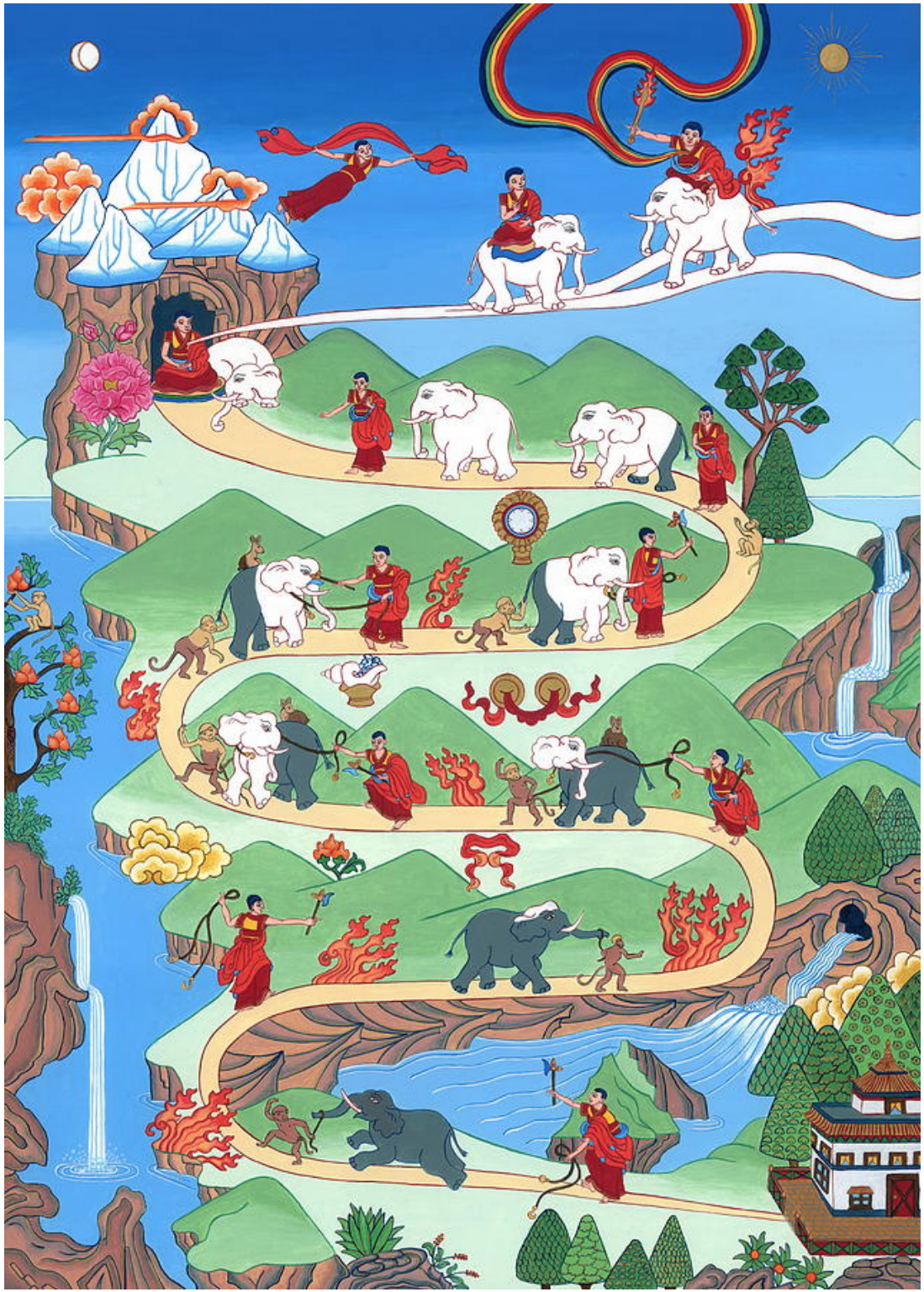
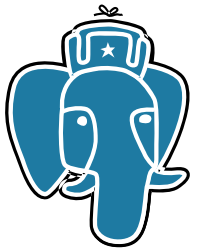




# Новые возможности FTS в PostgreSQL

Oleg Bartunov
Postgres Professional, Moscow University

Highload, Nov 8, 2016, Moscow

# FTS in PostgreSQL

- FTS is a powerful built-in text search engine
- No new features since 2006 !
- Popular complaints:
  - Slow ranking
  - No phrase search
  - No efficient alternate ranking
  - Working with dictionaries is tricky
  - Dictionaries are stored in the backend“s memory
  - FTS is flexible, but not enough

# 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
- 2016  — Postgres Professional

# FTS in PostgreSQL

- **tsvector** – data type for document optimized for search
- **tsquery** – textual data type for rich query language
- **Full text search operator:** tsvector @@ tsquery
- **SQL interface** to FTS objects (CREATE, ALTER)
  - Configuration: {tokens, {dictionaries}}
  - Parser: {tokens}
  - Dictionary: tokens → lexeme{s}
- **Additional functions and operators**
- **Indexes**: GiST, GIN, RUM

```
to_tsvector('english','a fat cat sat on a mat and ate a fat rat')
                              @@
to_tsquery('english','(cats | rat) & ate & !mice');
```

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..469.215 rows=47855 loops=1)
               Recheck Cond: (text_vector @@ '''titl'''::tsquery)
               Heap Blocks: exact=4913
               Buffers: shared hit=149804 read=87416
               ->  Bitmap Index Scan on ti2_index (actual time=6.117..6.117 rows=47855 loops
                     Index Cond: (text_vector @@ '''titl'''::tsquery)
                     Buffers: shared hit=1 read=12
 Planning time: 0.255 ms
 Execution time: 476.171 ms
(15 rows)
```

# 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

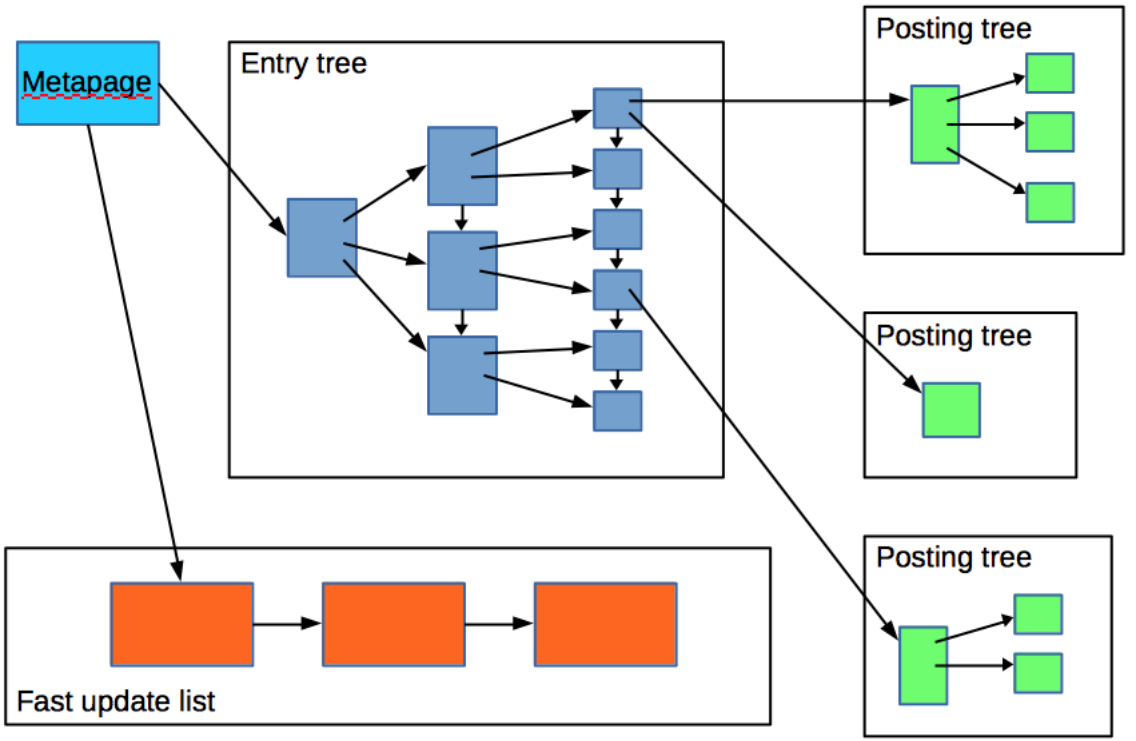
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics,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

**Posting list**
**Posting tree**

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

### B

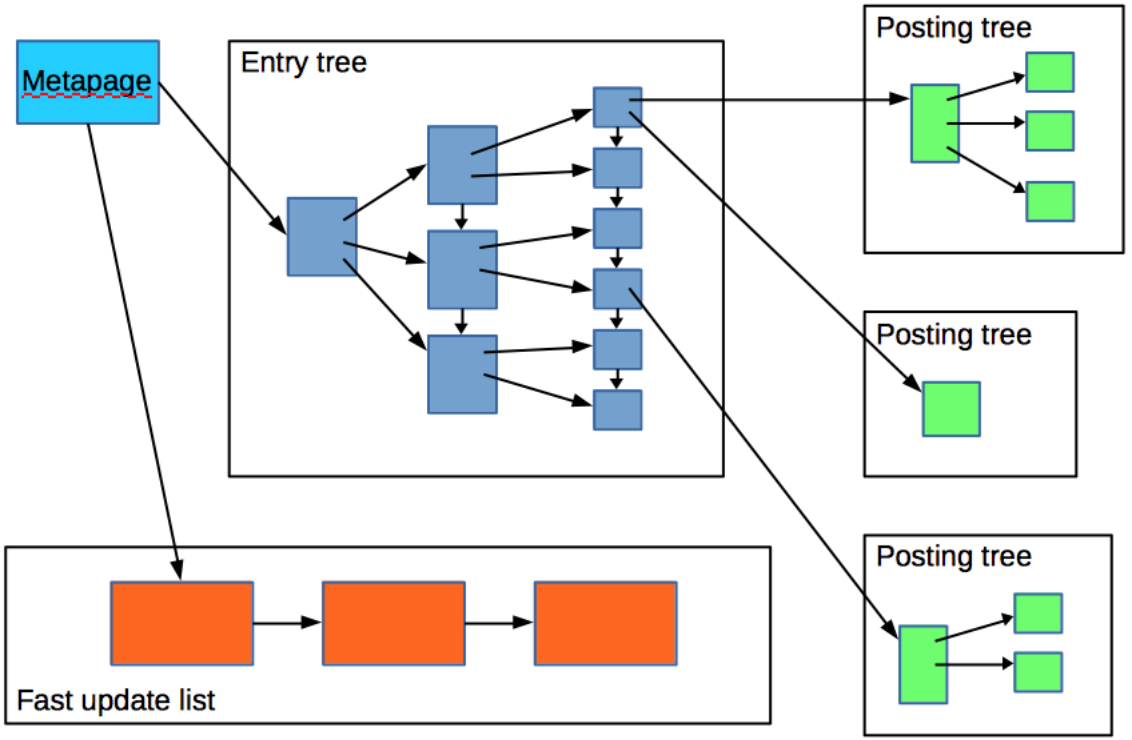
backward wave oscillators, 45

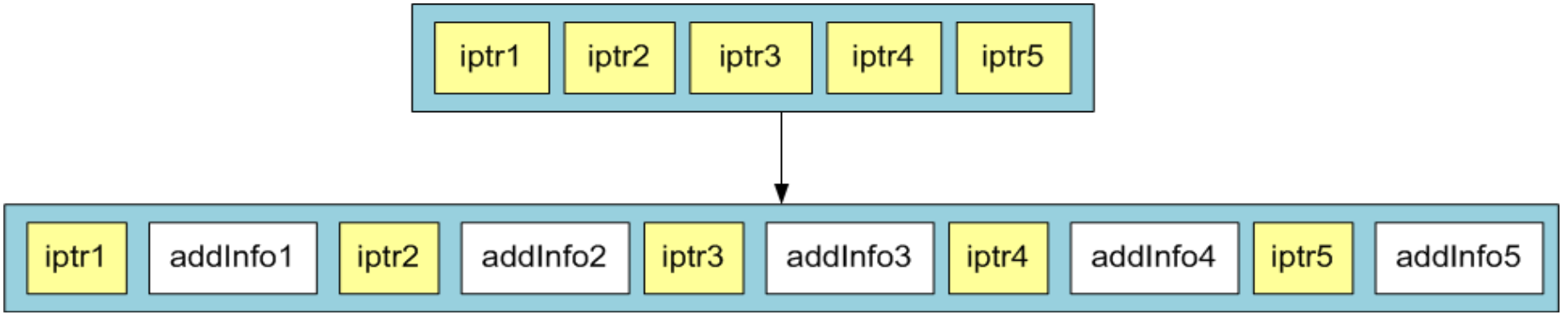


# Inproving GIN

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

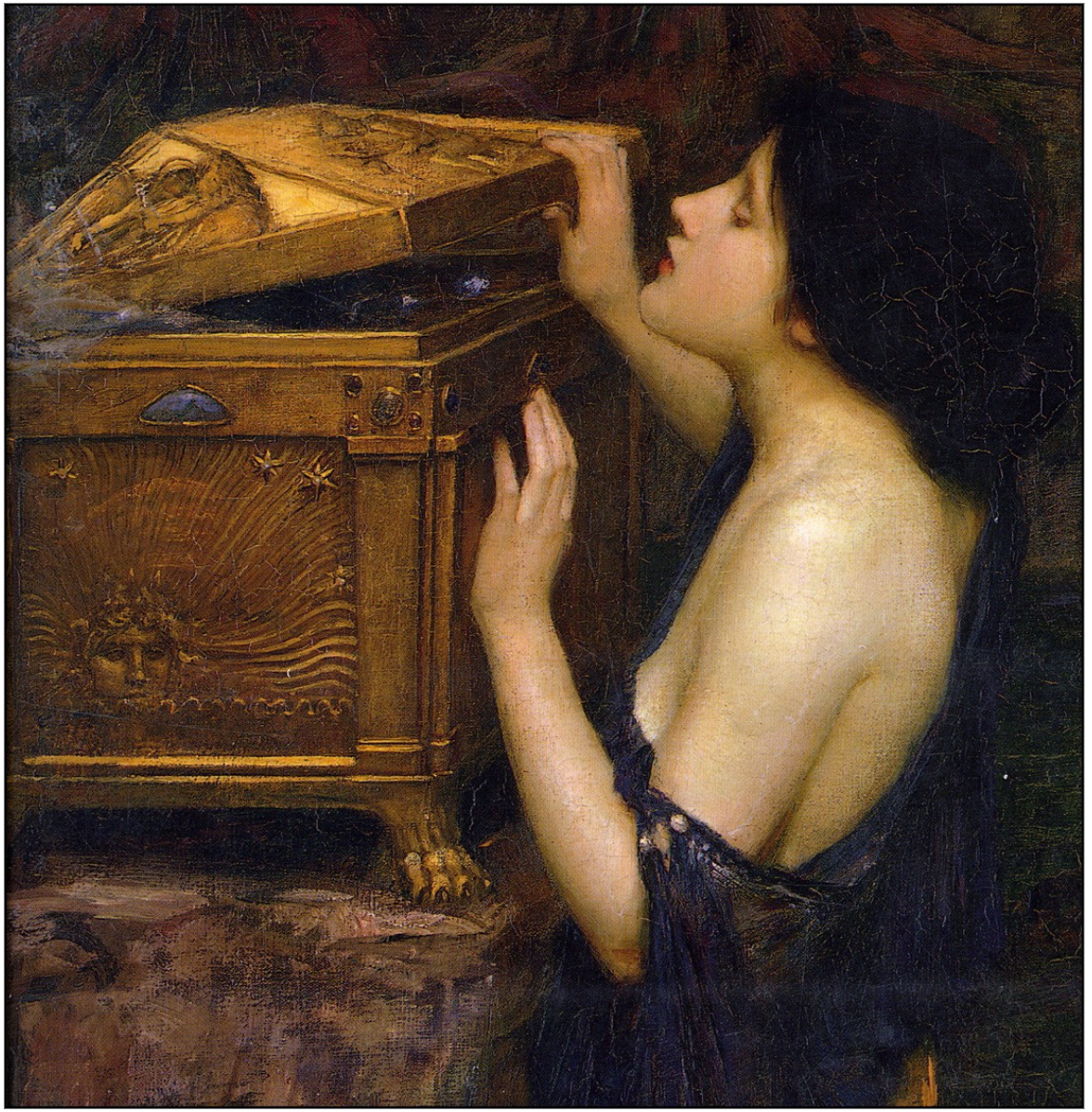
# Improving GIN



# 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 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
(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

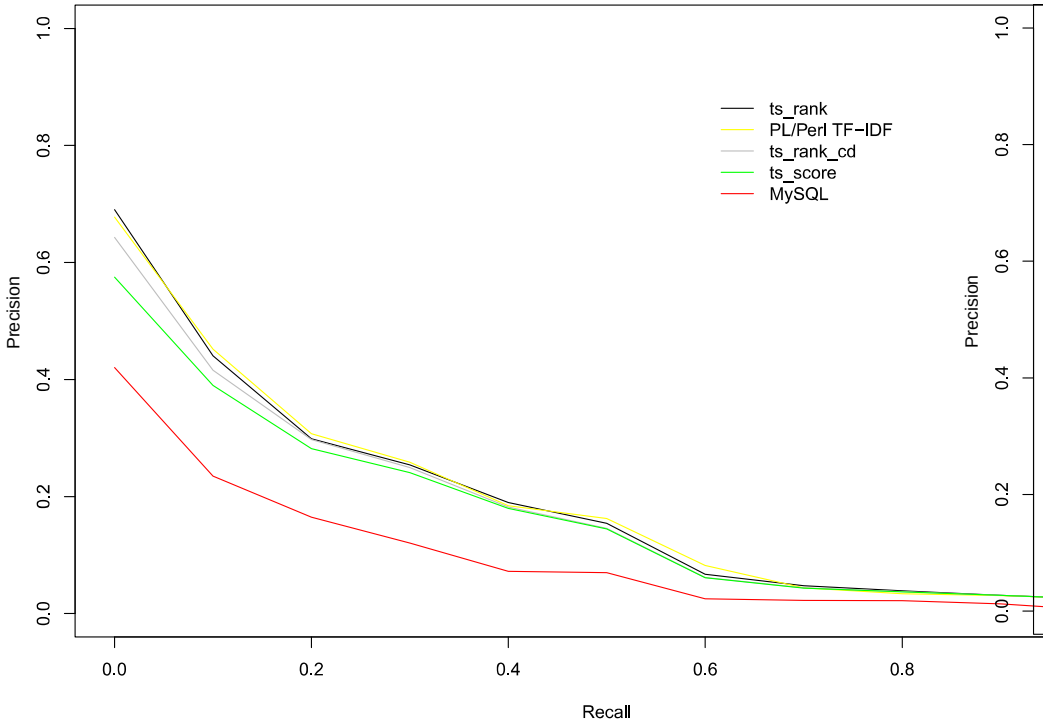
```
SELECT ts_rank(fts,plainto_tsquery('english', 'tom lane')) AS rank,
       ts_rank_cd (fts,plainto_tsquery('english', 'tom lane')) AS rank_cd ,
       fts <=> plainto_tsquery('english', 'tom lane') as score, subject
FROM pglist WHERE fts @@ plainto_tsquery('english', 'tom lane')
ORDER BY fts <=>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

# CREATE INDEX ... USING RUM

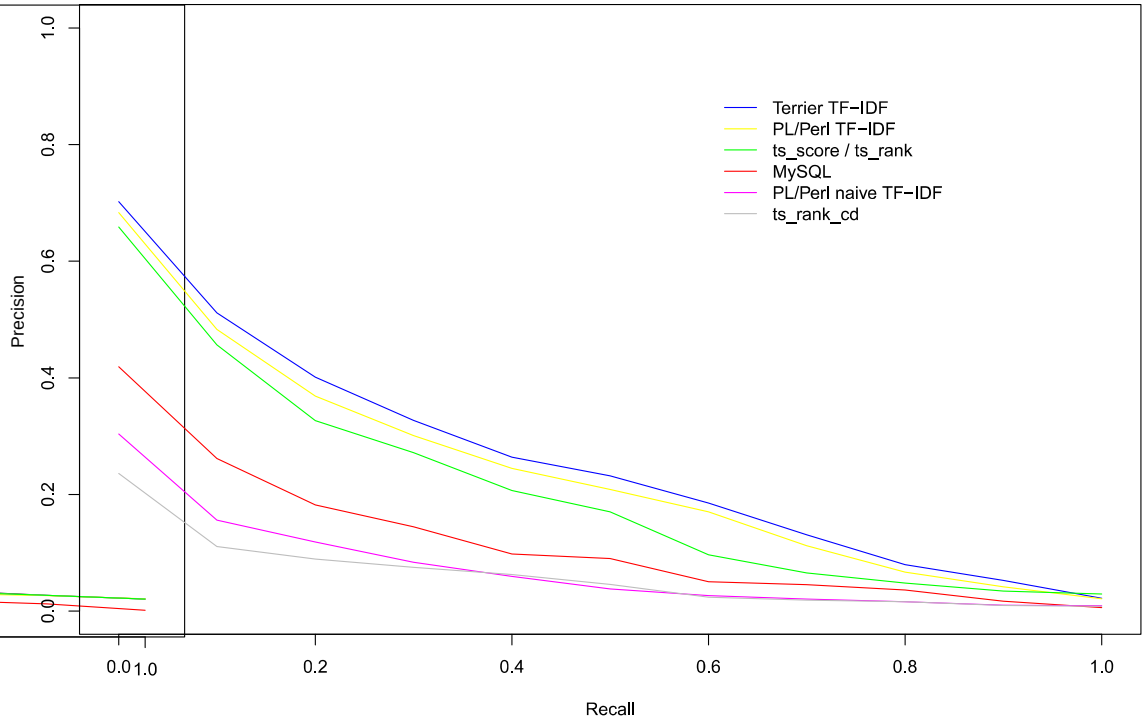
- RUM uses new ranking function (ts_score) — combination of ts_rank and ts_tank_cd

Precision-Recall (NIST TREC, AD-HOC coll.)

AND queries




OR queries



# 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

```
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;
      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 - 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)
```

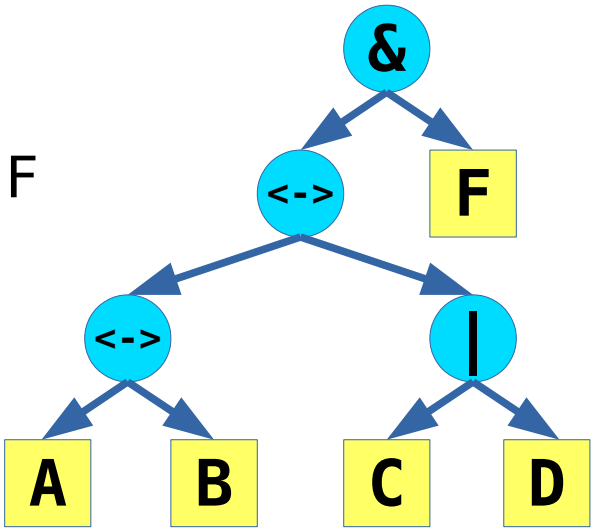
- `It's possible to combine tsquery's`

```
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

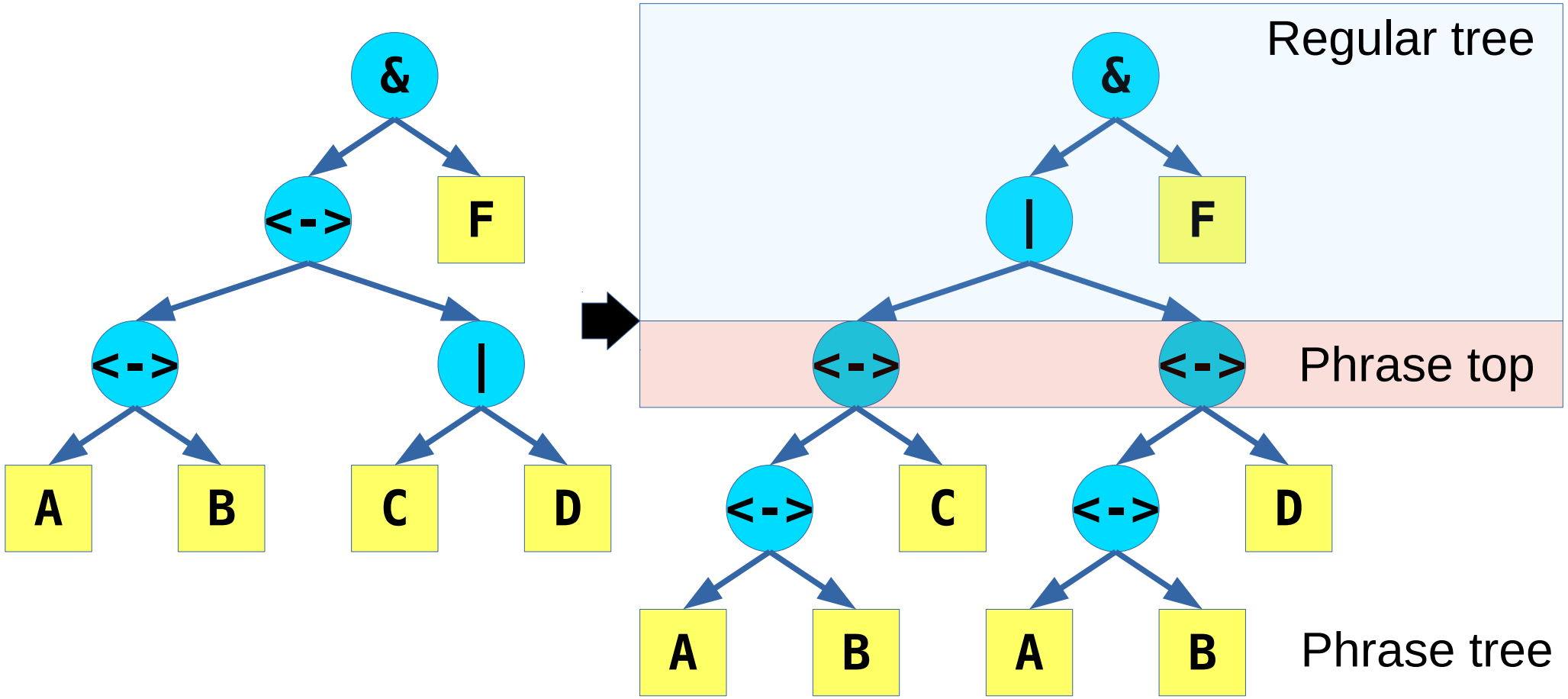
```
( (A <-> B) <-> (C | D) ) & F
```




- 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)
```

**Sequential Scan:** 2.6 s <-> vs 2.2 s &+regexp

```
select count(*) from pglist where fts @@ to_tsquery('english', 'tom <-> lane');
                                  QUERY PLAN
------------------------------------------------------------------------------
 Aggregate (actual time=2576.989..2576.989 rows=1 loops=1)
   ->  Seq Scan on pglist (actual time=0.310..2552.800 rows=222777 loops=1)
         Filter: (fts @@ '''tom'' <-> ''lane'''::tsquery)
         Rows Removed by Filter: 790993
 Planning time: 0.310 ms
 Execution time: 2577.019 ms
(6 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 &, considerable overhead

```
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,almost no overhead of phrase operator !

```
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)
```

# Some FTS problems: #3

- Combine FTS with ordering by timestamp[tz]
  - 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[tz]
  - 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 (subscription service)

```
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)
```

# 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))
```

# RUM vs GIN

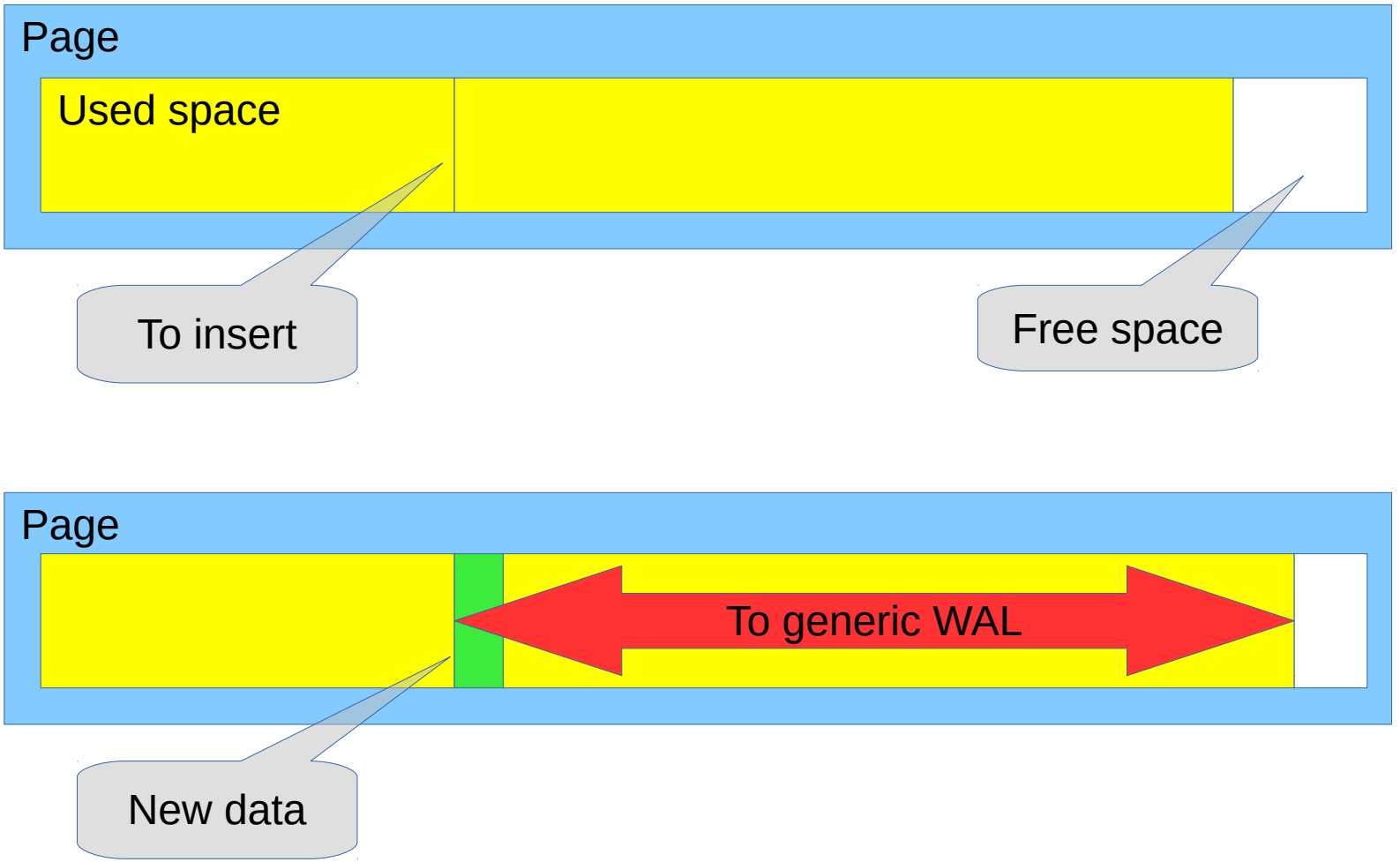
- 6 mln classifies, real fts quieries, concurrency 24, duration 1 hour
  - GIN — 258087
  - RUM — 1885698 ( 7x speedup )
- RUM has no pending list (not implemented) and stores more data.

  Insert 1 mln messages:

| 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


Thanks !

# 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 !

# 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
);
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
```

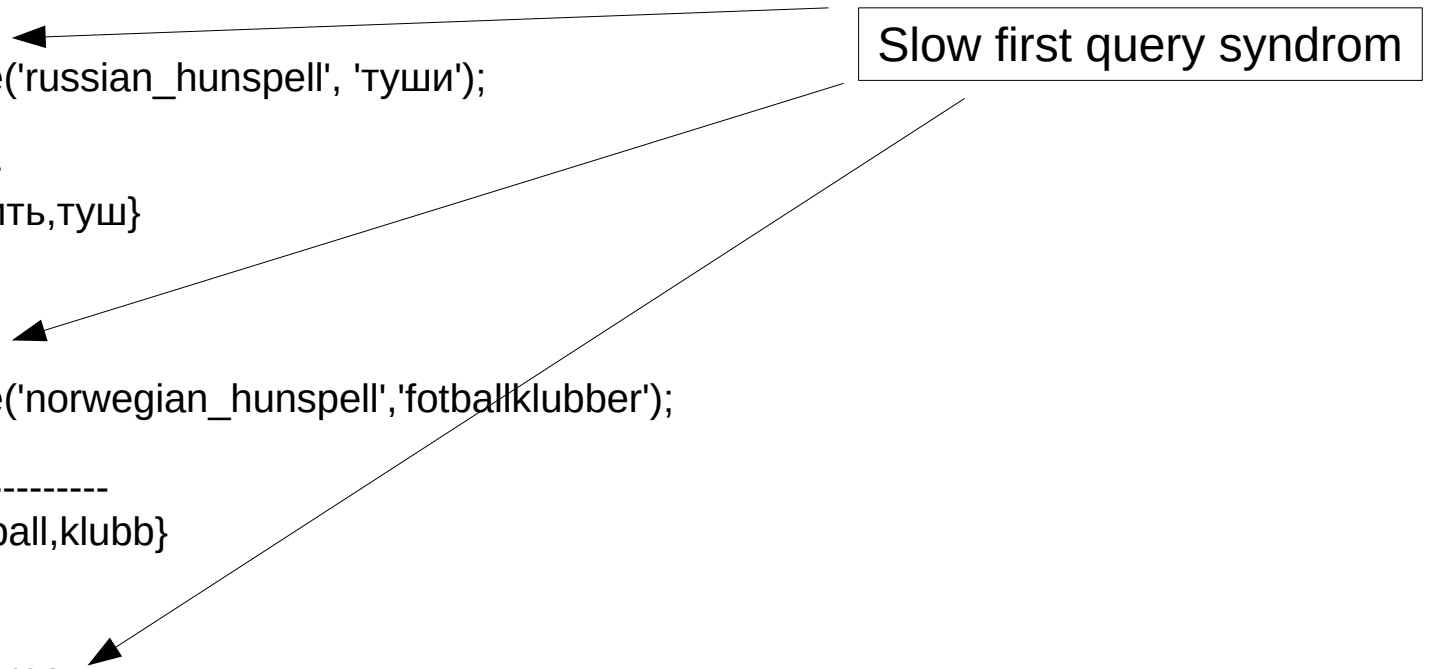
VS

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

# 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
```



# 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)
```

# Better FTS configurability

- The problem
  - Search multilingual collection requires processing by several language-specific  dictionaries. Currently, logic of processing is hidden from user and example would“nt works.

```
ALTER TEXT SEARCH CONFIGURATION multi_conf
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
    word, hword, hword_part
    WITH unaccent, german_ispell, english_ispell, simple;
```

- Logic of tokens processing in FTS configuration
  - Example: German-English collection

```
ALTER TEXT SEARCH CONFIGURATION multi_conf
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
    word, hword, hword_part
    WITH unaccent THEN (german_ispell AND english_ispell) OR simple;
```