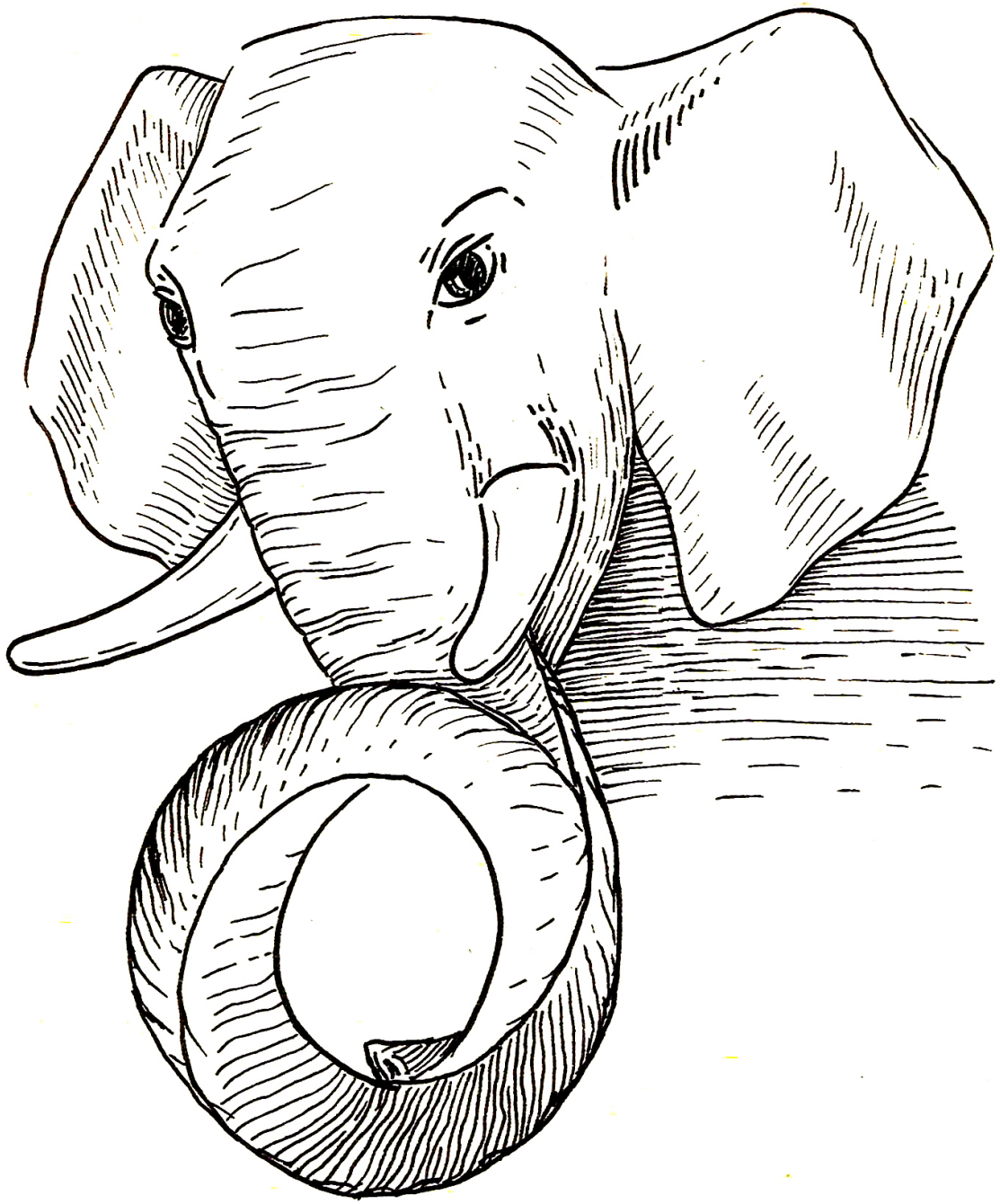


Postgres
для Хипстеров

Oleg Bartunov
Postgres Professional
Moscow University

#UfaDevConf, Ufa, October 14, 2017

Правильный выбор СУБД — важно !

- Архитектура системы должна быть расширяемой
- Расширяемость компонентов системы
- Костыли — средство расширяемости
- Костыль — средство добавления недостающей функциональности или исправления серьёзных дыр без должного редизайна системы
- Систем без «костылей» не бывает
- Костыли имеют тенденцию накапливаться

Система все-таки работает …

Поменять СУБД очень тяжело !

- Frontend-cache-backend-application-cache-файловое хранилище-база данных
- Смена СУБД — мучительный процесс, никакой автомагии
- Выбор СУБД на стадии разработки — важнейшая тема ! 99.99% проектов не требует разработки своей СУБД.

Выбор СУБД

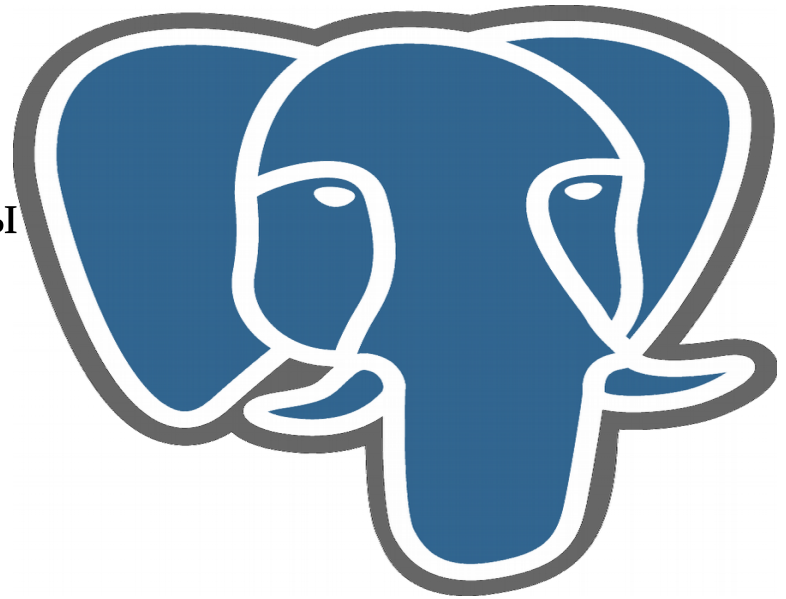
- Архитектор — функциональность, производительность, доступность
- Хакер-дба — удобство, привычка, религия
- Бизнес-маркетинг — сторонние факторы

Выбор СУБД

- **Расширяемость** СУБД — важнейший фактор вашего развития и независимости !
- Расширяемость:
  - Новая функциональность (типы данных, операции)
  - Своими руками
  - Без остановки сервера
  - Без ущерба производительности и надежности

Что такое PostgreSQL

PostgreSQL - это свободно распространяемая
объектно-реляционная СУБД (ORDBMS)



Расширяемая — типы данных, операторы, функции, индексы

Поддержка ANSI SQL (1992, 1999, 2003, 2011),
NoSQL (key-value, JSON, JSONB)

Разрабатывается независимым мировым
сообществом с существенным российским участием

Произношение: post-gress-Q-L, post-gres, пост-грес, pgsql (пэ-жэ-эс-ку-эль)

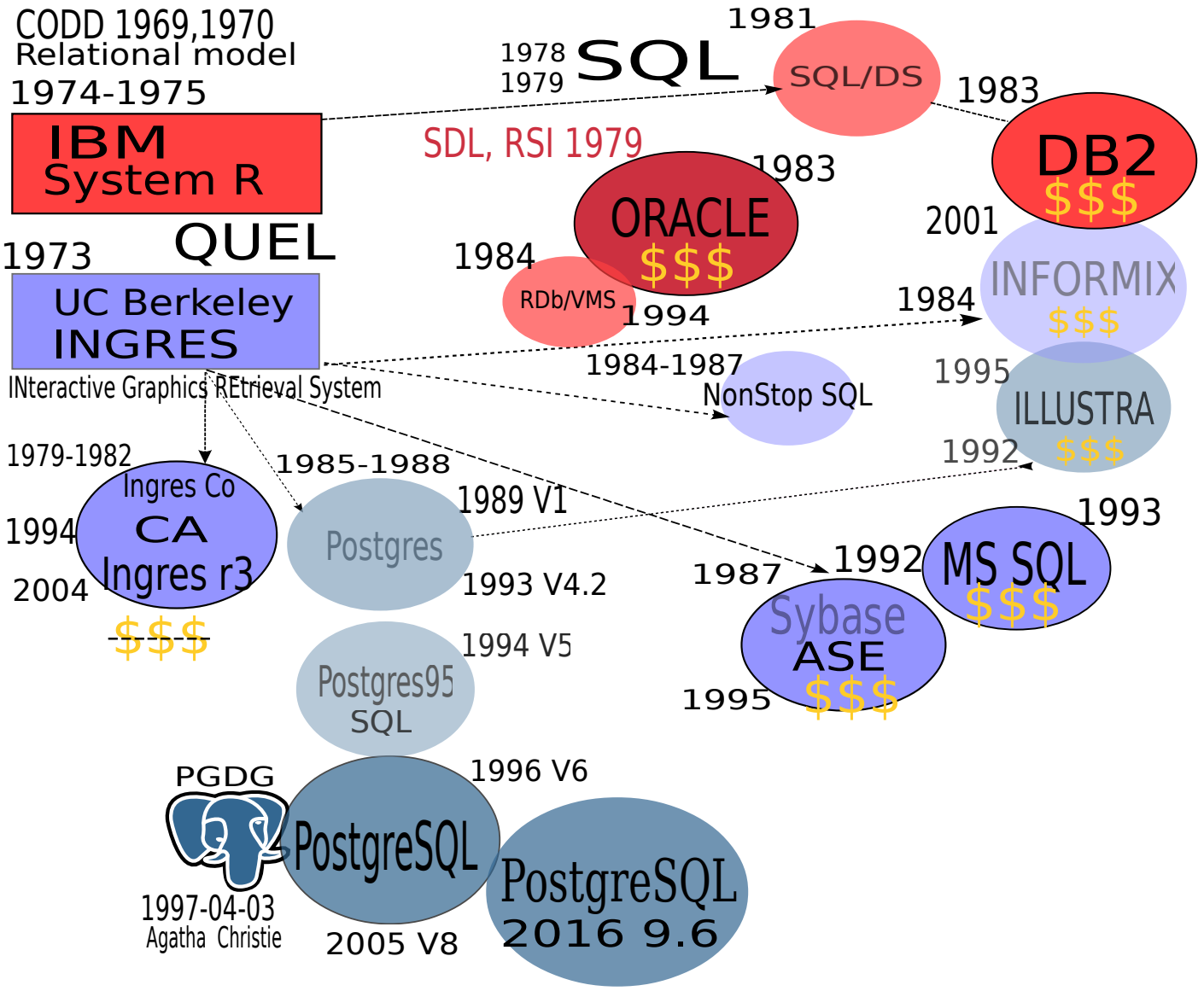
Web: http://www.postgresql.org, лицензия: BSD, MIT - like

Российский вендор - компания Postgres Professional

PROFESSIONAL Postgres

PostgreSQL History

Michael Stonebreaker
Turing Award, 2015



2017 10

Original design of Postgres

The main design goals of the new system are to:

1) provide better support for complex objects,

2) provide user extendibility for data types, operators and access methods,

3) provide facilities for active databases (i.e., alerters and triggers) and inferencing including forward- and backward-chaining,

4) simplify the DBMS code for crash recovery,

5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips, and

6) make as few changes as possible (preferably none) to the relational model. *

** Stonebraker M., Rowe L. A. The design of Postgres. – ACM, 1986. – T. 15. – №. 2. – C. 340-355.*

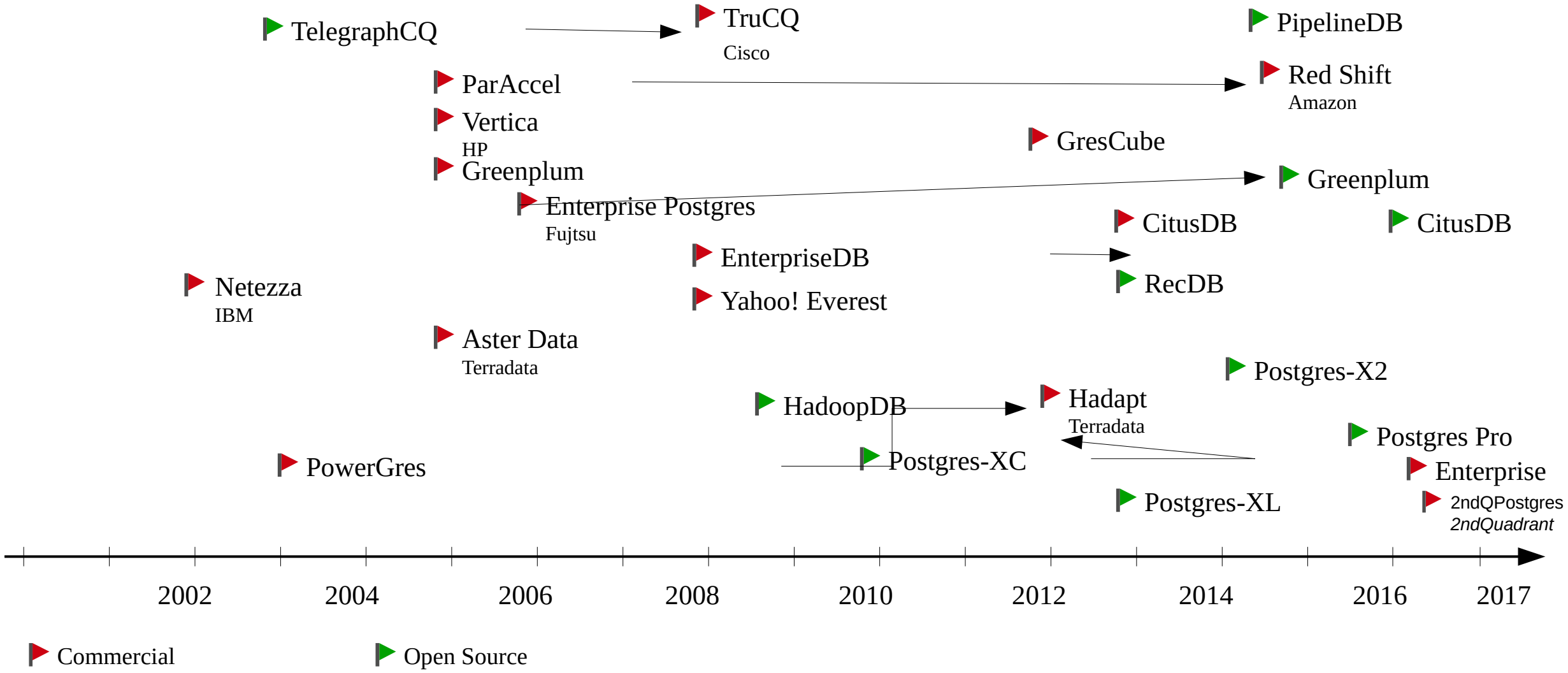
Extendability of PostgreSQL

"It is imperative that a user be able to construct new access methods to provide efficient access to instances of nontraditional base types"

Michael Stonebraker, Jeff Anton, Michael Hirohama.

Extendability in POSTGRES , IEEE Data Eng. Bull. 10 (2) pp.16-23, 1987

PostgreSQL Forks (we love forks!)



https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases

PostgreSQL is #4 !

334 systems in ranking, September 2017

Rank Sep 2017	Rank Aug 2017	Rank Sep 2016	DBMS	Database Model	Score Sep 2017	Score Aug 2017	Score Sep 2016
1.	1.	1.	Oracle	Relational DBMS	1359.09	-8.78	-66.47
2.	2.	2.	MySQL	Relational DBMS	1312.61	-27.69	-41.41
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1212.54	-12.93	+0.99
4.	4.	4.	PostgreSQL	Relational DBMS	372.36	+2.60	+56.01
5.	5.	5.	MongoDB	Document store	332.73	+2.24	+16.74
6.	6.	6.	DB2	Relational DBMS	198.34	+0.87	+17.15
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	128.81	+1.78	+5.50
8.	8.	↓ 7.	Cassandra	Wide column store	126.20	-0.52	-4.29
9.	9.	↑ 10.	Redis	Key-value store	120.41	-1.49	+12.61
10.	10.	↑ 11.	Elasticsearch	Search engine	120.00	+2.35	+23.52

https://db-engines.com/en/ranking

Важнейшие свойства PostgreSQL

Надежность и устойчивость PostgreSQL

Надежность PostgreSQL является известным и доказанным фактом на примере многих проектов, в которых PostgreSQL работает без единого сбоя и при больших нагрузках на протяжении нескольких лет.

Кроссплатформенность

PostgreSQL поддерживает все виды Unix, включая Linux, FreeBSD, Solaris, HPUX, Mac OS X, а также MS Windows.

Конкурентная работа при большой нагрузке

PostgreSQL использует многоверсионность (MVCC) для обеспечения надежной и быстрой работы в конкурентных условиях под большой нагрузкой.

Масштабируемость

PostgreSQL отлично использует современную архитектуру многоядерных процессоров - его производительность растет линейно до 64-х ядер. Кластерные решения на базе Postgres XL обеспечивают горизонтальную масштабируемость.

Расширяемость

Расширяемость PostgreSQL позволяет добавлять новую функциональность, в том числе и новые типы данных, без остановки сервера и своими силами.

Доступность

PostgreSQL распространяется под лицензией BSD, которая не накладывает никаких ограничений на коммерческое использование и не требует лицензионных выплат. Вы можете даже продавать PostgreSQL под своим именем !

Независимость

PostgreSQL не принадлежит ни одной компании, он развивается международным сообществом, в том числе и российскими разработчиками. Независимость PostgreSQL означает независимость вашего бизнеса от вендора и сохранность инвестиций.

Превосходная поддержка

Сообщество PostgreSQL предоставляет квалифицированную и быструю помощь. Коммерческие компании предлагают свои услуги по всему миру.

Postgres PROFESSIONAL

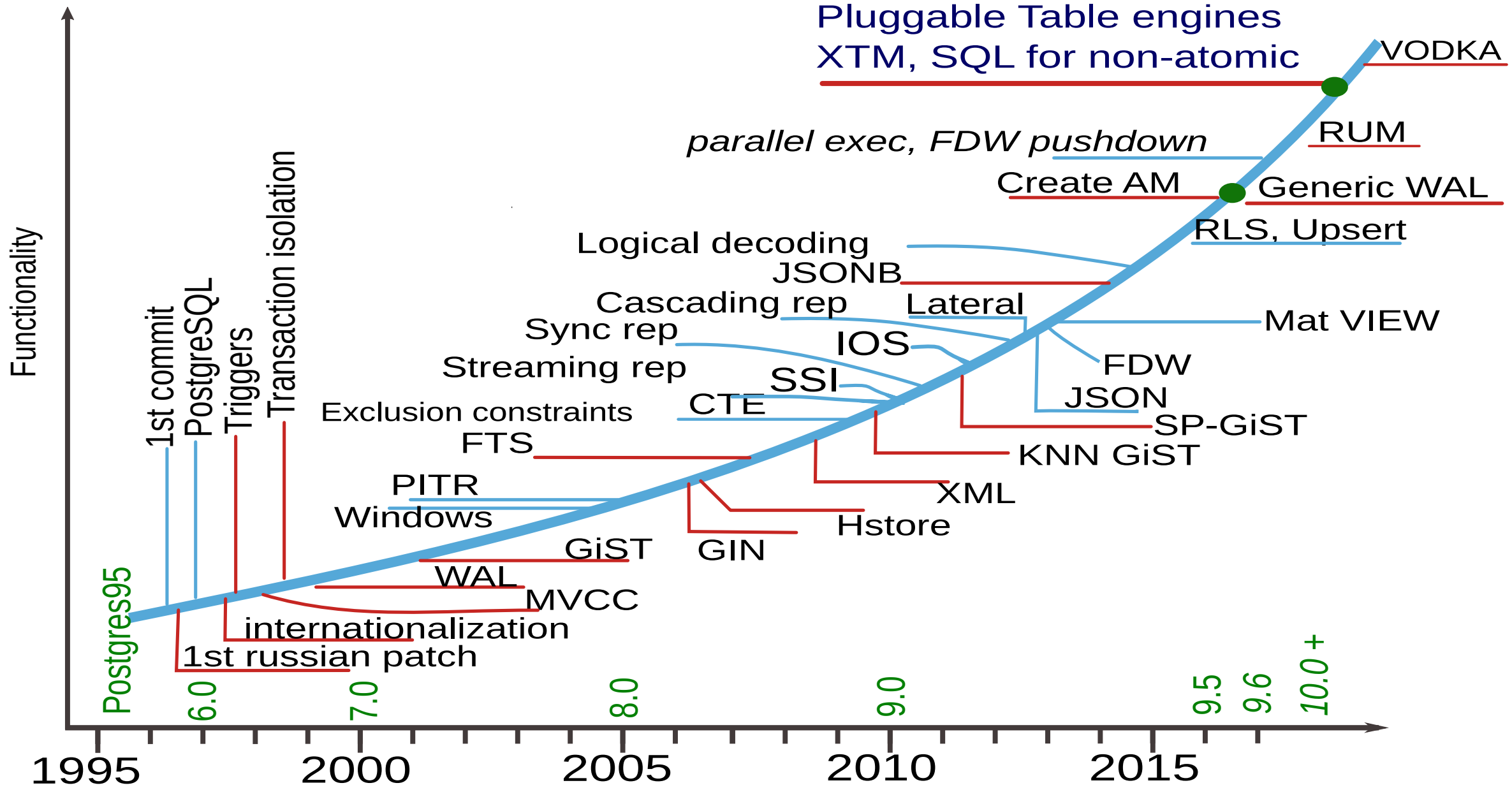
PostgreSQL users

+BIG RUSSIAN Enterprise !

Российское сообщество

- Самое организованное — несколько тысяч человек
- Митапы при поддержки крупных компаний
- Крупнейшие в мире конференции по постгресу:
  - летом PGDay.ru в Санкт-Петербурге (2014, 2015, 2016)
  - Зимой PGConf.ru в Москве (2015, 2016, 2017)
- Секции и квартирники на крупнейших конференциях
  - Highload++, RIT, Codefest, Stachka
- Участвуем в международных конференциях
  - PGConf.EU, PGCon.org
- Свободные курсы DBA1, DBA2, «Hacking Postgres» от Postgres Professional

20 years of PostgreSQL



Postgres Professional

Российский вендор PostgreSQL в России

- Поддержка, разработка, консалтинг, обучение
- Тему технологической независимости СУБД пропагандируем с 2011 г.
- Члены международного сообщества
- Участники и спонсоры международных конференций (Канада, Австрия, Бразилия)
- PgConf.Russia - крупнейшие в мире конференция по PostgreSQL

В направлениях, где мы ведем разработку, PostgreSQL является лидером* среди РСУБД

- геоинформационные системы, слабоструктурированные данные, полнотекстовый поиск, расширяемость

Все российские ключевые международно признанные разработчики PostgreSQL работают в нашей компании

В нашей команде 4 кандидата наук: 3 – по PostgreSQL и технологиям БД.

Сотрудничаем с МГУ и СПбГУ.

NoSQL (предпосылки)

- Relational DBMS - integrational
  - All APPs communicaties through RDBMS
  - SQL — universal language to work with data
  - All changes in RDBMS are available to all
  - Changes of the scheme are difficult, so → slow releases
  - Mostly for interactive work
    - Aggregates are mostly interested, not the data itself, SQL is needed
    - SQL takes cares about transactions, consistency … instead of human

The problem

- The world of data and applications is changing
- BIG DATA (**V**olume of data,**V**elocity of data in-out, **V**ariety of data)
- Web applications are service-oriented (SQL → HTTP)
  - No need for the monolithic database
  - Service itself can aggregate data and check consistency of data
  - High concurrency, simple queries
  - Simple database (key-value) is ok
  - Eventual consistency is ok, no ACID overhead (ACID → BASE)
- Application needs faster releases, «on-fly» schema change
- NoSQL databases match all of these — scalable, efficient, fault-tolerant, no rigid schema, ready to accept any data.

NoSQL databases (wikipedia) …+++

Document store

* Lotus Notes
* CouchDB
* MongoDB
* Apache Jackrabbit
* Colayer
* XML databases
  - o MarkLogic Server
  - o eXist

Graph

* Neo4j
* AllegroGraph

Tabular

* BigTable
* Mnesia
* Hbase
* Hypertable

Key/value store on disk

* Tuple space
* Memcachedb
* Redis
* SimpleDB
* flare
* Tokyo Cabinet
* BigTable

Key/value cache in RAM

* memcached
* Velocity
* Redis

Eventually-consistent key-value store

* Dynamo
* Cassandra
* Project Voldemort

Ordered key-value store

* NMDB
* Luxio
* Memcachedb
* Berkeley DB

Object database

* Db4o
* InterSystems Caché
* Objectivity/DB
* ZODB

The problem

- What if NoSQL functionality is not enough ?
- What if application needs ACID and flexibility of NoSQL ?

- Relational databases work with data with schema known in advance
- One of the major compaints to relational databases is rigid schema.
  It's not easy to change schema online (ALTER TABLE … ADD COLUMN...)
- Application should wait for schema changing, infrequent releases
- NoSQL uses json format, why not have it in relational database ?

Challenge to PostgreSQL !

- Full support of semi-stuctured data in PostgreSQL
  - Storage
  - Operators and functions
  - Efficiency (fast access to storage, indexes)
  - Integration with CORE (planner, optimiser)
- Actually, PostgreSQL is schema-less database since 2003 — hstore, one of the most popular extension !

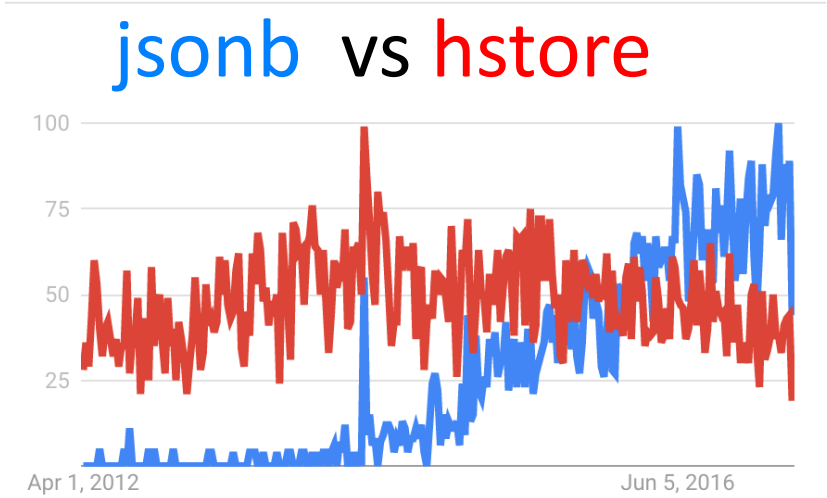
Introduction to Hstore

id	col1	col2	col3	col4	col5	Hstore key1=>val1, key2=>val2,.....

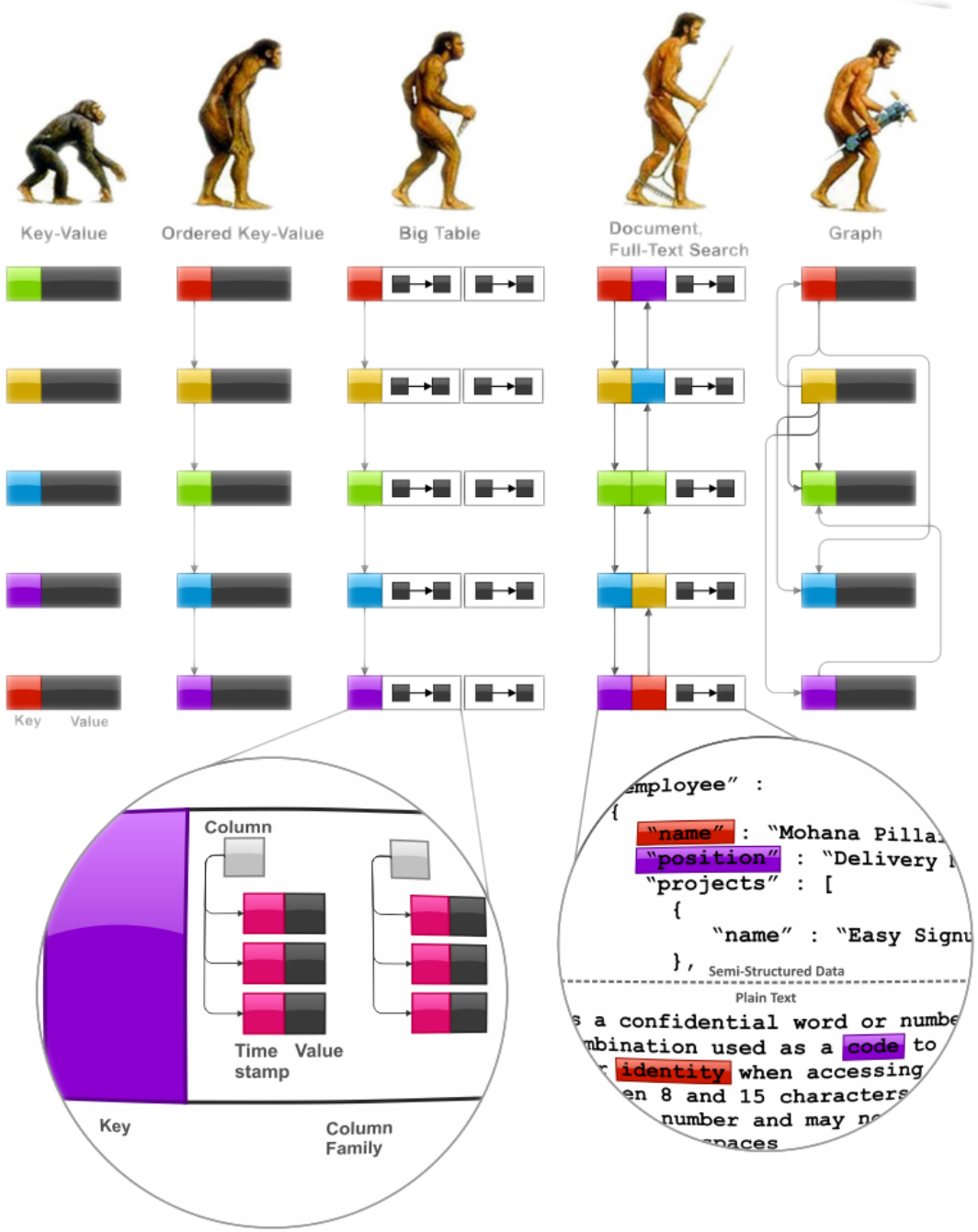
- Easy to add key=>value pair
- No need change schema, just change hstore.
- Schema-less PostgreSQL in 2003 !

NoSQL Postgres briefly

- 2003 — hstore (sparse columns, schema-less)
- 2006 — hstore as demo of GIN indexing, 8.2 release
- 2012 (sep) — JSON in 9.2 (verify and store)
- 2012 (dec) — nested hstore proposal
- 2013 — PGCon, Ottawa: nested hstore
- 2013 — PGCon.eu: binary storage for nested data
- 2013 (nov) — nested hstore & jsonb (better/binary)
- 2014 (feb-mar) — forget nested hstore for jsonb
- Mar 23, 2014 — jsonb committed for 9.4
- Autumn, 2018 — SQL/JSON for 10.X or 11 ?



PROFESSIONAL Postgres



PostgreSQL 9.4: NoSQL on ACID

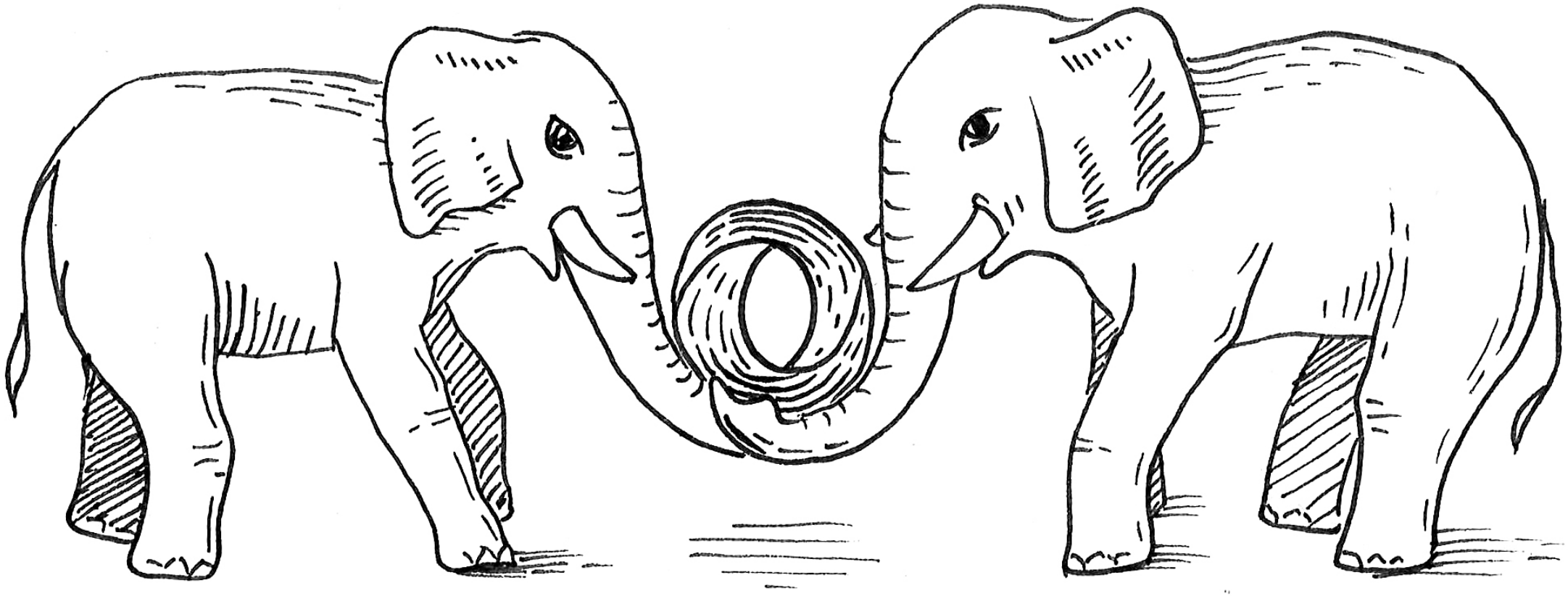
JSONB - 2014
- Binary storage
- Nesting objects & arrays
- Indexing

JSON - 2012
- Textual storage
- JSON verification

HSTORE - 2003
- Perl-like hash storage
- No nesting
- Indexing

PROFESSIONAL Postgres

Two JSON data types !!!

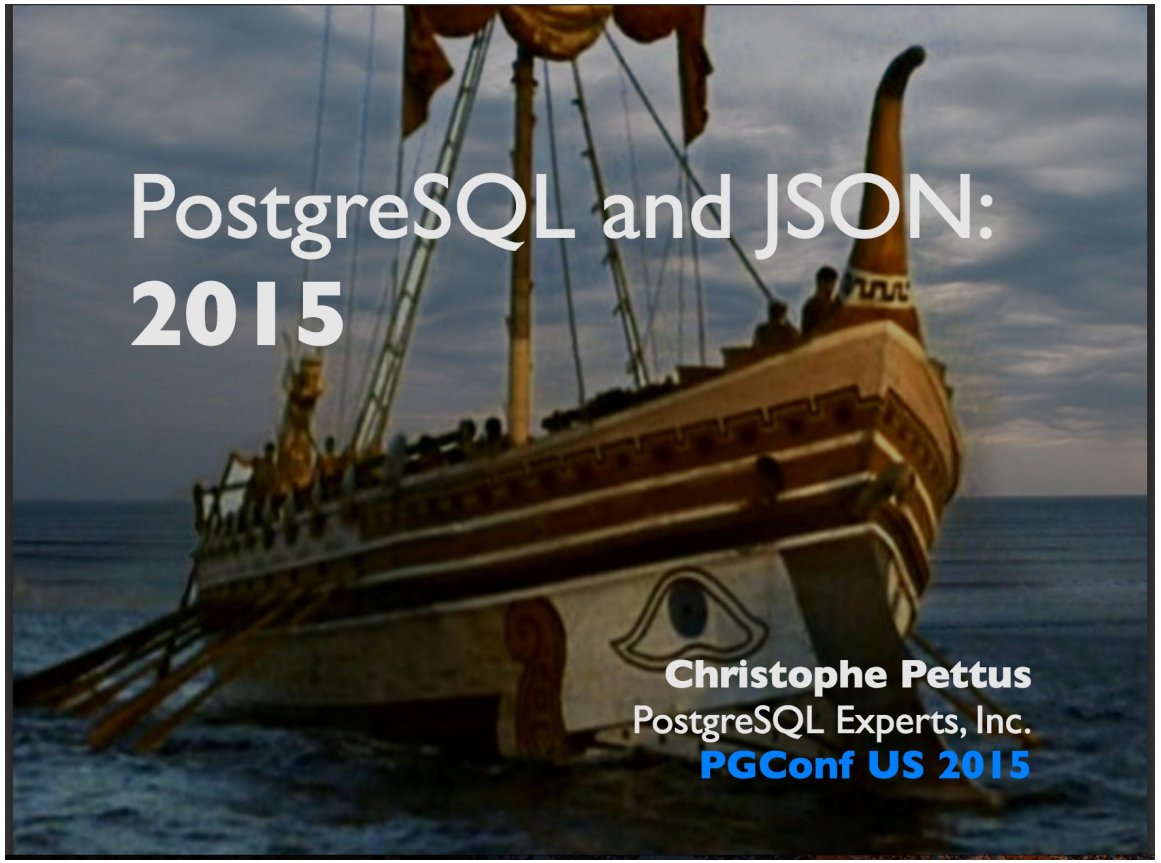


Jsonb vs Json

```
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}
(1 row)
```

- json: textual storage «as is»
- jsonb: no whitespaces
- jsonb: no duplicate keys, last key win
- jsonb: keys are sorted by (length, key)
- jsonb has a binary storage: no need to parse, has index support

Very detailed talk about JSON[B]



http://thebuild.com/presentations/json2015-pgconfus.pdf

Postgres PROFESSIONAL

JSONB is great, BUT there is No good query language — jsonb is a «black box» for SQL

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}], "colr": "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

JSON in SQL-2016

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

- It‘s not a new data type, it‘s a JSON data model for SQL
- PostgreSQL implementation is a subset of standard:
  - JSONB - ORDERED and UNIQUE KEYS
  - jsonpath data type for SQL/JSON path language
  - nine functions, implemented as SQL CLAUSEs

SQL/JSON in PostgreSQL

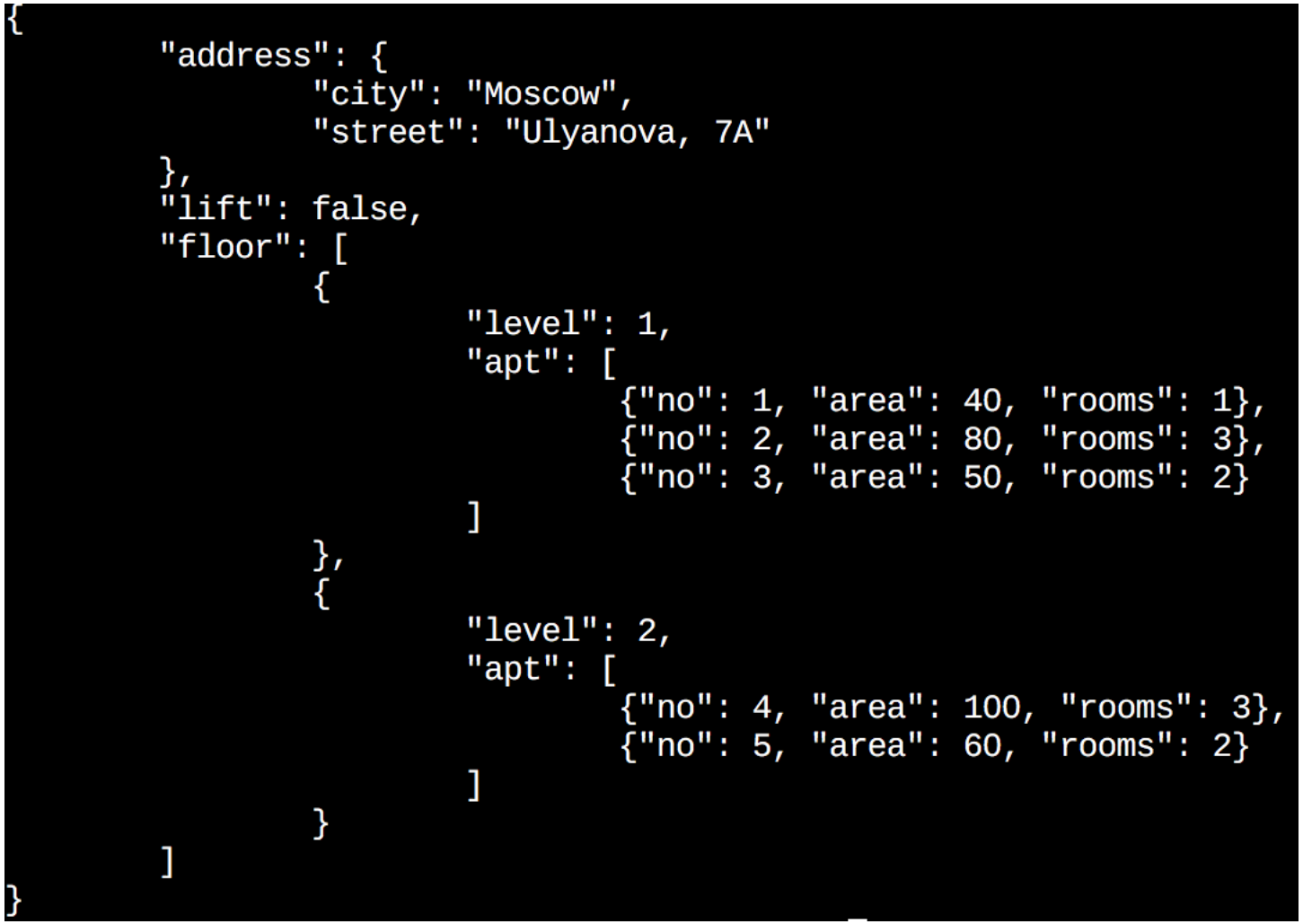
- **Jsonpath** provides an ability to operate (in standard specified way) with json structure at SQL-language level
  - Dot notation — $.a.b.c
  - Array - [*]
  - Filter ? - $.a.b.c ? (@.x > 10)
  - Methods - $.a.b.c.x.type()

```
SELECT * FROM js WHERE JSON_EXISTS(js, 'strict $.tags[*] ? (@.term ==
"NYC")');

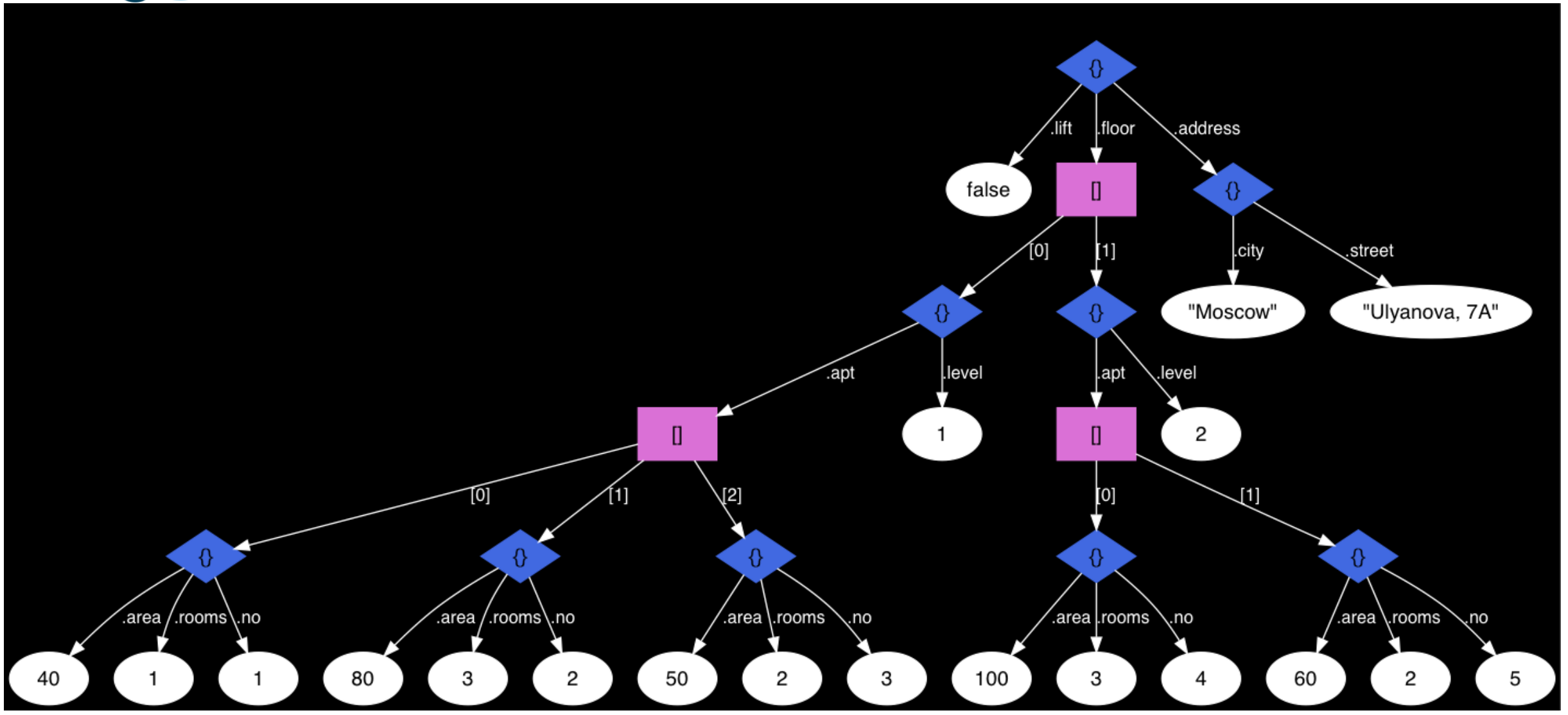
SELECT * FROM js WHERE js @> '{"tags": [{"term": "NYC"}]}';
```

PROFESSIONAL Postgres

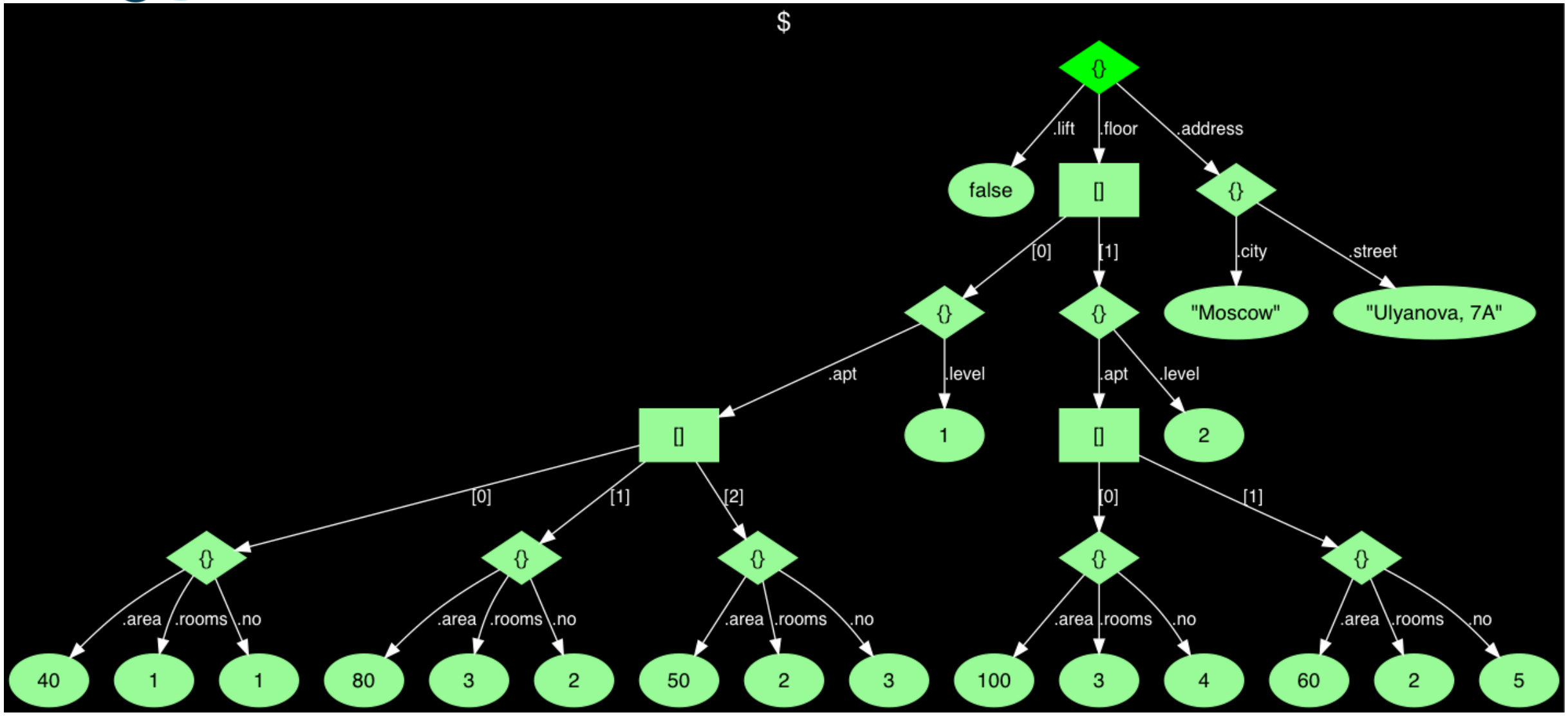
Visual guide on jsonpath



2-floors house

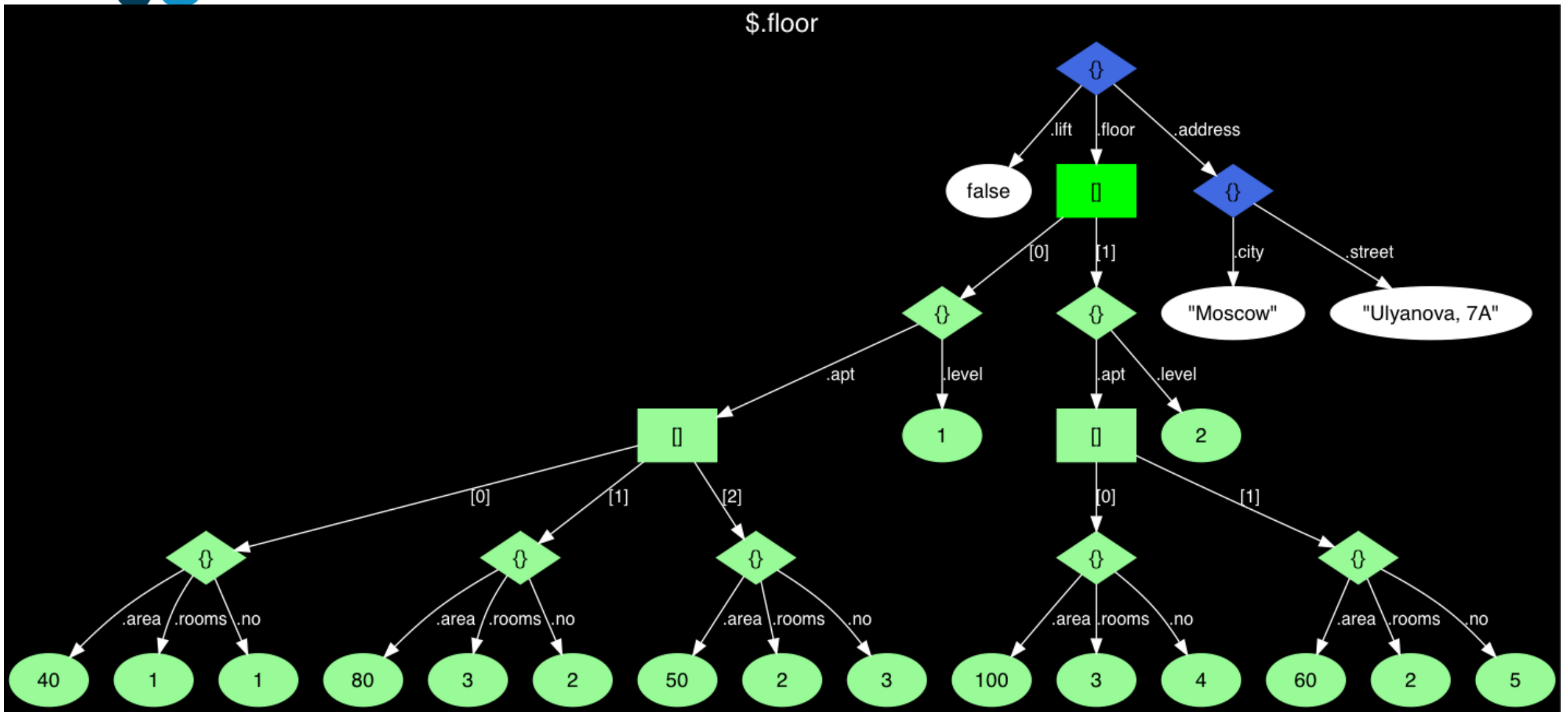


Everything

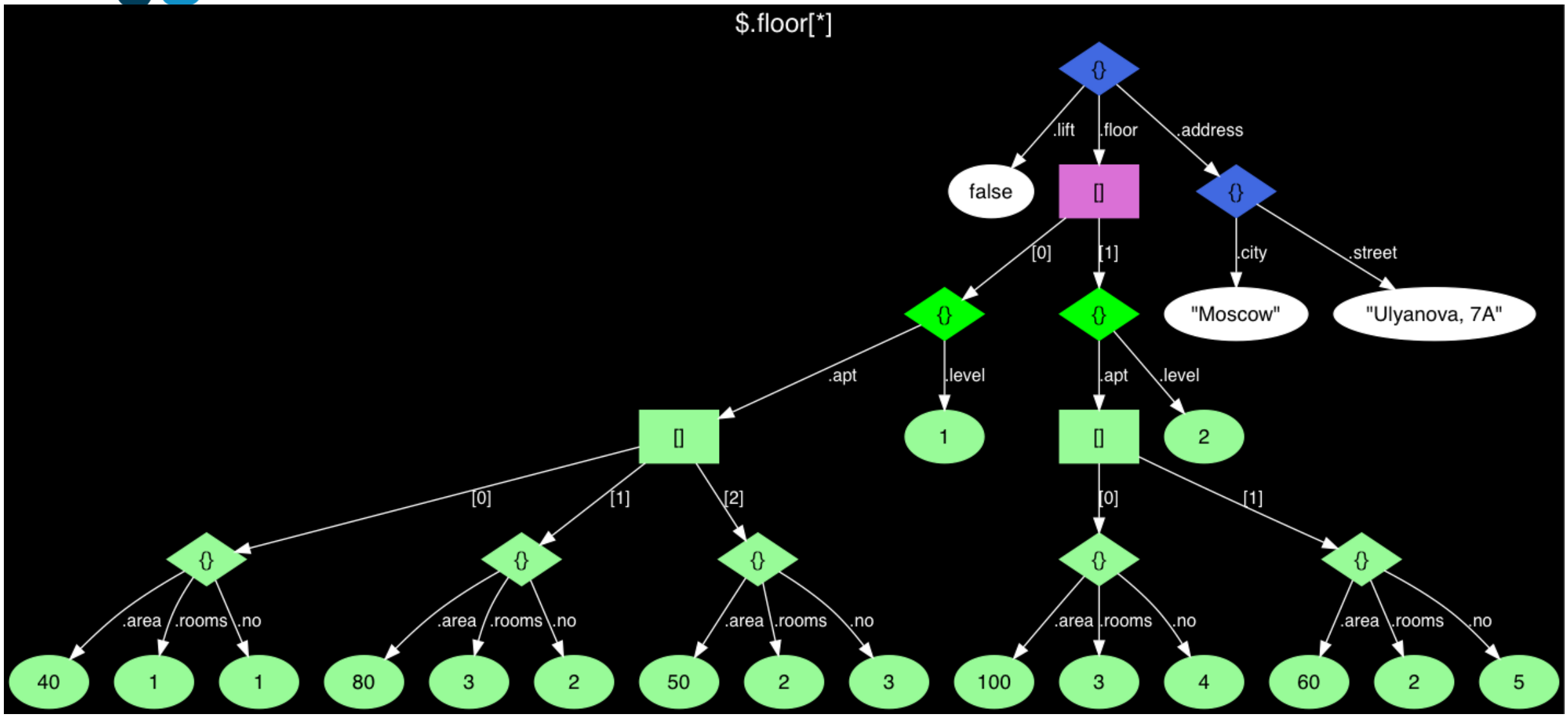

$
{}
.lift
.floor
.address
false
[]
{}
[0]
[1]
.city
.street
"Moscow"
"Ulyanova, 7A"
{}
{}
.apt
.level
.apt
.level
[]
1
[]
2
[0]
[1]
[2]
[0]
[1]
{}
{}
{}
{}
{}
.area
.rooms
.no
.area
.rooms
.no
.area
.rooms
.no
.area
.rooms
.no
.area
.rooms
.no
40
1
1
80
3
2
50
2
3
100
3
4
60
2
5

PROFESSIONAL Postgres

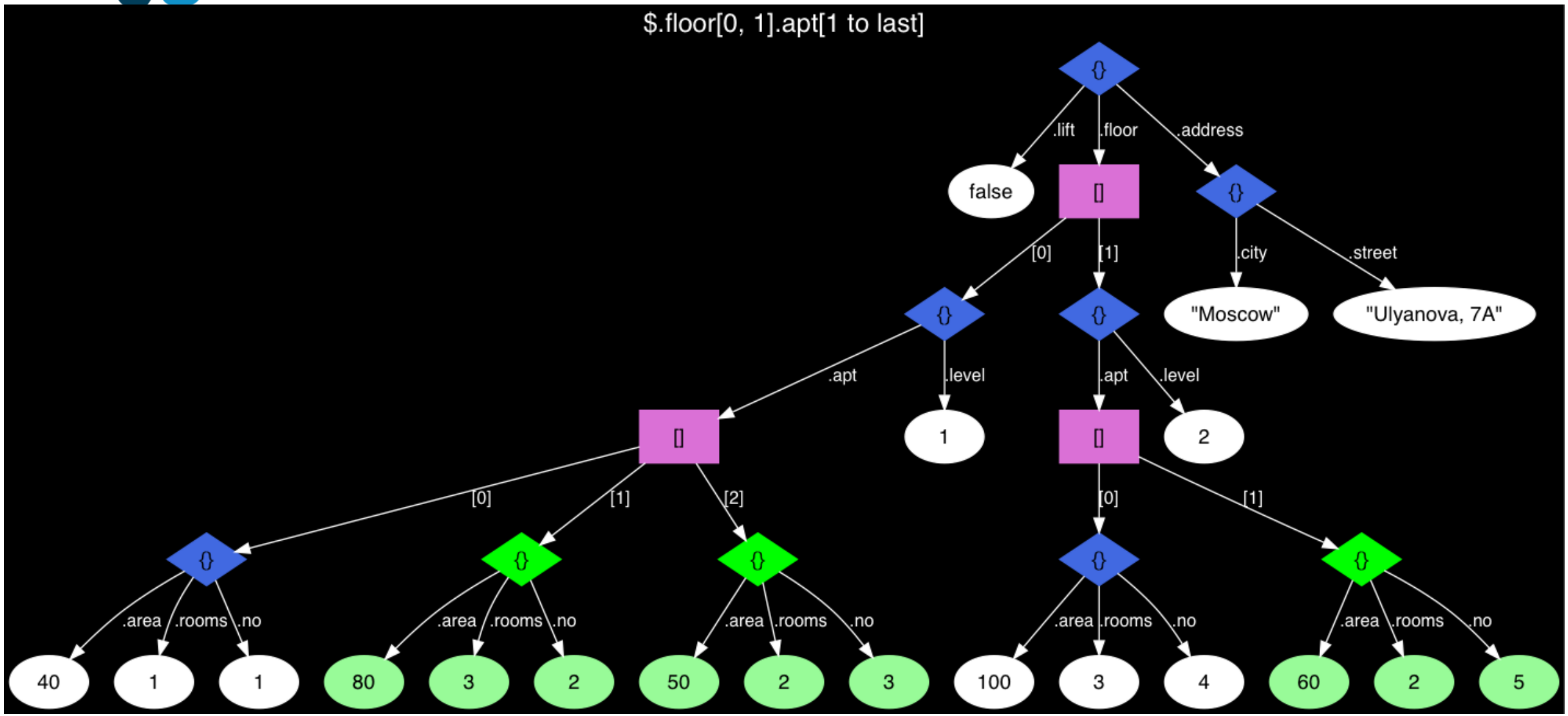
[Floors]



PROFESSIONAL Postgres



PROFESSIONAL Postgres



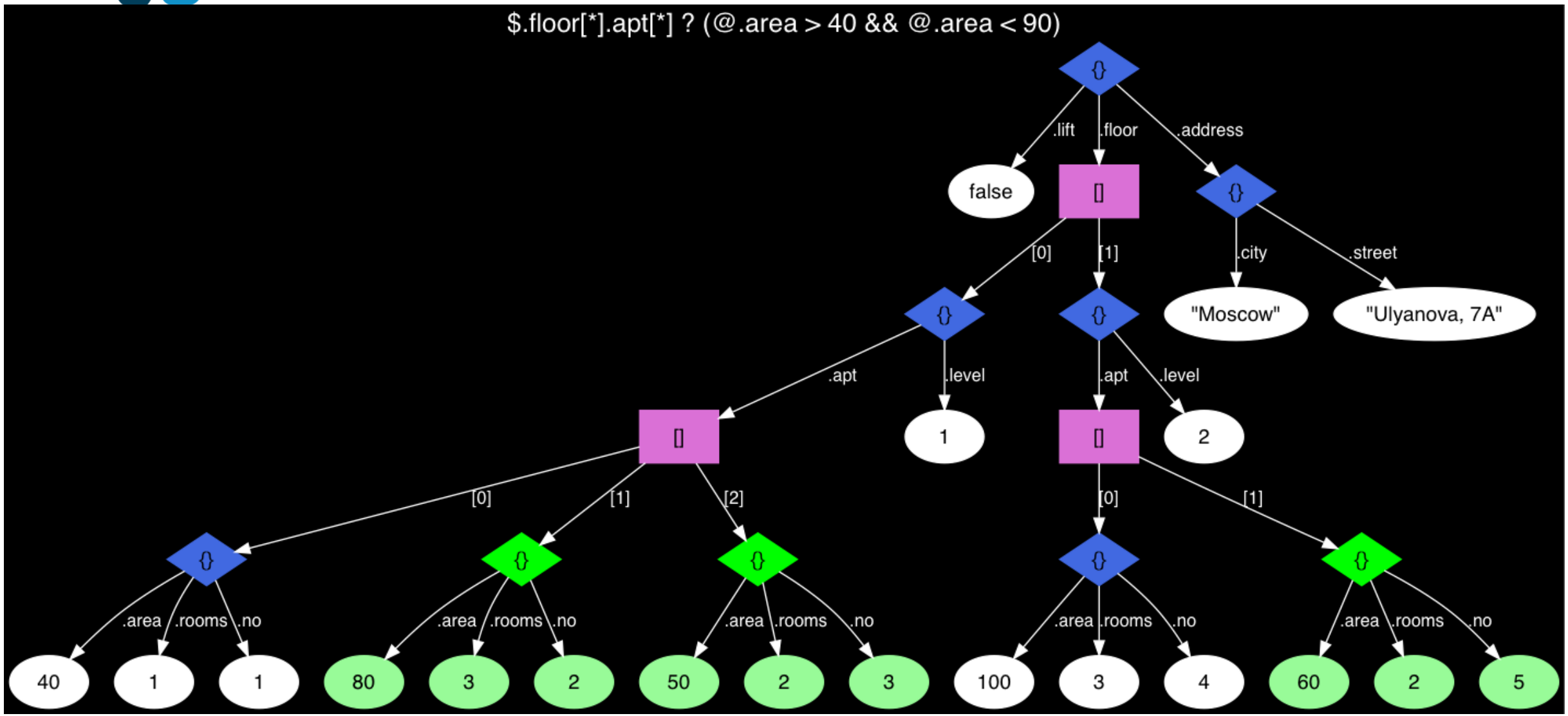
$.floor[0, 1].apt[1 to last]

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

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

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);
```

PROFESSIONAL Postgres



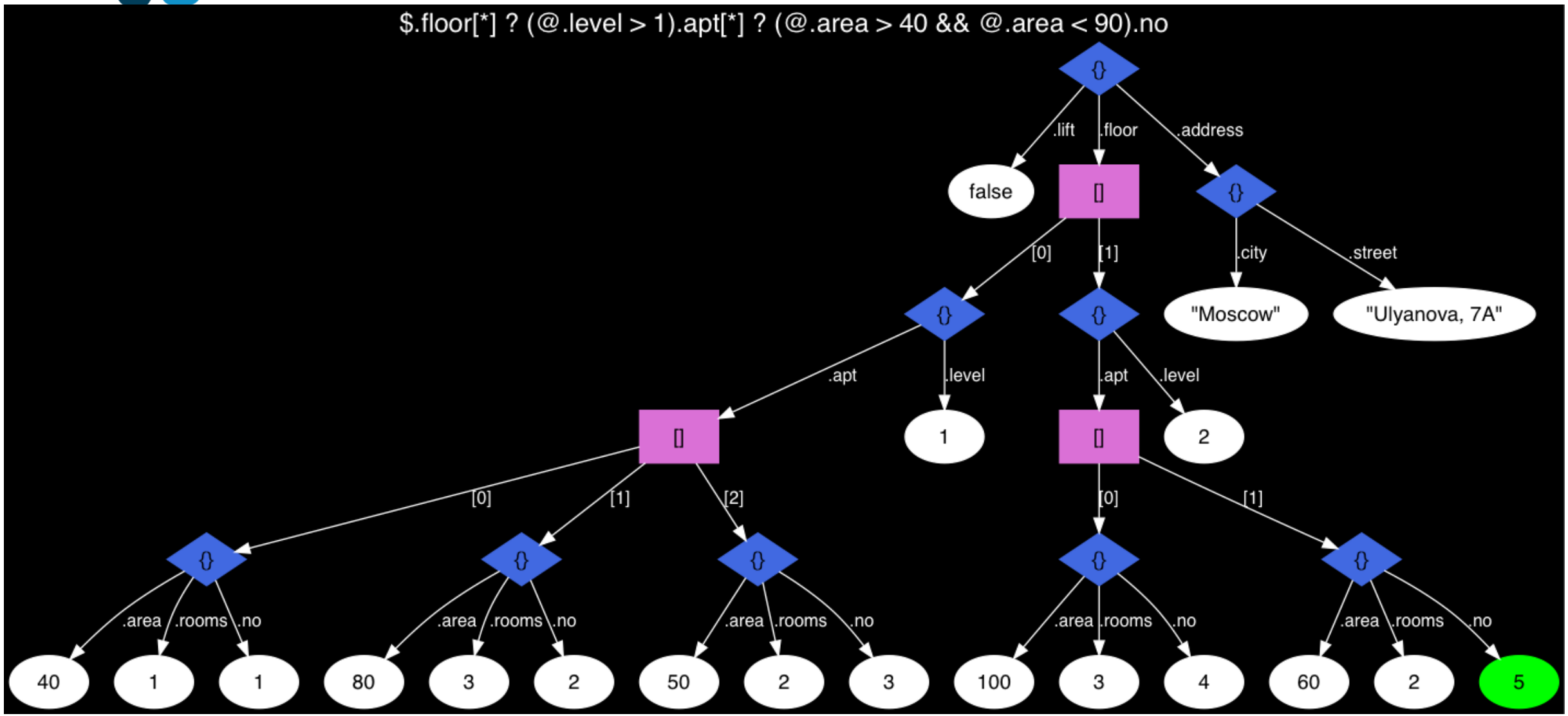
$.floor[*].apt[*] ? (@.area > 40 && @.area < 90)

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

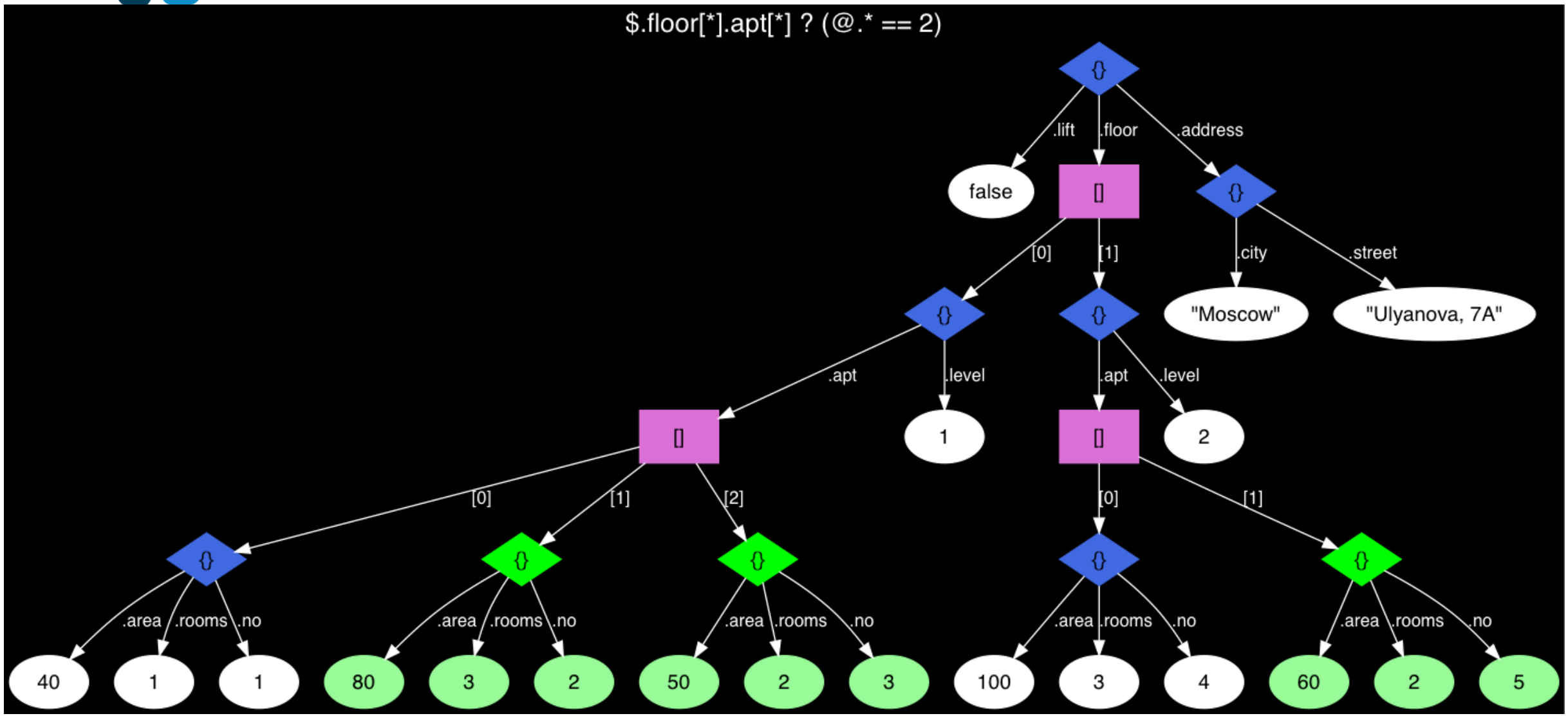
SELECT apt
FROM (SELECT jsonb_array_elements(jsonb_array_elements(js->'floor')->'apt')
      FROM house) apts(apt)
WHERE (apt->>'area')::int > 40 AND (apt->>'area')::int < 90;

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;
```

PROFESSIONAL Postgres



PROFESSIONAL Postgres

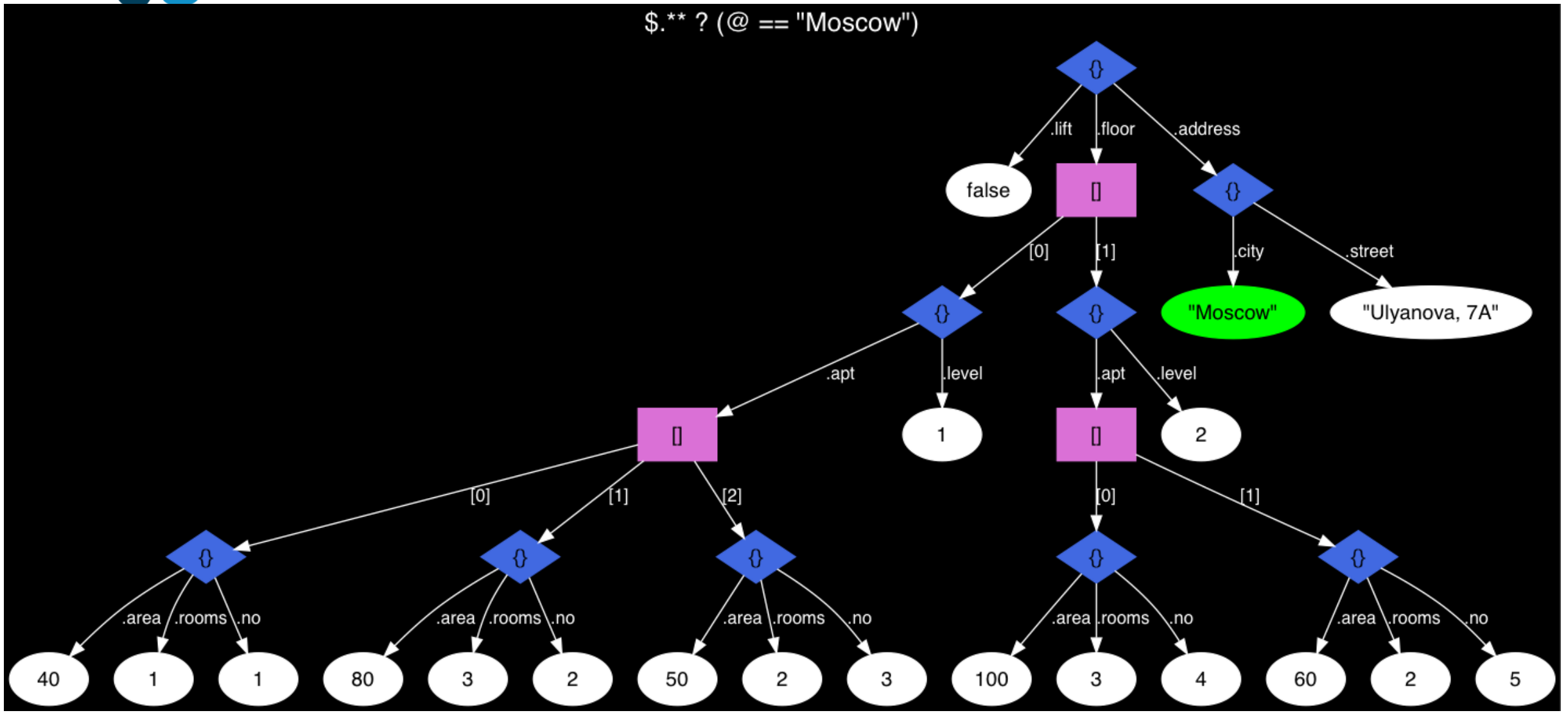


$.floor[*].apt[*] ? (@.* == 2)

```
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

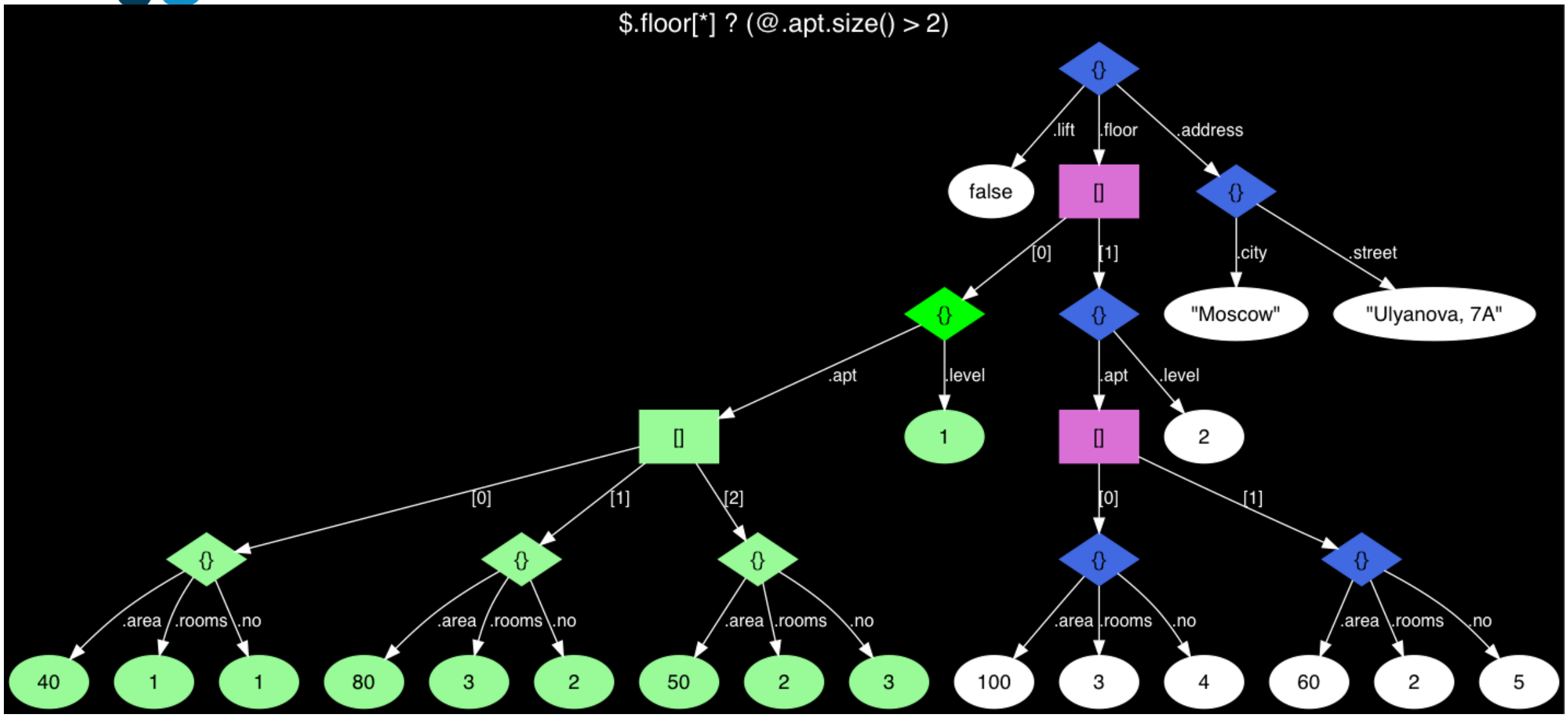


$.** ? (@ == "Moscow")

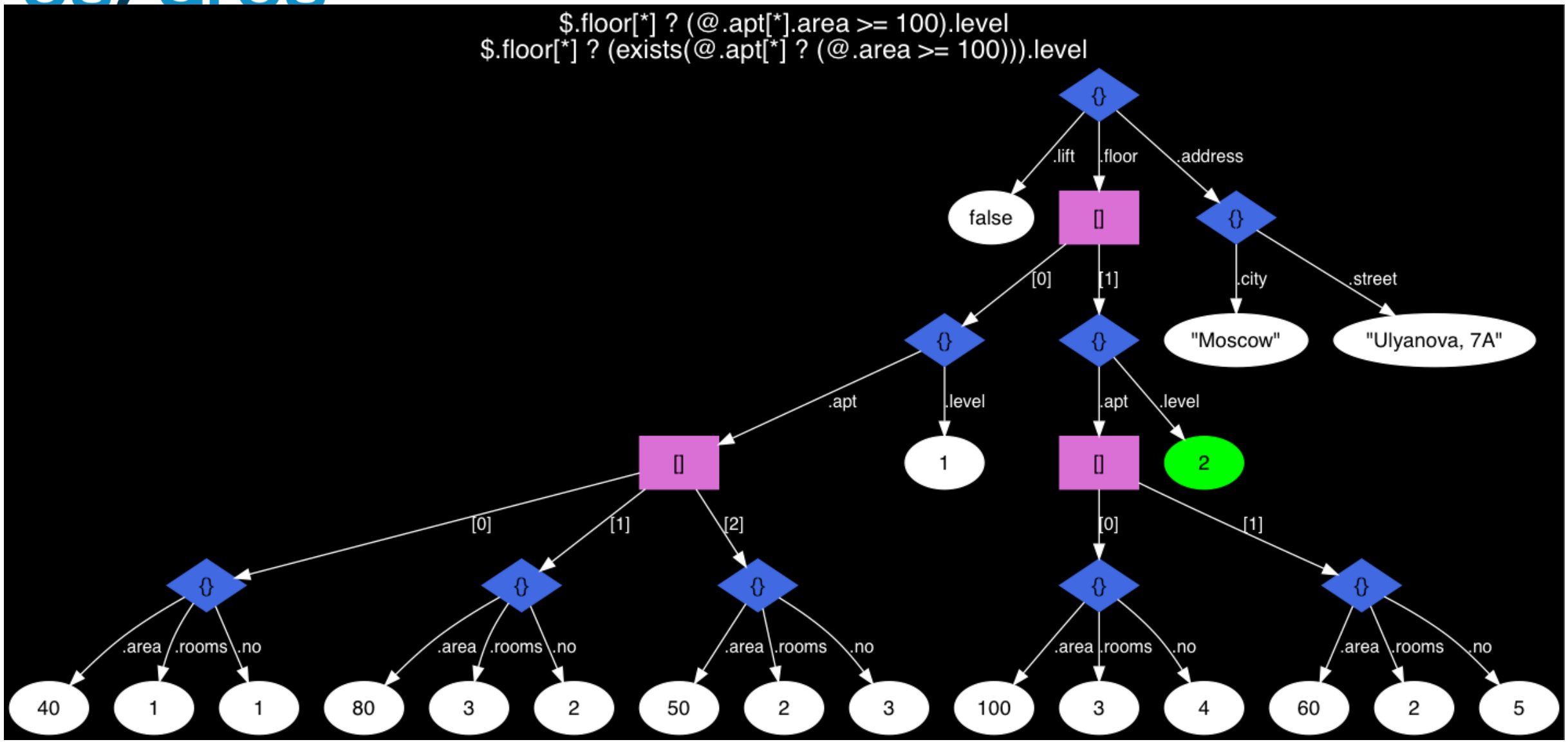
```
SELECT JSON_EXISTS(js, '$.** ? (@ == "Moscow")') FROM house;

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"');
```

PROFESSIONAL Postgres



PROFESSIONAL Postgres



SQL/JSON in PostgreSQL

```
SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
            PASSING 0 AS x, 2 AS y);
 ?column?
----------
 t
(1 row)

SELECT JSON_EXISTS(jsonb '{"a": 1, "b": 2}', '$.* ? (@ > $x && @ < $y)'
            PASSING 0 AS x, 1 AS y);
 ?column?
----------
 f
(1 row)
```

SQL/JSON in PostgreSQL

- The SQL/JSON **construction** functions:
  - JSON_OBJECT - serialization of an JSON object.
    - json[b]_build_object()
  - JSON_ARRAY - serialization of an JSON array.
    - json[b]_build_array()
  - JSON_ARRAYAGG - serialization of an JSON object from aggregation of SQL data
    - json[b]_agg()
  - JSON_OBJECTAGG - serialization of an JSON array from aggregation of SQL data
    - 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

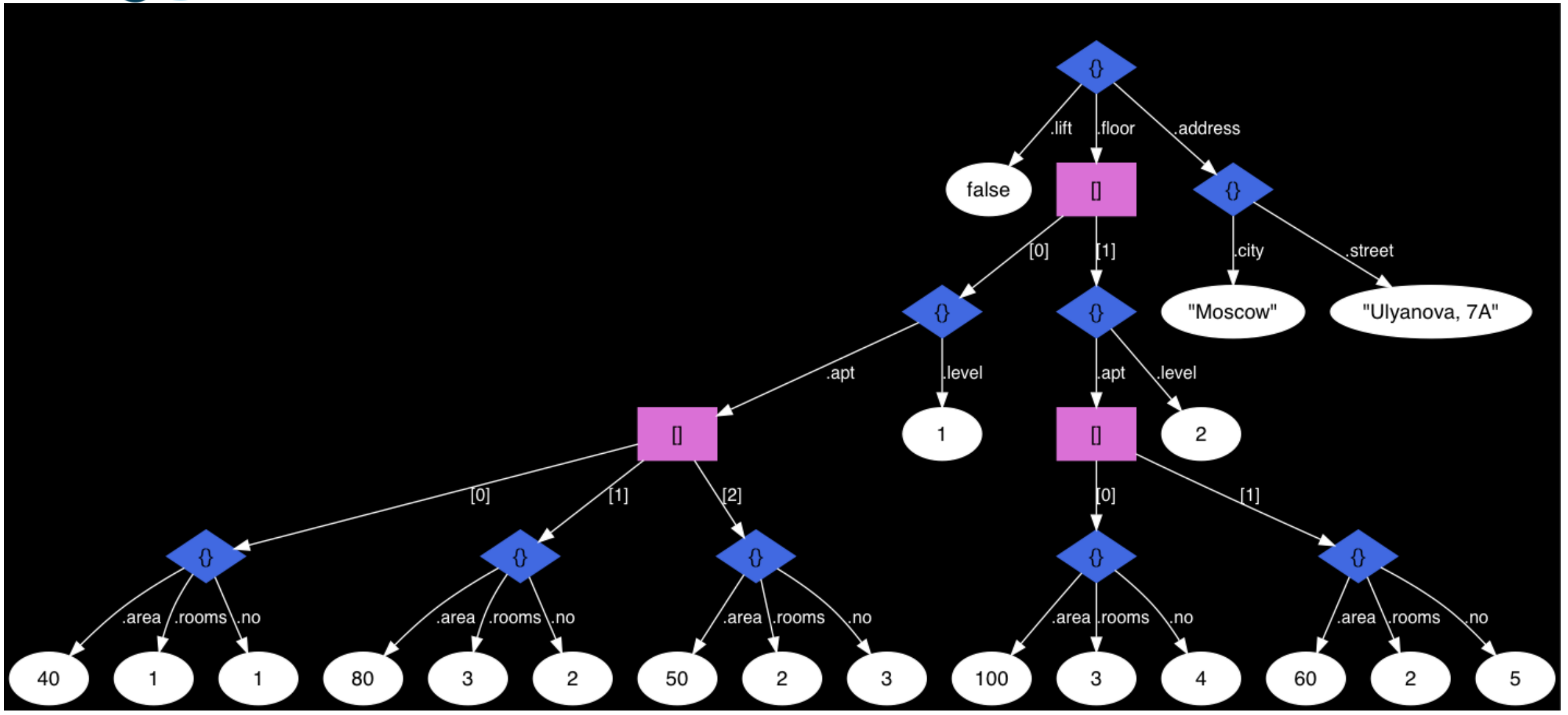
SQL/JSON examples: Constraints

```
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.

2-floors house



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)
```

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"';
```

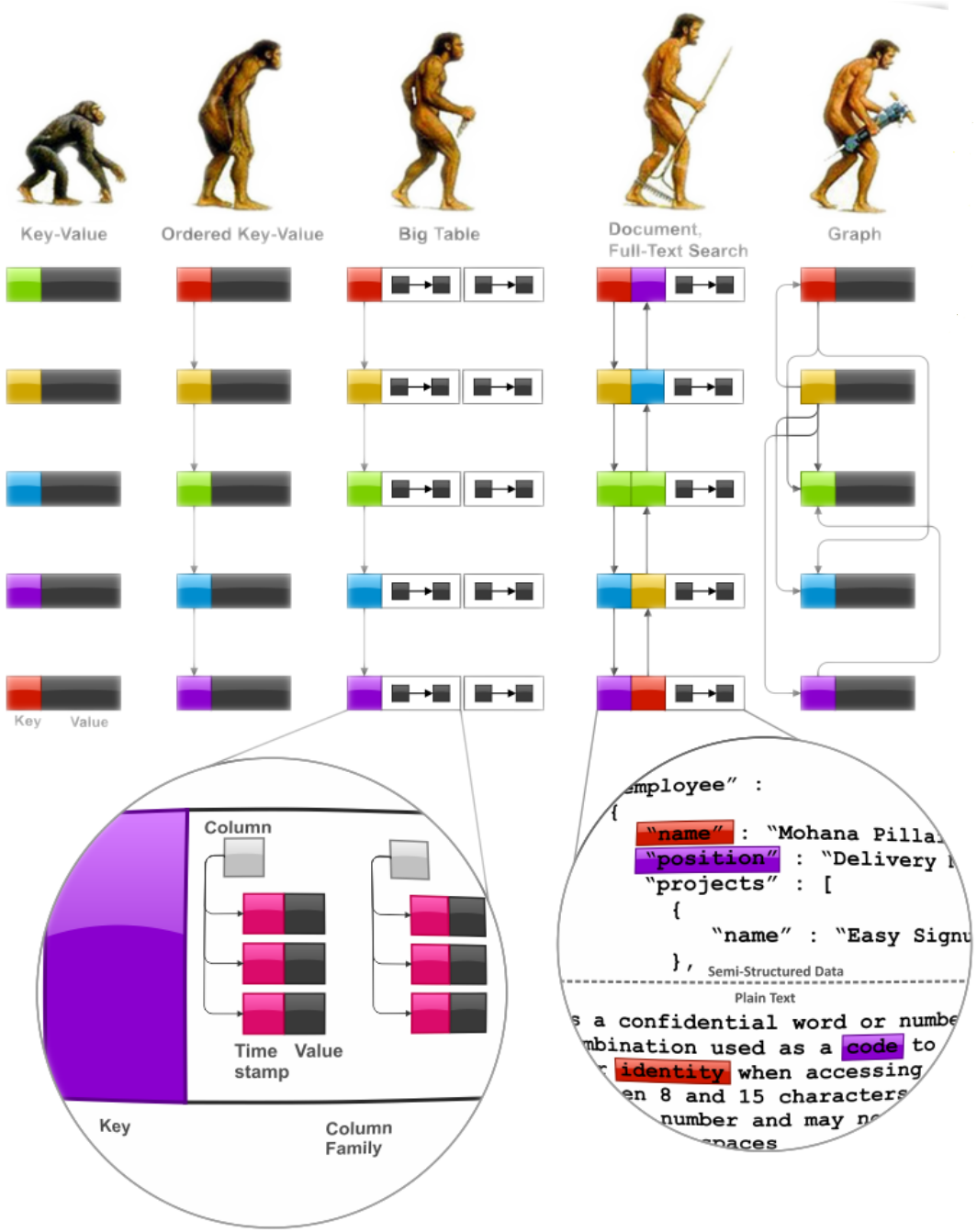
- **SQL/JSON 2016**

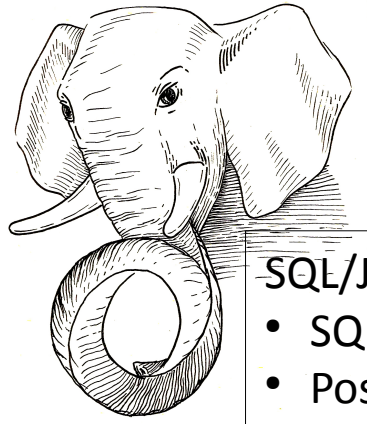
```
SELECT * FROM js_test WHERE
JSON_EXISTS( value,'$.**.color ?
(@ == "red")');
```
```

SQL/JSON availability

- Github Postgres Professional repository
  https://github.com/postgrespro/sqljson
- SQL/JSON examples
- WEB-interface to play with SQL/JSON
- Technical Report (SQL/JSON)
- BNF of SQL/JSON
- We need your feedback, bug reports and suggestions
- Help us writing documentation !

PROFESSIONAL Postgres





SQL/JSON - 2018
- SQL-2016 standard
- Postgres Pro - 2017

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

JSON - 2012
- Textual storage
- JSON verification

HSTORE - 2003
- Perl-like hash storage
- No nesting
- Indexing

PROFESSIONAL Postgres

JSONB COMPRESSION

Transparent compression of jsonb

+ access to the child elements without full decompression

jsonb compression: ideas

- **Keys replaced by their ID in the external dictionary**
- Delta coding for sorted key ID arrays
- Variable-length encoded entries instead of 4-byte fixed-size entries
- Chunked encoding for entry arrays
- Storing integer numerics falling into int32 range as variable-length encoded 4-byte integers

jsonb compression: implementation

- Custom column compression methods:

```
CREATE COMPRESSION METHOD name HANDLER handler_func

CREATE TABLE table_name (
      column_name data_type
      [ COMPRESSED cm_name [ WITH (option 'value' [, ... ]) ] ] ...
)

ALTER TABLE table_name ALTER column_name
      SET COMPRESSED cm_name [ WITH (option 'value' [, ... ]) ]

ALTER TYPE data_type SET COMPRESSED cm_name
```

- attcompression, attcmoptions in pg_catalog.pg_attributes

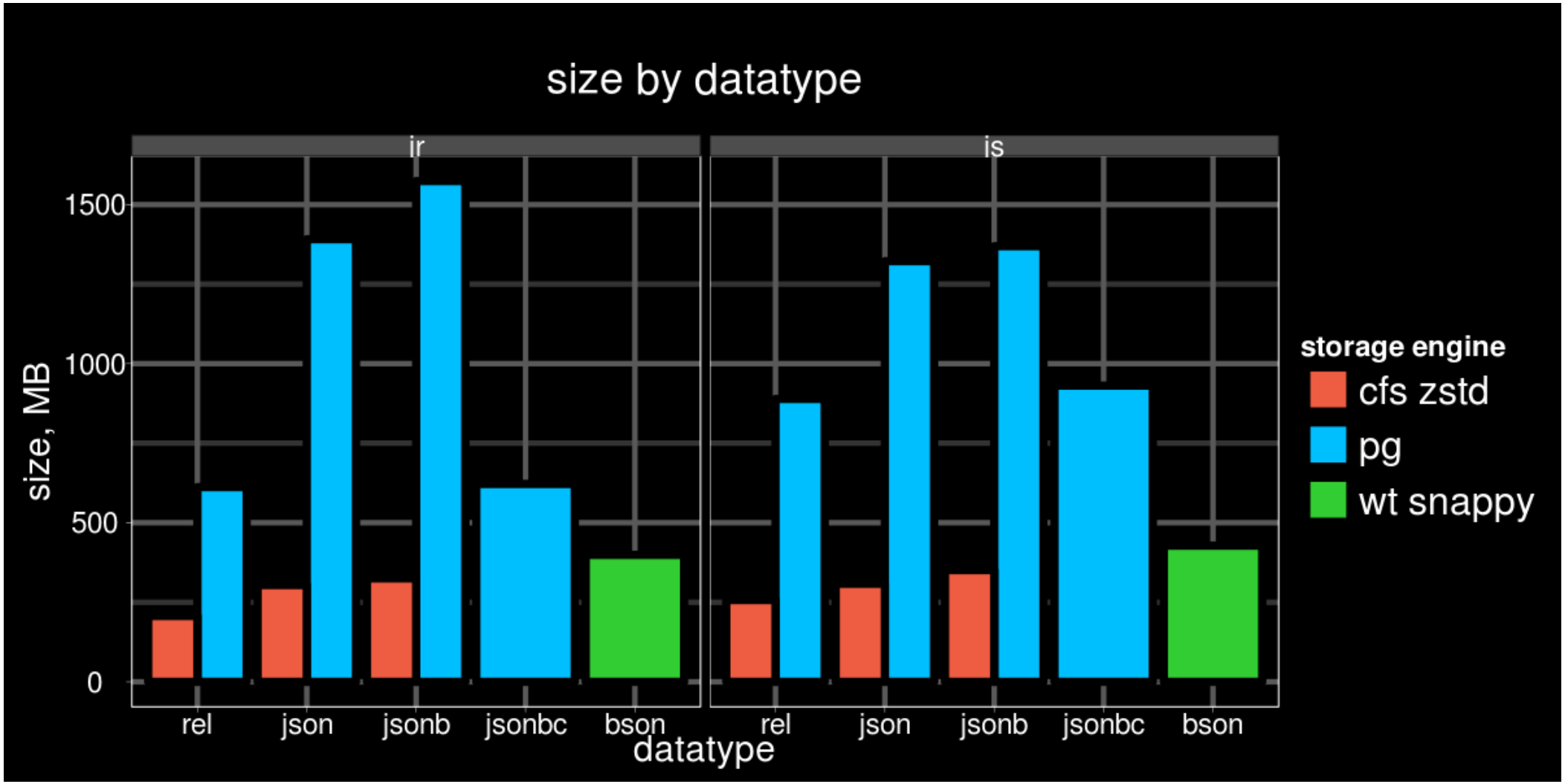
jsonb compression: results

Two datasets:

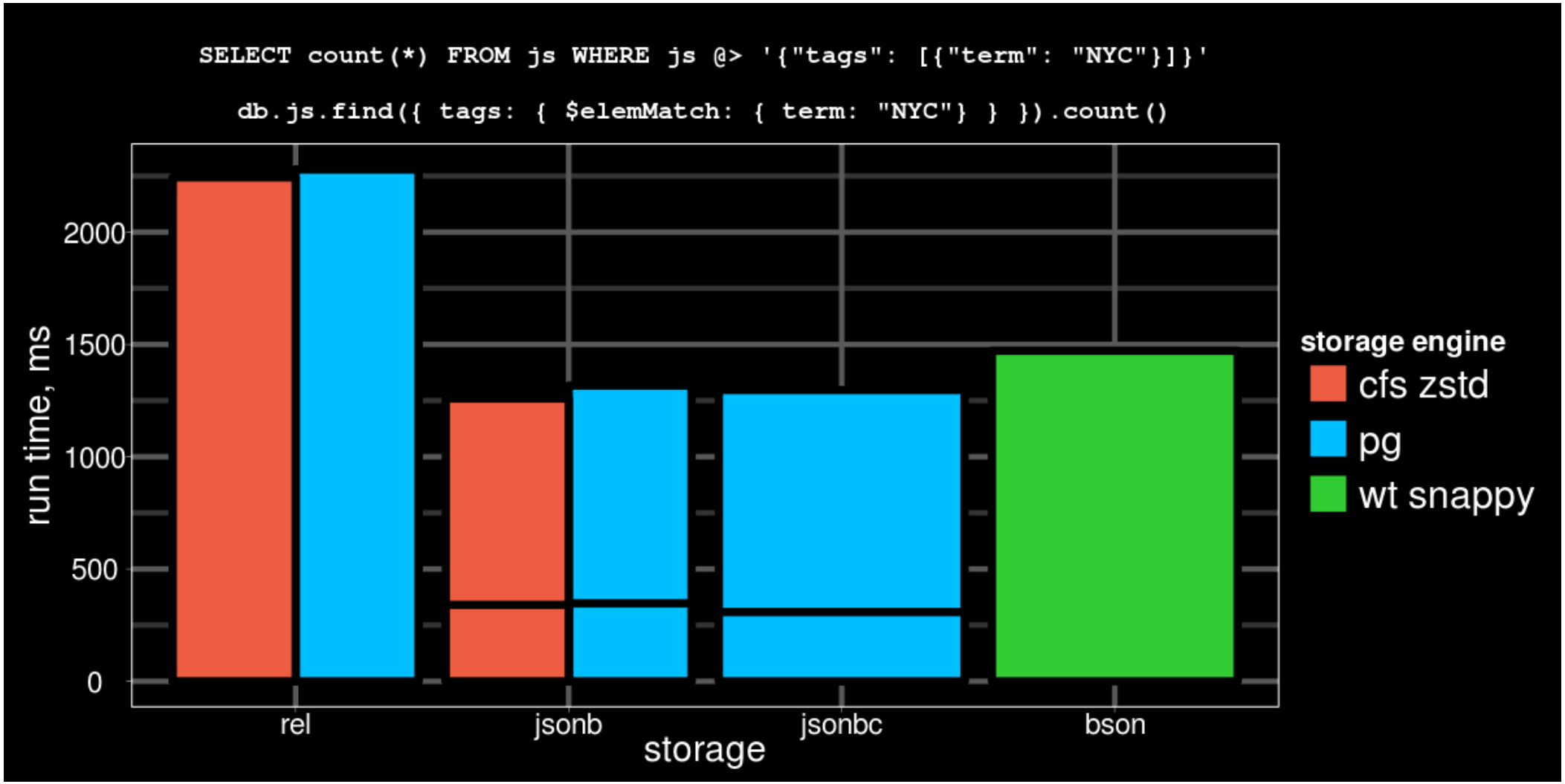
- js – Delicious bookmarks, 1.2 mln rows (js.dump.gz)
  - Mostly string values
  - Relatively short keys
  - 2 arrays (tags and links) of 3-field objects
- jr – customer reviews data from Amazon, 3mln (jr.dump.gz)
  - Rather long keys
  - A lot of short integer numbers

Also, jsonbc compared with CFS (Compressed File System) – page level compression and encryption in Postgres Pro Enterprise 9.6.

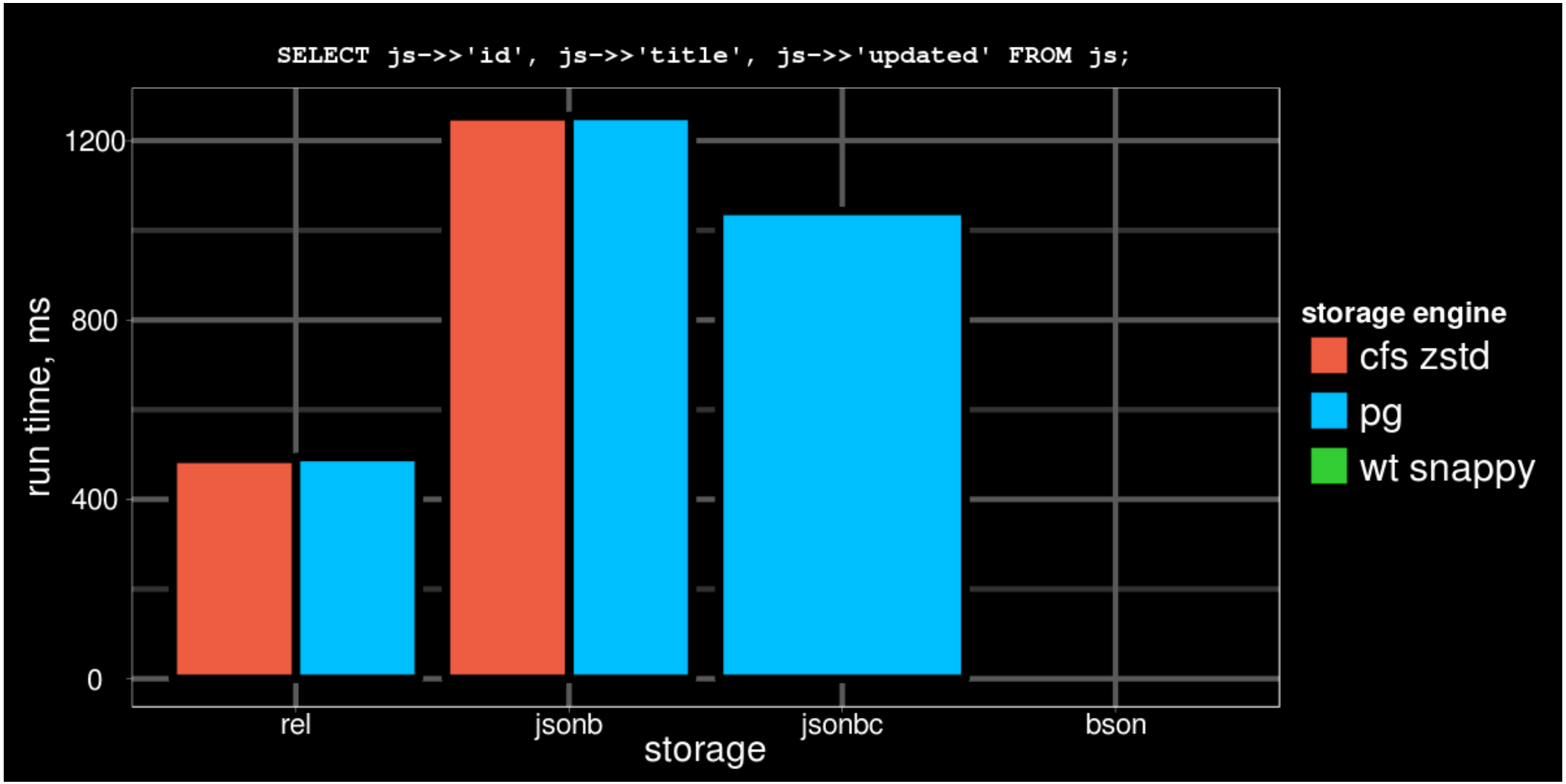
jsonb compression: table size



jsonb compression (js): performance



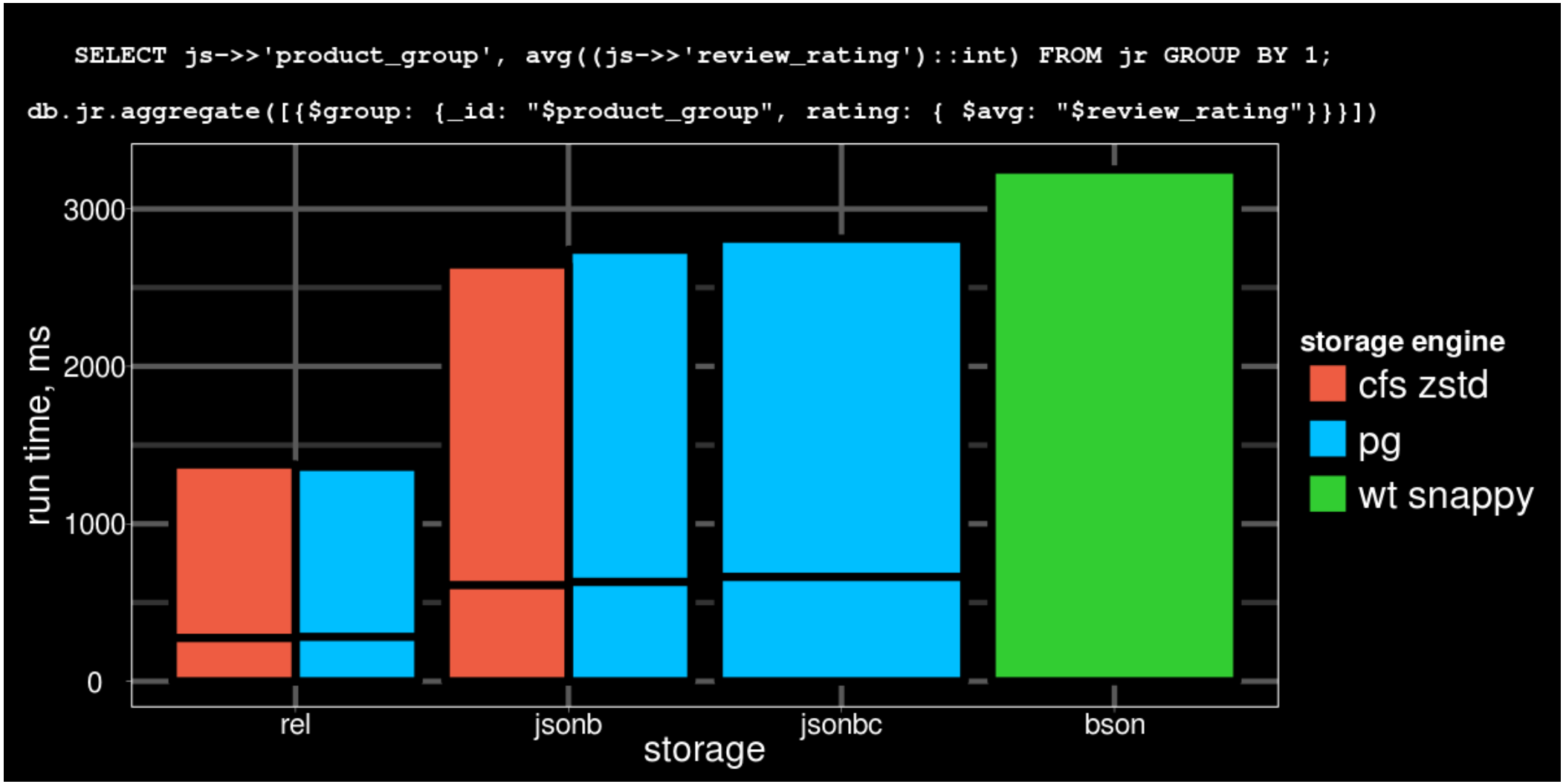
jsonb compression (js): performance



jsonb compression (jr): performance

```
SELECT js->>'product_group', avg((js->>'review_rating')::int) FROM jr GROUP BY 1;

db.jr.aggregate([{$group: {_id: "$product_group", rating: { $avg: "$review_rating"}}}])
```



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 methods
- Availability:

  https://github.com/postgrespro/postgrespro/tree/jsonbc

JSON[B] Text Search

- tsvector(configuration, json[b]) in Postgres 10

```
select to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::json)) foo(jb);
                                    to_tsvector
-------------------------------------------------------------------------------
'fals':10 'hous':18 'intern':17 'long':5 'moscow':16 'peac':12 'stori':6 'true':8 'war':14
```

```
select to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
                                    to_tsvector
-------------------------------------------------------------------------------
 'fals':14 'hous':18 'intern':17 'long':9 'moscow':16 'peac':1 'stori':10 'true':12 'war':3
```

JSON[B] Text Search

- Phrase search is [properly] supported !

```
select  phraseto_tsquery('english','war moscow') @@ to_tsvector(jb) from (values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 f
```

```
select  phraseto_tsquery('english','moscow international') @@ to_tsvector(jb) from
(values ('
{
     "abstract": "It is a very long story about true and false",
     "title": "Peace and War",
     "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 t
```

- Kudos to Dmitry Dolgov & Andrew Dunstan !

PROFESSIONAL Postgres

BENCHMARKS:
How NoSQL Postgres is fast

First (non-scientific) benchmark !

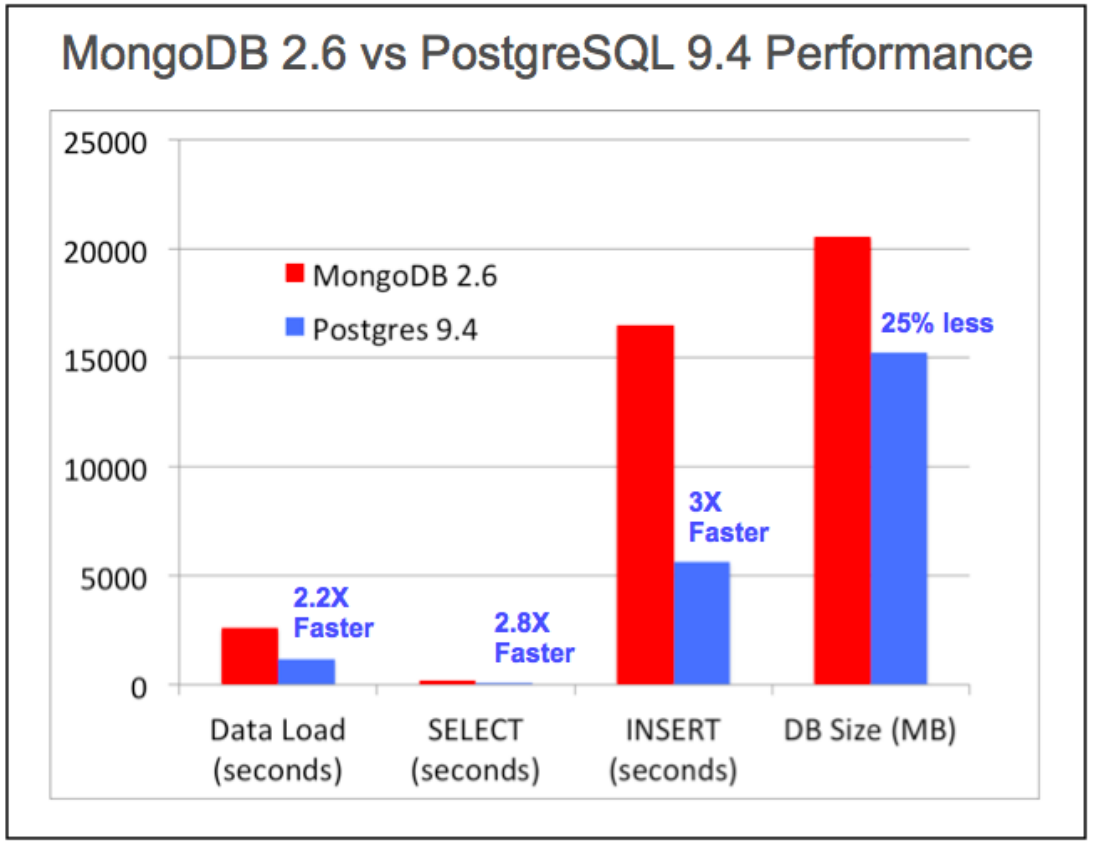
Summary: PostgreSQL 9.4 vs Mongo 2.6.0

- Search key=value (contains @>)
  - json : 10 s seqscan
  - jsonb : 8.5 ms GIN jsonb_ops
  - **jsonb : 0.7 ms GIN jsonb_path_ops**
  - mongo : 1.0 ms btree index
- Index size
  - jsonb_ops - 636 Mb (no compression, 815Mb)
    jsonb_path_ops - 295 Mb
  - jsonb_path_ops (tags) - 44 Mb USING gin((jb->'tags') jsonb_path_ops
  - mongo (tags) - 387 Mb
    mongo (tags.term) - 100 Mb

- Table size
  - postgres : 1.3Gb
  - mongo : 1.8Gb
- Input performance:
  - Text : 34 s
  - Json : 37 s
  - Jsonb : 43 s
  - mongo : 13 m

Engine Yard™

EDB NoSQL Benchmark

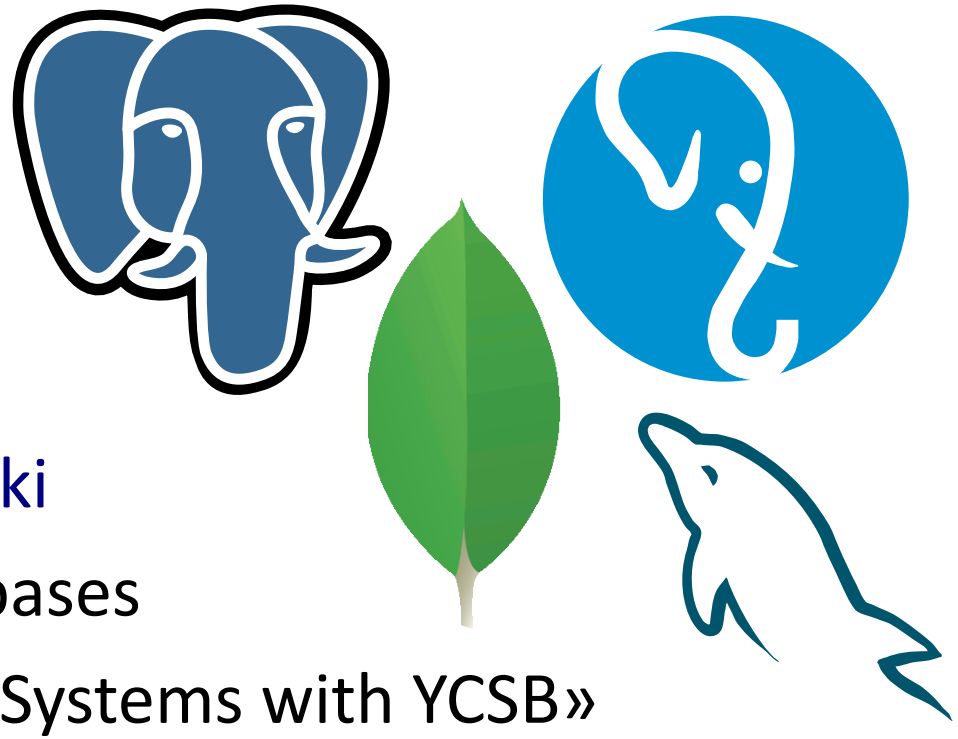


https://github.com/EnterpriseDB/pg_nosql_benchmark

Benchmarking NoSQL Postgres

- Both benchmarks were homemade by postgres people
- People tend to believe independent and «scientific» benchmarks
  - Reproducible
  - More databases
  - Many workloads
  - Open source

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, Postgres Pro Enterprise 2.0, MongoDB 3.4.2, Mysql 5.7.17
  - 1 server with 24 cores, 48 GB RAM for clients
  - 1 server with 24 cores, 48 GB RAM for database
  - 10Gbps switch

YCSB Benchmark: Core workloads

- Workload A: Update heavy - a mix of 50/50 reads and writes
- Workload B: Read mostly - a 95/5 reads/write mix
- Workload C: Read only — 100% read
- Workload D: Read latest - new records are inserted, and the most recently inserted records are the most popular
- Workload E: Short ranges - short ranges of records are queried
- Workload F: Read-modify-write - the client will read a record, modify it, and write back the changes
- All (except D) workloads uses Zipfian distribution for record selections

YCSB Benchmark: details (1)

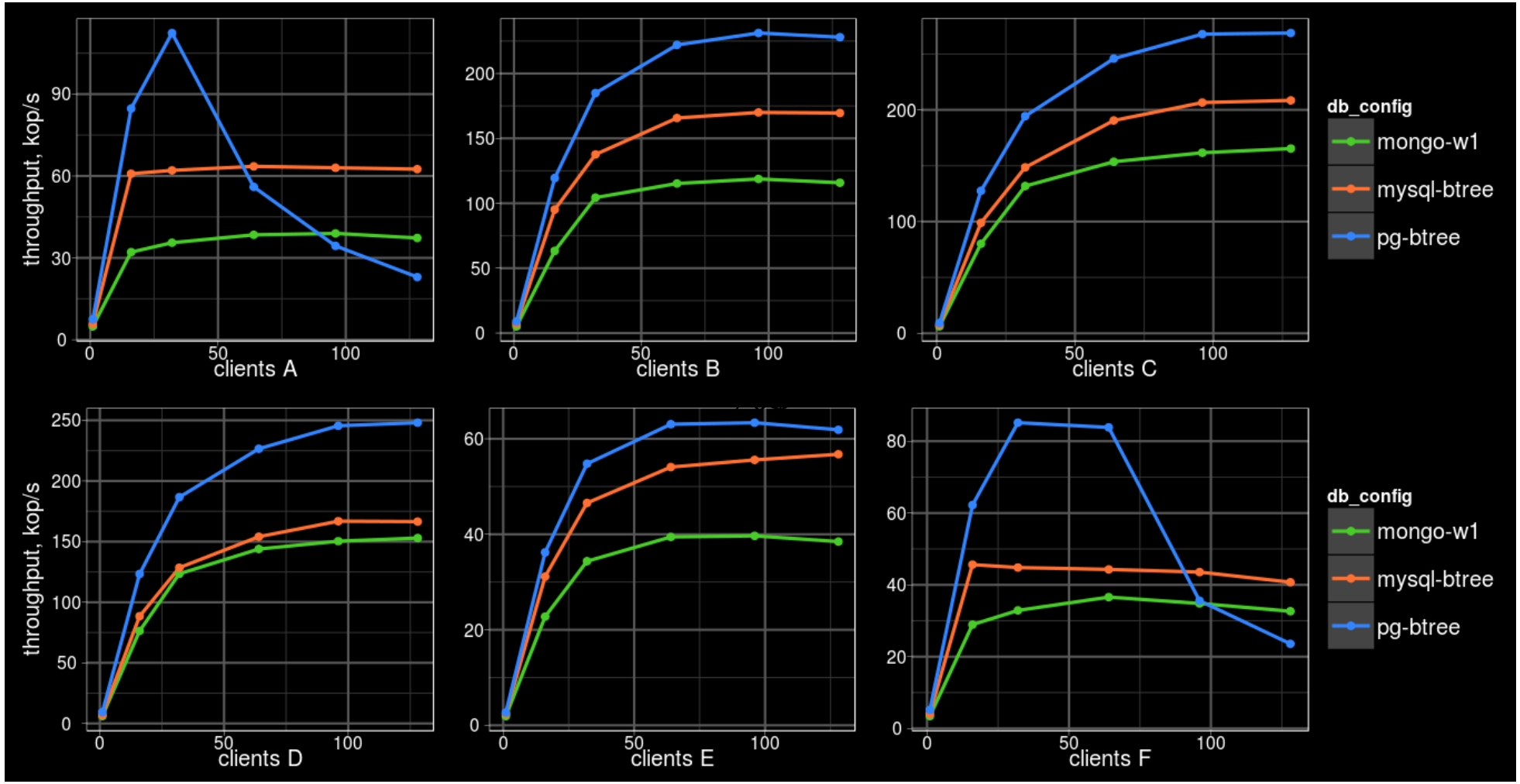
- Postgres (9.6, master), asynchronous commit=on
  Mongodb 3.4.2 (w1, j0) — 1 and 5 mln. rows
- Postgres (9.6, master), asynchronous commit=off
  Mongodb 3.4.2 (w1, j1) — 100K rows
- MySQL 5.7.17 + all optimization (Thanks, Alexey Kopytov)
- We tested:
  - Functional btree index for jsonb, jsonbc, sqljson, cfs (compressed) storage
  - Gin index (fastupdate=off) for jsonb, jsonb_build_object
  - Mongodb (wiredtiger with snappy compression)
  - Return a whole json, just one field, small range
  - 10 fields, 200 fields (TOASTed)

YCSB Benchmark: details (2)

- Client machine load:
  - Postgres <= 30%
    Mongodb <= 55%
- Server machine load:
  - Postgres — 100%
  - MySQL — 100%
  - MongoDB — 70%

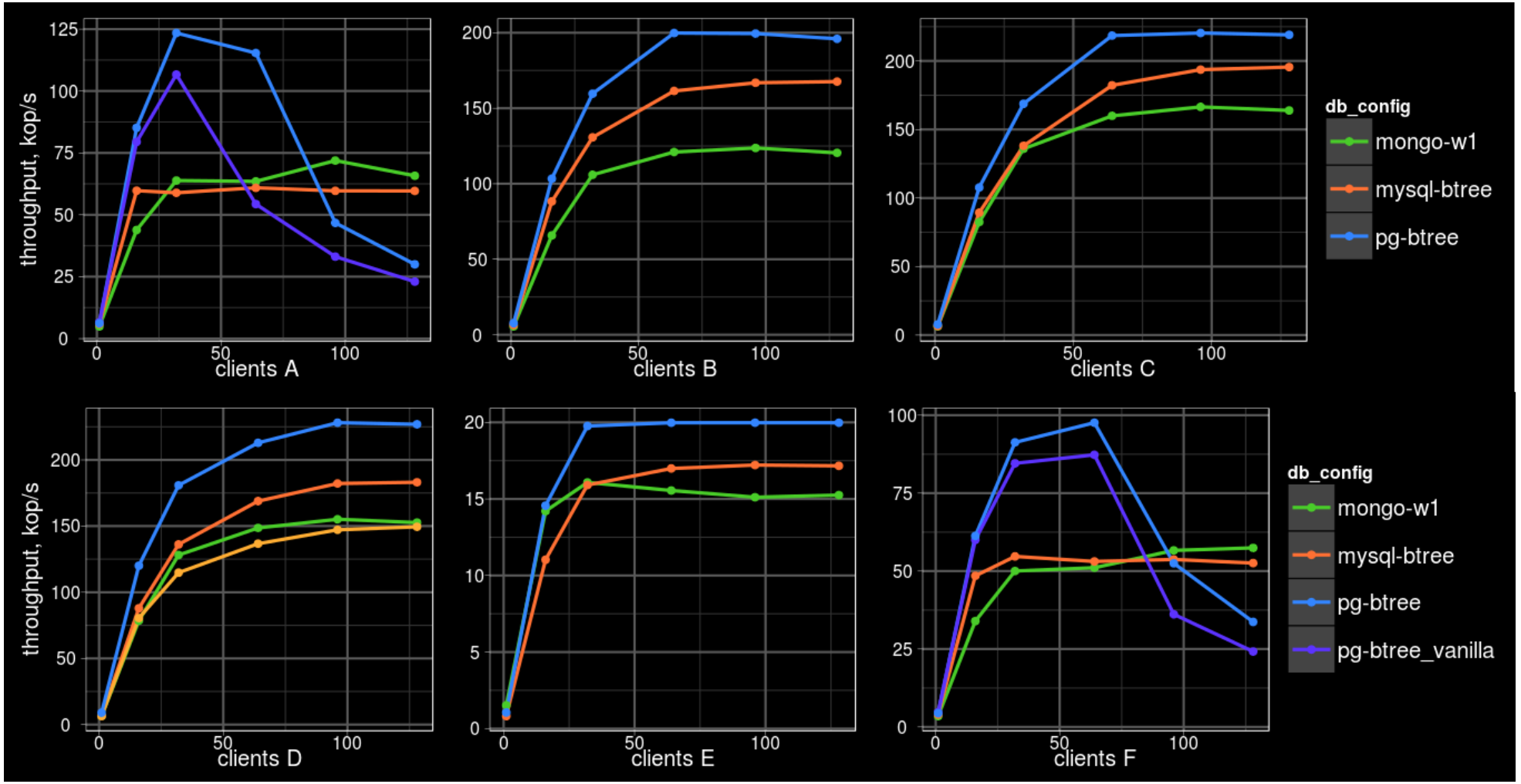
1 mln rows, 10 fields, select 1 key

- Postgres is better in all R/O workloads
- Postgres isn“t scaling well for heavy R/W workloads (a,f)



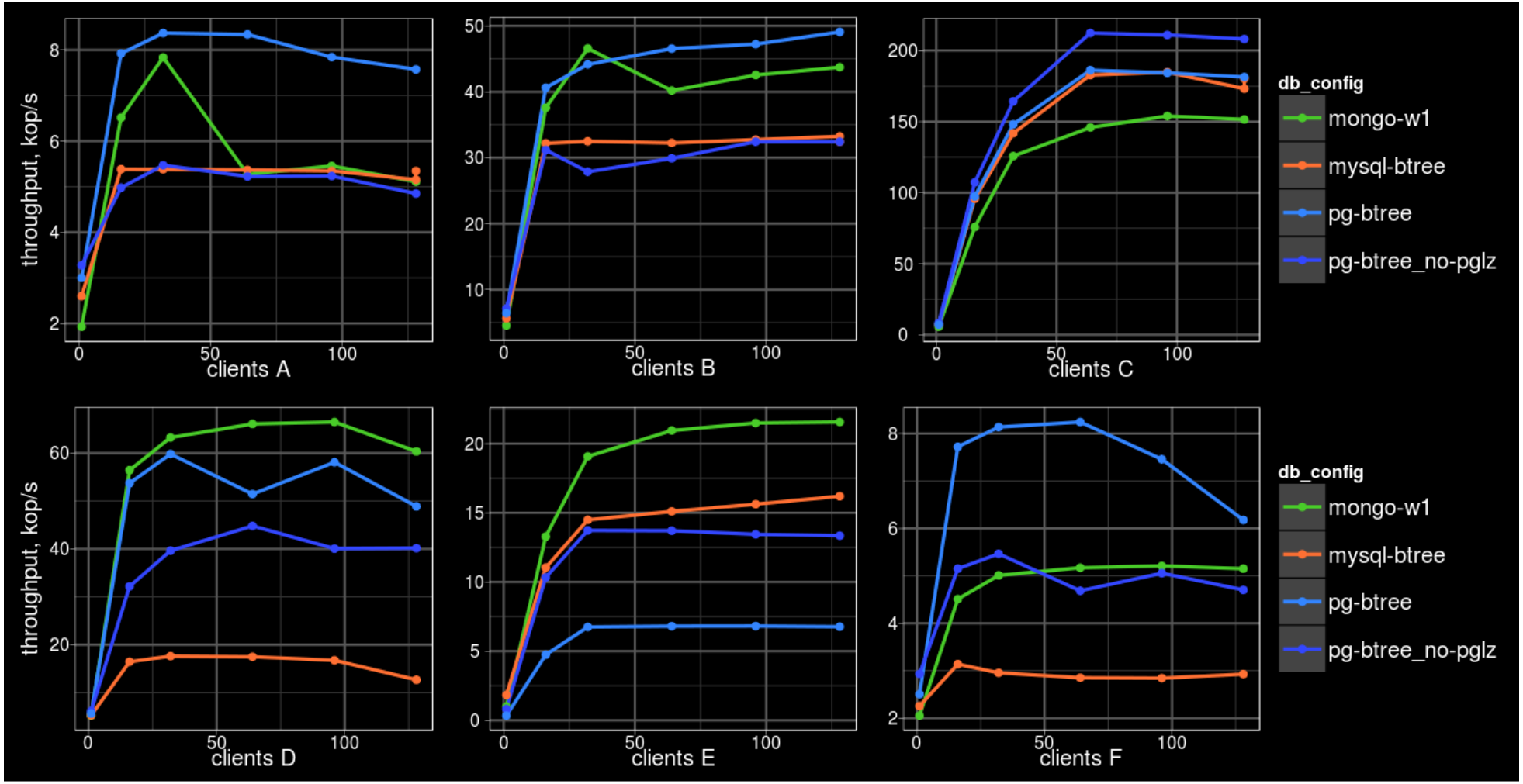
1 mln rows, 10 fields, select all keys

- Postgres is better in all R/O workloads
- Postgres isn“t scaling well for heavy R/W workloads (a,f)



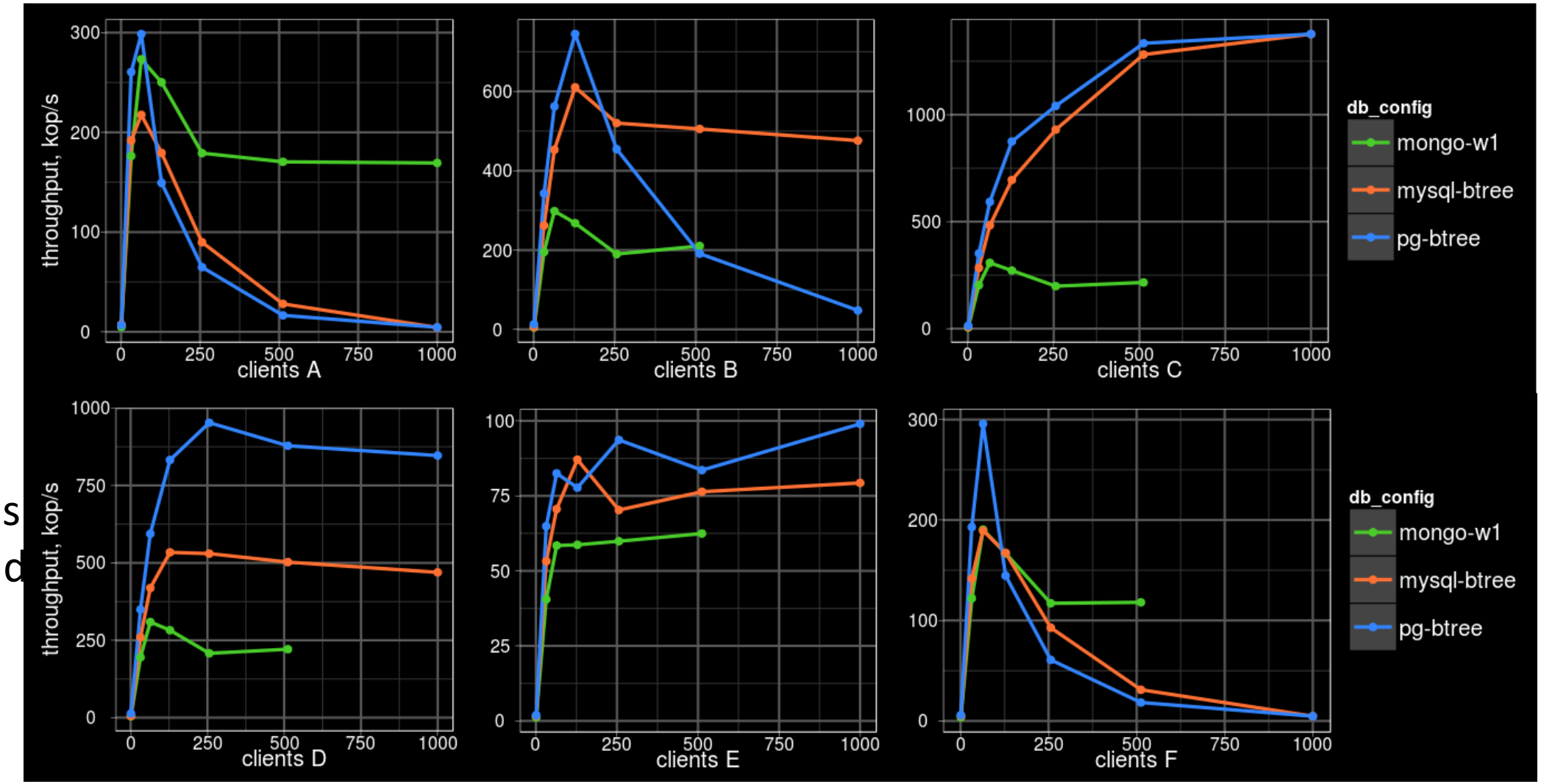
1mln rows, 200 fields, select 1 key

- TOASTed json are really bad
- Mongo win in D,E workloads



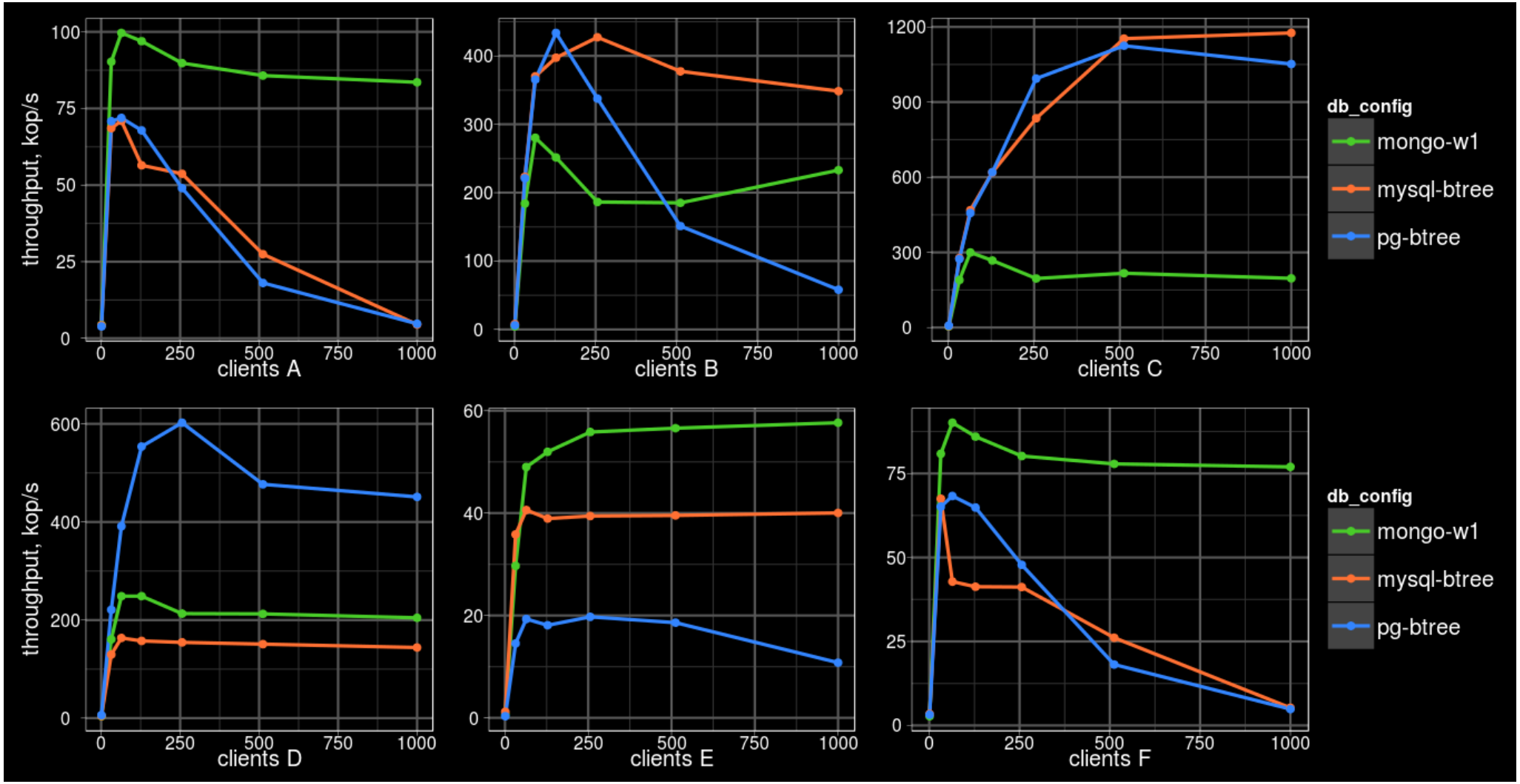
1 mln rows, 10 fields, select 1 key
BIG 144 cores, 3TB ram, 2 Tb SSD

- Postgres and MySQL better use multiple cores (1.5 mln ops/sec !)
- Postgres not scaled well in R/W workloads (huge overhead in isolations)



1 mln rows, 200 fields, select 1 key
BIG 144 cores, 3TB ram, 2 Tb SSD

- MongoDB win on A, E, F workloads ! One writer is better for Zipf distribution.



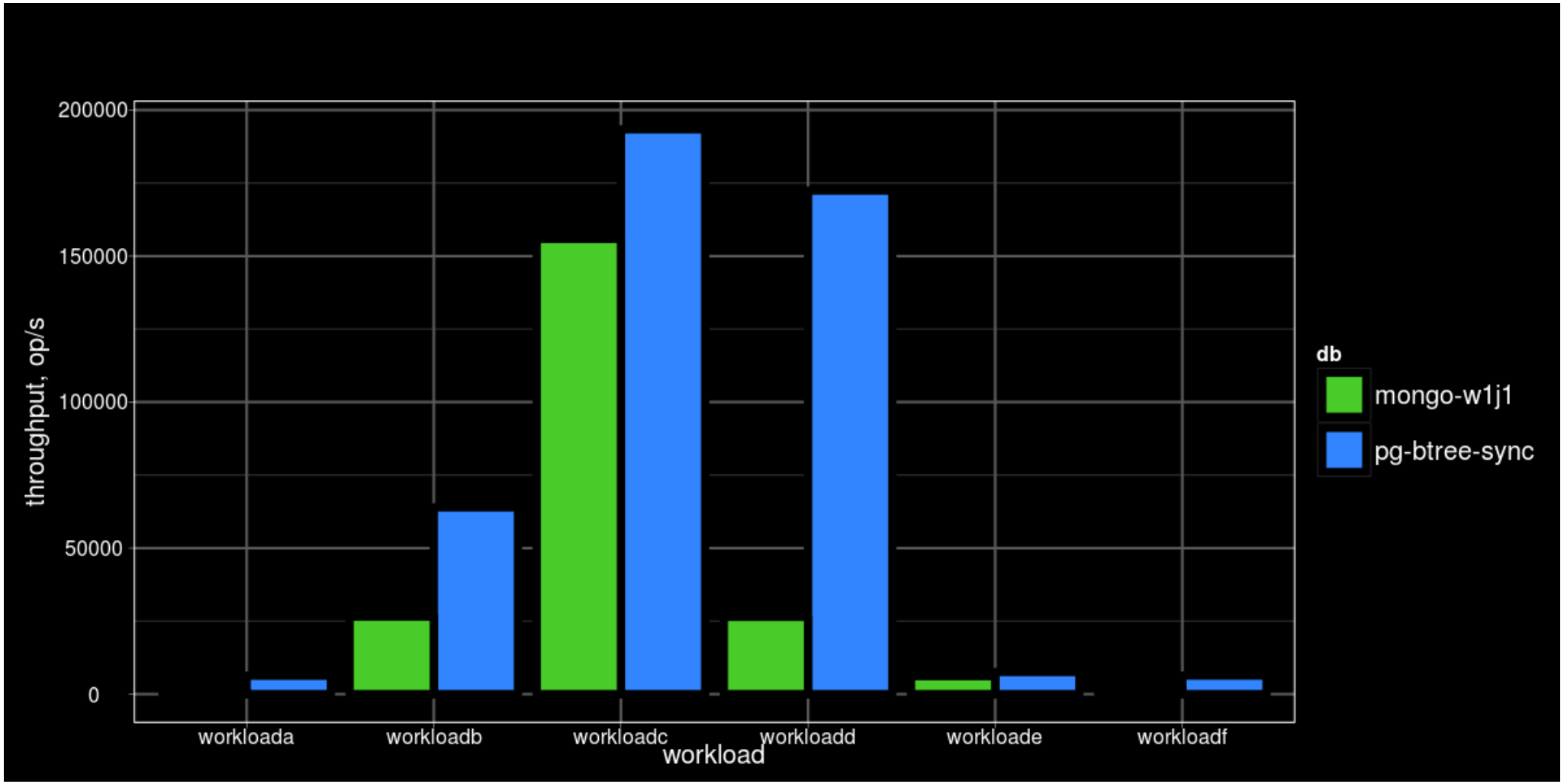
Summary

Low durability: synchronous_commit=off, j0

- Postgres and MySQL scales good on R/O workloads
- Postgres has inefficient transactions locking (isolation) on high contention (zipf distribution and large number of clients)
- PGLZ in TOAST is cpu-hungry, range queries (workload E) suffer.
- MySQL is better than Postgres on R/W (zipf distribution and large number of clients), especially in workload B (5% update).
- Mongo doesn“t degrades on R/W with high contention, especially on long json (one writer helps).
- Postgres (synchronous_commit=on) win Mongo with durability enabled (j1).

100K rows, 10 fields, journal on disk

- Mongo — j1
  Postgres - async.commit is on
- Postgres is better in all workloads !



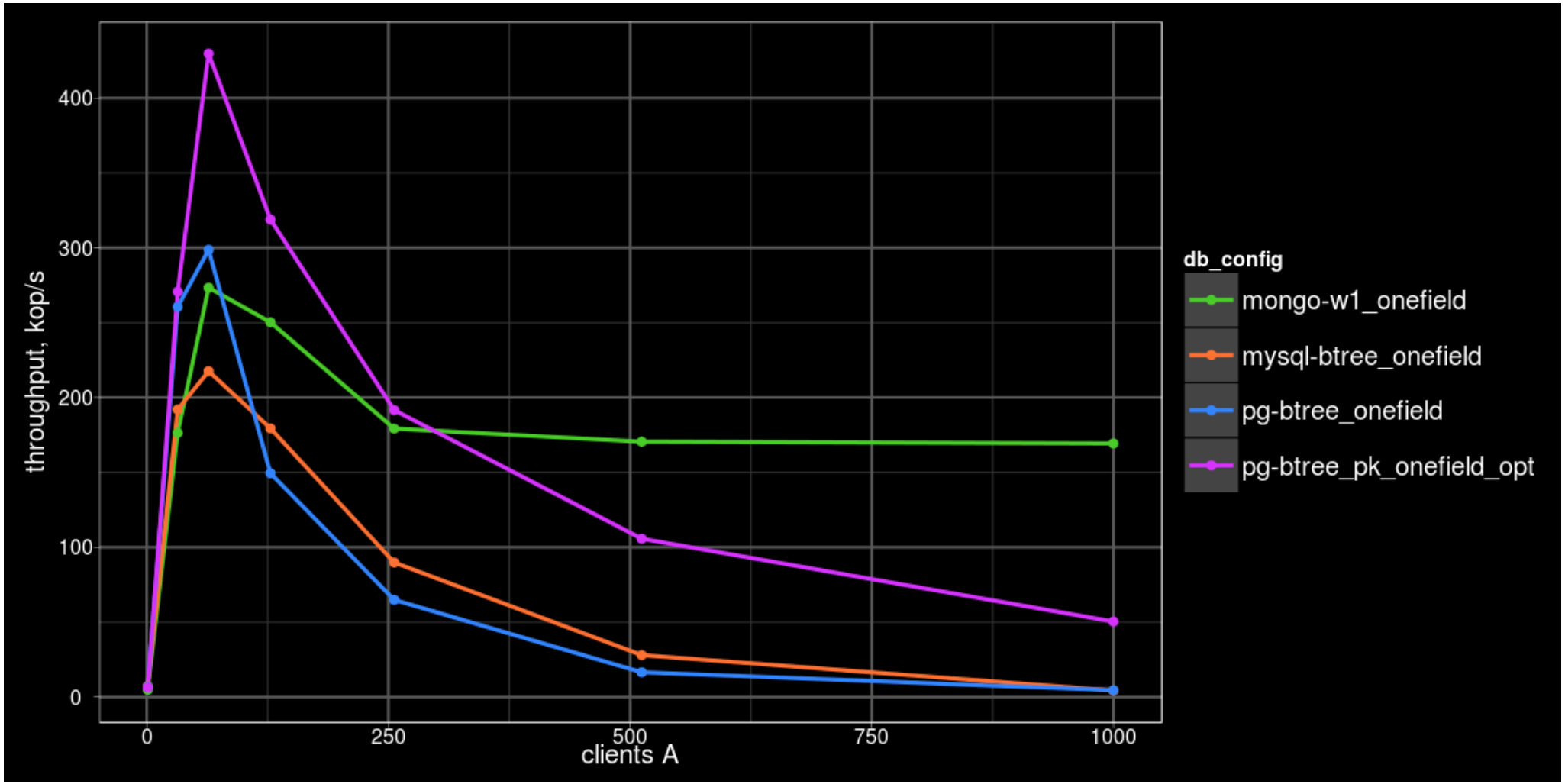
Optimization of high-contention write

- PostgreSQL 10.0 +
- Deadlock_timeout = 60 (Discussion)
- Use primary key instead of tid for lock tuple until end of transaction
- Primary key as separate column to eliminate HOT patch

```
CREATE TABLE usertable (
  ycsb_key text PRIMARY KEY,
  data jsonb
);
```

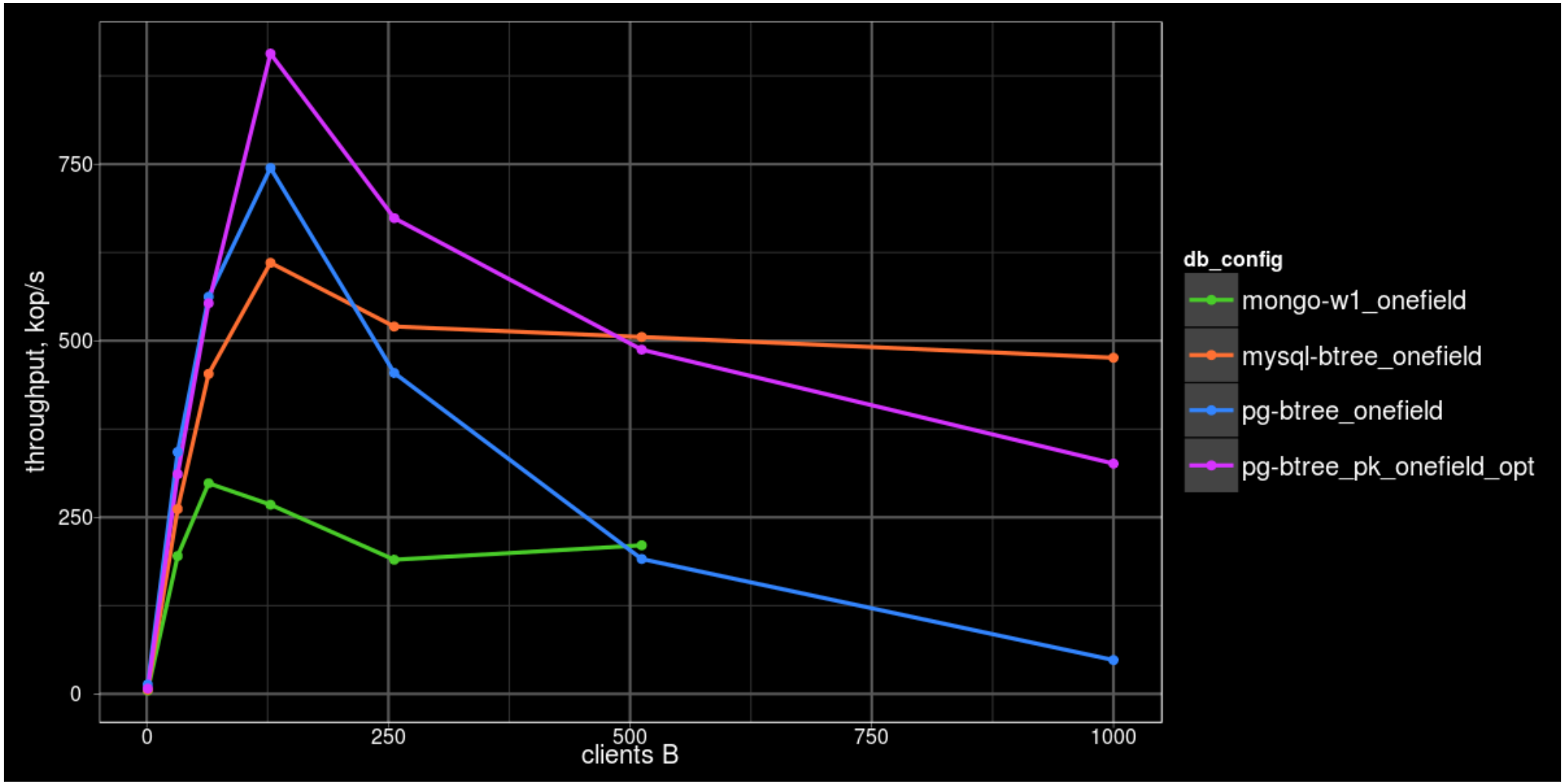
Big problem solved ? 50% updates

- 72 cores
- 3 TB RAM
- 2 TB SSD



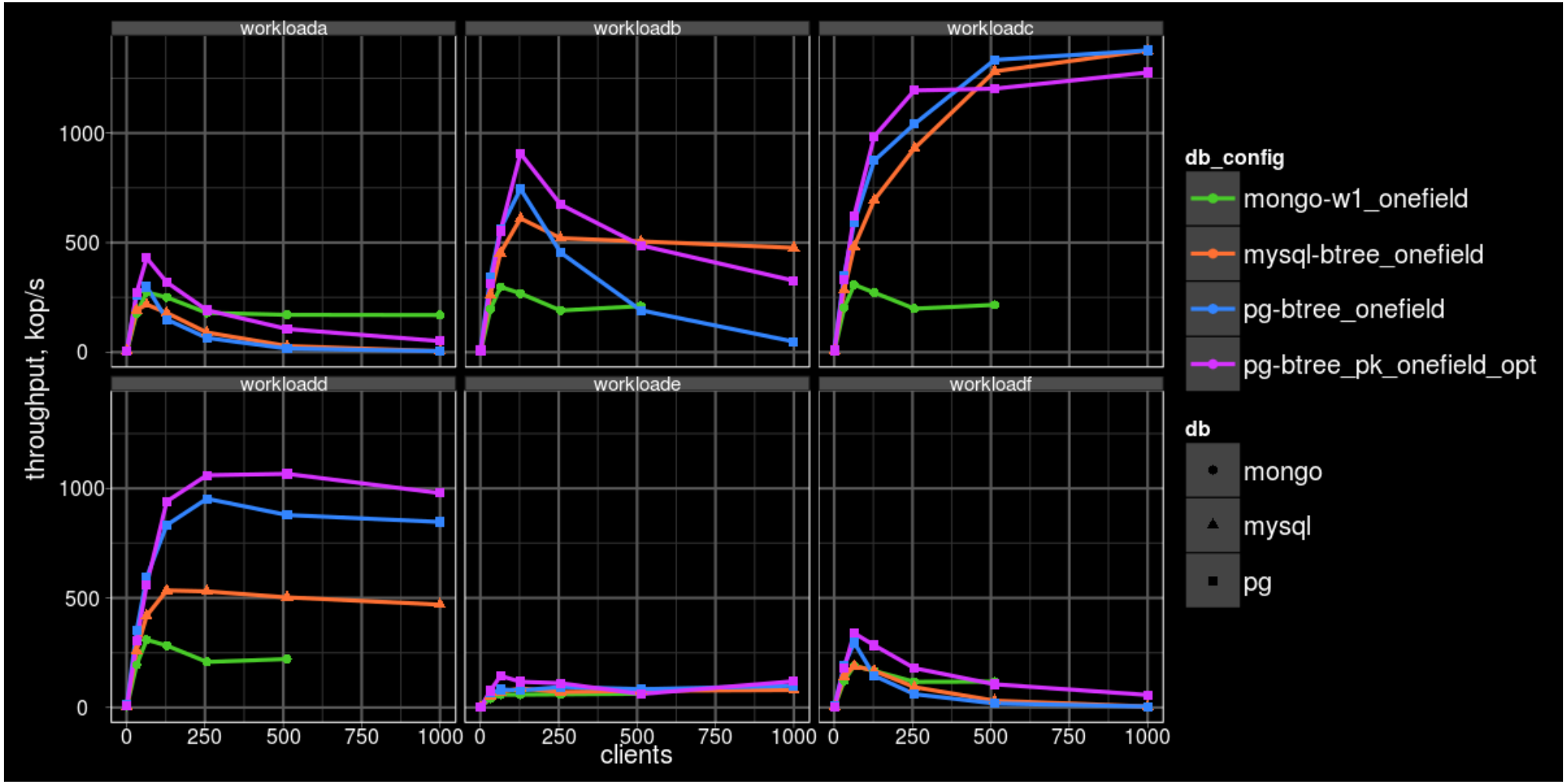
Big problem solved ? 5% updates

- 72 cores
- 3 TB RAM
- 2 TB SSD



Big problem on big machine: now better but still exists.

- 72 cores
- 3 TB RAM
- 2 TB SSD



Big problem solved ?

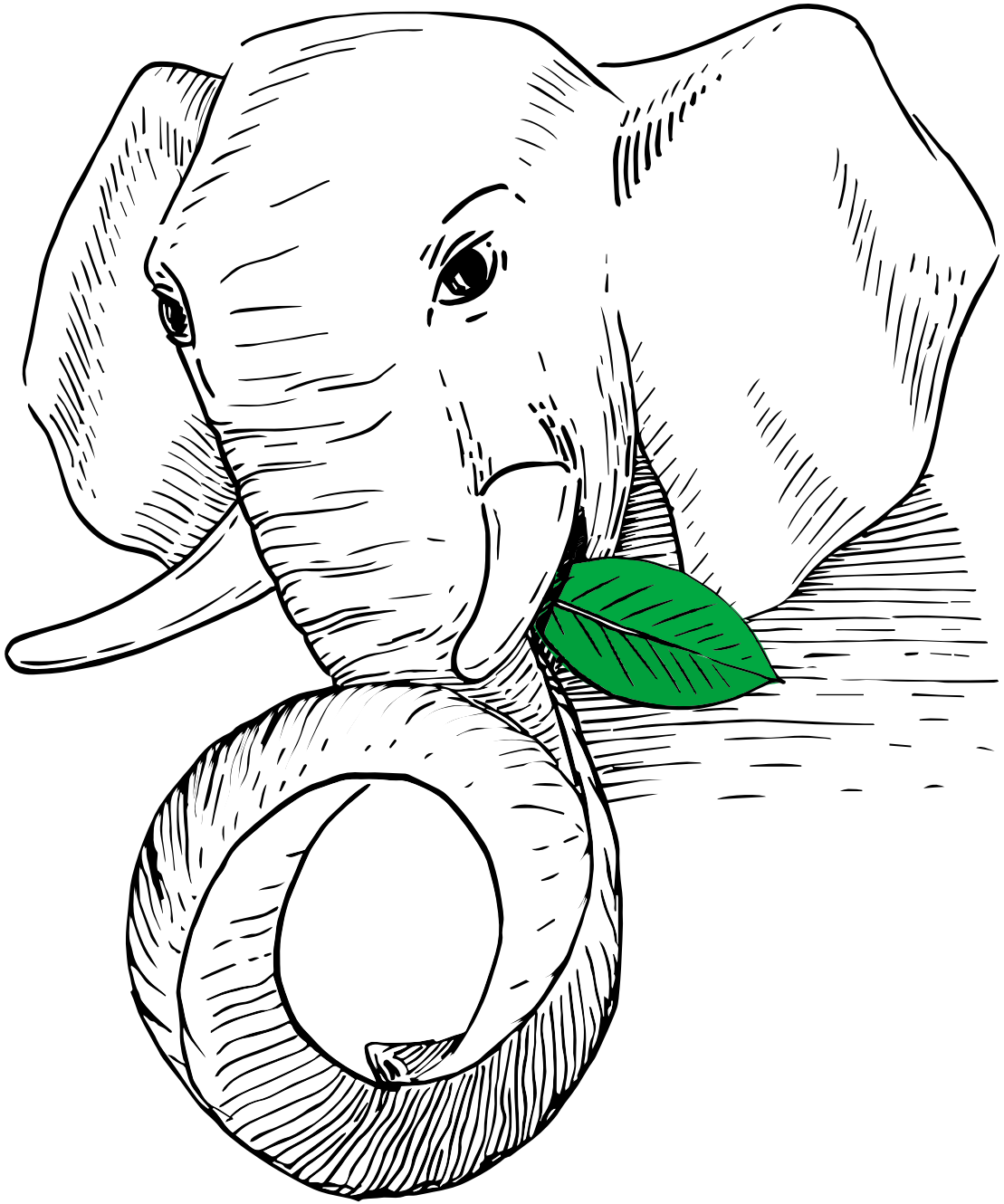
- PostgreSQL still degrades on high concurrency (#clients > 250) for workloads A,B (high contention writes)

Postgres PROFESSIONAL

PostgreSQL

beats

MongoDB !



PROFESSIONAL Postgres

Still need more tps ?

Use partitioning

- Upcoming version of pg_pathman supports partitioning by expression
- Delicious bookmarks dataset — 5 partitions

```
SELECT pathman.create_hash_partitions('jb', 'jb->>''id''', 5);
 create_hash_partitions
------------------------
                      5
(1 row)

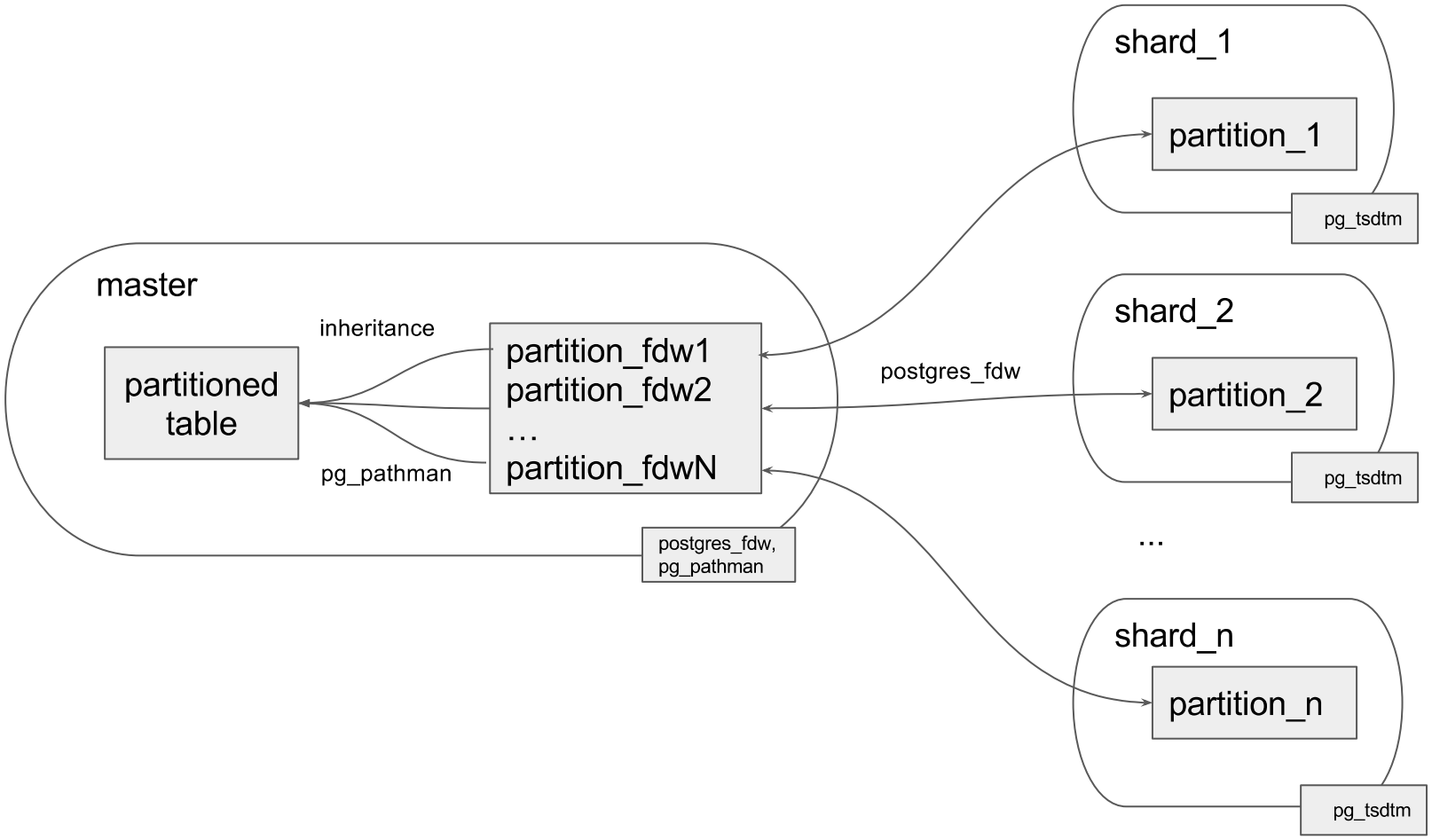
SELECT * FROM jb
WHERE (jb->>'id') = 'http://delicious.com/url/c91427110a17ad74de35eabaa296fa7a#kikodesign';
```

  - Vanilla 9.6 - 818, 274 (parallel) +pg_pathman - 173, 84 (parallel)
- Delicious bookmarks dataset — 1000 partitions
  - Vanilla 9.6 — 505 ms (27 ms) + pg_pathman — 1 ms (0.47 ms) !

PROFESSIONAL Postgres

Still need more tps ?

Use sharding !



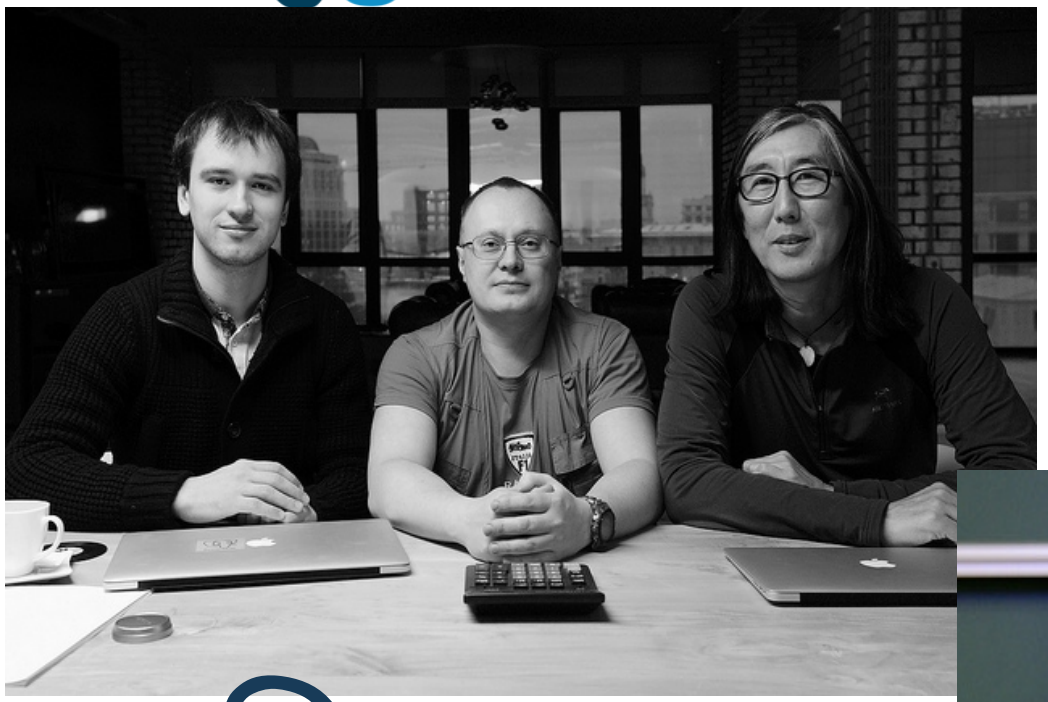
Sharding with postgres_cluster

- Master: fork postgres_cluster
  https://github.com/postgrespro/postgres_cluster
- Shards: pg_tsdtm
  https://github.com/postgrespro/pg_tsdtm

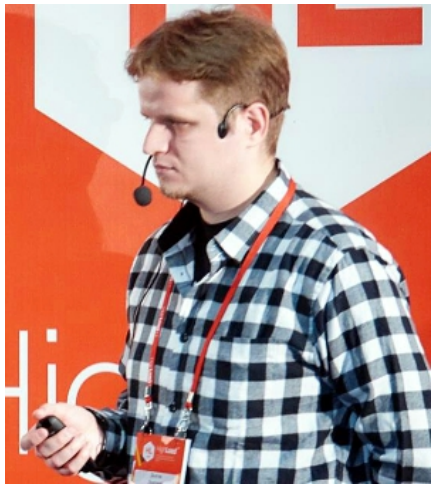
Summary

- Postgres is already a good NoSQL database + clear roadmap
- Move from NoSQL to Postgres to avoid nightmare !
- SQL/JSON will provide better flexibility and interoperability
  - Expect it in Postgres 11 (Postgres Pro 10)
  - Need community help (testing, documentation)
- JSONB dictionary compression (jsonbc) is really useful
  - Expect it in Postgres 11 (Postgres Pro 10)
- Postgres beats Mongodb in one node configuration
  - Postgres performance degrades on high-contention write (#clients > 100)
  - Next: YCSB benchmarks in distributed mode

PEOPLE BEHIND JSON[B]



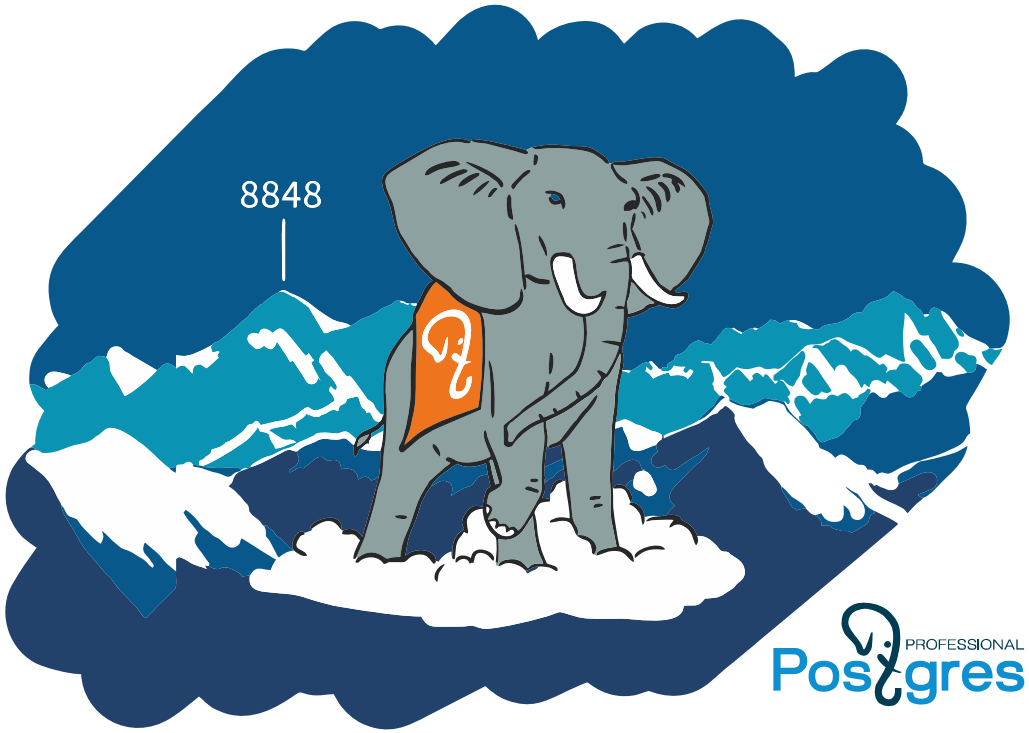
Nikita Glukhov



Контакты:

- Олег Бартунов, obartunov@postgrespro.ru
- Www.postgrespro.ru - смотрите Образование
- Реестр задач для разработчиков
- Hacking Postgres
- Developer FAQ
- Ресурсы для разработчиков на C
- Мой ЖЖ: obartunov.livejournal.ru
  (постгрес, горы, фото)
- Telegram: @pgsql
- Группа в FB: PostgreSQL в России

EVEREST MARATHON 2017



Thanks !