fiqs Documentation

Release 0.3

Pierre Mourlanne

Mar 20, 2019

Contents

1 Compatibility **3**

2 Contributing **5**

3 Contents **7**

3.1 Flatten result 7

3.2 Models 10

3.3 Making queries with fiqs 13

fiqs helps you make queries against Elasticsearch, and more easily consume the results. It is built on top of the official Python Elasticsearch client and the great Elasticsearch DSL library.

You can still dive closer to the Elasticsearch JSON DSL by accessing the Elasticsearch DSL client or even the Elasticsearch python client.

fiqs can help you in the following ways:

- A helper function can flatten the result dictionary returned by Elasticsearch
- A model class, a la Django:
  - Automatically generate a mapping
  - Less verbose aggregations and metrics
  - Less verbose filtering (soon)
  - Automatically add missing buckets (soon)

CHAPTER 1

Compatibility

fiqs is compatible with Elasticsearch 5.X and works with both Python 2.7 and Python 3.3

CHAPTER 2

Contributing

The fiqs project is hosted on GitHub

To run the tests on your machine use this command: `python setup.py test` Some tests are used to generate results output from Elasticsearch. To run them you will need to run a docker container on your machine: `docker run -d -p 8200:9200 -p 8300:9300 elasticsearch:5.0.2` and then run `py.test -k docker`.

CHAPTER 3

Contents

3.1 Flatten result

Consuming the results from an Elasticsearch query can be troublesome. fiqs exposes a `flatten_result` function that transforms an elasticsearch-dsl `Result`, or a dictionary, into the list of its nodes. You will lose access to some data (`doc_count_error_upper_bound`, `sum_other_doc_count`, the `hits` etc.) so beware.

Here is a basic example with an aggregation and a metric:

```
print(flatten_result({
    "_shards": {
        ...
    },
    "hits": {
        ...
    },
    "aggregations": {
        "shop": {
            "buckets": [
                {
                    "doc_count": 30,
                    "key": 1,
                    "total_sales": {
                        "value": 12345.0
                    },
                },
                {
                    "doc_count": 20,
                    "key": 2,
                    "total_sales": {
                        "value": 23456.0
                    },
                },
                {
```

(continues on next page)

(continued from previous page)

```
                    "doc_count": 10,
                    "key": 3,
                    "total_sales": {
                        "value": 34567.0
                    },
                },
            ],
            "doc_count_error_upper_bound": 0,
            "sum_other_doc_count": 0,
        },
    },
}))
# [
#     {
#         "shop": 1,
#         "doc_count": 30,
#         "total_sales": 12345.0,
#     },
#     {
#         "shop": 2,
#         "doc_count": 20,
#         "total_sales": 23456.0,
#     },
#     {
#         "shop": 3,
#         "doc_count": 10,
#         "total_sales": 34567.0,
#     },
# ]
```

`flatten_result` can handle multiple aggregations on the same level, and nested aggregations. It can also handled nested fields:

```
print(flatten_result({
    ...
    "aggregations": {
        "products": {
            "doc_count": 1540,
            "product_type": {
                "buckets": [
                    {
                        "avg_product_price": {
                            "value": 179.53889943074003,
                        },
                        "doc_count": 527,
                        "key": "product_type_3",
                    },
                    {
                        "avg_product_price": {
                            "value": 159.18296529968455,
                        },
                        "doc_count": 317,
                        "key": "product_type_2",
                    },
                    {
                        "avg_product_price": {
```

(continues on next page)

(continued from previous page)

```
                            "value": 152.76785714285714,
                        },
                        "doc_count": 280,
                        "key": "product_type_1",
                    },
                ],
                "doc_count_error_upper_bound": 0,
                "sum_other_doc_count": 0,
            },
        },
    }
}))
# [
#     {
#         "avg_product_price": 179.53889943074003,
#         "product_type": "product_type_3",
#         "doc_count": 527,
#     },
#     {
#         "avg_product_price": 159.18296529968455,
#         "product_type": "product_type_2",
#         "doc_count": 317,
#     },
#     {
#         "avg_product_price": 152.76785714285714,
#         "product_type": "product_type_1",
#         "doc_count": 280,
#     },
# ]
```

3.1.1 A word on reverse nested aggregations

`flatten_result` cannot distinguish between a nested bucket and a reverse nested aggregation. If you want to flatten an Elasticsearch result with reverse nested aggregations, make sure these aggregations' names start with `reverse_nested`:

```
{
    'aggs': {
        'products': {
            'aggs': {
                'product_id': {
                    'aggs': {
                        'reverse_nested_root': {  # This aggregation starts with␣
↪`reverse_nested`
                            'aggs': {
                                'avg_price': {
                                    'avg': {
                                        'field': 'price',
                                    },
                                },
                            },
                            'reverse_nested': {},
                        },
                    },
                    'terms': {
```

(continues on next page)

(continued from previous page)

```
                        'field': 'products.product_id',
                    },
                },
            },
            'nested': {
                'path': 'products',
            }
        },
    },
}
```

3.2 Models

fiqs lets you create Model classes, a la Django, which automatically generate an elasticsearch mapping, and allows you to write cleaner queries.

A model is a class inheriting from `fiqs.models.Model`. It needs to define a doc_type, an index and its fields:

```
from fiqs import fields, models

class Sale(models.Model):
    index = 'sale_data'
    doc_type = 'sale'

    id = fields.IntegerField()
    shop_id = fields.IntegerField()
    client_id = fields.KeywordField()

    timestamp = fields.DateField()
    price = fields.IntegerField()
    payment_type = fields.KeywordField(choices=['wire_transfer', 'cash', 'store_credit
↪'])
```

The `doc_type` will be used for the mapping, the `index` for the queries. Instead of defining these values as class attributes, you can override the class methods `get_index` and `get_doc_type`:

```
@classmethod
def get_index(cls, *args, **kwargs):
    if not cls.index:
        raise NotImplementedError('Model class should define an index')

    return cls.index

@classmethod
def get_doc_type(cls, *args, **kwargs):
    if not cls.doc_type:
        raise NotImplementedError('Model class should define a doc_type')

    return cls.doc_type
```

3.2.1 Model fields

This section contains all the API references for fields, including the field options and the field types.

Field options

The following arguments are available to all field types. All are optional, except `type`.

type

This is a string that tells fiqs which field datatype will be used in Elasticsearch. This option is **mandatory**.

choices

A list of possible values for the field. fiqs will use it to fill the missing buckets. It can also contains a list of tuples, where the first element is the key, and the second is a 'pretty key':

```
payment_type = fields.KeywordField(choices=[
    ('wire_transfer', _('Wire transfer')),
    ('cash', _('Cash')),
    ('store_credit', _('Store credit')),
])
```

data

A dictionary containing data used in the aggregations. For the time being, only `size` is used.

parent

Used for nested documents to define the name of the parent document. For example:

```
from fiqs import models

class Sale(models.Model):
    ...

    products = fields.NestedField()
    product_id = fields.KeywordField(parent='products')
    ...
    parts = fields.NestedField(parent='products')
    part_id = fields.KeywordField(parent='paths')
```

storage_field

The name of the field in your Elasticsearch cluster. By default fiqs will use the field's name. In the case of nested fields, fiqs will use the `storage_field` as the path.

unit

Not yet used.

verbose_name

A human-readable name for the field. If the verbose name isn't given, fiqs will use the field's name in the model. Not yet used.

Field types

TextField

A field with the `text` Elasticsearch data type.

KeywordField

A field with the `keyword` Elasticsearch data type.

DateField

A field with the `date` Elasticsearch data type.

LongField

A field with the `long` Elasticsearch data type.

IntegerField

A field with the `integer` Elasticsearch data type.

ShortField

A field with the `short` Elasticsearch data type.

ByteField

A field with the `byte` Elasticsearch data type.

DoubleField

A field with the `double` Elasticsearch data type.

FloatField

A field with the `float` Elasticsearch data type.

DayOfWeekField

A field inheriting from ByteField. It accepts `iso` as a keyword argument. Depending on the value of `iso`, this field will have data and choices matching weekdays or isoweekdays.

HourOfDayField

A field inheriting from ByteField. By default, it will be able to contain values betweek 0 and 23.

BooleanField

A field with the `boolean` Elasticsearch data type.

NestedField

A field with the `nested` Elasticsearch data type.

3.2.2 Mapping

Model classes expose a `get_mapping` class method, that returns a strict and dynamic elasticsearch-dsl Mapping object. You can use it to create or update the mapping in your Elasticsearch cluster:

```
from elasticsearch import Elasticsearch

client = Elasticsearch(['http://my.cluster.com'])
mapping = MyModel.get_mapping()
client.indices.create(index='my_index', body={'mappings': mapping.to_dict()})
```

3.3 Making queries with fiqs

3.3.1 The `FQuery` object

fiqs exposes a `FQuery` object which lets you write less verbose simple queries against ElasticSearch. It is built on top of the elasticsearch-dsl Search object. Here is a quick example of what `FQuery` can do, compared to elasticsearch-dsl:

```
from elasticsearch_dsl import Search
from fiqs.aggregations import Sum
from fiqs.query import FQuery

from .models import Sale

# The elasticsearch-dsl way
search = Search(...)
search.aggs.bucket(
    'shop_id', 'terms', field='shop_id',
).bucket(
    'client_id', 'terms', field='client_id',
).metric(
```

(continues on next page)

(continued from previous page)

```
    'total_sales', 'sum', field='price',
)
result = search.execute()

# The FQuery way
search = Search(...)
fquery = FQuery(search).values(
    total_sales=Sum(Sale.price),
).group_by(
    Sale.shop_id,
    Sale.client_id,
)
result = fquery.eval()
```

Loss of expresiveness

Let's start with a warning :> FQuery may allow you to write cleaner and more re-usable queries, but at the cost of a loss of expresiveness. For example, you will not be able to have metrics at multiple aggregation levels. You may not be able to use FQuery for all your queries, and that's OK!

`FQuery` options

A `FQuery` object only needs an elasticsearch-dsl object to get started. You may also configure the following options:

- `default_size`: the size used by default in aggregations built by this object.

`eval` call

To execute the Elasticsearch query, you need to call `eval` on the FQuery object. This call accepts the following arguments:

- `flat`: If *False*, will return the elasticsearch-dsl *Result* object, without flattening the result. Note that you cannot ask for a flat result if you used computed expressions. *True* by default.
- `fill_missing_buckets`: If *False*, FQuery will not try to fill the missing buckets. For more details see *Filling missing buckets*. Note that fiqs cannot fill the missing buckets in non flat mode. *True* by default.

3.3.2 Values

You need to call `values` on a FQuery object to specify the metrics you want to use in your request. values accepts both arguments and keyword arguments:

```
from fiqs.aggregation import Sum, Avg

from .models import Sale

FQuery(search).values(
    Avg(Sale.price),
    total_sales=Sum(Sale.price),
)
```

In this case, the nodes will contain two keys for the metrics: *total_sales*, and *sale__price__avg*, a string representation of the *Avg(Sale.price)* metric. A `values` call returns the FQuery object, to allow chaining calls.

fiqs contains several classes, which all take a field as argument, to help you make these metric calls:

Avg

Used for the Elasticsearch avg aggregation.

Cardinality

Used for the Elasticsearch cardinality aggregation

Count

Used if you only want to count the documents present in your search. This aggregation does not change the Elasticsearh request, since it always returns the number of documents in the doc_count.

Max

Used for the Elasticsearch max aggregation

Min

Used for the Elasticsearch min aggregation

Sum

Used for the Elasticsearch sum aggregation

Operations

fiqs lets you query computed fields, created with operations on a model's fields. For example:

```
from fiqs.aggregation import Sum

from .models import TrafficCount

FQuery(search).values(
    total_traffic=Addition(
        Sum(TrafficCount.in_count),
        Sum(TrafficCount.out_count),
    ),
    in_traffic_ratio=Ratio(
        Sum(TrafficCount.in_count),
        Addition(
            Sum(TrafficCount.in_count),
            Sum(TrafficCount.out_count),
        ),
    ),
)
```

The three existing operations are Addition, Subtraction and Ratio. **Do note that these operations cannot be used in non-flat mode.** For example this will not work:

```
fquery = FQuery(search).values(
    total_traffic=Addition(
        Sum(TrafficCount.in_count),
        Sum(TrafficCount.out_count),
    ),
).group_by(
    TrafficCount.shop_id,
)
results = fquery.eval(flat=False)  # Will raise an exception
```

ReverseNested

The `ReverseNested` class lets you make reverse nested aggregation. It takes as a first argument the path for the reverse nested aggregation (it can be empty) and a list of expressions:

```
class Sale(models.Model):
    price = fields.IntegerField()

    products = fields.NestedField()
    product_id = fields.KeywordField(parent='products')

    parts = fields.NestedField(parent='products')
    part_id = fields.KeywordField(parent='parts')

# Number of sales by product_id
FQuery(search).values(
    ReverseNested(
        '',
        Count(Sale),
    ),
).group_by(
    Sale.product_id,
)
# Number of products by part_id
FQuery(search).values(
    ReverseNested(
        Sale.products,  # You can give a field instead of a string
        Count(Sale.products),  # Or `Count(Sale)`, both work
    ),
).group_by(
    Sale.product_id,
    Sale.part_id,
)
# Total and average price by product id
FQuery(search).values(
    ReverseNested(
        Sale,  # Or `''`, both work
        avg_sale_price=Avg(Sale.price),
        total_sale_price=Sum(Sale.price),
    ),
).group_by(
    Sale.product_id,
)
```

3.3.3 Group by

You can call `group_by` on a FQuery object to add aggregations. Like `values`, `group_by` returns the FQuery object, to allow chaining. fiqs lets you build only one aggregation, which can be as deep as you need it to be. In a group_by call, you can use any fiqs Field, or Field subclass, object. fiqs also offers Field subclasses that help you configure your aggregation:

FieldWithChoices

A `FieldWithChoices` takes as argument an existing field, and a list of choice:

```
FieldWithChoices(Sale.shop_id, choices=(['Atlanta', 'Phoenix', 'NYC']))
```

This field is useful if you want to tune the capacity of FQuery to fill the missing buckets.

FieldWithRanges

A `FieldWithRanges` takes as argument an existing field, with a list of ranges. Ranges can either be a list of dictionaries forming an Elasticsearch range aggregation, or a list of tuples:

```
ranges = [
    {
        'from': 1,
        'to': 5,
        'key': '1 - 5',
    },
    {
        'from': 5,
        'to': 11,
        'key': '5 - 11',
    },
]
# Equivalent to :
ranges = [
    (1, 5),
    (5, 11),
]
FieldWithRanges(Sale.shop_id, ranges=ranges)
```

Do note that the *from* value (or the first tuple value) is **included**, and the *to* value (or the second tuple value) is **excluded**.

DataExtendedField

A `DataExtendedField` takes as argument an existing field, and a data dictionary:

```
DataExtendedField(Sale.shop_id, size=5)
```

This field is useful if you want to to fine tune the aggregation. In the example we changed the `size` parameter that will be used in the Elasticsearch aggregation.

GroupedField

A `GroupedField` aims to replicate the behavior of a filters aggregation. It takes as argument an existing field and a dictionary used to build the buckets:

```
shop_groups = {
    'group_a': [1, 2, 3, ],
    'group_b': [4, 5, 6, ],
}
# Number of Sale objects, grouped according to the `groups` argument
# One bucket will contain the Sale objects with shop_id in [1, 2, 3, ]
# The other bucket will contain the Sale objects with shop_id in [4, 5, 6, ]
fquery = FQuery(search).values(
    Count(Sale),
).group_by(
    GroupedField(Sale.shop_id, groups=groups),
)
```

3.3.4 Order by

You can call `order_by` on a FQuery object, to order the Elasticsearch result as you want. `order_by` returns the FQuery object, to allow chaining. order_by expects a dictionary that will be directly used in the aggregation as a sort:

```
FQuery(search).values(
    total_sales=Sum(Sale.price),
).group_by(
    Sale.shop_id,
).order_by(
    {'total_sales': 'desc'},
)
```

In this example, the Elasticsearch result will be ordered by total sales, in descending order.

3.3.5 Executing the query

Calling `eval` on the Fquery object will execute the Elasticsearch query and return the result.

Form of the result

FQuery will automatically flatten the result returned by Elasticsearch, as detailed *here*. It will also cast the value, depending on your model's fields.

Each field may implement a `get_casted_value` method. FQuery will use this method to cast values returned by Elasticsearch. For example:

```
class IntegerField(Field):
    def __init__(self, **kwargs):
        super(IntegerField, self).__init__('integer', **kwargs)

    def get_casted_value(self, v):
        return int(v) if v is not None else v
```

As of today, only the following fields implement this method:

- LongField, IntegerField, ShortField, ByteField and field inheriting from them cast values as int

- DoubleField and FloatField cast values as float
- DateField cast values as datetime, **ignoring the milliseconds**

Filling missing buckets

By default, FQuery will try to add buckets missing from the Elasticsearch result. FQuery uses several heuristics to determine which buckets are missing, as we will see below. FQuery will fill the group_by values with the missing keys, and the metric values with `None`.

- If a field in the group_by defines the `choices` attribute, FQuery will expect all the choices' keys to be present as keys in the Elasticsearch buckets:

```
# Our model
class Sale(Model):
    shop_id = fields.IntegerField(choices=(1, 2, 3, ))
    price = fields.IntegerField()

# Our query
results = FQuery(search).values(
    total_sales=Sum(Sale.price),
).group_by(
    Sale.shop_id,
).eval()

# Elasticsearch result, notice there is no bucket with shop_id 1
# {
#     [...],
#     "aggregations": {
#         "shop": {
#             "buckets": [
#                 {
#                     "doc_count": 20,
#                     "key": 2,
#                     "total_sales": {
#                         "value": 123,
#                     },
#                 },
#                 {
#                     "doc_count": 10,
#                     "key": 3,
#                     "total_sales": {
#                         "value": 456,
#                     },
#                 },
#             ],
#             [...],
#         },
#     },
# }

# FQuery result, with the empty line added
# [
#     {
#         'shop_id': 2,
#         'doc_count': 20,
#         'total_sales': 123,
```

(continues on next page)

(continued from previous page)

```
#     },
#     {
#         'shop_id': 3,
#         'doc_count': 10,
#         'total_sales': 456,
#     },
#     {
#         'shop_id': 1,
#         'doc_count': 0,
#         'total_sales': None,
#     },
# ]
```

- If an aggregate in the group_by returns a value when calling `choice_keys`, FQuery will expect all the keys to be present in the Elasticsearch buckets. Only available with daily DateHistogram for the time being.
- Finally, FQuery will look at all the values each key takes in the result buckets, and will expect all keys to be present in all buckets:

```
# Our model
class Sale(Model):
    shop_id = fields.IntegerField()
    price = fields.IntegerField()
    payment_type = fields.KeywordField(choices=('wire_transfer', 'cash', ))

# Our query
results = FQuery(search).values(
    total_sales=Sum(Sale.price),
).group_by(
    Sale.payment_type,
    Sale.shop_id,
).eval()

# Elasticsearch result
# {
#     [...],
#     "aggregations": {
#         "payment_type": {
#             "buckets": [
#                 {
#                     "key": "wire_transfer",
#                     "shop_id": {
#                         "buckets": [
#                             {
#                                 doc_count: 10,
#                                 "key": 1,
#                                 "total_sales": {
#                                     "value": 123,
#                                 },
#                             },
#                         ],
#                     },
#                 },
#                 {
#                     "key": "cash",
#                     "shop_id": {
#                         "buckets": [
```

(continues on next page)

(continued from previous page)

```
#                                 {
#                                     doc_count: 20,
#                                     "key": 2,
#                                     "total_sales": {
#                                         "value": 456,
#                                     },
#                                 },
#                             ],
#                         },
#                     },
#                 ],
#             },
#     },
# }

# FQuery result, with two empty lines added
# [
#     {
#         'shop_id': 1,
#         'doc_count': 10,
#         'total_sales': 123,
#         'payment_type': 'wire_transfer',
#     },
#     {
#         'shop_id': 2,
#         'doc_count': 0,
#         'total_sales': None,
#         'payment_type': 'wire_transfer',
#     },
#     {
#         'shop_id': 2,
#         'doc_count': 20,
#         'total_sales': 456,
#         'payment_type': 'cash',
#     },
#     {
#         'shop_id': 1,
#         'doc_count': 0,
#         'total_sales': None,
#         'payment_type': 'cash',
#     },
# ]
```