Advanced Database

COMP412

CHAPTER 05: FULL TEXT SEARCH

Introduction

Real life scenario

- Let’s say you have an IT store and the user searched for ‘**desktop and laptop**’.
- No problem there. But do you have a product whose title says ‘**XXX Desktop and Laptop**’ exactly as the user searched for?
- Most probably no! The search would fail to show any relevant results.
- The user probably wanted to list all the computers in your store that he or she can use as a desktop and a laptop, most likely a **convertible tablet**.
- Since the search failed to show any result to the user, the user may think you’re out of stock or you don’t have it in your IT store.

Real life scenario

- But you do have many convertible tablets that can be used as a desktop and a laptop in your store’s database.
- If the users can’t find it, you won’t get any sales.
- You do want to your website to list all the convertible computers you have in stock when users do a search query like that.

This is where Full Text Search comes into play.

Demo

- Create Store Database

```
createdb it_store
```

- Create products table

```
CREATE TABLE public.products
    (id serial primary key,
    title character varying(200) NOT NULL,
    description text NOT NULL )
```

Demo

- Insert some products into the **products** table.
  - INSERT INTO public.products(title, description)   VALUES ('HP ProDesk 400 G3 4GB RAM 1TB HDD', 'This is great and cheap desktop computer');
  - INSERT INTO public.products(title, description)   VALUES ('ASUS UX303UB 8GB RAM 256GB SSD', 'This is a great think and light laptop computer. it has good sound quality');
  - INSERT INTO public.products(title, description)   VALUES ('Microsoft Surface Book 2 1TB SSD', 'this is a great convertible laptop computer, you can use it as a laptop and desktop as well if you want.');

Demo

- In PostgreSQL, you use two functions to perform Full Text Search. They are **to_tsvector()** and **to_tsquery()**.

- **to_tsvector()** function breaks up the input string and creates tokens out of it, which is then used to perform Full Text Search using the **to_tsquery()** function.

```
select to_tsvector('I love linux. Linux IS a great operating system.');
```

- You can use **to_tsquery()** function as follows:

```
SELECT fieldNames FROM tableName
   WHERE to_tsvector(fieldName) @@ to_tsquery(conditions)
```

Demo

- If you’re looking for ‘laptop and desktop’, you should put **‘laptop & desktop’** to **to_tsquery()** function.
  - For ‘laptop or desktop’, the condition should be **‘laptop | desktop**’.

```
SELECT id, title, description FROM public.products where
to_tsvector(description) @@ to_tsquery('desktop & laptop')
```

- Let’s take a look at another example. The user is looking for all the laptops in your store but not the convertible ones. The user query may be ‘not convertible laptops’. The condition of to_tsquery() function may be ‘!convertible & laptops’

```
to_tsquery('!convertible & laptops')
```

FTS Configurations

- Most PostgreSQL distributions come packaged with over 10 FTS configurations. All these are installed in the pg_catalog schema.
- To see the listing of installed FTS configurations, run the query

```
SELECT cfgname FROM pg_ts_config;
```

- You’re not limited to built-in FTS configurations. You can create your own

FTS Configurations

- Install the popular hunspell configuration.
  - Start by downloading hunspell configurations from hunspell_dicts.
  - Copy **en_us.affix** and **en_us.dict** to your PostgreSQL installation directory **share/tsearch_data**.
  - Copy the **hunspell_en_us--*.sql** and **hunspell_en_us.control** files to your Post-greSQL installation directory **share/extension** folder.
  - Next, run: CREATE EXTENSION hunspell_en_us SCHEMA pg_catalog;

Default configuration

- Not sure which configuration is the default? Run:
  - SHOW default_text_search_config;

- To replace the default with another, run:
  - ALTER DATABASE postgresql_book SET default_text_search_config = 'pg_catalog.english';

TSVector

- To create a **tsvector** from text, you must specify the FTS configuration to use.
- The vectorization reduces the original text to a set of word skeletons, referred to as lexemes, by removing stop words.
- For each lexeme, the TSVector records where in the original text it appears.
- The more frequently a lexeme appears, the higher the weight. Each lexeme therefore is imbued with at least one position, much like a vector in the physical sense.
- Use the **to_tsvector** function to vectorize a text. This function will resort to the default FTS configuration unless you specify another.

Example: TSVector derived from different FTS configurations

- select to_tsvector('Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('english','Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('english_hunspell','Just dancing in the rain. I like to dance.'::text) ;
- select to_tsvector('simple','Just dancing in the rain. I like to dance.'::text)

TSVector

- English and Hunspell configurations remove all stop words, such as just and to.

- English and Hunspell also convert words to their normalized form as dictated by their dictionaries, so dancing becomes danc and dance, respectively.

- The to_tsvector function returns where each lexeme appears in the text. So, for example, 'danc':2,9 means that dancing and dance appear as the second and the ninth words.

Add FTS to the database

- To incorporate FTS into your database, add a tsvector column to your table.
  - You then either schedule the tsvector column to be updated regularly, or
  - add a trigger to the table so that whenever relevant fields update, the tsvector field recomputes.

Create table with tsvector column

- CREATE TABLE film1 (
-     film_id1 integer DEFAULT nextval('film_film_id_seq'::regclass) NOT NULL,
-     title character varying(255) NOT NULL,
-     description text,
-     fulltext tsvector NOT NULL
-);

- UPDATE film1 SET fulltext = setweight(to_tsvector(COALESCE(title,'')),'A')
                                                    ||
  setweight(to_tsvector(COALESCE(description,'')),'B');

Setweight and (||) operator

- We've introduced two new constructs, the setweight function and the concatenation operator (||), to tsvector.
- To distinguish the relative importance of different lexemes, you could assign a weight to each. The weights must be A, B, C, or D, with A ranking highest in importance.
- TSVectors can be formed from other tsvectors using the concatenation (||) operator.
  - We used it here to combine the title and description into a single tsvector.
  - This way when we search, we have to contend with only a single column.

Trigger to automatically update tsvector

- Should data change in one of the basis columns forming the tsvector, you must revectorize.
- To avoid having to manually run to_tsvector every time data changes, create a trigger that responds to updates.

  - CREATE TRIGGER trig_tsv_film_iu
  - BEFORE INSERT OR UPDATE OF title, description ON film FOR EACH ROW
  - EXECUTE PROCEDURE tsvector_update_trigger(fts,'pg_catalog.english',
  - title,description);

Create index for tsvector

- To speed up searches, we add a GIN index on the tsvector column.

  - CREATE INDEX ix_film1_fts_gin ON film1 USING gin (fulltext);

TSQueries

- We have already seen how to vectorize the searched text to create tsvector columns.
  - We now show you how to vectorize the search terms.
- FTS refers to vectorized search terms as tsqueries,
- PostgreSQL offers several functions that will convert plain-text search terms to tsqueries: to_tsquery, plainto_tsquery, and phraseto_tsquery.
  - The latter takes the ordering of words in the search term into consideration.

to_tsquery

- Using the to_tsquery functions against two configurations: the default English configuration and the Hunspell configuration.
  - SELECT to_tsquery('business & analytics');
  - SELECT to_tsquery('english_hunspell','business & analytics');
  - The and operator (&) means that both words must appear in the searched text.
  - The or operator (|) means one or both of the words must appear in the searched text.

plainto_tsquery

- A slight variant of to_tsquery is plain_totsquery. This function automatically inserts the and operator between words for you, saving you a few key clicks.
  - SELECT plainto_tsquery('business analytics');

phraseto_tsquery

- to_tsquery and plainto_tsquery look only at words, not their sequence.
- The phraseto_tsquery vectorizes the words, inserting the distance operator between the words.
- This means that the searched text must contain the words business and analytics in that order, upgrading a word search to a phrase search.
  - SELECT phraseto_tsquery('business analytics');
  - SELECT phraseto_tsquery('english_hunspell','business analytics');

Combining tsqueries

- SELECT plainto_tsquery('business analyst') || phraseto_tsquery('data scientist');
- SELECT plainto_tsquery('business analyst') && phraseto_tsquery('data scientist');

Using Full Text Search

- We have created a tsvector from our text; we have created a tsquery from our search terms.
  - Now, we can perform an FTS. We do so by using the @@ operator.
  - finds all films with a title or description containing the word hunter and either the word scientist, or the word chef, or both.

```
SELECT title , description
FROM film
WHERE fulltext @@ to_tsquery('hunter  & (scientist | chef)') AND title > '';
```

Using Full Text Search

- you can specify the proximity and order of words.

```
SELECT title , description
FROM film
WHERE fulltext @@ to_tsquery('hunter <4> (scientist | chef)') AND title > '';
```

Ranking Results