Django Prefetch Utils

Release 0.2.0

Jan 20, 2022

Contents

Install using pip:

```
pip install django-prefetch-utils
```

Add `django_prefetch_utils` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
   "django_prefetch_utils",
   ...
]
```

To use the identity map `prefetch_related_objects` implementation globally, provide the `PREFETCH_UTILS_DEFAULT_IMPLEMENTATION` setting:

```
PREFETCH_UTILS_DEFAULT_IMPLEMENTATION = (
    'django_prefetch_utils.identity_map.prefetch_related_objects'
)
```

See *Identity Map Usage* for more ways to use this library.

CHAPTER 1

Descriptors

This library provides a number of classes which allow the user to define relationships between models that Django does not provide support for out of the box. Importantly, all of the related objects are able to be prefetched for these relationships.

We'll take a look at the descriptors provided.

1.1 Basic descriptors

One of the simplest uses of these descriptors is when the relationship between the objects can be specified by a Django "lookup" which gives the path from the model we want to prefetch to the model where we're adding the descriptor:

```
class Author(models.Model):
    class Meta:
        ordering = ('name',)
    name = models.CharField(max_length=128)

class Book(models.Model):
    authors = models.ManyToManyField(
        Author,
        models.CASCADE,
        related_name='books'
    )

class Reader(models.Model):
    books_read = models.ManyToManyField(Book, related_name='read_by')
    authors_read = RelatedQuerySetDescriptorViaLookup(
       Author,
       'books__read_by'
    )
```

which allows us to do:

```
>>> reader = Reader.objects.prefetch_related('authors_read').first()
>>> len({author.name for author in reader.authors_read.all()})  # no queries done
10
```

In the case where there's just a single related object, we can use *RelatedSingleObjectDescriptorViaLookup* instead:

```
class Reader(models.Model):
   ...
    first_author_read = RelatedSingleObjectDescriptorViaLookup(
       Author,
       'books__read_by'
    )
```

This allows us to do:

```
>> reader = Reader.objects.prefetch_related('first_author_read').first()
>> reader.first_author_read.name  # no queries done
'Aaron Adams'
```

These can also come in useful to define relationships that span databases. For example, suppose we were to store our `Dog` and `Toy` models in separate databases. We can add a descriptor to `Dog.toys` to get the behavior as if `Toy.dog_id` had been a `ForeignKey`:

```
class Toy(models.Model):
    # Stored in database #1.  We can't use a ForeignKey to Dog
    # since the table for that model is in a separate database.
    dog_id = models.PositiveIntegerField()
    name = models.CharField(max_length=32)

class Dog(models.Model):
    # Stored in database #2
    name = models.CharField(max_length=32)

    # We can use a descriptor to get the same behavior as if
    # we had the reverse relationship from a ForeignKey
    toys = RelatedQuerySetDescriptorViaLookup(Toy, 'dog_id')
```

1.2 Equal fields

We sometimes have relationships between models which are necessarily defined by foreign key relationships. For example, consider the case where we have models for people and books, and they both have a column corresponding to a year:

```
class Book(models.Model):
    published_year = models.IntegerField()

class Person(models.Model):
    birth_year = models.IntegerField()
```

If we want to efficiently get all of the books published in the same year that the person is born, we can use the *EqualFieldsDescriptor* to define that relationship:

```
class Person(models.Model):
    birth_year = models.IntegerField()
    books_from_birth_year = EqualFieldsDescriptor(
        Book,
        [('birth_year', 'published_year')]
    )
```

Then we're able to do things like:

```
>>> person = Person.objects.prefetch_related('books_from_birth_year').first()
>>> Person.books_from_birth_year.count()  # no queries are done
3
```

1.3 Top child descriptor

In a situation with a one-to-many relationship (think parent / child), we are often interested in the first child under some ordering. For example, let's say we had a message thread (the parent) with many messages (the children) and we want to be able to efficiently fetch the most recent message. Then, we can do that with *TopChildDescriptorFromField*:

```
class MessageThread(models.Model):
    most_recent_message = TopChildDescriptorFromField(
        'my_app.Message.thread',
        order_by=('-added',)
    )

class Message(models.Model):
    added = models.DateTimeField(auto_now_add=True, db_index=True)
    thread = models.ForeignKey(MessageThread, on_deleted=models.PROTECT)
    text = models.TextField()
```

Then, we're able to do things like:

```
>>> thread = MessageThread.objects.prefetch_related('most_recent_message').first()
>>> thread.most_recent_message.text  # no queries are done
'Talk to you later!'
```

If the one-to-many relationship is given by a generic foreign key, then we can use *TopChildDescriptorFromGenericRelation* instead.

1.4 Annotated Values

In addition to being able to prefetch models, we can use the *AnnotationDescriptor* to prefetch values defined by an annotation on a queryset.

For example, let's say we're say interested in computing the number of

in a value which can be computed as an annotation on a queryset, but we'll also want to be able access that same value on a model even if that model did not come from a queryset which included that annotation:

```
from django.db import models
from django_prefetch_utils.descriptors import AnnotationDescriptor
```

(continues on next page)

(continued from previous page)

```
class Toy(models.Model):
    dog = models.ForeignKey('dogs.Dog')
    name = models.CharField(max_length=32)

class Dog(models.Model):
    name = models.CharField(max_length=32)
    toy_count = AnnotationDescriptor(models.Count('toy_set'))
```

```
>>> dog = Dog.objects.first()
>>> dog.toy_count
11
>>> dog = Dog.objects.prefetch_related('toy_count').first()
>>> dog.toy_count  # no queries are done
11
```

See *AnnotationDescriptor* for more information.

1.5 Generic base classes

If the functionality of the above classes isn't enough, then we can make use of the generic base classes to easily define custom desciptors which support prefetching. *GenericPrefetchRelatedDescriptor* is the abstract base class which we need to subclass. It has a number of abstract methods which need to be implemented:

- `get_prefetch_model_class()`: this needs to return the model class for the objects which are being prefetched.
- `filter_queryset_for_instances()`: this takes in a *queryset* for the models to be prefetched along with *instances* of the model on which the descriptor is found; it needs to return that *queryset* filtered to the objects which are related to the provided *instances*.
- `get_join_for_instance()`: this takes in an *instance* of the model on which the descriptor is found and returns a value used match it up with the prefetched objects.
- `get_join_value_for_related_obj()`: this takes in a prefetched object and returns a value used to match it up with the *instances* of the original model.

If we're only interested in a single object, then we can include `GenericSinglePrefetchRelatedDescriptorMixin` into our class. This will make it so that when we access the descriptor, we get the the object directly rather than a manager.

CHAPTER 2

Identity map

This library currently provides a replacement implementation of `prefetch_related_objects` which uses an identity map to automatically reduce the number of queries performed when prefetching.

For example, considered the following data model:

```
class Toy(models.Model):
    dog = models.ForeignKey('dogs.Dog')

class Dog(models.Model):
    name = models.CharField()
    favorite_toy = models.ForeignKey('toys.Toy', null=True)
```

With this library, we get don't need to do a database query to perform the prefetch for `favorite_toy` since that object had already been fetched as part of the prefetching for `toy_set`:

```
>>> dog = Dog.objects.prefetch_related('toys', 'favorite_toy')[0]
SELECT * from dogs_dog limit 1;
SELECT * FROM toys_toy where toys_toy.dog_id IN (1);
>>> dog.favorite_toy is dog.toy_set.all()[0]  # no queries done
True
```

2.1 Identity Map Usage

The *`django_prefetch_utils.identity_map.prefetch_related_objects()`* implementation use an identity map to provide a number of benefits over Django's default. See *Comparison with default implementation* for a discussion of the improvements. It should be a drop-in replacement, requiring no changes of user code.

2.1.1 Using the identity map globally

The easiest way to use the identity map implementation is to set the `PREFETCH_UTILS_DEFAULT_IMPLEMENTATION` setting:

```
PREFETCH_UTILS_DEFAULT_IMPLEMENTATION = (
    'django_prefetch_utils.identity_map.prefetch_related_objects'
)
```

This will make it so that all calls to `django.db.models.query.prefetch_related_objects` will use the identity map implementation.

If at any point you which to use Django's default implementation, you can use the *`use_original_prefetch_related_objects()`* context decorator:

```
from from django_prefetch_utils.selector import use_original_prefetch_related_objects

@use_original_prefetch_related_objects()
def some_function():
    return Dog.objects.prefetch_related("toys")[0]  # uses default impl.
```

2.1.2 Using the identity map locally

The *`use_prefetch_identity_map()`* context decorator can be used if you want to use identity map implementation without using it *globally*:

```
@use_prefetch_identity_map()
def some_function():
    return Dog.objects.prefetch_related('toys')[0]  # uses identity map impl.
```

2.1.3 Persisting the identity map across calls

There may be times where you want to use the same identity map across different calls to `prefetch_related_objects`. In that case, you can use the *`use_persistent_prefetch_identity_map()`*:

```
def some_function():
    with use_persistent_prefetch_identity_map() as identity_map:
        dogs = list(Dogs.objects.prefetch_related("toys"))

    with use_persistent_prefetch_identity_map(identity_map):
        # No queries are done here since all of the toys
        # have been fetched and stored in *identity_map*
        prefetch_related_objects(dogs, "favorite_toy")
```

It can also be used as a decorator:

```
@use_persistent_prefetch_identity_map()
def some_function():
    dogs = list(Dogs.objects.prefetch_related("toys"))
```

(continues on next page)

(continued from previous page)

```
    # The toy.dog instances will be identical (not just equal)
    # to the ones fetched on the line above
    toys = list(Toy.objects.prefetch_related("dog"))
    ...

@use_persistent_prefetch_identity_map(pass_identity_map=True)
def some_function(identity_map):
    dogs = list(Dogs.objects.prefetch_related("toys"))
    toys = list(Toy.objects.prefetch_related("dog"))
    ...
```

Note that when *use_persistent_prefetch_identity_map()* is active, then `QuerySet._fetch_all` will be monkey-patched so that any objects fetched will be added to / checked against the identity map.

2.2 Comparison with default implementation

The *django_prefetch_utils.identity_map.prefetch_related_objects()* implementation provides a number of benefits over Django's default implementation.

2.2.1 Database query reduction

One benefit of Django's `prefetch_related` system vs. `select_related` is that for the same prefetch lookup, equal model instances are identical. For example:

```
>>> toy1, toy2 = Toy.objects.prefetch_related("dog")
>>> toy1.dog == toy2.dog
True
>>> toy1.dog is toy2.dog
True
>>> toy1, toy2 = Toy.objects.select_related("dog")
>>> toy1.dog is toy2.dog
False
```

If for example, there is a `cached_property` on a the `Dog` model, then that would end up being shared by both `Toy` instances.

Now, consider a model like:

```
class Dog(models.Model):
    toys = models.ManyToManyField(Toy)
    favorite_toy = models.ForeignKey(Toy, null=True)
```

If we prefetch the toys and favorite toy, there will be two `Toy` objects which are equal but not identical. We get the following behavior with Django's default implementation:

```
>>> dog = Dog.objects.prefetch_related("toys", "favorite_toy")[0]
>>> only_toy = dog.toys.all()[0]
>>> only_toy == dog.favorite_toy
True
>>> only_toy is dog.favorite_toy
False
```

The identity map implementation keeps track of all of the objects fetched during the the process so that it can reuse them when possible . If we were to run the same code as above with the identity map implementation, we would have:

```
>>> only_toy is dog.favorite_toy
True
```

Additionally, since `favorite_toy` was already fetched when `toys` was prefetched, **less database queries are done**. The same code is executed with 2 database queries instead of 3.

2.2.2 Prefetch composition

One consequence of Django's default implementation of `prefetch_related` is that there are cases where it will silently not perform a requested prefetch. For example:

```
>>> toy_qs = Toy.objects.prefetch_related(
...     Prefetch("dog", queryset=Dog.objects.prefetch_related("owner"))
... )
>>> dog = Dog.objects.prefetch_related(
...     Prefetch("toy_set", queryset=toy_qs)
... )[0]
>>> toy = dog.toy_set.all()[0]
>>> toy.dog is dog
True
```

If we access `dog.owner`, then a database query is done even though it looks like we requested that it be prefetched. This happens because when the `dog` object is already set by the reverse relation when `toy_set__dog` is prefetched. Therefore, the `Dog.objects.prefetch_related("owner")` queryset is never taken into account. This makes it difficult programmatically compose querysets with prefetches inside other `Prefetch` objects.

`django_prefetch_utils.identity_map.prefetch_related_objects()` is implemented in a way does not ignore prefetches in cases like the above.

CHAPTER 3

Reference

3.1 django_prefetch_utils.descriptors

This module is provides a number of classes to help writing descriptors which play nicely with Django's `prefetch_related` system. A general guide to descriptors can be found in the Python documentation.

3.1.1 Base

class GenericPrefetchRelatedDescriptor

cache_name
Returns the dictionary key where the associated queryset will be stored on `instance._prefetched_objects_cache` after prefetching.

Return type str

contribute_to_class(*cls*, *name*)
Sets the name of the descriptor and sets itself as an attribute on the class with the same name.

This method is called by Django's `django.db.models.base.Modelbase` with the class the descriptor is defined on as well as the name it is being set up.

Returns `None`

filter_queryset_for_instances(*queryset*, *instances*)
Given a *queryset* for the related objects, returns that queryset filtered down to the ones related to *instance*.

Returns a queryset

get_join_value_for_instance(*instance*)
Returns the value used to associate *instance* with its related objects.

Parameters **instance** – an instance of `model`

get_join_value_for_related_obj(*rel_obj*)
Returns the value used to associate *rel_obj* with its related instance.

Parameters **rel_obj** – a related object

get_prefetch_model_class()
Returns the model class of the objects that are prefetched by this descriptor.

Returns subclass of `django.db.models.Model`

get_queryset(*queryset=None*)
Returns the default queryset to use for the related objects.

The purpose of taking the optional *queryset* parameter is so that a custom queryset can be passed in as part of the prefetching process, and any subclasses can apply their own filters to that.

Parameters **queryset** (*QuerySet*) – an optional queryset to use instead of the default queryset for the model

Return type `django.db.models.QuerySet`

manager_class
alias of *GenericPrefetchRelatedDescriptorManager*

update_queryset_for_prefetching(*queryset*)
Returns *queryset* updated with any additional changes needed when it is used as a queryset within `get_prefetch_queryset`.

Parameters **queryset** (*QuerySet*) – the queryset which will be returned as part of the `get_prefetch_queryset` method.

Return type `django.db.models.QuerySet`

class GenericPrefetchRelatedDescriptorManager(*descriptor*, *instance*)
A `django.db.models.Manager` to be used in conjunction with `RelatedQuerySetDescriptor`.

cache_name
Returns the name used to store the prefetched related objects.

Return type str

get_prefetch_queryset(*instances*, *queryset=None*)
This is the primary method used by Django's prefetch system to get all of the objects related to *instances*.

Parameters

- **instances** (*list*) – a list of instances of the class where this descriptor appears
- **queryset** – an optional queryset

Returns the 5-tuple needed by Django's prefetch system.

get_queryset()
Returns a queryset of objects related to `instance`. This method checks to see if the queryset has been cached in `instance._prefetched_objects_cache`.

Return type `django.db.models.QuerySet`

3.1.2 Via Lookup

class RelatedQuerySetDescriptorViaLookup(*prefetch_model*, *lookup*)
This provides a descriptor for access to related objects where the relationship between the instances on which this descriptor is defined and the related objects can be specified by a Django "lookup":

```
>>> class Author(models.Model):
...     pass
...
>>> class Book(models.Model):
...     authors = models.ManyToManyField(Author, related_name='books')
...
>>> class Reader(models.Model):
...      books_read = models.ManyToManyField(Book, related_name='read_by')
...      authors_read = RelatedQuerySetDescriptorViaLookupBase(
...          Author, 'books__read_by'
...      )
...
>>> reader = Reader.objects.prefetch_related('authors_read').first()
>>> reader.authors_read.count()  # no queries
42
```

The lookup specifies the path from the related object to the model on which the descriptor is defined.

get_prefetch_model_class()
Returns the model class of the objects that are prefetched by this descriptor.

Returns subclass of `django.db.models.Model`

lookup
Returns the Django lookup string which describes the relationship from the related object to the one on which this descriptor is defined.

Return type str

class RelatedQuerySetDescriptorViaLookupBase
This is a base class for descriptors which provide access to related objects where the relationship between the instances on which this descriptor is defined and the related objects can by specified by a Django "lookup" which specifies the path from the related object to the model on which the descriptor is defined.

filter_queryset_for_instances(*queryset*, *instances*)
Returns *queryset* filtered to the objects which are related to *instances*. If *queryset* is `None`, then `get_queryset()` will be used instead.

Parameters

- **instances** (*list*) – instances of the class on which this descriptor is found
- **queryset** (*QuerySet*) – the queryset to filter for *instances*

Return type `django.db.models.QuerySet`

get_join_value_for_instance(*instance*)
Returns the value used to join the *instance* with the related object. In this case, it is the primary key of the instance.

Return type int

get_join_value_for_related_obj(*related_obj*)
Returns the value used to join the *related_obj* with the original instance. In this case, it is the primary key of the instance.

Return type int

lookup
Returns the Django lookup string which describes the relationship from the related object to the one on which this descriptor is defined.

Return type str

obj_pk_annotation
Returns the name of an annotation to be used on the queryset so that we can easily get the primary key for the original object without having to instantiate any intermediary objects.

Return type str

update_queryset_for_prefetching(*queryset*)
Returns an updated *queryset* for use in `get_prefetch_queryset`.

We need to add an annotation to the queryset so that know which related model to associate with which original instance.

Parameters **queryset** (*QuerySet*) – the queryset which will be returned as part of the `get_prefetch_queryset` method.

Return type `django.db.models.QuerySet`

class RelatedSingleObjectDescriptorViaLookup(*prefetch_model*, *lookup*)
This provides a descriptor for access to a related object where the relationship to the instances on which this descriptor is defined and the related objects can be specified by a Django "lookup":

```
>>> class Author(models.Model):
...     pass
...
>>> class Book(models.Model):
...     authors = models.ManyToManyField(Author, related_name='books')
...
>>> class Reader(models.Model):
...      books_read = models.ManyToManyField(Book, related_name='read_by')
...      some_read_author = RelatedSingleObjectDescriptorViaLookup(
...          Author, 'books__read_by'
...      )
...
>>> reader = Reader.objects.prefetch_related('some_read_author').first()
>>> reader.some_read_author  # no queries
<Author: Jane>
```

The lookup specifies the path from the related object to the model on which the descriptor is defined.

3.1.3 Annotation

class AnnotationDescriptor(*annotation*)
This descriptor behaves like an annotated value would appear on a model. It lets you turn an annotation into a prefetch at the cost of an additional query:

```
>>> class Author(models.Model):
...     book_count = AnnotationDescriptor(Count('books'))
...
authors.models.Author
>>> author = Author.objects.get(name="Jane")
>>> author.book_count
11
>>> author = Author.objects.prefetch_related('book_count').get(name="Jane")
>>> author.book_count  # no queries done
11
```

It works by storing a `values_list` tuple containing the annotated value on *`cache_name`* on the object.

cache_name
Returns the name of the attribute where we will cache the annotated value. We are overriding `cache_name` from `GenericPrefetchRelatedDescriptor` so that we can just return the annotated value from `__get__`.

Return type str

filter_queryset_for_instances(*queryset*, *instances*)
Returns *queryset* filtered to the objects which are related to *instances*.

Parameters

- **instances** (*list*) – instances of the class on which this descriptor is found
- **queryset** (*QuerySet*) – the queryset to filter for *instances*

Return type `django.db.models.QuerySet`

get_join_value_for_instance(*instance*)
Returns the value used to associate *instance* with its related objects.

Parameters **instance** – an instance of `model`

get_join_value_for_related_obj(*annotation_value*)
Returns the value used to associate *rel_obj* with its related instance.

Parameters **rel_obj** – a related object

get_prefetch_model_class()
Returns the model class of the objects that are prefetched by this descriptor.

Returns subclass of `django.db.models.model`

3.1.4 Top Child

class TopChildDescriptor
An abstract class for creating prefetchable descriptors which correspond to the top child in a group of children associated to a parent model.

For example, consider a descriptor for the most recent message in a conversation. In this case, the children would be the messages, and the parent would be the conversation. The ordering used to determine the "top child" would be `-added`.

filter_queryset_for_instances(*queryset*, *instances*)
Returns a `QuerySet` which returns the top children for each of the parents in *instances*.

Note: This does not filter the set of child models which are included in the "consideration set". To do that, please override *get_child_filter_args()* and *get_child_filter_kwargs()*.

Parameters

- **queryset** (*QuerySet*) – the queryset of child objects to filter for *instances*
- **instances** (*list*) – a list of the parent models whose children we want to fetch.

Return type `django.db.models.QuerySet`

get_child_filter_args()
returns a tuple of all of the argument filters which should be used to filter the possible children returned.

Return type tuple

get_child_filter_kwargs(***kwargs*)
returns a dictionary of all of the keyword argument filters which should be used to filter the possible children returned.

Parameters **kwargs** (*dict*) – any overrides for the default filter

Return type dict

get_child_model()
returns the `model` class for the children.

get_child_order_by()
returns a tuple which will be used to place an ordering on the children so that we can return the "top" one.

Return type tuple

get_join_value_for_instance(*parent*)
Returns the value used to associate the *parent* with the child fetched during the prefetching process. In this case, it is the primary key of the parent.

Return type int

get_join_value_for_related_obj(*child*)
Returns the value used to associate the *child* with the parent object. In this case, it is the primary key of the parent.

Return type int

get_parent_model()
returns the `model` class for the parents.

get_parent_relation()
returns the string which specifies how to associate a parent model to a child.

for example, if the parent were `common.models.user` and the child were `services.models.service`, then this should be `'provider__user'`.

Return type str

get_prefetch_model_class()
Returns the model class of the objects that are prefetched by this descriptor.

Returns subclass of `django.db.models.model`

get_subquery()
returns a `queryset` for all of the child models which should be considered.

Return type `queryset`

get_top_child_pks(*parent_pks*)
Returns a queryset for the primary keys of the top children for the parent models whose primary keys are in *parent_pks*.

Parameters **parent_pks** (*list*) – a list of primary keys for the parent models whose children we want to fetch.

Return type `QuerySet`

parent_pk_annotation
Returns the name of the attribute which will be annotated on child instances and will correspond to the primary key of the associated parent.

Return type str

class TopChildDescriptorFromField(*field*, *order_by*)

get_child_field()
Returns the field on the child model which is a foreign key to the parent model.

Return type `django.db.models.fields.Field`

get_child_order_by()
returns a tuple which will be used to place an ordering on the children so that we can return the "top" one.

Return type tuple

class TopChildDescriptorFromFieldBase
A subclass of *TopChildDescriptor* for use when the children are related to the parent by a foreign key. In that case, anyone implementing a subclass of this only needs to implement *get_child_field()*.

get_child_field()
Returns the field on the child model which is a foreign key to the parent model.

Return type `django.db.models.fields.Field`

get_child_model()
returns the `model` class for the children.

get_parent_model()
returns the `model` class for the parents.

get_parent_relation()
returns the string which specifies how to associate a parent model to a child.

for example, if the parent were `common.models.user` and the child were `services.models.service`, then this should be `'provider__user'`.

Return type str

class TopChildDescriptorFromGenericRelation(*generic_relation*, *order_by*)
For further customization,

get_child_field()
Returns the generic relation on the parent model for the children.

Return type `django.contrib.contenttypes.fields.GenericRelation`

get_child_order_by()
returns a tuple which will be used to place an ordering on the children so that we can return the "top" one.

Return type tuple

class TopChildDescriptorFromGenericRelationBase
A subclass of *TopChildDescriptor* for use when the children are described by a `django.contrib.contenttypes.fields.GenericRelation`.

apply_content_type_filter(*queryset*)
Filters the (child) *queryset* to only be those that correspond to *content_type*.

Return type `django.db.models.QuerySet`

content_type
Returns the content type of the parent model.

get_child_field()
Returns the generic relation on the parent model for the children.

Return type `django.contrib.contenttypes.fields.GenericRelation`

get_child_model()
Returns the `Model` class for the children.

get_parent_model()
Returns the `Model` class for the parent.

get_parent_relation()
Returns the name of the field on the child corresponding to the object primary key.

Return type str

get_queryset(*queryset=None*)
Returns a `QuerySet` which returns the top children for each of the parents who have primary keys in *parent_pks*.

Return type `django.db.models.QuerySet`

get_subquery()
Returns a `QuerySet` for all of the child models which should be considered.

Return type `django.db.models.QuerySet`

3.1.5 Equal Fields

class EqualFieldsDescriptor(*related_model*, *join_fields*)
A descriptor which provides a manager for objects which are related by having equal values for a series of columns:

```
>>> class Book(models.Model):
...     title = models.CharField(max_length=32)
...     published_year = models.IntegerField()
>>> class Author(models.Model):
...     birth_year = models.IntegerField()
...     birth_books = EqualFieldsDescriptor(Book, [('birth_year', 'published_year
↪')])
...
>>> # Get the books published in the year the author was born
>>> author = Author.objects.prefetch_related('birth_books')
>>> author.birth_books.count()  # no queries are done here
10
```

filter_queryset_for_instances(*queryset*, *instances*)
Returns a `QuerySet` which returns the top children for each of the parents in *instances*.

Parameters **queryset** (*QuerySet*) – a queryset for the objects related to *instances*

Return type `django.db.models.QuerySet`

get_join_value_for_instance(*instance*)
Returns a tuple of the join values for *instance*.

Return type tuple

get_join_value_for_related_obj(*rel_obj*)
Returns a tuple of the join values for *rel_obj*.

Return type tuple

get_prefetch_model_class()
Returns the model class of the objects that are prefetched by this descriptor.

Returns subclass of `django.db.models.model`

preprocess_join_fields(*join_fields*)

Returns a list of `_FieldMapping` objects.

3.2 django_prefetch_utils.selector

This module provides utilities for changing the implementation of `prefetch_related_objects` that Django uses. In order for these to work, `enable_fetch_related_objects_selector()` must be called. This will be done in `AppConfig.ready` if `django_prefetch_utils` is added to `INSTALLED_APPS`.

Once that has been called, then *set_default_prefetch_related_objects()* can be called to override the default implementation globally:

```
from django_prefetch_related.selector import set_default_prefetch_related_objects
from django_prefetch_utils.identity_map import prefetch_related_objects

set_default_prefetch_related_objects(prefetch_related_objects)
```

This will be done as part of `AppConfig.ready` if the `PREFETCH_UTILS_DEFAULT_IMPLEMENTATION` setting is provided.

To change the implementation used on a local basis, the *override_prefetch_related_objects()* or *use_original_prefetch_related_objects()* context decorators can be used:

```
from django_prefetch_utils.identity_map import prefetch_related_objects

@use_original_prefetch_related_objects()
def some_function():
    dogs = list(Dog.objects.all())  # uses Django's implementation

    with override_prefetch_related_objects(prefetch_related_objects):
        toys = list(Toy.objects.all)  # uses identity map implementation
```

disable_prefetch_related_objects_selector()
Changes `django.db.models.query.prefetch_related_objects` to Django's original implementation of `prefetch_related_objects`.

enable_prefetch_related_objects_selector()
Changes `django.db.models.query.prefetch_related_objects` to an implemention which allows thread-local overrides.

get_prefetch_related_objects()
Returns the active implementation of `prefetch_related_objects`:

```
>>> from django_prefetch_utils.selector import get_prefetch_related_objects
>>> get_prefetch_related_objects()
<function django.db.models.query.prefetch_related_objects>
```

Returns a function

class override_prefetch_related_objects(*func*)
This context decorator allows one to chnage the implementation of `prefetch_related_objects` to be *func*.

When the context manager or decorator exits, the implementation will be restored to its previous value.

```
with override_prefetch_related_objects(prefetch_related_objects):
    dogs = list(Dog.objects.prefetch_related('toys'))
```

Note: This requires *enable_prefetch_related_objects_selector()* to be run before the changes are able to take effect.

remove_default_prefetch_related_objects()
Removes a custom default implementation of `prefetch_related_objects`:

```
>>> set_default_prefetch_related_objects(some_implementation)
>>> get_prefetch_related_objects()
<function some_implementation>
>>> remove_default_prefetch_related_objects()
>>> get_prefetch_related_objects()
<function django.db.models.query.prefetch_related_objects>
```

set_default_prefetch_related_objects(*func*)
Sets the default implementation of `prefetch_related_objects` to be *func*:

```
>>> get_prefetch_related_objects()
<function django.db.models.query.prefetch_related_objects>
>>> set_default_prefetch_related_objects(some_implementation)
>>> get_prefetch_related_objects()
<function some_implementation>
```

class use_original_prefetch_related_objects
This context decorator allows one to force the `prefetch_related_objects` implementation to be Django's default implementation:

```
with use_original_prefetch_related_objects():
    dogs = list(Dog.objects.prefetch_related('toys'))
```

3.3 django_prefetch_utils.identity_map

get_default_prefetch_identity_map()
Returns an empty default identity map for use during prefetching.

Return type *django_prefetch_utils.identity_map.maps.PrefetchIdentityMap*

get_prefetched_objects_from_list(*obj_list*, *through_attr*)
Returns all of the related objects in *obj_list* from *through_attr*.

Return type list

get_prefetcher(*obj_list*, *through_attr*, *to_attr*)
For the attribute *through_attr* on the given instance, finds an object that has a `get_prefetch_queryset()`.

Returns a 4 tuple containing:

- (the object with get_prefetch_queryset (or None),
- the descriptor object representing this relationship (or None),
- a boolean that is False if the attribute was not found at all,

- a list of the subset of *obj_list* that requires fetching

prefetch_related_objects(**args*, ***kwargs*)
Calls *prefetch_related_objects_impl()* with a new identity map from *get_default_prefetch_identity_map()*:

```
>>> from django_prefetch_utils.identity_map import prefetch_related_objects
>>> dogs = list(Dogs.objectss.all())
>>> prefetch_related_objects(dogs, 'toys')
```

Note: This will create will not preserve the identity map across different calls to `prefetched_related_objects`. For that, you need to use *django_prefetch_utils.identity_map.persistent.use_persistent_prefetch_identity_map()*

prefetch_related_objects_impl(*identity_map*, *model_instances*, **related_lookups*)
An implementation of `prefetch_related_objects` which makes use of *identity_map* to keep track of all of the objects which have been fetched and reuses them where possible.

use_prefetch_identity_map()
A context decorator which enables the identity map version of `prefetch_related_objects`:

```
with use_prefetch_identity_map():
    dogs = list(Dogs.objects.prefetch_related('toys'))
```

Note: A new identity map is created and used for each call of `prefetched_related_objects`.

3.3.1 Persistent Identity Map

class FetchAllDescriptor
This descriptor replaces `QuerySet._fetch_all` and applies an identity map to any objects fetched in a queryset.

disable_fetch_all_descriptor()
Sets `QuerySet._fetch_all` to be the original method.

enable_fetch_all_descriptor()
Replaces `QuerySet._fetch_all` with an instance of *FetchAllDescriptor*.

class use_persistent_prefetch_identity_map(*identity_map=None*, *pass_identity_map=False*)
A context decorator which allows the same identity map to be used across multiple calls to `prefetch_related_objects`.

```
with use_persistent_prefetch_identity_map():
    dogs = list(Dogs.objects.prefetch_related("toys"))

    # The toy.dog instances will be identitical (not just equal)
    # to the ones fetched on the line above
    with self.assertNumQueries(1):
        toys = list(Toy.objects.prefetch_related("dog"))
```

3.3.2 Maps

class ExtraIdentityMap(*extra*, *wrapped*)
This identity map wrapper

class PrefetchIdentityMap
This class represents an identity map used to help ensure that equal Django model instances are identical during the prefetch process.

```
>>> identity_map = PrefetchIdentityMap()
>>> a = Author.objects.first()
>>> b = Author.objects.first()
>>> a is b
False
>>> identity_map[a] is a
True
>>> identity_map[b] is a
True
```

It is implemented as a defaultdictionary whose keys correspond the to types of Django models and whose values are a `weakref.WeakValueDictionary` mapping primary keys to the associated Django model instance.

get_map_for_model(*model*)
Returns the the underlying dictionary

Return type `weakref.WeakValueDictionary`

class RelObjAttrMemoizingIdentityMap(*rel_obj_attr*, *wrapped*)
A wrapper for an identity map which provides a `rel_obj_attr()` to be returned from a `get_prefetch_queryset` method.

This is useful for cases when there is identifying information on the related object returned from the prefetcher which is not present on the equivalent object in the identity map.

3.3.3 Wrappers

class ForwardDescriptorPrefetchQuerySetWrapper(*identity_map*, *field*, *instances_dict*, *prefix*, *queryset*)

class ForwardDescriptorPrefetchWrapper(*identity_map*, *wrapped*)

class GenericForeignKeyPrefetchQuerySetWrapper(*identity_map*, *prefix*, *queryset*)
This wrapper yields the contents of `_self_prefix` before yielding the contents of the wrapped object.

class GenericForeignKeyPrefetchWrapper(*identity_map*, *wrapped*)

class IdentityMapIteratorWrapper(*identity_map*, *wrapped*)
This is a wrapper around an iterator which applies an identity map to each of the items returned.

class IdentityMapObjectProxy(*identity_map*, *wrapped*)
A generic base class for any wrapper which needs to have access to an identity map.

class IdentityMapPrefetchQuerySetWrapper(*identity_map*, *queryset*)
A generic wrapper for the `rel_qs` queryset returned by `get_prefetch_queryset` methods.

Subclasses should implement the `__iter__()` method to customize the behavior when the queryset is iterated over.

class IdentityMapPrefetcher(*identity_map*, *wrapped*)
A wrapper for any object which has a `get_prefetch_queryset` method.

class ManyToManyPrefetchQuerySetWrapper(*identity_map*, *queryset*, *rel_obj_attr*)

class ManyToManyRelatedManagerWrapper(*identity_map*, *wrapped*)

class ReverseManyToOneDescriptorPrefetchWrapper(*identity_map*, *wrapped*)

class ReverseManyToOnePrefetchQuerySetWrapper(*identity_map*, *field*, *instances_dict*, *queryset*)

class ReverseOneToOneDescriptorPrefetchWrapper(*identity_map*, *wrapped*)

class ReverseOneToOnePrefetchQuerySetWrapper(*identity_map*, *related*, *instances_dict*, *queryset*)

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

django-prefetch-utils could always use more documentation, whether as part of the official *django-prefetch-utils* docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/roverdotcom/django-prefetch-utils/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

4.4 Development

To set up *django-prefetch-utils* for local development:

1. Fork django-prefetch-utils (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/django-prefetch-utils.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 5

Authors

- Mike Hansen - https://github.com/mwhansen/

CHAPTER 6

Changelog

6.1 0.2.0 (2022-01-12)

- Added library of descriptors for defining relationships of Django models which can be prefetched.
- Add support for the latest version of Django and Python.
- Removed support for Python 2 and unsupported Django versions.
- Updated backport of prefetch_related_objects to latest version from Django 4.0.

6.2 0.1.0 (2019-07-16)

- First release on PyPI.

CHAPTER 7

Overview

docs	
tests	
package	

This library provides a number of utilities for working with and extending Django's `prefetch_related` system. Currently, it consists of:

- a collection of descriptors to define relationships between models which support prefetching
- a new implementation of `prefetch_related_objects` which supports an identity map so that multiple copies of the same object are not fetched multiple times.
- Free software: BSD 3-Clause License

7.1 Installation

```
pip install django-prefetch-utils
```

7.2 Documentation

https://django-prefetch-utils.readthedocs.io/

7.3 Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

G