Postgres Full-Text Search With Django

May 3, 2017

Image courtesy Pexels

Django has added support for Postgres's built-in full-text searching as of version 1.10. This is a great alternative to a heavier search system such as elasticsearch or SOLR, when we want decent search capabilities without having to setup and maintain another service. Postgres's full-text search is Good Enough for a lot of use cases.

With this quick guide, I'll show how to add full-text searching to a Django application. Simple use-cases are covered pretty well in the Django documentation, so I will focus on a slightly more advanced example that allows searching through multiple fields, including data through relationships, weighting fields differently, adding an index for speed, and methods to keep the search data up-to-date.

It should go without saying that this is focused on Django with a PostgreSQL database back-end. This will not work with SQLite or MySQL. I also assume familiarity with Django and a basic understanding of Postgres.

An example project is available on github that follows this guide.

Models

We will use these models as an example. It's simple data for a Blog-like application consisting of Posts with data both directly included and referenced through relationships. But most importantly, we have data we want to search through that exists as ManyToOne (author) and ManyToMany (tags)relationships.

```
class Author(models.Model):
    name = models.CharField(max_length=50)


class Tag(models.Model):
    name = models.CharField(max_length=20)


class Post(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField()
    author = models.ForeignKey(Author)
    tags = models.ManyToManyField(Tag)
```

We will use this sample data:

```
jim = Author.objects.create(name="Jim Blogwriter")
nancy = Author.objects.create(name="Nancy Blogaday")

databases = Tag.objects.create(name="Databases")
programming = Tag.objects.create(name="Programming")
python = Tag.objects.create(name="Python")
postgres = Tag.objects.create(name="Postgres")
django = Tag.objects.create(name="Django")

django_post = Post.objects.create(
    title="Django, the western character",
    content="Django is a character who appears in a number of spaghetti "
            "western films.",
    author=jim
)
django_post.tags.add(django)

python_post = Post.objects.create(
    title="Python is a programming language",
    content="Python is a programming language created by Guido van Rossum "
            "and first released in 1991. Django is written in Python. Python "
            "can connect to databases.",
    author=nancy
)
```

```
python_post.tags.add(django, programming, python)

postgres_post = Post.objects.create(
    title="What is Postgres",
    content="PostgreSQL, commonly Postgres, is an open-source, "
            "object-relational database (ORDBMS).",
    author=nancy
)
postgres_post.tags.add(databases, postgres)
```

Building Documents

The first step is going to be building *Documents* for our posts. Each Document will be a logical representation of a single post, including:

- title
- content
- Author's name
- All tag names

Here is an example Django query:

```
from django.db.models.functions import Concat
from django.db.models import TextField, Value as V
from django.contrib.postgres.aggregates import StringAgg

document=Concat(
    'title', V(' '),
    'content', V(' '),
    'author__name', V(' '),
    StringAgg('tags__name', delimiter=' '),
    output_field=TextField()
)
Post.objects.annotate(document=document).values_list('document', flat=True)
```

```
<QuerySet [
  "Django, the western character Django is a character who appears in a
    number of spaghetti western films. Jim Blogwriter Django",
  "Python is a programming language Python is a programming language
    created by Guido van Rossum and first released in 1991. Django is
    written in Python. Python can connect to databases. Nancy Blogaday
    Python Django Programming",
  "What is Postgres PostgreSQL, commonly Postgres, is an open-source,
    object-relational database (ORDBMS). Nancy Blogaday Postgres Databases"
]>
```

This includes all of our data for each post, separated by spaces.

Search Vectors

So now that we have our documents, we need to convert them into a format that Postgres can index and search through. Postgres calls these Vectors. Django has a class to encapsulate this functionality called SearchVector. A `SearchVector` can also take a weight, so we'll rewrite our query to create vectors.

```
from django.contrib.postgres.search import SearchVector
from django.contrib.postgres.aggregates import StringAgg

vector=SearchVector('title', weight='A') + \
       SearchVector('content', weight='C') + \
       SearchVector('author__name', weight='B') + \
       SearchVector(StringAgg('tags__name', delimiter=' '), weight='B')
Post.objects.annotate(document=vector).values_list('document', flat=True)
```

```
<QuerySet [
  "'appear':10C 'blogwrit':19B 'charact':4A,8C 'django':1A,5C,20B 'film':17C
    'jim':18B 'number':13C 'spaghetti':15C 'western':3A,16C",
  "'1991':20C 'blogaday':32B 'connect':28C 'creat':11C 'databas':30C
    'django':21C,34B 'first':17C 'guido':13C 'languag':5A,10C 'nanci':31B
    'program':4A,9C,35B 'python':1A,6C,25C,26C,33B 'releas':18C
    'rossum':15C 'van':14C 'written':23C",
  "'blogaday':18B 'common':5C 'databas':15C,20B 'nanci':17B 'object':13C
    'object-rel':12C 'open':10C 'open-sourc':9C 'ordbm':16C
    'postgr':3A,6C,19B 'postgresql':4C 'relat':14C 'sourc':11C"
]>
```

Each document has been normalized down to a common set of *word stems*. This includes dropping case, dropping common prefixes and suffixes (like 's', and 'es'), and removing common words like 'a', 'an' and 'the'. The numbers are where in the document it found that stem, and the letter is the weight. If we want to override the configuration setting for how Postgres processes the words, for example to specify a different language, then we can pass an optional `config` parameter to SearchVector. If not given, then it will use whatever the database is configured to use by default, most likely based on the configured locale.

Performing a search

So now that we have our documents, we can perform a search. The easiest way to do this is to filter on our documents.

```
vector=SearchVector('title', weight='A') + \
       SearchVector('content', weight='C') + \
       SearchVector('author__name', weight='B') + \
       SearchVector(StringAgg('tags__name', delimiter=' '), weight='B')
Post.objects.annotate(document=vector).filter(document='django')
```

```
<QuerySet [<Post: Django, the western character>,
           <Post: Python is a programming language>]>
```

By default, django will use the `plainto_tsquery()` function in Postgres to parse the query. The short of this is that it will search for documents that match all of the words. However, instead of a string we can pass a SearchQuery() instance, which can be combined with several boolean operators.

```
from django.contrib.postgres.search import SearchQuery

query = SearchQuery('django') & SearchQuery('program')
Post.objects.annotate(document=vector).filter(document=query)
```

If we use a custom `config` with SearchVector(), then we should use that same `config` with SearchQuery().

Ranking

Results are most useful if we can rank them, taking into consideration the different weights we've assigned to each part of the document. Django provides the SearchRank class for this purpose.

```
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank
from django.contrib.postgres.aggregates import StringAgg

vector=SearchVector('title', weight='A') + \
       SearchVector('content', weight='C') + \
       SearchVector('author__name', weight='B') + \
       SearchVector(StringAgg('tags__name', delimiter=' '), weight='B')
query = SearchQuery('django')
Post.objects\
    .annotate(document=vector, rank=SearchRank(vector, query))\
    .filter(document=query)\
    .order_by('-rank')\
    .values_list('title', 'rank')
```

```
<QuerySet [
  ('Django, the western character', 0.665342),
  ('Python is a programming language', 0.364756)
]>
```

This gives us the functionality we want, but this isn't the best way to go about it if we care about performance. Every time we run this query the database has to build all the documents, for all rows in the table before it can search and rank them. It's fine for a few rows, but after more than a couple hundred it's going to start slowing down to unacceptable speeds. If our documents only contained data from a single table, we could just build a GIN index, but that won't work in this case since we pull extra data from other tables. So what we really want to do is to pre-calculate all of the documents and store them in the database.

Storing Vectors with SearchVectorField

Django provides a special field that allows us to store pre-calculated vectors in a field called `SearchVectorField`. We'll add that field to our Post model:

```
from django.contrib.postgres.search import SearchVectorField

class Post(models.Model):
    ...
    search_vector = SearchVectorField(null=True)
```

Then we'll migrate this to add the field.

```
./manage.py makemigrations
./manage.py migrate
```

Let's update this field manually for now.

```
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank
from django.contrib.postgres.aggregates import StringAgg

vector=SearchVector('title', weight='A') + \
       SearchVector('content', weight='C') + \
       SearchVector('author__name', weight='B') + \
       SearchVector(StringAgg('tags__name', delimiter=' '), weight='B')
for post in Post.objects.annotate(document=vector):
    post.search_vector = post.document
    post.save(update_fields=['search_vector'])
```

Note: This will issue an UPDATE for each row in the table, which will take forever if our table has a lot of rows. If we are only including fields from a single model in our document, then something like this would be much more efficient:

```
vector=SearchVector('title', weight='A') + \
       SearchVector('content', weight='C')
```

```
Post.objects.update(search_vector=vector)
```

Django doesn't allow us to use aggregate functions with an update clause, but Postgres does, so if we really wanted, we could issue a query like this to refresh all of the documents at once:

```
UPDATE blog_post
SET search_vector = document.vector
FROM (
     SELECT post.id,
            setweight(to_tsvector(post.title), 'A') ||
            setweight(to_tsvector(post.content), 'C') ||
            setweight(to_tsvector(author.name), 'B') ||
            setweight(to_tsvector(COALESCE(string_agg(tag.name, ', '), '')), 'B')
              AS vector
     FROM blog_post AS post
     JOIN blog_author AS author ON author.id = post.author_id
     JOIN blog_post_tags AS post_tags ON post_tags.post_id = post.id
     JOIN blog_tag AS tag ON tag.id = post_tags.tag_id
     GROUP BY post.id, author.id
   ) AS document
WHERE blog_post.id = document.id;
```

Searching on search_vector

Now that we have our documents stored, we can easily search on them

```
from django.db.models import F
from django.contrib.postgres.search import SearchVector, SearchQuery, SearchRank

query = SearchQuery('django')
Post.objects.annotate(rank=SearchRank(F('search_vector'), query))\
    .filter(search_vector=query).order_by('-rank').values_list('title', 'rank')
```

```
<QuerySet [
  ('Django, the western character', 0.665342),
  ('Python is a programming language', 0.364756)
]>
```

Indexing

Now that the documents are stored in a field, we can create a GIN index to speed up searching. With Django 1.11, this is as easy as adding an `indexes` Meta option to our model, then create and apply a migration.

```
from django.contrib.postgres.indexes import GinIndex

class Post(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField()
    author = models.ForeignKey(Author)
    tags = models.ManyToManyField(Tag)
    search_vector = SearchVectorField(null=True)

    class Meta:
        indexes = [
            GinIndex(fields=['search_vector'])
        ]
```

For Django 1.10, we'd need to create an empty migration then add a `RunSQL` operation:

```
migrations.RunSQL(
    "CREATE INDEX blog_post_search_vector_idx ON blog_post USING gin(search_vector)",
    "DROP INDEX blog_post_search_vector_idx"
)
```

Updating the documents

This is great and all, but as soon as any of the data changes, the documents will be out of date, and the search results will be incorrect. The first way we could address this is with a cron or scheduled task that updates the whole table (like we did above) at regular intervals. This would be a good option for an application that processes a lot of updates, or does its updates in large batches. This way we don't add overhead to every update, and instead can more efficiently update all rows at once.

For other applications that have a slow steady stream of updates, it would be more appropriate to update every time the data changed. The upside is that the search data will always be up to date. The downside is that every update will incur extra overhead as the search_vector is calculated.

A compromise would be to queue up the update to the search_vector as an asynchronous process, so that it's updated fairly quickly, but the updates can still be done as a batch. This is outside the scope of this article, but it shouldn't be too difficult depending on the application architecture.

The best method is going to depend on the particular application. Here are a couple simple ways to update on every save.

Overriding save()

One method to update the document is to override the save() method on Post. With this method, every time the data that search relies on is updated, then the search_vector is also updated alongside it. So search results reflect data changes immediately. This does incur an overhead on every update to the database, though.

First we'll create a custom manager that adds documents to the queryset when we request it, so we can keep things DRY, and only have our SearchVectors defined in one place.

```
class PostManager(models.Manager):
    def with_documents(self):
        vector = SearchVector('title', weight='A') + \
                 SearchVector('content', weight='C') + \
                 SearchVector('author__name', weight='B') + \
                 SearchVector(StringAgg('tags__name', delimiter=' '), weight='B')
        return self.get_queryset().annotate(document=vector)
```

Now we'll update our Post model and add our custom manager and a custom save function. The idea here is to save the data to the database, then do a SELECT query that joins all the data together and then save the new search_vector. So each save will result in an UPDATE, SELECT, then another UPDATE.

```
from django.contrib.postgres.search import SearchVectorField, SearchVector

class Post(models.Model):
    title = models.CharField(max_length=50)
    content = models.TextField()
    author = models.ForeignKey(Author)
    tags = models.ManyToManyField(Tag)
    search_vector = SearchVectorField(null=True)

    objects = PostManager()

    def save(self, *args, **kwargs):
        super().save(*args, **kwargs)
        if 'update_fields' not in kwargs or 'search_vector' not in kwargs['update_fields
            instance = self._meta.default_manager.with_documents().get(pk=self.pk)
            instance.search_vector = instance.document
            instance.save(update_fields=['search_vector'])
```

Also, updates to authors and tags won't trigger the `save()`, so we'll also add signals for those that force a `save()` on the Post as well to update the search_vector.

```
from django.db.models.signals import post_save, m2m_changed
from django.dispatch import receiver

@receiver(post_save, sender=Author)
def author_changed(sender, instance, **kwargs):
    for post in instance.post_set.with_documents():
        post.search_vector = post.document
        post.save(update_fields=['search_vector'])


@receiver(m2m_changed, sender=Post.tags.through)
def post_tags_changed(sender, instance, action, **kwargs):
    if action in ('post_add', 'post_remove', 'post_clear'):
        instance.save()
```

Now any changes to a Post, Author or if tags are added, removed or cleared will cause the search data to be updated. If a tag is renamed, then we won't pick up on that without creating another signal handler.

Using triggers

It's also possible to install a few triggers in the database that will automatically update the search_vector whenever the data is changed. I won't go into too much detail, but they will look something like this. We can easily add these to a migration that uses a RunSQL operation to install them into our database. The idea is exactly the same as above, but since the database can do it all locally and not have to send data back and forth to Django, it will perform better.

```
-- Trigger on insert or update of blog.Post
CREATE OR REPLACE FUNCTION post_search_vector_trigger() RETURNS trigger AS $$
BEGIN
  SELECT setweight(to_tsvector(NEW.title), 'A') ||
         setweight(to_tsvector(NEW.content), 'C') ||
         setweight(to_tsvector(author.name), 'B') ||
         setweight(to_tsvector(COALESCE(string_agg(tag.name, ', '), '')), 'B')
  INTO NEW.search_vector
  FROM blog_post AS post
  JOIN blog_author AS author ON author.id = post.author_id
  JOIN blog_post_tags AS post_tags ON post_tags.post_id = post.id
  JOIN blog_tag AS tag ON tag.id = post_tags.tag_id
  WHERE post.id = NEW.id
  GROUP BY post.id, author.id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER search_vector_update BEFORE INSERT OR UPDATE ON blog_post
  FOR EACH ROW EXECUTE PROCEDURE post_search_vector_trigger();
```

```
-- Trigger after blog.Author is update
CREATE OR REPLACE FUNCTION author_search_vector_trigger() RETURNS trigger AS $$
BEGIN
  UPDATE blog_post SET id = id WHERE author_id = NEW.id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER search_vector_update AFTER INSERT OR UPDATE ON blog_author
  FOR EACH ROW EXECUTE PROCEDURE author_search_vector_trigger();

-- Trigger after blog.Post.tags are added, update or deleted
CREATE OR REPLACE FUNCTION tags_search_vector_trigger() RETURNS trigger AS $$
BEGIN
  IF (TG_OP = 'DELETE') THEN
    UPDATE blog_post SET id = id WHERE id = OLD.post_id;
    RETURN OLD;
  ELSE
    UPDATE blog_post SET id = id WHERE id = NEW.post_id;
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER search_vector_update AFTER INSERT OR UPDATE OR DELETE ON blog_post_tags
  FOR EACH ROW EXECUTE PROCEDURE tags_search_vector_trigger();
```

Conclusion

Now we have a working application using Postgres full-text search, and once set up, it can be mostly forgotten about. Compared to setting up elasticsearch or SOLR (even with Haystack), this is a breeze, and results are going to be good enough for most applications.

For more information and more features, such as language support, custom stemming, trigrams, accents, etc, please see the following sources:

- Official PostgreSQL Full-Text Search Documentation
- Official Django Postgres Search Documentation
- Postgres full-text search is Good Enough: Great article on the basics of Postgres full-text search.
- An example project that this blog post goes along with.

Share this:

Facebook Twitter Google+ Reddit Email

Comments

6 Comments Lotechnica JoshuaPK

Recommend 5 Share Sort by Best

Join the discussion…

Vlad Okhrimenko • 2 months ago

Great article, but I have a question: why we are creating the index and then dropping it?

Reply • Share ›

Nathan Shafer Mod → Vlad Okhrimenko • 2 months ago

If you're talking about the RunSQL migration, the first argument (the CREATE) runs when you apply the migration forward, the second argument (the DROP) runs if you roll it backwards. So this makes the migration work both ways.

Reply • Share ›

Vlad Okhrimenko → Nathan Shafer • 2 months ago

Thank you!

Reply • Share ›

SilentLennie • 4 months ago

I'm still hoping the Russian PostgreSQL hackers that are good with indexes and text search find the time/money to work on getting their patches incorporated into mainline PostgreSQL. A couple of years ago they showed at a conference that they could make PostgreSQL as fast or faster than those dedicated search tools. That would make it better than good enough.

Reply • Share ›

Flexic • 4 months ago

Great article, got search working just how I wanted on my piddly Django app :)

The options for updating the documents seem quite a bit larger than the simple method you showed (and I am using for a single table) in my search: Post.objects.update(search_vector=vector)

I guess that call will update on every record in the db every time it is called, vs your save overrides which will only update the row in question?

Reply • Share ›

Nathan Shafer Mod → Flexic • 4 months ago

Yeah, that updates all rows at once. It's fine if you have just a few rows, but if your table grows, it's going to start taking longer and longer. The other methods I showed, the save() and triggers, only update that single row when it's inserted into the DB, so it will scale better. Still not as good as offloading to an asynchronous process in the background, though.

• Reply • Share ›

ALSO ON **LOTECHNICA**

Fixing Lua indenting in Sublime Text

2 comments • 2 years ago•

Brandon Victor — Oh, I forgot to mention: thanks for putting this up in the first place.There was also another thing I added to this: It wasn't incrementing

Lotechnica

2 comments • 2 years ago•

travis mathis — I know this is an old post, but for anyone stumbling on it Bootstrap also has semantic markup using sass mixins if you use bootstrap-sass.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Privacy**

Pages

Home
Archives
Colophon

Categories

Graphics
Misc
Mobile
Web

Tags

android django docker elixir gideros ios lua math phoenix plug postgres pycharm responsive sublime-text web design

Nathan Shafer

Github
Google Plus
Normal Technologies LLC
Type-in Games