2.5. Querying the database

`Document` classes have an `objects` attribute, which is used for accessing the objects in the database associated with the class. The `objects` attribute is actually a `QuerySetManager`, which creates and returns a new `QuerySet` object on access. The `QuerySet` object may be iterated over to fetch documents from the database:

```
# Prints out the names of all the users in the database
for user in User.objects:
    print user.name
```

Note

As of MongoEngine 0.8 the querysets utilise a local cache. So iterating it multiple times will only cause a single query. If this is not the desired behaviour you can call `no_cache` (version **0.8.3+**) to return a non-caching queryset.

2.5.1. Filtering queries

The query may be filtered by calling the `QuerySet` object with field lookup keyword arguments. The keys in the keyword arguments correspond to fields on the `Document` you are querying:

```
# This will return a QuerySet that will only iterate over users whose
# 'country' field is set to 'uk'
uk_users = User.objects(country='uk')
```

Fields on embedded documents may also be referred to using field lookup syntax by using a double-underscore in place of the dot in object attribute access syntax:

```
# This will return a QuerySet that will only iterate over pages that have
# been written by a user whose 'country' field is set to 'uk'
uk_pages = Page.objects(author__country='uk')
```

Note

(version **0.9.1+**) if your field name is like mongodb operator name (for example type, lte, lt...) and you want to place it at the end of lookup keyword mongoengine automatically prepend $ to it. To avoid this use __ at the end of your lookup keyword. For example if your field name is `type` and you want to query by this field you must use `.objects(user__type__="admin")` instead of `.objects(user__type="admin")`

2.5.2. Query operators

Operators other than equality may also be used in queries — just attach the operator name to a key with a double-underscore:

```
# Only find users whose age is 18 or less
young_users = Users.objects(age__lte=18)
```

Available operators are as follows:

`ne` – not equal to

`lt` – less than

`lte` – less than or equal to

`gt` – greater than

`gte` – greater than or equal to

`not` – negate a standard check, may be used before other operators (e.g. `Q(age__not__mod=(5, 0))`)

`in` – value is in list (a list of values should be provided)

`nin` – value is not in list (a list of values should be provided)

`mod` – `value % x == y`, where `x` and `y` are two provided values

`all` – every item in list of values provided is in array

`size` – the size of the array is

`exists` – value for field exists

2.5.2.1. String queries

The following operators are available as shortcuts to querying with regular expressions:

`exact` – string field exactly matches value

`iexact` – string field exactly matches value (case insensitive)

`contains` – string field contains value

`icontains` – string field contains value (case insensitive)

`startswith` – string field starts with value

`istartswith` – string field starts with value (case insensitive)

`endswith` – string field ends with value

`iendswith` – string field ends with value (case insensitive)

`wholeword` – string field contains whole word

`iwholeword` – string field contains whole word (case insensitive)

`regex` – string field match by regex

`iregex` – string field match by regex (case insensitive)

`match` – performs an $elemMatch so you can match an entire document within an array

2.5.2.2. Geo queries

There are a few special operators for performing geographical queries. The following were added in MongoEngine 0.8 for `PointField`, `LineStringField` and `PolygonField`:

`geo_within` – check if a geometry is within a polygon. For ease of use it accepts either a geojson geometry or just the polygon coordinates eg:

```
loc.objects(point__geo_within=[[[40, 5], [40, 6], [41, 6], [40, 5]]])
loc.objects(point__geo_within={"type": "Polygon",
                        "coordinates": [[[40, 5], [40, 6], [41, 6], [40, 5]]]})
```

`geo_within_box` – simplified geo_within searching with a box eg:

```
loc.objects(point__geo_within_box=[(-125.0, 35.0), (-100.0, 40.0)])
loc.objects(point__geo_within_box=[<bottom left coordinates>, <upper right
coordinates>])
```

`geo_within_polygon` – simplified geo_within searching within a simple polygon eg:

```
loc.objects(point__geo_within_polygon=[[40, 5], [40, 6], [41, 6], [40, 5]])
loc.objects(point__geo_within_polygon=[ [ <x1> , <y1> ] ,
                                        [ <x2> , <y2> ] ,
                                        [ <x3> , <y3> ] ])
```

`geo_within_center` – simplified geo_within the flat circle radius of a point eg:

```
loc.objects(point__geo_within_center=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_center=[ [ <x>, <y> ] , <radius> ])
```

`geo_within_sphere` – simplified geo_within the spherical circle radius of a point eg:

```
loc.objects(point__geo_within_sphere=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_sphere=[ [ <x>, <y> ] , <radius> ])
```

`geo_intersects` – selects all locations that intersect with a geometry eg:

```
# Inferred from provided points lists:
loc.objects(poly__geo_intersects=[40, 6])
loc.objects(poly__geo_intersects=[[40, 5], [40, 6]])
loc.objects(poly__geo_intersects=[[[40, 5], [40, 6], [41, 6], [41, 5], [40, 5]]])

# With geoJson style objects
loc.objects(poly__geo_intersects={"type": "Point", "coordinates": [40, 6]})
loc.objects(poly__geo_intersects={"type": "LineString",
                                  "coordinates": [[40, 5], [40, 6]]})
loc.objects(poly__geo_intersects={"type": "Polygon",
                                  "coordinates": [[[40, 5], [40, 6], [41, 6], [41, 5],
[40, 5]]]})
```

`near` – find all the locations near a given point:

```
loc.objects(point__near=[40, 5])
loc.objects(point__near={"type": "Point", "coordinates": [40, 5]})
```

You can also set the maximum and/or the minimum distance in meters as well:

```
loc.objects(point__near=[40, 5], point__max_distance=1000)
loc.objects(point__near=[40, 5], point__min_distance=100)
```

The older 2D indexes are still supported with the `GeoPointField`:

`within_distance` – provide a list containing a point and a maximum distance (e.g. [(41.342, -87.653), 5])

`within_spherical_distance` – same as above but using the spherical geo model (e.g. [(41.342, -87.653), 5/earth_radius])

`near` – order the documents by how close they are to a given point

`near_sphere` – Same as above but using the spherical geo model

`within_box` – filter documents to those within a given bounding box (e.g. [(35.0, -125.0), (40.0, -100.0)])

`within_polygon` – filter documents to those within a given polygon (e.g. [(41.91,-87.69), (41.92,-87.68), (41.91,-87.65), (41.89,-87.65)]).

> **Note**
>
> Requires Mongo Server 2.0

`max_distance` – can be added to your location queries to set a maximum distance.

`min_distance` – can be added to your location queries to set a minimum distance.

2.5.2.3. Querying lists

On most fields, this syntax will look up documents where the field specified matches the given value exactly, but when the field refers to a `ListField`, a single item may be provided, in which case lists that contain that item will be matched:

```
class Page(Document):
    tags = ListField(StringField())

# This will match all pages that have the word 'coding' as an item in the
# 'tags' list
Page.objects(tags='coding')
```

It is possible to query by position in a list by using a numerical value as a query operator. So if you wanted to find all pages whose first tag was `db`, you could use the following query:

```
Page.objects(tags__0='db')
```

If you only want to fetch part of a list eg: you want to paginate a list, then the *slice* operator is required:

```
# comments - skip 5, limit 10
Page.objects.fields(slice__comments=[5, 10])
```

For updating documents, if you don't know the position in a list, you can use the $ positional operator

```
Post.objects(comments__by="joe").update(**{'inc__comments__$__votes': 1})
```

However, this doesn't map well to the syntax so you can also use a capital S instead

```
Post.objects(comments__by="joe").update(inc__comments__S__votes=1)
```

> **Note**
>
> Due to **Mongo**, currently the $ operator only applies to the first matched item in the query.

2.5.2.4. Raw queries

It is possible to provide a raw `PyMongo` query as a query parameter, which will be integrated directly into the query. This is done using the `__raw__` keyword argument:

```
Page.objects(__raw__={'tags': 'coding'})
```

Similarly, a raw update can be provided to the `update()` method:

```
Page.objects(tags='coding').update(__raw__={'$set': {'tags': 'coding'}})
```

And the two can also be combined:

```
Page.objects(__raw__={'tags': 'coding'}).update(__raw__={'$set': {'tags': 'coding'}})
```

2.5.2.5. Update with Aggregation Pipeline

It is possible to provide a raw `PyMongo` aggregation update parameter, which will be integrated directly into the update. This is done by using `__raw__` keyword argument to the update method and provide the pipeline as a list Update with Aggregation Pipeline

```
# 'tags' field is set to 'coding is fun'
Page.objects(tags='coding').update(__raw__=[
    {"$set": {"tags": {"$concat": ["$tags", "is fun"]}}}
    ],
)
```

New in version 0.23.2.

2.5.3. Sorting/Ordering results

It is possible to order the results by 1 or more keys using `order_by()`. The order may be specified by prepending each of the keys by "+" or "-". Ascending order is assumed if there's no prefix.:

```
# Order by ascending date
blogs = BlogPost.objects().order_by('date')    # equivalent to .order_by('+date')

# Order by ascending date first, then descending title
blogs = BlogPost.objects().order_by('+date', '-title')
```

2.5.4. Limiting and skipping results

Just as with traditional ORMs, you may limit the number of results returned or skip a number or results in you query. `limit()` and `skip()` methods are available on `QuerySet` objects, but the *array-slicing* syntax is preferred for achieving this:

```
# Only the first 5 people
users = User.objects[:5]

# All except for the first 5 people
users = User.objects[5:]

# 5 users, starting from the 11th user found
users = User.objects[10:15]
```

You may also index the query to retrieve a single result. If an item at that index does not exists, an `IndexError` will be raised. A shortcut for retrieving the first result and returning `None` if no result exists is provided (`first()`):

```
>>> # Make sure there are no users
>>> User.drop_collection()
>>> User.objects[0]
IndexError: list index out of range
>>> User.objects.first() == None
True
>>> User(name='Test User').save()
>>> User.objects[0] == User.objects.first()
True
```

2.5.4.1. Retrieving unique results

To retrieve a result that should be unique in the collection, use `get()`. This will raise `DoesNotExist` if no document matches the query, and `MultipleObjectsReturned` if more than one document matched the query. These exceptions are merged into your document definitions eg: *MyDoc.DoesNotExist*

A variation of this method, get_or_create() existed, but it was unsafe. It could not be made safe, because there are no transactions in mongoDB. Other approaches should be investigated, to ensure you don't accidentally duplicate data when using something similar to this method. Therefore it was deprecated in 0.8 and removed in 0.10.

2.5.5. Default Document queries

By default, the objects `objects` attribute on a document returns a `QuerySet` that doesn't filter the collection – it returns all objects. This may be changed by defining a method on a document that modifies a queryset. The method should accept two arguments – `doc_cls` and `queryset`. The first argument is the `Document` class that the method is defined on (in this sense, the method is more like a `classmethod()` than a regular method), and the second argument is the initial queryset. The method needs to be decorated with `queryset_manager()` in order for it to be recognised.

```
class BlogPost(Document):
    title = StringField()
    date = DateTimeField()

    @queryset_manager
    def objects(doc_cls, queryset):
        # This may actually also be done by defining a default ordering for
        # the document, but this illustrates the use of manager methods
        return queryset.order_by('-date')
```

You don't need to call your method `objects` – you may define as many custom manager methods as you like:

```
class BlogPost(Document):
    title = StringField()
    published = BooleanField()

    @queryset_manager
    def live_posts(doc_cls, queryset):
        return queryset.filter(published=True)

BlogPost(title='test1', published=False).save()
BlogPost(title='test2', published=True).save()
assert len(BlogPost.objects) == 2
assert len(BlogPost.live_posts()) == 1
```

2.5.6. Custom QuerySets

Should you want to add custom methods for interacting with or filtering documents, extending the `QuerySet` class may be the way to go. To use a custom `QuerySet` class on a document, set `queryset_class` to the custom class in a `Document`'s `meta` dictionary:

```
class AwesomerQuerySet(QuerySet):

    def get_awesome(self):
        return self.filter(awesome=True)

class Page(Document):
    meta = {'queryset_class': AwesomerQuerySet}

# To call:
Page.objects.get_awesome()
```

New in version 0.4.

2.5.7. Aggregation

MongoDB provides some aggregation methods out of the box, but there are not as many as you typically get with an RDBMS. MongoEngine provides a wrapper around the built-in methods and provides some of its own, which are implemented as Javascript code that is executed on the database server.

2.5.7.1. Counting results

Just as with limiting and skipping results, there is a method on a `QuerySet` object – `count()`:

```
num_users = User.objects.count()
```

You could technically use `len(User.objects)` to get the same result, but it would be significantly slower than `count()`. When you execute a server-side count query, you let MongoDB do the heavy lifting and you receive a single integer over the wire. Meanwhile, `len()` retrieves all the results, places them in a local cache, and finally counts them. If we compare the performance of the two operations, `len()` is much slower than `count()`.

2.5.7.2. Further aggregation

You may sum over the values of a specific field on documents using `sum()`:

```
yearly_expense = Employee.objects.sum('salary')
```

> **Note**
>
> If the field isn't present on a document, that document will be ignored from the sum.

To get the average (mean) of a field on a collection of documents, use `average()`:

```
mean_age = User.objects.average('age')
```

As MongoDB provides native lists, MongoEngine provides a helper method to get a dictionary of the frequencies of items in lists across an entire collection – `item_frequencies()`. An example of its use would be generating "tag-clouds":

```
class Article(Document):
    tag = ListField(StringField())

# After adding some tagged articles...
tag_freqs = Article.objects.item_frequencies('tag', normalize=True)

from operator import itemgetter
top_tags = sorted(tag_freqs.items(), key=itemgetter(1), reverse=True)[:10]
```

2.5.7.3. MongoDB aggregation API

If you need to run aggregation pipelines, MongoEngine provides an entry point to Pymongo's aggregation framework through `aggregate()`. Check out Pymongo's documentation for the syntax and pipeline. An example of its use would be:

```
class Person(Document):
    name = StringField()

Person(name='John').save()
Person(name='Bob').save()

pipeline = [
    {"$sort" : {"name" : -1}},
    {"$project": {"_id": 0, "name": {"$toUpper": "$name"}}}
    ]
data = Person.objects().aggregate(pipeline)
assert data == [{'name': 'BOB'}, {'name': 'JOHN'}]
```

2.5.8. Query efficiency and performance

There are a couple of methods to improve efficiency when querying, reducing the information returned by the query or efficient dereferencing .

2.5.8.1. Retrieving a subset of fields

Sometimes a subset of fields on a `Document` is required, and for efficiency only these should be retrieved from the database. This issue is especially important for MongoDB, as fields may often be extremely large (e.g. a `ListField` of `EmbeddedDocument`s, which represent the comments on a blog post. To select only a subset of fields, use `only()`, specifying the fields you want to retrieve as its arguments. Note that if fields that are not downloaded are accessed, their default value (or `None` if no default value is provided) will be given:

```
>>> class Film(Document):
...     title = StringField()
...     year = IntField()
...     rating = IntField(default=3)
...
>>> Film(title='The Shawshank Redemption', year=1994, rating=5).save()
>>> f = Film.objects.only('title').first()
>>> f.title
'The Shawshank Redemption'
>>> f.year   # None
>>> f.rating # default value
3
```

Note

The `exclude()` is the opposite of `only()` if you want to exclude a field.

If you later need the missing fields, just call `reload()` on your document.

2.5.8.2. Getting related data

When iterating the results of `ListField` or `DictField` we automatically dereference any `DBRef` objects as efficiently as possible, reducing the number the queries to mongo.

There are times when that efficiency is not enough, documents that have `ReferenceField` objects or `GenericReferenceField` objects at the top level are expensive as the number of queries to MongoDB can quickly rise.

To limit the number of queries use `select_related()` which converts the QuerySet to a list and dereferences as efficiently as possible. By default `select_related()` only dereferences any references to the depth of 1 level. If you have more complicated documents and want to dereference more of the object at once then increasing the `max_depth` will dereference more levels of the document.

2.5.8.3. Turning off dereferencing

Sometimes for performance reasons you don't want to automatically dereference data. To turn off dereferencing of the results of a query use `no_dereference()` on the queryset like so:

```
post = Post.objects.no_dereference().first()
assert(isinstance(post.author, DBRef))
```

You can also turn off all dereferencing for a fixed period by using the `no_dereference` context manager:

```
with no_dereference(Post) as Post:
    post = Post.objects.first()
    assert(isinstance(post.author, DBRef))

# Outside the context manager dereferencing occurs.
assert(isinstance(post.author, User))
```

2.5.9. Advanced queries

Sometimes calling a `QuerySet` object with keyword arguments can't fully express the query you want to use – for example if you need to combine a number of constraints using *and* and *or*. This is made possible in MongoEngine through the `Q` class. A `Q` object represents part of a query, and can be initialised using the same keyword-argument syntax you use to query documents. To build a complex query, you may combine `Q` objects using the `&` (and) and `|` (or) operators. To use a `Q` object, pass it in as the first positional argument to `Document.objects` when you filter it by calling it with keyword arguments:

```
from mongoengine.queryset.visitor import Q

# Get published posts
Post.objects(Q(published=True) | Q(publish_date__lte=datetime.now()))

# Get top posts
Post.objects((Q(featured=True) & Q(hits__gte=1000)) | Q(hits__gte=5000))
```

Warning

You have to use bitwise operators. You cannot use `or`, `and` to combine queries as `Q(a=a) or Q(b=b)` is not the same as `Q(a=a) | Q(b=b)`. As `Q(a=a)` equates to true `Q(a=a) or Q(b=b)` is the same as `Q(a=a)`.

2.5.10. Atomic updates

Documents may be updated atomically by using the **`update_one()`**, **`update()`** and **`modify()`** methods on a **`QuerySet`** or **`modify()`** and **`save()`** (with **`save_condition`** argument) on a **`Document`**. There are several different "modifiers" that you may use with these methods:

`set` – set a particular value

`set_on_insert` – set only if this is new document `need to add upsert=True`_

`unset` – delete a particular value (since MongoDB v1.3)

`max` – update only if value is bigger

`min` – update only if value is smaller

`inc` – increment a value by a given amount

`dec` – decrement a value by a given amount

`push` – append a value to a list

`push_all` – append several values to a list

`pop` – remove the first or last element of a list depending on the value

`pull` – remove a value from a list

`pull_all` – remove several values from a list

`add_to_set` – add value to a list only if its not in the list already

`rename` – rename the key name

The syntax for atomic updates is similar to the querying syntax, but the modifier comes before the field, not after it:

```
>>> post = BlogPost(title='Test', page_views=0, tags=['database'])
>>> post.save()
>>> BlogPost.objects(id=post.id).update_one(inc__page_views=1)
>>> post.reload()  # the document has been changed, so we need to reload it
>>> post.page_views
1
>>> BlogPost.objects(id=post.id).update_one(set__title='Example Post')
>>> post.reload()
>>> post.title
'Example Post'
>>> BlogPost.objects(id=post.id).update_one(push__tags='nosql')
>>> post.reload()
>>> post.tags
['database', 'nosql']
```

Note

If no modifier operator is specified the default will be `$set`. So the following sentences are identical:

```
>>> BlogPost.objects(id=post.id).update(title='Example Post')
>>> BlogPost.objects(id=post.id).update(set__title='Example Post')
```

Note

In version 0.5 the `save()` runs atomic updates on changed documents by tracking changes to that document.

The positional operator allows you to update list items without knowing the index position, therefore making the update a single atomic operation. As we cannot use the *$* syntax in keyword arguments it has been mapped to *S*:

```
>>> post = BlogPost(title='Test', page_views=0, tags=['database', 'mongo'])
>>> post.save()
>>> BlogPost.objects(id=post.id, tags='mongo').update(set__tags__S='mongodb')
>>> post.reload()
>>> post.tags
['database', 'mongodb']
```

From MongoDB version 2.6, push operator supports $position value which allows to push values with index:

```
>>> post = BlogPost(title="Test", tags=["mongo"])
>>> post.save()
>>> post.update(push__tags__0=["database", "code"])
>>> post.reload()
>>> post.tags
['database', 'code', 'mongo']
```

Note

Currently only top level lists are handled, future versions of mongodb / pymongo plan to support nested positional operators. See The $ positional operator.

2.5.11. Server-side javascript execution

Javascript functions may be written and sent to the server for execution. The result of this is the return value of the Javascript function. This functionality is accessed through the `exec_js()` method on `QuerySet()` objects. Pass in a string containing a Javascript function as the first argument.

The remaining positional arguments are names of fields that will be passed into you Javascript function as its arguments. This allows functions to be written that may be executed on any field in a collection (e.g. the `sum()` method, which accepts the name of the field to sum over as its argument). Note that field names passed in in this manner are automatically translated to the names used on the database (set using the `name` keyword argument to a field constructor).

Keyword arguments to `exec_js()` are combined into an object called `options`, which is available in the Javascript function. This may be used for defining specific parameters for your function.

Some variables are made available in the scope of the Javascript function:

`collection` – the name of the collection that corresponds to the `Document` class that is being used; this should be used to get the `Collection` object from `db` in Javascript code

`query` – the query that has been generated by the `QuerySet` object; this may be passed into the `find()` method on a `Collection` object in the Javascript function

`options` – an object containing the keyword arguments passed into `exec_js()`

The following example demonstrates the intended usage of `exec_js()` by defining a function that sums over a field on a document (this functionality is already available through `sum()` but is shown here for sake of example):

```
def sum_field(document, field_name, include_negatives=True):
    code = """
    function(sumField) {
        var total = 0.0;
        db[collection].find(query).forEach(function(doc) {
            var val = doc[sumField];
            if (val >= 0.0 || options.includeNegatives) {
                total += val;
            }
        });
        return total;
    }
    """
    options = {'includeNegatives': include_negatives}
    return document.objects.exec_js(code, field_name, **options)
```

As fields in MongoEngine may use different names in the database (set using the `db_field` keyword argument to a `Field` constructor), a mechanism exists for replacing MongoEngine field names with the database field names in Javascript code. When accessing a field on a collection object, use square-bracket notation, and prefix the MongoEngine field name with a tilde. The field name that follows the tilde will be translated to the name used in the database. Note that when referring to fields on embedded documents, the name of the `EmbeddedDocumentField`, followed by a dot, should be used before the name of the field on the embedded document. The following example shows how the substitutions are made:

```
class Comment(EmbeddedDocument):
    content = StringField(db_field='body')

class BlogPost(Document):
    title = StringField(db_field='doctitle')
    comments = ListField(EmbeddedDocumentField(Comment), name='cs')

# Returns a list of dictionaries. Each dictionary contains a value named
# "document", which corresponds to the "title" field on a BlogPost, and
# "comment", which corresponds to an individual comment. The substitutions
# made are shown in the comments.
BlogPost.objects.exec_js("""
function() {
    var comments = [];
    db[collection].find(query).forEach(function(doc) {
        // doc[~comments] -> doc["cs"]
        var docComments = doc[~comments];

        for (var i = 0; i < docComments.length; i++) {
            // doc[~comments][i] -> doc["cs"][i]
            var comment = doc[~comments][i];

            comments.push({
                // doc[~title] -> doc["doctitle"]
                'document': doc[~title],

                // comment[~comments.content] -> comment["body"]
                'comment': comment[~comments.content]
            });
        }
    });
    return comments;
}
""")
```