Django Delayed Union

Release 0.1.5

Apr 13, 2020

Contents

CHAPTER 1

Overview

docs	
tests	
package	

`django-delayed-union` is library designed to workaround some drawbacks with Django's union, intersection, and difference operations. In particular, once one of these operations is performed, certain methods on the queryset will silently not work:

```
>>> qs = User.objects.filter(id=1)
>>> unioned_qs = qs.union(qs)
>>> should_be_empty_qs = unioned_qs.exclude(id=1)
>>> user, = list(should_be_empty_qs); user.id
1
```

In order to work around this, `django-delayed-union` provides wrappers around a collection of querysets. These wrappers implement a similar interface to `QuerySet`, and delay performing the union, intersection, or difference operations until they are needed:

```
>>> from django_delayed_union import DelayedUnionQuerySet
>>> qs = User.objects.filter(id=1)
>>> unioned_qs = DelayedUnionQuerySet(qs, qs)
>>> empty_qs = unioned_qs.exclude(id=1)
>>> list(empty_qs)
[]
```

Operations which would typically return a new `QuerySet` instead return a new `DelayedQuerySet` with the operation applied to its collection of querysets.

One example of where this code has been useful with is when the the MySQL query planner has chosen an inefficient query plan for the queryset of a Django REST Framework view which used an `OR` condition. By using `DelayedUnionQuerySet`, subclasses could perform additional filters on the queryset while still maintaining the efficient query plan.

- Free software: BSD 3-Clause License

1.1 Installation

```
pip install django-delayed-union
```

1.2 Documentation

https://django-delayed-union.readthedocs.io/

1.3 Development

To run the all tests run:

```
tox
```

CHAPTER 2

Installation

At the command line:

```
pip install django-delayed-union
```

CHAPTER 3

Usage

To use Django Delayed Union in a project, import the wrapper corresponding to the operation needed:

```
from django_delayed_union import DelayedUnionQuerySet
from django_delayed_union import DelayedIntersectionQuerySet
from django_delayed_union import DelayedDifferenceQuerySet
```

Then, you use them where you would use Django's `union`, `intersection`, and `difference` methods:

```
>>> qs0.union(qs1, qs2)
>>> DelayedUnionQuerySet(qs0, qs1, qs2)

>>> qs0.union(qs1, qs2, all=True)
>>> DelayedUnionQuerySet(qs0, qs1, qs2, all=True)

>>> qs0.intersection(qs1)
>>> DelayedIntersectionQuerySet(qs0, qs1)

>>> qs0.difference(qs1)
>>> DelayedDifferenceQuerySet(qs0, qs1)
```

These wrappers implement the same public interface as Django's `QuerySet` so they should be able to be used by code which expects a `QuerySet`.

Note: `DelayedQuerySet` does not subclass `QuerySet` so any code which checks for whether or not an object is an instance of `QuerySet` will not work with these wrappers.

Note: If certain methods are unimplemented or will not work, they will raise a `NotImplementedError` as opposed to silently not working.

3.1 Custom QuerySet methods

Currently, the wrappers do not handle any custom methods that may have been added to the component querysets. For example, if `qs0` and `qs1` were instances of a subclass of `QuerySet` that had an `active()` method, then the following would not work:

```
>>> DelayedUnionQuerySet(q0, qs1).active()
Traceback (most recent call last)
 ...
AttributeError
```

Where this functionality is needed, it is straightforward to make a subclass of `DelayedUnionQuerySet` using which has this behavior:

```
from django_delayed_union.base import PassthroughMethod

class MyDelayedUnionQuerySet(DelayedUnionQuerySet):
    active = PassthroughMethod()

>>> MyDelayedUnionQuerySet(qs0, qs1).active()
```

Check out the other subclasses of `django_delayed_union.base.DelayedQuerySetDescriptor` if you need the resulting method to behave differently than `PassthroughMethod`.

CHAPTER 4

Reference

4.1 django_delayed_union

class django_delayed_union.base.**DelayedQuerySet**(**querysets*, ***kwargs*)

A class used to work around some of the issues with Django's built-in support for set operations with querysets (such as `UNION`). The primary issue is that after `` `.union()`` call is made any subsequent filtering will silently fail. This class works around that issue by maintaing all of the individual querysets and not applying an operation like `.union()` until it's needed.

For example, suppose we have `qs = DelayedUnionQuerySet(qs0, qs1)`` , then running `qs = qs.filter(id=42)` will be equivalent to doing `qs = DelayedUnionQuerySet(qs0.filter(id=42), qs1.filter(id=42))`. Then, when we actually need to evaluate the queryset say by doing `obj = qs.first()`, it will return `qs0.union(qs1).first()` behind the scenes.

Subclasses need to implement the `_apply_operation()`, which performs the operation such as `.union()` that is being delayed.

aggregate(**args*, ***kwargs*)

Raises `NotImplementedError`. Documentation for *aggregate*:

Returns a dictionary containing the calculations (aggregation) over the current queryset

If args is present the expression is passed as a kwarg using the Aggregate object's default alias.

all

Returns the a new delayed queryset with `all(...)` having been called on each of the component querysets.: Documentation for *all*:

Returns a new QuerySet that is a copy of the current one. This allows a QuerySet to proxy for a model manager in some cases.

annotate(**args*, ***kwargs*)

Returns the a new delayed queryset with `annotate(...)` having been called on each of the component querysets.: Documentation for *annotate*:

Return a query set in which the returned objects have been annotated with extra data or aggregations.

as_manager(*cls*)
Returns the a new delayed queryset with `as_manager(...)` having been called on the first component queryset, while the rest remain unchanged.

bulk_create(*objs*, *batch_size=None*)
Returns the result of calling `bulk_create(...)` on the first component queryset. Documentation for *bulk_create*:

Inserts each of the instances into the database. This does *not* call save() on each of the instances, does not send any pre/post save signals, and does not set the primary key attribute if it is an autoincrement field (except if features.can_return_ids_from_bulk_insert=True). Multi-table models are not supported.

complex_filter(*filter_obj*)
Returns the a new delayed queryset with `complex_filter(...)` having been called on each of the component querysets.: Documentation for *complex_filter*:

Returns a new QuerySet instance with filter_obj added to the filters.

filter_obj can be a Q object (or anything with an add_to_query() method) or a dictionary of keyword lookup arguments.

This exists to support framework features such as 'limit_choices_to', and usually it will be more natural to use other methods.

count
Returns the output of `count(...)` after having applied the delayed operation. Documentation for *count*:

Performs a SELECT COUNT() and returns the number of records as an integer.

If the QuerySet is already fully cached this simply returns the length of the cached results set to avoid multiple SELECT COUNT(*) calls.

create(***kwargs*)
Returns the result of calling `create(...)` on the first component queryset. Documentation for *create*:

Creates a new object with the given kwargs, saving it to the database and returning the created object.

dates(*field_name*, *kind*, *order='ASC'*)
Raises `NotImplementedError`. Documentation for *dates*:

Returns a list of date objects representing all available dates for the given field_name, scoped to 'kind'.

datetimes(*field_name*, *kind*, *order='ASC'*, *tzinfo=None*)
Raises `NotImplementedError`. Documentation for *datetimes*:

Returns a list of datetime objects representing all available datetimes for the given field_name, scoped to 'kind'.

db
Return the database that will be used if this query is executed now

defer(**fields*)
Returns the a new delayed queryset with `defer(...)` having been called on each of the component querysets.: Documentation for *defer*:

Defers the loading of data for certain fields until they are accessed. The set of fields to defer is added to any existing set of deferred fields. The only exception to this is if None is passed in as the only parameter, in which case all deferrals are removed (None acts as a reset option).

delete
Returns the output of `delete(...)` after having applied the delayed operation. Documentation for *delete*:

Deletes the records in the current QuerySet.

difference(**other_qs*)
Raises `NotImplementedError`.

distinct(**field_names*)
Raises `NotImplementedError`. Documentation for *distinct*:

Returns a new QuerySet instance that will select only distinct results.

earliest(*field_name=None*)
Returns the output of `earliest(...)` after having applied the delayed operation.

exclude(**args*, ***kwargs*)
Returns the a new delayed queryset with `exclude(...)` having been called on each of the component querysets.: Documentation for *exclude*:

Returns a new QuerySet instance with NOT (args) ANDed to the existing set.

exists
Returns the output of `exists(...)` after having applied the delayed operation.

extra(*select=None*, *where=None*, *params=None*, *tables=None*, *order_by=None*, *select_params=None*)
Returns the a new delayed queryset with `extra(...)` having been called on each of the component querysets.: Documentation for *extra*:

Adds extra SQL fragments to the query.

filter(**args*, ***kwargs*)
Returns the a new delayed queryset with `filter(...)` having been called on each of the component querysets.: Documentation for *filter*:

Returns a new QuerySet instance with the args ANDed to the existing set.

first
Returns the output of `first(...)` after having applied the delayed operation. Documentation for *first*:

Returns the first object of a query, returns None if no match is found.

get(**args*, ***kwargs*)
Performs the query and returns a single object matching the given keyword arguments.

Note: We cannot use `PostApplyMethod` for this since that does additional filtering which does not work with querysets that have been "unioned" for example.

---

get_or_create(*defaults=None*, ***kwargs*)
Raises `NotImplementedError`. Documentation for *get_or_create*:

Looks up an object with the given kwargs, creating one if necessary. Returns a tuple of (object, created), where created is a boolean specifying whether an object was created.

in_bulk(*id_list=None*)
Returns a dictionary mapping each of the given IDs to the object with that ID. If *id_list* isn't provided, the entire *DelayedQuerySet* is evaluated.

intersection(**other_qs*)
Raises `NotImplementedError`.

iterator
Returns the output of `iterator(...)` after having applied the delayed operation. Documentation for *iterator*:

An iterator over the results from applying this QuerySet to the database.

last
Returns the output of `last(...)` after having applied the delayed operation. Documentation for *last*:

Returns the last object of a query, returns None if no match is found.

latest(*field_name=None*)
Returns the output of `latest(...)` after having applied the delayed operation.

model
Returns the model class for the *DelayedQuerySet*.

none
Returns the a new delayed queryset with `none(...)` having been called on each of the component querysets.: Documentation for *none*:

Returns an empty QuerySet.

only(**fields*)
Returns the a new delayed queryset with `only(...)` having been called on each of the component querysets.: Documentation for *only*:

Essentially, the opposite of defer. Only the fields passed into this method and that are not already specified as deferred are loaded immediately when the queryset is evaluated.

order_by(**field_names*)
Returns a new `DelayedQuerySet`` instance with the ordering changed.

Note: We need to have a custom implementation for this because we want to change the ordering of the final queryset, not just the ordering within each component queryset.

---

ordered
Returns True if the *DelayedQuerySet* is ordered – i.e. has an order_by() clause.

Return type bool

prefetch_related(**lookups*)
Returns the a new delayed queryset with `prefetch_related(...)` having been called on the first component queryset, while the rest remain unchanged. Documentation for *prefetch_related*:

Returns a new QuerySet instance that will prefetch the specified Many-To-One and Many-To-Many related objects when the QuerySet is evaluated.

When prefetch_related() is called more than once, the list of lookups to prefetch is appended to. If prefetch_related(None) is called, the list is cleared.

query

raw(*raw_query*, *params=None*, *translations=None*, *using=None*)
Returns the output of `raw(...)` after having applied the delayed operation.

reverse()
Reverses the ordering of the *DelayedQuerySet*.

Note: We need to have a custom implementation for this because we want to reverse the ordering of the final queryset, not just the ordering within each component queryset.

---

select_for_update(*nowait=False*, *skip_locked=False*)
Raises `NotImplementedError`. Documentation for *select_for_update*:

Returns a new QuerySet instance that will select objects with a FOR UPDATE lock.

select_related(**fields*)
Returns the a new delayed queryset with `select_related(...)` having been called on each of the component querysets.: Documentation for *select_related*:

Returns a new QuerySet instance that will select related objects.

If fields are specified, they must be ForeignKey fields and only those related objects are included in the selection.

If select_related(None) is called, the list is cleared.

union(**other_qs*, ***kwargs*)
Raises `NotImplementedError`.

update(***kwargs*)
Raises `NotImplementedError`. Documentation for *update*:

Updates all elements in the current QuerySet, setting all the given fields to the appropriate values.

update_or_create(*defaults=None*, ***kwargs*)
Raises `NotImplementedError`. Documentation for *update_or_create*:

Looks up an object with the given kwargs, updating one with defaults if it exists, otherwise creates a new one. Returns a tuple (object, created), where created is a boolean specifying whether an object was created.

using(*alias*)
Returns the a new delayed queryset with `using(...)` having been called on each of the component querysets.: Documentation for *using*:

Selects which database this QuerySet should execute its query against.

values(**fields*, ***expressions*)
Returns the a new delayed queryset with `values(...)` having been called on each of the component querysets.:

values_list(**fields*, ***kwargs*)
Returns the a new delayed queryset with `values_list(...)` having been called on each of the component querysets.:

class django_delayed_union.**DelayedUnionQuerySet**(**querysets*, ***kwargs*)
Bases: *django_delayed_union.base.DelayedQuerySet*

distinct()
Returns a new *DelayedUnionQuerySet* instance that will select only distinct results.

update(***kwargs*)
Updates all elements in the component querysets, setting all the given fields to the appropriate values. Returns the total number of (not-necessarily distinct) rows updated.

class django_delayed_union.**DelayedIntersectionQuerySet**(**querysets*)
Bases: *django_delayed_union.base.DelayedQuerySet*

distinct()
Returns a new *DelayedIntersectionQuerySet* instance that will select only distinct results.

class django_delayed_union.**DelayedDifferenceQuerySet**(**querysets*)
Bases: *django_delayed_union.base.DelayedQuerySet*

distinct()
Returns a new *DelayedDifferenceQuerySet* instance that will select only distinct results.

CHAPTER 5

Contributing

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

5.1 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.

5.2 Documentation improvements

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

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/roverdotcom/django-delayed-union/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.
- Code contributions are always welcome :)

5.4 Development

To set up *django-delayed-union* for local development:

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

```
git clone git@github.com:your_name_here/django-delayed-union.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.

5.4.1 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`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -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 6

Authors

- Mike Hansen - https://www.rover.com/

CHAPTER 7

Changelog

7.1 0.1.4 (2019-10-19)

- Added query property to delayed querysets.
- Fixed bug with count() and select_related() in MySQL
- Added tests for Django 3.0

7.2 0.1.3 (2019-04-24)

- Added tests for Django 2.2

7.3 0.1.2 (2018-12-14)

- Added support for nested unions and intersections

7.4 0.1.1 (2018-07-16)

- Cached the queryset generated after applying the delayed operation.

7.5 0.1.0 (2018-03-14)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index