Django Postgres Stats Documentation

Release 0.1.0

RTI Center for Data Science

Jun 27, 2017

Contents

Django Postgres Stats exposes statistical and datetime functions specific to Postgres to Django without making the user write raw SQL. The plan is to expand the library over time to cover many Postgres-specific functions. For now, only functions personally used by the authors have been added.

Contents:

CHAPTER 1

Installation instructions

Download and installation

Installation with pip

You can use pip to install django-postgres-stats:

```
$ pip install django-postgres-stats
```

Using in Django

You will need to add the *postgres_stats* application to the INSTALLED_APPS setting of your Django project *settings.py* file.:

```
INSTALLED_APPS = (
    ...
    'postgres_stats',
)
```

Version control

Django Postgres Stats is hosted on GitHub:

```
https://github.com/rtidatascience/django-postgres-stats
```

CHAPTER 2

Postgres functions

Database functions

You can use these like the standard Django database functions.

class `postgres_stats.functions.`**`DateTrunc`**(*expression*, *precision*, ***extra*)

Accepts a single timestamp field or expression and returns that timestamp truncated to the specified *precision*. This is useful for investigating time series.

The *precision* named parameter can take:

- microseconds
- milliseconds
- second
- minute
- hour
- day
- week
- month
- quarter
- year
- decade
- century
- millennium

Usage example:

```
checkin = Checkin.objects.
    annotate(day=DateTrunc('logged_at', 'day'),
             hour=DateTrunc('logged_at', 'hour')).
    get(pk=1)

assert checkin.logged_at == datetime(2015, 11, 1, 10, 45, 0)
assert checkin.day == datetime(2015, 11, 1, 0, 0, 0)
assert checkin.hour == datetime(2015, 11, 1, 10, 0, 0)
```

class postgres_stats.functions.**Extract**(*expression*, *subfield*, ***extra*)

Accepts a single timestamp or interval field or expression and returns the specified *subfield* of that expression. This is useful for grouping data.

The *subfield* named parameter can take:

- century
- day
- decade
- dow (day of week)
- doy (day of year)
- epoch (seconds since 1970-01-01 00:00:00 UTC)
- hour
- isodow
- isodoy
- isoyear
- microseconds
- millennium
- milliseconds
- minute
- month
- quarter
- second
- timezone
- timezone_hour
- timezone_minute
- week
- year

See the Postgres documentation for details about the subfields.

Usage example:

```
checkin = Checkin.objects.
    annotate(day=Extract('logged_at', 'day'),
             minute=Extract('logged_at', 'minute'),
             quarter=Extract('logged_at', 'quarter')).
```

```
    get(pk=1)

assert checkin.logged_at == datetime(2015, 11, 1, 10, 45, 0)
assert checkin.day == 1
assert checkin.minute == 45
assert checkin.quarter == 4
```

Aggregations

You can use these like the standard Django aggregations.

class postgres_stats.aggregates.**Percentile**(*expression*, *percentiles*, *continuous=True*, ***extra*)

Accepts a numerical field or expression and a list of fractions and returns values for each fraction given corresponding to that fraction in that expression.

If *continuous* is True (the default), the value will be interpolated between adjacent values if needed. Otherwise, the value will be the first input value whose position in the ordering equals or exceeds the specified fraction.

You will likely have to declare the *output_field* for your results. Django cannot guess what type of value will be returned.

Usage example:

```
from django.contrib.postgres.fields import ArrayField

numbers = [31, 83, 237, 250, 305, 314, 439, 500, 520, 526, 527, 533,
           540, 612, 831, 854, 857, 904, 928, 973]
for n in numbers:
    Number.objects.create(n=n)

results = Number.objects.all().aggregate(
    median=Percentile('n', 0.5, output_field=models.FloatField()))
assert results['median'] == 526.5

results = Number.objects.all().aggregate(
    quartiles=Percentile('n', [0.25, 0.5, 0.75],
                         output_field=ArrayField(models.FloatField())))
assert results['quartiles'] == [311.75, 526.5, 836.75]

results = Number.objects.all().aggregate(
    quartiles=Percentile('n', [0.25, 0.5, 0.75],
                         continuous=False,
                         output_field=ArrayField(models.FloatField())))
assert results['quartiles'] == [305, 526, 831]
```

CHAPTER 3

Roadmap

In general, we want to create the ability to use Postgres functions that will give us better statistical methods. If a function can be easily accessed using Func, we do not need to implement it.

Next functions

- mode
- width_bucket
- corr, covar, and regr functions

After we have the set of functions, we can begin to introduce other parts of Postgres that are not yet available in Django.

Possible additions

- OVERLAPS operator
- Range types
- Window functions

CHAPTER 4

Indices and tables

- genindex
- search

Index