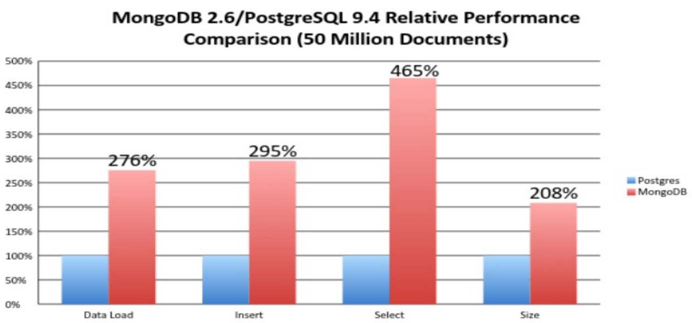
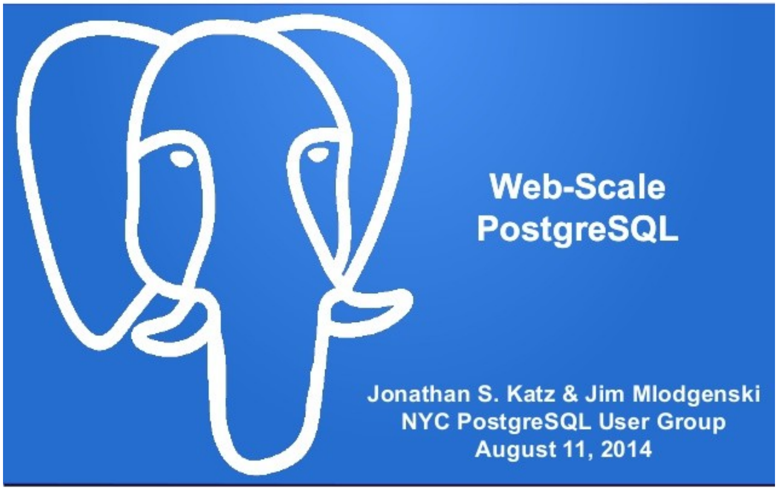
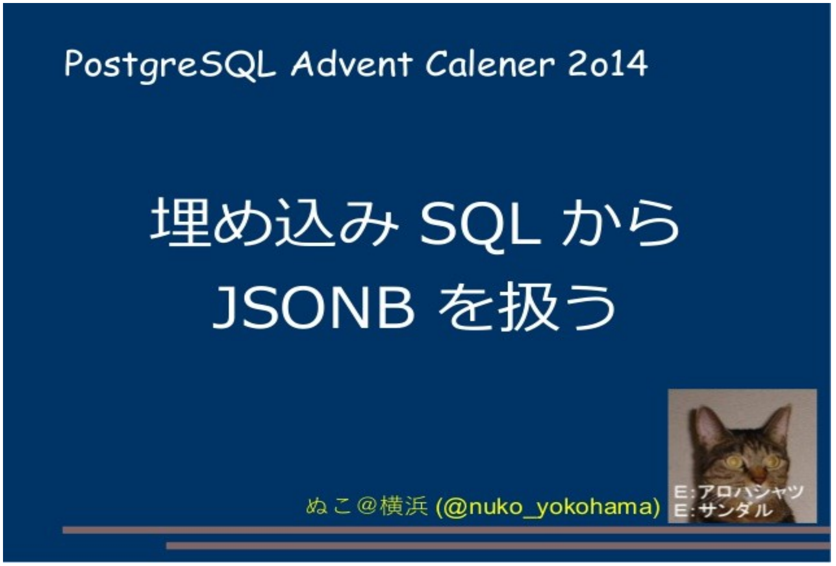
2015.secon.ru, Пенза





	Postgres	MongoDB
Data Load (s)	4,732	13,046
Insert (s)	29,236	86,253
Select (s)	594	2,763
Size (GB)	69	145



Postgres' NoSQ
Capabilities

- HSTORE
  - Key-value pair
  - Simple, fast and easy
  - Postgres v 8.2 – pre-dates many NoSQL-only solutions
  - Ideal for flat data structures that are sparsely populated
- JSON
  - Hierarchical document model
  - Introduced in Postgres 9.2, perfected in 9.3
- JSONB
  - Binary version of JSON
  - Faster, more operators and even more robust
  - Postgres 9.4

JSONB Features

- Equality operator
  - SELECT '{"a": 1, "b": 2}'::jsonb = '{"b":2, "a":1}'::jsonb
- Containment operator (Softserve)
  - SELECT '{"a": 1, "b": 2}'::jsonb @> {"b":2}::jsonb
- Existence
  - SELECT '{"a": 1, "b": 2}'::jsonb ? 'b';

serve works as well)
}'::jsonb = '{"a":[1,2]}'::jsonb

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

Agenda

- The problem
- Hstore
- Introduction to jsonb indexing
- Jsquery - Jsonb Query Language
- Exercises on jsonb GIN opclasses with Jsquery support
- VODKA access method

The problem

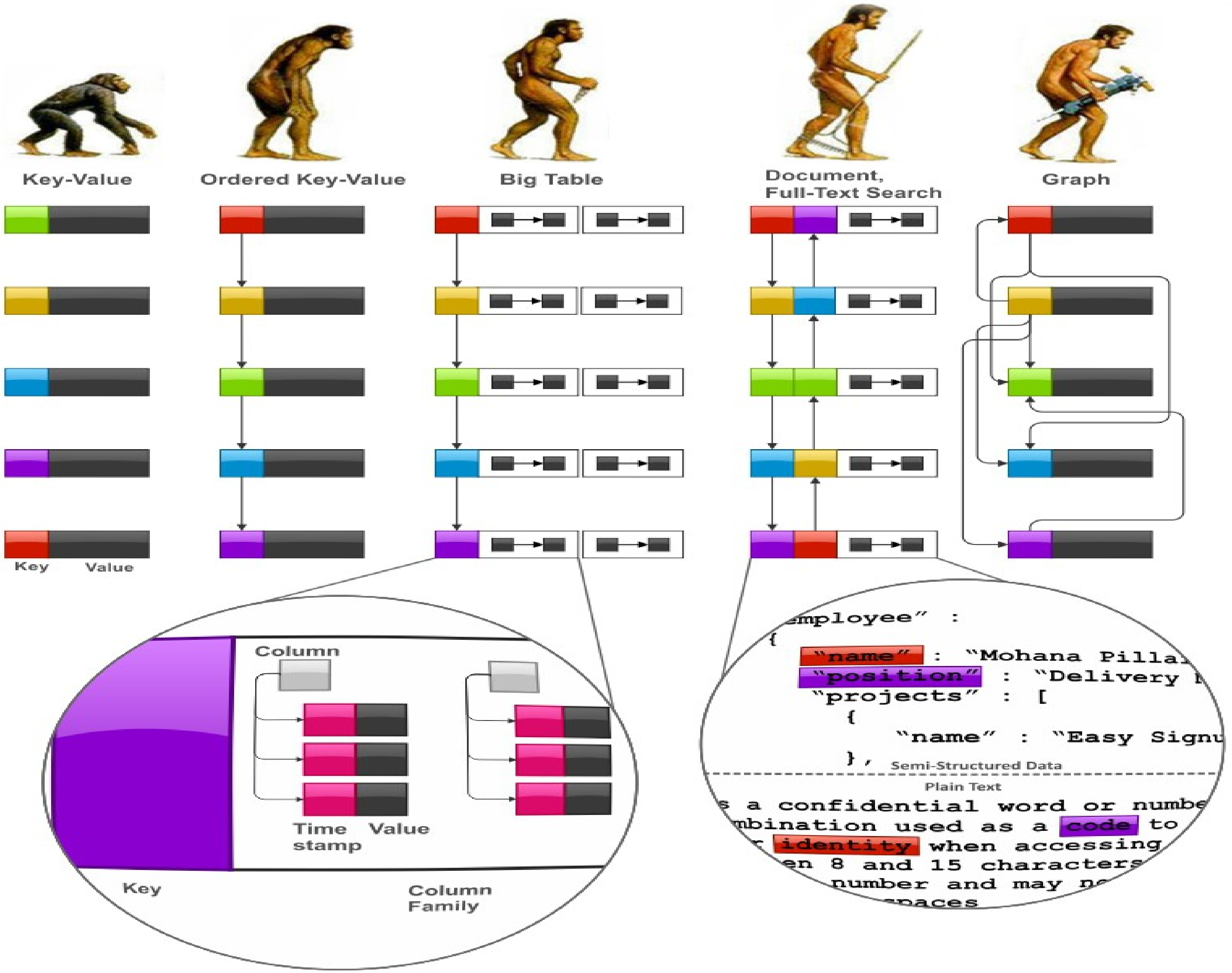
- The world of data and applications is changing
- BIG DATA (**V**olume of data,**V**elocity of data in-out, **V**ariety of data)
- Web applications are service-oriented
  - Service itself can aggregate data, check consistency of data
  - High concurrency, simple queries
  - Simple database (key-value)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

NoSQL (концептуальные предпосылки)

- 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


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

NoSQL

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

NoSQL databases (wikipedia) ...+++

Document store

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

Graph
* Neo4j
* AllegroGraph

Tabular
* BigTable
* Mnesia
* Hbase
* Hypertable

Key/value store on disk

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

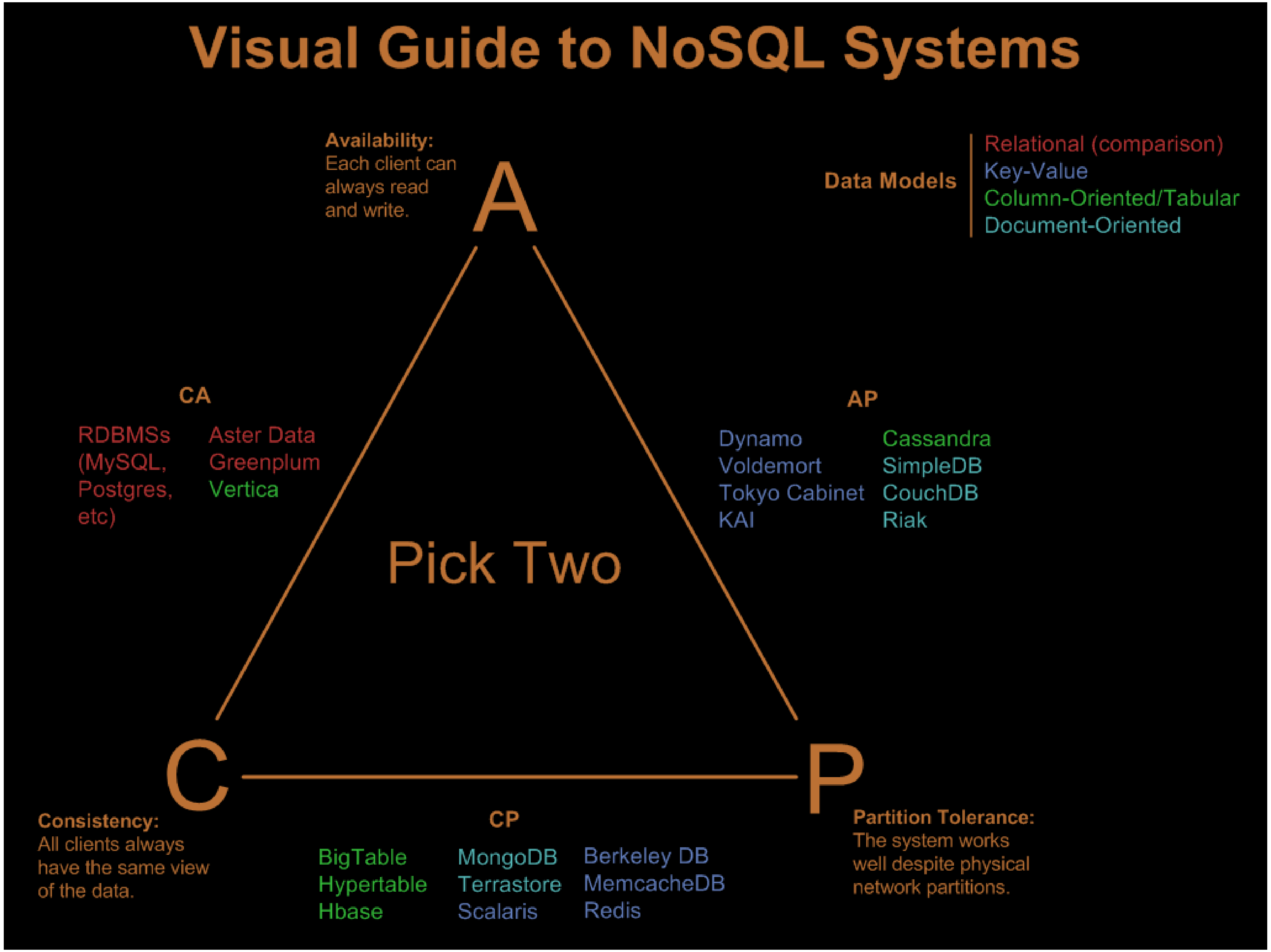
Key/value cache in RAM

* memcached
* Velocity
* Redis

Eventually-consistent key-value store
* Dynamo
* Cassandra
* Project Voldemort

Ordered key-value store
* NMDB
* Luxio
* Memcachedb
* Berkeley DB

Object database
* Db4o
* InterSystems Caché
* Objectivity/DB
* ZODB


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

CAP теорема: Наш вариант

The problem

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

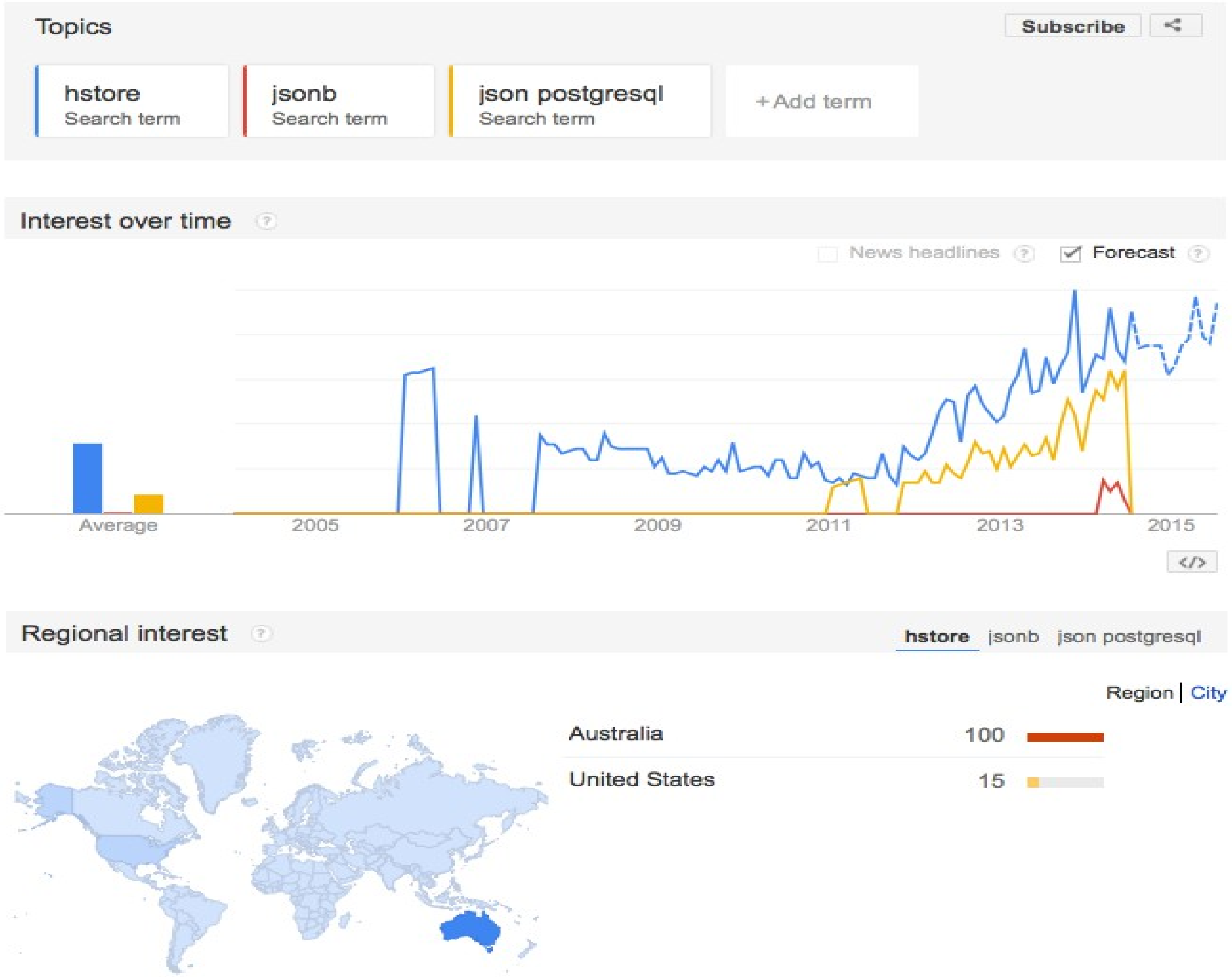
JSON in PostgreSQL

This is the challenge !

Challenge to PostgreSQL !

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

Google insights about hstore



Introduction to Hstore

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

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

Introduction to Hstore

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

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

Introduction to hstore

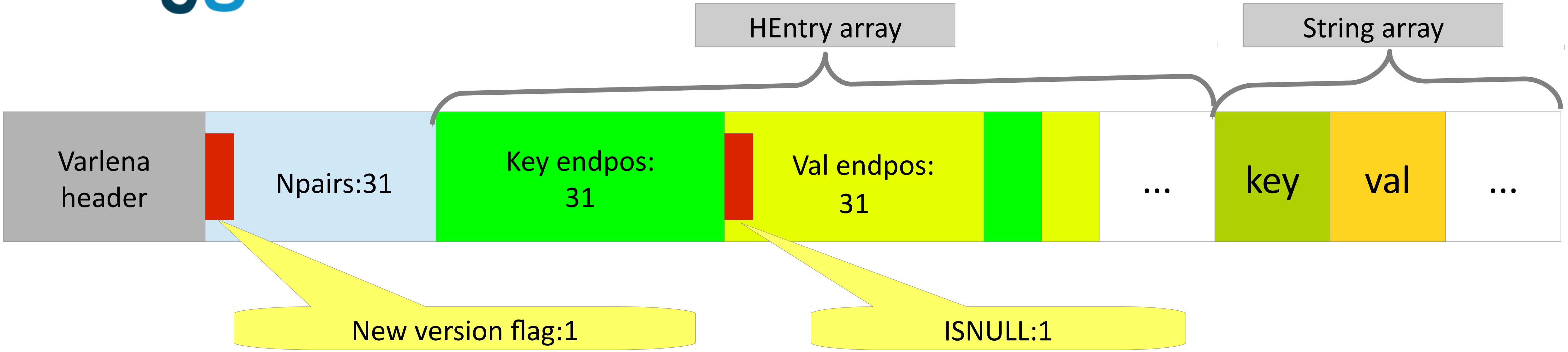
- Hstore — key/value binary storage (inspired by perl hash)

  ```
  '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
```

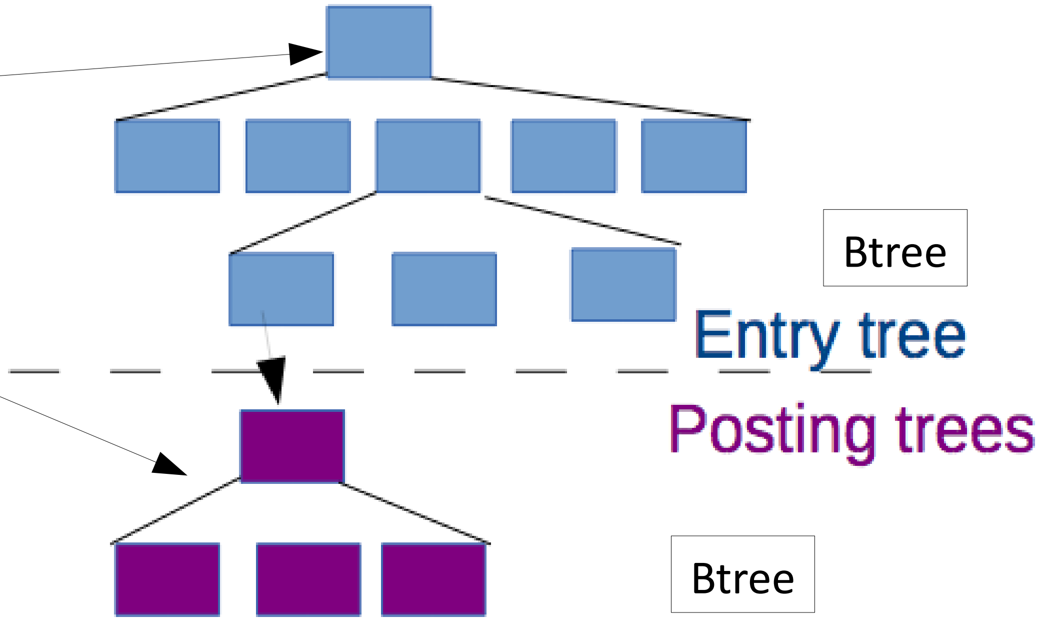
History of hstore development

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

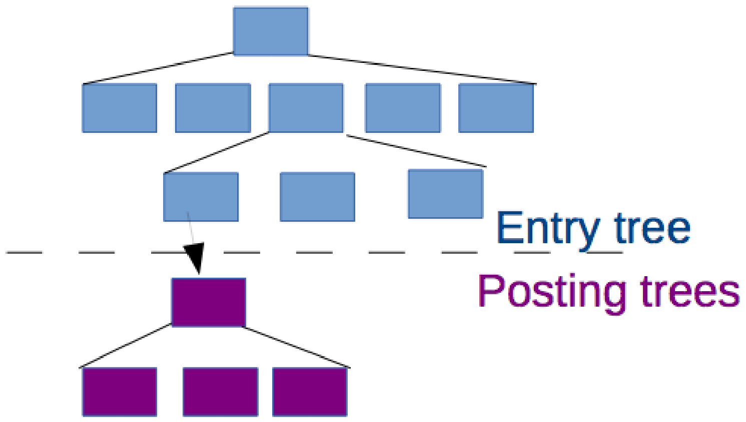
  (thanks, Hubert Depesz Lubaczewski!)
- May 23, 2007 - GIN index for hstore, PostgreSQL 8.3
- Sep, 20, 2010 - Andrew Gierth improved hstore, PostgreSQL 9.0

Inverted Index

Report Index



Inverted Index



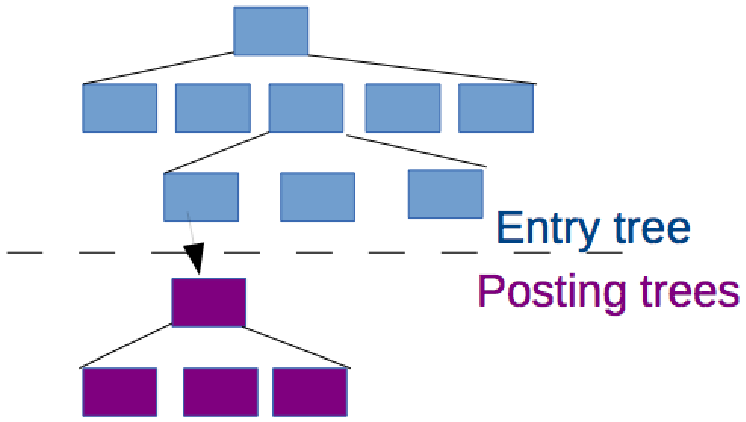
Report Index

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 !

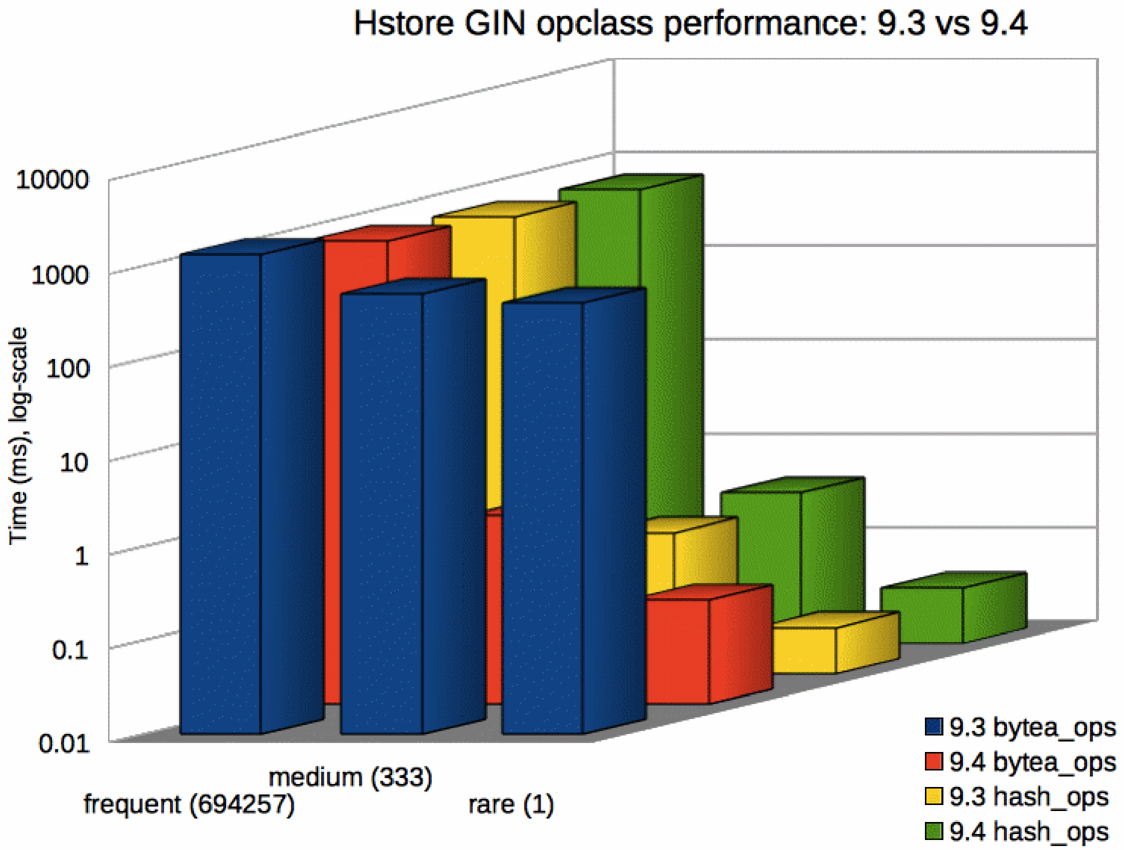
  - Default hstore GIN opclass considers keys and values separately
  - Keys are «frequent», value are «rare»
  - Contains query: hstore @> 'key=>value' improved a lot for «rare» values
  - Index size is smaller, less io

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)

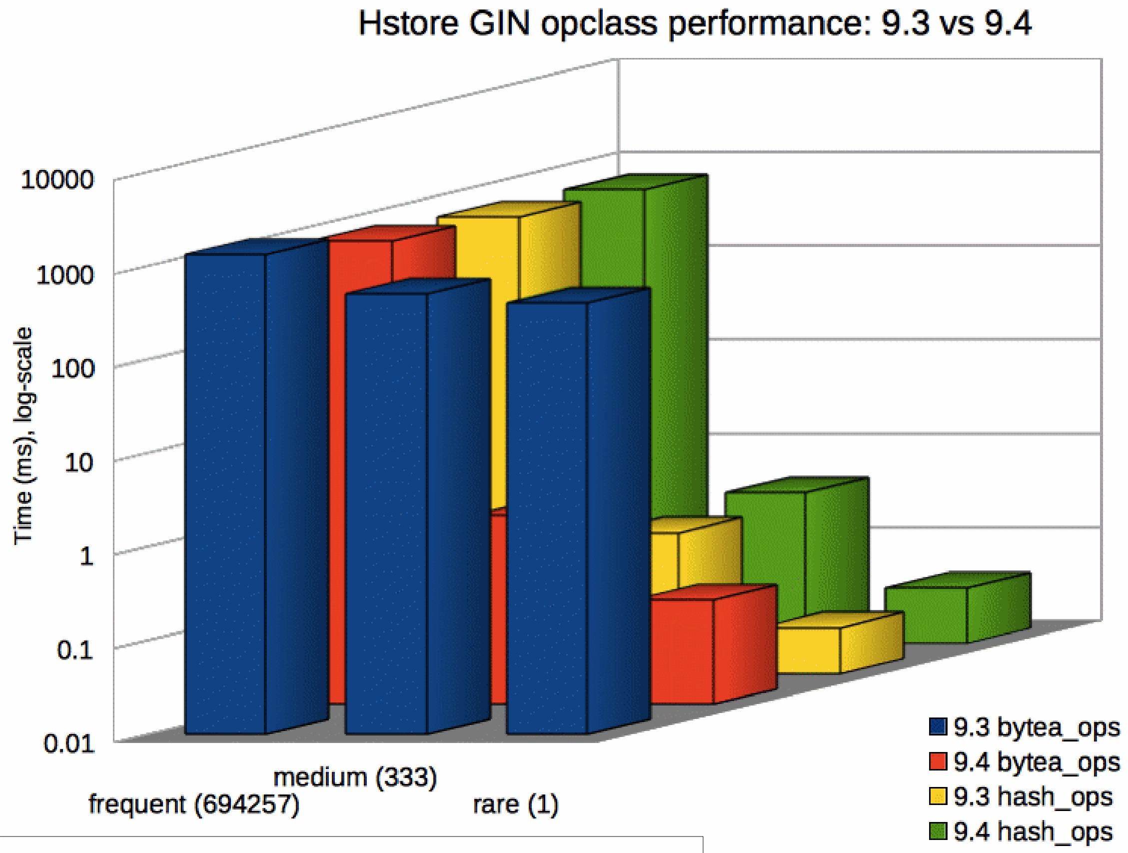
Hstore 9.3 vs 9.4

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)

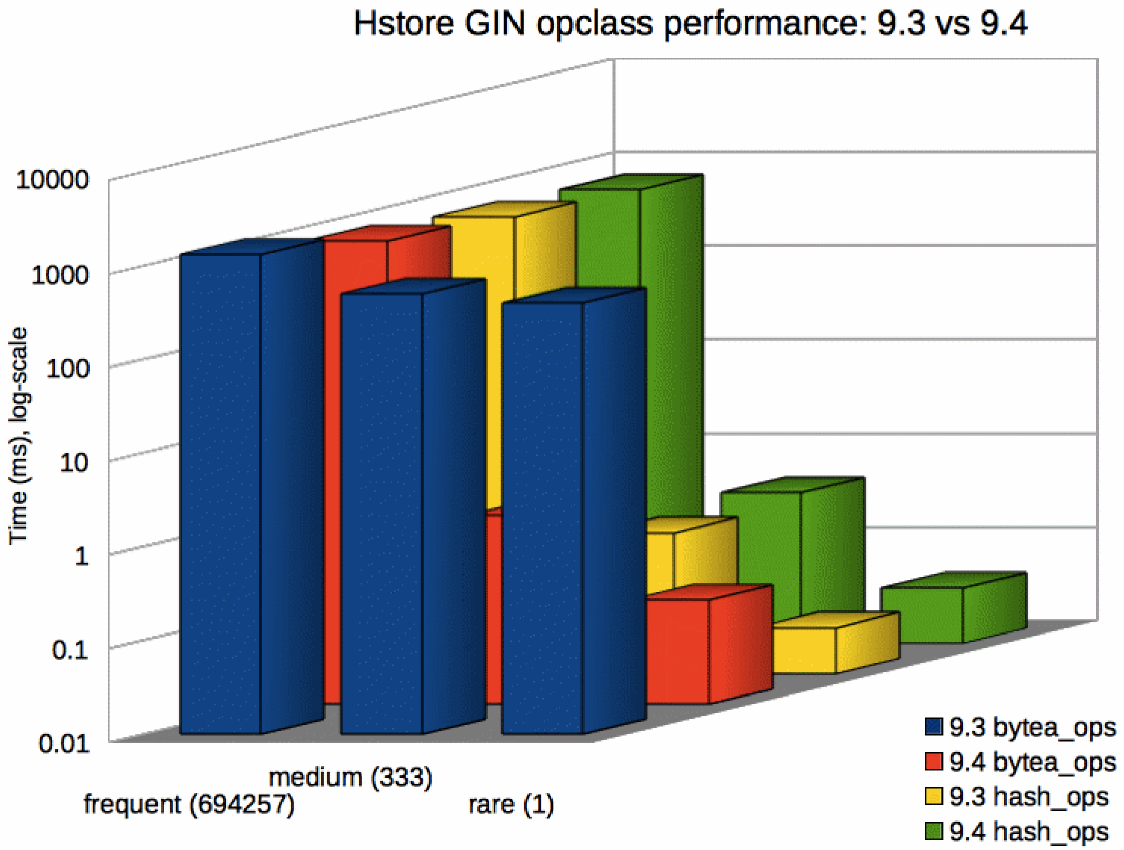
```
CREATE OPERATOR CLASS gin_hstore_ops FOR TYPE hstore
     ........................................................................................
     FUNCTION     1     bttextcmp(text,text),,
     ........................................................................................
STORAGE text;
```
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 !

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

PROFESSIONAL Postgres

Jsonb vs Jso


```
{
    "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"
}
```


- 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

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

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

Что сейчас может Jsonb ?

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

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

  Keys should be specified from root

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

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

  Only root keys supported

- Operators on jsonb parts (functional indexes)

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

  Very cumbersome, too many functional indexes

Найти что-нибудь красное

```
        Table "public.js_test"
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer | not null
 value  | jsonb   |

select * from js_test;

 id |                                 value
----+-----------------------------------------------------------------------
  1 | [1, "a", true, {"b": "c", "f": false}]
  2 | {"a": "blue", "t": [{"color": "red", "width": 100}]}
  3 | [{"color": "red", "width": 100}]
  4 | {"color": "red", "width": 100}
  5 | {"a": "blue", "t": [{"color": "red", "width": 100}], "color": "red"}
  6 | {"a": "blue", "t": [{"color": "blue", "width": 100}], "color": "red"}
  7 | {"a": "blue", "t": [{"color": "blue", "width": 100}], "colr": "red"}
  8 | {"a": "blue", "t": [{"color": "green", "width": 100}]}
  9 | {"color": "green", "value": "red", "width": 100}
(9 rows)
```

Найти что-нибудь красное

```
WITH RECURSIVE t(id, value) AS ( SELECT * FROM
js_test
 UNION ALL
    (
      SELECT
        t.id,
        COALESCE(kv.value, e.value) AS value
      FROM
        t
        LEFT JOIN LATERAL
jsonb_each(
CASE WHEN jsonb_typeof(t.value) =
'object' THEN t.value
          ELSE NULL END) kv ON true
        LEFT JOIN LATERAL
jsonb_array_elements(
          CASE WHEN
jsonb_typeof(t.value) = 'array' THEN t.value
          ELSE NULL END) e ON true
        WHERE
          kv.value IS NOT NULL OR e.value IS
NOT NULL
     )
)
```

```
SELECT
  js_test.*
FROM
  (SELECT id FROM t WHERE value @> '{"color":
"red"}' GROUP BY id) x
  JOIN js_test ON js_test.id = x.id;
```

- Весьма непростое решение !

Что хочется ?

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

Что хочется ?

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

```
jsonb @@ jsquery
```

PGCon-2014, Май, Оттава

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)

- # - 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)

- 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()

Найти что-нибудь красное

- ```
  WITH RECURSIVE t(id, value) AS ( SELECT * FROM
  js_test
   UNION ALL
      (
        SELECT
          t.id,
          COALESCE(kv.value, e.value) AS value
        FROM
          t
          LEFT JOIN LATERAL
  jsonb_each(
  CASE WHEN jsonb_typeof(t.value) =
  'object' THEN t.value
            ELSE NULL END) kv ON true
          LEFT JOIN LATERAL
  jsonb_array_elements(
            CASE WHEN
  jsonb_typeof(t.value) = 'array' THEN t.value
            ELSE NULL END) e ON true
          WHERE
            kv.value IS NOT NULL OR e.value IS
  NOT NULL
       )
  )
  ```

```
SELECT
  js_test.*
FROM
  (SELECT id FROM t WHERE value @> '{"color":
"red"}' GROUP BY id) x
  JOIN js_test ON js_test.id = x.id;
```

- **Jsquery**

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

Еще пример

- SQL

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

- Jsquery

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

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)
```

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"
  ]
}
```

Jsquery (indexes)

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

Jsquery (indexes)

- No statistics, no planning :(

Not selective, better not use index!

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

MongoDB 2.6.0

```
db.reviews.find(  {  $and :[ {similar_product_ids: { $in:["B000089778"]}},    {product_sales_rank:{$gt:10000, $lt:20000}}] } )
.explain()
{
		"n" : 45,
		 ....................
		"millis" : 7,
		"indexBounds" : {
			"similar_product_ids" : [
				[
					"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 !
(9 rows)
```

Jsquery (optimiser) — NEW !

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

```
explain (analyze, costs off) select count(*) from jr where
jr @@ 'similar_product_ids && ["B000089778"]
AND product_sales_rank( $ > 10000 AND $  < 20000)'
--------------------------------------------------------------------------------------------------------
 Aggregate (actual time=0.422..0.422 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.099..0.416 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
         Rows Removed by Index Recheck: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual time=0.060..0.060 rows=185 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
 Execution time: 0.480 ms vs 7 ms MongoDB !
```

Jsquery (optimiser) — 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 = *)

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

Jsquery (optimiser) — 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)!

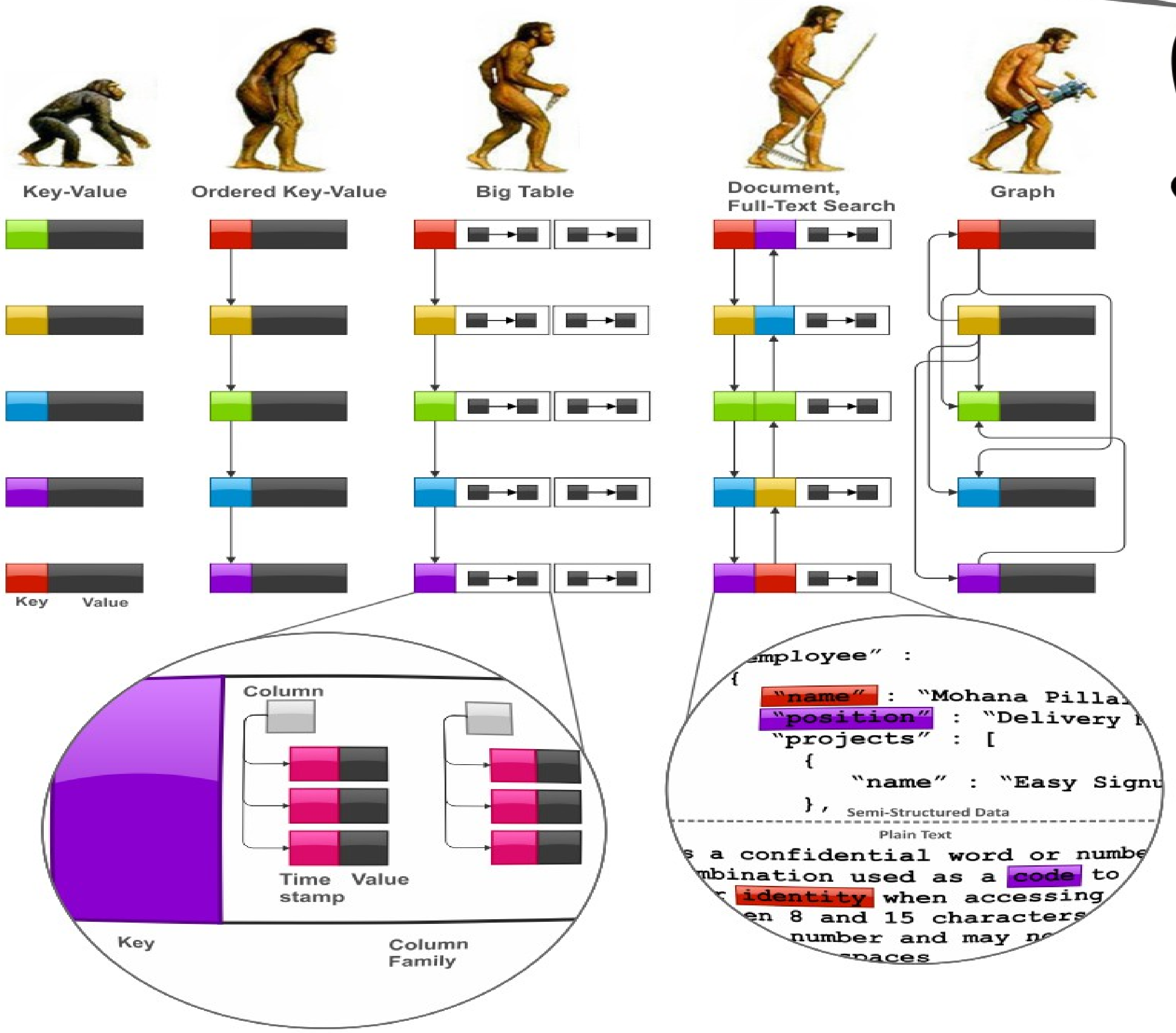
```
explain (analyze, costs off) select count(*) from jr where  jr @@ 'product_sales_rank > 10000'
----------------------------------------------------------------------------------------------------
 Aggregate (actual time=2507.410..2507.410 rows=1 loops=1)
   -> Bitmap Heap Scan on jr (actual time=1118.814..2352.286 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
```

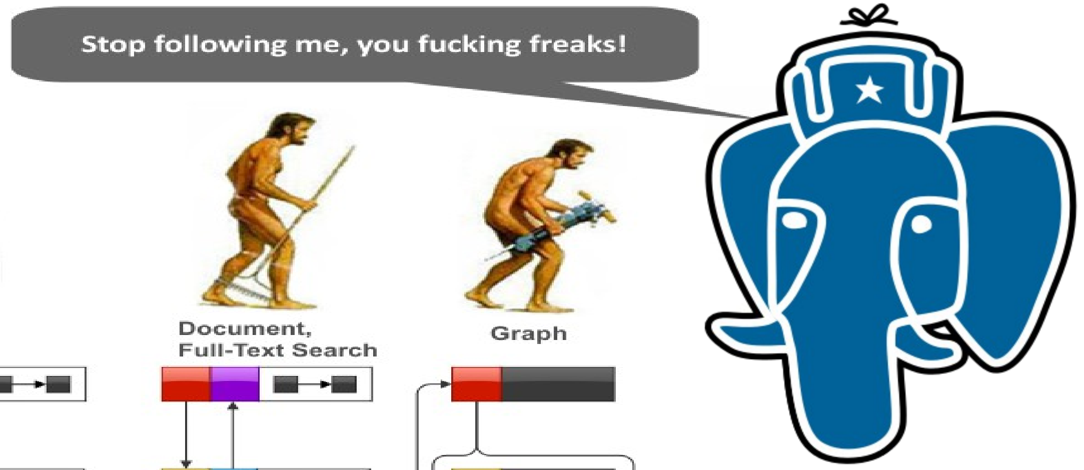
- Better not to use index — HINT /* --noindex */

```
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
```

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 !





PostgreSQL 9.4+
- Open-source
- Relational database
- Strong support of json

PROFESSIONAL Postgres

Thanks for support

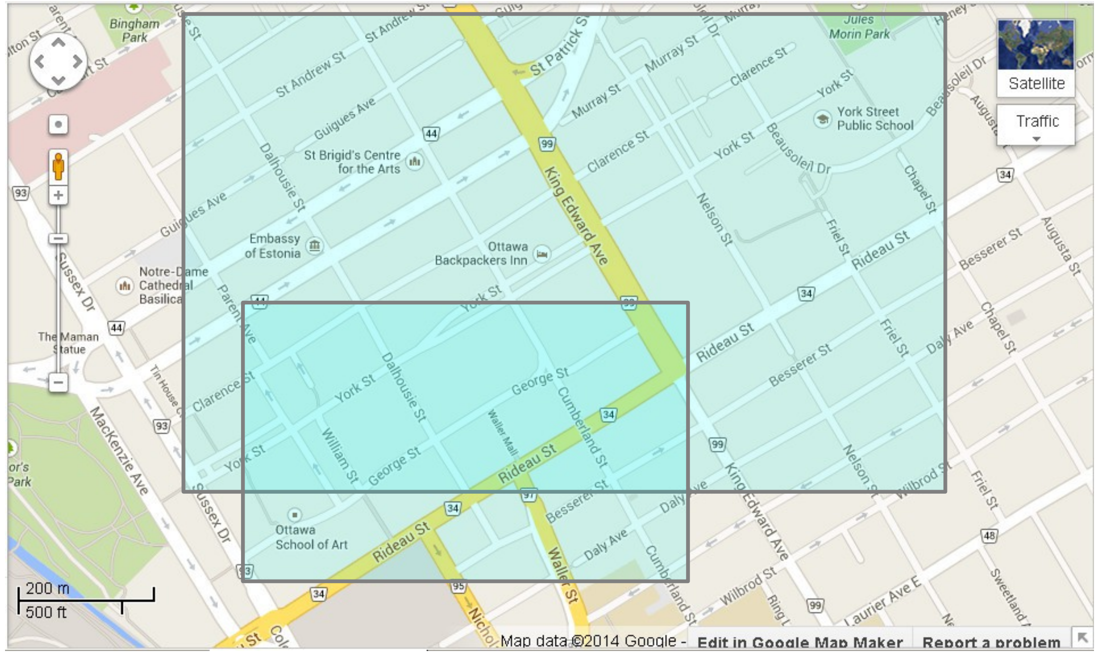
WORLD OF TANKS

WARGAMING.NET

LET'S BATTLE

Что дальше ?

- SQL-level jsquery (расширяемость, статистика)
- VODKA access method ! VODKA Optimized Dendriform Keys Array
  - Комбинация произвольных методов доступа



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.

Oleg Bartunov, Teodor Sigaev PostgreSQL Summit, Toronto, July 8-9, 2006

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:

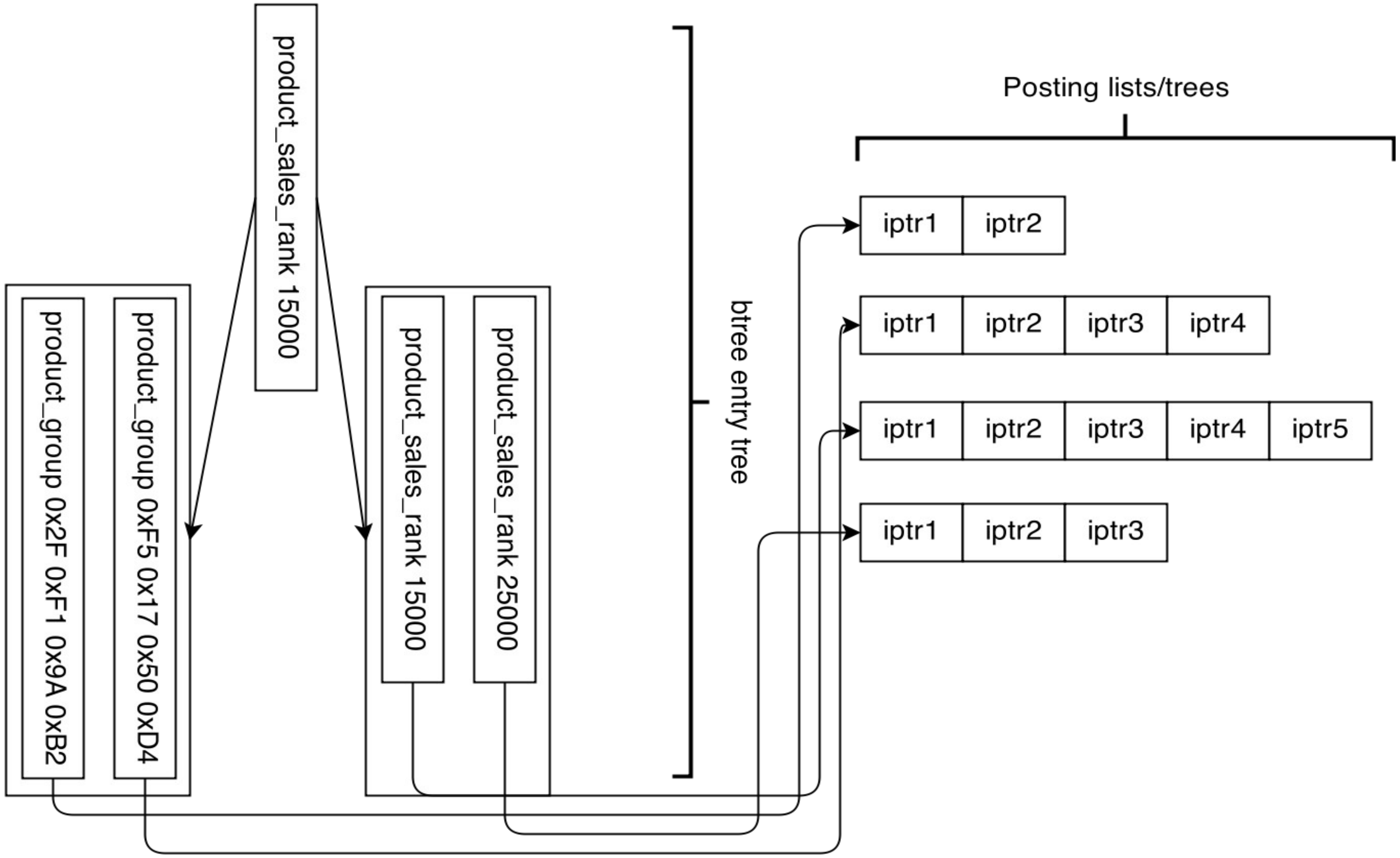
  * Opclasses for all types (no programming, just many catalog changes).

Distant future:

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

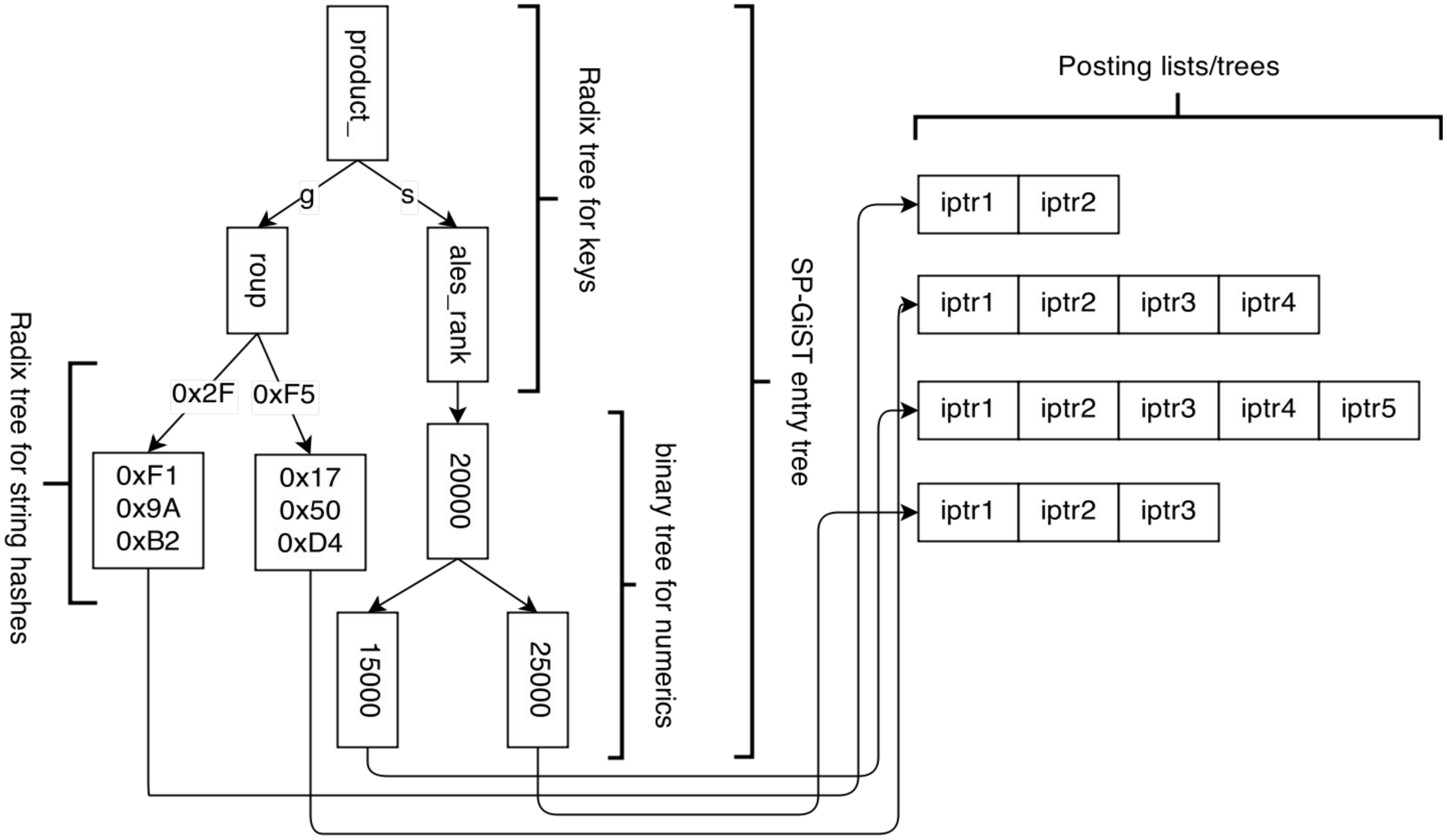
GIN index structure for jsonb

```
{
   "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

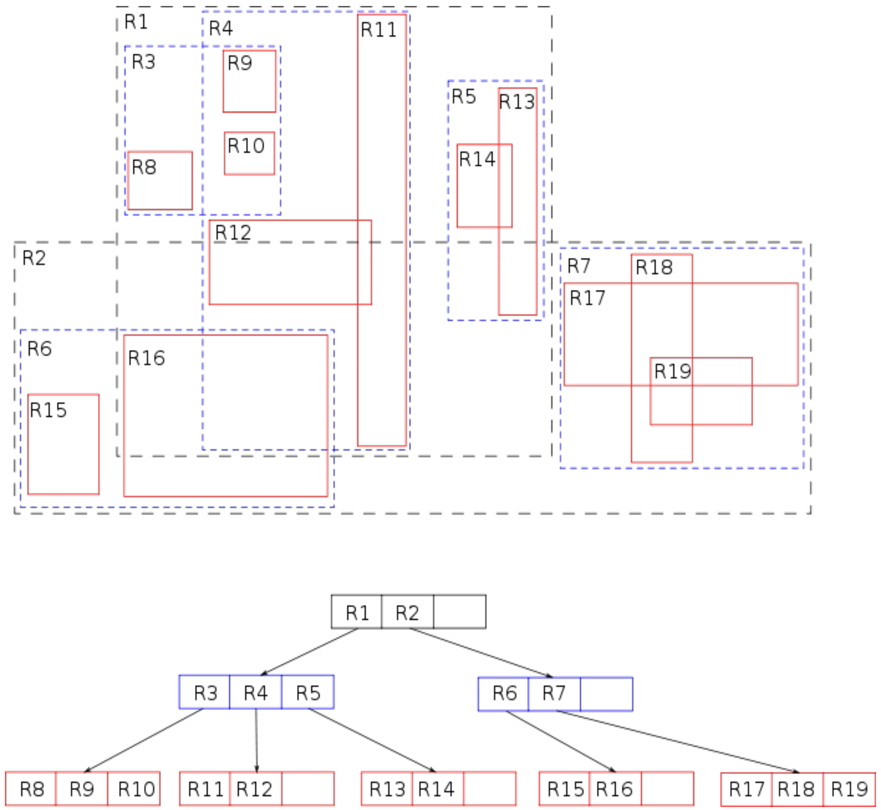
```
explain (analyze, costs off) select count(*) from jr where jr @@  ' similar_product_ids && ["B000089778"]';
                              QUERY PLAN
---------------------------------------------------------------------------------------
 Aggregate (actual time=0.200..0.200 rows=1 loops=1)
   ->  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

Spaghetti indexing ...

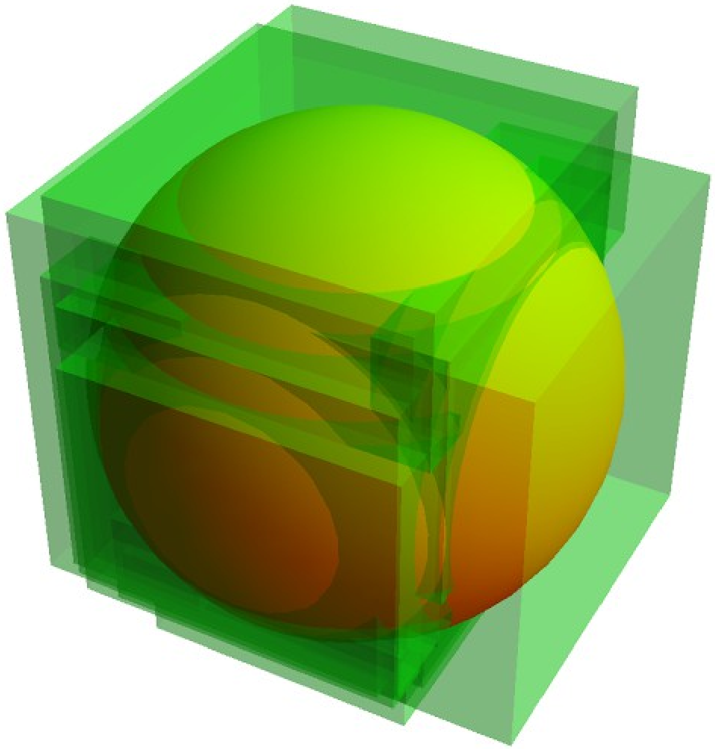
Find twirled spaghetti

Spaghetti indexing …



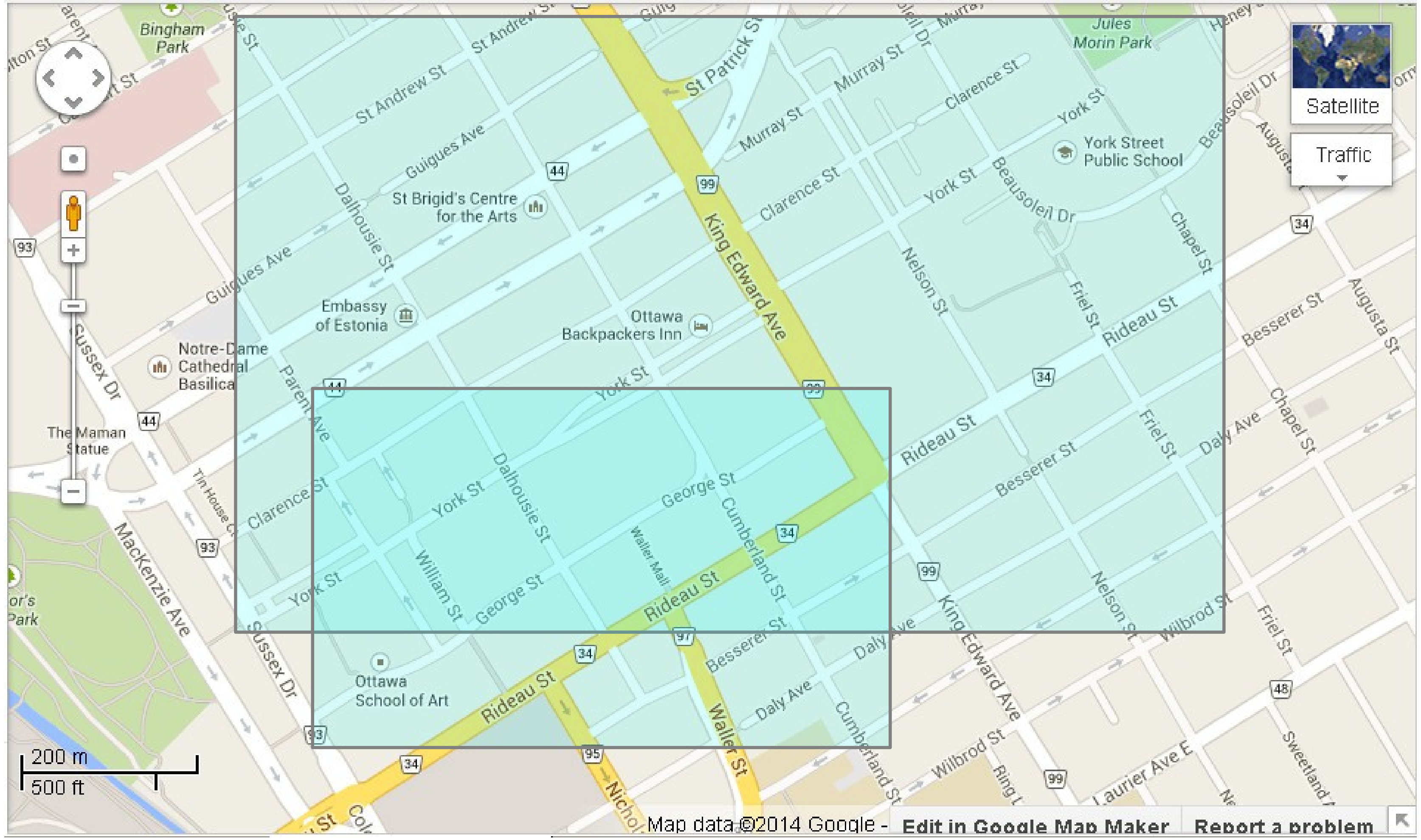
R-tree fails here – bounding box of each separate spaghetti is the same

Spaghetti indexing ...



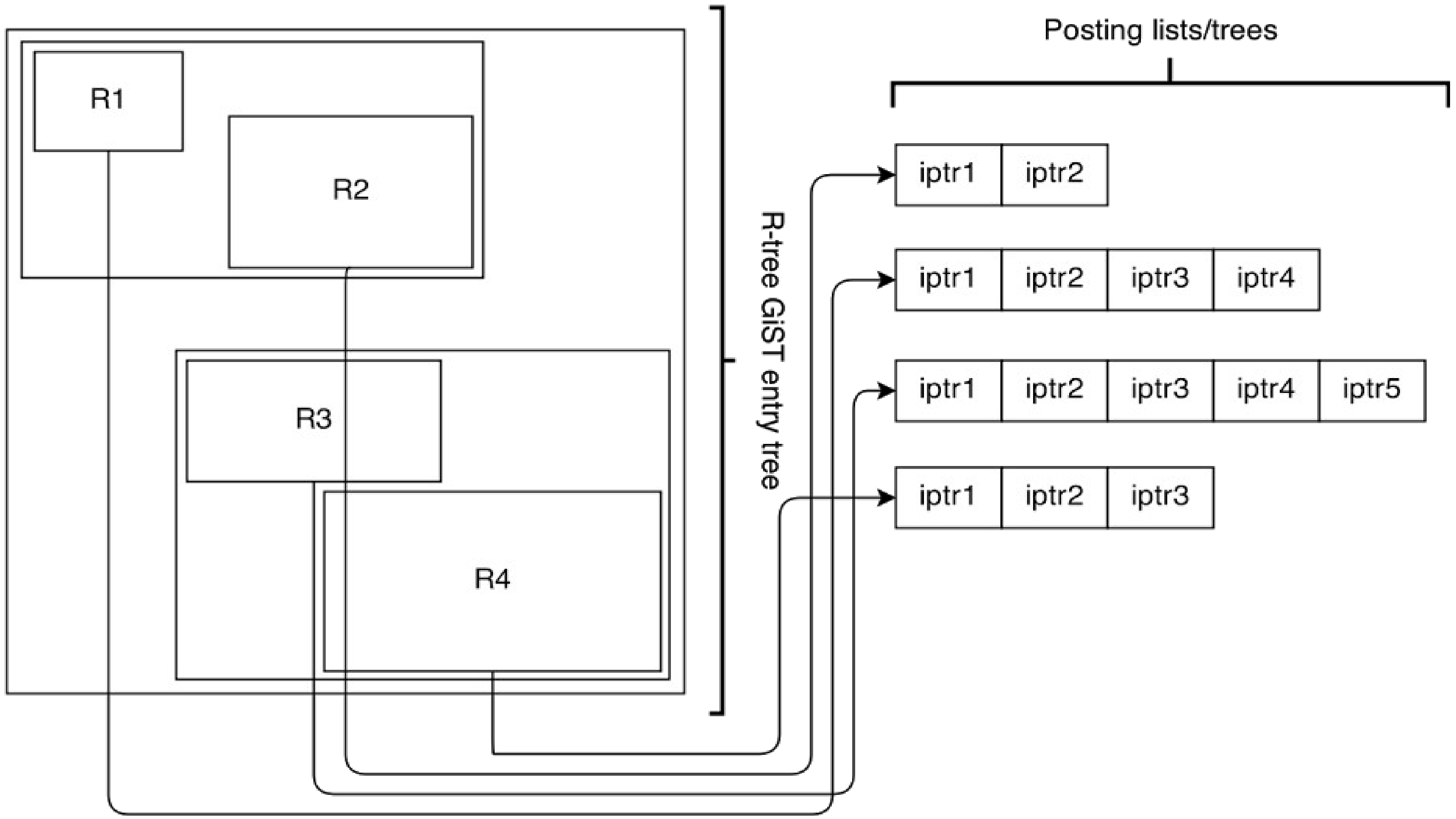
R-tree fails here – bounding box of each separate spaghetti is the same

Ottawa downtown: York and George streets



Idea: Use multiple boxes

Rtree Vodka



R1
R2
R3
R4
R-tree GiST entry tree
Posting lists/trees
iptr1 iptr2
iptr1 iptr2 iptr3 iptr4
iptr1 iptr2 iptr3 iptr4 iptr5
iptr1 iptr2 iptr3

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
- Plans for improving indexing infrastructure
- This work was supported by heroku

Another view on VODKA

- VODKA CONNECTING INDEXES
  - composite index, which combines different access methods
  - Nested search trees

Postgres PROFESSIONAL

postgrespro.ru

- Ищем инженеров:
  - 24x7 поддержка
  - Консалтинг & аудит
  - Разработка админских приложений
  - Пакеты
- Ищем си-шников для работы над постгресом:
  - Неубиваемый и масштабируемый кластер
  - Хранилища (in-memory, column-storage...)

VODKA Optimized Dendriform Keys Array