Django-MySQL Documentation

Release 4.13.0

Adam Johnson

Apr 26, 2024

CONTENTS

Django-MySQL extends Django's built-in MySQL and MariaDB support their specific features not available on other databases.

If you're new, check out the *Exposition* to see all the features in action, or get started with *Installation*. Otherwise, take your pick:

CHAPTER

ONE

EXPOSITION

Every feature in whistle-stop detail.

1.1 Checks

Extra checks added to Django's check framework to ensure your Django and MySQL configurations are optimal.

```
$ ./manage.py check
?: (django_mysql.W001) MySQL strict mode is not set for database connection 'default'
...
```

Read more

1.2 QuerySet Extensions

Django-MySQL comes with a number of extensions to `QuerySet` that can be installed in a number of ways - e.g. adding the `QuerySetMixin` to your existing `QuerySet` subclass.

1.2.1 Approximate Counting

`SELECT COUNT(*) ...` can become a slow query, since it requires a scan of all rows; the `approx_count` functions solves this by returning the estimated count that MySQL keeps in metadata. You can call it directly:

```
Author.objects.approx_count()
```

Or if you have pre-existing code that calls `count()` on a `QuerySet` you pass it, such as the Django Admin, you can set the `QuerySet` to do try `approx_count` first automatically:

```
qs = Author.objects.all().count_tries_approx()
# Now calling qs.count() will try approx_count() first
```

Read more

1.2.2 Query Hints

Use MySQL's query hints to optimize the SQL your `QuerySets` generate:

```
Author.objects.straight_join().filter(book_set__title__startswith="The ")
# Does SELECT STRAIGHT_JOIN ...
```

Read more

1.2.3 'Smart' Iteration

Sometimes you need to modify every single instance of a model in a big table, without creating any long running queries that consume large amounts of resources. The 'smart' iterators traverse the table by slicing it into primary key ranges which span the table, performing each slice separately, and dynamically adjusting the slice size to keep them fast:

```
# Some authors to fix
bad_authors = Author.objects.filter(address="Nowhere")

# Before: bad, we can't fit all these in memory
for author in bad_authors.all():
    pass

# After: good, takes small dynamically adjusted slices, wraps in atomic()
for author in bad_authors.iter_smart():
    author.address = ""
    author.save()
    author.send_apology_email()
```

Read more

1.2.4 Integration with pt-visual-explain

For interactive debugging of queries, this captures the query that the `QuerySet` represents, and passes it through `EXPLAIN` and `pt-visual-explain` to get a visual representation of the query plan:

```
>>> Author.objects.all().pt_visual_explain()
Table scan
rows           1020
+- Table
   table          myapp_author
```

Read more

1.3 Model Fields

Fields that use MariaDB/MySQL-specific features!

1.3.1 Dynamic Columns Field

Use MariaDB's Dynamic Columns for storing arbitrary, nested dictionaries of values:

```
class ShopItem(Model):
    name = models.CharField(max_length=200)
    attrs = DynamicField()
```

```
>>> ShopItem.objects.create(name="Camembert", attrs={"smelliness": 15})
>>> ShopItem.objects.create(name="Brie", attrs={"smelliness": 5, "squishiness": 10})
>>> ShopItem.objects.filter(attrs__smelliness_INTEGER__gte=10)
[<ShopItem: Camembert>]
```

Read more

1.3.2 EnumField

A field class for using MySQL's `ENUM` type, which allows strings that are restricted to a set of choices to be stored in a space efficient manner:

```
class BookCover(Model):
    color = EnumField(choices=["red", "green", "blue"])
```

Read more

1.3.3 FixedCharField

A field class for using MySQL's `CHAR` type, which allows strings to be stored at a fixed width:

```
class Address(Model):
    zip_code = FixedCharField(length=10)
```

Read more

1.3.4 Resizable Text/Binary Fields

Django's `TextField` and `BinaryField` fields are fixed at the MySQL level to use the maximum size class for the `BLOB` and `TEXT` data types - these fields allow you to use the other sizes, and migrate between them:

```
class BookBlurb(Model):
    blurb = SizedTextField(size_class=3)
    # Has a maximum length of 16MiB, compared to plain TextField which has
    # a limit of 4GB (!)
```

Read more

1.3.5 BIT(1) Boolean Fields

Some database systems, such as the Java Hibernate ORM, don't use MySQL's `bool` data type for storing boolean flags and instead use `BIT(1)`. This field class allows you to interact with those fields:

```
class HibernateModel(Model):
    some_bool = Bit1BooleanField()
    some_nullable_bool = NullBit1BooleanField()
```

Read more

1.4 Field Lookups

ORM extensions to built-in fields:

```
>>> Author.objects.filter(name__sounds_like="Robert")
[<Author: Robert>, <Author: Rupert>]
```

Read more

1.5 Aggregates

MySQL's powerful `GROUP_CONCAT` statement is added as an aggregate, allowing you to bring back the concatenation of values from a group in one query:

```
>>> author = Author.objects.annotate(book_ids=GroupConcat("books__id")).get(
...     name="William Shakespeare"
... )
>>> author.book_ids
"1,2,5,17,29"
```

Read more

1.6 Database Functions

MySQL-specific database functions for the ORM:

```
>>> Author.objects.annotate(
...     full_name=ConcatWS("first_name", "last_name", separator=" ")
... ).first().full_name
"Charles Dickens"
```

Read more

1.7 Migration Operations

MySQL-specific operations for django migrations:

```
from django.db import migrations
from django_mysql.operations import InstallPlugin


class Migration(migrations.Migration):
    dependencies = []

    operations = [InstallPlugin("metadata_lock_info", "metadata_lock_info.so")]
```

Read more

1.8 Cache

An efficient backend for Django's cache framework using MySQL features:

```
cache.set("my_key", "my_value")  # Uses only one query
cache.get_many(["key1", "key2"])  # Only one query to do this too!
cache.set("another_key", some_big_value)  # Compressed above 5kb by default
```

Read more

1.9 Locks

Use MySQL as a locking server for arbitrarily named locks:

```
with Lock("ExternalAPI", timeout=10.0):
    do_some_external_api_stuff()
```

Read more

1.10 Status

Easy access to global or session status variables:

```
if global_status.get("Threads_running") > 100:
    raise BorkError("Server too busy right now, come back later")
```

Read more

1.11 Management Commands

`dbparams` helps you include your database parameters from settings in commandline tools with `dbparams`:

```
$ mysqldump $(python manage.py dbparams) > dump.sql
```

Read more

1.12 Test Utilities

Set some MySQL server variables on a test case for every method or just a specific one:

```
class MyTests(TestCase):
    @override_mysql_variables(SQL_MODE="ANSI")
    def test_it_works_in_ansi_mode(self):
        self.run_it()
```

Read more

CHAPTER

TWO

INSTALLATION

2.1 Requirements

Python 3.8 to 3.12 supported.

Django 3.2 to 5.0 supported.

MySQL 8.0 supported.

MariaDB 10.4 to 10.8 supported.

mysqclient 1.3 to 1.4 supported.

2.2 Installation

Install it with **pip**:

```
$ python -m pip install django-mysql
```

Or add it to your project's `requirements.txt`.

Add `'django_mysql'` to your `INSTALLED_APPS` setting:

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

Django-MySQL comes with some extra checks to ensure your database configuration is optimal. It's best to run these now you've installed to see if there is anything to fix:

```
$ python manage.py check --database default
```

(The `--database` argument is new in Django 3.1.)

For help fixing any warnings, see *Checks*.

2.3 Extending your QuerySets

Half the fun features are extensions to `QuerySet`. You can add these to your project in a number of ways, depending on what is easiest for your code - all imported from `django_mysql.models`.

class Model

The simplest way to add the `QuerySet` extensions - this is a subclass of Django's `Model` that sets `objects` to use the Django-MySQL extended `QuerySet` (below) via `QuerySet.as_manager()`. Simply change your model base to get the goodness:

```
# from django.db.models import Model - no more!
from django_mysql.models import Model


class MySuperModel(Model):
    pass  # TODO: come up with startup idea.
```

class QuerySet

The second way to add the extensions - use this to replace your model's default manager:

```
from mythings import MyBaseModel
from django_mysql.models import QuerySet


class MySuperDuperModel(MyBaseModel):
    objects = QuerySet.as_manager()
    # TODO: what fields should this model have??
```

If you are using a custom manager, you can combine this like so:

```
from django.db import models
from django_mysql.models import QuerySet


class MySuperDuperManager(models.Manager):
    pass


class MySuperDuperModel(models.Model):
    objects = MySuperDuperManager.from_queryset(QuerySet)()
    # TODO: fields
```

class QuerySetMixin

The third way to add the extensions, and the container class for the extensions. Add this mixin to your custom `QuerySet` class to add in all the fun:

```
from django.db.models import Model
from django_mysql.models import QuerySetMixin
from stackoverflow import CopyPasteQuerySet


class MySplendidQuerySet(QuerySetMixin, CopyPasteQuerySet):
    pass
```

(continues on next page)

(continued from previous page)

```
class MySplendidModel(Model):
    objects = MySplendidQuerySet.as_manager()
    # TODO: profit
```

add_QuerySetMixin(*queryset*)

A final way to add the extensions, useful when you don't control the model class - for example with built in Django models. This function creates a subclass of a `QuerySet`'s class that has the `QuerySetMixin` added in and applies it to the `QuerySet`:

```
from django.contrib.auth.models import User
from django_mysql.models import add_QuerySetMixin

qs = User.objects.all()
qs = add_QuerySetMixin(qs)
# Now qs has all the extensions!
```

The extensions are described in *QuerySet Extensions*.

CHAPTER

THREE

CHECKS

Django-MySQL adds some extra checks to Django's system check framework to advise on your database configuration. If triggered, the checks give a brief message, and a link here for documentation on how to fix it.

Warning: From Django 3.1 onwards, database checks are not automatically run in most situations. You should use the `--database` argument to `manage.py check` to run the checks. For example, with just one database connection you can run `manage.py check --database default`.

Note: A reminder: as per the Django docs, you can silence individual checks in your settings. For example, if you determine `django_mysql.W002` doesn't require your attention, add the following to `settings.py`:

```
SILENCED_SYSTEM_CHECKS = [
    "django_mysql.W002",
]
```

3.1 django_mysql.W001: Strict Mode

This check has been removed since Django itself includes such a check, `mysql.W002`, since version 1.10. See its documentation.

3.2 django_mysql.W002: InnoDB Strict Mode

InnoDB Strict Mode is similar to the general Strict Mode, but for InnoDB. It escalates several warnings around InnoDB-specific statements into errors. Normally this just affects per-table settings for compression. It's recommended you activate this, but it's not very likely to affect you if you don't.

Docs: MySQL / MariaDB.

As above, the easiest way to set this is to add `SET` to `init_command` in your `DATABASES` setting:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "my_database",
        "OPTIONS": {
```

(continues on next page)

(continued from previous page)

```
            "init_command": "SET innodb_strict_mode=1",
        },
    }
}
```

Note: If you use this along with the `init_command` for W001, combine them as `SET sql_mode='STRICT_TRANS_TABLES', innodb_strict_mode=1`.

Also, as above for `django_mysql.W001`, it's better that you set it permanently for the server with `SET GLOBAL` and a configuration file change.

3.3 django_mysql.W003: utf8mb4

MySQL's `utf8` character set does not include support for the largest, 4 byte characters in UTF-8; this basically means it cannot support emoji and custom Unicode characters. The `utf8mb4` character set was added to support all these characters, and there's really little point in not using it. Django currently suggests using the `utf8` character set for backwards compatibility, but it's likely to move in time.

It's strongly recommended you change to the `utf8mb4` character set and convert your existing `utf8` data as well, unless you're absolutely sure you'll never see any of these 'supplementary' Unicode characters (note: it's very easy for users to type emoji on phone keyboards these days!).

Docs: MySQL / MariaDB.

Also see this classic blogpost: How to support full Unicode in MySQL databases.

The easiest way to set this up is to make a couple of changes to your `DATABASES` settings. First, add `OPTIONS` with `charset` to your MySQL connection, so `MySQLdb` connects using the `utf8mb4` character set. Second, add `TEST` with `COLLATION` and `CHARSET` as below, so Django creates the test database, and thus all tables, with the right character set:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "my_database",
        "OPTIONS": {
            # Tell MySQLdb to connect with 'utf8mb4' character set
            "charset": "utf8mb4",
        },
        # Tell Django to build the test database with the 'utf8mb4' character set
        "TEST": {
            "CHARSET": "utf8mb4",
            "COLLATION": "utf8mb4_unicode_ci",
        },
    }
}
```

Note this does not transform the database, tables, and columns that already exist. Follow the examples in the 'How to' blog post link above to fix your database, tables, and character set. It's planned to add a command to Django-MySQL to help you do this, see Issue 216.

CHAPTER

FOUR

QUERYSET EXTENSIONS

MySQL-specific Model and QuerySet extensions. To add these to your `Model`/`Manager`/`QuerySet` trifecta, see *Installation*. Methods below are all `QuerySet` methods; where standalone forms are referred to, they can be imported from `django_mysql.models`.

4.1 Approximate Counting

`django_mysql.models.`**`approx_count`**(*fall_back=True*, *return_approx_int=True*, *min_size=1000*)

By default a QuerySet's *count()* method runs *SELECT COUNT(*)* on a table. Whilst this is fast for `MyISAM` tables, for `InnoDB` it involves a full table scan to produce a consistent number, due to MVCC keeping several copies of rows when under transaction. If you have lots of rows, you will notice this as a slow query - Percona have some more details.

This method returns the approximate count found by running `EXPLAIN SELECT COUNT(*) ...`. It can be out by 30-50% in the worst case, but in many applications it is closer, and is good enough, such as when presenting many pages of results but users will only practically scroll through the first few. For example:

```
>>> Author.objects.count()  # slow
509741
>>> Author.objects.approx_count()  # fast, with some error
531140
```

Three arguments are accepted:

`fall_back=True`

If `True` and the approximate count cannot be calculated, `count()` will be called and returned instead, otherwise `ValueError` will be raised.

The approximation can only be found for `objects.all()`, with no filters, `distinct()` calls, etc., so it's reasonable to fall back.

`return_approx_int=True`

When `True`, an `int` is not returned (excpet when falling back), but instead a subclass called `ApproximateInt`. This is for all intents and purposes an `int`, apart from when cast to `str`, it renders as e.g. **'Approximately 12345'** (internationalization ready). Useful for templates you can't edit (e.g. the admin) and you want to communicate that the number is not 100% accurate. For example:

```
>>> print(Author.objects.approx_count())  # ApproximateInt
Approximately 531140
>>> print(Author.objects.approx_count() + 0)  # plain int
531140
```

(continues on next page)

(continued from previous page)

```
>>> print(Author.objects.approx_count(return_approx_int=False))  # plain int
531140
```

min_size=1000

The threshold at which to use the approximate algorithm; if the approximate count comes back as less that this number, `count()` will be called and returned instead, since it should be so small as to not bother your database. Set to `0` to disable this behaviour and always return the approximation.

The default of `1000` is a bit pessimistic - most tables won't take long when calling `COUNT(*)` on tens of thousands of rows, but it *could* be slow for very wide tables.

`django_mysql.models.`**`count_tries_approx`**(*activate=True*, *fall_back=True*, *return_approx_int=True*, *min_size=1000*)

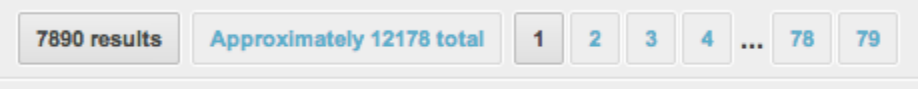
This is the 'magic' method to make pre-existing code, such as Django's admin, work with `approx_count`. Calling `count_tries_approx` sets the QuerySet up such that then calling `count` will call `approx_count` instead, with the given arguments.

To unset this, call `count_tries_approx` with `activate=False`.

To 'fix' an Admin class with this, simply do the following (assuming `Author` inherits from `django_mysql`'s `Model`):

```
class AuthorAdmin(ModelAdmin):
    def get_queryset(self, request):
        qs = super(AuthorAdmin, self).get_queryset(request)
        return qs.count_tries_approx()
```

You'll be able to see this is working on the pagination due to the word **'Approximately'** appearing:



You can do this at a base class for all your `ModelAdmin` subclasses to apply the magical speed increase across your admin interface.

4.2 Query Hints

The following methods add extra features to the ORM which allow you to access some MySQL-specific syntax. They do this by inserting special comments which pass through Django's ORM layer and get re-written by a function that wraps the lower-level `cursor.execute()`.

Because not every user wants these features and there is a (small) overhead to every query, you must activate this feature by adding to your settings:

```
DJANGO_MYSQL_REWRITE_QUERIES = True
```

Once you've done this, the following methods will work.

`django_mysql.models.`**`label`**(*comment*)

Allows you to add an arbitrary comment to the start of the query, as the second thing after the keyword. This can be used to 'tag' queries so that when they show in the *slow_log* or another monitoring tool, you can easily back track to the python code generating the query. For example, imagine constructing a QuerySet like this:

```
qs = Author.objects.label("AuthorListView").all()
```

When executed, this will have SQL starting:

```
SELECT /*AuthorListView*/ ...
```

You can add arbitrary labels, and as many of them as you wish - they will appear in the order added. They will work in `SELECT` and `UPDATE` statements, but not in `DELETE` statements due to limitations in the way Django performs deletes.

You should not pass user-supplied data in for the comment. As a basic protection against accidental SQL injection, passing a comment featuring `*/` will raise a `ValueError`, since that would prematurely end the comment. However due to executable comments, the comment is still prone to some forms of injection.

However this is a feature - by not including spaces around your string, you may use this injection to use executable comments to add hints that are otherwise not supported, or to use MySQL 5.7+ optimizer hints.

`django_mysql.models.straight_join()`

Adds the `STRAIGHT_JOIN` hint, which forces the join order during a `SELECT`. Note that you can't force Django's join order, but it tends to be in the order that the tables get mentioned in the query.

Example usage:

```
# Note from Adam: sometimes the optimizer joined books -> author, which
# is slow. Force it to do author -> books.
Author.objects.distinct().straight_join().filter(books__age=12)[:10]
```

Docs: MySQL / MariaDB.

The MariaDB docs also have a good page Index Hints: How to Force Query Plans" which covers some cases when you might want to use `STRAIGHT_JOIN`.

`django_mysql.models.sql_small_result()`

Adds the `SQL_SMALL_RESULT` hint, which avoids using a temporary table in the case of a `GROUP BY` or `DISTINCT`.

Example usage:

```
# Note from Adam: we have very few distinct birthdays, so using a
# temporary table is slower
Author.objects.values("birthday").distinct().sql_small_result()
```

Docs: MySQL / MariaDB.

`django_mysql.models.sql_big_result()`

Adds the `SQL_BIG_RESULT` hint, which forces using a temporary table in the case of a `GROUP BY` or `DISTINCT`.

Example usage:

```
# Note from Adam: for some reason the optimizer didn't use a temporary
# table for this, so we force it
Author.objects.distinct().sql_big_result()
```

Docs: MySQL / MariaDB.

`django_mysql.models.sql_buffer_result()`

Adds the `SQL_BUFFER_RESULT` hint, which forces the optimizer to use a temporary table to process the result. This is useful to free locks as soon as possible.

Example usage:

```
# Note from Adam: seeing a lot of throughput on this table. Buffering
# the results makes the queries less contentious.
HighThroughputModel.objects.filter(x=y).sql_buffer_result()
```

Docs: MySQL / MariaDB.

`django_mysql.models.sql_cache()`

Adds the `SQL_CACHE` hint, which means the result set will be stored in the Query Cache. This only has an effect when the MySQL system variable `query_cache_type` is set to `2` or `DEMAND`.

> **Warning:** The query cache was removed in MySQL 8.0, and is disabled by default from MariaDB 10.1.7.

Example usage:

```
# Fetch recent posts, cached in MySQL for speed
recent_posts = BlogPost.objects.sql_cache().order_by("-created")[:5]
```

Docs: MariaDB.

`django_mysql.models.sql_no_cache()`

Adds the `SQL_NO_CACHE` hint, which means the result set will not be fetched from or stored in the Query Cache. This only has an effect when the MySQL system variable `query_cache_type` is set to `1` or `ON`.

> **Warning:** The query cache was removed in MySQL 8.0, and is disabled by default from MariaDB 10.1.7.

Example usage:

```
# Avoid caching all the expired sessions, since we're about to delete
# them
deletable_session_ids = (
    Session.objects.sql_no_cache().filter(expiry__lt=now()).values_list("id",␣
↪flat=True)
)
```

Docs: MariaDB.

`django_mysql.models.sql_calc_found_rows()`

Adds the `SQL_CALC_FOUND_ROWS` hint, which means the total count of matching rows will be calculated when you only take a slice. You can access this count with the `found_rows` attribute of the `QuerySet` after filling its result cache, by e.g. iterating it.

This can be faster than taking the slice and then again calling `.count()` to get the total count.

> **Warning:** This is deprecated in MySQL 8.0.17+.

Example usage:

```
>>> can_drive = Customer.objects.filter(age=21).sql_calc_found_rows()[:10]
>>> len(can_drive)  # Fetches the first 10 from the database
```

(continues on next page)

(continued from previous page)

```
10
>>> can_drive.found_rows  # The total number of 21 year old customers
1942
```

Docs: MySQL / MariaDB.

`django_mysql.models.`**`use_index`**(**index_names*, *for_=None*, *table_name=None*)

Adds a `USE INDEX` hint, which affects the index choice made by MySQL's query optimizer for resolving the query.

Note that index names on your tables will normally have been generated by Django and contain a hash fragment. You will have to check your database schema to determine the index name.

If you pass any non-existent index names, MySQL will raise an error. This means index hints are especially important to test in the face of future schema changes.

`for_` restricts the scope that the index hint applies to. By default it applies to all potential index uses during the query; you may supply one of `'JOIN'`, `'ORDER BY'`, or `'GROUP BY'` to restrict the index hint to only be used by MySQL for index selection in their respective stages of query execution. For more information see the MySQL/MariaDB docs (link below).

`table_name` is the name of the table that the hints are for. By default, this will be the name of the table of the model that the `QuerySet` is for, however you can supply any other table that may be joined into the query (from e.g. `select_related()`). Be careful - there is no validation on the table name, and if it does not exist in the final query it will be ignored. Also it is injected raw into the resultant SQL, so you should not use user data otherwise it may open the potential for SQL injection.

Note that `USE INDEX` accepts no index names to mean 'use no indexes', i.e. table scans only.

Example usage:

```
# SELECT ... FROM `author` USE INDEX (`name_12345`) WHERE ...
>>> Author.objects.use_index("name_12345").filter(name="John")
# SELECT ... FROM `author` USE INDEX (`name_12345`, `name_age_678`) WHERE ...
>>> Author.objects.use_index("name_12345", "name_age_678").filter(name="John")
# SELECT ... FROM `author` USE INDEX FOR ORDER BY (`name_12345`) ... ORDER BY `name`
>>> Author.objects.use_index("name_12345", for_="ORDER BY").order_by("name")
# SELECT ... FROM `book` INNER JOIN `author` USE INDEX (`authbook`) ...
>>> Book.objects.select_related("author").use_index("authbook", table_name="author")
```

Docs: MySQL / MariaDB.

`django_mysql.models.`**`force_index`**(**index_names*, *for_=None*)

Similar to the above `use_index()`, but adds a `FORCE INDEX` hint. Note that unlike `use_index()` you must supply at least one index name. For more information, see the MySQL/MariaDB docs.

`django_mysql.models.`**`ignore_index`**(**index_names*, *for_=None*)

Similar to the above `use_index()`, but adds an `IGNORE INDEX` hint. Note that unlike `use_index()` you must supply at least one index name. For more information, see the MySQL/MariaDB docs.

4.3 'Smart' Iteration

Here's a situation we've all been in - we screwed up, and now we need to fix the data. Let's say we accidentally set the address of all authors without an address to "Nowhere", rather than the blank string. How can we fix them??

The simplest way would be to run the following:

```
Author.objects.filter(address="Nowhere").update(address="")
```

Unfortunately with a lot of rows ('a lot' being dependent on your database server and level of traffic) this will stall other access to the table, since it will require MySQL to read all the rows and to hold write locks on them in a single query.

To solve this, we could try updating a chunk of authors at a time; such code tends to get ugly/complicated pretty quickly:

```
min_id = 0
max_id = 1000
biggest_author_id = Author.objects.order_by("-id")[0].id
while True:
    Author.objects.filter(id__gte=min_id, id__lte=...)
    # I'm not even going to type this all out, it's so much code
```

Here's the solution to this boilerplate with added safety features - 'smart' iteration! There are two classes; one yields chunks of the given `QuerySet`, and the other yields the objects inside those chunks. Nearly every data update can be thought of in one of these two methods.

class django_mysql.models.**SmartChunkedIterator**(*queryset*, *atomically=True*, *status_thresholds=None*, *pk_range=None*, *chunk_time=0.5*, *chunk_size=2*, *chunk_min=1*, *chunk_max=10000*, *report_progress=False*, *total=None*)

Implements a smart iteration strategy over the given `queryset`. There is a method `iter_smart_chunks` that takes the same arguments on the `QuerySetMixin` so you can just:

```
bad_authors = Author.objects.filter(address="Nowhere")
for author_chunk in bad_authors.iter_smart_chunks():
    author_chunk.update(address="")
```

Iteration proceeds by yielding primary-key based slices of the queryset, and dynamically adjusting the size of the chunk to try and take `chunk_time` seconds. In between chunks, the *wait_until_load_low()* method of *GlobalStatus* is called to ensure the database is not under high load.

> **Warning:** Because of the slicing by primary key, there are restrictions on what `QuerySet`s you can use, and a `ValueError` will be raised if the queryset doesn't meet that. Specifically, only `QuerySet`s on models with integer-based primary keys, which are unsliced, and have no `order_by` will work.

There are a lot of arguments and the defaults have been picked hopefully sensibly, but please check for your case though!

queryset

The queryset to iterate over; if you're calling via `.iter_smart_chunks` then you don't need to set this since it's the queryset you called it on.

atomically=True

If true, wraps each chunk in a transaction via django's `transaction.atomic()`. Recommended for any write processing.

status_thresholds=None

A dict of status variables and their maximum tolerated values to be checked against after each chunk with *wait_until_load_low()*.

When set to `None`, the default, *GlobalStatus* will use its default of `{"Threads_running": 10}`. Set to an empty dict to disable status checking - but this is not really recommended, as it can save you from locking up your site with an overly aggressive migration.

Using `Threads_running` is the most recommended variable to check against, and is copeid from the default behaviour of `pt-online-schema-change`. The default value of 10 threads is deliberately conservative to avoid locking small database servers. You should tweak it up based upon the live activity of your server - check the running thread count during normal traffic and add some overhead.

pk_range=None

Controls the primary key range to iterate over with slices. By default, with `pk_range=None`, the QuerySet will be searched for its minimum and maximum `pk` values before starting. On QuerySets that match few rows, or whose rows aren't evenly distributed, this can still execute a long blocking table scan to find these two rows. You can remedy this by giving a value for `pk_range`:

- If set to `'all'`, the range will be the minimum and maximum PK values of the entire table, excluding any filters you have set up - that is, for `Model.objects.all()` for the given `QuerySet`'s model.
- If set to a 2-tuple, it will be unpacked and used as the minimum and maximum values respectively.

Note: The iterator determines the minimum and maximum at the start of iteration and does not update them whilst iterating, which is normally a safe assumption, since if you're "fixing things" you probably aren't creating any more bad data. If you do need to process *every* row then set `pk_range` to have a maximum far greater than what you expect would be reached by inserts that occur during iteration.

---

chunk_time=0.5

The time in seconds to aim for each chunk to take. The chunk size is dynamically adjusted to try and match this time, via a weighted average of the past and current speed of processing. The default and algorithm is taken from the analogous `pt-online-schema-change` flag –chunk-time.

chunk_size=2

The initial size of the chunk that will be used. As this will be dynamically scaled and can grow fairly quickly, the initial size of 2 should be appropriate for most use cases.

chunk_min=1

The minimum number of objects in a chunk. You do not normally need to tweak this since the dynamic scaling works very well, however it might be useful if your data has a lot of "holes" or if there are other constraints on your application.

chunk_max=10000

The maximum number of objects in a chunk, a kind of sanity bound. Acts to prevent harm in the case of iterating over a model with a large 'hole' in its primary key values, e.g. if only ids 1-10k and 100k-110k exist, then the chunk 'slices' could grow very large in between 10k and 100k since you'd be "processing" the non-existent objects 10k-100k very quickly.

report_progress=False

If set to true, display out a running counter and summary on `sys.stdout`. Useful for interactive use. The message looks like this:

```
AuthorSmartChunkedIterator processed 0/100000 objects (0.00%) in 0 chunks
```

And uses `\r` to erase itself when re-printing to avoid spamming your screen. At the end `Finished!` is printed on a new line.

total=None

By default the total number of objects to process will be calculated with `approx_count()`, with `fall_back` set to `True`. This `count()` query could potentially be big and slow.

`total` allows you to pass in the total number of objects for processing, if you can calculate in a cheaper way, for example if you have a read-replica to use.

class `django_mysql.models.`**SmartIterator**

A convenience subclass of `SmartChunkedIterator` that simply unpacks the chunks for you. Can be accessed via the `iter_smart` method of `QuerySetMixin`.

For example, rather than doing this:

```
bad_authors = Author.objects.filter(address="Nowhere")
for authors_chunk in bad_authors.iter_smart_chunks():
    for author in authors_chunk:
        author.send_apology_email()
```

You can do this:

```
bad_authors = Author.objects.filter(address="Nowhere")
for author in bad_authors.iter_smart():
    author.send_apology_email()
```

All the same arguments as `SmartChunkedIterator` are accepted.

class `django_mysql.models.`**SmartPKRangeIterator**

A subclass of `SmartChunkedIterator` that doesn't return the chunk's `QuerySet` but instead returns the start and end primary keys for the chunk. This may be useful when you want to iterate but the slices need to be used in a raw SQL query. Can be accessed via the `iter_smart_pk_ranges` method of `QuerySetMixin`.

For example, rather than doing this:

```
for authors_chunk in Author.objects.iter_smart_chunks():
    limits = author_chunk.aggregate(min_pk=Min("pk"), max_pk=Max("pk"))
    authors = Author.objects.raw(
        """
        SELECT name from app_author
        WHERE id >= %s AND id <= %s
    """,
        (limits["min_pk"], limits["max_pk"]),
    )
    # etc...
```

...you can do this:

```
for start_pk, end_pk in Author.objects.iter_smart_pk_ranges():
    authors = Author.objects.raw(
        """
        SELECT name from app_author
        WHERE id >= %s AND id < %s
    """,
        (start_pk, end_pk),
```

(continues on next page)

(continued from previous page)

```
    )
    # etc...
```

In the first format we were forced to perform a dumb query to determine the primary key limits set by `SmartChunkedIterator`, due to the `QuerySet` not otherwise exposing this information.

Note: There is a **subtle** difference between the two versions. In the first the end boundary, `max_pk`, is a closed bound, whereas in the second, the `end_pk` from `iter_smart_pk_ranges` is an open bound. Thus the <= changes to a <.

All the same arguments as `SmartChunkedIterator` are accepted.

4.4 Integration with pt-visual-explain

How does MySQL *really* execute a query? The `EXPLAIN` statement (docs: MySQL / MariaDB), gives a description of the execution plan, and the `pt-visual-explain` tool can format this in an understandable tree.

This function is a shortcut to turn a `QuerySet` into its visual explanation, making it easy to gain a better understanding of what your queries really end up doing.

`django_mysql.models.`**`pt_visual_explain`**(*display=True*)

Call on a `QuerySet` to print its visual explanation, or with `display=False` to return it as a string. It prepends the SQL of the query with 'EXPLAIN' and passes it through the `mysql` and `pt-visual-explain` commands to get the output. You therefore need the MySQL client and Percona Toolkit installed where you run this.

Example:

```
>>> Author.objects.all().pt_visual_explain()
Table scan
rows           1020
+- Table
   table          myapp_author
```

Can also be imported as a standalone function if you want to use it on a `QuerySet` that does not have the `QuerySetMixin` added, e.g. for built-in Django models:

```
>>> from django_mysql.models import pt_visual_explain
>>> pt_visual_explain(User.objects.all())
Table scan
rows           1
+- Table
   table          auth_user
```

CHAPTER

FIVE

MODEL FIELDS

More MySQL and MariaDB specific ways to store data!

Field classes should always be imported from `django_mysql.models`, similar to the home of Django's native fields in `django.db.models`.

5.1 DynamicField

MariaDB has a feature called **Dynamic Columns** that allows you to store different sets of columns for each row in a table. It works by storing the data in a blob and having a small set of functions to manipulate this blob. (Docs).

Django-MySQL supports the *named* Dynamic Columns of MariaDB 10.0+, as opposed to the *numbered* format of 5.5+. It uses the mariadb-dyncol python package to pack and unpack Dynamic Columns blobs in Python rather than in MariaDB (mostly due to limitations in the Django ORM).

class django_mysql.models.**DynamicField**(*spec=None*, ***kwargs*)

A field for storing Dynamic Columns. The Python data type is `dict`. Keys must be `str`s and values must be one of the supported value types in `mariadb-dyncol`:

- `str`
- `int`
- `float`
- `datetime.date`
- `datetime.datetime`
- `datetime.datetime`
- A nested dict conforming to this spec too

Note that there are restrictions on the range of values supported for some of these types, and that `decimal.Decimal` objects are not yet supported though they are valid in MariaDB. For more information consult the `mariadb-dyncol` documentation.

Values may also be `None`, though they will then not be stored, since dynamic columns do not store `NULL`, so you should use `.get()` to retrieve values that may be `None`.

To use this field, you'll need to:

1. Use MariaDB 10.0.2+
2. Install `mariadb-dyncol` (`python -m pip install mariadb-dyncol`)
3. Use either the `utf8mb4` or `utf8` character set for your database connection.

These are all checked by the field and you will see sensible errors for them when Django's checks run if you have a `DynamicField` on a model.

spec

This is an optional type specification that checks that the named columns, if present, have the given types. It is validated against on `save()` to ensure type safety (unlike normal Django validation which is only used in forms). It is also used for type information for lookups (below).

`spec` should be a `dict` with string keys and values that are the type classes you expect. You can also nest another such dictionary as a value for validating nested dynamic columns.

For example:

```
import datetime


class SpecModel(Model):
    attrs = DynamicField(
        spec={
            "an_integer_key": int,
            "created_at": datetime.datetime,
            "nested_columns": {
                "lat": int,
                "lon": int,
            },
        }
    )
```

This will enforce the following rules:

- `instance.attrs['an_integer_key']`, if present, is an `int`
- `instance.attrs['created_at']`, if present, is an `datetime.datetime`
- `instance.attrs['nested_columns']`, if present, is a `dict`
- `instance.attrs['nested_columns']['lat']`, if present, is an `int`
- `instance.attrs['nested_columns']['lon']`, if present, is an `int`

Trying to save a `DynamicField` with data that does not match the rules of its `spec` will raise `TypeError`. There is no automatic casting, e.g. between `int` and `float`. Note that columns not in `spec` will still be allowed and have no type enforced.

For example:

```
>>> SpecModel.objects.create(attrs={"an_integer_key": 1})  # Fine
>>> SpecModel.objects.create(attrs={"an_integer_key": 2.0})
Traceback (most recent call last):
...
TypeError: Key 'an_integer_key' should be of type 'int'
>>> SpecModel.objects.create(attrs={"non_spec_key": "anytype"})  # Fine
```

5.1.1 DynamicFields in Forms

By default a `DynamicField` has no form field, because there isn't really a practical way to edit its contents. If required, is possible to add extra form fields to a `ModelForm` that then update specific dynamic column names on the instance in the form's `save()`.

5.1.2 Querying DynamicField

You can query by names, including nested names. In cases where names collide with existing lookups (e.g. you have a column named `'exact'`), you might want to use the *`ColumnGet`* database function. You can also use the *`ColumnAdd`* and *`ColumnDelete`* functions for atomically modifying the contents of dynamic columns at the database layer.

We'll use the following example model:

```
from django_mysql.models import DynamicField, Model


class ShopItem(Model):
    name = models.CharField(max_length=200)
    attrs = DynamicField(
        spec={
            "size": str,
        }
    )

    def __str__(self):
        return self.name
```

Exact Lookups

To query based on an exact match, just use a dictionary.

For example:

```
>>> ShopItem.objects.create(name="Camembert", attrs={"smelliness": 15})
>>> ShopItem.objects.create(name="Cheddar", attrs={"smelliness": 15, "hardness": 5})

>>> ShopItem.objects.filter(attrs={"smelliness": 15})
[<ShopItem: Camembert>]
>>> ShopItem.objects.filter(attrs={"smelliness": 15, "hardness": 5})
[<ShopItem: Cheddar>]
```

Name Lookups

To query based on a column name, use that name as a lookup with one of the below SQL types added after an underscore. If the column name is in your field's `spec`, you can omit the SQL type and it will be extracted automatically - this includes keys in nested `dicts`.

The list of SQL types is:

- `BINARY` - `dict` (a nested `DynamicField`)
- `CHAR` - `str`

- `DATE` - `datetime.date`
- `DATETIME` - `datetime.datetime`
- `DOUBLE` - `float`
- `INTEGER` - `int`
- `TIME` - `datetime.time`

These will also use the correct Django ORM field so chained lookups based on that type are possible, e.g. `dynamicfield__age_INTEGER__gte=20`.

Beware that getting a named column can always return `NULL` if the column is not defined for a row.

For example:

```
>>> ShopItem.objects.create(name="T-Shirt", attrs={"size": "Large"})
>>> ShopItem.objects.create(
...     name="Rocketship",
...     attrs={"speed_mph": 300, "dimensions": {"width_m": 10, "height_m": 50}},
... )

# Basic template: DynamicField + '__' + column name + '_' + SQL type
>>> ShopItem.objects.filter(attrs__size_CHAR="Large")
[<ShopItem: T-Shirt>]

# As 'size' is in the field's spec, there is no need to give the SQL type
>>> ShopItem.objects.filter(attrs__size="Large")
[<ShopItem: T-Shirt>]

# Chained lookups are possible based on the data type
>>> ShopItem.objects.filter(attrs__speed_mph_INTEGER__gte=100)
[<ShopItem: Rocketship>]

# Nested keys can be looked up
>>> ShopItem.objects.filter(attrs__dimensions_BINARY__width_m_INTEGER=10)
[<ShopItem: Rocketship>]

# Nested DynamicFields can be queried as ``dict``s, as per the ``exact`` lookup
>>> ShopItem.objects.filter(attrs__dimensions_BINARY={"width_m": 10, "height_m": 50})
[<ShopItem: Rocketship>]

# Missing keys are always NULL
>>> ShopItem.objects.filter(attrs__blablabla_INTEGER__isnull=True)
[<ShopItem: T-Shirt>, <ShopItem: Rocketship>]
```

5.2 List Fields

Legacy

These field classes are only maintained for legacy purposes. They aren't recommended as comma separation is a fragile serialization format.

For new uses, you're better off using Django 3.1's `JSONField` that works with all database backends. On earlier versions of Django, you can use django-jsonfield-backport.

Two fields that store lists of data, grown-up versions of Django's `CommaSeparatedIntegerField`, cousins of `django.contrib.postgres`'s `ArrayField`. There are two versions: `ListCharField`, which is based on `CharField` and appropriate for storing lists with a small maximum size, and `ListTextField`, which is based on `TextField` and therefore suitable for lists of (near) unbounded size (the underlying `LONGTEXT` MySQL datatype has a maximum length of 2^{32} - 1 bytes).

class `django_mysql.models.`**ListCharField**(*base_field*, *size=None*, ***kwargs*)

A field for storing lists of data, all of which conform to the `base_field`.

base_field

The base type of the data that is stored in the list. Currently, must be `IntegerField`, `CharField`, or any subclass thereof - except from `ListCharField` itself.

size

Optionally set the maximum numbers of items in the list. This is only checked on form validation, not on model save!

As `ListCharField` is a subclass of `CharField`, any `CharField` options can be set too. Most importantly you'll need to set `max_length` to determine how many characters to reserve in the database.

Example instantiation:

```
from django.db.models import CharField, Model
from django_mysql.models import ListCharField


class Person(Model):
    name = CharField()
    post_nominals = ListCharField(
        base_field=CharField(max_length=10),
        size=6,
        max_length=(6 * 11),  # 6 * 10 character nominals, plus commas
    )
```

In Python simply set the field's value as a list:

```
>>> p = Person.objects.create(name="Horatio", post_nominals=["PhD", "Esq."])
>>> p.post_nominals
['PhD', 'Esq.']
>>> p.post_nominals.append("III")
>>> p.post_nominals
['PhD', 'Esq.', 'III']
>>> p.save()
```

Validation on save()

When performing the list-to-string conversion for the database, `ListCharField` performs some validation, and will raise `ValueError` if there is a problem, to avoid saving bad data. The following are invalid:

- Any member containing a comma in its string representation
- Any member whose string representation is the empty string

The default form field is *SimpleListField*.

class django_mysql.models.**ListTextField**(*base_field*, *size=None*, ***kwargs*)

The same as `ListCharField`, but backed by a `TextField` and therefore much less restricted in length. There is no `max_length` argument.

Example instantiation:

```
from django.db.models import IntegerField, Model
from django_mysql.models import ListTextField


class Widget(Model):
    widget_group_ids = ListTextField(
        base_field=IntegerField(),
        size=100,  # Maximum of 100 ids in list
    )
```

5.2.1 Querying List Fields

Warning: These fields are not built-in datatypes, and the filters use one or more SQL functions to parse the underlying string representation. They may slow down on large tables if your queries are not selective on other columns.

contains

The `contains` lookup is overridden on `ListCharField` and `ListTextField` to match where the set field contains the given element, using MySQL's `FIND_IN_SET` function (docs: MariaDB / MySQL docs).

For example:

```
>>> Person.objects.create(name="Horatio", post_nominals=["PhD", "Esq.", "III"])
>>> Person.objects.create(name="Severus", post_nominals=["PhD", "DPhil"])
>>> Person.objects.create(name="Paulus", post_nominals=[])

>>> Person.objects.filter(post_nominals__contains="PhD")
[<Person: Horatio>, <Person: Severus>]

>>> Person.objects.filter(post_nominals__contains="Esq.")
[<Person: Horatio>]

>>> Person.objects.filter(post_nominals__contains="DPhil")
[<Person: Severus>]
```

(continues on next page)

(continued from previous page)

```
>>> Person.objects.filter(
...     Q(post_nominals__contains="PhD") & Q(post_nominals__contains="III")
... )
[<Person: Horatio>]
```

Note: `ValueError` will be raised if you try `contains` with a list. It's not possible without using `AND` in the query, so you should add the filters for each item individually, as per the last example.

len

A transform that converts to the number of items in the list. For example:

```
>>> Person.objects.filter(post_nominals__len=0)
[<Person: Paulus>]

>>> Person.objects.filter(post_nominals__len=2)
[<Person: Severus>]

>>> Person.objects.filter(post_nominals__len__gt=2)
[<Person: Horatio>]
```

Index lookups

This class of lookups allows you to index into the list to check if the first occurrence of a given element is at a given position. There are no errors if it exceeds the `size` of the list. For example:

```
>>> Person.objects.filter(post_nominals__0="PhD")
[<Person: Horatio>, <Person: Severus>]

>>> Person.objects.filter(post_nominals__1="DPhil")
[<Person: Severus>]

>>> Person.objects.filter(post_nominals__100="VC")
[]
```

Warning: The underlying function, `FIND_IN_SET`, is designed for *sets*, i.e. comma-separated lists of unique elements. It therefore only allows you to query about the *first* occurrence of the given item. For example, this is a non-match:

```
>>> Person.objects.create(name="Cacistus", post_nominals=["MSc", "MSc"])
>>> Person.objects.filter(post_nominals__1="MSc")
[]  # Cacistus does not appear because his first MSc is at position 0
```

This may be fine for your application, but be careful!

Note: `FIND_IN_SET` uses 1-based indexing for searches on comma-based strings when writing raw SQL. However

these indexes use 0-based indexing to be consistent with Python.

Note: Unlike the similar feature on `django.contrib.postgres`'s `ArrayField`, 'Index transforms', these are lookups, and only allow direct value comparison rather than continued chaining with the base-field lookups. This is because the field is not a native list type in MySQL.

5.2.2 `ListF()` expressions

Similar to Django's `F` expression, this allows you to perform an atomic add and remove operations on list fields at the database level:

```
>>> from django_mysql.models import ListF
>>> Person.objects.filter(post_nominals__contains="PhD").update(
...     post_nominals=ListF("post_nominals").append("Sr.")
... )
2
>>> Person.objects.update(post_nominals=ListF("post_nominals").pop())
3
```

Or with attribute assignment to a model:

```
>>> horatio = Person.objects.get(name="Horatio")
>>> horatio.post_nominals = ListF("post_nominals").append("DSocSci")
>>> horatio.save()
```

class django_mysql.models.**ListF**(*field_name*)

You should instantiate this class with the name of the field to use, and then call one of its methods.

Note that unlike `F`, you cannot chain the methods - the SQL involved is a bit too complicated, and thus only single operations are supported.

append(*value*)

Adds the value of the given expression to the (right hand) end of the list, like `list.append`:

```
>>> Person.objects.create(name="Horatio", post_nominals=["PhD", "Esq.", "III"])
>>> Person.objects.update(post_nominals=ListF("post_nominals").append("DSocSci
↪"))
>>> Person.objects.get().full_name
"Horatio Phd Esq. III DSocSci"
```

appendleft(*value*)

Adds the value of the given expression to the (left hand) end of the list, like `deque.appendleft`:

```
>>> Person.objects.update(post_nominals=ListF("post_nominals").appendleft("BArch
↪"))
>>> Person.objects.get().full_name
"Horatio BArch Phd Esq. III DSocSci"
```

pop()

Takes one value from the (right hand) end of the list, like `list.pop`:

```
>>> Person.objects.update(post_nominals=ListF("post_nominals").pop())
>>> Person.objects.get().full_name
"Horatio BArch Phd Esq. III"
```

popleft()

Takes one value off the (left hand) end of the list, like `deque.popleft`:

```
>>> Person.objects.update(post_nominals=ListF("post_nominals").popleft())
>>> Person.objects.get().full_name
"Horatio Phd Esq. III"
```

Warning: All the above methods use SQL expressions with user variables in their queries, all of which start with `@tmp_`. This shouldn't affect you much, but if you use user variables in your queries, beware for any conflicts.

5.3 Set Fields

Legacy

These field classes are only maintained for legacy purposes. They aren't recommended as comma separation is a fragile serialization format.

For new uses, you're better off using Django 3.1's `JSONField` that works with all database backends. On earlier versions of Django, you can use django-jsonfield-backport.

Two fields that store sets of a base field in comma-separated strings - cousins of Django's `CommaSeparatedIntegerField`. There are two versions: `SetCharField`, which is based on `CharField` and appropriate for storing sets with a small maximum size, and `SetTextField`, which is based on `TextField` and therefore suitable for sets of (near) unbounded size (the underlying `LONGTEXT` MySQL datatype has a maximum length of 2^{32} - 1 bytes).

SetCharField(base_field, size=None, **kwargs):

A field for storing sets of data, which all conform to the `base_field`.

`django_mysql.models.`**base_field**

The base type of the data that is stored in the set. Currently, must be `IntegerField`, `CharField`, or any subclass thereof - except from `SetCharField` itself.

`django_mysql.models.`**size**

Optionally set the maximum number of elements in the set. This is only checked on form validation, not on model save!

As `SetCharField` is a subclass of `CharField`, any `CharField` options can be set too. Most importantly you'll need to set `max_length` to determine how many characters to reserve in the database.

Example instantiation:

```
from django.db.models import IntegerField, Model
from django_mysql.models import SetCharField
```

(continues on next page)

(continued from previous page)

```
class LotteryTicket(Model):
    numbers = SetCharField(
        base_field=IntegerField(),
        size=6,
        max_length=(6 * 3),  # 6 two digit numbers plus commas
    )
```

In Python simply set the field's value as a set:

```
>>> lt = LotteryTicket.objects.create(numbers={1, 2, 4, 8, 16, 32})
>>> lt.numbers
{1, 2, 4, 8, 16, 32}
>>> lt.numbers.remove(1)
>>> lt.numbers.add(3)
>>> lt.numbers
{32, 3, 2, 4, 8, 16}
>>> lt.save()
```

Validation on save()

When performing the set-to-string conversion for the database, `SetCharField` performs some validation, and will raise `ValueError` if there is a problem, to avoid saving bad data. The following are invalid:

- If there is a comma in any member's string representation
- If the empty string is stored.

The default form field is *SimpleSetField*.

SetTextField(base_field, size=None, **kwargs):

The same as `SetCharField`, but backed by a `TextField` and therefore much less restricted in length. There is no `max_length` argument.

Example instantiation:

```
from django.db.models import IntegerField, Model
from django_mysql.models import SetTextField


class Post(Model):
    tags = SetTextField(
        base_field=CharField(max_length=32),
    )
```

5.3.1 Querying Set Fields

Warning: These fields are not built-in datatypes, and the filters use one or more SQL functions to parse the underlying string representation. They may slow down on large tables if your queries are not selective on other columns.

contains

The `contains` lookup is overridden on `SetCharField` and `SetTextField` to match where the set field contains the given element, using MySQL's `FIND_IN_SET` (docs: MariaDB / MySQL).

For example:

```
>>> Post.objects.create(name="First post", tags={"thoughts", "django"})
>>> Post.objects.create(name="Second post", tags={"thoughts"})
>>> Post.objects.create(name="Third post", tags={"tutorial", "django"})

>>> Post.objects.filter(tags__contains="thoughts")
[<Post: First post>, <Post: Second post>]

>>> Post.objects.filter(tags__contains="django")
[<Post: First post>, <Post: Third post>]

>>> Post.objects.filter(Q(tags__contains="django") & Q(tags__contains="thoughts"))
[<Post: First post>]
```

Note: `ValueError` will be raised if you try `contains` with a set. It's not possible without using `AND` in the query, so you should add the filters for each item individually, as per the last example.

len

A transform that converts to the number of items in the set. For example:

```
>>> Post.objects.filter(tags__len=1)
[<Post: Second post>]

>>> Post.objects.filter(tags__len=2)
[<Post: First post>, <Post: Third post>]

>>> Post.objects.filter(tags__len__lt=2)
[<Post: Second post>]
```

5.3.2 SetF() expressions

Similar to Django's F expression, this allows you to perform an atomic add or remove on a set field at the database level:

```
>>> from django_mysql.models import SetF
>>> Post.objects.filter(tags__contains="django").update(
...     tags=SetF("tags").add("programming")
... )
2
>>> Post.objects.update(tags=SetF("tags").remove("thoughts"))
2
```

Or with attribute assignment to a model:

```
>>> post = Post.objects.earliest("id")
>>> post.tags = SetF("tags").add("python")
>>> post.save()
```

class django_mysql.models.**SetF**(*field_name*)

You should instantiate this class with the name of the field to use, and then call one of its two methods with a value to be added/removed.

Note that unlike F, you cannot chain the methods - the SQL involved is a bit too complicated, and thus you can only perform a single addition or removal.

add(*value*)

Takes an expression and returns a new expression that will take the value of the original field and add the value to the set if it is not contained:

```
post.tags = SetF("tags").add("python")
post.save()
```

remove(*value*)

Takes an expression and returns a new expression that will remove the given item from the set field if it is present:

```
post.tags = SetF("tags").remove("python")
post.save()
```

Warning: Both of the above methods use SQL expressions with user variables in their queries, all of which start with `@tmp_`. This shouldn't affect you much, but if you use user variables in your queries, beware for any conflicts.

5.4 EnumField

Using a `CharField` with a limited set of strings leads to inefficient data storage since the string value is stored over and over on disk. MySQL's `ENUM` type allows a more compact representation of such columns by storing the list of strings just once and using an integer in each row to refer to which string is there. `EnumField` allows you to use the `ENUM` type with Django.

Docs: MySQL / MariaDB.

class `django_mysql.models.`**EnumField**(*choices*, ***kwargs*)

A subclass of Django's `Charfield` that uses a MySQL `ENUM` for storage.

`choices` is a standard Django argument for any field class, however it is required for `EnumField`. It can either be a list of strings, or a list of two-tuples of strings, where the first element in each tuple is the value used, and the second the human readable name used in forms. The best way to form it is with Django's `TextChoices` enumeration type.

For example:

```
from django.db import models
from django_mysql.models import EnumField


class BookCoverColour(models.TextChoices):
    RED = "red"
    GREEN = "green"
    BLUE = "blue"


class BookCover(models.Model):
    colour = EnumField(choices=BookCoverColour.choices)
```

> **Warning:** It is possible to append new values to `choices` in migrations, as well as edit the *human readable* names of existing choices.
>
> However, editing or removing existing choice values will error if MySQL Strict Mode is on, and replace the values with the empty string if it is not.
>
> Also the empty string has strange behaviour with `ENUM`, acting somewhat like `NULL`, but not entirely. It is therefore recommended you ensure Strict Mode is on.

5.5 FixedCharField

Django's `CharField` uses the `VARCHAR` data type, which uses variable storage space depending on string length. This normally saves storage space, but for columns with a fixed length, it adds a small overhead.

The alternative `CHAR` data type avoids that overhead, but it has the surprising behaviour of removing trailing space characters, and consequently ignoring them in comparisons. `FixedCharField` provides a Django field for using `CHAR`. This can help you interface with databases created by other systems, but it's not recommended for general use, due to the trialing space behaviour.

Docs: MySQL / MariaDB.

class django_mysql.models.**FixedCharField**(**args*, *max_length: int*, ***kwargs*)

A subclass of Django's `Charfield` that uses the `CHAR` data type. `max_length` is as with `CharField`, but must be within the range 0-255.

For example:

```
from django_mysql.models import FixedCharField


class Address(Model):
    zip_code = FixedCharField(length=5)
```

5.6 Resizable Text/Binary Fields

Django's `TextField` and `BinaryField` fields are fixed at the MySQL level to use the maximum size class for the `BLOB` and `TEXT` data types. This is fine for most applications, however if you are working with a legacy database, or you want to be stricter about the maximum size of data that can be stored, you might want one of the other sizes.

The following field classes are simple subclasses that allow you to provide an extra parameter to determine which size class to use. They work with migrations, allowing you to swap them for the existing Django class and then use a migration to change their size class. This might help when taking over a legacy database for example.

Docs: MySQL / MariaDB.

class django_mysql.models.**SizedTextField**(*size_class: int*, ***kwargs*)

A subclass of Django's `TextField` that allows you to use the other sizes of `TEXT` data type. Set `size_class` to:

- `1` for a `TINYTEXT` field, which has a maximum length of 255 bytes
- `2` for a `TEXT` field, which has a maximum length of 65,535 bytes
- `3` for a `MEDIUMTEXT` field, which has a maximum length of 16,777,215 bytes (16MiB)
- `4` for a `LONGTEXT` field, which has a maximum length of 4,294,967,295 bytes (4GiB)

class django_mysql.models.**SizedBinaryField**(*size_class*, ***kwargs*)

A subclass of Django's `BinaryField` that allows you to use the other sizes of `BLOB` data type. Set `size_class` to:

- `1` for a `TINYBLOB` field, which has a maximum length of 255 bytes
- `2` for a `BLOB` field, which has a maximum length of 65,535 bytes
- `3` for a `MEDIUMBLOB` field, which has a maximum length of 16,777,215 bytes (16MiB)
- `4` for a `LONGBLOB` field, which has a maximum length of 4,294,967,295 bytes (4GiB)

5.7 BIT(1) Boolean Fields

Some database systems, such as the Java Hibernate ORM, don't use MySQL's `bool` data type for storing boolean flags and instead use `BIT(1)`. Django's default `BooleanField` and `NullBooleanField` classes can't work with this.

The following subclasses are boolean fields that work with `BIT(1)` columns that will help when connecting to a legacy database. If you are using `inspectdb` to generate models from the database, use these to replace the `TextField` output for your `BIT(1)` columns.

class Bit1BooleanField

A subclass of Django's `BooleanField` that uses the `BIT(1)` column type instead of `bool`.

class NullBit1BooleanField

Note: Django deprecated `NullBooleanField` in version 3.1 and retains it only for use in old migrations. `NullBit1BooleanField` is similarly deprecated.

A subclass of Django's `NullBooleanField` that uses the `BIT(1)` column type instead of `bool`.

CHAPTER

SIX

FIELD LOOKUPS

ORM extensions for filtering. These are all automatically added for the appropriate field types when `django_mysql` is in your `INSTALLED_APPS`. Note that lookups specific to included *model fields* are documented with the field, rather than here.

6.1 Case-sensitive String Comparison

MySQL string comparison has a case-sensitivity dependent on the collation of your tables/columns, as the Django manual describes. However, it is possible to query in a case-sensitive manner even when your data is not stored with a case-sensitive collation, using the `BINARY` keyword. The following lookup adds that capability to the ORM for `CharField`, `TextField`, and subclasses thereof.

6.1.1 case_exact

Exact, case-sensitive match for character columns, no matter the underlying collation:

```
>>> Author.objects.filter(name__case_exact="dickens")
[]
>>> Author.objects.filter(name__case_exact="Dickens")
[<Author: Dickens>]
```

6.2 Soundex

MySQL implements the Soundex algorithm with its `SOUNDEX` function, allowing you to find words sounding similar to each other (in English only, regrettably). These lookups allow you to use that function in the ORM and are added for `CharField` and `TextField`.

6.2.1 soundex

Match a given soundex string:

```
>>> Author.objects.filter(name__soundex="R163")
[<Author: Robert>, <Author: Rupert>]
```

SQL equivalent:

```
SELECT ... WHERE SOUNDEX(`name`) = 'R163'
```

6.2.2 sounds_like

Match the SOUNDEX of the given string:

```
>>> Author.objects.filter(name__sounds_like="Robert")
[<Author: Robert>, <Author: Rupert>]
```

SQL equivalent:

```
SELECT ... WHERE `name` SOUNDS LIKE 'Robert'
```

CHAPTER

SEVEN

AGGREGATES

MySQL-specific database aggregates for the ORM.

The following can be imported from `django_mysql.models`.

class django_mysql.models.**BitAnd**(*column*)

Returns an `int` of the bitwise `AND` of all input values, or 18446744073709551615 (a `BIGINT UNSIGNED` with all bits set to 1) if no rows match.

Docs: MySQL / MariaDB.

Example usage:

```
>>> Book.objects.create(bitfield=29)
>>> Book.objects.create(bitfield=15)
>>> Book.objects.all().aggregate(BitAnd("bitfield"))
{'bitfield__bitand': 13}
```

class django_mysql.models.**BitOr**(*column*)

Returns an `int` of the bitwise `OR` of all input values, or 0 if no rows match.

Docs: MySQL / MariaDB.

Example usage:

```
>>> Book.objects.create(bitfield=29)
>>> Book.objects.create(bitfield=15)
>>> Book.objects.all().aggregate(BitOr("bitfield"))
{'bitfield__bitor': 31}
```

class django_mysql.models.**BitXor**(*column*)

Returns an `int` of the bitwise `XOR` of all input values, or 0 if no rows match.

Docs: MySQL / MariaDB.

Example usage:

```
>>> Book.objects.create(bitfield=11)
>>> Book.objects.create(bitfield=3)
>>> Book.objects.all().aggregate(BitXor("bitfield"))
{'bitfield__bitxor': 8}
```

class django_mysql.models.**GroupConcat**(*column*, *distinct=False*, *separator=','*, *ordering=None*)

An aggregate that concatenates values from a column of the grouped rows. Useful mostly for bringing back lists of ids in a single query.

Docs: MySQL / MariaDB.

Example usage:

```
>>> from django_mysql.models import GroupConcat
>>> author = Author.objects.annotate(book_ids=GroupConcat("books__id")).get(
...     name="William Shakespeare"
... )
>>> author.book_ids
"1,2,5,17,29"
```

> **Warning:** MySQL will truncate the value at the value of `group_concat_max_len`, which by default is quite low at 1024 characters. You should probably increase it if you're using this for any sizeable groups.
>
> `group_concat_max_len` docs: MySQL / MariaDB.

Optional arguments:

`distinct=False`

If set to `True`, removes duplicates from the group.

`separator=','`

By default the separator is a comma. You can use any other string as a separator, including the empty string.

> **Warning:** Due to limitations in the Django aggregate API, this is not protected against SQL injection. Don't pass in user input for the separator.

`ordering=None`

By default no guarantee is made on the order the values will be in pre-concatenation. Set ordering to `'asc'` to sort them in ascending order, and `'desc'` for descending order. For example:

```
>>> Author.objects.annotate(book_ids=GroupConcat("books__id", ordering="asc"))
```

CHAPTER

EIGHT

DATABASE FUNCTIONS

MySQL/MariaDB-specific database functions for the ORM.

The following can be imported from `django_mysql.models.functions`.

8.1 Control Flow Functions

class django_mysql.models.functions.**If**(*condition*, *true*, *false=None*)

Evaluates the expression `condition` and returns the value of the expression `true` if true, and the result of expression `false` if false. If `false` is not given, it will be `Value(None)`, i.e. `NULL`.

Docs: MySQL / MariaDB.

Usage example:

```
>>> Author.objects.annotate(
...     is_william=If(Q(name__startswith="William "), True, False)
... ).values_list("name", "is_william")
[('William Shakespeare', True),
 ('Ian Fleming', False),
 ('William Wordsworth', True)]
```

8.2 Numeric Functions

class django_mysql.models.functions.**CRC32**(*expression*)

Computes a cyclic redundancy check value and returns a 32-bit unsigned value. The result is `NULL` if the argument is `NULL`. The argument is expected to be a string and (if possible) is treated as one if it is not.

Docs: MySQL / MariaDB.

Usage example:

```
>>> Author.objects.annotate(description_crc=CRC32("description"))
```

8.3 String Functions

class django_mysql.models.functions.**ConcatWS**(**expressions*, *separator=','*)

ConcatWS stands for Concatenate With Separator and is a special form of Concat (included in Django). It concatenates all of its argument expressions as strings with the given `separator`. Since `NULL` values are skipped, unlike in `Concat`, you can use the empty string as a separator and it acts as a `NULL`-safe version of `Concat`.

If `separator` is a string, it will be turned into a `Value`. If you wish to join with the value of a field, you can pass in an `F` object for that field.

Docs: MySQL / MariaDB.

Usage example:

```
>>> Author.objects.annotate(sales_list=ConcatWS("sales_eu", "sales_us"))
```

class django_mysql.models.functions.**ELT**(*number*, *values*)

Given a numerical expression `number`, it returns the `number`th element from `values`, 1-indexed. If `number` is less than 1 or greater than the number of expressions, it will return `None`. It is the complement of the `Field` function.

Note that if `number` is a string, it will refer to a field, whereas members of `values` that are strings will be wrapped with `Value` automatically and thus interpreted as the given string. This is for convenience with the most common usage pattern where you have the list pre-loaded in python, e.g. a `choices` field. If you want to refer to a column, use Django's `F()` class.

Docs: MySQL / MariaDB.

Usage example:

```
>>> # Say Person.life_state is either 1 (alive), 2 (dead), or 3 (M.I.A.)
>>> Person.objects.annotate(state_name=ELT("life_state", ["Alive", "Dead", "M.I.A.
↪"]))
```

class django_mysql.models.functions.**Field**(*expression*, *values*)

Given an `expression` and a list of strings `values`, returns the 1-indexed location of the `expression`'s value in `values`, or 0 if not found. This is commonly used with `order_by` to keep groups of elements together. It is the complement of the `ELT` function.

Note that if `expression` is a string, it will refer to a field, whereas if any member of `values` is a string, it will automatically be wrapped with `Value` and refer to the given string. This is for convenience with the most common usage pattern where you have the list of things pre-loaded in Python, e.g. in a field's `choices`. If you want to refer to a column, use Django's `F()` class.

Docs: MySQL / MariaDB.

Usage example:

```
>>> # Females, then males - but other values of gender (e.g. empty string) first
>>> Person.objects.all().order_by(Field("gender", ["Female", "Male"]))
```

8.4 XML Functions

class django_mysql.models.functions.**UpdateXML**(*xml_target*, *xpath_expr*, *new_xml*)

Returns the XML fragment `xml_target` with the single match for `xpath_expr` replaced with the xml fragment `new_xml`. If nothing matches `xpath_expr`, or if multiple matches are found, the original `xml_target` is returned unchanged.

This can be used for single-query updates of text fields containing XML.

Note that if `xml_target` is given as a string, it will refer to a column, whilst if either `xpath_expr` or `new_xml` are strings, they will be used as strings directly. If you want `xpath_expr` or `new_xml` to refer to columns, use Django's `F()` class.

Docs: MySQL / MariaDB.

Usage example:

```
# Remove 'sagacity' from all authors' xml_attrs
>>> Author.objects.update(xml_attrs=UpdateXML("xml_attrs", "/sagacity", ""))
```

class django_mysql.models.functions.**XMLExtractValue**(*xml_frag*, *xpath_expr*)

Returns the text (`CDATA`) of the first text node which is a child of the element(s) in the XML fragment `xml_frag` matched by the XPath expression `xpath_expr`. In SQL this function is called `ExtractValue`; the class has the `XML` prefix to make it clearer what kind of values are it extracts.

Note that if `xml_frag` is given as a string, it will refer to a column, whilst if `xpath_expr` is a string, it will be used as a string. If you want `xpath_expr` to refer to a column, use Django's `F()` class.

Docs: MySQL / MariaDB.

Usage example:

```
# Count the number of authors with 'sagacity' in their xml_attrs
>>> num_authors_with_sagacity = (
...     Author.objects.annotate(
...         has_sagacity=XMLExtractValue("xml_attrs", "count(/sagacity)")
...     )
...     .filter(has_sagacity="1")
...     .count()
... )
```

8.5 Regexp Functions

Note: These work with MariaDB 10.0.5+ only, which includes PCRE regular expressions and these extra functions to use them. More information can be found in its documentation.

class django_mysql.models.functions.**RegexpInstr**(*expression*, *regex*)

Returns the 1-indexed position of the first occurrence of the regular expression `regex` in the string value of `expression`, or 0 if it was not found.

Note that if `expression` is given as a string, it will refer to a column, whilst if `regex` is a string, it will be used as a string. If you want `regex` to refer to a column, use Django's `F()` class.

Docs: MariaDB.

Usage example:

```
>>> Author.objects.annotate(name_pos=RegexpInstr("name", r"ens")).filter(name_pos__
↪gt=0)
[<Author: Charles Dickens>, <Author: Robert Louis Stevenson>]
```

class django_mysql.models.functions.**RegexpReplace**(*expression*, *regex*, *replace*)

Returns the string value of `expression` with all occurrences of the regular expression `regex` replaced by the string `replace`. If no occurrences are found, then subject is returned as is.

Note that if `expression` is given as a string, it will refer to a column, whilst if either `regex` or `replace` are strings, they will be used as strings. If you want `regex` or `replace` to refer to columns, use Django's `F()` class.

Docs: MariaDB.

Usage example:

```
>>> Author.objects.create(name="Charles Dickens")
>>> Author.objects.create(name="Roald Dahl")
>>> qs = Author.objects.annotate(
...     surname_first=RegexpReplace("name", r"^(.*) (.*)$", r"\2, \1")
... ).order_by("surname_first")
>>> qs
[<Author: Roald Dahl>, <Author: Charles Dickens>]
>>> qs[0].surname_first
"Dahl, Roald"
```

class django_mysql.models.functions.**RegexpSubstr**(*expression*, *regex*)

Returns the part of the string value of `expression` that matches the regular expression `regex`, or an empty string if `regex` was not found.

Note that if `expression` is given as a string, it will refer to a column, whilst if `regex` is a string, it will be used as a string. If you want `regex` to refer to a column, use Django's `F()` class.

Docs: MariaDB.

Usage example:

```
>>> Author.objects.create(name="Euripides")
>>> Author.objects.create(name="Frank Miller")
>>> Author.objects.create(name="Sophocles")
>>> Author.objects.annotate(name_has_space=CharLength(RegexpSubstr("name", r"\s"))).
↪filter(
...     name_has_space=0
... )
[<Author: Euripides>, <Author: Sophocles>]
```

8.6 Information Functions

class django_mysql.models.functions.**LastInsertId**(*expression=None*)

With no argument, returns the last value added to an auto-increment column, or set by another call to `LastInsertId` with an argument. With an argument, sets the 'last insert id' value to the value of the given expression, and returns that value. This can be used to implement simple `UPDATE ... RETURNING` style queries.

This function also has a class method:

get(*using=DEFAULT_DB_ALIAS*)

Returns the value set by a call to `LastInsertId()` with an argument, by performing a single query. It is stored per-connection, hence you may need to pass the alias of the connection that set the `LastInsertId` as `using`.

Note: Any queries on the database connection between setting `LastInsertId` and calling `LastInsertId.get()` can reset the value. These might come from Django, which can issue multiple queries for `update()` with multi-table inheritance, or for `delete()` with cascading.

Docs: MySQL / MariaDB.

Usage examples:

```
>>> Countable.objects.filter(id=1).update(counter=LastInsertId("counter") + 1)
1
>>> # Get the pre-increase value of 'counter' as stored on the server
>>> LastInsertId.get()
242

>>> Author.objects.filter(id=1, age=LastInsertId("age")).delete()
1
>>> # We can also use the stored value directly in a query
>>> Author.objects.filter(id=2).update(age=LastInsertId())
1
>>> Author.objects.get(id=2).age
35
```

8.7 JSON Database Functions

These functions work with data stored in Django's `JSONField` on MySQL and MariaDB only. `JSONField` is built in to Django 3.1+ and can be installed on older Django versions with the django-jsonfield-backport package.

These functions use JSON paths to address content inside JSON documents - for more information on their syntax, refer to the docs: MySQL / MariaDB.

class django_mysql.models.functions.**JSONExtract**(*expression*, **paths*, *output_field=None*)

Given `expression` that resolves to some JSON data, extract the given JSON paths. If there is a single path, the plain value is returned; if there is more than one path, the output is a JSON array with the list of values represented by the paths. If the expression does not match for a particular JSON object, returns `NULL`.

If only one path is given, `output_field` may also be given as a model field instance like `IntegerField()`, into which Django will load the value; the default is `JSONField()`, as it supports all return types including the array of values for multiple paths.

Note that if `expression` is a string, it will refer to a field, whereas members of `paths` that are strings will be wrapped with `Value` automatically and thus interpreted as the given string. If you want any of `paths` to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

Usage examples:

```
>>> # Fetch a list of tuples (id, size_or_None) for all ShopItems
>>> ShopItem.objects.annotate(size=JSONExtract("attrs", "$.size")).values_list("id",
↪ "size")
[(1, '3m'), (3, '5nm'), (8, None)]
>>> # Fetch the distinct values of attrs['colours'][0] for all items
>>> ShopItem.objects.annotate(
...     primary_colour=JSONExtract("attrs", "$.colours[0]")
... ).distinct().values_list("primary_colour", flat=True)
['Red', 'Blue', None]
```

class django_mysql.models.functions.**JSONKeys**(*expression*, *path=None*)

Given `expression` that resolves to some JSON data containing a JSON object, return the keys in that top-level object as a JSON array, or if `path` is given, return the keys at that path. If the path does not match, or if `expression` is not a JSON object (e.g. it contains a JSON array instead), returns `NULL`.

Note that if `expression` is a string, it will refer to a field, whereas if `path` is a string it will be wrapped with `Value` automatically and thus interpreted as the given string. If you want `path` to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Fetch the top-level keys for the first item
>>> ShopItem.objects.annotate(keys=JSONKeys("attrs")).values_list("keys",␣
↪flat=True)[0]
['size', 'colours', 'age', 'price', 'origin']
>>> # Fetch the keys in 'origin' for the first item
>>> ShopItem.objects.annotate(keys=JSONKeys("attrs", "$.origin")).values_list(
...     "keys", flat=True
... )[0]
['continent', 'country', 'town']
```

class django_mysql.models.functions.**JSONLength**(*expression*, *path=None*)

Given `expression` that resolves to some JSON data, return the length of that data, or if `path` is given, return the length of the data at that path. If the path does not match, or if `expression` is `NULL` it returns `NULL`.

As per the MySQL documentation, the length of a document is determined as follows:

- The length of a scalar is 1.
- The length of an array is the number of array elements.
- The length of an object is the number of object members.
- The length does not count the length of nested arrays or objects.

Note that if `expression` is a string, it will refer to a field, whereas if `path` is a string it will be wrapped with `Value` automatically and thus interpreted as the given string. If you want `path` to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Which ShopItems don't have more than three colours?
>>> ShopItem.objects.annotate(num_colours=JSONLength("attrs", "$.colours")).filter(
...     num_colours__gt=3
... )
[<ShopItem: Rainbow Wheel>, <ShopItem: Hard Candies>]
```

class django_mysql.models.functions.**JSONInsert**(*expression*, *data*)

Given `expression` that resolves to some JSON data, adds to it using the dictionary `data` of JSON paths to new values. If any JSON path in the `data` dictionary does not match, or if `expression` is `NULL`, it returns `NULL`. Paths that already exist in the original data are ignored.

Note that if `expression` is a string, it will refer to a field, whereas keys and values within the `pairs` dictionary will be wrapped with `Value` automatically and thus interpreted as the given string. If you want a key or value to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Add power_level = 0 for those items that don't have power_level
>>> ShopItem.objects.update(attrs=JSONInsert("attrs", {"$.power_level": 0}))
```

class django_mysql.models.functions.**JSONReplace**(*expression*, *data*)

Given `expression` that resolves to some JSON data, replaces existing paths in it using the dictionary `data` of JSON paths to new values. If any JSON path within the `data` dictionary does not match, or if `expression` is `NULL`, it returns `NULL`. Paths that do not exist in the original data are ignored.

Note that if `expression` is a string, it will refer to a field, whereas keys and values within the `pairs` dictionary will be wrapped with `Value` automatically and thus interpreted as the given string. If you want a key or value to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Reset all items' monthly_sales to 0 directly in MySQL
>>> ShopItem.objects.update(attrs=JSONReplace("attrs", {"$.monthly_sales": 0}))
```

class django_mysql.models.functions.**JSONSet**(*expression*, *data*)

Given `expression` that resolves to some JSON data, updates it using the dictionary `data` of JSON paths to new values. If any of the JSON paths within the data dictionary does not match, or if `expression` is `NULL`, it returns `NULL`. All paths can be modified - those that did not exist before and those that did.

Note that if `expression` is a string, it will refer to a field, whereas keys and values within the `data` dictionary will be wrapped with `Value` automatically and thus interpreted as the given string. If you want a key or value to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Modify 'size' value to '10m' directly in MySQL
>>> shop_item = ShopItem.objects.latest()
>>> shop_item.attrs = JSONSet("attrs", {"$.size": "10m"})
>>> shop_item.save()
```

class django_mysql.models.functions.**JSONArrayAppend**(*expression*, *data*)

Given `expression` that resolves to some JSON data, adds to it using the dictionary `data` of JSON paths to new values. If a path selects an array, the new value will be appended to it. On the other hand, if a path selects a scalar or object value, that value is autowrapped within an array and the new value is added to that array. If any of the JSON paths within the data dictionary does not match, or if `expression` is `NULL`, it returns `NULL`.

Note that if `expression` is a string, it will refer to a field, whereas keys and values within the `data` dictionary will be wrapped with `Value` automatically and thus interpreted as the given string. If you want a key or value to refer to a field, use Django's `F()` class.

Docs: MySQL / MariaDB.

```
>>> # Append the string '10m' to the array 'sizes' directly in MySQL
>>> shop_item = ShopItem.objects.latest()
>>> shop_item.attrs = JSONArrayAppend("attrs", {"$.sizes": "10m"})
>>> shop_item.save()
```

8.8 Dynamic Columns Functions

These are MariaDB 10.0+ only, and for use with `DynamicField`.

class `django_mysql.models.functions.`**AsType**(*expression*, *data_type*)

A partial function that should be used as part of a `ColumnAdd` expression when you want to ensure that `expression` will be stored as a given type `data_type`. The possible values for `data_type` are the same as documented for the `DynamicField` lookups.

Note that this is not a valid standalone function and must be used as part of `ColumnAdd` - see below.

class `django_mysql.models.functions.`**ColumnAdd**(*expression*, *to_add*)

Given `expression` that resolves to a `DynamicField` (most often a field name), add/update with the dictionary `to_add` and return the new Dynamic Columns value. This can be used for atomic single-query updates on Dynamic Columns.

Note that you can add optional types (and you should!). These can not be drawn from the `spec` of the `DynamicField` due to ORM restrictions, so there are no guarantees about the types that will get used if you do not. To add a type cast, wrap the value with an `AsType` (above) - see examples below.

Docs: MariaDB.

Usage examples:

```
>>> # Add default 'for_sale' as INTEGER 1 to every item
>>> ShopItem.objects.update(attrs=ColumnAdd("attrs", {"for_sale": AsType(1, "INTEGER
↪")}))
>>> # Fix some data
>>> ShopItem.objects.filter(attrs__size="L").update(
...     attrs=ColumnAdd("attrs", {"size": AsType("Large", "CHAR")})
... )
```

class `django_mysql.models.functions.`**ColumnDelete**(*expression*, **to_delete*)

Given `expression` that resolves to a `DynamicField` (most often a field name), delete the columns listed by the other expressions `to_delete`, and return the new Dynamic Columns value. This can be used for atomic single-query deletions on Dynamic Columns.

Note that strings in `to_delete` will be wrapped with `Value` automatically and thus interpreted as the given string - if they weren't, Django would interpret them as meaning "the value in this (non-dynamic) column". If you do mean that, use `F('fieldname')`.

Docs: MariaDB.

Usage examples:

```
>>> # Remove 'for_sail' and 'for_purchase' from every item
>>> ShopItem.objects.update(attrs=ColumnDelete("attrs", "for_sail", "for_purchase"))
```

class django_mysql.models.functions.**ColumnGet**(*expression*, *name*, *data_type*)

Given `expression` that resolves to a `DynamicField` (most often a field name), return the value of the column `name` when cast to the type `data_type`, or `NULL` / `None` if the column does not exist. This can be used to select a subset of column values when you don't want to fetch the whole blob. The possible values for `data_type` are the same as documented for the `DynamicField` lookups.

Docs: MariaDB.

Usage examples:

```
>>> # Fetch a list of tuples (id, size_or_None) for all items
>>> ShopItem.objects.annotate(size=ColumnGet("attrs", "size", "CHAR")).values_list(
...     "id", "size"
... )
>>> # Fetch the distinct values of attrs['seller']['url'] for all items
>>> ShopItem.objects.annotate(
...     seller_url=ColumnGet(ColumnGet("attrs", "seller", "BINARY"), "url", "CHAR")
... ).distinct().values_list("seller_url", flat=True)
```

CHAPTER

NINE

MIGRATION OPERATIONS

MySQL-specific migration operations that can all be imported from `django_mysql.operations`.

9.1 Install Plugin

class django_mysql.operations.**InstallPlugin**(*name*, *soname*)

An `Operation` subclass that installs a MySQL plugin. Runs `INSTALL PLUGIN name SONAME soname`, but does a check to see if the plugin is already installed to make it more idempotent.

Docs: MySQL / MariaDB.

name

This is a required argument. The name of the plugin to install.

soname

This is a required argument. The name of the library to install the plugin from. Note that on MySQL you must include the extension (e.g. `.so`, `.dll`) whilst on MariaDB you may skip it to keep the operation platform-independent.

Example usage:

```
from django.db import migrations
from django_mysql.operations import InstallPlugin


class Migration(migrations.Migration):
    dependencies = []

    operations = [
        # Install https://mariadb.com/kb/en/mariadb/metadata_lock_info/
        InstallPlugin("metadata_lock_info", "metadata_lock_info.so")
    ]
```

9.2 Install SOName

class django_mysql.operations.**InstallSOName**(*soname*)

MariaDB only.

An `Operation` subclass that installs a MariaDB plugin library. One library may contain multiple plugins that work together, this installs all of the plugins in the named library file. Runs `INSTALL SONAME soname`. Note that unlike `InstallPlugin`, there is no idempotency check to see if the library is already installed, since there is no way of knowing if all the plugins inside the library are installed.

Docs: MariaDB.

soname

This is a required argument. The name of the library to install the plugin from. You may skip the file extension (e.g. `.so`, `.dll`) to keep the operation platform-independent.

Example usage:

```
from django.db import migrations
from django_mysql.operations import InstallSOName


class Migration(migrations.Migration):
    dependencies = []

    operations = [
        # Install https://mariadb.com/kb/en/mariadb/metadata_lock_info/
        InstallSOName("metadata_lock_info")
    ]
```

9.3 Alter Storage Engine

class django_mysql.operations.**AlterStorageEngine**(*name*, *to_engine*, *from_engine=None*)

An `Operation` subclass that alters the model's table's storage engine. Because Django has no knowledge of storage engines, you must provide the previous storage engine for the operation to be reversible.

name

This is a required argument. The name of the model to alter.

to_engine

This is a required argument. The storage engine to move the model to.

from_engine

This is an optional argument. The storage engine the model is moving from. If you do not provide this, the operation is not reversible.

Note: If you're using this to move from MyISAM to InnoDB, there's a page for you in the MariaDB knowledge base - Converting Tables from MyISAM to InnoDB.

Example usage:

```
from django.db import migrations
from django_mysql.operations import AlterStorageEngine


class Migration(migrations.Migration):
    dependencies = []

    operations = [AlterStorageEngine("Pony", from_engine="MyISAM", to_engine="InnoDB
↪")]
```

CHAPTER

TEN

FORM FIELDS

The following can be imported from `django_mysql.forms`.

10.1 SimpleListField

class `django_mysql.forms.`**SimpleListField**(*base_field*, *max_length=None*, *min_length=None*)

A simple field which maps to a list, with items separated by commas. It is represented by an HTML `<input>`. Empty items, resulting from leading, trailing, or double commas, are disallowed.

base_field

This is a required argument.

It specifies the underlying form field for the set. It is not used to render any HTML, but it does process and validate the submitted data. For example:

```
>>> from django import forms
>>> from django_mysql.forms import SimpleListField

>>> class NumberListForm(forms.Form):
...     numbers = SimpleListField(forms.IntegerField())
...

>>> form = NumberListForm({"numbers": "1,2,3"})
>>> form.is_valid()
True
>>> form.cleaned_data
{'numbers': [1, 2, 3]}

>>> form = NumberListForm({"numbers": "1,2,a"})
>>> form.is_valid()
False
```

max_length

This is an optional argument which validates that the list does not exceed the given length.

min_length

This is an optional argument which validates that the list reaches at least the given length.

SimpleListField is not particularly user friendly in most cases, however it's better than nothing.

10.2 SimpleSetField

class django_mysql.forms.**SimpleSetField**(*base_field*, *max_length=None*, *min_length=None*)

A simple field which maps to a set, with items separated by commas. It is represented by an HTML <input>. Empty items, resulting from leading, trailing, or double commas, are disallowed.

base_field

This is a required argument.

It specifies the underlying form field for the set. It is not used to render any HTML, but it does process and validate the submitted data. For example:

```
>>> from django import forms
>>> from django_mysql.forms import SimpleSetField

>>> class NumberSetForm(forms.Form):
...     numbers = SimpleSetField(forms.IntegerField())
...

>>> form = NumberSetForm({"numbers": "1,2,3"})
>>> form.is_valid()
True
>>> form.cleaned_data
{'numbers': set([1, 2, 3])}

>>> form = NumberSetForm({"numbers": "1,2,a"})
>>> form.is_valid()
False
```

max_length

This is an optional argument which validates that the set does not exceed the given length.

min_length

This is an optional argument which validates that the set reaches at least the given length.

User friendly forms

SimpleSetField is not particularly user friendly in most cases, however it's better than nothing.

CHAPTER

ELEVEN

VALIDATORS

The following can be imported from `django_mysql.validators`.

class `django_mysql.validators.`**ListMaxLengthValidator**

A subclass of django's `MaxLengthValidator` with list-specific wording.

class `django_mysql.validators.`**ListMinLengthValidator**

A subclass of django's `MinLengthValidator` with list-specific wording.

class `django_mysql.validators.`**SetMaxLengthValidator**

A subclass of django's `MaxLengthValidator` with set-specific wording.

class `django_mysql.validators.`**SetMinLengthValidator**

A subclass of django's `MinLengthValidator` with set-specific wording.

CHAPTER

TWELVE

CACHE

A MySQL-specific backend for Django's cache framework.

12.1 MySQLCache

An efficient cache backend using a MySQL table, an alternative to Django's database-agnostic `DatabaseCache`. It has the following advantages:

- Each operation uses only one query, including the `*_many` methods. This is unlike `DatabaseCache` which uses multiple queries for nearly every operation.
- Automatic client-side `zlib` compression for objects larger than a given threshold. It is also easy to subclass and add your own serialization or compression schemes.
- Faster probabilistic culling behaviour during write operations, which you can also turn off and execute in a background task. This can be a bottleneck with Django's `DatabaseCache` since it culls on every write operation, executing a `SELECT COUNT(*)` which requires a full table scan.
- Integer counters with atomic `incr()` and `decr()` operations, like the `MemcachedCache` backend.

12.1.1 Usage

To use, add an entry to your `CACHES` setting with:

- `BACKEND` set to `django_mysql.cache.MySQLCache`
- `LOCATION` set to `tablename`, the name of the table to use. This name can be whatever you want, as long as it's a valid table name that's not already being used in your database.

For example:

```
CACHES = {
    "default": {
        "BACKEND": "django_mysql.cache.MySQLCache",
        "LOCATION": "my_super_cache",
    }
}
```

You then need to make the table. The schema is *not* compatible with that of `DatabaseCache`, so if you are switching, you will need to create a fresh table.

Use the management command `mysql_cache_migration` to output a migration that creates tables for all the `MySQLCache` instances you have configured. For example:

```
$ python manage.py mysql_cache_migration
from django.db import migrations


class Migration(migrations.Migration):

    dependencies = [
        # Add a dependency in here on an existing migration in the app you
        # put this migration in, for example:
        # ('myapp', '0001_initial'),
    ]

    operations = [
        migrations.RunSQL(
            """
            CREATE TABLE `my_super_cache` (
                cache_key varchar(255) CHARACTER SET utf8 COLLATE utf8_bin
                                       NOT NULL PRIMARY KEY,
                value longblob NOT NULL,
                value_type char(1) CHARACTER SET latin1 COLLATE latin1_bin
                                   NOT NULL DEFAULT 'p',
                expires BIGINT UNSIGNED NOT NULL
            );
            """,
            "DROP TABLE `my_super_cache`"
        ),
    ]
```

Save this to a file in the `migrations` directory of one of your project's apps, and add one of your existing migrations to the file's `dependencies`. You might want to customize the SQL at this time, for example switching the table to use the `MEMORY` storage engine.

Django requires you to install sqlparse to run the `RunSQL` operation in the migration, so make sure it is installed.

Once the migration has run, the cache is ready to work!

12.1.2 Multiple Databases

If you use this with multiple databases, you'll also need to set up routing instructions for the cache table. This can be done with the same method that is described for `DatabaseCache` in the Django manual, apart from the application name is `django_mysql`.

Note: Even if you aren't using multiple MySQL databases, it may be worth using routing anyway to put all your cache operations on a second connection - this way they won't be affected by transactions your main code runs.

12.1.3 Extra Details

`MySQLCache` is fully compatible with Django's cache API, but it also extends it and there are, of course, a few details to be aware of.

incr/decr

Like `MemcachedCache` (and unlike `DatabaseCache`), `incr` and `decr` are atomic operations, and can only be used with `int` values. They have the range of MySQL's `SIGNED BIGINT` (-9223372036854775808 to 9223372036854775807).

max_allowed_packet

MySQL has a setting called `max_allowed_packet`, which is the maximum size of a query, including data. This therefore constrains the size of a cached value, but you're more likely to run up against it first with the `get_many`/`set_many` operations.

The MySQL 8.0 default is 4MB, and the MariaDB 10.2 default is 16MB. Most applications should be fine with these limits. You can tweak the setting as high as 1GB - if this isn't enough, you should probably be considering another solution!

culling

MySQL is designed to store data forever, and thus doesn't have a direct way of setting expired rows to disappear. The expiration of old keys and the limiting of rows to `MAX_ENTRIES` is therefore performed in the cache backend by performing a cull operation when appropriate. This deletes expired keys first, then if more than `MAX_ENTRIES` keys remain, it deletes 1 / `CULL_FREQUENCY` of them. The options and strategy are described in in more detail in the Django manual.

Django's `DatabaseCache` performs a cull check on *every* write operation. This runs a `SELECT COUNT(*)` on the table, which means a full-table scan. Naturally, this takes a bit of time and becomes a bottleneck for medium or large cache table sizes of caching. `MySQLCache` helps you solve this in two ways:

1. The cull-on-write behaviour is probabilistic, by default running on 1% of writes. This is set with the `CULL_PROBABILITY` option, which should be a number between 0 and 1. For example, if you want to use the same cull-on-*every*-write behaviour as used by `DatabaseCache` (you probably don't), set `CULL_PROBABILITY` to 1.0:

```
CACHES = {
    "default": {
        "BACKEND": "django_mysql.cache.MySQLCache",
        "LOCATION": "some_table_name",
        "OPTIONS": {"CULL_PROBABILITY": 1.0},
    }
}
```

2. The `cull()` method is available as a public method so you can set up your own culling schedule in background processing, never affecting any user-facing web requests. Set `CULL_PROBABILITY` to 0, and then set up your task. For example, if you are using **celery** you could use a task like this:

```
@shared_task
def clear_caches():
    caches["default"].cull()
    caches["other_cache"].cull()
```

This functionality is also available as the management command `cull_mysql_caches`, which you might run as a cron job. It performs `cull()` on all of your `MySQLCache` instances, or you can give it names to just cull those. For example, this:

```
$ python manage.py cull_mysql_caches default other_cache
```

...will call `caches['default'].cull()` and `caches['other_cache'].cull()`.

You can also disable the `MAX_ENTRIES` behaviour, which avoids the `SELECT COUNT(*)` entirely, and makes `cull()` only delete expired keys. To do this, set `MAX_ENTRIES` to -1:

```
CACHES = {
    "default": {
        "BACKEND": "django_mysql.cache.MySQLCache",
        "LOCATION": "some_table_name",
        "OPTIONS": {"MAX_ENTRIES": -1},
    }
}
```

Note that you should then of course monitor the size of your cache table well, since it has no bounds on its growth.

compression

Like the other Django cache backends, stored objects are serialized with `pickle` (except from integers, which are stored as integers so that the `incr()` and `decr()` operations will work). If pickled data has has a size in bytes equal to or greater than the threshold defined by the option `COMPRESS_MIN_LENGTH`, it will be compressed with `zlib` in Python before being stored, reducing the on-disk size in MySQL and network costs for storage and retrieval. The zlib level is set by the option `COMPRESS_LEVEL`.

`COMPRESS_MIN_LENGTH` defaults to 5000, and `COMPRESS_LEVEL` defaults to the `zlib` default of 6. You can tune these options - for example, to compress all objects >= 100 bytes at the maximum level of 9, pass the options like so:

```
CACHES = {
    "default": {
        "BACKEND": "django_mysql.cache.MySQLCache",
        "LOCATION": "some_table_name",
        "OPTIONS": {"COMPRESS_MIN_LENGTH": 100, "COMPRESS_LEVEL": 9},
    }
}
```

To turn compression off, set `COMPRESS_MIN_LENGTH` to 0. The options only affect new writes - any compressed values already in the table will remain readable.

custom serialization

You can implement your own serialization by subclassing `MySQLCache`. It uses two methods that you should override.

Values are stored in the table with two columns - `value`, which is the blob of binary data, and `value_type`, a single latin1 character that specifies the type of data in `value`. MySQLCache by default uses three codes for `value_type`:

- `i` - The blob is an integer. This is used so that counters can be deserialized by MySQL during the atomic `incr()` and `decr()` operations.
- `p` - The blob is a pickled Python object.
- `z` - The blob is a zlib-compressed pickled Python object.

For future compatibility, `MySQLCache` reserves all lower-case letters. For custom types you can use upper-case letters.

The methods you need to override (and probably call `super()` from) are:

`django_mysql.cache.`**`encode`**(*obj*)

Takes an object and returns a tuple `(value, value_type)`, ready to be inserted as parameters into the SQL query.

`django_mysql.cache.`**`decode`**(*value*, *value_type*)

Takes the pair of `(value, value_type)` as stored in the table and returns the deserialized object.

Studying the source of `MySQLCache` will probably give you the best way to extend these methods for your use case.

prefix methods

Three extension methods are available to work with sets of keys sharing a common prefix. Whilst these would not be efficient on other cache backends such as memcached, in an InnoDB table the keys are stored in order so range scans are easy.

To use these methods, it must be possible to reverse-map the "full" key stored in the databse to the key you would provide to `cache.get`, via a 'reverse key function'. If you have not set `KEY_FUNCTION`, `MySQLCache` will use Django's default key function, and can therefore default the reverse key function too, so you will not need to add anything.

However, if you have set `KEY_FUNCTION`, you will also need to supply `REVERSE_KEY_FUNCTION` before the prefix methods can work. For example, with a simple custom key function that ignores `key_prefix` and `version`, you might do this:

```
def my_key_func(key, key_prefix, version):
    return key  # Ignore key_prefix and version


def my_reverse_key_func(full_key):
    # key_prefix and version still need to be returned
    key_prefix = None
    version = None
    return key, key_prefix, version


CACHES = {
    "default": {
        "BACKEND": "django_mysql.cache.MySQLCache",
        "LOCATION": "some_table_name",
        "KEY_FUNCTION": my_key_func,
        "REVERSE_KEY_FUNCTION": my_reverse_key_func,
    }
}
```

Once you're set up, the following prefix methods can be used:

`django_mysql.cache.`**`delete_with_prefix`**(*prefix*, *version=None*)

Deletes all keys that start with the string `prefix`. If `version` is not provided, it will default to the `VERSION` setting. Returns the number of keys that were deleted. For example:

```
>>> cache.set_many({"Car1": "Blue", "Car4": "Red", "Truck3": "Yellow"})
>>> cache.delete_with_prefix("Truck")
1
```

(continues on next page)

(continued from previous page)

```
>>> cache.get("Truck3")
None
```

Note: This method does not require you to set the reverse key function.

`django_mysql.cache.`**`get_with_prefix`**(*prefix*, *version=None*)

Like `get_many`, returns a dict of key to value for all keys that start with the string `prefix`. If `version` is not provided, it will default to the `VERSION` setting. For example:

```
>>> cache.set_many({"Car1": "Blue", "Car4": "Red", "Truck3": "Yellow"})
>>> cache.get_with_prefix("Truck")
{'Truck3': 'Yellow'}
>>> cache.get_with_prefix("Ca")
{'Car1': 'Blue', 'Car4': 'Red'}
>>> cache.get_with_prefix("")
{'Car1': 'Blue', 'Car4': 'Red', 'Truck3': 'Yellow'}
```

`django_mysql.cache.`**`keys_with_prefix`**(*prefix*, *version=None*)

Returns a set of all the keys that start with the string `prefix`. If `version` is not provided, it will default to the `VERSION` setting. For example:

```
>>> cache.set_many({"Car1": "Blue", "Car4": "Red", "Truck3": "Yellow"})
>>> cache.keys_with_prefix("Car")
set(['Car1', 'Car2'])
```

12.1.4 Changes

Versions 0.1.10 -> 0.2.0

Initially, in Django-MySQL version 0.1.10, `MySQLCache` did not force the columns to use case sensitive collations; in version 0.2.0 this was fixed. You can upgrade by adding a migration with the following SQL, if you replace `yourtablename`:

```
ALTER TABLE yourtablename
    MODIFY cache_key varchar(255) CHARACTER SET utf8 COLLATE utf8_bin
                                  NOT NULL,
    MODIFY value_type char(1) CHARACTER SET latin1 COLLATE latin1_bin
                              NOT NULL DEFAULT 'p';
```

Or as a reversible migration:

```
from django.db import migrations


class Migration(migrations.Migration):
    dependencies = []

    operations = [
        migrations.RunSQL(
```

(continues on next page)

(continued from previous page)

```
            """
            ALTER TABLE yourtablename
                MODIFY cache_key varchar(255) CHARACTER SET utf8 COLLATE utf8_bin
                                              NOT NULL,
                MODIFY value_type char(1) CHARACTER SET latin1 COLLATE latin1_bin
                                          NOT NULL DEFAULT 'p'
            """,
            """
            ALTER TABLE yourtablename
                MODIFY cache_key varchar(255) CHARACTER SET utf8 NOT NULL,
                MODIFY value_type char(1) CHARACTER SET latin1 NOT NULL DEFAULT 'p'
            """,
        )
    ]
```

CHAPTER

THIRTEEN

LOCKS

The following can be imported from `django_mysql.locks`.

class `django_mysql.locks.`**Lock**(*name*, *acquire_timeout=10.0*, *using=None*)

MySQL can act as a locking server for arbitrary named locks (created on the fly) via its `GET_LOCK` function - sometimes called 'User Locks' since they are user-specific, and don't lock tables or rows. They can be useful for your code to limit its access to some shared resource.

This class implements a user lock and acts as either a context manager (recommended), or a plain object with `acquire` and `release` methods similar to `threading.Lock`. These call the MySQL functions `GET_LOCK`, `RELEASE_LOCK`, and `IS_USED_LOCK` to manage it.

The lock is only re-entrant (acquirable multiple times) on MariaDB.

Basic usage:

```
from django_mysql.exceptions import TimeoutError
from django_mysql.locks import Lock

try:
    with Lock("my_unique_name", acquire_timeout=2.0):
        mutually_exclusive_process()
except TimeoutError:
    print("Could not get the lock")
```

For more information on user locks refer to the `GET_LOCK` documentation on MySQL or MariaDB.

Warning: As the documentation warns, user locks are unsafe to use if you have replication running and your replication format (`binlog_format`) is set to `STATEMENT`. Most environments have `binlog_format` set to `MIXED` because it can be more performant, but do check.

name

This is a required argument.

Specifies the name of the lock. Since user locks share a global namespace on the MySQL server, it will automatically be prefixed with the name of the database you use in your connection from DATABASES and a full stop, in case multiple apps are using different databases on the same server.

MySQL enforces a maximum length on the total name (including the DB prefix that Django-MySQL adds) of 64 characters. MariaDB doesn't enforce any limit. The practical limit on MariaDB is maybe 1 million characters or more, so most sane uses should be fine.

acquire_timeout=10.0

The time in seconds to wait to acquire the lock, as will be passed to `GET_LOCK()`. Defaults to 10 seconds.

using=None

The connection alias from `DATABASES` to use. Defaults to Django's `DEFAULT_DB_ALIAS` to use your main database connection.

is_held()

Returns True iff a query to `IS_USED_LOCK()` reveals that this lock is currently held.

holding_connection_id()

Returns the MySQL `CONNECTION_ID()` of the holder of the lock, or `None` if it is not currently held.

acquire()

For using the lock as a plain object rather than a context manager, similar to `threading.Lock.acquire`. Note you should normally use `try` / `finally` to ensure unlocking occurs.

Example usage:

```
from django_mysql.locks import Lock

lock = Lock("my_unique_name")
lock.acquire()
try:
    mutually_exclusive_process()
finally:
    lock.release()
```

release()

Also for using the lock as a plain object rather than a context manager, similar to `threading.Lock.release`. For example, see above.

classmethod held_with_prefix(*prefix*, *using=DEFAULT_DB_ALIAS*)

Queries the held locks that match the given prefix, for the given database connection. Returns a dict of lock names to the `CONNECTION_ID()` that holds the given lock.

Example usage:

```
>>> Lock.held_with_prefix("Author")
{'Author.1': 451, 'Author.2': 457}
```

Note: Works with MariaDB 10.0.7+ only, when the `metadata_lock_info` plugin is loaded. You can install this in a migration using the *InstallSOName* operation, like so:

```
from django.db import migrations
from django_mysql.operations import InstallSOName


class Migration(migrations.Migration):
    dependencies = []

    operations = [
        # Install https://mariadb.com/kb/en/mariadb/metadata_lock_info/
        InstallSOName("metadata_lock_info")
    ]
```

---

class django_mysql.locks.**TableLock**(*write=None*, *read=none*, *using=None*)

MySQL allows you to gain a table lock to prevent modifications to the data during reads or writes. Most applications don't need to do this since transaction isolation should provide enough separation between operations, but occasionally this can be useful, especially in data migrations or if you are using a non-transactional storage such as MyISAM.

This class implements table locking and acts as either a context manager (recommended), or a plain object with `acquire()` and `release()` methods similar to `threading.Lock`. It uses the transactional pattern from the MySQL manual to ensure all the necessary steps are taken to lock tables properly. Note that locking has no timeout and blocks until held.

Basic usage:

```
from django_mysql.locks import TableLock

with TableLock(read=[MyModel1], write=[MyModel2]):
    fix_bad_instances_of_my_model2_using_my_model1_data()
```

Docs: MySQL / MariaDB.

read

A list of models or raw table names to lock at the `READ` level. Any models using multi-table inheritance will also lock their parents.

write

A list of models or raw table names to lock at the `WRITE` level. Any models using multi-table inheritance will also lock their parents.

using=None

The connection alias from `DATABASES` to use. Defaults to Django's `DEFAULT_DB_ALIAS` to use your main database connection.

acquire()

For using the lock as a plain object rather than a context manager, similar to `threading.Lock.acquire`. Note you should normally use `try` / `finally` to ensure unlocking occurs.

Example usage:

```
from django_mysql.locks import TableLock

table_lock = TableLock(read=[MyModel1], write=[MyModel2])
table_lock.acquire()
try:
    fix_bad_instances_of_my_model2_using_my_model1_data()
finally:
    table_lock.release()
```

release()

Also for using the lock as a plain object rather than a context manager, similar to `threading.Lock.release`. For example, see above.

Note: Transactions are not allowed around table locks, and an error will be raised if you try and use one inside of a transaction. A transaction is created to hold the locks in order to cooperate with InnoDB. There are a number of things you can't do whilst holding a table lock, for example accessing tables other than those you have locked - see the MySQL/MariaDB documentation for more details.

Note: Table locking works on InnoDB tables only if the `innodb_table_locks` is set to 1. This is the default, but may have been changed for your environment.

CHAPTER

FOURTEEN

STATUS

MySQL gives you metadata on the server status through its `SHOW GLOBAL STATUS` and `SHOW SESSION STATUS` commands. These classes make it easy to get this data, as well as providing utility methods to react to it.

The following can all be imported from `django_mysql.status`.

class `django_mysql.status.`**GlobalStatus**(*name*, *using=None*)

Provides easy access to the output of `SHOW GLOBAL STATUS`. These statistics are useful for monitoring purposes, or ensuring queries your code creates aren't saturating the server.

Basic usage:

```
from django_mysql.status import global_status

# Wait until a quiet moment
while global_status.get("Threads_running") >= 5:
    time.sleep(1)

# Log all status variables
logger.log("DB status", extra=global_status.as_dict())
```

Note that `global_status` is a pre-existing instance for the default database connection from `DATABASES`. If you're using more than database connection, you should instantiate the class:

```
>>> from django_mysql.status import GlobalStatus
>>> GlobalStatus(using="replica1").get("Threads_running")
47
```

To see the names of all the available variables, refer to the documentation: MySQL / MariaDB. They vary based upon server version, plugins installed, etc.

using=None

The connection alias from `DATABASES` to use. Defaults to Django's `DEFAULT_DB_ALIAS` to use your main database connection.

get(*name*)

Returns the current value of the named status variable. The name may not include SQL wildcards (`%`). If it does not exist, `KeyError` will be raised.

The result set for `SHOW STATUS` returns values in strings, so numbers and booleans will be cast to their respective Python types - `int`, `float`, or `bool`. Strings are be left as-is.

get_many(*names*)

Returns a dictionary of names to current values, fetching them in a single query. The names may not include wildcards (`%`).

Uses the same type-casting strategy as `get()`.

as_dict(*prefix=None*)

Returns a dictionary of names to current values. If `prefix` is given, only those variables starting with the prefix will be returned. `prefix` should not end with a wildcard (`%`) since that will be automatically appended.

Uses the same type-casting strategy as `get()`.

wait_until_load_low(*thresholds={'Threads_running': 10}*, *timeout=60.0*, *sleep=0.1*)

A helper method similar to the logic in `pt-online-schema-change` for waiting with –max-load.

Polls global status every `sleep` seconds until every variable named in `thresholds` is at or below its specified threshold, or raises a *django_mysql.exceptions.TimeoutError* if this does not occur within `timeout` seconds. Set `timeout` to 0 to never time out.

`thresholds` defaults to `{'Threads_running': 10}`, which is the default variable used in `pt-online-schema-change`, but with a lower threshold of 10 that is more suitable for small servers. You will very probably need to tweak it to your server.

You can use this method during large background operations which you don't want to affect other connections (i.e. your website). By processing in small chunks and waiting for low load in between, you sharply reduce your risk of outage.

class django_mysql.status.**SessionStatus**(*name*, *connection_name=None*)

This class is the same as GlobalStatus apart from it runs `SHOW SESSION STATUS`, so *some* variables are restricted to the current connection only, rather than the whole server. For which, you should refer to the documentation: MySQL / MariaDB.

Also it doesn't have the `wait_until_load_low` method, which only makes sense in a global setting.

Example usage:

```
from django_mysql.status import session_status

read_operations = session_status.get("Handler_read")
```

And for a different connection:

```
from django_mysql.status import SessionStatus

replica1_reads = SessionStatus(using="replica1").get("Handler_read")
```

CHAPTER

FIFTEEN

MANAGEMENT COMMANDS

MySQL-specific management commands. These are automatically available with your `manage.py` when you add `django_mysql` to your `INSTALLED_APPS`.

15.1 `dbparams` command

Outputs your database connection parameters in a form suitable for inclusion in other CLI commands, helping avoid copy/paste errors and accidental copying of passwords to shell history files. Knows how to output parameters in two formats - for `mysql` related tools, or the DSN format that some percona tools take. For example:

```
$ python manage.py dbparams && echo  # 'echo' adds a newline
--user=ausername --password=apassword --host=ahost.example.com mydatabase
$ mysql $(python manage.py dbparams)  # About the same as 'manage.py dbshell'
$ mysqldump $(python manage.py dbparams) | gzip -9 > backup.sql.gz  # Neat!
```

The format of parameters is:

```
python manage.py dbparams [--mysql | --dsn] <optional-connection-alias>
```

If the database alias is given, it should be alias of a connection from the `DATABASES` setting; defaults to 'default'. Only MySQL connections are supported - the command will fail for other connection vendors.

Mutually exclusive format flags:

15.1.1 `--mysql`

Default, so shouldn't need passing. Allows you to do, e.g.:

```
$ mysqldump $(python manage.py dbparams) | gzip -9 > backup.sql.gz
```

Which will translate to include all the relevant flags, including your database.

15.1.2 --dsn

Outputs the parameters in the DSN format, which is what many percona tools take, e.g.:

```
$ pt-duplicate-key-checker $(python manage.py dbparams --dsn)
```

Note: If you are using SSL to connect, the percona tools don't support SSL configuration being given in their DSN format; you must pass them via a MySQL configuration file instead. `dbparams` will output a warning on stderr if this is the case. For more info see the percona blog.

CHAPTER

SIXTEEN

TEST UTILITIES

The following can be imported from `django_mysql.test.utils`.

`django_mysql.test.utils.`**`override_mysql_variables`**(*using='default'*, ***options*)

Overrides MySQL system variables for a test method or for every test method in a class, similar to Django's `override_settings`. This can be useful when you're testing code that must run under multiple MySQL environments (like most of *django-mysql*). For example:

```
@override_mysql_variables(SQL_MODE="MSSQL")
class MyTests(TestCase):
    def test_it_works_in_mssql(self):
        run_it()

    @override_mysql_variables(SQL_MODE="ANSI")
    def test_it_works_in_ansi_mode(self):
        run_it()
```

During the first test, the `SQL_MODE` will be `MSSQL`, and during the second, it will be `ANSI`; each slightly changes the allowed SQL syntax, meaning they are useful to test.

Note: This only sets the system variables for the session, so if the tested code closes and re-opens the database connection the change will be reset.

`django_mysql.test.utils.`**`using`**

The connection alias to set the system variables for, defaults to 'default'.

CHAPTER

SEVENTEEN

EXCEPTIONS

Various exception classes that can be raised by `django_mysql` code. They can imported from the `django_mysql.exceptions` module.

exception `django_mysql.exceptions.`**TimeoutError**

Indicates a database operation timed out in some way.

CHAPTER

EIGHTEEN

CONTRIBUTING

18.1 Run the tests

1. Install tox.
2. Run a supported version of MySQL or MariaDB. This is easiest with the official Docker images. For example:

```
docker run --detach --name mariadb -e MYSQL_ROOT_PASSWORD=hunter2 --publish␣
↪3306:3306 mariadb:10.7
```

3. Run the tests by passing environment variables with your connection parameters. For the above Docker command:

```
DB_HOST=127.0.0.1 DB_USER=root DB_PASSWORD='hunter2' tox -e py310-django40
```

   tox environments are split per Python and Django version.

   You can run a subset of tests by passing them after `--` like:

```
DB_HOST=127.0.0.1 DB_USER=root DB_PASSWORD='hunter2' tox -e py310-django40 --␣
↪tests/testapp/test_cache.py
```

   You can also pass other pytest arguments after the `--`.

CHAPTER

NINETEEN

CHANGELOG

19.1 4.13.0 (2024-04-26)

19.2 4.12.0 (2023-10-11)

- Support Django 5.0.

19.3 4.11.0 (2023-07-10)

- Drop Python 3.7 support.

19.4 4.10.0 (2023-06-16)

- Support Python 3.12.

19.5 4.9.0 (2023-02-25)

- Support Django 4.2.
- Drop support for MySQL 5.7 and MariaDB 10.3. They will both reach EOL this year, and Django 4.2 does not support them.

19.6 4.8.0 (2022-12-06)

- Make `MySQLCache.touch()` return `True` if the key was touched, `False` otherwise. This return value was missing since the method was added for Django 2.1.
- Fix a bug where set fields' `contains` lookups would put SQL parameters in the wrong order.
- Remove deprecated database functions which exist in Django 3.0+:
  - `Sign`
  - `MD5`
  - `SHA1`

  - SHA2.

19.7 4.7.1 (2022-08-11)

- Ensure that changing `choices` on an `EnumField` triggers a migration on Django 4.1.

19.8 4.7.0 (2022-06-05)

- Support Python 3.11.
- Support Django 4.1.
- Drop support for MariaDB 10.2, as it is end of life.

19.9 4.6.0 (2022-05-10)

- Drop support for Django 2.2, 3.0, and 3.1.
- Support MariaDB 10.7 and 10.8.
- Drop `django_mysql.utils.connection_is_mariadb`. On Django 3.0+ you can simply check:

```
connection.vendor == "mysql" and connection.mysql_is_mariadb
```

- Deprecate database functions which exist in Django 3.0+:
  - `Sign`
  - `MD5`
  - `SHA1`
  - `SHA2`

19.10 4.5.0 (2022-01-23)

- Drop `pt_fingerprint()`. Its complicated threading code leaked processes. Switch to calling `pt-fingerprint` directly with `subprocess.run()` instead.
- Add model `FixedCharField` for storing fixed width strings using a `CHAR` type.

  Thanks to Caleb Ely in PR #883.

19.11 4.4.0 (2022-01-10)

- Drop Python 3.6 support.

19.12 4.3.0 (2021-12-07)

- Fix `DynamicField.deconstruct()` to correctly handle `blank`.
- Make JSON database functions work on MariaDB.

19.13 4.2.0 (2021-10-05)

- Support Python 3.10.

19.14 4.1.0 (2021-09-28)

- Support Django 4.0.

19.15 4.0.0 (2021-08-24)

- Test with MariaDB 10.6.
- Add type hints.
- Drop support for MySQL 5.6 and MariaDB 10.1, as they are both end of life.
- Drop `fix_datetime_columns` management command, which was useful when upgrading from MySQL < 5.6.
- Drop check for strict mode (`django_mysql.W001`) as this is now included in Django itself since version 1.10.
- Drop the `Greatest`, `Least`, `Abs`, `Ceiling`, `Floor` and `Round` database functions as they exist in Django core now. Swap to importing them from `django.db.models.functions`.
- Drop `JSONField` model and form fields. Django 3.1 provides a `JSONField` implementation that works with all database backends, use that instead. If you are on an earlier version of Django, use django-jsonfield-backport.
- Make JSON database functions work with Django's `JSONField`, and the backport. They remain MySQL only.
- Drop HANDLER functionality. This was not particularly robustly implemented and is somewhat dangerous to use due to its potential for dirty reads.

19.16 3.12.0 (2021-06-11)

- Fix index hints for tables with aliases.

  Thanks to Henrik Aarnio in PR #786.

- Stop distributing tests to reduce package size. Tests are not intended to be run outside of the tox setup in the repository. Repackagers can use GitHub's tarballs per tag.

19.17 3.11.1 (2021-01-26)

- Pass the `chunk_size` argument through in `QuerySetMixin.iterator()`. (Issue #752)

19.18 3.11.0 (2021-01-25)

- Support Django 3.2.
- Rework system checks for Django 3.1, which made database checks optional. To run the checks you now need to pass the `--database` argument to the `check` command, for example `python manage.py check --database default`.

19.19 3.10.0 (2020-12-09)

- Drop Python 3.5 support.
- Support Python 3.9.
- Deprecate the `Greatest`, `Least`, `Abs`, `Ceiling`, `Floor` and `Round` database functions as they exist in Django core now. Using the Django-MySQL versions now triggers a `DeprecationWarning`.
- Deprecate `JSONField`. Django 3.1 provides a `JSONField` implementation that works with all database backends, use that instead. If you are on an earlier version of Django, use django-jsonfield-backport.

19.20 3.9.0 (2020-10-11)

- Move license from BSD to MIT License.
- Fix form JSONField for to not use the `ensure_ascii` flag, making it support all unicode characters.

19.21 3.8.1 (2020-07-27)

- Fix one more RemovedInDjango40Warning message for django.utils.translation.

19.22 3.8.0 (2020-07-27)

- Drop Django 2.0 and 2.1 support.
- Test with MariaDB 10.5.
- Drop testing with MariaDB 10.0 (Django only officially supports MariaDB 10.1+ anyway).
- Fix RemovedInDjango40Warning messages for django.utils.translation.

19.23 3.7.1 (2020-06-24)

- Fix query rewriting to install for recreated database connections. (Issue #677)

19.24 3.7.0 (2020-06-15)

- Add Django 3.1 support.

19.25 3.6.0 (2020-06-09)

- Changed query rewriting to use Django's database instrumentation. (Issue #644)
- Added `JSONIn` lookup which only works with literal values (not with expressions nor subqueries).
- Fix `JSONContains` to make it work with scalar values again. (PR #668).

19.26 3.5.0 (2020-05-04)

- Add MySQL 8 support.

19.27 3.4.0 (2020-04-16)

- Prevent `collections.abc.Sequence` warning.
- Drop Django 1.11 support. Only Django 2.0+ is supported now.
- Prevent `JSONField` from adding `CAST(... AS JSON)` for `str`, `int`, and `float` objects.

19.28 3.3.0 (2019-12-10)

- Update Python support to 3.5-3.8.
- Converted setuptools metadata to configuration file. This meant removing the `__version__` attribute from the package. If you want to inspect the installed version, use `importlib.metadata.version("django-mysql")` (docs / backport).
- Fix `GroupConcat` to work with both `separator` and `ordering` set. (PR #596).

19.29 3.2.0 (2019-06-14)

- Update Python support to 3.5-3.7, as 3.4 has reached its end of life.
- Always cast SQL params to tuples in ORM code.

19.30 3.1.0 (2019-05-17)

- Remove authors file and documentation page. This was showing only 4 out of the 17 total contributors.
- Tested on Django 2.2. No changes were needed for compatibility.

19.31 3.0.0.post1 (2019-03-05)

- Remove universal wheel. Version 3.0.0 has been pulled from PyPI after being up for 3 hours to fix mistaken installs on Python 2.

19.32 3.0.0 (2019-03-05)

- Drop Python 2 support, only Python 3.4+ is supported now.

19.33 2.5.0 (2019-03-03)

- Drop Django 1.8, 1.9, and 1.10 support. Only Django 1.11+ is supported now.

19.34 2.4.1 (2018-08-18)

- Django 2.1 compatibility - no code changes were required, releasing for PyPI trove classifiers and documentation.

19.35 2.4.0 (2018-07-31)

- Added `JSONArrayAppend` database function that wraps the respective JSON-modifying function from MySQL 5.7.

19.36 2.3.1 (2018-07-22)

- Made `EnumField` escape its arguments in a `pymysql`-friendly fashion.

19.37 2.3.0 (2018-06-19)

- Started testing with MariaDB 10.3.
- Changed `GlobalStatus.wait_until_load_low()` to increase the default number of allowed running threads from 5 to 10, to account for the new default threads in MariaDB 10.3.
- Added `encoder` and `decoder` arguments to `JSONField` for customizing the way JSON is encoded and decoded from the database.
- Added a `touch` method to the `MySQLCache` to refresh cache keys, as added in Django 2.1.
- Use a temporary database connection in system checks to avoid application startup stalls.

19.38 2.2.2 (2018-04-24)

- Fixed some crashes from `DynamicField` instances without explicit `spec` definitions.
- Fixed a crash in system checks for `ListCharField` and `SetCharField` instances missing `max_length`.

19.39 2.2.1 (2018-04-14)

- Fixed `JSONField.deconstruct()` to not break the path for subclasses.

19.40 2.2.0 (2017-12-04)

- Add `output_field` argument to `JSONExtract` function.
- Improved DB version checks for `JSONField` and `DynamicField` so you can have just one connection that supports them.
- Django 2.0 compatibility.

19.41 2.1.1 (2017-10-10)

- Changed subprocess imports for compatibility with Google App Engine.
- (Insert new release notes below this line)
- Made `MySQLCache.set_many` return a list as per Django 2.0.

19.42 2.1.0 (2017-06-11)

- Django 1.11 compatibility
- Some fixes to work with new versions of `mysqlclient`

19.43 2.0.0 (2017-05-28)

- Fixed `JSONField` model field string serialization. This is a small backwards incompatible change.

  Storing strings mostly used to crash with MySQL error -1 "error totally whack", but in the case your string was valid JSON, it would store it as a JSON object at the MySQL layer and deserialize it when returned. For example you could do this:

```
>>> mymodel.attrs = '{"foo": "bar"}'
>>> mymodel.save()
>>> mymodel = MyModel.objects.get(id=mymodel.id)
>>> mymodel.attrs
{'foo': 'bar'}
```

  The new behaviour now correctly returns what you put in:

```
>>> mymodel.attrs
'{"foo": "bar"}'
```

- Removed the `connection.is_mariadb` monkey patch. This is a small backwards incompatible change. Instead of using it, use `django_mysql.utils.connection_is_mariadb`.

19.44 1.2.0 (2017-05-14)

- Only use Django's vendored six (`django.utils.six`). Fixes usage of `EnumField` and field lookups when `six` is not installed as a standalone package.
- Added `JSONInsert`, `JSONReplace` and `JSONSet` database functions that wraps the respective JSON-modifying functions from MySQL 5.7.
- Fixed `JSONField` to work with Django's serializer framework, as used in e.g. `dumpdata`.
- Fixed `JSONField` form field so that it doesn't overquote inputs when redisplaying the form due to invalid user input.

19.45 1.1.1 (2017-03-28)

- Don't allow NaN in `JSONField` because MySQL doesn't support it

19.46 1.1.0 (2016-07-22)

- Dropped Django 1.7 support
- Made the query hint functions raise `RuntimeError` if you haven't activated the query-rewriting layer in settings.

19.47 1.0.9 (2016-05-12)

- Fixed some features to work when there are non-MySQL databases configured
- Fixed `JSONField` to allow control characters, which MySQL does - but not in a top-level string, only inside a JSON object/array.

19.48 1.0.8 (2016-04-08)

- `SmartChunkedIterator` now fails properly for models whose primary key is a non-integer foreign key.
- `pty` is no longer imported at the top-level in `django_mysql.utils`, fixing Windows compatibility.

19.49 1.0.7 (2016-03-04)

- Added new `JSONField` class backed by the JSON type added in MySQL 5.7.
- Added database functions `JSONExtract`, `JSONKeys`, and `JSONLength` that wrap the JSON functions added in MySQL 5.7, which can be used with the JSON type columns as well as JSON data held in text/varchar columns.
- Added `If` database function for simple conditionals.

19.50 1.0.6 (2016-02-26)

- Now MySQL 5.7 compatible
- The final message from `SmartChunkedIterator` is now rounded to the nearest second.
- `Lock` and `TableLock` classes now have `acquire` and `release()` methods for using them as normal objects rather than context managers

19.51 1.0.5 (2016-02-10)

- Added `manage.py` command `fix_datetime_columns` that outputs the SQL necessary to fix any `datetime` columns into `datetime(6)`, as required when upgrading a database to MySQL 5.6+, or MariaDB 5.3+.
- `SmartChunkedIterator` output now includes the total time taken and number of objects iterated over in the final message.

19.52 1.0.4 (2016-02-02)

- Fixed the new system checks to actually work

19.53 1.0.3 (2016-02-02)

- Fixed `EnumField` so that it works properly with forms, and does not accept the `max_length` argument.
- `SmartChunkedIterator` output has been fixed for reversed iteration, and now includes a time estimate.
- Added three system checks that give warnings if the MySQL configuration can (probably) be improved.

19.54 1.0.2 (2016-01-24)

- New function `add_QuerySetMixin` allows addding the `QuerySetMixin` to arbitrary `QuerySet`s, for when you can't edit a model class.
- Added field class `EnumField` that uses MySQL's `ENUM` data type.

19.55 1.0.1 (2015-11-18)

- Added `chunk_min` argument to `SmartChunkedIterator`

19.56 1.0.0 (2015-10-29)

- Changed version number to 1.0.0 to indicate maturity.
- Added `DynamicField` for using MariaDB's Named Dynamic Columns, and related database functions `ColumnAdd`, `ColumnDelete`, and `ColumnGet`.
- `SmartChunkedIterator` with `report_progress=True` correctly reports 'lowest pk so far' when iterating in reverse.
- Fix broken import paths during `deconstruct()` for subclasses of all fields: `ListCharField`, `ListTextField`, `SetCharField`, `SetTextField`, `SizedBinaryField` and `SizedTextField`
- Added XML database functions - `UpdateXML` and `XMLExtractValue`.

19.57 0.2.3 (2015-10-12)

- Allow `approx_count` on QuerySets for which only query hints have been used
- Added index query hints to `QuerySet` methods, via query-rewriting layer
- Added `ordering` parameter to `GroupConcat` to specify the `ORDER BY` clause
- Added index query hints to `QuerySet` methods, via query-rewriting layer
- Added `sql_calc_found_rows()` query hint that calculates the total rows that match when you only take a slice, which becomes available on the `found_rows` attribute
- Made `SmartChunkedIterator` work with `reverse()`'d `QuerySets`

19.58 0.2.2 (2015-09-03)

- `SmartChunkedIterator` now takes an argument `chunk_size` as the initial chunk size
- `SmartChunkedIterator` now allows models whose primary key is a `ForeignKey`
- Added `iter_smart_pk_ranges` which is similar to `iter_smart_chunks` but yields only the start and end primary keys for each chunks, in a tuple.
- Added prefix methods to `MySQLCache` - `delete_with_prefix`, `get_with_prefix`, `keys_with_prefix`
- Added `Bit1BooleanField` and `NullBit1BooleanField` model fields that work with boolean fields built by other databases that use the `BIT(1)` column type

19.59 0.2.1 (2015-06-22)

- Added Regexp database functions for MariaDB - `RegexpInstr`, `RegexpReplace`, and `RegexpSubstr`
- Added the option to not limit the size of a `MySQLCache` by setting `MAX_ENTRIES` = -1.
- `MySQLCache` performance improvements in *get*, *get_many*, and *has_key*
- Added query-rewriting layer added which allows the use of MySQL query hints such as `STRAIGHT_JOIN` via QuerySet methods, as well as adding label comments to track where queries are generated.
- Added `TableLock` context manager

19.60 0.2.0 (2015-05-14)

- More database functions added - `Field` and its complement `ELT`, and `LastInsertId`
- Case sensitive string lookup added as to the ORM for `CharField` and `TextField`
- Migration operations added - `InstallPlugin`, `InstallSOName`, and `AlterStorageEngine`
- Extra ORM aggregates added - `BitAnd`, `BitOr`, and `BitXor`
- `MySQLCache` is now case-sensitive. If you are already using it, an upgrade `ALTER TABLE` and migration is provided at the end of the cache docs.
- (MariaDB only) The `Lock` class gained a class method `held_with_prefix` to query held locks matching a given prefix

- `SmartIterator` bugfix for chunks with 0 objects slowing iteration; they such chunks most often occur on tables with primary key "holes"
- Now tested against Django master for cutting edge users and forwards compatibility

19.61 0.1.10 (2015-04-30)

- Added the `MySQLCache` backend for use with Django's caching framework, a more efficient version of `DatabaseCache`
- Fix a `ZeroDivision` error in `WeightedAverageRate`, which is used in smart iteration

19.62 0.1.9 (2015-04-20)

- `pt_visual_explain` no longer executes the given query before fetching its `EXPLAIN`
- New `pt_fingerprint` function that wraps the `pt-fingerprint` tool efficiently
- For `List` fields, the new `ListF` class allows you to do atomic append or pop operations from either end of the list in a single query
- For `Set` fields, the new `SetF` class allows you to do atomic add or remove operatiosn from the set in a single query
- The `@override_mysql_variables` decorator has been introduced which makes testing code with different MySQL configurations easy
- The `is_mariadb` property gets added onto Django's MySQL `connection` class automatically
- A race condition in determining the minimum and maximum primary key values for smart iteration was fixed.

19.63 0.1.8 (2015-03-31)

- Add `Set` and `List` fields which can store comma-separated sets and lists of a base field with MySQL-specific lookups
- Support MySQL's `GROUP_CONCAT` as an aggregate!
- Add a `functions` module with many MySQL-specific functions for the new Django 1.8 database functions feature
- Allow access of the global and session status for the default connection from a lazy singleton, similar to Django's `connection` object
- Fix a different recursion error on `count_tries_approx`

19.64 0.1.7 (2015-03-25)

- Renamed `connection_name` argument to `using` on `Lock`, `GlobalStatus`, and `SessionStatus` classes, for more consistency with Django.
- Fix recursion error on `QuerySetMixin` when using `count_tries_approx`

19.65 0.1.6 (2015-03-21)

- Added support for `HANDLER` statements as a `QuerySet` extension
- Now tested on Django 1.8
- Add `pk_range` argument for 'smart iteration' code

19.66 0.1.5 (2015-03-11)

- Added `manage.py` command `dbparams` for outputting database paramters in formats useful for shell scripts

19.67 0.1.4 (2015-03-10)

- Fix release process

19.68 0.1.3 (2015-03-08)

- Added `pt_visual_explain` integration on `QuerySet`
- Added soundex-based field lookups for the ORM

19.69 0.1.2 (2015-03-01)

- Added `get_many` to `GlobalStatus`
- Added `wait_until_load_low` to `GlobalStatus` which allows you to wait for any high load on your database server to dissipate.
- Added smart iteration classes and methods for `QuerySet`s that allow efficient iteration over very large sets of objects slice-by-slice.

19.70 0.1.1 (2015-02-23)

- Added `Model` and `QuerySet` subclasses which add the `approx_count` method

19.71 0.1.0 (2015-02-12)

- First release on PyPI
- `Locks`
- `GlobalStatus` and `SessionStatus`

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

T

U

W

X