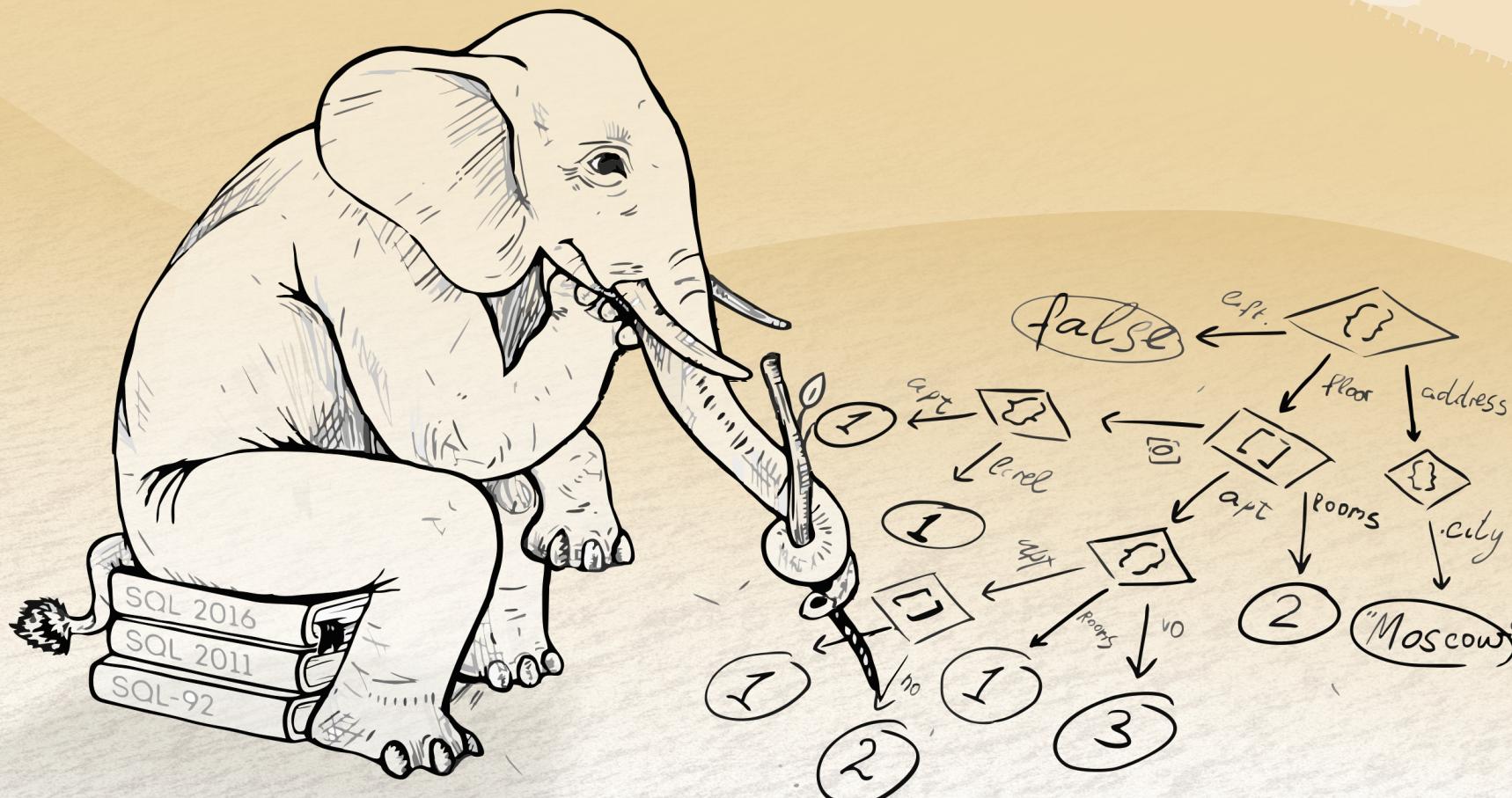


JSONPATH - a query language for json



Oleg Bartunov
Postgres Professional

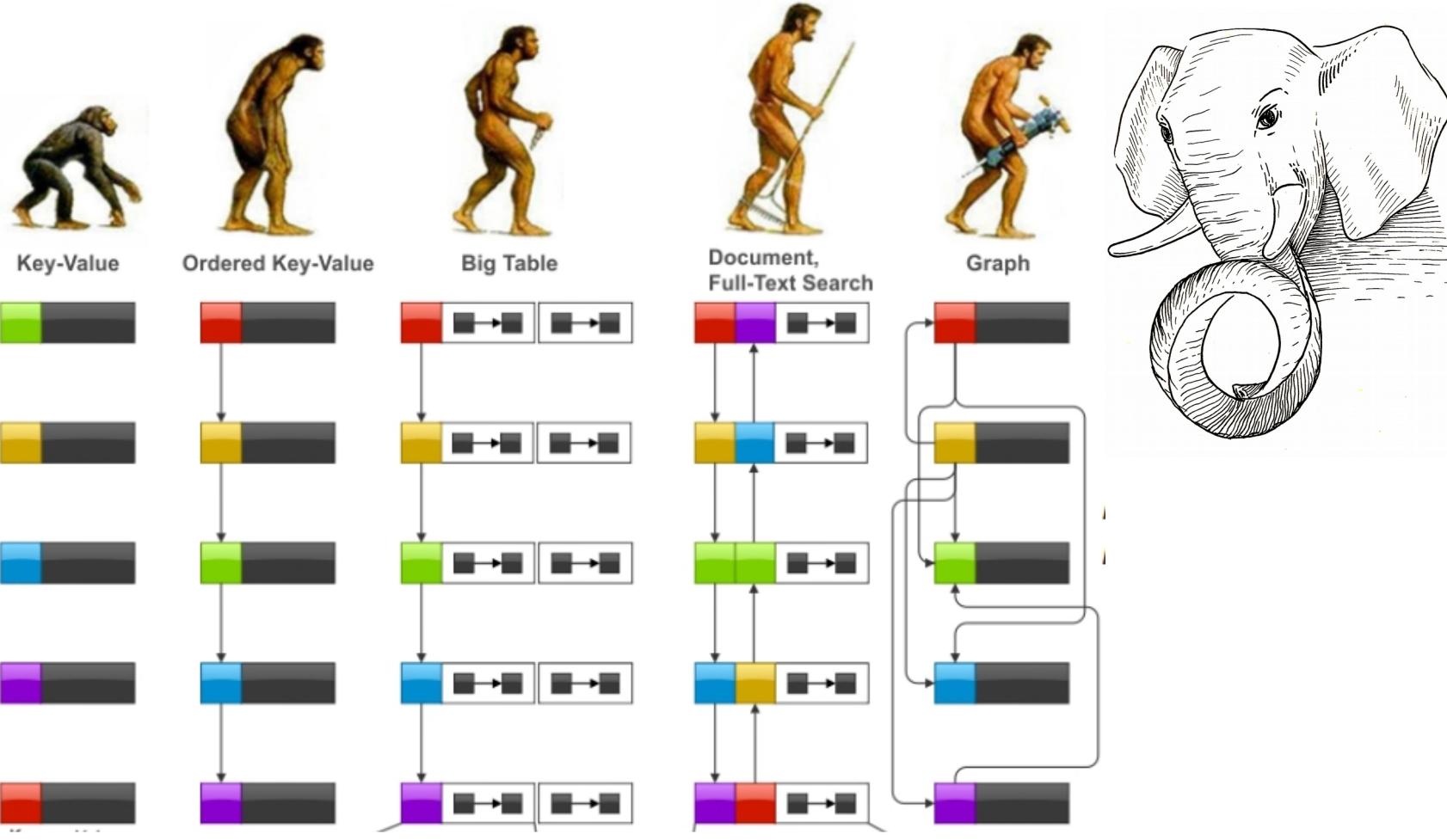
Since Postgres95



Research scientist,
Moscow University
CEO PostgreSQL Professional

PostgresPro

NOSQL POSTGRES IN SHORT



SQL/JSON — 2020

- Complete SQL/JSON
- Better indexing, syntax

JSONPATH - 2019

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

JSONB - 2014

- Binary storage
- Nesting objects & arrays
- Indexing

JSON - 2012

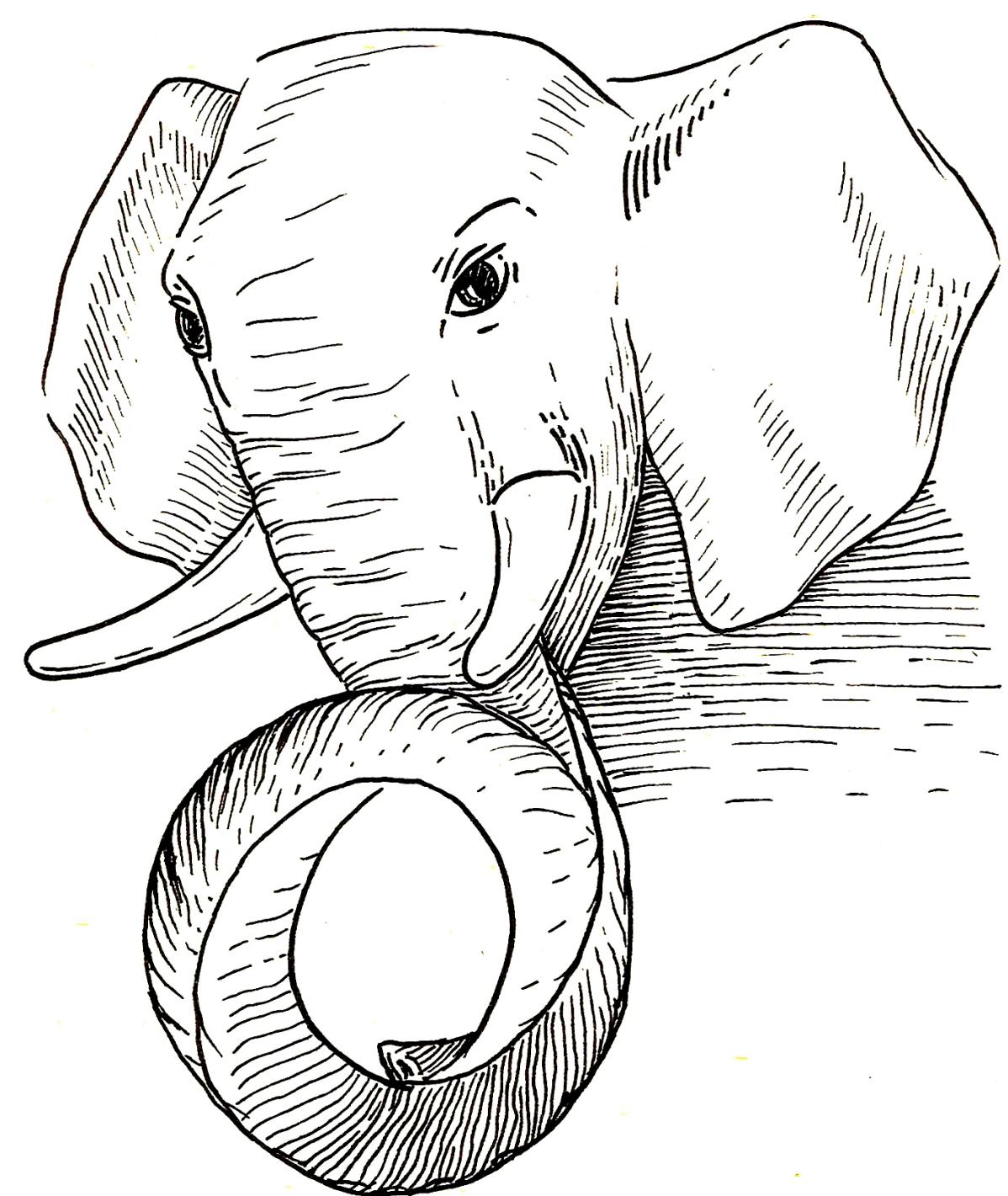
- Textual storage
- JSON verification

HSTORE - 2003

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



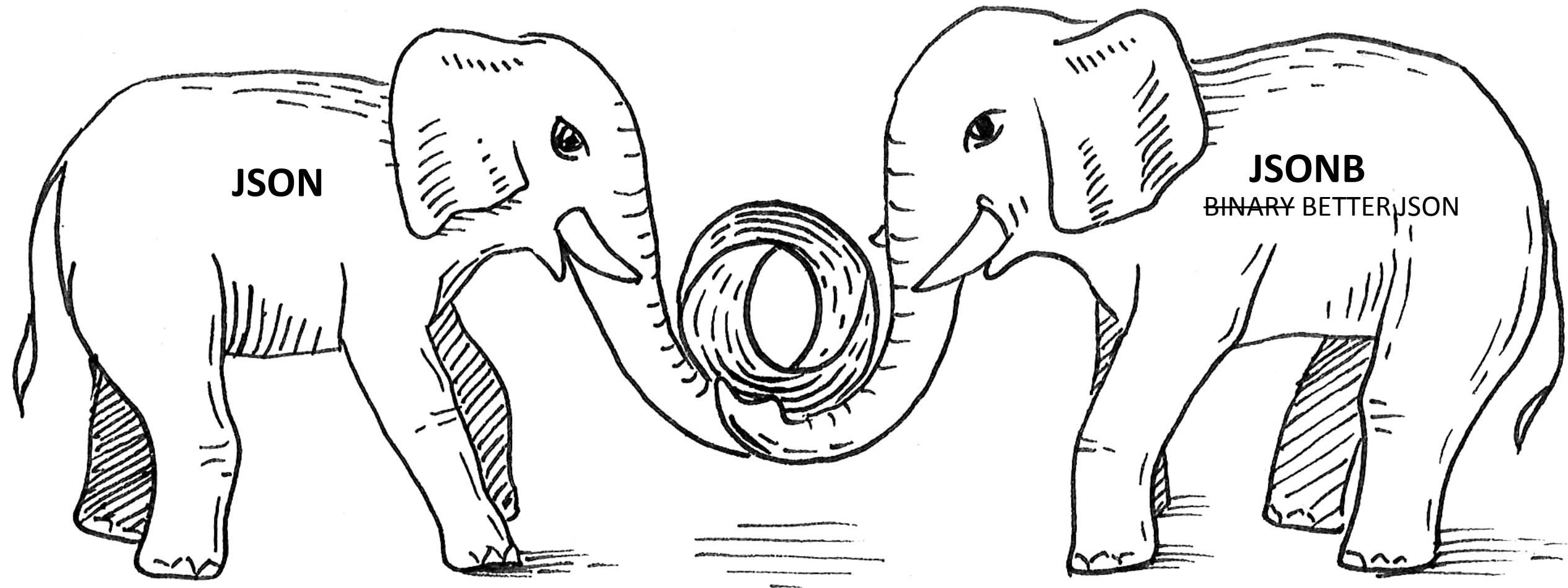
PostgresPro



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»
- jsonb: binary storage, no need to parse, has index support
- jsonb: no whitespaces, no duplicated keys (last key win)
- jsonb: keys are sorted by (length, key)
- jsonb: a rich set of functions (\df jsonb*)
- jsonb: great performance, thanks to indexes
- JsQuery - json query language with GIN indexing support

A painting featuring three women in a landscape. In the foreground, a woman with short brown hair and a blue headband looks up. Behind her, another woman with dark hair and a red headband looks down. In the background, a woman with long blonde hair and a green headband looks directly at the viewer. A white speech bubble originates from the woman in the background.

**JSONB is GREAT,
BUT ...**

JSON[B] is a black box for SQL

```
WITH RECURSIVE t(id, value) AS ( SELECT * FROM js_test
UNION ALL
(
  SELECT
    t.id,
    COALESCE(kv.value, e.value) AS value
  FROM
    t
    LEFT JOIN LATERAL
    jsonb_each(
CASE WHEN jsonb_typeof(t.value) =
'object' THEN t.value
      ELSE NULL END) kv ON true
    LEFT JOIN LATERAL jsonb_array_elements(
      CASE WHEN
        jsonb_typeof(t.value) = 'array' THEN t.value
        ELSE NULL END) e ON true
  WHERE
    kv.value IS NOT NULL OR e.value IS
NOT NULL
)
SELECT
  js_test.*
FROM
  (SELECT id FROM t WHERE value @> '{"color": "red"}') GROUP BY id) x
  JOIN js_test ON js_test.id = x.id;
```

```
SELECT * FROM js_test;
```

id	value
1	[1, "a", true, {"b": "c", "f": false}]
2	{"a": "blue", "t": [{"color": "red", "width": 100}]} [{"color": "red", "width": 100}]
4	{"color": "red", "width": 100}
5	{"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"} {"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"} {"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"} {"a": "blue", "t": [{"color": "green", "width": 100}]} {"color": "green", "value": "red", "width": 100}
(9 rows)	

Jsquery (2014)

<https://github.com/postgrespro/jsquery/>

```
SELECT * FROM js_test
WHERE
  value @@ '* .color = "red"';
```



**SQL Standard
now loves JSON !**

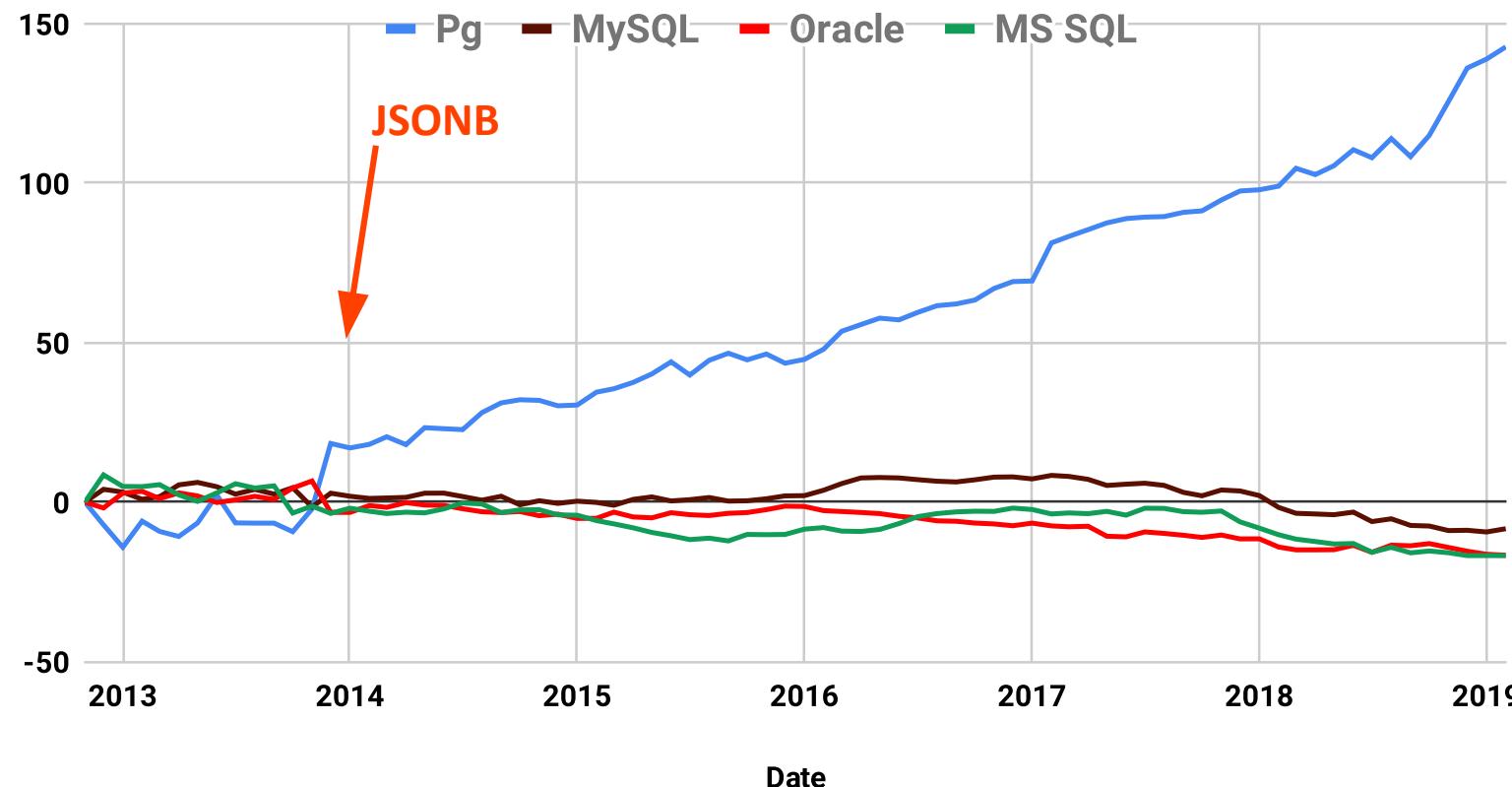
**JSONB and JsQuery are
GREAT, BUT ...**

OH, REALLY ?

Postgres revolution: embracing relational databases

- NoSQL users attracted by the NoSQL Postgres features

Relative Growth db-engines



18 декабря 2014

SQL/Foundation recognized JSON after 8 years

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

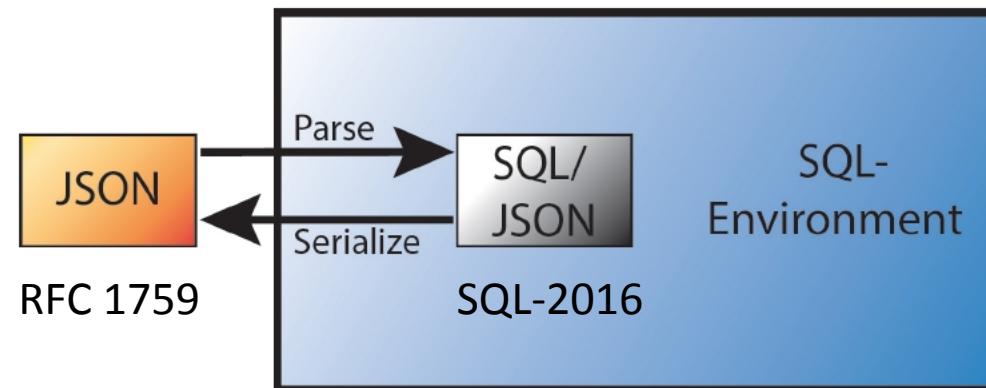
- A *sequence of SQL/JSON items*, each item can be (recursively) any of:
 - SQL/JSON scalar — non-null value of SQL types: Unicode character string, numeric, Boolean or datetime
 - SQL/JSON *null*, value that is distinct from any value of any SQL type (not the same as NULL)
 - JSON arrays, ordered list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

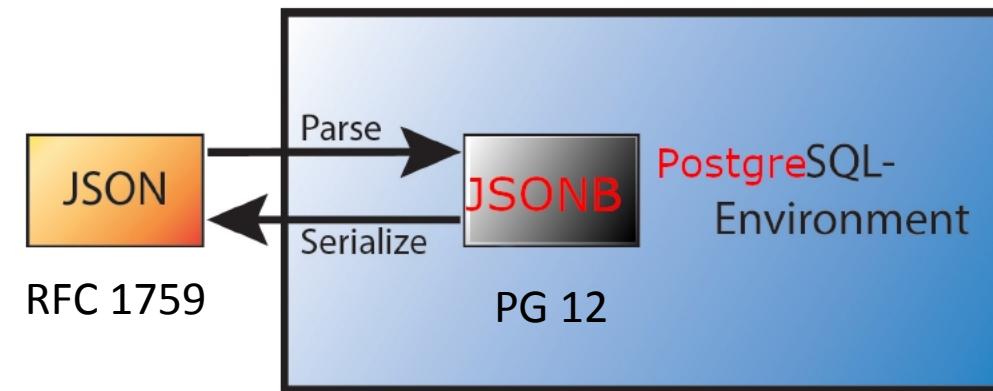
- SQL/JSON functions (9)

- 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 opclasses with jsonpath support**



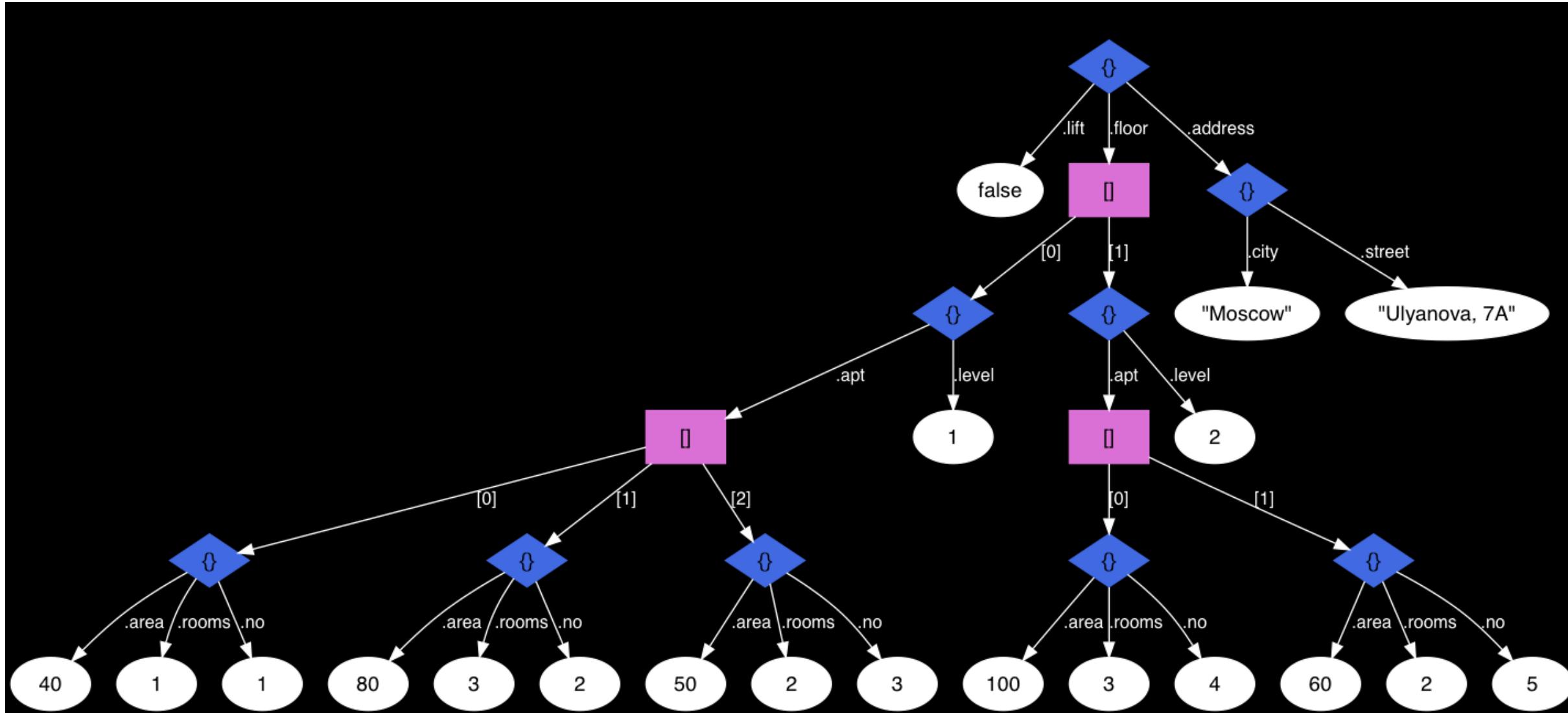
JSON Path query language

- **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)'
```

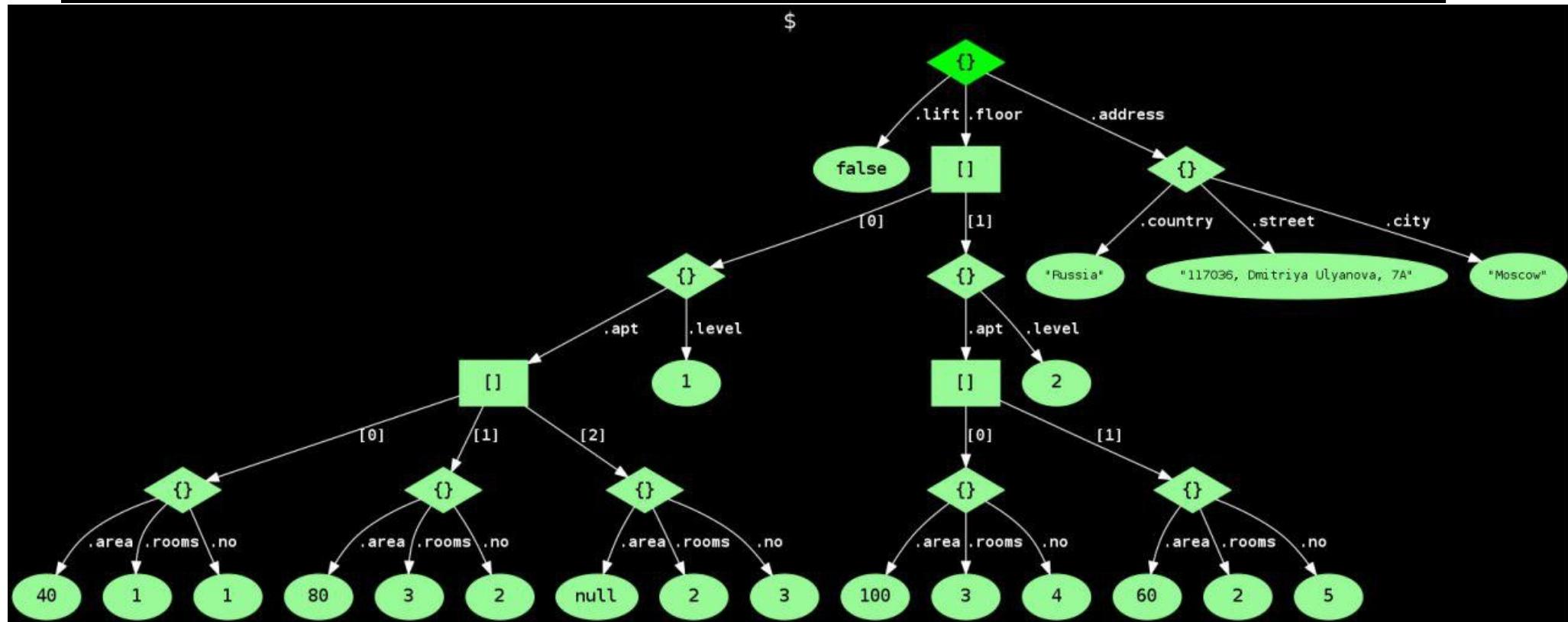
- Dot notation used for member access – '\$.a.b.c'
- \$ - the current context element
- [*], [0 to LAST] - array access (starts from zero!)
- Filter(s) - '\$.a.b.c ? (@.x > 10)'
- @ - current context in filter expression
- Item methods - '\$.a.b.c.x.type()' type(), size(), double(), ceiling(), floor(),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

Two floors house



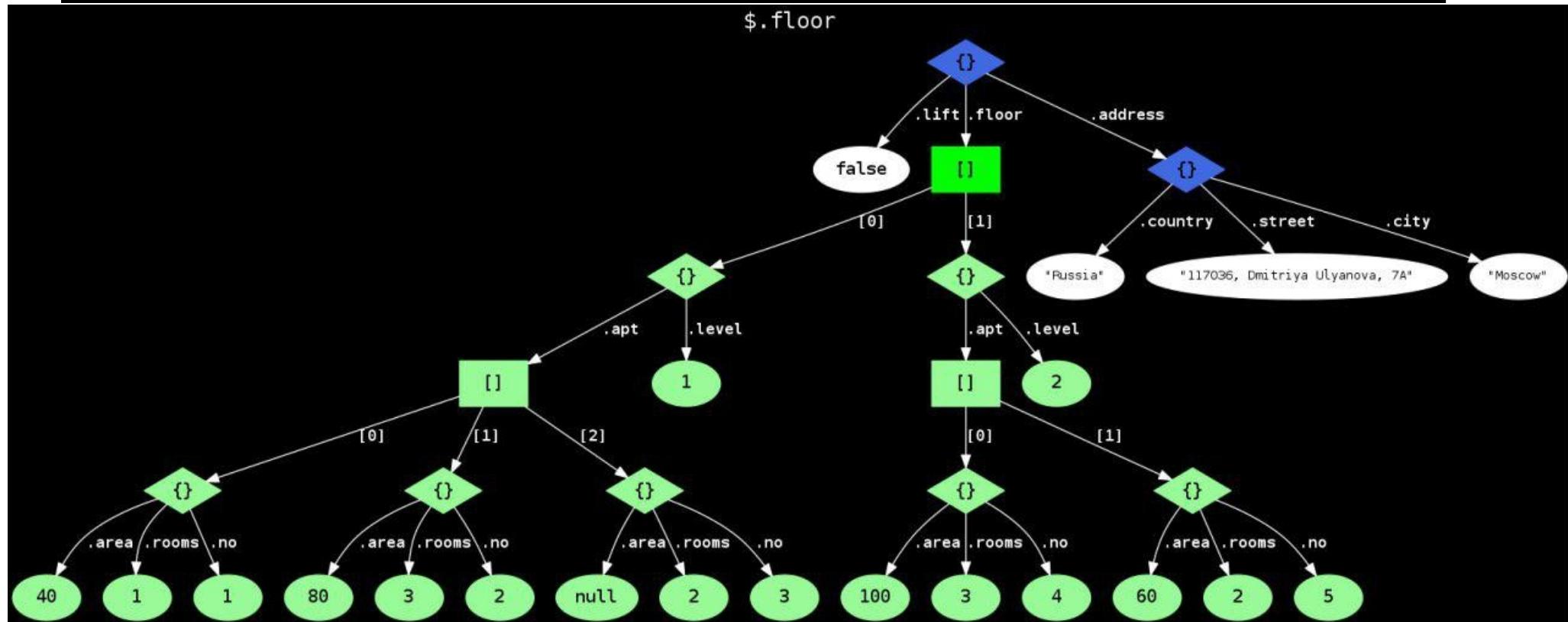
How path expression works (1)

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



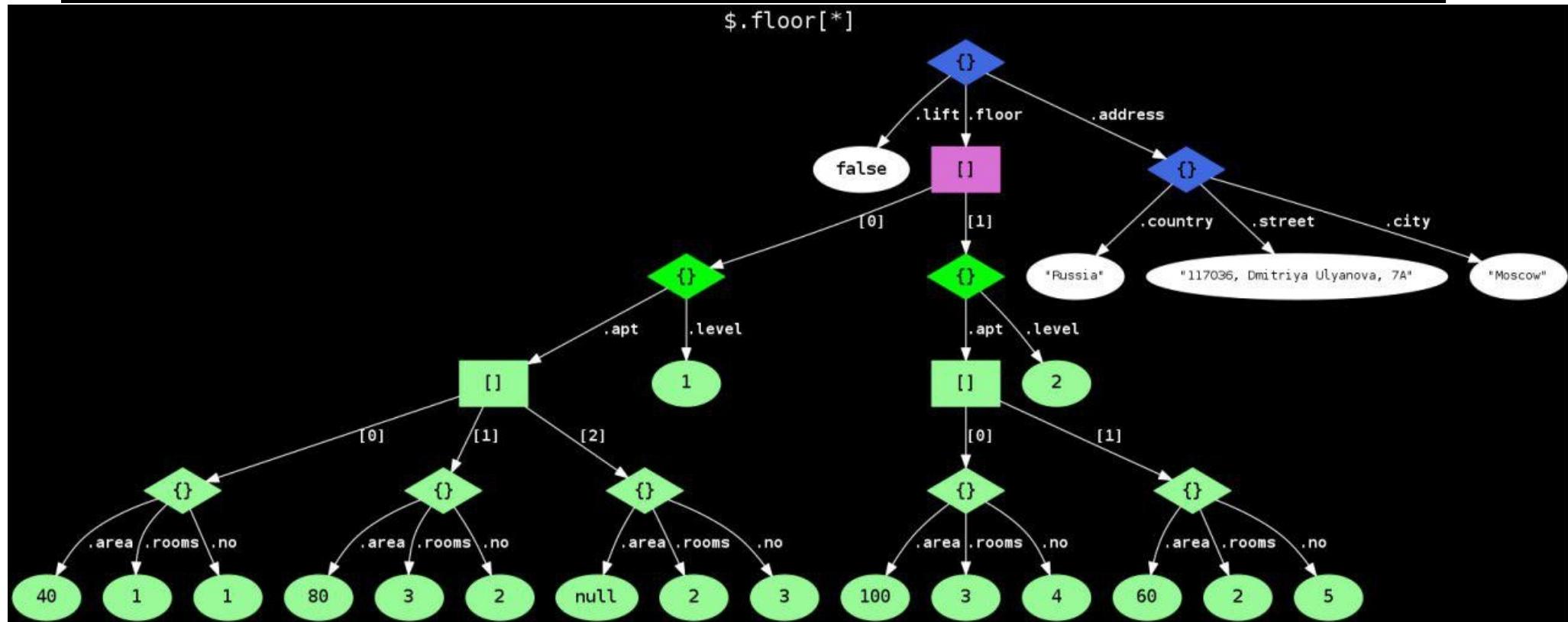
How path expression works (2)

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



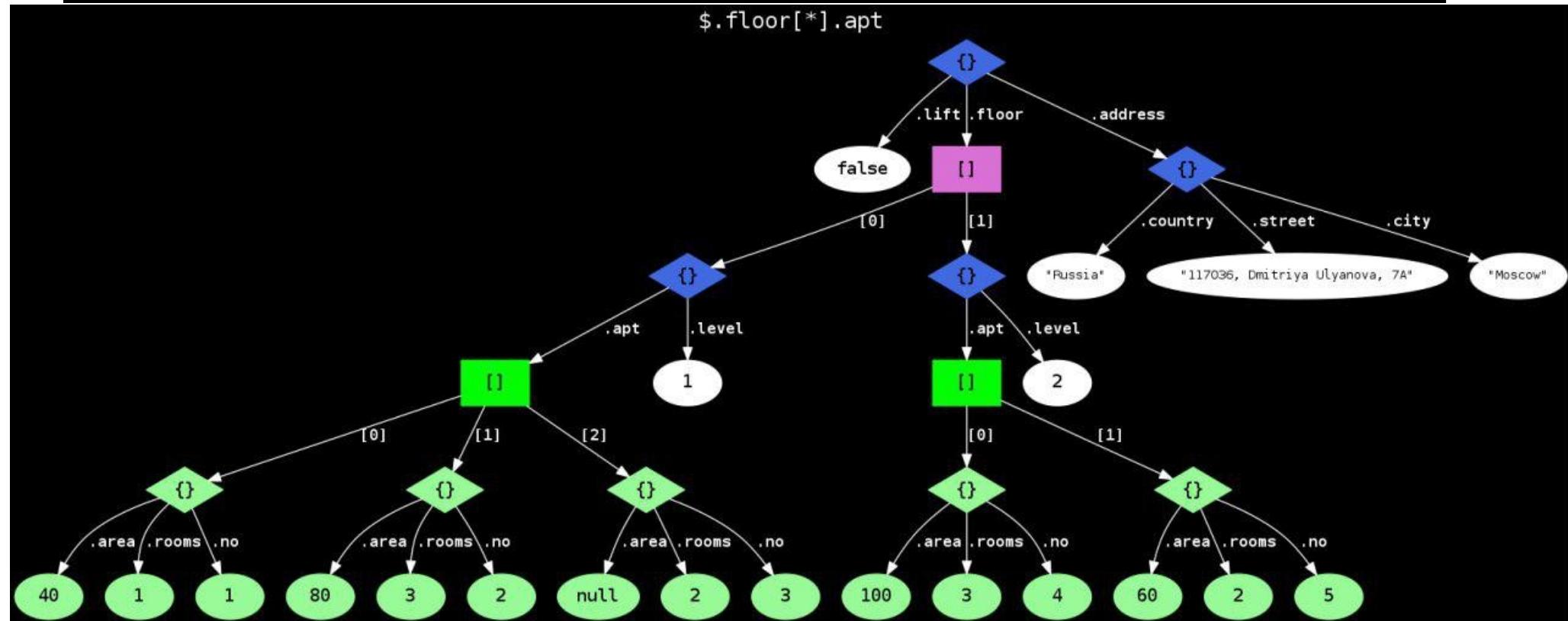
How path expression works (3)

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



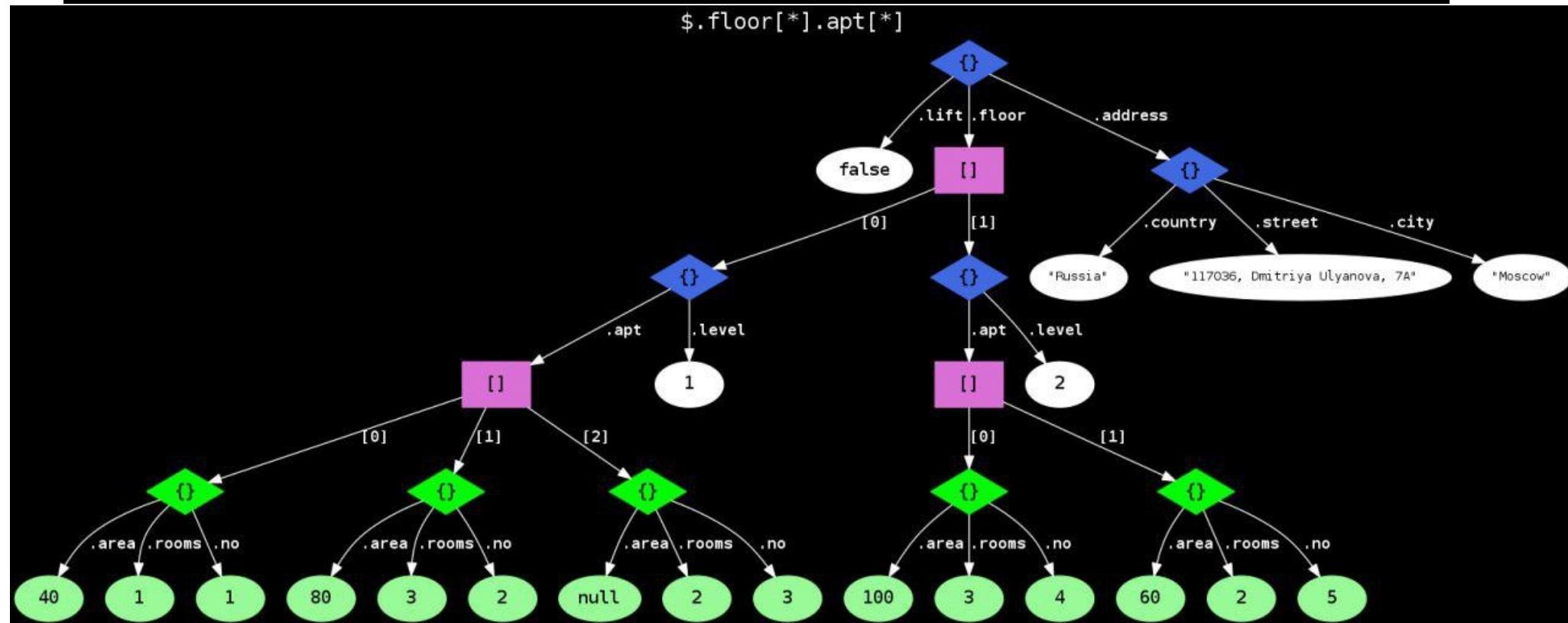
How path expression works (4)

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



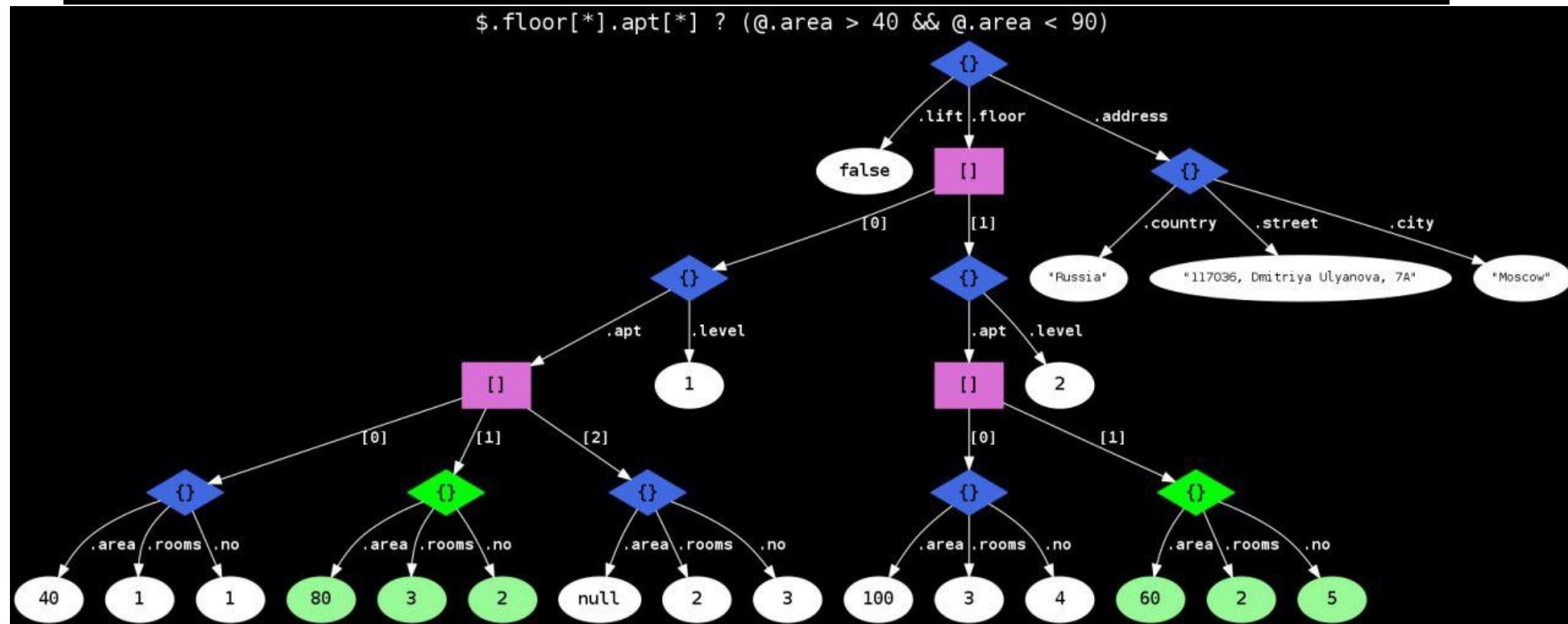
How path expression works (5)

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



How path expression works (6)

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



How path expression works (summary)

```
'$.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 apartment filtered by (@.area > 40 && @.area < 90) expression

The result is a sequence of two SQL/JSON items

JSON Path: [lax] vs strict

Lax and *strict* modes used to facilitate matching of the (sloppy) document structure and path expression

- Lax: missing keys ignored

```
SELECT jsonb '{"a":1}' @? 'lax $.b ? (@ > 1)';  
?column?
```

f

- Strict: missing keys resulted null

```
SELECT jsonb '{"a":1}' @? 'strict $.b ? (@ > 1)';  
?column?
```

(null)

JSON Path: [lax] vs strict

- Lax: arrays are unwrapped

```
SELECT jsonb '[1,2,[3,4,5]]' @? 'lax $[*] ? (@ == 5)';  
?column?  
-----  
t
```

- Strict: requires an exact nesting

```
SELECT jsonb '[1,2,[3,4,5]]' @? 'strict $[*] ? (@[*] == 5)';  
?column?  
-----  
t
```

JSON Path examples 1/3

- JSON Path expression 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).

'\$' -- the whole JSON document (context item)

'\$foo' -- variable "foo"

"bar" -- string literal

'12.345' -- numeric literal

'true' -- boolean literal

'null' -- null

'.floor' -- field accessor on \$

'.floor[*]' -- the same, followed by wildcard array accessor

JSON Path examples 2/3

- JSON Path expression 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).

-- complex path with filters and variables

```
'$.floor[*] ? (@.level < $max_level).apt[*] ? (@.area > $min_area).no'
```

-- arithmetic expressions:

'-\$a[*]' -- unary

'\$a + 3' -- binary

'2 * \$a - (3 / \$b + \$x.y)' -- complex expression with variables

JSON Path examples 3/3

- JSON Path expression 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).

-- parenthesized expression used as starting element of a path,
-- followed by two item methods ".abs()" and ".ceil()"
'(\$ + 1).abs().ceil()'

Syntactical errors in `jsonpath` are reported:

SELECT '\$a. >1'::jsonpath;

ERROR: bad jsonpath representation at character 8

DETAIL: syntax error, unexpected GREATER_P at or near ">"

JSON Path filter expression

- A filter expression is similar to a `WHERE` clause in SQL, it is used to remove SQL/JSON items from an SQL/JSON sequence if they do not satisfy a predicate.
- Syntax: ? (JSON path predicate)
- Result: True, False, Unknown
- Filter works as follows:
 - 1) In lax mode, any SQL/JSON arrays in the operand are unwrapped
 - 2) The predicate is evaluated for each SQL/JSON item in the SQL/JSON sequence
 - 3) The result is those SQL/JSON items for which the predicate resulted in True.
- The special variable @ in filter is a reference to the current SQL/JSON item in the SQL/JSON sequence. The value of @ is the current SQL/JSON item of the first operand of the innermost filter with @.

JSON Path filter expression

- Predicates:
 - *exists*, test if a path expression has a non-empty result
 - Comparison predicates ==, !=, <>, <, <=, >, and >=
 - *like_regex* for string pattern matching.
Optional *flag* can be combination of i, s (default), m, x.
 - *starts with* to test for an initial substring (prefix).
 - *is unknown* to test for *Unknown* results. Its operand should be in parentheses.

JSON Path filters

- Arithmetic errors in filters suppressed:
 - behaviour required by standard

```
SELECT jsonb_path_query('[1,0,2]', '$[*] ? (1 / @ >= 1)');  
jsonb_path_query  
-----  
1  
(1 row)
```

JSON Path methods

- Predefined methods attached to JSON Path expression

```
SELECT jsonb_path_query(jsonb '{"a":5, "b":2}', '$ ? (@.a > 1).keyvalue()')
FROM house;
          jsonb_path_query
-----
{"id": 0, "key": "a", "value": 5}
 {"id": 0, "key": "b", "value": 2}
(2 rows)
```

- Methods can be combined

```
SELECT jsonb_path_query(jsonb '{"a":5, "b":2}', '$ ? (@.a > 1).keyvalue().key')
FROM house;
          jsonb_path_query
-----
 "a"
 "b"
(2 rows)
```

JSON Path implementation in Postgres

Standard permits only string literals in JSON Path specification.

- JSON Path in Postgres implemented as **jsonpath** data type - the binary representation of parsed SQL/JSON path expression.
- To accelerate JSON Path queries using **existing** indexes for jsonb we need boolean operators for json[b] and jsonpath.
- Implementation as a type is much easier than integration of JSON path processing with executor (complication of grammar and executor).
- In simple cases, expressions with operators can be more concise than with SQL/JSON functions.
- It is Postgres way to use operators with custom query types (tsquery for FTS, lquery for ltree, jsquery for jsonb,...)

jsonpath functions

- **jsonb_path_exists()** => boolean
Test whether a JSON path expression returns any SQL/JSON items (operator @?).
- **jsonb_path_match()** => boolean
Evaluate JSON path predicate (operator @@).
- **jsonb_path_query()** => setof jsonb
Extract a sequence of SQL/JSON items from a JSON value.
- **jsonb_path_query_array()** => jsonb
Extract a sequence of SQL/JSON items wrapped into JSON array.
- **jsonb_path_query_first()** => jsonb
Extract the first SQL/JSON item from a JSON value.

Jsonpath functions

- All jsonb_path_xxx() functions have the same signature:

```
jsonb_path_xxx(  
    js jsonb,  
    jsp jsonpath,  
    vars jsonb DEFAULT '{}',  
    silent boolean DEFAULT false  
)
```

- "vars" is a jsonb object used for passing jsonpath variables:

```
SELECT jsonb_path_query_array('[1,2,3,4,5]', '$[*] ? (@ > $x)',  
                               vars => '{"x": 2}');  
  
jsonb_path_query_array  
-----  
[3, 4, 5]
```

Jsonpath functions

- "silent" flag enables suppression of errors:

```
SELECT jsonb_path_query('[]', 'strict $.a');
ERROR:  SQL/JSON member not found
DETAIL:  jsonpath member accessor can only be applied to an object
```

```
SELECT jsonb_path_query('[]', 'strict $.a', silent => true);
jsonb_path_query
-----
(0 rows)
```

Jsonpath functions: Examples

- jsonb_path_exists('{"a": 1}', '\$.a') => true
jsonb_path_exists('{"a": 1}', '\$.b') => false
- jsonb_path_match('{"a": 1}', '\$.a == 1') => true
jsonb_path_match('{"a": 1}', '\$.a >= 2') => false
- jsonb_path_query('{"a": [1,2,3,4,5]}', '\$.a[*] ? (@ > 2)') => 3, 4, 5 (3 rows)
jsonb_path_query('{"a": [1,2,3,4,5]}', '\$.a[*] ? (@ > 5)') => (0 rows)

Jsonpath functions: Examples

- `jsonb_path_query_array('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 2)') => [3, 4, 5]`
`jsonb_path_query_array('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 5)') => []`
- `jsonb_path_query_first('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 2)') => 3`
`jsonb_path_query_first('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ > 5)') => NULL`

Jsonpath: boolean operators for jsonb

- `jsonb @? jsonpath` (exists)

Test whether a JSON path expression returns any SQL/JSON items.

```
jsonb '[1,2,3]' @? '$[*] ? (@ == 3)' => true
```

- `jsonb @@ jsonpath` (match)

Get the result of a JSON path predicate.

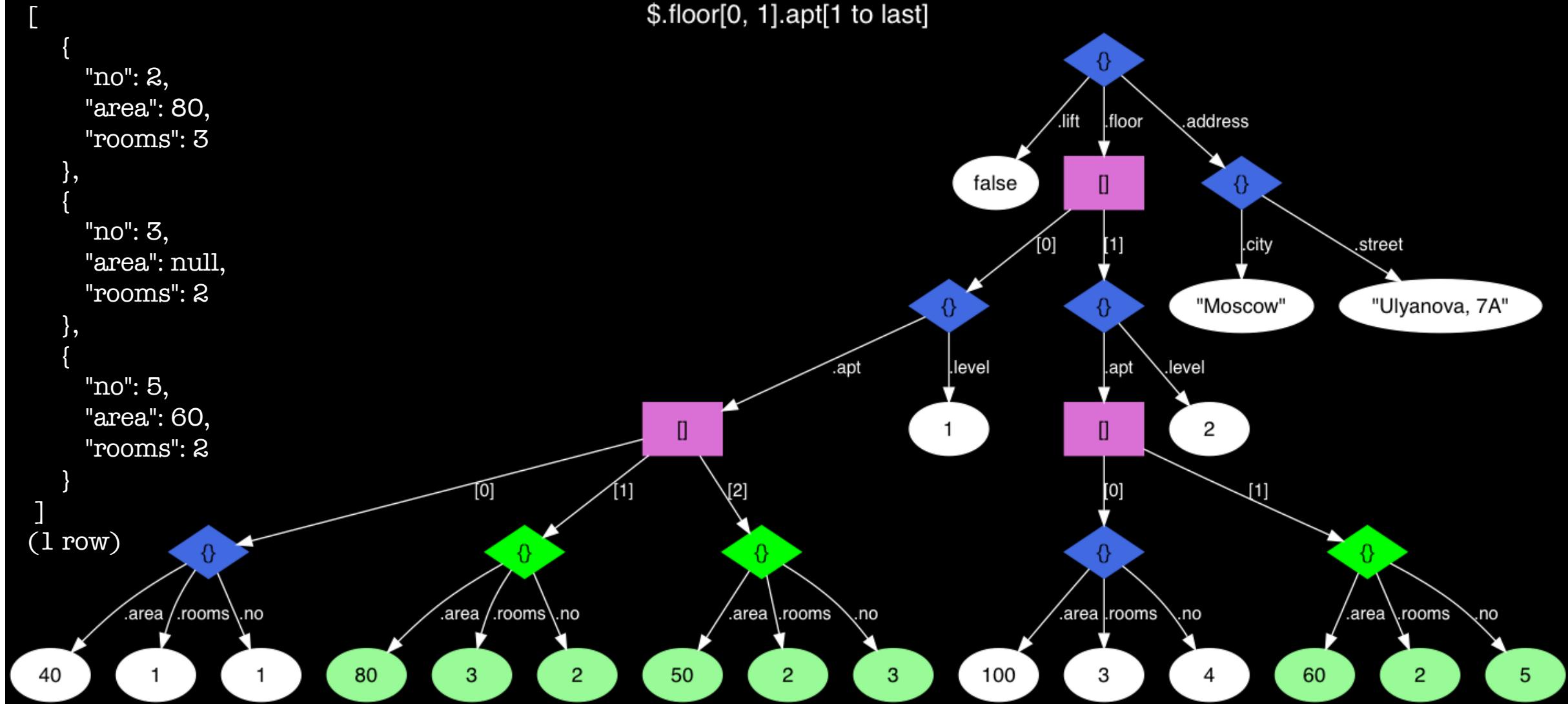
```
jsonb '[1,2,3]' @@ '$[*] == 3' => true
```

- These operators are interchangeable:

```
js @? '$.a' <=> js @@ 'exists($.a)'
```

```
js @@ '$.a == 1' <=> js @? '$ ? ($.a == 1)'
```

`$.floor[0,1].apt[1 to last]`



`$.floor[0, 1].apt[1 to last]`

- PG12 (jsonpath) query

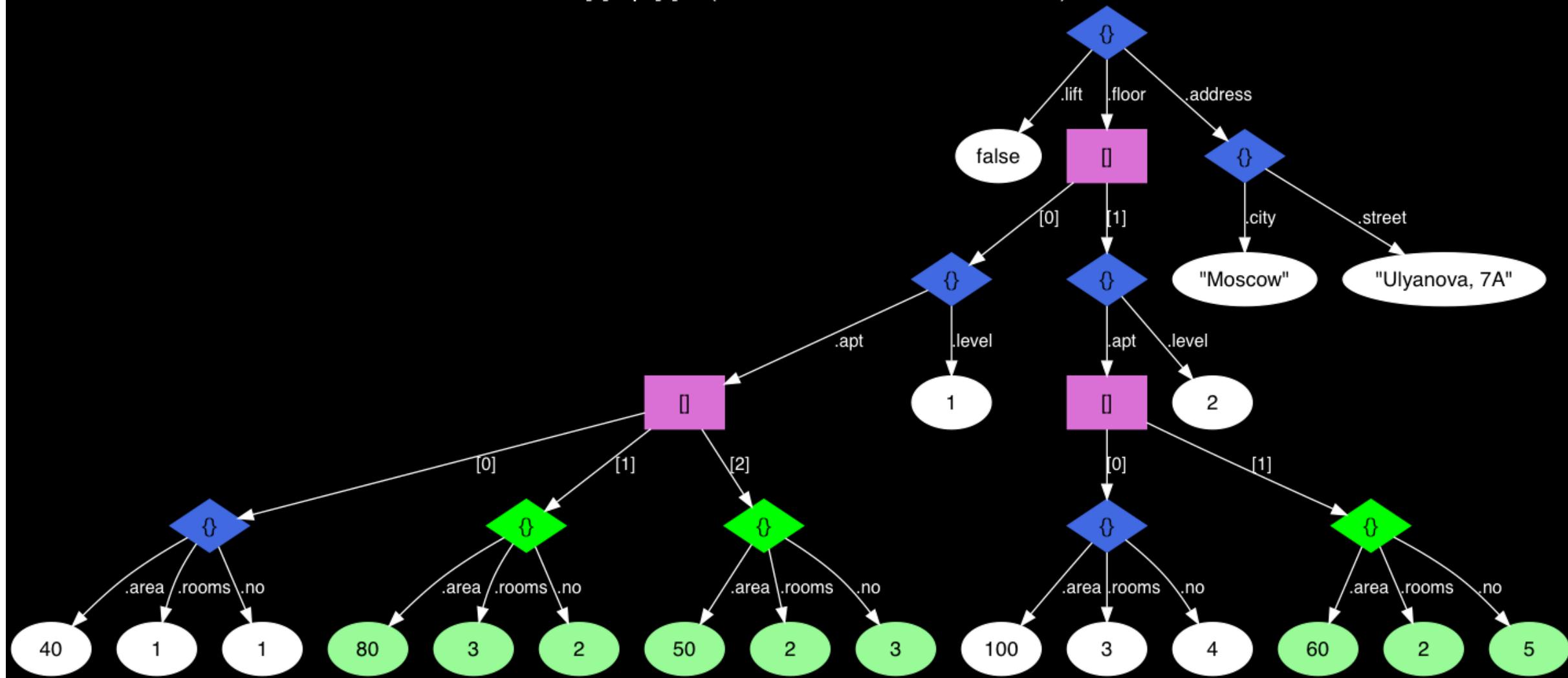
```
SELECT jsonb_path_query_array(js, '$.floor[0, 1].apt[1 to last]')
FROM house;
```

- PG11 query

```
SELECT jsonb_agg(apt)
FROM (SELECT apt->generate_series(1, jsonb_array_length(apt) - 1)
      FROM (SELECT js->'floor'->unnest(array[0, 1])->'apt'
            FROM house) appts(apt)) appts(apt);
```

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

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



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

- PG12 (jsonpath) query

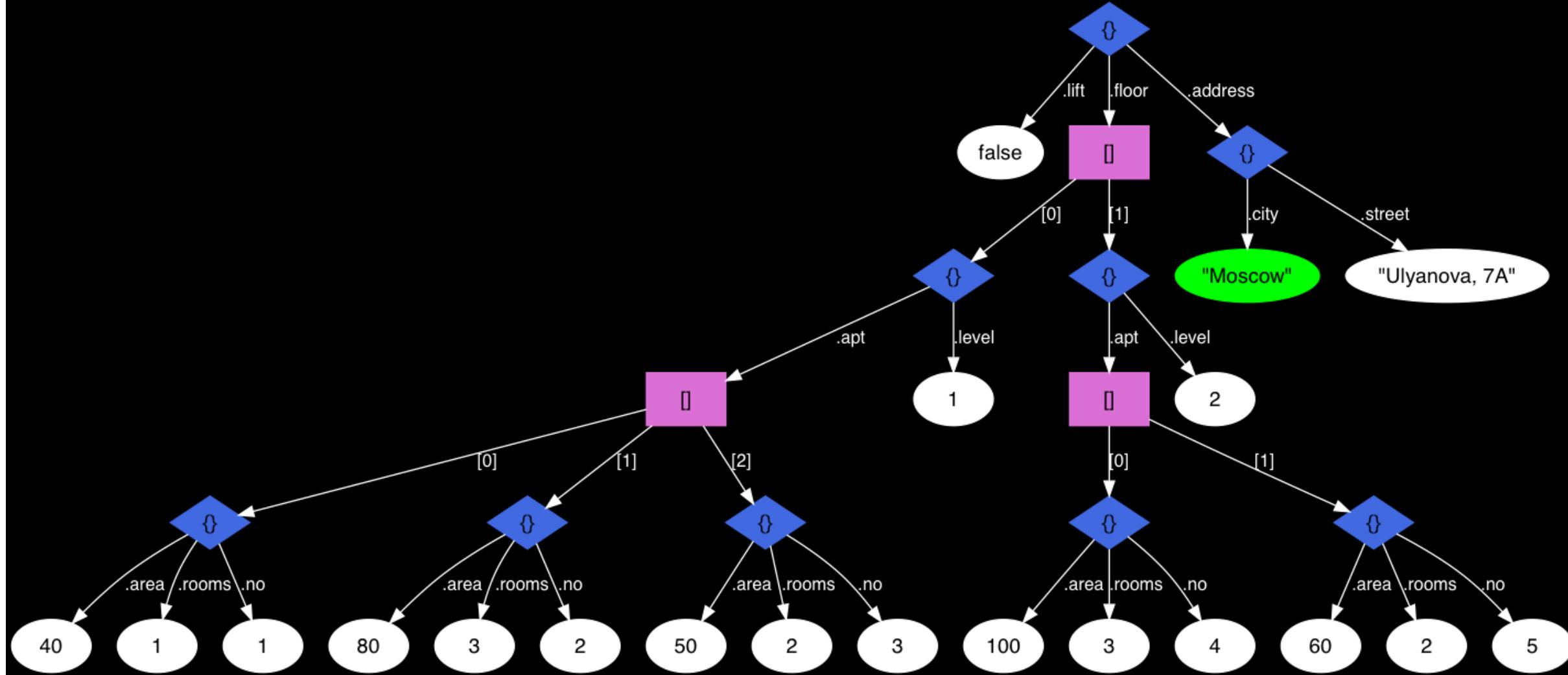
```
SELECT jsonb_path_query(js, '$.floor[*].apt[*] ?  
(@.area > 40 && @.area < 90)')  
FROM house;
```

- 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;
```

Extension: \$.** ? (@ == "Moscow")

\$.** ? (@ == "Moscow")



Extension: `$.*?(@ == "Moscow")`

- PG12 (jsonpath wildcard) query

```
SELECT jsonb_path_exists(js, '$.*?(@ == "Moscow")') FROM house;  
SELECT jsonb_path_exists(js, '$.*?{@ == "Moscow"}') FROM house;
```

- JSQUERY query

<https://github.com/postgrespro/jsquery>

```
SELECT  
js @@ '* = "Moscow"'::jsquery  
FROM house.
```

Extension: \$.** ? (@ == "Moscow")

- PG11 query

```
WITH RECURSIVE t(value) AS
  (SELECT * FROM house
   UNION ALL
   ( SELECT
       COALESCE(kv.value, e.value) AS value
     FROM
       t
     LEFT JOIN LATERAL jsonb_each(
       CASE WHEN jsonb_typeof(t.value) = 'object' THEN t.value ELSE NULL END
     ) kv ON true
     LEFT JOIN LATERAL jsonb_array_elements(
       CASE WHEN jsonb_typeof(t.value) = 'array' THEN t.value ELSE NULL END
     ) e ON true
     WHERE
       kv.value IS NOT NULL OR e.value IS NOT NULL)
  )
SELECT EXISTS (SELECT 1 FROM t WHERE value = '"Moscow"');
```

JSON Path in PG12: one missing feature

- `.datetime()` item method (T832) not supported in PG12:

```
-- behavior required by standard
SELECT jsonb_path_query('"13.03.2019"',
'$.datetime("DD.MM.YYYY")');
jsonb_path_query
```

```
-----  
"2019-03-13"  
(1 row)
```

```
-- behavior of PG12
SELECT jsonb_path_query('"13.03.2019"',
'$.datetime("DD.MM.YYYY")');
ERROR: bad jsonpath representation
```

- <https://commitfest.postgresql.org/23/2133/>

SQL/JSON standard conformance

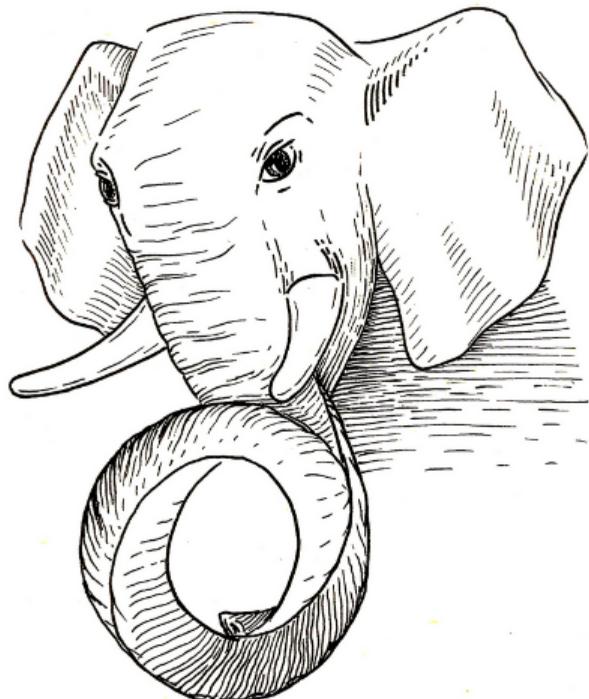
SQL/JSON feature	PostgreSQL 12	Oracle 18c	MySQL 8.0.4	SQL Server 2017
JSON PATH: 15	14/15	11/15	5/15	2/15

PostgreSQL 12 has the best implementation of JSON Path

More information about Jsonpath

<https://github.com/obartunov/sqljsondoc/blob/master/jsonpath.md>

Gentle Guide to JSONPATH in PostgreSQL



This document describes SQL/JSON implementation as committed to PostgreSQL 12, which consists of implementation of JSON Path - the JSON query language, and several functions and operators, which use the path language to work with jsonb data. Consider this document as a tutorial , the reference guide is available as a part of official PostgreSQL documentation for release 12.

Authors: Oleg Bartunov and Nikita Glukhov.

Introduction to SQL/JSON

SQL-2016 standard doesn't describes the JSON data type, but instead it introduced SQL/JSON data model (not JSON data type like XML) with string storage and path language used by certain SQL/JSON functions to query JSON. SQL/JSON data model is a sequences of items, each of which is consists of SQL scalar values with an additional SQL/JSON null value, and composite data structures using JSON arrays and objects.

JSONB 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 @>)

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.

Jsonpath queries could use existing jsonb indexes

- Find all authors with the same bookmarks as the given author

```
CREATE index ON bookmarks USING gin(jb jsonb_path_ops );
```

```
SELECT
  b1.jb->'author'
FROM
  bookmarks b1,
  bookmarks b2
WHERE
  b1.jb @@ format('$.title == %s && $.author != %s', b2.jb -> 'title', b2.jb -> 'author')::jsonpath
AND b2.jb @@ '$.author == "ant.on"'::jsonpath;
```

Seq scan: 35000 ms, Index scan: 6 ms

Jsonpath performance (simple queries)

- Test table with 3 mln rows

```
CREATE TABLE t AS
SELECT jsonb_build_object('x', jsonb_build_object('y', jsonb_build_object('z', i::text)))
FROM generate_series(1, 3000000) i;
```

```
SELECT * from t where jsonb_path_query_first(js, '$.x.y.z') = '"123"';
          js
-----
```

```
{"x": {"y": {"z": "123"}}}
(1 row)
```

Jsonpath performance (simple queries)

- Performance of arrow operators is slightly better for simple queries, but jsonpath allows more complex queries (see intra joins example).

query	time, ms
jsonb_path_query_first(js, '\$.x.y.z') = '"123"	1700
js->'x' ->'y' ->'z' = '"123"	1700
jsonb_path_query_first(js, '\$.x.y.z') ->>0 = '123'	600
js->'x' ->'y' ->>'z' = '123'	430
jsonb_path_exists(js, '\$? (\$.x.y.z == "123")')	1000
jsonb_path_match(js, '\$.x.y.z == "123"')	1000
jsonb_path_match(js, '\$.x.y.z == \$x', '{"x": "123"}')	1100
jsonb_path_match(js, '\$.x.y.z == \$x', jsonb_object(array['x'], array['123']))	1100
jsonb_path_match(js, '\$.x.y.z == \$x', jsonb_build_object('x', '123'))	2800
jsonb_extract_path(js, 'x', 'y', 'z') = '"123"	1670
jsonb_extract_path_text(js, 'x', 'y', 'z') = '123'	580

Jsonpath intra joins (joining parts of the same column)

Query: find all the actors && editors in **the same movie** (43808 out of 6378007 rows in names). Actress && editors — 7173.

- **Jsonpath:**

```
SELECT jb->'id' FROM names
WHERE jb @@ '$.roles[*] ? (@.role == "actor").title ==
            $.roles[*] ? (@.role == "editor").title'::jsonpath;
  Sequential Scan:          29748.223 ms
  Sequential Scan (parallel): 4678.925 ms
  Bitmap Index Scan (jsquery index): 2328.880 ms
```

- «old» way:

```
SELECT jb->'id' FROM names WHERE
jb @> '{"roles": [{"role": "actor"}, {"role": "editor"}]}' AND
(SELECT array_agg(r->>'title') FROM jsonb_array_elements(jb->'roles') roles(r)
 WHERE r->>'role' = 'actor') &&
(SELECT array_agg(r->>'title') FROM jsonb_array_elements(jb->'roles') roles(r)
 WHERE r->>'role' = 'editor');
```

Sequential scan: 20233.032 ms
Bitmap Index Scan: 3860.534 ms

```
"id": ....
"roles": [
  {
    "role": "actor",
    "title": ....
  }
  ...
]
```

Jsonpath intra joins (joining parts of the same column)

Jsonpath version is the fastest, since it has its own executor, no overheads.

- **Jsonpath:**

Sequential Scan:	29748.223 ms
Sequential Scan (parallel):	4678.925 ms
Bitmap Index Scan (jsquery index):	2328.880 ms

- **Arrow (old way):**

Sequential scan:	20233.032 ms
Bitmap Index Scan:	3860.534 ms

- **Relational way:**

Sequential Scan:	34840.434 ms
Sequential Scan (parallel, 6):	4233.829 ms
Bitmap Index Scan:	13745.517 ms
Bitmap Index Scan(parallel, 6):	3807.380 ms

- **Mongo:**

```
"id": ....  
"roles": [  
  {  
    "role": "actor",  
    "title": ....  
  }  
  ...  
  {}  
]
```

Roadmap (see Addendums)

- PG13: SQL/JSON functions from SQL-2016 standard
- PG13: datetime support in JSON Path (complete T832)
- PG13: Planner support functions
- PG13: Parameters for opclasses - jsonpath to specify parts of jsonb to index
- PG13: Jquery GIN opclasses to core
- PG13: Extend jsonpath syntax
 - array,object,sequence construction
 - object subscripting, lambda expressions
 - user-defined item methods and functions
- COPY with support of jsonpath
- Make one JSON data type !



Who need Mongo ?

**NoSQL Postgres
rulezz !**

Good Roadmap !

Summary

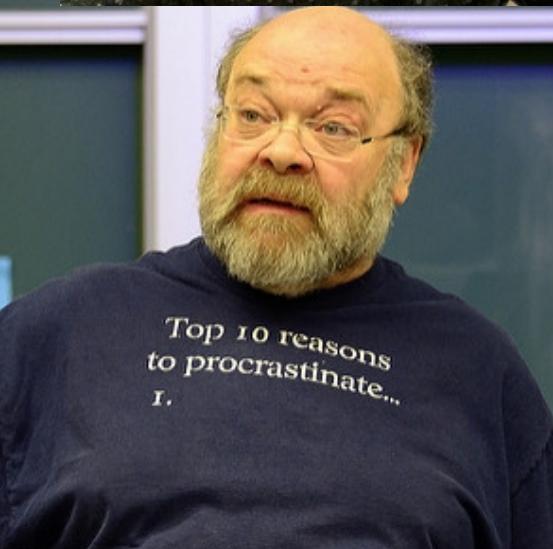
- PostgreSQL is already good NoSQL database
 - Great roadmap
- SQL/JSON provides better flexibility and interoperability
 - JSON Path implementation (PG12) is the best !
- Jsonpath is superior for complex queries (thanks to internal executor)

Move from NoSQL to Postgres !

References

- 1) This talk: <http://www.sai.msu.su/~megera/postgres/talks/jsonpath-china-2019.pdf>
- 2) Technical Report (SQL/JSON) - available for free
http://standards.iso.org/i/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip
- 3) Gentle introduction to JSON Path in PostgreSQL
<https://github.com/obartunov/sqljsonondoc/blob/master/jsonpath.md>
- 4) Jquery extension: <https://github.com/postgrespro/jqry/tree/sqljson>
<http://www.sai.msu.su/~megera/postgres/talks/pgconfeu-2014-jqry.pdf>
- 5) Play online with jsonpath
<http://sqlfiddle.postgrespro.ru/#!21/0/2379>
- 6) Parameters for opclasses
http://www.sai.msu.su/~megera/postgres/talks/opclass_pgconf.ru-2018.pdf
- 7) IMDB tables: <http://www.sai.msu.su/~megera/postgres/files/imdb/imdb/>

NOSQL POSTGRES IS A COMMUNITY PROJECT



ALL
YOU
NEED
IS
POSTGRES



ADDENDUM I

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

JSON_TABLE — relational view of json

- Table with rooms from 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,  
            rooms int  
        )  
    )) 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)

ADDENDUM II

Parameters for Opclasses

Parameters for opclasses

Operator class 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
- Indexing of non-atomic data (arrays, json[b], tsvector,...)
 - Specify what part of column to index — partial index only filters rows
- Use different algorithms to index
 - Specify what to use depending on data

Parameters for opclasses: syntax

- Parenthesized parameters added after column's opclass. Default opclass can be specified with DEFAULT keyword:

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

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

ADDENDUM III

Planner support for jsonpath

Planner support function for jsonpath functions

- PG12+: API for planner support functions that lets them create derived index conditions for their functions.

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

- `jsonb_path_match()` transforms to `jsonb @@ jsonpath` (uses index !)

Planner support function for jsonpath functions

- PG12+: API for planner support functions that lets them create derived index conditions for their functions.

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

Nested Loop

```
-> Seq Scan on t t2  
-> Bitmap Heap Scan on 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)
```

Planner support function for jsonpath functions

- PG12+: API for planner support functions that lets them create derived index conditions for their functions.

```
jsonb_path_match(b1.jb,  
                  '$.title == $title && $.author != $author',  
                  vars => b2.jb)  
AND b2.jb ->> 'author' = 'ant.on'
```

=>

```
b1.jb @@ jsonpath_embed_vars('$.title == $title &&  
                               $.author != $author', b2.jb)  
AND b2.jb @@ '$.author == "ant.on"'::jsonpath
```

ADDENDUM IV

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

- 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
```

Jsonpath syntax extensions

- User-defined item methods and functions (contrib/jsonpathx):

```
CREATE FUNCTION map(jsonpath_fcxt) RETURNS int8
AS 'MODULE_PATHNAME', 'jsonpath_map' LANGUAGE C;
```

```
typedef struct JsonPathFuncContext
{
    JsonPathExecContext *cxt;
    JsonValueList *result;
    const char      *funcname;
    JsonItem        *jb;      /* @ */
    JsonItem        *item;    /* NULL => func, non-NUL => method */
    JsonPathItem   *args;
    void           **argscache;
    int             nargs;
} JsonPathFuncContext;
```

ADDENDUM V

Performance of Intra joins

Jsonpath intra joins (joining parts of the same column)



mongoDB®

3808 ms

```
db.names.find({
  "roles.role": { $all: [ "actor", "editor" ] }, // find by index on "roles.role"
  $expr: {
    $setIntersection: [
      { $map: { // '$.roles[*] ? (@.role == "actor").title'
        input: {
          $filter: { // '$.roles[*] ? (@.role == "actor")'
            input: "$roles",
            as: "r1",
            cond: { $eq: ["$$r1.role", "actor"] }
          }
        },
        as: "t1",
        in: "$$t1.title"
      }},
      { $map: { // '$.roles[*] ? (@.role == "editor").title'
        input: {
          $filter: [ // '$.roles[*] ? (@.role == "editor")'
            input: "$roles",
            as: "r2",
            cond: { $eq: ["$$r2.role", "editor"] }
          }
        },
        as: "t2",
        in: "$$t2.title"
      }}
    ]
  }
}).explain("executionStats").executionStats.executionTimeMillis
```

PostgresPro

Jsonpath intra joins (joining parts of the same column)

- Query: find all the actors who were editors in **the same movie** (6378007 rows in names).
- Relational analogue of names table:

```
CREATE TABLE roles AS
SELECT
    id,
    r->>'role' AS "role",
    r->>'title' AS "title",
    r->>'character' AS "character",
    r->'ranks' AS "ranks"
FROM
    names,
    jsonb_array_elements(jb->'roles') roles(r);

CREATE INDEX ON roles(role);
CREATE INDEX ON roles (id, title, role); -- composite btree index
```

```
\d+
public | names          | table | 3750 MB
public | roles          | table | 5830 MB
```

```
\di+
public | names_jb_idx   | index | names | 1439 MB
public | roles_id_title_role_idx | index | roles | 4710 MB
```

Jsonpath intra joins (joining parts of the same column)

- Query: find all the actors who were editors in **the same movie** (6378007 rows in names).
- Relational analogue of names table:

```
SELECT DISTINCT r1.id
FROM roles r1
WHERE r1.role = 'editor' AND EXISTS (
    SELECT FROM roles r2 WHERE r2.id = r1.id AND r2.title = r1.title AND r2.role = 'actor'
);
```

Sequential Scan:	34840.434 ms
Sequential Scan (parallel,6):	4233.829 ms
Bitmap Index Scan:	13745.517 ms
Bitmap Index Scan(parallel,6):	3807.380 ms

ADDENDUM VI

Two floors house

```
CREATE TABLE house(js) 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}  
      ]  
    }  
  ]  
};
```