Django-dbindexer Documentation

Release 1.0

Waldemar Kornewald, Thomas Wanschik

March 19, 2013

CONTENTS

With django-dbindexer you can use SQL features on NoSQL databases and abstract the differences between NoSQL databases. For example, if your database doesn't support case-insensitive queries (`iexact`, `istartswith`, etc.) you can just tell the dbindexer which models and fields should support these queries and it'll take care of maintaining the required indexes for you. It's similar for JOINs. Tell the dbindexer that you would like to use in-memory JOINs for a specific query for example and the dbindexer will make it possible. Magically, previously unsupported queries will just work. Currently, this project is in an early development stage. The long-term plan is to support more complex JOINs and at least some simple aggregates, possibly even much more.

CHAPTER

ONE

TUTORIALS

- Getting started: Get SQL features on NoSQL with django-dbindexer
- JOINs for NoSQL databases via django-dbindexer - First steps

CHAPTER

TWO

DOCUMENTATION

Dependencies: djangotoolbox, django-autoload

CHAPTER

THREE

INSTALLATION

For installation see Get SQL features on NoSQL with django-dbindexer

CHAPTER

FOUR

HOW DOES DJANGO-DBINDEXER MAKE UNSUPPORTED FIELD LOOKUP TYPES WORK?

For each filter you want to use on a field for a given model, django-dbindexer adds an additional field to that model. For example, if you want to use the `contains` filter on a `CharField` you have to add the following index definition:

```
register_index(MyModel, {'name': 'contains'})
```

django-dbindexer will then store an additional `ListField` called 'idxf_<char_field_name>_l_contains' on `MyModel`. When saving an entity, django-dbindexer will fill the `ListField` with all substrings of the `CharField`'s reversed content i.e. if `CharField` stores `'Jiraiya'` then the `ListField` stores `['J', 'iJ', 'riJ', 'ariJ' ..., 'ayiariJ']`. When querying on that `CharField` using `contains`, django-dbindexer delegates this filter using `startswith` on the `ListField` with the reversed query string i.e. `filter(<char_field_name>__contains='ira')` => `filter('idxf_<char_field_name>_l_contains'__startswith='ari')` which matches the content of the list and gives back the correct result set. On App Engine `startswith` gets converted to ">=" and "<" filters for example.

In the following is listed which fields will be added for a specific filter/lookup type:

- `__iexact` using an additional `CharField` and a `__exact` query
- `__istartswith` creates an additional `CharField`. Uses a `__startswith` query
- `__endswith` using an additional `CharField` and a `__startswith` query
- `__iendswith` using an additional `CharField` and a `__startswith` query
- `__year` using an additional `IntegerField``and a ``__exact` query
- `__month` using an additional `IntegerField` and a `__exact` query
- `__day` using an additional `IntegerField` and a `__exact` query
- `__week_day` using an additional `IntegerField` and a `__exact` query
- `__contains` using an additional `ListField` and a `__startswith` query
- `__icontains` using an additional `ListField` and a `__startswith` query
- `__regex` using an additional `ListField` and a `__exact` query
- `__iregex` using an additional `ListField` and a `__exact` query

For App Engine users using djangoappengine this means that you can use all django field lookup types for example.

MongoDB users using django-mongodb-engine can benefit from this because case-insensitive filters can be handled as efficient case-sensitive filters for example.

For regex filters you have to specify which regex filter you would like to execute:

```
register_index(MyModel, {'name': ('iexact', re.compile('\/\*.*?\*\/', re.I)})
```

This will allow you to use the following filter:

```
MyModel.objects.all().filter(name__iregex='\/\*.*?\*\/')
```

CHAPTER

FIVE

BACKEND SYSTEM

django-dbindexer uses backends to resolve lookups. You can specify which backends to use via `DBINDEXER_BACKENDS`

```
# settings.py:

DBINDEXER_BACKENDS = (
    'dbindexer.backends.BaseResolver',
    'dbindexer.backends.InMemoryJOINResolver',
)
```

The `BaseResolver` is responsible for resolving lookups like `__iexact` or `__regex` for example. The `InMemoryJOINResolver` is used to resolve JOINs in-memory. The `ConstantFieldJOINResolver` uses denormalization in order to resolve JOINs. For more information see JOINs via denormalization for NoSQL coders, Part 1 is then done automatically by the `ConstantFieldJOINResolver` for you. :)

CHAPTER

SIX

LOADING INDEXES

First of all, you need to install django-autoload. Then you have to create a site configuration module which loads the index definitions. The module name has to be specified in the settings:

```
# settings.py:
AUTOLOAD_SITECONF = 'dbindexes'
```

Now, there are two ways to load database index definitions in the `AUTOLOAD_SITECONF` module: auto-detection or manual listing of modules.

Note: by default `AUTOLOAD_SITECONF` is set to your `ROOT_URLCONF`.

6.1 dbindexer.autodiscover

`autodiscover` will search for `dbindexes.py` in all `INSTALLED_APPS` and load them. It's like in django's admin interface. Your `AUTOLOAD_SITECONF` module would look like this:

```
# dbindexes.py:
import dbindexer
dbindexer.autodiscover()
```

6.2 Manual imports

Alternatively, you can import the desired index definition modules directly:

```
# dbindexes.py:
import myapp.dbindexes
import otherapp.dbindexes
```