Pro JSONB
на стероидах

Oleg Bartunov
Nikita Glukhov

Research scientist @ Moscow University
CEO Postgres Professional
Major PostgreSQL contributor

Since Postgres95
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
NOSQL POSTGRES IN SHORT

<table>
<thead>
<tr>
<th>JSONB - 2014</th>
<th>● Binary storage</th>
<th>● Nesting objects &amp; arrays</th>
<th>● Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSTORE - 2003</td>
<td>● Perl-like hash storage</td>
<td>● No nesting, no arrays</td>
<td>● Indexing</td>
</tr>
<tr>
<td>SQL/JSON — 202?</td>
<td>● Complete SQL/JSON</td>
<td>● Better indexing, syntax</td>
<td></td>
</tr>
<tr>
<td>JSONPATH - 2019</td>
<td>● SQL/JSON — 2016</td>
<td>● Indexing</td>
<td></td>
</tr>
<tr>
<td>JSON - 2012</td>
<td>● Textual storage</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>● JSON validation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Postgres revolution: embracing relational databases

• NoSQL users attracted by the NoSQL Postgres features
Json in PostgreSQL
(state of Art)
Two JSON data types !!!

2012

JSON

2014

JSONB
BINARY BETTER JSON
Jsonb vs Json

SELECT j::json AS json, j::jsonb AS jsonb FROM
(SELECT '{"cc":0, "aa": 2, "aa":1,"b":1}' AS j) AS foo;

<table>
<thead>
<tr>
<th>json</th>
<th>jsonb</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;cc&quot;:0, &quot;aa&quot;: 2, &quot;aa&quot;:1,&quot;b&quot;:1}</td>
<td>{&quot;b&quot;: 1, &quot;aa&quot;: 1, &quot;cc&quot;: 0}</td>
</tr>
</tbody>
</table>

- json: textual storage «as is», parsed many
- jsonb: binary storage, parsed once, great performance (indexing)
- jsonb: no whitespaces, no duplicated keys (last key win)
- jsonb: keys are sorted by (length, key)
- jsonb: a rich set of functions (\df jsonb*), "arrow" operators, FTS
- JsQuery ext. - json query language with GIN indexing support
SQL/Foundation recognized JSON after the success of Postgres


4.46 JSON data handling in SQL ................................................................. 174
4.46.1 Introduction .................................................................................. 174
4.46.2 Implied JSON data model ............................................................. 175
4.46.3 SQL/JSON data model ................................................................. 176
4.46.4 SQL/JSON functions ................................................................. 177
4.46.5 Overview of SQL/JSON path language ........................................ 178

5 Lexical elements .............................................................................. 181
5.1 <SQL terminal character> .............................................................. 181
5.2 <token> and <separator> ............................................................... 185
SQL/JSON in SQL-2016

- SQL/JSON data model
  - A sequence of SQL/JSON items, each item can be (recursively) any of:
    - SQL/JSON scalar — non-null value of SQL types: Unicode character string, numeric, Boolean or datetime
    - SQL/JSON null, value that is distinct from any value of any SQL type (not the same as NULL)
    - SQL/JSON arrays, ordered list of zero or more SQL/JSON items — SQL/JSON elements
    - SQL/JSON objects — unordered collections of zero or more SQL/JSON members (key, SQL/JSON item)

- JSON Path language
  - Describes a <projection> of JSON data to be used by SQL/JSON functions

- SQL/JSON functions (9)
  - Construction functions: values of SQL types to JSON values
  - Query functions: JSON values to SQL types
    JSON Path(JSON values) → SQL/JSON types -> converted to SQL types
SQL/JSON in PostgreSQL

- SQL/JSON data model
  - *jsonb* is the (practical) subset of SQL/JSON data model
    ORDERED and UNIQUE KEYS

- JSON Path language
  - Describes a <projection> of JSON data (to be used by SQL/JSON functions)
  - Most important part of SQL/JSON - committed to PG12, PG13 (15/15 features)!

- SQL/JSON functions - waiting for review (v55, v48)
  - Constructor functions: *json[b]* construction functions
  - Query functions: functions/operators with jsonpath support

- Indexes
  - Use already existing indexes (built-in, jquery)
    Added jsonpath support
JSONB Projects: What we were working on

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

State of PostgreSQL 2021 (Survey)

Psql telegram (6170) — 26.02.2021

- SELECT 8061/312083
- SQL 4473/144789
- JSON[B] 3116/88234
- TABLE 2997/129936
- JOIN 2345/108860
- INDEX 1519/74327
- BACKUP 1484/42618
- VACUUM 1470/53919
- REPLICA 707/31036
Top-priority: JSONB - 1st-class citizen in Postgres

• Popularity of JSONB — it’s mature data type, rich functionality

• Startups use Postgres and don’t care about compatibility to Oracle/MS SQL
  • Jsonpath is important and committed
  • There is rich user API to Jsonb, so SQL/JSON functions are not in top-priority list

• Not enough resources in community (developers, reviewers, committers)
  • SQL/JSON — 4 years, 55 versions
  • JSON/Table — 48 versions

• 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
Popular mistake: CREATE TABLE qq (jsonb)

(id, {…}::jsonb) vs (id, {…}::jsonb)

Large jsonb is TOASTed!
Reality: Unpredictable performance of jsonb

Small update cause 10 times slowdown!

```sql
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"}';

=#! 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=32548
Planning Time: 0.044 ms
Execution Time: 66.889 ms
(4 rows)
```

What is happened? Row gets TOASTed!
TOAST Explained
The Oversized-Attribute Storage Technique

• TOASTed value is pglz compressed
• Compressed value is 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

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

- TOAST pointers do not refer to heap tuples with chunks directly. Instead, they contain Oid chunk_id and we need to descent by index (chunk_id, chunk_seq).

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

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

- Only "extended" and "external" attributes (that were not TOASTed in the previous pass) are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.
TOAST pass #3

• Only "main" attributes are considered.
• Each attribute is compressed, until the resulting tuple size < 2KB.
TOAST pass #4

- Only "main" attributes are considered.
- Each attribute is TOASTed, until the resulting tuple size < 2KB.
Motivational example (synthetic test)

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

```sql
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:

```sql
SELECT jb -> 'keyN', jb -> 'keyN', ... jb -> 'keyN' FROM test_toast WHERE id = ?;
```
Motivational example (synthetic test)

Key access time for TOASTed jsonbs linearly increase with jsonb size, regardless of key size and position.
TOAST performance problems (synthetic test)

Key access time for TOASTed jsonbs linearly increase with jsonb size, regardless of key size and position.
Motivational example (IMDB test)

- Real-world JSON data extracted from IMDB database (imdb-22-04-2018-json.dump.gz)
- Typical IMDB «name» document looks like:
  ```json
  {
    "id": "Connors, Steve (V)",
    "roles": [     
      {
        "role": "actor",
        "title": "Copperhead Creek (????)"
      },    
      {
        "role": "actor",
        "title": "Ride the Wanted Trail (????)+"
      }
    ],
    "imdb_id": 1234567
  }
  ```
- There are many other infrequent fields, but only id, imdb_id are mandatory, and roles array is the **biggest** and most frequent (see next slide).
IMDB data set field statistics
Motivational example (IMDB test)

```sql
SELECT jb -> 'key' FROM imdb.names;
```
Motivation

• Decompression is the biggest problem. Big overhead of decompression of the whole jsonb limits the applicability of jsonb as document storage with partial access.
  • Need partial decompression

• Toast introduces additional overhead - read too many block
  • Read only necessary blocks — partial detoast
Jsonb deTOAST improvements

• Partial pglz decompression
  • Decompress only necessary part of jsonb.

• Sort jsonb object key by their length
  • Move long keys to the end, let short keys stay inlined.

• Partial deTOASTing using TOAST iterators
  • Detoast only necessary TOAST chunks

• Inline TOAST
  • Use inline storage to store part of the 1st chunk, let short keys stay inlined

• Shared TOAST
  • Share non-modified TOAST chunks between versions (partial update), save storage size and WAL traffic.
Partial decompression eliminates overhead of pglz decompression of the whole jsonb.

Jsonb is decompressed step by step: header, KV entries array, key name and key value. Only prefix of jsonb has to be decompressed to access a given key!
Jsonb partial decompression results (synthetic)

Access to key1 (red) in the prefix of jsonb was significantly improved:
• For inline compressed jsonb access time becomes constant
• For jsonb > 1MB acceleration is of order(s) of magnitude.
Jsonb partial decompression results (IMDB)

- Access to the first key «id» and rare key «height» was significantly improved.
- Access time to big key «roles» and short «imdb_id» remains mostly unchanged.

```
SELECT jb -> 'key' FROM imdb.names;
```
Sorting jsonb keys by length

In the original jsonb format object keys are sorted by (length,name), so the short keys with longer or alphabetically greater names are placed at the end and cannot benefit from the partial decompression. Sorting by length allows fast decompressions of the shortest keys (metadata).

original: keys names and values sorted by key names

new: keys values sorted by their length
JSONB Binary Format (src/include/utils/jsonb.h)

ORIGINAL: VALUES SORTED BY KEYS

VALUES SORTED BY THEIR LENGTH
Sorting jsonb keys by length results (synthetic)

Access time to the all short keys and medium-length key4 (excluding long key2, placed now at the end of jsonb) was significantly speed up:
Sorting jsonb keys by length results (IMDB)

- Access to the last short key «imdb_id» now also was speed up.
- There is a big difference in access time (~5x) between inline and TOASTed values.
Partial deTOASTing

• We used patch «de-TOAST'ing using a iterator» from the CommitFest. It was originally developed by Binguo Bao at GSOC 2019.

• This patch gives ability to deTOAST and decompress chunk by chunk. So if we need only the jsonb header and first keys from the first chunk, only that first chunk will be read (actually, some index blocks also will be read).

• We modified patch adding ability to decompress only the needed prefix of TOAST chunks.
Partial deTOASTing results (synthetic)

Partial deTOASTing speeds up only access to the short keys of long jsonbs, making access time almost independent of jsonb size.
Partial deTOASTing results (IMDB)

• Results are the same, but not so noticeable because there are not many big (> 100KB) jsonbs.
• A big gap in access time (~5x) between inline and TOASTed values is still there.
Partial deTOASTing results (IMDB)

- Effect of partial deTOASTing: Arrow operator (→) for short keys always read only 4 blocks (3 index and 1 heap).
Ininline TOAST

• Store first TOAST chunk containing jsonb header and possibly some short keys inline in the heap tuple.

• We added new typstorage «tapas», which is similar to «extended», except that it tries to fill the tuple to 2KB (if other attributes occupy less than 2KB) with the chunk cutted from the beginning of the compressed data.
Inline TOAST results (synthetic)

Partial inline TOAST completely removes gap in access time to short keys between long and mid-size jsonbs.
Inline TOAST results (IMDB)

• Results are the same as in synthetic test.
• There is some access time gap between compressed and non-compressed jsonbs.
Inline TOAST results (IMDB)

- Effect of inline TOAST: Arrow operator (→) for short keys read no additional blocks.
JSONB partial update

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

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

• TOAST storage is duplicated
• WAL traffic is increased in comparison with updates of non-TOASTED attributes, because the whole TOASTed values is logged
• Performance is too low
JSONB partial update: The problem

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

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

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

<table>
<thead>
<tr>
<th>heap_rel</th>
<th>heap_rel_size</th>
<th>toast_rel</th>
<th>toast_rel_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>512 kB</td>
<td>pg_toast.pg_toast_27227</td>
<td>78 MB</td>
</tr>
</tbody>
</table>

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;
```  

<table>
<thead>
<tr>
<th>compressed_size</th>
<th>original_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>6043</td>
<td>18904</td>
</tr>
</tbody>
</table>

```
SELECT chunk_id, count(chunk_seq) FROM pg_toast.pg_toast_47235 GROUP BY chunk_id LIMIT 1;
```  

<table>
<thead>
<tr>
<th>chunk_id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>57241</td>
<td>4</td>
</tr>
</tbody>
</table>
First, let's try to update of non-TOASTED int column id:

```sql
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 (was 512 kB) | pg_toast.pg_toast_47235 | 78 MB (not changed)
```
JSONB partial update: The problem

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

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

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

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

SELECT pg_size_pretty(pg_wal_lsn_diff('0/1DB10000','0/158E5B48')) AS wal_size;
               wal_size
----------     
   130 MB     (13 KB per row; was 1.5 MB, ~87x more)

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

heap_rel | heap_rel_size | toast_rel         | toast_rel_size
----------+---------------+-------------------+---------------
         | 1528 kB       | pg_toast.pg_toast_47235 | 156 MB
         | (was 1024 kB) | (was 78 MB, 2x more) |
```
Partial update using Shared TOAST

• The previous optimizations are great for SELECT, but don’t help with UPDATE, since TOAST consider jsonb as an atomic binary blob – change part, copy the whole.

• Idea: Keep INLINE short fields *(uncompressed)* and TOAST pointers to long fields to let update short fields without modification of TOAST chunks, which will be shared between versions.

• Currently, this works only for root objects fields, so the longest fields of jsonb object are TOASTed until the whole tuple fits into the page (typically, remaining size of jsonb becomes < ~2000 bytes).

• But this technique can also be applied to array elements or element ranges. We plan to try to implement it later, it needs more invasive jsonb API changes.

• Currently, jsonb hook is hardcoded into TOAST pass #1, but in the future it will become custom datatype TOASTer using pg_type.typtoast.
Shared TOAST – jsonb format extensions

- Added special “TOASTed container” JEntry type. JsonbContainer header is left inline, but the body is replaced with a pointer.
- Added “TOASTed object” JsonbContainer type to mark object with TOAST pointers.
- TOASTed subcontainers are stored as plain jsonb datums (varlena header added).
Shared TOAST – tuple structure

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

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

Gap in access time to short keys is completely removed. Access to mid-size fields is slow down, because they are TOASTed instead of stored inline (we need to fix this).
Shared TOAST – access results (IMDB)

- Results are the same as in synthetic test.
- All short keys is speed up as much as possible.
Shared TOAST – update results (synthetic)

• Update time of short keys does not depend on total jsonb size
• Update time of TOASTed fields depends only on their own size
Shared TOAST – update results (synthetic)

- WAL traffic due to update of short and mid-size keys is greatly decreased
Step-by-step results (synthetic)
Step-by-step results (IMDB)

```
SELECT jb -> 'key' FROM imdb.names;
```

<table>
<thead>
<tr>
<th>master</th>
<th>+part_decomp</th>
<th>+sorted_keys</th>
<th>+part_detoast</th>
<th>+inline_toast</th>
<th>+shared_toast</th>
</tr>
</thead>
</table>

- Execution time, µs
- Raw jsonb size, bytes

**Key Colors:**
- id
- roles
- height
- imdb_id
Popular mistake: CREATE TABLE qq (jsonb)

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

SELECT expr FROM test_toast;

raw jsonb size, bytes

number of chunks

master

+shared_toast

expr

id

jb->'id'

jb->'a'->'d'

PostgresPro
Popular mistake: CREATE TABLE qq (jsonb)

(id, {...}::jsonb) vs ({id,...}::jsonb)
Appendable bytea: Motivational example

• A table with 100 MB bytea (uncompressed):

```sql
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:

```sql
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.

• Thanks to Alexander ? who raised the problem of (non-effective) streaming into bytea at PGConf.Online !
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
Appendable bytea: 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
Appendable bytea: 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 (0,1) are shared.

Benefit: 7 chunks vs 14 (master)
Results – motivational example

• 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 (was 12665)
    -> 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
Appendable bytea: append to bytea (time)

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

exec time, ns

append size, bytes

10
100
1K
10K
100K
1M

data size (old size), bytes

OLD + NEW

APPEND SIZE
Appendable bytea: append to bytea (WAL)
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 WHERE id = 0; 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;

![Graph showing execution time vs. data size for different append sizes.](image-url)
Appendable bytea: stream (WAL)
Appendable bytea: stream (throughput MB/s)
Non-scientific comparison PG vs Mongo

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

• We demonstrated step-by-step performance improvements (with backward compatibility), which lead to significant (10X) speedup for SELECTs and much cheaper UPDATEs (OLTP Jsonb?)
  • Github: https://github.com/postgrespro/postgres/tree/jsonb_shared_toast

• The same technique can be applied to any data types with random access to parts of data (arrays, hstore, movie, pdf ...)

• Appendable bytea: 1000X performance improvements
  • https://github.com/postgrespro/postgres/tree/bytea_appendable_toast

• Jsonb is ubiquitous and is constantly developing
  • JSON[B] Roadmap V2, Postgres Professional Webinar, Sep 17, 2020
  • JSON[B] Roadmap V3, Postgres Build 2020, Dec 8, 2020
TODO (OLTP JSONB, OLAP JSONB)

- More benchmarks (YCSB, use cases), PG vs Mongo
- Extend shared TOAST to support strings, arrays, jsonb arrays
- Prepend, truncate, insert, delete for appendable bytea
- Pluggable TOAST
- How to integrate this new stuff into the CORE?
- WiredElephant – storage (non-TOASTED) for tree-like structures with big attributes?
Нам нужны Ваши кейсы (тестовые данные и запросы)!

Contact obartunov@postgrespro.ru, n.gluhov@postgrespro.ru for collaboration