django-cachalot Documentation

Release 1.0.2

Bertrand Bordage

May 25, 2015

Contents

1 Quick start **3**
- 1.1 Requirements . . . 3
- 1.2 Usage . . . 3
- 1.3 Settings . . . 4
- 1.4 Signal . . . 5

2 Limits **7**
- 2.1 Redis . . . 7
- 2.2 Memcached . . . 7
- 2.3 Locmem . . . 7
- 2.4 MySQL . . . 8
- 2.5 Raw SQL queries . . . 8
- 2.6 Multiple Servers . . . 8

3 API **9**

4 Benchmark **11**
- 4.1 Introduction . . . 11
- 4.2 Conditions . . . 11
- 4.3 Database results . . . 12
- 4.4 Cache results . . . 12
- 4.5 Database detailed results . . . 13
- 4.6 Cache detailed results . . . 13

5 What still needs to be done **15**

6 Bug reports, questions, discussion, new features **17**

7 How django-cachalot works **19**
- 7.1 Reverse engineering . . . 19
- 7.2 Monkey patching . . . 19

8 Legacy **21**

9 What's new in django-cachalot? **23**
- 9.1 1.0.2 . . . 23
- 9.2 1.0.1 . . . 23
- 9.3 1.0.0 . . . 23
- 9.4 1.0.0rc . . . 23

9.5 0.9.0 . . . 24
9.6 0.8.1 . . . 24
9.7 0.8.0 . . . 24
9.8 0.7.0 . . . 25
9.9 0.6.0 . . . 25
9.10 0.5.0 . . . 25
9.11 0.4.1 . . . 25
9.12 0.4.0 (**install broken**) . . . 25
9.13 0.3.0 . . . 25
9.14 0.2.0 . . . 26
9.15 0.1.0 . . . 26

Python Module Index **27**

Caches your Django ORM queries and automatically invalidates them.

Since version 1.0.0 it is **safe for 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. If you need to invalidate all django-cachalot cache keys from an external script – typically after restoring a SQL database –, simply run `./manage.py invalidate_cachalot`
6. 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
```

1.4 Signal

`cachalot.signals.post_invalidation` is available if you need to do something just after a cache invalidation (when you modify something in a SQL table). `sender` is the name of the SQL table invalidated, and a keyword argument `db_alias` explains which database is affected by the invalidation. Be careful when you specify `sender`, as it is sensible to string type. To be sure, use `Model._meta.db_table`.

Example:

```
from cachalot.signals import post_invalidation
from django.dispatch import receiver
from django.core.mail import mail_admins
from django.contrib.auth import *

# This prints a message to the console after each table invalidation
def invalidation_debug(sender, **kwargs):
    db_alias = kwargs['db_alias']
    print('%s was invalidated in the DB configured as %s'
          % (sender, db_alias))

post_invalidation.connect(invalidation_debug)

# Using the `receiver` decorator is just a nicer way
# to write the same thing as `signal.connect`.
# Here we specify `sender` so that the function is executed only if
# the table invalidated is the one specified.
# We also connect it several times to be executed for several senders.
@receiver(post_invalidation, sender=User.groups.through._meta.db_table)
@receiver(post_invalidation, sender=User.user_permissions.through._meta.db_table)
@receiver(post_invalidation, sender=Group.permissions.through._meta.db_table)
def warn_admin(sender, **kwargs):
    mail_admins('User permissions changed',
                'Someone probably gained or lost Django permissions.')
```

CHAPTER 2

Limits

2.1 Redis

By default, Redis will not evict persistent cache keys (those with a `None` timeout) when the maximum memory has been reached. The cache keys created by django-cachalot are persistent, so if Redis runs out of memory, django-cachalot and all other `cache.set` will raise `ResponseError: OOM command not allowed when used memory > 'maxmemory'.` because Redis is not allowed to delete persistent keys.

To avoid this, 2 solutions:

- If you only store disposable data in Redis, you can change `maxmemory-policy` to `allkeys-lru` in your Redis configuration. Be aware that this setting is global; all your Redis databases will use it. **If you don't know what you're doing, use the next solution or use another cache backend.**
- Increase `maxmemory` in your Redis configuration. You can start by setting it to a high value (for example half of your RAM) then decrease it by looking at the Redis database maximum size using `redis-cli info memory`.

For more information, read Using Redis as a LRU cache.

2.2 Memcached

By default, memcached is configured for small servers. The maximum amount of memory used by memcached is 64 MB, and the maximum memory per cache key is 1 MB. This latter limit can lead to weird unhandled exceptions such as `Error: error 37 from memcached_set: SUCCESS` if you execute queries returning more than 1 MB of data.

To increase these limits, set the `-I` and `-m` arguments when starting memcached. If you use Ubuntu and installed the package, you can modify */etc/memcached.conf*, add `-I 10` on a newline to set the limit per cache key to 10 MB, and if you want increase the already existing `-m 64` to something like `-m 1000` to set the maximum cache size to 1 GB.

2.3 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.

2.4 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.

2.5 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.

2.6 Multiple Servers

Django-cachalot relies on the computer clock to handle invalidation. If you deploy the same Django project on multiple machines, but with a centralized cache server, all the machines serving Django need to have their clocks as synchronize as possible. Otherwise, invalidations will happen with a latency from one server to another. A difference of even a few seconds can be harmful, so double check this!

To keep your clocks synchronised, use the Network Time Protocol.

CHAPTER 3

API

Use these tools if the automatic behaviour of django-cachalot is not enough. See *Raw SQL queries*.

cachalot.api.**invalidate_tables**(*tables*, *cache_alias=None*, *db_alias=None*)
Clears what was cached by django-cachalot implying one or more SQL tables from `tables`.

If `cache_alias` is specified, it only clears the SQL queries stored on this cache, otherwise queries from all caches are cleared.

If `db_alias` is specified, it only clears the SQL queries executed on this database, otherwise queries from all databases are cleared.

Parameters

- **tables** (*iterable of strings*) – SQL tables names
- **cache_alias** (*string or NoneType*) – Alias from the Django `CACHES` setting
- **db_alias** (*string or NoneType*) – Alias from the Django `DATABASES` setting

Returns Nothing

Return type NoneType

cachalot.api.**invalidate_models**(*models*, *cache_alias=None*, *db_alias=None*)
Shortcut for `invalidate_tables` where you can specify Django models instead of SQL table names.

Parameters

- **models** (iterable of `django.db.models.Model` subclasses) – Django models
- **cache_alias** (*string or NoneType*) – Alias from the Django `CACHES` setting
- **db_alias** (*string or NoneType*) – Alias from the Django `DATABASES` setting

Returns Nothing

Return type NoneType

cachalot.api.**invalidate_all**(*cache_alias=None*, *db_alias=None*)
Clears everything that was cached by django-cachalot.

If `cache_alias` is specified, it only clears the SQL queries stored on this cache, otherwise queries from all caches are cleared.

If `db_alias` is specified, it only clears the SQL queries executed on this database, otherwise queries from all databases are cleared.

Parameters

- **cache_alias** (*string or NoneType*) – Alias from the Django `CACHES` setting

- **db_alias** – Alias from the Django DATABASES setting

Returns Nothing

Return type NoneType

CHAPTER 4

Benchmark

Contents

- *Benchmark*
  - *Introduction*
  - *Conditions*
  - *Database results*
  - *Cache results*
  - *Database detailed results*
    - *MySQL*
    - *PostgreSQL*
    - *SQLite*
  - *Cache detailed results*
    - *File-based*
    - *Locmem*
    - *Memcached (python-memcached)*
    - *Memcached (pylibmc)*
    - *Redis*

4.1 Introduction

This benchmark does not intend to be exhaustive nor fair to SQL. It shows how django-cachalot behaves on an unoptimised application. On an application using perfectly optimised SQL queries only, django-cachalot may not be useful. Unfortunately, most Django apps (including Django itself) use unoptimised queries. Of course, they often lack useful indexes (even though it only requires 20 characters per index…). But what you may not know is that **the ORM currently generates totally unoptimised queries** [1].

4.2 Conditions

In this benchmark, a small database is generated, and each test is executed 20 times under the following conditions:

[1] The ORM fetches way too much data if you don't restrict it using `.only` and `.defer`. You can divide the execution time of most queries by 2-3 by specifying what you want to fetch. But specifying which data we want for each query is very long and unmaintainable. An automation using field usage statistics is possible and would drastically improve performance. Other performance issues occur with slicing. You can often optimise a sliced query using a subquery, like `YourModel.objects.filter(pk__in=YourModel.objects.filter(...)[10000:10050]).select_related(...)` instead of `YourModel.objects.filter(...).select_related(...)[10000:10050]`. I'll maybe work on these issues one day.

CPU	Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz
RAM	12281280 kB
Linux distribution	Ubuntu 14.04 trusty
Python	2.7.6
Django	1.7.8
cachalot	1.0.2
sqlite	3.8.2
PostgreSQL	9.4.2
MySQL	5.5.43
Redis	2.8.4
memcached	1.4.14
psycopg2	2.6
MySQLdb	1.3.6

4.3 Database results

- mysql is 2.1× slower then 0.9× faster
- postgresql is 1.1× slower then 14.0× faster
- sqlite is 1.1× slower then 9.1× faster

4.4 Cache results

- filebased is 1.2× slower then 8.5× faster
- locmem is 1.1× slower then 8.9× faster
- memcached is 1.2× slower then 6.6× faster
- pylibmc is 1.1× slower then 7.3× faster
- redis is 1.1× slower then 7.8× faster

4.5 Database detailed results

4.5.1 MySQL

4.5.2 PostgreSQL

4.5.3 SQLite

4.6 Cache detailed results

4.6.1 File-based

4.6.2 Locmem

4.6.3 Memcached (python-memcached)

4.6.4 Memcached (pylibmc)

4.6.5 Redis

CHAPTER 5

What still needs to be done

- Cache raw queries
- Test multi-location caches if possible
- Allow setting *CACHALOT_CACHE* to *None* in order to disable django-cachalot persistence. SQL queries would only be cached during transactions, so setting *ATOMIC_REQUESTS* to *True* would cache SQL queries only during a request-response cycle. This would be useful for websites with a lot of invalidations (social network for example), but with several times the same SQL queries in a single response-request cycle, as it occurs in Django admin.

CHAPTER 6

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 7

How django-cachalot works

7.1 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.

7.2 Monkey patching

Django-cachalot modifies Django in place during execution to add a caching tool just before SQL queries are executed. When a SQL query reads data, we save the result in cache. If that same query is executed later, we fetch that result from cache. When we detect `INSERT`, `UPDATE` or `DELETE`, we know which tables are modified. All the previous cached queries can therefore be safely invalidated.

CHAPTER 8

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 9

What's new in django-cachalot?

9.1 1.0.2

- Fixes an `AttributeError` occurring when excluding through a many-to-many relation on a child model (using multi-table inheritance)
- Stops caching queries with random subqueries – for example `User.objects.filter(pk__in=User.objects.order_`
- Optimises automatic invalidation
- Adds a note about clock synchronisation

9.2 1.0.1

- Fixes an invalidation issue discovered by Helen Warren that was occurring when updating a `ManyToManyField` after executing using `.exclude` on that relation. For example, `Permission.objects.all().delete()` was not invalidating `User.objects.exclude(user_permissions=None)`
- Fixes a `UnicodeDecodeError` introduced with python-memcached 1.54
- Adds a `post_invalidation` signal

9.3 1.0.0

Fixes a bug occurring when caching a SQL query using a non-ascii table name.

9.4 1.0.0rc

Added:

- Adds an *invalidate_cachalot* command to invalidate django-cachalot from a script without having to clear the whole cache
- Adds the benchmark introduction, conditions & results to the documentation
- Adds a short guide on how to configure Redis as a LRU cache

Fixed:

- Fixes a rare invalidation issue occurring when updating a many-to-many table after executing a queryset generating a `HAVING` SQL statement – for example, `User.objects.first().user_permissions.add(Permission.objects.first())` was not invalidating `User.objects.annotate(n=Count('user_permissions')).filter(n__gte=1)`
- Fixes an even rarer invalidation issue occurring when updating a many-to-many table after executing a queryset filtering nested subqueries by another subquery through that many-to-many table – for example:

```
User.objects.filter(
    pk__in=User.objects.filter(
        pk__in=User.objects.filter(
            user_permissions__in=Permission.objects.all())))
```

- Avoids setting useless cache keys by using table names instead of Django-generated table alias

9.5 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

9.6 0.8.1

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

9.7 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

9.8 0.7.0

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

9.9 0.6.0

- Adds memcached support

9.10 0.5.0

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

9.11 0.4.1

- Fixes `pip install`.

9.12 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

9.13 0.3.0

- Handles transactions
- Adds lots of tests for complex cases

9.14 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`

9.15 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.

C

cachalot.api, 9

C

cachalot.api (module), 9

I

invalidate_all() (in module cachalot.api), 9
invalidate_models() (in module cachalot.api), 9
invalidate_tables() (in module cachalot.api), 9