

ONE TOAST FITS ALL

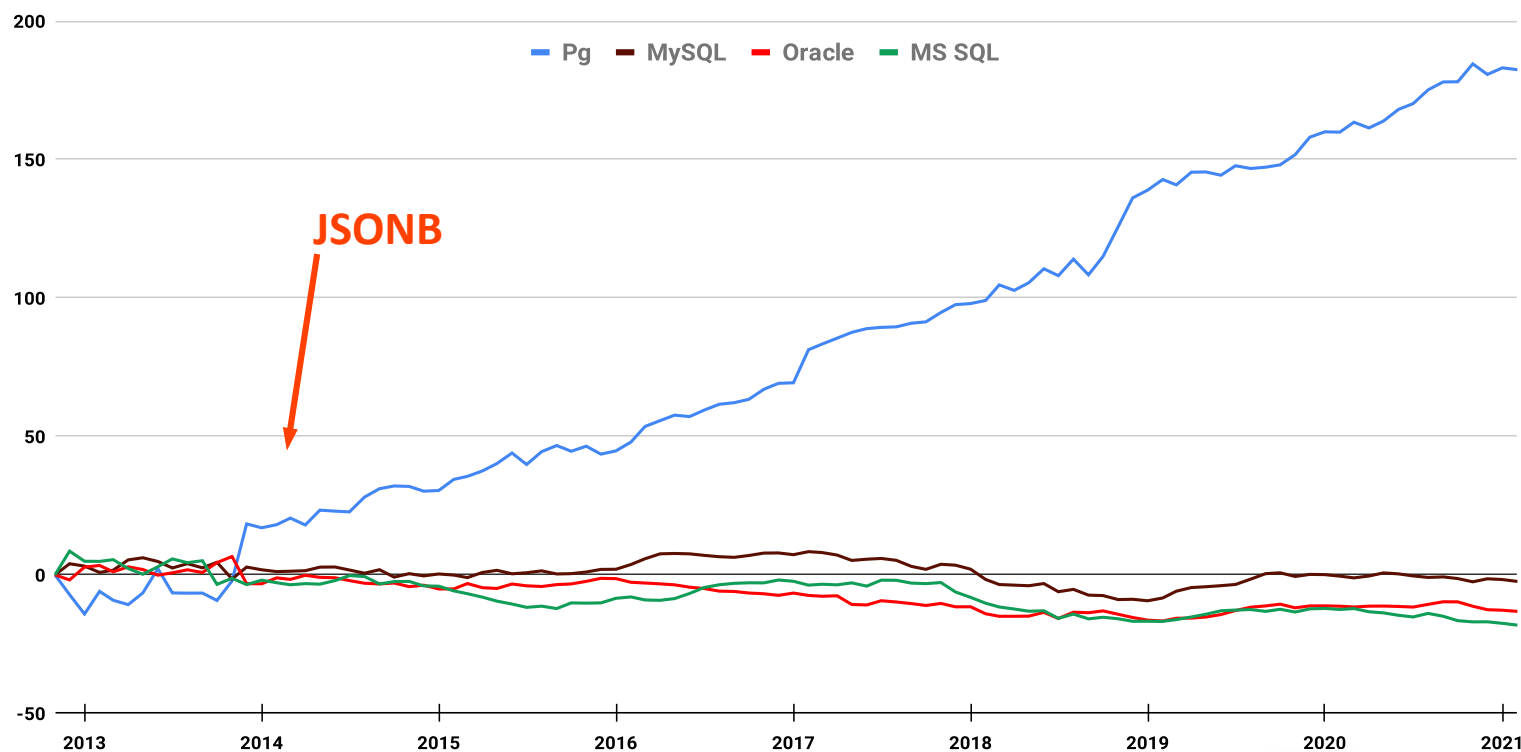
toast

<http://www.sai.msu.su/~megera/postgres/talks/toast-highload-2022.pdf>

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

Relative Growth



PG15: SQL/JSON/TABLE

2022 by #postgrespro



db-engines.com/en/ranking



PostgresPro

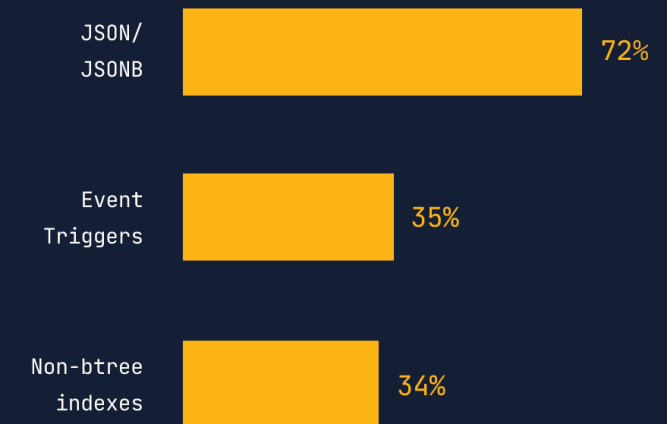
Why this talk ?

- Blossom of Microservice architecture
- Startups want/need JSON[B]
- JSONB is one of the main driver of Postgres popularity
- One-Type-Fits-All
- Client app — Backend - Database use JSON
- All server side languages support JSON
- JSON relaxed ORM (Object-Relational Mismatch), mitigate contradictions between code-centric and data-centric paradigms.
- Performance of JSONB (not only) can be improved by several orders of magnitude with proper modification of TOAST. **How to integrate improvements into PG CORE !?**

Top 3 features used to organize and access data in production apps

JSON/JSONB, Event triggers, and Non-btree indexes are the top 3 features respondents use in their production apps.

[View full question](#)





Case 1: TOAST for JSONB

The Curse of TOAST: Unpredictable performance

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

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

```
-----
Seq Scan on test  (actual rows=10000 loops=1)
  Buffers: shared hit=2500
  Planning Time: 0.050 ms
  Execution Time: 6.147 ms
(4 rows)
```

Small update cause significant slowdown !

```
=# 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 (actual rows=10000 loops=1)
  Buffers: shared hit=30064
  Planning Time: 0.105 ms
  Execution Time: 38.719 ms
(4 rows)
```

Pageinspect: 64 pages with 157 tuples per page

WHY 30064 pages !!!!

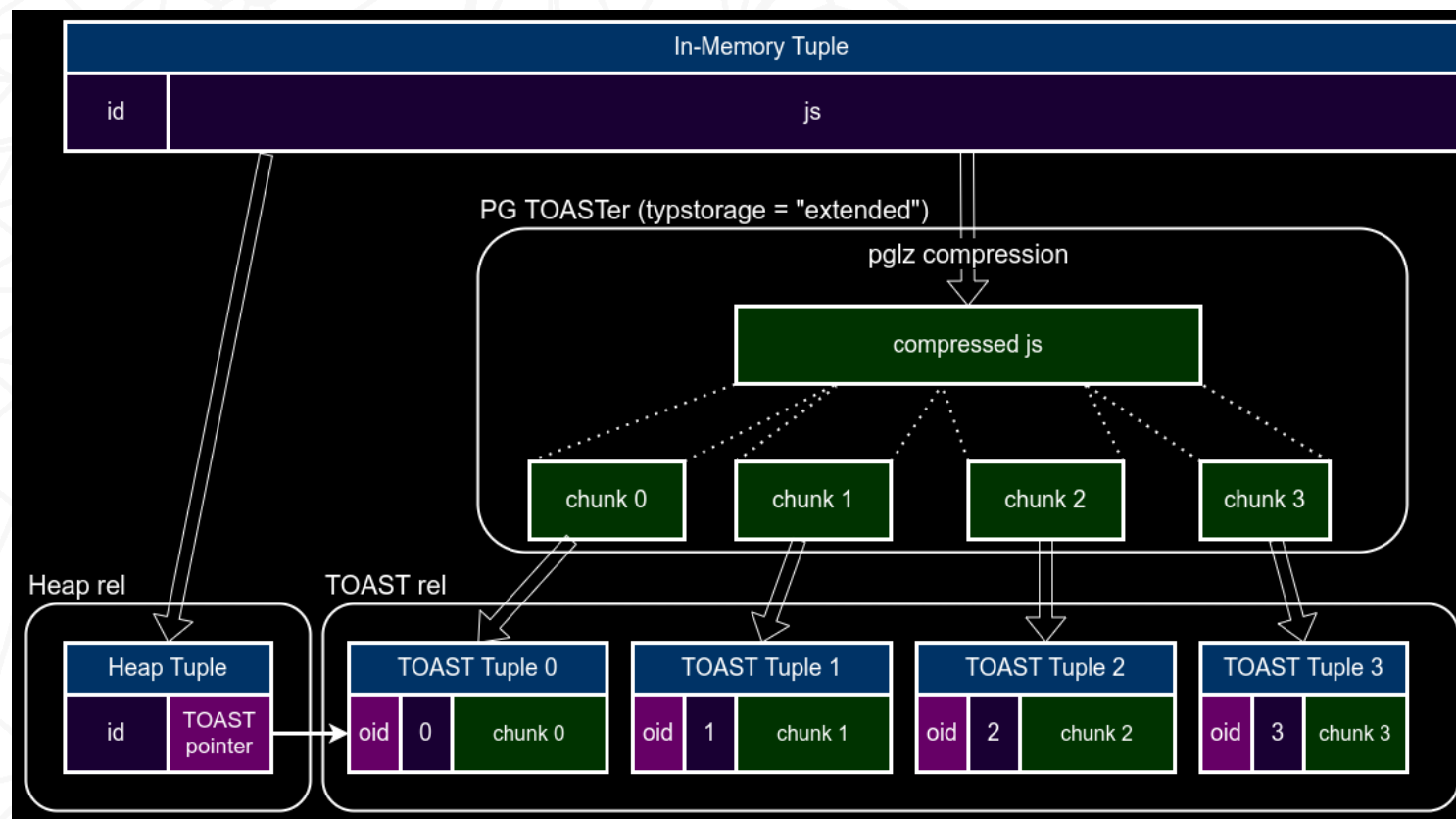
TOAST Explained

The Oversized-Attribute Storage Technique

- TOASTed (large field) values are compressed, then splitted into the fixed-size TOAST chunks (1996B for 8KB page)

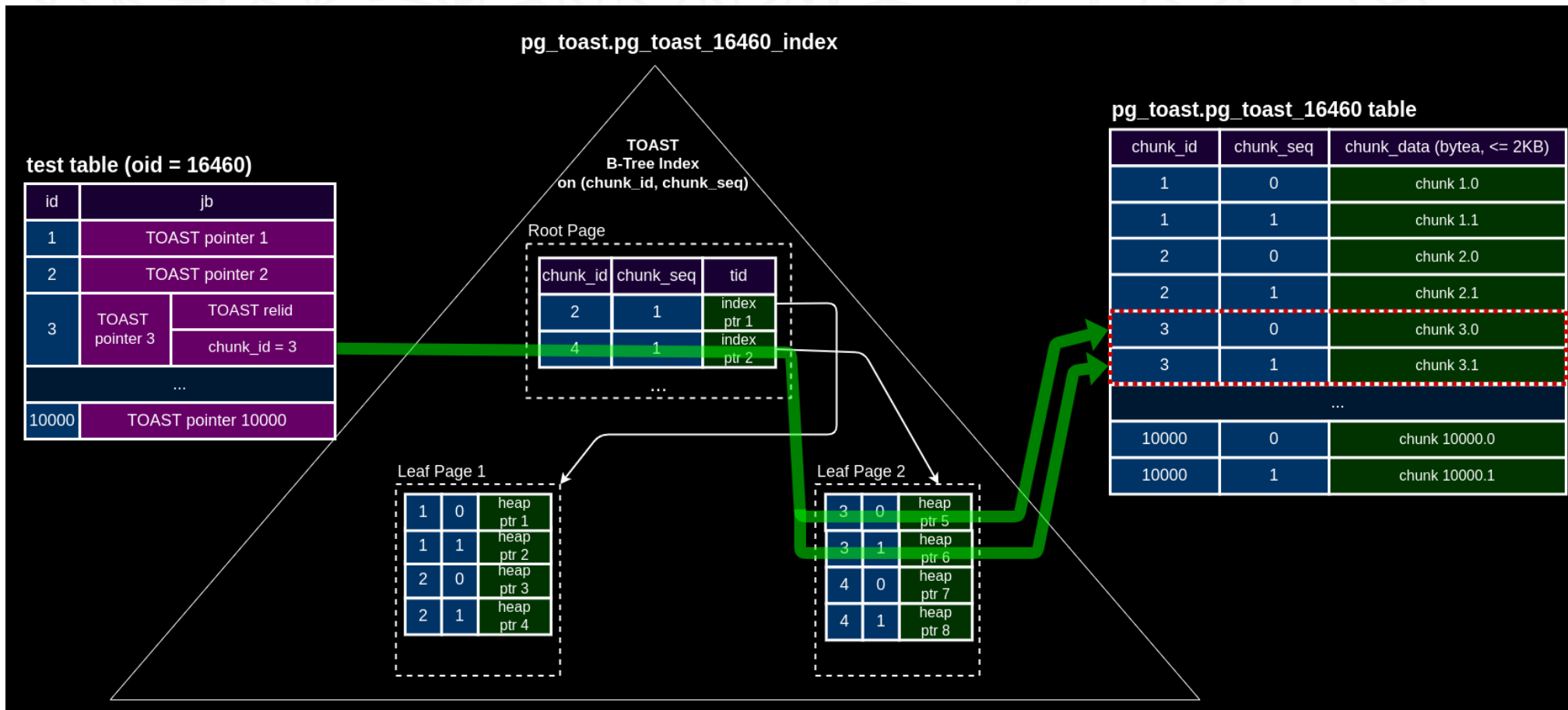
- TOAST chunks (along with generated Oid chunk_id and sequence number chunk_seq) stored in special TOAST relation `pg_toast.pg_toast_XXX`, created for each table containing TOASTable attributes

- 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 pointer has Oid chunk_id and refers to heap tuples with chunks using B-tree index (chunk_id, chunk_seq). Overhead to read only a few bytes from the first chunk can be 3,4 or even 5 index blocks.



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 can be read from the same page, but if JSONB size > Page size (8Kb), then more TOAST heap buffers

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

```
-----  
Seq Scan on test (actual rows=10000 loops=1)  
  Buffers: shared hit=30064  
Planning Time: 0.105 ms  
Execution Time: 38.719 ms  
(4 rows)
```

Table	64
TOAST index	2 * 10000
TOAST table	1 * 10000
Total	30064

Motivational example (synthetic test)

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

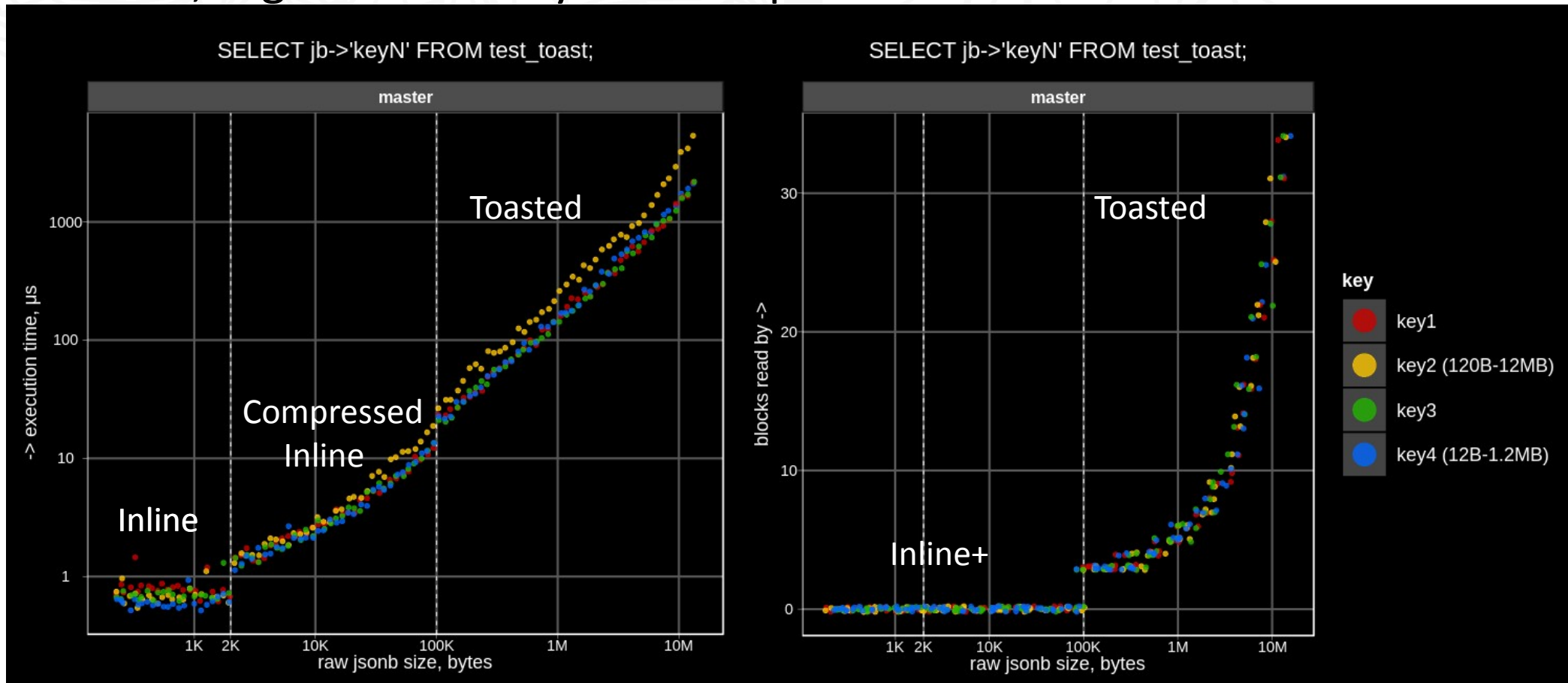
```
CREATE TABLE test_toast AS
SELECT
  i id,
  jsonb_build_object(
    'key1', i,
    'key2', (select jsonb_agg(0) from
              generate_series(1, pow(10, 1 + 5.0 * i / 100.0)::int)), -- 10-100k elems
    'key3', i,
    'key4', (select jsonb_agg(0) from
              generate_series(1, pow(10, 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 (raw size > 100 Kb) jsonbs linearly increase with jsonb size, regardless of key size and position.



Large jsonb is TOASTed !

Jsonb deTOAST improvements goal

Ideal goal: no dependency on jsonb size and position

- Access time $\sim O(\text{level})$
- Update time $\sim O(\text{level}) + O(\text{key size})$
- 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.

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_rankc * 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_rankc * key size)`

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

$O(\text{inline size})$ – short keys (inline size < 2KB)

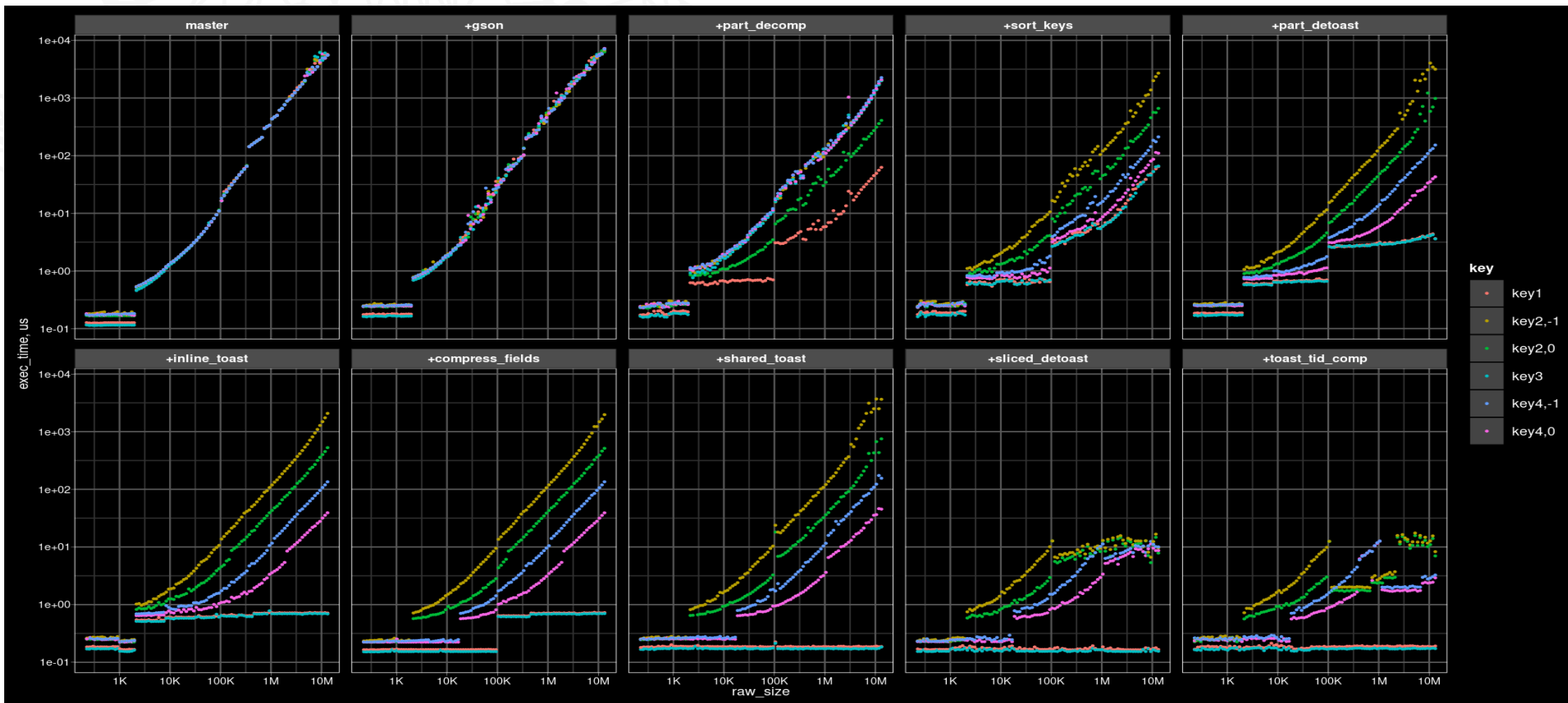
$O(\text{inline size}) + O(\text{key size})$ – keys in chunks

$O(\text{jsonb size})$ – inline TOAST

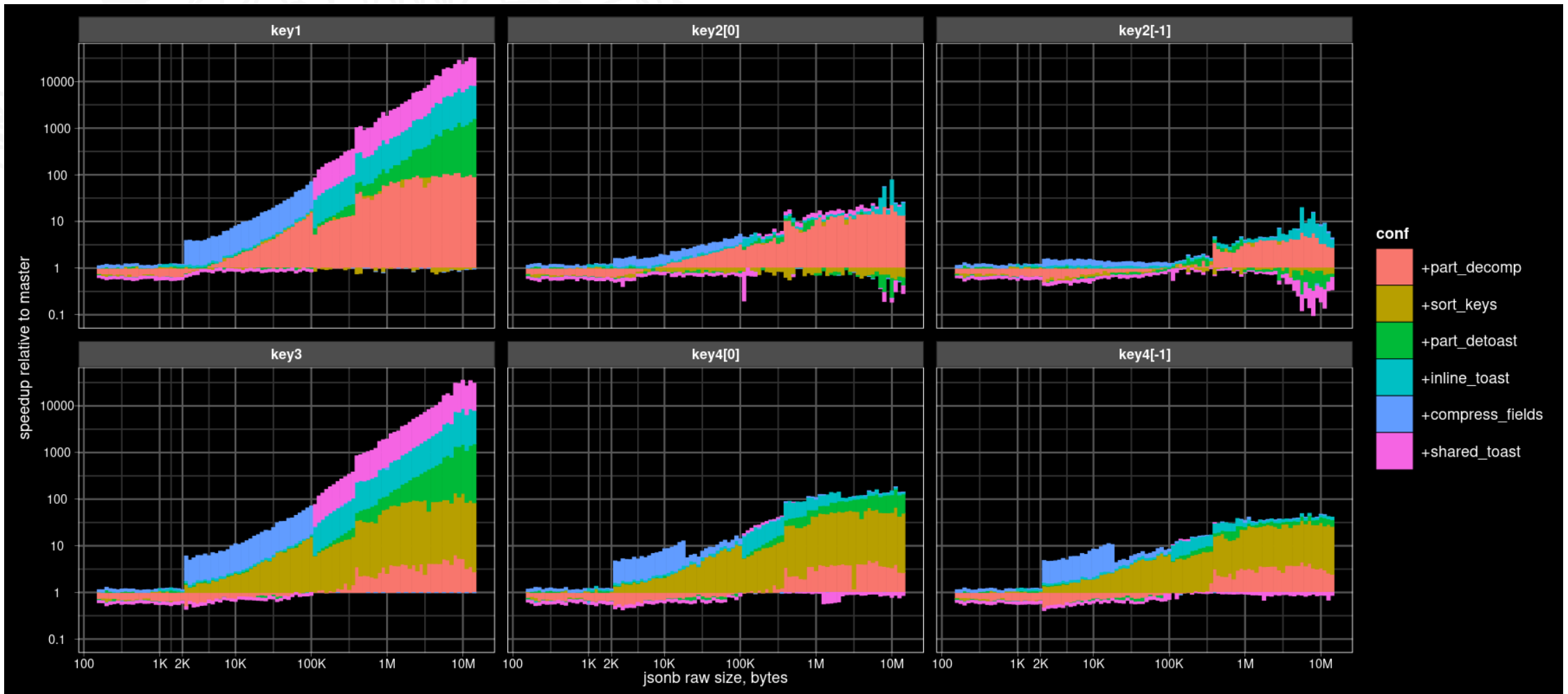
Jsonb deTOAST improvements

- 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:
 - $O(\text{element size})$ – if in-place update and new value fits into inline
 - $O(\text{array size})$ – otherwise

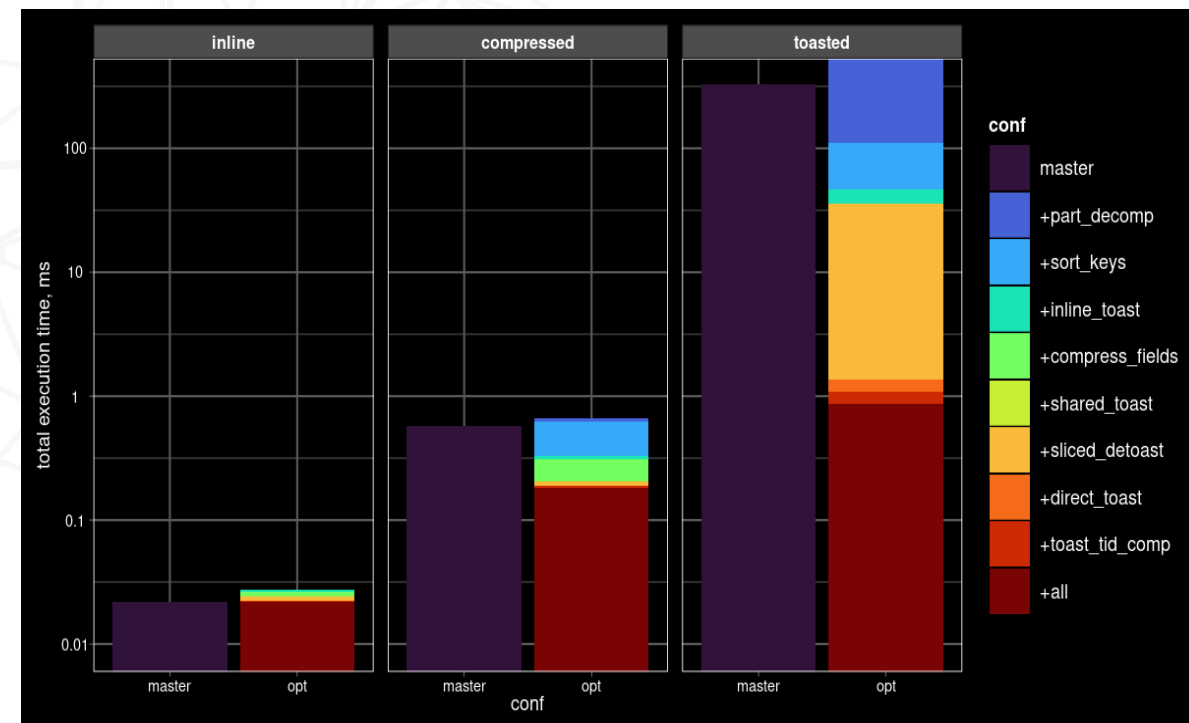
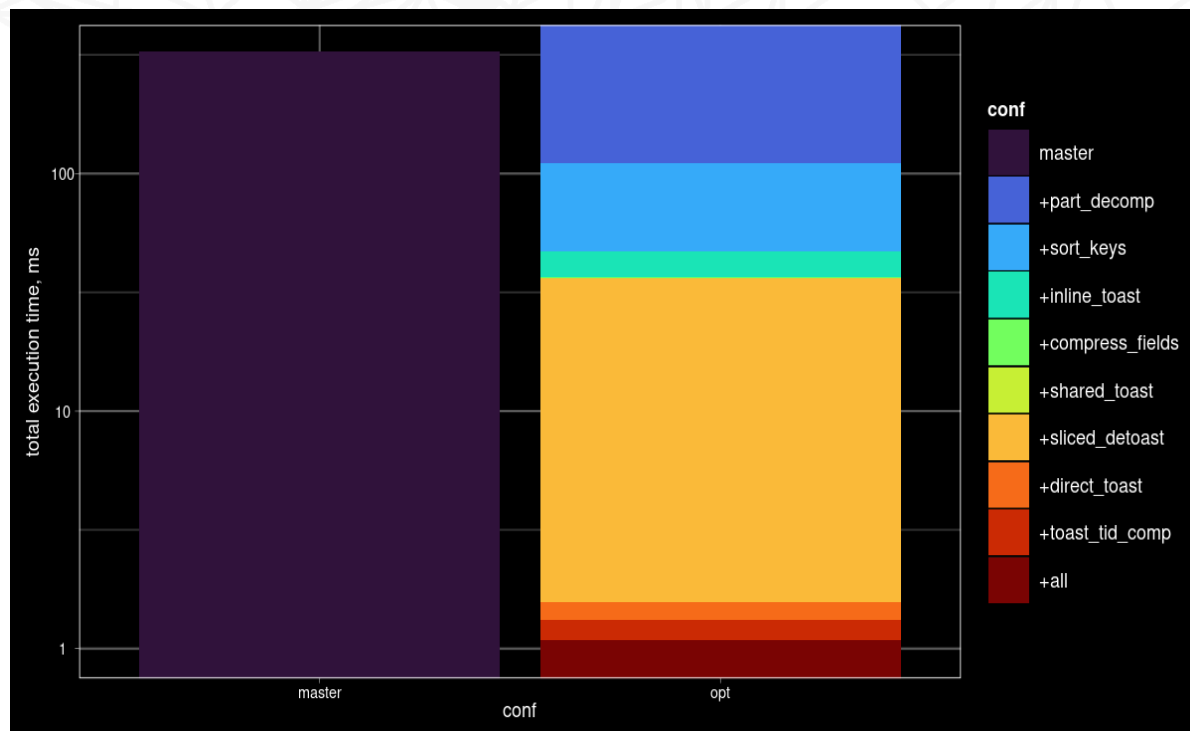
JSONB optimizations



Relative speedup: TOAST opt. vs Master



Jsonb aggregated statistics



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.

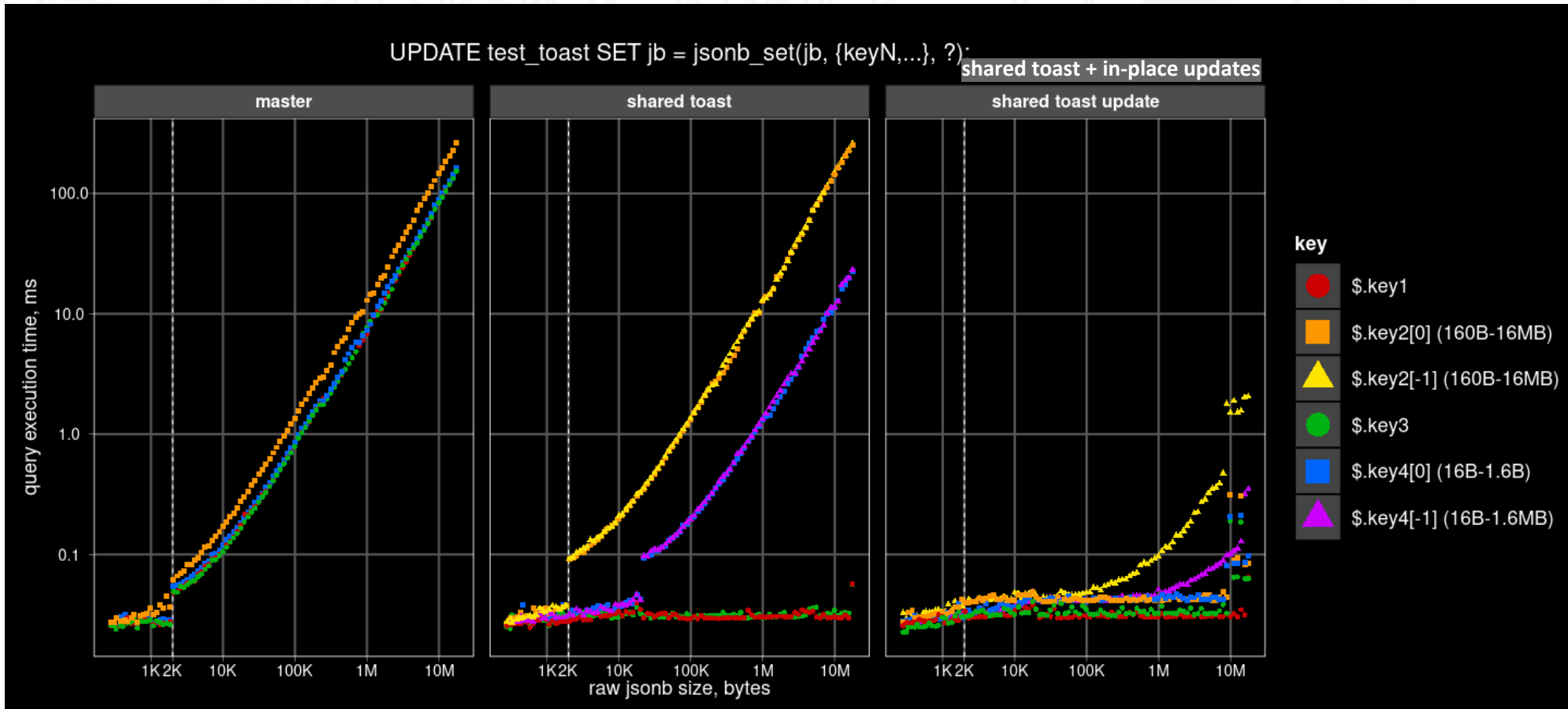
TOAST works only with binary BLOBs, it does not try to find differences between old and new values of updated attributes. So, when the TOASTed attribute is being updated (does not matter at the beginning or at the end and how much data is changed), its chunks are simply fully copied. The consequences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

UPDATE JSONB: Query execution time

Update time of array elements depends on their position:

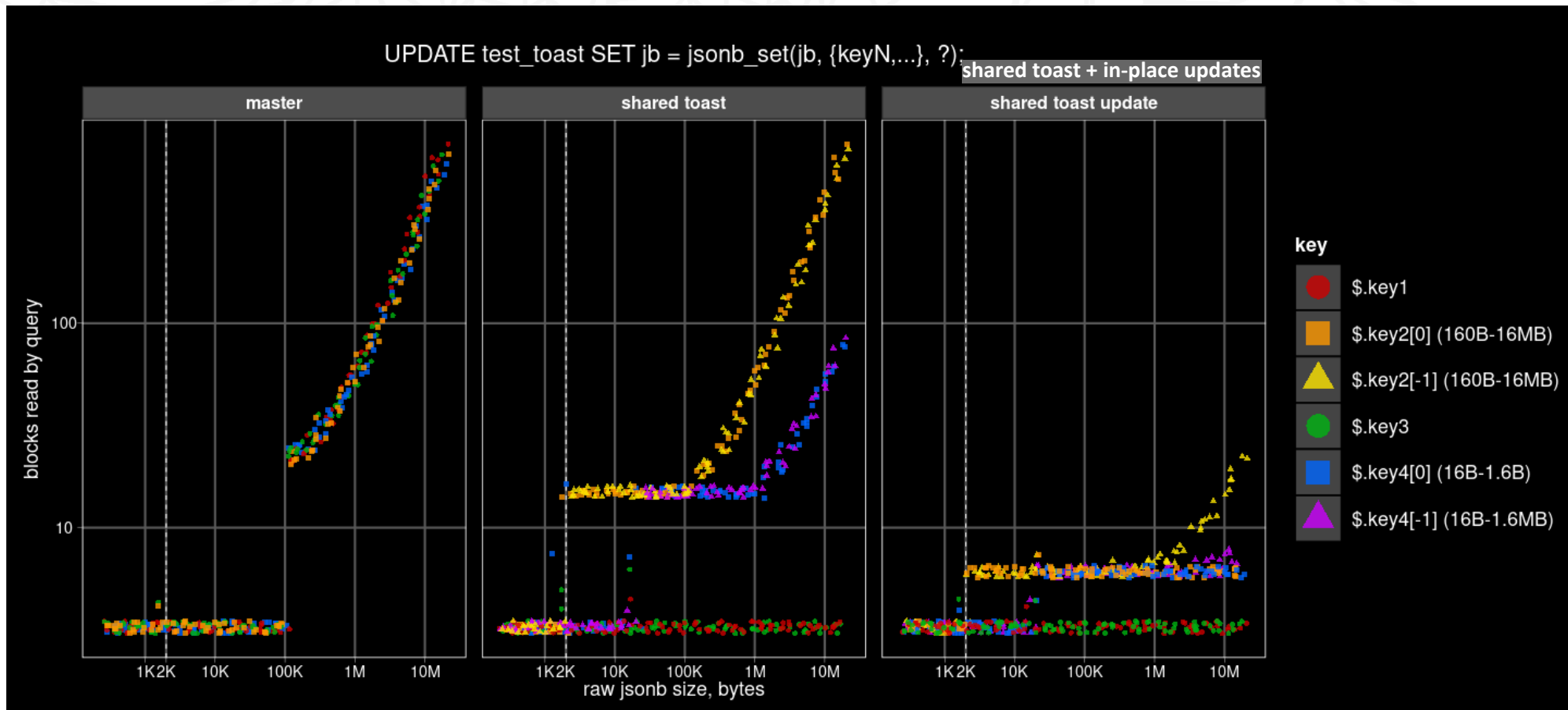
- First elements updated very fast (like inline fields)
- Last elements updated slower (need to read the whole JEntry array)



UPDATE JSONB: Blocks read

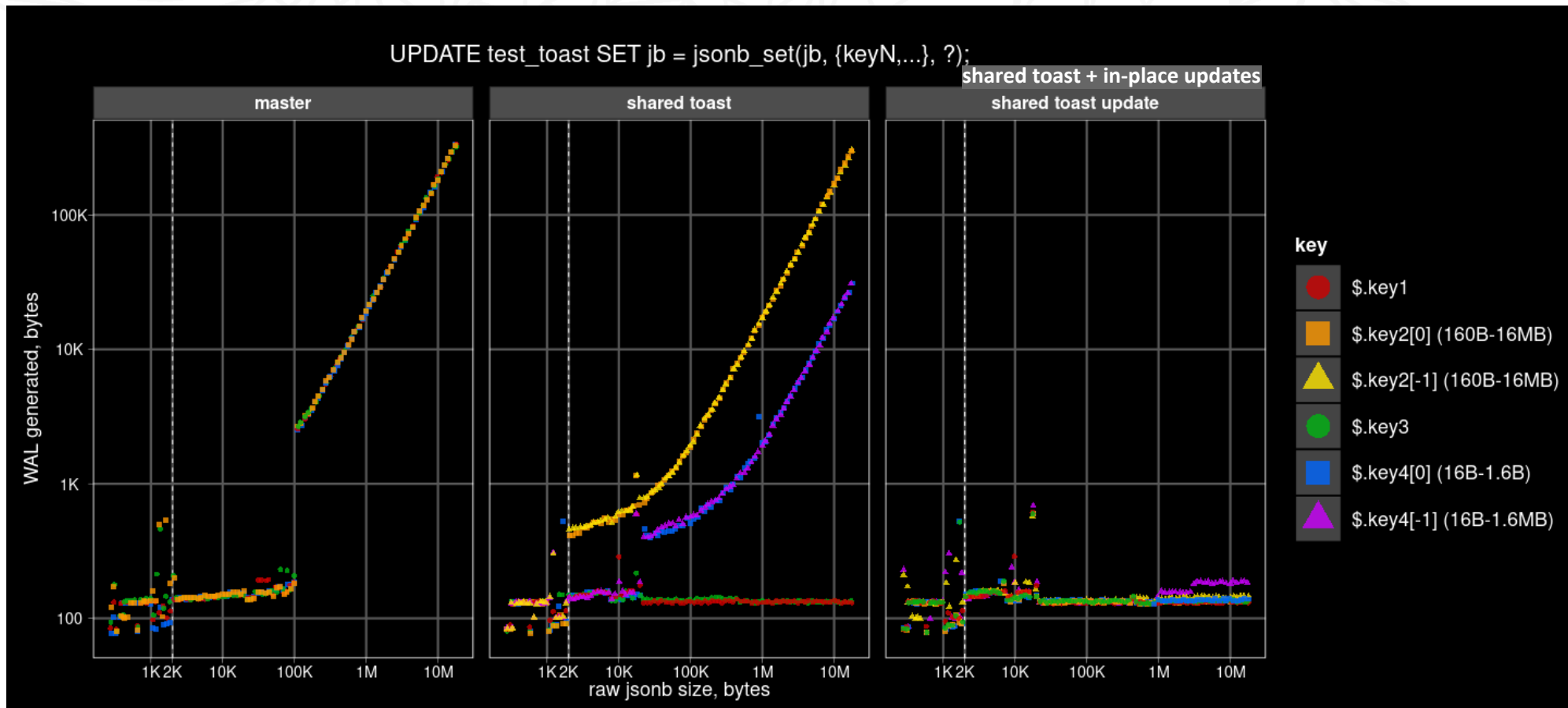
Number of blocks read depends on element position:

- First elements do not require reading of additional blocks
- Last elements require reading the whole JEntry array ($4B \times \text{array size}$)



UPDATE JSONB: WAL traffic

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



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)



Case 2: TOAST for Appendable BYTEA

Motivational example

- A table with 100 MB bytea (uncompressed):

```
CREATE TABLE test (data bytea);  
ALTER TABLE test ALTER COLUMN data SET STORAGE EXTERNAL;  
INSERT INTO test SELECT repeat('a', 100000000)::bytea data;
```

- Append 1 byte to bytea:

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)  
UPDATE test SET data = data || 'x'::bytea;
```

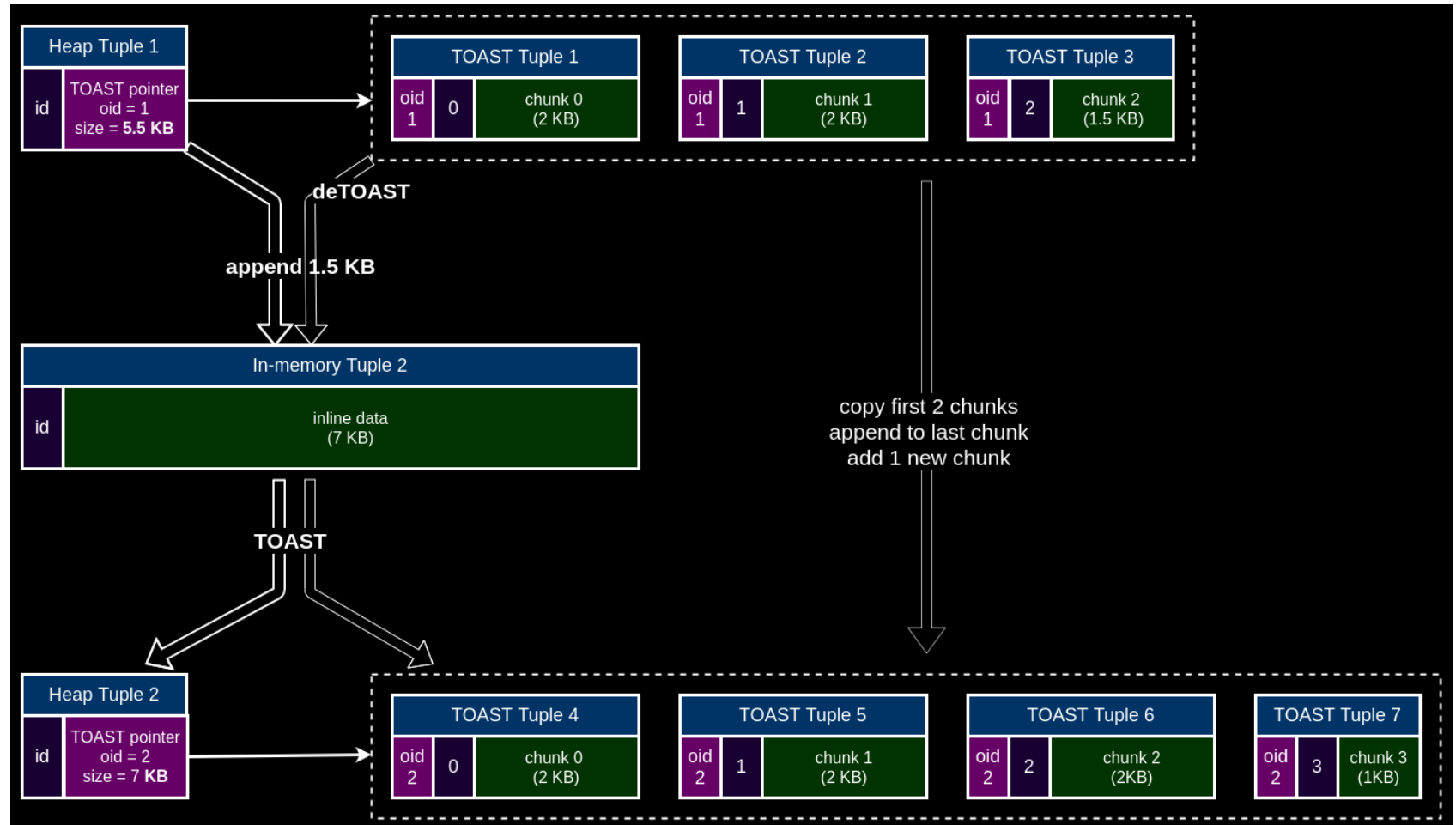
```
Update on test (actual time=1359.229..1359.232 rows=0 loops=1)  
  Buffers: shared hit=238260 read=12663 dirtied=25189 written=33840  
    -> Seq Scan on test (actual time=155.499..166.509 rows=1 loops=1)  
          Buffers: shared hit=12665  
Planning Time: 0.127 ms  
Execution Time: 1382.959 ms
```

- >1 second to append 1 byte !!!

Table size doubled to 200 MB, 100 MB of WAL generated.

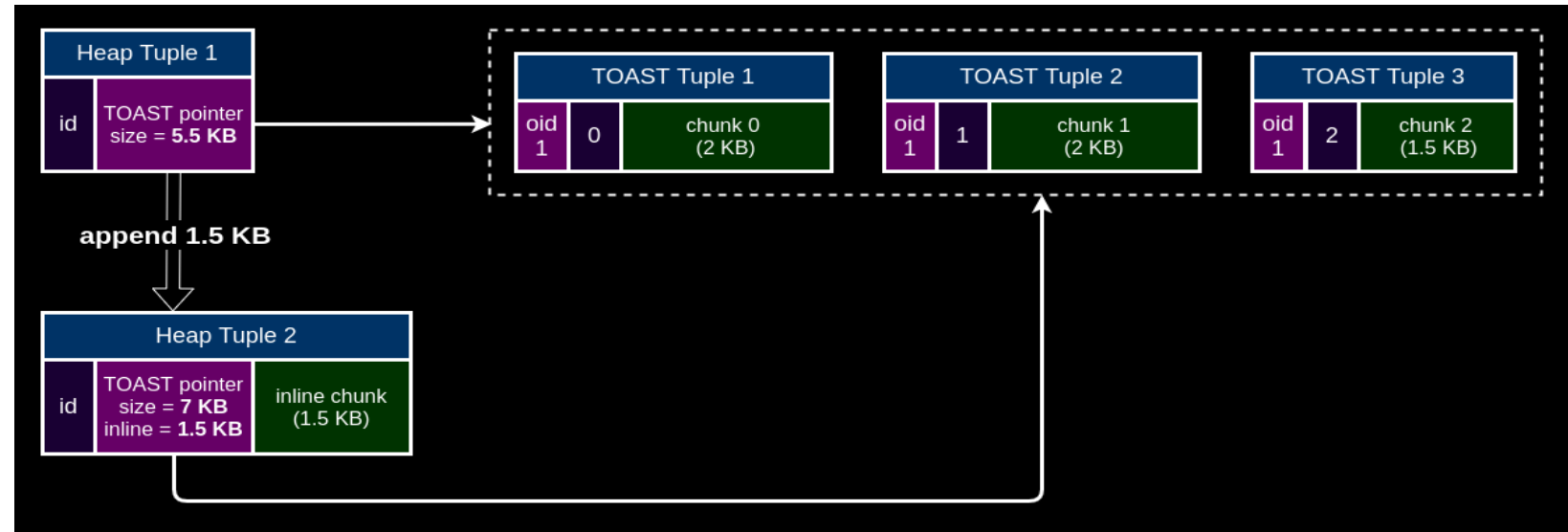
Motivational example (explanation)

- Current TOAST is not sufficient for partial updates
- All data is deTOASTed before in-memory modification
- Updated data is TOASTed back after modification with new TOAST oid



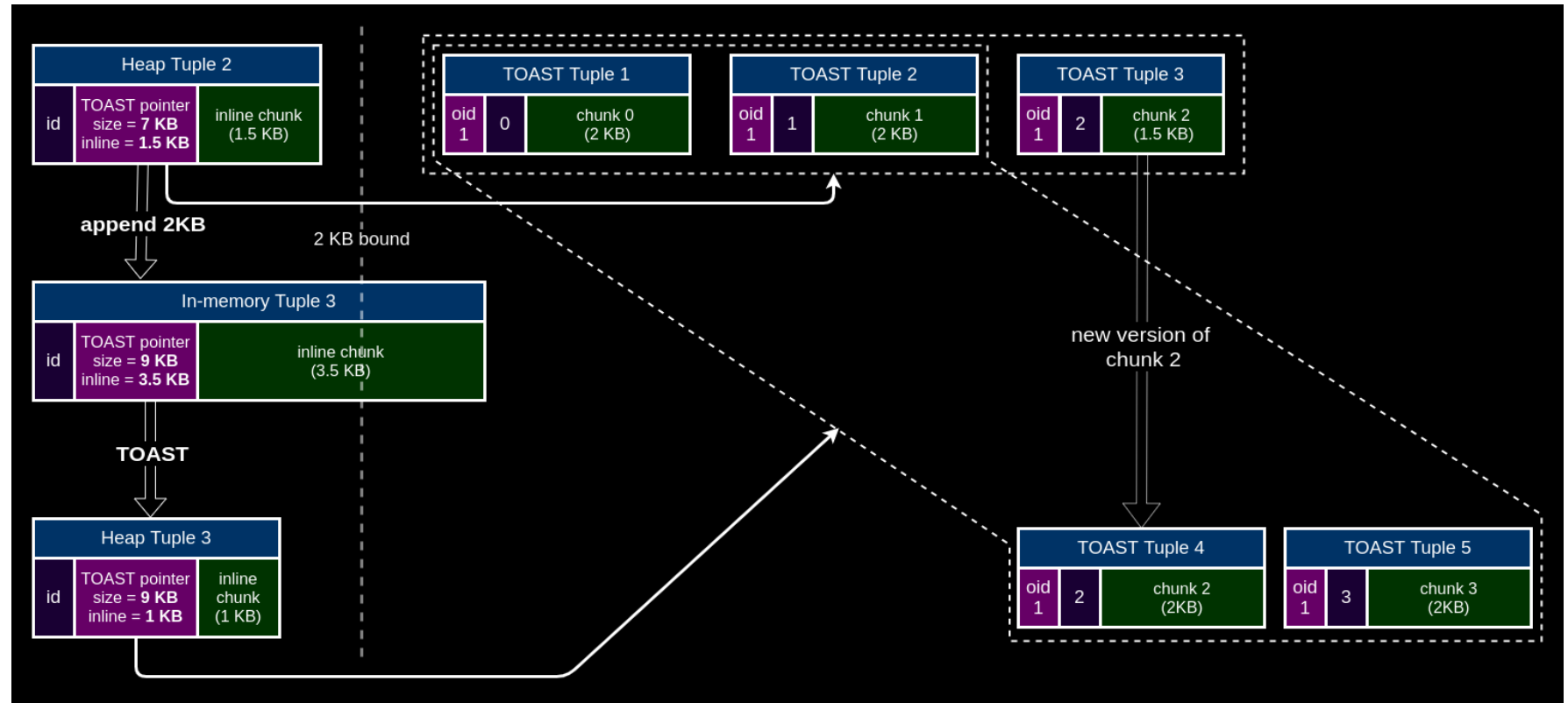
Solution

- Special datum format: TOAST pointer + inline data
- Inline data serves as a buffer for TOASTing
- Operator `||` does not deTOAST data, it appends inline data producing datum in the new format



Solution

- When size of inline data exceeds 2 KB, TOASTER recognizes changes in old and new datums and TOASTs only the new inline data with the same TOAST oid
- Last not filled chunk can be rewritten with creation of new tuple version
- First unmodified chunks are shared



TOAST optimized

- Append 1 byte to bytea:

```
EXPLAIN (ANALYZE, BUFFERS, COSTS OFF)  
UPDATE test SET data = data || 'x'::bytea;
```

Update on test (actual time=0.060..0.061 rows=0 loops=1)

Buffers: **shared hit=2**

-> Seq Scan on test (actual time=0.017..0.020 rows=1 loops=1)

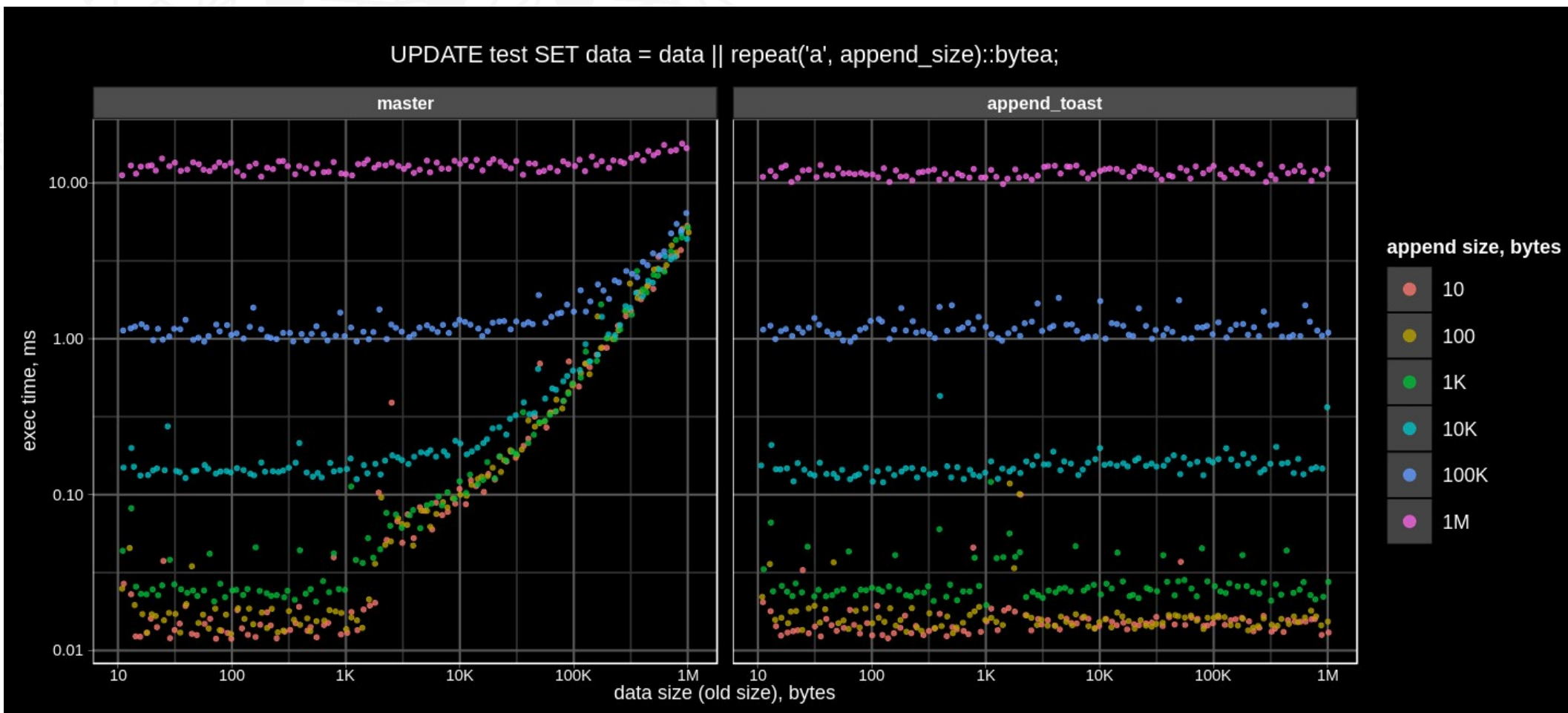
Buffers: shared hit=1

Planning Time: 0.727 ms

Execution Time: **0.496 ms (was 1382 ms)**

- 2750x** speed up!
- Table size remains 100 MB
- Only 143 bytes of WAL generated (was 100 MB)
- No unnecessary buffer reads and writes

Results – query execution time

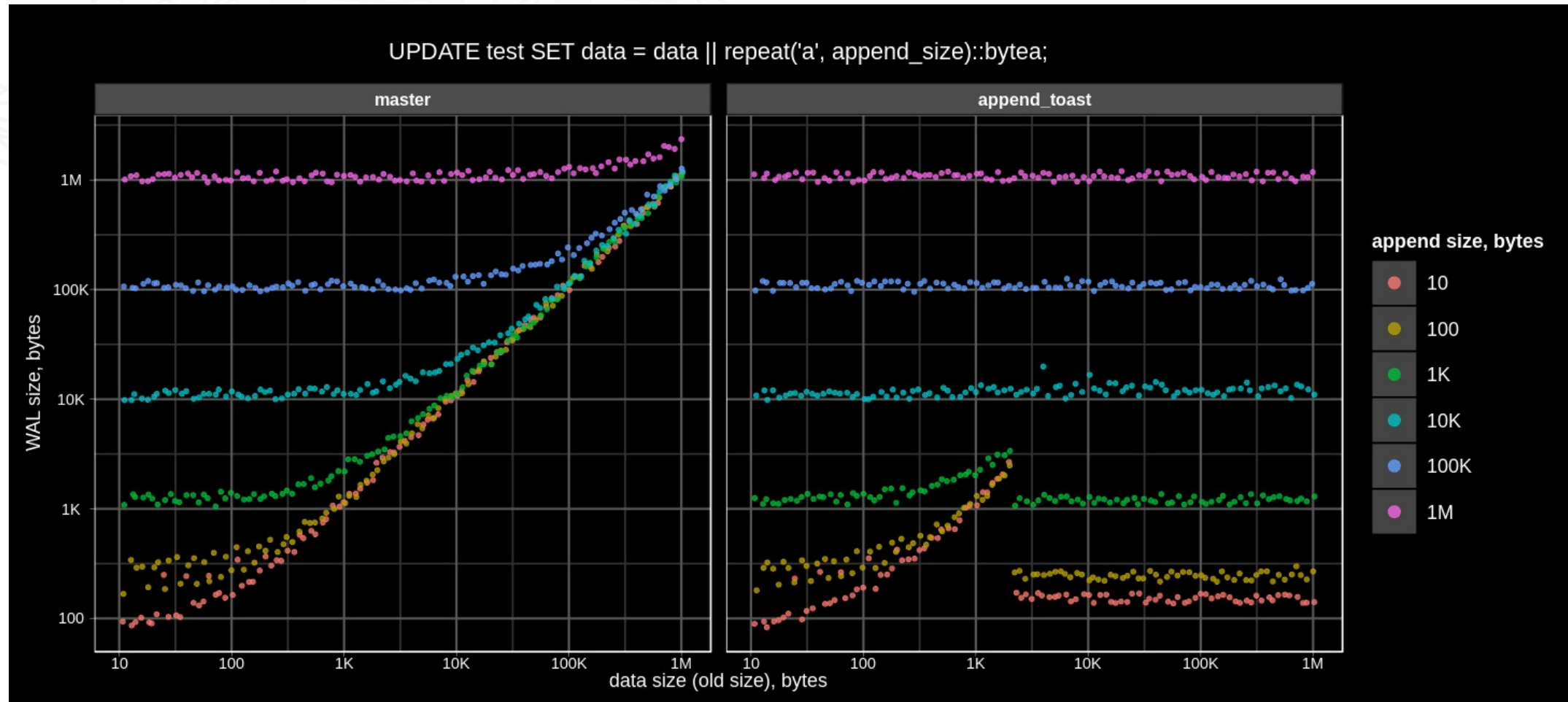


OLD + APPEND SIZE



APPEND SIZE

Results – WAL traffic



OLD + APPEND SIZE



INLINE OLD + APPEND SIZE

Appendable bytea: stream

Stream organized as follows:

- 1 row (id, bytea) grows from 0 up to 1Mb

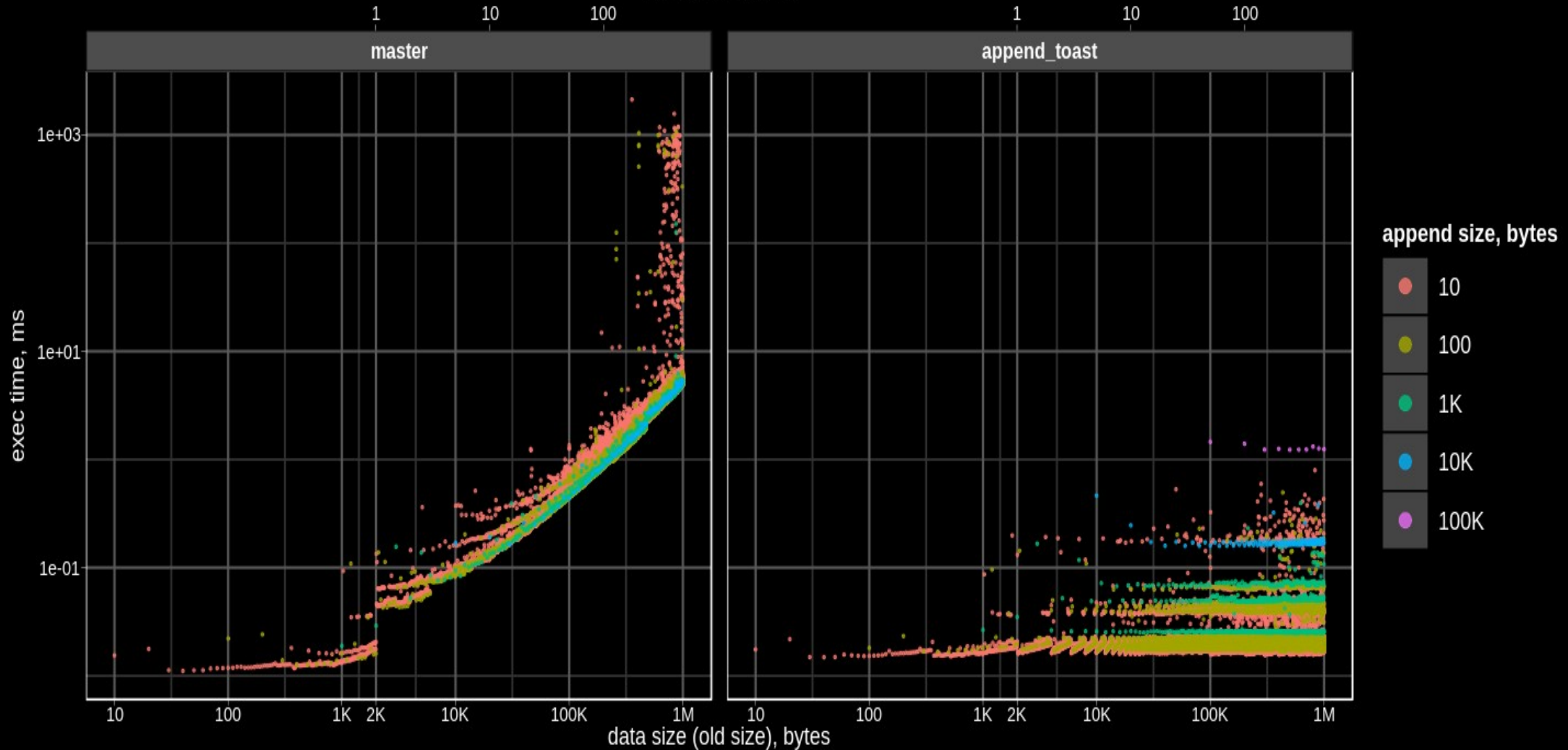
```
UPDATE test SET data = data || repeat('a', append_size)::bytea; COMMIT;
```

- append_size = 10b, 100b,...,100Kb
- pg_stat_statements: time, blocks r/rw, wal

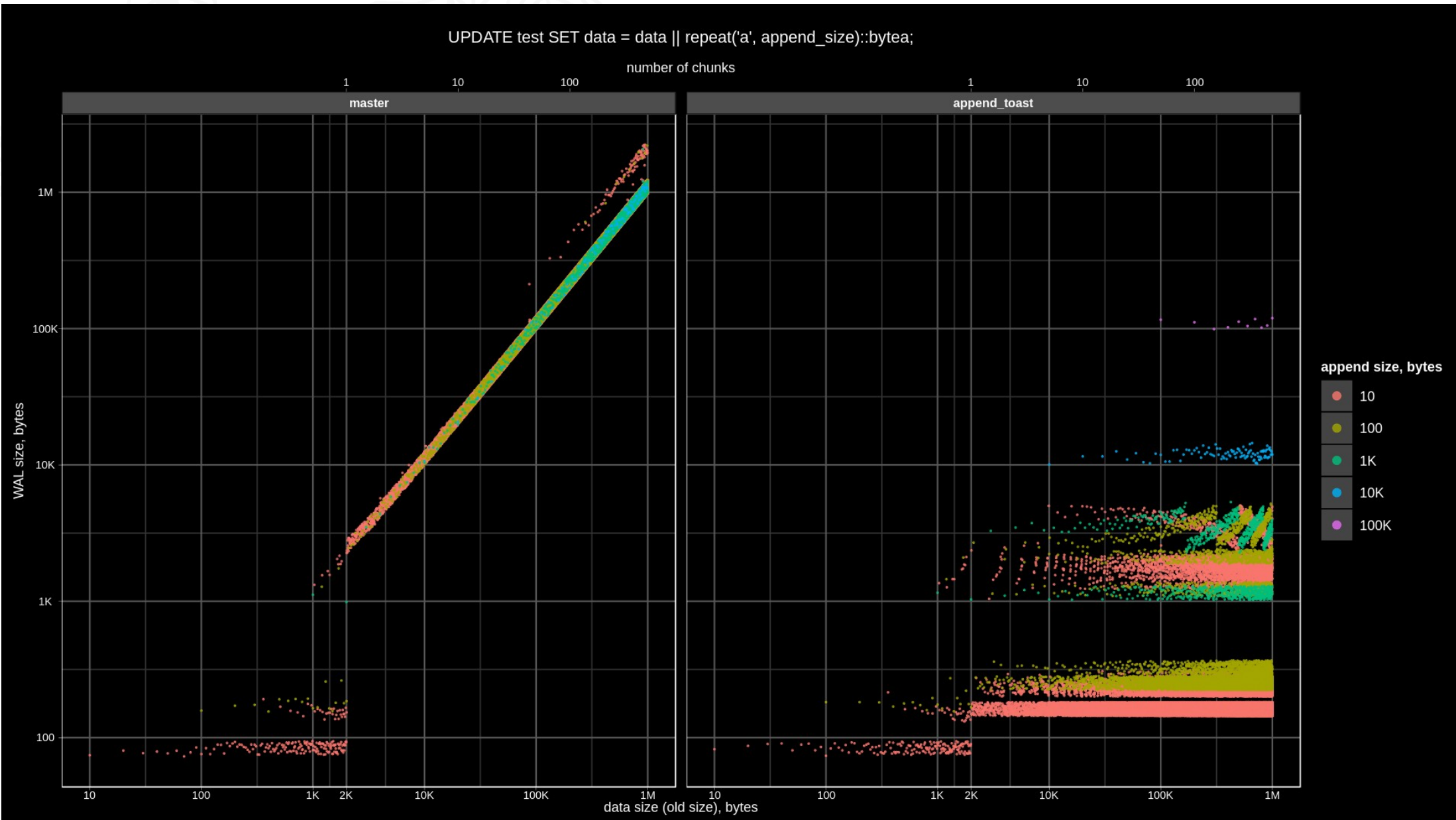
Appendable bytea: stream (time)

UPDATE test SET data = data || repeat('a', append_size)::bytea;

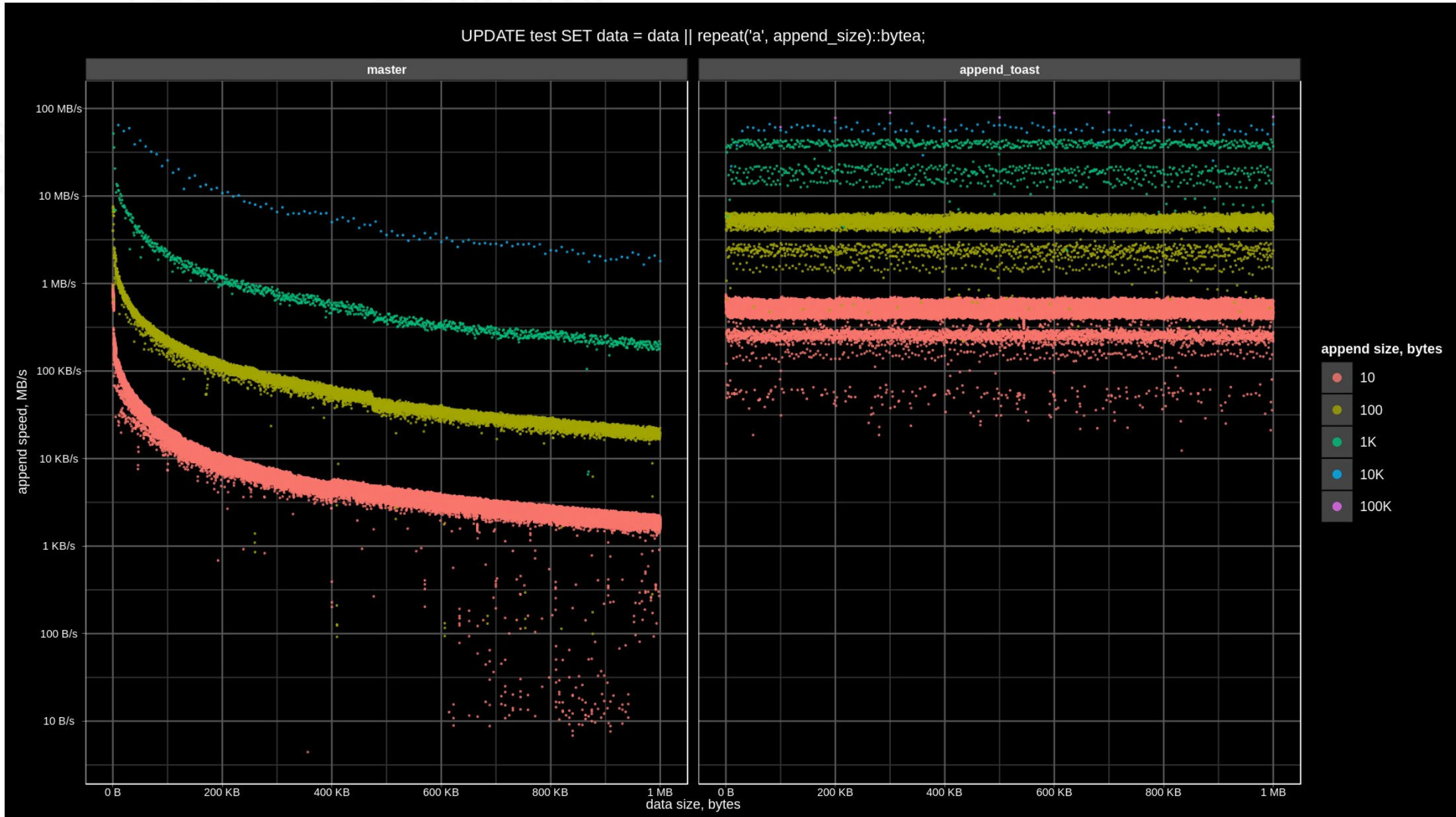
number of chunks



Appendable bytea: stream (WAL)



Appendable bytea: stream (throughput MB/s)




Conclusions

- TOAST in Postgres can be improved for specific data types: jsonb, appendable bytea
- How integrate them to the Postgres — that is the question !
- Data type aware TOAST — Pluggable TOAST

Extend Postgres Extensibility !

PLUGGABLE TOAST

PLUGGABLE

ONE TAST FITS ALL

Huge values — how to store?

- Page limit — 8KB by default (up to 64KB)
 - Obviously, not enough
- Large object
 - 2nd class citizen (no structure, isn't accessible at SQL-level, issues with backup)
- TOAST — slice to chain of small chunks
 - Could be compressed
 - Store in heap table with index (hidden)
 - TOAST doesn't know internals of data type, works with as just a long byte array.
 - Any update make a copy of whole value
 - TOAST is integrated in heap TAM
 - The single advantage - versatility

Extend Postgres Extensibility

- Let us extend Postgres even further!
- Create extension ...
- Toaster is «class» encapsulated all work with huge values
- Toaster could be specific for datatype (ex jsonb)
- Toaster could be specific for workload (ex append only)
- Toaster could be specific for column (different columns in table could use different toasters)
- Default toaster — compatibility with previous versions



Toaster - basis

CREATE EXTENSION name ;

Toaster - basis

```
CREATE TOASTER my_toaster HANDLER handler;
```

```
=# \d pg_toaster
```

```
Table "pg_catalog.pg_toaster"
```

Column	Type	Collation	Nullable	Default
oid	oid		not null	
tsrname	name		not null	
tsrhandler	regproc		not null	

Indexes:

```
"pg_toaster_oid_index" PRIMARY KEY, btree (oid)
```

```
"pg_toaster_name_index" UNIQUE CONSTRAINT, btree (tsrname)
```


Toaster - basis

```
CREATE TABLE tst1 (  
    foo text [STORAGE plain],  
    bar text STORAGE external TOASTER my_toaster,  
    id int4  
);  
  
ALTER TABLE tst1 ALTER COLUMN name SET TOASTER my_toaster;
```

```
=# \d+ tst1
```

Column	Type	Collation	Nullable	Default	Storage	Toaster	...
foo	text				plain	deftoast	...
bar	text				external	my_toaster	...
id	integer				plain		...

```
Access method: heap
```

Toaster - basis

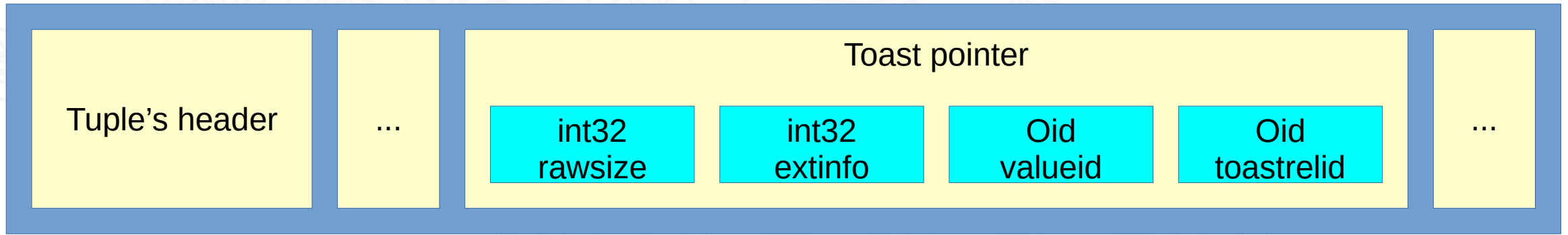
```
=# \d pg_attribute
```

```
Table "pg_catalog.pg_attribute"
```

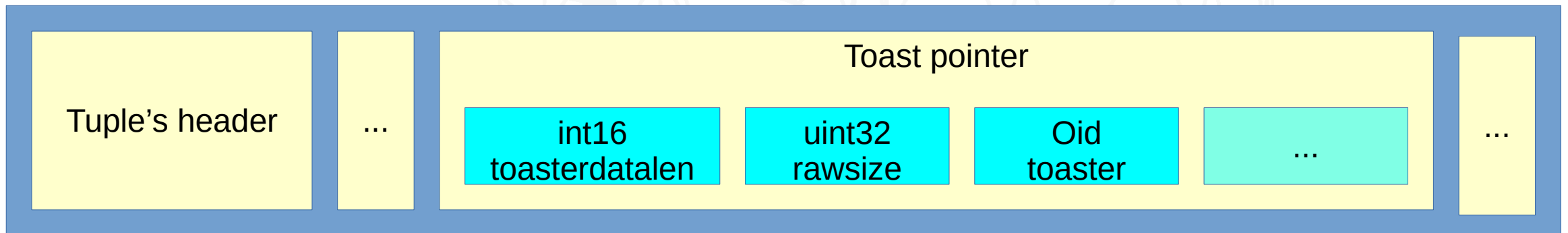
Column	Type	Collation	Nullable	Default
attrelid	oid		not null	
attname	name		not null	
atttypid	oid		not null	
...				
attstorage	"char"		not null	
atttoaster	oid		not null	
attcompression	"char"		not null	
....				

Toaster storage in tuple

Default (current) toaster (16 bytes, alignment 4)



Custom toaster (≥ 10 bytes, alignment 2)



Consequences

Toast pointer is aware of toaster id it was toasted by

- toaster could not be dropped (is it really necessary?)
- In one column could be data toasted by different toasters

Toaster - API

```
typedef struct TsrRoutine {  
    NodeTag      type;  
  
    /* interface functions */  
    toast_init init;  
    toast_function toast;  
    update_toast_function update_toast;  
    copy_toast_function copy_toast;  
    detoast_function detoast;  
    del_toast_function deltoast;  
    get_vtable_function get_vtable;  //?  
    toastervalidate_function toastervalidate;  //?  
}
```

Toaster — validate?

```
/* validate definition of a toaster Oid */  
typedef bool (*toastervalidate_function)  
    (Oid typeoid, char storage, char compression,  
     Oid amoid, bool false_ok);
```

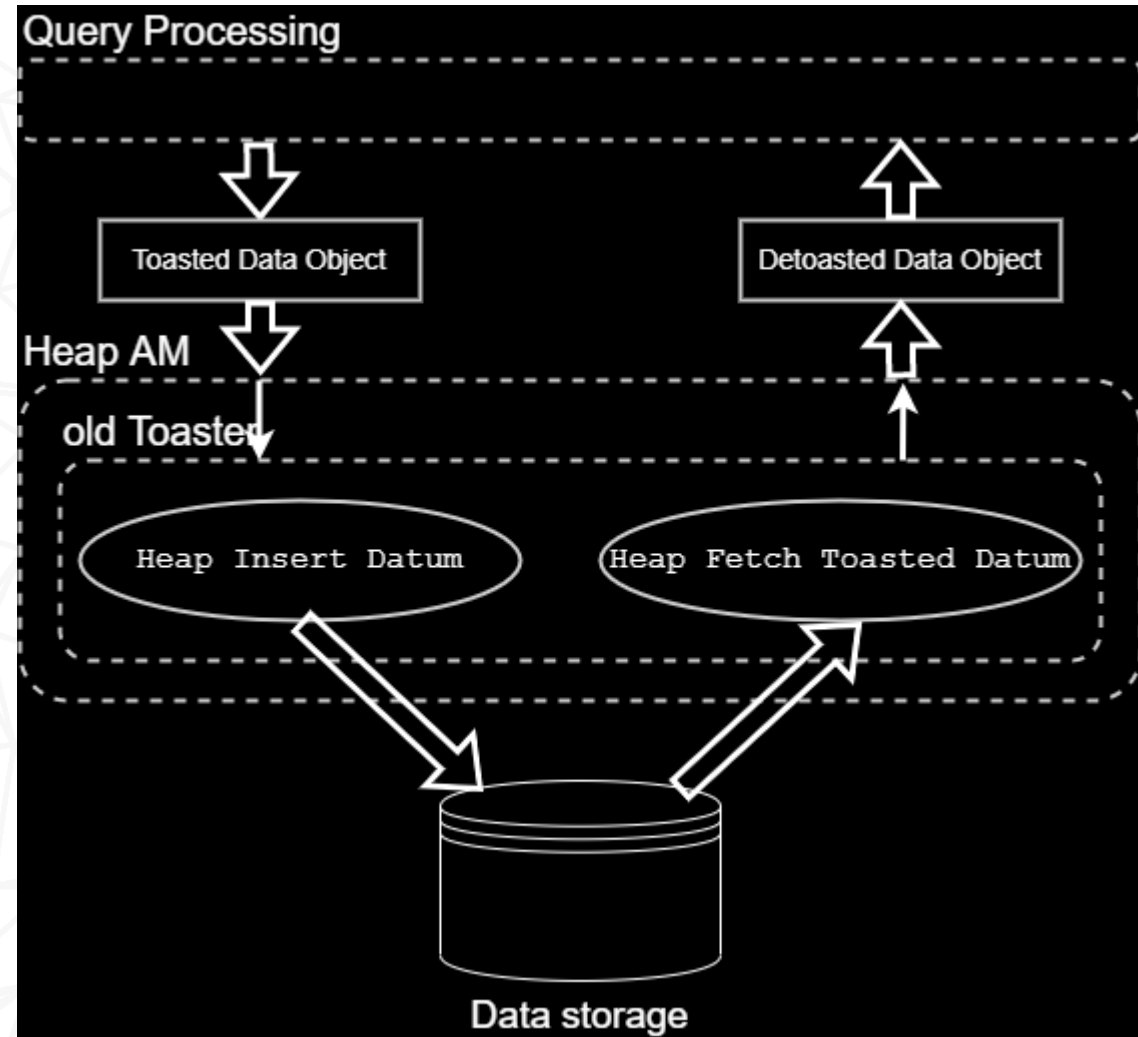
- Validate method is required for Toaster compatibility check.
- Toaster is specific for datatype and workload, it depends on compression and storage.

Toaster — vtable?

```
/* Return virtual table of functions, optional */  
typedef void * (*get_vtable_function)  
                (Datum toast_ptr);
```

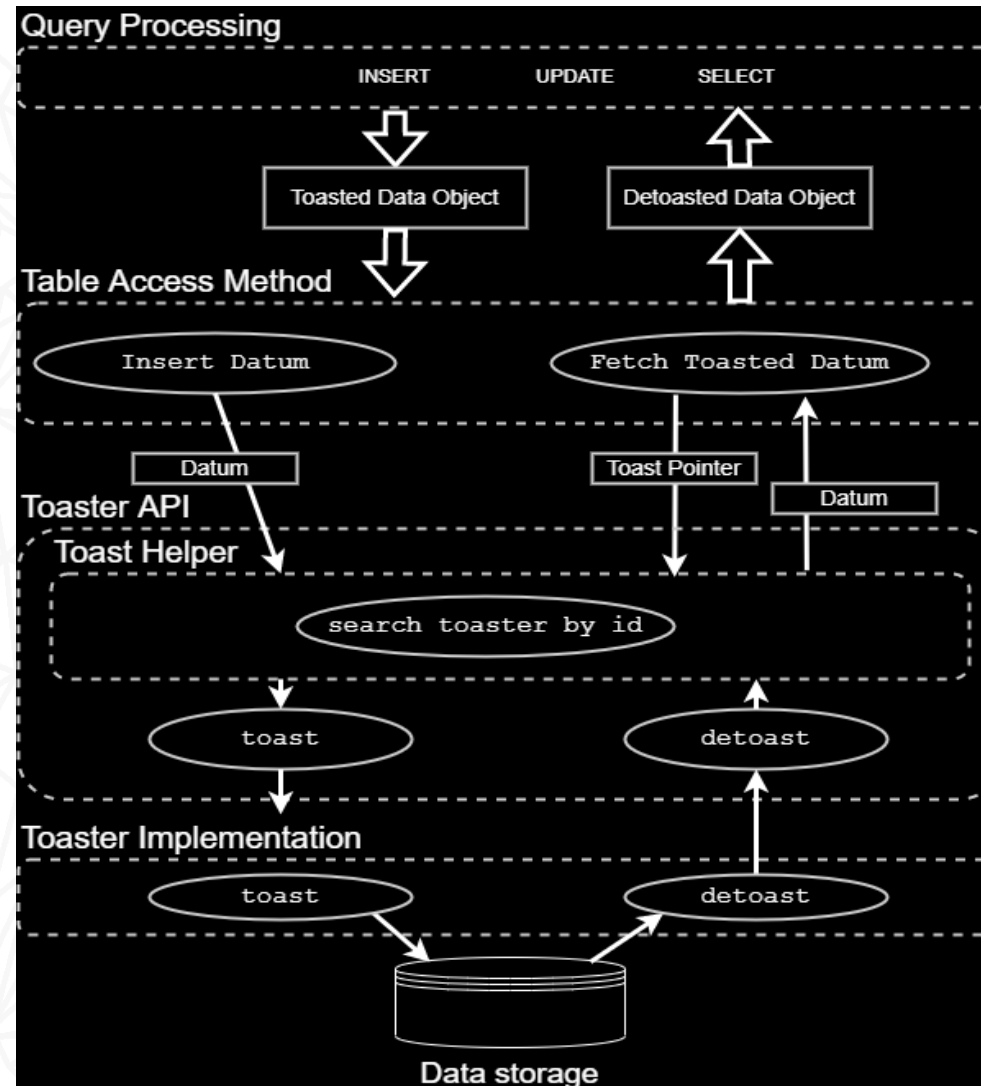
Current Toast/Detoast

- Part of the Heap AM
- Single Toast/Detoast strategy - full Toast/Detoast only
- Not extensible

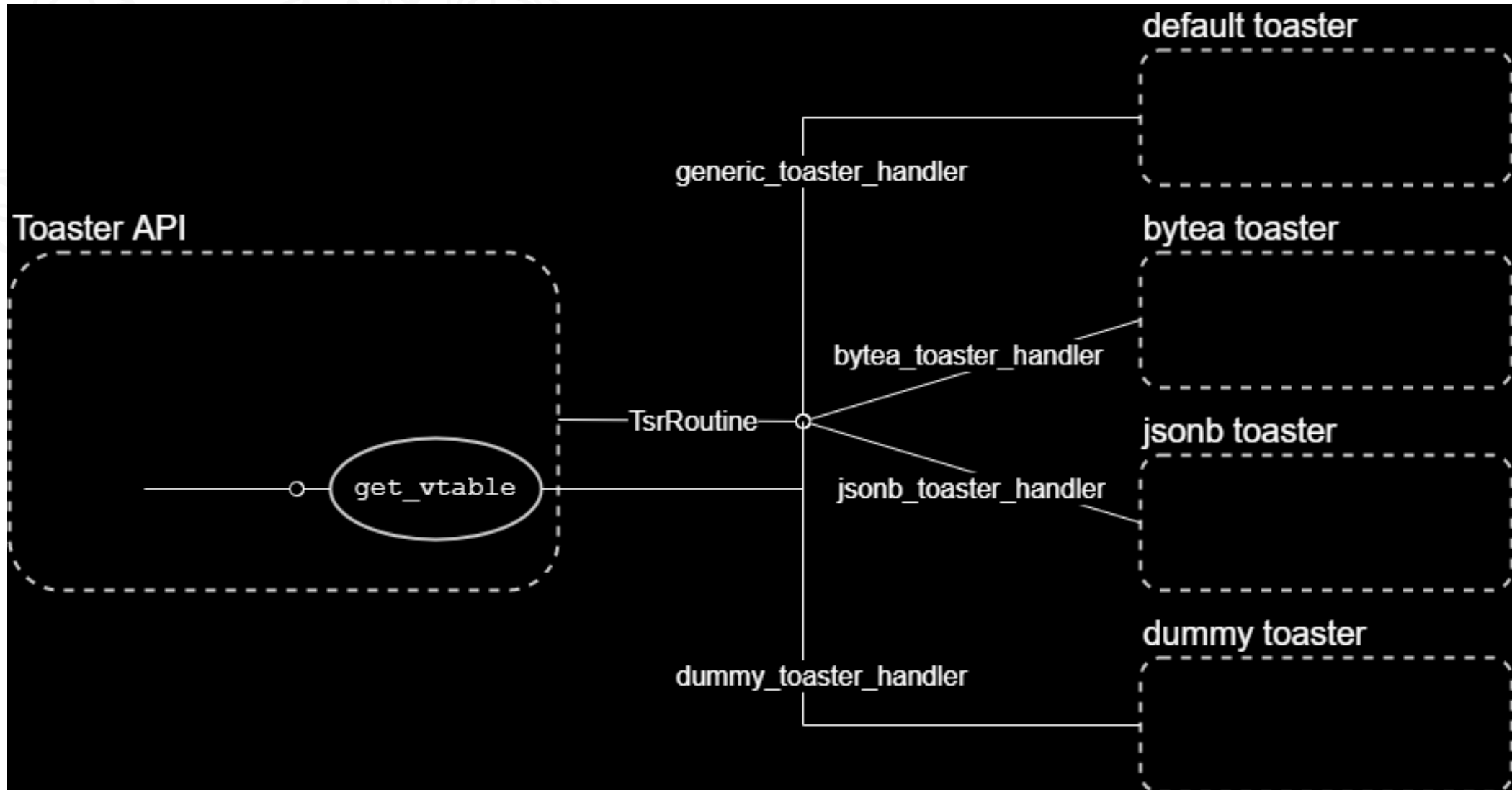


Toaster API Magic

- Detached from Heap AM, Independent
- Possibility extend with any Custom Toaster
- Does not affect performance



Toaster Handlers



Virtual Table of User-Defined Functions

vtable is an inner API of Toaster API. It allows Custom Toasters to have any user-defined function a developer wants – just put it into TsrRoutine virtual function table, and it is ready to use!

For example, bytea Toaster has append() function to append two bytea Datums instead of creating new (third) copy. These functions may not be directly used for toasting/detoasting, but could provide additional operations

```
static void *  
bytea_toaster_vtable(Datum toast_ptr)  
{  
    ByteaToastRoutine *routine = palloc0(sizeof(*routine));  
    routine->magic = BYTEA_TOASTER_MAGIC;  
    routine->append = bytea_toaster_append;  
    return routine;  
}
```

BYTEA and JSONB Toasters

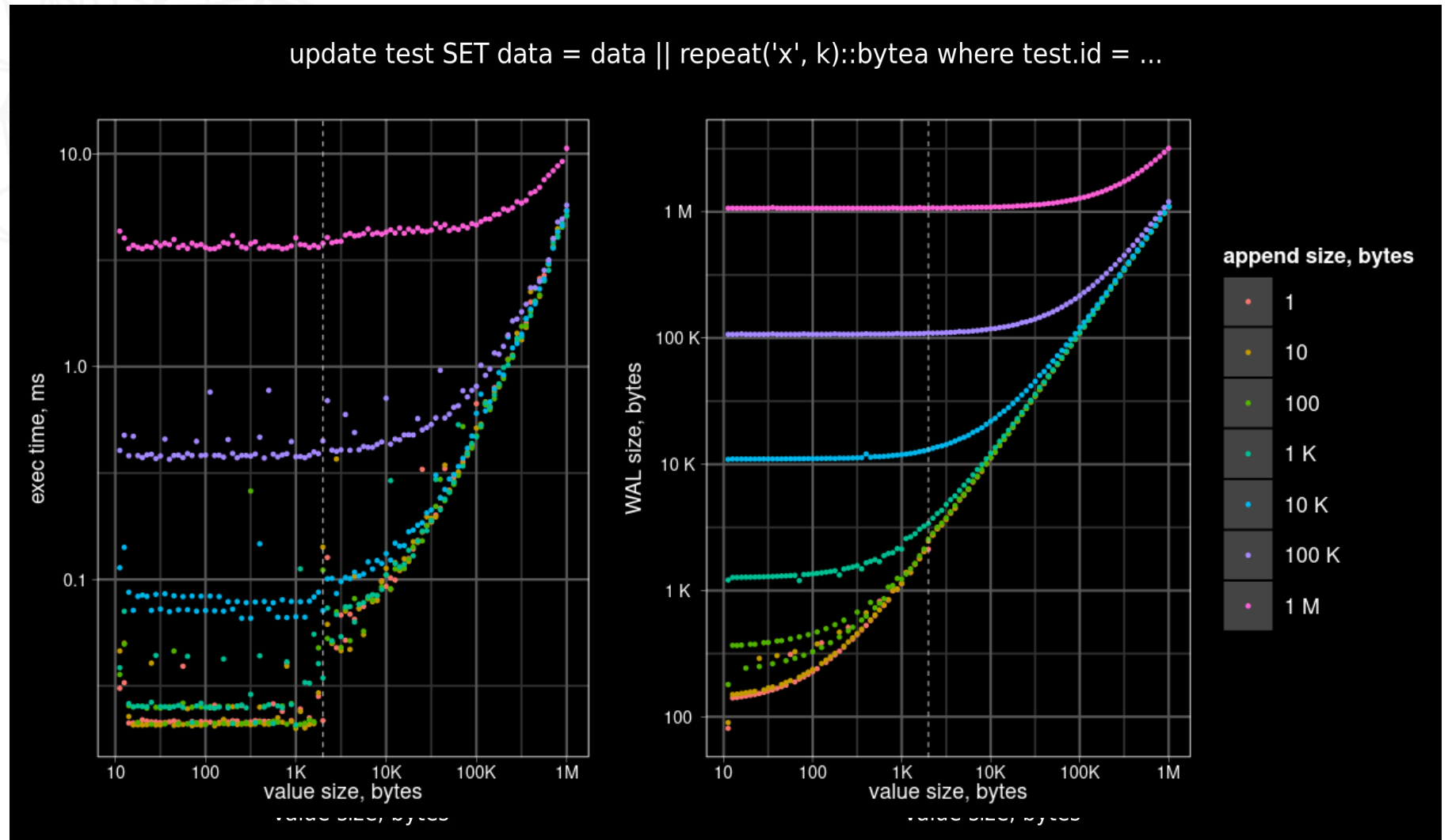
- We are glad to present two Toasters developed by PostgresPro Team – bytea Appendable Toaster and Jsonb Toaster.

Both types could store huge amounts of data, and current Toast mechanics does not perform well with accessing and updating:

- bytea type is suitable for streaming, which require special fast update mode – “append” without re-writing full data record;
- Values stored in JSONb objects are often accessed “by-key”, and full detoast to fetch just one key is very ineffective

Bytea Update Performance

- Update time depends on the size of the updated and appended values
- Huge WAL traffic is generated – full record is placed in WAL with update



The Solution is – bytea Appendable Toaster!

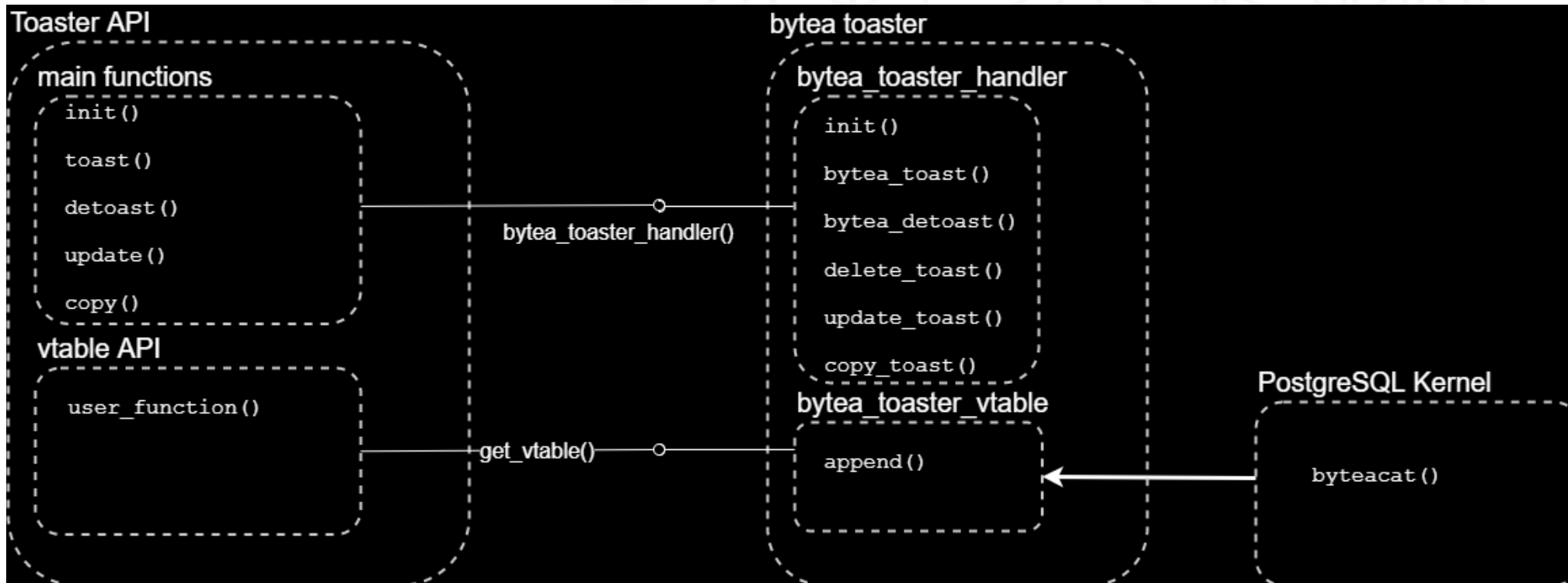
- Toaster package specially designed for bytea datatype.
- Fast, effective, extensible.

The main difference, along with the toast/detoast and update functions, is Custom ByteaAppendablePointer

- Modified tail is stored as inline data;
- When inline tail exceeds Toast size limit, it is toasted, but unmodified chunks are left as-is

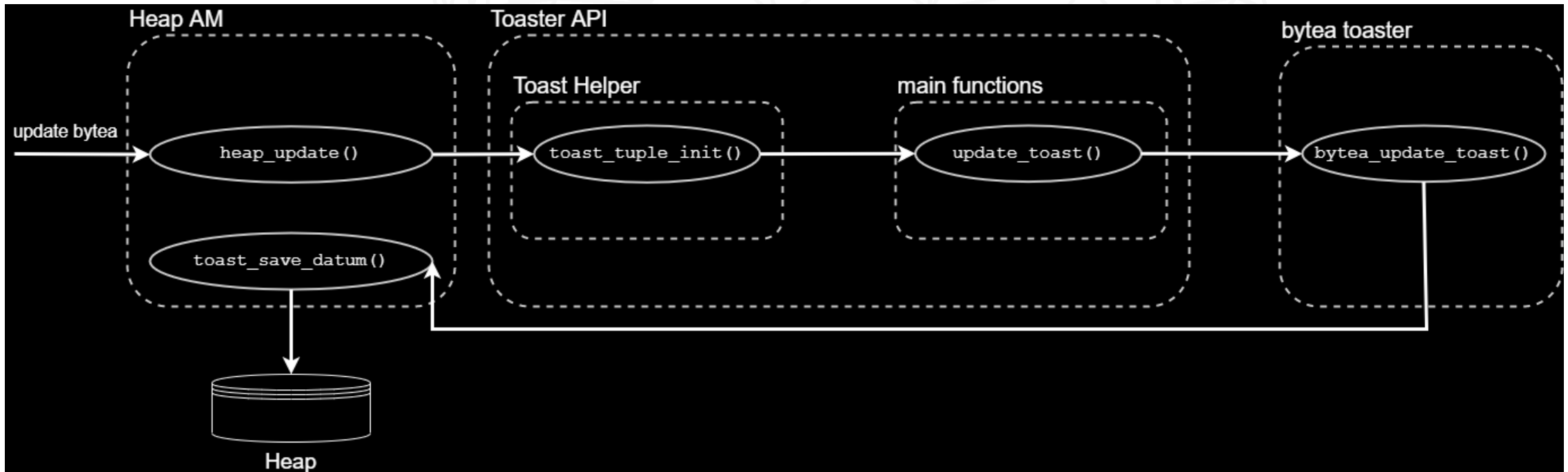
Bytea Toaster Extension

- Special datum format: TOAST pointer + inline data
- “append” operation - operator || does not deTOAST data, it appends inline data producing datum in new format
- TOASTER recognizes changes and TOASTs only the new inline data, possibly rewriting last chunk in chain

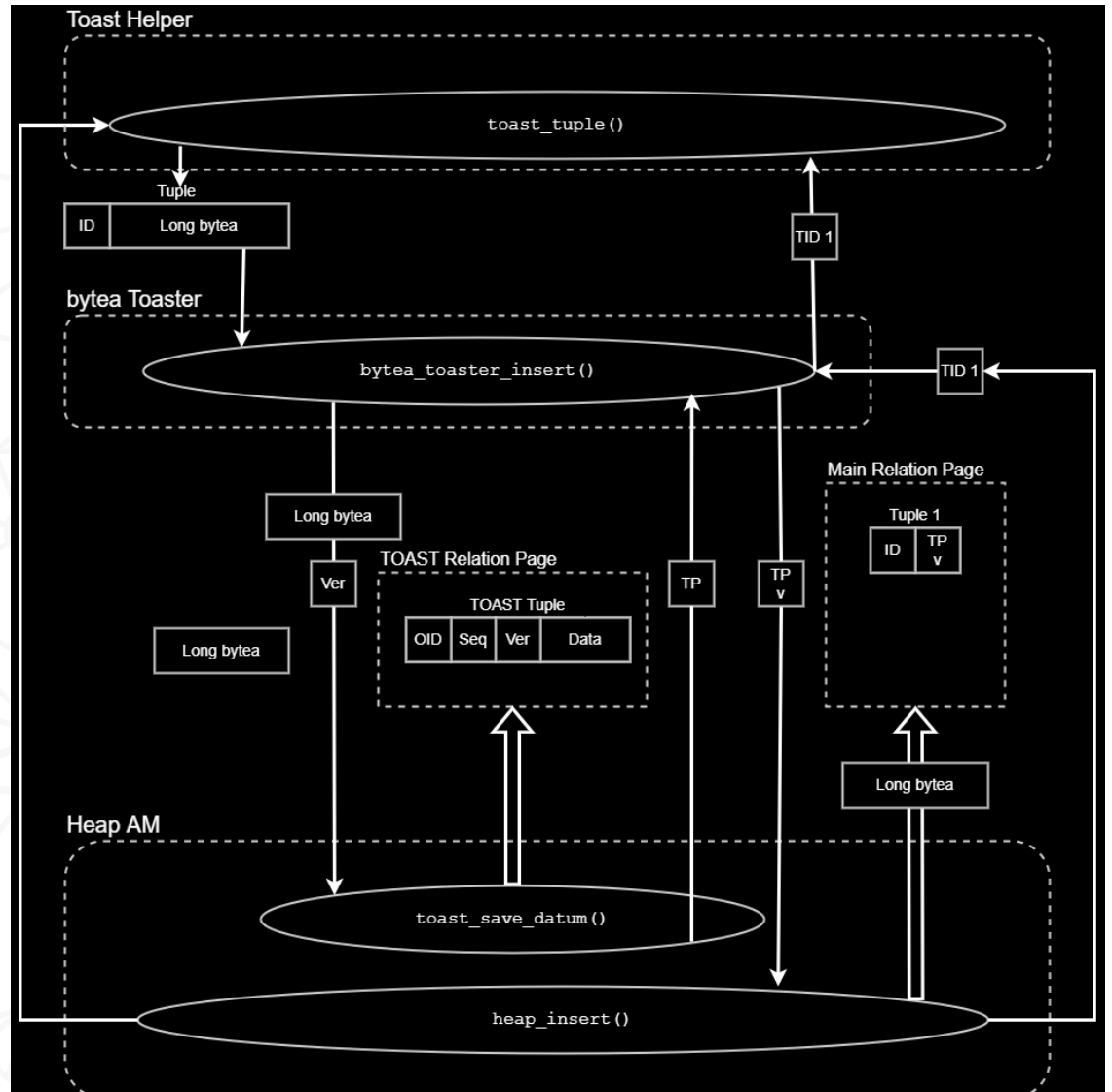
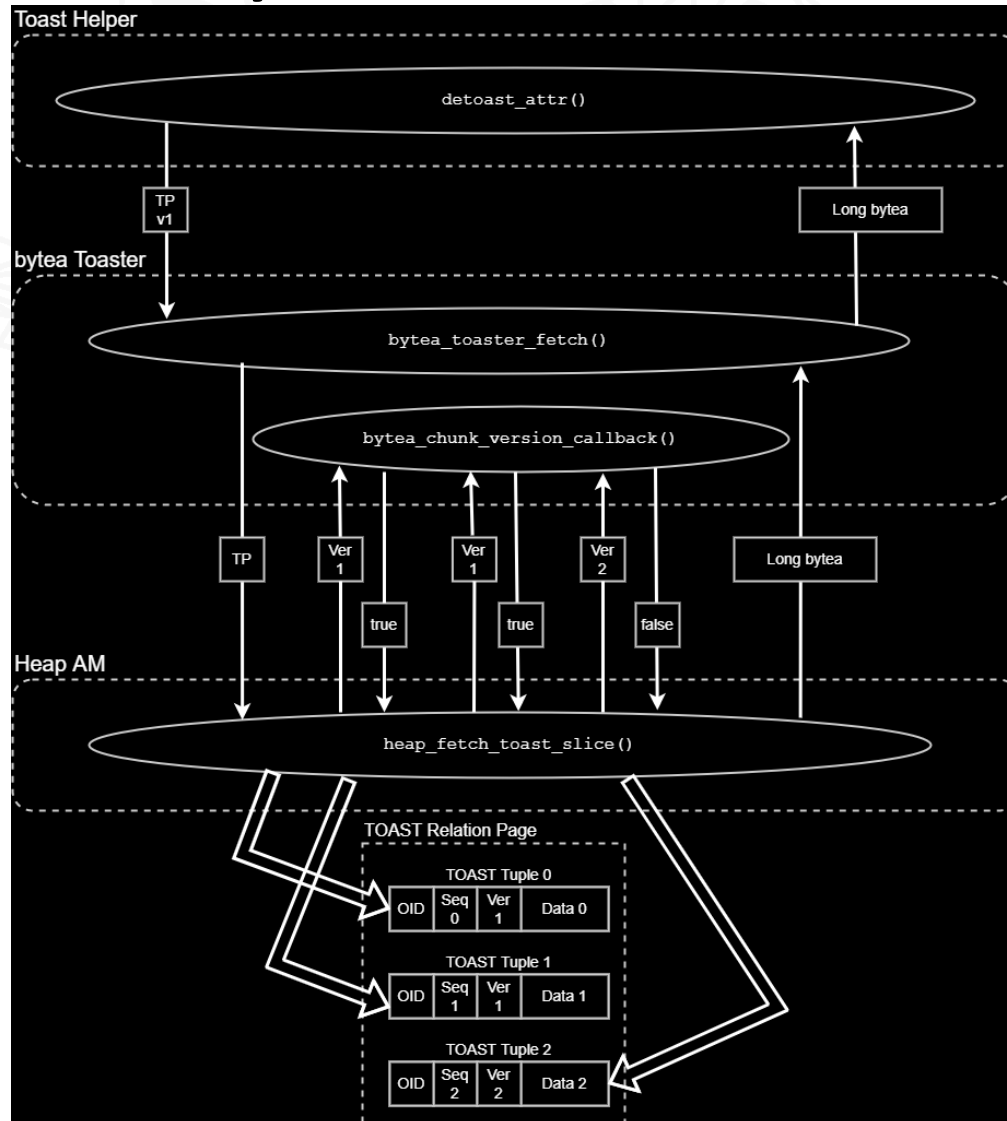


Toasting Appended Data

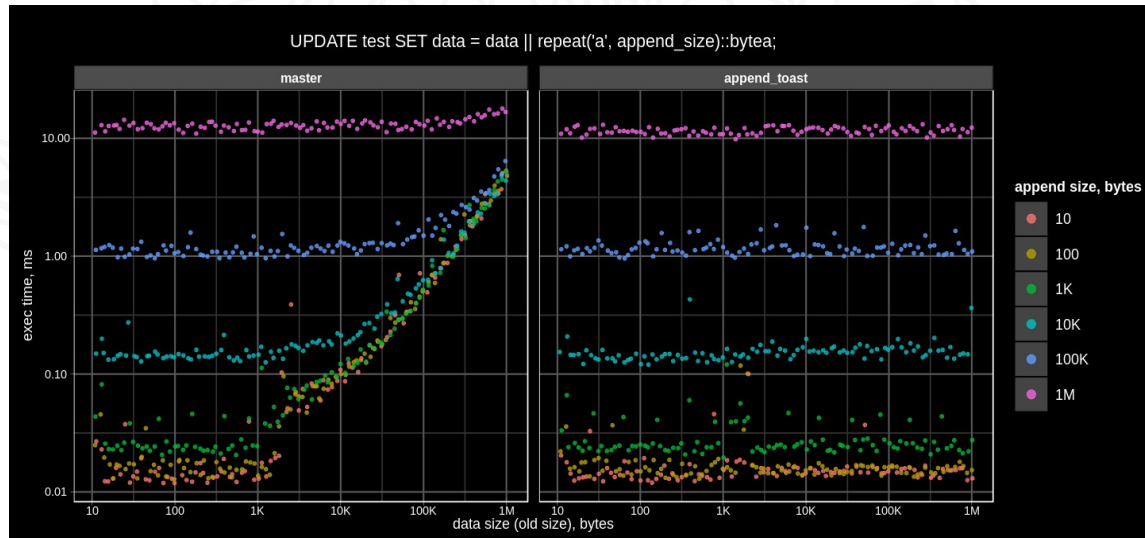
- Toaster API Bytea Toaster is called via Toast
- Last not filled chunk can be rewritten with creation of new tuple version
- First unmodified chunks are shared



Bytea Fetch and Insert



Bytea Toaster Performance

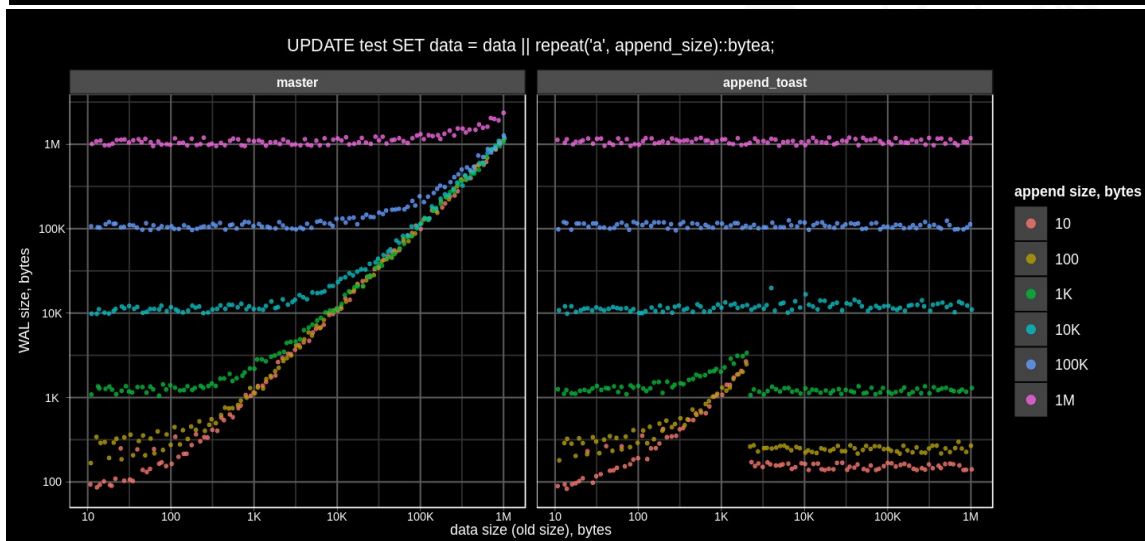


MASTER

- $T \sim \text{OLD SIZE} + \text{APPEND SIZE}$

BYTEA TOASTER

- $T \sim \text{APPEND SIZE}$



MASTER

- $\text{WAL} \sim \text{OLD SIZE} + \text{APPEND SIZE}$

BYTEA TOASTER

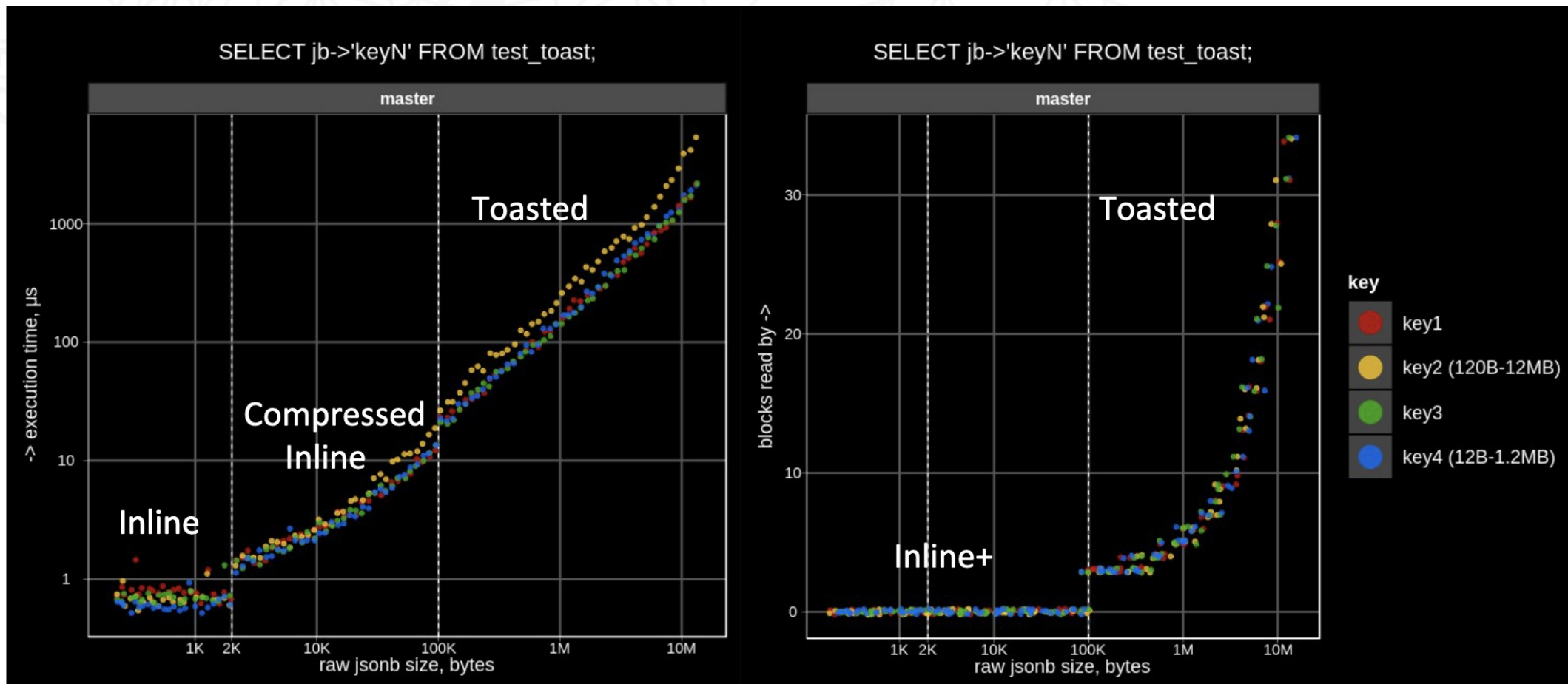
- $\text{WAL} \sim \text{INLINE} + \text{APPEND SIZE}$

JSONb

- Json is very popular datatype used by many applications;
- People want SQL/JSON and want it to be fast and effective;
- Json objects are mostly accessed by keys;
- JSONb is a PostgreSQL internal binary representation of Json objects;
- Full JSONb object needs to be detoasted to access single key-value pair – very ineffective;

Default JSONb Access Performance

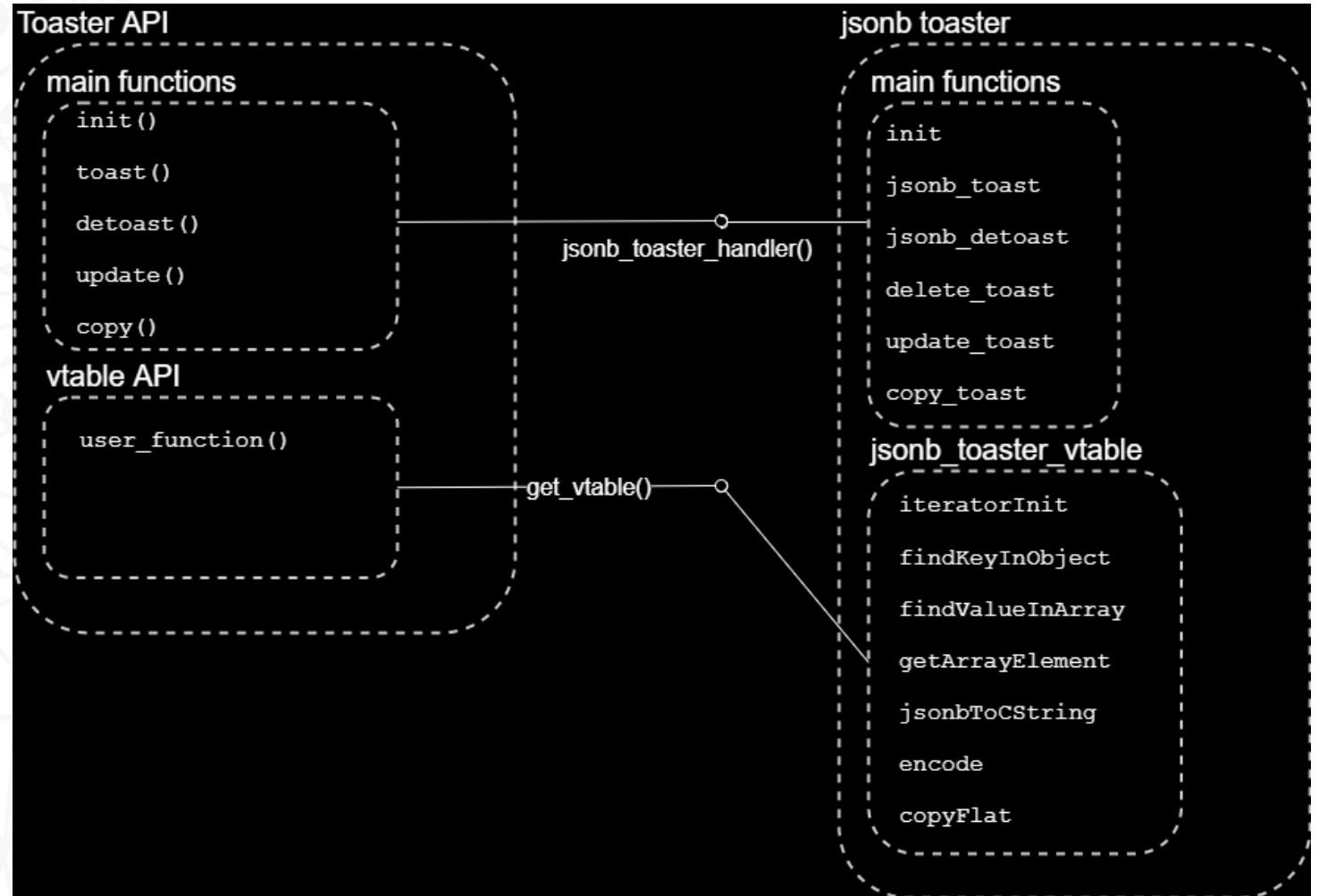
Key access time linearly increases with jsonb size, regardless of value size and position



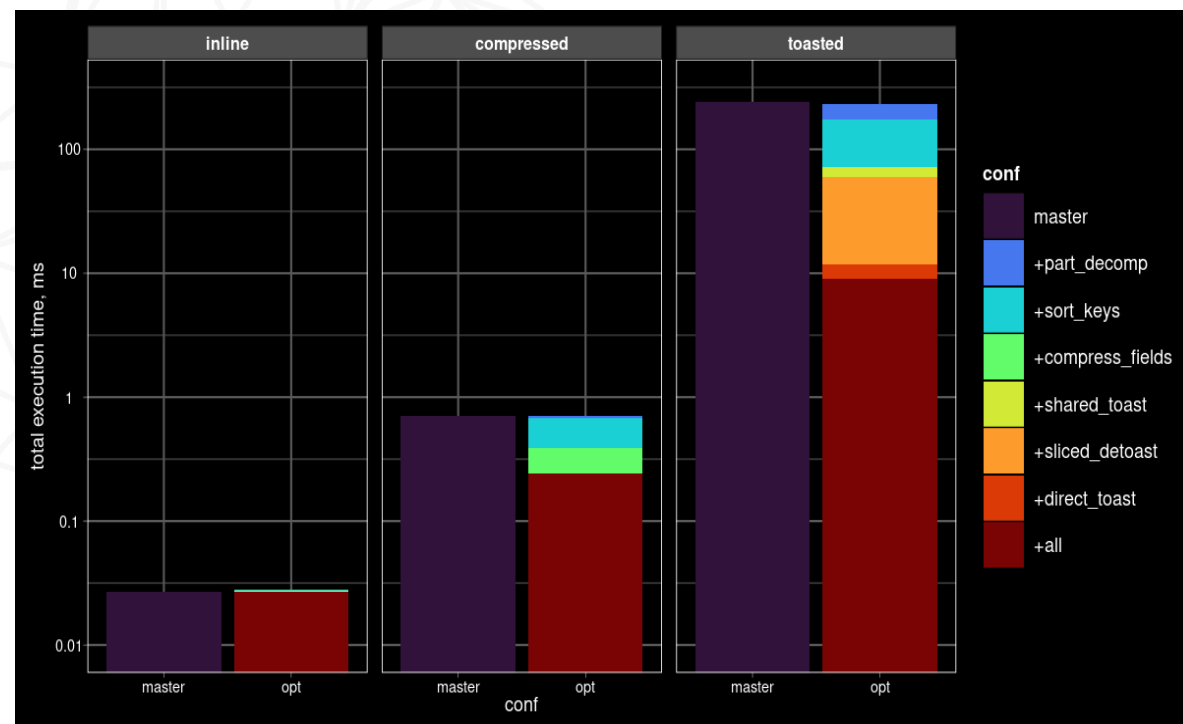
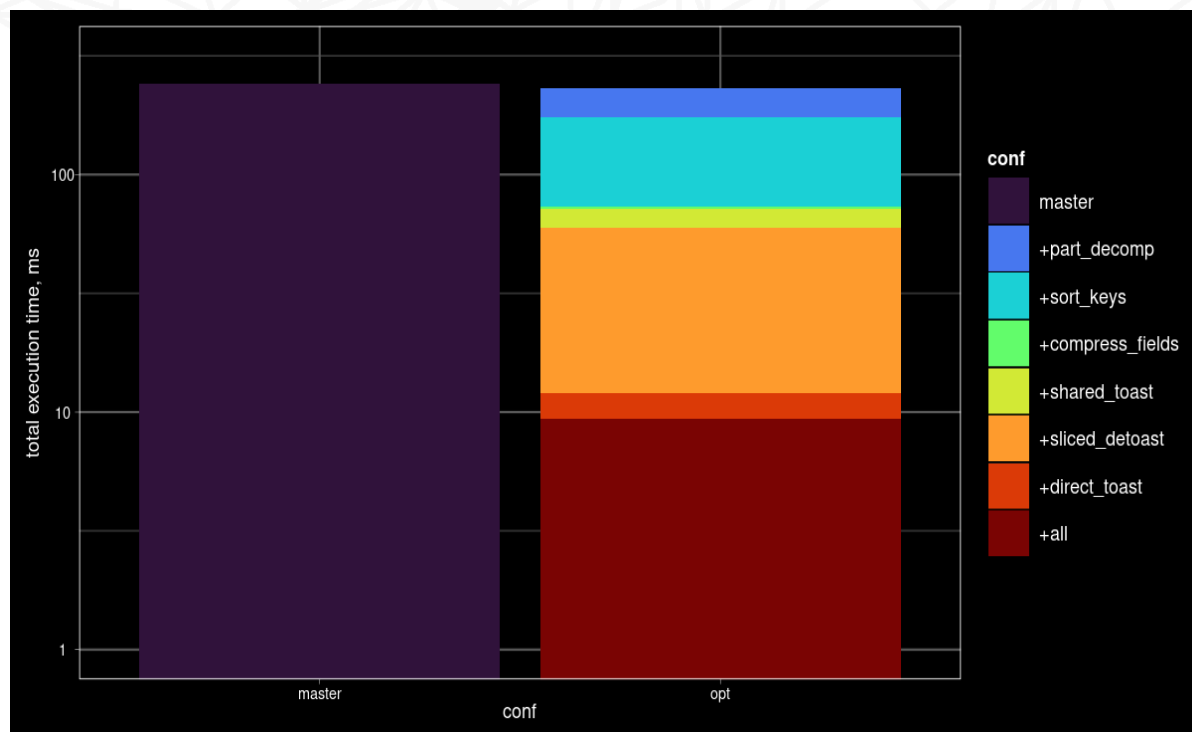
Jsonb Toaster Extension

Jsonb Toaster is plugged in with Toaster API.

Along with Toast/Detoast
Jsonb Toaster provides new
functions to work with
Jsonb Containers and
Iterators



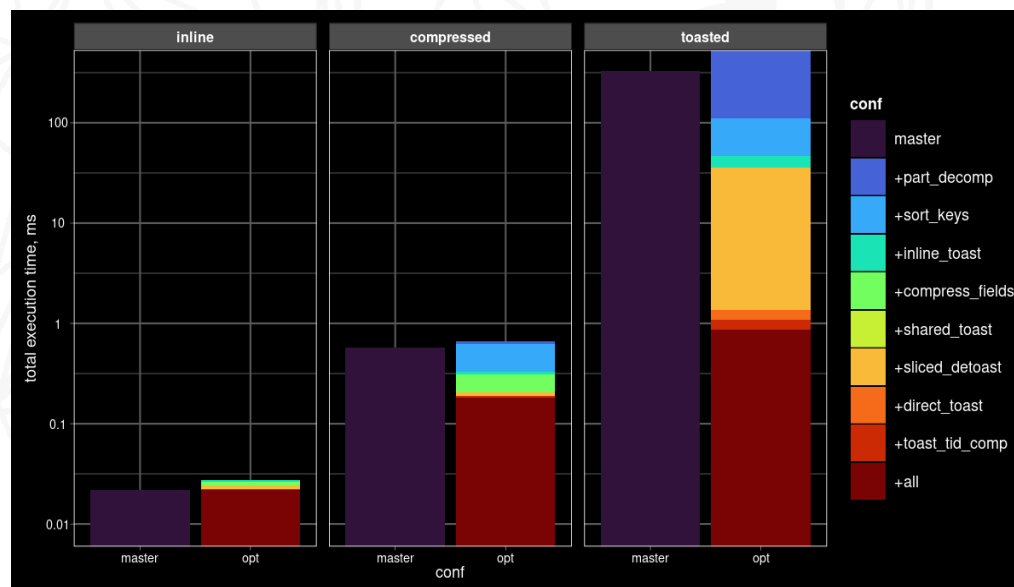
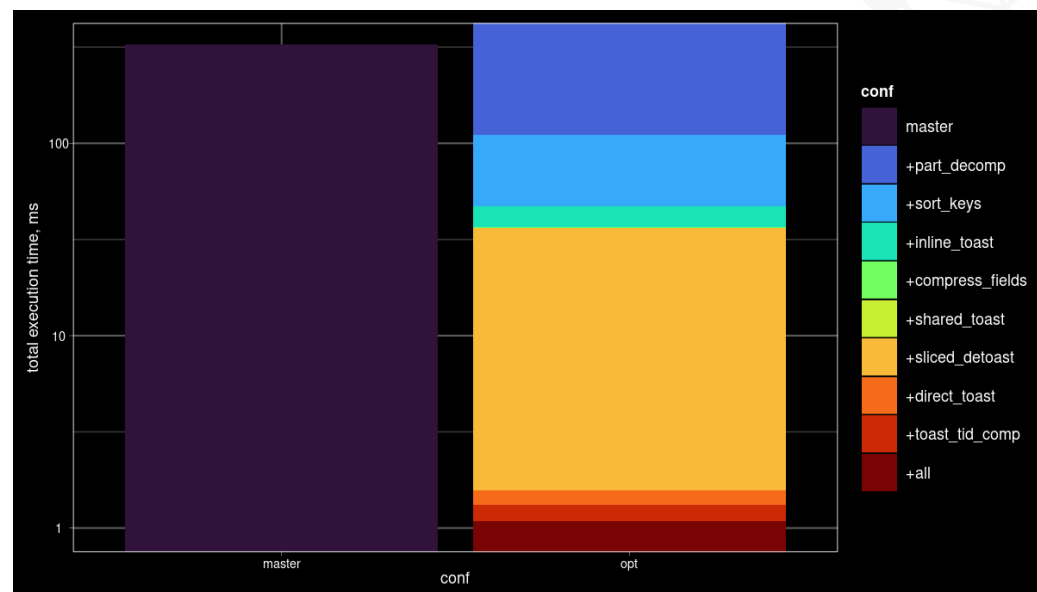
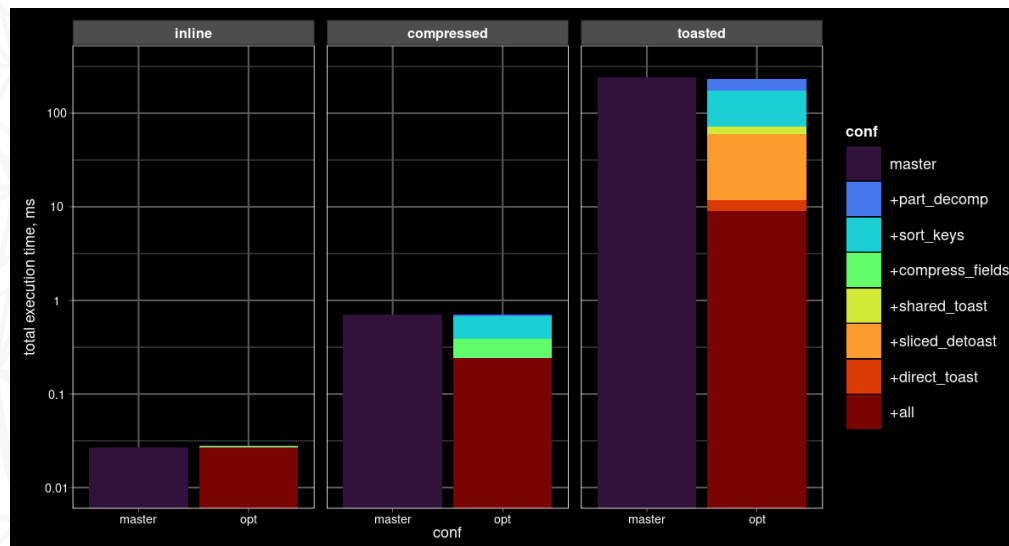
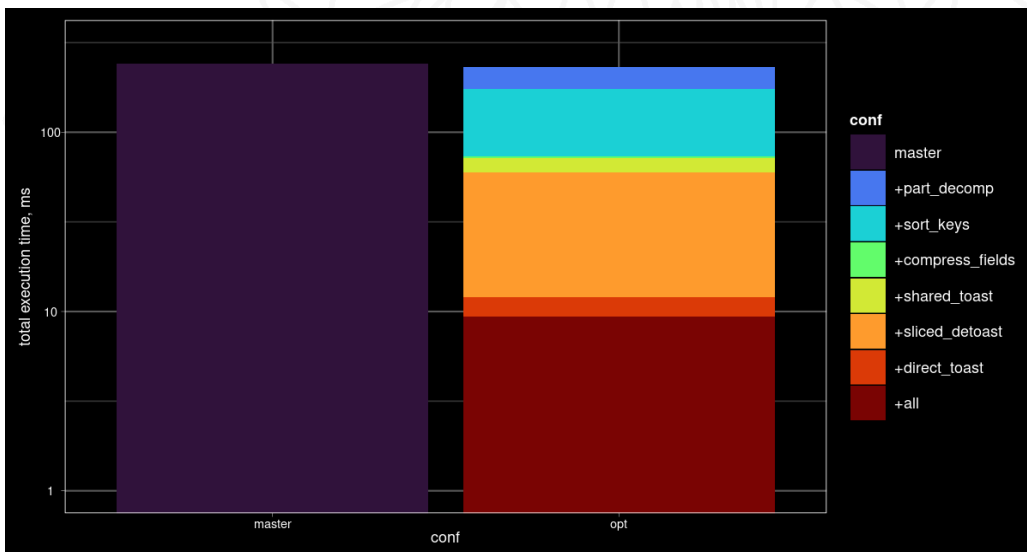
Jsonb Toaster aggregated statistics



Jsonb Toaster needs more work !

Jsonb Toaster

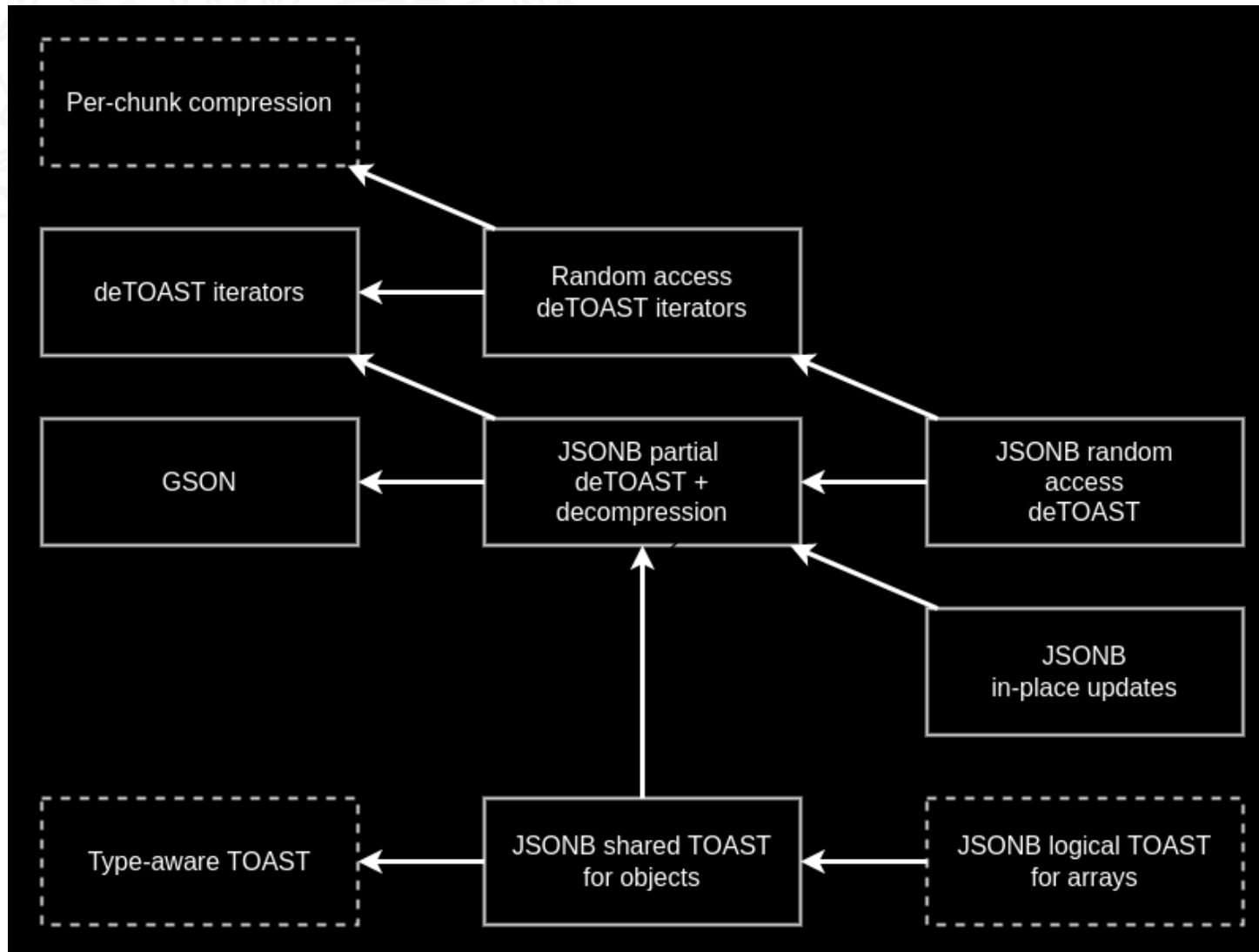
Patched Toaster



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)
- Additional Access Methods

Roadmap and patch set



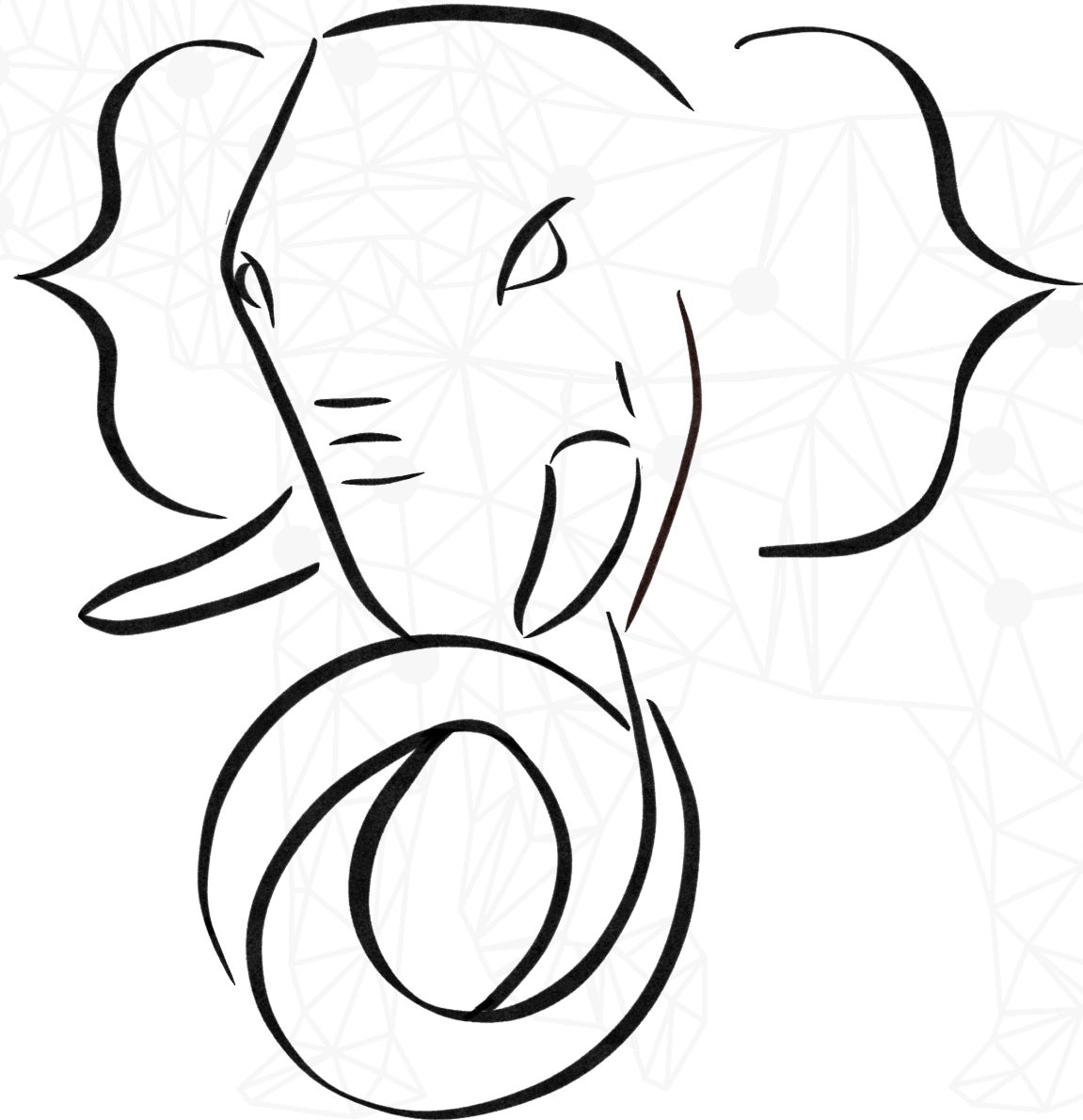
References

- Our experiments:
 - Understanding Jsonb performance
<http://www.sai.msu.ru/~megera/postgres/talks/jsonb-pgconfnyc-2021.pdf>
 - Details - <http://www.sai.msu.ru/~megera/postgres/talks/jsonb-pgvision-2021.pdf>
 - Slides of this talk
<http://www.sai.msu.ru/~megera/postgres/talks/toast-highload-2022.pdf>
 - Борьба с TOAST или будущее JSONB в PostgreSQL
<https://habr.com/ru/company/oleg-bunin/blog/646987/>
 - Pluggable TOAST at Commitfest
<https://commitfest.postgresql.org/38/3490/>
- 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

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



Let's not say that json is the wrong technology,
Let's make json a first class citizen instead.



ALL

YOU

NEED
POSTGRES

IS

