Postgres Innovations
Beyond High Performance and Availability

Oleg Bartunov, Alexander Korotkov
Postgres Professional

Disclaimer

This is only our opinion on topics under consideration.
It doesn't pretend to be the only truth.

Agenda

- Initial design of Postgres and innovations
- History of some particular innovative features of Postgres
- Some future horizons

Original design of Postgres

The main design goals of the new system are to:

1) provide better support for complex objects,

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

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

4) simplify the DBMS code for crash recovery,

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

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

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

Original design of Postgres

The main design goals of the new system are to:

1) provide better support for complex objects,

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

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

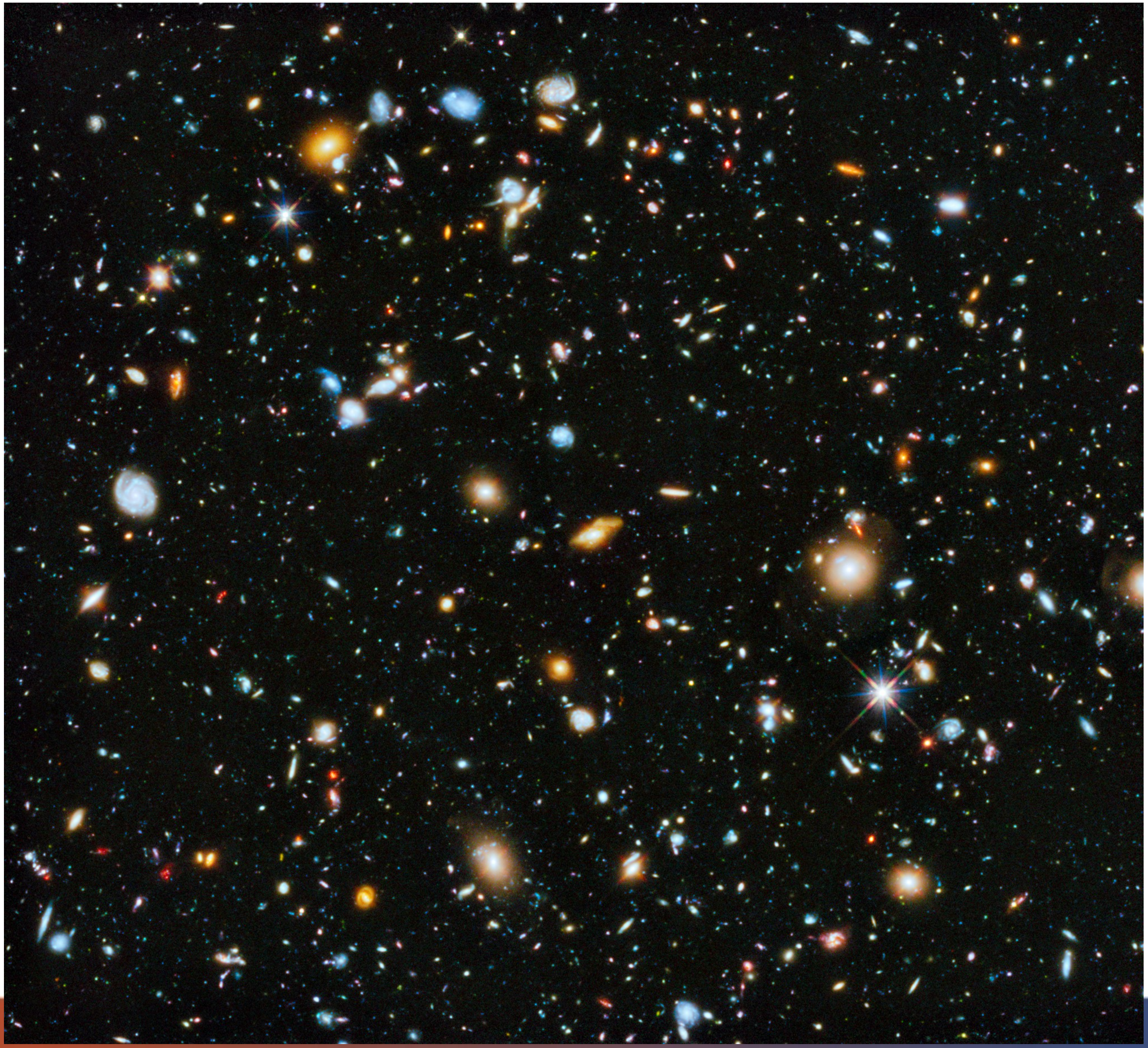
4) simplify the DBMS code for crash recovery,

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

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

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

How does extendability look like?



It's like a shopping mall

Rent a place in the mall

(vs. having your own shop)

Pro

- Use all common facilities of mall
- Use existing buyers base of the mall
- Concentrate on your own content

Cons

- Have to pay the rent

Writing extension to DBMS

(vs. writing your own specific DBMS)

Pro

- Use all common features of DBMS: concurrency, recovery, transactions etc.
- Use existing users base of the DBMS
- Concentrate on your domain specific logic

Cons

- Have to pay some overhead

Extendability need APIs

What can we extend in the DBMS?

- Data types
- How we can operate with this data types? (functions, operators, aggregates etc.)
- How we can search this data types? (indexes)
- What could be the source of data? (FDW)
- How could we store the data? (table engines) (not yet delivered to Postgres)

New types of indexes

are especially hard implement because we need to deal with:

- concurrency (low-level locking etc.),
- packing data into pages,
- WAL-logging,
- …

This is a very hard task. Only DBMS core developer could solve it. Application developer can't.

The solution: add nested API

The solution: add nested API

- Index access method is the template which could be applied to particular data type using operator class (opclass).
  - btree is template for different linear orderings
  - GiST is template for balanced trees
  - SP-GIST is template for imbalanced trees
  - GIN is template for inverted indexes of composite objects
  - BRIN is template for bounding aggregates per block ranges

Propagation of improvements

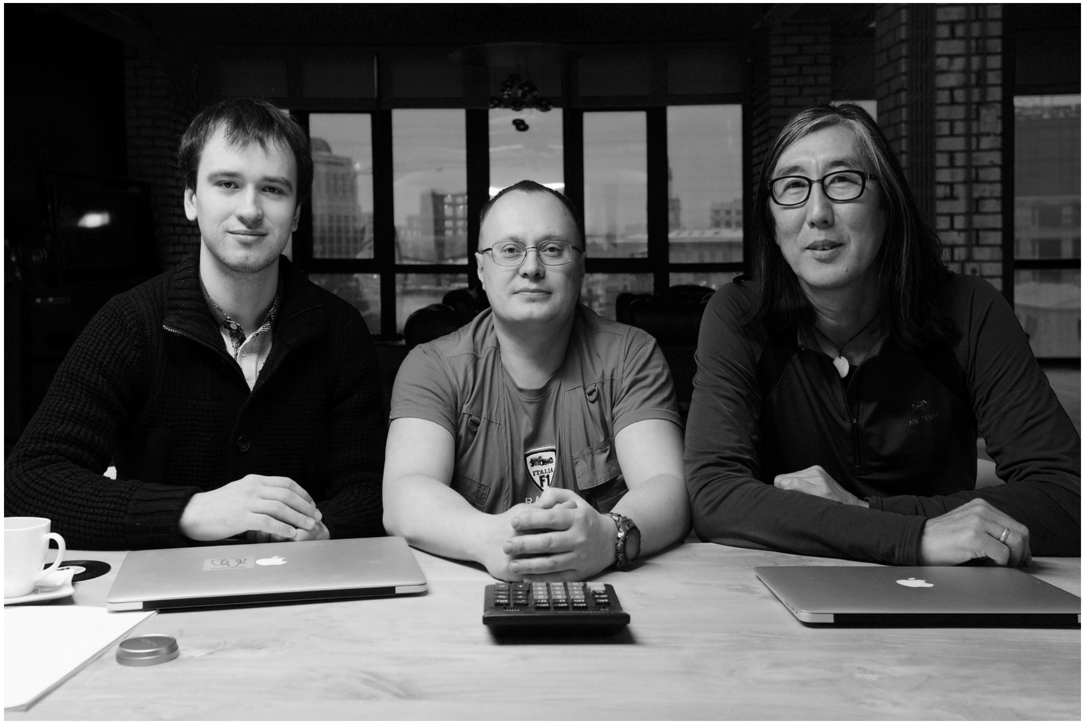
- If you upgrade your camera to another compatible which have higher resolution, this improvement will apply to all the compatible lenses.
- In PostgreSQL 9.4 GIN got 2 major improvements: posting list compression and fast scan. Opclasses received these improvements automatically.

Extendability

Provides fast feature developing

- Hstore (first version) — several hours
- FTS (tsearch2) — 1 week (NY holidays)
- KNN-GiST — 1 week
- jsonb_path_ops — several hours in restaurant
- Jsonb (prototype) — 2-3 months
- Jsquery — 2-3 months
- Quadtree — 360 loc

Alexander Korotkov, Teodor Sigaev, Oleg Bartunov



- Major contributors to PostgreSQL
- Co-founders of Postgres Professional

PostgreSQL CORE

- Locale support
- PostgreSQL extendability: GiST(KNN), GIN, SP-GiST
- Full Text Search (FTS)
- NoSQL (hstore, jsonb)
- Indexed regexp search
- Custom AM & Generic WAL
- Pluggable table engines (WIP)

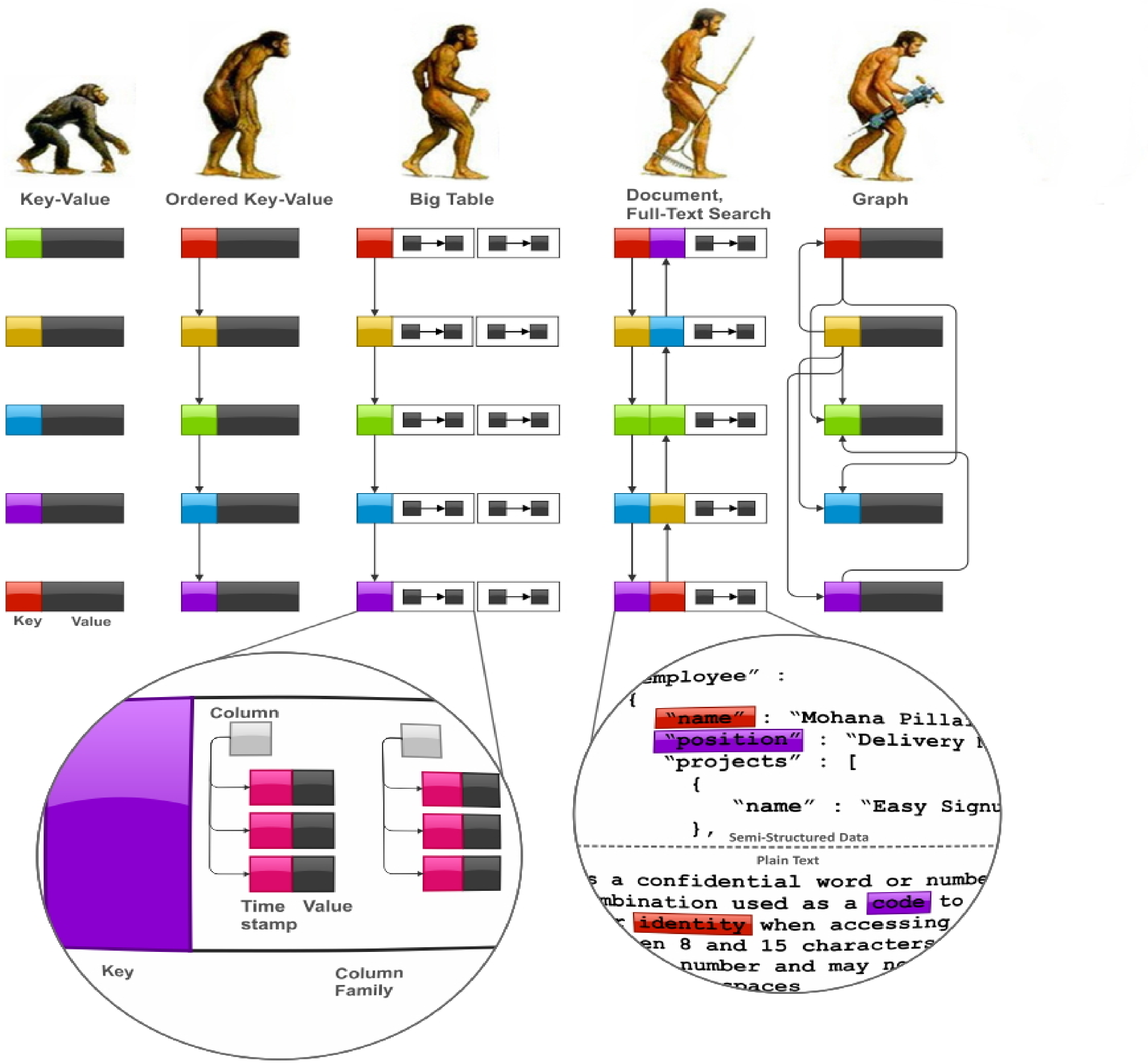
Extensions:
- Intarray
- Pg_trgm
- Ltree
- Hstore
- plantuner

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

- 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


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

The problem

- Some 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
  - Application should wait for schema changing, infrequent releases
- NoSQL uses json format, why not have it in relational database ?

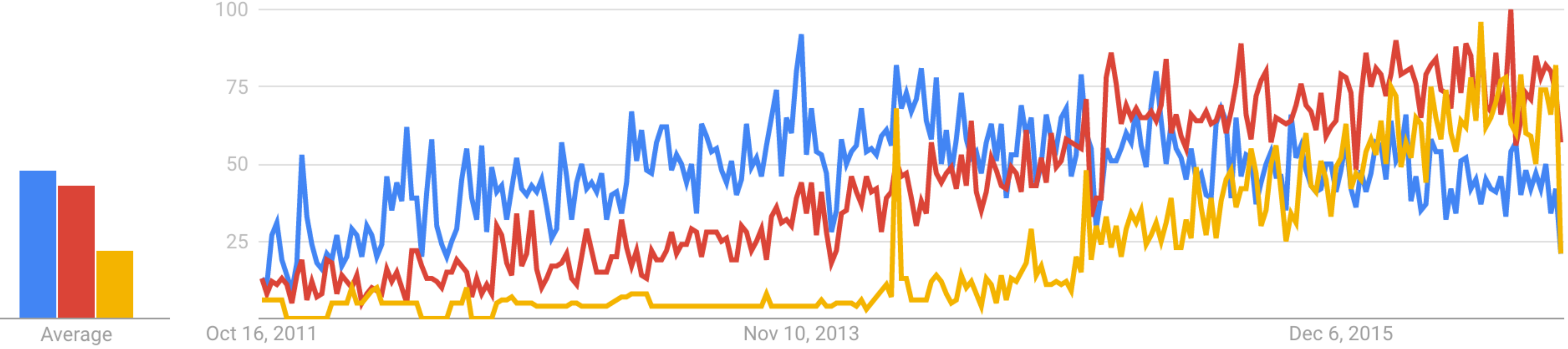
JSON in PostgreSQL

This is the challenge !

Challenge to PostgreSQL !

- Full support of schema-less data
  - 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 !

Jsonb is replacing hstore and json



•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) pa

Introduction to Hstore

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

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

Introduction to hstore

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

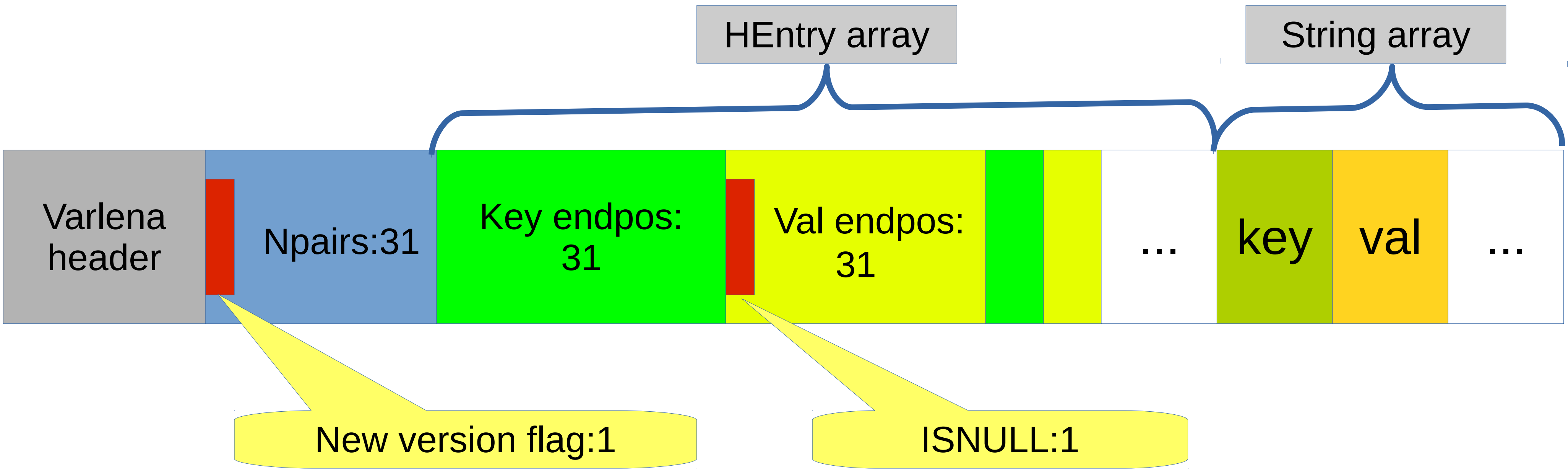
  `'a=>1, b=>2'::hstore`

  – 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 ...)
- 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

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.

Binary representation of jsonb

Object with 3 keys

```
00: 0300 0020 0400 0080  . . .   . . . .
08: 0400 0000 0400 0000  . . . . . . . .
10: 0300 0000 0300 0000  . . . . . . . .
18: 1500 0050 6b65 7931  . . . P k e y 1
20: 6b65 7932 6b65 7933  k e y 2 k e y 3
28: 6162 6378 797a 0000  a b c x y z . .
30: 0300 0040 0100 0080  . . . @ . . . .
38: 0100 0000 0100 0000  . . . . . . . .
40: 7475 76              t u v
```

```
{
  "key1":"abc",
  "key2":"xyz",
  "key3":
  ["t",
   "u",
   "v"]
}
```

Array with 3 elements

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


```json
{
    "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          jsonb - base = 0.4
Json:   9.6  s       json  - base = 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 w
(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"
                                                              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 loo
               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"}}}).
```

PostgreSQL 9.6 vs Mongo 3.2.10(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
    jsonb_path_ops - 295 Mb
  - jsonb_path_ops (tags)- 42 Mb
  - mongo (tags) - 87 (387) MB
    mongo (tags.term - 23 (100) Mb

- Table size
  - postgres : 1.3Gb
  - Mongo : 1.3(1.8)Gb
- Input performance:
  - Text : 34 s
  - Json : 37 s
  - Jsonb : 43 s
  - mongo : 50s(13 m)

Jsonb (Apr, 2014)

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

Jsonb query

- Currently, one can search jsonb data using
  - Contains operators - jsonb @> jsonb, jsonb <@ jsonb (GIN indexes)
    jb @> '{"tags":[{"term":"NYC"}]}'::jsonb

    Keys should be specified from root
- Equivalence operator — jsonb = jsonb (GIN indexes)
  - Exists operators — jsonb ? text, jsonb ?! text[], jsonb ?& text[] (GIN indexes)
    jb WHERE jb ?| '{tags,links}'
    Only root keys supported
  - Operators on jsonb parts (functional indexes)
    SELECT ('{"a": {"b":5}}'::jsonb -> 'a'->>'b')::int > 2;
    CREATE INDEX ....USING BTREE ((jb->'a'->>'b')::int);
    Very cumbersome, too many functional indexes

Jsonb query

- Need Jsonb query language
  - More operators on keys, values
  - Types support
  - Schema support (constraints on keys, values)
  - Indexes support
- Introduce Jsquery - textual data type and @@ match operator

```
jsonb @@ jsquery
```

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 name !

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

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 (contained)

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

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

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
```

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()

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=
         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

```
 Schema |        Name         | Type  |  Owner   | Table |  Size  | Description
--------+---------------------+-------+----------+-------+--------+-------------
 public | jb_gin_idx          | index | postgres | jb    | 632 MB |
 public | jb_path_idx         | index | postgres | jb    | 292 MB |
 public | jb_path_value_idx   | index | postgres | jb    | 414 MB |
 public | jb_tags_idx         | index | postgres | jb    | 41 MB  |
 public | jb_value_path_idx   | index | postgres | jb    | 420 MB |
```

Jsquery (indexes)

select count(*) from jb where jb @@ 'tags.#.term = "NYC"';
Execution time: 0.481 ms

select count(*) from jb where jb @@ '*.term = "NYC"';
Execution time: 0.716 ms

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",
  "review_date": {
    "$date": 31363200000
  },
  "review_helpful_votes": 0,
  "review_rating": 5,
  "review_votes": 10,
  "similar_product_ids": [
    "0471136174",
    "0910627312",
    "047112138X",
    "0786883561",
    "0201570483"
  ]
}
```

Jsquery (indexes)

```
select count(*) from jr
where jr @@  ' similar_product_ids && ["B000089778"]';

 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..0.0
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!

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

 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..126.0
               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.explain("executionStats").find
(
 { $and :[ {similar_product_ids: { $in:["B000089778"] }},
              {product_sales_rank:{$gt:10000, $lt:20000}}
             ]
 }
)
```

Filter product_sales_rank ,index on similar_product_ids

Jsquery (indexes)

- If we rewrite query to 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"]'::jsqu
         Filter: ((((jr ->> 'product_sales_rank'::text))::integer > 10000) AN
(((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
               Index Cond: (jr @@ '"similar_product_ids" && ["B000089778"]'::
 Execution time: 0.506 ms   Potentially,  query could be faster Mongo !
(9 rows)
```

Jsquery (optimizer)

- Jsquery has built-in optimiser for simple queries.

```
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)

- 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 (optimizer)

- Jsquery optimiser pushes non-selective operators to recheck

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

 Aggregate (actual time=0.422..0.422 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=0.099..0.416 rows=45 loops=1)
         Recheck Cond: (jr @@ '("similar_product_ids" && ["B000089778"] AND
"product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
         Rows Removed by Index Recheck: 140
         Heap Blocks: exact=107
         ->  Bitmap Index Scan on jr_path_value_idx (actual
time=0.060..0.060 rows=185 loops=1)
               Index Cond: (jr @@ '("similar_product_ids" && ["B000089778"]
AND "product_sales_rank"($ > 10000 AND $ < 20000))'::jsquery)
 Execution time: 0.480 ms
```

Jsquery (HINTING)

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

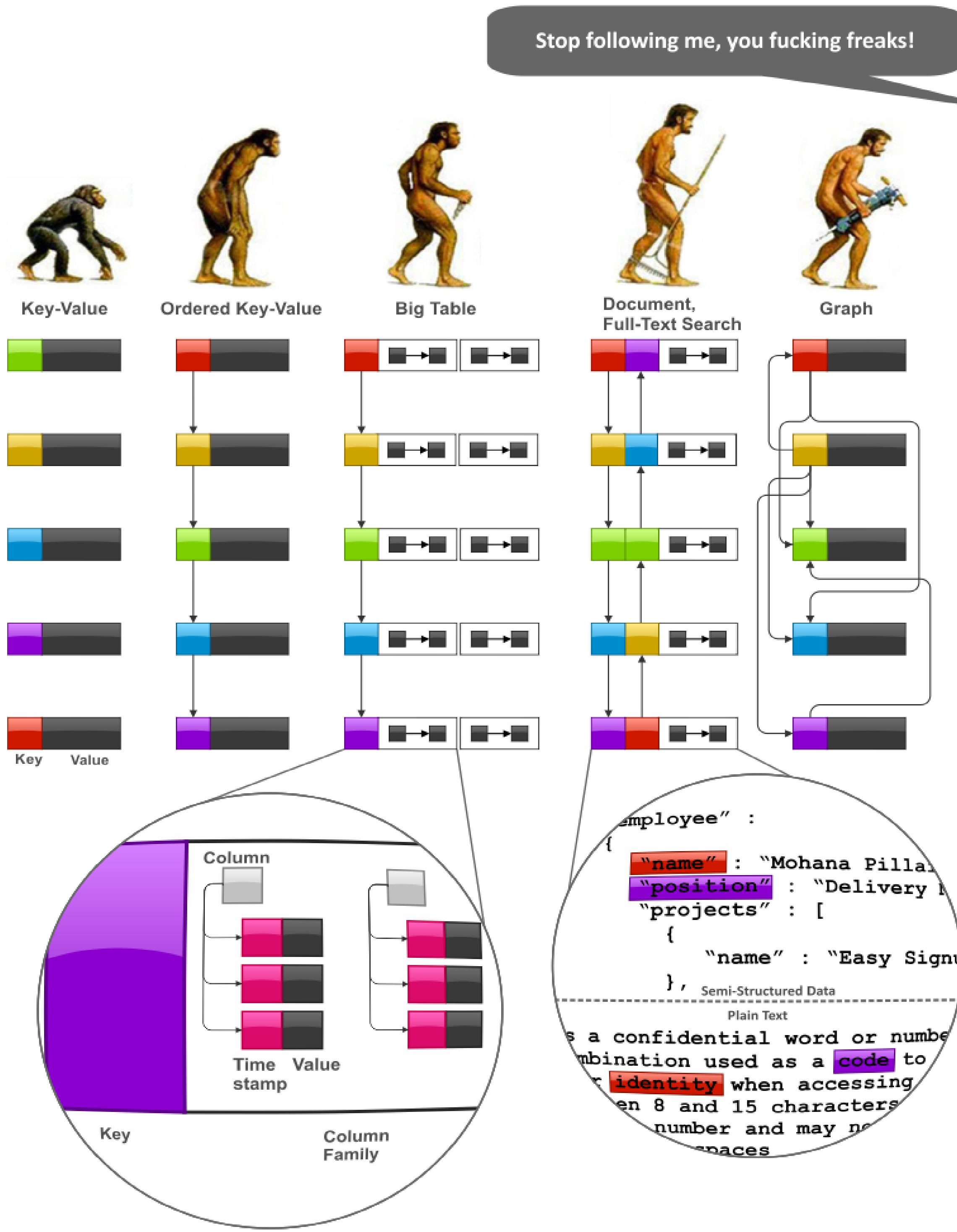
```
explain (analyze, costs off) select count(*) from jr where  jr @@ 'product_sales_rank > 10000'
----------------------------------------------------------------------------------------------------
 Aggregate (actual time=2507.410..2507.410 rows=1 loops=1)
   ->  Bitmap Heap Scan on jr (actual time=1118.814..2352.286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

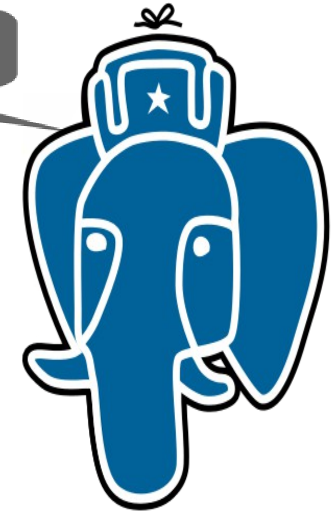
- Better not to use index — HINT /* --noindex */

```
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/postgrespro/jsquery
  - Supported by Postgres Professional





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

Better indexing ...

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

```
                                  List of relations
 Schema |             Name             | Type  |  Owner   |     Table      |  Size   |
--------+------------------------------+-------+----------+----------------+---------+-
 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  |t
 public | jb_uniq_paths_spgist_idx     | index | postgres | jb_uniq_paths  | 598 MB  |n
```

K-Nearest Neighbors Search

- Traditional search algorithms are not effective
  - Index doesn't helps, since there is no predicate
  - Full table scan -> sort -> limit
  - Ad-hoc solutions are not effective
- Postgres innovation
  - Use special index scan strategy to get k-tuples in "right" order
  - Several orders of magnitude speedup !
  - Use ORDER BY distance to express KNN in SQL
  - KNN-GiST, KNN-Btree, KNN-SPGiST

K-Nearest Neighbors Search

1,000,000 randomly distributed points

Find K-closest points to the point (0,0)

- Scan & Sort

```
SELECT * FROM qq ORDER BY point_distance(p,'(0,0)') ASC LIMIT 10;

 Limit (actual time=291.524..291.526 rows=10 loops=1)
   ->  Sort (actual time=291.523..291.523 rows=10 loops=1)
         Sort Key: (point_distance(p, '(0,0)'::point))
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Seq Scan on qq (actual time=0.011..166.091 rows=1000000 loops=1)
 Planning time: 0.048 ms
 Execution time: 291.542 ms
(7 rows)
```

K-Nearest Neighbors Search

1,000,000 randomly distributed points

Find K-closest points to the point (0,0)

- KNN-GiST (GiST index for points)

```
SELECT * FROM qq ORDER BY (p <-> '(0,0)') ASC LIMIT 10;


 Limit (actual time=0.046..0.058 rows=10 loops=1)
   ->  Index Scan using qq_p_s_idx on qq (actual time=0.046..0.058 rows=10 loops
         Order By: (p <-> '(0,0)'::point)
 Planning time: 0.052 ms
 Execution time: 0.081 ms
(5 rows)
```

KNN is 3500 times faster !

K-Nearest Neighbors Search

KNN-Btree

Find 10 closest events to the "Sputnik" launch

- Union of two selects (btree index on date)

```
select  *, date <-> '1957-10-04'::date as dt from (
    select * from (select id, date, event from events
     where date <= '1957-10-04'::date order by date desc limit 10) t1
union
    select * from ( select id, date, event from events
     where date >= '1957-10-04'::date order by date asc limit 10) t2) t3
order by dt asc limit 10;

 Execution time: 0.146 ms
```

K-Nearest Neighbors Search

KNN-Btree

Find 10 closest events to the "Sputnik" launch

- Parallel Btree iindex-scans in two directions

```
select id, date, event from events order by date <-> '1957-10-04'::date asc
limit 10;


 Limit (actual time=0.030..0.039 rows=10 loops=1)
   ->  Index Scan using btree_date_idx on events
(actual time=0.030..0.036 rows=10 loops=1)
         Order By: (date <-> '1957-10-04'::date)
 Planning time: 0.101 ms
 Execution time: 0.070 ms
(5 rows)
```

KNN is 2 times faster !

What is a Full Text Search ?

- Full text search
  - Find documents *matching* a query
  - Sort them in some order (optionally)

- Typical Search
  - Find documents with **all words** from query
  - Sorted documents by their relevance to a query

Why FTS in Databases ?

- Feed database content to external search engines
  - They are fast !

  BUT
- They can't index all documents - could be totally virtual
- They don't have access to attributes - no complex queries
- They have to be maintained — headache for DBA
- Sometimes they need to be certified
- They don't provide instant search (need time to download new data and reindex)
- They don't provide consistency — search results can be already deleted from database

FTS in Databases

- **FTS requirements**
  - **Full integration with database engine**
    - Transactions
    - Concurrent access
    - Recovery
    - Online index
  - Configurability (parser, dictionary...)
  - Scalability

Traditional text search operators

(TEXT op TEXT, op - ~, ~*, LIKE, ILIKE)

- No linguistic support
  - What is a word ?
  - What to index ?
  - Word «normalization» ?
  - Stop-words (noise-words)
- No ranking - all documents are equally similar to query
- Slow, documents should be seq. scanned
  9.3+ index support of ~* (pg_trgm)

select * from man_lines where man_line ~* '(?:(?:p(?:ostgres(?:ql)?|g?sql)|sql)) (?:(?:(?:mak|us)e|do|is))';

```
One of (postgresql,sql,postgres,pgsql,psql) space One of (do,is,use,make)
```

FTS in PostgreSQL

- OpenFTS — 2000, Pg as a storage
- GiST index — 2000, thanks Rambler
- Tsearch — 2001, contrib:no ranking
- Tsearch2 — 2003, contrib:config
- GIN —2006, thanks, JFG Networks
- FTS — 2006, in-core, thanks,EnterpriseDB
- FTS(ms) — 2012, some patches committed
- 2016 — Postgres Professional

FTS in PostgreSQL

- **tsvector** – data type for document optimized for search
  - Sorted array of lexems
  - Positional information
  - Structural information (importance)
- **tsquery** – textual data type for query with boolean operators & | ! ()
- **Full text search operator @@:** tsvector @@ tsquery
- Operators @>, <@ for tsquery
- **Functions**: to_tsvector, to_tsquery, plainto_tsquery, ts_lexize, ts_debug, ts_stat, ts_rewrite,ts_headline, ts_rank, ts_rank_cd, setweight,……………………..
- **Indexes**: GiST, GIN

FTS in PostgreSQL

What is the benefit ?

Document processed only once when inserting into a table, no overhead in search

- Document parsed into tokens using pluggable parser
- Tokens converted to lexems using pluggable dictionaries
- Words positions with labels (importance) are stored and can be used for ranking
- Stop-words ignored

FTS in PostgreSQL

- Query processed at search time
  - Parsed into tokens
  - Tokens converted to lexems using pluggable dictionaries
  - Tokens may have labels (weights)
  - Stop-words removed from query
  - It's possible to restrict search area

    `'fat:ab & rats & ! (cats | mice)'`
  - Prefix search is supported

    `'fa*:ab & rats & ! (cats | mice)'`
  - Query can be rewritten «on-the-go»

FTS summary

- FTS in PostgreSQL is a «collection of bricks» to build a custom search engine
  - Custom parser
  - Custom dictionaries
  - Use tsvector as a custom storage
  - + All power of SQL (FTS+Spatial+Temporal)
- For example, instead of textual documents consider chemical formulas or genome string

New advances in FTS

- Phrase search — 9.6
  - 92 posts with person 'Tom Good', but FTS finds > 34K posts.
- Fast relevance search — RUM ext.
- FTS with alternative ordering — RUM ext.
- Inverted FTS — RUM ext.

No positions in index !

Inverted Index in PostgreSQL

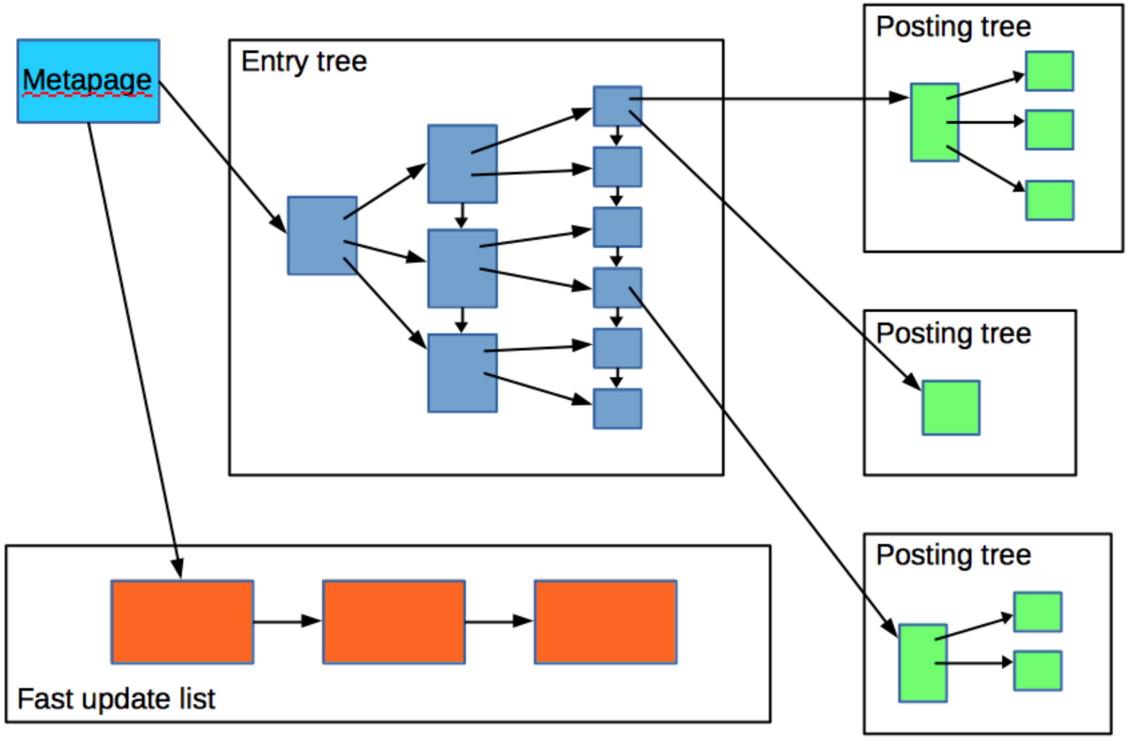
ENTRY TREE

Report Index

A

B

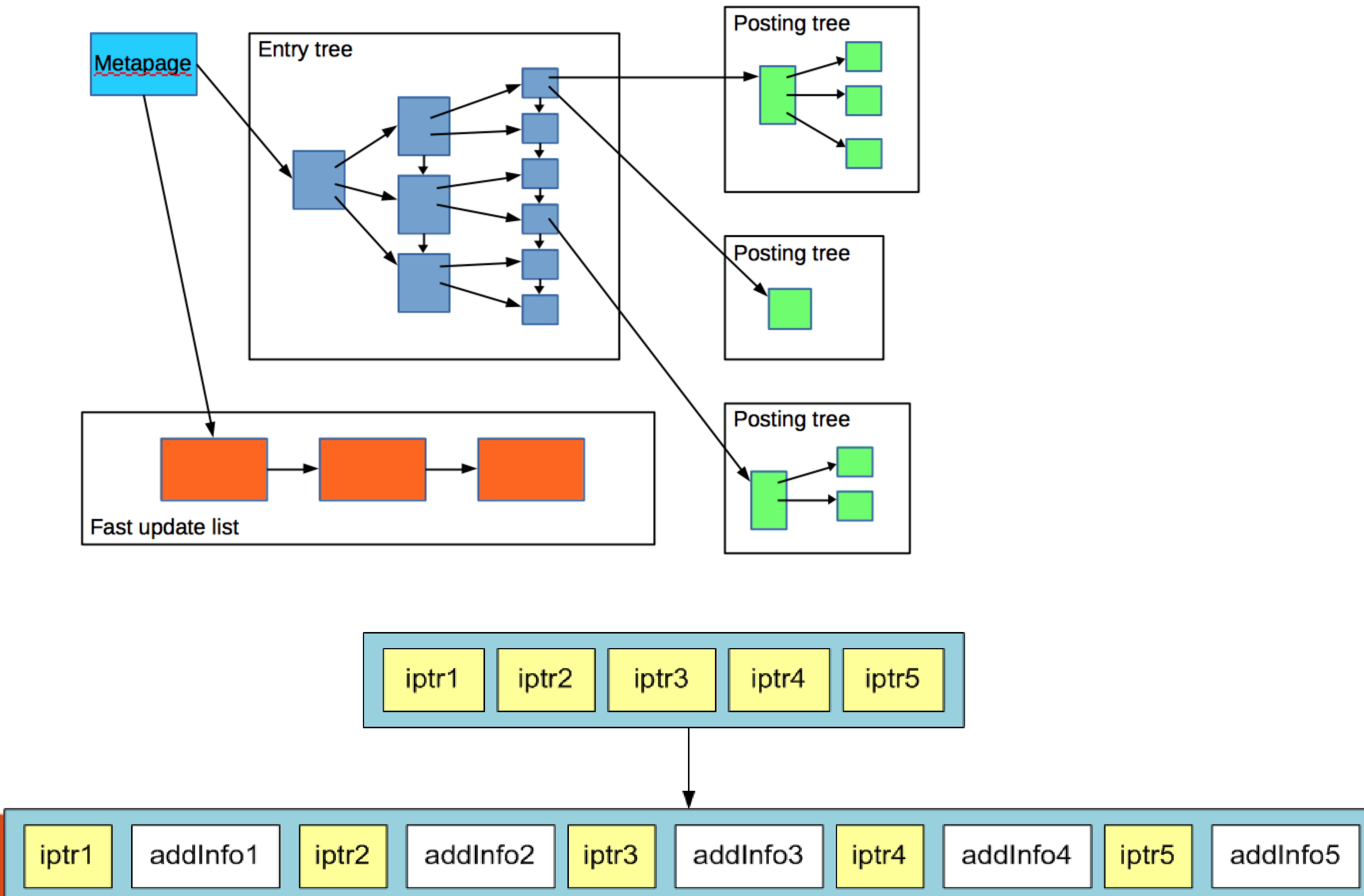
Posting list
Posting tree



Improving GIN

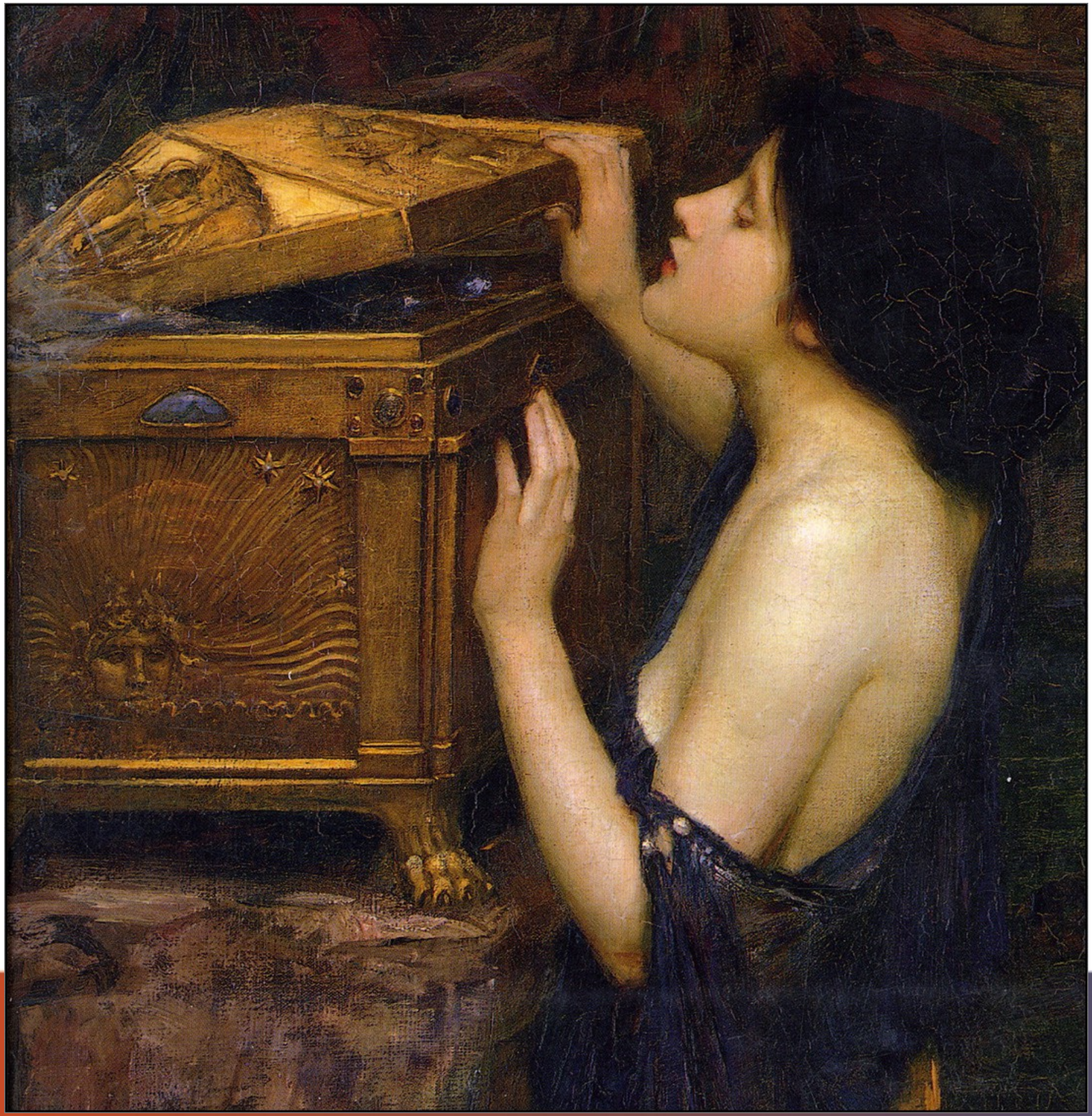
- Improve GIN index
  - Store additional information in posting tree, for example, lexemes positions or timestamps
  - Use this information to order results

Improving GIN



9.6 opens «Pandora box»

Create access methods as extension !   Let's call it RUM



CREATE INDEX ... USING RUM

- Use positions to calculate rank and order results
- Introduce distance operator tsvector <=> tsquery

Top-10 (out of 222813) postings with «Tom Lane»

  - GIN index — 1374.772 ms

```
create index pglist_rum_fts_idx on pglist using rum(fts rum_tsvector_ops);

SELECT subject  FROM pglist WHERE fts @@ plainto_tsquery('tom lane')
ORDER BY fts <=> plainto_tsquery('tom lane') LIMIT 10;
                                             QUERY PLAN
----------------------------------------------------------------------------------
 Limit (actual time=215.115..215.185 rows=10 loops=1)
   ->  Index Scan using pglist_rum_fts_idx on pglist (actual time=215.113..215.183 rows=10 lo
         Index Cond: (fts @@ plainto_tsquery('tom lane'::text))
         Order By: (fts <=> plainto_tsquery('tom lane'::text))
 Planning time: 0.264 ms
 Execution time: 215.833 ms 6X speedup !
(6 rows)
```

CREATE INDEX ... USING RUM

- RUM uses new ranking function (ts_score) — combination of ts_rank and ts_tank_cd
  - ts_rank doesn't supports logical operators
  - ts_rank_cd works poorly with OR queries

```
SELECT ts_rank(fts,plainto_tsquery('english', 'tom lane')) AS rank,
       ts_rank_cd (fts,plainto_tsquery('english', 'tom lane')) AS rank_cd ,
       fts <=> plainto_tsquery('english', 'tom lane') as score, subject
FROM pglist WHERE fts @@ plainto_tsquery('english', 'tom lane')
ORDER BY fts <=> plainto_tsquery('english', 'tom  lane') LIMIT 10;

   rank    | rank_cd |  score   |                            subject
----------+---------+----------+---------------------------------------------------------------
 0.999637 | 2.02857 | 0.487904 | Re: ATTN: Tom Lane
 0.999224 | 1.97143 | 0.492074 | Re: Bug #866 related problem (ATTN Tom Lane)
  0.99798 | 1.97143 | 0.492074 | Tom Lane
 0.996653 | 1.57143 | 0.523388 | happy birthday Tom Lane ...
 0.999697 | 2.18825 | 0.570404 | For Tom Lane
 0.999638 | 2.12208 | 0.571455 | Re: Favorite Tom Lane quotes
 0.999188 | 1.68571 | 0.593533 | Re: disallow LOCK on a view - the Tom Lane remix
 0.999188 | 1.68571 | 0.593533 | Re: disallow LOCK on a view - the Tom Lane remix
 0.999188 | 1.68571 | 0.593533 | Re: disallow LOCK on a view - the Tom Lane remix
 0.999188 | 1.68571 | 0.593533 | Re: [HACKERS] disallow LOCK on a view - the Tom Lane remix
(10 rows)
```

Phrase Search (8 years old!)

- Queries 'A & B'::tsquery and 'B & A'::tsquery produce the same result
- Phrase search - preserve order of words in a query
- New FOLLOWED BY (<->) operator:
  - Guarantee an order of operands
  - Distance between operands

$a <n> b == a \& b \& (\exists\ i,j : pos(b)i – pos(a)j = n)$

Phrase search - definition

- FOLLOWED BY operator returns:
  - false
  - true and array of positions of the **right** operand, which satisfy distance condition
- FOLLOWED BY operator requires positions

```
select 'a b c'::tsvector @@ 'a <-> b'::tsquery; – false, there no positions
 ?column?
----------
 f
(1 row)
select 'a:1 b:2 c'::tsvector @@ 'a <-> b'::tsquery;
 ?column?
----------
 t
(1 row)
```

Phrase search - properties

- 'A <-> B' = 'A<1>B'
- 'A <0> B' matches the word with two different forms (infinitives)

```
=# SELECT ts_lexize('ispell','bookings');
   ts_lexize
----------------
 {booking,book}
to_tsvector('bookings') @@ 'booking <0> book'::tsquery
```

Phrase search - properties

- Precendence of tsquery operators - '! <-> & |'

  Use parenthesis to control nesting in tsquery

```
select 'a & b <-> c'::tsquery, 'b <-> c & a'::tsquery;
      tsquery       |       tsquery
--------------------+--------------------
 'a' & 'b' <-> 'c' | 'b' <-> 'c' & 'a'
(1 row)

 select 'b <-> (c & a)'::tsquery;
          tsquery
---------------------------
 'b' <-> 'c' & 'b' <-> 'a'
```

Phrase search - example

- `TSQUERY phraseto_tsquery([CFG,] TEXT)`

  Stop words are taken into account.

```
select phraseto_tsquery('PostgreSQL can be extended by the user in many ways');
                    phraseto_tsquery
-----------------------------------------------------------
 'postgresql' <3> 'extend' <3> 'user' <2> 'mani' <-> 'way'
(1 row)
```

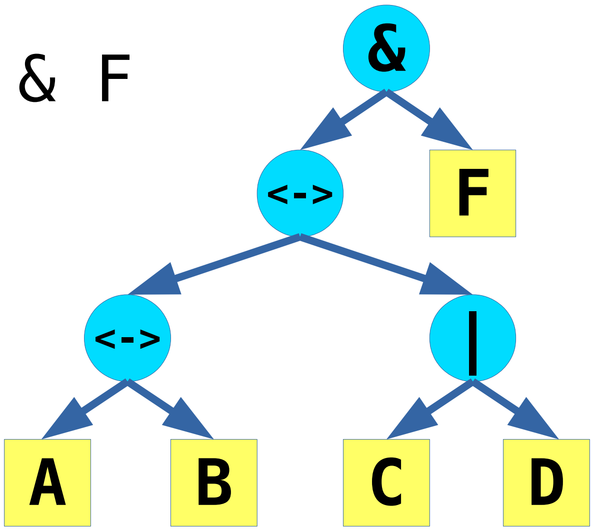
- `It’s possible to combine tsquery’s`

```
select phraseto_tsquery('PostgreSQL can be extended by the user in many ways') ||
                    to_tsquery('oho<->ho & ik');
                                   ?column?
-----------------------------------------------------------------------------------
 'postgresql' <3> 'extend' <3> 'user' <2> 'mani' <-> 'way' | 'oho' <-> 'ho' & 'ik'
(1 row)
```

Phrase search - internals

- Phrase search has overhead, since it requires access and operations on posting lists

```
( (A <-> B) <-> (C | D) ) & F
```

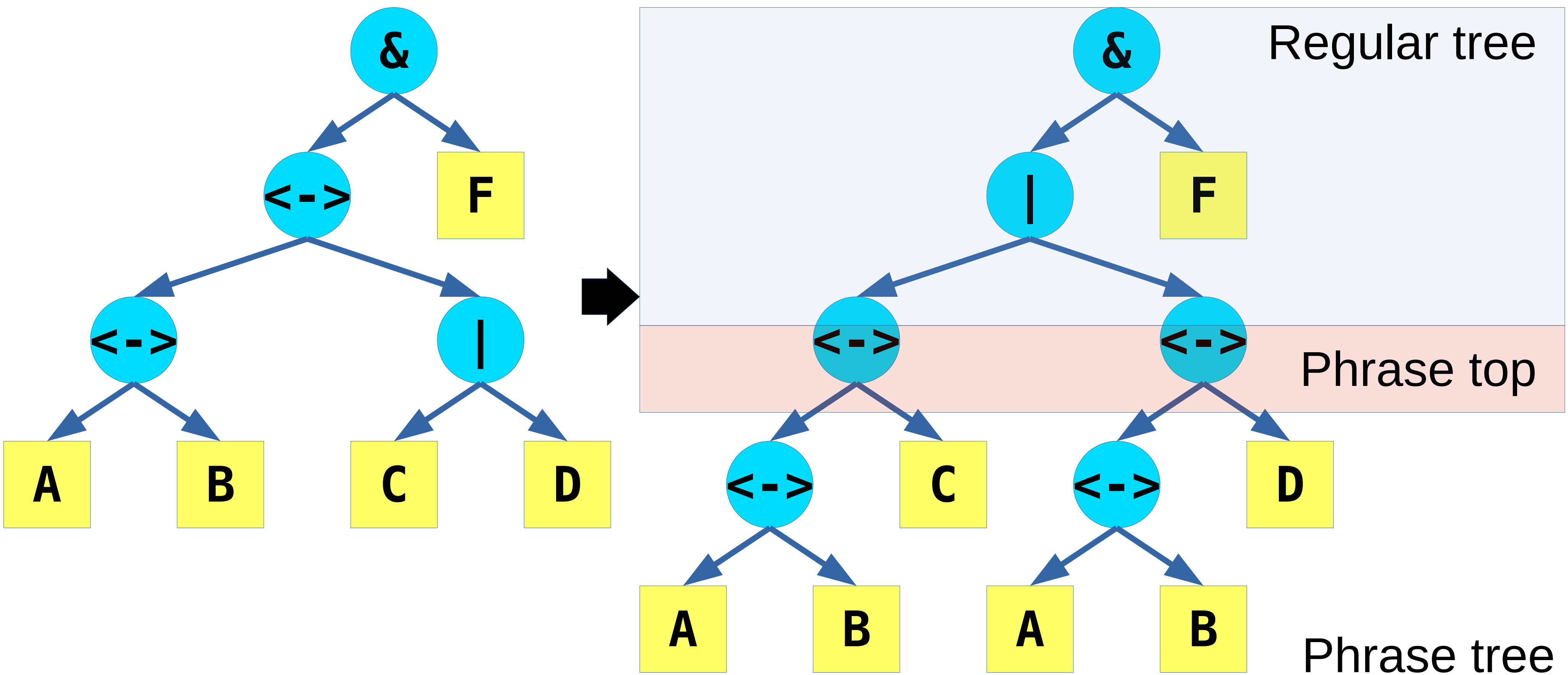


- We want to avoid slowdown FTS operators (& |), which do not need positions.

- Rewrite query, so any <-> operators pushed down in query tree and call phrase executor for the top <-> operator.

Phrase search - transformation

`( (A <-> B) <-> (C | D) ) & F`



`( 'A' <-> 'B' <-> 'C' | 'A' <-> 'B' <-> 'D' ) & 'F'`

Phrase search - push down

```
a <-> (b&c) => a<->b & a<->c

(a&b) <-> c => a<->c & b<->c

a <-> (b|c) => a<->b | a<->c

(a|b) <-> c => a<->c | b<->c

a <-> !b     => a & !(a<->b)
```

there is no position of A followed by B

```
!a <-> b     => !(a<->b) & b
```

there is no position of B precedenced by A

Phrase search - transformation

```
# select '( A | B ) <-> ( D | C )'::tsquery;
                     tsquery
  ----------------------------------------------
'A' <-> 'D' | 'B' <-> 'D' | 'A' <-> 'C' | 'B' <-> 'C'


# select 'A <-> ( B & ( C | ! D ) )'::tsquery;
                         tsquery
-----------------------------------------------------
'A' <-> 'B' & ( 'A' <-> 'C' | 'A' & !( 'A' <-> 'D' ) )
```

Phrase search - Examples

- 1.1 mln postings (postgres mailing lists)

```
select count(*) from pglist where fts @@ to_tsquery('english','tom <-> lane');
 count
--------
 222777
(1 row)
```

- There is overhead of phrase operator

	tom<->lane	'tom & lane'
SeqScan :	2.6s	2.2 s
GIN :	1.2s	0.48 s – need recheck
RUM :	0.5s	0.48 s – use positions to filter

- Phrase search with RUM index has negligible overhead

FTS with alternative ordering

- Store timestamp[tz] in additional information in timestamp[tz] order !

```
CREATE INDEX pglist_fts_ts_order_rum_idx ON pglist USING rum
  (fts rum_tsvector_timestamp_ops, sent)
WITH (attach = 'sent',to ='fts', order_by_attach = 't');

select sent,  subject from pglist
where fts @@ to_tsquery('tom & lane')
order by sent <=> '2000-01-01'::timestamp limit 5;
---------------------------------------------------------------------
 L Limit (actual time=84.866..84.870 rows=5 loops=1)
   ->  Index Scan using pglist_fts_ts_order_rum_idx on pglist (actual
time=84.865..84.869 rows=5 loops=1)
         Index Cond: (fts @@ to_tsquery('tom & lane'::text))
         Order By: (sent <=> '2000-01-01 00:00:00'::timestamp without
time zone)
 Planning time: 0.162 ms
 Execution time: 85.602 ms vs 645 ms !
(6 rows)
```

Some FTS problems: #3

- Combine FTS with ordering by timestamp
  - Store timestamps in additional information in timestamp order !

select sent, subject from pglist
where fts @@ to_tsquery('tom & lane') and sent < '2000-01-01'::timestamp order by sent desc
limit 5;

explain analyze select sent, subject from pglist
where fts @@ to_tsquery('tom & lane') order by sent <=| '2000-01-01'::timestamp limit 5;

```
Speedup ~ 1x,since 'tom lane' is popular →  filter
------------------------------------------------
select sent,  subject from pglist
where fts @@ to_tsquery('server & crashed') and sent < '2000-01-
01'::timestamp order by sent desc limit 5;

select sent,  subject from pglist
where fts @@ to_tsquery('server & crashed') order by sent <=| '2000-
01-01'::timestamp limit 5;

Speedup ~ 10x
```

Inverse FTS (FQS)

- Find queries, which match given document
  - Automatic text classification

```
SELECT * FROM queries;
                 q                  |  tag
-----------------------------------+-------
 'supernova' & 'star'              | sn
 'black'                           | color
 'big' & 'bang' & 'black' & 'hole' | bang
 'spiral' & 'galaxi'               | shape
 'black' & 'hole'                  | color
(5 rows)


SELECT * FROM queries WHERE
 to_tsvector('black holes never exists before we think about them')
@@ q;
        q         |  tag
------------------+-------
 'black'          | color
 'black' & 'hole' | color
(2 rows)
```

Inverse FTS (FQS)

- RUM index supported – store branches of query tree in addinfo
  - Find queries for the first message in postgres mailing lists

```
\d pg_query
   Table "public.pg_query"
 Column |  Type   | Modifiers
--------+---------+-----------
 q      | tsquery |
 count  | integer |
Indexes:
    "pg_query_rum_idx" rum (q)                33818 queries


select q from pg_query pgq, pglist where q @@ pglist.fts and pglist.id=1;
            q
--------------------------
 'one' & 'one'
 'postgresql' & 'freebsd'
(2 rows)
```

Inverse FTS (FQS)

- Monstrous postings – top 5 posts matches most queries

```
select id, t.subject, count(*) as cnt into pglist_q  from pg_query,
(select id, fts, subject from pglist) t where t.fts @@ q
group by id, subject order by cnt desc limit 1000;

select * from pglist_q  order by cnt desc limit 5;
   id   |                   subject                    | cnt
--------+----------------------------------------------+------
 248443 | Packages patch                               | 4472
 282668 | Re: release.sgml, minor pg_autovacuum changes | 4184
 282512 | Re: release.sgml, minor pg_autovacuum changes | 4151
 282481 | release.sgml, minor pg_autovacuum changes    | 4104
 243465 | Re: [HACKERS] Re: Release notes              | 3989
(5 rows))
```

RUM vs GIN

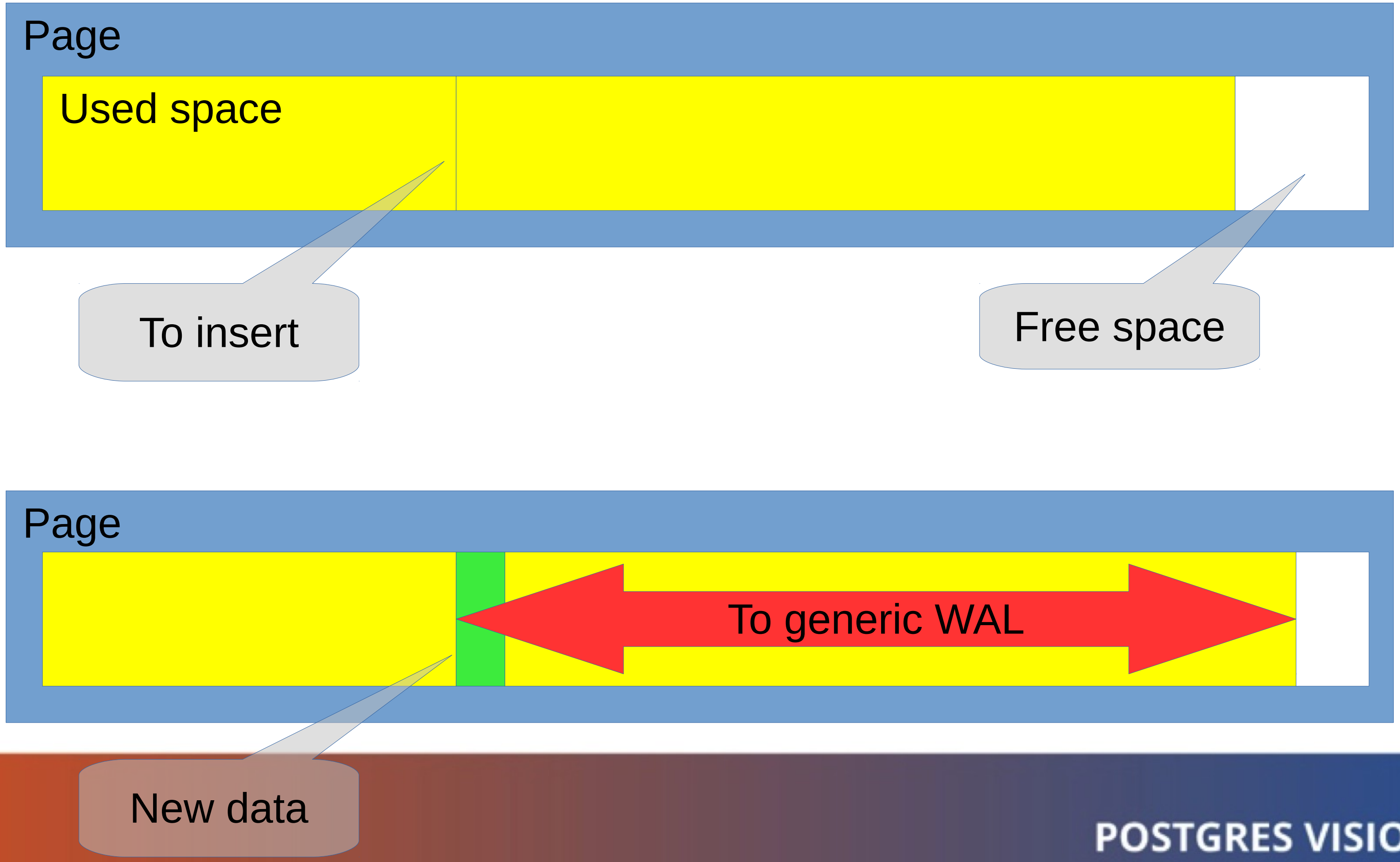
- 6 mln classifies, real fts quieries, concurrency 24, duration 1 hour
  - GIN — 258087
  - RUM — 1885698 (7x speedup)
- RUM has no pending list (not implemented) and stores more data.

  Insert 1 mln messages: `opt` — optimized config

	table	gin/opt	gin(no fast)	rum/opt	rum_nologged	gist
insert(min)	10	12/10	21	41/34	34	10.5
WAL size		9.5Gb/7.5	24Gb	37/29GB	41MB	3.5GB

RUM vs GIN

- CREATE INDEX
  - GENERIC WAL (9.6) generates too big WAL traffic



RUM vs GIN

- CREATE INDEX
  - GENERIC WAL(9.6) generates too big WAL traffic.

    It currently doesn't supports shift.
    rum(fts, ts+order) generates 186 Gb of WAL !
  - RUM writes WAL AFTER creating index

	table	gin	rum (fts	rum(fts,ts)	rum(fts,ts+order)
Create time		147 s	201	209	215
Size(mb)	2167/1302	534	980	1531	1921
WAL (Gb)		0.9	0.68	1.1	1.5

RUM Todo

- Allow multiple additional info
(lexemes positions + timestamp)
- add opclasses for array (similarity and as additional info) and int/float
- improve ranking function to support TF/IDF
- Improve insert time (pending list ?)
- Improve GENERIC WAL to support shift

Availability:

- 9.6+ only: https://github.com/postgrespro/rum

FTS dictionaries in shared memory

- Dictionaries are loaded into memory for every session (slow first query symptom) and eat memory.

```
time for i in {1..10}; do echo $i; psql postgres -c "select
ts_lexize('english_hunspell', 'evening')" > /dev/null;  done
1
2
3
4
5
6
7
8
9
10

real	0m0.656s
user	0m0.015s
sys	0m0.031s
```

For russian hunspell dictionary:

real 0m3.809s
user0m0.015s
sys 0m0.029s

Each session «eats» 20MB !

FTS dictionaries in shared memory

- Now it“s easy (Artur Zakirov, Postgres Professional + Thomas Vondra) https://github.com/postgrespro/shared_ispell

```
CREATE EXTENSION shared_ispell;
CREATE TEXT SEARCH DICTIONARY english_shared (
    TEMPLATE = shared_ispell,
    DictFile = en_us,
    AffFile = en_us,
    StopWords = english
);

time for i in {1..10}; do echo $i; psql postgres -c "select ts_lexize('russian_shared', 'туши')" > /dev/null;  done
1
2
…..
10

real  0m0.170s
user  0m0.015s
sys   0m0.027s
```

VS

```
real 0m3.809s
user0m0.015s
sys  0m0.029s
```

Dictionaries as extensions

- Now it's easy (Artur Zakirov, Postgres Professional) https://github.com/postgrespro/hunspell_dicts

```
CREATE EXTENSION hunspell_ru_ru; -- creates russian_hunspell dictionary
CREATE EXTENSION hunspell_en_us; -- creates english_hunspell dictionary
CREATE EXTENSION hunspell_nn_no; -- creates norwegian_hunspell dictionary
SELECT ts_lexize('english_hunspell', 'evening');
   ts_lexize
----------------
 {evening,even}
(1 row)

Time: 57.612 ms
SELECT ts_lexize('russian_hunspell', 'туши');
       ts_lexize
------------------------
 {туша,тушь,тушить,туш}
(1 row)

Time: 382.221 ms
SELECT ts_lexize('norwegian_hunspell','fotballklubber');
           ts_lexize
--------------------------------
 {fotball,klubb,fot,ball,klubb}
(1 row)

Time: 323.046 ms
```

Slow first query syndrom

Tsvector editing functions

- Stas Kelvich (Postgres Professional)
- setweight(tsvector, 'char', text[] - add label to lexemes from text[] array

```
select setweight( to_tsvector('english', '20-th anniversary of PostgreSQL'),
'A',  '{postgresql,20}');
                   setweight
------------------------------------------------
 '20':1A 'anniversari':3 'postgresql':5A 'th':2
(1 row)
```

- ts_delete(tsvector, text[]) - delete lexemes from tsvector

```
select ts_delete( to_tsvector('english', '20-th anniversary of PostgreSQL'),
'{20,postgresql}'::text[]);
       ts_delete
------------------------
 'anniversari':3 'th':2
(1 row)
```

Tsvector editing functions

- unnest(tsvector)

```
select * from unnest( setweight( to_tsvector('english',
'20-th anniversary of PostgreSQL'),'A',  '{postgresql,20}'));
   lexeme    | positions | weights
-------------+-----------+---------
 20          | {1}       | {A}
 anniversari | {3}       | {D}
 postgresql  | {5}       | {A}
 th          | {2}       | {D}
(4 rows)
```

- tsvector_to_array(tsvector) — tsvector to text[] array
  array_to_tsvector(text[])

```
select tsvector_to_array( to_tsvector('english',
'20-th anniversary of PostgreSQL'));
       tsvector_to_array
--------------------------------
 {20,anniversari,postgresql,th}
(1 row)
```

Tsvector editing functions

- ts_filter(tsvector,text[]) - fetch lexemes with specific label{s}

```
select ts_filter($$'20':2A 'anniversari':4C 'postgresql':1A,6A 'th':3$$::tsvector,
'{C}');
    ts_filter
------------------
 'anniversari':4C
(1 row)

select ts_filter($$'20':2A 'anniversari':4C 'postgresql':1A,6A 'th':3$$::tsvector,
'{C,A}');
                  ts_filter
-----------------------------------------------
 '20':2A 'anniversari':4C 'postgresql':1A,6A
(1 row)
```

Multilingual support

- New JOIN option for text search configuration mapping

```
CREATE TEXT SEARCH CONFIGURATION multi_conf (COPY=simple);
ALTER TEXT SEARCH CONFIGURATION multi_conf
ALTER MAPPING FOR asciiword, asciihword, hword_asciipart, word, hword, hword_part
 WITH german_hunspell (JOIN), english_hunspell;
```

Artur Zakirov, Postgres Professional

https://github.com/select-artur/pg_multilingual

Future of extendability: table engines

- Columnar table engine for OLAP
- Better OLTP table engine
  - Why?

Bottlenecks of OLTP performance

Some of OLTP bottlenecks

- Buffer manager: slow hash-table, pin, locks etc.
- Snapshots: for each new snapshot we have to iterate over each active transaction.
- Synchronous protocol.
- Slow xid allocation – a lot of locks.

PostgreSQL bottlenecks in numbers

- `SELECT val FROM tab WHERE id IN (:id1, ... :id10)` – 150K per second = 1.5M points per second, no gain. Bottleneck in locks.
- `10xSELECT 1` in single command – 2.2M queries per second. Taking snapshots is a bottleneck.
- `SELECT 1` with CSN patch (cheap snapshots) – 3.9M queries per second. Protocol is a bottleneck.
- `SELECT txid_current()`– 390K per second. Bottleneck in locks.

How can we improve PostgreSQL OLTP?

- Better table engine without buffer manager.
- CSN for faster snapshots.
- Asynchronous binary protocol for processing more short queries.
- Lockless xid allocation.

Conclusion

- Extendability was major innovation of Postgres as research project.
- Thanks to Open Source community, Postgres became production DBMS which saved its initial extendability, improved it and provided advanced features based on it.
- Time arises new challenges to Postgres. Answers should include another extendability improvement: pluggable table engines.

Thank you for attention!

You’ve Got Questions? We’ve Got Answers!