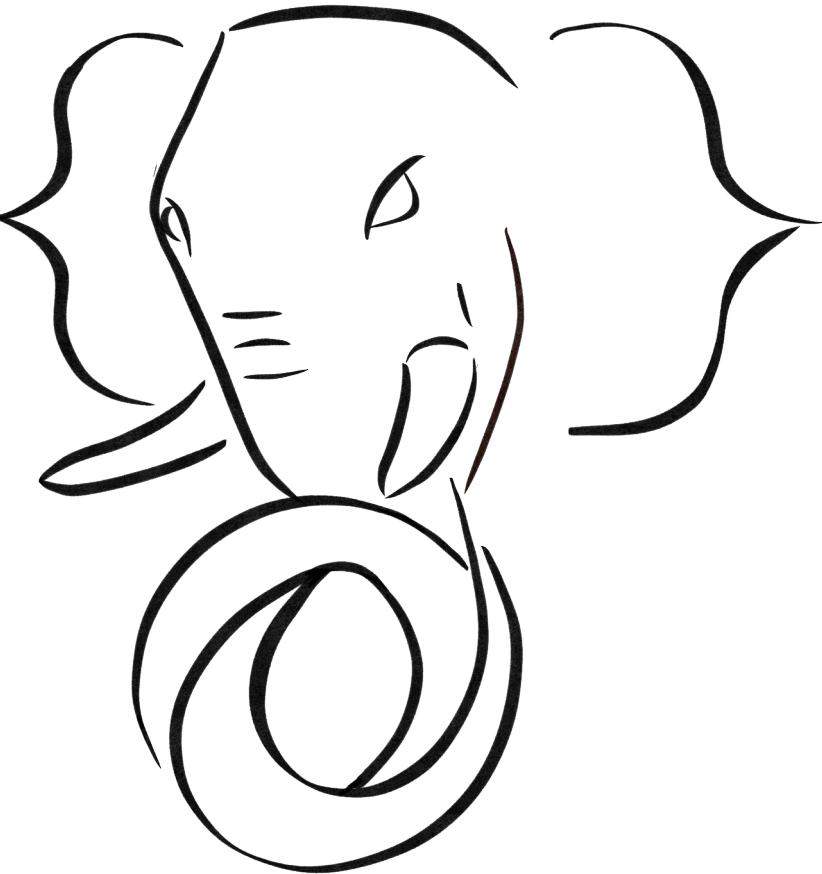




JSON[b] Roadmap



Oleg Bartunov, Postgres Professional

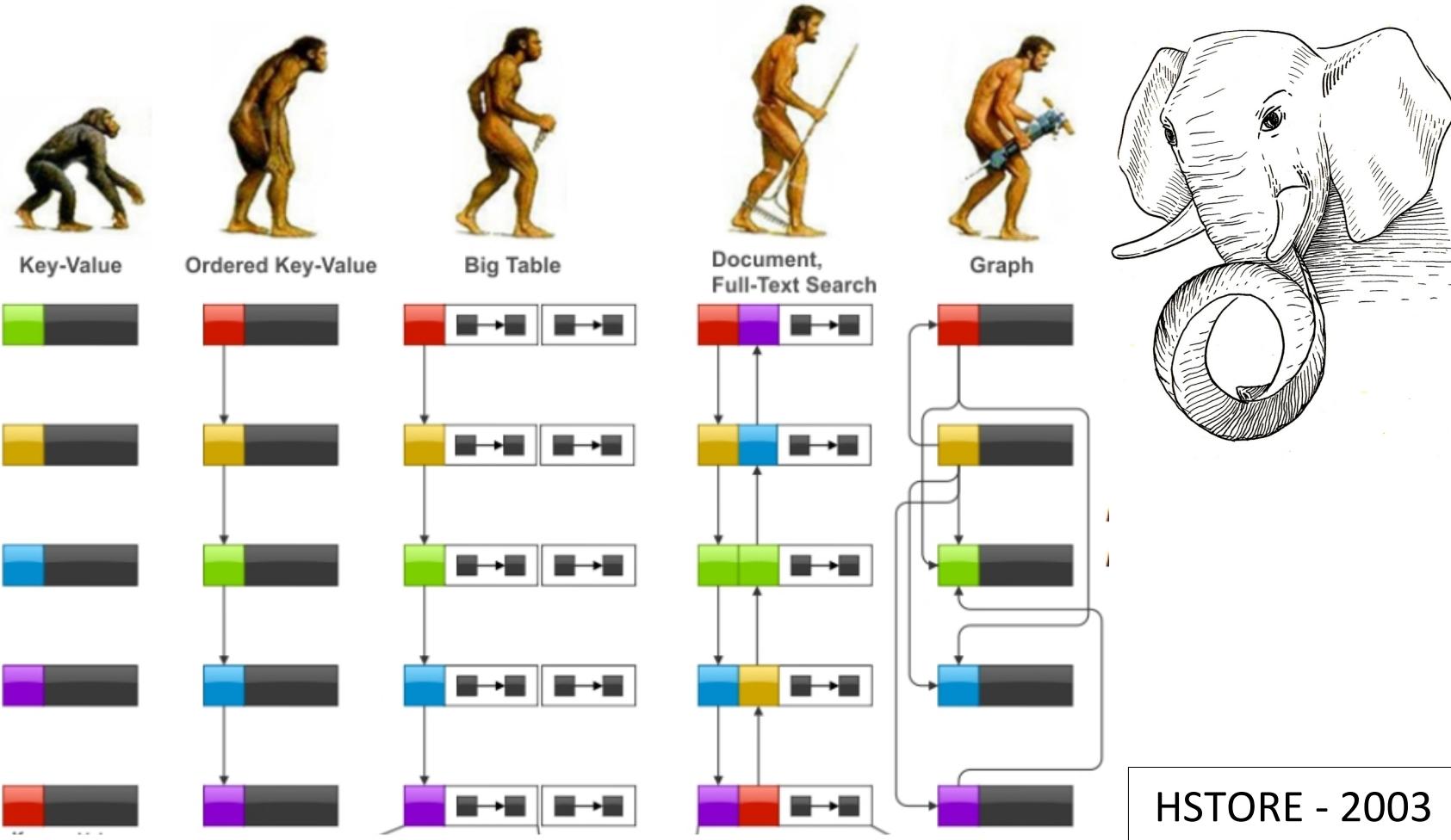
Since Postgres95



Research scientist @
Moscow University
CEO PostgreSQL Professional

NOSQL POSTGRES IN SHORT

?



HSTORE - 2003

- Perl-like hash storage
- No nesting, no arrays
- Indexing

SQL/JSON — 202?

- Complete SQL/JSON
- Better indexing, syntax

JSONPATH - 2019

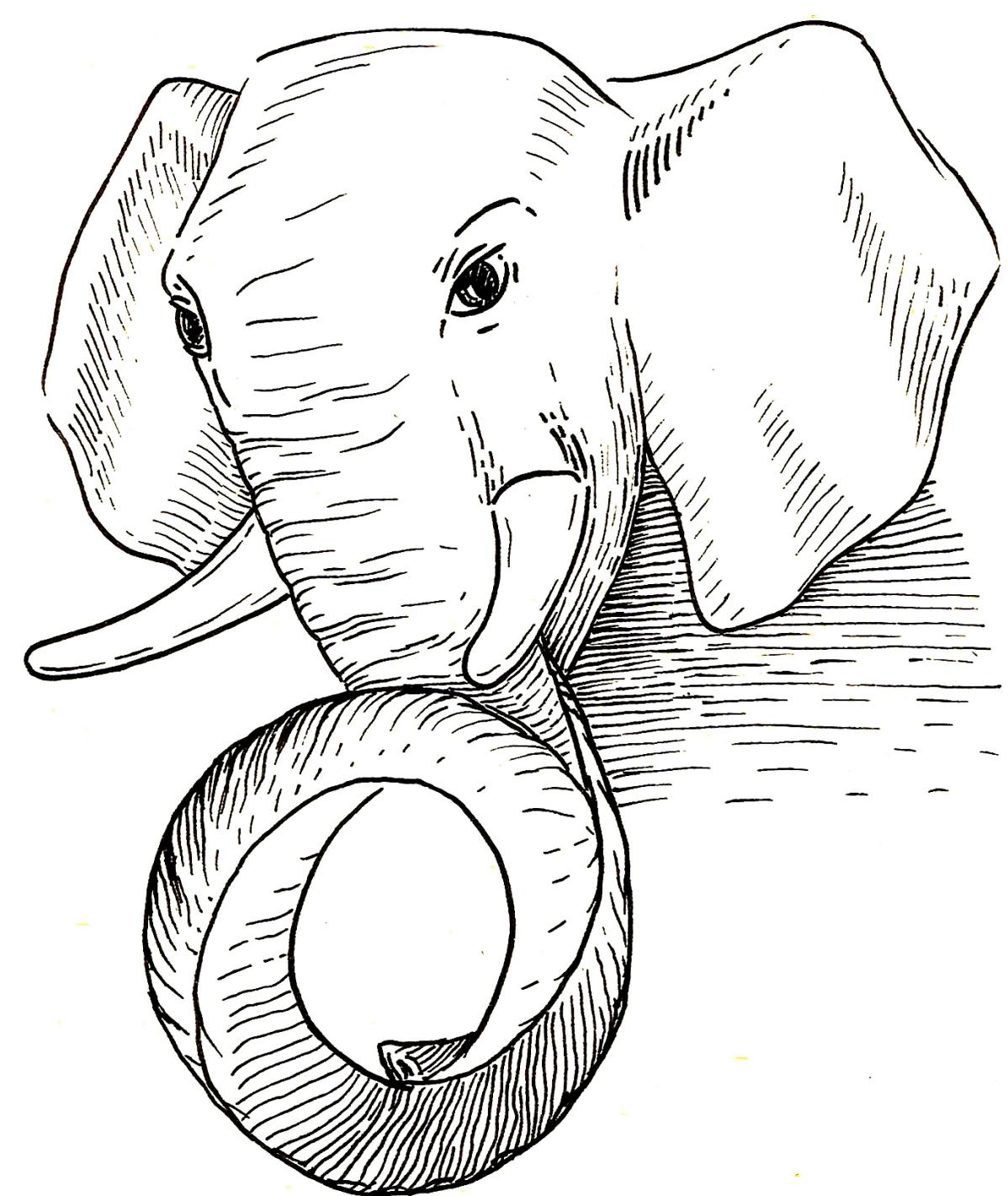
- SQL/JSON — 2016
- Indexing

JSONB - 2014

- Binary storage
- Nesting objects & arrays
- Indexing

JSON - 2012

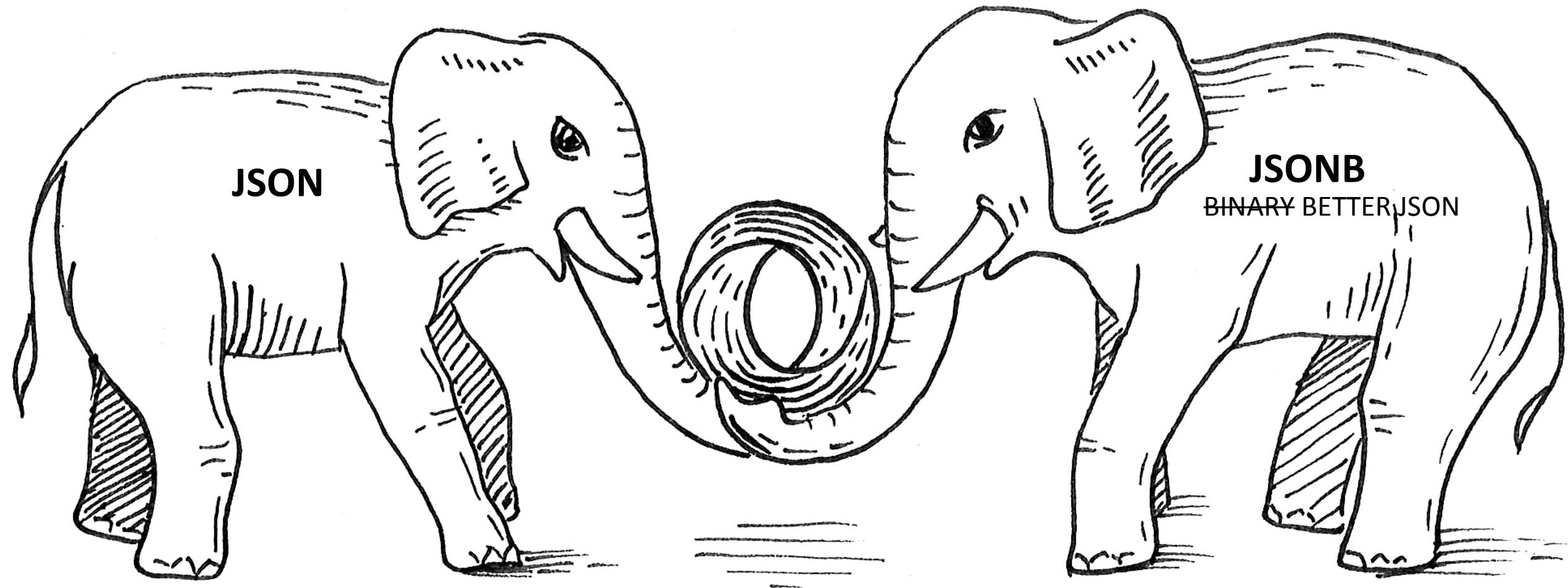
- Textual storage
- JSON validation



Json in PostgreSQL

(state of Art)

Two JSON data types !!!



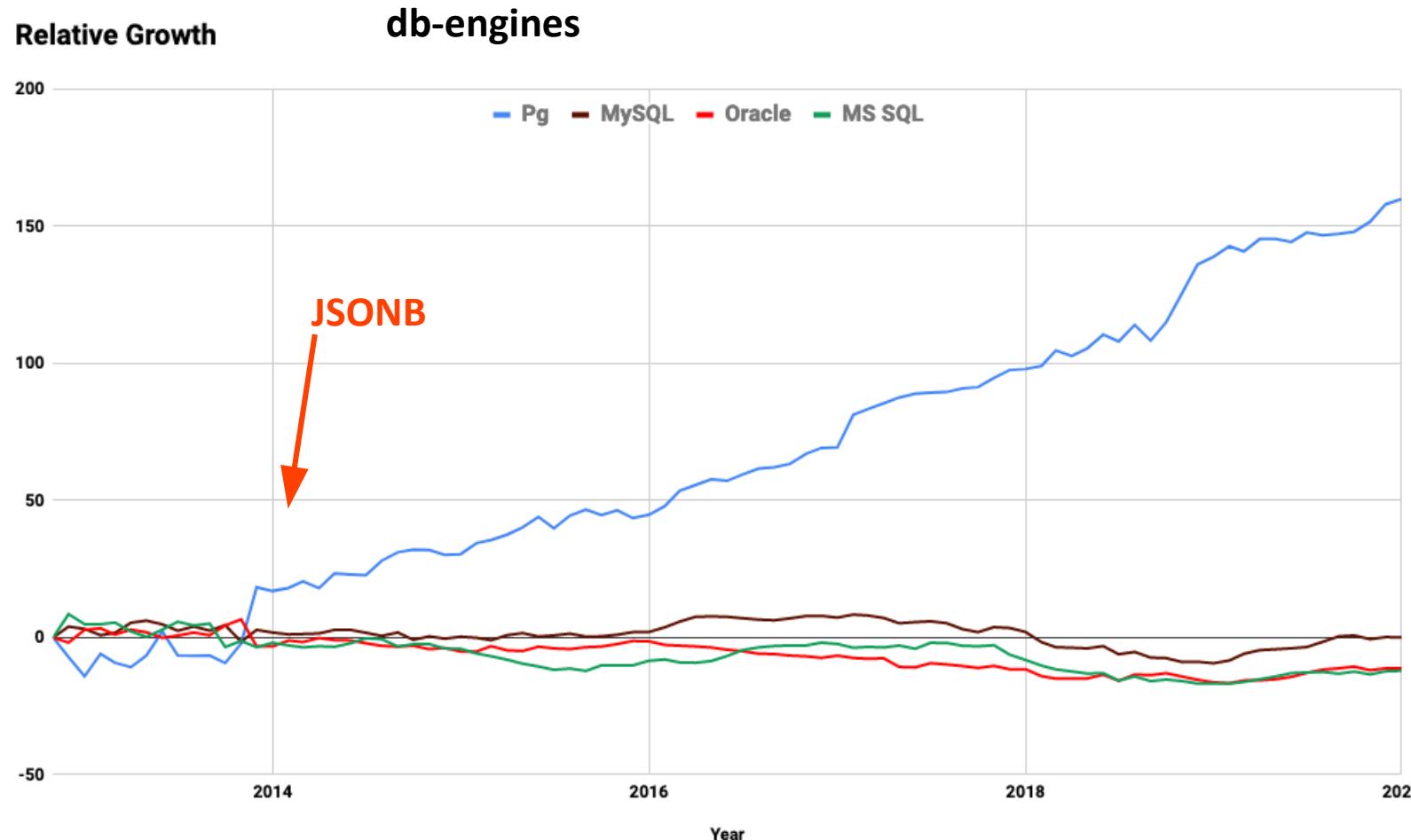
Jsonb vs Json

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

- 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

Postgres revolution: embracing relational databases

- NoSQL users attracted by the NoSQL Postgres features



Dec 18, 2014

SQL/Foundation recognized JSON after the success of Postgres

SQL:2016 — 22 JSON features out of 44 new optional.

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

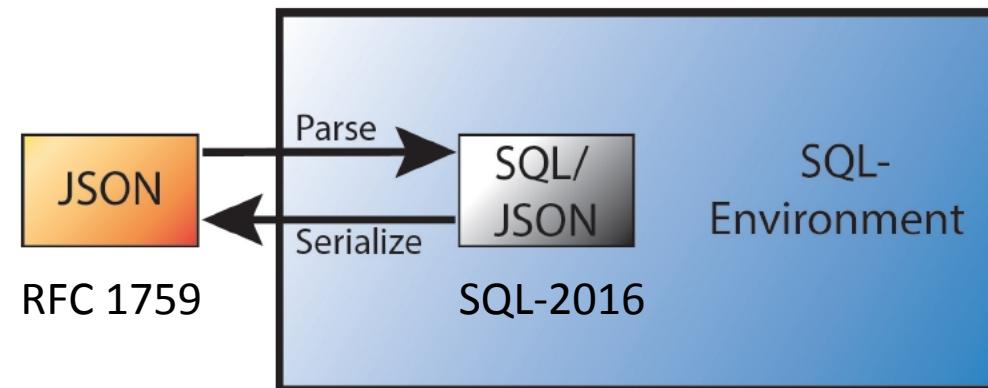
- A *sequence of SQL/JSON items*, each item can be (recursively) any of:
 - SQL/JSON scalar — non-null value of SQL types: Unicode character string, numeric, Boolean or datetime
 - SQL/JSON *null*, value that is distinct from any value of any SQL type (not the same as NULL)
 - SQL/JSON arrays, ordered 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

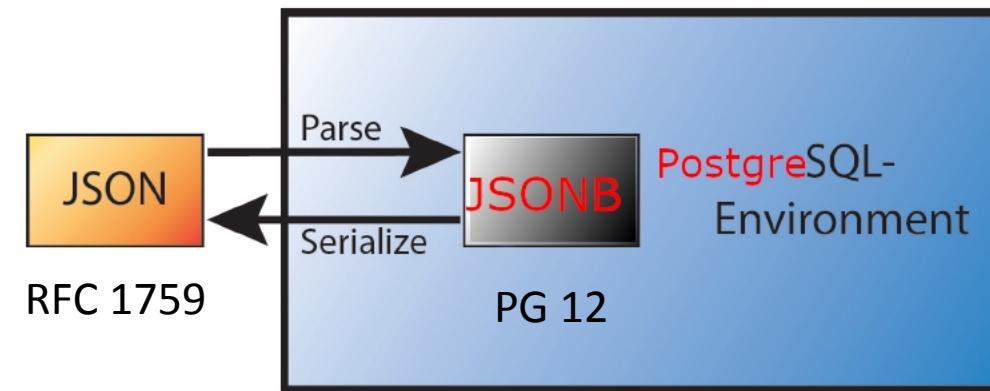
- SQL/JSON functions (9)

- 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 !**
- SQL/JSON functions
 - Constructor functions: **json[b] construction functions**
 - Query functions: **need some functions/operators with jsonpath support**
- Indexes
 - **Use already existing indexes (built-in, jsquery)**
 - **Add support of jsonpath to the existing opclasses**



JSON Path query language

- **JSON Path** expression specify the parts of json. It is an optional path mode 'strict' or 'lax' (default), followed by a *path* or unary/binary expression on *paths*. *Path* is a sequence of path elements, started from path variable, path literal or expression in parentheses and zero or more operators (JSON accessors, filters, and item methods)

```
'lax $.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'
```

- Dot notation used for member access – '\$.a.b.c'
- \$ - the current context element
- [*], [0 to LAST] – array access (starts from zero!)
- Filter(s) - '\$.a.b.c ? (@.x > 10)'
- @ - current context in filter expression
- Item methods - '\$.a.b.c.x.type()'
 - type(), size(), double(), ceiling(),
 - floor(), abs(), keyvalue(), datetime()

```
CREATE TABLE house(j) AS SELECT jsonb '  
{  
    "info": {  
        "contacts": "Postgres Professional\\n+7 (495) 150-06-91\\ninfo@postgrespro.ru",  
        "dates": ["01-02-2015", "04-10-1957 19:28:34 +00", "12-04-1961 09:07:00 +03"]  
    },  
    "address": {  
        "country": "Russia",  
        "city": "Moscow",  
        "street": "117036, Dmitriya Ulyanova, 7A"  
    },  
    "lift": false,  
    "floor": [  
        {  
            "level": 1,  
            "apt": [  
                {"no": 1, "area": 40, "rooms": 1},  
                {"no": 2, "area": 80, "rooms": 3},  
                {"no": 3, "area": null, "rooms": 2}  
            ]  
        },  
        {  
            "level": 2,  
            "apt": [  
                {"no": 4, "area": 100, "rooms": 3},  
                {"no": 5, "area": 60, "rooms": 2}  
            ]  
        }  
    ]  
};
```



Jsonpath queries are concise

- PG12+ (jsonpath) query

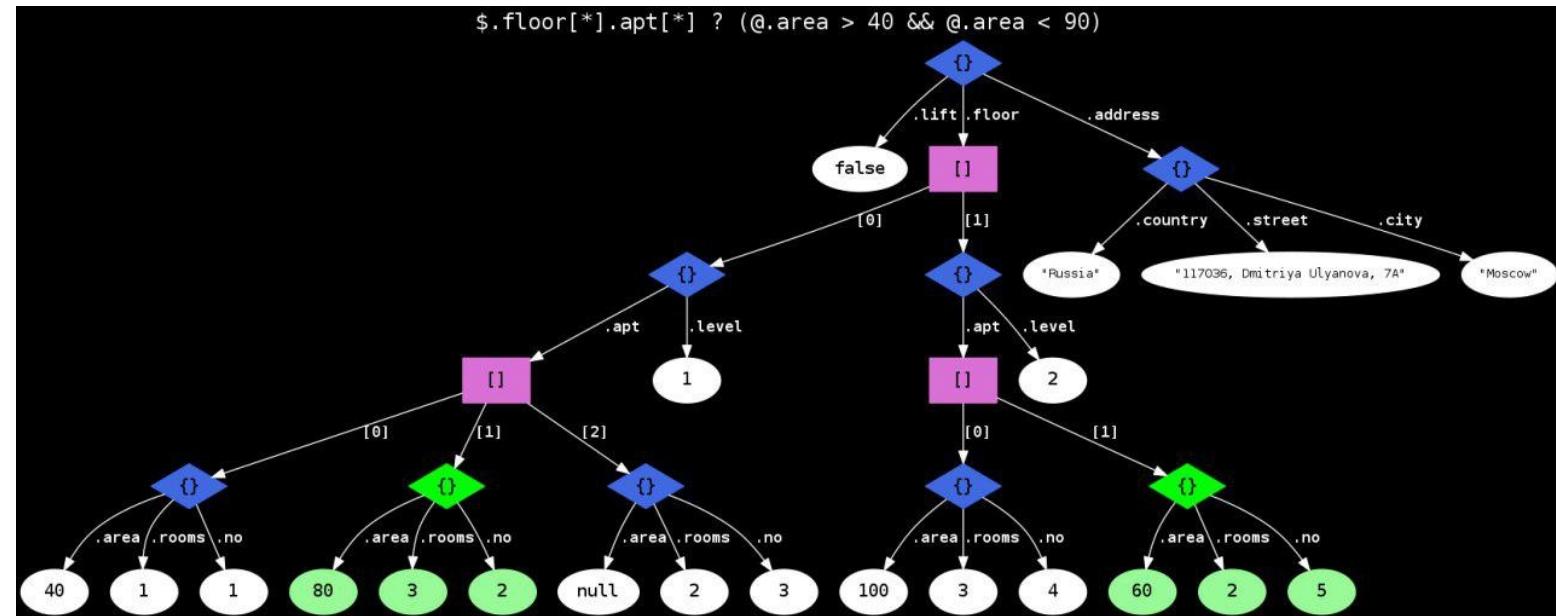
```
SELECT jsonb_path_query(js, '$.floor[*].apt[*] ?  
(@.area > 40 && @.area < 90)')  
FROM house;
```

- PG11 query

```
SELECT apt  
FROM (SELECT jsonb_array_elements(jsonb_array_elements(js->'floor')->'apt')  
      FROM house) apts(apt)  
WHERE (apt->>'area')::int > 40 AND (apt->>'area')::int < 90;
```

How path expression works

```
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'
```



- 1) \$ - json itself
- 2) .floor — an array floor
- 3) [*] – an array of two floors
- 4) .apt — two arrays of objects (apartments on each floor)
- 5) [*] - SQL/JSON seq. of length 5, extracts five objects (apartments)
- 6) Each apartment filtered by `(@.area > 40 && @.area < 90)` expression

The result is a sequence of two SQL/JSON items (green)

Why jsonpath at all ?

- It is **standard** query language for json
- It is **flexible** and **concise**, arrow operators are too primitive
 - Very useful for COPY and indexing json
- It is **faster** for complex processing

SQL/JSON standard conformance

SQL/JSON feature	PostgreSQL 13	Oracle 18c	MySQL 8.0.4	SQL Server 2017
JSON PATH: 15	15/15	11/15	5/15	2/15

PostgreSQL 13 has the best implementation of JSON Path

More information: JSONPATH@PGCONF.EU-2019

JSONB (jsonpath) indexing: built-in opclasses

Sample jsonb: {"k1": "v1", "k2": ["v2", "v3"]}

- **jsonb_ops** (default GIN opclass for jsonb) extracts keys and values
 - "k1", "k2", "v1", "v2", "v3"
 - Supports top-level key-exists operators ?, ?& and ?| , contains @> operator
 - Overlapping of large postings might be slow
- **jsonb_hash_ops** extracts hashes of paths:
 - hash("k1"."v1"), hash("k2".#."v2"), hash("k2".#."v3")
 - Supports only contains @> operator
 - Much faster and smaller than default opclass (for @>)
- Indexes provides jsonb performance on par with NoSQL database

Roadmap (ideas)

- Generic JSON API. The Grand Unification.
- Jsonb partial decompression
- SQL/JSON functions from SQL-2016 standard
- Planner support functions
- Jsonpath as parameter for jsonb opclasses
- Jsquery GIN opclasses to core
- Jsonpath syntax extension
- Simple Dot-Notation Access to JSON Data
- COPY (FORMAT json);

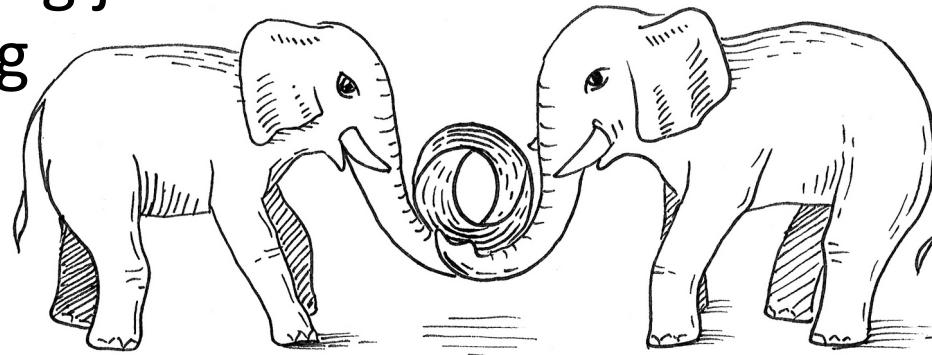
Generic JSON (GSON) API

The Grand Unification.



Why GSON ?

- JSON, JSONB – two implementations of users functions
- SQL/JSON currently supports only jsonb, adding json – even more overlapping



- Coming SQL standard (JSON 2.0) will specify JSON data type

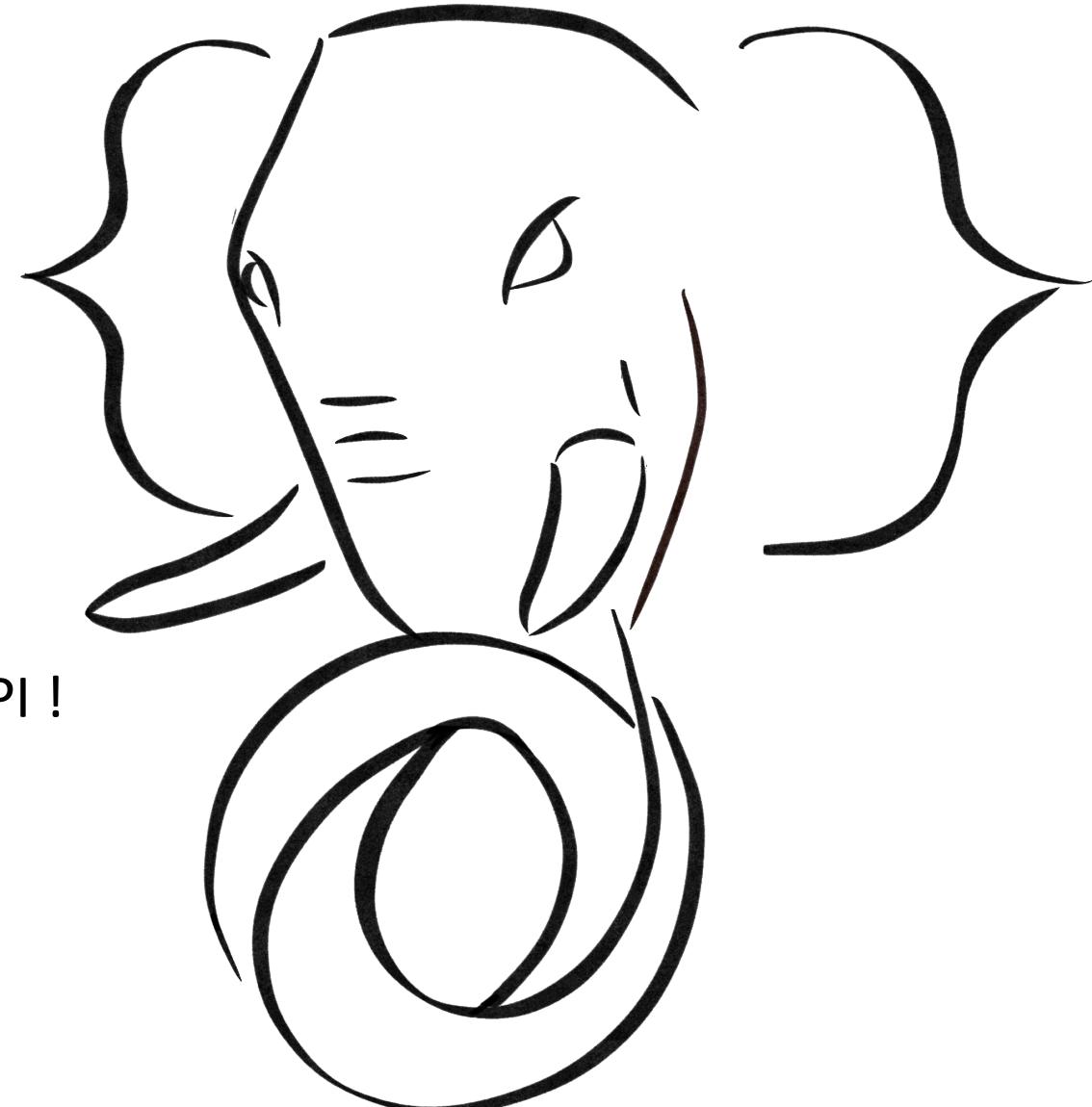
"The most important feature that is not specified in the standard [...] is the native support of JSON data type, the same way as the XML data type is implemented"

- Oracle 20c (preview) introduced JSON data type, which stored in binary format OSON

```
[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_array_element_text         json_object
json_array_elements             json_object_agg
json_array_elements_text        json_object_agg_finalfn
json_array_length               json_object_agg_transfn
json_build_array                json_object_field
json_build_object               json_object_field_text
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_array_element             jsonb_gt
jsonb_array_element_text        jsonb_hash
jsonb_array_elements            jsonb_hash_extended
jsonb_array_elements_text       jsonb_in
jsonb_array_length              jsonb_insert
jsonb_build_array               jsonb_le
jsonb_build_object              jsonb_lt
jsonb_cmp
jsonb_concat
jsonb_contained
jsonb_contains
jsonb_delete
jsonb_delete_path
jsonb_each
jsonb_each_text
jsonb_eq
jsonb_exists
jsonb_exists_all
jsonb_exists_any
[local]:5555 postgres@postgres# \df jsonb_
jsonb_path_match
jsonb_path_match_opr
jsonb_path_match_tz
jsonb_path_query
jsonb_path_query_array
jsonb_path_query_array_tz
jsonb_path_query_first
jsonb_path_query_first_tz
jsonb_path_query_tz
jsonb_populate_record
jsonb_populate_recordset
jsonb_pretty
jsonb_recv
jsonb_send
jsonb_set
jsonb_set_lax
jsonb_strip_nulls
jsonb_to_record
jsonb_to_recordset
jsonb_to_tsvector
jsonb_typeof
```

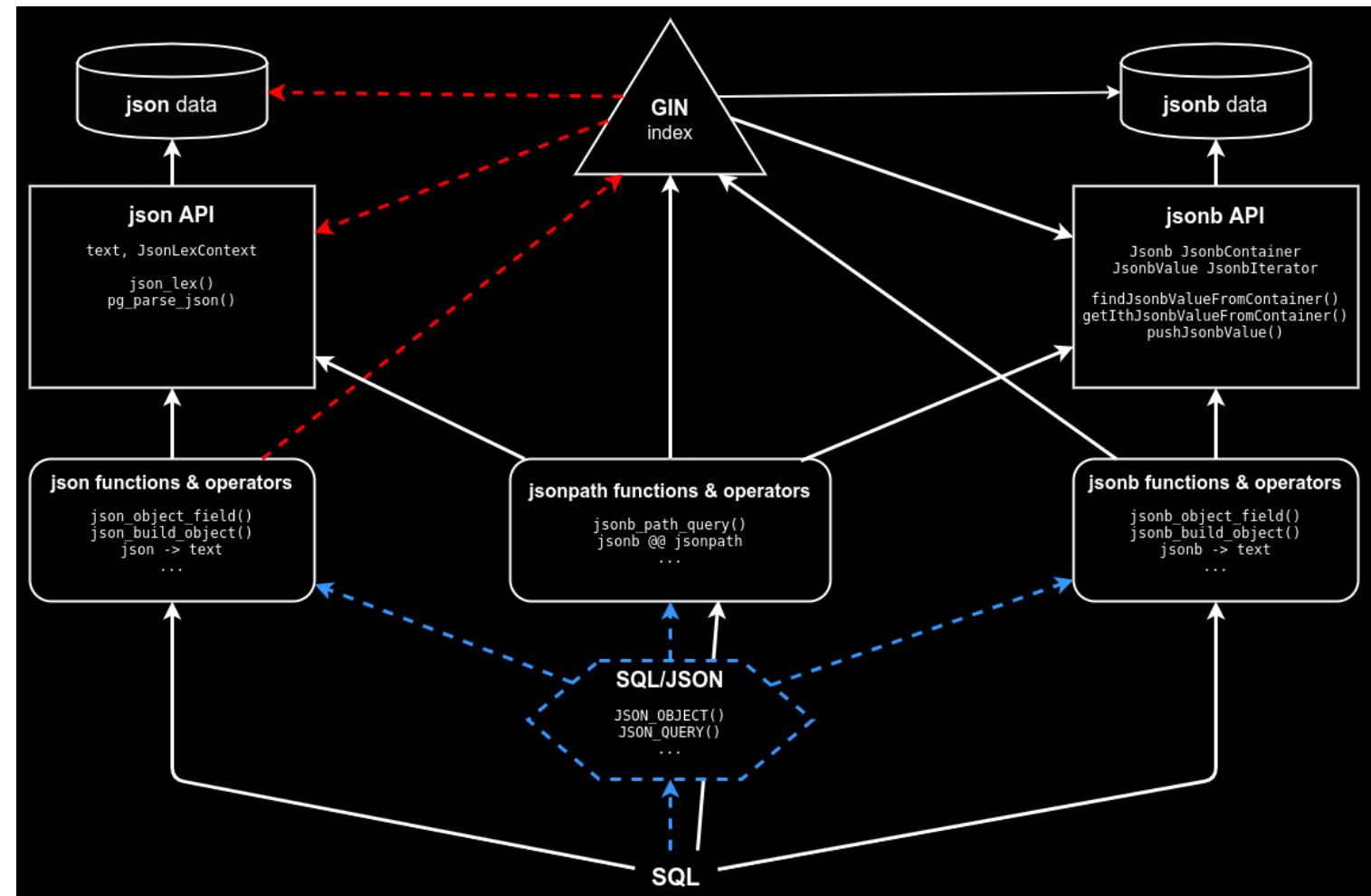
Why GSON ?

- For the sake of compatibility with SQL standard we need one JSON data type, which is internally
 - jsonb by default
 - Optionally behave as "old textual json"
 - Reduce code base
 - "Here comes The" idea of Generic JSON API !



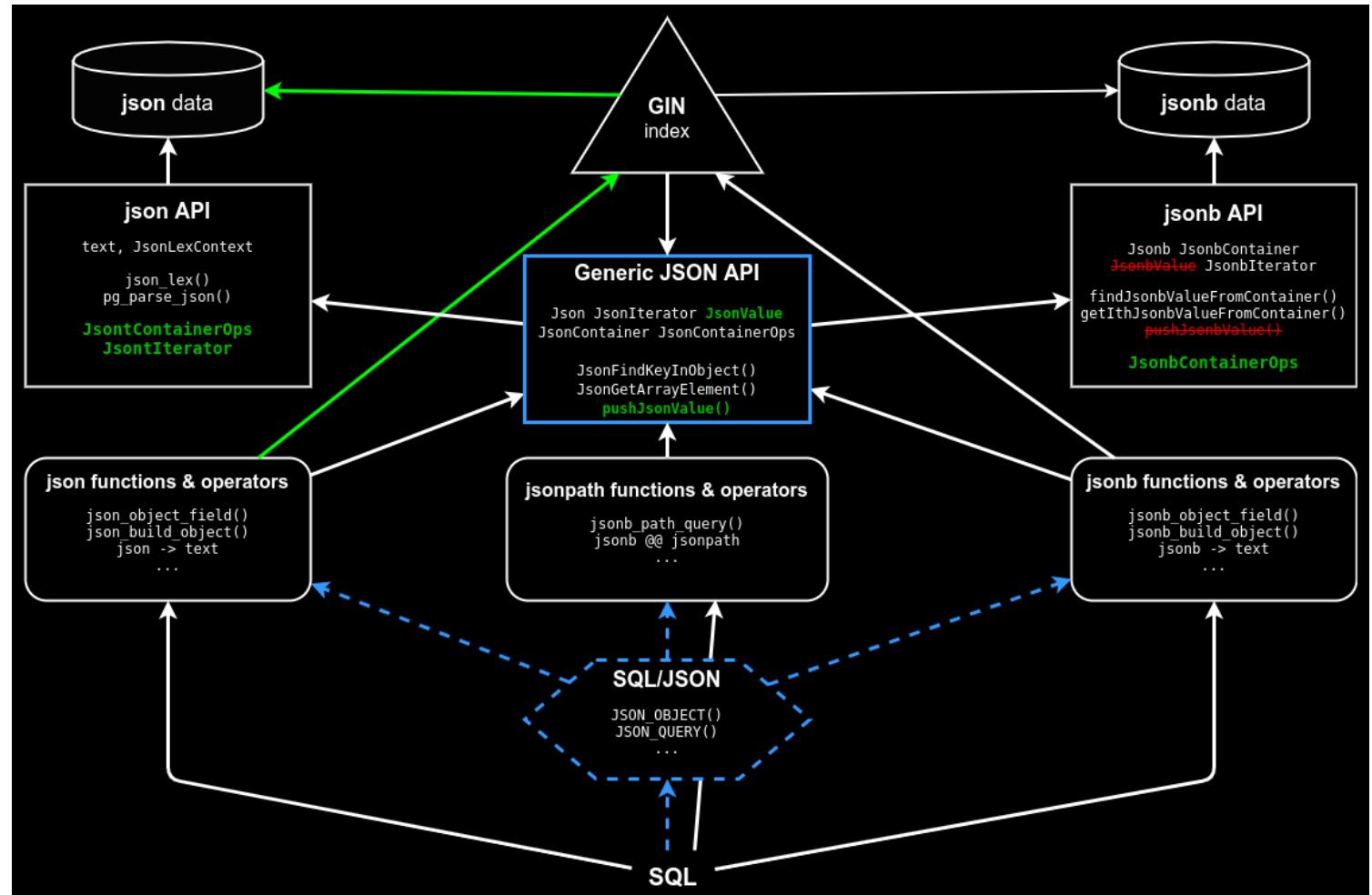
Generic JSON (GSON) – Current JSON[B] API

- Current JSON API is different for json and jsonb data types:
 - Json has lexer and parser with visitor interface
 - Jsonb uses Json lexer and parser for input, and several functions and iterators for access.
- This makes difficult to implement jsonpath functions for json (required by the SQL standard) and possibly GIN indexes for json.
- It is not easy to add new features like partial decompression/detoeasting or slicing, different storage formats (jsonbc, bson, oson,...).



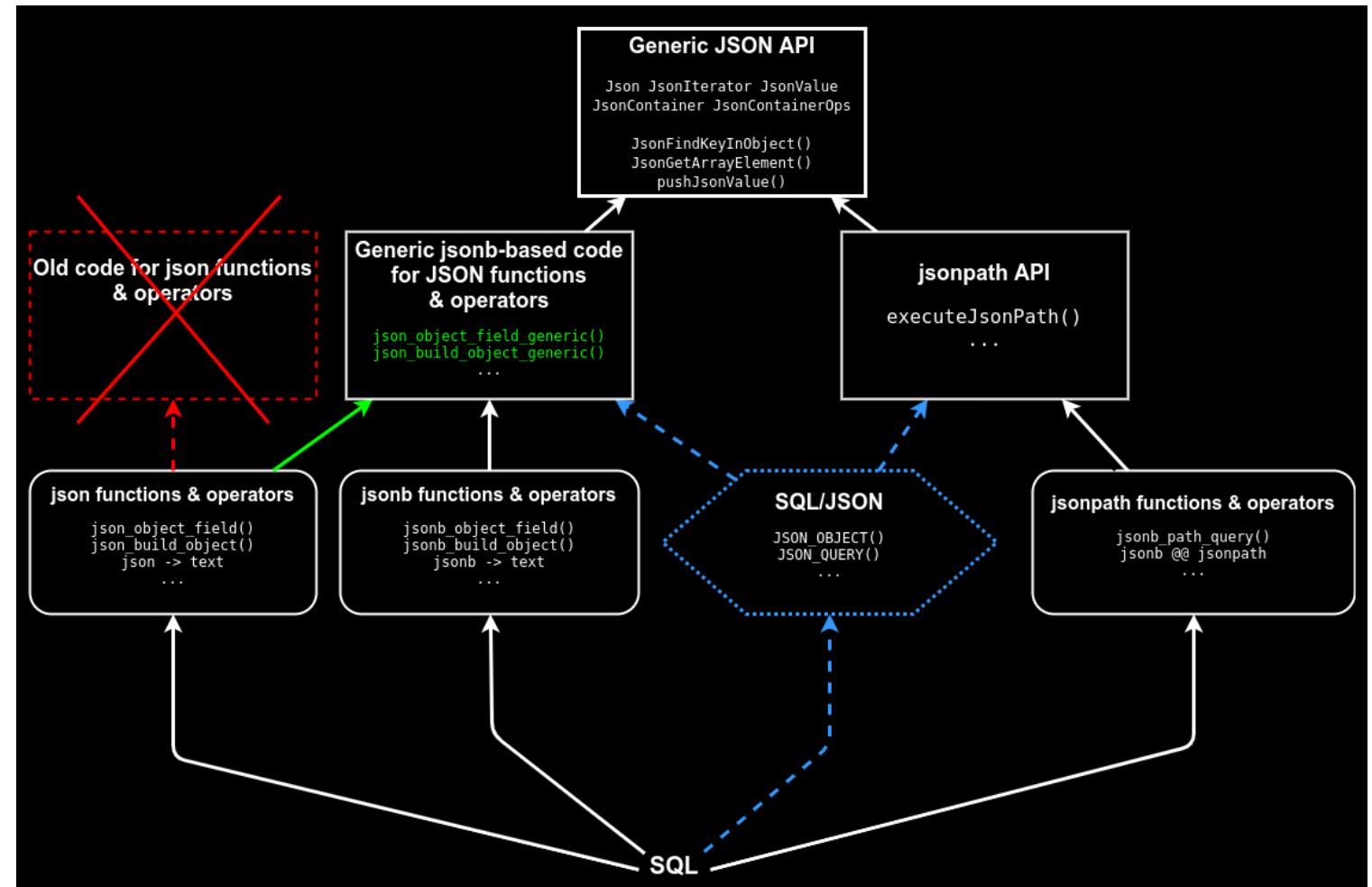
Generic JSON (GJSON) – New API

- New generic JSON API is based on jsonb API:
 - JSON datums, containers, and iterators are wrapped into generic Json, JsonContainer, and JsonIterator structures.
 - JsonbValue and its builder function pushJsonbValue() are renamed and used as is.
- All container-specific functions are hidden into JsonContainerOps, which has three implementations:
 - JsonbContainerOps for jsonb
 - JsontContainerOps for json
 - JsonvContainerOps for in-memory tree-like JsonValue (not shown)
- For json only iterators need to be implemented, access functions are implemented using these iterators.
- Details available in Addendum



Generic JSON (GSON) – Duplicated code removal

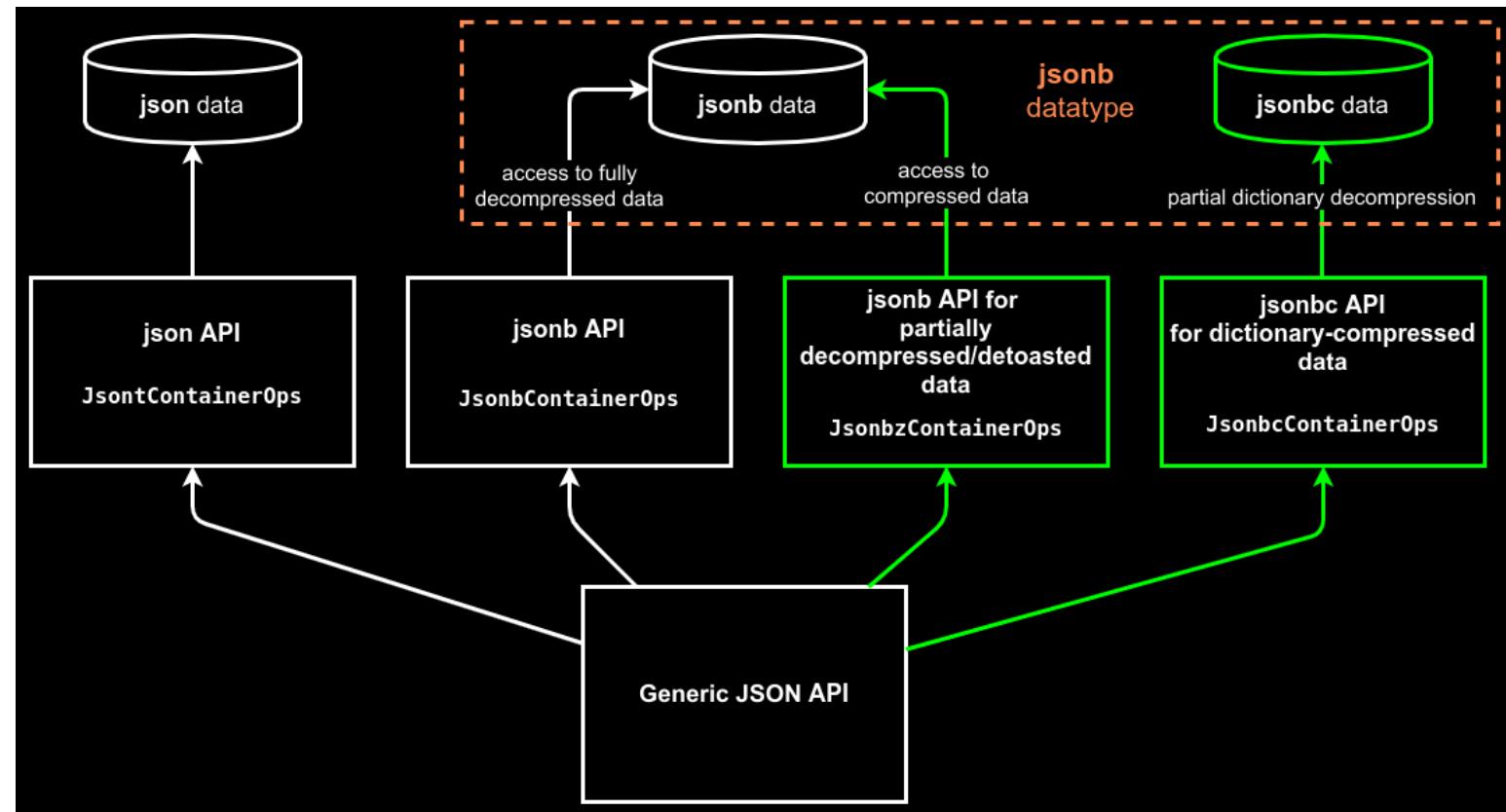
- Generic JSON API allows reuse the code of jsonb user functions and operators for json data type.
- The old code of json functions and operators is almost completely removed.
- Json and jsonb user functions have only the separate entry in which input datums are wrapped into Json structures. But they share the same body by the calling generic subroutine.
- SQL/JSON make use these generic subroutines and generic jsonpath API.



Generic JSON (GSON) – New features

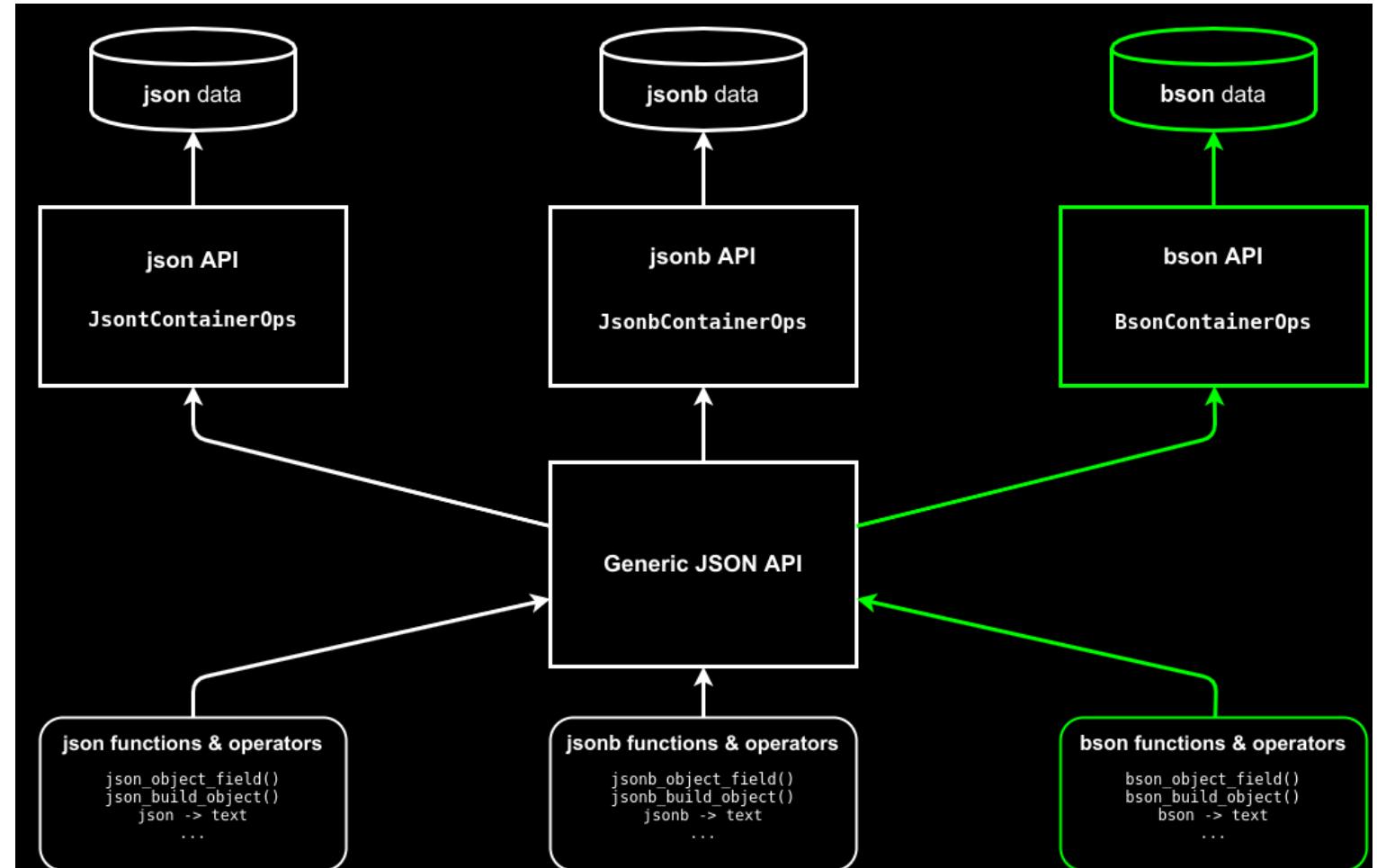
- It is easy to add new features like partial decompression/detoeasting or slicing, different storage formats (jsonbc, bson, oson,...).
- Need to implement ContainerOps interface:
Two Mandatory methods – init, IteratorInit
(other methods could be used from default implementation)

```
struct JsonContainerOps
{
    int          dataSize; /* size of
JsonContainer.data[] */
    void        (*init)(JsonContainerData *cont,
Datum value);
    JsonIterator *(*iteratorInit)(JsonContainer *cont);
    JsonValue   *(*findKeyInObject)(JsonContainer
*obj, const char *key, int len);
    JsonValue   *(*findValueInArray)(JsonContainer
*arr, const JsonValue *val);
    JsonValue   *(*getArrayElement)(JsonContainer
*arr, uint32 index);
    uint32      (*getArraySize)(JsonContainer *arr);
    char        *(*toString)(StringInfo out,
JsonContainer *, int estimatedlen);
    JsonContainer *(*copy)(JsonContainer *src);
};
```



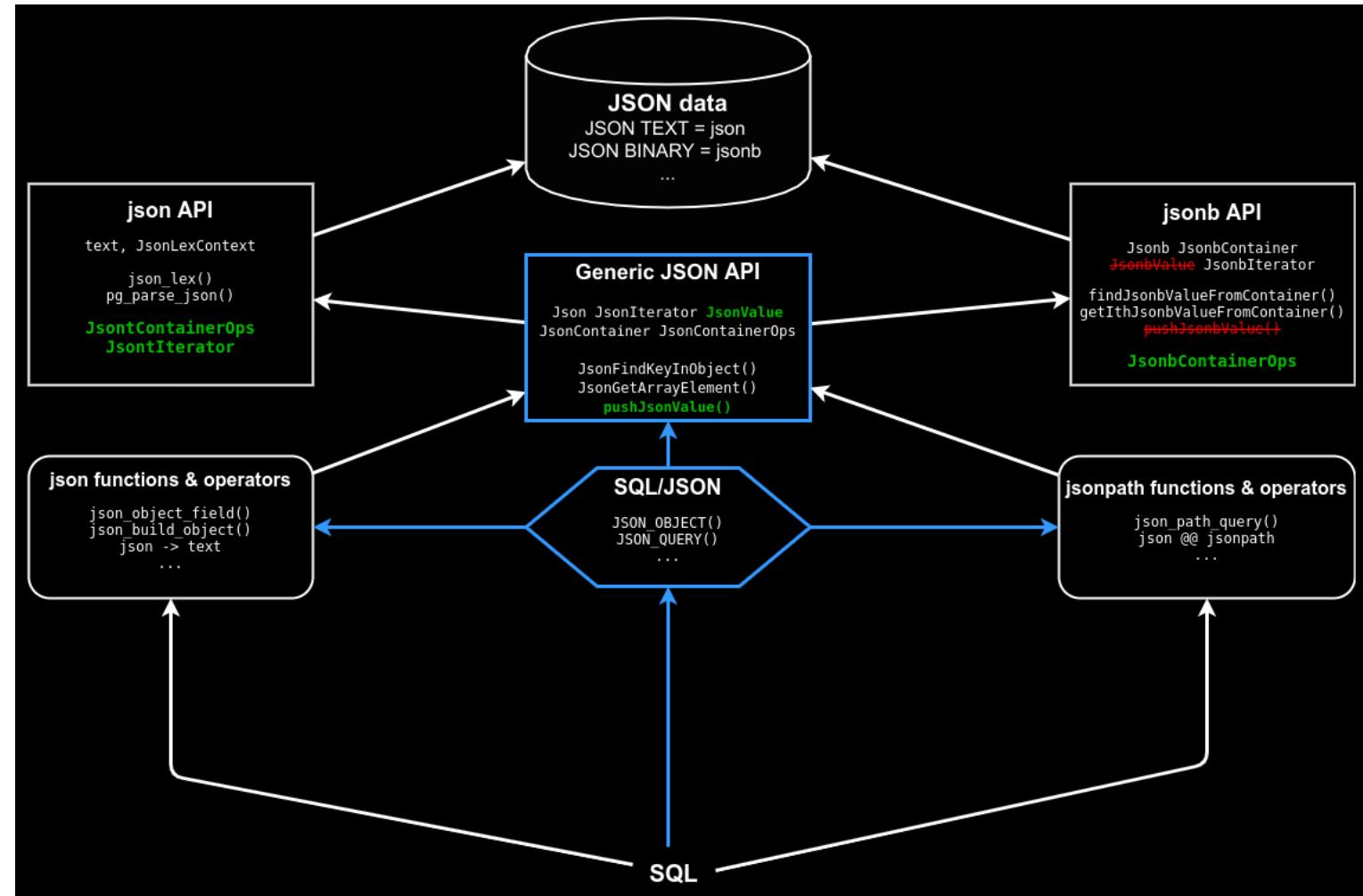
Generic JSON (GSON) – BSON data type

- New BSON data type conforming to BSON specification can be easily added by implementing `bsonInit()`, `bsonIteratorInit()` and one encoding function.
- This can give us support for binary storage of datatypes like `bytea`, `datetime` avoiding compatibility problems of extending `jsonb`. Only `JsonValue` needs to be extended.
- But the whole set of user functions, operators and opclasses needs to be duplicated:
 - `bson_build_object()`
 - `bson -> text`
 - `bson_path_ops`
 - ...



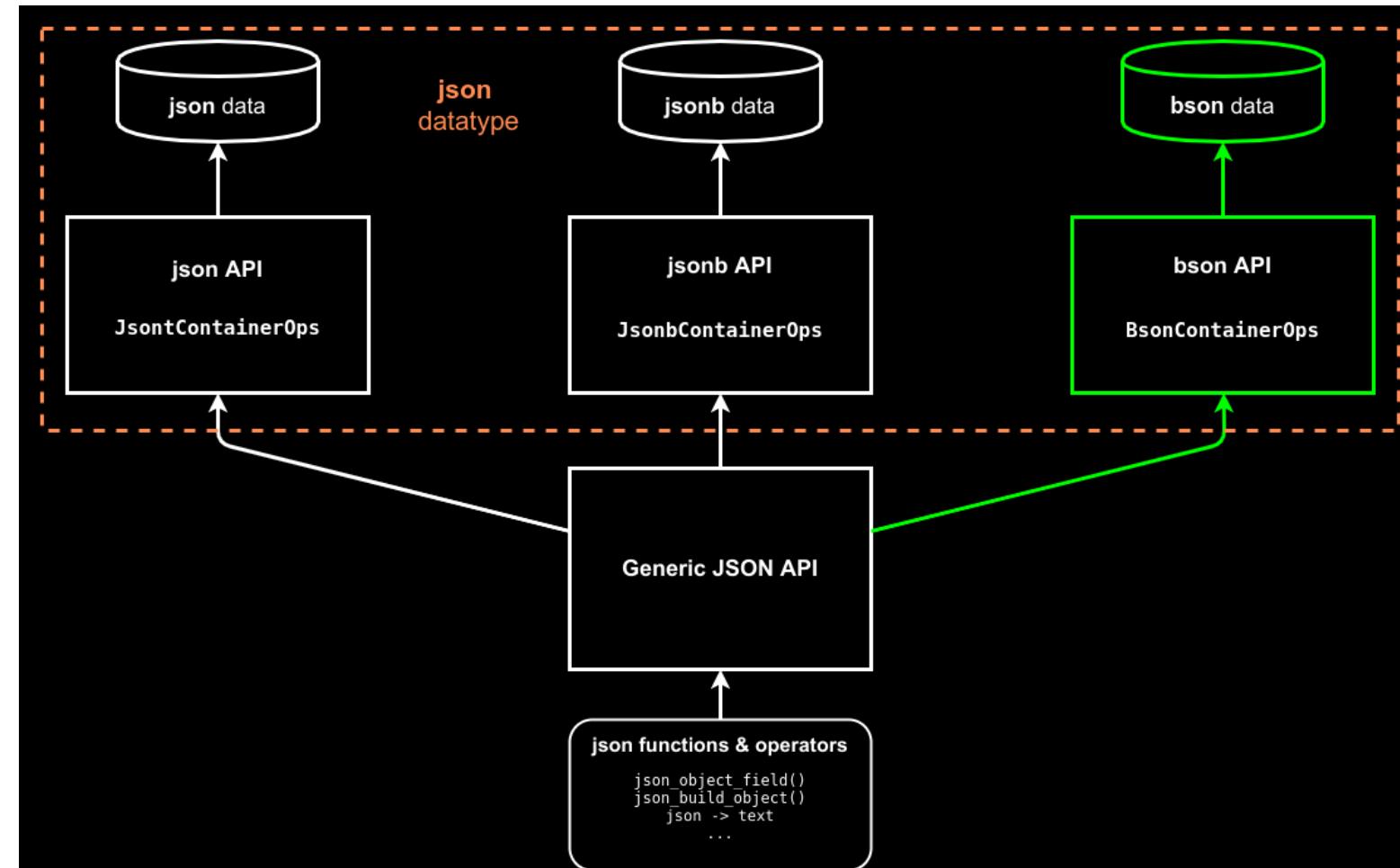
Generic JSON (GSON) – Unified JSON data type

- GSON is a step forward to Unified JSON data type, it unifies implementation of users functions.
- But it doesn't solve the problem of unified json data type as required by SQL Standard.
- We have three options:
 - 1) Noop, json (as from standard) applications will be slow
 - 2) SQL JSON as JSON (textual) or JSONB
 - 3) Implement pluggable storage method for data type. In case of JSON it means **only one** JSON data type and several storage formats: text, binary, compressed binary,..., and **one set** of functions !
- Possible variant of development:
1 → 2 → 3.



Generic JSON (GSON) – STORAGE METHODS

- The best way would probably be introduction of custom storage methods for data types. Json, jsonb, bson then will be storage methods for the single data type json.
- The possible SQL syntax:
CREATE TABLE tab (js json STORED bson);
- PRO: Only one set of user functions is needed for json datatype.
- CONTRA: many internal implementation problems:
 - Wrapping data extracted from tuple attributes into ExpandedObjects or storing storage method ID in additional data header
 - Passing JsonValues wrapped into ExpandedObjects between function calls instead of passing of binary datums



Indexing JSON

- GJSON allows to implement @> op and GIN opclass for indexing JSON

Schema	Name	Type	Owner	Persistence	Size	Description
public	jb	table	postgres	permanent	1369 MB	jb jsonb
public	js	table	postgres	permanent	1322 MB	js json

Indexes:

"jb_jb_idx" gin (jb jsonb_path_ops) Total: 1252973 bookmarks
"js_js_idx" gin (js json_path_ops)

```
SELECT count(*) FROM js WHERE js @> '{"tags": [{"term": "NYC"}]}';
```

Aggregate (actual time=3.749..3.750 rows=1 loops=1)

-> Bitmap Heap Scan on js (actual time=0.223..3.718 rows=285 loops=1)

 Recheck Cond: (js @> '{"tags": [{"term": "NYC"}]}'::json)

 Heap Blocks: exact=285

-> Bitmap Index Scan on js_js_idx (actual time=0.126..0.126 rows=285 loops=1)

 Index Cond: (js @> '{"tags": [{"term": "NYC"}]}'::json)

Planning Time: 1.813 ms

Execution Time: 3.910 ms vs 14234.623 ms (seq.scan) — **3640X** performance improvements !
(8 rows)

Indexing JSON

- GJSON allows to implement GIN opclass for indexing JSON. It's still slower than jsonb, but not much - 4X slower (3.9 ms vs 0.9 ms) vs 16000X

```
SELECT count(*) FROM jb WHERE jb @> '{"tags": [{"term": "NYC"}]}';  
QUERY PLAN
```

```
Aggregate (actual time=0.863..0.863 rows=1 loops=1)  
  -> Bitmap Heap Scan on jb (actual time=0.074..0.839 rows=285 loops=1)  
        Recheck Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)  
        Heap Blocks: exact=285  
    -> Bitmap Index Scan on jb_jb_idx (actual time=0.042..0.042 rows=285 loops=1)  
          Index Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)  
Planning Time: 0.217 ms  
Execution Time: 0.889 ms  
(8 rows)
```

JSONB as SQL JSON: compatibility solution

- Most people love jsonb and would be happy to use it as SQL JSON, some people still need "textual json", so we elaborate option 2 as follows below.
- We map PG jsonb type to the new standard SQL JSON type using special mode enabled by GUC **sql_json json|jsonb** (**json** by default), also specified in postgresql.conf.
- If **sql_json = jsonb**, then:
 - jsonb is alias for "json"
 - json is alias for "json text"

JSONB as SQL JSON: compatibility solution

- SQL TYPES vs PG Internal types

Type names are rewritten in SQL parser and in `format_type_extended()`

```
\dTS+ time*
```

Schema	Name	Internal name	List of data types	Description
pg_catalog	time with time zone	timetz		time of day with time zone
pg_catalog	time without time zone	time		time of day
pg_catalog	timestamp with time zone	timestamptz		date and time with time zone
pg_catalog	timestamp without time zone	timestamp		date and time

(4 rows)

JSONB as SQL JSON: compatibility solution

- SQL TYPES vs PG Internal types

Type names are rewritten in SQL parser and in `format_type_extended()`

```
\dTS+ js*
```

Schema	Name	Internal name	Description
pg_catalog	json	json	JSON stored as text
pg_catalog	jsonb	jsonb	Binary JSON
pg_catalog	jsonpath	jsonpath	JSON path

(3 rows)

```
SET sql_json = jsonb;
```

```
\dTS+ js*
```

Schema	Name	Internal name	Description
pg_catalog	json	jsonb	Binary JSON
pg_catalog	json text	json	JSON stored as text
pg_catalog	jsonpath	jsonpath	JSON path

(3 rows)

JSONB as SQL JSON: compatibility solution

- Example:

```
CREATE TABLE t1 (js json, jb jsonb);
```

```
\d t1
```

Column	Type	Collation	Nullable	Default
js	json			
jb	jsonb			

```
SET sql_json = jsonb;
```

```
\d t1
```

Column	Type	Collation	Nullable	Default
js	json text			
jb	json			

JSONB as SQL JSON: compatibility solution

- Example:

```
SET sql_json = jsonb;  
CREATE TABLE t2 (js json, jb jsonb, jt json text);  
\d t2
```

Table "public.t2"					
Column	Type	Collation	Nullable	Default	
js	json				
jb	json				
jt	json text				

```
SET sql_json = json;  
\d t2
```

Table "public.t2"					
Column	Type	Collation	Nullable	Default	
js	jsonb				
jb	jsonb				
jt	json				

JSONB as SQL JSON: compatibility solution

- Example:

```
postgres=# SELECT j::json AS json, j::jsonb AS jsonb, j::json text AS "json text"
FROM (VALUES ('>{"cc":0, "aa": 2, "aa":1,"b":1}')) AS foo(j);
      json       |      jsonb      |      json text
-----+-----+-----+
 {"cc":0, "aa": 2, "aa":1,"b":1} | {"b": 1, "aa": 1, "cc": 0} | {"cc":0, "aa": 2, "aa":1,"b":1}
(1 row)
```

```
postgres=# SET sql_json = jsonb;
SET
```

```
postgres=# SELECT j::json AS json, j::jsonb AS jsonb, j::json text AS "json text"
FROM (VALUES ('>{"cc":0, "aa": 2, "aa":1,"b":1}')) AS foo(j);
      json       |      jsonb      |      json text
-----+-----+-----+
 {"b": 1, "aa": 1, "cc": 0} | {"b": 1, "aa": 1, "cc": 0} | {"cc":0, "aa": 2, "aa":1,"b":1}
(1 row)
```

JSON[B] Unification (with GSON): Conclusion

- PRO:

- Compatibility to the old Postgres versions and SQL Standard
 - JSON users will get huge performance boost, still slower JSONB
- Jsonb users need nothing to change
- SET SQL_JSON = JSONB
- Jsonb users need nothing to change
- Still compatible to the SQL Standard
 - JSON users will get huge performance boost on par with JSONB
- GSON simplifies coding of new features

- CONTRA:

- Additional code

47 files changed, 6968 insertions(+), 3995 deletions(-)
58.3% src/backend/utils/adt/
3.6% src/common/
6.0% src/include/catalog/
7.8% src/include/utils/
14.2% src/test/regress/expected/
7.2% src/test/regress/sql/

Jsonb partial decompression

Jsonb partial decompression

- Current jsonb implementation works with fully decompressed binary jsonb data. Jsonb and JsonbContainer structures refers to plain binary data. So, it cannot benefit from partial pglz decompression.
- Generic JSON API interface allows easy implementation of partial pglz decompression (with resume support) for jsonb by implementing only new JsonbzContainerOps without modification of user functions.
- pglz code has modified to provide an ability to caller to preserve decompression state between successive calls of `pglz_decompress()`.
- Prototype currently supports only partial pglz decompression, but it is possible to use partial de-TOASTing.

Jsonb partial decompression - Example

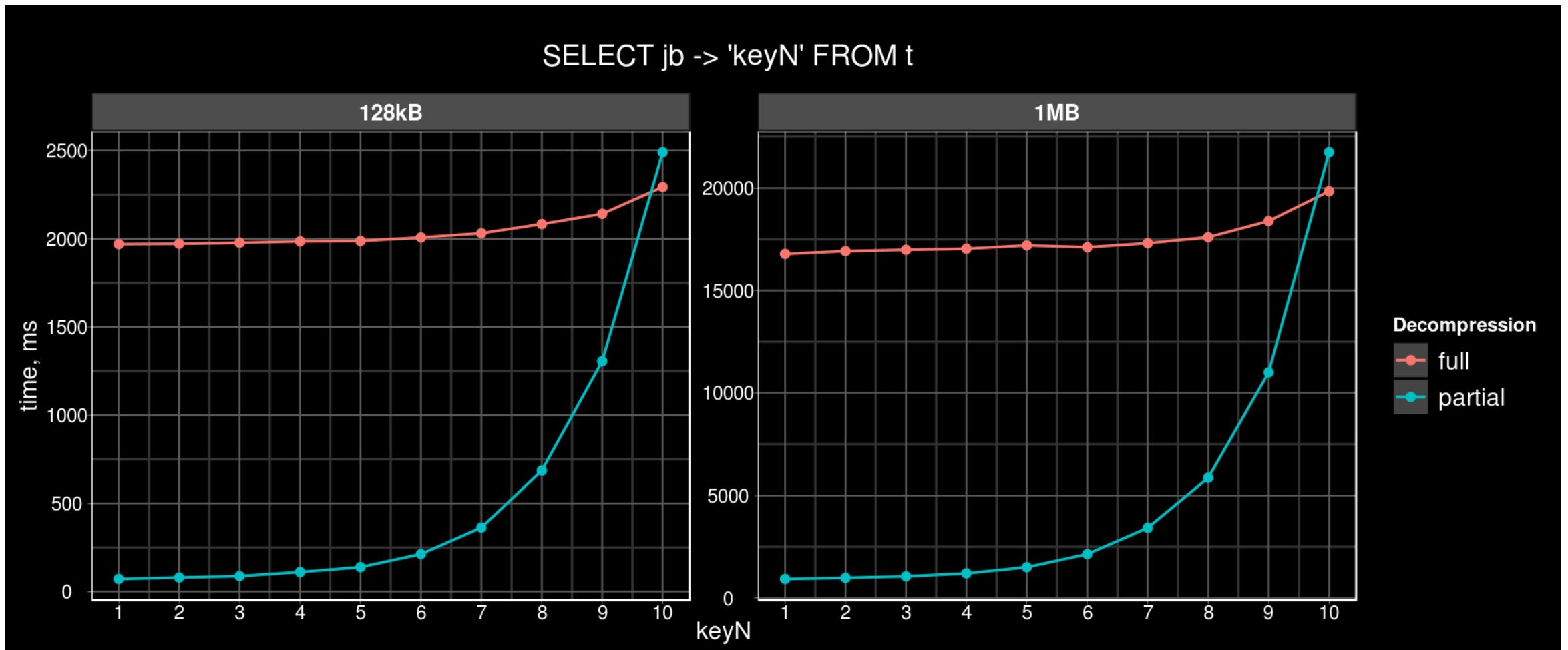
- Remember: keys in jsonb sorted by length, name
- Table with 100,000 128 KB jsonb compressed into 1.2KB (no TOAST):

```
-- { "key1": 2^6 "a", ... "key10": 2^16 "a" }
CREATE TABLE t AS SELECT (
    SELECT jsonb_object_agg('key' || i, repeat('a', pow(2, i + 6)::int)) jb
    FROM generate_series(1, 10) i
) FROM generate_series(1, 100000);
```

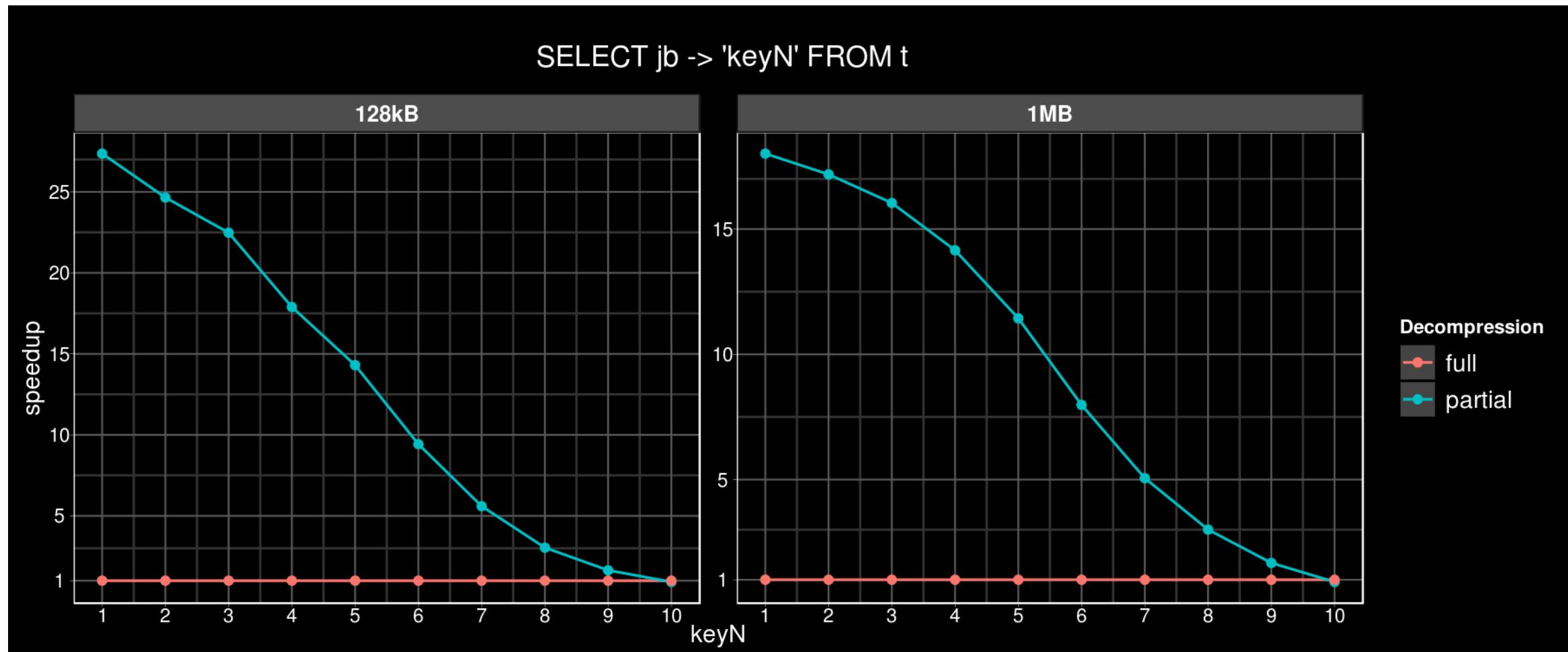
- Table with 100,000 1MB jsonb compressed into 12KB (with TOAST):

```
-- { "key1": 2^9 "a", ... "key10": 2^19 "a" }
CREATE TABLE t AS SELECT (
    SELECT jsonb_object_agg('key' || i, repeat('a', pow(2, i + 9)::int)) jb
    FROM generate_series(1, 10) i
) FROM generate_series(1, 100000);
```

Jsonb partial decompression - Example



Jsonb partial decompression - Example

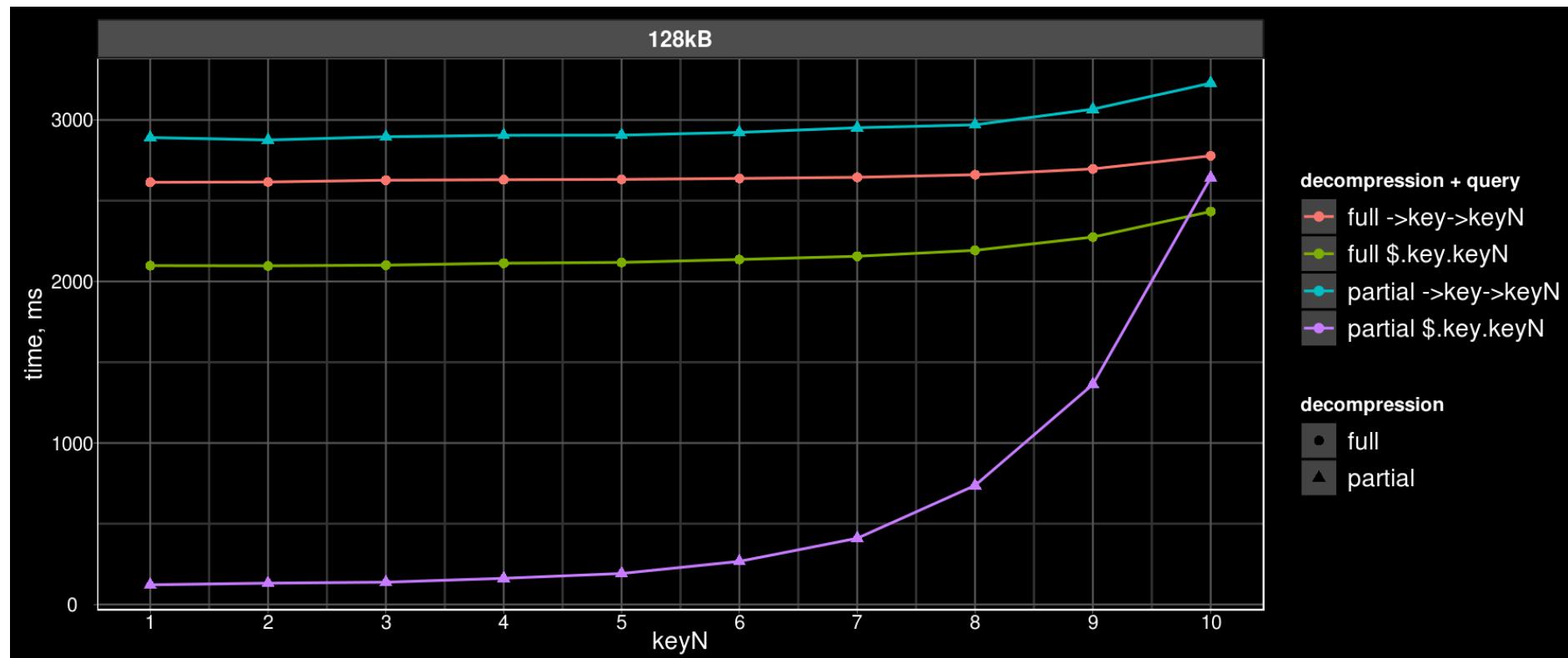


Jsonb partial decompression (jsonpath vs. ->)

- Table with 100,000 128 KB 2-level jsonb compressed into 1.2KB (no TOAST):

```
-- { "key": { "key1": 2^6 "a", ... "key10": 2^16 "a" } }
```

```
CREATE TABLE t AS SELECT (
    SELECT jsonb_build_object('key', jsonb_object_agg('key' || i, repeat('a', pow(2, i + 6)::int))) jb
    FROM generate_series(1, 10) i
) FROM generate_series(1, 100000);
```



Jsonb partial decompression - Summary

- Access to data in the beginning of compressed JSONB is much faster (without any modification of users functions). JSON support could be added.
- Multilevel JSONPATH is much faster than chained arrow operators/functions for compressed JSONB, since JSONPATH functions benefit from pglz resume between levels.
 - +Partial deToasting (prefix, sliced)
 - Life Hack: keep "important" keys in the beginning !

SQL/JSON FUNCTIONS

SQL/JSON FUNCTIONS

- The SQL/JSON **construction** functions (`json[b]_xxx()` functions):
 - `JSON_OBJECT` - construct a `JSON[b]` object.
 - `json[b]_build_object()`
 - `JSON_ARRAY` - construct a `JSON[b]` array.
 - `json[b]_build_array()`
 - `JSON_ARRAYAGG` - aggregates values as `JSON[b]` array.
 - `json[b]_agg()`
 - `JSON_OBJECTAGG` - aggregates name/value pairs as `JSON[b]` object.
 - `json[b]_object_agg()`

SQL/JSON in PostgreSQL

- The SQL/JSON **retrieval** functions:

- JSON_VALUE - Extract an SQL value of a predefined type from a JSON value.
- JSON_QUERY - Extract a JSON text from a JSON text using an SQL/JSON path expression.
- JSON_TABLE - Query a JSON text and present it as a relational table.
- IS [NOT] JSON - test whether a string value is a JSON text.
- JSON_EXISTS - test whether a JSON path expression returns any SQL/JSON items

PGOpen-2018 talk : [SQL/JSON in PostgreSQL](#)

JSON_TABLE — relational view of json

- Table with rooms from json

```
SELECT
  (floor -> 'level')::int AS level,
  (apt -> 'no')::int AS no,
  (apt ->> 'area')::int AS area,
  (apt -> 'rooms')::int AS num_rooms
FROM
  house,
  LATERAL jsonb_path_query(js, '$.floor[*]') floor,
  LATERAL jsonb_path_query(floor, '$.apt[*]') apt;
```

level	no	area	num_rooms
1	1	40	1
1	2	80	3
1	3	50	2
2	4	100	3
2	5	60	2

(5 rows)

JSON_TABLE — relational view of json

- Table with rooms from json - SQL/JSON

```
SELECT apt.*  
FROM  
house,  
JSON_TABLE(js, '$.floor[0, 1]' COLUMNS  
(  
    level int,  
    NESTED PATH '$.apt[1 to last]' COLUMNS  
(  
        no int,  
        area int,  
        num_rooms int PATH '$.rooms'  
    )  
)  
) apt;
```

level	no	area	num_rooms	
1	1	40		1
1	2	80		3
1	3	50		2
2	4	100		3
2	5	60		2
(5 rows)				

Parameters for Opclasses

Parameters for opclasses

Operator class (opclass) is a «glue» or named collection of:

- AM (access method)
- Set of operators
- AM specific support function

Examples:

- CREATE INDEX .. USING btree (textcolumn **text_pattern_ops**)
- CREATE INDEX .. USING gin (jsoncolumn **jsonb_ops**)
- CREATE INDEX .. USING gin (jsoncolumn **jsonb_path_ops**)

Extending Indexing infrastructure

- Opclasses have «hardcoded» constants (signature size)
 - Let user to define these constants for specific data — **PG13**
- Use different algorithms to index
 - Specify what to use depending on data
- Indexing of non-atomic data (arrays, json[b], tsvector,...)
 - Specify what part of column value to index unlike partial index
 - No need to use an exact expression in SQL as for functional index, since opclass knows what to index unlike functional index
- Talk @ PGCON-2018: Jsonb "smart" indexing. Parametrized opclasses.

Parameters for opclasses: syntax

- Parenthesized parameters added after column's opclass.

```
CREATE INDEX idx ON tab USING am (
    {expr opclass ({name=value} [, ...])} [, ...]
) ...
```

```
CREATE INDEX ON small_arrays USING gist (
    arr gist_intbig_ops(siglen=32)
);
CREATE INDEX ON small_arrays USING gist (
    arr gist_int_ops (num_ranges = 100)
);
CREATE INDEX bookmarks_selective_idx ON bookmarks USING
    gin(js jsonb_ops(projection='strict $.tags[*].term'));
```

Parameters for opclasses: jsonb selective indexing

- Opclass parameters can be used in GIN jsonb opclasses for selective indexing of parts of jsonb documents. We implemented 'projection' parameter for in-core jsonb_ops and jsonb_path_ops, which specify what part(s) of JSONB to index using jsonpath expression.

Example of creation selective index on `$.roles[*].role`:

```
CREATE INDEX names_role_prj_idx ON imdb.names USING gin
  (jb jsonb_path_ops (projection = '$.roles[*].role'));
```

- Using jsonpath extension for sequence constructions we can index multiple paths (branches of JSONB tree):

```
CREATE INDEX names_role_prj_idx ON imdb.names USING gin
  (jb jsonb_path_ops (projection = 'pg $.id, $.roles[*].role'));
```

Parameters for opclasses: jsonb selective indexing

- Comparison of full, functional, and selective jsonb indexes on imdb titles:

- full index:

```
CREATE INDEX titles_full_idx ON titles USING gin (jb jsonb_path_ops);
```

- functional index on \$.id, \$.kind \$.year[*]:

```
CREATE FUNCTION title_id_kind_year(title jsonb) RETURNS jsonb LANGUAGE sql AS
$$ SELECT jsonb_path_query_first(title, 'pg {id: $.id, kind: $.kind, year: $.year}') $$;
```

```
CREATE INDEX titles_func_idx ON titles USING gin (title_id_kind_year(jb) jsonb_path_ops);
```

- selective index on \$.id, \$.kind \$.year[*]:

```
CREATE INDEX titles_prj_idx ON titles USING gin
  (jb jsonb_path_ops (projection = 'pg $.id, $.kind, $.year'));
```

- Resulting index size and build time:

titles_full_idx		644 MB		118 s
titles_func_idx		219 MB		29 s
titles_prj_idx		214 MB		24 s

Parameters for opclasses: jsonb selective indexing

- Insertion into selective indexes is obviously much faster:

```
-- create a copy of imdb.names with 100000 rows, no indexes
CREATE TABLE names2 AS SELECT * FROM names LIMIT 100000;
Execution Time: 424.848 ms
```

```
-- full index
TRUNCATE names2;
CREATE INDEX names2_jb_full_idx ON names2 USING gin (jb jsonb_path_ops);
INSERT INTO names2 SELECT * FROM names LIMIT 100000;
Execution Time: 5315.943 ms
```

```
-- selective index
DROP INDEX names2_jb_full_idx;
TRUNCATE names2;
CREATE INDEX names2_jb_role_idx ON names2 USING gin
  (jb jsonb_path_ops (projection = '$.roles[*].role'));
INSERT INTO names2 SELECT * FROM names LIMIT 100000;
Execution Time: 1048.785 ms
```

The whole insertion is 5x faster, index insertion is about 10x faster.

Parameters for opclasses: jsonb selective indexing

- The same query can be accelerated by full and selective indexes

```
SELECT * FROM imdb.names
WHERE jb @@ '$.roles[*].role == "costume-designer"';  
  
Bitmap Heap Scan on names (actual rows=60791 loops=1)
  Recheck Cond: (jb @@ '("$.roles[*].role" == "costume-designer")'::jsonpath)
  Heap Blocks: exact=53773
    -> Bitmap Index Scan on names_role_prj_idx (actual rows=60791 loops=1)
      Index Cond: (jb @@ '("$.roles[*].role" == "costume-designer")'::jsonpath)
Planning Time: 0.139 ms
Execution Time: 519.332 ms
```

- Query should includes indexed expression for functional index

```
SELECT * FROM names
WHERE jsonb_path_query_array(jb, '$.roles[*].role') @> '"costume-designer"';  
  
Bitmap Heap Scan on names (actual rows=60791 loops=1)
  Recheck Cond: (jsonb_path_query_array(jb, '$.roles[*].role'::jsonpath, '{})::jsonb, false) @> '"costume-
designer"'::jsonb
  Heap Blocks: exact=53773
    -> Bitmap Index Scan on names_role_fn_idx (actual rows=60791 loops=1)
      Index Cond: (jsonb_path_query_array(jb, '$.roles[*].role'::jsonpath, '{})::jsonb, false) @> '"costume-
designer"'::jsonb
Planning Time: 0.130 ms
Execution Time: 556.334 ms
```

Parameters for opclasses: jsonb selective indexing

Example: simple query contains only indexed paths

- The same query can be accelerated by full and selective indexes

```
SELECT * FROM imdb.titles WHERE jb @> '{"kind": "episode", "year": [2016]}';
```

```
Bitmap Heap Scan on titles (actual rows=184431 loops=1)
  Recheck Cond: (jb @> '{"kind": "episode", "year": [2016]}'::jsonb)
    Heap Blocks: exact=42963
      -> Bitmap Index Scan on titles_prj_idx (actual rows=184431 loops=1)
        Index Cond: (jb @> '{"kind": "episode", "year": [2016]}'::jsonb)
Planning Time: 0.162 ms
Execution Time: 386.052 ms
```

- Query should includes indexed condition for functional index

```
SELECT * FROM imdb.titles
WHERE title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}';
```

```
Bitmap Heap Scan on titles (actual rows=184431 loops=1)
  Recheck Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}'::jsonb)
    Heap Blocks: exact=42963
      -> Bitmap Index Scan on titles_fn_idx3 (actual rows=184431 loops=1)
        Index Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}'::jsonb)
Planning Time: 0.174 ms
Execution Time: 590.238 ms (more complex recheck)
```

Parameters for opclasses: jsonb selective indexing

Complex queries for functional indexes require manually factorize indexable condition and include it into WHERE, possibly with duplication:

- selective index – single jsonpath condition

```
SELECT * FROM imdb.names
WHERE jb @@ '$.roles[*] ? (@.title starts with "Foo").role == "costume-designer"';
```

- functional index – two duplicate conditions (indexable functional + jsonpath @?)

```
SELECT * FROM imdb.names
WHERE
  jsonb_path_query_array(jb, '$.roles[*].role') @> '"costume-designer"'
  AND
  jb @? '$.roles[*] ? (@.title starts with "Foo" && @.role == "costume-designer")';
```

Parameters for opclasses: jsonb selective indexing

Example: simple query + non-indexable condition

- The same query can be accelerated by full and selective indexes

```
SELECT * FROM imdb.titles
WHERE jb @> '{"kind": "episode", "year": [2016], "title": "Telediario"}';

Bitmap Heap Scan on titles (actual rows=2 loops=1)
  Recheck Cond: (jb @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
  Rows Removed by Index Recheck: 184429
  Heap Blocks: exact=42963
    -> Bitmap Index Scan on titles_prj_idx (actual rows=184431 loops=1)
      Index Cond: (jb @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
Planning Time: 0.123 ms
Execution Time: 382.264 ms
```

- functional index (not working) — wrong query

```
SELECT * FROM titles
WHERE title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016], "title": "Telediario"}';

Bitmap Heap Scan on titles (actual rows=0 loops=1)
  Recheck Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
    -> Bitmap Index Scan on titles_fn_idx (actual rows=0 loops=1)
      Index Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
Planning Time: 0.161 ms
Execution Time: 0.141 ms
```

Parameters for opclasses: jsonb selective indexing

Example: simple query + non-indexable condition

- The same query can be accelerated by full and selective indexes:

```
SELECT * FROM imdb.titles
WHERE jb @> '{ "kind": "episode", "year": [2016], "title": "Telediario" }';

Bitmap Heap Scan on titles (actual rows=2 loops=1)
  Recheck Cond: (jb @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
  Rows Removed by Index Recheck: 184429
  Heap Blocks: exact=42963
    -> Bitmap Index Scan on titles_prj_idx (actual rows=184431 loops=1)
      Index Cond: (jb @> '{"kind": "episode", "year": [2016], "title": "Telediario"}'::jsonb)
Planning Time: 0.123 ms
Execution Time: 382.264 ms
```

- functional index - correct query:

```
SELECT * FROM titles
WHERE title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}'
  AND jb @> '{"title": "Telediario"}';

Bitmap Heap Scan on titles (actual rows=2 loops=1)
  Recheck Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}'::jsonb)
  Filter: (jb @> '{"title": "Telediario"}'::jsonb)
  Rows Removed by Filter: 184429
  Heap Blocks: exact=42963
    -> Bitmap Index Scan on titles_fn_idx (actual rows=184431 loops=1)
      Index Cond: (title_id_kind_year(jb) @> '{"kind": "episode", "year": [2016]}'::jsonb)
Planning Time: 0.186 ms
Execution Time: 651.831 ms
```

Parameters for opclasses: jsonb selective indexing

Selective jsonb opclasses support:

- Various user operators:

```
jb @@ '$.kind == "series" && $.year[*] == 2020'  
jb @? '$ ? (@.kind == "series" && @.year[*] == 2020)'  
jb @> '{ "kind": "series", "year": [ 2020 ] }'
```

- Arbitrary additional non-indexed conditions logically ANDed :

```
jb @@ '$.kind == "series" && $.id starts with "Foo"';  
jb @? '$ ? (@.id starts with "Foo").kind == "series"';  
jb @> '{ "kind": "series", "release_dates": [{ "country": "USA"}] }'
```

- Array subscripts matching:

```
jb @@ '$.year[0 to 2] == 2020';
```

- Strict/lax indexed/query paths combinations

Not supported:

- Filters in indexed paths (predicate implication proving is needed)
- Expressions (for example, \$.a + \$.b)
- Item methods (like .type(), .datetimetz())

JsQuery indexes to core

JSONB indexing: Jquery extension

- jsonb_path_value_ops
 - (hash(full_path);value)
 - exact and range queries on values, exact path searches
- jsonb_laxpath_value_ops (branch sqljson)
 - The same as above, but array path items are ignored, which greatly simplifies extraction of *lax* JSON path queries.
- jsonb_value_path_ops
 - (value; bloom(path_1) | bloom(path_2) | ... bloom(path_N))
 - Exact value search and wildcard path queries.
- Also, jquery provides debugging and query optimizer with hints.

Jsquery_laxpath_value_ops

- jsonb_path_value_ops is slow for long lax paths:

```
CREATE TABLE t AS SELECT '{}'::jsonb jb FROM generate_series(1, 1000000) i;
```

```
CREATE INDEX t_path_ops_idx ON t USING gin (jb jsonb_path_value_ops);
```

```
SELECT * FROM t WHERE jb @@ '$.k1.k2.k3.k4.k5.k6.k7.k8.k9 == 1'::jsonpath;
```

Planning Time: **0.574 ms**

Execution Time: **3.026 ms**

- For this jsonpath jsonb_path_value_ops extracts 2^{10} (1024) entries:

```
SELECT gin_debug_jsonpath_path_value('$.k1.k2.k3.k4.k5.k6.k7.k8.k9 == 1');
```

```
          gin_debug_jsonpath_path_value
```

OR

```
#.k1.#.k2.#.k3.#.k4.#.k5.#.k6.#.k7.#.k8.#.k9 = 1 , entry 0 +
```

```
#.k1.#.k2.#.k3.#.k4.#.k5.#.k6.#.k7.#.k8.#.k9.# = 1 , entry 1 +
```

```
...  
k1.k2.k3.k4.k5.k6.k7.k8.k9 = 1 , entry 1022 +
```

```
k1.k2.k3.k4.k5.k6.k7.k8.k9.# = 1 , entry 1023 +
```

(1 row)

jquery_laxpath_value_ops

- jsonb_laxpath_value_ops does not account arrays in paths and:

```
CREATE INDEX t_laxpath_ops_idx ON t USING gin (jb jsonb_laxpath_value_ops);  
SELECT * FROM t WHERE jb @@ '$.k1.k2.k3.k4.k5.k6.k7.k8.k9 == 1'::jsonpath;
```

Planning Time: **0.059 ms** (was 0.574 ms, 10x speedup)
Execution Time: **0.025 ms** (was 3.026 ms, 120x speedup)

- For this jsonpath jsonb_laxpath_value_ops extracts only 1 entry as expected:

```
SELECT gin_debug_jsonpath_laxpath_value('$.k1.k2.k3.k4.k5.k6.k7.k8.k9 == 1');  
  
gin_debug_jsonpath_laxpath_value  
-----  
k1.k2.k3.k4.k5.k6.k7.k8.k9 = 1 , entry 0 +  
(1 row)
```

Planner support for jsonpath

Planner support function for jsonpath functions

- PG12+: API for planner support functions allows transform functions to operator expression, which can be accelerated by index.

```
CREATE [OR REPLACE] FUNCTION
  name ([[argmode] [argname] argtype [{DEFAULT|=} default_expr] [,...]])
{
  .....
  | SUPPORT support_function
  .....
}
```

- `jsonb_path_match()` transforms to `jsonb @@ jsonpath` (uses index !) and pass variables (impossible for operators) !

Planner support function for jsonpath functions

- PG12+: API for planner support functions allows transform functions to operator expression, which can be accelerated by index.

```
SELECT * FROM t t1, t t2 WHERE  
jsonb_path_match(t1.js, '$.a == $a', vars => t2.js, silent => true);  
          index           QUERY PLAN
```

```
Nested Loop  
  -> Seq Scan on t t2  
  -> Bitmap Heap Scan n t t1  
      Filter: jsonb_path_match(js, '($."a" == $"a")'::jsonpath,  
t2.js, true)  
          -> Bitmap Index Scan on t_js_idx  
              Index Cond: (js @@ jsonpath_embed_vars('($."a" ==  
$"a")'::jsonpath, t2.js))  
(6 rows)
```

Jsonpath syntax extensions

Jsonpath syntax extensions

- Array construction syntax:

```
SELECT jsonb_path_query('[1,2,3]', '[0, $[*], 4]');
[0, 1, 2, 3, 4]
```

- Object construction syntax:

```
SELECT jsonb_path_query('[1,2,3]', '{a: $, "s": $.size()}' );
{"a": [1, 2, 3], "s": 3}
```

- Sequence construction syntax:

```
SELECT jsonb_path_query('[1,2,3]', '0, $[*], 4');
0
1
2
3
4
```

Jsonpath syntax extensions (constructors)

- Sequence construction syntax is useful for selective jsonb indexing (see "Opclass parameters"):

```
CREATE INDEX titles_prj_idx ON titles USING gin  
  (jb jsonb_path_ops(projection = 'pg $.id, $.kind, $.year[*]'));
```

Jsonpath syntax extensions (constructors)

- Object and array construction syntax is useful for constructing derived objects:

```
SELECT jsonb_path_query(jb, 'pg {id: $.id,
                                company_names: [$.companies[*].name]}')
FROM imdb.titles;
Execution Time: 13569.575 ms
```

- Without constructors query becomes much more complex and more slow (especially variant without jsonpath):

```
SELECT jsonb_build_object(
    'id', jb -> 'id',
    'company_names', jsonb_path_query_array(jb, '$.companies[*].name')
) FROM imdb.titles;
Execution Time: 17228,896 ms
```

```
SELECT jsonb_build_object(
    'id', jb -> 'id',
    'company_names', (SELECT jsonb_agg(company -> 'name')
                      FROM jsonb_array_elements(jb -> 'companies') company)
) FROM imdb.titles;
Execution Time: 29716,909 ms
```

Jsonpath syntax extensions (IN simplification)

- Array and sequence construction syntax can be useful for set-inclusion queries (like SQL IN predicate):

```
SELECT count(*) FROM imdb.names
WHERE jb @@ 'pg $.roles[*].role == ("editor", "producer", "director")';
Execution Time: 20622.929 ms
```

```
SELECT count(*) FROM imdb.names
WHERE jb @@ 'pg $.roles[*].role == ["editor", "producer", "director"]'; -- unwrapped in lax
Execution Time: 21508.483 ms
```

- Currently, one should use a sequence of AND conditions that even works slower:

```
SELECT count(*) FROM imdb.names
WHERE jb @? 'pg $.roles[*].role ?
(@ == "editor" || @ == "producer" || @ == "director")';
Execution Time: 29490.021 ms
```

- Such IN expressions with constants can be speed up with GIN indexes.

Jsonpath syntax extensions

- Object subscripting:

```
SELECT jsonb_path_query('{"a": 1}', '$["a"]');  
1
```

```
SELECT jsonb_path_query('{"a": 1, "b": "ccc"}', '$["a", "b"]');  
1  
"ccc"
```

```
SELECT jsonb_path_query('{"a": 1}', 'lax $["a", "b"]');  
1
```

```
SELECT jsonb_path_query('{"a": 1}', 'strict $["a", "b"]');  
ERROR: JSON object does not contain key "b"
```

Jsonpath syntax extensions

- Array item methods with lambda expressions (ECMAScript 6 style):

```
SELECT jsonb_path_query('[1,2,3]', '$.map(x => x + 10)');
[11, 12, 13]
```

```
SELECT jsonb_path_query('[1,2,3]', '$.reduce((x,y) => x + y)');
6
```

```
SELECT jsonb_path_query('[1,2,3]', '$.fold((x,y) => x + y, 10)');
16
```

```
SELECT jsonb_path_query('[1,2,3]', '$.max());
3
```

- Alternative syntax for lambdas: '\$.fold(\$1 + \$2, 10)'

Jsonpath syntax extensions

- Sequence functions with lambda expressions:

```
SELECT jsonb_path_query('[1,2,3]', 'map($[*], x => x + 10)');
11
12
13      -- sequence is returned, not array
```

```
SELECT jsonb_path_query('[1,2,3]', 'reduce($[*], (x,y) => x+y)');
6
```

```
SELECT jsonb_path_query('[1,2,3]', 'fold($[*], (x,y)=>x+y, 10)');
16
```

```
SELECT jsonb_path_query('[1,2,3]', 'max($[*])');
3
```

Simple Dot-Notation Access To JSON Data

Simple.Dot.Notation.Access.To.JSON.Data

- Simple dot-notation is handy for simple queries than arrow operators and functions
 - `j.key1.key2` instead of:
 - `j -> 'key1' -> 'key2'`
 - `JSON_QUERY(j, '$.key1.key2')`
 - `j.key1[1]` instead of:
 - `j -> 1`
 - `JSON_QUERY(j, '$.key1[1]')`
 - `j.key1[*]` instead of:
 - `json_array_elements(j -> 'key1') SRF ???`
 - `JSON_QUERY(j, '$.key1[*]') WITH ARRAY WRAPPER ???`

COPY ... (FORMAT json)

- Currently it's impossible to load data from JSON without temporary table
- COPY [TO|FROM] ... (FORMAT json)
- It's possible to load only selected data from JSON using jsonpath
 - Example:
Load array of json: `[{"a": ..., "b": ..., "c": ... }, ...]`
 - `COPY target_table FROM 'path/to/data.json' (FORMAT json, JSONPATH '$[*]');`

JSON[B] Unification (with GSON): Plan

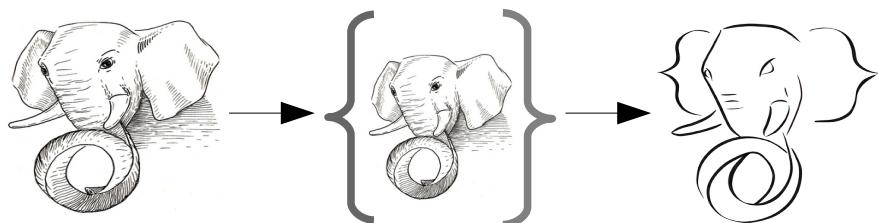
- A feasible plan for PG14:
 - Compatibility with SQL Standard
 - GUC variable `sql_json` — 7 files changed, 63 insertions(+), 3 deletions(-)
 - GSON API — 26 files changed, 4806 insertions(+), 4331 deletions(-)
 - SQL/JSON functions - 64 files changed, 16940 insertions(+), 318 deletions(-)
 - Extra
 - JSON_OPS (GIN opclass) - 5 files changed, 2294 insertions(+), 224 deletions(-)
 - Jsonpath syntax extensions
 - Jsonpath as parameter for jsonb opclasses
 - Jsonb partial decompression
 - Jsquery GIN opclasses to core
 - Planner support functions

JSON Sandbox

- The Grand Unification - JSON as JSONB @ Github
- Generic JSON API + Unification + json_path_ops - Generic JSON API @ Github
- SQL/JSON functions from SQL-2016 standard - SQL/JSON @ Github
- Jsonpath as parameter for jsonb opclasses - Jsonb selective opclasses @ Github
- Jsonb partial decompression - Jsonb partial decompression @ Github
- Planner support functions - Jsonb planner support functions @ Github
- Jsquery GIN opclasses to core - Jsquery GIN opclasses
- Jsonpath syntax extension - Jsonpath syntax extensions @ Github
- Simple Dot-Notation Access to JSON Data
- COPY with support of jsonpath (idea)
- Contact obartunov@postgrespro.ru, n.gluhov@postgrespro.ru for collaboration
Together we can do anything !

References

- 1) This talk: <http://www.sai.msu.su/~megera/postgres/talks//json-webinar-2020.pdf>
- 2) Jsonpath talk at PGConf-EU, Milan, 2019
<http://www.sai.msu.su/~megera/postgres/talks/jsonpath-pgconfeu-2019.pdf>
- 3) SQL/JSON functions: [SQL/JSON in PostgreSQL @ PGOpen, 2018](#)
- 4) Technical Report (SQL/JSON) - available for free
http://standards.iso.org/i/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip
- 5) Gentle introduction to JSON Path in PostgreSQL
<https://github.com/obartunov/sqljsonondoc/blob/master/jsonpath.md>
- 6) Jquery extension: <https://github.com/postgrespro/jquery/tree/sqljson>
<http://www.sai.msu.su/~megera/postgres/talks/pgconfeu-2014-jquery.pdf>
- 7) Parameters for opclasses
http://www.sai.msu.su/~megera/postgres/talks/opclass_pgconf.ru-2018.pdf
- 8) IMDB tables: <http://www.sai.msu.su/~megera/postgres/files/imdb/imdb/>



Original idea by Oleg Bartunov,
final implementation by Denis Rosa

ALL

YOU

NEED
POSTGRES

US



ADDENDUM

GSON - Generic JSON

API

JSON API

- **Json (replacement of Jsonb)**

```
typedef struct Json
{
    CompressedDatum    datum; /* original datum */
    JsonContainerData root;  /* root container (like Jsonb.root) */
} Json;
```

- **JsonContainer (replacement of JsonbContainer)**

```
typedef struct JsonContainerData
{
    JsonContainerOps *ops;    /* container vtable */

    /* fields extracted from container by its init() method */
    int             size;    /* number of object fields or array elements */
    JsonValueType   type;    /* array, object or scalar */

    /* container-specific data (binary jsonb, json text, JsonValue *) */
    void           *data[FLEXIBLE_ARRAY_MEMBER];
} JsonContainer;
```

JSON API – Container methods

- JSON container methods

```
struct JsonContainerOps
{
    int             dataSize; /* size of JsonContainer.data[] */
    void            (*init)(JsonContainerData *cont, Datum value);
    JsonIterator   *(*iteratorInit)(JsonContainer *cont);
    JsonValue      *(*findKeyInObject)(JsonContainer *obj, const char *key, int len);
    JsonValue      *(*findValueInArray)(JsonContainer *arr, const JsonValue *val);
    JsonValue      *(*getArrayElement)(JsonContainer *arr, uint32 index);
    uint32         (*getArraySize)(JsonContainer *arr);
    char           *(*toString)(StringInfo out, JsonContainer *, int estimatedlen);
    JsonContainer *(*copy)(JsonContainer *src);
};
```

- Three implementations:

jsonbContainerOps – for jsonb datatype

jsontContainerOps – for json datatype

jsonvContainerOps – for in-memory tree-like JsonValue

JSON API - Container methods

- `void init(JsonContainerData *cont, Datum value)`
Initialize JsonContainerData.data from datum, filling size and type fields.
- `JsonIterator *iteratorInit(JsonContainer *cont)`
Initialize JsonIterator for container (`JsonbIteratorInit()`)
- `JsonValue *findKeyInObject(JsonContainer *obj, const char *key, int len)`
Find key in object (`findJsonbValueFromContainer()`)
- `JsonValue *findValueInArray(JsonContainer *arr, const JsonValue *val)`
Find element in array (`findJsonbValueFromContainer()`)
- `JsonValue *getArrayElement(JsonContainer *arr, uint32 index)`
Extract array element by index (`getIthJsonbValueFromContainer()`)

JSON API - Container methods

- `uint32 getArraySize(JsonContainer *arr)`

Fetch array size if it is unknown (`JsonContainer.size == -1`). Used only for text JSONs.

- `char *toString(StringInfo out, JsonContainer *, int estimatedlen)`

Convert JSON to text representation.

`jsonb`, `jsonv`: called `JsonbToString()` which iterates through JSON
`jsont`: simply return text as is

- `JsonContainer *copy(JsonContainer *src)`

Copy container data.

`jsont`, `jsonb` — copy underlying flat binary data
`jsonv` — copy `JsonValue` tree

JSON API — support macros

JSON container methods are called through a set of macros:

```
JsonIteratorInit(js)
JsonFindKeyInObject(js, key)
JsonFindValueInArray(js, val)
JsonGetArrayElement(js, index)
JsonGetArraySize(js)
JsonCopy(js)
JsonToString(js, out)
```

Their definitions are trivial:

```
#define JsonFindKeyInObject(js, key) \
((js)->ops->findKeyInObject(js, key))
```

JSON API - Iterators

- Iterator API is absolutely similar to JsonbIterators

```
/* analogue of JsonbIteratorNext() */
typedef JsonIteratorToken (*JsonIteratorNextFunc)(JsonIterator **iterator,
                                                JsonValue *value,
                                                 bool skipNested);

/* analogue of JsonbIterator */
struct JsonIteratorData
{
    JsonIterator          *parent;
    JsonContainer        *container;
    JsonIteratorNextFunc next;
};

#define JsonIteratorNext(it, value, skipNested) \
    ((*it)->next(it, value, skipNested))
```

JSON API — Wrapping and unwrapping

- Wrapping json[b] datums is done inside DatumGetJsonX() macros:

```
#define DatumGetJsonbP(datum) DatumGetJson(datum, &jsonbContainer0ps)  
#define DatumGetJsonntP(datum) DatumGetJson(datum, &jsonntContainer0ps)
```

DatumGetJson() calls init() method.

JSON API – Wrapping and unwrapping

- Unwrapping and encoding into target datatype is done inside `JsonXGetDatum()` macros (typically right before returning from user functions):

```
#define PG_RETURN_JSONB_P(js) PG_RETURN_DATUM(JsonbPGetDatum(js))
#define PG_RETURN_JSONT_P(js) PG_RETURN_DATUM(JsontPGetDatum(js))

#define JsonbPGetDatum(js) JsonFlatten(JsonUniquify(js), JsonbEncode, \
                                    &jsonbContainerOps)()

#define JsontPGetDatum(js) CStringGetTextDatum(JsonToString(&js->root, NULL))

void JsonbEncode(StringInfoData *buffer, const JsonbValue *val);
```

`JsonFlatten()` calls the specified encoder function in the separate MemoryContext, passing initialized `StringInfo`.

JSON API – User functions

- Old json functions are almost completely removed, jsonb querying and building code is reused. Typical pair of user functions now look like this:

```
Datum
jsonb_object_field(PG_FUNCTION_ARGS)
{
    PG_RETURN_JSONB_P(json_object_field_internal(PG_GETARG_JSONB_P(0),
                                                PG_GETARG_TEXT_PP(1)));
}
```

```
Datum
json_object_field(PG_FUNCTION_ARGS)
{
    PG_RETURN_JSONT_P(json_object_field_internal(PG_GETARG_JSONT_P(0),
                                                PG_GETARG_TEXT_PP(1)));
}
```

```
static Json *
json_object_field_internal(Json *js, text *key) { ... }
```

JSON API – Uniquification of JSON objects

- Reusing jsonb code for JSON building, it is important to conditionally disable uniquification of object keys for json type that is done in `pushJsonbValue(WJB_END_OBJECT)`.
- Added **uniquified** flag to `JsonValue.val.object`, which is set in constructors after `pushJsonbValue(WJB_BEGIN_OBJECT)`.
- Also added several **uniquified** flags to `JsonValue.val.array`, `JsonValue.val.object`, `JsonValue.val.binary` structures, which determines key uniqueness of all their underlying subobjects.
- `JsonUniquify()` checks these flags and recursively uniquifies subobjects if necessary.

JSON API — Preserving formatting of text JSONs

- Reusing jsonb code for JSON building, it also is important to preserve differences in formatting of json/jsonb:

```
SELECT json_build_object ('a', 'b', 'c', ' [ 1]')::json;
{"a" : "b", "c" : [ 1]}
```

```
SELECT jsonb_build_object('a', 'b', 'c', ' [ 1]')::json;
{"a": "b", "c": [1]}
```

Json preserves formatting of its json parts, json object constructors add space before colon, json_agg() uses newline as array element separator.

- Several separator char fields added to JsonValue.val.array and JsonValue.val.object. They are initialized in builder functions and used in JsonToString().
- pushJsonbValue() now does not unpack binary jsons to preserve their formatting.



obartunov@postgrespro.ru