Возможности СУБД Postgres для работы с документноориентированными базами данных

Alexander Korotkov, Oleg Bartunov, Teodor Sigaev

Postgres Professional

Oleg Bartunov, Teodor Sigaev

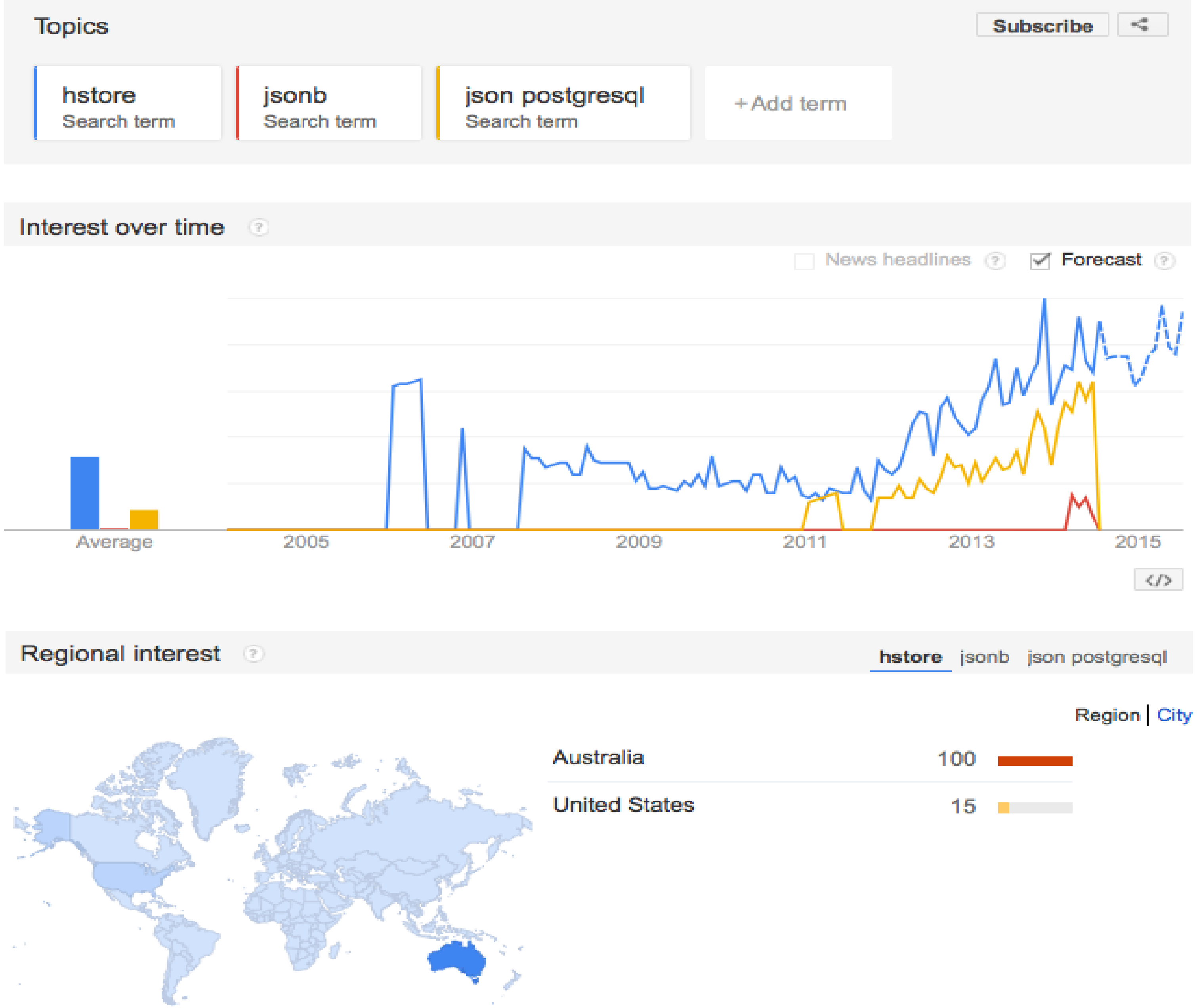
- Locale support
- Extendability (indexing)
  - GiST (KNN), GIN, SP-GiST
- Full Text Search (FTS)
- Jsonb, VODKA
- Extensions:
  - intarray
  - pg_trgm
  - ltree
  - hstore
  - plantuner

https://www.facebook.com/oleg.bartunov
obartunov@gmail.com, teodor@sigaev.ru
https://www.facebook.com/groups/postgresql/

Alexander Korotkov

- Indexed regexp search
- GIN compression & fast scan
- Fast GiST build
- Range types indexing
- Split for GiST
- Indexing for jsonb
- jsquery
- Generic WAL + create am (WIP)

aekorotkov@gmail.com


Topics
Subscribe
hstore
Search term
jsonb
Search term
json postgresql
Search term
+ Add term
Interest over time
News headlines
Forecast
Average
2005
2007
2009
2011
2013
2015
Regional interest
hstore
jsonb
json postgresql
Region | City
Australia 100
United States 15

Agenda

- Semi-structured data in PostgreSQL
- Intruduction into jsonb
- Jsonb indexing
- Jsquery - Jsonb Query Language
- Jsonb GIN opclasses with JsQuery support
- Future of Jsonb querying

Слабо-структурированные данные

- Слабо-структурированные данные возникают от лени :)
- Агрегирование структурированных данных приводит к слабо-структурированным данным (разреженная матрица)
- Все слабо-структурированные данные можно реализовать стандартными способами RDBMS
  - Неудобно, проблемы с производительностью
- json — жупел слабо-структурированных данных
- Реальная проблема — это schema-less данные
  - Реляционные СУБД трудно переживают изменение схемы
  - Key-value (NoSQL) хранилища таких проблем не имеют

Relational Databases

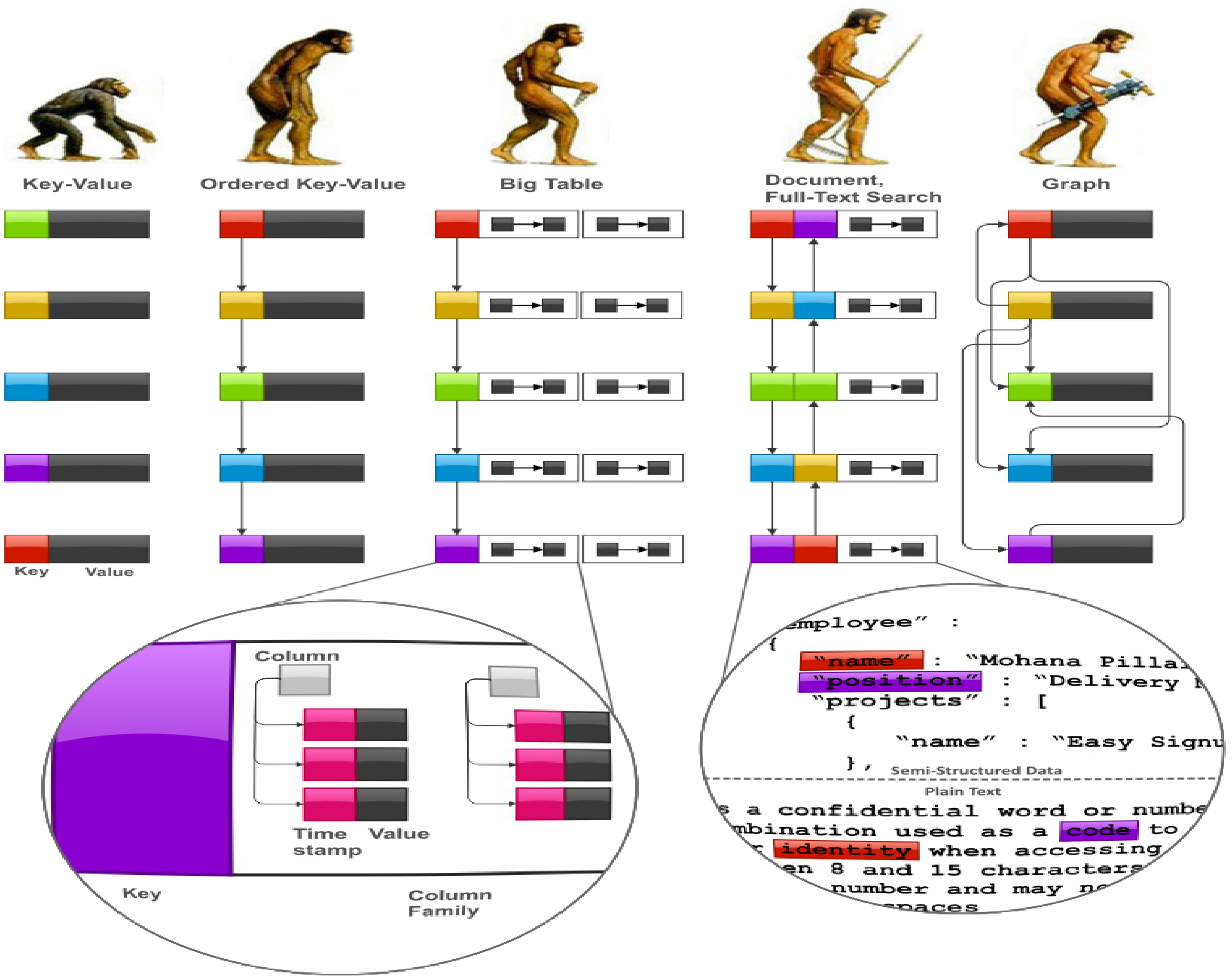
- Реляционные СУБД — интеграционные
  - Все приложения общаются через СУБД
  - SQL — универсальный язык работы с данными
  - Все изменения в СУБД доступны всем
  - Изменения схемы очень затратны, медл. релизы
  - Рассчитаны на интерактивную работу
    - Интересны агрегаты, а не сами данные, нужен SQL
    - SQL отслеживает транзакционность, ограничения целостности... вместо человека

NoSQL (концептуальные предпосылки)

- Сервисная архитектура изменила подход к СУБД
  - Приложение состоит из сервисов, SQL->HTTP
  - Сервисам не нужна одна монолитная СУБД
  - Часто достаточно простых key-value СУБД
  - Схема меняется «на ходу», быстрые релизы
  - ACID → BASE
  - Сервисы — это программы, которые могут сами заниматься агрегированием
  - Сервисы могут сами следить за целостностью данных
- Много данных, аналитика, большое кол-во одновременных запросов
  - Распределенность - кластеры дешевых shared-nothing машин
- NoSQL —горизонтальная масштабируемость и производительность

NoSQL

- Key-value databases
  - Ordered k-v для поддержки диапазонов
- Column family (column-oriented) stores
  - Big Table — value имеет структуру:
    - column families, columns, and timestamped versions (maps-of maps-of maps)
- Document databases
  - Value - произвольная сложность, индексы
    - Имена полей, FTS — значение полей
- Graph databases — эволюция ordered-kv



Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
Key
Value
Column
Time stamp
Value
Key
Column Family
Semi-Structured Data
Plain Text

Челлендж !

- Полноценная работа со слабо-структурированными данными в реляционной СУБД
  - Хранение (тип данных для хранение key-value данных)
  - Поиск (операторы и функции)
  - Производительность (бинарное хранилище, индексы)

Introduction to hstore

- Hstore — key/value storage (inspired by 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,......)

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
```

Introduction to hstore

- Hstore benefits
  - In provides a flexible model for storing a semi-structured data in relational database
  - hstore has binary storage
- 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 явно быстрее json даже на простых данных

```
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 !

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

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 type

- 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 bookmarks
- Server
  - MBA, 8 GB RAM, 256 GB SSD
- 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

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: hash path.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

MongoDB 2.6.0

- Load data - ~13 min **SLOW !** **Jsonb 43 s**

```
mongoimport --host localhost -c js --type json < delicious-rss-1250k
2014-04-08T22:47:10.014+0400                3700      1233/second
...
2014-04-08T23:00:36.050+0400                1252000   1547/second
2014-04-08T23:00:36.565+0400 check 9 1252973
2014-04-08T23:00:36.566+0400 imported 1252973 objects
```

- Search - ~ 1s (seqscan) **THE SAME**

```
db.js.find({tags: {$elemMatch:{ term: "NYC"}}}).count()
285
-- 980 ms
```

- Search - ~ 1ms (indexscan) **Jsonb 0.7ms**

```
db.js.ensureIndex( {"tags.term" : 1} )
db.js.find({tags: {$elemMatch:{ term: "NYC"}}}).
```

Summary: 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

Citus dataset

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


```
{
  "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"
  ]
}
```

Citus dataset: storage in jsonb

Heap size: 1588 MB
PK size: 65 MB
GIN index on product ids: 89 MB

```
                 Table "public.customer_reviews_jsonb"
 Column |  Type   |                  Modifiers
--------+---------+------------------------------------------------------
 id     | integer | not null default
        |         | nextval('customer_reviews_jsonb_id_seq'::regclass)
 jr     | jsonb   |
Indexes:
    "customer_reviews_jsonb_pkey" PRIMARY KEY, btree (id)
    "customer_reviews_jsonb_similar_product_ids_idx" gin
         ((jr -> 'similar_product_ids'::text))
```

Citus dataset: normalized storage

Heap size: 434 MB (main table) + 598 MB (similar products) = 1032 MB

PK size: 65 MB (main table) + 304 MB (similar products) = 369 MB

Index on similar product id: 426 MB

```
    Table "public.customer_reviews_flat"
        Column        | Type    | Modifiers
----------------------+---------+-----------
 customer_id          | text    | not null
 review_date          | date    | not null
...
 product_subcategory  | text    |
 id                   | integer | not null...
Indexes:
    "customer_reviews_flat_pkey"
        PRIMARY KEY, btree (id)
```

```
Table "public.customer_reviews_similar_product"
       Column       |  Type    | Modifiers
--------------------+----------+-----------
 product_id         | integer  | not null
 similar_product_id | bpchar   | not null
Indexes:
    "similar_product_product_id_idx"
        btree (product_id)
    "similar_product_similar_product_id_idx"
        btree (similar_product_id COLLATE "C")
```

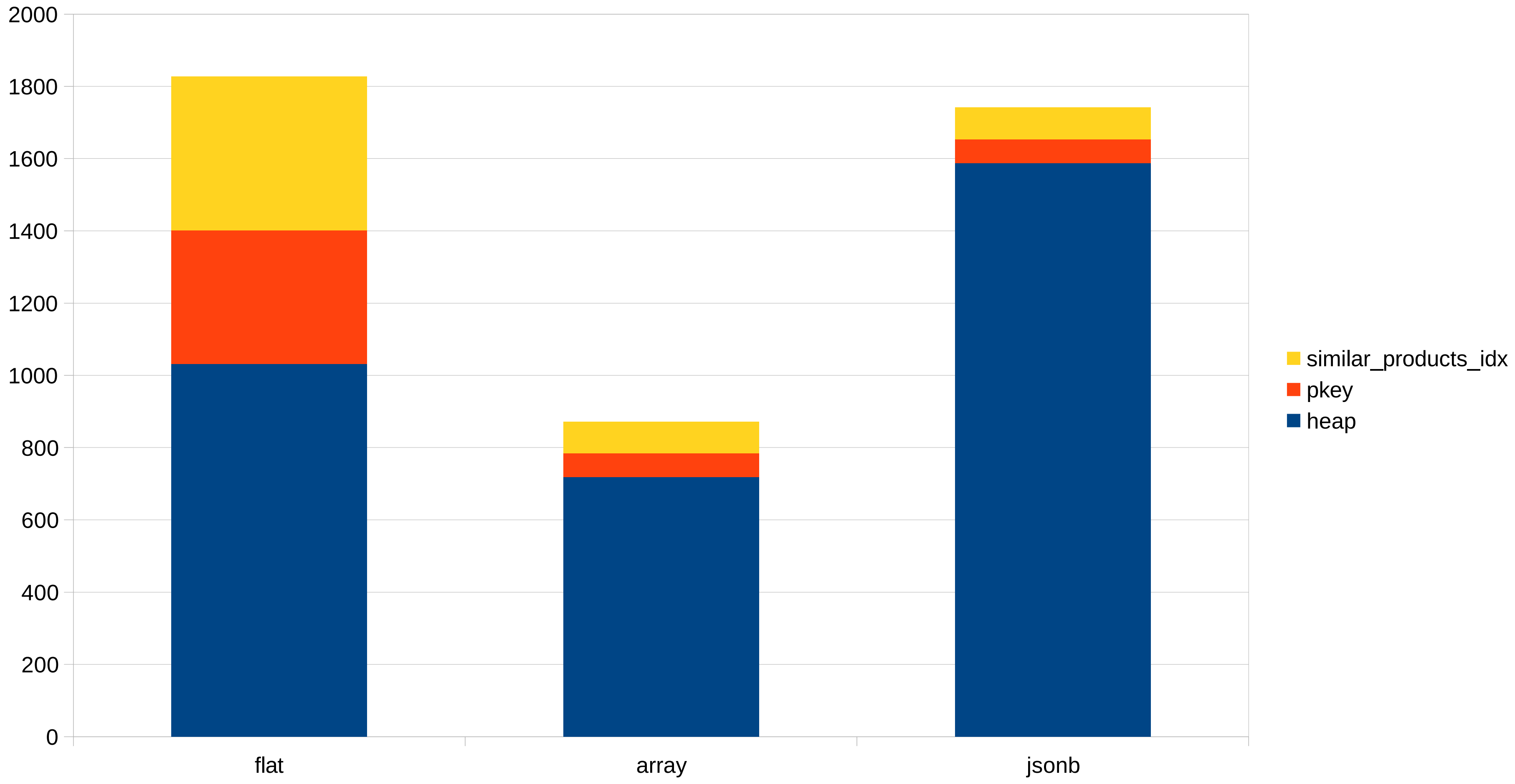
14 168 514 rows

Citus dataset: storage using array

Heap size: 719 MB
PK size: 65 MB
GIN index on product ids: 87 MB

```
         Table "public.customer_reviews_array"
        Column          |       Type        |  Modifiers
------------------------+-------------------+--------------
 id                     | integer           | not null …
 customer_id            | text              | not null
 review_date            | date              | not null
...
 similar_product_ids    | character(10)[]   |
Indexes:
    "customer_reviews_array_pkey" PRIMARY KEY, btree (id)
    "customer_reviews_array_similar_product_ids_idx gin
                      (similar_product_ids COLLATE "C")
```

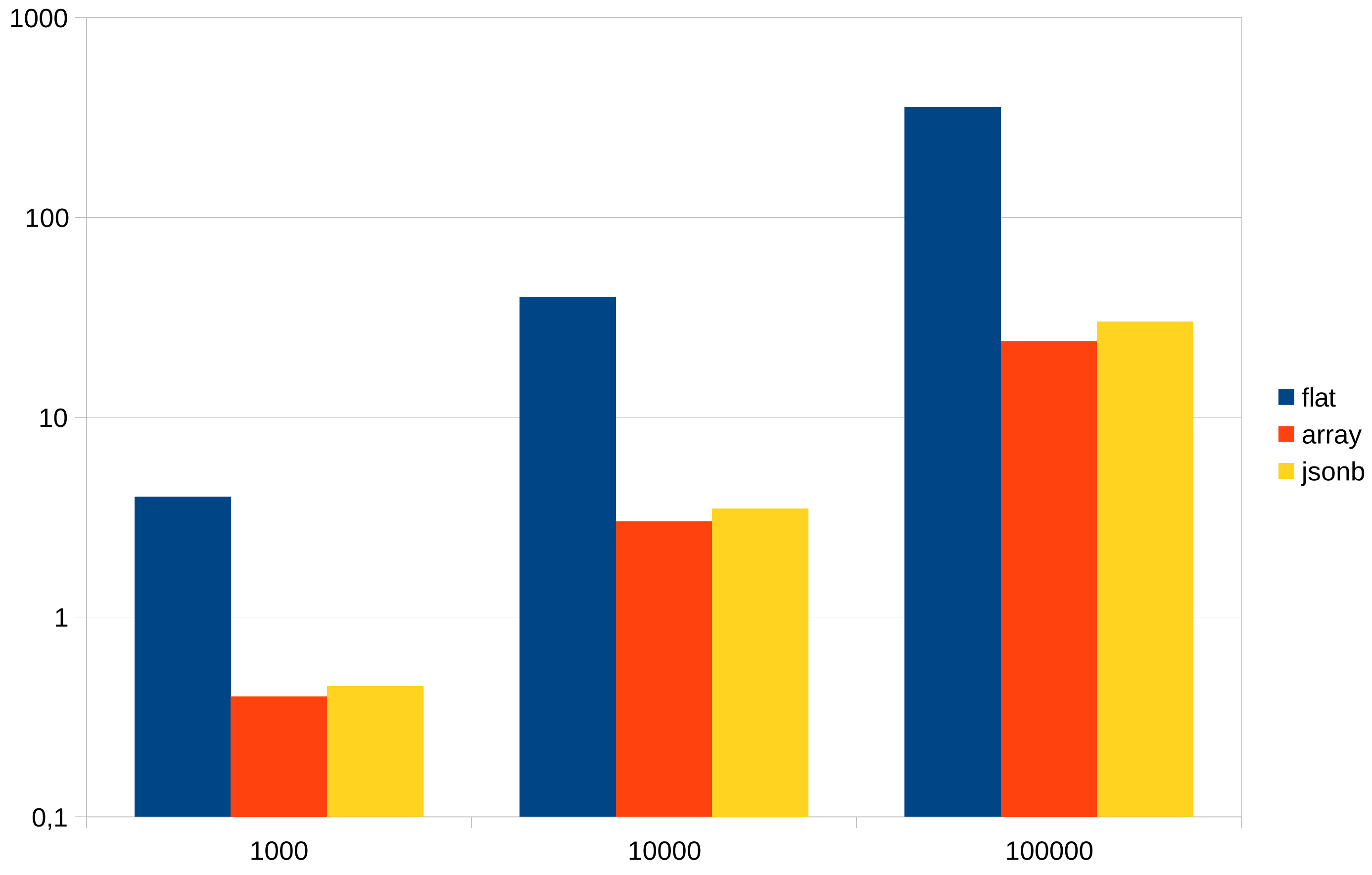
Citus dataset: storage size



Citus dataset: storage conclusion

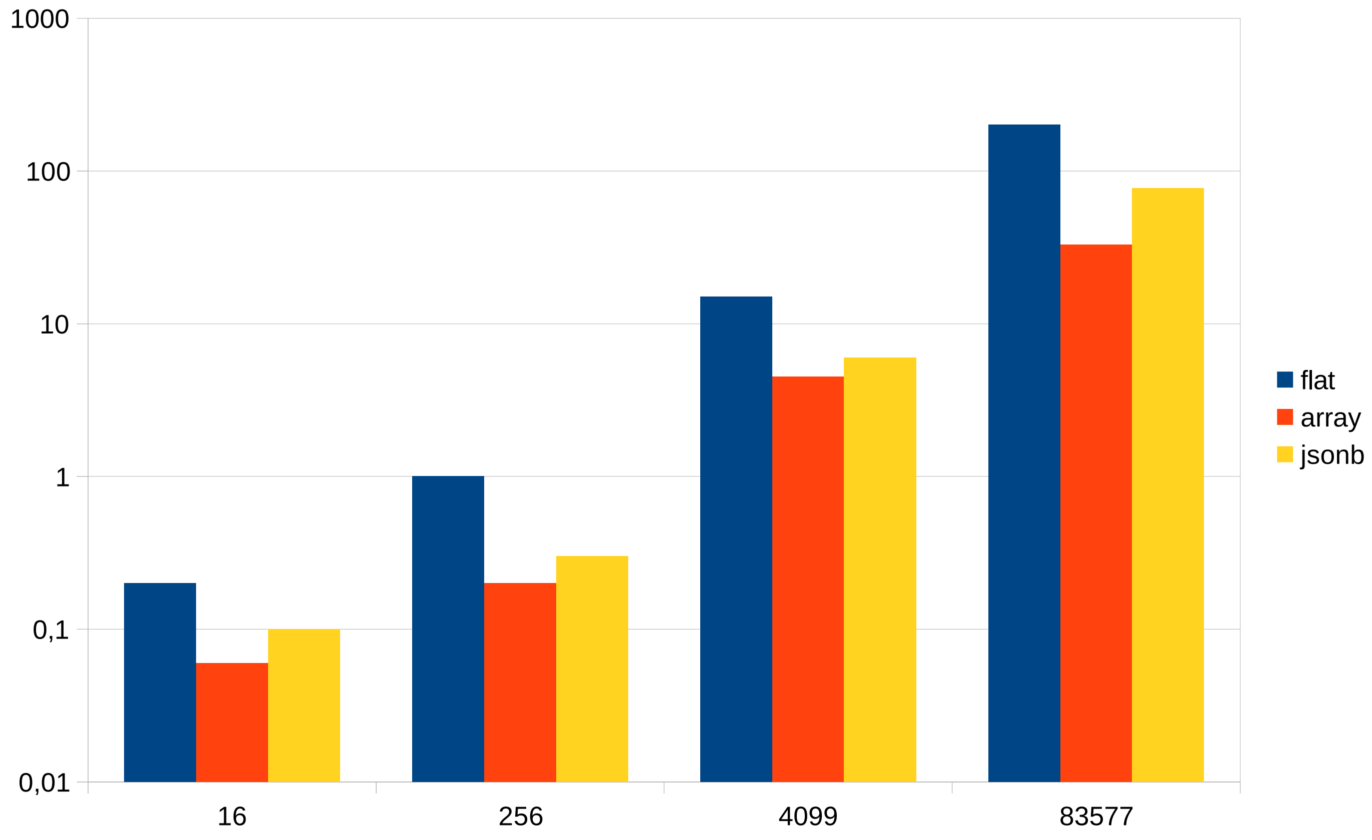
- Array storage is most compact. It doesn't have to store keys. Storage of similar ids as an array is very space efficient as well. However, it's not as flexible as jsonb.
- GIN handles duplicates very efficient.
- Jsonb and flat versions have about same size. Each version has its own overhead. Jsonb stores key names and extra headers. Flat version has to store extra tuple headers and larger btree indexes.

Querying objects by ID



Count	flat	array	jsonb
1000	4 ms	0,40 ms	0,45 ms
10 000	40 ms	3 ms	3,5 ms
100 000	357 ms	24 ms	30 ms

Querying similar objects



Count	flat, ms	array, ms	jsonb, ms
16	0,2	0,06	0,1
256	1	0,2	0,3
4099	15	4,5	6
83577	200	33	77

Citus dataset: querying objects

- With flat storage you have to query two tables in order to assemble objects. That gives significant overhead to flat storage.
- Performace of array and jsonb version is about the same. Array version is slightly faster because its storage is more compact.

Jsonb (Apr, 2014)

- Documentation
  - JSON Types, JSON Functions and Operators
- There are many functionality left in nested hstore
  - Can be an extension
- Need query language for jsonb
  - <,>,&& … operators for values
    a.b.c.d && [1,2,10]
  - Structural queries on paths
    *.d && [1,2,10]
  - Indexes !

Jsonb query

Currently, one can search jsonb data using:

- 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

Jsonb querying an array: simple case

Find bookmarks with tag «NYC»:

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

Jsonb querying an array: complex case

Find companies where CEO or CTO is called Neil.

One could write...

```
SELECT * FROM company
WHERE js @> '{"relationships":[{"person":
                 {"first_name":"Neil"}}]}' AND
         (js @> '{"relationships":[{"title":"CTO"}]}' OR
          js @> '{"relationships":[{"title":"CEO"}]}');
```

Jsonb querying an array: complex case

Each «@>» is processed independently.

```
SELECT * FROM company
WHERE js @> '{"relationships":[{"person":
                 {"first_name":"Neil"}}]}' AND
         (js @> '{"relationships":[{"title":"CTO"}]}' OR
          js @> '{"relationships":[{"title":"CEO"}]}');
```

Actually, this query searches for companies with some CEO or CTO and someone called Neil...

Jsonb querying an array: complex case

The correct version is so.

```
SELECT * FROM company
WHERE js @> '{"relationships":[{"title":"CEO",
                  "person":{"first_name":"Neil"}}]}' OR
      js @> '{"relationships":[{"title":"CTO",
                  "person":{"first_name":"Neil"}}]}';
```

When constructing complex conditions over same array element, query length can grow exponentially.

Jsonb querying an array: another approach

Using subselect and jsonb_array_elements:

```
SELECT * FROM company
WHERE EXISTS (
    SELECT 1
    FROM jsonb_array_elements(js -> 'relationships') t
    WHERE t->>'title' IN ('CEO', 'CTO') AND
          t ->'person'->>'first_name' = 'Neil');
```

Jsonb querying an array: summary

Using «@>»

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

Using subselect and jsonb_array_elements

- Pro
  - Full power of SQL can be used to express condition over element
- Cons
  - No indexing support
  - Heavy syntax

Jsonb query

- Need Jsonb query language
  - Simple and effective way to search in arrays (and other iterative searches)
  - More comparison operators
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support
- Introduce Jsquery - textual data type and @@ match operator

```
jsonb @@ jsquery
```

Jsonb query language (Jsquery)

- # - 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;
```

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

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

key_any   ::= key
        | NOT
```

Jsonb query language (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
```

```
value_list
        ::= scalar_value
          | value_list ',' scalar_value

array   ::= '[' value_list ']'

scalar_value
        ::= null
          | STRING
          | true
          | false
          | NUMERIC
          | OBJECT
        .......
```

Jsonb query language (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]';
```

```
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
```

Jsonb query language (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

Jsonb query language (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()
```

«#», «*», «%» 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
```

Jsonb query language (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	

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 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
```

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" : [
                                    [
                                                "B000089778",
                                                "B000089778"
                                    ]
                        ]
            },
}
```

index size = 400 MB just for similar_product_ids !!!

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 !
```

Jsquery (optimizer) — NEW !

- Jsquery now has built-in simple optimiser.

```
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 (optimizer) — NEW !

- Since GIN opclasses can't expose something special to explain output, jsquery optimiser has its own explain functions:
  - text gin_debug_query_path_value(jsquery) – explain for jsonb_path_value_ops

```
# SELECT gin_debug_query_path_value('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +
```

  - text gin_debug_query_value_path(jsquery) – explain for jsonb_value_path_ops

```
# SELECT gin_debug_query_value_path('x = 1 AND (*.y = 1 OR y = 2)');
 gin_debug_query_value_path
----------------------------
 AND                       +
   x = 1 , entry 0         +
   OR                      +
     *.y = 1 , entry 1     +
     y = 2 , entry 2       +
```

Jsquery (optimizer) — NEW !

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 = *)

Jsquery (optimizer) — NEW !

AND children can be put into recheck.

```
# SELECT gin_debug_query_path_value('x = 1 AND y > 0');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +
```

While OR children can't. We can't handle false negatives.

```
# SELECT gin_debug_query_path_value('x = 1 OR y > 0');
 gin_debug_query_path_value
----------------------------
 OR                        +
   x = 1 , entry 0         +
   y > 0 , entry 1         +
```

Jsquery (optimizer) — NEW !

Can't do much with NOT, because hash is lossy. After NOT false positives turns into false negatives which we can't handle.

```
# SELECT gin_debug_query_path_value('x = 1 AND (NOT y = 0)');
 gin_debug_query_path_value
----------------------------
 x = 1 , entry 0           +
```

Jsquery (optimizer) — NEW !

- 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) — NEW !

- 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 (HINTING) — NEW !

- If you know that inequality is selective then use HINT /* --index */

```
# explain (analyze, costs off) select count(*) from jr where jr @@ 'product_sales_rank /*-- index*/ > 
3000000 AND review_rating = 5'::jsquery;
                                                      QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Aggregate (actual time=12.307..12.307 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=11.259..12.244 rows=739 loops=1)
         Recheck Cond: (jr @@ '("product_sales_rank" /*-- index */  > 3000000 AND "review_rating" = 
5)'::jsquery)
         Heap Blocks: exact=705
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=11.179..11.179 rows=739 loops=1)
               Index Cond: (jr @@ '("product_sales_rank" /*-- index */  > 3000000 AND "review_rating" = 
5)'::jsquery)
 Execution time: 12.359 ms vs 1709.901 ms (without hint)
(7 rows)
```

Jsquery use case: schema specification

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

Non-numeric coordinates don't exist => All coordinates are numeric

Jsquery use case: schema specification

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

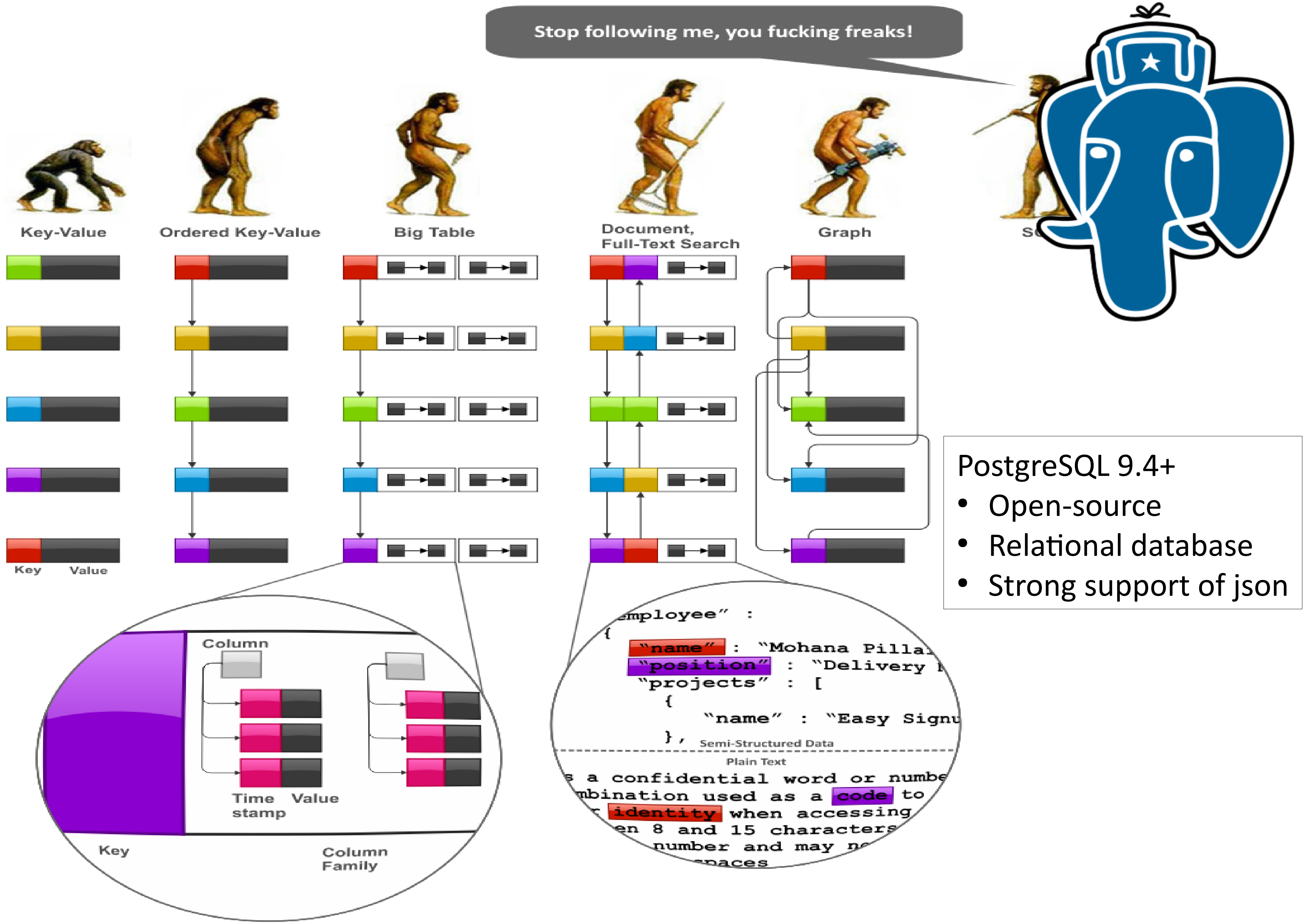
- ```
  # INSERT INTO js (v) VALUES ('{"name": 1, "coords": [{"x": 1, "y": 2}, {"x": "3", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint "js_v_check"
  ```

- ```
  # 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 !
  - We will release it after PostgreSQL 9.4 release
  - Need real sample data and queries !

Postgres PROFESSIONAL



PostgreSQL 9.4+

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

Better indexing ...

- GIN is a proven and effective index access method
- Need indexing for jsonb with operations on paths (no hash!) and values
  - B-tree in entry tree is not good - length limit, no prefix compression

List of relations

Schema	Name	Type	Owner	Table	Size	Description
public	jb	table	postgres		1374 MB	
public	jb_uniq_paths	table	postgres		912 MB	
public	jb_uniq_paths_btree_idx	index	postgres	jb_uniq_paths	885 MB	text_pattern_ops
public	jb_uniq_paths_spgist_idx	index	postgres	jb_uniq_paths	598 MB	now much less !

Better indexing ...

- Provide interface to change hardcoded B-tree in Entry tree
  - Use spgist opclass for storing paths and values as is (strings hashed in values)
- We may go further - provide interface to change hardcoded B-tree in posting tree
  - GIS aware full text search !
- New index access method

CREATE INDEX … USING VODKA

GIN History

- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev

Generalized Inverted Index

• An inverted index is an index structure storing a set of (key, posting list) pairs, where 'posting list' is a set of documents in which the key occurs.

• Generalized means that the index does not know which operation it accelerates. It works with custom strategies, defined for specific data types. GIN is similar to GiST and differs from B-Tree indices, which have predefined, comparison-based operations.

GIN History

- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev
- Supported by JFG Networks (France)
- «Gin stands for Generalized Inverted iNdex and should be considered as a genie, not a drink.»
- Alexander Korotkov, Heikki Linnakangas have joined GIN++ development in 2013

GIN History

- From GIN Readme, posted in -hackers, 2006-04-26

```
TODO
----

Nearest future:

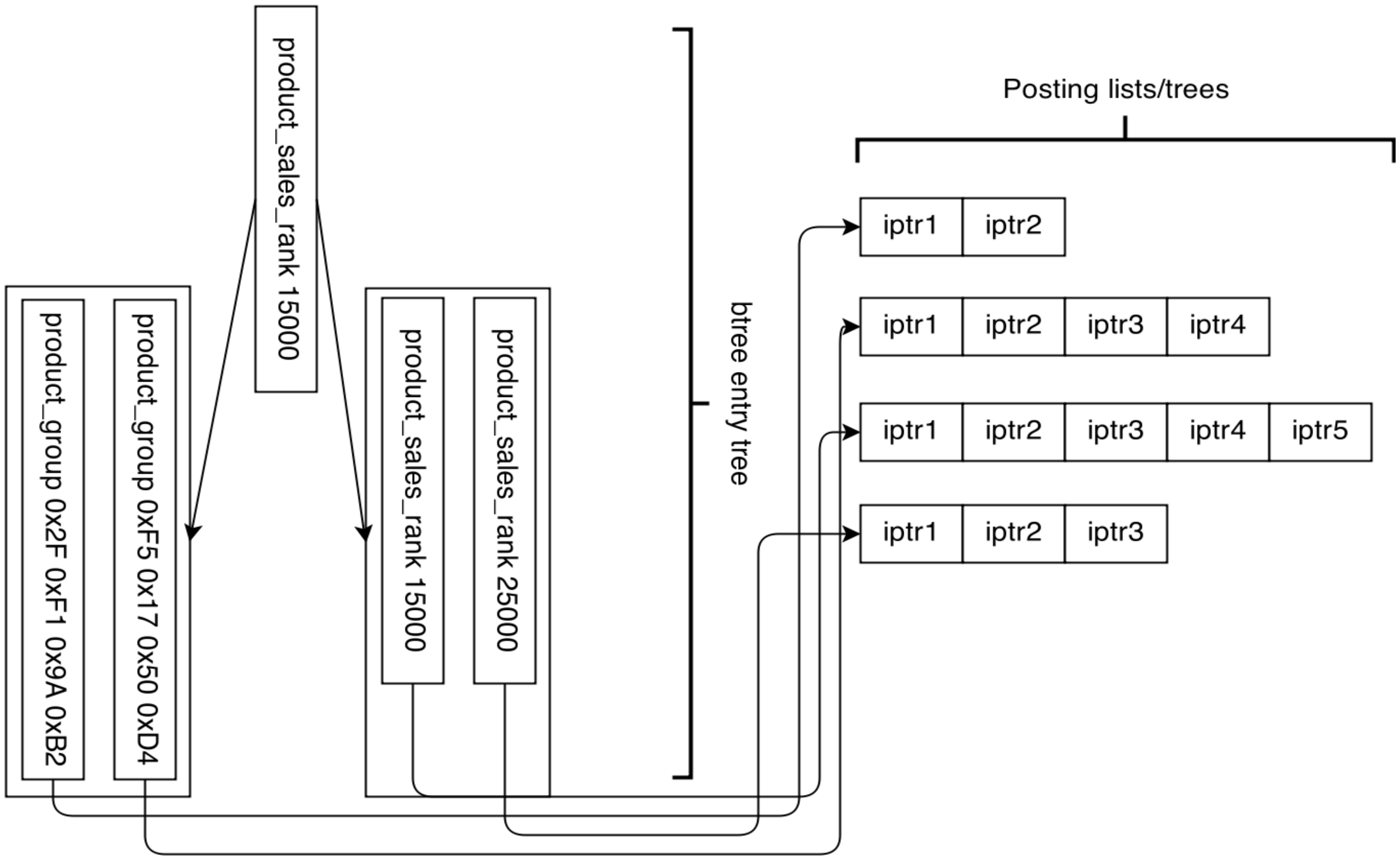
  * Opclasses for all types (no programming, just many catalog changes).

Distant future:

  * Replace B-tree of entries to something like GiST (VODKA ! 2014)
  * Add multicolumn support
  * Optimize insert operations (background index insertion)
```

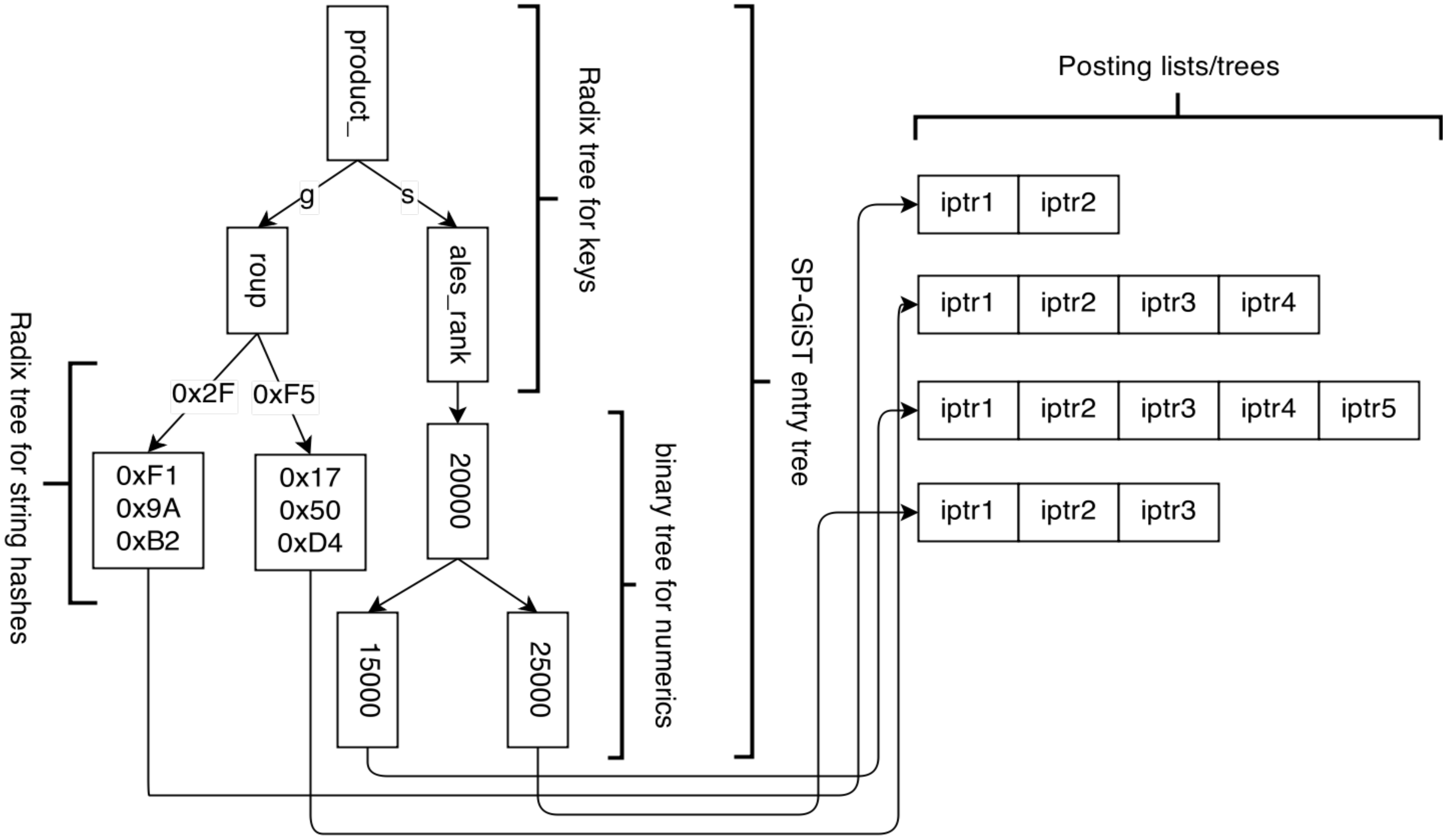
GIN index structure for jsonb

```
{
    "product_group": "Book",
    "product_sales_rank": 15000
},
{
    "product_group": "Music",
    "product_sales_rank": 25000
}
```



Vodka index structure for jsonb

```
{
   "product_group": "Book",
   "product_sales_rank": 15000
},
{
   "product_group": "Music",
   "product_sales_rank": 25000
}
```



CREATE INDEX … USING VODKA

- Delicious bookmarks, mostly text data

```
set maintenance_work_mem = '1GB';

                             List of relations
 Schema |         Name         | Type  |  Owner   | Table |  Size   | Description
--------+----------------------+-------+----------+-------+---------+-------------
 public | jb                   | table | postgres |       | 1374 MB |  1252973 rows
 public | jb_value_path_idx    | index | postgres | jb    | 306 MB  |   98769.096
 public | jb_gin_idx           | index | postgres | jb    | 544 MB  |  129860.859
 public | jb_path_value_idx    | index | postgres | jb    | 306 MB  |  100560.313
 public | jb_path_idx          | index | postgres | jb    | 251 MB  |   68880.320
 public | jb_vodka_idx         | index | postgres | jb    | 409 MB  |  185362.865
 public | jb_vodka_idx5        | index | postgres | jb    | 325 MB  |  174627.234 new spgist
(6 rows)
```

CREATE INDEX … USING VODKA

```
select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
--------------------------------------------------------------------------------
 Aggregate (actual time=0.423..0.423 rows=1 loops=1)
   -> Bitmap Heap Scan on jb (actual time=0.146..0.404 rows=285 loops=1)
        Recheck Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on jb_vodka_idx (actual time=0.108..0.108 rows=285 loops=1)
             Index Cond: (jb @@ '"tags".#."term" = "NYC"'::jsquery)
 Execution time: 0.456 ms  (0.634 ms,  GIN jsonb_value_path_ops)
```

```
select count(*) from jb where jb @@ '*.term = "NYC"';
--------------------------------------------------------------------------------
 Aggregate (actual time=0.495..0.495 rows=1 loops=1)
   -> Bitmap Heap Scan on jb (actual time=0.245..0.474 rows=285 loops=1)
        Recheck Cond: (jb @@ '*."term" = "NYC"'::jsquery)
        Heap Blocks: exact=285
        -> Bitmap Index Scan on jb_vodka_idx (actual time=0.214..0.214 rows=285 loops=1)
             Index Cond: (jb @@ '*."term" = "NYC"'::jsquery)
 Execution time: 0.526 ms  (0.716 ms, GIN jsonb_path_value_ops)
```

CREATE INDEX … USING VODKA

- CITUS data, text and numeric

```
set maintenance_work_mem = '1GB';

                                  List of relations
 Schema |         Name         | Type  |  Owner   | Table |  Size   | Description
--------+----------------------+-------+----------+-------+---------+-------------
 public | jr                   | table | postgres |       | 1573 MB |  3023162 rows
 public | jr_value_path_idx    | index | postgres | jr    | 196 MB  |  79180.120
 public | jr_gin_idx           | index | postgres | jr    | 235 MB  | 111814.929
 public | jr_path_value_idx    | index | postgres | jr    | 196 MB  |  73369.713
 public | jr_path_idx          | index | postgres | jr    | 180 MB  |  48981.307
 public | jr_vodka_idx3        | index | postgres | jr    | 240 MB  | 155714.777
 public | jr_vodka_idx4        | index | postgres | jr    | 211 MB  | 169440.130 new spgist

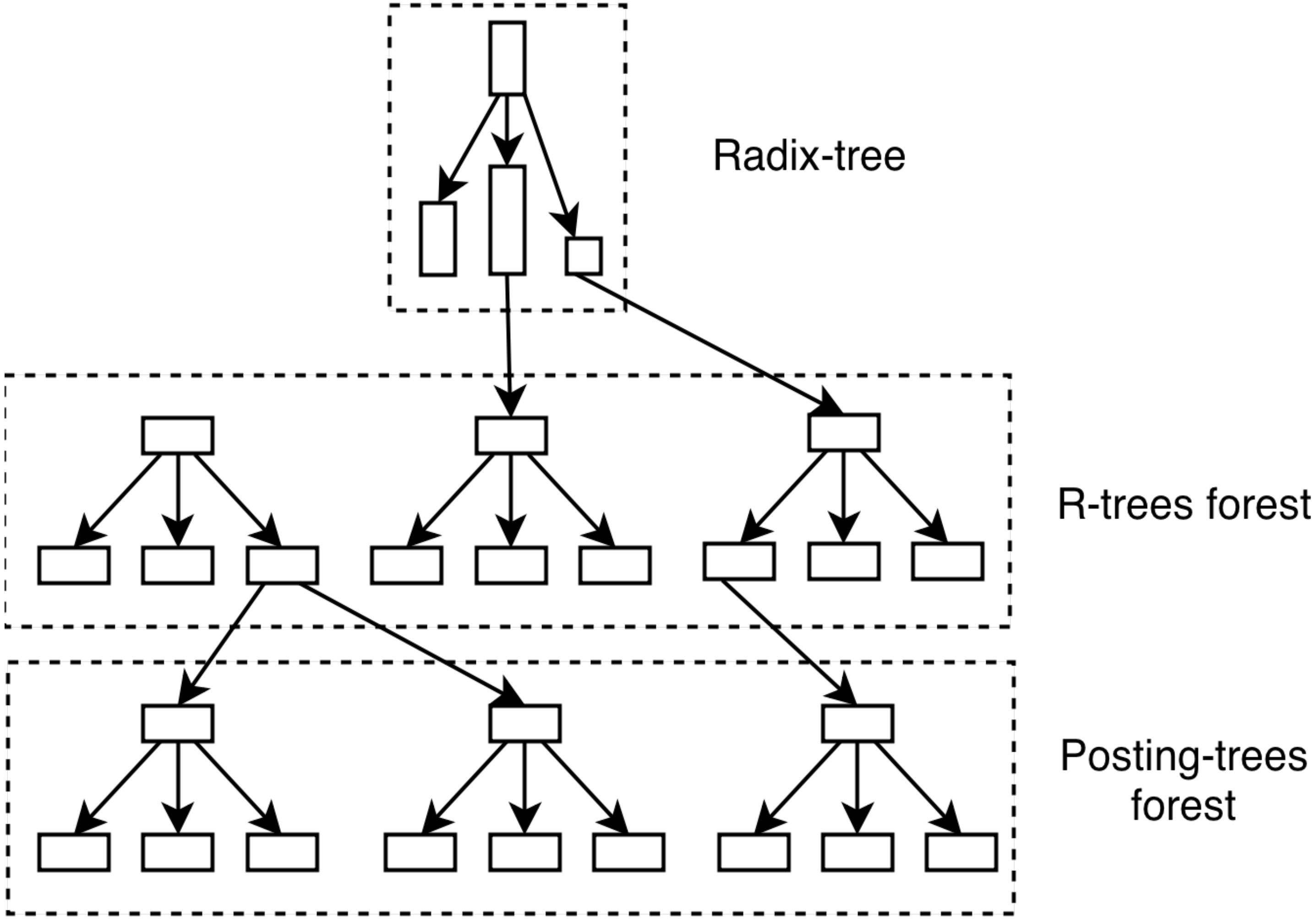
(6 rows)
```

CREATE INDEX … USING VODKA

```
explain (analyze, costs off) select count(*) from jr where jr @@  ' similar_product_ids && ["B000089778"]';
                              QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate (actual time=0.200..0.200 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.090..0.183 rows=185 loops=1)
         Recheck Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_vodka_idx (actual time=0.077..0.077 rows=185 loops=1)
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::jsquery)
 Execution time: 0.237 ms (0.394 ms, GIN jsonb_path_value_idx)
(7 rows)
```

There are can be different flavors of Vodka

New VODKA concept



- Posting list/tree is just a way of effective duplicate storage
- Entry tree can consist of multiple levels of different access methods
- VODKA is a way to combine different access method in single index: VODKA CONNECTING INDEXES

JsQuery limitations

- Variables are always on the left size

```
x = 1 – OK
1 = x – Error!
```

- No calculations in query

```
x + y = 0 — Error!
```

- No extra datatypes and search operators

```
point(x,y) <@ '((0,0),(1,1),(2,1),(1,0))'::polygon
```

JsQuery limitations

Users want jsquery to be as rich as SQL...

JsQuery limitations

Users want jsquery to be as rich as SQL ...

... But we will discourage them ;)

JsQuery language goals

- Provide rich enough query language for jsonb in 9.4.
- Indexing support for 'jsonb @@ jsquery':
  - Two GIN opclasses are in jsquery itself
  - VODKA opclasses was tested on jsquery

It's NOT intended to be solution for jsonb querying in long term!

What JsQuery is NOT?

It's **not** designed to be another **extendable, full weight**:

- Parser
- Executor
- Optimizer

It's NOT SQL inside SQL.

Jsonb querying an array: 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!

Jsonb query: future

Users want jsonb query language to be as rich as SQL. How to satisfy them?..

Jsonb query: future

Users want jsonb query language to be as rich as SQL. How to satisfy them?

Bring all required features to SQL-level!

Jsonb query: future

Functional equivalents:

- ```
  SELECT * FROM company WHERE EXISTS (SELECT 1
  FROM jsonb_array_elements(js->'relationships') t
  WHERE t->>'title' IN ('CEO', 'CTO') AND
        t->'person'->>'first_name' = 'Neil');
  ```
- ```
  SELECT count(*) FROM company WHERE js @@ 'relationships(#.title
  in ("CEO", "CTO") AND #.person.first_name = "Neil")'::jsquery;
  ```
- ```
  SELECT * FROM company WHERE
      ANYELEMENT OF js-> 'relationships' AS t (
          t->>'title' IN ('CEO', 'CTO') AND
          t ->'person'->>'first_name' = 'Neil');
  ```
  ```

Jsonb query: ANYELEMENT

Possible implementation steps:

- Implement ANYELEMENT just as syntactic sugar and only for arrays.
- Support for various data types (extendable?)
- Handle ANYLEMENT as expression not subselect (problem with alias).
- Indexing support over ANYELEMENT expressions.

Another idea about ANYLEMENENT

Functional equivalents:

- ```
  SELECT t
  FROM company,
       LATERAL (SELECT t FROM
                jsonb_array_elements(js->'relationships') t) el;
  ```

- ```
  SELECT t
  FROM company,
       ANYELEMENT OF js->'relationships' AS t;
  ```

Summary

- contrib/jsquery for 9.4
  - Jsquery - Jsonb Query Language
  - Two GIN opclasses with jsquery support
  - Grab it from https://github.com/akorotkov/jsquery (branch master)
- Prototype of VODKA access method (supported by Heroku)
- New VODKA concept
- Idea of Jsonb querying in SQL

Спасибо за внимание!