Bulkmodel Documentation

Release 0.1.0

Alan Illing

Aug 21, 2018

Getting Started

This projects adds a number of features missing from Django's ORM. It enables heterogeneous updates, concurrent writes, retrieving records after bulk-creating them, and offline connection management to name a few features it provides.

CHAPTER 1

Installation and Getting Started

Install the package from pypi:

```
pip install django-bulkmodel
```

Add *bulkmodel* app to your *INSTALLED_APPS* list in settings.py:

```
INSTALLED_APPS = [
    ...
    'bulkmodel',
]
```

1.1 Full Installation

Fully installing BulkModel requires inheriting your models from `bulkmodel.models.BulkModel`:

```
from django.db import models
from bulkmodel.models import BulkModel

class MyModel(BulkModel):
    ...
```

And migrate the database.

Create migrations

If you're creating a new app from scratch:

```
./manage.py makemigrations <name-of-your-app>
```

Do this for each new app you create that have BulkModels.

Otherwise, if this app already exists and has migrations:

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

Apply migrations

And apply the migrations:

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

1.2 Partial Installation

If you don't want to migrate your database schema for whatever reason you can skip that step and `BulkModel` will degrade gracefully. With this route you'll lose the ability to retrieve a queryset after bulk creating data, and some signals will lose functionality.

With this route you'll need to point your `objects` reference on each `BulkModel`.

```
from django.db import models
from bulkmodel.models import BulkModel
from bulkmodel.managers import BulkModelManager

class MyModel(BulkModel):
    ...

    objects = BulkModelManager()
```

1.3 Optional Settings

Place the following in your `settings.py` to set global behavior of your bulkmodels:

- `MAX_CONCURRENT_BATCH_WRITES`

When set, this is the maximum number of concurrent workers that will be available to any concurrent write across your entire project. The default leaves this value unset.

- `ALWAYS_USE_CONCURRENT_BATCH_WRITES`

If True, django-bulkmodel will always use concurrent writes. The default is False.

CHAPTER 2

Overview of All Features

The goal of django-bulkmodel is to expand on Django's ORM so that it's better suited for interacting with bulk data.

1. **Updating data heterogeneously**

   The `update()` method that ships with Django applies a **homogeneous update**. That is, all model instances in the queryset are updated to the be same value for the columns specified.

   A BulkModel includes a new method named `update_fields()`, which allows you to update the database with different values for each model instance in the queryset through a single query execution.

   For more details see *bulk update user guide* and the *queryset API reference*.

2. **Getting querysets of bulk-created data**

   Sometimes you need to create some data and then do some further processing on the created records. However the `bulk_create` method returns what the database returns: the number of records returned.

   A BulkModel allows you to optionally return the queryset of objects created. So unless you can predict the primary key ahead of time, or can uniquely identify the data being inserted from some other combination you won't be able to get back the inserted data as it's represented in the database, with an assigned primary key.

   For more details see the *bulk create user guide* and the *queryset API reference*.

3. **Concurrent writes**

   In many cases and with a sufficiently capable database server you can accelerate bulk loading of data into the database by executing a concurrent write.

   BulkModels make this very easy– exposing three parameters to give you full control over how your writes are constructed.

   In each queryset write method (which includes `bulk_create`, `copy_from_objects`, `update` and `update_fields`) has the following parameters:

   - `batch_size`: The size of each chunk to write into the database; this parameter can be used with or without concurrency

- `concurrent`: If true, a write will happen concurrently. The default is False
- `max_concurrent_workers`: The total number of concurrent workers involved in the event loop.

For more details see the *concurrent writes user guide* and the *queryset API reference*.

4. **Offline connection management**

   Django manages the database connection inside a request / response cycle. A BulkModel is expecting data to be interacted with "offline" (meaning outside of the webserver) and checks or refreshes the connection if necessary when interacting with data in bulk.

   You can force a database connection check / refresh with the `ensure_connected()` queryset method.

   For more details see the *connection management user guide* and the *queryset API reference*.

5. **Missing signals**

   Django ships with the following signals for interacting with data:

   - Saving a single instance: `pre_save` and `post_save`
   - Deleting data: `pre_delete` and `post_delete`
   - Changing a many to many relationship: `m2m_changed`

   What's missing from this list are signals when data is created in bulk and updated in bulk.

   A BulkModel adds these signals and optionally lets you turn them off when calling any bulk write function.

   For more details see the *signals user guide* and the *signals reference*.

6. **Copying data to / from buffers**

   A BulkModel allows you write and read data by copying from and to a buffer, for databases that support it.

   For details on how to do this see the *copy to/from user guide* and the *queryset API reference*.

CHAPTER 3

What BulkModel does, by example

Suppose you have the following model:

```
from bulkmodel.models import BulkModel

class Foo(BulkModel):
    name = models.CharField(max_length=50, blank=False)
    value = models.IntegerField(null=False)
```

Some things you can do:

Retrieve bulk-created model instances

```
from random import randint, random, string

ls = []
for i in range(10):
    ls.append(Foo(
        # random string
        name = ''.join(random.choices(string.ascii_uppercase, 25)),

        # random value
        value = randint(0, 1000),
    ))

# create instances and return a queryset of the created items
foos = Foo.objects.bulk_create(ls, return_queryset=True)
```

Heterogeneously update data

The `.update()` method on a queryset performs a *homogeneous* update. That is, one or more columns for all the records in the queryset are updated to the same value.

Django-bulkmodel lets you set different values for different primary keys, with a simple and intuitive API, by introducing a method on a queryset called `update_fields()`.

```
for foo in foos:
    foo.value += randint(100, 200)

# update all fields that changed
foos.update_fields()

# or update just the value field
foos.update_fields('value')
```

Concurrent writes

The `batch_size` flag that ships with django inserts data synchronously, blocking on each batch to be written into the database.

If your database hardware is sufficient and you're on Python 3.4+ you can decrease overall write time by batch inserting concurrently. With django-bulkmodel you simply turn on the `concurrency` flag into any write operation.

```
foos = ...

# concurrently write foos into the database
Foo.objects.bulk_create(foos, concurrent=True, batch_size=1000, max_concurrent_
↪workers=10)

# a regular (homogeneous) update can be written concurrently
foos.update(concurrent=True, batch_size=1000, max_concurrent_workers=10)

# and so can a heterogeneous update
foos.update_fields(concurrent=True, batch_size=1000, max_concurrent_workers=10)
```

3.1 Bulk Create

Django ships with a `bulk_create` method that supports a batch_size parameter for batch writing.

Django-bulkmodel expands on this queryset method with some new options.

3.1.1 Returning queryset

Creating data in bulk returns what the database returns: the number of records created.

However there many cases where you want to obtain the created records for further manipulation, and there's no way to do with this without have the primary keys associated with each record.

Django-bulkmodel exposes a parameter called `return_queryset` which returns created data as a queryset.

```
from random import randint, random, string
from bulkmodel.models import BulkModel

class Foo(BulkModel):
    name = models.CharField(max_length=50, blank=False)
    value = models.IntegerField(null=False)

foo_objects = []
for i in range(10):
```

(continues on next page)

(continued from previous page)

```
    foo_objects.append(Foo(
        # random string
        name = ''.join(random.choices(string.ascii_uppercase, 25)),

        # random value
        value = randint(0, 1000),
    ))

# create instances and return a queryset of the created items
foos = Foo.objects.bulk_create(foo_objects, return_queryset=True)
```

3.1.2 Writing data by copying from a buffer

Bulk create will perform several inserts. Depending on your schema and database it may be faster to load data from a path or buffer.

For supported databases, a BulkModel queryset exposes this functionality.

```
foos = []
for i in range(10):
    foos.append(Foo(
        # random string
        name = ''.join(random.choices(string.ascii_uppercase, 25)),

        # random value
        value = randint(0, 1000),
    ))

foos = Foo.objects.copy_from_objects(ls, return_queryset=True)
```

The `return_queryset` is available on all write methods. See the *Queryset Reference* for more details.

3.1.3 Missing signals

A `BulkModel` adds several signals, including signals around creating data in bulk.

These signals are coupled to the two methods of creating data, as documented above:

- `pre_bulk_create` / `post_bulk_create`: signals fired when data is created from `bulk_create`
- `pre_copy_from_instances` / `post_copy_from_instances`: signals fired when data is created using `copy_from_objects`

You can optionally turn off emitting signals when creating data.

```
foo_objects = ...

# do not send signals (the default is True)
Foo.objects.bulk_create(foo_objects, send_signals=False)
```

For more information see the *signals user guide* or the *signals API reference*.

3.1.4 Concurrent writes

You can accelerate the loading of data by splitting work into batches and writing each batch concurrently.

A BulkModel queryset exposes three parameters to give you full control over this process:

- `batch_size`: The size of each chunk to write into the database; this parameter can be used with or without concurrency
- `concurrent`: If true, a write will happen concurrently. The default is False
- `max_concurrent_workers`: The total number of concurrent workers involved in the event loop.

Example

```
foos = ...

# concurrently write foos into the database
Foo.objects.bulk_create(foos, concurrent=True, batch_size=1000, max_concurrent_
↪workers=10)

# a regular (homogeneous) update can be written concurrently
foos.update(concurrent=True, batch_size=1000, max_concurrent_workers=10)

# and so can a heterogeneous update
foos.update_fields(concurrent=True, batch_size=1000, max_concurrent_workers=10)
```

For more information see the *concurrent writes user guide* or the *queryset API reference*.

3.2 Bulk Updates

Django ships with an `update()` method to update data to the database.

This method is limited to updating data **homogeneously**– that is, all the values for the column(s) being updated is set to the *same value* throughout the queryset.

Django-bulkmodel adds to the functionality by providing a `update_fields()` method, which updates data **heterogeneously**– that is, the values for the column(s) being updated can have different values for each model instance in the queryset.

Suppose you have the following model:

```
from bulkmodel.models import BulkModel

class Foo(BulkModel):
    name = models.CharField(max_length=50, blank=False)
    value = models.IntegerField(null=False)
```

Using `update` you can change the value to be the same

```
foos = ... # a queryset

foos.update(value = 5)
```

Using `update_fields` you can update records to have different values for each item.

```python
for foo in foos:
    # different value for each model instance in the queryset
    foo.value += randint(100, 200)

# update all fields that changed
foos.update_fields()

# or update just the value field
foos.update_fields('value')
```

Importantly, this will issue a **single query** against the database.

See *Queryset Reference* for more details.

3.3 Concurrent writes

Django comes with a `batch_size` parameter on the `bulk_create` queryset method.

Django-bulkmodel expands on the concept of batching in two ways:

- Batching is enabled on all write methods, including `update()` and `update_fields()`
- You can optionally write data concurrently and specify a number of workers that makes sense for your database server and data size

Note that performance of concurrent writes won't increase linearly. In fact, if your database is constrained with CPU resources, it's not likely to impact performance at all and could actually slow down your write.

This is an advanced feature that should be used with care. However you can improve write performance dramatically when used correctly.

3.3.1 Parameters

All database write methods have the following options to control concurrent writes:

- `concurrent`: Set to true to enable concurrent writes. False by default
- `batch_size`: Number of records to include in a single write (applies whether writing synchronous or asynchronous)
- `max_concurrent_workers`: Maximum number of concurrent writers to use to apply the database operation

---

See *Queryset API Reference* for more details.

3.4 Connection Management

By default Django manages the connection within a request / response cycle.

Django-bulkmodel enables offline connection management, so that you won't lose your connection outside of this cycle.

To check or refresh your connection (if necessary), call `ensure_connected()` on your queryset.

Django-bulkmodel internally calls this method as appropriate.

3.4.1 Example

```
foos = ... # some queryset
foos.ensure_connected().filter(name = 'alice')
```

See *Queryset API Reference* for more details.

3.5 Copy TO / FROM support

For database engines (i.e., Postgres) that support copying data into and out of a buffer Django-bulkmodel exposes this functionality into the queryset.

There are now two methods you can call:

- `copy_from_objects`: writes data from the provided list of objects to the database.
- `copy_to_instances`: reads data out of a buffer and populates a list of objects

3.5.1 Examples

Suppose you have the following model:

```
from bulkmodel.models import BulkModel

class Foo(BulkModel):
    name = models.CharField(max_length=50, blank=False)
    value = models.IntegerField(null=False)
```

Populate it with some data and use copy_from_objects to write the data into the database:

```
ls = []
for i in range(1000):
    ls.append(Foo(
        name = random_str(),
        value = randint(0, 1000),
    ))

# returning the queryset is optional
foos = Foo.objects.copy_from_objects(ls, return_queryset=True)
```

Likewise you can fetch data out the database by populating a list of objects from a buffer:

```
objs = Foo.objects.copy_to_instances()
```

3.6 Signals

Django ships with the following signals for database operations:

- Saving a single instance: `pre_save` and `post_save`

- Deleting data: `pre_delete` and `post_delete`
- Changing a many to many relationship: `m2m_changed`

Missing from this list is the ability to attach signals before and after updating data and bulk-creating data.

3.6.1 Bulk-create signals

The following signals are fired when data is created in bulk:

- `pre_bulk_create` is fired just before data is created
- `post_bulk_create` is fired just after data is created

For copying data into the database from a buffer (i.e., using `copy_from_instances`):

- `pre_copy_from_instances` is fired just before data is copied
- `post_copy_from_instances` is fired just after data is copied

3.6.2 Update signals

There are three sets of signals attached to the three ways you can update data.

For homogeneous updates (i.e., use the classic `update()`):

- `pre_update` is fired just before data is updated
- `post_update` is fired just after data is updated

For heterogeneous updates (i.e., using `update_fields()`):

- `pre_update_fields` is fired just before data is updated
- `post_update_fields` is fired just after data is updated

See *Siganls Reference* for more details.

3.7 QuerySet API Reference

3.7.1 QuerySet

3.7.2 Model Manager

3.8 Signals API Reference

API reference for additional signals included in Django-bulkmodel

---

3.8.1 Bulk create signals

pre_bulk_create

Fired before bulk_create writes data to the database

Parameters:

- `instances`: a list of model instances about to be written to the database

post_bulk_create

Fired after bulk_create has written data to the database

Parameters:

- `instances`: a list of model instances that have been written to the database
- `queryset`: a queryset of records saved in the bulk create; only applies if `return_queryset=True` is passed to `bulk_create()`

Fired after a bulk-create is issued

3.8.2 Update signals

pre_update

Fired just before `update()` performs a homogeneous update

Parameters:

- `instances`: a list of instances about to be updated

post_update

Fired just after `update()` performs a homogeneous update

Parameters:

- `instances`: a list of instances that have been updated

pre_update_fields

Fired just before `update_fields()` performs a hetergenous update

Parameters:

- `instances`: a list of instances about to be updated
- `field_names`: a list of fieldnames being updated; if empty, all fields are being updated
- `field_defaults`: defaults for each field, provided as a dictionary
- `batch_size`: the batch size used for the update

post_update_fields

Fired just after `update_fields()` performs a heterogeneous update

Parameters:

- `instances`: a list of instances about to be updated
- `queryset`: a queryset of records updated, if `return_queryset=True` is passed to update_fields
- `field_names`: a list of fieldnames being updated; if empty, all fields are being updated
- `field_defaults`: defaults for each field, provided as a dictionary
- `batch_size`: the batch size used for the update
- `n`: number of instances updated

3.8.3 Copy to / from signals

pre_copy_from_instances

Fired just before `copy_from_instances` writes data to the database

Parameters:

- `instances`: a list of instances about to be updated

post_copy_from_instances

Fired just after `copy_from_instances` writes data to the database

Parameters:

- `instances`: a list of instances that have been updated

3.9 Helpers

3.10 Concurrency Executor

CHAPTER 4

Indices and tables

- genindex
- modindex
- search