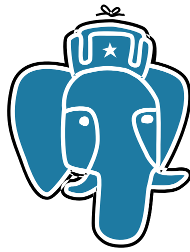


FTS is DEAD ? Long live FTS !

Oleg Bartunov, Teodor Sigaev
Postgres Professional, Moscow University

PGCon, April 20, 2016, Ottawa

PROFESSIONAL Postgres

Russian PostgreSQL developers

Oleg Bartunov, Teodor Sigaev

PostgreSQL CORE

- Locale support
- PostgreSQL extendability: GiST(KNN), GIN, SP-GiST
- Full Text Search (FTS)
- NoSQL (hstore, jsonb)
- VODKA access method (WIP)

Расширения:

- Intarray
- Pg_trgm
- Ltree
- Hstore
- plantuner
- JsQuery

Postgres PROFESSIONAL

Agenda

- Why built-in FTS in database
- Full text search in PostgreSQL
- Some FTS problems
- New features:
  - CREATE INDEX … USING RUM
  - Phrase search
  - Inverse FTS
  - Dictionaries as extensions
  - Dictionaries in shared memory
  - Tsvector editing functions

PostgresPROFESSIONAL

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

PostgresPROFESSIONAL

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

Postgres PROFESSIONAL

FTS in Databases

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

Postgres PROFESSIONAL

- 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)

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

Postgres PROFESSIONAL

FTS in PostgreSQL

What is the benefit ?

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

- Document parsed into tokens using pluggable parser
- Tokens converted to lexems using pluggable dictionaries
- Words coordinates and importance are stored and used for ranking
- Stop-words ignored

Postgres PROFESSIONAL

FTS in PostgreSQL

- Query processed in 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)'`
  - Query can be rewritten «on-the-go»

Postgres PROFESSIONAL

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
  - + All power of SQL (FTS+Spatial+Temporal)
  - Use tsvector as a custom storage

  - For example, instead of textual documents consider chemical formulas or genome string

PostgresPROFESSIONAL

Some FTS problems #1

156676 Wikipedia articles:

- Search is fast, ranking is slow.

postgres=# explain analyze
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;

HEAP IS SLOW
400 ms !

```
 Limit  (cost=8087.40..8087.41 rows=3 width=282) (actual time=433.750..433.752 rows=3 loops=
   ->  Sort  (cost=8087.40..8206.63 rows=47692 width=282)
(actual time=433.749..433.749 rows=3 loops=1)
         Sort Key: (ts_rank(text_vector, '''titl'''::tsquery))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Bitmap Heap Scan on ti2  (cost=529.61..7470.99 rows=47692 width=282)
(actual time=15.094..423.452 rows=47855 loops=1)
               Recheck Cond: (text_vector @@ '''titl'''::tsquery)
               ->  Bitmap Index Scan on ti2_index  (cost=0.00..517.69 rows=47692 width=0)
(actual time=13.736..13.736 rows=47855 loops=1)
                     Index Cond: (text_vector @@ '''titl'''::tsquery)
 Total runtime: 433.787 ms
```

Some FTS problems #2

- No phrase search
  - “A & B” is equivalent to “B & A»
  - 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 abs(sent - '2000-01-01'::timestamp) asc limit 5;

QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 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

(11 rows)
         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
```

Some FTS problems #4

- Working with dictionaries can be difficult and slow
  - Installing dictionaries can be complicated
  - 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 !

Postgres PROFESSIONAL

There are always more other problems !

PROFESSIONAL Postgres

Inproving GIN

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

Postgres PROFESSIONAL

No positions in index !

Inverted Index in PostgreSQL

ENTRY TREE

Report Index

A

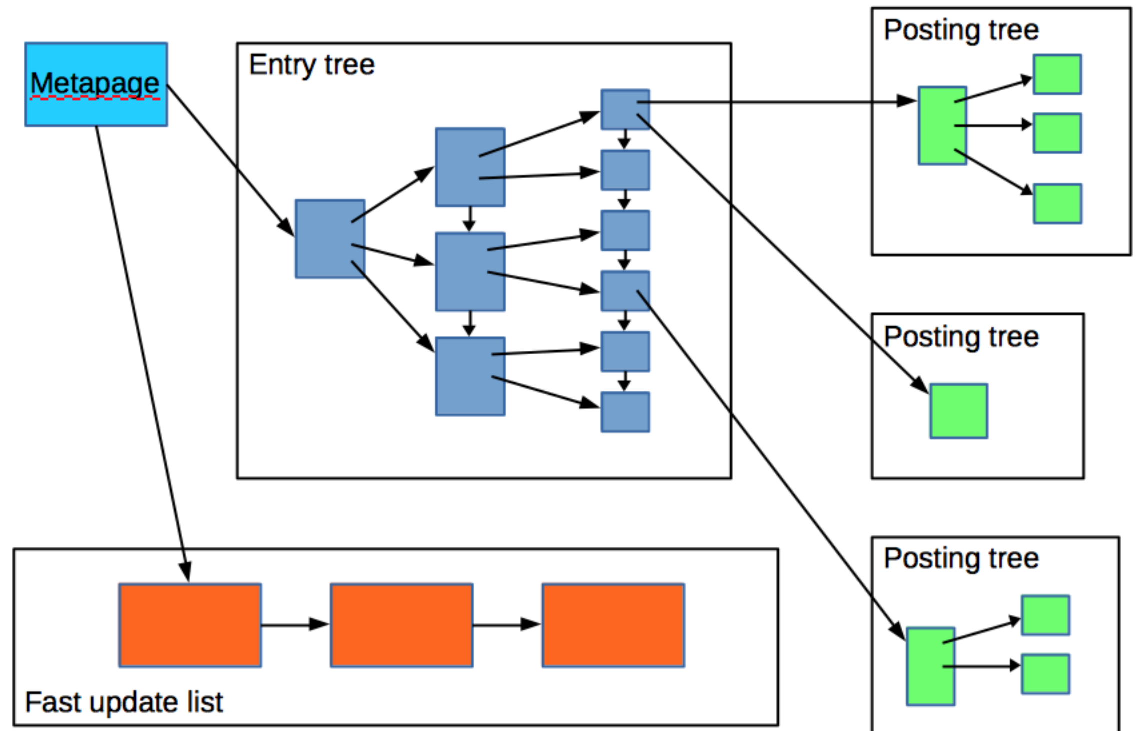
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 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

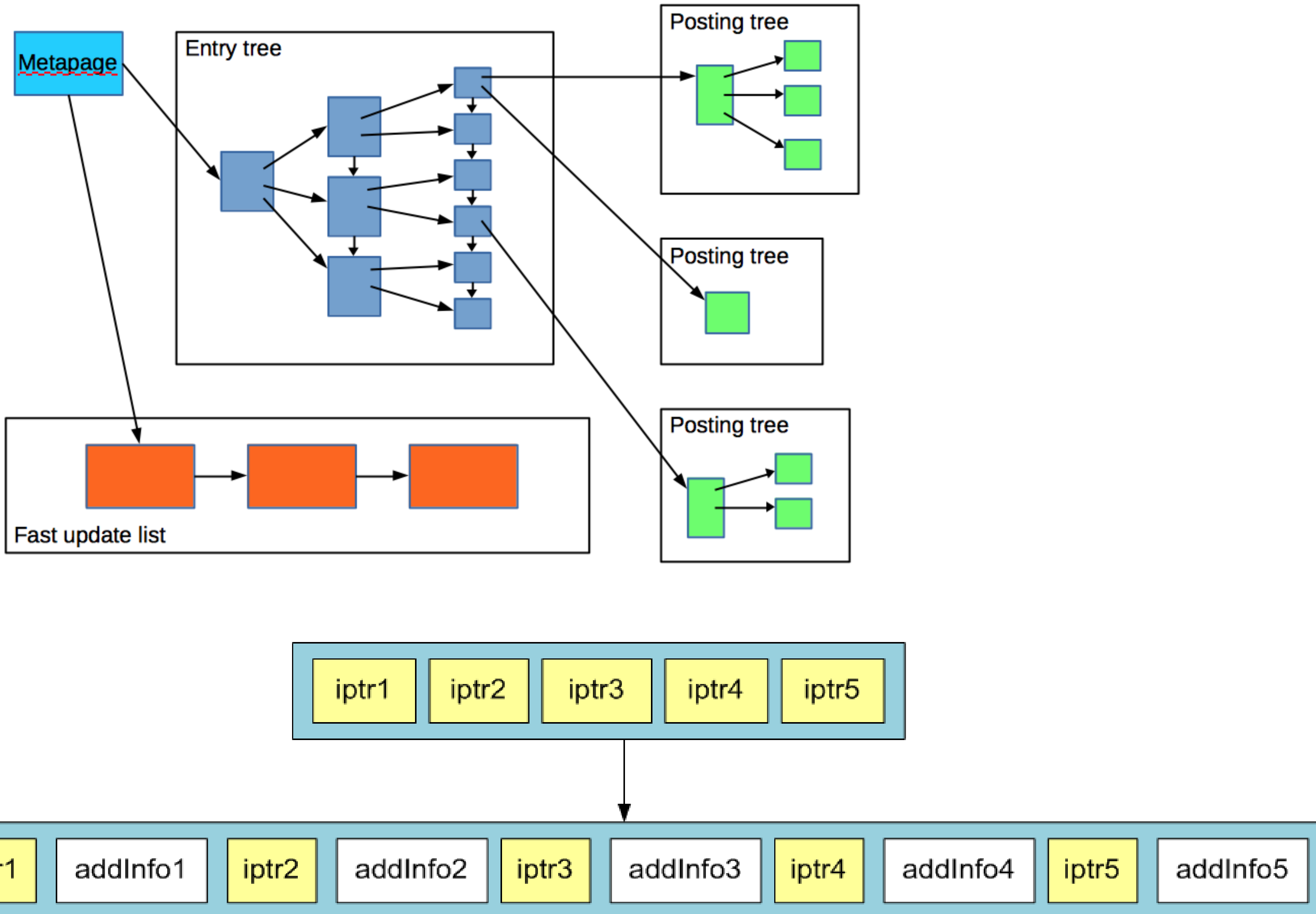
backward wave oscillators, 45

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



Improving GIN



9.6 opens «Pandora box»

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

PROFESSIONAL Postgres

CREATE INDEX ... USING RUM

- Use position information 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
------------------------------------------------------------------------------------------
 Limit (actual time=13.843..13.884 rows=3 loops=1)
   ->  Index Scan using ti2_rum_fts_idx on ti2 (actual time=13.841..13.881 rows=3 loops=1)
         Index Cond: (text_vector @@ '''titl'''::tsquery)
         Order By: (text_vector <-> '''titl'''::tsquery)
 Planning time: 0.134 ms
 Execution time: 14.030 ms vs 433 ms !
(6 rows)
```

Postgres PROFESSIONAL

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)
```

PostgresPRO PROFESSIONAL

CREATE INDEX ... USING RUM

- Top-10 (out of 222813) postings with «Tom Lane»
  - RUM index — 102.161 ms vs 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=101.381..101.486 rows=10 loops=1)
   ->  Index Scan using pglist_rum_fts_idx on pglist (actual time=101.380..101.485
rows=10 loops=1)
         Index Cond: (fts @@ plainto_tsquery('tom lane'::text))
         Order By: (fts <-> plainto_tsquery('tom lane'::text))
 Planning time: 0.282 ms
 Execution time: 102.161 ms
(6 rows)
```

Postgres PROFESSIONAL

6.7 mln classifieds

	Without patch	With patch	With patch functional index	Sphinx
Table size	6.0 GB	6.0 GB	2.87 GB	-
Index size	1.29 GB	1.27 GB	1.27 GB	1.12 GB
Index build time	216 sec	303 sec	718sec	180 sec*
Queries in 8h 9.2+patch (9.6+rum)	**3.0 mln. (27 mln)**	**42.7 mln. (41 mln)**	**42.7 mln.**	**32.0 mln. (61 mln)**

20 mln descriptions

	Without patch	With patch	With patch functional index	Sphinx
Table size	18.2 GB	18.2 GB	11.9 GB	-
Index size	2.28 GB	2.30 GB	2.30 GB	3.09 GB
Index build time	258 sec	684 sec	1712 sec	481 sec*
Queries in 8 h 9.2 +patch (9.6 rum)	**2.67 mln (95 mln)**	**38.7 mln. (136 mln)**	**38.7 mln.**	**26.7 mln. (59 mln)**

Postgres PROFESSIONAL

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 PHRASE (<->) operator:
  - Guarantee an order of operands
  - Distance between operands

a <n> b == a & b & (∃ i,j : pos(b)i – pos(a)j <= n)

A <n> B - A «phrase n» B

Phrase search - definition

- Phrase operator returns:
  - false
  - true and array of positions of the **right** operand, which satisfy distance condition (without positional information <-> is equivalent to &)

```
A <-> B ≠ B <-> A
```

Postgres PROFESSIONAL

Phrase search - properties

- 'A <n> B <m> C' → '(A <n> B) <m> C' → matched phrase length ≤ n + m
- 'A <n> (B <m> C)’ → matched phrase length ≤ n
- '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
```

Postgres PROFESSIONAL

Phrase search - example

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

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

Notice, stop words taken into account !

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

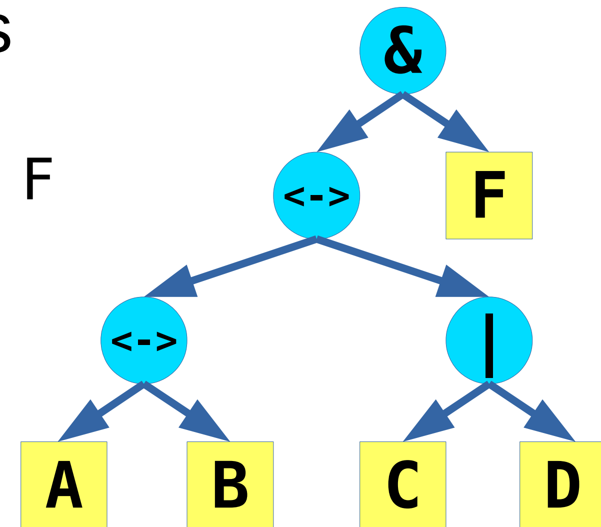

```
select phraseto_tsquery('english','PostgreSQL can be extended by the user in many ways');
                   phraseto_tsquery
-----------------------------------------------------------------------
 ( ( ( 'postgresql' <3> 'extend' ) <3> 'user' ) <2> 'mani' ) <-> 'way'
(1 row)
```

Postgres PROFESSIONAL

Phrase search - internals

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

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

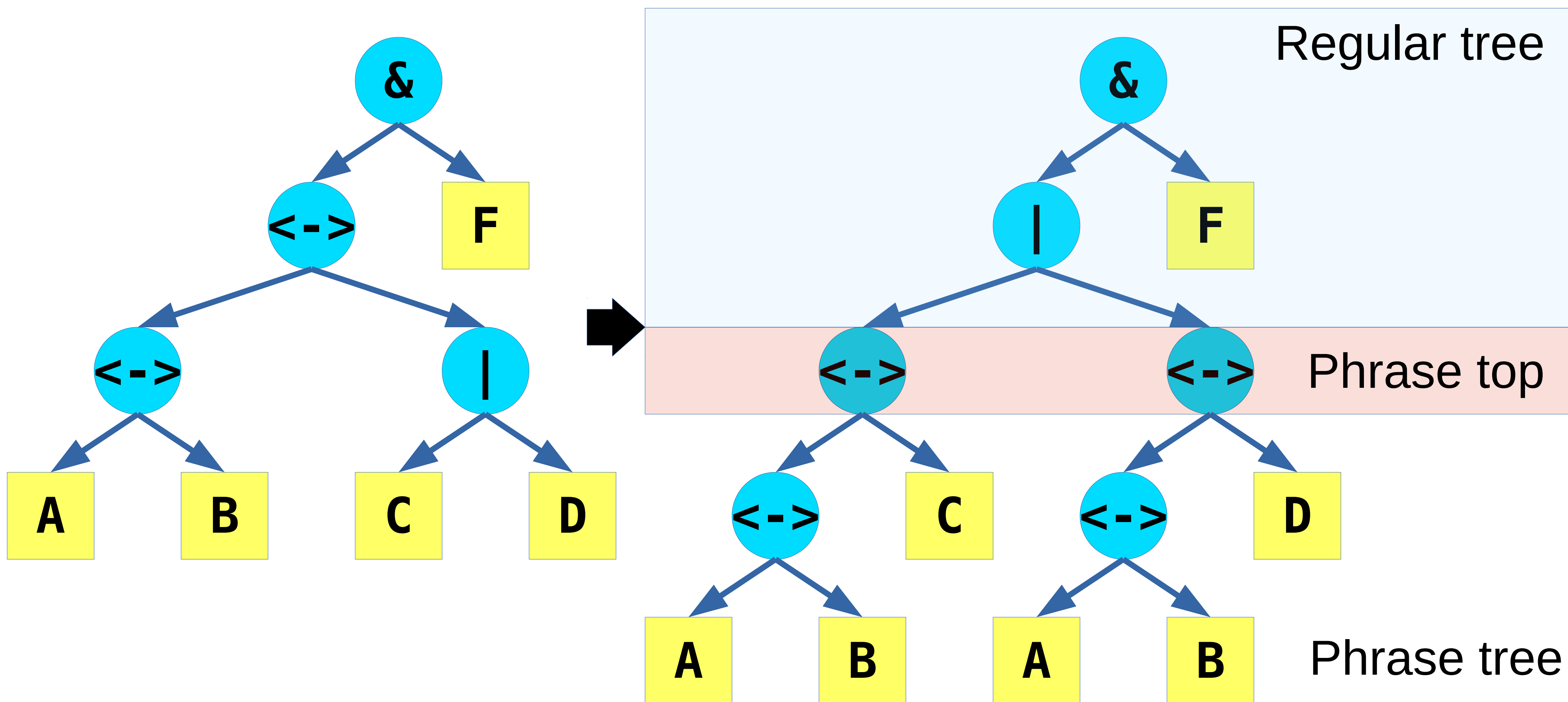


- We want to avoid slowdown FTS operators (& |), which do not need positions.
- Rewrite query, so any <-> operators pushed down in query tree and call phrase executor for the top <-> operator.

Postgres PROFESSIONAL

Phrase search - transformation

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



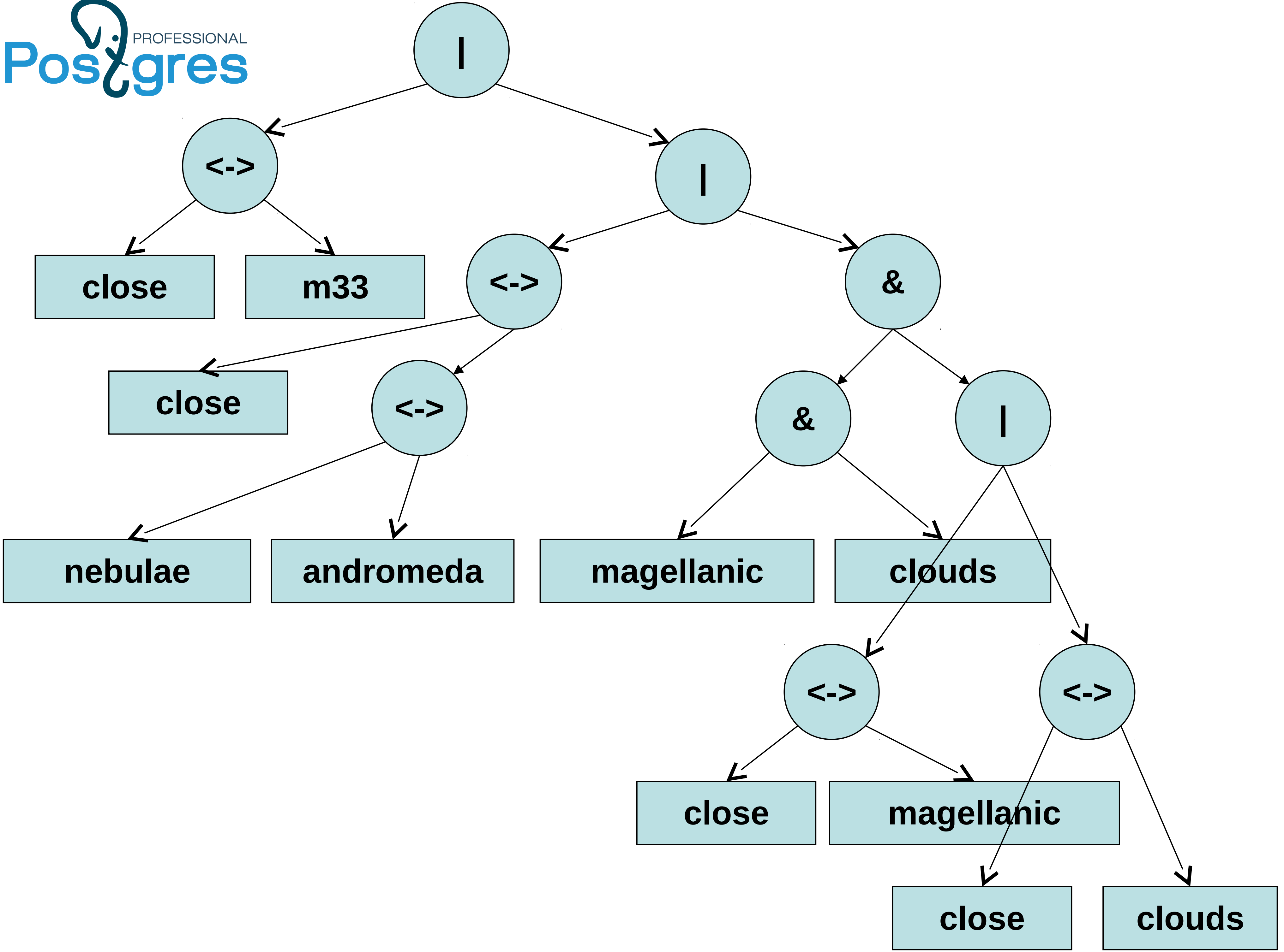
PostgresPROFESSIONAL

Example

Query: close <-> galaxies

After dictionary: close <-> (m33 |
(andromeda <-> nebulae | (magellanic & clouds))

Phrase: close <-> m33 |
(close <-> (andromeda <-> nebulae)) |
(magellanic & clouds &
(close <-> magellanic |
close <-> clouds
)
)



PROFESSIONAL
Postgres
|
<->
|
close
m33
<->
&
close
<->
&
|
nebulae
andromeda
magellanic
clouds
<->
<->
close
magellanic
close
clouds

Postgres PROFESSIONAL

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

PostgresPROFESSIONAL

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)

Sequential Scan (1.7 s (<-> vs 1.6 s (&)):

```
select count(*) from pglist where fts @@ to_tsquery('english', 'tom <-> lane');
                                           QUERY PLAN
--------------------------------------------------------------------------------
 Finalize Aggregate (actual time=1700.280..1700.280 rows=1 loops=1)
   ->  Gather (actual time=1700.228..1700.277 rows=3 loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate (actual time=1696.119..1696.119 rows=1 loops=3)
               ->  Parallel Seq Scan on pglist (actual time=2.356..1683.499 rows=74259
loops=3)
                     Filter: (fts @@ '''tom'' <-> ''lane'''::tsquery)
                     Rows Removed by Filter: 263664
 Planning time: 0.270 ms
 Execution time: 1709.092 ms
(10 rows)
```

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)

GIN index (1.1 s (<->) vs 0.48 s (&)): Use recheck, phrase is slow vs fts

```
select count(*) from pglist where fts @@ to_tsquery('english', 'tom <-> lane');
                                                 QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate (actual time=1074.983..1074.984 rows=1 loops=1)
   ->  Bitmap Heap Scan on pglist (actual time=84.424..1055.770 rows=222777 loops=1)
         Recheck Cond: (fts @@ '''tom'' <-> ''lane'''::tsquery)
         Rows Removed by Index Recheck: 36
         Heap Blocks: exact=105992
         ->  Bitmap Index Scan on pglist_gin_idx (actual time=53.628..53.628 rows=222813 
loops=1)
               Index Cond: (fts @@ '''tom'' <-> ''lane'''::tsquery)
 Planning time: 0.329 ms
 Execution time: 1075.157 ms
(9 rows)
```

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)
```

RUM index (0.5 s (<-> vs 0.48 s (&)): Use positions in addinfo, no overhead of phrase search !

```
select count(*) from pglist where fts @@ to_tsquery('english', tom <-> lane');
                                              QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate (actual time=513.517..513.517 rows=1 loops=1)
   ->  Bitmap Heap Scan on pglist (actual time=134.109..497.814 rows=221919 loops=1)
         Recheck Cond: (fts @@ to_tsquery('tom <-> lane'::text))
         Heap Blocks: exact=105509
         ->  Bitmap Index Scan on pglist_rum_fts_idx (actual time=98.746..98.746
rows=221919 loops=1)
               Index Cond: (fts @@ to_tsquery('tom <-> lane'::text))
 Planning time: 0.223 ms
 Execution time: 515.004 ms
(8 rows)
```

PostgresPROFESSIONAL

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)

- RUM index supported – store branches of query tree in addinfo

Find queries for the first message in postgres mailing lists

```
create index pg_query_rum_idx on pg_query using rum(q);
select q from pg_query pgq, pglist where q @@ pglist.fts and pglist.id=1;
                                          QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop (actual time=0.719..0.721 rows=2 loops=1)
   ->  Index Scan using pglist_id_idx on pglist
(actual time=0.013..0.013 rows=1 loops=1)
         Index Cond: (id = 1)
   ->  Bitmap Heap Scan on pg_query pgq
(actual time=0.702..0.704 rows=2 loops=1)
         Recheck Cond: (q @@ pglist.fts)
         Heap Blocks: exact=2
         ->  Bitmap Index Scan on pg_query_rum_idx
(actual time=0.699..0.699 rows=2 loops=1)
               Index Cond: (q @@ pglist.fts)
 Planning time: 0.212 ms
 Execution time: 0.759 ms
(10 rows)
```

PROFESSIONAL Postgres

Inverse FTS (FQS)

- RUM index supported – store branches of query tree in addinfo

Monstrous postings

```
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))
```

PROFESSIONAL Postgres

Some FTS problems #3

- Combine FTS with ordering by timestamp
  - Store timestamps in additional information
  - Order posting tree/list by timestamp (not yet)

```
create index pglist_tsvtime_rum_idx on pglist using rum(fts
rum_tsvector_timestamp_ops, sent) WITH (orderby = 'sent', addto =
'fts');

select sent,  subject from pglist
where fts @@ to_tsquery('tom & lane')
order by sent <-> '2000-01-01'::timestamp limit 5;
                                             QUERY PLAN
---------------------------------------------------------------------
 Limit (actual time=89.499..89.505 rows=5 loops=1)
   ->  Index Scan using pglist_tsvtime_rum_idx on pglist (actual
time=89.498..89.504 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: 0.286 ms
 Execution time: 90.317 ms vs 545 ms !
```

Postgres PROFESSIONAL

RUM Todo

- Ordering in posting tree not by item pointer
  30%-50% speedup for timestamp ordering
- Allow multiple additional info
  (lexemes positions + timestamp)
- Fix: Need WHERE clause for ordering
- add opclasses for array (similarity and as additional info) and int/float
- improve ranking function to support TF/IDF
- 9.6+ only: https://github.com/postgrespro/rum

Dictionaries as extensions

- Now it“s easy (Artur Zakirov, PostgresPro) 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

Dictionaries in shared memory

- Now it“s easy (Artur Zakirov, PostgresPro + 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
);
CREATE TEXT SEARCH DICTIONARY russian_shared (
    TEMPLATE = shared_ispell,
    DictFile = ru_ru,
    AffFile = ru_ru,
    StopWords = russian
);
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
```

Tsvector editing functions

- Stas Kelvich (PostgresPro)
- setweight(tsvector, «char», text[] - add label to lexemes from text[]

```
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_to_tsvector(text[])

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

PostgresPROFESSIONAL

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)
```

Thanks !

PROFESSIONAL Postgres

THANKS !