Broken-Down-Models

Shai Berger (Matific)

Sep 14, 2021

CONTENTS:

A library to help you break a large Django model down, transparently; that is, changing the structure of the model, while minimizing the changes this forces upon other parts of your project code.

CHAPTER

ONE

WHAT IS THIS?

In a Django project that goes on for several years, models tend to grow and accumulate fields. If you aren't very disciplined about this, you wake up one day, and find that one of your central tables, one with millions of rows, has 43 columns, including some TextFields. Most of them are not required most of the time, but the default (and common) use is to fetch all of them; also, since this table is queried a lot, the mere fact that it has so many columns makes some of the access slower.

When you realize that, you want to break it into components, such that only a few, most-important columns will participate in the large searches, while further details will be searched and fetched only when needed.

But that is a scary proposition – it might involve subtle code changes, break not just field access but also ORM queries... and this is a central model. The change imagined is open-heart surgery on a large project. Maybe, if we look the other way, it won't bother us too much...

Broken-Down-Models is here to help you. This is a library which can help you refactor your large model into a set of smaller ones, each with its own database table, while most of your project code remains unchanged.

1.1 How?

Django already includes a mechanism where fields for one model are stored in more than one table: Multi Table Inheritance (also known as MTI). That's what happens when we do "normal" inheritance of models, without specifying anything special in the Meta of either of the models.

Python also supports Multiple Inheritance – one class can have many parent classes. And this also works with Django's MTI – we can have multiple MTI.

Usually, when we think of a "core" set of attributes with different extensions, and we decide to implement it with MTI, we put this core set in a parent model, and make the extensions subclass it. But in the situation where we try to break down an existing model, this would mean that code which currently uses the large model will have to change, to recognize the new parts.

Broken-Down-Models puts this idea on its head: The extensions become parent models, and the core set is defined in a model which inherits them all. This way, all the fields are still fields of of the model we started with, for all purposes – including not just attribute access, but also ORM queries. For this to really work well, though, some further modifications are required; this is why this library exists.

CHAPTER

TWO

USING BROKEN-DOWN-MODELS

2.1 Installation

No surprises here:

```
pip install broken-down-models
```

You do not need to add anything to `INSTALLED_APPS` or any other Django setting.

2.1.1 Requirements

Broken-Down-Models is tested against CPython 3.7, 3.8 and 3.9, Django 2.2, 3.1, and 3.2, PostgreSQL and SQLite.

When using SQLite, Some migration operations require SQLite >= 3.3.0. See *CopyDataToPartial* for details – as far as we're aware, that is also the main hurdle to using the library with MySQL, Oracle, or any other DBMS (and like any good hurdle, hopping over it is not hard).

2.2 Usage

Assume we have a large, central model:

```
class Central(models.Model):
    a = models.IntegerField()
    b = models.CharField(max_length=100)
    c = models.DateTimeField()
    # ...
    z = models.IPV4AddressField()
```

We would like to break it down into groups of fields. Let's say that the first four fields are really core, useful almost whenever the model is used, but we want to separate out the others in groups of 5-6. We will rebuild the model as a set of related models.

When we are done, the `Central` model will have only five columns in its table – the four chosen fields, and `id`. Each of the other groups of fields will have their own model and be stored in their own table. The behavior of ORM queries will be changed, but (up to fringe limitations, see below) the changes will only affect performance, not semantics:

- For queries that just refer to the model (using any of the fields), Django will arrange the necessary joins for us behind the scenes.

- Queries that fetch `Central` objects will, by default, only bring in the core fields; the rest of the fields will be `deferred` – that is, the field value will be loaded from the database only when it is accessed, much like the way a `ForeignKey` is handled.

  This deferral is special, though: If any of the fields in a group is accessed, the whole group will be fetched.

2.2.1 Limitations

There is one obvious and hard limitation: We only handle objects accessed through the ORM, of course; raw SQL queries will not be magically adapted.

The library makes internal calls to `QuerySet.only()`; user calls to `only()` or `defer()` on querysets of broken-down models may interact with these calls in surprising ways.

The library does not handle the database constraints that should be imposed between a model and its broken-out components.

2.2.2 The Refactoring Process

The rewrite process involves two required steps – rewriting the models, and providing migrations – and a recommended step of optimizing queries. The next pages describe each of these in detail.

CHAPTER

THREE

REWRITING MODELS

Note: This continues the example defined in *the previous page*.

As mentioned previously, the separate groups are going to be parent classes for the new `Central`, so we'll have to define them first. These will be completely regular models, with one exception: We need to explicitly define their primary key, and give each of these primary keys a unique name. We can base this name on the model name; so we'll have something like:

```
class Group1(models.Model):
    group1_id = models.IntegerField(primary_key=True)  # New field
    e = models.BooleanField()  # This field is taken from Central
    f = models.TextField()     # This too
    # ...
    j = models.UUIDField(null=True)
```

Note that we're using an `IntegerField`, and not an `AutoField`, for the primary key; this is because we still assume that objects of this part of the `Central` model will not be created in isolation, but only as part of a complete `Central` object. In such creation, the primary key value will come from the complete object, and there is no need to generate it for each of the parts. In fact, an `AutoField` should work just as well – one is still allowed to set the value of an `AutoField` explicitly, and that is what a *`BrokenDownModel`* will do for its parents behind the scenes.

We'll define similarly the next groups:

```
class Group2(models.Model):
    group2_id = models.IntegerField(primary_key=True)
    k = models.BooleanField()
    # ...
    o = models.ForeignKey(SomeOtherModel, null=True, on_delete=models.CASCADE)

# and Group3, and...

class Group4(models.Model):
    group4_id = models.IntegerField(primary_key=True)
    # ...
    z = models.IPV4AddressField()
```

Now we can finally re-define the original model. We'll need to import some names from the library:

```
from bdmodels.fields import VirtualParentLink
from bdmodels.models import BrokenDownModel
```

and then:

```
class Central(BrokenDownModel, Group1, Group2, Group3, Group4):
    # Add an explicit PK here too
    id = models.AutoField(primary_key=True)

    # Add links to the parents
    group1_ptr = VirtualParentLink(Group1)
    group2_ptr = VirtualParentLink(Group2)
    group3_ptr = VirtualParentLink(Group3)
    group4_ptr = VirtualParentLink(Group4)

    # The original core fields we decided to leave in the model
    a = models.IntegerField()
    b = models.CharField(max_length=100)
    c = models.DateTimeField()
    d = models.DateField()
```

Note that we had to define the primary key explicitly here as well. This is because Django's default behavior for MTI is to use the parent-link to the first parent as the PK of the child. We do not want this.

The `VirtualPrentLink` fields defined explicitly, replace similarly-named `OneToOneField` fields which Django would generate, by default, to connect a child model with its MTI parents. They differ from such fields by all using the `id` column in the database – regular parent-link `OneToOneField` fields would each define their own column, although for our use case these columns would all be holding the same value (same as `id`).

With these definitions, our app is essentially ready to work against a database where the `Central` model has been broken down (up to some limitations, see below). But we still have to bring our database to this state. It is now time to talk about…

CHAPTER

FOUR

MIGRATIONS

Note: This continues the example started *before*.

If we go and change an existing model, central to our database, in the ways discussed in *Rewriting Models*, we need to change our database schema accordingly. As usual with Django, we will want to do this using migrations.

Regretfully, at the time this is written, there is no way to make Django's `makemigrations` aware of field or model types requiring special migration operations, so we will need to do some manual migration editing.

That said, `makemigrations` will still give us a good starting point, even if it will throw some fits on the way there. If we run it, having changed the models, some of the changes it sees are additions of *`VirtualParentLink`* fields named `*_ptr` to the original model. These `*_ptr` fields pose a problem to the automatic migration creator: As far as it understands, these are new non-nullable fields, and as such, they require a default value; it will ask us questions like:

```
You are trying to add a non-nullable field 'group1_ptr' to central without
a default; we can't do that (the database needs something to populate
existing rows).

Please select a fix:
1) Provide a one-off default now (will be set on all existing rows with a
   null value for this column)
2) Quit, and let me add a default in models.py

Select an option:
```

As explained above, these fields will not actually be represented by new columns in the database, and they do not need a default. But `makemigrations` cannot know that. To pacify it, we'll just give it a one-off default of 0, and edit this away later.

```
Select an option: 1

Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available, so you can
do e.g. timezone.now
Type 'exit' to exit this prompt
>>> 0
```

With this, `makemigrations` will manage to generate a migration. It will include the following changes:

- The new parent models are created
- The fields that were moved to parent models are removed from the existing model

- The `id` field on the existing model is changed to a `VirtualParentLink` (it really isn't, details shortly)
- The `*_ptr` `VirtualParentLink` fields are added to the existing model

It is interesting to note that migrations do not automatically change the model's superclass (list), and we will not change it either.

The migrations we want comprise four steps for each of the new parent models:

1. Create the new parent model.
2. Add the virtual parent link.
3. Transfer data from the existing model to the new parent model.
4. Remove from the existing model the fields that were duplicated on the parent.

The definition we provided for the `id` field exactly mimics the default provided by Django; it is there because without it, Django will try to use one of the parent-link keys as a PK. The generated operation to change it to a relation is created because Django tends to treat a relation field and the (usually hidden) `*_id` field it relies on as interchangeable; when it sees new relations which use `id` as their base field, it gets confused into thinking that `id` is the relation field. But we know better; we don't want `id` changed in any way by the migration, and we will remove this operation.

With all this in mind, we will edit the migration accordingly:

1. The new parent models are exactly as we need them, leave them be; remove the `AlterField` operation against the original model's `id` field.
2. We want the virtual parent link fields added, but we want them added only in the model and not in the database (that is why they are "virtual"). So, we want to replace the generated operations, which look like:

   ```
   migrations.AddField(
       model_name='central',
       name='group1_ptr',
       field=bdmodels.fields.VirtualParentLink(default=0, from_field='id', on_
   ↪delete=django.db.models.deletion.CASCADE, to='app.Group1'),
       preserve_default=False,
   ),
   ```

   with operations that do the right thing. The library provides this migration operation, we need to import it:

   ```
   from bdmodels import migration_ops
   ```

   and then we can use it:

   ```
   migration_ops.AddVirtualField(
       model_name='central',
       name='group1_ptr',
       field=bdmodels.fields.VirtualParentLink(from_field='id', on_delete=django.db.
   ↪models.deletion.CASCADE, to='app.Group1'),
   ),
   ```

   Note, that the default was removed from the field, and there is no `preserve_default=False` argument.

3. Now we'd like to transfer data from the existing full model to the new partial models. It is considered best practice to keep data-moving operations in separate migrations, and avoid mixing them with schema-changing operations. We'll make a new, empty migration to hold this operation:

   ```
   $ ./manage.py makemigrations --empty -n breakdown_copy app
   ```

Usually, data-moving in migrations is done with `RunPython` operations running functions which use the Django ORM. However, copying what is essentially a whole table efficiently requires using the SQL `INSERT-SELECT` construct, which is currently not supported by the ORM. We could write a `RunSQL` operation, but the library provides its own migration operation which writes the raw SQL for us, and even includes the reverse side of the operation.

As above, we will need to import the library migration operations:

```
from bdmodels import migration_ops
```

Then, we can write concise and clear operations:

```
operations = [
    migration_ops.CopyDataToPartial(
        full_model_name='Central',
        part_model_name='Group1',
    ),
    # ...
]
```

4. Finally, we can remove the now-redundant fields from the old model. We create another empty migration:

```
$ ./manage.py makemigrations --empty -n breakdown_cleanup app
```

and move into it all the `RemoveField` operations from the migration which `makemigrations` made for us.

If we look at it from the angle of the generated migration, we:

1. Kept the `CreateModel` operations;
2. Removed the `AlterField` operation;
3. Changed the `AddField` operations into *`AddVirtualField`* operations;
4. Added a 2nd migration with data-copying operations;
5. Moved the `RemoveField` operations to a 3rd migration which we added.

CHAPTER

FIVE

OPTIMIZING QUERIES

Breaking down a model is a trade-off: The main table will become smaller, queries which use or reference only the "core" fields are likely to become faster; but code which uses fields outside of the core can become much slower, and even trigger the infamous "1+N" behavior – processing a set of objects, which were all selected in one query before the breaking-down refactoring, may now require an additional query-per-object, if it involves fetching a field that has been moved out to a parent.

The library provides tools to overcome this – we can use *`select_related()`* to make specific queries join-in specific parents, or even *`fetch_all_parents()`* to join all of them; but in a large project, how can we find the places where this is needed?

5.1 Generally

nplusone is a library for detecting query inefficiencies in Python ORMs, which supports the Django ORM. In general, testing your code with this library can help you detect cases where your code is making 1+N queries. However, it is written for the typical case, where the problem is caused by following Foreign Keys. The way we set things up here, 1+N queries are caused by accessing previously-deferred fields, which are not necessarily Foreign Keys; nplusone cannot detect these.

While working on broken-down-models, we added to nplusone the feature of detecting instances of 1+N created by accessing deferred fields. Sadly, it seems that the original library is abandoned, and our pull-requests to improve it are not likely to be merged. But our fork is out there for your use.

Besides detecting more cases, this fork also adds a `TraceNotifier` which can be used to get reports with tracebacks when running your test-suite.

To do this, modify your `manage.py` as follows:

1. Add relevant imports:

```
from nplusone.core import profiler, notifiers
import nplusone.ext.django  # noqa -- required for profilers
```

2. Define a Profiler class, similar to this:

```
class Profiler(profiler.Profiler):
    def __init__(self, whitelist=None):
        from nplusone.ext.django.middleware import DjangoRule
        self.whitelist = [
            DjangoRule(**item)
            for item in (whitelist or [])
        ]
```

(continues on next page)

(continued from previous page)

```
        self.notifier = notifiers.TraceNotifier(
            {'NPLUSONE_LOG_LEVEL': logging.WARN}
        )

    def notify(self, message):
        if not message.match(self.whitelist):
            self.notifier.notify(message)
```

3. Apply the profiler to management command execution; replace the default

```
execute_from_command_line(sys.argv)
```

with:

```
with Profiler():
    execute_from_command_line(sys.argv)
```

With this, every potential case of 1+N queries will be logged with a full stack-trace, so you can find exactly where it comes from.

5.2 If it's the User model

`request.user` is often used in all sort of ways, and a broken-down user-model will, by default, cause many queries to be triggered for the acting user in the views – and, because it is placed in the request before your view gets control, you cannot fix these with code in the view.

If most of the uses of `request.user` only touch the core attributes, then that is fine. But if not, you may want to make sure that `request.user` is fetched with all the parts. If so, there's two (kinds of) places to take care of:

One is the fetching of users for authentication; django.contrib.auth uses, for this:

```
user = UserModel._default_manager.get_by_natural_key(username)
```

Since the other common use for `get_by_natural_key()` is for the `loaddata` and `dumpdata` commands, which deal with serialization, and where the whole user object is required as well, it makes sense to override the User model's default manager's `get_by_natural_key()` with something like:

```
def get_by_natural_key(self, username):
    return self.fetch_all_parents().get(**{self.model.USERNAME_FIELD: username})
```

The other is the code that fetches the user for the request when they're already logged in; this is "a kind of place" – the `get_user()` method of authentication backends. The canonical example is of course django.contrib.auth.backends.ModelBackend, whose method reads:

```
def get_user(self, user_id):
    try:
        user = UserModel._default_manager.get(pk=user_id)
    except UserModel.DoesNotExist:
        return None
    return user if self.user_can_authenticate(user) else None
```

For broken-down user models, you may prefer a backend with something like:

```
def get_user(self, user_id):
    """Overridden for BrokenDownModel support; used for fetching the request user"""
    user_manager = UserModel._default_manager.fetch_all_parents()
    try:
        user = user_manager.get(pk=user_id)
    except UserModel.DoesNotExist:
        return None
    return user if self.user_can_authenticate(user) else None
```

CHAPTER

SIX

WHAT IS REALLY GOING ON HERE

6.1 General Idea

Django already includes a mechanism where fields for one model are stored in more than one table – Multi Table Inheritance. That's what happens when we do "normal" inheritance of models, without specifying anything special in the Meta of either of the models.

If we have:

```
from django.db import models

class Parent(models.Model):
    parental = models.IntegerField()

class Child(Parent):
    childish = models.BooleanField()
```

Then we can use the `parental` field on the `Child` class as if it was defined there. Multiple inheritance is also supported, and the following almost works:

```
from django.db import models

class Mother(models.Model):
    motherly = models.IntegerField()

class Father(models.Model):
    fatherly = models.IntegerField()

# This DOES NOT WORK, just almost
class Child(Mother, Father):
    locale = models.ForeignKey("localization.Locale")
```

So – if we fix the little bump (details below), then we can break our large model into many small pieces. We can throw any field that's currently on the large model into its own model (and its own table); the large model will then subclass all of them. In principle, no other code will have to change.

Of course, that is a little too good to be true. Let us consider the…

6.2 Problems (and solutions)

6.2.1 Field clash

The first problem is that, as noted above, the models described above don't actually work. both `Mother` and `Father` have a field named `id` (the automatically generated PK), and the child cannot have two of them.

So, we just need to define explicitly the primary key fields on the parent tables:

```
from django.db import models

class Mother(models.Model):
    mother_id = models.IntegerField(primary_key=True)
    motherly = models.IntegerField()

class Father(models.Model):
    father_id = models.IntegerField(primary_key=True)
    fatherly = models.IntegerField()

# Now this does work
class Child(Mother, Father):
    locale = models.ForeignKey("localization.Locale")
```

6.2.2 Implicit Joins

The above already allows us to reduce the size of the large table, which we assume is the biggest problem. But still, with this, by default, queries on the large model would join in all of the parts (as if we called `select_related()` with all of them); in most use-cases, this is redundant and wasteful.

The solution is to limit the fields, by default, to the ones on the actual child model, by using the model _meta API to figure out which fields we want, and the QuerySet `only()` method. A special manager class for broken-down models has a `get_queryset()` which sets this up.

6.2.3 Broken `select_related()`

The solution to implicit joins works well. Actually, a little too well – in some cases, we'd want to have some part of the original model `select_related()`-ed, but naively using `only()` in the manager blocks it: Calling `select_related()` when all the relevant fields are deferred (by the `only()` call) achieves nothing. That is, as described so far,

```
Child.objects.select_related('locale')
```

works as expected, but

```
Child.objects.select_related('mother_ptr')
```

does not. Some special handling of `select_related()` is needed to make it behave as expected; thus, we need the special manager to be based on a special QuerySet class, and not just apply public API calls on a regular QuerySet.

6.2.4 Make accessed fields fetch their whole parent

With the above scheme, fields coming from parents all become deferred. This means that, when such a field is accessed for the first time, a database query is made to fetch its value. We'd prefer that, if a query is already made, we'll get all the fields from the relevant parent.

The way this query for the deferred field is done (internally in Django) is by calling the model method `refresh_from_db()`; that method can take an argument that tells it exactly which fields to fetch. Usually, when getting the value of a deferred field, the function is called with the name of that field only. We override it and make sure that whenever it is given names, we complement the list of names to include all the fields of relevant parent models.

6.2.5 Messed up id fields

On one hand: With Mutli Table Inheritance, for each of the parents, the child gets a `parent_ptr` one-to-one field – which means, there's also a `parent_ptr_id` column in the table (and field in the model, which we care a lot less about).

On the other hand, the pointer-field to the first parent is also taken as the Child's primary key – by default, `Child` has no id field.

We can make our own primary-key id field, that's easy; but with the kind of use we have in mind, we'd want all these `..._ptr_id` fields to also have just the same value as the `id` field. In fact, we don't want them at all – we'd much prefer if the original `id` field is used instead. To achieve this, we need to define these fields more-or-less explicitly, and set them to all point to the same database column. This requires some messing with internals (Django isn't really built to have columns shared between fields this way).

The solution involves a special type Foreign-Key field "family" – `VirtualForeignKey`, `VirtualOneToOneField` and `VirtualParentLink`; the former does the heavy lifting, and the latter two put a friendlier face on it. Making them work also requires some changes in the Django model `_meta` implementation – we define a subclass of the relevant Django class (`django.db.model.options.Options`) and plug it into the model.

CHAPTER

SEVEN

BDMODELS PACKAGE

7.1 bdmodels.models

class bdmodels.models.**BrokenDownModel**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Base class to replace models.Model for broken-down models.

When using it, make sure to make it the first base-class of your model, so that its modified metaclass replaces the regular Model metaclass.

It also specifies its own Manager, *BrokenDownManager*; if you have custom managers on your model, use that as your base manager.

Some Model methods are overridden just to change their implementation; notably, some checks are reimplemented and some checks are added.

The methods documented here are those which add functionality.

refresh_from_db(*using=None*, *fields=None*, ***, *all_parents: bool = False*)
This method is overridden for two purposes.

One is to make sure fetching any parent attribute fetches the whole parent.

The other is to add the all_parents argument, which can be used to reload the object in full, canceling deferrals. Since all_parents makes the model load all the fields, using it together with fields makes no sense and is an error.

getattr_if_loaded(*attr: str*, *default=None*)
Access an attribute (field), only if set specifically for the instance. This allows querying fields without causing unnecessary database round-trips.

class bdmodels.models.**BrokenDownManager**(**args*, ***kwargs*)
Basic Manager for broken-down models.

Connects the model to a *BrokenDownQuerySet* (and inherits its methods, as it is built from it).

class bdmodels.models.**BrokenDownQuerySet**(**args*, ***kwargs*)
Bases: django.db.models.query.QuerySet

Special queryset for use with broken-down models.

select_related(**fields*)
Fix select_related() for correct handling of parent deferrals.

Note: Of necessity, this means that if a parent is select_related, previous "only" is overridden and ignored. We make no effort to distinguish between deferrals created manually by the user, and those created

automatically to defer parents.

fetch_all_parents()
Select all fields in the model for immediate fetching, as if this was not a broken-down model.

This will make the query join all the parent tables.

bulk_create(*objs*, *batch_size=None*, *ignore_conflicts=False*)
Insert each of the instances into the database. Do *not* call save() on each of the instances, do not send any pre/post_save signals.

Setting the primary key attribute, if it is not set, is required for broken-down models; so if the PK is an autoincrement field, the database feature `can_return_rows_from_bulk_insert` (`can_return_ids_from_bulk_insert` on older Django versions) is required.

7.2 bdmodels.fields

class bdmodels.fields.**VirtualForeignKey**(*to*, *from_field*, *on_delete*, *related_name=None*, *related_query_name=None*, *limit_choices_to=None*, *parent_link=False*, *to_field=None*, *db_constraint=False*, ***kwargs*)
Bases: `django.db.models.fields.related.ForeignKey`

A reference to a foreign object, based on an existing field

This is just like a `ForeignKey` with the exception that, rather than creating a related `*_id` field to hold the id of the referenced object, it uses one of the existing fields of the model.

The name of the field to be used is given as the required parameter `from_field`.

Since the assumption is that the existing field serves other purposes (either it is interesting in itself, or the id it holds references more than one object), we limit changes through this field. Thus, Attempts to change the field's value are blocked. Accordingly, it must be non-editable, and its `on_delete` rule must not change the field's value. Similarly, a default does not make sense.

Adding constraints would make sense – but this is currently not supported.

If an index is needed, it should be defined on the concrete field.

class bdmodels.fields.**VirtualOneToOneField**(*to*, *from_field*, *on_delete*, *to_field=None*, ***kwargs*)
Bases: `django.db.models.fields.related.OneToOneField`, *bdmodels.fields.VirtualForeignKey*

One-to-one relationship based on existing field

This field is to a `OneToOneField` as a *VirtualForeignKey* is to a `ForeignKey`, and vice versa – it is also to *VirtualForeignKey* as a `OneToOneField` is to a `ForeignKey`.

class bdmodels.fields.**VirtualParentLink**(*to*, *from_field='id'*, *on_delete=<function CASCADE>*, *to_field=None*, ***kwargs*)
Bases: *bdmodels.fields.VirtualOneToOneField*

A *VirtualOneToOneField* that is also a parent link

The most common use for *VirtualOneToOneField* while breaking down models is for a field that is also a link to a parent model, and whose base field is the model's primary key.

This is mostly a shorthand for this use-case – the `parent_link` attribute is set to `True`, and the `from_field` has a default of `'id'` (using the model's actual PK is more involved, and is left for the future).

7.3 bdmodels.migration_ops

bdmodels.migration_ops.**AddVirtualField**(*, *model_name: str*, *name: str*, *field*)

A thin wrapper – limit AddField to act on the model and not on the database.

Parameters

- **model_name** – The model where the field is to be added
- **name** – The name of the field to be added
- **field** – The (virtual) field to be added

class bdmodels.migration_ops.**CopyDataToPartial**(**args*, ***kwargs*)

A migration operation for moving data from a complete model, to a model which has some of the complete model's fields, efficiently.

This is useful when breaking down a large model to parts.

This is a data operation – it moves data, does not change schema; the kind of operation typically written as a RunPython operation.

Implementation The forwards direction of the operation uses SQL INSERT-SELECT to create the rows in the table of the partial model. The backwards side uses UPDATE with a join to copy data from the partial model's table into (existing) rows of the complete model's table.

Compatibility While INSERT-SELECT is standard SQL, UPDATE with a join (A.K.A UPDATE-FROM) is not. The library currently uses the PostgreSQL syntax, which is also supported by SQLite >= 3.3.0; for this reason, the backwards side of this migration operation only works with these database backends. Until this is fixed, users who need this operation with other backends can write it as a RunSQL operation.

The SQLite documentation reviews support of this feature in different systems, see there for details.

__init__(*full_model_name: str*, *part_model_name: str*, *elidable: bool = True*)

Parameters

- **full_model_name** – The name of the full model (which at this point has all the fields)
- **part_model_name** – The name of the partial model (whose fields are a PK and some fields copied from the full model)
- **elidable** – Specifies if this operation can be elided when migrations are squashed

CHAPTER

EIGHT

CONTRIBUTION GUIDE

Contributions to this project, in all forms, are welcome. At this point, we do not have formal governance or roles; the community that we hope to form around this code will set them up as necessary. The project is originally developed and shepherded by Matific, AKA Slate Science.

8.1 Community

The project is run and managed on Github. For issues or pull requests, please use the tools provided there. For questions or support, please reach out to project contributors:

Contributor	Django Forum	Github	Other
Shai Berger	shaib	shaib	Twitter: @shaib_il

In all communications and actions related to this project we ask that you respect the code of conduct we blatantly copied from Django [*].

8.1.1 Code of Conduct

- **Be friendly and patient.**
- **Be welcoming.** We strive to be a community that welcomes and supports people of all backgrounds and identities. This includes, but is not limited to members of any race, ethnicity, culture, national origin, colour, immigration status, social and economic class, educational level, sex, sexual orientation, gender identity and expression, age, size, family status, political belief, religion, and mental and physical ability.
- **Be considerate.** Your work will be used by other people, and you in turn will depend on the work of others. Any decision you take will affect users and colleagues, and you should take those consequences into account when making decisions. Remember that we're a world-wide community, so you might not be communicating in someone else's primary language.
- **Be respectful.** Not all of us will agree all the time, but disagreement is no excuse for poor behavior and poor manners. We might all experience some frustration now and then, but we cannot allow that frustration to turn into a personal attack. It's important to remember that a community where people feel uncomfortable or threatened is not a productive one. Members of our community should be respectful when dealing with other members as well as with people outside our community.
- **Be careful in the words that you choose.** We are a community of professionals, and we conduct ourselves professionally. Be kind to others. Do not insult or put down other participants. Harassment and other exclusionary behavior aren't acceptable. This includes, but is not limited to:
  - Violent threats or language directed against another person.

  - Discriminatory jokes and language.
  - Posting sexually explicit or violent material.
  - Posting (or threatening to post) other people's personally identifying information ("doxing").
  - Personal insults, especially those using racist or sexist terms.
  - Unwelcome sexual attention.
  - Advocating for, or encouraging, any of the above behavior.
  - Repeated harassment of others. In general, if someone asks you to stop, then stop.

- **When we disagree, try to understand why.** Disagreements, both social and technical, happen all the time and this project is no exception. It is important that we resolve disagreements and differing views constructively. Remember that we're different. The strength of the project comes from its varied community, people from a wide range of backgrounds. Different people have different perspectives on issues. Being unable to understand why someone holds a viewpoint doesn't mean that they're wrong. Don't forget that it is human to err and blaming each other doesn't get us anywhere. Instead, focus on helping to resolve issues and learning from mistakes.

8.2 Technically

The code and documentation for the project are included in the same repository. Changes to code should be accompanied by respective changes to tests and documentation, where relevant.

The project is tested against Python>=3.7 and supported versions of Django (2.2.x, 3.1.x and 3.2.x at the time this is written). We strongly recommend the latest stable point-release of each of the above.

We use poetry to manage builds and tox to manage tests.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

F

G

M

R

S

V