django-postgres-metrics

Release 0.15.1.dev13+g3924d2d

Markus Holtermann and others

Oct 10, 2023

CONTENTS:

CHAPTER

ONE

BACKGROUND. WHY? WHAT?

The original idea for this library comes from Craig Kerstiens's talk "Postgres at any scale" at PyCon Canada 2017. In his talk, Craig pointed out a bunch of metrics one should look at to understand why a PostgreSQL database could be "slow" or not perform as expected.

In many cases, the root cause for "PostgreSQL being slow" is the lack of memory, full table scans, and others. For example, a cache hit ratio of less than 99% means that each of cache miss queries needs to go to the filesystem. Filesystem access is significantly slower than RAM access. Giving the PostgreSQL instance more memory will very likely improve performance.

This project aims at showing these and other metrics in the Django Admin making such bottlenecks visible.

CHAPTER

TWO

GETTING STARTED

2.1 Installation

Start by installing `django-postgres-metrics` from PyPI:

```
$ pip install django-postgres-metrics
```

You will also need to make sure to have `psycopg2` or `psycopg2-binary` installed which is already a requirement by Django for PostgreSQL support anyway.

Then you need to add `postgres_metrics` to your `INSTALLED_APPS` list. Due to the way postgres_metrics works, you need to include it *before* the `admin` app:

```
INSTALLED_APPS = [
    'postgres_metrics.apps.PostgresMetrics',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

You also need to make sure that the `request` context processor is included in the `TEMPLATES` setting. It is included by default for projects that were started on Django 1.8 or later:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'OPTIONS': {
            'context_processors': [
                ...,
                'django.template.context_processors.request',
                ...,
            ],
        },
    },
]
```

Lastly, you need to add a URL path to your global `urls.py` *before* the `admin` URL patterns.

```
from django.urls import include, path
urlpatterns = [
    path('admin/postgres-metrics/', include('postgres_metrics.urls')),
    path('admin/', admin.site.urls),
]
```

Congratulations, you made it!

Next, see the *Usage* section for the *Django Admin Integration* or the *Command Line Interface*.

CHAPTER

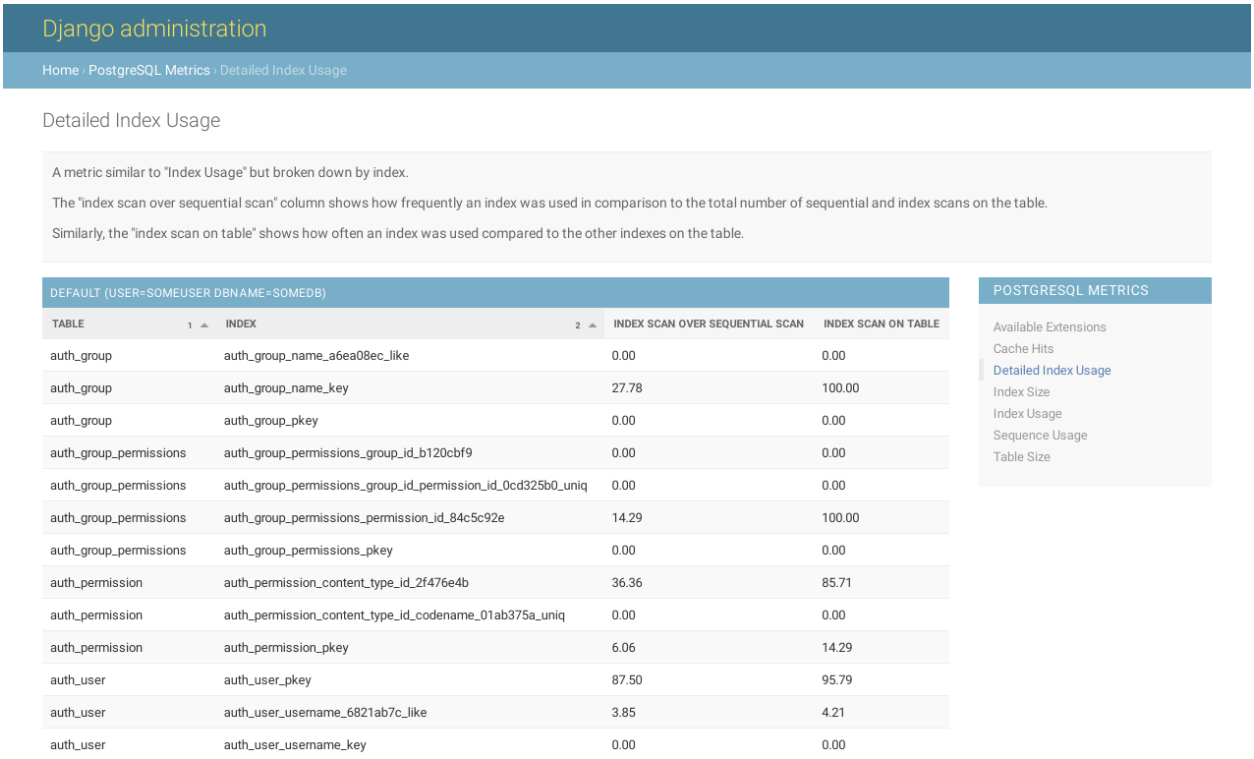
THREE

USING DJANGO-POSTGRES-METRICS

3.1 Django Admin Integration

When you now browse to the Django Admin with superuser permissions, you'll see a "PostgreSQL Metrics" section at the bottom left, listing all available metrics.

This is what a metric could look like:



3.2 Command Line Interface

While the Django Admin is often installed and used by many, there are numerous projects that do not use it. For them, django-postgres-metrics 0.13 brings a few management commands that provide the same information.

3.2.1 `pgm_list_metrics`

This command lists all available metrics.

3.2.2 `pgm_show_metric`

This command shows the metric's data. The command expects the `slug` from the `pgm_list_metrics` command output as the first argument.

CHAPTER

FOUR

DESIGNING YOUR OWN METRIC

First, you need to import a base class that all metrics inherit from. This will take care of fetching the data from the database, implement sorting if you want, and provide per record and per item styling. Furthermore, you need to make your metric know to django-postgres_metrics-metrics:

```
from django.utils.translation import ugettext_lazy as _

from postgres_metrics.metrics import Metric, registry


class DjangoMigrationStatistics(Metric):
    """
    Count the number of applied Django migrations per app and sort by
    descending count and ascending app name.
    """
    label = _('Migration Statistics')
    slug = 'django-migration-statistics'
    ordering = '-2.1'
    sql = '''
        SELECT
            app, count(*)
        FROM
            django_migrations
        GROUP BY
            app
        {ORDER_BY}
        ;
    '''


registry.register(DjangoMigrationStatistics)
```

A short explanation of what the metric class attributes do:

docstring
: This will be shown above the metric's output and allow you to give an explanation what you see.

label
: This is what you see on the Django Admin index page as a metric name.

slug
: A unique identifier for a metric. This will be used as part of the URL.

ordering

The default ordering you want to use in the metric. Use column indexes (one-indexed) and prefix with - for descending sorting.

sql

The actual SQL you want to run. The `{ORDER_BY}` part is replaced with `ORDER BY 2 DESC, 1` in the example.

4.1 Styling Metric Output

4.1.1 Styling Records

If you want to highlight some of the output rows you can define a *get_record_style* method on a metric class:

```
class MyMetric(Metric):
    ...

    def get_record_style(self, record):
        if record[0]:
            if record[1] > 1000:
                return 'critical'
            if record[1] > 100:
                return 'warning'
            if record[1] == 0:
                return 'info'
            return 'ok'
```

The `record` parameter is a tuple with all values of a single row in the output. This method will be called for every single row being outputted. Don't do any expensive calculations here!

django-postgres-metrics ships four pre-defined styles: `ok`, `warning`, `critical` and `info` that you can return.

4.1.2 Styling Record Items

Similarly, you can highlight a single value in the metric output by using the *get_record_item_style* method on a metric class:

```
class MyMetric(Metric):
    ...

    def get_record_item_style(self, record, itme, index):
        if index == 2 and record[1]:
            if item > 1000:
                return 'critical'
            if item > 100:
                return 'warning'
            if item == 0:
                return 'info'
            return 'ok'
```

Along with the `record` you get the current value or `item` and the (zero- indexed) position of the item in the record. The item is provided for convenience and is defined as `item = record[index]`.

CHAPTER

FIVE

CONTRIBUTING

5.1 Issuing a new Release

- Update the changelog (`docs/source/changelog.rst`) with the new desired version.
- Tag the new release and push it:

```
$ git tag -s "x.y.z"
$ git push --tags origin main
```

- Update the release nodes on GitHub for the newly pushed release.
- Update the versions in ReadTheDocs.

CHAPTER

SIX

REFERENCE

6.1 postgres_metrics package

6.1.1 Subpackages

postgres_metrics.management package

Subpackages

postgres_metrics.management.commands package

Submodules

postgres_metrics.management.commands.pgm_list_metrics module

class postgres_metrics.management.commands.pgm_list_metrics.**Command**(*stdout: TextIO | None = None*, *stderr: TextIO | None = None*, *no_color: bool = False*, *force_color: bool = False*)

Bases: RichCommand

handle(**args*, ***options*)

The actual logic of the command. Subclasses must implement this method.

help = 'List all available metrics.'

postgres_metrics.management.commands.pgm_show_metric module

class postgres_metrics.management.commands.pgm_show_metric.**Command**(*stdout: TextIO | None = None*, *stderr: TextIO | None = None*, *no_color: bool = False*, *force_color: bool = False*)

Bases: RichCommand

add_arguments(*parser*)

Entry point for subclassed commands to add custom arguments.

handle(**args*, ***options*)

The actual logic of the command. Subclasses must implement this method.

help = 'Show the selected metric.'

Module contents

Module contents

postgres_metrics.templatetags package

Submodules

postgres_metrics.templatetags.postgres_metrics module

postgres_metrics.templatetags.postgres_metrics.**get_postgres_metrics**(*context*)

Return an iterable over all registered metrics, sorted by their label.

The template tag will filter out all metrics the current user does not have access to.

See *MetricRegistry.sorted* for details.

postgres_metrics.templatetags.postgres_metrics.**record_item_style**(*context*)

Return the style to be usd for the metric's current record specific item.

Like the *record_style()* template tag, this expects a template context variable `'metric'` referring to the metric instance in question, as well as a context variable `'record'` containing the current record. Furthermore, the `'forloop.counter0'` context is expected with the current column index to be styled. This mean, for styling, unline on the SQL column numbering, columns are zero-indexed.

This template tag calls into *Metric.get_record_item_style* and will—if a return value was specified—prefix that one with `'pgm-'`.

postgres_metrics.templatetags.postgres_metrics.**record_style**(*context*)

Return the style to be usd for the metric's current record.

This expects a template context variable `'metric'` referring to the metric instance in question, as well as a context variable `'record'` containing the current record to be used for style determination.

This template tag calls into *Metric.get_record_style* and will—if a return value was specified—prefix that one with `'pgm-'`.

Module contents

6.1.2 Submodules

postgres_metrics.apps module

class postgres_metrics.apps.**PostgresMetrics**(*app_name*, *app_module*)

Bases: AppConfig

default_auto_field = 'django.db.models.AutoField'

name = 'postgres_metrics'

verbose_name = 'PostgreSQL Metrics'

postgres_metrics.metrics module

class postgres_metrics.metrics.**AvailableExtensions**(*ordering=None*)

Bases: *Metric*

PostgreSQL can be extended by installing extensions with the CREATE EXTENSION command. The list of available extensions on each database is shown below.

description = ' PostgreSQL can be extended by installing extensions with the CREATE EXTENSION command. The list of available extensions on each database is shown below. '

get_record_style(*record*)

Given a single record from *MetricResult*, decide how to style it. Most likely to be used with the template tag *record_style()*.

By default, django-postgres-metrics supports for styling classes:

- ok
- warning
- critical
- info

Override this method and return one of the above strings or None to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

label = 'Available Extensions'

The label is what is used in the Django Admin views. Consider making this string translateable.

ordering = '1'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on parsed_ordering.

permission_key = 'postgres_metrics.can_view_metric_available_extensions'

permission_name = 'can_view_metric_available_extensions'

slug = 'available-extensions'

A URL safe representation of the label and unique across all metrics.

sql = '\n SELECT\n name,\n default_version,\n installed_version,\n comment\n FROM\n pg_available_extensions\n {ORDER_BY}\n ;\n '

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string {ORDER_BY} in the query as necessary. For details on that value see get_order_by_clause().

class postgres_metrics.metrics.**CacheHits**(*ordering=None*)

Bases: *Metric*

The typical rule for most applications is that only a fraction of its data is regularly accessed. As with many other things data can tend to follow the 80/20 rule with 20% of your data accounting for 80% of the reads and often

times its higher than this. Postgres itself actually tracks access patterns of your data and will on its own keep frequently accessed data in cache. Generally you want your database to have a cache hit rate of about 99%.

(Source: http://www.craigkerstiens.com/2012/10/01/understanding-postgres-performance/)

description = ' The typical rule for most applications is that only a fraction of its data is regularly accessed. As with many other things data can tend to follow the 80/20 rule with 20% of your data accounting for 80% of the reads and often times its higher than this. Postgres itself actually tracks access patterns of your data and will on its own keep frequently accessed data in cache. Generally you want your database to have a cache hit rate of about 99%. \n\n (Source: http://www.craigkerstiens.com/2012/10/01/understanding-postgres-performance/) '

get_record_item_style(*record*, *item*, *index*)

Given a single record from *MetricResult*, the value of the current item within, and the current item's index, decide how to style it. Most likely to be used with the template tag *record_item_style()*.

By default, django-postgres-metrics supports for styling classes:

- ok
- warning
- critical
- info

Override this method and return one of the above strings or None to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

header_labels = ['Reads', 'Hits', 'Ratio']

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Cache Hits'

The label is what is used in the Django Admin views. Consider making this string translateable.

permission_key = 'postgres_metrics.can_view_metric_cache_hits'

permission_name = 'can_view_metric_cache_hits'

slug = 'cache-hits'

A URL safe representation of the label and unique across all metrics.

sql = "\n WITH cache AS (\n SELECT\n sum(heap_blks_read) heap_read,\n sum(heap_blks_hit) heap_hit,\n sum(heap_blks_hit) + sum(heap_blks_read) heap_sum\n FROM\n pg_statio_user_tables\n) SELECT\n heap_read,\n heap_hit,\n CASE\n WHEN heap_sum = 0 THEN 'N/A'\n ELSE (heap_hit / heap_sum)::text\n END ratio\n FROM\n cache\n {ORDER_BY}\n ;\n "

The actual SQL statement that is being used to query the database. In order to make use of the ordering, include the string {ORDER_BY} in the query as necessary. For details on that value see get_order_by_clause().

class postgres_metrics.metrics.**DetailedIndexUsage**(*ordering=None*)

Bases: *Metric*

A metric similar to "Index Usage" but broken down by index.

The "index scan over sequential scan" column shows how frequently an index was used in comparison to the total number of sequential and index scans on the table.

Similarly, the "index scan on table" shows how often an index was used compared to the other indexes on the table.

description = ' A metric similar to "Index Usage" but broken down by index. \n\n The "index scan over sequential scan" column shows how frequently an index was used in comparison to the total number of sequential and index scans on the table. \n\n Similarly, the "index scan on table" shows how often an index was used compared to the other indexes on the table. '

header_labels = ['Table', 'Index', 'Index Scan over Sequential Scan', 'Index Scan on table']

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Detailed Index Usage'

The label is what is used in the Django Admin views. Consider making this string translateable.

ordering = '1.2'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on `parsed_ordering`.

permission_key = 'postgres_metrics.can_view_metric_detailed_index_usage'

permission_name = 'can_view_metric_detailed_index_usage'

slug = 'detailed-index-usage'

A URL safe representation of the label and unique across all metrics.

sql = '\n SELECT\n t.relname,\n i.indexrelname,\n CASE t.seq_scan + t.idx_scan\n WHEN 0\n THEN round(0.0, 2)\n ELSE\n round(\n (100::float * i.idx_scan / (t.seq_scan + t.idx_scan))::numeric,\n 2::int\n)\n END,\n CASE t.idx_scan\n WHEN 0\n THEN round(0.0, 2)\n ELSE\n round((100::float * i.idx_scan / t.idx_scan)::numeric, 2::int)\n END\n FROM\n pg_stat_user_tables t\n INNER JOIN\n pg_stat_user_indexes i\n ON t.relid = i.relid\n {ORDER_BY}\n ;\n '

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see `get_order_by_clause()`.

class postgres_metrics.metrics.**IndexSize**(*ordering=None*)

Bases: *Metric*

description = ''

header_labels = ['Table', 'Index', 'Size']

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Index Size'

The label is what is used in the Django Admin views. Consider making this string translateable.

ordering = '1.2'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on `parsed_ordering`.

permission_key = 'postgres_metrics.can_view_metric_index_size'

permission_name = 'can_view_metric_index_size'

slug = 'index-size'

A URL safe representation of the label and unique across all metrics.

sql = '\n SELECT\n relname,\n indexrelname,\n pg_size_pretty(index_size)\n FROM (\n SELECT\n relname,\n indexrelname,\n pg_relation_size(indexrelid) AS index_size\n FROM\n pg_stat_user_indexes\n {ORDER_BY}\n) AS t\n ;\n '

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see `get_order_by_clause()`.

class postgres_metrics.metrics.**IndexUsage**(*ordering=None*)

Bases: *Metric*

While there is no perfect answer, if you're not somewhere around 99% on any table over 10,000 rows you may want to consider adding an index. When examining where to add an index you should look at what kind of queries you're running. Generally you'll want to add indexes where you're looking up by some other id or on values that you're commonly filtering on such as created_at fields.

(Source: http://www.craigkerstiens.com/2012/10/01/understanding-postgres-performance/)

description = ' While there is no perfect answer, if you're not somewhere around 99% on any table over 10,000 rows you may want to consider adding an index. When examining where to add an index you should look at what kind of queries you're running. Generally you'll want to add indexes where you're looking up by some other id or on values that you're commonly filtering on such as created_at fields. \n\n (Source: http://www.craigkerstiens.com/2012/10/01/understanding-postgres-performance/) '

get_record_style(*record*)

Given a single record from *MetricResult*, decide how to style it. Most likely to be used with the template tag *record_style()*.

By default, django-postgres-metrics supports for styling classes:

- `ok`
- `warning`
- `critical`
- `info`

Override this method and return one of the above strings or `None` to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

header_labels = ['Table', 'Index used (in %)', 'Num rows']

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Index Usage'

The label is what is used in the Django Admin views. Consider making this string translateable.

ordering = '2'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on `parsed_ordering`.

permission_key = 'postgres_metrics.can_view_metric_index_usage'

permission_name = 'can_view_metric_index_usage'

slug = 'index-usage'

A URL safe representation of the label and unique across all metrics.

sql = '\n SELECT\n relname,\n round(\n (100::float * idx_scan / (seq_scan + idx_scan))::numeric,\n 2::int\n),\n n_live_tup\n FROM\n pg_stat_user_tables\n WHERE\n seq_scan + idx_scan > 0\n {ORDER_BY}\n ;\n '

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see `get_order_by_clause()`.

class postgres_metrics.metrics.**Metric**(*ordering=None*)

Bases: `object`

The superclass for all Metric implementations. Inherit from this to define your own metric.

If you want to implement your own metric, here's a gist:

```
from django.utils.translation import ugettext_lazy as _
from postgres_metrics.metrics import Metric, registry


class DjangoMigrationStatistics(Metric):
    """
    Count the number of applied Django migrations per app and sort by
    descending count and ascending app name.
    """
    label = _('Migration Statistics')
    slug = 'django-migration-statistics'
    ordering = '-2.1'
    sql = '''
        SELECT
            app, count(*)
        FROM
            django_migrations
        GROUP BY
            app
        {ORDER_BY}
        ;
    '''


registry.register(DjangoMigrationStatistics)
```

description

Don't define this value directly. Instead define a docstring on the metric class.

The docstring will be processed by Python internals to trim leading white spaces and fix newlines. `'\r\n'` and `'\r'` line breaks will be normalized to `'\n'`. Two or more consecutive occurances of `'\n'` mark a paragraph which will be escaped and wrapped in HTML tags. Further, each paragraph will call into Django's `urlize()` method to create <a> HTML tags around links.

classmethod can_view(*user*)

Check that the given a user instance has access to the metric.

This requires the user instance to have the django.contrib.auth.models.User.is_superuser and django.contrib.auth.models.User.is_staff flags as well as the django.contrib.auth.models.User.has_perm() method.

Users with is_superuser=True will always have access to a metric. Users with is_staff=True will have access if and only if the user has the permission for a metric.

full_sql

The *sql* formatted with *get_order_by_clause()*.

get_data()

Iterate over all configured PostgreSQL database and execute the *full_sql* there.

Returns
Returns a list of *MetricResult* instances.

Return type
list

get_order_by_clause()

Turn an ordering string like 1.5.-3.-2.6 into the respective SQL.

SQL's column numbering starts at 1, so do we here. Given *ordering* as 1.5.-3.-2.6 return a string ORDER BY 1 ASC, 5 ASC, 3 DESC, 2 DESC, 6 ASC.

Ensures that each column (excluding the - prefix) is an integer by calling int() on it.

get_record_item_style(*record*, *item*, *index*)

Given a single record from *MetricResult*, the value of the current item within, and the current item's index, decide how to style it. Most likely to be used with the template tag *record_item_style()*.

By default, django-postgres-metrics supports for styling classes:

- ok
- warning
- critical
- info

Override this method and return one of the above strings or None to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

get_record_style(*record*)

Given a single record from *MetricResult*, decide how to style it. Most likely to be used with the template tag *record_style()*.

By default, django-postgres-metrics supports for styling classes:

- ok
- warning
- critical
- info

Override this method and return one of the above strings or None to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

header_labels = None

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

headers

A wrapper around the *header_labels* to make the tables in the admin sortable.

label = ''

The label is what is used in the Django Admin views. Consider making this string translateable.

max_pg_version = None

min_pg_version = None

ordering = ''

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on *parsed_ordering*.

parsed_ordering

Turn an ordering string like `1.5.-3.-2.6` into the respective abstraction.

Given *ordering* as `1.5.-3.-2.6` return a list of 2-tuples like `[('', 1), ('', 5), ('-', 3), ('-', 2), ('', 6)]`.

slug = ''

A URL safe representation of the label and unique across all metrics.

sql = ''

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see *get_order_by_clause()*.

class postgres_metrics.metrics.**MetricHeader**(*name*, *index*, *ordering*)

Bases: `object`

A single column header; mostly takes care of a column's sorting status.

ascending

`True` if the column is in ascending order `False` otherwise

static join_ordering(*ordering*)

sort_priority

The priority of the columns order. 1 (high), n(low). Default 0.

url_primary

Querystring value making this the primary sorting header.

url_remove

Querystring value removing this column from sorting.

url_toggle

Querystring value toggling ascending/descending for this header.

class postgres_metrics.metrics.**MetricMeta**(*name*, *bases*, *attrs*)

Bases: `type`

class postgres_metrics.metrics.**MetricRegistry**

Bases: `object`

register(*metric*)

Given a class (not class instance) of *Metric*, adds it to the available list of metrics to show it to a user.

property sorted

All registered metrics ordered by their label.

unregister(*slug*)

Remove the metric `slug` from the registry. Raises a `KeyError` if the metric isn't registered.

class postgres_metrics.metrics.**MetricResult**(*connection*, *records*)

Bases: `object`

Hold a metric's data for a single database.

alias

The alias under which a database connection is known to Django.

dsn

The PostgreSQL connection string per psycopg2 or psycopg.

records

The rows returned by a metric for the given database.

holds_data = True

class postgres_metrics.metrics.**NoMetricResult**(*connection*, *reason*)

Bases: *MetricResult*

Internal class to pass an error message to the template when a metric is not available.

holds_data = False

class postgres_metrics.metrics.**SequenceUsage**(*ordering=None*)

Bases: *Metric*

Show the sequence usage within a PostgreSQL database. A usage over 75% will be marked as red, and a usage over 50% will be marked as yellow.

description = ' Show the sequence usage within a PostgreSQL database. A usage over 75% will be marked as red, and a usage over 50% will be marked as yellow. '

get_record_style(*record*)

Given a single record from *MetricResult*, decide how to style it. Most likely to be used with the template tag *record_style()*.

By default, django-postgres-metrics supports for styling classes:

- `ok`
- `warning`
- `critical`
- `info`

Override this method and return one of the above strings or `None` to apply the given style to the entire record. In the Django Admin this will highlight the entire row.

header_labels = ['Table', 'Column', 'Sequence', 'Last value', 'Max value', 'Used (in %)']

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Sequence Usage'

The label is what is used in the Django Admin views. Consider making this string translateable.

min_pg_version = 100000

ordering = '-6.1.2.3'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on `parsed_ordering`.

permission_key = 'postgres_metrics.can_view_metric_sequence_usage'

permission_name = 'can_view_metric_sequence_usage'

slug = 'sequence-usage'

A URL safe representation of the label and unique across all metrics.

sql = "\n SELECT\n tabcls.relname,\n attrib.attname,\n seqcls.relname,\n seq.last_value,\n seq.max_value,\n round(\n (\n 100::float * COALESCE(seq.last_value, 0)\n / (seq.max_value - seq.start_value + 1)\n)::numeric,\n 2::int\n)\n FROM\n pg_class AS seqcls\n INNER JOIN\n pg_sequences AS seq\n ON seqcls.relname = seq.sequencename\n INNER JOIN\n pg_depend AS dep\n ON seqcls.relfilenode = dep.objid\n INNER JOIN\n pg_class AS tabcls\n ON dep.refobjid = tabcls.relfilenode\n INNER JOIN\n pg_attribute AS attrib\n ON\n attrib.attnum = dep.refobjsubid\n AND attrib.attrelid = dep.refobjid\n WHERE\n seqcls.relkind = 'S'\n {ORDER_BY}\n ;\n "

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see `get_order_by_clause()`.

class postgres_metrics.metrics.**TableSize**(*ordering=None*)

Bases: *Metric*

The "size" of a table in PostgreSQL can be different, depending on what to include when calculating "the size".

The "Total size" for a relation is equal to the "Table size" plus all indexes.

The "size of * fork" refers to the "main" data fork, the Free Space Map (fsm), Visibility Map (vm), and the initialization fork.

See also the PostgreSQL documentation on the physical storage: https://www.postgresql.org/docs/current/storage.html

description = ' The "size" of a table in PostgreSQL can be different, depending on what to include when calculating "the size". \n\n The "Total size" for a relation is equal to the "Table size" plus all indexes. \n\n The "size of * fork" refers to the "main" data fork, the Free Space Map (fsm), Visibility Map (vm), and the initialization fork. \n\n See also the PostgreSQL documentation on the physical storage: https://www.postgresql.org/docs/current/storage.html '

header_labels = ['Table', 'Total size', 'Table size', "Size of 'main' fork", "Size of 'fsm' fork", "Size of 'vm' fork", "Size of 'init' fork"]

A list of strings used as column headers in the admin. Consider making the strings translateable. If the attribute is undefined, the column names returned by the database will be used.

label = 'Table Size'

The label is what is used in the Django Admin views. Consider making this string translateable.

ordering = '1'

The default ordering that should be applied to the SQL query by default. This needs to be a valid ordering string as defined on `parsed_ordering`.

permission_key = 'postgres_metrics.can_view_metric_table_size'

permission_name = 'can_view_metric_table_size'

slug = 'table-size'

A URL safe representation of the label and unique across all metrics.

sql = "\n SELECT\n relname,\n pg_size_pretty(total_size),\n pg_size_pretty(table_size),\n pg_size_pretty(relation_size_main),\n pg_size_pretty(relation_size_fsm),\n pg_size_pretty(relation_size_vm),\n pg_size_pretty(relation_size_init)\n FROM (\n SELECT\n relname,\n pg_total_relation_size(relid) AS total_size,\n pg_table_size(relid) AS table_size,\n pg_relation_size(relid, 'main') AS relation_size_main,\n pg_relation_size(relid, 'fsm') AS relation_size_fsm,\n pg_relation_size(relid, 'vm') AS relation_size_vm,\n pg_relation_size(relid, 'init') AS relation_size_init\n FROM\n pg_stat_user_tables\n {ORDER_BY}\n) AS t\n ;\n "

The actual SQL statement that is being used to query the database. In order to make use of the *ordering*, include the string `{ORDER_BY}` in the query as necessary. For details on that value see `get_order_by_clause()`.

postgres_metrics.models module

class postgres_metrics.models.**Metric**(*id*)

Bases: `Model`

exception DoesNotExist

Bases: `ObjectDoesNotExist`

exception MultipleObjectsReturned

Bases: `MultipleObjectsReturned`

id

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

postgres_metrics.urls module

postgres_metrics.views module

postgres_metrics.views.**metrics_view**(*request*, *name*)

6.1.3 Module contents

CHAPTER

SEVEN

CHANGELOG

7.1 Under Development

- Dropped official support for Python 3.7

> **Warning:** Use version 0.15.0 for Python 3.7 support!

- Added support for Django 5.0.
- Added support for Python 3.12.
- Added support for PostgreSQL 16.

7.2 0.15.0 (2023-06-05)

- Dropped official support for Django 2.2 and Django 4.0

> **Warning:** Use version 0.14.0 for Django 2.2 or Django 4.0 support!

7.3 0.14.0 (2023-06-05)

- Dropped official support for Python 3.6.

> **Warning:** Use version 0.13.1 for Python 3.6 support!

- Added support for Django 4.1 and Django 4.2.
- Added support for Python 3.11.

7.4 0.13.1 (2022-04-02)

- Set help messages for management commands introduced in version 0.13.0.
- Fix slug attributes on `IndexSize` and `TableSize` metrics.
- Remove upper bound on django-rich dependency.

7.5 0.13.0 (2022-03-27)

- Dropped official support for Python 3.5.

  Warning: Use version 0.12 for Python 3.5 support!

- Add the `pgm_list_metrics` and `pgm_show_metric` management commands.

7.6 0.12.0 (2022-03-25)

- Dropped official support for Django 3.0 and 3.1.

  Warning: Use version 0.11 for Django 3.0 or 3.1 support!

- Loosened the requirements for psycopg2 on Django 3.1 and above. There's no need to limit to `psycopg2<2.9` anymore.

7.7 0.11.0 (2021-10-06)

- Added support for PostgreSQL 14.
- Added support for Django 4.0 and Python 3.10.
- Added dark-mode support. (PR #59)
- Fixed several accessibility issues. (PR #58)

7.8 0.10.1 (2021-04-06)

- Use pre-commit.ci for linting
- Use a single workflow file

7.9 0.10.0 (2021-03-01)

- Use humanized sorting in the `Index Size` and `Table Size` metrics.
- Extended the `Table Size` metric with additional information.

7.10 0.9.0 (2021-01-05)

- Added support for translatable column titles.
- Added support for metrics to only be available on certain PostgreSQL versions.
- Fixed an issue in the `Cache Hits` metric when a database doesn't track those metrics.

7.11 0.8.0 (2021-01-03)

- Dropped support for Django 1.11, 2.0, and 2.1.

> **Warning:** Use version 0.7 for Django 1.11, 2.0, or 2.1 support!

- Added a `Sequence Usage` that shows the extend to which a PostgreSQL sequence is been used.
- Added a screenshot of what a metric looks like to the README and docs (#39).

7.12 0.7.2 (2020-12-22)

- Fixed layout issues in Django's admin before 3.1.

7.13 0.7.1 (2020-12-22)

- Fixed layout issues with Django's new admin design in 3.1.

7.14 0.7.0 (2020-12-20)

- Updated project setup by moving to GitHub Actions
- Added compatibility for Django 1.11 to 3.1

> **Warning:** This is the last version to support Django < 2.2. Version 0.8 will only support Django 2.2, 3.0, and 3.1!

7.15 0.6.2 (2018-03-20)

- Added missing installation instruction.
- Documentation building is now part of the CI tests.

7.16 0.6.1 (2018-03-20)

- Fix release bundling.

7.17 0.6.0 (2018-03-20)

- Added permission support for metrics. Users with `is_staff=True` can now be given access to each metric individually.
- The *`get_postgres_metrics()`* template tag now returns only metrics the current user (taken from the `request` object in the template context) has access to. This means the `'django.template.context_processors.request'` context processor is now required.
- The documentation now has an intersphinx setup to Python 3 and Django
- The hard dependency on psycopg2 was dropped because installing wheel files of psycopg2 can be troublesome as outlined by the project maintainers. When using django-postgres-metrics you now need to install psycopg2 or psycopg2-binary explicitly. This is usually not an issue as either one is required by Django anyway to use Django's PostgreSQL database backend.
- Added a *`Detailed Index Usage metric`* that shows the index usage off a table per index.

7.18 0.5.0 (2017-12-25)

- Added the list of metrics on a metric's detail page

7.19 0.4.0 (2017-12-25)

- Underscores in metric column headers are now replaced by spaces
- The *`IndexUsage`* now shows floating point percentages
- Added *`IndexSize`* and *`TableSize`*
- Added per metric record and record item styling

7.20 0.3.0 (2017-11-28)

- Added description to `setup.py`
- The metric results can now be sorted on any column

7.21 0.2.0 (2017-11-21)

- Switched to Read The Docs theme for docs.
- Added "Available Extensions" metric.
- Fixed table styling in metric views. The tables now look more like Django Admin's ChangeList

7.22 0.1.1 (2017-11-21)

- Excluded `tests` from built packages.

7.23 0.1.0 (2017-11-21)

- Initial Release

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

J

L

M

N

O

P

R

S

T

U

V