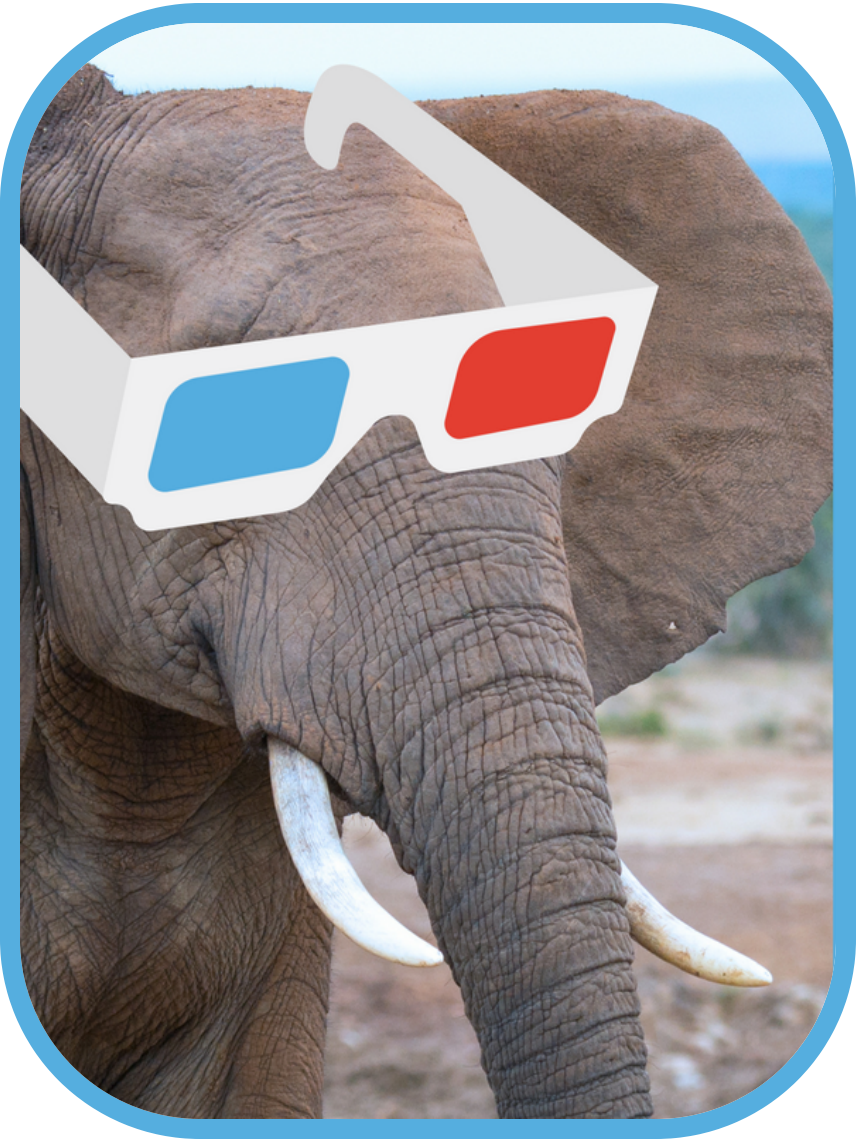
2ndQuadrant®
PostgreSQL



PostgreSQL is NOT your traditional SQL database

Gülçin Yıldırım Jelínek

PGDay Istanbul 2018

select * from me;

Board of Directors @ PostgreSQL Europe

Cloud Services Manager @ 2ndQuadrant

Main Organizer @ Prague PostgreSQL Meetup

Member @ Postgres Women

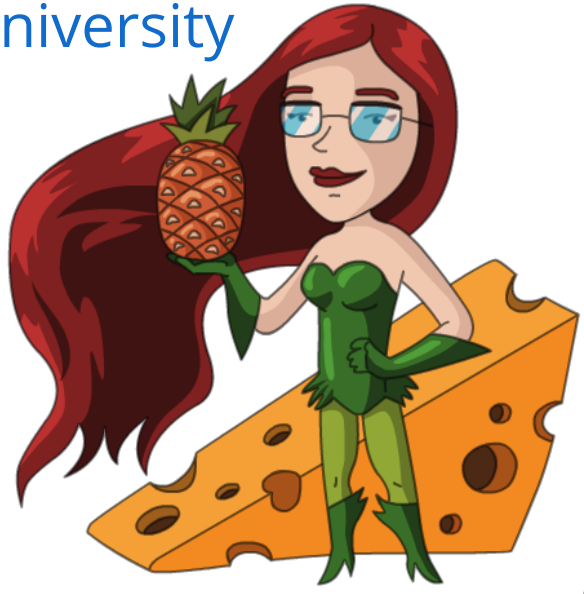
MSc, Computer & Systems Eng. @ TalTech

BSc, Applied Mathematics @ Yildiz Technical University

Writes on 2ndQuadrant blog

From Turkey

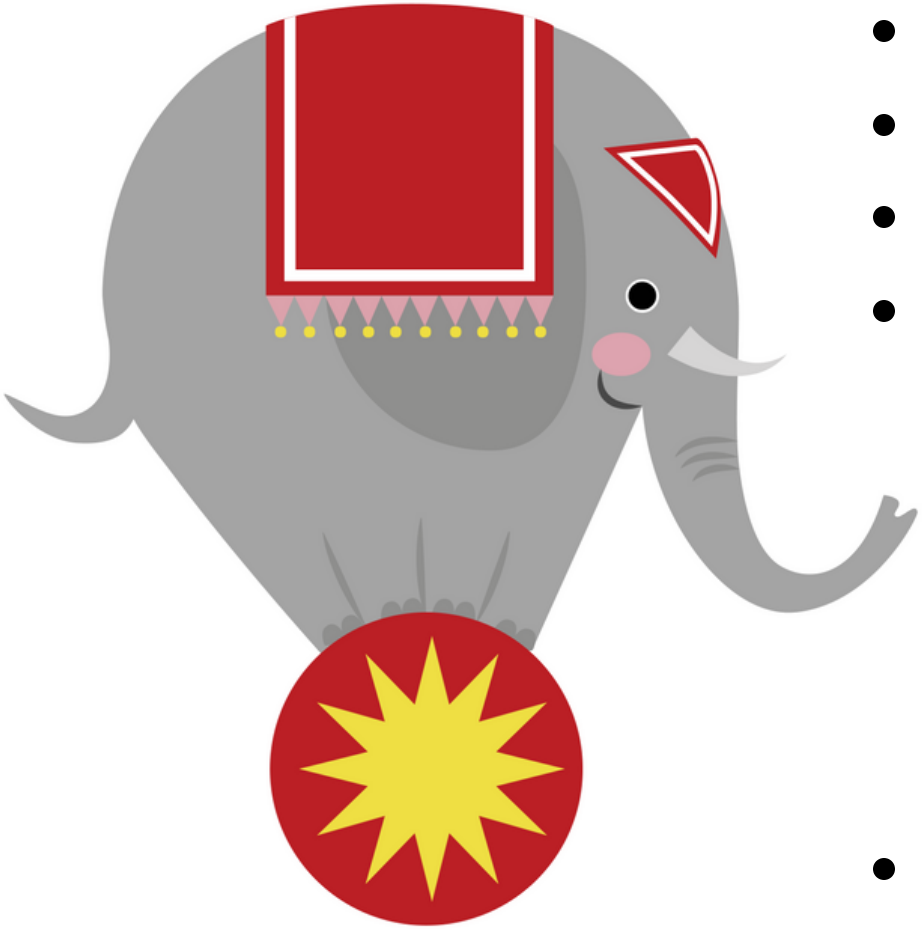
Lives in Prague



@apatheticmagpie

Github: gulcin

Agenda



- Design choices of PostgreSQL
- Arrays, Enum, JSON
- JSONB and GIN
- Full Text Search in PostgreSQL
  - tsvector, tsquery
  - Ranking
  - Misspelling
  - Accent support
  - Language support
- Why PostgreSQL?

Design Choices of PostgreSQL

- Conventional Relational PostgreSQL
  - Tables, Columns, Rows, Query Processing
- Object Relational PostgreSQL
  - Extensibility
    - Rich type system
    - Wide variety of index types
- Power of combining all
  - Following SQL standards
  - ACID properties

Arrays

- Standard arrays
- Array operators (@>, <@, &&, =, <> etc)
- Search in the array
- Process array elements from SQL directly
- Index them with GIN
  - This index access method allows PostgreSQL to index the contents of the arrays, rather than each array as an opaque value.

Arrays

```
                                    Table "public.film"
      Column       |            Type             | Collation | Nullable |
-------------------+-----------------------------+-----------+----------+-----------
 film_id           | integer                     |           | not null | nextval('f
 title             | text                        |           | not null |
 description       | text                        |           |          |
 release_year      | year                        |           |          |
 language_id       | smallint                    |           | not null |
 original_language_id | smallint                 |           |          |
 rental_duration   | smallint                    |           | not null | 3
 rental_rate       | numeric(4,2)                |           | not null | 4.99
 length            | smallint                    |           |          |
 replacement_cost  | numeric(5,2)                |           | not null | 19.99
 rating            | mpaa_rating                 |           |          | 'G'::mpaa_
 last_update       | timestamp with time zone    |           | not null | now()
 special_features  | text[]                      |           |          |
 fulltext          | tsvector                    |           | not null |
```

Arrays

```
fts_demo=> Select film_id, special_features from film
           where special_features @> array['Deleted Scenes'] limit 15;
 film_id |                  special_features
---------+-----------------------------------------------------
       1 | {"Deleted Scenes","Behind the Scenes"}
       2 | {Trailers,"Deleted Scenes"}
       3 | {Trailers,"Deleted Scenes"}
       5 | {"Deleted Scenes"}
       6 | {"Deleted Scenes"}
       7 | {Trailers,"Deleted Scenes"}
       9 | {Trailers,"Deleted Scenes"}
      10 | {Trailers,"Deleted Scenes"}
      12 | {Commentaries,"Deleted Scenes"}
      13 | {"Deleted Scenes","Behind the Scenes"}
      14 | {Trailers,"Deleted Scenes","Behind the Scenes"}
      19 | {Commentaries,"Deleted Scenes","Behind the Scenes"}
      20 | {Commentaries,"Deleted Scenes","Behind the Scenes"}
      23 | {Trailers,"Deleted Scenes"}
      26 | {Commentaries,"Deleted Scenes"}
(15 rows)
```

Arrays

```
fts_demo=> CREATE INDEX idx_sp_features ON film USING GIN(special_features);
CREATE INDEX

fts_demo=> Explain analyze (Select * from film
                            where special_features @> array['Deleted Scenes']);
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film  (cost=11.90..73.19 rows=503 width=386) (actual time=0.058
   Recheck Cond: (special_features @> '{"Deleted Scenes"}'::text[])
   Heap Blocks: exact=55
   ->  Bitmap Index Scan on idx_sp_features  (cost=0.00..11.77 rows=503 width=0) (ac
         Index Cond: (special_features @> '{"Deleted Scenes"}'::text[])
 Planning time: 0.512 ms
 Execution time: 0.267 ms
(7 rows)
```

Enum

- Lookup table
- Stores integer instead of whole value in table
- Denormalized, you don't need a separate table
- Faster reads
- Intended for static sets of values
- Takes very little space, four bytes on disk
- All of this is indexable! \o/

Enum

```
create type status as enum('backlog', 'in-progress', 'done', 'delivered');

create table issues
 (
   id bigint primary key,
   description   text,
   state status
 );

insert into issues(id, description, state)
     values (1, 'Implement Job for Switching DNS API Call', 'backlog'),
            (2, 'Report an issue mechanism for customers', 'in-progress'),
            (3, 'Cost reports', 'done'),
            (4, 'Scheduled Jobs Mechanism', 'delivered');

fts_demo=> Select * from issues where state = 'in-progress';
 id |              description                |    state
----+-----------------------------------------+-------------
  2 | Report an issue mechanism for customers | in-progress
(1 row)
```

Enum

```
fts_demo=> set enable_seqscan = off;
SET

fts_demo=> create index idx_state on issues(state);
CREATE INDEX

fts_demo=> Explain analyze (Select * from issues where state = 'in-progress');
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Index Scan using idx_state on issues  (cost=0.13..8.15 rows=1 width=44) (actual tim
   Index Cond: (state = 'in-progress'::status)
 Planning time: 0.054 ms
 Execution time: 0.023 ms
(4 rows)
```

JSON

- Validated as correct JSON
- Stores as text
- Keeps the same format as it sent
- Useful if;
  - you want to store bunch of JSON (fast)
  - you don't need to search in JSON itself
- Fast to write
  - you don't transform but only validate
- More intensive to search
  - you obviously interpret it every time you access it

JSON

```
create table js(id serial primary key, extra json);
insert into js(extra)
     values ('[1, 2, 3, 4]'),
            ('[2, 3, 5, 8]'),
            ('{"key": "value"}');

fts_demo=> select * from js where extra @> '2';
ERROR:  operator does not exist: json @> unknown
LINE 1: select * from js where extra @> '2';
                                     ^
HINT:  No operator matches the given name and argument type(s). You might need to ad

alter table js alter column extra type jsonb;

fts_demo=> select * from js where extra @> '2';
 id |    extra
----+--------------
  1 | [1, 2, 3, 4]
  2 | [2, 3, 5, 8]
(2 rows)
```

JSONB

- JSONB is already stored in (internal binary format) interpreted form. This means:
  - storing take a little while longer (more CPU process)
  - but processing (retrieval) faster
- The main thing is **all JSON document** can be indexed with a **single GIN** index. (jsonb_path_ops vs jsonb_ops)

```
fts_demo=> create index on js using gin (extra jsonb_path_ops);
CREATE INDEX
```

JSONB

```
fts_demo=> explain analyze (select * from js where extra @> '2');
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on js  (cost=8.00..12.01 rows=1 width=36) (actual time=0.011..0.01
   Recheck Cond: (extra @> '2'::jsonb)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on js_extra_idx  (cost=0.00..8.00 rows=1 width=0) (actual t
         Index Cond: (extra @> '2'::jsonb)
 Planning time: 0.054 ms
 Execution time: 0.031 ms
(7 rows)

fts_demo=> explain analyze (select * from js where extra @> '[2,3]');
                                                            QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on js  (cost=12.00..16.01 rows=1 width=36) (actual time=0.012..0.0
   Recheck Cond: (extra @> '[2, 3]'::jsonb)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on js_extra_idx  (cost=0.00..12.00 rows=1 width=0) (actual
         Index Cond: (extra @> '[2, 3]'::jsonb)
 Planning time: 0.053 ms
 Execution time: 0.032 ms
(7 rows)
```

JSONB

- Interpreted format is different than what you sent originally, it goes through normalisation:
  - keys are sorted
  - duplicated keys are removed and only first ones are saved
  - whitespaces removed etc.
- Fits into JSON standard (JSONB is Postgres' JSON)
  - schemaless PostgreSQL
  - heterogeneous set of documents all in a single relation
  - semi-structured data model

GIN

Generalised **I**nverted **I**ndex

Why?

forward indexes

list of documents and which words appear in them

- there is almost no duplication

backward (inverted) indexes

list of words and in which documents they appeared

- it is efficient
- duplicate data in values
- the more duplication the more efficient indices

GIN

ID	Document
1	PostgreSQL is awesome
2	Awesome things happen
3	Prague loves PostgreSQL
4	Prague is awesome too!
5	Thanks!

inverted index simplified

Term	Document ID
awesome	1, 2, 4
happen	2
is	1, 4
loves	3
prague	3, 4
postgresql	1, 3
thanks	5
things	2
too	4

(,)

key

posting list

GIN

- GIN is an index that allows indexing of complex data types
  - Postgres data types extract keys and positions of them
  - Key is data type specific
  - In the case of JSON it can store of the paths of JSONB documents. This is its key.
- GIN is very efficient in duplicate keys (GIN keys)
  - Keys of JSON != Keys of GIN
- GIN has more compact way of storing **duplicate values** (keys) than B Tree

FTS in PostgreSQL

FTS in PostgreSQL

- FTS is implemented in a similar fashion like JSONB type:
  - there are types like **ts_vector** which get text input and parses into lexemes
- Difference between JSONB:
  - ts_vector only stores info that is useful for FTS while JSONB stores the actual document as well
  - that has affect on how it is used afterwards:
    - JSONB is used as column type while ts_vector is mostly used for creating indexes as index definition or compound values (indexing multiple columns at the same time)

tsvector

tsvector which is a type suited to full-text search

```
fts_demo=# SELECT to_tsvector( Happiness is an allegory  unhappiness a story ');
                to_tsvector
-------------------------------------------
 'allegori':4 'happi':1 stori':7 'unhappi':5
(1 row)

fts_demo=# SELECT to_tsvector('Happiness is an allegory, unhappiness a story.')
           @@ 'happiness';
 ?column?
----------
 f
(1 row)
```

tsquery

tsquery stores lexemes that are to be searched for

```
fts_demo=# SELECT to_tsvector('Happiness is an allegory, unhappiness a story.')
           @@ to_tsquery('happiness');
 ?column?
----------
 t
(1 row)

fts_demo=# SELECT to_tsvector('Happiness is an allegory, unhappiness a story.')
           @@ to_tsquery('happiness & unhappiness');
 ?column?
----------
 t
(1 row)
```

Querying

```
Select title, description
from
    (select title, description, to_tsvector(title) ||
            to_tsvector(description) as searchterm
    from film) as q
where q.searchterm @@ to_tsquery('Human & Database')
limit 5;

      title      |                                                    descriptio
-----------------+-------------------------------------------------------------
 ANONYMOUS HUMAN | A Amazing Reflection of a Database Administrator And a Astronaut
 HUMAN GRAFFITI  | A Beautiful Reflection of a Womanizer And a Sumo Wrestler who mus
(2 rows)
```

Ranking

```
Select title, ts_rank(q.searchterm, to_tsquery('DINOSAUR | Feminist')) as searchrank
from
    (select title, description, setweight(to_tsvector(title), 'A') ||
            setweight(to_tsvector(description), 'B') as searchterm
    from film) as q
where q.searchterm @@ to_tsquery('DINOSAUR | Feminist')
order by searchrank desc
limit 5;
```

```
       title         | searchrank |                                                descri
---------------------+------------+-----------------------------------------------------
 ACADEMY DINOSAUR    |   0.425549 | A Epic Dram 1x Feminist And a Mad Scientist wh
 DINOSAUR SECRETARY  |   0.425549 | A Action-Packed Dram 1x Feminist And a Girl wh
 CENTER DINOSAUR     |   0.303964 | A B 0 ul Character Study of a Sumo Wrestler An
 SPY MILE            |   0.165491 | A Thrilling Documentary of a Feminist 3x Femin
 BUNCH MINDS         |   0.151982 | A Emotional Stor 2x Feminist And a Feminist wh
(5 rows)
```

Similarity Search Using Trigrams

hello

Trigram?

"h"	"h"
"he"	"ha"
"hel"	"hal"
"ell"	"all"
"llo"	"llo"
"lo"	"lo"
"o"	"o"

hallo

```
fts_demo=# Create extension pg_trgm;
CREATE EXTENSION

fts_demo=# select similarity('hello','hallo');
 similarity
------------
   0.333333
(1 row)
```

Similarity and Distance

%,<%, <->

```
fts_demo=# explain analyze select description from film
           where description %> 'Feminist';
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on film  (cost=10000000000.00..10000000067.50 rows=1 width=94) (actual tim
   Filter: (description %> 'Feminist'::text)
   Rows Removed by Filter: 916
 Planning time: 0.046 ms
 Execution time: 14.919 ms

fts_demo=# CREATE INDEX trgm_idx ON film USING GIN (description gin_trgm_ops);
CREATE INDEX

fts_demo=# explain analyze select description from film
           where description %> 'Feminist';
                                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film  (cost=76.01..80.02 rows=1 width=94) (actual time=0.113..1
   Recheck Cond: (description %> 'Feminist'::text)
   Rows Removed by Index Recheck: 29
   Heap Blocks: exact=49
   ->  Bitmap Index Scan on trgm_idx  (cost=0.00..76.01 rows=1 width=0) (actual time
         Index Cond: (description %> 'Feminist'::text)
 Planning time: 0.132 ms
 Execution time: 1.970 ms
```

Like Queries

LIKE, ILIKE, ~, ~*

```
fts_demo=# Explain analyze select description from film
           where description like '%Feminist%';
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on film  (cost=52.63..111.30 rows=81 width=94) (actual time=0.052.
   Recheck Cond: (description ~~ '%Feminist%'::text)
   Heap Blocks: exact=42
   ->  Bitmap Index Scan on trgm_idx  (cost=0.00..52.61 rows=81 width=0) (actual tim
         Index Cond: (description ~~ '%Feminist%'::text)
 Planning time: 0.108 ms
 Execution time: 0.135 ms
(7 rows)
```

Misspelling

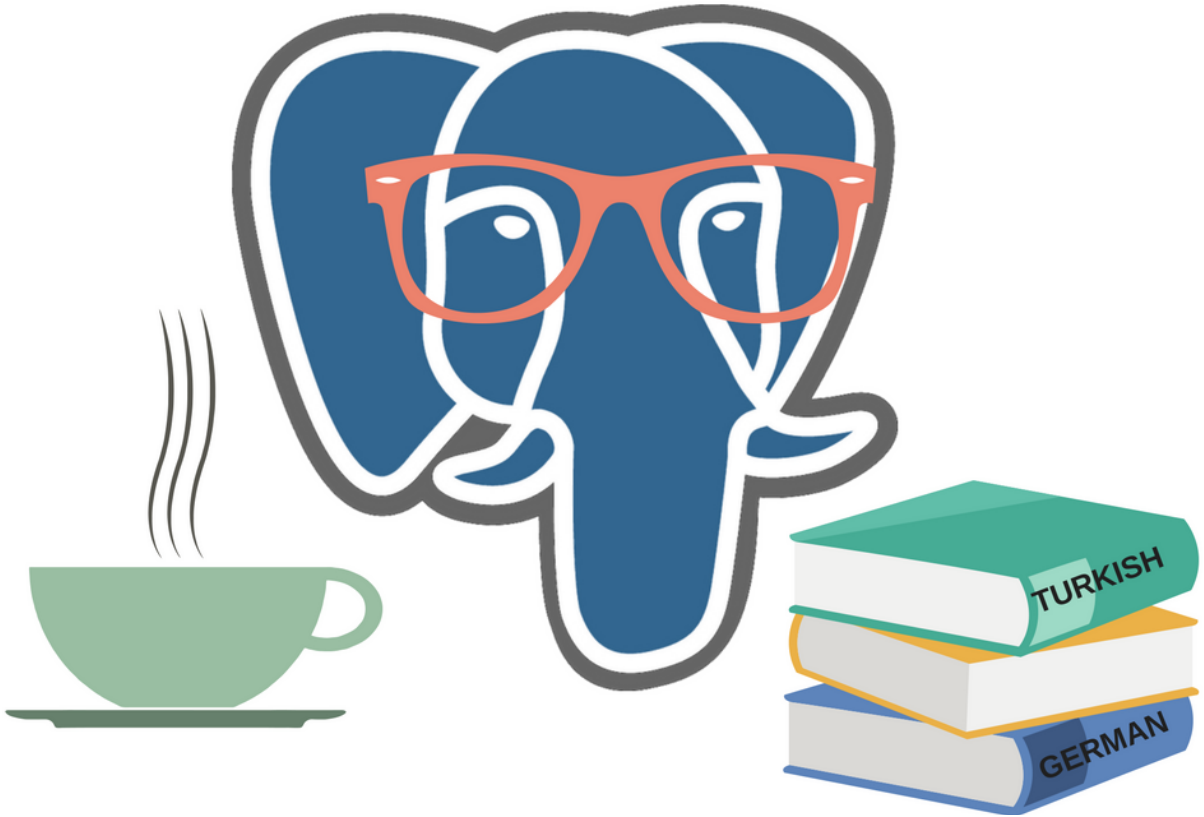
```
fts_demo=# CREATE TABLE unique_lexeme AS
           SELECT word FROM ts_stat(
           'SELECT to_tsvector(''simple'', first_name) ||
               to_tsvector(''simple'', last_name)
           FROM actor
           GROUP BY actor_id');

fts_demo=# CREATE INDEX lexeme_idx ON unique_lexeme USING GIN (word gin_trgm_ops);
CREATE INDEX

fts_demo=# SELECT word from unique_lexeme
           WHERE similarity(word, 'sinatro') > 0.5
           ORDER BY word <-> 'sinatro'
           LIMIT 10;
  word
---------
 sinatra
(1 row)
```

Multilingual PostgreSQL

Built-in text search for Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish.



Accent Support

```
CREATE EXTENSION unaccent;

SELECT unaccent('Gülçin Yıldırım Jelínek');
        unaccent
-------------------------
 Gulcin Yildirim Jelinek
(1 row)

fts_demo=# CREATE TEXT SEARCH CONFIGURATION tr ( COPY = turkish );
CREATE TEXT SEARCH CONFIGURATION
fts_demo=# ALTER TEXT SEARCH CONFIGURATION tr
           ALTER MAPPING FOR hword, hword_part, word WITH unaccent, turkish_stem;
ALTER TEXT SEARCH CONFIGURATION

fts_demo=# SELECT to_tsvector('tr', 'Gülçin') @@ to_tsquery('tr', 'gulcin') as resul
 result
--------
 t
(1 row)

fts_demo=# set default_text_search_config to 'tr';
SET
fts_demo=# SELECT to_tsvector('Gülçin') @@ to_tsquery('gulcin') as result;
 result
--------
 t
(1 row)
```

PostGIS

Geospatial search in PostgreSQL? GIN? Yes, ofc!

Why PostgreSQL?

Advantages of PostgreSQL over using a search engine:

- You can use the existing relations
- You can query related information (joins)
- You can do all in one query (transactional)
- When you update (insert, delete) your document, indexes are updated automatically
  - Rebuilding indexes are not a concern
  - FTS is always up-to-date (no 404)
- Same ACID properties
- You don’t need to maintain two techs (two dataset)

Why PostgreSQL?

JSONB

- Stable schema and flexibly evolving data in the same database
- Denormalisation without the downsides
  - No unnecesary tables
  - No unnecessary joins

```
fts_demo=# Select first_name, last_name, education from staff;
-[ RECORD 1 ]----------------------------------------------------------
first_name | Mike
last_name  | Hillyer
education  | {"properties": {"university": {"type": "oxford"}, "high school": {"name
-[ RECORD 2 ]----------------------------------------------------------
first_name | Jon
last_name  | Stephens
education  | {"properties": {"university": {"type": "tallinn university of technolog
```

References

- Thanks Petr Jelínek (<3) for the idea, proof-reading and all the support!
- Thanks Magnus Hagander for recommending the Pagila dataset.
- https://tapoueh.org/tags/data-types/
- http://rachbelaid.com/postgres-full-text-search-is-good-enough/
- http://www-old.bartlettpublishing.com/site/bartpub/blog/3/entry/350
- http://www.nomadblue.com/blog/django/from-like-to-full-text-search-part-ii/

Thank you! Questions?