Postgres Full Text Search Might Be Good Enough!

Nowadays when you have data, search becomes an essential part of your tool. It doesn't matter if you have a website or not, search is an important feature, which comes to mind before even starting to build the tool itself.

The most popular platforms in the last 5 years for search that we have seen are ElasticSearch and SOLR, both are Lucene based. They are very powerful and great tools, and solve almost all the tasks of search, but before going down the route of adding these massive tools in the existing toolset, let's look at something which is a bit lighter and should be good enough!

When we build a basic search feature in our app, we would at least want:

- JSON attributes support
- Indexing
- Stemming

Since the postgres > 8.3, it supports full text search with all of the above features and much more.

Lets see how easy it is to setup a basic search feature for your app.

Let's assume following schema and data:

```
> CREATE TABLE profiles (
    id SERIAL PRIMARY KEY,
    name TEXT
  );



> CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    account_id INTEGER NOT NULL,
    profile_id INTEGER NOT NULL
  );



> CREATE TABLE ads (
    account_id INTEGER NOT NULL,
    id SERIAL PRIMARY KEY,
    data jsonb
  );
```

```
> INSERT INTO profiles (id, name)
  VALUES (1, 'United Federation of Planets'), (2, 'Wayne Industries');


> INSERT INTO accounts (id, account_id, profile_id)
  VALUES (1, 999, 1), (2, 998, 1), (3, 666, 2);


> INSERT INTO ads (id, account_id, data)
  VALUES (1, 999, '{"name": "Plane Accessories on sale", "ad_id": 18976,
"type": "google"}'),
         (2, 999, '{"name": "Plane Taxi Service for free", "ad_id": 76859,
"type": "twitter"}'),
         (3, 998, '{"name": "Plane on rent for wedding", "ad_id": 34568,
"type": "youtube"}'),
         (4, 666, '{"name": "blow up plane training", "ad_id": 98576, "type":
"Facebook"}');
```

The setup is for any advertising platform, a profile, which has many accounts and each account has many ads.

When we talk about full text search, we usually talk about a document, simply put a document is like an article or content which is the subject area of search and may be a logical entity. It may be across multiple tables.

“In text retrieval, full text search refers to techniques for searching a single computer-stored document or a collection in a full text database.“

Document is not related to table schema but to data, for our use case the we could use any/all of the following as a document:

- profiles.name
- ads.data.name
- ads.data.ad_id
- ads.data.type

So let's create them.

```
> SELECT profiles.name || ' ' || cast(ads.data->>'name' as text) || ' ' ||
cast(ads.data->>'ad_id' as text) || ' ' || cast(ads.data->>'type' as text) as
document

FROM  ads JOIN accounts on accounts.account_id = ads.account_id

JOIN profiles on profiles.id = accounts.profile_id;
```

```
document
-----------------------------------------------------------------

Wayne Industries plane training 98576 Facebook

United Federation of Planets Plane on rent for wedding 34568 youtube

United Federation of Planets Plane Accessories on sale 18976 google

United Federation of Planets Plane Taxi Service for free 76859 twitter

(4 rows)
```

You can also use `string_agg()` function with `coalesce()` function in order to aggregate many strings and remove nulls.

Now our document is just a simple long string, and isn't really optimised to be searched, we need to format this in lexemes, which is more suited and understood by postgres for full text search.

*"A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token"*

To convert the above document to lexemes postgres provides us with `to_tsvector()`.

```
> SELECT to_tsvector(profiles.name) ||

  to_tsvector(cast(ads.data->>'name' as text)) ||

  to_tsvector(cast(ads.data->>'ad_id' as text)) ||

  to_tsvector(cast(ads.data->>'type' as text))

  as document

  FROM ads

  JOIN accounts on accounts.account_id = ads.account_id

  JOIN profiles on profiles.id = accounts.profile_id;


document

-----------------------------------------------------------------

'98576':7 'blow':3 'facebook':8 'industries':2 'plane':5 'training':6 'up':4
'wayne':1

'34568':10 'federation':2 'for':8 'of':3 'on':6 'plane':5 'planets':4 'rent':7
'united':1 'wedding':9 'youtube':11
```

```
'18976':9 'accessories':6 'federation':2 'google':10 'of':3 'on':7 'plane':5
'planets':4 'sale':8 'united':1

'76859':10 'federation':2 'for':8 'free':9 'of':3 'plane':5 'planets':4
'service':7 'taxi':6 'twitter':11 'united':1

(4 rows)
```

Now the result is slightly different.

A `tsvector` value is a sorted list of distinct lexemes which are words that have been normalized to make different variants of the same word look alike. For example, normalization almost always includes folding upper-case letters to lower-case and often involves removal of suffixes (such as ‘s’, ‘es’ or ‘ing’ in English). This allows searches to find variant forms of the same word without tediously entering all the possible variants. The number represents the occurrence of the word in the original string.

By default, Postgres uses ‘english’ as text search configuration for the function `to_tsvector` and it will also ignore english stopwords. That explains why the `tsvector` results have fewer elements than the ones in our sentence. We see later a bit more about languages and text search configuration.

Now in order to query a `tsvector` document, lets try following sql:

```
> select to_tsvector('If you can dream it, you can do it') @@ 'dream';

?column?

----------

t

(1 row)

> to_tsvector('It''s kind of fun to do the impossible') @@ 'impossible';

?column?

----------

f
```

The second result is false because we need to build this using lexemes and so we need two things `to_tsquery()` function and operator `@@`.

The following query shows the difference.

```
> SELECT 'impossible'::tsquery, to_tsquery('impossible');

tsquery | to_tsquery

--------------+------------

'impossible' | 'imposs'

(1 row)

> SELECT 'dream'::tsquery, to_tsquery('dream');

tsquery | to_tsquery

--------------+------------

'dream' | 'dream'

(1 row)
```

In the above case of ‘dream’ the stem is also dream but ‘impossible’ stem is ‘imposs’ and that's why our `to_tsvector` query returned false.

Now if we run the above query using `ts_query` we get

```
> SELECT to_tsvector('It''s kind of fun to do the impossible') @@
to_tsquery('impossible');


?column?

----------

t

(1 row)
```

`ts_query` function can take all the boolean operators `&` (AND), `|` (OR), and `!` (NOT)

Let's come back to the main problem and construct the final query, which should look like the following.

```
> SELECT profile_name, ad_name, ad_id, vendor

  FROM (SELECT profiles.name as profile_name,

  ads.data->>'name' as ad_name,

  ads.data->>'ad_id' as ad_id,
```

```
  ads.data->>'type' as vendor,
  to_tsvector(profiles.name) ||
  to_tsvector(cast(ads.data->>'name' as text)) ||
  to_tsvector(cast(ads.data->>'ad_id' as text)) ||
  to_tsvector(cast(ads.data->>'type' as text))
  as document
  FROM ads
  JOIN accounts on accounts.account_id = ads.account_id
  JOIN profiles on profiles.id = accounts.profile_id) as ad_search
  WHERE ad_search.document @@ to_tsquery('United');

profile_name | ad_name | ad_id | vendor
------------------------------+------------------------------+-------+-------
United Federation of Planets | Plane on rent for wedding | 34568 | youtube
United Federation of Planets | Plane Accessories on sale | 18976 | google
United Federation of Planets | Plane Taxi Service for free | 76859 | twitter
(3 rows)
```

This returns our results with united lexemes.

Now lets do some optimizations.Since the document is spread across multiple tables, its necessary to denormalize the data or create materialized view.

As per Postgres: “*Materialized views in PostgreSQL use the rule system like views do, but persist the results in a table-like form*”.

The main differences between:

```
> CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

vs

```
> CREATE TABLE mymatview AS SELECT * FROM mytab;
```

is that the materialized view cannot subsequently be directly updated and that the query used to create the materialized view is stored in exactly the same way that a view's query is stored, so that fresh data can be generated for the materialized view with:

```
> REFRESH MATERIALIZED VIEW mymatview;
```

So let's create one for our search.

```
> CREATE MATERIALIZED VIEW ad_search
AS SELECT profiles.name as profile_name,
ads.data->>'name' as ad_name,
ads.data->>'ad_id' as ad_id,
ads.data->>'type' as vendor,
to_tsvector(profiles.name) ||
to_tsvector(cast(ads.data->>'name' as text)) ||
to_tsvector(cast(ads.data->>'ad_id' as text)) ||
to_tsvector(cast(ads.data->>'type' as text))
as document
FROM ads
JOIN accounts on accounts.account_id = ads.account_id
JOIN profiles on profiles.id = accounts.profile_id;
```

Then reindexing the search engine will be as simple as periodically running

```
> REFRESH MATERIALIZED VIEW ad_search;
```

We can now add an index on the materialized view.

```
> CREATE INDEX idx_ad_search ON ad_search USING gin(document);
```

Now our query also becomes much simpler.

```
> SELECT profile_name, ad_name, ad_id, vendor
  FROM ad_search
  WHERE document @@ to_tsquery('United');
```

There are so many other things that we can do with postgres, like:

- Ranking – providing weight to results and then ordering on the basis of that using `ts_rank()` function
- Similarities search – covers misspelling and fuzzy search behaviours using `similarity()` function
- Support for multiple languages
- Support for accented characters
-

Conclusion

This article aims to give you a brief insight into building your own search with postgres and we have seen how easy it is to build a document based search using it . Postgres might not be as advanced as ElasticSearch and SOLR, but these two are dedicated full-text search tools whereas full-text search is only a feature of PostgreSQL and a pretty good one.