

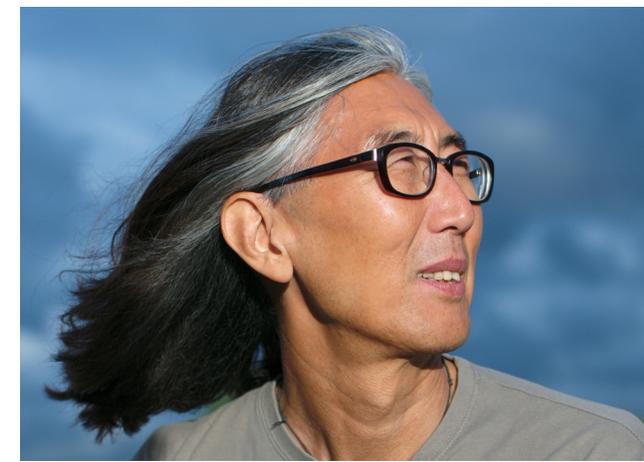
SQL/JSON in PostgreSQL



Oleg Bartunov
Postgres Professional,
Moscow University



Oleg Bartunov, major PostgreSQL contributor, since 1995



CEO, Postgres Professional
Moscow University

obartunov@postgrespro.ru





PostgreSQL Universal Database

How to choose a right database ?

- People usually choose a database looking on
 - Functionality, Performance
 - Availability - License, price
 - Local expertise, Personal experience
 - Compatibility to existing environment
 - Support
 - Other
- People start to think about one important feature of database after the project needs
 - Need new functionality, Better performance
- Project is in production, no way to change database
 - Starting to use various ugly «solutions»
 - System works, but looks pretty strange

Your system with wrong database

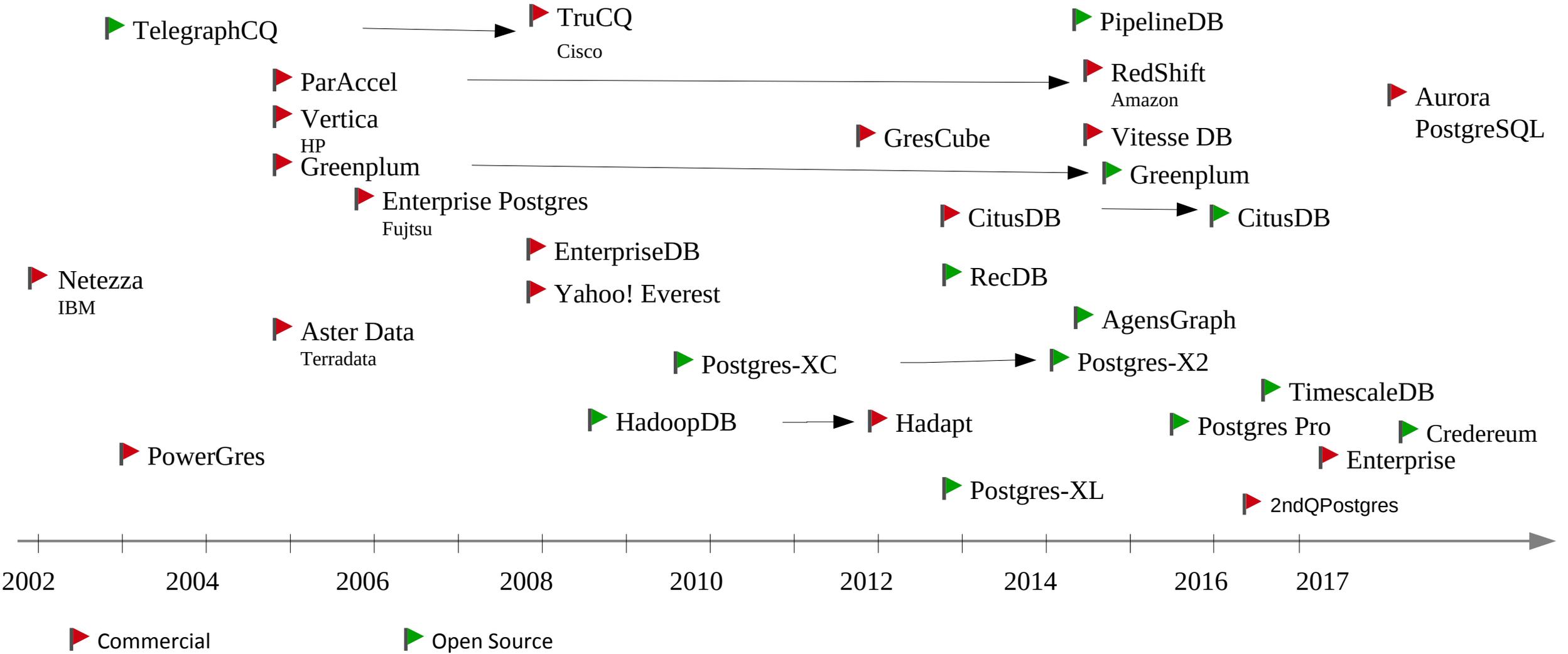


PostgreSQL Universal Database

- Any project could start with PostgreSQL
- PostgreSQL is a reliable and stable database with rich functionality and long history
- PostgreSQL has liberal BSD license, cross platform (~30)
- Developed by international community, no vendor lock

- PostgreSQL is **EXTENSIBLE**, this is the very important feature, which people miss ! It allow database to support
 - New workloads
 - New functionality
 - New environment
 - Often without restarting a server, no need core programmer.

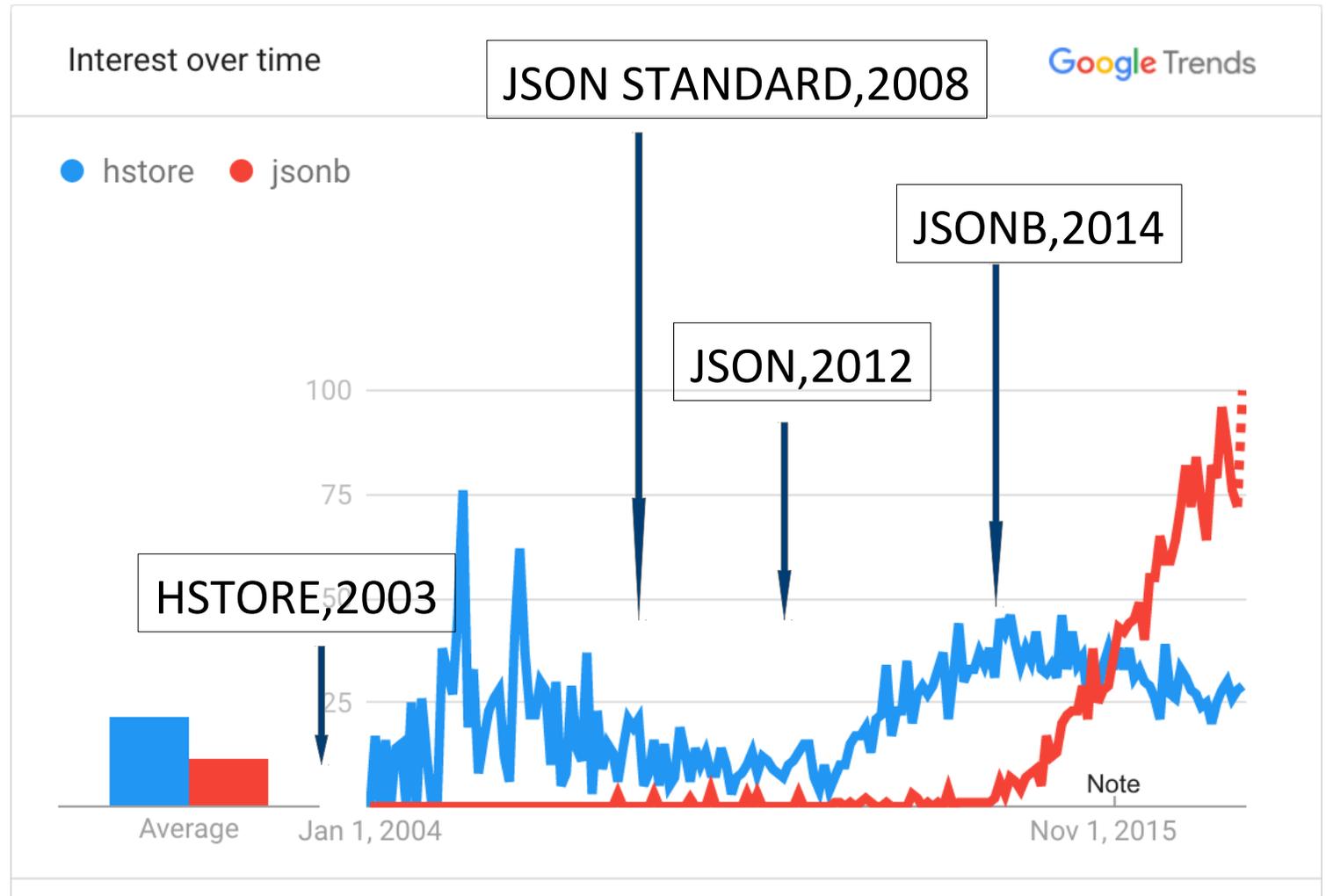
PostgreSQL: OLTP, MPP, OLAP, CLOUD, GIS, STREAM, TIMESERIES, GPU, BLOCKCHAIN





NoSQL PostgreSQL is MATURE

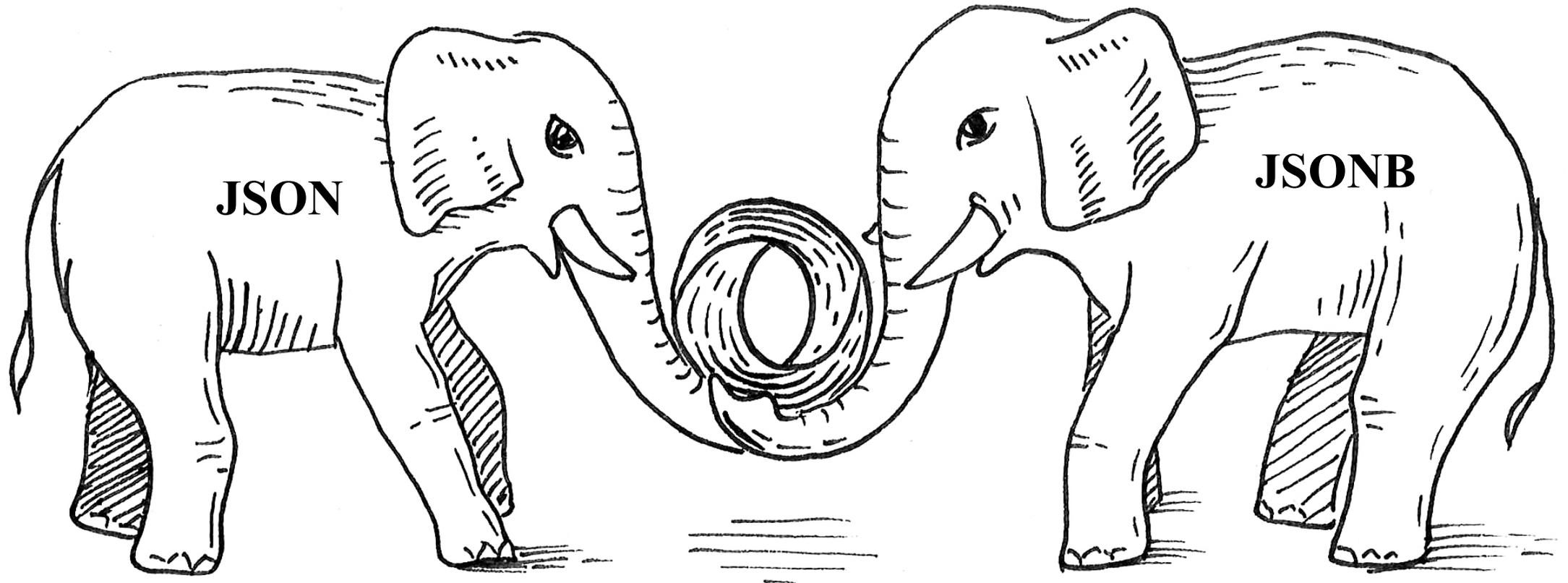
- HSTORE — binary key-value storage, index support
- 2003 — initial release
- 2006 — part of PostgreSQL



Two JSON data types !!!

- Textual storage «as is»

- Binary storage, index support



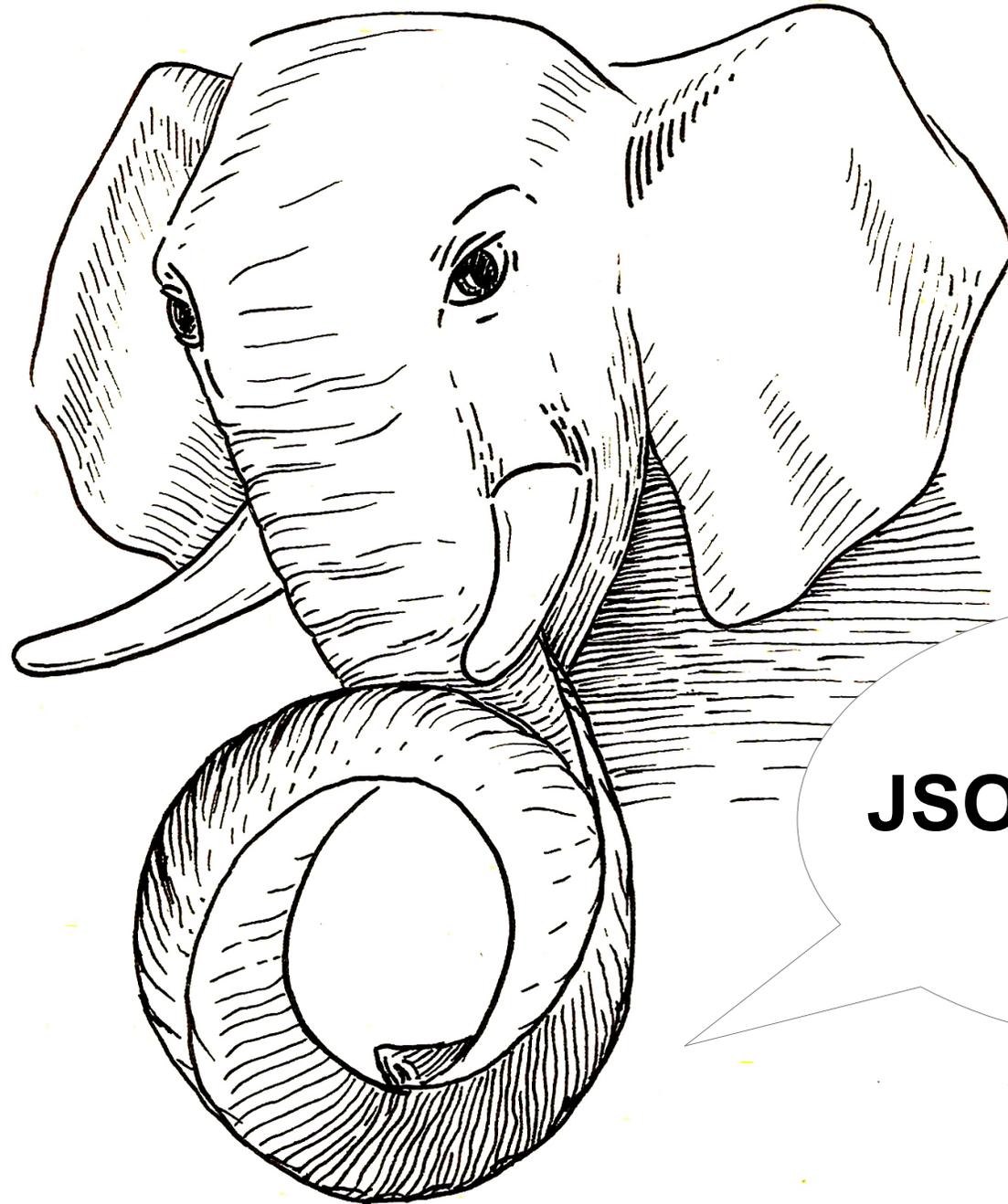
- A lot of functionality !

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;
```

```
-----+-----
{"cc":0, "aa": 2, "aa":1, "b":1} | {"b": 1, "aa": 1, "cc": 0}
(1 row)
```

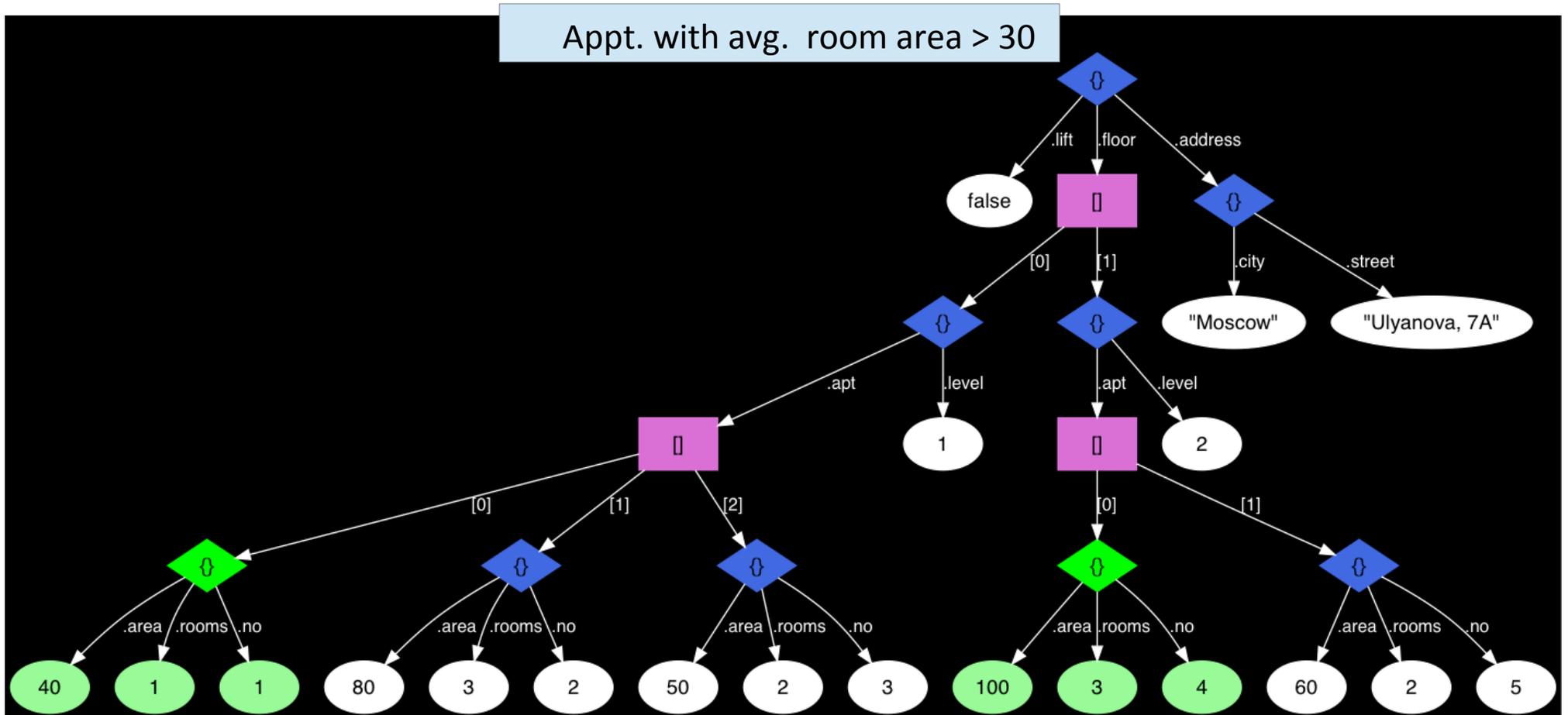
- json: textual storage «as is»
- Jsonb: binary storage, no need to parse, has index support
- jsonb: no whitespaces, no duplicated keys (last key win)
- jsonb: keys are sorted by (length, key)
- A rich set of functions to work with jsonb (`\df jsonb*`)



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

JSONB is great, BUT

1) JSONB is a «black box» for SQL, no good query language.



Find something «red»

- Table "public.js_test"

Column	Type	Modifiers
id	integer	not null
value	jsonb	

```
select * from js_test;
```

id	value
1	[1, "a", true, {"b": "c", "f": false}]
2	{"a": "blue", "t": [{"color": "red", "width": 100}]}
3	[{"color": "red", "width": 100}]
4	{"color": "red", "width": 100}
5	{"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"}
6	{"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}
7	{"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}
8	{"a": "blue", "t": [{"color": "green", "width": 100}]}
9	{"color": "green", "value": "red", "width": 100}

(9 rows)

Find something «red»

- **VERY COMPLEX SQL QUERY**

```
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;
```

id	value
2	{"a": "blue", "t": [{"color": "red", "width": 100}]}
3	[{"color": "red", "width": 100}]
4	{"color": "red", "width": 100}
5	{"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"}
6	{"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}

(5 rows)



Find something «red»

```
• 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;
```

- **Jsquery**

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

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

- A language to query jsonb data type
- Search in nested objects and arrays
- More comparison operators with indexes support



JSONB is a «black box» for SQL

Need good query language

JSON in SQL-2016

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



JSON in SQL-2016

- ISO/IEC 9075-2:2016(E) - <https://www.iso.org/standard/63556.html>
- BNF
<https://github.com/elliotchance/sqltest/blob/master/standards/2016/bnf.txt>
- Discussed at Developers meeting Jan 28, 2017 in Brussels
- [Post -hackers, Feb 28, 2017](#) (March commitfest)
«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 ...»
- Patch was too big (now about 16,000 loc) and too late for Postgres 10 :(



SQL/JSON in PostgreSQL

- SQL Standard describes XML data type, but not JSON data type
- SQL/JSON describes a JSON data model for SQL, uses string to store

- PostgreSQL implementation uses native data types
 - JSON, JSONB as ORDERED and UNIQUE KEYS
 - jsonpath data type for SQL/JSON path language
 - nine functions, implemented as SQL CLAUSES



SQL/JSON in PostgreSQL

- **Jsonpath** provides an ability to operate (in standard specified way) with json structure at SQL-language level

- Dot notation — \$.a.b.c
- \$ - the current context element
- Array - [*]
- Filter ? - \$.a.b.c ? (@.x > 10)
- @ - current context in filter expression
- Methods - \$.a.b.c.x.type()
 - type(), size(), double(), ceiling(), floor(), abs(), datetime(), keyvalue()

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



Why JSON path is a type ?

Standard permits only string literals in JSON path specification.

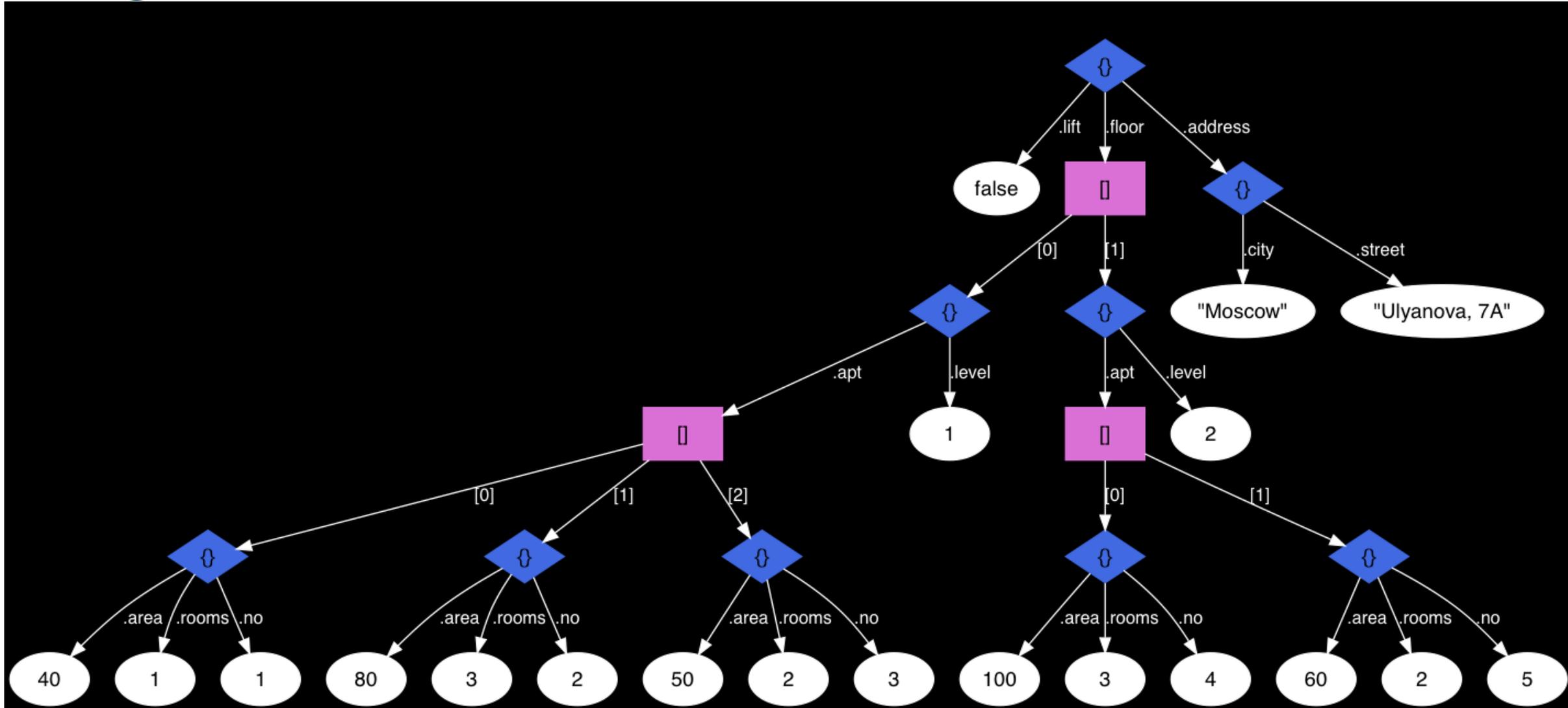
WHY a data type ?

- 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,...)

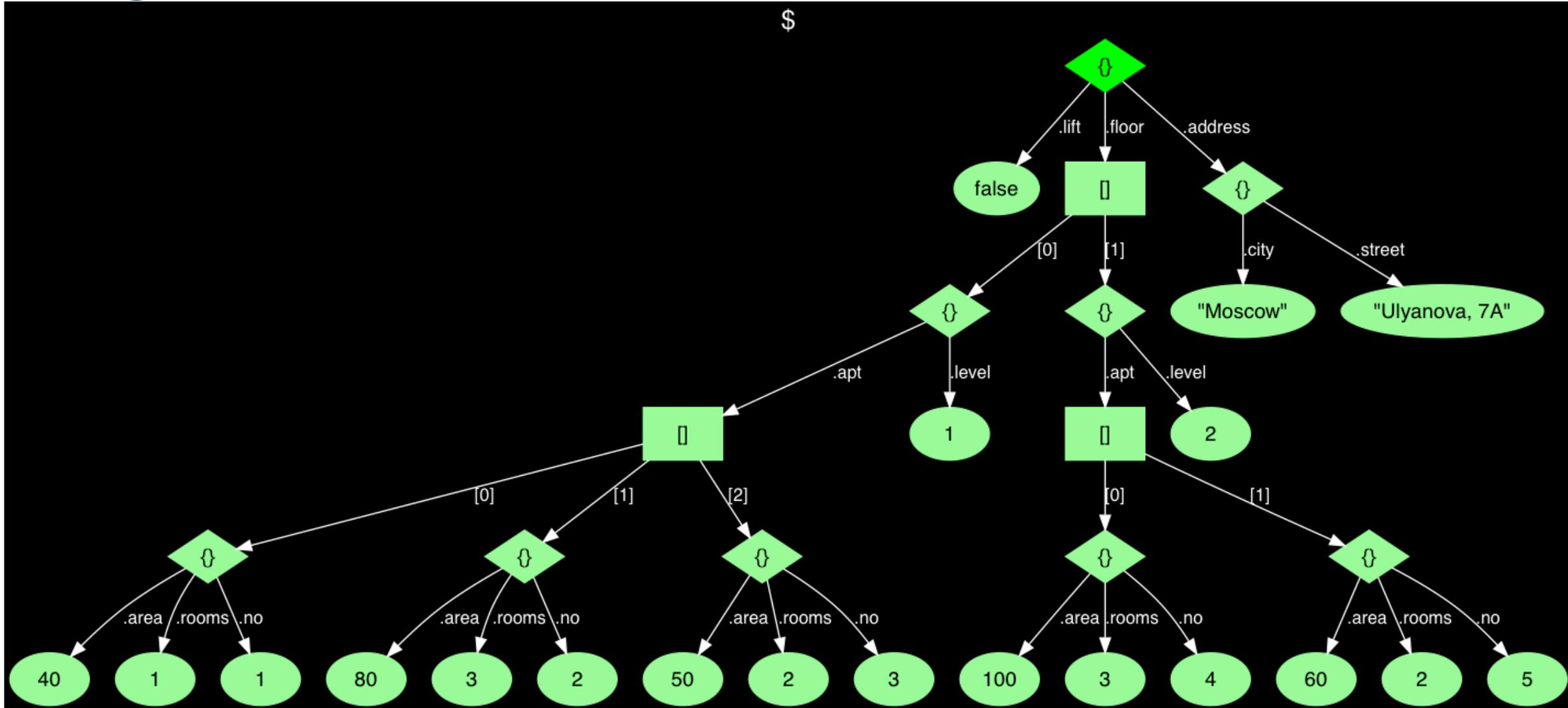
Visual guide on jsonpath

```
{
  "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": 50, "rooms": 2}
      ]
    },
    {
      "level": 2,
      "apt": [
        {"no": 4, "area": 100, "rooms": 3},
        {"no": 5, "area": 60, "rooms": 2}
      ]
    }
  ]
}
```

2-floors house

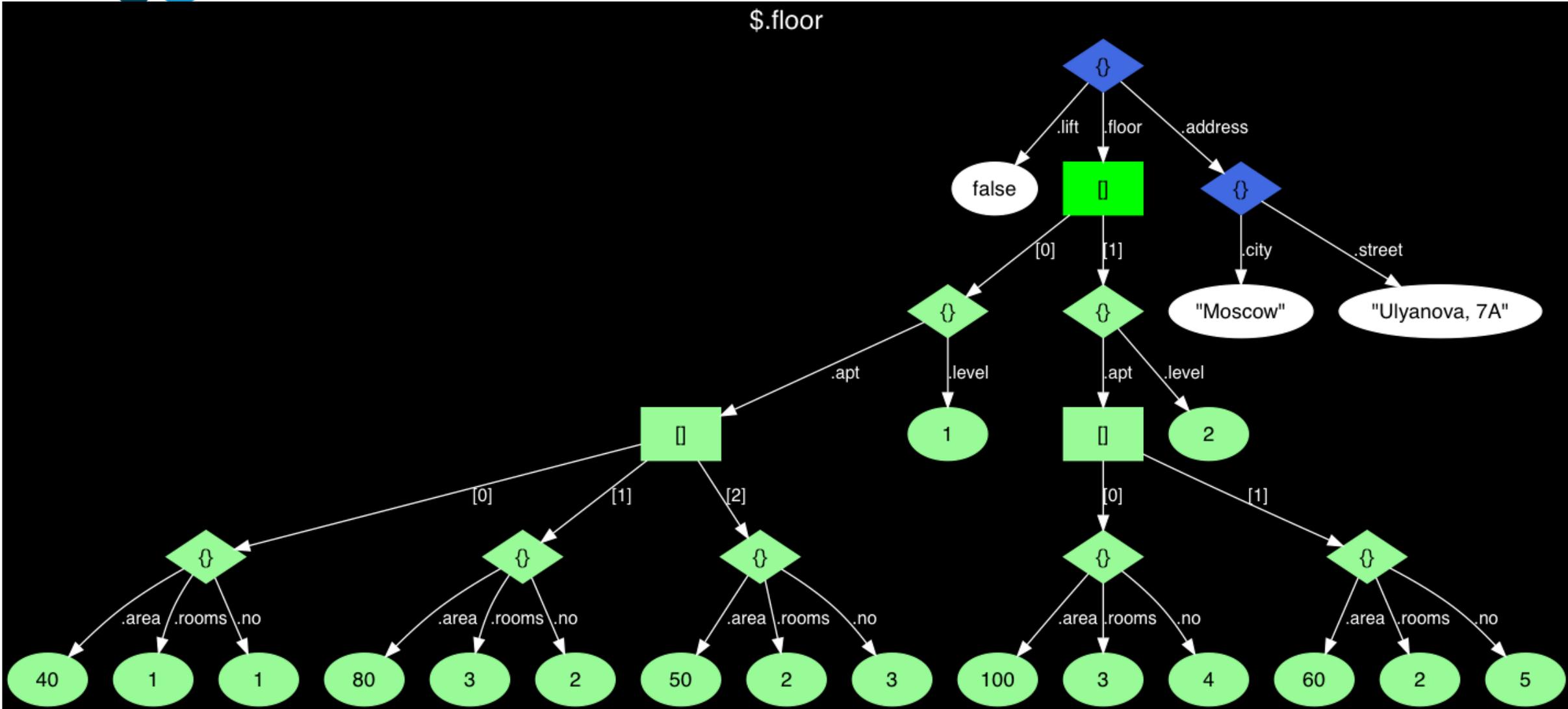


Extract everything



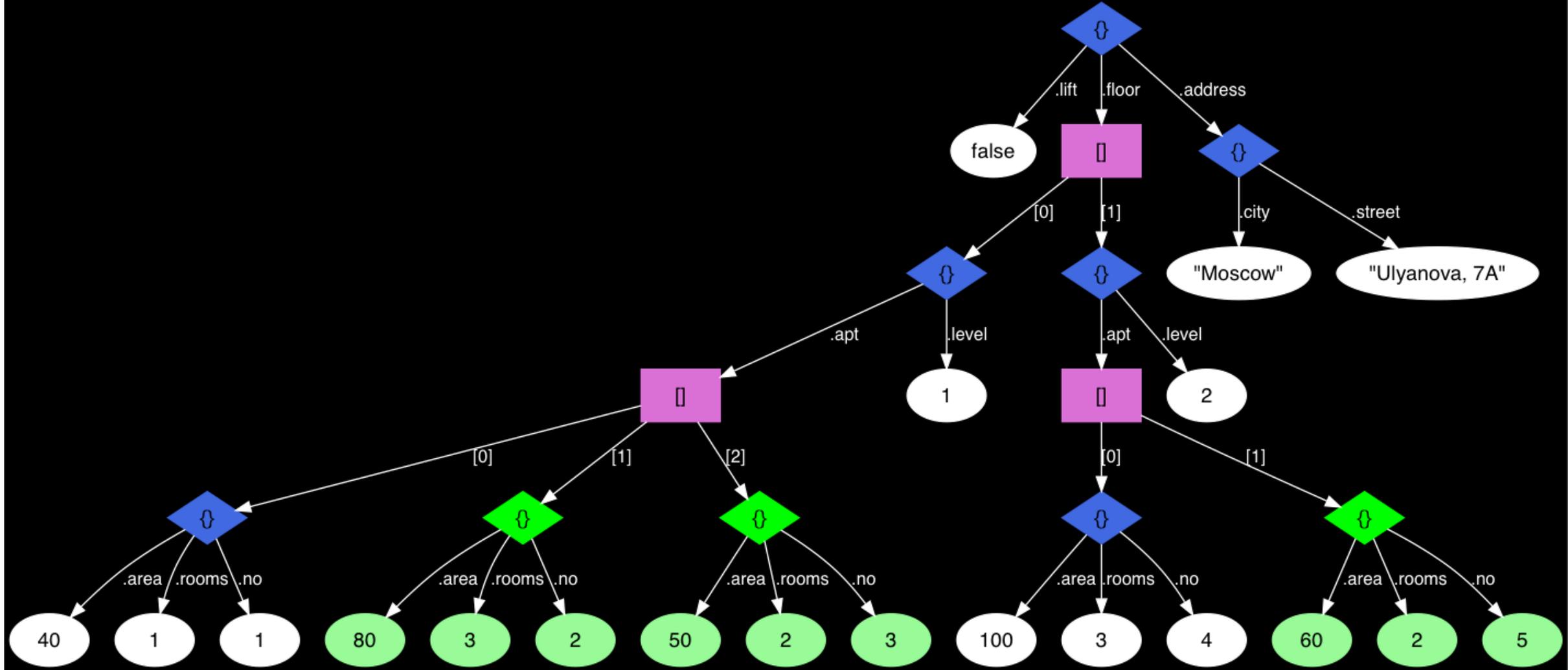
\$.floor

\$.floor



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

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



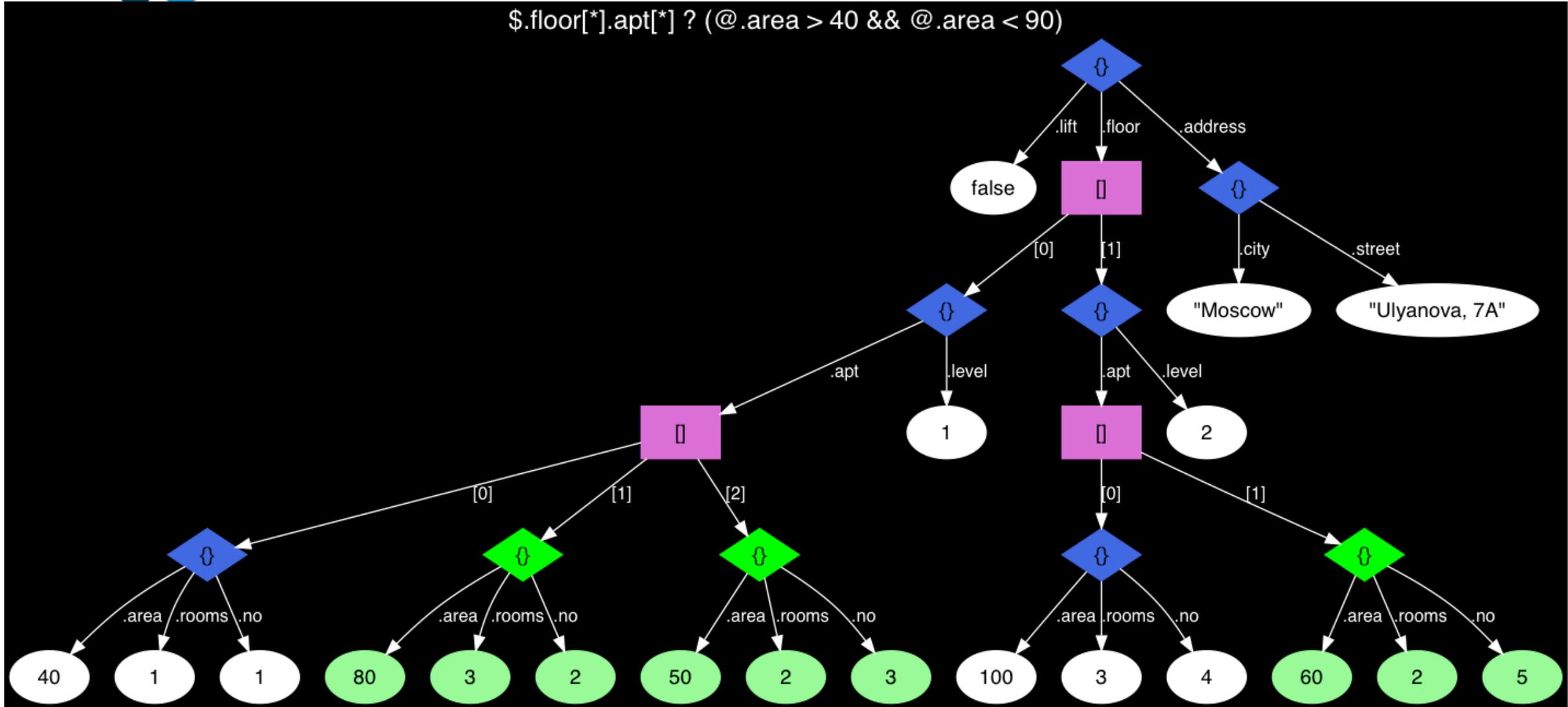


```
$.floor[0, 1].apt[1 to last]
```

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

```
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) apts(apt)) apts(apt);
```

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



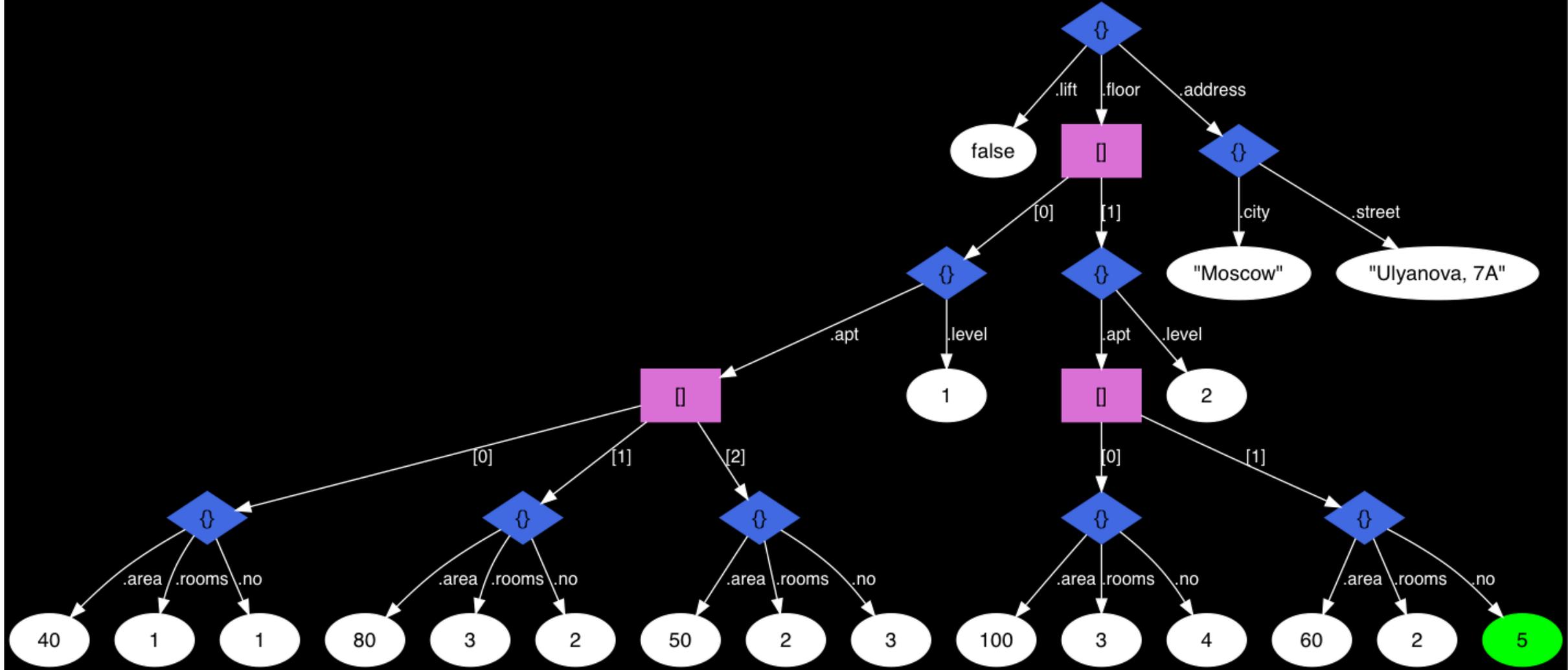


`$.floor[*].apt[*] ? (@.area >40 && @.area < 90)`

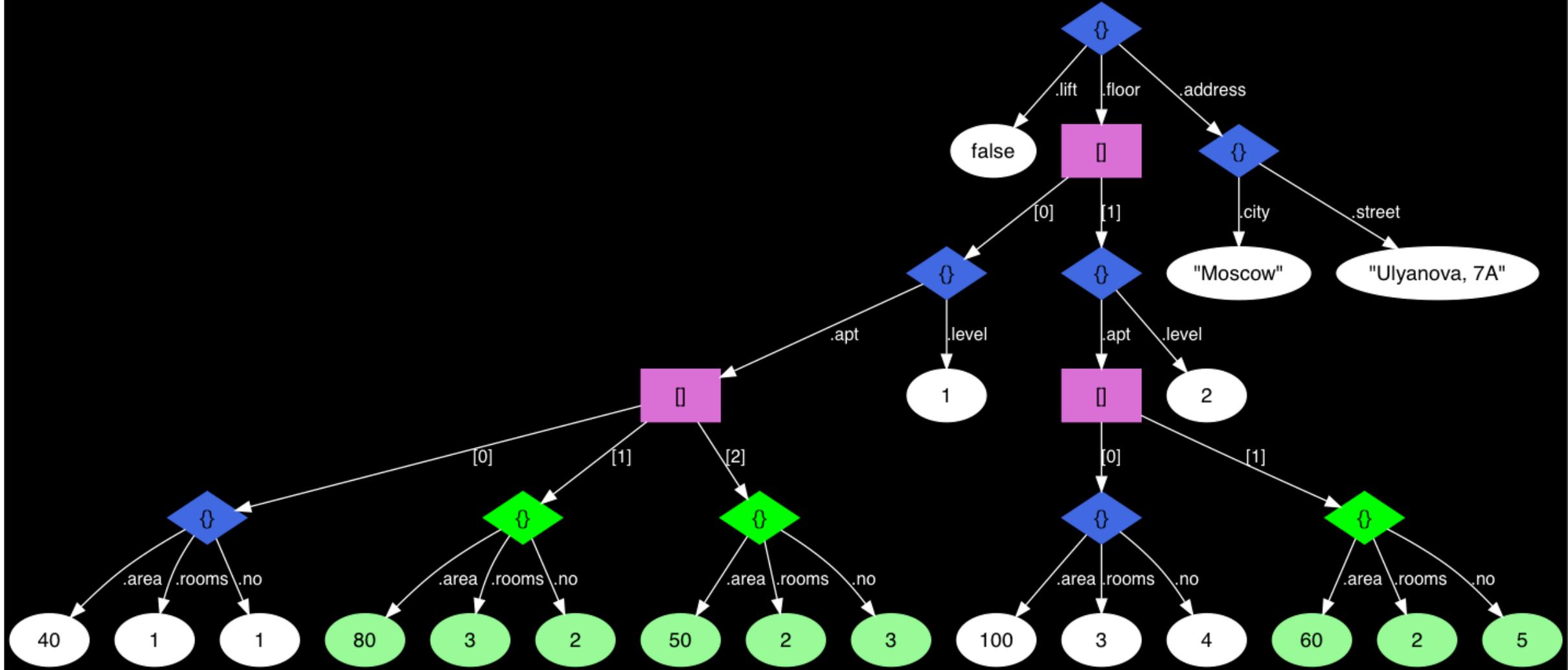
```
SELECT JSON_QUERY(js, '$.floor[*].apt[*] ? (@.area > $min && @.area < $max)'
PASSING 40 AS min, 90 AS max WITH WRAPPER) FROM house;
```

```
SELECT jsonb_agg(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;
```

`$.floor[*] ? (@.level > 1).apt[*] ? (@.area > 40 && @.area < 90).no`



`$.floor[*].apt[*] ? (@.* == 2)`





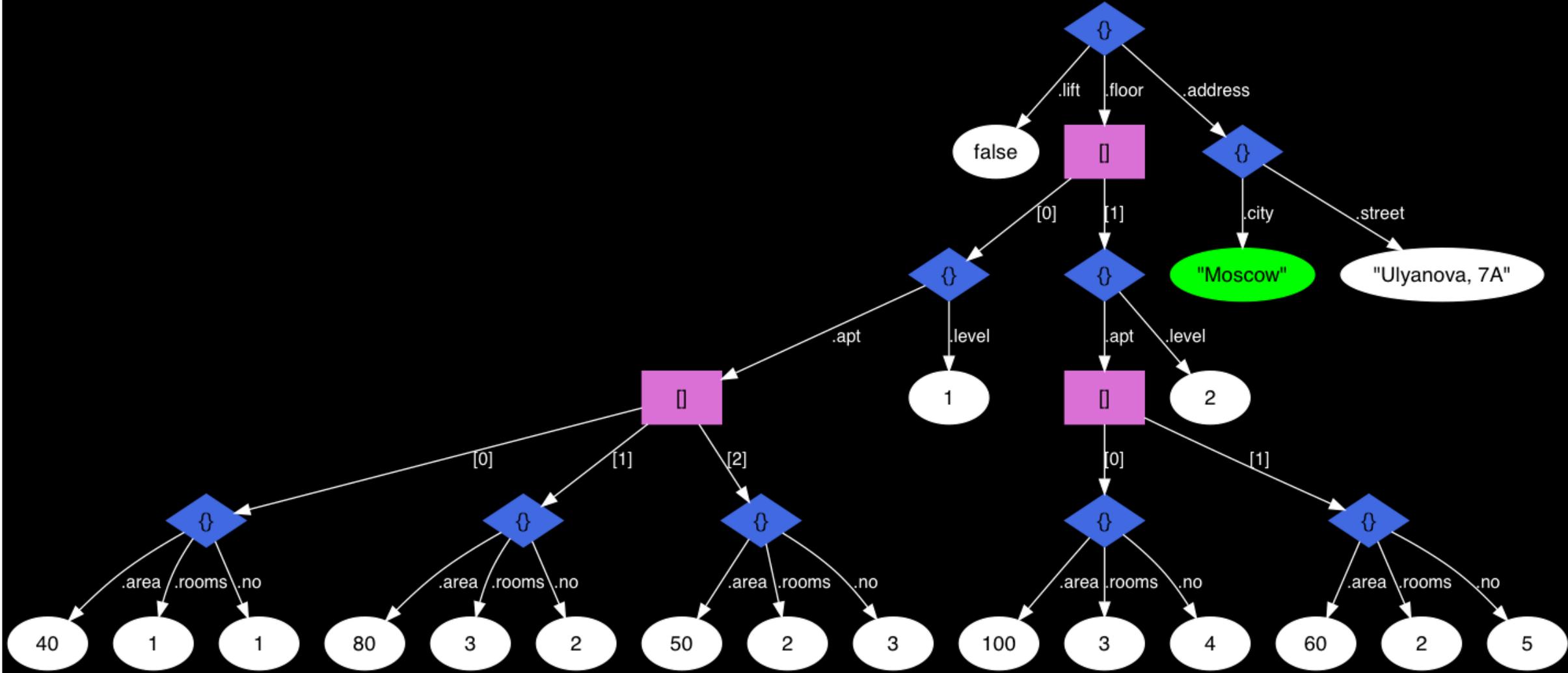
`$.floor[*].apt[*] ? (@.* == 2)`

```
SELECT JSON_QUERY(js, '$.floor[*].apt[*] ? (@.* == 2)' WITH WRAPPER) FROM house;

SELECT apt
FROM (SELECT jsonb_array_elements(jsonb_array_elements(js->'floor')->'apt')
      FROM house) apts(apt)
WHERE '2' = ANY(SELECT (jsonb_each(apt)).value);
```

Extension: wildcard search

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





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

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

```
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
```

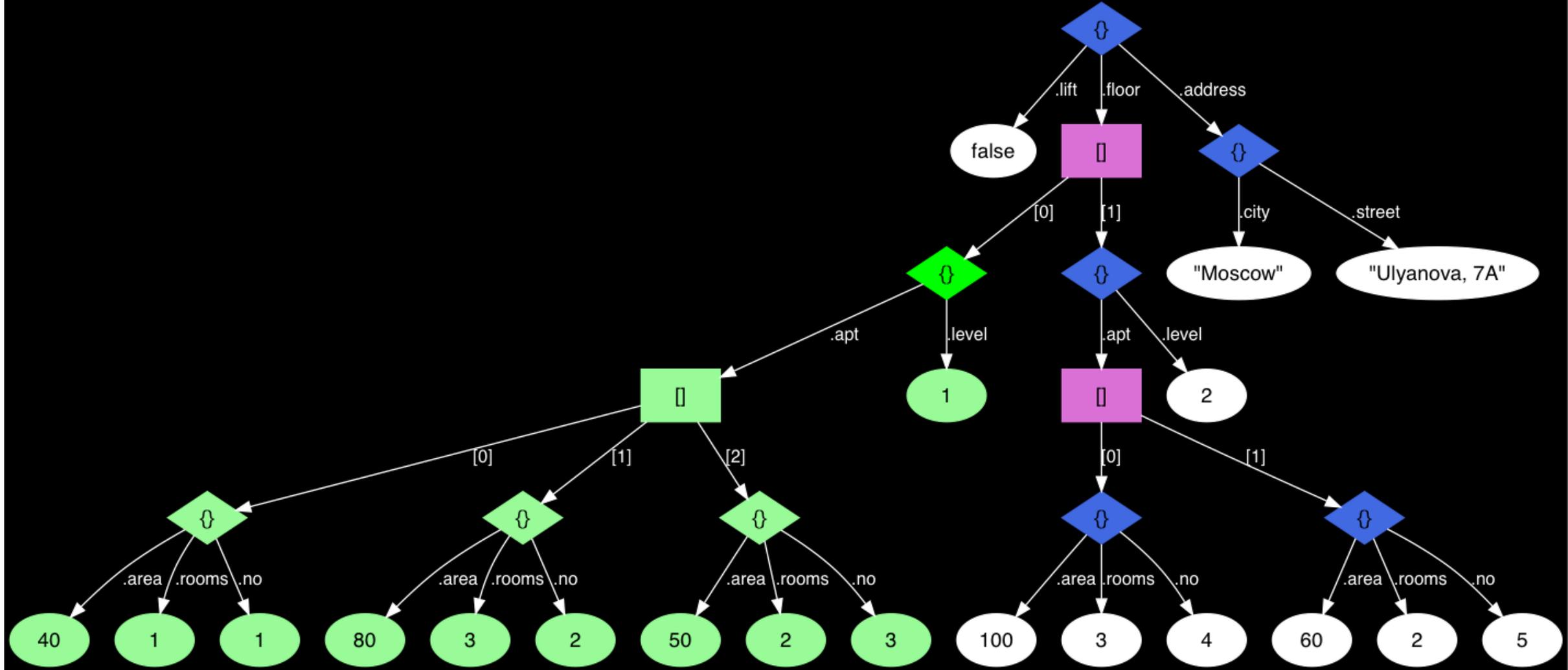
```
WHERE
```

```
  kv.value IS NOT NULL OR e.value IS NOT NULL)
```

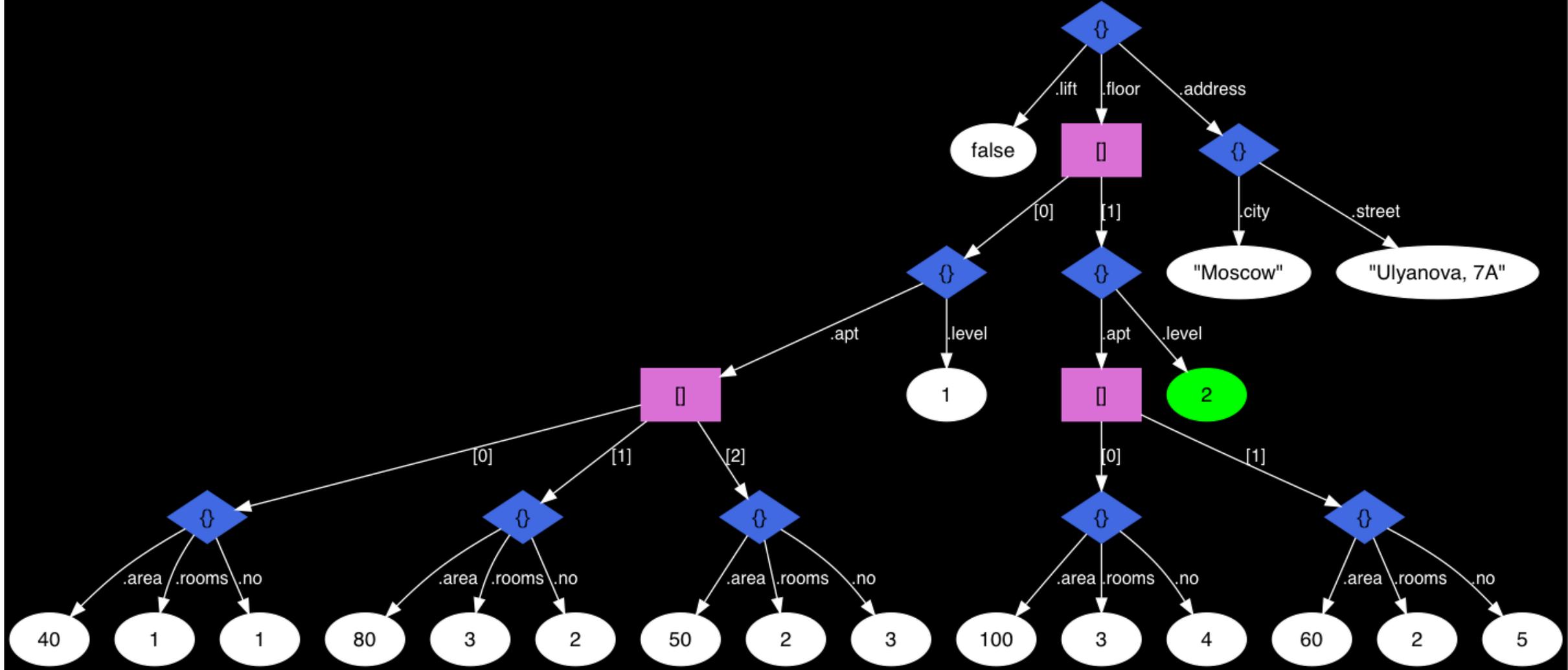
```
)
```

```
SELECT EXISTS (SELECT 1 FROM t WHERE value = "Moscow");
```

`$.floor[*] ? (@.apt.size() > 2)`



\$.floor[*] ? (@.apt[*].area >= 100).level
 \$.floor[*] ? (exists(@.apt[*] ? (@.area >= 100))).level



Find something «red»

```

• 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;

```

• Jquery

```

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

```

• **SQL/JSON 2016**

```

SELECT * FROM js_test WHERE
JSON_EXISTS( value, '$.*.color ?
(@ == "red")');

```



JSON PATH is COOL !

JSON PATH is COOL !
Let's Extend it !
JSON[b] OP JSONPATH

jsonpath operators: exists

- Exists: `json[b] @? jsonpath` returns bool.

```
SELECT '{"foo": [1,2,3]}'::jsonb @? '$.foo[*] ? (@ > 2)';
```

```
?column?
```

```
-----
```

```
t
```

```
(1 row)
```

```
SELECT '{"foo": [1,2,3]}'::jsonb @? '$.foo[*] ? (@ > 3)';
```

```
?column?
```

```
-----
```

```
f
```

```
(1 row)
```

jsonpath operators: match

- Match: `json[b] @~ jsonpath` returns `bool`.
Root expression must be a predicate (extension of the standard).

```
SELECT '{"foo": [1,2,3]} '::jsonb @~ '$.foo[*] > 2';
```

```
?column?
```

```
-----
```

```
t
```

```
(1 row)
```

```
SELECT '{"foo": [1,2,3]} '::jsonb @~ '$.foo[*] > 3';
```

```
?column?
```

```
-----
```

```
f
```

```
(1 row)
```



jsonpath operators: query

- Query: `json[b] @* jsonpath` returns setof `json[b]`.

```
SELECT '{"foo": [1,2,3]}'::jsonb @* '$.foo[*] ? (@ > 1)';
```

```
?column?
```

```
-----
```

```
2
```

```
3
```

```
(2 rows)
```



jsonpath operators

- `jsonb @? jsonpath` and `jsonb @~ jsonpath` are fast as `jsonb @> jsonb` (for equality operation)
- But, `jsonpath` supports more complex expressions:
Delicious bookmarks dataset

```
SELECT COUNT(*) FROM bookmarks
WHERE jb @~ '$.updated.datetime("Dy, dd MON YYYY
HH24:MI:SS") > "2009-09-11".datetime("YYYY-MM-DD")';
Count
-----
484341
(1 row)
```



jsonpath operators: index support

- Exists @? and match @~ operators can be speeded up by GIN index using built-in jsonb_ops or jsonb_path_ops.

```
SELECT COUNT(*) FROM bookmarks
WHERE jb @? '$.tags[*] ? (@.term == "NYC")';
```

QUERY PLAN

```
-----
Aggregate (actual time=0.529..0.529 rows=1 loops=1)
  -> Bitmap Heap Scan on bookmarks (actual time=0.080..0.502 rows=285 loops=1)
        Recheck Cond: (jb @? '$."tags"[*]?(@."term" == "NYC")'::jsonpath)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on bookmarks_jb_path_idx (actual time=0.045..0.045
rows=285 loops=1)
                Index Cond: (jb @? '$."tags"[*]?(@."term" == "NYC")'::jsonpath)
Planning time: 0.053 ms
Execution time: 0.553 ms
(8 rows)
```



jsonpath operators: index support

- Exists @? and match @~ operators can be accelerated by GIN index using built-in jsonb_ops or jsonb_path_ops.

```
SELECT COUNT(*) FROM bookmarks
WHERE jb @~ '$.tags[*].term == "NYC"';
```

QUERY PLAN

```
-----
Aggregate (actual time=0.930..0.930 rows=1 loops=1)
  -> Bitmap Heap Scan on bookmarks (actual time=0.133..0.884 rows=285 loops=1)
        Recheck Cond: (jb @~ '($.tags[*].term == "NYC")'::jsonpath)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on bookmarks_jb_path_idx (actual time=0.073..0.073
rows=285 loops=1)
              Index Cond: (jb @~ '($.tags[*].term == "NYC")'::jsonpath)
Planning time: 0.135 ms
Execution time: 0.973 ms
(8 rows)
```



Jsonpath operators: joins by index

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

```
SELECT JSON_VALUE(b1.jb, '$.author')
FROM bookmarks b1, bookmarks b2
WHERE JSON_EXISTS(b1.jb,
    '$ ? (@.title == $bookmark.title &&
        @.author != $bookmark.author)' PASSING b2.jb AS bookmark) AND
    JSON_VALUE(b2.jb, '$.author') = 'ant.on';
```

SEQ.SCAN: 6.7 sec

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

```
create index bookmarks_jb_path_idx on bookmarks using gin(jb jsonb_path_ops );
```

INDEX SCAN: 0.2 ms **33500 times faster !**



JSON PATH is COOL !



SQL/JSON in PostgreSQL

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

Mostly the same as json[b] construction 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: JSON_OBJECT

JSON_OBJECT -construct a JSON[b] object

Syntax:

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

```
json_value_expression ::=  
  expression [ FORMAT JSON ]
```



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)
```



SQL/JSON: JSON_OBJECTAGG

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

Syntax:

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

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)
```



SQL/JSON: JSON_ARRAY

JSON_ARRAY - construct a JSON[b] array

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

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

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



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', 123, TRUE, ARRAY[1,2,3],  
    '{"a": 1}'::jsonb, '[1, {"c": 3}]' FORMAT JSON  
);
```

?column?

```
-----  
["string", 123, true, [1,2,3], {"a": 1}, [1, {"c": 3}]]  
(1 row)
```

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

```
-----  
[1, 2, 3]  
(1 row)
```



SQL/JSON: JSON_ARRAY

NULL ON NULL clause is used for preserving NULL elements which are will be omitted otherwise (ABSENT ON NULL is by default).

```
=# SELECT JSON_ARRAY(1, NULL, 'a');
```

```
?column?
```

```
-----
```

```
[1, "a"]
```

```
(1 row)
```

```
=# SELECT JSON_ARRAY(1, NULL, 'a' NULL ON NULL);
```

```
?column?
```

```
-----
```

```
[1, null, "a"]
```

```
(1 row)
```



SQL/JSON: JSON_ARRAYAGG

JSON_ARRAYAGG - aggregates values as JSON[b] array.

Syntax:

```
JSON_ARRAYAGG (  
    expression [ FORMAT JSON ]  
    [ { NULL | ABSENT } ON NULL ]  
    [ RETURNING data_type [ FORMAT JSON ] ]  
)
```



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)
```



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



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)



SQL/JSON examples: JSON_QUERY

SELECT

```
JSON_QUERY(js FORMAT JSONB, '$'),  
JSON_QUERY(js FORMAT JSONB, '$' WITHOUT WRAPPER),  
JSON_QUERY(js FORMAT JSONB, '$' WITH CONDITIONAL WRAPPER),  
JSON_QUERY(js FORMAT JSONB, '$' WITH UNCONDITIONAL ARRAY WRAPPER),  
JSON_QUERY(js FORMAT 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)



SQL/JSON examples: Constraints

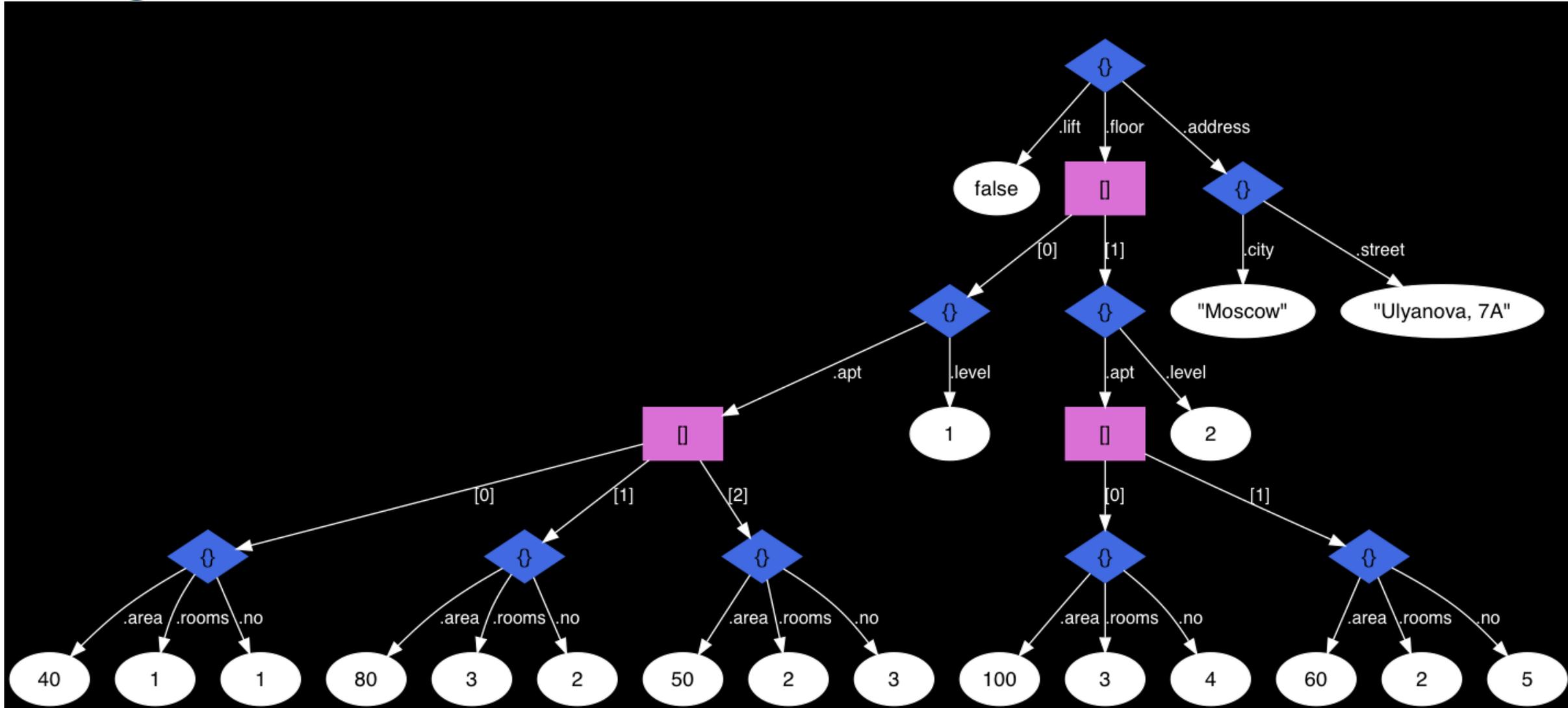
```
CREATE TABLE test_json_constraints (  
    js text,  
    i int,  
    x jsonb DEFAULT JSON_QUERY(jsonb '[1,2]', '$[*]' WITH WRAPPER)  
    CONSTRAINT test_json_constraint1  
        CHECK (js IS JSON)  
    CONSTRAINT test_json_constraint2  
CHECK (JSON_EXISTS(js FORMAT JSONB, '$.a' PASSING i +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



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)

2-floors house



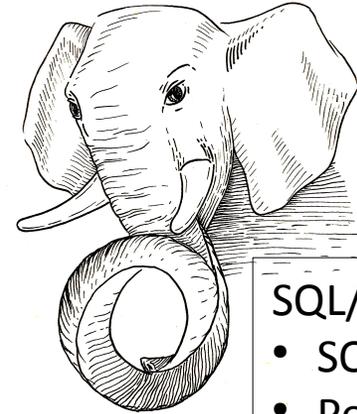
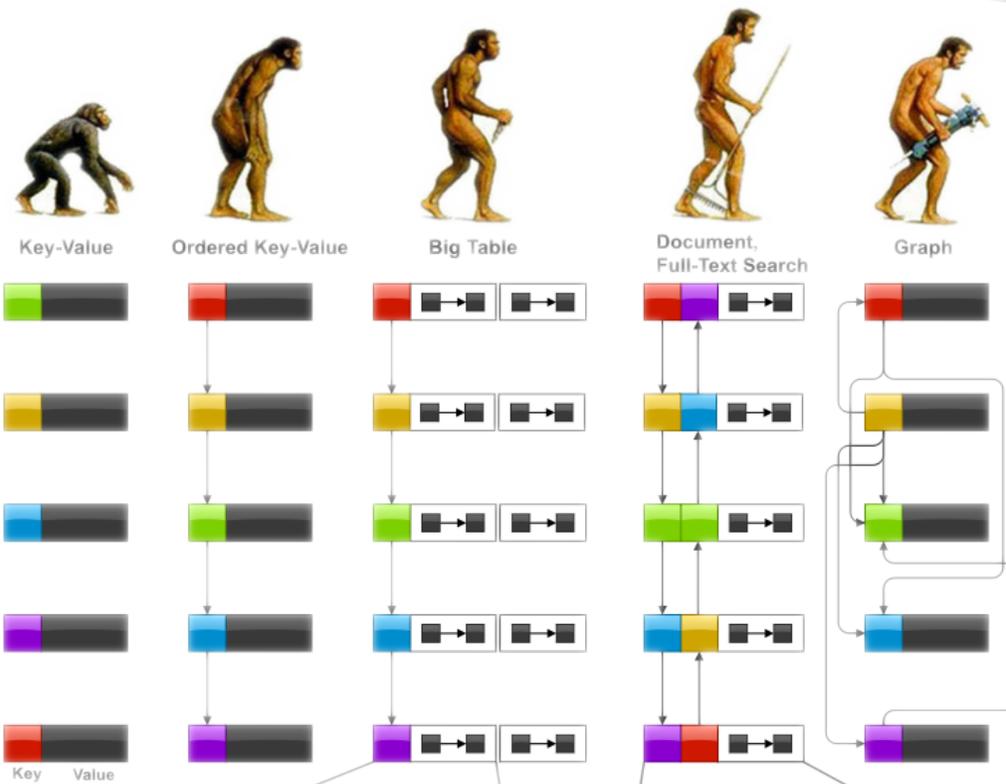
SQL/JSON examples: JSON_TABLE

Floors in relational form. **Returns parts !**

```
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)



SQL/JSON - 2019

- SQL-2016 standard
- Postgres Pro - 2018

JSONB - 2014

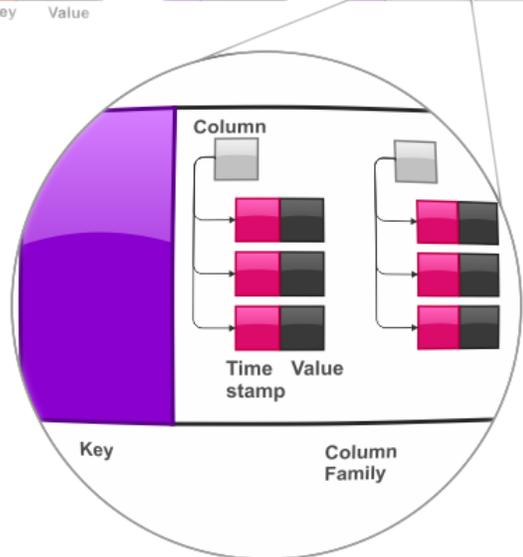
- Binary storage
- Nesting objects & arrays
- Indexing

JSON - 2012

- Textual storage
- JSON verification

HSTORE - 2003

- Perl-like hash storage
- No nesting
- Indexing



```

"employee" :
{
  "name" : "Mohana Pillai",
  "position" : "Delivery Manager",
  "projects" : [
    {
      "name" : "Easy Signatures"
    }
  ]
}, Semi-Structured Data
-----
Plain Text
is a confidential word or number
combination used as a code to
identify when accessing
an 8 and 15 characters
number and may not
spaces
  
```

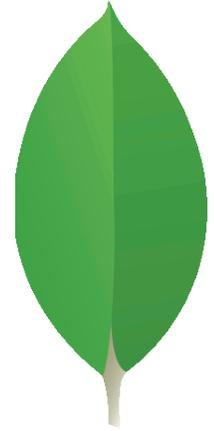
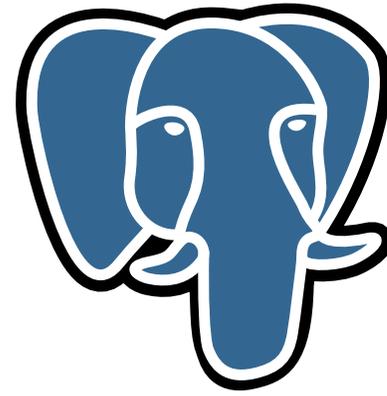


BENCHMARKS:

How NoSQL Postgres is fast



YCSB Benchmark

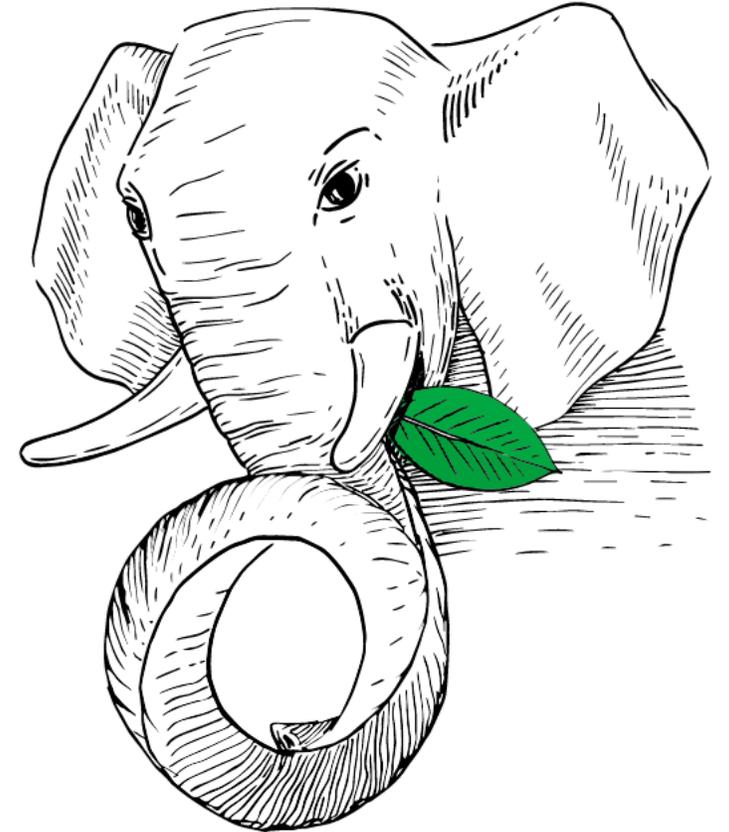


- Yahoo! Cloud Serving Benchmark -
<https://github.com/brianfrankcooper/YCSB/wiki>
- De-facto standard benchmark for NoSQL databases
- Scientific paper «Benchmarking Cloud Serving Systems with YCSB»
<https://www.cs.duke.edu/courses/fall13/cps296.4/838-CloudPapers/ycsb.pdf>
- We run YCBS for Postgres master, MongoDB 3.4.5
 - 1 server with 72 cores, 3 TB RAM, 2 TB SSD for clients
 - 1 server with 72 cores, 3 TB RAM, 2 TB SSD for database
 - 10Gbps switch



YCSB Benchmark

- For uniform queries
«Postgres optimized» outperforms MongoDB !
- For Zipfian queries on very big machines
for r/w workloads and clients > 250
MongoDB scales better.
- Persistent NoSQL is good case for Postgres.
max_connections up to 1000
- **SQL/JSON is fast as the JSONB**





SQL/JSON standard conformance

SQL/JSON feature	PostgreSQL 12	Oracle 18c	MySQL 8.0.4	MS SQL Server 2017
JSON_OBJECT: 7	6	5	1	0
JSON_ARRAY: 4	4	4	1	0
JSON_OBJECTAGG JSON_ARRAYAGG	2	2	2	0
RETURNING: 3	3	2	2	0
FORMAT JSON: 1	1	1	0	0
IS JSON: 2	2	1	0	0
JSON_EXISTS: 5	5	5	0	1
JSON_QUERY: 3	3	3	0	1



SQL/JSON standard conformance

SQL/JSON feature	PostgreSQL 12	Oracle 18c	MySQL 8.0.4	MS SQL Server 2017
JSON PATH: 15	15	11	5	2
TOTAL: 42	41/42	34/42	11/42	4/42

PostgreSQL 12 could have **the best implementation** of SQL/JSON standard !

SQL/JSON Standard Conformance in PostgreSQL, Oracle, MySQL, SQL Server
<http://obartunov.livejournal.com/200076.html>

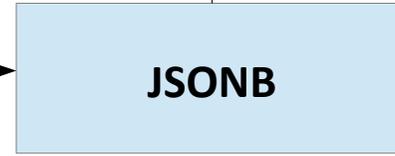


STANDARD

PERFORMANCE



2012



2014



2019



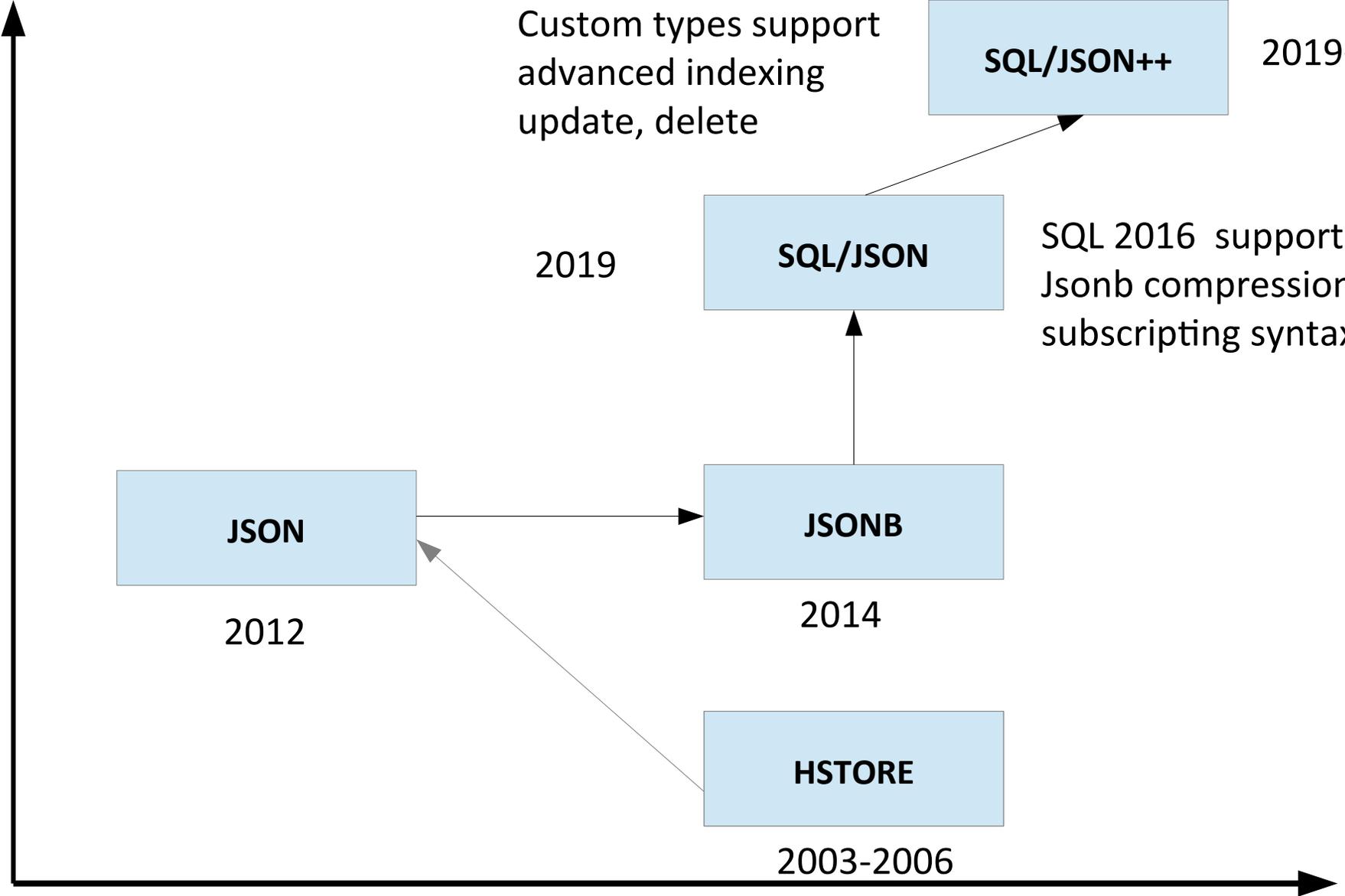
2019+

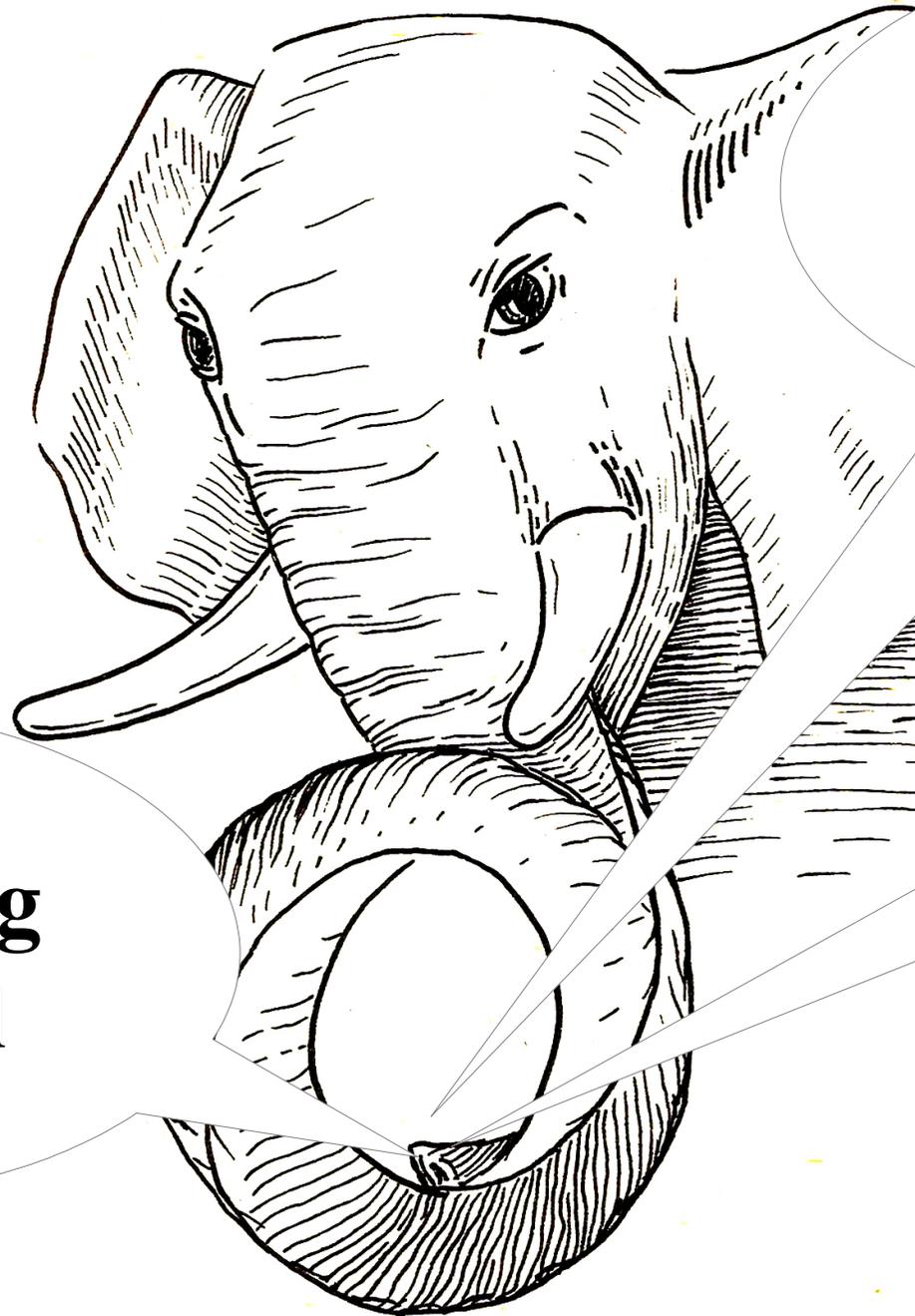


2003-2006

Custom types support
advanced indexing
update, delete

SQL 2016 support
Jsonb compression
subscripting syntax





**Hey, syntax for
UPDATE JSONB !**

ARRAYS use []

**Subscripting
j[a][b] = val**



JSONB subscripting syntax

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

```
UPDATE test_table set ARR[1] = 100;
```

```
SELECT JS['a']['a1']['a2'] FROM test_table;
```

```
UPDATE test_table SET JS['a']['b'] = '2'::jsonb;
```

Opclass parameters: 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)  
);
```



Opclass parameters: jsonb

- `projection` parameter for GIN opclasses for `jsonb` specifies which fragments of JSON document to index
- `projection` is a string in standard SQL/JSON path format
- **Selective** index is smaller, build time is shorter, supports more flexible queries
- Example of **selective** index:

```
CREATE INDEX bookmarks_selective_path_idx ON bookmarks USING  
gin(js jsonb_path_ops (projection='strict $.tags[*].term'));
```



JSONB COMPRESSION

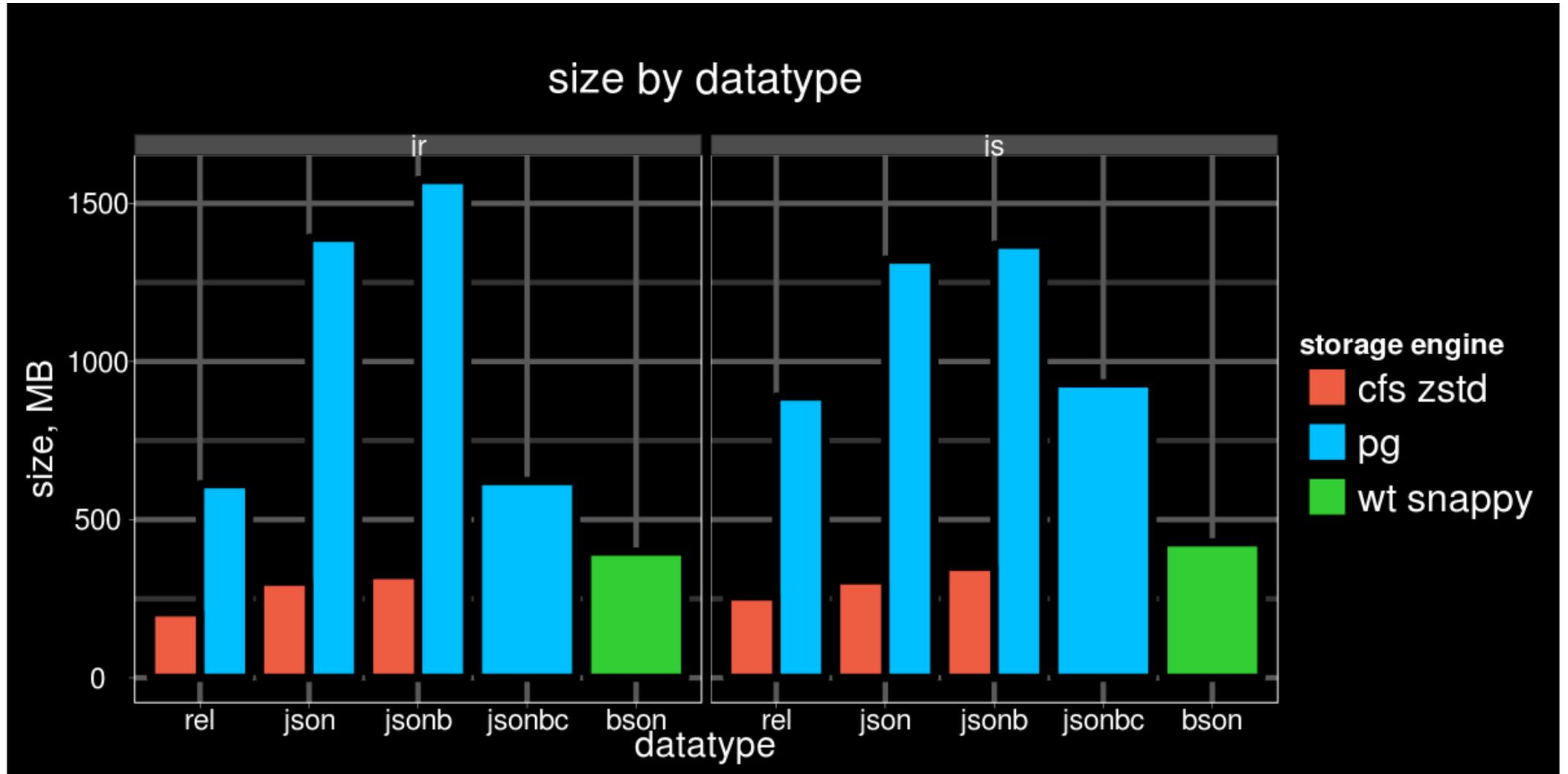
- JSONB is a «fat» data type — keys could be up to 2^{28} , 256 Mb !

«loooooooooooooooooooooooooooooooooong_key1»:1,

«veeeeeeeeery_loooooooooooooooooooooooooooooooooong_key2»:2

- Dictionary compression using
CUSTOM Compression API (PG 12)

jsonb compression: table size





jsonb compression: summary

- jsonbc can reduce jsonb column size to its relational equivalent size
- jsonbc has a very low CPU overhead over jsonb and sometimes can be even faster than jsonb
- jsonbc compression ratio is significantly lower than in-page level compression
- Availability:
 - Under review for PG 12



Summary

- Postgres is already a good NoSQL database with clear roadmap
- SQL/JSON provides better flexibility and interoperability
- SQL/JSON implementation in PostgreSQL is currently the best
- Move from NoSQL to Postgres !



SQL/JSON availability

- Github Postgres Professional repository
<https://github.com/postgrespro/sqljson>
- WEB-interface to play with SQL/JSON
<http://sqlfiddle.postgrespro.ru/#!21/0/1819>
- **Technical Report (SQL/JSON)** - available for free
http://standards.iso.org/ittf/PubliclyAvailableStandards/c067367_ISO_IEC_TR_19075-6_2017.zip
- We need your feedback, bug reports and suggestions



感謝諸位的時間

感謝大家!

THANKS FOR YOUR TIME

ALL

YOU

NEED
POSTERS

IS

