django-cachalot Documentation

Release 0.9.0

Bertrand Bordage

December 14, 2014

Contents

Caches your Django ORM queries and automatically invalidates them.

Currently in beta, do not use in production

CHAPTER 1

Quick start

1.1 Requirements

- Django 1.6 or 1.7
- Python 2.6, 2.7, 3.2, 3.3, or 3.4
- a cache configured as *default* with one of these backends:
  - django-redis
  - memcached (using either python-memcached or pylibmc (but pylibmc is only supported with Django >= 1.7))
  - filebased (only with Django >= 1.7 as it was not thread-safe before)
  - locmem (but it's not shared between processes, see *Limits*)
- one of these databases:
  - PostgreSQL
  - SQLite
  - **MySQL (but you probably don't need django-cachalot in this case,** see *Limits*)

1.2 Usage

1. `pip install django-cachalot`
2. Add `'cachalot',` to your `INSTALLED_APPS`
3. Be aware of *the few limits*
4. If you use django-debug-toolbar, you can add `'cachalot.panels.CachalotPanel',` to your `DEBUG_TOOLBAR_PANELS`
5. Enjoy!

1.3 Settings

1.3.1 CACHALOT_ENABLED

Default `True`

Description If set to `False`, disables SQL caching but keeps invalidating to avoid stale cache

1.3.2 CACHALOT_CACHE

Default `'default'`

Description Alias of the cache from `CACHES` used by django-cachalot

1.3.3 CACHALOT_CACHE_RANDOM

Default `False`

Description If set to `True`, caches random queries (those with `order_by('?')`)

1.3.4 CACHALOT_INVALIDATE_RAW

Default `True`

Description If set to `False`, disables automatic invalidation on raw SQL queries – read *Raw SQL queries* for more info

1.3.5 CACHALOT_QUERY_KEYGEN

Default `'cachalot.utils.get_query_cache_key'`

Description Python module path to the function that will be used to generate the cache key of a SQL query

1.3.6 CACHALOT_TABLE_KEYGEN

Default `'cachalot.utils.get_table_cache_key'`

Description Python module path to the function that will be used to generate the cache key of a SQL table

1.3.7 Dynamic overriding

Django-cachalot is built so that its settings can be dynamically changed.

For example:

```
from django.conf import settings
from django.test.utils import override_settings

with override_settings(CACHALOT_ENABLED=False):
    # SQL queries are not cached in this block
```

```
@override_settings(CACHALOT_CACHE='another_alias')
def your_function():
    # What's in this function uses another cache

# Globally disables SQL caching until you set it back to True
settings.CACHALOT_ENABLED = False
```

CHAPTER 2

Limits

2.1 Locmem

Locmem is a just a dict stored in a single Python process. It's not shared between processes, so don't use locmem with django-cachalot in a multi-processes project, if you use RQ or Celery for instance.

2.2 MySQL

This database software already provides by default something like django-cachalot: MySQL query cache. Django-cachalot will slow down your queries if that query cache is enabled. If it's not enabled, django-cachalot will make queries much faster. But you should probably better enable the query cache instead.

2.3 Raw SQL queries

Note: Don't worry if you don't understand what follow. That probably means you don't use raw queries, and therefore are not directly concerned by those potential issues.

By default, django-cachalot tries to invalidate its cache after a raw query. It detects if the raw query contains `UPDATE`, `INSERT` or `DELETE`, and then invalidates the tables contained in that query by comparing with models registered by Django.

This is quite robust, so if a query is not invalidated automatically by this system, please *send a bug report*. In the meantime, you can use *the API* to manually invalidate the tables where data has changed.

However, this simple system can be too efficient in some cases and lead to unwanted extra invalidations. In such cases, you may want to partially disable this behaviour by *dynamically overriding settings* to set *CACHALOT_INVALIDATE_RAW* to `False`. After that, use *the API* to manually invalidate the tables you modified.

CHAPTER 3

API

Use these tools if the automatic behaviour of django-cachalot is not enough. Typically, use `invalidate_tables` or `invalidate_models` after each raw SQL query modifying the database.

CHAPTER 4

What still needs to be done

4.1 For version 1.0

- Cache raw queries
- Test multi-location caches
- Add a management command to invalidate everything

CHAPTER 5

Bug reports, questions, discussion, new features

- If you spotted **a bug**, please file a precise bug report on GitHub
- If you have **a question** on how django-cachalot works or to **simply discuss**, go to our Google group.
- If you want **to add a feature**:
  - if you have an idea on how to implement it, you can fork the project and send a pull request, but **please open an issue first**, because someone else could already be working on it
  - if you're sure that it's a must-have feature, open an issue
  - if it's just a vague idea, please ask on google groups before

CHAPTER 6

How django-cachalot works

6.1 Reverse engineering

It's a lot of Django reverse engineering combined with a strong test suite. Such a test suite is crucial for a reverse engineering project. If some important part of Django changes and breaks the expected behaviour, you can be sure that the test suite will fail.

6.2 Monkey patching

django-cachalot modifies Django in place during execution to add a caching tool just before SQL queries are executed. We detect which cache keys must be removed when some data is created/changed/deleted on the database.

CHAPTER 7

Legacy

This work is highly inspired of johnny-cache, another easy-to-use ORM caching tool! It's working with Django <= 1.5. I used it in production during 3 years, it's an excellent module!

Unfortunately, we failed to make it migrate to Django 1.6 (I was involved). It was mostly because of the transaction system that was entirely refactored.

I also noticed a few advanced invalidation issues when using `QuerySet.extra` and some complex cases implying multi-table inheritance and related `ManyToManyField`.

CHAPTER 8

What's new in django-cachalot?

8.1 0.9.0

Added:

- Caches all queries implying `Queryset.extra`
- Invalidates raw queries
- Adds a simple API containing: `invalidate_tables`, `invalidate_models`, `invalidate_all`
- Adds file-based cache support for Django 1.7
- Adds a setting to choose if random queries must be cached
- Adds 2 settings to customize how cache keys are generated
- Adds a django-debug-toolbar panel
- Adds a benchmark

Fixed:

- Rewrites invalidation for a better speed & memory performance
- Fixes a stale cache issue occurring when an invalidation is done exactly during a SQL request on the invalidated table(s)
- Fixes a stale cache issue occurring after concurrent transactions
- Uses an infinite timeout

Removed:

- Simplifies `cachalot_settings` and forbids its use or modification

8.2 0.8.1

- Fixes an issue with pip if Django is not yet installed

8.3 0.8.0

- Adds multi-database support

- Adds invalidation when altering the DB schema using *migrate*, *syncdb*, *flush*, *loaddata* commands (also invalidates South, if you use it)
- Small optimizations & simplifications
- Adds several tests

8.4 0.7.0

- Adds thread-safety
- Optimizes the amount of cache queries during transaction

8.5 0.6.0

- Adds memcached support

8.6 0.5.0

- Adds `CACHALOT_ENABLED` & `CACHALOT_CACHE` settings
- Allows settings to be dynamically overridden using `cachalot_settings`
- Adds some missing tests

8.7 0.4.1

- Fixes `pip install`.

8.8 0.4.0 (install broken)

- Adds Travis CI and adds compatibility for:
  - Django 1.6 & 1.7
  - Python 2.6, 2.7, 3.2, 3.3, & 3.4
  - locmem & Redis
  - SQLite, PostgreSQL, MySQL

8.9 0.3.0

- Handles transactions
- Adds lots of tests for complex cases

8.10 0.2.0

- Adds a test suite
- Fixes invalidation for data creation/deletion
- Stops caching on queries defining `select` or `where` arguments with `QuerySet.extra`

8.11 0.1.0

Prototype simply caching all SQL queries reading the database and trying to invalidate them when SQL queries modify the database.

Has issues invalidating deletions and creations. Also caches `QuerySet.extra` queries but can't reliably invalidate them. No transaction support, no test, no multi-database support, etc.