

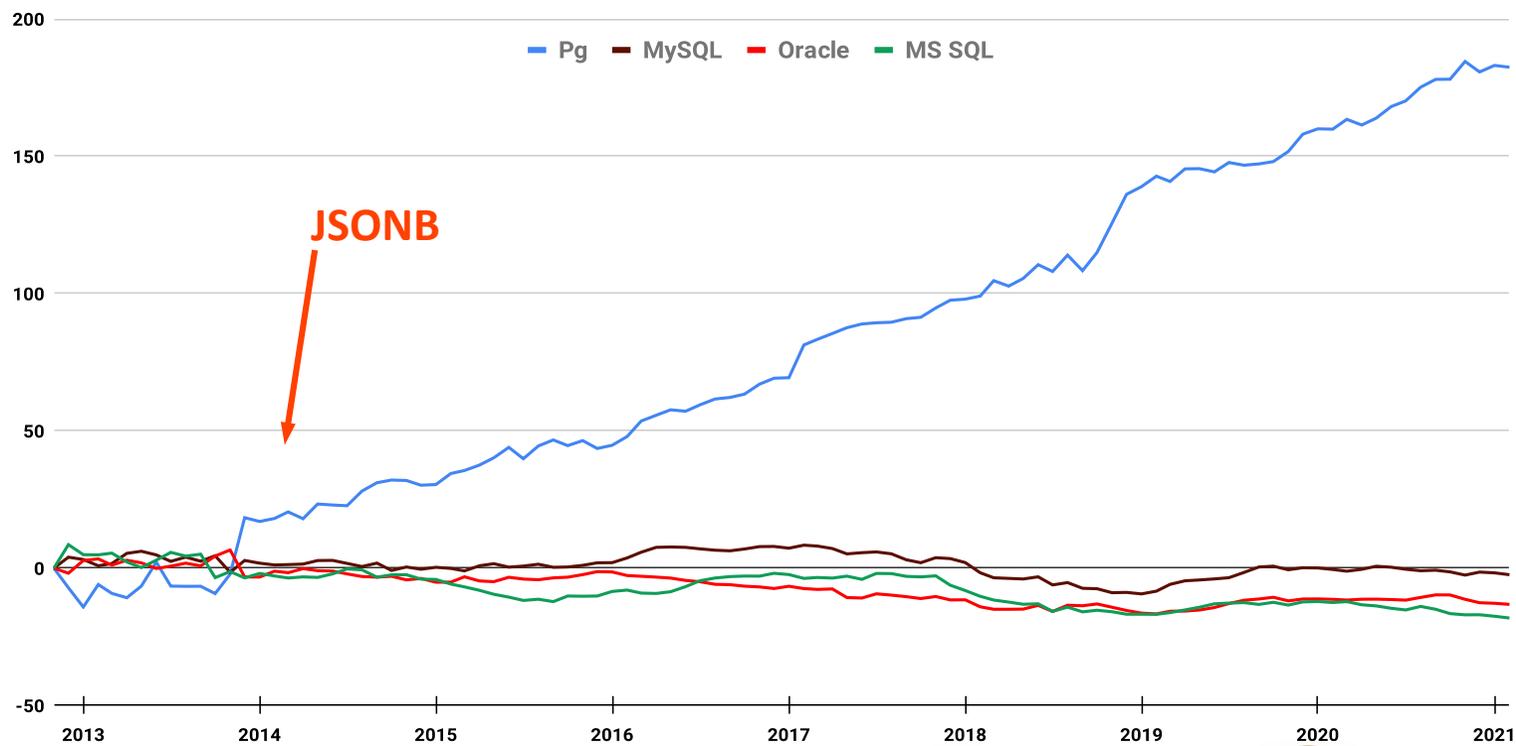
NEW TOAST IN TOWN

FOUR  
ONE TOAST FITS ALL

# Postgres breathed a second life into relational databases

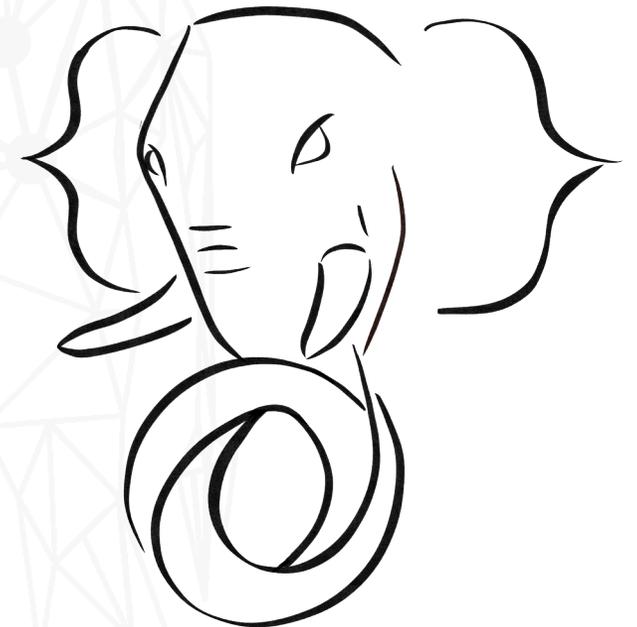
- Postgres innovation - the first relational database with NoSQL support
- NoSQL Postgres attracts the NoSQL users
- JSON became a part of SQL Standard 2016

Relative Growth



**PG15: SQL/JSON/TABLE** (#postgrespro)

Thanks, Andrew Dunstan for committing !



# Why this talk ?

- Startups want/need JSON[B]
- Blossom of Microservice architecture
- One-Type-Fits-All (JSON)
  - Client app — Frontend - Backend — Database
- JSONB is one of the main driver of Postgres popularity
- Results of our experiments in 2021:



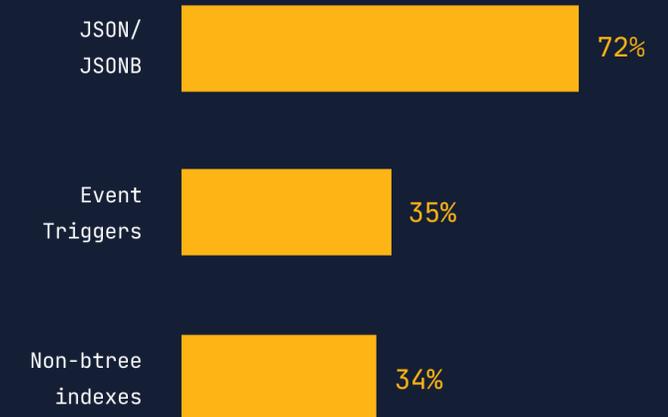
*Performance of JSONB ( not only ) can be improved by several orders of magnitude with proper modification of TOAST.*

- **Pluggable TOAST** - legal way to integrate our improvements into the 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](#)



# 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 !!!!

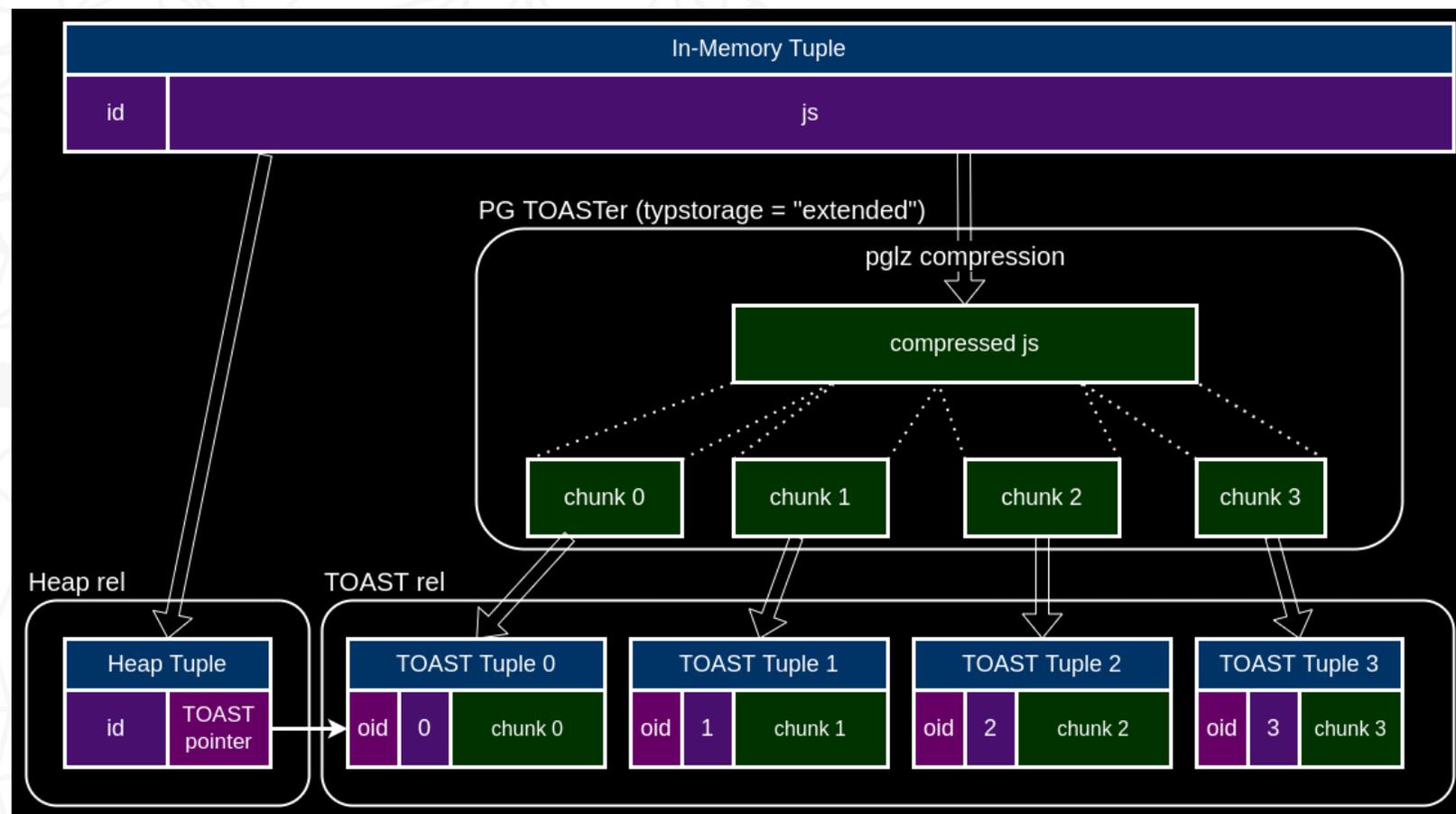
# TOAST Explained

The Oversized-Attribute Storage Technique

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

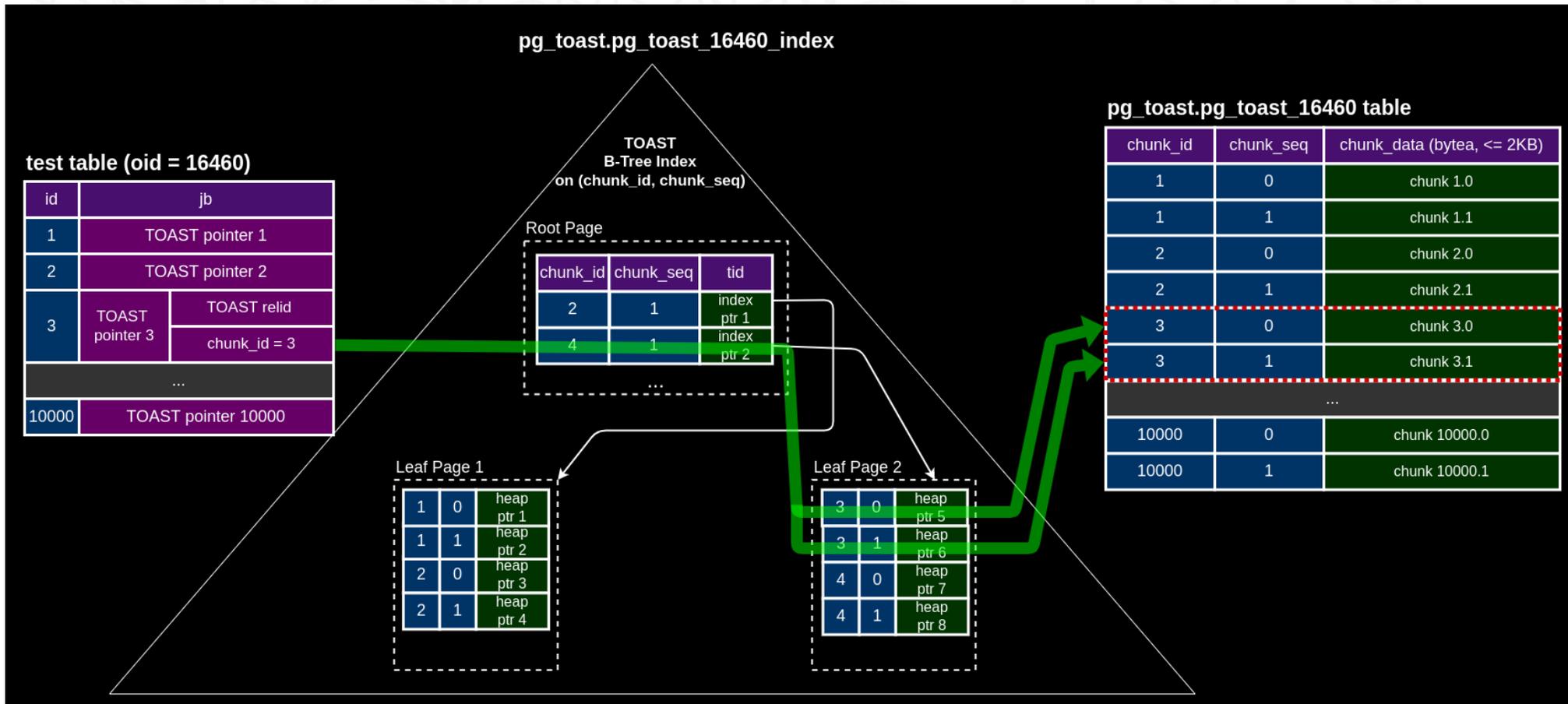
- TOAST chunks (along with generated Oid chunk\_id and sequence number chunk\_seq) stored in special TOAST relation `pg_toast.pg_toast_relid`, created for each table with TOASTed attributes.

- TOASTed 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 refers (by Oid `chunk_id`) 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    |
| <b>Total</b> | <b>30064</b> |

# The Curse of TOAST

- TOAST is a very stable technology, which just works !
- TOAST has several hard-coded strategies to work with different data types, but it is "ancient" and knows nothing about jsonb, arrays, and other non-atomic data types.
- TOAST makes no attempt to take into account a workload, for example, append-only data.
- TOAST works only with binary BLOBs, when the TOASTed attribute is being updated, 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, since the whole TOASTed values is logged
  - As a result, performance is too low
- **It's time to improve it !**

# Motivational example (synthetic test)

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

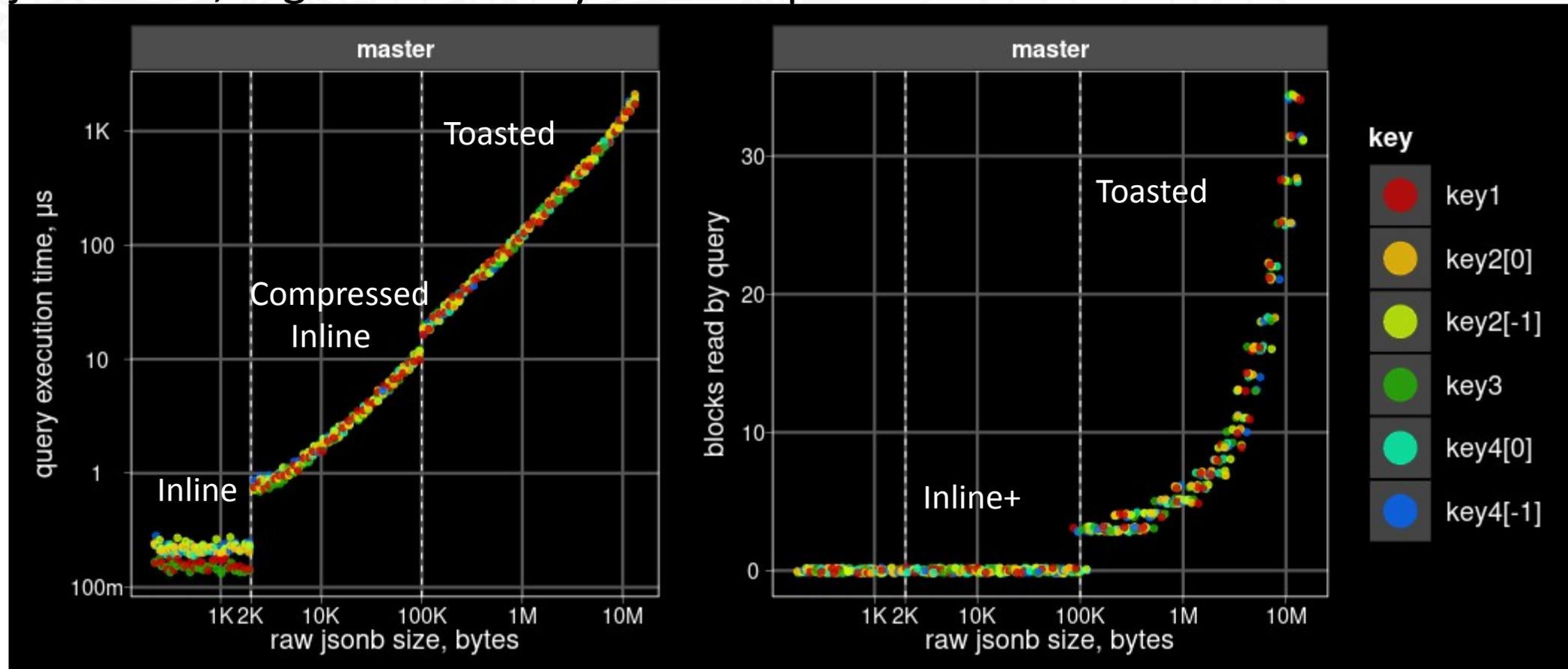
```
CREATE TABLE test_toast AS
SELECT
  i id,
  jsonb_build_object(
    'key1', i,
    'key2', (select jsonb_agg(0) from
              generate_series(1, pow(10, 1 + 5.0 * i / 100.0)::int)), -- 10-100k elems
    'key3', i,
    'key4', (select jsonb_agg(0) from
              generate_series(1, pow(10,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

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

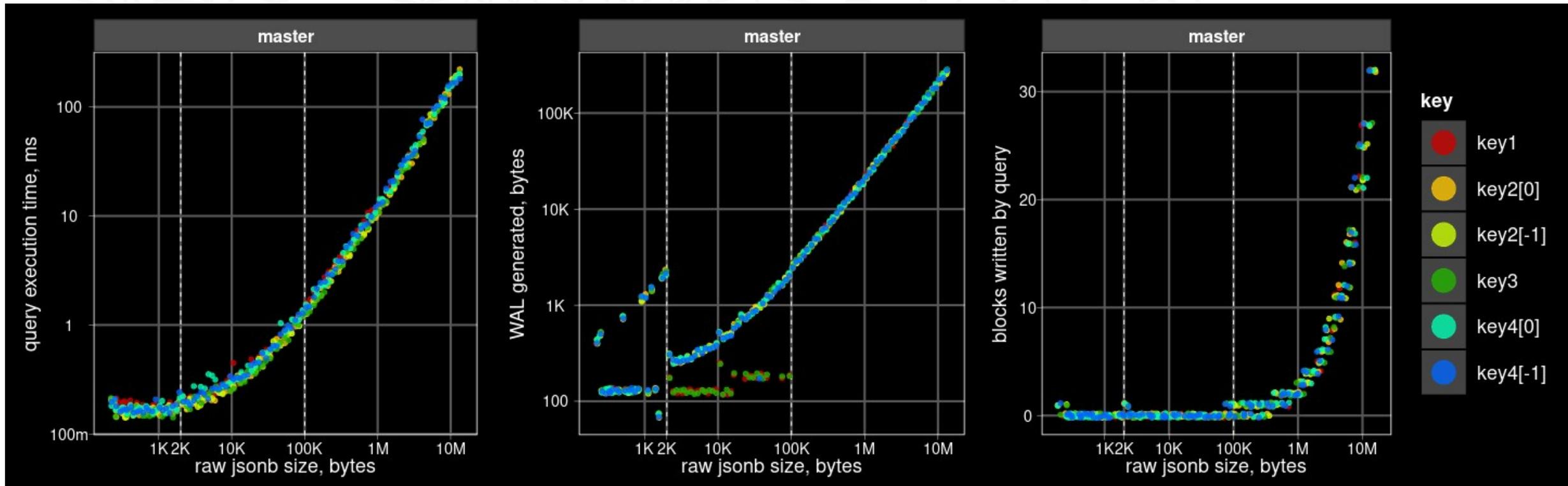


← Large jsonb is TOASTed ! →

# Motivational example: UPDATE

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

```
UPDATE test_toast SET jcb = jsonb_set (jcb, {keyN,...}, ?); COMMIT;
```



# Jsonb deTOAST improvements goal

Ideal goal: no dependency on jsonb size and position

- Access/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)`

# 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

- Per-chunk compression – compress each chunk separately instead of slicing value compressed as a whole. This does not give any speedup and only enables random access needed for the next optimization.
- Sliced deTOAST – read and decompress only the necessary chunks to speedup access to elements of long TOASTed arrays or objects. Access time stops to depend from the element offset.
  - Without random access we need decompress all the data from the beginning of jsonb to the end of needed element, including the whole JEntry[] array:

```
array: Decompress(Nelems * 4 + element_offset + element_size) +  
       Detoast (Nelems * 4 + element_offset + element_size)  
object: Decompress(Nkeys * 8 + key_value_offset + key_value_size) +  
       Detoast (Nkeys * 8 + key_value_offset + key_value_size)
```

- With random access we decompress only chunk containing JEntry and chunks containing element value:

```
array: Decompress(element_index * 4 % 2KB) + Detoast(1 chunk) +  
       Decompress(element_size + ~2KB) + Detoast(element_size + ~2KB)  
object: Decompress(Nkeys * 8 + key_name_offset + key_name_size) + Detoast(1 chunk) +  
       Decompress(key_value_size + ~2KB) + Detoast(key_value_size + ~2KB)
```

# Jsonb deTOAST improvements

- Direct TOAST – store inline chunk TID array together with TOAST pointer for elimination of index TOAST lookups.
  - Only ~330 (2KB/6B) chunks TIDs can fit into inline area.
  - A ~10x speedup for random access to individual chunks of values of  $\leq 660$  KB size.
  - But VACUUM FULL is broken, because we need to rewrite TIDs in TOAST pointers in the main table when chunks are moved inside the TOAST table (indirect access by index helped to avoid this).
- Compress TIDs – compress inline chunk TID array by the same technique that used in GIN indexes for compression of posting lists.
  - Increases number of inline TIDs up to 6x, so up to ~4MB of chunked data can be accessed directly from the TOAST pointer.

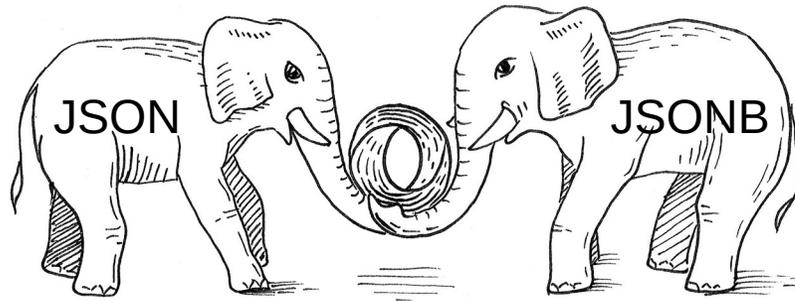
# Jsonb deTOAST improvements

- 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 not changed (in-place update) and new value fits into inline.
  - Old values 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

# Implementation details

- We implemented all optimizations inside TOAST (TOAST+) with the help of Generic JSON interface (GJSON), which allows easy experimenting. Details are in "Understanding Jsonb Performance", "Scaling Jsonb" talks (links in References slide).
- It is hard to imagine that PG community will accept our optimizations, which touched a lot of PG Core.
- The only legal way is to go by "Postgres way" - make TOAST extensible (with backward compatibility).
- TOAST API would allow developers to develop new toasters without touching the Postgres core (more details in Part 2).
- We moved all optimizations from TOAST+ to jsonb\_toaster (TOASTER).

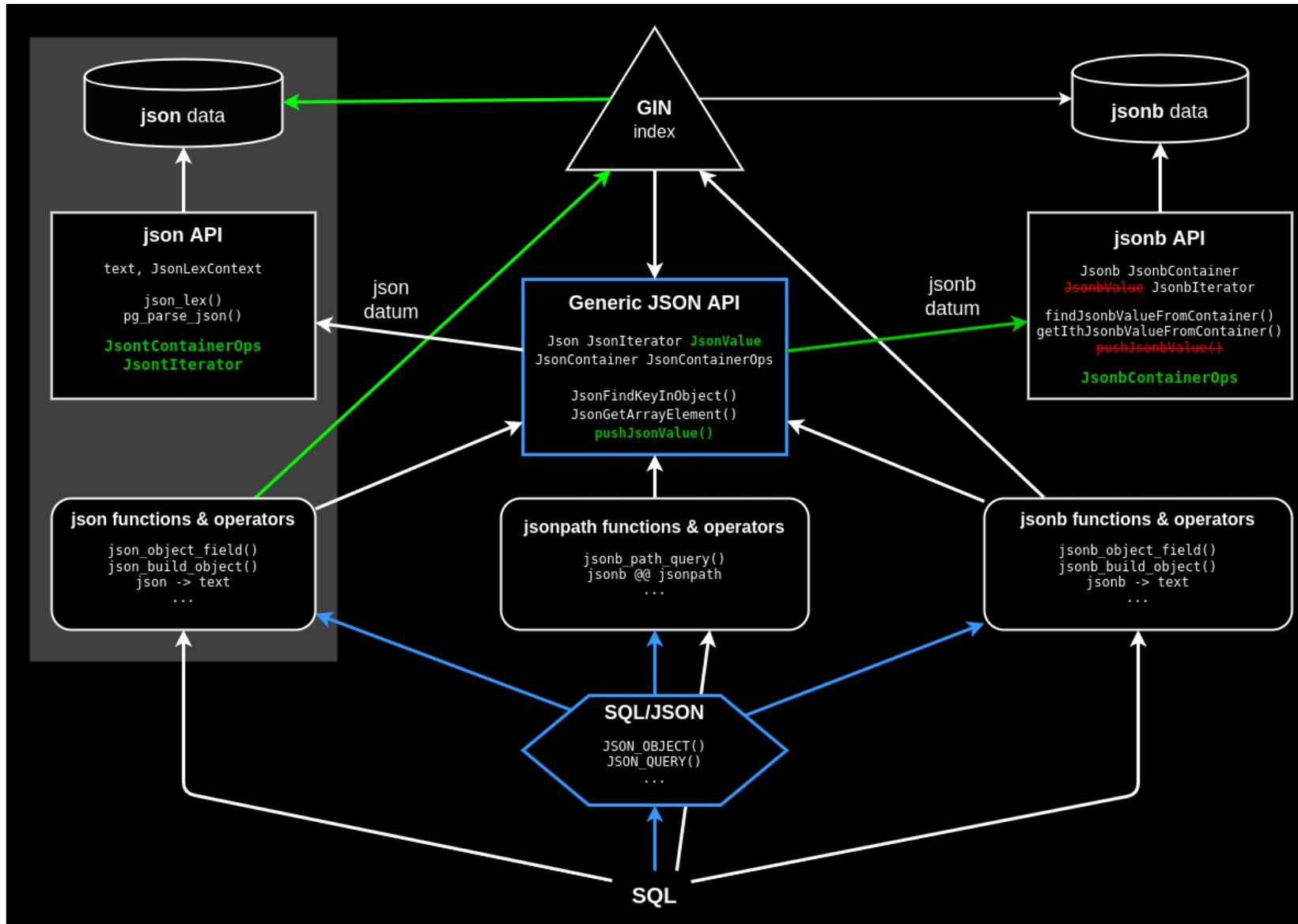
# Generic JSON (GJSON) API



- JSON, JSONB — two implementations of users functions, too many overlaps.
- Coming SQL standard (JSON 2.0) will specify JSON data type.
  - Oracle 20c already introduced JSON data type, which stored in binary format OSON
  - GJSON provides compatibility to SQL Standard (JSON data type) and performance of JSONB - SET SQL\_JSON = JSONB | JSON;
- More details in "JSON and JSONB Unification (GJSON)" talk.

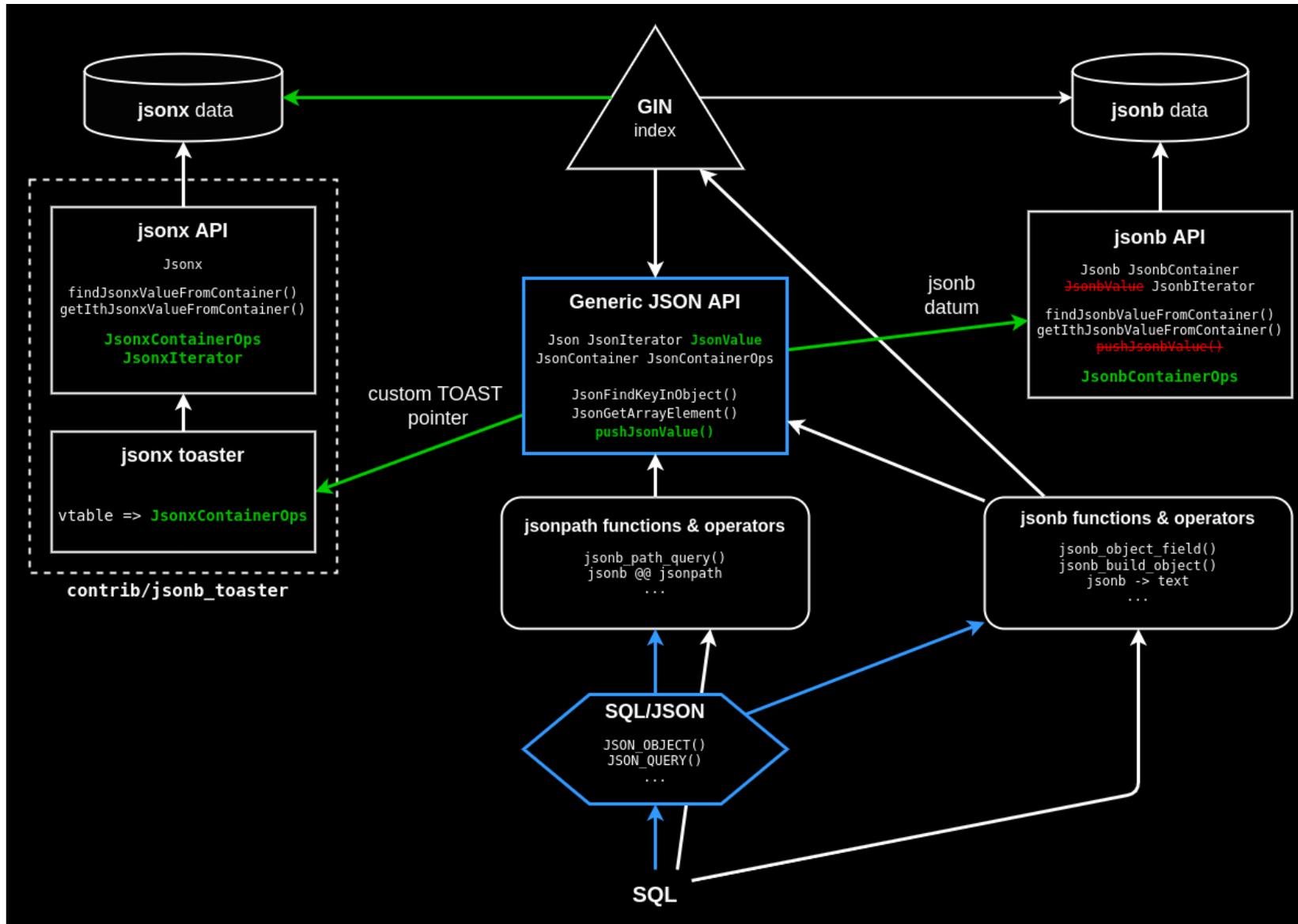
```
[local]:5555 postgres@postgres=# \df json_
json_agg                json_each_text         json_out
json_agg_finalfn       json_extract_path      json_populate_record
json_agg_transfn       json_extract_path_text json_populate_recordset
json_array_element     json_in                json_recv
json_array_element_text json_object            json_send
json_array_elements    json_object_agg       json_strip_nulls
json_array_elements_text json_object_agg_finalfn json_to_record
json_array_length      json_object_agg_transfn json_to_recordset
json_build_array       json_object_field     json_to_tsvector
json_build_object     json_object_field_text json_typeof
json_each              json_object_keys
[local]:5555 postgres@postgres=# \df jsonb_
jsonb_agg                jsonb_extract_path      jsonb_path_match
jsonb_agg_finalfn       jsonb_extract_path_text jsonb_path_match_opr
jsonb_agg_transfn      jsonb_ge                jsonb_path_match_tz
jsonb_array_element     jsonb_gt                jsonb_path_query
jsonb_array_element_text jsonb_hash              jsonb_path_query_array
jsonb_array_elements    jsonb_hash_extended    jsonb_path_query_array_tz
jsonb_array_elements_text jsonb_in                jsonb_path_query_first
jsonb_array_length      jsonb_insert            jsonb_path_query_first_tz
jsonb_build_array       jsonb_le                jsonb_path_query_tz
jsonb_build_object     jsonb_lt                jsonb_populate_record
jsonb_cmp               jsonb_ne                jsonb_populate_recordset
jsonb_concat            jsonb_object            jsonb_pretty
jsonb_contained         jsonb_object_agg       jsonb_recv
jsonb_contains          jsonb_object_agg_finalfn jsonb_send
jsonb_delete            jsonb_object_agg_transfn jsonb_set
jsonb_delete_path      jsonb_object_field     jsonb_set_lax
jsonb_each              jsonb_object_field_text jsonb_strip_nulls
jsonb_each_text        jsonb_object_keys      jsonb_to_record
jsonb_eq                jsonb_out              jsonb_to_recordset
jsonb_exists            jsonb_path_exists      jsonb_to_tsvector
jsonb_exists_all        jsonb_path_exists_opr  jsonb_typeof
jsonb_exists_any       jsonb_path_exists_tz
```

# GSON: Unification of Jsonb and Json



- Generic JSON API allows reuse the code of jsonb user functions and operators for json data type.
- The old code of json functions and operators is almost completely removed.
- Json and jsonb user functions have only the separate entry in which input datums are wrapped into Json structures. But they share the same body by the calling generic subroutine.
- SQL/JSON uses these generic subroutines and generic jsonpath API.
- It is easy to add new features like partial decompression/detoasting or slicing, different storage formats (jsonbc, bson, oson,...).
- GSON allows to implement GIN opclass for indexing JSON.

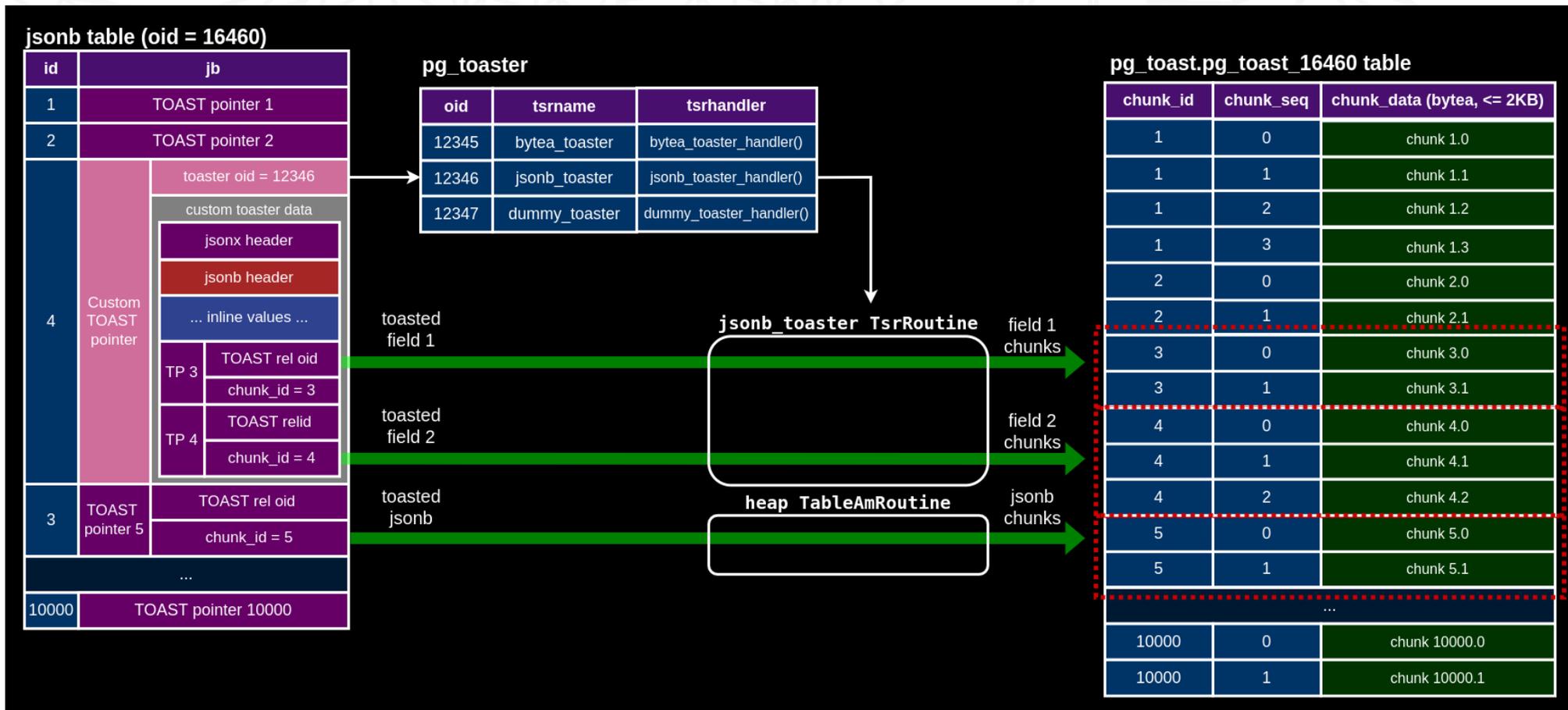
# GSON: Jsonx – optimized Jsonb for Toaster



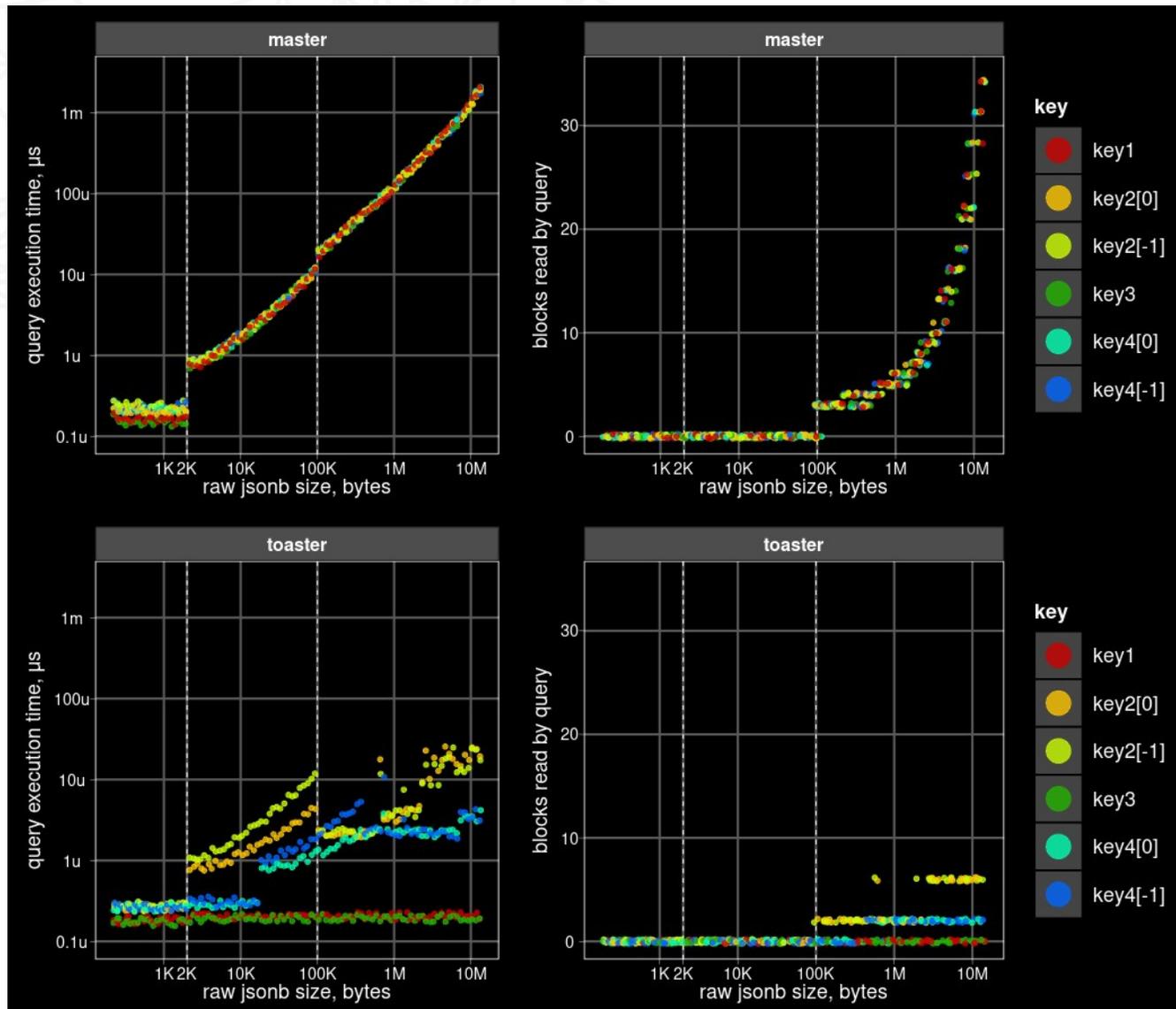
- We removed JSON from GSON for simplicity, it may be added in any time.
- GSON allows us to create JSONX – an internal representation of JSONB in jsonb toaster, where we moved all our optimizations.
- JSONB remains untouched.

# TOASTer access

- Pluggable TOAST introduces Custom TOAST Pointer with information about *toaster* and its data. In this example ( B-tree index is omitted for simplicity) *jsonb\_toaster* is used for toasting tuple with *id=4*, while other tuples are processed by *default toaster*, for example, tuple with *id=3*.

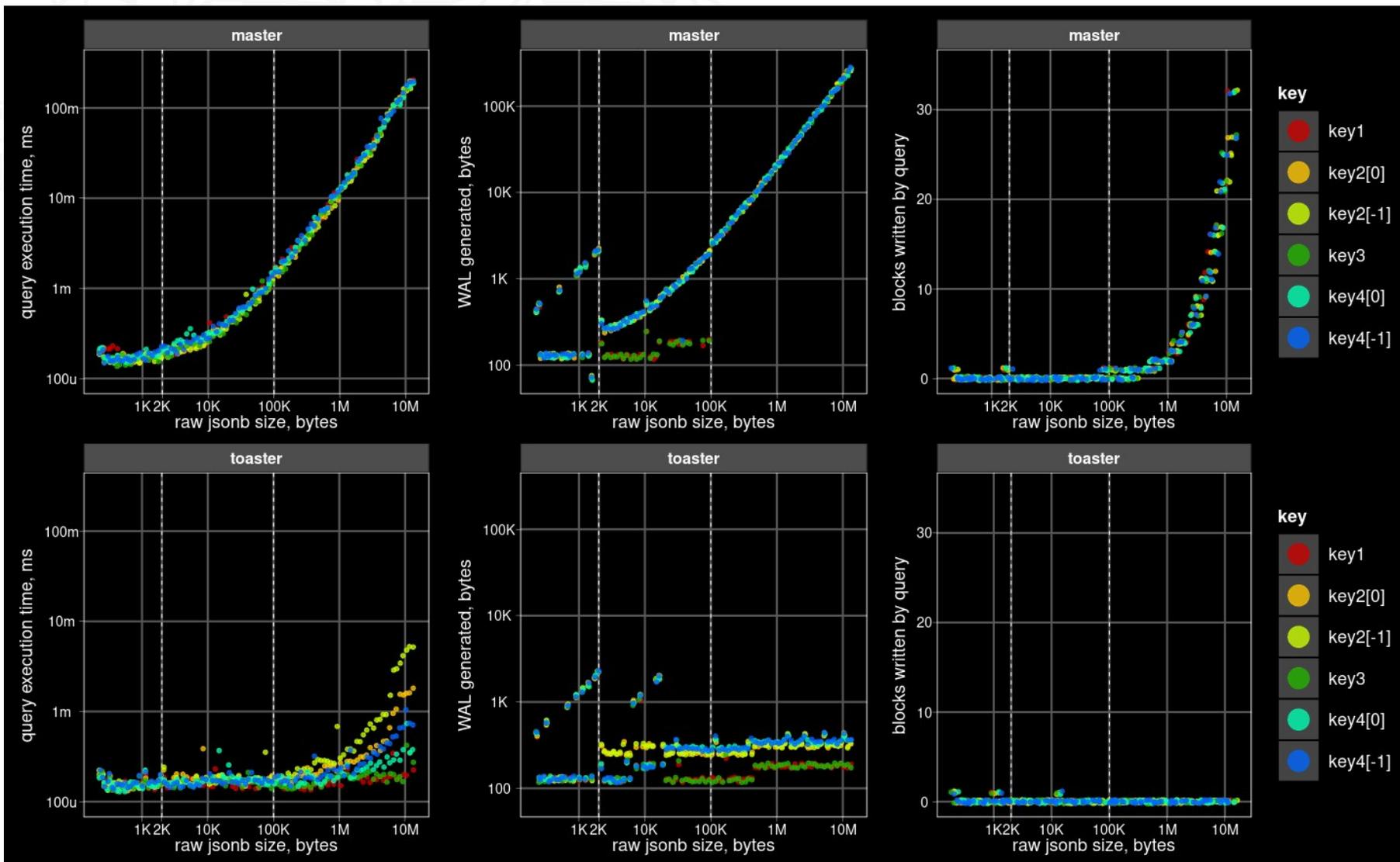


# JSONB SELECT: TOASTER vs MASTER

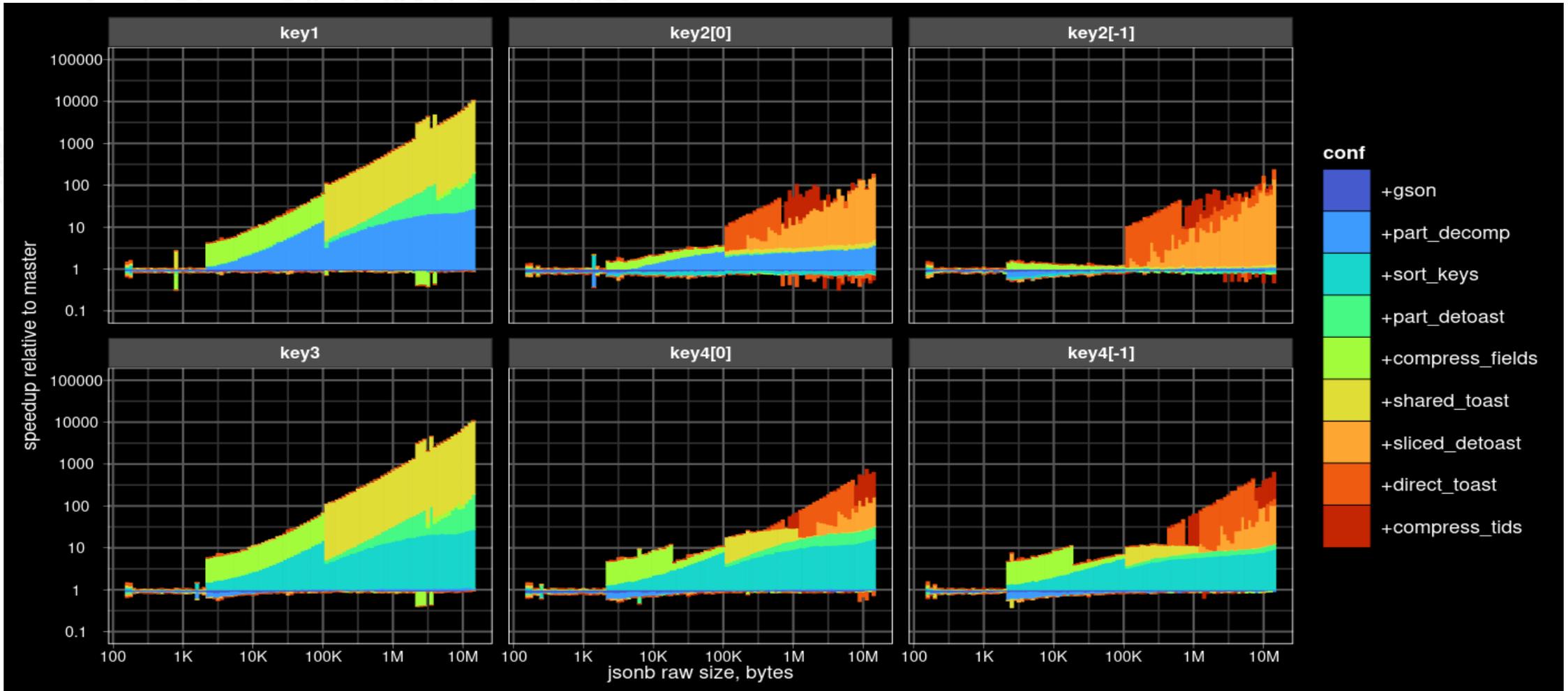


# JSONB UPDATE: TOASTER vs Master

```
UPDATE test_toast SET jb = jsonb_set (jb, {keyN,...}, ?); COMMIT;
```



# Select Speedup: TOASTER vs Master

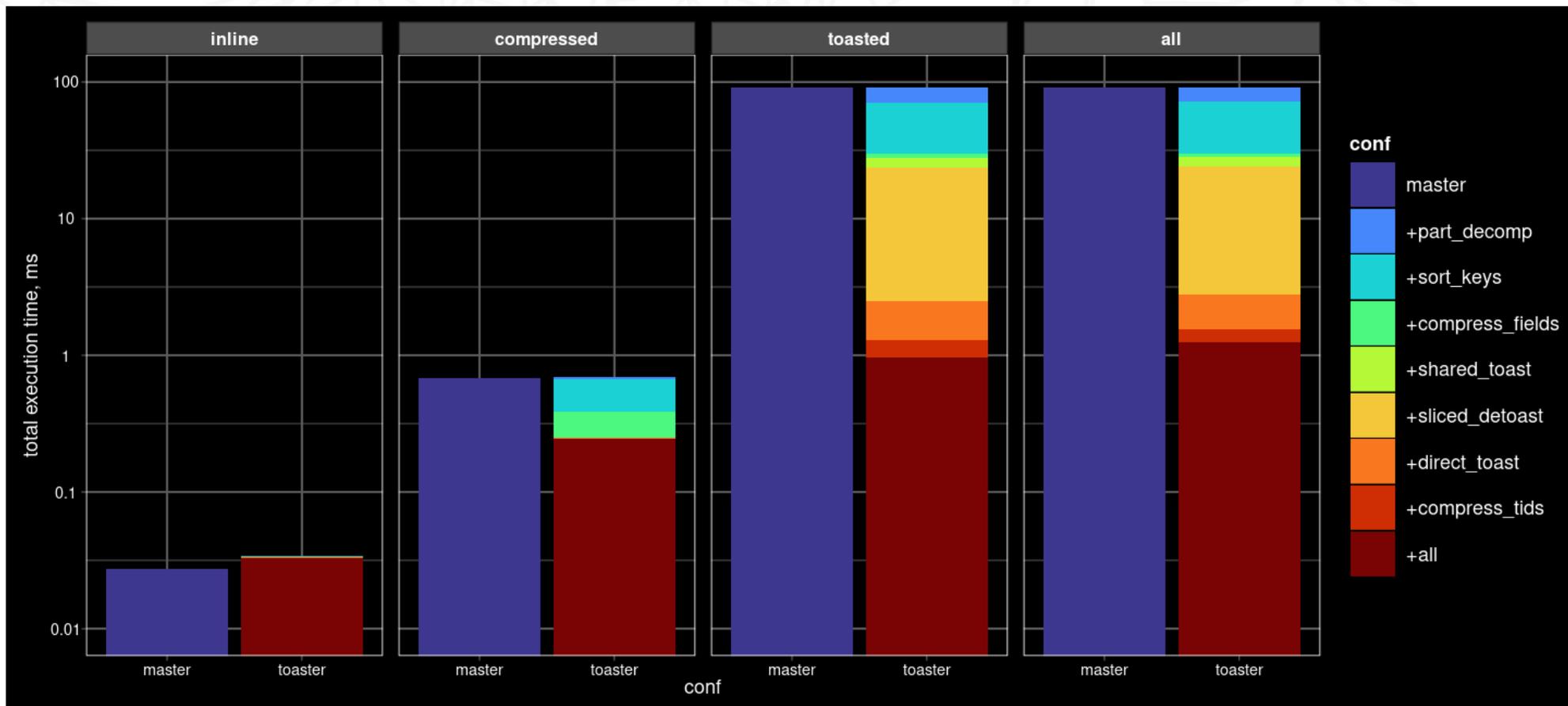


- Access to short keys (key1, key3) improved by 1000X, no dependency on size and positions
- Access to array elements also improved, but require more optimizations.

# SELECT JSONB: Aggregated statistics

TOASTER improves access to medium and long size (2 orders of magnitude) JSONB with minimal overhead for inline data.

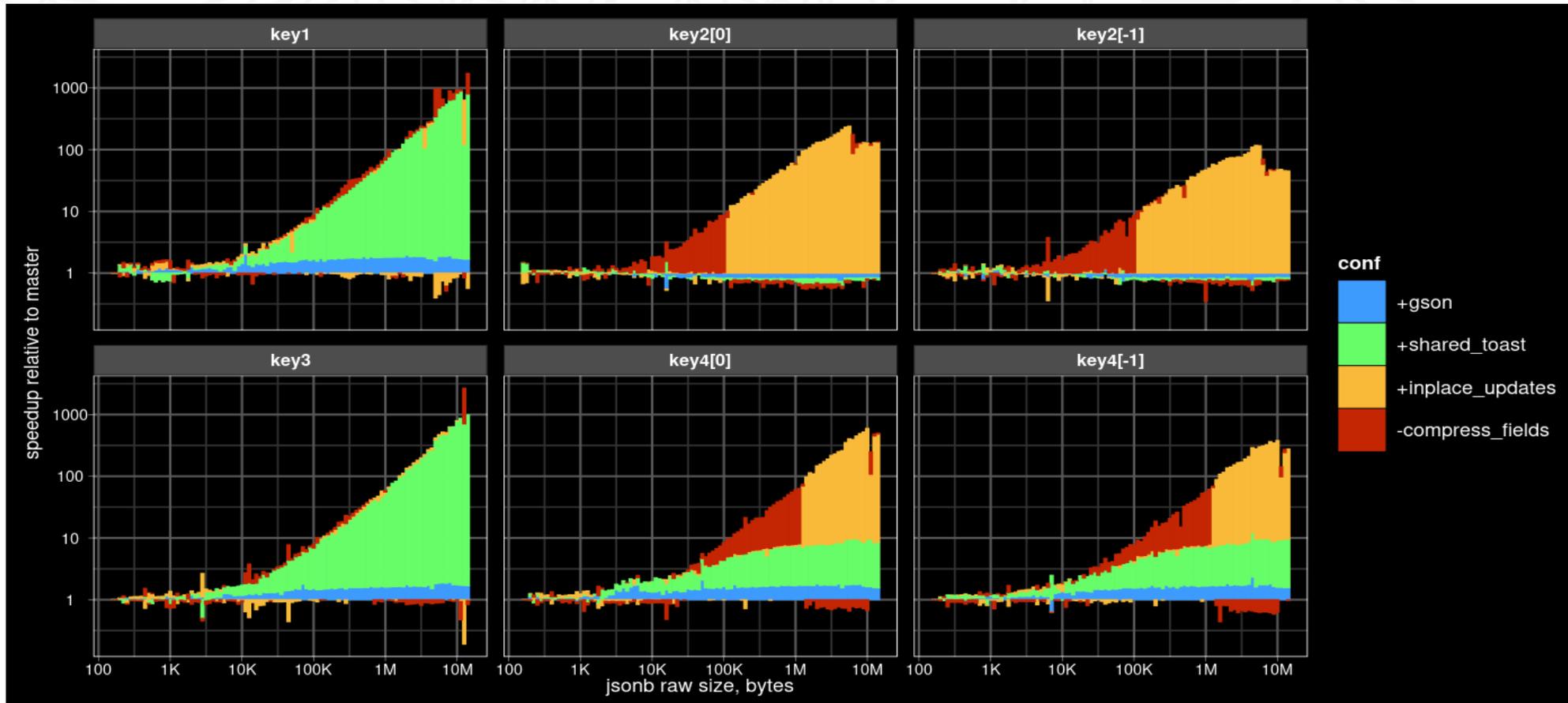
```
SELECT jb -> 'keyN' FROM test_toast WHERE id = ?;
```



# UPDATE Speedup: TOASTER vs Master

TOASTER greatly improves (up to 3 orders ) the performance of update (inline, in-place) of JSONB !

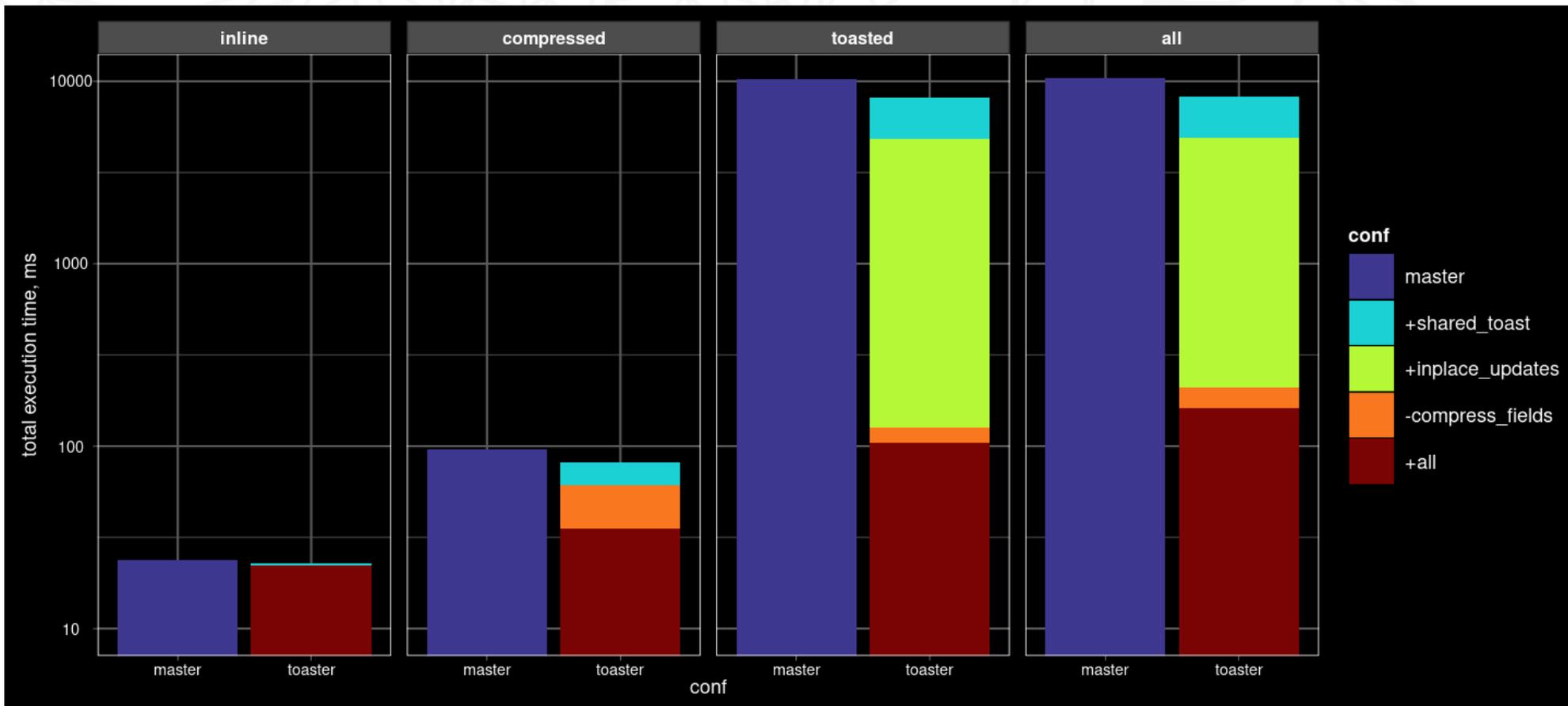
```
UPDATE test_toast SET js = jsonb_test (jb, {keyN,...}, ?);
```



# UPDATE JSONB: Aggregated statistics

TOASTER improves update of medium and long size (2 orders) JSONB with minimal overhead for inline data.

```
UPDATE test_toast SET js = jsonb_test (jb, {keyN,...}, ?);
```



# Appendable Bytea Toaster

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 (**TOAST**): 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**

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

Append 1 byte to bytea (**BYTEA TOASTER**):

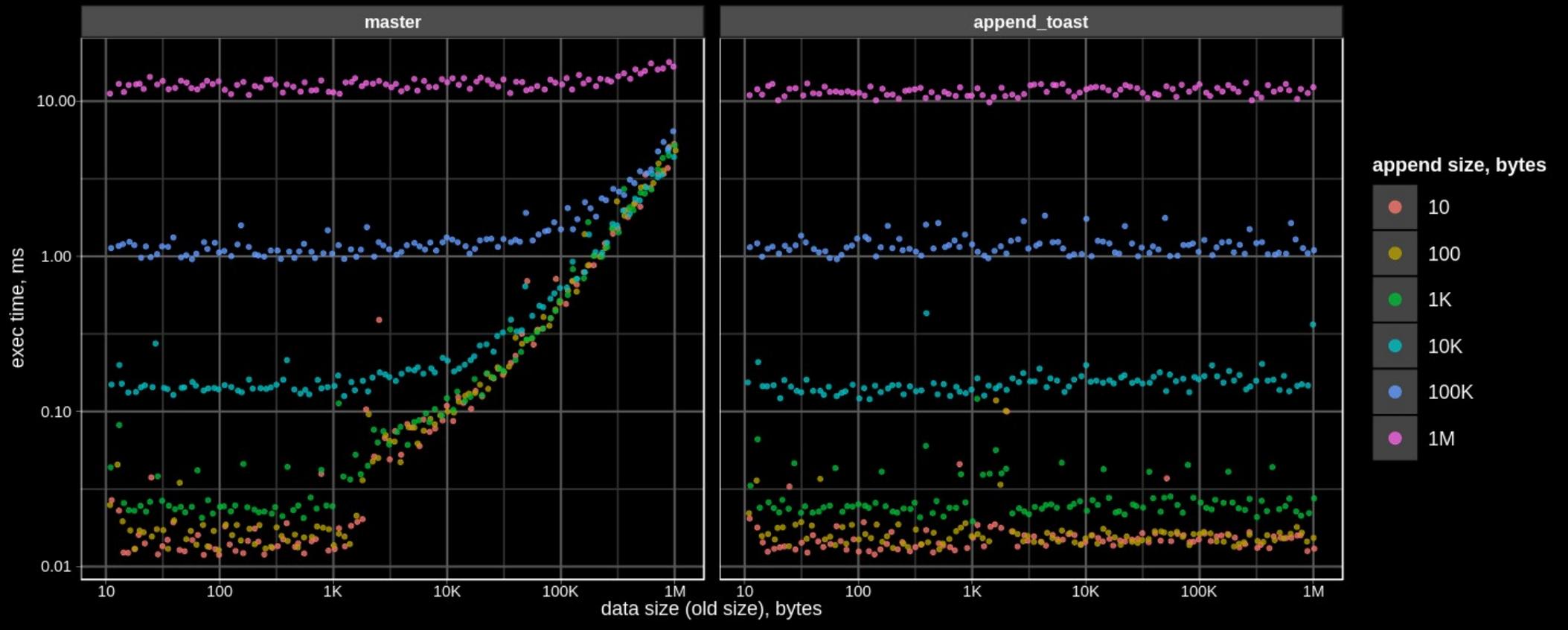
```
Update on test (actual time=0.060..0.061 rows=0 loops=1)  
  Buffers: shared hit=2  
-> Seq Scan on test (actual time=0.017..0.020 rows=1 loops=1)  
   Buffers: shared hit=1  
Planning Time: 0.727 ms
```

Execution Time: **0.496 ms** (**2750x**)

Table size remain 100 MB, 143 bytes of WAL generated.

# Results – query execution time

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



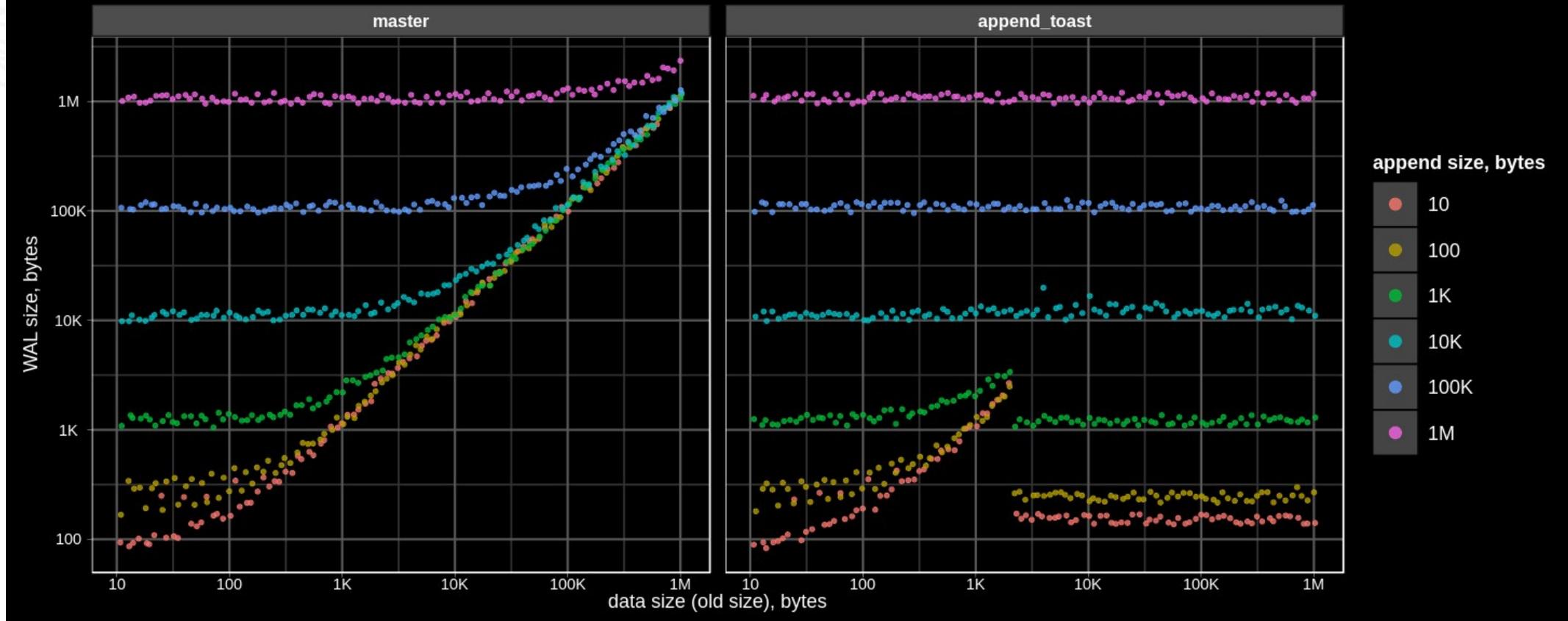
OLD + APPEND SIZE



APPEND SIZE

# Results – WAL traffic

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



OLD + APPEND SIZE



INLINE OLD + APPEND SIZE

# Conclusions

- TOAST optimizations based on knowledge of JSONB format resulted in a significant (several orders of magnitude) speedup of SELECT and UPDATE queries. Ideal goal: no dependency on jsonb size and position:
  - + SELECT ( need more research on *nested* data )
  - + Inline (short keys), in-place (diff) UPDATE (need more work)
- TOAST can be optimized for specific workload, see for example, "TOAST for appendable bytea" talk (link in References slide), where we demonstrated streaming of binary data into Postgres, practically impossible for vanilla version.
- Postgres Extensibility allows us to Extend Postgres further:  
Data type aware TOAST — Pluggable TOAST !

PLUGGABLE TOAST



ONE AST FITS ALL

# What is The Toaster?

- A set of methods implementing base operations for storage of oversized attributes using TOAST API.
- Aware of internal details of data it deals with
- Could be specific for column (different columns in table could use different toasters)...
- ...Or datatype (jsonb, bytea, etc.)
- ...Or even workload (bytea specific append operation)
- Default toaster — compatible with previous versions

# User's Point Of View

SQL – install custom Toaster:

```
CREATE EXTENSION custom_toaster;
```

```
select * from pg_toaster;
```

| oid   | tsrname        | tsrhandler              |
|-------|----------------|-------------------------|
| 9864  | deftoast       | default_toaster_handler |
| 32772 | custom_toaster | custom_toaster_handler  |

Behind the scenes – create new Toaster:

```
CREATE FUNCTION custom_toaster_handler(internal)
RETURNS toaster_handler
AS 'MODULE_PATHNAME'
LANGUAGE C;
```

```
CREATE TOASTER custom_toaster HANDLER custom_toaster_handler;
```

# Plug In Custom Toaster...

SQL – attach created extension to table column:

```
CREATE TABLE tst1 (  
    c1 text STORAGE plain,  
    c2 text STORAGE external TOASTER custom_toaster,  
    id int4  
);  
ALTER TABLE tst1 ALTER COLUMN c1 SET TOASTER custom_toaster;
```

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

| Column | Type    | Collation | Nullable | Default | Storage  | Toaster        | ... |
|--------|---------|-----------|----------|---------|----------|----------------|-----|
| c1     | text    |           |          |         | plain    | deftoast       | ... |
| c2     | text    |           |          |         | external | custom_toaster | ... |
| id     | integer |           |          |         | plain    |                | ... |

```
Access method: heap
```

# ...as Column Attribute

New table column attribute – “atttoaster”. Used to insert data, but not suitable for update and fetch (as for Storage and Compression options):

```
=# \d pg_attribute
```

```
Table "pg_catalog.pg_attribute"

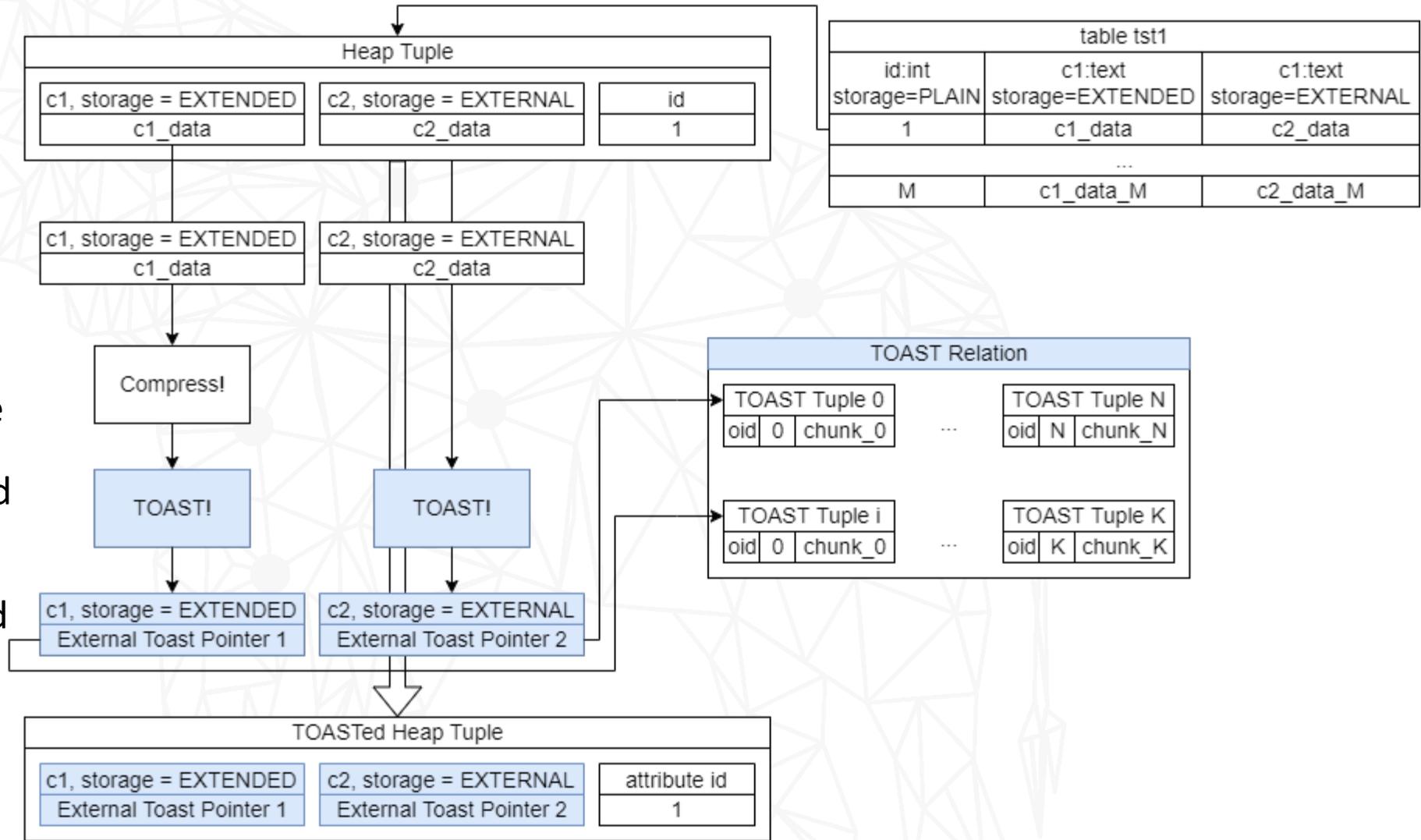
```

| Column            | Type       | Collation | Nullable        | Default |
|-------------------|------------|-----------|-----------------|---------|
| attrelid          | oid        |           | not null        |         |
| attname           | name       |           | not null        |         |
| ...               |            |           |                 |         |
| attstorage        | "char"     |           | not null        |         |
| <b>atttoaster</b> | <b>oid</b> |           | <b>not null</b> |         |
| attcompression    | "char"     |           | not null        |         |
| ...               |            |           |                 |         |

- Data in table can be TOASTed with different Toasters
- Dropping Toaster is not allowed – all data must be re-toasted with the default

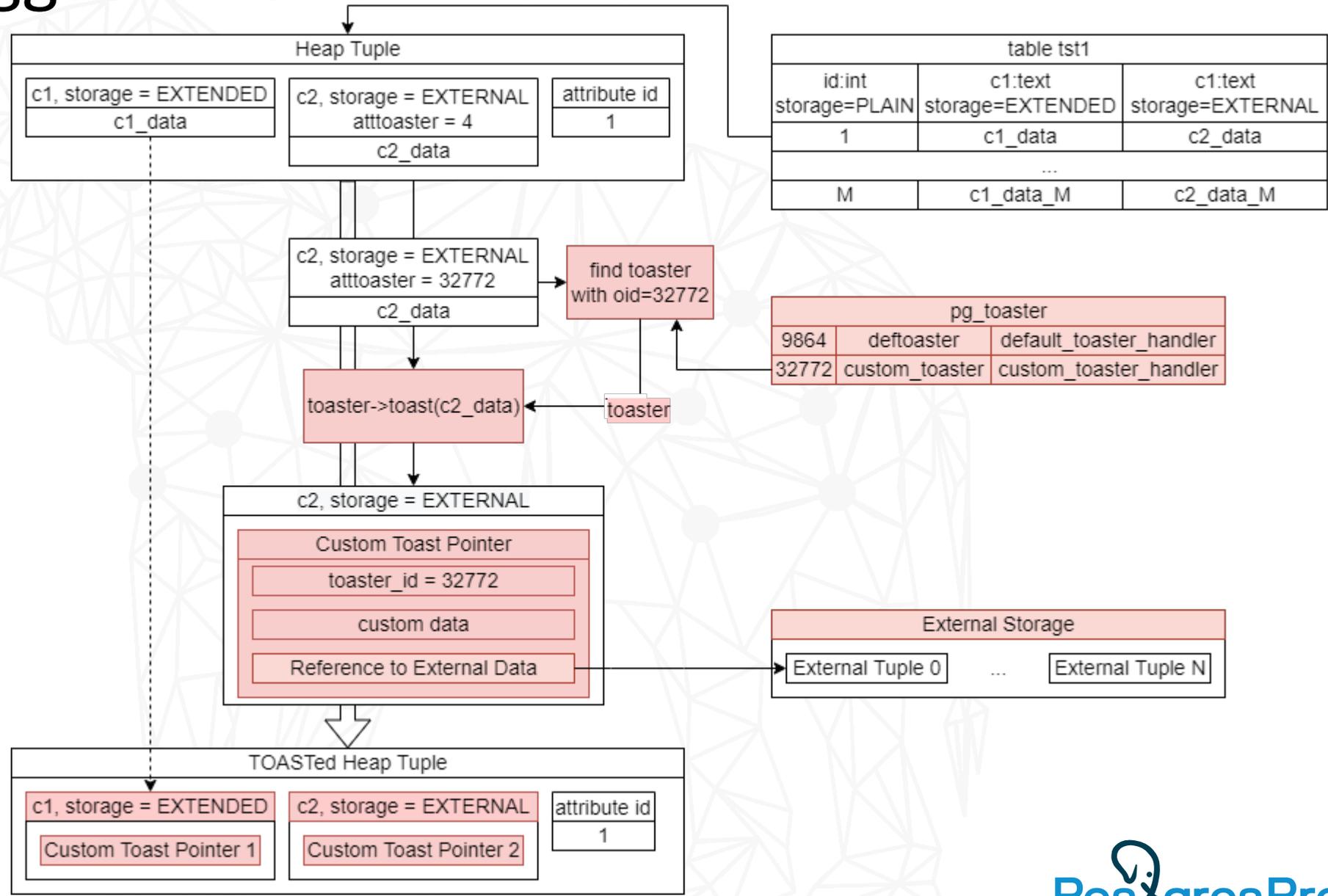
# How TOAST Works

- Attribute stored in TOAST relation and replaced with Toast Pointer
- 4 strategies depending on storage type
- EXTENDED and MAIN are compressed before TOASTing — impossible to use optimizations
- EXTERNAL are TOASTed as-is — any knowledge of internal structure or workload could be used for optimizations, and could be replaced with custom structure keeping storage type and TOASTed data



# How Pluggable TOAST Works

- Attribute is replaced with Custom Toast Pointer created by specific Toaster
- TOAST and storage details are hidden from AM behind the Toaster
- Custom Toast Pointer header contains only data necessary for Toaster lookup and required by the Executor
- Toaster can use any knowledge of data structure and workflow
- Same data could be TOASTed with different Toasters



# Under The Hood Of Pluggable TOAST

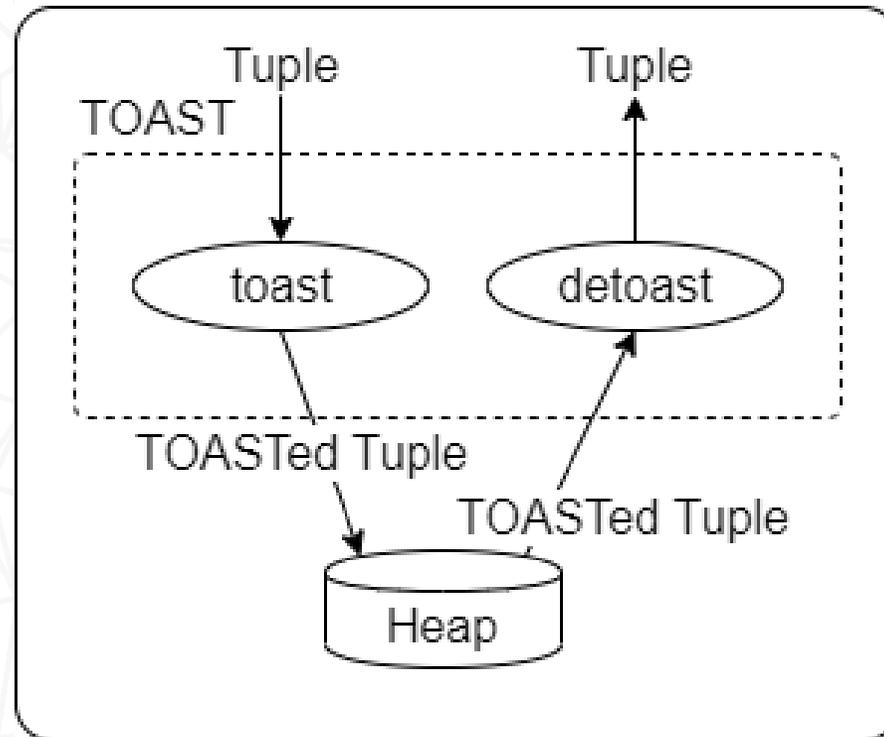


# Current TOAST

TOAST is slicing and storing an oversized value externally to Tuple

- Part of Heap AM (Core)
- Not extensible
- Same strategy for all datatypes
- Not effective for structured data (json) or data required special workflow (bytea)

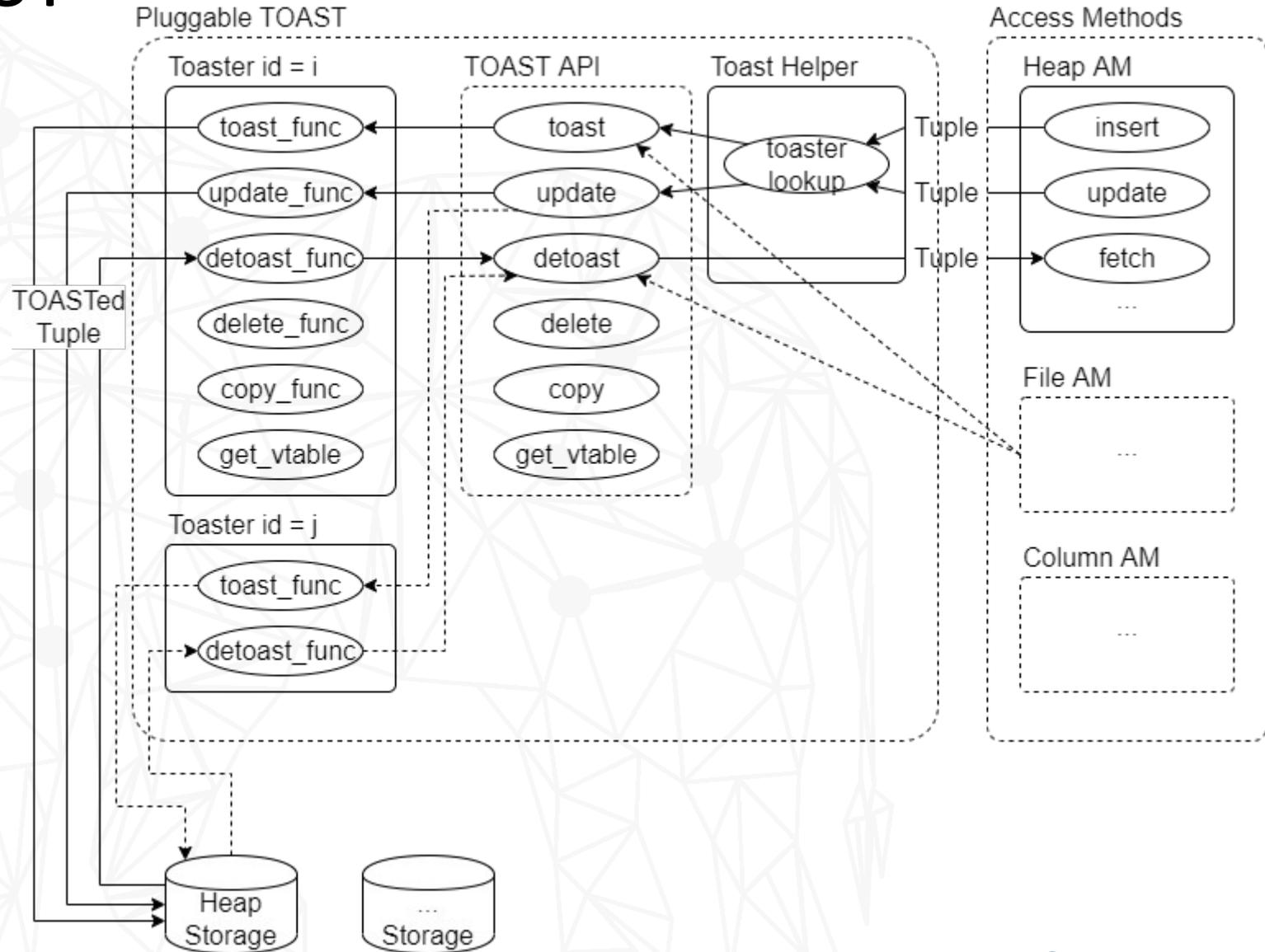
Heap AM



# Pluggable TOAST

TOAST is storing a value outside Tuple, with means and storage depending on Toaster

- Detached from Heap AM
- TOAST API
- Extensible
- Reference Toasters  
effective for heavy data types – json and bytea



# Custom Toast Pointer

- Existing External (Default) Toast Pointer – varatt\_external (=16 bytes header, 4 bytes alignment)

```
typedef struct varatt_external
{
    int32      va_rawsize;      /* Original data size (includes header) */
    uint32     va_extinfo;     /* External saved size (without header) and
                               * compression method */
    Oid        va_valueid;     /* Unique ID of value within TOAST table */
    Oid        va_toastrelid;  /* RelID of TOAST table containing it */
}
varatt_external;
```

- Introduce new Custom Toast Pointer – varatt\_custom (>=12 bytes header, 2 bytes alignment)

```
typedef struct varatt_custom
{
    uint32     va_toasterdatalen; /* total size of toast pointer, < BLCKSZ */
    uint32     va_rawsize;      /* Original data size (includes header) */
    uint32     va_toasterid;    /* Toaster ID, actually Oid */
    char       va_toasterdata[FLEXIBLE_ARRAY_MEMBER]; /* Custom toaster data */
}
varatt_custom;
```

# TOAST API Abstraction

Table Access Method

---

**TOAST Helper – search Toaster by Toaster ID, call Toaster functions via API**

---

**TOAST API – unified extensible TOAST interface**

---

**Toaster with TOAST API implementation – Default toaster is built-in, Custom toasters are plugged in as extensions**

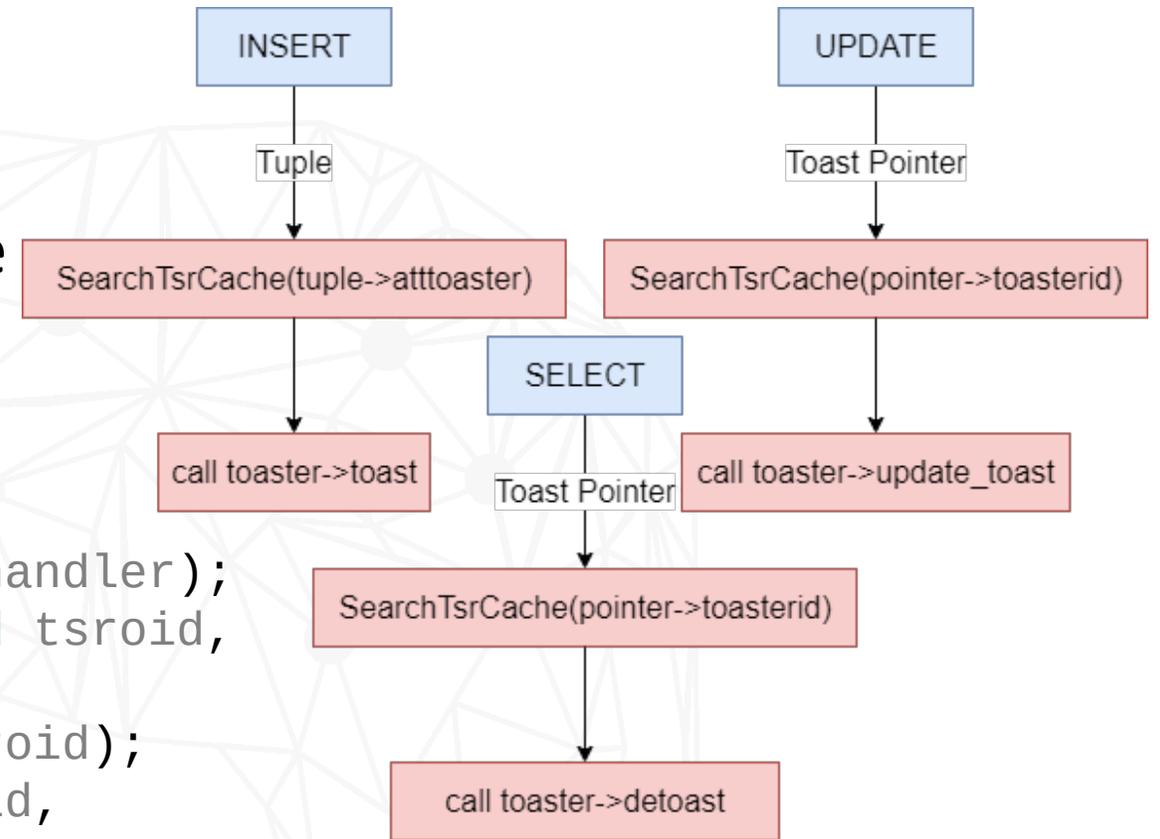
---

Storage – Toasters could use any Storage and Access Methods to store data, independently of TAM

# Toast Helper

- Toaster lookup functions
- input Toast Pointer type recognition
- `column::atttoaster` attribute check
- call Toaster

```
extern TsrRoutine *GetTsrRoutine(Oid tsrhandler);
extern TsrRoutine *GetTsrRoutineByOid(Oid tsroid,
    bool noerror);
extern TsrRoutine *SearchTsrCache(Oid tsroid);
extern bool validateToaster(Oid toasteroid,
    Oid typeoid,
    char storage,
    char compression,
    Oid amoid,
    bool false_ok);
```



# Backwards Compatibility

Default Toaster works with original Toast Pointers, and is enabled by default until changed by

```
ALTER TABLE table ALTER COLUMN column SET TOASTER toaster;
```

Keep EXTERNAL TOAST Pointer for compatibility:

```
if (VARATT_IS_EXTERNAL_ONDISK(attr))
{
    attr = PG_DETOAST_DATUM(PointerGetDatum(attr));
}
else if (VARATT_IS_CUSTOM(attr))
{
    Oid toasterid = VARATT_CUSTOM_GET_TOASTERID(attr);
    TsrRoutine *toaster = SearchTsrCache(toasterid);
    CustomToastRoutine *vtable = toaster->get_vtable(PointerGetDatum(attr));
    vtable-> ...
}
```

# TOAST API Core Functions

```
typedef struct TsrRoutine {
    NodeTag    type;
    /* interface functions */
    toast_init init;           // Initialization, storage creation,
etc
    toast_function toast;           // Toast
    update_toast_function update_toast; // Update toasted value
    copy_toast_function copy_toast;    // Copy toasted value
    detoast_function detoast;       // Detoast
    del_toast_function deltoast;       // Delete toasted value
    get_vtable_function get_vtable;    // Return table of virtual functions
    toastervalidate_function toastervalidate; // Toaster validation
}
```

*\*Required functions highlighted in red*

*\*\*For code sample see Addendum 1*

# Toaster Validation

```
/* 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 depending on type, compression and storage options;
- Validation function must be aware of Storage and Compression because this is currently handled outside of Toaster;
- Validation could take in account that Toaster could be specific for datatype and workload

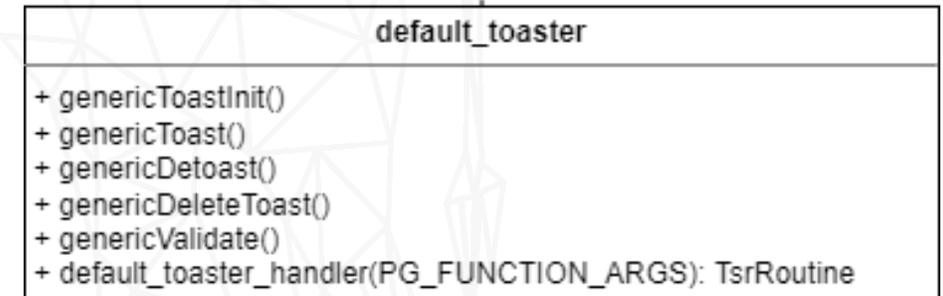
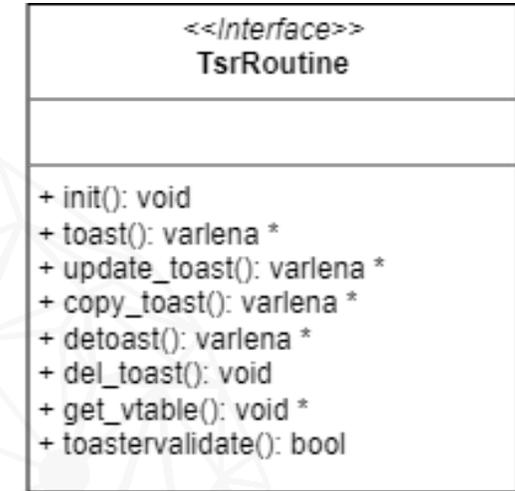
# Default TOAST via TOAST API

pg\_toaster table initial contents

```
{ oid => '9864', oid_symbol => 'DEFAULT_TOASTER_OID',  
  descr => 'default toaster',  
  tsrname => 'deftoaster', tsrhandler => 'default_toaster_handler' },
```

Default toaster handler

```
static bool  
genericValidate (Oid typeoid, char storage, char compression,  
                Oid amoid, bool false_ok)  
{  
    return true;  
}  
Datum  
default_toaster_handler(PG_FUNCTION_ARGS)  
{  
    TsrRoutine *tsrroutine = makeNode(TsrRoutine);  
    tsrroutine->init = genericToastInit;  
    tsrroutine->toast = genericToast;  
    tsrroutine->detoast = genericDetoast;  
    tsrroutine->deltoast = genericDeleteToast;  
    tsrroutine->get_vtable = NULL;  
    tsrroutine->copy_toast = NULL;  
    tsrroutine->update_toast = NULL;  
    tsrroutine->toastvalidate = genericValidate;  
    PG_RETURN_POINTER(tsrroutine);  
}
```

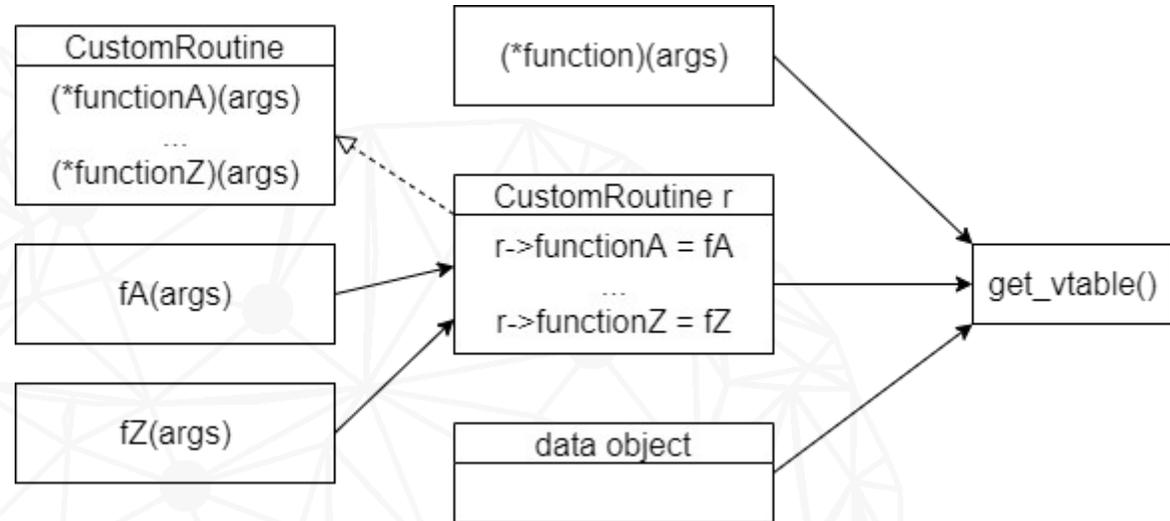


# Pimp My API – The Virtual Functions Table

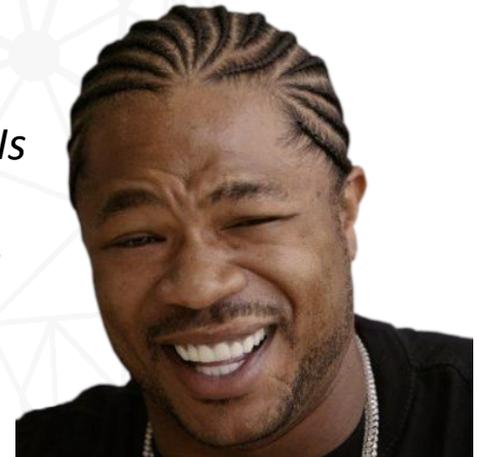
- User-defined
- Provides extensibility

```
void *  
get_vtable()
```

A void pointer can be casted to any type – object, function or container



*Yo dawg we heard you like APIs  
so we put an API in our API so  
you can extend Postgres while  
you extend Postgres*



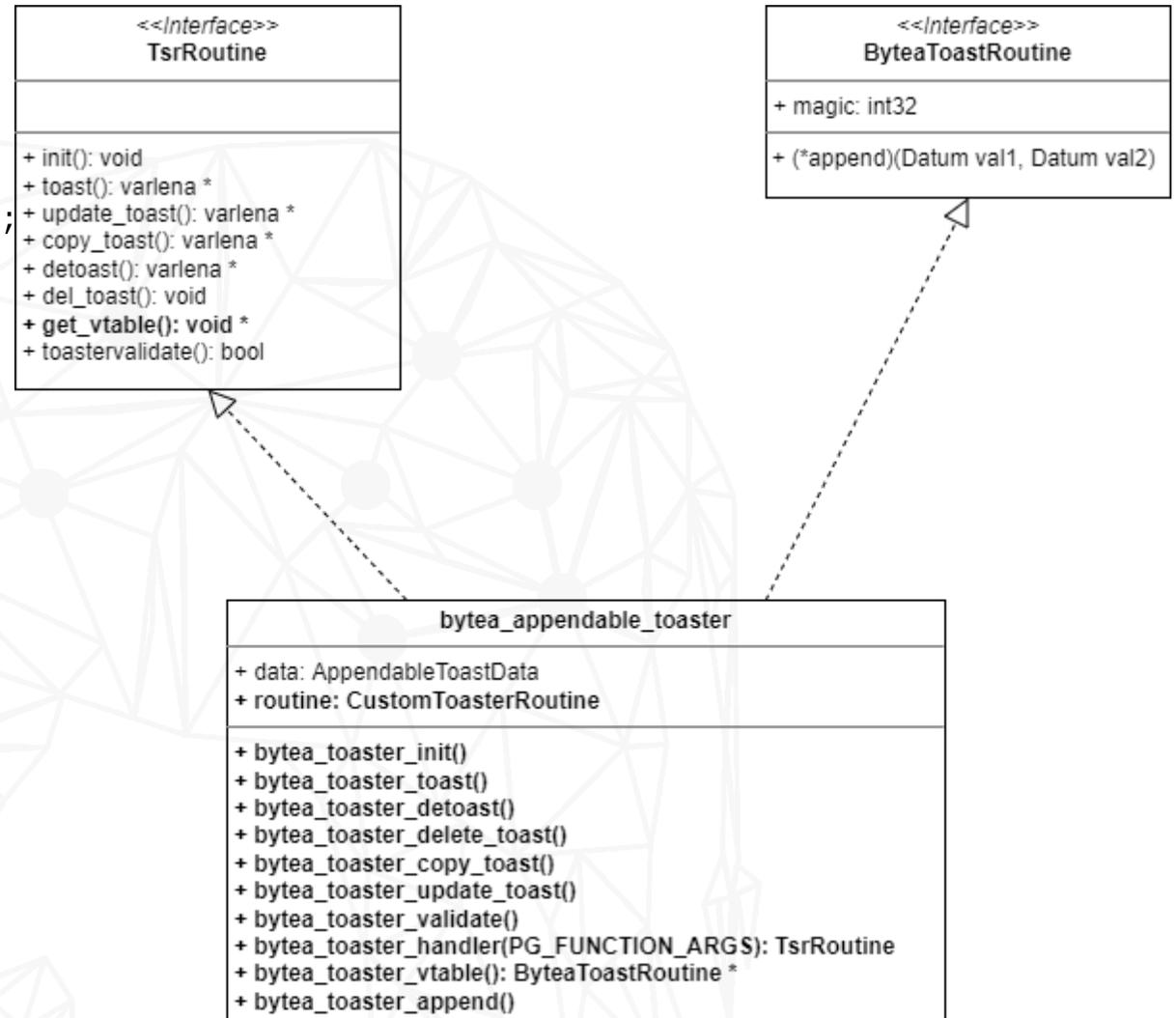
# Toaster with User Functions – bytea Toaster

## Code Sample: SQL

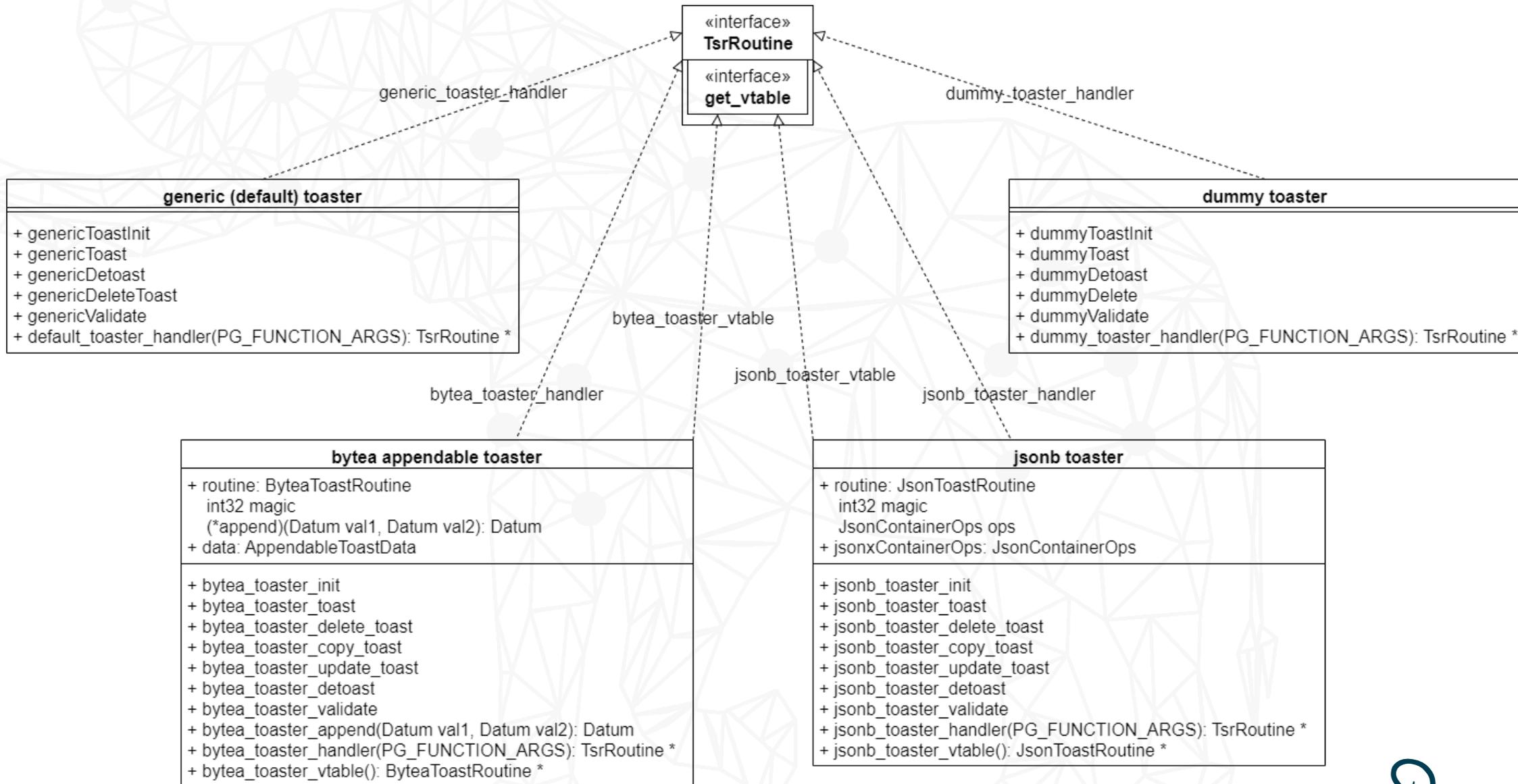
```
CREATE FUNCTION bytea_toaster_handler(internal)
RETURNS toaster_handler
AS 'MODULE_PATHNAME'
LANGUAGE C;
CREATE TOASTER bytea_toaster HANDLER bytea_toaster_handler;
```

## Code Sample: C

```
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;
}
PG_FUNCTION_INFO_V1(bytea_toaster_handler);
Datum
bytea_toaster_handler(PG_FUNCTION_ARGS)
{
    TsrRoutine *tsr = makeNode(TsrRoutine);
    tsr->init = bytea_toaster_init;
    tsr->toast = bytea_toaster_toast;
    tsr->deltoast = bytea_toaster_delete_toast;
    tsr->copy_toast = bytea_toaster_copy_toast;
    tsr->update_toast = bytea_toaster_update_toast;
    tsr->detoast = bytea_toaster_detoast;
    tsr->toastvalidate = bytea_toaster_validate;
    tsr->get_vtable = bytea_toaster_vtable;
    PG_RETURN_POINTER(tsr);
}
```



# Toasters



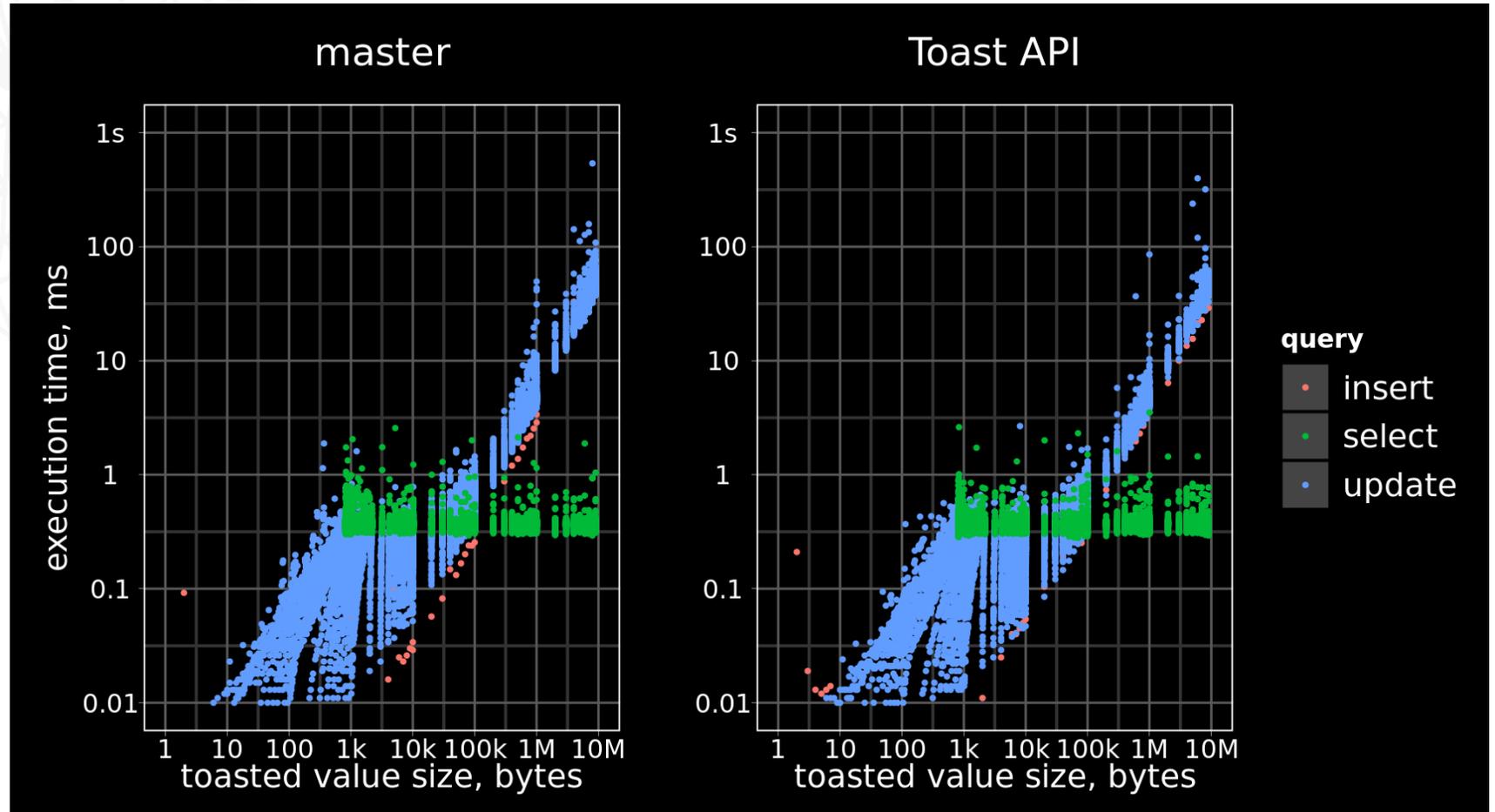
# Overhead of TOAST API

## Default Toaster

```
INSERT into  
test_toast_p VALUES  
(i, repeat('a',  
size));
```

```
UPDATE test_toast_p  
SET jb = repeat('b',  
pg_column_size(jb)+1)  
where id=i;
```

```
SELECT jb FROM  
test_toast_p WHERE  
id=i;
```



# TOAST API Deliverables

- TOAST API (SQL syntax, core changes, patch set for PG15-16)
- Documentation on SQL Toaster Syntax (patch set for PG15-16)
- Default toaster – built-in (Core changes, included in TOAST API patch set);
- Dummy toaster as an example for developers – extension */contrib/dummy\_toaster*
- Bytea appendable toaster – extension */contrib/bytea\_toaster*
- Jsonb toaster – extension */contrib/jsonb\_toaster*

# Conclusion

- The legal way to introduce powerful optimizations to internal data storage without changing or modifying Table AM;
- Extensible TOAST, pluggable Toasters;
- Toasters could use any Table Access Method and vice versa;
- Custom Toasters could be distributed under different licenses;
- Toasters could be extended not only to store, but to compress, encrypt and do other data transformations;
- Additional patch set – jsonb toaster as an extension, GSON API

# Open Issues

- Drop Toaster – whole DB needs to be re-toasted if DROP TOASTER possible, otherwise TOASTed tuples won't be restored;
- Move Storage and Compression options inside TOAST
- CREATE AS SELECT and INSERT AS SELECT operations re-toast data with Default Toaster no matter which Toaster is used in source table unless the target table is created and Toaster set before insertion

# TODO

## TOAST API

- Introduce iterators to Default Toaster
- Documentation for Toasters
- Benchmark TOAST API and Toasters
- Support Table AM and Compression in Toaster

## JSONB

- Updates for long chunked arrays (rewrite only affected chunks)
- GSON update API (maybe updatable iterators)
- GSON optimization (to reduce overhead of wrapping)
- TOAST for long scalars (needs JsonbValue refactoring)
- Support for renames of TOASTed keys (need more complex old-vs-new value comparison in `toast_update()`)
- **Goal – commit Pluggable Toast to PG16!**

# References

- Our experiments:
  - Understanding Jsonb performance  
<http://www.sai.msu.su/~megera/postgres/talks/jsonb-pgconfnyc-2021.pdf>
  - JSON and JSONB Unification (GSON)  
<http://www.sai.msu.su/~megera/postgres/talks/json-unification-database-meetup-2020.pdf>
  - Scaling JSONB - <http://www.sai.msu.su/~megera/postgres/talks/jsonb-pgvision-2021.pdf>
  - Slides of this talk <http://www.sai.msu.su/~megera/postgres/talks/toast-pgcon-2022.pdf>
  - Appendable Bytea TOAST  
<http://www.sai.msu.su/~megera/postgres/talks/bytea-pgconfonline-2021.pdf>
  - Pluggable TOAST at Commitfest <https://commitfest.postgresql.org/38/3490/>
  - Jsonb\_toaster @Github (check License.txt) :  
[https://github.com/postgrespro/postgres/tree/jsonb\\_toaster](https://github.com/postgrespro/postgres/tree/jsonb_toaster)
  - Meet us at GitHub [https://github.com/postgrespro/postgres/tree/toasterapi\\_clean](https://github.com/postgrespro/postgres/tree/toasterapi_clean)
- Jsonb is ubiquitous and is continuously developing
  - JSON[B] Roadmap V3, Postgres Build 2020, Dec 8, 2020

ALL

YOU

# POSTERS

IS

Slonik

Teodor Sigaev

Nikita Malakhov



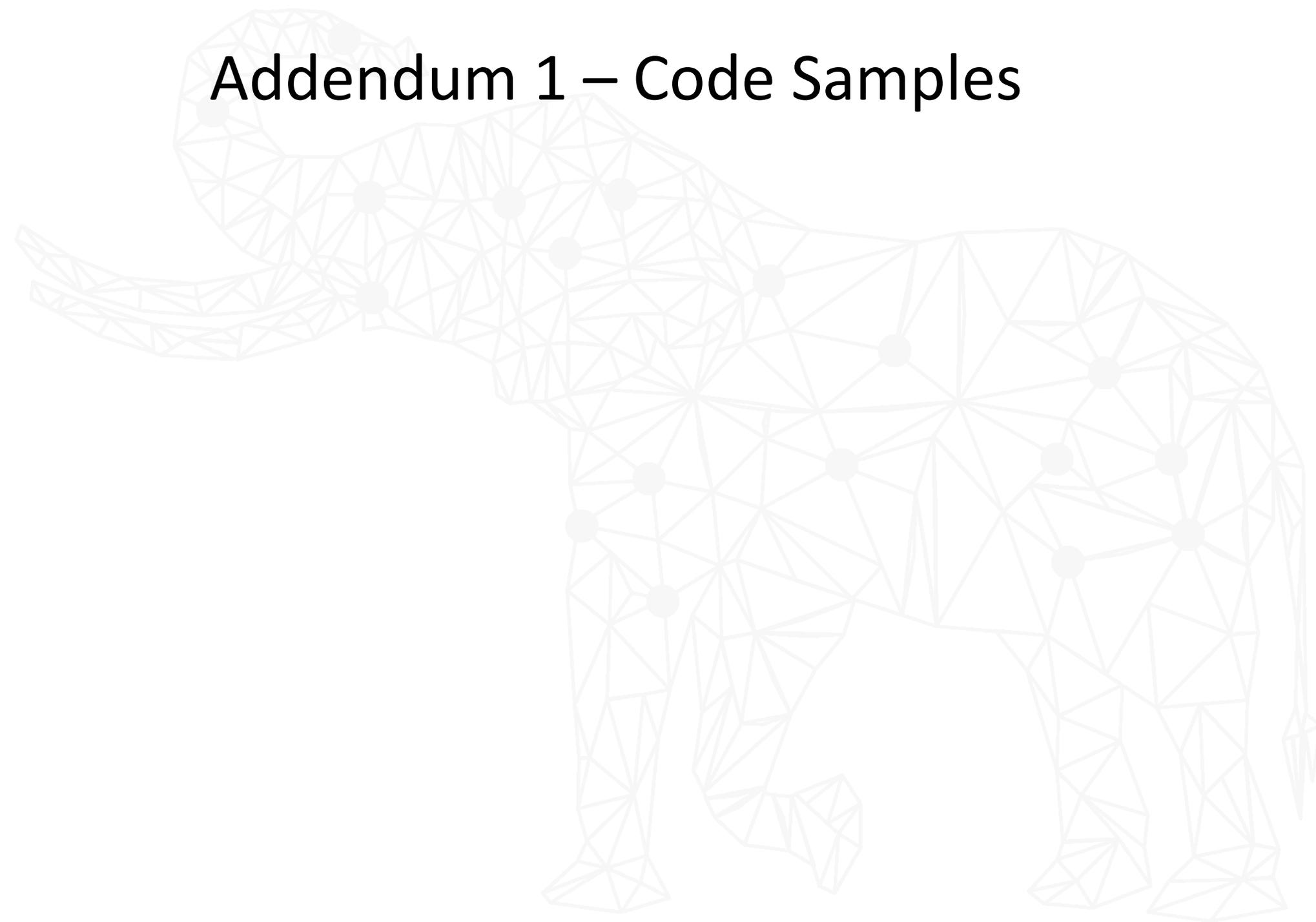
Nikita Glukhov



Oleg Bartunov



# Addendum 1 – Code Samples



# Pre-Defined Functions Header

```
typedef void (*toast_init)(Relation rel, Datum reloptions, LOCKMODE lockmode,
                          bool check, Oid OIDOldToast);
typedef struct varlena* (*toast_function) (Relation toast_rel,
                                          Oid toasterid,
                                          Datum value,
                                          Datum oldvalue,
                                          int max_inline_size,
                                          int options);
typedef struct varlena (*update_toast_function) (Relation toast_rel,
                                                Oid toasterid,
                                                Datum newvalue,
                                                Datum oldvalue,
                                                int options);
typedef struct varlena (*copy_toast_function) (Relation toast_rel,
                                              Oid toasterid,
                                              Datum newvalue,
                                              int options);

typedef struct varlena* (*detoast_function) (Datum toast_ptr,
                                           int offset, int length);
typedef void (*del_toast_function) (Datum value, bool is_speculative);
typedef void * (*get_vtable_function) (Datum toast_ptr);
typedef bool (*toast_validate_function) (Oid typeoid,
                                        char storage, char compression,
                                        Oid amoid, bool false_ok);
```

# Default Toaster Implementation

```
Datum
default_toaster_handler(PG_FUNCTION_ARGS)
{
    TsrRoutine *tsrroutine = makeNode(TsrRoutine);

    tsrroutine->init = genericToastInit;
    tsrroutine->toast = genericToast;
    tsrroutine->detoast = genericDetoast;
    tsrroutine->deltoast = genericDeleteToast;
    tsrroutine->get_vtable = NULL;
    tsrroutine->copy_toast = NULL;
    tsrroutine->update_toast = NULL;
    tsrroutine->toastervalidate = genericValidate;

    PG_RETURN_POINTER(tsrroutine);
}
```

# Toaster API with Virtual Table Example

```
static Datum
bytea_toaster_append(Datum d1, Datum d2)
{
    ...
}
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;
}
PG_FUNCTION_INFO_V1(bytea_toaster_handler);
Datum
bytea_toaster_handler(PG_FUNCTION_ARGS)
{
    TsrRoutine *tsr = makeNode(TsrRoutine);
    tsr->init = bytea_toaster_init;
    tsr->toast = bytea_toaster_toast;
    tsr->deltoast = bytea_toaster_delete_toast;
    tsr->copy_toast = bytea_toaster_copy_toast;
    tsr->update_toast = bytea_toaster_update_toast;
    tsr->detoast = bytea_toaster_detoast;
    tsr->toastvalidate = bytea_toaster_validate;
    tsr->get_vtable = bytea_toaster_vtable;
    PG_RETURN_POINTER(tsr);
}
```

# Custom Toast Pointer Code

```
/* TOAST pointers are a subset of varattrib_1b with an identifying tag byte */
typedef struct
{
    uint8      va_header;      /* Always 0x80 or 0x01 */
    uint8      va_tag;        /* Type of datum */
    char       va_data[FLEXIBLE_ARRAY_MEMBER]; /* Type-specific data */
} varattrib_1b_e;
/* varatt_custom is a subset of varattrib_1b_e */
typedef struct varatt_custom
{
    uint16     va_toasterdatalen; /* total size of toast pointer, < BLCKSZ */
    uint32align16 va_rawsize;     /* Original data size (includes header) */
    uint32align16 va_toasterid;   /* Toaster ID, actually Oid */
    char       va_toasterdata[FLEXIBLE_ARRAY_MEMBER]; /* Custom toaster data */
} varatt_custom;
#define VARDATA_1B_E(PTR)      (((varattrib_1b_e *) (PTR))->va_data)
#define VARATT_IS_EXTERNAL(PTR) VARATT_IS_1B_E(PTR)
#define VARATT_IS_CUSTOM(PTR) \
    (VARATT_IS_EXTERNAL(PTR) && VARTAG_EXTERNAL(PTR) == VARTAG_CUSTOM)
```