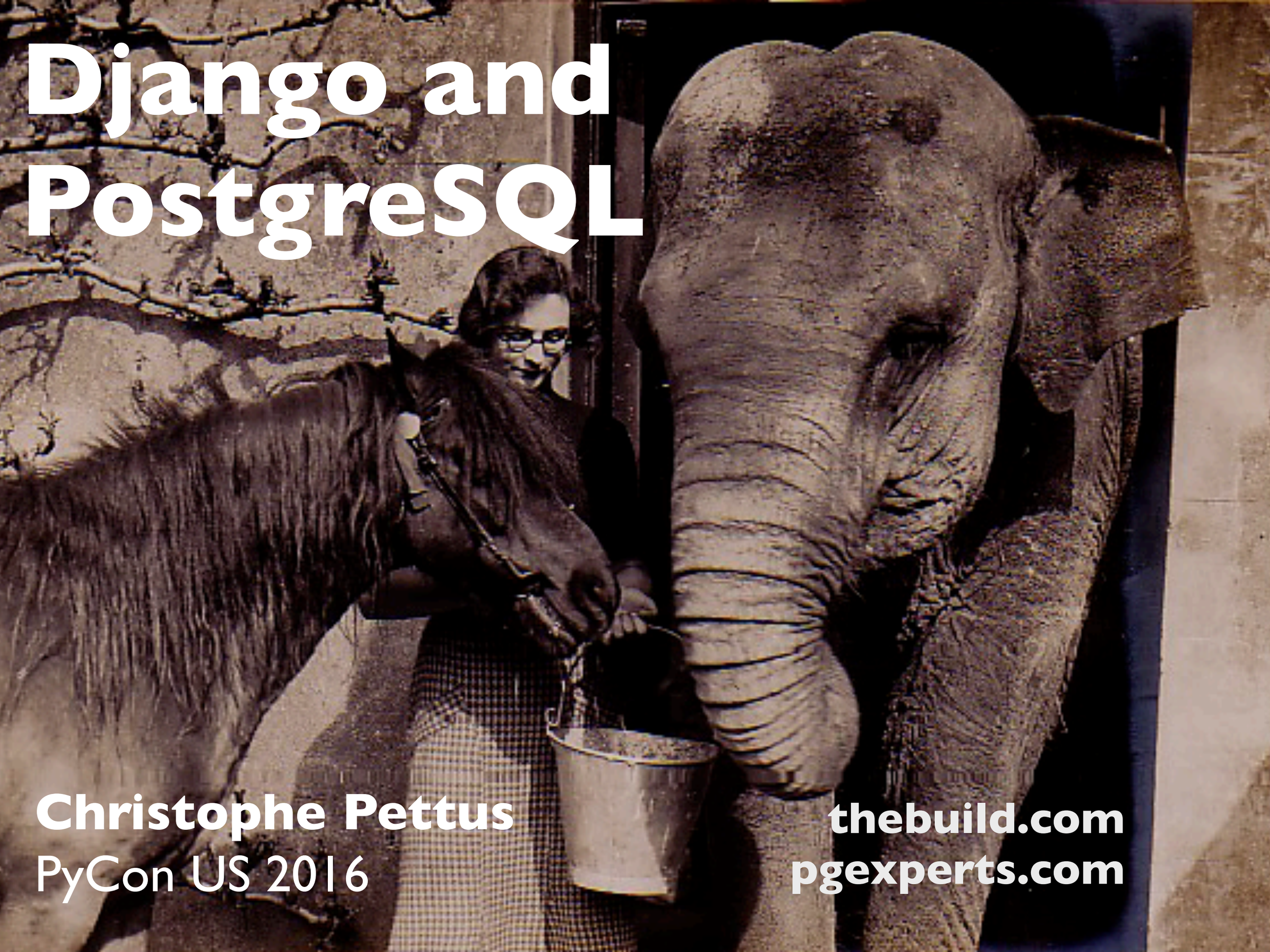
Django and PostgreSQL



Christophe Pettus
PyCon US 2016

thebuild.com
pgexperts.com

Welcome!

- Christophe Pettus
- CEO of PostgreSQL Experts, Inc.
- Based in sunny Alameda, California.
- Technical blog: **thebuild.com**
- Twitter: **@xof**
- **christophe.pettus@pgexperts.com**

PGX
POSTGRESQL
EXPERTS, INC.

So. Much. Stuff.

- Django 1.7 introduced native migrations.
- Django 1.8 introduced and 1.9 extended django.contrib.postgres,
- So many features, so little time!
- Work done by Marc Tamlyn, who deserves the praise of a grateful nation.

Migrations.

- Wait. Migrations aren't PostgreSQL-specific.
- They're an enabler for other features we'll talk about.
- And Django 1.7 migrations are just amazing.
- Thanks, Andrew!

A quick overview of migrations.

- Built around sequences of "operations."
- Each operation moves the database back or forward through its schema migration timeline.
- Lots of operations, but let's talk about:
  - RunSQL.
  - CreateExtension.

RunSQL

- Applies raw SQL directly to the database.
- Very useful for things that you can't do directly in the DB yet.
- Like creating indexes for the new types.

CreateExtension.

- Runs a CREATE EXTENSION command.
- hstore is an extension, and needs to be added before being used.
- This also adds a query to each connection.

New Field Types

- Array Field.
- Range Field.
- hstore Field.
- JSON Field.

Array Fields.

- Arrays are first-class types in PostgreSQL.
- ArrayField allows you use them directly.
- Maps into Python lists.
- PostgreSQL arrays are of homogeneous type, unlike Python lists.

PostgreSQL Arrays != Python Lists

- PostgreSQL arrays are of homogeneous types.
- PostgreSQL multidimensional arrays are rectangular (although individual entries can be NULL).

Array Field Queries: __contains

- Matches if the array field on the left contains all of the entries of the list on the right.
  - [‘a’,’b’,’c’] contains [‘a’,’b’] but
  - [‘a’,‘b’,’c’] does not contain [‘a’,‘d’]
- Order is not important in __contains

Array Field Queries: __contained_by

- Matches if the list on the right contains all of the entries of the field on the left.
  - [‘a’,’b’,’c’] is not contained by [‘a’,’b’] and
  - [‘a’,‘b’,’c’] is not contained by [‘a’, ‘d’] but
  - [‘a’,’b’] is contained by [‘a’,’b’,’c’]
- Order is not important here, either.

PGX
POSTGRESQL
EXPERTS, INC.

Array Field Queries: __overlaps

- Matches if the array field on the left contains any of the entries of the list on the right.
- [‘a’,‘b’,’c’] overlaps [‘a’, ‘d’] but
- [‘a’,’b’,’c’] does not overlap [‘d’]

Array Field Queries: __len

- Returns the length of the field on the left as an integer.
  - ['a','b'] _len == 2
- (Note: Unless you've created an expression index, does a full table scan.)

Array Field Transforms: Index

- Query the first element of an array:
  - .filter(array_field__0 = ‘a’)
- If there is no entry ‘n’, does not match rather than an error.
- You can’t specify the index programmatically in this syntax.
  - … except with kwargs, of course.

Array Field Transforms: Slices

- Slices also work!
  - .filter(array_field_0_1=['a','b'])
  - .filter(array_field_0_2__contains=['a'])

Indexing Array Fields.

- So, just specify db_index=True and you're set right?
- Wrong.
- This creates a b-tree index, which is pretty useless for array (and other non-scalar) types.

Sidebar: PostgreSQL Indexing.

- PostgreSQL supports multiple types of index.
- Most people are familiar with btree indexes; that's what you get with db_index=True
- btree indexes are fast, compact, and provide total ordering.

Great, but not perfect.

- btree indexes require a totally ordered type.
- Some types (points, arrays, hstore, etc.) don't have total ordering, but do have other operations (inclusion, key containment).
- For those, we have GIST and GIN indexes.

GIST vs GIN.

- GIN indexes are used for types that contain keys and values (arrays, hstore, jsonb).
- GIST indexes are used for types that partition a mathematical space (points, ranges).
- They “just work” once created.

Indexing Array Fields.

- Arrays support GIN (Generalized INverted indexes).
- Accelerates __contains, __contained_by, __overlaps.
- Does not help __len or slice operations.

Indexing Array Fields.

- `CREATE INDEX ON app_model USING GIN (field);`
- GIN indexes can be large (if there's a lot of data in the underlying table).
- They're not free to update.
- Don't create one unless you need it.
  - Small tables generally don't.

Indexing Array Fields.

- Indexing len:
  - `CREATE INDEX ON app_model ((array_length(field,1)));`
- Indexing slices:
  - `CREATE INDEX ON app_model ((field[1:2]));`
  - PostgreSQL arrays are 1-based.

Why use Array Fields?

- Underlying data is actually an array.
- Replacement for a many-to-many table.
- A denormalization to store results of an expensive query (proceed with caution here!).

hstore Fields.

- hstore is a semi-built-in "hash" data store.
- Like a dict that can only take strings as keys and values.
- Only way of storing unstructured data in PostgreSQL pre-JSON.

Getting hstore to work.

- Has to be installed in a PostgreSQL database (it's not part of core).
- django.contrib.postgres comes with a HStoreExtension migration operation to install it for you.
- Each connection must do a query to get the hstore type's OID. Usually not a big deal.

About hstore.

- Represented in Python as a dict.
- Keys and values must be strings.
- Translated to and from the database encoding.
  - Which is UTF-8… right?

hstore Field Queries.

- Supports __contains and __contained_by.
- Both key and value must match.
- __has_key matches fields containing a particular key.
- __has_keys matches fields containing all of the keys (takes a list).
- __has_any_keys matches fields with any of the keys (takes a list).

hstore Field Queries.

- __keys matches the list of the keys of the field. Generally used with other transforms:
  - .filter(field__keys__overlaps=['a','b'])
- __values does the same for the values of the hstore field.

Indexing hstore fields.

- hstore fields support GIN indexes as well:
  - `CREATE INDEX ON app_model USING GIN(field);`
- Accelerates __contains, __has_key, __has_keys (but not __contained_by).

Why use hstore fields?

- Great for storing very rare attributes.
  - If there are multiple fields that are NULL 95% of the time, consider an hstore field instead.
  - Although remember that NULL fields take zero space in PostgreSQL.
- User-defined attributes.

Why use hstore fields?

- In greenfield development, largely superseded by JSON.
  - About which more later.
- But no JSON field type in 1.8, so maybe hstore if you need it right away.
- Existing databases with hstore.

Range Fields.

- PostgreSQL has native range types!
- Range types span a range of a scalar type:
- For example, [1,8] as an int4range includes 1, 2, 3, 4, 5, 6, 7, 8.
- Bounds can be exclusive: [1,8) includes 1, 2, 3, 4, 5, 6, 7.
  - [) is the default.

To infinity and beyond!

- You can omit a bound to indicate "all values less/than greater than."
- Some types (for example, dates) also have a special "infinity" value.
- psycopg2 includes a Python "Range" base type that handles the various boundary values, and the infinity special cases.

Types of Ranges.

- Out of the box, Django 1.8 supports:
  - IntegerRange and BigIntegerRange.
  - FloatRange.
  - DateTimeRange.
  - DateRange.

Range Field Queries.

- __contains, __contained_by, __overlap work the way you'd expect.
- __fully_lt, __fully_gt is true if both the lower and upper bounds of the field are less/greater than the comparison value.
- __adjacent_to is true if the field and the comparison value share a boundary.

Range Field Queries.

- __not_lt is true if the field does not contain any points less than the comparison value.
- __not_gt is true if the field does not contain any points greater than the comparison value.

Indexing Range Fields.

- Range fields support GIST (General Index Storage Technique) indexes.
  - `CREATE INDEX ON app_model USING GIST(field);`
- Accelerates all of the comparison operations listed, woo-hoo!

A Hard Problem.

- "Don't allow two bookings to be inserted into the database for the same room where the dates overlap."
- There's no way to express this using traditional UNIQUE constraints.
- Constraint Exclusion to the rescue!

Constraint Exclusion.

- A generalization of the idea of UNIQUE indexes.
- "Don't allow two equal entries, based on this set of comparison operations, into the table."
- The operations can be any index-supported boolean predicate; they're ANDed together.

One Catch.

- It has to be a single index.
- Since RANGE types require a GIST index…

- The index has to be a GIST index.
- By default, simple scalar values don't have GIST indexing. Uh, oh.

btree_gist to the rescue!

- Allows the creation of GIST indexes on (most) simple scalar types.
- PostgreSQL extension, part of contrib.
- Has to be installed in the database, but:
- Ships with PostgreSQL.
- Use the CreateExtension migration operation.

PGX
POSTGRESQL
EXPERTS, INC.

How would we use this?

```
from django.db import models
from django.contrib.postgres.fields import DateRangeField


class Booking(models.Model):
    room = models.CharField(max_length=4)
    dates = DateRangeField()
```

Which gives us...

```
xof=# \d reservations_booking
                                    Table
"public.reservations_booking"
 Column |          Type          |
Modifiers
--------+------------------------
+-----------------------------------------
----------
 id     | integer                | not null default
nextval('reservations_booking_id_seq'::regclass)
 room   | character varying(4) | not null
 dates  | daterange              | not null
Indexes:
    "reservations_booking_pkey" PRIMARY KEY, btree (id)
```

And add constraint index...

```
xof=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
xof=# ALTER TABLE public.reservations_booking ADD EXCLUDE
USING gist ( room WITH =, dates WITH && );
ALTER TABLE
```

PGX
POSTGRESQL
EXPERTS, INC.

And profit!

```
>>> Booking(room='123', dates=DateRange(date(2015,9,1),
date(2015,9,2))).save()
>>> Booking(room='123', dates=DateRange(date(2015,9,2),
date(2015,9,7))).save()
>>> Booking(room='127', dates=DateRange(date(2015,9,2),
date(2015,9,7))).save()
>>> Booking(room='123', dates=DateRange(date(2015,9,5),
date(2015,9,9))).save()
(blah blah blah)
IntegrityError: conflicting key value violates exclusion
constraint "reservations_booking_room_dates_excl"
DETAIL:  Key (room, dates)=(123, [2015-09-05,2015-09-09))
conflicts with existing key (room, dates)=(123,
[2015-09-02,2015-09-07)).
```

PGX
POSTGRESQL
EXPERTS, INC.

Why use range fields?

- To represent ranges.
  - You probably figured that one out.
- More natural than the traditional (lo, hi) field pair.
- More database integrity and interesting operations available.

JSON Fields.

- New in 1.9.
- Fields that support arbitrary JSON structures.
- Stored as jsonb in PostgreSQL.

JSON Field Queries.

- Supports __contains and __contained_by.
- Both key and value must match.
- __has_key matches fields containing a particular key.
- __has_keys matches fields containing all of the keys (takes a list).
- __has_any_keys matches fields with any of the keys (takes a list).

JSON Field Queries.

- Can do path-type queries:
  - `Dog.objects.filter(data__owner__name='Bob')`
- Can use array indexes:
  - `Dog.objects.filter(data__owner__other_pets__0__name='Fishy')`

JSON vs JSONB.

- PostgreSQL has two JSON types: json and jsonb.
- json stores the raw text of the json blob, whitespace and all.
- jsonb is a compact, indexable representation.

Why use json instead of jsonb?

- json (vs jsonb) is faster to insert, since it doesn't have to process the data.
- json allows for two highly dubious "features" (duplicate object keys, stable object key order).
- OK if you are just logging json that you don't plan to extensively query.

Why use jsonb instead of json?

- All other applications want jsonb.
- jsonb can be indexed in useful ways, unlike json.
- The JSONField field type uses jsonb, so just roll with it.

Indexing JSON.

- jsonb has GIN indexing.
- Default type supports queries with the @>, ?, ?& and ?| operators.
- The query must be against the top-level object for the index to be useful.
- Can query nested objects, but only in paths rooted at the top level.

Why use JSON?

- Logging JSON data.
- Audit tables that work across multiple schemae.
- A friendly way of pickling Python objects.
- User-defined attributes and rare fields, a la hstore.

Other goodies.

- Admin widgets to go with many of the new types.
- hstore and JSON widgets are really only good for debugging.

PGX
POSTGRESQL
EXPERTS, INC.

Full-Text Search!

- PostgreSQL has had integrated full-text search for a long time.
- Django 1.10 (currently in alpha) contains model-level support for it.
- First, some concepts…

tsvector

- A "tsvector" is a block of text (blog entry, journal article) encoded for full-text search.
- Built-in PostgreSQL type.
- Turning text into a tsvector requires a "configuration," which includes things like language, stemming algorithm, list of stop words.

Full-Text Configurations

- PostgreSQL has a bunch of built-in configurations.
- Generally, just use those; making your own configuration is extra-for-experts.
- ‘english’ is a good example configuration.

to_tsvector

- Function that takes text, returns a tsvector.
- `to_tsvector('english', my_big_text_field)`
- Django calls this under the hood for you (in most situations).

__search

- Searches a text field using full-text searching.
- Calls to_tsvector for you.
- Without indexes, does a sequential scan.
- `Entry.objects.filter(body_text__search='Cheese')`

SearchVector

- Represents a tsvector object in Django.
- Lets you search more than one field at a time by combining them into a single tsvector.
- Merging and searching happen as the query runs (which can be expensive).

SearchQuery

- Represents a PostgreSQL tsquery object.
- Translates words into queries, using stemming, etc.
- Can be combined to form boolean predicates:
- `SearchQuery('potato') & SearchQuery('ireland')`
- `SearchQuery('potato') | SearchQuery('penguin')`

SearchRank

- Exposes the PostgreSQL ranking function.
- You can use it as annotation:
- `Entry.objects.annotate(rank=SearchRank(vector, query)).order_by('-rank')`

Lots more!

- Changing the rank weight of specific fields (weight title more heavily than body, etc.).
- Using different search configurations (different languages).

Indexing.

- For reasonable performance, you have to index fields.
- If you are searching on one field, in one language, a simple index will work:
- `CREATE INDEX ON t USING GIN (to_tsvector('english', f));`

Fancy Indexing.

- Combining multiple fields…
- … possibly with different weights.
- Create a separate tsvector field out of the fields.
- Django provides SearchVectorField for just this application.

Maintaining a SearchVectorField.

- Update it on each model instance change in the application (override save(), etc.).
- Use a trigger in PostgreSQL (recommended).
- See PostgreSQL documentation for details on building this trigger.

Bits and Bobs.

- PostgreSQL-specific aggregation functions (such as StringAgg).
- TransactionNow() function (returns time of transaction start).
- The unaccent filter which you can read about in the documentation.

Questions?

PGX

POSTGRESQL

EXPERTS, INC.

Thank you!

Christophe Pettus
thebuild.com
cpettus@pgexperts.com
@xof