# Full Text Search in PG12

Oleg Bartunov
Postgres Professional, Moscow University
obartunov@postgrespro.ru


Bengaluru, PGConf.in, Feb 27, 2020

# Postgres developer/contributor since 1995

PROFESSION: astronomer
PG: Extensibility, HSTORE, JSONB, FTS, Indexing, Community
LIKE: Running, trekking, foto, rock music





PROFESSIONAL
Postgres

# 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 the query
  - Return them sorted by relevance

# What is a document ?

- Arbitrary text attribute
- Combination of text attributes from the same or different tables (result of join[s])

```
msg (id, lid, subject, body);
lists (lid, list);

SELECT l.list || m.subject || m.body_plain as doc


Don’t forget about COALESCE (doc,’’)
```

# What is a query ?

- Arbitrary text

  ‘open source’
- Text with some query language

```
'postgresql "open source * database" -die +most'
```

# Why FTS in PostgreSQL ?

- 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

# Your system may looks like this

# FTS in PostgreSQL

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

# Text Search Operators

- Traditional text search operators
  ( TEXT op TEXT, op - ~, ~*, LIKE, ILIKE)

```
=# SELECT title FROM apod WHERE title ~* 'x-ray' limit 5;
                  title
-----------------------------------------
 The X-Ray Moon
 Vela Supernova Remnant in X-ray
 Tycho's Supernova Remnant in X-ray
 ASCA X-Ray Observatory
 Unexpected X-rays from Comet Hyakutake
(5 rows)

=# SELECT title FROM apod WHERE title ilike '%x-ray%' limit 5;
                 title
-----------------------------------------
 The Crab Nebula in X-Rays
 X-Ray Jet From Centaurus A
 The X-Ray Moon
 Vela Supernova Remnant in X-ray
 Tycho's Supernova Remnant in X-ray
(5 rows)
```

# Text Search Operators

- 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

- Built-in support of 22 languages (PG13)

```
[local]:6666 postgres@postgres=# \dF
               List of text search configurations
   Schema   |    Name    |              Description
------------+------------+---------------------------------------
 pg_catalog | arabic     | configuration for arabic language
 pg_catalog | danish     | configuration for danish language
 pg_catalog | dutch      | configuration for dutch language
 pg_catalog | english    | configuration for english language
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
 pg_catalog | greek      | configuration for greek language
 pg_catalog | hungarian  | configuration for hungarian language
 pg_catalog | indonesian | configuration for indonesian language
 pg_catalog | irish      | configuration for irish language
 pg_catalog | italian    | configuration for italian language
 pg_catalog | lithuanian | configuration for lithuanian language
 pg_catalog | nepali     | configuration for nepali language
 pg_catalog | norwegian  | configuration for norwegian language
 pg_catalog | portuguese | configuration for portuguese language
 pg_catalog | romanian   | configuration for romanian language
 pg_catalog | russian    | configuration for russian language
 pg_catalog | simple     | simple configuration
 pg_catalog | spanish    | configuration for spanish language
 pg_catalog | swedish    | configuration for swedish language
 pg_catalog | tamil      | configuration for tamil language
 pg_catalog | turkish    | configuration for turkish language
(23 rows)
```

# Basics of FTS

- **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

```
=# SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat':: tsquery;
 ?column?
----------
 t
(1 row)
```

# Basics of FTS

- tsvector and tsquery can be constructed
  - Inside database
  - Using external tools (perl, python,…)
- tsvector, tsquery and @@ - are building blocks for developing search engines ( text, molecular, mathematical formula, images).
- PostgreSQL provides infrastructure for building text search:
  - functions to make tsvector, tsquery
  - indexes to acceletate text search operator
  - functions for ranking results  and obtaining search snippets

# FTS in PostgreSQL

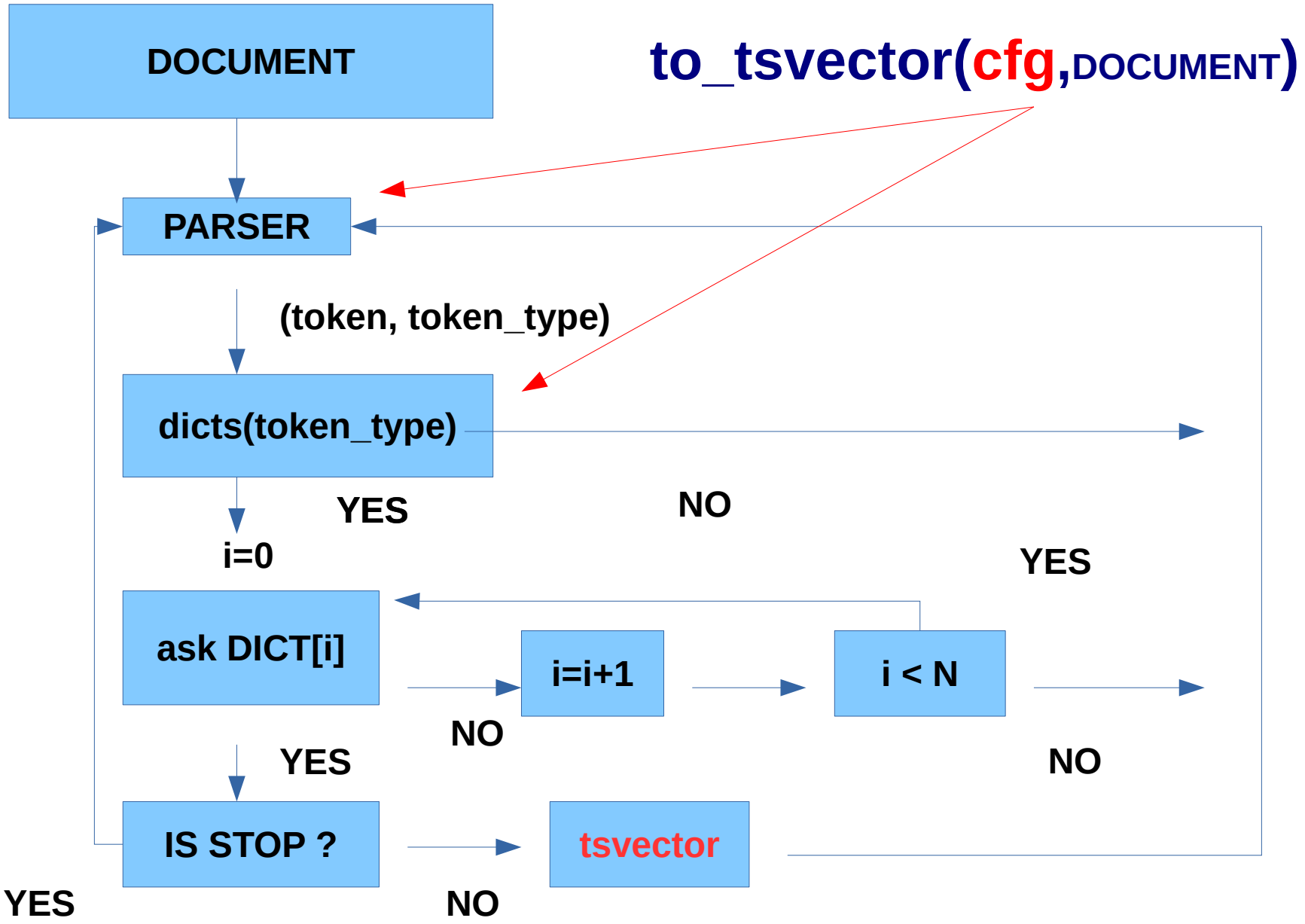
- FTS configuration defines parser and dictionaries used for document and query processing
  - Parser breaks text on to (token, type) pairs
  - Tokens converted to the lexems using dictionaries specific for token type

- Extendability:
  - Pluggable parser and dictionaries

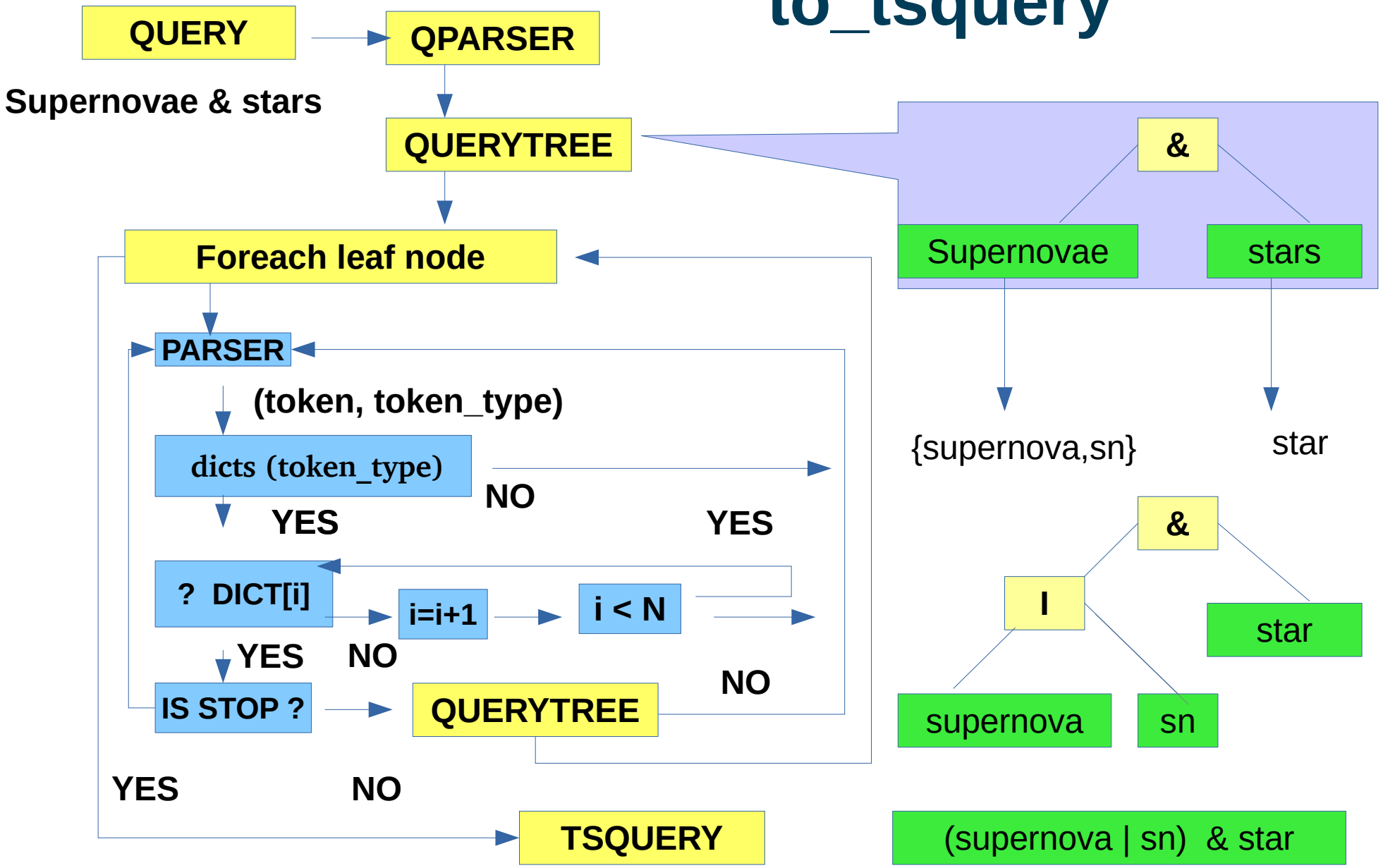
- \dF{,p,d}[+] [pattern] — psql FTS
- SQL interface:

```
{CREATE | ALTER | DROP} TEXT SEARCH {CONFIGURATION | DICTIONARY | PARSER}
```

# FTS PostgreSQL

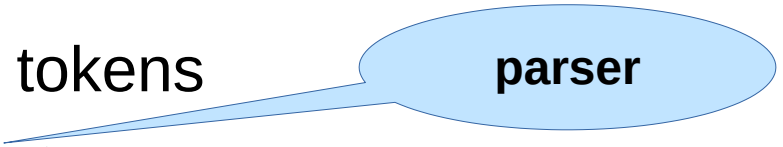


# FTS PostgreSQL
# to_tsquery



# FTS in PostgreSQL

- Parser breaks document into tokens

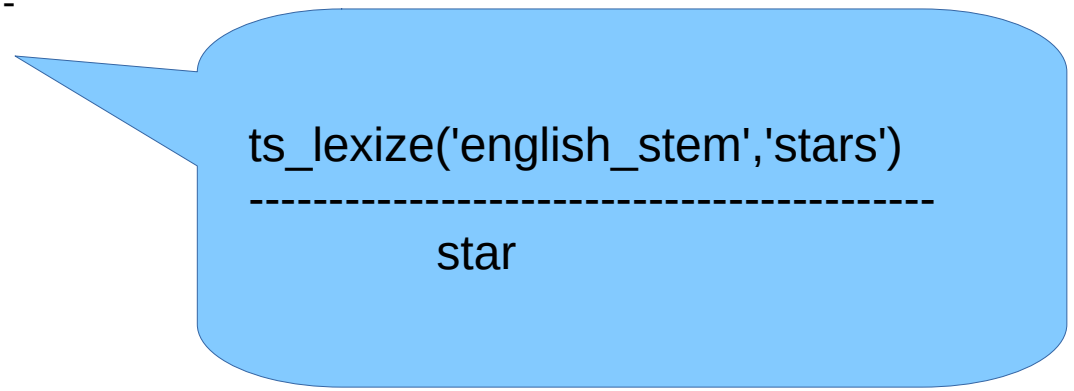



```
=# select * from ts_token_type('default');
 tokid |      alias      |                description
-------+-----------------+-------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
     5 | url             | URL
     6 | host            | Host
     7 | sfloat          | Scientific notation
     8 | version         | Version number
     9 | hword_numpart   | Hyphenated word part, letters and digits
    10 | hword_part      | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
    12 | blank           | Space symbols
    13 | tag             | XML tag
    14 | protocol        | Protocol head
    15 | numhword        | Hyphenated word, letters and digits
    16 | asciihword      | Hyphenated word, all ASCII
    17 | hword           | Hyphenated word, all letters
    18 | url_path        | URL path
    19 | file            | File or path name
    20 | float           | Decimal notation
    21 | int             | Signed integer
    22 | uint            | Unsigned integer
    23 | entity          | XML entity
(23 rows)
```

# FTS in PostgreSQL

- Each token processed by a set of dictionaries

```
=# \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

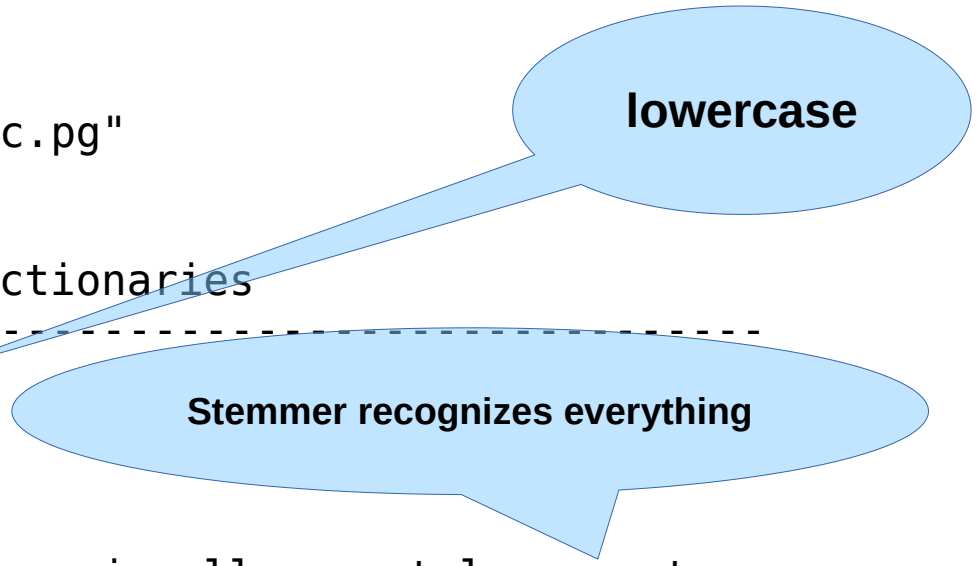


# FTS in PostgreSQL

- Token processed by dictionaries until it recognized
- It is discarded,  if it's not recognized

**Rule: from «specific» dictionary to a «common» dictionary**

```
=# \dF+ pg
                 Configuration "public.pg"
Parser name: "pg_catalog.default"
Locale: 'ru_RU.UTF-8' (default)
    Token     |                       Dictionaries
--------------+-------------------------------------------------------
 file         | pg_catalog.simple
 host         | pg_catalog.simple
 hword        | pg_catalog.simple
 int          | pg_catalog.simple
 lhword       | public.pg_dict,public.en_ispell,pg_catalog.en_stem
 lpart_hword  | public.pg_dict,public.en_ispell,pg_catalog.en_stem
 Lword        | public.pg_dict,public.en_ispell,pg_catalog.en_stem
 nlhword      | pg_catalog.simple
 nlpart_hword | pg_catalog.simple
```



# Dictionaries

- **Dictionary** – is a **program**, which accepts token on input and returns an array of lexems, NULL if token doesn‘t recognized and empty array for stop-word.
- ts_lexize(dictionary)

```
SELECT ts_lexize('english_hunspell','a') as stop,
ts_lexize('english_hunspell','elephants') AS elephants,
ts_lexize('english_hunspell','elephantus') AS unknown;
 stop |  elephants | unknown
------+------------+---------
 {}   | {elephant} | (null)
(1 row)
```

- Dictionary API allows to develop any custom dictionaries
  - Truncate too long numbers
  - Convert colors
  - Convert URLs to canonical way

    http://a.in/a/./index.html → http://a.in/a/index.html

# Dictionaries

- Examples of dictionaries

```
=# select ts_lexize('intdict', 11234567890);
  ts_lexize
-----------
 {112345}
=# select ts_lexize('roman', 'XIX');
 ts_lexize
------------
 {19}

=# select ts_lexize('colours','#FFFFFF');
ts_lexize
------------
{white}

SELECT ts_lexize('regex', 'ngc 1234');  – use pcre library
 ts_lexize
-----------                    (NGC|Abell|MKN|IC|H[DHR]|UGC|SAO|MWC)(\s|-)?((\d){1,6}[ABC]?) $1$3
 {ngc1234}
(1 row)
```

# Built-in Dictionaries

Dictionary templates:

1. Simple
   - convert the input token to lower case
   - exclude stop words
2. Synonym (also, contrib/xsyn)
   - replace word with a synonym

   Example of .syn file:

```
postgres    pgsql
postgresql pgsql
postgre     pgsql
```

# Built-in Dictionaries

3. Thesaurus
   - replace phrase by indexed phrase

   Example of .ths file:

```
booking tickets : order invitation cards
booking ? tickets : order invitation Cards
```

4. Snowball stemmer
   - reduce words by stemming algorithms
   - recognizes everything
   - exclude stop words

```
SELECT ts_lexize('portuguese_stem','responsáveis');
 ts_lexize
-----------
 {respons}
(1 row)
```

# Built-in Dictionaries

5. Ispell
   - normalize different linguistic forms of a word into the same lexeme. Try to reduce an input word to its infinitive form
   - support dictionary file formats: Ispell, MySpell, Hunspell
   - exclude stop words

```
      viva          |      vivo       |   viver
--------------------+-----------------+----------
 {viva,vivo,viver} | {vivo,viver} | {viver}
```

# Filter dictionary – unaccent

contrib/unaccent - unaccent text search dictionary and function to remove accents. It's filtering dictionary — processed lexeme will be passed to the next dictionary if any.

```
1. Unaccent dictionary does nothing and returns NULL

=# select ts_lexize('unaccent','Hotels') is NULL;
 ?column?
----------
 t

2. Unaccent dictionary removes accent and returns 'Hotel'.

=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
```

# Ispell shared dictionaries

- 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

real   0m0.656s
user   0m0.015s
sys0m0.031s

For russian hunspell dictionary:

real   0m3.809s
user   0m0.015s
sys0m0.029s

Each session «eats» 20MB !
```

# Dictionaries as extensions

- Easy installation of hunspell dictionaries (extension)

```
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

- Dictionaries are loaded into memory for every session (slow first query symptom) and eat memory.
  Solution: Use **shared_ispell** extension.

```
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
```

```
real 0m0.170s          real 0m3.809s
user0m0.015s    VS     user0m0.015s
sys  0m0.027s          sys  0m0.029s
```

# Tsvector functions

- Document to tsvector:
  - to_tsvector([cfg], text|json|jsonb)
    cfg — FTS configuration, default_text_search_config — GUC

```
SELECT to_tsvector('It is a very long story about true and false');
                to_tsvector
-------------------------------------
 'fals':10 'long':5 'stori':6 'true':8
(1 row)
```

Lowercase, no stop words

```
SELECT to_tsvector('simple', 'It is a very long story about true and false');
                                    to_tsvector
-------------------------------------------------------------------------------------
 'a':3 'about':7 'and':9 'false':10 'is':2 'it':1 'long':5 'story':6 'true':8 'very':4
(1 row)
```

# Query functions

- Query to tsquery:
  - to_tsquery([cfg], text)

    .

    Better, always specify *cfg* (immutable vs stable) !

```
select to_tsquery('supernovae & stars');
      to_tsquery
-----------------------
 'supernova' & 'star'
(1 row)
```

  - plainto_tsquery([cfg],text) – words are AND-ed

```
select plainto_tsquery('supernovae  stars');
   plainto_tsquery
-----------------------
 'supernova' & 'star'
(1 row)
```

# Query functions

- 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)
```

Stop words are taken into account !

- 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)
```

# Query functions

- websearch_to_tsquery([cfg], text)
  - Recognizes “phrases”, AND, OR, *, +word, -word

```
select websearch_to_tsquery('english','postgresql "open source *
database" -die +most');
                    websearch_to_tsquery
-------------------------------------------------------------
 'postgresql' & 'open' <-> 'sourc' <2> 'databas' & !'die'
(1 row)


select to_tsvector('english', 'PostgreSQL: The Worlds Most Advanced
Open Source Relational Database') @@
websearch_to_tsquery('english','postgresql "open source * database" -
die +most');
 ?column?
----------
 t
(1 row)
```

# FTS: additional functions

- setweight(tsvector, "char", text[] - add label "char"  to lexemes from text[]
- ts_delete(tsvector, text[]) - delete lexemes with specific label{s} from tsvector
- 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) – convert tsvector to text array
- array_to_tsvector(text[]) - convert text array to tsvector
- ts_filter(tsvector,text[]) - fetch lexemes with specific label{s}

# FTS: additional functions

- ts_debug(cfg, text) – good for debugging FTS configuration
- ts_stat – word frequencies
- ts_parse(parser, text) – produces (token_type, token) from a text
- ts_rewrite – rewrite query online, no reindexing needed
- ts_headline – pieces of documents with words from query
- Ordering result of FTS:
  - ts_rank – the more occurrences of words,the bigger rank
    good for OR queries, no query language
  - ts_rank_cd – the closer words, the bigger rank
    good for AND queries, supports query language
  - rum_ts_score (requires RUM extension) – combination of the above, the best (NIST TREC, AD-HOC coll.)

# FTS indexes

- CREATE INDEX … USING GIST/GIN/RUM (tsvector)
- GiST
  - document, query as a signature, documents → signature tree, Bloom filter used for search
  - Fast insert, small size, good for small collections
- GIN — inverted tree, basically it“s a B-tree
  - Optimized for storing a lot of duplicate keys
  - Duplicates are ordered by heap TID
  - Not as fast as GiST for updates, good performance and scalability
- RUM (extension) — GIN++
  - GIN with addtional information  (words positions, timestamp, ...)
  - Big size, best for mostly read-only workload, very fast for ranking, good for phrase search, no need tsvector column

# FTS summary

- FTS in PostgreSQL is a flexible search engine,

- It is a «collection of bricks» you can build your search engine using
  - Custom parser
  - Custom dictionaries

  + All power of SQL (FTS+Spatial+Temporal)

# Example:Search Mailing list archive

- https://postgrespro.com/list
- Custom parser — fixes several problems in default parser

```
select * from ts_parse('default','1914-1999');
 tokid | token
-------+-------
    22 | 1914
    21 | -1999
(2 rows)
```

```
select * from ts_parse('tsparser','1914-1999');
 tokid |   token
-------+-----------
    15 | 1914-1999
     9 | 1914
    12 | -
     9 | 1999
(4 rows)
```

```
select * from ts_parse('default','pg_catalog');
 tokid |  token
-------+---------
     1 | pg
    12 | _
     1 | catalog
(3 rows)
```

```
select * from ts_parse('tsparser','pg_catalog');
 tokid |   token
-------+------------
    16 | pg_catalog
    11 | pg
    12 | _
    11 | catalog
(4 rows)
```

# Search Mailing list archive

- https://postgrespro.com/list
- Custom parser — fixes several problems in default parser

```
\dF+ ppgweb_fts
                        Text search configuration "public.ppgweb_fts"
Parser: "public.tsparser"
      Token      |                                    Dictionaries
-----------------+-------------------------------------------------------------------------------
 asciihword      | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
 asciiword       | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
 hword_asciipart | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
 hword_numpart   | simple
 hword_part      | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | ppg_thesaurus,russian_synonym,english_synonym,dic_shared_ru,dic_shared_en,russian_stem
```

# Search Mailing list archive

- https://postgrespro.com/list
- Faceted search -  grouping search results by lists
- Strip citation from posts
- Uses pg_trgm for suggestions
- Advanced query language
  - Support «phrase» search

PROFESSIONAL Postgres

# Search Mailing list archive


server cras | Mailing lists | Search

- **server crash**
- **server crash**es
- **server crash**ing
- **server crash**ed
- **server crash**me

## server crash - Search results in mailing lists

server crash

List: All lists

Post date: anytime

Sort by: Date

Search

### pgsql-general (1037)

2018-10-16 21:25:54 | postgres **server** process **crashes** when using odbc_fdw (Ravi Krishna)

**server**. I also created foreign table. When I run a sql 'select * from odbctest' postgres **crashes**

Thread >> Search in thread (12)

2018-09-26 14:46:10 | Re: Setting up continuous archiving (Stephen Frost)

**server crashes** or there's some kind of issue with it after the rsync finishes

Thread

2018-08-29 04:02:45 | WAL replay issue from 9.6.8 to 9.6.10 (Dave Peticolas)

**server** to 9.6.8 and I was able to replay WAL past the point where 9.6.10 would PANIC and **crash**

Thread

2018-08-24 19:07:41 | Re: unorthodox use of PG for a customer (David Gauthier)

**crash** them. Of course any DB running would die too and have to be restarted/recovered. So the place for the DB is really elsewhere, on an external **server**

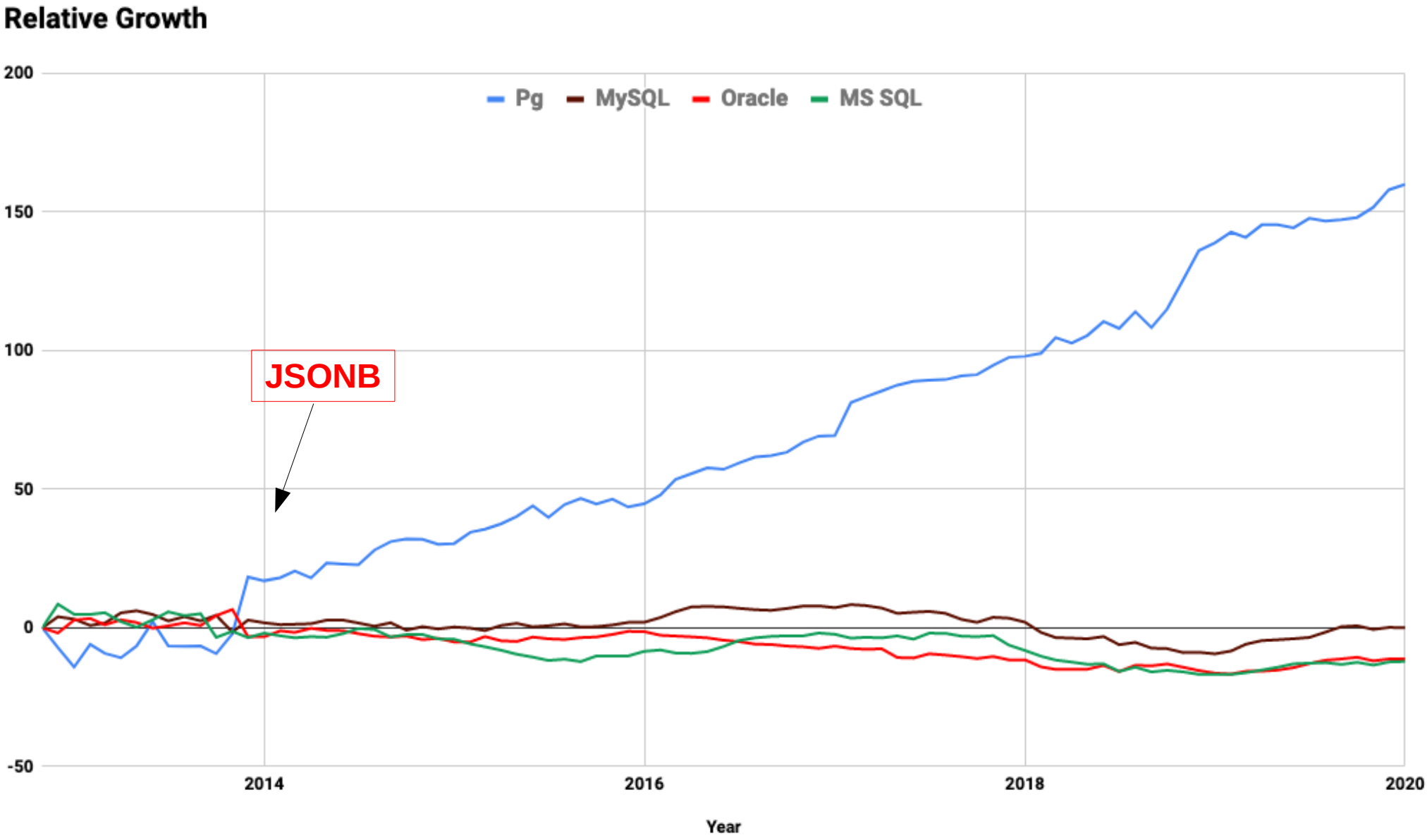
Thread

### pgsql-hackers (1199)

2018-10-23 21:06:49 | Re: [HACKERS] Transactions involving multiple postgres foreignservers, take 2 (Masahiko Sawada)

PROFESSIONAL Postgres

# JSONB helps Postgres popularity



# FTS for JSONB

- JSON[b] to tsvector:
  - Results are different for json and jsonb !
    (jsonb: keys are sorted, json: spaces are preserved)
  - Phrases are preserved

```
SELECT to_tsvector('simple', jb) AS tsvector_json[b] FROM (values ('
{
    "abstract": "It is a very long story about true and false",
    "title": "Peace and War",
    "publisher": "Moscow International house"
 }
'::json[b])) foo(jb);
                                    tsvector_json
-------------------------------------------------------------------------------------
 'a':3 'about':7 'and':9,13 'false':10 'house':18 'international':17 'is':2 'it':1
'long':5 'moscow':16 'peace':12 'story':6 'true':8 'very':4 'war':14

                                    tsvector_jsonb
-------------------------------------------------------------------------------------
 'a':7 'about':11 'and':2,13 'false':14 'house':18 'international':17 'is':6 'it':5
'long':9 'moscow':16 'peace':1 'story':10 'true':12 'very':8 'war':3
```

# FTS for JSONB

- Phrase search is supported

```
SELECT  phraseto_tsquery('simple','war moscow') @@ to_tsvector('simple',jb) FROM (values ('
{
    "abstract": "It is a very long story about true and false",
    "title": "Peace and War",
    "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 f
SELECT  phraseto_tsquery('simple','moscow international') @@ to_tsvector('simple',jb)
FROM(values ('
{
    "abstract": "It is a very long story about true and false",
    "title": "Peace and War",
    "publisher": "Moscow International house"
 }
'::jsonb)) foo(jb);
 ?column?
----------
 t
```

# FTS for JSONB

- json[b]_to_tsvector([cfg],json[b],json[b])
  - to_tsvector(json[b]) converts only string values and ignores **key, numeric, boolean**

```
SELECT to_tsvector(jsonb '{"aa":"3 4","b":[5,6], "c":"elephants", "bool": true}');
      to_tsvector
----------------------
 '3':3 '4':4 'eleph':1
SELECT jsonb_to_tsvector(jsonb '{"aa":"3 4","b":[5,6], "c":"elephants", "bool": true}',
                                        '["string","numeric"]');
         jsonb_to_tsvector
---------------------------------
 '3':7 '4':8 '5':1 '6':3 'eleph':5
SELECT jsonb_to_tsvector(jsonb '{"aa":"3 4","b":[5,6], "c":"elephants", "bool": true}',
                                        '["boolean"]');
 jsonb_to_tsvector
-----------------------
 'true':1
select jsonb_to_tsvector(jsonb '{"aa":"3 4","b":[5,6], "c":"elephants", "bool": true}','["all"]');
                              jsonb_to_tsvector
-------------------------------------------------------------------------------
 '3':13 '4':14 '5':3 '6':5 'aa':11 'b':1 'bool':16 'c':7 'eleph':9 'true':18
```

# FTS for JSONB

- PG12 adds **jsonpath** (part of SQL/JSON), which describes a <projection> of JSON data.

**PG11**

```
SELECT apt
FROM (SELECT jsonb_array_elements(
                    jsonb_array_elements(js->'floor')->'apt')
      FROM house) apts(apt)
WHERE (apt->>'area')::int > 40 AND (apt->>'area')::int < 90;
```

**PG12**

```
SELECT jsonb_path_query(js,'$.floor[*].apt[*] ?
                           (@.area > 40 && @.area < 90)')
FROM house;
```

# FTS for JSONB

- PG12 adds **jsonpath** (part of SQL/JSON), which describes a <projection> of JSON data.
- IMDB database example

```
SELECT id, to_tsvector('english',
jsonb_path_query_array(jb, '$.companies[*].name'))
FROM imdb limit 10;

SELECT id, to_tsvector('english', jsonb_build_array(
  jb->'title',
  jb->'trivia',
  jsonb_path_query_array(jb, '$.companies[*].name')
)) FROM imdb limit 10;

Syntax extension for PG13?

SELECT id, to_tsvector('english', jsonb_path_query
(jb, '[$.title, $.trivia, $.companies[*].name]')
) FROM imdb limit 10;
```

# Generated column instead of trigger

- PG12 adds support of generated columns.
  This is an SQL-standard feature that allows creating columns that are computed from expressions rather than assigned, similar to a view or materialized view but on a column basis.
- FTS can used them instead of trigger to populate tsvector column

```
CREATE TABLE tt( id serial primary key, doc jsonb,
fts tsvector  GENERATED ALWAYS AS (
                    to_tsvector('english',
                        coalesce((doc ->> 'name'),'')  || ' '||
                        coalesce((doc ->> 'title'),'')
                    ) ) STORED
);
INSERT INTO tt(doc) VALUES('{"name":"movies","title":"Die, Oracle, die !"}');
INSERT INTO tt(doc) VALUES('{"name":"movies","title":"Postgres rulez !"}');
SELECT id, fts FROM tt;
 id |              fts
----+-------------------------------
  1 | 'die':2,4 'movi':1 'oracl':3
  2 | 'movi':1 'postgr':2 'rulez':3
(2 rows)
```

# PG13 for FTS

- 7266d0997dd2a0632da38a594c78e25ff21df67e
  Allow functions-in-FROM to be pulled up if they reduce to constants.
  Author: Alexander Kuzmenkov and Aleksandr Parfenov

- Query 2 is more concise
  PG12: Query 2 is slower, so query 1 is used
  PG13: Queries 1 and 2 are equivalent

**1**

```
SELECT ts_headline(body,to_tsquery('english', 'supernovae | nebulae')),
ts_rank(fts,to_tsquery('english', 'supernovae | nebulae')) AS rank
FROM apod
WHERE fts  @@ to_tsquery('english', 'supernovae | nebulae')
ORDER BY rank DESC LIMIT 1;
```

Function-in-FROM should be immutable !

**2**

```
SELECT ts_headline(body,q),ts_rank(fts,q) AS rank
FROM apod, to_tsquery('english', 'supernovae | nebulae') q
WHERE fts  @@ q
ORDER BY rank DESC LIMIT 1;
```

# PG13 for FTS

## Query 2

**PG12**

```
Limit (actual time=1.455..1.455 rows=1 loops=1)
  ->  Result (actual time=1.454..1.454 rows=1 loops=1)
        ->  Sort (actual time=1.305..1.305 rows=1 loops=1)
              Sort Key: (ts_rank(apod.fts, q.q)) DESC
              Sort Method: top-N heapsort  Memory: 26kB
              ->  Nested Loop (actual time=0.117..1.135 rows=378 loops=1)
                    ->  Function Scan on q (actual time=0.002..0.004 rows=1 loops=1)
                    ->  Bitmap Heap Scan on apod (actual time=0.109..0.382 rows=378 loops=1)
                          Recheck Cond: (fts @@ q.q)
                          Heap Blocks: exact=241
                          ->  Bitmap Index Scan on apod_fts_idx (actual time=0.082..0.082 rows=378 loops=
                                Index Cond: (fts @@ q.q)
```

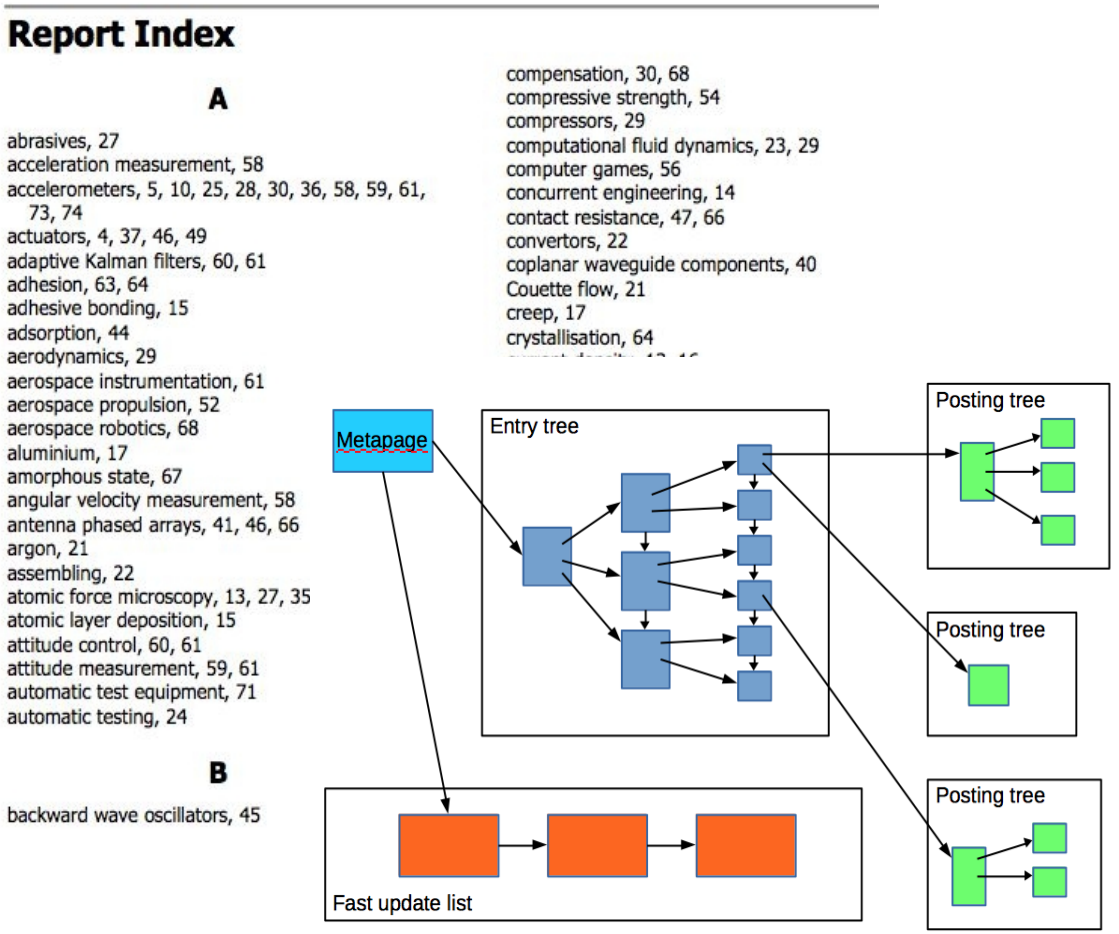
**PG13**

```
Limit (actual time=0.998..0.998 rows=1 loops=1)
  ->  Result (actual time=0.997..0.998 rows=1 loops=1)
        ->  Sort (actual time=0.861..0.862 rows=1 loops=1)
              Sort Key: (ts_rank(apod.fts, '"supernova" | "nebula"'::tsquery)) DESC
              Sort Method: top-N heapsort  Memory: 26kB
              ->  Bitmap Heap Scan on apod (actual time=0.096..0.753 rows=378 loops=1)
                    Recheck Cond: (fts @@ '"supernova" | "nebula"'::tsquery)
                    Heap Blocks: exact=241
                    ->  Bitmap Index Scan on apod_fts_idx (actual time=0.067..0.067 rows=378 loops=1)
                          Index Cond: (fts @@ '"supernova" | "nebula"'::tsquery)
```

# PG13 for FTS

- 4b754d6c16e16cc1a1adf12ab0f48603069a0efd
  Avoid full scan of GIN indexes when possible
  Author: Nikita Glukhov, Alexander Korotkov, Tom Lane, Julien Rouhaud

ENTRY TREE



# PG13 for FTS

- 4b754d6c16e16cc1a1adf12ab0f48603069a0efd
  Avoid full scan of GIN indexes when possible
  Author: Nikita Glukhov, Alexander Korotkov, Tom Lane, Julien Rouhaud
- **PG12** Queries like 'supernovae' AND !star' are very slow, since costs for scanning full GIN index for '!nebulae' is very high
- **PG13** GIN understand that searching for '!star' is useless and avoid full index scan for queries like above.

```
SELECT 1
FROM apod
WHERE fts  @@ to_tsquery('english', 'supernovae')
      AND fts  @@ to_tsquery('english', '!star');
```

# PG13 for FTS

- 4b754d6c16e16cc1a1adf12ab0f48603069a0efd
  Avoid full scan of GIN indexes when possible
  Author: Nikita Glukhov, Alexander Korotkov, Tom Lane, Julien Rouhaud

**PG12**

```
Bitmap Heap Scan on apod (actual time=3.405..3.419 rows=20 loops=1)
  Recheck Cond: ((fts @@ '"supernova"'::tsquery) AND (fts @@ '!"star"'::tsquery))
  Heap Blocks: exact=20
  Buffers: shared hit=138
  ->  Bitmap Index Scan on apod_fts_idx (actual time=3.400..3.400 rows=20 loops=1)
        Index Cond: ((fts @@ '"supernova"'::tsquery) AND (fts @@ '!"star"'::tsquery))
        Buffers: shared hit=118
Planning Time: 0.162 ms
Execution Time: 3.435 ms
```

**PG13**

```
Bitmap Heap Scan on apod (actual time=0.042..0.059 rows=20 loops=1)
  Recheck Cond: ((fts @@ '"supernova"'::tsquery) AND (fts @@ '!"star"'::tsquery))
  Heap Blocks: exact=20
  Buffers: shared hit=26
  ->  Bitmap Index Scan on apod_fts_idx (actual time=0.035..0.035 rows=20 loops=1)
        Index Cond: ((fts @@ '"supernova"'::tsquery) AND (fts @@ '!"star"'::tsquery))
        Buffers: shared hit=6
Planning Time: 0.205 ms
Execution Time: 0.075 ms
```

# References

- Slides of this talk
  http://www.sai.msu.su/~megera/postgres/talks/fts-pgconf.in-2020.pdf
- FTS talk at PGConf.EU 2018
  http://www.sai.msu.su/~megera/postgres/talks/pgconf.eu-2018-fts.pdf
- Dictionaries as extensions
  https://github.com/postgrespro/hunspell_dicts
- Improved text search parser
  https://github.com/postgrespro/pg_tsparser
- RUM index
  https://github.com/postgrespro/rum
- Shared ispell template
  https://github.com/postgrespro/shared_ispell
- Full text search example
  https://github.com/postgrespro/apod_fts
- Setrank - TF*IDF ranking
  https://github.com/obartunov/setrank
- Jsonpath in PostgreSQL
  http://www.sai.msu.su/~megera/postgres/talks/jsonpath-pgconfeu-2019.pdf

# References

- Dictionary for regular expressions
  https://github.com/obartunov/dict_regex
- Dictionary for roman numbers
  https://github.com/obartunov/dict_roman
- Faceted search in one query
  http://akorotkov.github.io/blog/2016/06/17/faceted-search/
- FTS real example: Search mailing list archives
  https://postgrespro.com/list
- FTS slides with a lot of info
  http://www.sai.msu.su/~megera/postgres/talks/fts_postgres_by_authors_2.pdf
- Pg_trgm documentation
  https://www.postgresql.org/docs/11/static/pgtrgm.html
- My postings about FTS
  https://obartunov.livejournal.com/tag/fts

# FTS in PostgreSQL

- Built-in support of 22 languages (PG13)

```
[local]:6666 postgres@postgres=# \dF
               List of text search configurations
   Schema   |    Name    |              Description
------------+------------+---------------------------------------
 pg_catalog | arabic     | configuration for arabic language
 pg_catalog | danish     | configuration for danish language
 pg_catalog | dutch      | configuration for dutch language
 pg_catalog | english    | configuration for english language
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
 pg_catalog | greek      | configuration for greek language
 pg_catalog | hungarian  | configuration for hungarian language
 pg_catalog | indonesian | configuration for indonesian language
 pg_catalog | irish      | configuration for irish language
 pg_catalog | italian    | configuration for italian language
 pg_catalog | lithuanian | configuration for lithuanian language
 pg_catalog | nepali     | configuration for nepali language
 pg_catalog | norwegian  | configuration for norwegian language
 pg_catalog | portuguese | configuration for portuguese language
 pg_catalog | romanian   | configuration for romanian language
 pg_catalog | russian    | configuration for russian language
 pg_catalog | simple     | simple configuration
 pg_catalog | spanish    | configuration for spanish language
 pg_catalog | swedish    | configuration for swedish language
 pg_catalog | tamil      | configuration for tamil language
 pg_catalog | turkish    | configuration for turkish language
(23 rows)
```

# FTS in PostgreSQL

- Built-in support of 22 languages
- No Hindi !
  - Actually, just one day of work, mostly copy-paste
  - Let's do for PG13 !
  - Hindi stemming algorithm exists
    Snowball implementation of
    "A. Ramanathan and D. Rao (2003) A Lightweight Stemmer for Hindi "

    https://snowballstem.org/algorithms/hindi/stemmer.html
    https://github.com/snowballstem/snowball
  - How nepali languages was added to PG12
    https://obartunov.livejournal.com/198032.html

# FTS in PostgreSQL

- Built-in support of 22 languages
- No Hindi !
  - Actually, just one day of work, mostly copy-paste
  - Let's do for PG13 !
  - Quick test on smartphone (termux):

```
u0_a236=# create table tt (a text);
CREATE TABLE
u0_a236=# insert into tt values('अभिनव');
INSERT 0 1
u0_a236=# insert into tt values('अभिनव अच्छा लड़का है।');
INSERT 0 1
u0_a236=# select * from tt;
                  a
--------------------
 अभिनव
 अभिनव अच्छा लड़का है।
(2 rows)

u0_a236=# select to_tsvector(a) from tt;
                to_tsvector
-----------------------------------
 'अभिनव':1
 'अच्छा':2 'अभिनव':1 'लड़का':3 'है':4
(2 rows)
```

```
u0_a236=# select to_tsvector(a) @@ to_tsquery('लड़का') from
tt;
 ?column?
----------
 f
 t
(2 rows)

u0_a236=# select to_tsvector(a) @@ to_tsquery('लड़के') from t
t;
 ?column?
----------
 f
 f
(2 rows)

u0_a236=#
```

Plural

Single

Stemmer will fix this

# FTS in PostgreSQL

- Built-in support of 22 languages
- No Kannada !
  - Need some work
  - Let's do for PG14 !
  - Kannada stemming algorithm exists
    https://github.com/Sahana-M/shabdkosh
  - Kannada hunspell dictionary
    https://github.com/sanchaya/kn-hunspell


# ध्यान के लिए धन्यवाद !


https://pgconf.org.np Submit your talk !