2016 杭州·云栖大会
THE COMPUTING CONFERENCE

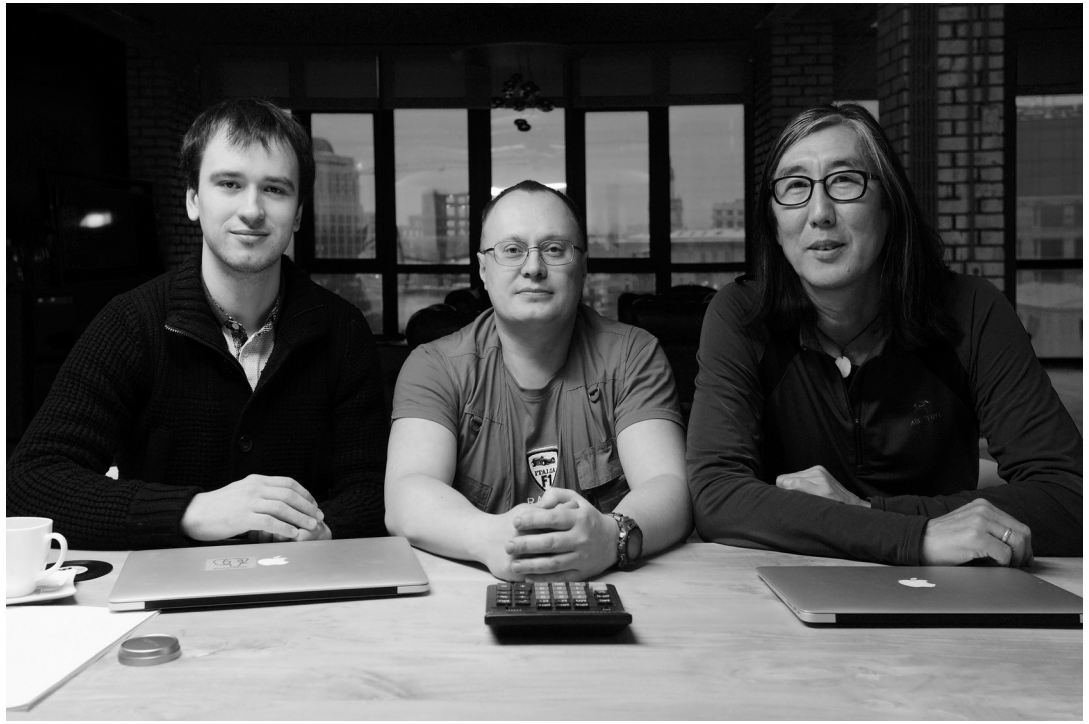
PostgreSQL 9.6 New advances in Full Text Search

Oleg Bartunov
Postgres Professional

主办单位：杭州 Alibaba Group 阿里巴巴集团 战略合作伙伴：intel

PROFESSIONAL Postgres

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, Hstore, Ltree

Postgres PROFESSIONAL

Agenda

- Initial design of Postgres and innovations
- History of some particular innovative features of Postgres
- Full Text Search in 9.6
- New RUM index

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. *

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. *

Extendability ...

Is 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 non-balanced trees
  - GIN is template for inverted indexes of composite objects
  - BRIN is template for bounding aggregates per block ranges

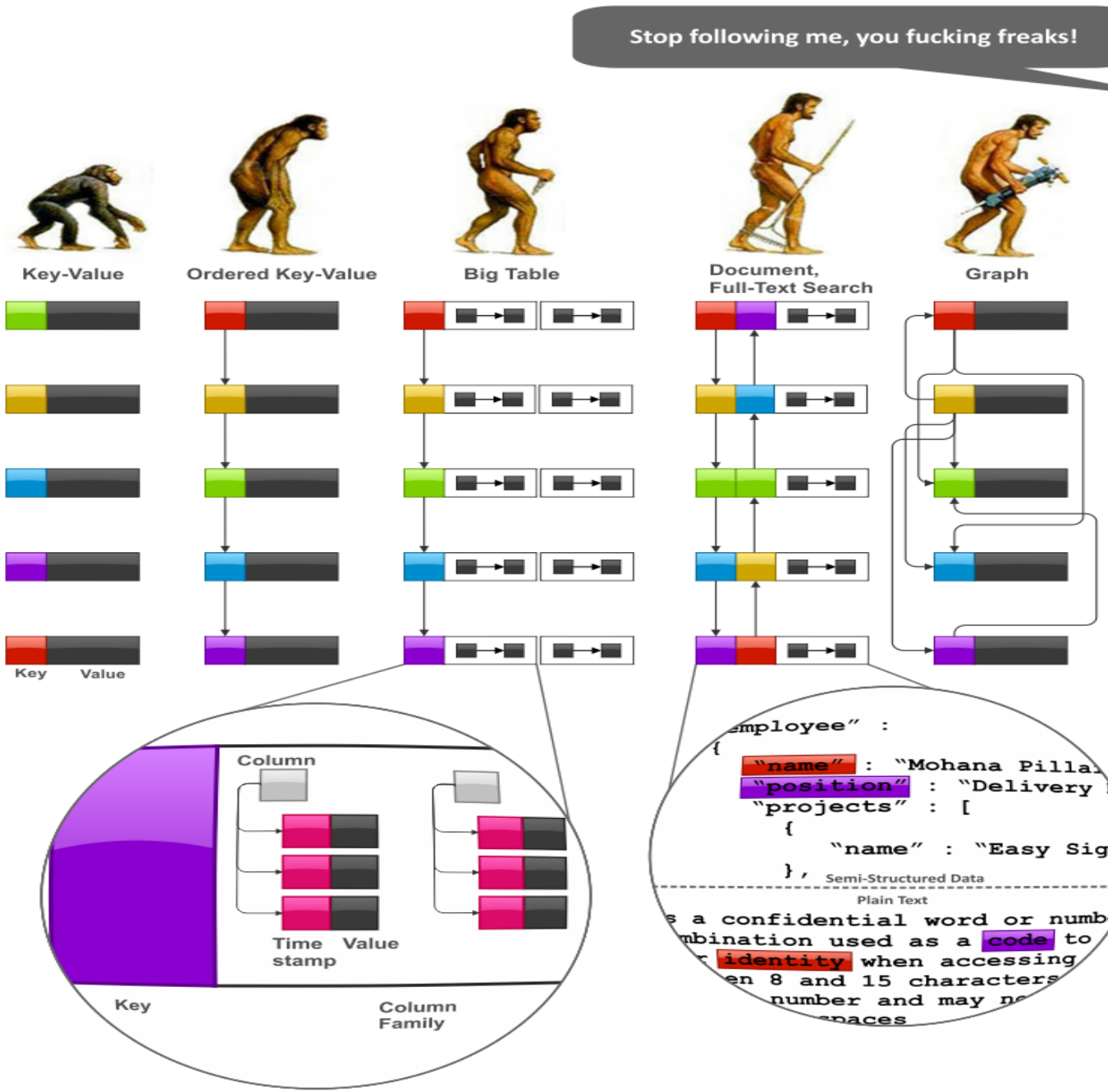
Propagation of improvements

- 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



PostgreSQL 9.4+

- Open-source
- Relational database
- Extendable database
- Strong support of NoSQL

Future of JSONB

Dictionary compression for jsonb

- Duplicate keys storage in jsonb is the problem.
- Pluggable compression mechanism (extendability!). Could be applied to any data type.
- Each jsonb column have own dictionary of keys. Conversion on the fly.
- Will be released soon by Postgres Professional.

Dictionary compression for jsonb

Customers reviews dataset

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

Less space than array version because of numerics compression!

Subscription for jsonb

- New query syntax:

```
UPDATE js SET js['key'] = 'value'
 WHERE  js['id'] = 1;
```

- Generic mecanism, extendable for any data type instead of hack for arrays we currently have.
- On commitfest by Postgres Professional

https://commitfest.postgresql.org/11/793/

K-Nearest Neighbors Search

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=1)
         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 !

Full Text Search

What is a Full Text Search ?

- Full text search
  - Find documents, which match a query
  - Sort them in some order (optionally)
- Typical Search
  - Find documents with **all words** from query
  - Return them sorted by relevance

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
- RUM – 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

http://www.postgresql.org/docs/current/static/textsearch.html

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 flexible search engine,
  but it is more than a complete solution

- It is a «collection of bricks» you can build your search engine with
  - 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

Some FTS problems: #1

156676 Wikipedia articles:

- Search is fast, ranking is slow.

```
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY rank DESC
LIMIT 3;
```

```
 Limit (actual time=476.106..476.107 rows=3 loops=1)
   Buffers: shared hit=149804 read=87416
   ->  Sort (actual time=476.104..476.104 rows=3 loops=1)
         Sort Key: (ts_rank(text_vector, '''titl'''::tsquery)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         Buffers: shared hit=149804 read=87416
         ->  Bitmap Heap Scan on ti2 (actual time=6.894..46[illegible]215 rows=47855 loops=1)
               Recheck Cond: (text_vector @@ '''titl'''::tsquery)
               Heap Blocks: exact=4913
               Buffers: shared hit=149804 rea[illegible]87416
               ->  Bitmap Index Scan on ti2_index (actual time=6.117..6.117 rows=47855 loops=1)
                     Index Cond: (text_vector @@ '''titl'''::tsquery)
                     Buffers: shared hit=1 read=12
 Planning time: 0.255 ms
 Execution time: 476.171 ms
(15 rows)
```

HEAP IS SLOW
470 ms !

Some FTS problems: #2

- No phrase search
  - “A & B” is equivalent to “B & A»
    There are only 92 posts with person 'Tom Good',
    but FTS finds 34039 posts

  - Combination of FTS + regular expression works, but slow and can be used only for simple queries.

Some FTS problems: #3

- Combine FTS with ordering by timestamp

```
SELECT sent,  subject from pglist
WHERE fts @@ to_tsquery('english', 'tom & lane')
ORDER BY abs(sent — '2000-01-01'::timestamp) ASC LIMIT 5;
```

```
 Limit (actual time=545.560..545.560 rows=5 loops=1)
   ->  Sort (actual time=545.559..545.559 rows=5 loops=1)
         Sort Key: (CASE WHEN ((sent - '2000-01-01 00:00:00'::timestamp without time zone) < '00:00:00'::interval) THEN (-
(sent - '2000-01-01 00:00:00'::timestamp without time zone)) ELSE (sent - '2000-01-01 00:00:00'::timestamp without time
zone) END)
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Bitmap Heap Scan on pglist (actual time=87.545..507.897 rows=222813 loops=1)
               Recheck Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
               Heap Blocks: exact=105992
               ->  Bitmap Index Scan on pglist_gin_idx (actual time=57.932..57.932 rows=222813 loops=1)
                     Index Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
 Planning time: 0.376 ms
 Execution time: 545.744 ms
```

```
          sent          |                                                        subject
------------------------+-------------------------------------------------------------------
 1999-12-31 13:52:55 | Re: [HACKERS] LIKE fixed(?) for non-ASCII collation orders
 2000-01-01 11:33:10 | Re: [HACKERS] dubious improvement in new psql
 1999-12-31 10:42:53 | Re: [HACKERS] LIKE fixed(?) for non-ASCII collation orders
 2000-01-01 13:49:11 | Re: [HACKERS] dubious improvement in new psql
 1999-12-31 09:58:53 | Re: [HACKERS] LIKE fixed(?) for non-ASCII collation orders
(5 rows)

Time: 568.357 ms
```

No positions in index !

Inverted Index in PostgreSQL

ENTRY TREE

Report Index

A

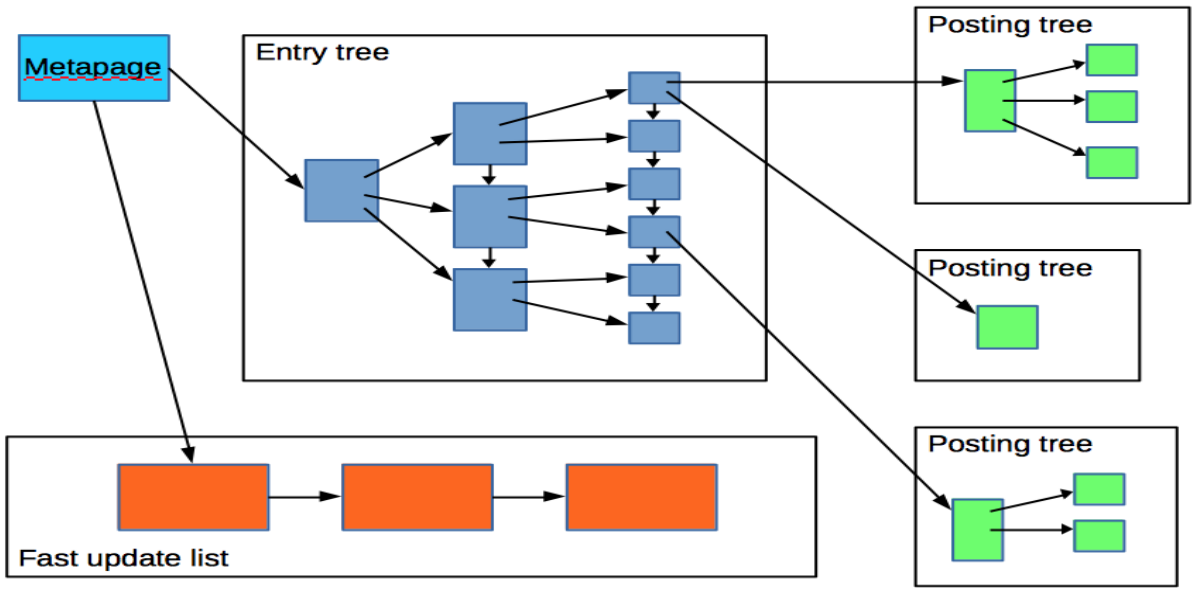
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B

backward wave oscillators, 45

compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
convertors, 22
coplanar waveguide components, 40
Couette flow, 21
creep, 17
crystallisation, 64

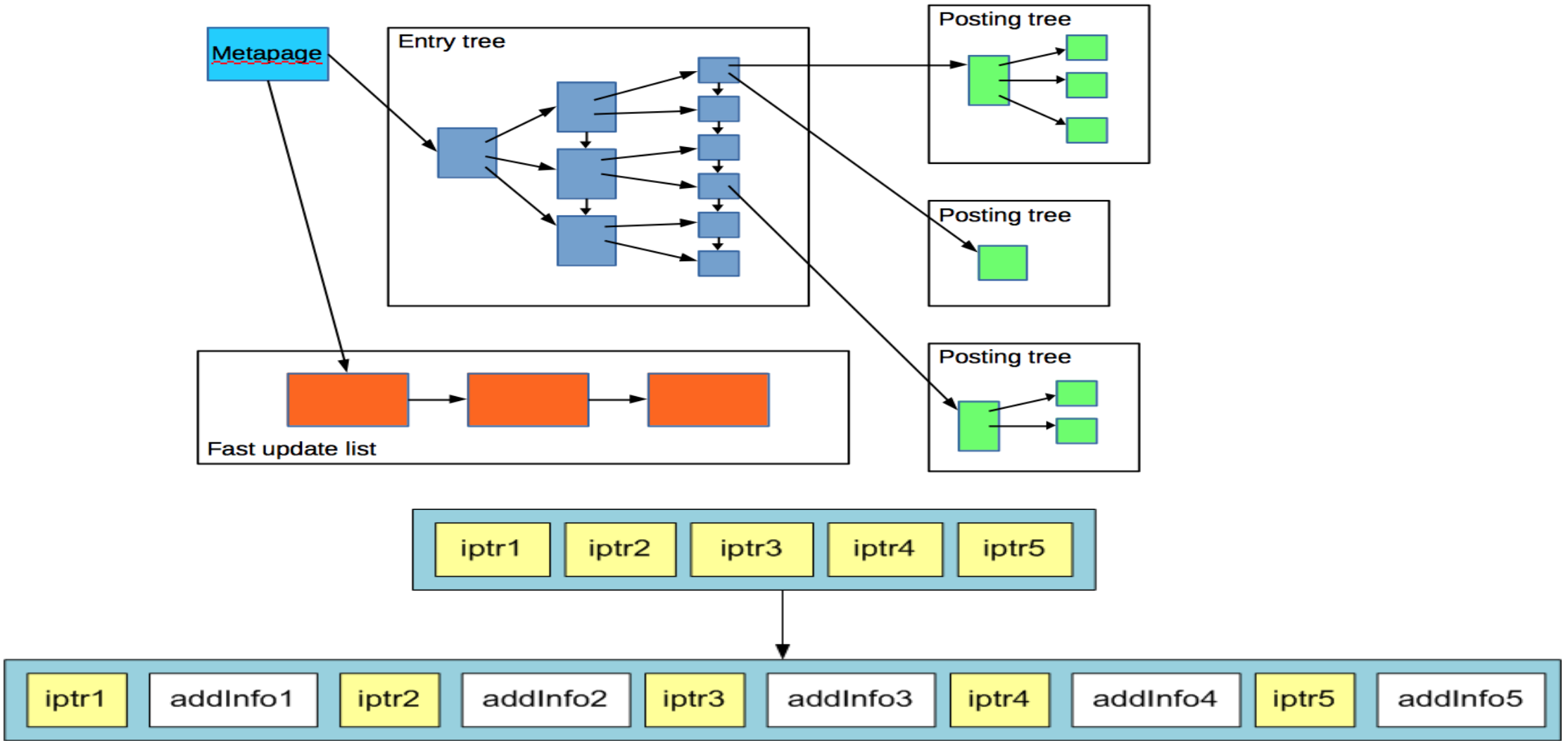
Posting list
Posting tree



Inproving GIN

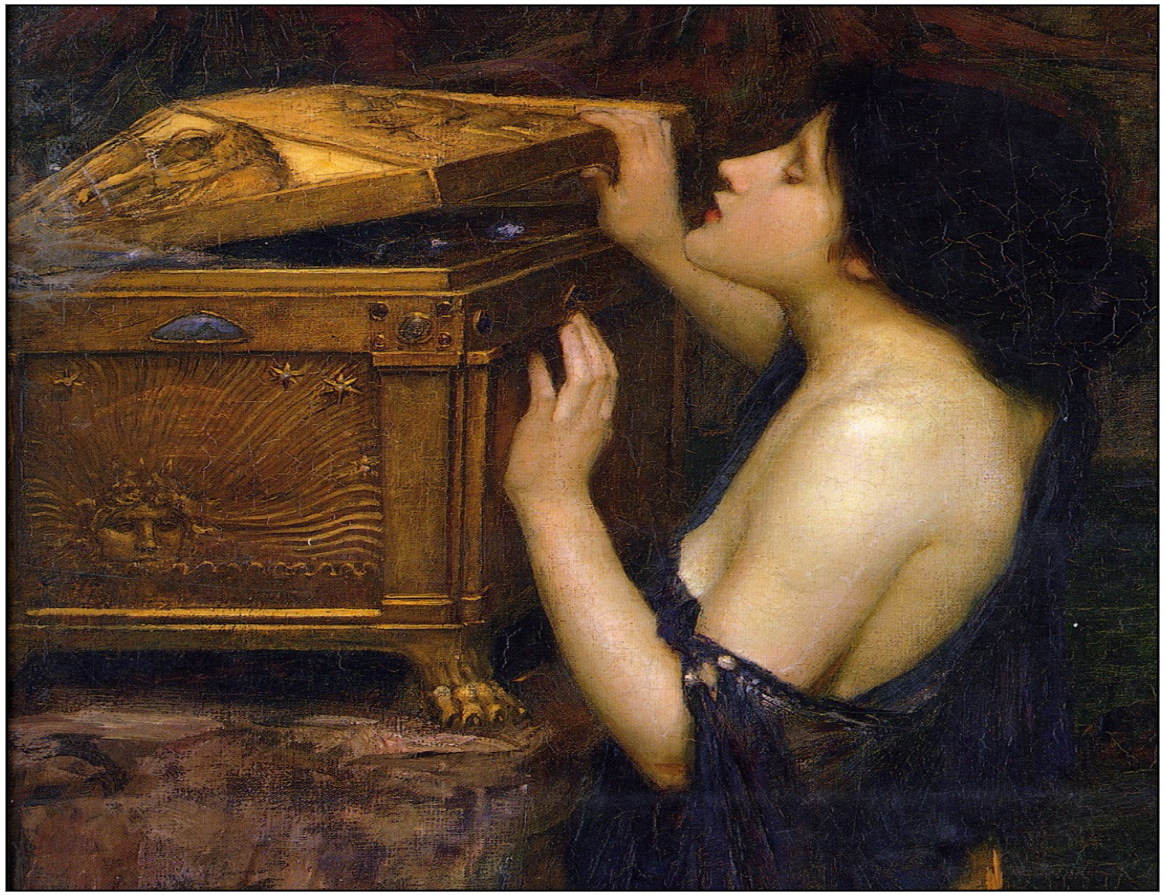
- Improve GIN index
  - Store additional information in posting tree, for example, lexemes positions or timestamps
  - Use this information to order results

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

```
CREATE INDEX ti2_rum_fts_idx ON ti2 USING rum(text_vector rum_tsvector_ops);

SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY
text_vector <=> plainto_tsquery('english','title') LIMIT 3;
                                        QUERY PLAN
-----------------------------------------------------------------------------------
 L Limit (actual time=54.676..54.735 rows=3 loops=1)
   Buffers: shared hit=355
   ->  Index Scan using ti2_rum_fts_idx on ti2 (actual time=54.675..54.733 rows=3 loops=1)
         Index Cond: (text_vector @@ '''titl'''::tsquery)
         Order By: (text_vector <=> '''titl'''::tsquery)
         Buffers: shared hit=355
 Planning time: 0.225 ms
 Execution time: 54.775 ms vs 476 ms !
(8 rows)
```

CREATE INDEX ... USING RUM

- Top-10 (out of 222813) postings with «Tom Lane»
  - GIN index — 1374.772 ms

```
SELECT subject, ts_rank(fts,plainto_tsquery('english', 'tom lane')) AS rank
FROM pglist WHERE fts @@ plainto_tsquery('english', 'tom lane')
ORDER BY rank DESC LIMIT 10;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Limit (actual time=1374.277..1374.278 rows=10 loops=1)
   ->  Sort (actual time=1374.276..1374.276 rows=10 loops=1)
         Sort Key: (ts_rank(fts, '''tom'' & ''lane'''::tsquery)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Bitmap Heap Scan on pglist (actual time=98.413..1330.994 rows=222813 loops=1)
               Recheck Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
               Heap Blocks: exact=105992
               ->  Bitmap Index Scan on pglist_gin_idx (actual time=65.712..65.712
rows=222813 loops=1)
                     Index Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
 Planning time: 0.287 ms
 Execution time: 1374.772 ms
(11 rows)
```

CREATE INDEX ... USING RUM

- Top-10 (out of 222813) postings with «Tom Lane»
  - RUM index — 216 ms vs 1374 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 loops=1)
         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
(6 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

Results for queries 'A & B' and 'B & A' should be different !

- Introduce new FOLLOWED BY (<->) operator:
  - Guarantee an order of operands
  - Distance between operands

a <n> b == a & b & (∃ 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
- 'A <-> B' = 'A<1>B'
- 'A <0> B' matches the word with two different forms (infinitives)
- `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)
```

Phrase search - properties

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

Use parenthesis to control nesting in tsquery

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

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

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

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 !

Some FTS problems: #3

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

RUM Todo

- Allow multiple additional info
- 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

More details about new FTS features
http://www.sai.msu.su/~megera/postgres/talks/pgopen-2016-rum.pdf

2016 The Computing Conference

THANKS

感谢大家！

THANKS FOR YOUR ATTENTION