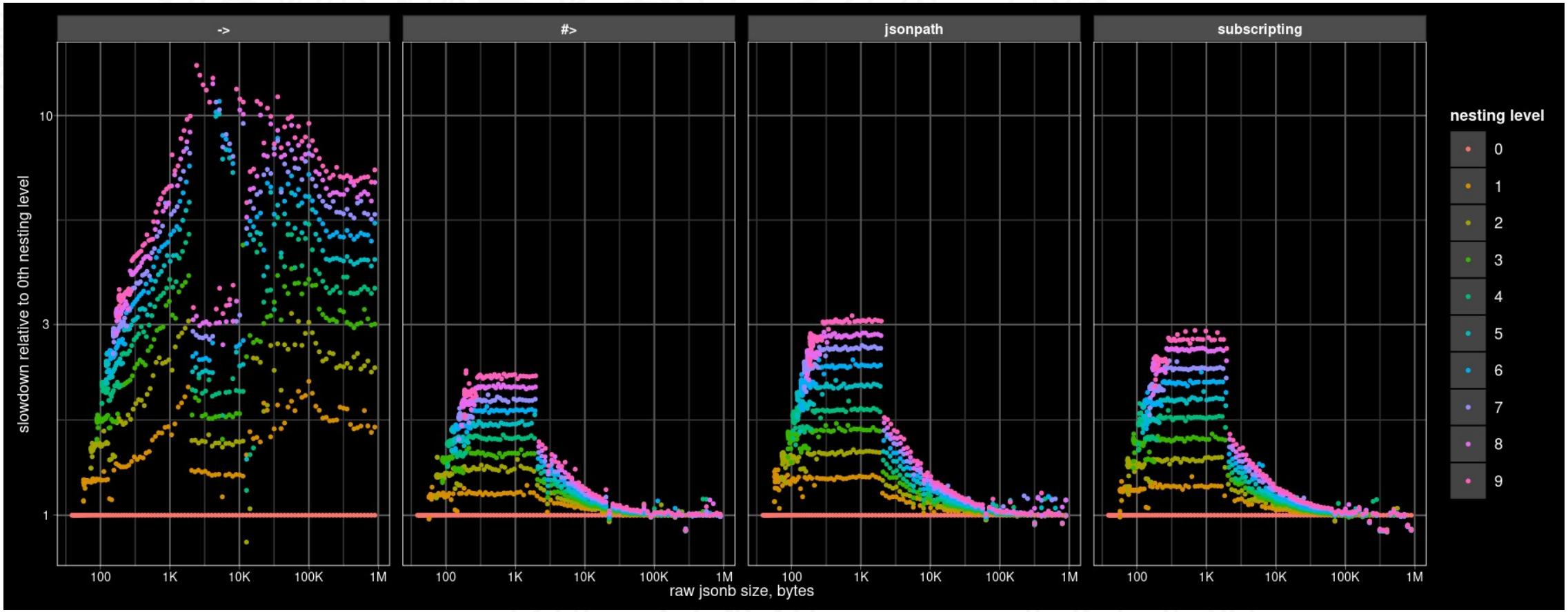
Expression execution time



PostgresPro

Nested containers performance

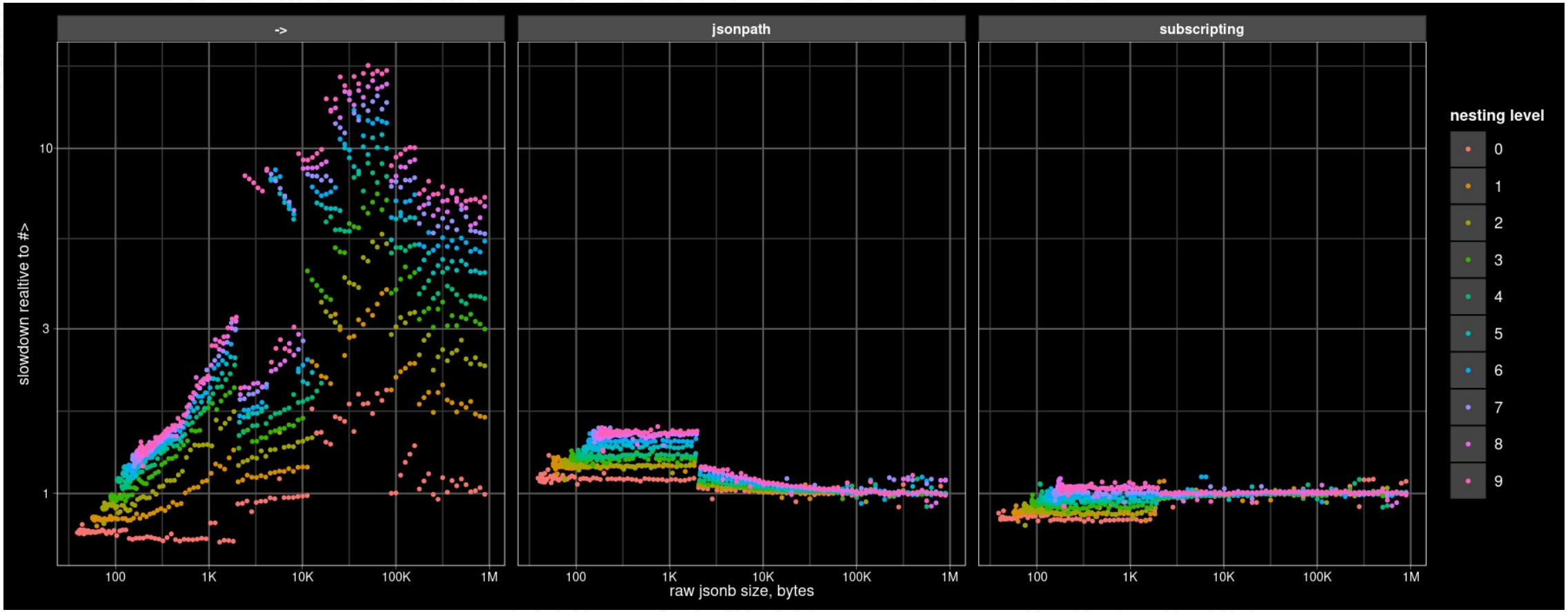
Expression slowdown relative to root level



PostgresPro

Nested containers performance

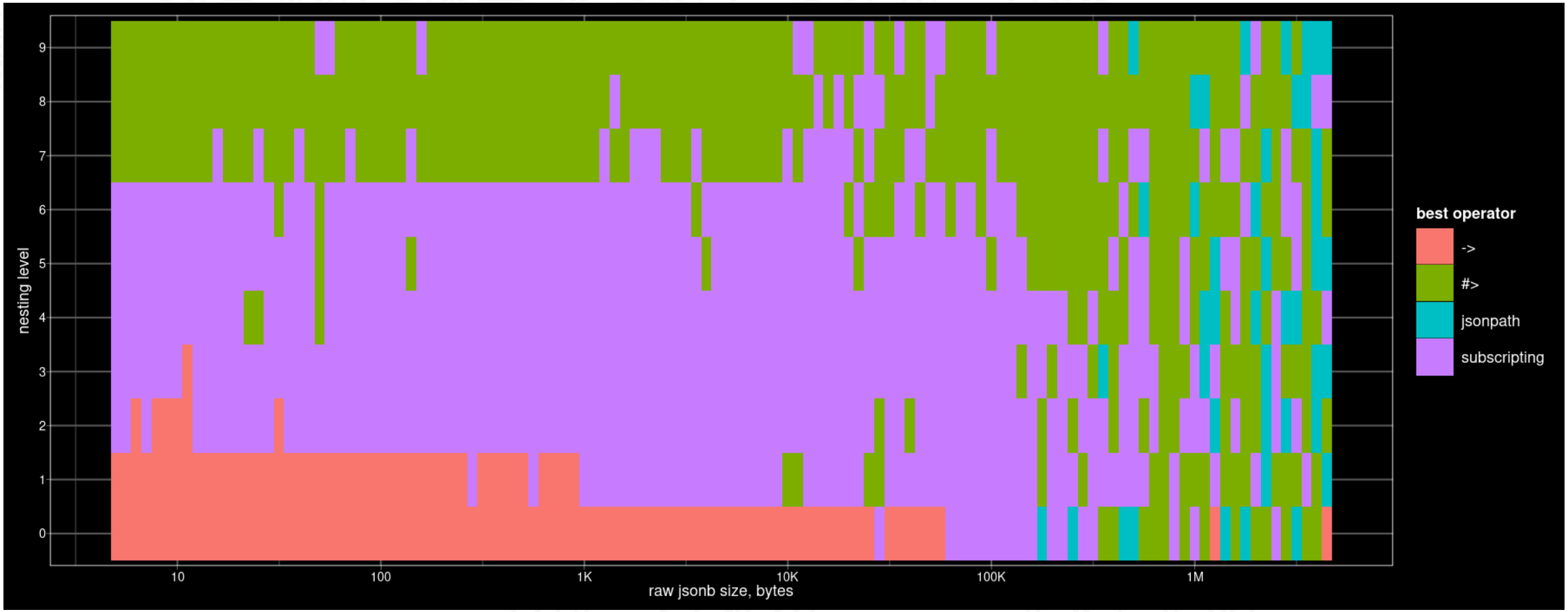
Expression slowdown relative to operator #>:



PostgresPro

Nested containers performance

Best operator depending on size and nesting level:



PostgresPro

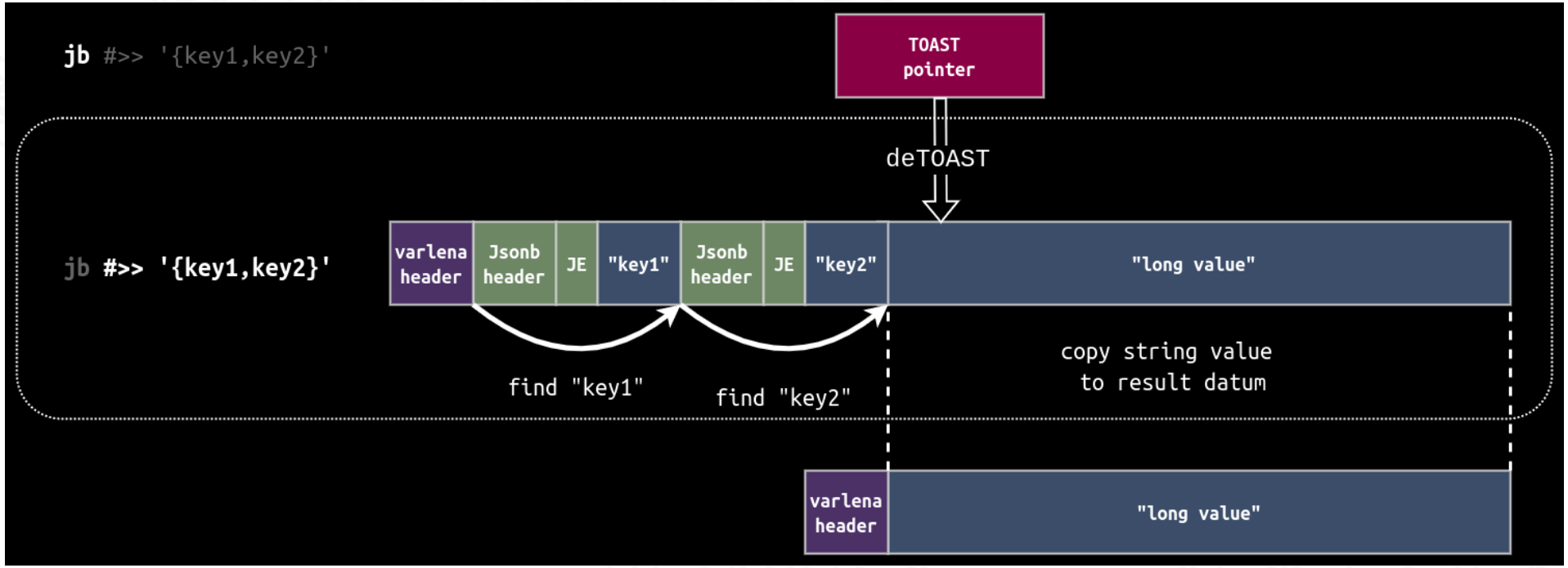
Nested containers performance

- All operators have a common overhead: deTOAST time + jsonb iteration time
- But they have a different additional overheads:
  - -> chain has minimal initialization overhead, but it needs to copy intermediate results to temporary datums, and quickly becomes the slowest with nesting and size growth.
  - #> deconstructs text[] path into text datums before iteration, so it is slower than -> on low nesting levels and small sizes.
  - Subscripting shares the same iteration code with #>, but it receives path as C-array of text datums, each of them are evaluated by PG executor separately. This works faster for lower levels of nesting (< 7), but it becomes a bit slower than #> with its array deconstruction on higher levels.
  - Jsonpath has the highest interpretation overheads, which become negligible only on big sizes where deTOASTing brings the major overhead.

PostgresPro

Nested containers performance

#> (subscripting, jsonpath) execution process



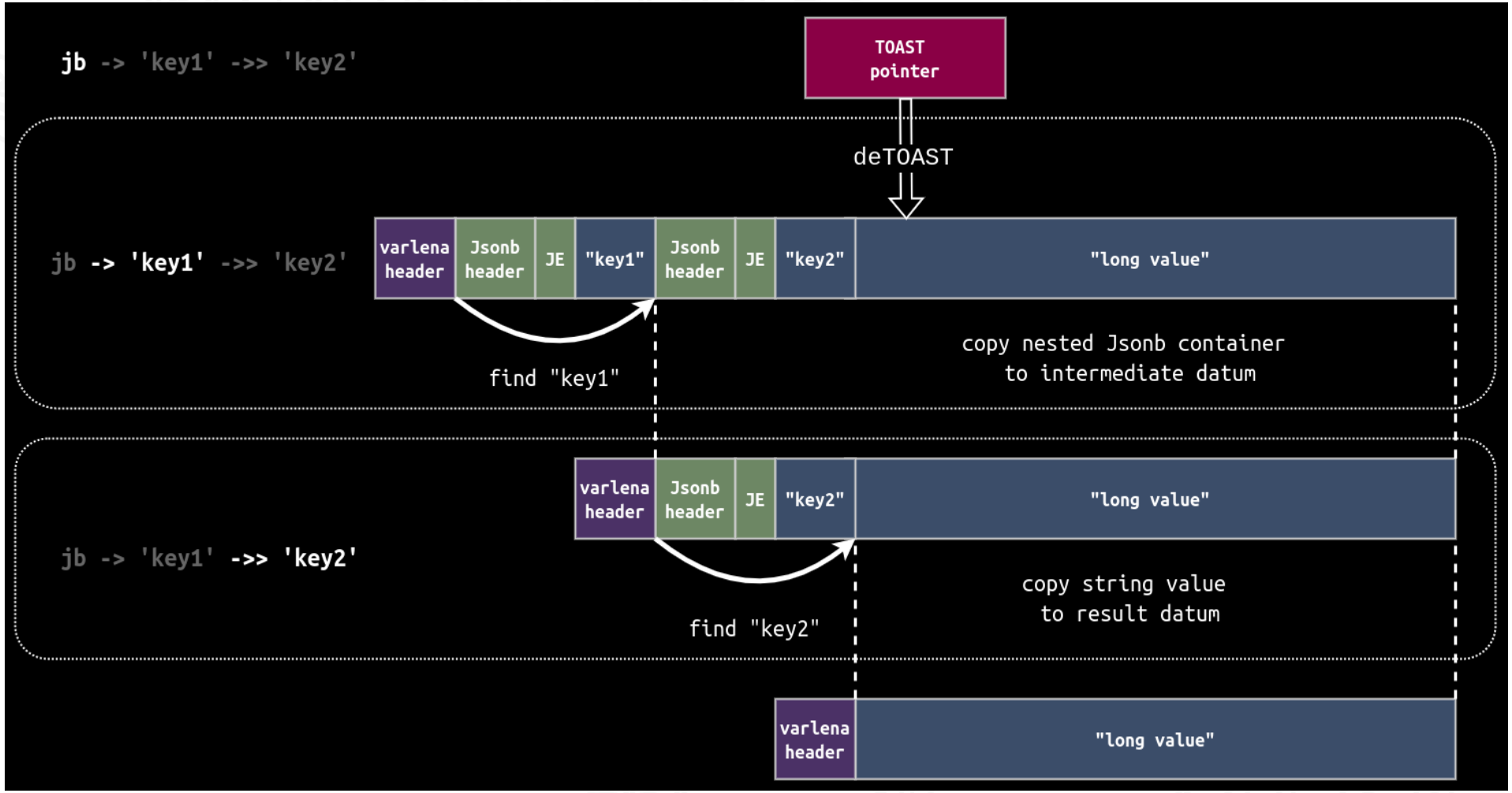
Time ~ Cinit + Cinterp * Nlevels + Size * Cdetoast

Cinit, Cinterp are different for #>, subscripting, jsonpath

PostgresPro

Nested containers performance

-> chain execution process



Time ~ Cinit + Cinterp * Nlevels + Size * (Cdetoast + Nlevels * Ccopy)

PostgresPro

Nested containers performance

- Safely use - > for root level of any size and for 1-st level for small jsonb (< 1kb)
- Use subscripting and #> for large jsonb and higher nesting level
- Jsonpath is the slowest, consider it for complex queries !

PostgresPro

Jsonpath vs jsonb contains vs SQL

- Sample table with arrays of various sizes (1 – 1M elements):

```
CREATE TABLE test_jsonb_array AS
SELECT id, size::int size,
  (SELECT jsonb_agg(i) FROM generate_series(0, size::int - 1) I) jb
FROM generate_series(0, 6 * 4) id, pow(10, id / 4) size;
```

- Test query:

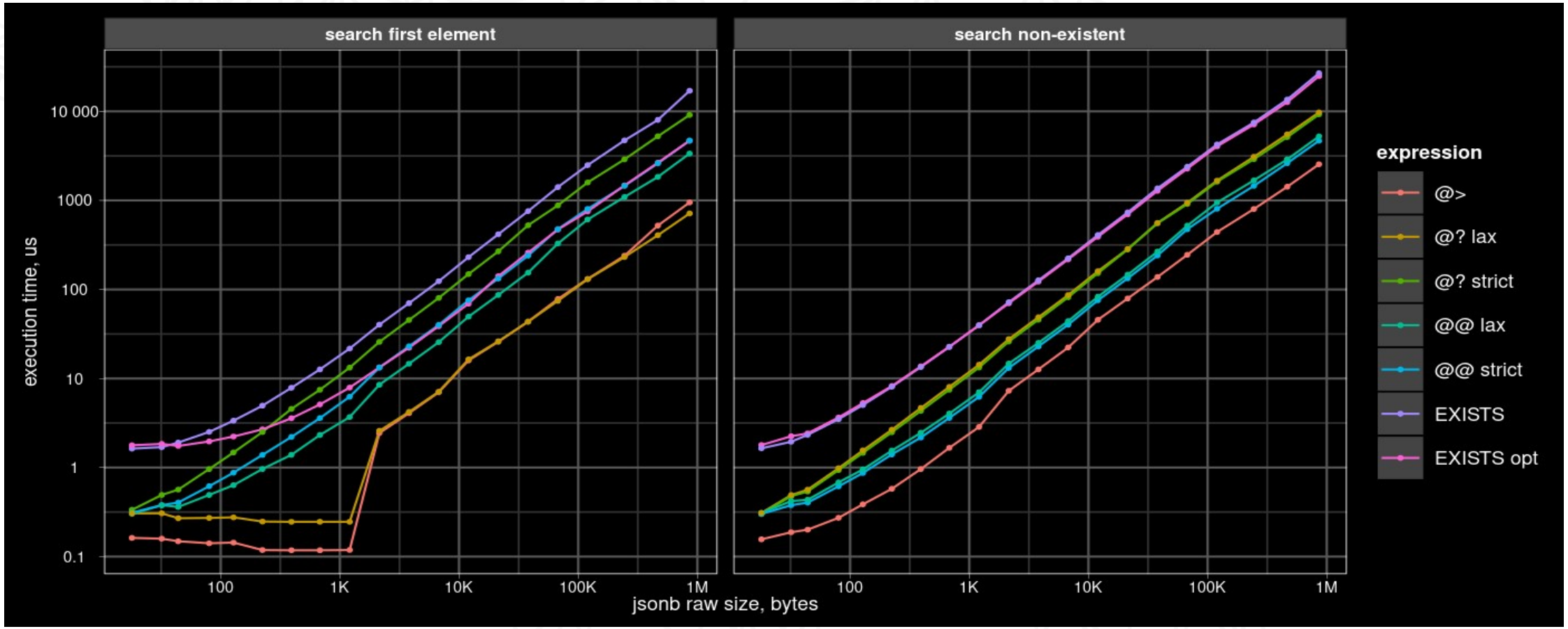
```
SELECT expr FROM test_jsonb_array WHERE id = ?;
```

- Test expressions:
  - `Contains: jb @> ?`
  - `Match:    jb @@ '$[*] == ?'`
  - `Exists:   jb @? '$[*] ? (@ == ?)'`
  - `EXISTS (SELECT FROM jsonb_array_elements(jb) e WHERE e = ?)`
  - `EXISTS (SELECT FROM jsonb_path_query_first(jb, '$') jb2,-- jb deTOASTed to jb2 only once`
    `generate_series(0, jsonb_array_length(jb2) – 1) i`
    `WHERE jb2 -> i = ?)`
- "Search first element": ? = 0
- "Search non-existent": ? = -1

PostgresPro

Jsonpath vs jsonb contains vs SQL

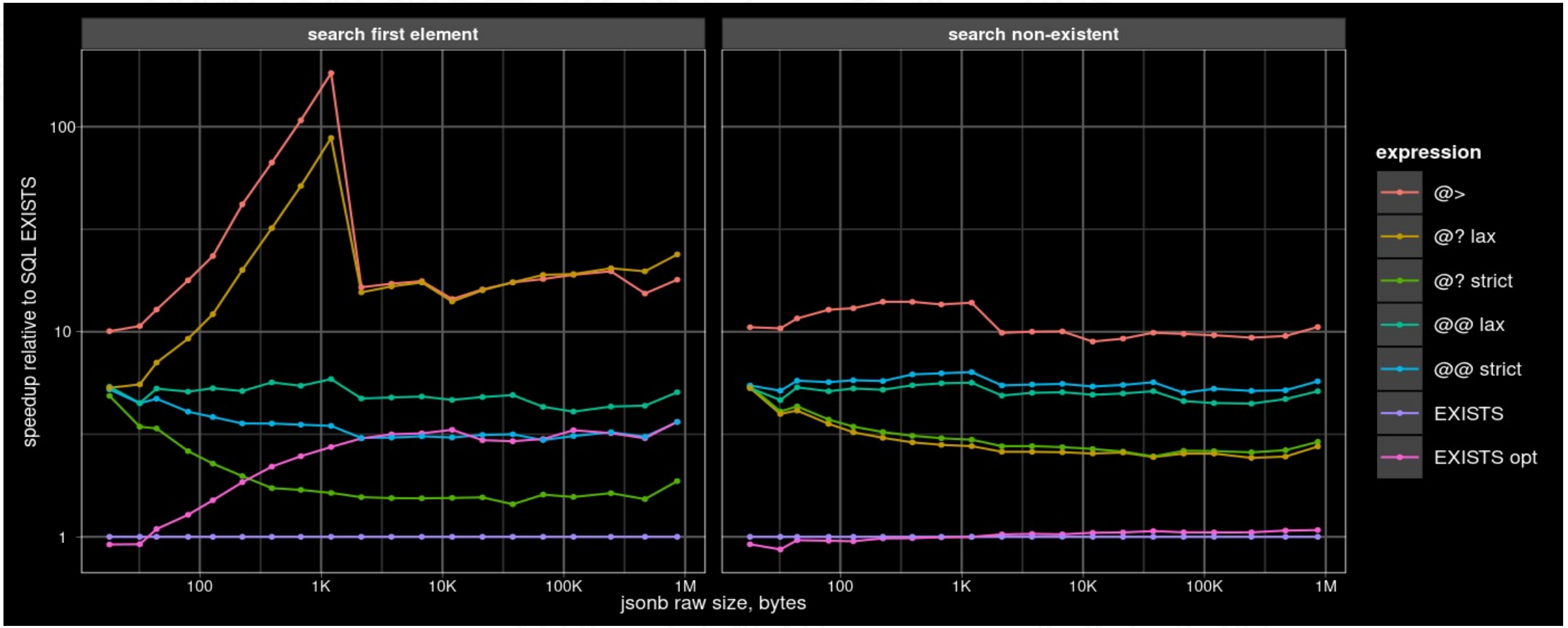
Execution time of expressions:



PostgresPro

Jsonpath vs jsonb contains vs SQL

Speedup relative to SQL EXISTS expression:



PostgresPro

Jsonpath vs jsonb contains vs SQL

- @> is the fastest in all cases
- SQL EXISTS is the slowest
- Relative performance of @@ and @? depends on the fraction of elements needed to be iterated. Predicate “==” always extracts all items of its operands to the temporary lists before comparison. (This is necessary for handling for possible incomparable items required by standard). So:
  - @@ is faster when all elements need to be iterated, because comparison of lists has less overhead than per-item comparison in filter
  - @? is faster when the desired element can be found quickly and no need to collect all items

The Curse of TOAST. Unpredictable performance

Small update cause 10 times slowdown !

```
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;
```

```
=# 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

PostgresPro

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)
```

PostgresPro

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)
```

PostgresPro

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)
```

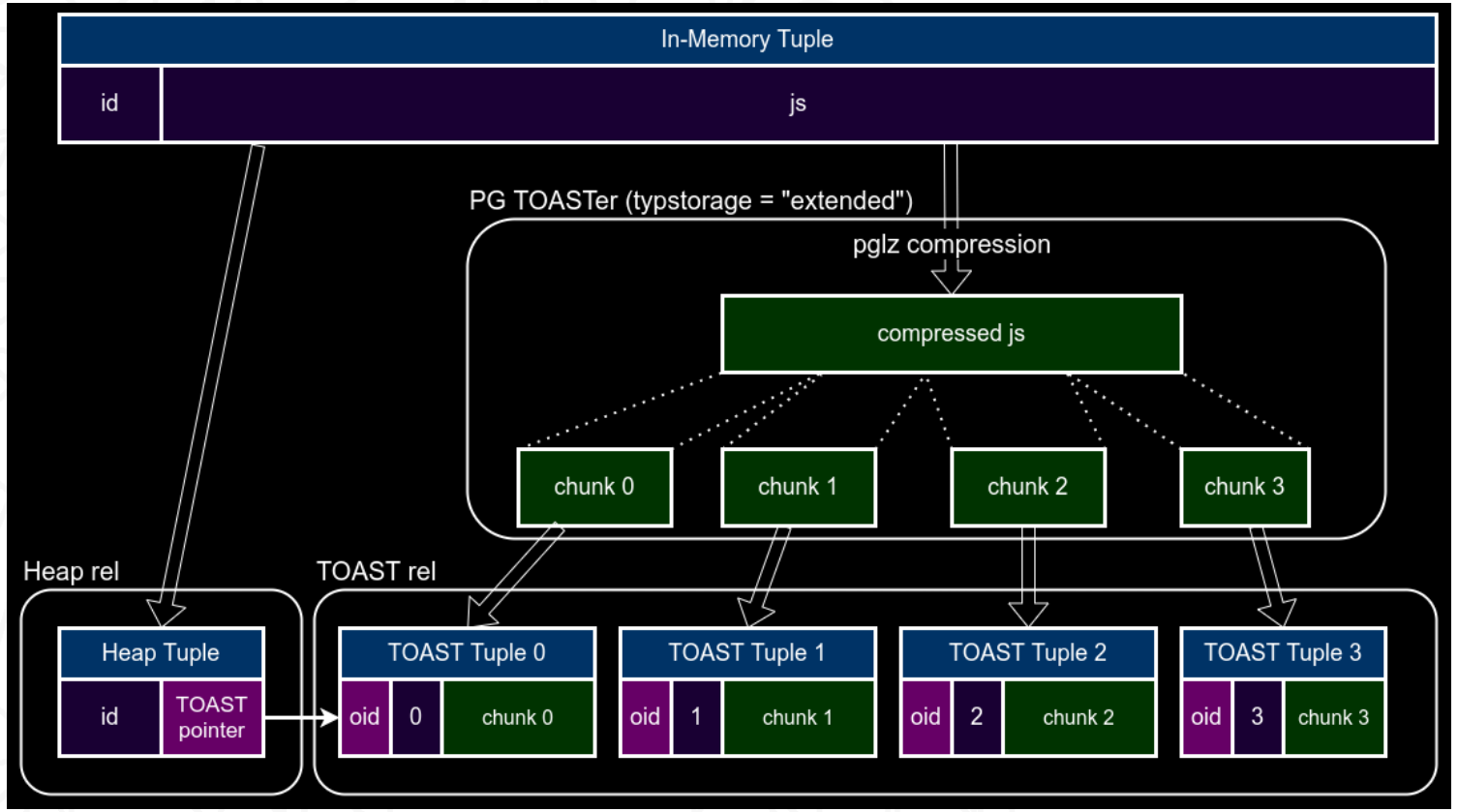
Table	TOAST
64 buffers	+ 3 buffers*10000

PostgresPro

TOAST Explained

The Oversized-Attribute Storage Technique

- TOASTed (large field) values are compressed, then splitted into the fixed-size TOAST chunks (1996B for 8KB page)
- TOAST chunks (along with generated Oid `chunk_id` and sequnce 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`

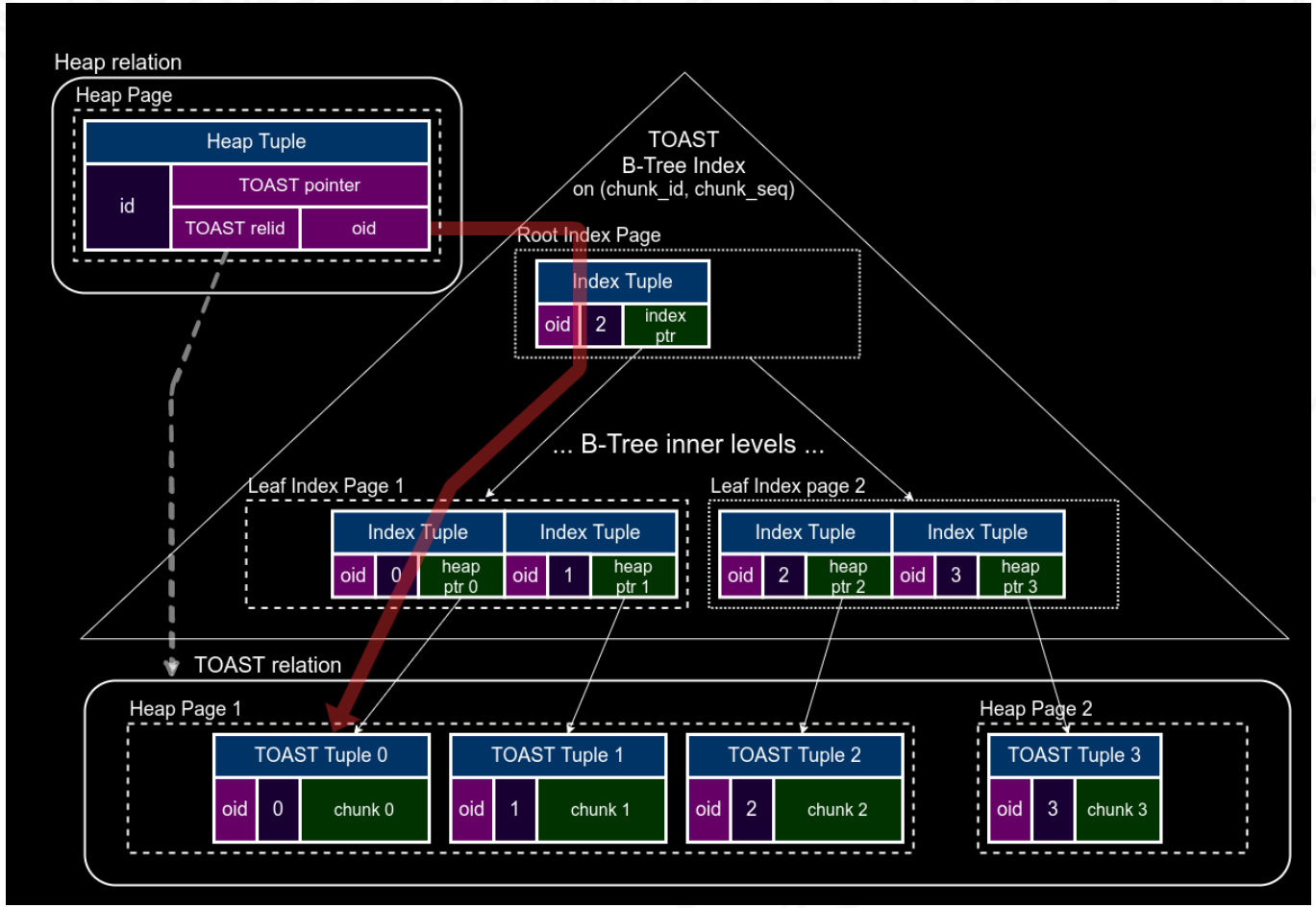


https://www.postgresql.org/docs/current/storage-toast.html

PostgresPro

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.

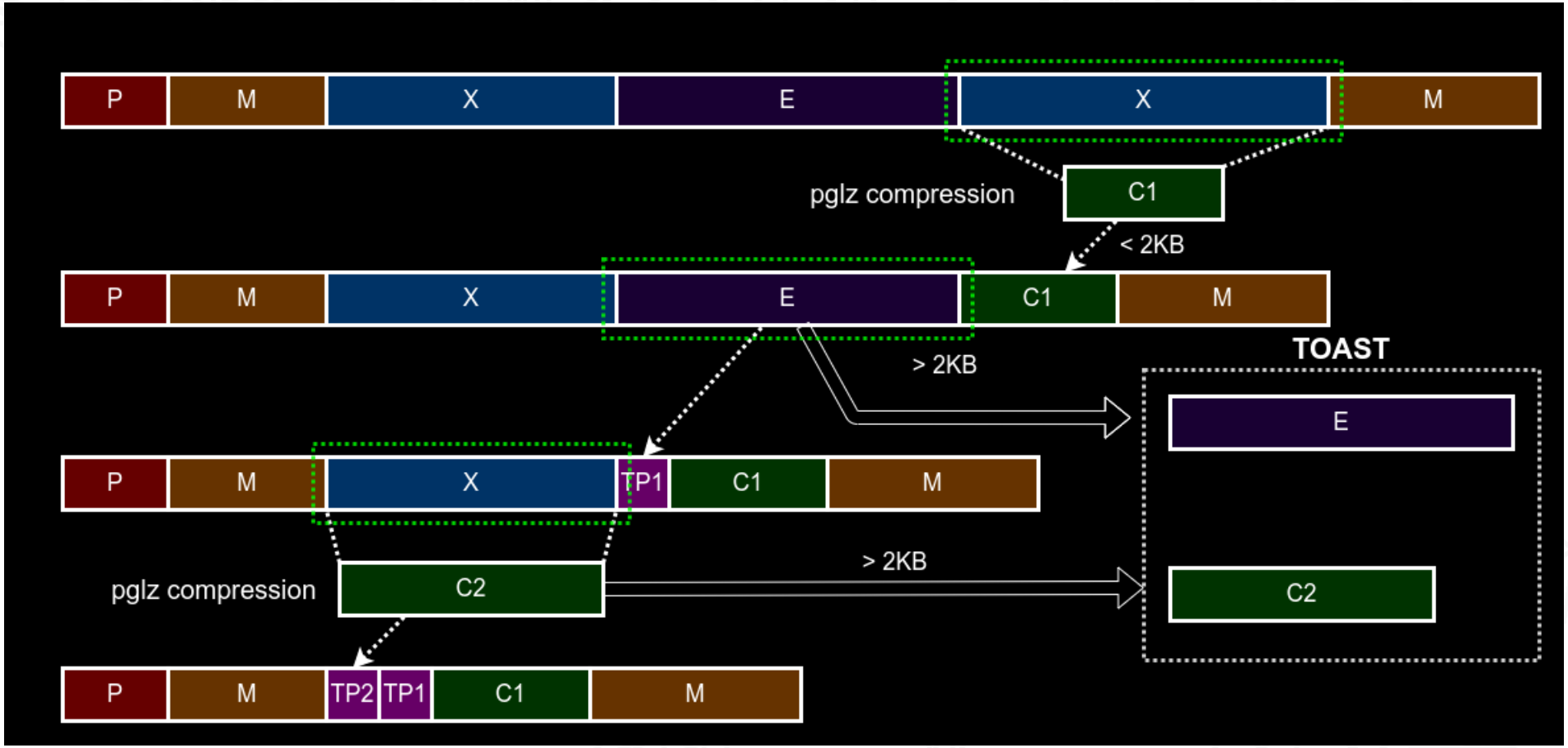
PostgresPro

TOAST passes

- Tuple is TOASTed if its size is more than 2KB (1/4 of page size).
- There are 4 TOAST passes.
- At the 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

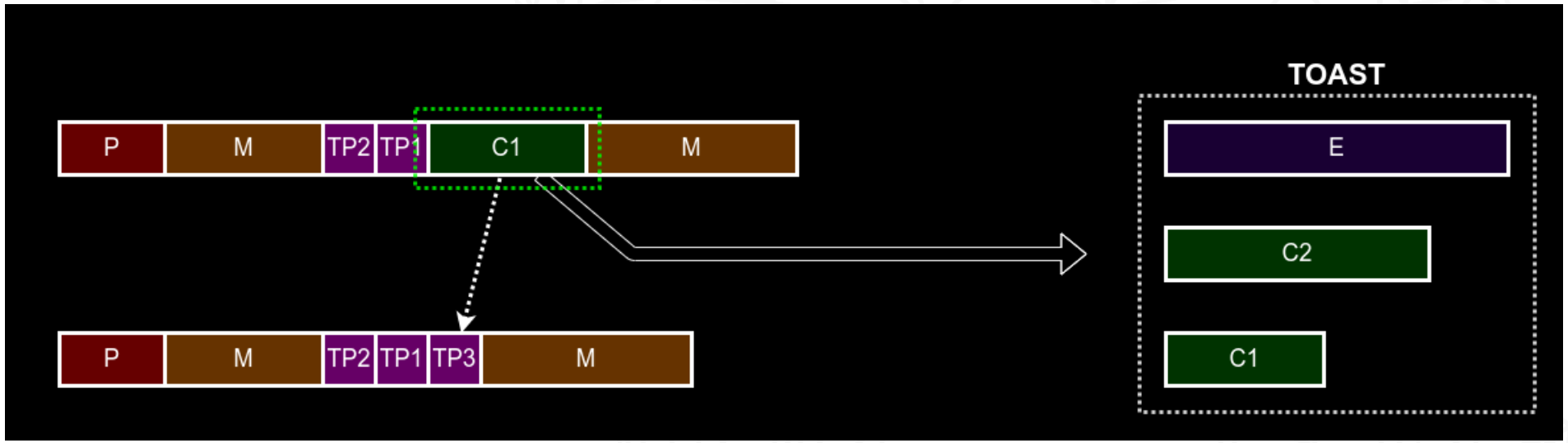
- Only "extended" and "external" attributes are considered, "extended" attributes are compressed. If their size is more than 2KB, they are TOASTed.



PostgresPro

TOAST pass #2

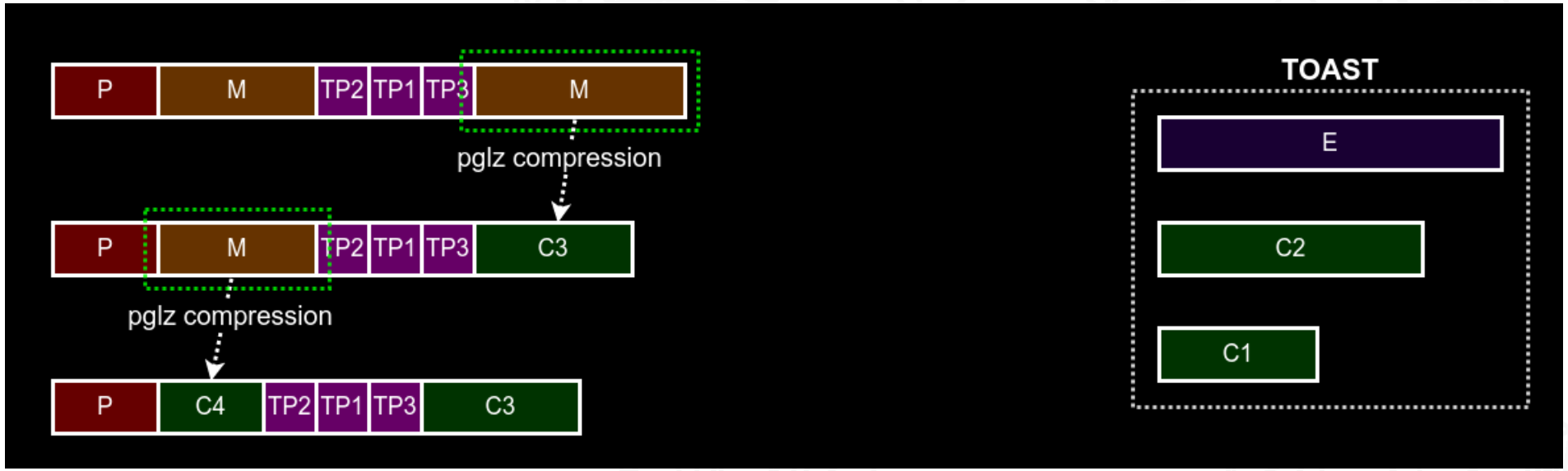
- Only "extended" and "external" attributes (that were not TOASTed in the previous pass) are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.



PostgresPro

TOAST pass #3

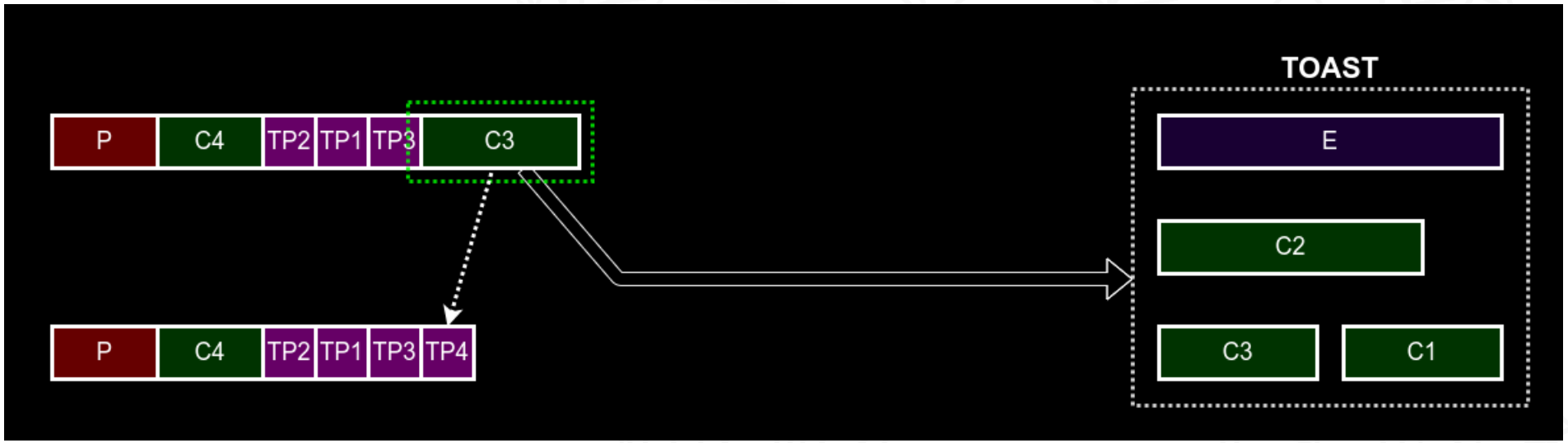
- Only "main" attributes are considered.
- Each attribute is compressed, until the resulting tuple size < 2KB.



PostgresPro

TOAST pass #4

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



PostgresPro

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;
```

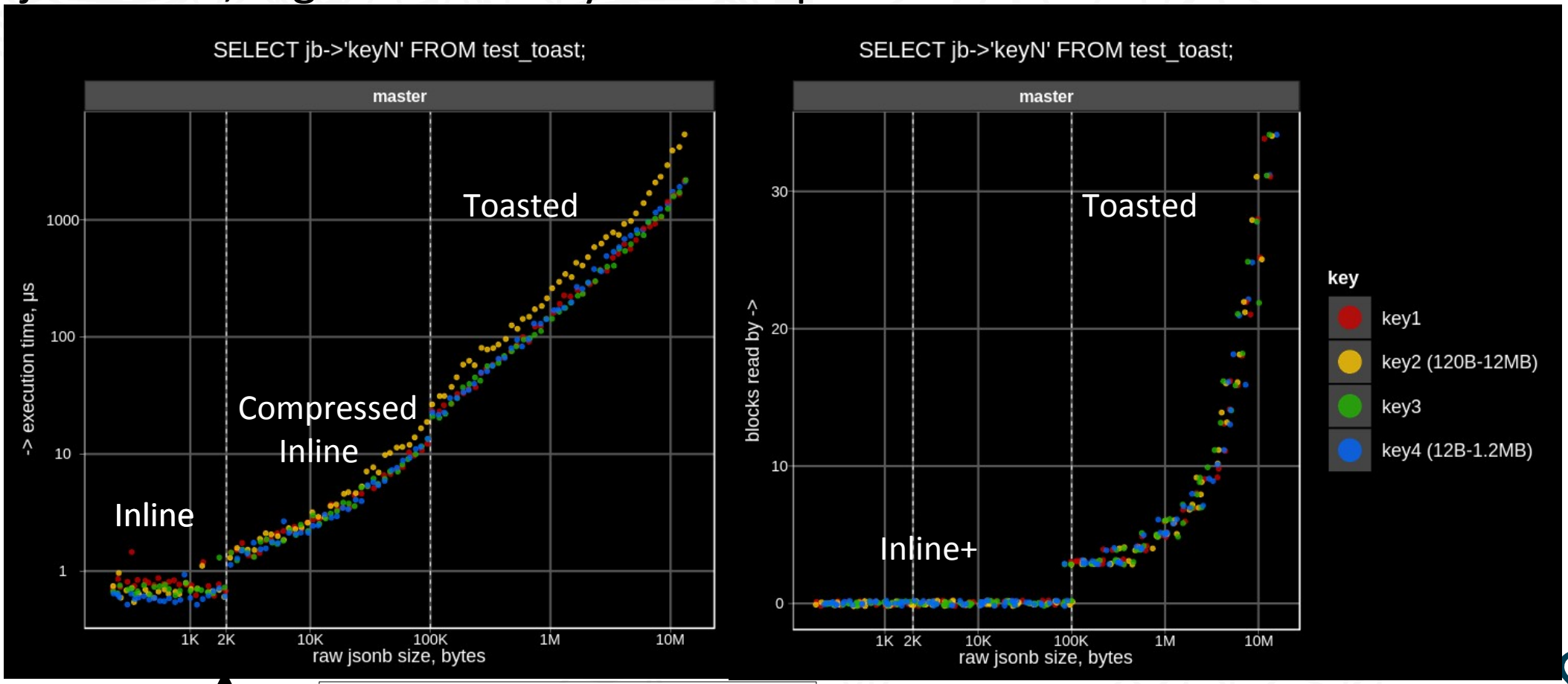
- Each jsonb looks like: `key1`, loooong `key2[]`, `key3`, long `key4[]`.
- We measure execution time of operator `->(jsonb, text)` for each row by repeating it 1000 times in the query:

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

PostgresPro

Motivational example (synthetic test)

Key access time for TOASTed (raw size > 100 Kb) jsonbs linearly increase with jsonb size, regardless of key size and position.

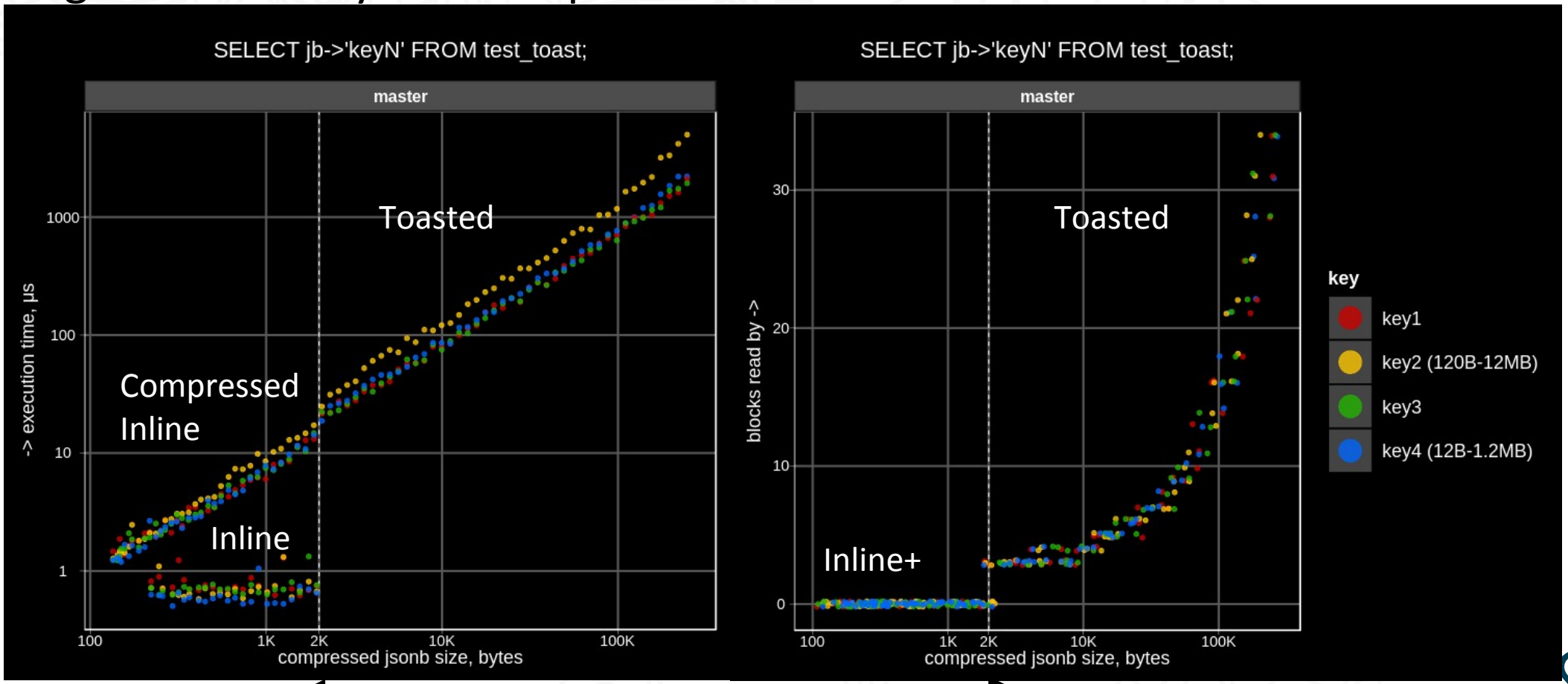


Large jsonb is TOASTed !

PostgresPro

TOAST performance problems (synthetic test)

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



Large jsonb is TOASTed !

PostgresPro

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 differencies 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

PostgresPro

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
```

PostgresPro

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)
```

PostgresPro

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       | pg_toast.pg_toast_47235 | 156 MB
            (was 1024 kB)                             (was 78 MB, 2x more)
```

PostgresPro

Jsonb deTOAST improvements goal

Ideal goal: no dependency on jsonb size and position

- Access time ~ O(level)
- Update time ~ O(level) + O(key size)

- Original TOAST doesn’t use inline, only TOAST pointers are stored
- Utilize inline (fast access) as much as possible:
  - Keep inline as much as possible uncompressed short fields and compressed medium-size fields
- Keep compressed long fields in TOAST chunks separately for independent access and update.

PostgresPro

Jsonb deTOAST improvements (root level)

- Partial (prefix) decompression - eliminates overhead of pglz decompression of the whole jsonb – FULL deTOAST and partial decompression:

  `Decompress(offset) + Detoast(jsonb compressed size),`

  offset depends on key position

- Sort jsonb object key by their length – good for short keys

  `Decompress(key_rank * key size) + Detoast(jsonb compressed size),`

  offset depends on key size

- Partial deTOAST and partial decompression (deTOASTing iterator)

  `Decompress(key_rank * key size) + Detoast(key_rank`$_C$ `* key size)`

- Inline TOAST – store inline prefix of compressed data (jsonb header and probably some short keys)

  `Decompress(key_rank * key size)` -- great benefit for inline short keys !

  `Decompress(key_rank * key size) + Detoast(key_rank`$_C$ `* key size)`

PostgresPro

Jsonb deTOAST improvements

- Compress_fields – compress fields sorted by size until jsonb fits inline, fallback to Inline TOAST.

  `O(1)` – short keys

  `Decompress(key size)` – mid size keys

- Shared TOAST – compress fields sorted by size until jsonb fits inline, fallback to store compressed fields separately in chunks, fallback to Inline TOAST if inline overfilled by toast pointers (too many fields).
  - Access

    `O(1)` – short keys

    `Decompress(key size)` – mid size keys

    `Decompress(key size) + Detoast(key size)` – long keys
  - Update

    `O(inline size)` – short keys (inline size < 2KB)

    `O(inline size) + O(key size)` – keys in chunks

    `O(jsonb size)` – inline TOAST

PostgresPro

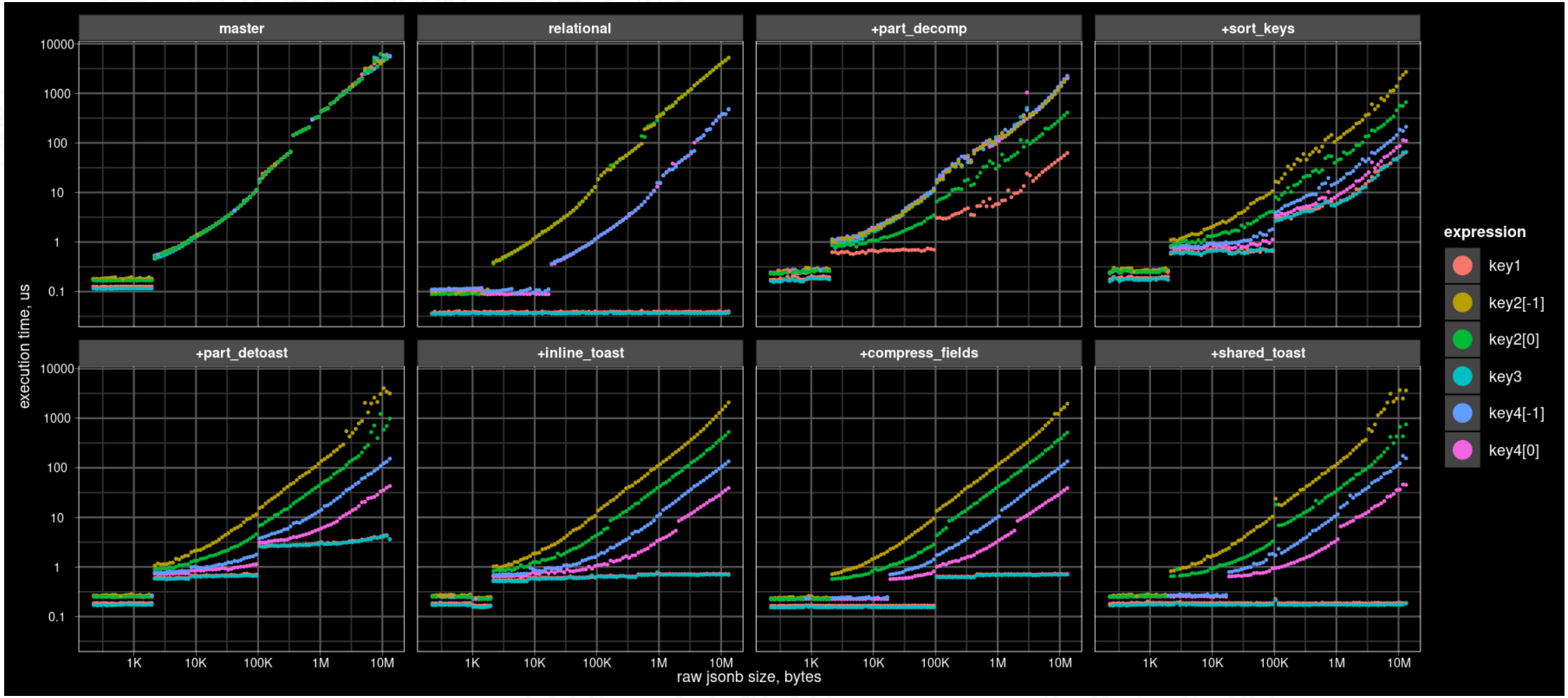
Jsonb deTOAST improvements

- In-place updates for TOASTed jsonb:
  - Store new element values, their offsets and lengths together with TOAST pointer (some kind of diff) instead of rewriting TOAST chunk chains, if element's size and type is not changed (in-place update) and new value fits into inline.
  - Old values are replaced with new ones during deTOASTing.

  - Update:
    - `O(element size)` – if in-place update and new value fits into inline
    - `O(array size)` – otherwise

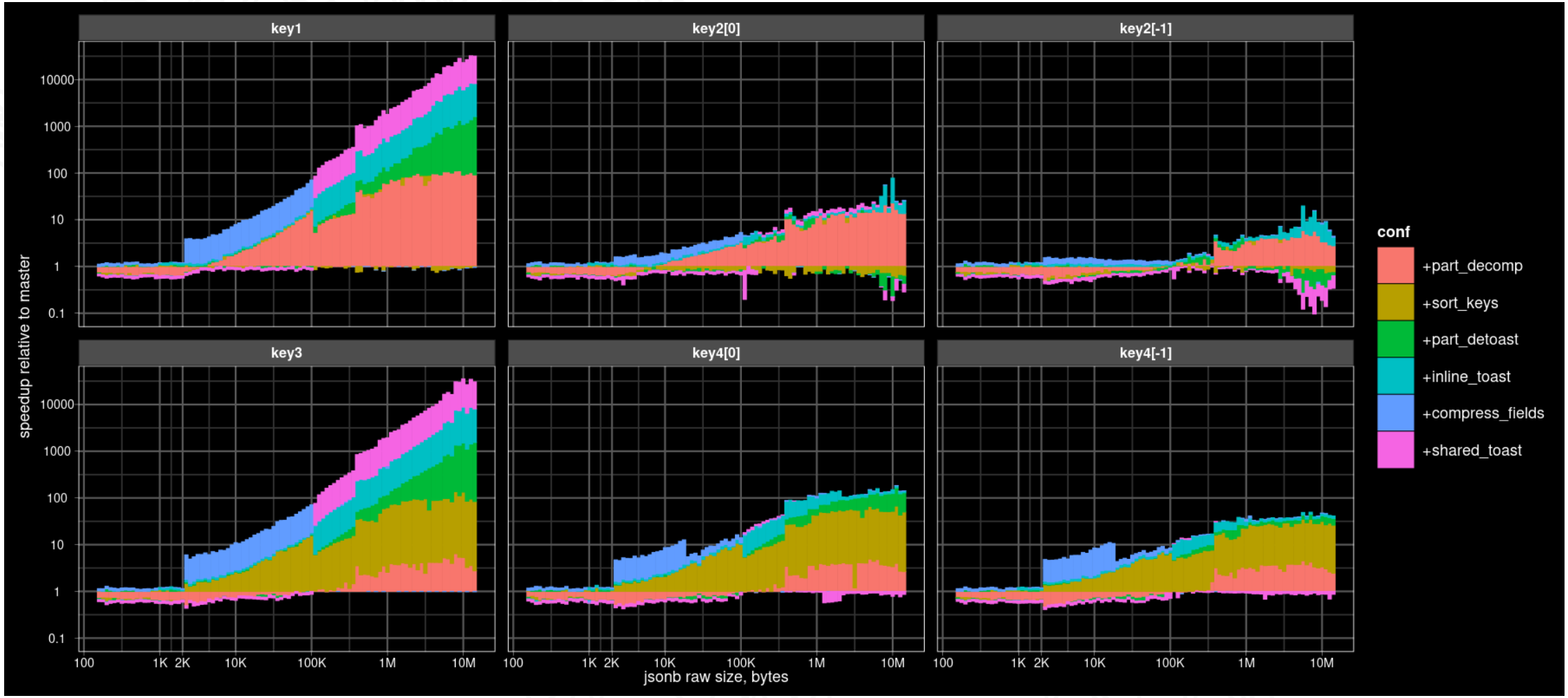
PostgresPro

TOAST optimizations: execution time graphs



PostgresPro

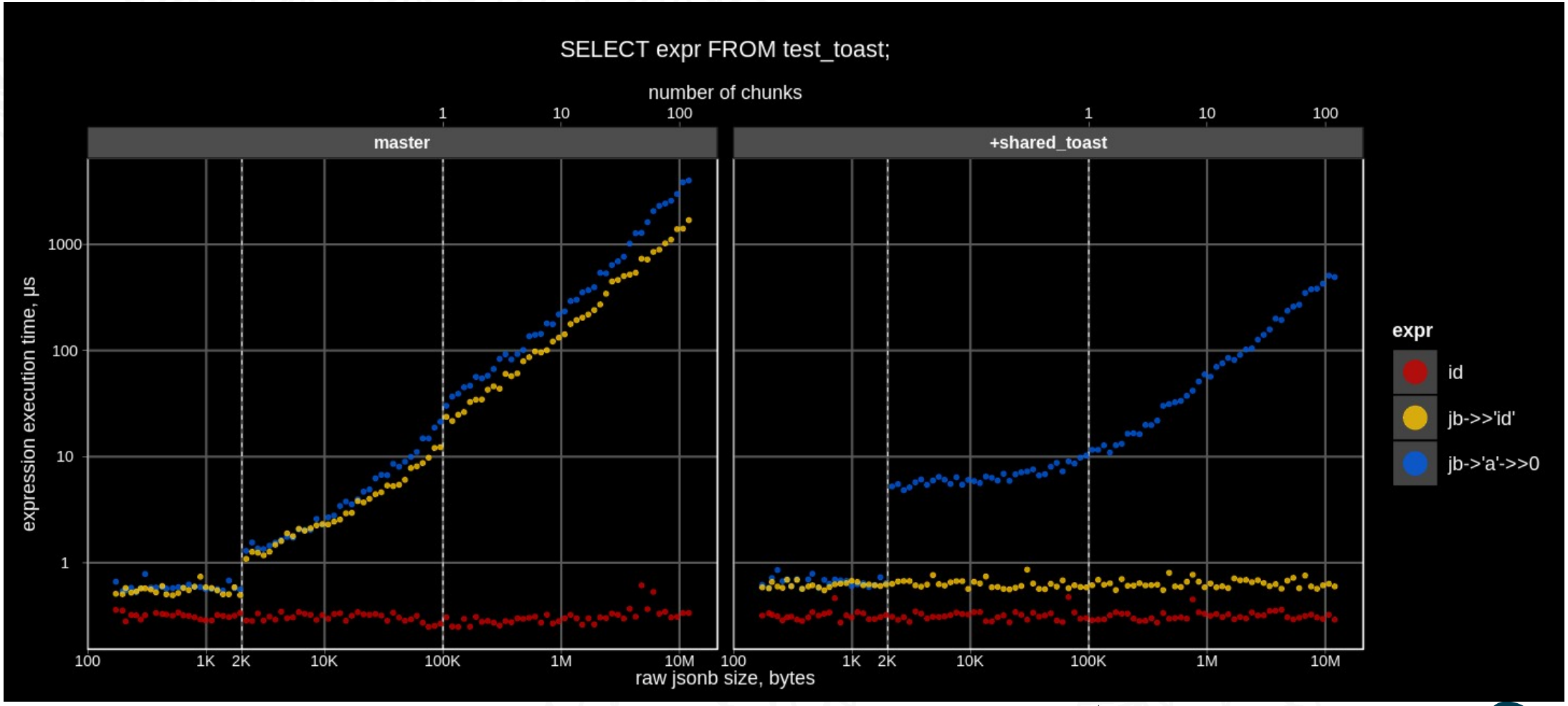
TOAST optimizations: stacked improvements



PostgresPro

Popular mistake: CREATE TABLE qq (jsonb)

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



Large jsonb is TOASTed !

PostgresPro

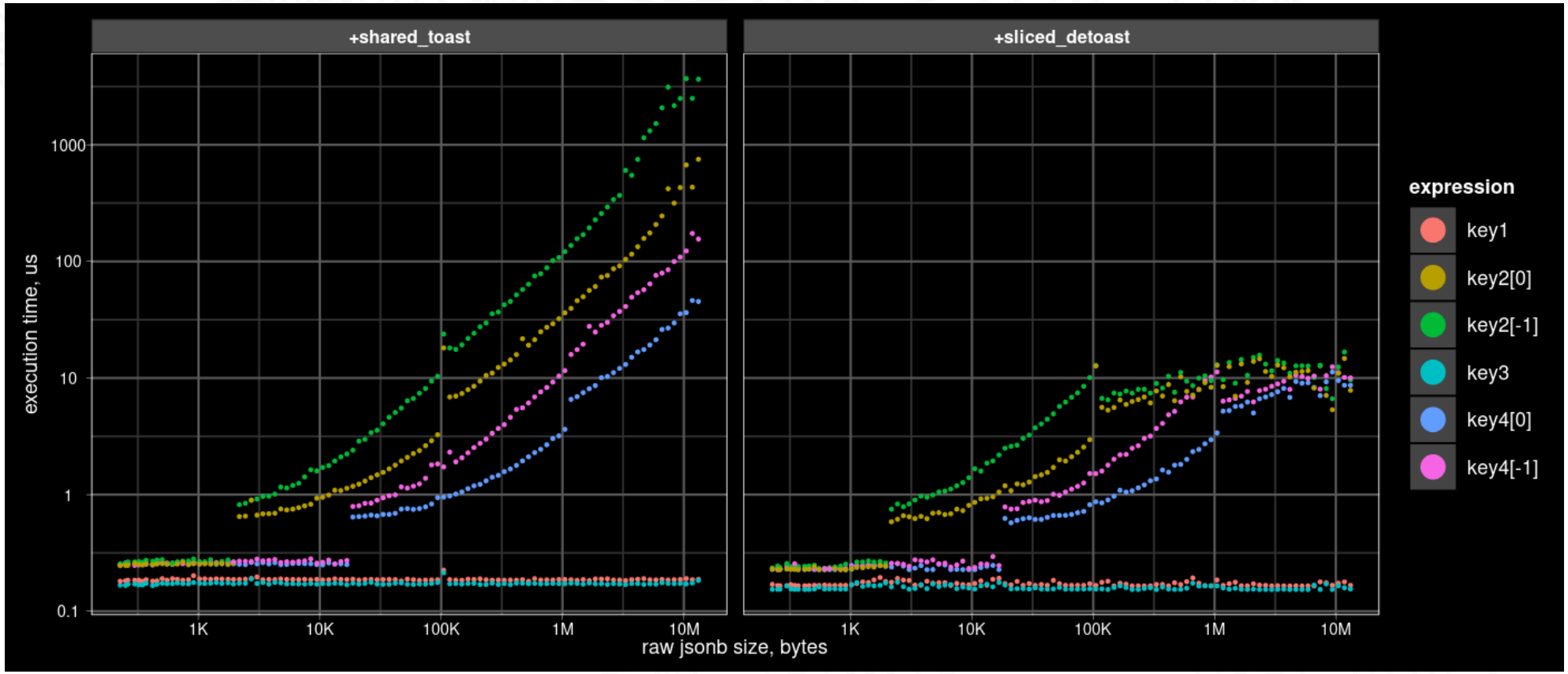
Jsonb deTOAST improvements (experimental)

- Sliced detoast — improve access to array element, stored in chunks. Decompress only needed slices of selected chunks (currently chunk compression is not supported).

PostgresPro

Further TOAST optimizations: random access

- Access time to array elements doesn't grow too much with array size.

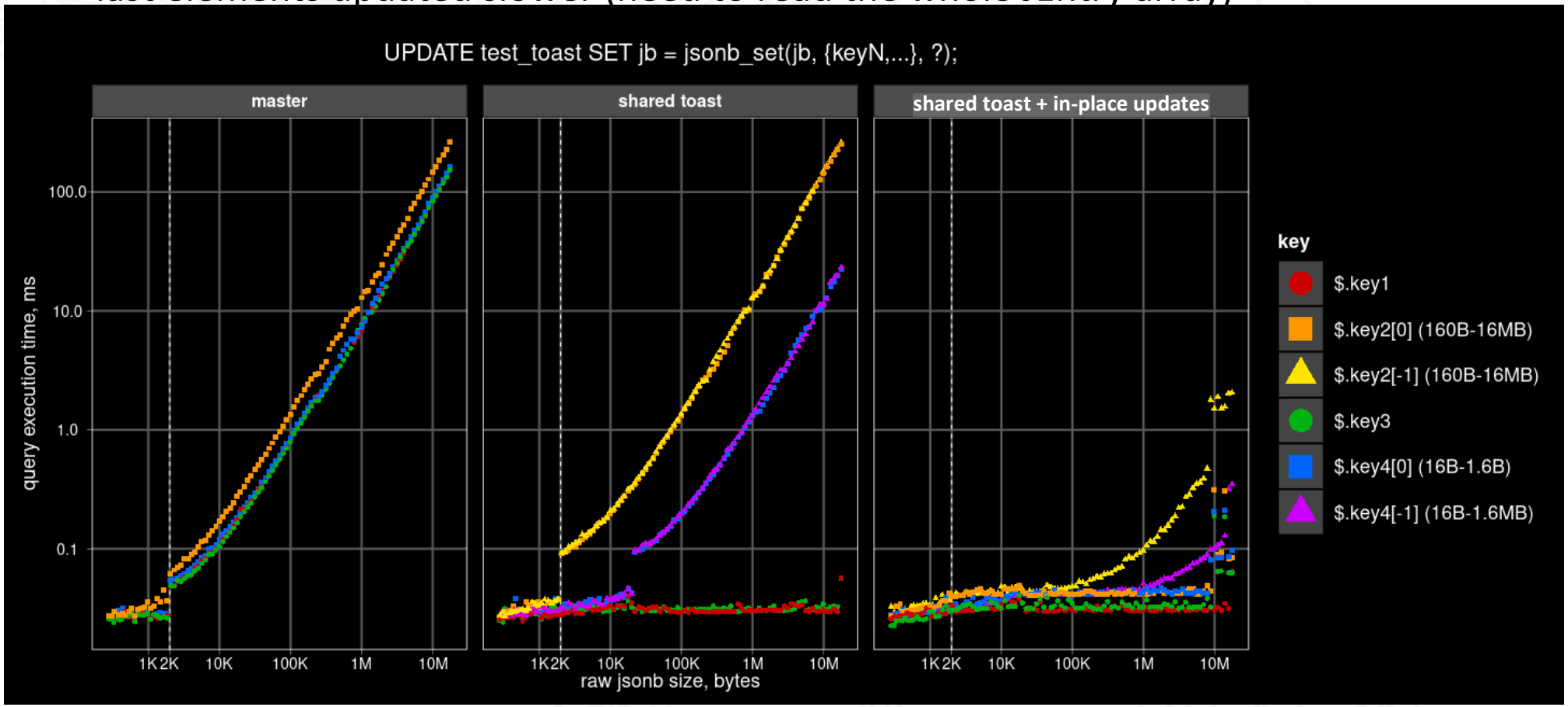


PostgresPro

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)

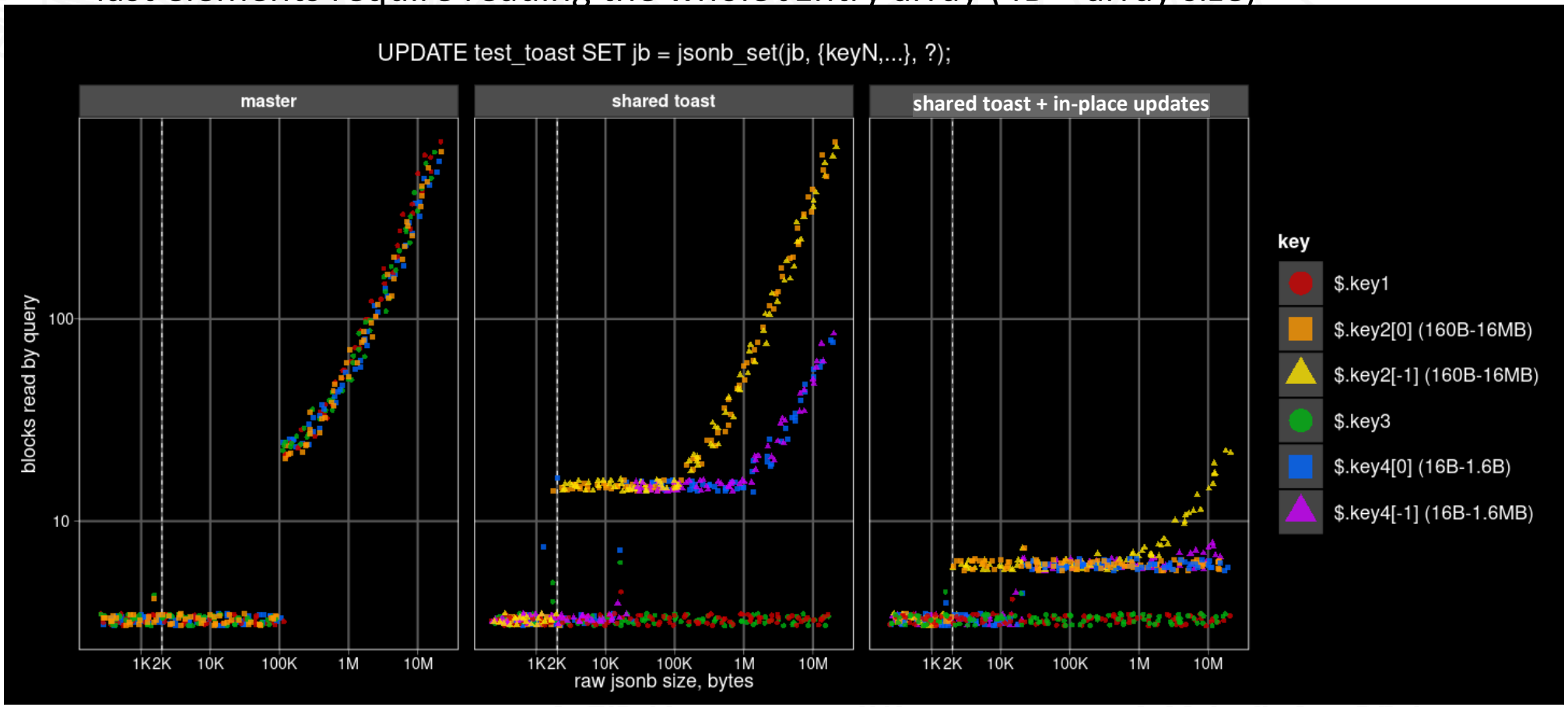


PostgresPro

Shared TOAST – in-place update results (synthetic)

Number of blocks read depends on element position:

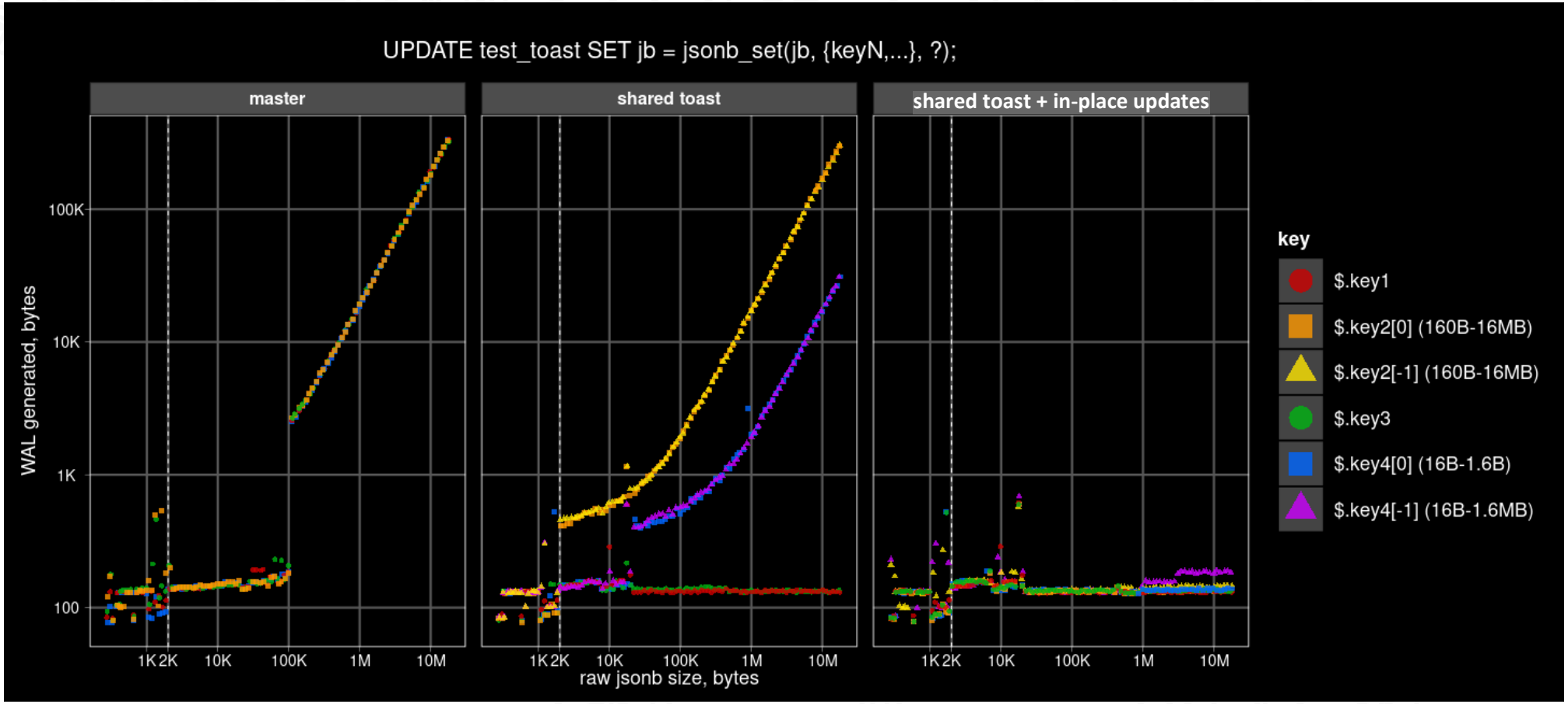
- first elements do not require reading of additional blocks
- last elements require reading the whole JEntry array (4B * array size)



PostgresPro

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



PostgresPro

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);
```

PostgresPro

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);
```

PostgresPro

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:

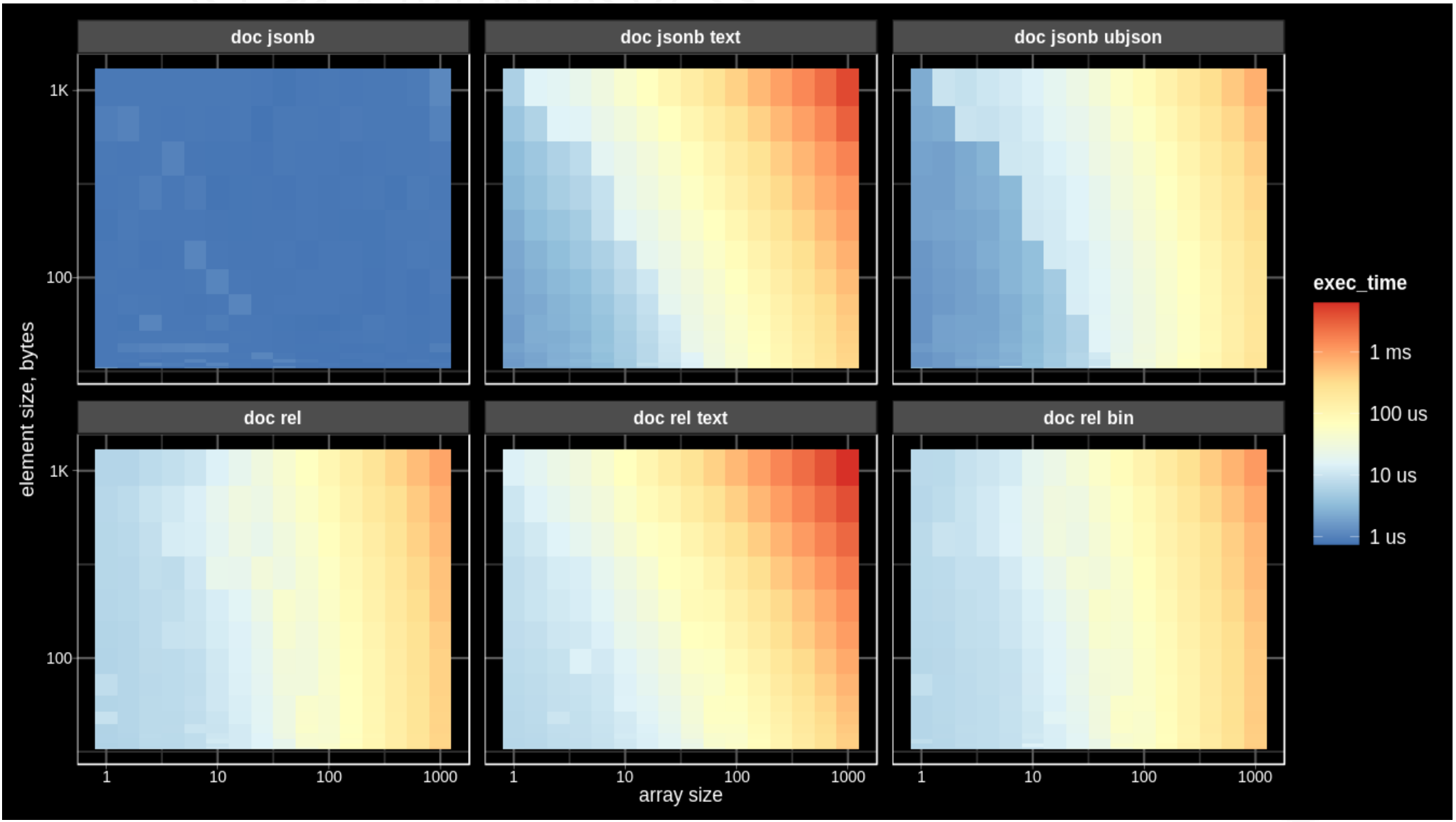
```
SELECT (SELECT array_agg(e.val)
        FROM test_jsonb_arrays_rel_elems e
        WHERE e.id = a.id)
FROM test_jsonb_arrays_rel a
WHERE array_size = $1 AND elem_size = $2;

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

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

PostgresPro

JSONB vs Relational: access whole document



PostgresPro

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
  ...
```

```
 2000 |   20 | {"key1": ["obN0zR2zmijOylIHiORHuNjs7LuCbhMC1YNJROFNxo5FoVnjnGdCDmW5VX425n3vp
N5KX8RIoxlcCXqUHtBU8ShKJBtL8x5DHR3U"], "key2": ["FKUItaFBhwvrboWfmysVCJja8i7QHkgH"], "key3"
: ["NSEjgn9Ueh"], "key4": ["uNw"], "key5": ["H"], "key6": [""], "key7": [""], "key8": [""],
 "key9": [""], "key10": [""]}
```

```
CREATE TABLE test_jsonb_object (id int, size 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)))
           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);
```

PostgresPro

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);
```

PostgresPro

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:

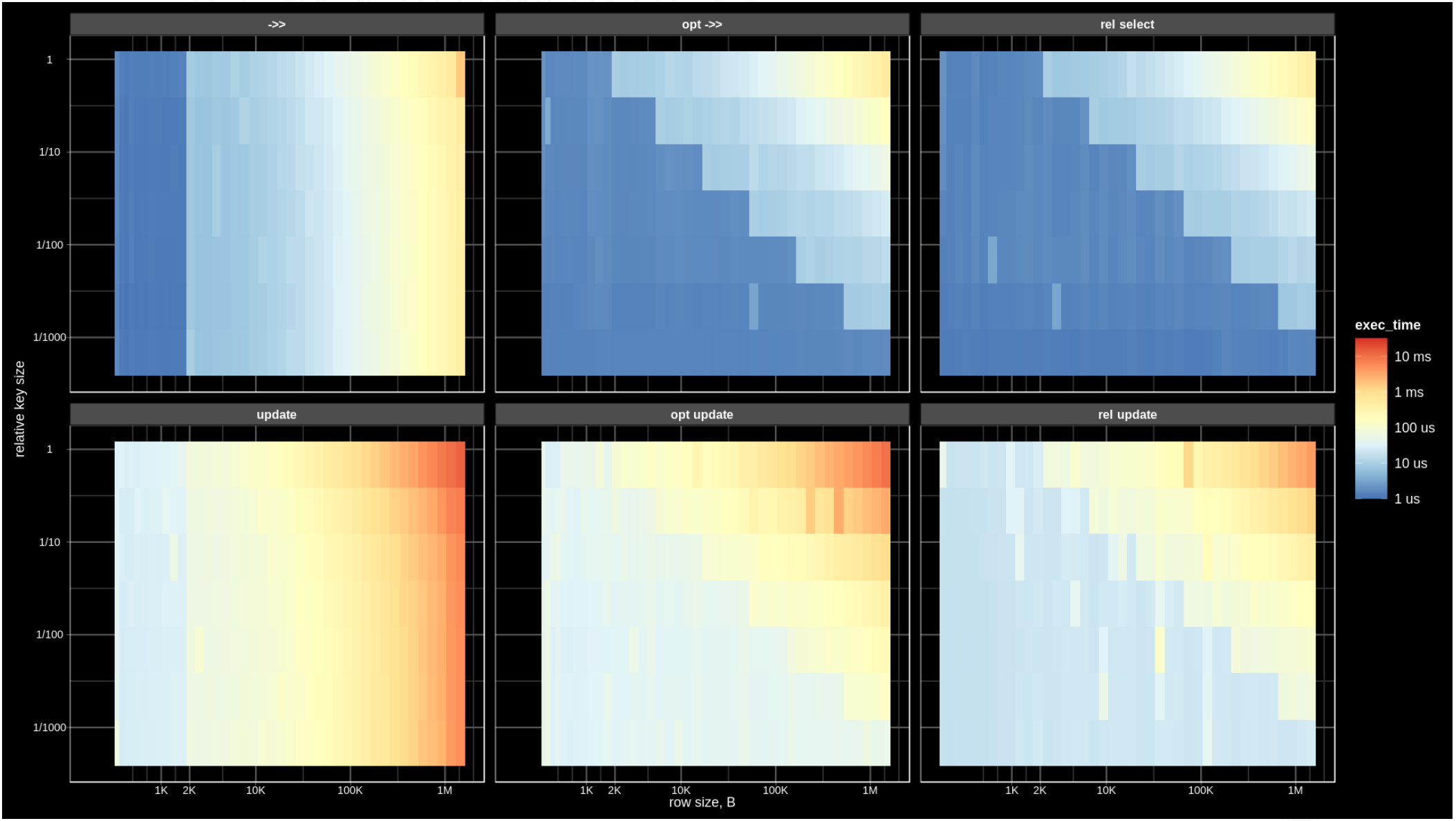
```
UPDATE test_jsonb_object
SET jb = jsonb_set(jb, '{key$1,0}', to_jsonb($2))
WHERE id = $3;

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
```

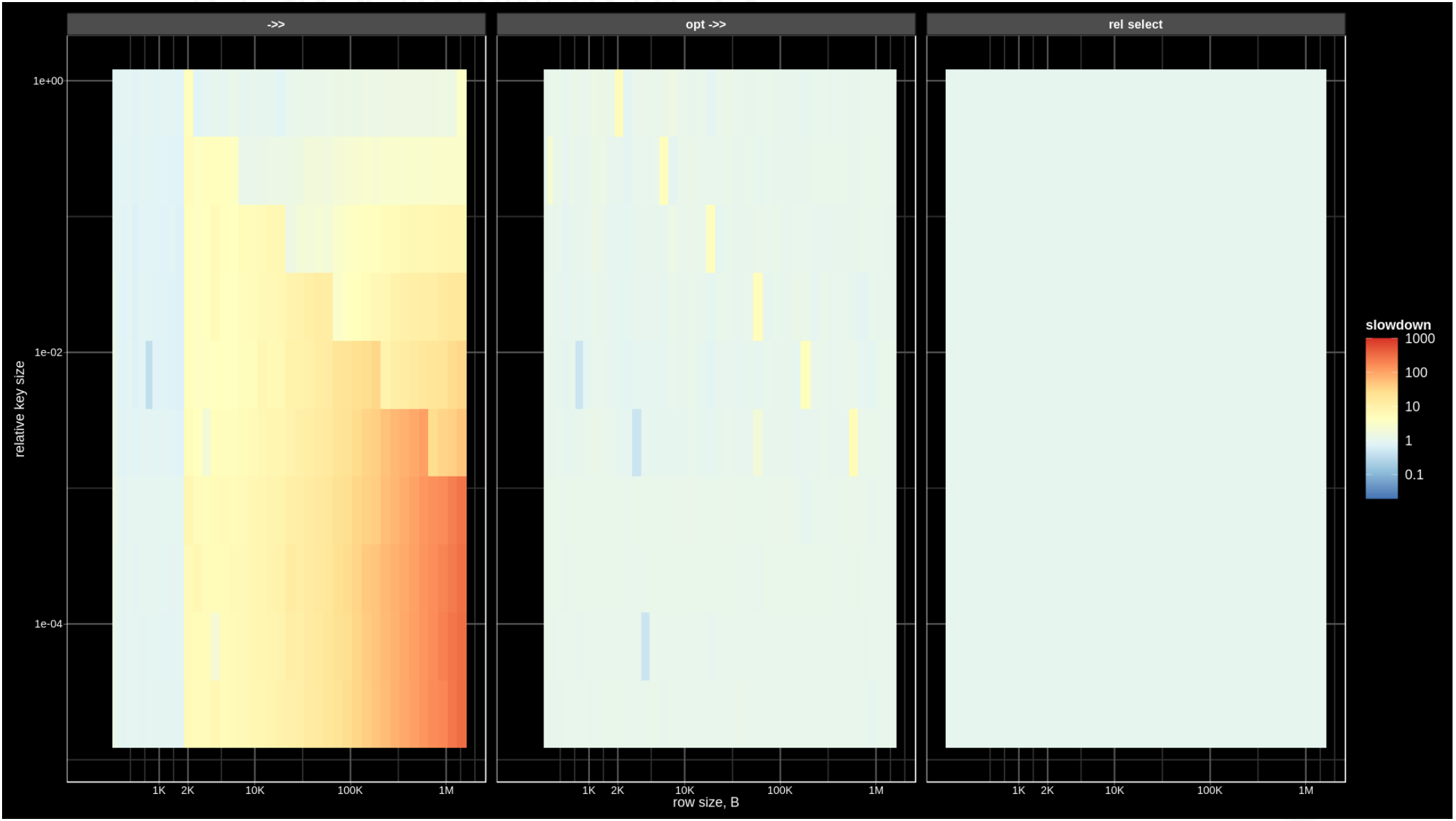
PostgresPro

JSONB vs Relational: access key, update



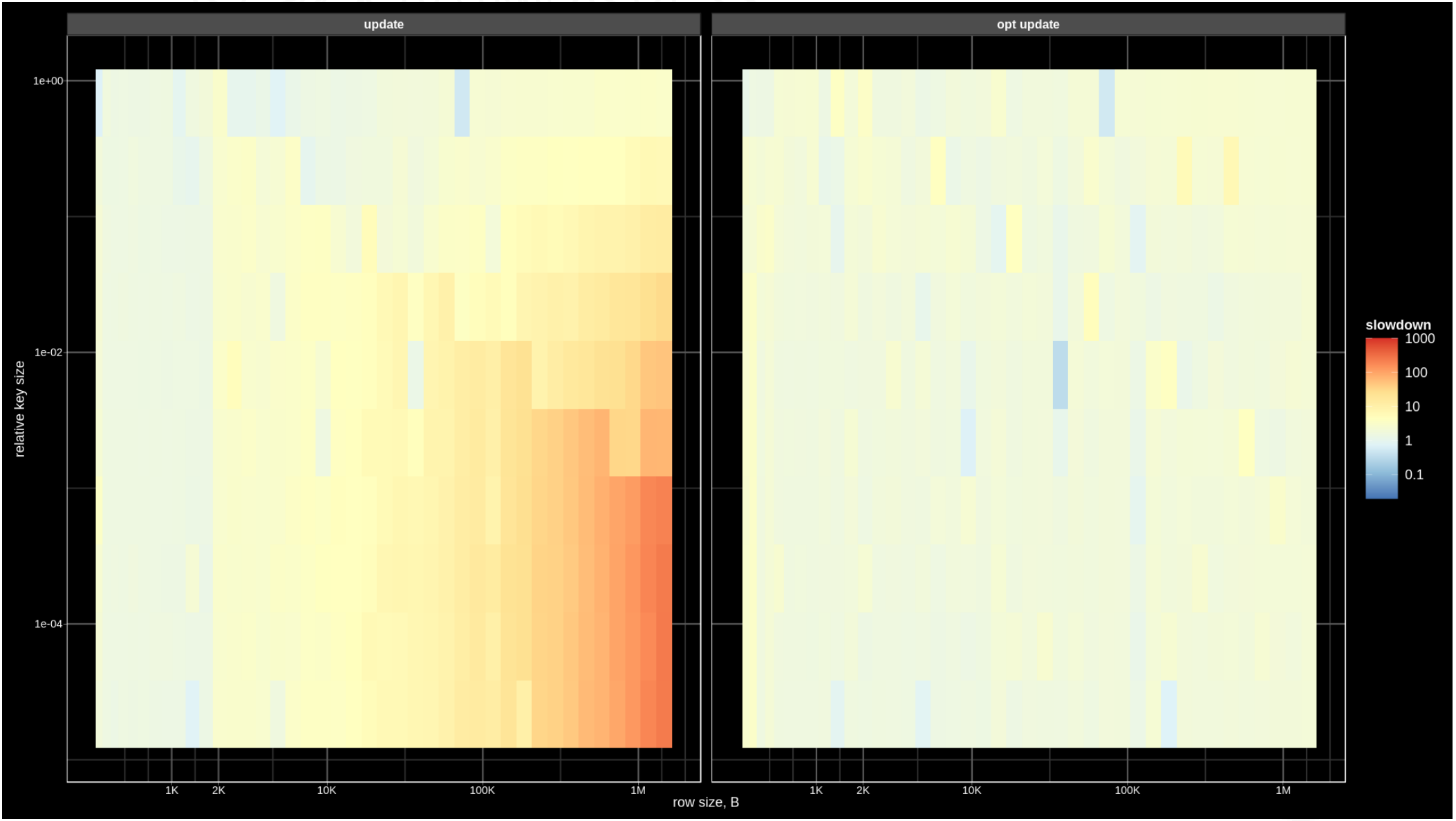
PostgresPro

JSONB vs Relational: access key slowdown



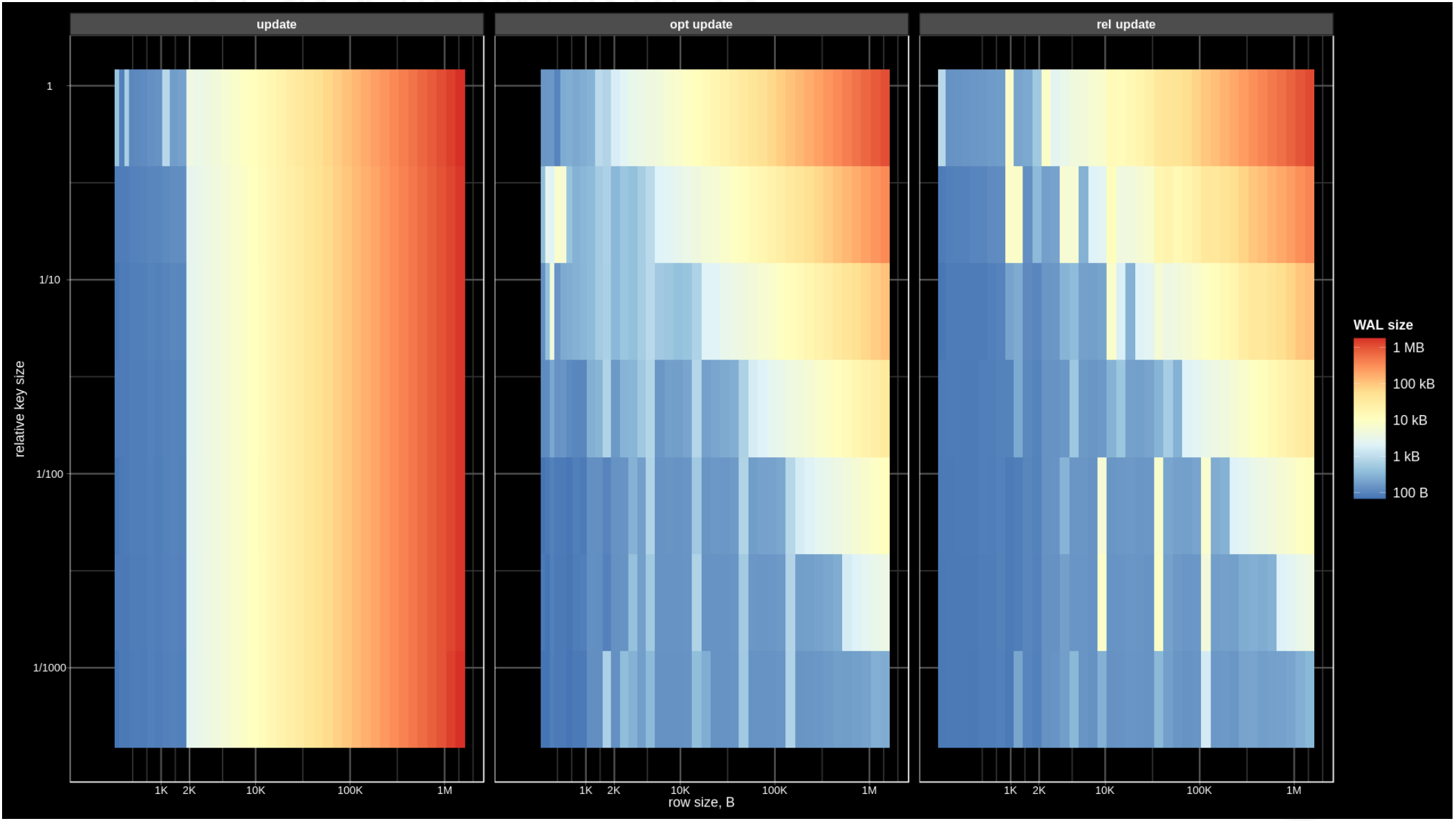
PostgresPro

JSONB vs Relational: update key slowdown



PostgresPro

JSONB vs Relational: update key WAL



PostgresPro

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:

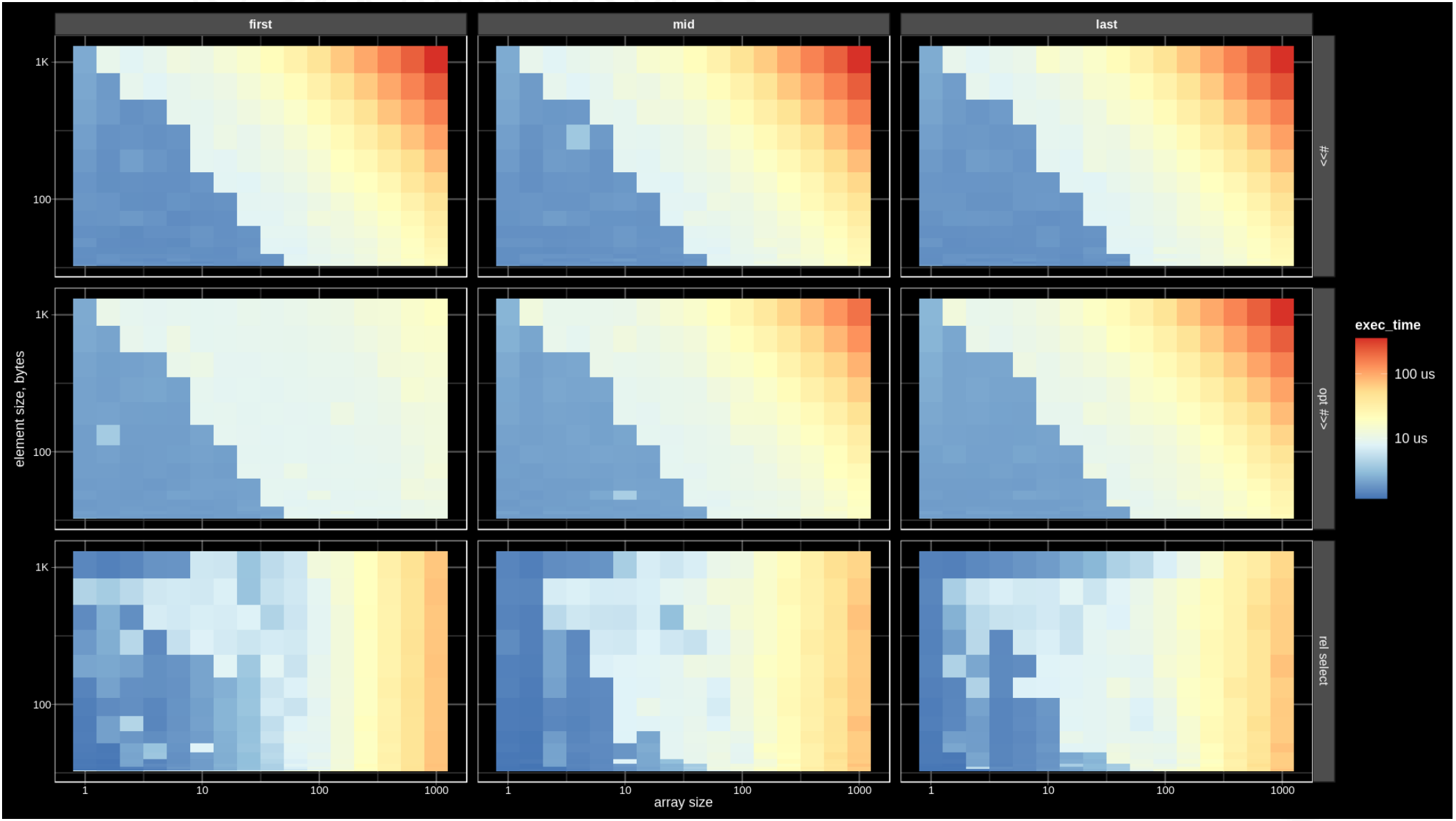
```
SELECT textsend(val)
FROM test_jsonb_arrays_rel_elems
WHERE idx = $1 AND array_size = $2 AND elem_size = $3;
```

- $1 =

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

PostgresPro

JSONB vs Relational: access array member



PostgresPro

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:

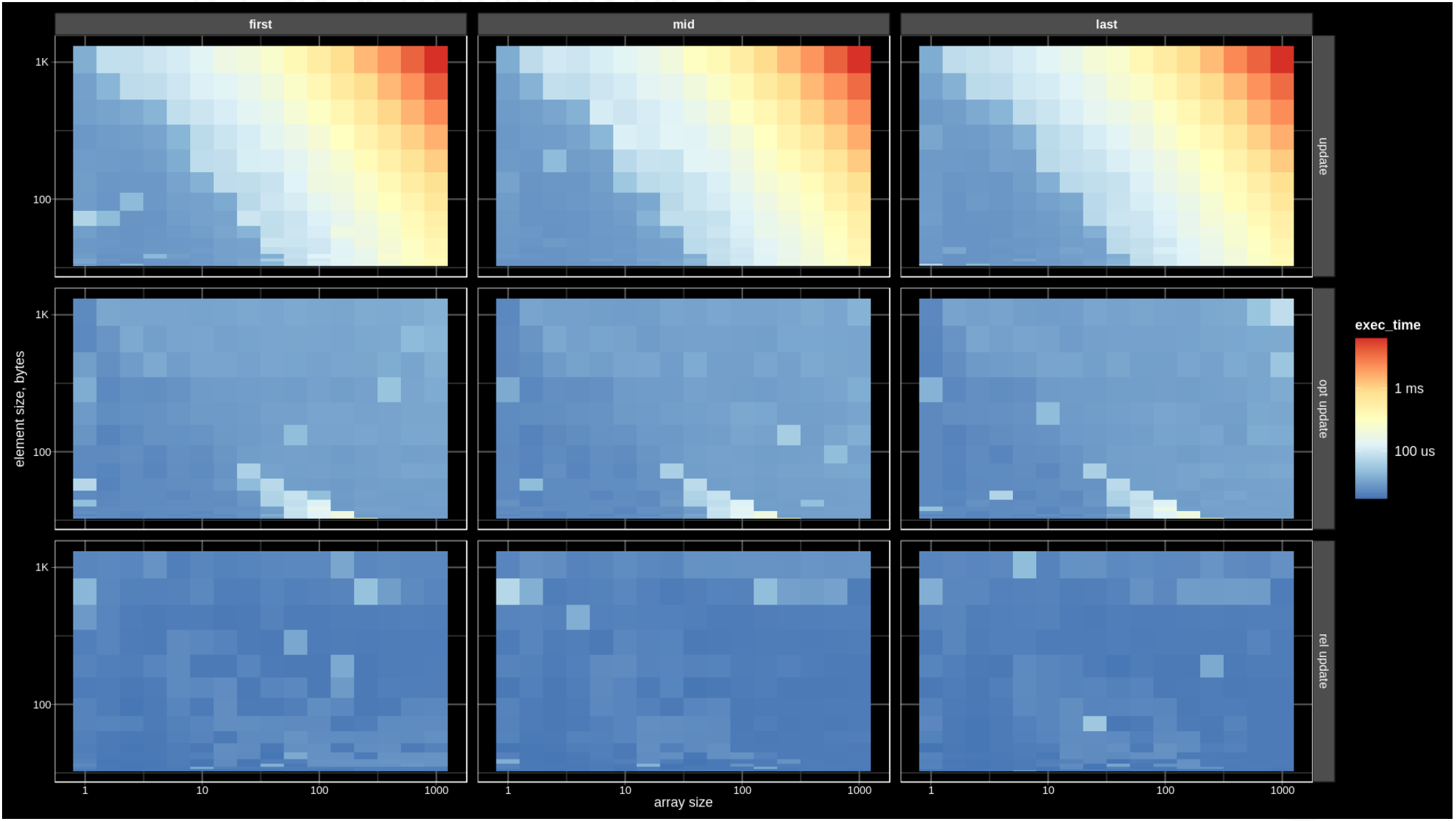
```
UPDATE test_jsonb_arrays_rel_elems
SET val = $3
WHERE id = $2 AND idx = $1;
```

- $1 =

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

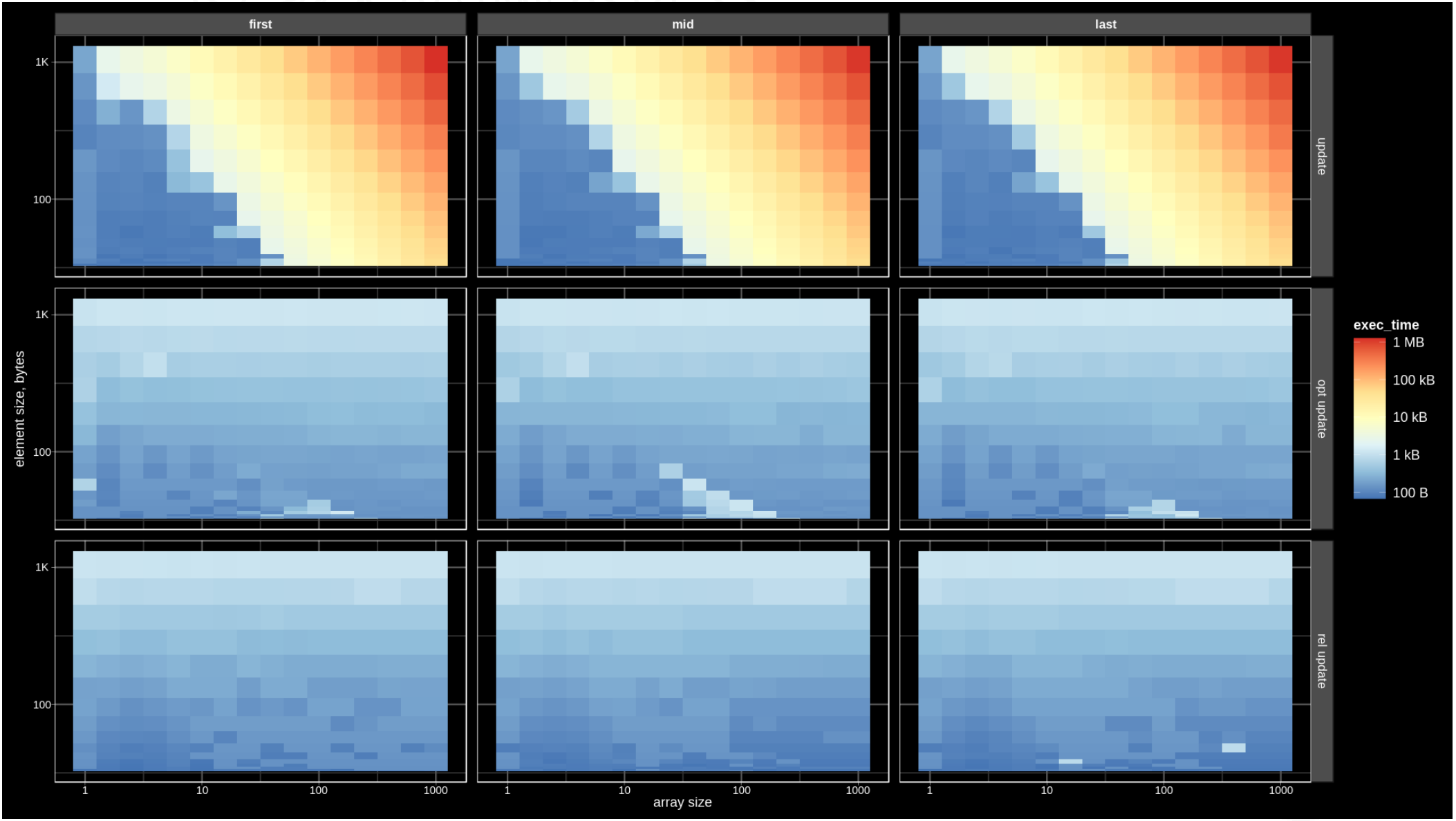
PostgresPro

JSONB vs Relational: update array member



PostgresPro

JSONB vs Relational: WAL update array member



PostgresPro

Conclusions

- JSONB is good
  - Full object access (microservices) — faster than relational way (joins, aggregate,difficult tuning)
  - Storing short metadata as a separate jsonb field
- Currently (PG14) not optimized for
  - TOASTed jsonb (updates)
  - Access to array members
- We demonstrated many optimizations
  - Order of magnitudes speedup for SELECT and UPDATE.
- How integrate them to the Postgres — that is the question !
  - Data type aware TOAST

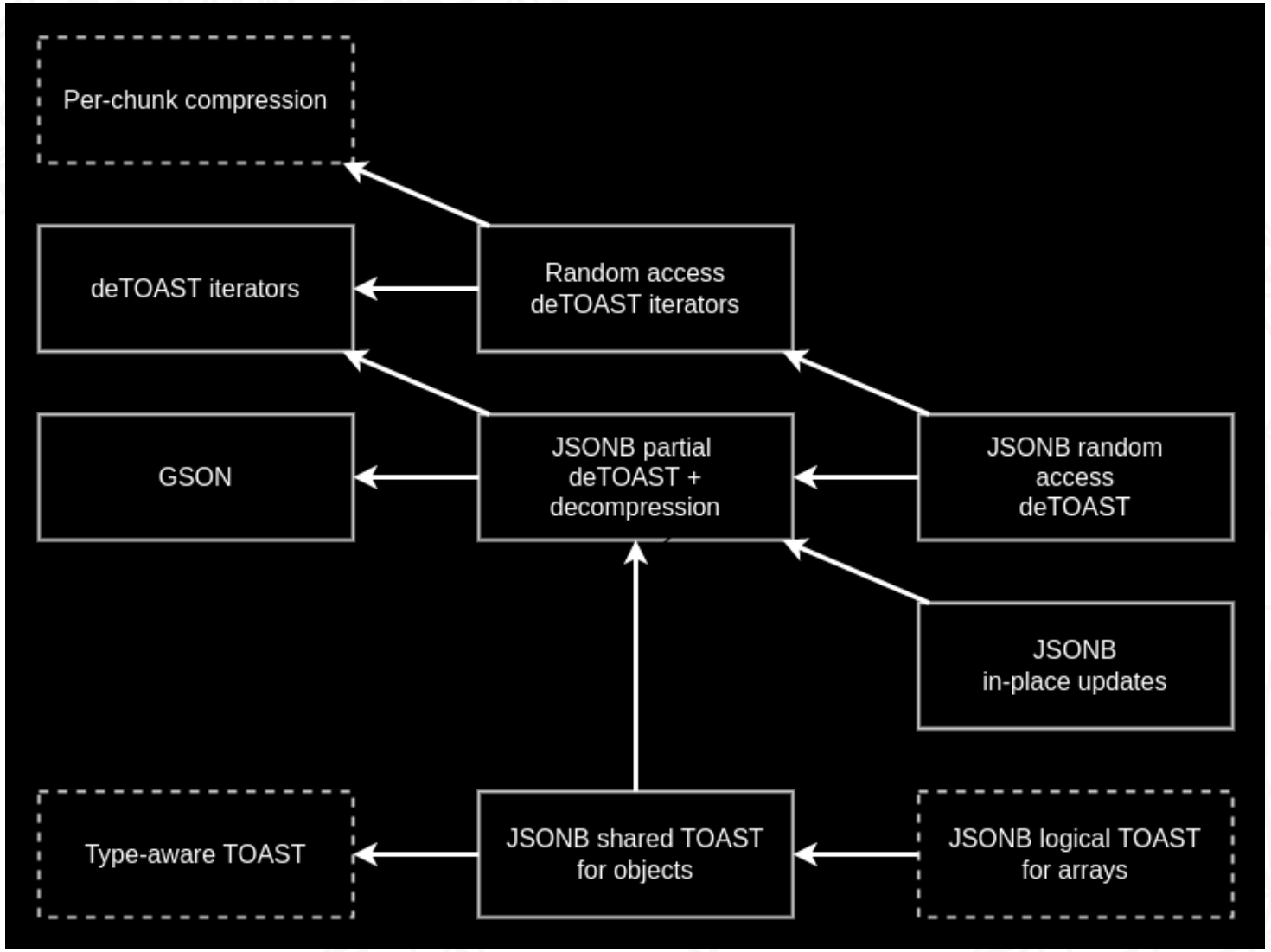
PostgresPro

TODO

- Random access to objects keys and array elements of TOAST-ed jsonb
  - Physical level — add compression to the sliced detoast (easy)
  - Logical level - shared toast with array support (difficult, require jsonb modification — new storage for array, JSONB API + range support)

PostgresPro

Roadmap and patch set



PostgresPro

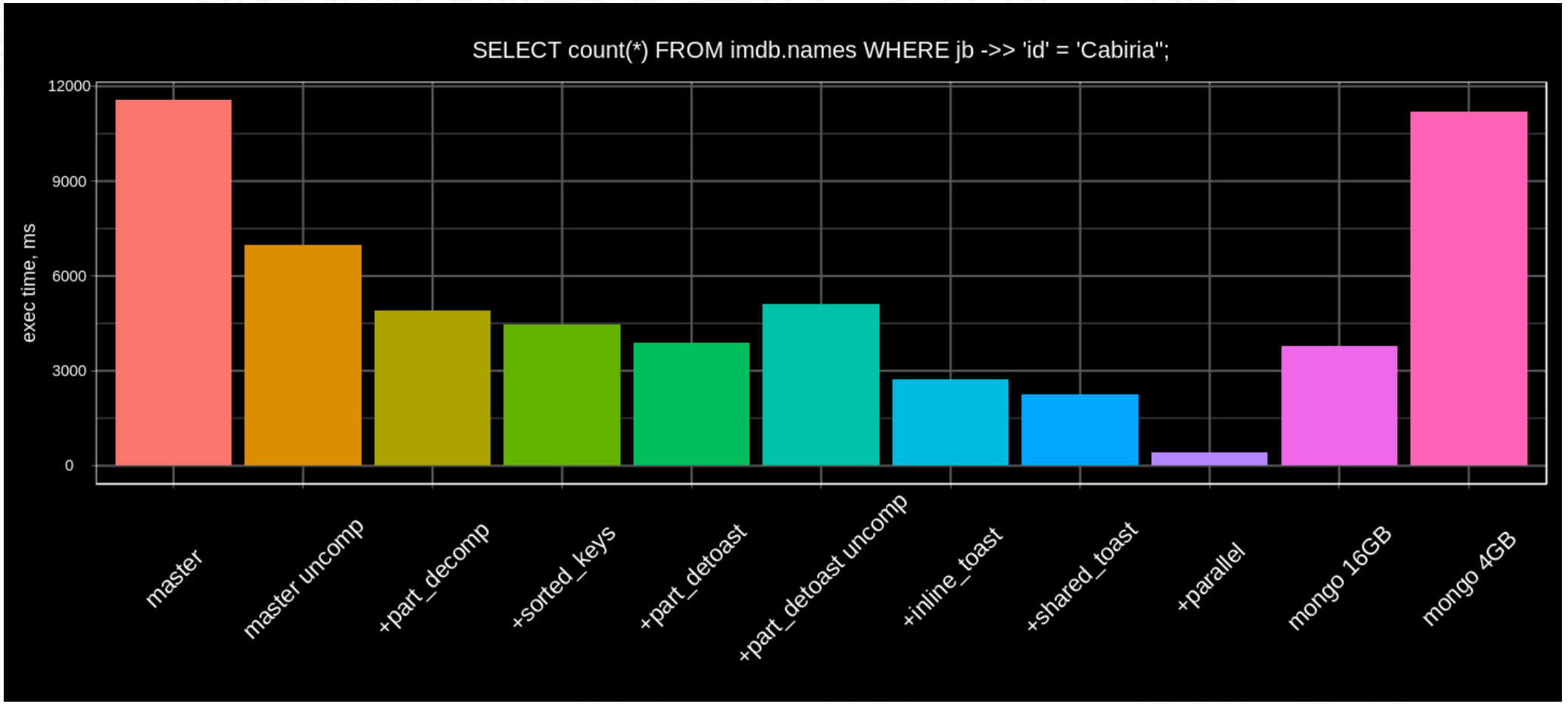
References

- Our experiments:
  - Details - http://www.sai.msu.su/~megera/postgres/talks/jsonb-pgvision-2021.pdf
  - Github: https://github.com/postgrespro/postgres/tree/jsonb_updatable_toast
  - 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
- Join JSONB development team:
  - obartunov@postgrespro.ru

PostgresPro

Non-scientific comparison PG vs Mongo

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



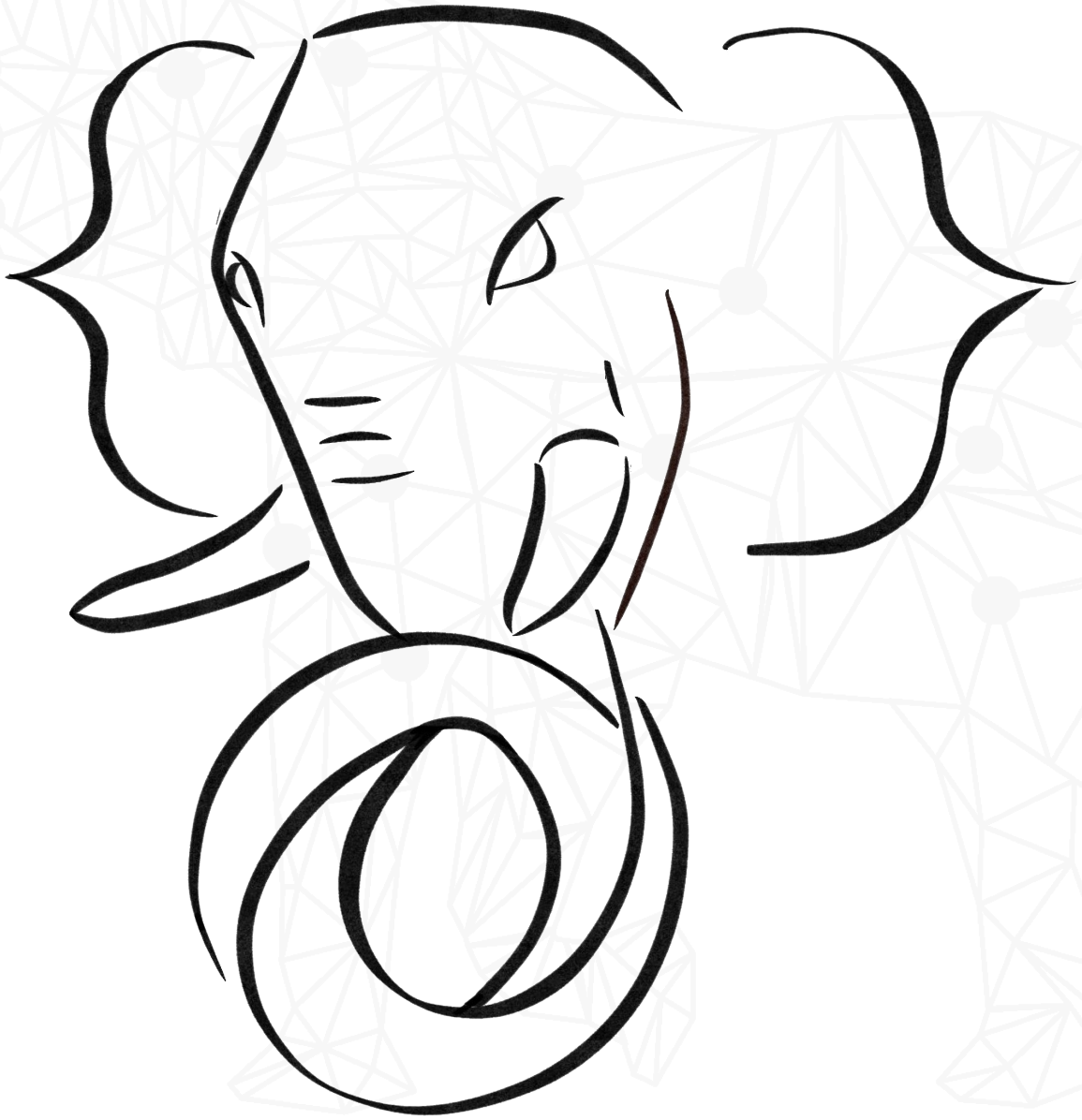
PostgresPro

When children climb trees and tear their pants off,
we can forbid them to do so or teach them climbing techniques.

PostgresPro

Let’s not say that json is the wrong technology,

Let’s make json a first class citizen instead.



PostgresPro


ALL
YOU
NEED
POSTGRES
IS