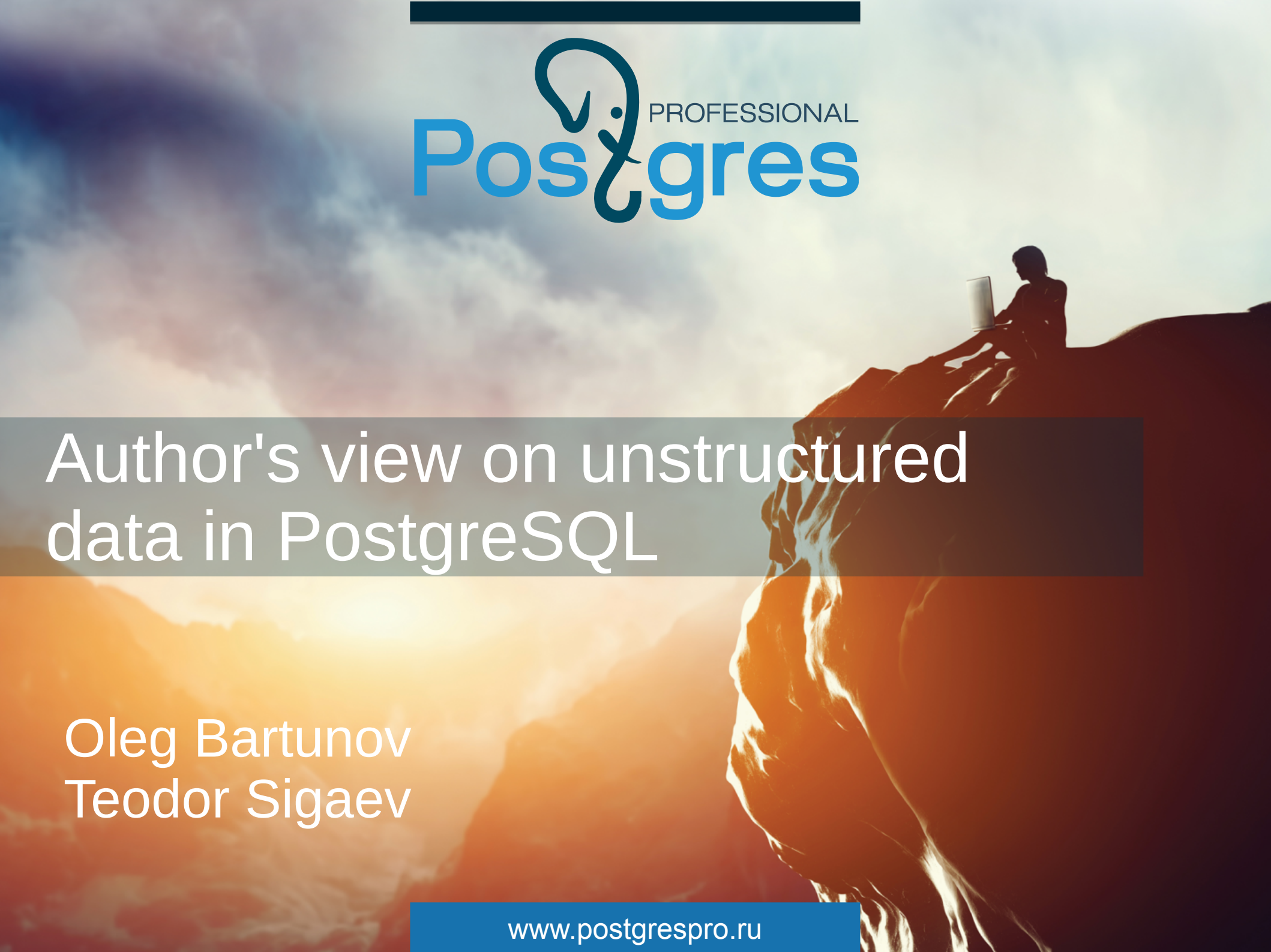


Author's view on unstructured data in PostgreSQL

Oleg Bartunov
Teodor Sigaev

www.postgrespro.ru

Russian PostgreSQL developers

Oleg Bartunov, Teodor Sigaev, Alexander Korotkov

PostgreSQL CORE

- Locale support
- PostgreSQL extendability:
- GiST(KNN), GIN, SP-GiST
- Full Text Search (FTS)
- NoSQL (hstore, jsonb)
- Indexed regexp search
- VODKA access method (WIP)

Расширения:

- Intarray
- Pg_trgm
- Ltree
- Hstore
- plantuner
- JsQuery

- Speakers at PGCon, PGConf: 20+ talks
- GSoC mentors
- PostgreSQL committers
- Conference organizers
- 50+ years of PostgreSQL expertise: development, audit and consulting
- Novartis, Raining Data, Heroku, Engine Yard, WarGaming, Rambler, Avito, 1c

Un/Semi-structured data

Who does use it?

Un/Semi-structured data

What is it?

Un/Semi-structured data

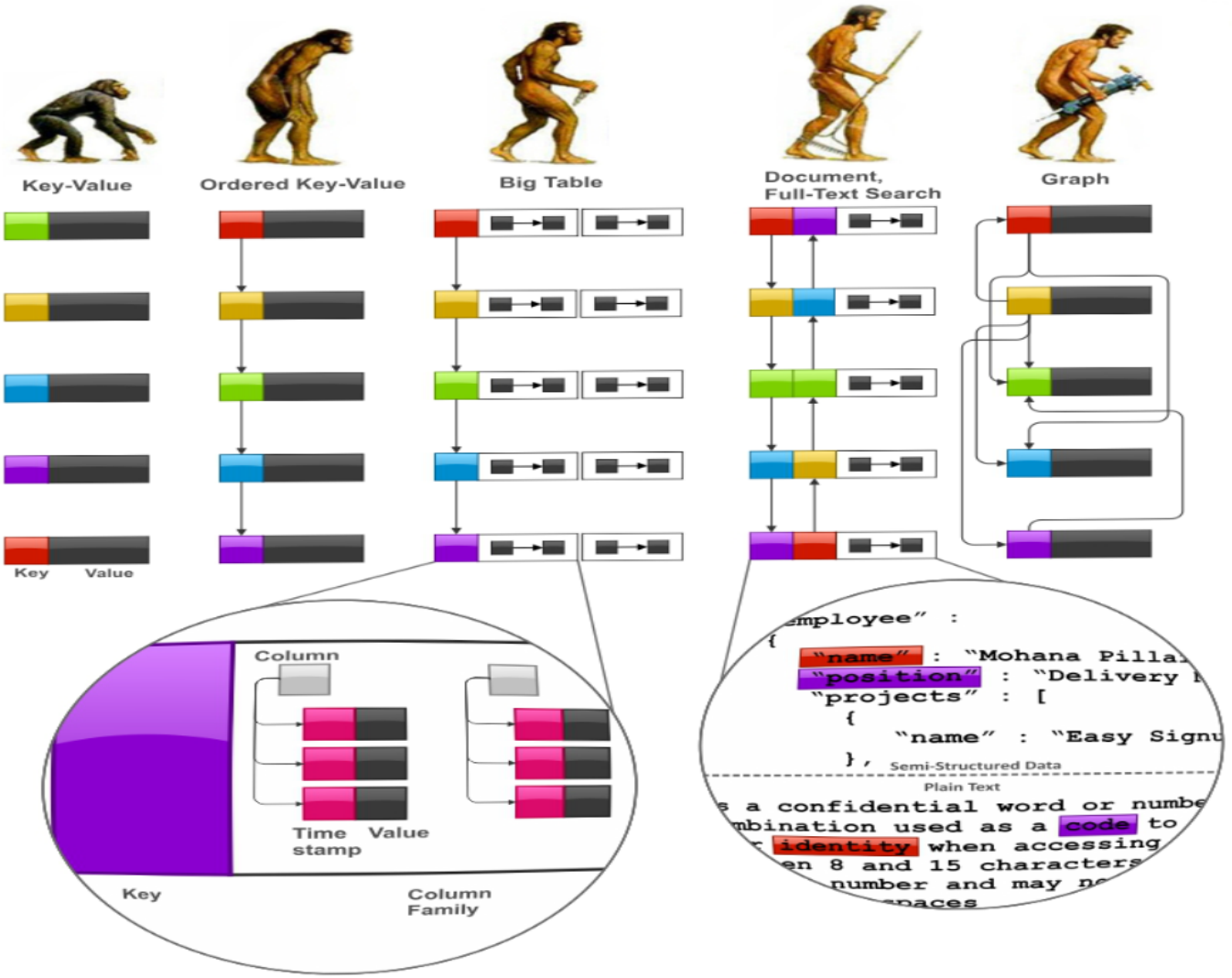
Structure exists but not well defined or too lazy to describe

Non/Semi-structured data

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

- Service itself can aggregate data, check consistency of data
- High concurrency, simple queries
- Simple database (key-value) is ok
- Eventual consistency is ok, no ACID overhead

Application needs faster releases
NoSQL databases match all of these — scalable, efficient, fault-tolerant, no rigid schema, ready to accept any data.


Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
Key
Value
Column
Time stamp
Value
Key
Column
Family
"name" : "Mohana Pilla
"position" : "Delivery
"projects" : [
{
"name" : "Easy Signu
},
Semi-Structured Data
Plain Text
a confidential word or numbe
mbination used as a code to
identity when accessing
en 8 and 15 characters
number and may n
paces

Postgres PROFESSIONAL

NoSQL

- Key-value databases
  - Ordered k-v for ranges support
- Column family (column-oriented) stores
  - Big Table — value has structure:
    - column families, columns, and timestamped versions (maps-of maps-of maps)
- Document databases
  - Value has arbitrary structure
- Graph databases — evolution of ordered-kv

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

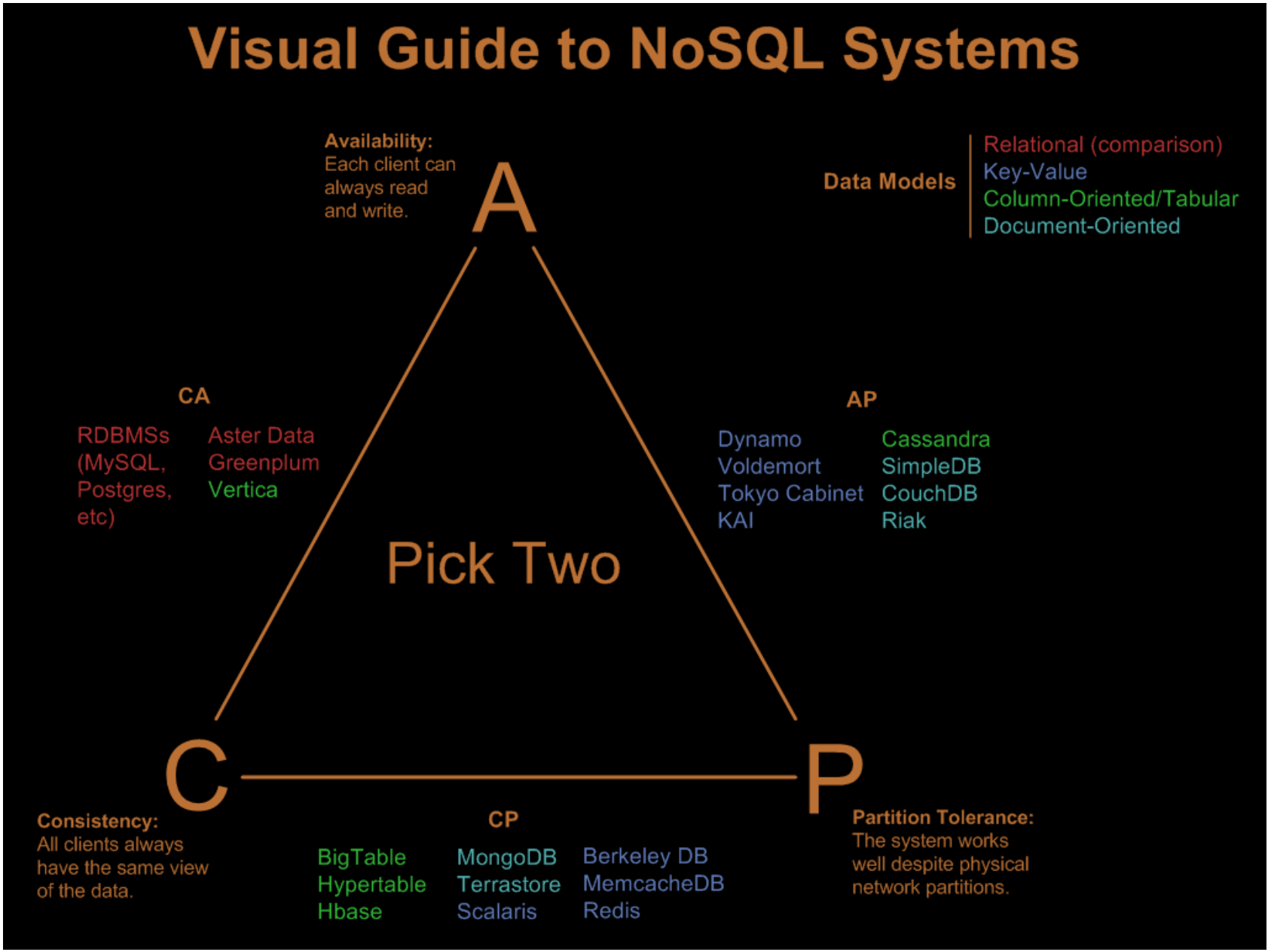
* Dynamo
* Cassandra
* Project Voldemort

Ordered key-value store

* NMDB
* Luxio
* Memcachedb
* Berkeley DB

Object database

* Db4o
* InterSystems Caché
* Objectivity/DB
* ZODB


Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica
AP
Dynamo
Voldemort
Tokyo Cabinet
KAI
Cassandra
SimpleDB
CouchDB
Riak
Pick Two
C
P
Consistency:
All clients always have the same view of the data.
CP
BigTable
Hypertable
Hbase
MongoDB
Terrastore
Scalaris
Berkeley DB
MemcacheDB
Redis
Partition Tolerance:
The system works well despite physical network partitions.

The problem

- 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 ?

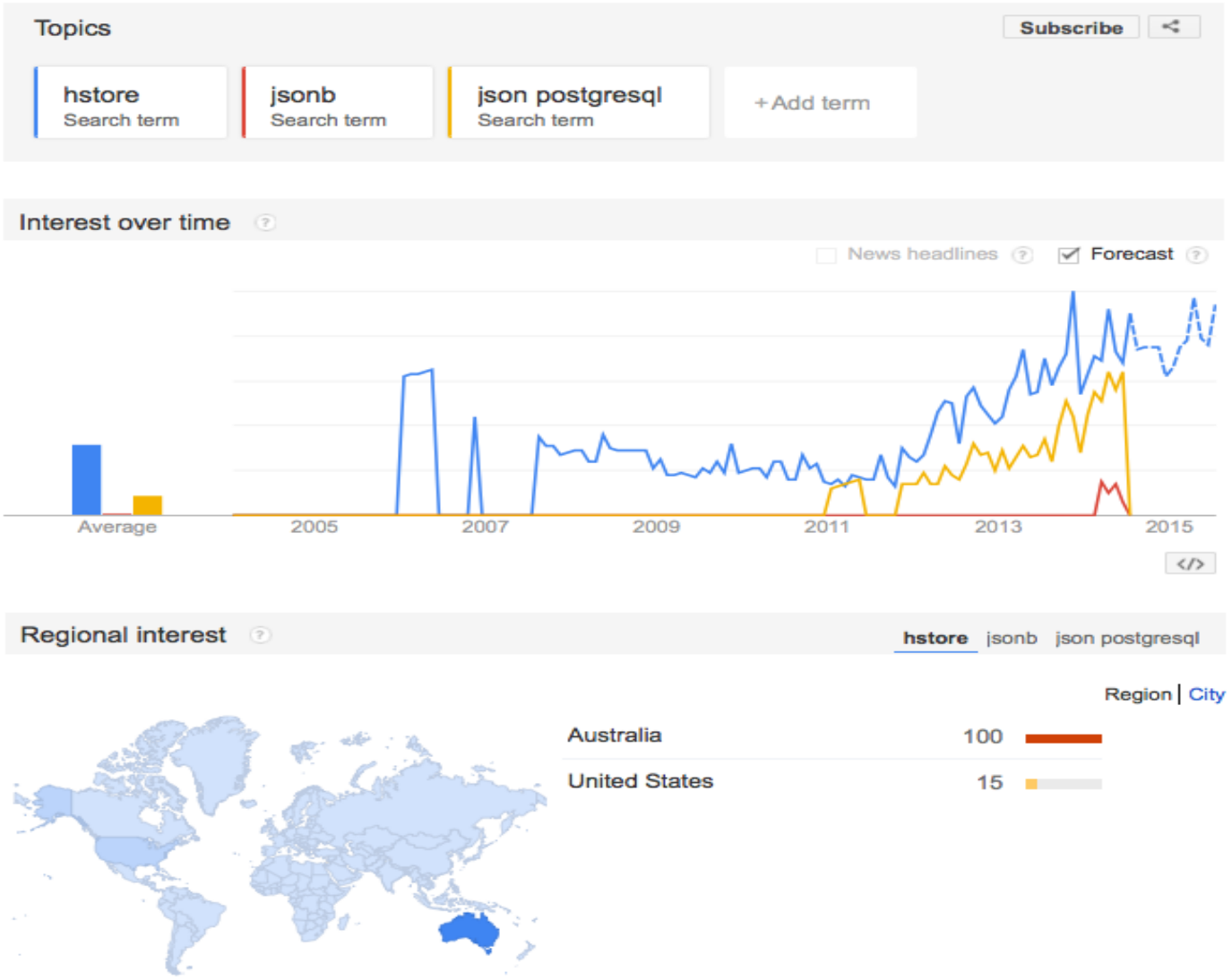
JSON in PostgreSQL

This is the challenge !

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 !

Google insights about hstore



Introduction to Hstore

id	col1	col2	col3	col4	col5	A lot of columns key1, …. keyN

- The problem:
- Total number of columns may be very large
- Only several fields are searchable (used in WHERE)
- Other columns are used only to output
  - These columns may not known in advance
- Solution
  - New data type (hstore), which consists of (key,value) pairs (a'la perl hash)

Introduction to Hstore

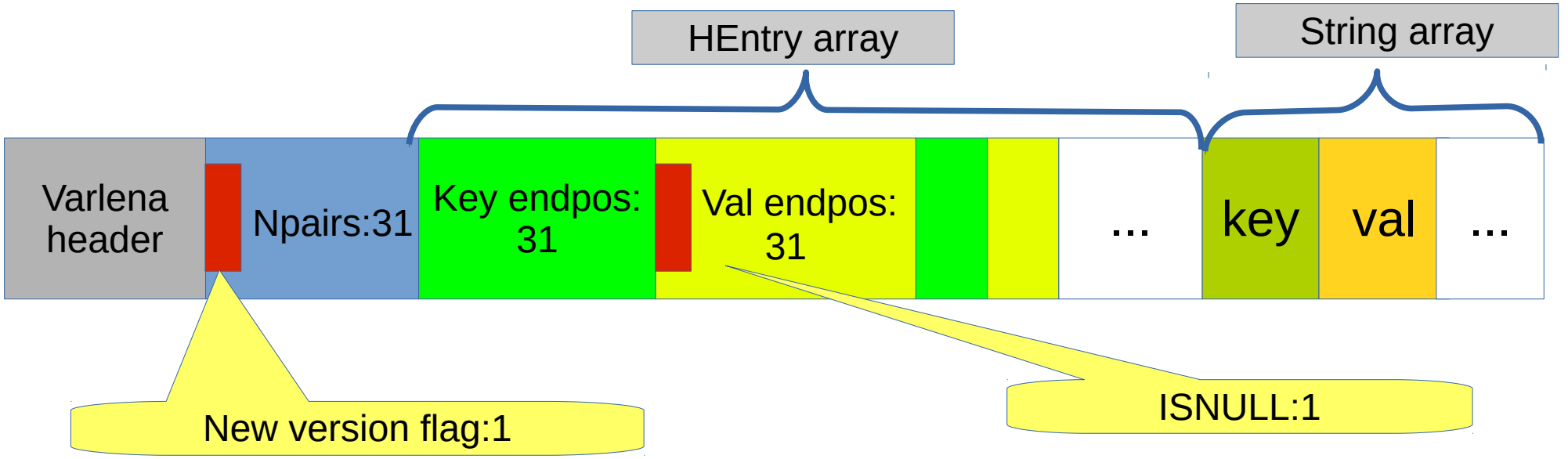
id	col1	col2	col3	col4	col5	**Hstore key1=>val1, key2=>val2,.....**

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

Introduction to hstore

- Hstore — key/value binary storage (perl hash)
  `'a=>1, b=>2'::hstore`
  - Key, value — strings
  - Get value for a key: hstore -> text
  - Operators with indexing support (GiST, GIN)
    Check for key: hstore ? text
    Contains: hstore @> hstore
  - check documentations for more
  - Functions for hstore manipulations (akeys, avals, skeys, svals, each,.....)
- Hstore provides PostgreSQL schema-less feature !
  - Faster releases, no problem with schema upgrade

Hstore binary storage



	Start	End
First key	0	HEntry[0]
i-th key	HEntry[i*2 - 1]	HEntry[i*2]
i-th value	HEntry[i*2]	HEntry[i*2 + 1]

Pairs are lexicographically ordered by key

History of hstore development

- May 16, 2003 — first version of hstore

```
Date: Fri, 16 May 2003 22:56:14 +0400
From: Teodor Sigaev <teodor@sigaev.ru>
To: Oleg Bartunov <oleg@sai.msu.su>, Alexey Slynko <slynko@tronet.ru>
Cc: E.Rodichev <er@sai.msu.su>
Subject: hash type (hstore)

Готова первайа версия:
zeus:~teodor/hstore.tgz

README написать не успел, поэтому здесь:
1 i/o типа hstore
2 операция hstore->text - извлечение значения по ключу text
select 'a=>q, b=>g'->'a';
 ?
------
 q

3 isexists(hstore), isdefined(hstore), delete(hstore,text) - полный перловый аналог
4 hstore || hstore - конкатенация, аналог в перле %a=( %b, %c );
5 text=>text - возвращает hstore
select 'a'=>'b';
 ?column?
----------
 "a"=>"b"


Все примеры есть в sql/hstore.sql
```

History of hstore development

- May 16, 2003 - first (unpublished) version of hstore for PostgreSQL 7.3
- Dec, 05, 2006 - hstore is a part of PostgreSQL 8.2

  (thanks, Hubert Depesz Lubaczewski!)
- May 23, 2007 - GIN index for hstore, PostgreSQL 8.3
- Sep, 20, 2010 - Andrew Gierth improved hstore, PostgreSQL 9.0

Introduction to hstore

- Hstore benefits
  - In provides a flexible model for storing a semi-structured data in relational database
  - hstore has binary storage and rich set of operators and functions, indexes
- Hstore drawbacks
  - Too simple model !
    Hstore key-value model doesn't supports tree-like structures as json (introduced in 2006, 3 years after hstore)
- Json — popular and standartized (ECMA-404 The JSON Data Interchange Standard, JSON RFC-7159)
- Json — PostgreSQL 9.2, textual storage

Hstore vs Json

- hstore is faster than json even on simple data

```
CREATE TABLE hstore_test AS (SELECT
'a=>1, b=>2, c=>3, d=>4, e=>5'::hstore AS v
FROM generate_series(1,1000000));

CREATE TABLE json_test AS (SELECT
'{"a":1, "b":2, "c":3, "d":4, "e":5}'::json AS v
FROM generate_series(1,1000000));
```

```
SELECT sum((v->'a')::text::int) FROM json_test;
851.012 ms

SELECT sum((v->'a')::int) FROM hstore_test;
330.027 ms
```

Hstore vs Json

- PostgreSQL already has json since 9.2, which supports document-based model, but
  - It's slow, since it has no binary representation and needs to be parsed every time
  - Hstore is fast, thanks to binary representation and index support
  - It's possible to convert hstore to json and vice versa, but current hstore is limited to key-value
  - Need hstore with document-based model. Share it's binary representation with json !

Nested hstore

abstract

Oleg Bartunov <obartunov@gmail.com> 12/18/12

to Teodor

Поправь, дополни.

Title: One step forward true json data type. Nested hstore with array support.

We present a prototype of nested hstore data type with array support. We consider the new hstore as a step forward true json data type.

Recently, PostgreSQL got json data type, which basically is a string storage with validity checking for stored values and some related functions. To be a real data type, it has to have a binary representation, which could be a big project if started from scratch. Hstore is a popular data type, we developed years ago to facilitate working with semi-structured data in PostgreSQL. Our idea is to extend hstore to be nested (value can be hstore) data type and add support of arrays, so its binary representation can be shared with json. We present a working prototype of a new hstore data type and discuss some design and implementation issues.

Nested hstore & jsonb

- Nested hstore at PGCon-2013, Ottawa, Canada (May 24) — thanks Engine Yard for support !

  One step forward true json data type.Nested hstore with arrays support
- Binary storage for nested data at PGCon Europe — 2013, Dublin, Ireland (Oct 29)

  Binary storage for nested data structuresand application to hstore data ty pe
- November, 2013 — binary storage was reworked, nested hstore and jsonb share the same storage. Andrew Dunstan joined the project.
- January, 2014 - binary storage moved to core

Nested hstore & jsonb

- Feb-Mar, 2014 - Peter Geoghegan joined the project, nested hstore was cancelled in favour to jsonb (Nested hstore patch for 9.3).
- Mar 23, 2014 Andrew Dunstan committed jsonb to 9.4 branch !

  pgsql: Introduce jsonb, a structured format for storing json.

Introduce jsonb, a structured format for storing json.

The new format accepts exactly the same data as the json type. However, it is stored in a format that does not require reparsing the orgiginal text in order to process it, making it much more suitable for indexing and other operations. Insignificant whitespace is discarded, and the order of object keys is not preserved. Neither are duplicate object keys kept - the later value for a given key is the only one stored.

Jsonb vs Json

```
SELECT '{"c":0,  "a":2,"a":1}'::json, '{"c":0,   "a":2,"a":1}'::jsonb;
         json           |      jsonb
------------------------+------------------
 {"c":0,  "a":2,"a":1} | {"a": 1, "c": 0}
(1 row)
```

- json: textual storage «as is»
- jsonb: no whitespaces
- jsonb: no duplicate keys, last key win
- jsonb: keys are sorted

Jsonb vs Json

- Data
  - 1,252,973 Delicious bookmar
- Server
  - MBA, 8 GB RAM, 256 GB SS
- Test
  - Input performance - copy data to table
  - Access performance - get value by key
  - Search performance contains @> operator


```
{
    "author": "mcasas1",
    "comments": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422",
    "guidislink": false,
    "id": "http://delicious.com/url/b5b3cbf9a9176fe43c27d7b4af94a422#mcasas1",
    "link": "http://www.theatermania.com/broadway/",
    "links": [
        {
            "href": "http://www.theatermania.com/broadway/",
            "rel": "alternate",
            "type": "text/html"
        }
    ],
    "source": {},
    "tags": [
        {
            "label": null,
            "scheme": "http://delicious.com/mcasas1/",
            "term": "NYC"
        }
    ],
    "title": "TheaterMania",
    "title_detail": {
        "base": "http://feeds.delicious.com/v2/rss/recent?min=1&count=100",
        "language": null,
        "type": "text/plain",
        "value": "TheaterMania"
    },
    "updated": "Tue, 08 Sep 2009 23:28:55 +0000",
    "wfw_commentrss": "http://feeds.delicious.com/v2/rss/url/b5b3cbf9a9176fe43c27d7b4af94a422"
}
```

Jsonb vs Json

- Data
  - 1,252,973 bookmarks from Delicious in json format (js)
  - The same bookmarks in jsonb format (jb)
  - The same bookmarks as text (tx)

```
=# \dt+
                      List of relations
 Schema | Name | Type  |  Owner   |  Size   | Description
--------+------+-------+----------+---------+-------------
 public | jb   | table | postgres | 1374 MB | overhead is < 4%
 public | js   | table | postgres | 1322 MB |
 public | tx   | table | postgres | 1322 MB |
```

Jsonb vs Json

- Input performance (parser)
  Copy data (1,252,973 rows) as text, json,jsonb

```
copy tt from  '/path/to/test.dump'
```

Text: 34 s - as is

Json: 37 s - json validation

Jsonb: 43 s - json validation, binary storage

Postgres PROFESSIONAL

Jsonb vs Json (binary storage)

- Access performance — get value by key
  - Base: SELECT js FROM js;
  - Jsonb: SELECT j->>'updated' FROM jb;
  - Json: SELECT j->>'updated' FROM js;

Base: 0.6 s

Jsonb: 1 s 0.4

Json: 9.6 s 9

Jsonb ~ 20X faster Json

Jsonb vs Json

```
EXPLAIN ANALYZE SELECt count(*) FROM js WHERE js #>>'{tags,0,term}' = 'NYC';
                                                     QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=187812.38..187812.39 rows=1 width=0)
(actual time=10054.602..10054.602 rows=1 loops=1)
   ->  Seq Scan on js  (cost=0.00..187796.88 rows=6201 width=0)
(actual time=0.030..10054.426 rows=123 loops=1)
         Filter: ((js #>> '{tags,0,term}'::text[]) = 'NYC'::text)
         Rows Removed by Filter: 1252850
 Planning time: 0.078 ms
 Execution runtime: 10054.635 ms
(6 rows)
```

Json: no contains @> operator,
search first array element

Jsonb vs Json (binary storage)

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE
   jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=191521.30..191521.31 rows=1 width=0)
(actual time=1263.201..1263.201 rows=1 loops=1)
   ->  Seq Scan on jb  (cost=0.00..191518.16 rows=1253 width=0)
 (actual time=0.007..1263.065 rows=285 loops=1)
         Filter: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Rows Removed by Filter: 1252688
 Planning time: 0.065 ms
 Execution runtime: 1263.225 ms          Execution runtime: 10054.635 ms
(6 rows)
```

Jsonb ~ 10X faster Json

Jsonb vs Json (GIN: key && value)

CREATE INDEX gin_jb_idx ON jb USING gin(jb);

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE
          jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=4772.72..4772.73 rows=1 width=0)
(actual time=8.486..8.486 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb  (cost=73.71..4769.59 rows=1253 width=0)
(actual time=8.049..8.462 rows=285 loops=1)
         Recheck Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on gin_jb_idx  (cost=0.00..73.40 rows=1253 width=0)
(actual time=8.014..8.014 rows=285 loops=1)
               Index Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
 Planning time: 0.115 ms
 Execution runtime: 8.515 ms           Execution runtime: 10054.635 ms
(8 rows)
```

Jsonb ~ 150X faster Json

Jsonb vs Json (GIN: hashpath.value)

CREATE INDEX gin_jb_path_idx ON jb USING gin(jb jsonb_path_ops);

```
EXPLAIN ANALYZE SELECT count(*) FROM jb WHERE
        jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                 QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=4732.72..4732.73 rows=1 width=0)
(actual time=0.644..0.644 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb  (cost=33.71..4729.59 rows=1253 width=0)
(actual time=0.102..0.620 rows=285 loops=1)
         Recheck Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on gin_jb_path_idx
(cost=0.00..33.40 rows=1253 width=0) (actual time=0.062..0.062 rows=285 loops=1)
               Index Cond: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
 Planning time: 0.056 ms
 Execution runtime: 0.668 ms       Execution runtime: 10054.635 ms
(8 rows)
```

Jsonb ~ 1800X faster Json

PostgreSQL 9.4 vs Mongo 2.6.0

- Operator 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

Features of Jsonb

- Contains operators - jsonb @> jsonb, jsonb <@ jsonb (GIN indexes)

```
jb @> '{"tags":[{"term":"NYC"}]}'::jsonb
```

  Keys should be specified from root

- Equivalence operator — jsonb = jsonb (GIN indexes)
- Exists operators — jsonb ? text, jsonb ?! text[], jsonb ?& text[] (GIN indexes)

```
jb WHERE jb ?| '{tags,links}'
```

  Only root keys supported

- Operators on jsonb parts (functional indexes)

```
SELECT ('{"a": {"b":5}}'::jsonb -> 'a'->>'b')::int > 2;
CREATE INDEX ....USING BTREE ( (jb->'a'->>'b')::int);
```

  Very cumbersome, too many functional indexes

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

- ```
  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;
```
```

Query

- Need Jsonb query language
  - Simple and effective way to search in arrays (and other iterative searches)
  - More comparison operators (currently only =)
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support

Postgres PROFESSIONAL

jsquery

```
Expr     ::= path value_expr
           | path HINT value_expr
           | NOT expr
           | NOT HINT value_expr
           | NOT value_expr
           | path '(' expr ')'
           | '(' expr ')'
           | expr AND expr
           | expr OR expr
```

```
value_expr
          ::= '=' scalar_value
            | IN '(' value_list ')'
            | '=' array
            | '=' '*'
            | '<' NUMERIC
            | '<' '=' NUMERIC
            | '>' NUMERIC
            | '>' '=' NUMERIC
            | '@' '>' array
            | '<' '@' array
            | '&' '&' array
            | IS ARRAY
            | IS NUMERIC
            | IS OBJECT
            | IS STRING
            | IS BOOLEAN
```

```
path      ::= key
            | path '.' key_any
            | NOT '.' key_any

key       ::= '*'
            | '#'
            | '%'
            | '$'
            | STRING
          ……..

key_any       ::= key
            | NOT
```

Jsquery

```
path     ::= key
           | path '.' key_any
           | NOT '.' key_any

key      ::= '*'
           | '#'
           | '%'
           | '$'
           | STRING
          ……..

key_any     ::= key
           | NOT
```

- # - any element array

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# = 2';
```

- % - any key

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ '%.b.# = 2';
```

- * - anything

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@ '*.# = 2';
```

- $ - current element

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# ($ = 2 OR $ < 3)';
```

- Use "double quotes" for key !

```
select 'a1."12222" < 111'::jsquery;
```

«#», «*», «%» usage rules

Each usage of «#», «*», «%» means separate element

- Find companies where CEO or CTO is called Neil.

```
SELECT count(*) FROM company WHERE
     js @@ 'relationships.#(title in ("CEO", "CTO") AND
                  person.first_name = "Neil")'::jsquery;
 count
-------
    12
```

- Find companies with some CEO or CTO and someone called Neil

```
SELECT count(*) FROM company WHERE
     js @@ 'relationships(#.title in ("CEO", "CTO") AND
                  #.person.first_name = "Neil")'::jsquery;
 count
-------
    69
```

Jsquery

- Scalar

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.# IN (1,2,5)';
```

- Test for key existence

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b = *';
```

- Array overlap

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b && [1,2,5]';
```

- Array contains

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b @> [1,2]';
```

- Array contained

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b <@ [1,2,3,4,5]';
```

Jsquery

- Type checking

```
select '{"x": true}' @@ 'x IS boolean'::jsquery,
       '{"x": 0.1}'  @@ 'x IS numeric'::jsquery;
 ?column? | ?column?
----------+----------
 t        | t
```

```
select '{"a":{"a":1}}' @@ 'a IS object'::jsquery;
 ?column?
----------
 t
```

```
select '{"a":["xxx"]}' @@ 'a IS array'::jsquery,
       '["xxx"]' @@ '$ IS array'::jsquery;
 ?column? | ?column?
----------+----------
 t        | t
```

IS BOOLEAN

IS NUMERIC

IS ARRAY

IS OBJECT

IS STRING

PROFESSIONAL Postgres

Jsquery

- How many products are similar to "B000089778" and have product_sales_rank in range between 10000-20000 ?
- SQL

```
SELECT count(*) FROM jr WHERE (jr->>'product_sales_rank')::int > 10000 and
 (jr->> 'product_sales_rank')::int < 20000 and
 ....boring stuff
```

- Jsquery

```
SELECT count(*) FROM jr WHERE jr @@ ' similar_product_ids && ["B000089778"] AND
product_sales_rank( $ > 10000 AND $ < 20000)'
```

- Mongodb

```
db.reviews.find( { $and :[ {similar_product_ids: { $in ["B000089778"]}},
{product_sales_rank:{$gt:10000, $lt:20000}}] } ).count()
```

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"';
```

Example

- SQL

```
SELECT * FROM js_test2 js
WHERE NOT EXISTS (
  SELECT 1
  FROM
jsonb_array_elements(js.value) el
  WHERE EXISTS (
    SELECT 1
    FROM jsonb_each(el.value) kv
    WHERE NOT
kv.value::text::numeric BETWEEN
0.0 AND 1.0));
```

Jsquery

```
SELECT * FROM js_test2
js
WHERE value @@
'#:.%:($ >= 0 AND $ <= 1)';
```

PROFESSIONAL Postgres

Jsquery

```
explain( analyze, buffers) select count(*) from jb where jb @> '{"tags":[{"term":"NYC"}]}'::jsonb;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=191517.30..191517.31 rows=1 width=0) (actual time=1039.422..1039.423 rows=1 loops=1)
   Buffers: shared hit=97841 read=78011
   ->  Seq Scan on jb  (cost=0.00..191514.16 rows=1253 width=0) (actual time=0.006..1039.310 rows=285 loops=1)
         Filter: (jb @> '{"tags": [{"term": "NYC"}]}'::jsonb)
         Rows Removed by Filter: 1252688
         Buffers: shared hit=97841 read=78011
 Planning time: 0.074 ms
 Execution time: 1039.444 ms

explain( analyze,costs off) select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
                        QUERY PLAN
-----------------------------------------------------------------------
 Aggregate (actual time=891.707..891.707 rows=1 loops=1)
   ->  Seq Scan on jb (actual time=0.010..891.553 rows=285 loops=1)
         Filter: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
         Rows Removed by Filter: 1252688
 Execution time: 891.745 ms
```

Jsquery (indexes)

- GIN opclasses with jsquery support
  - jsonb_value_path_ops — use Bloom filtering for key matching
    {"a":{"b":{"c":10}}} → 10.(bloom(a) or bloom(b) or bloom(c))
    - Good for key matching (wildcard support) , not good for range query
  - jsonb_path_value_ops — hash path (like jsonb_path_ops)
    {"a":{"b":{"c":10}}} → hash(a.b.c).10
    - No wildcard support, no problem with ranges

```
                                List of relations
 Schema |          Name             | Type  |  Owner   |    Table      |  Size   | Description
--------+---------------------------+-------+----------+---------------+---------+-------------
 public | jb                        | table | postgres |               | 1374 MB |
 public | jb_value_path_idx         | index | postgres | jb            | 306 MB  |
 public | jb_gin_idx                | index | postgres | jb            | 544 MB  |
 public | jb_path_value_idx         | index | postgres | jb            | 306 MB  |
 public | jb_path_idx               | index | postgres | jb            | 251 MB  |
```

PROFESSIONAL Postgres

Jsquery (indexes)

```
explain( analyze,costs off) select count(*) from jb where
                                                jb @@ 'tags.#.term = "NYC"';
                                QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate (actual time=0.609..0.609 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb (actual time=0.115..0.580 rows=285 loops=1)
         Recheck Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on jb_value_path_idx (actual time=0.073..0.073
                                                           rows=285 loops=1)
               Index Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
 Execution time: 0.634 ms
(7 rows)
```

Jsquery (indexes)

```
explain( analyze,costs off) select count(*) from jb where jb @@ '*.term = "NYC"';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=0.688..0.688 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb (actual time=0.145..0.660 rows=285 loops=1)
         Recheck Cond: (jb @@ '*."term" = "NYC"'::jsquery)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on jb_value_path_idx (actual time=0.113..0.113
                                                     rows=285 loops=1)
               Index Cond: (jb @@ '*."term" = "NYC"'::jsquery)
 Execution time: 0.716 ms
(7 rows)
```

Jsquery (indexes)

```
explain( analyze,costs off) select count(*) from jb where jb @@ '*.term = "NYC"';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=0.688..0.688 rows=1 loops=1)
   ->  Bitmap Heap Scan on jb (actual time=0.145..0.660 rows=285 loops=1)
         Recheck Cond: (jb @@ '*."term" = "NYC"'::jsquery)
         Heap Blocks: exact=285
         ->  Bitmap Index Scan on jb_value_path_idx (actual time=0.113..0.113
                                                     rows=285 loops=1)
               Index Cond: (jb @@ '*."term" = "NYC"'::jsquery)
 Execution time: 0.716 ms
(7 rows)
```

Citus dataset


```
{
  "customer_id": "AE22YDHSBFYIP",
  "product_category": "Business & Investing",
  "product_group": "Book",
  "product_id": "1551803542",
  "product_sales_rank": 11611,
  "product_subcategory": "General",
  "product_title": "Start and Run a Coffee Bar (Start & Run a)",
  "review_date": {
    "$date": 31363200000
  },
  "review_helpful_votes": 0,
  "review_rating": 5,
  "review_votes": 10,
  "similar_product_ids": [
    "0471136174",
    "0910627312",
    "047112138X",
    "0786883561",
    "0201570483"
  ]
}
```


- 3023162 reviews from Citus 1998-2000 years
- 1573 MB

Jsquery (indexes)

```
explain (analyze, costs off) select count(*) from jr where
                                jr @@  ' similar_product_ids && ["B000089778"]';
                                 QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=0.359..0.359 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.084..0.337 rows=185 loops=1)
         Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=0.057..0.057
                                                     rows=185 loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.394 ms
(7 rows)
```

Jsquery (indexes)

- No statistics, no planning :(

Not selective, better not use index!

```
explain (analyze, costs off) select count(*) from jr where
 jr @@ ' similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)';
                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=126.149..126.149 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=126.057..126.143 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] &
 "product_sales_rank"($ > 10000 & $ < 20000))'::jsquery)
         Heap Blocks: exact=45
         ->  Bitmap Index Scan on jr_path_value_idx
                       (actual time=126.029..126.029 rows=45 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] &
"product_sales_rank"($ > 10000 & $ < 20000))'::jsquery)
 Execution time: 129.309 ms !!!   No statistics
(7 rows)
```

MongoDB 2.6.0

```
db.reviews.find(  {  $and :[ {similar_product_ids: { $in:["B000089778"]}},
    {product_sales_rank:{$gt:10000, $lt:20000}}] } )
.explain()
{
      "n" : 45,
       ……..................
      "millis" : 7,
      "indexBounds" : {
            "similar_product_ids" : [       index size = 400 MB just for similar_product_ids !!!
                  [
                        "B000089778",
                        "B000089778"
                  ]
            ]
      },
}
```

Jsquery (indexes)

- If we rewrite query and use planner

```
explain (analyze,costs off) select count(*) from jr where
jr @@ ' similar_product_ids && ["B000089778"]'
and  (jr->>'product_sales_rank')::int>10000 and
      (jr->>'product_sales_rank')::int<20000;
------------------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=0.479..0.479 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.079..0.472 rows=45 loops=1)
         Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
         Filter: ((((jr ->> 'product_sales_rank'::text))::integer > 10000) AND
(((jr ->> 'product_sales_rank'::text))::integer < 20000))
         Rows Removed by Filter: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx
                        (actual time=0.041..0.041 rows=185 loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.506 ms   Potentially,  query could be faster Mongo !
(9 rows)
```

Jsquery (optimiser)

- Jsquery now has built-in optimiser for simple queries.

```
explain (analyze, costs off) select count(*) from jr where
jr @@ 'similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)'

-----------------------------------------------------------------------------------------------

 Aggregate (actual time=0.422..0.422 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.099..0.416 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
         Rows Removed by Index Recheck: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx
```

- (actual time=0.060..0.060 rows=185 loops=1)
  Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)

Execution time: **0.480 ms vs 7 ms MongoDB !**

Jsquery (optimiser)

- Jsquery now has built-in optimiser for simple queries.

  Analyze query tree and push non-selective parts to recheck (like filter)

  Selectivity classes:

  1) Equality (x = c)
  2) Range (c1 < x < c2)
  3) Inequality (c > c1)
  4) Is (x is type)
  5) Any (x = *)

```
SELECT gin_debug_query_path_value('similar_product_ids && ["B000089778"]
                        AND product_sales_rank( $ > 10000 AND $  < 20000)');
            gin_debug_query_path_value
-----------------------------------------------
 similar_product_ids.# = "B000089778" , entry 0 +
```

Jsquery (optimiser)

- Jsquery optimiser pushes non-selective operators to recheck

```
explain (analyze, costs off) select count(*) from jr where
jr @@ 'similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)'

-----------------------------------------------------------------------------------------------------------

 Aggregate (actual time=0.422..0.422 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.099..0.416 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000)'::jsquery)
         Rows Removed by Index Recheck: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx
```

-
```
                  (actual time=0.060..0.060 rows=185 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000)'::jsquery)
 Execution time: 0.480 ms
```

Jsquery (HINTING)

- Jsquery now has HINTING (if you don't like optimiser)!

```
explain (analyze, costs off) select count(*) from jr where
```

-
```
                                jr @@ 'product_sales_rank > 10000'
-------------------------------------------------------------------------------------------------------
 Aggregate (actual time=2507.410..2507.410 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=1118.814..2352.286 rows=2373140 loops=1)
         Recheck Cond: (jr @@ '"product_sales_rank" > 10000'::jsquery)
         Heap Blocks: exact=201209
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=1052.483..1052.48
rows=2373140 loops=1)
               Index Cond: (jr @@ '"product_sales_rank" > 10000'::jsquery)
 Execution time: 2524.951 ms
```

- Better not to use index — HINT /* --noindex */

```
explain (analyze, costs off) select count(*) from jr where
```

-
```
                                jr @@ 'product_sales_rank /*-- noindex */ > 10000';
-------------------------------------------------------------------------------
 Aggregate (actual time=1376.262..1376.262 rows=1 loops=1)
   ->  Seq Scan on jr (actual time=0.013..1222.123 rows=2373140 loops=1)
         Filter: (jr @@ '"product_sales_rank" /*-- noindex */  > 10000'::jsquery)
         Rows Removed by Filter: 650022
 Execution time: 1376.284 ms
```

Jsquery use case: schema specification

```
CREATE TABLE js (
    id serial primary key,
    v jsonb,
    CHECK(v @@ 'name IS STRING AND
                coords IS ARRAY AND
                coords.#: ( NOT (
                    x IS NUMERIC AND
                    y IS NUMERIC ) )'::jsquery));
```

Jsquery use case: schema specification

```
# INSERT INTO js (v) VALUES
  ('{"name": "abc", "coords": [{"x": 1, "y": 2}, {"x": 3, "y": 4}]}');
INSERT 0 1
```

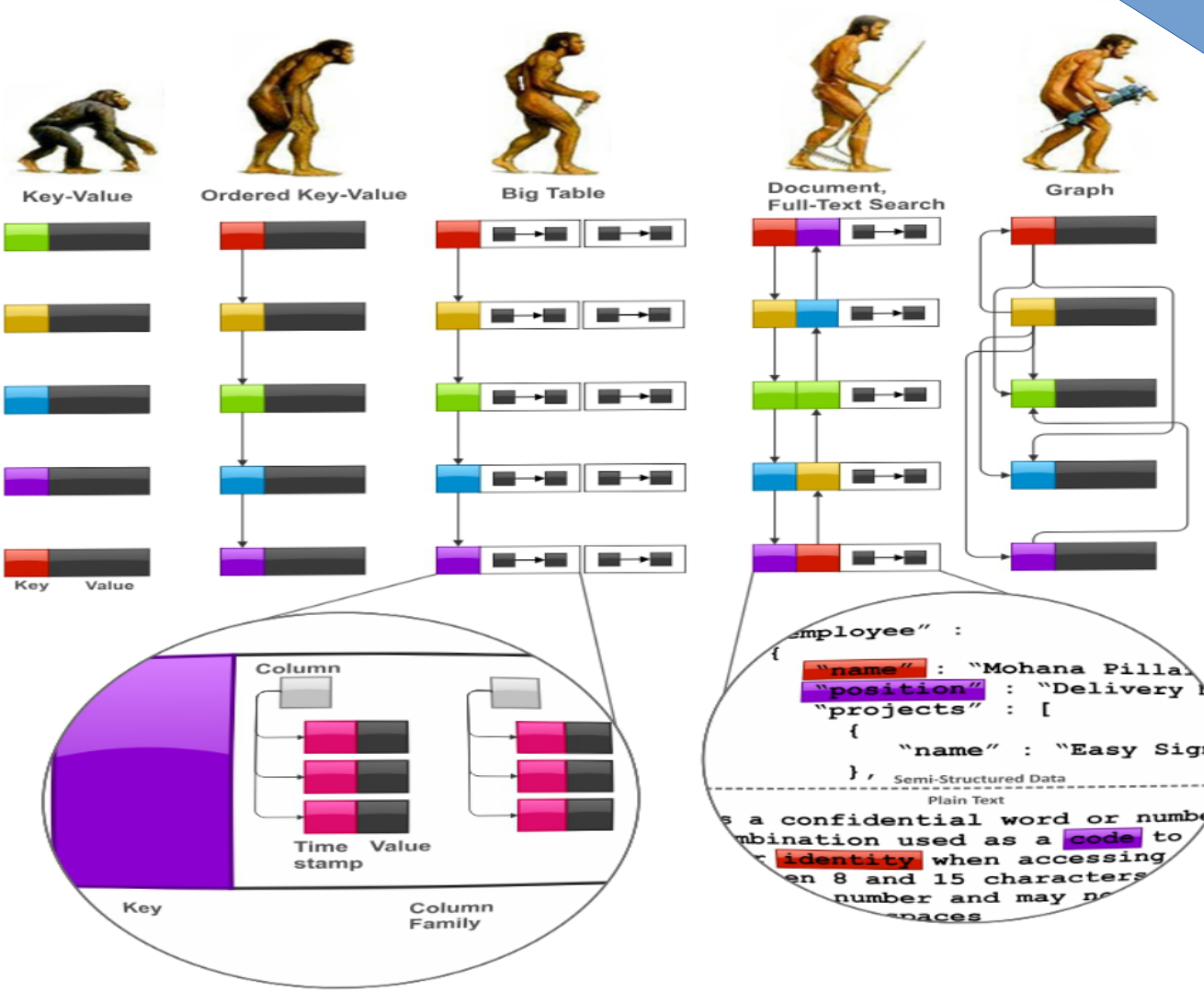
- ```
  # INSERT INTO js (v) VALUES
    ('{"name": 1, "coords": [{"x": 1, "y": 2}, {"x": "3", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint "js_v_check"
  ```

- ```
  # INSERT INTO js (v) VALUES
    ('{"name": "abc", "coords": [{"x": 1, "y": 2}, {"x": "zzz", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint "js_v_check"
  ```

Contrib/jsquery

- Jsquery index support is quite efficient
  (0.5 ms vs Mongo 7 ms !)
- Future direction
  - Make jsquery planner friendly
  - Need statistics for jsonb
- Availability
  - Jsquery + opclasses are available as extensions
  - Grab it from https://github.com/akorotkov/jsquery
    (branch master) , we need your feedback !

Stop following me !



PostgreSQL 9.4+

- Open-source
- Relational database
- Strong support of json

Jsonb querying: summary

Using «@>»

- Pro
  - Indexing support
- Cons
  - Checks only equality for scalars
  - Hard to explain complex logic

Using subselect and jsonb_array_elements

- Pro
  - SQL-rich
- Cons
  - No indexing support
  - Heavy syntax

JsQuery

- Pro
  - Indexing support
  - Rich enough for typical applications
- Cons
  - Not extendable

Still looking for a better solution!

Querying problem isn't new!

We already have similar problems with:

- Arrays
- Hstore
- FTS

How to search arrays by their elements?

```
Search for arrays overlapping {10,20}.
# SELECT * FROM t WHERE 10 = ANY(a) OR
                                        20 = ANY(a);
 Seq Scan on t
   Filter: ((10 = ANY (a)) OR (20 = ANY (a)))

Search for arrays containing {10,20}.
# SELECT * FROM t WHERE 10 = ANY(a) AND
                                        20 = ANY(a);
 Seq Scan on t
   Filter: ((10 = ANY (a)) AND (20 = ANY (a)))
```

No index - hard luck :(

How to make it use index?

```
Overlaps «&&» operator.
# SELECT * FROM t WHERE a && '{10,20}'::int[];
 Bitmap Heap Scan on t
   Recheck Cond: (a && '{10,20}'::integer[])
   ->  Bitmap Index Scan on t_idx
         Index Cond: (a && '{10,20}'::integer[])

Contains «@>» operator.
# SELECT * FROM t WHERE a @> '{10,20}'::int[];
 Bitmap Heap Scan on t
   Recheck Cond: (a @> '{10,20}'::integer[])
   ->  Bitmap Index Scan on t_idx
         Index Cond: (a @> '{10,20}'::integer[])
```

Why do we need search operators for arrays?

Currently PostgreSQL planner can use index for:

- WHERE col op value
- ORDER BY col
- ORDER BY col op value (KNN)

We had to introduce array operators to use index.

What can't be expressed by array operators?

Search for array elements greater than 10.

```
SELECT * FROM t WHERE 10 < ANY(a);
```

Search for array elements between 10 and 20.

```
SELECT * FROM t WHERE EXISTS (
    SELECT *
    FROM unnest(a) e
    WHERE e BETWEEN 10 AND 20);
```

Why GIN isn't used?

- Planner limitations: every search clause must be expressed as a single operator,
  no complex expressions!
- Current GIN implementation has no support for such queries
  - **DOABLE!**

Jsonb indexing: cumulative problems of arrays and jsonb

Expression	using gin			using btree ((js->'key'))
	default	path	jsquery	
js @> '{"key": ["value"]}'	+	+	-	–
js->'key' = 'value'	–	–	-	+
js->'key' > 'value'	–	–	-	+
js @@ 'key.# = "value""'	–	–	+	–
Subselects	–	–	–	–

Jsonb querying

@> operator

- Pro
  - Indexing support
- Cons
  - Checks only equality for scalars
  - Hard to explain complex logic

Using subselects

- Pro
  - Support all SQL features
- Cons
  - **No indexing support**
  - **Heavy syntax**

JsQuery

- Pro
  - Rich enough for typical applications
  - Indexing support
- Cons
  - Not extendable

It would be nice to workout these two!

Better syntax: «anyelement» feature

```
{ ANY | EACH }
{ ELEMENT | KEY | VALUE | VALUE ANYWHERE }
 OF container AS alias SATISFIES (expression)
```

https://github.com/postgrespro/postgres/tree/anyelement

PROFESSIONAL Postgres

Examples

- There is element divisible by 3 in array

```
SELECT * FROM array_test WHERE
ANY ELEMENT OF a AS e SATISFIES (e % 3 = 0);
```

- Each element of array is between 1 and 10

```
SELECT * FROM array_test WHERE
EACH ELEMENT OF a AS e SATISFIES (e BETWEEN 1 AND 10);
```

Postgres PROFESSIONAL

Examples

- All the scalars in jsonb are numbers

```
SELECT * FROM jsonb_test WHERE
EACH VALUE ANYWHERE OF j AS e SATISFIES
(jsonb_typeof(e) = 'number');
```

- There is at least one object in jsonb array

```
SELECT * FROM jsonb_test WHERE
ANY ELEMENT OF j AS e SATISFIES
(jsonb_typeof(e) = 'object');
```

PROFESSIONAL Postgres

Examples

- Find companies where CEO or CTO is called Neil.

```
SELECT * FROM companies
WHERE ANY ELEMENT OF c->'department' AS d
SATISFIES (
    ANY ELEMENT OF d->'staff' AS s SATISFIES (
        s->>'name' = 'Neil' AND
        s->>'post' IN ('CEO', 'CTO'))
);
```

PROFESSIONAL Postgres

Examples

- Find companies where exists department with all salaries greater than 1000.

```
SELECT * FROM companies WHERE
ANY ELEMENT OF c ->'department' AS d
SATISFIES (
    EACH ELEMENT OF d->'staff' AS s
        SATISFIES (
           (s->>'salary')::numeric > 1000
        )
);
```

Just a syntax sugar

```
CREATE VIEW v AS (SELECT * FROM (SELECT '{1,2,3}'::int[] a) t
WHERE ANY ELEMENT OF t.a AS e SATISFIES (e >= 1));

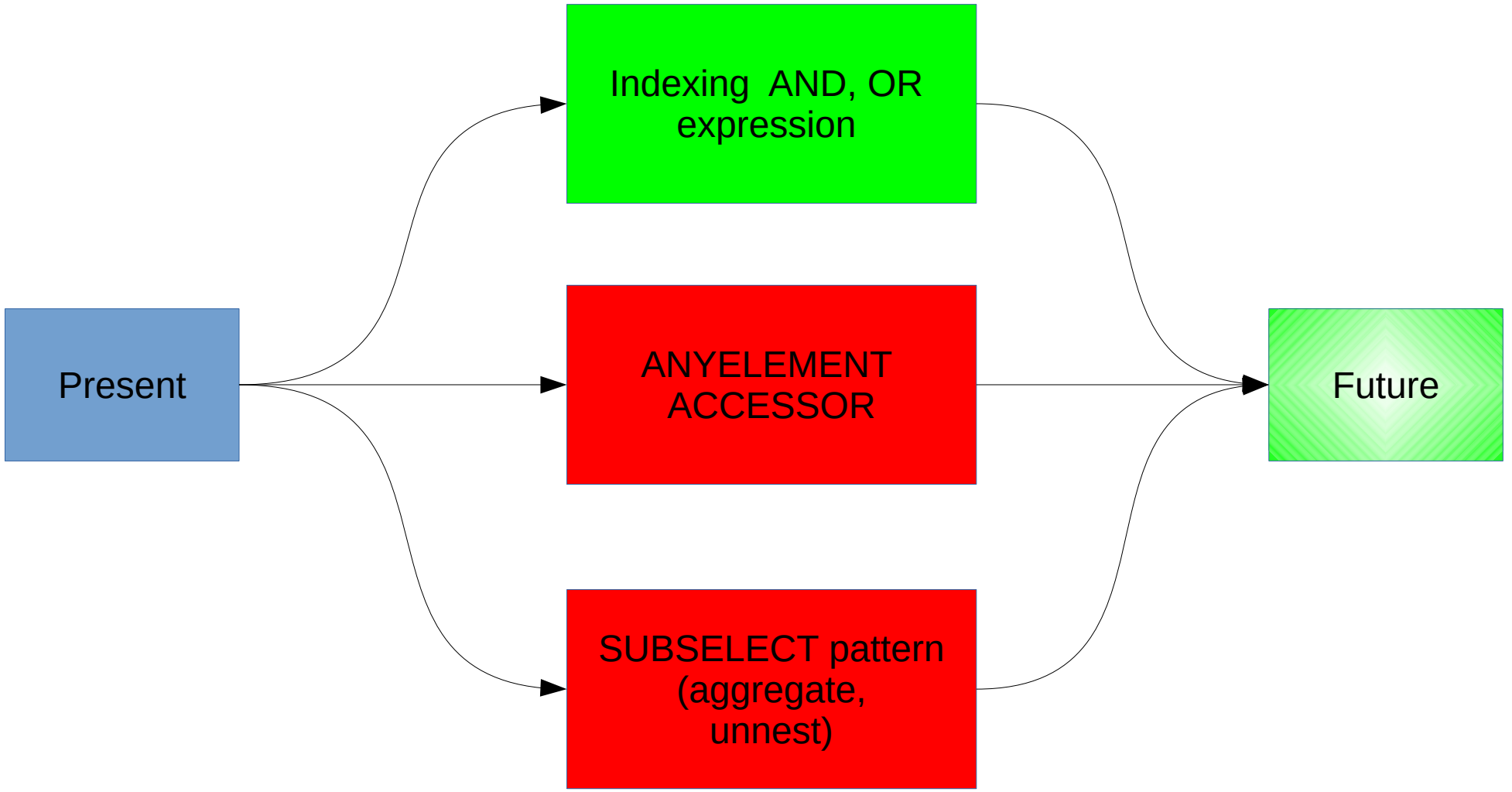
View definition:
 SELECT t.a
   FROM ( SELECT '{1,2,3}'::integer[] AS a) t
  WHERE ( SELECT
                CASE
                    WHEN count(*) = 0 AND t.a IS NOT NULL THEN false
                    ELSE bool_or_not_null(e.e >= 1)
                END AS bool_or_not_null
           FROM unnest_element(t.a, false) e(e));
```

Postgres PROFESSIONAL

Subselect?!

- USE INDEX!

What do we need?



Now

- Search keys of index:
- «An index scan has zero or more scan keys, which are implicitly ANDed»

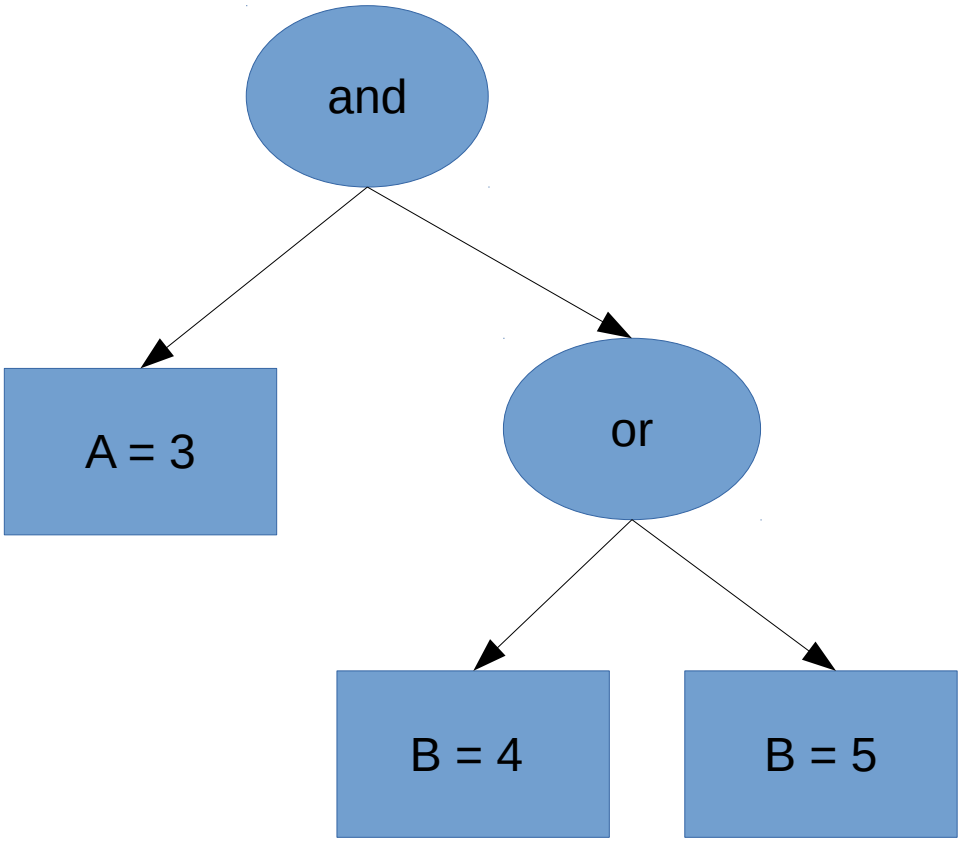
Closed future (patch exists)

- Polish notation for simple stack machine:
- (a=3)(b=4)(b=5)(OR)(AND)
- (b=4)(b=5)(OR)(a=3)(AND)
-
- Right now:

• GiST
• GIN
• BRIN

Also possible:

• Btree (difficult)
• SP-GiST
• Hash



GIN vs GIN

```
# EXPLAIN ANALYZE
SELECT count(*) FROM tbl WHERE (a && '{1}' OR a && '{2}') AND
                               (a && '{99901}' OR a && '{99902}');
                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=4503.28..4503.29 rows=1 width=0) (actual time=0.546..0.547 rows=1 loops=1)
   ->  Bitmap Heap Scan on tbl  (cost=522.79..4500.61 rows=1065 width=0) (actual time=0.541..0.542 rows=1 loops=1)
         Recheck Cond: (((a && '{99901}'::integer[]) OR (a && '{99902}'::integer[])) AND ((a && '{1}'::integer[]) OR (a && '{2}'::integer[])))
         Heap Blocks: exact=1
         ->  Bitmap Index Scan on idx  (cost=0.00..522.79 rows=1097 width=0) (actual time=0.537..0.537 rows=1 loops=1)
               Index Cond: (((a && '{99901}'::integer[]) OR (a && '{99902}'::integer[])) AND ((a && '{1}'::integer[]) OR (a && '{2}'::integer[])))
 Planning time: 0.151 ms
 Execution time: 0.603 ms
(8 rows)
                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=5139.85..5139.86 rows=1 width=0) (actual time=19.212..19.212 rows=1 loops=1)
   ->  Bitmap Heap Scan on tbl  (cost=1159.37..5137.19 rows=1065 width=0) (actual time=19.174..19.209 rows=1 loops=1)
         Recheck Cond: (((a && '{99901}'::integer[]) OR (a && '{99902}'::integer[])) AND ((a && '{1}'::integer[]) OR (a && '{2}'::integer[])))
         Rows Removed by Index Recheck: 2
         Heap Blocks: exact=3
         ->  BitmapAnd  (cost=1159.37..1159.37 rows=1097 width=0) (actual time=19.118..19.118 rows=0 loops=1)
               ->  BitmapOr  (cost=131.49..131.49 rows=9995 width=0) (actual time=0.023..0.023 rows=0 loops=1)
                     ->  Bitmap Index Scan on idx  (cost=0.00..65.48 rows=4998 width=0) (actual time=0.017..0.017 rows=3 loops=1)
                           Index Cond: (a && '{99901}'::integer[])
                     ->  Bitmap Index Scan on idx  (cost=0.00..65.48 rows=4998 width=0) (actual time=0.005..0.005 rows=2 loops=1)
                           Index Cond: (a && '{99902}'::integer[])
               ->  BitmapOr  (cost=1027.62..1027.62 rows=109745 width=0) (actual time=18.948..18.948 rows=0 loops=1)
                     ->  Bitmap Index Scan on idx  (cost=0.00..522.79 rows=55839 width=0) (actual time=10.593..10.593 rows=56902 loops=1)
                           Index Cond: (a && '{1}'::integer[])
                     ->  Bitmap Index Scan on idx  (cost=0.00..504.30 rows=53906 width=0) (actual time=8.354..8.354 rows=53541 loops=1)
                           Index Cond: (a && '{2}'::integer[])
 Planning time: 0.141 ms
 Execution time: 19.274 ms
```

GIN vs BTREE

```
EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=10000004782.31..10000004782.32 rows=1 width=0) (actual time=0.279..0.279 rows=1 loops=1)
   ->  Bitmap Heap Scan on tst  (cost=10000000057.50..10000004745.00 rows=14925 width=0) (actual
time=0.080..0.267 rows=173 loops=1)
         Recheck Cond: ((id = 5) OR (id = 500) OR (id = 5000))
         Heap Blocks: exact=172
         ->  Bitmap Index Scan on idx_gin  (cost=0.00..57.50 rows=15000 width=0) (actual time=0.059..0. 059 rows=147
loops=1)
               Index Cond: ((id = 5) OR (id = 500) OR (id = 5000))
 Planning time: 0.077 ms
 Execution time: 0.308 ms
 EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=51180.53..51180.54 rows=1 width=0) (actual time=796.766..796.766 rows=1 loops=1)
   ->  Index Only Scan using idx_btree on tst  (cost=0.42..51180.40 rows=55 width=0) (actual time=0.444..796.736
rows=173 loops=1)
         Filter: ((id = 5) OR (id = 500) OR (id = 5000))
         Rows Removed by Filter: 999829
         Heap Fetches: 1000002
 Planning time: 0.087 ms
 Execution time: 796.798 ms
```

GIN vs SeqScan

```
EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=10000004782.31..10000004782.32 rows=1 width=0) (actual time=0.279..0.279 rows=1 loops=1)
   ->  Bitmap Heap Scan on tst  (cost=10000000057.50..10000004745.00 rows=14925 width=0) (actual time=0.080..0.267 rows=175 loops=1)
         Recheck Cond: ((id = 5) OR (id = 500) OR (id = 5000))
         Heap Blocks: exact=172
         ->  Bitmap Index Scan on idx_gin  (cost=0.00..57.50 rows=15000 width=0) (actual time=0.059..0.059 rows=147 loops=1)
               Index Cond: ((id = 5) OR (id = 500) OR (id = 5000))
 Planning time: 0.077 ms
 Execution time: 0.308 ms
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Aggregate  (cost=21925.63..21925.64 rows=1 width=0) (actual time=160.412..160.412 rows=1 loops=1)
   ->  Seq Scan on tst  (cost=0.00..21925.03 rows=237 width=0) (actual time=0.535..160.362 rows=175 loops=1)
         Filter: ((id = 5) OR (id = 500) OR (id = 5000))
         Rows Removed by Filter: 999827
 Planning time: 0.459 ms
 Execution time: 160.451 ms
```

GiST OR-ed + KNN

FIND 10 closest to (0,45) POI from Antarctica and Arctica.

Total: 7240858 POI

```
EXPLAIN ANALYZE SELECT name,point FROM geo
WHERE point <@ circle('(90,90)',10) OR
      point <@ circle('(-90,90)',10)
ORDER BY point <-> '(0,45)' LIMIT 10;
```

Individual queries are blazing fast, thanks to KNN !

```
select name,point from geo where point <@ circle('(90,90)',10)
order by point <-> '(0,45)' limit 10;
```

0.561 ms

```
select name,point from geo where point <@ circle('(-90,90)',10)
order by point <-> '(0,45)' limit 10;
```

0.454 ms

GiST OR-ed + KNN

OLD:

```
 Limit  (cost=0.41..662.97 rows=10 width=30) (actual time=7228.580..7266.604 rows=10
loops=1)
   ->  Index Scan using geo_idx on geo  (cost=0.41..958987.91 rows=14474 width=30)
(actual time=7228.570..7266.585 rows=10 loops=1)
         Order By: (point <-> '(0,45)'::point)
         Filter: ((point <@ '<(90,90),10>'::circle) OR (point <@ '<(
90,90),10>'::circle))
         Rows Removed by Filter: 3956498
 Planning time: 0.300 ms
 Execution time: 7266.673 ms
```

NEW:

```
 Limit  (cost=0.41..322.73 rows=1 width=34) (actual time=0.123..0.147 rows=10
loops=1)
   ->  Index Scan using geo_idx on geo  (cost=0.41..322.73 rows=0 width=34) (actual
time=0.121..0.144 rows=10 loops=1)
         Index Cond: ((point <@ '<(90,90),10>'::circle) OR (point <@ '<(-
90,90),10>'::circle))
         Order By: (point <-> '(0,45)'::point)
 Planning time: 0.097 ms
 Execution time: 0.192 ms
```

~ 38 000 FASTER !

Index-only Scan

- Index-only scan now works with OR-ed conditions:
- Index: "id_idx_gist" gist (id)
- EXPLAIN ANALYZE SELECT id FROM geo WHERE id=1 OR id=2000;
- QUERY PLAN
- ---
- Index Only Scan using id_idx_gist on geo (cost=0.41..4.42 rows=0 width=4) (actual time=0.099..0.131 rows=2 loops=1)
  Index Cond: ((id = 1) OR (id = 2000))
  Heap Fetches: 0
  Planning time: 0.263 ms
  Execution time: 0.163 ms
  (5 rows)
-

FTS+KNN works !

- Index: "geo_point_fts_idx" gist (point, fts)
- EXPLAIN ANALYZE SELECT name, point FROM geo WHERE fts @@ to_tsquery('town') OR fts @@ to_tsquery('city')) ORDER BY point <-> '(0,45)'::point LIMIT 10;
- QUERY PLAN
- ---
- Limit (cost=0.67..34.25 rows=10 width=30) (actual time=0.594..2.922 rows=10 loops=1)
  -> Index Scan using geo_point_fts_idx on geo (cost=0.67..136789.20 rows=40735 width=30) (actual time=0.593..2.921 rows=10 loops=1)
  Index Cond: (fts @@ to_tsquery('town') or fts @@ to_tsquery('city'))
  Order By: (point <-> '(0,45)'::point)
  Planning time: 0.271 ms
  Execution time: 2.955 ms
- (6 rows)

Postgres PROFESSIONAL

EAV

Entity-attribute-model

- Normal form
- Join/select is not trivial

XML

- 8.2
- SQL:2003 Conformance
- XPath
- Successfully combines the disadvantages of both binary and text formats
- Use only for legacy

Postgres PROFESSIONAL

HSTORE

- 8.2
- Key/value only
- Strings only
- No nesting
- A lot of features including fast search
- Why not to use it? Especially with perl

Postgres PROFESSIONAL

JSON

- 9.2
- Store as text (like XML), preserve formatting
- No search, no index
- Use it only for storing or in case of needs of preserve duplicates or ordering keys

Postgres PROFESSIONAL

JSONB

- 9.4
- JSON + HSTORE
- Almost all features
- Use it if don't have a strong reason to use anything else

Postgres PROFESSIONAL

Summary

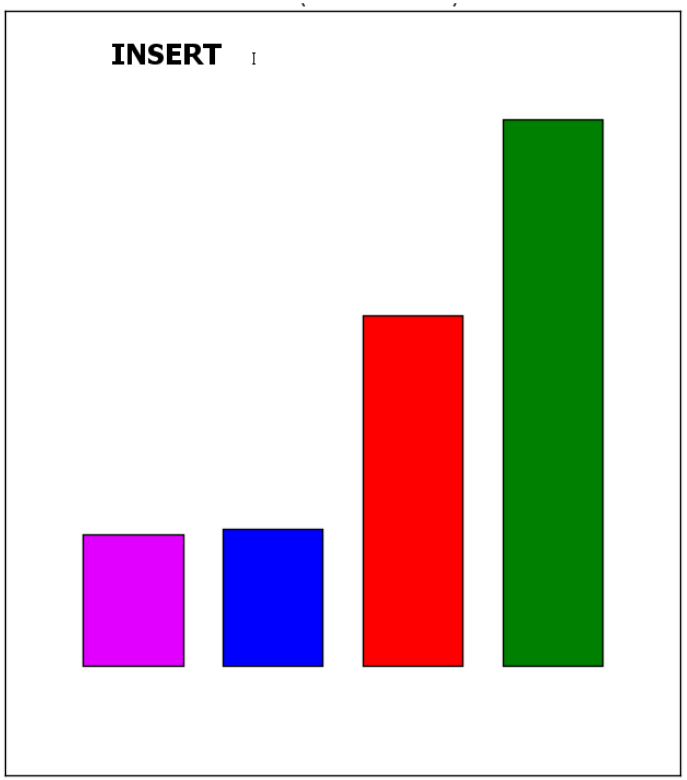
	EAV	XML	HSTORE	JSON	JSONB
Standard					
Binary representation					
Comparison					
Indexes					
Search					
Just storage					
Nesting					
Types					
Fast insert					
Fast dump					

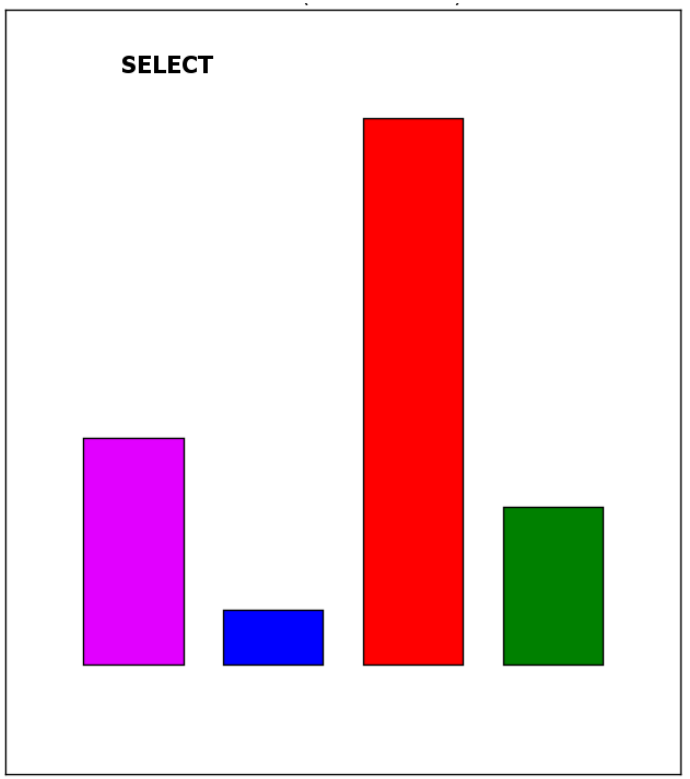
Performance study

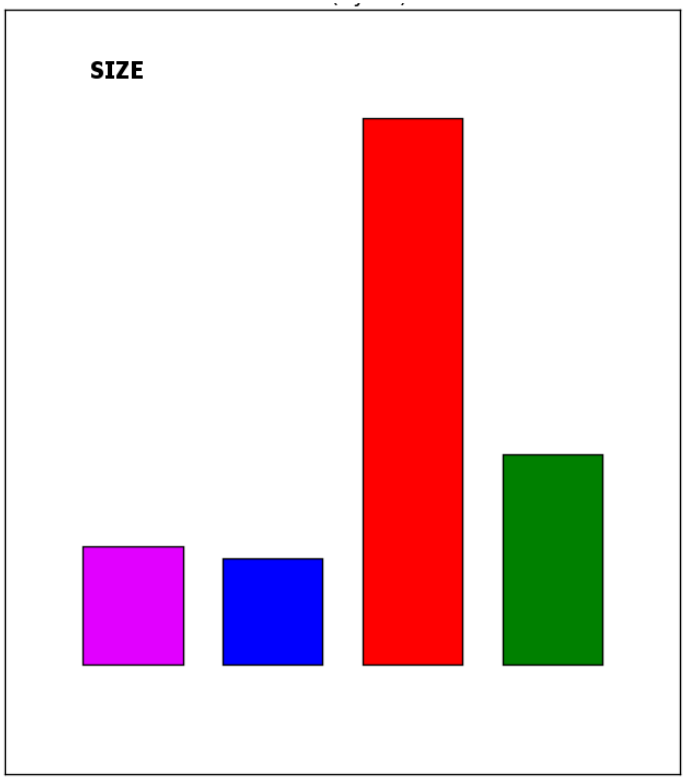
- NoSQL Benchmark
  - https://github.com/EnterpriseDB/pg_nosql_benchmark
  - Amazon instance m4.xlarge, Ubuntu 14.04
- Default configurations
  - PostgreSQL 9.5beta1
    - GIN index — default
    - GIN index - jsonb_path_ops
  - MongoDB 3.2.0rc2 (WiredTiger storage)
  - Mysql 5.7.9 (virtual columns)

PROFESSIONAL Postgres

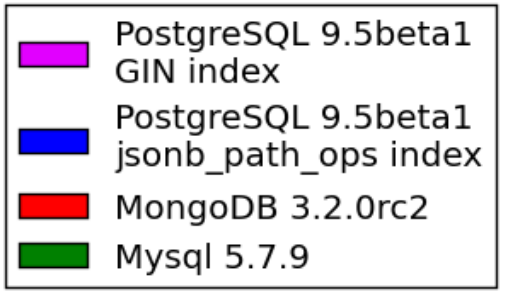
Performance study



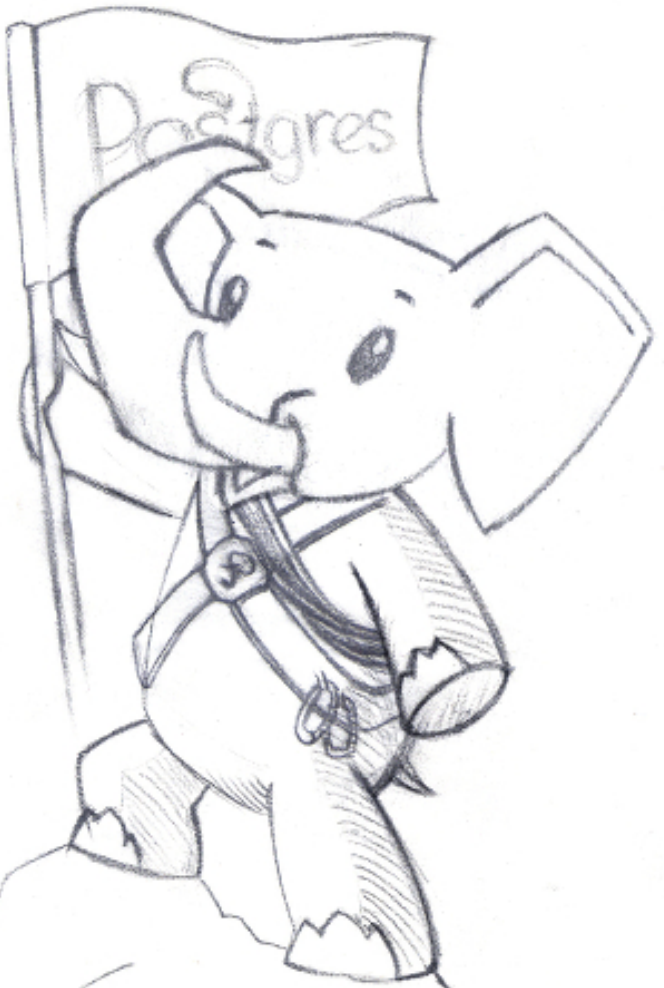




POSTGRES RULEZ !



PROFESSIONAL Postgres



POSTGRES RULEZ !

PROFESSIONAL Postgres