[Technical](https://www.caktusgroup.com/blog/categories/technical/)

**Digging Into Django QuerySets**

* [python](https://www.caktusgroup.com/blog/tags/python/),
* [django](https://www.caktusgroup.com/blog/tags/django/)

Jeff Bradberry

April 5, 2017

  

**Digging Into Django QuerySets**

[Object-relational mappers](https://en.wikipedia.org/wiki/Object-relational_mapping) (or ORMs for short), such as the one that comes built-in with Django, make it easy for even new developers to become productive without needing to have a large body of knowledge about how to make use of relational databases. They abstract away the details of database access, replacing tables with declarative model classes and queries with chains of method calls. Since this is all done in standard Python developers can build on top of it further, adding instance methods to a model to wrap reusable pieces of logic. However, the abstraction provided by ORMs is not perfect. There are pitfalls lurking for unwary developers, such as the **N + 1 problem**. On the bright side, it is not difficult to explore and gain a better understanding of Django's ORM. Taking the time and effort to do so will help you become a better Django developer.

In this article I'll be setting up a simple example app, consisting of nothing more than a few models, and then making use of the Django shell to perform various queries and examine the results. You don't have to follow along, but it is recommended that you do so.

First, create a clean virtualenv. Here I'll be using Python 3 all of the way, but there should be little difference with Python 2.

$ mkvirtualenv -p **$(**which python3**)** querysets

Already using interpreter /usr/bin/python3

Using base prefix '/usr'

New python executable in /home/jrb/.virtualenvs/querysets/bin/python3

Also creating executable in /home/jrb/.virtualenvs/querysets/bin/python

Installing setuptools, pkg\_resources, pip, wheel...done.

Next, install Django and IPython,

**(**querysets**)** $ pip install django ipython

Create the new project.

**(**querysets**)** $ django-admin.py startproject querysets

**(**querysets**)** $ cd querysets/

**(**querysets**)** $ ./manage.py startapp qs

Update querysets/settings.py to add 'qs', to the end of the INSTALLED\_APPS list. Then, edit qs/models.py to add the simple models we will be dealing with

**from** django.db **import** models

**class** OnlyOne**(**models**.**Model**):**

name **=** models**.**CharField**(**max\_length**=16)**

**class** MainModel**(**models**.**Model**):**

name **=** models**.**CharField**(**max\_length**=16)**

one **=** models**.**ForeignKey**(**OnlyOne**)**

**class** RelatedModel**(**models**.**Model**):**

name **=** models**.**CharField**(**max\_length**=16)**

main **=** models**.**ForeignKey**(**MainModel**,** related\_name**=**'many'**)**

Finally, set up the database.

**(**querysets**)** jrb@caktus025:~/caktus/querysets$ ./manage.py makemigrations qs

Migrations **for** 'qs':

qs/migrations/0001\_initial.py:

- Create model MainModel

- Create model OnlyOne

- Create model RelatedModel

- Add field one to mainmodel

**(**querysets**)** jrb@caktus025:~/caktus/querysets$ ./manage.py migrate

...

Running python manage.py shell should now pull up an IPython session.

Now that we have a working project set up, we'll need some means of keeping track of the quantity and the raw SQL of any queries sent to the database. [Django's TransactionTestCase class](https://github.com/django/django/blob/1.10.6/django/test/testcases.py#L957) provides an assertNumQueries method, which is interesting but too specific and too tied to the test suite for our needs. However, examining its implementation, we can see that it ultimately makes use of a context manager called CaptureQueriesContext, from the django.test.utils module. This context manager will cause a database connection to capture all of the SQL queries sent, even if such is currently turned off (i.e. if DEBUG = False is set), and make those queries available on the context object. I find this a useful tool to use in debugging to track down code that is issuing too many queries to the database, in situations where Django Debug Toolbar won't help.

At the time of writing, the most recent released version of Django is 1.10.6. I've copied the code for CaptureQueriesContext for this version below, with a few irrelevancies redacted.

**class** CaptureQueriesContext**(**object**):**

**def** \_\_init\_\_**(**self**,** connection**):**

self**.**connection **=** connection

**@property**

**def** captured\_queries**(**self**):**

**return** self**.**connection**.**queries**[**self**.**initial\_queries**:**self**.**final\_queries**]**

**def** \_\_enter\_\_**(**self**):**

self**.**force\_debug\_cursor **=** self**.**connection**.**force\_debug\_cursor

self**.**connection**.**force\_debug\_cursor **=** True

self**.**initial\_queries **=** len**(**self**.**connection**.**queries\_log**)**

self**.**final\_queries **=** None

request\_started**.**disconnect**(**reset\_queries**)**

**return** self

**def** \_\_exit\_\_**(**self**,** exc\_type**,** exc\_value**,** traceback**):**

self**.**connection**.**force\_debug\_cursor **=** self**.**force\_debug\_cursor

request\_started**.**connect**(**reset\_queries**)**

**if** exc\_type **is** **not** None**:**

**return**

self**.**final\_queries **=** len**(**self**.**connection**.**queries\_log**)**

So here we can see several things of interest to us. The context manager keeps a reference to the database connection (as self.connection), it sets and then unsets a flag on the connection (self.connection.force\_debug\_cursor) which tells the connection to do the captures, it stores the number of queries at the start and at the end (self.initial\_queries and self.final\_queries), and finally, it provides a slice of the actual queries captured as the property captured\_queries. Nothing here restricts its use to the test suite, so we'll be making use of it throughout in our IPython session.

Let's try it out now.

In **[1]:** **from** django.test.utils **import** CaptureQueriesContext

In **[2]:** **from** django.db **import** connection

In **[3]:** **from** qs **import** models

In **[4]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **print(**models**.**MainModel**.**objects**.**all**())**

**...:**

**<**QuerySet **[]>**

In **[5]:** **print(**context**.**initial\_queries**,** context**.**final\_queries**)**

**0** **1**

So we can see that there were no queries to start out with, and that a query was issued to the database by our code. Let's see what that looks like,

In **[6]:** **print(**context**.**captured\_queries**)**

**[{**'time'**:** '0.001'**,** 'sql'**:** 'SELECT "qs\_mainmodel"."id", "qs\_mainmodel"."name", "q

s\_mainmodel"."one\_id" FROM "qs\_mainmodel" LIMIT 21'}]

This shows us that the captured\_queries property gives us a list of dicts, and each dict contains the raw SQL and the time it took to execute. In the above query, note the LIMIT 21. This is there because the repr() of a QuerySet limits itself to showing no more than 20 of the items it contains. The additional twenty-first item is captured so that it knows whether or not to add an ellipsis at the end to indicate that there are more items available.

Let's create some data. First up, we need a quick and dirty way of populating the name fields

In **[7]:** **import** random

In **[8]:** **import** string

In **[9]:** **def** random\_name**():**

**...:** **return** ''**.**join**(**random**.**choice**(**string**.**ascii\_letters**)** **for** i **in** range**(16**

**...:** **))**

**...:**

In **[10]:** random\_name**()**

Out**[10]:** 'nRtybzKaSZWjHOBZ'

Now the objects

In **[11]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** models**.**OnlyOne**.**objects**.**bulk\_create**([**

**...:** models**.**OnlyOne**(**name**=**random\_name**())**

**...:** **for** i **in** range**(5)**

**...:** **])**

**...:** models**.**MainModel**.**objects**.**bulk\_create**([**

**...:** models**.**MainModel**(**name**=**random\_name**(),** one\_id**=**i **+** **1)**

**...:** **for** i **in** range**(5)**

**...:** **])**

**...:** models**.**RelatedModel**.**objects**.**bulk\_create**([**

**...:** models**.**RelatedModel**(**name**=**random\_name**(),** main\_id**=**i **+** **1)**

**...:** **for** i **in** range**(5)**

**...:** **for** x **in** range**(7)**

**...:** **])**

**...:**

In **[12]:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**6**

In **[13]:** **print(**context**.**captured\_queries**)**

**[{**'sql'**:** 'BEGIN'**,** 'time'**:** '0.000'**},** **{**'sql'**:** 'INSERT INTO "qs\_onlyone" ("name") S

ELECT \'TSdxoYKuGUnijmVY\' UNION ALL SELECT \'DNcSpIJbnVXjbabq\' UNION ALL SELEC

T \'suAQQAzEflBqLxuc\' UNION ALL SELECT \'hWtuPkfjdATxZhNV\' UNION ALL SELECT \'

GTwPkXTUSpZYBWCT\''**,** 'time'**:** '0.000'**},** **{**'sql'**:** 'BEGIN'**,** 'time'**:** '0.000'**},** **{**'sql'

**:** 'INSERT INTO "qs\_mainmodel" ("name", "one\_id") SELECT \'fsekHOfSJxdiGiqp\', 1

UNION ALL SELECT \'dHdPoqeKZzRCJEql\', 2 UNION ALL SELECT \'MiDwEPvqIuxCEArT\',

**3** UNION ALL SELECT \'yCCRaPiLzWnnUewS\', 4 UNION ALL SELECT \'ftfQWWfuZhXblNlF\'

**,** **5**', 'time': '**0.000**'}, {'sql': 'BEGIN', 'time': '**0.000**'}, {'sql': 'INSERT INTO

"qs\_relatedmodel" **(**"name"**,** "main\_id"**)** SELECT \'tMOCzPRjKZHbwBLb\', 1 UNION ALL S

ELECT \'SLxCmCtmxeCwpkAC\', 1 UNION ALL SELECT \'qccDQKsgmTFCIMsF\', 1 UNION ALL

SELECT \'LWAvYdGQvBdlsYjI\', 1 UNION ALL SELECT \'gLjaGTjoLoNMkbDl\', 1 UNION A

LL SELECT \'PqjvVLhCFoMOlVfH\', 1 UNION ALL SELECT \'BpHnEzhucSZNryWs\', 1 UNION

ALL SELECT \'CYOkzJgsrGYOLOoB\', 2 UNION ALL SELECT \'HheikdsLaQnWpZBj\', 2 UNI

ON ALL SELECT \'mXqccnNYLDrQOoiT\', 2 UNION ALL SELECT \'BipJWoDVlJoyNPxD\', 2 U

NION ALL SELECT \'BgvvYHlAHegyRjbF\', 2 UNION ALL SELECT \'GrlOlbMwnqfkPKZX\', 2

UNION ALL SELECT \'OFJchZLVjmXNAHjO\', 2 UNION ALL SELECT \'SYHRkSvBmupzUHXO\',

**3** UNION ALL SELECT \'imAQEQUyrNoRNRSG\', 3 UNION ALL SELECT \'ZEvmnPMurchiLfcd\

'**,** **3** UNION ALL SELECT \'kYtKQNoeoUuxpYPC\', 3 UNION ALL SELECT \'FvGFRSMariUanWs

L\', 3 UNION ALL SELECT \'VKXEeClDnrnruAng\', 3 UNION ALL SELECT \'eDnEaWAqWRWdC

vMc\', 3 UNION ALL SELECT \'wmIKiiqHBAJiOkMb\', 4 UNION ALL SELECT \'pzEMvmVqbSk

LICVO\', 4 UNION ALL SELECT \'dIclLsVIXaHIyUYk\', 4 UNION ALL SELECT \'nDyHLSYgB

AYIZAkP\', 4 UNION ALL SELECT \'GfrOYcPPYRXMBvmC\', 4 UNION ALL SELECT \'PZUiAwe

kQlmIMJAW\', 4 UNION ALL SELECT \'jnWbngcVgVPFAJNn\', 4 UNION ALL SELECT \'RQQyr

DQpTIPxItND\', 5 UNION ALL SELECT \'SaFNLtavfdceqzTE\', 5 UNION ALL SELECT \'CSm

oYuPNttJTFdlH\', 5 UNION ALL SELECT \'PxufMeDfIBeMAtQV\', 5 UNION ALL SELECT \'m

NaTQepfHkFMRFet\', 5 UNION ALL SELECT \'CHlOqOHXIDyzorfW\', 5 UNION ALL SELECT \

'BKgXGwdXJQBMQGJM\', 5'**,** 'time'**:** '0.000'**}]**

This looks pretty ugly, but we can see that each .bulk\_create() results in two queries, a BEGIN starting the transaction, and an INSERT INTO with a crazy set of SELECT and UNION ALL clauses following it.

Ok, now that we are finally all set up, let's explore. What happens if we just create a QuerySet and set it in a variable?

In **[14]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**all**()**

**...:**

In **[15]:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**0**

In **[16]:** **print(**context**.**captured\_queries**)**

**[]**

No queries were sent to the database! This is because a Django QuerySet is a *lazy* object. It contains all of the information it needs to populate itself from the database, but will not actually do so until the information is needed. Similarly, .filter(), .exclude(), and the other QuerySet-returning methods will not, by themselves, trigger a query sent to the database.

In **[17]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**filter**(**name**=**'foo'**)**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**0**

In **[18]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**filter**(**name**=**'foo'**)**

**...:** qs2 **=** qs**.**filter**(**name**=**'bar'**)**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**0**

Here we see that even chaining a filtered QuerySet off of another QuerySet is insufficient to cause a database access. However, non-QuerySet-returning methods such as .count() *will* result in a query sent to the database.

In **[19]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** count **=** models**.**MainModel**.**objects**.**count**()**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**1**

So, when will a QuerySet result in a round-trip to the database? Basically, this happens any time concrete results are needed from the QuerySet, such as looping explicitly or implicitly. Here are some of the more typical ones

In **[20]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** m **in** models**.**MainModel**.**objects**.**all**():**

**...:** obj **=** m

**...:** r **=** repr**(**models**.**OnlyOne**.**objects**.**all**())**

**...:** l **=** len**(**models**.**RelatedModel**.**objects**.**all**())**

**...:** list\_main **=** list**(**models**.**MainModel**.**objects**.**all**())**

**...:** b **=** bool**(**models**.**OnlyOne**.**objects**.**all**())**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**5**

Note that each of these triggers its own query. The [Django docs](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#when-querysets-are-evaluated) have a full list of the things that cause a QuerySet to trigger a query.

We've now seen that simply instantiating a QuerySet doesn't send a query to the database, and that obtaining data out of it does. The next most obvious question is, will a QuerySet ask for data from the database multiple times? Let's find out

In **[21]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**all**()**

**...:** L **=** list**(**qs**)**

**...:** L2 **=** list**(**qs**)**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**1**

Terrific! Just as we would hope, the QuerySet somehow reuses its previous data when we ask for it again. Keep in mind, though, if we attempt to further refine a QuerySet,

In **[22]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**all**()**

**...:** L **=** list**(**qs**)**

**...:** qs2 **=** qs**.**filter**(**name\_\_startswith**=**'b'**)**

**...:** L2 **=** list**(**qs2**)**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**2**

it does *not* re-use the data. So how does this work? The implementation of QuerySet can be found in [django.db.models.query](https://github.com/django/django/blob/1.10.6/django/db/models/query.py), but in particular, let's look at the implementation of the relevant methods

**def** \_\_iter\_\_**(**self**):**

self**.**\_fetch\_all**()**

**return** iter**(**self**.**\_result\_cache**)**

**def** \_fetch\_all**(**self**):**

**if** self**.**\_result\_cache **is** None**:**

self**.**\_result\_cache **=** list**(**self**.**iterator**())**

**if** self**.**\_prefetch\_related\_lookups **and** **not** self**.**\_prefetch\_done**:**

self**.**\_prefetch\_related\_objects**()**

**def** iterator**(**self**):**

**return** iter**(**self**.**\_iterable\_class**(**self**))**

So we can see that iterating over a QuerySet will check to see if a cache at .\_result\_cache is populated yet, and if not, populates it with a list of objects. This list, then, is what will be iterated over. Subsequent iterations will then get the cache, so no further queries are issued. Doing a chained .filter() call, though, results in a new QuerySet that does not share the cache of the previous one.

The [iterator() method](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#iterator) used above is a documented public method, which returns an iterator over a configurable iterable class of model instances. Note that it does *not* involve the cache, so subsequent calls will result in a new query to the database. So why is this a public method? Under what circumstances would it be useful to not populate the cache? The iterator() method is most useful when you have memory concerns when iterating over a particularly large QuerySet, or one that has a large amount of data stored in the fields, especially if it is known that the QuerySet will only be used once and then thrown away.

Interestingly, certain non-QuerySet-returning methods such as .count(),

In **[23]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** qs **=** models**.**MainModel**.**objects**.**all**()**

**...:** L **=** list**(**qs**)**

**...:** c **=** qs**.**count**()**

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**1**

can also make use of an already filled cache.

A common pattern that you will see is iterating over a QuerySet in a template, and rendering information about each item, which may involve access of related objects. To simulate this, let's loop and set the name of each item's OnlyOne into a variable.

In **[24]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** item **in** models**.**MainModel**.**objects**.**all**():**

**...:** name **=** item**.**one**.**name

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**6**

Six queries! What could possibly be going on here?

In **[25]:** **for** q **in** context**.**captured\_queries**:**

**...:** **print(**q**[**'sql'**])**

**...:**

SELECT "qs\_mainmodel"**.**"id"**,** "qs\_mainmodel"**.**"name"**,** "qs\_mainmodel"**.**"one\_id" FROM "qs\_mainmodel"

SELECT "qs\_onlyone"**.**"id"**,** "qs\_onlyone"**.**"name" FROM "qs\_onlyone" WHERE "qs\_onlyone"**.**"id" **=** **1**

SELECT "qs\_onlyone"**.**"id"**,** "qs\_onlyone"**.**"name" FROM "qs\_onlyone" WHERE "qs\_onlyone"**.**"id" **=** **2**

SELECT "qs\_onlyone"**.**"id"**,** "qs\_onlyone"**.**"name" FROM "qs\_onlyone" WHERE "qs\_onlyone"**.**"id" **=** **3**

SELECT "qs\_onlyone"**.**"id"**,** "qs\_onlyone"**.**"name" FROM "qs\_onlyone" WHERE "qs\_onlyone"**.**"id" **=** **4**

SELECT "qs\_onlyone"**.**"id"**,** "qs\_onlyone"**.**"name" FROM "qs\_onlyone" WHERE "qs\_onlyone"**.**"id" **=** **5**

As we can see, we have one query which populates the main QuerySet, but then as each item gets processed, each sends an *additional* query to get the item's associated OnlyOne object. This is referred to as the **N + 1 Problem**. But how can we fix it? It turns out that Django comes with a QuerySet method for just this purpose: select\_related(). If we adjust our code like this,

In **[26]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** item **in** models**.**MainModel**.**objects**.**select\_related**(**'one'**).**all**():**

**...:** name **=** item**.**one**.**name

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**1**

we drop back down to only one query again

In **[27]:** **for** q **in** context**.**captured\_queries**:**

**...:** **print(**q**[**'sql'**])**

**...:**

SELECT "qs\_mainmodel"**.**"id"**,** "qs\_mainmodel"**.**"name"**,** "qs\_mainmodel"**.**"one\_id"**,** "qs\_

onlyone"."id", "qs\_onlyone"."name" FROM "qs\_mainmodel" INNER JOIN "qs\_onlyone" O

N **(**"qs\_mainmodel"**.**"one\_id" **=** "qs\_onlyone"**.**"id"**)**

So .select\_related('one') tells Django to do an INNER JOIN across the foreign key, and make use of that information when instantiating the objects in Python. Great! The select\_related() method is capable of taking multiple arguments and will do a join for each of them. You can also join multiple tables deep by using Django's double-underscore syntax, for example .select\_related('foo\_\_bar') would join our main model's table with the table for 'foo', and then further join with the table for 'bar'. Note that other things that would cause a join in the sql, such as filtering on a field on the related object, will not cause that related object to be made available as a Python object; you still need to specify your .select\_related() fields explicitly.

This all works if the model we are querying has a foreign key to the other model. What if the relationship runs the other way, resulting in a one-to-many relationship?

In **[29]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** item **in** models**.**MainModel**.**objects**.**all**():**

**...:** **for** related **in** item**.**many**.**all**():**

**...:** name **=** related**.**name

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**6**

In **[30]:** **for** q **in** context**.**captured\_queries**:**

**...:** **print(**q**[**'sql'**])**

**...:**

SELECT "qs\_mainmodel"**.**"id"**,** "qs\_mainmodel"**.**"name"**,** "qs\_mainmodel"**.**"one\_id" FROM

"qs\_mainmodel"

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" = 1

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" = 2

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" = 3

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" = 4

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" = 5

As before, we get 6 queries. However, if we were to try to use .select\_related('many'), we would get a FieldError. For this situation, Django provides a different method to mitigate the problem: prefetch\_related.

In **[31]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** item **in** models**.**MainModel**.**objects**.**prefetch\_related**(**'many'**).**all**()**

**...:** **:**

**...:** **for** related **in** item**.**many**.**all**():**

**...:** name **=** related**.**name

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**2**

Two queries, that's at least better. What's going on here, though, why two? If we take a look at the queries generated, we see

In **[32]:** **for** q **in** context**.**captured\_queries**:**

**...:** **print(**q**[**'sql'**])**

**...:**

SELECT "qs\_mainmodel"**.**"id"**,** "qs\_mainmodel"**.**"name"**,** "qs\_mainmodel"**.**"one\_id" FROM

"qs\_mainmodel"

SELECT "qs\_relatedmodel"**.**"id"**,** "qs\_relatedmodel"**.**"name"**,** "qs\_relatedmodel"**.**"main

\_id" FROM "qs\_relatedmodel" WHERE "qs\_relatedmodel"."main\_id" IN (1, 2, 3, 4, 5)

So it turns out that Django first loads up the QuerySet for MainModel, then it determines what primary key values it received, and then does a second query on RelatedModel, filtering on those that have a foreign key to one of those values.

There is one thing that you should be aware of when prefetching one-to-many relationships in this manner. A fairly typical thing to do is to make use of Django model's object-oriented nature, and write instance methods that do some non-trivial computation, sometimes involving looping or filtering on one-to-many or many-to-many relationships. We'll simulate that here by just using a .filter() call in the inner loop

In **[33]:** **with** CaptureQueriesContext**(**connection**)** **as** context**:**

**...:** **for** item **in** models**.**MainModel**.**objects**.**prefetch\_related**(**'many'**).**all**()**

**...:** **:**

**...:** **for** related **in** item**.**many**.**filter**(**name\_\_startswith**=**'b'**):**

**...:** name **=** related**.**name

**...:** **print(**context**.**final\_queries **-** context**.**initial\_queries**)**

**...:**

**7**

And now we find that we're back up to seven queries, despite the use of .prefetch\_related(). What's going on here is that the prefetch is making item.many.all() act exactly like an already iterated-over QuerySet, like from earlier in this article, by filling its cache for later re-use. However, as in those earlier cases, if you do any further refinement of the QuerySet it does not share the cache with the new QuerySet. In many cases, it would simply be better to iterate over the relationship and filter using Python directly. Additionally, Django starting with version 1.7 introduced a [Prefetch](https://docs.djangoproject.com/en/1.10/ref/models/querysets/#django.db.models.Prefetch) object, which allows more control over the query used in the prefetch\_related() call. I advise using tools such as [Django Debug Toolbar](https://django-debug-toolbar.readthedocs.io/en/stable/), using real data, to determine what makes the most sense for your use.

There is another thing that you should be aware of when encapsulating queries involving one-to-many or many-to-many relationships. You may see code like this

**def** some\_expensive\_calculation**(**self**):**

related\_objs **=** RelatedModel**.**objects**.**filter**(**main**=**self**)**

**...**

This code, as we should now be able to see, is an anti-pattern that will *always* issue a query when called from a MainModel item, regardless of whatever optimizations have been used on the QuerySet which obtained the MainModel in the first place. It would be better to do this instead

**def** some\_expensive\_calculation**(**self**):**

related\_objs **=** self**.**many**.**all**()**

**...**

That way, if we have calling code that does this

**for** item **in** models**.**MainModel**.**objects**.**prefetch\_related**(**'many'**):**

result **=** item**.**some\_expensive\_calculation**()**

**...**

we should only get the two queries we expect, not one for the main set plus one each for however many items are in that set.

So now we've seen that the QuerySets that you use in your apps can have significant real-world performance implications. However, with some care and understanding of the simple concepts behind Django's QuerySets, you can improve your code and become a better Django developer. But more than that, I hope that you take away from this article the realization that you shouldn't be afraid to read Django's source code to see how something works, or to build minimal working examples or simple tools to explore problems within the Django shell.

[Read more Django posts on the Caktus blog.](https://www.caktusgroup.com/blog/search/?q=django)