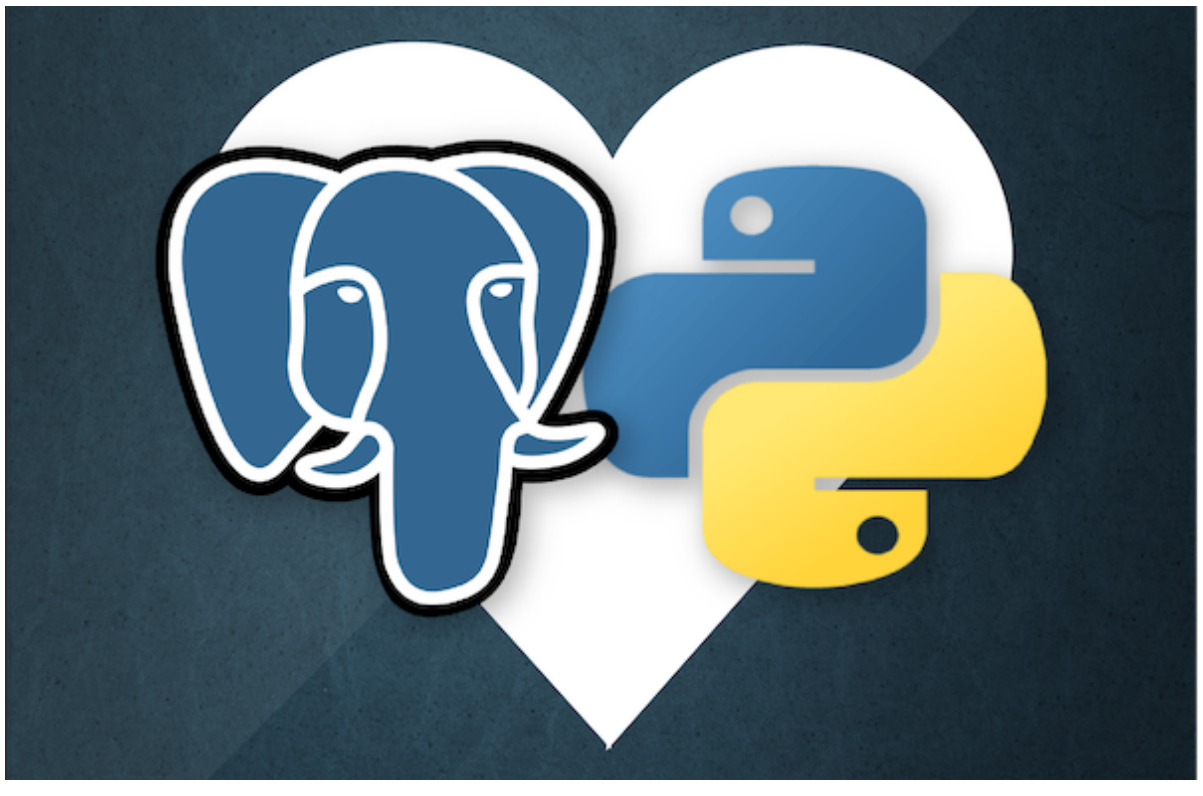


Optimizing Postgres full text search in Django

Dani Hodovic
June 12, 2019

Postgres provides great search capability out of the box. For the majority of Django apps there is no need to run and maintain an ElasticSearch cluster unless you need the advanced features ElasticSearch offers. Django integrates nicely with the Postgres search through the built in Postgres module.

For small data sets the default configuration performs well, however when your data grows in size the default search configuration becomes painfully slow and we need to enable certain optimizations to keep our queries fast.

This page will walk you through setting up Django and Postgres, indexing sample data and performing and optimizing full text search.

The examples go through a Django + Postgres setup, however the advice is generally applicable to any programming language or framework as long as it uses Postgres.

If you're already a Django veteran you can skip the first steps and jump right into the optimization.

Table of Contents

Project setup

Create the directories and setup the Django project.

```
mkdir django_postgres
cd django_postgres
python -m venv venv
source venv/bin/activate
pip install django
django-admin startproject full_text_search
cd full_text_search
./manage.py startapp web
```

Now we'll need to install 3 dependencies:

- psycopg2: the Postgres client library for Python
- wikipedia: a client library to retrieve Wikipedia articles
- django-extensions: to simplify debugging of SQL queries

```
pip install psycopg2 wikipedia django-extensions
```

We also need to run Postgres locally. I'll use a dockerized version of Postgres here since it's easier to set up, but feel free to install a Postgres binary if you'd like.

Open `full_text_search/docker-compose.yml`

```
---
version: '2.4'
services:
  postgres:
    image: postgres:11-alpine
    ports:
      - '5432:5432'
    environment:
      # Set the Postgres environment variables for bootstrapping the default
      # database and user.
      POSTGRES_DB: "my_db"
      POSTGRES_USER: "me"
      POSTGRES_PASSWORD: "password"
```

The project structure should now look like the output below. We'll ignore the venv directory as that is packed with files and irrelevant for now.

```
$ tree -I venv
.
└── full_text_search
    ├── docker-compose.yml
    ├── full_text_search
    │   ├── __init__.py
    │   ├── __pycache__
    │   │   ├── __init__.cpython-37.pyc
    │   │   └── settings.cpython-37.pyc
    │   ├── settings.py
    │   ├── urls.py
    │   └── wsgi.py
    ├── manage.py
    └── web
        ├── admin.py
        ├── apps.py
        ├── __init__.py
        ├── migrations
        │   └── __init__.py
        ├── models.py
        ├── tests.py
        └── views.py

5 directories, 15 files
```

We will modify the default database settings to use Postgres instead of SQLite. In settings.py change the `DATABASES` attribute:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "my_db",
        "USER": "me",
        "PASSWORD": "password",
        "HOST": "localhost",
        "PORT": "5432",
        "OPTIONS": {"connect_timeout": 2},
    }
}
```

We will also modify our `INSTALLED_APPS` to include a few apps:

- `django.contrib.postgres` the Postgres module for Django which is required for full text search
- `django_extensions` to print SQL logs when executing queries in Python
- our `web` app

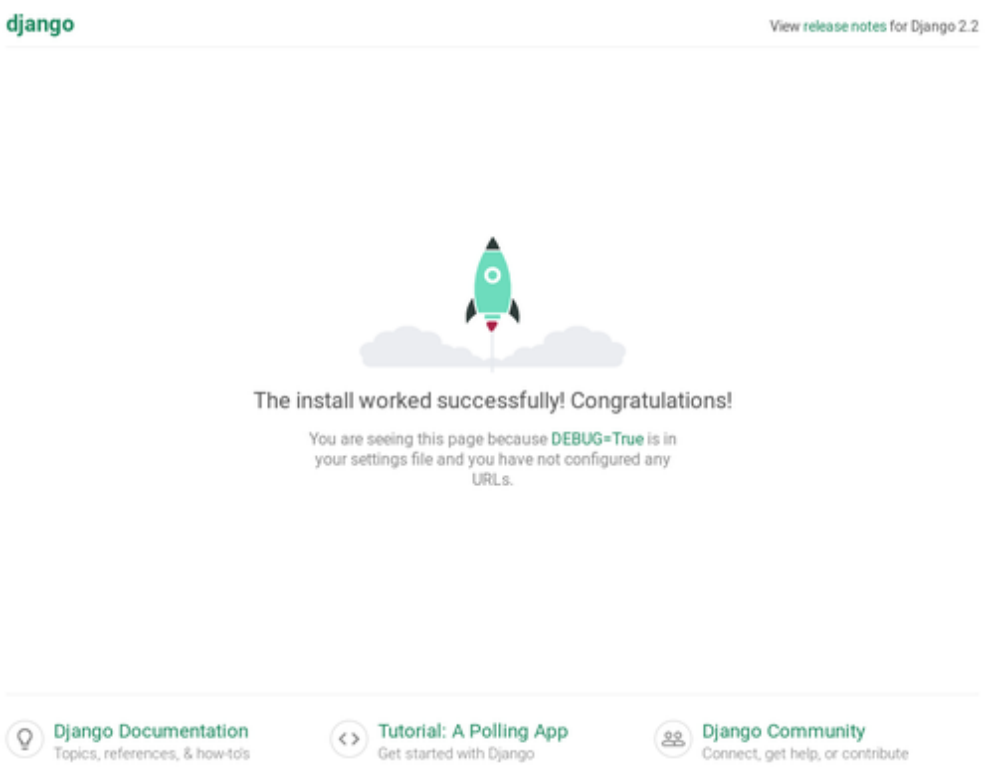
Open `full_text_search/settings.py` and modify:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Added apps below
    'django.contrib.postgres',
    'django_extensions',
    'web',
]
```

Start Postgres and Django.

```
cd full_text_search
docker-compose up -d
./manage.py runserver
```

If we open our browser and enter http://localhost:8000 we should see a successful installation.



Creating models and indexing sample data

Suppose we have a model which represents a Wikipedia page. For simplicity we'll use two fields: title and content.

Open `full_text_search/web/models.py`

```
from django.db import models

class Page(models.Model):
    title = models.CharField(max_length=100, unique=True)
    content = models.TextField()
```

Now run the migrations to create the model.

```
cd full_text_search
./manage.py makemigrations && ./manage.py migrate
No changes detected
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  # truncated the other output for brievity
```

We'll use a script to index random wikipedia articles and save the contents to Postgres.

Edit `web/index_wikipedia.py`

```
import logging
import wikipedia
from .models import Page

logger = logging.getLogger("django")

def index_wikipedia(num_pages):
    for _ in range(0, num_pages):
        p = wikipedia.random()
        try:
            wiki_page = wikipedia.page(p)
            Page.objects.update_or_create(title=wiki_page.title, defaults={
                "content": wiki_page.content
            })
        except Exception:
            logger.exception("Failed to index %s", p)
```

Now let's run our script to index Wikipedia. There will be errors when running the script, but don't worry about those as long as we manage to store a few hundred articles. The script will take a while to run so grab a cup of coffee and return in a few minutes.

```
./manage.py shell_plus

>>> from web.index_wikipedia import index_wikipedia
>>> index_wikipedia(200)

#
## A bunch of errors will be show here, ignore them.
#

>>> Page.objects.count()
183
```

Optimizing the search

Now suppose we want to allow users to perform a full text search on the content. We'll interactively query our dataset to test the full text search. Open a Django shell session:

```
$ ./manage.py shell_plus --print-sql

>>> Page.objects.filter(content__search='football')
SELECT t.oid,
       typarray
  FROM pg_type t
  JOIN pg_namespace ns
    ON typnamespace = ns.oid
 WHERE typname = 'hstore'

Execution time: 0.001440s [Database: default]

SELECT typarray
  FROM pg_type
 WHERE typname = 'citext'

Execution time: 0.000260s [Database: default]

SELECT "web_page"."id",
       "web_page"."title",
       "web_page"."content"
  FROM "web_page"
 WHERE to_tsvector(COALESCE("web_page"."content", '')) @@ (plainto_tsquery('football')) = true
 LIMIT 21

Execution time: 0.222619s [Database: default]

<QuerySet [<Page: Page object (2)>, <Page: Page object (7)>...]>
```

Django performs two preparatory queries and finally executes our search query. Looking at the last query we can at first glance see that the execution was ~315ms for the query execution and serialization alone. That's it far too slow when we want to keep our page load speeds in the double digits in milliseconds.

Let's take a closer look at why this query is performing so slowly. Open a second terminal where we'll use the excellent Postgres query analyzer. Copy the query from above and run `EXPLAIN ANALYZE`:

```
$ ./manage.py dbshell
psql (10.8 (Ubuntu 10.8-0ubuntu0.18.10.1), server 11.2)

Type "help" for help.

my_db=# explain analyze SELECT "web_page"."id",
my_db-#        "web_page"."title",
my_db-#        "web_page"."content"
my_db-#   FROM "web_page"
my_db-#  WHERE to_tsvector(COALESCE("web_page"."content", '')) @@ (plainto_tsquery('football')) = true
my_db-#  LIMIT 21
my_db-# ;
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.00..106.71 rows=1 width=643) (actual time=5.001..220.212 rows=18 loops=1)
   ->  Seq Scan on web_page  (cost=0.00..106.71 rows=1 width=643) (actual time=4.999..220.206 rows=18 loops=1)
         Filter: (to_tsvector(COALESCE(content, ''::text)) @@ plainto_tsquery('football'::text))
         Rows Removed by Filter: 165
 Planning Time: 3.336 ms
 Execution Time: 220.292 ms
(6 rows)
```

We can see that although the planning time is quite fast (~3ms) the execution time is very slow at ~220ms.

```
->  Seq Scan on web_page  (cost=0.00..106.71 rows=1 width=643) (actual time=4.999..220.206 rows=18 loops=1)
```

We can note that the query performs a sequential scan over the entire table in order to find matching records. We can likely optimize the query by using an index.

```
Filter: (to_tsvector(COALESCE(content, ''::text)) @@ plainto_tsquery('football'::text))
```

Additionally the query normalizes the `content` column from text to a tsvector using `to_tsvector` in order to perform the full text search.

```
WHERE to_tsvector(COALESCE("web_page"."content", '')) @@ (plainto_tsquery('football')) = true
```

The `tsvector` type is a tokenized version of our text which normalizes the search column (more on tokenization here). Postgres needs to perform this normalization for every row and each row contains an entire Wikipedia page. This is both a CPU intensive and slow operation.

Specialized search column and gin indexes

In order avoid on-the-fly casting of text to tsvectors we'll create a specialized column which is used only for search. The column should be populated on inserts or updates. When querying is performed we'll avoid the performance penalty of casting types.

Since we can now have a tsvector type we are also able to add a gin index to speed up the query. The gin index ensures that the search will be performed with a indexed scan instead of a sequential scan over all records.

Open our `web/models.py` file and make modifications to the Page model.

```
from django.db import models
from django.contrib.postgres.search import SearchVectorField
from django.contrib.postgres.indexes import GinIndex

class Page(models.Model):
    title = models.CharField(max_length=100, unique=True)
    content = models.TextField()

    # New modifications. A field and an index
    content_search = SearchVectorField(null=True)

    class Meta:
        indexes = [GinIndex(fields=["content_search"])]
```

Run the migrations

```
Migrations for 'web':
  web/migrations/0002_auto_20190525_0647.py
    - Add field content_search to page
    - Create index web_page_content_505071_gin on field(s) content_search of model page
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, web
Running migrations:
  Applying web.0002_auto_20190525_0647... OK
```

Postgres triggers

Theoretically our problem is now solved. We have a Gin indexed column which should perform well when we search on it, but by doing so we have introduced another problem: the optimized `content_search` column needs to be kept manually in sync and updated whenever the `content` column updates.

Luckily for us Postgres provides an additional feature which solves this problem, namely triggers. Triggers are Postgres functions that fire when a specific action is performed on a row. We will create a trigger that populates `content_search` whenever a `content` row is created or updated. That way Postgres will keep the two columns in sync without us having to write any Python code.

In order to add a trigger we need to craft a manual Django migration. This will add the trigger function and update all of our Pages rows to ensure the trigger is fired and `content_search` is populated at migration time for our existing records. If you have an extremely large data set you might not want to do this in production.

Add a new migration in `web/migrations/0003_create_text_search_trigger.py`. Make sure to modify the previous migration in `dependencies` because the previously autogenerated migration is likely different for you.

```
from django.db import migrations

class Migration(migrations.Migration):

    dependencies = [
        # NOTE: The previous migration probably looks different for you, so
        # modify this.
        ('web', '0002_auto_20190524_0957'),
    ]

    migration = '''
        CREATE TRIGGER content_search_update BEFORE INSERT OR UPDATE
        ON web_page FOR EACH ROW EXECUTE FUNCTION
        tsvector_update_trigger(content_search, 'pg_catalog.english', content);

        -- Force triggers to run and populate the text_search column.
        UPDATE web_page set ID = ID;
    '''


    reverse_migration = '''
        DROP TRIGGER content_search ON web_page;
    '''

    operations = [
        migrations.RunSQL(migration, reverse_migration)
    ]
```

Run the migrations

```
$ ./manage.py makemigrations && ./manage.py migrate
No changes detected
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, web
Running migrations:
  Applying web.0003_create_text_search_trigger... OK
```

Measuring the performance improvement

Finally on to the fun part, let's verify that are query performs faster than before. Open a Django shell again, but when filtering the rows use the indexed `content_search` column rather than the normal `content` column.

```
./manage.py shell_plus --print-sql

>>> from django.contrib.postgres.search import SearchQuery
>>> Page.objects.filter(content_search=SearchQuery('football', config='english'))
SELECT t.oid,
       typarray
  FROM pg_type t
  JOIN pg_namespace ns
    ON typnamespace = ns.oid
 WHERE typname = 'hstore'

Execution time: 0.000829s [Database: default]

SELECT typarray
  FROM pg_type
 WHERE typname = 'citext'

Execution time: 0.000310s [Database: default]

SELECT "web_page"."id",
       "web_page"."title",
       "web_page"."content",
       "web_page"."content_search"
  FROM "web_page"
 WHERE "web_page"."content_search" @@ (plainto_tsquery('english'::regconfig, 'football')) = true
 LIMIT 21

Execution time: 0.001359s [Database: default]

<QuerySet []>
```

Query execution time down from 0.220s to 0.001s!

Let's analyze the query again to see how Postgres executes it. Copy the query from above and run it through `EXPLAIN ANALYZE`.

```
./manage.py dbshell

my_db=# explain analyze SELECT "web_page"."id",
my_db-#        "web_page"."title",
my_db-#        "web_page"."content",
my_db-#        "web_page"."content_search"
my_db-#   FROM "web_page"
my_db-#  WHERE "web_page"."content_search" @@ (plainto_tsquery('english'::regconfig, 'football')) = true
my_db-#  LIMIT 21
my_db-# ;
                                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
----------------------
 Limit  (cost=8.01..12.02 rows=1 width=675) (actual time=0.022..0.022 rows=0 loops=1)
   ->  Bitmap Heap Scan on web_page  (cost=8.01..12.02 rows=1 width=675) (actual time=0.020..0.020 rows=0 loops=1)
         Recheck Cond: (content_search @@ '''football'''::tsquery)
         ->  Bitmap Index Scan on web_page_content_505071_gin  (cost=0.00..8.01 rows=1 width=0) (actual
time=0.017..0.017 rows=0 loops=1)
               Index Cond: (content_search @@ '''football'''::tsquery)
 Planning Time: 3.061 ms
 Execution Time: 0.165 ms
(7 rows)
```

Here are the interesting bits:

```
->  Bitmap Index Scan on web_page_content_505071_gin  (cost=0.00..8.01 rows=1 width=0) (actual time=0.017..0.017
rows=0 loops=1)
```

Instead of a sequential scan Postgres uses an index on the `content_search` column.

```
Index Cond: (content_search @@ '''football'''::tsquery)
```

We also no longer perform an expensive `to_tsquery` operation for each row and instead use the `content_search` column as is.

Drawbacks

Unfortunately there are tradeoffs when using this optimization technique.

- Because we're maintaining another column of our text for the sole purpose of search speed our table size takes up significantly more space. Additionally the gin index on the `content_search` column takes up space on it's own.
- Since the search column is updated on every UPDATE or INSERT it also slows down writes to the database.

If you're constrained by memory and disk or need quick writes this technique may not be suitable for your use case. However I suspect that the majority of CRUD apps out there are OK with sacrificing disk and write speed for lightning fast search.

Conclusion

Postgres offers excellent full text search capability, but it's a little slow out of the box. In order to speed up text searches we add a secondary column of type tsvector which is a search-optimized version of our text.

We add a Gin index on the search column to ensure Postgres performs an index scan rather than a sequential scan. This reduces the query execution time by an order of magnitude.

In order to keep the text column and the search column in sync we use a Postgres trigger which populates the search column on any modifications to our text column.

The full code example can be found at [Github].