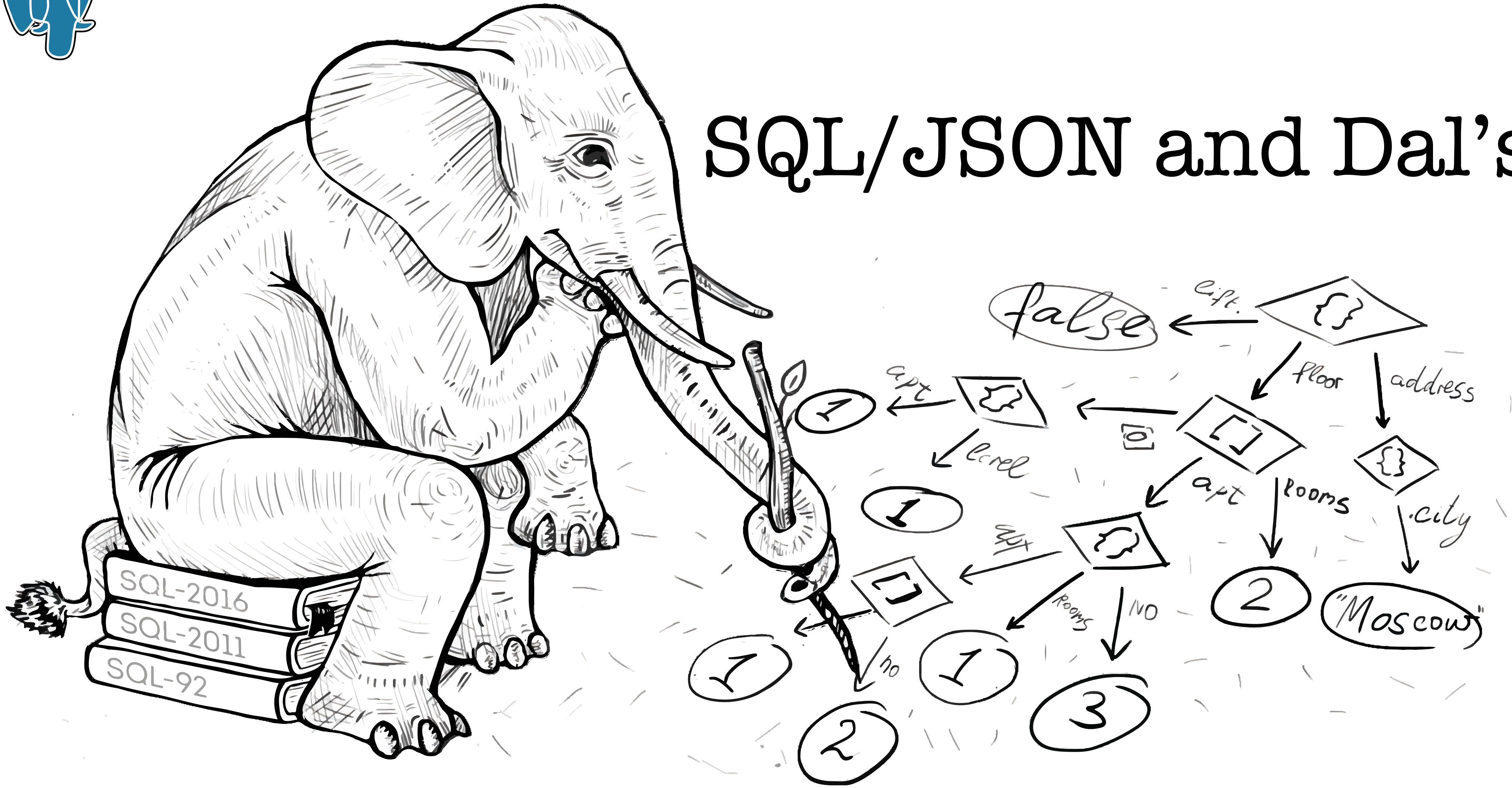
SQL/JSON and Dal’she

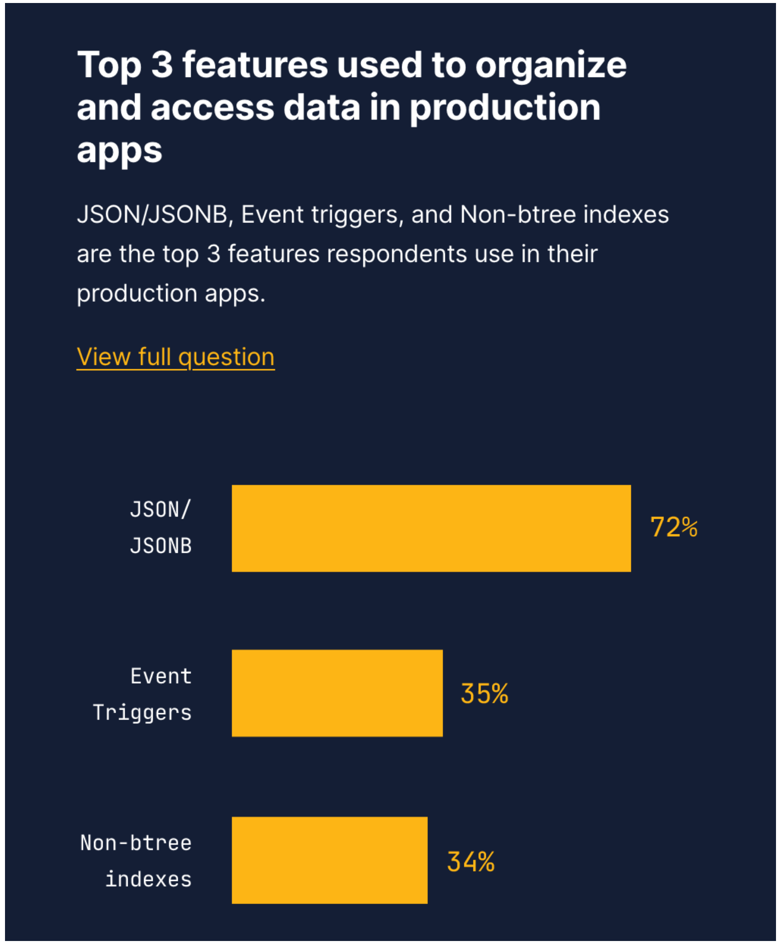


Oleg Bartunov, Nikita Glukhov

Why this talk ?

```
CREATE JSON products (
   ..................
);
```

- 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
    - Top-1 feature of Postgres used in production
    - 3rd popular topic in https://t.me/pgsql (8300+ members)

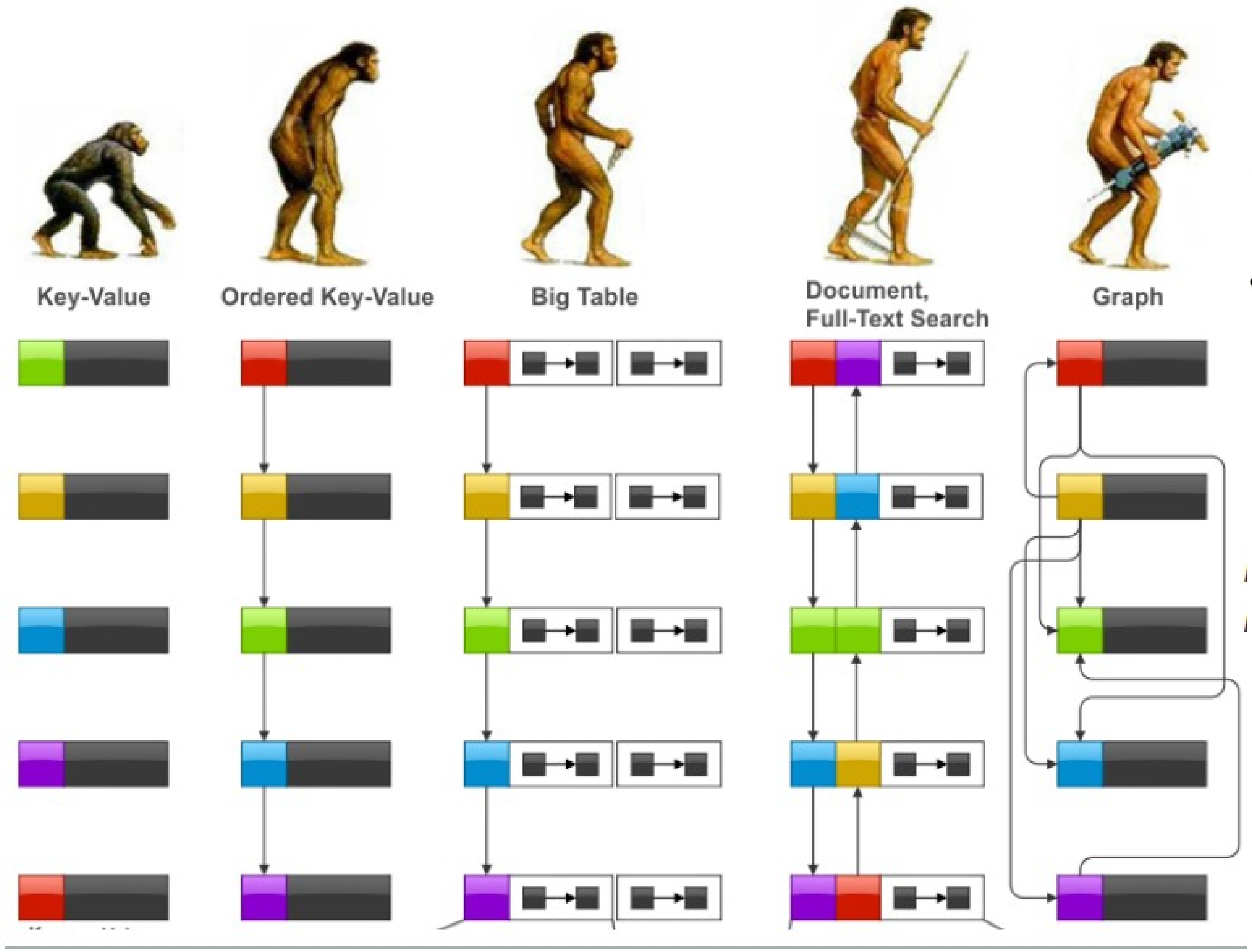


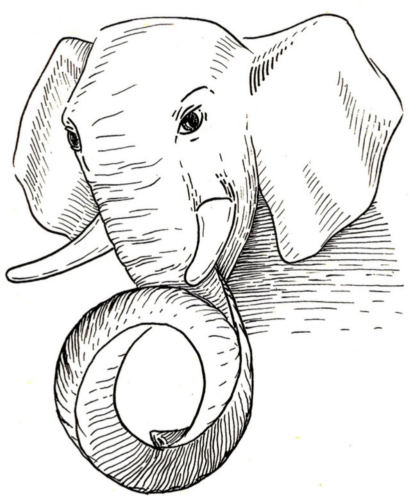
- Results of our experiments in 2021-2022:

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

NOSQL POSTGRES IN SHORT





SQL/JSON — PG15(2022)
- Complete SQL/JSON
- Better indexing, syntax

JSONPATH - 2019
- SQL/JSON — 2016
- Functions & operators
- Indexing

JSONB - 2014
- Binary storage
- Nesting objects & arrays
- Indexing

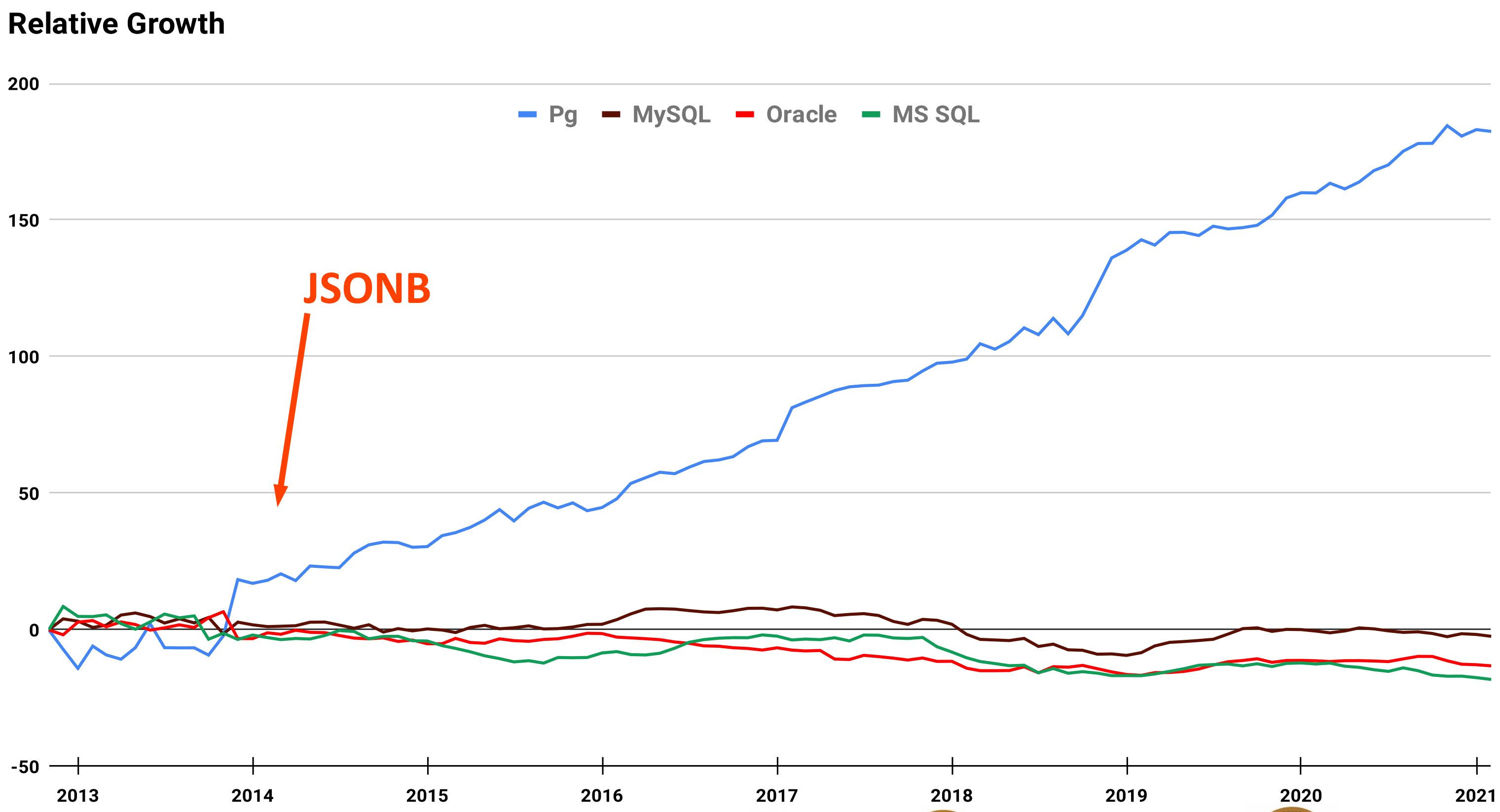
JSON - 2012
- Textual storage
- JSON verification

HSTORE - 2003
- Perl-like hash storage
- No nesting, no arrays
- Indexing

PostgresPro

Postgres breathed a second life into relational databases

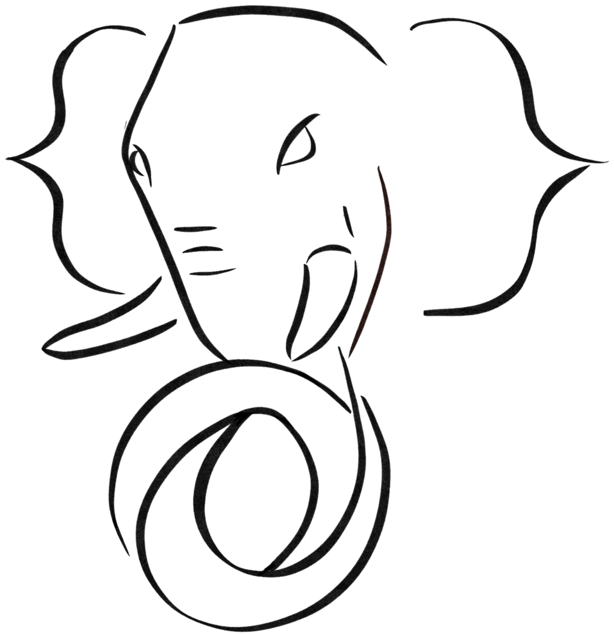
- Postgres innovation - the first relational database with NoSQL support
- NoSQL Postgres attracts the NoSQL users
- JSON became a part of SQL Standard 2016



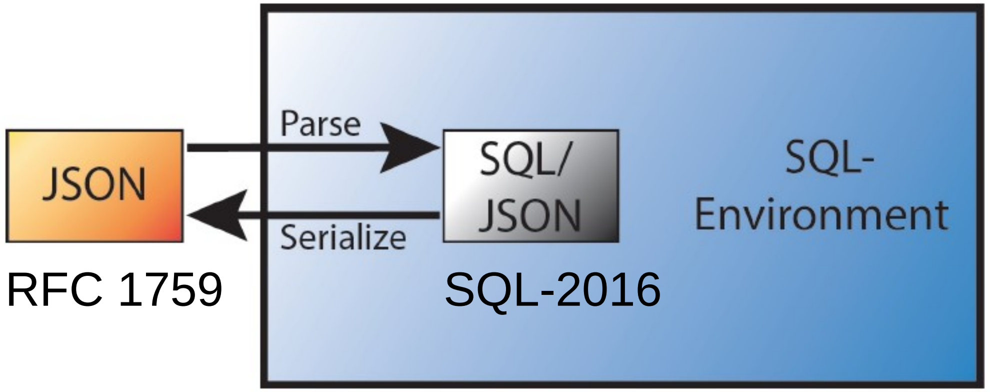
db-engines.com/en/ranking

PG15: SQL/JSON/TABLE (#postgrespro)

Tnanks, Andrew Dunstan for committing !



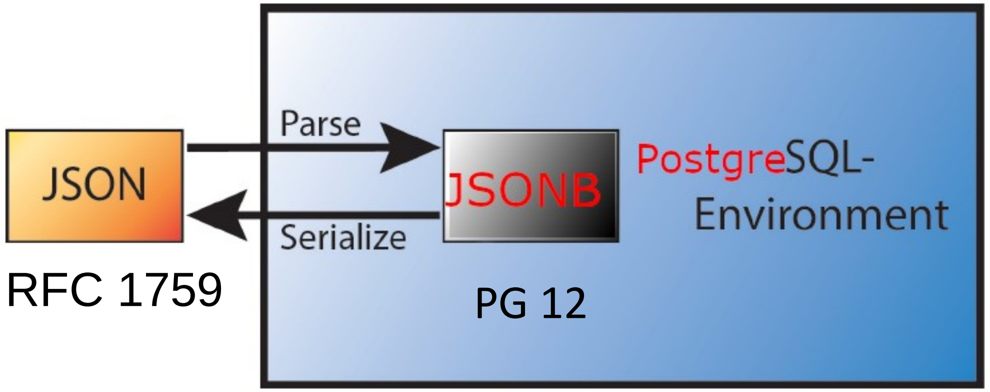
SQL/JSON in SQL-2016



- 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
  - 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

PostgresPro

SQL/JSON in PostgreSQL



- SQL/JSON data model
  - **Jsonb is the (practical) subset of SQL/JSON data model with ORDERED and UNIQUE KEYS**
- JSON Path language - **Committed into PG12,** June 19, 2019
  - Describes a <projection> of JSON data (to be used by SQL/JSON functions)
  - **The most complete and best implementation (15/15 features)**
  - Impemented as **jsonpath** data type (binary)
- SQL/JSON functions - **Committed into PG15 (September of 2022) !**
  - Constructor 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
- Indexes
  - **GIN opclasses for jsonb,** more in Jsquery extension

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)'`

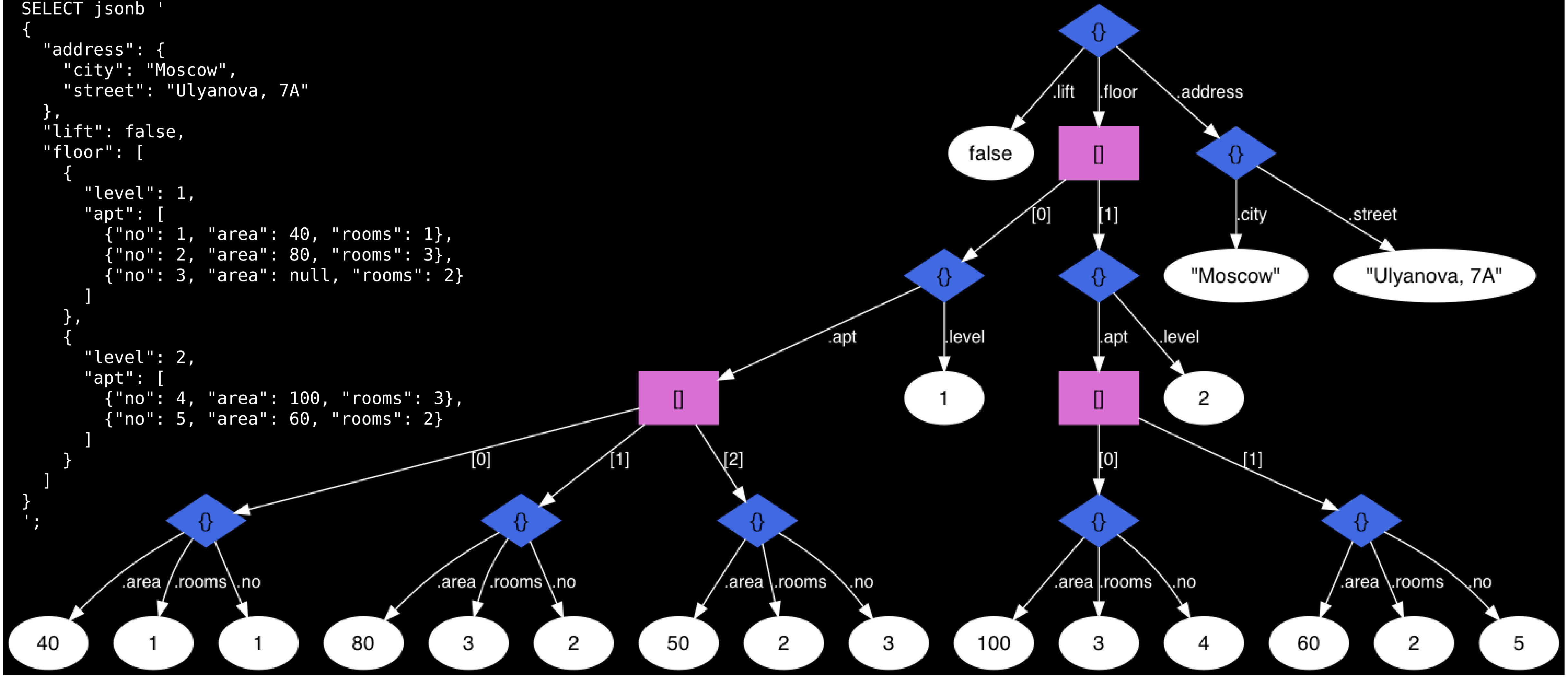
    `@ - current context in filter expression`
  - `Item methods - '$.a.b.c.x.type()'`

    `type(), size(), double(), ceiling(), floor(), abs(), keyvalue(), datetime()`

PostgresPro

Example: Two floors house

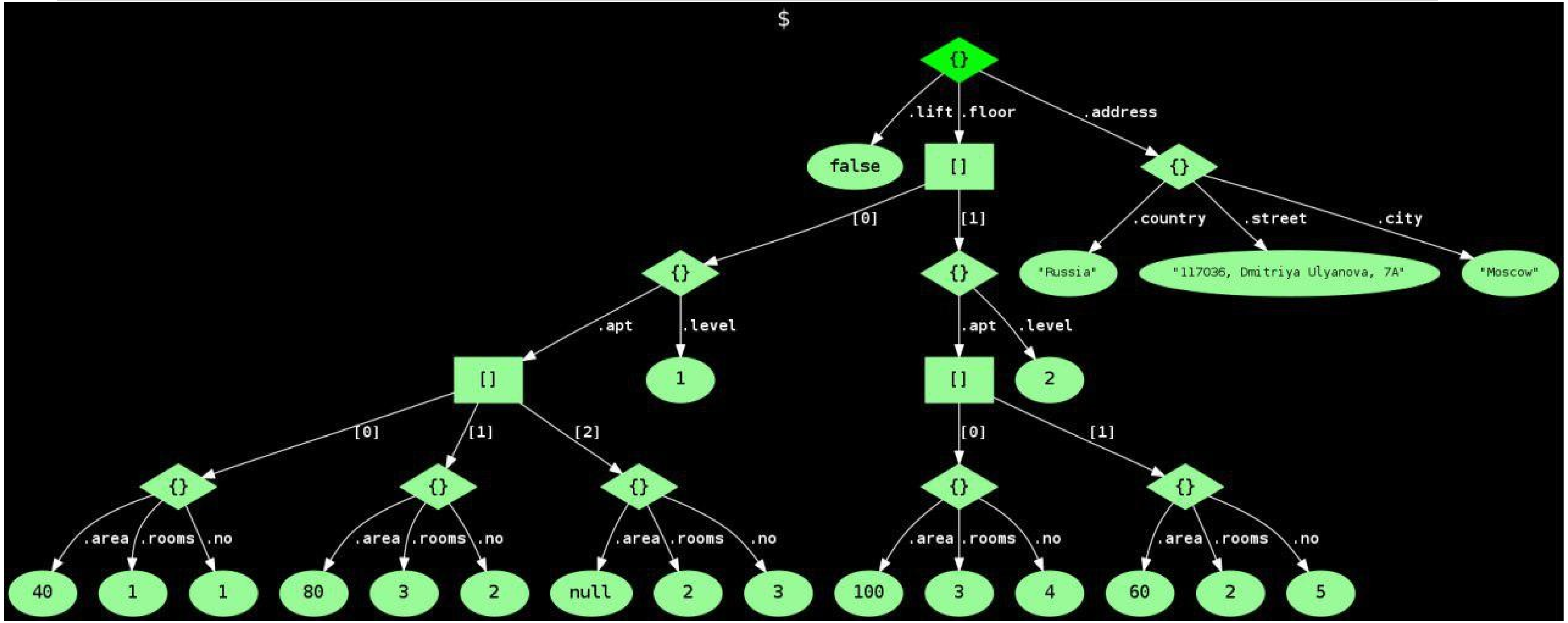
```
CREATE TABLE house(js) AS
SELECT jsonb '
{
  "address": {
    "city": "Moscow",
    "street": "Ulyanova, 7A"
  },
  "lift": false,
  "floor": [
    {
      "level": 1,
      "apt": [
        {"no": 1, "area": 40, "rooms": 1},
        {"no": 2, "area": 80, "rooms": 3},
        {"no": 3, "area": null, "rooms": 2}
      ]
    },
    {
      "level": 2,
      "apt": [
        {"no": 4, "area": 100, "rooms": 3},
        {"no": 5, "area": 60, "rooms": 2}
      ]
    }
  ]
}
';
```



PostgresPro

How path expression works (1)

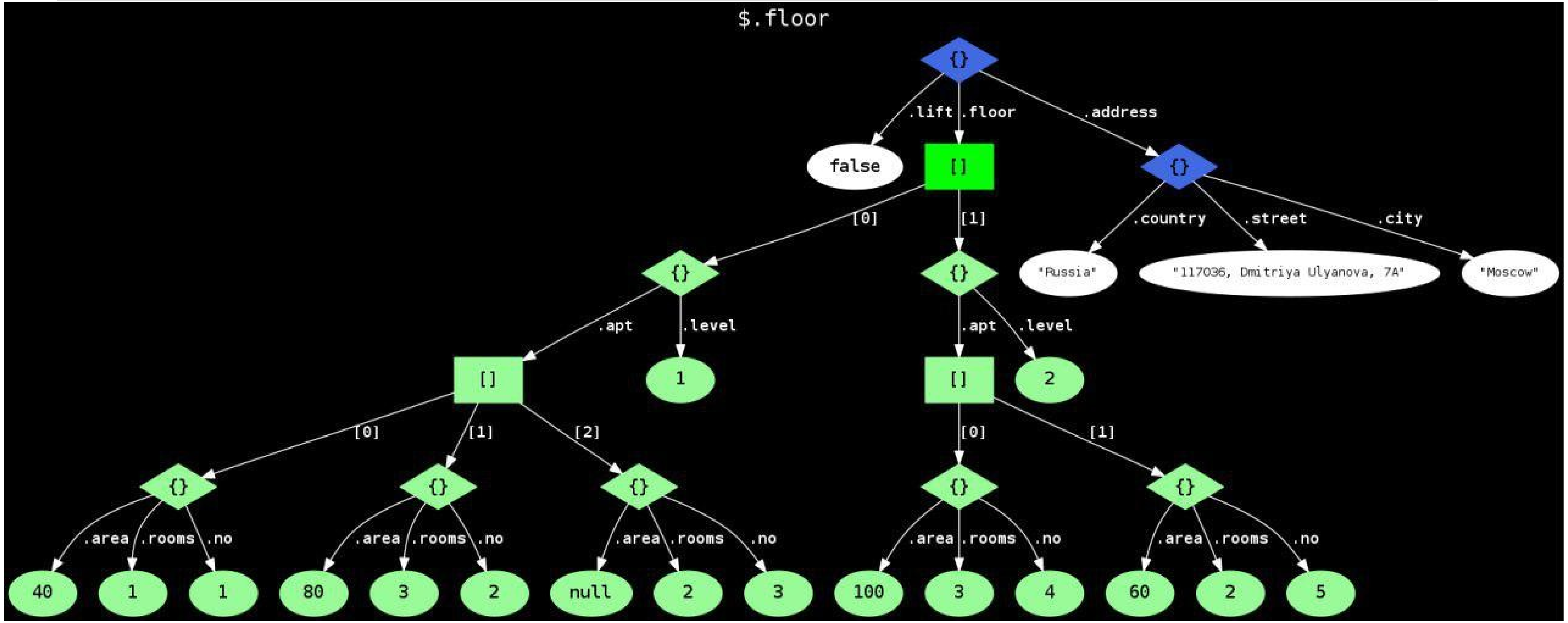
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'



PostgresPro

How path expression works (2)

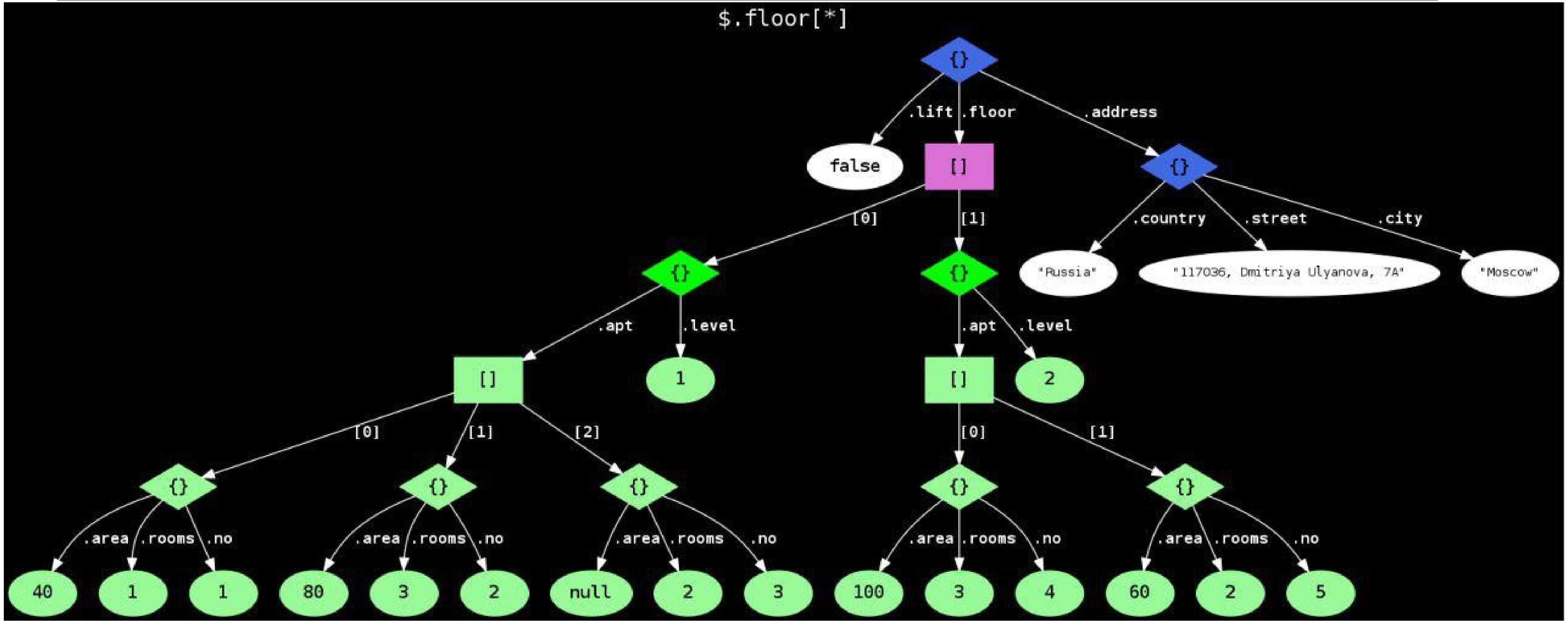
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'



PostgresPro

How path expression works (3)

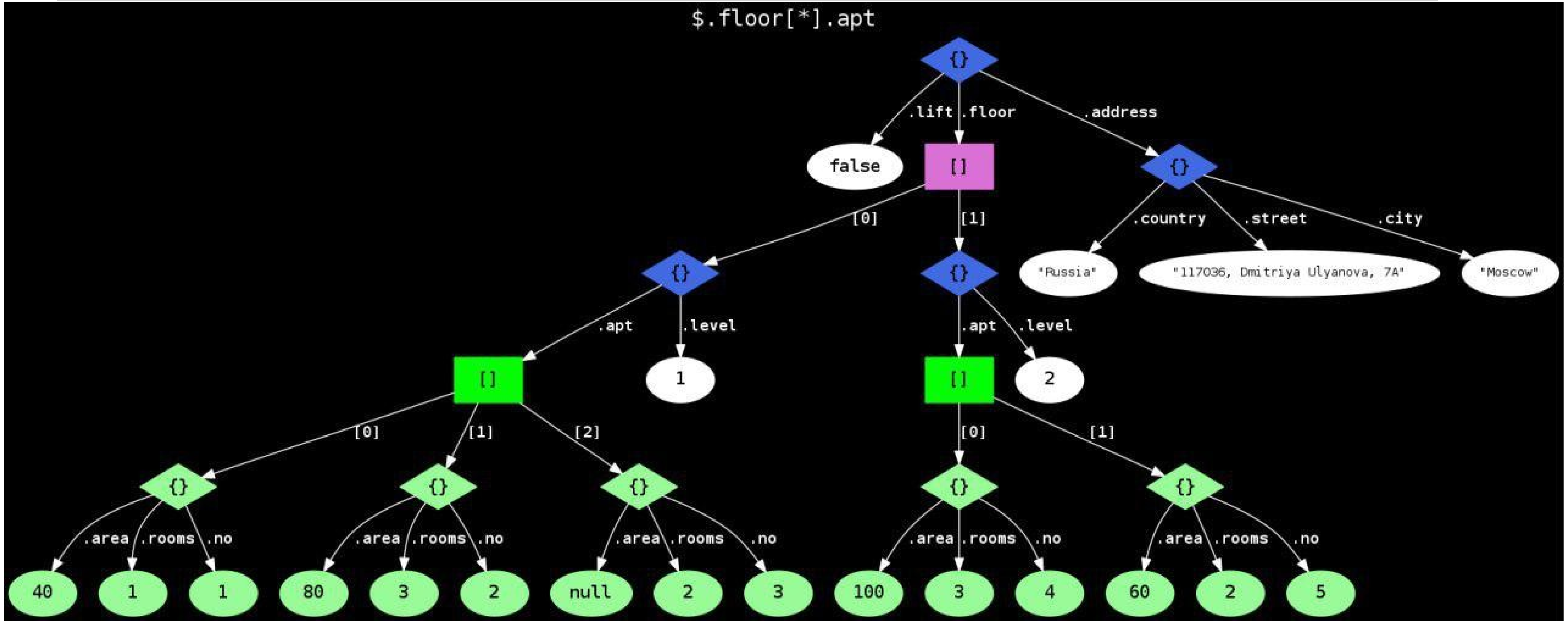
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'



PostgresPro

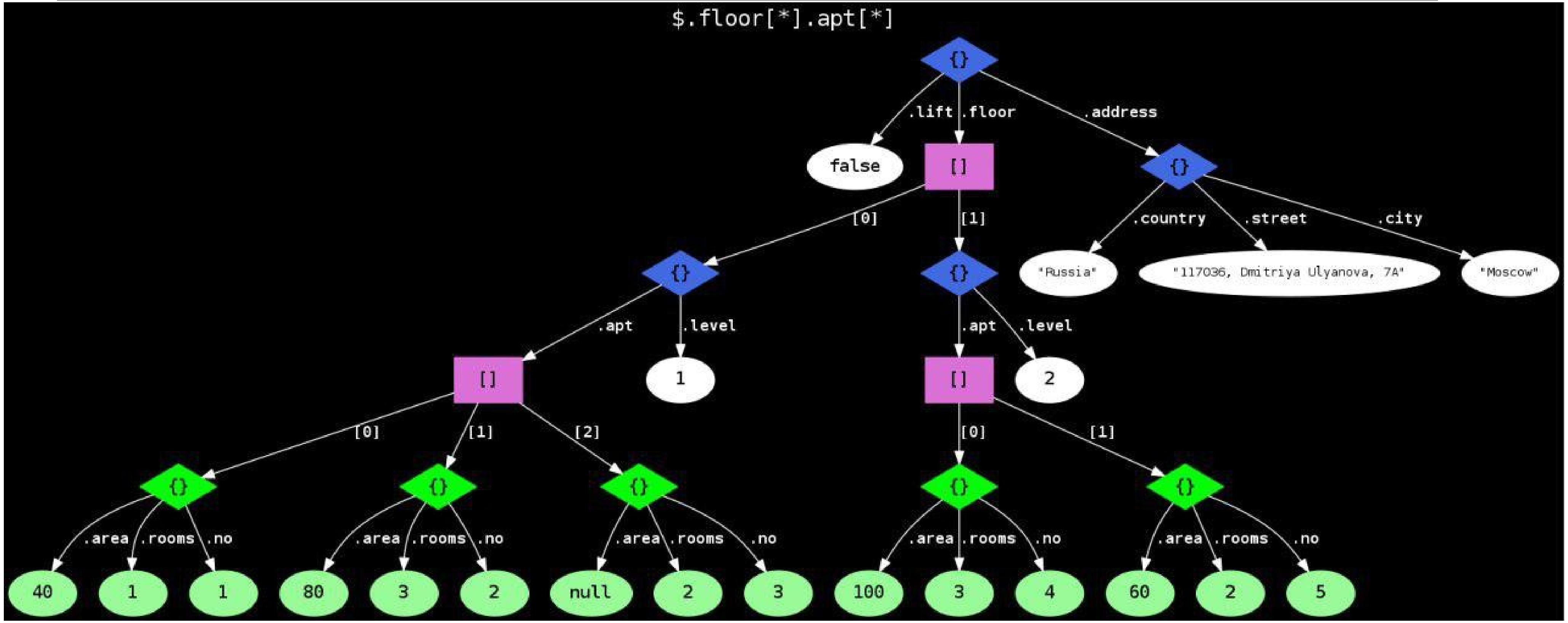
How path expression works (4)

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



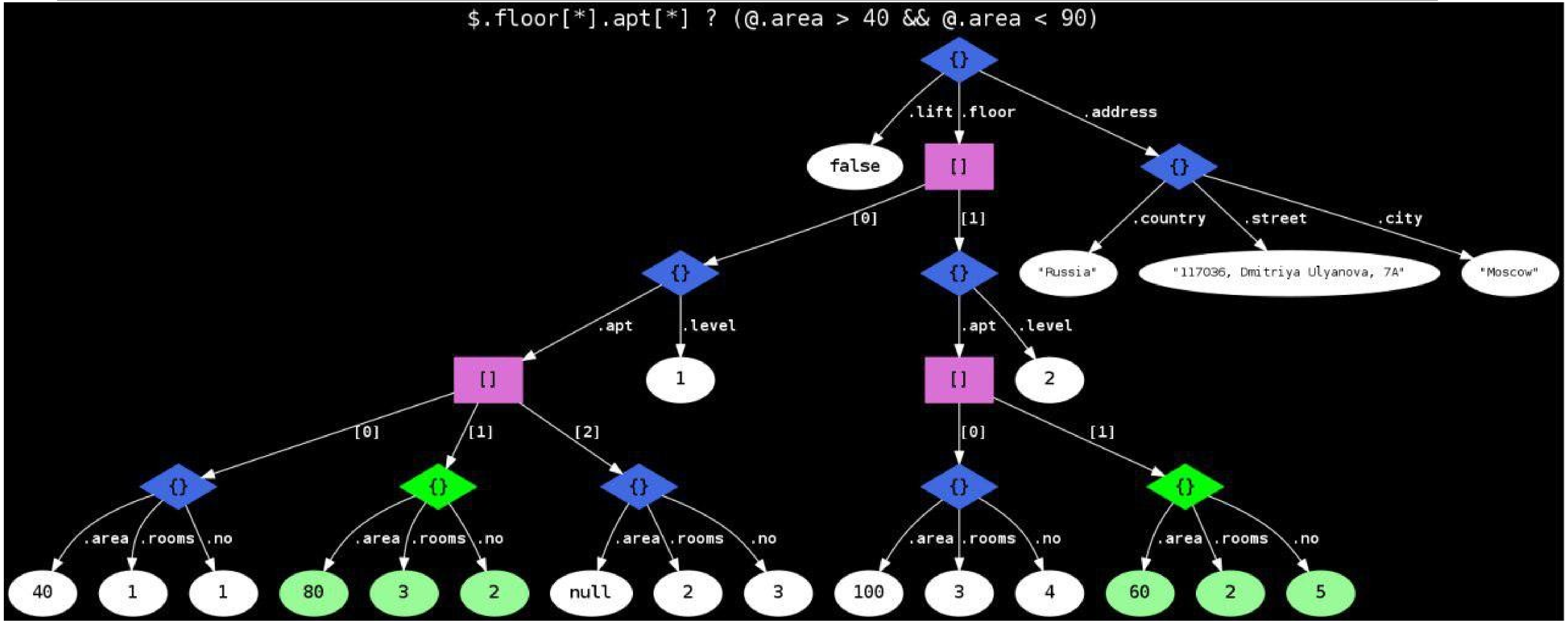
How path expression works (5)

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



How path expression works (6)

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



How path expression works

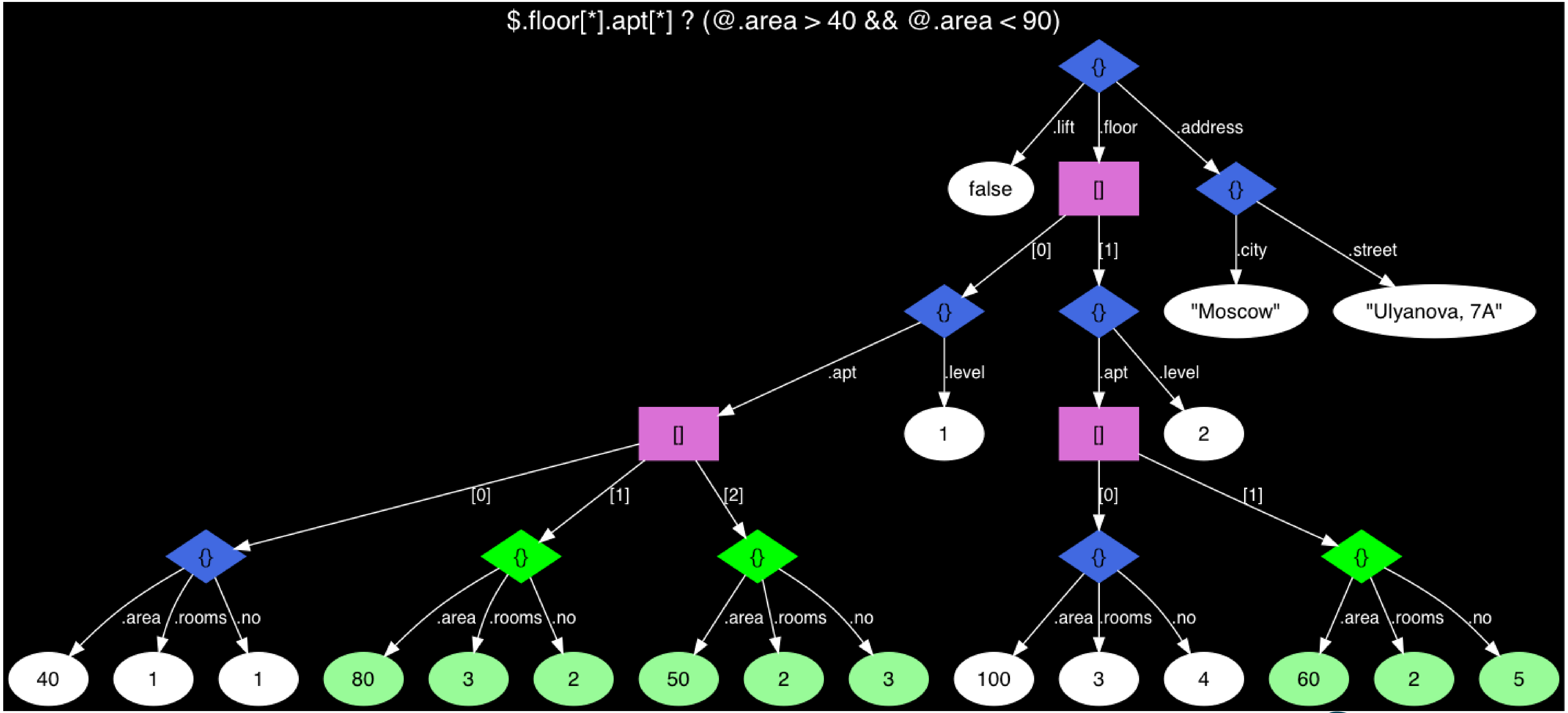
'$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)'

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

The result is a sequence of two SQL/JSON items

PostgresPro

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



PostgresPro

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

- PG12+ (jsonpath) query

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

  - More consise than plain SQL, better performance for complex queries

- 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;
```

PostgresPro

SQL/JSON Functions

- First discussed at developers meeting@Fosdem, Jan 28, 2017 in Brussels
- Post in -hackers, Feb 28, 2017, March CommitFest
- Committed March, 2022 !

"...Nikita Glukhov (who probably deserves an award for perseverance)..."

SQL/JSON in PostgreSQL

Lists:pgsql-hackers

From:	Oleg Bartunov <obartunov(at)gmail(dot)com>
To:	Pgsql Hackers <pgsql-hackers(at)postgresql(dot)org>, Nikita Glukhov <n(dot)gluhov(at)postgrespro(dot)ru>, Teodor Sigaev <teodor(at)postgrespro(dot)ru>, Alexander Korotkov <a(dot)korotkov(at)postgrespro(dot)ru>, andrew Dunstan <andrew(at)
Subject:	SQL/JSON in PostgreSQL
Date:	2017-02-28 19:08:43
Message-ID:	CAF4Au4w2x-5LTnN_bxky-mq4=WOqsGsxSpENCzHRAzSnEd8+WQ@mail.gmail.com
Views:	Raw Message | Whole Thread | Download mbox | Resend email
Lists:	pgsql-hackers

Hi there,

Attached patch is an implementation of SQL/JSON data model from SQL-2016 standard (ISO/IEC 9075-2:2016(E)), which was published 2016-12-15 and is available only for purchase from ISO web site (https://www.iso.org/standard/63556.html). Unfortunately I didn't find any public sources of the standard or any preview documents, but Oracle implementation of json support in 12c release 2 is very close (http://docs.oracle.com/database/122/ADJSN/json-in-oracle-database.htm), also we used https://livesql.oracle.com/ to understand some details.

Postgres has already two json data types - json and jsonb and implementing another json data type, which strictly conforms the standard, would be not a good idea. Moreover, SQL standard doesn't describe data type, but only data model, which "comprises SQL/JSON items and SQL/JSON sequences. The components of the SQL/JSON data model are:

1) An SQL/JSON item is defined recursively as any of the following:

a) An SQL/JSON scalar, defined as a non-null value of any of the following predefined (SQL) types:

SQL/JSON in PostgreSQL

- The SQL/JSON **construction** functions : values of SQL types to JSON data

  Mostly the same as json[b] construction functions

  - JSON - generates JSON[b] from text data (::json[b])
  - JSON_SCALAR — generates a JSON[b] scalar value from SQL data (to_json[b])
  - 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: JSON

JSON - generates JSON[b] from text data (something like cast to json/jsonb types)

Syntax:

```
JSON (
  expression [ FORMAT JSON [ ENCODING UTF8 ] ]
  [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
  [ RETURNING json_data_type ]
)
```

PostgresPro

SQL/JSON: JSON

JSON - generates JSON[b] from text data (something like cast to json/jsonb types)

```
=# SELECT JSON('{"a": 1, "a": 2}' RETURNING JSON), JSON('{"a": 1, "a": 2}' RETURNING JSONB) as jsonb;
      json        |  jsonb
------------------+----------
 {"a": 1, "a": 2} | {"a": 2}
(1 row)

=# SELECT JSON('{"a": 1, "a": 2}' RETURNING JSON), JSON('{"a": 1, "a": 2}'WITH UNIQUE KEYS  RETURNING JSONB) AS  jsonb;
ERROR:  duplicate JSON object key value
```

PostgresPro

SQL/JSON: JSON_SCALAR

JSON_SCALAR - generates a JSON[b] scalar value from SQL data (to_json[b])

Syntax:

```
JSON_SCALAR (
   expression
   [ RETURNING json_data_type ]
)
```

```
=# SELECT JSON_SCALAR(1), JSON_SCALAR('1'), JSON_SCALAR(NULL), JSON_SCALAR(false);
 json_scalar | json_scalar | json_scalar | json_scalar
-------------+-------------+-------------+-------------
 1           | "1"         | (null)      | false
(1 row)
```

PostgresPro

SQL/JSON: JSON_OBJECT

JSON_OBJECT -construct a JSON[b] object from SQL or JSON data

Syntax:

```
JSON_OBJECT (
  [ { key_expression { VALUE | ':' }
      value_expression [ FORMAT JSON [ ENCODING UTF8 ] ] }[, ...] ]
  [ { NULL | ABSENT } ON NULL ]
  [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
  [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ]
)
key_expression ::=   expression { VALUE | ':' }
value_expression ::= expression [ FORMAT JSON [ ENCODING UTF8 ] ]
```

PostgresPro

SQL/JSON: JSON_OBJECT

- Internally transformed into a json[b]_build_object() call
- RETURNING type:
  - Json by default
  - can be json, jsonb, string type, bytea or having cast from json
  - determines which function to use:

    jsonb => jsonb_build_object

    other => json_build_object
- There are two additional options:
  - key uniqueness check: `{WITH|WITHOUT} UNIQUES [KEYS]`
  - ability to omit keys with NULL values: `{ABSENT|NULL} ON NULL`

SQL/JSON: JSON_OBJECT

Key uniqueness check (disabled by default):

```
SELECT JSON_OBJECT('a': 1, 'a': 2 WITH UNIQUE KEYS);
ERROR: duplicate JSON key "a"

SELECT JSON_OBJECT('a': 1, 'a': 2);
      ?column?
--------------------
 {"a" : 1, "a" : 2}
(1 row)

SELECT JSON_OBJECT('a': 1, 'a': 2 RETURNING jsonb);
 ?column?
----------
 {"a": 2}
(1 row)
```

SQL/JSON: JSON_OBJECT

Omitting keys with NULL values (keys themselves are not allowed to be NULL):

```
SELECT JSON_OBJECT('a': 1, 'b': NULL);
       ?column?
-----------------------
 {"a" : 1, "b" : null}
(1 row)

SELECT JSON_OBJECT('a': 1, 'b': NULL ABSENT ON NULL);
 ?column?
-----------
 {"a" : 1}
(1 row)
```

PostgresPro

SQL/JSON: JSON_OBJECTAGG

JSON_OBJECTAGG - aggregates name/value pairs as JSON[b] object.

Syntax:

```
JSON_OBJECTAGG (
  [ { key_expression { VALUE | ':' } value_expression } ]
  [ { NULL | ABSENT } ON NULL ]
  [ { WITH | WITHOUT } UNIQUE [ KEYS ] ]
  [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ] ]
)
```

Options and RETURNING clause are the same as in JSON_OBJECT.

SQL/JSON: JSON_OBJECTAGG

JSON_OBJECTAGG is transformed into a json[b]_object_agg depending on RETURNING type.

```
=# SELECT JSON_OBJECTAGG('key' || i : 'val' ||  i)
   FROM generate_series(1, 3) i;

                       ?column?
-------------------------------------------------------
 { "key1" : "val1", "key2" : "val2", "key3" : "val3" }
(1 row)

=# SELECT pg_typeof(JSON_OBJECTAGG('key' || i : 'val' ||  i RETURNING JSONB))
   FROM generate_series(1, 3) i;
 pg_typeof
-----------
 jsonb
(1 row)
```

SQL/JSON: JSON_ARRAY

JSON_ARRAY - construct a JSON[b] array from SQL or JSON data

```
JSON_ARRAY (
  [ { expression [ FORMAT JSON ] }[, ...] ]
  [ { NULL | ABSENT } ON NULL ]
  [ RETURNING data_type [ FORMAT JSON[ ENCODING UTF8 ]] ]
)

JSON_ARRAY (
  query_expression
  [ RETURNING data_type [FORMAT JSON [ ENCODING UTF8] ] ]
)
```

Note: ON NULL clause is not supported in subquery variant.

PostgresPro

SQL/JSON: JSON_ARRAY

- Internally transformed into a json[b]_build_array() call
- RETURNING type:
  - json by default
  - can be json, jsonb, string type, bytea or having cast from json
  - determines which function to use:

    jsonb => jsonb_build_array

    other => json_build_array
- There is one additional option:
  - The ability to omit or keep elements with NULL values: `{ABSENT|NULL} ON NULL`

SQL/JSON: JSON_ARRAY

```
=# SELECT JSON_ARRAY('string', NULL, TRUE, ARRAY[1,2,3],
  '{"a": 1}'::jsonb, '[1, {"c": 3}]' FORMAT JSON); -- ABSENT ON NULL is by default
                    ?column?
-----------------------------------------------------------
 ["string", true, [1,2,3], {"a": 1}, [1, {"c": 3}]]

=# SELECT JSON_ARRAY('string', NULL, TRUE, ARRAY[1,2,3],
  '{"a": 1}'::jsonb, '[1, {"c": 3}]' FORMAT JSON NULL ON NULL);
                        json_array
------------------------------------------------------------
 ["string", null, true, [1, 2, 3], {"a": 1}, [1, {"c": 3}]]

=# SELECT JSON_ARRAY(SELECT * FROM generate_series(1, 3));
 ?column?
-----------
 [1, 2, 3]
```

SQL/JSON: JSON_ARRAYAGG

JSON_ARRAYAGG - aggregates SQL values into a JSON[b] array.

Syntax:

```
JSON_ARRAYAGG (
  [ value_expression ]
  [ ORDER BY sort_expression ]
  [ { NULL | ABSENT } ON NULL ]
  [ RETURNING data_type [ FORMAT JSON [ ENCODING UTF8 ] ]
]
)
```

SQL/JSON: JSON_ARRAYAGG

All is the same as in JSON_ARRAY except that JSON_ARRAYAGG is transformed into a json[b]_agg() call.

```
=# SELECT JSON_ARRAYAGG(i) FROM generate_series(1, 3) i;
 ?column?
-----------
 [1, 2, 3]
(1 row)
```

PostgresPro

SQL/JSON in PostgreSQL

- The SQL/JSON **retrieval** functions:
  - JSON_VALUE - Extract an SQL/JSON value from JSON data and return it as SQL scalar of specified type.
  - JSON_QUERY - Extract an SQL/JSON array or object from JSON data and returns JSON string
  - JSON_TABLE - Query a JSON text and present it as a relational table.
  - IS [NOT] JSON - test whether a string value is a valid JSON text.
  - JSON_EXISTS - test whether a JSON path expression returns any SQL/JSON items
- Supported only JSONB ! Need GSON (generalized JSON API) to support JSON and JSONB without code complication.

SQL/JSON examples: JSON_VALUE

```
SELECT x, JSON_VALUE(jsonb '{"a": 1, "b": 2}','$.* ? (@ > $x)' PASSING x AS x
                     RETURNING int
                     DEFAULT -1 ON EMPTY
                     DEFAULT -2 ON ERROR
                    ) y
FROM
generate_series(0, 2) x;
 x | y
---+----
 0 | -2
 1 |  2
 2 | -1
(3 rows)
```

PostgresPro

SQL/JSON examples: JSON_QUERY

```
SELECT
        JSON_QUERY(js::jsonb, '$'),
        JSON_QUERY(js::jsonb, '$' WITHOUT WRAPPER),
        JSON_QUERY(js::jsonb, '$' WITH CONDITIONAL WRAPPER),
        JSON_QUERY(js::jsonb, '$' WITH UNCONDITIONAL ARRAY WRAPPER),
        JSON_QUERY(js::jsonb, '$' WITH ARRAY WRAPPER)
FROM
        (VALUES
                ('null'),
                ('12.3'),
                ('true'),
                ('"aaa"'),
                ('[1, null, "2"]'),
                ('{"a": 1, "b": [2]}')
        ) foo(js);
     ?column?        |      ?column?       |      ?column?       |       ?column?       |       ?column?
---------------------+---------------------+---------------------+----------------------+----------------------
 null                | null                | [null]              | [null]               | [null]
 12.3                | 12.3                | [12.3]              | [12.3]               | [12.3]
 true                | true                | [true]              | [true]               | [true]
 "aaa"               | "aaa"               | ["aaa"]             | ["aaa"]              | ["aaa"]
 [1, null, "2"]      | [1, null, "2"]      | [1, null, "2"]      | [[1, null, "2"]]     | [[1, null, "2"]]
 {"a": 1, "b": [2]}  | {"a": 1, "b": [2]}  | {"a": 1, "b": [2]}  | [{"a": 1, "b": [2]}] | [{"a": 1, "b": [2]}]
(6 rows)
```

PostgresPro

SQL/JSON examples: Constraints

```
CREATE TABLE test_json_constraints (
       js text,
       i int,
       x jsonb DEFAULT JSON_QUERY(jsonb '[1,2]', '$[*]' WITH WRAPPER)
       CONSTRAINT test_json_constraint1
              CHECK (js IS JSON)
       CONSTRAINT test_json_constraint2
CHECK (JSON_EXISTS(js::jsonb, '$.a' PASSING i + 5 AS int, i::text AS txt))
       CONSTRAINT test_json_constraint3
CHECK (JSON_VALUE(js::jsonb, '$.a' RETURNING int DEFAULT ('12' || i)::int
       ON EMPTY ERROR ON ERROR) > i)
       CONSTRAINT test_json_constraint4
              CHECK (JSON_QUERY(js::jsonb, '$.a'
WITH CONDITIONAL WRAPPER EMPTY OBJECT ON ERROR) < jsonb '[10]')
);
```

PostgresPro

SQL/JSON examples: JSON_TABLE

- Creates a relational view of JSON data.
- Think about UNNEST — creates a row for each object inside JSON array and represent JSON values from within that object as SQL columns values.
- Build on top of XML_TABLE infrastructure (PG 10)

SQL/JSON examples: JSON_TABLE

Floors in relational form.

```
SELECT
  apt.*
FROM
  house,
  JSON_TABLE(
    js, '$.floor[*]' COLUMNS (
      level int,
      NESTED PATH '$.apt[*]' 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)
```

PostgresPro

JSONB subscripting syntax

- Based on «Generic type subscripting» on commitfest
  https://commitfest.postgresql.org/15/1062/
  Extends array syntax to support other types

```
=# SELECT js['info']['contacts'] FROM house;
                   js
----------------------------------------
 "Postgres Pro\n+7 (495) 150-06-91 ..."
(1 row)
=# UPDATE house SET js['info']['contacts'] = '"Oleg Bartunov"';
UPDATE 1
=# SELECT js['info']['contacts'] FROM house;
       js
-----------------
 "Oleg Bartunov"
(1 row)
```

PostgresPro

JSON_MODIFY

Missing SQL/JSON functionality

PostgresPro

JSON_MODIFY – motivational example

Example: add key “big” = true to all apartments having area greater than 70.

Simple desired jsonpath expression:

```
$.floor[*].apt[*]?(@.area > 70).big = true
```

Complex query with manual unnesting/aggregation:

```
SELECT jsonb_set(js, '{floor}', (
  SELECT jsonb_agg(jsonb_set(floor, '{apt}', (
    SELECT jsonb_agg(
      CASE WHEN jsonb_typeof(apt -> 'area') = 'number'
                AND (apt -> 'area')::int > 70
      THEN apt || '{"big": true}'
      ELSE apt END)
    FROM jsonb_array_elements(floor->'apt') apt)))
  FROM jsonb_array_elements(js->'floor') floor))
FROM house;
```

JSON_MODIFY – motivational example 2

Example: change contacts and street address.

Desired jsonpath expression:

```
$.info.contacts = 'new contacts', $.address.street = 'new address'
```

Ugly query with `jsonb_set()` chaining:

```
SELECT jsonb_set(jsonb_set(js, '{info,contacts}',
to_jsonb('new contacts'::text)), '{address,street}',
to_jsonb('new address'::text))
FROM house;
```

`UPDATE` can be done in more natural way with subscripting syntax:

```
UPDATE house SET js['address']['street'] = to_jsonb('new address'::text),
                 js['info']['contacts']  = to_jsonb('new contacts'::text);
```

JSON_MODIFY – syntax

```
JSON_MODIFY(jsonb_expr, operation, …
            [RETURNING type]
            [PASSING expr AS name, …])
```

```
Operations:
  • SET jsonpath = expr          [… ON EXISTING] [… ON MISSING] [… ON NULL]
  • REPLACE jsonpath = expr      [… ON MISSING]  [… ON NULL]
  • INSERT jsonpath = expr       [… ON EXISTING] [… ON NULL]
  • APPEND jsonpath = expr       [… ON MISSING]  [… ON NULL]
  • REMOVE jsonpath              [… ON MISSING]
  • RENAME jsonpath WITH expr    [… ON MISSING]
  • KEEP jsonpath, …             [… ON MISSING]   (not implemented yet !)
```

JSON_MODIFY – ON behaviors

`ON NULL` – executed when the new value is NULL

- `NULL ON NULL` – use JSON null as new value
- `IGNORE ON NULL` – do nothing
- `ERROR ON NULL` – raise an error
- `REMOVE ON NULL` – remove old value, if exists

`ON EXISTING` – executed when target JSON item exists

- `IGNORE ON EXISTING` – do nothing
- `ERROR ON EXISTING` – raise an error
- `REPLACE ON EXISTING` – replace old value with new value
- `REMOVE ON EXISTING` – remove old value

`ON MISSING` – executed when target JSON item does not exist

- `IGNORE ON MISSING` – do nothing
- `ERROR ON MISSING` – raise an error
- `CREATE ON MISSING` – insert new value

JSON_MODIFY – simplification example

Example: add key “big” = true to all apartments having area greater than 70.

Greatly simplified query using JSON_MODIFY:

```
SELECT JSON_MODIFY(js,
  SET '$.floor[*].apt[*] ? (@.area > $big_area).big' = true
  PASSING 70 AS big_area
)
FROM house;
```

Complex query with manual unnesting/aggregation:

```
SELECT jsonb_set(js, '{floor}', (
  SELECT jsonb_agg(jsonb_set(floor, '{apt}', (
    SELECT jsonb_agg(CASE WHEN (apt -> 'area')::int > 70
                     THEN apt || '{"big": true}'
                     ELSE apt END)
    FROM jsonb_array_elements(floor->'apt') apt)))
  FROM jsonb_array_elements(js->'floor') floor))
FROM house;
```

JSON_MODIFY – simplification example 2

Example: change contacts and street address.

Simplified query using JSON_MODIFY with multiple operations:

```
SELECT JSON_MODIFY(js, SET '$.info.contacts'  = 'new contacts',
                       SET '$.address.street' = 'new address')
FROM house;
```

- There is no need to wrap text values into jsonb, SQL types are automatically mapped to corresponding SQL/JSON item types.
- Multiple operations can be executed in the single pass through jsonb, what can speed up execution by $N_{operation}$ times (not yet implemented). This optimization is not possible in chained function calls.

JSON_MODIFY – simplest **object** case (queries)

```
CREATE TABLE test_object AS
SELECT i id, jsonb_build_object('x', i, 'z', (
    SELECT jsonb_agg(x)
    FROM generate_series(1, (10.0 ^ (i / 10.0))::int) x)) jb
FROM generate_series(1,50) i;
```

SET $.x = 0

- `JSON_MODIFY(jb, SET '$.x' = 0)`
- `jsonb_set(jb, '{x}', '0')`
- `jb || '{"x": 0}'`
- `(SELECT jsonb_object_agg(k, v)`
  `FROM (SELECT k, v FROM jsonb_each(jb) kv(k,v) UNION SELECT 'x', '0') kv(k, v))`

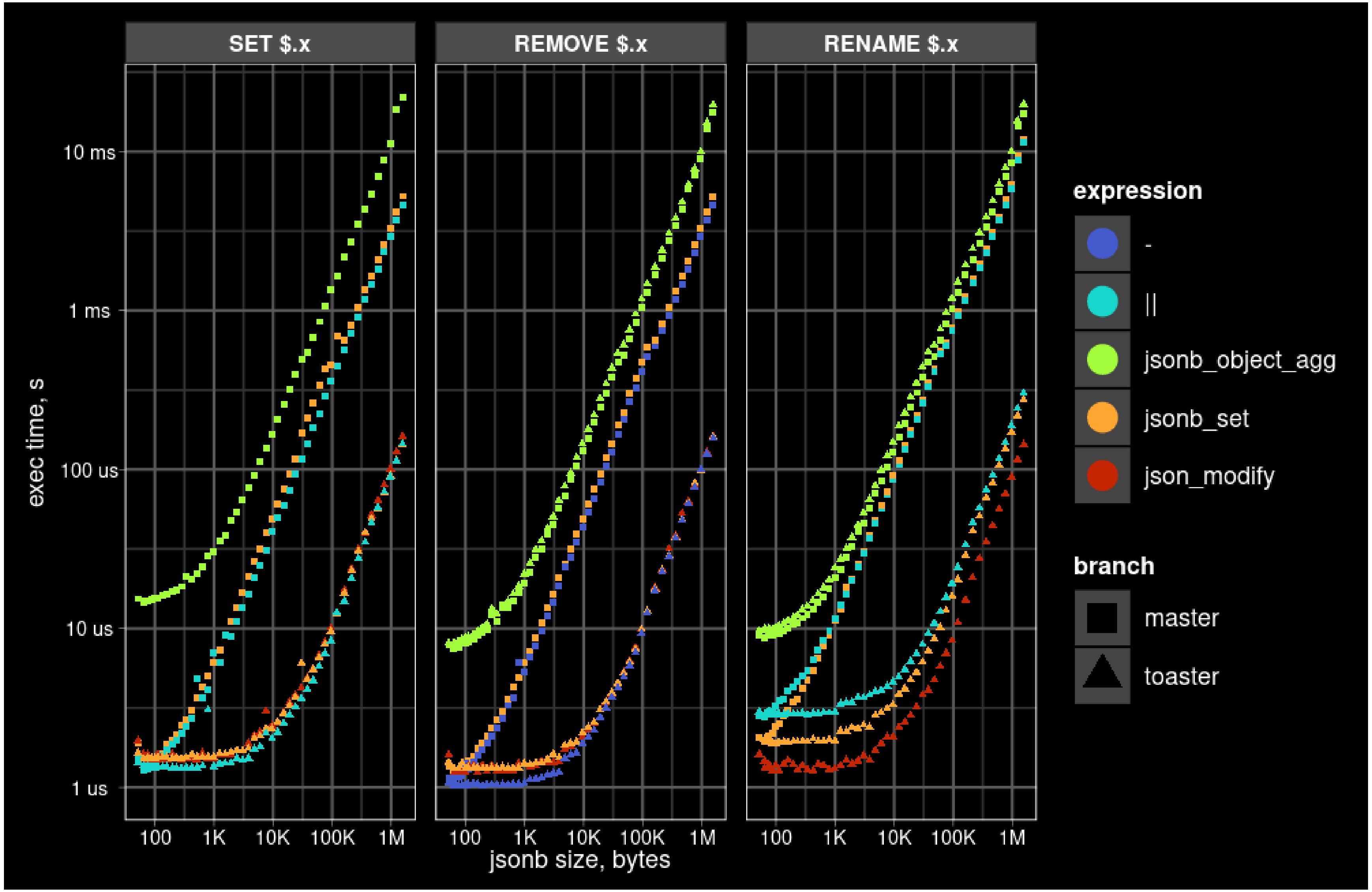
REMOVE $.x

- `JSON_MODIFY(jb, REMOVE '$.x')`
- `jsonb_set_lax(jb, '{x}', NULL, false, 'delete_key')`
- `jb - 'x'`
- `(SELECT jsonb_object_agg(k, v)`
  `FROM jsonb_each(jb) kv(k, v) WHERE k <> 'x')`

RENAME $.x

- `JSON_MODIFY(jb, RENAME '$.x' WITH 'y')`
- `jsonb_set(jb - 'x', '{y}', jb -> 'x')`
- `(jb - 'x') || jsonb_build_object('y', jb -> 'x')`
- `(SELECT jsonb_object_agg(CASE k WHEN 'x' THEN 'y' ELSE k END, v)`
  `FROM jsonb_each(jb) kv(k, v))`

JSON_MODIFY – simplest object case (results)



- Object with short “x” key and another one long key “z” (big array).
- SQL aggregation is always slower.
- We get ~30x speedup on the “gson” branch, because copying of containers (key “z”) was optimized – they are copied as binary blobs without iteration into its elements.

JSON_MODIFY – simplest **array** case (queries)

```
CREATE TABLE test_array AS
SELECT i id, (SELECT jsonb_agg(x) FROM generate_series(1, (10.0 ^ (i / 10.0))::int) x) jb
FROM generate_series(1,50) i;
```

SET $[0] = 0

- `JSON_MODIFY(jb, SET '$[0]' = 0)`
- `jsonb_set(jb, '{0}', '0')`
- ```
  (SELECT jsonb_agg(CASE idx WHEN 1 THEN '0' ELSE x END)
   FROM jsonb_array_elements(jb) WITH ORDINALITY AS elements(x, idx))
  ```

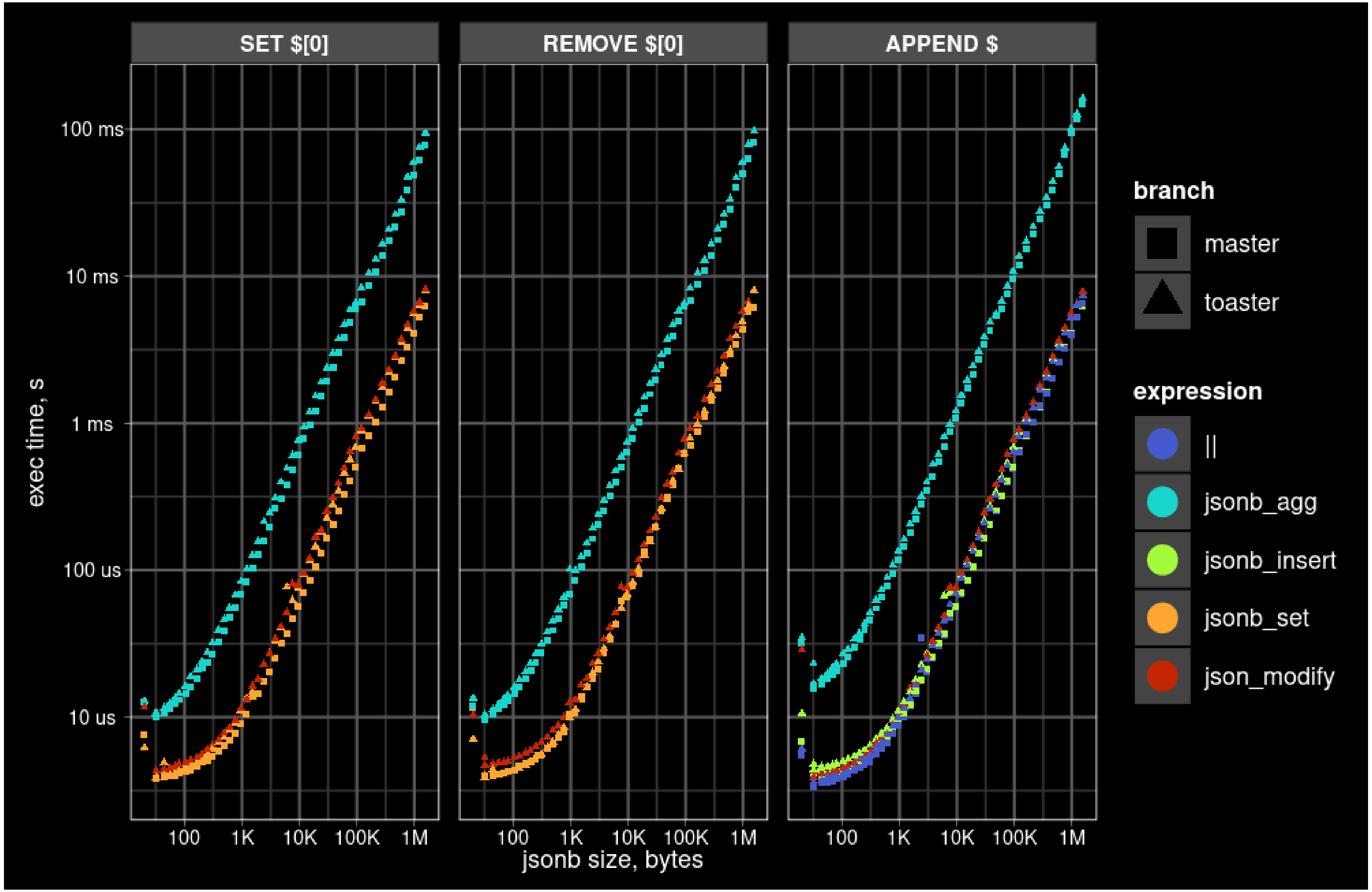
REMOVE $[0]

- `JSON_MODIFY(jb, REMOVE '$[0]')`
- `jsonb_set_lax(jb, '{0}', NULL, false, 'delete_key')`
- ```
  (SELECT coalesce(jsonb_agg(x), '[]')
   FROM jsonb_array_elements(jb) WITH ORDINALITY AS elements(x, idx)
   WHERE idx <> 1)
  ```

APPEND $

- `JSON_MODIFY(jb, APPEND '$' = 0)`
- `jsonb_insert(jb, '{-1}', '0', true)`
- `jb || '0'`
- ```
  (SELECT jsonb_agg(x)
   FROM (SELECT x FROM jsonb_array_elements(jb) x UNION SELECT '0'::jsonb x) y(x))
  ```
  ```

JSON_MODIFY – simplest **array** case (results)



- SQL aggregation is always slower.
- Other expressions have the same performance, because array always copied by element.

JSON_MODIFY – complex **array** case (queries)

`SET $[*] = 0` (multiple items matched, jsonb_set() is not applicable)

- `JSON_MODIFY(jb, SET '$[*]' = 0)`
- `(SELECT jsonb_agg(0) FROM jsonb_array_elements(jb) x)`

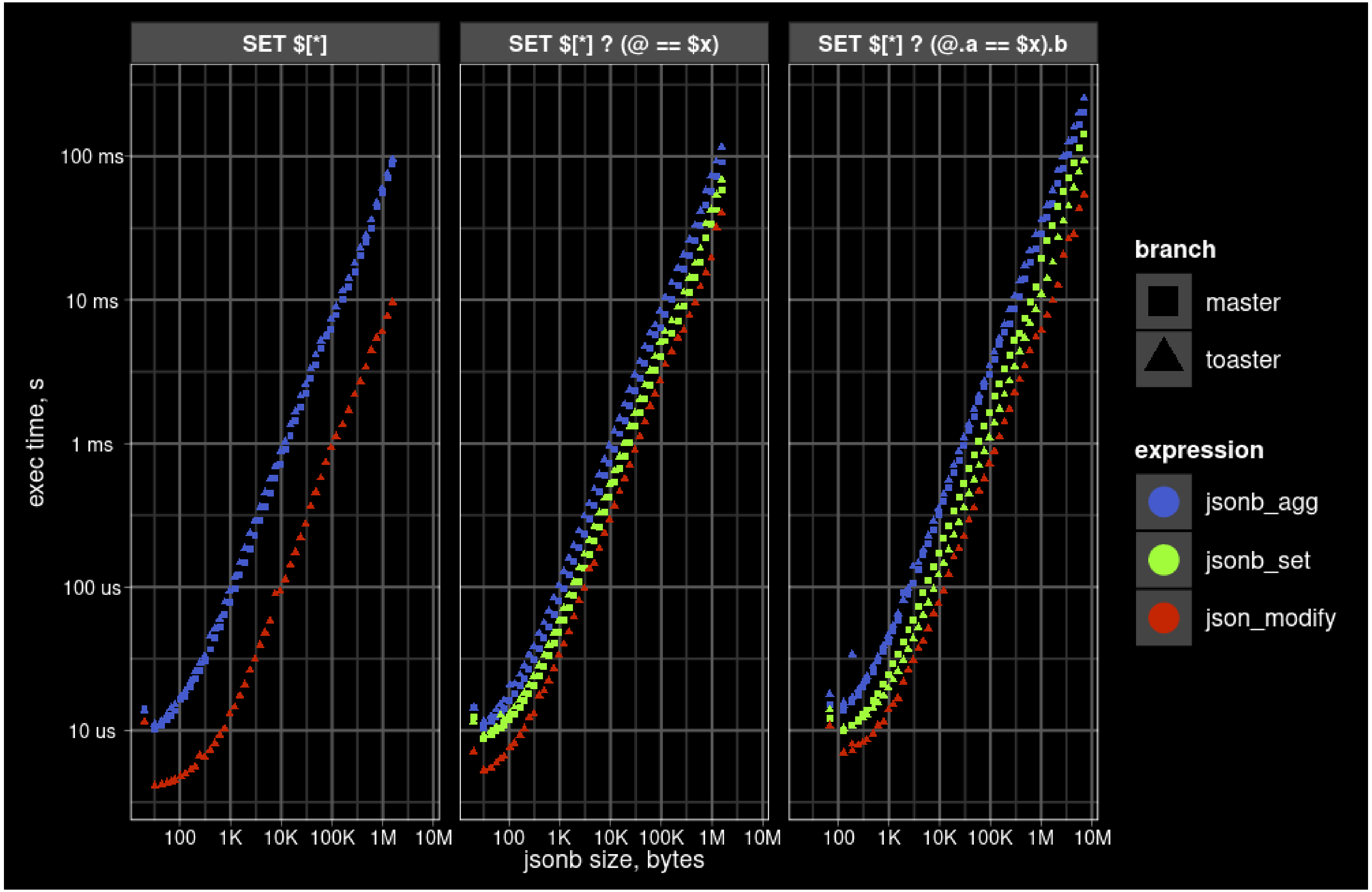
`SET $[*] ? (@ == $x) = 0`

- `JSON_MODIFY(jb, SET '$[*] ? (@ == $x)' = 0 PASSING 1 AS x)`
- `(SELECT jsonb_agg(CASE x WHEN '1' THEN '0' ELSE x END) FROM jsonb_array_elements(jb) x)`
-
```
jsonb_set(jb, ARRAY[(SELECT (idx - 1)::text
                     FROM jsonb_array_elements(jb) WITH ORDINALITY AS elements(x, idx)
                     WHERE x = '1')], '0')
```

`SET $[*] ? (@.a == $x).b = 0`

- `JSON_MODIFY(jb, SET '$[*] ? (@.a == $x).b' = 0 PASSING 1 AS x)`
-
```
(SELECT jsonb_agg(CASE x -> 'a' WHEN '1' THEN x || '{"b": 0}' ELSE x END)
 FROM jsonb_array_elements(jb) x)
```
-
```
jsonb_set(jb, ARRAY[(SELECT (idx - 1)::text
                     FROM jsonb_array_elements(jb) WITH ORDINALITY AS elements(x, idx)
                     WHERE x -> 'a' = '1'), 'b'], '0')
```

JSON_MODIFY – complex array case (results)



- SQL aggregation again is slower.
- json_modify is fastest because it does only the one pass

JSON_MODIFY – update with **toaster** (queries)

Test table:

```
CREATE TABLE test_toast AS
SELECT
   i id,
   jsonb_build_object(
    'key1', i,
    'key2', (SELECT jsonb_agg(jsonb_build_object('a', 1, 'b', 1)) FROM
             generate_series(1, pow(10, 1 + 4.0 * i / 100.0)::int)),
    'key3', i,
    'key4', (SELECT jsonb_agg(jsonb_build_object('a', 1, 'b', 1)) FROM
             generate_series(1, pow(10, 0 + 4.0 * i / 100.0)::int))
   ) jb
FROM generate_series(1, 100) i;

{ key1: id, key2: [{ a: 1, b: 1 }, …], key3: id, key4: [{ a: 1, b: 1 }, …] }
                    10-100k elements                     1-10k elements
```

Update queries:

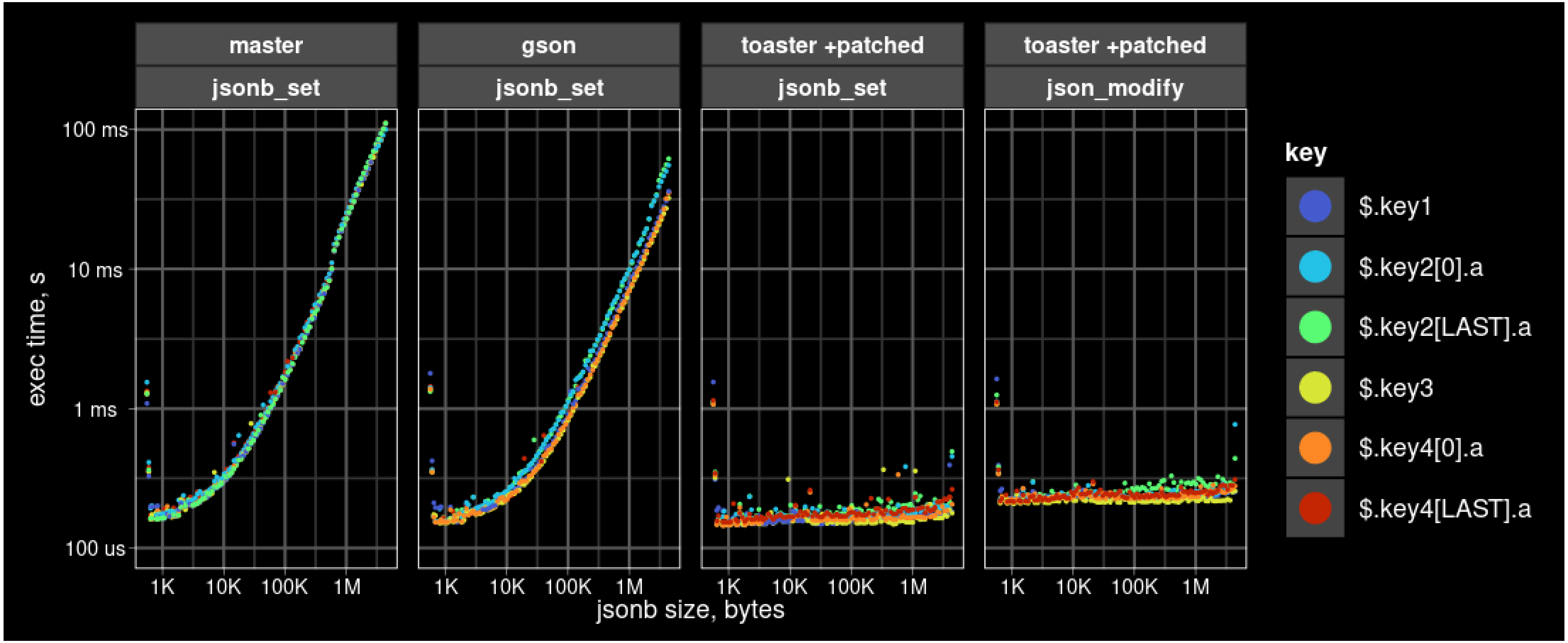
```
UPDATE test_toast SET jb = jsonb_set(jb, '{keyN,...}', ?);
UPDATE test_toast SET jb = json_modify(jb, SET '$.keyN.…' = ?);
```

Paths:

```
$.key1, $.key2[0].a, $.key2[LAST].a, $.key3.a, $.key4[0].a, $.key4[LAST].a
```

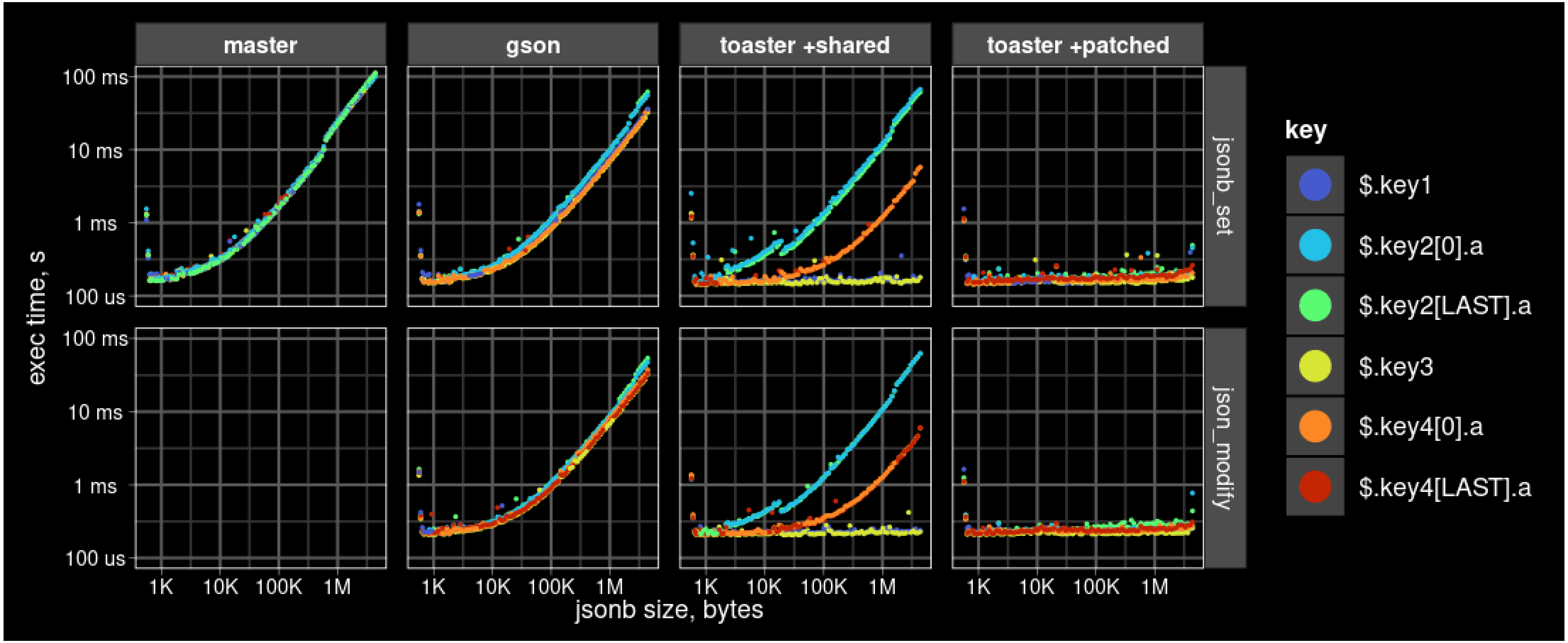
JSON_MODIFY – update with toaster (results)

Update time becomes O(1) when update+patch are applicable (new val size == old val size):



JSON_MODIFY – update with toaster (results)

For other types of updates, updated array still needs to be copied and reTOASTed, unmodified TOASTs are shared (toaster +shared):



JSON_MODIFY – TODO

- Implement KEEP operation
  - Needs completely new executor function
- Optimize multiple operations by executing them together in the single pass through jsonb
  - Check operation independence before
  - Group jsonpath accessors at each level
  - New executor also needed
- Implement passing of old JSON items to the new value expression for non-constant updates like increments
  - `JSON_MODIFY(jb, '$[*].counter' = counter + 1 PASSING OLD JSON AS counter)`

Summary

- PostgreSQL is already good NoSQL database
- SQL/JSON provides better flexibility and interoperability
  - Need JSON_MODIFY — a missing analog for jsonb_set
- JSONB is capable for great performance
  - TOAST API + JSONB TOASTER
- Projective indexing for JSONB — index what you want
- COPY with FORMAT JSONPATH - copy what you want
- Unification of JSON and JSONB - choose what you want

PostgresPro

References

- This talk: http://www.sai.msu.su/~megera/postgres/talks/sqljson-pgconfru-2022.pdf
- PG15 SQL/JSON docs: https://www.postgresql.org/docs/15/functions-json.html
- 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
- Pluggable TOAST talk: http://www.sai.msu.su/~megera/postgres/talks/toast-pgcon-2022.pdf
- Pluggable TOAST at Commitfest https://commitfest.postgresql.org/38/3490/
- TOAST API @GitHub https://github.com/postgrespro/postgres/tree/toasterapi_clean
- Jsonb_toaster @Github (check License.txt)
  https://github.com/postgrespro/postgres/tree/jsonb_toaster
- JSON_MODIFY @Github: https://github.com/postgrespro/postgres/tree/json_modify
- JSON[B] Roadmap V3, Postgres Build 2020, Dec 8, 2020

PostgresPro


ALL YOU NEED POSTGRES IS