django-denorm-iplweb Documentation

Release 0.5.0

Michał Pasternak

Jul 10, 2023

CONTENTS

django-denorm-iplweb provides a declarative way of denormalizing models in Django based applications while maintaining data consistency.

CHAPTER

ONE

CONTENTS

1.1 Tutorial

1.1.1 First steps

You must add denorm app to your INSTALLED_APPS within your settings.py:

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

You also need to run the initial migration for denorm app:

```
$ python manage.py migrate denorm
```

1.1.2 Counting related objects

Perhaps the most common use case for denormalization is to cache the number of objects associated with an object instance through a ForeignKey.

So let's say we are building a gallery application with models like this:

```
class Gallery(models.Model):
    name = models.TextField()

class Picture(models.Model):
    image = models.ImageField(...)
    gallery = models.ForeignKey(Gallery)
```

To calculate the number of pictures in a gallery, we would normally have a Django view with code similar to:

```
gallery = Gallery.objects.get(...)
number_of_pictures = gallery.picture_set.count()
```

However, this code will result in a COUNT query on the database for every time the page is viewed.

To speed this up we can cache the number of pictures inside the gallery:

```
from denorm import CountField

class Gallery(models.Model):
    name = models.TextField()
    picture_count = CountField('picture_set')

class Picture(models.Model):
    image = models.ImageField(...)
    gallery = models.ForeignKey(Gallery)
```

This will incrementally update the number when we add and delete related objects. Note that `CountField` updates are not lazy (like the callbacks described below), their value always gets updated immediately.

1.1.3 Creating denormalized fields using callback functions

A denormalized field can be created from a python function by using the `@denormalized` decorator. The decorator takes at least one argument: the database field type you want to use to store the computed value. Any additional arguments will be passed into the constructor of the specified field when it is actually created.

If you already use the `@property` decorator that comes with python to make your computed values accessible like attributes, you can often just replace `@property` with `@denormalized(..)` and you won't need to change any code outside your model.

Now whenever an instance of your model gets saved, the value stored in the field will get updated with whatever value the decorated function returns.

Example:

```
class SomeModel(models.Model):
    # the other fields

    @denormalized(models.CharField,max_length=100)
    def some_computation(self):
       # your code
       return some_value
```

in this example `SomeModel` will have a `CharField` named `some_computation`.

Note: You must add the column in the DB yourself (either manually or through a south migration) since denorm won't perform that operation for you.

Adding dependency information

The above example will only work correctly if the return value of the decorated function only depends on attributes of the same instance of the same model it belongs to.

If the value somehow depends on information stored in other models, it will get out of sync as those external information changes.

As this is a very undesirable effect, django-denorm-iplweb provides a mechanism to tell it what other model instances will effect the computed value. It provides additional decorators to attach this dependency information to the function before it gets turned into a field.

Depending on related models

In most cases your model probably contains a ForeignKey to some other model (forward foreign key relationship), an other model has a ForeignKey to the model containing the denormalized field (backward foreign key relationship) or the two models are connected through a ManyToManyField, and your function will somehow use the information in the related instance to compute its return value.

This kind of dependency can be expressed like this:

```
class SomeModel(models.Model):
    # the other fields
    other = models.ForeignKey('SomeOtherModel')

    @denormalized(models.CharField,max_length=100)
    @depend_on_related('SomeOtherModel')
    def some_computation(self):
       # your code
       return some_value
```

The `@depend_on_related` decorator takes the related model as an argument in the same was `ForeignKey` does, so you can use the same conventions here. `@depend_on_related` will then detect what kind (forward/backward/m2m) of relationship the two models have and update the value whenever the related instance of the other model changes.

In case of an ambiguous relationship (maybe there are multiple foreign keys to the related model) an error will be raised, and you'll need to specify the name of the ForeignKey to use like this:

```
...
    @depend_on_related('SomeOtherModel',foreign_key='other')
...
```

If this still is not enough information for django-denorm-iplweb to pick the right relation, there is probably a recursive dependency (on `self`). In that you also need to specify the direction of the relation:

```
...
    @depend_on_related('self',type='forward')
...
```

Denormalizing ForeignKeys

If you wish to denormalize a ForeignKey (for example to cache a relationship that is through another model), then your computation should return the primary key of the related model. For example:

```
class SomeOtherModel(models.Model):
    third_model = models.ForeignKey('ThirdModel')

class SomeModel(models.Model):
    # the other fields
    other = models.ForeignKey('SomeOtherModel')

   @denormalized(models.ForeignKey,to='ThirdModel',blank=True, null=True)
   @depend_on_related('SomeOtherModel')
   def third_model(self):
       return self.other.third_model.pk
```

Callbacks are lazy

Your fields won't get updated immediately after making changes to some data. Instead potentially affected rows are marked as dirty in a special table and the update will be done by the `denorm.flush` method.

Post-request flushing

The easiest way to call `denorm.flush` is to simply do it after every completed request. This can be accomplished by adding `DenormMiddleware` to `MIDDLEWARE_CLASSES` in your `settings.py`:

```
MIDDLEWARE_CLASSES = (
...
    'django.middleware.transaction.TransactionMiddleware',
    'denorm.middleware.DenormMiddleware',
...
)
```

As shown in the example, I recommend to place `DenormMiddleware` right after `TransactionMiddleware`.

Using the queue

If the above solution causes problem like slowing the webserver down because `denorm.flush` takes to much time to complete, you can use a background process to update the data. The process will check for dirty rows as it is being run and then it will re-check as soon as it gets a NOTIFY signal from the database server.

To run the process, which waits for NOTIFY command from PostgreSQL server, run:

```
./manage.py denorm_queue
```

The command runs in foreground. If you need to daemonize it, use specialized tools like supervisord

It could be tempting to run multiple of such processess - to perform flushing in a multi-threaded manner. Your objects could depend on other objects, and those objects could depend on even more objects. As django-denorm-iplweb tries to be a general-purpose tool, at this moment running multiple instances of `denorm_queue` is not recommended. In some situations this could be perfectly doable - in some, this could easily be a source of database deadlocks. Your milleage may vary - proceed with caution.

1.1.4 Final steps

Now that the models contain all information needed for the denormalization to work, we need to do some final steps to make the database use it. As django-denorm-iplweb uses triggers, those have to be created in the database with:

```
./manage.py denorm_init
```

This should be redone after every time you make changes to denormalized fields. On the other hand, unless you set `DENORM_INSTALL_TRIGGERS_AFTER_MIGRATE` variable to `False`, trigger installation will be performed every single time after `migrate` command is finished.

1.1.5 Testing denormalized apps

When testing a denormalized app you will need to instal the triggers in the setUp method. You could also use a tearDown procedure like:

```
from denorm import denorms

class TestDenormalisation(TestCase):

    def setUp(self):
        denorms.install_triggers()

    def tearDown(self):
        denorms.drop_triggers()
```

1.2 Reference

1.2.1 Decorators

1.2.2 Fields

1.2.3 Functions

1.2.4 Middleware

1.2.5 Management commands

denorm_init
denorm_drop
denorm_rebuild
denorm_flush
denorm_queue
denorm_sql

1.3 Changelog

1.3.1 0.5.5

- changes to reduce the chance of multiple `denorm_queue` processes trying to denormalize the same object

1.3.2 0.5.4

- don't wait for content_object when flushing queue, so we won't get deadlocks and Django exceptions

1.3.3 0.5.3

- select_for_update also for the updated object, so we won't get deadlocks

1.3.4 0.5.2

- include missing `conf` package.

1.3.5 0.5.1

- optimized denorms.rebuildall, using bulk_create,
- denorm_rebuild command gets 2 new command-line options, model_name and no_flush,
- ability to disable auto_now_add and auto_now fields during denorm flush, using settings – DENORM_DISABLE_AUTOTIME_DURING_FLUSH and field names DENORM_AUTOTIME_FIELD_NAMES,
- denorms.flush works in batches now.

1.3.6 0.5.0

- first release of django-denorm-iplweb,
- based on the high-quality code of the original django-denorm
- supported versions: Python 3.8, 3.9, Django 3.0, 3.1, 3.2,
- dropped support for MySQL,
- dropped support for SQLite,
- denorm_daemon becomes denorm_queue: - removed daemonzation code, - documented need to use supervisord or similar if background process needed, - used LISTEN/NOTIFY mechanisms from PostgreSQL,
- removed six dependency and __unicode__,
- added pre-commit hooks for autopep, flake8,
- added bumpver configuration,
- automatic trigger installation after post_migrate,
- documentation updated,
- post_migration signal causes trigger rebuild,
- `rebuild_triggers` command to rebuild triggers,
- deprecated command `denormalize` removed,
- field names given as a parameter to `skip` or `denorm_always_skip` are checked if they exist,
- triggers and functions names, generated for `@depend_on_related` include function (attribute) name,
- DirtyInstance includes func_name, which is a function name to rebuild only this single parameter

- ability to run multiple `denorm_queue` commands, which (thanks to the magic of row locking) should automatically process queue in a paralell manner.