Авторский взгляд на слабо-структурированные данные в PostgreSQL

Alexander Korotkov <a.korotkov@postgrespro.ru>
Teodor Sigaev <teodor@postgrespro.ru>
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

Кто использует?

PROFESSIONAL Postgres

Слабоструктурированные данные

Что ЭТО?

ССД

- Структура есть. Но плохо определена, либо изменчива, либо просто лень её описывать.
- Задача хранения (настройки аккаунта)
- Задача поиска (интернет-магазин, каталоги)

EAV

Entity-attribute-model

- Нормальная форма - норма жизни
- Сбор параметров - хорошо автоматизированная, но не тривиальная схема
- Ненормальное количество join в запросе, невысокая эффективноть

XML

- 8.2
- SQL:2003 Conformance
- XPath
- Удачно сочетает недостатки как бинарных так и текстовых форматов
- Когда использовать: когда нет выхода

HSTORE

- 8.2
- Ключ/Значения
- только строки
- Никакой вложенности
- Много полезняшек, в том числе быстрый поиск
- Когда использовать: почему бы и нет. Особенно с Perl.

JSON

- 9.2
- Храниться как текст (как и XML) (сохраняет форматирование)
- Нет индексов, операций сравнения
- Когда использовать: нет поисков, сохранение дубликатов и порядка

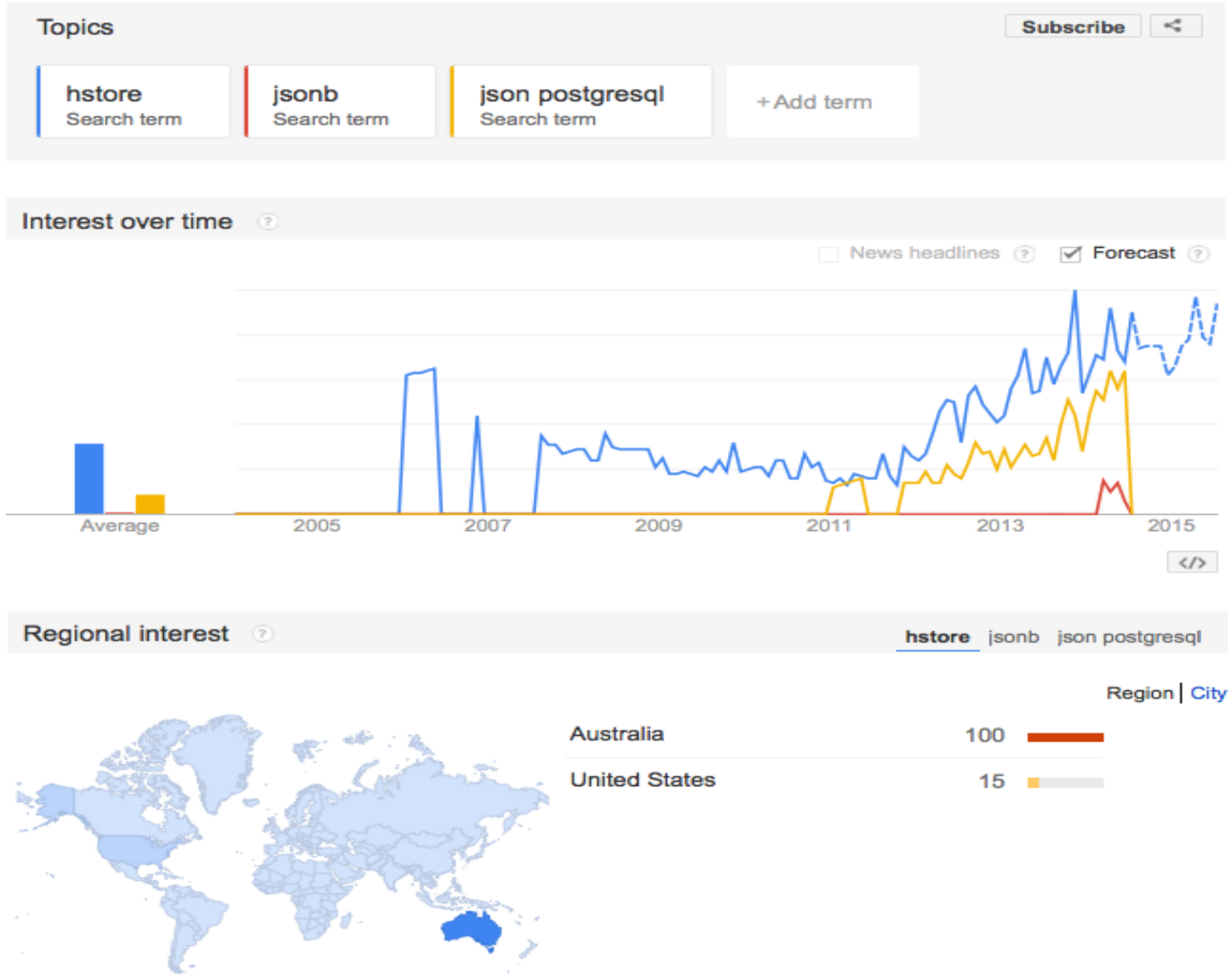
JSONB

- 9.4
- JSON + HSTORE
- Есть всё или около того.
- Когда использовать: когда у вас нет причин использовать что-то ещё

Сравнение

	EAV	XML	HSTORE	JSON	JSONB
Стандарт					
Бинарное представление					
Операции сравнения					
Индексная поддержка					
Скорость поиска					
Только хранение					
Вложенность					
Типы					
Скорость вставки					
Скорость дампа					

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

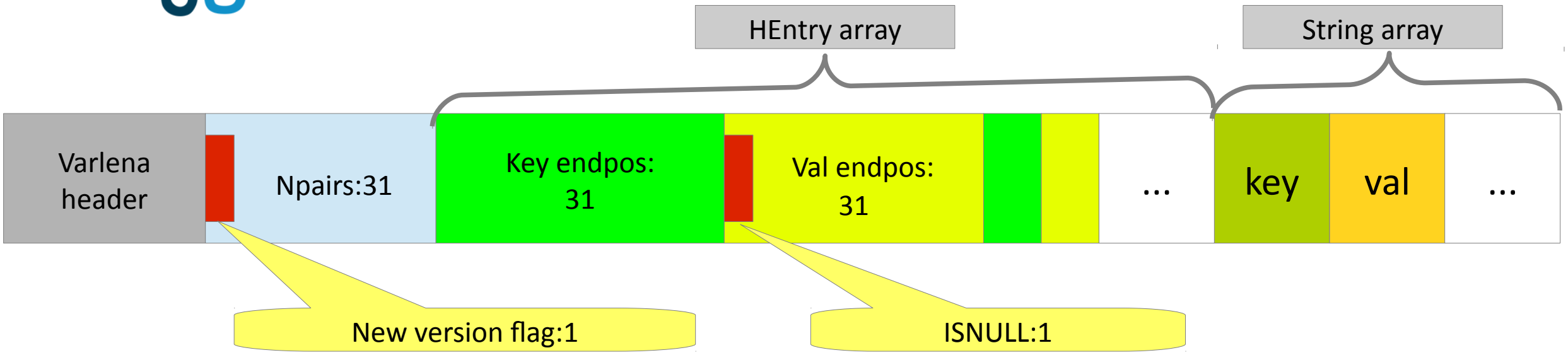
- Hstore — key/value binary storage (inspired by perl hash)

  ```
  '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,.....)
- 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

Hstore limitations

- Levels: unlimited
- Number of elements in array: 2^31
- Number of pairs in hash: 2^31
- Length of string: 2^31 bytes

2^31 bytes = 2 GB

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

Inverted Index

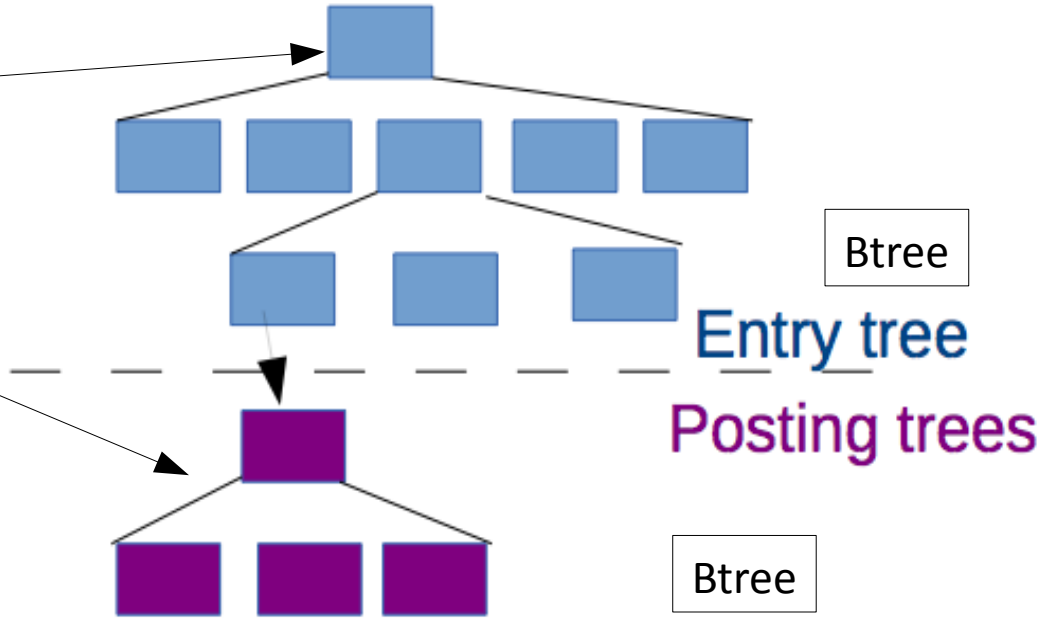
Report Index

A

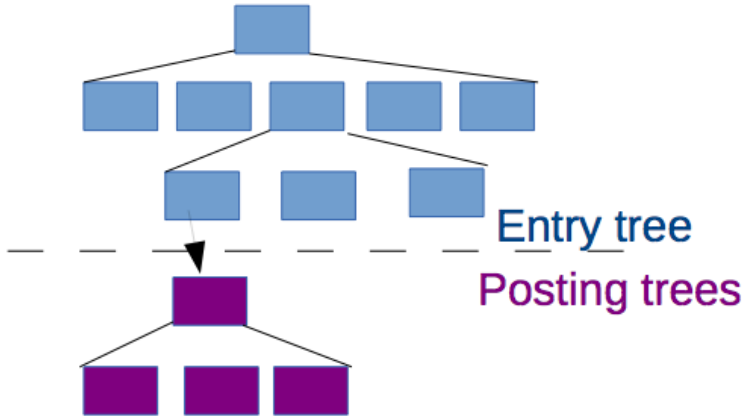
B

D

E



Inverted Index



Report Index

A

B

D

E

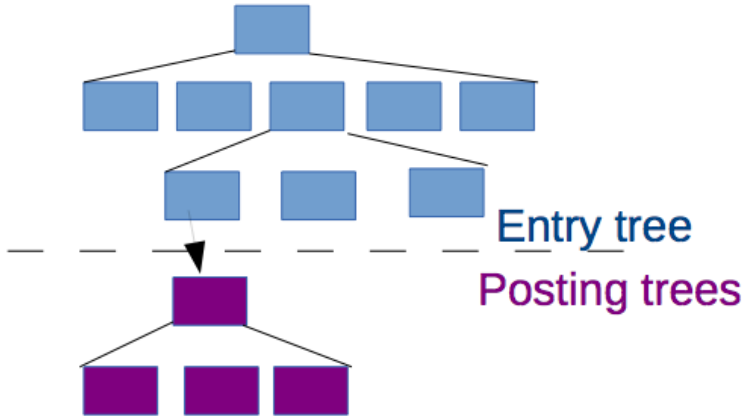
QUERY: compensation accelerometers

INDEX: accelerometers 5,10,25,28,**30**,36,58,59,61,73,74

compensation **30**,68

RESULT: **30**

GIN improvements



- GIN in 9.4 is greatly improved
  - Posting lists compression (varbyte encoding) — smaller indexes
    - 9.3: always 6 bytes (4 bytes blockNumber , 2 bytes offset): 90 bytes
      (0,8) (0,14) (0,17) (0,22) (0,26) (0,33) (0,34) (0,35) (0,45) (0,47) (0,48) (1,3) (1,4) (1,6) (1,8)
    - 9.4: 1-6 bytes per each item, deltas from previous item: 21 bytes
      (0,8) +6 +3 +5 +4 +7 +1 +1 +10 +2 +1 +2051 +1+2 +2

      SELECT g % 10 FROM generate_series(1,10000000) g; 11Mb vs 58Mb
  - Fast scan of posting lists - «rare & frequent» queries much faster
    - 9.3: read posting lists for «rare» and «frequent» and join them
      Time(frequent & rare) ~ Time(frequent)
    - 9.4: start from posting list for «rare» and skip «frequent» list if no match
      Time(frequent & rare) ~ Time(rare)

Hstore is DEAD ? No !

- How hstore benefits by GIN improvement in 9.4 ?

  GIN stands for Generalized Inverted Index, so virtually all data types, which use GIN, get benefit !

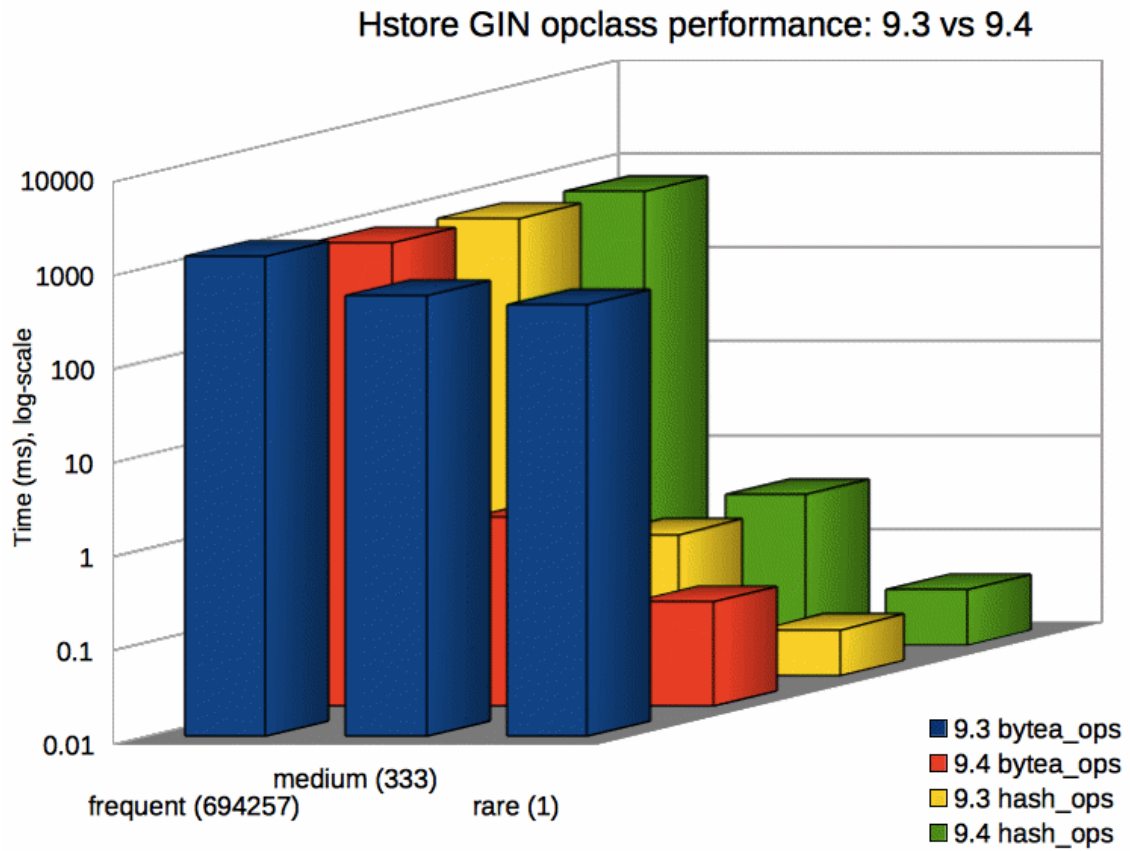
  - Default hstore GIN opclass considers keys and values separately
  - Keys are «frequent», value are «rare»
  - Contains query: hstore @> 'key=>value' improved a lot for «rare» values
  - Index size is smaller, less io

Hstore 9.3 vs 9.4

Total: 7240858 geo records:

"fcode"=>"RFSU",
"point"=>"(8.85,112.53333)",
 "fclass"=>"U",
"asciiname"=>"London Reefs",
"elevation"=>NULL,
"geonameid"=>"1879967",
"population"=>"0"

Query:
SELECT count(*) FROM geo
WHERE geo @> 'fcode=>STM';



opclass	frequent (694257)	medium (333)	rare (1)
9.3 bytea_ops	1353.844	511.196	402.662
9.4 bytea_ops	878.875	1.031	0.13
9.3 hash_ops	755.458	0.321	0.031
9.4 hash_ops	687.626	0.4	0.039

gin_hstore_ops: index keys and values
gin_hstore_bytea_ops = gin_hstore_ops, no collation comparison
gin_hstore_hash_ops: index hash(key.value)

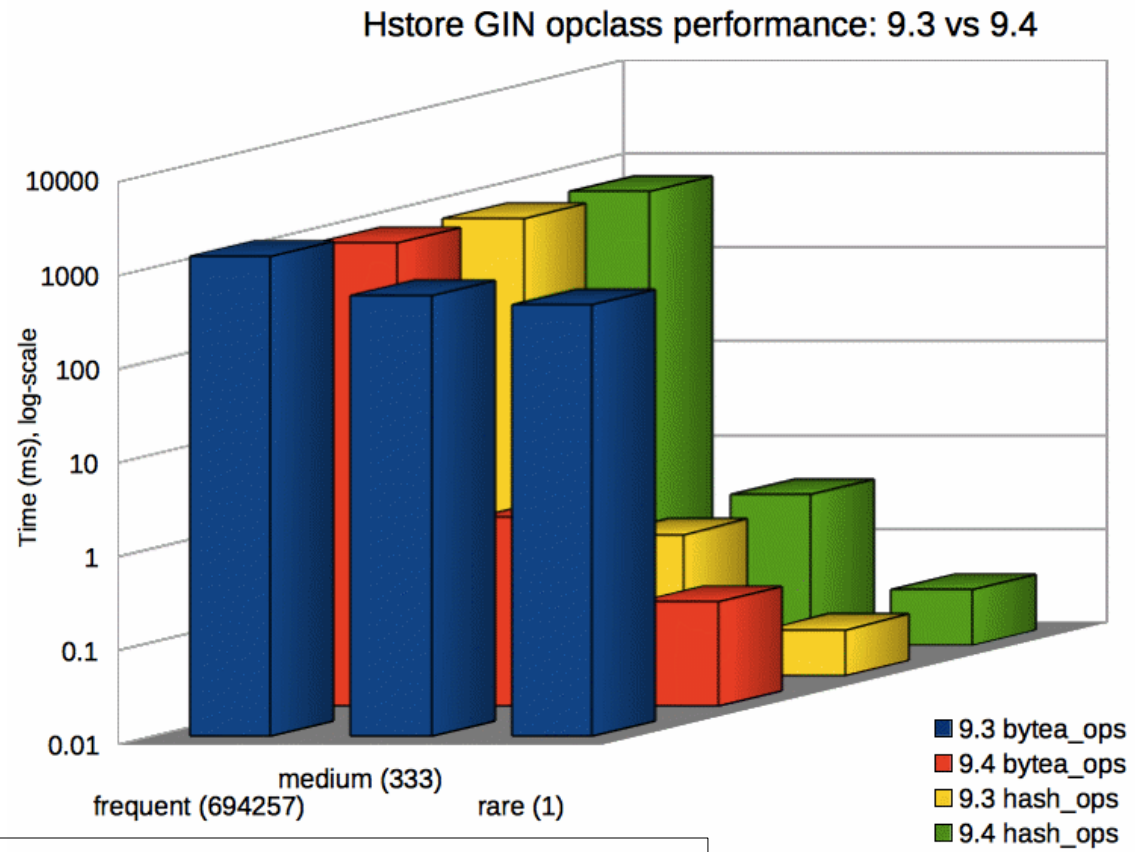
Hstore 9.3 vs 9.4

9.3

Name	Type	Owner	Table	Size
geo	table	postgres		1352 MB
geo_hstore_bytea_ops	index	postgres	geo	1680 MB
geo_hstore_hash_ops_idx	index	postgres	geo	1073 MB

9.4

Name	Type	Owner	Table	Size
geo	table	postgres		1352 MB
geo_hstore_bytea_ops	index	postgres	geo	1296 MB
geo_hstore_hash_ops_idx	index	postgres	geo	925 MB



CREATE OPERATOR CLASS gin_hstore_bytea_ops FOR TYPE hstore
.....
FUNCTION 1 byteacmp(bytea,bytea),
.....
STORAGE bytea;
CREATE INDEX: 239 s Much faster comparison (no collation)

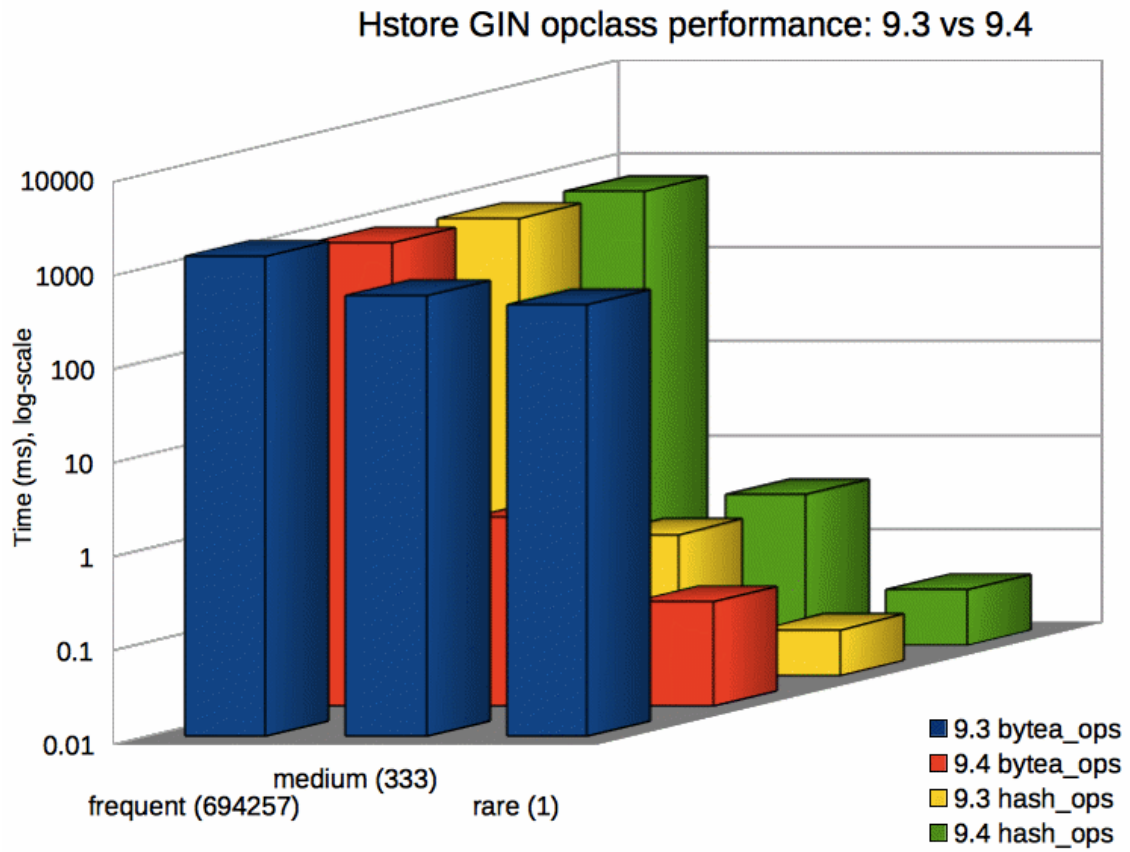
CREATE OPERATOR CLASS gin_hstore_ops FOR TYPE hstore
.....
FUNCTION 1 bttextcmp(text,text),,
.....
STORAGE text;
CREATE INDEX: 2870 s

Hstore 9.3 vs 9.4

SUMMARY:

- 9.4 GIN posting list compression:
  indexes are smaller
- 9.4 GIN is smart regarding 'freq & rare' queries:
  time (freq & rare) ~ time (rare) instead of
  time (freq & rare) ~ time (freq)
- gin_hstore_hash_ops is good on 9.3 & 9.4 and faster default gin opclass
- Use gin_hstore_bytea_ops instead of default gin_hstore_ops — much faster create index

Get hstore_ops from:
from https://github.com/akorotkov/hstore_ops



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));
```

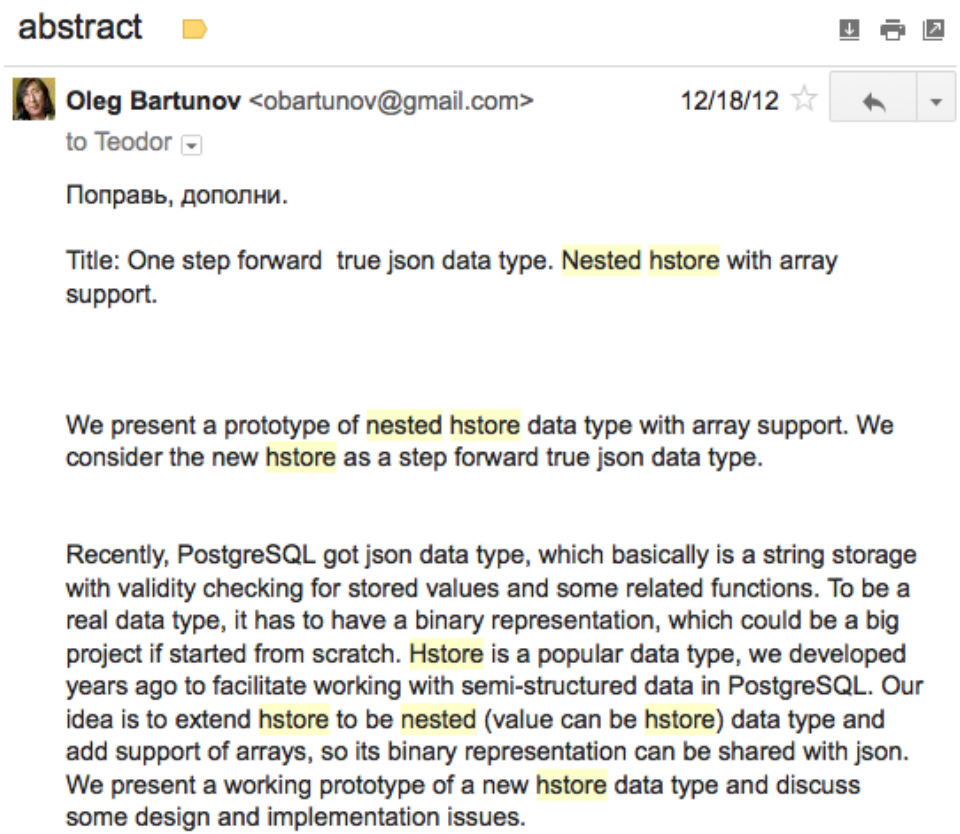
```
SELECT sum((v->'a')::text::int) FROM json_test;
851.012 ms

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 !

PROFESSIONAL Postgres

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

Engine Yard™

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")
```

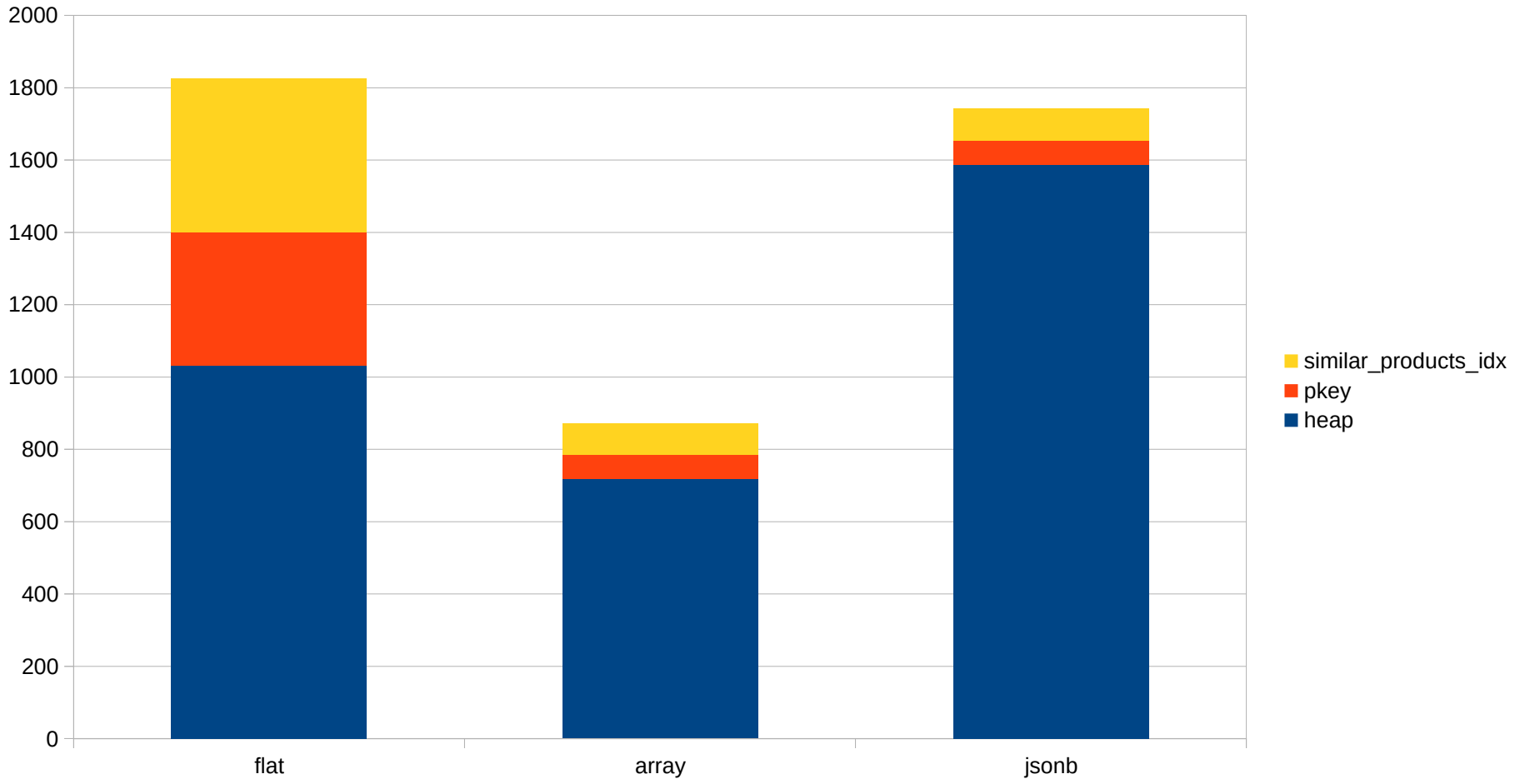
14 168 514 rows

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")
```

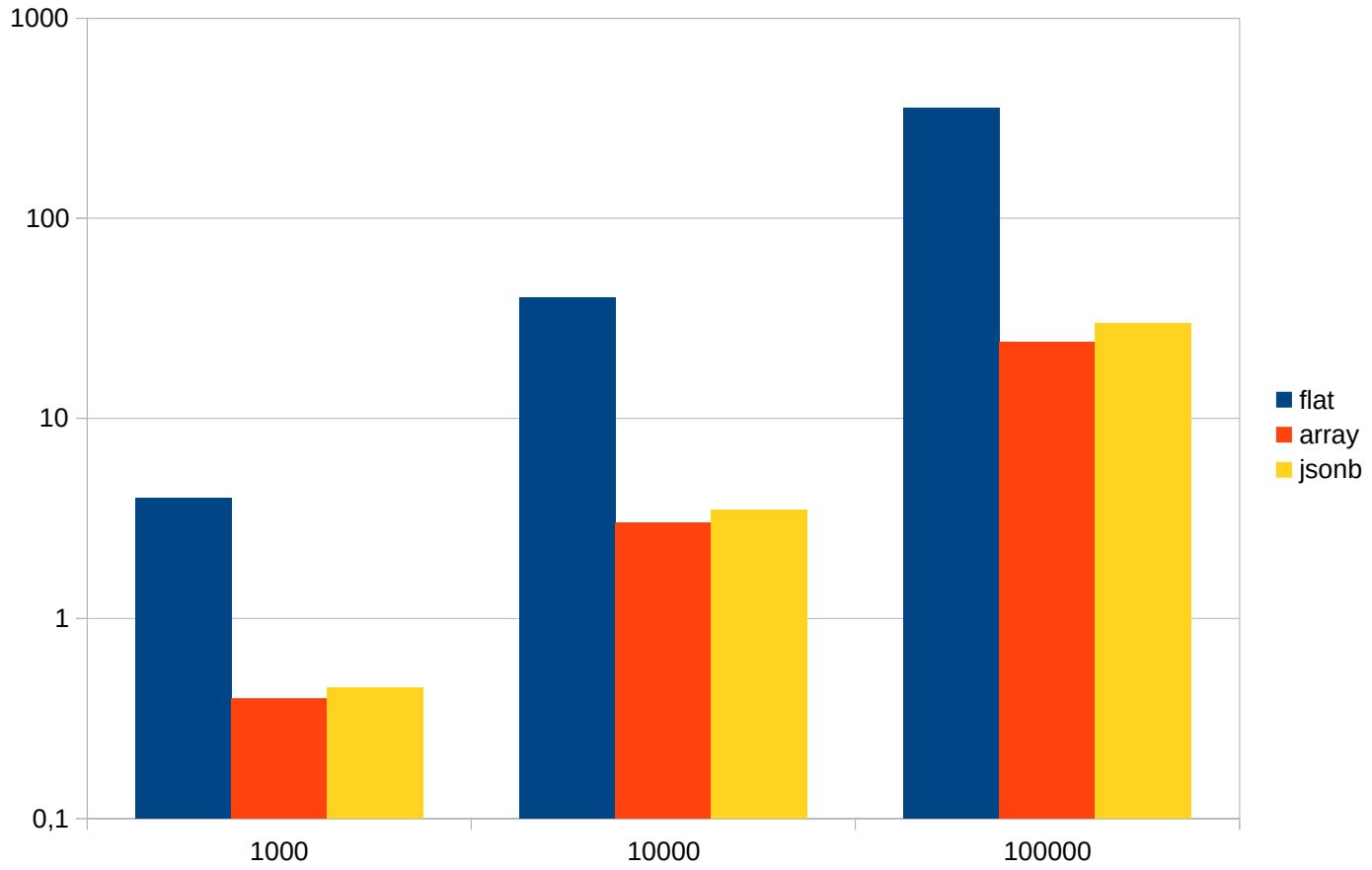
Citus dataset: storage size



Citus dataset: storage conclusion

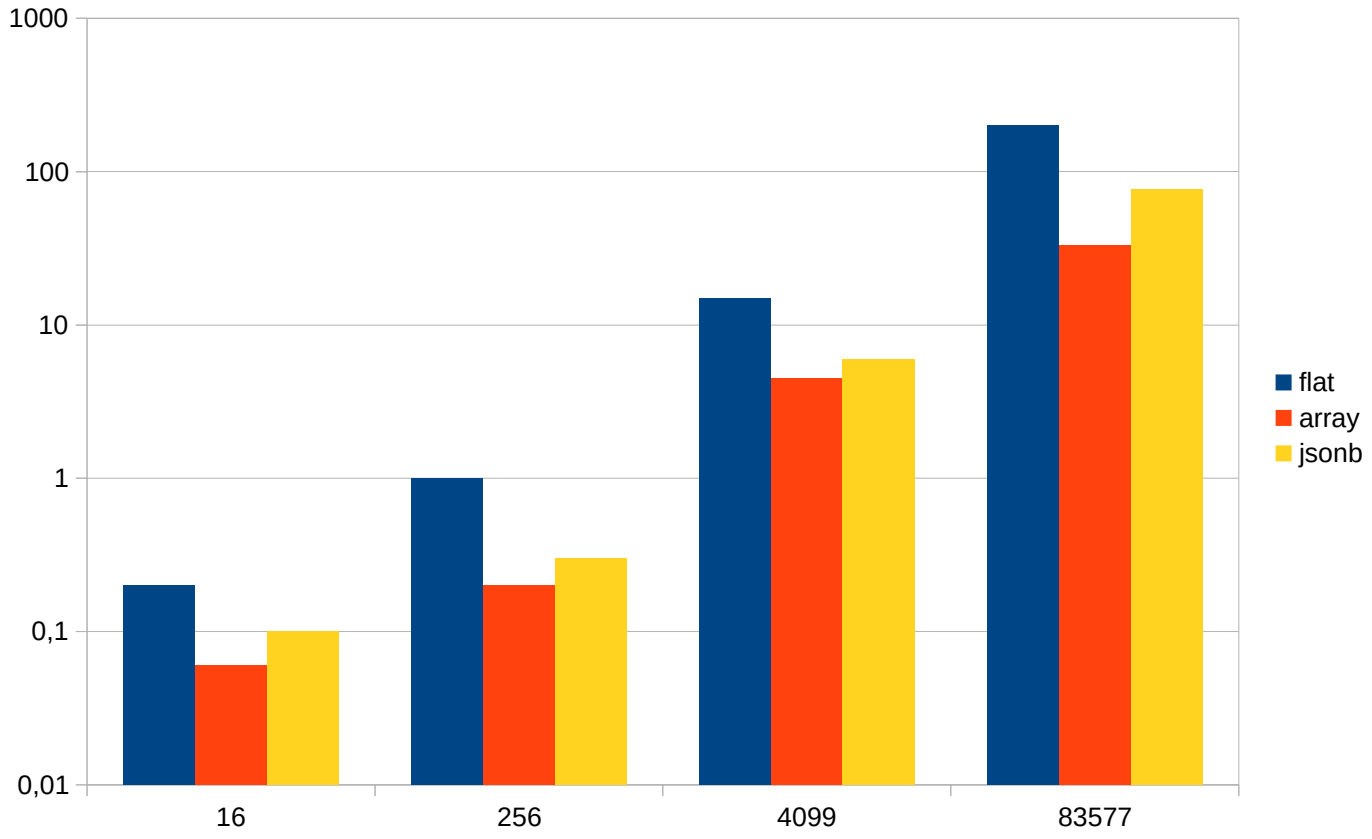
- Array storage is most compact. It doesn't have to store keys. Storage of similar ids as an array is very space efficient as well. However, it's not as flexible as jsonb.
- GIN handles duplicates very efficient.
- Jsonb and flat versions have about same size. Each version has its own overhead. Jsonb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

Builtin 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"}]}';
```

Builtin 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: 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)

- #: - every element in array

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

- %: - every key in object

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

- *: - everything

```
SELECT '{"a": {"b": [1,2,3]}}'::jsonb @@
                                    '*:($ IS OBJECT OR $ IS NUMERIC)';
```

- @# – array or object length

```
select '{"a": {"b": [1,2,3]}}'::jsonb @@ 'a.b.@# > 0';
```

```
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
```

IS BOOLEAN

IS NUMERIC

IS ARRAY

IS OBJECT

IS STRING

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

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
                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
```

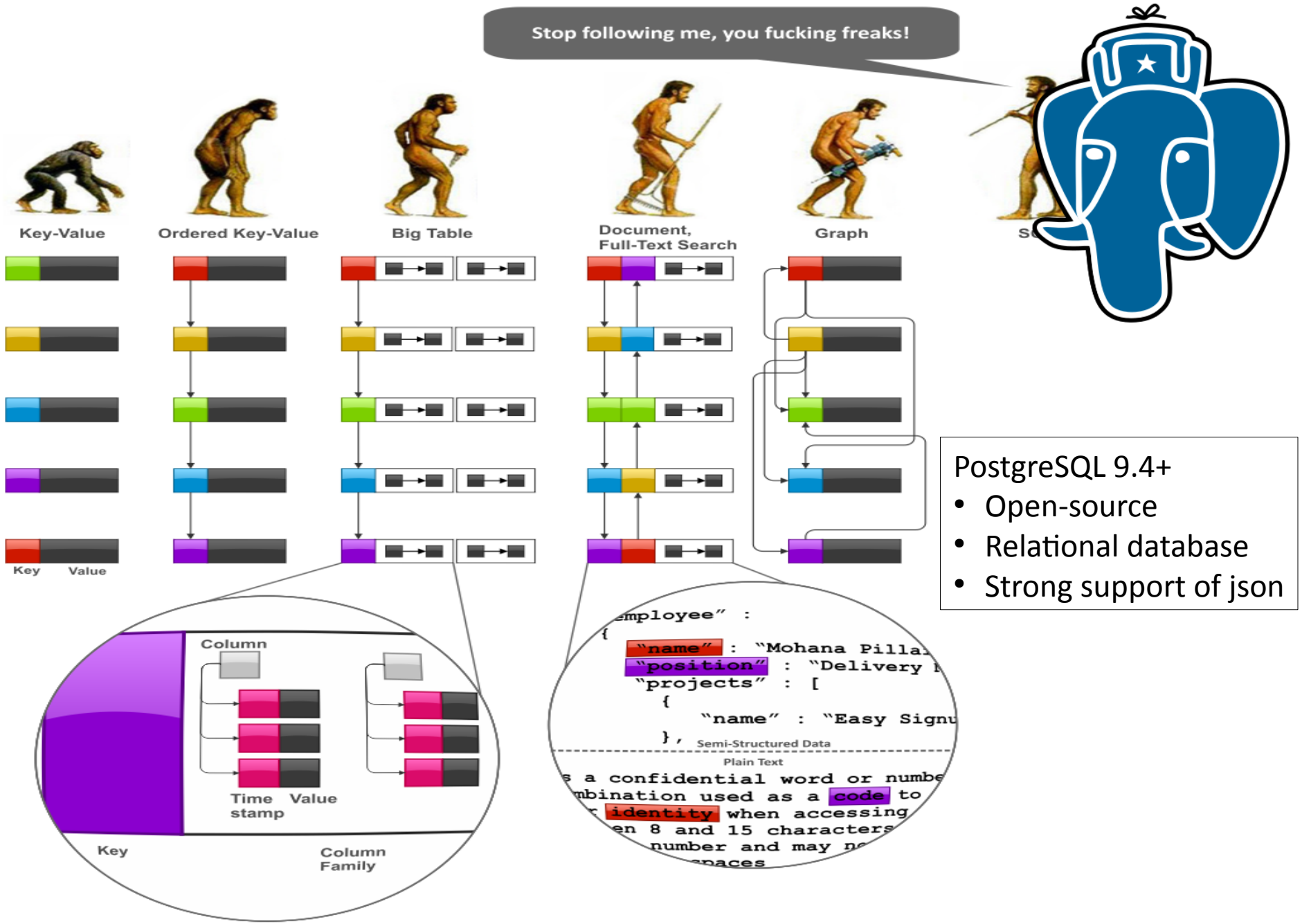
- ```
  # INSERT INTO js (v) VALUES ('{"name": 1, "coords": [{"x": 1,
  "y": 2}, {"x": "3", "y": 4}]}');
  ERROR:  new row for relation "js" violates check constraint
  "js_v_check"
  ```

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

PROFESSIONAL Postgres



PostgreSQL 9.4+

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

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!

Querying problem isn't new!

We already have similar problems with:

- Arrays
- Hstore

How to search arrays by their elements?

```
# CREATE TABLE t AS (SELECT
array_agg((random()*1000)::int) AS a FROM
generate_series(1,1000000) i GROUP BY (i-1)/10);
# CREATE INDEX t_idx ON t USING gin(a);

Simple query
# SELECT * FROM t WHERE 10 = ANY(a); -- ANY(a) = 10 ?
 Seq Scan on t
   Filter: (10 = ANY (a))
```

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!**

Array indexing support

Expression	gin (i)	gist (i)
i && '{1,2}', i@> '{1,2}'	+	+
i @@ '1&(2|3)'	+	+
1 = ANY(i)	–	–
1 < ANY(i)	–	–
Subselects	–	–

Array indexing support desired: add support of complex expressions

Expression	gin (i)	gist (i)
i && '{1,2}', i@> '{1,2}'	+	+
i @@ '1&(2|3)'	+	+
1 = ANY(i)	+	+
1 < ANY(i)	+	–
Subselects	+	+

How to search hstore?

```
# CREATE TABLE t AS (SELECT hstore(array_agg('k'||
(random()*10)::int::text), array_agg('v'||
(random()*100)::int::text)) h FROM
generate_series(1,1000000) g GROUP BY (g-1)/10);

# CREATE INDEX t_idx ON t USING gin(h);

# SELECT * FROM t WHERE h->'k0' = 'v0';
 Seq Scan on t
   Filter: ((h -> 'k0'::text) = 'v0'::text)
```

How to search hstore?

```
# SELECT * FROM t WHERE h @> '"k0"=>"v0"';
 Bitmap Heap Scan on t
   Recheck Cond: (h @> '"k0"=>"v0"'::hstore)
   ->  Bitmap Index Scan on t_idx
         Index Cond: (h @> '"k0"=>"v0"'::hstore)
```

@> operator can be used for index on hstore

How to search hstore?

What about btree indexes?

```
# DROP INDEX t_idx;
# CREATE INDEX t_idx ON t ((h->'k0'));

# SELECT * FROM t WHERE h @> '"k0"=>"v0"';
 Seq Scan on t
   Filter: (h @> '"k0"=>"v0"'::hstore)
```

Indexed expression should exactly match.
It should be in the form: h->'k0' opr value

How to search hstore?

```
# SELECT * FROM t WHERE h->'k0' = 'v0';
 Index Scan using t_idx on t
   Index Cond: ((h -> 'k0'::text) = 'v0'::text)

# SELECT * FROM t WHERE h->'k0' < 'v0';
 Index Scan using t_idx on t
   Index Cond: ((h -> 'k0'::text) < 'v0'::text)
```

Index is used because expressions exactly match.
You also use <, > etc which can't be expressed by @>.

Hstore index support

Expression	gin (h)	gist (h)	btree ((h->'key'))
h @> '"key" => "value"'	+	+	–
h -> 'key' = 'value'	–	–	+
h -> 'key' > 'value'	–	–	+

SQL becomes not so declarative: you specify what index query could use by the form of expression.

Hstore index support desired: add support of complex expressions

Expression	gin (h)	gist (h)	btree ((h->'key'))
h @> '"key" => "value"'	+	+	?
h -> 'key' = 'value'	+	+	+
h -> 'key' > 'value'	+	–	+

SQL becomes not so declarative: you specify what index query could use by the form of expression.

Jsonb indexing: cumulative problems of arrays and hstore

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

Another syntax idea: MAP/FILTER

```
SELECT … FROM tbl WHERE
array_length( WITH elem FILTER ( elem < 4 OR elem > 10 )
tbl.array_col ) > 2;

SELECT … FROM tbl WHERE
array_length( WITH elem FILTER ( elem < 4 OR elem > 10 )
              WITH elem MAP ( elem * 2 ) tbl.array_col ) > 2;
```

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);
```

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');
```

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')));
```

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 syntaxic 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));
```

How could we extend index usage?

- Opclass specify some grammar of index accelerated expressions.
- Now this grammar is trivial: limited set of «col opr value» expressiona.
- Need some better grammar to support wider set of expressions.

Use a-grammar for opclasses

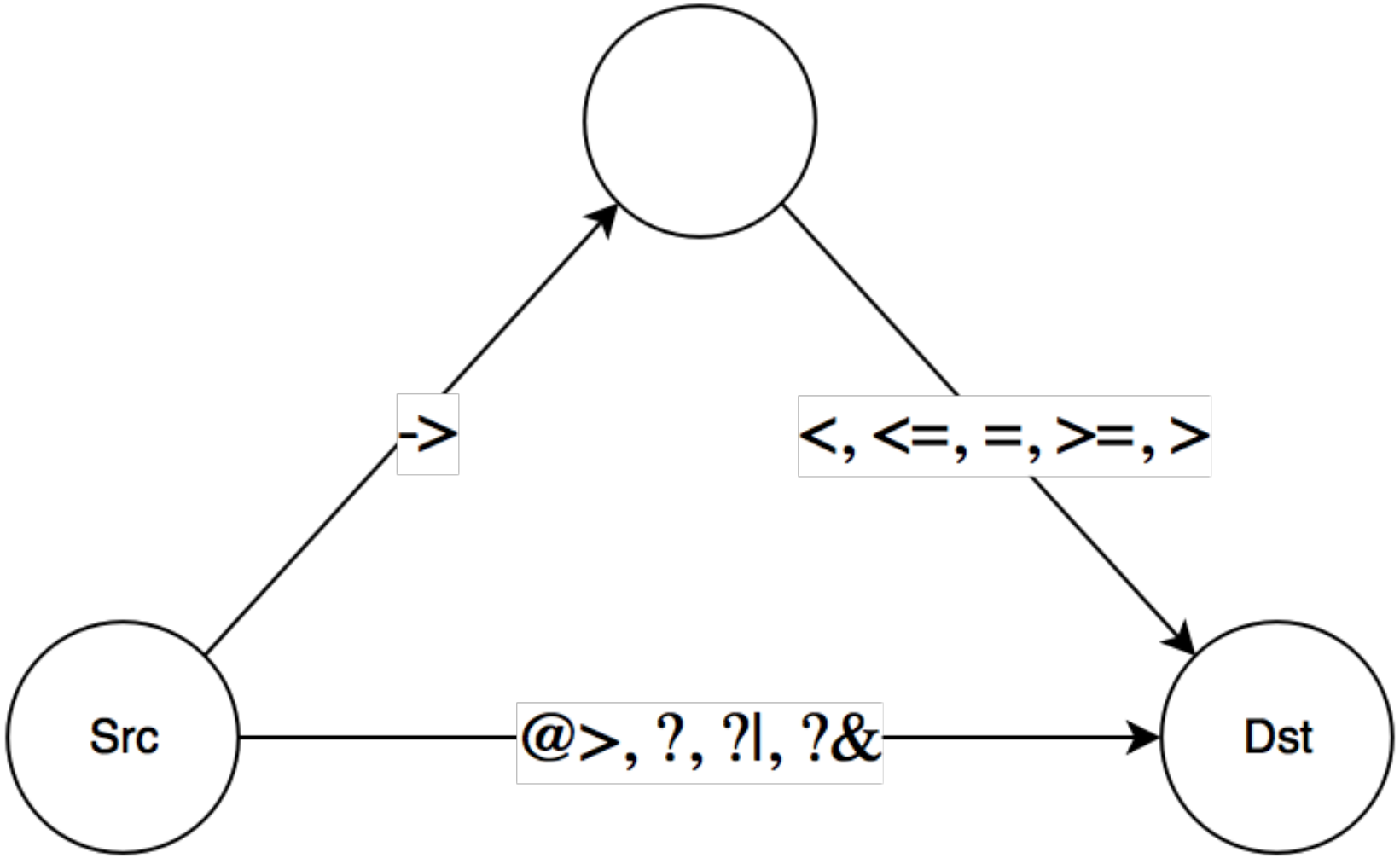
Automaton could be specified in opclass.

```
# \d pg_amop
       Table "pg_catalog.pg_amop"
     Column       |   Type   | Modifiers
------------------+----------+-----------
 amopfamily       | oid      | not null
 amoplefttype     | oid      | not null
 amoprighttype    | oid      | not null
 amopstrategy     | smallint | not null
 amoppurpose      | "char"   | not null
 amopopr          | oid      | not null
 amopmethod       | oid      | not null
 amopsortfamily   | oid      | not null
 amopsource       | smallint | not null
 amopdestination  | smallint | not null
```

New opclass format: hstore

```
CREATE OPERATOR CLASS gin_hstore_ops DEFAULT FOR TYPE hstore USING gin AS
OPERATOR        1       ->(hstore, text) 1 2,
OPERATOR        2       <(text, text) 2 0,
OPERATOR        3       <=(text, text) 2 0,
OPERATOR        4       =(text, text) 2 0,
OPERATOR        5       >=(text, text) 2 0,
OPERATOR        6       >(text, text) 2 0,
OPERATOR        7       @>,
OPERATOR        9       ?(hstore,text),
OPERATOR        10      ?|(hstore,text[]),
OPERATOR        11      ?&(hstore,text[]),
FUNCTION        1       bttextcmp(text,text),
FUNCTION        2       gin_extract_hstore(internal, internal),
FUNCTION        3       gin_extract_hstore_query(internal, internal, int2, interna
FUNCTION        4       gin_consistent_hstore(internal, int2, internal, int4, inte
STORAGE         text;
```

New opclass format: hstore



There is a lot of work!

- Changes in system catalog.
- Changes in planner to support complex expressions for indexes.
- Changes in access method interface. Array of ScanKeys isn't expressive enough! Need something called «ScanTrees».
- Changes in operator classes functions interface. «ScanTrees» should be passed into opclasses.

Postgres PROFESSIONAL

Thanks for support

WORLD OF TANKS

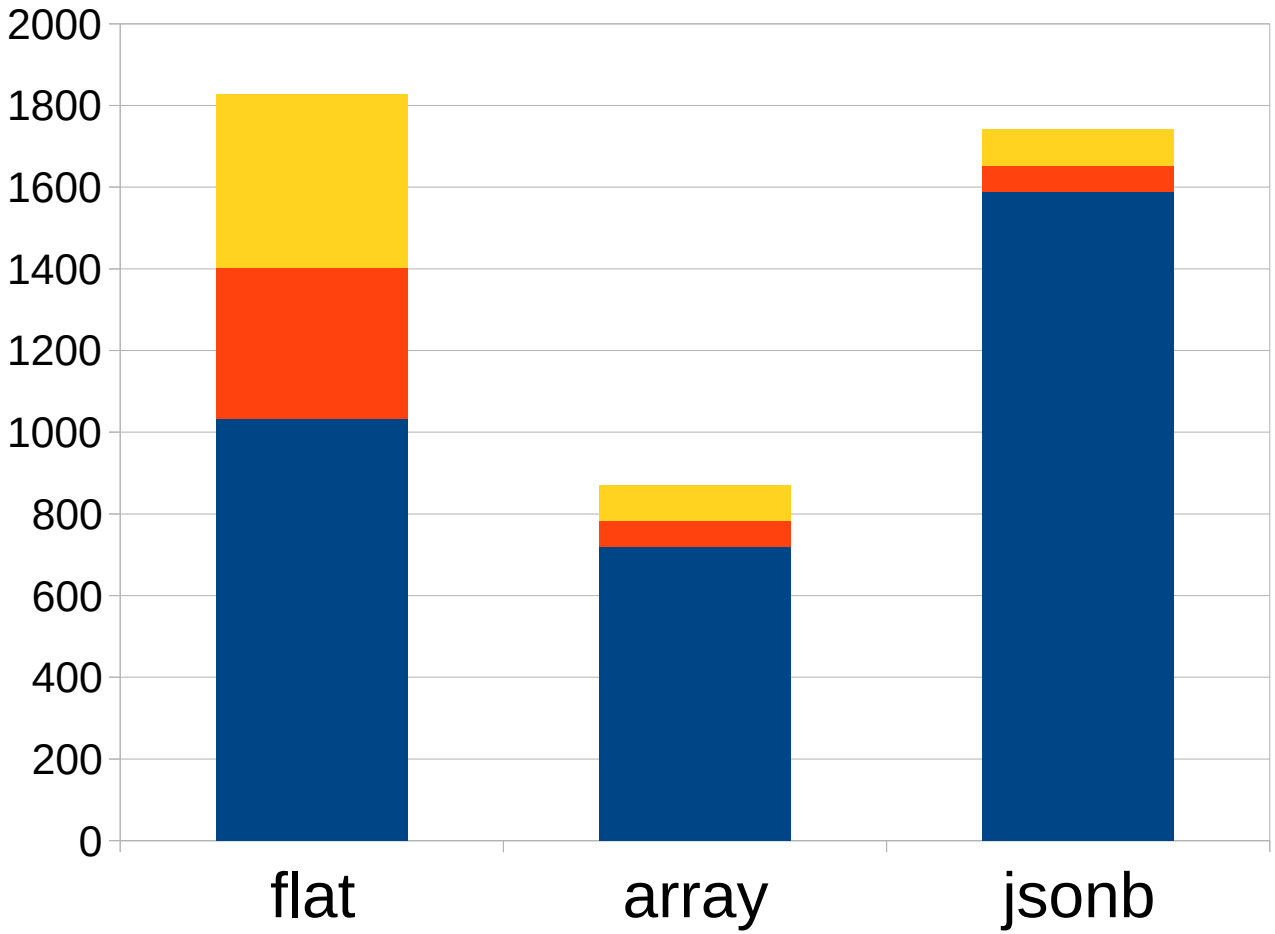
WARGAMING.NET

LET'S BATTLE

PROFESSIONAL Postgres

JSONB format compression

Citus dataset: storage size



Jsonb vs array overhead is huge!

Binary format of jsonb

```
{"product_id": "0613225783", "review_votes": 0, "product_group": "Book",
"product_title": "Walk in the Woods", "review_rating": 3, "product_sales_rank":
482809, "review_helpful_votes": 0}
```

```
07 00 00 20 0a 00 00 80  0c 00 00 00 0d 00 00 00  |... ............| Headers
0d 00 00 00 0d 00 00 00  12 00 00 00 14 00 00 00  |................|
0a 00 00 00 09 00 00 10  04 00 00 00 11 00 00 00  |................|
09 00 00 10 0a 00 00 10  08 00 00 10 70 72 6f 64  |............prod|
75 63 74 5f 69 64 72 65  76 69 65 77 5f 76 6f 74  |uct_idreview_vot| Key names
65 73 70 72 6f 64 75 63  74 5f 67 72 6f 75 70 70  |esproduct_groupp|
72 6f 64 75 63 74 5f 74  69 74 6c 65 72 65 76 69  |roduct_titlerevi|
65 77 5f 72 61 74 69 6e  67 70 72 6f 64 75 63 74  |ew_ratingproduct|
5f 73 61 6c 65 73 5f 72  61 6e 6b 72 65 76 69 65  |_sales_rankrevie|
77 5f 68 65 6c 70 66 75  6c 5f 76 6f 74 65 73 30  |w_helpful_votes0|
36 31 33 32 32 35 37 38  33 00 00 00 18 00 00 00  |613225783.......| Data
00 80 42 6f 6f 6b 57 61  6c 6b 20 69 6e 20 74 68  |..BookWalk in th|
65 20 57 6f 6f 64 73 00  20 00 00 00 00 80 03 00  |e Woods. .......|
28 00 00 00 01 80 30 00  f9 0a 00 00 18 00 00 00  |(.....0.........|
00 80                                             |..              |
```

Jsonb compression idea 1

- Maintain dictionary of keys
- Compress headers using varbyte encoding
- Compress small numbers using varbyte encoding

Cons:

- Maintain dictionary of schemas

Jsonbc extension: dictionary

```
CREATE TABLE jsonbc_dict
(
    id serial PRIMARY KEY,
    name text NOT NULL,
    UNIQUE (name)
);

extern int32 getIdByName(KeyName name);
extern KeyName getNameById(int32 id);
```

Binary format of jsonbc

```
{"product_id": "0613225783", "review_votes": 0, "product_group": "Book",
"product_title": "Walk in the Woods", "review_rating": 3, "product_sales_rank":
482809, "review_helpful_votes": 0}
```

```
3d 01 51 04 0b 01 21 01  89 01 01 0b 02 1b 03 0b  |=.Q...!.........| Headers
30 36 31 33 32 32 35 37  38 33 00 42 6f 6f 6b 57  |0613225783.BookW| Data
61 6c 6b 20 69 6e 20 74  68 65 20 57 6f 6f 64 73  |alk in the Woods|
06 f2 f7 3a 00                                    |...:.           |
```

Jsonbc extension

Customers reviews different format storage comparison

```
customer_reviews_jsonb          307 MB
customer_reviews_jsonbc         123 MB
customer_reviews_array          139 MB
```

Less than array because of numerics compression!

Jsonb compression idea 2

- Extract everything except raw data into schema
- Maintain dictionary of schemas
- Store only raw data and schema reference

Cons:

- Maintain dictionary of schemas
- There could be as many schemas as documents

Idea 2 estimation

Customers reviews different format storage comparison

```
customer_reviews_jsonb          307 MB
customer_reviews_jsonbc         123 MB
customer_reviews_jsonb2         106 MB (Estimation!)
customer_reviews_array          139 MB
```

16%
Does it worth the effort?

Postgres PROFESSIONAL

Спасибо за внимание!

Какие вопросы?

У нас на всё есть ответы!