pandagg Documentation

Release 0.1

Léonard Binet

Sep 23, 2021

Contents

CHAPTER 1

Principles

This library focuses on two principles:

- stick to the **tree** structure of Elasticsearch objects
- provide simple and flexible interfaces to make it easy and intuitive to use in an interactive usage

1.1 Elasticsearch tree structures

Many Elasticsearch objects have a **tree** structure, ie they are built from a hierarchy of **nodes**:

- a mappings (tree) is a hierarchy of fields (nodes)
- a query (tree) is a hierarchy of query clauses (nodes)
- an aggregation (tree) is a hierarchy of aggregation clauses (nodes)
- an aggregation response (tree) is a hierarchy of response buckets (nodes)

This library sticks to that structure by providing a flexible syntax distinguishing **trees** and **nodes**, **trees** all inherit from lighttree.Tree class, whereas nodes all inherit from lighttree.Node class.

1.2 Interactive usage

pandagg is designed for both for "regular" code repository usage, and "interactive" usage (ipython or jupyter notebook usage with autocompletion features inspired by pandas design).

Some classes are not intended to be used elsewhere than in interactive mode (ipython), since their purpose is to serve auto-completion features and convenient representations.

Namely:

- `IMapping`: used to interactively navigate in mapping and run quick aggregations on some fields
- *`IResponse`*: used to interactively navigate in an aggregation response

These use case will be detailed in following sections.

CHAPTER 2

User Guide

pandagg library provides interfaces to perform **read** operations on cluster.

2.1 Search

Search class is intended to perform requests, and refers to Elasticsearch search api:

```
>>> from pandagg.search import Search
>>>
>>> client = ElasticSearch(hosts=['localhost:9200'])
>>> search = Search(using=client, index='movies')\
>>>     .size(2)\
>>>     .groupby('decade', 'histogram', interval=10, field='year')\
>>>     .groupby('genres', size=3)\
>>>     .aggs('avg_rank', 'avg', field='rank')\
>>>     .agg('avg_nb_roles', 'avg', field='nb_roles')\
>>>     .filter('range', year={"gte": 1990})
```

```
>>> search
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "year": {
              "gte": 1990
            }
          }
        }
      ]
    }
  },
```

(continues on next page)

(continued from previous page)

```
  "aggs": {
    "decade": {
      "histogram": {
        "field": "year",
        "interval": 10
      },
      "aggs": {
        "genres": {
          "terms": {
            "field": "genres",
            "size": 3
          },
          "aggs": {
            "avg_rank": {
              "avg": {
                "field": "rank"
              }
            },
            "avg_nb_roles": {
              "avg": {
                "field": "nb_roles"
              }
            }
          }
        }
      }
    }
  },
  "size": 2
}
```

It relies on:

- *Query* to build queries, **query** or **post_filter** (see *Query*),
- *Aggs* to build aggregations (see *Aggregation*)

Note: All methods described below return a new *Search* instance, and keep unchanged the initial search request.

```
>>> from pandagg.search import Search
>>> initial_s = Search()
>>> enriched_s = initial_s.query('terms', genres=['Comedy', 'Short'])
```

```
>>> initial_s.to_dict()
{}
```

```
>>> enriched_s.to_dict()
{'query': {'terms': {'genres': ['Comedy', 'Short']}}}
```

2.1.1 Query part

The **query** or **post_filter** parts of a *Search* instance are available respectively under **_query** and **_post_filter** attributes.

```
>>> search._query.__class__
pandagg.tree.query.abstract.Query
>>> search._query.show()
<Query>
bool
└── filter
    └── range, field=year, gte=1990
```

To enrich **query** of a search request, methods are exactly the same as for a *Query* instance.

```
>>> Search().must_not('range', year={'lt': 1980})
{
  "query": {
    "bool": {
      "must_not": [
        {
          "range": {
            "year": {
              "lt": 1980
            }
          }
        }
      ]
    }
  }
}
```

See section *Query* for more details.

2.1.2 Aggregations part

The **aggregations** part of a *Search* instance is available under **_aggs** attribute.

```
>>> search._aggs.__class__
pandagg.tree.aggs.aggs.Aggs
>>> search._aggs.show()
<Aggregations>
decade                                          <histogram, field="year", interval=10>
└── genres                                             <terms, field="genres", size=3>
    ├── avg_nb_roles                                          <avg, field="nb_roles">
    └── avg_rank                                              <avg, field="rank">
```

To enrich **aggregations** of a search request, methods are exactly the same as for a *Aggs* instance.

```
>>> Search()\
>>> .groupby('decade', 'histogram', interval=10, field='year')\
>>> .agg('avg_rank', 'avg', field='rank')
{
  "aggs": {
    "decade": {
      "histogram": {
        "field": "year",
        "interval": 10
      },
      "aggs": {
        "avg_rank": {
```

(continues on next page)

(continued from previous page)

```
          "avg": {
            "field": "rank"
          }
        }
      }
    }
  }
}
```

See section *Aggregation* for more details.

2.1.3 Other search request parameters

size, **sources**, **limit** etc, all those parameters are documented in *Search* documentation and their usage is quite self-explanatory.

2.1.4 Request execution

To a execute a search request, you must first have bound it to an Elasticsearch client beforehand:

```
>>> from elasticsearch import Elasticsearch
>>> client = Elasticsearch(hosts=['localhost:9200'])
```

Either at instantiation:

```
>>> from pandagg.search import Search
>>> search = Search(using=client, index='movies')
```

Either with `using()` method:

```
>>> from pandagg.search import Search
>>> search = Search()\
>>> .using(client=client)\
>>> .index('movies')
```

Executing a *Search* request using *execute()* will return a *Response* instance (see more in *Response*).

```
>>> response = search.execute()
>>> response
<Response> took 58ms, success: True, total result >=10000, contains 2 hits
>>> response.__class__
pandagg.response.Response
```

2.2 Query

The `Query` class provides :

- multiple syntaxes to declare and udpate a query
- query validation (with nested clauses validation)
- ability to insert clauses at specific points
- tree-like visual representation

2.2.1 Declaration

2.2.1.1 From native "dict" query

Given the following query:

```
>>> expected_query = {'bool': {'must': [
>>>     {'terms': {'genres': ['Action', 'Thriller']}},
>>>     {'range': {'rank': {'gte': 7}}},
>>>     {'nested': {
>>>         'path': 'roles',
>>>         'query': {'bool': {'must': [
>>>             {'term': {'roles.gender': {'value': 'F'}}},
>>>             {'term': {'roles.role': {'value': 'Reporter'}}}]}}
>>>          }
>>>     }}
>>> ]}}
```

To instantiate `Query`, simply pass "dict" query as argument:

```
>>> from pandagg.query import Query
>>> q = Query(expected_query)
```

A visual representation of the query is available with `show()`:

```
>>> q.show()
<Query>
bool
└── must
    ├── nested, path="roles"
    │   └── query
    │       └── bool
    │           └── must
    │               ├── term, field=roles.gender, value="F"
    │               └── term, field=roles.role, value="Reporter"
    ├── range, field=rank, gte=7
    └── terms, genres=["Action", "Thriller"]
```

Call `to_dict()` to convert it to native dict:

```
>>> q.to_dict()
{'bool': {
    'must': [
        {'range': {'rank': {'gte': 7}}},
        {'terms': {'genres': ['Action', 'Thriller']}},
        {'bool': {'must': [
            {'term': {'roles.role': {'value': 'Reporter'}}},
            {'term': {'roles.gender': {'value': 'F'}}}]}}}
        ]}
    ]
}}
```

```
>>> from pandagg.utils import equal_queries
>>> equal_queries(q.to_dict(), expected_query)
True
```

Note: *equal_queries* function won't consider order of clauses in must/should parameters since it actually doesn't

matter in Elasticsearch execution, ie

```
>>> equal_queries({'must': [A, B]}, {'must': [B, A]})
True
```

2.2.1.2 With DSL classes

Pandagg provides a DSL to declare this query in a quite similar fashion:

```
>>> from pandagg.query import Nested, Bool, Range, Term, Terms
```

```
>>> q = Bool(must=[
>>>     Terms(genres=['Action', 'Thriller']),
>>>     Range(rank={"gte": 7}),
>>>     Nested(
>>>         path='roles',
>>>         query=Bool(must=[
>>>             Term(roles__gender='F'),
>>>             Term(roles__role='Reporter')
>>>         ])
>>>     )
>>> ])
```

All these classes inherit from `Query` and thus provide the same interface.

```
>>> from pandagg.query import Query
>>> isinstance(q, Query)
True
```

2.2.1.3 With flattened syntax

In the flattened syntax, the query clause type is used as first argument:

```
>>> from pandagg.query import Query
>>> q = Query('terms', genres=['Action', 'Thriller'])
```

2.2.2 Query enrichment

All methods described below return a new `Query` instance, and keep unchanged the initial query.

For instance:

```
>>> from pandagg.query import Query
>>> initial_q = Query()
>>> enriched_q = initial_q.query('terms', genres=['Comedy', 'Short'])
```

```
>>> initial_q.to_dict()
None
```

```
>>> enriched_q.to_dict()
{'terms': {'genres': ['Comedy', 'Short']}}
```

Note: Calling `to_dict()` on an empty Query returns *None*

```
>>> from pandagg.query import Query
>>> Query().to_dict()
None
```

2.2.2.1 query() method

The base method to enrich a `Query` is `query()`.

Considering this query:

```
>>> from pandagg.query import Query
>>> q = Query()
```

`query()` accepts following syntaxes:

from dictionnary:

```
>>> q.query({"terms": {"genres": ['Comedy', 'Short']})
```

flattened syntax:

```
>>> q.query("terms", genres=['Comedy', 'Short'])
```

from Query instance (this includes DSL classes):

```
>>> from pandagg.query import Terms
>>> q.query(Terms(genres=['Action', 'Thriller']))
```

2.2.2.2 Compound clauses specific methods

`Query` instance also exposes following methods for specific compound queries:

(TODO: detail allowed syntaxes)

Specific to bool queries:

- `bool()`
- `filter()`
- `must()`
- `must_not()`
- `should()`

Specific to other compound queries:

- `nested()`
- `constant_score()`
- `dis_max()`
- `function_score()`
- `has_child()`

- has_parent()
- parent_id()
- pinned_query()
- script_score()
- boost()

2.2.2.3 Inserted clause location

On all insertion methods detailed above, by default, the inserted clause is placed at the top level of your query, and generates a bool clause if necessary.

Considering the following query:

```
>>> from pandagg.query import Query
>>> q = Query('terms', genres=['Action', 'Thriller'])
>>> q.show()
<Query>
terms, genres=["Action", "Thriller"]
```

A bool query will be created:

```
>>> q = q.query('range', rank={"gte": 7})
>>> q.show()
<Query>
bool
└── must
    ├── range, field=rank, gte=7
    └── terms, genres=["Action", "Thriller"]
```

And reused if necessary:

```
>>> q = q.must_not('range', year={"lte": 1970})
>>> q.show()
<Query>
bool
├── must
│   ├── range, field=rank, gte=7
│   └── terms, genres=["Action", "Thriller"]
└── must_not
    └── range, field=year, lte=1970
```

Specifying a specific location requires to name queries :

```
>>> from pandagg.query import Nested
```

```
>>> q = q.nested(path='roles', _name='nested_roles', query=Term('roles.gender', value=
↪'F'))
>>> q.show()
<Query>
bool
├── must
│   ├── nested, _name=nested_roles, path="roles"
│   │   └── query
│   │       └── term, field=roles.gender, value="F"
```

(continues on next page)

(continued from previous page)

```
│   ├── range, field=rank, gte=7
│   └── terms, genres=["Action", "Thriller"]
└── must_not
    └── range, field=year, lte=1970
```

Doing so allows to insert clauses above/below given clause using *parent*/*child* parameters:

```
>>> q = q.query('term', roles__role='Reporter', parent='nested_roles')
>>> q.show()
<Query>
bool
├── must
│   ├── nested, _name=nested_roles, path="roles"
│   │   └── query
│   │       └── bool
│   │           └── must
│   │               ├── term, field=roles.role, value="Reporter"
│   │               └── term, field=roles.gender, value="F"
│   ├── range, field=rank, gte=7
│   └── terms, genres=["Action", "Thriller"]
└── must_not
    └── range, field=year, lte=1970
```

TODO: explain *parent_param*, *child_param*, *mode* merging strategies on same named clause etc..

2.3 Aggregation

The `Aggs` class provides :

- multiple syntaxes to declare and udpate a aggregation
- aggregation clause validation
- ability to insert clauses at specific locations (and not just below last manipulated clause)

2.3.1 Declaration

2.3.1.1 From native "dict" query

Given the following aggregation:

```
>>> expected_aggs = {
>>>   "decade": {
>>>     "histogram": {"field": "year", "interval": 10},
>>>     "aggs": {
>>>       "genres": {
>>>         "terms": {"field": "genres", "size": 3},
>>>         "aggs": {
>>>           "max_nb_roles": {
>>>             "max": {"field": "nb_roles"}
>>>           },
>>>           "avg_rank": {
>>>             "avg": {"field": "rank"}
>>>           }
```

(continues on next page)

(continued from previous page)

```
>>>           }
>>>         }
>>>       }
>>>     }
>>> }
```

To declare `Aggs`, simply pass "dict" query as argument:

```
>>> from pandagg.agg import Aggs
>>> a = Aggs(expected_aggs)
```

A visual representation of the query is available with `show()`:

```
>>> a.show()
<Aggregations>
decade                                          <histogram, field="year", interval=10>
└── genres                                             <terms, field="genres", size=3>
    ├── max_nb_roles                                           <max, field="nb_roles">
    └── avg_rank                                                   <avg, field="rank">
```

Call `to_dict()` to convert it to native dict:

```
>>> a.to_dict() == expected_aggs
True
```

2.3.1.2 With DSL classes

Pandagg provides a DSL to declare this query in a quite similar fashion:

```
>>> from pandagg.agg import Histogram, Terms, Max, Avg
>>>
>>> a = Histogram("decade", field='year', interval=10, aggs=[
>>>     Terms("genres", field="genres", size=3, aggs=[
>>>         Max("max_nb_roles", field="nb_roles"),
>>>         Avg("avg_rank", field="range")
>>>     ]),
>>> ])
```

All these classes inherit from `Aggs` and thus provide the same interface.

```
>>> from pandagg.agg import Aggs
>>> isinstance(a, Aggs)
True
```

2.3.1.3 With flattened syntax

In the flattened syntax, the first argument is the aggregation name, the second argument is the aggregation type, the following keyword arguments define the aggregation body:

```
>>> from pandagg.query import Aggs
>>> a = Aggs('genres', 'terms', size=3)
>>> a.to_dict()
{'genres': {'terms': {'field': 'genres', 'size': 3}}}
```

2.3.2 Aggregations enrichment

Aggregations can be enriched using two methods:

- `aggs()`
- `groupby()`

Both methods return a new `Aggs` instance, and keep unchanged the initial Aggregation.

For instance:

```
>>> from pandagg.aggs import Aggs
>>> initial_a = Aggs()
>>> enriched_a = initial_a.agg('genres_agg', 'terms', field='genres')
```

```
>>> initial_q.to_dict()
None
```

```
>>> enriched_q.to_dict()
{'genres_agg': {'terms': {'field': 'genres'}}}
```

Note: Calling `to_dict()` on an empty Aggregation returns *None*

```
>>> from pandagg.agg import Aggs
        >>> Aggs().to_dict()
        None
```

TODO >>> Aggs().to_dict() None

TODO

2.4 Response

When executing a search request via *execute()* method of *Search*, a *Response* instance is returned.

```
>>> from elasticsearch import Elasticsearch
>>> from pandagg.search import Search
>>>
>>> client = ElasticSearch(hosts=['localhost:9200'])
>>> response = Search(using=client, index='movies')\
>>>     .size(2)\
>>>     .filter('term', genres='Documentary')\
>>>     .agg('avg_rank', 'avg', field='rank')\
>>>     .execute()
```

```
>>> response
<Response> took 9ms, success: True, total result >=10000, contains 2 hits
```

```
>>> response.__class__
pandagg.response.Response
```

ElasticSearch raw dict response is available under *data* attribute:

```
>>> response.data
{
    'took': 9, 'timed_out': False, '_shards': {'total': 1, 'successful': 1, 'skipped
↪': 0, 'failed': 0},
    'hits': {'total': {'value': 10000, 'relation': 'gte'},
    'max_score': 0.0,
    'hits': [{'_index': 'movies', ...}],
    'aggregations': {'avg_rank': {'value': 6.496829211219546}}
}
```

2.4.1 Hits

Hits are available under *hits* attribute:

```
>>> response.hits
<Hits> total: >10000, contains 2 hits
```

```
>>> response.hits.total
{'value': 10000, 'relation': 'gte'}
```

```
>>> response.hits.hits
[<Hit 642> score=0.00, <Hit 643> score=0.00]
```

Those hits are instances of *Hit*.

Directly iterating over *Response* will return those hits:

```
>>> list(response)
[<Hit 642> score=0.00, <Hit 643> score=0.00]
```

```
>>> hit = next(iter(response))
```

Each hit contains the raw dict under *data* attribute:

```
>>> hit.data
{'_index': 'movies',
 '_type': '_doc',
 '_id': '642',
 '_score': 0.0,
 '_source': {'movie_id': 642,
  'name': '10 Tage in Calcutta',
  'year': 1984,
  'genres': ['Documentary'],
  'roles': None,
  'nb_roles': 0,
  'directors': [{'director_id': 33096,
    'first_name': 'Reinhard',
    'last_name': 'Hauff',
    'full_name': 'Reinhard Hauff',
    'genres': ['Documentary', 'Drama', 'Musical', 'Short']}],
  'nb_directors': 1,
  'rank': None}}
```

```
>>> hit._index
'movies'
```

```
>>> hit._source
{'movie_id': 642,
 'name': '10 Tage in Calcutta',
 'year': 1984,
 'genres': ['Documentary'],
 'roles': None,
 'nb_roles': 0,
 'directors': [{'director_id': 33096,
   'first_name': 'Reinhard',
   'last_name': 'Hauff',
   'full_name': 'Reinhard Hauff',
   'genres': ['Documentary', 'Drama', 'Musical', 'Short']}],
 'nb_directors': 1,
 'rank': None}
```

If pandas dependency is installed, hits can be parsed as a dataframe:

```
>>> hits.to_dataframe()
     _index  _score _type
↪
↪       directors         genres  movie_id                      name  nb_directors
↪nb_roles  rank roles  year
_id
642  movies     0.0  _doc  [{'director_id': 33096, 'first_name': 'Reinhard', 'last_
↪name': 'Hauff', 'full_name': 'Reinhard Hauff', 'genres': ['Documentary', 'Drama',
↪'Musical', 'Short']}]  [Documentary]       642        10 Tage in Calcutta
↪  1         0  None  None  1984
643  movies     0.0  _doc                                  [{'director_id': 32148,
↪'first_name': 'Tanja', 'last_name': 'Hamilton', 'full_name': 'Tanja Hamilton',
↪'genres': ['Documentary']}]  [Documentary]       643  10 Tage, ein ganzes Leben
↪        1         0  None  None  2004
```

2.4.2 Aggregations

Aggregations are handled differently, the *aggregations* attribute of a *Response* returns a *Aggregations* instance, that provides specific parsing abilities in addition to exposing raw aggregations response under *data* attribute.

Let's build a bit more complex aggregation query to showcase its functionalities:

```
>>> from elasticsearch import Elasticsearch
>>> from pandagg.search import Search
>>>
>>> client = Elasticsearch(hosts=['localhost:9200'])
>>> response = Search(using=client, index='movies')\
>>>     .size(0)\
>>>     .groupby('decade', 'histogram', interval=10, field='year')\
>>>     .groupby('genres', size=3)\
>>>     .agg('avg_rank', 'avg', field='rank')\
>>>     .aggs('avg_nb_roles', 'avg', field='nb_roles')\
>>>     .filter('range', year={"gte": 1990})\
>>>     .execute()
```

Note: for more details about how to build aggregation query, consult *Aggregation* section

Using *data* attribute:

```
>>> response.aggregations.data
{'decade': {'buckets': [{'key': 1990.0,
'doc_count': 79495,
'genres': {'doc_count_error_upper_bound': 0,
 'sum_other_doc_count': 38060,
 'buckets': [{'key': 'Drama',
   'doc_count': 12232,
   'avg_nb_roles': {'value': 18.518067364290385},
   'avg_rank': {'value': 5.981429367965072}},
  {'key': 'Short',
...
```

2.4.2.1 Tree serialization

Using *to_normalized()*:

```
>>> response.aggregations.to_normalized()
{'level': 'root',
 'key': None,
 'value': None,
 'children': [{'level': 'decade',
   'key': 1990.0,
   'value': 79495,
   'children': [{'level': 'genres',
     'key': 'Drama',
     'value': 12232,
     'children': [{'level': 'avg_rank',
       'key': None,
       'value': 5.981429367965072},
      {'level': 'avg_nb_roles', 'key': None, 'value': 18.518067364290385}]},
    {'level': 'genres',
     'key': 'Short',
     'value': 12197,
     'children': [{'level': 'avg_rank',
       'key': None,
       'value': 6.311325829450123},
    ...
```

Using *to_interactive_tree()*:

```
>>> response.aggregations.to_interactive_tree()
<IResponse>
root
├── decade=1990                                    79495
│   ├── genres=Documentary                          8393
│   │   ├── avg_nb_roles                3.7789824854045038
│   │   └── avg_rank                     6.517093241977517
│   ├── genres=Drama                               12232
│   │   ├── avg_nb_roles                18.518067364290385
│   │   └── avg_rank                     5.981429367965072
│   └── genres=Short                               12197
│       ├── avg_nb_roles                 3.023284414200213
│       └── avg_rank                     6.311325829450123
└── decade=2000                                    57649
    ├── genres=Documentary                          8639
    │   ├── avg_nb_roles                 5.581433036231045
```

(continues on next page)

(continued from previous page)

```
    │   └── avg_rank                    6.980897812811443
    ├── genres=Drama                                11500
    │   ├── avg_nb_roles               14.385391304347825
    │   └── avg_rank                    6.269675415719865
    └── genres=Short                                13451
        ├── avg_nb_roles                4.053081555274701
        └── avg_rank                     6.83625304327684
```

2.4.2.2 Tabular serialization

Doing so requires to identify a level that will draw the line between:

- grouping levels: those which will be used to identify rows (here decades, and genres), and provide **doc_count** per row
- columns levels: those which will be used to populate columns and cells (here avg_nb_roles and avg_rank)

The tabular format will suit especially well aggregations with a T shape.

Using *to_dataframe()*:

```
>>> response.aggregations.to_dataframe()
                      avg_nb_roles  avg_rank  doc_count
decade genres
1990.0 Drama           18.518067  5.981429      12232
       Short            3.023284  6.311326      12197
       Documentary      3.778982  6.517093       8393
2000.0 Short            4.053082  6.836253      13451
       Drama           14.385391  6.269675      11500
       Documentary      5.581433  6.980898       8639
```

Using *to_tabular()*:

```
>>> response.aggregations.to_tabular()
(['decade', 'genres'],
 {(1990.0, 'Drama'): {'doc_count': 12232,
   'avg_rank': 5.981429367965072,
   'avg_nb_roles': 18.518067364290385},
  (1990.0, 'Short'): {'doc_count': 12197,
   'avg_rank': 6.311325829450123,
   'avg_nb_roles': 3.023284414200213},
  (1990.0, 'Documentary'): {'doc_count': 8393,
   'avg_rank': 6.517093241977517,
   'avg_nb_roles': 3.7789824854045038},
  (2000.0, 'Short'): {'doc_count': 13451,
   'avg_rank': 6.83625304327684,
   'avg_nb_roles': 4.053081555274701},
  (2000.0, 'Drama'): {'doc_count': 11500,
   'avg_rank': 6.269675415719865,
   'avg_nb_roles': 14.385391304347825},
  (2000.0, 'Documentary'): {'doc_count': 8639,
   'avg_rank': 6.980897812811443,
   'avg_nb_roles': 5.581433036231045}})
```

Note: TODO - explain parameters:

- index_orient
- grouped_by
- expand_columns
- expand_sep
- normalize
- with_single_bucket_groups

2.5 Interactive features

Features described in this module are primarly designed for interactive usage, for instance in an *ipython shell<https://ipython.org/>*_, since one of the key features is the intuitive usage provided by auto-completion.

2.5.1 Cluster indices discovery

`discover()` function list all indices on a cluster matching a provided pattern:

```
>>> from elasticsearch import Elasticsearch
>>> from pandagg.discovery import discover
>>> client = Elasticsearch(hosts=['xxx'])
>>> indices = discover(client, index='mov*')
>>> indices
<Indices> ['movies', 'movies_fake']
```

Each of the indices is accessible via autocompletion:

```
>>> indices.movies
 <Index 'movies'>
```

An `Index` exposes: settings, mapping (interactive), aliases and name:

```
>>> movies = indices.movies
>>> movies.settings
{'index': {'creation_date': '1591824202943',
  'number_of_shards': '1',
  'number_of_replicas': '1',
  'uuid': 'v6Amj9x1Sk-trBShI-188A',
  'version': {'created': '7070199'},
  'provided_name': 'movies'}}
```

```
>>> movies.mapping
<Mapping>

_
├── directors                                                [Nested]
│   ├── director_id                                           Keyword
│   ├── first_name                                            Text
│   │   └── raw                                            ~ Keyword
│   ├── full_name                                             Text
│   │   └── raw                                            ~ Keyword
│   ├── genres                                                Keyword
│   └── last_name                                             Text
```

(continues on next page)

(continued from previous page)

```
│       └── raw                                                ~ Keyword
├── genres                                                       Keyword
├── movie_id                                                     Keyword
├── name                                                         Text
│   └── raw                                                    ~ Keyword
├── nb_directors                                                 Integer
├── nb_roles                                                     Integer
├── rank                                                         Float
├── roles                                                       [Nested]
│   ├── actor_id                                                 Keyword
│   ├── first_name                                               Text
│   │   └── raw                                                ~ Keyword
│   ├── full_name                                                Text
│   │   └── raw                                                ~ Keyword
│   ├── gender                                                   Keyword
│   ├── last_name                                                Text
│   │   └── raw                                                ~ Keyword
│   └── role                                                     Keyword
└── year                                                         Integer
```

2.5.2 Navigable mapping

The *Index* **mapping** attribute returns a `IMapping` instance that provides navigation features with autocompletion to quickly discover a large mapping:

```
>>> movies.roles
<Mapping subpart: roles>
roles                                                   [Nested]
├── actor_id                                             Integer
├── first_name                                           Text
│   └── raw                                            ~ Keyword
├── gender                                               Keyword
├── last_name                                            Text
│   └── raw                                            ~ Keyword
└── role                                                 Keyword
>>> movies.roles.first_name
<IMapping subpart: roles.first_name>
first_name                                          Text
└── raw                                           ~ Keyword
```

Note: a navigable mapping can be obtained directly using `IMapping` class without using discovery module:

```
>>> from pandagg.mapping import IMapping
>>> from examples.imdb.load import mapping
>>> m = IMapping(mapping)
>>> m.roles.first_name
<Mapping subpart: roles.first_name>
first_name                                                   Text
└── raw                                                    ~ Keyword
```

To get the complete field definition, just call it:

```
>>> movies.roles.first_name()
<Mapping Field first_name> of type text:
{
    "type": "text",
    "fields": {
        "raw": {
            "type": "keyword"
        }
    }
}
```

A **IMapping** instance can be bound to an Elasticsearch client to get quick access to aggregations computation on mapping fields.

Suppose you have the following client:

```
>>> from elasticsearch import Elasticsearch
>>> client = Elasticsearch(hosts=['localhost:9200'])
```

Client can be bound at instantiation:

```
>>> movies = IMapping(mapping, client=client, index_name='movies')
```

Doing so will generate a **a** attribute on mapping fields, this attribute will list all available aggregation for that field type (with autocompletion):

```
>>> movies.roles.gender.a.terms()
[('M', {'key': 'M', 'doc_count': 2296792}),
('F', {'key': 'F', 'doc_count': 1135174})]
```

Note: Nested clauses will be automatically taken into account.

2.5.3 Navigable aggregation response

When executing a *Search* request with aggregations, resulting aggregations can be parsed in multiple formats as described *Response*.

Suppose we execute the following search request:

```
>>> from elasticsearch import Elasticsearch
>>> from pandagg.search import Search
>>>
>>> client = ElasticSearch(hosts=['localhost:9200'])
>>> response = Search(using=client, index='movies')\
>>>     .size(0)\
>>>     .groupby('decade', 'histogram', interval=10, field='year')\
>>>     .groupby('genres', size=3)\
>>>     .agg('avg_rank', 'avg', field='rank')\
>>>     .aggs('avg_nb_roles', 'avg', field='nb_roles')\
>>>     .filter('range', year={"gte": 1990})\
>>>     .execute()
```

One of the available serialization methods for aggregations, *to_interactive_tree()*, generates an interactive tree of class *IResponse*:

```
>>> tree = response.aggregations.to_interactive_tree()
>>> tree
<IResponse>
root
├── decade=1990                                    79495
│   ├── genres=Documentary                          8393
│   │   ├── avg_nb_roles                 3.7789824854045038
│   │   └── avg_rank                      6.517093241977517
│   ├── genres=Drama                               12232
│   │   ├── avg_nb_roles                18.518067364290385
│   │   └── avg_rank                      5.981429367965072
│   └── genres=Short                               12197
│       ├── avg_nb_roles                  3.023284414200213
│       └── avg_rank                      6.311325829450123
└── decade=2000                                    57649
    ├── genres=Documentary                          8639
    │   ├── avg_nb_roles                  5.581433036231045
    │   └── avg_rank                      6.980897812811443
    ├── genres=Drama                               11500
    │   ├── avg_nb_roles                14.385391304347825
    │   └── avg_rank                      6.269675415719865
    └── genres=Short                               13451
        ├── avg_nb_roles                  4.053081555274701
        └── avg_rank                       6.83625304327684
```

This tree provides auto-completion on each node to select a subpart of the tree:

```
>>> tree.decade_1990
<IResponse subpart: decade_1990>
decade=1990                                        79495
├── genres=Documentary                              8393
│   ├── avg_nb_roles                     3.7789824854045038
│   └── avg_rank                          6.517093241977517
├── genres=Drama                                   12232
│   ├── avg_nb_roles                    18.518067364290385
│   └── avg_rank                          5.981429367965072
└── genres=Short                                   12197
    ├── avg_nb_roles                      3.023284414200213
    └── avg_rank                          6.311325829450123
```

```
>>> tree.genres_Drama
<IResponse subpart: decade_1990.genres_Drama>
genres=Drama                                       12232
├── avg_nb_roles                        18.518067364290385
└── avg_rank                              5.981429367965072
```

get_bucket_filter() returns the query that filters documents belonging to the given bucket:

```
>>> tree.decade_1990.genres_Drama.get_bucket_filter()
{'bool': {
    'must': [
        {'term': {'genres': {'value': 'Drama'}}},
        {'range': {'year': {'gte': 1990.0, 'lt': 2000.0}}}
    ],
    'filter': [{'range': {'year': {'gte': 1990}}}]
    }
}
```

`list_documents()` method actually execute this query to list documents belonging to bucket:

```
>>> tree.decade_1990.genres_Drama.list_documents(size=2, _source={"include": ['name']}
↪)
{'took': 10,
 'timed_out': False,
 '_shards': {'total': 1, 'successful': 1, 'skipped': 0, 'failed': 0},
 'hits': {'total': {'value': 10000, 'relation': 'gte'},
  'max_score': 2.4539857,
  'hits': [{'_index': 'movies',
    '_type': '_doc',
    '_id': '706',
    '_score': 2.4539857,
    '_source': {'name': '100 meter fri'}},
   {'_index': 'movies',
    '_type': '_doc',
    '_id': '714',
    '_score': 2.4539857,
    '_source': {'name': '100 Proof'}}]}}
```

Note: Examples will be based on *IMDB dataset* data.

Search class is intended to perform request (see *Search*)

```
>>> from pandagg.search import Search
>>>
>>> client = ElasticSearch(hosts=['localhost:9200'])
>>> search = Search(using=client, index='movies')\
>>>     .size(2)\
>>>     .groupby('decade', 'histogram', interval=10, field='year')\
>>>     .groupby('genres', size=3)\
>>>     .agg('avg_rank', 'avg', field='rank')\
>>>     .aggs('avg_nb_roles', 'avg', field='nb_roles')\
>>>     .filter('range', year={"gte": 1990})
```

```
>>> search
{
  "query": {
    "bool": {
      "filter": [
        {
          "range": {
            "year": {
              "gte": 1990
            }
          }
        }
      ]
    }
  },
  "aggs": {
    "decade": {
      "histogram": {
        "field": "year",
        "interval": 10
      },
```

(continues on next page)

(continued from previous page)

```
      "aggs": {
        "genres": {
          "terms": {
        ...
        ..truncated..
        ...
      }
    }
  },
  "size": 2
}
```

It relies on:

- *Query* to build queries (see *Query*),
- *Aggs* to build aggregations (see *Aggregation*)

```
>>> search._query.show()
<Query>
bool
└── filter
    └── range, field=year, gte=1990
```

```
>>> search._aggs.show()
<Aggregations>
decade                                          <histogram, field="year",␣
↪interval=10>
└── genres                                             <terms, field="genres",␣
↪size=3>
    ├── avg_nb_roles                                           <avg, field="nb_
↪roles">
    └── avg_rank                                               <avg, field=
↪"rank">
```

Executing a *Search* request using *execute()* will return a *Response* instance (see *Response*).

```
>>> response = search.execute()
>>> response
<Response> took 58ms, success: True, total result >=10000, contains 2 hits
```

```
>>> response.hits.hits
[<Hit 640> score=0.00, <Hit 641> score=0.00]
```

```
>>> response.aggregations.to_dataframe()
                       avg_nb_roles  avg_rank  doc_count
decade genres
1990.0 Drama           18.518067  5.981429      12232
       Short            3.023284  6.311326      12197
       Documentary      3.778982  6.517093       8393
2000.0 Short            4.053082  6.836253      13451
       Drama           14.385391  6.269675      11500
       Documentary      5.581433  6.980898       8639
```

On top of that some interactive features are available (see *Interactive features*).

CHAPTER 3

IMDB dataset

You might know the Internet Movie Database, commonly called IMDB.

Well it's a simple example to showcase some of Elasticsearch capabilities.

In this case, relational databases (SQL) are a good fit to store with consistence this kind of data. Yet indexing some of this data in a optimized search engine will allow more powerful queries.

3.1 Query requirements

In this example, we'll suppose most usage/queries requirements will be around the concept of movie (rather than usages focused on fetching actors or directors, even though it will still be possible with this data structure).

The index should provide good performances trying to answer these kind question (non-exhaustive):

- in which movies this actor played?
- what movies genres were most popular among decades?
- which actors have played in best-rated movies, or worst-rated movies?
- which actors movies directors prefer to cast in their movies?
- which are best ranked movies of last decade in Action or Documentary genres?
- ...

3.2 Data source

I exported following SQL tables from MariaDB following these instructions.

Relational schema is the following:

imdb tables

3.3 Index mappings

3.3.1 Overview

The base unit (document) will be a movie, having a name, rank (ratings), year of release, a list of actors and a list of directors.

Schematically:

```
Movie:
 - name
 - year
 - rank
 - [] genres
 - [] directors
 - [] actor roles
```

3.3.2 Which fields require nesting?

Since genres contain a single keyword field, in no case we need it to be stored as a nested field. On the contrary, actor roles and directors require a nested field if we consider applying multiple simultanous query clauses on their sub-fields (for instance search movie in which actor is a woman AND whose role is nurse). More information on distinction between array and nested fields here.

3.3.3 Text or keyword fields?

Some fields are easy to choose, in no situation gender will require a full text search, thus we'll store it as a keyword. On the other hand actors and directors names (first and last) will require full-text search, we'll thus opt for a text field. Yet we might want to aggregate on exact keywords to count number of movies per actor for instance. More inforamtion on distinction between text and keyword fields here

3.3.4 Mappings

```
<Mappings>
_
├── directors                                          [Nested]
│   ├── director_id                                     Keyword
│   ├── first_name                                      Text
│   │   └── raw                                      ~ Keyword
│   ├── full_name                                       Text
│   │   └── raw                                      ~ Keyword
│   ├── genres                                          Keyword
│   └── last_name                                       Text
│       └── raw                                      ~ Keyword
├── genres                                             Keyword
├── movie_id                                           Keyword
├── name                                               Text
│   └── raw                                          ~ Keyword
├── nb_directors                                       Integer
├── nb_roles                                           Integer
├── rank                                               Float
├── roles                                             [Nested]
```

(continues on next page)

(continued from previous page)

```
│   ├── actor_id                                          Keyword
│   ├── first_name                                        Text
│   │   └── raw                                         ~ Keyword
│   ├── full_name                                         Text
│   │   └── raw                                         ~ Keyword
│   ├── gender                                            Keyword
│   ├── last_name                                         Text
│   │   └── raw                                         ~ Keyword
│   └── role                                              Keyword
└── year                                                  Integer
```

3.4 Steps to start playing with your index

You can either directly use the demo index available *here* with credentials user: `pandagg`, password: `pandagg`:

Access it with following client instantiation:

```
from elasticsearch import Elasticsearch
client = Elasticsearch(
    hosts=['https://beba020ee88d49488d8f30c163472151.eu-west-2.aws.cloud.es.io:9243/
↪'],
    http_auth=('pandagg', 'pandagg')
)
```

Or follow below steps to install it yourself locally. In this case, you can either generate yourself the files, or download them from here (file md5 `b363dee23720052501e24d15361ed605`).

3.4.1 Dump tables

Follow instruction on bottom of https://relational.fit.cvut.cz/dataset/IMDb page and dump following tables in a directory:

- movies.csv
- movies_genres.csv
- movies_directors.csv
- directors.csv
- directors_genres.csv
- roles.csv
- actors.csv

3.4.2 Clone pandagg and setup environment

```
git clone git@github.com:alkemics/pandagg.git
cd pandagg

virtualenv env
python setup.py develop
pip install pandas simplejson jupyter seaborn
```

Then copy `conf.py.dist` file into `conf.py` and edit variables as suits you, for instance:

```
# your cluster address
ES_HOST = 'localhost:9200'

# where your table dumps are stored, and where serialized output will be written
DATA_DIR = '/path/to/dumps/'
OUTPUT_FILE_NAME = 'serialized.json'
```

3.4.3 Serialize movie documents and insert them

```
# generate serialized movies documents, ready to be inserted in ES
# can take a while
python examples/imdb/serialize.py

# create index with mappings if necessary, bulk insert documents in ES
python examples/imdb/load.py
```

3.4.4 Explore pandagg notebooks

An example notebook is available to showcase some of `pandagg` functionalities: here it is.

Code is present in `examples/imdb/IMDB exploration.py` file.

CHAPTER 4

pandagg package

4.1 Subpackages

4.1.1 pandagg.interactive package

4.1.1.1 Submodules

pandagg.interactive.mappings module

class pandagg.interactive.mappings.**IMappings**(*mappings*, *client=None*, *index=None*, *depth=1*, *root_path=None*, *initial_tree=None*)

Bases: *pandagg.utils.DSLMixin*, lighttree.interactive.TreeBasedObj

Interactive wrapper upon mappings tree, allowing field navigation and quick access to single clause aggregations computation.

pandagg.interactive.response module

class pandagg.interactive.response.**IResponse**(*tree*, *search*, *depth*, *root_path=None*, *initial_tree=None*)

Bases: lighttree.interactive.TreeBasedObj

Interactive aggregation response.

get_bucket_filter()

Build filters to select documents belonging to that bucket, independently from initial search query clauses.

search()

4.1.1.2 Module contents

4.1.2 pandagg.node package

4.1.2.1 Subpackages

pandagg.node.aggs package

Submodules

pandagg.node.aggs.abstract module

pandagg.node.aggs.abstract.**A**(*name*, *type_or_agg=None*, ***body*)
Accept multiple syntaxes, return a AggNode instance.

Parameters

- **type_or_agg** –
- **body** –

Returns AggNode

class pandagg.node.aggs.abstract.**AggClause**(*meta=None*, ***body*)
Bases: pandagg.node._node.Node

Wrapper around elasticsearch aggregation concept. https://www.elastic.co/guide/en/elasticsearch/reference/2.3/search-aggregations.html

Each aggregation can be seen both a Node that can be encapsulated in a parent agg.

Define a method to build aggregation request.

BLACKLISTED_MAPPING_TYPES = None

KEY = None

VALUE_ATTRS = None

WHITELISTED_MAPPING_TYPES = None

classmethod extract_bucket_value(*response*, *value_as_dict=False*)

extract_buckets(*response_value*)

get_filter(*key*)
Return filter query to list documents having this aggregation key.

Parameters **key** – string

Returns elasticsearch filter query

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

to_dict()
ElasticSearch aggregation queries follow this formatting:

```
{
    "<aggregation_name>" : {
        "<aggregation_type>" : {
            <aggregation_body>
```

(continues on next page)

(continued from previous page)

```
        }
        [,"meta" : {  [<meta_data_body>] } ]?
    }
}
```

to_dict() returns the following part (without aggregation name):

```
{
    "<aggregation_type>" : {
        <aggregation_body>
    }
    [,"meta" : {  [<meta_data_body>] } ]?
}
```

classmethod valid_on_field_type(*field_type*)

class pandagg.node.aggs.abstract.**BucketAggClause**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.AggClause*

Bucket aggregation have special abilities: they can encapsulate other aggregations as children. Each time, the extracted value is a 'doc_count'.

Provide methods: - to build aggregation request (with children aggregations) - to to extract buckets from raw response - to build query to filter documents belonging to that bucket

Note: the aggs attribute's only purpose is for children initiation with the following syntax: >>> from pandagg.aggs import Terms, Avg >>> agg = Terms(>>> field='some_path', >>> aggs={ >>> 'avg_agg': Avg(field='some_other_path') >>> } >>>)

VALUE_ATTRS = None

extract_buckets(*response_value*)

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.abstract.**FieldOrScriptMetricAgg**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MetricAgg*

Metric aggregation based on single field.

VALUE_ATTRS = None

class pandagg.node.aggs.abstract.**MetricAgg**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.AggClause*

Metric aggregation are aggregations providing a single bucket, with value attributes to be extracted.

VALUE_ATTRS = None

extract_buckets(*response_value*)

get_filter(*key*)
Return filter query to list documents having this aggregation key.

Parameters **key** – string

Returns elasticsearch filter query

class pandagg.node.aggs.abstract.**MultipleBucketAgg**(*keyed=None*, *key_path='key'*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.BucketAggClause*

IMPLICIT_KEYED = False

VALUE_ATTRS = None

extract_buckets(*response_value*)

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.abstract.**Pipeline**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

VALUE_ATTRS = None

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.abstract.**Root**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.AggClause*

Not a real aggregation. Just the initial empty dict (used as lighttree.Tree root).

KEY = '_root'

classmethod extract_bucket_value(*response*, *value_as_dict=False*)

extract_buckets(*response_value*)

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.aggs.abstract.**ScriptPipeline**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = None

VALUE_ATTRS = 'value'

class pandagg.node.aggs.abstract.**UniqueBucketAgg**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.BucketAggClause*

Aggregations providing a single bucket.

VALUE_ATTRS = None

extract_buckets(*response_value*)

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

pandagg.node.aggs.bucket module

Not implemented aggregations include: - children agg - geo-distance - geo-hash grid - ipv4 - sampler - significant terms

class pandagg.node.aggs.bucket.**Composite**(*keyed=None*, *key_path='key'*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'composite'

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**DateHistogram**(*field*, *interval=None*, *calendar_interval=None*, *fixed_interval=None*, *meta=None*, *keyed=False*, *key_as_string=True*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'date_histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**DateRange**(*field*, *key_as_string=True*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.bucket.Range*

KEY = 'date_range'

KEY_SEP = '::'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

class pandagg.node.aggs.bucket.**Filter**(*filter=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'filter'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Filters**(*filters*, *other_bucket=False*, *other_bucket_key=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

DEFAULT_OTHER_KEY = '_other_'

IMPLICIT_KEYED = True

KEY = 'filters'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Global**(*meta=None*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'global'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Histogram**(*field*, *interval*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'h

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**MatchAll**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.bucket.Filter*

class pandagg.node.aggs.bucket.**Missing**(*field*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

BLACKLISTED_MAPPING_TYPES = []

KEY = 'missing'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Nested**(*path*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Range**(*field*, *ranges*, *keyed=False*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'range'

KEY_SEP = '-'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'h

from_key

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

to_key

class pandagg.node.aggs.bucket.**ReverseNested**(*path=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'reverse_nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.node.aggs.bucket.**Terms**(*field*, *missing=None*, *size=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

Terms aggregation.

BLACKLISTED_MAPPING_TYPES = []

KEY = 'terms'

VALUE_ATTRS = ['doc_count', 'doc_count_error_upper_bound', 'sum_other_doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

pandagg.node.aggs.composite module

class pandagg.node.aggs.composite.**Composite**(*sources*, *size=None*, *after_key=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.BucketAggClause*

KEY = 'composite'

VALUE_ATTRS = ['doc_count']

extract_buckets(*response_value*)

get_filter(*key*)
In composite aggregation, key is a map, source name -> value

pandagg.node.aggs.metric module

class pandagg.node.aggs.metric.**Avg**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'avg'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**Cardinality**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'cardinality'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.metric.**ExtendedStats**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'extended_stats'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum', 'sum_of_squares', 'variance', 'std

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**GeoBound**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'geo_bounds'

VALUE_ATTRS = ['bounds']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.node.aggs.metric.**GeoCentroid**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'geo_centroid'

VALUE_ATTRS = ['location']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.node.aggs.metric.**Max**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'max'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**Min**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'min'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**PercentileRanks**(*field*, *values*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'percentile_ranks'

VALUE_ATTRS = ['values']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**Percentiles**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

Percents body argument can be passed to specify which percentiles to fetch.

KEY = 'percentiles'

VALUE_ATTRS = ['values']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**Stats**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'stats'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**Sum**(*field=None*, *script=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'sum'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.node.aggs.metric.**TopHits**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MetricAgg*

KEY = 'top_hits'

VALUE_ATTRS = ['hits']

class pandagg.node.aggs.metric.**ValueCount**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

BLACKLISTED_MAPPING_TYPES = []

KEY = 'value_count'

VALUE_ATTRS = ['value']

pandagg.node.aggs.pipeline module

Pipeline aggregations: https://www.elastic.co/guide/en/elasticsearch/reference/2.3/search-aggregations-pipeline.html

class pandagg.node.aggs.pipeline.**AvgBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'avg_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**BucketScript**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_script'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**BucketSelector**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_selector'

VALUE_ATTRS = None

class pandagg.node.aggs.pipeline.**BucketSort**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_sort'

VALUE_ATTRS = None

class pandagg.node.aggs.pipeline.**CumulativeSum**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'cumulative_sum'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**Derivative**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'derivative'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**ExtendedStatsBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'extended_stats_bucket'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum', 'sum_of_squares', 'variance', 'std

class pandagg.node.aggs.pipeline.**MaxBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'max_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**MinBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'min_bucket'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**MovingAvg**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'moving_avg'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**PercentilesBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'percentiles_bucket'

VALUE_ATTRS = ['values']

class pandagg.node.aggs.pipeline.**SerialDiff**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'serial_diff'

VALUE_ATTRS = ['value']

class pandagg.node.aggs.pipeline.**StatsBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'stats_bucket'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum']

class pandagg.node.aggs.pipeline.**SumBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)

Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'sum_bucket'

VALUE_ATTRS = ['value']

Module contents

pandagg.node.mappings package

Submodules

pandagg.node.mappings.abstract module

class pandagg.node.mappings.abstract.**ComplexField**(***body*)
Bases: *pandagg.node.mappings.abstract.Field*

KEY = None

is_valid_value(*v*)

class pandagg.node.mappings.abstract.**Field**(*multiple=None*, *nullable=True*, ***body*)
Bases: pandagg.node._node.Node

KEY = None

body

is_valid_value(*v*)

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.mappings.abstract.**RegularField**(***body*)
Bases: *pandagg.node.mappings.abstract.Field*

KEY = None

is_valid_value(*v*)

pandagg.node.mappings.field_datatypes module

https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html

class pandagg.node.mappings.field_datatypes.**Alias**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Defines an alias to an existing field.

KEY = 'alias'

class pandagg.node.mappings.field_datatypes.**Binary**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'binary'

class pandagg.node.mappings.field_datatypes.**Boolean**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'boolean'

class pandagg.node.mappings.field_datatypes.**Byte**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'byte'

class pandagg.node.mappings.field_datatypes.**Completion**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To provide auto-complete suggestions

KEY = 'completion'

class pandagg.node.mappings.field_datatypes.**ConstantKeyword**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'constant_keyword'

class pandagg.node.mappings.field_datatypes.**Date**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date'

class pandagg.node.mappings.field_datatypes.**DateNanos**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date_nanos'

class pandagg.node.mappings.field_datatypes.**DateRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date_range'

class pandagg.node.mappings.field_datatypes.**DenseVector**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record dense vectors of float values.

KEY = 'dense_vector'

class pandagg.node.mappings.field_datatypes.**Double**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'double'

class pandagg.node.mappings.field_datatypes.**DoubleRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'double_range'

class pandagg.node.mappings.field_datatypes.**Flattened**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Allows an entire JSON object to be indexed as a single field.

KEY = 'flattened'

class pandagg.node.mappings.field_datatypes.**Float**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'float'

class pandagg.node.mappings.field_datatypes.**FloatRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'float_range'

class pandagg.node.mappings.field_datatypes.**GeoPoint**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

For lat/lon points

KEY = 'geo_point'

class pandagg.node.mappings.field_datatypes.**GeoShape**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

For complex shapes like polygons

KEY = 'geo_shape'

class pandagg.node.mappings.field_datatypes.**HalfFloat**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'half_float'

class pandagg.node.mappings.field_datatypes.**Histogram**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

For pre-aggregated numerical values for percentiles aggregations.

KEY = 'histogram'

class pandagg.node.mappings.field_datatypes.**IP**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

for IPv4 and IPv6 addresses

KEY = 'ip'

class pandagg.node.mappings.field_datatypes.**Integer**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'integer'

class pandagg.node.mappings.field_datatypes.**IntegerRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'integer_range'

class pandagg.node.mappings.field_datatypes.**Join**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Defines parent/child relation for documents within the same index

KEY = 'join'

class pandagg.node.mappings.field_datatypes.**Keyword**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'keyword'

class pandagg.node.mappings.field_datatypes.**Long**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'long'

class pandagg.node.mappings.field_datatypes.**LongRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'long_range'

class pandagg.node.mappings.field_datatypes.**MapperAnnotatedText**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To index text containing special markup (typically used for identifying named entities)

KEY = 'annotated-text'

class pandagg.node.mappings.field_datatypes.**MapperMurMur3**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

To compute hashes of values at index-time and store them in the index

KEY = 'murmur3'

class pandagg.node.mappings.field_datatypes.**Nested**(***body*)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'nested'

class pandagg.node.mappings.field_datatypes.**Object**(***body*)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'object'

class pandagg.node.mappings.field_datatypes.**Percolator**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Accepts queries from the query-dsl

KEY = 'percolator'

class pandagg.node.mappings.field_datatypes.**RankFeature**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record numeric feature to boost hits at query time.

KEY = 'rank_feature'

class pandagg.node.mappings.field_datatypes.**RankFeatures**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record numeric features to boost hits at query time.

KEY = 'rank_features'

class pandagg.node.mappings.field_datatypes.**ScaledFloat**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'scaled_float'

class pandagg.node.mappings.field_datatypes.**SearchAsYouType**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

A text-like field optimized for queries to implement as-you-type completion

KEY = 'search_as_you_type'

class pandagg.node.mappings.field_datatypes.**Shape**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

For arbitrary cartesian geometries.

KEY = 'shape'

class pandagg.node.mappings.field_datatypes.**Short**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'short'

class pandagg.node.mappings.field_datatypes.**SparseVector**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record sparse vectors of float values.

KEY = 'sparse_vector'

class pandagg.node.mappings.field_datatypes.**Text**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'text'

class pandagg.node.mappings.field_datatypes.**TokenCount**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To count the number of tokens in a string

KEY = 'token_count'

class pandagg.node.mappings.field_datatypes.**WildCard**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'wildcard'

pandagg.node.mappings.meta_fields module

class pandagg.node.mappings.meta_fields.**FieldNames**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

All fields in the document which contain non-null values.

KEY = '_field_names'

class pandagg.node.mappings.meta_fields.**Id**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The document's ID.

KEY = '_id'

class pandagg.node.mappings.meta_fields.**Ignored**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

All fields in the document that have been ignored at index time because of ignore_malformed.

KEY = '_ignored'

class pandagg.node.mappings.meta_fields.**Index**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

The index to which the document belongs.

KEY = '_index'

class pandagg.node.mappings.meta_fields.**Meta**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

Application specific metadata.

KEY = '_meta'

class pandagg.node.mappings.meta_fields.**Routing**(*multiple=None*, *nullable=True*, **body)
Bases: *pandagg.node.mappings.abstract.Field*

A custom routing value which routes a document to a particular shard.

KEY = '_routing'

class pandagg.node.mappings.meta_fields.**Size**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The size of the _source field in bytes, provided by the mapper-size plugin.

KEY = '_size'

class pandagg.node.mappings.meta_fields.**Source**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The original JSON representing the body of the document.

KEY = '_source'

class pandagg.node.mappings.meta_fields.**Type**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The document's mappings type.

KEY = '_type'

Module contents

pandagg.node.query package

Submodules

pandagg.node.query.abstract module

class pandagg.node.query.abstract.**AbstractSingleFieldQueryClause**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

class pandagg.node.query.abstract.**FlatFieldQueryClause**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

Query clause applied on one single field. Example:

Exists: {"exists": {"field": "user"}} -> field = "user" -> body = {"field": "user"} >>> from pandagg.query import Exists >>> q = Exists(field="user")

DistanceFeature: {"distance_feature": {"field": "production_date", "pivot": "7d", "origin": "now"}} -> field = "production_date" -> body = {"field": "production_date", "pivot": "7d", "origin": "now"} >>> from pandagg.query import DistanceFeature >>> q = DistanceFeature(field="production_date", pivot="7d", origin="now")

class pandagg.node.query.abstract.**KeyFieldQueryClause**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

Clause with field used as key in clause body:

Term: {"term": {"user": {"value": "Kimchy", "boost": 1}}} -> field = "user" -> body = {"user": {"value": "Kimchy", "boost": 1}} >>> from pandagg.query import Term >>> q1 = Term(user={"value": "Kimchy", "boost": 1}}) >>> q2 = Term(field="user", value="Kimchy", boost=1}})

Can accept a "_implicit_param" attribute specifying which is the equivalent key when inner body isn't a dict but a raw value. For Term: _implicit_param = "value" >>> q = Term(user="Kimchy") {"term": {"user": {"value": "Kimchy"}}} -> field = "user" -> body = {"term": {"user": {"value": "Kimchy"}}}

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.query.abstract.**LeafQueryClause**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.QueryClause*

class pandagg.node.query.abstract.**MultiFieldsQueryClause**(*fields*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.query.abstract.**ParentParameterClause**
Bases: *pandagg.node.query.abstract.QueryClause*

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

pandagg.node.query.abstract.**Q**(*type_or_query=None*, ***body*)
Accept multiple syntaxes, return a QueryClause node.

Parameters

- **type_or_query** –
- **body** –

Returns QueryClause

class pandagg.node.query.abstract.**QueryClause**(*_name=None*, *accept_children=True*, *keyed=True*, *_children=None*, ***body*)
Bases: pandagg.node._node.Node

KEY = None

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

name

to_dict()

pandagg.node.query.compound module

class pandagg.node.query.compound.**Bool**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

```
>>> Bool(must=[], should=[], filter=[], must_not=[], boost=1.2)
```

KEY = 'bool'

class pandagg.node.query.compound.**Boosting**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'boosting'

class pandagg.node.query.compound.**CompoundClause**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.QueryClause*

Compound clauses can encapsulate other query clauses:

class pandagg.node.query.compound.**ConstantScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'constant_score'

class pandagg.node.query.compound.**DisMax**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'dis_max'

class pandagg.node.query.compound.**FunctionScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'function_score'

pandagg.node.query.full_text module

class pandagg.node.query.full_text.**Common**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'common'

class pandagg.node.query.full_text.**Intervals**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'intervals'

class pandagg.node.query.full_text.**Match**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match'

class pandagg.node.query.full_text.**MatchBoolPrefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_bool_prefix'

class pandagg.node.query.full_text.**MatchPhrase**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_phrase'

class pandagg.node.query.full_text.**MatchPhrasePrefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_phrase_prefix'

class pandagg.node.query.full_text.**MultiMatch**(*fields*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'multi_match'

class pandagg.node.query.full_text.**QueryString**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'query_string'

class pandagg.node.query.full_text.**SimpleQueryString**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'simple_string'

pandagg.node.query.geo module

class pandagg.node.query.geo.**GeoBoundingBox**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_bounding_box'

class pandagg.node.query.geo.**GeoDistance**(*distance*, ***body*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

KEY = 'geo_distance'

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.query.geo.**GeoPolygone**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_polygon'

class pandagg.node.query.geo.**GeoShape**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_shape'

pandagg.node.query.joining module

class pandagg.node.query.joining.**HasChild**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_child'

class pandagg.node.query.joining.**HasParent**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_parent'

class pandagg.node.query.joining.**Nested**(*path*, ***kwargs*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'nested'

class pandagg.node.query.joining.**ParentId**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'parent_id'

pandagg.node.query.shape module

class pandagg.node.query.shape.**Shape**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'shape'

pandagg.node.query.span module

pandagg.node.query.specialized module

class pandagg.node.query.specialized.**DistanceFeature**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'distance_feature'

class pandagg.node.query.specialized.**MoreLikeThis**(*fields*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'more_like_this'

class pandagg.node.query.specialized.**Percolate**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'percolate'

class pandagg.node.query.specialized.**RankFeature**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'rank_feature'

class pandagg.node.query.specialized.**Script**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'script'

class pandagg.node.query.specialized.**Wrapper**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'wrapper'

pandagg.node.query.specialized_compound module

class pandagg.node.query.specialized_compound.**PinnedQuery**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'pinned'

class pandagg.node.query.specialized_compound.**ScriptScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'script_score'

pandagg.node.query.term_level module

class pandagg.node.query.term_level.**Exists**(*field*, *_name=None*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'exists'

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.query.term_level.**Fuzzy**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'fuzzy'

class pandagg.node.query.term_level.**Ids**(*values*, *_name=None*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'ids'

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

to_dict(*with_name=True*)

class pandagg.node.query.term_level.**Prefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'prefix'

class pandagg.node.query.term_level.**Range**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'range'

class pandagg.node.query.term_level.**Regexp**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'regexp'

class pandagg.node.query.term_level.**Term**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'term'

class pandagg.node.query.term_level.**Terms**(***body*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

KEY = 'terms'

class pandagg.node.query.term_level.**TermsSet**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'terms_set'

class pandagg.node.query.term_level.**Type**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'type'

class pandagg.node.query.term_level.**Wildcard**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'wildcard'

Module contents

pandagg.node.response package

Submodules

pandagg.node.response.bucket module

class pandagg.node.response.bucket.**Bucket**(*value*, *key=None*, *level=None*)
Bases: *pandagg.node.response.bucket.BucketNode*

attr_name
Determine under which attribute name the bucket will be available in response tree. Dots are replaced by _ characters so that they don't prevent from accessing as attribute.

Resulting attribute unfit for python attribute name syntax is still possible and will be accessible through item access (dict like), see more in 'utils.Obj' for more details.

line_repr(***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.node.response.bucket.**BucketNode**
Bases: pandagg.node._node.Node

Module contents

4.1.2.2 Submodules

pandagg.node.types module

4.1.2.3 Module contents

4.1.3 pandagg.tree package

4.1.3.1 Submodules

pandagg.tree.aggs module

class pandagg.tree.aggs.**Aggs**(*aggs=None*, *mappings=None*, *nested_autocorrect=None*, *_groupby_ptr=None*)
Bases: pandagg.tree._tree.Tree

Combination of aggregation clauses. This class provides handful methods to build an aggregation (see *aggs()* and *groupby()*), and is used as well to parse aggregations response in easy to manipulate formats.

Mappings declaration is optional, but doing so validates aggregation validity and automatically handles missing nested clauses.

Accept following syntaxes:

from a dict: >>> Aggs({"per_user": {"terms": {"field": "user"}}})

from an other Aggs instance: >>> Aggs(Aggs({"per_user": {"terms": {"field": "user"}}}))

dict with AggClause instances as values: >>> from pandagg.aggs import Terms, Avg >>> Aggs({'per_user': Terms(field='user')})

Parameters **mappings** – `dict` or `pandagg.tree.mappings.Mappings` Mappings of requested indice(s). If provided, will

check aggregations validity. :param nested_autocorrect: `bool` In case of missing nested clauses in aggregation, if True, automatically add missing nested clauses, else raise error. Ignored if mappings are not provided. :param _groupby_ptr: `str` identifier of aggregation clause used as grouping element (used by *clone* method).

agg(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=False*, ***body*)
Insert provided agg clause in copy of initial Aggs.

Accept following syntaxes for type_or_agg argument:

string, with body provided in kwargs >>> Aggs().agg(name='some_agg', type_or_agg='terms', field='some_field')

python dict format: >>> Aggs().agg(name='some_agg', type_or_agg={'terms': {'field': 'some_field'})

AggClause instance: >>> from pandagg.aggs import Terms >>> Aggs().agg(name='some_agg', type_or_agg=Terms(field='some_field'))

Parameters

- **name** – inserted agg clause name
- **type_or_agg** – either agg type (str), or agg clause of dict format, or AggClause instance
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root
- **body** – aggregation clause body when providing string type_of_agg (remaining kwargs)

Returns copy of initial Aggs with provided agg inserted

aggs(*aggs*, *insert_below=None*, *at_root=False*)
Insert provided aggs in copy of initial Aggs.

Accept following syntaxes for provided aggs:

python dict format: >>> Aggs().aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}})

Aggs instance: >>> Aggs().aggs(Aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}}))

dict with Agg clauses values: >>> from pandagg.aggs import Terms, Avg >>> Aggs().aggs({'some_agg': Terms(field='some_field'), 'other_agg': Avg(field='age')})

Parameters

- **aggs** – aggregations to insert into existing aggregation
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root

Returns copy of initial Aggs with provided aggs inserted

applied_nested_path_at_node(*nid*)
Return nested path applied at a clause.

Parameters **nid** – clause identifier

Returns None if no nested is applied, else applied path (str)

apply_reverse_nested(*nid=None*)

groupby(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=None*, ***body*)
Insert provided aggregation clause in copy of initial Aggs.

Given the initial aggregation:

```
A──> B
└──> C
```

If *insert_below* = 'A':

```
A──> new──> B
       └──> C
```

```
>>> Aggs().groupby('per_user_id', 'terms', field='user_id')
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> Aggs().groupby('per_user_id', {'terms': {"field": "user_id"}})
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> from pandagg.aggs import Terms
>>> Aggs().groupby('per_user_id', Terms(field="user_id"))
{"per_user_id":{"terms":{"field":"user_id"}}}
```

Return type *pandagg.aggs.Aggs*

grouped_by(*agg_name=None*, *deepest=False*)
Define which aggregation will be used as grouping pointer.

Either provide an aggregation name, either specify 'deepest=True' to consider deepest linear eligible aggregation node as pointer.

node_class
alias of *pandagg.node.aggs.abstract.AggClause*

show(**args*, *line_max_length=80*, ***kwargs*)
Return compact representation of Aggs.

```
>>> Aggs({
>>>     "genres": {
>>>         "terms": {"field": "genres", "size": 3},
>>>         "aggs": {
>>>             "movie_decade": {
>>>                 "date_histogram": {"field": "year", "fixed_interval":
↪"3650d"}
>>>             }
>>>         },
>>>     }
>>> }).show()
<Aggregations>
genres                                          <terms, field="genres",␣
↪size=3>
└── movie_decade          <date_histogram, field="year", fixed_interval="3650d
↪">
```

All *args and **kwargs are propagated to *lighttree.Tree.show* method. :return: str

to_dict(*from_=None*, *depth=None*)
Serialize Aggs as dict.

Parameters **from** – identifier of aggregation clause, if provided, limits serialization to this clause and its

children (used for recursion, shouldn't be useful) :param depth: integer, if provided, limit the serialization to a given depth :return: dict

pandagg.tree.mappings module

class pandagg.tree.mappings.**Mappings**(*properties=None*, *dynamic=False*, ***kwargs*)

Bases: pandagg.tree._tree.Tree

list_nesteds_at_field(*field_path*)

List nested paths that apply at a given path.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.list_nesteds_at_field('id')
[]
>>> mappings.list_nesteds_at_field('comments')
['comments']
>>> mappings.list_nesteds_at_field('comments.comment_text')
['comments']
```

mapping_type_of_field(*field_path*)

Return field type of provided field path.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.mapping_type_of_field('id')
'keyword'
>>> mappings.mapping_type_of_field('comments')
'nested'
>>> mappings.mapping_type_of_field('comments.comment_text')
'text'
```

nested_at_field(*field_path*)

Return nested path applied on a given path. Return *None* is none applies.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.nested_at_field('id')
None
```

(continues on next page)

(continued from previous page)

```
>>> mappings.nested_at_field('comments')
'comments'
>>> mappings.nested_at_field('comments.comment_text')
'comments'
```

node_class
alias of *pandagg.node.mappings.abstract.Field*

to_dict (*from_=None*, *depth=None*)
Serialize Mappings as dict.

Parameters **from** – identifier of a field, if provided, limits serialization to this field and its

children (used for recursion, shouldn't be useful) :param depth: integer, if provided, limit the serialization to a given depth :return: dict

validate_agg_clause (*agg_clause*, *exc=True*)
Ensure that if aggregation clause relates to a field (*field* or *path*) this field exists in mappings, and that required aggregation type is allowed on this kind of field.

Parameters

- **agg_clause** – AggClause you want to validate on these mappings
- **exc** – boolean, if set to True raise exception if invalid

Return type boolean

validate_document (*d*)

pandagg.tree.query module

class pandagg.tree.query.**Query** (*q=None*, *mappings=None*, *nested_autocorrect=False*)
Bases: pandagg.tree._tree.Tree

applied_nested_path_at_node (*nid*)
Return nested path applied at a clause.

Parameters **nid** – clause identifier

Returns None if no nested is applied, else applied path (str)

bool (*must=None*, *should=None*, *must_not=None*, *filter=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

```
>>> Query().bool(must={"term": {"some_field": "yolo"}})
```

boosting (*positive=None*, *negative=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

constant_score (*filter=None*, *boost=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

dis_max (*queries*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

filter (*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

function_score (*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

has_child (*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

has_parent (*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

must (*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

Create copy of initial Query and insert provided clause under "bool" query "must".

```
>>> Query().must('term', some_field=1)
>>> Query().must({'term': {'some_field': 1}})
>>> from pandagg.query import Term
>>> Query().must(Term(some_field=1))
```

Keyword Arguments

- *insert_below* (`str`) – named query clause under which the inserted clauses should be placed.
- *compound_param* (`str`) – param under which inserted clause will be placed in compound query
- *on* (`str`) – named compound query clause on which the inserted compound clause should be merged.
- *mode* (`str` one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones
  - 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
  - 'replace_all' : existing compound clause is completely replaced by the new one

must_not(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

nested(*path*, *query=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

node_class

alias of *pandagg.node.query.abstract.QueryClause*

pinned_query(*organic*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

query(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *compound_param=None*, ***body*)

Insert provided clause in copy of initial Query.

```
>>> from pandagg.query import Query
>>> Query().query('term', some_field=23)
{'term': {'some_field': 23}}
```

```
>>> from pandagg.query import Term
>>> Query()\
>>> .query({'term': {'some_field': 23})\
>>> .query(Term(other_field=24))\
{'bool': {'must': [{'term': {'some_field': 23}}, {'term': {'other_field
↪': 24}}]}}
```

Keyword Arguments

- *insert_below* (`str`) – named query clause under which the inserted clauses should be placed.
- *compound_param* (`str`) – param under which inserted clause will be placed in compound query
- *on* (`str`) – named compound query clause on which the inserted compound clause should be merged.
- *mode* (`str` one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones

- 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
- 'replace_all' : existing compound clause is completely replaced by the new one

script_score(*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

should(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

show(**args*, *line_max_length=80*, ***kwargs*)

Return compact representation of Query.

```
>>> Query()         >>> .must({"exists": {"field": "some_field"}})         >>> .
↪must({"term": {"other_field": {"value": 5}}})         >>> .show()
<Query>
bool
└── must
    ├── exists                                                    field=some_
↪field
    └── term                                              field=other_field,␣
↪value=5
```

All *args and **kwargs are propagated to *lighttree.Tree.show* method. :return: str

to_dict(*from_=None*)

Serialize Query as dict.

pandagg.tree.response module

class pandagg.tree.response.**AggsResponseTree**(*aggs*, *raw_response=None*)

Bases: pandagg.tree._tree.Tree

Tree shaped representation of an ElasticSearch aggregations response.

bucket_properties(*bucket*, *properties=None*, *end_level=None*, *depth=None*)

Recursive method returning a given bucket's properties in the form of an ordered dictionnary. Travel from current bucket through all ancestors until reaching root.

Parameters

- **bucket** – instance of pandagg.buckets.buckets.Bucket
- **properties** – OrderedDict accumulator of 'level' -> 'key'
- **end_level** – optional parameter to specify until which level properties are fetched
- **depth** – optional parameter to specify a limit number of levels which are fetched

Returns OrderedDict of structure 'level' -> 'key'

get_bucket_filter(*nid*)

Build query filtering documents belonging to that bucket. Suppose the following configuration:

```
Base                        <- filter on base
  |── Nested_A                  no filter on A (nested still must be applied␣
↪for children)
  |     |── SubNested A1
  |     └── SubNested A2    <- filter on A2
  └── Nested_B              <- filter on B
```

node_class
alias of *pandagg.node.response.bucket.BucketNode*

parse(*raw_response*)
Build response tree from ElasticSearch aggregation response

Parameters **raw_response** – ElasticSearch aggregation response

Returns self

show(***kwargs*)
Return tree structure in hierarchy style.

Parameters

- **nid** – Node identifier from which tree traversal will start. If None tree root will be used
- **filter_** – filter function performed on nodes. Nodes excluded from filter function nor their children won't be displayed
- **reverse** – the `reverse` param for sorting `Node` objects in the same level
- **display_key** – boolean, if True display keyed nodes keys
- **reverse** – reverse parameter applied at sorting
- **line_type** – display type choice
- **limit** – int, truncate tree display to this number of lines
- **kwargs** – kwargs params passed to node `line_repr` method

:param line_max_length :rtype: unicode in python2, str in python3

4.1.3.2 Module contents

4.2 Submodules

4.2.1 pandagg.aggs module

class pandagg.aggs.**Aggs**(*aggs=None*, *mappings=None*, *nested_autocorrect=None*, *_groupby_ptr=None*)
Bases: `pandagg.tree._tree.Tree`

Combination of aggregation clauses. This class provides handful methods to build an aggregation (see *aggs()* and *groupby()*), and is used as well to parse aggregations response in easy to manipulate formats.

Mappings declaration is optional, but doing so validates aggregation validity and automatically handles missing nested clauses.

Accept following syntaxes:

from a dict: >>> Aggs({"per_user": {"terms": {"field": "user"}}})

from an other Aggs instance: >>> Aggs(Aggs({"per_user": {"terms": {"field": "user"}}}))

dict with AggClause instances as values: >>> from pandagg.aggs import Terms, Avg >>> Aggs({'per_user': Terms(field='user')})

Parameters **mappings** – `dict` or `pandagg.tree.mappings.Mappings` Mappings of requested indice(s). If provided, will

check aggregations validity. :param nested_autocorrect: `bool` In case of missing nested clauses in aggregation, if True, automatically add missing nested clauses, else raise error. Ignored if mappings are not provided. :param _groupby_ptr: `str` identifier of aggregation clause used as grouping element (used by *clone* method).

agg(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=False*, ***body*)
Insert provided agg clause in copy of initial Aggs.

Accept following syntaxes for type_or_agg argument:

string, with body provided in kwargs >>> Aggs().agg(name='some_agg', type_or_agg='terms', field='some_field')

python dict format: >>> Aggs().agg(name='some_agg', type_or_agg={'terms': {'field': 'some_field'})

AggClause instance: >>> from pandagg.aggs import Terms >>> Aggs().agg(name='some_agg', type_or_agg=Terms(field='some_field'))

Parameters

- **name** – inserted agg clause name
- **type_or_agg** – either agg type (str), or agg clause of dict format, or AggClause instance
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root
- **body** – aggregation clause body when providing string type_of_agg (remaining kwargs)

Returns copy of initial Aggs with provided agg inserted

aggs(*aggs*, *insert_below=None*, *at_root=False*)
Insert provided aggs in copy of initial Aggs.

Accept following syntaxes for provided aggs:

python dict format: >>> Aggs().aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}})

Aggs instance: >>> Aggs().aggs(Aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}}))

dict with Agg clauses values: >>> from pandagg.aggs import Terms, Avg >>> Aggs().aggs({'some_agg': Terms(field='some_field'), 'other_agg': Avg(field='age')})

Parameters

- **aggs** – aggregations to insert into existing aggregation
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root

Returns copy of initial Aggs with provided aggs inserted

applied_nested_path_at_node(*nid*)
Return nested path applied at a clause.

Parameters **nid** – clause identifier

Returns None if no nested is applied, else applied path (str)

apply_reverse_nested(*nid=None*)

groupby(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=None*, ***body*)
Insert provided aggregation clause in copy of initial Aggs.

Given the initial aggregation:

```
A──> B
└──> C
```

If *insert_below* = 'A':

```
A──> new──> B
       └──> C
```

```
>>> Aggs().groupby('per_user_id', 'terms', field='user_id')
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> Aggs().groupby('per_user_id', {'terms': {"field": "user_id"}})
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> from pandagg.aggs import Terms
>>> Aggs().groupby('per_user_id', Terms(field="user_id"))
{"per_user_id":{"terms":{"field":"user_id"}}}
```

Return type *pandagg.aggs.Aggs*

grouped_by(*agg_name=None*, *deepest=False*)
Define which aggregation will be used as grouping pointer.

Either provide an aggregation name, either specify 'deepest=True' to consider deepest linear eligible aggregation node as pointer.

node_class
alias of *pandagg.node.aggs.abstract.AggClause*

show(**args*, *line_max_length=80*, ***kwargs*)
Return compact representation of Aggs.

```
>>> Aggs({
>>>     "genres": {
>>>         "terms": {"field": "genres", "size": 3},
>>>         "aggs": {
>>>             "movie_decade": {
>>>                 "date_histogram": {"field": "year", "fixed_interval":
↪"3650d"}
>>>             }
>>>         },
>>>     }
>>> }).show()
<Aggregations>
genres                                          <terms, field="genres",␣
↪size=3>
└── movie_decade          <date_histogram, field="year", fixed_interval="3650d
↪">
```

All *args and **kwargs are propagated to *lighttree.Tree.show* method. :return: str

to_dict(*from_=None*, *depth=None*)
Serialize Aggs as dict.

Parameters **from** – identifier of aggregation clause, if provided, limits serialization to this clause and its

children (used for recursion, shouldn't be useful) :param depth: integer, if provided, limit the serialization to a given depth :return: dict

class pandagg.aggs.**Terms**(*field*, *missing=None*, *size=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

Terms aggregation.

BLACKLISTED_MAPPING_TYPES = []

KEY = 'terms'

VALUE_ATTRS = ['doc_count', 'doc_count_error_upper_bound', 'sum_other_doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Filters**(*filters*, *other_bucket=False*, *other_bucket_key=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

DEFAULT_OTHER_KEY = '_other_'

IMPLICIT_KEYED = True

KEY = 'filters'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Histogram**(*field*, *interval*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**DateHistogram**(*field*, *interval=None*, *calendar_interval=None*, *fixed_interval=None*, *meta=None*, *keyed=False*, *key_as_string=True*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'date_histogram'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['date']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Range**(*field*, *ranges*, *keyed=False*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MultipleBucketAgg*

KEY = 'range'

KEY_SEP = '-'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

from_key

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

to_key

class pandagg.aggs.**Global**(*meta=None*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'global'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Filter**(*filter=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'filter'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Missing**(*field*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

BLACKLISTED_MAPPING_TYPES = []

KEY = 'missing'

VALUE_ATTRS = ['doc_count']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Nested**(*path*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**ReverseNested**(*path=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.UniqueBucketAgg*

KEY = 'reverse_nested'

VALUE_ATTRS = ['doc_count']

WHITELISTED_MAPPING_TYPES = ['nested']

get_filter(*key*)
Provide filter to get documents belonging to document of given key.

class pandagg.aggs.**Avg**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'avg'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**Max**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'max'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**Sum**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'sum'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**Min**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'min'

VALUE_ATTRS = ['value']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**Cardinality**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'cardinality'

VALUE_ATTRS = ['value']

class pandagg.aggs.**Stats**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'stats'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**ExtendedStats**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'extended_stats'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum', 'sum_of_squares', 'variance', 'std

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**Percentiles**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

Percents body argument can be passed to specify which percentiles to fetch.

KEY = 'percentiles'

VALUE_ATTRS = ['values']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'ha

class pandagg.aggs.**PercentileRanks**(*field*, *values*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'percentile_ranks'

VALUE_ATTRS = ['values']

WHITELISTED_MAPPING_TYPES = ['long', 'integer', 'short', 'byte', 'double', 'float', 'h

class pandagg.aggs.**GeoBound**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'geo_bounds'

VALUE_ATTRS = ['bounds']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.aggs.**GeoCentroid**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

KEY = 'geo_centroid'

VALUE_ATTRS = ['location']

WHITELISTED_MAPPING_TYPES = ['geo_point']

class pandagg.aggs.**TopHits**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.MetricAgg*

KEY = 'top_hits'

VALUE_ATTRS = ['hits']

class pandagg.aggs.**ValueCount**(*field=None*, *script=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.FieldOrScriptMetricAgg*

BLACKLISTED_MAPPING_TYPES = []

KEY = 'value_count'

VALUE_ATTRS = ['value']

class pandagg.aggs.**AvgBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'avg_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**Derivative**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'derivative'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MaxBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'max_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MinBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'min_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**SumBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'sum_bucket'

VALUE_ATTRS = ['value']

class pandagg.aggs.**StatsBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'stats_bucket'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum']

class pandagg.aggs.**ExtendedStatsBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'extended_stats_bucket'

VALUE_ATTRS = ['count', 'min', 'max', 'avg', 'sum', 'sum_of_squares', 'variance', 'std

class pandagg.aggs.**PercentilesBucket**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'percentiles_bucket'

VALUE_ATTRS = ['values']

class pandagg.aggs.**MovingAvg**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'moving_avg'

VALUE_ATTRS = ['value']

class pandagg.aggs.**CumulativeSum**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'cumulative_sum'

VALUE_ATTRS = ['value']

class pandagg.aggs.**BucketScript**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_script'

VALUE_ATTRS = ['value']

class pandagg.aggs.**BucketSelector**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_selector'

VALUE_ATTRS = None

class pandagg.aggs.**BucketSort**(*script*, *buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.ScriptPipeline*

KEY = 'bucket_sort'

VALUE_ATTRS = None

class pandagg.aggs.**SerialDiff**(*buckets_path*, *gap_policy=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.Pipeline*

KEY = 'serial_diff'

VALUE_ATTRS = ['value']

class pandagg.aggs.**MatchAll**(*meta=None*, ***body*)
Bases: *pandagg.node.aggs.bucket.Filter*

class pandagg.aggs.**Composite**(*sources*, *size=None*, *after_key=None*, *meta=None*, ***body*)
Bases: *pandagg.node.aggs.abstract.BucketAggClause*

KEY = 'composite'

VALUE_ATTRS = ['doc_count']

extract_buckets(*response_value*)

get_filter(*key*)
In composite aggregation, key is a map, source name -> value

4.2.2 pandagg.connections module

class pandagg.connections.**Connections**
Bases: object

Class responsible for holding connections to different clusters. Used as a singleton in this module.

add_connection(*alias*, *conn*)
Add a connection object, it will be passed through as-is.

configure(***kwargs*)
Configure multiple connections at once, useful for passing in config dictionaries obtained from other sources, like Django's settings or a configuration management tool.

Example:

```
connections.configure(
    default={'hosts': 'localhost'},
    dev={'hosts': ['esdev1.example.com:9200'], 'sniff_on_start': True},
)
```

Connections will only be constructed lazily when requested through get_connection.

create_connection(*alias='default'*, ***kwargs*)
Construct an instance of elasticsearch.Elasticsearch and register it under given alias.

get_connection(*alias='default'*)
Retrieve a connection, construct it if necessary (only configuration was passed to us). If a non-string alias has been passed through we assume it's already a client instance and will just return it as-is.

Raises KeyError if no client (or its definition) is registered under the alias.

remove_connection(*alias*)
Remove connection from the registry. Raises KeyError if connection wasn't found.

4.2.3 pandagg.discovery module

class pandagg.discovery.**Index**(*name*, *settings*, *mappings*, *aliases*, *client=None*)
Bases: object

search(*nested_autocorrect=True*, *repr_auto_execute=True*)

class pandagg.discovery.**Indices**(***kwargs*)
Bases: lighttree.interactive.Obj

pandagg.discovery.**discover**(*using*, *index='*'*)

Parameters

- **using** – Elasticsearch client
- **index** – Comma-separated list or wildcard expression of index names used to limit the request.

4.2.4 pandagg.exceptions module

exception pandagg.exceptions.**AbsentMappingFieldError**
Bases: *pandagg.exceptions.MappingError*

Field is not present in mappings.

exception pandagg.exceptions.**InvalidAggregation**
Bases: Exception

Wrong aggregation definition

exception pandagg.exceptions.**InvalidOperationMappingFieldError**
Bases: *pandagg.exceptions.MappingError*

Invalid aggregation type on this mappings field.

exception pandagg.exceptions.**MappingError**
Bases: Exception

Basic Mappings Error

exception pandagg.exceptions.**VersionIncompatibilityError**
Bases: Exception

Pandagg is not compatible with this ElasticSearch version.

4.2.5 pandagg.mappings module

class pandagg.mappings.**Mappings**(*properties=None*, *dynamic=False*, ***kwargs*)
Bases: pandagg.tree._tree.Tree

list_nesteds_at_field(*field_path*)
List nested paths that apply at a given path.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.list_nesteds_at_field('id')
[]
>>> mappings.list_nesteds_at_field('comments')
['comments']
>>> mappings.list_nesteds_at_field('comments.comment_text')
['comments']
```

mapping_type_of_field(*field_path*)
Return field type of provided field path.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.mapping_type_of_field('id')
'keyword'
>>> mappings.mapping_type_of_field('comments')
'nested'
>>> mappings.mapping_type_of_field('comments.comment_text')
'text'
```

nested_at_field(*field_path*)
Return nested path applied on a given path. Return *None* is none applies.

```
>>> mappings = Mappings(dynamic=False, properties={
>>>     'id': {'type': 'keyword'},
>>>     'comments': {'type': 'nested', 'properties': {
>>>         'comment_text': {'type': 'text'},
>>>         'date': {'type': 'date'}
>>>     }}
>>> })
>>> mappings.nested_at_field('id')
None
>>> mappings.nested_at_field('comments')
'comments'
>>> mappings.nested_at_field('comments.comment_text')
'comments'
```

node_class
alias of *pandagg.node.mappings.abstract.Field*

to_dict(*from_=None*, *depth=None*)
Serialize Mappings as dict.

Parameters **from** – identifier of a field, if provided, limits serialization to this field and its

children (used for recursion, shouldn't be useful) :param depth: integer, if provided, limit the serialization to a given depth :return: dict

validate_agg_clause(*agg_clause*, *exc=True*)
Ensure that if aggregation clause relates to a field (*field* or *path*) this field exists in mappings, and that required aggregation type is allowed on this kind of field.

Parameters

- **agg_clause** – AggClause you want to validate on these mappings
- **exc** – boolean, if set to True raise exception if invalid

Return type boolean

validate_document(*d*)

class pandagg.mappings.**IMappings**(*mappings*, *client=None*, *index=None*, *depth=1*, *root_path=None*, *initial_tree=None*)
Bases: *pandagg.utils.DSLMixin*, lighttree.interactive.TreeBasedObj

Interactive wrapper upon mappings tree, allowing field navigation and quick access to single clause aggregations computation.

class pandagg.mappings.**Text**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'text'

class pandagg.mappings.**Keyword**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'keyword'

class pandagg.mappings.**ConstantKeyword**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'constant_keyword'

class pandagg.mappings.**WildCard**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'wildcard'

class pandagg.mappings.**Long**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'long'

class pandagg.mappings.**Integer**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'integer'

class pandagg.mappings.**Short**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'short'

class pandagg.mappings.**Byte**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'byte'

class pandagg.mappings.**Double**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'double'

class pandagg.mappings.**HalfFloat**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'half_float'

class pandagg.mappings.**ScaledFloat**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'scaled_float'

class pandagg.mappings.**Date**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date'

class pandagg.mappings.**DateNanos**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date_nanos'

class pandagg.mappings.**Boolean**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'boolean'

class pandagg.mappings.**Binary**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'binary'

class pandagg.mappings.**IntegerRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'integer_range'

class pandagg.mappings.**Float**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'float'

class pandagg.mappings.**FloatRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'float_range'

class pandagg.mappings.**LongRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'long_range'

class pandagg.mappings.**DoubleRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'double_range'

class pandagg.mappings.**DateRange**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

KEY = 'date_range'

class pandagg.mappings.**Object**(**body)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'object'

class pandagg.mappings.**Nested**(**body)
Bases: *pandagg.node.mappings.abstract.ComplexField*

KEY = 'nested'

class pandagg.mappings.**GeoPoint**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

For lat/lon points

KEY = 'geo_point'

class pandagg.mappings.**GeoShape**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

For complex shapes like polygons

KEY = 'geo_shape'

class pandagg.mappings.**IP**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

for IPv4 and IPv6 addresses

KEY = 'ip'

class pandagg.mappings.**Completion**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To provide auto-complete suggestions

KEY = 'completion'

class pandagg.mappings.**TokenCount**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To count the number of tokens in a string

KEY = 'token_count'

class pandagg.mappings.**MapperMurMur3**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To compute hashes of values at index-time and store them in the index

KEY = 'murmur3'

class pandagg.mappings.**MapperAnnotatedText**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

To index text containing special markup (typically used for identifying named entities)

KEY = 'annotated-text'

class pandagg.mappings.**Percolator**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Accepts queries from the query-dsl

KEY = 'percolator'

class pandagg.mappings.**Join**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Defines parent/child relation for documents within the same index

KEY = 'join'

class pandagg.mappings.**RankFeature**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record numeric feature to boost hits at query time.

KEY = 'rank_feature'

class pandagg.mappings.**RankFeatures**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record numeric features to boost hits at query time.

KEY = 'rank_features'

class pandagg.mappings.**DenseVector**(**body)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record dense vectors of float values.

KEY = 'dense_vector'

class pandagg.mappings.**SparseVector**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Record sparse vectors of float values.

KEY = 'sparse_vector'

class pandagg.mappings.**SearchAsYouType**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

A text-like field optimized for queries to implement as-you-type completion

KEY = 'search_as_you_type'

class pandagg.mappings.**Alias**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Defines an alias to an existing field.

KEY = 'alias'

class pandagg.mappings.**Flattened**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

Allows an entire JSON object to be indexed as a single field.

KEY = 'flattened'

class pandagg.mappings.**Shape**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

For arbitrary cartesian geometries.

KEY = 'shape'

class pandagg.mappings.**Histogram**(***body*)
Bases: *pandagg.node.mappings.abstract.RegularField*

For pre-aggregated numerical values for percentiles aggregations.

KEY = 'histogram'

class pandagg.mappings.**Index**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The index to which the document belongs.

KEY = '_index'

class pandagg.mappings.**Type**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The document's mappings type.

KEY = '_type'

class pandagg.mappings.**Id**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The document's ID.

KEY = '_id'

class pandagg.mappings.**FieldNames**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

All fields in the document which contain non-null values.

KEY = '_field_names'

class pandagg.mappings.**Source**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The original JSON representing the body of the document.

KEY = '_source'

class pandagg.mappings.**Size**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

The size of the _source field in bytes, provided by the mapper-size plugin.

KEY = '_size'

class pandagg.mappings.**Ignored**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

All fields in the document that have been ignored at index time because of ignore_malformed.

KEY = '_ignored'

class pandagg.mappings.**Routing**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

A custom routing value which routes a document to a particular shard.

KEY = '_routing'

class pandagg.mappings.**Meta**(*multiple=None*, *nullable=True*, ***body*)
Bases: *pandagg.node.mappings.abstract.Field*

Application specific metadata.

KEY = '_meta'

4.2.6 pandagg.query module

class pandagg.query.**Query**(*q=None*, *mappings=None*, *nested_autocorrect=False*)
Bases: pandagg.tree._tree.Tree

applied_nested_path_at_node(*nid*)
Return nested path applied at a clause.

Parameters **nid** – clause identifier

Returns None if no nested is applied, else applied path (str)

bool(*must=None*, *should=None*, *must_not=None*, *filter=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

```
>>> Query().bool(must={"term": {"some_field": "yolo"}})
```

boosting(*positive=None*, *negative=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

constant_score(*filter=None*, *boost=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

dis_max(*queries*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

filter(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

function_score(*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

has_child(*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

has_parent (*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

must (*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

Create copy of initial Query and insert provided clause under "bool" query "must".

```
>>> Query().must('term', some_field=1)
>>> Query().must({'term': {'some_field': 1}})
>>> from pandagg.query import Term
>>> Query().must(Term(some_field=1))
```

Keyword Arguments

- *insert_below* (str) – named query clause under which the inserted clauses should be placed.
- *compound_param* (str) – param under which inserted clause will be placed in compound query
- *on* (str) – named compound query clause on which the inserted compound clause should be merged.
- *mode* (str one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones
  - 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
  - 'replace_all' : existing compound clause is completely replaced by the new one

must_not (*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

nested (*path*, *query=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

node_class

alias of *pandagg.node.query.abstract.QueryClause*

pinned_query (*organic*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

query (*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *compound_param=None*, ***body*)

Insert provided clause in copy of initial Query.

```
>>> from pandagg.query import Query
>>> Query().query('term', some_field=23)
{'term': {'some_field': 23}}
```

```
>>> from pandagg.query import Term
>>> Query()\
>>> .query({'term': {'some_field': 23}})\
>>> .query(Term(other_field=24))\
{'bool': {'must': [{'term': {'some_field': 23}}, {'term': {'other_field
↪': 24}}]}}
```

Keyword Arguments

- *insert_below* (str) – named query clause under which the inserted clauses should be placed.
- *compound_param* (str) – param under which inserted clause will be placed in compound query
- *on* (str) – named compound query clause on which the inserted compound clause should be merged.

- *mode* (`str` one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones
  - 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
  - 'replace_all' : existing compound clause is completely replaced by the new one

script_score(*query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

should(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

show(**args*, *line_max_length=80*, ***kwargs*)

Return compact representation of Query.

```
>>> Query()         >>> .must({"exists": {"field": "some_field"}})         >>> .
↪must({"term": {"other_field": {"value": 5}}})         >>> .show()
<Query>
bool
└── must
    ├── exists                                                    field=some_
↪field
    └── term                                              field=other_field,␣
↪value=5
```

All *args and **kwargs are propagated to *lighttree.Tree.show* method. :return: str

to_dict(*from_=None*)

Serialize Query as dict.

class pandagg.query.**Exists**(*field*, *_name=None*)

Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'exists'

line_repr(*depth*, ***kwargs*)

Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.query.**Fuzzy**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)

Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'fuzzy'

class pandagg.query.**Ids**(*values*, *_name=None*)

Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'ids'

line_repr(*depth*, ***kwargs*)

Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

to_dict(*with_name=True*)

class pandagg.query.**Prefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)

Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'prefix'

class pandagg.query.**Range**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)

Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'range'

class pandagg.query.**Regexp**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'regexp'

class pandagg.query.**Term**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'term'

class pandagg.query.**Terms**(***body*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

KEY = 'terms'

class pandagg.query.**TermsSet**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'terms_set'

class pandagg.query.**Type**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'type'

class pandagg.query.**Wildcard**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'wildcard'

class pandagg.query.**Intervals**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'intervals'

class pandagg.query.**Match**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match'

class pandagg.query.**MatchBoolPrefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_bool_prefix'

class pandagg.query.**MatchPhrase**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_phrase'

class pandagg.query.**MatchPhrasePrefix**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'match_phrase_prefix'

class pandagg.query.**MultiMatch**(*fields*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'multi_match'

class pandagg.query.**Common**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'common'

class pandagg.query.**QueryString**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'query_string'

class pandagg.query.**SimpleQueryString**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'simple_string'

class pandagg.query.**Bool**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

```
>>> Bool(must=[], should=[], filter=[], must_not=[], boost=1.2)
```

KEY = 'bool'

class pandagg.query.**Boosting**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'boosting'

class pandagg.query.**ConstantScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'constant_score'

class pandagg.query.**FunctionScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'function_score'

class pandagg.query.**DisMax**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'dis_max'

class pandagg.query.**Nested**(*path*, ***kwargs*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'nested'

class pandagg.query.**HasParent**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_parent'

class pandagg.query.**HasChild**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'has_child'

class pandagg.query.**ParentId**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'parent_id'

class pandagg.query.**Shape**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'shape'

class pandagg.query.**GeoShape**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_shape'

class pandagg.query.**GeoPolygone**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_polygon'

class pandagg.query.**GeoDistance**(*distance*, ***body*)
Bases: *pandagg.node.query.abstract.AbstractSingleFieldQueryClause*

KEY = 'geo_distance'

line_repr(*depth*, ***kwargs*)
Control how node is displayed in tree representation. _ ├── one end │ └── two myEnd └── three

class pandagg.query.**GeoBoundingBox**(*field=None*, *_name=None*, *_expand__to_dot=True*, ***params*)
Bases: *pandagg.node.query.abstract.KeyFieldQueryClause*

KEY = 'geo_bounding_box'

class pandagg.query.**DistanceFeature**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'distance_feature'

class pandagg.query.**MoreLikeThis**(*fields*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.MultiFieldsQueryClause*

KEY = 'more_like_this'

class pandagg.query.**Percolate**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'percolate'

class pandagg.query.**RankFeature**(*field*, *_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.FlatFieldQueryClause*

KEY = 'rank_feature'

class pandagg.query.**Script**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'script'

class pandagg.query.**Wrapper**(*_name=None*, ***body*)
Bases: *pandagg.node.query.abstract.LeafQueryClause*

KEY = 'wrapper'

class pandagg.query.**ScriptScore**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'script_score'

class pandagg.query.**PinnedQuery**(*_name=None*, ***body*)
Bases: *pandagg.node.query.compound.CompoundClause*

KEY = 'pinned'

4.2.7 pandagg.response module

class pandagg.response.**Aggregations**(*data*, *search*)
Bases: object

get(*key*)

keys()

serialize(*output='tabular'*, ***kwargs*)

Parameters

- **output** – output format, one of "raw", "tree", "interactive_tree", "normalized", "tabular", "dataframe"
- **kwargs** – tabular serialization kwargs

Returns

to_dataframe(*grouped_by=None*, *normalize_children=True*, *with_single_bucket_groups=False*)

to_interactive_tree()

to_normalized()

to_tabular(*index_orient=True*, *grouped_by=None*, *expand_columns=True*, *expand_sep='|'*, *normalize=True*, *with_single_bucket_groups=False*)

Build tabular view of ES response grouping levels (rows) until 'grouped_by' aggregation node included is reached, and using children aggregations of grouping level as values for each of generated groups (columns).

Suppose an aggregation of this shape (A & B bucket aggregations):

```
A──> B──> C1
     ├──> C2
     └──> C3
```

With grouped_by='B', breakdown ElasticSearch response (tree structure), into a tabular structure of this shape:

		C1	C2	C3
A	B			
wood	blue	10	4	0
	red	7	5	2
steel	blue	1	9	0
	red	23	4	2

Parameters

- **index_orient** – if True, level-key samples are returned as tuples, else in a dictionnary
- **grouped_by** – name of the aggregation node used as last grouping level
- **normalize** – if True, normalize columns buckets

Returns index_names, values

to_tree()

class pandagg.response.**Hit**(*data*)
Bases: object

class pandagg.response.**Hits**(*hits*)
Bases: object

to_dataframe(*expand_source=True*, *source_only=True*)
Return hits as pandas dataframe. Requires pandas dependency. :param expand_source: if True, *_source* sub-fields are expanded as columns :param source_only: if True, doesn't include hit metadata (except id which is used as dataframe index)

class pandagg.response.**Response**(*data*, *search*)
Bases: object

success

4.2.8 pandagg.search module

class pandagg.search.**MultiSearch**(**kwargs*)
Bases: *pandagg.search.Request*

Combine multiple Search objects into a single request.

add(*search*)
Adds a new Search object to the request:

```
ms = MultiSearch(index='my-index')
ms = ms.add(Search(doc_type=Category).filter('term', category='python'))
ms = ms.add(Search(doc_type=Blog))
```

execute()
Execute the multi search request and return a list of search results.

to_dict()

class pandagg.search.**Request**(*using*, *index=None*)
Bases: object

index(**index*)
Set the index for the search. If called empty it will remove all information.

Example:

s = Search() s = s.index('twitter-2015.01.01', 'twitter-2015.01.02') s = s.index(['twitter-2015.01.01', 'twitter-2015.01.02'])

params(**kwargs*)
Specify query params to be used when executing the search. All the keyword arguments will override the current values. See https://elasticsearch-py.readthedocs.io/en/master/api.html#elasticsearch.Elasticsearch.search for all available parameters.

Example:

```
s = Search()
s = s.params(routing='user-1', preference='local')
```

using(*client*)
Associate the search request with an elasticsearch client. A fresh copy will be returned with current instance remaining unchanged.

Parameters **client** – an instance of elasticsearch.Elasticsearch to use or an alias to look up in elasticsearch_dsl.connections

class pandagg.search.**Search**(*using=None*, *index=None*, *mappings=None*, *nested_autocorrect=False*, *repr_auto_execute=False*)

Bases: *pandagg.utils.DSLMixin*, *pandagg.search.Request*

agg(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=False*, ***body*)

Insert provided agg clause in copy of initial Aggs.

Accept following syntaxes for type_or_agg argument:

string, with body provided in kwargs >>> Aggs().agg(name='some_agg', type_or_agg='terms', field='some_field')

python dict format: >>> Aggs().agg(name='some_agg', type_or_agg={'terms': {'field': 'some_field'})

AggClause instance: >>> from pandagg.aggs import Terms >>> Aggs().agg(name='some_agg', type_or_agg=Terms(field='some_field'))

Parameters

- **name** – inserted agg clause name
- **type_or_agg** – either agg type (str), or agg clause of dict format, or AggClause instance
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root
- **body** – aggregation clause body when providing string type_of_agg (remaining kwargs)

Returns copy of initial Aggs with provided agg inserted

aggs(*aggs*, *insert_below=None*, *at_root=False*)

Insert provided aggs in copy of initial Aggs.

Accept following syntaxes for provided aggs:

python dict format: >>> Aggs().aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}})

Aggs instance: >>> Aggs().aggs(Aggs({'some_agg': {'terms': {'field': 'some_field'}}, 'other_agg': {'avg': {'field': 'age'}}}))

dict with Agg clauses values: >>> from pandagg.aggs import Terms, Avg >>> Aggs().aggs({'some_agg': Terms(field='some_field'), 'other_agg': Avg(field='age')})

Parameters

- **aggs** – aggregations to insert into existing aggregation
- **insert_below** – name of aggregation below which provided aggs should be inserted
- **at_root** – if True, aggregation is inserted at root

Returns copy of initial Aggs with provided aggs inserted

bool(*must=None*, *should=None*, *must_not=None*, *filter=None*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

```
>>> Query().bool(must={"term": {"some_field": "yolo"}})
```

count()

Return the number of hits matching the query and filters. Note that only the actual number is returned.

delete() *executes the query by delegating to delete_by_query()*

exclude(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

Must not wrapped in filter context.

execute()
Execute the search and return an instance of `Response` wrapping all the data.

filter(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

classmethod from_dict(*d*)
Construct a new *Search* instance from a raw dict containing the search body. Useful when migrating from raw dictionaries.

Example:

```
s = Search.from_dict({
    "query": {
        "bool": {
            "must": [...]
        }
    },
    "aggs": {...}
})
s = s.filter('term', published=True)
```

groupby(*name*, *type_or_agg=None*, *insert_below=None*, *at_root=None*, ***body*)
Insert provided aggregation clause in copy of initial Aggs.

Given the initial aggregation:

```
A──> B
└──> C
```

If *insert_below* = 'A':

```
A──> new──> B
       └──> C
```

```
>>> Aggs().groupby('per_user_id', 'terms', field='user_id')
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> Aggs().groupby('per_user_id', {'terms': {"field": "user_id"}})
{"per_user_id":{"terms":{"field":"user_id"}}}
```

```
>>> from pandagg.aggs import Terms
>>> Aggs().groupby('per_user_id', Terms(field="user_id"))
{"per_user_id":{"terms":{"field":"user_id"}}}
```

Return type *pandagg.aggs.Aggs*

highlight(**fields*, ***kwargs*)
Request highlighting of some fields. All keyword arguments passed in will be used as parameters for all the fields in the `fields` parameter. Example:

```
Search().highlight('title', 'body', fragment_size=50)
```

will produce the equivalent of:

```
{
    "highlight": {
        "fields": {
```

(continues on next page)

(continued from previous page)

```
            "body": {"fragment_size": 50},
            "title": {"fragment_size": 50}
        }
    }
}
```

If you want to have different options for different fields you can call `highlight` twice:

```
Search().highlight('title', fragment_size=50).highlight('body', fragment_
↪size=100)
```

which will produce:

```
{
    "highlight": {
        "fields": {
            "body": {"fragment_size": 100},
            "title": {"fragment_size": 50}
        }
    }
}
```

highlight_options(***kwargs*)

Update the global highlighting options used for this request. For example:

```
s = Search()
s = s.highlight_options(order='score')
```

must(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

Create copy of initial Query and insert provided clause under "bool" query "must".

```
>>> Query().must('term', some_field=1)
>>> Query().must({'term': {'some_field': 1}})
>>> from pandagg.query import Term
>>> Query().must(Term(some_field=1))
```

Keyword Arguments

- *insert_below* (`str`) – named query clause under which the inserted clauses should be placed.
- *compound_param* (`str`) – param under which inserted clause will be placed in compound query
- *on* (`str`) – named compound query clause on which the inserted compound clause should be merged.
- *mode* (`str` one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones
  - 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
  - 'replace_all' : existing compound clause is completely replaced by the new one

must_not(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

post_filter(**args*, ***kwargs*)

query(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, ***body*)

Insert provided clause in copy of initial Query.

```
>>> from pandagg.query import Query
>>> Query().query('term', some_field=23)
{'term': {'some_field': 23}}
```

```
>>> from pandagg.query import Term
>>> Query()\
>>> .query({'term': {'some_field': 23})\
>>> .query(Term(other_field=24))\
{'bool': {'must': [{'term': {'some_field': 23}}, {'term': {'other_field
↪': 24}}]}}
```

Keyword Arguments

- *insert_below* (str) – named query clause under which the inserted clauses should be placed.
- *compound_param* (str) – param under which inserted clause will be placed in compound query
- *on* (str) – named compound query clause on which the inserted compound clause should be merged.
- *mode* (str one of 'add', 'replace', 'replace_all') – merging strategy when inserting clauses on a existing compound clause.
  - 'add' (default) : adds new clauses keeping initial ones
  - 'replace' : for each parameter (for instance in 'bool' case : 'filter', 'must', 'must_not', 'should'), replace existing clauses under this parameter, by new ones only if declared in inserted compound query
  - 'replace_all' : existing compound clause is completely replaced by the new one

scan()

Turn the search into a scan search and return a generator that will iterate over all the documents matching the query.

Use params method to specify any additional arguments you with to pass to the underlying scan helper from elasticsearch-py - https://elasticsearch-py.readthedocs.io/en/master/helpers.html#elasticsearch.helpers.scan

script_fields(***kwargs*)

Define script fields to be calculated on hits. See https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-script-fields.html for more details.

Example:

```
s = Search()
s = s.script_fields(times_two="doc['field'].value * 2")
s = s.script_fields(
    times_three={
        'script': {
            'inline': "doc['field'].value * params.n",
            'params': {'n': 3}
        }
    }
)
```

should(*type_or_query*, *insert_below=None*, *on=None*, *mode='add'*, *bool_body=None*, ***body*)

size(*size*)
Equivalent to:

```
s = Search().params(size=size)
```

sort(**keys*)
Add sorting information to the search request. If called without arguments it will remove all sort requirements. Otherwise it will replace them. Acceptable arguments are:

```
'some.field'
'-some.other.field'
{'different.field': {'any': 'dict'}}
```

so for example:

```
s = Search().sort(
    'category',
    '-title',
    {"price" : {"order" : "asc", "mode" : "avg"}}
)
```

will sort by `category`, `title` (in descending order) and `price` in ascending order using the `avg` mode.

The API returns a copy of the Search object and can thus be chained.

source(*fields=None*, ***kwargs*)
Selectively control how the _source field is returned.

Parameters **fields** – wildcard string, array of wildcards, or dictionary of includes and excludes

If `fields` is None, the entire document will be returned for each hit. If fields is a dictionary with keys of 'includes' and/or 'excludes' the fields will be either included or excluded appropriately.

Calling this multiple times with the same named parameter will override the previous values with the new ones.

Example:

```
s = Search()
s = s.source(includes=['obj1.*'], excludes=["*.description"])

s = Search()
s = s.source(includes=['obj1.*']).source(excludes=["*.description"])
```

suggest(*name*, *text*, ***kwargs*)
Add a suggestions request to the search.

Parameters

- **name** – name of the suggestion
- **text** – text to suggest on

All keyword arguments will be added to the suggestions body. For example:

```
s = Search()
s = s.suggest('suggestion-1', 'Elasticsearch', term={'field': 'body'})
```

to_dict(*count=False*, ***kwargs*)
Serialize the search into the dictionary that will be sent over as the request's body.

Parameters **count** – a flag to specify if we are interested in a body for count - no aggregations, no pagination bounds etc.

All additional keyword arguments will be included into the dictionary.

update_from_dict(*d*)
Apply options from a serialized body to the current instance. Modifies the object in-place. Used mostly by `from_dict`.

4.2.9 pandagg.utils module

class pandagg.utils.**DSLMixin**
Bases: `object`

Base class for all DSL objects - queries, filters, aggregations etc. Wraps a dictionary representing the object's json.

class pandagg.utils.**DslMeta**(*name*, *bases*, *attrs*)
Bases: `type`

Base Metaclass for DslBase subclasses that builds a registry of all classes for given DslBase subclass (== all the query types for the Query subclass of DslBase).

It then uses the information from that registry (as well as *name* and *deserializer* attributes from the base class) to construct any subclass based on it's name.

pandagg.utils.**equal_queries**(*d1*, *d2*)
Compares if two queries are equivalent (do not consider nested list orders).

pandagg.utils.**equal_search**(*s1*, *s2*)

pandagg.utils.**ordered**(*obj*)

4.3 Module contents

CHAPTER 5

Contributing to Pandagg

We want to make contributing to this project as easy and transparent as possible.

5.1 Our Development Process

We use github to host code, to track issues and feature requests, as well as accept pull requests.

5.2 Pull Requests

We actively welcome your pull requests.

1. Fork the repo and create your branch from `master`.
2. If you've added code that should be tested, add tests.
3. If you've changed APIs, update the documentation.
4. Ensure the test suite passes.
5. Make sure your code lints.

5.3 Any contributions you make will be under the MIT Software License

In short, when you submit code changes, your submissions are understood to be under the same MIT License that covers the project. Feel free to contact the maintainers if that's a concern.

5.4 Issues

We use GitHub issues to track public bugs. Please ensure your description is clear and has sufficient instructions to be able to reproduce the issue.

5.5 Report bugs using Github's issues

We use GitHub issues to track public bugs. Report a bug by opening a new issue; it's that easy!

5.6 Write bug reports with detail, background, and sample code

Great Bug Reports tend to have:

- A quick summary and/or background
- Steps to reproduce
  - Be specific!
  - Give sample code if you can.
- What you expected would happen
- What actually happens
- Notes (possibly including why you think this might be happening, or stuff you tried that didn't work)

5.7 License

By contributing, you agree that your contributions will be licensed under its MIT License.

5.8 References

This document was adapted from the open-source contribution guidelines of briandk's gist

pandagg is a Python package providing a simple interface to manipulate ElasticSearch queries and aggregations. It brings the following features:

- flexible aggregation and search queries declaration
- query validation based on provided mapping
- parsing of aggregation results in handy format: interactive bucket tree, normalized tree or tabular breakdown
- mapping interactive navigation

CHAPTER 6

Installing

pandagg can be installed with pip:

```
$ pip install pandagg
```

Alternatively, you can grab the latest source code from GitHub:

```
$ git clone git://github.com/alkemics/pandagg.git
$ python setup.py install
```

CHAPTER 7

Usage

The *User Guide* is the place to go to learn how to use the library.

An example based on publicly available IMDB data is documented in repository *examples/imdb* directory, with a jupyter notebook to showcase some of *pandagg* functionalities: here it is.

The *pandagg package* documentation provides API-level documentation.

CHAPTER 8

License

pandagg is made available under the Apache 2.0 License. For more details, see LICENSE.txt.

CHAPTER 9

Contributing

We happily welcome contributions, please see *Contributing to Pandagg* for details.

Python Module Index

p

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W